Държавен изпит

Компютърни науки

Записки по конспекта от 30.06.2025

ФМИ, Софийски университет „Св. Климент Охридски“

Как да използвате записките

Дефинициите въвеждат понятия и означения. Те не се доказват. Теоремите, лемите, твърденията и следствията формулират проверими резултати; след тях има отделен блок с доказателство. Аксиомите и схемите изрично посочват приетите изходни предпоставки.

- Синя рамка: дефиниция.
- Зелена рамка: теорема, лема, твърдение или следствие.
- Златна рамка със звезда: основна дефиниция, резултат или аксиома.
- Сива затворена рамка: доказателство, завършващо със символ □.

Цветът казва какъв *вид* обект е даден блок. Той не казва какво *изисква* конспектът — дефиниция, само формулировка, пълно доказателство, конструкция, псевдокод, реализация, задача или чертеж. Затова всяка глава започва със сива рамка „По конспекта: какво трябва да мога“, в която всяко изискване на официалната анотация е изписано с думи и посочва раздела, който го изпълнява. Там стоят и официалните бележки кое ще бъде избрано на изпита (въпроси 6, 7, 8 и 18), кое е предварителен материал, кое е допълнително и къде покритието още не е пълно. Подробните списъци „Изпитен фокус“ в края на главите остават за повторение.

Държавният изпит е в две части в два дни: практическа (решаване на задачи) и теоретична, оценявани поотделно. Пълните указания са на стр. 3 на конспекта от 30.06.2025.

Номерата в квадратни скоби в бележките за източниците сочат официалната литература на конспекта, възпроизведена в края на книгата.

Формулировките и номерацията следват конспекта от 30.06.2025. В тази редакция са отделени определенията от обяснителния текст и са допълнени доказателства към резултати, оставени без доказателство в предходната редакция. Допълнените аргументи са учебна разработка, а не дословен текст от конспекта. Използваните външни резултати са посочени в съответното доказателство.

Бележките за липсващи първични източници по въпроси 11 и 12 остават в началото на тези глави. Проверката на рамките и наличието на доказателства не замества академичната проверка по литературата.

Съдържание

I Основи на компютърните науки **21**

1 Множества. Декартово произведение. Релации. Функции. **22**

1.1 Множества и аксиоматичен подход . . . 22

1.1.1 Аксиоми за построяване на множества . . . 23

1.2 Математическа индукция . . . 24

1.3 Операции върху множества . . . 26

1.4 Наредени двойки и декартови произведения . . . 28

1.5 Релации . . . 29

1.5.1 Свойства на бинарните релации . . . 29

1.5.2 Еквивалентност и класове на еквивалентност . . . 30

1.5.3 Частични и пълни наредби . . . 30

1.5.4 Линейно разширение и топологично сортиране . . . 31

1.6 Функции . . . 32

1.7 Крайни и изброими множества. Принцип на Дирихле . . . 33

1.8 Примерни задачи . . . 34

1.9 Изпитен фокус . . . 34

2 Основни комбинаторни принципи и конфигурации. Рекурентни уравнения. **36**

2.1 Комбинаторни принципи . . . 36

2.2 Основни комбинаторни конфигурации . . . 39

2.2.1 Конфигурации с наредба и повторение . . . 39

2.2.2 Конфигурации с наредба без повторение . . . 40

2.2.3 Конфигурации без наредба и без повторение . . . 40

2.2.4 Конфигурации без наредба с повторение . . . 41

2.3 Биномни коефициенти и двукратно броене . . . 41

2.4 Рекурентни уравнения . . . 43

2.4.1 Хомогенни линейни рекурентни уравнения . . . 43

2.4.2 Нехомогенни линейни рекурентни уравнения . . . 45
2.5 Примерни задачи по анотацията . . . 45
2.5.1 Задача 1: броене на функции между крайни множества . . . 45
2.5.2 Задача 2: целочислени решения с ограничения . . . 47
2.5.3 Задача 3: решаване на зададено рекурентно уравнение . . . 48
2.6 Изпитен фокус . . . 48

3 Графи. Дървета. Обхождания на графи. 50
3.1 Релации на еквивалентност и класове на еквивалентност . . . 50
3.2 Графи и основни понятия . . . 52
3.2.1 Неориентирани графи . . . 52
3.2.2 Ориентирани графи и мултиграфи . . . 53
3.3 Пътища, цикли и свързаност . . . 54
3.3.1 Пътища и цикли . . . 54
3.3.2 Свързаност . . . 54
3.4 Дървета . . . 55
3.5 Обхождания на графи . . . 56
3.5.1 Стек и опашка . . . 57
3.5.2 Схема за обхождане . . . 57
3.5.3 Обхождане в широчина . . . 57
3.5.4 Обхождане в дълбочина . . . 58
3.6 Ойлерови пътища и цикли . . . 59
3.7 Примерни задачи по анотацията . . . 60
3.7.1 Задача 1: покриващо дърво в широчина и в дълбочина . . . 60
3.7.2 Задача 2: Ойлеров цикъл или доказателство, че такъв няма . . . 61
3.7.3 Задача 3: разбиване на ребрата на минимален брой пътища без общо ребро . . . 62
3.8 Изпитен фокус . . . 63

4 Характеризация на регулярните езици. Теорема на Майхил-Нероуд. 65
4.1 Основни понятия . . . 65
4.2 Индукция върху естествените числа . . . 67
4.3 Характеризация чрез крайни автомати . . . 68
4.3.1 От регулярни изрази към автомати . . . 68
4.4 Лема за изпомпване на регулярните езици . . . 70
4.5 Релацията на Майхил–Нероуд . . . 70
4.6 Автоматът на Майхил–Нероуд . . . 71

4.7 Теорема на Майхил–Нероуд . . . 72
4.8 Минимизация на даден ДКА . . . 73
4.9 Изпитен фокус . . . 74

5 Лема за разрастването за контекстносвободни езици. Незатвореност относно сечение и допълнение. 76
5.1 Контекстносвободни граматики и дървета на извода . . . 77
5.2 Лема за разрастването . . . 78
5.3 Приложение: език, който не е контекстносвободен . . . 80
5.4 Затвореност относно регулярните операции . . . 81
5.5 Незатвореност относно сечение . . . 82
5.6 Незатвореност относно допълнение . . . 83
5.7 Изпитен фокус . . . 83

6 Сортиране чрез сравнения във време O(n lg n). 85
6.1 Сортиране чрез сравнения . . . 85
6.2 Двоична пирамида . . . 86
6.2.1 Определение и представяне с масив . . . 86
6.2.2 Пирамидални инверсии . . . 87
6.3 Наивно построяване на макс-пирамида . . . 87
6.4 Бързо построяване на пирамида . . . 89
6.4.1 Операцията Heapify . . . 89
6.4.2 Линейно построяване на пирамида . . . 90
6.5 Heapsort . . . 91
6.6 Mergesort и схемата Разделяй и владей . . . 92
6.6.1 Схемата Разделяй и владей . . . 92
6.6.2 Сливане на два сортирани подмасива . . . 93
6.6.3 Алгоритъмът Mergesort . . . 94
6.7 Броене на инверсии чрез Mergesort . . . 95
6.8 Изпитен фокус . . . 97

7 Минимални покриващи дървета. 99
7.1 Задача за минимално покриващо дърво . . . 99
7.2 Съгласувани множества и безопасни ребра . . . 100
7.3 Алгоритъм на Прим . . . 102
7.3.1 Идея и поддържани данни . . . 102
7.3.2 Използвани операции . . . 103

7.3.3 Псевдокод . . . 103
7.3.4 Пример . . . 104
7.3.5 Коректност . . . 105
7.3.6 Сложност на алгоритъма на Прим . . . 105
7.4 Алгоритъм на Крускал . . . 106
7.4.1 Използвани операции . . . 106
7.4.2 Псевдокод . . . 107
7.4.3 Пример . . . 107
7.4.4 Коректност . . . 108
7.5 Union–Find структури от данни . . . 108
7.5.1 Реализация с гора от коренови дървета . . . 109
7.5.2 Union by rank . . . 109
7.5.3 Компресия на пътя . . . 110
7.6 Сложност на алгоритъма на Крускал . . . 111
7.7 Изпитен фокус . . . 111

8 Най-къси пътища в тегловни графи. 113
8.1 Постановка на задачата . . . 114
8.1.1 Варианти на задачата . . . 114
8.1.2 Отрицателни тегла и оптимална подструктура . . . 115
8.2 Релаксация на ребра . . . 115
8.3 Алгоритъм на Дийкстра . . . 117
8.3.1 Вариант на задачата и идея . . . 117
8.3.2 Използвани операции . . . 117
8.3.3 Псевдокод . . . 118
8.3.4 Пример . . . 119
8.3.5 Коректност . . . 120
8.3.6 Сложност . . . 121
8.4 Най-къси пътища в тегловен DAG . . . 121
8.4.1 Алгоритъм . . . 121
8.4.2 Коректност . . . 122
8.4.3 Сложност и предимства . . . 122
8.5 Алгоритъм на Белман–Форд . . . 123
8.5.1 Вариант на задачата и псевдокод . . . 123
8.5.2 Коректност при липса на достижими отрицателни цикли . . . 123
8.5.3 Откриване на отрицателен цикъл . . . 124

8.5.4 Сложност . . . 125
8.6 Алгоритъм на Флойд–Уоршал . . . 126
8.6.1 Вариант на задачата . . . 126
8.6.2 Динамично програмиране . . . 126
8.6.3 Псевдокод . . . 126
8.6.4 Коректност . . . 127
8.6.5 Сложност по време и памет . . . 127
8.7 Изпитен фокус . . . 128

II Ядро на компютърните науки 129

9 Компютърни архитектури. Формати на данните. Вътрешна структура на централен процесор. 130
9.1 Обща структура на компютрите и компютърни архитектури . . . 130
9.1.1 Архитектура на фон Нойман . . . 131
9.2 Формати на данните . . . 132
9.2.1 Цели двоични числа . . . 132
9.2.2 Двоично-десетични числа . . . 134
9.2.3 Двоични числа с плаваща запетая . . . 134
9.2.4 Знакови данни и кодови таблици . . . 136
9.3 Централен процесор . . . 137
9.3.1 Регистри . . . 137
9.3.2 Аритметико-логическо устройство . . . 139
9.3.3 Блок за управление . . . 139
9.3.4 Инструкции . . . 140
9.3.5 Връзка с паметта . . . 141
9.4 Концептуално изпълнение на инструкциите . . . 141
9.4.1 Извличане на инструкция . . . 142
9.4.2 Декодиране . . . 142
9.4.3 Изчисляване и извличане на операнди . . . 142
9.4.4 Изпълнение и записване на резултата . . . 142
9.4.5 Недиректна стъпка . . . 142
9.4.6 Прекъсвания . . . 143
9.4.7 Преходи . . . 143
9.5 Конвейерна обработка . . . 144
9.6 Изпитен фокус . . . 144

10 Структура и йерархия на паметта. Сегментна и странична преадресация. Прекъсвания. 146

10.1 Структура и йерархия на паметта . . . 146
10.1.1 Основни понятия . . . 146
10.1.2 Принцип на йерархията . . . 147
10.1.3 Кеш памет . . . 148
10.1.4 Основна и виртуална памет . . . 148
10.2 Странична преадресация . . . 149
10.2.1 Страници и физически блокове . . . 149
10.2.2 Каталог на страниците и описател . . . 149
10.2.3 Обработка на достъп до страница . . . 150
10.2.4 Стратегии за подмяна на страници . . . 150
10.3 Сегментна преадресация . . . 151
10.3.1 Сегменти и адреси . . . 151
10.3.2 Глобална и локална дескрипторна таблица . . . 152
10.3.3 Алгоритъм на сегментната преадресация . . . 152
10.4 Система за прекъсване . . . 153
10.4.1 Предназначение на прекъсванията . . . 153
10.4.2 Векторна таблица и обработка . . . 153
10.4.3 Видове прекъсвания . . . 154
10.4.4 Конкурентност и приоритети . . . 154
10.4.5 Контролер на прекъсванията . . . 155
10.5 Обобщена връзка между преадресация, памет и прекъсвания . . . 156
10.6 Изпитен фокус . . . 156

11 Файлова система. Функции, структура и реализация. 158

11.1 Файлове и файлова система . . . 159
11.2 Физическо представяне и ext2/ext3 . . . 159
11.3 Ускоряване и надеждност . . . 160
11.4 POSIX функции за файлове . . . 160
11.5 Типови задачи . . . 161
11.5.1 Извличане на интервали от двоичен файл . . . 161
11.5.2 Сортиране на файл от 8-битови числа . . . 161
11.5.3 Прилагане на двоична кръпка . . . 161
11.6 Пълна програма: извличане на интервали . . . 162
11.7 Изпитен фокус . . . 165

12 Управление на процеси и междупроцесни комуникации. 166
12.1 Процес и основни примитиви . . . 167
12.2 Права, групи и сесии . . . 167
12.3 Сигнали и програмни канали . . . 168
12.4 UNIX System V IPC . . . 168
12.4.1 Обща памет . . . 169
12.4.2 Семафори . . . 169
12.4.3 Опашки от съобщения . . . 169
12.5 Синхронизационни схеми . . . 169
12.5.1 Родител и дете . . . 169
12.5.2 Производител–потребител . . . 170
12.6 Пълна програма: родител, дете и програмен канал . . . 170
12.7 Изпитен фокус . . . 173

13 Компютърни мрежи и протоколи. OSI модел. Маршрутизация. IPv4, IPv6, TCP, DNS. 174
13.1 Компютърни мрежи и протоколи . . . 174
13.2 OSI модел . . . 175
13.2.1 Обща характеристика . . . 175
13.2.2 Нива на OSI модела . . . 175
13.2.3 Съпоставяне на OSI и TCP/IP . . . 177
13.3 Разпределена маршрутизация . . . 178
13.3.1 Маршрутизация с дистантен вектор . . . 178
13.3.2 Маршрутизация със следене на състоянието на линията . . . 179
13.4 IPv4 адресация . . . 179
13.4.1 Класова адресация . . . 180
13.4.2 Безкласова адресация . . . 180
13.5 IPv6 . . . 181
13.6 TCP и трикратно договаряне . . . 182
13.7 DNS и резолвинг на имена . . . 183
13.7.1 Процес на резолвинг . . . 183
13.7.2 IPv4 и IPv6 записи . . . 183
13.8 Изпитен фокус . . . 184

14 Процедурно програмиране – основни конструкции. 185
14.1 Процедурно програмиране . . . 185
14.1.1 Принципи на структурното програмиране . . . 186

14.2 Управление на изчислителния процес . . . 187
14.3 Условни оператори . . . 187
14.3.1 Оператор `if` . . . 187
14.3.2 Условна, или тернарна, операция . . . 188
14.3.3 Оператор за многозначен избор `switch` . . . 189
14.4 Оператори за цикъл . . . 190
14.4.1 Оператор `for` . . . 190
14.4.2 Оператор `while` . . . 191
14.4.3 Оператор `do while` . . . 192
14.5 Променливи и присвояване . . . 192
14.5.1 Дефиниране и инициализиране . . . 192
14.5.2 Оператор за присвояване . . . 193
14.5.3 Типове на променливите . . . 194
14.5.4 Локални и глобални променливи . . . 194
14.6 Функции и процедури . . . 195
14.6.1 Дефиниране на функция . . . 195
14.6.2 Декларация и извикване . . . 196
14.6.3 Връщане на резултат . . . 196
14.6.4 Стекова рамка при извикване . . . 197
14.6.5 Предаване на параметри . . . 197
14.7 Символни низове . . . 200
14.7.1 Основни операции със символни низове . . . 200
14.7.2 Функции от библиотеката `cstring` . . . 201
14.8 Изпитен фокус . . . 202

15 Процедурно програмиране – указатели, масиви и рекурсия. 203
15.1 Указатели . . . 203
15.1.1 Оператори за работа с указатели . . . 204
15.1.2 Указатели към указатели . . . 205
15.1.3 Указатели и константи . . . 205
15.2 Масиви . . . 206
15.2.1 Достъп до елементи . . . 206
15.2.2 Многомерни масиви . . . 207
15.3 Указателна аритметика . . . 207
15.4 Указатели и низови константи . . . 208
15.5 Сортиране и търсене в едномерен масив . . . 208

15.5.1 Selection sort . . . 208
15.5.2 Bubble sort . . . 208
15.5.3 Insertion sort . . . 209
15.5.4 Quick sort . . . 210
15.5.5 Merge sort . . . 210
15.5.6 Линейно търсене . . . 211
15.5.7 Двоично търсене . . . 211
15.6 Рекурсия . . . 212
15.6.1 Линейна рекурсия . . . 212
15.6.2 Разклонена рекурсия . . . 213
15.7 Символни низове . . . 213
15.8 Изпитен фокус . . . 215

16 ООП. Основни принципи. Класове и обекти. Наследяване и капсулация. 216
16.1 Обектно-ориентирано програмиране . . . 216
16.2 Класове и обекти . . . 217
16.2.1 Модификатори за достъп . . . 218
16.2.2 Конструктори . . . 219
16.2.3 Деструктори и управление на ресурси . . . 220
16.2.4 Методи и указателят `this` . . . 221
16.2.5 Предефиниране на операции . . . 222
16.2.6 Статични полета и методи . . . 222
16.3 Наследяване . . . 223
16.3.1 Конструиране на производен обект . . . 224
16.3.2 Подтипово отношение . . . 225
16.3.3 Влагане и наследяване . . . 226
16.3.4 Вложени класове . . . 226
16.4 Капсулация и скриване на информация . . . 227
16.4.1 Полиморфизъм при наследяване . . . 228
16.5 Изпитен фокус . . . 229

17 ООП. Подтипов и параметричен полиморфизъм. Множествено наследяване. 230
17.1 Обектно-ориентирано програмиране и полиморфизъм . . . 230
17.2 Подтипов полиморфизъм и виртуални функции . . . 231
17.2.1 Предефиниране на член-функция . . . 231
17.2.2 Статично и динамично свързване . . . 233

17.2.3 Виртуална таблица и управление на паметта . . . 234
17.3 Абстрактни методи и абстрактни класове . . . 234
17.3.1 Чисто виртуални функции . . . 234
17.4 Масиви от обекти и указатели към обекти . . . 235
17.5 Параметричен полиморфизъм . . . 236
17.5.1 Идея и шаблони . . . 236
17.5.2 Шаблони на класове . . . 237
17.6 Множествено наследяване . . . 238
17.6.1 Основна конструкция . . . 238
17.6.2 Диамантен проблем . . . 238
17.6.3 Виртуално наследяване . . . 239
17.7 Изпитен фокус . . . 240

18 Структури от данни. Стек, опашка, списък, дърво. 242
18.1 Структури от данни . . . 243
18.2 Трите категории реализации . . . 243
18.3 Стек . . . 244
18.3.1 Последователно представяне на стек . . . 245
18.3.2 Свързано представяне на стек . . . 246
18.3.3 Клас за стек (динамична реализация) . . . 247
18.4 Опашка . . . 249
18.4.1 Последователно представяне на опашка . . . 249
18.4.2 Свързано представяне на опашка . . . 251
18.4.3 Клас за опашка (свързана реализация) . . . 252
18.5 Списък . . . 254
18.5.1 Едносвързан списък . . . 254
18.5.2 Двусвързан списък . . . 256
18.5.3 Клас за списък (двусвързана реализация) . . . 257
18.6 Дърво . . . 259
18.6.1 Двойно дърво . . . 260
18.6.2 Операции върху двойно дърво . . . 260
18.6.3 Представяния на двойно дърво . . . 262
18.6.4 Клас за двойно кореново дърво . . . 262
18.7 Двойно подредено дърво . . . 264
18.7.1 Търсене . . . 265
18.7.2 Включване . . . 265

18.7.3 Изключване . . . 266
18.7.4 Обхождане на двойно подредено дърво . . . 267
18.7.5 Клас за двойно подредено дърво . . . 267
18.8 Сравнение на структурите . . . 270
18.8.1 Сложност на операциите . . . 270
18.9 Изпитен фокус . . . 270

19 Функционално програмиране. Обща характеристика. Модели на оценяване. Функции от по-висок ред. 272
19.1 Функционално програмиране . . . 273
19.1.1 Обща характеристика на функционалния стил . . . 273
19.1.2 Изрази и стойности в Scheme . . . 274
19.1.3 Дефиниране и използване на функции . . . 274
19.1.4 Условни форми и локални дефиниции . . . 276
19.2 Оценяване на изрази . . . 277
19.2.1 Общо правило за оценяване на комбинация . . . 277
19.2.2 Оценяване на приложение на функция . . . 277
19.3 Модели на оценяване . . . 278
19.3.1 Апликативно оценяване . . . 278
19.3.2 Нормално оценяване . . . 279
19.4 Функции от по-висок ред . . . 280
19.4.1 Определение . . . 280
19.4.2 Къринг . . . 281
19.4.3 Анонимни функции . . . 282
19.4.4 Функции, които връщат функции . . . 282
19.5 Изпитен фокус . . . 283

20 Функционално програмиране. Списъци. Потоци и отложено оценяване. 285
20.1 Функционално програмиране . . . 285
20.2 S-изрази и списъци в Scheme . . . 286
20.2.1 S-изрази и наредени двойки . . . 286
20.2.2 Определение и представяне на списък . . . 286
20.2.3 Основни операции със списъци . . . 288
20.2.4 Равенство и търсене . . . 289
20.3 Списъци в Haskell . . . 290
20.3.1 Конструиране и типове . . . 290
20.3.2 Отделяне на списъци . . . 291

20.4 Функционално обработване на списъци . . . 292
20.4.1 Функции от по-висок ред . . . 292
20.4.2 Трансформиране чрез `map` . . . 292
20.4.3 Филтриране чрез `filter` . . . 293
20.4.4 Акумулиране и свиване . . . 293
20.5 Отложено оценяване . . . 295
20.5.1 Обещания и отложени операции . . . 295
20.5.2 Потоци . . . 296
20.5.3 Потоци в Scheme . . . 296
20.5.4 Функции от по-висок ред за потоци . . . 297
20.6 Отложено оценяване и списъци в Haskell . . . 298
20.7 Изпитен фокус . . . 299

21 Синтаксис и семантика на предикатното смятане от първи ред. 301
21.1 Език и синтаксис . . . 302
21.1.1 Предикатен език от първи ред . . . 302
21.1.2 Термове . . . 302
21.1.3 Формули . . . 303
21.1.4 Област на действие; свободни и свързани променливи . . . 303
21.2 Структури, оценки и стойности на термове . . . 304
21.3 Семантика на формулите . . . 306
21.3.1 Вярност при оценка . . . 306
21.3.2 Зависимост само от свободните променливи . . . 306
21.3.3 Тъждествена вярност и предикатни тавтологии . . . 307
21.4 Субституции . . . 308
21.4.1 Субституция на термове . . . 308
21.4.2 Допустимост на заместването във формули . . . 308
21.5 Определимост и изпълнимост . . . 311
21.5.1 Определими свойства . . . 311
21.5.2 Изпълнимост на множества от формули . . . 311
21.6 Изпитен фокус . . . 312

22 Изводимост и компютърно генериране на доказателства. 313
22.1 Основни понятия . . . 313
22.2 Резолюция за дизюнкти . . . 315
22.2.1 Същинска резолвента . . . 315
22.2.2 Резолютивен извод . . . 316

22.3 Предикатна либерална резолюция . . . 317
22.3.1 Субституции и унификация . . . 317
22.3.2 Предикатна либерална резолвента . . . 318
22.3.3 Коректност и пълнота . . . 319
22.4 Компютърно генериране на доказателства . . . 320
22.5 Хорнови дизюнкти и Пролог . . . 321
22.6 Изпитен фокус . . . 323

23 Бази от данни. Релационен модел на данните. 324
23.1 Бази от данни . . . 324
23.1.1 Предназначение на базите от данни . . . 324
23.2 Релационен модел на данните . . . 325
23.2.1 Математическа основа . . . 325
23.2.2 Схема и екземпляр на релация . . . 326
23.3 Операции върху релационна база от данни . . . 327
23.3.1 DDL операции . . . 327
23.3.2 DML операции . . . 328
23.3.3 Заявки към релационната база от данни . . . 328
23.4 Релационна алгебра . . . 328
23.4.1 Обединение . . . 329
23.4.2 Разлика . . . 329
23.4.3 Сечение . . . 330
23.4.4 Проекция . . . 330
23.4.5 Селекция . . . 331
23.4.6 Декартово произведение . . . 331
23.4.7 Тета-съединение . . . 332
23.4.8 Естествено съединение . . . 332
23.4.9 Преименуване . . . 333
23.4.10Класификация и приоритет . . . 333
23.5 Примерни задачи . . . 334
23.5.1 Съставяне на SQL заявки . . . 334
23.5.2 Пример за DDL . . . 335
23.5.3 Тригери . . . 335
23.6 Изпитен фокус . . . 336

24 Бази от данни. Нормални форми. 338
24.1 Бази от данни и нормални форми . . . 338

24.1.1 Релационни бази от данни и проектиране на схеми . . . 338
24.1.2 Аномалии . . . 339
24.1.3 Ограничения и ключове . . . 340
24.1.4 Функционални зависимости . . . 340
24.1.5 Аксиоми на Армстронг . . . 341
24.1.6 Първа нормална форма . . . 343
24.1.7 Втора нормална форма . . . 343
24.1.8 Трета нормална форма . . . 343
24.1.9 Нормална форма на Бойс–Код . . . 344
24.1.10 Многозначни зависимости . . . 345
24.1.11 Аксиоми и правила за многозначни зависимости . . . 346
24.1.12 Четвърта нормална форма . . . 347
24.1.13 Съединение без загуба . . . 348
24.1.14 Запазване на функционалните зависимости . . . 348
24.1.15 Примерна задача за нормализация . . . 349
24.1.16 Методика за решаване на задачи . . . 350
24.2 Изпитен фокус . . . 350

25 Изкуствен интелект. Пространство на състоянията. 352
25.1 Изкуствен интелект и пространство на състоянията . . . 352
25.1.1 Основни понятия . . . 352
25.1.2 Видове задачи и търсене . . . 354
25.1.3 Информирани методи за търсене . . . 355
25.1.4 Генетични алгоритми . . . 357
25.1.5 Задачи за удовлетворяване на ограничения . . . 359
25.1.6 Избор на следващ ход при игра за двама играчи . . . 361
25.2 Изпитен фокус . . . 363

26 Съвременни софтуерни технологии. 365
26.1 Софтуерен продукт, софтуерен процес и съвременни софтуерни технологии 365
26.1.1 Управление на софтуерен проект и ресурси . . . 366
26.2 Основни фази на софтуерните процеси . . . 366
26.3 Модели на софтуерни процеси . . . 367
26.3.1 Модел на водопада . . . 367
26.3.2 Модел на бързата разработка . . . 368
26.3.3 Еволюционни модели . . . 368
26.3.4 Прототипен модел . . . 368

26.3.5 Спираловиден модел . . . 369
26.3.6 Сравнителна характеристика . . . 369
26.4 Гъвкави методи . . . 370
26.4.1 Extreme Programming . . . 370
26.4.2 SCRUM . . . 371
26.5 Изисквания към софтуерните системи . . . 371
26.5.1 Функционални изисквания . . . 372
26.5.2 Нефункционални изисквания . . . 372
26.6 Анализ и проектиране на софтуерните изисквания . . . 372
26.6.1 Проектиране на софтуера . . . 373
26.6.2 Обектно-ориентиран дизайн . . . 373
26.7 Езици за описание и UML . . . 374
26.8 Верификация и валидация на софтуера . . . 375
26.9 Управление на качеството . . . 375
26.10Изпитен фокус . . . 376

27 Архитектури на софтуерни системи. 378
27.1 Понятие за софтуерна архитектура . . . 378
27.1.1 Структури и изгледи . . . 379
27.1.2 Необходимост и влияние на архитектурата . . . 381
27.2 Изисквания към качеството на системата . . . 382
27.2.1 Надеждност, наличност и сигурност . . . 383
27.3 Архитектурни компоненти и конектори . . . 383
27.4 Архитектурни стилове . . . 384
27.4.1 Многослоен стил . . . 384
27.4.2 Модел–изглед–контролер . . . 385
27.4.3 Поточна архитектура . . . 385
27.4.4 Клиент-сървърна архитектура . . . 386
27.4.5 Споделени данни . . . 386
27.4.6 Неявно извикване и предаване на съобщения . . . 386
27.5 Архитектура, ориентирана към услуги . . . 387
27.6 Проектиране на софтуерната архитектура . . . 387
27.6.1 Процес на проектиране . . . 387
27.6.2 Избор на структури . . . 388
27.6.3 Архитектурен анализ . . . 388
27.7 Тактики за постигане на качествени атрибути . . . 389

27.7.1 Тактики за изправност и надеждност . . . 389
27.7.2 Тактики за производителност . . . 390
27.7.3 Тактики за изменяемост . . . 390
27.7.4 Тактики за сигурност . . . 390
27.7.5 Тактики за тестируемост и използваемост . . . 391
27.8 Документиране на софтуерната архитектура . . . 391
27.8.1 Предназначение . . . 391
27.8.2 Основни елементи на документацията . . . 392
27.9 Изпитен фокус . . . 392

III Математика и приложения 394

28 Уравнения на права в равнината. 395
28.1 Параметрични уравнения на права . . . 395
28.2 Общо уравнение на права . . . 396
28.3 Взаимно положение на две прави . . . 398
28.4 Декартово уравнение . . . 399
28.5 Нормално уравнение и разстояние от точка до права . . . 400
28.6 Полуравнини . . . 402
28.7 Изпитен фокус . . . 403

29 Уравнения на права и равнина в пространството. 405
29.1 Равнина в тримерното пространство . . . 405
29.1.1 Векторно-параметрично и координатно параметрично уравнение . 405
29.2 Общо уравнение на равнина . . . 406
29.2.1 Получаване от точка и два направляващи вектора . . . 406
29.2.2 Равнина през три точки . . . 408
29.2.3 Нормален вектор и условие за компланарност . . . 408
29.3 Взаимни положения на две равнини . . . 409
29.4 Нормално уравнение на равнина и разстояние . . . 411
29.5 Полупространства . . . 412
29.6 Уравнения на права в пространството . . . 414
29.6.1 Векторно-параметрично и координатно параметрично уравнение . 414
29.6.2 Права като пресечница на две равнини . . . 414
29.7 Изпитен фокус . . . 415

30 Симетрични оператори в крайномерни евклидови пространства. Теорема за диагонализация. 417

30.1 Предварителни понятия . . . 418

30.2 Симетрични оператори . . . 422

30.2.1 Матрица на симетричен оператор . . . 422

30.3 Собствени стойности и собствени вектори . . . 424

30.3.1 Реалност на характеристичните корени . . . 425

30.3.2 Ортогоналност на собствени вектори . . . 426

30.4 Инвариантност на ортогоналното допълнение . . . 427

30.5 Теорема за диагонализация . . . 428

30.5.1 Връзка със симетричните матрици . . . 430

30.6 Обобщение на свойствата . . . 431

30.7 Изпитен фокус . . . 431

31 Симетрична и алтернативна група. Теорема на Кейли. Теорема за хомоморфизмите. 433

31.1 Симетрична група и цикличен запис . . . 433

31.2 Спрягане в S_n . . . 435

31.3 Транспозиции, четност и алтернативна група . . . 436

31.4 Хомоморфизми, ядро и образ . . . 438

31.5 Теорема за хомоморфизмите . . . 440

31.6 Теорема на Кейли . . . 441

31.7 Изпитен фокус . . . 442

32 Теорема на Ферма. Теореми за средните стойности (Рол, Лагранж, Коши). Формула на Тейлър. 444

32.1 Локални екстремуми и теорема на Ферма . . . 445

32.2 Теореми за средните стойности . . . 446

32.2.1 Теорема на Рол . . . 446

32.2.2 Теорема на Лагранж . . . 447

32.2.3 Теорема на Коши . . . 448

32.3 Формула на Тейлър . . . 451

32.3.1 Полином на Тейлър . . . 451

32.3.2 Остатъчен член във формата на Лагранж . . . 451

32.4 Изпитен фокус . . . 455

33 Определен интеграл. Суми на Дарбу. Интегруемост. Теорема на Нютон-Лайбниц. 457

33.1 Определен интеграл и суми на Дарбу . . . 457
33.1.1 Разбиване на интервал . . . 458
33.1.2 Малки и големи суми на Дарбу . . . 458
33.1.3 Горен и долен интеграл на Дарбу . . . 460
33.2 Критерий за интегруемост . . . 460
33.3 Интегруемост на непрекъснатите функции . . . 461
33.4 Основни свойства на римановия интеграл . . . 463
33.5 Теорема за средната стойност за интеграли . . . 464
33.6 Теорема на Нютон–Лайбниц . . . 466
33.6.1 Изчисляване на определени интеграли . . . 467
33.7 Изпитен фокус . . . 468

34 Итерационни методи за решаване на нелинейни уравнения. 470
34.1 Неподвижни точки и свеждане на нелинейно уравнение до итерационен процес . . . 470
34.2 Липшицови и свиващи изображения . . . 472
34.3 Ред на сходимост . . . 475
34.4 Методи на хордите, секущите и Нютон . . . 475
34.4.1 Метод на хордите . . . 475
34.4.2 Метод на секущите . . . 478
34.4.3 Метод на Нютон . . . 479
34.5 Сравнение на разгледаните итерационни методи . . . 480
34.6 Изпитен фокус . . . 480

35 Случайни величини с дискретни разпределения – биномно, геометрично, Поасоново. 483
35.1 Дискретни случайни величини и разпределения . . . 483
35.2 Пораждаща функция на дискретна случайна величина . . . 485
35.3 Биномно разпределение . . . 486
35.3.1 Схема на Бернули . . . 486
35.3.2 Пораждаща функция, математическо очакване и дисперсия . . . 488
35.4 Геометрично разпределение . . . 489
35.4.1 Дефиниция и интерпретация . . . 489
35.4.2 Пораждаща функция, математическо очакване и дисперсия . . . 490
35.5 Поасоново разпределение . . . 491
35.5.1 Дефиниция и задачи, в които възниква . . . 491
35.5.2 Връзка с биномното разпределение . . . 492

35.5.3 Пораждаща функция, математическо очакване и дисперсия 493
35.6 Сравнение на разглежданите разпределения 494
35.7 Изпитен фокус 494

Литература по конспекта **496**

Част I

Основи на компютърните науки

Въпрос 1

Множества. Декартово произведение. Релации. Функции.

По конспекта: какво трябва да мога

Дефиниции: аксиома за обема, схема за отделянето, аксиома за степенното множество, индуктивно генерирани множества — §1.1, стр. 22.
Дефиниции: операции върху множества и свойствата им — §1.3, стр. 26.
Дефиниции: наредена двойка, наредена n-орка, декартово и обобщено декартово произведение — §1.4, стр. 28.
Дефиниции: n-местна релация; свойства на бинарните релации; релация на еквивалентност и класове; частична и пълна наредба; диаграми на Хасе; минимален и максимален елемент — §1.5, стр. 29.
Дефиниции: частични и тотални функции; инекции, биекции, сюрекции — §1.6, стр. 32.
Дефиниции: крайно множество, кардиналност, изброимо безкрайно множество — §1.7, стр. 33.
Метод: математическа индукция — §1.2, стр. 24.
Метод: топологично сортиране като влагане на частична наредба в пълна — §1.5, стр. 29.
Формулировка: принцип на Дирихле — §1.7, стр. 33.
Типове задачи: петте, назовани от анотацията — равенство на множества; доказване или опровергаване на свойство на бинарна релация; класове на еквивалентност; доказателство по индукция; доказателство чрез принципа на Дирихле — §1.8, стр. 34.
Литература: [12], [33], [40].

1.1 Множества и аксиоматичен подход

Понятието *множество* се приема за първично: използва се без дефиниция. Елементите на множество A се означават с $x \in A$, а твърдението, че x не е негов елемент, с $x \notin A$.

Дефиниция 1.1 (Подмножество). Множеството A е подмножество на B, означено с $A \subseteq B$, когато всеки елемент на A е елемент и на B:

$$A \subseteq B \quad \Longleftrightarrow \quad (\forall x)\,(x \in A \Rightarrow x \in B).$$

★ **Аксиома 1.2 (За обема).** Две множества A и B са равни, ако и само ако имат едни и същи елементи:

$$A = B \quad \Longleftrightarrow \quad (\forall x)\,(x \in A \Leftrightarrow x \in B).$$

Формалният запис е

$$(\forall X)(\forall Y)\,((\forall z)\,(z \in X \Leftrightarrow z \in Y) \Rightarrow X = Y)\,.$$

1.1.1 Аксиоми за построяване на множества

★ **Схема 1.3 (За отделянето).** Нека A е множество и $\varphi(x, u_1, \ldots, u_n)$ е свойство. Тогава съществува множество, състоящо се точно от онези елементи на A, които удовлетворяват това свойство:

$$B = \{x \in A \mid \varphi(x, u_1, \ldots, u_n)\}.$$

Формално:

$$(\forall u_1) \cdots (\forall u_n)(\forall A)(\exists B)(\forall x)\,\big(x \in B \Leftrightarrow \big(x \in A \wedge \varphi(x, u_1, \ldots, u_n)\big)\big)\,.$$

Схемата за отделянето позволява да се образуват подмножества на вече дадено множество чрез зададено условие. Например множеството на четните естествени числа, които са не по-големи от 20, може да се запише като

$$\{x \in \mathbb{N} \mid x \leq 20 \text{ и } x \text{ е четно}\}.$$

★ **Аксиома 1.4 (За степенното множество).** За всяко множество A съществува множество от всички негови подмножества, наречено *степенно множество* на A и означавано с $\mathcal{P}(A)$:

$$\mathcal{P}(A) = \{X \mid X \subseteq A\}.$$

Например

$$\mathcal{P}(\varnothing) = \{\varnothing\}, \qquad \mathcal{P}(\{a, b\}) = \{\varnothing, \{a\}, \{b\}, \{a, b\}\}.$$

За крайно множество A с $|A| = n$ е валидно

$$|\mathcal{P}(A)| = 2^{|A|} = 2^n.$$

★ **Аксиома 1.5 (За индуктивно генерираните множества).** Нека U е множество, $\varnothing \neq M_0 \subseteq U$, а F е множество от операции $f : U^{k_f} \to U$ с крайна положителна арност k_f. Тогава множеството M се строи по следния начин:

1. *База:* $M_0 \subseteq M$, т.е. в M са всички елементи на M_0.
2. *Индуктивно предположение:* нека M_i е вече построеното на i-тата стъпка множество. То е непразно, защото $\varnothing \neq M_0 \subseteq M_i$.
3. *Индуктивна стъпка:* в M_{i+1} включваме всички досегашни елементи и всички елементи, които се получават от тях чрез прилагане на операциите от F.
4. *Заключение:* в M няма други елементи освен базовите и получените чрез краен брой индуктивни стъпки.

При едноместни операции конструкцията се записва формално като

$$M_{i+1} = M_i \cup \{f(m) \mid f \in F,\ m \in M_i\}, \qquad M = \bigcup_{i=0}^{\infty} M_i.$$

При многоместна операция f се прилагат всички допустими наредени групи от елементи на M_i. Построеното множество се означава с

$$M := \langle M_0, F \rangle.$$

Заключението изразява минималността на построеното множество: елементите му са точно базовите елементи и елементите, достижими чрез краен брой приложения на допустимите операции.

† **Допълнение 1.6 (Бележка).** Обединението с M_i в индуктивната стъпка е съществено: то запазва вече получените елементи при преминаването към следващото ниво.

1.2 Математическа индукция

Математическата индукция е метод за доказване на твърдения за естествени числа. Нека $m \in \mathbb{N}$ и $\varphi(n)$ е твърдение, зависещо от n.

★ **Теорема 1.7 (Принцип на слабата математическа индукция).** Ако са изпълнени:

1. базата: $\varphi(m)$ е вярно;
2. индуктивната стъпка:

$$(\forall k \geq m)\,\big(\varphi(k) \Rightarrow \varphi(k+1)\big),$$

то

$$(\forall n \in \mathbb{N},\ n \geq m)\,\varphi(n).$$

Доказателство. Използваме принципа за най-малкия елемент на естествените числа. Ако има $n \geq m$, за което $\varphi(n)$ е невярно, нека n_0 е най-малкото такова число. Базата дава $n_0 > m$. От минималността $\varphi(n_0 - 1)$ е вярно, а индуктивната стъпка тогава дава $\varphi(n_0)$ — противоречие. Следователно няма контрапример. Принципът за най-малкия елемент и индукцията са еквивалентни основни свойства на $\mathbb{N}$; тук първият е приет за изходен. □

Прилагането на принципа има ясен строеж. Първо се доказва твърдението за началната стойност. След това се приема, че е вярно за произволно, но фиксирано k, и при това допускане се доказва за $k+1$. Така от верността за m следва верността за $m+1$, после за $m+2$ и т.н.

★ **Теорема 1.8 (Принцип на пълната математическа индукция).** Ако $\varphi(m)$ е вярно и за всяко $k > m$ от

$$\varphi(m) \wedge \varphi(m+1) \wedge \cdots \wedge \varphi(k-1)$$

следва $\varphi(k)$, то $\varphi(n)$ е вярно за всяко естествено число $n \geq m$.

Доказателство. Прилагаме слабата индукция към $\psi(k) \equiv \bigwedge_{j=m}^{k} \varphi(j)$. Базата $\psi(m)$ е дадена. Ако $\psi(k)$ е вярно, предпоставката на пълната индукция дава $\varphi(k+1)$, следователно и $\psi(k+1)$. Получаваме $\psi(k)$ за всяко $k \geq m$, откъдето следва $\varphi(k)$. □

Разликата е в индуктивното предположение: при слабата индукция се използва само $\varphi(k)$, а при пълната индукция може да се използва верността на твърдението за всички предишни стойности.

▷ **Пример 1.9.** Да се докаже, че за всяко $n \in \mathbb{N}$:

$$0 + 1 + \cdots + n = \frac{n(n+1)}{2}.$$

За $n = 0$ равенството е вярно. Нека е вярно за $n = k$. Тогава

$$0 + 1 + \cdots + k + (k+1) = \frac{k(k+1)}{2} + (k+1) = \frac{(k+1)(k+2)}{2}.$$

Следователно формулата е вярна и за $k+1$, а по математическа индукция — за всяко $n \in \mathbb{N}$.

1.3 Операции върху множества

Дефиниция 1.10 (Операции върху множества)**.** Нека A и B са множества. Основните операции са следните:

$$A \cup B = \{x \mid x \in A \vee x \in B\},$$

$$A \cap B = \{x \mid x \in A \wedge x \in B\},$$

$$A \setminus B = \{x \mid x \in A \wedge x \notin B\},$$

$$A \triangle B = (A \setminus B) \cup (B \setminus A).$$

Обединението съдържа елементите, които принадлежат поне на едно от множествата; сечението съдържа общите им елементи; разликата $A \setminus B$ съдържа елементите на A, които не са в B. Симетричната разлика съдържа елементите, които принадлежат точно на едно от двете множества.

Дефиниция 1.11 (Допълнение)**.** При фиксиран универсум U, съдържащ разглежданите множества, *допълнение* на A до U е

$$\overline{A} = U \setminus A = \{x \in U \mid x \notin A\}.$$

Дефиниция 1.12 (Обобщено обединение и сечение)**.** Нека $A_i \subseteq U$ за всяко $i \in I$, където U е фиксиран универсум. Използват се обобщенията

$$\bigcup_{i \in I} A_i = \{x \mid (\exists i \in I)\ x \in A_i\},$$

$$\bigcap_{i \in I} A_i = \{x \in U \mid (\forall i \in I)\ x \in A_i\}.$$

При $I = \varnothing$ обединението е $\varnothing$, а сечението е U.

Твърдение 1.13 (Основни свойства)**.** За произволни множества A, B и C са валидни:

$$\begin{aligned}
A \cup A &= A, & A \cap A &= A;\\
A \cup B &= B \cup A, & A \cap B &= B \cap A;\\
(A \cup B) \cup C &= A \cup (B \cup C), & (A \cap B) \cap C &= A \cap (B \cap C);\\
A \cup (B \cap C) &= (A \cup B) \cap (A \cup C), & A \cap (B \cup C) &= (A \cap B) \cup (A \cap C);\\
A \cup \varnothing &= A, & A \cap U &= A;\\
A \cup U &= U, & A \cap \varnothing &= \varnothing;\\
A \cup \overline{A} &= U, & A \cap \overline{A} &= \varnothing;\\
\overline{\overline{A}} &= A.
\end{aligned}$$

Освен това са в сила законите на Де Морган:

$$\overline{A \cup B} = \overline{A} \cap \overline{B}, \qquad \overline{A \cap B} = \overline{A} \cup \overline{B}.$$

Доказателство. Фиксираме произволен $x \in U$ и означаваме $p \equiv x \in A$, $q \equiv x \in B$, $r \equiv x \in C$. Принадлежността на обединение, сечение и допълнение се превежда съответно като $\vee$, $\wedge$ и $\neg$. Идемпотентността, комутативността, асоциативността и дистрибутивността следват от едноименните булеви тъждества (проверими за всички осем оценки на p, q, r). Принадлежността на $\varnothing$ е неистина, а на U — истина, което дава четирите равенства с тези множества. Законите за допълнението следват от $p \vee \neg p = 1$, $p \wedge \neg p = 0$ и $\neg\neg p = p$. Накрая $\neg(p \vee q) \equiv \neg p \wedge \neg q$ и $\neg(p \wedge q) \equiv \neg p \vee \neg q$ дават двата закона на Де Морган. Във всеки случай двете страни имат едни и същи елементи и аксиомата за обема дава равенството. □

▷ **Пример 1.14.** Нека

$$A = \{x \in \mathbb{N} \mid x > 1\}, \qquad B = \{x \in \mathbb{N} \mid x > 3\}.$$

Тогава $B \subseteq A$ и следователно

$$A \cup B = A, \qquad A \cap B = B,$$

$$A \setminus B = \{x \in \mathbb{N} \mid 1 < x \leq 3\}, \qquad B \setminus A = \varnothing,$$

$$A \triangle B = \{x \in \mathbb{N} \mid 1 < x \leq 3\}.$$

1.4 Наредени двойки и декартови произведения

★ **Дефиниция 1.15 (Наредена двойка).** Наредена двойка от елементите a и b се означава с (a, b). Тя се определя чрез характеристичното свойство

$$(a, b) = (c, d) \iff a = c \wedge b = d.$$

Редът е съществен: по принцип $(a, b) \neq (b, a)$. Една теоретико-множествена реализация на наредена двойка е конструкцията на Куратовски:

$$(a, b) = \big\{\{a\}, \{a, b\}\big\}.$$

★ **Дефиниция 1.16 (Наредена n-орка).** Полагаме $(a_1) = a_1$ като база на рекурсивния запис. За $n \geq 2$ наредената n-орка от $a_1, \ldots, a_n$ се означава с

$$(a_1, \ldots, a_n)$$

и може рекурсивно да се разбира като

$$(a_1, (a_2, \ldots, a_n)).$$

★ **Дефиниция 1.17 (Декартово произведение).** За множества A и B тяхното декартово произведение е

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}.$$

По принцип $A \times B \neq B \times A$, защото се променя редът на компонентите. Например

$$\{1, 2\} \times \{a\} = \{(1, a), (2, a)\},$$

докато

$$\{a\} \times \{1, 2\} = \{(a, 1), (a, 2)\}.$$

★ **Дефиниция 1.18 (Обобщено декартово произведение).** За множества $A_1, \ldots, A_n$:

$$\prod_{i=1}^{n} A_i = A_1 \times \cdots \times A_n = \{(a_1, \ldots, a_n) \mid a_i \in A_i,\ 1 \leq i \leq n\}.$$

1.5 Релации

★ **Дефиниция 1.19 (n-местна релация).** Нека $A_1, \ldots, A_n$ са множества. Всяко подмножество

$$R \subseteq A_1 \times A_2 \times \cdots \times A_n$$

се нарича n-местна, или n-арна, релация над домейните $A_1, \ldots, A_n$.

При $n = 2$ говорим за бинарна релация. Ако домейните са еднакви, например $R \subseteq A \times A$, релацията е хомогенна. Вместо $(a, b) \in R$ често се пише aRb.

Дефиниция 1.20 (Домейн и множество от стойности на релация)**.** За $R \subseteq A \times B$ се дефинират:

$$\mathrm{dom}(R) = \{a \in A \mid (\exists b \in B)\ (a, b) \in R\},$$

$$\mathrm{rng}(R) = \{b \in B \mid (\exists a \in A)\ (a, b) \in R\}.$$

Това са съответно домейнът и множеството от стойности на релацията.

1.5.1 Свойства на бинарните релации

Дефиниция 1.21 (Свойства на бинарна релация)**.** Нека $R \subseteq A \times A$. Казваме, че R е:

- *рефлексивна*, ако
$$(\forall x \in A)\ xRx;$$
- *антирефлексивна*, или *ирефлексивна*, ако
$$(\forall x \in A)\ \neg xRx;$$
- *симетрична*, ако
$$(\forall x, y \in A)\ (xRy \Rightarrow yRx);$$
- *антисиметрична*, ако
$$(\forall x, y \in A)\ (xRy \wedge yRx \Rightarrow x = y);$$
- *силно антисиметрична*, ако за всеки две различни $x, y \in A$ е изпълнено точно едно от xRy и yRx;
- *асиметрична*, ако
$$(\forall x, y \in A)\ (xRy \Rightarrow \neg yRx);$$
- *транзитивна*, ако
$$(\forall x, y, z \in A)\ (xRy \wedge yRz \Rightarrow xRz).$$

Антисиметричността не означава, че за различни елементи винаги съществува сравнение. Тя забранява едновременното наличие на xRy и yRx при $x \neq y$. Силната антисиметричност добавя изискването за сравнимост на всяка двойка различни елементи.

1.5.2 Еквивалентност и класове на еквивалентност

★ **Дефиниция 1.22 (Релация на еквивалентност).** Релация $R \subseteq A \times A$ е релация на еквивалентност, ако е рефлексивна, симетрична и транзитивна.

★ **Дефиниция 1.23 (Клас на еквивалентност).** Нека R е релация на еквивалентност над A и $a \in A$. Класът на еквивалентност на a е

$$[a]_R = \{b \in A \mid aRb\}.$$

Когато релацията е ясна от контекста, се пише просто $[a]$.

Теорема 1.24. Всяка релация на еквивалентност над множество A поражда разбиване на A на класове на еквивалентност.

Доказателство. По рефлексивност $a \in [a]_R$, затова класовете са непразни и покриват A. Ако $c \in [a]_R \cap [b]_R$, то aRc и bRc, откъдето по симетричност и транзитивност aRb. За всяко $x \in [a]_R$ имаме $bRaRx$, следователно $x \in [b]_R$. Обратното включване е аналогично. Значи два класа или съвпадат, или са непресичащи се, както изисква определението за разбиване. □

Дефиниция 1.25 (Разбиване на множество)**.** Фамилията $\{A_i \mid i \in I\}$ е *разбиване* на A, ако всички A_i са непразни, обединението им е A и различните части нямат общи елементи:

$$\bigcup_{i \in I} A_i = A,$$

$$(\forall i, j \in I)\,(A_i \cap A_j \neq \varnothing \Rightarrow A_i = A_j).$$

▷ **Пример 1.26.** В $\mathbb{Z}$ задаваме aRb, ако a и b дават еднакъв остатък при деление на 3. Това е релация на еквивалентност. Класовете са числата с остатък 0, 1 и 2 при деление на 3.

1.5.3 Частични и пълни наредби

★ **Дефиниция 1.27 (Частична наредба).** Релация $R \subseteq A \times A$ е нестрога частична наредба, ако е рефлексивна, антисиметрична и транзитивна.

Дефиниция 1.28 (Строга частична наредба)**.** Строга частична наредба върху A е антирефлексивна и транзитивна релация $R \subseteq A \times A$. Тези условия влекат асиметричност: от xRy и yRx по транзитивност би следвало xRx, противоречие. Допълнителната антисиметричност в изходната формулировка е излишна.

★ **Дефиниция 1.29 (Пълна наредба).** Частична наредба R над A е пълна, ако всеки два различни елемента са сравними:

$$(\forall a, b \in A)\ (a \neq b \Rightarrow (aRb \vee bRa))\,.$$

Релациите $\leq$ и $<$ над естествените числа са примери за пълни наредби в нестрогия и строгия вариант.

★ **Дефиниция 1.30 (Минимален и максимален елемент).** Нека R е частична наредба над A. Елементът $a \in A$ е минимален, ако

$$(\forall b \in A)\,(b \neq a \Rightarrow \neg bRa).$$

Той е максимален, ако

$$(\forall b \in A)\,(b \neq a \Rightarrow \neg aRb).$$

Теорема 1.31. Всяка частична наредба над непразно крайно множество притежава минимален и максимален елемент.

Доказателство. Избираме $a_0 \in A$. Ако той не е минимален, има различен a_1 с $a_1 R a_0$. Продължаваме, докато достигнем минимален елемент. Повторение $a_i = a_j$ с $i < j$ е невъзможно: транзитивността и антисиметричността биха дали $a_i = a_{i+1}$, въпреки строгия избор. Понеже A е крайно, процесът спира. Прилагаме същото разсъждение към обратната наредба R^{-1} и получаваме максимален елемент. □

★ **Дефиниция 1.32 (Диаграма на Хасе).** За крайно частично наредено множество $(A, \leq)$ диаграмата на Хасе има върхове елементите на A. Ребро от x нагоре към y се чертае точно когато $x < y$ и няма z с $x < z < y$. Тогава y покрива x. Рефлексивните примки и ребрата, следващи по транзитивност, не се рисуват. Това уточнява общото описание на граф на релация в източника.

При частична наредба тя позволява удобно да се видят несравнимите елементи, както и минималните и максималните елементи.

1.5.4 Линейно разширение и топологично сортиране

★ **Теорема 1.33 (Влагане в пълна наредба).** Нека A е непразно крайно множество и R е частична наредба над A. Съществува пълна наредба S над A, за която

$$R \subseteq S.$$

Доказателство. Индукция по $|A|$. За едноелементно множество твърдението е ясно. Избираме минимален елемент a; той съществува по предходната теорема. По индукция ограничението на R върху $A \setminus \{a\}$ има линейно разширение. Поставяме a пред всички елементи на това разширение. Така получаваме пълна наредба. Няма

отношение bRa за $b \neq a$ по минималността, а всички останали отношения на R се запазват по индукционната хипотеза. Следователно новата наредба съдържа R. □

Дефиниция 1.34 (Линейно разширение)**.** Релацията S се нарича линейно разширение на R. Построяването му е известно като топологично сортиране.

```
Вход: крайно множество A и частична наредба R над A
Изход: редица B, задаваща линейно разширение

i <- 1
докато A не е празно:
    избери минимален елемент a на A спрямо R
    B[i] <- a
    изтрий a от A и ограничи R върху оставащите елементи
    i <- i + 1
върни B
```

Идеята за коректност е следната. Във всяка непразна стъпка оставащото крайно частично наредено множество има минимален елемент, следователно алгоритъмът може да продължи. Ако елемент a предшества b по R, b не може да бъде избран преди a, защото тогава b не би бил минимален сред оставащите елементи. Следователно редицата запазва всички отношения от R и задава пълна наредба, съдържаща R.

1.6 Функции

★ **Дефиниция 1.35 (Частична функция).** Нека A и B са множества. Релация $f \subseteq A \times B$ е частична функция от A в B, ако за всеки $a \in A$ има най-много един $b \in B$, за който $(a, b) \in f$:

$$(\forall a \in A)(\forall b_1, b_2 \in B)\left((a, b_1) \in f \wedge (a, b_2) \in f \Rightarrow b_1 = b_2\right).$$

При частична функция не е задължително всеки елемент на A да има стойност.

★ **Дефиниция 1.36 (Тотална функция).** Частичната функция $f \subseteq A \times B$ е тотална функция от A в B, ако

$$\mathrm{dom}(f) = A.$$

Еквивалентно:

$$(\forall a \in A)(\exists! b \in B)\ (a, b) \in f.$$

Пише се $f : A \to B$, а вместо $(a, b) \in f$ се пише $f(a) = b$.

★ **Дефиниция 1.37 (Инекция, сюрекция и биекция).** Нека $f : A \to B$ е тотална функция.

- f е *инекция*, ако
$$(\forall a_1, a_2 \in A)\,(f(a_1) = f(a_2) \Rightarrow a_1 = a_2).$$
- f е *сюрекция*, ако
$$(\forall b \in B)(\exists a \in A)\; f(a) = b.$$
- f е *биекция*, ако е едновременно инекция и сюрекция.

Инекцията не допуска различни аргументи с една и съща стойност. Сюрекцията изисква всяка стойност от B да бъде достигната. Биекцията установява съответствие един към един между елементите на A и B.

1.7 Крайни и изброими множества. Принцип на Дирихле

★ **Дефиниция 1.38 (Крайно множество и кардиналност).** Множество A е крайно, ако $A = \varnothing$ или съществува $n \in \mathbb{N}$, за което има биекция
$$A \longrightarrow \{0, 1, \ldots, n-1\}.$$
Числото n се нарича кардиналност на A и се означава с $|A| = n$. За празното множество $|\varnothing| = 0$.

★ **Дефиниция 1.39 (Изброимо безкрайно множество).** Множество A е изброимо безкрайно, ако съществува биекция между A и $\mathbb{N}$.

★ **Теорема 1.40 (Принцип на Дирихле).** Нека X и Y са крайни множества и
$$|X| > |Y|.$$
Тогава за всяка тотална функция $f : X \to Y$ съществуват различни $a, b \in X$, за които
$$f(a) = f(b).$$
Следователно не съществува инекция от X в Y.

Доказателство. Ако никои два различни аргумента нямат един и същ образ, всяко влакно $f^{-1}(\{y\})$ има най-много един елемент. Тези влакна са непресичащи се и покриват X, понеже f е тотална. Следователно
$$|X| = \sum_{y \in Y} |f^{-1}(\{y\})| \leq |Y|,$$

което противоречи на хипотезата. По същия начин, ако всяка от n кутии съдържа най-много k обекта, общият брой е най-много kn, което доказва и обобщената форма. □

Обобщената форма гласи: ако $kn + 1$ обекта се разпределят в n кутии, то поне в една кутия има повече от k обекта.

▷ **Пример 1.41.** При разпределяне на 13 студенти в 12 месеца според месеца им на раждане поне двама студенти са родени в един и същ месец. Това е пряко приложение на принципа на Дирихле.

1.8 Примерни задачи

▷ **Пример 1.42.** Нека

$$A = \{1, 2, 3\}, \qquad B = \{2, 3, 4\}.$$

Тогава

$$A \cup B = \{1, 2, 3, 4\}, \qquad A \cap B = \{2, 3\},$$
$$A \setminus B = \{1\}, \qquad A \triangle B = \{1, 4\}.$$

▷ **Пример 1.43.** Нека $A = \{1, 2\}$ и $B = \{a, b\}$. Тогава

$$A \times B = \{(1, a), (1, b), (2, a), (2, b)\}.$$

Релацията

$$R = \{(1, a), (2, b)\} \subseteq A \times B$$

е тотална функция от A към B. Тя е биекция, защото различните елементи на A имат различни образи и всеки елемент на B е образ на елемент от A.

▷ **Пример 1.44.** Нека $A = \{1, 2, 3, 6\}$ и над A е зададена релацията делимост:

$$aRb \iff a \mid b.$$

Тя е частична наредба. Елементите 2 и 3 са несравними, защото нито $2 \mid 3$, нито $3 \mid 2$. Минимален елемент е 1, а максимален елемент е 6.

▷ **Пример 1.45.** Нека R над $\mathbb{Z}$ е зададена с

$$aRb \iff a - b \text{ е четно.}$$

Релацията е рефлексивна, симетрична и транзитивна, следователно е релация на еквивалентност. Тя разделя $\mathbb{Z}$ на два класа: клас на четните и клас на нечетните цели числа.

1.9 Изпитен фокус

- Да се формулират аксиомата за обема, схемата за отделянето, аксиомата за степенното множество и идеята за индуктивно генерирано множество.

- Да се обяснят слабата и пълната математическа индукция и да се проведе индукционно доказателство с база и индуктивна стъпка.
- Да се дефинират обединение, сечение, разлика, симетрична разлика, допълнение и степенно множество; да се използват законите на Де Морган.
- Да се дефинират наредена двойка, наредена n-орка, декартово и обобщено декартово произведение.
- Да се дефинира n-местна релация, както и домейн и множество от стойности на бинарна релация.
- Да се формулират рефлексивност, антирефлексивност, симетричност, антисиметричност, силна антисиметричност, асиметричност и транзитивност.
- Да се дефинират релация на еквивалентност, клас на еквивалентност и разбиване на множество.
- Да се разграничават частична и пълна наредба; да се определят минимални и максимални елементи и да се интерпретира диаграма на Хасе.
- Да се възпроизведе идеята на топологичното сортиране: многократно избиране и премахване на минимален елемент, докато се получи линейно разширение.
- Да се дефинират частична и тотална функция, инекция, сюрекция и биекция.
- Да се дефинират крайно множество, кардиналност и изброимо безкрайно множество.
- Да се формулира принципът на Дирихле и да се разпознава приложението му като доказателство, че две различни стойности имат еднакъв образ.

Анотацията назовава пет типа задачи. Всеки трябва да може да се реши, а не само да се разпознае:

1. **Равенство на множества, описани с изрази** — чрез таблица на принадлежността или чрез законите за операциите (§1.3).
2. **Да се докаже или опровергае, че дадена бинарна релация има дадено свойство** — рефлексивност, симетричност, антисиметричност, транзитивност; опровергаването изисква *конкретен* контрапример, не разсъждение (§1.5).
3. **Определяне на класовете на еквивалентност** на зададена релация (§1.5).
4. **Доказателство по индукция** на твърдение за естествените числа — с изрично посочени база и индукционна стъпка (§1.2).
5. **Доказателство чрез принципа на Дирихле** — построява се явно изображение от по-голямо в по-малко множество и се посочва кое е „гнездото“ (§1.7).

Въпрос 2

Основни комбинаторни принципи и конфигурации. Рекурентни уравнения.

По конспекта: какво трябва да мога

Формулировки: седемте принципа — Дирихле, биекция, събиране, изваждане, умножение, деление, включване и изключване — §2.1, стр. 36.
Доказателство: принцип за включване и изключване — §2.1, стр. 36.
Дефиниции и извеждане на формулите: четирите основни конфигурации — с или без наредба, с или без повтаряне — §2.2, стр. 39.
Формулировка и доказателства: биномни коефициенти, теорема на Нютон, комбинаторни тъждества чрез двукратно броене — §2.3, стр. 41.
Алгоритъм: решаване на линейни рекурентни уравнения с константни коефициенти, хомогенни и нехомогенни — §2.4, стр. 43.
Задача 1: броене на частични функции, тотални функции, инекции и сюрекции — §2.5.1, стр. 45.
Задача 2: целочислени решения на $x_1 + \cdots + x_n = M$ с ограничения — §2.5.2, стр. 47.
Задача 3: решаване на зададено рекурентно уравнение до край — §2.5.3, стр. 48.
Предварително (не се изпитва тук): свойствата на бинарните релации, включително доказването и опровергаването им, принадлежат на Въпрос 1.
Литература: [12], [33], [40].

2.1 Комбинаторни принципи

В тази глава всички разглеждани множества са крайни, освен ако изрично не е посочено друго. Основната задача на изброителната комбинаторика е да се намери броят на обектите в дадена крайна конфигурация, като вместо непосредствено изброяване се използват структурни свойства на множествата и функциите.

★ **Теорема 2.1 (Принцип на Дирихле).** Нека X и Y са крайни множества и $|X| > |Y|$. Тогава не съществува инекция

$$f : X \to Y.$$

Еквивалентно: за всяка тотална функция $f : X \to Y$ съществуват различни $a, b \in X$, за които $f(a) = f(b)$.

Доказателство. Ако f е инекция, всеки образ има най-много един прообраз. Непресичащите се влакна $f^{-1}(\{y\})$, $y \in Y$, покриват X, затова $|X| = \sum_{y \in Y} |f^{-1}(\{y\})| \leq |Y|$, противоречие. Това е същият принцип, доказан във Въпрос 1. □

Принципът изразява, че ако повече обекти се разпределят в по-малко на брой „кутии“, поне една кутия съдържа поне два обекта.

★ **Теорема 2.2 (Принцип на биекцията).** За крайни множества A и B е изпълнено

$$|A| = |B|$$

тогава и само тогава, когато съществува биекция $f : A \to B$.

Доказателство. Ако $|A| = |B| = n$, избираме номерации $a : \{1, \ldots, n\} \to A$ и $b : \{1, \ldots, n\} \to B$; тогава $b \circ a^{-1}$ е биекция. Обратно, биекция $f : A \to B$ пренася всяка номерация на A в номерация на B със същия брой елементи. При празни множества се използва празната биекция. □

Принципът на биекцията е особено полезен, когато разглежданата конфигурация се съпостави еднозначно с друга, чиято мощност вече е известна.

★ **Теорема 2.3 (Принцип на събирането).** Нека

$$A = S_1 \cup S_2 \cup \cdots \cup S_k,$$

където множествата $S_1, \ldots, S_k$ са две по две непресичащи се. Тогава

$$|A| = \sum_{i=1}^{k} |S_i|.$$

Доказателство. Номерираме последователно елементите на S_1, после на S_2 и така до S_k. Непресичането гарантира, че няма повторения, а обединението гарантира, че не пропускаме елемент на A. Получената номерация има точно $\sum_i |S_i|$ позиции. □

Условието за непресичане е съществено: всеки елемент на A трябва да бъде преброен точно веднъж.

★ **Теорема 2.4 (Принцип на изваждането).** Нека $A \subseteq U$, където U е универсумът на разглежданите обекти. Тогава

$$|A| = |U| - |U \setminus A|.$$

Доказателство. Множествата A и $U \setminus A$ са непресичащи се и обединението им е U. Принципът на събирането дава $|U| = |A| + |U \setminus A|$. Пренасяме последното събираемо от другата страна. □

★ **Теорема 2.5 (Принцип на умножението).** За крайни множества A и B е в сила

$$|A \times B| = |A| \cdot |B|.$$

По-общо,

$$|X_1 \times X_2 \times \cdots \times X_r| = \prod_{i=1}^{r} |X_i|.$$

Доказателство. Разбиваме $A \times B$ на непресичащите се множества $\{a\} \times B$ за $a \in A$. Всяко е в биекция с B чрез $(a, b) \mapsto b$. Принципът на събирането дава $|A \times B| = \sum_{a \in A} |B| = |A||B|$. Индукция по положителния брой r дава общата формула. Ако някой множител е празен, и произведението е празно. □

Този принцип се прилага, когато обектът се изгражда последователно чрез независим избор на компоненти.

★ **Теорема 2.6 (Принцип на делението).** Нека A е непразно крайно множество и $R \subseteq A^2$ е релация на еквивалентност върху него. Ако A има k класа на еквивалентност и всеки клас има по m елемента, то

$$|A| = km, \qquad \text{следователно} \qquad m = \frac{|A|}{k}.$$

Доказателство. Класовете на еквивалентност образуват разбиване. По принципа на събирането $|A|$ е сумата на размерите им, тоест k еднакви събираеми m. Следователно $|A| = km$. Понеже разглеждаме непразно A, имаме $k, m > 0$ и двете формули с деление са допустими. □

При използване за преброяване обикновено са известни $|A|$ и размерът m на всеки клас; тогава броят на класовете е $|A|/m$.

★ **Теорема 2.7 (Принцип на включването и изключването).** За крайни множества $A_1, \ldots, A_n$ е изпълнено

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{i=1}^{n} |A_i| - \sum_{1 \le i < j \le n} |A_i \cap A_j| + \sum_{1 \le i < j < k \le n} |A_i \cap A_j \cap A_k| - \cdots + (-1)^{n-1} |A_1 \cap \cdots \cap A_n|.$$

Доказателство. Доказателството е с индукция по n. При $n = 1$ равенството е очевидно. За индукционната стъпка записваме

$$\left|\bigcup_{i=1}^{n} A_i\right| = \left|\left(\bigcup_{i=1}^{n-1} A_i\right) \cup A_n\right|.$$

За две множества е в сила

$$|B \cup C| = |B| + |C| - |B \cap C|.$$

Следователно

$$\left|\bigcup_{i=1}^{n} A_i\right| = \left|\bigcup_{i=1}^{n-1} A_i\right| + |A_n| - \left|\left(\bigcup_{i=1}^{n-1} A_i\right) \cap A_n\right|.$$

От дистрибутивния закон

$$\left(\bigcup_{i=1}^{n-1} A_i\right) \cap A_n = \bigcup_{i=1}^{n-1} (A_i \cap A_n).$$

Прилагат се индукционната хипотеза към двете обединения. След събиране на членовете се получават всички пресичания на $A_1, \ldots, A_n$ със съответните редуващи се знаци. □

2.2 Основни комбинаторни конфигурации

Нека основното множество има n елемента. Разглеждаме избиране на m елемента, като са възможни четири случая според това дали редът е важен и дали са разрешени повторения.

2.2.1 Конфигурации с наредба и повторение

Дефиниция 2.8. Множеството от конфигурациите с наредба и повторение с дължина m над множество A, $|A| = n$, се означава с $K_{\text{нп}}(n, m)$. Неговите елементи са наредени m-торки

$$(a_1, \ldots, a_m), \qquad a_i \in A.$$

Теорема 2.9.

$$|K_{\text{нп}}(n, m)| = n^m.$$

Доказателство. За всяка от m-те позиции има независимо по n възможности. Принципът на умножението дава

$$\underbrace{n \cdot n \cdots n}_{m \text{ пъти}} = n^m.$$

Еквивалентно, конфигурациите са функциите от m-елементно множество към n-елементно множество. □

▷ **Пример 2.10.** Броят на думите с дължина 4, образувани от азбука с n символа, е n^4, ако един и същ символ може да се появява многократно.

2.2.2 Конфигурации с наредба без повторение

Дефиниция 2.11. Конфигурация с наредба без повторение е наредена m-торка от различни елементи на n-елементно множество. Множеството на тези конфигурации се означава с $K_{\text{н}}(n, m)$.

Теорема 2.12.

$$|K_{\text{н}}(n, m)| = n(n-1)\cdots(n-m+1).$$

При $m > n$ броят е 0.

Доказателство. За първата позиция има n възможности. След избора й остават $n-1$ възможности за втората позиция, после $n-2$ за третата и т.н. Принципът на умножението дава формулата. Ако $m > n$, инекция от m-елементно множество в n-елементно множество не съществува по принципа на Дирихле. □

При $m = n$ конфигурациите се наричат *пермутации* и броят им е

$$n! = n(n-1)\cdots 2\cdot 1.$$

2.2.3 Конфигурации без наредба и без повторение

Дефиниция 2.13. Конфигурация без наредба и без повторение е m-елементно подмножество на n-елементно множество. Броят им се означава с

$$\binom{n}{m}.$$

★ **Теорема 2.14.** За $0 \le m \le n$ е изпълнено

$$\binom{n}{m} = \frac{n(n-1)\cdots(n-m+1)}{m!} = \frac{n!}{m!(n-m)!}.$$

Ако $m > n$, то

$$\binom{n}{m} = 0.$$

Доказателство. Нека разгледаме множеството $K_{\text{н}}(n, m)$ на наредените конфигурации без повторение. Въвеждаме релация R чрез

$$X\,R\,Y \quad \Longleftrightarrow \quad X \text{ и } Y \text{ съдържат едни и същи елементи.}$$

Това е релация на еквивалентност. Всеки клас на еквивалентност съдържа всички наредби на едно и също m-елементно множество, следователно има $m!$ елемента.

Класовете са точно конфигурациите без наредба и без повторение. По принципа на делението

$$\binom{n}{m} = \frac{|K_{\text{н}}(n,m)|}{m!} = \frac{n(n-1)\cdots(n-m+1)}{m!}.$$

□

2.2.4 Конфигурации без наредба с повторение

Дефиниция 2.15. Конфигурация без наредба с повторение с големина m над n-елементно основно множество е мултимножество с обща кардиналност m. Множеството от тези конфигурации се означава с $K_a(n,m)$.

Теорема 2.16.

$$|K_a(n,m)| = \binom{n+m-1}{m} = \binom{n+m-1}{n-1}.$$

Доказателство. Нека x_i е броят на избраните обекти от i-тия вид. Тогава

$$x_1 + x_2 + \cdots + x_n = m, \qquad x_i \geq 0.$$

Представяме решението чрез m звездички и $n-1$ разделителя. Например броят звездички преди първия разделител е x_1, между първия и втория е x_2 и т.н. Така получаваме биекция между избора на мултимножество и избора на позициите на m звездички сред общо $m+n-1$ позиции. Следователно броят е

$$\binom{m+n-1}{m}.$$

□

2.3 Биномни коефициенти и двукратно броене

Твърдение 2.17. За цели $0 \leq m \leq n$ са в сила следните равенства; тези с избор на 1 или $n-1$ елемента се отнасят за $n \geq 1$:

$$\binom{n}{0} = \binom{n}{n} = 1, \qquad \binom{n}{1} = \binom{n}{n-1} = n,$$

$$\binom{n}{m} = \binom{n}{n-m},$$

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n.$$

Доказателство. Празното множество и цялото множество са единствените подмножества с размер съответно 0 и n. За $n \geq 1$ едноелементните подмножества са n, както и допълненията им. Съответствието $B \mapsto A \setminus B$ е биекция между подмножествата с размер m и $n-m$. Накрая всяко от n-те различни елемента може независимо да бъде включено или не, затова всички подмножества са 2^n; разбиването им по размер дава сумата. □

Равенството

$$\binom{n}{m} = \binom{n}{n-m}$$

следва от биекцията, която на всяко m-елементно подмножество B на A съпоставя допълнението $A \setminus B$, имащо $n-m$ елемента.

Равенството

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n$$

се получава, като всички подмножества на n-елементно множество се преброят по два начина. От една страна те са 2^n; от друга страна ги разбиваме според кардиналността им.

★ **Теорема 2.18 (Формула на Паскал).** За $n, m \geq 1$ е в сила

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}.$$

Доказателство. Нека A е n-елементно множество и фиксираме $a \in A$. Разбиваме m-елементните подмножества на A на две непресичащи се групи: тези, които не съдържат a, и тези, които съдържат a. В първата група има

$$\binom{n-1}{m}$$

подмножества. За всяко подмножество от втората група, след премахване на a, остава $(m-1)$-елементно подмножество на $A \setminus \{a\}$, следователно такива са

$$\binom{n-1}{m-1}.$$

Принципът на събирането дава търсеното равенство. □

★ **Теорема 2.19 (Биномна теорема на Нютон).** За $x, y \in \mathbb{R}$ и $n \in \mathbb{N}$ е изпълнено

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}.$$

Доказателство. Произведението $(x+y)^n$ съдържа n множителя от вида $(x+y)$. При разкриване на скобите за получаване на член $x^k y^{n-k}$ трябва да изберем точно k от n-

те множителя, от които да вземем x; от останалите се взема y. Броят на тези избори е $\binom{n}{k}$. Сумирането по всички възможни k дава формулата. □

▷ **Пример 2.20.** Чрез двукратно броене се получава тъждеството

$$\binom{2n}{n} = \sum_{k=0}^{n} \binom{n}{k}^2.$$

Лявата страна брои n-елементните подмножества на множество, разделено на две n-елементни части. Ако от първата част са избрани k елемента, то от втората трябва да са избрани $n-k$ елемента. Броят е

$$\binom{n}{k}\binom{n}{n-k} = \binom{n}{k}^2.$$

Сумираме по всички k.

2.4 Рекурентни уравнения

Дефиниция 2.21. Рекурентно отношение от ред r е зависимост, при която за $n \geq r$ членът a_n на редица се определя чрез предходните r члена:

$$a_n = f(a_{n-1}, \ldots, a_{n-r}, n).$$

Заедно с началните условия то задава редицата.

▷ **Пример 2.22.** Отношението

$$l(n) = \begin{cases} l(n-1) + n, & n \in \mathbb{N}^*, \\ 1, & n = 0 \end{cases}$$

задава числова редица. Решаването на рекурентно уравнение означава да се намери явна формула за n-тия член на същата редица.

2.4.1 Хомогенни линейни рекурентни уравнения

Разглеждаме линейно рекурентно уравнение от ред k с постоянни коефициенти:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k},$$

където $c_k \neq 0$, а $a_0, \ldots, a_{k-1}$ са дадени начални условия.

★ **Дефиниция 2.23 (Характеристично уравнение).** На горното рекурентно уравнение съответства характеристичното уравнение

$$x^k - c_1 x^{k-1} - c_2 x^{k-2} - \cdots - c_k = 0.$$

То се получава формално чрез замяна на a_n с x^n:

$$x^n = c_1 x^{n-1} + \cdots + c_k x^{n-k},$$

след което се дели на x^{n-k}.

Теорема 2.24. Ако характеристичното уравнение има k различни корена

$$\alpha_1, \ldots, \alpha_k,$$

то общото решение е

$$a_n = A_1 \alpha_1^n + \cdots + A_k \alpha_k^n,$$

където константите $A_1, \ldots, A_k$ се определят от началните условия.

Доказателство. Работим над $\mathbb{C}$ (за реална редица константите се избират така, че решението да е реално). Всяка редица α_i^n удовлетворява рекурентното уравнение чрез пряко заместване. Пространството на решенията е k-мерно, защото първите k стойности се избират произволно и определят еднозначно всички следващи. Матрицата на първите k стойности на тези редици е матрица на Вандермонд с детерминанта $\prod_{i<j}(\alpha_j - \alpha_i) \neq 0$. Следователно редиците образуват базис и константите се определят еднозначно от началните условия. □

Теорема 2.25. Нека различните корени на характеристичното уравнение са

$$\beta_1, \ldots, \beta_l,$$

като коренът β_i има кратност r_i и

$$r_1 + \cdots + r_l = k.$$

Тогава към корена β_i съответстват членовете

$$A_{i,1}\beta_i^n + A_{i,2} n \beta_i^n + \cdots + A_{i,r_i} n^{r_i - 1} \beta_i^n.$$

Общото решение е сумата на тези изрази за всички различни корени.

Доказателство. Нека E е операторът на изместване $(Ea)_n = a_{n+1}$ и $P(x) = \prod_i (x - \beta_i)^{r_i}$ е характеристичният полином. Уравнението е $P(E)a = 0$. Понеже $c_k \neq 0$, всички β_i са ненулеви. За полином Q имаме

$$(E - \beta)(\beta^n Q(n)) = \beta^{n+1}\big(Q(n+1) - Q(n)\big).$$

Разликата намалява степента, затова $(E - \beta_i)^{r_i}$ анулира $\beta_i^n Q(n)$ при $\deg Q < r_i$. Тези r_i редици са независими, защото ненулев полином не се анулира върху всички естествени числа. За различни i пространствата са независими: при предполагаема зависимост прилагаме $\prod_{j \neq i}(E - \beta_j)^{r_j}$. Той анулира всички други слагаеми, а върху $\beta_i^n Q(n)$ е обратим, защото всяко $E - \beta_j$ има ненулев диагонален коефициент $\beta_i - \beta_j$ в полиномиалния базис. Следователно и i-тото слагаемо е нула. Получените общо k независими решения образуват базис на k-мерното пространство на решенията. □

След намиране на общия вид се заместват дадените начални условия. Получава се система от k линейни уравнения относно неизвестните константи, която се решава.

2.4.2 Нехомогенни линейни рекурентни уравнения

Нехомогенно линейно рекурентно уравнение с постоянни коефициенти има вида

$$a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k} + p_1(n) b_1^n + \cdots + p_l(n) b_l^n,$$

където $p_1(n), \ldots, p_l(n)$ са полиноми, а $b_1, \ldots, b_l$ са константи.

Първо се разглежда хомогенната част

$$a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k}$$

и се намират корените на нейното характеристично уравнение. Нехомогенната част е представена като сума от изрази от вида

$$b^n P(n),$$

където $P(n)$ е полином.

За частно решение на член $b^n P_d(n)$ се търси израз

$$a_n^{(p)} = n^s b^n Q_d(n),$$

където Q_d е полином от същата степен d, а s е кратността на b като корен на характеристичния полином на хомогенната част. Ако b не е корен, тогава $s = 0$. Неизвестните коефициенти на Q_d се намират чрез заместване и сравняване на коефициентите. За сума от такива членове се събират съответните частни решения. Общото решение е

$$a_n = a_n^{(h)} + a_n^{(p)},$$

а свободните константи се определят от началните условия.

2.5 Примерни задачи по анотацията

Анотацията към този въпрос назовава три типа задачи. Методите за тях са изложени по-горе; тук всеки тип е решен докрай, защото „методът е обяснен“ и „задачата е решена“ са различни неща на изпита.

2.5.1 Задача 1: броене на функции между крайни множества

Нека $|A| = m$ и $|B| = n$. Търсим броя на частичните функции, на тоталните функции, на инекциите и на сюрекциите от A в B.

Твърдение 2.26 (Частични и тотални функции)**.** Броят на тоталните функции $f : A \to B$ е n^m, а броят на частичните функции от A в B е $(n+1)^m$.

Доказателство. Тотална функция се задава чрез независим избор на образ за всеки от m-те елемента на A; по принципа на умножението изборите са n^m. Това е точно броят на конфигурациите с наредба и повторение.
Частична функция се задава чрез избор за всеки елемент на A на една от $n+1$ възможности: някой от n-те елемента на B или „недефинирана“. Изборите отново са независими, откъдето $(n+1)^m$. Еквивалентно: разширяваме B с един нов елемент $\perp$ и броим тоталните функции $A \to B \cup \{\perp\}$. □

Твърдение 2.27 (Инекции)**.** Броят на инекциите $f : A \to B$ е

$$n(n-1)\cdots(n-m+1) = \frac{n!}{(n-m)!},$$

и е 0 при $m > n$.

Доказателство. Подреждаме елементите на A и избираме образите последователно. Първият има n възможности, вторият — $n-1$, тъй като образът на първия е зает, и така нататък. Това е точно броят на конфигурациите с наредба без повторение. При $m > n$ инекция не съществува по принципа на Дирихле. □

Твърдение 2.28 (Сюрекции)**.** Броят на сюрекциите $f : A \to B$ е

$$\sum_{k=0}^{n} (-1)^k \binom{n}{k} (n-k)^m.$$

Доказателство. За $b \in B$ нека A_b е множеството от тоталните функции $A \to B$, които *не* приемат стойността b. Сюрекциите са точно функциите извън $\bigcup_{b\in B} A_b$. За фиксирано k-елементно $K \subseteq B$ пресичането $\bigcap_{b\in K} A_b$ се състои от функциите $A \to B \setminus K$, а те са $(n-k)^m$ на брой. Подмножествата K с k елемента са $\binom{n}{k}$. Принципът за включване и изключване дава

$$\left|\bigcup_{b\in B} A_b\right| = \sum_{k=1}^{n} (-1)^{k-1} \binom{n}{k} (n-k)^m,$$

а изваждането от общия брой n^m дава сумата от твърдението, чийто член при $k = 0$ е именно n^m. □

▷ **Пример 2.29.** При $m = n$ сюрекциите съвпадат с биекциите. Формулата дава $\sum_{k=0}^{n} (-1)^k \binom{n}{k} (n-k)^n = n!$, а при $m < n$ тя дава 0, което се съгласува с невъзможността да се покрие по-голямо множество.

2.5.2 Задача 2: целочислени решения с ограничения

Твърдение 2.30 (Неотрицателни решения)**.** Броят на решенията на

$$x_1 + x_2 + \cdots + x_n = M$$

в неотрицателни цели числа е $\binom{M+n-1}{n-1}$.

Доказателство. Всяко решение се кодира еднозначно с редица от M еднакви звездички и $n-1$ прегради: звездичките между $(i-1)$-вата и i-тата преграда са стойността x_i. Различните решения дават различни редици и всяка редица от $M+n-1$ позиции с избрани $n-1$ позиции за прегради дава решение. По принципа на биекцията броят е броят на подмножествата с $n-1$ елемента, тоест $\binom{M+n-1}{n-1}$. Това е конфигурация без наредба с повторение. □

Твърдение 2.31 (Долни граници)**.** Нека $c_1, \ldots, c_n$ са цели числа. Броят на решенията на $x_1 + \cdots + x_n = M$ с $x_i \geq c_i$ за всяко i е

$$\binom{M - \sum_{i=1}^{n} c_i + n - 1}{n-1},$$

когато $M \geq \sum_i c_i$, и 0 в противен случай.

Доказателство. Полагаме $y_i = x_i - c_i$. Изображението $(x_i) \mapsto (y_i)$ е биекция между решенията с $x_i \geq c_i$ и неотрицателните решения на

$$y_1 + \cdots + y_n = M - \sum_{i=1}^{n} c_i.$$

Прилагаме предходното твърдение. Ако дясната страна е отрицателна, решения няма. □

▷ **Пример 2.32 (Горни граници).** Ограничение отгоре се обработва с включване и изключване, а не с изместване. За

$$x_1 + x_2 + x_3 = 10, \qquad 0 \leq x_i \leq 5,$$

общият брой неотрицателни решения е $\binom{12}{2} = 66$. Нека B_i е множеството на решенията с $x_i \geq 6$. Изместването $x_i \mapsto x_i - 6$ свежда B_i до неотрицателните решения на сума $10 - 6 = 4$ с три неизвестни, тоест $|B_i| = \binom{4+2}{2} = \binom{6}{2} = 15$, а $|B_i \cap B_j| = 0$, защото $6 + 6 = 12 > 10$. Следователно търсеният брой е

$$66 - 3 \cdot 15 = 21.$$

▷ **Пример 2.33 (Смесени условия).** За $x_1 + x_2 + x_3 + x_4 = 20$ с $x_1 \geq 2$, $x_2 \geq 3$, $x_3, x_4 \geq 0$ изместваме първите две неизвестни и получаваме $\binom{20-5+3}{3} = \binom{18}{3} = 816$.

2.5.3 Задача 3: решаване на зададено рекурентно уравнение

▷ **Пример 2.34 (Хомогенно уравнение с начални условия).** Да се реши

$$a_n = 5a_{n-1} - 6a_{n-2}, \qquad a_0 = 1, \quad a_1 = 0.$$

Характеристичното уравнение е $x^2 - 5x + 6 = 0$ с различни корени 2 и 3, затова $a_n = A \cdot 2^n + B \cdot 3^n$. Началните условия дават

$$A + B = 1, \qquad 2A + 3B = 0,$$

откъдето $B = -2$ и $A = 3$. Следователно

$$a_n = 3 \cdot 2^n - 2 \cdot 3^n.$$

Проверка: $a_0 = 3 - 2 = 1$, $a_1 = 6 - 6 = 0$ и $a_2 = 12 - 18 = -6 = 5 \cdot 0 - 6 \cdot 1$.

▷ **Пример 2.35 (Нехомогенно уравнение с резонанс).** Да се реши

$$a_n = 3a_{n-1} - 2a_{n-2} + 2^n, \qquad a_0 = 0, \quad a_1 = 1.$$

Хомогенната част има характеристично уравнение $x^2 - 3x + 2 = 0$ с корени 1 и 2, тоест $a_n^{(h)} = A + B \cdot 2^n$. Основата 2 на нехомогенния член е корен с кратност $s = 1$, а полиномът пред него е от степен 0. Затова търсим $a_n^{(p)} = C\, n\, 2^n$. Заместваме и делим на 2^{n-2}:

$$4Cn = 6C(n-1) - 2C(n-2) + 4,$$

което се свежда до $0 = -2C + 4$, тоест $C = 2$ и $a_n^{(p)} = n \cdot 2^{n+1}$. Общото решение е

$$a_n = A + B \cdot 2^n + n \cdot 2^{n+1}.$$

Началните условия дават $A + B = 0$ и $A + 2B + 4 = 1$, откъдето $B = -3$ и $A = 3$:

$$a_n = 3 - 3 \cdot 2^n + n \cdot 2^{n+1}.$$

Проверка: $a_2 = 3 - 12 + 16 = 7 = 3 \cdot 1 - 2 \cdot 0 + 4$ и $a_3 = 3 - 24 + 48 = 27 = 3 \cdot 7 - 2 \cdot 1 + 8$.

Забележка 2.36. Типична грешка е спирането при общото решение. Анотацията иска *решено* уравнение, тоест константите да са определени от началните условия и отговорът да е проверен поне за първите няколко члена.

2.6 Изпитен фокус

- Формулировки на принципите на Дирихле, биекцията, събирането, изваждането, умножението, делението и включването и изключването.
- Доказателствената схема с индукция за принципа на включването и изключването.
- Различаване между четирите вида комбинаторни конфигурации и формулите

$$n^m, \qquad n(n-1)\cdots(n-m+1), \qquad \binom{n}{m}, \qquad \binom{n+m-1}{m}.$$

- Доказване на формулата за комбинации чрез релация на еквивалентност и принципа на делението.
- Комбинаторни доказателства на симетрията на биномните коефициенти, формулата на Паскал, равенството
$$\sum_{i=0}^{n} \binom{n}{i} = 2^n$$
и биномната теорема на Нютон.
- Алгоритъм за хомогенно линейно рекурентно уравнение: характеристично уравнение, различни и кратни корени, общ вид на решението и определяне на константите от началните условия.
- Разделяне на нехомогенно рекурентно уравнение на хомогенна и нехомогенна част; за член $b^n P_d(n)$ да се търси частно решение $n^s b^n Q_d(n)$, където s е кратността на b като характеристичен корен.
- Задача 1 от анотацията: броене на частични функции $(n+1)^m$, тотални функции n^m, инекции $n!/(n-m)!$ и сюрекции $\sum_k (-1)^k \binom{n}{k}(n-k)^m$ — с обосновката за всяка от четирите.
- Задача 2 от анотацията: неотрицателни целочислени решения чрез звездички и прегради; изместване $y_i = x_i - c_i$ при долни граници $x_i \geq c_i$; включване и изключване при горни граници.
- Задача 3 от анотацията: пълно решаване на зададено хомогенно и на зададено нехомогенно уравнение — характеристични корени, частно решение с резонансен множител n^s, определяне на константите от началните условия и числена проверка.
- Свойствата на бинарните релации (рефлексивност, симетричност, транзитивност и опровергаването им) принадлежат на Въпрос 1, а не на този въпрос.

Въпрос 3

Графи. Дървета. Обхождания на графи.

По конспекта: какво трябва да мога

Дефиниции: краен ориентиран и краен неориентиран (мулти)граф — §3.2, стр. 52.
Дефиниции: път и цикъл в ориентиран и неориентиран мултиграф; свързаност и свързани компоненти — §3.3, стр. 54.
Дефиниции: дърво, кореново дърво, покриващо дърво — §3.4, стр. 55.
Дефиниции: Ойлеров път и Ойлеров цикъл — §3.6, стр. 59.
Доказателства: всяко кореново дърво е дърво и $|V| = |E| + 1$ — §3.4, стр. 55.
Доказателство: теорема за съществуване на Ойлеров цикъл — §3.6, стр. 59.
Формулировка: критерий за съществуване на Ойлеров път — §3.6, стр. 59.
Алгоритми: обхождане в широчина и в дълбочина — §3.5, стр. 56.
Задача 1: покриващо дърво в широчина или в дълбочина — §3.7.1, стр. 60.
Задача 2: Ойлеров цикъл или път, или доказателство, че няма — §3.7.2, стр. 61.
Задача 3: разбиване на ребрата на минимален брой пътища без общо ребро — §3.7.3, стр. 62.
Предварително (не се изпитва тук): §3.1 повтаря релациите на еквивалентност от Въпрос 1; тук от тях се използва само връзката „достижимост = еквивалентност, класовете = компоненти“.
Литература: [12], [33], [40].

3.1 Релации на еквивалентност и класове на еквивалентност

Предварителен раздел, който конспектът не изисква по този въпрос. Анотацията към Въпрос 3 започва от дефинициите за граф; релациите на еквивалентност принадлежат на Въпрос 1. Разделът е тук само защото достижимостта в неориентиран граф е релация на еквивалентност, а класовете й са свързаните компоненти — това е използвано по-нататък. При повторение той може да се прескочи и да се започне направо от §3.2.

Дефиниция 3.1. Нека A е множество. Релация $\sim \subseteq A \times A$ се нарича *релация на еквивалентност*, ако за всички $x, y, z \in A$ са изпълнени:

1. *рефлексивност*: $x \sim x$;
2. *симетричност*: ако $x \sim y$, то $y \sim x$;
3. *транзитивност*: ако $x \sim y$ и $y \sim z$, то $x \sim z$.

Дефиниция 3.2. Нека $\sim$ е релация на еквивалентност върху A и $a \in A$. *Класът на еквивалентност* на a е множеството

$$[a]_\sim = \{x \in A \mid x \sim a\}.$$

Когато релацията е ясна от контекста, пишем просто $[a]$.

За да се определят класовете на еквивалентност, за всеки още необработен елемент $a \in A$ се намират всички елементи x, за които $x \sim a$. Полученото множество е $[a]$. След това всички негови елементи се отбелязват като обработени и процедурата се повтаря с елемент извън вече намерените класове.

Твърдение 3.3. Ако $\sim$ е релация на еквивалентност върху A, то за произволни $a, b \in A$ класовете $[a]$ и $[b]$ или съвпадат, или са непресичащи се.

Доказателство. Да допуснем, че съществува $x \in [a] \cap [b]$. Тогава $x \sim a$ и $x \sim b$. От симетричността следва $a \sim x$, а от транзитивността с $x \sim b$ получаваме $a \sim b$.
Нека $y \in [a]$. Тогава $y \sim a$ и $a \sim b$, следователно $y \sim b$, т.е. $y \in [b]$. Значи $[a] \subseteq [b]$. Аналогично $[b] \subseteq [a]$, следователно $[a] = [b]$. □

Следствие 3.4. Класовете на еквивалентност на релация на еквивалентност образуват разбиение на множеството A: всеки елемент принадлежи на точно един клас.

Доказателство. Рефлексивността дава $a \in [a]$, така че класовете покриват A. Ако $c \in [a] \cap [b]$, симетричността и транзитивността дават $a \sim b$, а после $x \sim a \iff x \sim b$. Значи $[a] = [b]$. Следователно различните класове са непресичащи се и всеки елемент е в точно един от тях. □

▷ **Пример 3.5.** Върху $\mathbb{Z}$ да е зададена релацията

$$a \sim b \iff 3 \mid (a - b).$$

Тя разделя целите числа на три класа:

$$[0] = \{\ldots, -6, -3, 0, 3, 6, \ldots\},$$
$$[1] = \{\ldots, -5, -2, 1, 4, 7, \ldots\},$$
$$[2] = \{\ldots, -4, -1, 2, 5, 8, \ldots\}.$$

За да се намери класът на дадено число, е достатъчно да се определи остатъкът му при деление на 3.

3.2 Графи и основни понятия

3.2.1 Неориентирани графи

Дефиниция 3.6. Краен неориентиран граф е наредена двойка

$$G = (V, E),$$

където V е непразно крайно множество, чиито елементи се наричат *върхове*, а

$$E \subseteq \{X \subseteq V \mid |X| = 2\}$$

е множество от *ребра*.

Тъй като всяко ребро е двуелементно множество, в неориентиран граф реброто между u и v се записва като $\{u, v\}$; редът на върховете няма значение. В тази дефиниция няма примки и две еднакви ребра не могат да се срещат.

Дефиниция 3.7. Нека $G = (V, E)$ е граф.

1. Ако $E = \varnothing$, графът се нарича *празен*.
2. Ако $|V| = 1$, графът се нарича *тривиален*.
3. Върховете $u, v \in V$ са *съседни*, ако $\{u, v\} \in E$.
4. Ребрата $e_1, e_2 \in E$ са *инцидентни*, ако
$$e_1 \cap e_2 \neq \varnothing.$$
5. Реброто $e \in E$ е инцидентно с върха $u \in V$, ако $u \in e$.

Дефиниция 3.8. За връх $u \in V$ дефинираме:

$$N(u) = \{v \in V \mid u \text{ и } v \text{ са съседни}\},$$

$$N[u] = N(u) \cup \{u\},$$

$$I(u) = \{e \in E \mid u \in e\}.$$

Числото

$$d(u) = |I(u)|$$

се нарича *степен* на върха u.

Максималната и минималната степен на графа се означават съответно с

$$\Delta(G) = \max\{d(u) \mid u \in V\}, \qquad \delta(G) = \min\{d(u) \mid u \in V\}.$$

3.2.2 Ориентирани графи и мултиграфи

Дефиниция 3.9. Краен ориентиран граф е наредена двойка

$$G = (V, E),$$

където V е непразно крайно множество от върхове и

$$E \subseteq (V \times V) \setminus \{(u, u) \mid u \in V\}.$$

Елементите на E се наричат ориентирани ребра или дъги.

Дъгата (u, v) има начало u и край v. За разлика от неориентираното ребро, като цяло $(u, v) \neq (v, u)$.

Дефиниция 3.10. Неориентиран мултиграф е наредена тройка

$$G = (V, E, f_G),$$

където V е непразно множество от върхове, E е множество от ребра, $V \cap E = \varnothing$, и

$$f_G : E \longrightarrow \{X \subseteq V \mid |X| = 2\}$$

е свързваща функция.

Свързващата функция указва краищата на всяко ребро. Различни елементи на E могат да имат един и същ образ чрез f_G, поради което мултиграфът позволява успоредни ребра.

Дефиниция 3.11. Ориентиран мултиграф е наредена тройка

$$G = (V, E, f_G),$$

където V е непразно множество от върхове, E е множество от ребра, $V \cap E = \varnothing$, и

$$f_G : E \longrightarrow V \times V$$

е свързваща функция.

† **Допълнение 3.12 (Бележка).** В представените материали за ориентиран мултиграф се допуска $f_G(e) = (u, u)$, тоест примки. Това се различава от дадената по-горе дефиниция за обикновен ориентиран граф, в която примките са изключени.

3.3 Пътища, цикли и свързаност

3.3.1 Пътища и цикли

Дефиниция 3.13. Нека $G = (V, E, f_G)$ е мултиграф. *Път* в G е алтерниращa редица от върхове и ребра

$$p = (u_0, e_{k_0}, u_1, e_{k_1}, \ldots, e_{k_{l-1}}, u_l),$$

където $l \geq 0$, $u_i \in V$ за $0 \leq i \leq l$, $e_{k_j} \in E$ за $0 \leq j \leq l-1$, и всяко ребро e_{k_j} свързва последователните върхове u_j и u_{j+1}.

При ориентиран мултиграф изискването е преминаването по всяко ребро да е в посоката, зададена от свързващата функция.

Върховете u_0 и u_l се наричат съответно начало и край на пътя, а останалите върхове са вътрешни. Дължината на пътя е броят ребра в него:

$$\|p\| = l.$$

Дефиниция 3.14. Пътят p се нарича *прост*, ако всички негови върхове и ребра са различни.

Дефиниция 3.15. Пътят

$$p = (u_0, e_{k_0}, u_1, \ldots, e_{k_{l-1}}, u_l)$$

се нарича *цикъл*, ако $u_0 = u_l$. Той е *прост цикъл*, ако има поне едно ребро, върховете $u_0, u_1, \ldots, u_{l-1}$ са по двойки различни и ребрата му са по двойки различни. С други думи, единственото допустимо повторение на връх е $u_0 = u_l$.

3.3.2 Свързаност

Дефиниция 3.16. Нека $G = (V, E)$ е неориентиран граф. Върховете $u, v \in V$ са *свързани*, ако съществува път от u до v. Графът G е *свързан*, ако всеки два негови върха са свързани.

Дефиниция 3.17. Ориентиран граф е *слабо свързан*, ако неориентираният граф, получен чрез премахване на посоките на дъгите, е свързан. Той е *полусвързан*, ако за всеки два върха u и v има ориентиран път поне в едната посока: от u до v или от v до u. Той е *силно свързан*, ако за всеки два върха има ориентиран път и от първия до втория, и от втория до първия.

Дефиниция 3.18. Нека $G = (V, E)$ е неориентиран граф. Релацията на достижимост R_G върху V е зададена чрез

$$R_G \subseteq V \times V, \qquad uR_Gv \iff u \text{ и } v \text{ са свързани.}$$

Твърдение 3.19. Релацията R_G на достижимост в неориентиран граф е релация на еквивалентност.

Доказателство. За всеки връх u съществува тривиален път с дължина 0 от u до u, следователно uR_Gu.
Ако има път от u до v, същата последователност от ребра, обходена в обратен ред, е път от v до u. Следователно uR_Gv предполага vR_Gu.
Накрая, ако има път от u до v и път от v до w, те могат да се съединят в път от u до w. Следователно релацията е транзитивна. □

Дефиниция 3.20. Подграфите на G, индуцирани от класовете на еквивалентност на R_G, се наричат *свързани компоненти* на G.

Еквивалентно, свързаните компоненти са максималните по включване свързани подграфи. Следователно намирането на свързаните компоненти е конкретен случай на определяне на класовете на еквивалентност на релация.

3.4 Дървета

Дефиниция 3.21. *Дърво* е неориентиран граф, който е свързан и ацикличен.

Тривиалният граф е дърво. В дървото между два върха няма цикъл; при добавяне на нов връх към дърво и свързването му с точно едно ребро към стар връх не се създава цикъл.

Дефиниция 3.22. *Кореново дърво* определяме индуктивно. Базата е тривиалният граф

$$(\{r\}, \varnothing),$$

който има корен r. В индуктивната стъпка към вече построено кореново дърво се добавя нов връх u и едно ребро, което го свързва с връх v от старото дърво. Коренът се запазва.

Теорема 3.23. Всяко кореново дърво е дърво.

Доказателство. Доказателството е чрез индукция по построението. Базовият едновърхов граф е свързан и няма цикъл.
Нека $T = (V, E)$ е свързано и ациклично дърво. Добавяме нов връх $u \notin V$ и едно ребро $\{v, u\}$, където $v \in V$. Полученият граф е свързан, защото от всеки стар връх има път до v, а след това реброто $\{v, u\}$ води до u. Новият връх е инцидентен само с едно ребро, затова не може да участва в цикъл. Старите ребра не образуват цикъл по индукционното предположение. Следователно полученият граф също е дърво. □

Обратно, всяко крайно дърво с избран корен може да се получи чрез това построение. Ако има повече от един връх, дървото има листо, различно от корена. Премахването на листото и единственото му инцидентно ребро оставя дърво със същия корен; твър-

дението следва с индукция по броя върхове, като накрая листото и реброто се добавят обратно.

Следователно кореновото дърво може еквивалентно да се разглежда като дърво с избран връх r, наречен *корен*. Изборът на корен задава йерархия: всеки връх, различен от корена, има единствен непосредствен предшественик по пътя към корена.

Теорема 3.24. Нека $T = (V, E)$ е кореново дърво. Тогава

$$|V| = |E| + 1.$$

Доказателство. Използваме индукция по построението на кореновото дърво. За базата

$$T = (\{r\}, \varnothing)$$

имаме $|V| = 1 = 0 + 1 = |E| + 1$.
При индуктивната стъпка се добавят едновременно един нов връх и едно ново ребро. Ако за старото дърво е изпълнено $|V| = |E| + 1$, то за новото дърво T' е в сила

$$|V(T')| = |V| + 1 = |E| + 2 = |E(T')| + 1.$$

□

Дефиниция 3.25. Нека $G = (V, E)$ е граф. *Покриващо дърво* на G е дърво

$$T = (V, E'), \qquad E' \subseteq E.$$

Покриващото дърво съдържа всички върхове на графа, но само част от неговите ребра.

Теорема 3.26. Краен непразен неориентиран граф $G = (V, E)$ има покриващо дърво тогава и само тогава, когато е свързан.

Доказателство. Ако има покриващо дърво, пътят в него между всяка двойка върхове е път и в G, следователно G е свързан. Обратно, за краен непразен свързан неориентиран граф започваме с един връх. Докато не сме включили всички върхове, свързаността осигурява ребро от включен връх към невключен. Добавяме това ребро и новия връх. Свързаността на построеното подмножество се запазва и цикъл не възниква, защото новият връх има само едно добавено ребро. След $|V| - 1$ стъпки получаваме покриващо дърво. □

3.5 Обхождания на графи

Обхождането систематично посещава върховете, достижими от начален връх. При откриването на нов връх се запомня реброто, по което е достигнат; така получените ребра образуват покриващо дърво на достигнатата свързана компонента.

3.5.1 Стек и опашка

Дефиниция 3.27. *Стек* е структура от тип LIFO: последно добавеният елемент се извежда първи. Празният стек означаваме с $\{\}$. Ако S е стек и x е елемент, то push(S, x) добавя x на върха на стека, top(S) връща горния елемент, а pop(S) премахва горния елемент.

Дефиниция 3.28. *Опашка* е структура от тип FIFO: първо добавеният елемент се извежда първи. Операцията push(Q, x) добавя елемент в края, top(Q) връща първия елемент, а pop(Q) го премахва.

3.5.2 Схема за обхождане

Нека върховете на ориентиран мултиграф са означени с $1, \ldots, n$, а началният връх е 1. Поддържа се булев масив visited$[1, n]$. В началото всички стойности са `False`; началният връх се поставя в помощна структура S и се маркира като посетен.

Докато S не е празна, от нея се извежда връх x. За всяка дъга (x, y), ако y не е посещаван, той се маркира като посетен и се добавя в S. Изборът на конкретна структура за S определя вида обхождане.

3.5.3 Обхождане в широчина

★ **Дефиниция 3.29.** *Обхождане в широчина* (BFS) е обхождане по общата схема, при което помощната структура е опашка.

BFS първо разглежда всички върхове на разстояние 1 от началния връх, след това върховете на разстояние 2 и т.н. Ако при откриването на y чрез x запомним $\pi[y] = x$, стойностите π задават дърво на обхождането с корен началния връх.

```
BFS(G=(V,E))
for i <- 1 to n
    colour[i] <- white
    pi[i] <- NIL
    d[i] <- infinity
colour[1] <- gray
d[1] <- 0
create queue Q
Enqueue(Q,1)
while not isempty(Q)
    x <- Dequeue(Q)
    for y in adj(x)
        if colour[y] = white
            colour[y] <- gray
            pi[y] <- x
            d[y] <- d[x] + 1
```

```
            Enqueue(Q,y)
    colour[x] <- black
return colour,pi,d
```

★ **Теорема 3.30.** Обхождането в широчина намира най-късите пътища от началния връх до всички достижими върхове в графа.

Доказателство. Разстоянието тук е броят на ребрата. Началният връх s получава $d[s] = 0$. FIFO опашката обработва изцяло върховете на слой k, преди върховете на слой $k+1$. По индукция нека всички върхове с действително разстояние най-много k са открити с правилни стойности. Връх v на разстояние $k+1$ има предшественик на разстояние k по най-къс път; при обработката му v се открива най-късно със стойност $k+1$. Всяка присвоена стойност е дължина на реален път и не може да е по-малка от най-късото разстояние. Така $d[v] = k+1$. Всички достижими върхове попадат в някой слой, което завършва доказателството. □

Стойността $d[y]$, присвоена при първото откриване на y, е дължината на най-къс път от началния връх до y. Това следва от факта, че опашката обработва върховете по нарастване на разстоянието от началния връх.

3.5.4 Обхождане в дълбочина

★ **Дефиниция 3.31.** *Обхождане в дълбочина* (DFS) е обхождане, при което се продължава към необходим съсед, докато това е възможно; когато няма такъв съсед, алгоритъмът се връща назад. Итеративната реализация използва стек, тоест структура LIFO.

Следната рекурсивна реализация изразява същата идея:

```
DFS(G=(V,E))
for i <- 1 to n
    colour[i] <- white
    N[i] <- NIL
DFS-Visit(G,1)
return colour,N

DFS-Visit(G=(V,E), x in V)
colour[x] <- gray
foreach y in adj[x]
    if colour[y] = white
        N[y] <- x
        DFS-Visit(G,y)
colour[x] <- black
```

При достигане на необходим връх y от x, стойността $N[y] = x$ запомня непосредствения предшественик на y в дървото на DFS. Ако графът не е свързан, едно обхождане от из-

бран начален връх посещава само неговата компонента; за обхождане на всички върхове алгоритъмът се стартира от всеки все още необходен връх.

И DFS, и BFS могат да бъдат изменени за намиране на път между два върха: единият се избира за начален, а обхождането се прекратява при достигане на другия. Пътят се възстановява чрез запомнените предшественици.

3.6 Ойлерови пътища и цикли

Дефиниция 3.32. Нека G е свързан мултиграф. *Ойлеров път* е път, който преминава точно веднъж през всяко ребро на графа. *Ойлеров цикъл* е Ойлеров път, чийто начален и краен връх съвпадат. Граф, който притежава Ойлеров цикъл, се нарича *Ойлеров граф*.

★ **Теорема 3.33 (Критерий за Ойлеров цикъл в свързан мултиграф).** Нека G е свързан мултиграф (в частност — свързан граф) с поне едно ребро. Тогава G е Ойлеров тогава и само тогава, когато всеки негов връх има четна степен. Кратните ребра и примките не изискват отделно разглеждане: всяка примка добавя 2 към степента на върха си, а кратните ребра се броят поотделно.

Доказателство. Нека G има Ойлеров цикъл. Всеки път, когато цикълът влиза във връх, той трябва да го напусне по друго ребро. Следователно ребрата, инцидентни с произволен връх, се групират по двойки: едно за влизане и едно за излизане. Значи степента на всеки връх е четна.
Обратно, нека G е свързан и всеки връх има четна степен. Избираме начален връх и вървим по неизползвани ребра, като всяко използвано ребро се маркира. Пътят не може да завърши в връх, различен от началния: при такъв краен връх броят използвани инцидентни ребра би бил нечетен, което противоречи на четността на степента му. Така получаваме цикъл.
Ако цикълът съдържа всички ребра, той е Ойлеров. В противен случай поради свързаността съществува връх от получения цикъл, инцидентен с неизползвано ребро. От този връх се построява нов цикъл по неизползвани ребра и се вмъква в стария. Процесът се повтаря. При всяка стъпка броят неизползвани ребра намалява, затова след краен брой стъпки се получава цикъл, съдържащ всяко ребро точно веднъж. □

Следствие 3.34. Свързан мултиграф има Ойлеров път, който не е цикъл, тогава и само тогава, когато точно два негови върха са с нечетна степен, а всички останали са с четна степен.

Доказателство. При Ойлеров път всяко посещение на вътрешен връх използва двойка ребра: за влизане и излизане. Само началото и краят имат по едно несдвоено ребро, затова са точно двата върха с нечетна степен. Обратно, ако нечетните върхове са u, v, добавяме още едно ребро между тях. Всички степени стават четни, а графът остава свързан, затова по теоремата за Ойлеров цикъл има такъв цикъл.

Премахваме добавеното ребро и прочитаме останалата част от цикъла от единия му край: получаваме Ойлеров път от u до v. □

3.7 Примерни задачи по анотацията

Анотацията назовава три типа задачи. Общите алгоритми са дадени по-горе; тук и трите са изпълнени докрай върху конкретни входове.

Навсякъде в този раздел работим с графа

$$G: \quad V = \{1,2,3,4,5,6\}, \qquad E = \{12,\ 13,\ 23,\ 24,\ 35,\ 45,\ 46,\ 56\},$$

където uv означава реброто между u и v. Списъците на съседите са подредени по нарастване, за да е обхождането еднозначно:

$$1:2,3 \quad 2:1,3,4 \quad 3:1,2,5 \quad 4:2,5,6 \quad 5:3,4,6 \quad 6:4,5.$$

Степените са

$$\deg 1 = 2, \quad \deg 2 = 3, \quad \deg 3 = 3, \quad \deg 4 = 3, \quad \deg 5 = 3, \quad \deg 6 = 2.$$

3.7.1 Задача 1: покриващо дърво в широчина и в дълбочина

▷ **Пример 3.35 (BFS от връх 1).** Опашката се обработва отляво надясно; при всеки връх се обхождат необходените му съседи по реда от списъка.

Изваден връх	Нови върхове	Ребра на дървото	Опашка след стъпката
1	2, 3	12, 13	2, 3
2	4	24	3, 4
3	5	35	4, 5
4	6	46	5, 6
5	—	—	6
6	—	—	—

Покриващото дърво е $\{12, 13, 24, 35, 46\}$ — пет ребра при шест върха, както изисква $|V| = |E| + 1$. Разстоянията от началния връх са

$$d(1) = 0, \quad d(2) = d(3) = 1, \quad d(4) = d(5) = 2, \quad d(6) = 3,$$

и това са дължините на най-късите пътища, защото BFS обхожда върховете по нарастващо разстояние.

▷ **Пример 3.36 (DFS от връх 1).** Рекурсията влиза колкото може по-дълбоко, преди да се върне:

$$1 \to 2 \to 3 \to 5 \to 4 \to 6,$$

след което от 6 няма необходени съседи и обхождането се връща по стека до 1. Записаните предшественици са

$$N[2] = 1, \quad N[3] = 2, \quad N[5] = 3, \quad N[4] = 5, \quad N[6] = 4,$$

тоест покриващото дърво е $\{12, 23, 35, 45, 46\}$ и е път. Ребрата 13, 24 и 56 остават извън дървото — те са обратни ребра към вече посетени върхове.

Забележка 3.37. Двете дървета са различни, макар графът и началният връх да са едни и същи. При такава задача винаги се посочва и редът на съседите: без него отговорът не е еднозначен.

3.7.2 Задача 2: Ойлеров цикъл или доказателство, че такъв няма

▷ **Пример 3.38 (Отрицателен отговор за G).** В G върховете $2, 3, 4, 5$ са с нечетна степен. По критерия Ойлеров цикъл няма, защото не всички степени са четни; по следствието няма и Ойлеров път, защото върховете с нечетна степен не са точно два, а четири. Това е пълен отговор — позоваване на критерия, не търсене на цикъл „на ръка“.

▷ **Пример 3.39 (Построяване по Hierholzer).** Нека K_5 е пълният граф върху $\{1, 2, 3, 4, 5\}$. Всички степени са 4, графът е свързан, следователно има Ойлеров цикъл. Строим го на три стъпки.
Първо тръгваме от 1 по неизползвани ребра, докато се върнем в 1:

$$C_0 : \ 1 \to 2 \to 3 \to 1 \qquad (\text{ребра } 12, 23, 13).$$

Връх 2 още има неизползвани ребра. Строим цикъл от 2:

$$2 \to 4 \to 5 \to 2 \qquad (\text{ребра } 24, 45, 25),$$

и го вмъкваме в C_0 на мястото на 2:

$$C_1 : \ 1 \to 2 \to 4 \to 5 \to 2 \to 3 \to 1.$$

Остават $14, 15, 34, 35$. Строим цикъл от 3:

$$3 \to 4 \to 1 \to 5 \to 3 \qquad (\text{ребра } 34, 14, 15, 35),$$

и го вмъкваме на мястото на 3:

$$C_2 : \ 1 \to 2 \to 4 \to 5 \to 2 \to 3 \to 4 \to 1 \to 5 \to 3 \to 1.$$

Използвани са всички $\binom{5}{2} = 10$ ребра, всяко по веднъж, а началото съвпада с края. Това е Ойлеров цикъл.

3.7.3 Задача 3: разбиване на ребрата на минимален брой пътища без общо ребро

Тук „път“ се разбира в смисъла на §3.3: върхове могат да се повтарят. Иска се разбиване на E, тоест всяко ребро принадлежи на точно един от пътищата и се използва в него точно веднъж.

★ **Теорема 3.40 (Минимален брой пътища).** Нека G е свързан мултиграф с поне едно ребро и нека точно $2k$ от върховете му са с нечетна степен (броят на нечетните върхове винаги е четен). Тогава минималният брой пътища, на които може да се разбие E така, че никои два да нямат общо ребро, е

$$\max(k, 1).$$

Доказателство. *Долна граница.* Разглеждаме произволно разбиване на E на пътища и фиксираме връх v. Всяко преминаване на път през v като вътрешен връх използва две от инцидентните на v ребра; същото важи и когато път започва и завършва в v. Само път, за който v е точно единият от двата края, използва нечетен брой инцидентни на v ребра. Тъй като ребрата са разпределени между пътищата без повторение,

$$\deg v \equiv \#\{\text{пътища, за които } v \text{ е точно единият край}\} \pmod 2.$$

Следователно всеки връх с нечетна степен е край на поне един път. Всеки път има най-много два края, а нечетните върхове са $2k$, откъдето пътищата са поне k. Освен това $E \neq \varnothing$, тоест поне един път е нужен винаги.
Конструкция, достигаща границата. Ако $k = 0$, всички степени са четни и по критерия за Ойлеров цикъл целият граф е един-единствен път (при това затворен), тоест 1 път е достатъчен.
Нека $k \geq 1$ и нека нечетните върхове са $u_1, v_1, \ldots, u_k, v_k$ — $2k$ различни върха, разбити произволно на k двойки. Добавяме k нови ребра $u_i v_i$. Всяка степен нараства с точно 1 при нечетните върхове и остава непроменена при останалите, затова в получения мултиграф G' всички степени са четни; G' остава свързан. По критерия G' има Ойлеров цикъл C.
Премахваме от C добавените k ребра. Понеже $2k$-те върха са различни, никои две добавени ребра нямат общ връх, следователно никои две не стоят една до друга в C; значи между всеки две последователни премахнати ребра остава поне едно първоначално ребро. Циклично C се разпада на точно k непразни дъги. Всяка дъга е път, дъгите нямат общо ребро, а обединението им е точно E. Получихме k пътя. □

Следствие 3.41. За граф с няколко свързани компоненти минималният брой пътища е сумата от $\max(k_i, 1)$ по компонентите i, съдържащи поне едно ребро. Изолираните върхове не изискват път.

Доказателство. Път е свързан подграф, затова никой път не пресича две компоненти. Задачата се разпада независимо по компонентите и се прилага теоремата за всяка от тях. □

▷ **Пример 3.42 (Разбиване на G).** В G нечетните върхове са $2, 3, 4, 5$, тоест $2k = 4$ и $k = 2$: два пътя са необходими и достатъчни.
Двойките избираме кръстосано — $(2, 5)$ и $(3, 4)$ — защото тези ребра още не съществуват. Добавяме $e_a = 25$ и $e_b = 34$. Сега всички степени са четни:

$$\deg 1 = 2,\ \deg 2 = 4,\ \deg 3 = 4,\ \deg 4 = 4,\ \deg 5 = 4,\ \deg 6 = 2.$$

Един Ойлеров цикъл в разширения мултиграф е

$$2 \xrightarrow{12} 1 \xrightarrow{13} 3 \xrightarrow{e_b} 4 \xrightarrow{24} 2 \xrightarrow{23} 3 \xrightarrow{35} 5 \xrightarrow{56} 6 \xrightarrow{46} 4 \xrightarrow{45} 5 \xrightarrow{e_a} 2.$$

Премахваме e_a и e_b и оставащите две дъги са търсените пътища:

$$P_1 :\ 2 - 1 - 3, \qquad P_2 :\ 4 - 2 - 3 - 5 - 6 - 4 - 5.$$

Проверка: P_1 използва $12, 13$; P_2 използва $24, 23, 35, 56, 46, 45$. Заедно това са всичките 8 ребра, всяко по веднъж, и общо ребро няма. Краищата на двата пътя са $2, 3$ и $4, 5$ — точно четирите нечетни върха, както изисква долната граница.

Забележка 3.43. Отговорът не е единствен: друг избор на двойки или друг Ойлеров цикъл дава друга двойка пътища. Единствен е *броят* им. На изпита се иска и обосновка на минималността, тоест аргументът за долната граница чрез нечетните върхове.

3.8 Изпитен фокус

- Да се дадат определения за неориентиран граф, ориентиран граф, мултиграф, съседство, инцидентност, степен на връх, път, прост път и цикъл.
- Да се дефинират свързан граф, слаба и силна свързаност, релация на достижимост и свързани компоненти.
- Да се възпроизведе доказателствената идея, че достижимостта в неориентиран граф е релация на еквивалентност; класовете ⊠ са свързаните компоненти.
- Да се дефинират дърво, кореново дърво и покриващо дърво.
- Да се докаже чрез индукция, че за кореново дърво е изпълнено $|V| = |E| + 1$.
- Да се формулира критерият: граф има покриващо дърво точно когато е свързан.
- Да се опишат BFS и DFS, ролята на опашката при BFS и на стека или рекурсията при DFS, както и дървото на обхождането.
- Да се формулира, че BFS намира най-къси пътища от началния връх.
- Да се дефинират Ойлеров път и Ойлеров цикъл.
- Да се формулира и обясни критерият за Ойлеров цикъл: свързан граф е Ойлеров точно когато всички върхове са с четна степен.
- Да се опише идеята на алгоритъма на Hierholzer: строене на цикъл по неизползвани ребра и вмъкване на допълнителни цикли.

- Да се формулира условието за Ойлеров път, който не е цикъл: точно два върха са с нечетна степен.
- **Задача 1 от анотацията:** да се построи покриващо дърво на зададен граф в широчина или в дълбочина, с изписан ред на обхождането и на ребрата на дървото (§3.7.1).
- **Задача 2 от анотацията:** да се построи Ойлеров цикъл или път в зададен мултиграф чрез вмъкване на цикли, или да се докаже, че такъв няма, чрез броя на върховете с нечетна степен (§3.7.2).
- **Задача 3 от анотацията:** да се разбият ребрата на минимален брой пътища без общо ребро — долна граница k от $2k$-те нечетни върха, конструкция чрез сдвояване, Ойлеров цикъл и премахване на добавените ребра (§3.7.3).
- Релациите на еквивалентност в §3.1 са предварителен материал от Въпрос 1; по този въпрос от тях се използва само, че достижимостта е еквивалентност и класовете ѝ са свързаните компоненти.

Въпрос 4

Характеризация на регулярните езици. Теорема на Майхил-Нероуд.

По конспекта: какво трябва да мога

Дефиниции: регулярен език, детерминиран краен автомат, автоматен език — §4.1, стр. 65.
Формулировка, без доказателство: теорема на Клини — §4.3, стр. 68.
Дефиниция: релацията $\approx_L$, чрез $\alpha^{-1}(L)$ или като релация на Майхил–Нероуд — §4.5, стр. 70.
Доказателство: $\approx_L$ е релация на еквивалентност и е дясно инвариантна — §4.5, стр. 70.
Доказателство: индекс $n \Rightarrow$ тотален ДКА с n състояния — §4.6, стр. 71.
Доказателства: ДКА с n състояния $\Rightarrow$ индекс не повече от n; и оттам: L е регулярен точно когато $\approx_L$ има краен индекс — §4.7, стр. 72.
Задача 1: по даден регулярен език да се построи *минимален* тотален ДКА, с обосновка — §4.6 и §4.8, стр. 73.
Задача 2: да се определи с обосновка дали даден език е регулярен — §4.7, стр. 72.
† **Допълнително:** доказателството на теоремата на Клини и построенията за затвореност (§4.3), лемата за изпомпване (§4.4) и алгоритъмът за минимизация (§4.8) надхвърлят изискването; §4.2 е предварителен материал от Въпрос 1.
Литература: [34], [38], [43].

4.1 Основни понятия

Дефиниция 4.1 (Азбука, дума и формален език)**.** Нека Σ е фиксирана непразна крайна азбука. С Σ^* означаваме множеството на всички крайни думи над Σ, включително празната дума ε. Формален език над Σ е произволно множество $L \subseteq \Sigma^*$.

Дефиниция 4.2 (Операции върху езици)**.** Конкатенацията на думи $u, v \in \Sigma^*$ се означава с uv. За езици $L_1, L_2 \subseteq \Sigma^*$ дефинираме

$$L_1 L_2 = \{uv \mid u \in L_1,\ v \in L_2\}.$$

Освен това

$$L^0 = \{\varepsilon\}, \qquad L^{n+1} = L^n L, \qquad L^* = \bigcup_{n \in \mathbb{N}} L^n, \qquad L^+ = LL^*.$$

Дефиниция 4.3. Регулярните изрази над Σ се задават индуктивно:

$$r ::= \varnothing \mid \varepsilon \mid a \mid (r_1 + r_2) \mid (r_1 r_2) \mid r_1^*, \qquad a \in \Sigma.$$

Езикът, зададен от регулярен израз r, се означава с $\mathcal{L}[\![r]\!]$ и се определя чрез

$$\mathcal{L}[\![\varnothing]\!] = \varnothing, \qquad \mathcal{L}[\![\varepsilon]\!] = \{\varepsilon\}, \qquad \mathcal{L}[\![a]\!] = \{a\},$$

$$\mathcal{L}[\![r_1 + r_2]\!] = \mathcal{L}[\![r_1]\!] \cup \mathcal{L}[\![r_2]\!],$$

$$\mathcal{L}[\![r_1 r_2]\!] = \mathcal{L}[\![r_1]\!]\mathcal{L}[\![r_2]\!], \qquad \mathcal{L}[\![r_1^*]\!] = \mathcal{L}[\![r_1]\!]^*.$$

Езикът L се нарича *регулярен*, ако $L = \mathcal{L}[\![r]\!]$ за някакъв регулярен израз r.

Еквивалентно, регулярните езици са най-малкият клас езици, съдържащ

$$\varnothing, \qquad \{\varepsilon\}, \qquad \{a\} \quad (a \in \Sigma),$$

и затворен относно обединение, конкатенация и звезда на Клини.

Дефиниция 4.4. Детерминиран краен автомат (ДКА) е петорка

$$\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle,$$

където Q е крайно множество от състояния, $q_0 \in Q$ е начално състояние, $F \subseteq Q$ е множество от приемащи състояния и

$$\delta : Q \times \Sigma \longrightarrow Q$$

е тотална функция на преходите.

Дефиниция 4.5 (Разширена функция на преходите)**.** Разширената функция на преходите

$$\delta^* : Q \times \Sigma^* \longrightarrow Q$$

се задава рекурсивно:

$$\delta^*(q, \varepsilon) = q, \qquad \delta^*(q, wa) = \delta(\delta^*(q, w), a),$$

за $q \in Q$, $w \in \Sigma^*$ и $a \in \Sigma$.

Дефиниция 4.6. Езикът, разпознаван от $\mathcal{A}$, е

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}.$$

Езикът се нарича *автоматен*, ако се разпознава от някакъв ДКА.

4.2 Индукция върху естествените числа

† **Допълнение 4.7 (Предварителен материал).** Принципът на индукцията принадлежи на Въпрос 1. Тук е повторен само защото всички доказателства по-нататък вървят по индукция върху дължината на дума. При повторение разделът може да се прескочи.

Доказателствата за думи често се извършват чрез индукция по дължината им. Това е приложение на принципа на математическата индукция.

Твърдение 4.8 (Принцип на математическата индукция)**.** Нека $P(n)$ е твърдение за всяко $n \in \mathbb{N}$. Ако:

1. $P(0)$ е вярно;
2. за всяко $n \in \mathbb{N}$ от $P(n)$ следва $P(n+1)$,

то $P(n)$ е вярно за всяко $n \in \mathbb{N}$.

Доказателство. Ако множеството от контрапримери е непразно, то има най-малък елемент n. От базата $n \neq 0$, а от минималността $P(n-1)$ е вярно. Индуктивната стъпка дава $P(n)$, противоречие. Тук използваме принципа за най-малкия елемент на $\mathbb{N}$, както във Въпрос 1. □

При индукция по дума обикновено се доказва, че твърдението е вярно за ε, а после – че от верността му за дума w следва верността му за wa, където $a \in \Sigma$.

Твърдение 4.9. За всеки ДКА $\mathcal{A}$ и всички $q \in Q$, $u, v \in \Sigma^*$ е изпълнено

$$\delta^*(q, uv) = \delta^*(\delta^*(q, u), v).$$

Доказателство. Доказваме твърдението чрез индукция по $|v|$.
При $v = \varepsilon$ имаме

$$\delta^*(q, u\varepsilon) = \delta^*(q, u) = \delta^*(\delta^*(q, u), \varepsilon).$$

Нека равенството е вярно за дума v. За произволна буква $a \in \Sigma$ получаваме

$$\begin{aligned} \delta^*(q, uva) &= \delta(\delta^*(q, uv), a) \\ &= \delta(\delta^*(\delta^*(q, u), v), a) \\ &= \delta^*(\delta^*(q, u), va). \end{aligned}$$

Следователно то е вярно и за va. По индукция твърдението е доказано за всяка дума v. □

Това равенство изразява, че автоматът може първо да прочете префикса u, а след това суфикса v, без това да променя крайното състояние.

4.3 Характеризация чрез крайни автомати

† **Допълнение 4.10 (Обхват по конспекта).** Анотацията изисква теоремата на Клини да бъде *формулирана*, но не и доказана. Целият този раздел — построенията за затвореност и доказателството чрез езиците $L(i,j,k)$ — е *допълнителен* спрямо изискваното. Той е запазен, защото обяснява защо теоремата е вярна и се използва в §4.7, но при разпределяне на времето за подготовка има по-нисък приоритет от доказателствата за $\approx_L$, които конспектът изисква изрично.

4.3.1 От регулярни изрази към автомати

Основните езици $\varnothing$, $\{\varepsilon\}$ и $\{a\}$ се разпознават от крайни автомати. За да преминем от регулярни изрази към автомати, е необходимо да използваме затвореността на автоматните езици относно регулярните операции.

Теорема 4.11. Класът на автоматните езици е затворен относно обединение, конкатенация, звезда на Клини и допълнение.

Доказателство. Нека L_1 и L_2 са автоматни езици.
За обединението се вземат автомати за L_1 и L_2 и се построява недетерминиран автомат, който има за начални състояния началните състояния на двата автомата. Той приема дума тогава и само тогава, когато поне едно от двете изчисления я приема. Следователно разпознава $L_1 \cup L_2$.
За конкатенацията се използва недетерминиран автомат, който започва работа като автомат за L_1. При достигане на приемащо състояние той може недетерминирано да започне разпознаване с автомата за L_2. Така една дума се приема точно когато може да се представи като uv, където $u \in L_1$ и $v \in L_2$. Следователно се разпознава L_1L_2.
За L^+ се използва автомат, който след всяко успешно разпознаване на дума от L може да започне ново разпознаване на дума от L. По този начин се приемат точно конкатенациите на положителен брой думи от L. Тъй като

$$L^* = \{\varepsilon\} \cup L^+,$$

и $\{\varepsilon\}$ е автоматен, следва, че L^* също е автоматен.
Накрая нека

$$\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$$

е ДКА за L. Понеже δ е тотална, за всяка дума автоматът достига точно едно крайно състояние. Ако заменим F с $Q \setminus F$, получаваме ДКА, който приема точно думите, неприети от $\mathcal{A}$. Следователно $\Sigma^* \setminus L$ е автоматен. □

† **Допълнение 4.12 (Бележка).** При построенията за обединение, конкатенация и звезда естествено се използва недетерминизъм. За да се получи ДКА, се прилага теоремата на Рабин–Скот: всеки недетерминиран краен автомат има еквивалентен детерминиран краен автомат.

Теорема 4.13 (Клини). За всеки език $L \subseteq \Sigma^*$ са еквивалентни:

$$L \text{ е регулярен} \quad \Longleftrightarrow \quad L \text{ е автоматен}.$$

Доказателство. Нека първо L е регулярен. Той се получава от основните езици чрез краен брой приложения на обединение, конкатенация и звезда. Основните езици са автоматни, а класът на автоматните езици е затворен относно тези операции. Следователно L е автоматен.
Обратно, нека

$$\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$$

е ДКА. Нека

$$Q = \{q_0, q_1, \ldots, q_{n-1}\}.$$

За $0 \leq i, j < n$ и $0 \leq k \leq n$ означаваме с $L(i,j,k)$ множеството от думите, които отвеждат автомата от q_i до q_j, като всички междинни състояния са сред

$$q_0, \ldots, q_{k-1}.$$

Ще докажем по индукция по k, че всеки език $L(i,j,k)$ е регулярен.
При $k = 0$ няма позволени междинни състояния. Тогава

$$L(i,j,0) = \begin{cases} \{\varepsilon\} \cup \{a \in \Sigma \mid \delta(q_i, a) = q_j\}, & i = j, \\ \{a \in \Sigma \mid \delta(q_i, a) = q_j\}, & i \neq j. \end{cases}$$

Това е краен език, следователно е регулярен.
Нека всички $L(i,j,k)$ са регулярни. Един път, чиито междинни състояния са с индекси най-много k, или изобщо не минава през q_k, или минава през q_k поне веднъж. В първия случай думата е в $L(i,j,k)$. Във втория случай тя има вид:

$$L(i,k,k)\, L(k,k,k)^*\, L(k,j,k).$$

Следователно

$$L(i,j,k+1) = L(i,j,k) \cup L(i,k,k) L(k,k,k)^* L(k,j,k).$$

Дясната страна е регулярна по индукционната хипотеза и затвореността на регулярните езици.
За $k = n$ получаваме всички възможни пътища между състоянията. Следователно

$$\mathcal{L}(\mathcal{A}) = \bigcup_{q_j \in F} L(0,j,n),$$

което е регулярен език. □

4.4 Лема за изпомпване на регулярните езици

★ **Теорема 4.14 (Лема за изпомпване).** Ако езикът L е регулярен, съществува число $p \geq 1$ такова, че всяка дума $w \in L$ с $|w| \geq p$ може да се представи като

$$w = xyz,$$

където $|xy| \leq p$, $|y| \geq 1$ и

$$xy^i z \in L \qquad \text{за всяко } i \geq 0.$$

Доказателство. Нека ДКА за L има p състояния. При прочитане на първите p символа на достатъчно дълга приета дума автоматът посещава $p+1$ състояния, ако се брои и началното. По принципа на Дирихле някое състояние се повтаря. Частта y между двете посещения е непразен цикъл и лежи в първите p символа. Цикълът може да бъде обходен произволен брой пъти, без да се промени приемането на остатъка z. □

Лемата дава необходимо, но не достатъчно условие за регулярност. За доказване на нерегулярност се предполага, че L е регулярен, избира се подходяща дума w и се показва, че за всяко допустимо разлагане $w = xyz$ съществува i, за което $xy^i z \notin L$.

4.5 Релацията на Майхил–Нероуд

Дефиниция 4.15. Нека $L \subseteq \Sigma^*$ и $u, v \in \Sigma^*$. Казваме, че u и v са *еквивалентни относно* L, и пишем

$$u \approx_L v,$$

ако

$$(\forall x \in \Sigma^*)\big(ux \in L \iff vx \in L\big).$$

Релацията $\approx_L$ се нарича релация на Майхил–Нероуд за L.

Еквивалентно, ако за дума u дефинираме лявата производна

$$u^{-1}(L) = \{x \in \Sigma^* \mid ux \in L\},$$

то

$$u \approx_L v \iff u^{-1}(L) = v^{-1}(L).$$

Твърдение 4.16. Релацията $\approx_L$ е релация на еквивалентност върху Σ^*.

Доказателство. Рефлексивността следва от

$$ux \in L \iff ux \in L.$$

Симетричността следва от симетричността на логическата еквивалентност. За транзитивност, ако $u \approx_L v$ и $v \approx_L w$, то за всяко $x \in \Sigma^*$ имаме

$$ux \in L \iff vx \in L \iff wx \in L,$$

следователно $u \approx_L w$. □

Класът на еквивалентност на думата u означаваме с

$$[u]_L = \{v \in \Sigma^* \mid u \approx_L v\}.$$

Твърдение 4.17 (Дясна инвариантност)**.** За всички $u, v \in \Sigma^*$ и $a \in \Sigma$ е изпълнено

$$u \approx_L v \Longrightarrow ua \approx_L va.$$

Доказателство. Нека $u \approx_L v$. За всяка дума $z \in \Sigma^*$ имаме

$$(ua)z = u(az) \in L \Longleftrightarrow v(az) = (va)z \in L.$$

Следователно $ua \approx_L va$. □

Твърдение 4.18. За всяка дума $u \in \Sigma^*$ е изпълнено

$$u \in L \Longleftrightarrow [u]_L \subseteq L.$$

Доказателство. Нека $u \in L$ и $v \in [u]_L$. Понеже $u \approx_L v$, за $x = \varepsilon$ получаваме

$$u \in L \Longleftrightarrow v \in L.$$

Следователно $v \in L$, тоест $[u]_L \subseteq L$.
Обратно, $u \in [u]_L$ по рефлексивност. Следователно от $[u]_L \subseteq L$ следва $u \in L$. □

4.6 Автоматът на Майхил–Нероуд

Дефиниция 4.19. За език $L \subseteq \Sigma^*$ автоматът на Майхил–Нероуд е

$$M_L = \langle \Sigma, Q_{M_L}, [\varepsilon]_L, \delta_{M_L}, F_{M_L} \rangle,$$

където

$$Q_{M_L} = \{[u]_L \mid u \in \Sigma^*\},$$
$$\delta_{M_L}([u]_L, a) = [ua]_L,$$
$$F_{M_L} = \{[u]_L \mid [u]_L \subseteq L\}.$$

Дясната инвариантност гарантира, че функцията на преходите е добре дефинирана: ако $[u]_L = [v]_L$, то $u \approx_L v$, откъдето $ua \approx_L va$, тоест

$$[ua]_L = [va]_L.$$

Твърдение 4.20. За всички $u, w \in \Sigma^*$ е изпълнено

$$\delta^*_{M_L}([u]_L, w) = [uw]_L.$$

Доказателство. Доказваме чрез индукция по $|w|$. За $w = \varepsilon$:

$$\delta^*_{M_L}([u]_L, \varepsilon) = [u]_L = [u\varepsilon]_L.$$

Нека твърдението е вярно за w. За $a \in \Sigma$:

$$\begin{aligned}\delta^*_{M_L}([u]_L, wa) &= \delta_{M_L}(\delta^*_{M_L}([u]_L, w), a) \\ &= \delta_{M_L}([uw]_L, a) \\ &= [uwa]_L.\end{aligned}$$

Следователно твърдението е вярно за всяка дума. □

Твърдение 4.21. Автоматът M_L разпознава L, тоест

$$\mathcal{L}(M_L) = L.$$

Доказателство. За $w \in \Sigma^*$:

$$\begin{aligned}w \in \mathcal{L}(M_L) &\iff \delta^*_{M_L}([\varepsilon]_L, w) \in F_{M_L} \\ &\iff [w]_L \subseteq L \\ &\iff w \in L.\end{aligned}$$

□

4.7 Теорема на Майхил–Нероуд

★ **Теорема 4.22 (Майхил–Нероуд).** За език $L \subseteq \Sigma^*$ са еквивалентни:

L е регулярен $\iff$ $\{[u]_L \mid u \in \Sigma^*\}$ е крайно множество.

Освен това, ако L е регулярен, M_L е минимален ДКА, който разпознава L.

Доказателство. Нека L е регулярен. По теоремата на Клини съществува ДКА

$$\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$$

с $\mathcal{L}(\mathcal{A}) = L$.
Нека $u, v \in \Sigma^*$ удовлетворяват

$$\delta^*(q_0, u) = \delta^*(q_0, v).$$

Тогава за всяко $x \in \Sigma^*$:

$$\delta^*(q_0, ux) = \delta^*(\delta^*(q_0, u), x) = \delta^*(\delta^*(q_0, v), x) = \delta^*(q_0, vx).$$

Следователно ux е приета дума тогава и само тогава, когато vx е приета дума. Значи

$$u \approx_L v.$$

Следователно думите, които отвеждат $\mathcal{A}$ в едно и също състояние, принадлежат на един и същ клас на Майхил–Нероуд. Понеже Q е крайно, броят на класовете на $\approx_L$ не превишава броя на състоянията на $\mathcal{A}$. Следователно M_L има краен брой състояния. Обратно, ако M_L има краен брой състояния, то той е ДКА и, както вече беше доказано,

$$\mathcal{L}(M_L) = L.$$

Следователно L е автоматен, а по теоремата на Клини – регулярен.
Нека сега $\mathcal{A}$ е произволен ДКА, който разпознава L. Предишното разсъждение показва, че всяко състояние на $\mathcal{A}$ може да съответства на най-много един клас на Майхил–Нероуд. Следователно броят на състоянията на $\mathcal{A}$ е поне броя на класовете $[u]_L$. Автоматът M_L има точно по едно състояние за всеки такъв клас. Значи никой ДКА за L не може да има по-малко състояния от M_L. □

Следствие 4.23. Ако могат да се намерят безкрайно много думи $u_0, u_1, u_2, \ldots$ такива, че за всеки две различни i, j съществува дума x с

$$u_i x \in L \quad \Longleftrightarrow \quad u_j x \notin L,$$

то L не е регулярен.

Доказателство. Условието означава, че думите u_i са по двойки нееквивалентни относно $\approx_L$. Следователно $\approx_L$ има безкрайно много класове. По теоремата на Майхил–Нероуд езикът не е регулярен. □

▷ **Пример 4.24.** Нека

$$L = \{a^n b^n \mid n \geq 0\}.$$

За $m \neq n$ думите a^m и a^n не са еквивалентни относно L. Наистина, ако например $m < n$, избираме $x = b^m$. Тогава

$$a^m b^m \in L, \qquad a^n b^m \notin L.$$

Следователно има безкрайно много класове на Майхил–Нероуд и L не е регулярен.

4.8 Минимизация на даден ДКА

Нека

$$\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$$

е ДКА. За $q \in Q$ дефинираме езика, приеман при начало от q:

$$\mathcal{L}_{\mathcal{A}}(q) = \{w \in \Sigma^* \mid \delta^*(q, w) \in F\}.$$

Две състояния са еквивалентни, ако не могат да бъдат различени с никакъв суфикс:

$$p \equiv_{\mathcal{A}} q \Longleftrightarrow \mathcal{L}_{\mathcal{A}}(p) = \mathcal{L}_{\mathcal{A}}(q).$$

Преди раздробяването се премахват всички състояния, които не са достижими от q_0. Това не променя разпознавания език, а е необходимо за минималност: недостижим клас не трябва да остава в резултата. По-нататък с Q означаваме само множеството на достижимите състояния, а множеството на финалните състояния се заменя с $F \cap Q$.

За построяване на класовете се използват приближенията

$$p \equiv^n_{\mathcal{A}} q \iff \{w \mid |w| \le n,\ \delta^*(p,w) \in F\} = \{w \mid |w| \le n,\ \delta^*(q,w) \in F\}.$$

При $n = 0$ думите с дължина най-много нула са само ε, затова непразните начални класове са

$$F \qquad \text{и} \qquad Q \setminus F.$$

Преходът от $\equiv^n_{\mathcal{A}}$ към $\equiv^{n+1}_{\mathcal{A}}$ се определя чрез

$$p \equiv^{n+1}_{\mathcal{A}} q \iff p \equiv^n_{\mathcal{A}} q \ \wedge\ (\forall a \in \Sigma)\, \delta(p,a) \equiv^n_{\mathcal{A}} \delta(q,a).$$

Алгоритъмът последователно раздробява класовете, докато при две последователни стъпки разбиението остане същото. Той завършва, защото всяко нетривиално раздробяване увеличава броя на класовете, а този брой не може да превиши $|Q|$. Получените непразни класове са състоянията на минималния еквивалентен ДКА.

4.9 Изпитен фокус

- Да се формулират Σ^*, конкатенация, степен на език, L^* и L^+.
- Да се даде индуктивната дефиниция на регулярен израз и семантиката му.
- Да се дефинират ДКА, разширена функция δ^* и разпознаван език.
- (Предварително, от Въпрос 1.) Да се докаже по индукция твърдение за естествените числа; стандартен пример е

  $$\delta^*(q, uv) = \delta^*(\delta^*(q,u), v).$$

- Да се обяснят построенията, доказващи затвореност на автоматните езици относно обединение, конкатенация, звезда и допълнение.
- Да се *формулира* теоремата на Клини: регулярен $\Leftrightarrow$ автоматен. Конспектът иска само формулировката; доказателството чрез езиците $L(i,j,k)$ и построенията за затвореност (§4.3) са допълнителни.
- Да се формулира лемата за изпомпване и да се използва като необходимо условие за регулярност при доказване на нерегулярност.
- Да се дефинира релацията $\approx_L$ и да се докаже, че е релация на еквивалентност и е дясно инвариантна.
- Да се построи автоматът M_L, да се провери, че преходната му функция е добре дефинирана, и да се докаже

  $$\delta^*_{M_L}([u]_L, w) = [uw]_L.$$

- Да се формулира и докаже теоремата на Майхил–Нероуд, включително минималността на M_L.
- Да се използва теоремата за доказване на нерегулярност чрез безкрайно много по двойки различими префикси.
- Да се опише алгоритъмът за минимизация чрез разбиенията $\equiv^n_{\mathcal{A}}$.
- **Задача 1 от анотацията:** по даден регулярен език да се построи *минимален* тотален ДКА и построението да се обоснове — чрез класовете на $\approx_L$ и автомата M_L (§4.6), или чрез минимизация на произволен ДКА (§4.8); минималността се обосновава с теоремата на Майхил–Нероуд.
- **Задача 2 от анотацията:** да се определи с обосновка дали даден език е регулярен — чрез краен индекс на $\approx_L$ или чрез безкрайно много по двойки различими префикси.

Въпрос 5

Лема за разрастването за контекстносвободни езици. Незатвореност относно сечение и допълнение.

По конспекта: какво трябва да мога

Дефиниции: контекстносвободна граматика, извод, контекстносвободен език, дърво на извод — §5.1, стр. 77.
Конструкции, без доказателства: затвореност относно регулярните операции — обединение, конкатенация, звезда на Клини — §5.4, стр. 81.
Формулировка и доказателство: лема за разрастването за КСЕ — §5.2, стр. 78.
Пример с доказателство: език, който не е контекстносвободен — §5.3, стр. 80.
Доказателство: незатвореност относно сечение — §5.5, стр. 82.
Доказателство: незатвореност относно допълнение — §5.6, стр. 83.
Типове задачи: по даден КСЕ да се построи КСГ с обосновка; да се определи с обосновка дали даден език е контекстносвободен.
Литература: [34], [38], [43].

5.1 Контекстносвободни граматики и дървета на извода

Дефиниция 5.1. Контекстносвободна граматика (КСГ) е четворка

$$G = \langle V, \Sigma, R, S \rangle,$$

където V е крайно множество от нетерминали, Σ е крайна азбука от терминали, $V \cap \Sigma = \varnothing$, $S \in V$ е начален нетерминал и

$$R \subseteq V \times (V \cup \Sigma)^*$$

е крайно множество от правила. Правилото $(A, \alpha) \in R$ се записва като $A \to \alpha$.

Дефиниция 5.2 (Едностъпков и краен извод)**.** Едностъпковият извод, породен от G, се означава с $\Rightarrow_G$. По определение

$$\lambda A \rho \Rightarrow_G \lambda \alpha \rho,$$

когато $A \to \alpha$ е правило на G и $\lambda, \rho \in (V \cup \Sigma)^*$. С $\Rightarrow_G^*$ означаваме рефлексивно-транзитивното затваряне на тази релация.

Дефиниция 5.3. Езикът, породен от граматиката G, е

$$L(G) = \{\omega \in \Sigma^* \mid S \Rightarrow_G^* \omega\}.$$

Език $L \subseteq \Sigma^*$ се нарича *контекстносвободен език* (КСЕ), ако съществува КСГ G, за която $L = L(G)$.

Дефиниция 5.4 (Дърво на извод)**.** Дърво на извод в G е крайно наредено кореново дърво. Коренът е означен с начален нетерминал. Ако вътрешен връх е означен с A, последователните му деца са означени със символите от дясната страна на някое правило $A \to \alpha$. При $A \to \varepsilon$ използваме едно листо ε. Резултатът е редицата от означенията на листата отляво надясно, като ε не добавя символ. За завършено дърво всички листа са терминали или ε.

Дефиниция 5.5 (Най-ляв извод)**.** Извод е най-ляв, когато във всяка стъпка се заменя най-левият нетерминал в текущата дума.

Твърдение 5.6 (Изводи и дървета)**.** За дума $\omega \in \Sigma^*$ са еквивалентни следните твърдения:

1. $S \Rightarrow_G^* \omega$, тоест $\omega \in L(G)$;
2. съществува дърво на извод с корен S и резултат ω;
3. съществува най-ляв извод на ω от S.

Доказателство. От извод строим дървото по индукция по броя стъпки: началното дърво е един лист S, а замяната на нетерминал разширява съответния лист с децата от приложеното правило. Резултатът на частичното дърво е текущата дума. Обратно, за дадено завършено дърво разширяваме винаги най-левия останал нетерминален лист според неговите деца в дървото. Крайността гарантира завършване и получаваме най-ляв извод на резултата. Всеки най-ляв извод е частен случай на извод. Това доказва трите импликации. □

5.2 Лема за разрастването

Лемата за разрастването, наричана още лема за покачването, дава необходимо условие един език да е контекстносвободен. Тя е особено полезна за доказване, че даден език *не* е контекстносвободен.

★ **Теорема 5.7 (Лема за разрастването за контекстносвободни езици).** Нека $L \subseteq \Sigma^*$ е контекстносвободен език. Тогава съществува число $p > 0$, наречено *константа на разрастването*, такова че за всяка дума $\alpha \in L$ с $|\alpha| \geq p$ съществува разлагане

$$\alpha = xyuvw,$$

за което са изпълнени:

1. $|yv| \geq 1$;
2. $|yuv| \leq p$;
3. за всяко $i \in \mathbb{N}$ е изпълнено
$$xy^iuv^iw \in L.$$

Идеята е, че достатъчно дълга дума има достатъчно високо дърво на извод. По един път в това дърво неизбежно се повтаря нетерминал. Частта от дървото между двете срещания на същия нетерминал може да се повтаря произволен брой пъти или да се премахва.

Доказателство. Нека $G = \langle V, \Sigma, R, S\rangle$ е КСГ, за която $L = L(G)$. Без ограничение на общността приемаме, че G е в нормална форма на Чомски. За непразните думи, които разглеждаме, всяко използвано правило е от един от видовете

$$A \to BC \qquad \text{или} \qquad A \to a,$$

където $A, B, C \in V$ и $a \in \Sigma$.
Нека

$$k = |V| \qquad \text{и} \qquad p = 2^k.$$

Ще покажем, че p е подходяща константа.
Използваме следното елементарно наблюдение за двоични дървета: ако във всяко корен–листо пътче има по-малко от k ребра, то дървото има по-малко от 2^k листа. Действително, на всяко ниво броят на върховете може най-много да се удвои, а листата са не повече от броя на възможните пътища с дължина, по-малка от k.

Нека сега $\alpha \in L$ и $|\alpha| \geq p = 2^k$. Разглеждаме дърво на извод за α и в него избираме най-дълъг корен–листо път. Тъй като граматиката е в нормална форма на Чомски, дървото е двоично: всеки вътрешен връх има най-много две деца. Броят на листата е $|\alpha|$, следователно дървото има поне 2^k листа. Ако всеки път имаше най-много k нетерминални върха, щеше да има най-много $k-1$ двоични разклонявания и най-много 2^{k-1} терминални листа. Понеже листата са поне 2^k, избраният най-дълъг път има поне $k+1$ нетерминални върха.
По този път се срещат поне $k+1$ нетерминала, а нетерминалите в V са само k. Следователно, по принципа на Дирихле, по пътя има два върха, означени с един и същ нетерминал A.
Сред последните $k+1$ срещания на нетерминали по този най-дълъг път избираме две срещания на един и същ нетерминал A. Съответстващата част от дървото дава разлагане

$$\alpha = xyuvw$$

и изводи

$$S \Rightarrow_G^* xAw, \qquad A \Rightarrow_G^* yAv, \qquad A \Rightarrow_G^* u.$$

Оттук за всяко $i \in \mathbb{N}$ получаваме

$$A \Rightarrow_G^* y^i A v^i \Rightarrow_G^* y^i u v^i,$$

а следователно

$$S \Rightarrow_G^* xy^i u v^i w.$$

Значи

$$xy^i u v^i w \in L$$

за всяко $i \in \mathbb{N}$.
Остава да проверим първите две условия. Изводът $A \Rightarrow_G^* yAv$ между двете различни срещания на A съдържа поне една стъпка. При първото разклоняване по избрания път другото дете е корен на поддърво, което в нормална форма на Чомски извежда поне един терминален символ. Този символ попада в y или във v. Следователно

$$|yv| \geq 1.$$

Горното срещане на A е сред последните $k+1$ нетерминални върха на избрания най-дълъг път. Затова от него до листото има най-много k разклонявания. Понеже избраният път е най-дълъг в цялото дърво, никой страничен път в поддървото на това срещане не е по-дълъг. Следователно височината на поддървото е най-много k. То е двоично, затова има най-много $2^k = p$ терминални листа. Тези листа дават точно думата yuv, откъдето

$$|yuv| \leq p.$$

Така трите условия са доказани. □

Забележка 5.8. В лемата е съществено разлагането да съществува за всяка достатъчно дълга дума, но то не е предварително зададено. При приложение за доказване на неконтекстносвободност трябва да се разгледат *всички* разлагания, удовлетворяващи услови-

ята на лемата, и за всяко от тях да се намери стойност на i, при която получената дума не принадлежи на разглеждания език.

5.3 Приложение: език, който не е контекстносвободен

▷ **Пример 5.9.** Езикът

$$L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$$

не е контекстносвободен.

Доказателство. Да допуснем противното: нека L е контекстносвободен. Нека p е константата от лемата за разрастването. Избираме думата

$$\alpha = a^p b^p c^p \in L.$$

Тя има дължина поне p, следователно според лемата има разлагане

$$\alpha = xyuvw,$$

за което

$$|yuv| \leq p, \qquad |yv| \geq 1,$$

и

$$xy^i uv^i w \in L$$

за всяко $i \in \mathbb{N}$.
Тъй като всяка от трите еднородни части a^p, b^p, c^p има дължина p, поддумата yuv с дължина най-много p не може да съдържа едновременно буквите a, b и c. Следователно y и v засягат най-много две съседни групи от буквите.
Разглеждаме думата при $i = 2$:

$$xy^2uv^2w.$$

Поне едно от y, v е непразно.
Ако y и v са съставени само от една буква, повторението увеличава броя на поне една от буквите a, b, c, без да увеличава броя и на трите по един и същ начин. Следователно в получената дума броят на a, b и c не е еднакъв, така че тя не принадлежи на L.
Ако някое от y и v съдържа две различни букви, то то пресича границата между две от групите a^p, b^p, c^p. При повтарянето му се нарушава формата

$$a^*b^*c^*.$$

Следователно и в този случай

$$xy^2uv^2w \notin L.$$

Получаваме противоречие с третото условие на лемата за разрастването. Следователно L не е контекстносвободен. □

5.4 Затвореност относно регулярните операции

Конспектът иска *конструкции* за затвореността относно регулярните операции — обединение, конкатенация и звезда на Клини — без доказателства. Конструкциите по-долу работят за произволни входни граматики, а не само за конкретните примери от следващите раздели.

Нека

$$G_1 = \langle V_1, \Sigma, R_1, S_1 \rangle \qquad \text{и} \qquad G_2 = \langle V_2, \Sigma, R_2, S_2 \rangle$$

са контекстносвободни граматики над обща азбука Σ. Преди всяка от конструкциите преименуваме нетерминалите така, че

$$V_1 \cap V_2 = \varnothing,$$

и избираме нов начален нетерминал $S \notin V_1 \cup V_2$. Преименуването не променя породения език: то е просто биекция върху множеството от нетерминали, приложена едновременно към лявата и дясната страна на всяко правило. Без него двете граматики биха си споделили нетерминал и изводите им биха се смесили.

Твърдение 5.10 (Конструкция за обединение)**.** Езикът $L(G_1) \cup L(G_2)$ се поражда от граматиката

$$G_\cup = \langle V_1 \cup V_2 \cup \{S\},\ \Sigma,\ R_1 \cup R_2 \cup \{S \to S_1,\ S \to S_2\},\ S \rangle.$$

Доказателство. Първата стъпка на всеки извод от S е $S \to S_1$ или $S \to S_2$; други правила за S няма. Тъй като $V_1 \cap V_2 = \varnothing$, след избора на S_1 всички достижими нетерминали са от V_1 и приложимите правила са точно тези на R_1, тоест изводът продължава като извод в G_1; аналогично за S_2 и G_2. Следователно $S \Rightarrow^*_{G_\cup} \omega$ точно когато $S_1 \Rightarrow^*_{G_1} \omega$ или $S_2 \Rightarrow^*_{G_2} \omega$. □

Твърдение 5.11 (Конструкция за конкатенация)**.** Езикът $L(G_1)L(G_2) = \{\omega_1\omega_2 \mid \omega_1 \in L(G_1),\ \omega_2 \in L(G_2)\}$ се поражда от граматиката

$$G_\cdot = \langle V_1 \cup V_2 \cup \{S\},\ \Sigma,\ R_1 \cup R_2 \cup \{S \to S_1S_2\},\ S \rangle.$$

Доказателство. Единственото правило за S дава $S \Rightarrow S_1S_2$. Двата нетерминала разполагат с непресичащи се множества нетерминали, затова поддървото над S_1 използва само правила от R_1, а поддървото над S_2 — само правила от R_2. Резултатът е конкатенация на дума, изводима от S_1 в G_1, и дума, изводима от S_2 в G_2; обратно, всяка такава двойка дава извод в $G_\cdot$. □

Твърдение 5.12 (Конструкция за звезда на Клини)**.** Езикът $L(G_1)^*$ се поражда от граматиката

$$G_* = \langle V_1 \cup \{S\},\ \Sigma,\ R_1 \cup \{S \to S_1S,\ S \to \varepsilon\},\ S \rangle.$$

Доказателство. Правилата за S пораждат точно думите S_1^k за $k \in \mathbb{N}$: k пъти се прилага $S \to S_1 S$ и веднъж $S \to \varepsilon$. Всяко от k-те срещания на S_1 извежда независимо дума от $L(G_1)$, тъй като $S \notin V_1$ и правилата на R_1 не могат да въведат S. При $k = 0$ получаваме ε, което принадлежи на $L(G_1)^*$ по определение. □

Забележка 5.13. Трите конструкции обясняват и защо аналогичен подход не работи за сечение: няма как с едно правило да се наложи *една и съща* дума да бъде изведена едновременно в двете граматики. Следващите два раздела показват, че такава конструкция изобщо не съществува.

5.5 Незатвореност относно сечение

Класът на контекстносвободните езици е затворен относно обединение, конкатенация и звездата на Клини — това дадоха конструкциите от предходния раздел. В частност, ако L_1 и L_2 са КСЕ над една и съща азбука, то $L_1 \cup L_2$ е КСЕ. За сечение обаче съответното свойство не е вярно.

Теорема 5.14. Класът на контекстносвободните езици не е затворен относно сечение.

Доказателство. Разглеждаме езиците

$$L_1 = \{a^n b^n c^k \mid n, k \in \mathbb{N}\}$$

и

$$L_2 = \{a^n b^k c^k \mid n, k \in \mathbb{N}\}.$$

И двата езика са контекстносвободни. Например за L_1 може да се използва граматиката с правила

$$S \to AC, \qquad A \to aAb \mid \varepsilon, \qquad C \to cC \mid \varepsilon.$$

Нетерминалът A поражда еднакъв брой букви a и b, а C поражда произволен брой букви c.
Аналогично за L_2 може да се използва граматиката

$$S \to AD, \qquad A \to aA \mid \varepsilon, \qquad D \to bDc \mid \varepsilon.$$

Тук A поражда произволен брой букви a, а D — еднакъв брой букви b и c.
В дума от сечението $L_1 \cap L_2$ броят на a и b трябва да е еднакъв поради принадлежността към L_1, а броят на b и c трябва да е еднакъв поради принадлежността към L_2. Следователно

$$L_1 \cap L_2 = \{a^n b^n c^n \mid n \in \mathbb{N}\}.$$

Но този език не е контекстносвободен според предходния пример. Следователно сечението на два контекстносвободни езика не е непременно контекстносвободно. □

5.6 Незатвореност относно допълнение

Допълнението на език $L \subseteq \Sigma^*$ винаги се разбира относно фиксираната азбука Σ:

$$\overline{L} = \Sigma^* \setminus L.$$

Теорема 5.15. Класът на контекстносвободните езици не е затворен относно допълнение.

Доказателство. Да допуснем противното: нека за всеки контекстносвободен език L и неговото допълнение $\overline{L}$ също е контекстносвободно.
Нека L_1 и L_2 са контекстносвободните езици от доказателството за незатвореност относно сечение. По допускане $\overline{L_1}$ и $\overline{L_2}$ са КСЕ. Тъй като КСЕ са затворени относно обединение,

$$\overline{L_1} \cup \overline{L_2}$$

е КСЕ. Прилагаме отново допускането за затвореност относно допълнение и получаваме, че

$$\overline{\overline{L_1} \cup \overline{L_2}}$$

също е КСЕ.
По закона на де Морган обаче

$$\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2.$$

Това противоречи на вече доказаната незатвореност относно сечение. Следователно класът на контекстносвободните езици не е затворен относно допълнение. □

† **Допълнение 5.16 (Бележка).** От незатвореността относно сечение сама по себе си не следва непосредствено незатвореност относно допълнение. В доказателството е необходимо и известното свойство, че контекстносвободните езици са затворени относно обединение, за да се приложи законът на де Морган.

5.7 Изпитен фокус

- Да се дадат определенията за контекстносвободна граматика, извод, език $L(G)$ и контекстносвободен език.
- Да се обясни връзката между извод, най-ляв извод и дърво на синтактичен анализ.
- Да се дадат конструкциите за затвореност относно обединение, конкатенация и звезда на Клини за *произволни* входни граматики: преименуване на нетерминалите до непресичащи се множества, нов начален нетерминал и правилата $S \to S_1 \mid S_2$, $S \to S_1 S_2$, $S \to S_1 S \mid \varepsilon$. Конспектът иска конструкциите, но не и доказателства към тях.
- Да се формулира точно лемата за разрастването: разлагането $\alpha = xyuvw$, условията $|yv| \geq 1$, $|yuv| \leq p$ и $xy^i uv^i w \in L$ за всяко $i \in \mathbb{N}$.

- Да се възпроизведе доказателствената идея чрез дърво в нормална форма на Чомски: двоично дърво, дълъг път, принцип на Дирихле и повторение на нетерминал.
- Да се обясни защо повтореният нетерминал позволява „напомпване“ на частите y и v.
- Да се докаже с лемата, че $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ не е КСЕ, като се разгледат всички допустими разлагания.
- Да се дадат КСЕ $L_1 = \{a^n b^n c^k\}$ и $L_2 = \{a^n b^k c^k\}$, чието сечение е $\{a^n b^n c^n\}$.
- Да се изведе незатвореността относно допълнение от незатвореността относно сечение чрез затвореността относно обединение и закона на де Морган.

Въпрос 6

Сортиране чрез сравнения във време O(n lg n).

По конспекта: какво трябва да мога

> ★ **Бележка на конспекта:** На изпита ще се падне **или** Heapsort, **или** Mergesort. Това казва какво може да бъде изтеглено, а не какво може да се пропусне при подготовката.

Дефиниция и свойства: двоична пирамида — §6.2, стр. 86.
Доказателство за коректност и сложност по време: наивно построяване на пирамида — §6.3, стр. 87.
Доказателство за коректност и сложност по време: бързо построяване чрез Heapify — §6.4, стр. 89.
Псевдокод, коректност, сложност по време: Heapsort — §6.5, стр. 91.
Псевдокод, коректност, сложност по време: Mergesort като пример за схемата Разделяй и владей — §6.6, стр. 92.
Приложение с доказателство за коректност: броене на инверсии в масив — §6.7, стр. 95.
Литература: [31].

6.1 Сортиране чрез сравнения

Разглеждаме алгоритми, които получават масив от сравними ключове и определят реда им единствено чрез сравнения. Два класически алгоритъма от този тип са Heapsort и Mergesort. И двата работят за време $O(n \lg n)$, но използват различни идеи: първият поддържа структура от данни „двоична пирамида“, а вторият следва схемата *Разделяй и владей*.

В настоящата глава ще използваме индексиране на масивите от 1. Масивът $A[1 \ldots n]$ е сортиран в ненамаляващ ред, когато

$$A[1] \leq A[2] \leq \cdots \leq A[n].$$

6.2 Двоична пирамида

6.2.1 Определение и представяне с масив

Дефиниция 6.1. Двоична пирамида, или *binary heap*, е попълнено двоично дърво с ключове, което удовлетворява едно от следните свойства:

- при *макс-пирамида* ключът на всеки връх е по-голям или равен на ключовете на неговите деца;
- при *мин-пирамида* ключът на всеки връх е по-малък или равен на ключовете на неговите деца.

В тази глава основно използваме макс-пирамиди. Тогава по всеки път от корена към листо стойностите са ненарастващи. Еквивалентно, всяка стойност е не по-малка от стойностите на всички свои потомци.

Дефиниция 6.2. Нека $T = (V, E)$ е попълнено двоично дърво с n върха. Неговата *линеаризация* е масивът $A[1 \dots n]$, в който $A[i]$ е ключът на върха с адрес i.

При това представяне връзките в дървото се определят директно от индексите:

$$\text{Parent}(i) = \begin{cases} \lfloor i/2 \rfloor, & i \geq 2, \\ i, & i = 1, \end{cases}$$

а ако съответните върхове съществуват,

$$\text{Left}(i) = 2i, \qquad \text{Right}(i) = 2i + 1.$$

Връх i е листо точно когато

$$i > \left\lfloor \frac{n}{2} \right\rfloor.$$

Твърдение 6.3. За непразно попълнено двоично дърво с $n \geq 1$ върха, височина в ребра и цяло $k \geq 0$ са изпълнени следните свойства:

1. височината му е $\lfloor \lg n \rfloor$;
2. броят на листата е $\lceil n/2 \rceil$;
3. броят на вътрешните върхове е $\lfloor n/2 \rfloor$;
4. броят на върховете с височина поне k е $\lfloor n/2^k \rfloor$;
5. броят на върховете с височина точно k е

$$\left\lfloor \frac{n}{2^k} \right\rfloor - \left\lfloor \frac{n}{2^{k+1}} \right\rfloor,$$

и в частност е най-много $\lceil n/2^{k+1} \rceil$.

Доказателство. Приемаме $n \geq 1$, височината се брои в ребра, а $k \geq 0$ е цяло число. Номерираме върховете по нива с $1, \ldots, n$. Децата на i имат индекси $2i$ и $2i+1$, когато тези индекси не надхвърлят n. Следователно последният слой е $\lfloor \lg n \rfloor$, а вътрешните върхове са точно $1, \ldots, \lfloor n/2 \rfloor$. Листата са останалите $\lceil n/2 \rceil$. Най-малкият индекс на потомък на i на разстояние k е $2^k i$, затова височината на i е поне k точно когато $i \leq \lfloor n/2^k \rfloor$. Изваждането на броя за $k+1$ дава броя с височина точно k. Ако $n = 2^{k+1}q + r$, $0 \leq r < 2^{k+1}$, тази разлика е $q + \lfloor r/2^k \rfloor \leq q + \mathbf{1}_{r>0} = \lceil n/2^{k+1} \rceil$. □

Забележка 6.4. Свойството на макс-пирамидата не означава, че масивът е сортиран. То сравнява всеки връх само с децата му. Например масивът $[10, 8, 9, 1, 2, 3, 4]$ е линеаризация на макс-пирамида, но не е сортиран в ненамаляващ ред.

6.2.2 Пирамидални инверсии

Дефиниция 6.5. Нека $A[1 \ldots n]$ е масив. *Пирамидална инверсия* е наредена двойка индекси (i, j), за която $i < j$, връх i е предшественик на връх j и

$$A[i] < A[j].$$

Пирамидална инверсия е *директна*, ако $i = \text{Parent}(j)$.

Твърдение 6.6. Масивът $A[1 \ldots n]$ представя макс-пирамида тогава и само тогава, когато няма пирамидални инверсии.

Доказателство. Ако масивът представя макс-пирамида, всеки предшественик има ключ, не по-малък от ключа на всеки свой потомък; следователно пирамидални инверсии няма.
Обратно, ако няма пирамидални инверсии, в частност няма директни пирамидални инверсии. Значи за всеки връх ключът на родителя му е не по-малък от ключа на самия връх. Това е точно свойството на макс-пирамида. □

6.3 Наивно построяване на макс-пирамида

Идеята е елементите да се включват един след друг. Когато добавим новия елемент на позиция i, той е най-дясното листо в попълненото дърво. Ако е по-голям от родителя си, го разменяме с родителя; повтаряме, докато възстановим свойството на пирамидата.

```
NaiveBuildHeap(A[1..n])
    for i <- 2 to n
        k <- i
        while A[Parent(k)] < A[k] do
            swap(A[k], A[Parent(k)])
            k <- Parent(k)
```

Дефиниция 6.7. *Почти-пирамида* е попълнено двоично дърво, чиито ключове удовлетворяват свойството за макс-пирамида, евентуално с изключение на най-дясното листо на последното ниво.

При започване на итерацията с дадено i, подмасивът $A[1 \dots i-1]$ вече е пирамида, а добавянето на $A[i]$ може да наруши свойството единствено по пътя от позиция i към корена.

Лема 6.8. Нека $A[1 \dots i]$ е почти-пирамида. Изпълнението на вътрешния цикъл на NaiveBuildHeap терминира и след него $A[1 \dots i]$ е макс-пирамида.

Доказателство. При всяка итерация на цикъла стойността на k се заменя с

$$\text{Parent}(k) = \left\lfloor \frac{k}{2} \right\rfloor .$$

Ако $k \geq 2$, новата стойност е строго по-малка. Следователно след краен брой стъпки или достигаме корена, или условието на цикъла става лъжа. Значи цикълът терминира.
Първоначално евентуалните пирамидални инверсии съдържат новодобавения връх i и негови предшественици. Ако родителят на текущата позиция k има по-малък ключ, размяната премества по-големия ключ една позиция нагоре. След размяната връзката между стария родител и неговото дете вече удовлетворява свойството на макс-пирамида. Единственото възможно ново нарушение се премества към новата позиция на k, тоест с една стъпка по-близо до корена.
Когато цикълът завърши, текущият ключ или е в корена, или неговият родител е не по-малък от него. По пътя под него свойството е запазено, защото преди размяната поддърветата са били пирамиди, а по-големият ключ се премества нагоре. Следователно не остава пирамидална инверсия и $A[1 \dots i]$ е макс-пирамида. □

Теорема 6.9. За всеки входен масив $A[1 \dots n]$ алгоритъмът NaiveBuildHeap построява макс-пирамида.

Доказателство. Използваме инвариант на външния цикъл:

При всяко достигане на началото на итерация с индекс i, подмасивът $A[1 \dots i-1]$ е макс-пирамида.

При първата итерация $i = 2$, а масивът $A[1]$ с един елемент тривиално е пирамида. Това доказва базата.
Да приемем, че инвариантът е верен в началото на итерация с индекс i. Тогава $A[1 \dots i-1]$ е пирамида. Добавянето на позиция i образува почти-пирамида. По предишната лема вътрешният цикъл превръща $A[1 \dots i]$ в пирамида. След увеличаването на i инвариантът отново е изпълнен.
След последната итерация е построена пирамида върху $A[1 \dots n]$. □

Теорема 6.10. Времевата сложност в най-лошия случай на NaiveBuildHeap е $\Theta(n \lg n)$.

Доказателство. Външният цикъл има $\Theta(n)$ итерации. При итерация с индекс i новият елемент може да се придвижи най-много по височината на дървото $A[1 \dots i]$, която е $O(\lg i)$. Следователно общото време е ограничено от

$$\sum_{i=2}^{n} O(\lg i) = O(\lg(n!)) = O(n \lg n).$$

За долната граница разглеждаме строго нарастващ масив. Всеки нов елемент е по-голям от всички предишни и се изкачва от последната позиция до корена, тоест извършва $\lfloor \lg i \rfloor$ размени. Следователно общият брой размени е

$$\sum_{i=2}^{n} \lfloor \lg i \rfloor = \Omega(n \lg n).$$

Заедно с горната граница получаваме $\Theta(n \lg n)$. □

6.4 Бързо построяване на пирамида

Наивният алгоритъм поправя пирамидата след всяко добавяне. По-бързият метод използва обратната посока: започва от листата, които вече са тривиални пирамиди, и обработва вътрешните върхове отдясно наляво.

6.4.1 Операцията Heapify

Нека за индекс i поддърветата с корени $\text{Left}(i)$ и $\text{Right}(i)$, ако съществуват, вече са макс-пирамиди. Възможно е единствено $A[i]$ да е по-малък от някое свое дете. Операцията Heapify премества този ключ надолу, като на всяка стъпка го разменя с по-голямото му дете.

```
Heapify(A[1..n], i)
    j <- i
    while j <= floor(n / 2) do
        left <- 2 * j
        right <- 2 * j + 1
        if A[left] > A[j]
            largest <- left
        else
            largest <- j
        if right <= n and A[right] > A[largest]
            largest <- right
        if largest != j
            swap(A[j], A[largest])
```

```
            j <- largest
        else
            break
```

Теорема 6.11. Нека поддърветата с корени Left(i) и Right(i), когато съществуват, са макс-пирамиди. След приключване на Heapify поддървото с корен i е макс-пирамида.

Доказателство. Използваме следния инвариант на цикъла:

При всяко достигане на условието на цикъла поддърветата на децата на текущия връх j, ако съществуват, са макс-пирамиди; всички възможни нарушения в поддървото с корен i са съсредоточени в поддървото с корен j.

В началото $j = i$, а предпоставката на теоремата дава, че двете поддървета на i са пирамиди. Следователно инвариантът е изпълнен.
В една итерация алгоритъмът избира най-големия от $A[j]$, A[Left(j)] и A[Right(j)], като несъществуващо дясно дете не се разглежда. Ако най-голям е $A[j]$, свойството на пирамидата е изпълнено в корена j и тъй като двете поддървета са пирамиди, цялото разглеждано поддърво е пирамида.
Иначе най-голямото дете се разменя с $A[j]$. След размяната новият ключ на позиция j е не по-малък от ключовете и на двете деца, така че свойството е възстановено в j. Единственото възможно нарушение е пренесено в поддървото, чийто корен е детето, с което е извършена размяната. Неговите деца не са били променяни и остават корени на пирамиди. Инвариантът се запазва.
При всяка размяна j се премества едно ниво надолу. Следователно цикълът терминира. Ако достигне листо, там няма деца и не е възможно нарушение. Ако приключи чрез `break`, текущият връх е не по-малък от децата си. Във всеки случай няма останало нарушение, затова поддървото с корен i е макс-пирамида. □

Твърдение 6.12. Ако поддървото с корен i съдържа m елемента, Heapify работи за $O(\lg m)$ време; в най-лошия случай времето е $\Theta(\lg m)$.

Доказателство. Във всяка итерация се извършва константен брой сравнения и евентуално една размяна, след която текущият елемент се премества с едно ниво надолу. Дължината на такъв път е ограничена от височината на поддървото, която е $O(\lg m)$. □

6.4.2 Линейно построяване на пирамида

```
BuildHeap(A[1..n])
    for i <- floor(n / 2) downto 1
        Heapify(A, i)
```

Теорема 6.13. Алгоритъмът BuildHeap построява макс-пирамида от масива $A[1 \dots n]$.

Доказателство. Използваме инвариант на цикъла: преди обработването на индекс i, всяко поддърво с корен в позиция j, където $j > i$, е макс-пирамида.
При първото $i = \lfloor n/2 \rfloor$ всички позиции с по-голям индекс са листа, а всяко листо е тривиална пирамида. Това е базата.
Да приемем, че инвариантът е изпълнен преди обработването на i. Ако i е вътрешен връх, неговите деца, когато съществуват, са на позиции по-големи от i и следователно корените им принадлежат на пирамиди. По коректността на Heapify след извикването и поддървото с корен i е пирамида. Така инвариантът се запазва за следващата стойност на i.
Когато цикълът приключи, поддървото с корен 1 е пирамида. То обхваща целия масив. □

Теорема 6.14. Времевата сложност на BuildHeap е $\Theta(n)$.

Доказателство. Не е достатъчно да умножим броя на извикванията на Heapify по $O(\lg n)$, защото повечето върхове са близо до листата и съответните извиквания са кратки.
Броят на върховете с височина k е от порядък $n/2^{k+1}$. За връх с такава височина времето на Heapify е $O(k)$. Следователно общото време е

$$\sum_{k=1}^{\lfloor \lg n \rfloor} O\left(\frac{n}{2^k} k\right) = O\left(n \sum_{k=1}^{\infty} \frac{k}{2^k}\right).$$

Редът

$$\sum_{k=1}^{\infty} \frac{k}{2^k}$$

е сходящ, следователно е ограничен с константа. Общото време е $O(n)$. Тъй като алгоритъмът обхожда $\lfloor n/2 \rfloor$ вътрешни върха, времето е и $\Omega(n)$. Следователно сложността е $\Theta(n)$. □

6.5 Heapsort

Алгоритъмът първо построява макс-пирамида. Най-големият елемент тогава е в корена. Той се разменя с последния елемент на текущата пирамида и се изключва от нея. След това Heapify възстановява свойството на пирамидата върху останалия префикс.

```
Heapsort(A[1..n])
    BuildHeap(A)
    for i <- n downto 2
        swap(A[1], A[i])
        Heapify(A[1..i-1], 1)
```

Теорема 6.15. Heapsort е коректен сортиращ алгоритъм.

Доказателство. Нека $A_0[1 \dots n]$ е първоначалният масив. Използваме следния инвариант на цикъла:

- суфиксът $A[i+1 \dots n]$ съдържа $n-i$-те най-големи елемента на $A_0[1 \dots n]$ в сортиран ред;
- префиксът $A[1 \dots i]$ е макс-пирамида.

Преди първата итерация $i = n$. Суфиксът е празен, а $A[1 \dots n]$ е макс-пирамида по коректността на BuildHeap. Базата е доказана.
Да приемем, че инвариантът е верен за текущо i. Коренът $A[1]$ е най-големият елемент на пирамидата $A[1 \dots i]$. След размяната на $A[1]$ и $A[i]$ той заема позиция i. Следователно $A[i \dots n]$ съдържа $n-i+1$-те най-големи елемента от първоначалния масив в сортиран ред: новодобавеният в началото на този суфикс е не по-голям от всички вече намиращи се вдясно елементи.
Размяната може да наруши свойството на пирамидата само в корена на $A[1 \dots i-1]$. Поддърветата на неговите деца остават пирамиди. По коректността на Heapify след извикването $A[1 \dots i-1]$ отново е макс-пирамида. Инвариантът се запазва.
При завършване на цикъла $i = 1$. Суфиксът $A[2 \dots n]$ съдържа $n-1$-те най-големи елемента в сортиран ред, а оставащият елемент $A[1]$ е най-малкият. Значи целият масив е сортиран. □

Теорема 6.16. Heapsort работи за време $\Theta(n \lg n)$.

Доказателство. Построяването на пирамидата изисква $\Theta(n)$ време. В цикъла се извършват $n-1$ извиквания на Heapify, като извикването върху пирамида с $i-1$ елемента работи за $O(\lg i)$ време. Следователно

$$\sum_{i=2}^{n} O(\lg i) = O(n \lg n).$$

В най-лошия случай сумата е от ред $\Theta(n \lg n)$, а линейното построяване не променя тази оценка. Следователно времето е $\Theta(n \lg n)$. □

Забележка 6.17. При итеративната реализация на Heapify Heapsort използва $\Theta(1)$ допълнителна памет. Алгоритъмът не е стабилен, тъй като размяната на елементи може да промени относителния ред на равни ключове.

6.6 Mergesort и схемата Разделяй и владей

6.6.1 Схемата Разделяй и владей

При схемата *Разделяй и владей* решаването на задача преминава през три фази:

1. *Разделяй*: входът се разделя на по-малки части, ако размерът му е достатъчно голям;
2. *Владей*: всяка част се решава рекурсивно;
3. *Комбинирай*: решенията на подзадачите се обединяват в решение на първоначалната задача.

При Mergesort масивът се разделя на две приблизително равни части. Те се сортират рекурсивно, а после се сливат в една сортирана последователност.

6.6.2 Сливане на два сортирани подмасива

Процедурата Merge приема, че $A[l \dots mid]$ и $A[mid+1 \dots h]$ вече са сортирани. Тя копира тези части във временни масиви L и R. Стойността ∞ е страж в края на всеки временен масив.

```
Merge(A[1..n], l, mid, h)
    n1 <- mid - l + 1
    n2 <- h - mid
    create L[1..n1+1] and R[1..n2+1]
    L[1..n1] <- A[l..mid]
    R[1..n2] <- A[mid+1..h]
    L[n1+1] <- infinity
    R[n2+1] <- infinity
    i <- 1
    j <- 1
    for k <- l to h
        if L[i] <= R[j]
            A[k] <- L[i]
            i <- i + 1
        else
            A[k] <- R[j]
            j <- j + 1
```

Теорема 6.18. Ако $A[l \dots mid]$ и $A[mid+1 \dots h]$ са сортирани, след Merge масивът $A[l \dots h]$ съдържа точно техните елементи в сортиран ред.

Доказателство. Използваме инвариант на цикъла:

- $A[l \dots k-1]$ съдържа $k-l$-те най-малки вече избрани елемента от L и R в сортиран ред;
- $L[i]$ и $R[j]$ са най-малките съответно в L и R елементи, които още не са копирани в A.

В началото $k = l$, префиксът $A[l \dots k-1]$ е празен, а първите елементи на L и R са най-малките неупотребени елементи. Инвариантът е изпълнен.

В произволна итерация не е възможно едновременно $L[i]$ и $R[j]$ да са стражите ∞: иначе всички реални елементи биха били копирани преди да са запълнени всички позиции от l до h. Следователно едно от двете текущи числа е най-малкото още некопирано число. Алгоритъмът записва именно него в $A[k]$ и увеличава съответния индекс. Така новият елемент е не по-малък от предишния, а инвариантът се запазва. При завършване $k = h+1$, затова $A[l \dots h]$ съдържа всички $h-l+1$ елемента от двете входни части в сортиран ред. □

6.6.3 Алгоритъмът Mergesort

```
Mergesort(A[1..n], l, h)
    if l < h
        mid <- floor((l + h) / 2)
        Mergesort(A, l, mid)
        Mergesort(A, mid + 1, h)
        Merge(A, l, mid, h)
```

Теорема 6.19. При начално извикване Mergesort$(A, 1, n)$ алгоритъмът сортира масива $A[1 \dots n]$.

Доказателство. Доказваме по индукция по $h - l$.
При $h - l = 0$ разглежданият подмасив съдържа точно един елемент. Той е сортиран и алгоритъмът не извършва рекурсивни извиквания.
Нека $h - l > 0$ и приемем, че алгоритъмът сортира коректно всички подмасиви с по-малка разлика между границите. Индексът mid разделя $A[l \dots h]$ на два подмасива с по-малък размер. По индукционната хипотеза рекурсивните извиквания сортират $A[l \dots mid]$ и $A[mid + 1 \dots h]$. По коректността на Merge последното извикване слива тези две части в сортиран $A[l \dots h]$.
Следователно началното извикване сортира $A[1 \dots n]$. □

Теорема 6.20. Времевата сложност на Mergesort е $\Theta(n \lg n)$.

Доказателство. За масив с размер n алгоритъмът извършва две рекурсивни извиквания върху части с размер приблизително $n/2$, а сливането отнема $\Theta(n)$ време. Следователно

$$T(n) = 2T(n/2) + \Theta(n).$$

По втория случай на мастер-теоремата

$$T(n) = \Theta(n \lg n).$$

□

Забележка 6.21. Mergesort използва $\Theta(n)$ допълнителна памет за временните масиви при сливането. Алгоритъмът е стабилен: при равни ключове условието $L[i] \leq R[j]$

избира първо елемента от лявата част, която в първоначалния масив предхожда дясната.

6.7 Броене на инверсии чрез Mergesort

Дефиниция 6.22. Инверсия в масив $A[1 \dots n]$ е двойка индекси (i, j), за която

$$1 \leq i < j \leq n \qquad \text{и} \qquad A[i] > A[j].$$

Твърдение 6.23. Масивът $A[1 \dots n]$ е сортиран в ненамаляващ ред тогава и само тогава, когато няма инверсии.

Доказателство. Ако масивът е сортиран, за всеки $i < j$ е изпълнено $A[i] \leq A[j]$, така че няма инверсия. Обратно, ако няма инверсии, в частност няма такава между съседни позиции, тоест $A[i] \leq A[i+1]$ за всяко $i < n$. Това е точно условието за ненамаляващ ред. □

При разделяне на масива на две части всяка инверсия е точно от един от следните три вида:

- и двата й индекса са в лявата част;
- и двата й индекса са в дясната част;
- първият индекс е в лявата, а вторият – в дясната част.

Първите два вида се броят рекурсивно. Третият вид се брои при сливането.

```
InversionCount(A[1..n], l, h)
    if l >= h
        return 0
    mid <- floor((l + h) / 2)
    x <- InversionCount(A, l, mid)
    y <- InversionCount(A, mid + 1, h)
    z <- MergeModified(A, l, mid, h)
    return x + y + z
```

Модифицираното сливане е същото като Merge, но при избиране на елемент от дясната част се добавя броят на всички още некопирани елементи от лявата част.

† **Допълнение 6.24 (Бележка).** В предоставения вариант на псевдокода за MergeModified индексите i и j са инициализирани с 0, въпреки че временните масиви са индексирани от 1. По-долу използваме съгласуваното с Merge индексиране $i \leftarrow 1, j \leftarrow 1$.

```
MergeModified(A[1..n], l, mid, h)
    n1 <- mid - l + 1
    n2 <- h - mid
    create L[1..n1+1] and R[1..n2+1]
    L[1..n1] <- A[l..mid]
    R[1..n2] <- A[mid+1..h]
    L[n1+1] <- infinity
    R[n2+1] <- infinity
    i <- 1
    j <- 1
    c <- 0
    for k <- l to h
        if L[i] <= R[j]
            A[k] <- L[i]
            i <- i + 1
        else
            A[k] <- R[j]
            j <- j + 1
            c <- c + n1 - i + 1
    return c
```

Лема 6.25. При изпълнение на MergeModified стойността c е броят на инверсиите (p, q), за които

$$l \leq p \leq mid < q \leq h.$$

Доказателство. Коректността на сортирането следва по същия инвариант, както при Merge. Остава да се обоснове броячът.
Ако $L[i] \leq R[j]$, тогава $L[i]$ не образува инверсия с $R[j]$. Тъй като R е сортиран, той не образува инверсия и с никой от следващите елементи на R. Затова при избиране отляво не се добавя нищо към c.
Ако $L[i] > R[j]$, то $R[j]$ образува инверсия с $L[i]$. Освен това L е сортиран, така че

$$L[i] \leq L[i+1] \leq \cdots \leq L[n_1].$$

Следователно $R[j]$ образува инверсия с всички още некопирани елементи

$$L[i], L[i+1], \ldots, L[n_1].$$

Техният брой е $n_1 - i + 1$, което е точно добавената стойност. Всяка такава кръстосана инверсия се отчита веднъж: в момента, в който нейният десен елемент се копира преди съответния оставащ ляв елемент. □

Теорема 6.26. InversionCount връща точния брой инверсии в $A[l \ldots h]$.

Доказателство. Доказваме по индукция по размера на подмасива. При $l \geq h$ масивът има най-много един елемент и няма инверсии.

За по-голям подмасив рекурсивните извиквания връщат точния брой инверсии изцяло в лявата и съответно изцяло в дясната половина. По предишната лема MergeModified връща точния брой кръстосани инверсии. Трите множества са непресичащи се и съдържат всички инверсии на $A[l \dots h]$. Следователно сумата $x + y + z$ е търсеният брой. □

Забележка 6.27. Времевата сложност на броенето на инверсии е $\Theta(n \lg n)$: модифицираното сливане запазва линейната сложност на обикновеното сливане, а рекурсивното уравнение остава

$$T(n) = 2T(n/2) + \Theta(n).$$

6.8 Изпитен фокус

- Да се даде определение за попълнено двоично дърво, макс-пирамида и мин-пирамида; да се знае масивното представяне чрез Parent, Left и Right.
- Да се формулират свойствата за височината, листата и вътрешните върхове на попълненото двоично дърво.
- Да се обяснят пирамидалните инверсии и еквивалентността между липсата им и свойството на макс-пирамида.
- Да се възпроизведе NaiveBuildHeap, инвариантът на външния цикъл и идеята, че нов елемент се изкачва по пътя към корена.
- Да се обоснове сложността $\Theta(n \lg n)$ на наивното построяване.
- Да се възпроизведат Heapify и линейният BuildHeap; да се обясни защо Heapify премества единственото възможно нарушение надолу.
- Да се докаже коректността на BuildHeap чрез обхождане от последния вътрешен връх към корена.
- Да се обоснове сумата
$$\sum_{k \geq 1} \frac{nk}{2^k} = O(n)$$
и линейното време на бързото построяване.
- Да се възпроизведе Heapsort, неговият инвариант за пирамидалния префикс и сортирания суфикс, както и оценката $\Theta(n \lg n)$.
- Да се обясни схемата Разделяй и владей и трите й фази.
- Да се възпроизведат Merge и Mergesort; да се докаже Merge чрез инвариант за вече записания сортиран префикс.
- Да се докаже коректността на Mergesort по индукция и да се реши рекурентното уравнение
$$T(n) = 2T(n/2) + \Theta(n).$$

- Да се дефинира инверсия, да се разделят инверсиите на леви, десни и кръстосани и да се обясни добавката $n_1 - i + 1$ в MergeModified.

Въпрос 7

Минимални покриващи дървета.

По конспекта: какво трябва да мога

★ **Бележка на конспекта:** На изпита ще се падне **или** алгоритъмът на Прим, **или** алгоритъмът на Крускал.

Дефиниция на задачата: какъв вид графи се разглеждат и какво се търси — §7.1, стр. 99.
Теорема с доказателство: за съгласуваното множество — условия за нарастване на подмножество на МПД — §7.2, стр. 100.
Псевдокод, коректност, сложност при различни структури от данни: алгоритъм на Прим — §7.3, стр. 102.
Псевдокод и коректност: алгоритъм на Крускал — §7.4, стр. 106.
Структура от данни: Union–Find и приложението му за ефикасна реализация — §7.5, стр. 108.
Сложност по време: на алгоритъма на Крускал — §7.6, стр. 111.
Литература: [31].

7.1 Задача за минимално покриващо дърво

Дефиниция 7.1. Тегловен граф е наредената двойка (G, ω), където $G = (V, E)$ е граф, а

$$\omega : E \to \mathbb{R}$$

е теглова функция, която задава тегло, цена или дължина на всяко ребро.

В задачата за минимално покриващо дърво се разглежда неориентиран свързан тегловен граф. Ориентацията не е част от задачата, защото понятието „дърво“ тук описва свързан ацикличен неориентиран подграф.

Дефиниция 7.2. Нека $G = (V, E)$ е граф. Покриващ подграф на G е всеки подграф от вида

$$G' = (V, E'), \qquad E' \subseteq E.$$

Покриващо дърво на G е покриващ подграф, който е дърво.

Тъй като графът G е свързан, той има поне едно покриващо дърво. Всяко покриващо дърво съдържа всички върхове на G и точно $|V| - 1$ ребра.

Дефиниция 7.3. Нека $G = (V, E)$ е свързан тегловен граф с теглова функция ω, а T е покриващо дърво на G. Тегло на T наричаме сумата

$$\omega(T) = \sum_{e \in E(T)} \omega(e).$$

Покриващото дърво T е *минимално покриващо дърво*, съкратено МПД, ако за всяко покриващо дърво T' на G е изпълнено

$$\omega(T) \leq \omega(T').$$

★ **Дефиниция 7.4.** Задачата за минимално покриващо дърво има следния вид. Екземплярът е наредена двойка $\langle G, \omega \rangle$, където $G = (V, E)$ е неориентиран свързан граф, а $\omega : E \to \mathbb{R}$ е теглова функция.
Решението е минимално покриващо дърво T на G.

Теглата не е необходимо да са положителни и не е необходимо да са различни. При равни тегла могат да съществуват различни минимални покриващи дървета.

7.2 Съгласувани множества и безопасни ребра

Алчните алгоритми за МПД изграждат множество от ребра A, което постепенно се разширява. Основният въпрос е кога към A може безопасно да се добави ново ребро.

Дефиниция 7.5. Нека $G = (V, E)$ е тегловен граф и $A \subseteq E$. Казваме, че A е *съгласувано* множество, ако съществува минимално покриващо дърво T на G, за което

$$A \subseteq E(T).$$

Дефиниция 7.6. Нека $A \subseteq E$ е съгласувано множество и $e \in E \setminus A$. Реброто e е *сигурно* за A, ако съществува минимално покриващо дърво T' на G, за което

$$A \cup \{e\} \subseteq E(T').$$

Дефиниция 7.7. Срез в графа $G = (V, E)$ е разделяне на множеството от върхове на две непразни части:

$$S = \{V_1, V_2\}, \qquad V_1 \cap V_2 = \varnothing, \qquad V_1 \cup V_2 = V.$$

Ребро $e = \{u, v\}$ *пресича* среза S, ако единият му край е във V_1, а другият — във V_2. Срезът S е *съобразен* с множество $A \subseteq E$, ако нито едно ребро от A не пресича S. Ребро, пресичащо среза, е *леко* за този срез, ако теглото му е минимално измежду теглата на всички ребра, които пресичат среза.

★ **Теорема 7.8 (Теорема за съгласуваното множество).** Нека $G = (V, E)$ е свързан тегловен граф с теглова функция ω. Нека $A \subseteq E$ е съгласувано множество и S е срез, съобразен с A. Ако e е произволно леко ребро, което пресича S, то e е сигурно за A.

Доказателство. Тъй като A е съгласувано, съществува минимално покриващо дърво T, за което

$$A \subseteq E(T).$$

Ако $e \in E(T)$, твърдението следва веднага.
Нека сега $e \notin E(T)$. Понеже T е дърво, между двата края на e има точно един път в T. След добавяне на e към T се получава точно един цикъл C.
Реброто e пресича среза S. Следователно, при обхождането на цикъла C трябва да се премине поне още веднъж от едната част на среза в другата. Значи в C съществува ребро

$$e' \neq e,$$

което също пресича S.
Тъй като срезът е съобразен с A, реброто e' не принадлежи на A. Премахваме e' от цикъла C и получаваме графа

$$T' = T - \{e'\} + \{e\}.$$

Това е покриващо дърво: след премахването на едно ребро от единствения цикъл графът остава свързан и става ацикличен. Освен това

$$A \cup \{e\} \subseteq E(T'),$$

защото $e' \notin A$.
Тъй като e е леко ребро за среза, а e' също пресича този срез, имаме

$$\omega(e) \leq \omega(e').$$

Следователно

$$\omega(T') = \omega(T) - \omega(e') + \omega(e) \leq \omega(T).$$

Но T е минимално покриващо дърво, следователно $\omega(T) \leq \omega(T')$. Оттук

$$\omega(T') = \omega(T),$$

така че T' също е минимално покриващо дърво. Значи e е сигурно за A. □

Теоремата обосновава общия алчен подход:

```
Generic_MST(G = (V,E), omega)
A <- empty set                  // избраните ребра; винаги гора
while |A| < |V| - 1 do              // "A още не е покриващо дърво"
    избери ребро e, сигурно за A
    A <- A union {e}
return A
```

Условието на цикъла е записано чрез броя ребра, защото гора върху $|V|$ върха е покриващо дърво тогава и само тогава, когато има точно $|V| - 1$ ребра.

Ако на всяка стъпка A е съгласувано и се добавя сигурно ребро, то след добавянето множеството остава съгласувано. Когато то вече образува покриващо дърво, това дърво трябва да е минимално.

Схемата не казва как се намира сигурно ребро — това е единствената разлика между двата алгоритъма по-долу. Прим държи един срез между построеното дърво и останалите върхове и взема леко ребро за него; Крускал разглежда ребрата в ненамаляващ ред и за всяко прието ребро подходящият срез се намира впоследствие.

7.3 Алгоритъм на Прим

Алгоритъмът на Прим строи едно дърво, започвайки от избран начален връх. На всяка стъпка той добавя най-лекото налично ребро, което свързва вече построената част на дървото с връх извън нея.

7.3.1 Идея и поддържани данни

Нека $r \in V$ е начален връх. Поддържа се приоритетна опашка Q от върховете, които все още не са добавени в дървото. За всеки връх v се пазят две полета:

- $v.key$ — теглото на най-лекото намерено досега ребро, което свързва v с вече добавен връх; това е приоритетът на v в опашката;
- $v.\pi$ — другият край на същото това ребро, тоест предшественикът на v в строящото се дърво.

Ако още не е намерено ребро от дървото към v, тогава $v.key = \infty$ и $v.\pi = \text{Nil}$. За началния връх се поставя ключ 0, за да бъде избран първи.

Дървото никъде не се пази явно. То се пази разпределено, в указателите π: щом връх бъде извлечен от опашката, реброто $\{v, v.\pi\}$ е окончателно избрано и вече не се променя.

★ **Дефиниция 7.9.** Нека $X = V \setminus Q$ е множеството от вече извлечените от опашката върхове. Връх от Q е граничен, ако има съсед в X; останалите върхове от Q са неизвестни. Граничните върхове имат краен ключ, а неизвестните — ключ ∞.

Върховете от X са точно тези, за които реброто към дървото е избрано окончателно; Extract-Min винаги избира граничен връх, докато Q съдържа поне един такъв, тъй като графът е свързан.

7.3.2 Използвани операции

Псевдокодът извиква операции на приоритетна опашка. Те не са част от алгоритъма на Прим: всяка реализация, поддържаща ги, е допустима, а изборът на реализация влияе само на сложността.

Дефиниция 7.10 (Приоритетна опашка по минимум)**.** Приоритетна опашка е структура от данни, която пази множество от елементи, всеки с числов ключ, и поддържа операциите:

- `Insert(Q,x)` — добавя елемента x с текущия му ключ;
- `Extract-Min(Q)` — премахва от Q и връща елемент с минимален ключ;
- `Decrease-Key(Q,x,k)` — заменя ключа на x с новата стойност k, която не е по-голяма от старата, и възстановява вътрешния ред на структурата.

В псевдокода се използват още две означения:

- $Q \leftarrow V$ — построяване на опашка от всички върхове с текущите им ключове, тоест $|V|$ операции `Insert` или еднократно построяване;
- $Adj[u]$ — списъкът на съседство на u, тоест списъкът на върховете, съседни на u.

7.3.3 Псевдокод

```
MST_Prim(G = (V,E), omega, r)
// 1. никой връх още не е свързан с дървото
for each v in V do
    v.key <- infinity        // най-лекото ребро от дървото до v
    v.pi  <- NIL             // другият край на това ребро
r.key <- 0                   // за да бъде r извлечен пръв
Q <- V                       // всички върхове чакат в опашката
// 2. всяка итерация добавя по един връх към дървото
while Q is not empty do
    u <- Extract-Min(Q)      // най-евтино присъединимият връх
    for each v in Adj[u] do  // ребрата от u към съседите му
        if v in Q and omega({u,v}) < v.key then
            v.pi  <- u       // u е новият най-евтин вход към v
            v.key <- omega({u,v})
            Decrease-Key(Q, v, v.key)
// 3. дървото се чете от указателите pi
return {{v, v.pi} : v in V and v.pi != NIL}
```

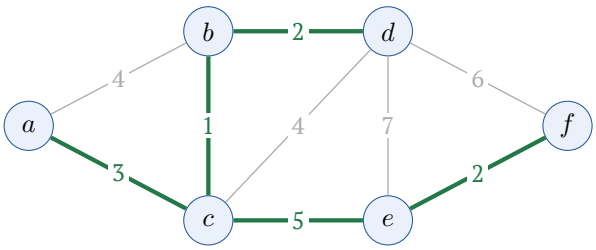
Означенията в псевдокода са следните.

- $G = (V, E)$, ω и r са входът: свързан неориентиран граф, теглова функция и начален връх.
- Q е приоритетната опашка на върховете, които още не са в дървото; $V \setminus Q$ са вече добавените върхове.
- Тялото на `while` се изпълнява по веднъж за всеки връх, защото `Extract-Min` премахва връх и нищо не се връща обратно в Q.
- Върнатото множество съдържа по едно ребро $\{v, v.\pi\}$ за всеки връх, различен от r; само за r остава $v.\pi = \text{Nil}$.

В неориентиран граф всяко ребро се среща в списъците на съседство на двата си края. Проверката $v \in Q$ може да се реализира за константно време с отделна булева или битова информация дали върхът е в опашката.

7.3.4 Пример

▷ **Пример 7.11.** Разглеждаме графа от фигура 7.1, започвайки от връх a.



Фигура 7.1: Примерен тегловен граф. Удебелените ребра образуват минималното покриващо дърво с тегло 13.

След инициализацията $a.key = 0$, а всички останали ключове са ∞. Всеки ред от таблицата отговаря на една итерация на цикъла `while`: първо е извлеченият връх и реброто, което той внася в дървото, а после ключовете, с които останалите върхове чакат в Q.

Извлечен u	Ребро $\{u, u.\pi\}$	Тегло	Ключове в Q след обхождането
a	няма		b:4(a), c:3(a), d:∞, e:∞, f:∞
c	$\{a, c\}$	3	b:1(c), d:4(c), e:5(c), f:∞
b	$\{c, b\}$	1	d:2(b), e:5(c), f:∞
d	$\{b, d\}$	2	e:5(c), f:6(d)
e	$\{c, e\}$	5	f:2(e)
f	$\{e, f\}$	2	$Q = \varnothing$

Забележете двете места, на които `Decrease-Key` променя вече намерено ребро: ключът на b пада от 4(a) на 1(c) и ключът на f пада от 6(d) на 2(e). Реброто $\{d, e\}$ с тегло 7 никога не подобрява ключа на e, а $\{a, b\}$ и $\{c, d\}$ се губят при пресмятанията на съответните ключове.
Полученото дърво е

$$\{a, c\},\ \{c, b\},\ \{b, d\},\ \{c, e\},\ \{e, f\},$$

с общо тегло $3 + 1 + 2 + 5 + 2 = 13$.

7.3.5 Коректност

★ **Теорема 7.12.** За всеки свързан претеглен неориентиран граф алгоритъмът на Прим връща минимално покриващо дърво.

Доказателство. Нека A е множеството от ребрата, определени чрез указателите π за върховете, извлечени преди текущата итерация, без началния връх. Ще покажем по индукция по итерациите на цикъла, че A е съгласувано множество.
Преди първата итерация $A = \varnothing$. Празното множество е съгласувано, защото свързаният претеглен неориентиран граф има минимално покриващо дърво.
Да предположим, че преди дадена итерация A е съгласувано, и нека u е върхът, избран с `Extract-Min`. Ако $u = r$, не се добавя ребро, A остава празно и инвариантът се запазва. В останалите итерации $u \neq r$. Нека преди извличането на u

$$X = V \setminus Q$$

е непразното множество от върхове, вече добавени към построяваното дърво. Реброто

$$e = \{u, u.\pi\}$$

свързва $u \in V \setminus X$ с връх от X.
Разглеждаме среза

$$S = \{X, V \setminus X\}.$$

Той е съобразен с A, защото всички ребра от A имат и двата си края в X. По време на обхожданията на списъците на съседство алгоритъмът поддържа за всеки връх $v \in Q$ теглото на най-лекото ребро между v и X. Следователно изборът на u с минимален ключ означава, че $\{u, u.\pi\}$ пресича среза S и е леко ребро за него.
По теоремата за съгласуваното множество реброто e е сигурно за A. Следователно

$$A \cup \{e\}$$

също е съгласувано. Индуктивната хипотеза се запазва.
След завършване на алгоритъма всички върхове са извлечени. Всеки връх, различен от r, има точно един предшественик, следователно върнатото множество съдържа $|V| - 1$ ребра. Всяко ново ребро свързва нов връх с вече построената част, затова не може да образува цикъл и всички върхове са свързани. Следователно резултатът е покриващо дърво. Понеже множеството от неговите ребра е съгласувано, то се съдържа в някое минимално покриващо дърво; а и двете дървета имат по $|V| - 1$ ребра. Значи върнатото дърво е минимално покриващо. □

7.3.6 Сложност на алгоритъма на Прим

Нека $n = |V|$ и $m = |E|$. При списъци на съседство всички обхождания на редовете с `for each` разглеждат общо $O(m)$ съседства. Всеки връх се извлича от приоритетната

опашка точно веднъж. Операцията `Decrease-Key` се извиква най-много по веднъж за всяко разгледано съседство, следователно най-много $O(m)$ пъти.

Твърдение 7.13. При реализация на Q с двоична пирамида алгоритъмът на Прим работи за време

$$O((n+m)\log n) = O(m \log n)$$

за свързан граф.

Доказателство. Построяването на двоичната пирамида от всички върхове отнема $O(n)$ време. Има n операции `Extract-Min`, всяка за $O(\log n)$ време, общо $O(n \log n)$. Операцията `Decrease-Key` също отнема $O(\log n)$ и се извиква най-много $O(m)$ пъти, което дава $O(m \log n)$.
Следователно общото време е

$$O(n + m + n \log n + m \log n) = O((n+m)\log n).$$

Понеже при свързан граф $m \geq n - 1$, получаваме $O(m \log n)$. □

Твърдение 7.14. При реализация на Q с пирамида на Фибоначи алгоритъмът на Прим работи за амортизирано време

$$O(m + n \log n).$$

Доказателство. При пирамида на Фибоначи операцията `Decrease-Key` е с амортизирана цена $O(1)$, а `Extract-Min` — с амортизирана цена $O(\log n)$. Затова $O(m)$ намалявания на ключа струват $O(m)$, а n извличания на минимум струват $O(n \log n)$. □

Забележка 7.15. При двоична пирамида често се записва и оценката $O(m \log m)$, когато логаритъмът се изразява чрез броя на ребрата. За прост неориентиран граф $m < n^2$, откъдето $\log m = O(\log n)$; така двете оценки са съвместими в рамките на обичайните предположения за графа.

7.4 Алгоритъм на Крускал

Алгоритъмът на Крускал строи не едно растящо дърво, а растяща гора. Ребрата се разглеждат в ненамаляващ ред по тегло. Ребро се добавя точно когато краищата му принадлежат на различни дървета в текущата гора.

7.4.1 Използвани операции

Псевдокодът извиква три операции върху разбиване на множеството от върхове на непресичащи се дялове. Всеки дял е една компонента на свързаност на текущата гора (V, A).

- `Make-Set(v)` — създава нов дял, съдържащ само v;

- `Find-Set(v)` — връща представителя (лидера) на дяла, съдържащ v; два върха са в един и същ дял тогава и само тогава, когато `Find-Set` връща един и същ представител за тях;
- `Union(u,v)` — слива дяловете на u и v в един.

Заедно те образуват структурата Union–Find, разгледана подробно след доказателството за коректност. Засега е достатъчна само тази спецификация: нейната реализация не влияе на коректността на Крускал, а само на сложността.

7.4.2 Псевдокод

```
MST_Kruskal(G = (V,E), omega)
A <- empty set                    // избраните ребра
for each v in V do
    Make-Set(v)                   // всеки връх е отделен дял
sort E in nondecreasing order by omega(e)
for each edge {u,v} in E do       // в този сортиран ред
    if Find-Set(u) != Find-Set(v) then    // различни компоненти
        A <- A union {{u,v}}      // не затваря цикъл: приема се
        Union(u,v)                // компонентите се сливат
return A
```

Проверката `Find-Set(u) != Find-Set(v)` е точно проверката дали реброто $\{u, v\}$ би затворило цикъл: цикъл се получава тогава и само тогава, когато двата края вече са свързани чрез приетите ребра. Затова липсва отделна проверка за цикъл — дяловете я вършат.

При свързан граф цикълът приема точно $|V| - 1$ ребра, така че изпълнението може да спре веднага щом $|A| = |V| - 1$; останалите ребра неизбежно биха затворили цикъл.

7.4.3 Пример

▷ **Пример 7.16.** За графа от фигура 7.1 сортираният ред на ребрата е

$$\{b,c\}{:}1,\ \{b,d\}{:}2,\ \{e,f\}{:}2,\ \{a,c\}{:}3,\ \{a,b\}{:}4,\ \{c,d\}{:}4,\ \{c,e\}{:}5,\ \{d,f\}{:}6,\ \{d,e\}{:}7.$$

Ребро	Тегло	Действие	Компоненти след стъпката
$\{b,c\}$	1	приема се	$\{a\}$, $\{b,c\}$, $\{d\}$, $\{e\}$, $\{f\}$
$\{b,d\}$	2	приема се	$\{a\}$, $\{b,c,d\}$, $\{e\}$, $\{f\}$
$\{e,f\}$	2	приема се	$\{a\}$, $\{b,c,d\}$, $\{e,f\}$
$\{a,c\}$	3	приема се	$\{a,b,c,d\}$, $\{e,f\}$
$\{a,b\}$	4	отхвърля се	без промяна
$\{c,d\}$	4	отхвърля се	без промяна
$\{c,e\}$	5	приема се	$\{a,b,c,d,e,f\}$

Приети са $5 = |V| - 1$ ребра, така че ребрата $\{d, f\}$ и $\{d, e\}$ вече не се разглеждат. Полученото дърво съвпада с намереното от Прим и има тегло 13, макар че двата алгоритъма добавят ребрата в различен ред.

7.4.4 Коректност

★ **Теорема 7.17.** Алгоритъмът на Крускал връща минимално покриващо дърво.

Доказателство. В началото $A = \varnothing$, следователно A е съгласувано. Нека разгледаме стъпка, в която алгоритъмът приема реброто

$$e = \{u, v\}.$$

Условието

$$\text{Find-Set}(u) \neq \text{Find-Set}(v)$$

означава, че u и v са в различни компоненти на гората (V, A). Нека C е компонентата, съдържаща u, и разгледаме среза

$$S = \{C, V \setminus C\}.$$

Този срез е съобразен с A, тъй като по определение няма ребро от A, което да свързва различни компоненти на гората.
Реброто e пресича среза. То е леко за този срез: ако имаше пресичащо ребро с по-малко тегло, то би се намирало по-рано в сортирания ред. Когато то е било разглеждано, краищата му са били в различни компоненти, защото компонентите могат само да се сливат, а не да се разделят. Следователно това по-леко ребро би било добавено в A, което противоречи на съобразеността на среза с A.
От теоремата за съгласуваното множество следва, че e е сигурно за A. Следователно след всяко прието ребро множеството A остава съгласувано.
Алгоритъмът никога не образува цикъл, понеже приема ребро само между различни компоненти. Тъй като графът е свързан, след разглеждане на всички ребра всички върхове принадлежат на една компонента. Тогава A е свързан и ацикличен граф, тоест покриващо дърво. Понеже A е съгласувано и само е покриващо дърво, то е минимално покриващо дърво. □

7.5 Union–Find структури от данни

За ефикасната реализация на Крускал е необходима структура от данни, която поддържа разбиване на множество на непразни непripокриващи се дялове.

Дефиниция 7.18. Union–Find структура върху опорно множество S поддържа следните операции:

- `Make-Set(x)` създава едноелементния дял $\{x\}$;
- `Find-Set(x)` връща представител, наричан лидер, на дяла, съдържащ x;
- `Union(x,y)` слива двата дяла, съдържащи x и y.

В алгоритъма на Крускал всяка компонента на текущата гора е един дял. Така проверката дали реброто $\{u, v\}$ би образувало цикъл е проверката

$$\text{Find-Set}(u) \neq \text{Find-Set}(v).$$

Ако е вярна, реброто се приема и се изпълнява `Union(u,v)`.

7.5.1 Реализация с гора от коренови дървета

Разбиването може да се представи чрез ориентирана гора. Върховете на гората са елементите на S, а всяко дърво съответства на един дял. Коренът на дървото е лидерът на съответния дял. Нека $\pi[x]$ е родителят на x; за корен е изпълнено $\pi[x] = x$. Тази гора няма нищо общо с гората, която Крускал строи — тя само записва кои върхове са в една компонента, а не кои ребра са приети.

Полето *rank*$[x]$ по-долу е горна граница за височината на дървото с корен x и се използва само от евристиката *union by rank*; при `Make-Set` то е 0.

```
Make-Set(x)
    pi[x] <- x
    rank[x] <- 0

Find-Set(x)
    if x != pi[x] then
        return Find-Set(pi[x])
    else
        return x
```

Без допълнителни правила операцията `Union` може да образува дърво с линейна височина. Тогава `Find-Set` би изисквала линейно време в най-лошия случай.

7.5.2 Union by rank

Дефиниция 7.19 (Обединение по ранг)**.** При евристиката *union by rank* коренът на дървото с по-малък ранг се закача към корена на дървото с по-голям ранг. При равни рангове единият корен се избира за нов корен и неговият ранг се увеличава с единица.

```
Union(x,y)
    rx <- Find-Set(x)              // корените, а не самите x и y
    ry <- Find-Set(y)
    if rx = ry then
        return
    if rank[rx] < rank[ry] then
        pi[rx] <- ry
    else if rank[rx] > rank[ry] then
        pi[ry] <- rx
    else
        pi[ry] <- rx
        rank[rx] <- rank[rx] + 1
```

Лема 7.20. Започвайки от едноелементни множества, след произволна последователност от операции `Union` с union by rank височината на всяко дърво е $O(\log n)$, където n е броят върхове в дървото.

Доказателство. Поддържаме едновременно инвариантите, че височината на дървото е най-много ранга на корена и че дърво с ранг h има поне 2^h върха. Те са верни за едноелементно дърво: височината и рангът са 0, а броят върхове е 1.
Ако корен с по-малък ранг се закачи към корен с по-голям ранг, новата височина е най-много по-големия ранг: височината на закаченото дърво плюс едно е най-много този ранг. Рангът на новия корен не се променя. При равни рангове h новата височина е най-много $h+1$, а рангът на новия корен също става $h+1$.
За втория инвариант е достатъчно да разгледаме момента, в който рангът се увеличава от h на $h+1$. Тогава се сливат две дървета с ранг h, всяко с поне 2^h върха. Новото дърво има поне

$$2^h + 2^h = 2^{h+1}$$

върха.
Следователно, ако дърво с n върха има ранг h, то

$$2^h \leq n,$$

тоест

$$h \leq \log_2 n.$$

Понеже височината е най-много ранга, тя също е най-много $\log_2 n$. Затова `Find-Set` работи за $O(\log n)$ време. □

7.5.3 Компресия на пътя

Дефиниция 7.21 (Компресия на пътя)**.** Втората евристика е *компресия на пътя*. При изпълнение на `Find-Set(x)` всички върхове по намерения път към корена се насочват непосредствено към корена.

```
Find-Set(x)
```

```
if x != pi[x] then
    pi[x] <- Find-Set(pi[x])
return pi[x]
```

При едновременно използване на union by rank и компресия на пътя общата цена на последователност от `Make-Set`, `Find-Set` и `Union` операции е

$$O\big((n+m)\alpha(n)\big),$$

където α е обратната функция на Акерман. Тя расте изключително бавно и за практически размери има много малка стойност.

7.6 Сложност на алгоритъма на Крускал

Нека $n = |V|$ и $m = |E|$.

Инициализирането с n операции `Make-Set` отнема $O(n)$ време. Сортирането на ребрата е с време

$$O(m \log m).$$

При реализация с union by rank и компресия на пътя операциите на Union–Find за целия алгоритъм струват

$$O\big((n+m)\alpha(n)\big).$$

Следователно общата времева сложност е

$$O\big(m \log m + (n+m)\alpha(n)\big).$$

За прост неориентиран граф е изпълнено $m < n^2$, откъдето

$$\log m = O(\log n).$$

Сортирането доминира останалата работа, така че сложността на алгоритъма на Крускал обичайно се записва като

$$O(m \log m) = O(m \log n).$$

† **Допълнение 7.22 (Бележка).** Ако се използва само union by rank, без компресия на пътя, операциите `Find-Set` са $O(\log n)$, а алгоритъмът на Крускал отново има оценка $O(m \log n)$ поради сортирането. Компресията на пътя подобрява цената на операциите върху разбиването, макар че не променя доминиращата асимптотична оценка на Крускал при сравняващо сортиране.

7.7 Изпитен фокус

- Да се дефинират тегловен граф, покриващо дърво, тегло на дърво и минимално покриващо дърво.
- Да се формулира задачата: неориентиран свързан граф с тегла на ребрата; търси се покриващо дърво с минимална сума от теглата.

- Да се дефинират съгласувано множество, сигурно ребро, срез, съобразен срез и леко ребро.
- Да се формулира и докаже теоремата за съгласуваното множество чрез добавяне на ребро, получаване на цикъл и размяна с друго ребро от същия срез.
- Да се възпроизведе псевдокодът на Прим, значението на key, π и приоритетната опашка, както и операциите `Extract-Min` и `Decrease-Key`, които той използва.
- Да се обясни коректността на Прим чрез среза между вече избраните и неизбраните върхове и теоремата за съгласуваното множество.
- Да се изведат сложностите на Прим: $O(m \log n)$ с двоична пирамида и $O(m + n \log n)$ с пирамида на Фибоначи.
- Да се възпроизведе псевдокодът на Крускал и ролята на сортирането на ребрата.
- Да се обясни коректността на Крускал: прието ребро свързва различни компоненти и е леко за подходящ съобразен срез.
- Да се дефинират `Make-Set`, `Find-Set` и `Union`, както и реализацията им чрез гора от коренови дървета.
- Да се обяснят union by rank и компресията на пътя и приложението им в Крускал.
- Да се изведе сложността на Крускал:

$$O\big(m \log m + (n + m)\alpha(n)\big) = O(m \log n).$$

Въпрос 8

Най-къси пътища в тегловни графи.

По конспекта: какво трябва да мога

★ **Бележка на конспекта:** На изпита ще се падне **или** алгоритъмът на Дийкстра *и* алгоритъмът за тегловен ацикличен граф, **или** алгоритъмът на Белман–Форд *и* алгоритъмът на Флойд–Уоршал. Двойките са неразделни.

Определение и варианти на задачата: плюс потенциалните проблеми при отрицателни тегла — §8.1, стр. 114.
Вариант, псевдокод, коректност, сложност при различни структури: алгоритъм на Дийкстра — §8.3, стр. 117.
Вариант, псевдокод, коректност, сложност, предимства пред Дийкстра: най-къси пътища в тегловен ориентиран ацикличен граф — §8.4, стр. 121.
Вариант, псевдокод, коректност, сложност, откриване на отрицателен цикъл: алгоритъм на Белман–Форд — §8.5, стр. 123.
Вариант, псевдокод, коректност, сложност по време и по памет: алгоритъм на Флойд–Уоршал — §8.6, стр. 126.
Литература: [31].

8.1 Постановка на задачата

Дефиниция 8.1 (Тегловен граф и тегло на маршрут)**.** Нека $G = (V, E)$ е тегловен граф с теглова функция

$$\omega : E \to \mathbb{R}.$$

Разглеждаме преди всичко ориентирани графи. Теглото на маршрут $p = (v_0, e_1, v_1, \ldots, e_k, v_k)$ е сумата от теглата на ребрата му, броени с повторенията:

$$\omega(p) = \sum_{i=1}^{k} \omega(e_i).$$

Дефиниция 8.2. Нека $u, v \in V$. С $\text{Paths}(u, v)$ означаваме множеството от пътища от u до v. Най-късото разстояние от u до v се означава с $\delta(u, v)$ и е

$$\delta(u,v) = \begin{cases} +\infty, & \text{ако } \text{Paths}(u,v) = \emptyset, \\ -\infty, & \text{ако е изпълнено } C(u,v), \\ \min\{\omega(p) \mid p \in \text{Paths}(u,v)\}, & \text{иначе.} \end{cases}$$

Тук $C(u, v)$ означава, че има маршрут от u до v, минаващ през отрицателен цикъл. Допускат се повторения на върхове и ребра; при липса на такъв цикъл минимумът се достига от прост път. Под *най-къс път* ще разбираме път с най-малко тегло.

Дефиниция 8.3. В ориентиран тегловен граф *отрицателен цикъл* е цикъл, чиято сума от теглата на ребрата е отрицателна.

† **Допълнение 8.4 (Бележка).** При неориентиран граф всяко ребро с отрицателно тегло позволява обхождане в двете посоки с отрицателно общо тегло. Следователно наличието на отрицателно ребро е съществено препятствие за стандартното понятие за най-къс път, дори ако се разглеждат само прости цикли.

8.1.1 Варианти на задачата

Основните варианти на задачата за най-къси пътища са следните.

- *Най-къс път между дадена двойка върхове* (*single-pair shortest paths*). Дадени са G, ω, начален връх s и краен връх t. Търси се $\delta(s, t)$.
- *Най-къси пътища от един източник* (*single-source shortest paths*, SSSP). Дадени са G, ω и начален връх s. Търсят се всички стойности

$$\delta(s, v), \qquad v \in V.$$

- *Най-къси пътища до един краен връх* (*single-destination shortest paths*). Дадени са G, ω и краен връх t. Търсят се всички стойности

$$\delta(v, t), \qquad v \in V.$$

- *Най-къси пътища между всички двойки върхове* (*all-pairs shortest paths*, APSP). Дадени са G и ω. Търсят се всички стойности

$$\delta(u, v), \qquad u, v \in V.$$

Задачата с един краен връх може да се свежда до задача с един източник чрез обръщане на всички ребра. Задачата за една двойка е частен случай на задачата с един източник, но специализиран алгоритъм може да прекрати работата си по-рано, когато крайният връх е достатъчно обработен.

8.1.2 Отрицателни тегла и оптимална подструктура

Отрицателните тегла на ребрата сами по себе си не правят задачата безсмислена. Проблемът възниква при отрицателен цикъл, който е достижим по път от началния до крайния връх. Тогава цикълът може да се обходи произволно много пъти и да се получават пътища с все по-малко тегло. Поради това няма път с минимално тегло и стойността е $-\infty$.

Забележка 8.5. Ако от началния връх не е достижим отрицателен цикъл, за всеки достижим краен връх може да се избере прост най-къс път. Наистина, ако най-къс маршрут съдържа цикъл с неотрицателно тегло, премахването му не увеличава теглото; отрицателен цикъл по маршрута би бил достижим от началния връх. При цикъл с нулево тегло може да има и непрости най-къси маршрути, но винаги съществува прост представител със същото тегло.

Теорема 8.6 (Оптимална подструктура на най-късите пътища)**.** Нека p е най-къс път от s до t и нека u предхожда v по p. Тогава подпътят на p от u до v е най-къс път от u до v.

Доказателство. Нека подпътят от u до v в p е q. Ако съществува път q' от u до v с

$$\omega(q') < \omega(q),$$

то можем да заменим q с q' в пътя p. Полученият път от s до t има тегло

$$\omega(p) - \omega(q) + \omega(q') < \omega(p),$$

което противоречи на избора на p. □

8.2 Релаксация на ребра

Алгоритмите за най-къси пътища поддържат за всеки връх v две полета:

- оценка $v.d$ за разстоянието от началния връх s до v; тя е тегло на някакъв вече намерен маршрут от s до v, тоест горна граница за $\delta(s, v)$;
- предшественик $v.\pi$ — върхът, от който този маршрут влиза в v. Пътят се възстановява, като се тръгне от v и се следват указателите π назад до s.

Използват се още две означения: $w((x,y))$ е теглото на реброто (x,y), а $adj(x)$ е списъкът на съседство на x, тоест списъкът на върховете y, за които $(x,y) \in E$.

Инициализацията е следната; името `ISS` идва от *initialize single source*.

```
ISS(G, s)
    for each v in V(G)
        v.d  <- infinity       // още не е намерен маршрут до v
        v.pi <- Nil            // предшественик по този маршрут
    s.d <- 0                   // до източника води празният маршрут
```

Основната операция е *релаксация*.

```
Relax((x, y))
    if y.d > x.d + w((x, y))   // през x се стига по-евтино до y
        y.d  <- x.d + w((x, y))  // оценката се подобрява
        y.pi <- x                // и се запомня как е постигната
```

Тя проверява дали вече известният път до x, последван от реброто (x,y), подобрява текущата оценка за y. Релаксацията никога не увеличава оценка и никога не поставя стойност, която не е тегло на реален маршрут; сравнението със сума, съдържаща $x.d = +\infty$, е винаги лъжа, така че недостижимите досега върхове не се променят.

Всички алгоритми по-долу се различават само по това *в какъв ред* релаксират ребрата; самата операция е една и съща.

Лема 8.7 (Неравенство за ребро)**.** За всяко ребро $(x,y) \in E$ е изпълнено

$$\delta(s,y) \leq \delta(s,x) + \omega((x,y)).$$

Доказателство. Ако x не е достижим от s, дясната страна е $+\infty$ и неравенството е вярно. Ако до x може да се достигне чрез път, съдържащ отрицателен цикъл, тогава и до y може да се достигне по такъв път и и двете релевантни стойности са $-\infty$.
В останалия случай съществува най-къс път от s до x. Добавянето на реброто (x,y) дава път от s до y с тегло

$$\delta(s,x) + \omega((x,y)).$$

По определение $\delta(s,y)$ не може да е по-голямо от теглото на този път. □

Лема 8.8 (Свойство на оценките)**.** След инициализация и след всяка произволна последователност от релаксации за всеки връх v е изпълнено

$$v.d \geq \delta(s,v).$$

Освен това, щом за даден връх се достигне равенството

$$v.d = \delta(s,v),$$

то то се запазва при всички следващи релаксации.

Доказателство. Твърдението се доказва с индукция по броя извършени релаксации. След инициализацията $s.d = 0$, а за останалите върхове оценката е $+\infty$, следователно твърдението е вярно.
При релаксация на (x, y) може да се промени само $y.d$. Ако се промени, новата му стойност е

$$x.d + \omega((x, y)) \geq \delta(s, x) + \omega((x, y)) \geq \delta(s, y),$$

където последното неравенство е от предишната лема. Релаксацията никога не увеличава оценки. Следователно достигнато точно разстояние не може по-късно да бъде развалено. □

Лема 8.9 (Релаксация по най-къс път)**.** Нека p е най-къс път от s до v и нека (u, v) е последното му ребро. Ако преди релаксацията на (u, v) е изпълнено

$$u.d = \delta(s, u),$$

то след тази релаксация е изпълнено

$$v.d = \delta(s, v).$$

Доказателство. По оптималната подструктура префиксът на p от s до u е най-къс път, следователно

$$\delta(s, v) = \delta(s, u) + \omega((u, v)).$$

След релаксацията

$$v.d \leq u.d + \omega((u, v)) = \delta(s, v).$$

От свойството на оценките имаме и $v.d \geq \delta(s, v)$, откъдето следва равенство. □

8.3 Алгоритъм на Дийкстра

8.3.1 Вариант на задачата и идея

Алгоритъмът на Дийкстра решава задачата SSSP в ориентиран тегловен граф, когато всички тегла са неотрицателни:

$$\omega(e) \geq 0, \qquad e \in E.$$

Той поддържа множество D от окончателно обработени върхове. На всяка стъпка се избира необработен връх с най-малка текуща оценка и той се добавя в D. След това се релаксират всички изходящи от него ребра.

8.3.2 Използвани операции

Псевдокодът извиква операции на приоритетна опашка. Те не са част от самия алгоритъм: всяка реализация, която ги поддържа, е допустима, а изборът на реализация влияе само на сложността.

Дефиниция 8.10 (Приоритетна опашка по минимум)**.** Приоритетна опашка е структура от данни, която пази множество от елементи, всеки с числов ключ, и поддържа операциите:

- `Insert(Q,x)` — добавя елемента x с текущия му ключ;
- `Extract-Min(Q)` — премахва от Q и връща елемент с минимален ключ;
- `Decrease-Key(Q,x,k)` — заменя ключа на x с новата стойност k, която не е по-голяма от старата, и възстановява вътрешния ред на структурата.

В алгоритъма на Дийкстра ключът на връх v е текущата оценка $v.d$. Записът $Q \leftarrow V(G)$ означава построяване на опашка от всички върхове с текущите им ключове, тоест $|V|$ операции `Insert` или еднократно построяване.

8.3.3 Псевдокод

```
Dijkstra(G, w, s)
    ISS(G, s)                  // всички оценки са infinity, s.d = 0
    D <- emptyset              // окончателно обработените върхове
    create priority queue Q    // ключ на връх v е оценката v.d
    Q <- V(G)                  // всички върхове чакат в опашката
    while Q is not empty
        x <- Extract-Min(Q)    // необработен връх с най-малък d
        D <- D union {x}       // от този момент x.d е окончателно
        for each y in adj(x)   // релаксация на изходящите ребра
            if y in Q and y.d > x.d + w((x, y))
                y.d  <- x.d + w((x, y))
                y.pi <- x
                Decrease-Key(Q, y, y.d)   // синхронизира опашката
```

Означенията в псевдокода са следните.

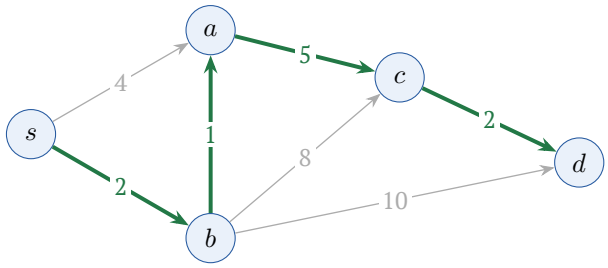
- G, w и s са входът: ориентиран граф, теглова функция с неотрицателни стойности и начален връх.
- Q съдържа необработените върхове, а $D = V \setminus Q$ — обработените; двете множества винаги се допълват, така че D се пази само за удобство на изложението и може да се пропусне.
- Тялото на `while` се изпълнява по веднъж за всеки връх: `Extract-Min` премахва връх и нищо не се връща обратно в Q.
- Трите реда в тялото на `if` са точно `Relax((x,y))`, следвана от `Decrease-Key`; последната само съобщава на опашката, че ключът на y е намалял.

Проверката $y \in Q$ не позволява промяна на вече окончателно обработен връх; тя се реализира за константно време с отделна булева информация дали върхът е в опашката.

Резултатът от алгоритъма са оценките $v.d$ и указателите $v.\pi$, които образуват дърво на най-късите пътища с корен s.

8.3.4 Пример

▷ **Пример 8.11.** Разглеждаме графа от фигура 8.1 с начален връх s.



Фигура 8.1: Ориентиран граф с неотрицателни тегла. Удебелените ребра са ребрата $\{(v.\pi, v)\}$ на полученото дърво на най-късите пътища.

След `ISS` е $s.d = 0$, а всички останали оценки са ∞. Всеки ред от таблицата отговаря на една итерация на цикъла `while`: извлеченият връх, окончателната му оценка и състоянието на останалите оценки след релаксациите. Записът $3(b)$ означава $d = 3$ и $\pi = b$.

Извлечен x	$x.d$	Релаксирани ребра	Оценки в Q след стъпката
s	0	(s,a), (s,b)	a:$4(s)$, b:$2(s)$, c:∞, d:∞
b	2	(b,a), (b,c), (b,d)	a:$3(b)$, c:$10(b)$, d:$12(b)$
a	3	(a,c)	c:$8(a)$, d:$12(b)$
c	8	(c,d)	d:$10(c)$
d	10	няма изходящи ребра	$Q = \varnothing$

Върховете се извличат в реда s, b, a, c, d, тоест по нарастващи окончателни оценки $0, 2, 3, 8, 10$, а не по номер или по брой ребра. Три пъти се задейства `Decrease-Key`: оценката на a пада от $4(s)$ на $3(b)$, на c — от $10(b)$ на $8(a)$, и на d — от $12(b)$ на $10(c)$. Обърнете внимание, че a е бил с оценка 4, докато още е чакал в Q; именно затова окончателна става само оценката на извлечения връх.
Полученият резултат е

$$\delta(s,s) = 0, \quad \delta(s,a) = 3, \quad \delta(s,b) = 2, \quad \delta(s,c) = 8, \quad \delta(s,d) = 10,$$

а указателите π дават най-късия път до d:

$$s \to b \to a \to c \to d.$$

† **Допълнение 8.12 (Бележка).** Алгоритъмът на Дийкстра не е коректен при отрицателни тегла. Причината е, че връх може да бъде окончателно избран, а по-късно да се открие по-лек път до него през ребро с отрицателно тегло.

▷ **Пример 8.13 (Дийкстра греши при отрицателно тегло).** Нека графът има върхове s, a, b и ребра

$$(s,a) \text{ с тегло } 2, \qquad (s,b) \text{ с тегло } 3, \qquad (b,a) \text{ с тегло } -2.$$

Отрицателен цикъл няма — графът е ацикличен. Истинското разстояние е

$$\delta(s,a) = 3 + (-2) = 1.$$

Алгоритъмът обаче извлича първо s и получава a:2, b:3. След това минималната оценка в Q е тази на a, затова a се извлича и попада в D с $a.d = 2$. Когато по-късно се обработи b и се релаксира (b,a), проверката $y \in Q$ е лъжа, защото a вече не е в опашката, и оценката остава $2 \neq 1$.
Точно това не може да се случи при неотрицателни тегла: тогава удължаването на маршрут никога не го прави по-лек, което е и същината на доказателството по-долу.

8.3.5 Коректност

★ **Теорема 8.14 (Коректност на Дийкстра).** Нека G е ориентиран тегловен граф без отрицателни тегла и s е негов връх. След завършване на алгоритъма на Дийкстра за всеки връх v е изпълнено

$$v.d = \delta(s,v).$$

Доказателство. Ще докажем, че когато връх v бъде добавен в D, вече е изпълнено

$$v.d = \delta(s,v).$$

Да допуснем противното и нека v е първият връх, добавен в D с неточна оценка. От свойството на оценките може да има само случая

$$v.d > \delta(s,v).$$

Връх v не е s, защото $s.d = 0 = \delta(s,s)$.
Нека p е най-къс път от s до v. В момента непосредствено преди добавянето на v в D разглеждаме първия връх u по p, който не принадлежи на D. Нека a е предшественикът на u по p. Тогава $a \in D$ и при обработването на a реброто (a,u) е било релаксирано.
Тъй като a е добавен преди v, по избора на v имаме

$$a.d = \delta(s,a).$$

От лемата за релаксация по най-къс път следва

$$u.d = \delta(s,u).$$

Върховете u и v са извън D, а v е избран от приоритетната опашка с минимална оценка. Следователно

$$v.d \leq u.d.$$

Понеже теглата са неотрицателни и u предхожда v по най-късия път p, имаме

$$\delta(s,u) \leq \delta(s,v).$$

А от предположението $v.d > \delta(s,v)$ получаваме

$$v.d \leq u.d = \delta(s,u) \leq \delta(s,v) < v.d,$$

което е противоречие. Следователно всеки връх се добавя в D с точна оценка. При завършване всички върхове са обработени. □

8.3.6 Сложност

Нека $n = |V|$ и $m = |E|$. Времето зависи от реализацията на приоритетната опашка.

- При неупорядочен масив намирането и извличането на минимум струва $\Theta(n)$ на итерация. Общата сложност е
$$\Theta(n^2 + m),$$
обикновено записвана като $\Theta(n^2)$.
- При двоична пирамида извличането на минимум и намаляването на ключ струват $\Theta(\log n)$. Има най-много n извличания и до m релаксации, които могат да намалят ключ. Получава се
$$\Theta((n+m)\log n).$$
- При по-сложни реализации на приоритетни опашки анализът може да бъде по-добър за операцията по намаляване на ключа, но основната идея на алгоритъма не се изменя.

8.4 Най-къси пътища в тегловен DAG

8.4.1 Алгоритъм

Ориентираният ацикличен граф, или DAG, допуска топологична наредба на върховете: за всяко ребро (u,v) връх u се намира преди v в наредбата. Това позволява върховете да се обработят така, че преди даден връх да са обработени всички възможни предшественици по път от източника.

Алгоритъмът решава SSSP в тегловен DAG и допуска отрицателни тегла.

```
DAG-SSSP(G, w, s)
    Toposort(G)
    ISS(G, s)
    for each x in topological order
        for each y in adj(x)
            Relax((x, y))
```

8.4.2 Коректност

★ **Теорема 8.15 (Коректност на алгоритъма за DAG).** Нека G е тегловен ориентиран ацикличен граф и $s \in V$. След завършването на `DAG-SSSP` за всеки връх v е изпълнено

$$v.d = \delta(s, v).$$

Доказателство. Нека върховете са разгледани в топологична наредба. За всеки връх, стоящ преди s, няма път от s до него, защото всеки път следва топологичната наредба. Следователно за него

$$v.d = +\infty = \delta(s, v).$$

За s след инициализацията е изпълнено

$$s.d = 0 = \delta(s, s),$$

тъй като в DAG няма цикли, а следователно няма и отрицателни цикли.
Нека v е произволен връх след s в топологичната наредба. Всеки вътрешен връх на път от s до v стои преди v. Нека

$$U = \{u \in V \mid (u, v) \in E \text{ и има път от } s \text{ до } u\}.$$

До момента, в който започне обработването на v, всички върхове от U вече са обработени. По индуктивното предположение за тях е вярно

$$u.d = \delta(s, u).$$

При релаксиране на всички входящи към v възможности, реализирани чрез обработката на техните начала, оценката става

$$v.d = \min_{u \in U}\{\delta(s, u) + \omega((u, v))\}.$$

По оптималната подструктура това е точно $\delta(s, v)$. Ако $U = \emptyset$, връх v не е достижим от s; тогава и двете стойности са $+\infty$. □

8.4.3 Сложност и предимства

Топологичното сортиране има сложност $\Theta(n+m)$. Вложеният цикъл обхожда всеки връх и всяко ребро най-много веднъж, затова общата сложност е

$$\Theta(n + m).$$

Алгоритъмът има две важни предимства пред алгоритъма на Дийкстра върху DAG:

- работи с отрицателни тегла, защото в DAG няма цикли и следователно няма отрицателни цикли;

- има линейна сложност $\Theta(n+m)$, която е асимптотично по-добра от сложността на Дийкстра с приоритетна опашка.

Забележка 8.16. Чрез обръщане на неравенството в операцията за релаксация алгоритъмът за DAG може аналогично да намира най-дълги пътища. Това е възможно именно поради липсата на цикли.

8.5 Алгоритъм на Белман–Форд

8.5.1 Вариант на задачата и псевдокод

Алгоритъмът на Белман–Форд решава SSSP в ориентиран тегловен граф, който може да съдържа отрицателни ребра. Освен това открива наличие на отрицателен цикъл, достижим от началния връх.

```
Bellman-Ford(G, w, s)
    ISS(G, s)
    for i <- 1 to n - 1
        for each (x, y) in E(G)
            Relax((x, y))
    for each (x, y) in E(G)
        if y.d > x.d + w((x, y))
            return False
    return True
```

Ако алгоритъмът върне `True`, отрицателен цикъл, достижим от s, не е открит и оценките представляват най-късите разстояния. Ако върне `False`, съществува достижим отрицателен цикъл.

8.5.2 Коректност при липса на достижими отрицателни цикли

Лема 8.17. Нека в графа няма отрицателен цикъл, достижим от s. След изпълнение на първите $n-1$ обхождания на всички ребра от алгоритъма на Белман–Форд за всеки връх v е изпълнено

$$v.d = \delta(s, v).$$

Доказателство. При липса на отрицателни цикли, достижими от s, могат да се изберат прости най-къси пътища от s до достижимите върхове и техните предшественици да се разглеждат като дърво на най-късите пътища с корен s. Недостижимите върхове запазват оценка $+\infty$: ако към такъв връх имаше ребро от достижим връх, той също би бил достижим.
Ще използваме индукция по нивата на това дърво. Преди първото обхождане е вярно

$$s.d = 0 = \delta(s, s).$$

Да допуснем, че след някакъв брой обхождания всички върхове до ниво $i-1$ имат точни оценки. Нека t е връх на ниво i, а v е неговият родител в дървото на най-късите пътища. При следващото обхождане реброто (v,t) се релаксира. Тъй като

$$v.d = \delta(s,v),$$

от лемата за релаксация по най-къс път следва

$$t.d = \delta(s,t).$$

Това равенство се запазва.
Всеки прост път съдържа най-много $n-1$ ребра. Следователно след $n-1$ пълни обхождания са достигнати всички нива на дървото на най-късите пътища и всички оценки са точни. □

Лема 8.18. Ако в графа няма отрицателен цикъл, достижим от s, то след първите $n-1$ обхождания за всяко ребро (x,y) е изпълнено

$$y.d \leq x.d + \omega((x,y)).$$

Доказателство. Ако x е достижим от s, тогава и y е достижим и след първите $n-1$ обхождания имаме $x.d = \delta(s,x)$ и $y.d = \delta(s,y)$. Твърдението следва от неравенството за ребро. Ако x не е достижим, тогава $x.d = +\infty$ и неравенството е тривиално. □

Теорема 8.19. Ако в графа няма отрицателен цикъл, достижим от s, алгоритъмът на Белман–Форд връща `True` и за всеки връх v е изпълнено

$$v.d = \delta(s,v).$$

Доказателство. Точността на оценките след първия двоен цикъл следва от първата лема в този подраздел. От следващата лема тогава за никое ребро условието в последния цикъл не е изпълнено, поради което алгоритъмът достига `return True`. □

8.5.3 Откриване на отрицателен цикъл

Лема 8.20. Ако в графа има отрицателен цикъл, достижим от s, то при проверката след $n-1$ обхождания съществува ребро (x,y), за което

$$y.d > x.d + \omega((x,y)).$$

Доказателство. Да допуснем противното: за всяко ребро (x,y) е изпълнено

$$y.d \leq x.d + \omega((x,y)).$$

Нека отрицателният цикъл е

$$v_1, e_1, v_2, e_2, \ldots, v_k, e_k, v_1.$$

Тъй като цикълът е достижим от s, след извършените обхождания всички негови върхове имат крайни оценки. Прилагаме предположеното неравенство за всички ребра на цикъла:

$$d(v_2) \leq d(v_1) + \omega(e_1),$$
$$d(v_3) \leq d(v_2) + \omega(e_2),$$
$$\ldots$$
$$d(v_1) \leq d(v_k) + \omega(e_k).$$

Като съберем неравенствата, всички оценки се съкращават и получаваме

$$0 \leq \omega(e_1) + \omega(e_2) + \cdots + \omega(e_k).$$

Това противоречи на отрицателното тегло на цикъла. □

★ **Теорема 8.21 (Коректност на Белман–Форд).** Алгоритъмът на Белман–Форд връща `True` точно когато в графа няма отрицателен цикъл, достижим от s; в този случай за всеки връх v е изпълнено

$$v.d = \delta(s, v).$$

Ако върне `False`, има достижим отрицателен цикъл.

Доказателство. Ако няма достижим отрицателен цикъл, всеки достижим връх има най-къс прост път с най-много $|V| - 1$ ребра. По индукция по дължината на такъв път след k пълни обхода на ребрата е получена правилната стойност за всеки връх с най-къс път от най-много k ребра: поредният обход релаксира последното ребро след вече получения префикс. Оценките не могат да са по-малки от истинското разстояние, защото всяка крайна оценка е тегло на реален маршрут. След $|V| - 1$ обхода всички оценки са точни и никое ребро не допуска подобрение; алгоритъмът връща `True`.
Ако има достижим отрицателен цикъл, всички негови върхове имат крайни оценки след тези обходи. Ако никое негово ребро не допуска подобрение, за всяко (u, v) от цикъла е $v.d \leq u.d + w(u, v)$. Сумирането съкращава оценките и дава $0 \leq w(C)$, противоречие. Значи проверката открива подобрение и връща `False`. Недостижимите върхове остават с $+\infty$ и не предизвикват такова подобрение. □

8.5.4 Сложност

Външният цикъл се изпълнява $n - 1$ пъти, а при всяко изпълнение се обхождат всички m ребра. Последната проверка има сложност $\Theta(m)$. Следователно времевата сложност е

$$\Theta(nm).$$

В най-лошия случай, когато $m = \Theta(n^2)$, тя е кубична:

$$\Theta(n^3).$$

8.6 Алгоритъм на Флойд–Уоршал

8.6.1 Вариант на задачата

Алгоритъмът на Флойд–Уоршал решава задачата APSP: намира най-късите разстояния между всички двойки върхове. Графът е ориентиран и е номериран с върхове

$$V = \{1, \ldots, n\}.$$

Входът се представя с матрица на теглата ω, където

$$\omega[i,j] = \begin{cases} 0, & i = j, \\ \omega((i,j)), & i \neq j \text{ и } (i,j) \in E, \\ +\infty, & i \neq j \text{ и } (i,j) \notin E. \end{cases}$$

8.6.2 Динамично програмиране

За път p нека intern(p) е множеството от неговите вътрешни върхове. Нека $D^{(k)}[i,j]$ означава теглото на най-къс път от i до j, чиито вътрешни върхове са само измежду

$$\{1, \ldots, k\}.$$

При $k = 0$ не се разрешават вътрешни върхове, затова

$$D^{(0)} = \omega.$$

За $k > 0$ има две възможности за най-къс път от i до j с разрешени вътрешни върхове от $\{1, \ldots, k\}$:

- връх k не участва като вътрешен връх; тогава стойността е $D^{(k-1)}[i,j]$;
- връх k участва като вътрешен връх; тогава пътят се разделя на път от i до k и път от k до j, чиито вътрешни върхове са от $\{1, \ldots, k-1\}$.

Следователно рекурентната формула е

$$D^{(k)}[i,j] = \min\left\{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\right\}.$$

8.6.3 Псевдокод

```
Floyd-Warshall(w)
    D <- w
```

```
for k <- 1 to n
    for i <- 1 to n
        for j <- 1 to n
            D[i, j] <- min(D[i, j], D[i, k] + D[k, j])
return D
```

8.6.4 Коректност

★ **Теорема 8.22 (Коректност на Флойд–Уоршал).** При отсъствие на отрицателни цикли алгоритъмът на Флойд–Уоршал връща матрица D, за която

$$D[i, j] = \delta(i, j)$$

за всички $i, j \in \{1, \ldots, n\}$.

Доказателство. Ще докажем с индукция по k, че след завършване на итерацията за k матрицата съдържа точно $D^{(k)}$.
За $k = 0$ това е вярно по дефиницията на входната матрица ω: позволени са само пътища без вътрешни върхове.
Нека твърдението е вярно за $k-1$. За всяка двойка i, j рекурентната формула разглежда точно двата възможни случая: връх k не помага или k е вътрешен връх на пътя. Във втория случай оптималният път се разделя на два подпътя с вътрешни върхове, номерирани най-много с $k - 1$. Следователно присвояването изчислява точно $D^{(k)}[i, j]$.
При $k = n$ всички върхове са разрешени като вътрешни. При липса на отрицателни цикли всеки най-къс път може да бъде избран прост, така че получената стойност е $\delta(i, j)$. □

8.6.5 Сложност по време и памет

Трите вложени цикъла извършват $\Theta(n^3)$ операции. Следователно времевата сложност е

$$\Theta(n^3).$$

Буквалното съхраняване на всички матрици

$$D^{(0)}, D^{(1)}, \ldots, D^{(n)}$$

изисква

$$\Theta(n^3)$$

памет. Достатъчно е обаче да се пазят само две работни матрици: една за $D^{(k-1)}$ и една за $D^{(k)}$. Така използваната памет става

$$\Theta(n^2).$$

Възможна е и реализация на място, при която се използва една матрица D и се изпълнява

$$D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}.$$

Тогава извън входната матрица е нужна $\Theta(1)$ допълнителна памет, но входната матрица се презаписва. При липса на отрицателни цикли случаите $i = k$ или $j = k$ не пречат на коректността, защото диагоналните стойности са нула и връх k не може да бъде вътрешен връх на прост път, който започва или завършва в k.

8.7 Изпитен фокус

- Да се дефинират тегло на път, $\delta(u, v)$, отрицателен цикъл и четирите варианта на задачата за най-къси пътища.
- Да се обясни защо отрицателен цикъл, разположен по път от u до v, прави стойността $\delta(u, v)$ равна на $-\infty$, както и защо при липса на достижим отрицателен цикъл може да се избере прост най-къс път.
- Да се възпроизведат и използват оптималната подструктура на най-късите пътища, инициализацията `ISS`, операцията `Relax` и инвариантът $v.d \geq \delta(s, v)$.
- За Дийкстра: да се посочат условието за неотрицателни тегла, псевдокодът, множеството D, операциите `Extract-Min` и `Decrease-Key` на приоритетната опашка и ролята на проверката $y \in Q$, доказателствената идея с първия неправилно избран връх и сложностите при различни реализации на опашката.
- Да се посочи контрапример, показващ, че при отрицателно тегло Дийкстра дава грешен резултат дори без отрицателен цикъл.
- За DAG-SSSP: да се посочат топологичната наредба, релаксирането на всички ребра в тази наредба, доказателството по реда на върховете, сложността $\Theta(n + m)$ и предимствата пред Дийкстра.
- За Белман–Форд: да се възпроизведат $n - 1$ пълни обхождания на всички ребра, финалната проверка, доказателството чрез нивата в дърво на най-късите пътища и доказателството за откриване на отрицателен цикъл чрез сумиране на неравенства по цикъла.
- За Флойд–Уоршал: да се дефинира $D^{(k)}[i, j]$, да се изведе рекурентната формула според това дали връх k участва, да се даде псевдокодът, индуктивното доказателство и сложностите $\Theta(n^3)$ по време и $\Theta(n^2)$ памет при реализация с една или две матрици.

Част II

Ядро на компютърните науки

Въпрос 9

Компютърни архитектури. Формати на данните. Вътрешна структура на централен процесор.

По конспекта: какво трябва да мога

Официално заглавие: „Компютърни архитектури — Формати на данните. Вътрешна структура на централен процесор — *блокове и конвейерна обработка, инструкции*.“ Заглавието на главата е съкратено; конвейерът е изискване, не допълнение.
Описание: обща структура на компютрите, запомнена програма — §9.1, стр. 130.
Описание: концептуално изпълнение на инструкциите — §9.4, стр. 141.
Формати на данните: цели двоични числа, двоично-десетични числа, числа с плаваща запетая, знакови данни и кодови таблици — §9.2, стр. 132.
Централен процесор: регистри, АЛУ, регистри на състоянията и флаговете, блокове за управление, връзка с паметта, дешифрация на инструкциите, преходи — §9.3, стр. 137.
Конвейерна обработка: §9.5, стр. 144.
Литература: [5], [36], [45].

9.1 Обща структура на компютрите и компютърни архитектури

Компютърната архитектура описва основните компоненти на компютъра, начина на свързването им и принципите, по които се изпълняват програмите. В общия случай компютърът се състои от:

- централен процесор;
- основна памет;
- входно-изходни устройства.

Тези компоненти са взаимносвързани и работят съвместно, за да изпълняват програми. Централният процесор обработва данните и управлява изпълнението на инструкциите. Основната памет съхранява инструкции и данни. Входно-изходните устройства осигуряват обмен на информация между компютъра и външната среда.

Връзката между компонентите се осъществява чрез шини.

Дефиниция 9.1 (Шина)**.** Шината е съвкупност от проводници, по които се предава информация между компонентите на компютъра. Обичайно се разграничават три шини:

- *адресна шина* — по нея процесорът подава адреса на клетката от паметта или на устройството, към което се обръща;
- *шина за данни* — по нея се пренася самата стойност, която се чете или записва;
- *контролна шина* — по нея се предават управляващите сигнали, например дали текущото обръщение е четене или запис, и сигналите за прекъсване.

9.1.1 Архитектура на фон Нойман

Повечето съвременни компютри се основават на архитектурата на Джон фон Нойман. Тя се характеризира със следните основни идеи:

1. Инструкциите и данните се съхраняват в памет, която позволява четене и запис.
2. Съдържанието на паметта се адресира чрез адрес, независимо от вида на данните, които се намират на съответното място.
3. Инструкциите се изпълняват последователно, от една инструкция към следващата, освен ако текущата инструкция не промени реда на изпълнение.

Последната особеност е съществена за разбирането на програмния брояч. Той съдържа адреса на инструкцията, която трябва да бъде извлечена и изпълнена. При обикновено последователно изпълнение програмният брояч се насочва към следващата инструкция. При преход или друга инструкция, която променя управлението, в него се записва различен адрес.

Дефиниция 9.2 (Запомнена програма)**.** Запомнена програма е програма, чиито инструкции са представени в двоичен вид и са съхранени в паметта заедно с данните, върху които работи програмата. Процесорът извлича инструкциите от паметта и ги изпълнява последователно, като редът може да бъде променен чрез инструкции за преход.

Така програмата не е външна поредица от действия, която процесорът следва по специален начин, а е информация, записана в паметта. Процесорът интерпретира тази информация като инструкции и данни според съответния формат.

9.2 Формати на данните

Дефиниция 9.3 (Формат на данни)**.** Форматът на данните определя каква битова последователност представя дадена стойност. Една и съща последователност от битове може да има различно значение в зависимост от това дали се разглежда като цяло число, двоично-десетично число, число с плаваща запетая или код на символ.

9.2.1 Цели двоични числа

Целите двоични числа се представят в двоична бройна система. При знаковите числа един от битовете обикновено се използва за представяне на знака. Разглеждат се различни кодове за представяне на отрицателните числа.

Прав код

Дефиниция 9.4 (Прав код)**.** При правия код най-старшият бит в n-битовото поле е знаков. Стойност 0 означава положително число, а стойност 1 означава отрицателно число. Останалите битове представят абсолютната стойност на числото в двоична бройна система.

Следователно в прав код има отделно поле за знака и отделно поле за абсолютната стойност. Основните недостатъци на този начин на представяне са:

- нулата има две представяния – положителна и отрицателна;
- събирането и изваждането изискват по-сложна апаратна реализация.

Диапазонът на представимите стойности е от $-(2^{n-1}-1)$ до $2^{n-1}-1$.

Двоично-допълнителен код

Дефиниция 9.5 (Двоично-допълнителен код)**.** При двоично-допълнителния код n-битовата последователност $b_{n-1}b_{n-2}\dots b_0$ представя числото

$$-b_{n-1}2^{n-1}+\sum_{i=0}^{n-2} b_i 2^i.$$

Тоест всички битове имат обичайните си тегла, но най-старшият участва с отрицателно тегло. Затова той отново определя знака: стойност 1 дава отрицателно число. Еквивалентно, отрицателното число $x<0$ се записва като двоичния запис на 2^n+x; оттук идва и името на кода.

От определението следва диапазонът на представимите стойности:

$$[-2^{n-1},\ 2^{n-1}-1],$$

защото най-малката стойност се получава при $b_{n-1} = 1$ и всички останали битове нула, а най-голямата — при $b_{n-1} = 0$ и всички останали единици.

Този код използва едно представяне на нулата и позволява операциите със знакови числа да се реализират чрез обикновено двоично събиране, като се работи с фиксиран брой битове.

Смяната на знака на число в двоично-допълнителен код се извършва в две стъпки:

1. всички битове се инвертират побитово;
2. към получения резултат се прибавя 1.

Твърдение 9.6. Нека x е представено в n-битов двоично-допълнителен код и нека $\overline{x}$ е побитовото му инвертиране. Тогава $\overline{x} + 1$ е представянето на $-x$ при пресмятане с n бита.

Доказателство. Инвертирането на бит го заменя с 1 минус стойността му, затова при разглеждане на битовете като безнакови тегла

$$x + \overline{x} = \underbrace{11\ldots1}_{n} = 2^n - 1.$$

Следователно

$$\overline{x} + 1 = 2^n - x.$$

При работа с n бита преносът извън най-старшия бит се отхвърля, тоест пресмята се по модул 2^n, а

$$2^n - x \equiv -x \pmod{2^n}.$$

□

▷ **Пример 9.7.** Нека числото 3_{10} бъде представено с четири бита:

$$3_{10} = 0011_2.$$

Инвертирането на битовете дава

$$0011 \longrightarrow 1100.$$

След прибавяне на 1 се получава

$$1100 + 1 = 1101.$$

Следователно 1101_2 е представянето на -3_{10} в четирибитов двоично-допълнителен код.

При събиране на число с противоположното му число резултатът е нула в рамките на използвания брой битове. Преносът извън най-старшия бит не се включва в резултата.

† **Допълнение 9.8 (Бележка).** При работа с двоично-допълнителен код броят на битовете трябва да бъде фиксиран предварително. Например в четирибитово поле резултатът от събиране може да съдържа пренос извън полето, който не е част от представената стойност.

9.2.2 Двоично-десетични числа

Дефиниция 9.9 (Двоично-десетичен код)**.** При двоично-десетичното представяне всяка десетична цифра се заменя с уникална двоична последователност. За една десетична цифра се използват четири бита. Понеже с четири бита могат да се образуват 16 комбинации, а десетичните цифри са само десет, шест комбинации остават неизползвани.

Предимството на двоично-десетичното представяне е, че числата са удобни за извеждане и за работа с десетични цифри. Недостатък е усложненото изпълнение на аритметичните операции.

Различават се две основни форми:

Непакетирана форма. Всяка десетична цифра се съхранява в отделен байт.

Пакетирана форма. В един байт се съхраняват две десетични цифри, като всяка цифра заема по четири бита.

При непакетираната форма се използва повече памет, но обработката на отделните цифри е по-пряка. При пакетираното представяне паметта се използва по-икономично, защото два четирибитови кода се съхраняват в един байт.

9.2.3 Двоични числа с плаваща запетая

Числата с плаваща запетая се използват за представяне на нецели числа. Общата идея е стойността да се представи чрез знак, мантиса и характеристика, тоест като произведение на мантиса и степен на основата:

$$\pm S \times B^{\pm E}.$$

Дефиниция 9.10 (Мантиса, характеристика, отклонение)**.** В представянето $\pm S \times B^{\pm E}$:

- S е *мантисата* — значещите цифри на числото;
- B е *основата* на представянето, при двоичните формати $B = 2$;
- E е *характеристиката*, наричана още *експонента* — степента, на която се повдига основата, тоест с колко позиции се измества запетаята.

Полето за характеристиката съдържа не самата стойност E, а изместена стойност, за да се представят и отрицателни експоненти без отделен знаков бит. Фиксираната добавка се нарича *отклонение*: истинската експонента се получава, като отклонението се извади от записаното в полето.

В един двоичен формат числото може да бъде разделено на три части:

- един бит за знака;
- поле за характеристиката, или експонентата;
- поле за мантисата.

Посоченият пример използва формат с общо 32 бита:

- 1 бит за знака;
- 8 бита за характеристиката;
- 23 бита за мантисата.

За характеристиката обикновено се използва отклонение. Отклонението е фиксирана стойност, която се изважда от записаната характеристика, за да се получи действителната стойност на експонентата. При k бита за характеристиката обичайно се използва отклонение

$$2^{k-1} - 1.$$

При $k = 8$ това отклонение е 127. Източникът посочва диапазон на действителните стойности на характеристиката от -127 до 128.

За да се избегнат различни представяния на една и съща стойност, числата се нормализират. Без такова условие едно и също число има много записи: например

$$1{,}5 = 1{,}5 \times 2^0 = 0{,}75 \times 2^1 = 0{,}375 \times 2^2 = \ldots$$

Дефиниция 9.11 (Нормализирано число)**.** Ненулево число с плаваща запетая е *нормализирано*, ако характеристиката му е избрана така, че мантисата да попада в предварително фиксиран интервал. При основа $B = 2$ това означава

$$1 \leq S < 2,$$

тоест числото се записва във вида

$$\pm 1{,}f \times 2^E.$$

При това условие всяко ненулево число има точно едно представяне.

Тъй като при двоична основа старшата цифра на нормализирана мантиса е винаги 1, тя не се записва в полето за мантисата; говори се за *скрит бит*. Така при 23 бита се получава точност от 24 значещи двоични цифри. Нулата не може да се нормализира и за нея се запазва отделно представяне.

† **Допълнение 9.12 (Бележка).** Изречението, което определя нормализираното число, е останало незавършено в източника: формулата след него липсва. Дадената по-горе форма е стандартната за двоичните формати и е добавена, за да бъде определението използваемо.

Аритметиката на числата с плаваща запетая е по-сложна от аритметиката на целите числа. Стандартното аритметико-логическо устройство не е достатъчно за непосредственото изпълнение на всички операции върху тях. Затова се използва отделно изчислително звено – блок за плаваща запетая, наричан още *Floating Point Unit*.

Инструкциите за работа с числа с плаваща запетая имат специфични особености. Поради това те се разглеждат отделно от инструкциите за обикновени цели двоични числа.

† **Допълнение 9.13 (Бележка).** При числата с плаваща запетая точният диапазон и конкретните специални представяния зависят от избрания формат. Тук се разглежда само общата структура, дадена в източника: знак, характеристика и мантиса.

9.2.4 Знакови данни и кодови таблици

Дефиниция 9.14 (Кодова таблица)**.** Кодова таблица е съответствие между символите от определена азбука и битови последователности с определена дължина, чрез които тези символи се съхраняват и обработват в компютъра.

Кодовите таблици позволяват на процесора и на програмите да обработват букви, цифри, препинателни знаци и други символи като цифрови данни.

Източникът посочва два стандарта:

ASCII. Използва седем бита за кодиране на английската азбука, цифрите, пунктуацията и някои други символи.

Unicode. Стандарт за представяне на символи чрез кодови стойности, предназначен за работа с различни азбуки и знакови системи.

9.3 Централен процесор

Централният процесор изпълнява инструкциите на програмата и управлява обработката на данните. За тази цел той трябва да извършва няколко основни действия:

1. да извлича инструкции от паметта;
2. да интерпретира инструкциите;
3. да извлича необходимите операнди;
4. да обработва данните;
5. да записва резултатите.

Извличането на инструкция означава прочитане на инструкция от достъпно за процесора място – регистри, кеш или основна памет. Интерпретирането включва дешифрация на инструкцията, за да се установи каква операция трябва да бъде изпълнена и какви са нейните операнди.

Изпълнението може да изисква четене на данни от паметта или от входно-изходен модул. Самата обработка може да представлява аритметична или логическа операция. След приключването ѝ резултатът се записва в регистър, памет или входно-изходно устройство.

Основните компоненти на процесора са:

- аритметико-логическо устройство;
- блок за управление;
- регистри.

9.3.1 Регистри

Дефиниция 9.15 (Регистър)**.** Регистрите са вътрешни места за временно съхраняване на данни, адреси, инструкции и информация за състоянието на процесора. Те позволяват на процесора да работи с междинни резултати и да запомня позицията си в програмата.

Регистрите могат да се разделят на две основни групи:

1. потребителски видими регистри;
2. регистри за управление и състояние.

Потребителски видими регистри

Дефиниция 9.16 (Потребителски видим регистър)**.** Потребителски видими са регистрите, които могат да бъдат достъпвани чрез машинни инструкции. Те позволяват на програмиста или на компилатора да съхранява междинни резултати и да намалява необходимостта от обръщения към основната памет.

Основните подкатегории са:

Регистри с общо предназначение. Съхраняват междинни резултати и могат да се използват при адресиране.

Регистри за данни. Предназначени са за съхраняване на данни, но не и за изчисляване на адрес на операнд.

Регистри за адреси. Използват се за адресиране. Те могат да бъдат регистри с общо предназначение или да са предназначени за конкретен режим на адресиране, например като индексни регистри, указатели на стека или сегментни указатели.

Границата между потребителски видимите регистри и регистрите за управление и състояние не е еднаква при всички процесори. Някои регистри, използвани основно от операционната система, могат да бъдат достъпни чрез машинни инструкции, макар да не се използват пряко от обикновените програми.

Регистри за управление и състояние

Четири регистъра са особено важни за изпълнението на инструкциите:

Програмен брояч. Съдържа адреса на инструкцията, която трябва да бъде прочетена. При последователно изпълнение този адрес се променя така, че да сочи към следващата инструкция.

Регистър на инструкциите. Съдържа последно извлечената инструкция, която се декодира и изпълнява.

Регистър на адреса на паметта (MAR). Съдържа адреса на място в паметта, към което ще се извърши обръщение.

Буферен регистър на паметта (MBR). Съдържа данните, които трябва да бъдат записани в паметта, или последно прочетените от паметта данни.

Много процесори имат и регистър на флаговете.

Дефиниция 9.17 (Флаг и регистър на състоянието)**.** *Флаг* е отделен бит, който записва едно двоично условие — най-често свойство на резултата от последната операция на аритметико-логическото устройство или режим на работа на процесора. Регистърът, в който са събрани тези битове, се нарича *регистър на състоянието* или *регистър на флаговете*.
Флаговете не се четат като обикновени данни, а се използват от инструкциите за условен преход: те решават коя да бъде следващата инструкция.

Сред често срещаните флагове са:

- флаг за знак;
- флаг за нулев резултат;
- флаг за пренос;
- флаг за препълване;
- флаг за разрешени или забранени прекъсвания.

Флагът за знак показва знака на резултата, флагът за нула показва дали резултатът е нулев, а флаговете за пренос и препълване описват определени условия при аритметични операции. Флагът за прекъсване участва в управлението на възможността процесорът да приема прекъсвания.

9.3.2 Аритметико-логическо устройство

Дефиниция 9.18 (Аритметико-логическо устройство)**.** Аритметико-логическото устройство, съкратено АЛУ, извършва аритметични и логически операции върху операнди. То получава данни от регистрите или от пътя за данни, изпълнява избраната операция и връща резултата към регистър или към следващ етап на обработката.

Работата на АЛУ се определя от управляващи сигнали. Те задават каква операция трябва да бъде извършена и как резултатът да бъде прехвърлен. В зависимост от резултата АЛУ може да промени и флаговете в регистъра на състоянието.

9.3.3 Блок за управление

Дефиниция 9.19 (Микрооперация)**.** Микрооперацията е най-малкото неделимо действие, което процесорът извършва за един такт — например прехвърляне на съдържанието на един регистър в друг, подаване на адрес към паметта или активиране на конкретна функция на аритметико-логическото устройство. Изпълнението на една машинна инструкция се разлага на последователност от микрооперации.

Блокът за управление координира работата на процесора. Той изпълнява две основни задачи:

1. задвижва процесора през поредица от микрооперации в правилната последователност;
2. генерира управляващи сигнали, които предизвикват изпълнението на микрооперациите.

Управляващите сигнали контролират прехвърлянето на данни между регистрите, избора на операция на АЛУ и обмена с паметта и входно-изходните устройства. Те управляват отварянето и затварянето на логически пътища, чрез които данните достигат до съответните регистри и функционални блокове.

Входове за блока за управление са:

- тактовият сигнал;
- съдържанието на регистъра на инструкциите;
- флаговете на процесора;
- управляващи сигнали от контролната шина.

Тактовият сигнал задава последователността на микрооперациите. Регистърът на инструкциите предоставя кода на операцията и информацията за режима на адресиране. Флаговете показват състоянието на процесора и резултатите от предишни операции. Управляващите сигнали от контролната шина осигуряват информация за състоянието на паметта и входно-изходните устройства.

Изходите на блока за управление са:

- управляващи сигнали вътре в процесора;
- управляващи сигнали към контролната шина.

Сигналите вътре в процесора осигуряват прехвърляне на данни от един регистър в друг и активират конкретни функции на АЛУ. Сигналите към контролната шина управляват обмена с паметта и входно-изходните устройства.

9.3.4 Инструкции

Машинната инструкция е двоично кодирана команда, която определя операцията и необходимите за нея данни. Обикновено една инструкция може да съдържа:

- код на операцията;
- указател към един или повече входни операнди;
- указател към изходен операнд;
- указател към следващата инструкция.

Дефиниция 9.20 (Код на операцията)**.** Кодът на операцията, или *opcode*, е двоична част от инструкцията, която определя какво действие трябва да извърши процесорът.

Дефиниция 9.21 (Режим на адресиране)**.** Режимът на адресиране определя как от полето на инструкцията се получава действителният адрес, или самата стойност, на операнда. Сред основните режими са:

- *непосредствен* — операндът е записан в самата инструкция;
- *регистров* — операндът е в посочения регистър;
- *директен* — инструкцията съдържа адреса на операнда в паметта;
- *недиректен* — инструкцията съдържа адрес, на който е записан адресът на операнда, така че е необходимо още едно обръщение към паметта.

Операндите могат да се намират в регистри, в паметта или във входно-изходно устройство. За да ги използва, процесорът трябва да определи местоположението им според зададения режим на адресиране.

9.3.5 Връзка с паметта

Връзката между процесора и паметта се реализира посредством шини, сред които особено значение има контролната шина. Процесорът подава управляващи сигнали, които указват дали се извършва четене или запис, а чрез регистрите MAR и MBR се задават адресът и обменяните данни.

При четене адресът на необходимото място се поставя в MAR. След осъществяване на обръщението прочетените данни се приемат в MBR. При запис адресът се поставя в MAR, а данните за запис – в MBR. Блокът за управление подава съответния управляващ сигнал към паметта.

9.4 Концептуално изпълнение на инструкциите

Състоянието на процесора образува контекста, необходим за изпълнението на следващата инструкция. Важна част от този контекст е програмният брояч, който съдържа адреса на инструкцията, която трябва да бъде извлечена.

На концептуално ниво изпълнението на инструкцията включва следните фази:

1. извличане;
2. декодиране и интерпретиране;
3. изчисляване на адресите на операндите;
4. извличане на операндите;

5. изпълнение;
6. записване на резултата;
7. проверка и обслужване на прекъсване.

9.4.1 Извличане на инструкция

При извличането процесорът използва адреса от програмния брояч. Този адрес се подава към паметта, а инструкцията се прочита и се поставя в регистъра на инструкциите. След извличането програмният брояч обикновено се насочва към следващата инструкция.

9.4.2 Декодиране

Блокът за управление анализира съдържанието на регистъра на инструкциите. От кода на операцията се определя какво действие трябва да бъде извършено. Разпознават се също операндите и режимът на адресиране.

Дешифрацията на инструкциите е необходима, защото една и съща обща процедура трябва да обслужва различни операции – например аритметични операции, логически операции, прехвърляния на данни и преходи.

9.4.3 Изчисляване и извличане на операнди

Ако инструкцията използва операнд от паметта или от входно-изходно устройство, процесорът изчислява адреса на операнда. След това извършва необходимото обръщение и извлича данните.

Ако операндът вече се намира в регистър, допълнително обръщение към основната памет не е необходимо. Във всички случаи блокът за управление трябва да осигури подаването на правилните данни към АЛУ или към друг функционален блок.

9.4.4 Изпълнение и записване на резултата

След извличането на операндите се изпълнява операцията, зададена от инструкцията. При аритметична или логическа операция АЛУ обработва данните. Резултатът може да бъде записан в регистър, в паметта или към входно-изходно устройство.

Ако операцията променя състоянието на процесора, се променят и съответните флагове. След това процесорът продължава със следващата инструкция или изпълнява преход, ако инструкцията е променила програмния брояч.

9.4.5 Недиректна стъпка

След извличането на инструкцията може да бъде необходимо да се извърши недиректна стъпка. Тя се прилага, когато инструкцията използва недиректна адресация. В такъв случай извлечената от инструкцията информация не е непосредствено крайният адрес

на операнда. Процесорът трябва да извърши допълнително обръщение, за да получи необходимия адрес.

9.4.6 Прекъсвания

Дефиниция 9.22 (Прекъсване)**.** Прекъсването е сигнал, който кара процесора да спре временно изпълнението на текущата програма и да премине към предварително определена обслужваща процедура. Сигналът може да идва от входно-изходно устройство, от таймер или от самото изпълнение на инструкция, например при опит за деление на нула.
Прекъсването не се проверява по средата на инструкция: то се обслужва след завършването на текущата инструкция, което е и последната фаза от концептуалното изпълнение.

Ако прекъсванията са разрешени и възникне прекъсване, процесорът запазва текущото състояние на програмата и преминава към обслужване на прекъсването.

Запазеното състояние включва:

- съдържанието на програмния брояч, тоест адреса на следващата инструкция;
- релевантна информация за изпълняваната програма;
- състоянието на регистрите;
- състоянието на входно-изходните устройства.

След обслужването на прекъсването запазеното състояние позволява изпълнението на прекъснатата програма да бъде продължено.

9.4.7 Преходи

При обикновено изпълнение програмният брояч сочи към следващата инструкция. Инструкцията за преход променя това поведение, като записва нов адрес в програмния брояч. Новият адрес може да зависи от флаговете на процесора.

Например при инструкция от вида `increment-and-skip-if-zero` блокът за управление увеличава програмния брояч, ако флагът за нулев резултат е установен. Така флаговете могат да участват пряко в избора на следващата инструкция.

† **Допълнение 9.23 (Бележка).** Формулировката е тази от източника. Става дума за *допълнително* увеличаване: програмният брояч се увеличава при всяко извличане на инструкция, а при изпълнена проверка се увеличава още веднъж, с което следващата инструкция се прескача. Оттам идва и *skip* в името на инструкцията.

9.5 Конвейерна обработка

Изпълнението на една инструкция може да бъде представено като последователност от фази: извличане, декодиране, извличане на операнди, изпълнение и записване на резултата. В процесора тези фази се реализират чрез поредица от микрооперации, управлявани от блока за управление.

Дефиниция 9.24 (Конвейерна обработка)**.** При конвейерната обработка изпълнението на инструкциите се разделя на последователни етапи, всеки от които се обслужва от отделна част на процесора. Докато една инструкция преминава през даден етап, следващата вече може да се обработва в предходния. По този начин отделните части на процесора работят едновременно върху различни инструкции. Конвейерът не съкращава времето за изпълнение на една инструкция, а увеличава броя инструкции, завършвани за единица време.

Основата на конвейерната обработка е съгласуваното управление на етапите чрез тактовия сигнал. Блокът за управление генерира управляващите сигнали в правилната последователност, така че данните да преминават между регистрите, АЛУ, паметта и останалите блокове.

При инструкции за преход нормалната последователност може да бъде променена, тъй като програмният брояч получава нов адрес. Затова преходите зависят от състоянието на процесора и от резултатите на предходни операции.

9.6 Изпитен фокус

При подготовка за устен изпит трябва да могат да бъдат възпроизведени:

- трите основни компонента на компютъра – централен процесор, основна памет и входно-изходни устройства;
- трите основни идеи на архитектурата на фон Нойман;
- понятието запомнена програма;
- представянето на цели числа в прав и двоично-допълнителен код;
- алгоритъмът за смяна на знака в двоично-допълнителен код и защо той работи: $x + \overline{x} = 2^n - 1$, откъдето $\overline{x} + 1 = 2^n - x \equiv -x \pmod{2^n}$;
- пакетираното и непакетираното двоично-десетично представяне;
- частите на числото с плаваща запетая – знак, характеристика и мантиса — и ролята на отклонението при характеристиката;
- какво означава нормализирано число и защо старшият бит на мантисата може да не се записва;
- ролята на кодовите таблици, ASCII и Unicode;

- основните блокове на процесора – регистри, АЛУ и блок за управление;
- предназначението на програмния брояч, регистъра на инструкциите, MAR и MBR;
- флаговете за знак, нула, пренос, препълване и прекъсване;
- задачите и входовете и изходите на блока за управление;
- съставът на машинната инструкция – opcode и указатели към операнди и следваща инструкция;
- трите шини — адресна, за данни и контролна — и ролята на контролната шина при връзката с паметта;
- понятието режим на адресиране и разликата между директно и недиректно адресиране;
- понятието микрооперация и връзката му с работата на блока за управление;
- фазите на изпълнение на инструкцията;
- изчисляването на адреси и извличането на операнди;
- какво е прекъсване и защо се обслужва след завършването на текущата инструкция;
- запазването на състоянието при прекъсване;
- ролята на програмния брояч и флаговете при изпълнение на преходи;
- идеята на конвейерната обработка и последователността от микрооперации.

Въпрос 10

Структура и йерархия на паметта. Сегментна и странична преадресация. Прекъсвания.

По конспекта: какво трябва да мога

Официално заглавие: „Структура и йерархия на паметта. Сегментна и странична преадресация. Система за прекъсване — *приоритети и обслужване*.“
Описание: структура на основната памет; йерархия кеш — основна — виртуална памет — §10.1, стр. 146.
Странична преадресация: каталог на страниците, описател, стратегии за подмяна на страниците — §10.2, стр. 149.
Сегментна преадресация: селектор, дескриптор, таблици и регистри — §10.3, стр. 151.
Система за прекъсване: видове прекъсвания, структура и обработка, конкурентност и приоритети, контролери — §10.4, стр. 153.
Литература: [5], [45], [50].

10.1 Структура и йерархия на паметта

10.1.1 Основни понятия

Паметта е частта от компютърната система, в която се съхраняват програми и данни. Най-малката единица информация е битът. Той може да има една от две стойности: 0 или 1. Паметта е организирана като множество от клетки, подредени последователно. На всяка клетка е съпоставен адрес, чрез който тя се посочва при операция за четене или запис.

Ако паметта съдържа n клетки, адресите им са числата от 0 до $n-1$. Съседните клетки имат съседни адреси. Всички клетки съдържат еднакъв брой битове. Ако една клетка съдържа k бита, тя може да представя 2^k различни комбинации. В съвременните процесори широко се използва 8-битова клетка, наречена байт. Байтовете се групират в думи, като много операции се извършват върху цели думи.

Дефиниция 10.1 (Адрес)**.** Адресът е номерът, чрез който се идентифицира клетка или друга адресируема единица на паметта.

Паметта може да се разглежда и според начина, по който се извършва достъпът до нея. Основната памет се реализира чрез памет с произволен достъп — RAM (*Random Access Memory*). При нея всяка клетка може да бъде избрана непосредствено по адрес, а времето за операцията не зависи съществено от местоположението на данните. Оперативната памет съдържа инструкциите, които процесорът изпълнява, и данните, върху които тези инструкции работят.

RAM е променлива памет: при загуба на захранване тя губи съдържанието си. Според начина на реализация източникът различава два основни типа:

- SRAM (*Static RAM*) — използва се за кеш памет;
- DRAM (*Dynamic RAM*) — използва се като основна памет.

SRAM има по-проста логика на достъп в сравнение с необходимостта от периодично опресняване при DRAM. При DRAM електрическият сигнал трябва периодично да се възстановява чрез опресняване на паметта.

† **Допълнение 10.2 (Терминологична бележка).** В предоставения материал на едно място е посочено, че вътрешната памет се достъпва чрез входно-изходен модул и е отбелязано „SRAM“. Това противоречи на разграничението, направено в същия материал: кешът е бърза вътрешна памет, а основната памет е RAM, обикновено реализирана чрез DRAM. В изложението се използва последното, съгласувано разграничение.

10.1.2 Принцип на йерархията

Паметта е организирана в йерархия според два основни критерия: обем и скорост на достъп. Колкото по-близо до процесора се намира дадено ниво, толкова по-бързо обикновено се осъществява достъпът до него, но капацитетът му е по-малък. По-отдалечените нива имат по-голям обем, но достъпът е по-бавен.

Причината за съществуването на йерархията е, че памет с достатъчно голям обем и едновременно с това със скоростта на процесора би била трудна за реализация в необходимия обем. За да бъде паметта достъпвана със скоростта на процесора, бързата памет трябва да бъде разположена непосредствено в процесора или много близо до него. Това ограничава нейния капацитет.

Йерархията се основава на предположението, че за кратък период от време програмата използва малък брой адреси. Най-често използваните данни се задържат в по-бързите нива, а останалите се съхраняват в по-бавни и по-големи нива.

Дефиниция 10.3 (Времева локалност)**.** Времевата локалност е свойството, според което данни, използвани веднъж, вероятно ще бъдат използвани отново в близък момент.

Дефиниция 10.4 (Пространствена локалност)**.** Пространствената локалност е свойството, според което при достъп до даден адрес е вероятно скоро да бъдат използвани и адреси, разположени около него.

Времевата локалност обосновава задържането на скоро използваните данни в по-високите нива на йерархията. Пространствената локалност обосновава пренасянето не само на търсената клетка или дума, но и на съседни данни.

10.1.3 Кеш памет

Кешът е малко количество бърза памет, разположено в процесора или непосредствено до него. Типичното време за достъп до него е от порядъка на един такт. Основната идея е най-често използваните инструкции и данни да бъдат достъпни без непосредствено обръщение към основната памет.

При заявка от процесора първо се проверява кешът. Ако търсените данни се намират там, достъпът се обслужва от кеша. Ако не се намират, се извършва търсене в основната памет, след което данните могат да бъдат поставени в кеша.

Кеш паметта се разглежда на няколко нива:

- кеш от ниво L1 — най-бърз и с най-малък капацитет; посоченият в източника типичен диапазон е от 8 до 128 килобайта;
- кешове от нива L2 и L3 — по-бавни от L1, но с по-голям капацитет.

Кешът L1 обикновено се разделя на кеш за инструкции и кеш за данни. Кешът за инструкции съхранява инструкции, необходими за изпълняваната операция, а кешът за данни съхранява данните, върху които операцията работи. В разглежданата организация всяко процесорно ядро разполага със собствени кешове L1 и L2, докато кешът L3 се споделя между ядрата.

Забележка 10.5. В източника е използвано означението $1\text{KB} = 1024\text{B}$. Това означение се запазва в смисъла на учебния материал.

10.1.4 Основна и виртуална памет

Основната памет е паметта, до която процесорът може да се обръща непосредствено при изпълнение на програми. Тя съдържа текущо необходимите инструкции и данни. Кешът ускорява достъпа до често използваните елементи, но не замества основната памет като място за изпълняваните програми.

С нарастването на програмите се появява необходимост от адресни пространства, по-големи от реално наличната основна памет. Въвежда се принципът на виртуалната памет: програмата работи с виртуално адресно пространство, което може да бъде по-голямо от физическото адресно пространство на основната памет. Част от информацията може да се съхранява във външна памет и да се адресира така, сякаш е част от основната памет.

Дефиниция 10.6 (Виртуално и физическо адресно пространство)**.** Виртуалното адресно пространство е множеството от адреси, с които работи програмата. Физическото адресно пространство е множеството от адреси на реалните места в основната памет.

Дефиниция 10.7 (Адресна транслация)**.** Разделянето между виртуално и физическо пространство позволява на програмата да използва логическа организация, независима от конкретното разположение на данните в основната памет. Преобразуването от виртуален към физически адрес се нарича преадресация или транслация.

10.2 Странична преадресация

10.2.1 Страници и физически блокове

При страничната организация виртуалното адресно пространство се разделя на фрагменти с еднакъв размер, наречени страници. Физическото адресно пространство се разделя на фрагменти със същия размер, наречени физически блокове.

Виртуалният адрес има два компонента:

$$\text{виртуален адрес} = (\text{номер на виртуална страница}, \text{отместване}).$$

Номерът на виртуалната страница се преобразува в номер на физически блок. Отместването в страницата не се променя. Следователно физическият адрес има вида:

$$\text{физически адрес} = (\text{номер на физически блок}, \text{отместване}).$$

Дефиниция 10.8 (Странична преадресация)**.** Страничната преадресация е преобразуване, при което виртуалната страница се съпоставя с физически блок, а отместването в страницата се запазва.

10.2.2 Каталог на страниците и описател

Съответствието между виртуалните страници и физическите блокове се поддържа от таблица, наричана каталог или таблица на страниците. За всяка виртуална страница има запис, който описва текущото ѝ състояние.

Записът може да съдържа:

- номер на физическия блок, в който се намира страницата;
- бит за наличност;
- бит за промяна, наричан още *dirty* или *modified* бит.

Битът за наличност показва дали страницата се намира в основната памет. Ако стойността му е 1, страницата е разположена в основната памет. Битът за промяна показва дали съдържанието на страницата е било изменено след зареждането ѝ.

Дефиниция 10.9 (Описател на страница)**.** Описателят на страница е записът в каталога на страниците, който съдържа информация за състоянието и разположението на съответната виртуална страница.

Най-скоро използваните записи от каталога могат да се кешират, за да се ускори транслацията. Така при повтарящи се обръщения към вече използвани страници не е необходимо всеки път да се извършва пълно търсене в каталога.

10.2.3 Обработка на достъп до страница

При заявка за достъп процесът на транслация протича концептуално по следния начин:

1. Виртуалният адрес се разделя на номер на виртуална страница и отместване.
2. Проверява се записът за виртуалната страница в каталога.
3. Ако страницата е налична в основната памет, номерът й се преобразува в номер на физически блок.
4. Към номера на физическия блок се добавя непроменето отместване.
5. Ако страницата не е налична, тя се зарежда от външната памет.
6. След зареждането се извършва отново преобразуване към физически адрес.

Ако в основната памет няма свободен физически блок, трябва да бъде избрана друга страница за подмяна. Ако избраната страница е била променена, нейното актуално съдържание трябва да бъде записано обратно във външната памет. Това се определя чрез бита *dirty* или *modified*.

† **Допълнение 10.10 (Терминологична бележка).** В източника заглавието на този раздел е изписано като „статична преадресация“, но описанието разглежда разделяне на виртуалното пространство на страници, каталог на страниците и подмяна на страници. Поради това съдържателно става дума за странична преадресация.

10.2.4 Стратегии за подмяна на страници

Когато няма свободно място, се избира страница, която да бъде извадена от основната памет. В предоставения материал е посочена стратегията FIFO (*First In, First Out*).

Дефиниция 10.11 (FIFO подмяна)**.** При FIFO се подменя страницата, която е заредена най-отдавна, независимо от това кога последно е била използвана.

Стратегията разглежда времето на зареждане на страниците. Най-старо заредената страница се избира първа за изваждане. Ако тя е променена, първо се записва във външната памет, след което физическият блок може да бъде използван за новата страница.

LRU (Least Recently Used). Подменя страницата, която не е използвана най-дълго време. Идеята следва локалността на обръщенията, но точното поддържане на реда на всички достъпи е скъпо и обикновено се приближава чрез хардуерни битове и периодично отчитане.

Оптимална стратегия на Белади. Подменя страницата, чиято следваща употреба е най-далеч в бъдещето. Тя дава минимален брой пропуски за дадена последователност, но не е реализируема онлайн, защото изисква знание за бъдещите обръщения. Използва се като еталон.

Clock, или second chance. Физическите блокове се разглеждат циклично. Ако битът за обръщение на текущата страница е 1, той се изчиства и указателят продължава; първата страница с бит 0 се избира за подмяна. Това е практическо приближение на LRU с малък разход.

FIFO е проста, но не отчита употребата и може да прояви аномалията на Белади: повече физически блокове понякога водят до повече пропуски. LRU и Clock използват информация за скорошна употреба, а оптималната стратегия служи само за сравнение.

10.3 Сегментна преадресация

10.3.1 Сегменти и адреси

Дефиниция 10.12 (Сегмент)**.** Освен на страници виртуалното пространство може да бъде организирано на сегменти. За разлика от страниците, сегментите са с променлива дължина и могат да се припокриват. Всеки сегмент представлява собствено адресно пространство и се описва чрез базов адрес и дължина.

Адресът при сегментна организация има два компонента:

$$\text{сегментен адрес} = (\text{селектор на сегмент}, \text{отместване}).$$

Селекторът избира сегментния дескриптор, а отместването посочва позицията вътре в избрания сегмент.

Дефиниция 10.13 (Сегментен дескриптор)**.** Сегментният дескриптор е запис, който описва сегмент чрез началния му адрес, размера му и битове за състояние.

Дефиниция 10.14 (Сегментен селектор)**.** Сегментният селектор е стойност, която избира дескриптор на сегмент от глобална или локална дескрипторна таблица.

Селекторът съдържа индекс, който показва позицията на дескриптора в дескрипторната таблица, както и бит, показващ дали трябва да се използва глобалната или локалната таблица. За да бъде използван, селекторът се зарежда в сегментен регистър.

10.3.2 Глобална и локална дескрипторна таблица

Дескрипторните таблици съдържат сегментни дескриптори. Всеки дескриптор определя началния адрес на сегмента, неговата дължина и съответните битове за състояние.

Глобалната дескрипторна таблица е единствена за системата. Тя се използва основно за дескриптори на системни сегменти, но може да съдържа и дескриптори на локални дескрипторни таблици.

Локалната дескрипторна таблица се свързва с отделен процес и се използва за дескриптори на приложни сегменти. В даден момент може да се използва точно една локална дескрипторна таблица.

Съответният сегментен селектор указва коя от двете таблици да бъде използвана. Индексът в селектора определя конкретния дескриптор в избраната таблица.

10.3.3 Алгоритъм на сегментната преадресация

Получаването на адреса чрез сегментен селектор и отместване се извършва в следната последователност:

1. Селекторът се зарежда в сегментен регистър.
2. От селектора се определя дали дескрипторът се търси в глобалната или локалната дескрипторна таблица.
3. Индексът от селектора се използва за намиране на дескриптора.
4. От дескриптора се получават базовият адрес и дължината на сегмента.
5. Проверява се дали зададеното отместване попада в границите на сегмента.
6. Към базовия адрес се прибавя отместването.

Ако дескрипторът пази дължина L, допустимото условие е

$$0 \leq \text{отместване} < L.$$

Ако архитектурата пази включена горна граница (limit), условието е отместване $\leq$ limit. При нарушение се сигнализира грешка на сегментната защита. При успешно преминаване на проверката се получава линеен адрес:

$$\text{линеен адрес} = \text{базов адрес} + \text{отместване}.$$

Дефиниция 10.15 (Линейно адресно пространство)**.** Линейното адресно пространство е пространството, в което се получава адресът след сегментната преадресация.

Полученият линеен адрес може да се обработи по два начина:

- линейното адресно пространство се изобразява директно върху физическото адресно пространство;

- линейният адрес се разглежда като виртуален и се подава към блока за странична преадресация.

Във втория случай системата комбинира двете схеми: първо сегментният селектор и отместването дават линеен адрес, а след това линейният адрес се преобразува във физически чрез странична преадресация.

10.4 Система за прекъсване

10.4.1 Предназначение на прекъсванията

Прекъсването е механизъм, при който процесорът временно прекратява нормалното изпълнение на текущата програма, съхранява необходимата информация и преминава към предварително определена програма за обслужване на настъпило събитие.

Дефиниция 10.16 (Прекъсване)**.** Прекъсването е събитие, което променя временно последователността на изпълнение, като процесорът запазва текущото си състояние и предава управлението на процедура за обработка.

След приключване на обработката управлението се връща към прекъснатата програма и тя продължава изпълнението си от мястото, на което е била прекъсната.

Системата за прекъсване позволява във всеки момент да бъде регистрирано настъпило събитие. Тя е особено полезна при обмен на информация с бавни входно-изходни устройства и при сигнализация за особени състояния на процесора. Без прекъсванията процесорът би трябвало периодично да проверява флагове, за да установи дали е настъпило събитие.

10.4.2 Векторна таблица и обработка

На всяко прекъсване се съпоставя определен номер. За всеки вид прекъсване съществува специална програма, която се изпълнява при възникването му. В основната памет се намира фиксирана област, наречена таблица на векторите на прекъсванията.

Всеки вектор съдържа указател към програмата за обработка на прекъсването и информация за състоянието, необходимо при преминаването към нея. Номерът на прекъсването се използва за намиране на съответния вектор.

Обработката може да бъде представена чрез следната последователност:

1. Възниква прекъсване.
2. Процесорът прекратява временно изпълнението на текущата програма.
3. Запазват се програмният брояч и регистърът на състоянието в стека.
4. Чрез номера на прекъсването се намира съответният вектор.
5. От вектора се получава новото състояние и адресът на програмата за обслужване.

6. Изпълнява се процедурата за обработка.
7. Процедурата завършва с инструкция за връщане от прекъсване.
8. От върха на стека се възстановява предишното състояние и се продължава прекъснатата програма.

Запазването на програмния брояч е необходимо, за да се знае мястото, към което трябва да се върне изпълнението. Запазването на регистъра на състоянието позволява възстановяване на предишното състояние на процесора. Стекът осигурява място за съхраняване на тази информация, включително при възможност за вложени прекъсвания.

Дефиниция 10.17 (Обработчик на прекъсване)**.** Обработчикът на прекъсване е специалната програма, която се изпълнява за обслужване на конкретен вид прекъсване.

10.4.3 Видове прекъсвания

В предоставения материал са разграничени следните видове прекъсвания:

- прекъсвания при машинна грешка — възникват при грешка в апаратурата и са с висок приоритет;
- входно-изходни прекъсвания — активират се в резултат от изпълнение на входно-изходна операция;
- външни прекъсвания — свързани са с външен елемент;
- програмни прекъсвания — синхронни са с изпълняваната програма;
- програмно активирани прекъсвания — текущата програма издава специална инструкция за предизвикване на прекъсване.

Според възможността да бъдат временно игнорирани прекъсванията се разделят на маскируеми и немаскируеми.

Дефиниция 10.18 (Маскируемо прекъсване)**.** Маскируемото прекъсване може временно да бъде игнорирано от процесора чрез механизма за маскиране.

Дефиниция 10.19 (Немаскируемо прекъсване)**.** Немаскируемото прекъсване не може да бъде игнорирано и трябва да бъде обработено.

Немаскируемите прекъсвания имат приоритет над маскируемите. В текста са разграничени и прекъсванията, възникващи от инструкцията `INT`, прекъсванията при особени случаи и прекъсванията в стъпков режим, използвани при дебъгване.

10.4.4 Конкурентност и приоритети

Възможно е едновременно да възникнат няколко прекъсвания. За да се определи кое да бъде обслужено първо, се въвежда приоритет. Той отразява важността на настъпилото събитие.

По време на обслужване на прекъсване текущата програма или по-нископриоритетно прекъсване може да бъде прекъснато от прекъсване с по-висок приоритет. Това създава възможност за вложени прекъсвания. В материала е отбелязано, че обикновено прекъсванията с по-малки номера имат по-висок приоритет.

Когато няколко прекъсвания са възникнали едновременно, посоченият ред на разглеждане е:

1. прекъсвания от инструкцията `INT` и прекъсвания при особени случаи;
2. прекъсване в стъпков режим за дебъгване;
3. немаскируеми прекъсвания;
4. маскируеми прекъсвания.

Този ред трябва да се разбира като схема за определяне на приоритет при конкурентно възникнали събития. Конкретното обслужване започва след запазване на текущото процесорно състояние и намиране на съответния вектор.

Забележка 10.20. Източникът представя реда на приоритетите в учебна, обобщена форма. Когато се разглежда конкретен процесор, точната реализация на приоритетите и имената на отделните източници зависят от неговата система за прекъсване.

10.4.5 Контролер на прекъсванията

Контролерът на прекъсванията е специално логическо устройство, което управлява заявките за прекъсване. Хардуерните устройства подават заявки по линии IRQ (*Interrupt Request*). Контролерът приема тези заявки, определя приоритетите им и изпраща към процесора подходяща заявка за прекъсване.

Дефиниция 10.21 (Контролер на прекъсванията)**.** Контролерът на прекъсванията е логическо устройство, което приема заявки от външни устройства, подрежда ги според приоритет и управлява подаването им към процесора.

Контролерът предотвратява хаотичното подаване на множество заявки към процесора. Той определя кое устройство да комуникира с процесора и в какъв ред. Това е особено важно при устройства, които се нуждаят от бърза реакция, за да не бъде загубена информация.

† **Допълнение 10.22 (Терминологична бележка).** В предоставения текст е срещано означението „IPQ“ във връзка с бързата реакция при устройствата. В останалата част са използвани линиите IRQ, които са стандартното означение за заявки за прекъсване. Поради липса на допълнително уточнение „IPQ“ не се разглежда като отделен механизъм.

10.5 Обобщена връзка между преадресация, памет и прекъсвания

Сегментната и страничната преадресация могат да се използват последователно. При сегментната преадресация селекторът избира дескриптор, дескрипторът дава базов адрес и дължина, а сборът от базовия адрес и отместването образува линеен адрес. Ако линейното адресно пространство е виртуално, този адрес се обработва допълнително чрез странична преадресация.

При страничната преадресация линейният или виртуалният адрес се разделя на номер на страница и отместване. Каталогът на страниците определя физическия блок, а отместването се запазва. При неналична страница се налага зареждане от външната памет и евентуална подмяна на друга страница.

Тези операции могат да породят събития, които изискват намеса на операционната система. Например при липсваща страница трябва да се изпълни специална обработка за зареждане и евентуално записване на страница. По-общо, прекъсванията осигуряват механизъм за предаване на управлението към специална процедура при входно-изходни събития, машинни грешки и други състояния.

10.6 Изпитен фокус

За устно възпроизвеждане трябва да се познават:

- понятието за адресируема клетка, адрес, байт и дума;
- предназначението на RAM, различията между SRAM и DRAM и ролята на кеша;
- йерархията кеш–основна памет–виртуална памет;
- времевата и пространствената локалност;
- виртуалното и физическото адресно пространство;
- представянето на виртуален адрес като номер на страница и отместване;
- каталогът на страниците, описателят, битът за наличност и *dirty* битът;
- алгоритъмът на страничната преадресация;
- необходимостта от подмяна на страница и стратегията FIFO;
- съставът на сегментния адрес — селектор и отместване;
- предназначението на сегментния селектор, сегментния регистър и дескриптора;
- ролята и различията между глобалната и локалната дескрипторна таблица;
- изчисляването на линейния адрес като базов адрес плюс отместване;
- проверката за излизане извън дължината на сегмента;

- възможността линейният адрес да се преобразува допълнително чрез странична преадресация;
- определението и предназначението на прекъсването;
- таблицата на векторите и съдържанието на вектора;
- последователността: прекъсване, запазване на състоянието, намиране на вектор, обработчик и връщане от прекъсване;
- видовете прекъсвания и разграничението между маскируеми и немаскируеми;
- конкурентността на прекъсванията, вложените прекъсвания и приоритетите;
- ролята на контролера на прекъсванията и линиите IRQ.

Въпрос 11

Файлова система. Функции, структура и реализация.

По конспекта: какво трябва да мога

Описание: файлове и основни атрибути; единно пространство от именувани обекти; структура, физическо представяне, точки на монтиране; видове обекти — файлове, директории, устройства, информационни канали; основни функции — достъп, изолация и защита, управление на правата — §11.1, стр. 159.
Реализация: кратък обзор на ext2/ext3 — §11.2, стр. 159.
Ускоряване и надеждност: кеширане на вход/изхода, отлагане на записа, алгоритъм на асансьора; журнална файлова система, RAID1 и RAID5 — §11.3, стр. 160.
POSIX функции: `open`, `close`, `read`, `write`, `lseek`, включително частичен вход/изход — §11.4, стр. 160.
Задача 1: извличане на интервали от двоичен файл — схема в §11.5.1, пълна програма в §11.6, стр. 162.
Задача 2: сортиране на файл от 8-битови числа — §11.5.2, стр. 161.
Задача 3: прилагане на двоична кръпка — §11.5.3, стр. 161.

Статус на покритието: изрично се оценява скорост, пропорционална на реално обработваните данни. Задача 1 е дадена като пълна програма; задачи 2 и 3 са схеми, не програми. Първичните източници за този въпрос липсват в корпуса — главата следва официалната анотация и подлежи на проверка по литературата.
Литература: [42, четвърта глава], [48], [49, глави 10–12].

† **Допълнение 11.1 (Статус на източниците).** В наличния корпус липсват първичните материали за този въпрос. Главата следва подробната официална анотация и е сверена с предоставения сравнителен конспект. Преди окончателно академично издание тя трябва да се провери по посочената литература: [42, четвърта глава], [48], [49, глави 10–12].

11.1 Файлове и файлова система

Дефиниция 11.2 (Файл). Файлът е именуван информационен обект, чието съдържание се пази устойчиво и се достъпва като последователност от байтове или записи.

Операционната система свързва името на файла с метаданни и с физическите блокове, в които е разположено съдържанието. Типични атрибути са тип, размер, собственик, група, права за достъп и времена за последна промяна на съдържанието, метаданните и достъпа. Наличието на отделно време на създаване зависи от конкретната файлова система.

Дефиниция 11.3 (Файлова система). Файловата система е абстракция и реализация за именуване, организиране, съхраняване, намиране, защита и споделяне на информационни обекти.

Дефиниция 11.4 (Видове обекти и директория). В Unix-подобните системи обектите образуват единно дърво с корен `/`. В него участват обикновени файлове, директории, символни връзки, специални файлове за устройства, именувани програмни канали (FIFO) и сокети. Директорията съпоставя име с обект на файловата система; самото съдържание на обикновения файл не съдържа името му.

Дефиниция 11.5 (Монтиране). Отделна файлова система се включва в дървото чрез *монтиране* върху съществуваща директория — точка на монтиране. Така различни устройства и реализации се виждат през едно пространство от пътища.

Изолацията и защитата се основават на идентичността на процеса и метаданните на обекта. Класическият Unix модел различава права за четене, запис и изпълнение за собственик, група и останали. За директорията правото за изпълнение означава преминаване и търсене по имена, а правото за запис — промяна на записите в нея.

11.2 Физическо представяне и ext2/ext3

Дискът се дели на блокове. Файловата система пази служебни структури за свободните и заетите блокове, метаданните на файловете и записите в директориите. При ext2 основни елементи са:

- *superblock* с общи параметри и състояние на файловата система;
- групи от блокове с битови карти за свободни блокове и inode-и;
- таблица от inode структури;
- блокове с данни и директории.

Дефиниция 11.6 (Inode)**.** Inode е структура с метаданните на обекта и указатели към блоковете на съдържанието му. Името се пази в запис на директория, който сочи към номер на inode.

Малките файлове се адресират чрез преки указатели, а по-големите — и чрез единично, двойно и тройно косвени блокове с указатели. Твърдата връзка е допълнителен запис на директория към същия inode; символичната връзка е отделен обект, който съдържа път.

Дефиниция 11.7 (Журнална файлова система)**.** ext3 развива ext2 чрез журнал. Преди критични промени те се записват като транзакция в журнала, което позволява след срив файловата система да възстанови съгласувано състояние без пълно сканиране на всички блокове. Режимът на журнализиране определя дали се журнализират само метаданните или и потребителските данни.

11.3 Ускоряване и надеждност

Дефиниция 11.8 (Кеш, отложен запис и алгоритъм на асансьора)**. Кеширане.** Често използвани блокове и файлови страници се пазят в основната памет. Четенето напред (read-ahead) зарежда вероятно нужни следващи блокове.

Отложен запис. Промените първо се натрупват като замърсени кеширани страници и по-късно се записват групово. Това ускорява работата, но при срив незаписаните данни могат да се загубят; `fsync` изисква устойчиво записване на съответните промени.

Алгоритъм на асансьора. При въртящ диск заявките се подреждат подобно на SCAN: главата обслужва позиции в едната посока, после сменя посоката. Целта е по-малко механично преместване; ползата не е същата при SSD.

Дефиниция 11.9 (RAID1 и RAID5)**.** RAID използва няколко диска като едно блоково устройство. RAID1 огледално дублира данните и понася отказ на един диск във всяка огледална двойка. RAID5 разпределя данните и паритета между поне три диска и обичайно понася отказ на един диск, но записът изисква обновяване на данни и паритет. RAID подобрява наличността, но не е резервно копие: не пази от изтриване, повреда от софтуер или загуба на целия масив.

11.4 POSIX функции за файлове

```
int     open(const char *path, int flags, ...);
int     close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t   lseek(int fd, off_t offset, int whence);
```

`open` връща файлов дескриптор или -1. Флаговете задават например четене, запис, създаване и съкращаване. `read` връща броя реално прочетени байтове, 0 при край на файл и -1 при грешка. `write` също може да запише по-малко от поисканото, затова надеждната програма повтаря операцията до обработване на целия буфер. Прекъсване със сигнал може да даде `EINTR` и да изисква повторение.

`lseek` променя текущото байтово отместване спрямо началото, текущата позиция или края. Тя не извършва вход/изход и не е приложима към всеки тип дескриптор, например към обикновен програмен канал. Всички резултати трябва да се проверяват, а отворените дескриптори да се затварят и при грешка.

11.5 Типови задачи

11.5.1 Извличане на интервали от двоичен файл

Файлът `f1` съдържа двойки $\langle x_i, y_i \rangle$ от `uint32_t`, а `f2` — масив от такива числа. За всяка двойка се проверява, че интервалът е в границите, позицията се задава на

$$x_i \cdot \text{sizeof}(\texttt{uint32_t}),$$

и се прехвърлят точно y_i числа към `f3` на блокове. Може да се използва `lseek` и `read` или позиционно четене. Не трябва да се обхождат пропуснатите части на `f2`; времето е пропорционално на броя двойки и размера на изхода. Проверяват се препълване при умножението, частично четене и преждевременен край на файла.

11.5.2 Сортиране на файл от 8-битови числа

Понеже има само 256 възможни стойности, се използва броене:

1. входът се чете последователно на блокове;
2. за всеки байт се увеличава един от 256 брояча с достатъчно широк тип;
3. в изхода се записват последователно съответният брой нули, единици и т.н. до 255.

Така времето е $O(n+256) = O(n)$, а допълнителната памет е $O(256)$ независимо от размер до 2 GB. Изходът също се записва на блокове, а не байт по байт.

11.5.3 Прилагане на двоична кръпка

Първо `f1` се копира в `f2`. Всеки запис в `patch` е тройка

$$(\texttt{uint16_t offset}, \texttt{uint8_t old}, \texttt{uint8_t new}).$$

Тройките се обработват последователно. За всяка се проверява, че отместването съществува и текущият байт в работното копие `f2` е `old`; тогава той се заменя с `new`. При несъответствие, непълна тройка или I/O грешка програмата прекратява по контролиран начин.

† **Допълнение 11.10 (Приета интерпретация: повторено отместване).** Официалната формулировка не казва какво става, когато две тройки сочат едно и също отместване. Тук е приета следната интерпретация и тя трябва да се *каже на глас* при отговор: сравнението е винаги с *текущото* състояние на работното копие `f2`, а не с неизменния `f1`. Следствие: при тройки (o, a, b) и (o, b, c) и двете се прилагат и байтът става c; при (o, a, b) и (o, a, c) втората се проваля, защото байтът вече е $b \neq a$. Другата интерпретация — сравнение винаги с `f1` — е също защитима, но дава различно поведение и трябва да бъде обявена преди решаването, а не след това.

Двоичният формат поставя и въпроси за размера и подредбата на числата по байтове. Ако файловете се обменят между различни платформи, форматът трябва изрично да зададе byte order; не бива сляпо да се разчита на представянето на C структура в паметта.

11.6 Пълна програма: извличане на интервали

Разделът по-горе описва решенията в схема. Схема и работеща програма са различни неща: разликата е в обработката на грешки, частичния вход/изход и затварянето на дескрипторите. По-долу задача 1 е написана докрай; другите две следват същата рамка — сменя се само тялото на цикъла.

```
/* Задача 1. f1 съдържа двойки <x, y> от uint32_t;
   f2 е масив от uint32_t. За всяка двойка в f3 се
   записват y числа от f2, започвайки от индекс x.
   Компилиране: cc -std=c11 -O2 -o extract extract.c   */
#include <errno.h>
#include <fcntl.h>
#include <inttypes.h>
#include <stdio.h>
#include <unistd.h>

#define CHUNK 1024

/* read може да върне по-малко от поисканото;
   EINTR не е грешка, а прекъсване.                */
static ssize_t read_full(int fd, void* buf, size_t n) {
  size_t done = 0;
  while (done < n) {
    ssize_t r = read(fd, (char*)buf + done, n - done);
    if (r < 0) {
      if (errno == EINTR) continue;
      return -1;
    }
    if (r == 0) break;                    /* край на файла */
    done += (size_t)r;
  }
  return (ssize_t)done;
```

```
}

/* write също може да запише частично --- това е
   най-често пропусканото място.                  */
static int write_full(int fd, const void* buf, size_t n) {
  size_t done = 0;
  while (done < n) {
    ssize_t w = write(fd, (const char*)buf + done, n - done);
    if (w < 0) {
      if (errno == EINTR) continue;
      return -1;
    }
    done += (size_t)w;
  }
  return 0;
}

int main(int argc, char** argv) {
  if (argc != 4) {
    fprintf(stderr, "употреба: %s f1 f2 f3\n", argv[0]);
    return 2;
  }

  int f1 = open(argv[1], O_RDONLY);
  int f2 = open(argv[2], O_RDONLY);
  int f3 = open(argv[3], O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (f1 < 0 || f2 < 0 || f3 < 0) {
    perror("open");
    return 1;
  }

  /* колко числа има в f2 */
  off_t size2 = lseek(f2, 0, SEEK_END);
  if (size2 < 0) {
    perror("lseek");
    return 1;
  }
  uint64_t count2 = (uint64_t)size2 / sizeof(uint32_t);

  int status = 0;
  uint32_t pair[2];
  ssize_t got;

  const ssize_t PAIR = (ssize_t)sizeof pair;

  while ((got = read_full(f1, pair, sizeof pair)) == PAIR) {
```

```
    uint64_t from = pair[0];
    uint64_t len  = pair[1];

    /* Проверката е написана така, че from + len да не прелее. */
    if (from > count2 || len > count2 - from) {
      fprintf(stderr, "интервал извън f2: [%" PRIu64
              ", %" PRIu64 ")\n", from, from + len);
      status = 1;
      break;
    }

    /* Позиционираме се; пропуснатите части на f2
       изобщо не се четат. */
    if (lseek(f2, (off_t)(from * sizeof(uint32_t)), SEEK_SET) < 0) {
      perror("lseek");
      status = 1;
      break;
    }

    uint32_t buf[CHUNK];
    while (len > 0) {
      size_t take  = len < CHUNK ? (size_t)len : CHUNK;
      size_t bytes = take * sizeof(uint32_t);
      if (read_full(f2, buf, bytes) != (ssize_t)bytes) {
        fprintf(stderr, "непълно четене от f2\n");
        status = 1;
        break;
      }
      if (write_full(f3, buf, bytes) != 0) {
        perror("write");
        status = 1;
        break;
      }
      len -= take;
    }
    if (status != 0) break;
  }

  if (got < 0) {
    perror("read");
    status = 1;
  } else if (status == 0 && got != 0) {
    fprintf(stderr, "непълна двойка в края на f1\n");
    status = 1;
  }
```

```
  if (close(f1) < 0 || close(f2) < 0 || close(f3) < 0) {
    /* грешка при close е загуба на данни */
    perror("close");
    status = 1;
  }
  return status;
}
```

Четирите решения, които отличават работеща програма от схема:

- `read_full` и `write_full` — частичен вход/изход и `EINTR` се обработват на едно място, вместо да се повтарят във всеки цикъл;
- проверката `len > count2 - from` вместо `from + len > count2` — втората прелива при големи стойности и пропуска невалиден интервал;
- `lseek` вместо четене на пропуснатите части — времето остава пропорционално на броя двойки плюс размера на изхода, а не на размера на `f2`;
- резултатът на `close` се проверява: при отложен запис грешката се проявява точно там.

† **Допълнение 11.11 (Обхват).** Другите две задачи (§11.5.2 и §11.5.3) са дадени като схеми, не като пълни програми. За сортирането на байтове това е достатъчно — схемата от три реда е целият алгоритъм и рамката по-горе се пренася дословно. За кръпката пълната програма изисква и разрешаването на въпроса с повторените отмествания, обявен по-горе.

11.7 Изпитен фокус

- Файловата система като единно именувано дърво; обекти, метаданни, права и точки на монтиране.
- Ролята на superblock, inode, записите в директориите и преките/косвените блокове при ext2; журналът при ext3.
- Кеш, read-ahead, отложен запис, алгоритъм на асансьора, RAID1 и RAID5; защо RAID не е backup.
- Семантика и проверка на резултатите на `open`, `close`, `read`, `write` и `lseek`, включително частичен вход/изход.
- Решение на трите типови задачи с време, пропорционално на реално обработваните данни.
- Пълна програма за задача 1 (§11.6): цикли за частичен `read`/`write`, обработка на `EINTR`, проверка без препълване, `lseek` вместо обхождане и проверка на `close`.
- Обявяване на приетата интерпретация при повторено отместване в задачата за кръпката, преди решаването.

Въпрос 12

Управление на процеси и междупроцесни комуникации.

По конспекта: какво трябва да мога

Примитиви: създаване на процес, изпълнение на програма, завършване, синхронизация със завършването на процеса-син — §12.1, стр. 167.
Права и идентичности: потребителски идентификатори на процес, групи процеси и сесия — §12.2, стр. 167.
Междупроцесна комуникация: сигнали и програмни канали — §12.3, стр. 168.
Междупроцесна комуникация: IPC пакетът на UNIX System V — обща памет, семафори, съобщения — §12.4, стр. 168.
Схема: родител–дете — §12.5.1, като пълна програма в §12.6, стр. 170.
Схема: производител–потребител — §12.5.2, стр. 170.
Типове задачи: задачи, съответни на съдържанието на въпроса.

Статус на покритието: схемата родител–дете е дадена като пълна програма; производител–потребител остава схема. Първичните източници липсват в корпуса — главата подлежи на проверка по литературата.
Литература: [21], [46].

† **Допълнение 12.1 (Статус на източниците).** В наличния корпус липсват първичните материали за този въпрос. Главата следва подробната официална анотация и е сверена с предоставения сравнителен конспект. Преди окончателно академично издание тя трябва да се провери по посочената литература [21] и [46].

12.1 Процес и основни примитиви

Дефиниция 12.2 (Процес)**.** Процесът е изпълняващ се екземпляр на програма заедно с адресното му пространство, регистрите, отворените файлови дескриптори, идентичността и останалото състояние, което ядрото управлява.

В Unix създаването и зареждането на програма са отделни операции.

Дефиниция 12.3 (Системни примитиви за процеси)**.** **`fork`.** Създава процес-син като логическо копие на родителя. И двата процеса продължават след извикването: в сина резултатът е 0, а в родителя — PID на сина. При грешка родителят получава -1. Реализацията обичайно използва copy-on-write.

`exec`. Семейството `exec` заменя програмния образ на текущия процес с нова програма. При успех не се връща; PID остава същият, а запазването на дескриптори зависи от флага close-on-exec.

`exit` и `_exit`. Завършват процеса със статус. `exit` изпълнява регистрираните потребителски обработчици и flush на C потоците, докато `_exit` пропуска тази потребителска финализация и предава завършването на ядрото. Ядрото все пак трябва да освободи ресурсите и да затвори дескрипторите, което може да изисква време. След неуспешен `exec` в дете често се използва `_exit`.

`wait`/`waitpid`. Родителят изчаква промяна на състоянието на дете и получава статуса му. Завършило, но неприбрано дете е zombie. Ако родителят не иска да блокира, `waitpid` позволява подходящи опции.

Класическият шаблон е: `fork`; детето подготвя дескрипторите и извиква `exec`; родителят продължава паралелно или извиква `waitpid`. Кодът винаги различава трите резултата на `fork`.

12.2 Права, групи и сесии

Дефиниция 12.4 (Идентичност на процес)**.** Всеки процес има реални и ефективни потребителски и групови идентификатори. Реалните UID/GID описват произхода, а ефективните обичайно участват в проверките за достъп. Допълнителни групи и механизми като set-user-ID променят конкретните правила, затова не е достатъчно да се гледа само PID.

Дефиниция 12.5 (Процесна група)**.** Процесната група е множество процеси с общ PGID, към което терминалът или друг процес може да адресира сигнал като към едно цяло.

Дефиниция 12.6 (Сесия)**.** Сесията е множество процесни групи. Обичайно обвивката създава отделна група за всяка задача и използва сесията и управляващия терминал за foreground/background управление.

Всяка процесна група принадлежи на точно една сесия. `setpgid` създава или променя принадлежността към процесна група при допустимите ограничения, а `getpgrp` връща текущата група. `setsid` създава нова сесия, нова процесна група и прави извикващия процес техен лидер; `getsid` връща идентификатора на сесията. Тези операции стоят в основата на управлението на задачи от командната обвивка.

12.3 Сигнали и програмни канали

Дефиниция 12.7 (Сигнал)**.** Сигналът е асинхронно известие за събитие. Процесът може да установи обработчик, да остави стандартното действие или за някои сигнали да ги игнорира. Сигналите `SIGKILL` и `SIGSTOP` не могат да бъдат прихващани, блокирани или игнорирани. Маска блокира временно доставянето на други сигнали; обработчикът трябва да използва само операции, безопасни в сигнален контекст. `SIGCHLD` уведомява родителя за промяна в състоянието на дете. При обичайното поведение родителят прибира статуса чрез `wait`/`waitpid`; при изрично игнориране на `SIGCHLD` или флаг `SA_NOCLDWAIT` завършилото дете не се оставя като чакащ zombie процес.

Дефиниция 12.8 (Програмен канал)**.** Програмният канал (pipe) е еднопосочен байтов поток с край за четене и край за запис. `pipe` връща два файлови дескриптора, които могат да се наследят през `fork`.

След `fork` всеки процес затваря неизползваните краища. Четенето връща край на файл едва когато всички дескриптори за запис са затворени; забравен записващ край може да причини безкрайно блокиране. Записите до определена малка граница са атомарни, но pipe не пази структура на произволно големи съобщения. Именуваният канал FIFO позволява несвързани процеси да отворят общ обект от файловата система.

12.4 UNIX System V IPC

System V IPC идентифицира обекти чрез ключове и системни идентификатори; правата им са отделни от обикновените файлови дескриптори.

12.4.1 Обща памет

Дефиниция 12.9 (Обща памет)**.** Общата памет позволява няколко процеса да прикачат един сегмент към адресните си пространства. Типичните операции са създаване/намиране, прикачване, откачване и управление чрез `shmget`, `shmat`, `shmdt` и `shmctl`. След прикачването обменът е бърз, защото няма копиране през ядрото при всяко обръщение, но общата памет сама по себе си не осигурява взаимно изключване или ред на достъп.

12.4.2 Семафори

Дефиниция 12.10 (Семафор)**.** Семафорът е целочислен синхронизационен обект, променян атомарно. Операцията P (wait/down) изчаква положителна стойност и я намалява, а V (signal/up) я увеличава и може да събуди чакащ процес. Двоичен семафор може да пази критична секция; броящ семафор представя наличен брой еднотипни ресурси. При System V `semget` създава или намира множество от семафори, `semop` прилага атомарно вектор от операции, а `semctl` управлява множеството. Неправилен ред на придобиване може да доведе до взаимно блокиране.

12.4.3 Опашки от съобщения

Дефиниция 12.11 (Опашка от съобщения)**.** Опашката пази отделни съобщения с тип и съдържание. `msgget` създава или намира опашка, `msgsnd` изпраща, `msgrcv` получава, а `msgctl` управлява жизнения цикъл и параметрите ѝ. Получателят може да избере следващото съобщение или съобщение от определен тип. За разлика от pipe границите между съобщенията се запазват. Капацитетът е ограничен и операциите могат да блокират или да се изпълнят неблокиращо.

System V обектите могат да останат в ядрото след приключване на процеса, докато не бъдат изрично премахнати. Затова програмата трябва да има ясен собственик и път за почистване.

12.5 Синхронизационни схеми

12.5.1 Родител и дете

За изпълнение на програма и получаване на резултата родителят създава дете с `fork`; детето извиква `exec`; родителят извиква `waitpid` и проверява дали детето е завършило нормално и какъв е статусът му. Ако между тях има pipe, краищата се създават преди `fork`, пренасочват се при нужда с `dup2`, а всички излишни копия се затварят преди чакането.

12.5.2 Производител–потребител

Нека общ кръгов буфер има N места. Използват се броящи семафори

$$empty = N, \qquad full = 0$$

и двоичен семафор

$$mutex = 1.$$

Производителят изпълнява $P(empty)$, $P(mutex)$, добавя елемент, после $V(mutex)$ и $V(full)$. Потребителят изпълнява $P(full)$, $P(mutex)$, премахва елемент, после $V(mutex)$ и $V(empty)$. Броящите семафори предотвратяват препълване и четене от празен буфер, а `mutex` пази индексите и съдържанието. Изчакването за свободно/заето място трябва да е преди заключването на `mutex`; обратният ред може да задържи критичната секция и да блокира другата страна.

12.6 Пълна програма: родител, дете и програмен канал

Схемата в §12.5.1 казва *кои* примитиви се използват. Работещата програма трябва да реши и четирите въпроса, които схемата премълчава: кой край на канала кой затваря, защо детето излиза с `_exit`, какво се прави при `EINTR` и как се чете статусът.

```
/* Родителят стартира външна програма в дете и чете
   изхода й през канал.
   Компилиране: cc -std=c11 -O2 -o run run.c        */
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main(void) {
  int fd[2];                 /* [0] за четене, [1] за писане */
  if (pipe(fd) < 0) {
    perror("pipe");
    return 1;
  }

  pid_t pid = fork();
  if (pid < 0) {
    perror("fork");
    return 1;
  }

  if (pid == 0) {              /* ------- дете ------- */
    /* четящият край не му трябва */
    if (close(fd[0]) < 0) _exit(127);
    if (dup2(fd[1], STDOUT_FILENO) < 0) _exit(127);
```

```
  /* дубликатът вече е stdout */
  if (close(fd[1]) < 0) _exit(127);
  execlp("ls", "ls", "-l", (char*)NULL);
  _exit(127);              /* стига се само при неуспех */
}

/* ---------- родител ---------- */
/* без това read никога не вижда край на файл */
if (close(fd[1]) < 0) {
  perror("close");
  return 1;
}

int status = 0;
char buf[4096];
for (;;) {
  ssize_t n = read(fd[0], buf, sizeof buf);
  /* детето е затворило канала */
  if (n == 0) break;
  if (n < 0) {
    if (errno == EINTR) continue;
    perror("read");
    status = 1;
    break;
  }
  /* изходът също може да се провали: пълен диск, счупена
     тръба, дескриптор само за четене */
  if (fwrite(buf, 1, (size_t)n, stdout) != (size_t)n) {
    perror("fwrite");
    status = 1;
    break;
  }
}
close(fd[0]);

/* буферът се изпразва тук, докато има кой да съобщи за
   грешката; при изход от main е вече късно */
if (fflush(stdout) != 0 || ferror(stdout)) {
  perror("stdout");
  status = 1;
}

int wstatus;
while (waitpid(pid, &wstatus, 0) < 0) {
  if (errno != EINTR) {
    perror("waitpid");
```

```
      return 1;
    }
  }

  if (WIFSIGNALED(wstatus)) {
    fprintf(stderr, "детето е прекратено от сигнал %d\n",
            WTERMSIG(wstatus));
    return 1;
  }
  if (!WIFEXITED(wstatus)) return 1;

  fprintf(stderr, "детето завърши с код %d\n",
          WEXITSTATUS(wstatus));
  /* успех само ако и детето, и изходът са наред */
  return (WEXITSTATUS(wstatus) == 0 && status == 0) ? 0 : 1;
}
```

Решенията, които се питат на изпита:

- **Затварянето на неизползваните краища не е чистота, а коректност.** Каналът дава край на файл при четене чак когато *всички* записващи дескриптори са затворени. Ако родителят задържи `fd[1]`, неговият `read` блокира вечно, макар детето отдавна да е завършило.
- **Детето излиза с `_exit`, не с `exit`.** След `fork` детето има копие на буферите на стандартната библиотека; `exit` би ги изчистило втори път и същият текст би се отпечатал два пъти. `_exit` завършва процеса без тези действия.
- **`waitpid` се повтаря при `EINTR`.** Пристигнал сигнал прекъсва чакането; това не е грешка и не означава, че детето е завършило.
- **Без `waitpid` детето остава *zombie***: записът в таблицата на процесите се пази, докато родителят не прочете статуса му.
- `execlp` се връща само при неуспех — затова редът след него е изход с код, а не продължение на програмата.
- **Записът към изхода се проверява точно както четенето.** `fwrite` връща броя записани елементи, а не признак за успех; при пълен диск, затворена тръба надолу по конвейера или дескриптор, отворен само за четене, тя записва по-малко и програмата трябва да го забележи. Пропуснато ли се, програмата връща 0, след като е загубила част от изхода — най-лошият възможен изход, защото извикващият скрипт продължава напред.
- **`fflush` се извиква изрично, преди да се докладва успех.** Стандартната библиотека буферира; ако изпразването се остави на изхода от `main`, грешката настъпва след последната възможност да бъде съобщена и кодът за връщане е вече определен. `ferror` улавя и грешка, отбелязана в по-ранно частично записване.

- **Кодът за връщане обединява двете условия.** Успех се докладва само когато детето е завършило с 0 *и* целият изход е стигнал до местоназначението си.

† **Допълнение 12.12 (Обхват).** Схемата производител–потребител (§12.5.2) е дадена със семафорни операции, но не като пълна програма: тя изисква и създаване на System V обектите, права, обработка на `EINTR` при `semop` и изрично премахване на обектите при изход, защото те надживяват процеса. Това остава да се напише.

12.7 Изпитен фокус

- Разлика между процес и програма; семантика и резултати на `fork`, `exec`, `exit`/`_exit`, `wait`/`waitpid`; zombie процес.
- Реални и ефективни UID/GID, процесни групи, сесии и управляващ терминал.
- Сигнали и маски; специалният статус на `SIGKILL` и `SIGSTOP`; `SIGCHLD`.
- Pipe/FIFO, наследяване и правилно затваряне на краищата.
- Обща памет, семафори и опашки от съобщения в System V IPC, включително основните семейства `shm*`, `sem*` и `msg*`, жизнен цикъл и права.
- Синхронизация родител–дете и схема производител–потребител с `empty`, `full` и `mutex`.
- Пълна програма родител–дете през канал (§12.6): ред на затваряне на краищата, `dup2`, `_exit` вместо `exit` в детето, повтаряне при `EINTR` и четене на статуса с `WIFEXITED`/`WEXITSTATUS`.
- Защо незатворен записващ край блокира `read` завинаги и защо липсващият `waitpid` оставя zombie процес.

Въпрос 13

Компютърни мрежи и протоколи. OSI модел. Маршрутизация. IPv4, IPv6, TCP, DNS.

По конспекта: какво трябва да мога

Описание: OSI модел — най-обща характеристика на нивата и съпоставяне с модела TCP/IP — §13.2, стр. 175.
Алгоритми: разпределена маршрутизация — с дистантен вектор и със следене на състоянието на линията — §13.3, стр. 178.
Адресация: IPv4 — класова и безкласова — §13.4, стр. 179.
Описание: основни характеристики на IPv6 — §13.5, стр. 181.
Процедура: трикратно договаряне при TCP — §13.6, стр. 182.
Описание: DNS — резолвинг на имената по IPv4 и IPv6 — §13.7, стр. 183.
Литература: [3], [35], [51].

13.1 Компютърни мрежи и протоколи

Дефиниция 13.1 (Компютърна мрежа и протокол)**.** Компютърната мрежа е съвкупност от свързани устройства, които обменят данни посредством комуникационни канали и съгласувани правила. Тези правила се наричат протоколи. Протоколът определя формата на съобщенията, реда на обмена им и действията, които извършват участващите устройства при различни ситуации.

Дефиниция 13.2 (Енкапсулация и декапсулация)**.** Комуникацията в мрежа обикновено се описва чрез слоеве. Всеки слой предоставя услуги на слоя над него и използва услуги от слоя под него. При изпращане на данни всеки слой добавя собствена служебна информация към получените от по-горния слой данни. Този процес се нарича енкапсулация. При получаване на данните служебната информация се отстранява последователно от съответните слоеве; процесът се нарича декапсулация.

13.2 OSI модел

13.2.1 Обща характеристика

Дефиниция 13.3 (OSI модел). OSI (*Open System Interconnection*) е референтен модел, който стандартизира начина, по който отворени системи могат да комуникират помежду си. Под отворена система се разбира система, която е подготвена за комуникация с други системи чрез стандартизирани интерфейси и протоколи.

Моделът има седем нива. При разделянето на комуникационните функции на нива са следвани няколко основни принципа:

- отделно ниво се създава, когато е необходимо различно ниво на абстракция;
- всяко ниво трябва да изпълнява точно определена функционалност;
- функциите трябва да бъдат подбрани така, че да позволяват използването на международно стандартизирани протоколи;
- границите между нивата трябва да ограничават информационния поток през интерфейсите;
- броят на нивата трябва да бъде достатъчно голям, за да не се смесват различни функции, но и достатъчно малък, за да не стане архитектурата ненужно сложна.

Дефиниция 13.4 (Услуга, интерфейс и протокол). OSI моделът разграничава ясно понятията услуга, интерфейс и протокол. Услугата описва какво предоставя дадено ниво на по-горното ниво. Интерфейсът определя начина, по който това се предоставя. Протоколът определя правилата за обмен между едноименното ниво на две комуникиращи системи.

13.2.2 Нива на OSI модела

Приложно ниво

Дефиниция 13.5 (Приложно ниво). Приложното ниво предоставя мрежови услуги на потребителските приложения. Клиентските приложения могат да заявяват услуги или информация, а сървърните приложения могат да се регистрират и да предоставят услуги в мрежата.

Това ниво представлява съвкупност от протоколи, които се използват директно от процесите. Като примери могат да се посочат HTTP и DNS. Приложното ниво не означава самото потребителско приложение, а протоколите и функциите, чрез които то осъществява мрежова комуникация.

Презентационно ниво

Дефиниция 13.6 (Презентационно ниво)**.** Презентационното ниво се занимава със синтаксиса и семантиката на предаваната информация. Обменяните структури от данни се описват абстрактно и се използва стандартно кодиране, така че системи с различни вътрешни представяния да могат да обменят информация.

Към това ниво се отнасят преобразуването на формата на данните, криптирането и декриптирането, както и компресирането и декомпресирането на информацията.

Сесийно ниво

Дефиниция 13.7 (Сесийно ниво)**.** Сесийното ниво управлява диалога между две комуникиращи програми. То организира обмена на данни и определя начина, по който протича комуникацията.

Разграничават се три типа диалог:

- двупосочен едновременен диалог;
- двупосочен алтернативен диалог;
- еднопосочен диалог.

Сесийното ниво може да използва синхронизиращи елементи. Те позволяват прекъснат диалог да бъде възстановен от мястото, до което е достигнал.

Транспортно ниво

Дефиниция 13.8 (Транспортно ниво)**.** Транспортното ниво улеснява ефективното и надеждно предаване на данни от край до край. То сегментира данните, получени от по-горните нива, и се грижи за тяхната правилна доставка.

В зависимост от използваната услуга транспортното ниво може да осигурява безгрешна и последователна доставка. Работата от край до край изолира горните нива от промените в хардуерната реализация на мрежата и ограничава влиянието на грешките върху приложенията.

Мрежово ниво

Дефиниция 13.9 (Мрежово ниво)**.** Мрежовото ниво отговаря за изпращането и получаването на пакети, наричани още дейтаграми, както и за тяхната доставка през мрежата. Основна негова функция е маршрутизацията – изборът на път, по който пакетът да достигне до мрежата на получателя.

На това ниво се реализират и логическата адресация на машините, комуникирането на пакетите между мрежи и действията, свързани с предотвратяване на претоварването. Функциите на мрежовото ниво обикновено се реализират програмно.

Канално ниво

Дефиниция 13.10 (Канално ниво)**.** Канално ниво осигурява обмена на данни между непосредствено свързани устройства, без да се интересува от начина, по който данните се преобразуват в електрически или други физически сигнали.

Типична функция е откриването и коригирането на грешки при предаването. Данните обикновено се обменят на порции с фиксиран формат, наречени кадри. Форматът на кадъра се определя от протокола на канално ниво. Реализацията на това ниво е апаратно-програмна.

Физическо ниво

Дефиниция 13.11 (Физическо ниво)**.** Физическото ниво предава битове по комуникационен канал. То обхваща техническите средства, които реализират предаването през физическата среда.

Основната му функция е кодиране и декодиране на сигналите, представящи двоичните цифри 0 и 1. Физическото ниво не се интересува от предназначението на битовете и от смисъла на предаваните данни.

13.2.3 Съпоставяне на OSI и TCP/IP

OSI е референтен модел, а не конкретна мрежова архитектура. Той не задава точно кои услуги и протоколи трябва да се използват във всяко ниво. Силата му е в ясното разделяне на функциите и в удобството му при описване и обсъждане на мрежовата комуникация.

TCP/IP моделът е изграден около стандартни протоколи, които са широко използвани. Разпределената му организация позволява маршрутизиращите протоколи да намерят алтернативен път при наличие на резервна свързаност, но възстановяването зависи от топологията, използваните протоколи и времето им за сходимост. За функционирането му е необходимо сравнително малко централизирано управление.

И двата модела използват слоеве и енкапсулация. Съществената разлика е, че OSI прави ясно разграничение между услуга, интерфейс и протокол, докато TCP/IP не поддържа това разграничение в същата степен. В OSI протоколите са по-добре обособени и полесно могат да бъдат заменяни, тъй като моделът е общ. TCP/IP е построен на базата на конкретни протоколи и отделните му части са по-тясно свързани с тях.

Презентационното и сесийното ниво на OSI обикновено не се представят като самостоятелни нива в TCP/IP. Техните функции се включват в приложното ниво. Опростеното съпоставяне е:

OSI	TCP/IP
Приложно	Приложно
Презентационно	Приложно
Сесийно	Приложно
Транспортно	Транспортно
Мрежово	Интернет
Канално	Мрежов достъп
Физическо	Мрежов достъп

13.3 Разпределена маршрутизация

Дефиниция 13.12 (Разпределена маршрутизация)**.** Разпределената маршрутизация позволява на всички маршрутизатори в мрежата да взимат собствени решения за маршрутизирането, като следват формален протокол. Всеки маршрутизатор поддържа информация за достижимите дестинации и за предпочитаната изходна линия.

13.3.1 Маршрутизация с дистантен вектор

Дефиниция 13.13 (Дистантен вектор)**.** При маршрутизацията с дистантен вектор всеки маршрутизатор поддържа таблица, в която за всяка дестинация се записват най-доброто известно разстояние и връзката, която трябва да се използва.

Типичен ред от таблицата съдържа:

- дестинацията;
- предпочитаната изходна линия или следващия маршрутизатор;
- оценка на разстоянието до дестинацията.

Метриката може да бъде брой стъпки, пропускателна способност или друга величина, чрез която се сравняват маршрутите. На определен интервал всеки маршрутизатор изпраща до съседите си съдържанието на маршрутната таблица. След получаване на таблица от съсед маршрутизаторът преизчислява собствените си оценки и при необходимост променя маршрутите.

Нека $c(x, v)$ е цената на пряката връзка от маршрутизатор x до негов съсед v, а $D_v(y)$ е оценката на v за дестинация y. Тогава x обновява своята оценка по правилото на Белман–Форд

$$D_x(y) = \min_{v \in N(x)} \{c(x, v) + D_v(y)\}.$$

Съседът, при който се достига минимумът, става следваща стъпка към y.

Предимството на този подход е по-малката потребност от мрежови и изчислителни ресурси. Недостатък е по-ниската скорост на сходимост. Добрите новини, например появата на по-кратък маршрут, се разпространяват сравнително бързо. Лошите новини,

например отпадането на връзка, могат да доведат до постепенно нарастващи погрешни оценки — проблема *count to infinity*. Split horizon не рекламира маршрут обратно по линията, от която е научен, а poisoned reverse го рекламира обратно с безкрайна метрика; тези техники ограничават, но не премахват всички възможни цикли.

13.3.2 Маршрутизация със следене на състоянието на линията

При маршрутизацията със следене на състоянието на линията всеки маршрутизатор събира информация за непосредствените си съседи и за цената на връзките към тях. След това тази информация се разпространява до останалите маршрутизатори.

Основните действия са:

1. Маршрутизаторът открива съседите си и научава техните мрежови адреси. Това може да стане чрез изпращане на специални ехо-пакети, на които съседите отговарят с информация за себе си.
2. Определя се метриката или цената за достигане до всеки съсед. Тя може да бъде установена автоматично или зададена от мрежов оператор.
3. Конструира се пакет, съдържащ адреса на подателя, пореден номер, възраст и списък на съседите с цената за достигане до тях.
4. Пакетът се изпраща до мрежата и се приемат аналогични пакети от останалите маршрутизатори. За бързо и надеждно достигане до всички се използва наводняване. Възрастта на пакета се намалява с течение на времето; когато стане нула, информацията се изхвърля.
5. Всеки маршрутизатор изчислява най-краткия път до останалите маршрутизатори, като използва алгоритъма на Дийкстра.

Разликата спрямо дистантния вектор е, че маршрутизаторът не получава само готови оценки за разстоянията, а изгражда картина на състоянието на връзките в мрежата. Върху тази картина се изчисляват най-кратките пътища.

13.4 IPv4 адресация

IPv4 адресите са 32-битови двоични числа. Обикновено се записват в точков десетичен формат като четири октета, разделени с точки. Всеки октет съдържа 8 бита и има стойност между 0 и 255. Например:

$$128.208.2.151$$

IP адресът се използва в полетата за адрес на източника и адрес на дестинацията в IP пакета.

IPv4 адресът има йерархична структура. Той се разделя на мрежова част, разположена в старшите битове, и хостова част, разположена в младшите битове. Мрежовата част идентифицира мрежата, а хостовата част идентифицира конкретен възел в нея. Маршрутизаторите могат да насочват пакетите към мрежата, без да знаят разположението на всеки отделен хост в тази мрежа.

13.4.1 Класова адресация

Исторически при класовата адресация адресите са разделени на класове според началните си битове. В съвременния Internet това разпределяне е заменено от CIDR.

- Клас A започва с битов префикс 0. Първият октет е в диапазона от 1 до 127. Адресите от вида $127.x.x.x$ са предназначени за loopback, а $0.0.0.0$ има специално метазначение. При стандартното класово разделяне остават $2^7 - 2 = 126$ мрежи и до $2^{24} - 2$ адреса за хостове във всяка мрежа.
- Клас B започва с префикс 10. Първият октет е в диапазона от 128 до 191. Броят на мрежите е 2^{14}, а броят на адресите за хостове във всяка мрежа е $2^{16} - 2$.
- Клас C започва с префикс 110. Първият октет е в диапазона от 192 до 223. Броят на мрежите е 2^{21}, а броят на адресите за хостове във всяка мрежа е $2^8 - 2$.
- Клас D започва с префикс 1110 и е предназначен за multicast. При multicast пакетът не е насочен към един конкретен хост.
- Историческият клас E започва с префикс 1111; диапазонът $240.0.0.0/4$ остава специално резервиран, а не обикновено unicast пространство.

Частните IPv4 адреси са предназначени за използване във вътрешни мрежи на организации. Посочените диапазони са $10.x.x.x$, $172.16.x.x$ до $172.31.x.x$ и $192.168.x.x$. Адресите от тези мрежи не се маршрутизират в глобалния интернет.

† **Допълнение 13.14 (Бележка).** Класовите формули в изходния материал са повредени при представянето им. Използвани са стандартните значения, следващи от броя битове за мрежова и хостова част.

Основен недостатък на класовата адресация е неефективното разпределяне на адресното пространство. Например голям дял от адресите принадлежи към клас A, но клас A предоставя само ограничен брой мрежи с много голям брой хостове.

13.4.2 Безкласова адресация

При безкласовата адресация и маршрутизация, означавана като CIDR, адресът се представя заедно с дължината на мрежовия префикс. Например:

$$192.0.2.96/28$$

означава, че първите 28 бита са мрежова част, а останалите 4 бита са хостова част.

Когато изпраща пакет, крайното устройство сравнява мрежовата маска със своя адрес и с адреса на получателя. Ако мрежовите части съвпадат, двата адреса принадлежат към една и съща мрежа и пакетът се доставя локално. Ако не съвпадат, пакетът се изпраща към локалния маршрутизатор.

Безкласовият подход позволява мрежовата част да има различна дължина според потребностите на мрежата. Това използва адресното пространство по-гъвкаво и се съчетава с маршрутизация, основана на префикси.

CIDR позволява агрегиране на съседни мрежи в по-общ префикс и така намалява маршрутните таблици. Когато няколко записа съвпадат с адреса на получателя, се избира най-специфичният — правилото за най-дълъг съвпадащ префикс (*longest-prefix match*).

13.5 IPv6

IPv6 поддържа основните функции на IPv4, но не е обратно съвместим с него. Най-съществената промяна е размерът на адреса: IPv6 използва 128-битови адреси. Така се решава проблемът с изчерпването на IPv4 адресите.

Основните характеристики на IPv6 са:

- по-голямо адресно пространство;
- опростен хедър;
- разширяем хедър чрез опционални допълнителни хедъри;
- автоматична конфигурация, включително stateful и stateless конфигурация;
- възможност за пряко адресиране от край до край, когато маршрутизацията и политиките за сигурност го позволяват;
- опростен основен хедър, който може да намали разхода за обработка в обичайния случай, без сам по себе си да гарантира по-висока скорост;
- липса на broadcast, като комуникацията с множество адреси се реализира чрез multicast;
- anycast, при който един адрес е присвоен на множество интерфейси, а пакетът се насочва към най-близкия според метриката на маршрутизиращия протокол;
- възможност за мобилност чрез отделните механизми на Mobile IPv6: домашен адрес, временен care-of адрес, home agent и актуализирани свързвания; само автоконфигурация не е достатъчна.

Пълният IPv6 запис съдържа осем групи от по четири шестнадесетични цифри, разделени с двоеточие. Водещите нули в група могат да се пропуснат, а една поредица от нулеви групи може да се съкрати с `::`. При global unicast адресите обичайно се различават глобален префикс, идентификатор на подмрежа и идентификатор на интерфейс; това разлагане не е универсална структура за всички специални типове адреси. Дължината на префикса се записва след наклонена черта.

Основните типове IPv6 адреси са:

- неопределен адрес: ::/128;
- loopback адрес: :: 1/128;
- multicast адреси: $FF00 :: /8$;
- link-local unicast адреси: $FE80 :: /10$;
- global unicast адреси: обичайно от префикса $2000 :: /3$.

IPv6 няма broadcast адреси. Изброяването не означава, че всяка стойност извън специалните префикси е глобален unicast адрес: адресното пространство съдържа и резервирани или предназначени за други цели области.

† **Допълнение 13.15 (Корекция на източника).** В източника IPv6 е наречен безусловно „по-сигурен“ от IPv4. Това не е общо свойство на самата версия на протокола: сигурността зависи от използваните механизми, настройки и приложения.

13.6 TCP и трикратно договаряне

Дефиниция 13.16 (TCP)**.** TCP (*Transmission Control Protocol*) е протокол от транспортното ниво. Целта му е надеждно да пренася поток от байтове, получен от по-горните нива.

Преди преноса двете крайни системи установяват връзка. Необходимо е всеки хост да съобщи началния номер на байтовата последователност, която ще изпраща, и да получи потвърждение, че този номер е приет.

Установяването се извършва чрез процедура на трикратно договаряне:

1. Клиентът изпраща сегмент със SYN и начален пореден номер x.
2. Сървърът отговаря със SYN с начален номер y и с $ACK = x + 1$.
3. Клиентът изпраща $ACK = y + 1$. SYN заема една позиция в пространството на поредните номера.

След третата стъпка и двата хоста са обменили необходимата начална информация и връзката може да се използва за пренос на потока от байтове.

13.7 DNS и резолвинг на имена

Дефиниция 13.17 (DNS)**.** DNS (*Domain Name System*) е йерархична система за именуване, базирана на домейни, и разпределена система от бази данни, която реализира тази схема. Основното ѝ приложение е съпоставянето на текстови имена на хостове с IP адреси.

Имената са организирани като дърво. В основата са top-level домейните. Те могат да бъдат универсални, например `com`, `edu`, `org` и `net`, или домейни за държави, например `bg`, `uk` и `us`. Всеки TLD се поддържа от регистър или делегиран TLD оператор. Регистраторите предоставят на заявителите услуга за регистриране на имена в допустимите домейни. Следващото ниво са second-level домейните, например `abv.bg` и `sony.nl`.

Името на домейна се образува като път от дадения домейн нагоре към корена, като компонентите се разделят с точки. DNS сървърите поддържат записи, чрез които имената се съпоставят с IP адреси. Възможен е и обратният процес – намиране на име по даден IP адрес.

13.7.1 Процес на резолвинг

Когато приложение се нуждае от IP адреса на дадено име, то извиква библиотечна процедура, наречена резолвър. Резолвърът получава името и изпраща заявка до локалния DNS сървър.

Stub резолвърът обичайно изпраща рекурсивна заявка към рекурсивен DNS резолвър: иска краен отговор, а не само насочване. Ако отговорът не е кеширан, рекурсивният резолвър обикновено извършва поредица от итеративни заявки. Коренов сървър го насочва към сървър за съответния TLD, той — към авторитетния сървър за домейна, а авторитетният сървър връща искания запис. При итеративна заявка всеки сървър може да върне най-доброто известно насочване, вместо сам да довърши търсенето.

Полученият резултат се връща към stub резолвъра и приложението. Записите и насочванията се кешират до срока, определен от тяхната TTL стойност. Кеширането намалява времето за следващо търсене и броя на обръщенията към други DNS сървъри.

13.7.2 IPv4 и IPv6 записи

Дефиниция 13.18 (DNS запис A)**.** За резолвинг на IPv4 адреси DNS използва записи от тип A. Те съдържат 32-битови IPv4 адреси.

Дефиниция 13.19 (DNS запис AAAA)**.** За резолвинг на IPv6 адреси се използват записи от тип AAAA. Те съдържат 128-битови IPv6 адреси.

Следователно за едно име могат да съществуват A записи, AAAA записи или и двата вида. Приложението или мрежовият стек избира адрес от подходящия тип според използвания IP протокол.

IPv4 и IPv6 са независими на ниво протокол: IPv4 хост не може непосредствено да комуникира с IPv6 хост чрез самия IPv4 или IPv6 протокол. DNS може да предостави адрес от единия или другия тип, но самото наличие на DNS запис не преобразува единия протокол в другия.

† **Допълнение 13.20 (Бележка).** В изходния материал DNS и NAT са посочени като механизми, позволяващи комуникация между IPv4 и IPv6. Това е неточно формулирано: DNS съпоставя имена с адреси, а NAT преобразува адреси в определени сценарии. Те не правят IPv4 и IPv6 обратно съвместими сами по себе си.

13.8 Изпитен фокус

- Определение на OSI модела, принципи за разделяне на нивата и ролята на енкапсулацията.
- Функции на седемте нива: приложно, презентационно, сесийно, транспортно, мрежово, канално и физическо.
- Разлика между референтния OSI модел и протоколно ориентирания TCP/IP модел.
- Определение на разпределена маршрутизация.
- Алгоритъм с дистантен вектор: правилото на Белман–Форд, следваща стъпка, обмен със съседи, count to infinity, split horizon и poisoned reverse.
- Алгоритъм със следене на състоянието на линията: откриване на съседи, измерване на цена, създаване и наводняване на пакети, възраст, алгоритъм на Дийкстра.
- IPv4 адрес, октет, мрежова и хостова част.
- Класова адресация: класове A–E, частни адреси и недостатък на класовото разпределяне.
- Безкласова адресация CIDR и смисълът на записа `адрес/дължина`.
- IPv6: 128-битови адреси, пълен и съкратен запис, префикс, автоматична конфигурация, multicast, anycast, Mobile IPv6 и основни типове адреси.
- TCP като транспортен протокол и трите стъпки на SYN–SYN–ACK/ACK договарянето с началните поредни номера.
- DNS като йерархична разпределена система от имена.
- DNS резолвинг: рекурсивна заявка от stub към recursive resolver, итеративни насочвания през root/TLD/authoritative сървъри и кеширане според TTL.
- Разлика между A запис за IPv4 и AAAA запис за IPv6.

Въпрос 14

Процедурно програмиране – основни конструкции.

По конспекта: какво трябва да мога

★ **Бележка на конспекта:** По този въпрос е възможно да бъдат дадени задачи за *практическата* част на изпита.

Принципи: структурно програмиране — §14.1, стр. 185.
Конструкции: управление на изчислителния процес — §14.2, стр. 187.
Конструкции: условни оператори — §14.3; оператори за цикъл — §14.4, стр. 190.
Променливи: локални и глобални; инициализация; оператор за присвояване — §14.5, стр. 192.
Функции и процедури: видове параметри; предаване *по име* и *по стойност*; типове и проверка за съответствие на тип — §14.6, стр. 195.
Терминология: предаването „по име“ е това, което конспектът назовава; §14.6.5 го различава от предаването чрез референция в C++ и препраща към Въпрос 19, §19.3.2.
Литература: [24], [27], [47].

14.1 Процедурно програмиране

Дефиниция 14.1 (Процедурно програмиране)**.** Процедурното програмиране е програмна парадигма, при която поведението на програмата се реализира чрез процедури, наричани още функции. Процедурите представляват самостоятелни части от програмата, които изпълняват определена задача и могат да бъдат извиквани многократно. Една процедура може да извиква друга процедура, поради което цялата програма се представя като поредица от стъпки, организирани в йерархия от извиквания.

Процедурното програмиране се класифицира като императивно програмиране. При него програмата съдържа директни команди, които определят какво да бъде изпълнено и в какъв ред. Основни понятия са променливата, операторът, блокът, функцията и областта на действие.

Дефиниция 14.2 (Блок)**.** Блок е последователност от оператори, оградена с фигурни скоби {}. Блокът се разглежда като една съставна конструкция и обикновено определя областта на действие на локално дефинираните в него променливи.

Дефиниция 14.3 (Област на действие)**.** Областта на действие на даден идентификатор е частта от програмния текст, в която този идентификатор може да бъде използван.

Понятията блок и област на действие са съществени за структурирането на програмата. Чрез тях се ограничава достъпът до данните и се отделят отделните части на алгоритъма.

14.1.1 Принципи на структурното програмиране

Основната идея на структурното програмиране е изграждането на ясна и разбираема структура на програмата. Програмата се разделя на отделни същности, всяка от които има ясно определена задача и граници. Пример за такава същност е функцията.

Основен метод за постигане на ясна структура е декомпозицията.

Дефиниция 14.4 (Декомпозиция)**.** Декомпозицията е разделяне на общата задача на по-малки подзадачи, всяка от които може да бъде реализирана чрез отделна процедура или функция.

При декомпозиция сложната задача се представя чрез йерархия от по-прости задачи. Функциите, които реализират тези задачи, могат да бъдат извиквани от други функции. Така програмата се организира като множество от взаимосвързани, но логически обособени компоненти.

★ **Дефиниция 14.5 (Модулност).** Модулността е принцип, според който функционалността на програмата се разпределя между отделни, взаимозаменяеми модули. Всеки модул трябва да съдържа всичко необходимо за изпълнението на един ясно определен аспект от програмата.

Модулността има две основни цели:

- всяка част от програмата да има ясно определена отговорност;
- отделните части да могат да бъдат използвани, проверявани и при необходимост заменяни независимо една от друга.

Добре структурираната процедурна програма се характеризира с последователност от ясно разграничени действия, функции с определено предназначение и ограничена област на действие на данните.

14.2 Управление на изчислителния процес

Дефиниция 14.6 (Изчислителен процес)**.** Изчислителният процес представлява изпълнение на поредица от стъпки в определен ред. Той има начална точка, последователност от междинни действия и крайна точка.

В C++ изпълнението на програмата започва от функцията `main`. Функцията `main` може да извиква други функции. При нормално изпълнение програмата завършва, когато приключи изпълнението на `main`.

Една от специалните форми на дефиниция на `main` е:

```
int main(int argc, char* argv[])
```

Параметърът `argc` указва броя на елементите в масива `argv`, а `argv` съдържа параметрите, подадени при стартиране на програмата от терминала. Възможна е и форма без параметри:

```
int main()
```

Управлението на изчислителния процес се осъществява чрез управляващи конструкции. Те променят нормалния линеен ред на изпълнение, като позволяват:

- избор между различни действия;
- многократно изпълнение на дадена последователност от оператори;
- прекратяване на част от текущото изпълнение.

Основните управляващи конструкции са условните оператори и операторите за цикъл.

14.3 Условни оператори

Условният оператор позволява даден оператор или блок да бъде изпълнен само когато определено условие е вярно.

14.3.1 Оператор `if`

Общият синтаксис е:

```
if (условие_0)
    оператор_0
else if (условие_1)
    оператор_1
else
    оператор_else
```

Частите `else if` могат да се повторят произволен брой пъти, а частта `else` е незадължителна и може да се срещне най-много веднъж.

Когато операторът се състои от повече от една команда, командите се обединяват в блок:

```
if (условие) {
    оператор_1
    оператор_2
}
```

Изпълнението протича по следния начин:

1. Оценява се първото условие.
2. Ако то има стойност `true`, изпълнява се съответният оператор или блок.
3. След това останалите части на условния оператор се прескачат.
4. Ако условието е `false`, се проверява следващото условие в `else if`.
5. При първото условие със стойност `true` се изпълнява съответният оператор и проверката приключва.
6. Ако всички условия са `false` и съществува `else`, се изпълнява операторът след `else`.

▷ **Пример 14.7.** Следната програма проверява знака на число:

```
int x;
cin >> x;

if (x > 0) {
    cout << "positive";
} else if (x < 0) {
    cout << "negative";
} else {
    cout << "zero";
}
```

При изпълнението се извършва само един от трите блока.

Ако няма част `else` и всички условия са неверни, не се изпълнява нито един от операторите в конструкцията.

14.3.2 Условна, или тернарна, операция

Условната операция е триместна операция със синтаксис:

$$\langle \text{булев израз} \rangle \ ? \ \langle \text{израз}_1 \rangle : \langle \text{израз}_2 \rangle$$

Първо се оценява булевият израз. Ако резултатът е `true`, се оценява и връща резултатът на $\langle \text{израз}_1 \rangle$. Ако резултатът е `false`, се оценява и връща резултатът на $\langle \text{израз}_2 \rangle$.

Пример:

```
int x = (y < 2) ? y + 1 : y - 2;
```

Ако $y = 1$, на `x` се присвоява стойност 2. Ако $y = 2$, на `x` се присвоява стойност 0.

Тернарната операция е подходяща, когато трябва да се избере една от две стойности. Тя връща резултат и затова може да се използва като част от по-голям израз.

14.3.3 Оператор за многозначен избор `switch`

Операторът `switch` служи за проверка на стойността на израз спрямо множество зададени стойности.

Синтаксисът има вида:

```
switch (израз) {
    case константен_израз_1:
        оператори_1
        break;

    case константен_израз_2:
        оператори_2
        break;

    default:
        оператори_default
}
```

Оценяваната стойност на израза се сравнява със стойностите след етикетите `case`. Когато бъде намерено съвпадение, изпълнението започва от съответния `case`.

▷ **Пример 14.8.**
```
int day;
cin >> day;

switch (day) {
    case 1:
        cout << "Monday";
        break;
    case 2:
        cout << "Tuesday";
        break;
    case 3:
        cout << "Wednesday";
        break;
    default:
```

```
        cout << "Unknown day";
}
```

Ако не е намерен съответстващ `case` и съществува етикет `default`, се изпълняват операторите след `default`. Ако няма съвпадение и няма `default`, операторът приключва без изпълнение на case-блок.

След намиране на съответстващ `case` изпълнението продължава последователно надолу в блока на `switch`. Самото достигане на следващ етикет `case` не прекратява изпълнението. За прекратяване се използва операторът `break`.

† **Допълнение 14.9 (Бележка).** Ако в съответстващия `case` липсва `break`, се осъществява последователно преминаване към следващите case-блокове. Това поведение е част от описаната семантика на оператора `switch`, а не грешка в синтаксиса.

Изпълнението на `switch` приключва при едно от следните събития:

- достигнат е краят на блока на `switch`;
- изпълнен е оператор за управление на потока като `break` или `return`;
- нормалното изпълнение е прекъснато по друга причина.

Ползата от `switch` спрямо множество `if--else if--else` конструкции е, че управляващият израз се оценява веднъж и след това се сравнява с различните константни стойности.

14.4 Оператори за цикъл

Цикълът е конструкция за управление на изчислителния процес, чрез която даден оператор или блок се изпълнява многократно. Повторението продължава, докато условието има стойност `true`. Когато условието стане `false`, цикълът приключва.

В C++ основните оператори за цикъл са `for`, `while` и `do while`.

14.4.1 Оператор `for`

Синтаксисът е:

```
for (инициализация; условие; корекция)
    тяло
```

При използване на блок:

```
for (инициализация; условие; корекция) {
    оператори
}
```

Изпълнението протича в следния ред:

1. Изпълнява се инициализацията. Тя се изпълнява само веднъж при влизане в цикъла.
2. Оценява се условието.
3. Ако условието е `true`, изпълнява се тялото на цикъла.
4. След тялото се изпълнява корекцията.
5. Отново се оценява условието.
6. Процесът се повтаря, докато условието стане `false`.

Най-често инициализацията служи за дефиниране и начално задаване на стойност на променлива, а корекцията увеличава или намалява тази променлива.

▷ **Пример 14.10.**
```
for (int i = 0; i < 5; i++) {
    cout << i << " ";
}
```
Изпълняват се стойностите 0, 1, 2, 3 и 4. След като `i` стане 5, условието `i < 5` е невярно и цикълът приключва.

Променлива, дефинирана в инициализационната част на `for`, има област на действие в цикъла, тоест може да бъде използвана до края на блока на цикъла.

14.4.2 Оператор `while`

Синтаксисът е:

```
while (условие)
    тяло
```

или:

```
while (условие) {
    оператори
}
```

При изпълнение първо се оценява условието. Ако то е `true`, се изпълнява тялото и след това отново се проверява условието. Ако условието е `false`, тялото не се изпълнява и цикълът приключва.

▷ **Пример 14.11.**
```
int i = 0;

while (i < 5) {
    cout << i << " ";
    i++;
}
```

При `while` е необходимо тялото на цикъла или друга част от изпълнението да води до промяна, която в даден момент да направи условието невярно. В противен случай повторението може да продължи без край.

14.4.3 Оператор `do while`

Операторът `do while` е сходен с `while`, но проверката на условието се извършва след изпълнението на тялото.

Синтаксисът е:

```
do {
    оператори
} while (условие);
```

Редът на изпълнение е:

1. изпълнява се тялото на цикъла;
2. оценява се условието;
3. ако условието е `true`, тялото се изпълнява отново;
4. ако условието е `false`, цикълът приключва.

Следователно тялото на `do while` се изпълнява поне веднъж, преди да бъде направена първата проверка.

▷ **Пример 14.12.** `int x;`

```
do {
    cin >> x;
} while (x < 0);
```

Тялото се изпълнява, след което се проверява дали въведеното число е отрицателно.

14.5 Променливи и присвояване

Дефиниция 14.13 (Променлива)**.** Променливата е именувана област в паметта, свързана с име, тип, адрес и стойност.

Името на променливата е идентификатор, чрез който тя се използва в програмния текст. Адресът указва мястото в паметта, отредено за променливата. Типът определя как компилаторът интерпретира съдържанието на това място и какви операции могат да се извършват със стойността. Стойността е текущото съдържание на променливата.

14.5.1 Дефиниране и инициализиране

Общият синтаксис за дефиниране на променливи е:

$$\langle\text{тип}\rangle\ \langle\text{идентификатор}\rangle[= \langle\text{израз}\rangle]\{,\ \langle\text{идентификатор}\rangle[= \langle\text{израз}\rangle]\}\ ;$$

Например:

```
int a;
double x = 3.5;
int p = 1, q = 2;
```

Дефинирането създава променлива от даден тип. Променливата е инициализирана, когато при създаването ѝ е зададена начална стойност.

В C++ могат да се използват например следните форми:

```
int x = 5;
int y(5);
int z{5};
```

При използване на присвояване с = се използва копираща инициализация. При формите с кръгли или фигурни скоби началната стойност се задава непосредствено при инициализацията.

† **Допълнение 14.14 (Бележка).** Източниците описват неинициализираната локална променлива като съдържаща непредвидима стойност. Практическото следствие е, че такава променлива не трябва да се използва, преди да ѝ бъде зададена определена стойност.

14.5.2 Оператор за присвояване

Синтаксисът на присвояването е:

$$\langle\text{идентификатор}\rangle = \langle\text{израз}\rangle;$$

Операторът оценява израза отдясно и записва получената стойност в променливата отляво.

```
int x;
x = 7;
x = x + 1;
```

Резултатът от присвояването е стойността, записана в променливата. Поради това присвояванията могат да се влагат едно в друго.

Дефиниция 14.15 (lvalue и rvalue). $lvalue$ е място в паметта, което има стойност и може да бъде променяно. Обикновена променлива е пример за $lvalue$.
$rvalue$ е стойност, която не представлява самостоятелно променяемо място в паметта. Константа, литералът и резултатът от пресмятане са примери за $rvalue$.

При израза

$$a = b = c = 2$$

присвояването се извършва отдясно наляво:

$$a = (b = (c = 2)).$$

Следователно първо на c се присвоява 2, полученият резултат се присвоява на b, а след това и на a.

14.5.3 Типове на променливите

Типът указва на компилатора как да интерпретира съдържанието на паметта и носи семантична информация. Променливите от един и същи тип имат сходни характеристики, включително размер и допустими операции.

Типовете могат да бъдат разглеждани като:

- скаларни типове;
- съставни типове.

Към скаларните типове спадат булевият, целочисленият, символният, изброимият тип, типовете за числа с плаваща запетая, указателите и референциите.

Към съставните типове спадат масивите, структурите и класовете.

14.5.4 Локални и глобални променливи

Областта на действие определя къде в програмния текст променливата може да бъде достъпвана.

Дефиниция 14.16 (Локална променлива)**.** Локална е променлива, дефинирана в блок или във функция. Областта ѝ на действие започва от мястото на дефинирането и продължава до края на блока, в който е дефинирана.

▷ **Пример 14.17.**
```
int main() {
    int x = 1;

    if (x > 0) {
        int y = 2;
        cout << y;
    }

    // y вече не е достъпна тук
}
```

Дефиниция 14.18 (Глобална променлива)**.** Глобална е променлива, дефинирана извън тялото на функция. Областта ѝ на действие е от мястото на дефинирането до края на файла.

Локалната област на действие ограничава използването на променливата до частта от програмата, в която тя е необходима. Глобалната променлива може да бъде достъпвана от по-голяма част от програмния файл.

14.6 Функции и процедури

Дефиниция 14.19 (Функция)**.** Функцията е самостоятелен фрагмент от програмата, съдържащ поредица от команди, предназначени за изпълнение на определена задача и позволяващи многократно използване.

Функцията извършва пресмятане и връща резултат. Процедурата изпълнява поредица от оператори, без задължително да връща резултат. В C++ и двете понятия се реализират чрез функции. Функция, която не връща резултат, има резултатен тип `void`.

14.6.1 Дефиниране на функция

Общият синтаксис е:

```
тип_резултат име(формални_параметри) {
    тяло
}
```

Например:

```
int square(int x) {
    return x * x;
}
```

В дефиницията се съдържат:

- резултатен тип;
- име на функцията;
- списък от формални параметри;
- тяло на функцията.

Резултатният тип може да бъде `void` или друг допустим тип, но в съвременния C++ трябва да бъде зададен изрично. Пропуснат резултатен тип не означава подразбиращ се `int`; такава стара конвенция е съществувала в ранни версии на C, но не е правило на C++.

Функция без параметри може да бъде представена с празен списък или с `void`:

```
void printMessage() {
    cout << "Hello";
}
```

Формалните параметри имат вида:

⟨тип⟩ ⟨идентификатор⟩{, ⟨тип⟩ ⟨идентификатор⟩}.

Например:

```
int add(int a, int b) {
    return a + b;
}
```

14.6.2 Декларация и извикване

Функцията може да бъде само декларирана, без да се задава тялото ѝ. Декларацията представлява обещание, че дефиниция на функцията ще бъде предоставена.

Примерна декларация е:

```
int add(int a, int b);
```

Извикването на функция се записва чрез нейното име и фактическите параметри:

```
име(фактически_параметри)
```

Например:

```
int result = add(2, 3);
```

Фактическите параметри, наричани още аргументи, са изрази:

⟨израз⟩{, ⟨израз⟩}.

При извикването типът на всеки фактически параметър се съпоставя с типа на съответния формален параметър. Ако е необходимо, се извършва преобразуване на типовете.

14.6.3 Връщане на резултат

Операторът за връщане има вида:

```
return израз;
```

Той връща стойността на израза като резултат от извикването на функцията и незабавно прекратява изпълнението на функцията.

▷ **Пример 14.20.**
```
int maximum(int a, int b) {
    if (a > b) {
        return a;
```

```
    }
    return b;
}
```

Типът на израза след `return` се съпоставя с резултатния тип на функцията. Ако е необходимо, се извършва преобразуване.

Функция с резултатен тип `void` може да изпълнява оператори, без да връща стойност. В този случай работата ѝ приключва при достигане на края на тялото или чрез `return` без резултат.

14.6.4 Стекова рамка при извикване

При извикване на функция се заделя нова стекова рамка. В нея се пазят фактическите параметри, адресът за връщане и локалните променливи на функцията. Рамката се намира на върха на програмния стек.

Следователно всяко извикване има собствен контекст. Локалните променливи и параметрите на едно извикване са свързани с неговата стекова рамка. При приключване на функцията изпълнението се връща към адреса, от който тя е била извикана.

14.6.5 Предаване на параметри

Предаване по стойност

При предаване по стойност първо се пресмята стойността на фактическия параметър. В стековата рамка на функцията се създава копие на тази стойност.

```
void change(int x) {
    x = 10;
}

int a = 3;
change(a);
```

Промяната на `x` остава локална за функцията. Стойността на `a` не се променя. При завършване на функцията копието и направените върху него промени изчезват.

Предаване чрез референция

Предаването чрез референция се използва, когато промените във формалния параметър трябва да се отразят върху фактическия параметър.

Синтаксисът на параметър-референция е:

```
тип& идентификатор
```

Пример:

```
int add5(int& x) {
    x += 5;
    return x;
}

int a = 3;
cout << add5(a) << " " << a;
```

Резултатът е:

```
8 8
```

В този случай параметърът x се свързва с променливата a. Затова промяната на x променя и a. При предаване чрез референция фактическият параметър трябва да бъде $lvalue$, тоест променяемо място в паметта.

Предаване по име и терминологията на конспекта

Конспектът назовава изрично „предаване на параметри — *по име* и *по стойност*“. Това е класическата двойка механизми от Алгол 60, а не двойката „по стойност / по референция“ на C++. Трите механизма трябва да се различават, защото на изпита се пита именно за първата двойка.

Дефиниция 14.21 (Предаване по стойност)**.** Фактическият параметър се оценява *веднъж*, преди влизането в тялото, и резултатът се копира в локалната променлива на формалния параметър. Промените върху формалния параметър не се виждат отвън.

Дефиниция 14.22 (Предаване по име)**.** Фактическият параметър *не* се оценява при извикването. Всяко срещане на формалния параметър в тялото се замества с целия израз на фактическия параметър и този израз се оценява наново при всяко срещане, в средата на извикващия.

Дефиниция 14.23 (Предаване чрез референция)**.** Формалният параметър става друго име за същото място в паметта. Адресът на фактическия параметър се пресмята *веднъж*, при извикването; след това всяко използване работи върху него.

Разликите се виждат на два примера.

▷ **Пример 14.24 (Многократно оценяване).** Нека

```
int брояч = 0;
int next() { return ++брояч; }

int twice(int x) { return x + x; }
```

При предаване по стойност `twice(next())` оценява `next()` веднъж: резултатът е $1+1=2$, а `брояч` става 1. При предаване по име тялото се чете като `next() + next()`:

резултатът е $1 + 2 = 3$, а `брояч` става 2. Референция тук изобщо не е приложима — `next()` не е $lvalue$.

▷ **Пример 14.25 (Устройството на Йенсен).** Процедура със заглавие `sum(i, n, term)`, в която `i` и `term` се предават по име, може да пресметне $\sum_{i=1}^{n} term$ за произволен израз `term`, зависещ от `i`: присвояването на `i` в тялото променя стойността, с която се преоценява `term` при следващото му използване. С предаване по стойност или по референция това е невъзможно, защото `term` е вече пресметнато число.

	По стойност	По име	Чрез референция
Кога се оценява	веднъж, при извикване	при всяко използване	адресът: веднъж
Какво се предава	копие на стойността	самият израз	мястото в паметта
Промени навън	не се виждат	през присвояване в израза	виждат се
Изисква $lvalue$	не	не	да
В C++	`f(int x)`	няма пряк аналог	`f(int& x)`

Забележка 14.26. C++ няма предаване по име. Двата му ефекта се получават поотделно: ефектът „вижда се отвън“ — чрез референция, разгледана по-горе в §14.6.5, а ефектът „изразът се оценява наново при всяко използване“ — чрез предаване на функция или ламбда, тоест чрез отлагане на оценяването. Пълното сравнение на апликативния ред (*call-by-value*) и нормалния ред (*call-by-name*) е дадено при функционалното програмиране във Въпрос 19, §19.3.2; там ленивото оценяване е основен механизъм, а не рядкост.

Съответствие на типовете

По време на компилация се проверява съответствието между типовете на формалните и фактическите параметри. Проверява се и съответствието между типа на връщания израз и резултатния тип на функцията. Когато е необходимо, се извършва преобразуване на типовете.

★ **Дефиниция 14.27 (Модел на нормално извикване и връщане).** При извикване на функция се оценяват фактическите параметри, съпоставят се със съответните формални параметри и се създава контекст за изпълнение на функцията. При нормално завършване функцията връща управлението към мястото на извикването. При функция, връщаща стойност, резултатът се задава чрез `return`; достигането на края е допустимо за `void` функции и за `main`. Описанието не определя реда на оценяване на отделните аргументи.

Това твърдение обобщава връзката между процедурната декомпозиция, предаването на данни и управлението на изчислителния процес.

14.7 Символни низове

Дефиниция 14.28 (Символен низ)**.** Символният низ е последователност от символи. В C++ той се представя като масив от елементи от тип `char`, след последния символ на който се записва терминиращият символ `'\0'`.

Терминиращият символ има код 0 в ASCII таблицата. Той показва къде завършва низът, независимо от размера на масива, в който е записан.

Пример:

```
char word[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Същият низ може да бъде зададен чрез литерал:

```
char word[] = "Hello";
```

При дефиниране на по-голям буфер:

```
char word1[100] = "Hello";
```

се заделя място за 100 символа. Низът заема символите на `Hello` и терминиращия символ, а останалото място остава част от масива. Максималната дължина на низа в такъв буфер е 99 символа, тъй като един елемент трябва да бъде запазен за `'\0'`.

14.7.1 Основни операции със символни низове

Въвеждането чрез `cin >>` чете до разделител, например интервал, табулация или нов ред.

```
char word[100];
cin >> word;
```

Функцията `getline` въвежда символи до нов ред, но трябва да й бъде зададен размер на буфера:

```
cin.getline(word, 100);
```

В този случай се въвеждат най-много 99 символа, така че да остане място за терминиращия символ.

Извеждането се извършва чрез `cout`:

```
cout << word;
```

До отделен символ се достига чрез индексиране:

```
char word[] = "Hello";
cout << word[1];
```

Резултатът е символът `'e'`, защото индексирането започва от 0.

14.7.2 Функции от библиотеката `cstring`

За работа със символни низове се използват функции от заглавния файл `cstring`.

- `strlen(низ)` връща дължината на низа, тоест броя символи до първото срещане на `'\0'`;
- `strcpy(буфер, низ)` прехвърля символите на низа в буфера и връща буфера;
- `strcmp(низ1, низ2)` сравнява два низа лексикографски;
- `strcat(низ1, низ2)` добавя символите на `низ2` в края на `низ1`;
- `strchr(низ, символ)` търси първото срещане на символ и връща адрес към него или нулев указател;
- `strstr(низ, подниз)` търси първото срещане на подниз и връща адрес към него или нулев указател.

Функцията `strcmp` връща:

- 0, ако низовете съвпадат;
- число, по-малко от 0, ако `низ1` е преди `низ2`;
- число, по-голямо от 0, ако `низ1` е след `низ2`.

При `strcat` програмистът трябва да осигури достатъчно място в първия буфер за символите на двата низа и за новия терминиращ символ.

Примерна реализация на `strlen` е:

```
int my_strlen(const char* str) {
    int length = 0;

    while (str[length] != '\0') {
        ++length;
    }

    return length;
}
```

Алгоритъмът започва с дължина 0 и последователно проверява символите на низа. При всеки символ, различен от `'\0'`, дължината се увеличава с единица. Когато бъде достигнат терминиращият символ, текущата дължина се връща.

14.8 Изпитен фокус

При устен отговор трябва да могат да бъдат възпроизведени:

- определението за процедурно програмиране и връзката му с императивното програмиране;
- ролята на блоковете и областта на действие;
- принципите на декомпозицията и модулността;
- началото и краят на изпълнението чрез функцията `main`;
- синтаксисът и семантиката на `if`, `else if`, `else`, тернарния оператор и `switch`;
- поведението на `switch` при липса на `break`;
- синтаксисът и редът на изпълнение при `for`, `while` и `do while`;
- понятието променлива, нейните име, тип, адрес и стойност;
- дефиниране, инициализиране и присвояване;
- разликата между $lvalue$ и $rvalue$;
- разликата между локална и глобална променлива;
- видовете скаларни и съставни типове;
- структурата на функцията, декларацията, дефиницията, извикването и `return`;
- предаването на параметри *по стойност* и *по име* — двойката, която конспектът назовава: кога се оценява фактическият параметър, какво точно се предава и защо C++ няма пряк аналог на предаването по име;
- предаването чрез референция в C++ и разликата му от предаването по име (адресът се пресмята веднъж, изразът не се преоценява);
- съпоставянето на типовете на формалните и фактическите параметри;
- стековата рамка при извикване на функция;
- представянето на символен низ чрез масив от `char` и терминиращ символ;
- основните операции и функциите от `cstring`;
- алгоритъма за реализация на `strlen`.

Въпрос 15

Процедурно програмиране – указатели, масиви и рекурсия.

По конспекта: какво трябва да мога

★ **Бележка на конспекта:** По този въпрос е възможно да бъдат дадени задачи за *практическата* част на изпита.

Указатели: указатели — §15.1; указателна аритметика — §15.3, стр. 207.
Масиви: едномерни и многомерни; основни операции; индексиране — §15.2, стр. 206.
Алгоритми: сортиране и търсене в едномерен масив — §15.5, стр. 208.
Символни низове: представяне в паметта — §15.4; основни операции — §15.7, стр. 213.
Рекурсия: пряка и косвена, линейна и разклонена — §15.6, стр. 212.
Литература: [24], [27], [47].

15.1 Указатели

Дефиниция 15.1. Указателят е обект, чиято стойност е адрес в паметта. Адресът обикновено е адресът на началото на стойност, а типът на указателя указва какъв е типът на стойността, намираща се на този адрес.

Указател се декларира чрез оператора $*$:

```
<тип>* <име> [= <израз>];
```

Например:

```
int x{53};
int* ptr{&x};
```

В този пример променливата x има стойност 53, а ptr съдържа адреса на x. На 32-битова архитектура указателят обикновено заема 4 байта, а на 64-битова архитектура — 8 байта. Физически адресът може да се представи като цяло число, но указателят не трябва да се разглежда просто като обикновено цяло число: неговият тип определя как се интерпретира адресът и как се извършват операциите с него.

15.1.1 Оператори за работа с указатели

Операторът за реферииране (достъпване до адрес) & е унарен оператор. Той връща адреса на своя операнд:

```
int x{53};
cout << &x << '\n';
```

Извежданата стойност е адресът на променливата x.

Операторът за дереференция * също е унарен. Той приема указател и връща стойността, съхранявана на адреса, към който указателят сочи:

```
int x{53};
cout << *(&x) << '\n';
```

Резултатът от програмния фрагмент е 53. Ако ptr сочи към x, изразът *ptr обозначава самата стойност на x и може да участва както в четене, така и в промяна на стойността, ако това е допустимо.

Указател, който не е инициализиран, съдържа неопределена стойност. Използването му като валиден адрес може да доведе до неправилно поведение. Специалната стойност nullptr означава, че указателят не сочи към обект:

```
int* pi;
double* pd = nullptr;
```

Указателят pd не сочи към никакъв адрес. Дереферирането на nullptr е недопустимо.

Основните операции с указатели са:

- получаване на адрес чрез &;
- дереферииране чрез *;
- сравнение чрез операторите ==, !=, <, >, <=, >=;
- извеждане чрез <<.

Указател не се въвежда директно чрез оператора >>. Обикновено се въвежда стойност в обект, след което указателят се насочва към този обект.

15.1.2 Указатели към указатели

Тъй като указателят е обект, той също има адрес. Следователно може да се дефинира указател към указател:

```
double d = 1.23;
double* dptr = &d;
double** dptrptr = &dptr;
```

Тук `dptr` сочи към `d`, а `dptrptr` сочи към самия указател `dptr`. Изразът `*dptrptr` дава указателя `dptr`, а `**dptrptr` дава стойността 1.23.

15.1.3 Указатели и константи

Различават се две важни конструкции.

Константният указател е указател, чиято посока не може да се променя:

```
int x = 5;
int* p = &x;
int* const q = p;

*q = 5;
q = p + 2;        // грешка
```

Посочваната стойност може да бъде променяна чрез `q`, но на `q` не може да се присвои друг адрес.

Указателят към константа не позволява промяна на стойността чрез този указател:

```
int x = 5;
const int* q = &x;

q++;              // допустимо: променя се адресът
*q = 5;           // грешка
```

Записът `const int*` е еквивалентен на `int const*`. Ако `x` е константа, то `&x` е указател към константа. Не може безусловно да се присвои указател към константа на обикновен указател, защото така би се премахнала гаранцията, че стойността няма да бъде променяна.

15.2 Масиви

Дефиниция 15.2. Масивът е съставен тип данни, представляващ крайна редица от елементи от един и същи тип с фиксирана дължина. Всеки елемент се достига по индекс. Индексите са последователни цели числа, като първият елемент има индекс 0.

Едномерни масиви се дефинират например така:

```
bool b[10];
double x[3] = {1.2, 0.5, 3.14};
int a[] = {2, 3};
```

Масивът `b` има десет елемента. Ако не е инициализиран, елементите му съдържат неопределени стойности. При `x` дължината и инициализиращият списък са зададени явно. При `a` размерът е пропуснат и се извежда от броя на елементите в списъка.

Елементите на масива са разположени последователно в паметта. В повечето изрази масивът се преобразува в указател към първия си елемент; самият масив не е указателен обект. Например при `sizeof(a)` и `&a` това преобразуване не се извършва. Това обяснява тясната връзка между масивите и указателите.

15.2.1 Достъп до елементи

Достъпът до елемент се извършва чрез оператора `[]`:

```
<масив>[<цяло число>]
```

Например:

```
int a[] = {8, 2};
int y = a[1];
```

След изпълнението `y` има стойност 2. За масив `x[3]` допустимите индекси са 0, 1 и 2. В C++ не се извършва автоматична проверка дали индексът е в допустимите граници. Отрицателен индекс или индекс, по-голям от последния валиден индекс, може да доведе до достъп до чужда памет.

За вградените масиви няма присвояване на цели масиви и няма автоматично поелементно сравнение. Например два масива не могат да бъдат сравнени с един оператор така, че да се сравнят всичките им елементи.

15.2.2 Многомерни масиви

Дефиниция 15.3 (Многомерен масив)**.** Масив, чиито елементи са едномерни масиви, се нарича двумерен масив. По аналогичен начин се дефинират тримерни и по-общо N-мерни масиви:

```
<тип> <име>[<размер1>][<размер2>] ...[<размерN>];
```

Пример за двумерен масив е:

```
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

Той има две редици с по три елемента. Достъпът до елемент се извършва с два индекса, например `a[1][2]`.

При инициализиране на многомерен масив първата размерност може да бъде пропусната, ако тя може да се изведе от инициализиращия списък. Останалите размерности трябва да бъдат известни, за да може компилаторът да изчислява адреса на елементите.

15.3 Указателна аритметика

Указателната аритметика позволява чрез указател да се достигат съседни елементи в паметта. Ако `p` е указател към тип `T`, тогава:

```
p+i
```

обозначава адрес, отместен с i елемента от тип `T`, а не просто с i байта. Реалното отместване в байтове е:

$$i \cdot \text{sizeof}(T).$$

Съответно `p-i` се отмества назад с i елемента. Например, ако `int` заема 4 байта, то `p+2` е адрес, отстоящ на 8 байта от `p`.

При масив `a` имаме равенствата:

$$\texttt{a[i]} \iff \texttt{*(a+i)} \iff \texttt{*(i+a)} \iff \texttt{i[a]}.$$

Следователно следният код е еквивалентен по смисъл:

```
int a[5] = {1, 2, 3, 4, 5};

cout << *a << '\n';
*(a + 1) = 7;
*(a + 4) = 4;
```

След присвояването вторият елемент е 7, а последният елемент е 4. Изразът `a` се интерпретира като адреса на първия елемент, а `a+i` като адреса на елемента с индекс i.

15.4 Указатели и низови константи

Дефиниция 15.4 (Символен низ). Символният низ в C++ може да се представи като масив от символи, завършващ с терминиращия символ `'\0'`. Масивът от символи в повечето изрази се преобразува в указател към първия символ. Обикновеният низов литерал в C++ е масив от константни символи и при такова преобразуване дава указател към константни символи.

```
char const* p = "Hello";
char q[] = "Hello1";

p = q;
q[1] = 'o';
```

Чрез `p` символите на низовата константа не могат да се променят. За разлика от това, `q` е масив от символи и неговите елементи могат да бъдат променяни, ако масивът не е константен.

15.5 Сортиране и търсене в едномерен масив

15.5.1 Selection sort

Selection sort разделя масива на сортирана и несортирана част. На всяка стъпка се намира най-малкият елемент в несортираната част и се разменя с нейния най-ляв елемент. Така сортираната част нараства с един елемент.

```
void selectionSort(int* a, int n) {
    for (int i = 0; i < n - 1; ++i) {
        int min = i;
        for (int j = i + 1; j < n; ++j) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        if (min != i) {
            swap(a[i], a[min]);
        }
    }
}
```

Алгоритъмът има времева сложност $O(n^2)$.

15.5.2 Bubble sort

Bubble sort многократно преминава през несортираната част. Сравнява съседни елементи и ги разменя, ако левият е по-голям от десния. След едно преминаване най-големият

елемент на разглежданата част се оказва най-вдясно. Процесът се повтаря до сортиране или до преминаване без нито една размяна.

```
void bubbleSort(int* a, int n) {
    for (int i = 0; i < n - 1; ++i) {
        bool swapped = false;

        for (int j = 0; j < n - 1 - i; ++j) {
            if (a[j] > a[j + 1]) {
                swap(a[j], a[j + 1]);
                swapped = true;
            }
        }

        if (!swapped) {
            break;
        }
    }
}
```

Времевата сложност е $O(n^2)$.

15.5.3 Insertion sort

Insertion sort поддържа вече сортирана част от масива. На всяка итерация избира следващ елемент, наречен ключ, и го вмъква на подходящото място, като по-големите елементи се изместват надясно.

```
void insertionSort(int* a, int len) {
    int key;

    for (int i = 1; i < len; ++i) {
        key = a[i];
        int j = i - 1;

        while (j >= 0 && key < a[j]) {
            a[j + 1] = a[j];
            --j;
        }

        a[j + 1] = key;
    }
}
```

Времевата сложност, посочена за алгоритъма, е $O(n^2)$.

15.5.4 Quick sort

Quick sort избира разделящ елемент, наречен pivot, и разделя останалите елементи на подмасиви според това дали са по-малки или по-големи от pivot-а. След това двата подмасива се сортират рекурсивно.

Една възможна схема за разделяне е:

```
int partition(int* a, int l, int r) {
    int pivot = a[r];
    int i = l - 1;

    for (int j = l; j < r; ++j) {
        if (a[j] <= pivot) {
            ++i;
            swap(a[i], a[j]);
        }
    }

    swap(a[i + 1], a[r]);
    return i + 1;
}

void quickSort(int* a, int l, int r) {
    if (l >= r) {
        return;
    }

    int p = partition(a, l, r);
    quickSort(a, l, p - 1);
    quickSort(a, p + 1, r);
}
```

† **Допълнение 15.5 (Бележка).** В предоставените материали е посочено както $O(n^2)$, така и средна сложност $O(n \log n)$ за quick sort. Тези твърдения се отнасят до различни случаи: $O(n^2)$ е посочено за неблагоприятния случай, а $O(n \log n)$ — като средна сложност. Изборът на pivot влияе върху полученото разделяне.

15.5.5 Merge sort

Merge sort разделя масива на две части, след това всяка част отново се разделя, докато се получат подмасиви с по един елемент. После подмасивите се сливат два по два, като при сливането се запазва сортираната наредба.

Сливането на два вече сортирани интервала използва помощен масив:

```
void merge(int* a, int l, int m, int r, int* tmp) {
```

```
    int i = l;
    int j = m + 1;
    int k = 0;

    while (i <= m && j <= r) {
        if (a[i] <= a[j]) {
            tmp[k++] = a[i++];
        } else {
            tmp[k++] = a[j++];
        }
    }

    while (i <= m) {
        tmp[k++] = a[i++];
    }

    while (j <= r) {
        tmp[k++] = a[j++];
    }

    memcpy(a + l, tmp, k * sizeof(int));
}

void mergeSort(int* a, int l, int r, int* tmp) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(a, l, m, tmp);
        mergeSort(a, m + 1, r, tmp);
        merge(a, l, m, r, tmp);
    }
}
```

Посочената времева сложност е $O(n \log n)$.

15.5.6 Линейно търсене

При линейното търсене елементите се разглеждат последователно от началото на масива. Търсенето приключва при намиране на търсения елемент или при достигане на края на масива. Времевата сложност е $O(n)$.

15.5.7 Двоично търсене

Двоичното търсене се прилага върху сортиран масив. На всяка стъпка се разглежда средният елемент. Ако той е търсеният, алгоритъмът приключва. Ако средният елемент е по-малък от търсения, се продължава в дясната половина; в противен случай — в лявата половина.

```
int binarySearch(int* a, int n, int x) {
    int l = 0;
    int r = n - 1;

    while (l <= r) {
        int m = l + (r - l) / 2;

        if (a[m] == x) {
            return m;
        }

        if (a[m] < x) {
            l = m + 1;
        } else {
            r = m - 1;
        }
    }

    return -1;
}
```

Ако елементът бъде намерен, функцията връща индекса му. При липса на елемента връща -1.

15.6 Рекурсия

Дефиниция 15.6. Рекурсивна е функция, която извиква себе си пряко или косвено.

При пряка рекурсия функцията извиква сама себе си в собственото си тяло. При косвена рекурсия се образува цикъл от извиквания:

$$F_1 \to F_2 \to \cdots \to F_n \to F_1.$$

Коректната рекурсивна функция трябва да има условие за край, наричано базов случай. Освен това всяко рекурсивно извикване трябва да води към този случай.

15.6.1 Линейна рекурсия

При линейната рекурсия функцията извършва едно рекурсивно извикване за решаването на даден подпроблем. Пример е факториелът:

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    }
```

```
    return n * factorial(n - 1);
}
```

Базовият случай е `n == 0`. При всяко следващо извикване аргументът намалява с единица.

† **Допълнение 15.7 (Бележка).** В предоставения материал до примера за факториел е записана сложност $n \log n$. Самата рекурсивна схема извършва по едно извикване за всяка стойност от n до 0, поради което посоченото твърдение не съответства на показания код. В изложението се използва структурата на алгоритъма: линейна рекурсия с линейно число извиквания.

15.6.2 Разклонена рекурсия

При разклонената рекурсия функцията извиква себе си два или повече пъти за един подпроблем. Типичен пример е рекурсивната реализация на числата на Фибоначи:

```
int fib(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }

    return fib(n - 2) + fib(n - 1);
}
```

Тук всяко нетривиално извикване поражда две нови извиквания. Получава се дърво на извикванията, а не една линейна верига.

Рекурсията е особено удобна при рекурсивни структури, обхождане на графи, търсене с връщане назад и алгоритми от типа „разделяй и владей“. Quick sort и merge sort също естествено се описват рекурсивно, тъй като решават същия вид задача върху по-малки подмасиви.

Основен недостатък на рекурсията е скритото използване на памет за стекови рамки. При всяко извикване се съхраняват параметри, локални променливи и информация за връщането. При прекалено дълбока рекурсия стекът може да се изчерпи.

15.7 Символни низове

Дефиниция 15.8. Символният низ е поредица от символи. В C++ той може да се представи като масив от тип `char`, след последния значим символ на който се записва терминиращият символ `'\0'`.

Например:

```
char word[] = {'H', 'i', '\0'};
```

Терминиращият символ има код 0 и показва края на низа.

Основните операции са:

- операторът `>>` въвежда символи до разделител, например интервал, нов ред или табулация;
- `cin.getline` въвежда до нов ред, но не повече от зададения брой символи;
- операторът `<<` извежда низа;
- операторът `[]` осигурява достъп до отделен символ.

Функциите за работа с низове се намират в заглавния файл `<cstring>`:

- `strlen(str)` връща броя на символите преди `'\0'`;
- `strcpy(buffer, str)` копира низа в буфера и връща буфера;
- `strcmp(str1, str2)` извършва лексикографско сравнение;
- `strcat(str1, str2)` добавя втория низ към края на първия;
- `strchr(str, symbol)` връща адреса на първото срещане на символа или нулев указател;
- `strstr(str, sub)` връща адреса на първото срещане на подниза или нулев указател.

При `strcmp` резултатът е 0, ако низовете съвпадат, отрицателен, ако първият низ предхожда втория, и положителен, ако първият следва втория.

При `strcat` програмистът трябва да осигури достатъчно място в първия масив за получения низ и неговия терминиращ символ.

Една възможна реализация на `strlen` е:

```
size_t strlen(const char* str) {
    int length = 0;

    while (str[length] != '\0') {
        ++length;
    }

    return length;
}
```

Функцията обхожда символите последователно, докато достигне `'\0'`, и брои видимите символи.

15.8 Изпитен фокус

- Определение на указател и предназначение на операторите & и *.
- Разлика между `nullptr`, неинициализиран указател, константен указател и указател към константа.
- Представяне на масив в паметта, индексиране и липса на автоматична проверка на границите.
- Еквивалентността `a[i] = *(a+i) = *(i+a) = i[a]`.
- Формулата за указателно отместване: $i \cdot \text{sizeof}(T)$.
- Едномерни и многомерни масиви.
- Идеята и сложността на selection sort, bubble sort, insertion sort, quick sort и merge sort.
- Разлика между линейно и двоично търсене и условието двоичното търсене да се извършва върху сортиран масив.
- Определение за пряка и косвена рекурсия.
- Разлика между линейна и разклонена рекурсия, включително ролята на базовия случай.
- Предимства и недостатъци на рекурсивния подход, по-специално използването на стекови рамки.
- Представяне на символен низ чрез масив от `char` с терминиращ символ `'\0'` и предназначение на основните функции от `<cstring>`.

Въпрос 16

ООП. Основни принципи. Класове и обекти. Наследяване и капсулация.

По конспекта: какво трябва да мога

★ **Бележка на конспекта:** По този въпрос е възможно да бъдат дадени задачи за *практическата* част на изпита.

Класове и обекти: абстракция със структури от данни; декларация на клас и на обект; основни видове конструктори; управление на динамичната памет и ресурсите (RAII); методи — декларация, предаване на параметри, връщане на резултат — §16.2, стр. 217.
Статични членове: статични полета и методи — §16.2.6, стр. 222.
Наследяване: производни класове; достъп до наследените компоненти — §16.3, стр. 223.
Вложени класове: декларация на тип в тялото на друг клас; разликата от влагането на обект (композиция) — §16.3.4, стр. 226.
Капсулация: капсулация и скриване на информацията — §16.4, стр. 227.
Литература: [23], [24], [26], [47].

16.1 Обектно-ориентирано програмиране

Дефиниция 16.1 (Обектно-ориентирано програмиране)**.** Обектно-ориентираното програмиране (ООП) е подход за изграждане на програми, при който основните програмни единици са обекти. Обектът обединява данни, описващи неговото състояние, и операции, определящи неговото поведение. По този начин програмата се организира около програмно-дефинирани типове данни, които съдържат едновременно състояние и поведение.

Основните идеи на ООП са:

- абстракция;

- капсулация и скриване на информация;
- наследяване;
- полиморфизъм.

Дефиниция 16.2 (Абстракция)**.** Абстракцията позволява да се отдели съществено-то описание на дадена структура от конкретната ѝ реализация. При използване на даден тип е необходимо да се знае какви операции предоставя и какъв резултат имат те, но не непременно как са реализирани вътрешно. Това изолира потребителя на типа от промени в реализацията.

Дефиниция 16.3 (Капсулация)**.** Капсулацията е свързана с групирането на данни и функциите, които работят с тях, в единен програмен обект. Вътрешното състояние се скрива и се достъпва чрез контролирано множество от операции. Така се запазват инвариантите на обекта, тоест условията, които трябва да са изпълнени за коректното му състояние.

Наследяването позволява чрез вече съществуващ клас да се създаде нов клас, който разширява неговите данни и поведение. То изразява връзка от вида „е“ (*is-a*). Полиморфизмът позволява една и съща операция да има различно поведение в зависимост от конкретния тип на обекта.

16.2 Класове и обекти

Дефиниция 16.4. Класът е програмно-дефиниран тип, който описва множество от обекти с обща структура и общо поведение. Обектът е конкретна инстанция на класа.

Класът играе ролята на шаблон. Той описва:

- член-данни, наричани още полета, които представят състоянието;
- член-функции, наричани още методи, които определят поведението;
- правила за създаване и унищожаване на обектите;
- достъпа до отделните компоненти.

Дефиниция 16.5 (Поле на обект и статично поле)**.** Нормалното поле принадлежи на конкретен обект и всеки обект има собствено копие от него. Статичното поле принадлежи на самия клас и се споделя от всички негови обекти.

Декларацията на клас в C++ има вида:

```
class ИмеНаКласа {
    // член-данни и член-функции
};
```

Методите могат да бъдат дефинирани вътре в класа или извън него. При дефиниране извън класа пред името на метода се записват името на класа и операторът за достъп ::

```
class Counter {
private:
    int value;

public:
    Counter(int initial);
    void increment();
    int getValue() const;
};

Counter::Counter(int initial) : value{initial} {
}

void Counter::increment() {
    ++value;
}

int Counter::getValue() const {
    return value;
}
```

Обектите могат да се създават като автоматични променливи, като динамично заделени обекти или като елементи на други обекти:

```
Counter c{0};
Counter* p = new Counter{10};

c.increment();
p->increment();

delete p;
```

Връзката между клас и обект е аналогична на връзката между тип и променлива. Класът описва как изглеждат и как действат обектите, а обектът съдържа конкретни стойности на полетата.

16.2.1 Модификатори за достъп

Член-данните и член-функциите могат да бъдат обявени с три модификатора за достъп:

- `private` – компонентите са достъпни само от член-функциите и приятелите на класа;

- `protected` – компонентите не са достъпни от външни функции, но са достъпни в производните класове;
- `public` – компонентите могат да се използват от външен код според правилата на езика.

При `class` достъпът по подразбиране е `private`, а при `struct` е `public`. Обикновено полетата се поставят в частта `private`, а публичният интерфейс се реализира чрез методи.

16.2.2 Конструктори

Дефиниция 16.6. Конструкторът е специална член-функция, която се изпълнява автоматично при създаване на обект и извършва неговата инициализация.

Конструкторът има същото име като класа и няма изрично зададен тип на резултата. Може да има параметри и инициализиращ списък:

```
class Pair {
private:
    int x;
    int y;

public:
    Pair(int first, int second) : x{first}, y{second} {
    }
};
```

Инициализиращият списък е предназначен за инициализиране на полетата преди изпълнението на тялото на конструктора. Той е особено важен при константни полета, препратки и обекти, за които няма конструктор без параметри.

Възможните видове конструктори, разгледани в източниците, са:

1. Обикновен конструктор с параметри. Той приема стойности, с които се изгражда началното състояние на обекта.
2. Конструктор с параметри по подразбиране. Някои или всички параметри имат предварително зададени стойности. Ако даден параметър има стойност по подразбиране, всички параметри след него също трябва да имат стойности по подразбиране.
3. Системно генериран конструктор по подразбиране. Ако програмистът не дефинира конструктор, компилаторът може да създаде такъв, който извършва необходимата стандартна инициализация.
4. Конструктор за копиране. Той изгражда нов обект като копие на друг обект от същия клас. Обичайният му параметър е от вида `const ИмеНаКласа&`.

5. Конструктор за преобразуване на тип. Конструктор с точно един параметър може да задава правило за създаване на обект от стойност от друг тип. Така C++ може да използва неявно преобразуване, когато се очаква обект от даден клас.
6. Преместващ конструктор. Той приема обект, чиито ресурси могат да бъдат премес-тени в новия обект, вместо да бъдат копирани.

Пример за конструктор за копиране:

```
class Pair {
private:
    int x{3};
    int y{3};

public:
    Pair(int first, int second) : x{first}, y{second} {
    }

    Pair(Pair const& other) : x{other.x}, y{other.y} {
    }
};
```

Конструкторът за преобразуване може да доведе до неявни преобразувания. Когато това не е желано, конструкторът може да бъде обявен като `explicit`; тази възможност не е разгледана подробно в източниците и тук се посочва само като практическа забележка към описаното правило.

† **Допълнение 16.7 (Бележка).** В източника е посочено, че типът на резултата на конструктора е `this`. В C++ конструкторът няма тип на резултата и не се декларира като функция, връщаща `this`. `this` е специален указател към текущия обект, достъ-пен в нестатичните член-функции.

16.2.3 Деструктори и управление на ресурси

Дефиниция 16.8. Деструкторът е специална член-функция, която се изпълнява ав-томатично при унищожаване на обект и освобождава ресурсите, притежавани от не-го.

Името на деструктора се образува от знака ~ и името на класа. Един клас има най-много един деструктор:

```
class Resource {
public:
    ~Resource() {
        // освобождаване на ресурс
    }
};
```

Деструкторът се извиква при достигане края на областта на действие на автоматичен обект или при използване на `delete` за динамично заделен обект. Ако не бъде дефиниран явно, системата може да генерира деструктор по подразбиране.

Динамичната памет се заделя в областта, наречена heap. Тя може да бъде заделяна и освобождавана по време на изпълнението на програмата. Програмистът носи отговорност за правилното освобождаване на динамично заделената памет.

Принципът RAII (*Resource Acquisition Is Initialization*) свързва притежаването на ресурс с живота на обект. Ресурсът се придобива при конструиране на обекта, а се освобождава при неговото унищожаване. Така правилното управление на ресурса се превръща в свойство на класа.

При ръчно управление на външни ресурси са важни така наречените специални операции:

- конструктор;
- конструктор за копиране;
- деструктор;
- оператор за присвояване.

Източникът нарича този набор „голяма четворка“. Тези функции могат да бъдат генерирани системно, но се дефинират изрично, когато класът управлява външен ресурс, например динамична памет.

При управлението на динамична памет трябва ясно да е определено кой указател притежава паметта. При модел със собственик един указател е отговорен за освобождаването, а останалите само използват паметта. При модела с умни указатели собствеността може да се споделя; поддържа се броят на указателите към даден ресурс и той се освобождава, когато броят стане нула.

16.2.4 Методи и указателят `this`

При извикване на нестатичен метод той се свързва с конкретен обект. В тялото на метода полетата могат да се използват непосредствено, без да се изписва името на обекта. Това се обяснява с наличието на специален указател `this`, който сочи към текущия обект.

Концептуално методът:

```
int Counter::getValue() const {
    return value;
}
```

може да се разглежда като функция, която получава текущия обект чрез скрит параметър. Достъпът до `value` е еквивалентен на достъп до поле през `this`.

Константният метод се отбелязва с `const` след списъка с параметри:

```
int getValue() const;
```

Той гарантира, че методът няма да променя състоянието на обекта и няма да извиква неконстантни методи върху него. Затова константните методи могат да се извикват и за константни обекти.

Параметрите на методите се предават по стойност или по препратка, а методът може да върне един резултат, включително обект от даден клас.

16.2.5 Предефиниране на операции

C++ позволява повечето вградени операции да бъдат предефинирани за обекти от потребителски клас. Предефинирането се реализира чрез специален метод или функция с име `operator`, следвано от символа на операцията.

Общият вид е:

```
тип operator операция(параметри) const {
    // тяло
}
```

Пример:

```
Pair Pair::operator*(Pair const& other) const {
    return Pair(x * other.x, y * other.y);
}
```

Така изразът `p1 * p2` може да се използва за обекти от тип `Pair`. Предефинирането трябва да съответства на естествения смисъл на операцията и да не нарушава очакванията за работа с класа.

Според източника не могат да бъдат предефинирани операциите `?:`, `::`, `.`, `sizeof`, `#` и `##`.

16.2.6 Статични полета и методи

Статичното поле е общо за всички обекти на класа. То не принадлежи на конкретна инстанция и съществува дори когато няма създаден обект от класа.

```
class Foo {
public:
    static const int constantValue{43};
    static int value;
};

int Foo::value{13};
```

Статичното поле може да бъде достъпено чрез името на класа и оператора ::.

Статичният метод също принадлежи на класа, а не на конкретен обект:

```
class IDGenerator {
private:
    static inline int nextID{1};

public:
    static int getID() {
        return nextID++;
    }
};
```

Извикването има вида:

```
int id = IDGenerator::getID();
```

Статичният метод няма указател `this`. Следователно той може да достъпва статични полета и статични методи, но не може непосредствено да използва нестатични полета, защото няма конкретен обект, към който те да принадлежат.

16.3 Наследяване

Дефиниция 16.9. Наследяването е механизъм за създаване на производен клас чрез използване на данни и поведение на базов клас.

Дефиниция 16.10 (Базов и производен клас)**.** Класът, от който се наследява, се нарича базов, родителски или основен клас. Новият клас се нарича производен клас, наследник или подклас.

Наследяването изразява връзка „е“. Ако `Dog` наследява `Animal`, кучето е животно. За разлика от това, влагането на обект от един клас в друг изразява връзка „има“. Например автомобилът има двигател.

Синтаксисът на наследяване е:

```
class Производен : public Базов {
    // тяло
};
```

C++ позволява и наследяване от повече от един базов клас:

```
class A {
};

class B {
};

class C : public A, public B {
};
```

При наследяване се задава видимостта на базовия клас: `public`, `protected` или `private`. При `class` наследяването по подразбиране е `private`.

	`public`	`protected`	`private`
`public` наследяване	`public`	`protected`	недостъпно
`protected` наследяване	`protected`	`protected`	недостъпно
`private` наследяване	`private`	`private`	недостъпно

Производният клас получава компонентите на базовия клас като част от своята структура, но не получава достъп до неговите `private` компоненти. Те се съдържат в базовата част на обекта, но могат да се използват само чрез интерфейса на базовия клас. Производният клас има достъп до наследените `public` и `protected` компоненти според вида на наследяването.

Производният клас може да добавя собствени полета и методи. Той може да дефинира компонент със същото име като компонент от базовия клас. Това се нарича предефиниране на име и може да скрие компонента от базовия клас в даден контекст.

16.3.1 Конструиране на производен обект

Производният обект съдържа базова част и производна част. При създаването му действията се извършват в следния ред:

1. заделя се памет за целия производен обект;
2. конструира се базовата част;
3. инициализират се полетата на производната част чрез инициализиращия списък;
4. изпълнява се тялото на конструктора на производния клас.

Ако производният конструктор не зададе явно кой конструктор на базовия клас да се използва, се използва конструкторът му по подразбиране, ако такъв е наличен.

```
class Base {
private:
    int id{};

public:
    Base(int value = 0) : id{value} {
        std::cout << "Base\n";
    }

    int getID() const {
        return id;
    }
};
```

```
class Derived : public Base {
private:
    double cost{};

public:
    Derived(double value = 0.0, int id = 0)
        : Base{id}, cost{value} {
        std::cout << "Derived\n";
    }

    double getCost() const {
        return cost;
    }
};
```

При създаване на `Derived d{1.3, 5}` първо се изпълнява конструкторът на `Base`, а след това този на `Derived`. Извикванията на `d.getID()` и `d.getCost()` използват съответните публични методи.

При унищожаване редът е обратен: първо се унищожава производната част, а след това базовата част.

16.3.2 Подтипово отношение

При публично наследяване производният клас се разглежда като подтип на базовия клас. Всеки обект от производния тип може да бъде използван там, където се очаква обект от базовия тип. Това преобразуване е неявно за обекти, указатели и препратки:

```
Derived d;
Derived* pd = &d;
Derived& rd = d;

Base* pb = pd;
Base& rb = rd;
```

Обратното не е общовалидно: обект от базовия клас може да е обикновен базов обект, а може и да се окаже част от производен обект. Затова не всеки обект от базовия тип може да се използва като обект от производния тип.

16.3.3 Влагане и наследяване

Дефиниция 16.11 (Композиция и наследяване)**.** Влагането представлява включване на обект като поле на друг клас и изразява отношение „има“. Например клас `Car` може да съдържа обект от клас `Engine`. Наследяването изразява отношение „е“ и позволява наследените операции да се използват като част от интерфейса на производния клас.

При двата механизма използваният клас участва във физическото представяне на обекта, цикличните зависимости са забранени и се използват негови полета и методи. При влагането могат да се съдържат няколко обекта от един и същи клас, а включеният обект може да бъде представен и чрез указател. При наследяването наследените методи са част от интерфейса на производния клас и могат да бъдат извиквани чрез него.

16.3.4 Вложени класове

Конспектът изисква „производни и *вложени* класове“. Двете думи в предходния раздел — влагане на *обект* — назовават композицията и не са същото нещо. Разликата трябва да се казва изрично, защото и двете се превеждат на български с „влагане“.

Дефиниция 16.12 (Вложен клас)**.** Вложен клас е клас, *декларацията* на който стои в тялото на друг клас. Вложеното е самият тип, а не обект. Името му е достъпно като `Външен::Вътрешен` и е подчинено на модификаторите за достъп на обхващащия клас, тъй като е негов член.

```
class List {
private:
    // вложен тип: съществува само вътре в List
    struct Node {
        int data;
        Node* next;
    };

    Node* first;

public:
    // вложен клас, видим отвън като List::Iterator
    class Iterator {
    public:
        explicit Iterator(Node* p) : current(p) {}
        int value() const { return current->data; }
        bool done() const { return current == nullptr; }
        void advance() { current = current->next; }
    private:
        Node* current;
    };
```

```
    Iterator begin() const { return Iterator(first); }
};
```

	Вложен клас	Влагане на обект (композиция)
Какво е вложено	типът (декларацията)	поле от вече съществуващ тип
Отношение	област на видимост	„има“
Влияе ли на размера	не	да, обектът е част от представянето
Достъп	`Външен::Вътрешен`	през името на полето

Вложеният клас е обикновен член на обхващащия: той има достъп до неговите `private` статични членове и типове, но *няма* собствен указател към обхващащ обект — в C++, за разлика от вътрешните класове на Java, връзка между инстанциите не се създава автоматично. Ако е нужна, тя се предава явно, както `Iterator` получава `Node*` по-горе.

Типичната употреба е точно тази: помощен тип, който е част от реализацията или от интерфейса на един клас и няма смисъл извън него — `Node` на списък, `Iterator` на контейнер, тип за състояние на автомат. Влагането държи името му извън глобалното пространство и изразява принадлежността му.

16.4 Капсулация и скриване на информация

Дефиниция 16.13. Капсулацията е принципът, при който данните и операциите над тях се обединяват в клас, а конкретната реализация и вътрешното състояние се скриват от външния код.

Класът защитава собственото си състояние чрез модификаторите за достъп. Полетата обикновено са `private`, а публичните методи образуват интерфейса на класа.

Пример:

```
class BankAccount {
private:
    double balance{};

public:
    BankAccount(double initialBalance)
        : balance{initialBalance} {
    }

    double getBalance() const {
        return balance;
    }

    void deposit(double amount) {
        balance += amount;
```

```
    }
};
```

Външният код не променя директно `balance`. Той използва предоставените операции. Така класът може да контролира промените в състоянието и да запазва необходимите свойства на обекта.

Капсулацията отделя интерфейса от реализацията. Методите могат да бъдат променени, без да се променя начинът, по който потребителят на класа ги извиква. Това изолира промените и намалява зависимостите между различните части на програмата.

16.4.1 Полиморфизъм при наследяване

При наследяване производният клас може да предостави собствена реализация на метод, наследен от базовия клас. За да се избира реализацията според действителния тип на обекта, базовият метод трябва да бъде виртуален.

```
class Animal {
public:
    virtual void talk() {
        std::cout << "Animal\n";
    }

    virtual ~Animal() {
    }
};

class Dog : public Animal {
public:
    void talk() override {
        std::cout << "Dog\n";
    }
};
```

Тогава:

```
Animal* animal = new Dog();
animal->talk();
delete animal;
```

извиква реализацията на `Dog::talk`. Това е подтипов полиморфизъм и се реализира чрез динамично свързване. Ключовата дума `override` не е задължителна, но показва намерението методът да предефинира виртуален метод от базовия клас.

Ако обект от производен клас се унищожава чрез указател към базов клас, деструкторът на базовия клас трябва да бъде виртуален. В противен случай унищожаването може да не извика правилно деструктора на производния клас.

Чисто виртуалният метод се записва така:

```
virtual void print() = 0;
```

Клас, който съдържа поне един чисто виртуален метод, е абстрактен. От абстрактен клас не могат да се създават обекти, но могат да се създават указатели и препратки към него. Производните класове трябва да реализират чисто виртуалните методи, за да могат да бъдат инстанцирани.

16.5 Изпитен фокус

За устен отговор трябва да могат да се възпроизведат:

- определението за ООП и основните му принципи: абстракция, капсулация, наследяване и полиморфизъм;
- разликата между клас и обект;
- структурата на класа: полета, методи и модификатори за достъп;
- предназначението и видовете конструктори;
- предназначението на деструктора и идеята на RAII;
- ролята на конструктора за копиране, преместващия конструктор и оператора за присвояване;
- указателят `this` и константните методи;
- предефинирането на операции и ограниченията върху него;
- разликата между статични и нестатични полета и методи;
- определението за базов и производен клас;
- публично, защитено и частно наследяване;
- редът на конструиране и унищожаване при наследяване;
- подтиповото отношение между производен и базов клас;
- разликата между влагане на *обект* — композиция, отношение „има“ — и наследяване, отношение „е“;
- определението за *вложен клас* (декларация на тип в тялото на друг клас) и разликата му от композицията; конспектът иска „производни и вложени класове“, което са две различни неща;
- капсулацията като скриване на реализацията и защита на инвариантите;
- ролята на виртуалните методи, `override` и виртуалния деструктор;
- чисто виртуалните методи и абстрактните класове.

Въпрос 17

ООП. Подтипов и параметричен полиморфизъм. Множествено наследяване.

По конспекта: какво трябва да мога

★ **Бележка на конспекта:** По този въпрос е възможно да бъдат дадени задачи за *практическата* част на изпита.

Подтипов полиморфизъм: виртуални функции, динамично свързване — §17.2, стр. 231.
Абстракция: абстрактни методи и абстрактни класове — §17.3, стр. 234.
Работа с обекти: масиви от обекти и масиви от указатели към обекти — §17.4, стр. 235.
Параметричен полиморфизъм: шаблони на функция и на клас — §17.5, стр. 236.
Множествено наследяване: §17.6, стр. 238.
Литература: [23], [24], [26], [47].

17.1 Обектно-ориентирано програмиране и полиморфизъм

Обектно-ориентираното програмиране (ООП) организира програмата около обекти, които съчетават данни и операции над тези данни. Данните се представят чрез член-данни на клас, а операциите – чрез член-функции. Класът описва общата структура и поведение, а обектът е конкретна инстанция на класа.

Дефиниция 17.1 (Наследяване). Наследяването позволява да се дефинира нов клас на основата на вече съществуващ клас. Съществуващият клас се нарича базов клас или родител, а новият – производен клас, наследник или подтип. Производният клас получава достъпните членове и методи на базовия клас и може да добавя нови или да променя поведението на наследените методи.

Дефиниция 17.2 (Полиморфизъм)**.** Полиморфизмът е свойството един елемент да има различно поведение в различни ситуации. Идеята е една и съща операция или интерфейс да може да бъде използвана с различни реализации.

В разглеждания материал се използват два основни вида полиморфизъм:

- подтипов полиморфизъм – реализира се чрез базови типове и техни подтипове;
- параметричен полиморфизъм – реализира се чрез типови параметри, например шаблони в C++.

Подтиповият полиморфизъм позволява обект от производен клас да бъде разглеждан чрез интерфейса на базовия клас, като при извикване на виртуална функция се изпълнява реализацията, съответстваща на действителния тип на обекта. Параметричният полиморфизъм позволява една обща функция или клас да се използва с различни типове.

17.2 Подтипов полиморфизъм и виртуални функции

17.2.1 Предефиниране на член-функция

Дефиниция 17.3 (Предефиниране на виртуална функция)**.** В C++ производен клас може да предефинира виртуална член-функция на базовия клас с подходяща сигнатура, съвместим връщан тип и съответстващи квалификатори. Това е function overriding и позволява динамично избиране на реализацията. Еднакво име на невиртуална функция само по себе си не осигурява динамичен полиморфизъм. Спецификаторът `override` изисква компилаторът да провери, че действително се предефинира виртуална функция.

```
class Base {
public:
    void func() {
        cout << "Base" << endl;
    }
};

class Derived : public Base {
public:
    void func() {
        cout << "Derived" << endl;
    }
};

int main() {
    Derived d;
    d.func();
```

```
    Base* b = new Derived();
    b->func();
}
```

Извикването чрез обекта d използва функцията от `Derived`. Извикването чрез указател от тип `Base*` обаче използва функцията от `Base`, когато функцията в базовия клас не е виртуална. Причината е, че изборът на функцията се основава на статичния тип на указателя.

За да се постигне подтипов полиморфизъм при работа с указател или референция към базов клас, функцията в базовия клас трябва да бъде обявена с ключовата дума `virtual`.

```
class Animal {
public:
    virtual void talk() {
        cout << "Hmmm" << endl;
    }
};

class Dog : public Animal {
public:
    void talk() override {
        cout << "Bau" << endl;
    }
};

class Cat : public Animal {
public:
    void talk() override {
        cout << "Meow" << endl;
    }
};
```

Могат да се създадат указатели към базовия клас, които сочат към обекти от различни производни класове:

```
Animal* a1 = new Dog();
Animal* a2 = new Cat();
Animal* a3 = new Animal();

a1->talk();
a2->talk();
a3->talk();
```

Резултатът е съответно:

```
Bau
Meow
Hmmm
```

И трите указателя имат един и същ статичен тип `Animal*`, но сочат към обекти с различен действителен тип. Поради това виртуалното извикване избира различна реализация на `talk`.

Дефиниция 17.4 (Подтипов полиморфизъм)**.** Подтиповият полиморфизъм е реализиране на различно поведение чрез подтипове, които споделят общ интерфейс с базов тип. В C++ той се реализира чрез предефиниране на виртуални член-функции и се проявява по време на изпълнение.

Ключовата дума `override` не е задължителна. Тя се поставя в производния клас след декларацията на предефинираната функция и показва намерението тя да предефинира виртуална функция от базовия клас. Употребата ѝ е препоръчителна, защото позволява да се открият грешки в името, параметрите или типа на функцията.

17.2.2 Статично и динамично свързване

За да определи коя функция да извика, компилаторът сравнява действителните и формалните параметри и избира най-точното съвпадение. Когато изборът се извършва по време на компилация, говорим за статично, или ранно, свързване.

При динамичното свързване изборът на извикваната функция се извършва по време на изпълнение. При виртуална функция той зависи от действителния клас на обекта, към който сочи указателят или чиято референция се използва.

```
Animal* d = new Dog();    d->talk();
```

В този случай се извиква методът `talk` на `Dog`, защото действителният обект е от този клас.

Статичното свързване се използва например при претоварване на функции (function overloading). При претоварването има няколко функции с едно и също име, но с различни параметри, а конкретната функция се избира според типовете на аргументите. Предефинирането на невиртуална функция също не осигурява динамичен полиморфизъм.

† **Допълнение 17.5 (Синтаксис на извикването).** В примерите с указатели извикването трябва да бъде записано като `b->func()` или `a1->talk()`, а не като `b.func()` при `b`, който е указател. Съществото за полиморфизма е използването на виртуална функция, а не тази синтактична разлика.

В C++ функциите са статично свързани по подразбиране. Динамичното свързване за член-функции се задава чрез `virtual`. Конструкторите не могат да бъдат виртуални. Виртуални могат да бъдат член-функции на класа.

17.2.3 Виртуална таблица и управление на паметта

За класове с виртуални функции се създава таблица с указатели към виртуалните функции, наричана виртуална таблица. Всеки обект от такъв клас съдържа указател към съответната виртуална таблица. При извикване на виртуална функция чрез указател или референция към базов клас се използва тази информация, за да се избере реализацията на действителния клас.

Виртуалният полиморфизъм често се използва заедно с динамично заделяне на памет:

```
Base* b = new Derived();
```

След приключване на употребата на обекта паметта трябва да бъде освободена:

```
delete b;
```

Ако това не се направи, възниква изтичане на памет. Освен това, ако базовият клас се използва полиморфно, неговият деструктор трябва да бъде виртуален. В противен случай изтриването чрез указател към базовия клас може да не извика правилно деструктора на производния обект и поведението е неопределено.

```
class Base {
public:
    virtual ~Base() {}
    virtual void func() {}
};
```

Наличието на виртуални функции е основание деструкторът на базовия клас да бъде виртуален, когато обектите ще се унищожават чрез указатели към базовия клас.

17.3 Абстрактни методи и абстрактни класове

17.3.1 Чисто виртуални функции

Чисто виртуална функция е виртуална функция, за която в базовия клас не е дадена реализация:

```
virtual void talk() = 0;
```

Чисто виртуалната функция задава изискване към производните класове. Те трябва да предоставят собствена реализация, ако трябва да могат да се инстанцират.

Дефиниция 17.6 (Абстрактен клас)**.** Абстрактен е клас, който съдържа поне една чисто виртуална функция.

От абстрактен клас не могат да се създават обекти. Възможно е обаче да се създават указатели и референции към него. Такива указатели и референции се използват като общ интерфейс към обекти от конкретни производни класове.

```
class Animal {
public:
    virtual void talk() = 0;
};

class Dog : public Animal {
public:
    void talk() override {
        cout << "Bau" << endl;
    }
};
```

Класът `Animal` е абстрактен и не може да се използва за създаване на обект. `Dog` реализира чисто виртуалната функция и поради това може да се инстанцира. Ако производен клас не предефинира чисто виртуалната функция, той също остава абстрактен.

Когато класът съдържа само абстрактни методи и няма нестатични член-данни, той може да се разглежда като интерфейс в смисъла, използван например в Java. В този случай класът задава операции, които конкретните производни класове трябва да реализират.

17.4 Масиви от обекти и указатели към обекти

Масивът е хомогенен контейнер: неговите елементи са от един и същи тип. Наследяването и полиморфизмът позволяват масив от указатели към базов клас да сочи към обекти от различни производни класове.

Масив от самите базови обекти не осигурява полиморфно съхраняване на производни обекти:

```
Base arr[] = { Base(), Derived() };
```

При такава конструкция елементите на масива са обекти от тип `Base`. Производната част на `Derived` не се съхранява като отделен полиморфен елемент в масива.

Полиморфното поведение се постига чрез масив от указатели:

```
Base* arr[] = {
    new Base(),
    new Derived()
};
```

Елементите на масива са от един и същи тип `Base*`, но могат да сочат към обекти от различни класове. Ако съответните функции са виртуални, извикването през елемент на масива ще избере реализацията според действителния тип на посочения обект.

При използване на динамично заделени обекти трябва да се освободи паметта за всеки елемент. Ако се изтриват чрез указатели към базовия клас, деструкторът на базовия клас трябва да бъде виртуален.

17.5 Параметричен полиморфизъм

17.5.1 Идея и шаблони

Параметричният полиморфизъм е възможността функция или клас да работи с различни типове чрез въвеждане на типов параметър. Вместо да се създава отделна реализация за всеки конкретен тип, се описва обща форма, която използва параметричен тип.

В C++ параметричният полиморфизъм се реализира чрез шаблони. Шаблонът е правило за генериране на код. Той описва обща функция или клас, който работи с неопределен тип по унифициран начин.

Пример за шаблонна функция е функция за намиране на по-големия от два елемента:

```
template <typename T>
T getMax(const T& a, const T& b) {
    return a > b ? a : b;
}
```

Типовият параметър T участва в типа на връщания резултат и в типовете на параметрите. Той може да участва и в тялото на функцията.

Функцията може да бъде използвана с различни типове:

```
getMax(1, 5);          // 5
getMax(1.2, 3.14);     // 3.14
getMax('u', 'a');      // 'u'
```

Шаблонът работи за типовете, за които е дефинирана необходимата операция. В конкретния пример типът трябва да поддържа операцията >.

Дефиниция 17.7 (Инстанциране на шаблон)**.** При използване на шаблон с конкретен тип се генерира конкретна функция, която след това се компилира. За различни типове могат да бъдат генерирани различни конкретни функции.

Самият шаблон не е конкретна функция за определен тип. Той е описание, от което компилаторът генерира функции при употреба. Поради това параметричният полиморфизъм се реализира по време на компилация.

Шаблоните могат да имат повече от един типов параметър. Общата форма може да бъде представена така:

```
template <typename T, typename U>
```

Параметрите могат да имат и стойности по подразбиране, когато това е позволено от синтаксиса на конкретния шаблон.

17.5.2 Шаблони на класове

Шаблоните се използват не само за функции, но и за класове. Типовите параметри могат да участват в типовете на член-данните, в типовете на параметрите на член-функциите, в типа на връщания резултат и в телата на член-функциите.

```
template <typename T>
class Array {
private:
    T* arr;
    int size, capacity;

public:
    Array(int _cap) : capacity(_cap) {
        arr = new T[capacity];
        size = 0;
    }

    void add(const T& elem) {
        if (size < capacity) {
            arr[size++] = elem;
        }
    }
};
```

Класът може да бъде използван с конкретно указан тип:

```
Array<int> a(5);
a.add(4);
```

В този случай се създава конкретен клас, в който `T` е заместен с `int`. Възможно е да се използват и други типове, например `double` или потребителски класове, при условие че необходимите операции са налични.

† **Допълнение 17.8 (Размер на параметричен тип).** В източниците е отбелязано, че не може да се създаде обект директно от неспециализирания шаблон на клас, тъй като размерът на `T` не е известен. Обект се създава от конкретна инстанциация, например `Array<int>`. Това е причината изразът `sizeof(T)` да има смисъл едва след заместване на `T` с конкретен тип.

Както при шаблонните функции, шаблонът на клас не се компилира като конкретен клас сам по себе си. При използването му с конкретни параметри се генерира съответният тип. Компилират се тези член-функции, които са необходими при употребата на конкретната инстанциация.

Параметричният и подтиповият полиморфизъм се различават по начина, по който се избира поведението:

- при подтипов полиморфизъм изборът на виртуална реализация става по време на изпълнение;
- при параметричен полиморфизъм конкретният код се генерира и компилира за конкретните типове.

17.6 Множествено наследяване

17.6.1 Основна конструкция

В C++ един клас може да наследява едновременно от няколко базови класа:

```
class A {
};

class B {
};

class C : public A, public B {
};
```

Класът `C` наследява достъпните членове и методи на `A` и `B`. Производният клас може да добавя собствени членове и да предефинира наследени виртуални функции.

Конструкторите на базовите класове се извикват в реда, в който базовите класове са записани в декларацията на производния клас. При `C : public A, public B` редът е:

$$A \longrightarrow B \longrightarrow C.$$

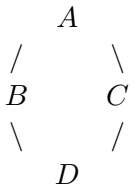
Първо се конструира базовата част на `A`, след това базовата част на `B`, а накрая се изпълнява конструкторът на `C`. Деструкторите се извикват в обратен ред:

$$C \longrightarrow B \longrightarrow A.$$

Така първо се унищожава производната част, след това базовата част на `B`, а накрая базовата част на `A`.

17.6.2 Диамантен проблем

При множественото наследяване може да възникне диамантен проблем. Нека имаме следната структура:



Класовете B и C наследяват A, а класът D наследява и B, и C:

```
class A {
};

class B : public A {
};

class C : public A {
};

class D : public B, public C {
};
```

При създаване на обект от тип D в него се включват две базови части от тип A: едната идва през B, а другата – през C. Следователно данните и методите на A са представени два пъти.

Това води до двусмислие. Ако се извика член на A през обект от тип D, компилаторът не може да определи дали се има предвид копието, наследено чрез B, или копието, наследено чрез C. В резултат достъпът може да доведе до грешка при компилация.

17.6.3 Виртуално наследяване

Решението на дублирането на общата базова част е виртуалното наследяване. То се задава при наследяването на A от B и C:

```
class B : virtual public A {
};

class C : virtual public A {
};

class D : public B, public C {
};
```

При виртуално наследяване класът D съдържа една обща базова част от тип A, независимо че до нея водят два наследствени пътя. Конструкторът на виртуалния базов клас се извиква само веднъж.

Виртуалните базови класове се конструират преди останалите базови части. При създаване на обект от тип D най-производният клас, тоест D, е отговорен за директното извикване на конструктора на виртуалния базов клас A. След това се конструират останалите базови части в реда на декларацията, а накрая – самата производна част.

† **Допълнение 17.9 (Неоднозначност при виртуално наследяване).** Виртуалното наследяване премахва дублирането на общата базова подобектна част, но само по себе си не решава всяка възможна неоднозначност между членове или операции, наследени от `B` и `C`. Ако двата класа предоставят различни членове или различни реализации със същото име, производният клас трябва да има еднозначен избор. Следователно `virtual` решава проблема с общата базова част, но не замества разрешаването на всички други конфликти в интерфейсите.

Основният резултат от виртуалното наследяване е следният:

- без виртуално наследяване `D` съдържа две копия на `A`;
- с виртуално наследяване `D` съдържа една обща виртуална базова част `A`;
- конструкторът на виртуалния базов клас се извиква един път;
- конструирането и унищожаването следват специален ред, съобразен с виртуалната базова част.

17.7 Изпитен фокус

За устен отговор трябва да могат да бъдат възпроизведени следните определения и идеи:

- определение за ООП, клас, обект, базов клас и производен клас;
- определение за полиморфизъм и разлика между подтипов и параметричен полиморфизъм;
- ролята на `virtual` и `override`;
- разликата между статично и динамично свързване;
- защо извикването през `Base*` може да избере метода на производния клас;
- чисто виртуална функция и абстрактен клас;
- защо масив от указатели към базов клас може да съдържа обекти от различни производни класове;
- шаблонна функция и шаблон на клас;
- кога и как се инстанцират шаблоните;
- изискването типовете, използвани с шаблон, да поддържат операциите от тялото на шаблона;
- синтаксисът и редът на конструиране и унищожаване при множествено наследяване;
- диамантеният проблем и причината за възникването на две копия на общия базов клас;

- виртуално наследяване и ролята му за споделянето на една обща базова част;
- необходимостта от виртуален деструктор при полиморфно използване на базов клас.

Въпрос 18

Структури от данни. Стек, опашка, списък, дърво.

По конспекта: какво трябва да мога

Официално заглавие: „Структури от данни. Стек, опашка, списък, дърво. *Основни операции върху тях. Реализация*.“

> ★ **Бележка на конспекта:** За изпита ще бъдат избрани **две** от посочените структури, които да бъдат описани. Възможни са и задачи за *практическата* част на изпита.

Дефиниция: на понятието структура от данни — §18.1, стр. 243.
За всяка структура се иска еднакъв набор: логическо описание; характеристики на реализациите; сложност на операциите; **дефиниране на клас**, използващ една от реализациите.
Реализации: статична, динамична и свързана — назованите от конспекта три категории — §18.2, стр. 243.
Списък: с една и две връзки; сложност на добавяне, премахване и намиране — §18.5; клас — §18.5.3, стр. 257.
Стек: логическо описание и представяния — §18.3; клас — §18.3.3, стр. 247.
Опашка: логическо описание и представяния — §18.4; клас — §18.4.3, стр. 252.
Дървовидни структури: кореново и двоично кореново дърво; начини за представяне в паметта — §18.6; клас — §18.6.4, стр. 262.
Двоично кореново дърво за търсене: логическо описание, представяне, сложност — §18.7; клас — §18.7.5, стр. 267.
Обобщение: таблица със сложностите и допусканията — §18.8.1, стр. 270.
Литература: [11], [15], [25], [28].

18.1 Структури от данни

Дефиниция 18.1 (Структура от данни)**.** Структура от данни е организирана информация, която може да бъде описана, създадена и обработвана с помощта на програма.

При определяне на една структура от данни се разграничават два аспекта:

- логическо описание на структурата – описание чрез нейната декомпозиция, чрез по-прости структури и чрез операциите над нея, сведени до по-прости операции;
- физическо представяне – методът, по който структурата се разполага в паметта на компютъра.

Логическата структура определя какви са елементите, как са свързани и какви операции са допустими. Физическото представяне определя как тази информация се реализира чрез клетки памет, масиви, указатели и други програмни средства.

В настоящата глава се разглеждат четири основни структури от данни: стек, опашка, списък и дърво.

18.2 Трите категории реализации

Конспектът различава *статична*, *динамична* и *свързана* реализация за стека и за опашката. Разликата е в това как се управлява паметта, а не в логическото описание, което остава едно и също.

Дефиниция 18.2 (Статична (последователна) реализация)**.** Елементите се пазят в масив с *фиксиран* капацитет N, заделен предварително. Достъпът по индекс е константен, но при N елемента следващото включване е невъзможно: настъпва *препълване*. Паметта е заета изцяло независимо от броя на реално съхранените елементи.

Дефиниция 18.3 (Динамична реализация (масив с преоразмеряване))**.** Елементите се пазят в масив, заделен в динамичната памет, чийто капацитет се *удвоява*, когато се запълни: заделя се нов блок с двоен размер, елементите се копират и старият блок се освобождава. Няма предварителна граница за броя елементи; достъпът по индекс остава константен.

Твърдение 18.4 (Амортизирана цена на включването)**.** При удвояване на капацитета всяко включване струва $O(1)$ *амортизирано*, макар отделно включване да струва $O(n)$, когато предизвика преоразмеряване.

Доказателство. Разглеждаме редица от n включвания в стек, започнал с капацитет $c_0 \geq 1$. Преоразмеряванията стават при размери $c_0, 2c_0, 4c_0, \ldots$, а цената на всяко е пропорционална на броя копирани елементи. Общата цена на копиранията е

$$c_0 + 2c_0 + 4c_0 + \cdots + 2^k c_0 < 2 \cdot 2^k c_0 \leq 2n \cdot \frac{2^k c_0}{n},$$

където $2^k c_0 \leq n$ е последният капацитет преди спиране; следователно сумата е по-малка от $2n$. Заедно с n-те константни записа общата цена е $O(n)$, тоест $O(1)$ на включване средно. □

Дефиниция 18.5 (Свързана реализация)**.** Всеки елемент се пази в отделен възел с указател към следващия. Няма преоразмеряване и няма препълване, но за всеки елемент се плаща допълнителна памет за указателя, а достъпът до i-тия елемент е $O(i)$, защото се минава през веригата.

Забележка 18.6. Изборът не е въпрос на вкус. Статичната реализация се използва при известна горна граница и при забрана за динамична памет; динамичната — когато е нужен пряк достъп по индекс без такава граница; свързаната — когато преобладават включвания и изключвания в известна позиция.

18.3 Стек

Дефиниция 18.7 (Стек)**.** Стекът е крайна редица от елементи от един и същи тип, при която включването и изключването на елементи са допустими само в единия край на редицата. Този край се нарича връх на стека.

Стекът е линейна динамична структура от данни. При него е възможен пряк достъп само до елемента на върха. За останалите елементи достъпът се осъществява последователно, започвайки от върха.

★ **Дефиниция 18.8 (LIFO).** Правилото LIFO (*Last In, First Out*) означава, че последно включеният в стека елемент е първият изключен от него.

Основните операции върху стек са:

- *включване* (*push*) – добавяне на елемент на върха;
- *изключване* (*pop*) – премахване на елемента от върха;
- *достъп до върха* – прочитане на елемента на върха без премахването му;
- *проверка за празен стек*;
- *търсене* на елемент.

Включването и изключването на елемент са константни операции, тоест имат сложност $O(1)$. Търсенето на елемент има линейна сложност $O(n)$, тъй като в най-общия случай трябва да се прегледат всички елементи.

18.3.1 Последователно представяне на стек

При последователното, или статичното, представяне предварително се заделя блок от памет, обикновено реализиран чрез масив. В този блок стекът нараства и се свива. Поддържа се индекс или указател към върха.

Ако масивът е с размер N, стекът може да бъде описан чрез масива `data` и целочислен индекс `top`. При празен стек `top` има стойност, която показва, че няма валиден елемент. При включване стойността на `top` се увеличава и новият елемент се записва на съответната позиция. При изключване елементът на `top` се извлича, след което `top` се намалява.

Примерна реализация на стек от цели числа чрез масив е:

```
const int Capacity = 100;

struct Stack {
  int data[Capacity];
  int top;
};

void init(Stack& s) {
  s.top = -1;
}

bool empty(const Stack& s) {
  return s.top == -1;
}

bool full(const Stack& s) {
  return s.top == Capacity - 1;
}

bool push(Stack& s, int value) {
  if (full(s)) {
    return false;
  }

  ++s.top;
  s.data[s.top] = value;
  return true;
}

bool pop(Stack& s, int& value) {
  if (empty(s)) {
    return false;
  }

  value = s.data[s.top];
```

```
  --s.top;
  return true;
}
```

При тази реализация възниква състояние *препълване*, ако се направи опит за включване в стек, който е достигнал капацитета си. При изключване от празен стек възниква състояние *празен стек* или неуспешна операция. Размерът на последователно представения стек е ограничен от предварително заделения блок.

18.3.2 Свързано представяне на стек

При свързаното представяне последователните елементи на стека могат да се намират на различни места в паметта. Връзката между два съседни елемента се осъществява чрез указател към следващия елемент. Указателят на последния елемент обикновено има стойност `nullptr`.

За задаване на стека е достатъчен указател към неговия връх. Един елемент на стек от цели числа може да има вида:

```
struct StackItem {
  int data;
  StackItem* link;
};
```

Полето `data` съдържа информацията, а полето `link` съдържа адреса на следващия елемент към вътрешността на стека.

```
struct LinkedStack {
  StackItem* top;
};

void init(LinkedStack& s) {
  s.top = nullptr;
}

bool empty(const LinkedStack& s) {
  return s.top == nullptr;
}

void push(LinkedStack& s, int value) {
  StackItem* item = new StackItem;
  item->data = value;
  item->link = s.top;
  s.top = item;
}

bool pop(LinkedStack& s, int& value) {
```

```
  if (empty(s)) {
    return false;
  }

  StackItem* item = s.top;
  value = item->data;
  s.top = item->link;
  delete item;
  return true;
}
```

При включване новият елемент се свързва с досегашния връх и става нов връх. При изключване се запазва стойността на върха, указателят се премества към следващия елемент, а старият елемент се освобождава. И двете операции са с константна сложност.

Свързаното представяне не изисква предварително зададен максимален брой елементи, но за всеки елемент се съхранява и указател. То е подходящо, когато размерът на стека се изменя динамично.

18.3.3 Клас за стек (динамична реализация)

Конспектът иска *дефиниране на клас* за стек, използващ една от реализациите. Функциите по-горе работят върху отворена структура: всеки може да пипне `top` и да наруши съгласуваността. Класът затваря представянето и поема отговорност за паметта.

Инвариант на класа: `data` сочи блок от точно `capacity` клетки в динамичната памет, $0 \le$ `size` $\le$ `capacity`, валидни са елементите `data[0..size-1]`, а върхът е `data[size-1]`.

```
#include <stdexcept>

class Stack {
  int* data;       // блок с capacity клетки
  int size;        // брой съхранени елементи
  int capacity;    // размер на блока

  void resize(int newCapacity) {
    int* moved = new int[newCapacity];
    for (int i = 0; i < size; ++i) moved[i] = data[i];
    delete[] data;
    data = moved;
    capacity = newCapacity;
  }

public:
  Stack() : data(new int[4]), size(0), capacity(4) {}

  Stack(const Stack& other)
```

```
      : data(new int[other.capacity]),
        size(other.size), capacity(other.capacity) {
    for (int i = 0; i < size; ++i) data[i] = other.data[i];
  }

  Stack& operator=(const Stack& other) {
    if (this != &other) {
      int* moved = new int[other.capacity];
      for (int i = 0; i < other.size; ++i)
        moved[i] = other.data[i];
      delete[] data;
      data = moved;
      size = other.size;
      capacity = other.capacity;
    }
    return *this;
  }

  ~Stack() { delete[] data; }

  bool empty() const { return size == 0; }

  void push(int value) {
    if (size == capacity) resize(2 * capacity);
    data[size++] = value;
  }

  int pop() {
    if (empty())
      throw std::underflow_error("pop от празен стек");
    return data[--size];
  }

  int top() const {
    if (empty())
      throw std::underflow_error("top от празен стек");
    return data[size - 1];
  }
};
```

Трите члена — копиращ конструктор, операция за присвояване и деструктор — вървят заедно. Ако липсват, копирането на `Stack` копира указателя, двата обекта освобождават един и същи блок и програмата се разрушава. Това е управлението на ресурси, което Въпрос 16 нарича RAII: заделянето става в конструктора, освобождаването — в деструктора, и никой път през кода не може да ги пропусне.

Същият клас в *статична* реализация се получава така: `data` става масив с фиксиран капацитет, `resize` отпада, а `push` хвърля `std::overflow_error` при пълен стек. В *свързана* реализация трите полета се заменят с единствен указател `top` към възел, както в §18.3.2.

18.4 Опашка

Дефиниция 18.9 (Опашка)**.** Опашката е крайна редица от елементи от един и същи тип, при която включването е допустимо в единия край, а изключването – в другия край.

Краят, в който се добавят елементи, се нарича *край на опашката*, а краят, от който се премахват елементи, се нарича *начало на опашката*. Пряк достъп е възможен само до елемента в началото на опашката.

★ **Дефиниция 18.10 (FIFO).** Правилото FIFO (*First In, First Out*) означава, че първият включен в опашката елемент е първият изключен от нея.

Основните операции върху опашка са:

- включване на елемент в края;
- изключване на елемент от началото;
- достъп до елемента в началото;
- проверка за празна опашка;
- търсене на елемент.

Добавянето и премахването на елемент са операции с константна сложност $O(1)$. Търсенето, когато е разрешено, е линейно по броя на елементите.

18.4.1 Последователно представяне на опашка

При последователното представяне предварително се заделя блок памет, реализиран чрез масив. Обикновено блокът се разглежда като цикличен. Това означава, че след достигане на последната позиция следва първата позиция, ако в началото има освободено място.

Необходимо е да се пазят поне два индекса:

- индекс на началото на опашката;
- индекс на следващата свободна позиция в края на опашката.

Необходимо е също да се знае броят на елементите или да се използва допълнителен признак, който различава празна от пълна опашка.

```
const int Capacity = 100;

struct Queue {
  int data[Capacity];
  int first;
  int last;
  int count;
};

void init(Queue& q) {
  q.first = 0;
  q.last = 0;
  q.count = 0;
}

bool empty(const Queue& q) {
  return q.count == 0;
}

bool full(const Queue& q) {
  return q.count == Capacity;
}

bool enqueue(Queue& q, int value) {
  if (full(q)) {
    return false;
  }

  q.data[q.last] = value;
  q.last = (q.last + 1) % Capacity;
  ++q.count;
  return true;
}

bool dequeue(Queue& q, int& value) {
  if (empty(q)) {
    return false;
  }

  value = q.data[q.first];
  q.first = (q.first + 1) % Capacity;
  --q.count;
  return true;
}
```

При включване елементът се записва на позицията, определена от `last`, след което индексът се придвижва циклично. При изключване се извлича елементът на позиция `first` и началният индекс също се придвижва циклично.

18.4.2 Свързано представяне на опашка

Свързаното представяне на опашка е подобно на свързаното представяне на стек. Елементите се намират на различни места в паметта и са свързани чрез указатели. За разлика от стека, при опашката са нужни два указателя:

- указател към началото на опашката;
- указател към края на опашката.

Елементът на опашката може да бъде описан така:

```
struct QueueItem {
  int data;
  QueueItem* link;
};
```

При празна опашка и двата указателя имат стойност `nullptr`. При първото включване и двата трябва да бъдат насочени към новия елемент. При следващи включвания новият елемент се свързва след досегашния край, а указателят към края се премества към него. При изключване се премахва елементът от началото и указателят към началото се премества към следващия елемент. Ако след премахването опашката стане празна, указателят към края също се установява на `nullptr`.

```
struct LinkedQueue {
  QueueItem* first;
  QueueItem* last;
};

void init(LinkedQueue& q) {
  q.first = nullptr;
  q.last = nullptr;
}

bool empty(const LinkedQueue& q) {
  return q.first == nullptr;
}

void enqueue(LinkedQueue& q, int value) {
  QueueItem* item = new QueueItem;
  item->data = value;
  item->link = nullptr;
```

```
  if (q.last == nullptr) {
    q.first = item;
    q.last = item;
  } else {
    q.last->link = item;
    q.last = item;
  }
}

bool dequeue(LinkedQueue& q, int& value) {
  if (empty(q)) {
    return false;
  }

  QueueItem* item = q.first;
  value = item->data;
  q.first = item->link;

  if (q.first == nullptr) {
    q.last = nullptr;
  }

  delete item;
  return true;
}
```

При наличие на указател към края включването е константна операция. Без такъв указател намирането на последния елемент би изисквало преминаване през опашката.

18.4.3 Клас за опашка (свързана реализация)

Инвариант: или `head == nullptr` и `tail == nullptr` (празна опашка), или `head` сочи първия възел, `tail` — последния, `tail->next == nullptr`, и всеки възел е достижим от `head`.

```
#include <stdexcept>

class Queue {
  struct Node {
    int data;
    Node* next;
  };

  Node* head;   // оттук се изключва
  Node* tail;   // тук се включва
```

```
  void clear() {
    while (head != nullptr) {
      Node* item = head;
      head = head->next;
      delete item;
    }
    tail = nullptr;
  }

  void copyFrom(const Queue& other) {
    head = tail = nullptr;
    for (Node* it = other.head; it != nullptr; it = it->next)
      enqueue(it->data);
  }

public:
  Queue() : head(nullptr), tail(nullptr) {}
  Queue(const Queue& other) { copyFrom(other); }

  Queue& operator=(const Queue& other) {
    if (this != &other) {
      clear();
      copyFrom(other);
    }
    return *this;
  }

  ~Queue() { clear(); }

  bool empty() const { return head == nullptr; }

  void enqueue(int value) {
    Node* item = new Node{value, nullptr};
    if (tail != nullptr) tail->next = item;
    else head = item;
    tail = item;
  }

  int dequeue() {
    if (empty())
      throw std::underflow_error("dequeue от празна опашка");
    Node* item = head;
    int value = item->data;
    head = head->next;
    if (head == nullptr) tail = nullptr;
```

```
    delete item;
    return value;
  }

  int front() const {
    if (empty())
      throw std::underflow_error("front от празна опашка");
    return head->data;
  }
};
```

Двата указателя са това, което прави и включването, и изключването $O(1)$: без `tail` включването в края би изисквало обхождане на цялата верига. Присвояването на `tail = nullptr` при изпразване не е козметика — без него `tail` остава да сочи освободена памет.

18.5 Списък

Дефиниция 18.11 (Списък)**.** Списъкът е крайна редица от елементи от един и същи тип, при която включването и изключването са допустими от произволно място в редицата.

За разлика от стека и опашката списъкът позволява работа с произволна позиция. Възможен е достъп до всички елементи, който може да бъде пряк или непряк в зависимост от физическото представяне.

Разглеждат се три основни начина за представяне на списък:

- последователно представяне;
- едносвързано представяне;
- двусвързано представяне.

Последователното представяне не се използва като основен начин за списък, тъй като включването и изключването в произволно място обикновено изискват преместване на елементи.

18.5.1 Едносвързан списък

При едносвързаното представяне елементите се съхраняват на различни места в паметта. Всеки елемент съдържа информация и указател към следващия елемент. Указателят на последния елемент е `nullptr`.

За задаване на списъка е достатъчен указател към началото. Ако се добави и указател към края, включването в края може да се извършва за константно време.

```
struct ListItem {
  int data;
  ListItem* next;
};

struct List {
  ListItem* first;
  ListItem* last;
};
```

Основните операции са:

- включване в началото;
- включване в края;
- включване след даден елемент;
- изключване от началото;
- изключване след даден елемент;
- търсене;
- обхождане на списъка.

Включването и изключването в началото са константни операции. Включването в края също е константно, ако се поддържа указател към последния елемент. Включването или изключването в произволна позиция, когато позицията трябва първо да бъде намерена, е линейно. Търсенето също е линейна операция.

```
void insertFirst(List& list, int value) {
  ListItem* item = new ListItem;
  item->data = value;
  item->next = list.first;
  list.first = item;

  if (list.last == nullptr) {
    list.last = item;
  }
}

bool removeFirst(List& list, int& value) {
  if (list.first == nullptr) {
    return false;
  }

  ListItem* item = list.first;
  value = item->data;
```

```
  list.first = item->next;

  if (list.first == nullptr) {
    list.last = nullptr;
  }

  delete item;
  return true;
}

void insertLast(List& list, int value) {
  ListItem* item = new ListItem;
  item->data = value;
  item->next = nullptr;

  if (list.last == nullptr) {
    list.first = item;
    list.last = item;
  } else {
    list.last->next = item;
    list.last = item;
  }
}
```

За изключване на елемент, който не е първи, е необходим указател към предходния елемент. След промяната указателят на предходния елемент трябва да прескочи изключвания елемент и да сочи към следващия.

18.5.2 Двусвързан списък

При двусвързания списък всеки елемент има два указателя: към следващия и към предходния елемент. Елементът може да бъде описан чрез:

```
struct DLListItem {
  int data;
  DLListItem* next;
  DLListItem* prev;
};

struct DLList {
  DLListItem* first;
  DLListItem* last;
};
```

За списъка обикновено се пазят указатели към началото и края. Допълнително може да се пази указател към текущия елемент, което улеснява някои операции по вмъкване, изключване и търсене.

При едносвързан списък движението по връзките е само напред. При двусвързан списък може да се преминава и в двете посоки. Изключването на текущ елемент се извършва чрез свързване на неговия предходен и следващ елемент един с друг. При крайни елементи единият от съответните указатели е `nullptr`.

```
void insertAfter(DLList& list,
                 DLListItem* current,
                 int value) {
  if (current == nullptr) {
    return;
  }

  DLListItem* item = new DLListItem;
  item->data = value;
  item->prev = current;
  item->next = current->next;

  if (current->next != nullptr) {
    current->next->prev = item;
  } else {
    list.last = item;
  }

  current->next = item;
}
```

Двусвързаното представяне изисква повече памет заради втория указател, но улеснява изключването и движението в обратна посока.

18.5.3 Клас за списък (двусвързана реализация)

Инвариант: `count` е броят на възлите; предшественикът на `first` и наследникът на `last` са `nullptr`; за всеки възел v с наследник е изпълнено `v->next->prev == v`; при празен списък и двата указателя са `nullptr`.

```
class List {
  struct Node {
    int data;
    Node* prev;
    Node* next;
  };

  Node* first;
  Node* last;
  int count;
```

```
  Node* nodeAt(int pos) const {      // изисква 0 <= pos < count
    Node* it = first;
    for (int i = 0; i < pos; ++i) it = it->next;
    return it;
  }

  void clear() {
    while (first != nullptr) {
      Node* item = first;
      first = first->next;
      delete item;
    }
    last = nullptr;
    count = 0;
  }

  void copyFrom(const List& other) {
    first = last = nullptr;
    count = 0;
    for (Node* it = other.first; it != nullptr; it = it->next)
      pushBack(it->data);
  }

public:
  List() : first(nullptr), last(nullptr), count(0) {}
  List(const List& other) { copyFrom(other); }

  List& operator=(const List& other) {
    if (this != &other) {
      clear();
      copyFrom(other);
    }
    return *this;
  }

  ~List() { clear(); }

  int size() const { return count; }
  bool empty() const { return count == 0; }

  void pushBack(int value) {
    Node* item = new Node{value, last, nullptr};
    if (last != nullptr) last->next = item;
    else first = item;
    last = item;
    ++count;
```

```
  }

  void insertAt(int pos, int value) {  // изисква 0 <= pos <= count
    if (pos == count) {
      pushBack(value);
      return;
    }
    Node* next = nodeAt(pos);
    Node* item = new Node{value, next->prev, next};
    if (next->prev != nullptr) next->prev->next = item;
    else first = item;
    next->prev = item;
    ++count;
  }

  void removeAt(int pos) {           // изисква 0 <= pos < count
    Node* item = nodeAt(pos);
    if (item->prev != nullptr) item->prev->next = item->next;
    else first = item->next;
    if (item->next != nullptr) item->next->prev = item->prev;
    else last = item->prev;
    delete item;
    --count;
  }

  // индекс на първото срещане или -1
  int find(int value) const {
    int i = 0;
    for (Node* it = first; it != nullptr; it = it->next, ++i)
      if (it->data == value) return i;
    return -1;
  }
};
```

Двете връзки правят премахването на *известен* възел константно: при едносвързан списък трябва да се търси предшественикът, тоест $O(n)$. Цената на `insertAt` и `removeAt` тук е $O(pos)$ — разходът е в намирането на позицията, не в пренавързването, което е $O(1)$.

18.6 Дърво

Дефиниция 18.12 (Дърво). Дърво е свързан ацикличен граф.

Дървото е разклонена структура от данни. Елементите му се наричат *върхове* или *възли*. Един от върховете се разглежда като корен. Всеки връх може да има произволен брой наследници. Връзките между върховете определят йерархична структура.

Рекурсивното определение на дърво е:

- празното дърво е дърво;
- дървото с единствен връх, наречен корен, е дърво;
- дърво с корен и произволен брой наследници, всеки от които е корен на дърво, също е дърво.

Предоставя се пряк достъп до корена, а до останалите върхове достъпът се осъществява чрез връзките в структурата.

18.6.1 Двойно дърво

Дефиниция 18.13 (Двойно дърво)**.** Двойното дърво от тип T е рекурсивна структура от данни, която е или празна, или е образувана от данна от тип T, наречена корен, двойно дърво от тип T, наречено ляво поддърво, и двойно дърво от тип T, наречено дясно поддърво.

Множеството от върховете на двойното дърво се определя рекурсивно. Празното двойно дърво няма върхове. Непразното двойно дърво има корен и всички върхове на лявото и дясното поддърво.

Левият наследник на връх е коренът на лявото му поддърво, ако такова поддърво съществува. Аналогично се определя десният наследник.

Дефиниция 18.14 (Листа и вътрешни върхове)**.** Върхове без наследници се наричат *листа*. Вътрешни върхове са върховете с поне едно дете, включително корена, когато той има дете.

Свързаното представяне на двойно дърво се реализира чрез указател към структура с три полета: информационно поле и два адреса за лявото и дясното поддърво.

```
struct Node {
  int data;
  Node* left;
  Node* right;
};
```

Празното поддърво се представя с указател `nullptr`.

18.6.2 Операции върху двойно дърво

Основните операции върху двойно дърво са:

- достъп до връх;
- включване на връх;

- изключване на връх;
- обхождане.

До корена се осъществява пряк достъп, а до останалите върхове – непряк чрез лявата и дясната връзка. Включването и изключването могат да се извършват на произволно място, като резултатът трябва да бъде двойно дърво от същия тип.

Дефиниция 18.15 (Обхождане)**.** Обхождането е метод, чрез който се осъществява достъп до всеки връх на дървото точно един път.

При рекурсивното обхождане се изпълняват три действия в определен ред:

1. обхождане на корена;
2. обхождане на лявото поддърво;
3. обхождане на дясното поддърво.

В зависимост от реда на тези действия се получават различни обходи. Когато редът е корен, ляво поддърво, дясно поддърво, се получава предварително обхождане. Когато първо се обхожда лявото поддърво, след това коренът, а накрая дясното поддърво, се получава смесено обхождане. При обхождане ляво поддърво, дясно поддърво, корен се получава последващо обхождане.

```
void preorder(Node* tree) {
  if (tree == nullptr) {
    return;
  }

  visit(tree->data);
  preorder(tree->left);
  preorder(tree->right);
}

void inorder(Node* tree) {
  if (tree == nullptr) {
    return;
  }

  inorder(tree->left);
  visit(tree->data);
  inorder(tree->right);
}

void postorder(Node* tree) {
  if (tree == nullptr) {
    return;
```

```
  }

  postorder(tree->left);
  postorder(tree->right);
  visit(tree->data);
}
```

18.6.3 Представяния на двойно дърво

Използват се три начина за представяне на двойно дърво:

1. свързано представяне;
2. верижно представяне чрез масиви;
3. представяне чрез списък на бащите.

При свързаното представяне всеки възел съдържа стойност и два указателя. Този начин е естествен за рекурсивните операции.

При верижното представяне се използват три масива $a[N]$, $b[N]$ и $c[N]$, където N е броят на върховете. Върховете са номерирани от 0 до $N-1$. В $a[i]$ се съхранява стойността на i-тия връх, в $b[i]$ – индексът на левия наследник, а в $c[i]$ – индексът на десния наследник. Ако съответният наследник липсва, се записва -1. Отделно се пази индексът на корена.

При представяне чрез списък на бащите се използва един масив $p[N]$. В $p[i]$ се съхранява индексът на единствения баща на върха i. За корена се записва -1.

† **Допълнение 18.16 (Бележка).** За общо двойно дърво източникът посочва сложност $O(n)$ за търсене, а на друго място посочва $O(\log n)$ за добавяне. Тези оценки зависят от конкретната структура и от начина на търсене. При обикновено двойно дърво без свойство за подреждане търсенето в най-общия случай е линейно. Сложността $O(\log n)$ е характерна за операции в дърво с подходяща височина, например балансирано двойно подредено дърво.

18.6.4 Клас за двойно кореново дърво

Класът притежава корена и отговаря за освобождаването на цялото дърво. Инвариант: `root` е `nullptr` (празно дърво) или сочи възел, чиито поддървета са непресичащи се — никой възел не е достижим по два различни пътя, тоест структурата е дърво, а не общ граф.

```
#include <iostream>

class BinaryTree {
  struct Node {
    int data;
    Node* left;
```

```
    Node* right;
  };

  Node* root;

  static void destroy(Node* v) {
    if (v == nullptr) return;
    destroy(v->left);
    destroy(v->right);
    delete v;
  }

  static Node* clone(const Node* v) {
    if (v == nullptr) return nullptr;
    return new Node{v->data, clone(v->left), clone(v->right)};
  }

  static int heightOf(const Node* v) {
    if (v == nullptr) return 0;
    int l = heightOf(v->left);
    int r = heightOf(v->right);
    return 1 + (l > r ? l : r);
  }

  static int countOf(const Node* v) {
    if (v == nullptr) return 0;
    return 1 + countOf(v->left) + countOf(v->right);
  }

  static void inorder(const Node* v) {
    if (v == nullptr) return;
    inorder(v->left);
    std::cout << v->data << ' ';
    inorder(v->right);
  }

public:
  BinaryTree() : root(nullptr) {}

  // Рекурсивното определение, изразено като конструктор.
  BinaryTree(int value,
             const BinaryTree& left, const BinaryTree& right)
      : root(new Node{value,
                      clone(left.root), clone(right.root)}) {}

  BinaryTree(const BinaryTree& other)
```

```
      : root(clone(other.root)) {}

  BinaryTree& operator=(const BinaryTree& other) {
    if (this != &other) {
      Node* copied = clone(other.root);   // първо копираме,
      destroy(root);                      // после освобождаваме
      root = copied;
    }
    return *this;
  }

  ~BinaryTree() { destroy(root); }

  bool empty() const { return root == nullptr; }
  int size() const { return countOf(root); }
  int height() const { return heightOf(root); }
  void printInorder() const { inorder(root); }
};
```

Конструкторът с три аргумента е точно рекурсивната дефиниция: стойност, ляво поддърво, дясно поддърво. Затова

```
BinaryTree leaf4(4, BinaryTree(), BinaryTree());
BinaryTree leaf12(12, BinaryTree(), BinaryTree());
BinaryTree t(10, leaf4, leaf12);
```

строи дървото с корен 10 и наследници 4 и 12, без нито един `new` в кода на потребителя.

В `operator=` копирането предхожда освобождаването: ако `clone` хвърли изключение поради недостиг на памет, обектът остава непроменен вместо да остане с освободен корен.

18.7 Двойно подредено дърво

Дефиниция 18.17 (Двойно подредено дърво)**.** Двойно подредено дърво от тип T е двойно дърво, за което върховете на лявото поддърво са по-малки от корена, върховете на дясното поддърво са по-големи от корена, а лявото и дясното поддърво също са двойно подредени дървета от тип T.

Празното двойно дърво се приема за двойно подредено дърво. В източника е посочено, че не се включват елементи със същата стойност.

Например за дърво с корен 10, ляво поддърво с корен 4 и дясно поддърво с корен 12 са валидни само поддървета, които спазват същото правило за подреждане.

18.7.1 Търсене

Търсенето на елемент x започва от корена:

1. ако дървото е празно, елементът не е намерен;
2. ако x съвпада със стойността на корена, елементът е намерен;
3. ако x е по-малък от корена, търсенето продължава в лявото поддърво;
4. ако x е по-голям от корена, търсенето продължава в дясното поддърво.

```
Node* search(Node* tree, int value) {
  if (tree == nullptr || tree->data == value) {
    return tree;
  }

  if (value < tree->data) {
    return search(tree->left, value);
  }

  return search(tree->right, value);
}
```

Източникът посочва сложност $O(\log n)$ за търсене. Тази оценка се получава при дърво с логаритмична височина. При небалансирано дърво височината може да бъде линейна и тогава търсенето е $O(n)$.

18.7.2 Включване

Включването също започва от корена. Ако дървото е празно, се създава нов възел с празни ляво и дясно поддърво. Ако стойността е по-малка от корена, включването продължава в лявото поддърво. Ако е по-голяма, продължава в дясното. При същата стойност включването не се извършва.

```
Node* insert(Node* tree, int value) {
  if (tree == nullptr) {
    Node* item = new Node;
    item->data = value;
    item->left = nullptr;
    item->right = nullptr;
    return item;
  }

  if (value < tree->data) {
    tree->left = insert(tree->left, value);
  } else if (value > tree->data) {
```

```
    tree->right = insert(tree->right, value);
  }

  return tree;
}
```

Посочената в източника сложност е $O(\log n)$ при дърво с логаритмична височина. При небалансирано дърво включването може да бъде $O(n)$.

18.7.3 Изключване

При изключване първо се търси елементът по правилото за подреденото дърво. Ако елементът не съществува, изключване не се извършва.

Разграничават се следните случаи:

1. Изтриваният връх е лист. Той може да бъде премахнат, като съответният указател на неговия баща стане `nullptr`.
2. Върхът има само ляво поддърво. На негово място се поставя лявото поддърво.
3. Върхът има само дясно поддърво. На негово място се поставя дясното поддърво.
4. Върхът има и ляво, и дясно поддърво. Избира се най-левият връх на дясното поддърво. Неговата стойност се поставя на мястото на изтриваната стойност, след което избраният връх се изключва от дясното поддърво.

При последния случай избраният връх е най-малкият елемент в дясното поддърво и следователно запазва свойството за подреждане.

```
Node* removeMin(Node* tree, int& value) {
  if (tree->left == nullptr) {
    value = tree->data;
    Node* right = tree->right;
    delete tree;
    return right;
  }

  tree->left = removeMin(tree->left, value);
  return tree;
}

Node* remove(Node* tree, int value) {
  if (tree == nullptr) {
    return nullptr;
  }

  if (value < tree->data) {
```

```
    tree->left = remove(tree->left, value);
    return tree;
  }

  if (value > tree->data) {
    tree->right = remove(tree->right, value);
    return tree;
  }

  if (tree->left == nullptr) {
    Node* right = tree->right;
    delete tree;
    return right;
  }

  if (tree->right == nullptr) {
    Node* left = tree->left;
    delete tree;
    return left;
  }

  int successor;
  tree->right = removeMin(tree->right, successor);
  tree->data = successor;
  return tree;
}
```

Както при търсенето и включването, източникът посочва сложност $O(\log n)$ за изключване. Тя се отнася до дърво с логаритмична височина. В общия случай сложността се определя от височината на дървото.

18.7.4 Обхождане на двойно подредено дърво

При смесено обхождане – ляво поддърво, корен, дясно поддърво – елементите на двойното подредено дърво се получават във възходящ ред.

При обхождане корен, дясно поддърво, ляво поддърво елементите се получават в низходящ ред. Това следва от правилото, че всички елементи в лявото поддърво са по-малки от корена, а всички в дясното са по-големи.

18.7.5 Клас за двойно подредено дърво

Инвариант: за всеки възел v всички стойности в лявото поддърво са строго по-малки от `v->data`, а всички в дясното — строго по-големи; равни стойности не се съхраняват. Смесеното обхождане на класа затова връща нарастваща редица.

```
class SearchTree {
```

```
struct Node {
  int data;
  Node* left;
  Node* right;
};

Node* root;

static void destroy(Node* v) {
  if (v == nullptr) return;
  destroy(v->left);
  destroy(v->right);
  delete v;
}

static Node* clone(const Node* v) {
  if (v == nullptr) return nullptr;
  return new Node{v->data, clone(v->left), clone(v->right)};
}

static Node* insert(Node* v, int x) {
  if (v == nullptr) return new Node{x, nullptr, nullptr};
  if (x < v->data) v->left = insert(v->left, x);
  else if (x > v->data) v->right = insert(v->right, x);
  return v;              // x == v->data: нищо не се променя
}

static Node* minNode(Node* v) {
  while (v->left != nullptr) v = v->left;
  return v;
}

static Node* remove(Node* v, int x) {
  if (v == nullptr) return nullptr;
  if (x < v->data) { v->left = remove(v->left, x); return v; }
  if (x > v->data) { v->right = remove(v->right, x); return v; }

  if (v->left == nullptr) {      // нула или един наследник
    Node* r = v->right;
    delete v;
    return r;
  }
  if (v->right == nullptr) {
    Node* l = v->left;
    delete v;
    return l;
```

```
    }
    // два наследника: най-малкият вдясно
    Node* s = minNode(v->right);
    v->data = s->data;
    v->right = remove(v->right, s->data);
    return v;
  }

public:
  SearchTree() : root(nullptr) {}
  SearchTree(const SearchTree& other)
      : root(clone(other.root)) {}

  SearchTree& operator=(const SearchTree& other) {
    if (this != &other) {
      Node* copied = clone(other.root);
      destroy(root);
      root = copied;
    }
    return *this;
  }

  ~SearchTree() { destroy(root); }

  bool contains(int x) const {
    const Node* v = root;
    while (v != nullptr) {
      if (x == v->data) return true;
      v = (x < v->data) ? v->left : v->right;
    }
    return false;
  }

  void insert(int x) { root = insert(root, x); }
  void remove(int x) { root = remove(root, x); }
};
```

Изключването на връх с два наследника заменя стойността му с най-малката в дясното поддърво и после премахва *нея* — този възел по построение няма ляв наследник, затова попада в лесния случай. Заменящата стойност е по-голяма от всичко вляво и по-малка от всичко останало вдясно, следователно инвариантът се запазва.

И трите операции — `contains`, `insert` и `remove` — слизат по един-единствен път от корена, затова струват $O(h)$, където h е височината. Това е $O(\log n)$ само при височина от порядъка на $\log n$; при включване на вече наредени стойности дървото се изражда в списък и трите операции стават $O(n)$.

18.8 Сравнение на структурите

Стекът и опашката са линейни структури, но се различават по реда на достъп. При стека се използва LIFO, а при опашката – FIFO. Списъкът е по-обща линейна структура, защото допуска включване и изключване от произволно място.

Дървото е разклонена структура. При двойното дърво всеки връх има най-много ляв и десен наследник. Физическото представяне чрез указатели следва естествено рекурсивното определение на дървото, докато масивните представяния използват индекси вместо адреси.

18.8.1 Сложност на операциите

Таблицата обобщава цената на основните операции. Навсякъде n е броят на съхранените елементи, а h — височината на дървото. Оценките са в *най-лошия* случай, освен изрично отбелязаните като амортизирани. Приема се, че сравнението и копирането на един елемент струват $O(1)$; при елементи с по-скъпо копиране цената на преоразмеряването и на `clone` се умножава съответно.

Структура	Реализация	Включване	Изключване	Намиране
Стек	статична (масив)	$O(1)$	$O(1)$	$O(n)$
Стек	динамична (удвояване)	$O(1)^1$	$O(1)$	$O(n)$
Стек	свързана	$O(1)$	$O(1)$	$O(n)$
Опашка	статична (циклична)	$O(1)$	$O(1)$	$O(n)$
Опашка	динамична	$O(1)^1$	$O(1)$	$O(n)$
Опашка	свързана	$O(1)$	$O(1)$	$O(n)$
Списък	едносвързан	$O(1)^2$	$O(n)^3$	$O(n)$
Списък	двусвързан	$O(1)^2$	$O(1)^4$	$O(n)$
Списък	последователен	$O(n)$	$O(n)$	$O(n)^5$
Двойно дърво	свързана	$O(1)^6$	$O(1)^6$	$O(n)$
ДПД	свързана	$O(h)$	$O(h)$	$O(h)$

[1] амортизирано
[2] в края; при двусвързан и в началото
[3] по стойност — търси се предшественикът
[4] при вече известен възел
[5] $O(\log n)$ при сортиран масив
[6] при вече известен баща

Единственият ред, който заблуждава, ако се чете без уговорката, е последният: $O(h)$ е $\Theta(\log n)$ при балансирано дърво и $\Theta(n)$ при изродено. Балансирането не се разглежда в този въпрос, затова оценките за ДПД се дават чрез h, а не чрез $\log n$.

18.9 Изпитен фокус

- Да се дефинира структура от данни и да се различат логическо описание и физическо представяне.
- Да се обяснят стекът и правилото LIFO.

- Да се опишат операциите включване, изключване, достъп до върха и търсене в стек.
- Да се различат трите реализации, назовани от конспекта: статична (масив с фиксиран капацитет), динамична (масив с удвояване, $O(1)$ амортизирано включване) и свързана (възли с указатели).
- Да се сравнят последователното и свързаното представяне на стек.
- Да се обяснят опашката и правилото FIFO.
- Да се опишат началото и краят на опашката и цикличното последователно представяне.
- Да се обясни защо при свързана опашка са нужни указатели към началото и края.
- Да се дефинира списък и да се сравнят едносвързаното, двусвързаното и последователното представяне.
- Да се покажат връзките в едносвързан и двусвързан списък.
- Да се дефинира дърво и двойно дърво рекурсивно.
- Да се обяснят корен, наследник, листо, вътрешен връх, ляво и дясно поддърво.
- Да се опишат свързаното представяне, верижното представяне чрез масиви и списъкът на бащите.
- Да се обясни обхождането и редът на действията при предварително, смесено и последващо обхождане.
- Да се дефинира двойно подредено дърво.
- Да се възпроизведат алгоритмите за търсене, включване и изключване в двойно подредено дърво.
- Да се обясни изключването на връх с нула, един или два наследника.
- Да се посочи, че оценките $O(\log n)$ за операциите в двойно подредено дърво предполагат логаритмична височина, докато в общия случай височината може да доведе до линейна сложност.
- **Да се напише клас** за всяка от петте структури — списък, стек, опашка, двойно кореново дърво и двойно подредено дърво — с инвариант, конструктор, основни операции и деструктор. Конспектът иска „дефиниране на клас“, а не само свободни функции над отворена структура.
- Да се обясни защо копиращ конструктор, операция за присвояване и деструктор вървят заедно при клас, който притежава динамична памет, и какво се случва, ако някой от тях липсва.
- Да се възпроизведе таблицата със сложностите (§18.8.1) заедно с допусканията: най-лош случай, $O(1)$ на елемент за сравнение и копиране, и $O(h)$ вместо $O(\log n)$ за ДПД.

Въпрос 19

Функционално програмиране. Обща характеристика. Модели на оценяване. Функции от по-висок ред.

По конспекта: какво трябва да мога

Официално заглавие: „Функционално програмиране. Обща характеристика на функционалния стил на програмиране. *Дефиниране и използване на функции.* Модели на оценяване. Функции от по-висок ред.“

> ★ **Бележка на конспекта:** По този въпрос е възможно да бъдат дадени задачи за *практическата* част на изпита.

Обща характеристика: характерни особености на функционалния стил; основни компоненти на функционалните програми; примитивни изрази; средства за комбиниране и абстракция — §19.1, стр. 273.
Оценяване: оценяване на израз; дефиниране на функция и оценяване на приложение на функция — §19.2, стр. 277.
Модели на оценяване: апликативно (стриктно, *call-by-value*) и нормално (лениво, *call-by-name*) — §19.3, стр. 278.
Функции от по-висок ред: функциите като параметри и като резултати; анонимни (ламбда) функции — §19.4, стр. 280.
Литература: [14], [22], [29], [53].

19.1 Функционално програмиране

19.1.1 Обща характеристика на функционалния стил

Дефиниция 19.1 (Функционален стил)**.** Функционалното програмиране е стил на програмиране, в който основните логически единици са функциите. Функцията е програмна единица, която прилага определено преобразуване върху аргументи и връща резултат. Програмата се изгражда чрез дефиниране, използване и комбиниране на функции.

Основното действие във функционалната програма е обръщението към функция. Основният начин за разделяне на програмата на части е дефинирането на име и свързването му с израз, който изчислява стойността на функцията. Основният начин за комбиниране е суперпозицията на функции: резултатът от една функция се подава като аргумент на друга.

Функциите във функционалния стил могат:

- да имат параметри;
- да се прилагат върху аргументи;
- да приемат други функции като аргументи;
- да връщат функции като резултат;
- да бъдат дефинирани рекурсивно, тоест чрез обръщения към самите себе си.

Функционалният стил е представител на декларативния стил на програмиране. При декларативния подход програмата описва какво трябва да бъде полученият резултат, вместо подробно да задава последователност от промени в паметта. Във функционалната програма обикновено липсват традиционни процедурни елементи като присвояване, цикли, заделяне и промяна на клетки в паметта, както и управление на изпълнението чрез прескачане.

Програмите във функционалния стил могат да се разглеждат като списък от дефиниции на функции или като списък от равенства, които описват зависимостите между стойностите. Изпълнението се свежда до оценяване на изрази и прилагане на функции.

Дефиниция 19.2. *Функционално програмиране* е стил на програмиране, при който програмите се изграждат основно чрез дефиниране, прилагане и композиция на функции, а изчисляването се представя като оценяване на изрази.

Съществен компонент на функционалната програма е липсата на странични ефекти в чистия функционален език. Когато изразът е чист и се оценява в един и същ контекст, той дава една и съща стойност. Това позволява програмата да се разглежда като описание на зависимости между стойности, а не като поредица от изменения на общо състояние.

19.1.2 Изрази и стойности в Scheme

За илюстрация на функционалните понятия ще бъде използван синтаксисът на Scheme. Изразът е атом или комбинация от изрази.

Атомарните изрази не се декомпозират повече. Типични атомарни изрази в Scheme са:

- булевите константи `#t` и `#f`;
- числовите константи, например `15`, `2/3` и `-1.23`;
- знаковите константи, например `#\a` и `#\newline`;
- низовите константи, например `"Scheme"` и `"hi"`;
- идентификаторите, тоест имената, които могат да бъдат свързани със стойности;
- вградените функции, например `+`, `-`, `*`, `/` и `sqrt`.

Комбинация в Scheme е израз, конструиран като списък от изрази, заградени в скоби:

```
(<expression0> <expression1> ... <expressionn>)
```

Дефиниция 19.3 (Оператор и операнди)**.** Първият израз, означен като `<expression0>`, се нарича оператор. Очаква се неговата оценка да бъде функция. Останалите изрази са операнди. Стойността на комбинацията се получава, като функцията, зададена чрез оператора, се приложи върху стойностите на операндите.

Например:

```
(+ 2 3)
```

е комбинация, чийто оператор е `+`, а операндите са `2` и `3`. Оценката на комбинацията е резултатът от прилагането на функцията за събиране върху двете числа.

Правилата за оценяване на някои основни изрази са следните:

- оценката на число е самото число;
- оценката на низ е самият низ;
- оценката на вграден оператор е реализацията на този оператор;
- оценката на име е стойността, свързана с името в текущата среда.

Дефиниция 19.4. *Среда* е контекстът, който съдържа свързвания между имена и стойности. При оценяване на име се търси стойността, свързана с това име в текущата среда.

19.1.3 Дефиниране и използване на функции

Дефиниция 19.5 (Дефиниране на функция)**.** Дефинирането на функция представлява свързване на име с израз, който описва изчисляването на резултата. Дефиницията е средство за абстракция, защото позволява сложното изчисление да бъде скрито зад име и използвано многократно.

Дефиниране на име чрез израз в Scheme:

```
(define <name> <expression>)
```

Например:

```
(define size 2)
```

свързва името `size` със стойността 2.

Функция с формални параметри може да се дефинира чрез:

```
(define (<name> <formal-parameters>) <body>)
```

Пример:

```
(define (square x)
  (* x x))
```

Тук `square` е името на функцията, `x` е формален параметър, а `(* x x)` е тялото. При обръщението

```
(square 5)
```

функцията се прилага върху аргумента 5 и резултатът е 25.

Дефиниция 19.6. *Формален параметър* е име, използвано в дефиницията на функцията, което представя стойността на съответния аргумент при конкретно приложение.

Дефиниция 19.7. *Аргумент* е изразът или стойността, подадени при конкретно обръщение към функция.

Формалните параметри и действителните аргументи се съпоставят по позиция. При прилагането на функцията формалните параметри получават стойностите на съответните аргументи, след което се оценява тялото на функцията.

Рекурсията е основен метод за дефиниране на функции. При рекурсивна дефиниция функцията се обръща към самата себе си. Източниците посочват рекурсията като характерен начин за дефиниране във функционалното програмиране, без да налагат конкретна рекурсивна задача или пример.

19.1.4 Условни форми и локални дефиниции

Условната форма `if` има синтаксис:

```
(if <condition> <expression1> <expression2>)
```

Първо се оценява условието. Ако неговата стойност е `#t`, се оценява и връща `<expression1>`. В противен случай се оценява и връща `<expression2>`.

Тъй като се оценява само избраното разклонение, `if` е специална форма, а не обикновена функция. Специалните форми са конструкции, които са изключения от общото правило за оценяване на комбинациите.

За многозначен избор се използва `cond`:

```
(cond
  (<condition1> <expression1>)
  ...
  (<conditionn> <expressionn>)
  (else <expressionm>))
```

Условията се оценяват последователно. При първото условие със стойност `#t` се оценява съответният израз и неговата стойност става стойност на цялата форма. Ако нито едно условие не е изпълнено, се избира клонът `else`.

Локални имена могат да се създават чрез специалната форма `let`:

```
(let ((<symbol1> <expression1>)
      ...
      (<symboln> <expressionn>))
  <body>)
```

Оценката на `let` в среда E се описва по следния начин:

1. създава се нова среда E_1, която е разширение на E;
2. всеки израз $\texttt{<expression}_i\texttt{>}$ се оценява в E;
3. получената стойност се свързва със съответното име $\texttt{<symbol}_i\texttt{>}$ в E_1;
4. тялото се оценява в E_1 и неговата стойност се връща като стойност на `let`.

Символично:

$$E_1 = E \cup \{\texttt{<symbol}_i\texttt{>} \mapsto \text{eval}(\texttt{<expression}_i\texttt{>}, E) \mid 1 \leq i \leq n\},$$

а резултатът е:

$$\text{eval}(\texttt{<body>}, E_1).$$

В Haskell локални дефиниции могат да се задават чрез `where`:

```
<function-definition>
  where
    <definition1>
    ...
    <definitionn>
```

Областта на действие на тези дефиниции е ограничена до дефиницията на функцията. Дефинициите в `where` могат да бъдат взаимно рекурсивни.

19.2 Оценяване на изрази

19.2.1 Общо правило за оценяване на комбинация

Общото правило за оценяване на обикновена комбинация се състои от следните стъпки:

1. оценяват се подизразите на комбинацията;
2. оценката на най-левия подизраз се разглежда като функция;
3. тази функция се прилага върху оценките на останалите подизрази.

Например:

```
(* (+ 2 (* 4 6)) (+ 3 5 7))
```

се оценява чрез последователността:

```
(* (+ 2 24) 15)
(* 26 15)
390
```

Тук общото правило се прилага към вложените комбинации `(* 4 6)`, `(+ 2 (* 4 6))`, `(+ 3 5 7)` и външната комбинация.

★ **Дефиниция 19.8.** При обикновена комбинация първо се оценяват операторът и операндите, а след това функцията, получена като стойност на оператора, се прилага върху стойностите на операндите.

Конструкциите `define`, `if` и `cond` не следват непосредствено това правило. Те са специални форми и имат собствени правила за оценяване.

19.2.2 Оценяване на приложение на функция

При комбинация, чийто оператор е функция, се оценяват елементите на комбинацията и се прилага процедурата, която е стойност на оператора, върху стойностите на операндите.

При дефиницията:

```
(define (square x)
  (* x x))
```

специалната форма `define` създава свързване между името `square` и зададената функция. Това свързване се добавя в текущата среда. Функцията може по-късно да бъде приложена чрез:

```
(square 4)
```

Моделът на заместването представя прилагането на функцията чрез заместване на формалните параметри със съответните аргументи. За `(square 4)` тялото

```
(* x x)
```

се разглежда след заместването като

```
(* 4 4)
```

и се оценява до 16. Оценката на последния израз от тялото е оценката на обръщението към функцията.

Този модел е концептуален: той обяснява как аргументите се свързват с формалните параметри. Реалната реализация може да използва друга вътрешна техника, но поведението се описва чрез същите зависимости.

19.3 Модели на оценяване

При модела на заместването са възможни два основни подхода за оценяване на аргументите: апликативен и нормален.

19.3.1 Апликативно оценяване

Дефиниция 19.9 (Апликативно оценяване)**.** Апликативното оценяване се нарича още строго оценяване или оценяване по стойност (*call-by-value*). При него първо се оценяват операндите, след което функцията се прилага върху получените стойности.

Схематично:

оценяване на операндите $\longrightarrow$ получаване на стойности $\longrightarrow$ прилагане на функцията.

Ако имаме приложение от вида

```
(f e1 e2)
```

апликативният подход първо оценява `e1` и `e2`, а след това прилага `f` върху получените резултати.

Предимството на този подход, посочено в източниците, е, че преди прилагането на функцията всеки аргумент е сведен до една стойност. Това е пестеливо откъм памет, тъй като не се налага да се съхраняват неоценени изрази. Аргументът е, че се извършва оценяване дори когато стойността му в крайна сметка не е необходима на тялото на функцията.

19.3.2 Нормално оценяване

Дефиниция 19.10 (Нормално оценяване)**.** Нормалното оценяване се нарича още лениво оценяване или оценяване по име (*call-by-name*). При него аргументите не се оценяват предварително. Вместо това имената на аргументите се заместват в тялото на функцията и изразите се оценяват, когато това стане необходимо.

Схематично:

$$\text{заместване в тялото} \longrightarrow \text{примитивни операции}$$
$$\longrightarrow \text{оценяване на необходимите изрази.}$$

Следователно нормалният подход първо разгъва функцията и чак след това оценява аргументите, които участват в получения израз.

Нормалното оценяване може да избегне оценяване на аргумент, който не се използва. То обаче може да доведе до многократно оценяване на един и същ израз, ако той се появява повече от веднъж след заместването. Така се пести време за предварително оценяване само когато изразът не е необходим, но може да се изразходва повече време при многократно пресмятане. Същевременно е необходимо да се съхраняват или повторно да се разгръщат неоценени изрази, поради което подходът е по-непестелив откъм памет.

★ **Дефиниция 19.11.** При апликативното оценяване аргументите се оценяват преди прилагането на функцията. При нормалното оценяване аргументите се заместват и се оценяват само когато са необходими.

Забележка 19.12. Една и съща програма може да се държи различно при апликативен и нормален подход, особено когато функцията не използва някой аргумент или когато оценяването на аргумент би довело до проблем. Източниците посочват също, че в стандартните езици има лениво поведение в отделни конструкции, например при конюнкция, когато след лъжлив операнд останалите операнди не се пресмятат.

† **Допълнение 19.13 (Уточнение за терминологията).** В учебната литература изразът „лениво оценяване“ често се използва по-широко от „call-by-name“ и може да включва запомняне на вече изчислена стойност. Разглежданият тук материал представя нормалния подход чрез заместване и акцентира върху възможното многократно оценяване на един и същ израз.

19.4 Функции от по-висок ред

19.4.1 Определение

Дефиниция 19.14. *Функция от по-висок ред* е функция, която приема функция като параметър или връща функция като резултат.

Функциите от по-висок ред са естествено следствие от идеята, че функциите са стойности, с които програмата може да работи. Те могат да се подават на други функции, да се връщат като резултат и да се използват за абстрахиране на повтарящи се схеми на изчисление.

Функция, която приема друга функция като параметър, може да прилага подадената функция върху дадена стойност. Например предикатът за неподвижна точка може да бъде записан в Scheme като:

```
(define (fixed-point f x)
  (= (f x) x))
```

Тук `f` е параметър, чиято стойност трябва да бъде функция, а `x` е стойност, върху която `f` се прилага.

Практически примери за функции от по-висок ред са `map`, `filter`, `foldl`, `foldr` и `apply`. Функцията `map` прилага дадена функция върху всеки елемент на списък и връща нов списък:

```
(map (lambda (x) (* x x)) '(1 2 3))
```

Резултатът е:

```
'(1 4 9)
```

Функцията `filter` приема предикат и списък и връща елементите, за които предикатът е истина:

```
(filter even? '(1 2 3 4 5))
```

Резултатът е:

```
'(2 4)
```

Функциите `foldr` и `foldl` свиват списък чрез подадена функция и начална стойност. За пример:

```
(foldr - 0 '(1 2 3))
```

се интерпретира като:

$$1 - (2 - (3 - 0)).$$

При `foldl` обработката се извършва отляво надясно чрез акумулиране на междинния резултат:

```
(foldl - 0 '(1 2 3))
```

В Racket функцията за `foldl` получава първо текущия елемент и после акумулатора, затова изразът се интерпретира като:

$$3 - (2 - (1 - 0)) = 2.$$

Този ред на аргументите е различен от често използваната Haskell конвенция, при която акумулаторът е първи.

Функцията `apply` приема функция и списък от аргументи, като използва елементите на списъка като отделни аргументи:

```
(apply + '(1 2 3))
```

е еквивалентно на:

```
(+ 1 2 3)
```

и дава резултат 6.

19.4.2 Къринг

Дефиниция 19.15 (Къринг)**.** В Haskell функция с няколко аргумента може да се разглежда като функция на един аргумент, която връща функция, очакваща следващия аргумент. Това представяне се нарича къринг.

Ако функцията има тип:

$$t_1 \to t_2 \to t_3,$$

тя приема аргумент от тип t_1 и връща функция от тип:

$$t_2 \to t_3.$$

Следователно частичното прилагане на функцията създава нова функция. Например:

```
div50 x = div 50 x
```

може да се разглежда като дефиниране на:

```
div50 = div 50
```

и след това:

```
div50 4
```

дава 12.

Дефиниция 19.16. *Къринг* е представяне на функция с няколко аргумента като последователност от функции, всяка от които приема един аргумент и връща функция или краен резултат.

19.4.3 Анонимни функции

Анонимна, или ламбда, функция е функция без зададено име. Тя може да бъде конструирана на мястото, където е необходима, особено като аргумент на функция от по-висок ред.

Синтаксисът в Scheme е:

```
(lambda (<parameters>) <body>)
```

Пример:

```
(lambda (x) (+ x 3))
```

оценява до функционален обект с параметър `x` и тяло `(+ x 3)`. Този обект може да бъде приложен непосредствено:

```
((lambda (x) (+ x 3)) 5)
```

Резултатът е 8.

Анонимните функции позволяват на програмиста да задава поведението на функция от по-висок ред без предварително именуване на помощна функция. В предишния пример ламбда-функцията е подадена директно на `map`.

19.4.4 Функции, които връщат функции

Функция от по-висок ред може да връща нова функция като резултат. Пример е функцията `twice`, която приема функция `f` и връща функция, прилагаща `f` два пъти:

```
(define (twice f)
  (lambda (x) (f (f x))))
```

Обръщението:

```
(twice square)
```

връща функция. Прилагането на получения резултат върху `3` дава:

```
((twice square) 3)
```

Това е еквивалентно на прилагане на `square` два пъти:

$$(3^2)^2 = 81.$$

Друг пример е функция, която създава функция за прибавяне на фиксирано число:

```
(define (n+ n)
  (lambda (i) (+ i n)))
```

Така може да се дефинира:

```
(define 1+ (n+ 1))
```

Полученото `1+` е функция, която прибавя фиксирана стойност към своя аргумент.

† **Допълнение 19.17 (Бележка към примерната стойност).** При дефиницията `(define 1+ (n+ 1))` обръщението `(1+ 4)` според самото тяло `(+ i n)` дава `5`. Среща се и изписване на друга резултатна стойност в учебния материал; тя не следва от показаната дефиниция, затова тук е запазена зависимостта, зададена от кода.

19.5 Изпитен фокус

- Да се дефинира функционалното програмиране и да се опишат функциите като основни програмни единици.
- Да се обясни декларативният характер на функционалния стил, композицията на функции и ролята на рекурсията.
- Да се различат атом, комбинация, оператор и операнд в Scheme.
- Да се възпроизведат синтаксисът и смисълът на `define`, `if`, `cond`, `let` и локалните дефиниции чрез `where`.
- Да се изложи общото правило за оценяване на комбинация и да се проследи примерът с вложено събиране и умножение.
- Да се обясни моделът на заместването при прилагане на функция.
- Да се сравнят апликативното оценяване *call-by-value* и нормалното оценяване *call-by-name* по ред на оценяване, време и памет.
- Да се дефинира функция от по-висок ред и да се дадат примери с функции като параметри и резултати.

- Да се обяснят кърингът, частичното прилагане и анонимните ламбда-функции.
- Да се проследи примерът `twice`, който връща функция, прилагаща подадена функция два пъти.

Въпрос 20

Функционално програмиране. Списъци. Потоци и отложено оценяване.

По конспекта: какво трябва да мога

★ **Бележка на конспекта:** По този въпрос е възможно да бъдат дадени задачи за *практическата* част на изпита.

Списъци: представяне и основни операции в Scheme — §20.2, стр. 286.
Списъци: представяне и основни операции в Haskell — §20.3, стр. 290.
Функции от по-висок ред: за работа със списъци — §20.4, стр. 292.
Потоци: безкрайни потоци; отложено оценяване; основни операции и функции от по-висок ред — §20.5, стр. 295.
Потоци: безкрайни списъци и отложено оценяване в Haskell — §20.6, стр. 298.
Литература: [14], [22], [29], [53].

20.1 Функционално програмиране

Дефиниция 20.1 (Функционален стил)**.** Функционалното програмиране е стил на програмиране, при който изчислението се описва чрез функции и тяхното комбиниране. Особено важна роля имат функциите от по-висок ред — функции, които приемат други функции като аргументи или връщат функции като резултат. В разглежданите езици Scheme и Haskell основен обект за обработка са списъците. Чрез рекурсия и функции от по-висок ред върху списъците се реализират трансформиране, филтриране и акумулиране.

В Scheme списъците са изградени върху по-общото понятие за S-израз. В Haskell списъците са параметризирани по тип и могат да бъдат крайни или безкрайни. Отложеното оценяване позволява елементите на един списък или поток да се изчисляват само когато са необходими.

20.2 S-изрази и списъци в Scheme

20.2.1 S-изрази и наредени двойки

Дефиниция 20.2. S-израз наричаме обект, който е атом или наредена двойка от два S-израза.

Атоми са булеви стойности, числа, знаци, символи, низове и функции. Наредената двойка се записва формално като

$$(S_1 \,.\, S_2),$$

където S_1 и S_2 са S-изрази. Понеже и двата компонента могат отново да бъдат S-изрази, чрез наредени двойки могат да се изграждат произволно сложни структури.

В Scheme наредена двойка се конструира чрез процедурата `cons`:

```
(cons <израз1> <израз2>)
```

Първо се оценяват двата израза, а след това се създава наредена двойка от получените стойности. Достъпът до компонентите се осъществява чрез процедурите `car` и `cdr`:

```
(car <израз>)
(cdr <израз>)
```

`car` връща първия компонент на наредената двойка, а `cdr` — втория компонент.

В реализацията на Scheme обектите се представят чрез указатели към клетки. Клетката, съответстваща на примитивен обект, съдържа представянето на този обект. Клетката, съответстваща на точкова двойка, съдържа два указателя: към представянията на нейния първи и втори компонент.

Например:

```
(cons 1 2)
```

създава двойката $(1 \,.\, 2)$. При списъците вторият компонент на всяка двойка е или следваща двойка, или празният списък.

20.2.2 Определение и представяне на списък

Дефиниция 20.3. В Scheme списък е празният списък или наредена двойка, чийто втори компонент е списък.

Празният списък се записва като

```
'()
```

Дефиниция 20.4 (Глава и опашка на списък). Ако t е списък, то двойката $(h\,.\,t)$ също е списък. Елементът h се нарича глава, а списъкът t — опашка.

Крайният списък

$$(a_1\ a_2\ \ldots\ a_n)$$

има вътрешно представяне

$$(a_1\ .\ (a_2\ .\ (\ldots .\ (a_n\ .\ ()))))).$$

Следователно списъкът е верига от наредени двойки, завършваща с празен списък. В Scheme обичайният печатен запис скрива точките и скобите на вътрешното представяне.

Списък може да се конструира чрез `list`:

```
(list <a1> <a2> ... <an>)
```

Този израз е еквивалентен на последователно влагане на `cons`:

```
(cons <a1>
      (cons <a2>
            ...
            (cons <an> '())))
```

Например:

```
(list 'a 'b 'c)
```

и

```
(cons 'a (cons 'b (cons 'c '())))
```

създават списъка `(a b c)`.

Ако към вече съществуващ списък се добави елемент чрез `cons`, елементът се поставя в началото:

```
(cons 'a '(b c d))
```

Резултатът е

```
(a b c d)
```

Трябва да се различават правилният списък и произволна наредена двойка. Например `(1 . 2)` е наредена двойка, но не е списък, защото вторият й компонент не е списък. За проверки се използват:

```
(null? x)
(pair? x)
(list? x)
```

`null?` проверява дали стойността е празният списък, `pair?` — дали е наредена двойка, а `list?` — дали е правилен списък, завършващ с `'()`.

20.2.3 Основни операции със списъци

За непразен списък `l`:

```
(car l)
```

връща първия му елемент, а

```
(cdr l)
```

връща опашката му. Прилагането на `car` или `cdr` върху празен списък не е допустимо.

Дължината на списък е броят на неговите елементи. В Scheme тя се намира чрез `length`:

```
(length l)
```

Една итеративна дефиниция на същата операция е:

```
(define (length l)
  (define (length-iter arg count)
    (if (null? arg)
        count
        (length-iter (cdr arg) (+ 1 count))))
  (length-iter l 0))
```

Вътрешната процедура обхожда списъка, като премахва по един елемент чрез `cdr` и увеличава брояча. При достигане на празния списък броячът е дължината на първоначалния списък.

Обединяването на списъци се извършва чрез `append`:

```
(append l1 l2 ...)
```

Резултатът съдържа елементите на `l1`, следвани от елементите на `l2` и така нататък.

```
(append '(a b) '(c d))
```

дава

```
(a b c d)
```

Обръщането на списък се извършва чрез `reverse`:

```
(reverse '(a b c d))
```

дава

```
(d c b a)
```

20.2.4 Равенство и търсене

Scheme предоставя няколко предиката за сравнение. `eq?` проверява идентичност на обектите: резултатът е `#t`, когато двата аргумента сочат към един и същ обект в паметта, и `#f` в противен случай.

```
(eq? s1 s2)
```

`equal?` проверява структурна еквивалентност на S-изразите. При списъци това означава рекурсивно сравнение на елементите и на вложените структури.

```
(equal? '(1 (2) 3) '(1 (2) 3))
```

връща `#t`, въпреки че двата списъка могат да бъдат различни обекти в паметта.

† **Допълнение 20.5 (Бележка).** За числа и знаци в различни реализации се разграничават още `eqv?` и други стойностни сравнения. В настоящата тема основното разграничение е между идентичност чрез `eq?` и структурна еквивалентност чрез `equal?`.

Търсенето в списък чрез `memq` използва `eq?`:

```
(memq item l)
```

Ако елементът не бъде намерен, резултатът е `#f`. Ако бъде намерен, резултатът е подсписъкът, започващ от първото му срещане. Например:

```
(memq 'b '(a b c d))
```

връща

```
(b c d)
```

Процедурата `member` извършва същото търсене, но сравнява елементите чрез `equal?`:

```
(member item l)
```

Това е съществено, когато елементите са списъци или други съставни структури.

```
(member '(a b) '((a b) (c d)))
```

може да намери структурно равния елемент, докато memq би изисквал идентичност на обектите.

Елемент по позиция може да се извлече рекурсивно. В следващата дефиниция номерацията започва от 1:

```
(define (n-th n l)
  (if (= 1 n)
      (car l)
      (n-th (- n 1) (cdr l))))
```

При всяка рекурсивна стъпка се премахва първият елемент и номерът се намалява с единица. Извикването трябва да бъде с допустим номер и достатъчно дълъг непразен списък.

20.3 Списъци в Haskell

20.3.1 Конструиране и типове

В Haskell списъкът е последователност от елементи от един и същ тип. Типът на списък с елементи от тип a се записва като $[a]$. Празният списък е $[]$.

Основният конструктор е операторът `(:)`, наричан „cons“. Той добавя елемент в началото на списък:

```
(:) :: a -> [a] -> [a]
```

Операторът е дясноасоциативен. Например списъкът

```
[1,2,3,4]
```

има представяне

```
1 : 2 : 3 : 4 : []
```

което се разбира като

```
1 : (2 : (3 : (4 : [])))
```

Литералният запис на списъка е само по-удобна форма на този конструктор.

Основните функции за списъци са:

```
head :: [a] -> a
tail :: [a] -> [a]
```

```
null :: [a] -> Bool
length :: [a] -> Int
reverse :: [a] -> [a]
elem :: Eq a => a -> [a] -> Bool
init :: [a] -> [a]
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
```

`head` връща главата на непразен списък, а `tail` — опашката. `null` проверява дали списъкът е празен. `length` намира броя на елементите, а `reverse` обръща реда им. `elem` проверява принадлежност. `init` връща списъка без последния елемент. `take n l` връща първите n елемента, а `drop n l` връща списъка след премахване на първите n елемента.

```
take 2 ['a','b','c','d'] == "ab"
drop 2 ['a','b','c','d'] == "cd"
```

20.3.2 Отделяне на списъци

Haskell поддържа списъчни генератори, чрез които се дефинира нов списък от дадени списъци. Общият вид е:

```
[ израз | генератор, условие, ... ]
```

Генераторът има вида

```
образец <- израз
```

където изразът вдясно е списък, а образецът се напасва към неговите елементи. За всеки генериран елемент, който удовлетворява всички условия, се изчислява изразът пред вертикалната черта.

Например:

```
[ x^2 | x <- [1..5] ]
```

дава

```
[1,4,9,16,25]
```

При повече от един генератор се разглеждат комбинации от елементите:

```
[ (x,y) | x <- [1,2,3],
          y <- [5,6,7],
          x+y <= 8 ]
```

Резултатът е

```
[(1,5),(1,6),(1,7),(2,5),(2,6),(3,5)]
```

Списъчният генератор е удобна форма за комбиниране на генериране, проверка и трансформиране на елементи.

20.4 Функционално обработване на списъци

20.4.1 Функции от по-висок ред

Дефиниция 20.6. Функция от по-висок ред е функция, която приема функция като аргумент или връща функция като резултат.

Този механизъм позволява общият алгоритъм за обхождане на списък да бъде отделен от конкретната операция върху неговите елементи.

20.4.2 Трансформиране чрез `map`

`map` прилага една и съща функция върху всеки елемент и изгражда нов списък от резултатите.

```
map :: (a -> b) -> [a] -> [b]
```

Рекурсивната дефиниция е:

```
map _ []     = []
map f (x:xs) = f x : map f xs
```

Например:

```
map (^2) [1,2,3] == [1,4,9]
```

Функцията може да се прилага и към списък от функции:

```
map ($ 2) [(^2),(1+),(*3)]
```

Резултатът е

```
[4,3,6]
```

Тук `($ 2)` означава прилагане на всяка функция към аргумента 2.

20.4.3 Филтриране чрез `filter`

`filter` избира елементите, които удовлетворяват предикат.

```
filter :: (a -> Bool) -> [a] -> [a]
```

Рекурсивната схема е:

```
filter _ [] = []
filter p (x:xs)
  | p x       = x : rest
  | otherwise = rest
  where
    rest = filter p xs
```

Например:

```
filter odd [1..5] == [1,3,5]
```

Филтрирането не променя избраните елементи, а изгражда списък, съдържащ само тези, за които предикатът има стойност `True`.

В Racket аналогичната операция е:

```
(filter even? '(1 2 3 4 5))
```

която връща списък с четните елементи.

20.4.4 Акумулиране и свиване

Акумулирането комбинира елементите на списъка по зададено правило. В Scheme общата рекурсивна схема може да се запише така:

```
(define (acc combiner init lst)
  (if (null? lst)
      init
      (combiner (car lst)
                (acc combiner init (cdr lst)))))
```

Параметърът `combiner` е функцията за комбиниране, `init` е началната стойност, а `lst` е списъкът. Например:

```
(acc * 1 '(2 3 4))
```

изчислява произведението на елементите.

В Haskell акумулирането се реализира чрез `foldr` и `foldl`.

Дясното свиване има тип:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

За списък с елементи $x_1, x_2, \ldots, x_n$:

$$\text{foldr } op\ nv\ [x_1, x_2, \ldots, x_n] = x_1\ op\ (x_2\ op\ (\ldots\ op\ (x_n\ op\ nv)\ldots)).$$

Рекурсивната дефиниция е:

```
foldr _   nv []     = nv
foldr op  nv (x:xs) = x `op` foldr op nv xs
```

Дясното свиване първо достига до края на списъка, след което комбинира резултатите обратно към началото. Например:

```
foldr (-) 0 [1,2,3]
```

се разбира като

```
1 - (2 - (3 - 0))
```

и дава 2.

Лявото свиване има тип:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

То обработва списъка отляво надясно, като натрупаният резултат се подава към следващата стъпка:

$$\text{foldl } op\ nv\ [x_1, x_2, \ldots, x_n] = (\ldots((nv\ op\ x_1)\ op\ x_2)\ldots)\ op\ x_n.$$

Дефиницията е:

```
foldl _   nv []     = nv
foldl op  nv (x:xs) = foldl op (nv `op` x) xs
```

За разлика от `foldr`, при `foldl` акумулаторът се обновява при всяко преминаване през елемент. Например:

```
foldl (-) 0 [1,2,3]
```

се разбира като

```
((0 - 1) - 2) - 3
```

и дава -6.

† **Допълнение 20.7 (Бележка).** В предоставените материали примерът за `foldl (-) 0 '(1 2 3)` е изписан с резултат 2, което съответства на друга подредба на изваждането. По дефиницията на лявото свиване резултатът е $((0-1)-2)-3=-6$.

При операции, за които редът няма значение, например събиране или умножение, двата вида свиване дават еднакъв резултат. При неасоциативни операции като изваждане и деление разликата е съществена.

За свиване на непразен списък съществуват и `foldr1` и `foldl1`. Те не приемат начална стойност, а използват елемент от списъка като начална стойност. Затова не могат да се прилагат към празен списък.

Обръщането на списък може да се реализира чрез ляво свиване. В Scheme-подобен запис идеята е:

```
reverse l = foldl (flip (:)) [] l
```

където `flip f x y = f y x`. При всяка стъпка текущият елемент се поставя в началото на акумулатора.

20.5 Отложено оценяване

20.5.1 Обещания и отложени операции

Дефиниция 20.8. Отложена операция, или обещание, е функция или изчисление, което ще бъде изчислено и ще върне стойност в бъдещ момент от изпълнението на програмата.

Обещанието може да се изчислява асинхронно, паралелно с основната програма, или синхронно, когато основната програма изиска стойността му. В разглеждания модел се използва синхронно изчисляване при поискване.

В Scheme се използват примитивните операции `delay` и `force`:

```
(delay <израз>)
(force <обещание>)
```

`delay` не изчислява непосредствено израза, а връща обещание за бъдещото му изчисляване. `force` форсира обещанието и връща стойността на отложения израз.

Пример:

```
(define undefined (delay (+ a 3)))
(define a 5)
(force undefined)
```

Резултатът от последния израз е 8. При създаването на обещанието изразът `(+ a 3)` не е изчислен. Той се изчислява при `force`, когато `a` вече има стойност 5.

Обещанията в Scheme имат страничен ефект: вече изчислената стойност се мемоизира. Следователно при повторно форсиране се използва запазената стойност, вместо изчислението да се извършва отново.

20.5.2 Потоци

Дефиниция 20.9. Поток е списък, чиито елементи се изчисляват отложено.

По-точно потокът е празният списък или двойка

$$(h\ .\ t),$$

където h е глава, съдържаща произволен елемент, а t е обещание за останалата част на потока. При потоците се предполага строго последователен достъп: за да се достигне до следващ елемент, първо се обработва текущата глава и се форсира обещанието за опашката.

Това представяне позволява потоци с краен или безкраен брой елементи. При безкраен поток не се създават всички елементи предварително. В даден момент се изчислява само необходимата част.

20.5.3 Потоци в Scheme

За дефиниране на поток е необходима специална форма, която да отлага опашката. Следната схема използва съгласувано стандартните обещания `delay` и `force`:

```
(define-syntax cons-stream
  (syntax-rules ()
    ((cons-stream h t)
     (cons h (delay t)))))
```

Тук `cons-stream` създава наредена двойка, чиято глава е изчислена, а опашката е отложена. Основните помощни операции са:

```
(define empty-stream '())
(define head car)
(define (tail s)
  (force (cdr s)))
(define empty-stream? null?)
```

`head` извлича главата, а `tail` форсира обещанието в опашката и връща следващия поток. Стандартното обещание мемоизира резултата, така че повторно извикване на `tail` не пресмята същата опашка отново.

За краен интервал може да се дефинира:

```
(define (enum a b)
```

```
  (if (> a b)
      empty-stream
      (cons-stream a
                   (enum (+ a 1) b))))
```

Отлагането позволява дефиниране на безкраен поток от естествени числа:

```
(define (from n)
  (cons-stream n (from (+ n 1))))

(define nats (from 0))
```

`nats` започва с 0, следвано от $1, 2, 3$ и така нататък. Рекурсивното извикване на `from` не води до безкрайно изчисление при дефинирането, защото е опашка на `cons-stream` и се оценява при нужда.

20.5.4 Функции от по-висок ред за потоци

Трансформиране на поток се реализира чрез `map-stream`:

```
(define (map-stream f s)
  (cons-stream (f (head s))
               (map-stream f (tail s))))
```

Функцията изчислява текущата глава, прилага f върху нея и отлага обработването на останалата част.

Филтрирането има вида:

```
(define (filter p? s)
  (if (p? (head s))
      (cons-stream (head s)
                   (filter p? (tail s)))
      (filter p? (tail s))))
```

Ако текущата глава удовлетворява предиката, тя се включва в резултата. В противен случай се пропуска и се търси следващ елемент. За безкраен поток филтърът е приложим, когато в него се срещат достатъчно елементи, удовлетворяващи предиката.

Комбинирането на два потока чрез дадена бинарна операция се реализира чрез `zip-streams`:

```
(define (zip-streams op s1 s2)
  (cons-stream (op (head s1) (head s2))
               (zip-streams op
                            (tail s1)
                            (tail s2))))
```

Главите на двата потока се комбинират, след което рекурсивно се комбинират съответните им опашки.

Потоците могат да се дефинират и чрез пряка рекурсия:

```
(define ones
  (cons-stream 1 ones))
```

Този поток съдържа само единици. Друг пример е потокът от естествени числа:

```
(define nats
  (cons-stream 0
               (map-stream 1+ nats)))
```

Тук всяка следваща стойност се получава чрез прилагане на `1+` към съответната стойност от предходната позиция.

20.6 Отложено оценяване и списъци в Haskell

В Haskell аргументите се разглеждат като обещания, които се изчисляват при нужда. Поради това списъците в Haskell могат да се използват като потоци. Дефиницията на списък не изисква незабавно изчисляване на всички негови елементи.

Безкраен списък от последователни стойности може да се зададе чрез:

```
[n..]
```

Той представлява

$$[n, n+1, n+2, \ldots].$$

Например:

```
nats = [0..]
```

дефинира безкрайния списък на естествените числа.

Крайна част от безкраен списък се извлича чрез `take`:

```
take 6 ['a'..]
```

дава

```
"abcdef"
```

Без `take` опитът да се изведе целият безкраен списък не би завършил. Важната идея е, че само първите шест елемента са необходими и съответно само те се изчисляват.

Може да се зададе и аритметична последователност със стъпка:

```
[n,n+dx..]
```

Тя представя списъка

$$[n, n + \Delta x, n + 2\Delta x, \ldots].$$

Например четните и нечетните числа могат да се опишат като:

```
evens = [0,2..]
odds  = [1,3..]
```

Аналогично:

```
take 3 ['a','e'..]
```

дава

```
"aei"
```

Комбинирането на отложено оценяване с `map`, `filter`, `foldr` и списъчни генератори позволява да се описват изчислителни процеси върху безкрайни последователности. Изчислява се само крайният префикс, който е поискан от програмата.

20.7 Изпитен фокус

За устно представяне на темата трябва да могат да бъдат възпроизведени следните определения, операции и идеи:

- Определение за S-израз, атом, наредена двойка и списък в Scheme.
- Вътрешното представяне на списък като верига от наредени двойки, завършваща с празния списък.
- Ролята на `cons`, `car`, `cdr`, `list`, `null?`, `pair?` и `list?`.
- Операциите `length`, `append`, `reverse`, `eq?`, `equal?`, `memq` и `member`.
- Разликата между идентичност на обекти и структурна еквивалентност.
- Конструирането на списъци в Haskell чрез `(:)` и празния списък `[]`.
- Предназначението на `head`, `tail`, `null`, `length`, `reverse`, `elem`, `init`, `take` и `drop`.
- Синтаксисът и смисълът на списъчните генератори.
- Определенията и типовете на `map` и `filter`.

- Разликата между дясно свиване чрез `foldr` и ляво свиване чрез `foldl`, включително реда на асоцииране.
- Понятието за функция от по-висок ред и схемата за акумулиране с комбинираща функция и начална стойност.
- Значението на `delay` и `force`, както и мемоизирането на форсирано обещание.
- Определението за поток като списък с отложено изчисляване на опашката.
- Дефинирането на `cons-stream`, `head`, `tail` и `empty-stream`.
- Реализациите на `map-stream`, филтриране и комбиниране на потоци.
- Дефинирането на безкрайни потоци чрез рекурсия, например потока от естествени числа и потока от единици.
- Съответствието между потоците в Scheme и отложено оценяваните списъци в Haskell.
- Генерирането на безкрайни списъци чрез `[n..]` и извличането на краен префикс чрез `take`.

Въпрос 21

Синтаксис и семантика на предикатното смятане от първи ред.

По конспекта: какво трябва да мога

Дефиниции (синтаксис): език на предикатното смятане от първи ред, терм, формула, област на действие на квантор, свободна и свързана променлива, затворена формула — §21.1, стр. 302.
Дефиниции (семантика): структура и оценка на индивидните променливи — §21.2, стр. 304.
Дефиниции (семантика): вярност на формула в структура при оценка, тъждествена вярност, предикатна тавтология — §21.3, стр. 306.
Доказателство: верността при оценка не зависи от стойностите на променливите, които не участват свободно във формулата — §21.3, стр. 306.
Формулировка: лема за субституциите в *общия* ѝ вид — за произволна формула и допустима субституция — §21.4, стр. 308.
Доказателство: лемата за субституциите в *частния случай* на безкванторна формула. Само този случай се иска с доказателство — §21.4, стр. 308.
Типове задачи: определимост на свойства в дадена структура; изпълнимост на множество формули чрез посочване на модел — §21.5, стр. 311.
Литература: [13], [19], [37].

21.1 Език и синтаксис

21.1.1 Предикатен език от първи ред

Дефиниция 21.1. Езикът на предикатното смятане от първи ред $\mathcal{L}$ има логическа и нелогическа част.
Логическата част съдържа изброимо множество от индивидуални променливи

$$\mathrm{Var} = \{x, y, z, x_0, x_1, \ldots\},$$

логическите връзки $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$, кванторите $\forall, \exists$ и помощните символи $(,)$ и запетая. По избор езикът може да съдържа и формален символ за равенство $\doteq$.
Нелогическата част се състои от:

- множество $\mathrm{Const}_{\mathcal{L}}$ от индивидуални константи;
- множество $\mathrm{Func}_{\mathcal{L}}$ от функционални символи;
- множество $\mathrm{Pred}_{\mathcal{L}}$ от предикатни символи.

На всеки функционален и предикатен символ се съпоставя положително естествено число, наречено *арност*. Ако f е n-местен функционален символ, пишем $\#(f) = n$; аналогично за предикатен символ p.

Забележка 21.2. Езикът е от *първи ред*, защото кванторите се отнасят само за елементи на универсума. Не се квантифицира пряко по функции, предикати или множества от елементи.

21.1.2 Термове

Дефиниция 21.3. Множеството на термовете на $\mathcal{L}$ се дефинира индуктивно:

1. всяка индивидуална променлива е терм;
2. всяка индивидуална константа е терм;
3. ако $f \in \mathrm{Func}_{\mathcal{L}}$, $\#(f) = n$, и $\tau_1, \ldots, \tau_n$ са термове, то

$$f(\tau_1, \ldots, \tau_n)$$

е терм.

▷ **Пример 21.4.** Ако c е константа, f е едноместен, а g е двуместен функционален символ, то

$$x, \qquad c, \qquad f(x), \qquad g(f(x), c)$$

са термове.

За терм τ с $\mathrm{Var}[\tau]$ означаваме множеството от променливите, които участват в него:

$$\mathrm{Var}[x] = \{x\}, \qquad \mathrm{Var}[c] = \varnothing,$$

$$\mathrm{Var}[f(\tau_1, \ldots, \tau_n)] = \bigcup_{i=1}^{n} \mathrm{Var}[\tau_i].$$

Дефиниция 21.5 (Затворен терм)**.** Терм без променливи се нарича *затворен* или *основен* терм.

21.1.3 Формули

Дефиниция 21.6. Ако $p \in \mathrm{Pred}_{\mathcal{L}}$ е n-местен и $\tau_1, \ldots, \tau_n$ са термове, то

$$p(\tau_1, \ldots, \tau_n)$$

се нарича *атомарна формула*. Ако езикът е с формално равенство, то и

$$\tau_1 \doteq \tau_2$$

е атомарна формула.

Дефиниция 21.7. Множеството на формулите на $\mathcal{L}$ се задава индуктивно:

1. атомарните формули са формули;
2. ако φ и ψ са формули, то $\neg\varphi$,

   $$(\varphi \wedge \psi), \quad (\varphi \vee \psi), \quad (\varphi \Rightarrow \psi), \quad (\varphi \Leftrightarrow \psi)$$

   са формули;
3. ако x е индивидуална променлива и φ е формула, то

   $$\forall x\, \varphi, \qquad \exists x\, \varphi$$

   са формули.

Дефиниция 21.8. Формула ψ е *подформула* на формула φ, ако ψ се получава като част от синтактичното построение на φ.

21.1.4 Област на действие; свободни и свързани променливи

Дефиниция 21.9. В подформула от вида

$$\forall x\, \psi \qquad \text{или} \qquad \exists x\, \psi$$

формулата ψ се нарича *област на действие* на съответния квантор.

Дефиниция 21.10. Едно участие на променливата x във формула е *свързано*, ако лежи в областта на действие на квантор по x. В противен случай участието е *свободно*.

Най-близкият квантор по x, чиято област на действие съдържа дадено участие, определя дали това участие е свързано. Една и съща променлива може да има както свободни, така и свързани участия.

▷ **Пример 21.11.** Във формулата

$$p(x) \wedge \forall x\, q(x, y)$$

първото участие на x е свободно, а второто е свързано. Всички участия на y са свободни.

С $\mathrm{Var}^{\mathrm{free}}[\varphi]$ означаваме множеството на свободните променливи на φ, а с $\mathrm{Var}^{\mathrm{bd}}[\varphi]$ — множеството на променливите с поне едно свързано участие. Важни рекурсивни равенства са:

$$\mathrm{Var}^{\mathrm{free}}[p(\tau_1, \ldots, \tau_n)] = \bigcup_{i=1}^{n} \mathrm{Var}[\tau_i],$$

$$\mathrm{Var}^{\mathrm{free}}[\neg\varphi] = \mathrm{Var}^{\mathrm{free}}[\varphi],$$

$$\mathrm{Var}^{\mathrm{free}}[\varphi \circ \psi] = \mathrm{Var}^{\mathrm{free}}[\varphi] \cup \mathrm{Var}^{\mathrm{free}}[\psi], \qquad \circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\},$$

$$\mathrm{Var}^{\mathrm{free}}[\forall x\, \varphi] = \mathrm{Var}^{\mathrm{free}}[\varphi] \setminus \{x\}, \qquad \mathrm{Var}^{\mathrm{free}}[\exists x\, \varphi] = \mathrm{Var}^{\mathrm{free}}[\varphi] \setminus \{x\}.$$

Дефиниция 21.12. Формула φ се нарича *затворена*, ако

$$\mathrm{Var}^{\mathrm{free}}[\varphi] = \varnothing.$$

21.2 Структури, оценки и стойности на термове

Дефиниция 21.13. Структура за езика $\mathcal{L}$ е двойка

$$\mathcal{A} = \langle A, I \rangle,$$

където $A \neq \varnothing$ е *универсум* или *носител* на структурата, а I е интерпретация на нелогическите символи:

- за всяка константа c е зададен елемент $c^{\mathcal{A}} \in A$;
- за всеки n-местен функционален символ f е зададена функция

$$f^{\mathcal{A}} : A^n \to A;$$

- за всеки n-местен предикатен символ p е зададена релация

$$p^{\mathcal{A}} \subseteq A^n.$$

Забележка 21.14. Функционалният символ f е синтактичен знак, докато $f^{\mathcal{A}}$ е функция върху универсума. Аналогично p е знак, а $p^{\mathcal{A}}$ е релация.

▷ **Пример 21.15.** За език с константа 0, двуместен функционален символ $+$ и двуместен предикатен символ $<$ естествените числа могат да се разглеждат като структура

$$\mathcal{N} = \langle \mathbb{N}, 0^{\mathcal{N}}, +^{\mathcal{N}}, <^{\mathcal{N}} \rangle.$$

Дефиниция 21.16. *Оценка на индивидуалните променливи* в структура $\mathcal{A}$ е функция

$$v : \mathrm{Var} \to A.$$

За $a \in A$ с v_x^a означаваме оценката, модифицирана в x с a:

$$v_x^a(y) = \begin{cases} a, & y = x, \\ v(y), & y \neq x. \end{cases}$$

Стойността на терм τ в $\mathcal{A}$ при оценка v, означавана с

$$[\![\tau]\!]_v^{\mathcal{A}},$$

се определя рекурсивно:

$$[\![x]\!]_v^{\mathcal{A}} = v(x), \qquad [\![c]\!]_v^{\mathcal{A}} = c^{\mathcal{A}},$$
$$[\![f(\tau_1, \ldots, \tau_n)]\!]_v^{\mathcal{A}} = f^{\mathcal{A}}\big([\![\tau_1]\!]_v^{\mathcal{A}}, \ldots, [\![\tau_n]\!]_v^{\mathcal{A}}\big).$$

Твърдение 21.17. Нека τ е терм, а v_1, v_2 са оценки в $\mathcal{A}$. Ако

$$v_1(x) = v_2(x) \quad \text{за всяко } x \in \mathrm{Var}[\tau],$$

то

$$[\![\tau]\!]_{v_1}^{\mathcal{A}} = [\![\tau]\!]_{v_2}^{\mathcal{A}}.$$

Доказателство. Доказателството е индукция по построението на τ. За променлива твърдението следва от условието; за константа стойността не зависи от оценката. При

$$\tau = f(\tau_1, \ldots, \tau_n)$$

оценките съвпадат върху променливите на всеки τ_i. Индукционната хипотеза дава равенство на стойностите на всички аргументи, а после и равенство на стойностите след прилагане на $f^{\mathcal{A}}$. □

Следствие 21.18. Стойността на затворен терм не зависи от оценката.

Доказателство. Множеството от променливи на затворен терм е празно. Всякакви две оценки съвпадат върху това множество. Предходното твърдение за зависимостта само от променливите на терма тогава дава еднакви стойности при двете оценки. □

21.3 Семантика на формулите

21.3.1 Вярност при оценка

За формула φ, структура $\mathcal{A}$ и оценка v записваме

$$\mathcal{A} \models \varphi[v]$$

и четем: „φ е вярна в $\mathcal{A}$ при оценка v“. Релацията се определя рекурсивно.
За атомарна формула:

$$\mathcal{A} \models p(\tau_1, \ldots, \tau_n)[v] \quad \iff \quad \left([\![\tau_1]\!]_v^{\mathcal{A}}, \ldots, [\![\tau_n]\!]_v^{\mathcal{A}}\right) \in p^{\mathcal{A}}.$$

При наличие на формално равенство:

$$\mathcal{A} \models (\tau_1 \doteq \tau_2)[v] \quad \iff \quad [\![\tau_1]\!]_v^{\mathcal{A}} = [\![\tau_2]\!]_v^{\mathcal{A}}.$$

За логическите връзки се използва класическата двузначна семантика:

$$\mathcal{A} \models \neg\varphi[v] \quad \iff \quad \mathcal{A} \not\models \varphi[v],$$

$$\mathcal{A} \models (\varphi \wedge \psi)[v] \quad \iff \quad \mathcal{A} \models \varphi[v] \text{ и } \mathcal{A} \models \psi[v],$$
$$\mathcal{A} \models (\varphi \vee \psi)[v] \quad \iff \quad \mathcal{A} \models \varphi[v] \text{ или } \mathcal{A} \models \psi[v],$$
$$\mathcal{A} \models (\varphi \Rightarrow \psi)[v] \quad \iff \quad \mathcal{A} \not\models \varphi[v] \text{ или } \mathcal{A} \models \psi[v].$$

За кванторите:

$$\mathcal{A} \models \forall x\, \varphi[v] \quad \iff \quad \text{за всяко } a \in A : \mathcal{A} \models \varphi[v_x^a],$$
$$\mathcal{A} \models \exists x\, \varphi[v] \quad \iff \quad \text{съществува } a \in A : \mathcal{A} \models \varphi[v_x^a].$$

▷ **Пример 21.19.** Нека $p^{\mathcal{A}} \subseteq A$ е едноместна релация. Тогава

$$\mathcal{A} \models \forall x\, p(x)[v]$$

означава, че $p^{\mathcal{A}} = A$, а

$$\mathcal{A} \models \exists x\, p(x)[v]$$

означава, че $p^{\mathcal{A}} \neq \varnothing$.

21.3.2 Зависимост само от свободните променливи

★ **Теорема 21.20.** Нека φ е формула, а v_1, v_2 са оценки в структурата $\mathcal{A}$. Ако

$$v_1(x) = v_2(x) \quad \text{за всяко } x \in \mathrm{Var}^{\mathrm{free}}[\varphi],$$

то

$$\mathcal{A} \models \varphi[v_1] \quad \iff \quad \mathcal{A} \models \varphi[v_2].$$

Доказателство. Доказваме с индукция по построението на φ.
Нека $\varphi = p(\tau_1, \ldots, \tau_n)$. Всички променливи на τ_i са свободни във формулата. По твърдението за термовете

$$[\![\tau_i]\!]^{\mathcal{A}}_{v_1} = [\![\tau_i]\!]^{\mathcal{A}}_{v_2} \qquad (1 \le i \le n).$$

Следователно едната наредена n-торка принадлежи на $p^{\mathcal{A}}$ точно тогава, когато принадлежи другата. Случаят с формално равенство е аналогичен.
При отрицание и при двоичните логически връзки твърдението следва непосредствено от индукционната хипотеза и класическата семантика.
Нека

$$\varphi = \forall x\, \psi.$$

Тогава свободните променливи на φ са свободните променливи на ψ, различни от x. За произволно $a \in A$ оценките $v^a_{1\,x}$ и $v^a_{2\,x}$ съвпадат върху всички свободни променливи на ψ: за x и двете дават a, а за останалите следва от условието. По индукционната хипотеза

$$\mathcal{A} \models \psi[v^a_{1\,x}] \iff \mathcal{A} \models \psi[v^a_{2\,x}].$$

Следователно всеобщото квантифициране дава еднаква стойност. Случаят $\exists x\, \psi$ е аналогичен. □

Следствие 21.21. Ако φ е затворена формула, нейната вярност в дадена структура не зависи от оценката.

Доказателство. За затворена формула $\mathrm{FV}(\varphi) = \varnothing$. Всякакви две оценки съвпадат върху свободните ѝ променливи. По доказаната теорема за независимост от останалите променливи истинностната стойност е една и съща. □

21.3.3 Тъждествена вярност и предикатни тавтологии

Дефиниция 21.22. Формулата φ е *тъждествено вярна в структурата* $\mathcal{A}$, ако

$$\mathcal{A} \models \varphi[v]$$

за всяка оценка v в $\mathcal{A}$. Пишем

$$\mathcal{A} \models \varphi.$$

Дефиниция 21.23. Формулата φ е *предикатна тавтология* или *общовалидна*, ако е тъждествено вярна във всяка структура за езика:

$$\models \varphi.$$

Забележка 21.24. За затворена формула е достатъчно да се провери една произволна оценка: по предходното следствие всички оценки дават една и съща истинностна стойност.

21.4 Субституции

21.4.1 Субституция на термове

Дефиниция 21.25. Субституция е съпоставяне, което на всяка индивидуална променлива задава терм. Често се задава крайно чрез

$$s = \left\{ \frac{x_1}{t_1}, \ldots, \frac{x_n}{t_n} \right\},$$

където $x_1, \ldots, x_n$ са различни променливи. За терм τ с τs означаваме терма, получен при едновременно заместване на съответните променливи.

Лема 21.26 (Лема за субституциите на термове)**.** Нека $\tau(x_1, \ldots, x_n)$ е терм, $t_1, \ldots, t_n$ са термове, а v е оценка в $\mathcal{A}$. Ако ω е оценка, зададена чрез

$$\omega(x_i) = [\![t_i]\!]_v^{\mathcal{A}} \quad (1 \le i \le n),$$

и съвпадаща с v върху останалите променливи, то

$$[\![\tau[x_1/t_1, \ldots, x_n/t_n]]\!]_v^{\mathcal{A}} = [\![\tau]\!]_\omega^{\mathcal{A}}.$$

Доказателство. Доказателството е индукция по построението на τ. За $\tau = x_i$ равенството следва от дефиницията на ω; за променлива, която не се заменя, и за константа е непосредствено. При

$$\tau = f(\tau_1, \ldots, \tau_k)$$

се прилага индукционната хипотеза към всеки аргумент и после дефиницията на стойността на функционален терм. □

21.4.2 Допустимост на заместването във формули

При замяна във формула се заменят само свободните участия на съответната променлива. Трябва да се избягва *захващане* на променлива.

Дефиниция 21.27. Променлива y е опасна за формула φ при субституция s, ако y участва в терма $s(x)$ за някоя свободна променлива x на φ, за която $s(x) \neq x$.

Заместването е допустимо, когато свободно участие, заменено с терм, не попада в област на действие на квантор по променлива, участваща в този терм.

▷ **Пример 21.28.** Нека

$$\varphi = \forall y\,(q(x, y) \Rightarrow p(y)).$$

Замяната на свободното x с $f(y)$ не е допустима: полученото

$$\forall y\,(q(f(y), y) \Rightarrow p(y))$$

променя ролята на y, защото то става свързано от квантора.

Дефиниция 21.29 (Допустима субституция)**.** Субституцията s е *допустима* за формулата φ, ако за всяка свободна променлива x на φ с $s(x) \neq x$ никое свободно участие на x в φ не попада в областта на действие на квантор по променлива, която участва в терма $s(x)$. Тоест никоя променлива на заместващия терм не се свързва при заместването.

Конспектът иска твърдението за стойността при замяна на променливи с термове да бъде *формулирано* в общия си вид и да бъде *доказано* в частния случай на безкванторна формула. Затова тук стоят и двете: общата формулировка, която трябва да може да се възпроизведе точно, и подробната индукция за безкванторния случай.

★ **Теорема 21.30 (Лема за субституциите, общ вид).** Нека φ е произволна формула, s е субституция, *допустима* за φ, v е оценка в структурата $\mathcal{A}$, а ω е оценката

$$\omega(x) = [\![xs]\!]^{\mathcal{A}}_{v}$$

за всяка променлива x. Тогава

$$\mathcal{A} \models (\varphi s)[v] \quad \Longleftrightarrow \quad \mathcal{A} \models \varphi[\omega].$$

Забележка 21.31. Изискването за допустимост не може да се пропусне. За $\varphi = \exists y\,(x \neq y)$ и $s(x) = y$ лявата страна дава $\exists y\,(y \neq y)$, което е невярно във всяка структура, докато дясната е вярна във всяка структура с поне два елемента. Точно това е захващането на променлива от предходния пример.

Доказателство. Индукция по построението на φ. Атомарните случаи и случаите с $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ не използват допустимостта и са изписани подробно в теоремата за безкванторния случай по-долу; повторени тук, те дават базата и пропозиционалните стъпки.

Остава случаят с квантор. Нека $\varphi = \forall y\,\psi$ и нека s_y е субституцията, която съвпада с s извън y и полага $s_y(y) = y$: заместват се само свободните участия, а y е свързана в φ. Тогава $\varphi s = \forall y\,(\psi s_y)$ и s_y е допустима за ψ. По дефиниция на семантиката

$$\mathcal{A} \models (\varphi s)[v] \quad \Longleftrightarrow \quad \text{за всяко } a \in A:\ \mathcal{A} \models (\psi s_y)[v^a_y].$$

Прилагаме индукционната хипотеза за ψ, s_y и оценката v^a_y. Тя дава

$$\mathcal{A} \models (\psi s_y)[v^a_y] \quad \Longleftrightarrow \quad \mathcal{A} \models \psi[\omega_a], \qquad \omega_a(x) = [\![xs_y]\!]^{\mathcal{A}}_{v^a_y}.$$

Сравняваме ω_a с ω^a_y върху свободните променливи на ψ. За $x = y$ имаме $ys_y = y$, тоест $\omega_a(y) = a = \omega^a_y(y)$. За свободна променлива $x \neq y$ на ψ, която е свободна и в φ, имаме $xs_y = xs$. Ако $s(x) = x$, стойността е $v(x)$ и в двата случая. Ако $s(x) \neq x$, допустимостта на s за φ казва, че y не участва в терма $s(x)$; тогава по твърдението за зависимостта на стойността на терм само от участващите в него променливи

$$[\![s(x)]\!]^{\mathcal{A}}_{v^a_y} = [\![s(x)]\!]^{\mathcal{A}}_{v} = \omega(x) = \omega^a_y(x).$$

Следователно ω_a и ω^a_y съвпадат върху $\mathrm{Var}^{\mathrm{free}}[\psi]$ и по теоремата за зависимост само от свободните променливи

$$\mathcal{A} \models \psi[\omega_a] \quad \Longleftrightarrow \quad \mathcal{A} \models \psi[\omega^a_y].$$

Събирайки еквивалентностите за всяко $a \in A$, получаваме

$$\mathcal{A} \models (\varphi s)[v] \iff \text{за всяко } a \in A : \ \mathcal{A} \models \psi[\omega_y^a] \iff \mathcal{A} \models \forall y\, \psi\, [\omega].$$

Случаят $\varphi = \exists y\, \psi$ се получава по същия начин, като „за всяко a“ се замени със „съществува a“. □

★ **Теорема 21.32 (Лема за субституциите, безкванторен случай).** Нека φ е безкванторна формула, s е субституция, v е оценка в $\mathcal{A}$, а ω е оценката

$$\omega(x) = [\![xs]\!]_v^{\mathcal{A}}$$

за всяка променлива x. Тогава

$$\mathcal{A} \models (\varphi s)[v] \quad \iff \quad \mathcal{A} \models \varphi[\omega].$$

Това е *частният случай*, чието доказателство конспектът изисква. При безкванторна φ всяка субституция е допустима, защото няма квантор, който да свърже променлива, затова хипотезата за допустимост отпада.

Доказателство. Доказваме с индукция по построението на безкванторната формула φ.
Нека

$$\varphi = p(\tau_1, \ldots, \tau_n).$$

По лемата за субституциите на термове:

$$[\![\tau_i s]\!]_v^{\mathcal{A}} = [\![\tau_i]\!]_\omega^{\mathcal{A}} \qquad (1 \le i \le n).$$

Следователно

$$\begin{aligned} \mathcal{A} \models (\varphi s)[v] &\iff \left([\![\tau_1 s]\!]_v^{\mathcal{A}}, \ldots, [\![\tau_n s]\!]_v^{\mathcal{A}}\right) \in p^{\mathcal{A}} \\ &\iff \left([\![\tau_1]\!]_\omega^{\mathcal{A}}, \ldots, [\![\tau_n]\!]_\omega^{\mathcal{A}}\right) \in p^{\mathcal{A}} \\ &\iff \mathcal{A} \models \varphi[\omega]. \end{aligned}$$

Случаят с равенство е аналогичен.
Нека $\varphi = \neg\psi$. По индукционната хипотеза

$$\mathcal{A} \models (\psi s)[v] \iff \mathcal{A} \models \psi[\omega].$$

След отрицание получаваме търсената еквивалентност за $\neg\psi$.
Нека

$$\varphi = (\psi \wedge \chi).$$

Тогава

$$(\psi \wedge \chi)s = (\psi s \wedge \chi s).$$

По индукционната хипотеза:

$$\mathcal{A} \models (\psi s)[v] \iff \mathcal{A} \models \psi[\omega],$$

$$\mathcal{A} \models (\chi s)[v] \iff \mathcal{A} \models \chi[\omega].$$

Следователно

$$\mathcal{A} \models ((\psi \wedge \chi)s)[v] \iff \mathcal{A} \models (\psi \wedge \chi)[\omega].$$

Случаите за $\vee$, $\Rightarrow$ и $\Leftrightarrow$ са аналогични. Понеже φ е безквантorна, случаи с квантори няма. □

21.5 Определимост и изпълнимост

21.5.1 Определими свойства

Нека $\varphi[x_1, \ldots, x_n]$ означава, че всички свободни променливи на φ са сред $x_1, \ldots, x_n$. Формулата определя в структурата $\mathcal{A}$ множеството

$$D_\varphi = \{(a_1, \ldots, a_n) \in A^n \mid \mathcal{A} \models \varphi[v] \text{ за оценка } v(x_i) = a_i\}.$$

Това е коректно определено именно защото истинността на формулата зависи само от свободните ѝ променливи.

▷ **Пример 21.33.** В структурата

$$\mathcal{N} = \langle \mathbb{N}, 0, +, < \rangle$$

формулата

$$\exists z\,(x = z + z)$$

определя множеството на четните естествени числа.

21.5.2 Изпълнимост на множества от формули

Дефиниция 21.34. Множество Γ от формули е *изпълнимо в структурата* $\mathcal{A}$, ако съществува оценка v, за която

$$\mathcal{A} \models \psi[v] \qquad \text{за всяка } \psi \in \Gamma.$$

Дефиниция 21.35. Структурата $\mathcal{A}$ е *модел* на Γ, ако

$$\mathcal{A} \models \psi \qquad \text{за всяка } \psi \in \Gamma.$$

Пишем

$$\mathcal{A} \models \Gamma.$$

▷ **Пример 21.36.** Нека езикът съдържа едноместен предикат P и двуместен предикат R. Множеството

$$\Gamma = \{\forall x\, \exists y\, R(x,y),\ \forall x\, (P(x) \Rightarrow \exists y\, R(y,x)),\ \exists x\, P(x)\}$$

е изпълнимо. Модел е структура с универсум $A = \{a\}$, за която

$$P^{\mathcal{A}} = \{a\}, \qquad R^{\mathcal{A}} = \{(a,a)\}.$$

Действително, всяка квантифицирана променлива може да получи стойност a, а всички формули от Γ са верни.

† **Допълнение 21.37 (Бележка).** При формули със свободни променливи трябва да се различават:

„има една оценка, която удовлетворява всички формули“

и

„всички формули са тъждествено верни в структурата“.

Второто е по-силно условие. За затворени формули двете понятия съвпадат, защото истинността не зависи от оценката.

21.6 Изпитен фокус

- Да се дефинират език от първи ред, арност, терм, атомарна формула и предикатна формула.
- Да се обяснят област на действие на квантор, свободно и свързано участие, свободна променлива и затворена формула.
- Да се дефинират структура, универсум, интерпретация, оценка и модифицирана оценка v_x^a.
- Да се даде рекурсивната семантика на термовете и формулите, особено на $\forall$ и $\exists$.
- Да се различават вярност при оценка $\mathcal{A} \models \varphi[v]$, тъждествена вярност $\mathcal{A} \models \varphi$ и предикатна тавтология $\models \varphi$.
- Да се възпроизведе доказателствената идея, че стойността на формула зависи само от свободните ѝ променливи: индукция по построението на формулата; при кванторите се сравняват модифицираните оценки.
- Да се формулира лемата за субституциите на термове.
- Да се формулира лемата за субституциите в *общия* ѝ вид — за произволна формула φ и субституция s, *допустима* за φ: $\mathcal{A} \models (\varphi s)[v] \Leftrightarrow \mathcal{A} \models \varphi[\omega]$, където $\omega(x) = [\![xs]\!]_v^{\mathcal{A}}$. Конспектът иска формулировката в общия вид.
- Да се докаже лемата за субституциите в частния случай на *безкванторна* формула чрез индукция по построението ѝ. Само този частен случай се изисква с доказателство.
- Да се обясни защо е нужна допустимост на заместването и да се даде пример за захващане на променлива.
- Да се решават задачи за определимост чрез формула и за изпълнимост чрез явно посочване на структура, интерпретации и при нужда оценка.

Въпрос 22

Изводимост и компютърно генериране на доказателства.

По конспекта: какво трябва да мога

Дефиниции: литерал, дизюнкт, верен в структура дизюнкт, изпълнимо множество от дизюнкти — §22.1, стр. 313.
Дефиниции: резолвента и резолютивен извод — §22.2; предикатна либерална резолвента — §22.3, стр. 317.
Формулировки: теоремите за коректност и за пълнота на резолютивната изводимост — §22.3, стр. 317.
Доказателство: на теоремата за *коректност* — §22.3, стр. 317.
Типове задачи: опровержения чрез резолюция — §22.4; дефиниране на предикати на Пролог — §22.5, стр. 321.
† **Допълнително:** доказателството на пълнотата (§22.3) използва теоремата на Ербран като външен резултат и надхвърля изискването — анотацията иска пълнотата само формулирана.
Литература: [13], [19], [37].

22.1 Основни понятия

Методът на резолюциите е доказателствен метод, който свежда проверката за логическо следване до проверка за неизпълнимост. Идеята е проста: за да се докаже, че от множество формули Γ следва формулата φ, разглеждаме множеството

$$\Gamma \cup \{\neg\varphi\}.$$

Ако то е неизпълнимо, то няма структура, в която всички формули от Γ са верни, а φ е невярна; следователно $\Gamma \models \varphi$. Резолюцията търси синтактично свидетелство за тази неизпълнимост: извеждане на празния дизюнкт.

Дефиниция 22.1. Литерал е атомарна формула или отрицание на атомарна формула. Ако L е литерал, то противоположният му литерал ще означаваме с $\overline{L}$:

$$\overline{P(t_1,\ldots,t_n)} = \neg P(t_1,\ldots,t_n), \qquad \overline{\neg P(t_1,\ldots,t_n)} = P(t_1,\ldots,t_n).$$

Дефиниция 22.2. Елементарна дизюнкция е безкванторна формула, изградена от литерали само с дизюнкция. Формула е в *конюнктивна нормална форма* (КНФ), ако е конюнкция от елементарни дизюнкции.

В резолютивния метод не е необходимо да се пазят редът и повторенията на литералите в една дизюнкция. Затова дизюнкциите се представят като множества.

Дефиниция 22.3. Дизюнкт е крайно множество от литерали. Допуска се и празният дизюнкт, означаван с $\square$ или $\varnothing$.

Например формулата

$$P(x) \vee \neg Q(f(x)) \vee P(x)$$

съответства на дизюнкта

$$\{P(x), \neg Q(f(x))\}.$$

Празният дизюнкт няма литерали и съответства на лъжа: той е неверен при всяка оценка.

Дефиниция 22.4. Нека $\mathcal{M}$ е структура, а v е оценка на индивидуалните променливи. Дизюнктът D е *верен в $\mathcal{M}$ при v*, ако поне един негов литерал е верен:

$$\mathcal{M} \models_v D \quad \Longleftrightarrow \quad (\exists L \in D)\, \mathcal{M} \models_v L.$$

Той е неверен в $\mathcal{M}$ при v, ако всички негови литерали са неверни.

Дефиниция 22.5. Дизюнктът D е *тъждествено верен в структурата $\mathcal{M}$*, ако

$$(\forall v)\, \mathcal{M} \models_v D.$$

Множество от дизюнкти Γ е *изпълнимо*, ако съществува структура $\mathcal{M}$, в която всеки дизюнкт от Γ е тъждествено верен:

$$\mathcal{M} \models \Gamma \quad \Longleftrightarrow \quad (\forall D \in \Gamma)\,(\forall v)\, \mathcal{M} \models_v D.$$

Тогава $\mathcal{M}$ се нарича модел на Γ.

Важно е да се различават дизюнктът $\{p, \neg p\}$ и множеството от формули $\{p, \neg p\}$. Първият е тавтологичен дизюнкт, понеже съдържа противоположни литерали. Второто множество е неизпълнимо, понеже изисква едновременно p и $\neg p$ да са верни.

Забележка 22.6. Ако формула е приведена в КНФ

$$C_1 \wedge \cdots \wedge C_m,$$

то на нея съответства множеството от дизюнкти $\{D_1, \ldots, D_m\}$, където D_i е множеството от литералите в C_i. Изпълнимостта на формулата е еквивалентна на изпълнимостта на съответното множество от дизюнкти.

22.2 Резолюция за дизюнкти

22.2.1 Същинска резолвента

Най-напред разглеждаме случая, в който два дизюнкта съдържат буквално противоположни литерали.

Дефиниция 22.7. Нека D_1 и D_2 са дизюнкти, $L \in D_1$ и $\overline{L} \in D_2$. Дизюнктът

$$R_L(D_1, D_2) = (D_1 \setminus \{L\}) \cup (D_2 \setminus \{\overline{L}\})$$

се нарича *непосредствена резолвента* на D_1 и D_2 относно литерала L.

Например:

$$\{p, q\}, \qquad \{\neg p, r\}$$

имат резолвента

$$\{q, r\}.$$

При резолюция се премахва точно една двойка противоположни литерали. Ако два дизюнкта съдържат повече от една противоположна двойка, не се премахват всички двойки едновременно.

▷ **Пример 22.8.** За дизюнктите

$$D_1 = \{p, q, r\}, \qquad D_2 = \{\neg p, \neg q, r\}$$

резолвентите са

$$\{q, \neg q, r\} \quad \text{и} \quad \{p, \neg p, r\},$$

но не и $\{r\}$.

Лема 22.9 (Резолютивна лема)**.** Нека D е резолвента на D_1 и D_2. За всяка булева интерпретация I е вярно:

$$I \models D_1 \text{ и } I \models D_2 \quad \Longrightarrow \quad I \models D.$$

Доказателство. Нека

$$D = (D_1 \setminus \{L\}) \cup (D_2 \setminus \{\overline{L}\}).$$

Разглеждаме два случая.
Ако $I \models L$, то $I \not\models \overline{L}$. Понеже $I \models D_2$, някой литерал от D_2, различен от $\overline{L}$, трябва да е верен. Той принадлежи на D, следователно $I \models D$.
Ако $I \not\models L$, то понеже $I \models D_1$, някой литерал от D_1, различен от L, е верен. Той принадлежи на D, следователно $I \models D$. □

22.2.2 Резолютивен извод

Дефиниция 22.10. Нека Γ е множество от дизюнкти. *Резолютивен извод* на дизюнкта δ от Γ е крайна редица

$$\delta_0, \delta_1, \ldots, \delta_n, \qquad \delta_n = \delta,$$

такава че за всеки δ_i е изпълнено поне едно от условията:

1. $\delta_i \in \Gamma$;
2. δ_i е резолвента на два дизюнкта, срещащи се по-рано в редицата.

Казваме, че δ се извежда от Γ с резолюция и пишем

$$\Gamma \vdash_r \delta.$$

★ **Теорема 22.11 (Коректност на резолютивната изводимост).** Ако

$$\Gamma \vdash_r \square,$$

то Γ е неизпълнимо.

Доказателство. Нека

$$\delta_0, \delta_1, \ldots, \delta_n = \square$$

е резолютивен извод от Γ. Ще докажем по индукция по i, че всеки модел на Γ удовлетворява δ_i.
Нека $\mathcal{M} \models \Gamma$. Ако $\delta_i \in \Gamma$, то по дефиниция

$$\mathcal{M} \models \delta_i.$$

Ако δ_i е резолвента на два по-ранни дизюнкта δ_j и δ_k, то по индукционната хипотеза

$$\mathcal{M} \models \delta_j \quad \text{и} \quad \mathcal{M} \models \delta_k.$$

От резолютивната лема следва

$$\mathcal{M} \models \delta_i.$$

Следователно всеки модел на Γ би трябвало да удовлетворява и $\square$. Но празният дизюнкт няма верен литерал и е неверен при всяка интерпретация. Следователно Γ няма модел. $\square$

▷ **Пример 22.12.** Нека

$$\Gamma = \big\{\{p, q\}, \{\neg p, q\}, \{p, \neg q\}, \{\neg p, \neg q\}\big\}.$$

Получаваме извода

1. $\{p, \neg q\}$ $\in \Gamma$,
2. $\{p, q\}$ $\in \Gamma$,
3. $\{p\}$ резолвента на 1 и 2 относно q,
4. $\{\neg p, \neg q\}$ $\in \Gamma$,
5. $\{\neg p, q\}$ $\in \Gamma$,
6. $\{\neg p\}$ резолвента на 4 и 5 относно q,
7. $\square$ резолвента на 3 и 6 относно p.

Следователно Γ е неизпълнимо.

22.3 Предикатна либерална резолюция

При предикатните дизюнкти противоположните литерали невинаги са синтактично еднакви. Например $P(x)$ и $\neg P(f(y))$ могат да станат противоположни след подходяща субституция. Затова е необходима унификация.

22.3.1 Субституции и унификация

Дефиниция 22.13. Субституция е крайно множество

$$\sigma = \left\{ \frac{x_1}{t_1}, \ldots, \frac{x_n}{t_n} \right\},$$

където $x_1, \ldots, x_n$ са различни индивидуални променливи, а $t_1, \ldots, t_n$ са термове. Резултатът от прилагането на σ върху дизюнкта ε се означава с $\varepsilon\sigma$.

Дефиниция 22.14. Субституцията σ е унификатор на термовете $t_1, \ldots, t_m$, ако

$$t_1\sigma = \cdots = t_m\sigma.$$

Тя е най-общ унификатор, ако всеки друг унификатор се получава от нея чрез последваща субституция.

При унификацията трябва да се спазва проверката за участие: не може да се замества x с терм, в който участва самото x. Например уравнението

$$x = f(x)$$

няма допустим унификатор.

▷ **Пример 22.15.** Литералите

$$P(x, f(y)) \quad \text{и} \quad \neg P(f(z), f(a))$$

се унифицират чрез

$$\sigma = \left\{ \frac{x}{f(z)}, \frac{y}{a} \right\}.$$

След прилагане на σ получаваме противоположните литерали

$$P(f(z), f(a)) \quad \text{и} \quad \neg P(f(z), f(a)).$$

22.3.2 Предикатна либерална резолвента

Дефиниция 22.16 (Предикатна либерална резолвента)**.** Нека C_1, C_2 са дизюнкти с предварително преименувани, непресичащи се множества от променливи. Избираме непразни множества $L_1 \subseteq C_1$ от положителни литерали и $L_2 \subseteq C_2$ от отрицателни литерали. Ако атомите в L_1 и отрицанията на литералите в L_2 имат общ унификатор σ, дизюнктът

$$D = \big((C_1 \setminus L_1) \cup (C_2 \setminus L_2)\big)\sigma$$

е либерална резолвента. Може да се избира най-общ унификатор; двата родителя могат да разменят ролите си. Дизюнктите са множества, така че повторените литерали се сливат. При $|L_1| = |L_2| = 1$ получаваме бинарна резолюция.

Забележка 22.17. Изборът на няколко литерала включва необходимото за пълнотата сливане на унифицируеми литерали. Еквивалентно представяне използва бинарна резолюция и отделно правило за факторизация. Само бинарната стъпка от предишната редакция не описва цялото използвано тук правило.

▷ **Пример 22.18.** Нека

$$\varepsilon_1 = \{P(x), Q(x)\}, \qquad \varepsilon_2 = \{\neg P(f(y)), R(y)\}.$$

За $P(x)$ и $\neg P(f(y))$ най-общ унификатор е

$$\sigma = \left\{ \frac{x}{f(y)} \right\}.$$

Следователно либералната резолвента е

$$\{Q(f(y)), R(y)\}.$$

Дефиниция 22.19. Нека Γ е множество от предикатни дизюнкти. *Предикатен резолютивен извод* на δ от Γ е крайна редица от дизюнкти, в която всеки член е или елемент на Γ, или предикатна либерална резолвента на два по-ранни члена. Отново пишем

$$\Gamma \vdash_r \delta.$$

† **Допълнение 22.20 (Бележка).** В някои представяния предикатната резолюция се описва чрез дизюнкти с ограничения, които записват равенствата между аргументите на резолвираните литерали. След елиминиране на ограниченията чрез алгоритъм за унификация се получава същата резолвента с приложена субституция. Тук използваме непосредственото описание чрез най-общ унификатор.

22.3.3 Коректност и пълнота

★ **Теорема 22.21 (Коректност на предикатната резолютивна изводимост).** Ако

$$\Gamma \vdash_r \square,$$

то множеството от предикатни дизюнкти Γ е неизпълнимо.

Доказателство. Нека структура $\mathcal{M}$ удовлетворява всички изходни дизюнкти при всяка оценка. Универсално верният дизюнкт остава верен след замяна на променливи с термове: оценката на термовете задава съответната оценка на заменените променливи. Следователно двата родителя след прилагане на σ са верни. При фиксирана оценка избраните положителни литерали имат една обща стойност A, а отрицателните — $\neg A$. Ако резолвентата беше невярна, всички останали литерали и в двата родителя биха били неверни. Тогава първият родител би изисквал A, а вторият $\neg A$, противоречие. Значи резолвентата е вярна при всяка оценка. Индукция по дължината на извода показва, че всеки негов член е верен в $\mathcal{M}$. Празният дизюнкт е винаги неверен, затова изводът му е несъвместим със съществуването на модел. $\square$

Доказателствената идея е същата като при същинската резолюция. Ако дадена структура удовлетворява двата родителски дизюнкта при всички оценки, то след прилагане на унифициращата субституция тя удовлетворява и резолвентата. Следователно всеки дизюнкт в резолютивния извод е логическо следствие на изходното множество. Появата на $\square$ противоречи на съществуването на модел.

† **Допълнение 22.22 (Обхват: доказателството по-долу е допълнително).** Конспектът иска пълнотата да бъде *формулирана*; с доказателство се иска само коректността. Доказателството, което следва, използва и външния резултат — теоремата на Ербран. Формулировката трябва да се знае точно; доказателството е за разбиране, не за възпроизвеждане на изпита.

★ **Теорема 22.23 (Пълнота на резолютивната изводимост).** За множество Γ от първоредови дизюнкти в език без формално равенство е в сила:

$$\Gamma \text{ е неизпълнимо} \iff \Gamma \vdash_r \square.$$

Доказателство. Използваме теоремата на Ербран: неизпълнимо множество от универсално затворени дизюнкти без равенство има крайно неизпълнимо множество от основни частни случаи. Ако езикът няма константа, добавяме нова, за да е непразен универсумът от основни термове. Обозначаваме полученото крайно множество с S.

За основни дизюнкти пълнотата се доказва чрез елиминиране на атом. Премахваме тавтологиите и избираме атом A. Запазваме дизюнктите без A и $\neg A$ и добавяме всички резолвенти между $A \vee C$ и $\neg A \vee D$. Ако новото множество има модел, но той не може да се разшири с никоя стойност на A, някой C и някой D са едновременно неверни. Тогава добавеният $C \vee D$ е неверен, противоречие. Значи неизпълнимостта

се запазва. Индукция по крайния брой атоми достига неизпълнимо множество без атоми, което съдържа празния дизюнкт; всички добавени дизюнкти имат резолютивен извод от S.
Повдигаме получения основен извод. Във всяка негова стъпка основната замяна унифицира всички литерали от родителите, които са се слели в елиминирания атом. Тя е частен случай на най-общ унификатор σ. Либералната резолюция с тези множества литерали дава дизюнкт, чийто основен частен случай е съответната основна резолвента. Преименуването на променливите прави замените на двата родителя съвместими. По индукция всяка стъпка се повдига; дизюнкт с празен основен частен случай сам е празен. Получаваме $\Gamma \vdash_r \square$. Обратната импликация е доказаната коректност.
Тук теоремата на Ербран е използван външен резултат. Конспектът изисква формулировка на пълнотата; тази редукция е допълнително доказателство с изрично посочена предпоставка. $\square$

При същинската резолюция пълнотата може да се изрази чрез резолютивното затваряне. Нека

$$R(S) = S \cup \{D \mid D \text{ е резолвента на два дизюнкта от } S\},$$

$$S_0 = S, \qquad S_{n+1} = R(S_n), \qquad S^* = \bigcup_{n \geq 0} S_n.$$

Тогава

$$S \vdash_r D \quad \iff \quad D \in S^*,$$

а в частност

$$S \text{ е неизпълнимо} \quad \iff \quad \square \in S^* \quad \iff \quad S \vdash_r \square.$$

За предикатния случай пълнотата е свързана с теоремата на Ербран. След сколемизация и преминаване към затворени универсални формули се разглеждат техните затворени частни случаи. Неизпълнимостта на първоредовото множество се свежда до булева неизпълнимост на подходящо крайно множество от такива случаи, за което резолюцията е пълна.

Ако равенството е част от логическия език със стандартната си семантика, обикновената резолюция сама не е достатъчна за тази формулировка на пълнотата. Необходими са подходящи аксиоми за равенство или специализирани правила като параметодулация/суперпозиция.

Забележка 22.24. Пълнотата не означава, че всяка стратегия за избор на двойки дизюнкти непременно ще намери празния дизюнкт за краен брой стъпки. Теоремата твърди, че при неизпълнимо множество съществува краен резолютивен извод на $\square$.

22.4 Компютърно генериране на доказателства

Резолюцията е особено подходяща за автоматизация, защото всяка стъпка има краен и проверим вид: посочват се два родителски дизюнкта, избрани литерали и унификатор. Така доказателството може да се пази като дърво или като линейна последователност от стъпки.

Общият процес има следните етапи:

1. Формулира се задачата като проверка на неизпълнимост, обикновено чрез добавяне на отрицанието на целта.
2. Отстраняват се импликациите и еквивалентностите.
3. Отрицанията се свеждат непосредствено пред атомарни формули.
4. Формулата се привежда в пренексна форма и се сколемизира, когато има квантори за съществуване.
5. Отпадат универсалните квантори и матрицата се привежда в КНФ.
6. Генерират се резолвенти чрез унификация.
7. При получаване на $\square$ се извежда неизпълнимостта на изходното множество.

▷ **Пример 22.25 (Предикатно доказване на неизпълнимост).** Нека са дадени формулите

$$\forall x \big(Student(x) \Rightarrow Learns(x)\big),$$

$$Student(ivan),$$

$$\neg Learns(ivan).$$

След преминаване към дизюнкти получаваме:

$$D_1 = \{\neg Student(x), Learns(x)\}, \qquad D_2 = \{Student(ivan)\}, \qquad D_3 = \{\neg Learns(ivan)\}.$$

Резолюцията на D_1 и D_2 използва унификатора

$$\sigma = \left\{\frac{x}{ivan}\right\}$$

и дава

$$D_4 = \{Learns(ivan)\}.$$

След това D_4 и D_3 дават

$$\square.$$

Следователно трите изходни формули са неизпълними като множество.

22.5 Хорнови дизюнкти и Пролог

Дефиниция 22.26. Хорнов дизюнкт е дизюнкт с най-много един положителен литерал.

Най-често се разглеждат следните три вида:

$$\{P\}$$

е факт;

$$\{P, \neg Q_1, \ldots, \neg Q_n\}$$

е правило, което се чете като

$$Q_1 \wedge \cdots \wedge Q_n \Rightarrow P;$$

$$\{\neg Q_1, \ldots, \neg Q_n\}$$

е цел, която съответства на въпроса дали могат да се докажат едновременно $Q_1, \ldots, Q_n$.

Хорновите правила и факти образуват хорнова програма. Резолюцията за хорнови дизюнкти е основата на логическото програмиране и на езика Пролог. Вместо да се генерират произволни резолвенти, обикновено се работи с цел и с правило, чиято глава се унифицира с избран подцелеви атом.

▷ **Пример 22.27 (Дефиниране на предикати на Пролог).** Следната програма описва предиката `predok(X,Y)`: „`X` е предшественик на `Y`“.

```
roditel(ivan, maria).
roditel(maria, petar).

predok(X, Y) :- roditel(X, Y).
predok(X, Y) :- roditel(X, Z), predok(Z, Y).
```

Фактът `roditel(ivan,maria).` съответства на едноелементния дизюнкт

$$\{roditel(ivan, maria)\}.$$

Правилото

```
predok(X, Y) :- roditel(X, Z), predok(Z, Y).
```

съответства на хорновия дизюнкт

$$\{predok(X, Y), \neg roditel(X, Z), \neg predok(Z, Y)\}.$$

▷ **Пример 22.28 (Резолютивно доказване на цел).** Нека целта е

```
?- predok(ivan, petar).
```

Тя се представя с отрицателния дизюнкт

$$\{\neg predok(ivan, petar)\}.$$

Резолюция с рекурсивното правило дава нова цел

$$\{\neg roditel(ivan, Z), \neg predok(Z, petar)\}.$$

Фактът $roditel(ivan, maria)$ унифицира Z с $maria$, откъдето получаваме

$$\{\neg predok(maria, petar)\}.$$

Резолюция с основното правило за $predok$ дава

$$\{\neg roditel(maria, petar)\}.$$

> Този литерал се резолвира с факта $roditel(maria, petar)$ и се получава $\square$. Следователно целта е следствие на програмата.

Забележка 22.29. В Пролог обикновено се използва SLD-резолюция: на всяка стъпка се избира подцел, унифицира се с главата на правило или факт и се заменя с тялото на правилото. Натрупаните унификатори дават отговора за променливите в заявката.

22.6 Изпитен фокус

- Да се дефинират: литерал, елементарна дизюнкция, КНФ, дизюнкт, верен дизюнкт в структура при оценка, тъждествено верен дизюнкт и изпълнимо множество от дизюнкти.
- Да се знае, че $\square$ е празният дизюнкт и е неверен при всяка интерпретация.
- Да се дефинира непосредствена резолвента:

$$R_L(D_1, D_2) = (D_1 \setminus \{L\}) \cup (D_2 \setminus \{\overline{L}\}).$$

- Да се дефинират субституция, унификатор и най-общ унификатор.
- Да се дефинира предикатна либерална резолвента чрез преименуване на променливите, унификация на противоположни атоми и прилагане на унификатора към остатъка от дизюнктите.
- Да се дефинира резолютивен извод и означението $\Gamma \vdash_r \delta$.
- Да се формулират теоремите за коректност и пълнота:

$$\Gamma \vdash_r \square \Rightarrow \Gamma \text{ е неизпълнимо}, \qquad \Gamma \text{ е неизпълнимо} \Rightarrow \Gamma \vdash_r \square.$$

  Пълнотата в този вид се отнася за първоредов език без формално равенство; при равенство са нужни допълнителни аксиоми или правила.
- Да се възпроизведе доказателството за коректност: индукция по стъпките на извода, използваща резолютивната лема; празният дизюнкт не може да бъде удовлетворен.
- Да се обясни ролята на Ербрановите частни случаи и унификацията при предикатната резолюция.
- Да се решава задача за превеждане на предикатни формули в дизюнкти и за извеждане на $\square$.
- Да се разпознават факти, правила и цели като хорнови дизюнкти и да се обясни връзката им с Пролог и SLD-резолюцията.

Забележка 22.30 (Проверка на допълнените доказателства). Допълнителен източник за правилата и предпоставките: Temple University: CIS587, Logic, Part 2.

Въпрос 23

Бази от данни. Релационен модел на данните.

По конспекта: какво трябва да мога

Дефиниции: домейн, релация, кортежи, атрибути, схема на релация, схема на релационна база от данни — §23.2, стр. 325.
Описание: видове операции върху релационната база от данни и заявки към нея — §23.3, стр. 327.
Релационна алгебра: основни операции — обединение, разлика, декартово произведение, проекция, селекция; допълнителни — сечение, съединение, естествено съединение — §23.4, стр. 328.
Типове задачи: съставяне на SQL заявки, DDL и DML команди, тригери — §23.5, стр. 334.
Литература: [32].

23.1 Бази от данни

23.1.1 Предназначение на базите от данни

Дефиниция 23.1 (База от данни)**.** База от данни е организирана колекция от данни, предназначена за съхраняване, обработване и извличане на информация. Данните се описват чрез определена структура, а операциите върху тях се изпълняват посредством специализиран език за работа с бази от данни.

Дефиниция 23.2 (Релационна база от данни)**.** Релационната база от данни представлява колекция от релации. В практическа реализация релациите обикновено се представят като таблици, но математическото им разглеждане се основава на множества от кортежи. Релационният модел дава възможност данните да бъдат описани чрез формална схема и да бъдат обработвани чрез операции от релационната алгебра или чрез заявки на SQL.

Проектирането на релационна база от данни включва следните основни стъпки:

1. определяне на данните, които ще се съхраняват;
2. определяне на връзките между обектите;
3. определяне на ключовете и свързващите колони;
4. определяне на ограниченията върху обектите и връзките между тях;
5. отстраняване на евентуални недостатъци и излишества;
6. реализиране на базата от данни.

Тези стъпки водят от описание на предметната област към конкретна структура от релации и атрибути.

23.2 Релационен модел на данните

23.2.1 Математическа основа

Дефиниция 23.3 (Релация и кортеж)**.** Релационният модел се основава на математическото понятие за n-членна релация. Една n-членна релация е множество от елементи, всеки от които се състои от n компоненти. Тези елементи се наричат n-торки или кортежи.

Чрез една релация се моделира даден клас от обекти, а всеки кортеж от релацията представя конкретен обект от този клас. Например релация за филми може да описва класа от всички филми, като всеки неин кортеж представя един конкретен филм.

Дефиниция 23.4. Домейн е именувано множество, разглеждано като множество от допустими стойности на дадена величина.

Домейнът определя какви стойности могат да се използват за даден атрибут. Например атрибутът `title` може да има домейн от низове, а атрибутът `year` може да има домейн от цели числа:

$$\texttt{title} : \texttt{string}, \qquad \texttt{year} : \texttt{integer}.$$

Дефиниция 23.5. В контекста на базите от данни релация е таблица, чиито редове са кортежи, а стойностите в една колона принадлежат на домейна, асоцииран със съответния атрибут.

Релацията е множество от уникални кортежи. Следователно в математическия релационен модел един и същ кортеж не се повтаря.

Дефиниция 23.6. Атрибут е име на колона в релацията, с което е свързан определен домейн.

Например релацията за филми може да има атрибути `title`, `year`, `length` и `filmType`. Съответните домейни могат да бъдат низове, цели числа и други прости типове.

Дефиниция 23.7. Кортеж е ред от релацията, съдържащ конкретна стойност за всеки атрибут.

Ако атрибутите са подредени като

$$(\texttt{title}, \texttt{year}, \texttt{length}, \texttt{filmType}),$$

примерен кортеж е

$$(\texttt{Star wars}, 1977, 124, \texttt{Color}).$$

Компонентите на всеки кортеж принадлежат на домейните на съответните атрибути. Кортежите следват предварително определената подредба на атрибутите от схемата на релацията.

Релационният модел изисква стойностите на компонентите да бъдат атомарни. Това означава, че във всяка позиция на кортежа се съдържа една проста стойност, например цяло число или низ. Сложни стойности като списъци и масиви не се разглеждат като един компонент на кортежа в основния релационен модел.

23.2.2 Схема и екземпляр на релация

Дефиниция 23.8. Схема на релация е името на релацията заедно с множеството от нейните атрибути.

Схемата се записва чрез името на релацията, последвано от имената на атрибутите в скоби:

$$\mathit{Movies}(\mathit{title}, \mathit{year}, \mathit{length}, \mathit{filmType}).$$

В математическото описание атрибутите образуват множество. При записването на схемата обаче се фиксира стандартна подредба, за да бъде еднозначно определена позицията на всеки компонент в кортежа.

Дефиниция 23.9. Екземпляр на релация е множеството от кортежи, които в даден момент се съдържат в релацията.

Схемата описва структурата на релацията, докато екземплярът описва нейното текущо съдържание. Схемата обикновено се променя по-рядко, а екземплярът може да се променя при добавяне, промяна или изтриване на данни.

Дефиниция 23.10. Кардиналност на екземпляр на релация е броят на кортежите в този екземпляр.

Ако релацията *Movies* съдържа 120 кортежа, нейната кардиналност за съответния момент е 120.

Дефиниция 23.11. Схема на релационна база от данни е съвкупността от схемите на всички релации в базата от данни.

Например една база от данни може да има следната схема:

$$\begin{gathered} \mathit{Movie}(\mathit{Title}, \mathit{Year}, \mathit{Length}), \\ \mathit{StarsIn}(\mathit{MovieTitle}, \mathit{MovieYear}, \mathit{StarName}). \end{gathered}$$

Подчертаването се използва, за да се означат атрибути, които в дадения пример участват в идентифицирането на кортежите. Така в *Movie* заглавието, годината и дължината са показани като определящи атрибути, а в *StarsIn* са показани *MovieTitle* и *StarName*. Връзката между двете релации се изразява чрез съответните атрибути за филм и година.

23.3 Операции върху релационна база от данни

Операциите върху релационната база от данни могат да бъдат разгледани според това дали описват структурата, обработват данните или управляват достъпа и запазването на промените.

23.3.1 DDL операции

Data Definition Language, или DDL, е подезик за описание и промяна на схемата на базата от данни. Основните DDL команди са `CREATE`, `ALTER` и `DROP`.

Пример за създаване на таблица е:

```
CREATE TABLE movies (
  id INTEGER PRIMARY KEY,
  title CHAR(50) NOT NULL,
  length INTEGER NULL
);
```

Тук се създава таблица `movies` с три колони. Колоната `id` е от тип `INTEGER` и е обявена като първичен ключ. Колоната `title` съдържа низ с дължина до 50 символа и е обявена като задължителна чрез `NOT NULL`. За `length` е разрешена празна стойност чрез `NULL`.

Промяна на схемата може да се извърши чрез `ALTER TABLE`:

```
ALTER TABLE movies ADD release_date DATE;
ALTER TABLE movies DROP COLUMN length;
```

Първата команда добавя атрибута `release_date`, а втората премахва атрибута `length`. Изтриването на цялата таблица се извършва чрез:

```
DROP TABLE movies;
```

Моментът, в който DDL промяната става постоянна, зависи от конкретната СУБД и от транзакционния режим. Някои системи изпълняват автоматично потвърждение, а други позволяват DDL операцията да бъде част от явна транзакция.

23.3.2 DML операции

Data Manipulation Language, или DML, е подезик за работа със съдържанието на релациите. Основните DML команди са `SELECT`, `INSERT`, `UPDATE` и `DELETE`.

Командата `SELECT` извлича данни:

```
SELECT title FROM movies
WHERE length IS NOT NULL AND length < 120
ORDER BY release_date DESC
```

Заявката извлича заглавията на филмите, чиято дължина не е празна и е по-малка от 120. Резултатът се подрежда по `release_date` в низходящ ред.

Добавяне на кортеж се извършва с `INSERT`:

```
INSERT INTO movies (id, title, length, release_date)
VALUES (1, 'Star Wars', 124, DATE '1984-04-09')
```

Промяна на вече съществуващи данни се извършва с `UPDATE`:

```
UPDATE movies
SET title = 'Star Wars'
WHERE title = 'Star wars'
```

Изтриване на кортежи се извършва чрез `DELETE`:

```
DELETE FROM movies
WHERE length < 120
```

При изключен автоматичен режим за потвърждение DML промените трябва да бъдат потвърдени с `COMMIT`, за да станат постоянни. При включен autocommit всяка инструкция се потвърждава автоматично; конкретното поведение зависи от СУБД и настройките на сесията.

23.3.3 Заявки към релационната база от данни

Заявка е описание на желан резултат, получен чрез обработване на една или повече релации. В SQL заявките най-често се използва командата `SELECT`, която може да включва избиране на колони, условие за подбор на редове и подреждане на резултата.

В релационната алгебра заявката се представя чрез алгебричен израз. Всеки оператор приема релация или релации и връща нова релация. По този начин сложните заявки могат да се изграждат чрез комбиниране на по-прости операции.

23.4 Релационна алгебра

Дефиниция 23.12. Релационната алгебра е междинен език за изчисляване на заявки, който задава правила за обработване на релации чрез алгебрични изрази.

Операциите в релационната алгебра се изпълняват върху релации. В ядрото на релационната алгебра релациите се разглеждат като множества от кортежи. Разширената релационна алгебра разглежда релациите като мултимножества от кортежи, при които могат да се срещат повторения.

Основните класове операции са:

- операции върху множества;
- операции за отсяване на части от релациите;
- операции за комбиниране на кортежи от две релации;
- операция за преименуване.

Основните операции са селекция, проекция, обединение, разлика, декартово произведение и преименуване. Сечение, тета-съединение и естествено съединение могат да се изразят чрез основните операции и поради това се разглеждат като допълнителни.

23.4.1 Обединение

Обединението на две релации R и S се означава с

$$R \cup S.$$

Операцията е приложима, когато релациите имат съвместими схеми: еднакъв брой атрибути, които си съответстват. Резултатът е релация със същата схема, съдържаща всички кортежи, които принадлежат на R или на S.

Обединението е бинарна операция, комутативна и асоциативна:

$$R \cup S = S \cup R,$$

$$(R \cup S) \cup T = R \cup (S \cup T).$$

Повтарящите се кортежи не се включват многократно, тъй като релациите в ядрото на релационната алгебра са множества.

23.4.2 Разлика

Разликата на R и S се означава с

$$R - S.$$

Както при обединението, схемите трябва да бъдат съвместими. Резултатът съдържа всички кортежи, които са в R, но не са в S:

$$R - S = \{t \mid t \in R \text{ и } t \notin S\}.$$

Разликата не е комутативна. В общия случай

$$R - S \neq S - R.$$

Редът на операндите има значение: първата релация определя множеството, от което се изваждат кортежи.

23.4.3 Сечение

Сечението на R и S се означава с

$$R \cap S.$$

Операцията се прилага върху релации със съвместими схеми и връща релация, съдържаща само кортежите, общи за двете релации:

$$R \cap S = \{t \mid t \in R \text{ и } t \in S\}.$$

Сечението е комутативно и асоциативно:

$$R \cap S = S \cap R,$$

$$(R \cap S) \cap T = R \cap (S \cap T).$$

То може да бъде изразено чрез основните операции:

$$R \cap S = R - (R - S).$$

23.4.4 Проекция

Проекцията е унарна операция, която избира определени атрибути от релация. Означава се с

$$\pi_{A_1, A_2, \ldots, A_k}(R),$$

където $A_1, A_2, \ldots, A_k$ е списък от атрибути.

Резултатът съдържа само посочените колони. Ако след премахването на останалите колони се получат еднакви кортежи, повторенията се премахват. Например

$$\pi_{\mathit{Title},\mathit{Year},\mathit{Length}}(\mathit{Movie})$$

избира трите посочени атрибута от релацията *Movie*.

Проекцията променя схемата, като намалява броя на атрибутите. Затова тя често се описва като вертикално отсяване на релацията.

23.4.5 Селекция

Селекцията, наричана още рестрикция, е унарна операция, която избира кортежите, удовлетворяващи определено условие. Означава се с

$$\sigma_C(R),$$

където C е предикат.

Предикатът може да сравнява два атрибута или атрибут с константа. Допустимите оператори са

$$=, \quad \neq, \quad <, \quad >, \quad \leq, \quad \geq .$$

Пример:

$$\sigma_{\mathit{Year}<1980}(\mathit{Movie}).$$

Резултатът съдържа само филмите, чиято година е по-малка от 1980. Схемата остава същата, но броят на кортежите може да намалее. Поради това селекцията се разглежда като хоризонтално отсяване на релацията.

Селекцията е комутативна при последователно прилагане:

$$\sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_2}(\sigma_{C_1}(R)).$$

23.4.6 Декартово произведение

Декартовото произведение на R и S се означава с

$$R \times S.$$

Резултатът съдържа всички двойки от кортежи, при които първият кортеж е от R, а вторият е от S. Схемата на резултата е обединение на схемите на R и S.

Ако R има m кортежа, а S има n кортежа, резултатът съдържа всички възможни комбинации между тях. При еднакви имена на атрибути се използва квалифицирано име от вида

имеНаРелация.имеНаАтрибут.

Например могат да се използват *Movie.Year* и *StarsIn.MovieYear*.

При различни имена на атрибутите декартовото произведение е комутативно и асоциативно до канонично преименуване и пренареждане на атрибутите. Ако схемите се разглеждат като наредени списъци, $R \times S$ и $S \times R$ не са буквално една и съща релация; те са изоморфни след размяна на съответните компоненти. При съвпадащи имена първо се използва квалифициране или явно преименуване.

23.4.7 Тета-съединение

Тета-съединението на R и S при условие C се означава с

$$R \bowtie_C S.$$

То се получава чрез декартово произведение, последвано от селекция:

$$R \bowtie_C S = \sigma_C(R \times S).$$

Резултатът има схема, получена чрез обединяване на схемите на R и S, и съдържа само онези двойки кортежи, които удовлетворяват условието C.

Ако условието включва само съвпадение между атрибути, тета-съединението се нарича еквисъединение. Например условието

$$\textit{Movie.Year} = \textit{StarsIn.MovieYear}$$

свързва кортежи, за които стойностите на съответните атрибути съвпадат.

23.4.8 Естествено съединение

Естественото съединение се означава с

$$R \bowtie S.$$

То свързва релациите по всички атрибути с еднакви имена. Резултатът съдържа кортежите от декартовото произведение, за които стойностите на едноименните атрибути съвпадат. Повтарящите се колони се премахват автоматично.

Естественото съединение може да бъде представено чрез тета-съединение и проекция:

$$R \bowtie S = \pi_L\left(\sigma_C(R \times S)\right),$$

където C е условието за равенство по едноименните атрибути, а L съдържа всички атрибути на R и онези атрибути на S, които не присъстват в R.

Ако общите атрибути са $A_1, \ldots, A_n$, условието е от вида

$$C = (R.A_1 = S.A_1) \text{ AND } \cdots \text{ AND } (R.A_n = S.A_n).$$

23.4.9 Преименуване

Преименуването се означава с оператора ρ. То променя името на релацията и имената на нейните атрибути, без да променя кортежите.

Записът

$$R_1 := \rho_{R_1(A_1,\ldots,A_n)}(R_2)$$

означава, че R_1 е релация с атрибути $A_1, \ldots, A_n$ и със същите кортежи като R_2.

Преименуването е полезно, когато една и съща релация участва повече от веднъж в алгебричен израз или когато имената на атрибутите трябва да бъдат разграничени.

23.4.10 Класификация и приоритет

Основните операции на релационната алгебра са:

- обединение;
- разлика;
- декартово произведение;
- проекция;
- селекция;
- преименуване.

Допълнителните операции са:

- сечение;
- тета-съединение;
- естествено съединение.

Те могат да се получат от основните операции чрез следните зависимости:

$$R \cap S = R - (R - S),$$

$$R \bowtie_C S = \sigma_C(R \times S),$$

$$R \bowtie S = \pi_L\left(\sigma_C(R \times S)\right).$$

Приоритетът на операторите в алгебричните изрази е:

1. унарни оператори – селекция, проекция и преименуване;

2. декартово произведение и съединение;
3. сечение;
4. обединение и разлика;
5. изрази в скоби.

При неоднозначност се използват скоби.

23.5 Примерни задачи

23.5.1 Съставяне на SQL заявки

Нека е дадена таблицата `movies` с колоните `id`, `title`, `length` и `release_date`.

Задача: да се изведат заглавията на филмите с ненулева дължина, по-малка от 120, подредени по дата на издаване в низходящ ред.

```
SELECT title FROM movies
WHERE length IS NOT NULL AND length < 120
ORDER BY release_date DESC
```

Задача: да се добави филм.

```
INSERT INTO movies (id, title, length, release_date)
VALUES (1, 'Star Wars', 124, DATE '1984-04-09')
```

Задача: да се промени заглавието на филма.

```
UPDATE movies
SET title = 'Star Wars'
WHERE title = 'Star wars'
```

Задача: да се изтрият филмите с дължина под 120.

```
DELETE FROM movies
WHERE length < 120
```

При изключен autocommit след промени чрез `INSERT`, `UPDATE` или `DELETE` се изпълнява:

```
COMMIT
```

23.5.2 Пример за DDL

```
CREATE TABLE movies (
  id INTEGER PRIMARY KEY,
  title CHAR(50) NOT NULL,
  length INTEGER NULL
);
```

Добавяне на атрибут:

```
ALTER TABLE movies ADD release_date DATE;
```

Премахване на атрибут:

```
ALTER TABLE movies DROP COLUMN length;
```

Изтриване на таблицата:

```
DROP TABLE movies;
```

23.5.3 Тригери

Тригер е правило, което изпълнява автоматично действие при определено събитие върху таблица или изглед. При задаването му се посочват:

- събитието — например `INSERT`, `UPDATE` или `DELETE`;
- моментът — `BEFORE`, `AFTER` или, когато СУБД го поддържа, `INSTEAD OF`;
- гранулярността — веднъж за цялата инструкция или по веднъж за всеки засегнат ред;
- условие и действие; при редов тригер старите и новите стойности обикновено се означават чрез `OLD` и `NEW`.

Синтаксисът е зависим от СУБД. Следва пример за PostgreSQL, който записва промяната на заглавие:

```
CREATE TABLE movie_title_log (
  movie_id INTEGER,
  old_title CHAR(50),
  new_title CHAR(50),
  changed_at TIMESTAMP
);

CREATE FUNCTION log_movie_title_change()
RETURNS TRIGGER AS $$
```

```
BEGIN
  INSERT INTO movie_title_log
    VALUES (OLD.id, OLD.title, NEW.title, CURRENT_TIMESTAMP);
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER movie_title_changed
AFTER UPDATE OF title ON movies
FOR EACH ROW
WHEN (OLD.title IS DISTINCT FROM NEW.title)
EXECUTE FUNCTION log_movie_title_change();
```

Тук събитието е промяна на `title`, моментът е след промяната, гранулярността е по ред, условието пропуска непроменени стойности, а действието добавя запис в журналната таблица. При друга СУБД същата логика може да има различен синтаксис.

23.6 Изпитен фокус

За устно изложение трябва да могат да бъдат възпроизведени:

- определенията за домейн, релация, атрибут, кортеж, схема и екземпляр;
- разликата между схема на релация и текущ екземпляр на релация;
- определението за схема на релационна база от данни;
- изискването компонентите на кортежите да бъдат атомарни;
- стъпките при проектиране и реализиране на релационна база от данни;
- предназначението на DDL и командите `CREATE`, `ALTER`, `DROP`;
- предназначението на DML и командите `SELECT`, `INSERT`, `UPDATE`, `DELETE`;
- ролята на `COMMIT` след DML операции;
- дефинициите и условията за приложимост на обединение, разлика и сечение;
- разликата между селекция и проекция;
- определението за декартово произведение;
- изразяването на тета-съединение чрез декартово произведение и селекция;
- особеността на естественото съединение да свързва по едноименни атрибути и да премахва повтарящите се колони;
- ролята на преименуването;
- класификацията на основни и допълнителни операции;

- зависимостите

$$R \cap S = R - (R - S), \qquad R \bowtie_C S = \sigma_C(R \times S);$$

- приоритетът на операторите в релационната алгебра;
- съставянето на примерни SQL заявки за извличане, добавяне, промяна и изтриване на данни;
- примерите за DDL операции и предназначението на тригерите.

Въпрос 24

Бази от данни. Нормални форми.

По конспекта: какво трябва да мога

Описание: проектиране на схемите; аномалии; ограничения и ключове — §24.1.1–§24.1.3, стр. 338.
Дефиниции и аксиоми: функционални зависимости — §24.1.4; аксиоми на Армстронг — §24.1.5, стр. 341.
Нормални форми: първа, втора, трета и нормална форма на Бойс–Код — §24.1.6–§24.1.9, стр. 343.
Многозначни зависимости: дефиниция — §24.1.10; аксиоми и правила — §24.1.11, стр. 346.
Четвърта нормална форма: §24.1.12; съединение без загуба — §24.1.13, стр. 348.
Тип задача: привеждане на схема с дадени функционални зависимости към зададена нормална форма — §24.1.15, методика в §24.1.16, стр. 350.

Статус на покритието: разработен пример има за привеждане по функционални зависимости; отделен решен пример за 4NF по многозначна зависимост липсва.
Литература: [32].

24.1 Бази от данни и нормални форми

24.1.1 Релационни бази от данни и проектиране на схеми

Релационната база от данни представя информацията чрез релации. Релацията може да се разглежда като таблица, чиито колони са атрибути, а редовете са кортежи. Схемата на релацията задава името на релацията и множеството от нейните атрибути, например

$$\text{БИБЛИОТЕКА}(\text{Биб\#}, \text{Адрес}, \text{Книга}, \text{Брой}).$$

Екземплярът на релацията е конкретното множество от кортежи, което се съхранява в даден момент.

Проектирането на схемите на релационните бази от данни има за цел информацията да бъде организирана така, че:

- всяка релация да описва логически свързана съвкупност от обекти или факти;

- данните да не се повтарят излишно;
- добавянето, промяната и изтриването на данни да не водят до загуба на информация или до противоречия;
- да могат да се налагат ограничения за целостност;
- връзките между релациите да бъдат ясно представени чрез ключове.

Нормализацията е процесът по структуриране на релационна база от данни в съответствие с поредица от нормални форми. Нормалните форми са условия върху релациите, при изпълнението на които се ограничават аномалиите от определен вид. Обикновено релация, която не удовлетворява дадена нормална форма, се декомпозира на по-малки релации, за които условията на формата са изпълнени.

24.1.2 Аномалии

Аномалиите са недостатъци на конкретна релационна схема. Те възникват най-често поради смесването в една релация на различни видове информация и поради повторението на едни и същи данни.

Разглеждаме схемата

$$\text{БИБЛИОТЕКА}(\text{Биб\#}, \text{Адрес}, \text{Книга}, \text{Брой}),$$

където:

- Биб# е номерът на библиотеката;
- Адрес е адресът на библиотеката;
- Книга е името на книгата;
- Брой е броят на екземплярите от книгата.

Една библиотека се определя еднозначно от номера си. За всяка книга, която се съхранява в дадена библиотека, се записва и адресът на библиотеката. Ключът на схемата е съставен от Биб# и Книга.

Аномалия от излишество. Адресът на една и съща библиотека се повтаря за всяка книга в нея. Ако библиотеката съдържа много книги, един и същ адрес се записва много пъти.

Аномалия при добавяне. Ако се създаде нова библиотека, но все още няма книга, която да бъде въведена за нея, адресът на библиотеката не може да бъде записан в тази релация без добавяне на несъществуваща или неизвестна книга.

Аномалия при обновяване. При промяна на адреса трябва да се променят всички кортежи, отнасящи се до библиотеката. Ако бъде променен само един от тях, в базата ще съществуват различни адреси за една и съща библиотека.

Аномалия при изтриване. Ако всички книги на дадена библиотека бъдат преместени временно на друго място и кортежите за тях бъдат изтрити, заедно с тях ще бъде изгубен и адресът на библиотеката. Така изтриването на информация за книгите води до загуба на независима информация за библиотеката.

Естественото решение е разделяне на схемата:

$$\text{БИБЛИОТЕКА}(\text{Биб\#}, \text{Адрес})$$

и

$$\text{НАЛИЧНОСТ}(\text{Биб\#}, \text{Книга}, \text{Брой}).$$

Тогава адресът се съхранява веднъж за всяка библиотека, а информацията за книгите се съхранява отделно.

24.1.3 Ограничения и ключове

Ограниченията дефинират правила за допустимите стойности на атрибутите и са стандартен механизъм за налагане на целостност.

Ограниченията за ключа изискват стойностите на ключа да бъдат уникални. В практическа релационна база това се реализира чрез ограничения като `UNIQUE` и `PRIMARY KEY`. Обикновено атрибутите на първичния ключ не могат да приемат стойност `NULL`, което се задава чрез `NOT NULL`.

Ограничението за референциална цялост гарантира, че стойност, сочена от елемент на базата, действително съществува. Външният ключ се задава чрез `FOREIGN KEY`. За всяка ненулева стойност на външния ключ трябва да съществува съответна ключова стойност в друга релация.

Ограниченията на домейна ограничават стойностите на даден атрибут до определено множество. В SQL например това може да се постигне чрез `CHECK`. Допълнителни общи ограничения също могат да бъдат част от схемата на базата.

★ **Дефиниция 24.1.** Нека R е релационна схема и $K = \{A_1, A_2, \ldots, A_n\}$ е множество от нейни атрибути. K е ключ за R, ако:

1. K функционално определя всички атрибути на R;
2. за нито едно собствено подмножество на K не е изпълнено първото условие.

Ако K удовлетворява първото условие, но не и второто, то K е суперключ.

Следователно всеки ключ е суперключ, но не всеки суперключ е ключ. Ключът е минимален суперключ. Когато ключът съдържа повече от един атрибут, той се нарича съставен ключ.

24.1.4 Функционални зависимости

Функционалната зависимост описва зависимост между атрибутите в рамките на една релация.

Дефиниция 24.2. Нека X и Y са множества от атрибути на релацията R. Казваме, че X функционално определя Y, и записваме

$$X \to Y,$$

ако за всеки два кортежа t и u от екземпляр на R, от

$$t[X] = u[X]$$

следва

$$t[Y] = u[Y].$$

Функционална зависимост се нарича такава, защото стойностите на X определят единствена стойност или единствен набор от стойности на Y. Най-основният пример е зависимостта от ключа към всички атрибути:

$$K \to R.$$

За схемата на библиотеката са естествени зависимостите

$$\text{Биб\#} \to \text{Адрес}$$

и

$$\text{Биб\#}, \text{Книга} \to \text{Брой}.$$

От тях чрез обединение следва

$$\text{Биб\#}, \text{Книга} \to \text{Адрес}, \text{Брой}.$$

Това показва, че двойката $(\text{Биб\#}, \text{Книга})$ определя всички атрибути на релацията.

Зависимостта $X \to Y$ е тривиална, ако $Y \subseteq X$. Например

$$\{\text{Име}, \text{Адрес}\} \to \text{Адрес}$$

е тривиална зависимост. Тя е нетривиална, ако поне един атрибут от Y не принадлежи на X. Зависимостта е напълно нетривиална, ако нито един атрибут от Y не принадлежи на X.

24.1.5 Аксиоми на Армстронг

Аксиомите на Армстронг са правила за извеждане на функционални зависимости. Те позволяват от дадено множество зависимости да се създават нови зависимости.

Теорема 24.3. За множества от атрибути X, Y, Z и W са валидни следните аксиоми:

1. *Рефлексивност:* ако $Y \subseteq X$, то $X \to Y$;
2. *Разширение:* ако $X \to Y$, то $XW \to YW$;
3. *Транзитивност:* ако $X \to Y$ и $Y \to Z$, то $X \to Z$.

Доказателство. Нека t, u са произволни кортежи на релацията. Ако съвпадат по X и $Y \subseteq X$, съвпадат и по Y: рефлексивност. Ако съвпадат по XW, съвпадат по X и по W; от $X \to Y$ съвпадат и по Y, значи по YW: разширение. Ако съвпадат по X, от $X \to Y$ съвпадат по Y, а от $Y \to Z$ — по Z: транзитивност. Това доказва семантичната коректност на трите правила. □

От тези аксиоми се извеждат следните правила.

Твърдение 24.4. *Обединение*: ако $X \to Y$ и $X \to Z$, то

$$X \to YZ.$$

Доказателство. Нека два кортежа съвпадат по X. От двете предпоставки те съвпадат по Y и по Z, следователно по обединението YZ. Това е определението за $X \to YZ$. □

Твърдение 24.5. *Псевдотранзитивност:* ако $X \to Y$ и $WY \to Z$, то

$$XW \to Z.$$

Доказателство. Два кортежа, съвпадащи по XW, съвпадат по X, оттам по Y по първата зависимост, и също по W. Значи съвпадат по WY, а втората зависимост дава съвпадане по Z. □

Твърдение 24.6. *Декомпозиция:* ако $X \to Y$ и $Z \subseteq Y$, то

$$X \to Z.$$

Доказателство. Ако два кортежа съвпадат по X, по предпоставката съвпадат по Y. Понеже $Z \subseteq Y$, съвпадат и по Z, тоест $X \to Z$. □

▷ **Пример 24.7.** Нека са дадени

$$\text{КодКурс} \to \text{ИмеКурс}$$

и

$$\text{Студент}, \text{КодКурс} \to \text{Оценка}.$$

Чрез разширение на първата зависимост с Студент получаваме

$$\text{Студент}, \text{КодКурс} \to \text{Студент}, \text{ИмеКурс}.$$

Чрез декомпозиция на получената зависимост може да се изведе

$$\text{Студент}, \text{КодКурс} \to \text{ИмеКурс}.$$

Ако е дадена и зависимостта

$$\text{ИмеКурс} \to \text{Катедра},$$

тогава транзитивността дава

$$\text{Студент}, \text{КодКурс} \to \text{Катедра}.$$

24.1.6 Първа нормална форма

Дефиниция 24.8. Релационната схема R е в първа нормална форма, ако всеки компонент на всеки кортеж съдържа атомарна стойност. Екземплярите на атрибутите принадлежат на домейни от неделими стойности.

Таблица, която действително се разглежда като релация, е в първа нормална форма. Нарушение би имало, ако в една клетка се съхранява списък от книги, множество от адреси или друга съставна структура вместо една атомарна стойност.

24.1.7 Втора нормална форма

Дефиниция 24.9. Релацията R е във втора нормална форма, ако е в първа нормална форма и всеки неключов атрибут е в пълна функционална зависимост от ключа. Това означава, че атрибутът зависи от целия ключ, а не от собствено подмножество на ключа.

Проблемът при втората нормална форма възниква при съставен ключ. Ако ключът е едноатрибутен, собствено нетривиално подмножество на ключа няма и условието за пълна зависимост се изпълнява непосредствено.

В схемата

$$\text{БИБЛИОТЕКА}(\text{Биб\#}, \text{Адрес}, \text{Книга}, \text{Брой})$$

ключът е $(\text{Биб\#}, \text{Книга})$, но

$$\text{Биб\#} \to \text{Адрес}.$$

Следователно Адрес зависи само от част от ключа. Това е частична функционална зависимост и схемата не е във втора нормална форма.

Декомпозираме:

$$\text{БИБЛИОТЕКА}(\text{Биб\#}, \text{Адрес})$$

и

$$\text{НАЛИЧНОСТ}(\text{Биб\#}, \text{Книга}, \text{Брой}).$$

В първата релация Биб# е ключ. Във втората $(\text{Биб\#}, \text{Книга})$ е ключ и Брой зависи от целия ключ.

24.1.8 Трета нормална форма

Дефиниция 24.10 (Първичен и неключов атрибут)**.** Атрибут, който е част от някой ключ, се нарича първичен атрибут. Атрибут, който не е част от ключ, се нарича неключов атрибут.

Дефиниция 24.11. Релацията R е в трета нормална форма, ако за всяка нетривиална функционална зависимост

$$X \to A$$

в R е изпълнено поне едно от следните условия:

1. X е суперключ на R;
2. A е първичен атрибут, тоест принадлежи на някой ключ.

Условието за трета нормална форма забранява зависимост, при която неключов атрибут определя друг неключов атрибут, освен ако лявата страна не е суперключ. Така се предотвратяват транзитивни зависимости на неключови атрибути от ключа.

▷ **Пример 24.12.** Нека е дадена схемата

$$R(\text{Студент\#}, \text{Курс\#}, \text{ИмеКурс}, \text{Катедра})$$

със зависимости

$$\text{Студент\#}, \text{Курс\#} \to \text{ИмеКурс}, \text{Катедра}$$

и

$$\text{Курс\#} \to \text{ИмеКурс}, \text{Катедра}.$$

Ключът е $(\text{Студент\#}, \text{Курс\#})$. Зависимостта

$$\text{Курс\#} \to \text{ИмеКурс}$$

има лява страна, която не е суперключ, а ИмеКурс не е първичен атрибут. Следователно схемата не е във втора нормална форма, а следователно не е и в трета нормална форма.
Подходящо разделяне е

$$\text{КУРС}(\text{Курс\#}, \text{ИмеКурс}, \text{Катедра})$$

и

$$\text{ЗАПИСВАНЕ}(\text{Студент\#}, \text{Курс\#}).$$

24.1.9 Нормална форма на Бойс–Код

Дефиниция 24.13. Релацията R е в нормална форма на Бойс–Код, ако за всяка нетривиална функционална зависимост

$$X \to Y$$

в R множеството X е суперключ на R.

BCNF е по-строго условие от третата нормална форма. Всяка релация в BCNF удовлетворява условието на 3НФ, но условието на 3НФ допуска и втория случай: дясната страна да бъде първичен атрибут.

При проверка за BCNF се разглежда всяка нетривиална функционална зависимост. Ако съществува зависимост $X \to Y$, за която X не е суперключ, релацията не е в BCNF.

24.1.10 Многозначни зависимости

Функционалните зависимости не описват всички видове независимост между атрибути. Възможно е всички атрибути на една релация да участват в ключ, така че релацията да е в BCNF, но въпреки това да съдържа излишество. Тогава се използват многозначни зависимости.

Дефиниция 24.14. Нека X, Y и Z са множества от атрибути на R, като Z съдържа останалите атрибути:

$$Z = R \setminus (X \cup Y).$$

Многозначната зависимост

$$X \twoheadrightarrow Y$$

е изпълнена в R, ако за всеки два кортежа t и u от R, за които

$$t[X] = u[X],$$

съществува кортеж v от R, такъв че

$$v[X] = t[X] = u[X],$$

$$v[Y] = t[Y],$$

и

$$v[Z] = u[Z].$$

Интуитивно, ако два кортежа съвпадат по X, компонентите им по Y могат да бъдат разменени с компонентите по останалите атрибути, като получените кортежи също принадлежат на релацията.

Многозначната зависимост изразява независимост между две групи атрибути. Например при схема

$$R(\text{Лице}, \text{Език}, \text{Хоби})$$

зависимостта

$$\text{Лице} \twoheadrightarrow \text{Език}$$

означава, че множеството от езици, които лицето знае, е независимо от множеството от неговите хобита. Ако лицето знае два езика и има две хобита, релацията трябва да съдържа съответните комбинации.

24.1.11 Аксиоми и правила за многозначни зависимости

Дефиниция 24.15 (Тривиална многозначна зависимост)**.** Многозначната зависимост е тривиална, ако

$$Y \subseteq X$$

или ако

$$X \cup Y = R.$$

В останалите случаи тя е нетривиална. Нетривиалността означава, че Y не се съдържа в X и че $X \cup Y$ не съдържа всички атрибути на релацията.

За многозначните зависимости се използват следните правила:

Твърдение 24.16. *Транзитивно правило:* ако

$$X \twoheadrightarrow Y$$

и

$$Y \twoheadrightarrow Z,$$

то в общия случай

$$X \twoheadrightarrow Z \setminus Y.$$

Доказателство. Нека t, u съвпадат по X и нека $W = Z \setminus Y$. От $X \twoheadrightarrow Y$ получаваме кортеж v, който взема Y от u, а останалите атрибути от t (по X стойностите съвпадат). Сега v и u съвпадат по Y. От $Y \twoheadrightarrow Z$ получаваме w, който взема Z от v, а останалите атрибути от u. По $Z \setminus Y$ той съвпада с t; по $Z \cap Y$ съвпада с u, а извън Z също съвпада с u. По X няма конфликт, понеже $t[X] = u[X]$. Значи w взема W от t и допълнението извън XW от u, което доказва $X \twoheadrightarrow W$. □

Изваждането на Y е съществено: за разлика от функционалните зависимости, от тези две предпоставки не следва непременно $X \twoheadrightarrow Z$.

Твърдение 24.17. *Правило на допълнението:* ако

$$X \twoheadrightarrow Y,$$

то

$$X \twoheadrightarrow Z,$$

където

$$Z = R \setminus (X \cup Y).$$

Доказателство. За два кортежа t, u с еднакви стойности по X прилагаме $X \twoheadrightarrow Y$ в обратния ред (u, t). Полученият кортеж взема Y от u и $Z = R \setminus (X \cup Y)$ от t. Той е точно кортежът, изискван от определението за $X \twoheadrightarrow Z$. По X двата кортежа съвпадат. □

Твърдение 24.18. *Правило на обединението:* ако

$$X \twoheadrightarrow Y$$

и

$$X \twoheadrightarrow Z,$$

то

$$X \twoheadrightarrow Y \cup Z.$$

Доказателство. Нека t, u съвпадат по X. От $X \twoheadrightarrow Y$ построяваме v, който взема Y от t и останалите атрибути от u. Кортежите t, v съвпадат по X. Прилагаме $X \twoheadrightarrow Z$ към тях и получаваме w, който взема Z от t и останалите атрибути от v. Така w съвпада с t по $Y \cup Z$, а с u извън $X \cup Y \cup Z$. Това е необходимият кортеж за $X \twoheadrightarrow Y \cup Z$. □

Твърдение 24.19. *FD-IS-AN-MVD:* всяка функционална зависимост е и многозначна зависимост:

$$X \to Y \quad \Longrightarrow \quad X \twoheadrightarrow Y.$$

Доказателство. Нека t и u са два кортежа, които съвпадат по X. От функционалната зависимост $X \to Y$ следва, че те съвпадат и по Y. За определението на многозначната зависимост може да се избере $v = u$. Тогава v съвпада с t по X, има по Y същите стойности като t, защото $t[Y] = u[Y]$, и съвпада с u по всички останали атрибути, тъй като $v = u$. Следователно е изпълнено $X \twoheadrightarrow Y$. □

24.1.12 Четвърта нормална форма

BCNF разглежда само функционалните зависимости. За да се ограничат излишествата, породени от независими многозначни зависимости, се въвежда четвъртата нормална форма.

Дефиниция 24.20. Релацията R е в четвърта нормална форма, ако за всяка нетривиална многозначна зависимост

$$X \twoheadrightarrow Y$$

в R множеството X е суперключ на R.

Ако в схемата

$$R(\text{Лице}, \text{Език}, \text{Хоби})$$

е изпълнена нетривиалната зависимост

$$\text{Лице} \twoheadrightarrow \text{Език},$$

но Лице не е суперключ, схемата не е в 4НФ. Декомпозицията е

$$R_1(\text{Лице}, \text{Език})$$

и

$$R_2(\text{Лице}, \text{Хоби}).$$

Така независимите множества от езици и хобита се представят в отделни релации и не се образуват излишни комбинации в една обща таблица.

24.1.13 Съединение без загуба

При декомпозиция на релация е необходимо да се провери дали информацията може да бъде възстановена чрез съединение. Декомпозицията трябва да запази смисъла на първоначалните данни, без съединението да поражда несъществуващи кортежи или да губи съществуващи кортежи.

Дефиниция 24.21 (Декомпозиция без загуба)**.** Нека релационната схема R бъде декомпозирана на схемите S и T, като $S \cup T = R$, и нека F е множеството от функционални зависимости. Декомпозицията е *без загуба относно* F, ако за всеки допустим екземпляр r на R, който удовлетворява F, е изпълнено

$$r = \pi_S(r) \bowtie \pi_T(r).$$

Равенството означава, че естественото съединение възстановява точно първоначалните кортежи и не поражда паразитни комбинации.

За бинарна декомпозиция условието е еквивалентно на поне едно от следните включвания във F^+:

$$(S \cap T) \to (S \setminus T) \qquad \text{или} \qquad (S \cap T) \to (T \setminus S).$$

Еквивалентно, общите атрибути трябва функционално да определят всички атрибути на поне една от двете схеми.

24.1.14 Запазване на функционалните зависимости

Нека F_S и F_T са проекциите на F^+ съответно върху S и T, тоест всички зависимости, логически следващи от F, чиито атрибути са изцяло в съответната схема. Декомпозицията *запазва зависимостите*, ако

$$(F_S \cup F_T)^+ = F^+.$$

Тогава първоначалните ограничения могат да бъдат проверявани чрез ограниченията върху отделните релации, без предварително съединение. Това свойство е различно от съединението без загуба.

Следователно при оценка на декомпозиция трябва да се разграничават:

- запазване на зависимостите: ограниченията могат да се проверяват в отделните релации;
- съединение без загуба: първоначалната информация се възстановява точно чрез съединение;
- достигане на определена нормална форма.

24.1.15 Примерна задача за нормализация

▷ **Пример 24.22.** Дадена е релационната схема

$$R(\text{Биб\#}, \text{Адрес}, \text{Книга}, \text{Брой})$$

с функционални зависимости

$$\text{Биб\#} \to \text{Адрес}$$

и

$$\text{Биб\#}, \text{Книга} \to \text{Брой}.$$

Да се приведе схемата към трета нормална форма.

Стъпка 1: намиране на ключ. От дадените зависимости:

$$\text{Биб\#}, \text{Книга} \to \text{Адрес}$$

чрез разширение или чрез факта, че $\text{Биб\#} \to \text{Адрес}$, а също

$$\text{Биб\#}, \text{Книга} \to \text{Брой}.$$

Следователно

$$\text{Биб\#}, \text{Книга} \to \text{Адрес}, \text{Брой}.$$

Двойката $(\text{Биб\#}, \text{Книга})$ е суперключ. Нито Биб#, нито Книга самостоятелно определят всички атрибути според зададените зависимости, така че двойката е ключ.

Стъпка 2: проверка за 2НФ. Атрибутът Адрес е неключов, но

$$\text{Биб\#} \to \text{Адрес}.$$

Следователно той зависи от собствено подмножество на съставния ключ. Релацията не е във 2НФ.

Стъпка 3: декомпозиция. Създаваме:

$$R_1(\text{Биб\#}, \text{Адрес})$$

и

$$R_2(\text{Биб\#}, \text{Книга}, \text{Брой}).$$

В R_1 зависимостта е

$$\text{Биб\#} \to \text{Адрес},$$

като Биб# е ключ. В R_2 зависимостта е

$$\text{Биб\#}, \text{Книга} \to \text{Брой},$$

като лявата страна е ключ. Следователно премахнатата частична зависимост не съществува в новите релации.

Стъпка 4: проверка за 3НФ и BCNF. В R_1 лявата страна на единствената нетривиална зависимост е суперключ. В R_2 лявата страна на зависимостта също е суперключ. Следователно двете релации удовлетворяват условието за BCNF, а оттам и условието за 3НФ.

24.1.16 Методика за решаване на задачи

При задача за привеждане на схема към зададена нормална форма може да се следва следната последователност:

1. Записва се схемата R и се отделят всички зададени функционални и многозначни зависимости.
2. Намират се суперключовете и минималните ключове.
3. Определят се първичните атрибути и неключовите атрибути.
4. Проверява се 1НФ чрез атомарността на стойностите.
5. При проверка за 2НФ се търсят зависимости на неключови атрибути от собствено подмножество на съставен ключ.
6. При проверка за 3НФ за всяка нетривиална функционална зависимост $X \to A$ се проверява дали X е суперключ или A е първичен атрибут.
7. При проверка за BCNF се изисква всяка нетривиална функционална зависимост да има суперключ за лява страна.
8. При проверка за 4НФ се разглеждат и нетривиалните многозначни зависимости; тяхната лява страна трябва да бъде суперключ.
9. При декомпозиция се описват новите релации, ключовете и външните ключове.
10. Проверява се дали зависимостите са запазени и дали съединението възстановява първоначалната информация без загуба.

24.2 Изпитен фокус

За устен отговор трябва да могат да бъдат възпроизведени:

- предназначението на нормализацията и видовете аномалии;
- разликата между ключ и суперключ;
- ограниченията за ключ, домейн и референциална цялост;
- дефиницията на функционална зависимост;
- трите аксиоми на Армстронг и правилата за обединение, псевдотранзитивност и декомпозиция;
- определенията за 1НФ, 2НФ, 3НФ и BCNF;

- разликата между функционална и многозначна зависимост;
- дефиницията и правилата за многозначни зависимости;
- твърдението, че всяка функционална зависимост е многозначна зависимост;
- определението за 4НФ;
- разликата между запазване на зависимости и съединение без загуба;
- последователността за намиране на ключове, проверка на нормална форма и декомпозиция на схема.

Забележка 24.23 (Проверка на допълнените доказателства). Допълнителен източник за правилата и предпоставките: Database System Concepts, глава 28.

Въпрос 25

Изкуствен интелект. Пространство на състоянията.

По конспекта: какво трябва да мога

Определение: пространство на състоянията; характеристики на състоянията — §25.1.1, стр. 352.
Описание: видове задачи и търсене в пространство на състояния — §25.1.2, стр. 354.
Локални търсещи алгоритми: локално търсене в лъч, изкачване на хълмове, симулирано втвърдяване — §25.1.3, стр. 355.
Алгоритми: генетични алгоритми — §25.1.4, стр. 357.
Задачи за удовлетворяване на ограничения: определение, търсене с възврат, намаляване на конфликтите — §25.1.5, стр. 359.
Избор на ход при игра за двама играчи: мини-макс процедура и алфа-бета отсичане — §25.1.6, стр. 361.

Статус на покритието: присъствието на назования материал е проверено; коректността на отделните алгоритми не е самостоятелно сертифицирана в тази редакция.
Литература: [41].

25.1 Изкуствен интелект и пространство на състоянията

25.1.1 Основни понятия

При решаването на задача чрез методи на изкуствения интелект е необходимо задачата да бъде описана по начин, който позволява постепенно построяване на решение. Подходящ модел за това е пространството на състоянията.

Дефиниция 25.1 (Състояние)**.** Състояние е представяне на задачата в даден момент от процеса на нейното решаване. То описва цялата информация, необходима за определяне на възможните следващи действия.

Различават се начални, междинни и крайни състояния. Началното състояние описва изходната конфигурация на задачата. Междинните състояния се получават в процеса на решаване. Крайно, или целево, е състояние, което удовлетворява поставената цел.

Дефиниция 25.2 (Оператор)**.** Оператор е правило, действие или функция, чрез което от едно състояние се получава друго състояние.

Операторите трябва да бъдат приложими само когато са изпълнени съответните условия за приложимост. Например при задача за преместване на плочка оператор може да бъде преместване на плочка в празна съседна позиция. При задача за намиране на път операторите могат да бъдат дъгите, по които се преминава между върхове на граф.

Дефиниция 25.3 (Пространство на състоянията)**.** Пространството на състоянията е съвкупността от всички състояния, които могат да бъдат получени от дадено начално състояние чрез последователно прилагане на допустими оператори.

Пространството на състоянията може да бъде представено чрез ориентиран граф. Върховете на графа са състоянията, а ориентираните дъги съответстват на операторите, които превръщат едно състояние в друго. Когато всеки възел се разглежда заедно с отделни копия на своите наследници, представянето има вид на дърво на търсенето. В графа едно и също състояние може да бъде достигнато по различни пътища, докато в дървото на търсенето могат да се появят няколко възела, описващи едно и също състояние.

Дефиниция 25.4 (Решение)**.** Решение е последователност от действия, която води от началното състояние до крайно или целево състояние.

Еквивалентно, решението може да се представи като път в пространството на състоянията. То може да бъде записано като списък от състояния или като списък от оператори, приложени последователно.

Основните действия при работа с пространство на състоянията са:

- генериране на състоянията, тоест построяване на следващи наследници или на цялото пространство;
- оценяване на състоянията чрез булева функция, която определя дали е изпълнена целта, или чрез числова оценка;
- търсене на път от началното състояние до целево състояние;
- сравняване на решенията, например по тяхната дължина или обща цена.

Дефиниция 25.5 (Стратегия за търсене)**.** Стратегията за търсене е правило за избор на реда, в който се разширяват възлите на дървото или графа на търсенето.

Качеството на стратегията се оценява чрез няколко показателя:

Пълнота. Стратегията е пълна, ако винаги намира решение, когато такова съществува.

Времева сложност. Определя се чрез броя на генерираните или разширените възли.

Пространствена сложност. Определя се чрез максималния брой възли, съхранявани в паметта.

Оптималност. Стратегията е оптимална, ако винаги намира решение с най-малка цена.

Нека d е дълбочината на най-плиткото целево състояние, m е максималната дълбочина на пространството или дървото, а b е коефициентът на разклонение, тоест броят на наследниците, които един възел може да има. Тези параметри се използват при оценяване на сложността на алгоритмите.

25.1.2 Видове задачи и търсене

Основните типове задачи, свързани с пространството на състоянията, са:

1. генериране на пространството на състоянията;
2. решаване на задача върху вече генерирано пространство;
3. комбинирано генериране и търсене, при което само необходимите части от пространството се построяват по време на търсенето.

Обикновено под „търсене“ се разбира едновременно избор на следващи състояния и, ако е необходимо, тяхното генериране.

Най-често се търси път от начално състояние до определено целево състояние или до някое от множество целеви състояния. Вариант на тази задача е намирането на път с минимална цена.

Според използваната информация методите за търсене се делят на неинформирани и информирани.

Неинформирано търсене. При неинформираното, или сляпо, търсене се използва само информацията, зададена при формулировката на проблема. Примери са търсене в дълбочина, търсене в ширина, лимитирано търсене в дълбочина и итеративно задълбочаващо се търсене.

Търсенето в дълбочина използва стек и избира най-дълбокия неразширен възел. Търсенето в ширина използва опашка и разглежда пространството ниво по ниво. При търсене с равномерен напредък се използва приоритетна опашка, като се избира възелът с най-малка натрупана цена $g(n)$.

Информирано търсене. Информираните методи използват допълнителна информация за целевото състояние. Тази информация се представя чрез евристична функция.

Дефиниция 25.6 (Евристична функция)**.** Евристична функция $h(n)$ е оценка на това колко е близо състоянието или възелът n до целта, обикновено чрез приблизителната цена за достигане на целево състояние.

Евристиката не описва непременно точния път до целта. Тя се използва за подреждане на възлите според тяхната перспективност. Примери за информирани методи са алчно търсене в най-добрия случай, търсене в лъч, изкачване на хълмове и A^*.

Според обхвата на разглежданото пространство методите се делят на глобално и локално търсещи. Глобално търсещите методи поддържат информация за значителна част от

пространството и могат да сравняват различни открити пътища. Локално търсещите методи разглеждат само текущото състояние и неговата локална област. Ако решението се намира извън разглежданата област, локалният метод може да не го открие.

25.1.3 Информирани методи за търсене

Търсене в най-добрия случай

Дефиниция 25.7 (Търсене с избор на най-перспективния връх)**.** Търсенето в най-добрия случай, наричано още *best-first search*, използва оценяваща функция, чрез която определя желателността на всеки възел. На всяка стъпка се разширява най-перспективният неразширен възел.

Обикновено се поддържа приоритетна опашка от възли, сортирани според оценката им. Ако се използва само евристичната функция, оценката е

$$f(n) = h(n).$$

Алгоритъмът е ефективен, когато евристиката добре отразява близостта до целта. Той обаче не е нито пълен, нито оптимален в общия случай. В най-лошия случай времевата и пространствената сложност са от експоненциален порядък, обикновено означавани с $O(b^m)$, тъй като могат да бъдат достигнати и съхранявани всички състояния до дълбочина m.

Локално търсене в лъч

Дефиниция 25.8 (Търсене в лъч)**.** Търсенето в лъч, или *beam search*, е вариант на търсенето в ширина, при който на всяко ниво се запазват само фиксиран брой най-перспективни възли.

Нека ширината на лъча е n. След разширяването на възлите от текущото ниво всички получени наследници се оценяват и се запазват само първите n според евристичната функция. Останалите възли се отстраняват от търсенето.

Типична схема е:

1. създава се началният лъч;
2. разширяват се всички възли от текущия лъч;
3. наследниците се оценяват чрез $h(n)$;
4. запазват се n-те най-добри наследници;
5. процедурата се повтаря до намиране на цел или до изчерпване на възможностите.

Ограничаването на броя съхранявани възли намалява разхода на памет, но може да отстрани възел, който е необходим за решение. Затова методът не е нито пълен, нито оптимален. В източниците времевата сложност е дадена като $O(bmn)$, като зависи от разглежданото изрязано пространство и от евристиката, а пространствената сложност е $O(bn)$.

Изкачване на хълмове

Дефиниция 25.9 (Изкачване на хълмове)**.** Изкачването на хълмове, или *hill climbing*, е локален метод, който на всяка стъпка преминава към най-добрия наследник на текущото състояние, но само ако наследникът е по-добър от текущото състояние.

При минимизираща евристика се избира наследник с по-малка стойност на h, а при максимизираща оценка се избира наследник с по-голяма стойност. Методът не поддържа алтернативни пътища и няма възможност за възврат. Следователно той използва много малко памет, но може да прекрати търсенето, без да е открил целта.

Описанието на основния цикъл е:

1. избира се текущо състояние;
2. генерират се неговите наследници;
3. намира се най-добрият наследник според евристиката;
4. ако той е по-добър от текущото състояние, става ново текущо състояние;
5. в противен случай търсенето спира.

Методът не е нито пълен, нито оптимален. В източниците времевата му сложност е дадена като $O(bm)$, а пространствената като $O(b)$.

Възможни са следните проблемни ситуации:

Локален екстремум. Текущото състояние е по-добро от непосредствените си наследници, но не е най-доброто състояние в цялото пространство.

Плато. Текущото състояние и наследниците му имат еднакви или практически неразличими оценки. Няма ясно направление за подобрение.

Хребет. Нито един отделен оператор не води до по-добро състояние, макар че комбинация от два или повече оператора би могла да доведе до такова състояние.

Забележка 25.10. Търсенето в лъч запазва няколко алтернативи, докато изкачването на хълмове запазва само една текуща алтернатива. Това обяснява по-малката памет на втория метод и по-големия риск от блокиране в локална област.

Симулирано втвърдяване

Дефиниция 25.11 (Симулирано втвърдяване)**.** Симулираното втвърдяване е стохастичен локален метод, който винаги може да приеме подобряващ ход, а влошаващ ход приема с вероятност, намаляваща с времето.

При задача за минимизация нека промяната на оценката е

$$\Delta = E(\text{ново}) - E(\text{текущо}).$$

Ако $\Delta \leq 0$, новото състояние се приема. Ако $\Delta > 0$, то се приема с вероятност

$$\exp\left(-\frac{\Delta}{T}\right),$$

където $T > 0$ е температура. В началото високата температура позволява чести излизания от локални минимуми. По схема за охлаждане T постепенно намалява и поведението се доближава до изкачване на хълмове.

Основният цикъл избира случаен съсед, прилага правилото за приемане и намалява температурата, докато се изпълни условие за край. Твърде бързото охлаждане лесно блокира метода, а твърде бавното изисква много време. Теоретични гаранции за глобален оптимум съществуват само при специални, много бавни схеми; произволна практическа схема не гарантира нито пълнота, нито оптималност.

25.1.4 Генетични алгоритми

Дефиниция 25.12 (Генетичен алгоритъм)**.** Генетичният алгоритъм е вариант на стохастично търсене в лъч, при който новите състояния се получават чрез комбиниране на двойки родителски състояния и чрез случайни мутации.

Генетичните алгоритми са локални методи, които имитират еволюционен процес. Вместо едно текущо състояние се поддържа популация от възможни решения.

Дефиниция 25.13 (Хромозома, ген и функция за пригодност)**.** Състоянията се представят като низове над крайна азбука. Такъв низ се нарича хромозома, а отделните му позиции или символи се разглеждат като гени. За всяка хромозома се изчислява оценяваща функция, наречена *fitness function*. Тя измерва пригодността на състоянието, тоест неговата близост до целта или степента, в която удовлетворява поставените изисквания. По-голямата стойност на fitness обикновено означава по-добро състояние.

Алгоритъмът започва с популация от k случайно генерирани състояния. След това поколенията се получават чрез селекция, кръстосване и мутация.

1. Генерира се начална популация.
2. Оценява се пригодността на всички индивиди.
3. Избират се родители.

4. От родителите чрез кръстосване се създават наследници.
5. Прилага се мутация върху малка част от новите индивиди.
6. Получава се нова популация.
7. Проверява се условието за край; ако то не е изпълнено, процесът продължава с оценяване на новата популация.

Дефиниция 25.14 (Селекция)**.** Селекция е етапът, при който се избират представители на популацията, които ще бъдат използвани като родители при създаването на следващото поколение.

Дефиниция 25.15 (Кръстосване)**.** Кръстосването е генетичен оператор, при който от две или повече родителски хромозоми се създава хромозома-наследник.

Нека дължината на хромозомите е L. Използват се различни типове кръстосване.

Едноточково кръстосване. Избира се случайна позиция между 1 и $L-1$. Хромозомите се разделят на тази позиция, след което се разменят вторите им половини. Така наследникът получава начална част от единия родител и крайна част от другия.

Двуточково, или order, кръстосване. Избират се две точки. Поднизът между точките от първия родител се копира в наследника. Останалите позиции се запълват чрез символи от втория родител, като се спазва техният ред, започвайки след изрязания интервал. Този вариант е подходящ за представяния, при които редът на гените е важен.

Циклично кръстосване. При цикличното кръстосване се идентифицират цикли между позициите на двата родителски хромозома. За първия наследник първият цикъл се копира от първия родител, вторият цикъл от втория родител, третият отново от първия и така нататък. Вторият наследник се получава със сменена роля на родителите.

Например за родителите

$$P_1 = (8,4,7,3,6,2,5,1,9,0), \qquad P_2 = (0,1,2,3,4,5,6,7,8,9)$$

се получават цикли, съдържащи съответно стойностите $(8,9,0)$ и $(4,1,7,2,5,6)$, като стойността 3 образува отделен цикъл. Един от наследниците може да бъде

$$D_1 = (8,1,2,3,4,5,6,7,9,0),$$

а другият

$$D_2 = (0,4,7,3,6,2,5,1,8,9).$$

Partially mapped crossover. При този вариант се избира интервал от хромозомата. Поднизът от интервала на първия родител се поставя в съответния интервал на втория родител. След това се определя съпоставяне между символите в двата интервала. Съпоставянето се използва за коригиране на останалите позиции така, че наследникът да спазва необходимите инварианти, например всички символи да бъдат различни.

Равномерно кръстосване. За всяка позиция се избира символ от един от родителите на случаен принцип. Така отделните позиции на наследника могат независимо да произхождат от различни родители.

Дефиниция 25.16 (Мутация)**.** Мутация е случайна промяна в малка част от новата популация.

Мутацията осигурява възможност за достигане до различни области на пространството и намалява опасността алгоритъмът да остане в локален екстремум. Примери за мутация са:

- произволна промяна на символ;
- размяна на два символа;
- преместване на символ на друга позиция, при което останалите символи се изместват;
- обръщане на реда на символите във вътрешен интервал.

Генетичният алгоритъм спира при достигане на целево състояние, при достатъчно добра fitness оценка или при изпълнение на друго предварително зададено условие за край. Методът е стохастичен: различни случайни начални популации, селекции и мутации могат да доведат до различни резултати.

25.1.5 Задачи за удовлетворяване на ограничения

Дефиниция 25.17 (Задача за удовлетворяване на ограничения)**.** Задача за удовлетворяване на ограничения е задача, при която трябва да се зададат стойности на множество променливи така, че всички зададени ограничения да бъдат изпълнени.

Нека са дадени променливи

$$V = \{v_1, v_2, \ldots, v_n\}$$

и множество от ограничения. Всяко ограничение задава условие върху една или повече променливи.

Дефиниция 25.18 (Целево състояние при ограничения)**.** Целево състояние е свързване на стойности към променливите, което удовлетворява всички ограничения.

Примери за такива задачи са задачата за осемте царици, оцветяването на географска карта, различни пъзели, судоку и задачи за съставяне на разписания.

Сред използваните методи са генериране и тестване, търсене с възврат, разпространяване на ограниченията и локално търсене чрез намаляване на конфликтите.

Генериране и тестване

При генериране и тестване се построяват възможни свързвания на стойности към променливите. Всяко получено свързване се проверява срещу всички ограничения. Ако проверката е успешна, е намерено решение; ако не е, се генерира следващ кандидат.

Методът е прост, но може да генерира много частични или пълни свързвания, които могат да бъдат отхвърлени още преди да е необходимо да се довършат.

Търсене с възврат

Търсенето с възврат, или *backtracking*, изгражда решението постепенно. На всяка стъпка се избира променлива и се задава стойност. Ако частичното свързване наруши ограничение, текущият избор се отменя и се пробва друга стойност.

Основната схема е:

1. избира се незадала стойност променлива;
2. задава се една от допустимите ѝ стойности;
3. проверява се дали частичното решение нарушава някое ограничение;
4. ако няма нарушение, търсенето продължава рекурсивно;
5. ако всички възможности са изчерпани, се извършва възврат към предишния избор.

Когато всички променливи са получили стойности и ограниченията са изпълнени, е намерено решение. Ако нито една стойност не води до решение, задачата се връща към предишната променлива и променя нейния избор.

Ефективността зависи от правилата за избор на променлива и стойност. Може да се избира променлива, която участва в най-голям брой ограничения. Ако една променлива има повече възможни стойности от друга, за предпочитане е да се избере тази с по-малко възможности. Такива евристики ограничават броя на разглежданите частични решения.

Разпространяване на ограниченията

При разпространяване на ограниченията информацията от вече направените избори се използва за ограничаване на възможните стойности на останалите променливи. Така противоречията могат да бъдат открити по-рано, преди да бъде изградено пълно свързване.

За прилагането на този подход са необходими два типа правила:

- правила, чрез които ограниченията се разпространяват към други променливи;
- правила, чрез които се правят хипотези за следващи стойности.

Разпространяването на ограниченията обикновено се комбинира с търсене с възврат. След всяка хипотеза ограниченията се разпространяват. При откриване на противоречие хипотезата се отменя и се избира друга възможност.

Намаляване на конфликтите

Дефиниция 25.19 (Метод на намаляване на конфликтите)**.** Намаляването на конфликтите, или *min-conflicts*, е локален метод за задачи за удовлетворяване на ограничения, при който се променя стойността на променлива така, че да се намали броят на нарушените ограничения.

Методът започва с някакво пълно или частично свързване, което може да нарушава ограничения. Избира се променлива, участваща в конфликт, и за нея се търси стойност, която води до най-малък брой конфликти. Процесът продължава, докато всички ограничения бъдат изпълнени или докато се достигне предварително определено условие за край.

Това е локално търсене. То не изгражда непременно решение чрез последователно фиксиране на всички променливи и не пази пълно дърво на алтернативите. Следователно методът може да бъде бърз за големи задачи, но не гарантира намиране на решение във всеки случай.

25.1.6 Избор на следващ ход при игра за двама играчи

Друг основен тип задачи за търсене в пространство на състоянията е формирането на стратегия при игра за двама играчи.

Класическата постановка разглежда интелектуални игри, които:

- се играят от двама играчи;
- се изпълняват последователно;
- имат пълна информация за направените ходове и текущото състояние;
- не зависят от случайни фактори;
- имат противоположни интереси на играчите.

Примери са шахматът и морският шах. Задачата обикновено е да се намери най-добрият ход на играча, който трябва да играе в текущото състояние.

В зависимост от разглежданите свойства игрите могат да бъдат с перфектна или неперфектна информация, детерминистични или включващи елементи на шанс, както и с пълна или непълна информация. Мини-макс и алфа-бета отсичането се отнасят до постановката с двама играчи, пълна информация и детерминистични ходове.

Мини-макс процедура

Дефиниция 25.20 (Мини-макс процедура)**.** Мини-макс процедурата е метод за избор на най-добър ход при предположението, че противникът също играе оптимално.

Състоянията на играта образуват дърво на играта. Върховете са позиции, а дъгите са допустими ходове. На листата се задават оценки, които показват колко благоприятна е съответната крайна или достатъчно дълбока позиция за максимизиращия играч.

Нека единият играч е максимизиращият, означен с MAX, а другият е минимизиращият, означен с MIN. При възел на MAX се избира най-голямата стойност на наследниците:

$$\text{value}(n) = \max_{c \in \text{Children}(n)} \text{value}(c).$$

При възел на MIN се избира най-малката стойност:

$$\text{value}(n) = \min_{c \in \text{Children}(n)} \text{value}(c).$$

Оценките се разпространяват от листата към корена. В корена максимизиращият играч избира хода, който води до наследник с най-голяма стойност. На следващото ниво минимизиращият играч избира най-малката стойност, защото се приема, че действа в своя полза. Редуването продължава до листата.

> ▷ **Пример 25.21.** Ако трите непосредствени наследници на корена имат стойности 4, 7 и 5, максимизиращият играч избира наследника със стойност 7. Ако един възел на минимизиращия играч има наследници със стойности 10, 4 и 8, неговата стойност е 4.

Мини-макс процедурата намира най-добрия ход спрямо разгледаното дърво и зададените оценки. За да се приложи директно, трябва да се построи цялото дърво на играта и да се оценят неговите листа. При реални игри дървото може да бъде много голямо, поради което процедурата често е неефективна или практически нереализуема с наличните изчислителни ресурси.

Алфа-бета отсичане

Алфа-бета отсичането подобрява мини-макс процедурата, като избягва генерирането и оценяването на поддървета, което не може да промени крайния избор.

Дефиниция 25.22 (Алфа-стойност)**.** Алфа-стойността на възел на максимизиращия играч е текущата установена долна граница на възможните оценки на този възел.

Дефиниция 25.23 (Бета-стойност)**.** Бета-стойността на възел на минимизиращия играч е текущата установена горна граница на възможните оценки на този възел.

Вместо първо да се генерира цялото дърво, а след това да се оценяват листата, при алфа-бета отсичането листата се оценяват непосредствено след генерирането им. Получените оценки се разпространяват нагоре, а границите се обновяват по време на обхода.

Нека се разглежда възел на MIN, който има максимизиращ родител с текуща алфа-стойност α. Ако стойността на възела на MIN стане по-малка или равна на α, останалите му наследници не е необходимо да се генерират. Това е алфа-отсичане:

$$\beta \leq \alpha.$$

Причината е, че максимизиращият родител вече разполага с възможност, оценена поне с α, и няма да избере този клон, ако минимизиращият играч може да го ограничи до стойност, не по-голяма от α.

Съответно, ако се разглежда възел на MAX с минимизиращ родител с текуща бета-стойност β, и стойността на възела стане по-голяма или равна на β, останалите наследници могат да бъдат отсечени. Това е бета-отсичане:

$$\alpha \geq \beta.$$

Мини-макс и алфа-бета отсичането дават един и същ резултат, ако се използват еднакво дърво и еднакви оценки. Разликата е, че алфа-бета методът може да не генерира части от дървото, които не оказват влияние върху избрания ход. При подходящ ред на разглеждане на наследниците броят на извършените операции може да бъде намален значително.

Забележка 25.24. Алфа-бета отсичането не променя критерия за избор на ход. То променя начина на обход и използва вече получени долни и горни граници, за да пропусне безполезни клони.

25.2 Изпитен фокус

За устно изложение трябва да могат да бъдат възпроизведени:

- определенията за състояние, оператор, пространство на състоянията, решение и стратегия;
- представянето на пространството чрез ориентиран граф или дърво на търсенето;
- критериите пълнота, времева сложност, пространствена сложност и оптималност;
- разликата между неинформирано, информирано, глобално и локално търсене;
- идеята и оценката на търсенето в лъч;
- идеята, стъпките и недостатъците на изкачването на хълмове, включително локален екстремум, плато и хребет;
- правилото за приемане при симулирано втвърдяване, ролята на температурата и компромисът в схемата за охлаждане;

- представянето на хромозоми и ролята на fitness функцията в генетичните алгоритми;
- етапите селекция, кръстосване и мутация;
- основните типове кръстосване и примерите за мутация;
- определението за задача за удовлетворяване на ограничения и ролята на променливите, стойностите и ограниченията;
- принципът на търсенето с възврат и идеята на разпространяването на ограниченията;
- принципът на метода *min-conflicts*;
- постановката на игри за двама играчи с пълна информация и противоположни интереси;
- мини-макс правилата: max на възлите на максимизиращия играч и min на възлите на минимизиращия играч;
- алфа- и бета-стойностите и условията за алфа-бета отсичане;
- фактът, че алфа-бета отсичането дава същия резултат като мини-макс, но може да спести генериране и оценяване на цели поддървета.

Въпрос 26

Съвременни софтуерни технологии.

По конспекта: какво трябва да мога

Понятия: софтуерен продукт и софтуерен процес — §26.1, стр. 365.
Фази: основни фази на софтуерните процеси — §26.2, стр. 366.
Модели и сравнение: модел на водопада, на бързата разработка, еволюционни модели, прототипен модел, спираловиден модел, плюс сравнителна характеристика — §26.3, стр. 367.
Гъвкави методи: Extreme Programming (XP) и SCRUM — §26.4, стр. 370.
Изисквания: функционални и нефункционални — §26.5, стр. 371.
Анализ и проектиране: на софтуерните изисквания — §26.6, стр. 372.
Езици за описание: UML — §26.7, стр. 374.
Качество: верификация и валидация — §26.8; управление на качеството — §26.9, стр. 375.
Литература: [9], [39], [44].

26.1 Софтуерен продукт, софтуерен процес и съвременни софтуерни технологии

Дефиниция 26.1 (Софтуерен продукт)**.** Софтуерен продукт е програма или съвкупност от взаимодействащи си програми, записани върху технически носител и придружени от съответната документация. В по-широк смисъл продуктът включва не само изпълнимия код, но и описанията, инструкциите и останалите материали, необходими за неговото използване, внедряване и съпровождане.

Дефиниция 26.2 (Софтуерен процес)**.** Софтуерен процес е структуриран набор от дейности, необходими за разработването на софтуерна система в определен срок и с изисквано качество. Към основните дейности се отнасят комуникацията, планирането, моделирането, конструирането и внедряването.

Софтуерният процес обхваща както техническата работа по създаване на системата, така и управлението на разработването. В него участват различни заинтересовани лица: клиенти, крайни потребители, ръководители на проекта, анализатори, проектанти, програмисти, тестери и други специалисти. Различните модели на софтуерни процеси предполагат различна степен и начин на участие на тези лица.

Дефиниция 26.3 (Съвременни софтуерни технологии)**.** Софтуерните технологии представляват систематичен подход към разработването на качествен софтуер, както и към неговото предлагане, внедряване, експлоатация и съпровождане.

Съвременната разработка на софтуер е многостепенен процес. Тя изисква организиране на комуникацията, прецизно определяне на бъдещите функции, планиране, моделиране, конструиране, проверка и поддръжка. Целта е да се създаде продукт, който удовлетворява потребителските потребности и едновременно с това е надежден, поддържаем, преносим и способен да се развива.

При разработването се използват модулен подход, подходящо моделиране, итеративно усъвършенстване, прототипи и стандартизация. Модулният подход разделя системата на модули и компоненти. Моделирането представя системата на подходящо ниво на абстракция. Итеративността позволява системата да се усъвършенства на малки стъпки, а прототипите подпомагат изясняването на потребителските нужди. Стандартизацията улеснява изграждането на интерфейси между частите на системата и подпомага повторяемостта на решенията.

26.1.1 Управление на софтуерен проект и ресурси

Дефиниция 26.4 (Управление на софтуерен проект)**.** Управлението на софтуерен проект представлява извършване на съвкупност от разнородни дейности за решаване на сложен, нестандартен проблем при ограничения относно време, разходи, качество и организация на работата.

Проектът има предварително определени цели и изисквания към продължителността, разходите и качеството. Възможно е изпълнението му да наложи преструктуриране на организацията, промяна в стила на работа и промяна в управлението на човешките ресурси. Тъй като софтуерният проект обикновено включва много участници и взаимосвързани дейности, управлението осигурява контрол, обратна връзка и съгласуване между тях.

26.2 Основни фази на софтуерните процеси

Независимо от избрания модел, софтуерният процес може да бъде разглеждан чрез няколко основни групи дейности:

1. **Комуникация и определяне на изискванията.** Изучават се потребностите на клиента и потребителите, уточнява се предметната област и се определя каква система трябва да бъде създадена.

2. **Планиране.** Определят се обхватът, сроковете, необходимите ресурси, отговорностите, рисковете и начинът на организиране на работата.
3. **Анализ и моделиране.** Информацията за системата се анализира и се създават модели, които описват нейните функции, структура и поведение.
4. **Проектиране.** Изискванията се превръщат в описание на архитектурата, компонентите, интерфейсите и взаимодействията между частите на системата.
5. **Конструиране.** Извършват се програмиране, интегриране на компонентите и създаване на необходимата документация.
6. **Верификация и валидация.** Проверява се дали системата е разработена съгласно спецификацията и дали реализира действителните потребности на клиента.
7. **Внедряване и експлоатация.** Системата се предоставя на потребителите, използва се в реална среда и се наблюдава нейното функциониране.
8. **Съпровождане.** Извършват се корекции, адаптиране към промени и допълнително развитие на продукта.

Тези фази не винаги се изпълняват еднократно и строго последователно. При итеративните и гъвкавите процеси те се повтарят за отделни части на системата. Така във всеки цикъл може да се получи работещ инкремент, който се оценява и усъвършенства.

26.3 Модели на софтуерни процеси

Дефиниция 26.5 (Модел на софтуерен процес)**.** Моделът на софтуерния процес е абстрактно описание на софтуерния процес, представено от определена перспектива. Той задава основните дейности, тяхната организация и връзките между тях.

Моделът не описва всеки детайл от конкретния проект. Неговото предназначение е да даде разбираема схема за планиране, управление и оценяване на разработката.

26.3.1 Модел на водопада

Дефиниция 26.6 (Модел на водопада)**.** Моделът на водопада организира разработката като последователност от фази. Обикновено се преминава през определяне на изискванията, анализ и проектиране, конструиране, тестване, внедряване и съпровождане. Резултатите от една фаза служат като основа за следващата.

Предимството на модела е ясната организация. Във всеки момент се знае коя фаза се изпълнява и какви резултати трябва да бъдат получени. Това улеснява планирането, документацията и контрола.

Основният недостатък е трудното реагиране на промени. Ако съществена грешка в изискванията бъде открита късно, може да се наложи връщане към предишни фази и преработване на голяма част от системата. Поради това моделът е по-подходящ, когато изискванията са добре познати и стабилни.

26.3.2 Модел на бързата разработка

Дефиниция 26.7 (Бърза разработка)**.** Моделът на бързата разработка е ориентиран към създаване и доставяне на софтуер за кратко време. Той насърчава ускорено моделиране, повторно използване на готови компоненти, бърза реализация и ранно предоставяне на работещи части от продукта.

Предимството е съкращаването на времето до първата работеща версия и възможността потребителите рано да дадат обратна връзка. Недостатък е, че при недостатъчно планиране и сложна система може да се получи слаба архитектура, трудна поддръжка или непълно покритие на изискванията.

26.3.3 Еволюционни модели

Дефиниция 26.8 (Еволюционен модел)**.** Еволюционните модели приемат, че изискванията и разбирането на системата се развиват постепенно. Продуктът се създава чрез поредица от версии, като всяка версия добавя или подобрява функционалност.

При този подход потребителите могат да оценяват работещите части и да уточняват бъдещите потребности. Разработката се приспособява към промените, а най-важните функции могат да бъдат реализирани първи.

Слабост е възможното влошаване на структурата, ако към системата се добавят изменения без достатъчно преработване и архитектурен контрол. Освен това крайната система може да бъде трудна за планиране, когато бъдещите изисквания са неясни.

26.3.4 Прототипен модел

Дефиниция 26.9 (Прототипен модел)**.** Прототипният модел използва прототип за изясняване на основните потребителски нужди и за демонстрация на бъдещи функционалности. Прототипът е предварителен модел на системата или на част от нея. Той може да бъде изграден с ограничена функционалност, но трябва да бъде достатъчно разбираем за обсъждане и оценяване.

Прототипът помага на заинтересованите лица да видят как би изглеждала системата и да открият неясни или неправилно разбрани изисквания. След получаване на обратна връзка прототипът може да бъде доуточняван. Той може да бъде изхвърлен след анализа или да се превърне в основа на окончателната реализация, според избраната организация на разработката.

Предимството е намаляването на риска от създаване на система, която не съответства на очакванията. Недостатък е опасността потребителите да приемат прототипа за завършен продукт, въпреки че той може да не притежава необходимите качества, сигурност и поддържаемост.

26.3.5 Спираловиден модел

Дефиниция 26.10 (Спираловиден модел)**.** Спираловидният модел организира разработката като повтарящи се цикли. Във всеки цикъл се извършват планиране, анализ на рисковете, разработване и оценяване на резултата. След това се определят целите на следващия цикъл.

Моделът съчетава елементи на последователната разработка и на еволюционното усъвършенстване. Особено значение има анализът на рисковете. Чрез ранно откриване на технически, организационни и изисквателни рискове могат да се вземат решения за ограничаването им.

Предимството е възможността сложни и рискови проекти да се разработват постепенно, с редовна оценка. Недостатък е по-голямата сложност на управлението и необходимостта от опит при идентифициране и оценяване на рисковете.

26.3.6 Сравнителна характеристика

Модел	**Организация**	**Основно предимство**	**Основно ограничение**
Водопад	Последователни фази	Ясен план, документация и контрол	Трудно приемане на промени
Бърза разработка	Ускорена реализация и доставка	Кратко време до работещ продукт	Риск за архитектурата и качеството
Еволюционен	Поредица от развиващи се версии	Адаптиране към уточняващи се изисквания	Възможна структурна деградация
Прототипен	Предварителен модел и обратна връзка	Изясняване на потребителските нужди	Прототипът може да бъде погрешно възприет като завършен
Спираловиден	Повтарящи се цикли с анализ на риска	Управление на сложността и риска	Сложно управление и оценяване

Изборът на модел зависи от стабилността на изискванията, размера и сложността на проекта, риска, необходимия срок и степента на участие на потребителите. Не съществува един модел, който да е най-подходящ за всички системи.

26.4 Гъвкави методи

Дефиниция 26.11 (Гъвкави методи)**.** Гъвкавите методи са ориентирани към бързо разработване и доставяне на работещ софтуер. Те се основават на итеративно разработване и на ограничаване на излишните забавяния в процеса.

Основните принципи са:

- активно включване на потребителя в процеса на разработване;
- поетапно и инкрементално развитие на продукта;
- приоритизиране на функционалностите, така че най-важните да бъдат реализирани първи;
- приемане на неизбежността на промените;
- адаптиране на разработката спрямо новите знания и променените потребности;
- редовно получаване на обратна връзка и оценяване на работещия резултат.

26.4.1 Extreme Programming

Дефиниция 26.12 (Extreme Programming)**.** Extreme Programming, или XP, е влиятелен гъвкав метод, който съчетава организационни и инженерни практики. Неговата цел е често да се доставят малки, работещи версии, като потребителските потребности се описват разбираемо и разработката се проверява непрекъснато.

В XP се използват потребителски истории. Потребителската история описва желана възможност от гледната точка на потребителя. Историите подпомагат комуникацията и служат като основа за планиране на малки функционалности.

Важна практика е разработката „първо тестване“. При нея изпълнимите тестове се създават преди реализацията на функционалността. След това се разработва кодът, необходим за преминаването на тестовете. Този подход е известен като Test Driven Development.

Забележка 26.13. Разработката, основана на тестове, насочва програмиста към изпълнение на конкретно проверими изисквания. Нейно ограничение е опасността вниманието да се съсредоточи върху преминаването на тестовете, без да се отчетат достатъчно глобалната структура на кода и използваните алгоритми.

XP използва и непрекъсната интеграция, при която промените се интегрират редовно в общия продукт. Така проблемите при съвместяването на отделни части могат да се открият по-рано. Сред останалите характерни практики са рефакторингът, комуникацията между програмисти и потребители, малките версии, споделената собственост върху продукта и колективната работа.

Feature Driven Development, или FDD, е итеративен и инкрементален подход, при който работата се организира около малки функционалности, наричани features. Отправна точка е функционалността от гледната точка на потребителя или клиента. Подходът

включва разработване на цялостен модел, изграждане на план за функционалностите, планиране и проектиране на всяка функционалност, конструиране и доставка на малки завършени части.

Основната цел е през договорени времеви интервали да се доставя работещ софтуер. Разделянето на задачите на малки функционалности улеснява проследяването, разпределянето на отговорностите и оценяването на напредъка.

26.4.2 SCRUM

Дефиниция 26.14 (Scrum)**.** SCRUM е гъвкав метод, който се съсредоточава върху гъвкаво планиране и управление. За разлика от XP, той не определя задължителни инженерни практики. Екипът може да избере техническите практики, които счита за подходящи за конкретния продукт.

Разработката се организира в спринтове. Спринтът е фиксиран период, обикновено от две до четири седмици, през който се разработва инкремент на продукта. Инкрементът трябва да бъде резултат, който може да бъде оценен и да допринася към развитието на системата.

Основни практики в SCRUM са:

- **Продуктов беклог** — списък на работните елементи, които трябва да бъдат реализирани. На всеки спринт се избират част от тези елементи.
- **Спринтове** — фиксирани периоди, в които екипът планира, разработва и получава завършен инкремент.
- **Самоорганизиращ се екип** — разработващите организират работата чрез обсъждане и споразумение между членовете на екипа.

XP и SCRUM могат да бъдат комбинирани. SCRUM може да даде управленска рамка със спринтове и беклог, а XP да предостави инженерни практики като тестове преди кода, непрекъсната интеграция и рефакторинг.

26.5 Изисквания към софтуерните системи

Дефиниция 26.15 (Софтуерно изискване)**.** Софтуерното изискване е описание на необходима функция, услуга, качество или ограничение на системата или на процеса за нейното разработване.

Изискванията свързват потребностите на заинтересованите лица с бъдещата реализация. Те трябва да бъдат достатъчно ясни, непротиворечиви и осъществими. Резултатът от анализа и специфицирането трябва да бъде нареден списък от изисквания, който може да служи като основа за проектиране, програмиране и проверка.

26.5.1 Функционални изисквания

Дефиниция 26.16 (Функционално изискване)**.** Функционалното изискване описва функция или услуга, която системата трябва да предоставя, начина, по който тя трябва да реагира на конкретни входни данни, и поведението ѝ в определени ситуации.

Функционалните изисквания отговарят главно на въпроса „какво трябва да прави системата“. Те могат да описват обработване на вход, получаване на резултат, взаимодействие с потребителя, реакция при грешка или изпълнение на конкретна бизнес операция.

26.5.2 Нефункционални изисквания

Дефиниция 26.17 (Нефункционално изискване)**.** Нефункционалното изискване задава ограничение върху функционалността на системата или върху процеса на разработка и определя системни характеристики като надеждност и време за отговор.

Нефункционалните изисквания описват главно въпроса „как трябва да работи системата“. Те се отнасят до качествата на софтуера, например надеждност, производителност, време за отговор, сигурност, поддържаемост и преносимост. Те могат да бъдат и изисквания към процеса, например използване на конкретен програмен език или метод на разработване.

Функционалните и нефункционалните изисквания са взаимосвързани. Функцията може да е правилно реализирана, но системата да не удовлетворява потребителя, ако отговорът е твърде бавен, данните не са защитени или продуктът е труден за поддръжка.

26.6 Анализ и проектиране на софтуерните изисквания

Дефиниция 26.18 (Анализ на изискванията)**.** Анализът на изискванията представлява систематично събиране, изследване и уточняване на информацията за бъдещата система. Различните заинтересовани лица могат да имат различни потребности и очаквания. Затова анализът включва комуникация, интервюта, преглед на документи, обсъждания с потребители и експертни оценки.

Основните дейности са:

1. **Проучване на осъществимостта.** Определя се дали разработването на системата си струва и дали е възможно при зададените ограничения.
2. **Идентифициране и анализ на изискванията.** Събира се информация за предлаганата система и от нея се извличат потребителските и системните изисквания.
3. **Специфициране на изискванията.** Изискванията се записват в подходящ вид, така че да бъдат разбрани и използвани от заинтересованите лица.

4. **Валидиране на изискванията.** Проверява се дали изискванията определят системата, която клиентът действително иска.

5. **Управление на изискванията.** Проследяват се промените, приоритетите, одобренията и връзките между изискванията и резултатите от разработката.

При валидирането се търсят непълни, противоречиви, неясни или неосъществими изисквания. Важно е изискванията да бъдат одобрени, приоритизирани и управлявани през целия жизнен цикъл. Промяната в едно изискване може да засегне архитектурата, компонентите, тестовете и документацията.

26.6.1 Проектиране на софтуера

Проектирането започва при налични изисквания и обхваща дейностите по превръщането им в описание, което определя съдържанието и функциите на програмите. Целта е решението да бъде опростено и недвусмислено за всички заинтересовани лица. При проектирането се използват абстракция, декомпозиция и моделиране.

Архитектурата се оформя от основополагащи функционални, качествени и бизнес изисквания, наричани архитектурни драйвери. Те влияят върху избора на структура, компоненти, интерфейси и начини на взаимодействие.

Основен принцип е последователната декомпозиция: първо се изследва и разбира проблемът, след което той се разделя на подпроблеми и отговорности. Така сложната система се превръща в съвкупност от по-малки части, чиито функции и връзки могат да бъдат описани.

26.6.2 Обектно-ориентиран дизайн

Дефиниция 26.19 (Обектно-ориентиран дизайн)**.** Обектно-ориентираният дизайн представя системата като група от структурирани обекти, които си взаимодействат.

Дефиниция 26.20 (Обект)**.** Обектът е атомарна същност, която има състояние, поведение и индивидуалност.

Дефиниция 26.21 (Клас)**.** Класът е шаблон за създаване на обекти. Той описва данните, операциите и поведението на обектите от съответния тип.

Класът може да съдържа данни от определени типове и структурни връзки към други обекти. Обектно-ориентираният дизайн определя софтуерно решение чрез обекти, класове, характеристики, интерфейси, компоненти и операции, които трябва да удовлетворят изискванията.

Обикновено началото се поставя с кандидат-обекти, открити при анализа. При проектирането те се променят, обединяват или разделят, когато това подобрява структурата на решението. Важни са не само самите обекти, но и комуникацията между тях, разпределянето на отговорностите и съответствието с изискванията.

26.7 Езици за описание и UML

Езиците за описание могат да се разделят на две основни групи:

- **Домейн-специфични езици** (DSL) — специализирани езици с ограничена изразителност, насочени към определена предметна област;
- **Езици с общо предназначение** (GPL) — езици, предназначени за писане на всякакъв вид програми с различна логика, например C++ и Java.

Дефиниция 26.22 (UML)**.** UML (Unified Modeling Language) е обектно-ориентиран език за специфициране, визуализиране, конструиране и документиране на елементите на софтуерна система.

Като моделиращ език UML съдържа елементи на модела, визуално представяне на тези елементи и насоки за тяхната употреба. UML не е програмен език. Той служи за описание и комуникация на структурата и поведението на системата.

Сред използваните UML диаграми са:

- диаграми на случаите на употреба, които представят взаимодействието между потребители и система;
- диаграми на дейността, които описват потоци от дейности;
- диаграми на последователността, които представят подредбата на съобщенията във времето;
- диаграми на класовете, които представят класове, атрибути, операции и връзки;
- диаграми на състоянията, които описват измененията на състоянието;
- диаграми на компонентите, пакетите и разпределението, които описват структурната организация и разполагането на системата;
- диаграми на комуникацията и други диаграми на взаимодействието, които представят комуникацията между елементите.

Диаграмите на случаите на употреба подпомагат анализа на функционалните изисквания. Диаграмите на класовете подпомагат обектно-ориентираното проектиране. Диаграмите на последователността и дейността представят поведението и взаимодействията. Така една и съща система може да бъде разглеждана от различни перспективи.

26.8 Верификация и валидация на софтуера

Дефиниция 26.23 (Верификация). Верификацията проверява дали софтуерът е разработен съгласно зададените изисквания, правила и спецификации. Тя отговаря на въпроса: „Изграждаме ли системата правилно?“.

Дефиниция 26.24 (Валидация). Валидацията проверява дали създаденият софтуер реализира системата, която клиентът и потребителите действително искат. Тя отговаря на въпроса: „Изграждаме ли правилната система?“.

Софтуерното тестване е процес, при който програмите се изследват за установяване на съответствието им с характеристики, правила и изисквания. Целта е да се открият грешки и да се покаже, че програмата прави очакваното от разработчиците.

Основните видове тестване, посочени в обхвата, са:

- **Функционално тестване** — проверява функциите на системата и цели откриване на възможно най-много грешки;
- **Потребителско тестване** — проверява дали продуктът е използваем и подходящ за крайните потребители;
- **Тестване на натовареността и производителността** — проверява дали софтуерът работи достатъчно бързо и може да поеме очакваното натоварване;
- **Тестване на сигурността** — проверява целостта на софтуера и способността му да защитава потребителската информация от кражба и повреда.

Тестването трябва да бъде планирано и управлявано. Управлението включва оценяване на тестови сценарии, създаване на тестова среда, провеждане на тестовете, регистриране и проследяване на дефектите и поддържане на показатели за системата.

26.9 Управление на качеството

Дефиниция 26.25 (Управление на качеството на софтуера). Управлението на качеството на софтуера е съвкупност от дейности за осигуряване на необходимото ниво на качество чрез изграждане и прилагане на процеси, стандарти и контролни механизми.

Управлението на качеството цели софтуерът да има малък брой грешки и да достига изискваните стандарти за поддържаемост, надеждност, преносимост и други качества. То не се свежда само до крайното тестване. Качеството трябва да се планира и оценява през целия процес.

На организационно ниво управлението на качеството изгражда процеси и стандарти, които подпомагат създаването на висококачествен софтуер. На ниво проект се създава план за качество. Той определя целите за качество, използваните процеси, стандартите и начините за контрол.

Качеството включва:

- характеристики и предпочитани качества на продукта;
- методи за контрол и оценяване;
- описание на продукта и неговите ограничения;
- цели за качество;
- процеси за управление на риска.

Постоянната оценка на качеството позволява проблемите да бъдат откривани по-рано. За целта се използват проверки на изискванията, прегледи на проектирането, контрол на програмните компоненти, тестове и наблюдение на системните показатели. Качеството е свързано и със сигурността, надеждността, поддържаемостта и способността на системата да се развива.

26.10 Изпитен фокус

За устен отговор трябва да могат да бъдат възпроизведени:

- определенията за софтуерен продукт, софтуерен процес, софтуерни технологии и модел на софтуерен процес;
- основните фази: комуникация, планиране, анализ, проектиране, конструиране, верификация и валидация, внедряване и съпровождане;
- характеристиките, предимствата и ограниченията на водопадния, бързия, еволюционния, прототипния и спираловидния модел;
- принципите на гъвкавите методи и разликата между XP и SCRUM;
- потребителските истории, разработката „първо тестване“, непрекъснатата интеграция и ролята на рефакторинга;
- продуктовият беклог, спринтът, инкрементът и самоорганизиращият се екип в SCRUM;
- разликата между функционални и нефункционални изисквания;
- дейностите по анализа на изискванията: осъществимост, идентифициране, специфициране, валидиране и управление;
- принципът на декомпозицията и ролята на архитектурните драйвери при проектирането;
- понятията обект, клас и обектно-ориентиран дизайн;
- предназначението на UML и основните групи UML диаграми;
- разликата между верификация и валидация;

- видовете тестване: функционално, потребителско, на натовареността и производителността и на сигурността;
- управлението на качеството на организационно и проектно ниво, както и ролята на стандартите, плана за качество, тестовете и управлението на риска.

Въпрос 27

Архитектури на софтуерни системи.

По конспекта: какво трябва да мога

Понятие: софтуерна архитектура; структури и изгледи; необходимост; влияние върху проекта и организацията — §27.1, стр. 378.
Качество: изисквания към качеството — нефункционални изисквания — §27.2, стр. 382.
Архитектурни стилове (назованият минимум): многослоен, модел–изглед–контролер, поточен (*pipe-and-filter*) — §27.4, стр. 384.
Архитектурен стил: ориентирана към услуги архитектура — §27.5, стр. 387.
Проектиране: процес за проектиране, избор на подходящи структури, последователност на създаване — §27.6, стр. 387.
Тактики: за постигане на желаните качествени показатели — §27.7, стр. 389.
Документиране: предназначение и основни елементи на документацията — §27.8, стр. 391.
† **Допълнително:** клиент-сървърният стил, споделените данни и неявното извикване (§27.4.4–§27.4.6, стр. 386), както и архитектурният анализ (§27.6.3), не са част от назования минимум.
Литература: [30], [52].

27.1 Понятие за софтуерна архитектура

Дефиниция 27.1. Софтуерната архитектура на дадена система е съвкупност от структури, които показват софтуерните елементи на системата, външно видимите им свойства и връзките между тях.

Архитектурата описва фундаменталната организация на системата: нейните основни елементи, тяхното поведение, взаимодействията между тях и решенията, свързани с качествените характеристики. Софтуерните елементи обикновено се разглеждат като черни кутии. Това означава, че архитектурата се интересува от тяхната роля, интерфейси и

взаимодействия, а не от вътрешната реализация на отделните алгоритми, структурите от данни или конкретния програмен код.

Софтуерната архитектура се създава в началните етапи на проектирането. Тя трябва да осигури възможност системата да удовлетвори най-важните си изисквания, особено нефункционалните. Архитектурата не е равнозначна на подробния дизайн. В подробния дизайн се уточняват алгоритми, представяне на данните и конкретни реализации, докато архитектурата определя основните граници, отговорности, зависимости и механизми за взаимодействие.

В зависимост от обхвата могат да се разграничат три взаимосвързани равнища:

- организационна архитектура – основните процеси и бизнес стратегии в организацията;
- системна архитектура – организацията на инфраструктурата, върху която се изпълняват програмите;
- приложна архитектура – организацията на конкретно приложение, подсистема или компонент.

В контекста на разработването на софтуерни системи особено значение имат системната и приложната архитектура. Архитектурните решения обаче могат да бъдат повлияни и от организационни фактори, наличните технологии, хардуера, опита на екипа и ограниченията на средата.

27.1.1 Структури и изгледи

Дефиниция 27.2. Структура е съвкупност от софтуерни елементи, техните външно видими свойства и връзките между тях.

Дефиниция 27.3. Изглед е конкретно документирано представяне на дадена структура, предназначено да покаже определен аспект на архитектурата.

Една структура не може да опише всички свойства на системата едновременно. Например разпределението на функционалностите между модулите е различно от разпределението на процесите по процесори. Затова архитектурата се описва чрез множество взаимосвързани структури и изгледи.

Основните групи структури са:

1. структури на модулите;
2. структури на процесите;
3. структури на разположението.

Структури на модулите

Елементите на модулните структури са модулите – логически единици, които групират функционалност или друга свързана отговорност. Тези структури отговарят на въпроси като:

- коя функционалност в кой модул се реализира;
- кои модули използват даден модул;
- какво съдържа даден модул;
- какви са зависимостите между модулите;
- как модулите се групират и декомпозират.

Важни разновидности са декомпозицията на модулите, употребата на модулите, структурата на слоевете и йерархията на класовете.

При декомпозицията се използва отношение от вида „X е подмодул на Y“. Разбиването продължава до достигане на достатъчно малки и разбираеми единици. Добрата декомпозиция групира функционално сходни отговорности, увеличава кохезията и намалява свързаността между различните части на системата. Тя улеснява разработването, поддръжката и промяната, а също така може да служи като основа за разпределяне на работата между екипите.

Структурата на употребата на модулите представя зависимости от вида „X използва Y“. Връзката може да бъде уточнена чрез конкретен интерфейс или ресурс на използвания модул. Тази структура позволява последователна разработка и показва кои модули трябва да бъдат налични, за да се реализира дадена функционалност.

Структурата на слоевете е частен случай на структурата за употреба на модулите. При нея зависимостите са ограничени чрез правила. Обикновено даден слой използва услуги само от по-долния слой и предоставя услуги само на по-горния слой. Слоевете могат да бъдат реализирани като пакети, виртуални машини или цели подсистеми. При спазване на договорените интерфейси цял слой може да бъде заменен с друга реализация.

Йерархията на класовете представя отношенията между класове в обектно-ориентиран контекст, включително наследяване и полиморфизъм. Тя показва защо определено поведение е обособено в даден клас и защо друг клас го разширява.

Структури на процесите

Елементите на структурите на процесите се проявяват по време на изпълнението. Това са процеси, нишки, изчислителни единици и средствата за комуникация между тях.

Тези структури показват:

- кои са основните изчислителни процеси;
- как процесите си взаимодействат;

- как споделят ресурси;
- как се синхронизират и паралелизират;
- как изменят данните в системата.

Структурите на процесите имат пряко отношение към бързодействието, паралелизма и надеждността на системата по време на изпълнение. Разновидност е структурата на потока от данни, която показва движението на информацията между елементите.

Структури на разположението

Структурите на разположението показват връзката между софтуерните елементи и тяхната среда по време на разработката или изпълнението. Те дават информация:

- на кой процесор или хардуерно устройство се изпълняват елементите;
- в кои файлове и артефакти се намират;
- кой екип разработва отделните части.

Основни разновидности са структурата на внедряването, файловата структура и структурата на разпределението на работата.

Структурата на внедряването показва как софтуерът се разполага върху хардуера. Нейни елементи са процеси, хардуерни устройства и комуникационни канали. Тя е особено важна при разпределени системи, тъй като подпомага анализа на производителността, разпределението на отговорностите и целостта на данните.

Файловата структура показва в кои файлове и артефакти се помещават модулите по време на различните фази на реализацията. Структурата на разпределението на работата показва кой екип разработва даден модул. Тя подпомага обособяването на общи функционалности и възлагането им на подходящи специализирани екипи.

27.1.2 Необходимост и влияние на архитектурата

Архитектурата е необходима, защото сложната система не може да бъде проектирана и разбрана като неделимо цяло. Тя разделя проблема на елементи с ясни отговорности и определя как тези елементи си взаимодействат. Така архитектурата осигурява общ модел за комуникация между заинтересованите лица – клиенти, архитекти, разработчици, тестери, администратори и ръководители.

Архитектурата влияе върху проекта по няколко начина:

- определя основната организация на системата и границите между подсистемите;
- влияе върху избора на технологии и инфраструктура;
- определя възможностите за паралелна работа на екипите;
- влияе върху разходите за разработване, внедряване и поддръжка;

- предопределя до голяма степен производителността, надеждността, сигурността и изменяемостта;
- помага за откриване на рисковете още преди реализацията на системата.

След като се оформят първите нива на декомпозиция, е възможно различни екипи да работят независимо по съответните модули. За тази цел се дефинират интерфейси, създават се скелет на системата и временни реализации или стъбове. Обикновено първо се реализират компонентите, свързани с изпълнението и взаимодействието между архитектурните компоненти, тоест middleware. След това се реализират избрани функционалности, като конкретният ред зависи от рисковете, наличния персонал и пазарните приоритети.

27.2 Изисквания към качеството на системата

Дефиниция 27.4 (Функционални и нефункционални изисквания)**.** Изискванията към софтуера се разделят на функционални и нефункционални. Функционалните изисквания определят какви функции предоставя системата. Нефункционалните изисквания определят как системата изпълнява тези функции и какви ограничения трябва да спазва.

Нефункционалните изисквания се наричат още качествени изисквания или изисквания към качествените атрибути.

Дефиниция 27.5. Качествен атрибут е измеримо или тестируемо свойство на системата, което служи като индикатор доколко системата удовлетворява нуждите на заинтересованите лица.

Примери за технологични качествени атрибути са надеждност, наличност, производителност, сигурност, използваемост, скалируемост, преносимост, изменяемост и тестируемост. Качеството може да включва и бизнес характеристики, например време до пускане на пазара, себестойност, очаквана печалба и предвидено време на живот на системата. Съществуват и архитектурни качества като идейна цялост, коректност спрямо изискванията и градивност, тоест колко лесно и успешно екипът може да изгради системата с избраната архитектура.

Качеството е различно за различните заинтересовани лица. Поради това изрази като „висока производителност“ или „лесна поддръжка“ не са достатъчно еднозначни. Необходимо е качествените изисквания да бъдат формализирани чрез сценарии за качество.

Дефиниция 27.6. Сценарий за качество е специфично изискване към поведението на системата в дадена ситуация, формулирано така, че резултатът да може да бъде оценен чрез количествени параметри.

Сценарият за качество съдържа следните компоненти:

1. *източник* – генераторът на въздействието;

2. *въздействие* – състоянието или събитието, което подлежи на обработка;
3. *обект* – системата или компонентът, върху който се случва въздействието;
4. *контекст* – условията, при които обектът се намира по време на въздействието;
5. *резултат* – действието, което обектът предприема;
6. *количествени параметри* – измерими показатели, чрез които се проверява дали изискването е изпълнено.

Например сценарий за производителност може да гласи: при постъпване на заявка от потребител към системата, при нормално натоварване, системата трябва да върне резултат за не повече от зададен интервал. Източникът е потребителят, въздействието е заявката, обектът е системата, контекстът е нормалното натоварване, резултатът е отговорът, а количественият параметър е максималното време за реакция.

27.2.1 Надеждност, наличност и сигурност

Дефиниция 27.7 (Надеждност)**.** Надеждността е мярка доколко системата се държи по очаквания начин за разработчиците и потребителите. Тя включва честотата на неизправностите, поведението при откази и възможността за възстановяване.

Наличността е свързана с това системата да бъде достъпна за легитимна употреба в необходимия момент. Надеждността и наличността се влияят от избора на компоненти, комуникационни механизми, резервиране, наблюдение и възстановяване.

Сигурността е способността на системата да устоява на опити за неразрешена употреба, без това да пречи на легитимните потребители. При сигурните системи се разглеждат поверителност, интегритет, наличност, проверяемост и невъзможност за отричане на извършени действия.

27.3 Архитектурни компоненти и конектори

Дефиниция 27.8. Компонент е изчислителна единица с определена функционалност, достъпна чрез добре дефинирани интерфейси.

Компонентът може да бъде изграден от конкретни програмни артефакти, но архитектурно представлява абстракция на група функционалности. Той има входни и изходни интерфейси и скрива вътрешната си реализация.

Дефиниция 27.9. Конектор е правило и механизъм за комуникация и взаимодействие между компоненти.

Конекторите също имат интерфейси, наричани роли. Те осигуряват независими от конкретното приложение механизми за взаимодействие и обикновено не се виждат директно в кода като отделни обекти. Примери са извикване на процедура, съобщение, споделено хранилище, комуникационен канал или опашка.

Конекторите могат да се класифицират според ролята си:

- *комуникатори* – отделят комуникацията от изчисленията и влияят върху производителността, скалируемостта и сигурността;
- *координатори* – отделят управлението от изчисленията и координират предаването на данни;
- *конвертори* – осигуряват взаимодействие между независимо разработени или несъответстващи компоненти, например чрез адаптери и обвивки;
- *фасилитатори* – улесняват взаимодействието между компоненти, предназначени да работят заедно, например медиатори и оптимизатори.

Изборът на конектори трябва да се основава на качествените изисквания. Един и същ набор от компоненти може да има различни характеристики в зависимост от това дали комуникацията е синхронна или асинхронна, пряка или посредствана, локална или мрежова. Следователно архитектът трябва да познава възможните конектори и влиянието им върху системата.

27.4 Архитектурни стилове

Дефиниция 27.10. Архитектурният стил определя семейство от системи чрез модел на структурна организация.

Стилът определя речника от допустими компоненти и конектори, както и ограниченията за начина, по който те могат да бъдат комбинирани. Стилът не е напълно готова архитектура, а образец, който се конкретизира според изискванията на определена система.

27.4.1 Многослоен стил

Дефиниция 27.11 (Многослоен стил)**.** При многослойния стил системата се разделя на слоеве. Всеки слой използва услуги само от по-долния слой и предоставя услуги само на по-горния слой. Например една система може да съдържа слой на потребителския интерфейс, слой на приложната логика и слой за обработка на данните.

Стилът осигурява няколко нива на абстракция и капсулация. Всеки слой групира сходни функции, което подпомага кохезията и улеснява замяната на цял слой. Многослойният стил се използва в операционни системи, в протоколни модели и в приложения с клиент-сървърна организация.

Недостатък е възможното влошаване на производителността поради преминаването през много слоеве и трудността да се приложи строго слоестата организация към всяка система. В някои случаи се използват вертикални слоеве или контролирани изключения от правилото.

27.4.2 Модел–изглед–контролер

Дефиниция 27.12. Модел–изглед–контролер, или MVC, е стил, който разделя данните и бизнес състоянието, представянето пред потребителя и обработката на потребителските действия.

Основните части са:

- *модел* – представлява знанията и състоянието на системата и управлява данните;
- *изглед* – управлява представянето на информацията пред потребителя;
- *контролер* – обработва потребителското взаимодействие, събитията и операциите между модела и изгледа.

Основната полза е независимостта между данните, тяхното представяне и потребителския контрол. Това улеснява поддръжката и разширяването, особено при клиент-сървърни уеб приложения. Стилът може да бъде излишно сложен за малки приложения. При чести промени в модела може да се появят проблеми с производителността, които могат да се ограничат чрез индексиране и кеширане на заявки към базата данни.

27.4.3 Поточна архитектура

Дефиниция 27.13. При стила Pipe-and-Filter системата е съставена от филтри, които обработват данни, и тръби, които пренасят резултата към следващия филтър.

Всеки филтър получава вход, преобразува го и подава изход към следващия филтър. Името произлиза от употребата на тръби в UNIX, където изходът на една команда може да бъде вход за друга.

Възможни са няколко варианта:

- batch-sequential – изходът на един филтър е вход за един следващ филтър;
- parallelism – един изход се подава към няколко филтъра, които работят паралелно;
- loopback – филтър е свързан обратно към по-ранен филтър.

Стилът е интуитивен и улеснява повторната употреба на филтрите. Той е подходящ за последователна обработка на потоци от данни. Недостатък може да бъде по-ниската производителност поради необходимостта от преобразуване или анализиране на данните между филтрите.

† **Допълнение 27.14 (Обхват: стиловете по-долу са допълнителни).** Конспектът назовава поименно само четири стила: многослоен, модел–изглед–контролер, поточен и ориентиран към услуги (§27.5). Клиент-сървърният стил, споделените данни и неявното извикване са добавени за пълнота на картината и за сравнение; на изпита те не са част от изисквания минимум.

27.4.4 Клиент-сървърна архитектура

Дефиниция 27.15 (Клиент-сървърен стил)**.** При клиент-сървърния стил системата е организирана като множество клиенти и сървъри. Сървърите предоставят услуги, а клиентите ги използват. Тънкият клиент предоставя основно потребителски интерфейс, изпраща заявки и визуализира резултата. Богатият клиент изпълнява и част от приложната обработка.

Често се използва трислойна организация:

1. слой на представянето;
2. слой на приложната логика;
3. слой за обработка и съхраняване на данните.

Предимствата са централизирано управление на данните и услугите, подобрена сигурност, лесно архивиране и възстановяване и възможност за отдалечено администриране. Сървърът обаче може да се превърне в критична точка за отказ, претоварване или атака. Затова са необходими подходящи тактики за резервиране и управление на натоварването.

27.4.5 Споделени данни

Дефиниция 27.16 (Стил със споделени данни)**.** При стила със споделени данни един компонент отговаря за общ източник на данни, например СУБД, а останалите компоненти извършват изчисления и използват този източник.

При Repository данните се споделят пасивно и компонентите сами ги извличат. При Blackboard публикуването на данни към общия компонент уведомява останалите компоненти.

Стилът позволява работа с големи количества данни, централизирано управление и паралелно изпълнение. Недостатъците са трудностите при разпределено изпълнение и превръщането на общото хранилище в рискова точка за откази, забавяния или атаки.

27.4.6 Неявно извикване и предаване на съобщения

Дефиниция 27.17 (Неявно извикване)**.** При неявното извикване компонентите не извикват директно функции или методи на други компоненти. Те създават и получават събития. Обикновено събитията се предават чрез общ буфер или event bus.

Стилът е известен като Publish–Subscribe или архитектура, управлявана от събития. Той осигурява слабо свързване: компонентите могат да бъдат разнородни и лесно заменяеми. Това го прави подходящ за разпределени системи и системи, управлявани от събития. Producer–Consumer е различен модел за координация чрез буфер или опашка; той може да се реализира със съобщения, но не е синоним на неявното извикване.

Недостатъците са по-трудното проследяване и отстраняване на грешки, неяснотата дали дадено събитие ще бъде обработено и трудностите при пренасяне на сложни данни. Отказът на споделения буфер може да засегне цялата система.

27.5 Архитектура, ориентирана към услуги

Дефиниция 27.18. Уеб услуга е софтуерна система, която предоставя достъп от и до други софтуерни услуги чрез мрежова комуникация.

Архитектурата, ориентирана към услуги, организира системата като услуги с ясно дефинирани интерфейси. Всяка услуга предоставя определена възможност и може да бъде използвана от други части на системата чрез стандартизиран механизъм за комуникация.

Монолитът е стил, при който компонентите са силно зависими, свързани са в едно приложение и се изпълняват като една услуга. При микроуслугите отделните услуги са относително независими, комуникират чрез интерфейсите си и всяка е организирана около конкретна бизнес потребност.

Микроуслугите могат да подобрят скалируемостта, адаптируемостта и възможността за използване на различни технологии, тъй като отделните компоненти могат да се разработват и разполагат независимо. Това обаче увеличава значението на мрежовата комуникация, разпределените откази и управлението на съгласуваността между услугите.

Свързани с ориентираните към услуги архитектури са клиент-сървърният стил и някои облачни стилове, например опашка, кеш и sharding. При опашката заявките се съхраняват като задания и се обработват според определен ред или приоритет. Кешът съхранява резултати от скорошни заявки, за да се избегне повторното им изчисляване. При sharding голям набор от данни се разделя хоризонтално между няколко хранилища.

При sharding данните обикновено се разделят според стойността на избран атрибут. Този атрибут се нарича ключ за sharding и трябва да бъде достатъчно стабилен. Предимствата са намаляване на натоварването на отделно хранилище и възможност за географско разпределение. Недостатъците включват труден избор на ключ, неравномерно разпределение и допълнително време за намиране на правилното хранилище.

27.6 Проектиране на софтуерната архитектура

27.6.1 Процес на проектиране

Проектирането на архитектура е цикличен процес. Обикновено се извличат изискванията, определят се най-важните архитектурни драйвери, предлага се архитектурно решение и се анализира влиянието му върху системата и средата. Получените резултати могат да променят разбирането за изискванията, поради което стъпките се повтарят.

Архитектурни драйвери са изискванията и ограниченията с най-голямо влияние върху архитектурата. Те включват особено важни качествени сценарии, бизнес цели, технологични ограничения и рискови функционалности.

Подходяща формална процедура е Attribute-Driven Design, или ADD:

1. избира се модул за декомпозиция, за който са известни изискванията;
2. модулът се детайлизира чрез избор на архитектурни драйвери и архитектурен стил или модел;
3. създават се подмодули, определят се техните интерфейси и функционалности;
4. определят се и други необходими структури, например процесни или разположенчески;
5. проверява се дали избраните модули изпълняват функционалните изисквания и дали са спазени всички качествени изисквания и ограничения;
6. стъпките се повтарят за всички модули, които се нуждаят от допълнителна декомпозиция.

ADD задължително води до декомпозиция на модулите. След оформяне на първите нива могат да се създадат екипи и да се започне работа по отделните части. Добра практика е първоначално да се изгради скелет на системата с интерфейси и стъбове, след което да се реализират компонентите за комуникация и изпълнение. Най-рисковите части могат да бъдат реализирани рано, за да се провери архитектурната осъществимост.

27.6.2 Избор на структури

Не всички структури са еднакво важни за всяка система. Изборът зависи от заинтересованите лица, качествените изисквания и етапа на разработката.

Модулните структури са подходящи за описание на отговорностите, зависимостите и разпределението на работата. Структурите на процесите са важни при паралелизъм, производителност и комуникация по време на изпълнение. Структурите на разположението са необходими при разпределени системи и при планиране на внедряването.

Практически изборът може да се направи чрез:

1. съставяне на таблица с приоритетите на заинтересованите лица относно различните изгледи;
2. комбиниране на близки перспективи, когато това намалява броя на изгледите без загуба на информация;
3. задаване на приоритети на избраните перспективи;
4. документиране на структурите с най-голямо значение за проекта.

27.6.3 Архитектурен анализ

След проектирането архитектурата трябва да бъде анализирана, за да се провери дали ще доведе до система, удовлетворяваща изискванията.

ATAM, или Architecture Tradeoff Analysis Method, оценява архитектурата чрез качествените сценарии и компромисите между качествата. Една архитектурна тактика може да подобри производителността, но да усложни поддръжката или да увеличи разходите. ATAM подпомага откриването на такива зависимости.

CBAM се основава на ATAM и добавя технико-икономически анализ. Целта е да се съпоставят ползите от архитектурните решения с разходите за тяхното реализиране, така че да се увеличат ползите и да се ограничат разходите за производство и поддръжка.

Архитектурата може да бъде визуализирана чрез графичен език за описание на архитектури или чрез архитектурен език, който поддържа преобразуване към графично представяне. Като примери за ADL в източниците са посочени ACME и Xtext.

27.7 Тактики за постигане на качествени атрибути

Дефиниция 27.19. Тактика е архитектурно решение, чрез което се контролира резултатът от сценарий за качество.

Наборът от използвани тактики образува архитектурна стратегия. Тактиките не са независими от контекста. Изборът им трябва да отчита компромисите между различните качествени атрибути.

27.7.1 Тактики за изправност и надеждност

Тактиките за откриване на откази включват:

- *ping/echo* – компонент изпраща сигнал към друг компонент и очаква отговор в определен интервал;
- *heartbeat/keepalive* – компонент периодично излъчва сигнал, който друг компонент очаква;
- изключения – компонент сигнализира проблем чрез изключение, което се обработва на същото или на по-високо ниво.

За предотвратяване или ограничаване на откази се използват:

- активен излишък – резервните компоненти поддържат същото състояние като основния и могат почти незабавно да го заместят;
- пасивен излишък – резервният компонент получава информация за промените и се актуализира при необходимост;
- резервни ресурси – допълнителни мощности, които се стартират при отказ;
- извеждане от употреба – компонентът временно или периодично се премахва, за да се избегне отказът му;

- наблюдение на процесите – специален процес открива отказал процес и го премахва, рестартира или замества.

За повторно въвеждане в употреба се използват паралелна работа и контролни точки. Повреденият компонент може да работи паралелно известно време, преди отново да стане основен. Контролната точка е запис на консистентно състояние, към което системата може да се върне след изпадане в неконсистентно състояние.

27.7.2 Тактики за производителност

Целта е системата да реагира на събитие в определен интервал. Тактиките се разделят на три групи.

Намаляване на изискванията към ресурсите се постига чрез подобряване на алгоритмите, избягване на ненужни изчисления, разреждане на периодичните операции, ограничаване на времето за изпълнение и използване на опашки с краен размер. Когато опашката се запълни, новите заявки могат да бъдат отхвърлени.

Управлението на ресурсите включва паралелна обработка, използване на допълнителни ресурси при нужда и разпределяне на излишни данни или процеси. Арбитражът на ресурсите определя коя заявка да бъде обработена с предимство при недостиг. Възможни стратегии са FIFO, фиксиран или динамичен приоритет и предварително планиране на изпълнението.

27.7.3 Тактики за изменяемост

Изменяемостта се подобрява чрез:

- локализиране на промените – намаляване на броя модули, които трябва да бъдат променени;
- групиране на семантично свързани функционалности в един модул;
- предвиждане на вероятни промени и ограничаване на възможните варианти;
- предотвратяване на ефекта на вълната – промяната да засяга само действително зависимите модули;
- ограничаване на комуникациите между модулите;
- използване на адаптери и фасади вместо промяна на множество съществуващи модули;
- отлагане на свързването до момента, когато са известни необходимите конкретни решения.

27.7.4 Тактики за сигурност

Тактиките за сигурност се групират в устояване на атаки, откриване на атаки и възстановяване след атаки.

Устояването включва автентикация, оторизация, криптиране, проверки чрез хешове или контролни суми, ограничаване на публичния интерфейс и ограничаване на достъпа чрез подходящи мрежови и физически механизми.

Откриването на атаки може да се реализира чрез система за откриване на проникванията, която анализира трафика или действията върху данните.

Възстановяването след атаки включва възстановяване на състоянието и идентифициране на извършителя. За тази цел се поддържа audit trail – запис на транзакциите заедно с идентификатора на извършилия ги потребител или компонент.

27.7.5 Тактики за тестируемост и използваемост

Тестируемостта се подобрява чрез специализирани интерфейси за тестване и вградени модули за мониторинг. Специализираният интерфейс може да предоставя допълнителни метаданни, които не се показват на обикновения потребител.

Тактиките за използваемост по време на изпълнение предвиждат възможните действия на потребителя и го насочват към правилните следващи стъпки. Тактиките, свързани с потребителския интерфейс, ограничават неподходящите действия, валидират входа и използват специализирани полета и форми.

27.8 Документиране на софтуерната архитектура

27.8.1 Предназначение

Дори добре проектираната архитектура е практически безполезна, ако не е документирана правилно и разбираемо. Документацията подпомага разработчиците, тестерите, ръководителите, администраторите и всички други заинтересовани лица, които трябва да разбират или променят системата.

Документацията трябва:

- да бъде съобразена с предназначението и читателя;
- да бъде достатъчно абстрактна, за да представя архитектурните решения;
- да съдържа достатъчно детайли, за да се избегнат двусмислици;
- да бъде последователна и поддържана при промяна на архитектурата.

Един документ не може да удовлетвори всички заинтересовани лица. За едни читатели архитектурата е предписание за разработка, а за други – описание на съществуваща система. Поради това архитектурната документация обикновено представлява набор от взаимосвързани документи.

Основният принцип е първо да се документират отделните структури, а след това да се добави информацията, която обединява няколко структури. Така сложният проблем се разделя на по-малки задачи: избор на структури, описание на елементите и връзките им, а след това описание на съответствията между изгледите.

27.8.2 Основни елементи на документацията

За всяка структура документацията може да съдържа:

1. *първично представяне* – графично или текстово представяне на елементите и най-важната информация;
2. *описание на елементите и връзките* – смисъл, роля, интерфейси и зависимости;
3. *описание на обкръжението* – взаимодействия с други системи, интерфейси и протоколи;
4. *описание на възможните варианти* – алтернативи и условията за избор между тях;
5. *архитектурна обосновка* – причините за избора, предположенията и аналитичните резултати;
6. *терминологичен речник* – стандартните и въведените за проекта термини;
7. *допълнителна информация* – всичко необходимо за по-доброто разбиране на архитектурата, включително автори и история на промените.

Първичното представяне може да бъде диаграма или текстово описание. В него се показват елементите и основните връзки, без да се претоварва читателят с детайли. Следващите части уточняват интерфейсите, ролите и взаимодействията.

Документацията на цялостната архитектура трябва да съдържа и каталог на структурите, шаблон за описанието им, кратко описание на системата, карта на съответствията между структурите, индекс на софтуерните елементи и разширен терминологичен речник. В нея трябва да бъде ясно защо архитектурата е проектирана по избрания начин и как решенията са свързани с качествените изисквания.

27.9 Изпитен фокус

За устен изпит трябва да могат да бъдат възпроизведени:

- определението за софтуерна архитектура, структура, изглед, компонент и конектор;
- трите групи структури: модулни, процесни и структури на разположението;
- разновидностите на модулните структури и ролята на декомпозицията;
- разликата между функционални и нефункционални изисквания;
- определението за качествен атрибут и шестте компонента на сценария за качество;
- основните технологични, бизнес и архитектурни качествени атрибути;
- определението за архитектурен стил и ограниченията, които той задава;
- многослойният стил, MVC и Pipe-and-Filter с техните предимства и ограничения;

- клиент-сървърният стил, споделените данни и неявното извикване;
- разликата между монолитна архитектура и архитектура с микроуслуги;
- цикличният процес на проектиране и стъпките на ADD;
- предназначението на ATAM и CBAM;
- определението за архитектурна тактика и връзката между тактика и архитектурна стратегия;
- тактики за надеждност, производителност, изменяемост, сигурност, тестируемост и използваемост;
- предназначението на архитектурната документация и основните елементи в описанието на структура;
- принципът за отделно документиране на структурите и последващо обединяване на изгледите.

Част III

Математика и приложения

Въпрос 28

Уравнения на права в равнината.

По конспекта: какво трябва да мога

Уравнения: векторно-параметрично и координатни (скаларни) параметрични уравнения на права — §28.1, стр. 395.
Теорема с доказателство: за общото уравнение на права в равнината — §28.2, стр. 396.
Класификация: взаимни положения между две прави в равнината — §28.3, стр. 398.
Уравнение: декартово уравнение на права — §28.4, стр. 399.
Уравнение и формула с извеждане: нормално уравнение и разстояние от точка до права — §28.5, стр. 400.
Описание: полуравнини — §28.6, стр. 402.
Литература: [20].

28.1 Параметрични уравнения на права

Нека в афинната равнина е фиксирана афинна координатна система $K = Oxy$. Права g се определя еднозначно от точка M_0 от нея и ненулев вектор $\vec{p}$, колинеарен на правата. Векторът $\vec{p}$ се нарича направляващ вектор на g.

Дефиниция 28.1. Нека $M_0 \in g$ и $\vec{p} \neq \vec{0}$, $\vec{p} \parallel g$. Векторното параметрично уравнение на правата g е

$$\vec{r} = \vec{r}_0 + s\vec{p}, \qquad s \in \mathbb{R},$$

където $\vec{r} = \overrightarrow{OM}$ е радиус-векторът на текущата точка M, а $\vec{r}_0 = \overrightarrow{OM_0}$ е радиус-векторът на фиксираната точка M_0.

Действително,

$$M \in g \iff \overrightarrow{M_0M} \parallel \vec{p} \iff \exists s \in \mathbb{R} : \overrightarrow{M_0M} = s\vec{p}.$$

Понеже

$$\overrightarrow{M_0M} = \overrightarrow{OM} - \overrightarrow{OM_0} = \vec{r} - \vec{r}_0,$$

получаваме векторното параметрично уравнение.

Дефиниция 28.2. Нека $M_0(x_0, y_0)$ и $\vec{p} = (p_1, p_2) \neq (0, 0)$. Координатните (скаларни) параметрични уравнения на правата през M_0, колинеарна на $\vec{p}$, са

$$g: \quad \begin{cases} x = x_0 + sp_1, \\ y = y_0 + sp_2, \end{cases} \qquad s \in \mathbb{R}.$$

Параметърът s задава положението на точката върху правата. При $s = 0$ се получава началната точка M_0. Различни параметризации могат да задават една и съща права: може да се избере друга точка от правата или ненулев колинеарен на $\vec{p}$ вектор.

▷ **Пример 28.3.** Правата през $M_0(2, -1)$ с направляващ вектор $\vec{p} = (3, 2)$ има уравнения

$$\vec{r} = (2, -1) + s(3, 2), \qquad s \in \mathbb{R},$$

съответно

$$\begin{cases} x = 2 + 3s, \\ y = -1 + 2s, \end{cases} \qquad s \in \mathbb{R}.$$

Например при $s = 1$ се получава точката $(5, 1)$.

28.2 Общо уравнение на права

★ **Теорема 28.4 (Теорема за общото уравнение на права).** Всяка права в равнината има уравнение от вида

$$Ax + By + C = 0, \qquad (A, B) \neq (0, 0).$$

Обратно, всяко уравнение от този вид определя точно една права в равнината.

Доказателство. Нека правата g минава през $M_0(x_0, y_0)$ и има направляващ вектор $\vec{p} = (p_1, p_2) \neq (0, 0)$. Точка $M(x, y)$ лежи върху g тогава и само тогава, когато векторите

$$\overrightarrow{M_0M} = (x - x_0, y - y_0) \quad \text{и} \quad \vec{p} = (p_1, p_2)$$

са колинеарни. Координатното условие за колинеарност е

$$\begin{vmatrix} x - x_0 & y - y_0 \\ p_1 & p_2 \end{vmatrix} = 0.$$

Следователно

$$p_2(x - x_0) - p_1(y - y_0) = 0,$$

тоест

$$p_2x - p_1y - p_2x_0 + p_1y_0 = 0.$$

При полагането

$$A = p_2, \qquad B = -p_1, \qquad C = -p_2x_0 + p_1y_0$$

получаваме уравнение от търсения вид. Понеже $\vec{p} \neq \vec{0}$, то $(A, B) = (p_2, -p_1) \neq (0, 0)$.

Обратно, нека е дадено

$$Ax + By + C = 0, \qquad (A, B) \neq (0, 0).$$

Нека $M_0(x_0, y_0)$ е негова точка. Разглеждаме вектора

$$\vec{p} = (-B, A) \neq \vec{0}.$$

Съществува единствена права през M_0, колинеарна на $\vec{p}$. Нейните параметрични уравнения са

$$x = x_0 - sB, \qquad y = y_0 + sA, \qquad s \in \mathbb{R}.$$

Оттук

$$A(x - x_0) + B(y - y_0) = 0.$$

Тъй като $Ax_0 + By_0 + C = 0$, последното уравнение е еквивалентно на $Ax + By + C = 0$. Следователно даденото уравнение задава точно тази права. □

Дефиниция 28.5. Уравнението

$$g: \ Ax + By + C = 0, \qquad (A, B) \neq (0, 0),$$

се нарича общо уравнение на правата g спрямо координатната система $K = Oxy$.

Коефициентите на общото уравнение дават два важни вектора:

$$\vec{n} = (A, B) \perp g, \qquad \vec{p} = (-B, A) \parallel g.$$

Векторът $\vec{n}$ се нарича нормален вектор на правата, а $\vec{p}$ е един от нейните направляващи вектори.

Твърдение 28.6. Нека

$$g: \ Ax + By + C = 0, \qquad (A, B) \neq (0, 0).$$

Векторът $\vec{q} = (q_1, q_2)$ е колинеарен на g тогава и само тогава, когато

$$Aq_1 + Bq_2 = 0.$$

Доказателство. Направляващ вектор на g е $\vec{p} = (-B, A)$. Условието $\vec{q} \parallel \vec{p}$ е

$$\begin{vmatrix} -B & A \\ q_1 & q_2 \end{vmatrix} = 0,$$

което е равносилно на $Aq_1 + Bq_2 = 0$. □

▷ **Пример 28.7.** Да се намери общото уравнение на правата през $M_0(1, 2)$ с направляващ вектор $\vec{p} = (4, -3)$.
От доказателството на теоремата получаваме

$$A = p_2 = -3, \qquad B = -p_1 = -4,$$

а

$$C = -p_2x_0 + p_1y_0 = 3 + 8 = 11.$$

Следователно

$$-3x - 4y + 11 = 0,$$

или еквивалентно

$$3x + 4y - 11 = 0.$$

28.3 Взаимно положение на две прави

Нека са дадени правите

$$g_1 : \ A_1x + B_1y + C_1 = 0, \qquad g_2 : \ A_2x + B_2y + C_2 = 0,$$

където

$$(A_i, B_i) \neq (0, 0), \qquad i = 1, 2.$$

Техните общи точки са решенията на системата

$$\begin{cases} A_1x + B_1y + C_1 = 0, \\ A_2x + B_2y + C_2 = 0. \end{cases}$$

Определителят на системата по неизвестните е

$$\Delta = \begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix} = A_1B_2 - A_2B_1.$$

★ **Теорема 28.8.** За взаимното положение на g_1 и g_2 са в сила следните критерии.

1. Правите се пресичат в точно една точка тогава и само тогава, когато
$$\Delta \neq 0.$$
2. Правите съвпадат тогава и само тогава, когато съществува $k \neq 0$, за което
$$A_1 = kA_2, \qquad B_1 = kB_2, \qquad C_1 = kC_2.$$
3. Правите са различни успоредни тогава и само тогава, когато съществува $k \neq 0$, за което
$$A_1 = kA_2, \qquad B_1 = kB_2, \qquad C_1 \neq kC_2.$$

Доказателство. Ако $\Delta \neq 0$, системата има единствено решение, следователно правите имат точно една обща точка.
Ако $\Delta = 0$, двойките (A_1, B_1) и (A_2, B_2) са пропорционални. Тогава или и свободните членове са пропорционални със същия коефициент, което прави двете уравнения еквивалентни, или не са. В първия случай правите съвпадат, а във втория системата е несъвместима и правите са различни успоредни. □

Забележка 28.9. Записите с отношения, например

$$\frac{A_1}{A_2} = \frac{B_1}{B_2},$$

са удобни само когато съответните знаменатели са различни от нула. Критериите чрез общ ненулев множител k или чрез определителя Δ нямат това ограничение.

† **Допълнение 28.10 (Бележка).** Среща се неточен критерий, според който пресичането се описва с неравенства между всички отношения на коефициентите. Точният критерий за пресичане е

$$A_1B_2 - A_2B_1 \neq 0.$$

Свободните членове C_1, C_2 определят положението на пресечната точка, но не влияят на това дали две прави с различни направления се пресичат.

▷ **Пример 28.11.** За правите

$$g_1 : \ 2x - y + 1 = 0, \qquad g_2 : \ 4x - 2y - 3 = 0$$

е изпълнено

$$\Delta = \begin{vmatrix} 2 & -1 \\ 4 & -2 \end{vmatrix} = 0.$$

Първите два коефициента на g_2 са два пъти съответните коефициенти на g_1, но свободният член -3 не е два пъти 1. Следователно правите са различни успоредни.

28.4 Декартово уравнение

Оттук нататък при метричните понятия приемаме, че $K = Oxy$ е ортонормирана координатна система.

Дефиниция 28.12. Нека

$$g : \ Ax + By + C = 0, \qquad B \neq 0.$$

След изразяване на y получаваме

$$y = -\frac{A}{B}x - \frac{C}{B}.$$

Ако

$$k = -\frac{A}{B}, \qquad n = -\frac{C}{B},$$

то уравнението

$$g : \ y = kx + n$$

се нарича декартово уравнение на правата g.

Коефициентът k се нарича ъглов коефициент на правата. Ако ориентираният ъгъл между положителната посока на Ox и правата е α, то

$$k = \tan\alpha.$$

Един направляващ вектор на правата $y = kx + n$ е $(1, k)$.

Твърдение 28.13. Права има декартово уравнение $y = kx + n$ тогава и само тогава, когато не е успоредна на координатната ос Oy.

Доказателство. Декартовото уравнение се получава точно когато $B \neq 0$. Ако $B = 0$, общото уравнение има вид $Ax+C = 0$, тоест описва права, успоредна на Oy. Обратно, когато $B \neq 0$, y се изразява еднозначно чрез x. □

Забележка 28.14. Правата $y = kx+n$ пресича оста Oy в точката $(0, n)$. Тя пресича оста Ox, ако $k \neq 0$, в точката

$$\left(-\frac{n}{k}, 0\right).$$

† **Допълнение 28.15 (Бележка).** В някои материали точката $(0, n)$ е означена като пресечна точка с Ox. Това е несъответствие: при уравнение $y = kx + n$ тази точка лежи на оста Oy.

▷ **Пример 28.16.** Правата

$$3x - 2y + 6 = 0$$

има декартово уравнение

$$y = \frac{3}{2}x + 3.$$

Тя има ъглов коефициент $k = \frac{3}{2}$ и пресича Oy в точката $(0, 3)$.

28.5 Нормално уравнение и разстояние от точка до права

Нека

$$g : \ Ax + By + C = 0, \qquad (A, B) \neq (0, 0).$$

Нормален вектор на правата е $\vec{n} = (A, B)$, а неговата дължина е

$$|\vec{n}| = \sqrt{A^2 + B^2}.$$

Следователно един от двата единични нормални вектора е

$$\vec{n}_1 = \left(\frac{A}{\sqrt{A^2 + B^2}}, \frac{B}{\sqrt{A^2 + B^2}}\right).$$

Дефиниция 28.17. Нормално уравнение на правата g се нарича общо уравнение на същата права, при което коефициентите пред x и y са координати на единичен нормален вектор. Нормалните уравнения на g са

$$\frac{Ax + By + C}{\sqrt{A^2 + B^2}} = 0 \qquad \text{и} \qquad -\frac{Ax + By + C}{\sqrt{A^2 + B^2}} = 0.$$

Следователно всяка права има точно две нормални уравнения. Те се получават чрез двата противоположно насочени единични нормални вектора.

★ **Теорема 28.18 (Разстояние от точка до права).** Нека

$$g:\ Ax+By+C=0, \qquad (A,B)\neq(0,0),$$

и $P(x_1,y_1)$ е точка в равнината. Тогава разстоянието от P до g е

$$d(P,g)=\frac{|Ax_1+By_1+C|}{\sqrt{A^2+B^2}}.$$

Доказателство. Нека нормалното уравнение на g е

$$A_1x+B_1y+C_1=0,$$

където $A_1^2+B_1^2=1$. Нека $H(x_H,y_H)$ е ортогоналната проекция на P върху g. Тогава $\overrightarrow{HP}$ е колинеарен на единичния нормален вектор (A_1,B_1), тоест за точно едно $\delta\in\mathbb{R}$ е изпълнено

$$\overrightarrow{HP}=\delta(A_1,B_1).$$

Следователно

$$x_H=x_1-\delta A_1, \qquad y_H=y_1-\delta B_1.$$

Тъй като $H\in g$, имаме

$$A_1x_H+B_1y_H+C_1=0.$$

След заместване и използване на $A_1^2+B_1^2=1$ получаваме

$$\delta=A_1x_1+B_1y_1+C_1.$$

Числото δ е ориентираното разстояние, а търсеното разстояние е неговата абсолютна стойност. След заместване на A_1,B_1,C_1 се получава посочената формула. □

Забележка 28.19. Ако уравнението на правата вече е нормално, тоест

$$A^2+B^2=1,$$

формулата се опростява до

$$d(P,g)=|Ax_1+By_1+C|.$$

Без абсолютна стойност изразът

$$\delta(P,g)=\frac{Ax_1+By_1+C}{\sqrt{A^2+B^2}}$$

е ориентираното разстояние, определено от избрания единичен нормален вектор.

▷ **Пример 28.20.** Да се намери разстоянието от $P(1,-2)$ до правата

$$g:\ 3x+4y-5=0.$$

Изчисляваме

$$d(P,g)=\frac{|3\cdot1+4\cdot(-2)-5|}{\sqrt{3^2+4^2}}=\frac{|-10|}{5}=2.$$

28.6 Полуравнини

Всяка права разделя равнината на две части, наречени отворени полуравнини. Нека

$$g:\ Ax + By + C = 0, \qquad (A, B) \neq (0, 0),$$

и въведем линейната функция

$$l(x, y) = Ax + By + C.$$

За точките от правата е изпълнено $l(x, y) = 0$. За всяка точка извън правата стойността на $l(x, y)$ е или положителна, или отрицателна.

★ **Теорема 28.21.** Нека $M_1(x_1, y_1)$ и $M_2(x_2, y_2)$ не лежат на правата g. Те лежат в различни полуравнини спрямо g тогава и само тогава, когато

$$l(x_1, y_1)\, l(x_2, y_2) < 0.$$

Доказателство. Нека първо M_1 и M_2 са в различни полуравнини. Тогава отсечката M_1M_2 пресича g в точка $M_0(x_0, y_0)$, която е вътрешна за отсечката. Следователно съществува $k < 0$, за което

$$\overrightarrow{M_0M_1} = k\overrightarrow{M_0M_2}.$$

Оттук

$$x_0 = \frac{x_1 - kx_2}{1 - k}, \qquad y_0 = \frac{y_1 - ky_2}{1 - k}.$$

Тъй като $M_0 \in g$, след заместване в $l(x_0, y_0) = 0$ получаваме

$$l(x_1, y_1) - k\, l(x_2, y_2) = 0.$$

Следователно

$$k = \frac{l(x_1, y_1)}{l(x_2, y_2)} < 0,$$

откъдето

$$l(x_1, y_1) l(x_2, y_2) < 0.$$

Обратно, нека

$$l(x_1, y_1) l(x_2, y_2) < 0.$$

Тогава

$$k = \frac{l(x_1, y_1)}{l(x_2, y_2)} < 0.$$

Точката

$$M_0\left(\frac{x_1 - kx_2}{1 - k}, \frac{y_1 - ky_2}{1 - k}\right)$$

удовлетворява $l(x_0, y_0) = 0$, следователно лежи на g. Освен това

$$\overrightarrow{M_0M_1} = k\overrightarrow{M_0M_2}, \qquad k < 0.$$

Векторите са противопосочни, значи M_0 е вътрешна точка на отсечката M_1M_2. Следователно M_1 и M_2 са в различни полуравнини. □

Дефиниция 28.22. Двете отворени полуравнини с граница g се задават аналитично чрез

$$\lambda_g = \{M(x,y) : Ax + By + C > 0\}$$

и

$$\overline{\lambda}_g = \{M(x,y) : Ax + By + C < 0\}.$$

Знакът на $l(x,y)$ е постоянен във всяка от двете полуравнини. Това дава практичен начин за проверка от коя страна на права се намира дадена точка.

▷ **Пример 28.23.** Нека

$$g : \ 2x - y - 1 = 0.$$

За точките $P(0,0)$ и $Q(2,0)$ имаме

$$l(0,0) = -1, \qquad l(2,0) = 3.$$

Техният произведение е отрицателно, следователно P и Q лежат в различни полуравнини спрямо g.

28.7 Изпитен фокус

- Да се дефинира права чрез точка M_0 и ненулев направляващ вектор $\vec{p}$, както и да се запише векторно-параметричното уравнение

$$\vec{r} = \vec{r_0} + s\vec{p}.$$

- Да се преминава към координатни параметрични уравнения

$$x = x_0 + sp_1, \qquad y = y_0 + sp_2.$$

- Да се формулира и докаже теоремата за общото уравнение

$$Ax + By + C = 0, \qquad (A,B) \neq (0,0),$$

в двете посоки, чрез условието за колинеарност и вектора $(-B,A)$.

- Да се разпознават нормален вектор (A,B) и направляващ вектор $(-B,A)$ на права с общо уравнение.

- Да се определя взаимното положение на две прави чрез

$$\Delta = A_1B_2 - A_2B_1$$

и чрез пропорционалността на тройките коефициенти.

- Да се извежда декартовото уравнение

$$y = kx + n, \qquad k = -\frac{A}{B}, \qquad n = -\frac{C}{B},$$

при $B \neq 0$, и да се обясни защо правите, успоредни на Oy, нямат такава форма.

- Да се нормализира общо уравнение на права и да се запишат двете нормални уравнения.
- Да се възпроизвежда формулата

  $$d(P, g) = \frac{|Ax_1 + By_1 + C|}{\sqrt{A^2 + B^2}}$$

  и идеята за доказателството чрез ортогоналната проекция на точката върху правата.
- Да се описват полуравнините чрез знака на $l(x, y) = Ax + By + C$ и да се прилага критерият

  $$l(x_1, y_1)l(x_2, y_2) < 0$$

  за точки, разположени в различни полуравнини.

Въпрос 29

Уравнения на права и равнина в пространството.

По конспекта: какво трябва да мога

Уравнения: векторно-параметрично и координатни (скаларни) параметрични уравнения на равнина — §29.1, стр. 405.
Теорема с доказателство: за общото уравнение на равнина — §29.2, стр. 406.
Класификация: взаимни положения между две равнини — §29.3, стр. 409.
Уравнение и формула с извеждане: нормално уравнение и разстояние от точка до равнина — §29.4, стр. 411.
Описание: полупространства — §29.5, стр. 412.
Уравнения на права в пространството: параметрично представяне — §29.6.1; права като пресечница на две *непаралелни* равнини — §29.6.2, стр. 414.
Литература: [20].

29.1 Равнина в тримерното пространство

Нека $K = Oxyz$ е афинна координатна система в тримерното пространство. Равнината се определя еднозначно от точка, която принадлежи на нея, и два линейно независими вектора, компланарни с нея.

Дефиниция 29.1. Нека M_0 е фиксирана точка и $\vec{p}, \vec{q}$ са линейно независими вектора. Множеството от всички точки M, за които векторът $\overrightarrow{M_0M}$ е линейна комбинация на $\vec{p}$ и $\vec{q}$, се нарича равнина през M_0, успоредна на $\vec{p}$ и $\vec{q}$.

Линейната независимост на $\vec{p}$ и $\vec{q}$ е съществена: ако те са колинеарни, зададените данни определят права, а не равнина.

29.1.1 Векторно-параметрично и координатно параметрично уравнение

Нека

$$\vec{r}_0 = \overrightarrow{OM_0}, \qquad \vec{r} = \overrightarrow{OM},$$

където M е произволна точка. Тогава

$$\overrightarrow{M_0M} = \vec{r} - \vec{r}_0.$$

Точката M лежи в равнината α, определена от M_0, $\vec{p}$ и $\vec{q}$, точно тогава, когато

$$\overrightarrow{M_0M} = \lambda\vec{p} + \mu\vec{q}$$

за някои независими реални параметри λ, μ. Следователно

$$\boxed{\vec{r} = \vec{r}_0 + \lambda\vec{p} + \mu\vec{q}, \qquad (\lambda, \mu) \in \mathbb{R}^2.}$$

Това се нарича *векторно-параметрично уравнение на равнина*.

★ **Дефиниция 29.2.** Ако $M_0(x_0, y_0, z_0)$,

$$\vec{p} = (p_1, p_2, p_3), \qquad \vec{q} = (q_1, q_2, q_3),$$

и $\vec{p}, \vec{q}$ са линейно независими, то координатните параметрични уравнения на равнината α са

$$\boxed{\begin{cases} x = x_0 + \lambda p_1 + \mu q_1, \\ y = y_0 + \lambda p_2 + \mu q_2, \\ z = z_0 + \lambda p_3 + \mu q_3, \end{cases} \qquad \lambda, \mu \in \mathbb{R}.}$$

▷ **Пример 29.3.** Равнината, която минава през $M_0(1, -1, 2)$ и е успоредна на векторите

$$\vec{p} = (1, 2, 0), \qquad \vec{q} = (2, 0, 1),$$

има параметрични уравнения

$$\begin{cases} x = 1 + \lambda + 2\mu, \\ y = -1 + 2\lambda, \\ z = 2 + \mu, \end{cases} \qquad \lambda, \mu \in \mathbb{R}.$$

Векторите $\vec{p}$ и $\vec{q}$ не са колинеарни, следователно действително определят равнина.

29.2 Общо уравнение на равнина

29.2.1 Получаване от точка и два направляващи вектора

Нека равнината α е зададена чрез точка

$$M_0(x_0, y_0, z_0)$$

и линейно независими компланарни вектори

$$\vec{p} = (p_1, p_2, p_3), \qquad \vec{q} = (q_1, q_2, q_3).$$

За точка $M(x, y, z)$ условието $M \in \alpha$ е равносилно на компланарност на векторите

$$\overrightarrow{M_0M} = (x - x_0, y - y_0, z - z_0), \qquad \vec{p}, \qquad \vec{q}.$$

Координатното условие за компланарност е

$$\begin{vmatrix} x-x_0 & p_1 & q_1 \\ y-y_0 & p_2 & q_2 \\ z-z_0 & p_3 & q_3 \end{vmatrix} = 0.$$

След развиване по първата колона се получава

$$A(x-x_0)+B(y-y_0)+C(z-z_0)=0,$$

където

$$A=\begin{vmatrix} p_2 & q_2 \\ p_3 & q_3 \end{vmatrix}, \qquad B=\begin{vmatrix} p_3 & q_3 \\ p_1 & q_1 \end{vmatrix}, \qquad C=\begin{vmatrix} p_1 & q_1 \\ p_2 & q_2 \end{vmatrix}.$$

При

$$D=-Ax_0-By_0-Cz_0$$

уравнението придобива вида

$$Ax+By+Cz+D=0.$$

★ **Теорема 29.4 (Теорема за общото уравнение на равнина).** Спрямо всяка афинна координатна система всяка равнина има уравнение от вида

$$\boxed{Ax+By+Cz+D=0, \qquad (A,B,C)\neq(0,0,0).}$$

Обратно, всяко уравнение от този вид определя точно една равнина.

Доказателство. Първата част следва от условието за компланарност, разгледано по-горе. Ако едновременно $A=B=C=0$, всички съответни минори от втори ред биха били нулеви. Това би означавало, че $\vec{p}$ и $\vec{q}$ са линейно зависими, което противоречи на избора им.
За обратното твърдение нека

$$Ax+By+Cz+D=0, \qquad (A,B,C)\neq(0,0,0).$$

Уравнението има поне едно решение $M_0(x_0,y_0,z_0)$. Например, ако $A\neq 0$, могат да се изберат векторите

$$\vec{p}=(-B,A,0), \qquad \vec{q}=\left(-\frac{C}{A},0,1\right).$$

Те са линейно независими. Равнината през M_0, успоредна на тях, има уравнение, получено от условието за компланарност, и то е еквивалентно на

$$Ax+By+Cz+D=0,$$

където $D=-Ax_0-By_0-Cz_0$. Следователно даденото уравнение определя равнина, а множеството на неговите решения я определя еднозначно. □

Забележка 29.5. Умножаването на общото уравнение на равнина с число $k\neq 0$ не променя равнината:

$$Ax+By+Cz+D=0 \quad\Longleftrightarrow\quad kAx+kBy+kCz+kD=0.$$

Следователно една равнина има безброй общи уравнения.

29.2.2 Равнина през три точки

Три неколинеарни точки

$$M_1(x_1, y_1, z_1), \quad M_2(x_2, y_2, z_2), \quad M_3(x_3, y_3, z_3)$$

определят единствена равнина. За нея може да се използва точката M_1 и векторите

$$\overrightarrow{M_1M_2} = (x_2 - x_1, y_2 - y_1, z_2 - z_1),$$
$$\overrightarrow{M_1M_3} = (x_3 - x_1, y_3 - y_1, z_3 - z_1).$$

Те са линейно независими точно когато трите точки не са колинеарни. Уравнението на равнината е

$$\boxed{\begin{vmatrix} x - x_1 & x_2 - x_1 & x_3 - x_1 \\ y - y_1 & y_2 - y_1 & y_3 - y_1 \\ z - z_1 & z_2 - z_1 & z_3 - z_1 \end{vmatrix} = 0.}$$

▷ **Пример 29.6.** Да се намери равнината през точките

$$A(1, 0, 1), \qquad B(2, 1, 1), \qquad C(1, 1, 3).$$

Имаме

$$\overrightarrow{AB} = (1, 1, 0), \qquad \overrightarrow{AC} = (0, 1, 2).$$

Следователно

$$\begin{vmatrix} x - 1 & 1 & 0 \\ y & 1 & 1 \\ z - 1 & 0 & 2 \end{vmatrix} = 0.$$

След развиване получаваме

$$2x - 2y + z - 3 = 0.$$

29.2.3 Нормален вектор и условие за компланарност

Твърдение 29.7. Ако

$$\alpha : Ax + By + Cz + D = 0, \qquad (A, B, C) \neq (0, 0, 0),$$

то векторът

$$\vec{n}_\alpha = (A, B, C)$$

е перпендикулярен на равнината α и се нарича неин *нормален вектор*.

Доказателство. Нека P, Q лежат в α. Изваждаме уравненията им и получаваме $A(Q_x - P_x) + B(Q_y - P_y) + C(Q_z - P_z) = 0$. Следователно $(A, B, C) \cdot \overrightarrow{PQ} = 0$ за всеки направляващ вектор на равнината. Векторът (A, B, C) е ненулев по условие, следователно е нормален към равнината. □

Действително, ако $\vec{v} = (v_1, v_2, v_3)$ е компланарен с α, то той е ортогонален на нормалния вектор. Координатно това условие е

$$\boxed{Av_1 + Bv_2 + Cv_3 = 0.}$$

По този начин условието за компланарност на вектор с равнина може да се проверява чрез скаларно произведение.

29.3 Взаимни положения на две равнини

Нека са дадени равнините

$$\alpha_1 : A_1x + B_1y + C_1z + D_1 = 0,$$

$$\alpha_2 : A_2x + B_2y + C_2z + D_2 = 0,$$

където

$$A_i^2 + B_i^2 + C_i^2 \neq 0, \qquad i = 1, 2.$$

Техните нормални вектори са

$$\vec{n}_1 = (A_1, B_1, C_1), \qquad \vec{n}_2 = (A_2, B_2, C_2).$$

★ **Теорема 29.8.** Две равнини в пространството са в точно едно от следните взаимни положения:

1. съвпадат;
2. успоредни са и са различни;
3. пресичат се по права.

Доказателство. Ако нормалните вектори са пропорционални, след умножаване на едното уравнение лявата линейна част става еднаква. При еднакви свободни членове равнините съвпадат, а при различни нямат обща точка и са успоредни. Ако нормалите не са пропорционални, матрицата на двете уравнения има ранг 2, равен и на ранга на разширената матрица. Системата е съвместима с едно свободно неизвестно и решението е афинна права. Случаите са взаимно изключващи се и изчерпват възможностите. □

Твърдение 29.9. Равнините α_1 и α_2 съвпадат точно тогава, когато съществува $k \neq 0$, за което

$$(A_1, B_1, C_1, D_1) = k(A_2, B_2, C_2, D_2).$$

Еквивалентно,

$$\operatorname{rank} \begin{pmatrix} A_1 & B_1 & C_1 & D_1 \\ A_2 & B_2 & C_2 & D_2 \end{pmatrix} = 1.$$

Доказателство. Пропорционални уравнения имат точно едни и същи решения. Обратно, съвпадащите равнини имат едно и също двумерно направляващо пространство, чието ортогонално допълнение е едномерно. Затова ненулевите им нормали са пропорционални. Заместването на обща точка показва същата пропорционалност и на свободните членове. Това е еквивалентно на ранг 1 на матрицата с двата ненулеви реда. □

Твърдение 29.10. Равнините α_1 и α_2 са успоредни и различни точно тогава, когато нормалните им вектори са колинеарни, но коефициентите на двете уравнения не са пропорционални изцяло. Рангово:

$$\operatorname{rank}\begin{pmatrix} A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \end{pmatrix} = 1,$$

а

$$\operatorname{rank}\begin{pmatrix} A_1 & B_1 & C_1 & D_1 \\ A_2 & B_2 & C_2 & D_2 \end{pmatrix} = 2.$$

Доказателство. Колинеарните нормали дават еднакви направляващи пространства. След изравняване на линейните части непропорционалността на пълните уравнения означава различни свободни членове, следователно няма общо решение. Обратно, различни успоредни равнини имат колинеарни нормали, но уравненията им не са пропорционални, иначе биха съвпадали. Първото условие е ранг 1 на коефициентната матрица, а второто — ранг 2 на разширената. □

Твърдение 29.11. Равнините α_1 и α_2 се пресичат по права точно тогава, когато техните нормални вектори не са колинеарни, т.е.

$$\operatorname{rank}\begin{pmatrix} A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \end{pmatrix} = 2.$$

Доказателство. При независими нормали коефициентната матрица има ранг 2, както и разширената, понеже има само два реда. Общото решение има един свободен параметър и е права. Обратно, при колинеарни нормали равнините или съвпадат, или са различни и успоредни по вече доказаните критерии; сечението им не може да е точно права. □

† **Допълнение 29.12 (Бележка).** Условията с отношения от вида

$$\frac{A_1}{A_2} = \frac{B_1}{B_2} = \frac{C_1}{C_2}$$

са удобни само когато съответните знаменатели са ненулеви. Ранговите условия или формулировката чрез пропорционалност на векторите са приложими без това ограничение. За пресичащи се равнини не е коректно да се изисква всички такива отношения да са различни, защото някои коефициенти могат да са нулеви или отделни отношения да съвпадат.

▷ **Пример 29.13.** Да се определят взаимните положения на

$$\alpha : 2x - y + z - 3 = 0, \qquad \beta : 4x - 2y + 2z + 1 = 0.$$

Нормалните вектори са

$$\vec{n}_\alpha = (2, -1, 1), \qquad \vec{n}_\beta = (4, -2, 2) = 2\vec{n}_\alpha.$$

Следователно равнините са успоредни или съвпадат. Тъй като свободните членове не са в същото отношение:

$$1 \neq 2 \cdot (-3),$$

равнините са успоредни и различни.

29.4 Нормално уравнение на равнина и разстояние

Оттук нататък се работи в ортонормирана координатна система. Нека

$$\alpha : Ax + By + Cz + D = 0, \qquad (A, B, C) \neq (0, 0, 0).$$

Дължината на нормалния вектор е

$$|\vec{n}_\alpha| = \sqrt{A^2 + B^2 + C^2}.$$

Единичните нормални вектори са

$$\vec{n}_0 = \pm\frac{\vec{n}_\alpha}{|\vec{n}_\alpha|} = \pm\left(\frac{A}{\sqrt{A^2 + B^2 + C^2}}, \frac{B}{\sqrt{A^2 + B^2 + C^2}}, \frac{C}{\sqrt{A^2 + B^2 + C^2}}\right).$$

★ **Дефиниция 29.14.** Общото уравнение

$$Ax + By + Cz + D = 0$$

се нарича *нормално уравнение на равнината*, ако

$$A^2 + B^2 + C^2 = 1.$$

Всяка равнина има точно две нормални уравнения:

$$\boxed{\pm\frac{Ax + By + Cz + D}{\sqrt{A^2 + B^2 + C^2}} = 0.}$$

Двете уравнения съответстват на двата противоположно насочени единични нормални вектора.

★ **Теорема 29.15 (Разстояние от точка до равнина).** Нека

$$\alpha : Ax + By + Cz + D = 0$$

и $M_0(x_0, y_0, z_0)$ е точка. Тогава разстоянието от M_0 до α е

$$\boxed{d(M_0, \alpha) = \frac{|Ax_0 + By_0 + Cz_0 + D|}{\sqrt{A^2 + B^2 + C^2}}.}$$

Ако уравнението на равнината е нормално, формулата се свежда до

$$d(M_0, \alpha) = |Ax_0 + By_0 + Cz_0 + D|.$$

Доказателство. Нека H е ортогоналната проекция на M_0 върху α. Векторът $\overrightarrow{M_0H}$ е колинеарен на единичен нормален вектор $\vec{n}_0$, следователно

$$\overrightarrow{M_0H} = \delta\vec{n}_0$$

за някое $\delta \in \mathbb{R}$. Тогава

$$d(M_0, \alpha) = |\overrightarrow{M_0H}| = |\delta|.$$

След заместване на координатите на H в нормалното уравнение се получава

$$\delta = -\frac{Ax_0 + By_0 + Cz_0 + D}{\sqrt{A^2 + B^2 + C^2}},$$

с точност до избора на посока на единичния нормален вектор. Вземането на абсолютна стойност дава формулата. □

▷ **Пример 29.16.** Да се намери разстоянието от $P(1, 2, -1)$ до равнината

$$\alpha : 2x - y + 2z + 3 = 0.$$

Получаваме

$$d(P, \alpha) = \frac{|2 \cdot 1 - 2 + 2 \cdot (-1) + 3|}{\sqrt{2^2 + (-1)^2 + 2^2}} = \frac{1}{3}.$$

29.5 Полупространства

Нека равнината

$$\alpha : Ax + By + Cz + D = 0, \qquad (A, B, C) \neq (0, 0, 0),$$

е зададена в афинна координатна система. Нека означим

$$E(x, y, z) = Ax + By + Cz + D.$$

Дефиниция 29.17. Двете множества

$$\lambda_\alpha = \{M(x,y,z) : E(x,y,z) > 0\},$$

$$\overline{\lambda}_\alpha = \{M(x,y,z) : E(x,y,z) < 0\}$$

се наричат отворени полупространства с гранична равнина α.

Самата равнина е множеството от точки, за които $E(x,y,z) = 0$. Така пространството се разделя на двете полупространства и граничната им равнина.

★ **Теорема 29.18.** Нека $M_1(x_1,y_1,z_1)$ и $M_2(x_2,y_2,z_2)$ не лежат на α. Те са в различни полупространства спрямо α точно тогава, когато

$$\boxed{E(x_1,y_1,z_1)\,E(x_2,y_2,z_2) < 0.}$$

Доказателство. Ако M_1 и M_2 са в различни полупространства, отсечката M_1M_2 пресича равнината в точка M_0. Тъй като M_0 е вътрешна точка на отсечката, съществува $k < 0$, така че

$$\overrightarrow{M_0M_1} = k\overrightarrow{M_0M_2}.$$

От координатите на тази зависимост и от $E(M_0) = 0$ следва

$$E(M_1) - kE(M_2) = 0.$$

Следователно

$$k = \frac{E(M_1)}{E(M_2)} < 0,$$

откъдето $E(M_1)E(M_2) < 0$.
Обратно, ако $E(M_1)E(M_2) < 0$, то числото

$$k = \frac{E(M_1)}{E(M_2)}$$

е отрицателно. Точката, която дели правата M_1M_2 съгласно това отношение, удовлетворява $E(M_0) = 0$ и следователно лежи в α. Понеже $k < 0$, тя е вътрешна за отсечката M_1M_2. Следователно краищата на отсечката са в различни полупространства. □

Следствие 29.19. Две точки, нележащи на α, са в едно и също полупространство точно тогава, когато стойностите на E в тях имат един и същ знак.

Доказателство. Полупространствата, определени от α, са точно $\{P : E(P) > 0\}$ и $\{P : E(P) < 0\}$. Във всяко от тях отсечката между две точки остава вътре, защото $E((1-t)P + tQ) = (1-t)E(P) + tE(Q)$ има същия знак за $0 \le t \le 1$. При противоположни знаци този афинен израз се анулира за някое $0 < t < 1$, тоест отсечката пресича α. Това дава твърдения критерий. □

29.6 Уравнения на права в пространството

29.6.1 Векторно-параметрично и координатно параметрично уравнение

Нека M_0 е фиксирана точка и $\vec{p} \neq \vec{0}$ е фиксиран вектор. Съществува точно една права g, която минава през M_0 и има направление $\vec{p}$.

Точката M лежи на g точно тогава, когато векторите $\overrightarrow{M_0M}$ и $\vec{p}$ са колинеарни, т.е.

$$\overrightarrow{M_0M} = s\vec{p}, \qquad s \in \mathbb{R}.$$

Следователно векторно-параметричното уравнение на правата е

$$\boxed{\vec{r} = \vec{r}_0 + s\vec{p}, \qquad s \in \mathbb{R}.}$$

★ **Дефиниция 29.20.** Ако

$$M_0(x_0, y_0, z_0), \qquad \vec{p} = (p_1, p_2, p_3) \neq \vec{0},$$

то координатните параметрични уравнения на правата g са

$$\boxed{\begin{cases} x = x_0 + sp_1, \\ y = y_0 + sp_2, \\ z = z_0 + sp_3, \end{cases} \qquad s \in \mathbb{R}.}$$

Векторът $\vec{p}$ се нарича направляващ вектор на правата.

▷ **Пример 29.21.** Правата през $M_0(2, -1, 3)$ с направляващ вектор $\vec{p} = (1, 2, -2)$ има уравнения

$$\begin{cases} x = 2 + s, \\ y = -1 + 2s, \\ z = 3 - 2s, \end{cases} \qquad s \in \mathbb{R}.$$

29.6.2 Права като пресечница на две равнини

Нека

$$\alpha_1 : A_1x + B_1y + C_1z + D_1 = 0,$$
$$\alpha_2 : A_2x + B_2y + C_2z + D_2 = 0$$

са две непресичащи се само в смисъл на съвпадане или успоредност изключени равнини, т.е. те не съвпадат и не са успоредни. Тогава

$$\alpha_1 \cap \alpha_2 = g$$

е права. Тя се задава чрез системата

$$\boxed{\begin{cases} A_1x + B_1y + C_1z + D_1 = 0, \\ A_2x + B_2y + C_2z + D_2 = 0. \end{cases}}$$

Ако, например,

$$A_1B_2 - A_2B_1 \neq 0,$$

от системата могат да се изразят x и y като линейни функции на z:

$$x = az + p, \qquad y = bz + q.$$

Тези уравнения, заедно с избора

$$z = s,$$

водят до параметрично представяне

$$\begin{cases} x = p + as, \\ y = q + bs, \\ z = s, \end{cases} \qquad s \in \mathbb{R}.$$

Следователно пресечницата минава през точката $(p, q, 0)$ и има направляващ вектор

$$(a, b, 1).$$

▷ **Пример 29.22.** Да се намери параметрично уравнение на пресечницата на равнините

$$\alpha_1 : x + y + z - 1 = 0, \qquad \alpha_2 : x - y + z - 3 = 0.$$

Изваждаме второто уравнение от първото:

$$2y + 2 = 0, \qquad y = -1.$$

От първото уравнение следва

$$x + z = 2.$$

Полагаме $z = s$. Тогава

$$x = 2 - s, \qquad y = -1, \qquad z = s.$$

Следователно търсената права има уравнения

$$\begin{cases} x = 2 - s, \\ y = -1, \\ z = s, \end{cases} \qquad s \in \mathbb{R},$$

или минава през $(2, -1, 0)$ и е успоредна на вектора $(-1, 0, 1)$.

29.7 Изпитен фокус

- Да се формулира как се определя равнина: точка и два линейно независими компланарни вектора.
- Да се запишат векторно-параметричното и координатните параметрични уравнения на равнина:

  $$\vec{r} = \vec{r}_0 + \lambda\vec{p} + \mu\vec{q}$$

  и съответната система от три координатни уравнения.

- Да се изведе общото уравнение на равнина от условието за компланарност на $\overrightarrow{M_0M}$, $\vec{p}$ и $\vec{q}$.
- Да се възпроизведе теоремата за общото уравнение:

  $$Ax + By + Cz + D = 0, \qquad (A, B, C) \neq (0, 0, 0),$$

  включително идеята за обратното твърдение чрез точка от равнината и два линейно независими вектора.
- Да се построи уравнение на равнина през три неколинеарни точки чрез детерминанта.
- Да се разпознава нормалният вектор $\vec{n} = (A, B, C)$ и да се използва условието

  $$Av_1 + Bv_2 + Cv_3 = 0$$

  за компланарност на вектор с равнина.
- Да се определят взаимните положения на две равнини чрез пропорционалност на нормалните вектори и чрез ранговете на матриците на коефициентите.
- Да се запише нормалното уравнение на равнина и да се знае, че има две нормални уравнения:

  $$\pm\frac{Ax + By + Cz + D}{\sqrt{A^2 + B^2 + C^2}} = 0.$$

- Да се прилага формулата за разстояние от точка до равнина:

  $$d(M_0, \alpha) = \frac{|Ax_0 + By_0 + Cz_0 + D|}{\sqrt{A^2 + B^2 + C^2}}.$$

  Идеята на доказателството е ортогоналната проекция върху равнината и разлагане по единичния нормален вектор.
- Да се опишат полупространствата чрез знака на

  $$E(x, y, z) = Ax + By + Cz + D$$

  и да се възпроизведе критерият

  $$E(M_1)E(M_2) < 0$$

  за точки в различни полупространства.
- Да се запишат векторно-параметричното и координатните параметрични уравнения на права:

  $$\vec{r} = \vec{r}_0 + s\vec{p}, \qquad \begin{cases} x = x_0 + sp_1, \\ y = y_0 + sp_2, \\ z = z_0 + sp_3. \end{cases}$$

- Да се намира параметрично представяне на права, дадена като пресечница на две *непаралелни* равнини — тоест две различни равнини, чиито нормални вектори не са колинеарни и които затова се пресичат по права — чрез решаване на системата на техните уравнения (§29.6.2).

Въпрос 30

Симетрични оператори в крайномерни евклидови пространства. Теорема за диагонализация.

По конспекта: какво трябва да мога

Официално заглавие: „Симетрични оператори в крайномерни евклидови пространства. *Основни свойства.* Теорема за диагонализация.“
Дефиниция: симетричен оператор — §30.2; матрицата му спрямо ортонормиран базис — §30.2.1, стр. 422.
Доказателство: всички характеристични корени на симетричен оператор са реални — §30.3.1, стр. 425.
Доказателство: всеки два собствени вектора за различни собствени стойности са ортогонални — §30.3.2, стр. 426.
Лема с доказателство: инвариантност на ортогоналното допълнение — използва се в теоремата за диагонализация — §30.4, стр. 427.
Теорема с доказателство: съществува ортонормиран базис, в който матрицата на симетричен оператор е диагонална — §30.5, стр. 428.
Предварително (не се изпитва тук): §30.1 събира понятията от линейната алгебра — евклидово пространство, скаларно произведение, ортонормиран базис, линеен оператор и матрицата му. Анотацията започва от §30.2.
Литература: [18].

30.1 Предварителни понятия

† **Допълнение 30.1 (Предварителен материал).** Анотацията към Въпрос 30 започва от определението за симетричен оператор (§30.2). Този раздел събира понятията от линейната алгебра, които се използват по-нататък — евклидово пространство, скаларно произведение, ортонормиран базис, линеен оператор и матрицата му. Той е предпоставка, а не част от изпитния въпрос; при повторение може да се прегледа бегло.

Понятията в този раздел са от курса по линейна алгебра. Изброяваме ги, защото всяко от тях се използва по-нататък в главата: без тях нито определението за симетричен оператор, нито доказателството на теоремата за диагонализация са напълно записани.

Дефиниция 30.2 (Скаларно произведение и евклидово пространство)**.** Нека E е векторно пространство над $\mathbb{R}$. Изображението

$$(\cdot,\cdot) : E \times E \to \mathbb{R}$$

се нарича *скаларно произведение*, ако за всички $x, y, z \in E$ и $\alpha \in \mathbb{R}$ са изпълнени:

- симетричност: $(x, y) = (y, x)$;
- адитивност и хомогенност по първия аргумент:
$$(x + z, y) = (x, y) + (z, y), \qquad (\alpha x, y) = \alpha(x, y);$$
- положителна определеност: $(x, x) \geq 0$, като $(x, x) = 0$ точно тогава, когато $x = 0$.

От симетричността следва същото и по втория аргумент, тоест скаларното произведение е *билинейно*. Двойката $\big(E, (\cdot,\cdot)\big)$ се нарича *евклидово пространство*; то е крайномерно, ако $\dim E < \infty$.

Стандартният пример е $\mathbb{R}^n$ със скаларно произведение

$$(x, y) = \sum_{i=1}^{n} x_i y_i = x^t y,$$

което ще наричаме *стандартно евклидово скаларно произведение*.

Дефиниция 30.3 (Норма и ортогоналност)**.** Нека E е евклидово пространство. *Норма* на вектора x е числото

$$\|x\| = \sqrt{(x,x)},$$

което е определено, защото $(x,x) \geq 0$. Вектор с $\|x\| = 1$ се нарича *единичен*; за $x \neq 0$ векторът $x/\|x\|$ е единичен и се получава чрез *нормализиране* на x.
Векторите x и y са *ортогонални*, което се записва $x \perp y$, ако

$$(x,y) = 0.$$

Две подмножества $A, B \subseteq E$ са ортогонални, $A \perp B$, ако $x \perp y$ за всички $x \in A$ и $y \in B$.

Дефиниция 30.4 (Ортонормиран базис)**.** Системата вектори $f_1, \ldots, f_k$ е *ортогонална*, ако $(f_i, f_j) = 0$ при $i \neq j$, и *ортонормирана*, ако освен това всички вектори са единични, тоест

$$(f_i, f_j) = \delta_{ij} = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

Ортонормиран базис на E е базис, който е ортонормирана система.

Ортонормираният базис е удобен именно защото свежда скаларното произведение до координатно пресмятане: ако $e_1, \ldots, e_n$ е ортонормиран базис и

$$x = \sum_{i=1}^{n} x_i e_i, \qquad y = \sum_{j=1}^{n} y_j e_j,$$

то от билинейността

$$(x,y) = \sum_{i=1}^{n}\sum_{j=1}^{n} x_i y_j (e_i, e_j) = \sum_{i=1}^{n} x_i y_i = x^t y.$$

В частност $x_i = (x, e_i)$: координатите на вектор спрямо ортонормиран базис са скаларните му произведения с базисните вектори.

Твърдение 30.5 (Съществуване на ортонормиран базис)**.** Всяко крайномерно евклидово пространство E с $\dim E = n \geq 1$ притежава ортонормиран базис.

Доказателство. Нека $v_1, \ldots, v_n$ е произволен базис на E. По метода на Грам–Шмит строим индуктивно

$$w_1 = v_1, \qquad w_k = v_k - \sum_{i=1}^{k-1} \frac{(v_k, w_i)}{(w_i, w_i)}\, w_i, \qquad k = 2, \ldots, n.$$

Допускаме индуктивно, че $w_1, \ldots, w_{k-1}$ са ненулеви, попарно ортогонални и

$$\mathcal{L}(w_1, \ldots, w_{k-1}) = \mathcal{L}(v_1, \ldots, v_{k-1}).$$

Тогава дробите са определени, защото $(w_i, w_i) > 0$ за ненулево w_i. За $j < k$ пресмятаме

$$(w_k, w_j) = (v_k, w_j) - \sum_{i=1}^{k-1} \frac{(v_k, w_i)}{(w_i, w_i)}(w_i, w_j) = (v_k, w_j) - (v_k, w_j) = 0,$$

тъй като в сумата оцелява само събираемото с $i = j$. Освен това $w_k \neq 0$: иначе v_k би лежал в $\mathcal{L}(w_1, \ldots, w_{k-1}) = \mathcal{L}(v_1, \ldots, v_{k-1})$, което противоречи на линейната независимост на базиса. От построението следва и равенството на обвивките за k.
Накрая полагаме $e_i = w_i/\|w_i\|$. Системата $e_1, \ldots, e_n$ е ортонормирана. Всяка ортонормирана система е линейно независима: ако $\sum_i \alpha_i e_i = 0$, скаларното умножение с e_j дава $\alpha_j = 0$. Значи n такива вектора образуват базис на E. □

Дефиниция 30.6 (Линейна обвивка и ортогонално допълнение)**.** С $\mathcal{L}(v_1, \ldots, v_k)$ означаваме *линейната обвивка* на векторите $v_1, \ldots, v_k$, тоест множеството от всички техни линейни комбинации; това е най-малкото подпространство, което ги съдържа. В частност за $u \neq 0$

$$\mathcal{L}(u) = \{\alpha u \mid \alpha \in \mathbb{R}\}$$

е правата, породена от u.
Ортогонално допълнение на подпространството $U \subseteq E$ е множеството

$$U^\perp = \{x \in E \mid (x, u) = 0 \text{ за всяко } u \in U\}.$$

То е подпространство на E, защото условието е линейно по x.

Твърдение 30.7 (Ортогонално разлагане)**.** Нека E е крайномерно евклидово пространство и $U \subseteq E$ е подпространство. Тогава

$$E = U \oplus U^\perp, \qquad \dim U^\perp = \dim E - \dim U.$$

Доказателство. Ако $U = \{0\}$, твърдението е очевидно, защото тогава $U^\perp = E$. Нека $\dim U = k \geq 1$ и нека $u_1, \ldots, u_k$ е ортонормиран базис на U; такъв съществува по предходното твърдение, приложено към U със същото скаларно произведение.
За произволно $x \in E$ полагаме

$$p = \sum_{i=1}^{k} (x, u_i) u_i \in U, \qquad r = x - p.$$

За всяко j имаме

$$(r, u_j) = (x, u_j) - \sum_{i=1}^{k} (x, u_i)(u_i, u_j) = (x, u_j) - (x, u_j) = 0,$$

защото базисът е ортонормиран. Понеже r е ортогонален на всички u_j, по линейност той е ортогонален на всяка тяхна линейна комбинация, тоест $r \in U^\perp$. Следователно $x = p + r$ и

$$E = U + U^\perp.$$

Сумата е директна: ако $x \in U \cap U^{\perp}$, то x е ортогонален на себе си, тоест $(x,x) = 0$, откъдето $x = 0$ поради положителната определеност. Оттук следва и равенството за размерностите. □

Дефиниция 30.8 (Собствена стойност, собствен вектор, собствено подпространство)**.** Нека $\varphi : E \to E$ е линеен оператор. Числото $\lambda \in \mathbb{R}$ е *собствена стойност* на φ, ако съществува вектор $x \neq 0$, за който

$$\varphi(x) = \lambda x.$$

Всеки такъв x се нарича *собствен вектор* за λ. Множеството

$$E_\lambda = \{x \in E \mid \varphi(x) = \lambda x\}$$

е подпространство на E и се нарича *собствено подпространство* за λ; то се състои от собствените вектори за λ и нулевия вектор.

Дефиниция 30.9 (Характеристичен полином и характеристичен корен)**.** Нека $A \in M_{n\times n}(\mathbb{R})$. Полиномът

$$\chi_A(\lambda) = \det(A - \lambda I)$$

от степен n се нарича *характеристичен полином* на A. Неговите корени в $\mathbb{C}$ се наричат *характеристични корени*.
Подобните матрици имат един и същ характеристичен полином, затова матриците на един и същ оператор спрямо различни базиси дават един и същ полином. Той се нарича характеристичен полином на оператора φ.

Връзката между двете понятия е следната. Ако A е матрицата на φ спрямо някакъв базис, то за $\lambda \in \mathbb{R}$

$$\varphi(x) = \lambda x \text{ за някое } x \neq 0 \iff (A-\lambda I)X = 0 \text{ има ненулево решение} \iff \det(A-\lambda I) = 0.$$

Следователно *собствените стойности на φ са точно реалните характеристични корени*. Затова твърдението, че всички характеристични корени на симетричен оператор са реални, е по същество твърдение за пълнотата на спектъра му.

Забележка 30.10. По основната теорема на алгебрата всеки полином от степен $n \geq 1$ с комплексни коефициенти има корен в $\mathbb{C}$. Затова всеки оператор в пространство с $\dim E \geq 1$ има поне един характеристичен корен, но той не е задължително реален; именно това е препятствието, което симетричността премахва.

Дефиниция 30.11 (Инвариантно подпространство и ограничение)**.** Подпространството $U \subseteq E$ е *инвариантно* относно линейния оператор $\varphi : E \to E$, ако

$$\varphi(U) \subseteq U.$$

Тогава *ограничението*

$$\varphi|_U : U \to U, \qquad \varphi|_U(x) = \varphi(x),$$

е коректно определен линеен оператор в U.

Дефиниция 30.12 (Ортогонална матрица)**.** Матрицата $Q \in M_{n\times n}(\mathbb{R})$ се нарича *ортогонална*, ако

$$Q^tQ = I,$$

тоест $Q^{-1} = Q^t$. Еквивалентно, стълбовете на Q образуват ортонормирана система спрямо стандартното скаларно произведение в $\mathbb{R}^n$.

30.2 Симетрични оператори

Оттук нататък E е крайномерно евклидово пространство над $\mathbb{R}$ със скаларно произведение $(\cdot,\cdot)$ и $\dim E = n$.

Дефиниция 30.13. Линейният оператор $\varphi : E \to E$ се нарича *симетричен*, ако за всеки два вектора $x, y \in E$ е изпълнено

$$(\varphi(x), y) = (x, \varphi(y)).$$

Това определение изразява съгласуваност между действието на оператора и скаларното произведение. При симетричен оператор прехвърлянето на оператора от първия към втория аргумент на скаларното произведение не променя стойността му.

Забележка 30.14. В по-общия случай на унитарно пространство над $\mathbb{C}$ аналогичното понятие е *ермитов оператор*. Тогава условието е

$$\langle\varphi(x), y\rangle = \langle x, \varphi(y)\rangle.$$

В настоящата глава разглеждаме евклидови пространства над $\mathbb{R}$, поради което използваме термина *симетричен оператор*.

30.2.1 Матрица на симетричен оператор

Нека $e_1, \ldots, e_n$ е ортонормиран базис на E и

$$A = (a_{ij}) \in M_{n\times n}(\mathbb{R})$$

е матрицата на φ спрямо този базис. По определение координатите на $\varphi(e_j)$ са елементите от j-тия стълб на A, тоест

$$\varphi(e_j) = \sum_{i=1}^{n} a_{ij} e_i.$$

Дефиниция 30.15. Матрицата $A = (a_{ij}) \in M_{n\times n}(\mathbb{R})$ се нарича *симетрична*, ако

$$A^t = A.$$

Еквивалентно,

$$a_{ij} = a_{ji} \qquad (1 \leq i, j \leq n).$$

Твърдение 30.16. Нека $\varphi : E \to E$ е линеен оператор и $e_1, \ldots, e_n$ е ортонормиран базис на E. Тогава следните условия са еквивалентни:

1. φ е симетричен оператор;
2. матрицата на φ спрямо базиса $e_1, \ldots, e_n$ е симетрична.

Доказателство. Нека първо φ е симетричен оператор и $A = (a_{ij})$ е неговата матрица спрямо дадения ортонормиран базис. За произволни $i, j \in \{1, \ldots, n\}$ имаме

$$a_{ji} = (\varphi(e_i), e_j).$$

Тъй като φ е симетричен,

$$(\varphi(e_i), e_j) = (e_i, \varphi(e_j)).$$

От ортонормираността на базиса следва

$$(e_i, \varphi(e_j)) = a_{ij}.$$

Следователно

$$a_{ji} = a_{ij}$$

за всички i, j, откъдето $A^t = A$.
Обратно, нека матрицата A е симетрична. Тогава

$$(\varphi(e_i), e_j) = a_{ji} = a_{ij} = (e_i, \varphi(e_j))$$

за всички базисни вектори e_i, e_j.
Нека сега

$$x = \sum_{i=1}^{n} x_i e_i, \qquad y = \sum_{j=1}^{n} y_j e_j.$$

Използвайки линейността на φ и билинейността на скаларното произведение, получаваме

$$\begin{aligned}(\varphi(x), y) &= \left(\varphi\left(\sum_{i=1}^{n} x_i e_i\right), \sum_{j=1}^{n} y_j e_j\right) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} x_i y_j (\varphi(e_i), e_j) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} x_i y_j (e_i, \varphi(e_j)) \\ &= \left(\sum_{i=1}^{n} x_i e_i, \varphi\left(\sum_{j=1}^{n} y_j e_j\right)\right) \\ &= (x, \varphi(y)).\end{aligned}$$

Следователно φ е симетричен оператор. □

Следствие 30.17. Линеен оператор в крайномерно евклидово пространство е симетричен точно тогава, когато матрицата му спрямо произволен ортонормиран базис е симетрична.

Доказателство. В ортонормиран базис скаларното произведение има вид $x^t y$. Ако A е матрицата на оператора, условието за симетричност е $(Ax)^t y = x^t A y$ за всички x, y, тоест $x^t(A^t - A)y = 0$. Избирането на координатни вектори за x, y показва, че това е еквивалентно на $A^t = A$. Аргументът важи във всеки ортонормиран базис. □

† **Допълнение 30.18 (Бележка).** За матрици с комплексни елементи условието

$$\overline{A}^{\,t} = A$$

описва ермитова, а не непременно симетрична матрица. В реалния случай $\overline{A} = A$, така че това условие се свежда до $A^t = A$.

30.3 Собствени стойности и собствени вектори

Нека $\varphi : E \to E$ е симетричен оператор. Основните му спектрални свойства са, че характеристичните му корени са реални и че собствените подпространства за различни собствени стойности са ортогонални.

30.3.1 Реалност на характеристичните корени

Теорема 30.19. Всички характеристични корени на симетричен оператор в крайномерно евклидово пространство са реални числа.

Доказателство. Нека A е матрицата на φ спрямо ортонормиран базис на E. Съгласно предходното твърдение A е реална симетрична матрица.
Нека $\lambda \in \mathbb{C}$ е характеристичен корен на A. Тогава съществува ненулев комплексен стълб-вектор

$$z = \begin{pmatrix} z_1 \\ \vdots \\ z_n \end{pmatrix} \in \mathbb{C}^n$$

такъв, че

$$Az = \lambda z.$$

Умножаваме отляво с ред-вектора

$$\overline{z}^t = (\overline{z_1}, \ldots, \overline{z_n}).$$

Получаваме

$$\overline{z}^t A z = \lambda \overline{z}^t z = \lambda \sum_{i=1}^{n} |z_i|^2.$$

Нека

$$q = \overline{z}^t A z.$$

Тъй като A е реална симетрична матрица и q е скалар,

$$\overline{q} = z^t A \overline{z} = (z^t A \overline{z})^t = \overline{z}^t A^t z = \overline{z}^t A z = q.$$

Следователно q е реално число. От друга страна, комплексното спрягане на равенството $Az = \lambda z$ дава

$$A\overline{z} = \overline{\lambda}\,\overline{z}.$$

Следователно

$$q = \overline{q} = z^t A \overline{z} = \overline{\lambda} \sum_{i=1}^{n} |z_i|^2.$$

Значи

$$\lambda \sum_{i=1}^{n} |z_i|^2 = \overline{\lambda} \sum_{i=1}^{n} |z_i|^2.$$

Понеже $z \neq 0$, то

$$\sum_{i=1}^{n} |z_i|^2 > 0.$$

Следователно

$$\lambda = \overline{\lambda},$$

което означава, че $\lambda \in \mathbb{R}$. □

Забележка 30.20. Доказателството разглежда характеристичните корени като комплексни числа. Това е необходимо, защото характеристичният полином на реална матрица може да има комплексни корени. За симетрична матрица доказаното свойство показва, че всички тези корени всъщност са реални.

Следствие 30.21. Всеки симетричен оператор $\varphi : E \to E$ има собствена стойност, ако $\dim E \geq 1$.

Доказателство. Характеристичният полином на матрицата на φ има комплексен корен. По теоремата всички негови корени са реални, следователно този корен е реална собствена стойност на φ. □

30.3.2 Ортогоналност на собствени вектори

Твърдение 30.22. Нека $\varphi : E \to E$ е симетричен оператор. Ако $u, v \in E \setminus \{0\}$ са собствени вектори, съответстващи съответно на различните собствени стойности λ и μ, то

$$u \perp v.$$

Доказателство. Имаме

$$\varphi(u) = \lambda u, \qquad \varphi(v) = \mu v,$$

където $\lambda \neq \mu$. От симетричността на φ следва

$$(\varphi(u), v) = (u, \varphi(v)).$$

След заместване получаваме

$$(\lambda u, v) = (u, \mu v),$$

тоест

$$\lambda(u, v) = \mu(u, v).$$

Следователно

$$(\lambda - \mu)(u, v) = 0.$$

Понеже $\lambda \neq \mu$, имаме

$$(u, v) = 0.$$

Следователно $u \perp v$. □

Следствие 30.23. Собствените подпространства на симетричен оператор, съответстващи на различни собствени стойности, са ортогонални.

Доказателство. Нека E_λ и E_μ са собствените подпространства за λ и μ, където $\lambda \neq \mu$. Всеки ненулев вектор от E_λ е собствен вектор за λ, а всеки ненулев вектор от E_μ е

собствен вектор за μ. По предходното твърдение всеки такъв чифт вектори е ортогонален. Следователно

$$E_\lambda \perp E_\mu.$$

□

Забележка 30.24. Условието за различност на собствените стойности е съществено. Два собствени вектора, отговарящи на една и съща собствена стойност, не са задължително ортогонални. В собственото подпространство обаче може да се избере ортонормиран базис.

30.4 Инвариантност на ортогоналното допълнение

За доказателството на теоремата за диагонализация е необходимо свойството, че ортогоналното допълнение на собствен вектор е инвариантно относно симетричния оператор.

Лема 30.25. Нека $\varphi : E \to E$ е симетричен оператор и $u \in E \setminus \{0\}$ е собствен вектор:

$$\varphi(u) = \lambda u.$$

Тогава ортогоналното допълнение

$$\mathcal{L}(u)^\perp$$

е инвариантно относно φ, тоест

$$\varphi\big(\mathcal{L}(u)^\perp\big) \subseteq \mathcal{L}(u)^\perp.$$

Доказателство. Нека $x \in \mathcal{L}(u)^\perp$. Тогава

$$(x, u) = 0.$$

Ще докажем, че $\varphi(x) \in \mathcal{L}(u)^\perp$. От симетричността на φ следва

$$(\varphi(x), u) = (x, \varphi(u)).$$

Тъй като $\varphi(u) = \lambda u$, получаваме

$$(\varphi(x), u) = (x, \lambda u) = \lambda(x, u) = 0.$$

Следователно

$$\varphi(x) \perp u,$$

тоест

$$\varphi(x) \in \mathcal{L}(u)^\perp.$$

□

Лема 30.26. При условията на предходната лема ограничението

$$\varphi\big|_{\mathcal{L}(u)^\perp} : \mathcal{L}(u)^\perp \to \mathcal{L}(u)^\perp$$

е симетричен оператор.

Доказателство. Нека $x, y \in \mathcal{L}(u)^\perp$. Понеже φ е симетричен в E, имаме

$$(\varphi(x), y) = (x, \varphi(y)).$$

Съгласно предходната лема $\varphi(x)$ и $\varphi(y)$ принадлежат на $\mathcal{L}(u)^\perp$, следователно това равенство е именно условието ограничението на φ върху това подпространство да е симетрично. □

30.5 Теорема за диагонализация

★ **Теорема 30.27 (Теорема за диагонализация на симетричен оператор).** Нека E е крайномерно евклидово пространство и $\varphi : E \to E$ е симетричен оператор. Тогава съществува ортонормиран базис

$$e_1, \ldots, e_n$$

на E, съставен от собствени вектори на φ.
Еквивалентно, съществува ортонормиран базис на E, спрямо който матрицата на φ е диагонална:

$$[\varphi]_{(e_1,\ldots,e_n)} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix},$$

където $\lambda_1, \ldots, \lambda_n \in \mathbb{R}$ са собствени стойности на φ, като някои от тях могат да съвпадат.

Доказателство. Ще докажем твърдението с индукция по

$$n = \dim E.$$

При $n = 1$ твърдението е очевидно. Ако e_1 е единичен вектор, то $E = \mathcal{L}(e_1)$ и поради линейността на φ съществува $\lambda_1 \in \mathbb{R}$, за което

$$\varphi(e_1) = \lambda_1 e_1.$$

Следователно e_1 е ортонормиран базис от собствени вектори.
Нека твърдението е вярно за всички евклидови пространства с размерност $n-1$, и нека сега $\dim E = n$.
По доказаното по-горе симетричният оператор φ има реална собствена стойност λ_1.
Нека

$$u_1 \in E \setminus \{0\}$$

е съответен собствен вектор:

$$\varphi(u_1) = \lambda_1 u_1.$$

Нормализираме u_1 и полагаме

$$e_1 = \frac{u_1}{\|u_1\|}.$$

Тогава $\|e_1\| = 1$ и

$$\varphi(e_1) = \lambda_1 e_1.$$

Нека

$$U = \mathcal{L}(e_1)^\perp.$$

По твърдението за ортогоналното разлагане $E = \mathcal{L}(e_1) \oplus U$, а тъй като $\dim \mathcal{L}(e_1) = 1$, това е подпространство с размерност $n - 1$. От лемата следва, че U е инвариантно относно φ. Следователно ограничението

$$\varphi|_U : U \to U$$

е коректно определен линеен оператор. Освен това то е симетрично.
По индукционното предположение за оператора $\varphi|_U$ съществува ортонормиран базис

$$e_2, \ldots, e_n$$

на U, съставен от собствени вектори на ограничението. Следователно съществуват реални числа $\lambda_2, \ldots, \lambda_n$, за които

$$\varphi(e_i) = \lambda_i e_i, \qquad i = 2, \ldots, n.$$

Тъй като разлагането

$$E = \mathcal{L}(e_1) \oplus U$$

е ортогонално — всеки вектор от U е ортогонален на e_1 по определението за U — системата

$$e_1, e_2, \ldots, e_n$$

е ортонормиран базис на E. Всеки негов вектор е собствен вектор на φ. Следователно матрицата на φ спрямо този базис е

$$\begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix}.$$

Така индукционната стъпка е доказана. □

Следствие 30.28. Всеки симетричен оператор в крайномерно евклидово пространство е диагонализируем спрямо ортонормиран базис.

Доказателство. По теоремата за диагонализация съществува ортонормиран базис от собствени вектори $e_1, \ldots, e_n$. Ако $\varphi(e_i) = \lambda_i e_i$, i-тата колона на матрицата в този базис има единствена възможна ненулева координата λ_i на ред i. Следователно матрицата е $\mathrm{diag}(\lambda_1, \ldots, \lambda_n)$. □

Забележка 30.29. Теоремата не твърди само съществуване на базис, в който матрицата е диагонална. Същественото допълнително свойство е, че базисът може да бъде избран ортонормиран. Именно това е специфичната особеност на симетричните оператори.

30.5.1 Връзка със симетричните матрици

Нека $A \in M_{n\times n}(\mathbb{R})$ е симетрична матрица. Разглеждаме линейния оператор

$$\varphi : \mathbb{R}^n \to \mathbb{R}^n, \qquad \varphi(x) = Ax,$$

спрямо стандартното евклидово скаларно произведение. Матрицата на φ спрямо каноничния ортонормиран базис е A, следователно φ е симетричен оператор.

От теоремата за диагонализация следва, че съществува ортонормиран базис от собствени вектори на A. Ако Q е матрицата, чиито стълбове са тези базисни вектори, то Q е ортогонална матрица и в този базис матрицата на оператора е диагонална. Следователно симетричната матрица може да бъде приведена до диагонален вид чрез преход към ортонормиран базис.

▷ **Пример 30.30.** Нека

$$A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}.$$

Матрицата е симетрична, защото

$$A^t = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} = A.$$

Характеристичният полином е

$$\det(A - \lambda I) = \begin{vmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{vmatrix} = (2-\lambda)^2 - 1.$$

Следователно характеристичните корени са

$$\lambda_1 = 3, \qquad \lambda_2 = 1.$$

За $\lambda_1 = 3$ получаваме собствен вектор

$$u_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix},$$

а за $\lambda_2 = 1$ — собствен вектор

$$u_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}.$$

Техният скаларен продукт е

$$(u_1, u_2) = 1 \cdot 1 + 1 \cdot (-1) = 0,$$

което илюстрира ортогоналността на собствени вектори за различни собствени стойности.
След нормализиране получаваме ортонормирани собствени вектори

$$e_1 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \qquad e_2 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}.$$

Спрямо ортонормирания базис e_1, e_2 матрицата на оператора е

$$\begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix}.$$

30.6 Обобщение на свойствата

Нека $\varphi : E \to E$ е симетричен оператор в крайномерно евклидово пространство. Тогава са изпълнени следните основни свойства:

1. За всеки $x, y \in E$ е изпълнено
$$(\varphi(x), y) = (x, \varphi(y)).$$
2. Матрицата на φ спрямо всеки ортонормиран базис е реална симетрична матрица.
3. Всички характеристични корени, съответно всички собствени стойности, на φ са реални.
4. Ако u и v са собствени вектори за различни собствени стойности, то
$$(u, v) = 0.$$
5. Ако u е собствен вектор, то подпространството
$$\mathcal{L}(u)^{\perp}$$
е инвариантно относно φ.
6. Съществува ортонормиран базис на E, съставен от собствени вектори на φ.
7. Спрямо този ортонормиран базис матрицата на φ е диагонална, а елементите по главния диагонал са реалните собствени стойности на оператора.

30.7 Изпитен фокус

- Да се даде определението за симетричен оператор:
$$(\varphi(x), y) = (x, \varphi(y)) \qquad \text{за всички } x, y \in E.$$

- Да се знае, че матрицата на симетричен оператор спрямо ортонормиран базис е симетрична, и да се възпроизведе доказателството чрез равенството

$$a_{ji} = (\varphi(e_i), e_j) = (e_i, \varphi(e_j)) = a_{ij}.$$

- Да се знае и обратното: симетрична матрица спрямо ортонормиран базис определя симетричен оператор.

- Да се формулира и докаже, че всички характеристични корени на симетричен оператор са реални, като се използва равенството

$$\lambda \sum_{i=1}^{n} |z_i|^2 = \overline{\lambda} \sum_{i=1}^{n} |z_i|^2.$$

- Да се формулира и докаже ортогоналността на собствени вектори за различни собствени стойности:

$$(\lambda - \mu)(u, v) = 0.$$

- Да се знае, че ако u е собствен вектор, то $\mathcal{L}(u)^\perp$ е инвариантно относно симетричния оператор, защото

$$(\varphi(x), u) = (x, \varphi(u)) = \lambda(x, u) = 0.$$

- Да се формулира теоремата за диагонализация: всеки симетричен оператор в крайномерно евклидово пространство има ортонормиран базис от собствени вектори.

- Да се възпроизведе доказателствената идея за теоремата: индукция по $\dim E$, избор и нормализиране на собствен вектор, преминаване към неговото ортогонално допълнение и прилагане на индукционното предположение върху ограничението на оператора.

Въпрос 31

Симетрична и алтернативна група. Теорема на Кейли. Теорема за хомоморфизмите.

По конспекта: какво трябва да мога

Симетрична група: представяне на елементите като произведение на независими цикли — §31.1, стр. 433.
Спрягане: на елементите на S_n — §31.2, стр. 435.
Транспозиции: представяне като произведение на транспозиции; четност; алтернативна група — §31.3, стр. 436.
Хомоморфизми: хомоморфизъм при групи, ядро и образ — §31.4, стр. 438.
Теорема с доказателство: за хомоморфизмите при групи — §31.5, стр. 440.
Теорема с доказателство: на Кейли — всяка крайна група е изоморфна на подгрупа на симетрична група — §31.6, стр. 441.
Литература: [17].

31.1 Симетрична група и цикличен запис

Дефиниция 31.1. Нека M е множество. Множеството

$$\mathrm{Sym}(M) = \{f : M \to M \mid f \text{ е биекция}\}$$

е група относно композицията на изображенията. Тя се нарича *симетрична група на множеството* M. Неутралният ⊠ елемент е тъждественото изображение

$$\mathrm{Id}_M : M \to M, \qquad \mathrm{Id}_M(x) = x.$$

Ако $|M| = n$, то след избиране на номерация $M = \{1, \ldots, n\}$ симетричната група се бележи със S_n и се нарича *симетрична група от степен* n. Нейните елементи се наричат

пермутации. Всяка пермутация е биекция на $\{1, \ldots, n\}$, следователно се определя еднозначно от редицата

$$\sigma(1), \sigma(2), \ldots, \sigma(n),$$

която е пренареждане на числата $1, \ldots, n$. Поради това

$$|S_n| = n!.$$

Дефиниция 31.2. За $\sigma \in S_n$ множеството

$$\mathrm{Supp}(\sigma) = \{i \in \{1, \ldots, n\} \mid \sigma(i) \neq i\}$$

се нарича *носител* на σ.

Дефиниция 31.3. Пермутацията $\xi \in S_n$ е *цикъл с дължина* k, ако има k различни числа

$$i_1, \ldots, i_k$$

такива, че

$$\xi(i_s) = i_{s+1} \quad (1 \leq s < k), \qquad \xi(i_k) = i_1,$$

а всички останали числа се оставят неподвижни. Тя се записва като

$$\xi = (i_1, i_2, \ldots, i_k).$$

Един и същ цикъл има много записи, получени чрез циклично преместване:

$$(i_1, i_2, \ldots, i_k) = (i_2, \ldots, i_k, i_1).$$

Редът на елементите в цикличния запис обаче е съществен: обикновено

$$(i_1, i_2, \ldots, i_k) \neq (i_k, \ldots, i_2, i_1).$$

Дефиниция 31.4. Два цикъла се наричат *независими*, ако носителите им не се пресичат.

Независимите цикли комутират, тъй като всеки от тях оставя неподвижни всички елементи от носителя на другия. Следователно редът на множителите в произведение на независими цикли няма значение.

Теорема 31.5. Всяка нетъждествена пермутация $\sigma \in S_n \setminus \{\mathrm{Id}\}$ се представя като произведение на независими цикли. Представянето е единствено с точност до реда на множителите и до цикличното преместване в записа на всеки отделен цикъл.

Доказателство. Избираме $i_1 \in \mathrm{Supp}(\sigma)$ и разглеждаме последователността

$$i_1,\ \sigma(i_1),\ \sigma^2(i_1), \ldots.$$

Тъй като множеството $\{1, \ldots, n\}$ е крайно, някой член трябва да се повтори. Първото повторение е именно i_1: ако $\sigma(i_r) = i_s$ за $1 \leq s < r$, то от инективността на σ следва

$i_{r-1} = i_{s-1}$, което противоречи на избора на първо повторение, освен когато $s = 1$. Получава се цикъл

$$(i_1, \sigma(i_1), \ldots, \sigma^{k-1}(i_1)).$$

Множеството от елементи на този цикъл се запазва от σ, а върху допълнението му σ отново е пермутация. Повтаряйки процедурата върху останалите неподвижни още елементи от носителя, получаваме разлагане на σ в независими цикли.
За единствеността нека имаме две разлагания в независими цикли. Ако i принадлежи на цикъл от първото разлагане, то в другото разлагане има точно един цикъл, който мести i. И в двете разлагания последователните образи

$$i,\ \sigma(i),\ \sigma^2(i), \ldots$$

са едни и същи; следователно двата цикъла съвпадат. След премахването им разсъждението се прилага за останалите цикли. □

▷ **Пример 31.6.** Нека $\sigma \in S_8$ е зададена чрез

$$\sigma(1) = 4, \quad \sigma(4) = 7, \quad \sigma(7) = 1,$$

$$\sigma(2) = 6, \quad \sigma(6) = 2, \quad \sigma(3) = 3, \quad \sigma(5) = 5, \quad \sigma(8) = 8.$$

Тогава

$$\sigma = (1, 4, 7)(2, 6).$$

Фиксираните точки 3, 5 и 8 обикновено не се записват.

31.2 Спрягане в S_n

Дефиниция 31.7. Елементите a и b на група G се наричат *спрегнати*, ако съществува $c \in G$, за което

$$b = cac^{-1}.$$

В симетричната група спрягането има особено ясен смисъл: то просто преименува елементите, участващи в цикъла.

Лема 31.8. За $\rho \in S_n$ и цикъл $(i_1, \ldots, i_k) \in S_n$ е изпълнено

$$\rho(i_1, \ldots, i_k)\rho^{-1} = \big(\rho(i_1), \ldots, \rho(i_k)\big).$$

Доказателство. Нека $1 \le s \le k$, като индексите в цикъла се разглеждат по модул k. Тогава

$$\big[\rho(i_1, \ldots, i_k)\rho^{-1}\big](\rho(i_s)) = \rho(i_1, \ldots, i_k)(i_s) = \rho(i_{s+1}).$$

Това е точно действието на цикъла

$$\big(\rho(i_1), \ldots, \rho(i_k)\big).$$

Ако j не участва в първоначалния цикъл, то $\rho(j)$ не участва в получения цикъл и и двете страни оставят $\rho(j)$ неподвижно. □

Ако

$$\sigma = \alpha_1 \alpha_2 \cdots \alpha_r$$

е разлагане в независими цикли, тогава

$$\rho\sigma\rho^{-1} = (\rho\alpha_1\rho^{-1})(\rho\alpha_2\rho^{-1}) \cdots (\rho\alpha_r\rho^{-1}).$$

Следователно при спрягане се запазват дължините и броят на независимите цикли.

Дефиниция 31.9. Две пермутации от S_n имат *еднакъв цикличен строеж*, ако в разлаганията си в независими цикли имат еднакъв брой цикли от всяка дължина.

Твърдение 31.10. Две пермутации $\sigma, \tau \in S_n$ са спрегнати в S_n тогава и само тогава, когато имат еднакъв цикличен строеж.

Доказателство. Ако $\tau = \rho\sigma\rho^{-1}$, лемата показва, че всеки цикъл на σ се преобразува в цикъл със същата дължина. Значи цикличният строеж се запазва.
Обратно, нека σ и τ имат еднакъв цикличен строеж. Сдвояваме циклите им с еднаква дължина и дефинираме ρ така, че да изпраща последователните елементи на всеки цикъл на σ в последователните елементи на съответния цикъл на τ. Върху фиксираните точки ρ се избира произволно като биекция към фиксираните точки на τ. Тогава лемата дава

$$\tau = \rho\sigma\rho^{-1}.$$

□

31.3 Транспозиции, четност и алтернативна група

Дефиниция 31.11. Цикъл с дължина 2 се нарича *транспозиция*. Той има вида

$$(i, j)$$

и разменя i и j, като оставя всички останали елементи неподвижни.

За всяка транспозиция е изпълнено

$$(i, j)^{-1} = (i, j), \qquad (i, j)^2 = \mathrm{Id}\,.$$

Твърдение 31.12. Всеки цикъл с дължина $k \geq 2$ се представя като произведение на $k - 1$ транспозиции:

$$(i_1, i_2, \ldots, i_k) = (i_1, i_k)(i_1, i_{k-1}) \cdots (i_1, i_3)(i_1, i_2).$$

Доказателство. Прилагаме множителите отдясно наляво. Елементът i_1 се изпраща в i_2 от първата транспозиция и следващите го фиксират. За $2 \leq s < k$ елементът i_s остава неподвижен до (i_1, i_s), която го изпраща в i_1; следващата (i_1, i_{s+1}) го изпра-

ща в i_{s+1}, където остава. Елементът i_k се изпраща в i_1 от последната транспозиция. Всички останали елементи са фиксирани. Това е точно действието на цикъла. □

Следствие 31.13. Всяка пермутация от S_n се представя като произведение на транспозиции.

Доказателство. Разлагаме пермутацията на независими цикли по доказаната теорема. Заменяме всеки цикъл с дължина поне 2 с произведението от транспозиции от предходното твърдение. Неподвижните точки не изискват множител; тъждествената пермутация се представя с празното произведение. □

Наистина, първо разлагаме пермутацията на независими цикли, след което всеки цикъл заменяме с произведението от транспозиции, дадено по-горе.

Теорема 31.14. Ако

$$\tau_1\tau_2\cdots\tau_k = \mathrm{Id},$$

където $\tau_1,\ldots,\tau_k$ са транспозиции, то k е четно число. Следователно ако една и съща пермутация е представена като произведение съответно на k и на m транспозиции, то

$$k \equiv m \pmod 2.$$

Доказателство. На пермутация σ съпоставяме пермутационната матрица P_σ, която изпраща e_i в $e_{\sigma(i)}$. Тогава $P_{\sigma\rho} = P_\sigma P_\rho$. Матрицата на транспозиция се получава от единичната с размяна на две колони, затова детерминантата ѝ е -1. От произведението, равно на тъждествената пермутация, получаваме $1 = (-1)^k$, следователно k е четно. Ако два записа на една пермутация имат дължини k, m, умножаваме първия по обратния на втория. Получаваме запис на тъждествената с $k+m$ транспозиции, така че $k+m$ е четно, тоест $k \equiv m \pmod 2$. □

Теоремата позволява да се въведе четността на пермутация независимо от избраното разлагане на транспозиции.

Дефиниция 31.15. Пермутацията $\sigma \in S_n$ се нарича *четна*, ако в произволно нейно представяне като произведение на транспозиции броят на транспозициите е четен. Тя се нарича *нечетна*, ако този брой е нечетен.

Тъждествената пермутация е четна. Произведението на две четни пермутации е четно; обратната на четна пермутация също е четна. Аналогично, произведението на четна и нечетна пермутация е нечетно, а произведението на две нечетни пермутации е четно.

Забележка 31.16. Нека $\sigma \in S_n$ и разгледаме редицата

$$\sigma(1),\ldots,\sigma(n).$$

Двойката $\sigma(p), \sigma(q)$ образува *инверсия*, ако

$$1 \le p < q \le n \quad \text{и} \quad \sigma(p) > \sigma(q).$$

Четността на пермутацията съвпада с четността на броя на нейните инверсии.

Дефиниция 31.17. За $n \geq 2$ множеството

$$A_n = \{\sigma \in S_n \mid \sigma \text{ е четна}\}$$

се нарича *алтернативна група от степен* n.

Теорема 31.18. За всяко $n \geq 2$ множеството A_n е нормална подгрупа на S_n с индекс 2. В частност

$$A_n \triangleleft S_n, \qquad [S_n : A_n] = 2, \qquad |A_n| = \frac{n!}{2}.$$

Доказателство. Критерият за подгрупа показва, че $A_n \leq S_n$: ако $\sigma, \tau \in A_n$, то $\sigma\tau^{-1}$ е произведение на четен брой транспозиции и следователно принадлежи на A_n.
Нека (i, j) е фиксирана транспозиция. Произведението $(i, j)\sigma$ е нечетно за всяка $\sigma \in A_n$, така че

$$(i, j)A_n \subseteq S_n \setminus A_n.$$

Обратно, ако τ е нечетна, то $(i, j)\tau$ е четна. Следователно

$$\tau = (i, j)\big((i, j)\tau\big) \in (i, j)A_n.$$

Така

$$S_n \setminus A_n = (i, j)A_n.$$

Получаваме разлагането на S_n в два леви съседни класа:

$$S_n = A_n \dot\cup (i, j)A_n.$$

Следователно индексът е 2. Подгрупа с индекс 2 е нормална, а от теоремата на Лагранж следва

$$|A_n| = \frac{|S_n|}{[S_n : A_n]} = \frac{n!}{2}.$$

□

31.4 Хомоморфизми, ядро и образ

Дефиниция 31.19. Нека $(G_1, \cdot)$ и $(G_2, *)$ са групи. Отображението

$$\varphi \colon G_1 \to G_2$$

се нарича *хомоморфизъм на групи*, ако за всички $a, b \in G_1$ е изпълнено

$$\varphi(a \cdot b) = \varphi(a) * \varphi(b).$$

Ако груповите операции са означени еднакво, обичайно се пише просто

$$\varphi(ab) = \varphi(a)\varphi(b).$$

Твърдение 31.20. Нека $\varphi\colon G_1 \to G_2$ е хомоморфизъм, а e_1 и e_2 са неутралните елементи на G_1 и G_2. Тогава:

$$\varphi(e_1) = e_2;$$
$$\varphi(a^{-1}) = \varphi(a)^{-1} \qquad \text{за всяко } a \in G_1.$$

Доказателство. От $e_1e_1 = e_1$ получаваме

$$\varphi(e_1) = \varphi(e_1)\varphi(e_1).$$

Умножаваме отляво с $\varphi(e_1)^{-1}$ и получаваме $\varphi(e_1) = e_2$.
Освен това

$$e_2 = \varphi(e_1) = \varphi(aa^{-1}) = \varphi(a)\varphi(a^{-1}),$$

поради което $\varphi(a^{-1})$ е обратен на $\varphi(a)$. □

Дефиниция 31.21. За хомоморфизъм $\varphi\colon G_1 \to G_2$ множествата

$$\ker\varphi = \{a \in G_1 \mid \varphi(a) = e_2\}$$

и

$$\operatorname{Im}\varphi = \{\varphi(a) \mid a \in G_1\}$$

се наричат съответно *ядро* и *образ* на φ.

Твърдение 31.22. Ако $\varphi\colon G_1 \to G_2$ е хомоморфизъм, то

$$\ker\varphi \leq G_1, \qquad \operatorname{Im}\varphi \leq G_2.$$

Доказателство. За $a, b \in \ker\varphi$ имаме

$$\varphi(ab^{-1}) = \varphi(a)\varphi(b)^{-1} = e_2e_2^{-1} = e_2.$$

Следователно $ab^{-1} \in \ker\varphi$ и критерият за подгрупа доказва първото твърдение.
Ако $x = \varphi(a)$ и $y = \varphi(b)$ принадлежат на $\operatorname{Im}\varphi$, то

$$xy^{-1} = \varphi(a)\varphi(b)^{-1} = \varphi(ab^{-1}) \in \operatorname{Im}\varphi.$$

Следователно и образът е подгрупа. □

За подгрупа $H \leq G$ левият съседен клас с представител $a \in G$ е

$$aH = \{ah \mid h \in H\}.$$

Ако H е нормална подгрупа, пишем $H \triangleleft G$ и левите и десните съседни класове съвпадат. Тогава множеството

$$G/H = \{aH \mid a \in G\}$$

е факторгрупа относно операцията

$$(aH)(bH) = abH.$$

Твърдение 31.23. Нека $\varphi\colon G_1 \to G_2$ е хомоморфизъм и $H = \ker\varphi$. Тогава:

$$\varphi(a) = \varphi(b) \iff aH = bH$$

за всички $a, b \in G_1$.

Доказателство. Имаме

$$\varphi(a) = \varphi(b) \iff \varphi(a^{-1}b) = e_2 \iff a^{-1}b \in H.$$

Последното условие е еквивалентно на $b \in aH$, а то от своя страна е еквивалентно на $aH = bH$. □

31.5 Теорема за хомоморфизмите

★ **Теорема 31.24 (Теорема за хомоморфизмите на групи).** Нека

$$\varphi\colon G_1 \to G_2$$

е хомоморфизъм. Тогава

$$\ker\varphi \triangleleft G_1$$

и φ индуцира изоморфизъм

$$\overline{\varphi}\colon G_1/\ker\varphi \longrightarrow \operatorname{Im}\varphi, \qquad \overline{\varphi}(a\ker\varphi) = \varphi(a).$$

За естествения хомоморфизъм

$$\pi\colon G_1 \to G_1/\ker\varphi, \qquad \pi(a) = a\ker\varphi,$$

е изпълнено

$$\overline{\varphi}\circ\pi = \varphi.$$

Следователно

$$G_1/\ker\varphi \cong \operatorname{Im}\varphi.$$

Доказателство. За $a \in G_1$ и $h \in \ker\varphi$ имаме

$$\varphi(aha^{-1}) = \varphi(a)\varphi(h)\varphi(a)^{-1} = \varphi(a)e_2\varphi(a)^{-1} = e_2.$$

Следователно $aha^{-1} \in \ker\varphi$, тоест $\ker\varphi \triangleleft G_1$.
Дефинираме

$$\overline{\varphi}(a\ker\varphi) = \varphi(a).$$

Това е коректно дефинирано, защото от

$$a\ker\varphi = b\ker\varphi$$

следва $\varphi(a) = \varphi(b)$.

За $a, b \in G_1$:

$$\overline{\varphi}\big((a \ker \varphi)(b \ker \varphi)\big) = \overline{\varphi}(ab \ker \varphi) = \varphi(ab) = \varphi(a)\varphi(b),$$

тоест $\overline{\varphi}$ е хомоморфизъм.
Той е сюрективен по дефиницията на образа: всеки елемент на $\operatorname{Im}\varphi$ има вид $\varphi(a)$ и е образ на $a \ker \varphi$. Ако

$$\overline{\varphi}(a \ker \varphi) = \overline{\varphi}(b \ker \varphi),$$

то $\varphi(a) = \varphi(b)$, откъдето $a \ker \varphi = b \ker \varphi$. Значи $\overline{\varphi}$ е и инективен.
Накрая:

$$(\overline{\varphi} \circ \pi)(a) = \overline{\varphi}(a \ker \varphi) = \varphi(a).$$

□

▷ **Пример 31.25.** Нека

$$\varphi \colon \mathbb{Z} \to \mathbb{Z}_m, \qquad \varphi(k) = \overline{k},$$

където $\mathbb{Z}$ е адитивната група на целите числа, а $\mathbb{Z}_m$ е адитивната група по модул m. Тогава

$$\ker \varphi = m\mathbb{Z}, \qquad \operatorname{Im}\varphi = \mathbb{Z}_m.$$

Теоремата за хомоморфизмите дава

$$\mathbb{Z}/m\mathbb{Z} \cong \mathbb{Z}_m.$$

31.6 Теорема на Кейли

★ **Теорема 31.26 (Теорема на Кейли).** Всяка крайна група G от ред n е изоморфна на подгрупа на симетричната група S_n.

Доказателство. Разглеждаме действието на G върху самата себе си чрез леви транслации. За всяко $a \in G$ дефинираме

$$\lambda_a \colon G \to G, \qquad \lambda_a(x) = ax.$$

Отображението λ_a е биекция, защото

$$\lambda_a^{-1} = \lambda_{a^{-1}}.$$

Следователно $\lambda_a \in \operatorname{Sym}(G)$.
Нека

$$\Phi \colon G \to \operatorname{Sym}(G), \qquad \Phi(a) = \lambda_a.$$

За $a, b, x \in G$ е изпълнено

$$(\Phi(a) \circ \Phi(b))(x) = \Phi(a)(bx) = a(bx) = (ab)x = \Phi(ab)(x).$$

Следователно Φ е хомоморфизъм.
Неговото ядро е тривиално. Действително, ако $\Phi(a) = \mathrm{Id}_G$, то

$$ab = b \qquad \text{за всяко } b \in G.$$

За $b = e$ получаваме $a = e$. Значи

$$\ker \Phi = \{e\}.$$

По теоремата за хомоморфизмите

$$G/\{e\} \cong \operatorname{Im} \Phi.$$

Но $G/\{e\} \cong G$, а $\operatorname{Im} \Phi$ е подгрупа на $\mathrm{Sym}(G)$. Тъй като $|G| = n$, имаме

$$\mathrm{Sym}(G) \cong S_n.$$

Следователно G е изоморфна на подгрупа на S_n. □

† **Допълнение 31.27 (Бележка).** По-общото твърдение, доказано чрез левите транслации, е че всяка група G е изоморфна на подгрупа на $\mathrm{Sym}(G)$. Формулировката с S_n е специализацията за крайна група с n елемента.

31.7 Изпитен фокус

- Да се дадат определенията за $\mathrm{Sym}(M)$, S_n, носител на пермутация, цикъл, независими цикли, транспозиция, четна и нечетна пермутация, A_n, хомоморфизъм, ядро, образ, нормална подгрупа и факторгрупа.
- Да се обясни алгоритъмът за разлагане на пермутация в независими цикли: започва се от елемент от носителя, проследяват се последователните му образи до завръщане в началния елемент и процедурата се повтаря върху останалите елементи.
- Да се възпроизведе формулата

  $$(i_1, \ldots, i_k) = (i_1, i_k)(i_1, i_{k-1}) \cdots (i_1, i_2)$$

  и изводът, че всяка пермутация е произведение на транспозиции.
- Да се знае, че броят на транспозициите в две разлагания на една и съща пермутация има една и съща четност, и да се използва това за определяне на четни и нечетни пермутации.
- Да се формулира и докаже, че

  $$A_n \triangleleft S_n, \qquad [S_n : A_n] = 2, \qquad |A_n| = \frac{n!}{2}.$$

- Да се използва формулата за спрягане на цикъл:

  $$\rho(i_1, \ldots, i_k)\rho^{-1} = (\rho(i_1), \ldots, \rho(i_k)).$$

  Да се обясни критерият: две пермутации в S_n са спрегнати точно когато имат еднакъв цикличен строеж.

- Да се докажат основните свойства на хомоморфизма:

$$\varphi(e_1) = e_2, \qquad \varphi(a^{-1}) = \varphi(a)^{-1}, \qquad \ker\varphi \leq G_1, \qquad \operatorname{Im}\varphi \leq G_2.$$

- Да се формулира теоремата за хомоморфизмите:

$$G_1/\ker\varphi \cong \operatorname{Im}\varphi,$$

  като се обясни защо отображението

$$a\ker\varphi \longmapsto \varphi(a)$$

  е коректно дефинирано, хомоморфизъм, сюрективно и инективно.

- Да се възпроизведе доказателствената идея на теоремата на Кейли: групата действа върху себе си чрез леви транслации $x \mapsto ax$; получава се хомоморфизъм към симетрична група с тривиално ядро.

Въпрос 32

Теорема на Ферма. Теореми за средните стойности (Рол, Лагранж, Коши). Формула на Тейлър.

По конспекта: какво трябва да мога

Дефиниция: локален екстремум на функция на една променлива — §32.1, стр. 445.
Формулировка и доказателство: необходимо условие за локален екстремум при диференцируеми функции — теорема на Ферма — §32.1, стр. 445.
Доказателства: теоремите на Рол, Лагранж и Коши, заедно с $g(a) \neq g(b)$ при предположенията на Коши — §32.2, стр. 446.
Извеждане: формула на Тейлър с остатъчен член във формата на Лагранж — §32.3, стр. 451.
† **Допълнително:** теоремата на Вайерщрас се *използва без доказателство* при теоремата на Рол — това е изрично разрешено от анотацията.
Литература: [6], [8], [10].

32.1 Локални екстремуми и теорема на Ферма

Дефиниция 32.1 (Локален екстремум)**.** Нека $f : D \to \mathbb{R}$, където $D \subseteq \mathbb{R}$, и нека $x_0 \in D$.
Казваме, че f има *локален минимум* в точката x_0, ако съществува $\delta > 0$, за което

$$(x_0 - \delta, x_0 + \delta) \subseteq D$$

и

$$f(x_0) \leq f(x) \quad \text{за всяко } x \in (x_0 - \delta, x_0 + \delta).$$

Аналогично, f има *локален максимум* в x_0, ако съществува $\delta > 0$, за което

$$(x_0 - \delta, x_0 + \delta) \subseteq D$$

и

$$f(x_0) \geq f(x) \quad \text{за всяко } x \in (x_0 - \delta, x_0 + \delta).$$

Точка, в която функцията има локален минимум или локален максимум, се нарича *точка на локален екстремум*.

★ **Теорема 32.2 (Теорема на Ферма — необходимо условие за локален екстремум).** Нека $f : D \to \mathbb{R}$ и нека x_0 е точка на локален екстремум на f. Ако f е диференцируема в x_0, то

$$f'(x_0) = 0.$$

Доказателство. Нека първо x_0 е точка на локален минимум. Тогава съществува $\delta > 0$, такова че

$$f(x_0) \leq f(x)$$

за всяко $x \in (x_0 - \delta, x_0 + \delta)$.
Ако $x \in (x_0, x_0 + \delta)$, то $x - x_0 > 0$ и $f(x) - f(x_0) \geq 0$. Следователно

$$\frac{f(x) - f(x_0)}{x - x_0} \geq 0.$$

При преминаване към граница отдясно получаваме

$$f'(x_0) \geq 0.$$

Ако $x \in (x_0 - \delta, x_0)$, то $x - x_0 < 0$, докато отново $f(x) - f(x_0) \geq 0$. Затова

$$\frac{f(x) - f(x_0)}{x - x_0} \leq 0.$$

При преминаване към граница отляво следва

$$f'(x_0) \leq 0.$$

Тъй като производната съществува, левият и десният предел съвпадат. Следователно едновременно е изпълнено

$$f'(x_0) \geq 0 \qquad \text{и} \qquad f'(x_0) \leq 0,$$

тоест $f'(x_0) = 0$.
Ако x_0 е точка на локален максимум на f, то тя е точка на локален минимум на $-f$. От вече доказаното

$$(-f)'(x_0) = 0,$$

откъдето $-f'(x_0) = 0$, т.е. $f'(x_0) = 0$. □

Забележка 32.3. Теоремата на Ферма дава необходимо, но не достатъчно условие за локален екстремум. Условието $f'(x_0) = 0$ само посочва възможна точка на екстремум.

32.2 Теореми за средните стойности

В този раздел нека $a < b$ и f е непрекъсната в затворения интервал $[a, b]$ и диференцируема в отворения интервал (a, b).

За доказателството на първата теорема използваме теоремата на Вайерщрас: всяка непрекъсната функция върху краен затворен интервал достига най-голямата и най-малката си стойност.

32.2.1 Теорема на Рол

★ **Теорема 32.4 (Теорема на Рол).** Ако

$$f(a) = f(b),$$

то съществува точка $c \in (a, b)$, за която

$$f'(c) = 0.$$

Доказателство. Понеже f е непрекъсната в $[a, b]$, от теоремата на Вайерщрас следва, че тя достига своя минимум и максимум в някои точки

$$x_{\min}, x_{\max} \in [a, b].$$

Ако поне една от точките $x_{\min}$ и $x_{\max}$ принадлежи на (a, b), то в нея функцията има локален екстремум. Функцията е диференцируема в тази вътрешна точка, следователно по теоремата на Ферма производната в нея е нула. Получаваме търсената точка c.
Остава случаят, когато и минимумът, и максимумът се достигат само в краищата на интервала. Тъй като

$$f(a) = f(b),$$

минималната и максималната стойност са равни. Следователно функцията е константна в $[a, b]$. Тогава

$$f'(x) = 0 \quad \text{за всяко } x \in (a, b),$$

и всяка точка от (a, b) може да бъде избрана за c. □

† **Допълнение 32.5 (Бележка).** В едно от предоставените изложения при разглеждането на вътрешна точка на минимум е записано $f(x_{\min}) = 0$. От теоремата на Ферма следва не равенство за стойността на функцията, а

$$f'(x_{\min}) = 0.$$

Именно това равенство се използва в доказателството на теоремата на Рол.

Забележка 32.6. Геометрично теоремата на Рол гласи, че ако графиката на функцията започва и завършва на една и съща височина, то между тези две точки има точка с хоризонтална допирателна.

32.2.2 Теорема на Лагранж

★ **Теорема 32.7 (Теорема на Лагранж за крайните нараствания).** Съществува точка $c \in (a, b)$, такава че

$$f(b) - f(a) = f'(c)(b - a).$$

Еквивалентно,

$$f'(c) = \frac{f(b) - f(a)}{b - a}.$$

Доказателство. Да разгледаме функцията

$$h(x) = f(x) - kx,$$

където константата k ще изберем така, че да е изпълнено условието на теоремата на Рол:

$$h(a) = h(b).$$

Това равенство е еквивалентно на

$$f(a) - ka = f(b) - kb,$$

или, тъй като $a \neq b$,

$$k = \frac{f(b) - f(a)}{b - a}.$$

Функцията h е непрекъсната в $[a, b]$ и диференцируема в (a, b), защото f има същите свойства. При избраното k е изпълнено $h(a) = h(b)$. По теоремата на Рол съществува $c \in (a, b)$, за което

$$h'(c) = 0.$$

Но

$$h'(x) = f'(x) - k.$$

Следователно

$$0 = h'(c) = f'(c) - k,$$

тоест

$$f'(c) = k = \frac{f(b) - f(a)}{b - a}.$$

След умножение по $b-a$ получаваме

$$f(b)-f(a)=f'(c)(b-a).$$

□

Забележка 32.8. Числото

$$\frac{f(b)-f(a)}{b-a}$$

е средната скорост на изменение на функцията върху $[a,b]$, а $f'(c)$ е моментната скорост на изменение в подходяща вътрешна точка c.

32.2.3 Теорема на Коши

★ **Теорема 32.9 (Обобщена теорема на Лагранж — теорема на Коши).** Нека f и g са непрекъснати в $[a,b]$ и диференцируеми в (a,b). Нека освен това

$$g'(x)\neq 0 \quad \text{за всяко } x\in(a,b).$$

Тогава съществува $c\in(a,b)$, такова че

$$\frac{f(b)-f(a)}{g(b)-g(a)}=\frac{f'(c)}{g'(c)}.$$

Доказателство. Стъпка 1: знаменателят е ненулев. Да допуснем противното, тоест $g(a)=g(b)$. Тогава g е непрекъсната в $[a,b]$, диференцируема в (a,b) и приема равни стойности в двата края. По теоремата на Рол съществува $\xi\in(a,b)$, за което

$$g'(\xi)=0.$$

Това противоречи на условието $g'(x)\neq 0$ за всяко $x\in(a,b)$. Следователно

$$g(b)-g(a)\neq 0,$$

така че лявата страна на доказваното равенство е определена.
Стъпка 2: спомагателната функция. Полагаме

$$H(x)=\big(f(b)-f(a)\big)\big(g(x)-g(a)\big)-\big(g(b)-g(a)\big)\big(f(x)-f(a)\big).$$

Числата $f(b)-f(a)$ и $g(b)-g(a)$ са константи, затова H е линейна комбинация на f и g с добавена константа. Значи H е непрекъсната в $[a,b]$ и диференцируема в (a,b), защото f и g имат тези свойства.
Стъпка 3: равни стойности в краищата. Пресмятаме двете стойности:

$$H(a)=\big(f(b)-f(a)\big)\underbrace{\big(g(a)-g(a)\big)}_{=0}-\big(g(b)-g(a)\big)\underbrace{\big(f(a)-f(a)\big)}_{=0}=0,$$

$$H(b)=\big(f(b)-f(a)\big)\big(g(b)-g(a)\big)-\big(g(b)-g(a)\big)\big(f(b)-f(a)\big)=0,$$

тъй като двете произведения в $H(b)$ съвпадат. Следователно

$$H(a)=H(b)=0.$$

Стъпка 4: прилагане на теоремата на Рол. Функцията H удовлетворява трите условия на Рол, затова съществува $c \in (a, b)$, за което

$$H'(c) = 0.$$

Стъпка 5: разписване на производната. Диференцираме H, като константните множители излизат пред производната, а производните на $g(x) - g(a)$ и $f(x) - f(a)$ са съответно $g'(x)$ и $f'(x)$:

$$H'(x) = \big(f(b) - f(a)\big)g'(x) - \big(g(b) - g(a)\big)f'(x).$$

Полагаме $x = c$ и използваме $H'(c) = 0$:

$$0 = \big(f(b) - f(a)\big)g'(c) - \big(g(b) - g(a)\big)f'(c),$$

тоест

$$\big(f(b) - f(a)\big)g'(c) = \big(g(b) - g(a)\big)f'(c).$$

Стъпка 6: деление. По условие $g'(c) \neq 0$, а по стъпка 1 $g(b) - g(a) \neq 0$. Следователно произведението

$$\big(g(b) - g(a)\big)g'(c) \neq 0$$

и можем да разделим на него двете страни на последното равенство:

$$\frac{\big(f(b) - f(a)\big)g'(c)}{\big(g(b) - g(a)\big)g'(c)} = \frac{\big(g(b) - g(a)\big)f'(c)}{\big(g(b) - g(a)\big)g'(c)}.$$

Отляво се съкращава $g'(c)$, а отдясно — $g(b) - g(a)$, откъдето

$$\frac{f(b) - f(a)}{g(b) - g(a)} = \frac{f'(c)}{g'(c)}.$$

□

Твърдение 32.10. При условията на теоремата на Коши е изпълнено

$$g(a) \neq g(b).$$

Доказателство. Да допуснем противното:

$$g(a) = g(b).$$

Тогава функцията g удовлетворява условията на теоремата на Рол. Следователно съществува $c \in (a, b)$, за което

$$g'(c) = 0.$$

Това противоречи на предположението, че $g'(x) \neq 0$ за всяко $x \in (a, b)$. Следователно

$$g(a) \neq g(b).$$

□

Второ доказателство на теоремата на Коши. От предходното твърдение знаменателят $g(b) - g(a)$ е различен от нула. Нека

$$k = \frac{f(b) - f(a)}{g(b) - g(a)}$$

и дефинираме

$$h(x) = f(x) - kg(x).$$

Като линейна комбинация на f и g, функцията h е непрекъсната в $[a, b]$ и диференцируема в (a, b).
От избора на k следва

$$k\big(g(b) - g(a)\big) = f(b) - f(a).$$

Следователно

$$f(a) - kg(a) = f(b) - kg(b),$$

тоест

$$h(a) = h(b).$$

По теоремата на Рол съществува $c \in (a, b)$, за което

$$h'(c) = 0.$$

Тъй като

$$h'(x) = f'(x) - kg'(x),$$

получаваме

$$f'(c) - kg'(c) = 0.$$

Понеже $g'(c) \neq 0$, можем да разделим на $g'(c)$:

$$\frac{f'(c)}{g'(c)} = k.$$

От определението на k следва

$$\frac{f'(c)}{g'(c)} = \frac{f(b) - f(a)}{g(b) - g(a)}.$$

□

Следствие 32.11. Теоремата на Лагранж е частен случай на теоремата на Коши при

$$g(x) = x.$$

Доказателство. За $g(x) = x$ имаме $g'(x) = 1$ и $g(b) - g(a) = b - a$. Формулата на Коши приема вида

$$\frac{f(b) - f(a)}{b - a} = f'(c).$$

□

32.3 Формула на Тейлър

32.3.1 Полином на Тейлър

Дефиниция 32.12 (Полином на Тейлър)**.** Нека функцията f притежава производни до ред n в точката x_0. Полиномът

$$T_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

се нарича *полином на Тейлър от ред* n на функцията f около точката x_0.
Разписан, той има вида

$$T_n(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n.$$

Лема 32.13. Ако f притежава производни до ред n в x_0, то

$$T_n^{(i)}(x_0) = f^{(i)}(x_0), \qquad i = 0, 1, \ldots, n.$$

Доказателство. При диференциране i пъти всеки член

$$\frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

с $k < i$ изчезва. За $k \geq i$ се получава множител

$$k(k-1)\cdots(k-i+1)(x - x_0)^{k-i}.$$

При $x = x_0$ всички членове с $k > i$ са нула, защото съдържат положителна степен на $x - x_0$. Остава само членът с $k = i$, който е равен на

$$f^{(i)}(x_0).$$

Следователно

$$T_n^{(i)}(x_0) = f^{(i)}(x_0).$$

□

32.3.2 Остатъчен член във формата на Лагранж

Полиномът на Тейлър приближава f около x_0; разликата между функцията и приближението получава собствено име.

Дефиниция 32.14 (Остатъчен член)**.** Нека f притежава производни до ред n в x_0 и нека T_n е нейният полином на Тейлър около x_0. Функцията

$$R_n(x) = f(x) - T_n(x)$$

се нарича *остатъчен член от ред* n. По построение

$$f(x) = T_n(x) + R_n(x).$$

От лемата по-горе $T_n^{(i)}(x_0) = f^{(i)}(x_0)$ за $i = 0, 1, \ldots, n$, следователно

$$R_n^{(i)}(x_0) = 0, \qquad i = 0, 1, \ldots, n.$$

Остатъкът и първите му n производни се анулират в x_0; въпросът е с каква скорост той расте при отдалечаване от x_0. Теоремата по-долу дава точен отговор чрез стойност на $(n+1)$-вата производна във вътрешна точка.

★ **Теорема 32.15 (Формула на Тейлър с остатъчен член на Лагранж).** Нека са дадени цяло число $n \geq 0$, точка $x_0 \in \mathbb{R}$ и число $\delta > 0$. Нека функцията

$$f : (x_0 - \delta, x_0 + \delta) \to \mathbb{R}$$

притежава производни до ред $n+1$ включително във *всяка* точка на интервала, тоест $f^{(n+1)}(t)$ съществува за всяко $t \in (x_0 - \delta, x_0 + \delta)$.
Тогава за всяко

$$x \in (x_0 - \delta, x_0 + \delta), \qquad x \neq x_0,$$

съществува точка c, разположена *строго* между x_0 и x, тоест

$$c \in (x_0, x) \text{ при } x > x_0, \qquad c \in (x, x_0) \text{ при } x < x_0,$$

за която

$$f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k + \frac{f^{(n+1)}(c)}{(n+1)!}(x - x_0)^{n+1}.$$

Еквивалентно, съществува $\theta \in (0, 1)$, за което $c = x_0 + \theta(x - x_0)$.
При $x = x_0$ равенството е изпълнено тривиално: сумата се свежда до $f(x_0)$, а остатъчният член е нула.

Условието е върху отворен интервал около x_0, а не върху затворен: така за всяко x от интервала целият затворен интервал с краища x_0 и x лежи вътре в областта, върху която $f^{(n+1)}$ съществува. От съществуването на $f^{(k+1)}$ следва непрекъснатост на $f^{(k)}$, затова $f, f', \ldots, f^{(n)}$ са непрекъснати в целия интервал и отделно предположение за това не е нужно.

Доказателство. Случаят $x = x_0$ е непосредствен: всички степени $(x - x_0)^k$ при $k \geq 1$ са нула, така че и двете страни са равни на $f(x_0)$.
Нека по-нататък $x \neq x_0$ е фиксирано. Означаваме с

$$I = [\min(x_0, x),\ \max(x_0, x)]$$

затворения интервал с краища x_0 и x. Тъй като и двете точки лежат в $(x_0-\delta, x_0+\delta)$, а този интервал е изпъкнал, имаме $I \subseteq (x_0-\delta, x_0+\delta)$. Достатъчно е да докажем, че

$$R_n(x) = \frac{f^{(n+1)}(c)}{(n+1)!}(x-x_0)^{n+1}$$

за подходяща вътрешна точка c на I, защото $f(x) = T_n(x) + R_n(x)$.
Стъпка 1: спомагателните функции. При фиксирано x разглеждаме функциите на променливата t

$$\varphi(t) = f(x) - \sum_{k=0}^{n} \frac{f^{(k)}(t)}{k!}(x-t)^k = f(x) - f(t) - \frac{f'(t)}{1!}(x-t) - \cdots - \frac{f^{(n)}(t)}{n!}(x-t)^n$$

и

$$\psi(t) = (x-t)^{n+1}.$$

Обърнете внимание, че тук x е константа, а t пробягва I; в φ производните на f се пресмятат в t, а не в x_0.
Функцията φ е крайна сума от произведения на $f^{(k)}(t)$ с полиноми на t. За $k \le n$ производната $f^{(k)}$ е диференцируема в целия интервал $(x_0-\delta, x_0+\delta)$, защото $f^{(k+1)}$ съществува там. Следователно φ е диференцируема, а значи и непрекъсната, в отворено множество, съдържащо I. Същото важи очевидно и за полинома ψ. В частност φ и ψ са непрекъснати в I и диференцируеми във вътрешността му.
Стъпка 2: стойности в краищата. При $t = x$ всички степени $(x-t)^k$ с $k \ge 1$ се анулират, така че от сумата остава само членът с $k = 0$:

$$\varphi(x) = f(x) - f(x) = 0.$$

При $t = x_0$ сумата е точно полиномът на Тейлър:

$$\varphi(x_0) = f(x) - \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x-x_0)^k = f(x) - T_n(x) = R_n(x).$$

Аналогично

$$\psi(x) = 0, \qquad \psi(x_0) = (x-x_0)^{n+1} \neq 0.$$

Стъпка 3: разписване на φ'. Диференцираме почленно, като за $k \ge 1$ използваме правилото за произведение:

$$\frac{d}{dt}\left[\frac{f^{(k)}(t)}{k!}(x-t)^k\right] = \frac{f^{(k+1)}(t)}{k!}(x-t)^k - \frac{f^{(k)}(t)}{k!}\,k\,(x-t)^{k-1},$$

където вторият член идва от $\frac{d}{dt}(x-t)^k = -k(x-t)^{k-1}$. За $k = 0$ вторият член липсва, тъй като $(x-t)^0 = 1$ е константа. Понеже $\dfrac{k}{k!} = \dfrac{1}{(k-1)!}$, получаваме

$$\varphi'(t) = -\sum_{k=0}^{n} \frac{f^{(k+1)}(t)}{k!}(x-t)^k + \sum_{k=1}^{n} \frac{f^{(k)}(t)}{(k-1)!}(x-t)^{k-1}.$$

Във втората сума полагаме $j = k-1$ и тя се превръща в

$$\sum_{j=0}^{n-1} \frac{f^{(j+1)}(t)}{j!}(x-t)^j,$$

тоест в първата сума без нейния последен член. Двете суми се унищожават член по член и остава

$$\varphi'(t) = -\frac{f^{(n+1)}(t)}{n!}(x-t)^n.$$

Това е причината да се избере точно тази спомагателна функция: сумата е телескопична и нейната производна съдържа само $(n+1)$-вата производна на f.
За ψ имаме направо

$$\psi'(t) = -(n+1)(x-t)^n.$$

За всяко t от вътрешността на I е $t \neq x$, следователно $(x-t)^n \neq 0$ и

$$\psi'(t) \neq 0.$$

Стъпка 4: прилагане на теоремата на Коши. Функциите φ и ψ са непрекъснати в I, диференцируеми във вътрешността му и ψ' не се анулира там. По теоремата на Коши съществува вътрешна точка c на I, тоест точка строго между x_0 и x, за която

$$\frac{\varphi(x)-\varphi(x_0)}{\psi(x)-\psi(x_0)} = \frac{\varphi'(c)}{\psi'(c)}.$$

Теоремата на Коши е формулирана за интервал $[a,b]$ с $a<b$. Ако $x<x_0$, тя се прилага с $a=x$ и $b=x_0$; лявата страна не се променя, защото при размяна на краищата числителят и знаменателят сменят знака си едновременно.
Стъпка 5: заместване. Използваме стойностите от стъпка 2 и производните от стъпка 3:

$$\frac{0-R_n(x)}{0-(x-x_0)^{n+1}} = \frac{-\dfrac{f^{(n+1)}(c)}{n!}(x-c)^n}{-(n+1)(x-c)^n}.$$

Отляво знаците се съкращават. Отдясно c е вътрешна точка, затова $c \neq x$ и $(x-c)^n \neq 0$, така че този множител може да бъде съкратен; освен това

$$\frac{1}{n!\,(n+1)} = \frac{1}{(n+1)!}.$$

Получаваме

$$\frac{R_n(x)}{(x-x_0)^{n+1}} = \frac{f^{(n+1)}(c)}{(n+1)!},$$

а след умножение по $(x-x_0)^{n+1} \neq 0$

$$R_n(x) = \frac{f^{(n+1)}(c)}{(n+1)!}(x-x_0)^{n+1}.$$

Тъй като $f(x) = T_n(x) + R_n(x)$, това е точно формулата на Тейлър с остатъчен член на Лагранж. □

Забележка 32.16. Точката c в остатъчния член зависи както от x, така и от n; тя не е дадена явно, а само съществува. Затова често се означава с $\xi(x)$:

$$f(x) = f(x_0) + f'(x_0)(x-x_0) + \cdots + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n + \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x-x_0)^{n+1}.$$

Винаги $\xi(x)$ е между x_0 и x.

▷ **Пример 32.17.** За $n = 1$ формулата на Тейлър приема вида

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(c)}{2}(x - x_0)^2,$$

където c е между x_0 и x.

32.4 Изпитен фокус

- Да се дефинират локален минимум, локален максимум и локален екстремум, като се подчертае съществуването на околност на точката, съдържаща се в дефиниционната област.
- Да се формулира и докаже теоремата на Ферма чрез знака на отношението

  $$\frac{f(x) - f(x_0)}{x - x_0}$$

  отляво и отдясно на точка на локален минимум; случаят на локален максимум се свежда до функцията $-f$.
- Да се формулира теоремата на Рол и да се възпроизведе доказателството чрез теоремата на Вайерщрас и теоремата на Ферма. Трябва да се разгледа и случаят, когато минимумът и максимумът се достигат само в краищата.
- Да се формулира теоремата на Лагранж и да се изведе от Рол чрез спомагателната функция

  $$h(x) = f(x) - kx, \qquad k = \frac{f(b) - f(a)}{b - a}.$$
- Да се формулира теоремата на Коши, включително условието

  $$g'(x) \neq 0 \quad \text{за } x \in (a, b).$$

  Да се докаже предварително, че то влече $g(a) \neq g(b)$, чрез противоречие с теоремата на Рол.
- Да се докаже теоремата на Коши чрез функцията

  $$h(x) = f(x) - kg(x), \qquad k = \frac{f(b) - f(a)}{g(b) - g(a)}.$$
- Да се знае полиномът на Тейлър

  $$T_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

  и свойството, че неговите производни до ред n в x_0 съвпадат със съответните производни на f.
- Да се формулира формулата на Тейлър с остатъчен член във формата на Лагранж: предположението е за съществуване на $f^{(n+1)}$ във всяка точка на отворен интервал около x_0, а точката c е строго между x_0 и x.

- Да се възпроизведе доказателството: фиксира се x, въвеждат се

$$\varphi(t) = f(x) - \sum_{k=0}^{n} \frac{f^{(k)}(t)}{k!}(x-t)^k, \qquad \psi(t) = (x-t)^{n+1},$$

пресмятат се $\varphi(x) = 0$, $\varphi(x_0) = R_n(x)$, $\psi(x) = 0$, $\psi(x_0) = (x-x_0)^{n+1}$, разписва се телескопичната производна

$$\varphi'(t) = -\frac{f^{(n+1)}(t)}{n!}(x-t)^n,$$

и към φ и ψ се прилага теоремата на Коши.

Въпрос 33

Определен интеграл. Суми на Дарбу. Интегруемост. Теорема на Нютон-Лайбниц.

По конспекта: какво трябва да мога

Официално заглавие: „Определен интеграл. *Дефиниция и свойства. Интегруемост на непрекъснатите функции.* Теорема на Нютон–Лайбниц.“
Дефиниции, последователно: разбиване на интервал — §33.1.1; големи и малки суми на Дарбу — §33.1.2; риманов интеграл чрез подхода на Дарбу — §33.1.3, стр. 460.
Доказателство: при добавяне на нови точки големите суми на Дарбу не нарастват, а малките не намаляват. Чертеж е *желателен* — §33.1.2, стр. 458.
Доказателство: критерият: f е интегруема точно когато за всяко $\varepsilon > 0$ има суми със $S - s < \varepsilon$ — §33.2, стр. 460.
Доказателство: всяка непрекъсната в краен затворен интервал функция е интегруема по Риман; теоремата на Кантор се използва *без доказателство* — §33.3, стр. 461.
Изброяване, без доказателство: основните свойства на римановия интеграл — §33.4, стр. 463.
Доказателство: теорема за средната стойност: съществува c с $\int_a^b f = f(c)(b-a)$; интегрирането на неравенства и теоремата за междинните стойности се използват *без доказателство* — §33.5, стр. 464.
Доказателство и приложение: теорема на Нютон–Лайбниц и използването ѝ за изчисляване на определени интеграли — §33.6, стр. 466.
Литература: [6], [8], [10].

33.1 Определен интеграл и суми на Дарбу

Нека $f\colon [a,b] \to \mathbb{R}$, където $a < b$. Целта на определения интеграл е да се постави на строга основа идеята за натрупана величина: например лице под графика, изминат път при зададена скорост или общ принос на непрекъснато изменяща се функция.

33.1.1 Разбиване на интервал

Дефиниция 33.1. *Разбиване* на интервала $[a, b]$ наричаме всяка крайна система от точки

$$\tau: \quad a = x_0 < x_1 < \cdots < x_n = b.$$

Точките $x_0, x_1, \ldots, x_n$ се наричат *делящи точки* на разбиването. С разбиването са свързани подинтервалите

$$[x_{i-1}, x_i], \qquad i = 1, \ldots, n,$$

чийто съюз е $[a, b]$ и два от които или не се пресичат, или имат обща само крайна точка.

Дефиниция 33.2. *Диаметър* на разбиването τ се нарича числото

$$d(\tau) = \max_{1 \le i \le n} (x_i - x_{i-1}).$$

Колкото по-малък е диаметърът, толкова по-фино е разбит интервалът. При изследване на интегруемостта ще използваме разбивания, за които $d(\tau)$ е достатъчно малък.

33.1.2 Малки и големи суми на Дарбу

Нека функцията f е ограничена върху $[a, b]$ и нека

$$\tau: \quad a = x_0 < x_1 < \cdots < x_n = b$$

е разбиване. За всеки подинтервал полагаме

$$m_i = \inf\{f(x) \mid x \in [x_{i-1}, x_i]\}, \qquad M_i = \sup\{f(x) \mid x \in [x_{i-1}, x_i]\}.$$

Тъй като f е ограничена, числата m_i и M_i са крайни.

Дефиниция 33.3. *Малка сума на Дарбу* на f по разбиването τ се нарича числото

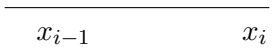
$$s_f(\tau) = \sum_{i=1}^{n} m_i (x_i - x_{i-1}).$$

Голяма сума на Дарбу на f по разбиването τ се нарича числото

$$S_f(\tau) = \sum_{i=1}^{n} M_i (x_i - x_{i-1}).$$

Геометрично малката сума се получава, ако над всеки подинтервал се построи правоъгълник с височина m_i, а голямата — с височина M_i. Следователно малката сума дава долна, а голямата — горна оценка на търсената натрупана величина.

над $[x_{i-1}, x_i]$ височината на долния правоъгълник е m_i,
а височината на горния правоъгълник е M_i :



По-точно, при добавяне на делящи точки долните правоъгълници могат да се повишат, а горните — да се понижат.

Лема 33.4. Нека τ^* се получава от разбиването τ чрез добавяне на една или повече нови делящи точки. Тогава

$$s_f(\tau^*) \geq s_f(\tau), \qquad S_f(\tau^*) \leq S_f(\tau).$$

Доказателство. Достатъчно е да разгледаме добавянето на една нова точка, тъй като общият случай следва чрез многократно прилагане на същото разсъждение.
Нека $x^* \in (x_{i-1}, x_i)$ е добавена точка. Единствено приносът на интервала $[x_{i-1}, x_i]$ се променя. Нека

$$m_i' = \inf_{x \in [x_{i-1}, x^*]} f(x), \qquad m_i'' = \inf_{x \in [x^*, x_i]} f(x).$$

Понеже

$$[x_{i-1}, x^*] \subseteq [x_{i-1}, x_i], \qquad [x^*, x_i] \subseteq [x_{i-1}, x_i],$$

имаме

$$m_i' \geq m_i, \qquad m_i'' \geq m_i.$$

Следователно

$$\begin{aligned} s_f(\tau^*) - s_f(\tau) &= m_i'(x^* - x_{i-1}) + m_i''(x_i - x^*) - m_i(x_i - x_{i-1}) \\ &\geq m_i(x^* - x_{i-1}) + m_i(x_i - x^*) - m_i(x_i - x_{i-1}) \\ &= 0. \end{aligned}$$

Значи $s_f(\tau^*) \geq s_f(\tau)$.
За големите суми разсъждението е аналогично. Ако M_i' и M_i'' са супремумите на функцията върху двата нови подинтервала, то

$$M_i' \leq M_i, \qquad M_i'' \leq M_i.$$

Затова

$$S_f(\tau^*) \leq S_f(\tau).$$

□

† **Допълнение 33.5 (Бележка).** В част от предоставените записи означенията за горен и долен интеграл са разменени. Съгласно определенията чрез суми на Дарбу горният интеграл се задава чрез инфимум на *големите* суми, а долният — чрез супремум на *малките* суми. Това е и единственото съгласувано с неравенството $s_f(\tau) \leq S_f(\tau)$ означение.

33.1.3 Горен и долен интеграл на Дарбу

За всяко разбиване τ е изпълнено

$$s_f(\tau) \leq S_f(\tau).$$

Освен това всяка малка сума е не по-голяма от всяка голяма сума. Действително, ако τ_1 и τ_2 са две разбивания, разглеждаме общото им доразбиване $\tau_1 \cup \tau_2$. От Лема 33.4 следва

$$s_f(\tau_1) \leq s_f(\tau_1 \cup \tau_2) \leq S_f(\tau_1 \cup \tau_2) \leq S_f(\tau_2).$$

Дефиниция 33.6. *Долен интеграл на Дарбу* на ограничената функция f върху $[a,b]$ наричаме

$$\underline{\int_a^b} f = \sup\{s_f(\tau) \mid \tau \text{ е разбиване на } [a,b]\}.$$

Горен интеграл на Дарбу наричаме

$$\overline{\int_a^b} f = \inf\{S_f(\tau) \mid \tau \text{ е разбиване на } [a,b]\}.$$

От разгледаното неравенство между малките и големите суми следва, че

$$\underline{\int_a^b} f \leq \overline{\int_a^b} f.$$

Дефиниция 33.7. Ограничената функция $f\colon [a,b] \to \mathbb{R}$ се нарича *интегруема по Риман* върху $[a,b]$, ако

$$\underline{\int_a^b} f = \overline{\int_a^b} f.$$

Общата стойност се нарича *риманов определен интеграл* на f върху $[a,b]$ и се означава с

$$\int_a^b f(x)\,dx.$$

Така определението чрез Дарбу формализира идеята, че долните и горните стъпаловидни оценки могат да бъдат направени произволно близки.

33.2 Критерий за интегруемост

Теорема 33.8 (Критерий на Дарбу)**.** Нека $f\colon [a,b] \to \mathbb{R}$ е ограничена. Тогава f е интегруема по Риман тогава и само тогава, когато за всяко $\varepsilon > 0$ съществува разбиване τ на $[a,b]$, за което

$$S_f(\tau) - s_f(\tau) < \varepsilon.$$

Доказателство. Нека първо f е интегруема и

$$I = \int_a^b f(x)\,dx = \underline{\int_a^b} f = \overline{\int_a^b} f.$$

Нека $\varepsilon > 0$. От свойството на супремума съществува разбиване τ_1, за което

$$s_f(\tau_1) > I - \frac{\varepsilon}{2}.$$

От свойството на инфимума съществува разбиване τ_2, за което

$$S_f(\tau_2) < I + \frac{\varepsilon}{2}.$$

Нека $\tau = \tau_1 \cup \tau_2$ е общото доразбиване. Според Лема 33.4,

$$s_f(\tau) \geq s_f(\tau_1), \qquad S_f(\tau) \leq S_f(\tau_2).$$

Следователно

$$S_f(\tau) - s_f(\tau) \leq S_f(\tau_2) - s_f(\tau_1) < \left(I + \frac{\varepsilon}{2}\right) - \left(I - \frac{\varepsilon}{2}\right) = \varepsilon.$$

Обратно, нека за всяко $\varepsilon > 0$ има разбиване τ, за което

$$S_f(\tau) - s_f(\tau) < \varepsilon.$$

За всяко разбиване е изпълнено

$$s_f(\tau) \leq \underline{\int_a^b} f \leq \overline{\int_a^b} f \leq S_f(\tau).$$

Затова

$$0 \leq \overline{\int_a^b} f - \underline{\int_a^b} f \leq S_f(\tau) - s_f(\tau) < \varepsilon.$$

Това е вярно за всяко $\varepsilon > 0$, откъдето

$$\overline{\int_a^b} f = \underline{\int_a^b} f.$$

Следователно f е интегруема по Риман. □

33.3 Интегруемост на непрекъснатите функции

Ще използваме без доказателство теоремата на Кантор: всяка непрекъсната функция върху краен затворен интервал е равномерно непрекъсната. Ще използваме също, че непрекъсната функция върху $[a, b]$ е ограничена.

Теорема 33.9. Всяка непрекъсната функция $f\colon [a, b] \to \mathbb{R}$ е интегруема по Риман.

Доказателство. Нека $\varepsilon > 0$. Понеже f е равномерно непрекъсната върху $[a, b]$, съществува $\delta > 0$, такова че за всички $x', x'' \in [a, b]$ от

$$|x' - x''| < \delta$$

следва

$$|f(x') - f(x'')| < \frac{\varepsilon}{b-a}.$$

Избираме разбиване

$$\tau: \quad a = x_0 < x_1 < \cdots < x_n = b, \qquad d(\tau) < \delta.$$

За произволен индекс i и произволни точки

$$x', x'' \in [x_{i-1}, x_i]$$

имаме

$$|x' - x''| \leq x_i - x_{i-1} \leq d(\tau) < \delta.$$

Следователно

$$|f(x') - f(x'')| < \frac{\varepsilon}{b-a}.$$

Оттук осцилацията на f върху всеки подинтервал удовлетворява

$$M_i - m_i \leq \frac{\varepsilon}{b-a}.$$

Тогава

$$\begin{aligned} S_f(\tau) - s_f(\tau) &= \sum_{i=1}^{n} (M_i - m_i)(x_i - x_{i-1}) \\ &\leq \frac{\varepsilon}{b-a} \sum_{i=1}^{n} (x_i - x_{i-1}) \\ &= \frac{\varepsilon}{b-a}(b-a) = \varepsilon. \end{aligned}$$

По Теорема 33.8 функцията f е интегруема по Риман. □

33.4 Основни свойства на римановия интеграл

† **Допълнение 33.10 (Обхват: изброяване без доказателство).** Анотацията иска основните свойства на римановия интеграл да бъдат *изброени без доказателство*. Доказателствата в този раздел са запазени, но са допълнителни. Изисквани с доказателство в тази глава са само: монотонността на сумите на Дарбу при сгъстяване (§33.1.2), критерият за интегруемост (§33.2), интегруемостта на непрекъснатите функции (§33.3), теоремата за средната стойност (§33.5) и теоремата на Нютон–Лайбниц (§33.6). Теоремата на Кантор, свойството за интегриране на неравенства и теоремата за междинните стойности се *използват* без доказателство — това е изрично разрешено.

Следващите свойства са формулирани с доказателства. Нека разглежданите функции са интегруеми по Риман върху съответните интервали.

Твърдение 33.11 (Линейност)**.** Ако $f, g\colon [a,b] \to \mathbb{R}$ са интегруеми и $\lambda, \mu \in \mathbb{R}$, то $\lambda f + \mu g$ е интегруема и

$$\int_a^b \bigl(\lambda f(x) + \mu g(x)\bigr)\,dx = \lambda \int_a^b f(x)\,dx + \mu \int_a^b g(x)\,dx.$$

В частност,

$$\int_a^b (f(x) + g(x))\,dx = \int_a^b f(x)\,dx + \int_a^b g(x)\,dx,$$

$$\int_a^b \lambda f(x)\,dx = \lambda \int_a^b f(x)\,dx.$$

Доказателство. За всяко деление τ и избор на точки ξ Римановите суми удовлетворяват $\sigma(\lambda f + \mu g, \tau, \xi) = \lambda\sigma(f,\tau,\xi) + \mu\sigma(g,\tau,\xi)$. При диаметър на делението, стремящ се към нула, двете суми вдясно клонят равномерно по избора на ξ към съответните интеграли. Линейната им комбинация също има граница, което едновременно доказва интегруемостта и формулата. □

Твърдение 33.12 (Адитивност по интервала)**.** Ако $c \in (a,b)$, то f е интегруема върху $[a,b]$ тогава и само тогава, когато е интегруема върху $[a,c]$ и $[c,b]$. В този случай

$$\int_a^b f(x)\,dx = \int_a^c f(x)\,dx + \int_c^b f(x)\,dx.$$

Доказателство. Използваме критерия на Дарбу. Ако f е интегруема върху целия интервал, добавяме c към деление с произволно малка разлика между горна и долна сума. Добавянето на точка не увеличава разликата. Всяка от двете частични разлики е неотрицателна и не по-голяма от общата, затова ограниченията са интегруеми. Обратно, обединяваме деления на двата интервала с разлики под $\varepsilon/2$; общата разлика е под ε. При деления, съдържащи c, всяка Риманова сума е сбор на двете частични суми. Преминаването към граница дава равенството. □

Дефиниция 33.13 (Смяна на границите). За $a < b$ и интегруема функция f интегралът с обратни граници се определя чрез

$$\int_b^a f(x)\,dx = -\int_a^b f(x)\,dx.$$

Твърдение 33.14 (Позитивност и интегриране на неравенства). Ако $f(x) \geq 0$ за всяко $x \in [a,b]$, то

$$\int_a^b f(x)\,dx \geq 0.$$

По-общо, ако $f(x) \leq g(x)$ за всяко $x \in [a,b]$, то

$$\int_a^b f(x)\,dx \leq \int_a^b g(x)\,dx.$$

Доказателство. Всяка Риманова сума на неотрицателна функция е неотрицателна, защото дължините на подинтервалите са положителни. Следователно и границата им е неотрицателна. За второто твърдение прилагаме това към интегруемата функция $g - f \geq 0$ и използваме линейността. □

Следствие 33.15. Ако f е интегруема върху $[a,b]$, то $|f|$ също е интегруема и

$$\left|\int_a^b f(x)\,dx\right| \leq \int_a^b |f(x)|\,dx.$$

Доказателство. От $\big||f(x)| - |f(y)|\big| \leq |f(x) - f(y)|$ следва, че колебанието на $|f|$ върху всеки подинтервал не надхвърля това на f. Критерият на Дарбу тогава дава интегруемост на $|f|$. Интегрираме $-|f| \leq f \leq |f|$ и получаваме $-\int_a^b |f| \leq \int_a^b f \leq \int_a^b |f|$, еквивалентно на твърдението. □

▷ **Пример 33.16.** За константната функция $f(x) = k$ е изпълнено

$$\int_a^b k\,dx = k(b-a).$$

33.5 Теорема за средната стойност за интеграли

Теорема 33.17 (Теорема за средната стойност). Нека $f\colon [a,b] \to \mathbb{R}$ е непрекъсната. Тогава съществува $c \in [a,b]$, такова че

$$\int_a^b f(x)\,dx = f(c)(b-a).$$

Доказателство. Понеже f е непрекъсната върху затворения интервал $[a, b]$, тя приема най-малка и най-голяма стойност. Нека

$$m = \min_{x\in[a,b]} f(x), \qquad M = \max_{x\in[a,b]} f(x).$$

Тогава

$$m \leq f(x) \leq M, \qquad x \in [a, b].$$

Чрез свойството за интегриране на неравенства получаваме

$$\int_a^b m\,dx \leq \int_a^b f(x)\,dx \leq \int_a^b M\,dx.$$

Следователно

$$m(b-a) \leq \int_a^b f(x)\,dx \leq M(b-a).$$

Тъй като $b - a > 0$,

$$m \leq \frac{1}{b-a}\int_a^b f(x)\,dx \leq M.$$

Непрекъснатата функция приема всички стойности между своя минимум и максимум. Следователно съществува $c \in [a, b]$, за което

$$f(c) = \frac{1}{b-a}\int_a^b f(x)\,dx.$$

Умножаваме по $b - a$ и получаваме

$$\int_a^b f(x)\,dx = f(c)(b-a).$$

□

Числото

$$\frac{1}{b-a}\int_a^b f(x)\,dx$$

се нарича средна стойност на функцията f върху $[a, b]$. Теоремата показва, че за непрекъсната функция тази средна стойност действително се достига като стойност $f(c)$ в поне една точка от интервала.

33.6 Теорема на Нютон–Лайбниц

★ **Теорема 33.18 (Теорема на Нютон–Лайбниц).** Нека $f\colon [a,b] \to \mathbb{R}$ е непрекъсната и

$$F(x) = \int_a^x f(t)\,dt, \qquad x \in [a,b].$$

Тогава F е диференцируема във всяка вътрешна точка на $[a,b]$ и

$$F'(x) = f(x).$$

В краищата a и b същото равенство се разбира чрез съответната едностранна производна.

Доказателство. Нека $x_0 \in [a,b]$ е фиксирана точка и $h \neq 0$ е такова, че

$$x_0 + h \in [a,b].$$

От адитивността на интеграла следва

$$F(x_0+h) - F(x_0) = \int_{x_0}^{x_0+h} f(t)\,dt.$$

Следователно

$$\frac{F(x_0+h)-F(x_0)}{h} = \frac{1}{h}\int_{x_0}^{x_0+h} f(t)\,dt.$$

Изваждаме $f(x_0)$ и използваме, че

$$f(x_0) = \frac{1}{h}\int_{x_0}^{x_0+h} f(x_0)\,dt.$$

Получаваме

$$\frac{F(x_0+h)-F(x_0)}{h} - f(x_0) = \frac{1}{h}\int_{x_0}^{x_0+h} \big(f(t)-f(x_0)\big)\,dt.$$

Нека $\varepsilon > 0$. От непрекъснатостта на f в x_0 съществува $\delta > 0$, такова че

$$|t - x_0| < \delta \quad \Longrightarrow \quad |f(t) - f(x_0)| < \varepsilon.$$

Ако $|h| < \delta$, то всяка точка t между x_0 и x_0+h удовлетворява

$$|t - x_0| \le |h| < \delta.$$

При $h > 0$, използвайки неравенството за интеграла, имаме

$$\begin{aligned}\left|\frac{F(x_0+h)-F(x_0)}{h} - f(x_0)\right| &\le \frac{1}{h}\int_{x_0}^{x_0+h} |f(t)-f(x_0)|\,dt \\ &\le \frac{1}{h}\int_{x_0}^{x_0+h} \varepsilon\,dt = \varepsilon.\end{aligned}$$

При $h < 0$ аналогично,

$$\left|\frac{F(x_0+h)-F(x_0)}{h} - f(x_0)\right| \leq \frac{1}{|h|}\int_{x_0+h}^{x_0} |f(t)-f(x_0)|\,dt \leq \frac{1}{|h|}\int_{x_0+h}^{x_0} \varepsilon\,dt = \varepsilon.$$

Следователно

$$\lim_{h\to 0}\frac{F(x_0+h)-F(x_0)}{h} = f(x_0).$$

Значи $F'(x_0) = f(x_0)$. Тъй като x_0 е произволна точка, твърдението е доказано. □

Теоремата показва, че функцията

$$x \longmapsto \int_a^x f(t)\,dt$$

е примитивна на непрекъснатата функция f.

33.6.1 Изчисляване на определени интеграли

Следствие 33.19 (Формула на Нютон–Лайбниц)**.** Нека $f\colon [a,b] \to \mathbb{R}$ е непрекъсната и Φ е примитивна на f върху $[a,b]$, тоест

$$\Phi'(x) = f(x).$$

Тогава

$$\int_a^b f(x)\,dx = \Phi(b) - \Phi(a).$$

Обичайното означение е

$$\int_a^b f(x)\,dx = \Phi(x)\Big|_a^b.$$

Доказателство. По Теорема 33.18 функцията

$$F(x) = \int_a^x f(t)\,dt$$

също е примитивна на f. Следователно за някоя константа C е изпълнено

$$\Phi(x) = F(x) + C.$$

При $x = a$ получаваме

$$\Phi(a) = F(a) + C = C,$$

защото

$$F(a) = \int_a^a f(t)\,dt = 0.$$

Следователно

$$\Phi(x) = \int_a^x f(t)\,dt + \Phi(a),$$

или

$$\int_a^x f(t)\,dt = \Phi(x) - \Phi(a).$$

При $x = b$ получаваме търсената формула. □

▷ **Пример 33.20.** Да се изчисли

$$\int_0^1 3x^2\,dx.$$

Примитивна на $3x^2$ е $\Phi(x) = x^3$. Следователно

$$\int_0^1 3x^2\,dx = x^3\big|_0^1 = 1^3 - 0^3 = 1.$$

Практическият алгоритъм за пресмятане на определен интеграл на непрекъсната функция е следният:

1. Намира се примитивна функция Φ на подинтегралната функция f.
2. Изчисляват се стойностите $\Phi(a)$ и $\Phi(b)$.
3. Прилага се формулата
$$\int_a^b f(x)\,dx = \Phi(b) - \Phi(a).$$

33.7 Изпитен фокус

- Да се дефинират разбиване на $[a, b]$, делящи точки и диаметър $d(\tau)$.
- Да се дефинират m_i, M_i, малка сума $s_f(\tau)$ и голяма сума $S_f(\tau)$ на Дарбу.
- Да се възпроизведе доказателството, че при доразбиване малките суми не намаляват, а големите не нарастват.
- Да се дефинират долен и горен интеграл на Дарбу и интегруемост по Риман чрез равенството им.
- Да се докаже критерият на Дарбу:
$$f \text{ е интегруема} \iff \forall \varepsilon > 0\ \exists \tau : S_f(\tau) - s_f(\tau) < \varepsilon.$$
- Да се обясни доказателството за интегруемост на непрекъсната функция: теорема на Кантор, избор на разбиване с малък диаметър и оценка на $S_f(\tau) - s_f(\tau)$.
- Да се изброят линейност, адитивност, позитивност, интегриране на неравенства, смяна на границите и неравенството с $|f|$.
- Да се докаже теоремата за средната стойност за интеграли чрез минимум, максимум, интегриране на неравенства и свойството за междинните стойности.

- Да се докаже теоремата на Нютон–Лайбниц чрез разликата

$$\frac{1}{h}\int_x^{x+h}\big(f(t)-f(x)\big)\,dt$$

  и непрекъснатостта на f.

- Да се използва формулата

$$\int_a^b f(x)\,dx = \Phi(b)-\Phi(a)$$

  за изчисляване на определен интеграл чрез примитивна функция.

- Да се различава кое се иска *с* доказателство и кое *без*: с доказателство — монотонността на сумите на Дарбу при сгъстяване, критерият $S-s<\varepsilon$, интегруемостта на непрекъснатите функции, теоремата за средната стойност и теоремата на Нютон–Лайбниц; само изброяване — основните свойства на интеграла; използват се без доказателство — теоремата на Кантор, интегрирането на неравенства и теоремата за междинните стойности.

Въпрос 34

Итерационни методи за решаване на нелинейни уравнения.

По конспекта: какво трябва да мога

Дефиниция и доказателство: неподвижна точка; непрекъснато изображение на $[a, b]$ в себе си има поне една неподвижна точка; свеждане на $f(x) = 0$ до намиране на неподвижна точка — §34.1, стр. 470.
Дефиниция и доказателство: свиващо изображение; при константа на Липшиц $q < 1$ — единственост на корена, сходимост и оценката $|x_n - \xi| \leq (b - a)q^n$; следствието за $|\varphi'(\xi)| < 1$ — §34.2, стр. 472.
Дефиниция: ред на сходимост — §34.3, стр. 475.
Геометрична илюстрация, формула и ред на сходимост: метод на хордите — фиг. 34.1, стр. 476, §34.4.1.
Геометрична илюстрация, формула и ред на сходимост: метод на секущите — фиг. 34.2, стр. 478, §34.4.2.
Геометрична илюстрация, формула и ред на сходимост: метод на Нютон — фиг. 34.3, стр. 479, §34.4.3. Чертежът е изискване на анотацията, не украса.
Доказателство: при метода на хордите сходимостта е със скоростта на геометрична прогресия, при корен, отделен в достатъчно малък интервал — §34.4.1, стр. 475.
Литература: [1], [2], [16].

34.1 Неподвижни точки и свеждане на нелинейно уравнение до итерационен процес

Много от методите за приближено решаване на нелинейни уравнения са итерационни. При тях се избира начално приближение x_0, по определено правило се намира x_1, след това x_2 и т.н. Целта е да се построи редица

$$x_0, x_1, \ldots, x_n, \ldots,$$

която клони към корена ξ на уравнението

$$f(x) = 0.$$

Дефиниция 34.1. Точката α се нарича *неподвижна точка* на изображението φ, ако

$$\varphi(\alpha) = \alpha.$$

Дефиниция 34.2. Казваме, че φ е *изображение на интервала* $[a, b]$ *в себе си*, ако

$$\varphi(x) \in [a, b] \qquad \text{за всяко } x \in [a, b].$$

Пише се

$$\varphi : [a, b] \to [a, b].$$

Теорема 34.3. Ако $\varphi : [a, b] \to [a, b]$ е непрекъснато изображение, то φ има поне една неподвижна точка в $[a, b]$.

Доказателство. Ако $\varphi(a) = a$, то a е неподвижна точка. Аналогично, ако $\varphi(b) = b$, то b е неподвижна точка.
Нека сега

$$\varphi(a) \neq a, \qquad \varphi(b) \neq b.$$

Тъй като φ изобразява $[a, b]$ в себе си, от $\varphi(a) \in [a, b]$ и $\varphi(a) \neq a$ следва

$$a - \varphi(a) < 0.$$

По същия начин от $\varphi(b) \in [a, b]$ и $\varphi(b) \neq b$ следва

$$b - \varphi(b) > 0.$$

Да разгледаме функцията

$$r(x) = x - \varphi(x).$$

Тя е непрекъсната в $[a, b]$, като

$$r(a) = a - \varphi(a) < 0, \qquad r(b) = b - \varphi(b) > 0.$$

Следователно съществува $\xi \in [a, b]$, за която

$$r(\xi) = 0.$$

Тогава

$$\xi - \varphi(\xi) = 0,$$

тоест

$$\varphi(\xi) = \xi.$$

Следователно ξ е неподвижна точка на φ. □

Решаването на уравнението $f(x) = 0$ може да се сведе до търсене на неподвижна точка. Достатъчно е уравнението да се запише в еквивалентния вид

$$x = \varphi(x).$$

Например, към двете страни на $f(x) = 0$ може да се прибави x:

$$f(x) = 0 \quad \Longleftrightarrow \quad x = x - f(x).$$

Така може да се положи

$$\varphi(x) = x - f(x).$$

По-общо, избира се такова еквивалентно преобразование, че корените на f да съвпадат с неподвижните точки на φ.

Ако ξ е корен на $f(x) = 0$ и уравнението е записано като $x = \varphi(x)$, то

$$\xi = \varphi(\xi).$$

Следователно намирането на корен на нелинейно уравнение се свежда до намиране на неподвижна точка.

Основният итерационен процес е методът на последователните приближения:

$$x_{n+1} = \varphi(x_n), \qquad n = 0, 1, 2, \ldots,$$

където x_0 е избрано начално приближение. Не всяко еквивалентно преобразование $x = \varphi(x)$ обаче гарантира сходимост. Решаващо е свойството φ да бъде свиващо изображение.

34.2 Липшицови и свиващи изображения

Дефиниция 34.4. Изображението $\varphi : [a, b] \to \mathbb{R}$ се нарича *Липшицово* в $[a, b]$ с константа $L > 0$, ако за всички $x, y \in [a, b]$ е изпълнено

$$|\varphi(x) - \varphi(y)| \leq L|x - y|.$$

Липшицовото условие означава, че изменението на стойностите на φ се контролира от изменението на аргумента. В частност Липшицовата функция е непрекъсната.

Дефиниция 34.5. Липшицово изображение $\varphi : [a, b] \to \mathbb{R}$ се нарича *свиващо* в $[a, b]$, ако може да се избере Липшицова константа

$$q \in (0, 1).$$

Тоест за всички $x, y \in [a, b]$ е изпълнено

$$|\varphi(x) - \varphi(y)| \leq q|x - y|, \qquad q < 1.$$

Теорема 34.6. Нека $\varphi : [a,b] \to [a,b]$ е непрекъснато и свиващо изображение с Липшицова константа $q < 1$. Тогава:

1. Уравнението
$$x = \varphi(x)$$
има единствен корен $\xi \in [a,b]$.

2. При всяко начално приближение $x_0 \in [a,b]$ редицата, зададена чрез
$$x_{n+1} = \varphi(x_n), \qquad n = 0, 1, 2, \ldots,$$
клони към ξ. Освен това за всяко n е изпълнена оценката
$$|x_n - \xi| \leq (b-a)q^n.$$

Доказателство. Понеже φ е непрекъснато изображение на $[a,b]$ в себе си, от предходната теорема следва, че тя има поне една неподвижна точка $\xi \in [a,b]$.
Ще докажем единствеността. Да допуснем, че ξ_1 и ξ_2 са две неподвижни точки:
$$\xi_1 = \varphi(\xi_1), \qquad \xi_2 = \varphi(\xi_2).$$
Тогава
$$|\xi_1 - \xi_2| = |\varphi(\xi_1) - \varphi(\xi_2)| \leq q|\xi_1 - \xi_2|.$$
Ако $\xi_1 \neq \xi_2$, то $|\xi_1 - \xi_2| > 0$ и след деление би се получило
$$1 \leq q,$$
което противоречи на $q < 1$. Следователно
$$\xi_1 = \xi_2.$$
Нека сега $x_0 \in [a,b]$ е произволно. Понеже φ изобразява $[a,b]$ в себе си, всички членове на редицата
$$x_{n+1} = \varphi(x_n)$$
принадлежат на $[a,b]$. За грешката спрямо неподвижната точка имаме
$$|x_n - \xi| = |\varphi(x_{n-1}) - \varphi(\xi)| \leq q|x_{n-1} - \xi|.$$
Прилагайки неравенството последователно, получаваме
$$|x_n - \xi| \leq q|x_{n-1} - \xi| \leq q^2|x_{n-2} - \xi| \leq \cdots \leq q^n|x_0 - \xi|.$$
Тъй като $x_0, \xi \in [a,b]$, е вярно, че
$$|x_0 - \xi| \leq b - a.$$
Следователно
$$|x_n - \xi| \leq q^n(b-a).$$
Понеже $0 < q < 1$, то $q^n \to 0$ при $n \to \infty$, откъдето
$$x_n \to \xi.$$
□

Получената оценка има и практически смисъл: грешката се намалява най-много q пъти при всяка нова итерация. Колкото по-малка е константата q, толкова по-бързо се очаква да се сближават приближенията.

Лема 34.7. Нека $\varphi : [a,b] \to [a,b]$ е непрекъснато изображение, има производна във всяка точка от (a,b) и е изпълнено

$$|\varphi'(x)| \leq q < 1 \qquad \text{за всяко } x \in (a,b).$$

Тогава φ е свиващо изображение на $[a,b]$.

Доказателство. Нека $x, y \in [a,b]$, $x \neq y$. По теоремата за крайните приращения съществува точка η между x и y, за която

$$\varphi(x) - \varphi(y) = \varphi'(\eta)(x-y).$$

Следователно

$$|\varphi(x) - \varphi(y)| = |\varphi'(\eta)|\,|x-y| \leq q|x-y|.$$

Следователно φ е Липшицово, съответно свиващо изображение с константа q. □

Следствие 34.8. Нека ξ е корен на уравнението

$$x = \varphi(x),$$

а φ има непрекъсната производна в околност U на ξ. Ако

$$|\varphi'(\xi)| < 1,$$

то при достатъчно добро начално приближение x_0 итерационният процес

$$x_{n+1} = \varphi(x_n)$$

е сходящ към ξ със скоростта на геометрична прогресия. По-точно, съществуват $C > 0$ и $q \in (0,1)$, за които

$$|x_n - \xi| \leq Cq^n.$$

Доказателство. От непрекъснатостта на φ' в околност на ξ и от неравенството $|\varphi'(\xi)| < 1$ следва, че съществуват $q < 1$ и $\varepsilon > 0$, за които

$$|\varphi'(t)| \leq q \qquad \text{при } t \in [\xi - \varepsilon, \xi + \varepsilon].$$

За t от този интервал, по теоремата за крайните приращения,

$$|\varphi(t) - \xi| = |\varphi(t) - \varphi(\xi)| \leq q|t - \xi| \leq q\varepsilon < \varepsilon.$$

Следователно

$$\varphi(t) \in [\xi - \varepsilon, \xi + \varepsilon].$$

Значи φ е свиващо изображение на достатъчно малкия интервал

$$[\xi - \varepsilon, \xi + \varepsilon]$$

в себе си. Ако началното приближение x_0 е в този интервал, приложима е теоремата за свиващите изображения и редицата е сходяща към ξ с геометрична скорост. $\square$

34.3 Ред на сходимост

Дефиниция 34.9. Нека $x_n \to \xi$ и $e_n = x_n - \xi$. Казваме, че итерационният процес има *ред на сходимост* $p \geq 1$, ако съществува константа μ с $0 < \mu < \infty$, за която

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^p} = \mu.$$

При $p = 1$ и $0 < \mu < 1$ сходимостта е линейна; грешката намалява асимптотично с постоянен коефициент. При $p = 2$ се говори за квадратична сходимост. Еквивалентна практическа оценка за достатъчно големи n е $|e_{n+1}| \leq C|e_n|^p$.

34.4 Методи на хордите, секущите и Нютон

Ще разглеждаме уравнението

$$f(x) = 0$$

при следните предположения:

$$f \in C^2[a,b],$$

$$f(a)f(b) < 0,$$

$$f'(x)f''(x) \neq 0 \qquad \text{за всяко } x \in [a,b].$$

Първото условие означава, че f е два пъти диференцируема в $[a,b]$. Второто условие гарантира съществуването на корен в интервала. От третото условие следва, че f' не променя знака си, следователно f е строго монотонна и коренът е единствен. Освен това f'' има постоянен знак: ако

$$f''(x) > 0,$$

графиката е изпъкнала, а ако

$$f''(x) < 0,$$

графиката е вдлъбната.

34.4.1 Метод на хордите

При метода на хордите единият край на интервала се фиксира, а другото приближение се получава като пресечна точка с абсцисната ос на хордата, свързваща фиксираната точка от графиката с текущата точка.

Тъй като f'' има постоянен знак и $f(a)$ и $f(b)$ са с различни знаци, точно за един от краищата a или b е изпълнено

$$f(c)f''(c) > 0.$$

Този край c се избира за постоянен край на всички хорди. За x_0 се избира другият край на интервала.

Ако постоянният край е a, формулата на метода е

$$x_{n+1} = x_n - \frac{f(x_n)}{f(x_n) - f(a)}(x_n - a), \qquad n = 0, 1, 2, \ldots.$$

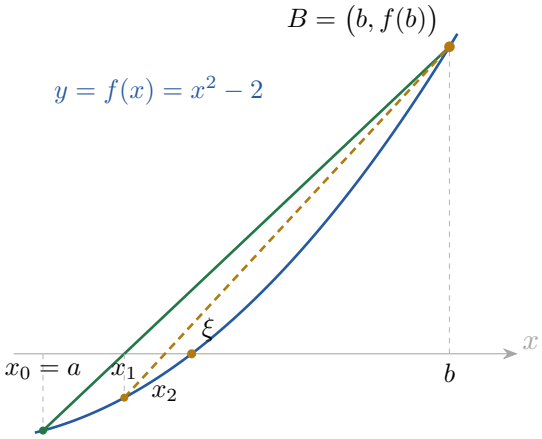
Ако постоянният край е b, съответната формула е

$$x_{n+1} = x_n - \frac{f(x_n)}{f(x_n) - f(b)}(x_n - b), \qquad n = 0, 1, 2, \ldots.$$

Геометрично, за да се построи x_{n+1}, се прекарва хордата през точките

$$(c, f(c)) \qquad \text{и} \qquad (x_n, f(x_n)).$$

Абсцисата на пресечната точка на тази хорда с оста Ox е x_{n+1}. При правилния избор на постоянния край и при достатъчно малък интервал получените точки се движат към единствения корен ξ.



Фигура 34.1: Метод на хордите. Постоянният край е $B = (b, f(b))$, защото $f(b)f''(b) > 0$. Всяка хорда свързва B с текущата точка $(x_n, f(x_n))$; абсцисата на пресичането ѝ с Ox е x_{n+1}.

▷ **Пример 34.10 (Числена илюстрация).** За $f(x) = x^2 - 2$ в $[0{,}5;\ 3]$ имаме $f' > 0$ и $f'' > 0$, а $f(3)f''(3) > 0$, затова постоянният край е $b = 3$, а $x_0 = 0{,}5$. Формулата дава

$$x_1 = 1, \qquad x_2 = 1{,}25, \qquad x_3 \approx 1{,}35294, \qquad x_4 \approx 1{,}39189,$$

при $\xi = \sqrt{2} \approx 1{,}41421$. Приближенията се качват към корена от една и съща страна — характерно за метода на хордите — и грешката намалява с приблизително постоянен множител, тоест геометрично.

Ще докажем геометричната сходимост в случая, когато фиксираният край е b. Тогава методът е породен от функцията

$$\varphi(x) = x - \frac{f(x)}{f(b) - f(x)}(b - x).$$

За $x \in (a, b)$ уравнението

$$x = \varphi(x)$$

е еквивалентно на

$$f(x) = 0.$$

Наистина, ако $x \neq b$, то

$$x = x - \frac{f(x)}{f(b) - f(x)}(b - x)$$

е възможно точно когато $f(x) = 0$.

Твърдение 34.11. При метода на хордите, ако коренът ξ е отделен в достатъчно малък интервал, последователните приближения са сходящи към ξ със скоростта на геометрична прогресия.

Доказателство. Нека ξ е единственият корен в разглеждания интервал. От

$$f(\xi) = 0$$

и от дефиницията на φ следва

$$\varphi(\xi) = \xi.$$

Диференцирайки израза за φ и заместваики $x = \xi$, получаваме

$$\varphi'(\xi) = 1 - f'(\xi)\frac{b - \xi}{f(b)}.$$

Следователно

$$\varphi'(\xi) = \frac{f(b) - f'(\xi)(b - \xi)}{f(b)}.$$

Да приложим формулата на Тейлър за числителя:

$$f(b) = f(\xi) + f'(\xi)(b - \xi) + \frac{f''(\eta_1)}{2}(b - \xi)^2,$$

където η_1 е между ξ и b. Понеже $f(\xi) = 0$, получаваме

$$f(b) - f'(\xi)(b - \xi) = \frac{f''(\eta_1)}{2}(b - \xi)^2.$$

За знаменателя, по теоремата за крайните приращения,

$$f(b) = f(\xi) + f'(\eta_2)(b - \xi) = f'(\eta_2)(b - \xi),$$

където η_2 е между ξ и b. Следователно

$$\varphi'(\xi) = \frac{f''(\eta_1)(b - \xi)}{2f'(\eta_2)}.$$

Нека

$$M = \max_{t \in [a,b]} |f''(t)|, \qquad m = \min_{t \in [a,b]} |f'(t)|.$$

Понеже $f'(t) \neq 0$ в $[a, b]$, имаме $m > 0$. Тогава

$$|\varphi'(\xi)| \leq \frac{M}{2m}|b - \xi|.$$

Когато интервалът, отделящ корена, е достатъчно малък, величината $|b-\xi|$ е достатъчно малка и следователно

$$|\varphi'(\xi)| < 1.$$

От следствието за локалната сходимост на итерационния процес получаваме, че редицата, породена от φ, клони към ξ със скоростта на геометрична прогресия:

$$|x_n - \xi| \le Cq^n, \qquad 0 < q < 1.$$

□

34.4.2 Метод на секущите

Методът на секущите заменя допирателната в текущата точка със секуща през две последователни точки от графиката. Избират се начални приближения x_0 и x_1, като x_0 може да бъде избрано за край на интервала, за който

$$f(x_0)f''(x_0) > 0,$$

а x_1 се взема достатъчно близо до x_0.

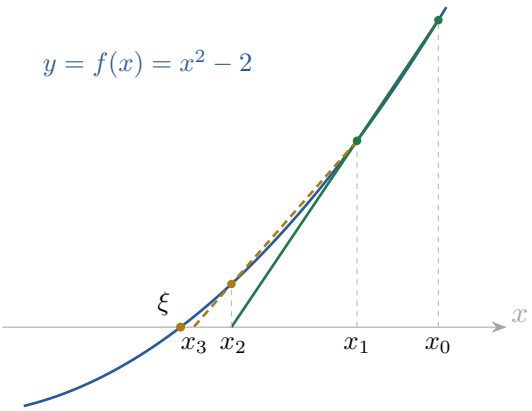
Итерационната формула е

$$x_{n+1} = x_n - \frac{f(x_n)}{f(x_n) - f(x_{n-1})}(x_n - x_{n-1}), \qquad n = 1, 2, \ldots.$$

Геометрично, през точките

$$(x_{n-1}, f(x_{n-1})) \qquad \text{и} \qquad (x_n, f(x_n))$$

се прекарва секуща. Нейната пресечна точка с абсцисната ос има абсциса x_{n+1}. За разлика от метода на хордите тук не се фиксира постоянен край; всяка следваща секуща използва две последователни приближения.



Фигура 34.2: Метод на секущите. Няма постоянен край: всяка секуща минава през последните две приближения $(x_{n-1}, f(x_{n-1}))$ и $(x_n, f(x_n))$, а пресичането ѝ с Ox дава x_{n+1}.

▷ **Пример 34.12 (Числена илюстрация).** Пак за $f(x) = x^2 - 2$, но с $x_0 = 3$ и $x_1 = 2{,}5$:

$$x_2 \approx 1{,}72727, \qquad x_3 \approx 1{,}49462, \qquad x_4 \approx 1{,}42203.$$

> Броят на верните цифри расте по-бързо, отколкото при хордите, но по-бавно, отколкото при Нютон — това е редът $\frac{1+\sqrt{5}}{2} \approx 1{,}618$.

Методът на секущите има ред на сходимост

$$p = \frac{1+\sqrt{5}}{2}.$$

Това е число между 1 и 2, така че методът е по-бърз от геометричната сходимост, но по-бавен от квадратичната сходимост на метода на Нютон.

34.4.3 Метод на Нютон

Методът на Нютон се нарича още метод на допирателните. Избира се начално приближение $x_0 = a$ или $x_0 = b$, така че

$$f(x_0)f''(x_0) > 0.$$

Итерационната формула е

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \qquad n = 0, 1, 2, \ldots.$$

Геометрично, в точката

$$(x_n, f(x_n))$$

към графиката $y = f(x)$ се прекарва допирателна. Абсцисата на пресечната точка на тази допирателна с оста Ox е новото приближение x_{n+1}.

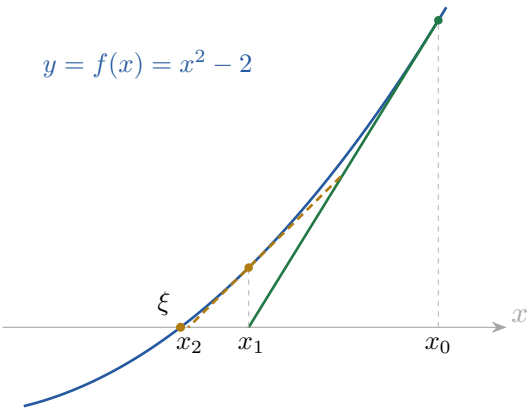
Формулата се получава от уравнението на допирателната към графиката в точката $(x_n, f(x_n))$:

$$y - f(x_n) = f'(x_n)(x - x_n).$$

Като положим $y = 0$, намираме пресечната точка с абсцисната ос:

$$-f(x_n) = f'(x_n)(x_{n+1} - x_n),$$

откъдето следва формулата на Нютон.



Фигура 34.3: Метод на Нютон (на допирателните). В точката $\big(x_n, f(x_n)\big)$ се прекарва допирателна към графиката; абсцисата на пресичането ѝ с Ox е x_{n+1}. Началното x_0 се избира с $f(x_0)f''(x_0) > 0$, за да остават приближенията в интервала.

▷ **Пример 34.13 (Числена илюстрация).** За $f(x) = x^2 - 2$ с $x_0 = 3$ методът на Нютон дава

$$x_1 \approx 1{,}83333, \qquad x_2 \approx 1{,}46212, \qquad x_3 \approx 1{,}41500, \qquad x_4 \approx 1{,}4142138,$$

тоест броят на верните цифри приблизително се удвоява на всяка стъпка — това е квадратичната сходимост $p = 2$.

Методът на Нютон има ред на сходимост

$$p = 2.$$

Следователно при подходящо начално приближение грешката намалява квадратично, което обяснява високата ефективност на метода в близост до корена.

34.5 Сравнение на разгледаните итерационни методи

Методът на хордите използва стойността на функцията в текущата точка и стойността й в един постоянен край на интервала. Той не изисква пресмятане на производна, а сходимостта му при отделен в достатъчно малък интервал корен е със скоростта на геометрична прогресия.

Методът на секущите също не използва производна. Вместо това той използва последните две приближения. Неговият ред на сходимост е

$$\frac{1+\sqrt{5}}{2}.$$

Методът на Нютон изисква пресмятане на $f'(x)$, но има квадратичен ред на сходимост:

$$p = 2.$$

Поради това той е особено ефективен при добро начално приближение.

И при трите метода геометричната конструкция се основава на замяна на графиката на f с по-проста права: хорда с постоянен край, секуща през две приближения или допирателна в текущото приближение. Пресечната точка на съответната права с абсцисната ос дава следващото приближение към корена.

Трите рисунки в §34.4 показват точно това върху един и същ пример: сменя се само правата, с която графиката се заменя. Конспектът иска геометрична илюстрация, формула за последователните приближения и ред на сходимост за всеки от трите метода — рисунката не е украса, а част от изисквания отговор.

34.6 Изпитен фокус

- Да се възпроизведат трите геометрични илюстрации на ръка: графика, текуща точка (и постоянен край при хордите), заменящата права — хорда, секуща или допирателна — и пресичането й с Ox като следващо приближение. Конспектът изисква илюстрацията изрично.

- Да се дефинират неподвижна точка, изображение на $[a,b]$ в себе си, Липшицово изображение и свиващо изображение.
- Да се докаже, че непрекъснато изображение $\varphi : [a,b] \to [a,b]$ има неподвижна точка, като се разгледа функцията
$$r(x) = x - \varphi(x).$$
- Да се обясни свеждането на $f(x) = 0$ до уравнение за неподвижна точка
$$x = \varphi(x),$$
и да се запише итерацията
$$x_{n+1} = \varphi(x_n).$$
- Да се възпроизведе теоремата за свиващото изображение: съществуване и единственост на неподвижната точка, сходимост при всяко $x_0 \in [a,b]$ и оценката
$$|x_n - \xi| \leq (b-a)q^n.$$
- Да се докаже единствеността чрез неравенството
$$|\xi_1 - \xi_2| \leq q|\xi_1 - \xi_2|,$$
както и оценката на грешката чрез многократно прилагане на Липшицовото условие.
- Да се обясни локалният критерий
$$|\varphi'(\xi)| < 1$$
и използването на непрекъснатостта на φ' за намиране на малък интервал, върху който φ е свиващо изображение.
- Да се дефинира ред на сходимост и да се различават геометрична, секуща и квадратична сходимост.
- Да се знаят геометричната идея и формулите на методите:
$$x_{n+1} = x_n - \frac{f(x_n)}{f(x_n) - f(c)}(x_n - c)$$
за хордите с постоянен край c,
$$x_{n+1} = x_n - \frac{f(x_n)}{f(x_n) - f(x_{n-1})}(x_n - x_{n-1})$$
за секущите и
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$
за Нютон.

- Да се знаят редовете на сходимост: геометрична прогресия за хордите,

  $$p = \frac{1+\sqrt{5}}{2}$$

  за секущите и

  $$p = 2$$

  за Нютон.

- Да се възпроизведе доказателствената идея за хордите: методът се представя като $x = \varphi(x)$, пресмята се $\varphi'(\xi)$ и с формулата на Тейлър се показва, че при достатъчно малък интервал

  $$|\varphi'(\xi)| < 1.$$

Въпрос 35

Случайни величини с дискретни разпределения – биномно, геометрично, Поасоново.

По конспекта: какво трябва да мога

Официално заглавие: „Случайни величини с дискретни разпределения — биномно, геометрично и Поасоново разпределение. *Задачи, в които възникват.*“
Дефиниция: дискретно разпределение на случайна величина; свойства на вероятностите — неотрицателност и нормираност — §35.1, стр. 483.
Дефиниция и свойства, без доказателства: пораждаща функция — §35.2, стр. 485.
Пример, пораждаща функция, очакване и дисперсия: биномно разпределение — §35.3, стр. 486.
Пример, пораждаща функция, очакване и дисперсия: геометрично разпределение — §35.4, стр. 489.
Пример, пораждаща функция, очакване и дисперсия: Поасоново разпределение — §35.5, стр. 491.
† **Допълнително:** доказателствата на свойствата на пораждащата функция (§35.2) и сравнението на трите разпределения (§35.6) надхвърлят изискването.
Литература: [7, глави 2.3 (стр. 54–56), 3.2 (стр. 71–74), 6.1 (примери 1–4)], [4, тема „Дискретни разпределения“].

35.1 Дискретни случайни величини и разпределения

Дефиниция 35.1 (Случайна величина). Върху вероятностно пространство $(\Omega, \mathcal{A}, \mathbb{P})$ случайна величина е измерима функция $X : \Omega \to \mathbb{R}$, тоест $\{\omega : X(\omega) \leq x\} \in \mathcal{A}$ за всяко $x \in \mathbb{R}$. Ако $\omega \in \Omega$ е елементарен изход, то стойността $X(\omega)$ зависи от реализирания изход ω.

Дефиниция 35.2. Случайната величина X се нарича *дискретна*, ако множеството от стойностите, които тя приема, е крайно или изброимо.

Нека $H_1, H_2, \ldots$ е разлагане на Ω и нека $x_1, x_2, \ldots$ са различни реални числа. Дискретната случайна величина може да се представи като

$$X(\omega) = \sum_j x_j I_{H_j}(\omega),$$

където

$$I_{H_j}(\omega) = \begin{cases} 1, & \omega \in H_j, \\ 0, & \omega \notin H_j \end{cases}$$

е индикаторът на множеството H_j.

Дефиниция 35.3. Нека X е дискретна случайна величина, която приема стойностите

$$x_1, x_2, \ldots.$$

Дискретно разпределение на X наричаме съответствието

$$\begin{array}{c|ccccc} X & x_1 & x_2 & \cdots & x_j & \cdots \\ \hline \mathbb{P} & p_1 & p_2 & \cdots & p_j & \cdots \end{array},$$

където

$$p_j = \mathbb{P}(X = x_j).$$

Вероятностите в таблицата трябва да удовлетворяват две основни условия:

$$p_j \geq 0 \quad \text{за всяко } j, \qquad \sum_j p_j = 1.$$

Първото условие изразява неотрицателността на вероятността, а второто — нормираността: случайната величина трябва да приеме някоя от възможните си стойности с вероятност 1.

По-общо, вероятността е функция, дефинирана върху σ-алгебра $\mathcal{A}$ от събития. За всяко $A \in \mathcal{A}$ е изпълнено

$$\mathbb{P}(A) \geq 0, \qquad \mathbb{P}(\Omega) = 1.$$

Ако A и B са несъвместими събития, то

$$\mathbb{P}(A \cup B) = \mathbb{P}(A) + \mathbb{P}(B).$$

От свойствата на вероятността следва и монотонността:

$$A \subseteq B \quad \Longrightarrow \quad \mathbb{P}(A) \leq \mathbb{P}(B).$$

Дефиниция 35.4 (Математическо очакване)**.** За дискретна случайна величина с вероятности $p_j = \mathbb{P}(X = x_j)$ математическото очакване е

$$\mathbb{E}X = \sum_j x_j p_j,$$

когато редът $\sum_j |x_j| p_j$ е сходящ.

Дефиниция 35.5 (Дисперсия)**.** Дисперсията се определя чрез

$$\mathbb{D}X = \mathbb{E}\big[(X - \mathbb{E}X)^2\big] = \sum_j (x_j - \mathbb{E}X)^2 p_j.$$

При $\mathbb{E}X^2 < \infty$ разкриването на квадрата и линейността на очакването дават равенството

$$\mathbb{D}X = \mathbb{E}X^2 - (\mathbb{E}X)^2.$$

35.2 Пораждаща функция на дискретна случайна величина

† **Допълнение 35.6 (Обхват: свойствата се дават без доказателство).** Анотацията иска „дефиниция и свойства (*без доказателства*) на пораждаща функция“. Доказателствата в този раздел са допълнителни. Изискването за всяко от трите разпределения е: пример, при който възниква, извеждане на пораждащата функция и пресмятане на математическото очакване и дисперсията — със или без пораждаща функция.

Дефиниция 35.7. Нека X е неотрицателна целочислена дискретна случайна величина, за която

$$\mathbb{P}(X = k) = p_k, \qquad k = 0, 1, 2, \ldots.$$

Пораждаща функция на X се нарича функцията

$$G_X(s) = \mathbb{E}s^X = \sum_{k=0}^{\infty} p_k s^k,$$

за онези стойности на s, за които редът е сходящ.

Коефициентът пред s^k в реда за $G_X(s)$ е именно вероятността

$$\mathbb{P}(X = k).$$

Следователно пораждащата функция кодира цялото дискретно разпределение.

Твърдение 35.8. Нека X е неотрицателна целочислена случайна величина. За пораждащата ѝ функция са в сила следните свойства, като производните при 1 са леви граници и за формулата за дисперсията се изисква $\mathbb{E}X^2 < \infty$:

$$G_X(1) = 1,$$

$$G'_X(1) = \mathbb{E}X,$$

а при съществуваща втора производна

$$\mathbb{D}X = G''_X(1) + G'_X(1) - \big(G'_X(1)\big)^2.$$

Доказателство. Действително, от нормираността на разпределението непосредствено се получава

$$G_X(1) = \sum_{k=0}^{\infty} p_k = 1.$$

Първата производна има вида

$$G'_X(s) = \sum_{k=1}^{\infty} k p_k s^{k-1},$$

За $0 \leq s < 1$ почленното диференциране е допустимо вътре в радиуса на сходимост. При $s \uparrow 1$ монотонната сходимост на неотрицателните членове дава граничната производна, откъдето следва формулата за математическото очакване. Аналогично,

$$G''_X(1) = \sum_{k=0}^{\infty} k(k-1) p_k.$$

Тъй като

$$k^2 = k(k-1) + k,$$

получаваме

$$\mathbb{E}X^2 = G''_X(1) + G'_X(1),$$

а след това и формулата за дисперсията. □

Пораждащите функции са удобни, защото чрез едно и също пресмятане дават както вероятностите, така и основните моменти на разпределението.

35.3 Биномно разпределение

35.3.1 Схема на Бернули

Разглеждаме опит с два възможни изхода: успех и неуспех. Нека вероятността за успех е p, а вероятността за неуспех е

$$q = 1 - p.$$

Случайна величина Y, която приема стойност 1 при успех и стойност 0 при неуспех, има разпределение на Бернули. За нея

$$\mathbb{P}(Y=1)=p, \qquad \mathbb{P}(Y=0)=q.$$

Следователно

$$\mathbb{E}Y=p, \qquad \mathbb{D}Y=pq.$$

Биномното разпределение възниква при фиксиран брой независими опити на Бернули, когато вероятността за успех е една и съща във всеки опит.

Дефиниция 35.9. Нека са извършени n независими опита на Бернули с вероятност за успех p. Ако X е броят на успехите в тези n опита, казваме, че X има *биномно разпределение* с параметри n и p и пишем

$$X \sim \mathrm{Bin}(n,p).$$

Случайната величина X приема стойностите

$$0,1,\ldots,n.$$

За да се получат точно k успеха, трябва да се изберат k от n-те позиции, в които успехът да настъпи. За всяка такава подредба вероятността е

$$p^k q^{n-k}.$$

Броят на възможните избори е $\binom{n}{k}$, затова

$$\mathbb{P}(X=k)=\binom{n}{k}p^k q^{n-k}, \qquad k=0,1,\ldots,n.$$

Твърдение 35.10. Вероятностите на биномното разпределение са нормировани:

$$\sum_{k=0}^{n}\binom{n}{k}p^k q^{n-k}=1.$$

Доказателство. По биномната формула на Нютон

$$\sum_{k=0}^{n}\binom{n}{k}p^k q^{n-k}=(p+q)^n.$$

Тъй като $p+q=1$, получаваме

$$(p+q)^n=1.$$

□

▷ **Пример 35.11.** Нека една монета има вероятност p да се падне ези и се хвърля n пъти независимо. Ако X е броят на падналите се ези, то

$$X \sim \mathrm{Bin}(n,p).$$

При правилна монета $p=\frac{1}{2}$.

▷ **Пример 35.12.** Правилен зар се хвърля 5 пъти. Нека X е броят на получените шестици. Всеки опит има успех с вероятност

$$p = \frac{1}{6},$$

следователно

$$X \sim \text{Bin}\left(5, \frac{1}{6}\right).$$

Например вероятността да се получат точно две шестици е

$$\mathbb{P}(X = 2) = \binom{5}{2}\left(\frac{1}{6}\right)^2\left(\frac{5}{6}\right)^3.$$

35.3.2 Пораждаща функция, математическо очакване и дисперсия

За $X \sim \text{Bin}(n, p)$ пораждащата функция е

$$G_X(s) = \sum_{k=0}^{n} \binom{n}{k} p^k q^{n-k} s^k.$$

Като обединим p^k и s^k, получаваме

$$G_X(s) = \sum_{k=0}^{n} \binom{n}{k} (ps)^k q^{n-k}.$$

Отново по биномната формула на Нютон:

$$G_X(s) = (q + ps)^n.$$

Първата производна е

$$G'_X(s) = np(q + ps)^{n-1}.$$

Следователно

$$\mathbb{E}X = G'_X(1) = np(p + q)^{n-1} = np.$$

Втората производна е

$$G''_X(s) = n(n-1)p^2(q + ps)^{n-2}.$$

При $s = 1$:

$$G''_X(1) = n(n-1)p^2.$$

Затова

$$\begin{aligned}\mathbb{D}X &= G''_X(1) + G'_X(1) - \left(G'_X(1)\right)^2 \\ &= n(n-1)p^2 + np - n^2p^2 \\ &= np(1-p) = npq.\end{aligned}$$

★ **Теорема 35.13.** Ако

$$X \sim \text{Bin}(n, p),$$

то

$$\mathbb{E}X = np, \qquad \mathbb{D}X = npq = np(1-p).$$

Доказателство. Същите формули следват и от представянето

$$X = X_1 + \cdots + X_n,$$

където X_i са независими случайни величини на Бернули с параметър p. Тогава

$$\mathbb{E}X = \mathbb{E}X_1 + \cdots + \mathbb{E}X_n = np,$$

а по независимостта

$$\mathbb{D}X = \mathbb{D}X_1 + \cdots + \mathbb{D}X_n = npq.$$

□

35.4 Геометрично разпределение

35.4.1 Дефиниция и интерпретация

Геометричното разпределение описва повтаряне на независими опити на Бернули до настъпването на първия успех. Броят на опитите не е предварително ограничен.

Дефиниция 35.14. Нека се извършват независими опити на Бернули с вероятност за успех p, където

$$0 < p < 1.$$

Нека X е броят на неуспехите преди първия успех. Тогава X има *геометрично разпределение* с параметър p, означавано с

$$X \sim \text{Geo}(p),$$

ако

$$\mathbb{P}(X = k) = q^k p, \qquad k = 0, 1, 2, \ldots,$$

където $q = 1 - p$.

Събитието $\{X = k\}$ означава, че първите k опита са неуспешни, а следващият опит е успешен. Поради независимостта вероятността му е

$$q^k p.$$

Твърдение 35.15. Вероятностите на геометричното разпределение са нормировани:

$$\sum_{k=0}^{\infty} q^k p = 1.$$

Доказателство. Тъй като $0 < q < 1$, редът е геометрична прогресия:

$$\sum_{k=0}^{\infty} q^k p = p \sum_{k=0}^{\infty} q^k = p \frac{1}{1-q}.$$

Понеже $1 - q = p$, следва

$$p \frac{1}{1-q} = \frac{p}{p} = 1.$$

□

▷ **Пример 35.16.** Хвърляме правилен зар, докато се падне шестица. Ако X е броят на неуспешните хвърляния преди първата шестица, то

$$X \sim \text{Geo}\left(\frac{1}{6}\right).$$

Например събитието $X = 4$ означава, че са се получили четири нешестици, последвани от шестица, и има вероятност

$$\mathbb{P}(X = 4) = \left(\frac{5}{6}\right)^4 \frac{1}{6}.$$

† **Допълнение 35.17 (Бележка).** В някои формулировки случайна величина се дефинира като броя на *опитите* до първия успех. Тогава тя приема стойности $1, 2, \ldots$ и се различава с единица от използваната тук величина. В настоящата глава следваме дефиницията, при която X брои неуспехите преди първия успех; затова стойностите са $0, 1, 2, \ldots$ и

$$\mathbb{P}(X = k) = q^k p.$$

35.4.2 Пораждаща функция, математическо очакване и дисперсия

За $X \sim \text{Geo}(p)$ имаме

$$\begin{aligned} G_X(s) &= \sum_{k=0}^{\infty} \mathbb{P}(X = k) s^k \\ &= \sum_{k=0}^{\infty} q^k p s^k \\ &= p \sum_{k=0}^{\infty} (qs)^k. \end{aligned}$$

Като използваме сумата на геометрична прогресия, получаваме

$$G_X(s) = \frac{p}{1 - qs}.$$

Първата производна е

$$G'_X(s) = \frac{pq}{(1 - qs)^2}.$$

Следователно

$$\mathbb{E}X = G'_X(1) = \frac{pq}{(1-q)^2} = \frac{pq}{p^2} = \frac{q}{p}.$$

Втората производна е

$$G''_X(s) = \frac{2pq^2}{(1-qs)^3}.$$

Следователно

$$G''_X(1) = \frac{2pq^2}{(1-q)^3} = \frac{2q^2}{p^2}.$$

Прилагаме формулата за дисперсията:

$$\begin{aligned}\mathbb{D}X &= G''_X(1) + G'_X(1) - \left(G'_X(1)\right)^2 \\ &= \frac{2q^2}{p^2} + \frac{q}{p} - \frac{q^2}{p^2} \\ &= \frac{q^2+pq}{p^2} \\ &= \frac{q(p+q)}{p^2} \\ &= \frac{q}{p^2}.\end{aligned}$$

★ **Теорема 35.18.** Ако

$$X \sim \text{Geo}(p),$$

където X е броят неуспехи преди първия успех, то

$$\mathbb{E}X = \frac{q}{p}, \qquad \mathbb{D}X = \frac{q}{p^2}.$$

Доказателство. За $0 < p \leq 1$ и $q = 1-p$ пораждащата функция е $G(s) = p/(1-qs)$. Получаваме $G'(1) = pq/(1-q)^2 = q/p$ и $G''(1) = 2pq^2/(1-q)^3 = 2q^2/p^2$. По формулата за дисперсията

$$\mathbb{D}X = \frac{2q^2}{p^2} + \frac{q}{p} - \frac{q^2}{p^2} = \frac{q(q+p)}{p^2} = \frac{q}{p^2}.$$

При $p = 1$ величината е тъждествено нула и формулите също са валидни. □

35.5 Поасоново разпределение

35.5.1 Дефиниция и задачи, в които възниква

Поасоновото разпределение се използва за броя на редки събития при голям брой възможности за настъпването им. То възниква като граничен случай на биномното разпределение, когато броят на опитите е голям, вероятността за успех във всеки опит е малка, а средният брой успехи остава краен.

Дефиниция 35.19. Казваме, че случайната величина X има *Поасоново разпределение* с параметър $\lambda > 0$, ако приема стойностите

$$0, 1, 2, \ldots$$

с вероятности

$$\mathbb{P}(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}, \qquad k = 0, 1, 2, \ldots.$$

Означаваме

$$X \sim \text{Po}(\lambda).$$

Параметърът λ е средният брой настъпвания на разглежданото събитие.

▷ **Пример 35.20.** Дисплей има $1\,000\,000$ пиксела. Вероятността един пиксел да изгори е

$$p = \frac{1}{1\,000\,000}.$$

При $n = 1\,000\,000$ възможности средният брой изгорели пиксели е

$$\lambda = np = 1.$$

Броят X на изгорелите пиксели може да се моделира с Поасоново разпределение с параметър $\lambda = 1$. Например

$$\mathbb{P}(X = 2) = \frac{1^2 e^{-1}}{2!} = \frac{1}{2e}.$$

Твърдение 35.21. Вероятностите на Поасоновото разпределение са нормировани:

$$\sum_{k=0}^{\infty} \frac{e^{-\lambda}\lambda^k}{k!} = 1.$$

Доказателство. Използваме реда за експоненциалната функция:

$$e^{\lambda} = \sum_{k=0}^{\infty} \frac{\lambda^k}{k!}.$$

Тогава

$$\sum_{k=0}^{\infty} \frac{e^{-\lambda}\lambda^k}{k!} = e^{-\lambda} \sum_{k=0}^{\infty} \frac{\lambda^k}{k!} = e^{-\lambda} e^{\lambda} = 1.$$

□

35.5.2 Връзка с биномното разпределение

Нека

$$X \sim \text{Bin}(n, p).$$

Когато n е голямо, p е малко и

$$np \longrightarrow \lambda,$$

биномната вероятност

$$\mathbb{P}(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

се стреми към

$$\frac{e^{-\lambda}\lambda^k}{k!}, \qquad k = 0, 1, 2, \ldots.$$

Това обяснява защо Поасоновото разпределение е подходящ модел за редки успехи сред много на брой опити.

35.5.3 Пораждаща функция, математическо очакване и дисперсия

Нека

$$X \sim \text{Po}(\lambda).$$

Тогава

$$\begin{aligned} G_X(s) &= \sum_{k=0}^{\infty} \mathbb{P}(X = k) s^k \\ &= \sum_{k=0}^{\infty} \frac{e^{-\lambda}\lambda^k}{k!} s^k \\ &= e^{-\lambda} \sum_{k=0}^{\infty} \frac{(\lambda s)^k}{k!}. \end{aligned}$$

От реда за експоненциалната функция следва

$$G_X(s) = e^{-\lambda} e^{\lambda s} = e^{\lambda(s-1)}.$$

Първата производна е

$$G'_X(s) = \lambda e^{\lambda(s-1)}.$$

Следователно

$$\mathbb{E}X = G'_X(1) = \lambda.$$

Втората производна е

$$G''_X(s) = \lambda^2 e^{\lambda(s-1)}.$$

Следователно

$$G''_X(1) = \lambda^2.$$

Тогава

$$\begin{aligned} \mathbb{D}X &= G''_X(1) + G'_X(1) - \left(G'_X(1)\right)^2 \\ &= \lambda^2 + \lambda - \lambda^2 \\ &= \lambda. \end{aligned}$$

★ **Теорема 35.22.** Ако

$$X \sim \text{Po}(\lambda),$$

то

$$\mathbb{E}X = \lambda, \qquad \mathbb{D}X = \lambda.$$

Доказателство. Пораждащата функция е $G(s) = e^{\lambda(s-1)}$. Следователно $G'(1) = \lambda$ и $G''(1) = \lambda^2$. От доказаните формули за моментите следва $\mathbb{E}X = \lambda$ и $\mathbb{D}X = \lambda^2 + \lambda - \lambda^2 = \lambda$. □

Равенството

$$\mathbb{E}X = \mathbb{D}X = \lambda$$

е характерна особеност на Поасоновото разпределение.

35.6 Сравнение на разглежданите разпределения

Трите разпределения описват различни въпроси, свързани с опити с успех и неуспех.

При биномното разпределение броят опити е фиксиран, а се броят успехите:

$$X \sim \text{Bin}(n, p), \qquad \mathbb{P}(X = k) = \binom{n}{k} p^k q^{n-k}.$$

То е подходящо за въпроса: „Колко успеха ще има в точно n независими опита?“

При геометричното разпределение опитите продължават до първия успех, а се броят неуспехите преди него:

$$X \sim \text{Geo}(p), \qquad \mathbb{P}(X = k) = q^k p.$$

То е подходящо за въпроса: „Колко неуспеха ще има преди първия успех?“

При Поасоновото разпределение се брои броят на редки събития:

$$X \sim \text{Po}(\lambda), \qquad \mathbb{P}(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}.$$

То е подходящо, когато има много възможности за настъпване на събитие, всяка с малка вероятност, и средният брой настъпвания е λ.

Основните формули са:

Разпределение	Параметри	$\mathbb{E}X$	$\mathbb{D}X$
$\text{Bin}(n,p)$	$n,\ p,\ q = 1-p$	np	npq
$\text{Geo}(p)$	$p,\ q = 1-p$	$\frac{q}{p}$	$\frac{q}{p^2}$
$\text{Po}(\lambda)$	$\lambda > 0$	λ	λ

Съответните пораждащи функции са:

$$G_X(s) = \begin{cases} (q+ps)^n, & X \sim \text{Bin}(n,p), \\ \dfrac{p}{1-qs}, & X \sim \text{Geo}(p), \\ e^{\lambda(s-1)}, & X \sim \text{Po}(\lambda). \end{cases}$$

35.7 Изпитен фокус

- Да се дефинира дискретна случайна величина и да се представи нейното разпределение чрез стойности x_j и вероятности $p_j = \mathbb{P}(X = x_j)$.

- Да се посочат условията
$$p_j \geq 0, \qquad \sum_j p_j = 1,$$
като израз на неотрицателността и нормираността на вероятностите.

- Да се дефинира пораждащата функция
$$G_X(s) = \sum_{k\geq 0} \mathbb{P}(X = k)s^k$$
и да се използват формулите
$$G_X(1) = 1, \qquad \mathbb{E}X = G'_X(1), \qquad \mathbb{D}X = G''_X(1) + G'_X(1) - \big(G'_X(1)\big)^2.$$

- Да се дефинира биномното разпределение като брой успехи в n независими опита на Бернули:
$$\mathbb{P}(X = k) = \binom{n}{k} p^k q^{n-k}.$$
Да се изведе
$$G_X(s) = (q + ps)^n, \qquad \mathbb{E}X = np, \qquad \mathbb{D}X = npq.$$

- Да се обясни геометричното разпределение като брой неуспехи преди първия успех:
$$\mathbb{P}(X = k) = q^k p.$$
Да се изведе
$$G_X(s) = \frac{p}{1 - qs}, \qquad \mathbb{E}X = \frac{q}{p}, \qquad \mathbb{D}X = \frac{q}{p^2}.$$

- Да се дефинира Поасоновото разпределение:
$$\mathbb{P}(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}.$$
Да се изведе
$$G_X(s) = e^{\lambda(s-1)}, \qquad \mathbb{E}X = \lambda, \qquad \mathbb{D}X = \lambda.$$

- Да се разпознават типичните задачи: фиксиран брой опити за биномното разпределение, чакане до първи успех за геометричното разпределение и редки събития при много възможности за Поасоновото разпределение.

- Да се възпроизведат идеите за проверка на нормираността: биномната формула на Нютон при биномното разпределение, сумата на геометрична прогресия при геометричното разпределение и редът за e^x при Поасоновото разпределение.

- Свойствата на пораждащата функция се *формулират*; конспектът не иска доказателства за тях. Доказателствата в §35.2 са допълнителни.

Литература по конспекта

[1] Андреев, А. и др., Сборник от задачи по числени методи, Университетско издателство „Св. Климент Охридски“, София, 1994.

[2] Боянов, Б., Лекции по числени методи, Дарба, София, 1998.

[3] Боянов, Л., К. Боянов и др., Компютърни мрежи и телекомуникации, изд. „Авангард Прима“, София, 2014.

[4] Вындев, Д., Записки по теория на вероятностите, Електронно издание: https://intranet.fmi.uni-sofia.bg/index.php/s/ji3f3bMSFFZwc5f.

[5] Горслайн, Дж., Фамилия Intel 8086/8088, Техника, София, 1990.

[6] Джаков, П., Р. Леви, Р. Малеев, С. Троянски, Диференциално и интегрално смятане, ФМИ-СУ, София, 2004.

[7] Димитров, Б., К. Янев, Вероятности и статистика, Университетско издателство „Св. Климент Охридски”, София, 1998.

[8] Дойчинов, Д., Математически анализ, Университетско издателство „Св. Климент Охридски“, София, 1994.

[9] Ескенази А., Манева Н., Софтуерни технологии, 2006.

[10] Любенова, Е., П. Недевски, К. Николов, Л. Николова, В. Попов, Ръководство по Математически анализ, София, 1998.

[11] Майерс, С., По-ефективен C++. 35 нови начина да подобрите своите програми и проекти, ЗеСТ Прес.

[12] Манев, К., Увод в дискретната математика. Издателство „КЛМН – Красимир Манев“, София, пето редактирано издание, 2012, ISBN 954-535-136-5.

[13] Метакидес, Д., А. Нероуд, Принципи на логиката и логическото програмиране, Виртех, София, 2000.

[14] Нишева, М., П. Павлов, Функционално програмиране на езика Scheme, София, 2004.

[15] Седжуик, Р., Алгоритми на C, ч.1-4: Основи, структури от данни, сортиране, търсене, СофтПрес.

[16] Сендов, Бл., В. Попов, Числени методи, I ч., Университетско издателство „Св. Климент Охридски“, София, 1996.

[17] Сидеров, Пл., Чакърян, К., Записки по алгебра: групи, пръстени, полиноми, Веди, София, 2014.

[18] Сидеров, Пл., Чакърян, К., Записки по алгебра: линейна алгебра, Веди, София, 2014.

[19] Скордев, Д., Логическо програмиране (записки). Електронно издание: http://www.fmi.uni-sofia.bg/fmi/logic/skordev/ln/lp/new/sydyrzha.htm.

[20] Станилов, Гр., Аналитична геометрия, Софтех, София, 1998.

[21] Стивенс, У. UNIX: взаимодействие процессов. СПб.: Питер, 2003.

[22] Тодорова, М., Езици за функционално и логическо програмиране, I ч.: Функционално програмиране. Сиела, София, 2003.

[23] Тодорова, М., Обектно-ориентирано програмиране на базата на езика C++, Сиела софт енд паблишинг АД, 2008.

[24] Тодорова, М., Програмиране на C++, I и II част. Ciela, София, 2002.

[25] Тодорова, М., Структури от данни и програмиране на C++, Сиела Норма АД, 2011.

[26] Тодорова, М., Армянов, П., Николов, К., Сборник от задачи по програмиране на C++: част втора, Обектно-ориентирано програмиране, ТехноЛогика, 2008.

[27] Тодорова, М., Армянов, П., Петкова, Д., Николов, К., Сборник от задачи по програмиране на C++: част първа, Увод в програмирането, ТехноЛогика, 2008.

[28] Уирт, Н., Алгоритми + структури от данни = програми, BG soft group, София.

[29] Abelson, H., & Sussman, G. J. Structure and interpretation of computer programs. The MIT Press, 1996.

[30] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Merson P., Nord R., Stafford J., Documenting Software Architectures: Views and Beyond, Addison-Wesley Professional, 2010, ISBN: 0-321-55268-7.

[31] Cormen, T., Ch. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, Third Edition, MIT Press, 2009.

[32] Garcia-Molina, H., J. Ullman, J. Widom, Database Systems: The Complete Book, Prentice Hall, 2002.

[33] Grimaldi, R., Discrete and Combinatorial Mathematics: An Applied Introduction, Pearson, 5 edition, 2003, ISBN-13: 978-0201726343.

[34] Hopcroft, John E., and Jeffrey D. Ullman. 1979. Introduction to Automata Theory, Languages, and Computation. First. Addison-Wesley.

[35] Larry L. Peterson and Bruce S. Davie, Computer Networks: A Systems Approach Fifth Edition, © 2012 Elsevier, Inc.

[36] Martin, J., Introduction to Languages and the Theory of Computation, McGraw-Hill, 4 edition, 2010, ISBN-13: 978-0073191461.

[37] Nilsson, U., J. Maluszynski, Logic, Programming and Prolog (2nd ed.). John Willey & Sons, 1995. Електронно издание: http://www.ida.liu.se/~ulfni/lpp/.

[38] Papadimitriou, Christos, and Harry Lewis. 1998. Elements of the Theory of Computation. Prentice-Hall.

[39] Pressman R., Maxim B., Software Engineering: A Practitioner's Approach, McGraw-Hill, 8/e, 2015, ISBN: 0078022126.

[40] Rosen, K., Discrete Mathematics and Its Applications, McGraw-Hill Education, 7 edition, 2012, ISBN 9780073383095.

[41] Russell, S., P. Norvig. Artificial Intelligence: A Modern Approach (4th ed.). Pearson, 2020.

[42] Silberschatz, Galvin, and Gagne, Operating System Concepts, 7th Edition

[43] Sipser, Michael. 2012. Introduction to the Theory of Computation. 3rd ed. Cengage Learning.

[44] Sommerville I., Software Engineering, Addison-Wesley 10th ed., 2015.

[45] Stallings, W., Computer Organization and Architecture. Design for Performance, Prentice Hall, 2000. http://www.williamstallings.com/COA5e.html.

[46] Stevens W.R. Advanced Programming in the UNIX Environment, Addison-Wesley, Reading, Mass, 1992.

[47] Stroustrup, B., C++ Programming Lanquage. Third Edition, Addison-Wesley, 1997.

[48] Tanenbaum, A., Modern Operating systems, 2nd ed., Prentice Hall, 2002.

[49] Tanenbaum, A., Bos, H., Modern Operating Systems, 4th ed. Pearson, 2014, (глави 10-12).

[50] Tanenbaum, A., Structured Computer Organization. Prentice Hall, 2002.

[51] Tannenbaum Andrew S., Wetherall David J., Computer Networks, 5th ed., Prentice Hall, 2011, http://libgen.org/book/index.php?md5=1990789686fa1463f09d2fb230d4301c.

[52] Taylor R., Medvidovic N., Dashofy E Software Architecture: Foundations, Theory, and Practice, John Wiley & Sons, 2009, ISBN 0470167742.

[53] Thompson, S. Haskell: The Craft of Functional Programming (2nd ed.). Addison-Wesley, 1999.