



Лекции по Алгоритми

работна версия, 2020 година

Минко Марков

Copyright © 2014 – 2020 Минко Марков (Minko Markov)
All rights reserved.

Maple is a trademark of Waterloo Maple Inc.

Съдържание

1 Въведение в алгоритмите.	1
1.1 Алгоритъм	1
1.1.1 Опит за дефиниция	1
1.1.2 Изчислителни задачи	4
1.1.3 Алгоритъм и изчислителна задача	5
1.1.4 Елементарни инструкции	6
1.1.5 Knuth за алгоритмите	9
1.2 Евклидовият алгоритъм	12
1.2.1 Оригиналният Евклидов алгоритъм	13
1.2.2 Съвременни варианти на Евклидовия алгоритъм.	15
1.3 Как описваме алгоритми	15
1.3.1 Псевдокод	15
1.3.2 Други видове описание на алгоритми	16
2 Анализ на алгоритми.	19
2.1 Анализ на коректността	20
2.1.1 Доказателство за финитност	20
2.1.2 Доказателство за коректност по индукция	23
2.1.3 Доказателство за коректност с инвариант	24
2.2 Анализ на сложността	30
2.2.1 Големина на входа	30
2.2.2 Въведение в сложността по време	31
2.2.3 Определение на сложността по време	33
2.2.4 Анализ на сложността по памет	37
2.2.5 Проблеми със сложността по време, до които води нашия модел	37
2.3 Асимптотични нотации	38
2.3.1 Опит за намиране на точната сложност по време на SELECTION SORT и INSERTION SORT	38
2.3.2 Отказ от точна оценка и търсене на приблизителна оценка на сложността	40
2.3.3 Асимптотична нотация Тета	42
2.3.4 Други асимптотични нотации: О-голямо, Омега-голямо, о-малко, омега-малко	46
2.3.5 Релации \asymp , \leq , $<$, \geq и $>$	50
2.3.6 Релациите \leq и \geq като преднаредби	60
2.3.7 Релациите на асимптотични сравнения при $n \in \mathbb{N}^+$	65
2.3.8 Асимптотични еквивалентности на често срещани функции	67
2.3.9 Асимптотичните нотации и релациите на асимптотични сравнения	75
2.3.10 Асимптотични сравнения на функции на повече от една променлива	76
2.3.11 Релация \sim : друга релация на асимптотична еквивалентност	76

2.4	Винаги ли е лоша високата сложност	77
3	Сортиране. Елементарни сортиращи алгоритми.	81
3.1	Обща дефиниция	81
3.2	Зашо сортирането е важно	82
3.2.1	Двоично търсене	82
3.2.2	Най-близки елементи	83
3.2.3	Уникалност на елементите	84
3.2.4	Мода	85
3.2.5	Медиана	89
3.3	Анализ на сортиращи алгоритми. Стабилност	90
3.4	Елементарни Сортиращи Алгоритми	92
3.4.1	INSERTION SORT	92
3.4.2	SELECTION SORT	100
4	Двоична Пирамида. HEAPSORT. Приоритетна Опашка.	103
4.1	Двоични дървета и пирамиди	103
4.1.1	Попълнено двоично дърво	103
4.1.2	Адреси в попълнено двоично дърво	109
4.1.3	Двоична пирамида	114
4.1.4	Реализиране на пирамида чрез масив	115
4.1.5	Подпирамида	118
4.2	Построяване на пирамида	120
4.2.1	Наивно построяване на пирамида	120
4.2.2	Бързо построяване на пирамида: алгоритъм BUILD HEAP	122
4.3	Сортиращ алгоритъм HEAPSORT	132
4.4	Приоритетни опашки	133
4.4.1	Абстрактен Тип Данни (АТД)	133
4.4.2	Приоритетна опашка: вид АТД	134
4.4.3	Реализация на приоритетни опашки с двоични пирамиди	135
4.4.4	Функцията INCREASE-KEY	136
5	Рекурсивни Алгоритми, Разделяй-и-Владей и Рекурентни Уравнения.	139
5.1	Рекурсивни алгоритми и Разделяй-и-Владей	139
5.2	Рекурентни уравнения	141
5.2.1	Определение на рекурентно уравнение	141
5.2.2	Обобщения на рекурентно уравнение	143
5.2.3	Рекурентни уравнения и рекурсивни алгоритми	144
5.2.4	Класифициране на рекурентните уравнения	145
5.2.5	Решаване на рекурентни уравнения	147
5.2.6	Рекурентни уравнения в изследването на алгоритми	148
5.3	Методи за решаване на рекурентни уравнения	150
5.3.1	Чрез разазване	150
5.3.2	Чрез дървото на рекурсията	153
5.3.3	По индукция	156
5.3.4	Чрез Мастьър Теоремата (МТ)	161
5.3.5	Чрез Теорема 20	168
5.3.6	Чрез метода с характеристичното уравнение	169
5.4	Примери за ефикасни Разделяй-и-Владей алгоритми	174

5.4.1	Най-близки елементи, 2D	174
5.4.2	Избор на елемент по големина	188
6	Сортиращи алгоритми MERGESORT и QUICKSORT.	194
6.1	MERGESORT	194
6.2	QUICKSORT	201
6.2.1	Имплементация на фазата Разделяй чрез PARTITION–HOARE	202
6.2.2	Имплементация на фазата Разделяй чрез PARTITION–LOMUTO	205
6.2.3	Самият QUICKSORT	209
6.2.4	Сложност по време на QUICKSORT	209
7	Долни граници върху сложност на задачи.	213
7.1	Долни граници на сложността на изчислителни задачи	213
7.2	Дървета за вземане на решения	215
7.2.1	Задачите THE BALANCE PUZZLE и THE TWELVE-COIN PUZZLE	215
7.2.2	Долна граница $\Omega(n \lg n)$ за СОРТИРАНЕ	219
7.2.3	Долна граница $\Omega(n \lg n)$ за УНИКАЛНОСТ НА ЕЛЕМЕНТИ	225
7.2.4	Долна граница $\Omega(\lg n)$ за ТЪРСЕНЕ	229
7.3	Редукции	229
7.3.1	Долна граница $\Omega(n \lg n)$ за НАЙ-БЛИЗКА ДВОЙКА ЕЛЕМЕНТИ	229
7.3.2	Долна граница $\Omega(n \lg n)$ за МОДА	231
7.4	Аргументиране чрез противник (Adversary argument)	232
7.4.1	Неформално обяснение	232
7.4.2	Формално определение	238
7.4.3	Примери за доказване на долни граници чрез противник	239
8	Сортиране в линейно време. COUNTING SORT. RADIX SORT.	252
8.1	Сортиране в линейно време	252
8.2	COUNTING SORT	252
8.3	RADIX SORT	257
9	Въведение в алгоритмите върху графи. Обхождания на графи.	260
9.1	Рекапитулация на графите	260
9.2	Обхождания на графи	264
9.3	Обща схема за обхождане	266
9.4	BFS	267
9.4.1	Псевдокод на BFS от един стартов връх	269
9.4.2	Псевдокод на BFS, обхождащ целия граф	274
9.4.3	Приложения на BFS	274
9.5	DFS	275
9.5.1	Неформално въведение и сравнение с BFS	275
9.5.2	Псевдокод на DFS	277
9.5.3	Свойства на DFS	278
9.5.4	Приложения на DFS	282
10	Топосортиране, силно св. комп., срязващи върхове, мостове	285
10.1	Фундамент	285
10.2	Алгоритъм на Tarjan за топологическо сортиране	288
10.3	Алгоритъм на Kahn за топологическо сортиране	290
10.4	Алгоритъм за намиране на силно свързаните компоненти	293

10.5 Алгоритми за намиране на срязващите върхове и на мостовете	298
10.5.1 Фундамент	298
10.5.2 Алгоритъм за ефикасно намиране на срязващите върхове	301
10.5.3 Алгоритъм за за ефикасно намиране на мостовете	305
11 Минимални покриващи дървета. Алгоритми на Prim и Kruskal.	307
11.1 Фундамент	307
11.2 Алчни алгоритми	313
11.3 Алгоритъм на Prim	315
11.3.1 Базов вариант на алгоритъма на Prim	316
11.3.2 Изтънчен вариант на алгоритъма на Prim	318
11.4 Алгоритъм на Kruskal	321
11.5 Union-Find структури данни	325
11.5.1 Два крайни подхода, които не вършат работа	325
11.5.2 Реализация на разбиване чрез ориентирана гора	326
12 Най-къси пътища. Dijkstra, Bellman-Ford, Floyd-Warshall.	335
12.1 Фундамент	335
12.1.1 Базови дефиниции	335
12.1.2 Пак за отрицателните тегла	342
12.1.3 Релаксация	345
12.2 Алгоритъмът на Dijkstra в два варианта	347
12.3 Най-къси пътища в дагове	349
12.4 Алгоритъмът на Bellman-Ford	351
12.5 Най-къс път за всяка двойка върхове	351
12.5.1 Алгоритъм, реализиращ матрично умножение	352
12.5.2 Алгоритъмът на Floyd-Warshall	357
13 Динамично програмиране.	361
13.1 Прости примери	361
13.1.1 Задачата за опашката пред касата	361
13.1.2 Числата на Fibonacci	363
13.2 Фундамент	364
13.3 Matrix-Chain Multiplication	367
13.4 Minimum Weight Triangulation	378
13.5 Факторизация при неасоциативно умножение	385
13.6 Деривация в контекстно-свободна граматика	386
13.7 Set Partition	388
13.8 2 Equal Sum Subsets	390
13.9 Knapsack	392
13.9.1 Unbounded Knapsack	393
13.9.2 0-1 Knapsack	394
13.9.3 Bounded Knapsack	395
13.10 Interval Scheduling	397
13.10.1 Алчно решение	398
13.10.2 Решение по схемата Динамично Програмиране	400
13.11 Longest Increasing Subsequence	403
13.11.1 Решение по схемата Дин. Прогр. със сложност $\Theta(n^2)$	404
13.11.2 Решение по схемата Дин. Прогр. със сложност $\Theta(n \lg n)$	405

13.11.3 PATIENCESORT	405
14 Алгоритмична неподатливост. NP-пълнота и NP-трудност.	406
15 SAT и теоремата на Cook.	407
16 Основни NP-пълни задачи.	408
17 Заобикаляне на неподатливостта: апроксимиране и параметризиране.	409
Благодарности	410
Библиография	411

Списък на фигурите

1.1	Dixit algorismi.	2
1.2	Блок-схема на Евклидовия алгоритъм.	17
2.1	Видове сложност по време.	36
2.2	Графиките на n^2 , $3n^2$ и $(2 + \sin n)n^2$	44
2.3	Графиките на n , $n^{1+\sin n}$, 1 и n^2	53
2.4	Графиките на $2^{2^{\lfloor \ln \rfloor}}$ и $2^{2^{\lceil n \rceil}}$	54
2.5	Диаграма на преднаредба.	62
2.6	Фактор-релация на пълна преднаредба.	64
2.7	Преднаредба, която не е пълна.	64
2.8	Фактор-релация на преднаредба, която не е пълна.	65
2.9	Графиките на n и $n^{1+\sin 2\pi n}$	66
2.10	$\ln n!$ е голямата сума на Darboux за $\ln(x)$ за $x \in [0, n]$	68
2.11	$\int_1^n \frac{1}{x} dx$ като площ на район.	72
2.12	$\sum_{i=1}^{n-1} \frac{1}{i}$ като площ на район.	73
2.13	$\sum_{i=2}^n \frac{1}{i}$ като площ на район.	74
3.1	Блок-схема на COMPUTE MODE.	87
4.1	Височините на върховете в попълнено двоично дърво T	108
4.2	Изтриване на листата на попълнено двоично дърво.	109
4.3	Пирамида с 26 върха.	119
4.4	Линеаризацията на пирамида от Фигура 4.3.	119
4.5	Листата на пирамидата от Фигура 4.3.	119
4.6	Подпирамидата $A[3]$	119
4.7	Листата на $A[3]$ в червено.	119
4.8	Пирамидата от Фигура 4.3 с един добавен връх.	121
4.9	Илюстрация на работата на NAIVE BUILD HEAP.	121
4.10	Понякога по-изгодно коренът да бъде разменян с детето с по-малкия ключ.	123
4.11	Ако разменяме връх с корена на детето с по-голям ключ	123
4.12	След 4 размени на връх с детето с по-голям ключ получаваме пирамида.	124
4.13	След 2 размени на връх с детето с по-малък ключ получаваме пирамида.	124
4.14	Може размените на връх с по-малкото дете да доведе до $\Omega(n)$ размени.	124
4.15	Пирамидата, получена от дървото на Фигура 4.14 с $\Omega(n)$ размени.	125
4.16	АТД приоритетна опашка като черна кутия.	135
4.17	Попълнено дърво с 14 върха и техните адреси.	137
5.1	Дървото на рекурсията, отговарящо на $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$	153
5.2	Схема на дървото на рекурсията, отговарящо на $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$	154
5.3	Друга схема на дървото при $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$	155

5.4	Дърво на рекурсията при $T(n) = 2T(n - 1) + 1$	156
5.5	Дърво на рекурсията при $T(n) = T(n - 1) + 2T(n - 2) + 2$	157
5.6	МТ: дърво на рекурсията при $T(n) = aT\left(\frac{n}{b}\right) + f(n)$	162
5.7	Точките, които ще ползваме за пример в НАЙ-БЛИЗКА ДВОЙКА, 2D.	175
5.8	Бисекторът ℓ	179
5.9	Дегенеративни случаи. ℓ си остава бисектор.	179
5.10	2d-лентата.	182
5.11	2d-лентата в примера от Фигура 5.7.	183
5.12	Възможните други краища на къса прекосяваща отсечка са малко.	185
5.13	Прозорецът върху примера от Фигура 5.7.	185
5.14	Прозорецът може да обхваща най-много осем точки.	186
5.15	Диаграма на Hasse на данните след ред 9 на SELECT.	191
5.16	Диаграма на Hasse на данните след ред 11 на SELECT.	192
7.1	Схема за измервания за THE BALANCE PUZZLE.	216
7.2	Схема от измервания за THE TWELVE-COIN PUZZLE.	217
7.3	Половината от дървото на сравненията на INSERTION SORT върху $\langle a_1, a_2, a_3 \rangle$	221
7.4	Дървото на сравненията на INSERTION SORT върху $\langle a_1, a_2, a_3 \rangle$	222
7.5	Аргументация с противник: два алгоритъма.	238
7.6	Диаграма на състоянията и преходите при сравняване на числа.	244
9.1	Обхождане на дърво от BFS и от DFS.	277
9.2	Ориентиран граф със сдвоени ребра и неориентиран граф.	281
9.3	Различен брой ребра назад.	283
10.1	Ориентиран граф и неговият фактор-граф.	294
10.2	SCC-DUMMY при различни наредби на върховете.	294
10.3	Транспонираният граф на графа от Фигура 10.1.	297
10.4	Срязващ връх и срязващо ребро.	299
10.5	Разцепване на връх.	300
10.6	Блокове в графи.	301
11.1	Граф и неговите покриващи дървета.	309
11.2	Прехвърляне на връх от границата в дървото при Prim.	317
11.3	Два варианта за Union.	327
11.4	Union by rank: правилен и неправилен.	329

Списък на таблиците

9.1 Сложностите на действията върху графи при двете представления. 263

Списък на допълненията

1	За произхода на думата “алгоритъм”	2
2	Функциите са повече от алгоритмите!	5
3	Изчислителни модели	6
4	Алгоритмичната нерешимост на СТОП ЗАДАЧАТА	11
5	Понятие за рандомизиран алгоритъм	12
6	За разликата между <i>ефективност</i> и <i>ефикасност</i>	19
7	Когато коректността не е стопроцентова.	20
8	Не-финитност и частични функции	20
9	Коректността на Евклидовия алгоритъм	29
10	Колко са класовете на екв. на входа при сортиране	34
11	Друга интерпретация на появата на Θ вляво	45
12	Нотациите O и o в анализа	49
13	За преднаредбите	60
14	Апроксимация на Stirling	67
15	Коректността на COMPUTE MODE	85
16	Неформално за доказателствата за коректност	93
17	За броя на пирамидите на n върха	137
18	Решение на $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$	163
19	Условието за регулярност в МТ влече Случай 3	165
20	Едно разширение на Теорема 18	167
21	За точното решение на $T(n) = 4T(n - 3) + 1$	172
22	Бързо намиране на броя на инверсиите в масив.	198
23	За избора на pivot в QUICKSORT.	201
24	Долни граници, задачи и изчислителни модели	214
25	Decision trees са изчислителни модели	224
26	Опровергаване на $\Omega(g(n))$ не влече $o(g(n))$	230
27	МОРСКИ ШАХ с дявола	234
28	Разликата между BFS и DFS, илюстрирана с шах	276
29	Произход на “топологическо сортиране”.	287
30	Команда <code>tsort</code>	287
31	За броя на покриващите дървета	308
32	Частична наредба \sqsubseteq върху разбиванията	324
33	Генерирането на всички най-къси пътища е неефикасно.	338
34	За структурата на най-дългите пътища	342
35	Намиране на арбитражи чрез отрицателни цикли	343
36	Колко са булевите вектори без съседни единици	362
37	Не всеки Разделяй-и-Владей води до Дин. Прогр.	365
38	За произхода на “Динамично Програмиране”	366
39	За числата на Catalan	369

40 Броят на триангулациите на изпъкнал n -ъгълник	379
---	-----

Лекция 1

Въведение в алгоритмите.

Резюме: Въвеждаме понятията изчислителна задача и алгоритъм. Дискутираме елементните инструкции, от които са съставени алгоритмите. Даваме като пример Евклидовия алгоритъм: най-старият известен истински алгоритъм. Разглеждаме различните начини да бъдат описвани алгоритми.

1.1 Алгоритъм

1.1.1 Опит за дефиниция

Понятието *алгоритъм* е първично. За него няма общоприета прецизна формална дефиниция, също както няма дефиниция на *множество*. Задълбочена дискусия на тема дефиниция на алгоритъм има в статията на Blass и Gurevich [12]. Според Steve Skiena [63],

Алгоритъм е процедура, която решава дадена *задача*. Алгоритъм е идеята, която стои зад всяка смислена компютърна програма.

За да бъде интересен, един алгоритъм трябва да решава задача, която е обща и добре формулирана. Всяка задача се определя от множеството на своите *примери* и от това, какъв трябва да бъде изходът върху всеки от тези примери.

Често в литературата на тема алгоритми се казва, че алгоритъм е *механична процедура*. Добре известен е следният цитат от статия на Edward Forrest Moore [48]:

The methods given in this paper require no foresight or ingenuity, and hence deserve to be called algorithms. They would be especially suited for use in machine, either a special-purpose or a general-purpose digital computer.

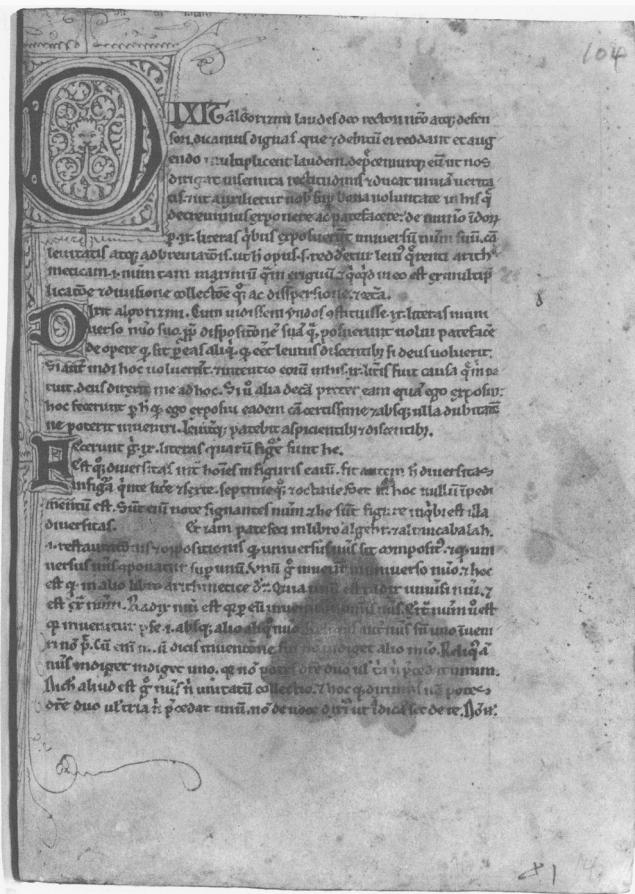
“Механична процедура” е, точно както в цитата от Moore, действия, чието извършване не изисква човешка досетливост и могат да бъдат извършени от машина. Преди ерата на цифровите компютри такива машини са били механични и хората са си представяли (нещо като) часовникови механизми, които ги извършват. Днес вместо “механична процедура” бихме казали “алгоритъм”[†], защото механичната имплементация не е съществена – процедурата може да се имплементира върху електрическа, електронна или квантово-фотонна машина [68]. Същественото е, че процедурата в никой момент не използва човешкия талант да “вижда” решение чрез прозрение, а работата ѝ се реализира от машина, базирана единствено върху законите на физиката.

[†]Естествено, няма да се опитваме да дефинираме “алгоритъм” чрез “алгоритъм”.

Допълнение 1: За произхода на думата “алгоритъм”

Подробно разглеждане на историята ключовия термин *алгоритъм* има в първия том от култовата книга *The Art of Computer Programming* на Donald Knuth [36, стр. 1–2]. Терминът е поевропейчен вариант на името на великия персийски учен Мохамад ибн Муса ал-Хорезми (около 780 г. – около 850 г. сл. Хр.) В англоезичната литература името се пише—когато целта е записът да отразява максимално точно оригиналното произнасяне—Muḥammad ibn Mūsa al-Khwārizmi. Буквално, името означава “Мухамад, син на Муса, Хорезмиецът”. Частицата “ал‐” на арабски е определителен член. Хорезм е древен град в Централна Азия, който днес се намира в Узбекистан и се нарича Хива. “Ал‐Хорезми” означава “който е от Хорезм”, накратко “хорезмиецът”. Името предполага, че Мухамад е роден в Хорезм, макар че това не е сигурно. Това, което днес се смята от историците за сигурна истина е, че ал‐Хорезми живее и работи в дворъ на халифа в Багдад по време на златния период на арабската култура, в периода около 813–830 г. сл. Хр. Ал‐Хорезми е автор на трактат, който се смята за основовоположен за алгебрата, на трактати по география и астрономия, но от алгоритмична гледна точка, важният му принос е трактат върху създадената няколко века по‐рано в Индия десетична позиционна бройна система, използваща нулата като пълноправна цифра наред с другите девет цифри. За съжаление, оригиналът на този текст изглежда загубен във времето. Предполага се, че се е казвал “За изчислението с индийски цифри”. До нас е достигнал превод на латински от 12 век, който по всяка вероятност е правен от друг превод на латински, който е правен от копие на арабски на оригиналния трактат на ал‐Хорезми. Черно‐бяло факсимиile на първата страница от въпросния превод на латински от 12 век (който се съхранява в библиотеката на Кеймбриджкия Университет) е показано на Фигура 1.1. Много подробна информация за този превод на трактата ал‐Хорезми, както и негов превод на английски, има в статията на Crossley и Henry [29], свободно достъпна онлайн на sciencedirect.com.

Фигура 1.1 : Първата страница на превода на латински на трактата на ал-Хорезми за изчислението с цифри. Първите думи са “Dixit algorismi”.



Както се вижда, трактатът започва с думите “Dixit algorismi”. На латински това означава, “Така каза Алгоризми”, където “Алгоризми” е неправилно написаното име на ал-Хорезми. “Dixit algorismi” е и началото на втория параграф.

Оттук в западноевропейските езици се появява думата “algorism”, която означава процедура за работа с числа, написани в индо-арабската бройна система. По онова време, а именно късното средновековие, в Европа все още използват римската не-позиционна бройна система. Предимствата на индо-арабската позиционна бройна система са очевидни за всеки непредубеден човек, но минават няколко века, преди тя да се наложи. По времето, когато двете бройни системи се ползват паралелно, смятащите хора в Европа са разделени на две: абакистите, които ползват римската бройна система и *абака* (вид устройство за смятане), и алгористите, които ползват индо-арабската бройна система. Правилата за работа с индо-арабската система са се називали “алгоризми”. Имало е нескрит антагонизъм между тези две групировки [65], докато индо-арабската система не се е наложила напълно.

Много по-късно думата *algorism* се трансформира в *algorithm* и смисълът ѝ става това, което днес разбираме под “алгоритъм”.

1.1.2 Изчислителни задачи

Разликата между обща задача и конкретна задача е донякъде въпрос на мнение и интерпретация, защото всяка задача може да бъде обобщавана и обобщавана[†], но с два примера ще покажем общоприетото разбиране за разликата между конкретна и обща задача.

- Да бъдат сумирани две конкретни естествени числа, например 7 и 5, е конкретна задача с решение 12. Да бъдат сумирани две произволни естествени числа k и m е по-обща задача. Да бъдат сумирани n на брой естествени числа е още по-обща задача.
- Да бъдат сортирани[‡] две естествени числа, например 7 и 5, е конкретна задача с решение векторът $(5, 7)$. Да бъдат сортирани две естествени числа a_1 и a_2 е по-обща задача, но не достатъчно обща. Нейното решение е просто $(\min\{a_1, a_2\}, \max\{a_1, a_2\})$. Да бъдат сортирани n на брой естествени числа a_1, \dots, a_n е достатъчно обща задача.

Примерите на задачата за сортирането са всички n -елементни вектори от естествени числа за $n \in \mathbb{N}^+$. Очевидно става дума за безкрайно много примери, но **изброямо** безкрайно. При практически всички алгоритми, които ще разглеждаме, множеството от примерите на задачата е (изброямо) безкрайно. На всеки пример съответства един единствен обект, който е *решението на задачата за този пример*. За илюстрация да разгледаме сортирането. Ако примерът е $(5, 12, 3, 1)$, решението за него е $(1, 3, 5, 12)$.

Формално, **всяка изчислителна задача е функция**, чийто домейн е множеството от примери, а кодомейнът е множеството от решенията. Иначе казано, от формална гледна точка, изчислителната задача е множество от наредени двойки от вида

(пример, решение)

Въвеждаме понятието *общ пример на изчислителна задача*, като общият пример е описание, от което получаваме представа за всеки конкретен пример. Аналогично, *общо решение на изчислителна задача* е описание, от което получаваме представа за решението за всеки конкретен пример. За краткост, вместо “общ пример” ще казваме просто “пример”, и вместо “общо решение” ще казваме просто “решение”[§]. Следователно, всяка изчислителна задача се определя напълно от примера и решението. Ето как описваме изчислителните задачи.

Изч. Задача: Числено Сортиране

пример: вектор $A = (a_1, a_2, \dots, a_n)$ от естествени числа

решение: перmutация $(a'_1, a'_2, \dots, a'_n)$ на числата от A , такава че $a'_1 \leq a'_2 \leq \dots \leq a'_n$

Особен интерес представлява частният случай, в който изчислителната задача има отговор Да или НЕ. Тоест, множеството от решенията е {ДА, НЕ}. На английски такива задачи се наричат *decision problems*. На български, професор Манев въвежда термина *задача за разпознаване* за задача, чийто отговор е Да или НЕ. Във формалното описание на задачи за разпознаване, вместо “решение” ще казваме “въпрос”. Ето пример: задачата за най-къс път в графи във вариант-задача за разпознаване. Общият пример е наредена четворка от граф, два върха и число, но за простота не използваме нотация за наредена n -орка.

[†]Всяка задача има обобщение, което е алгоритмично нерешима задача.

[‡]Говорейки за сортиране, винаги имаме предвид сортиране във възходящ ред.

[§]На английски, “изчислителна задача” е *computational problem*, а “примерите” са *the instances*. “Общ пример” е *generic instance*, а “решение” е *solution*.

Изч. Задача: НАЙ-КЪС ПЪТ В ГРАФИ, РАЗПОЗНАВАНЕ

пример: Ориентиран тегловен граф $G = (V, E, w)$; връх $s \in V$; връх $t \in V$; число $k \in \mathbb{N}$.

въпрос: Има ли в $G = (V, E, w)$ ориентиран път с начало s , край t и тегло $\leq k$?

На пръв поглед, този вариант на задачата за най-къс път е безполезен на практика. Но, както ще видим в някои следващи лекции, задачите за разпознаване имат голямо значение в теорията на изчислителната сложност.

1.1.3 Алгоритъм и изчислителна задача

Изчислителна задача е нещо съвсем различно от алгоритъм. Изчислителната задача е *функцията*, която трябва да бъде реализирана. Съществуват много алгоритми, които решават дадена изчислителна задача. Всеки от тези алгоритми е *реализация* на въпросната функция.

Допълнение 2: Функциите са повече от алгоритмите!

Заслужава да се отбележи, че въпросните функции са повече от алгоритмите в теоретико-множествения смисъл на по-голяма безкрайна кардиналност. Алгоритмите, както ще видим, са безкрайно много, но *изброимо* безкрайно. Функциите, от друга страна, са *неизброимо* безкрайно.

Сега ще покажем с диагоналния метод, че дори само множеството от функции от вида $f : \mathbb{N} \rightarrow \{0, 1\}$ е неизброимо безкрайно. Да допуснем, че функциите от този вид може да бъдат изброени като f_0, f_1, f_2, \dots .

Да разгледаме редицата $f_0(0), f_1(1), f_2(2), \dots$. Това е никаква функция с домейн \mathbb{N} и кодомейн $\{0, 1\}$. Да дефинираме нотацията $\overline{f_k(k)}$ по следния начин: за всяко $k \in \mathbb{N}$:

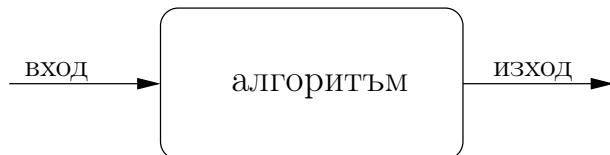
$$\overline{f_k(k)} = \begin{cases} 1, & \text{ако } f_k(k) = 0 \\ 0, & \text{в противен случай.} \end{cases}$$

Сега да разгледаме редицата $\overline{f_0(0)}, \overline{f_1(1)}, \overline{f_2(2)}, \dots$, която също е редица от нули и единици, което означава, функция с домейн \mathbb{N} и кодомейн $\{0, 1\}$. Но функцията $\overline{f_0(0)}, \overline{f_1(1)}, \overline{f_2(2)}, \dots$ се *различава*, за **всяко** $j \in \mathbb{N}$, от f_j върху поне една стойност, а именно j . Следователно, $\overline{f_0(0)}, \overline{f_1(1)}, \overline{f_2(2)}, \dots$ не е нито една от функциите в наредбата. Щом функциите от вида $f : \mathbb{N} \rightarrow \{0, 1\}$ са неизброимо безкрайно много, то и функциите от вида $f : \mathbb{N} \rightarrow \mathbb{N}$ също са неизброимо безкрайно много. Следва, че възможните функции са повече от алгоритмите в същия смисъл, в който реалните числа са повече от естествените. Това означава, че само безкрайно малко подмножество от тези функции имат алгоритми!

Алгоритъмът е преобразувател на информация, той има *вход* и *изход*, като на входа “влиза” някой от примерите, а на изхода “излиза” съответното решение[†] на изчислителната задача. Използвайки термините “вход” наместо “пример” и “изход” наместо “решение”, подчертаваме, че гледаме на алгоритъма като на процес, който преобразува информация, а не като на

[†]При всяко конкретно пускане на алгоритъма, входът е някой конкретен пример, а изходът, някое конкретно решение. Когато разглеждаме алгоритъма по принцип, входът е общият пример, а изходът, общото решение.

статично съответствие между примери и решения. Тази фигура илюстрира интуитивното ни разбиране за алгоритъм:



От ключово значение е това, че се интересуваме не просто от дефиницията на функцията, изобразяваща входа в изхода, а от **реализацията** на функцията. Тоест, интересуваме от механизма, който е **вътре в кутията**. Повече за въпросната реализация ще кажем в следващата подсекция. Тук само ще повторим следното; то не може да се нарече “определение”, а е просто разяснение.

Разяснение 1: Задачи и алгоритми

При дадена изчислителна задача, която е функция, изобразяваща примерите в решението, *алгоритъм за тази задача* е всяка реализация на тази функция на някакво приемливо ниво на детайлност.

Допълнение 3: Изчислителни модели

Подробностите по реализацията на функцията-изчислителна задача са предмет на *изчислителния модел* (на английски, *computational model* или *model of computation*), който сме приели. В тези лекционни записи няма да разглеждаме изчислителни модели експлицитно. Изчислителният модел, който ние възприемаме, е близък до машина с произволен достъп (на английски е RAM, идва от *Random Access Machine*, да не се бърка с *Random Access Memory!*). За повече информация за изчислителните модели вижте [55]. RAM моделът там се въвежда на страница 19.

1.1.4 Елементарни инструкции

И така, алгоритъм е реализация на някаква функция. Тази реализацията се състои от крайна последователност от елементарни инструкции. **Това** е съдържанието на кутията, за която вече стана дума – последователност от елементарни инструкции. Какви инструкции са допустими? Няма общоприета схема за това, какви са елементарните инструкции, но е очевидно, че ако допуснем съвсем произволни инструкции, ще обезсмислим конструирането на алгоритми. Например, ако допуснем елементарна инструкция, която сортира n числа за произволно n , въпростът за конструиране на сортиращ алгоритъм е “решен” – алгоритъмът се състои от тази единствена инструкция. Такова решение би било напълно безполезно на практика. Полезните, смислените елементарни инструкции са тези, които могат да бъдат реализирани **физически** така, че да се изпълняват за време, близко до единица (един машинен такт), върху реален компютърен процесор. Например,

събери числата, намиращи се в променливите a и b , и запиши резултата в a

и

сравни променливите a и b и ако a е по-малко или равно на b , премини към изпълнение на инструкция номер 45, в противен случай премини към изпълнение на инструкция 99

са съмислени елементарни инструкции. С много примери нататък ще илюстрираме ясно какви инструкции и какво ниво на детайлация имаме предвид. Засега ще кажем, че инструкциите се делят на императивни, условни, инструкции за извиквания на други алгоритми и входно-изходни.

Смятаме, че информацията на нашия алгоритъм—а именно информацията на входа, информацията на изхода и междинната информация по време на работа на алгоритъма—е структурирана в променливи. Всяко късче информация е част от някаква променлива. Има съставни променливи, например масиви, и атомарни (елементарни) променливи, например целите числа. Това, разбира се, е абстракция по отношение на работата на истински компютър. Всеки истински компютър работи не директно с числа или букви, а с *представяне*, или *кодиране*, на числа (или букви). Тоест, със стрингове над някаква азбука—най-често азбуката е $\{0, 1\}$ —които може да кодират числа. За целите на този курс е много удобно да избегнем разглеждането на кодиранията и да смятаме, че на най-ниското ниво имаме обекти като числа, с които се работи директно. Елементарните типове данни са `boolean`, `integer`, `real`, `char`, `pointer` и `vertex` (на граф). В това разделение има голяма доза условност, например върховете на графа може да се идентифицират с естествени числа, булевите стойности да са 0 и не-нула и т. н.

Приемаме, че елементарните инструкции се изпълняват в една стъпка независимо от големината на операндите. Например, събирането и умножаването на числа стават за една стъпка *независимо от големината на числата*. От практическа гледна точка, разбира се, това е нереалистично допускане по отношение на много големите числа. За всеки реален компютър има числа, които са достатъчно големи, за да не се побират (по-точно, тяхното представяне да не се побира) в една машинна дума. Нещо повече, всеки реален компютър може да работи само с крайно подмножество на безкрайното множество на целите числа; независимо от избраното кодиране на числата, само безкрайно малка част от всички тях са достъпни за него. Но за целите на курса ще приемаме, че всяко число може да се представи върху нашата абстрактна машина и че основните операции върху кои да е две числа стават за единица време. Това допускане ни позволява да се съсредоточим върху основните аспекти на алгоритмите, а и днешните компютри са толкова мощни, че допускането не е нереалистично за типично възникващите на практика данни.

Изпълнението на елементарните инструкции става в дискретно време (стъпки). Както казахме, инструкциите на алгоритъма са подредени и са краен брой. В изложението нататък ще смятаме, че са номерирани. Започвайки от първата инструкция на алгоритъма, във всеки следващ момент (стъпка) се изпълнява точно една инструкция[†]. Коя ще е следващата инструкция след инструкция i се определя така:

- Ако инструкция i е императивна или входно-изходна, но не е инструкция за край и не е последната инструкция, то след нея се изпълнява инструкция $i + 1$.
- Ако инструкция i е условна, например

`if <някакъв израз с булева интерпретация> then`

или е начало на цикъл, например

[†]С други думи, няма паралелизъм.

`while <някакъв израз с булева интерпретация> do`

или

`for <някакъв израз с булева интерпретация> do`

то следващата инструкция, която ще се изпълни, зависи от конкретния алгоритъм и стойностите на променливите в момента.

- Ако инструкция i е безусловна инструкция за преход

`goto k`

то след нея се изпълнява инструкция k (допускаме, че k е валиден номер на инструкция).

- Ако инструкция i е инструкция за викане на друг алгоритъм, да кажем на алгоритъм ALGX, има две възможности. Ще ги наречем условно “прагматичната” и “теоретичната”.

◆ Прагматичната възможност е да мислим, че ALGX присъства като последователност от номерирани инструкции и следващата инструкция, която ще се изпълни, е първата инструкция на ALGX. Това е програмистката гледна точка: ако от някаква функция викаме друга функция, кодът на другата функция трябва да присъства и да е достъпен, и след извикването започваме да изпълняваме няя. В такъв случай, последната изпълнена инструкция на ALGX няма да е последната изпълнена инструкция на алгоритъма, с който започнахме – след последната инструкция на ALGX ще се изпълни инструкция $i + 1$ от викация алгоритъм (а ако няма такава, то това ще е краят на изпълнението).

◆ Теоретичната възможност е точно обратното нещо: мислим, че ALGX не присъства като конкретни инструкции. Извиквайки ALGX, **някак си, няма значение как**, получаваме ефекта, който очакваме от него. Например, ако ALGX е сортиращ алгоритъм, то ефектът от викането на $\text{ALGX}(a_1, \dots, a_n)$ е, че числата се оказват сортирани и толкова. С други думи, гледаме на ALGX като на черна кутия. Това е абстракция и като всяка абстракция позволява да игнорираме детайли, които смятаме за маловажни (конкретиката на ALGX), и да се фокусираме върху това, което смятаме за важно (конкретиката на викация алгоритъм).

При това можем да отчитаме времето за изпълнение на ALGX^{\dagger} , но може да отидем още по-далече в абстракцията и да мислим, че ALGX се изпълнява **мигновено**, тоест за една единица от дискретното време. Ако дори не отчитаме времето за изпълнение на ALGX, казваме, че ALGX е **оракул**. В реалния свят оракули няма, но това също е полезна абстракция.

След всички тези обяснения: ако гледаме на ALGX като на черна кутия, то следващата инструкция след инструкция i от викация алгоритъм ще бъде, разбира се, инструкция $i + 1$ от викация алгоритъм, ако има такава; а ако няма такава, целият алгоритъм ще приключи.

- Ако инструкция i е за край на алгоритъма, след нея не се изпълнява нищо. Има две възможности за това: тази инструкция да е последната в алгоритъма, без да е част

[†]Въпреки че в тази възможност не гледаме на ALGX като на последователност от инструкции, ние можем да знаем някакви долни граници за задачата, която ALGX решава, и да искаме да отчитаме това време като част от общото време за изпълнение на целия алгоритъм

от цикъл, или да бъде специална инструкция за прекратяване на работата, наречена например `exit` или `halt`.

По принцип не е невъзможно алгоритъмът да се състои в изпълнението на първата инструкция, последвано от втората, третата и така нататък до последната, но това би бил един твърде прост и скучен алгоритъм. Тъй като имаме условни инструкции и възможност за преминаване към изпълнение на произволна инструкция, можем да реализираме цикли. Има изискване за финитност: всеки алгоритъм трябва да завърши изпълнението си за краен брой стъпки върху *всеки* възможен вход. Вече обяснихме при какво условия завърши своята работа алгоритъмът. Има два начина алгоритъм да не завърши:

- Може да “зацикли”, което означава да влезе повторно в конфигурация, в която вече е бил. *Конфигурация* е съвкупността от стойностите на всички използвани променливи, тоест състоянието на паметта. Прост пример за фрагмент от алгоритъм, написан на синтаксиса на C, който гарантира зациклияне, е

```
for (;;);
```

- Може, без да влиза никога повторно в конфигурация, в която вече е бил, да мени някои променливи неограничено, например

```
for (i = 0, j = 1; i < 2; j ++);
```

Теоретично говорейки, върху истински компютър вторият сценарий не може да се реализира, тъй като всеки истински компютър има само крайна памет и краен, макар и невъобразимо голям, брой възможни състояния[†], така че ако бъде оставен да работи достатъчно дълго, той неизбежно ще влезе в състояние, в което вече е бил. В посочения пример, паметта, която реализира променливата `j`, рано или късно ще се препълни и `j` пак ще стане единица.

От практическа гледна точка вторият сценарий лесно може да се реализира върху истински компютър, защото, ако времето за повторно влизане в дадена конфигурация е по-голямо от човешки живот или дори живота на звездите във Вселената, то спокойно можем да смятаме, че до повторение на конфигурация няма да се стигне.

1.1.5 Knuth за алгоритмите

Knuth [36, стр. 6] изброява пет ключови свойства, които трябва да има всеки алгоритъм

Финитност. Всеки алгоритъм, за всеки свой вход, трябва да приключи работата си за краен брой стъпки. Процедура, която има всички свойства на алгоритъм, може би с изключение на свойството финитност, се нарича *изчисителен метод*.

Дефинитност. Всяка стъпка на алгоритъма трябва да бъде дефинирана прецизно и недвусмислено. Такава недвусмисленост имат алгоритмите, описани на някакъв конкретен език за програмиране. Ако описваме алгоритъм на естествен език има рисък от двусмислие – различни читатели може да тълкуват по различен начин дадени инструкции.

[†]Тук се твърди имплицитно, че всеки реален компютър е детерминиран краен автомат – броят на състоянията, макар и фантастично голям, е ограничен от две-на-степен-обема-на-цялата-памет, и преходът от едно състояние към друго е напълно детерминиран. Това е така само ако компютърът е изолиран от околния свят; ако компютърът ползва `input` от околния свят, който `input` е случаен, то не може да твърдим с абсолютна сигурност, че е детерминиран краен автомат. `Input` от околния свят може да е нещо, което идва по мрежата, или нещо, въведено от потребителя.

Вход. Всеки алгоритъм има нула или повече *входни данни*, които са зададени преди началото на изпълнението или се задават динамично по време на изпълнението. Тези данни се вземат от някакви определени множества от обекти.

Изход. Всеки алгоритъм има една или повече *изходни данни*, които са в определена релация[†] с входните данни.

Ефективност. Всеки алгоритъм трябва да е ефективен в смисъл, че всяка от неговите елементарни инструкции, наречени от Knuth “operations”, трябва да може принципно да бъде извършена в *крайно време* от човек с лист и молив.

Моделът с човека с лист и молив е именно модел. Това е идеализация, която не отчита ресурсите, необходими за целта: време, количество хартия (памет), количество моливи и така нататък. По отношение на ефективността, игнорираме ресурсите и допускаме, че човекът може да се занимава произволно дълго (но крайно!) време с тази операция без умора, без грешки, без да оstarява и умира, без да му свърши хартията или молива, и се интересуваме дали при тези допускания операцията може да бъде завършена за крайно време.

Не всяка операция е ефективна. Добре известно е, че има алгоритмично нерешими задачи. Всяка от тези задачи може да се използва за конструирането на операция, която не е ефективна. Пример за не-ефективна операция е:

Ако $P(x)$ терминира, то ...

където P е произволна програма, а x е произволен неин вход. Тази операция, или елементарна инструкция в нашата терминология, не е ефективна, защото задачата дали дадена програма терминира върху даден вход е алгоритмично нерешима (вж. Допълнение 4).

Knuth дава друг пример за не-ефективна инструкция, слагайки в условна инструкция булево условие, чийто отговор е същият като на някаква нерешена от никого до момента математическа хипотеза. Ето пример за такава не-ефективна инструкция (примерът на Knuth е друг, но идеята е същата):

Ако съществува четно число, по-голямо от 2, което не е сума на две прости числа,
то ...

Не-ефективността тук идва от това, че досега никой не е доказал или опровергал хипотезата на Goldbach[‡]. В съвременната формулировка, тя звучи по следния начин.

Хипотеза 1: Goldbach

Всяко четно число, по-голямо от 2, е сума на две прости числа.

Докато хипотезата на Goldbach остава недоказана и неопровергана, никаква условна инструкция, чието булево условие зависи от истинността на тази хипотеза, не може да бъде смятана за ефективна.

[†]Става дума за функцията, която бива реализирана от алгоритъма—да си припомним, че всяка функция е релация.

[‡]Повече информация за хипотезата на Goldbach има в [69]. Оригиналното писмо на Goldbach до Euler, изказващо хипотезата, може да бъде видяно на [този сайт](#).

Допълнение 4: Алгоритмичната нерешимост на Стоп задачата

Alan Turing е доказал [67], че няма алгоритъм, който решава следната задача.

Изч. Задача: Стоп задача

пример: Компютърна програма \mathfrak{P} и неин вход \mathcal{I}

въпрос: Дали \mathfrak{P} с вход \mathcal{I} терминира?

Следва опростено доказателство, че СТОП ЗАДАЧАТА е алгоритмично нерешима. За по-задълбочено разглеждане, добър източник е, например, книгата на Sipser [62].

Теорема 1

СТОП ЗАДАЧАТА е алгоритмично нерешима.

Доказателство:

Да допуснем противното. Тогава съществува програма $\mathfrak{Q}(\mathfrak{P}, \mathcal{I})$, чийто вход се състои от компютърна програма \mathfrak{P} и неин вход \mathcal{I}^a , като $\mathfrak{Q}(\mathfrak{P}, \mathcal{I})$ връща TRUE, ако \mathfrak{P} с вход \mathcal{I} терминира, и FALSE в противен случай. Да дефинираме програма $\mathfrak{S}(\mathfrak{P})$ по следния начин: $\mathfrak{S}(\mathfrak{P}) = \mathfrak{Q}(\mathfrak{P}, \mathfrak{P})$. Забележете, че $\mathfrak{S}(\mathfrak{P})$ се състои от един единствен ред:

return $\mathfrak{Q}(\mathfrak{P}, \mathfrak{P})$

Дефинираме още една програма $\mathfrak{T}(\mathfrak{P})$ по следния начин:

*if $\mathfrak{S}(\mathfrak{P})$ then loop forever
else return TRUE*

Да анализираме $\mathfrak{T}(\mathfrak{T})$. Ако $\mathfrak{T}(\mathfrak{T})$ не терминира, трябва да е вярно, че $\mathfrak{S}(\mathfrak{T})$ връща TRUE. Но тогава трябва да е вярно, че $\mathfrak{Q}(\mathfrak{T}, \mathfrak{T})$ връща TRUE. Тогава трябва да е вярно, че \mathfrak{T} терминира с вход \mathfrak{T} . Това е противоречие.

Тогава да допуснем, че $\mathfrak{T}(\mathfrak{T})$ терминира. Тогава тя връща TRUE. Тогава трябва да е вярно, че $\mathfrak{S}(\mathfrak{T})$ връща FALSE. Тогава $\mathfrak{Q}(\mathfrak{T}, \mathfrak{T})$ връща FALSE. Тогава \mathfrak{T} не терминира с вход \mathfrak{T} . Но това също е противоречие. \square

^aВсъщност, входът на \mathfrak{Q} е **кодирането** на \mathfrak{P} и **кодирането** на нейния вход \mathcal{I} .

Базирайки се на казаното от Knuth, има два начина дадена елементарна инструкция да е неподходяща за инструкция на алгоритъм: тя може да е лошо дефинирана или не-ефективна. Примери за лошо дефинирани инструкции са

Ако красотата ще спаси света, то ...

и

Ако Моцарт е по-добър композитор е Бах, то ...

Това са субективни твърдения и мнения, които нямат нищо общо с формалната математическа прецизност на алгоритмичните инструкции. Примери за не-ефективни инструкции вече дадохме: условните инструкции, базирани на булево условие, съдържащо алгоритмично

нерешима задача, или недоказана и неопровергана хипотеза. Подчертаваме, че въпросните два начина са принципно различни.

Има и трети начин една елементарна инструкция да е неподходяща: да е не-ефикасна, което ще рече, допускането, че тя може да бъде изпълнена за единица време, да е прекалено нереалистично. За разликата между ефективно и ефикасно, вижте Допълнение 6. Пример за такава инструкция е

```
if isprime(n) then ...
```

където n е естествено число, а `isprime(n)` е TRUE тогава и само тогава, когато n е просто число. Доколкото е известно в момента, задачата дали дадено число е просто или не, е **достатъчно алгоритмично трудна**, за да не може да твърдим, че се извършва в единица време. Следователно, `isprime(n)` не може да е елементарна инструкция. В контраст на това,

```
if iseven(n) then ...
```

може да бъде елементарна инструкция, защото тестването за четност е алгоритмично три-виална задача и можем да мислим, че се извършва за единица време.

Към петте свойства на алгоритмите, посочени от Knuth, ще добавим още едно:

Детерминираност. За всеки алгоритъм A и за всеки негов вход x , работата на A с вход x е напълно определена от x : изходът зависи единствено от входа и от нищо друго. Нещо повече, броят на изпълнените стъпки, както и тяхната последователност, зависи единствено от входа и от нищо друго. Колкото пъти да пускаме A с вход x , изходът ще е един и същи, както и последователността от номерата на последователно изпълнените инструкции ще е една и съща.

Допълнение 5: Понятие за рандомизиран алгоритъм

Съществуват и *рандомизирани алгоритми*, за които се предполага, че имат достъп до източник на случайни битове. При тях както изходът, така и последователността от изпълнените инструкции, зависят не само от входа, а и от прочетените от алгоритъма случайни битове. Рандомизираните алгоритми очевидно не притежават свойството детерминираност.

В този курс няма да разглеждаме рандомизирани алгоритми, така че всички алгоритми ще са детерминирани.

1.2 Евклидовият алгоритъм

Въпреки че терминът “алгоритъм” идва от името на човек, живял през IX век, алгоритми са били известни и използвани много преди това. Най-старият известен нетривиален алгоритъм е *Евклидовият алгоритъм*. Евклид е един от най-великите умове на античния свят, математик, известен като *бапти на геометрията*. Неговата поредица от книги, наречена *Елементи*, е в основата на курсовете по класическа геометрия и до днес. В седмата книга от *Елементи* [22] е описана процедура за изчисляване на най-големия общ делител на две цели положителни числа. Тази процедура има всички характеристики на алгоритъм според Knuth (вж. Подсекция 1.1.5), поради което я наричаме “алгоритъм”. Въпросният алгоритъм се съдържа в Задача 1 [22, стр. 296] и Задача 2 [22, стр. 298].

1.2.1 Оригиналният Евклидов алгоритъм

Задача 1: Кога две числа са взаимно прости

Дадени са две числа, като по-малкото бива изваждано последователно от по-голямото. Ако числото, което се получава накрая, никога не дели по-голямото^a, и най-накрая остане единица, оригиналните числа са взаимно прости.

^aИма се предвид, в процеса на изваждане.

Аргументация

Нека по-малкото от две неравни числа AB и CD^{\dagger} бива изваждано последователно от по-голямото и нека получаваното число никога не дели това преди него, докато накрая се получи единица; твърдя, че AB и CD са взаимно прости, тоест, само единицата ги дели и двете.

Зашпото, ако AB и CD не са взаимно прости, някакво число ги дели[‡]. Нека ги дели някакво число и нека то да е E ; нека CD , което дели BF , оставя някакво FA , по-малко от себе си.

Нека AF , което дели DG , оставя по-малко от себе си GC , и нека GC , което дели FH , оставя единица HA .

Но тъй като E дели CD , и CD дели BF , то E дели BF .

Но то също така дели цялото BA ; следователно, то дели и остатъка AF .

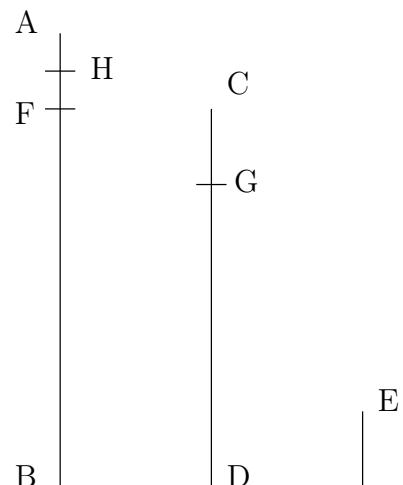
Но AF дели DG ; затова E дели и DG .

Но то също така дели и цялото DC , затова дели и остатъка CG .

Но CG дели FH ; затова E също дели и FH .

Но то дели и цялото FA , затова то още дели и остатъка, а именно единицата AH , въпреки че то[§] е число. А това е невъзможно.

Затова никое число не дели числата AB и CD ; затова те са взаимно прости. \square



[†]Древните гърци са мислели за числата като за дължини на отсечки.

[‡]Древните гърци не са смятали единицата за число, а за нулата дори не са имали понятие.

[§]Става дума отново за Е.

Задача 2: Как се намира най-голям общ делител

Дадени са две числа, които не са взаимно прости, да се намери техният най-голям общ делител.

Аргументация

Нека AB и CD са две дадени числа, които не са взаимно прости.

Иска се да се намери най-големият общ делител на AB , CD .

Ако CD дели AB —тогава то дели и себе си— CD е общ делител на AB и CD .

И е очевидно, че то освен това е най-големият общ делител; защото никое число, по-голямо от CD , не дели CD .

Но ако CD не дели AB , тогава по-малкото от числата AB , CD , ако бъде изваждано последователно от по-голямото, то ще остане число, което дели това преди себе си.

Единица няма да остане; инак, AB , CD ще се окажат взаимно прости (виж Задача 1), което противоречи на хипотезата.

Затова ще остане число, което дели това преди него.

Нека CD , което дели BE , оставя EA , което е по-малко от него самото, което дели DF , оставяйки FC , което е по-малко от него самото[†], и нека CF дели AE .

Тъй като CF дели AE , и AE дели DF , следва, че CF дели DF .

Но то дели освен това себе си, затова дели цялото CD .

Но CD дели BE , следователно CF също така дели BE .

Но то[‡] също така дели EA , затова то също така дели цялото BA .

Но то също така дели CD , затова CF дели AB , CD .

Затова CF е общ делител на AB , CD .

Сега аз казвам, че това число е най-голямото такова.

Зашто, ако CF не е най-големият общ делител на AB , CD , някое число, по-голямо от CF ще дели числата AB , CD .

Нека едно такова число е G .

Понеже G дели CD , тъй като CD дели BE , G също така дели и BE .

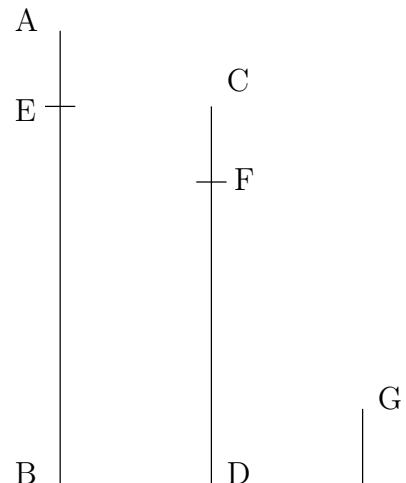
Но то също дели и цялото BA ; затова то дели и остатъка AE .

Но AE дели DF ; затова и G дели DF .

Но то също дели и цялото DC ; затова то дели и остатъка CF , тоест, по-голямото дели по-малкото: което е невъзможно.

Затова никое число, което е по-голямо от CF , не дели числата AB , CD .

Затова CF е най-големият общ делител на AB , CD . □



Този текст ни звучи непривично заради особената нотация и конвенции. Очевидно по Евклидово време не са имали съвременната идея за индукция; той дори не ползва никаква форма на “и така нататък” при аргументацията, а разглежда трикратно прилагане на конструкцията на вземане на (това, което днес бихме нарекли) остатъци при деление и смята, че това е

[†]Вече се има предвид от EA .

[‡] CF

достатъчно убедително. В Задача 1, трикратното прилагане е:

AB и CD дават остатък FA → остатък GC → остатък HA, равен на единица

Въпреки необичайното за нас изразяване, тези две задачи от седмата книга на Евклид съдържат истинско описание на алгоритъм, напълно недвусмислено, с аргументация за привършване на процедурата и за коректност на получения резултат. Анализ на сложността по време, естествено, няма. В десетата книга на Евклид се дискутира същата идея, но върху числа-отсечки, които днес бихме нарекли “реални”; в Евклидовата терминология, това са отсечки, чието отношение на дължините не е рационално число (*incommensurables* в английския превод). Изключително задълбочена дискусия за Евклидовия алгоритъм има във втория том от поредицата на Knuth [37].

1.2.2 Съвременни варианти на Евклидовия алгоритъм.

Съвременната формулировка на Евклидовия алгоритъм ползва не последователни изваждания, а значително по-бързото изчисляване на остатък при деление (което може да се имплементира чрез изваждания-докато-е-възможно, но това би била много тромава от днешна гледна точка имплементация). Две възможни реализации на този алгоритъм са итеративната и рекурсивната, които показваме сега.

EUCLID, ITERATIVE ($a, b \in \mathbb{N}^+, a \geq b$) 1 while $a \bmod b > 0$ do 2 $r \leftarrow a \bmod b$ 3 $a \leftarrow b$ 4 $b \leftarrow r$ 5 return b	EUCLID, REC ($a, b \in \mathbb{N}^+, a \geq b$) 1 if $a \bmod b = 0$ 2 return b 3 else 4 return EUCLID, REC($b, a \bmod b$)
---	--

В Допълнение 9 ще докажем коректността на Евклидовия алгоритъм.

1.3 Как описваме алгоритми

1.3.1 Псевдокод

Реализациите на Евклидовия алгоритъм на тази страница са добра илюстрация за използването на *псевдокод*. Псевдокод е езикът, който ползваме за описание на алгоритми. Това не е истински език за програмиране, а измислен от нас език, който е пределно изчистен от ненужни детайли. С псевдокод не можем да направим синтактична грешка – понеже няма общоприети правила за псевдокод, синтаксисът си го измисляме ние, пишейки псевдокода. Псевдокодът, който ще използваме в тези лекции, е почти същият като псевдокода в учебника на Cormen, Leiserson, Rivest и Stein [15]. Той прилича далечно на синтаксиса на езика за програмиране Pascal. Читател, свикнал със синтаксиса на езиците C или Java лесно ще чете псевдокода, ако има предвид няколко негови особености.

- ползваме стрелка наляво “ \leftarrow ” за присвояване (на английски, *assignment*), а не “ $:=$ ” като на Pascal или “ $=$ ” като на C.
- ползваме знак за равенство “ $=$ ” за сравняване на обекти, също като на Pascal, а не “ $==$ ” като на C.

- Масивите се индексират от едно, а не от нула. С други думи, по подразбиране, началният елемент на масива A е $A[1]$, а не $A[0]$. $A[0]$ може да се използва само ако масивът е дефиниран като, например, $A[0, \dots, n]$.
- Цикъл с постусловие **repeat-until** се “върти”, докато постусловието е лъжа; при първото достигане на постусловието, в което то е TRUE, изпълнението на цикъла се прекратява и управлението минава към следващата инструкция, ако има такава. Така е в Pascal. В C и Java, постуловията са изградени по обратната конвенция: следващо изпълнение има и само ако постусловието е TRUE.
- Не се използват ограничители като “{” и “}” за означаване на блокове от код, било в цикъл, било в условна инструкция. Блоковете код се маркират с различен отстъп от лявата страна. Това е като в езика за програмиране Python. Само ако алгоритъмът е много дълъг и окото на читателя може да се обърка, ще ползваме “{” и “}” за означаване на блокове от код.

Писането на псевдокод има потенциален недостатък. Свободата при писането на псевдокода създава опасност да започнем да пишем елементарни инструкции, които:

- не са ефикасни,
- или не са ефективни,
- или дори не са добре дефинирани.

1.3.2 Други видове описание на алгоритми

Вече видяхме един начин за описание на алгоритми: в псевдокод. Алгоритми може да се описват и на естествен човешки език, например описанията на Евклид в аргументациите на Задача 1 и Задача 2. Но естественият език е подходящ само за описание на прости алгоритми или за описание на високо ниво, без детайлите. Ако трябва да опишем сложен алгоритъм на човешки език, и то в детайли, неизбежно ще започнем да използваме някаква смесица от естествен език и псевдокод. Колкото по-сложно и детайлно е описанietо, толкова по-вероятно е това да бъде неясно[†]. Защо тогава да не ползваме пределно изчистения и недвусмислен псевдокод поначало?

Алгоритми могат да се описват и на истински езици за програмиране като C или Java. В този случай не правим разлика между описание на алгоритъм и неговата програмна реализация – програмната реализация е алгоритъмът. Този начин за описание на алгоритми има съществен недостатък: колкото и да е изчистен езикът за програмиране, програмата, написана на него, съдържа детайли, които са ирелевантни за алгоритъма и пречат да се разбере идеята. Вече видяхме цитатът от Skiena на стр. 1, казващ, че алгоритъмът е **е идеята, която стои зад всяка смислена компютърна програма**. Ако вярваме в това, то има смисъл да правим разлика между описанietо на идеята и на конкретна програма, имплементираща идеята. Разбира се, това не трябва да се абсолютизира. Прост алгоритъм като Евклидовият можеше да бъде описан на чисто C и описанietо щеше да е толкова ясно, колкото е псевдокодът. Но колкото по-изтънчен и труден за възприемане е алгоритъмът, толкова повече се виждат предимствата на описанietо в псевдокод пред описанietо на C. Най-малкото, ако пишем на псевдокод, винаги можем да изберем на какво ниво на детализация да е описанietо и можем да опишем практически всеки алгоритъм съвсем кратко.

[†]Опитайте се да опишете на чист български език алгоритъм, в който има сложни if-then-else условия с многократна вложеност.

И накрая, алгоритми може да се описват чрез *блок-схеми*[†]. Фигура 1.2 показва блок-схема на Евклидовия алгоритъм (алгоритъм EUCLID, ITERATIVE на стр. 15). Блок-схемата е диаграма, изобразяваща ориентиран граф с четири вида върхове:

- връх-начало и връх-край, които се рисуват със заoblени правоъгълници,
- върхове-условия, които се рисуват с ромбове, в които са написани условията,
- върхове-императивни инструкции, в които са написани императивните инструкции,
- може да има и четвърти вид върхове, да ги наречем *ветрила*, които не съответстват на инструкции, а са нещо като точки, в които може да дойде изпълнението от повече от едно места.

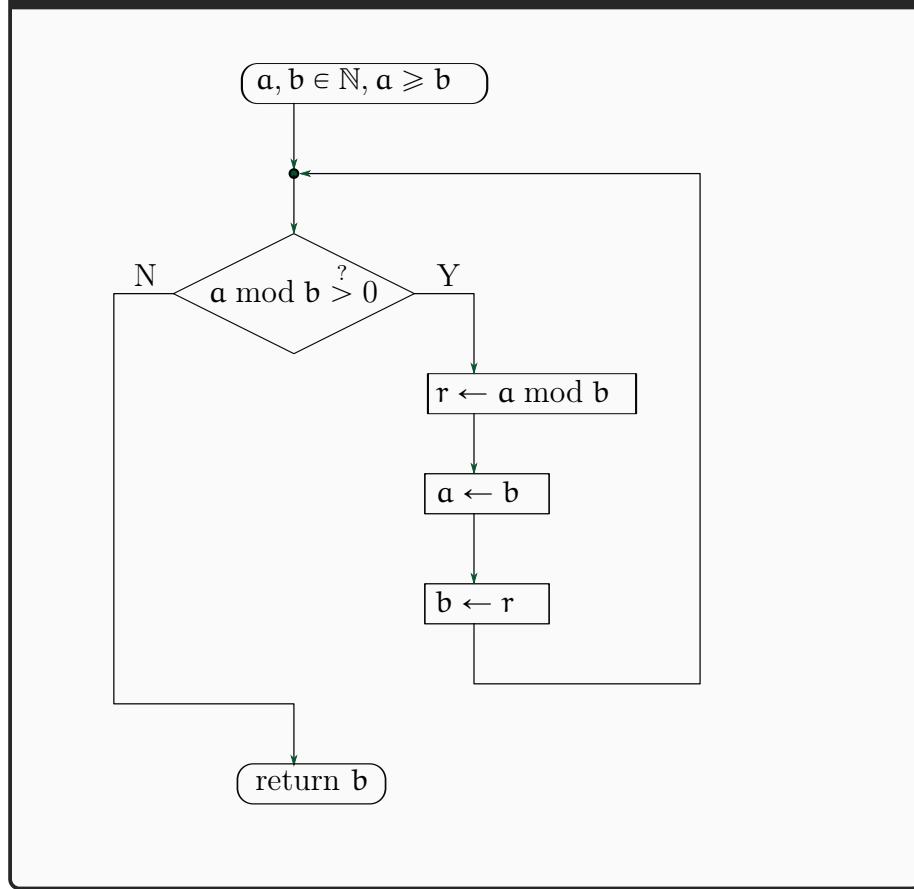
Стрелките показват как върви изпълнението, което по правило се изобразява отгоре надолу. Върхът-начало има степен на входа нула, защото от него започва изпълнението, и степен на изхода единица. Върховете-условия и върховете-императивни инструкции, както и върхът-край, имат степен на входа единица. Върхът-край има степен на изхода нула, върховете-условия има степен на изхода две, а върховете-императивни инструкции имат степен на изхода едно. Ако в някой връх може да се “влезе” по повече от един начина[‡], преди него слагаме връх от тип ветрило, който има повече от един входове и само един изход. Връх-ветрило рисуваме с дебела точка.

Ориентираните ребра показват изпълнението на алгоритъма. Както казахме, върховете-условия имат степен на изхода две, като едното от излизящите ребра е маркирано с Y, а другото, с N, които очевидно идват от Yes и No и показват къде отива изпълнението на алгоритъма, ако условието е съответно истина или лъжа.

[†]На английски “блок-схема” е *flowchart*.

[‡]Например на Фигура 1.2 в условието “ $a \bmod b > 0$ ” може да се влезе по два начина: “отгоре” и “отдолу”.

Фигура 1.2 : Блок-схема на Евклидовия алгоритъм.



Описанието с блок-схеми е удачно за прости алгоритми. На практика много често алгоритми се описват с блок-схеми, ако описание е предназначено за хора, които не са специалисти по алгоритми (съответно е безсмислено да им се дава псевдокод) и освен това алгоритмите са прости, като често дори нямат цикли[†]. Пример за прост алгоритъм без цикли, който трябва да бъде разчетен от човек, който не е специалист по алгоритми, е алгоритъм за това какво да правим, ако автомобилът ни отказва да запали. Ето един [такъв алгоритъм](#). Но за сложни алгоритми, особено ако описание е върху няколко страници, блок-схемите не са подходящи.

[†]Алгоритъмът да няма цикли е същото като графът на блок-схемата да е ацикличен, тоест стрелките да вървят само отгоре надолу.

Лекция 2

Анализ на алгоритми.

Резюме: Разглеждаме различните аспекти на анализа на алгоритмите: анализ на коректността и анализ на сложността, който на свой ред се разбива на анализ по време и анализ по памет. Въвеждаме големина на входа на алгоритъм и видовете сложност като функции от големината на входа. Извеждаме точни изрази за сложността на конкретни прости алгоритми. Показаваме необходимостта от по-малко прецизна, по-груба мярка за сложността на алгоритмите. Въвеждаме петте общоприети асимптотични нотации. Разискваме дали високата сложност е непременно лошо нещо.

Да бъде *анализиран* даден алгоритъм означава да бъде доказано, че той е коректен, и ако е коректен, да бъде изчислено колко ресурси ползва—за всеки възможен вход—като ресурсите са основно време и памет. Това са различни аспекти на анализа. Първият аспект—коректността—е свързан с понятието *ефективност*, а вторият аспект е свързан с понятието *ефикасност*.

Допълнение 6: За разликата между *ефективност* и *ефикасност*

В стандартния български език е прието *ефективност* и *ефикасност* да се смятат за синоними, но на английски има значителна разлика между съответните *effective* и *efficient*. Въпросът дали даден алгоритъм е *effective* има отговор или Да, или НЕ; ако алгоритъмът върши работата, която е определена от заданието (изчислителната задача), то той е **ефективен**, понеже неговата работа дава желания **ефект**; в противен случай той не е ефективен. При ефикасността въпросът не е дали, а до колко. Ефикасността е свързана с ресурсите, които алгоритъмът ползва.

При тази езикова конвенция, “ефективен” е свързан с резултата, а “ефикасен”, с пътя, по който достигаме до резултата. Ако зададем въпроса дали даден алгоритъм е ефикасен и очакваме отговор или Да, или НЕ, то ние сме задали един безсмислен въпрос^a. Обаче има смисъл да твърдим, че един алгоритъм е по-ефикасен от друг алгоритъм по отношение на някакъв ресурс, ако първият ползва по-малко от този ресурс от втория за всички входове; например, ако винаги работи по-бързо. От друга страна, сравняването на ефективността на два алгоритъма е безсмислено, ако и двата са коректни – в такъв случай и двата са ефективни, и това е всичко по отношение на ефективността.

Терминологичната разлика между “ефективен” и “ефикасен” е много полезна на практика и ние ще я спазваме.

^aОсвен, разбира се, ако предварително не сме се разбрали, че “ефикасен” означава нещо количествено определено. В теорията на алгоритмите съществува конвенция, според която “ефикасен” означава “работещ в полиномиално време”. При наличие на такава конвенция, въпросът дали даден алгоритъм

е ефикасен с отговор или Да, или Не, е смислен. При липса на такава конвенция обаче, ефикасността не се описва с Да или Не.

Допълнение 7: Когато коректността не е стопроцентова.

Съществуват сложни изчислителни задачи, за които са приемливи алгоритми, които не винаги работят коректно, но все пак са полезни при условие, че можем да ограничим количествено броя на входовете, върху които алгоритъмът греши. Ако броят на входовете, върху които алгоритъмът не работи коректно, е пренебрежимо малък спрямо броя на всички входове, то този алгоритъм може да е интересен и полезен при условие, че не разполагаме с други алгоритми за тази задача или разполагаме, но те са прекалено бавни. В такъв случай, ефективността не е булева величина Да/Не, а нещо по-сложно, например положително реално число.

В този курс такива задачи и алгоритми няма да разглеждаме, така че ефективността за нас ще е именно булева. Ако някакъв алгоритъм не работи коректно дори за един единствен вход, ще казваме, че не е коректен.

И така, анализирайки един алгоритъм, първо ще доказваме, че е ефективен, тоест коректен, и след това ще изследваме колко е ефикасен, тоест какви ресурси ползва. Когато разглеждаме сложността на **даден алгоритъм за дадена задача**, искаме алгоритъмът да е колко е възможно по-ефикасен (естествено, бивайки ефективен). Когато разглеждаме сложността на **задачата**, фактът, че за нея няма бърз алгоритъм може да бъде от голям теоретичен интерес, но може и да е полезен на практика, както ще стане ясно в Подсекция 2.4.

2.1 Анализ на коректността

Анализът на коректността е по-важният вид анализ в смисъл, че ако се окаже, че даден алгоритъм не е коректен, това е краят на анализа му. Колко бързо работи и колко памет ползва некоректен алгоритъм не ни интересува.

Нека е даден алгоритъм A , който решава някаква изчислителна задача Π . Както казахме вече, можем да мислим за Π като за функция от множеството на конкретните примери в множеството на конкретните решения, а алгоритъмът A е една възможна реализация на тази функция чрез някакви основни елементи-инструкции. Анализът на коректността е доказателство, че наистина A реализира именно **тази** функция, а не някаква друга. Ще ползваме главно две техники, за да доказваме коректност на алгоритми: чрез индукция (вж страница 23), когато става дума за рекурсивни алгоритми, и чрез *инвариант на цикъл* (вж страница 24), когато става дума за итеративни алгоритми.

2.1.1 Доказателство за финитност

Важна част от доказателството за коректност е доказателството за финитност (вж. Подсекция 1.1.5). С други думи, че алгоритъмът завършва работата си върху всеки вход.

Допълнение 8: Не-финитност и частични функции

Както вече казахме много пъти, алгоритъмът е реализация на някаква функция с домейн примерите и кодомейн, решенията, която функция отговаря еднозначно на из-

числителната задача. При липса на финитност, според терминологията на Knuth (вж. Подсекция 1.1.5), казваме “изчислителен метод” вместо “алгоритъм”.

Свойствата финитност и нефинитност може да бъдат описани елегантно в терминологията на функциите: при наличие на финитност говорим за алгоритъм, който реализира **тотална** функция, а при липса на финитност говорим за изчислителен метод, който реализира **частична** функция^a. В контекста на теорията на алгоритмите ние се интересуваме от реализациите на изображението (частична или тотална функция) чрез елементарни инструкции. Има смисъл да кажем, че даден елемент x от домейна се изобразява в елемент от кодомейна ако и само ако реализациите от елементарни инструкции, стартирайки с вход x , терминира; а ако не терминира, то x не се изобразява в кодомейна. Тогава можем да кажем, че алгоритмите реализират тотални функции, докато изчислителните методи реализират по-общите частични функции.

В този контекст, да докажем финитността на алгоритъма е да докажем, че той е реализация на тотална функция. Повече информация за съответствието между терминиращи и не непременно терминиращи процедури, от една страна, и тотални функции и частични функции, от друга страна, има в книгата на Manna [46].

^aДа си припомним, че “частична функция” е обобщение на “тотална функция”, при което може да има елементи от домейна, на които не съответстват елементи-изображения от кодомейна.

Доказателството за финитност често се пренебрегва, защото бива считано за очевидно. В много случаи наистина е очевидно, но не винаги е така. Може да бъде изключително трудно да се отговори дали даден алгоритъм винаги завършва. Както е показано в Допълнение 4, **не съществува** алгоритъм, който по дадено кодиране на **произволен** друг алгоритъм плюс вход да определи дали той (другият) ще завърши работата си върху този вход, или не. Алгоритмичната нерешимост на СТОП ЗАДАЧАТА обръча на провал всеки опит да подходим алгоритмично към анализа на финитността на произволен алгоритъм. Можем да докажем—ако успеем!—финитността на **даден** алгоритъм, но нямаме алгоритмичен начин да правим това за **всеки** алгоритъм.

Нещо повече. Дори за конкретен даден алгоритъм, да анализираме неговата финитност е същото като да решим някаква математическа задача. Верността на математическо твърдение е **кодирана** в това, дали алгоритъмът терминира винаги, или не! Ще разгледаме три примера за това.

Първи пример: хипотезата на Goldbach Следният пример беше предложен на автора от Георги Георгиев. Хипотезата на Goldbach (вж. Хипотеза 1) може да бъде използвана, за да се конструира алгоритъм, чиято финитност зависи директно от това, дали хипотезата е истина.

DUMMY–GOLDBACH(\emptyset)

```

1  n ← 2
2  do
3      n ← n + 2
4      more ← FALSE
5      for i ← 2 to  $\frac{n}{2}$ 
6          if isprime(i) and isprime(n - i)
7              more ← TRUE
8  while more
```

Очевидно DUMMY–GOLDBACH терминира ако и само ако Хипотезата на Goldbach е невярна. Тъй като засега не знаем дали хипотезата е вярна, не можем да кажем и дали този алгоритъм терминира, или не.

Втори пример: голямата теорема на Fermat

Теорема 2: Голяма Теорема на Fermat

Уравнението $a^n + b^n = c^n$ няма решения в цели положителни числа за $n \geq 3$.

Голяма Теорема на Fermat има свой алгоритмичен аналог – да докажем, че следният код, написан на C, не терминира, е същото като да докажем, че тя е вярна:

```
k = 3;
for (;;) {
    for(a = 1; a <= k; a++)
        for(b = 1; b <= k; b++)
            for(c = 1; c <= k; c++)
                for(n = 3; n <= k; n++)
                    if (pow(a,n) + pow(b,n) == pow(c,n))
                        exit();
    k++;
}
```

Голямата теорема на Fermat е доказана—за разлика от Хипотезата на Goldbach—така че ние знаем, че тази процедура не терминира и поради това не е алгоритъм. Но тази теорема е **изключително трудна** за доказване. Доказателството ѝ е било непреодолимо предизвикателство в продължение на стотици години. През 90те години на 20 век най-накрая Andrew Wiles направи доказателство [72], чието прочитане (с разбиране ...) остава разбираемо за сравнително малко хора. За подробности вижте [тази онлайн, свободно достъпна, книга на Nigel Boston](#).

Трети пример: хипотезата на Collatz

Хипотеза 2: Collatz

Нека $C : \mathbb{N}^+ \rightarrow \mathbb{N}^+$ е дефинирана така:

$$C(n) = \begin{cases} \frac{n}{2}, & \text{ако } n \text{ е четно,} \\ 3n + 1, & \text{ако } n \text{ е нечетно.} \end{cases}$$

Нека $C^{(0)}(n) = n$ и $C^{(k+1)}(n) = C(C^k(n))$, за всяко $n \in \mathbb{N}^+$ и всяко $k \in \mathbb{N}$. Тогава,

$$\forall n \exists m : C^m(n) = 1$$

Неформално, хипотезата казва, че от което и цяло положително число n да започнем, неизбежно ще достигнем до 1 в процеса, в който заменяме n със $C(n)$:

$$n \mapsto C(n) \mapsto C(C(n)) \mapsto C(C(C(n))) \mapsto \dots \mapsto 1$$

Например, ако започнем с 14, ще минем през следните стойности:

$$14 \mapsto 7 \mapsto 22 \mapsto 11 \mapsto 34 \mapsto 17 \mapsto 52 \mapsto 26 \mapsto 13 \mapsto 40 \mapsto 20 \mapsto 10 \mapsto 5 \mapsto 1$$

Тъй като $C(1) = 1$, достигнем ли веднъж 1, оставаме в 1^\dagger . Това, което не е доказано е, че от всяко n ще достигнем единицата. Въпреки значителните усилия на много математици в продължение на много години, хипотезата остава именно хипотеза – нито е намерен контрапример, нито има доказателство. За повече информация вижте статията на Belaga и Mignotte [5] и огромната библиография от статии за хипотезата на Collatz [40], [41].

Следният алгоритъм имплементира идеята за финитност, необходимо и достатъчно условие за която е хипотезата на Collatz да е истина.

DUMMY-COLLATZ($n \in \mathbb{N}^+$)

```

1 while  $n \neq 1$  do
2   if  $\text{iseven}(n)$ 
3      $n \leftarrow \frac{n}{2}$ 
4   else
5      $n \leftarrow 3n + 1$ 
```

Очевидно DUMMY-COLLATZ терминира за всяко n ако и само ако Хипотеза 2 е истина.

Наблюдение 2

Доказателството за финитност, което е само един аспект от доказателството за коректност на даден алгоритъм, може да бъде изключително трудно и да изисква дълбоки познания по математика.

Забележка

Алгоритмите, които разглеждаме в този курс, са значително по-прости от посочените в тази подсекция и доказателствата за тяхното терминиране по правило са тривиални. За типичния алгоритъм в курса ние дори не споменаваме това, че той терминира, когато правим анализ на коректността му – няма смисъл да подчертаваме очевидното.

2.1.2 Доказателство за коректност по индукция

Ето прост пример за доказателство за коректност чрез индукция.

НАРАСТВАНЕ С ЕДИНИЦА(n : естествено число)

```

1 if  $n = 0$ 
2   return 1
3 else
4   if  $n \bmod 2 = 0$ 
5     return  $n + 1$ 
6   else
7     return  $2 \times \text{НАРАСТВАНЕ С ЕДИНИЦА}(\lfloor \frac{n}{2} \rfloor)$ 
```

[†]Щом $C(1) = 1$, казваме, че 1 е *фиксирана точка* на функцията $C(n)$. На английски терминът е *fixed point*. Ако хипотезата е вярна, тази функция няма други фиксирани точки.

Теорема 3

Алгоритъмът НАРАСТВАНЕ С ЕДИНИЦА връща $n + 1$ за всеки вход n .

Доказателство:

С индукция по n .

База. $n = 0$. В такъв случай булевият израз на ред 1 е истина и алгоритъмът връща $0 + 1 = 1$ чрез присвояването на ред 2. ✓

Индуктивно Предположение. Нека за всяко $m < n$ алгоритъмът връща $m + 1$.

Индуктивна стъпка. Да разгледаме работата на НАРАСТВАНЕ С ЕДИНИЦА върху вход $n \geq 1$. Първо да допуснем, че n е четно. Тогава булевият израз на ред 4 е истина, следователно ред 5 се изпълнява и алгоритъмът връща $n + 1$. ✓

Сага да допуснем, че n е нечетно. Булевият израз на ред 4 е лъжа, следователно ред 7 се изпълнява и алгоритъмът връща $2 \times$ НАРАСТВАНЕ С ЕДИНИЦА ($\lfloor \frac{n}{2} \rfloor$). Тъй като $\lfloor \frac{n}{2} \rfloor < n$, може да приложим индуктивното предположение. Съгласно него, НАРАСТВАНЕ С ЕДИНИЦА ($\lfloor \frac{n}{2} \rfloor$) връща $\lfloor \frac{n}{2} \rfloor + 1$. Тъй като n е нечетно, $n = 2k + 1$ за някое $k \in \mathbb{N}$, и изходът е

$$2 \times \left(\left\lfloor \frac{2k+1}{2} \right\rfloor + 1 \right) = 2 \times \left(\left\lfloor k + \frac{1}{2} \right\rfloor + 1 \right) = 2 \times (k + 1) = 2k + 2 = n + 1 \quad \checkmark$$

□

2.1.3 Доказателство за коректност с инвариант

Доказателствата с инвариант на цикъла по същество също са доказателства по индукция, но се прилагат при итеративни алгоритми. Инвариантата е *едноместен предикат*, свързан с цикъла; по правило, този предикат е свързан с инструкцията, която съдържа условието за край на цикъла. Този предикат трябва да е:

- верен първия път, когато изпълнението на алгоритъма стигне до въпросния ред, и
- верността му трябва да се запазва по време на всяко изпълнение, и
- в момента на напускане на цикъла (такъв момент настъпва неизбежно, щом алгоритъмът завършва) неговата вярност трябва да влече директно това, което искаме да докажем за алгоритъма.

Очевидно става дума за вид доказателство по индукция, само че върху краен домейн, а именно, множеството $\{1, 2, \dots, m\}$, където m е броят на достиганията на реда, съдържащ условието за край на цикъла. Най-общо, предикатът се формулира така:

Инвариант

При k -тото достигане на ред ℓ на алгоритъм НЯКАКЪВ Алгоритъм е изпълнено «**някакво-твърдение-формулирано-чрез-k**».

Ако предикатът е $P(k)$, то доказателството следва позната схема на доказателства по индукция върху **краен** домейн (който е индуктивно дефинирано множество):

$$P(1) \wedge \forall k_{1 \leq k < m} (P(k) \rightarrow P(k + 1)) \vdash \forall k_{1 \leq k \leq m} (P(k))$$

Съществува възможност предикатът да се дефинира по-просто – директно чрез управляващата променлива на цикъла, ако това е възможно. Нека цикълът е **for**-цикъл с инкрементиране на управляващата променлива с единица:

```
for i ← n1 to n2
    «тяло-на-цикъла»
```

за никакви фиксиранни n_1 и n_2 и освен това i не бива променяна вътре в тялото на цикъла. Тогава може да вземем за домейн на предиката множеството $\{n_1, \dots, n_2 + 1\}$ [†] и да дефинираме предиката директно като $P(i)$, където i е управляващата променлива на цикъла. Но при по-сложни цикли, примерно **while**-цикли, в които управляващата променлива се мени в тялото на цикъла по никакъв сложен начин, по-удачно е предикатът да е върху броя пъти на достигането на реда с условието.

Ще илюстрираме такова доказателство с пример.

$\text{ALGX}(A[1, 2, \dots, n]: \text{array of integers})$

```
1   max ← −∞
2   i ← 1
3   while i ≤ n do
4       if A[i] > max
5           max ← A[i]
6       i ← i + 1
7   return max
```

Теорема 4

ALGX връща стойността на максимален елемент от входния масив $A[1, 2, \dots, n]$.

Доказателство: Доказателството на теоремата се основава на следното помошно твърдение, което ще докажем първо.

Инвариант

Всеки път, когато изпълнението на ALGX е на ред 3, променливата max съдържа стойността на максимален елемент в подмасива $A[1, \dots, i - 1]$.

Доказателството на инвариантата е по индукция по броя на достиганията на ред 3.

База. Да разгледаме първото достигане на ред 3. От една страна, подмасивът $A[1, \dots, i - 1]$ в този момент е празен, защото i съдържа стойност 1 заради присвояването на ред 2, следователно $A[1, \dots, i - 1]$ всъщност е $A[1, \dots, 0]$, а това е празен подмасив. Максималният елемент на празния масив е $-\infty$, защото $-\infty$ е неутралният елемент на операцията максимум. Следователно, математически погледнато, максималният елемент на $A[1, \dots, i - 1]$ е $-\infty$. От друга страна, променливата max съдържа $-\infty$ заради присвояването на ред 2. ✓

[†]Това е множеството от стойностите, които управляващата променлива взема. Забележете, че най-голямата от тях е $n_2 + 1$, а не n_2 , понеже при последното достигане на реда с условието—когато условието вече не е изпълнено и тялото на цикъла няма да се изпълнява повече—управляващата променлива е именно $n_2 + 1$.

Индуктивно предположение. Да допуснем, че твърдението е вярно при някое достигане на ред 3, което не е последното[†]. Да означим текущата стойност на i с i' .

Преди да продължим с доказателството, подчертаваме, че би било грешка да изпъснем уточнението “което не е последното”, защото искаме да се убедим, че инвариантата се запазва при преминаване през тялото на цикъла, а след последното достигане на ред 3 повече преминаване през тялото цикъла **няма**.

Индуктивна стъпка. Следните две възможности са изчерпателни.

- $A[i']$ е по-голям от всеки елемент в $A[1, \dots, i' - 1]$. Съгласно индуктивното предположение, \max съдържа стойността на максимален елемент в $A[1, \dots, i' - 1]$. Тогава $\max < A[i']$. Следователно, условието на ред 4 е TRUE и изпълнението отива на ред 5, където \max получава стойността на $A[i']$. И така, \max съдържа стойността на максимален елемент в $A[1, \dots, i']$. Тогава обаче i се инкрементира с единица (ред 6) и вече съдържа $i' + 1$. Очевидно, при следващото достигане на ред 3, \max съдържа стойността на максимален елемент в подмасива $A[1, \dots, (i' + 1) - 1]$. Написано чрез стойността на променливата i в този момент, последното става $A[1, \dots, i - 1]$. Виждаме, че инвариантата се запазва в този случай.
- Не е вярно, че $A[i']$ е по-голям от всеки елемент в $A[1, \dots, i']$. Тогава $A[i']$ е по-малък или равен на максимален елемент в $A[1, \dots, i' - 1]$, който на свой ред е равен на \max съгласно индуктивното предположение. Заключаваме, че \max съдържа стойността на максимален елемент не само в $A[1, \dots, i' - 1]$, а и в $A[1, \dots, i']$. В този случай, $\max \notin A[i']$. Следователно, условието на ред 4 е FALSE и присвояването на ред 5 не се случва. После i бива инкрементирана с единица (ред 6) и вече съдържа $i' + 1$. Очевидно, при следващото достигане на ред 3, \max съдържа стойността на максимален елемент в подмасива $A[1, \dots, (i' + 1) - 1]$. Написано чрез стойността на променливата i в този момент, последното става $A[1, \dots, i - 1]$. Виждаме, че инвариантата се запазва и в този случай.

С това доказателството на инвариантата приключва.

Да се върнем към доказателството на теоремата.

Следствие от инвариантата. ALGX очевидно терминира, така че някое достигане на ред 3 е последно. Да разгледаме моментът, в който ред 3 се изпълнява за последен път. Тогава i съдържа стойността $n + 1$. Да заместим i с $n + 1$ в инвариантата. Получаваме буквально “ \max съдържа стойността на максимален елемент в подмасива $A[1, \dots, (n + 1) - 1]$ ”. По-просто казано, \max съдържа стойността на максимален елемент в подмасива $A[1, \dots, n]$.

Доказателството е почти готово. Забелязваме, че веднага след това се изпълнява ред 7. ALGX връща \max , която променлива съдържа стойността на максимален елемент на входния масив. Следователно, ALGX връща стойността на максимален елемент на входния масив, което и трябваше да покажем. \square

В следващите доказателства за коректност с инвариантата няма правим следствие от инвариантата, както сторихме току-що, а в доказателството на инвариантата ще добавяме още една

[†]Тъй като алгоритмите трябва да са финитни, ако разглеждаме именно алгоритъм, а не изчислителна процедура, последно достижане не може да няма.

фаза, наречена **Терминация**. Освен това, за краткост ще сливаме фазите **Индуктивно предположение** и **Индуктивна стъпка** в една единствена фаза, наречена **Поддръжка**. Въпросните доказателства имат следната структура (по [15]). Инвариантата е едноместен предикат, да го наречем $P(k)$, който е свързан с този ред от алгоритъма, който съдържа условието, управляващо цикъла. В най-общия случай, променливата k е броят на достиганията на въпросния ред от началото на изпълнението, включително и сегашното достижане. Общата схема на доказателството, както стана ясно от изложението досега, е:

- Показваме, че $P(1)$ е истина при първото достижане на въпросния ред. Това е базата на доказателството.
- Допускайки, че $P(k)$ е в сила за някое достижане, което не е последното, показваме, че $P(k+1)$ е в сила при следващото достижане. Това е поддръжката на доказателството.
- Разглеждаме момента, в който изпълнението е на въпросния ред от алгоритъма за последен път. Нека k' е броят на всички достижания до въпросния ред, включително и последното (текущото). Замествайки k с k' в инвариантата, получаваме точно това твърдение за алгоритъма, което ни трябва. Това е терминациията на доказателството.

Важен частен случай, който вече дискутирахме горе, е следният. Да кажем, че цикълът е **for**-цикъл от вида (ℓ е номерът на съответния ред в алгоритъма):

$\ell \quad \text{for } i \leftarrow a \text{ to } b$

a и b са константи, а променливата i не се променя в тялото на цикъла. Тогава за простота променливата на предиката е управляващата променлива на цикъла, тоест предикатът е $P(i)$. Общата схема на доказателството в този частен случай е:

- Показваме, че $P(a)$ е вярно при първото достижане на ред ℓ .
- Допускайки, че $P(i)$ е вярно при някое достижане на ред ℓ , което не е последното, (което значи, че $i \leq b$), доказваме $P(i + 1)$ при следващото достижане на ред ℓ .
- Разглеждаме момента, в който доказателството е на ред ℓ за последен път. Тогава трябва i да съдържа $b + 1$. Замествайки i с $b + 1$, получаваме получаваме точно това твърдение за алгоритъма, което ни трябва.

Дебело подчертаваме следните две неща.

- Първо, може да има много начини да се мине през тялото на цикъла в зависимост от стойностите както на контролната променлива, така и на останалите променливи. Например, може тялото на цикъла да има сложна вложена **if-else-if** структура. Доказателството трябва да следва всеки възможен път на изпълнение, тоест всяка възможност за преминаване през тази структура.

Ето пример. Нека алгоритъмът има променливи-естествени числа a , b и t и тялото на

цикъла има такава структура:

```

if i mod 5 = 1 then
    if t is a perfect square then
        a ← b + 1
    else if t < 200 then
        a ← b + 2
    else
        a ← b + 3
else if i mod 5 = 2 then
    b ← b2
else
    b ← b – t

```

Пълно доказателство трябва да разглежда всички начини да се “мине” през тези случаи и подслучаи, а това зависи от делимостта на i на 5 и освен това, в случай, че остатъкът е 1, от стойността на t.

Ето друг пример. Ако тялото на цикъла има k на брой последователни **if-else** блока:

```

if «булево условие 1» then
    «някакви императивни инструкции»
else
    «някакви императивни инструкции»
if «булево условие 2» then
    «някакви императивни инструкции»
else
    «някакви императивни инструкции»
...
if «булево условие k» then
    «някакви императивни инструкции»
else
    «някакви императивни инструкции»

```

то от най-общи комбинаторни съображения има 2^k начина, по които изпълнението може да мине през всички тези **if-else**. Доказателството, което разглежда всеки от тези 2^k начина, може да е изключително дълго.

- Въпросният предикат P трябва не просто да е верен, а освен това да ни върши работа за доказателството. Не всеки верен предикат ни върши работа. Като пример, да разгледаме следния алгоритъм.

ALGY($A[1, 2, \dots, n]$): масив от цели числа)

```

1   for i ← 1 to n
2       A[i] ← i

```

Да разгледаме невалидно доказателство, че това е сортиращ алгоритъм (какъвто той не е), което се основава на следната инвариантна.

Инвариантна

Всеки път, когато изпълнението на ALGY е на ред 1, подмассивът $A[1, \dots, i - 1]$ е сортиран.

Тази инвариантна вярна ли е? Очевидно да, и доказването ѝ е тривиално. ALGY сортиращ алгоритъм ли е? Очевидно не, защото той унищожава входните данни; това, че реализира сортиран масив, а именно $[1, 2, \dots, n]$, не е достатъчно, за да е сортиращ. Защо вярната инвариантна не доказва коректността на ALGY? Защото, за да бъде един алгоритъм сортиращ, той трябва не просто да реализира сортиран масив, а трябва да реализира сортиран масив, който **се състои точно от входните елементи**. Още веднъж: инвариантата трябва да е не просто вярна, а и полезна за целта на доказателството.

Наблюдение 3

Не всяка вярна инвариантна е полезна за това, което искаме да докажем.

Допълнение 9: Коректността на Евклидовия алгоритъм

Ето отново итеративната версия на Евклидовия алгоритъм.

EUCLID, ITERATIVE($a, b \in \mathbb{N}^+, a \geq b$)

```

1   while a mod b > 0 do
2       r ← a mod b
3       a ← b
4       b ← r
5   return b
```

Доказателството за коректността ѝ се основава на следния прост факт. Нека $\gcd(a, b)$ е най-големият общ делител на a и b .

Лема 1

Нека $a, b \in \mathbb{N}^+$ и $a \geq b$. Тогава

$$\gcd(a, b) = \begin{cases} b, & \text{ако } b \text{ дели } a \\ \gcd(b, a \bmod b), & \text{в противен случай} \end{cases}$$

□

Доказателството на Лема 1 е **тривиално**.

Инвариантата

Нека a' и b' означават съответно входните a и b в алгоритъм EUCLID, ITERATIVE. При всяко достигане на ред 1, $\text{gcd}(a, b) = \text{gcd}(a', b')$.

База. При първото достигане на ред 1, $a' = a$ и $b' = b$, така че твърдението е очевидно вярно. ✓

Поддръжка. Да допуснем, че твърдението е вярно за някое достигане на ред 1, което не е последното. Да означим с a_1 и b_1 текущите стойности на a и b . Да означим с a_2 и b_2 стойностите на a и b при следващото достигане на ред 1. Допускането е, че $\text{gcd}(a_1, b_1) = \text{gcd}(a', b')$. Щом достигането на реда не е последното, то $a_1 \bmod b_1 > 0$; тоест, b_1 не дели a_1 . Съгласно Лема 1, $\text{gcd}(a_1, b_1) = \text{gcd}(b_1, a_1 \bmod b_1)$. Но очевидно присвояванията на редове 2, 3 и 4 имат следния ефект: $a_2 = b_1$ и $b_2 = a_1 \bmod b_1$. Показахме, че $\text{gcd}(a_1, b_1) = \text{gcd}(a_2, b_2)$. Щом $\text{gcd}(a_1, b_1) = \text{gcd}(a', b')$ (от допускането) и $\text{gcd}(a_1, b_1) = \text{gcd}(a_2, b_2)$, то $\text{gcd}(a', b') = \text{gcd}(a_2, b_2)$. ✓

Терминиране. Да разгледаме последното достигане на ред 1. Да наречем съответно \hat{a} и \hat{b} текущите стойности на a и b . Щом условието на ред 1 е лъжа, очевидно $\hat{a} \bmod \hat{b} = 0$. Тогава, съгласно Лема 1, $\text{gcd}(\hat{a}, \hat{b}) = \hat{b}$. От инвариантата знаем, че $\text{gcd}(a', b') = \text{gcd}(\hat{a}, \hat{b})$. Веднага следва, че $\text{gcd}(a', b') = \hat{b}$.

Но на ред 5 алгоритъмът връща именно \hat{b} . Заключаваме, че алгоритъмът връща $\text{gcd}(a', b')$. □

2.2 Анализ на сложността

“Сложност на алгоритъм” е мярка за това, колко бързо (или бавно ...) работи алгоритъмът. “Сложен” в този смисъл е “бавен”. Има и други видове сложност, които дават представа за качествата на алгоритъма, като основна сред тях е свързана с употребата на памет. Затова ще говорим за сложност по време и сложност по памет (на английски съответно *time complexity* и *space complexity*), които ще определим сега. Само ще споменем, че има и други видове сложност, например при разпределено изчисление (*distributed computation*) се говори за комуникационна сложност (*communication complexity*), измерваща количеството комуникация между независими изчисляващи агенти.

2.2.1 Големина на входа

Големина на входа на алгоритъм, или още размер на входа (на английски *input size*), е ключово понятие за дефинирането на сложността на алгоритмите, защото сложността е функция на размера на входа. Прецизното определение е следното.

Определение 1: Големина на входа – определение, което няма да използваме

За произволен алгоритъм, големина на входа при конкретен вход е дължината на стринга, който кодира този вход в избраната система за кодиране.

Това определение е полезно за теория на сложността, която отчита и кодирането на данните.

Тъй като вече приехме, че няма да разглеждаме кодирания, Определение 1 не върши работа, поне не за целите на този курс. Терминът “големина на входа” ще дефинираме непрецизно, с голяма доза условност. Ако алгоритъмът е върху числен масив, например ако задачата е свързана със сортиране, то големината на входа е броят на числата в масива. Ако алгоритъмът е върху графи, големината на входа е сумата от броя на върховете и броя на ребрата. Това важи както за графи без тегла, така и за тегловни графи. Ако алгоритъмът е върху стрингове, големината на входа е броят на символите. Това разбиране за големина на входа отговаря добре на модела, който вече приехме, в който числата са неделими елементи, а елементарните операции върху тях стават в единица време.

Практиката е показвала, че това разбиране за големина на входа обикновено води до смислени резултати. Както ще видим нататък, има и изключения.

2.2.2 Въведение в сложността по време

Да разгледаме някакъв прост алгоритъм, например сортирация алгоритъм INSERTION SORT. Псевдокодът, който ще разгледаме, е по учебника CLR (Cormen, Leiserson, Rivest) [15].

INSERTION SORT($A[1, 2, \dots, n]$: масив от естествени числа)

```

1  for i ← 2 to n
2      key ← A[i]
3      j ← i - 1
4      while j > 0 and A[j] > key do
5          A[j + 1] ← A[j]
6          j ← j - 1
7      A[j + 1] ← key

```

Колко бързо работи той? От практическа гледна точка, този въпрос е за физическото време, което отнема пускането на алгоритъма, и по-точно на някаква практическа реализация, върху истински компютър. От тази гледна точка трябва да отчитаме следните фактори:

- За какъв вход става дума. На свой ред, този фактор има две компоненти:
 - ◆ Колко числа има във входа. Интуитивно е ясно, че колкото по-голямо е n (големината на входа), толкова по-бавно работи алгоритъмът.
 - ◆ При една и съща големина, какви са конкретните стойности на числата. Лесно се забелязва, че INSERTION SORT ще работи по-бързо, ако входът вече е сортиран, например $[1, 2, 3, 4]$. Нататък ще видим, че този алгоритъм работи най-бавно, когато входът е сортиран обратно, например $[4, 3, 2, 1]$.
- Каква е конкретната програмна реализация. Според Skiena (вж. определението на стр. 1), алгоритъмът е идеята зад някаква програма. Но истинските програми са повече от идеите зад тях, те имат конкретна реализация на конкретен език за програмиране. Очевидно някои реализации ще са по-брзи от други, защото са по-грамотно написани. Компилаторът не може да компенсира неграмотно писане на софтуер.
- Какъв е компилаторът.
- Каква е виртуалната машина, ако има такава.
- Каква е операционната система.

- Каква е компютърната архитектура и по-точно какъв е процесорът.

Да се даде точен отговор колко бързо би работила конкретна програма—например, с точност до милисекунда или наносекунда—реализираща INSERTION SORT, на конкретен език, върху конкретен компютър, е практически невъзможно дори за фиксиран вход[†]. Толкова прецизен отговор не е и необходим.

За да може да правим смислени изводи за бързината на алгоритми, правим серия от опростявания. Първото от тях е, че се фокусираме именно върху алгоритмите, а не върху програмните им реализации. Второто е допускането, че всички елементарни инструкции се изпълняват за единица време[‡], така че времето за изпълнение на алгоритъма върху някакъв вход се измерва с броя на елементарните инструкции, които се изпълняват преди завършването му.

Разликата между бързодействието на две програми за една и съща задача, които реализират два различни алгоритъма, по правило са много по-драстични от разликите в бързодействието на две програми, реализиращи един и същи алгоритъм. **По отношение на сложните, нетривиални задачи, печелившата стратегия за бързодействието е подобрение на алгоритъма.** Печалбата от по-бързия алгоритъм е толкова по-видима, колкото по-голям е входът. Като пример да разгледаме задачата СОРТИРАНЕ. Алгоритъмът INSERTION SORT, както ще видим нататък, в най-лошия случай работи във време, пропорционално на квадрата на размера на входа. Тоест, във време, пропорционално на n^2 . В следващи лекции ще разгледаме по-бързи алгоритми за сортиране, работещи във време, пропорционално на произведението $n \times \log n$. Да разгледаме стойностите на n^2 и $n \times \log n$ за няколко различни n . За целите на този пример логаритъмът е с основа 10.

n	10	100	1 000	100 000	1 000 000	100 000 000
n^2	100	10 000	1 000 000	10 000 000 000	1 000 000 000 000	10 000 000 000 000 000
$n \log n$	10	200	3 000	500 000	6 000 000	800 000 000

Разбира се, числата в тази таблица не са точният брой елементарни инструкции; точният брой, както казахме, е пропорционален на тези числа. Но на практика тези кофициенти на пропорционалност не са големи. И така, при размер на входа от сто милиона, огромната разлика между 10 000 000 000 000 000 и 800 000 000—**осем десетични порядъка**—води до това, че програмата на по-бързия алгоритъм е **смазващо** по-бърза[§] от тази на INSERTION SORT независимо от неща като използван компилатор, ниво на оптимизация на компилирането, операционна система, процесор, програмистки трикове за подобряване на скоростта и така нататък. Предимството, което дава по-бързият алгоритъм, по правило доминира с много над предимството на по-бързата компютърна технология, било хардуер, било софтуер.

Съществуват много по-екстремни примери за предимствата, които дават бързите алгоритми. За множество интересни и важни задачи наивният алгоритъм—който може да бъде предложен от най-общи съображения—се изпълнява в най-лошия случай за брой елементарни инструкции, който е пропорционален на експоненциална в n функция, да кажем 2^n , докато за същата задача има по-съвършени алгоритми, чието изпълнение като брой елементарни инструкции е пропорционален на, например, n^2 . Разликата между нарастването на 2^n и n^2 е много по-драстична от разликата между n^2 и $n \lg n$. Както ще стане ясно

[†]Съществуват специализирани компютри, изпълняващи критични дейности в *реално време*, при които наистина са необходими твърди гаранции за бързодействието на програмите, но това е далече отвъд материала в този курс.

[‡]Инструкциите на истински процесор се изпълняват за различен брой тактове

[§]При нормално грамотна реализация на двата алгоритъма.

от някоя следваща лекция[†], 2^{200} като брой инструкции, които трябва да бъдат изпълнени, е далече отвъд възможностите на всеки истински компютър, както в момента, така и в бъдеще. Това заслужава да бъде повторено.

Забележка

В реалния свят, програма, която трябва да изпълни 2^{200} стъпки, няма да завърши работата си **НИКОГА**, независимо от изпъзаната хардуерна и софтуерна технология за реализация на платформата, върху която тази програма работи. Ограничението идва от фундаментални принципи на физиката и **НЕ МОЖЕ** да бъде заобиколено чрез никакви технологични иновации и подобрения.

Следователно, опростеният модел, в който:

- разглеждаме само алгоритъма, а не програмната му реализация, и
- всяка елементарна инструкция се изпълнява за единица време,

е полезен от практическа гледна точка. Ако в този модел алгоритъм A е по-бърз от алгоритъм B за една и съща задача, най-вероятно програмната реализация на A ще е по-бърза от тази на B, като разликата ще е толкова по-очевидна, колкото по-голям е входът.

2.2.3 Определение на сложността по време

Сложността по време е функция на големината на входа. Но има много входове с една и съща големина. По-лошо, в нашия опростен модел, в който всяко число има един и същи размер (единица), има безброй много входове за всяка големина на входа. Да разгледаме отново СОРТИРАНЕ. Големината на входа n може да е всяко естествено число. Множеството от входовете с големина 1 е \mathbb{N} . Множеството от входовете с големина 2 е $\mathbb{N} \times \mathbb{N}$. Множеството от входовете с големина 3 е $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$. И така нататък. В общия случай, множеството от входовете с големина n е \mathbb{N}^n . Виждаме, че множеството от входовете е безкрайно (и по-точно, изброимо безкрайно) за всяко n .

Следното съображение ни позволява да разглеждаме само краен брой входове за всяка големина на входа. Съображението е по отношение на СОРТИРАНЕ, но лесно може да бъде пренесено и при други изчислителни задачи[‡]. Разглеждаме само такива сортирания, които се базират на **директни сравнения** на числата (които биват сортирани)[§]. Лесно се вижда, че в такъв случай сортирането на следните входове

$$\begin{aligned} & (2, 1, 3) \\ & (2000, 1000, 3000) \\ & (2 \times 10^9, 1, 10^{10}) \end{aligned}$$

[†] Става дума за принципа на Ландауер [42], който налага добра граница за енергията, необходима за обръщането на един бит от нула в единица или обратно.

[‡] Всъщност е доста трудно да кажем колко са съществено различните входове за, да речем, алгоритъма на Крускал. Тук се има предвид следното – ясно е, че за всеки алгоритъм, който ще разглеждаме, безкрайното множество от входове с големина n може да бъде разбито на краен брой класове на еквивалентност, като върху входовете от даден клас алгоритъмът ще работи еднакво бързо.

[§] Съществуват и други възможности за сортиране, ако са известни някакви ограничения за входа. Например, ако сортираме само нули и единици, може просто да преbroим колко са нулите.

ще протече по един и същи начин, и то в много силен смисъл. Всеки сортиращ алгоритъм, базиран на директни сравнения, ще завърши работата си в същия брой стъпки за всеки от тези три входа, като на всяка стъпка ще изпълнява една и съща инструкция. Разбира се, това е при не много реалистичното допускане, че $10^{10^{10}}$ има размер единица и действията върху него стават за единица време.

Сега ще дефинираме *неразличими по отношение на брзодействието входове*, за произволен алгоритъм за СОРТИРАНЕ, базиран на директни сравнения. За всяка големина на входа n , тоест за всяко $n \in \mathbb{N}^+$, въвеждаме *релация на неразличимост* R_n така: за всеки два входа X и Y с големина n , $(X, Y) \in R_n$ тогава и само тогава, когато:

1. X и Y имат един и същи брой от малкия елемент, един и същи брой от следващият по големина, и така нататък, имат един и същи брой от най-големия си елемент[†]. Нека m е броят на различните големини на елементи в X и Y , където $1 \leq m \leq n$.
2. За $1 \leq k \leq m$, всички k -и по големина елементи се намират на едни и същи позиции в X и в Y .

Например, нека $n = 5$ и нека има два най-малки елемента, два по-големи и един най-голям. Тогава $m = 3$. Ако $X_1 = (6, 55, 6, 55, 7075)$ и $Y_1 = (36, 37, 36, 37, 38)$, то $(X_1, Y_1) \in R_5$, понеже двата най-малки елемента са на позиция 1 и 3 както в X_1 , така и в Y_1 , средните по големина два елемента са на позиции 2 и 4 както в X_1 , така и в Y_1 , и най-големият елемент е на позиция 5 както в X_1 , така и в Y_1 . От друга страна, ако $X_2 = (55, 6, 6, 55, 7075)$, то $(X_2, Y_1) \notin R_5$, защото в X_2 най-малките елементи са на позиции 2 и 3.

Тривиално се показва, че за всяко n , R_n е релация на еквивалентност. Нейните класове на еквивалентност са само краен брой за всяко n . Интересен въпрос, на който сега ще отговорим, е колко точно са тези класове на еквивалентност. Ако не допускаме повтаряне на елементи, то класовете на еквивалентност са $n!$, защото всеки клас се определя от това на коя позиция е първият елемент, на коя е вторият, и така нататък до n -ия.

Допълнение 10: Колко са класовете на екв. на входа при сортиране

В по-общия случай, когато допускаме повтаряне на елементи, нека броят на уникалните елементи е m , където $1 \leq m \leq n$. Броят на начините числата a_1, a_2, \dots, a_n да бъдат групирани в точно m групи, във всяка от които числата са равни, а числата от две различни групи са различни, е $\binom{n}{m}$. Да си припомним, че $\binom{n}{m}$ за $1 \leq m \leq n$ е броят на разбиванията на n -елементно множество на точно m подмножества. Например, $\binom{4}{2} = 7$, и действително има точно 7 начина a_1, a_2, a_3 и a_4 да имат точно две различни големини:

- a_1, a_2 и a_3 с една и съща големина, a_4 с друга големина
- a_1, a_2 и a_4 с една и съща големина, a_3 с друга големина
- a_1, a_3 и a_4 с една и съща големина, a_2 с друга големина
- a_2, a_3 и a_3 с една и съща големина, a_1 с друга големина
- a_1 и a_2 с една и съща големина, a_3 и a_4 с друга големина
- a_1 и a_3 с една и съща големина, a_2 и a_4 с друга големина
- a_1 и a_4 с една и съща големина, a_2 и a_3 с друга големина

Броят на класовете на класовете на еквивалентност на R за дадено m е $m! \binom{n}{m}$. Напри-

[†]Тук разглеждаме общия случай, в който във входа може да има повтарящи се елементи. Ако елементите на входа са уникални, то има точно един най-малък елемент, точно един по-голям, и така нататък, точно един най-голям елемент.

мер, при $n = 4$ и $m = 2$, има точно 14 начина за подредбата на различните големини на числата:

- a_1, a_2 и a_3 с една и съща големина, a_4 с друга големина, и $a_1, a_2, a_3 < a_4$
- a_1, a_2 и a_3 с една и съща големина, a_4 с друга големина, и $a_1, a_2, a_3 > a_4$
- a_1, a_2 и a_4 с една и съща големина, a_3 с друга големина, и $a_1, a_2, a_4 < a_3$
- a_1, a_2 и a_4 с една и съща големина, a_3 с друга големина, и $a_1, a_2, a_4 > a_3$
- a_1, a_3 и a_4 с една и съща големина, a_2 с друга големина, и $a_1, a_3, a_4 < a_2$
- a_1, a_3 и a_4 с една и съща големина, a_2 с друга големина, и $a_1, a_3, a_4 > a_2$
- a_2, a_3 и a_3 с една и съща големина, a_1 с друга големина, и $a_2, a_3, a_4 < a_1$
- a_2, a_3 и a_3 с една и съща големина, a_1 с друга големина, и $a_2, a_3, a_4 > a_1$
- a_1 и a_2 с една и съща големина, a_3 и a_4 с друга големина, и $a_1, a_2 < a_3, a_4$
- a_1 и a_2 с една и съща големина, a_3 и a_4 с друга големина, и $a_1, a_2 > a_3, a_4$
- a_1 и a_3 с една и съща големина, a_2 и a_4 с друга големина, и $a_1, a_3 < a_2, a_4$
- a_1 и a_3 с една и съща големина, a_2 и a_4 с друга големина, и $a_1, a_3 > a_2, a_4$
- a_1 и a_4 с една и съща големина, a_2 и a_3 с друга големина, и $a_1, a_4 < a_2, a_3$
- a_1 и a_4 с една и съща големина, a_2 и a_3 с друга големина, и $a_1, a_4 > a_2, a_3$

Броят на класовете на еквивалентност на R_n за дадено n е

$$\sum_{m=1}^n m! \left\{ \begin{matrix} n \\ m \end{matrix} \right\}$$

Да резюмираме. В нашия модел, в който всички числа са с големина единица и елементарните операции върху всяко число стават за единица време, по отношение на произволен сортиращ алгоритъм A , базиран на директни сравнения, множеството от входове $\times_{i=1}^n \mathbb{N}$ се разбива на $\sum_{m=1}^n m! \left\{ \begin{matrix} n \\ m \end{matrix} \right\}$ класа, като за всеки два входа X и Y от един и същи клас, $A(X)$ и $A(Y)$ работят в един и същи брой стъпки. От гледна точка на скоростта на изпълнение, X и Y са *неразличими*.

За други изчислителни задачи разбиването на (изброймо) безкрайното множество от входовете с дадена големина на краен брой класове на еквивалентност става може да е по друг начин, но за всяка задача можем да дефинираме смислено такова разбиване, като за всеки два входа X и Y от един и същи клас, $A(X)$ и $A(Y)$ извършва в един и същи брой стъпки за всеки алгоритъм A за нея. Това ни позволява, когато разглеждаме сложността по време, да смятаме, че входовете от дадена големина са само краен брой.

Определение 2: Сложност по време

Нека Π е изчислителна задача и A е алгоритъм за нея. За всяка големина на входа $n \in \mathbb{N}^+$, нека $\mathfrak{I}(n)$ е крайното множество от различните входове с големина n и за всеки вход κ нека $f(\kappa)$ е броят стъпки, които се изпълняват от A върху него. Тогава, за всяко n , *сложността по време на A в най-лошия случай* е

$$T_A(n) = \max \{f(\kappa) \mid \kappa \in \mathfrak{I}(n)\}$$

сложността по време на A в най-добрия случай е

$$P_A(n) = \min \{f(\kappa) \mid \kappa \in \mathfrak{I}(n)\}$$

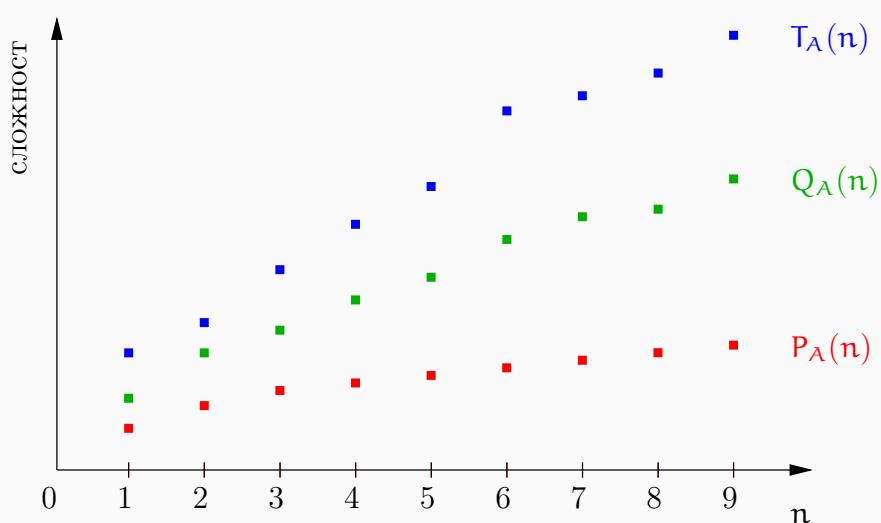
средната сложност по време на A е

$$Q_A(n) = \frac{1}{|\mathfrak{I}(n)|} \sum_{\kappa \in \mathfrak{I}(n)} f(\kappa)$$

Забележка: използваната нотацията, например " $T_A(n)$ " и т. н., не е общоприета.

Фигура 2.1 показва нагледно трите вида сложност.

Фигура 2.1 : Сложност по време в най-лошия случай $T_A(n)$, в най-добрия случай $P_A(n)$ и средна сложност $Q_A(n)$. Ординатата е сложността като брой стъпки.



Средната сложност винаги е между сложността в най-лошия и сложността в най-добрия случай. По правило функциите на сложността са строго нарастващи, защото (по правило) един и същи алгоритъм работи по-бавно върху по-голям вход, но има и изключения, както ще видим надолу в примера с Евклидовия алгоритъм.

На практика най-често използвана е сложността по време в най-лошия случай (*worst-case time complexity* на английски), по две основни причини. Първо, изследването на сложността в най-лошия случай е много по-лесно от изследването на средната сложност, в което ще се убедим в следваща лекция при анализа на **QUICKSORT**. Анализирането на средната сложност изисква значително по-задълбочени математически познания и значително по-сложни

техники, дори при неявното допускане в определението на $Q_A(n)$, че всички входове с дадена големина са еднакво вероятни. Втората причина е, че резултатът за най-лошия случай е твърда гаранция, че по-лошо не може да бъде.

Сложността в най-добраия случай не се ползва, тъй като практически всеки важен алгоритъм може да бъде подобрен като скорост за само един вход (по един такъв вход за всяка големина). Например, всеки алгоритъм A за задачата ХАМИЛТОНОВ Път, която ще дефинираме нататък, може да бъде направен много бърз върху един конкретен вход (за всяка големина). По-точно, независимо как точно работи A , той може да бъде модифициран, като в самото му начало се добави тестване на една конкретна пермутация на върховете – дали задава хамилтонов път. Ако се окаже, че тя задава хамилтонов път, то отговорът на въпроса дали има хамилтонов път очевидно е утвърдителен и прекратяваме изпълнението. Ако се окаже, че тя не задава хамилтонов път, от това не следва, че няма такъв, и продължаваме с това, което A прави. С други думи, към всеки алгоритъм, дори най-бавния, може да се пришие напълно изкуствено тестване на някакъв конкретен вход. Този трик би подобрил изкуствено сложността в най-добраия случай до теоретично оптималната, без това да ни казва нищо съществено за това, колко е бърз A . В примера с ХАМИЛТОНОВ Път, според преобладаващото мнение, всеки алгоритъм за тази задача е с експоненциална сложност както в най-лошия, така и в средния случай, но чрез споменатия трик може да направим кой да е алгоритъм за тази задача да работи в линейно време в най-добраия случай. Такова изкуствено увеличаване на бързодействието чрез подобреие върху само един вход е напълно безполезно.

2.2.4 Анализ на сложността по памет

Сложността по памет на даден алгоритъм A върху даден вход е броят на елементите памет, които A ползва, без да броим паметта, в която се разполага входа. С други думи, гледаме само работната памет на алгоритъма. Сложност по памет в най-лошия, средния и най-добраия случай се дефинира по начин, аналогичен на сложността по време.

Една значителна принципна разлика между сложността по време и сложността по памет е, че сложността по време (било в най-лошия случай, било средната) най-често е строго растяща функция на големината на входа, докато по отношение на сложността по памет е напълно възможно даден алгоритъм да ползва само константна работна памет, за всяка големина на входа. Алгоритми, които ползват константна работна памет, се наричат *in-place*.

2.2.5 Проблеми със сложността по време, до които води нашия модел

Да разгледаме отново Евклидовия алгоритъм в модерната му формулировка, независимо дали рекурсивната или итеративната. Да се опитаме да направим груб анализ на сложността му по време в най-лошия случай. Очевидно, за някои двойки входни числа, а именно тези, за които a е кратно на b , алгоритъмът ще завърши много бързо. За други двойки входни числа, алгоритъмът ще работи много повече. От [37] става ясно, че за всяко цяло положително число m , колкото и голямо да е, има двойка (a, b) , такава че Евклидовият алгоритъм с този вход работи за повече от m стъпки. Но в нашия опростен модел, всички двойки (a, b) имат една и съща големина, а именно 2, защото се състоят от две числа, а ние приехме, че всяко число има големина единица. Тогава всеки вход има големина точно 2. Следва, че функциите на сложността, които имат аргумент-големината на входа, не могат да бъдат дефинирани смислено, защото имаме само една големина на вход, а именно 2. За тази големина имаме входове, за които алгоритъмът работи в повече стъпки от всяко предварително избрано число. Излиза, че сложността по време е безкрайност ...

Този парадоксален извод се дължи само на допускането, че всяко число има големина единица. Ако приемем обаче по-сложния модел, в който всяко число има големина, пропорционална на логаритъм от него при основа 2^{\dagger} , парадоксът изчезва. Тогава Евклидовият алгоритъм работи във време, пропорционално на квадрата на големината на входа (вж. [37]).

Както вече казахме в подсекция 1.1.5, моделите налагат някакви опростявания, иначе нямаше да са модели. Доколкото тези опростявания са смислени и полезни на практика, съответните модели също са смислени и полезни. Но в случаите, в които опростяванията и допусканията водят до безсмислени и безполезни резултати, съответните модели стават неизползваеми и трябва да се търсят други модели, по-сложни и детайлни. Така че моделът, в който всяко число има големина единица, не е иманентно лош или събркан, просто в някои случаи е полезен (например, сортиращите алгоритми), в други, не е (например, Евклидовият алгоритъм).

Аналогично, пътната карта на България е модел на истинската пътна мрежа (защото съдържа само най-важната информация). Тази карта, в която градовете са отбелязани с точки, е много полезна в някои случаи и напълно безполезна в други случаи. Ако искаме да стигнем с кола от София до Варна, например, и не знаем пътя, пътната карта, в която Варна е точка, е полезна до момента, в който влезем във Варна. За отиването до конкретен адрес там, моделът, в който Варна е точка, вече не върши работа. Но това не означава, че пътната карта е излишна. Просто в някои случаи тя е полезен модел, в други, не. По същия начин, моделът, в който числата имат големина единица, е полезен в някои случаи и безполезен в други.

2.3 Асимптотични нотации

2.3.1 Опит за намиране на точната сложност по време на SELECTION SORT и INSERTION SORT

Да разгледаме отново алгоритъма INSERTION SORT, въведен на стр. 31. Въпреки че в подсекция 2.2.2 установихме, че е безсмислено да търсим точен израз за сложността му по време като функция от големината на входа, с учебна цел сега ще се опитаме да направим точно това. Спазваме допусканията, че числата имат големина единица и че всяка елементарна инструкция отнема единица време.

INSERTION SORT($A[1, 2, \dots, n]$): масив от цели числа)

```

1   for i ← 2 to n
2       key ← A[i]
3       j ← i - 1
4       while j > 0 and A[j] > key do
5           A[j + 1] ← A[j]
6           j ← j - 1
7       A[j + 1] ← key

```

Колко стъпки отнема изпълнението на този алгоритъм върху n числа? Използваме следните съображения.

[†] $\log_2 n$ е горе-долу броят на битовете, необходими за записването на n в двоична позиционна бройна система.

- В нашия опростен модел това, което е на ред 1—а именно инициализацията на управляващата променлива i , проверката за продължаване и инкрементирането—е една инструкция.[†] Тя се изпълнява n пъти.

Да поясним защо ред 1 се изпълнява n пъти. По принцип, всяка инструкция от вида **for** $i \leftarrow a$ **to** b се изпълнява точно $b - a + 2$ пъти (ако $a \leq b$), защото:

- ◆ тялото на цикъла се изпълнява точно $b - a + 1$ пъти,
- ◆ а **for** $i \leftarrow a$ **to** b се изпълнява веднъж за всяко изпълнение на тялото цикъла и после **още веднъж**, когато условието за продължаване вече не е вярно.

В този случай, $n - 2 + 2$ е точно n .

- Тялото на цикъла **for** на редове 1–7 се изпълнява, както казахме, $n - 2 + 1 = n - 1$ пъти. Следователно, всеки от редовете 2, 3 и 7 се изпълнява $n - 1$ пъти, и тъй като всеки от тях има по една инструкция, която се изпълнява за една стъпка, това са общо $3(n - 1)$ стъпки.
- При всяко изпълнение на **for** цикъла, а такива има $n - 1$ на брой, изпълнението достига ред 4 (началото на **while** цикъла) поне веднъж. Но при дадено изпълнение на **for** цикъла, ред 4 може да бъде изпълняван повече от веднъж – това зависи от **конкретния вход**. С други думи, точно колко пъти ще се изпълни вложеният цикъл при някакво i зависи от сравненията между $A[j]$ и key . Освен това, вложеният цикъл **не се изпълнява за една инструкция**. Следователно, за да оценим точно колко пъти ще бъдат изпълнени всеки от редовете 4–6, трябва да извършим по-подробен анализ.

Нека k_i е броят на изпълненията на ред 4 за всяко $i \in \{2, \dots, n\}$. Очевидно $k_i \in \{1, \dots, i\}$, защото ред 4 се изпълнява поне веднъж и най-много i пъти. Защо най-много i пъти? – защото j може да стане най-малко 0, бивайки инициализирано със стойност $i - 1$ на ред 3.

Всеки от редове 5 и 6 се изпълнява $k_i - 1$ пъти. Тогава ред 4 се изпълнява $\sum_{i=2}^n k_i$ пъти **общо за цялото изпълнение на алгоритъма**, а редове 5 и 6 се изпълняват по $\sum_{i=2}^n (k_i - 1)$ пъти. Общо изпълнението на **while** цикъла “струва” $\sum_{i=2}^n k_i + 2 \sum_{i=2}^n (k_i - 1) = (3 \sum_{i=2}^n k_i) - 2(n - 1)$ стъпки по време на цялото изпълнение на алгоритъма.

И така, в нашия опростен модел, изпълнението на алгоритъма става в

$$n + 3(n - 1) + 3 \left(\sum_{i=2}^n k_i \right) - 2(n - 1) = 2n - 1 + 3 \sum_{i=2}^n k_i$$

[†]При програмиране на истински компютър това не е така. Следният тривиален **for** цикъл на езика C:

```
for (i=2; i <= n; i++);
```

като фрагмент от програма се компилира върху Intel iCore3 процесор, Ubuntu Linux, 64 битов gcc компилатор без оптимизация, в следния код (адресите и отместванията, естествено, варират):

```
0x4005bb <main+30> movl $0x2,-0x4(%rbp)
0x4005c2 <main+37> jmp 0x4005c8 <main+43>
0x4005c4 <main+39> addl $0x1,-0x4(%rbp)
0x4005c8 <main+43> mov -0x8(%rbp),%eax
0x4005cb <main+46> cmp %eax,-0x4(%rbp)
0x4005ce <main+49> jle 0x4005c4 <main+39>
```

Както виждаме, при истинския компютър инкрементирането с единица **addl**, проверката за край на изпълнението на цикъла **cmp**, и условният скок **jle** в изпълнението, са отделни инструкции.

стъпки. За да довършим анализа, трябва да определим минималната и максималната възможна стойност на сумата. k_i може да е най-малко 1, когато тялото на **while** цикъла не се изпълнява изобщо, и най-много i , когато тялото на **while** цикъла се изпълнява $i - 1$ пъти.

Показахме, че сложността по време в най-добрния случай е

$$P(n) = 2n - 1 + 3 \sum_{i=2}^n 1 = 2n - 1 + 3n - 3 = 5n - 4 \quad (2.1)$$

и сложността по време в най-лошия случай е

$$T(n) = 2n - 1 + 3 \sum_{i=2}^n (i - 1) = 2n - 1 + \frac{3n(n - 1)}{2} = \frac{3n^2 + n}{2} - 1 \quad (2.2)$$

Ще анализираме сложността на още един алгоритъм по аналогичен начин.

SELECTION SORT($A[1, 2, \dots, n]$: масив от цели числа)

```

1   for i ← 1 to n - 1
2       for j ← i + 1 to n
3           if A[j] < A[i]
4               swap(A[i], A[j])

```

Ред 1 се изпълнява общо $n - 1 - 1 + 2 = n$ пъти. Ред 2 се изпълнява $n - i + 1$ за всяко i , общо $\sum_{i=1}^{n-1} n - i + 1 = \frac{(n+2)(n-1)}{2}$. Ред 3 се изпълнява $n - i$ за всяко i , общо $\sum_{i=1}^{n-1} n - i = \frac{n(n-1)}{2}$. Ред 3 се изпълнява общо k пъти за някакво k , такова че $0 \leq k \leq \frac{n(n-1)}{2}$, в зависимост от конкретния вход. Общо сложността е

$$n + \frac{(n+2)(n-1)}{2} + \frac{n(n-1)}{2} + k = n^2 + n - 1 + k$$

Тогава сложността в най-добрия случай е

$$P'(n) = n^2 + n - 1 \quad (2.3)$$

и сложността в най-лошия случай е

$$T'(n) = n^2 + n - 1 + \frac{n(n-1)}{2} = \frac{3n^2 + n}{2} - 1 \quad (2.4)$$

2.3.2 Отказ от точна оценка и търсене на приблизителна оценка на сложността

От изложението в подсекция 2.2.2 трябва да е станало ясно, че точни изрази като (2.1), (2.2), (2.3) и (2.4) не са необходими. Нашият модел на сложността е достатъчно далече от реалността, така че събирамото -1 , което имаме в (2.4), няма практически никакво значение по отношение на каквато и да е софтуерна реализация на SELECTION SORT. Както казахме, не можем да предвидим времето за изпълнение с точност до милисекунда, така че по отношение на реално изпълнение, събирамото -1 не ни казва *нищо*. Нещо повече, цялото събирамо $\frac{1}{2}n - 1$ е безсмислено. Ако разполагаме с конкретна реализация на алгоритъма и я тестваме върху различни тестови входове, за които имаме най-лоша сложност (обратно сортирани последователности) и после анализираме данните, ще стане ясно, че бързодействието се интерполира от някаква квадратична функция, и толкова. Фактори като конкретна

операционна система, компилатор, архитектура и виртуална машина, както и умението на програмиста да пише бързи програми върху конкретната машина, със сигурност ще се отразят повече на реалното бързодействие.

Това, което има значение в (2.4) е фактът, че от трите събирами най-бързо растящото е квадратичната функция $\frac{3}{2}n^2$. Тя задава **квадратична сложност по време**. Ако входът е достатъчно голям, дори и най-добрата конкретна реализация ще има квадратична сложност в n . Никакъв избор на конкретна операционна система, компилатор, програмистки трикове и така нататък не могат да подобрят квадратичната сложност до, да речем, линейна (след малко ще дефинираме линейна сложност).

И така, най-същественото за сложностите на INSERTION SORT и SELECTION SORT е, че в най-лошия случай[†] те са квадратични функции.

В нашата поредица от допускания, които опростяват нещата и ни позволяват изобщо да правим анализ на сложността на алгоритми, следва допускането, че **степента на нарастване** на функцията на сложността е всичко, което ни интересува за нея. Защо? **Защото се интересуваме само от тенденцията на изменение на сложността при неограничено нарастване на големината на входа.** Никаква конкретна големина на входа не ни интересува. Никаква нейна конкретна стойност, ако ще да е $10\ 000\ 000^{10\ 000\ 000^{10\ 000\ 000}}$, не е достатъчно голяма, за да спрем до нея. След малко ще дадем прецизно определение на “степен на нарастване”, засега ще се задоволим с примера с анализа на двата алгоритъма горе. И на двата функциите на сложността в най-лошия случай са квадратични. Ще смятаме, че сложността им е еднаква: квадратична.

И така, правим следните допускания:

Допускане 1. Ако изразът за сложността на даден алгоритъм е сума, разглеждаме най-

бързо растящата функция между събирамите, в *асимптотичния смисъл*. А именно, ако изразът е $f_1(n) + f_2(n) + \dots + f_k(n)$ и $\lim_{n \rightarrow \infty} \frac{f_i(n)}{f_1(n)} = \infty$ за всяко $i \in \{2, \dots, k\}$, то $f_1(n)$

е най-бързо растящата в асимптотичния смисъл функция между $f_1(n), \dots, f_k(n)$ [‡].

Ще смятаме, че най-бързо растящата в този смисъл функция доминира над останалите и тях можем да пренебрегнем.

Допускане 2. За два различни алгоритъма, ако водещите функции в горния смисъл в из-

разите за сложността са $f_1(n)$ и $g_1(n)$ и $0 < \lim_{n \rightarrow \infty} \frac{f_1(n)}{g_1(n)} < \infty$, ще смятаме, че тези алгоритми работят еднакво бързо.

Допускане 1 не винаги е реалистично. Смисълът да се разглежда само тенденция на нарастването, когато n клони към безкрайност, е ясен: по-големите входове са по-интересни. Но на практика това е вярно само донякъде. На практика, наистина поведението на алгоритъм за сортиране върху вход с големина $10\ 000\ 000$ е по-интересно от поведението му върху вход с големина $10\ 000\ 000^{10\ 000\ 000}$ е още по-интересно, а това върху вход с големина $10\ 000\ 000^{10\ 000\ 000^{10\ 000\ 000}}$, дори още по-интересно. Последните две числа са напълно отвъд възможностите на всеки истински компютър, сегашен и бъдещ, така че програмата, реализираща въпросния алгоритъм, никога няма да има възможност да демонстрира бързината си върху входове с такива размери. Когато се фокусираме само върху водещото събирамо, ние твърдим, че измежду два алгоритъма, единият от които има сложност $n^2 + n$, а другият, $n\sqrt{n} + 10\ 000\ 000^{10\ 000\ 000^{10\ 000\ 000}} n$,

[†]Казахме, че сложността в най-добрая случай не е особено информативна и не се ползва, но си заслужава да се отбележи, че в най-добрая случай INSERTion sort има сложност, която е линейна функция, без алгоритъмът да е модифициран чрез изкуствено пришити инструкции за само един най-добър случай.

[‡]Тук предполагаме, че всички те са положителни функции.

вторият работи по-бързо, защото неговата водеща функция $n\sqrt{n}$ расте по-бавно от n^2 , водещата функция на първия. Очевидно съществува стойност на големината на входа n_0 , такава че за всяко $n \geq n_0$, вторият алгоритъм печели като бързодействие, но това n_0 е твърде голямо. Както вече стана ясно, то е число, което никога не е големина на вход в реалния свят.

Допускане 2, тоест игнорирането на мултипликативната константа във водещата функция, също не винаги е реалистично. Според него, два алгоритъма, единият от които има сложност $n^2 + n$, а другият има сложност $10000000^{10000000^{1000000}} n^2$, са с една и съща сложност, а именно квадратична. “Оправданието” ни е, че алгоритмите, които се използват на практика и почиват върху прагматични идеи, по правило имат сложности, в които тези мултипликативни константи не са фрапиращо големи и никога не са от порядъка на $10000000^{10000000^{1000000}}$, така че опростяването не е прекалено грубо. В теорията обаче са известни алгоритми, при които тази мултипликативна константа е кула от степени на двойката

$$2^{2^{2^{\cdot^{\cdot^{\cdot}}^2}}}$$

чиято височина е кула от степени на двойката, чиято височина е кула от степени на двойката, и така нататък няколко пъти [30]. Очевидно такъв алгоритъм е непрактичен дори за вход с големина единица.

Да обобщим. Когато разглеждаме тенденцията на нарастването при неограничено нарастване на аргумента n , ние просто правим поредното опростяващо допускане. Причината да го правим е, че **така е по-лесно**. Освен това, практиката показва, че често—но **не винаги**—результатите, които ни дава това опростяване, са добре корелирани с качествата на реалните програми, реализиращи съответните алгоритми.

В края на подсекцията, един цитат от книгата “Computational Complexity” на Papadimitriou [51, стр. 7].

Any attempt, in any field of mathematics, to capture an intuitive, real-life notion (for example, that of a “smooth function” in real analysis) by a mathematical concept (such as C_∞) is bound to include certain undesirable specimens, while excluding others that arguably should be embraced.

Следователно, както и да опитваме да формализираме житейското понятие “бърз алгоритъм” или дори “бърза програма”, неизбежно е според формалните ни дефиниции някои алгоритми, които не бихме нарекли бързи, да бъдат класифицирани като такива, а от друга страна, алгоритми, които смятаме за прилично бързи, да бъдат класифицирани като бавни.

2.3.3 Асимптотична нотация Тета

Използвайки асимптотичните нотации, които сега ще дефинираме, можем да изразяваме сложността на изследваните от нас алгоритми **приближително**. Практиката е показвала, че тези приближения са смислени: ако сложността на алгоритъм X , изразена в Θ -нотация, е по-малка от сложността на алгоритъм Y , пак в Θ -нотация, твърде възможно е практическата реализация на алгоритъм X да работи по-бързо от практическата реализация на алгоритъм Y .

Забележете, че дефинициите не използват нотацията \lim .

Конвенция 1: функции от \mathbb{R}^+ в \mathbb{R}^+

Отсега нататък допускаме, че всички функции, които разглеждаме, имат домейн и кодомейн \mathbb{R}^+ , освен ако изрично не е казано нещо друго. Въпреки това, типичното име на променливата е n , а не x .

Ако n е големина на входа на алгоритъм, то n , разбира се, е **цяло** положително число. Но в тази глава разглеждаме само математическите основи на сложността на алгоритмите и затова си позволяваме да допуснем, че $n \in \mathbb{R}^+$.

Определение 3: $\Theta(g(n))$

За всяка функция $g(n)$:

$$\Theta(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \exists c_1, c_2 > 0 \exists n_0 \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

Формално, $\Theta(g(n))$ е безкрайно множество от функции $f(n)$, за които е в сила някакво условие. Ако искаме да кажем, че $h(n)$ е една от тях, пишем $h(n) = \Theta(g(n))$ заместо формално коректното $h(n) \in \Theta(g(n))$. Причината е историческа – прието е да се използва знакът за равенство, а не формално коректният знак за принадлежност към множество. Изразът “ $h(n) = \Theta(g(n))$ ” се чете: “ $h(n)$ е Тета-голямо от $g(n)$ ”.

Ето пример за много подробно доказателство, че една функция е Тета-голямо от друга функция. Ще покажем, че $n^2 + 10n + 12 = \Theta(n^2)$, използвайки Определение 3. Съгласно определението, трябва да съществуват константи $c_1, c_2 > 0$ и стойност n_0 на аргумента, такива че за всяко $n \geq n_0$ да е вярно

$$0 \leq c_1 n^2 \leq n^2 + 10n + 12 \leq c_2 n^2$$

Това са всъщност три неравенства. Първото неравенство $0 \leq c_1 n^2$ е очевидно: щом $c_1 > 0$, то можем да вземем $n_0 = 1$ и тогава $\forall n \geq n_0 : 0 \leq c_1 n^2$. Да разгледаме второто неравенство $c_1 n^2 \leq n^2 + 10n + 12$. Имаме право да разделим двете страни на n^2 , защото разглеждаме само положителни n :

$$c_1 n^2 \leq n^2 + 10n + 12 \Leftrightarrow c_1 \leq 1 + \frac{10}{n} + \frac{12}{n^2}$$

Можем да вземем просто $c_1 = 1$, понеже $\forall n \geq n_0 : 1 \leq 1 + \frac{10}{n} + \frac{12}{n^2}$. Тук отново n_0 е 1. Да разгледаме третото неравенство $n^2 + 10n + 12 \leq c_2 n^2$:

$$n^2 + 10n + 12 \leq c_2 n^2 \Leftrightarrow 1 + \frac{10}{n} + \frac{12}{n^2} \leq c_2$$

Ако искаме да използваме същата стойност на n_0 , която използвахме в предните две неравенства, а именно $n_0 = 1$, можем да вземем някакво голямо c_2 , да кажем $c_2 = 100$, и тогава лесно се вижда, че $\forall n \geq n_0 : 1 + \frac{10}{n} + \frac{12}{n^2} \leq 100$. Други стойности за c_2 и n_0 също биха свършили работа за доказателството, примерно $c_2 = 3$ и $n_0 = 12$.

Ако в първото, второто и третото неравенство сме открили различни стойности на n_0 , всяка от които върши работа за съответното неравенство, то за цялото доказателство е достатъчно да вземем най-голямата от тях. В конкретния случай, ако за първото и второто неравенство $n_0 = 1$ върши работа, а за третото работа върши $n_0 = 12$, то за цялото доказателство ще вземем $n_0 = 12$.

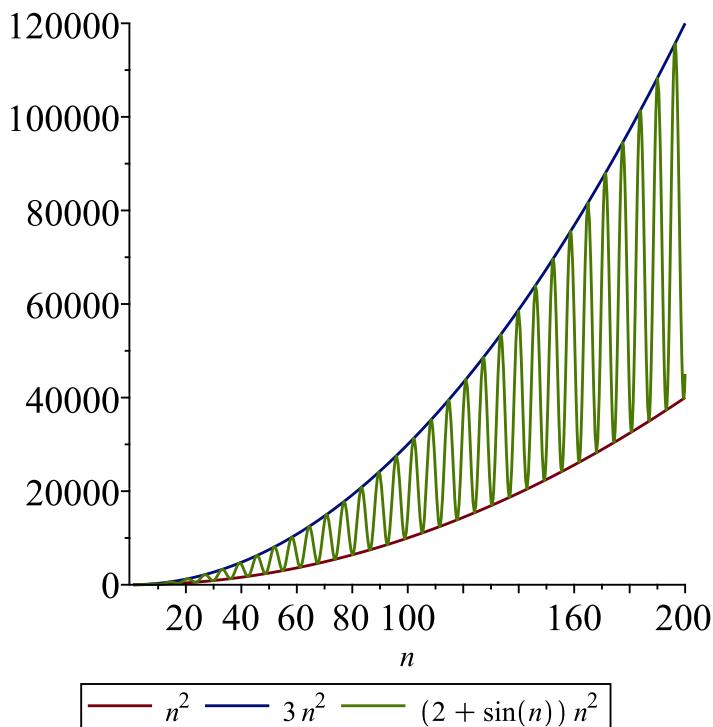
И накрая отбелязваме, че съществуват повече от една наредени тройки (c_1, c_2, n_0) , които може да се използват в доказателството.

Наблюдение 4

Забележете, че “ $\lim_{n \rightarrow \infty} \frac{h(n)}{g(n)}$ съществува и е равно на някакво L , такова че $0 < L < \infty$ ” е по-силно твърдение от “ $h(n) = \Theta(g(n))$ ”. С други думи, Тета-нотацията е по-обща от изразяването чрез граница, защото границата $\lim_{n \rightarrow \infty} \frac{h(n)}{g(n)}$ може да не съществува и въпреки това $h(n)$ да е Тета-голямо от $g(n)$.

Като пример за това да разгледаме $(2 + \sin n)n^2$ и n^2 . Вярно е, че $(2 + \sin n)n^2 = \Theta(n^2)$. За доказателството можем да вземем константи $c_1 = 1$ и $c_2 = 3$. Обаче границата $\lim_{n \rightarrow \infty} \frac{(2 + \sin n)n^2}{n^2} = \lim_{n \rightarrow \infty} 2 + \sin n$ не съществува. Разгледайте Фигура 2.2[†]. Графиката на $(2 + \sin n)n^2$ осцилира между графиките на n^2 и $3n^2$, като амплитудата на осцилирането нараства неограничено с нарастването на аргумента. Следователно, във формализма на границите не бихме могли да изразим директно, че $(2 + \sin n)n^2$ и n^2 са асимптотично “близки”.

Фигура 2.2 : Графиките на n^2 , $3n^2$ и $(2 + \sin n)n^2$.



Лема 2

За всички функции $f(n)$, $g(n)$ и $h(n)$:

$$f(n) = \Theta(f(n)) \tag{2.5}$$

$$f(n) = \Theta(g(n)) \rightarrow g(n) = \Theta(f(n)) \tag{2.6}$$

$$f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \rightarrow f(n) = \Theta(h(n)) \tag{2.7}$$

Доказателство: Равенствата 2.5 и 2.7 следват очевидно от рефлексивността и транзитивността на релацията “ \leqslant ” в Определение 3. Да разгледаме равенство 2.6. В термините на

[†]Фигура 2.2 е направена с Maple(TM).

Определение 3, то казва:

$$\begin{aligned} (\exists c_1, c_2 > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)) \rightarrow \\ (\exists c'_1, c'_2 > 0 \exists n'_0 > 0 \forall n \geq n'_0 : 0 \leq c'_1 \cdot f(n) \leq g(n) \leq c'_2 \cdot f(n)) \end{aligned}$$

Но това е очевидно вярно, ако вземем $c'_2 = \frac{1}{c_1}$, $c'_1 = \frac{1}{c_2}$, и $n'_0 = n_0$. \square

Следствие 1

От равенство 2.6 на Лема 2 веднага следва, че $f(n) = \Theta(g(n))$ тогава и само тогава, когато $g(n) = \Theta(f(n))$.

Определение 4

Линеен е всеки алгоритъм, чиято сложност в най-лошия случай е $\Theta(n)$. Аналогично, ако сложността в най-лошия случай е $\Theta(n^2)$, $\Theta(n^3)$ или $\Theta(n^4)$, казваме съответно, че алгоритъмът е *квадратичен, кубичен или квартичен*.

Използвайки Тета-нотацията, можем да запишем сложността в най-лошия случай $T(n)$ на INSERTION SORT и $T'(n)$ на SELECTION SORT така:

$$\begin{aligned} T(n) &= \Theta(n^2) \\ T'(n) &= \Theta(n^2) \end{aligned}$$

От гледна точка на Тета-асимптотиката, по-прецизен анализ на тези алгоритми няма. Те имат квадратична сложност по време и толкова.

Конвенция 2: Еднопосочност на знака за равенство при Θ

Записът $\Theta(n^2) = T(n)$ е **некоректен** и не се използва. Знакът за равенство в Тета-изразите е еднопосочен: Тета-нотацията може да се появява или само вдясно, което видяхме досега, или вдясно и вляво, което ще видим след малко, но никога само вляво. Примери за появата на Тета вляво и вдясно са изрази като

$$n^2 + \Theta(n) = \Theta(n^2)$$

В този случай Тета-нотациите вляво и вдясно имат съвсем различен смисъл. Тета-нотацията вдясно представлява безкрайно множество от функции, а тази вляво е *анонимна функция*. С други думи, Тета-нотацията вляво представлява не множество функции, а само една, като всичко, което знаем за нея е, че принадлежи на множеството $\Theta(n)$. Целият израз се чете така: сумата на n^2 и коя да е функция от множеството $\Theta(n)$ е функция от множеството $\Theta(n^2)$.

Допълнение 11: Друга интерпретация на появата на Θ вляво

Според Knuth [35], изрази като “ $n^2 + \Theta(n) = \Theta(n^2)$ ” се четат по следния начин. Нека A и B са множества от функции. Тогава $A+B$ е множеството от функции $\{f+g \mid f \in A, g \in B\}$. В примера “ $n^2 + \Theta(n) = \Theta(n^2)$ ” изразът вляво се интерпретира точно по този начин: нека $A = \{n^2\}$, а B е безкрайното множество от функции $\Theta(n)$ според Определение 3;

тогава “ $n^2 + \Theta(n)$ ” е множеството от функции, всяка от които е сума на n^2 и някоя функция от $\Theta(n)$.

Тогава знакът за равенство се интерпретира не като принадлежност към множество (множеството вдясно), а като знак за **подмножество**. При тази интерпретация, “ $n^2 + \Theta(n) = \Theta(n^2)$ ” се чете така “множеството от функции, всяка от които е сумата на n^2 и някоя функция от $\Theta(n)$, е подмножество на множеството $\Theta(n^2)$.”

Конвенция 3: В $\Theta()$ се слага най-простият израз

Формално погледнато, дали ще пишем

$$10\sqrt{7}n^2 + 15.5n\sqrt{n} + 35n = \Theta(n^2)$$

или

$$n^2 = \Theta(10\sqrt{7}n^2 + 15.5n\sqrt{n} + 35n)$$

е без значение, и двете твърдения са верни. На практика вторият запис **не се ползва**. В скобите на Тета-нотацията слагаме възможно най-простиия израз.

2.3.4 Други асимптотични нотации: О-голямо, Омега-голямо, о-малко, омега-малко

Както казахме, що се отнася до асимптотиката на нарастването, Тета-нотацията дава възможно най-подробната информация. Въпреки че Тета-нотацията е доста “хлабава”—примерно, $\frac{n^3}{1000}$ и $1000^{1000}n^3$ са Тета една от друга—понякога не можем да намерим дори Тета-оценката на някаква функция. За такива случаи има други нотации, по-слаби от Тета-та, които са все пак информативни за асимптотиката на нарастването.

Определение 5: $O(g(n)), \Omega(g(n)), o(g(n)), \omega(g(n))$

За всяка функция $g(n)$:

$$O(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \exists c > 0 \exists n_0 \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\} \quad (2.8)$$

$$\Omega(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \exists c > 0 \exists n_0 \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\} \quad (2.9)$$

$$o(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \forall c > 0 \exists n_0 \forall n \geq n_0 : 0 \leq f(n) < c \cdot g(n)\} \quad (2.10)$$

$$\omega(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \forall c > 0 \exists n_0 \forall n \geq n_0 : 0 \leq c \cdot g(n) < f(n)\} \quad (2.11)$$

Както и при Тета-нотацията, принадлежността към тези множества означаваме не с “ ϵ ”, а с “ $=$ ”, примерно пишем $f(n) = O(g(n))$, $h(n) = \omega(\phi(n))$ и така нататък.

Ако $f(n) = O(g(n))$, казваме, че $g(n)$ е *асимптотична горна граница* за $f(n)$. Примерно, вярно е, че:

$$n = O(n^2) \quad n^2 = O(n^2) \quad n^2 + 1000n + 10000 = O(n^2) \quad 1 = O(n^2)$$

Съответно, функцията n^2 е асимптотична горна граница за всяка от функциите n , n^2 , $n^2 + 1000n + 10000$ и 1. От друга страна обаче, $\frac{1}{1000000}n^3 \neq O(n^2)$. Съответно, функцията n^2 не е асимптотична горна граница за функцията $\frac{1}{1000000}n^3$.

Забележете съществената разлика между дефинициите на $O(g(n))$ и $o(g(n))$ (същото важи и при $\Omega(g(n))$ и $\omega(g(n))$). Ако $f(n) = O(g(n))$, то за **някаква** положителна константа c , $f(n)$ не надвишава $c \cdot g(n)$ [†]. Ако $f(n) = o(g(n))$, то за **всяка** положителна константа c , $f(n)$ “изостава” от $c \cdot g(n)$ [‡]. Неформално говорейки, нотацията o -малко казва, че можем да “отдалечим” $f(n)$ “надолу” от $g(n)$ на каквато си искаем мултипликативна константа, стига да разглеждаме достатъчно големи стойности на аргумента. Затова, ако $f(n) = o(g(n))$, казваме, че $g(n)$ е *строга асимптотична горна граница* за $f(n)$.

Аналогично, $\Omega()$ и $\omega()$ задават съответно *асимптотична добра граница* и *строга асимптотична добра граница*.

В Лема 3 и Лема 4, нотациите и от двете страни на равенствата означават множества (съгласно дефинициите), а знаците за равенство означават равенства между множества (а не принадлежност към множества).

Лема 3

За всяка функция $g(n)$, $O(g(n)) \cap \Omega(g(n)) = \Theta(g(n))$.

Доказателство: Да разгледаме произволна $f(n) \in O(g(n)) \cap \Omega(g(n))$. Съгласно Определение 5(2.8) и Определение 5(2.9), това е същото като

$$(\exists c' > 0 \exists n'_0 \forall n \geq n'_0 : f(n) \leq c' \cdot g(n)) \wedge (\exists c'' > 0 \exists n''_0 \forall n \geq n''_0 : c'' \cdot g(n) \leq f(n))$$

Но тогава очевидно съществуват положителни c' , c'' , такива че за всяко $n \geq \max\{n'_0, n''_0\}$ е изпълнено

$$c'' \cdot g(n) \leq f(n) \leq c' \cdot g(n)$$

Съгласно Определение 3, $f(n) \in \Theta(g(n))$. В обратната посока, да допуснем, че $f(n) \in \Theta(g(n))$. Тогава съгласно Определение 3:

$$\exists c_1, c_2 > 0 \exists n_0 \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Тривиално следва, че

$$(\exists c > 0 \exists n_0 \forall n \geq n_0 : f(n) \leq c \cdot g(n)) \wedge (\exists c > 0 \exists n_0 \forall n \geq n_0 : c \cdot g(n) \leq f(n))$$

Съгласно Определение 5, това е същото като

$$f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$$

□

Лема 4

За всяка функция $g(n)$:

$$o(g(n)) \cap \omega(g(n)) = \emptyset \tag{2.12}$$

$$O(g(n)) \cap \omega(g(n)) = \emptyset \tag{2.13}$$

$$o(g(n)) \cap \Omega(g(n)) = \emptyset \tag{2.14}$$

[†]И то не непременно навсякъде, а за всяко n от **някакво** n_0 **нататък**.

[‡]И то не непременно навсякъде, а за всяко n от **някакво** n_0 **нататък**.

Доказателство: Ще докажем (2.12). Да разгледаме произволна $f(n) \in o(g(n)) \cap \omega(g(n))$. Съгласно Определение 5(2.10) и Определение 5(2.11), това е същото като

$$(\forall c > 0 \exists n_0 \forall n \geq n_0 : f(n) < c \cdot g(n)) \wedge (\forall c > 0 \exists n_0 \forall n \geq n_0 : c \cdot g(n) < f(n))$$

Щом и двата израза започват със “за всяко c ”, то имаме право да вземем едно и също c' и в двата. Заключаваме, че има стойност на аргумента \tilde{n} , такава че за всяко $n > \tilde{n}$:

$$f(n) < c' \cdot g(n) \wedge f(n) > c' \cdot g(n)$$

Това очевидно е невъзможно.

Ще докажем (2.13). Да разгледаме произволна $f(n) \in O(g(n)) \cap \omega(g(n))$. Съгласно Определение 5(2.8) и Определение 5(2.11), това е същото като

$$(\exists c > 0 \exists n_0 \forall n \geq n_0 : f(n) \leq c \cdot g(n)) \wedge (\forall c > 0 \exists n_0 \forall n \geq n_0 : c \cdot g(n) < f(n))$$

Щом изразът вдясно съдържа $\forall c$, в частност той е истина за стойността на c , за която изразът вляво е истина. Заключаваме, че има $c > 0$ и има стойност на аргумента \tilde{n} , такава че за всяко $n > \tilde{n}$:

$$f(n) \leq c \cdot g(n) \wedge f(n) > c \cdot g(n)$$

Това очевидно е невъзможно.

Ще докажем (2.14). Да разгледаме произволна $f(n) \in o(g(n)) \cap \Omega(g(n))$. Съгласно Определение 5(2.10) и Определение 5(2.9), това е същото като

$$(\forall c > 0 \exists n_0 \forall n \geq n_0 : f(n) < c \cdot g(n)) \wedge (\exists c > 0 \exists n_0 \forall n \geq n_0 : c \cdot g(n) \leq f(n))$$

Щом изразът вляво съдържа $\forall c$, в частност той е истина за стойността на c , за която изразът вдясно е истина. Заключаваме, че има $c > 0$ и има стойност на аргумента \tilde{n} , такава че за всяко $n > \tilde{n}$:

$$f(n) > c \cdot g(n) \wedge f(n) \leq c \cdot g(n)$$

Това очевидно е невъзможно. □

Нотациите O и Ω са свързани. Също така и нотациите o и ω са свързани. Ето как.

Лема 5

За всички функции $f(n)$, $g(n)$:

$$f(n) = O(g(n)) \leftrightarrow g(n) = \Omega(f(n)) \tag{2.15}$$

$$f(n) = o(g(n)) \leftrightarrow g(n) = \omega(f(n)) \tag{2.16}$$

Доказателство: Ще докажем (2.15). Наистина, съгласно Определение 5(2.8), $f(n) = O(g(n))$ е същото като

$$\exists c > 0 \exists n_0 \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

Но $f(n) \leq c \cdot g(n)$ е същото като $\frac{1}{c}f(n) \leq g(n)$; тъй като $c > 0$, $\frac{1}{c}$ е дефинирано. Заключаваме, че

$$\exists c' > 0 \exists n_0 \forall n \geq n_0 : c' \cdot f(n) \leq g(n)$$

а именно за $c' = \frac{1}{c}$. Но това е същото като $g(n) = \Omega(f(n))$ съгласно Определение 5(2.9).

Ще докажем (2.16). Наистина, съгласно Определение 5(2.10), $f(n) = o(g(n))$ е същото като

$$\forall c > 0 \exists n_0 \forall n \geq n_0 : f(n) < c \cdot g(n)$$

Но $f(n) < c \cdot g(n)$ е същото като $\frac{1}{c}f(n) < g(n)$; тъй като $c > 0$, $\frac{1}{c}$ е дефинирано. Нещо повече, всяко положително число може да се представи като $\frac{1}{c}$, за някое положително c . Заключаваме, че

$$\forall c' > 0 \exists n_0 \forall n \geq n_0 : c' \cdot f(n) < g(n)$$

Но това е същото като $g(n) = \Omega(f(n))$ съгласно Определение 5(2.9). \square

Нотациите o и ω може да се дефинират чрез граници.

Лема 6

За всички функции $f(n)$, $g(n)$:

$$f(n) = o(g(n)) \leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = \omega(g(n)) \leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

Допълнение 12: Нотациите O и o в анализа

В математическия анализ нотациите O и o се ползват от 19 век, тоест, много преди теорията на алгоритмите. Да разгледаме разглеждането на функцията $\sin(x)$ в ред на Taylor в околността на точка 0:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \dots$$

Често срещан запис на това разглеждане е:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + O(x^7) \quad (2.17)$$

(2.17) е по-прецизен запис от

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!}$$

зашто “ \approx ” може да означава какво ли не. (2.17) казва, че $\sin(x)$ е точно равен на сумата на $x - \frac{x^3}{3!} + \frac{x^5}{5!}$ и някаква функция, която в околността на точка 0 намалява, в асимптотичния смисъл, като x^{7a} . Очевидно нотацията “ $O(x^7)$ ” в (2.17) има смисъл, различен от този на Определение 5(2.8). В (2.17) се има предвид не тенденция при неограничено нарастване на аргумента—като е в Определение 5—а тенденция, когато аргументът клони към определена стойност, в случая 0.

Записът “ $f(x) = O(g(x))$ при $x \rightarrow a$ ” е кратък запис за следното:

$$\exists c > 0 \exists \delta > 0 : |f(x)| \leq c g(x), \text{ ако } 0 < |x - a| < \delta$$

В (2.17) “ $x \rightarrow 0$ ” е изпуснато, но то се подразбира.

За нотацията Θ нещата са аналогични. Записът “ $f(x) = \Theta(g(x))$ при $x \rightarrow a$ ” е кратък запис за следното:

$$\forall c > 0 \exists \delta > 0 : |f(x)| \leq cg(x), \text{ ако } 0 < |x - a| < \delta$$

В заключение, нотациите Θ и Θ винаги изразяват тенденции за нарастване (или намаляване) на функции. Ако x е аргументът, тези тенденции са или при $x \rightarrow a$ за някакво фиксирано a , или при $x \rightarrow \infty$. При изследването на сложността на алгоритми се интересуваме само от тенденцията при $x \rightarrow \infty$, поради което асимптотичните нотации за целите на тези лекционни записи винаги са спрямо тенденцията на неограничено нарастване на аргумента.

^aТъй като аргументът е по-малък от единица, x^7 изразява намаляване, а не нарастване.

2.3.5 Релации \asymp , \leq , $<$, \geq и $>$

Определение 6: Релация \asymp над множеството на функциите

За всички функции $f(n)$ и $g(n)$, $f(n) \asymp g(n)$ тогава и само тогава, когато $f(n) = \Theta(g(n))$.

Нотацията, използваща символа “ \asymp ”, е алтернатива на нотацията, използваща $\Theta()$: заместо “ $f(n) = \Theta(g(n))$ ” можем да запишем “ $f(n) \asymp g(n)$ ” и обратно.

Ако ползваме записа с “ \asymp ” заместо “ Θ ”, необходимостта от Конвенция 2 изчезва по очевидни причини: в израза няма “ Θ ”. Също така изчезва и необходимостта от Конвенция 3, понеже двете страни на знака \asymp са равнопоставени. Еrgo, следните два записа са еднакво приемливи:

$$10\sqrt{7}n^2 + 15.5n\sqrt{n} + 35n \asymp n^2$$

$$n^2 \asymp 10\sqrt{7}n^2 + 15.5n\sqrt{n} + 35n$$

Определение 7: Релации \leq , $<$, \geq и $>$

За всички функции $f(n)$ и $g(n)$:

- $f(n) \leq g(n)$ тогава и само тогава, когато $f(n) = O(g(n))$,
- $f(n) < g(n)$ тогава и само тогава, когато $f(n) = o(g(n))$,
- $f(n) \geq g(n)$ тогава и само тогава, когато $f(n) = \Omega(g(n))$,
- $f(n) > g(n)$ тогава и само тогава, когато $f(n) = \omega(g(n))$.

Лема 7 се доказва тривиално с Лема 2 и Следствие 1.

Лема 7

\asymp е релация на еквивалентност. За всеки две функции $f(n)$ и $g(n)$, ако $f(n) \asymp g(n)$, казваме, че $f(n)$ и $g(n)$ са асимптотично еквивалентни.

Определение 8: Релации на асимптотични сравнения

Петте релации \asymp , \leq , $<$, \geq и $>$ са известни като *релациите на асимптотични сравнения*. За всеки две функции $f(n)$ и $g(n)$, ако поне една от тези пет релации е в сила между тях, казваме, че $f(n)$ и $g(n)$ са асимптотично сравними. В противен случай казваме, че $f(n)$ и $g(n)$ са асимптотично несравними.

От симетрията на \asymp и транспонираната симетрия (вж. Следствие 4) на $\{\leq, \geq\}$ и на $\{<, >\}$ следва, че по отношение на асимптотичната сравнимост няма значение коя от $f(n)$ и $g(n)$ е записана вляво и коя, вдясно. С други думи, ако за някоя от петте релации, да я запищем като \triangleright , е вярно, че $f(n) \triangleright g(n)$, то съществува релация измежду петте, да я запищем като \triangleleft , такава че $g(n) \triangleleft f(n)$.

Съгласно Теорема 5(2.30), асимптотично несравними функции съществуват.

Определение 9: Асимптотично по-бавно и асимптотично по-бързо нарастване

Ако $f(n) < g(n)$, казваме, че $f(n)$ расте асимптотично по-бавно от $g(n)$. Ако $\phi(n) > \psi(n)$, казваме, че $\phi(n)$ расте асимптотично по-бързо от $\psi(n)$.

Заради транспонираната симетрия (вж. Следствие 4) на $\{<, >\}$ имаме право освен това да кажем, че $g(n)$ расте асимптотично по-бързо от $f(n)$ и че $\psi(n)$ расте асимптотично по-бавно от $\phi(n)$.

Лема 3 и Определение 6 влекат непосредствено Следствие 2.

Следствие 2

За всички функции $f(n)$ и $g(n)$, $f(n) \asymp g(n) \leftrightarrow f(n) \leq g(n) \wedge f(n) \geq g(n)$.

Лема 4 и Определение 7 влекат непосредствено Следствие 3.

Следствие 3: Несъвместими релации на асимптотични сравнения

Не съществуват функции $f(n)$ и $g(n)$, такива че:

$$f(n) < g(n) \wedge f(n) > g(n)$$

$$f(n) \leq g(n) \wedge f(n) > g(n)$$

$$f(n) < g(n) \wedge f(n) \geq g(n)$$

Лема 5 и Определение 7 влекат непосредствено Следствие 4.

Следствие 4: Транспонирана симетрия

За всички функции $f(n)$, $g(n)$:

$$f(n) \leq g(n) \leftrightarrow g(n) \geq f(n)$$

$$f(n) < g(n) \leftrightarrow g(n) > f(n)$$

Лема 8 е точно съответствие на Лема 6:

Лема 8

За всички функции $f(n)$, $g(n)$:

$$f(n) < g(n) \leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) > g(n) \leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

Лема 9

За всеки две функции $f(n)$ и $g(n)$:

$$f(n) < g(n) \rightarrow f(n) \leq g(n) \quad (2.18)$$

$$f(n) > g(n) \rightarrow f(n) \geq g(n) \quad (2.19)$$

Доказателство: Доказателството е тривиално и в двата случая. За (2.18), ако за всяко положително c и всички достатъчно големи стойности на аргумента n е вярно, че $f(n) < c \cdot g(n)$, то очевидно съществува положително c , такова че за всички достатъчно големи стойности на аргумента n : $f(n) \leq c \cdot g(n)$. За (2.19) доказателството е аналогично. \square

Лема 10

Не е вярно, че за всеки две функции $f(n)$ и $g(n)$:

$$f(n) \leq g(n) \rightarrow f(n) < g(n) \quad (2.20)$$

$$f(n) \geq g(n) \rightarrow f(n) > g(n) \quad (2.21)$$

Доказателство: Като контрапример и за (2.20), и за (2.21) да разгледаме $f(n) = n$ и $g(n) = n$. Очевидно $f(n) \leq g(n)$ и $f(n) \geq g(n)$, но $f(n) \neq g(n)$ и $f(n) \neq g(n)$. \square

Строгите релации $<$ и $>$ са несъвместими съответно със \geq и \leq .

Лема 11

За всеки две функции $f(n)$ и $g(n)$:

$$f(n) \geq g(n) \rightarrow f(n) \neq g(n) \quad (2.22)$$

$$f(n) < g(n) \rightarrow f(n) \neq g(n) \quad (2.23)$$

$$f(n) \leq g(n) \rightarrow f(n) \neq g(n) \quad (2.24)$$

$$f(n) > g(n) \rightarrow f(n) \neq g(n) \quad (2.25)$$

Доказателство: Всяка от тези импликации се доказва тривиално с допускане на противното. Да разгледаме само (2.22). Противното е

$$\neg(f(n) \geq g(n) \rightarrow f(n) \neq g(n)) \equiv \quad (* \text{ свойства на импликацията } *)$$

$$\neg(\neg(f(n) \geq g(n)) \vee \neg(f(n) < g(n))) \equiv \quad (* \text{ з-н на De Morgan, з-н за дв. отриц. } *)$$

$$f(n) \geq g(n) \wedge f(n) < g(n)$$

Но от Следствие 3 знаем, че такива $f(n)$ и $g(n)$ няма. \square

Строгите релации $<$ и $>$ са несъвместими с асимптотичната еквивалентност \asymp .

Лема 12

За всеки две функции $f(n)$ и $g(n)$:

$$f(n) \asymp g(n) \rightarrow f(n) \prec g(n) \quad (2.26)$$

$$f(n) < g(n) \rightarrow f(n) \neq g(n) \quad (2.27)$$

$$f(n) \asymp g(n) \rightarrow f(n) \succ g(n) \quad (2.28)$$

$$f(n) > g(n) \rightarrow f(n) \neq g(n) \quad (2.29)$$

Доказателство: Следват от Лема 11 и Следствие 2. Да разгледаме само (2.26). От Следствие 2 имаме $f(n) \asymp g(n) \rightarrow f(n) \geq g(n)$. От Лема 11 имаме $f(n) \geq g(n) \rightarrow f(n) \prec g(n)$. Прилагаме правилото хипотетичен силогизъм от съждителната логика $p \rightarrow q \wedge q \rightarrow r \vdash p \rightarrow r$ и получаваме желания резултат. \square

Въведохме пет релации на асимптотични сравнения на функции. От най-общи комбинаторни съображения има не повече от $2^5 = 32$ възможности тези релации да са или да не са в сила за произволни функции. Теорема 5 казва, че тези възможности са точно шест.

Теорема 5

За всички функции $f(n)$ и $g(n)$ в сила е точно едно от следните:

$$f(n) \prec g(n) \wedge f(n) \not\leq g(n) \wedge f(n) \neq g(n) \wedge f(n) \succ g(n) \wedge f(n) \not\geq g(n) \quad (2.30)$$

$$f(n) \prec g(n) \wedge f(n) \leq g(n) \wedge f(n) \neq g(n) \wedge f(n) \succ g(n) \wedge f(n) \not\geq g(n) \quad (2.31)$$

$$f(n) < g(n) \wedge f(n) \leq g(n) \wedge f(n) \neq g(n) \wedge f(n) \succ g(n) \wedge f(n) \not\geq g(n) \quad (2.32)$$

$$f(n) \prec g(n) \wedge f(n) \leq g(n) \wedge f(n) \asymp g(n) \wedge f(n) \succ g(n) \wedge f(n) \geq g(n) \quad (2.33)$$

$$f(n) \prec g(n) \wedge f(n) \not\leq g(n) \wedge f(n) \neq g(n) \wedge f(n) \succ g(n) \wedge f(n) \geq g(n) \quad (2.34)$$

$$f(n) \prec g(n) \wedge f(n) \not\leq g(n) \wedge f(n) \neq g(n) \wedge f(n) > g(n) \wedge f(n) \geq g(n) \quad (2.35)$$

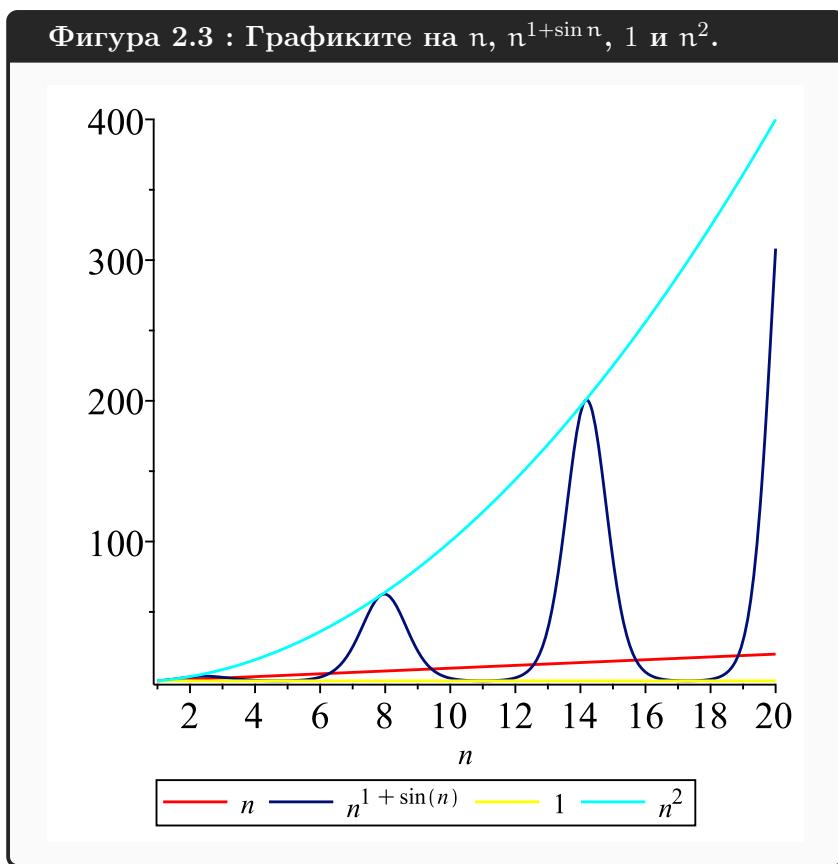
Доказателство: Първо ще покажем по една двойка функции като пример за всяка от тези шест възможности. После ще покажем, че останалите комбинации са невъзможни.

(2.30): Този случай се характеризира с това, че $f(n) \not\leq g(n)$ и $f(n) \not\geq g(n)$, понеже е достатъчно и \leq , и \geq да не са в сила, за да не е в сила нито една от останалите три релации. Това се доказва лесно с негацията на Следствие 2 и контрапозитивните на твърденията от Лема 9.

Следният пример за такива функции е от [15, стр. 52]. Нека $f(n) = n$ и $g(n) = n^{1+\sin n}$. Разгледайте Фигура 2.3[†], която илюстрира графиките на $f(n)$ и $g(n)$, както и на функциите 1 и n^2 . Тъй като $1 + \sin n$ осцилира между 0 и 2 , то $n^{1+\sin n}$ осцилира между 1 и n^2 [‡]. Ясно е, че за всяка константа $c > 0$ и за всяка стойност на аргумента n_0 съществува $n' > n_0$, такова че $f(n') < c \cdot g(n')$ и съществува $n'' > n_0$, такова че $f(n'') > c \cdot g(n'')$. Следователно, нито една от релациите \leq и \geq не е в сила.

[†]Фигура 2.3 е направена с Maple(TM).

[‡]В този случай осцилирането е в степенния показател, а не в множител, както беше в примера на Фигура 2.2, така че тези функции не са Тита една от друга, за разлика от функциите, показани на Фигура 2.2.



(2.31): Нека $f(n) = n^{1+\sin n}$ и $g(n) = n^2$ (Фигура 2.3 илюстрира и тези разсъждения). Вярно е, че $f(n) \leq g(n)$: да вземем $n_0 = 1$ и $c = 1$.

От друга страна, $f(n) \not\leq g(n)$, понеже съществува константа $c > 0$, такава че за всяка стойност на аргумента n_0 съществува $n' > n_0$, такова че $f(n') \geq c \cdot g(n')$, например, $c = 1$.

Също така е вярно, че $f(n) \not\geq g(n)$, понеже за всяка константа $c > 0$ и за всяка стойност на аргумента n_0 съществува $n' > n_0$, такова че $f(n') < c \cdot g(n')$.

От това, че $f(n) \leq g(n)$ и $f(n) \not\geq g(n)$ следва, че $f(n) \neq g(n)$ съгласно Следствие 2.

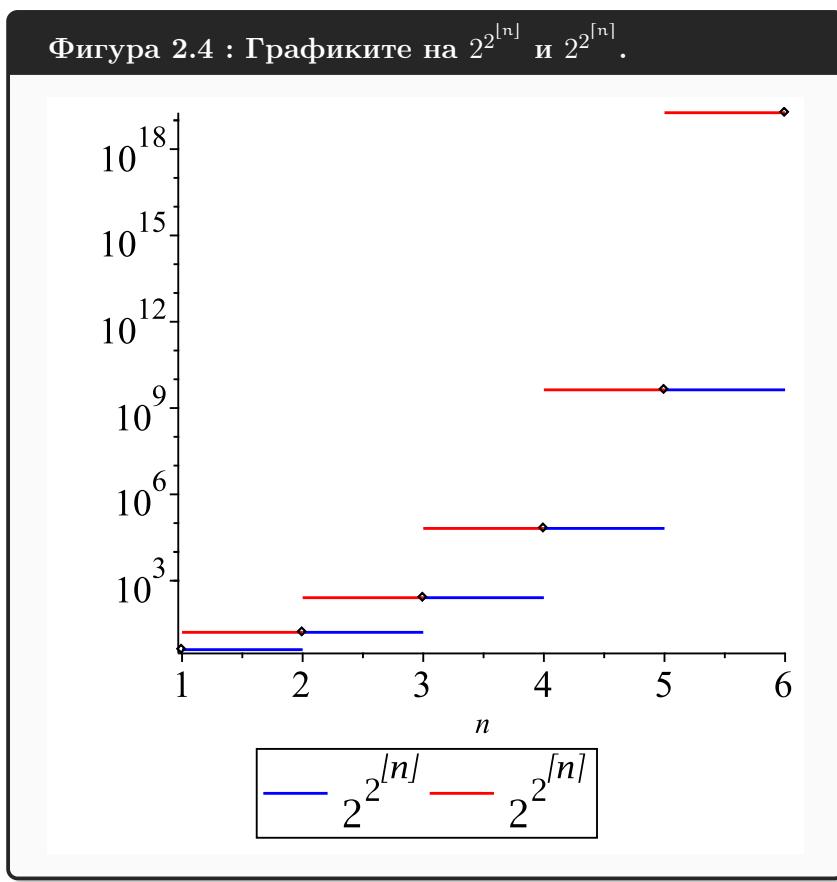
И накрая, $f(n) \not\asymp g(n)$, което следва от това, че $f(n) \leq g(n)$ и Лема 11.

Друга двойка функции, която е подходящ пример, е $f(n) = 2^{2^{\lfloor n \rfloor}}$ и $g(n) = 2^{2^{\lceil n \rceil}}$. Ако $n \in \mathbb{N}$, то $f(n) = g(n)$. Ако обаче $n \in \mathbb{R}^+ \setminus \mathbb{N}$, $f(n) = \sqrt{g(n)}$, понеже

$$n \in \mathbb{R}^+ \setminus \mathbb{N} \rightarrow \lfloor n \rfloor = \lceil n \rceil - 1 \rightarrow 2^{\lfloor n \rfloor} = 2^{\lceil n \rceil - 1} \rightarrow 2^{2^{\lfloor n \rfloor}} = 2^{2^{\lceil n \rceil - 1}} = 2^{\frac{1}{2}2^{\lceil n \rceil}} = \sqrt{2^{2^{\lceil n \rceil}}}$$

Фигура 2.4[†] показва графиките на тези две функции. Функциите се изравняват върху всяко цяло число. След това, в отворения интервал до следващото цяло число, $f(n) = 2^{2^{\lfloor n \rfloor}}$ остава същата, а $2^{2^{\lceil n \rceil}}$ взема стойност, която е предната ѝ стойност, повдигната на квадрат. Фигурата е с логаритмичен мащаб по ординатата поради изключително бързото нарастване на двойната експонента.

[†]Фигура 2.4 е направена с Maple(TM).



(2.32): Пример за такива функции са $f(n) = n$ и $g(n) = n^2$. Това, че $f(n) < g(n)$, $f(n) \leq g(n)$, $f(n) \neq g(n)$, $f(n) \succ g(n)$ и $f(n) \not\geq g(n)$, е очевидно.

(2.33): Пример за такива функции са $f(n) = n$ и $g(n) = n + 1$. Това, че $f(n) \neq g(n)$, $f(n) \leq g(n)$, $f(n) \asymp g(n)$, $f(n) \succ g(n)$ и $f(n) \geq g(n)$, е очевидно.

(2.34): Двата примера от (2.31), но с разменени $f(n)$ и $g(n)$, са валидни за този случай.

(2.35): Пример за такива функции са $f(n) = n^2$ и $g(n) = n$. □

Сега да разгледаме невъзможните комбинации. Всички комбинации с поне едно от следните:

$$f(n) > g(n) \wedge f(n) \not\geq g(n)$$

$$f(n) < g(n) \wedge f(n) \not\leq g(n)$$

са невъзможни, защото $>$ влече \geq и $<$ влече \leq (Лема 9). Всички комбинации с поне едно от следните:

$$f(n) \asymp g(n) \wedge f(n) \not\geq g(n)$$

$$f(n) \not\leq g(n) \wedge f(n) \asymp g(n)$$

$$f(n) \leq g(n) \wedge f(n) \neq g(n) \wedge f(n) \geq g(n)$$

са невъзможни, защото \asymp е в сила тогава и само тогава, когато и \geq , и \leq са в сила (Следствие 2). И накрая, всички комбинации с поне едно от следните:

$$f(n) < g(n) \wedge f(n) \geq g(n)$$

$$f(n) \leq g(n) \wedge f(n) > g(n)$$

са невъзможни, защото $<$ и \geq са несъвместими, а също така \leq и $>$ са несъвместими (Следствие 3). □

Теорема 6

За всеки две функции $f(n)$ и $g(n)$ и за всяка константа $k \in \mathbb{R}^+$:

$$f(n) \asymp g(n) \leftrightarrow (f(n))^k \asymp (g(n))^k$$

Доказателство: В едната посока, нека

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

за някакви положителни константи c_1 и c_2 и за всички $n \geq n_0$, за някакво $n_0 > 0$. Повдигаме двете неравенства на k -та степен (k е положително) и получаваме

$$c_1^k (g(n))^k \leq (f(n))^k \leq c_2^k (g(n))^k, \text{ за всяко } n \geq n_0$$

Но тогава $(f(n))^k \asymp (g(n))^k$, понеже c_1^k и c_2^k са положителни константи.

В другата посока, нека

$$c_1 (g(n))^k \leq (f(n))^k \leq c_2 (g(n))^k, \text{ за всяко } n \geq n_0$$

за някакви положителни константи c_1 и c_2 и за всички $n \geq n_0$, за някакво $n_0 > 0$. Повдигаме двете неравенства на $\frac{1}{k}$ -та степен ($\frac{1}{k}$ е положително) и получаваме

$$\sqrt[k]{c_1} g(n) \leq f(n) \leq \sqrt[k]{c_2} g(n), \text{ за всяко } n \geq n_0$$

Но тогава $f(n) \asymp g(n)$, понеже $\sqrt[k]{c_1}$ и $\sqrt[k]{c_2}$ са положителни константи. □

Теорема 7

Нека $f(n) = g(n) + h(n)$, където $h(n) < f(n)$. Тогава $f(n) \asymp g(n)$.

Доказателство: Ще покажем, че

$$\exists c_1 \exists n' \forall n \geq n' : c_1 \cdot g(n) \leq f(n) \quad (2.36)$$

$$\exists c_2 \exists n'' \forall n \geq n'' : f(n) \leq c_2 \cdot g(n) \quad (2.37)$$

Да вземем $c_1 = 1$ и $n_1 = 1$. Тъй като разглеждаме само положителни функции, то $f(n)$, $g(n)$ и $h(n)$ са положителни. Тогава $f(n) > g(n) \rightarrow f(n) \geq 1 \cdot g(n)$, за всички n . Доказваме (2.36).

Разглеждаме (2.37). Използваме това, че $h(n) < f(n)$, което е кратък запис за

$$\forall c > 0 \exists n_0 \forall n > n_0 : h(n) < cf(n)$$

В частност, вярно е, че $h(n) < \frac{1}{2}f(n)$ за всички n , надхвърлящи някакво n_0 . Но, тъй като $g(n) = f(n) - h(n)$, то $g(n) \geq \frac{1}{2}f(n)$ за всички n , надхвърлящи някакво n_0 . С други думи, $f(n) \leq 2g(n)$ за всички n , надхвърлящи някакво n_0 . Тогава за нашето доказателство вземаме $c_2 = 2$ и $n'' = n_0$. Доказваме (2.37). □

С Теорема 7 веднага доказваме, че всеки полином с положителни коефициенти[†] има асимптотика, зависеща само от степента му.

[†]Да настояваме за положителни коефициенти е прекалено рестриктивно – достатъчно е коефициентът пред n^k да е положителен. Но, ако допуснем само това, може да имаме формални проблеми, тъй като останалата част на полинома може да е отрицателна функция и тогава, чисто формално, нямаме право да ползваме асимптотичните релации при текущите дефиниции.

Следствие 5

Нека $p(n)$ е произволен полином от степен k с положителни коефициенти. Тогава $p(n) \asymp n^k$.

Доказателство: Нека $p(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0$. Но $a_{k-1} n^{k-1} + \dots + a_1 n + a_0 < p(n)$, понеже $\lim_{n \rightarrow \infty} \frac{a_{k-1} n^{k-1} + \dots + a_1 n + a_0}{a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0} = 0$. Сега прилагаме Теорема 7. \square

Теорема 8 казва, че експоненциалната трансформация върху две функции “изостря” асимптотичните разлики в нарастването им, ако функциите са неограничено нарастващи и имат различно асимптотично нарастване.

Теорема 8

Нека $f(n)$ и $g(n)$ са растящи и неограничени функции и нека a е константа, по-голяма от 1. Тогава

$$f(n) < g(n) \rightarrow a^{f(n)} < a^{g(n)}$$

Доказателство: Трябва да докажем, че:

$$\forall c > 0 \exists n' \forall n \geq n': 0 \leq a^{f(n)} < c \cdot a^{g(n)} \quad (2.38)$$

Да разгледаме $k = \log_a c$. Тоест, $c = a^k$. Да препишем (2.38) така:

$$\forall k \exists n' \forall n \geq n': 0 \leq a^{f(n)} < a^k a^{g(n)} \quad (2.39)$$

Вземаме логаритъм при основа a от двете неравенства и получаваме:

$$\forall k \exists n' \forall n \geq n': 0 \leq f(n) < k + g(n) \quad (2.40)$$

Трябва да докажем (2.40). По условие имаме:

$$\forall c > 0 \exists n_0 \forall n \geq n_0 : 0 \leq f(n) < c \cdot g(n)$$

Щом това е в сила за всяко $c > 0$, в частност то е в сила за $c = \frac{1}{2}$. И така:

$$\exists n_0 \forall n \geq n_0 : 0 \leq f(n) < \frac{g(n)}{2} \quad (2.41)$$

Но $g(n)$ е растяща и неограничена. Тогава:

$$\forall k \exists n_1 \forall n \geq n_1 : 0 < k + \frac{g(n)}{2} \quad (2.42)$$

Да препишем (2.42) така:

$$\forall k \exists n_1 \forall n \geq n_1 : \frac{g(n)}{2} < k + g(n) \quad (2.43)$$

От (2.41) и (2.43) заключаваме, че:

$$\forall k \exists n'' \forall n \geq n'' : 0 \leq f(n) < k + g(n) \quad (2.44)$$

Но (2.44) е същото като (2.40). С това доказателството е готово. \square

Следствие 6

Нека $f(n)$ и $g(n)$ са растящи и неограничени функции. Тогава

$$\lg f(n) < \lg g(n) \rightarrow f(n) < g(n)$$

Теорема 9 в някакъв смисъл е конверсна на Теорема 8. Според Теорема 9, ако образите на две нарастващи и неограничени функции след експоненциална трансформация са асимптотично еквивалентни, то и самите функции са асимптотично еквивалентни.

Теорема 9

Нека $f(n)$ и $g(n)$ са растящи и неограничени функции и нека a е константа, по-голяма от 1. Тогава

$$a^{f(n)} \asymp a^{g(n)} \rightarrow f(n) \asymp g(n)$$

Доказателство: Нека

$$\exists c_1 > 0 \exists c_2 > 0 \exists n_0 > 0 \forall n > n_0 : c_1 \cdot a^{g(n)} \leq a^{f(n)} \leq c_2 \cdot a^{g(n)}$$

Щом $a > 1$ и $f(n)$ и $g(n)$ са положителни, то за всички достатъчно големи n е вярно, че експонентите са строго по-големи от единица. Тогава вземаме логаритъм при основа a от двете страни и получаваме:

$$\log_a c_1 + g(n) \leq f(n) \leq \log_a c_2 + g(n)$$

Забелязваме, че щом $g(n)$ е нарастваща и неограничена, то за всяка константа k_1 , такава че $0 < k_1 < 1$, в сила е $k_1 \cdot g(n) \leq \log_a c_1 + g(n)$, за всички достатъчно големи n . И това е така независимо от това, дали логаритъмът е положителен или отрицателен или е нула. После забелязваме, че щом $g(n)$ е нарастваща и неограничена, то за всяка константа k_2 , такава че $k_2 > 1$, в сила е $\log_a c_2 + g(n) \leq k_2 \cdot g(n)$ за всички достатъчно големи n . И това е така независимо от това, дали логаритъмът е положителен или отрицателен или е нула.

Заключаваме, че съществува n_1 , такова че:

$$\forall n \geq n_1 : k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$$

Но тогава $f(n) \asymp g(n)$. □

Следствие 7

Нека $f(n)$ и $g(n)$ са растящи и неограничени функции. Тогава

$$f(n) \asymp g(n) \rightarrow \lg f(n) \asymp \lg g(n)$$

Определение 10: Полилогаритмично растяща функция

За всяка $f(n)$, такава че $f(n) \asymp (\log_a n)^k$, където $a, k \in \mathbb{R}^+$ са константи и $a > 1$, казваме, че $f(n)$ *расте полилогаритмично*.

Определение 11: Полиномиално растяща функция

За всяка $f(n)$, такава че $f(n) \asymp n^\epsilon$, където $\epsilon \in \mathbb{R}^+$ е константа, казваме, че $f(n)$ *расте полиномиално*.

Забележете, че в математиката е прието полиномите да имат цели степенни показатели, така че функцията $f(n) = n^{\frac{1}{2}}$ не е полиномиална. Но съгласно Определение 11, $f(n) = n^{\frac{1}{2}}$ също расте полиномиално.

Определение 12: Експоненциално растяща функция

За всяка $f(n)$, такава че $f(n) \asymp a^{n^k}$, където $a, k \in \mathbb{R}^+$ са константи и $a > 1$, казваме, че $f(n)$ *расте експоненциално*.

Когато чуем “експоненциална функция”, обикновено си представяме 2^n , e^n и изобщо a^n за някое $a > 1$. Съгласно Определение 12, функции като 2^{n^2} и 3^{n^5} и $(\sqrt{2})^{n^{\sqrt{2}}}$ също са експоненциално растящи функции.

Теорема 10

Нека $a, k, \epsilon \in \mathbb{R}^+$ са произволни константи, като $a > 1$. В сила е:

$$(\log_a n)^k < n^\epsilon$$

Доказателство:

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{n^\epsilon}{(\log_a n)^k} &= \quad // \text{полагаме } b \leftarrow \frac{\epsilon}{k} \\ \lim_{n \rightarrow \infty} \frac{(n^b)^k}{(\log_a n)^k} &= \\ \lim_{n \rightarrow \infty} \left(\frac{n^b}{\log_a n} \right)^k &= \quad // k \text{ е положително} \\ \lim_{n \rightarrow \infty} \frac{n^b}{\log_a n} &= \quad // \text{прилагаме правилото на l'Hôpital} \\ \lim_{n \rightarrow \infty} \frac{bn^{b-1}}{\left(\frac{1}{\ln a} \right) \left(\frac{1}{n} \right)} &= \\ \lim_{n \rightarrow \infty} (\ln a) b n^b &= \infty \end{aligned}$$

□

Теорема 11

Нека $a, \epsilon \in \mathbb{R}^+$ са произволни константи, като $a > 1$. В сила е:

$$n^\epsilon < a^n$$

Доказателство: Да вземем \log_a от двете страни. Лявата страна става $\epsilon \cdot \log_a n$ и дясната страна става n . Но $\epsilon \cdot \log_a n < n$ съгласно Теорема 10. Тогава $\epsilon \cdot \log_a n < n$. Прилагаме Следствие 6 и получаваме желания резултат. □

Следствие 8

Нека $a, \epsilon, k \in \mathbb{R}^+$ са произволни константи, като $a > 1$. В сила е:

$$n^\epsilon < a^{n^k}$$

От Теорема 10 и Следствие 8 веднага заключаваме следното.

Следствие 9

Всяка полилогаритмично растяща функция расте асимптотично по-бавно от произволна полиномиално растяща функция, а всяка полиномиално растяща функция расте асимптотично по-бавно от произволна експоненциално растяща функция.

2.3.6 Релациите \leq и \geq като преднаредби

Както видяхме в Лема 7, \asymp е релация на еквивалентност, подобно на релацията на равенство върху числата. Да разгледаме \leq и \geq . Те не са антисиметрични, защото може за различни функции $f(n)$ и $g(n)$ да е изпълнено $f(n) \leq g(n)$ и $g(n) \leq f(n)$, примерно $n^2 \leq n^2 + n$ и $n^2 + n \leq n^2$. Следователно \leq и \geq не са релации на частична наредба, а са по-общите **релации на преднаредба**. Освен това, както вече видяхме в доказателството на Теорема 5(2.30), \leq и \geq не са пълни, защото може за различни функции $f(n)$ и $g(n)$ да не е изпълнено нито $f(n) \leq g(n)$, нито $g(n) \leq f(n)$. Следният допълнителен материал съдържа достатъчно, за да бъде осмислено това.

Допълнение 13: За преднаредбите

В това допълнение ще обясним в детайли понятието *преднаредба*, което не се изучава в курса по Дискретна Математика, но е важно за осмислянето на релациите \leq и \geq . Да си припомним някои дефиниции за релациите по принцип. Сравнете това изложение с изложението в [53, стр. 49–50].

Определение 13: Видови релации

Нека A е произволно множество и $R \subseteq A \times A$ е някаква бинарна релация над него. Казваме, че:

- R е *рефлексивна*, ако $\forall a \in A : aRa$.
- R е *ирефлексивна*, още наречена *стриктна*, ако $\forall a \in A : \neg aRa$.
- R е *симетрична*, ако $\forall a, b \in A : aRb \rightarrow bRa$.
- R е *антисиметрична*, ако $\forall a, b \in A : aRb \wedge bRa \rightarrow a = b$.
- R е *пълна*, ако $\forall a, b \in A : a \neq b \rightarrow aRb \vee bRa$.
- R е *транзитивна*, ако $\forall a, b, c \in A : aRb \wedge bRc \rightarrow aRc$.
- R е *преднаредба*, още наричана *квази-наредба*^a, ако е рефлексивна и транзитивна.
- R е *строга преднаредба*, ако е ирефлексивна и транзитивна.
- R е *частична наредба*, ако е антисиметрична преднаредба.
- R е *линейна наредба*, ако е пълна частична наредба.
- R е *релация на еквивалентност*, ако е симетрична преднаредба.

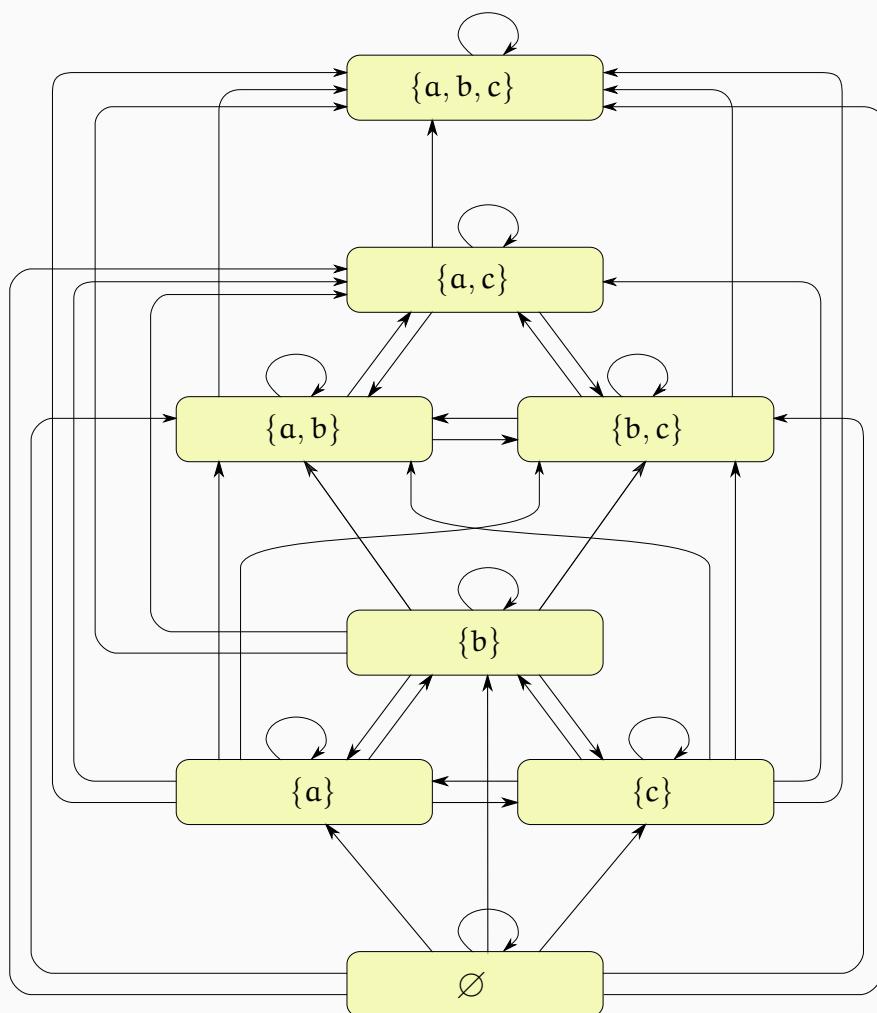
Казваме, че R' е *рефлексивното затваряне на R*, ако R' е най-малката релация, такава че $R \subseteq R'$ и R' е рефлексивна. Казваме, че R' е *симетричното затваряне на R*, ако R' е най-малката релация, такава че $R \subseteq R'$ и R' е симетрична. Казваме, че R' е *транзитивното затваряне на R*, ако R' е най-малката релация, такава че $R \subseteq R'$ и R' е транзитивна.

^aНа английски, *preorder* или *quasi-order*.

Пример за “истинска” преднаредба. Да разгледаме един пример за преднаредба, която не е нито релация на еквивалентност, нито на частична наредба; тоест, “истинска” преднаредба. Нека S е крайно множество. Нека $\mathcal{R}' \subseteq 2^S \times 2^S$ е дефинирана така:

$$\forall X \in 2^S \quad \forall Y \in 2^S : (X, Y) \in \mathcal{R}' \leftrightarrow |X| \leq |Y| \quad (2.45)$$

Очевидно \mathcal{R}' е рефлексивна и транзитивна – това следва от свойствата на релацията \leq върху естествените числа. Забележете, че \mathcal{R}' не е нито симетрична, нито антисиметрична. Нека $S = \{a, b, c\}$. Тъй като, примерно, $(\{a, b\}, \{a, b, c\}) \in \mathcal{R}'$ и $(\{a, b, c\}, \{a, b\}) \notin \mathcal{R}'$, то \mathcal{R}' не е симетрична. От друга страна, тъй като, примерно, $(\{a, b\}, \{b, c\}) \in \mathcal{R}'$ и $(\{b, c\}, \{a, b\}) \in \mathcal{R}'$, то \mathcal{R}' не е антисиметрична. Така че \mathcal{R}' е “истинска” преднаредба. Диаграмата на \mathcal{R}' при $S = \{a, b, c\}$ е показана на Фигура 2.5. Забележете, че \mathcal{R}' е и пълна релация, защото за всеки две подмножества на S , поне за едното е вярно, че има мощност поне колкото другото.

Фигура 2.5 : Диаграма на преднаредбата \mathcal{R}' от (2.45).

Следното определение и лема са Lemma 1 в книгата на Birkhoff [11, стр. 21].

Определение 14: Фактор-релация

Нека A е произволно множество и $R \subseteq A \times A$ е някаква преднаредба над него. Нека $\simeq \subseteq A \times A$ ^a е следната релация: $\forall a, b \in A : a \simeq b \leftrightarrow aRb \wedge bRa$. Очевидно \simeq е релация на еквивалентност над A . При това, ако E и F са два класа на еквивалентност на \simeq , тогава или $\forall x \in E \forall y \in F : xRy$, или $\forall x \in E \forall y \in F : \neg xRy$. Нека \mathfrak{X} е множеството от класовете на еквивалентност на \simeq . *Фактор-релацията спрямо R* ^b е релацията $\mathfrak{S} \subseteq \mathfrak{X} \times \mathfrak{X}$, дефинирана така:

$$\forall P, Q \in \mathfrak{X} : (P, Q) \in \mathfrak{S} \leftrightarrow \exists a \in P \exists b \in Q : aRb$$

^aОригиналният символ е \sim , а не \simeq . Тук ползваме \simeq , за да няма объркване с релацията за асимптотична еквивалентност от Подсекция 2.3.11.

^bВ книгата на Birkhoff се използва терминът *quotient set*.

Лема 13

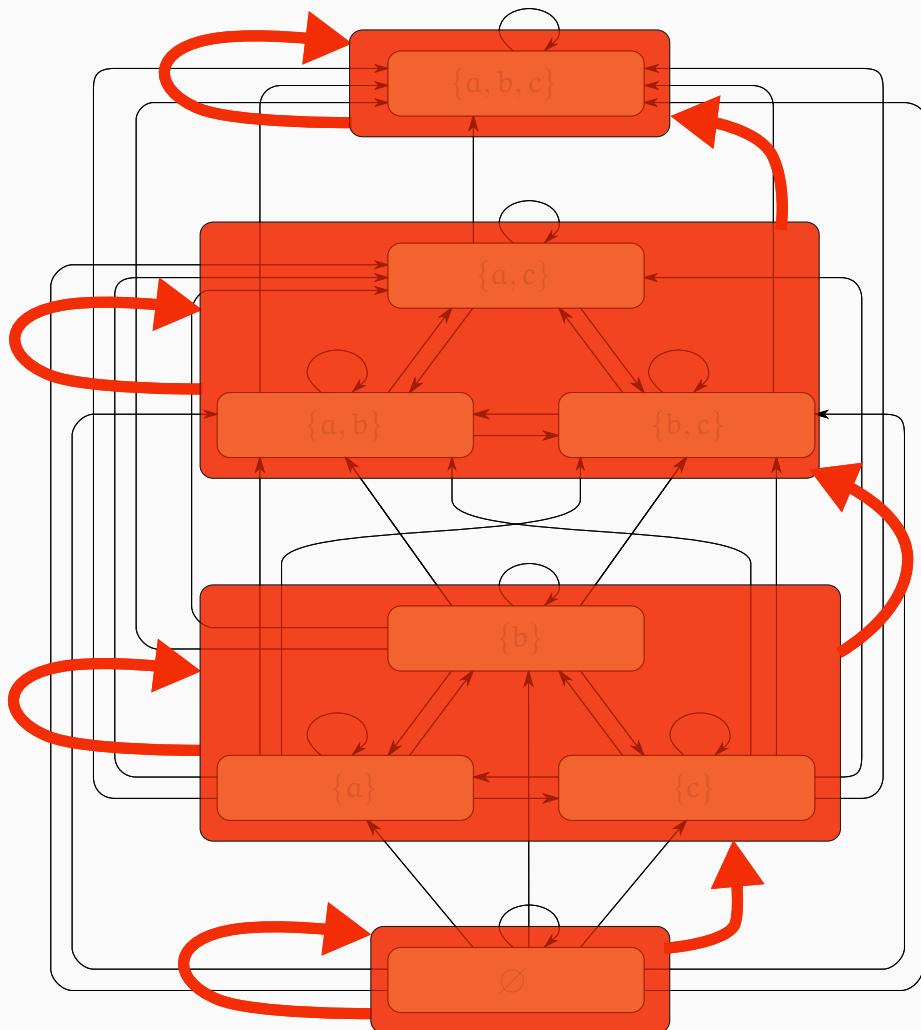
Фактор-релацията е релация на частична наредба.

Нотация 1

Означаваме фактор-релацията с R/\simeq .

Лесно се вижда, че всяка преднаредба $R \subseteq A \times A$ е пълна тогава и само тогава, когато R/\simeq е линейна наредба (върху класовете на еквивалентност на \simeq). Примерно, релацията R' , дефинирана в (2.45) е пълна преднаредба. Фигура 2.6 показва R'/\simeq , която очевидно е линейна.

Фигура 2.6 : Фактор-релацията R'/\simeq е линейна. Класовете на еквивалентност на \simeq са очертани с полуупрозрачни червени кутии, а диаграмата на R'/\simeq е нарисувана върху тях с червени стрелки.



Пример за преднаредба, която не е пълна. Нека B е следното множество от тримерни вектори от естествени числа:

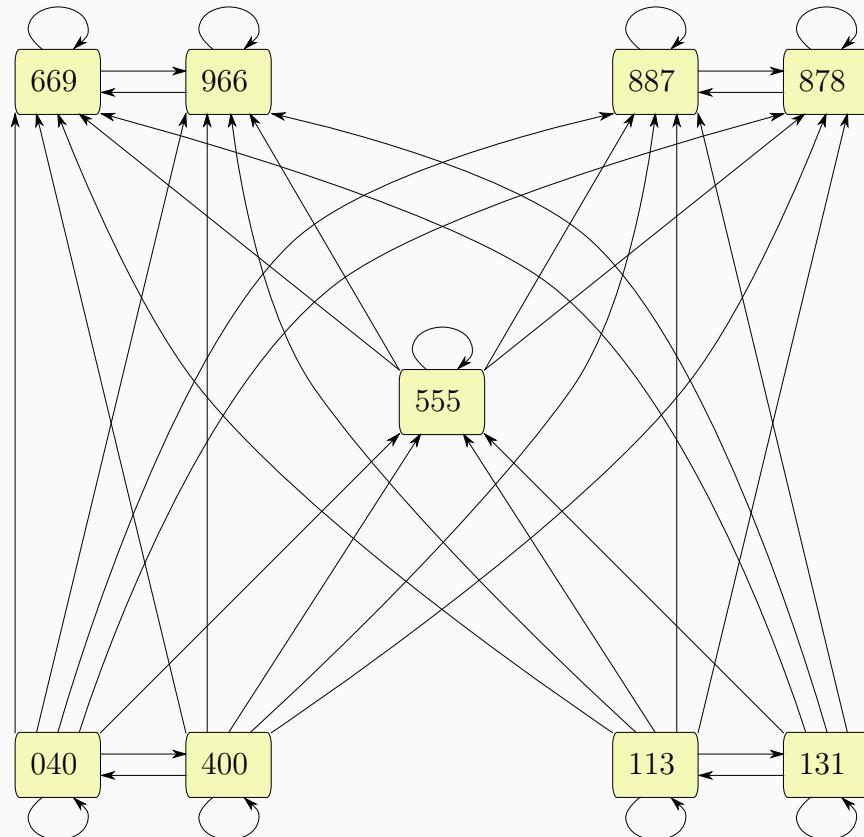
$$B = \{040, 400, 113, 131, 555, 779, 977, 886, 868\}$$

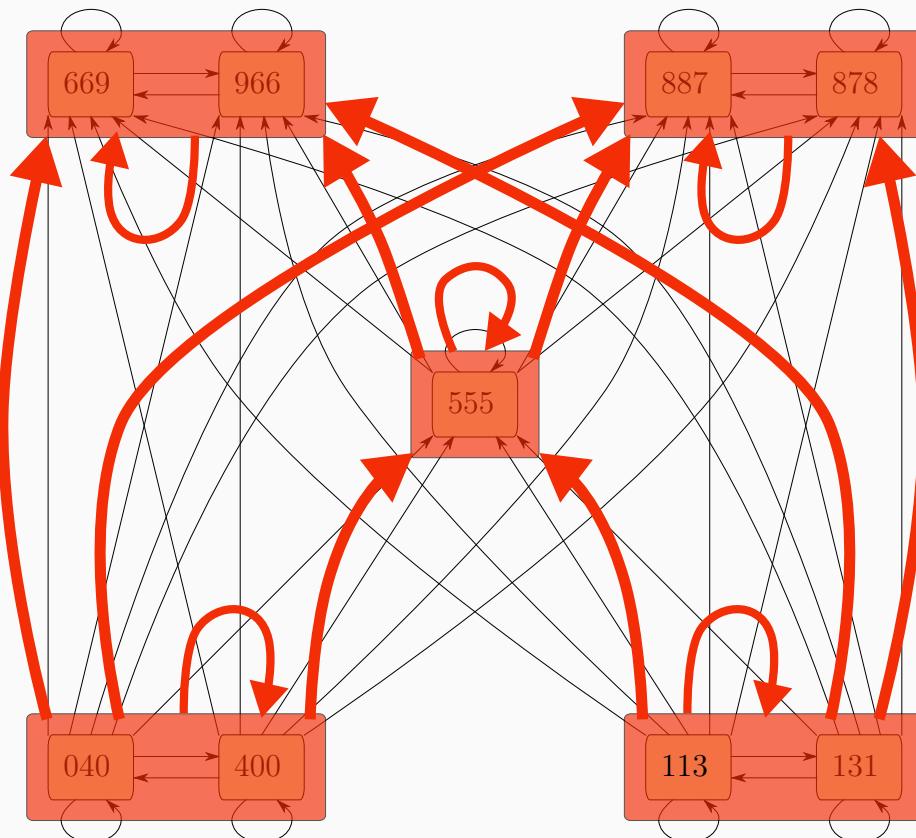
Нека Ω е следната релация над B :

$$\forall \mathbf{x}, \mathbf{y} \in B : (\mathbf{x}, \mathbf{y}) \in \Omega \leftrightarrow \sum \mathbf{x} = \sum \mathbf{y} \text{ или } (x_i \leq y_i, \text{ за } i \in \{1, 2, 3\})$$

където $\sum \mathbf{x}$ означава сумата от елементите на вектора \mathbf{x} (аналогично и за \mathbf{y}), а x_i е i -ият елемент на \mathbf{x} (аналогично и за \mathbf{y}). Диаграмата на Ω е показана на Фигура 2.7. Фигура 2.8 показва фактор-релацията.

Фигура 2.7 : Преднаредбата Ω не е пълна.



Фигура 2.8 : Фактор-релацията Q/\simeq не е линейна.

Наблюдение 5

Релациите $<$ и $>$ са ирефлексивни и транзитивни, а \leq и \geq са рефлексивни и транзитивни. Тогава $<$ и $>$ са строги преднаредби, а \leq и \geq са преднаредби върху множеството на функциите.

Преднаредбите \leq и \geq не са антисиметрични.

Преднаредбите \leq и \geq не са пълни.

Тъй като \simeq е релация на еквивалентност, имаме право да говорим за фактор-релациите \leq/\simeq и \geq/\simeq . Техните класове на еквивалентност са (неизброимо безкрайните, неизбройно безкрайно на брой) максимални по включване подмножества от функции, които са Тита една от друга.

Фактор-релациите \leq/\simeq и \geq/\simeq са “истински” частични наредби, в които има несравними елементи.

2.3.7 Релациите на асимптотични сравнения при $n \in \mathbb{N}^+$

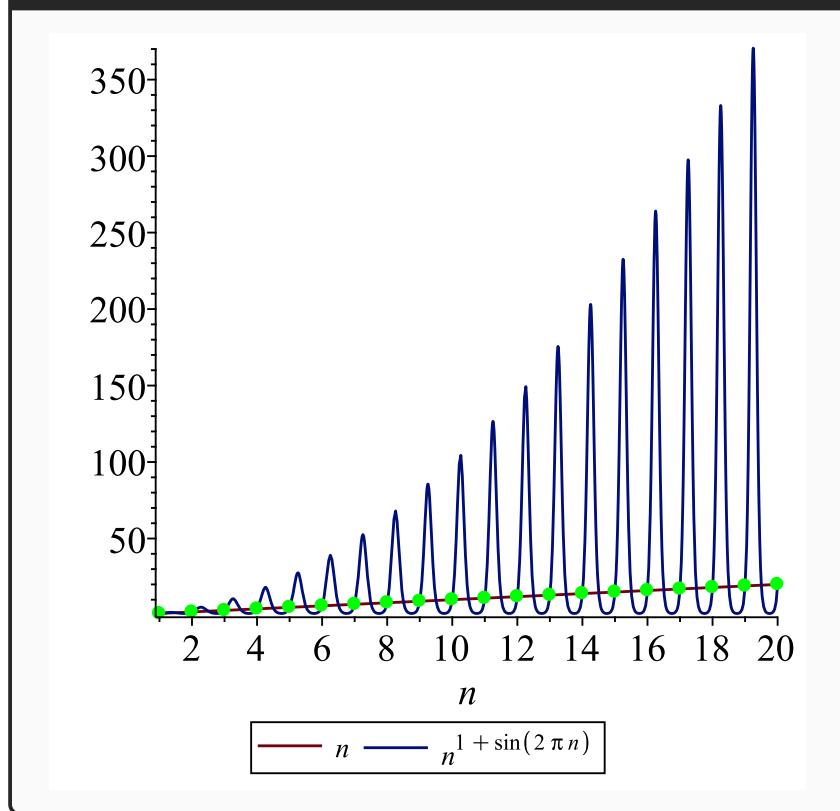
Вече приехме, че разглеждаме функции с домейн \mathbb{R}^+ (вж. Конвенция 1). Дали нещо ще се промени, ако преминем към домейн \mathbb{N}^+ ? От гледна точка на алгоритмичния анализ, функциите са с домейн именно \mathbb{N}^+ , понеже големината на входа е цяло положително число.

При “добре държащи се” функции като n , n^2 , 2^n и така нататък, нищо не се променя при преминаване от домейн \mathbb{R}^+ към домейн \mathbb{N}^+ . Примерно, $n^2 \asymp n^2 + n$, $2^n > n$, $n2^n < 3^n$ и ако ги разглеждаме върху \mathbb{R}^+ , и ако ги разглеждаме върху \mathbb{N}^+ .

Дори функцията да може да връща не-цяло число при целочислен аргумент, примерно $\frac{n}{5}$, \sqrt{n} или $\log_2 n$, същото остава в сила, стига функцията да е “добре държаща се”. Посочените функции са именно такива, така че, примерно, $\frac{n}{5} \geq \sqrt{n}$ е изпълнено и върху \mathbb{R}^+ , и върху \mathbb{N}^+ .

Има обаче “лошо държащи се” функции, при които резултатът от асимптотичното сравнение е чувствителен към избора на домейн (\mathbb{N}^+ или \mathbb{R}^+). Лесно е да конструираме пример за такава функция. Да си припомним функцията $g(n) = n^{1+\sin n}$ на стр. 53. Както стана ясно там, $f(n) = n$ и $g(n) = n^{1+\sin n}$ са асимптотично несравними. Напълно аналогично, функциите n и $n^{1+\sin 2\pi n}$ са асимптотично несравними, когато $n \in \mathbb{R}^+$, понеже $1 + \sin 2\pi n$ осцилира между 0 и 2, също както $1 + \sin n$. Да разгледаме n и $n^{1+\sin 2\pi n}$, когато n е цяло число. Функцията n , която е “добре държаща се”, има същото поведение при $n \in \mathbb{N}^+$. От друга страна, $n^{1+\sin 2\pi n} = n$ сега се държи по съвсем различен начин, понеже $\forall n \in \mathbb{N}^+ : \sin 2\pi n = 0$. Излиза, че върху естествените числа, функциите n и $n^{1+\sin 2\pi n}$ съвпадат. В сила е $n \asymp n^{1+\sin 2\pi n}$, ако $n \in \mathbb{N}^+$. Фигура 2.9 илюстрира всичко това – графиките съвпадат точно върху целите стойности на аргумента.

Фигура 2.9 : Графиките на n и $n^{1+\sin 2\pi n}$.



Друга двойка функции, асимптотичното сравнение на които се променя при целочислен аргумент, е $f(n) = 2^{2^{\lfloor n \rfloor}}$ и $g(n) = 2^{2^{\lceil n \rceil}}$ на стр. 54. Както видяхме там, те съвпадат при $n \in \mathbb{N}^+$. Ако домейнът е \mathbb{N}^+ , може да даваме примери за осцилиращи функции не чрез използване на тригонометрични функции, а просто чрез различно дефиниране върху четните и нечетните

числа. Примерно, следните две функции са асимптотично несравними:

$$f(n) = n^2$$

$$g(n) = \begin{cases} n, & \text{ако } n \text{ е четно} \\ n^3, & \text{ако } n \text{ е нечетно} \end{cases}$$

2.3.8 Асимптотични еквивалентности на често срещани функции

Допълнение 14: Апроксимация на Stirling

В сила е следният важен и нетривиален резултат за асимптотиката на факториела. Ползваме релацията \sim , дефинирана в Подсекция 2.3.11.

Теорема 12: Апроксимация на Stirling

$$n! \sim \sqrt{2\pi n} \frac{n^n}{e^n}$$

Доказателства на Теорема 12 има в статията на Dan Romik [54], както и в тази експозиционна статия на Keith Conrad, но тези доказателства изискват задълбочени познания по анализ и тук няма да ги разглеждаме.

Експозиционната статия на Byron Schmuland [56] съдържа резултат, по-слаб от Теорема 12—тя не показва, че мултипликативната константа е именно $\sqrt{2\pi}$ —но с по-леко доказателство. Ако искаме само да се убедим, че $n! \asymp \sqrt{n} \frac{n^n}{e^n}$, то тази статия е напълно достатъчна. Ето нейният основен резултат.

Теорема 13: Апроксимация на Stirling, по-слаба версия

Съществува константа c , такава че:

$$c\sqrt{n} \left(\frac{n}{e}\right)^n \leq n! \leq c\sqrt{n} \left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}$$

Доказателство: Да разгледаме естествения логаритъм на факториела:

$$\ln n! = \sum_{i=1}^n \ln i$$

Да умножим всеки суманд с единица 1:

$$\sum_{i=1}^n \ln i = \sum_{i=1}^n ((\ln i) \times 1)$$

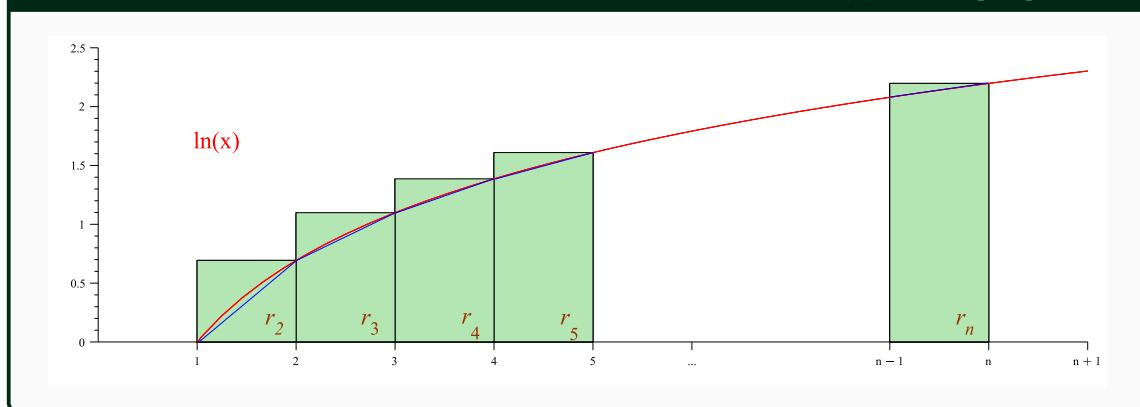
Тогава

$$\ln n! = \sum_{i=1}^n ((\ln i) \times 1)$$

Тогава $\ln n!$ е точно сумата от лицата на n на брой правоъгълници $r_1, r_2, r_3, \dots, r_n$, показани на Фигура 2.10^a, като r_1 е изроден правоъгълник с лице 0. На фигурата е

показана и графиката на функцията $\ln(x)$ и ясно се вижда, че сумата от лицата на тези правоъгълници е голямата сума на Darboux за $\ln(x)$, където x е в интервала $[0, n]$ с точки на разбиване $1, 2, \dots, n - 1$.

Фигура 2.10 : $\ln n!$ е голямата сума на Darboux за $\ln(x)$ за $x \in [0, n]$.



Да означим с $S(r_i)$ лицето на r_i , за $2 \leq i \leq n$. От една страна, $S(r_i) = (\ln i) \times 1 = \ln i$. От друга страна:

$$S(r_i) = y_i + t_i - z_i$$

където

- y_i е лицето на района R_i в r_i , който лежи под кривата $\ln x$.
- t_i е лицето на триъгълника T_i с върхове $(i - 1, \ln(i - 1))$, $(i - 1, \ln i)$ и $(i, \ln i)$.
- z_i е лицето на припокриването на R_i и T_i .

Тогава:

$$\ln i = y_i + t_i - z_i \quad (2.46)$$

Тези припокривания с лица z_2, z_3 и така нататък стават все по-малки много бързо. На фигурата се вижда ясно само първото от тях, чието лице е z_2 ; става дума за района в r_2 между червената крива и синята отсечка под нея. Очевидно:

$$\begin{aligned} y_i &= \int_{i-1}^i \ln x \, dx \\ t_i &= \frac{\ln i - \ln(i-1)}{2} \\ z_i &= y_i - u_i - t'_i \end{aligned} \quad (2.47)$$

където u_i е лицето на правоъгълника с върхове $(i - 1, 0)$, $(i - 1, \ln(i - 1))$, $(i, \ln(i - 1))$ и $(i, 0)$, а t'_i е лицето на триъгълника с върхове $(i - 1, \ln(i - 1))$, $(i, \ln i)$ и $(i, \ln(i - 1))$. Очевидно:

$$u_i = \ln(i - 1) \quad (2.48)$$

$$t'_i = t_i = \frac{\ln i - \ln(i-1)}{2} \quad (2.49)$$

За y_i е вярно следното:

$$\begin{aligned} y_i &= \int_{i-1}^i \ln x \, dx = i \ln i - i - ((i-1) \ln (i-1) - (i-1)) \\ &= \ln (i-1) - i \ln (i-1) - 1 + i \ln i \end{aligned} \quad (2.50)$$

Заместваме (2.48), (2.49) и (2.50) в (2.47) и получаваме:

$$\begin{aligned} z_i &= \underbrace{\ln (i-1) - i \ln (i-1) - 1 + i \ln i}_{y_i} - \underbrace{\ln (i-1)}_{u_i} - \underbrace{\frac{\ln i - \ln (i-1)}{2}}_{t_i} \\ &= \left(1 - i + -1 + \frac{1}{2}\right) \ln (i-1) + \left(i - \frac{1}{2}\right) \ln i - 1 \\ &= \left(i - \frac{1}{2}\right) \ln i - \left(i - \frac{1}{2}\right) \ln (i-1) - 1 \\ &= \left(i - \frac{1}{2}\right) \ln \left(\frac{i}{i-1}\right) - 1 \end{aligned} \quad (2.51)$$

Лема 14

Редът $\sum_{i=2}^{\infty} z_i$ е сходящ.

Доказателство: Да разгледаме развиването в степенен ред на $\ln(1+x)$:

$$\ln(1+x) = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} x^k$$

Тогава:

$$\begin{aligned} \ln(1-x) &= \ln(1+(-x)) = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} (-x)^k = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} (-1)^k x^k \\ &= \sum_{k=1}^{\infty} \frac{(-1)^{2k+1}}{k} x^k = -\sum_{k=1}^{\infty} \frac{x^k}{k} \end{aligned}$$

Оттук следва, че:

$$\begin{aligned} \ln(1+x) - \ln(1-x) &= \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} x^k + \sum_{k=1}^{\infty} \frac{x^k}{k} = \sum_{k=1}^{\infty} \frac{(-1)^{k+1} + 1}{k} x^k \\ &= 2x + \frac{2x^3}{3} + \frac{2x^5}{5} + \frac{2x^7}{7} + \dots \end{aligned}$$

Изведохме, че:

$$\frac{1}{2} \ln \left(\frac{1+x}{1-x} \right) = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \dots$$

Заместваме x с $\frac{1}{2i-1}$ и получаваме:

$$\begin{aligned} \frac{1}{2} \ln \left(\frac{1 + \frac{1}{2i-1}}{1 - \frac{1}{2i-1}} \right) &= \frac{1}{2i-1} + \frac{1}{3(2i-1)^3} + \frac{1}{5(2i-1)^5} + \frac{1}{7(2i-1)^7} + \dots \leftrightarrow \\ \frac{2i-1}{2} \ln \left(\frac{\frac{2i}{2i-1}}{\frac{2i-2}{2i-1}} \right) &= 1 + \frac{1}{3(2i-1)^2} + \frac{1}{5(2i-1)^4} + \frac{1}{7(2i-1)^6} + \dots \leftrightarrow \\ \left(i - \frac{1}{2} \right) \ln \left(\frac{i}{i-1} \right) - 1 &= \frac{1}{3(2i-1)^2} + \frac{1}{5(2i-1)^4} + \frac{1}{7(2i-1)^6} + \dots \end{aligned}$$

откъдето:

$$\left(i - \frac{1}{2} \right) \ln \left(\frac{i}{i-1} \right) - 1 \leq \frac{1}{3} \left(\frac{1}{(2i-1)^2} + \frac{1}{(2i-1)^4} + \frac{1}{(2i-1)^6} + \dots \right)$$

Но за $i > 1$, $\frac{1}{(2i-1)^2} + \frac{1}{(2i-1)^4} + \frac{1}{(2i-1)^6} + \dots$ е сходящ геометричен ред със сума:

$$\frac{\frac{1}{(2i-1)^2}}{1 - \frac{1}{(2i-1)^2}} = \frac{\frac{1}{(2i-1)^2}}{\frac{(2i-1)^2 - 1}{(2i-1)^2}} = \frac{1}{4i^2 - 4i + 1 - 1} = \frac{1}{4} \left(\frac{1}{i(i-1)} \right) = \frac{1}{4} \left(\frac{1}{i-1} - \frac{1}{i} \right)$$

Оттук следва, че:

$$\underbrace{\left(i - \frac{1}{2} \right) \ln \left(\frac{i}{i-1} \right) - 1}_{z_i} \leq \frac{1}{3} \left(\frac{1}{4} \left(\frac{1}{i-1} - \frac{1}{i} \right) \right) = \frac{1}{12} \left(\frac{1}{i-1} - \frac{1}{i} \right) \quad (2.52)$$

Лявата страна на неравенството (2.52) е z_i съгласно (2.51). Тогава:

$$z_i \leq \frac{1}{12} \left(\frac{1}{i-1} - \frac{1}{i} \right)$$

Тогава, за всяко $m > 1$ и за всяко $k > 0$:

$$\begin{aligned} z_m &\leq \frac{1}{12(m-1)} - \frac{1}{12m} \\ z_{m+1} &\leq \frac{1}{12m} - \frac{1}{12(m+1)} \\ &\dots \\ z_{m+k} &\leq \frac{1}{12(m+k-1)} - \frac{1}{12(m+k)} \end{aligned}$$

Сумираме тези неравенства и получаваме:

$$z_m + z_{m+1} + \dots + z_{m+k} \leq \frac{1}{12(m-1)} - \frac{1}{12(m+k)}$$

И така:

$$\sum_{i=m}^{\infty} z_i \leq \frac{1}{12(m-1)} \quad (2.53)$$

и в частност:

$$\sum_{i=2}^{\infty} z_i \leq \frac{1}{12} \quad (2.54)$$

Показваме, че $\sum_{i=2}^{\infty} z_i$ е сходящ.

□

Сега се връщаме на уравнение (2.46). Сумираме за $2 \leq i \leq n$ и получаваме:

$$\begin{aligned} \underbrace{\sum_{i=2}^n S(r_i)}_{\ln n!} &= \underbrace{\sum_{i=2}^n \int_{i-1}^i \ln x dx}_{\int_1^n \ln x dx = 1 + n \ln n - n} + \underbrace{\sum_{i=2}^n \frac{\ln i - \ln(i-1)}{2}}_{\frac{\ln n}{2}} - \sum_{i=2}^n z_i \leftrightarrow \\ \ln n! &= 1 + n \ln n - n + \frac{\ln n}{2} - \underbrace{\left(\sum_{i=2}^{\infty} z_i - \sum_{i=n+1}^{\infty} z_i \right)}_{\text{ограничено от константа съгласно (2.54)}} \end{aligned} \quad (2.55)$$

Вземаме експонента при основа e от двете страни и получаваме:

$$\begin{aligned} n! &= e^{1 \cdot e^{n \ln n} \cdot e^{-n} \cdot e^{\frac{\ln n}{2}} \cdot e^{-\sum_{i=2}^{\infty} z_i} \cdot e^{\sum_{i=n+1}^{\infty} z_i}} \\ &= e \left(\frac{n}{e} \right)^n \sqrt{n} \cdot e^{-\sum_{i=2}^{\infty} z_i} \cdot e^{\sum_{i=n+1}^{\infty} z_i} = \left(e^{1 - \sum_{i=2}^{\infty} z_i} \right) \sqrt{n} \left(\frac{n}{e} \right)^n \left(e^{\sum_{i=n+1}^{\infty} z_i} \right) \end{aligned}$$

Съгласно (2.54), $\sum_{i=2}^{\infty} z_i$ не надхвърля $\frac{1}{12}$, откъдето $e^{\frac{11}{12}} \leq e^{1 - \sum_{i=2}^{\infty} z_i} \leq e$. От (2.53) следва, че $e^{\sum_{i=n+1}^{\infty} z_i} \leq e^{\frac{1}{12n}}$. Следователно, съществува константа c , такава че $e^{\frac{11}{12}} \leq c \leq e$ и

$$c \sqrt{n} \left(\frac{n}{e} \right)^n \leq n! \leq c \sqrt{n} \left(\frac{n}{e} \right)^n e^{\frac{1}{12n}}$$

Истинската стойност на c е $\sqrt{2\pi} \approx 2.50662827463101$, $e^{\frac{11}{12}} \approx 2.50094001366213$, $e \approx 2.71828182845905$, и наистина

$$2.50094001366213 \leq 2.50662827463101 \leq 2.71828182845905$$

□

^aФигура 2.10 е направена с Maple(TM).

Теорема 14

$$\lg n! \asymp n \lg n.$$

Доказателство: Знаем, че $n! \asymp \sqrt{n} \frac{n^n}{e^n}$ (вж. Теорема 12, която казва нещо по-силно). Прилагаме Следствие 7 и получаваме:

$$\lg n! \asymp \lg \left(\sqrt{n} \frac{n^n}{e^n} \right)$$

Но

$$\lg \left(\sqrt{n} \frac{n^n}{e^n} \right) = \frac{1}{2} \lg n + n \lg n - n \lg e$$

Забелязваме, че събирамото $n \lg n$ доминира асимптотично над останалите събирами. Това, че $-n$ е отрицателна функция, а ние дефинираме асимптотичните нотации (в частност, релацията \asymp) само върху положителни функции, не е съществен проблем. Очевидно $n \ln n - n \lg e$ е положителна функция за всички достатъчно големи стойности на аргумента и $\frac{1}{2} \lg n + n \ln n - n \lg e \asymp n \ln n$. Доказвахме, че $\lg n! \asymp n \lg n$. \square

Важно свойство на биномния коефициент е, че средният биномен коефициент $\binom{n}{\frac{n}{2}}$ расте, в асимптотичния смисъл, почти като 2^n . За удобство, да допуснем, че n е четно.

Теорема 15

$$\binom{n}{\frac{n}{2}} \asymp \frac{1}{\sqrt{n}} 2^n.$$

Доказателство: Знаем, че:

$$\binom{n}{\frac{n}{2}} = \frac{n!}{\left(\frac{n}{2}!\right)^2}$$

Прилагаме Теорема 12 върху двата факториела и получаваме

$$\frac{n!}{\left(\frac{n}{2}!\right)^2} \asymp \frac{\sqrt{2\pi n} \frac{n^n}{e^n}}{\left(\sqrt{2\pi \frac{n}{2}} \frac{\frac{n}{2}^{\frac{n}{2}}}{e^{\frac{n}{2}}}\right)^2} = \sqrt{2\pi} \sqrt{n} \frac{n^n}{e^n} \times \frac{e^n}{\pi n^{\frac{n}{2}}} = \sqrt{\frac{2}{\pi}} \times \frac{1}{\sqrt{n}} \times 2^n$$

\square

Нотация 2: H_n

С “ H_n ” означаваме n -тата парциална сума на хармоничния ред. Иначе казано:

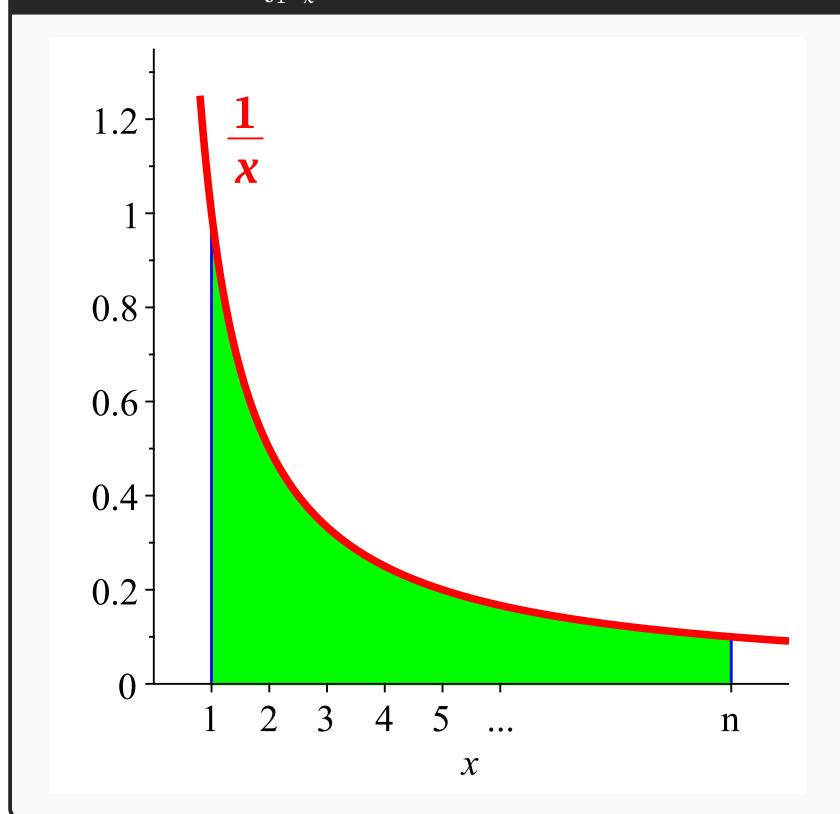
$$H_n = \sum_{k=1}^n \frac{1}{k}$$

Теорема 16

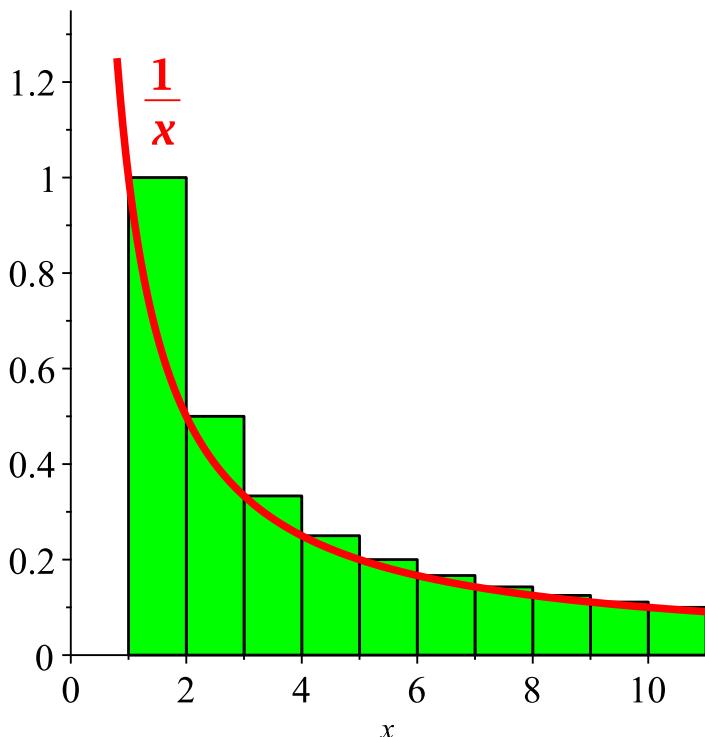
$$H_n \asymp \lg n.$$

Доказателство: Ще ограничим $\int_1^n \frac{1}{x} dx$ отгоре и отдолу със стъпаловидни функции, отговарящи на парциални суми на хармоничния ред. Знаем, че за всяко $n \in \mathbb{N}^+$, $\int_1^n \frac{1}{x} dx$ е площта на района, ограден от графиката на $\frac{1}{x}$, абсцисата и двете вертикални линии през точките 1 и n от абсцисата (вж. Фигура 2.11[†]).

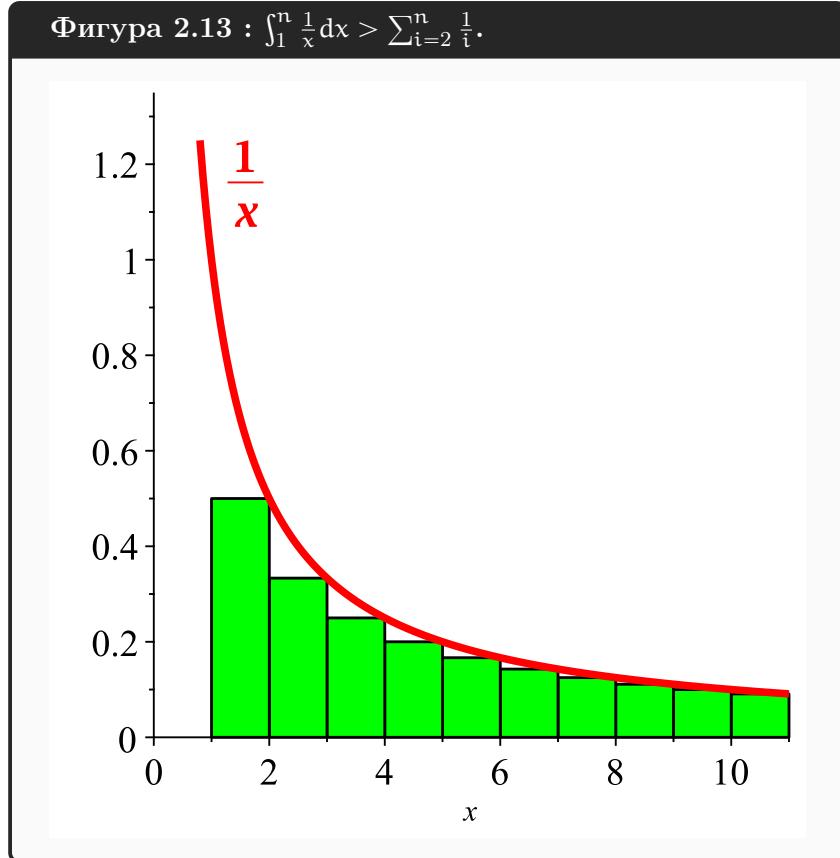
[†]Фигура 2.11 е направена с Maple(TM).

Фигура 2.11 : $\int_1^n \frac{1}{x} dx$ е площта на зеления район.

Да разгледаме стъпаловидната функция $\frac{1}{[x]}$ върху интервала $[1, \dots, n]$. Вярно е, че $\forall x \in [1, \dots, n] : \frac{1}{[x]} \geq \frac{1}{x}$, понеже $[x] \leq x$ за всяко x . Забелязваме, че площта на района, ограден от $\frac{1}{[x]}$, абсцисата и двете вертикални линии през точките 1 и n от абсцисата е точно $\sum_{i=1}^{n-1} \frac{1}{i}$, защото този район се разбива на $n - 1$ правоъгълника, едната страна на всеки от които е с дължина единица, а другата е $\frac{1}{i}$ за някое $i \in \{1, \dots, n - 1\}$. Очевидно този район има площ, по-голяма от площта под $\frac{1}{x}$. С други думи, $\sum_{i=1}^{n-1} \frac{1}{i} > \int_1^n \frac{1}{x} dx$. Това е илюстрирано на Фигура 2.12.

Фигура 2.12 : $\sum_{i=1}^{n-1} \frac{1}{i} > \int_1^n \frac{1}{x} dx$.

Сега да разгледаме стъпаловидната функция $\frac{1}{\lceil x \rceil}$ върху интервала $[1, \dots, n]$. $\forall x \in [1, \dots, n] : \frac{1}{\lceil x \rceil} \leq \frac{1}{x}$, понеже $\lceil x \rceil \geq x$ за всяко x . Забелязваме, че площта на района, ограден от $\frac{1}{\lceil x \rceil}$, абсцисата и двете вертикални линии през точките 1 и n от абсцисата е точно $\sum_{i=2}^n \frac{1}{i}$, защото този район се разбива на $n - 1$ правоъгълника, едната страна на всеки от които е с дължина единица, а другата е $\frac{1}{i}$ за някое $i \in \{2, \dots, n\}$. Очевидно този район има площ, по-малка от площта под $\frac{1}{x}$. С други думи, $\int_1^n \frac{1}{x} dx > \sum_{i=2}^n \frac{1}{i}$. Това е илюстрирано на Фигура 2.13.

Фигура 2.13 : $\int_1^n \frac{1}{x} dx > \sum_{i=2}^n \frac{1}{i}$.

Показахме, че

$$\sum_{i=2}^n \frac{1}{i} < \int_1^n \frac{1}{x} dx < \sum_{i=1}^{n-1} \frac{1}{i}$$

Но $\sum_{i=2}^n \frac{1}{i} = (\sum_{i=1}^n \frac{1}{i}) - 1 = H_n - 1$, а $\sum_{i=1}^{n-1} \frac{1}{i} = H_{n-1}$. Тогава

$$H_n - 1 < \int_1^n \frac{1}{x} dx < H_{n-1}$$

Но $\int_1^n \frac{1}{x} dx = \ln x \Big|_1^n = \ln n - \ln 1 = \ln n - 0 = \ln n$. Тогава:

$$H_n - 1 < \ln n < H_{n-1}$$

Очевидно съществуват положителни константи c_1 и c_2 и стойност на аргумента n_0 , такива че за всички $n > n_0$ е вярно

$$c_1 \cdot H_n \leq \ln n \leq c_2 \cdot H_n$$

Например, $c_1 = \frac{1}{2}$, $c_2 = 1$ и $n_0 = 1000$.

Заключаваме, че $\ln n \asymp H_n$. □

2.3.9 Асимптотичните нотации и релациите на асимптотични сравнения

На пръв поглед, релациите \asymp и така нататък са доста по-удобни и прегледни от съответните асимптотични нотации като Θ и така нататък. Но, както казва Knuth в [35], има случаи, в

които е много по-удобно да ползваме асимптотичните нотации, а не релационните символи. А именно, когато нотациите се появяват вътре в някакви изрази (а не вдясно от знака “=”). Knuth говори за това, което ние нарекохме “анонимна функция” (вж. Конвенция 2).

Knuth в [35] дава следния пример:

$$\left(1 + \frac{H_n}{n}\right)^{H_n} = e^{\frac{H_n^2}{n} + O\left(\frac{(\lg n)^3}{n^2}\right)}$$

Същото това нещо, написано не с голямо-О нотация, а с релацията \leq , би било тромаво и грозно, защото, за да се използва релационният символ, всичко друго освен $\frac{(\lg n)^3}{n^2}$ би трябвало да се намира от едната му страна.

2.3.10 Асимптотични сравнения на функции на повече от една променлива

Определение 15 е обобщение на Определение 3 за функции на две променливи.

Определение 15: $\Theta(g(n, m))$

За всяка функция $g(n, m)$:

$$\Theta(g(n, m)) \stackrel{\text{def}}{=} \{f(n, m) \mid \exists c_1, c_2 > 0 \exists n_0, m_0 \forall n \geq n_0 \forall m \geq m_0 : 0 \leq c_1 \cdot g(n, m) \leq f(n, m) \leq c_2 \cdot g(n, m)\}$$

Очевидно е как да обобщим и другите четири асимптотични нотации за функции на две променливи. Също така е очевидно как да обобщаваме и за повече от две променливи.

2.3.11 Релация \sim : друга релация на асимптотична еквивалентност

Определение 16: Релация \sim над множеството на функциите

За всички функции $f(n)$ и $g(n)$, $f(n) \sim g(n)$ тогава и само тогава, когато $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$.

Очевидно релацията \sim е рефлексивна, симетрична и транзитивна, поради което е релация на еквивалентност над множеството на функциите, също като \asymp . Да си припомним Наблюдение 4. От него следва, че \sim е по-силна от \asymp , защото:

$$\begin{aligned} f(n) \sim g(n) &\rightarrow f(n) \asymp g(n) \\ \neg(f(n) \asymp g(n)) &\rightarrow f(n) \sim g(n) \end{aligned}$$

Поради това доказателството, че за някакви две функции $f(n)$ и $g(n)$ е в сила $f(n) \sim g(n)$, може да е по-трудно от доказателството, че $f(n) \asymp g(n)$. В тези лекции почти няма да ползваме по-силната релация \sim и ще смятаме, че ако сме показали $f(n) \asymp g(n)$, то ние сме намерили най-точната възможна асимптотична близост между $f(n)$ и $g(n)$.

Забележете, че току-що дефинираната \sim и фактор-релацията \asymp от Допълнение 13 (вижте Определение 14) нямат нищо общо.

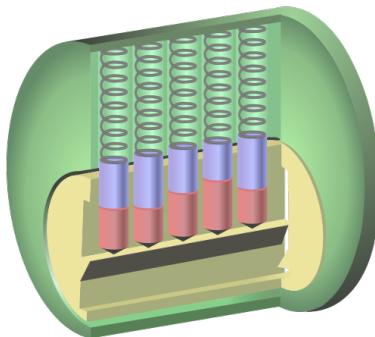
2.4 Винаги ли е лоша високата сложност

В заключение ще разискваме доколко бавно е синоним на лошо в света на алгоритмите. Наистина, ние търсим колкото е възможно по-бързи алгоритми и свикваме да мислим, че бавен алгоритъм е лош алгоритъм, а бърз алгоритъм е добър алгоритъм. Има важни изчислителни задачи, за които не са известни бързи алгоритми, а има и задачи, за които е доказано, при определени допускания, че няма бързи алгоритми. От конвенционалната гледна точка, липсата на бързи алгоритми—независимо от това дали се дължи на човешкото незнание или е иманентна—е нещо лошо; ако мислим, че бърз=добър и бавен=лош, то високата сложност е нещо като проклятие.

Има обаче друга гледна точка, от която високата сложност е благословия. Това е гледната точка на сигурността. Първо да разгледаме съвсем обикновен пример: ключ от секретна брава като този

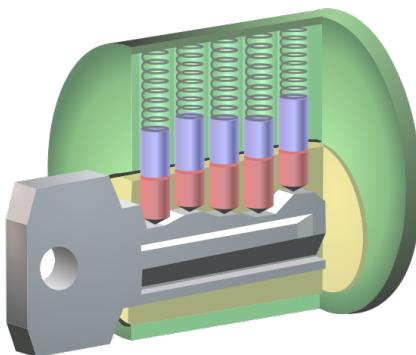


Добре известно е, че секретна брава се отваря при завъртането на цилиндъра в нейната ключалка. Цилиндърът стои пълтно в гнездо в ключалката. Ако в ключалката няма ключ, този цилиндър не може да бъде завъртян, защото в него има няколко щифта (обикновено не повече от 6 на брой), сложени в странични напречни отвори (също с цилиндрична форма). Тези отвори са наредени в редица и продължават в гнездото. Всеки щифт е едновременно и в цилиндъра, и в гнездото отстрани. Всеки щифт е срязан на някаква височина, така че всъщност той се състои от два щифта, допрени пълтно. Ако няма ключ в ключалката, едно от парчетата на всеки щифт стои едновременно и в цилиндъра, и в гнездото, и така щифтовене пречат на цилиндъра да се върти.



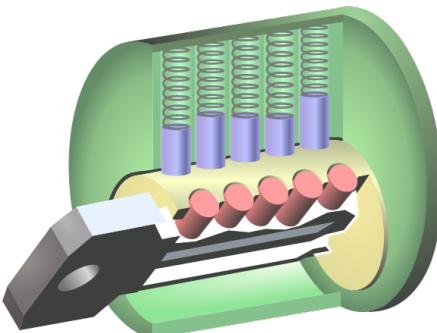
файлът е от wikipedia под cc-3.0

При вкарване на ключ в ключалката, назъбената му повърхност минава точно под щифтовете и ги избутва навън, на различни разстояния съгласно височините на зъбите. Правилният ключ размества щифтовете в отворите им така, че всички срезове на всички щифтове се оказват подравнени точно с границата между цилиндъра и гнездото



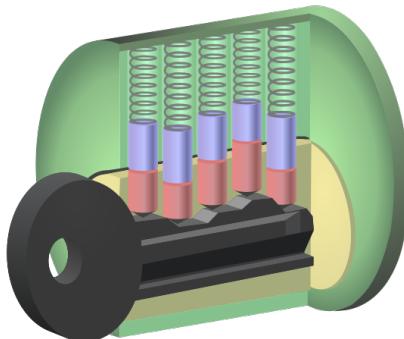
файлът е от wikipedia под cc-3.0

и вече нищо не пречи на цилиндъра да бъде завъртян, което води до издърпване на езика на бравата и вратата се отваря.



файлът е от wikipedia под cc-3.0

Ако обаче в ключалката бъде сложен неправилен ключ, поне един щифт ще пречи на завъртането на цилиндъра, понеже срезът му няма да е подравнен с границата между цилиндъра и гнездото



файлът е от wikipedia под cc-3.0

и бравата няма да бъде отворена.

Броят на щифтовете за дадена ключалка не е тайна. Всеки щифт е срязан на една измежду няколко възможни позиции, обикновено не повече от 10 различни позиции. Бройката на възможните позиции също не е тайна. Тайната на ключа е, точно на каква височина измежду всички възможни височини е срязан всеки щифт. Нека броят на щифтовете е n и има m възможности за различни височини на срязване на всеки щифт. Тогава, от информационна гледна точка, ключът е елемент от $\{1, 2, \dots, m\}^n$. С други думи, векторът от височините, на които са срязани щифтовете, е пълно описание на ключа (заедно с информацията за профила му и за разстоянията между щифтовете, но да игнорираме тези детайли). Всички възможни ключове са най-много краен брой, а именно не повече от m^n . Например, ако $m = 10$ и $n = 6$, щифтовете са най-много 1 000 000.

Сигурността при физическите ключалки идва оттам, че атака срещу ключалката, която се състои в изработване на всички възможни ключове и последователното им пробване, е абсолютно непрактична. Виждаме, че в някакъв смисъл **сигурността идва от високата**

сложност на атаката, която се състои в опитване на всички възможности; в случая, сложността на изработване на стотици хиляди ключове-проби и последователното им опитване.

В компютърната сигурност, високата сложност дава сигурност по доста аналогичен начин. Тази материя е далеч отвъд началния курс по алгоритми. Тук само ще споменем, че в криптографията[†], която е от огромно значение за компютърната сигурност, в съвременния си вид се основава на използване на ключове (но чисто информационни, а не метални парчета). Съвременната криптография е основана на едно допускане, известно като *Принцип на Kerckhoffs*: противникът знае всички детайли от криптографската система—алгоритми и протоколи—единственото, което **не знае**, е ключа[‡]. Наистина, използваните в криптографията протоколи не са тайна – несериозно е да се мисли, че система, която е замислена и конструирана от множество хора и се използва с години от много повече хора, може да остане тайна за добре мотивиран противник.

Най-общо казано, криптирането на информацията работи така. Даден участник в комуникацията, която трябва да остане тайна от подслушвача, прилага определен алгоритъм с два входа: съобщението, което трябва да се предаде, и ключа. Изходът от този алгоритъм е стринг, който е неразбираем при прочитане. Процесът на превръщане на разбирамото съобщение в неразбираем стринг се нарича *шифриране*. Приемащата страна (или страни, ако съобщението се праща масово) прилага същия или друг алгоритъм с два входа: полученото неразбираемо съобщение и ключа, и изходът е първоначалното разбирамо съобщение. Процесът на превръщане на неразбираемия стринг в началното разбирамо съобщение се нарича *десифриране*. Подслушвачът има достъп само до междинния, шифриран вариант на съобщението, от който не се научава нищо. Подслушвачът знае въпросните алгоритми, но не знае ключа. Ако подслушвачът не разполага с никаква информация за ключа и не разполага с алтернативни методи, единственото, което може да направи, е да започне да генерира на сляпо ключ след ключ и да опитва последователно да ги прилага, разчитайки, че ще налучка правилния ключ и ще разбере съобщението. Отново, **сигурността идва от високата сложност на атаката, която се състои в опитване на всички възможности.**

Важен дял от криптографията е криптографията с отворен ключ. При нея, всеки потребител има два ключа, публичен и секретен, като публичният ключ се излага—както показва и името—публично, а секретният се пази в тайна. Публичното излагане на секретния ключ би било фатално за сигурността. Ако потребител А иска да прати съобщение на потребител В, тя взема публичният ключ на В, шифрира с него съобщението си и го праща на В. На свой ред В, използвайки секретния си ключ, десифрира съобщението. Съобщението може да се десифрира само чрез секретния ключ на В. Подслушвачът вижда шифрирания вариант, но от него не научава нищо, защото не разполага със секретния ключ на В. Интересното е, че за всеки потребител, двата ключа, публичен и секретен, са взаимно зависими, но сложността на задачата за намиране на секретния ключ от публичния е огромна. Още веднъж, **сигурността идва от високата алгоритмична сложност на атаката.**

Криптографията с отворен ключ има решаващо значение за (почти) сигурното комуникиране в несигурна, публично достъпна среда. Ахилесовата пета на другата криптография (само с един ключ) е разпространението на ключа. Компютърните услуги като Интернет банкиране са немислими в масовия си вариант без криптография с публичен ключ.

[†]Криптография, съвсем накратко, е науката за предаване на информация в среда, в която има подслушвач, по такъв начин, че подслушвачът да не може да разбере какво се предава, въпреки че вижда ясно това, което се предава, и е наясно, че то носи важна информация.

[‡]Аналогично, при секретните брави цялата сигурност е в информацията за височините на зъбите на ключа, при допускането, че противникът знае всичко останало като формата на профила на ключа, броя на щифтовете и т. н., и разполага с необходимата техника, за да си направи физически ключ от описанietо му.

Задълбочено изложение на връзката между високата сложност на дадени изчислителни задачи и криптографията с отворен ключ има в статиите на Fortnow [19] и Impagliazzo [28]. Изключително детайлно въведение в криптографията е култовата книга Applied Cryptography на Bruce Schneier [57].

Лекция 3

Сортиране. Елементарни сортиращи алгоритми.

Резюме: Разглеждаме задачата СОРТИРАНЕ и нейната важност за решаването на други задачи. Въвеждаме понятието *стабилност на сортиращ алгоритъм*. Разглеждаме два елементарни сортиращи алгоритъма: INSERTION SORT и SELECTION SORT и анализираме тяхната коректност, сложност по време и памет и стабилност. Особено внимание отделяме на коректността, като въвеждаме доказване чрез инварианта на цикъл.

3.1 Обща дефиниция

Понятието “преднаредба” е дефинирано и обяснено подробно в Допълнение 13.

Изч. Задача: СОРТИРАНЕ

пример: В контекста на някакво множество A , чиито елементи са подходящи за сортиране и върху които е дефинирана релация на пълна преднаредба \leq , е даден вектор $X = (a_1, a_2, \dots, a_n)$, такъв че $\forall i \in \{1, \dots, n\} : a_i \in A$.

решение: перmutация $(a'_1, a'_2, \dots, a'_n)$ на елементите от X , такава че $a'_1 \leq a'_2 \leq \dots \leq a'_n$

Подчертаваме, че примерът на задачата е **редицата** (a_1, a_2, \dots, a_n) , а не (мулти)множеството $\{a_1, \dots, a_n\}_M$.

Елементите a_1, \dots, a_n не са непременно естествени числа. “Елементи, подходящи за сортиране” означава елементи, за които е реалистично допускането, че имат големина единица и че може да бъдат сравнявани и местени в паметта в единица време. Може да са реални числа (тогава е особено важно да се подчертава това допускане). Може да са дори комплексни числа, като тогава има различни възможности за дефиниране на релацията \leq . Може да са някакви по-общи структури. На практика най-често възниква необходимостта да се сортират записи, всеки от които има **ключ** и някаква **сателитна информация**, като релацията \leq е дефинирана само върху ключовете. Ключовете във всеки запис може да са повече един и в такъв случай говорим за първичен ключ, вторичен ключ и така нататък.

С учебна цел ще разглеждаме основно сортиране на цели числа, а релацията \leq ще бъде добре познатата “по-малко или равно” \leqslant .

3.2 Защо сортирането е важно

По правило в практиката сортирането не е самоцелно, а е само първи етап от решаването на някаква задача. Има доста примери за важни задачи, за които бързото решение използва сортирането като предварителна обработка. Алгоритмичният фолклор казва, че ако съставител на алгоритми е изправен пред непозната задача, едно от първите неща, които трябва да опита, за да направи ефикасен алгоритъм, е да сортира някакви данни и да види дали от това няма да произтече нещо полезно [63, стр. 106].

Наблюдение 6

Предимството на сортираниите данни над несортирани се корени в транзитивността на релацията \leq . Ако редицата (a_1, a_2, \dots, a_n) е сортирана и е известно, че $a_i \leq x$ за някое i и някакво x , то от транзитивността на \leq заключаваме, че $a_j \leq x$ за всяко $j < i$.

3.2.1 Двоично търсене

В простия си вариант като задача за разпознаване, Търсене се дефинира така.

Изч. Задача: Търсене, версия за разпознаване

пример: a_1, a_2, \dots, a_n , key: обекти от един и същи тип

въпрос: Дали има елемент измежду a_1, a_2, \dots, a_n , който е равен на key?

На практика по-полезна е следната версия на задачата. Ще я наречем *оптимизационната версия*.

Изч. Задача: Търсене, оптимизационна версия

пример: a_1, a_2, \dots, a_n , key: обекти от един и същи тип

решение: Ако има елемент измежду a_1, a_2, \dots, a_n , който е равен на key, индексът на един такъв елемент. В противен случай индикация, че такъв елемент няма.

Ако входът на произволен алгоритъм за Търсене е произволна редица a_1, a_2, \dots, a_n , то единствената възможност за действие е проверяване дали $a_i = ?$ key по всички $i \in \{1, \dots, n\}$. Това се нарича *последователно търсене*. Най-лошият случай по отношение на сложността е, key да не е нито един от a_1, a_2, \dots, a_n . Тогава всеки алгоритъм би трябало да направи n сравнения в най-лошия случай. Очевидно е, че ако сме пропуснали да проверим дали $a_i = ?$ key за поне едно i , няма как да сме убедени, че key не се среща.

Ако обаче елементите a_1, a_2, \dots, a_n са *уникални*[†] и са сортирани можем да приложим *двоично търсене* (на английски е *binary search*), което е много по-бързо от последователното търсене.

[†]Двоичното търсене не е приложимо, ако има елементи с еднаква стойност, които все пак различаваме някак—примерно, еднакъв първичен ключ, но различен вторичен ключ—и търсим не просто по стойност, а точно определен елемент измежду тези с дадена стойност. С други думи, за да работи двоичното търсене, трябва елементите да са подредени с линейна наредба.

```

BINSEARCH1(A[1, ..., n], key)
1 (* итеративен вариант *)
2 (* A е сортиран *)
3  $\ell \leftarrow 1$ 
4  $h \leftarrow n$ 
5 while  $\ell < h$  do
6   mid  $\leftarrow \lfloor \frac{\ell+h}{2} \rfloor$ 
7   if A[mid] = key
8     return mid
9   else if key < A[mid]
10    h  $\leftarrow mid - 1$ 
11  else  $\ell \leftarrow mid + 1$ 
12 return -1

```

```

BINSEARCH2(A[1, ..., n], key,  $\ell$ ,  $h$ )
1 (* рекурсивен вариант *)
2 (* A е сортиран *)
3 if  $\ell > h$ 
4   return -1
5 else
6   mid  $\leftarrow \lfloor \frac{\ell+h}{2} \rfloor$ 
7   if A[mid] = key
8     return mid
9   else if key < A[mid]
10    return
11   BINSEARCH2(A, key,  $\ell$ , mid - 1)
12 else
13   return
14   BINSEARCH2(A, key, mid + 1,  $h$ )

```

Началното викане на рекурсивната версия е $\text{BINSEARCH2}(A[1, \dots, n], \text{key}, 1, n)$. Посочените два алгоритъма решават оптимизационната версия на ТЪРСЕНЕ. Накратко, тяхната коректност се обосновава така: ако изобщо елементът key се намира в масива A , той се намира в подмасива $A[\ell, \dots, h]$. Това твърдение остава вярно след промените в индексите ℓ и h само защото масивът е сортиран предварително.

Сложността по време и на двета алгоритъма е $\Theta(\lg n)$ в най-лошия случай, която обосновка остава на читателя. Нещо повече, точната сложност като брой сравнения $A[\text{mid}] \stackrel{?}{=} \text{key}$ е $\lfloor \lg n \rfloor + 1$ в най-лошия случай. Разликата между последователното търсене в линейно време и двоичното търсене в логаритмично време е огромна. Ако, примерно, $n = 7\,000\,000\,000$, с не повече от 33 достъпа до масива можем да установим, че даден елемент не е във входа, ако ползваме двоично търсене. Сравнете 33 с 7 000 000 000!

3.2.2 Най-близки елементи

Ето два варианта на задачата.

Изч. Задача: НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, ВАР. 1

пример: a_1, a_2, \dots, a_n : цели числа; $n \geq 2$

решение: $\min \{|a_i - a_j| : 1 \leq i < j \leq n\}$

Изч. Задача: НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, ВАР. 2

пример: a_1, a_2, \dots, a_n : цели числа; $n \geq 2$

решение: Двойка елементи (a_i, a_j) , такива че разликата $i \neq j$ и $|a_i - a_j|$ е минимална.

Възможно е масивът да съдържа еднакви елементи. В такъв случай, в първата версия решението е 0, а във втората, наредена двойка (a_i, a_j) за кои да е a_i, a_j , такива че $a_i = a_j$.

Не е задължително елементите да са цели числа, но трябва да бъдат от множество, върху което е дефинирана пълна преднаредба, и трябва да е дефинирано разстояние между всеки

два елемента, което да се изчислява в константно време. В горната дефиниция разстоянието е $|a_i - a_j|$.

Наивното решение на задачата е да се тестват всички двуелементни подмножества така.

CLOSEST PAIR, NAÏVE($A[1, \dots, n]$)

```

1 closest ← ∞
2 for i ← 1 to n - 1
3   for j ← i + 1 to n
4     if |A[i] - A[j]| < closest
5       closest ← |A[i] - A[j]|
6       first ← A[i]
7       second ← A[j]
8 return (first, second)

```

Сложността по време на това решение е $\Theta(n^2)$.

По-добро решение е, първо елементите да бъдат сортирани и след това с едно единствено сканиране на сортираната последователност отляво надясно най-близката двойка ще бъде намерена. Сега не е необходимо да ползваме нотация за абсолютна стойност около $A[i+1] - A[i]$, защото след сортирането $A[i+1] - A[i]$ е задължително неотрицателно.

CLOSEST PAIR, SOPHISTICATED($A[1, \dots, n]$)

```

1 SORT(A)
2 closest ← ∞
3 for i ← 1 to n - 1
4   if A[i + 1] - A[i] < closest
5     closest ← A[i + 1] - A[i]
6     first ← A[i]
7     second ← A[i + 1]
8 return (first, second)

```

Доказателството за коректност се основава на простото наблюдение, че ако минималното разстояние между кои да два елемента от входа е δ , то след сортирането задържително има съседни елементи, разстоянието между които е δ ; в противен случай би имало два елемента от входа, разстоянието между които е дори по-малко.

Както ще видим в следваща лекция, сортирането може да бъде имплементирано със сложност $\Theta(n \lg n)$, така че бързото решение има сложност

$$\underbrace{\Theta(n \lg n)}_{\text{сортиране}} + \underbrace{\Theta(n)}_{\text{сканиране}} = \Theta(n \lg n),$$

което е много по-добре от $\Theta(n^2)$.

3.2.3 Уникалност на елементите

Изч. Задача: Уникалност на елементите

пример: a_1, a_2, \dots, a_n : цели числа; $n \geq 2$

въпрос: Вярно ли е, че няма нито два еднакви елемента измежду дадените?

Напълно аналогично на НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, наивното решение се състои в тестване на всички двуелементни подмножества, което означава $\Theta(n^2)$ тестове в най-лошия случай. Ако обаче първо сортираме числата, ако има еднакви елементи, всички елементи, които са два по два еднакви, задължително ще се появят в непрекъсната последователност в сортиранията наредба. С едно сканиране отляво надясно ще установим дали има еднакви елементи. Отново, сложността на подхода, използващ първо сортиране във време $\Theta(n \lg n)$, е $\Theta(n \lg n)$.

3.2.4 Мода

Мода е най-често срещана стойност между дадени стойности. Забележете, че тази стойност не е непременно уникална – ако няма повторения на елементи, всеки елемент е мода (поради което не е коректно да се казва “най-често срещаната стойност”). Мода може да се изчисли във време $\Theta(n \lg n)$ чрез предварително сортиране, което работи във време $\Theta(n \lg n)$. Ето едно възможно решение за намиране на мода в $\Theta(n \lg n)$ чрез сортиране, последвано от едно сканиране на сортираниите данни.

COMPUTE MODE($A[1, 2, \dots, n]$)

```

1  SORT(A)
2  mode ← A[1]
3  m ← 1
4  for i ← 2 to n
5      if A[i - 1] = A[i]
6          if A[i - 1] = mode
7              m ← m + 1
8      else
9          s ← s + 1
10         if s > m
11             mode ← A[i]
12             m ← s
13     else
14         s ← 1
15 return mode

```

Допълнение 15: Коректността на COMPUTE MODE

Както вече казахме, модата не е непременно уникална. Алгоритъмът връща *най-малката мода*. За краткост, навсякъде до края на Допълнение 15, когато казваме “moda” имаме предвид най-малката мода.

Следното определение е в сила **само в рамките на това доказателство**.

Определение 17: група

Нека $A[1, 2, \dots, n]$ е масив от числа. *Група* в A е всеки максимален по включване подмасив от еднакви елементи. *Дължината на групата* е броят на елементите ѝ. Ако групата се назава X , дължината ѝ се бележи с $|X|$. *Представител на групата* е стойността на кой да е от нейните елементи.

По отношение на даден индекс $i \in \{2, \dots, n\}$, казваме, че $A[i]$ разширява група, ако $A[i] = A[i - 1]$, и че $A[i]$ започва нова група, ако $A[i] \neq A[i - 1]$. $A[i]$ разширява група-мода, ако $A[i]$ разширява група и $A[i - 1]$ е мода на $A[1, \dots, i - 1]$. $A[i]$ превръща група в група-мода, ако $A[i]$ разширява група и $A[i - 1]$ не е мода на $A[1, \dots, i - 1]$, но $A[i]$ е мода на $A[1, \dots, i]$. $A[i]$ разширява група-немода, ако $A[i]$ разширява група, но не е мода на $A[1, \dots, i]$. \square

Забележете, че ако масивът е сортиран и $A[i]$ започва нова група, то $A[i]$ не може да е мода при $i \geq 2$.

Наблюдение 7

Нека $A[1, 2, \dots, n]$ има точно t различни елемента и е сортиран. Тогава A се състои от точно t групи, чиято сумарна дължина е n и чиито представители се появяват в строго нарастващ ред.

Наблюдение 8

За всяко $i \in \{2, \dots, n\}$, има точно четири, две по две изключващи се, възможности за $A[i]$. Ще бележим тези възможности с буквите Q_1, \dots, Q_4 .

Q_1 : $A[i]$ разширява група-мода.

Q_2 : $A[i]$ разширява група-немода.

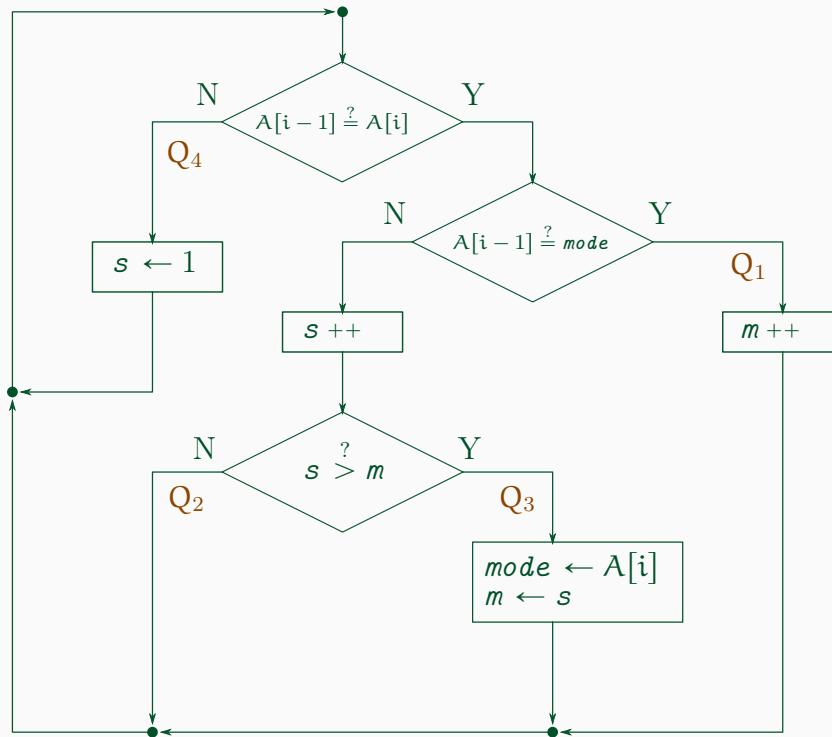
Q_3 : $A[i]$ превръща група в група-мода.

Q_4 : $A[i]$ започва нова група.

Удобно е да мислим за Q_1, \dots, Q_4 като за *състояния*. След сортирането, алгоритъмът сканира единократно масива отляво надясно и след прочитането на всеки пореден елемент преминава неявно в едно от четирите състояния. Неявно, защото никъде не се говори експлицитно за състояния. Но трите вложени **if**-а в някакъв смисъл ни дават именно това: четири различни състояния. Най-вътрешният **if** няма **else**, но това е без значение – можем да мислим, че има празен **else**. При всяко преминаване през тялото на цикъла, в зависимост от истинността на трите булеви условия, точно едно от четирите възможни състояние се реализира (имплицитно). Фигура 3.1 показва блок-схема на тялото на цикъла на COMPUTE MODE, на която ясно се вижда кое преминаване през тялото на цикъла на кое състояние отговаря.

Въвеждането на тези четири състояния няма отношение към доказателството за коректност – то не ги използва никъде. Въвеждаме тези четири състояния, за да обосновем конструкцията на алгоритъма. Иначе казано, те са полезни за **дизайна**, а не за **анализа** на алгоритъма.

Фигура 3.1 : Блок-схема на цикъла на COMPUTE MODE.



Следното твърдение е инварианта на **for**-цикъла (редове 1–14).

Инвариант 1: COMPUTE MODE

При всяко достигане на ред 4 на COMPUTE MODE:

- ① $mode$ съдържа стойността на модата на $A[1, \dots, i - 1]$.
- ② m съдържа броя на нейните появи в $A[1, \dots, i - 1]$.
- ③ Ако $A[i - 1]$ не е мода на $A[1, \dots, i - 1]$, то s съдържа дължината на най-дясната група в $A[1, \dots, i - 1]$.

База. Нека изпълнението е на ред 4 за първи път. Тогава $i = 2$ и подмасивът $A[1, \dots, i - 1]$ е всъщност $A[1]$. Модата на $A[1]$ очевидно е $A[1]$ и броят на нейните появи е 1. Да видим дали всички съждения на инвариантата са верни.

- ① $mode = A[1]$ заради присвояването на ред 2. ✓
- ② $m = 1$ и заради присвояването на ред 3. ✓
- ③ Това твърдение също е вярно. То е импликация, чийто антецедент е лъжа, понеже $A[i - 1]$ е мода на $A[1, \dots, i - 1]$. Импликацията, чийто антецедент е лъжа, е истина. ✓

Поддръжка. Да допуснем, че инвариантата е в сила на някое изпълнение на ред 4, което не е последно.

Случай I $A[i-1] = A[i]$. Условието на ред 5 е истина и изпълнението отива на ред 6.

Случай I.1 $A[i-1]$ е мода на $A[1, \dots, i-1]$. Този подслучай е Q_1 : $A[i]$ разширява група-мода. Съгласно индуктивното предположение, изпълнено е $A[i-1] = mode$. Очевидно и $A[i] = mode$. Условието на ред 6 е истина и изпълнението отива на ред 7. Да видим дали всички съждения на инвариантата са верни.

- ① Щом $mode$ е мода на $A[1, \dots, i-1]$ и $A[i] = mode$, то $mode$ е мода и на $A[1, \dots, i]$. Преди следващото достигане на ред 4, i се инкрементира. Спрямо новото i е вярно, че $mode$ е мода на $A[1, \dots, i-1]$. ✓
- ② Изпълнява се ред 7 и m се инкрементира. Сега вече m съдържа броя на появите на модата на $A[1, \dots, i]$. Преди следващото достигане на ред 4, i се инкрементира. Спрямо новото i е вярно, че m съдържа броя на появите на модата на $A[1, \dots, i-1]$. ✓
- ③ Знаем, че $A[i-1]$ е мода на $A[1, \dots, i-1]$ преди инкрементирането на i . От това и от факта, че $A[i-1] = A[i]$ преди инкрементирането на i следва, че след инкрементирането на i отново е вярно, че $A[i-1]$ е мода на $A[1, \dots, i-1]$. Следователно, антецедентът на импликацията е лъжа, а импликацията е истина. ✓

Случай I.2 $A[i-1]$ не е мода на $A[1, \dots, i-1]$. Тогава, съгласно индуктивното предположение, $A[i-1] \neq mode$.

Първо ще покажем, че част ③ на инвариантата се запазва и при двете възможности—лъжа или истина—за булевото условие на ред 10. Тези две възможности отговарят съответно на Q_2 : $A[i]$ разширява група-немода и Q_3 : $A[i]$ превръща група в група-мода. Щом $A[i-1]$ не е мода на $A[1, \dots, i-1]$, то част ③ от индуктивното предположение казва, че s съдържа дълчината на най-дясната група в $A[1, \dots, i-1]$. Става дума за момента, в който изпълнението е на ред 8. Изпълнява се ред 9, където s се инкрементира. Тъй като $A[i-1] = A[i]$, очевидно новата стойност на s съдържа дълчината на най-дясната група в $A[1, \dots, i]$. Независимо от това дали булевото условие на ред 10 е истина или лъжа, s не се променя до следващото достигане на ред 4 (редове 11 и 12 не променят s). Точно преди следващото достигане на ред 4, i се инкрементира. Спрямо новата стойност на i е вярно, че s съдържа дълчината на най-дясната група в $A[1, \dots, i-1]$. И така, от една страна текущото $A[i]$ не е мода на текущия $A[1, \dots, i-1]$, а от друга страна s съдържа дълчината на най-дясната група в текущото $A[1, \dots, i-1]$. Следва, че част ③ на инвариантата се запазва в **Случай I.2**. ✓

Да се върнем към момента, в който изпълнението беше на ред 8. Сега ще разгледаме части ① и ② на инвариантата. Помним, че променливата s , съгласно индуктивното предположение, съдържа дълчината на най-дясната група на $A[1, \dots, i-1]$. Изпълнението отива на ред 9, където s се инкрементира. Тъй като $A[i-1] = A[i]$, то новата стойност на s съдържа дълчината на най-дясната група в $A[1, \dots, i]$. Изпълнението отива на ред 10.

Случай I.2.a $A[i]$ не е мода на $A[1, \dots, i]$. В текущия контекст, това е същото като да кажем, че най-дясната група на $A[1, \dots, i]$, чиято дължина се съдържа в променливата s , има по-голяма дължина от броя на появите на модата на $A[1, \dots, i]$, който брой

се съдържа в променлива m поради индуктивното предположение и това, че $A[i]$ не е мода на $A[1, \dots, i]$. С други думи, **Случай I.2.a** отговаря на Q_2 : $A[i]$ разширява група-немода. Щом $s \leq m$, то условието на ред 10 е лъжа и изпълнението се връща на ред 4. $mode$ и m остават непроменени. Променливата i се инкрементира. Спрямо новата ѝ стойност, $mode$ съдържа модата на $A[1, \dots, i - 1]$, а m съдържа броя на появите ѝ. Показваме, че части ① и ② на инвариантата се запазват.

Случай I.2.b $A[i]$ е мода на $A[1, \dots, i]$. В текущия контекст, това е същото като да кажем, че най-дясната група на $A[1, \dots, i]$, чиято дължина се съдържа в инкрементираната на ред 9 променлива s , има по-голяма дължина (с единица) от броя на появите на модата на $A[1, \dots, i - 1]$, който брой се съдържа в променлива m (от индуктивното предположение). Тогава текущото s съдържа броя на появите на модата на $A[1, \dots, i]$. И така, $A[i - 1]$ не е мода на $A[1, \dots, i - 1]$, но същата стойност ($A[i]$ е равна на $A[i - 1]$) е мода на $A[1, \dots, i]$. С други думи, **Случай I.2.b** отговаря на Q_3 : $A[i]$ превръща група в група-мода. Щом $s > m$, то условието на ред 10 е истина и се изпълняват редове 11 и 12. Сега $mode$ съдържа модата на $A[1, \dots, i]$, а m съдържа броя на появите ѝ. После променливата i се инкрементира. Спрямо новата ѝ стойност, $mode$ съдържа модата на $A[1, \dots, i - 1]$, а m съдържа броя на появите ѝ. Показваме, че части ① и ② на инвариантата се запазват.

Случай II $A[i - 1] \neq A[1]$ (изпълнението е на ред 5). Лесно се вижда, че в този случай $A[i]$ не може да е мода на $A[1, \dots, i]$. Следователно, модата на $A[1, \dots, i]$ е същата като на $A[1, \dots, i - 1]$, и броят на появите ѝ в $A[1, \dots, i]$ е същият като в $A[1, \dots, i - 1]$. Съгласно индуктивното предположение, $mode$ съдържа модата на $A[1, \dots, i]$, а m съдържа броя на появите ѝ $A[1, \dots, i]$.

Условието на ред 5 е лъжа и изпълнението отива на ред 13, след което на ред 14, променливата s става единица и изпълнението се връща на ред 4, като i се инкрементира. Спрямо новата стойност на i :

- ① $mode$ е мода на $A[1, \dots, i - 1]$. ✓
- ② m съдържа броя на появите на модата на $A[1, \dots, i - 1]$. ✓
- ③ От една страна, $A[i - 1]$ не е мода на $A[1, \dots, i - 1]$ (говорим за новата стойност на i), от друга страна най-дясната група на $A[1, \dots, i - 1]$ спрямо новата стойност на i има дължина единица, а от трета страна s съдържа единица. ✓

□

Модата не може да бъде изчислена във време, което не е $\Omega(n \lg n)$. Това е доказано в Подсекция 7.3.2.

3.2.5 Медиана

Нека е даден масив от n стойности. *Медиана* е стойността, от която половината стойности са по-малки или равни, а другата половина, по-големи или равни. Строго погледнато, тази дефиниция на медиана е смислена само когато броят на стойностите е нечетно число и под “половината стойности” разбираме “ $\lfloor \frac{n}{2} \rfloor$ стойности”. Следното определение е смислено за произволен брой на елементите.

Определение 18: Медиана

Нека $A[1, \dots, n]$ е масив от цели числа. *Медианата* на A е елементът $A[\lfloor \frac{n+1}{2} \rfloor]$ след сортиране на A .

Съществува още по-детайлно определение на “медиана”. Тъй като при четно n има два естествени кандидата за медиана, това определение казва, че масивите с четен брой елементи имат две медиани: елементът на позиция $\lfloor \frac{n+1}{2} \rfloor$ в сортирания масив е *долната медиана*, а този на позиция $\lceil \frac{n+1}{2} \rceil$ в сортирания масив е *горната медиана*. Вижте [15, стр. 213].

Медиана се пресмята лесно, ако използваме сортиране, понеже самата дефиниция на медиана използва сортиране неявно: медианата е елементът, стоящ в средата на сортираната последователност на тези елементи. В следния алгоритъм $(n/2)$ означава целочислено деление.

COMPUTE MEDIAN($A[1, 2, \dots, n]$, n е нечетно)

```
1 SORT(A)
2 return A[(n/2) + 1]
```

Заслужава да се отбележи, че медианата може да бъде изчислена в $\Theta(n)$ чрез алгоритъма, намиращ k -тото по големина число между n дадени числа в *линейно време* за произволно k . Този алгоритъм може да бъде намерен в [15], а и ние ще го дискутираме в следваща лекция, която разглежда алгоритми, изградени по схемата Разделяй-и-Владей.

3.3 Анализ на сортиращи алгоритми. Стабилност.

Анализът на даден сортиращ алгоритъм се състои в това, което избрахме миналата лекция за анализа на произволен алгоритъм: доказателство на коректността и анализ на сложността по време и по памет. В контекста на сортиращите алгоритми има още едно свойство, което представлява интерес. То се нарича *стабилност*. Сега ще обясним в детайли какво означава сортиращ алгоритъм да бъде стабилен с един пример[†] от [38].

[†]Кнут означава с N броя на елементите във входа, а сортираната пермутация с “ $K_{p(1)} \leq K_{p(2)} \leq \dots \leq K_{p(N)}$ ”.

Задача 3: задача 2 на стр. 5 в [38]

Да допуснем, че всеки запис R_j в даден файл съдържа два ключа, първичен ключ K_j и вторичен ключ k_j , като върху ключовете е дефинирана линейна наредба $<$. Дефинираме *лексикографска наредба* върху ключовете по стандартния начин:

$$(K_i, k_i) < (K_j, k_j) \text{ ако } K_i < K_j \text{ или } K_i = K_j \text{ и } k_i < k_j$$

Alice взема файла и го сортира първо по първичните ключове, получавайки n групи от записи с еднакви първични ключове във всяка група:

$$K_{p(1)} = \dots = K_{p(i_1)} < K_{p(i_1+1)} = \dots = K_{p(i_2)} < \dots < K_{p(i_{n-1}+1)} = \dots = K_{p(i_n)},$$

където $i_n = N$. След това тя сортира всяка от n на брой групите по вторичния ключ. Bill взема оригиналния файл и първо го сортира по вторичния ключ, а след това, по първичния.

Chris взема оригиналния файл и прави едно единствено сортиране, като обаче ползва и двата ключа с лексикографската наредба.

Дали всеки от тях е получил непременно един и същи резултат?

Отговорът е, със сигурност Alice и Chris ще получат един и същи резултат. Резултатът на Bill ще е гарантирано същият само при условие, че неговото второ сортиране—това по първичния ключ—не променя относителния ред на записи, имащи еднакъв (първичен) ключ, който ред е създаден от вече извършеното първо сортиране. Ето малък пример: нека наредените двойки от ключовете са $(1, 2), (5, 2), (4, 5), (2, 5), (3, 1), (1, 4), (3, 2)$. Сортираната с лексикографска наредба последователност е:

$$(1, 2), (1, 4), (2, 5), (3, 1), (3, 2), (4, 5), (5, 2)$$

Това е резултатът, който ще получат Alice и Chris след своите сортириания.

Какво точно ще получи Bill след **първото** сортиране—това по вторичните ключове—зависи от сортирация алгоритъм, който ползва. Със сигурност

- на първа позиция ще е наредената двойка $(3, 1)$,
- после ще са трите наредени двойки с втори елемент 2, но не е ясно в какъв ред
- после $(1, 4)$,
- и накрая двете наредени двойки с втори елемент 5, но не е ясно в какъв ред.

Примерно, Bill може да получи това след първото сортиране:

$$(3, 1), (5, 2), (3, 2), (1, 2), (1, 4), (4, 5), (2, 5)$$

но може да получи и това:

$$(3, 1), (3, 2), (5, 2), (1, 2), (1, 4), (2, 5), (4, 5)$$

Сега Bill прави **второ** сортиране, този път по първичните ключове. В получената наредба на първите две места са наредените двойки $(1, 2)$ и $(1, 4)$. Но забележете, че при произволно

сортиране по първи ключ няма как да знаем дали $(1, 2)$ ще е вляво от $(1, 4)$ или не. От гледна точка на коректна лексикографска сортировка, разбира се, искаме началото да бъде

$(1, 2), (1, 4), \dots$

но може да се получи и

$(1, 4), (1, 2), \dots$

тъй като при сортирането по първични ключове не “тледа” вторичните ключове. **По отношение на сортирането само по първични ключове, $(1, 2)$ и $(1, 4)$ са еднакви елементи!** Първата фаза—сортирането по вторични ключове—със сигурност слага $(1, 2)$ преди $(1, 4)$, защото $2 < 4$, но втората фаза може потенциално да сложи $(1, 4)$ вляво от $(1, 2)$, ако ги счита за еднакви.

Определение 19: Стабилно сортиране

Стабилно сортиране е сортиране, което не променя взаимното разположение на елементи с еднакви ключове.

При стабилно сортиране, за всеки два елемента с еднакви ключове, които елементи са все пак различими (заради, примерно, вторични ключове), този, който е вляво от другия **преди** сортирането, задължително ще е вляво от него и **след** сортирането.

Наблюдение 9

Всеки нестабилен сортиращ алгоритъм може да бъде направен стабилен, ако се добави първоначалната позиция на всеки елемент като вторичен ключ^a и се сортира по наредените двойки (първичен ключ, вторичен ключ) – като Chris от Задача 3. Ако направим това обаче, може да влошим сложността по памет, понеже $\Theta(n)$ допълнителна памет ще се използва от новите ключове, използвани за стабилизиране. Ако преди това изкуствено стабилизиране сложността е била $\Theta(1)$, ще я влошим до $\Theta(n)$. Следователно, няма как да използваме този трик върху алгоритъм, който е in-place и той да остане in-place.

^aТова е в случай, че поначало има само един ключ. Ако поначало има k ключа, добавяме още един, $k + 1$ -ви, най-младши ключ с първоначалните позиции.

3.4 Елементарни Сортиращи Алгоритми

Разделянето на сортиращите алгоритми на елементарни и други, неелементарни, е доста условно. Под елементарни сортиращи алгоритми разбираме такива, които биха хрумнали бързо на човек, който за пръв път се сблъсква със СОРТИРАНЕ. По правило те работят във време $\Theta(n^2)$ в най-лошия случай и не използват никакви изтънчени структури данни или нетривиални идеи.

3.4.1 INSERTION SORT

INSERTION SORT сортира по начин, който далечно напомня на начина, по който картоиграч сортира картите, които държи в ръка (които са в произволна наредба в началото) –

плъзгайки поглед отляво надясно, той или тя разглежда последователно картите. При това всяка карта, която е обект на внимание в момента, бива сложена на правилното ѝ място в подпоследователността от картите вляво. Има една важна разлика между този начин на нареждане и работата на алгоритъма INSERTION SORT: карта може да бъде пъхната между другите карти мигновено, докато ако искаме да “пъхнем” дадено a_i някъде вляво, първо трябва да му направим място, за да не бъде затрит елементът, който вече е там.

Да си припомним псевдокода на INSERTION SORT

INSERTION SORT($A[1, 2, \dots, n]$: array of integers)

```

1   for i ← 2 to n
2       key ← A[i]
3       j ← i - 1
4       while j > 0 and A[j] > key do
5           A[j + 1] ← A[j]
6           j ← j - 1
7       A[j + 1] ← key

```

Коректност на INSERTION SORT

Доказателството за коректност ще направим чрез *инвариант на цикъла*. Тъй като сега за първи път ще демонстрираме тази техника, ще обясним подробно как става доказателство чрез инвариант. Да се върнем на доказателството на коректността на INSERTION SORT.

Целта е да докажем, че INSERTION SORT сортира всеки свой вход.

Допълнение 16: Неформално за доказателствата за коректност

Доказателствата за коректност на алгоритми са тежки, защото обектите, за които трябва да изказваме и доказваме твърдения, са динамични. Променливите се менят с работата на алгоритъма заради присвояванията. Ако в доказателството за коректност на INSERTION SORT говорим за $A[i]$, какво имаме предвид? Дали елемента в клетка i на входа, тоест преди началото на алгоритъма? Или елемента, който е бил в клетка i на масива в началото на текущата итерация на външния цикъл? Или имаме предвид клетка i в даден конкретен момент от изпълнението, който момент разглеждаме в доказателството? В общия случай това са **различни** елементи.

В това допълнение ще дадем интуицията зад формалното доказателство за коректността, а Лема 15 и Теорема 17 ще бъдат сухото и формално доказателство за коректност.

Най-общо казано, схемата на доказателството на коректността на INSERTION SORT е следната. Това, което искаме да докажем е, че алгоритъмът терминира върху всеки вход и винаги, когато терминира, масивът A се състои от началните елементи, но в сортиран вид. Това е прекалено общо и е в сила за всеки сортиращ алгоритъм.

Сега да говорим конкретно за INSERTION SORT. По-детайлно казано, схемата за доказване на коректността му е такава. Разглеждаме **само едно изпълнение** на външния цикъл (редове 1–7). По отношение на **това изпълнение** на външния цикъл, ефектът от работата на алгоритъма е следният. Ако в начало му е вярно, че:

- подмасивът $A[1, \dots, i - 1]$ се състои от същите елементи като входния подмасив $A[1, \dots, i - 1]$, но в сортиран вид

- и подмасивът $A[i, \dots, n]$ е същият като входния $A[i, \dots, n]^a$,

то в самия му край—това означава, точно след изпълнението на ред 7, но преди следващото инкрементиране на управляващата променлива i на ред 1—е вярно, че:

- подмасивът $A[1, \dots, i]$ се състои от същите елементи като входния $A[1, \dots, i]$, но в сортиран вид;
- и подмасивът $A[i + 1, \dots, n]$ е същият като входния $A[i + 1, \dots, n]$.

Разговорно казано, сортираната зона в A “върви” от левия край надясно и на всяко изпълнение на външния цикъл нараства с една клетка, но задължително се състои от същите елементи, които са били в нея в началото на алгоритъма; а частта извън сортираната зона, която пък намалява с една клетка, също се състои от същите елементи, които са били там в началото.

И всичко това формулирахме без изобщо да разглеждаме вътрешния цикъл. Дали то е приемливо за доказателство за коректност на алгоритъма или не, зависи от това, колко прецизно доказателство искаме. Посоченото доказателство е прекалено недетайлно и неформално и, ако искаме голяма прецизност, то е незадоволително – не е доказано, че ефектът от едно изпълнение на външния цикъл е такъв, какъвто тук се твърди, че е. Това не е доказателство, а схема, по която може да бъде направено детайлно доказателство.

Сега ще обясним как става нарастването на сортираната зона с една позиция. Разглеждаме в детайли **едно изпълнение на външния цикъл**. Допускаме, че в началния му момент подмасивът $A[1, \dots, i - 1]$ е сортиран и се състои точно от елементите, които са били на позиции $1, \dots, i - 1$ във входа. В този момент $A[i]$ може да “не си е на мястото” в $A[1, \dots, i]$, защото $A[i]$ може да е по-малък от някакви елементи от $A[1, \dots, i - 1]$. В първата стъпка от изпълнението на **for**-цикъла, в променливата *key* се слага $A[i]$ и после се изпълнява вътрешният цикъл (редове 4–7). Ефектът от работата на вътрешния цикъл е следният. Нека t_1 е моментът на първоначалното достигане на ред 4, тоест самото начало на изпълнението на вътрешния цикъл, а t_2 е моментът на достигане на ред 7, тоест веднага след излизането от вътрешния цикъл. И така, в t_2 :

- $A[1, \dots, j]$ е същият като $A[1, \dots, j]$ от t_1 и се състои точно от тези елементи на $A[1, \dots, i - 1]$ от t_1 , които са по-малки или равни на *key*, в сортиран вид;
- $A[j + 2, \dots, i]$ е същият като $A[j + 1, \dots, i - 1]$ от t_1 и се състои точно от тези елементи на $A[1, \dots, i - 1]$ от t_1 , които са по-големи от *key*, в сортиран вид;
- клетка $A[j + 1]$ може да се смята за празна, защото стойността, която е била в нея в t_1 , е запазена другаде. А именно,
 - ◆ в $A[j + 2]$, ако вътрешният цикъл е бил изпълнен поне веднъж, или
 - ◆ в *key*, в противен случай.

Разговорно казано, ако гледаме само $A[1, \dots, i]$, в момента t_2 , $j + 1$ е “правилното място” на елемента, който се е намирал в $A[i]$ в t_1 . “Правилното място” е по отношение на елементите, които са по-малки или равни на него. Те се намират в $A[1, \dots, j]$ в t_1 и не се местят от вътрешния цикъл⁶. Вътрешният цикъл пресмята това $j + 1$, като освен това

премества с единица надясно точно тези елементи на $A[1, \dots, i]$, които са по-големи от елемента $A[i]$ в t_1 , без да ги размества взаимно; тоест, мести ги като блок.

Това вече е много по-детайлно обяснение, но все още не е достатъчно детайлно – все още не сме видели как действа една итерация на вътрешния цикъл. Сега разглеждаме в детайли **едно изпълнение на вътрешния цикъл**. Нека t_1 и t_2 имат същото значение, каквото имаха досега, и нека t' е моментът на някое достигане на ред 4, което не е последното, а t'' е следващото достигане на ред 4 (очевидно $t_1 < t' < t'' < t_2$). Ако в t' :

- $A[j + 2, \dots, i]$ е сортиран и всеки елемент от него е по-голям от key и
- $A[1, \dots, j]$ е сортиран и е същият като в t_1 и
- клетка $A[j + 1]$ може да се счита за празна

то в t'' :

- $A[j + 1, \dots, i]$ е сортиран и всеки елемент от него е по-голям от key и
- $A[1, \dots, j - 1]$ е сортиран и е същият като в t_1 и
- клетка $A[j]$ може да се счита за празна

Това вече е пълна схема на доказателство, което може да бъде извършено по индукция. Естественият ред на формалното доказателство по отношение на вложените цикли е, отвътре навън.

^aТова е абсолютно очевидно от кода на алгоритъма, но за пълнота го формулираме отделно.

^bЗабележете, че $j + 1$ е правилното място за $A[i]$ именно по отношение на елементите, които са по-малки или равни на i и на него. По отношение на елементите, които са по-големи от него, правилното място е j , а не $j + 1$. Но това няма значение, защото сме избрали да мести елементите, които са по-големи от $A[i]$, а да оставяме на място тези, които са по-малки или равни.

Навсякъде в доказателствата долу допускаме, че $n > 1$. Използваме израза “клетка номер x на масива Y може да бъде смятана за свободна” – това означава, че съдържанието ѝ е запазено другаде и може да бъде презаписана, без първоначално намиращият се там елемент да бъде унищожен безвъзвратно.

Лема 15

Спрямо едно единствено изпълнение на **for** цикъла на INSERTION SORT (редове 1–7), нека наричаме масива A непосредствено преди изпълнението на **while** цикъла (редове 4–6) с името A' . Да допуснем, че подмасивът $A'[1, \dots, i-1]$ е сортиран. Тогава в момента на приключване на **while** цикъла е вярно, че:

- ① $A[1, \dots, j]$ се състои точно от елементите на $A'[1, \dots, i-1]$, които са по-малки или равни на key , в сортиран вид;
- ② $A[j+2, \dots, i]$ се състои точно от елементите на $A'[j+1, \dots, i-1]$, които са по-големи от key , в сортиран вид;
- ③ клетката $A[j+1]$ е свободна;
- ④ $A[i+1, \dots, n]$ е същият като $A'[i+1, \dots, n]$.

Доказателство: Следното твърдение е инвариантта за **while** цикъла.

Инвариант 2: Вътрешният цикъл на INSERTION SORT

Всеки път, когато изпълнението е на ред 4:

- ❶ $A[1, \dots, j] = A'[1, \dots, j]$;
- ❷ $A[j+2, \dots, i] = A'[j+1, \dots, i-1]$;
- ❸ $\forall x \in A[j+2, \dots, i] : x > key$;
- ❹ $A[i+1, \dots, n] = A'[i+1, \dots, n]$.

Както казахме вече на стр. 27, доказателството на инвариантата е доказателство по индукция. Започва с **база**, в която верността на предиката просто се проверява. В обикновените доказателства по индукция следващите две фази са, индуктивно предположение и индуктивна стъпка. Тук ги сливаме в една фаза от доказателството, която наричаме **поддръжка**. И накрая има последна фаза **терминация**, която няма еквивалент при “обикновените” доказателства по индукция, които са върху безкрайни индуктивни дефинирани множества. При доказателствата за коректност на алгоритми винаги доказваме твърдение върху крайно индуктивно генерирано множество, твой като алгоритмите винаги терминират. Терминацията се отнася до последното достигане на условието на цикъла – когато тялото няма да се изпълни нито веднъж повече. Когато заместим стойността на управляващата променлива от този момент в инвариантата, трябва да получим точно това твърдение, което ни трябва за доказателство на коректността.

База. Ще покажем ❶. Когато изпълнението достигне до ред 4 за първи път, текущият масив A е A' по определение, така че наистина $A[1, \dots, j] = A'[1, \dots, j]$.

Ще покажем ❷. Когато изпълнението достигне до ред 4 за първи път, $j = i - 1$ заради присвояването на ред 3. Тогава $A[j+2, \dots, i]$ въщност е $A[i+1, \dots, i]$, който подмасив е празен, а $A'[j+1, \dots, i-1]$ въщност е $A'[i, \dots, i-1]$, който подмасив също е празен.

❸ е изпълнено, защото $A[j+2, \dots, i]$ е празен, а за всеки елемент на празното множество е изпълнен всеки предикат. ❹ е очевидно.

Поддръжка. Да допуснем, че твърдението е в сила в даден момент t , в който изпълнението е на ред 4 и **while** ще бъде изпълнен поне още веднъж. Последното влече, че $j > 0$ и $A[j] > key$ в t . Нека t_{next} означава момента на следващото достигане на ред 4.

Ще покажем, че ❶ е в сила в t_{next} . Първо разглеждаме присвояването на ред 5. То не променя нищо в подмасива $A[1, \dots, j - 1]$ и той остава равен на $A'[1, \dots, j - 1]$. Сега разглеждаме декрементирането на ред 6. Непосредствено след него, спрямо новата стойност на j , същият този подмасив може да бъде означен с “ $A[1, \dots, j]$ ”, а също така и с “ $A'[1, \dots, j]$ ”. И така, подмасивът $A[1, \dots, j]$ в t_{next} е същият като $A'[1, \dots, j]$ и е сортиран.

Ще покажем, че ❷ е в сила в t_{next} . Първо разглеждаме присвояването на ред 5. От индуктивното предположение ❸ знаем, че $A[j + 2, \dots, i] = A'[j + 1, \dots, i - 1]$. От индуктивното предположение ❶ следва, че $A[j] = A'[j]$. Тогава след $A[j + 1] \leftarrow A[j]$ на ред 5 е вярно, че $A[j + 1, j + 2, \dots, i] = A'[j, j + 1, \dots, i - 1]$. Обаче подмасивът $A'[j, j + 1, \dots, i - 1]$ е сортиран по допускане, така че текущият подмасив $A[j + 1, j + 2, \dots, i]$ е сортиран. Сега разглеждаме декрементирането на ред 6. Непосредствено след него, спрямо новата стойност на j е вярно, че $A[j + 2, \dots, i] = A'[j + 1, \dots, i - 1]$ и това е сортиран подмасив. В t_{next} това остава вярно.

Ще покажем, че ❹ е в сила в t_{next} . Допуснали сме, че $\forall x \in A[j + 2, \dots, i] : x > key$ в t . Знаем, че $A[j] > key$, иначе тази итерация на **while**-а не би се изпълнявала. Първо разглеждаме присвояването на ред 5. Непосредствено след него е вярно, че $\forall x \in A[j + 1, \dots, i] : x > key$. Сега разглеждаме декрементирането на ред 6. След него, спрямо новата стойност на j е вярно, че $\forall x \in A[j + 2, \dots, i] : x > key$ при следващото достигане на ред 4.

Това, че ❻ е в сила при следващото достигане на ред 4, е очевидно.

Терминация. Да разгледаме момента, в който изпълнението е на ред 4 и условието там е FALSE. Тогава $j \leq 0$ или $key \geq A[j]$.

Случай 1: $j \leq 0$. Тъй като j намалява с единица, то не може да стане отрицателно, следователно $j = 0$. Заместваме j с 0 в инвариантата и получаваме:

- ❶ $A[1, \dots, 0] = A'[1, \dots, 0]$;
- ❷ $A[2, \dots, i] = A'[1, \dots, i - 1]$;
- ❸ $\forall x \in A[2, \dots, i] : x > key$;
- ❹ $A[i + 1, \dots, n] = A'[i + 1, \dots, n]$.

Очевидно в този случай в $A'[1, \dots, i - 1]$ няма елементи, по-малки или равни на key . Но тогава е вярно, че

- ❶ $A[1, \dots, 0]$ се състои точно от елементите на $A'[1, \dots, i - 1]$, които са по-малки или равни на key , в сортиран вид; // става дума за празен подмасив
- ❷ $A[2, \dots, i]$ се състои точно от елементите на $A'[1, \dots, i - 1]$, които са по-големи от key , в сортиран вид; // става дума за целия $A'[1, \dots, i - 1]$
- ❸ клетката $A[1]$ е свободна;
- ❹ $A[i + 1, \dots, n]$ е същият като $A'[i + 1, \dots, n]$.

Което е точно твърдението на лемата при $j = 0$.

Случай 1: Сега да допуснем, че $j > 0$. Тогава $key \geq A[j]$. Да разгледаме отново инвариантата. Тя е:

- ❶ $A[1, \dots, j] = A'[1, \dots, j]$;

- ❷ $A[j+2, \dots, i] = A'[j+1, \dots, i-1]$;
- ❸ $\forall x \in A[j+2, \dots, i] : x > key$;
- ❹ $A[i+1, \dots, n] = A'[i+1, \dots, n]$.

От ❶ следва, че $A[j] = A'[j]$. Но тогава $key \geq A'[j]$. Щом $A[1, \dots, j] = A'[1, \dots, j]$ и $A[1, \dots, i-1]$ е сортиран, то $A[1, \dots, j]$ е сортиран. Щом $A[1, \dots, j]$ е сортиран и $key \geq A'[j]$, то $\forall x \in A[1, \dots, j] : x \leq key$. От това и ❷ и ❸ следва, че

- ❶ $A[1, \dots, j]$ се състои точно от елементите на $A'[1, \dots, i-1]$, които са по-малки или равни на key , в сортиран вид.

Да разгледаме пак ❷ и ❸. От тях и ❶ и фактът, че $A'[1, \dots, i-1]$ е сортиран следва, че

- ❷ $A[j+2, \dots, i]$ се състои точно от елементите на $A'[1, \dots, i-1]$, които са по-големи от key , в сортиран вид.

От ❶, ❷ и присвояването на ред ❲ следва, че

- ❸ клетката $A[j+1]$ е свободна.

Това, че ❹ е вярно, е очевидно. Доказахме лемата. □

Теорема 17

INSERTION SORT е сортиращ алгоритъм.

Доказателство: Да наречем входа A'' . Следното твърдение е инварианта за **for** цикъла:

Инвариантът 3: Външният цикъл на INSERTION SORT

Всеки път, когато изпълнението на INSERTION SORT е на ред ❶, текущият подмасив $A[1, \dots, i-1]$ се състои точно от същите елементи като $A''[1, \dots, i-1]$, но в сортиран вид. Освен това, текущият подмасив $A[i, \dots, n]$ се състои от точно от същите елементи като $A''[i, \dots, n]$.

База. Първият път, когато изпълнение достигне ред ❶ е вярно, че $i = 2$. Текущият подмасив $A[1, \dots, 1]$ се състои от един единствен елемент $A''[1]$ и е тривиално сортиран. Освен това, текущият подмасив $A[2, \dots, n]$ се състои от точно от същите елементи като $A''[2, \dots, n]$. ✓

Поддръжка. Да допуснем, че твърдението е изпълнено при някое достигане на ред ❶ и **for** цикълът ще бъде изпълнен още един път. Нека \tilde{A} е името на A при **това достигане** на ред ❶. Индуктивното допускане, в изразено чрез \tilde{A} , е:

$\tilde{A}[1, \dots, i-1]$ се състои точно от същите елементи като $A''[1, \dots, i-1]$, но в сортиран вид, и $\tilde{A}[i, \dots, n]$ се състои от точно от същите елементи като $A''[i, \dots, n]$.

Изпълнява се ред ❲ и сега key съдържа $\tilde{A}[i]$. Съгласно индуктивното допускане, това е еквивалентно на твърдението, че сега key съдържа $A''[i]$. Следва $j \leftarrow i-1$. После се изпълнява **while**-цикълът. Изпълнението на **while**-цикъла терминира. В момента, в който **while** цикълът терминира, съгласно Лема 15 са в сила следните твърдения:

- ❶ $A[1, \dots, j]$ се състои точно от елементите на $\tilde{A}[1, \dots, i-1]$, които са по-малки или равни на key , в сортиран вид;

- ② $A[j+2, \dots, i]$ се състои точно от елементите на $\tilde{A}[j+1, \dots, i-1]$, които са по-големи от key , в сортиран вид;
- ③ клетката $A[j+1]$ е свободна;
- ④ $A[i+1, \dots, n]$ е същият като $\tilde{A}[i+1, \dots, n]$.

Имайки предвид индуктивното предположение, тези твърдения стават:

- ① $A[1, \dots, j]$ се състои точно от елементите на $A''[1, \dots, i-1]$, които са по-малки или равни на key , в сортиран вид;
- ② $A[j+2, \dots, i]$ се състои точно от елементите на $A''[1, \dots, i-1]$, които са по-големи от key , в сортиран вид;
- ③ клетката $A[j+1]$ е свободна;
- ④ $A[i+1, \dots, n]$ е същият като $A''[i+1, \dots, n]$.

Тъй като в key вече се намира съдържанието на $A''[i]$, можем да твърдим, че в момента след присвояването на ред 7, текущият $A[1, \dots, i]$ се състои от същите елементи като $A''[1, \dots, i]$, но в сортиран вид. Изпълнението отива отново на ред 1 и i се инкрементира. Спрямо новата стойност на i е вярно, че:

текущият $A[1, \dots, i]$ се състои от същите елементи като $\tilde{A}[1, \dots, i]$, а $A[1, \dots, i-1]$ се състои от същите елементи като $A''[1, \dots, i-1]$.

И така, инвариантата се запазва.

Терминация. Да разгледаме момента, в който изпълнението е на ред 1 за последен път. Очевидно i е равно на $n + 1$. Заместваме i с $n + 1$ в инвариантата и получаваме “текущият подмассив $A[1, \dots, (n + 1) - 1]$ се състои от същите елементи като $A''[1, \dots, (n + 1) - 1]$, но в сортиран вид”. Накракто, “текущият подмассив $A[1, \dots, n]$ се състои от същите елементи като $A''[1, \dots, n]$, но в сортиран вид”. \square

Сложност по време на INSERTION SORT

Вече показвахме в (2.2) на стр. 40, че сложността по време в най-лошия случай на INSERTION SORT е $\Theta(n^2)$.

Сложност по памет на INSERTION SORT

INSERTION SORT очевидно ползва $\Theta(1)$ допълнителна памет.

Стабилност на INSERTION SORT

INSERTION SORT е стабилен сортиращ алгоритъм. Прецизното доказателство остава за упражнение. Тук само ще споменем, че алгоритъмът е стабилен заради използването на строго неравенство $>$ на ред 4; while цикълът престава да се изпълнява за първата (максималната) стойност на j , за която $A[j]$ става равен на key и заради това всички двойки равни елементи остават в същия относителен ред, в какъвто са били в началото. Ако наместо строго равенство ползвахме \geq на това място, алгоритъмът нямаше да е стабилен.

3.4.2 SELECTION SORT

Да си припомним псевдокода на SELECTION SORT.

SELECTION SORT($A[1, 2, \dots, n]$: array of integers)

```

1   for i ← 1 to n - 1
2       for j ← i + 1 to n
3           if A[j] < A[i]
4               swap(A[i], A[j])

```

Коректност на SELECTION SORT

Следващите две леми обосновават коректността на SELECTION SORT. Това, че SELECTION SORT само размества елементите на входа и в края на алгоритъма елементите в масива са точно тези, които са били в началото, е очевидно: единствените промени в A стават на ред 4 чрез размени (swaps). Това отличава тази реализация на SELECTION SORT от реализацията на INSERTION SORT, която вече разглеждахме. В последната не беше очевидно, че елементите в края са точно тези, които са били в началото, защото реализацията на INSERTION SORT ползва не размени, а присвоявания. Поради това при анализа на INSERTION SORT се наложи да обосновем защо нито един от оригиналните елементи не се “туби”, бивайки презаписан с нещо друго. При анализа на SELECTION SORT нямаме такива притеснения.

Лема 16

По отношение на едно изпълнение на външния **for** цикъл (редове 1–4) на SELECTION SORT, изпълнението на вътрешния **for** цикъл (редове 2–4) има следния ефект: $A[i]$ е най-малък елемент в $A[i, \dots, n]$.

Доказателство:

По отношение на едно изпълнение на външния **for** цикъл, следното твърдение е инвариант за вътрешния **for** цикъл:

Инвариант 4: Вътрешният цикъл на SELECTION SORT

Всеки път, когато изпълнението достигне ред 2, текущият $A[i]$ е минимален елемент в $A[i, \dots, j - 1]$.

База. Първият път, когато изпълнението на вътрешния **for** цикъл е на ред 2 е вярно, че $j = i + 1$. Тогава $A[i]$ е тривиално най-малък елемент в $A[i, \dots, (i + 1) - 1]$.

Поддръжка. Да допуснем, че твърдението е вярно в някой момент, когато изпълнението е на ред 2 и предстои вътрешният **for** цикъл да бъде изпълнен още веднъж. Следните два случая са изчерпателни.

Случай 1. $A[j] < A[i]$. Условието на ред 3 е TRUE и размяната на ред 4 се случва. Съгласно допускането, $A[i]$ е минимален елемент в $A[i, \dots, j - 1]$. Тъй като $A[j] < A[i]$, съгласно транзитивността на релацията $<$, $A[j]$ е минималният елемент в подмасива $A[i, \dots, j]$ преди размяната. Тогава $A[i]$ е минималният елемент в $A[i, \dots, j]$ след размяната. След размяната j нараства с единица и изпълнението отива на ред 2. По отношение на новата стойност на j е вярно, че $A[i]$ е минималният елемент в $A[i, \dots, j - 1]$.

Случай 2. $A[j] \leq A[i]$. Тогава условието на ред 3 е FALSE и размяната на ред 4 не се случва. Съгласно предположението, $A[i]$ е минимален елемент в $A[i, \dots, j - 1]$. Тъй като $A[i] \leq A[j]$,

очевидно $A[i]$ е минимален елемент в $A[i, \dots, j]$. После j нараства с единица и изпълнението отива на ред 2. По отношение на новата стойност на j е вярно, че $A[i]$ е минимален елемент в $A[i, \dots, j - 1]$.

Терминация. Да разгледаме момента, в който изпълнението е на ред 2 за последен път. Очевидно j е равно на $n + 1$. Заместваме $n + 1$ на мястото на j в инвариантата и получаваме “текущият $A[i]$ е минимален елемент в $A[i, \dots, (n + 1) - 1]$ ”. \square

Лема 17

SELECTION SORT е сортиращ алгоритъм.

Доказателство:

На наречем оригиналния масив $A'[1, \dots, n]$. Следното е инвариантта за външния **for** цикъл (редове 1–4):

Инвариант 5: Външният цикъл на SELECTION SORT

Всеки път, когато изпълнението на SELECTION SORT е на ред 1, текущият подмасив $A[1, \dots, i - 1]$ се състои от $i - 1$ на брой най-малки елементи на $A'[1, \dots, n]$ в сортиран вид.

База. При първото достигане на ред 1 е вярно, че $i = 1$. Текущият подмасив $A[1, \dots, i - 1]$ е празен и се състои от нула на брой най-малки елементи от $A'[1, \dots, n]$ в сортиран вид.

Поддръжка. Да допуснем, че твърдението е в сила при някакво достигане на ред 1, което не е последното. Нека наричаме масива A в този момент с името A'' . Съгласно Лема 16, ефектът на вътрешния **for** цикъл е, че поставя на i -тата позиция най-малка стойност от $A''[i, \dots, n]$. От друга страна, съгласно индуктивното допускане, $A''[1, \dots, i - 1]$ се състои от $i - 1$ на брой най-малки елементи на $A'[1, \dots, n]$ в сортиран вид. Заключаваме, че в края на изпълнението на външния **for** цикъл, текущият $A[1, \dots, i]$ се състои от i на брой най-малки елемента от $A'[1, \dots, n]$ в сортиран вид. После i нараства с единица и изпълнението отива на ред 1. По отношение на новата стойност на i е вярно, че текущият $A[1, \dots, i - 1]$ съдържа $i - 1$ на брой най-малки елемента от $A'[1, \dots, n]$ в сортиран вид.

Терминация. Да разгледаме момента, в който изпълнението е на ред 1 за последен път. Очевидно i е равно на n . Заместваме n на мястото на i в инвариантата и получаваме “текущият подмасив $A[1, \dots, n - 1]$ се състои от $n - 1$ на брой най-малки елемента на $A'[1, \dots, n]$ в сортиран вид”. Но тогава текущият $A[n]$ е максимален елемент от $A'[1, \dots, n]$. С това доказателството за коректност на SELECTION SORT приключва. \square

Сложност по време на SELECTION SORT

Вече показвахме в (2.4) на стр. 40, че сложността по време в най-лошия случай на SELECTION SORT е $\Theta(n^2)$.

Сложност по памет на SELECTION SORT

SELECTION SORT очевидно ползва $\Theta(1)$ допълнителна памет.

Стабилност на SELECTION SORT

В този си вид SELECTION SORT не е стабилен. За да се убедим в това, да разгледаме малък подходящ пример. Нека входът е

2, 3, 4, 2, 1

За да различаваме двойките, да ги означим така:

2', 3, 4, 2'', 1

Очевидно след първото изпълнение на външния **for** цикъл единицата ще отиде на най-лява позиция, бивайки разменена с 2':

1, 3, 4, 2'', 2'

и крайният резултат ще е

1, 2'', 2', 3, 4

Лекция 4

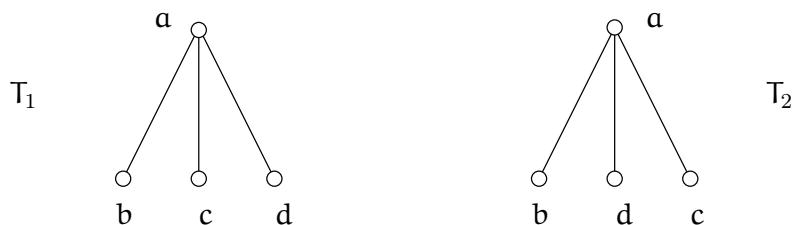
Двоична Пирамида. HEAPSORT. Приоритетна Опашка.

Резюме: Въвеждаме понятието попълнено двоично дърво и двоична пирамида. Показваме два начина да се строи двоична пирамида: наивен начин във време $\Theta(n \lg n)$ и бързото построяване във време $\Theta(n)$ на Floyd. Демонстрираме сортирация алгоритъм HEAPSORT и го анализираме. Въвеждаме понятието приоритетна опашка като абстрактен тип данни и разискваме имплементация на приоритетна опашка чрез двоична пирамида.

4.1 Двоични дървета и пирамиди

4.1.1 Попълнено двоично дърво

Тъй като теорията на графите е сравнително нова област, няма универсално приета пълна единна терминология и всяко сериозно изложение, използващо графови понятия, трябва да започва с изчерпателни определения, за да е ясно точно какво се има предвид. Терминологията на английски ще изложим според американския Национален Институт за Стандарти и Технологии [NIST](#). Техните определения за дървета са изложени на [тази страница](#). Не допускame дървета без върхове. Разглеждаме само коренови дървета, така че когато кажем “дърво” (tree), имаме предвид кореново дърво (rooted tree). В контекста на структури от данни по правило дърветата са *наредени*[†] (ordered trees), което означава, че децата на вътрешните върхове са наредени с линейна наредба. Следните две дървета T_1 и T_2 са различни като наредени дървета (наредбата е отляво надясно), но са едно и също дърво, ако ги разглеждаме като коренови дървета:



Ако T е кореново дърво с корен r , за всеки връх u в него с $T[u]$ бележим поддървото, вкоренено в u . Формално,

[†]За разлика от “чистата” теория на графите, в която по правило кореновите дървета не са наредени.

- ако $u = r$, то $T[u]$ е самото T ,
- иначе, $T[u]$ е това от двете дървета на $T - (u, \text{parent}(u))$, което съдържа u .

Височина на връх u в кореново дърво е максималното разстояние между u и кое да е листо в $T[u]$. *Височина на дървото* е височината на корена. *Дълбочина на връх u* в кореново дърво е разстоянието между него и корена. *Ниво* в дърво е множеството от всички върхове, които са на една и съща дълбочина. *Номер на ниво* в дърво е дълбочината на кой да е връх от това ниво. *Съседни нива* са нива, чиито дълбочини се различават с единица. *Двоично дърво* е кореново дърво (наредено или не), в което всеки връх има не повече от две деца.

Определение 20: попълнено двоично дърво, не-индуктивно определение

Попълнено двоично дърво (complete binary tree) е наредено двоично дърво, в което:

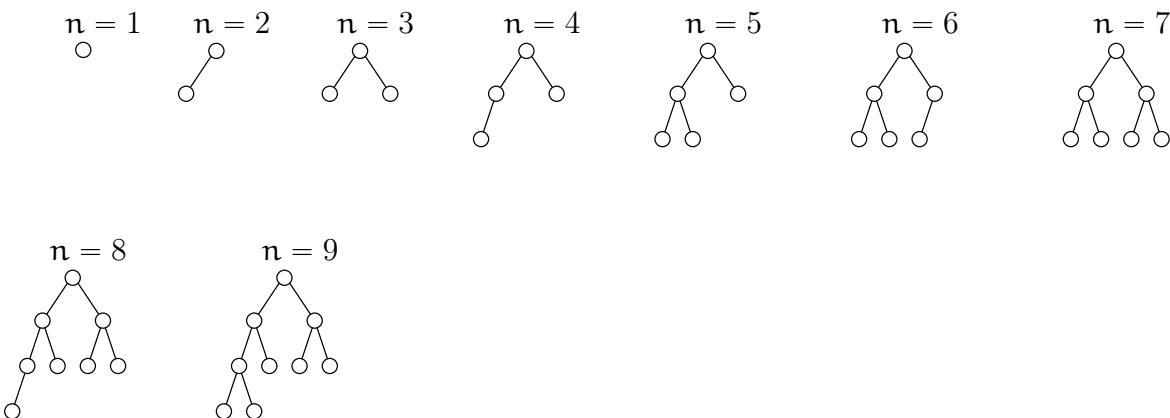
- всички нива, с изключение може би на последното, имат максималния възможен брой^a върхове^b.
- ако листата са на две нива, листата на последното ниво са максимално вляво. □

^a Ниво номер k може да има най-много 2^k върха в себе си.

^b Следствие от това е, че всички листа са или в едно ниво, или в две нива, които са съседни.

На английски терминът е *complete binary tree* ([49]).

За всяко $n \in \mathbb{N}^+$ има едно единствено попълнено дърво с n върха:



Съвършено двоично дърво (perfect binary tree) е попълнено двоично дърво, в което всички листа са на едно и също ниво [50].

Определение 20 е еквивалентно на Определение 21.

Определение 21: попълнено двоично дърво, индуктивно определение

Попълнено двоично дърво е всяко дърво, което може да бъде получено от краен брой прилагания на следните конструкции:

1. $(\{u\}, \emptyset)$ е попълнено двоично дърво с корен u , множество от листа $\{u\}$, множество от вътрешни върхове \emptyset и височина 0. u няма деца.
2. $(\{u, v\}, \{(u, v)\})$ е попълнено двоично дърво с корен u , множество от листа $\{v\}$, множество от вътрешни върхове $\{u\}$ и височина 1. v е лявото дете на u , а дясното дете липсва.
3. Нека T' е съвършено кореново дърво с корен u' , множество от листа W' , и височина $h - 1 \geq 0$. Нека T'' е попълнено кореново дърво с корен u'' , множество от листа W'' , и височина височина $h - 1$. Нека r е връх извън T' и T'' . Тогава

$$(V(T') \cup V(T'') \cup \{r\}, E(T') \cup E(T'') \cup \{(r, u'), (r, u'')\}),$$

където u' е лявото дете на r , а u'' е дясното дете на r , е попълнено кореново дърво с корен r , множество от листа $W' \cup W''$ и височина h .

4. Нека T' е попълнено кореново дърво с корен u' , множество от листа W' , и височина $h - 1 \geq 1$. Нека T'' е съвършено кореново дърво с корен u'' , множество от листа W'' , и височина $h - 2$. Нека r е връх извън T' и T'' . Тогава

$$(V(T') \cup V(T'') \cup \{r\}, E(T') \cup E(T'') \cup \{(r, u'), (r, u'')\}),$$

където u' е лявото дете на r , а u'' е дясното дете на r , е попълнено кореново дърво с корен r , множество от листа $W' \cup W''$ и височина h .

Доказателството за еквивалентност на Определение 20 и Определение 21 остават за читателя.

Както обикновено, пишейки $\lg n$, имаме предвид $\log_2 n$.

Лема 18

За всяко попълнено двоично дърво с n върха, за височината h е изпълнено $h = \lfloor \lg n \rfloor$.

Доказателство: Чрез структурна индукция съгласно Определение 21. В базовите случаи 1 и 2 твърдението е вярно: имаме $0 = \lfloor \lg 1 \rfloor$ и $1 = \lfloor \lg 2 \rfloor$. ✓

В случай 3, нека T' има p върха и T'' има q върха. Съгласно индуктивното предположение, $h - 1 = \lfloor \lg p \rfloor$ и $h - 1 = \lfloor \lg q \rfloor$. Но $\lfloor x \rfloor$ е най-голямото цяло число, ненадхвърлящо x , за всяко x . Можем да твърдим, че:

$$h - 1 \leq \lg p < h$$

$$h - 1 \leq \lg q < h$$

Тогава,

$$2^{h-1} \leq p < 2^h$$

$$2^{h-1} \leq q < 2^h$$

Тъй като p и q са цели числа, това е същото като:

$$\begin{aligned} 2^{h-1} &\leq p \leq 2^h - 1 \\ 2^{h-1} &\leq q \leq 2^h - 1 \end{aligned}$$

Тогава

$$\begin{aligned} 2 \times 2^{h-1} &\leq p + q \leq 2 \times 2^h - 2 & \leftrightarrow \\ 2^h &\leq p + q \leq 2^{h+1} - 2 & \leftrightarrow \\ 2^h &< p + q + 1 < 2^{h+1} \end{aligned}$$

Но $n = p + q + 1$. Тогава,

$$\begin{aligned} 2^h &< n < 2^{h+1} & \leftrightarrow \\ h &< \lg n < h + 1 \end{aligned}$$

Следователно, $h = \lfloor \lg n \rfloor$. ✓

В случай 4, нека T' има p върха и T'' има q върха. Съгласно индуктивното предположение, $h-1 = \lfloor \lg p \rfloor$ и $h-2 = \lfloor \lg q \rfloor$. Но q е число от вида $2^k - 1$ за някое $k \geq 0$, защото T'' е съвършено дърво. Тогава $q + 1 = 2^k$. Следователно, $\lfloor \lg (q + 1) \rfloor = \lfloor \lg q \rfloor + 1$, така че $h - 1 = \lfloor \lg (q + 1) \rfloor$. С разсъждения, аналогични на предния случай, получаваме

$$\begin{aligned} h - 1 &\leq \lg p < h \\ h - 1 &\leq \lg (q + 1) < h \end{aligned}$$

Тогава,

$$\begin{aligned} 2^{h-1} &\leq p < 2^h \\ 2^{h-1} &\leq q + 1 < 2^h \end{aligned}$$

Оттук имаме

$$\begin{aligned} 2^h &\leq p + q + 1 < 2^{h+1} & \leftrightarrow \\ 2^h &\leq n < 2^{h+1} \end{aligned}$$

Тогава,

$$h \leq \lg n < h + 1$$

Следователно, $h = \lfloor \lg n \rfloor$. ✓

□

Лема 19

Всяко попълнено двоично дърво T има точно $\lceil \frac{n}{2} \rceil$ листа.

Доказателство: Чрез структурна индукция съгласно Определение 21. За двата базови случаи твърдението е вярно: в случай 1, $n = 1$ и наистина има $\lceil \frac{1}{2} \rceil = 1$ листа, а в случай 2, $n = 2$ и наистина има $\lceil \frac{2}{2} \rceil = 1$ листа. ✓

Нека T е конструирано чрез случай 3. Нека T' има p върха и T'' има q върха. Очевидно $n = p + q + 1$. И T' , и T'' са попълнени дървета и индуктивното предположение тях казват,

че те имат съответно $\left\lceil \frac{p}{2} \right\rceil$ и $\left\lceil \frac{q}{2} \right\rceil$ листа. Наготово използваме факта, че T' има нечетен брой върхове, понеже е съвършено двоично дърво. Нека $p = 2k + 1$. Съгласно Определение 21, множеството от листата на T се разбива на множествата от листата на T' и T'' , така че броят на листата на T е:

$$\begin{aligned} \left\lceil \frac{p}{2} \right\rceil + \left\lceil \frac{q}{2} \right\rceil &= \left\lceil \frac{2k+1}{2} \right\rceil + \left\lceil \frac{q}{2} \right\rceil = k+1 + \left\lceil \frac{q}{2} \right\rceil = \left\lceil k+1 + \frac{q}{2} \right\rceil = \left\lceil \frac{2k+2+q}{2} \right\rceil = \\ &= \left\lceil \frac{2k+1+q+1}{2} \right\rceil = \left\lceil \frac{p+q+1}{2} \right\rceil = \left\lceil \frac{n}{2} \right\rceil \quad \checkmark \end{aligned}$$

Нека T е конструирано чрез случай 4. Нека T' има p върха и T'' има q върха. Очевидно $n = p + q + 1$. И T' , и T'' са попълнени дървета и индуктивното предположение тях казват, че те имат съответно $\left\lceil \frac{p}{2} \right\rceil$ и $\left\lceil \frac{q}{2} \right\rceil$ листа. Наготово използваме факта, че T'' има нечетен брой върхове, понеже е съвършено двоично дърво. Нека $q = 2k + 1$. Съгласно Определение 21, множеството от листата на T се разбива на множествата от листата на T' и T'' , така че броят на листата на T е:

$$\begin{aligned} \left\lceil \frac{p}{2} \right\rceil + \left\lceil \frac{q}{2} \right\rceil &= \left\lceil \frac{p}{2} \right\rceil + \left\lceil \frac{2k+1}{2} \right\rceil = \left\lceil \frac{p}{2} \right\rceil + k+1 = \left\lceil \frac{p}{2} + k+1 \right\rceil = \left\lceil \frac{p+2k+2}{2} \right\rceil = \\ &= \left\lceil \frac{p+2k+1+1}{2} \right\rceil = \left\lceil \frac{p+q+1}{2} \right\rceil = \left\lceil \frac{n}{2} \right\rceil \quad \checkmark \end{aligned}$$

□

Следствие 10

Вътрешните върхове на произволно попълнено двоично дърво с n върха са $\left\lfloor \frac{n}{2} \right\rfloor$.

Доказателство: Следва веднага от Лема 19 и факта, че $\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n$ за всяко цяло n . □

Лема 20: от страница 71 в [21]

Нека $f(x)$ е произволна реална монотонно растяща функция със свойството:

$f(x)$ е цяло $\rightarrow x$ е цяло

Тогава,

$$\lfloor f(x) \rfloor = \lfloor f(\lfloor x \rfloor) \rfloor \text{ и } \lceil f(x) \rceil = \lceil f(\lceil x \rceil) \rceil$$

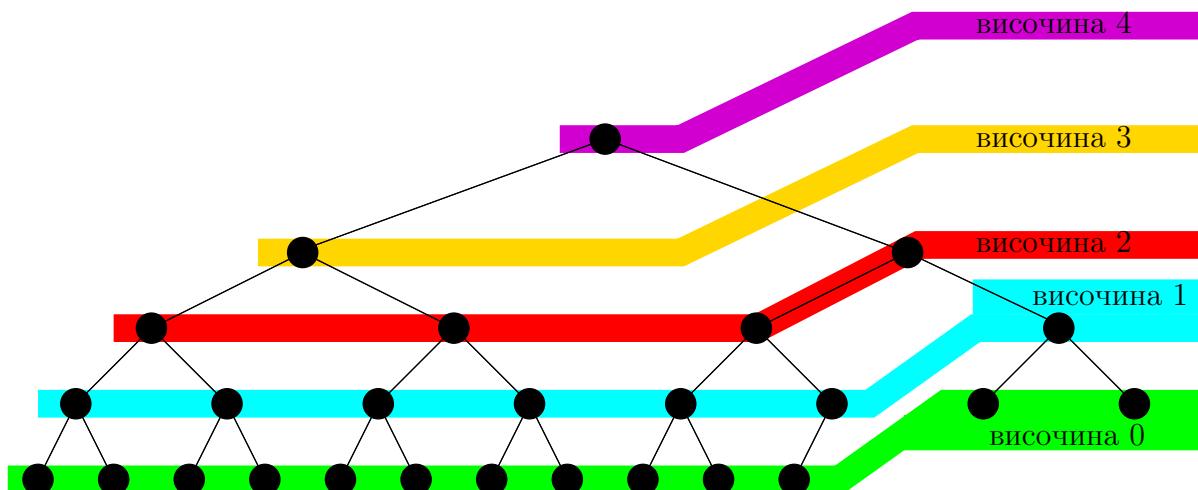
□

От Лема 20 веднага имаме това следствие.

Следствие 11

$$\forall x \in \mathbb{R}^+ \forall b \in \mathbb{N}^+ : \left(\left\lfloor \frac{\lfloor x \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{b} \right\rfloor \text{ и } \left\lceil \frac{\lceil x \rceil}{b} \right\rceil = \left\lceil \frac{x}{b} \right\rceil \right)$$

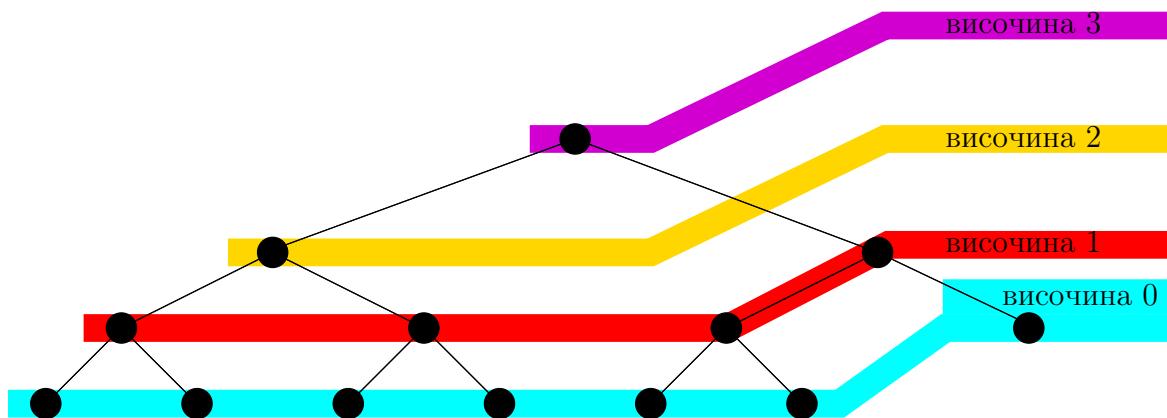
Доказателство: Прилагаме Лема 20 с $f(x) = \frac{x}{b}$. □

Фигура 4.1: Височините на върховете в попълнено двоично дърво T .**Следствие 12**

$$\forall x \in \mathbb{R}^+ \forall a \in \mathbb{N}^+ \forall b \in \mathbb{N}^+ : \left(\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor \text{ и } \left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil \right)$$

Доказателство:Прилагаме Следствие 11 с $\frac{x}{a}$ заместо x . □**Лема 21**Във всяко попълнено двоично дърво T с n върха има точно $\left\lfloor \frac{n}{2^k} \right\rfloor$ върха с височина $\geq k$.**Доказателство:** Да си припомним, че *височина на връх u* е максималното разстояние от него до кое да е листо в поддървото $T[u]$. Пример за височините на върхове има на Фигура 4.1. Доказателството е по индукция по k .**База.** $k = 0$. Върховете с височина ≥ 0 са точно върховете на T , които са n на брой. Забележете, че $n = \left\lfloor \frac{n}{2^0} \right\rfloor$. ✓**Индуктивно предположение.** Нека твърдението е в сила за някаква височина k , която не е максималната.**Индуктивна стъпка.** Да изтрием всички върхове с височина $< k$ от T . Нека полученото дърво е T' и нека n' е броят на неговите върхове. Листата на T' , тоест върховете с височина 0 в T' , очевидно са точно върховете с височина k в T . Всички върхове в T' са точно върховете с височина $\geq k$ в T . Съгласно индуктивното предположение, $n' = \left\lfloor \frac{n}{2^k} \right\rfloor$. Примерно, да разгледаме дървото T от Фигура 4.1 с $k = 1$: след изтриването на всички върхове с височина < 1 получаваме дървото T' от Фигура 4.2.

Съгласно Следствие 10, има точно $\left\lfloor \frac{n'}{2} \right\rfloor$ вътрешни върхове в T' . Но вътрешните върхове на T' са точно върховете с височина $\geq k + 1$ в T . Следователно, има $\left\lfloor \frac{\lfloor n/2^k \rfloor}{2} \right\rfloor$ върха с височина



Фигура 4.2: Попълненото дърво T' , което се получава от дървото T от Фигура 4.1 след изтриване на всички върхове с височина < 1 , тоест с височина 0, тоест листата. Това изтриване намалява с точно единица височините на останалите върхове.

$\geq k + 1$ в T . Съгласно Лема 20, $\left\lfloor \frac{\lfloor \frac{n}{2^k} \rfloor}{2} \right\rfloor = \left\lfloor \frac{n}{2 \times 2^k} \right\rfloor$, и очевидно $\left\lfloor \frac{n}{2 \times 2^k} \right\rfloor = \left\lfloor \frac{n}{2^{k+1}} \right\rfloor$. \square

Следствие 13

Във всяко попълнено двоично дърво T с n върха има точно $\left\lceil \frac{\lfloor \frac{n}{2^k} \rfloor}{2} \right\rceil$ върха с височина k .

Доказателство: Съгласно Лема 21, има точно $\left\lfloor \frac{n}{2^k} \right\rfloor$ върха с височина $\geq k$. Върховете с височина k са точно листата в поддървото на T , индуцирано от върховете с височина $\geq k$. Съгласно Лема 19, тези листа са точно $\left\lceil \frac{\lfloor \frac{n}{2^k} \rfloor}{2} \right\rceil$. \square

4.1.2 Адреси в попълнено двоично дърво

Следното определение задава процедура, която дава адреси-стрингове на върховете на попълнено двоично дърво, използвайки Определение 21. С “ ϵ ” означаваме празния стринг.

Определение 22: Адреси на върхове в попълнено двоично дърво

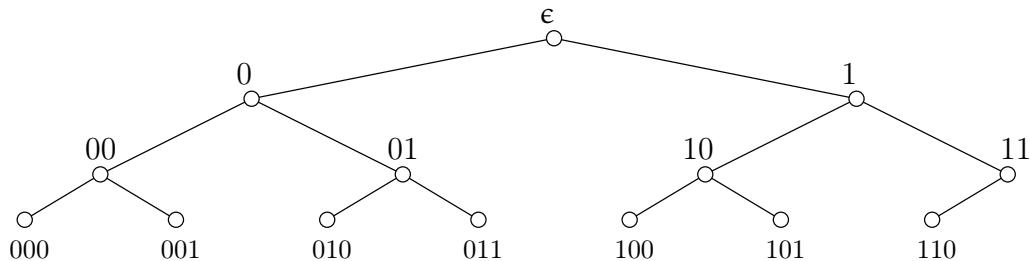
Всеки връх x в попълненото дърво има *адрес*, който бележим с $\text{addr}(x)$. Адресите се конструират от следната процедура от две фази.

фаза I: В базовия случай 1, $\text{addr}(u) = \epsilon$. В базовия случай 2, $\text{addr}(u) = \epsilon$ и $\text{addr}(v) = 0$.
В случаи 3 и 4:

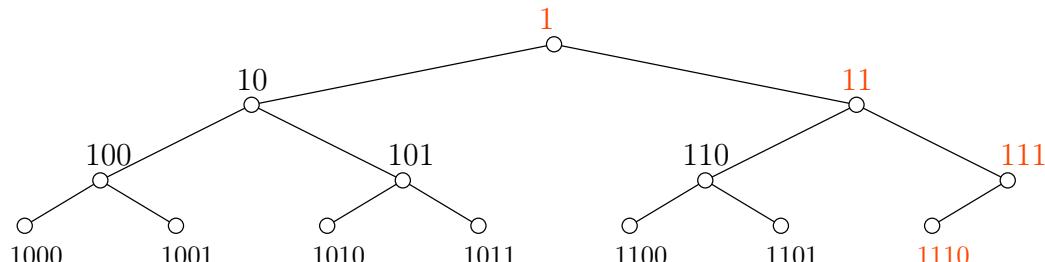
- $\text{addr}(u) = \epsilon$,
- за всеки връх x от T' , новият адрес на x се получава от стария с конкатенация на нула в левия край.
- за всеки връх x от T'' , новият адрес на x се получава от стария с конкатенация на единица в левия край.

фаза II: За всеки връх x от T , новият адрес се получава от стария чрез конкатенация на единица в началото. □

Ето пример за конструиране на адресите от **фаза I** за попълненото дърво с 14 върха:



Продължавайки със същия пример, ето окончателните адреси след **фаза II**:



Според Определение 22 адресите на върховете в попълнените двоични дървета са стрингове. В изложението надолу понякога ще злоупотребяваме с това и ще третираме адресите като числа—а именно, числата, записани с тези стрингове.

Лема 22

Нека T е произволно попълнено двоично дърво и u е произволен връх в него. Ако u не е листо:

- ако u има две деца v и w , където v е лявото дете, в сила е $\text{addr}(v) = 2 \times \text{addr}(u)$ и $\text{addr}(w) = 2 \times \text{addr}(u) + 1$.
- ако u има точно едно дете v , в сила е $\text{addr}(v) = 2 \times \text{addr}(u)$.

Ако u не е корен и v е родителят на u , то $\text{addr}(v) = \left\lfloor \frac{\text{addr}(u)}{2} \right\rfloor$.

Доказателство, част I: Първо ще докажем твърдението за адресите на децата. Ще докажем твърдението по дълбочината на u .

Базовият случай е: u има дълбочина нула, тоест u е коренът. Ако u има две деца v и w , като v е лявото, $\text{addr}(u) = 1$, $\text{addr}(v) = 10$ и $\text{addr}(w) = 11$, така че твърдението е вярно. Ако u има точно едно дете, очевидно става дума за дърво, построено чрез базова конструкция 2 на Определение 21, и съгласно Определение 22, $\text{addr}(u) = 1$ и $\text{addr}(v) = 10$, следователно твърдението пак е вярно.

Нека u има дълбочина d и u има поне едно дете. Ако u има точно две деца v и w , като v е лявото, $\text{addr}(v) = \text{addr}(u)0$ и $\text{addr}(w) = \text{addr}(u)1$, следователно твърдението е вярно, когато разглеждаме адресите като числа, записани в двоична позиционна бройна система.

Доказателство, част II: Ще докажем твърдението за адреса на родителя на всеки връх, който не е коренът. Но това твърдение е очевидно, имайки предвид **част I**. Адресът на u , изразен чрез адресите на децата си, е

$$\text{addr}(u) = \frac{\text{addr}(v)}{2} \quad \text{и} \quad \text{addr}(u) = \frac{\text{addr}(w) - 1}{2}$$

като, ако u има само едно дете, очевидно само първият израз е в сила. Тъй като $\frac{\text{addr}(v)}{2}$ е задължително четно, а $\frac{\text{addr}(w) - 1}{2}$, ако съществува, е задължително нечетно, исканото твърдение следва веднага. \square

За всяко $n \in \mathbb{N}^+$, $\text{bin}(n)$ означава стринга-представяне на n в двоична позиционна бройна система (с водеща единица). Нека $n, k \in \mathbb{N}^+$ и $m = \lfloor \log_2 n \rfloor + 1$. Тогава $\text{bin}_k(n)$ се дефинира така:

$$\text{bin}_k(n) = \begin{cases} 0^{k-m} \text{bin}(n), & \text{ако } k > m \\ \text{bin}(n), & \text{ако } k = m \\ \text{суфиксът с дължина } k \text{ на } \text{bin}(n), & \text{ако } k < m. \end{cases}$$

Забележете, че в случая m е дължината на представянето на n в двоична позиционна бройна система. Също така забележете, че $\text{bin}(n)$ и $\text{bin}_k(n)$ са стрингове над азбуката $\{0, 1\}$, а не числа.

Нека x и y са произволни два върха от едно и също ниво и връх c е общият предшественик с най-малка дълбочина на x и y . Казваме, че x е вляво от y , ако x е връх от $T[a]$ и y е връх от $T[b]$, където a е лявото дете, а b е дясното дете на c .

Лема 23

Нека $T = (V, E)$ е произволно попълнено двоично дърво с n върха. Нека височината на T е h . Нека V_d е подмножеството от върховете на дълбочина d , за $0 \leq d \leq h$. Тогава

- (a) За $0 \leq d \leq h - 1$, множеството от адресите на върховете от V_d е множеството от всички булеви стрингове с дължина $d + 1$ и водеща единица.
- (b) Множеството от адресите на върховете от V_h е множеството от булевите стрингове с дължина $h + 1$ и водеща единица от 10^h до $\text{bin}(n)$ в лексикографската наредба.
- (v) Нещо повече, във всяко ниво върховете са лексикографски сортирани по адреси отляво надясно в T .

Доказателство: Нека $\text{addr}'(x)$ е адресът на x след **фаза I**, за всеки връх $x \in V$. В термините на $\text{addr}'(x)$, твърдения (a) и (b) са еквивалентни съответно на:

- (a') За $0 \leq d \leq h - 1$, множеството от адресите на върховете от V_d е множеството от всички булеви стрингове с дължина d .
- (b') Множеството от адресите на върховете от V_h е множеството от булевите стрингове с дължина h от 0^h до $\text{bin}_h(n)$ в лексикографската наредба.

Това ще докажем чрез структурна индукция съгласно Определение 21 на попълнено дърво, използвайки Определение 22 на адреси на върховете на попълнено дърво.

База. В случай 1 на Определение 21, единственият връх има адрес ϵ . В случай 2,

- в ниво 0 единственият връх наистина има адрес ϵ , който има дължина 0,
- в последното ниво 1 наистина адресите са от 0 до $\text{bin}_1(2) = 0$. ✓

Индуктивна стъпка Да разгледаме случай 3.

- На ниво 0, коренът r наистина получава адрес ϵ .
- Да разгледаме произволно ниво d , където $1 \leq d \leq h - 1$. Тривиално е да се докаже, че върховете от ниво d в T са 2^d на брой и че тяхната наредба отляво надясно се състои от върховете на ниво $d - 1$ в T' , последвани от върховете от ниво $d - 1$ на T'' . Тъй като $0 \leq d - 1 \leq h - 2$, ниво $d - 1$ не е последно ниво нито в T' , нито в T'' . Тогава, съгласно част (a') на индуктивното допускане, адресите на върховете от ниво $d - 1$ и в T' , и в T'' са булевите стрингове с дължина $d - 1$, които и в T' , и в T'' са сортирани отляво надясно. Очевидно след слагане на 0 вляво на всеки адрес от ниво $d - 1$ в T' и на 1 вляво на всеки адрес от ниво $d - 1$ в T'' получаваме—по отношение на T —всички булеви стрингове с дължина d , сортирани лексикографски.
- Да разгледаме последното ниво h в T . В случай 3 то се състои от всички 2^{h-1} върхове от ниво $h - 1$ на T' , последвани вдясно от върховете от ниво $h - 1$ на T'' .

Да разгледаме последното ниво $h - 1$ в T' . Нека T' има p върха и T'' има q върха. Съгласно част (b') на индуктивното предположение, отляво надясно адресите в ниво $h - 1$ на T' са

$$\underbrace{00\dots 0}_{\text{дължина } h-1}, \quad \underbrace{00\dots 01}_{\text{дължина } h-1}, \dots, \underbrace{\text{bin}_{h-1}(p)}_{\text{дължина } h-1}$$

Но $p = 2^{h-1+1} - 1 = 2^h - 1$, защото T' е съвършено дърво, така че $\text{bin}_{h-1}(p) = 1^{h-1}$ единици. Тогава адресите в ниво $h - 1$ на T' са, отляво надясно,

$$\underbrace{00\dots 0}_{\text{дължина } h-1}, \underbrace{00\dots 01}_{\text{дължина } h-1}, \dots, \underbrace{11\dots 1}_{\text{дължина } h-1}$$

тоест всички булеви стрингове с дължина $h - 1$.

Да разгледаме последното ниво $h - 1$ в T'' . Съгласно част (a') на индуктивното предположение, отляво надясно адресите в ниво $h - 1$ на T'' са

$$\underbrace{00\dots 0}_{\text{дължина } h-1}, \underbrace{00\dots 01}_{\text{дължина } h-1}, \dots, \underbrace{\text{bin}_{h-1}(q)}_{\text{дължина } h-1}$$

Следователно, в цялото дърво T , адресите на върховете от ниво h са, отляво надясно

$$\underbrace{00\dots 0}_{\text{дължина } h}, \underbrace{00\dots 01}_{\text{дължина } h}, \dots, \underbrace{011\dots 1}_{\text{дължина } h}, \underbrace{10\dots 0}_{\text{дължина } h}, \underbrace{10\dots 01}_{\text{дължина } h}, \dots, \underbrace{1\text{bin}_{h-1}(q)}_{\text{дължина } h}$$

Тъй като $n = p + q + 1$ и $p = 2^h - 1$, то $n = 2^h + q$. Но тъй като $q \leq p$, дължината на $\text{bin}(q)$ също е по-малка от h и $\text{bin}_{h-1}(q)$ се получава от $\text{bin}(q)$ с добавяне на ≥ 0 на брой нули вляво. Имайки предвид, че $\text{bin}(2^h) = 10^h$, заключаваме, че стрингът $1\text{bin}_{1-h}(q)$ е равен на $\text{bin}(n)$. Заключаваме, че адресите от последното ниво на T' отляво надясно, следвани от адресите от последното ниво на T'' отляво надясно, представляват последователността

$$\underbrace{00\dots 0}_{\text{дължина } h}, \underbrace{00\dots 01}_{\text{дължина } h}, \dots, \underbrace{011\dots 1}_{\text{дължина } h}, \underbrace{10\dots 0}_{\text{дължина } h}, \underbrace{10\dots 01}_{\text{дължина } h}, \dots, \underbrace{\text{bin}(n)}_{\text{дължина } h}$$

Да разгледаме случай 4.

- На ниво 0, коренът r наистина получава адрес ϵ .
- Да разгледаме произволно ниво d в T , където $1 \leq d \leq h - 1$. Аналогично на предния случай, върховете от ниво d в T са 2^d на брой и тяхната наредба отляво надясно се състои от върховете на ниво $d - 1$ в T' , последвани от върховете от ниво $d - 1$ на T'' . Тъй като $0 \leq d - 1 \leq h - 2$, ниво $d - 1$ не е последното ниво в T' , но е последното ниво в съвършеното дърво T'' при $d - 1 = h - 2$.

Адресите на върховете от ниво $d - 1$ в T' са булевите стрингове с дължина $d - 1$ и са сортирани отляво надясно съгласно част (a') на индуктивното предположение. Ако $d - 1 < h - 2$, адресите на върховете от ниво $d - 1$ в T'' са булевите стрингове с дължина $d - 1$ и са сортирани отляво надясно съгласно част (a') на индуктивното предположение.

Да разгледаме случая $d - 1 = h - 2$ по отношение на T'' . Сега част (б') на индуктивното предположение е приложима. Съгласно нея, адресите на върховете от ниво $d - 1$ в T'' са булевите стрингове от 0^{h-2} до $\text{bin}_{h-2}(q)$ в лексикографската наредба, където q е броят на върховете в T'' , и те са сортирани отляво надясно лексикографски. Но тъй като T'' е съвършено дърво с височина $h - 2$, вярно е, че $q = 2^{h-2+1} - 1 = 2^{h-1} - 1$, следователно $\text{bin}_{h-2}(q) = 1^{d-1}$.

Заключаваме, че за $1 \leq d \leq h - 1$, адресите на върховете от ниво $d - 1$ в T' са всички булеви стрингове с дължина $d - 1$, сортирани отляво надясно, а също така адресите на

върховете от ниво $d - 1$ в T'' са всички булеви стрингове с дължина $d - 1$ сортирани отляво надясно. Очевидно след слагане на 0 вляво на всеки адрес от ниво $d - 1$ в T' и на 1 вляво на всеки адрес от ниво $d - 1$ в T'' получаваме—по отношение на дървото T —всички булеви стрингове с дължина d , сортирани лексикографски отляво надясно.

- Да разгледаме последното ниво h в T . То е последното ниво $h - 1$ на T' , така че част (б') на индуктивното предположение е приложима. Съгласно нея, адресите на върховете от ниво $h - 1$ в T' са булевите стрингове от 0^{h-1} до $\text{bin}_{h-1}(p)$, където p е броят на върховете в T' , и те са сортирани отляво надясно лексикографски. Тогава във **фаза I** тези върхове получават адреси от 0^h до $0\text{bin}_{h-1}(p)$, сортирани отляво надясно лексикографски.

Да си припомним, че $n = p + q + 1$, където q е броят на върховете в съвършеното дърво T'' . Тъй като T'' е съвършено и с височина $h - 2$, вярно е, че $q = 2^{h-1} - 1$, следователно $q + 1 = 2^{h-1}$, следователно $\text{bin}(q + 1) = 10^{h-1}$. От друга страна, $p \geq q + 1$, така че $\text{bin}(p) = 1\sigma$ за някой булев стринг σ с дължина $h - 1$. Тогава $\text{bin}(p + q + 1) = 10\sigma$. С други думи, $\text{bin}(n) = 10\sigma$. Но тогава $\text{bin}_h(n) = 0\sigma = 0\text{bin}_{h-1}(p)$.

Доказвахме, че наистина адресите на върховете от последното ниво на T са лексикографски сортирани стрингове от 0^h до $\text{bin}_h(n)$. \square

4.1.3 Двоична пирамида

Сега допускаме, че всеки връх има асоциирана стойност, наречена *ключ* (key). Въз основа на “попълнено двоично дърво” въвеждаме “двоична пирамида” (binary heap, вж. [49]).

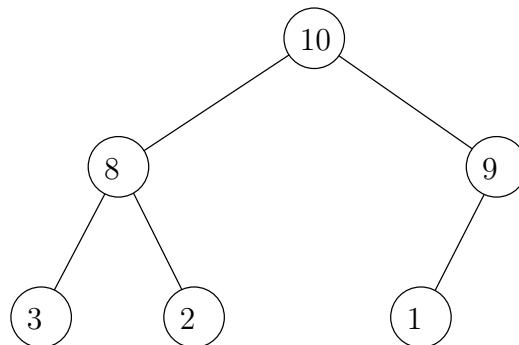
Определение 23: Двоична пирамида

Двоична пирамида е попълнено двоично дърво, в което:

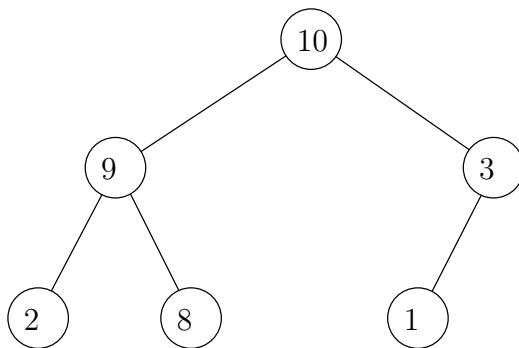
- или ключът на всеки връх по-голям или равен на ключовете на неговите деца и тогава пирамидата е *максимална пирамида* (max heap),
- или ключът на всеки връх е по-малък или равен на ключовете на неговите деца и тогава пирамидата е *минимална пирамида* (min heap).

Тук ще разглеждаме максимални пирамиди и казвайки “пирамида” ще имаме предвид “максимална пирамида”. Но изложението може да бъде променено лесно, така че да се отнася за минимални пирамиди. Също така, ще изпускаме “двоични”, понеже не разглеждаме пирамиди, които не са двоични.

Ето пример за пирамида:



Очевидно формата на дървото е фиксирана от броя на върховете, но подредбата на ключовете не е фиксирана[†]. Най-големият ключ 10 непременно е в корена и 9 непременно е в дете на корена, но известно вариране е възможно. Примерно, това също е пирамида със тези ключове:



Наблюдение 10

Попълнено двоично дърво с ключове е пирамида тогава и само тогава, когато за всяко листо u , по (уникалния) път p , свързващ корена с u , стойностите на ключовете са ненарастващи по p в посока от корена към u .

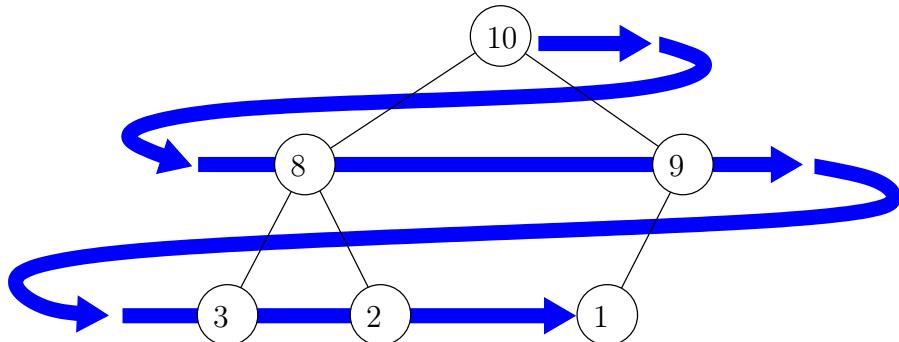
4.1.4 Реализиране на пирамида чрез масив

Определение 24

Нека $T = (V, E)$ е попълнено двоично дърво с n върха. *Линеаризация* на T се нарича масивът $A[1, 2, \dots, n]$ такъв че за всяко i , $1 \leq i \leq n$, $A[i]$ е ключът на върха, чийто адрес е i .

За да съкратим изложението, често игнорираме разликата между “пирамида” и “линеаризация на пирамида” и, казвайки просто “пирамида”, имаме предвид “линеаризация на пирамида”. С други думи, самият масив-линеаризация бива наричан “пирамида”.

Съгласно Лема 23, съществува биекция между множеството от адресите и множеството $\{1, 2, \dots, n\}$, така че определението е смислено. И така, пирамидите може да се реализират много ефикасно чрез масиви. Линеаризацията на последния пример е $[10, 8, 9, 3, 2, 1]$. Линеаризирането можем да си представяме като обхождане на дървото по нива, във всяко ниво, отляво надясно, и последователно записване на ключовете в масива:



[†] В Допълнение 17 дискутираме броя на различните пирамиди с n върха

Лема 23 гарантира, че в нивата няма “празнини”, така че в масива няма “дупки”.

При тази реализация се избягва разходът на памет, който е неизбежен при дърветата по принцип. Ако искаме да представим двоично дърво в общия случай—не непременно попълнено—трябва по никакъв начин да укажем за всеки връх кой е родителят, или кои са децата. Това ни струва допълнителна памет. Докато пирамида може да се реализира без никаква допълнителна памет, като при това обхождането е много ефикасно като сложност по време. Последното твърдение се основава на факта (виж Наблюдение 11), че ако масивът $A[1, \dots, n]$ е линеаризация на пирамида, то за всеки елемент $A[i]$ можем в $\Theta(1)$ време да определим дали $A[i]$ е корен или не, ако не е, то кой е родителят му, дали е вътрешен връх или листо, и ако е вътрешен връх, кое е лявото му дете $A[j]$, ако такова съществува, и дясното му дете $A[k]$, ако такова съществува.

Следното наблюдение е просто приложение на Лема 22.

Наблюдение 11

Нека T е пирамида с n елемента и масивът $A[1, \dots, n]$ е нейната линеаризация. Тогава за всеки елемент $A[i]$:

1. елементът, който отговаря на родителя на $A[i]$, е $A[\lfloor \frac{i}{2} \rfloor]$, ако върхът на пирамидата, който отговаря на $A[i]$, не е коренът.
2. елементът, който отговаря на лявото дете на $A[i]$, е $A[2 \times i]$, ако върхът на пирамидата, отговарящ на $A[i]$, има ляво дете,
3. елементът, който отговаря на дясното дете на $A[i]$, е $A[2 \times i + 1]$, ако върхът на пирамидата, отговарящ на $A[i]$, има дясно дете. □

Наблюдение 11 ни дава основание да дефинираме следните изчислителни примитиви[†], чито

[†]“Примитив” е, неформално казано, лесна за възприемане функция с малка сложност, която се използва често от алгоритмите, които ни интересуват.

имена са достатъчно информативни:

PARENT(i)

```
return (if  $i \geq 2$  then  $\lfloor \frac{i}{2} \rfloor$  else  $i$ )
```

LEFT(i)

```
return (if  $2i \leq n$  then  $2i$  else UNDEFINED)
```

RIGHT(i)

```
return (if  $2i + 1 \leq n$  then  $2i + 1$  else UNDEFINED)
```

LEVEL(i)

```
return  $\lceil \log_2 i \rceil$ 
```

ISLEAF(i)

```
return (if  $i > \lfloor \frac{n}{2} \rfloor$  then YES else NO)
```

ISINTERNALVERTEX(i)

```
return not ISLEAF( $i$ )
```

ISROOT(i)

```
return (if  $i = 1$  then YES else NO)
```

Освен това, за $t \geq 1$:

$$\text{PARENT}^t(i) = \begin{cases} \text{PARENT}(i), & \text{ако } t = 1 \\ \text{PARENT}(\text{PARENT}^{t-1}(i)), & \text{в противен случай} \end{cases}$$

Казваме, че $A[j]$ е *предшественик* на $A[i]$, ако $j = \text{PARENT}^t(i)$ за някое $t \geq 1$.

Определение 25: пирамидална инверсия в масив

Нека $A[1, \dots, n]$ е масив от ключове. *Пирамидална инверсия* в A наричаме всяка наредена двойка индекси $\langle i, j \rangle$, такава че $A[i]$ е предшественик на $A[j]$ и $A[i] < A[j]$. Наредената двойка $\langle A[i], A[j] \rangle$ също наричаме инверсия. *Директна пирамидална инверсия*, или накратко *директна инверсия*, ако от контекста е ясно, че става дума за пирамидални инверсии, е всяка пирамидална инверсия $\langle i, j \rangle$, такава че $i = \text{PARENT}(j)$. Пирамидални инверсии, които не са директни, се казват *индиректни*.

Например, на Фигура 4.8 е показан обект (като попълнено дърво, но читателят лесно може да си представи съответния масив), би бил пирамида, ако не беше последният елемент (найдясното листо). Има три пирамидални инверсии, които са между последния елемент и негови предшественици. Има точно една директна инверсия: между елементът с ключ 6 (индекс 13) и елементът с ключ 35 (индекс 27).

Наблюдение 12

Масивът от ключове A е пирамида тогава и само тогава, когато няма нито една пирамidalна инверсия. □

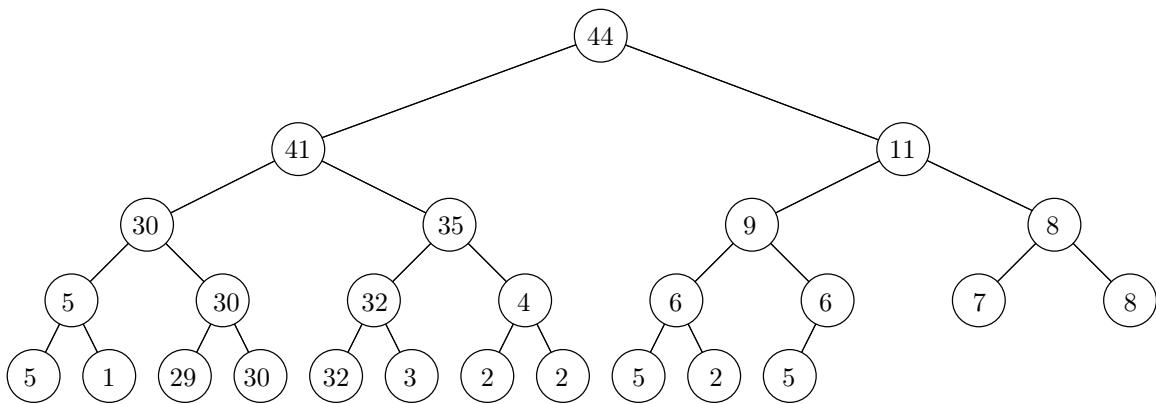
4.1.5 Подпирамида

Нека T е пирамида с n върха, реализирана с масив $A[1, 2, \dots, n]$. За всеки връх $u \in V(T)$, подпирамидата с корен u е $T[u]$. По отношение на масива A , който преставя пирамидата, u е индекс на елемент. Нотацията $A[u]$ означава реализацията на $T[u]$ в масива A . Очевидно, в общия случай $A[u]$ не е непрекъснат подмасив, а се състои от няколко подмасива—на брой колкото е височината на съответното поддърво—всеки от които е непрекъсната подпоследователност на A (вижте Фигура 4.6).

Наблюдение 13

Ако $A[1, \dots, n]$ е пирамида, то за всяко $k \in \{1, \dots, n\}$, $A[k]$ е пирамида. □

Като пример за пирамида и нейната линеаризация, да разгледаме пирамидата, показана на Фигура 4.3. Фигура 4.4 показва нейната линеаризация. Фигура 4.5 показва листата на пирамидата. На Фигура 4.6 е показана подпирамидата $A[3]$. Връх 3 е третият връх в масива (дясното дете на корена, ако разсъждаваме в термините на дървото). Фигура 4.7 показва листата на $A[3]$ в червено.



Фигура 4.3: Пирамида с 26 върха.

A	44 41 11 30 35 9 8 5 30 32 4 6 6 7 8 5 1 29 30 32 3 2 2 5 2 5
	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Фигура 4.4: Линеаризацията на пирамида от Фигура 4.3.

A	44 41 11 30 35 9 8 5 30 32 4 6 6 7 8 5 1 29 30 32 3 2 2 5 2 5
	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Фигура 4.5: Листата на пирамидата от Фигура 4.3 са отбелязани с жълто.

A	44 41 11 30 35 9 8 5 30 32 4 6 6 7 8 5 1 29 30 32 3 2 2 5 2 5
	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Фигура 4.6: Подпирамидата $A[3]$.

A	44 41 11 30 35 9 8 5 30 32 4 6 6 7 8 5 1 29 30 32 3 2 2 5 2 5
	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Фигура 4.7: Листата на $A[3]$ в червено.

Използвайки дефинираните примитиви, следният алгоритъм обхожда $T[u]$. “Обхожда” означава “извежда ключовете на $T[u]$ в preorder[†]” (а не по нива).

TRAVERSE BINARY SUBHEAP($A[1, \dots, n]$: пирамида, i : число от $\{1, \dots, n\}$)

```

1 print i
2 left ← LEFT(i)
3 right ← RIGHT(i)
4 if left ≤ n
5   TRAVERSE BINARY SUBHEAP( $A$ ,  $left$ )
6 if right ≤ n
7   TRAVERSE BINARY SUBHEAP( $A$ ,  $right$ )

```

4.2 Построяване на пирамида

Даден е масив $A[1, \dots, n]$ от ключове. За простота допускаме, че ключовете се естествени числа. Числата са в произволен порядък. Искаме да разместим числата така, че масивът да стане пирамида. Дори числата да са две по две различни, има много начини да направим пирамида от дадени числа. За подробна дискусия на броя на пирамидите вижте Допълнение 17.

4.2.1 Наивно построяване на пирамида

Най-естественият начин да бъде превърнат произволен масив от числа в пирамида чрез размещване е чрез последователно добавяне на елементи в частично построена пирамида.

NAIVE BUILD HEAP, OUTLINE($A[1, \dots, n]$: масив от цели числа)

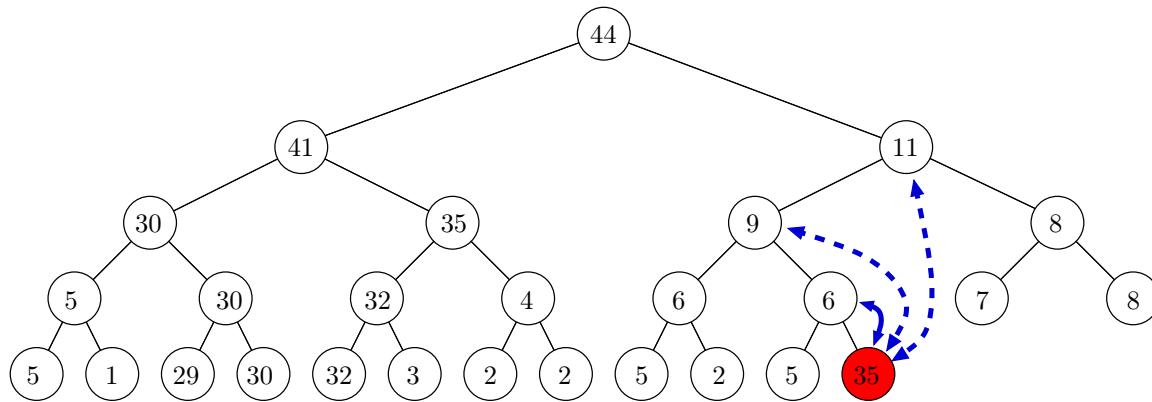
```

1 for i ← 2 to n
2   добави  $A[i]$  към досега построената пирамида  $A[1, \dots, i-1]$ ,
   така че  $A[1, \dots, i]$  да стане пирамида

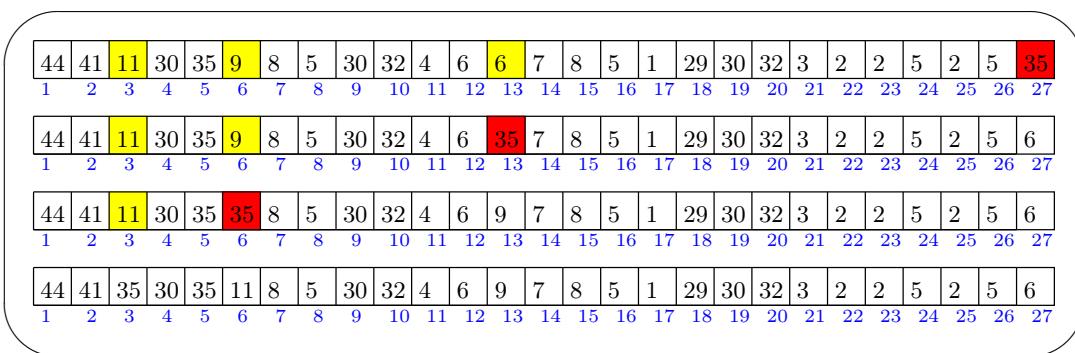
```

Това описание е прекалено общо, но идеята е ясна: “добави към $A[1, \dots, i-1]$ ” не е просто увеличаване на индекса i . Дори $A[1, \dots, i-1]$ да е пирамида, $A[1, \dots, i-1, i]$ не е непременно пирамида, защото може да има пирамидални инверсии $\langle PARENT^t(i), i \rangle$. Това е илюстрирано на Фигура 4.8, на която структурата е показана като дърво, а не като масив. Дървото съдържа пирамидата от Фигура 4.3 плюс още един връх, оцветен в червено. Ключът на този нов връх е 35 и той участва в пирамидални инверсии с три от своите предшественици. Съществуват много начини да бъдат разместени елементите на $A[1, \dots, i]$, така че да няма пирамидални инверсии. Най-естественият и прост начин е, да ликвидираме пирамидалните инверсии, в които участва новодобавеният $A[i]$, без да разместваме никакви елементи, които не участват в тези инверсии. Ако мислим за обекта като дърво: очевидно въпросните инверсии са между новодобавения $A[i]$ и върхове по пътя от него към корена, които индуцират подпът (тоест, по пътя от $A[i]$ към корена, върховете, участващи в инверсии, са непрекъсната последователност.) Очевидното решение тогава е да разменяме $A[i]$ с $A[PARENT(i)]$, докато има инверсии. Всяка такава размяна ликвидира една инверсия и не извежда нови инверсии, защото ключът, който застава “отгоре”, е по-голям от този, който преди е бил там преди размяната, следователно този елемент (който застава отгоре) остава не по-малък от другия си наследник (ако има такъв).

[†]За обхождания на дървета, включително в preorder, вижте примерно [61].



Фигура 4.8: Пирамидата от Фигура 4.3 с един добавен връх (в червено). Полученият обект с 27 елемента не е пирамида, понеже има три пирамидални инверсии между новодобавения елемент и негови предшественици. Пирамидалните инверсии са показани със сини стрелки, като с непрекъсната линия е показана единствената директна пирамидална инверсия, а с прекъснати линии – двете индиректни пирамидални инверсии.



Фигура 4.9: Линеаризацията на пирамидата от Фигура 4.3 е най-горе. Елементът, който “не си е на мястото”, е накрая в червено. Елементите, които образуват пирамидални инверсии с него, са в жълто. Следващите три състояния на масива отгоре надолу илюстрират работата на NAIVE BUILD HEAP.

NAIVE BUILD HEAP, DETAILED($A[1, \dots, n]$): масив от ест. числа)

```

1 for i ← 2 to n
2   k ← i
3   while k > 1 and A[PARENT(k)] < A[k] do
4     swap(A[k], A[PARENT(k)])
5     k ← PARENT(k)

```

Ще илюстрираме работата на наивното построяване, използвайки за пример пирамидата от Фигура 4.3. Работата на алгоритъма е показана на Фигура 4.9.

Доказателството за коректност е напълно тривиално и остава за читателя. Ще изследваме сложността по време на наивното построяване на пирамида. Външният цикъл се изпълнява $\Theta(n)$ пъти без оглед на конкретния вход. Вътрешният цикъл се изпълнява, в най-лошия случай, $\lfloor \lg i \rfloor$ пъти за всяко i , понеже височината на подпирамидата с корен $A[i]$ е $\lfloor \lg i \rfloor$. Тогава сложността се определя от сумата

$$\sum_{i=2}^n \lfloor \lg i \rfloor = \Theta \left(\sum_{i=2}^n \lg i \right) = \Theta (\lg (2 \times 3 \times \dots \times (n-1) \times n)) = \Theta (\lg n!) \quad (4.1)$$

От Теорема 14) знаем, че $\lg n! \asymp n \lg n$. Следователно, сложността на наивното построяване е $\Theta(n \lg n)$.

4.2.2 Бързо построяване на пирамида: алгоритъм BUILD HEAP

Алгоритъмът-предмет на тази подсекция е предложен от Robert Floyd [17]. Самият алгоритъм е показан на стр. 130. Той използва функция HEAPIFY, която показваме в два варианта: итеративен и рекурсивен.

Предварителни обяснения

Да допуснем, че е дадено попълнено двоично дърво с ключове T . Нека коренът на T е u и децата на u са v и w . За целта на това обяснение, нека ключовете са съответно $u.key$, $v.key$ и $w.key$. Нека $T[v]$ и $T[w]$ са пирамиди. Ако $u.key \geq v.key$ и $u.key \geq w.key$, то T е пирамида. Да допуснем, че $u.key < v.key$ или $u.key < w.key$. Без ограничение на общността, нека $u.key < v.key$ и $u.key < w.key$. Тогава в T може да има много пирамидални инверсии. Пирамидалните инверсии може да са $n - 1$ на брой, където n е броят на върховете на T , следователно може всеки елемент от $T[v]$ и $T[w]$ да участва в инверсия с корена u .

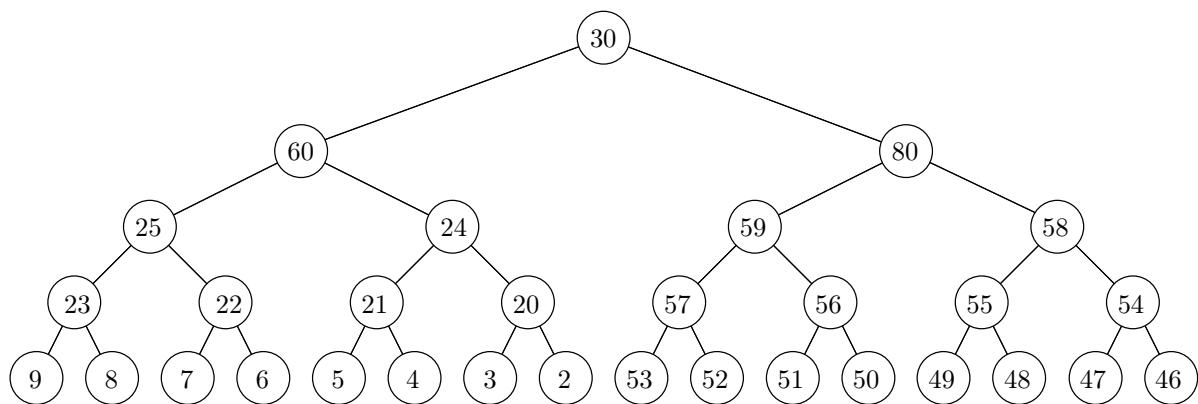
Ключовото наблюдение е, че с $O(\lg n)$ размени на елементи може да премахнем всички пирамидални инверсии, тоест да направим T пирамида. Интуитивно е ясно, че трябва да разменим u с този връх измежду v и w , чийто ключ е максимален[†]. Без ограничение на общността, нека $v.key < w.key$. Тогава ще разменим u с w . Със сигурност след размяната w няма да участва в инверсия с нито един връх от $T[v]$, така че след размяната ще инверси—ако изобщо има—само в поддървото, чийто корен сега е u . Грубо казано, броят на инверсиите ще намалее наполовина или повече.[‡]

Забележете, че има примери, в които е по-изгодно да разменим корена не с детето с по-големия ключ, а с детето с по-малкия ключ. Такова дърво е показано на Фигура 4.10. Първо да видим какво ще стане, ако върху това дърво прилагаме размяна на връх с това от двете деца, което има по-голям ключ. Фигура 4.11 показва дървото след първата размяна – елементите 30 и 80 са разменени, а с червено е очертан пътят, който ще “измине” върхът с ключ 30, движейки се надолу, докато не стане листо. На Фигура 4.12 е показано дървото след последната размяна, като е очертан пътят, който е “изминал” върхът с ключ 30. Това дърво е пирамида, получена след четири размени. За дървото от Фигура 4.10, ако разменим корена с детето с по-малък ключ, повече размени в лявото поддърво няма да има; ако разменим след това новия корен с ключ 60 с дясното дете с ключ 80, ще получим пирамидата, показана на Фигура 4.13, със само две размени.

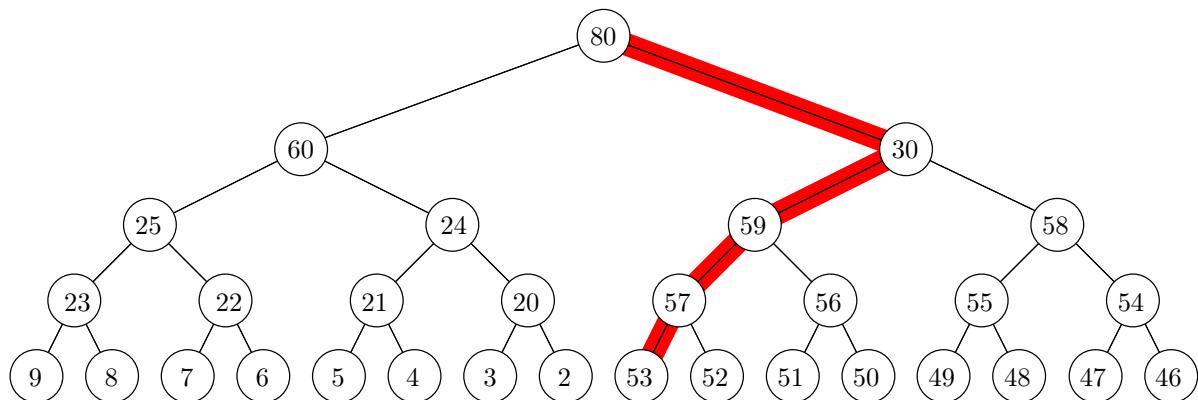
Примерът, който видяхме току-що, се машабира за всяка размер на дървото, така че наистина има случаи, в които размяната с детето с по-малък ключ води до $\Theta(1)$ размени, а размяната с детето с по-голям ключ води до $\Theta(\lg n)$ размени. Но ние се интересуваме от сложността в най-лошия случай. В най-лошия случай, размяната с детето с по-малък ключ може да доведе до $\Omega(n)$ размени, преди да получим пирамида, докато размяната с детето с по-голям ключ води до $O(\lg n)$ размени винаги. Пример за дърво, върху което размяната с детето с по-малък ключ би довела до $\Omega(n)$ размени (примерът се машабира за безброй много n) е показан на Фигура 4.14. На Фигура 4.15 е показана пирамидата, която се получава от дървото на Фигура 4.14 след $\Omega(n)$ размени.

[†]Ако v и w имат еднакви ключове, ще разменим u с кой да е от тях.

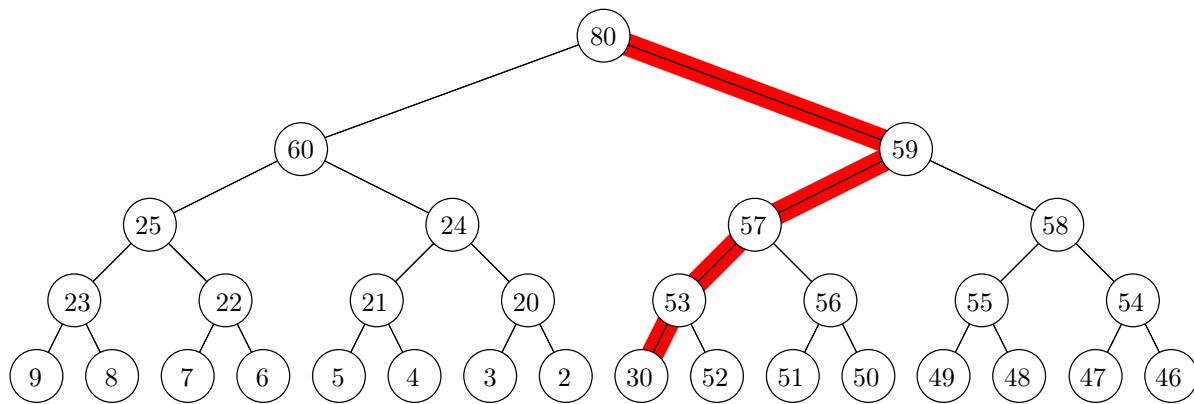
[‡]Лесно е да се съобрази, че максималният брой инверсии след размяната, изразен чрез общия брой на върховете, е $\frac{2(n-2)}{3}$, където $n - 2$ се дели на 3. Дори броят на инверсиите да намалява след всяка размяна с деление на $\frac{3}{2}$, броят на размените е логаритмичен.



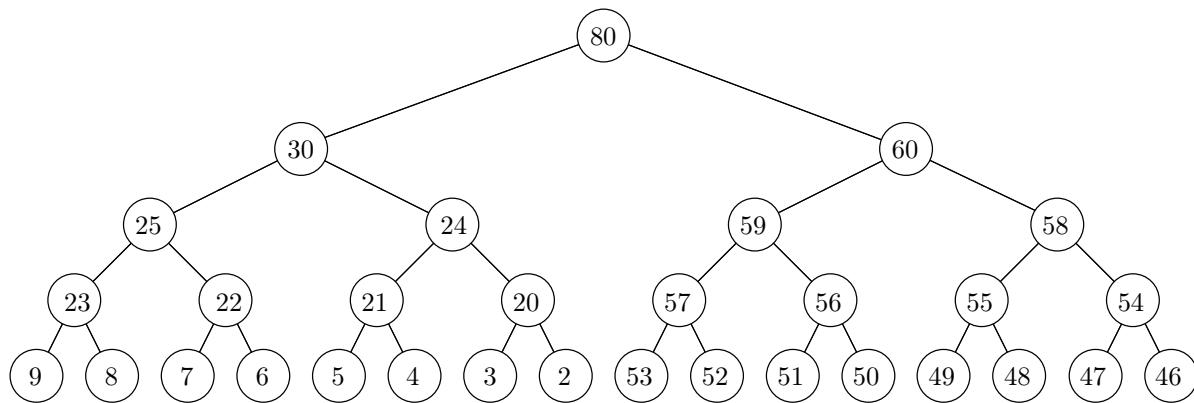
Фигура 4.10: Попълнено дърво с ключове, в което двете деца на корена са корени на поддървата-пирамиди, а ключът на корена е по-малък от ключовете на двете му деца. По-изгодно като минимален брой размени е коренът да бъде разменен с детето с по-малкия ключ.



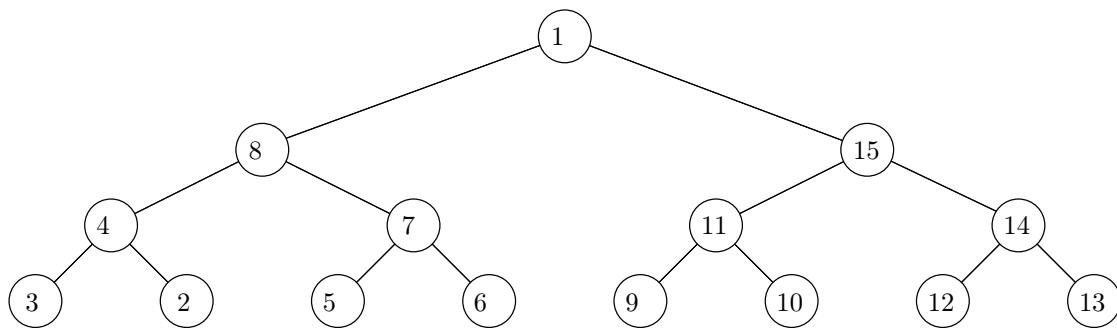
Фигура 4.11: По отношение на дървото от Фигура 4.10, ако разменим корена с детето с по-голям ключ и продължим размените по същия начин—с детето с по-голям ключ—ще направим 4 размени, докато върхът с ключ 30 не стане листо. Тук е показано дървото след първата размяна и е очертан пътят, който “изминава” върхът с ключ 30.



Фигура 4.12: По отношение на дървото от Фигура 4.11 е показано дървото след всички размени, където размените са от вида “разменяме корен с това дете, което има по-голям ключ”. Извършени са 4 размени и дървото е пирамида. Очертан е пътят, който е “изминал” върхът с ключ 30.

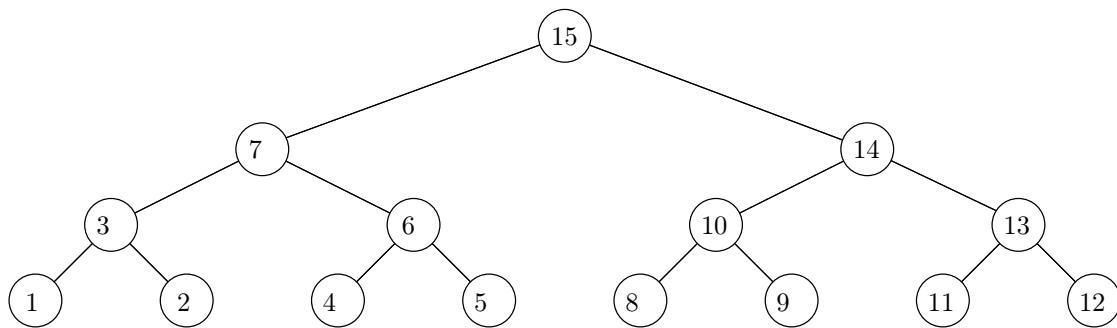


Фигура 4.13: По отношение на дървото от Фигура 4.10, ако разменяме връх с корена с детето с по-малък ключ, ще направим само 2 размени и ще получим показаната пирамида.



Фигура 4.14: Пример за дърво, в което, ако разменяме корена с детето с по-малък ключ (и после правим още една размяна на новия корен с детето с по-голям ключ), броят на размените е по-голям от броя на върховете.

От изложението трябва да е станало ясно, че предложената идея би имала сложност $O(\lg n)$, ако се реализира грамотно: с $O(\lg n)$ размени на елементи унищожаваме всички пирамидални инверсии, а всяка размяна става в константно време.



Фигура 4.15: Ако в дървото, показано на Фигура 4.14, систематично разменяме първо корен с детето с по-малък ключ, и после новия корен с детето с по-голям ключ, ще получим пирамидата, показана тук. Броят на размените е по-голям от броя на върховете.

Неформално въвеждане на HEAPIFY

В изложението досега “пирамидизирахме” цялото дърво – при допускането, че само коренът има какъв да е ключ, а лявото и дясното поддърво са пирамиди. Да разгледаме по-общата възможност: разглеждаме произволен вътрешен връх u в дървото T , такъв че и двете поддървета (или само едно, в екстремния случай, че дясно дете няма), вкоренени в неговите деца, са пирамиди, а ключът на u е произволен, и целта е да “пирамидизираме” $T[u]$.

И така, алгоритъмът HEAPIFY има два входа. Единият е масив A , другият е индекс i в A . Ако $i > \lfloor \frac{n}{2} \rfloor$, то $A[i]$ отговаря на листо и няма какво да правим – поддървото $A[i]$ е тривиална пирамида.

Нека $i \leq \lfloor \frac{n}{2} \rfloor$; с други думи, нека $A[i]$ отговаря на вътрешен връх. Алгоритъмът размества елементите на $A[i]$ така, че $A[i]$ става пирамида, без да променя нищо извън $A[i]$. Това се постига с последователни размени на елемента, който в началото е на позиция i , с това от децата му, което има по-голям ключ, докато или този елемент не стане листо, или и двете му деца (или единственото му дете, ако няма дясно дете) не се окажат по-малки или равни на него.

HEAPIFY в итеративен вариант

ITERATIVE HEAPIFY($A[1, 2, \dots, n]$): масив от цели числа, i : индекс в A)

```

1   j ← i
2   while  $j \leq \lfloor \frac{n}{2} \rfloor$  do
3       left ← LEFT(j)
4       right ← RIGHT(j)
5       if  $A[left] > A[j]$ 
6           (* Няма смисъл да проверяваме дали  $left \leq n$ , понеже  $j \leq \lfloor \frac{n}{2} \rfloor$  *)
7           largest ← left
8       else
9           largest ← j
10      if  $right \leq n$  and  $A[right] > A[largest]$ 
11          largest ← right
12      if  $largest \neq j$ 
13          swap( $A[j], A[largest]$ )
14          j ← largest

```

```

15      else
16          break

```

Лема 24

При допускането, че $A[1, \dots, n]$ и i са такива, че всеки от $A[\text{LEFT}(i)]$ и $A[\text{RIGHT}(i)]$, ако съществува, е пирамида, ефектът от ITERATIVE HEAPIFY е, че $A[i]$ е пирамида при неговото приключване.

Доказателство: Следното твърдение е инвариант за while-цикъла (редове 1–16).

Инвариант 6: Цикълът на ITERATIVE HEAPIFY

Всеки път, когато изпълнението е на ред 2, единствените възможни директни инверсии в $A[i]$ са $\langle j, \text{LEFT}(j) \rangle$ и $\langle j, \text{RIGHT}(j) \rangle^a$, ако $A[\text{LEFT}(j)]$ и $A[\text{RIGHT}(j)]$ съществуват.

^aПо друг начин казано, ако съществуват пирамидални инверсии в $A[i]$, то те са в $A[j]$.

База. $j = i$. Съгласно допусканията, всеки от $A[\text{LEFT}(i)]$ и $A[\text{RIGHT}(i)]$, ако съществува, е пирамида, така че твърдението е в сила. ✓

Поддръжка. Допускаме, че твърдението е в сила в някой момент, в който изпълнението е на ред 2 и изпълнението ще бъде на ред 2 поне още веднъж.[§] И така, $j \leq \lfloor \frac{n}{2} \rfloor$ и поне едно от $\text{LEFT}(j) \leq n$ и $\text{RIGHT}(j) \leq n$ е изпълнено, а именно $\text{LEFT}(j) \leq n$. Тогава поне $A[\text{LEFT}(j)]$ е дефинирано. Без ограничение на общността ще допуснем, че и $\text{LEFT}(j) \leq n$, и $\text{RIGHT}(j) \leq n$, за да избегнем ненужни подслучаи.

Случай I: $A[\text{left}] > A[j] > A[\text{right}]$ в началото на изпълнението на цикъла, и освен това $A[\text{right}] > A[\text{left}]$. Условието на ред 5 е ИСТИНА и присвояването на ред 7 се случва. Условието на ред 10 също е ИСТИНА, така че и присвояването на ред 11 се случва. Когато изпълнението е на ред 12, largest е равно на right . Условието на ред 12 е ИСТИНА и на ред 13, $A[j]$ и $A[\text{largest}]$ биват разменени. Твърдим, че единствените възможни пирамидални инверсии в $A[i]$ след размяната са $\langle \text{right}, \text{LEFT}(\text{right}) \rangle$ и $\langle \text{right}, \text{RIGHT}(\text{right}) \rangle$:

- нито един елемент от $A[i]$, който е извън $A[j]$, не е променян;
- $A[j] > A[\text{left}]$, така че $\langle j, \text{LEFT}(j) \rangle$ не може да е пирамидална инверсия;
- $A[j] > A[\text{right}]$, така че $\langle j, \text{RIGHT}(j) \rangle$ не може да е пирамидална инверсия;
- $A[\text{left}]$ не е бил модифициран;
- нито един от $A[\text{LEFT}(\text{right})]$ и $A[\text{RIGHT}(\text{right})]$ не е бил модифициран.

Но на ред 14, j става равен на right . Следователно, инвариантата е в сила при следващото достигане на ред 2.

Случай II: $A[\text{left}] > A[j] > A[\text{right}]$ в началото на изпълнението на цикъла, но $A[\text{right}] \not> A[\text{left}]$. Условието на ред 5 е ИСТИНА и присвояването на ред 7 се случва. Условието на ред 10 е ЛЪЖА, следователно присвояването на ред 11 не се случва и когато

[§]Последното имплицира, че ред 16 няма да бъде изпълнен.

изпълнението е на ред 12, largest е равно на left . Условието на ред 12 е ИСТИНА и на ред 13, $A[j]$ и $A[\text{largest}]$ биват разменени. Твърдим, че единствените възможни пирамидални инверсии в $A[i]$ след размяната са $\langle \text{left}, \text{LEFT}(\text{left}) \rangle$ и $\langle \text{left}, \text{RIGHT}(\text{left}) \rangle$:

- нито един елемент от $A[i]$, който е извън $A[j]$, не е променян;
- $A[j] > A[\text{left}]$, така че $\langle j, \text{LEFT}(j) \rangle$ не може да е пирамидална инверсия;
- $A[j] \geq A[\text{right}]$, така че $\langle j, \text{RIGHT}(j) \rangle$ не може да е пирамидална инверсия;
- $A[\text{right}]$ не е бил модифициран;
- нито един от $A[\text{LEFT}(\text{left})]$ и $A[\text{RIGHT}(\text{left})]$ не е бил модифициран.

Но на ред 14, j става равен на left . Следователно, инвариантата е в сила при следващото достигане на ред 2.

Случай iii: $A[\text{left}] \not> A[j]$ и $A[\text{right}] > A[j]$ в началото на изпълнението на цикъла. Условието на ред 5 е ЛЪЖА пристояването на ред 9 се случва. Условието на ред 10 е TRUE, така че присвояването на ред 11 се случва и когато изпълнението е на ред 12, largest е равно на right . Условието на ред 12 е ИСТИНА и на ред 13, $A[j]$ и $A[\text{largest}]$ биват разменени. Твърдим, че единствените възможни пирамидални инверсии в $A[i]$ след размяната са $\langle \text{right}, \text{LEFT}(\text{right}) \rangle$ и $\langle \text{right}, \text{RIGHT}(\text{right}) \rangle$. Доказателството е точно като доказателството на аналогичното твърдение в **Случай I**. Но на ред 14, j става равен на largest . Следователно, инвариантата е в сила при следващото достигане на ред 2.

Случай IV: $A[\text{left}] > A[j]$ и $A[\text{right}] \not> A[j]$ в началото на изпълнението на цикъла. Условието на ред 5 е TRUE, така че присвояването на ред 7 се случва. Условието на ред 10 е FALSE, така че присвояването на ред 11 не се случва и когато изпълнението е на ред 12, largest е равно на left . Условието на ред 12 е TRUE и на ред 13, $A[j]$ и $A[\text{largest}]$ биват разменени. Твърдим, че единствените в $A[i]$ след размяната са $\langle \text{left}, \text{LEFT}(\text{left}) \rangle$ и $\langle \text{left}, \text{RIGHT}(\text{left}) \rangle$. Доказателството е точно като доказателството на аналогичното твърдение в **Случай II**. Но на ред 14, j става равен на largest . Следователно, инвариантата е в сила при следващото достигане на ред 2.

Случай V: $A[\text{left}] \not> A[j]$ и $A[\text{right}] \not> A[j]$ в началото на изпълнението на цикъла. Но това е невъзможно при текущите допускания, защото очевидно тогава ред 16 би бил достигнат и изпълнението не би се върнало повече на ред 2.

Терминация. Цикълът може да бъде напуснат по два начина: през ред 2, когато $j > \lfloor \frac{n}{2} \rfloor$, и през ред 16. Да разгледаме първата възможност. Тогава и $\text{LEFT}(j)$, и $\text{RIGHT}(j)$ са индекси извън $A[1, \dots, n]$. В този случай инвариантата казва, че няма пирамидални инверсии в $A[i]$ изобщо, понеже $A[\text{LEFT}(j)]$ и $A[\text{RIGHT}(j)]$ не съществуват. Следователно, $A[i]$ е пирамида.

Да разгледаме втората възможност, а именно **while**-цикълът да бъде напуснат през ред 16. Очевидно, за да бъде достигнат ред 16, трябва да бъде вярно, че $\text{largest} = j$ на ред 12. А за да е вярно това, и $A[\text{left}] \leq A[j]$, и $A[\text{right}] \leq A[j]$ трябва да са в сила в началото на изпълнението на цикъла, понеже това е единственият начин ред 9 да бъде достигнат и ред 11 да не бъде достигнат. Но ако $A[\text{left}] \leq A[j]$ и $A[\text{right}] \leq A[j]$, пирамидални инверсии в $A[i]$ няма, така че $A[i]$ е пирамида. \square

HEAPIFY в рекурсивен вариант

RECURSIVE HEAPIFY($A[1, 2, \dots, n]$: масив от цели числа, i : индекс в A)

1 $\text{left} \leftarrow \text{LEFT}(i)$

```

2   right  $\leftarrow$  RIGHT(i)
3   if left  $\leq n$  and  $A[\text{left}] > A[i]$ 
4     largest  $\leftarrow$  left
5   else
6     largest  $\leftarrow$  i
7   if right  $\leq n$  and  $A[\text{right}] > A[\text{largest}]$ 
8     largest  $\leftarrow$  right
9   if largest  $\neq i$ 
10    swap( $A[i], A[\text{largest}]$ )
11    RECURSIVE HEAPIFY( $A, \text{largest}$ )

```

Лема 25

При допускането, че $A[1, \dots, n]$ и i са такива, че всеки от $A[\text{LEFT}(i)]$ и $A[\text{RIGHT}(i)]$, ако съществува, е пирамида, ефектът от ITERATIVE HEAPIFY е, че $A[i]$ е пирамида при неговото приключване.

Доказателство: По индукция по височината h на $A[i]$.

Преди да направим доказателството, ще подчертаем следното: допусканията за $A[\text{LEFT}(i)]$ и $A[\text{RIGHT}(i)]$ –а именно, че са пирамиди –са нещо напълно различно от индуктивното предположение. Индуктивното предположение е твърдението, което доказваме, формулирано чрез някаква стойност на параметра, на която сме дали име. Индуктивното предположение не е част от условията на лемата, докато споменатите допускания са част от условията на лемата.

База. $h = 0$. Тогава $A[i]$ съдържа един единствен елемент A . С други думи, това е листо. Тогава и $\text{LEFT}(i)$, и $\text{RIGHT}(i)$ са индекси извън A . Да проследим изпълнението на RECURSIVE HEAPIFY: условието на ред 3 е ЛЪЖА, следователно присвояването на ред 6 се случва. Условието на ред 7 също е ЛЪЖА, така че ред 8 не се изпълнява и изпълнението преминава към ред 9. Условието там е ЛЪЖА и текущото рекурсивно извикване терминира. Очевидно $A[i]$ е пирамида при терминирането. ✓

Индуктивно предположение. Да допуснем, че за всяко $A[j]$ от височина $\leq h - 1$ с корен някой j , такъв че $A[j]$ е в $A[i]$, е изпълнено, че RECURSIVE HEAPIFY(A, j) превръща $A[j]$ чрез размествания в пирамида.

Индуктивна стъпка. Да разгледаме изпълнението на RECURSIVE HEAPIFY(A, i). Без ограничение на общността, да допуснем, че $\text{LEFT}(i) \leq n$ и $\text{RIGHT}(i) \leq n$, така че редове 3 и 7 са съответно

if $A[\text{left}] > A[i]$

и

if $A[\text{right}] > A[\text{largest}]$

Случай I: $A[\text{left}] > A[i]$, $A[\text{right}] > A[i]$ и $A[\text{right}] > A[\text{left}]$. Условието на ред 3 е ИСТИНА и присвояването на ред 4 се случва. Условието на ред 7 също е ИСТИНА, така че и присвояването на ред 8 се случва. Когато изпълнението е на ред 9, largest е равно

на $right$. Условието на ред 9 е ИСТИНА и на ред 10, $A[i]$ и $A[largest]$ биват разменени. Рекурсивното повикване на ред 11 се случва, като бившият $A[i]$ сега е на позиция $right$. Съгласно индуктивното предположение, това рекурсивно викане построява пирамида от от $A[right]$. От друга страна, $A[left]$ е пирамида, защото не е бил променян по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[right]$, е:

- по-голям от текущия $A[left]$ по допускане;
- не по-малък от текущия $A[right]$, по следните причини. Първоначално, $A[right]$ е пирамида, така че бившият $A[right]$ е не по-малък от всеки друг елемент от $A[right]$ в онзи момент (в началото). В края, елементите на $A[right]$ се състоят от началните елементи без началния $A[right]$, плюс началния $A[i]$. Тъй като началният $A[right]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[right]$, текущият $A[i]$ не е по-малък от текущия $A[right]$.

Тогава, $A[i]$ е пирамида.

Случай II: $A[left] > A[i]$, $A[right] > A[i]$ и $A[right] \not> A[left]$. Условието на ред 3 е ИСТИНА и присвояването на ред 4 се случва. Условието на ред 7 е ЛЪЖА, така че присвояването на ред 8 не се случва и когато изпълнението е на ред 9, $largest$ е равно на $left$. Условието на ред 9 е ИСТИНА и на ред 10, $A[i]$ и $A[largest]$ биват разменени. Рекурсивното извикване на ред 11 се случва, с бившия $A[i]$ сега на позиция $left$. Съгласно индуктивното предположение, това рекурсивно извикване построява пирамида от $A[left]$. От друга страна, $A[right]$ е пирамида, защото не е бил променян по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[left]$, е:

- не по-малък от текущия $A[right]$ по допускане
- не по-малък от текущия $A[left]$, по следните причини. Първоначално, $A[left]$ е пирамида, така че бившият $A[left]$ е не по-малък от всеки друг елемент на $A[left]$ в онзи момент (в началото). В края, елементите на $A[left]$ се състоят от началните елементи без началния $A[left]$, плюс началния $A[i]$. Тъй като началният $A[left]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[left]$, текущият $A[i]$ не е по-малък от текущия $A[left]$.

Тогава, $A[i]$ е пирамида.

Случай III: $A[left] \not> A[i]$ и $A[right] > A[i]$. Условието на ред 3 е ЛЪЖА и присвояването на ред 6 се случва. Условието на ред 7 е ИСТИНА, така че присвояването на ред 8 се случва и когато изпълнението е на ред 9, $largest$ е равно на $right$. Условието на ред 9 е ИСТИНА и на ред 10, $A[i]$ и $A[largest]$ биват разменени. Рекурсивното извикване на ред 11 се случва, с бившия $A[i]$ сега на позиция $right$. Съгласно индуктивното предположение, това рекурсивно извикване построява пирамида от $A[right]$. От друга страна, $A[left]$ е пирамида, защото не е бил променян по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[right]$, е:

- по-голям от текущия $A[left]$ заради допусканията и транзитивността на неравенства;
- не по-малък от текущия $A[right]$ по следните причини. Първоначално, $A[right]$ е пирамида, така че бившият $A[right]$ е не по-малък от всеки друг елемент на $A[right]$ в онзи момент (в началото). В края, елементите на $A[right]$ се състоят от началните елементи без началния $A[right]$, плюс началния $A[i]$. Тъй като началният $A[right]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[right]$, текущият $A[i]$ не е по-малък от текущия $A[right]$.

Тогава, $A[i]$ е пирамида.

Случай IV: $A[left] > A[i]$ и $A[right] \not> A[i]$. Условието на ред 3 е ИСТИНА и присвояването на ред 4 се случва. Условието на ред 7 е ЛЪЖА, така че присвояването на ред 8 не се случва и когато изпълнението е на ред 9, $largest$ е равно $left$. Условието на ред 9 е ИСТИНА и на ред 10, $A[i]$ и $A[largest]$ биват разменени. Рекурсивното извикване на ред 11 се случва, с бившия $A[i]$ сега на позиция $left$. Съгласно индуктивното предположение, това рекурсивно извикване построява пирамида от $A[left]$. От друга страна, $A[right]$ е пирамида, защото не е бил променян по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[left]$, е:

- по-голям от текущия $A[right]$ заради допусканията и транзитивността на неравенствата;
- не по-малък от текущия $A[left]$ поради следните причини. Първоначално, $A[left]$ е пирамида, така че бившият $A[left]$ е не по-малък от всеки друг елемент от $A[left]$ в онзи момент (в началото). В края, елементите на $A[left]$ се състоят от началните елементи без началния $A[left]$, плюс началния $A[i]$. Тъй като началният $A[left]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[left]$, текущият $A[i]$ не е по-малък от текущия $A[left]$.

Тогава, $A[i]$ е пирамида.

Случай V: $A[left] \not> A[i]$ и $A[right] \not> A[i]$. Условието на ред 3 е ЛЪЖА и присвояването на ред 6 се случва. Условието на ред 7 също е ЛЪЖА, така че присвояването на ред 8 не се случва и когато изпълнението достигне ред 9, $largest$ е равно на i . Условието на ред 9 е ЛЪЖА и изпълнението терминира, оставяйки $A[i]$ непроменен. Съгласно допусканията, $A[left]$ и $A[right]$ са пирамиди и $A[left] \not> A[i]$ и $A[right] \not> A[i]$. Тогава, $A[i]$ е пирамида. \square

Алгоритъм BUILD HEAP

Идеята на бързия алгоритъм за построяване на пирамида е да сканира масива отляво надясно, иначе казано, от листата към корена, започвайки от първото не-листо. Листата са тривиални пирамиди.

За всяко не-листо $A[i]$ в указания ред, при допускането, че и двете му деца (или едното му дете, ако е само едно) са корени на подпирамиди, ако $A[i]$ е по-голям или равен от двете си деца (или едното си дете), то $A[i]$ е подпирамида. Ако това не е вярно, ще направим $A[i]$ пирамида чрез серия от размени на елементи в нея, разменяйки корен с по-голямото дете, избутвайки малкия елемент, който образува пирамидални инверсии, в посока към листата, докато той или не стане листо, или не се окаже по-голям или равен на децата си (детето си).

Това всъщност означава, че викаме HEAPIFY върху всяко нелисто, отляво наляво. Функцията HEAPIFY в следния алгоритъм е или RECURSIVE HEAPIFY на стр. 127, или ITERATIVE HEAPIFY на стр. 125.

BUILD HEAP($A[1, 2, \dots, n]$: array of integers)

```

1   for  $i \leftarrow \lfloor \frac{n}{2} \rfloor$  downto 1
2       HEAPIFY( $A, i$ )

```

Лема 26

За всеки вход A , BUILD HEAP построява пирамида от него.

Доказателство: Следното твърдение е инвариант на **for**-цикъла (редове 1–2).

Инвариант 7: Цикълът на BUILD HEAP

Всеки път, когато изпълнението е на ред 1, $A[i + 1], A[i + 2], \dots, A[n]$ са пирамиди.

База. При първото достигане на ред 1, i е равно на $\lfloor \frac{n}{2} \rfloor$. Тогава $A[\lfloor \frac{n}{2} \rfloor + 1], A[\lfloor \frac{n}{2} \rfloor + 2], \dots, A[n]$ са точно листата на попълненото дърво, чиято линеаризация е A . Очевидно всяко листо е пирамида. ✓

Поддръжка. Да разгледаме някое достигане на ред 1, което не е последното. И $A[LEFT(i)]$, и $A[RIGHT(i)]$ (ако съществува) са пирамиди съгласно индуктивното предположение, понеже $i < LEFT(i)$ и $i < RIGHT(i)$. Тогава използваме доказаната коректност на HEAPIFY и заключаваме, че $A[i]$ става пирамида. След това i намалява с единица. Спрямо новата стойност на i , инвариантата е в сила. ✓

Терминация. Когато изпълнението е на ред 2 за последден път, $i = 0$. Тогава $A[0 + 1], A[0 + 2], \dots, A[n]$ са пирамиди. В частност, $A[1]$ е пирамида. Но $A[1] = A$. Това доказва лемата. ✓ □

Лема 27

BUILD HEAP работи във време $\Theta(n)$.

Доказателство:

Това, че сложността по време е $\Omega(n)$, е очевидно. Ще покажем, че сложността е $O(n)$. Асимптотична горна граница за сложността на BUILD HEAP е сумата по всички височини от 1 (BUILD HEAP прескача листата) до $\lfloor \lg n \rfloor$ (височината на дървото) от произведението от броя на върховете с дадената височина и самата височина (всеки връх може да се “придвижва” надолу към листата с най-много толкова размени, колкото е височината му поначало). Съгласно Следствие 13, броят на върховете с височина k е $\left\lceil \frac{\lfloor \frac{n}{2^k} \rfloor}{2} \right\rceil$. И така, ако $T(n)$ е функцията на сложността, то в най-лошия случай,

$$T(n) = \sum_{k=1}^{\lfloor \lg n \rfloor} \left\lceil \frac{\lfloor \frac{n}{2^k} \rfloor}{2} \right\rceil \Theta(k) = \sum_{k=1}^{\lfloor \lg n \rfloor} \Theta\left(\frac{n}{2^k} k\right)$$

Да разгледаме $\sum_{k=1}^{\lfloor \lg n \rfloor} \frac{n}{2^k} k$. В сила е

$$\sum_{k=1}^{\lfloor \lg n \rfloor} \frac{n}{2^k} k \leq \sum_{k=1}^{\infty} \frac{n}{2^k} k = n \sum_{k=1}^{\infty} \frac{k}{2^k}$$

Тривиално се доказва, че редът $\sum_{k=1}^{\infty} \frac{k}{2^k}$ е сходящ, примерно с критерия на d'Alembert:

$$\lim_{k \rightarrow \infty} \frac{\frac{k+1}{2^{k+1}}}{\frac{k}{2^k}} = \frac{1}{2} < 1$$

Щом $\sum_{k=1}^{\infty} \frac{k}{2^k}$ е ограничен от константа, то $n \sum_{k=1}^{\infty} \frac{k}{2^k} \asymp n$ и очевидно $\sum_{k=1}^{\lfloor \lg n \rfloor} \frac{n}{2^k} k \asymp n$. Тогава $T(n) \asymp n$. □

4.3 Сортиращ алгоритъм HEAPSORT

Следният сортиращ алгоритъм е предложен от Williams [73]. Той е първият от бързите сортиращи алгоритми, които ще разгледаме. “Бърз” наричаме алгоритъм, чиято сложност по време е $O(n \lg n)$.

Нека $A.size$ относно масива $A[1, \dots, n]$ е число, такова че $1 \leq A.size \leq n$ и HEAPIFY работи върху подмасива $A[1, \dots, A.size]$, а не върху $A[1, \dots, n]$.

$\text{HEAPSORT}(A[1, 2, \dots, n])$

- 1 BUILD HEAP(A)
- 2 $A.size \leftarrow n$
- 3 **for** $i \leftarrow n$ **downto** 2
- 4 swap($A[1], A[i]$)
- 5 $A.size \leftarrow A.size - 1$
- 6 HEAPIFY(A , 1)

Лема 28

HEAPSORT е сортиращ алгоритъм.

Доказателство: Нека $A'[1, \dots, n]$ означава първоначалния масив. Следното твърдение е инвариант на цикъла за **for**-цикъла (редове 3–6).

Инвариант 8: Цикълът на HEAPSORT

Всеки път, когато изпълнението на HEAPSORT е на ред 3:

- Текущият подмасив $A[i + 1, \dots, n]$ се състои от $n - i$ на брой най-големи елементи на $A'[1, \dots, n]$ в сортиран вид.
- Освен това, текущият $A[1, \dots, i]$ е пирамида.

База. При първото достигане на ред 3, $i = n$. Подмасивът $A[i + 1, \dots, n]$ е празен, следователно, в празния смисъл (*vacuously*), той се състои от нула на брой най-големи елементи от $A'[1, \dots, n]$, в сортиран вид, следователно първата част на инвариантата е в сила. $A[1, \dots, n]$ е пирамида съгласно Лема 26, приложена към ред 1, така че и втората част на инвариантата е в сила. ✓

Поддръжка. Да допуснем, че твърдението е в сила при някакво достигане на ред 3, което не е последното. Нека да наричаме масива A в този момент, A'' . Съгласно първата част на индуктивното предположение, $A''[i + 1, \dots, n]$ се състои от $n - i$ на брой елементи от A' в сортиран вид. Съгласно втората част на индуктивното предположение, $A''[1]$ е максимален елемент от $A''[1, \dots, i]$. След размяната на ред 4, $A''[i, \dots, n]$ съдържа $n - i + 1$ на брой най-големи елемента на A' в сортиран вид. Спрямо новата стойност на i при следващото достигане на ред 3, първата част на инвариантата е в сила.

Ще докажем, че втората част на инвариантата е в сила. Да приложим Лема 24 или Лема 25 в зависимост от това дали ползваме рекурсивна или итератична HEAPIFY функция на ред 6. Трябва да се има предвид, че HEAPIFY работи върху $A[1, \dots, i - 1]$, защото i е равно на $A.size$.

в момента, в който изпълнението е на ред 6, а на ред 5, $A.size$ е получил стойност $i - 1$. Заради това, на ред 6 текущият $A[i]$ е “извън обхвата” на HEAPIFY.

Терминация. Да разгледаме момента, в който изпълнението е на ред 3 за последен път. Очевидно, $i = 1$. Да заместим 1 на мястото на i в инвариантата. Получаваме “текущият подмассив $A[2, \dots, n]$ се състои от $n - 1$ на брой най-големи елементи на $A'[1, \dots, n]$ в сортиран вид”. Но тогава $A[1]$ е минимален елемент на $A'[1, \dots, n]$. С това доказателството за коректност на HEAPSORT приключва. \square

Да разгледаме сложността по време на HEAPSORT. Сложността на BUILD HEAP е $\Theta(n)$, което доказваме в Лема 27. Сложността на **for**-цикъла (редове 3–6) $\Theta(n \lg n)$, защото за всяко i , сложността на HEAPIFY в най-лошия случай е $\Theta(\lg i)$, а ние вече изследвахме сумата $\sum_{i=1}^n \lg i$ (вж. (4.1)) и установихме, че асимптотично ѝ нараствае $\Theta(n \lg n)$. Следователно, сложността на HEAPSORT е $\Theta(n) + \Theta(n \lg n) = \Theta(n \lg n)$.

Сложността по памет на HEAPSORT е $\Theta(1)$.

HEAPSORT не е стабилен сортиращ алгоритъм. За да се убедим в това, достатъчно е да разгледаме работата му над вход от еднакви ключове. При дадения псевдокод, BUILD HEAP няма да промени нищо, след което още първото изпълнение на **for**-цикъла ще размени еднаквите елементи $A[1]$ и $A[n]$, като този елемент, който бива записан в $A[n]$, ще остане там до края на алгоритъма.

Дори да сложим проверка на ред 4, такава размяната да не се изпълнява, ако $A[1]$ и $A[i]$ са равни, това няма да направи алгоритъма стабилен. Да си представим вход от k ключа 2 и k ключа 1, като двойките са преди единиците. BUILD HEAP няма да промени нищо, след което още първото изпълнение на **for**-цикъла ще размени двойката в $A[1]$ с единицата в $A[n]$, при което двойката, която е била вляво от всички други двойки, ще се окаже вдясно от тях, и ще остане така до края на алгоритъма.

4.4 Приоритетни опашки

4.4.1 Абстрактен Тип Данни (АТД)

Приоритетна опашка (на английски, *priority queue*) е вид *абстрактен тип данни* (на английски, *abstract data type*, или ADT). Първо ще изясним понятието абстрактен тип данни (АТД). Според Sedgewick и Wayne [61, стр. 64]:

An abstract data type (ADT) is a data type whose representation is hidden from the client.

...

When using an ADT, we focus on the operations specified in the API and pay no attention to the data representation; when implementing an ADT, we focus on the data, then implement operations on that data.

To specify the behavior of an abstract data type, we use an application programming interface (API), ...

И така, АТД е множество от данни, обикновено от един и същи тип, заедно с множество функции върху тях, което множество от функции се нарича *интерфейс*. “Абстрактен” означава, че конкретната реализация остава скрита, а цялото “общуване” с данните става през интерфейса. Конкретната реализация би могла да бъде променяна, като това остава прозрачно за софтуера, който ползва АТД-то (или поне на теория софтуерът-потребител не би трябвало да “усеща” разликата, ако няма бъгове и интерфейсът на новата реализация е изграден

съгласно спецификациите). Конкретната реализация остава напълно скрита зад интерфейса и можем да си представяме АТД-то като черна кутия (*black box*) плюс интерфейс. Прости примери за АТД са *стек* и *опашка*, наричани още съответно LIFO и FIFO [61, стр. 121] (вж. Фигура 4.16 за илюстрация на черна кутия с интерфейс).

4.4.2 Приоритетна опашка: вид АТД

Приоритетните опашки имат елементи, всеки от които има две стойности: ключ и данни, свързани с този ключ. Ключовете обикновено са цели числа, макар че всеки тип данни, който може да се използва в контекста на сортирането, може да е типът данни на ключовете. Приоритетните опашки, подобно на пирамидите, са максимални и минимални. Тук ще дискутираме само максимални приоритетни опашки, като изложението за минималния тип е напълно аналогично. Казвайки “приоритетна опашка”, имаме предвид максимална приоритетна опашка.

Интерфейсът на приоритетните опашки обикновено се ограничава до следните три функции. Ако приоритетната опашка се назова S , функциите от интерфейса са:

- MAXIMUM(S): връща елемента с максимален ключ;
- EXTRACT-MAX(S): връща елемента с максимален ключ и го премахва от S ;
- INSERT(S, x): вмъква в S елемент x , който е от подходящ за S тип.

Написаното тук е според учебника на Cormen и др. [15, стр. 162]. Авторите дават и четвърта функция от интерфейса, а именно INCREASE-KEY(S, x, k), която има смисъл на “увеличи ключа на x на нова стойност k , при допускането, че тя е не по-малка от стария ключ на x ”. Тази функция, която ще дискутираме след малко, не е част от стандартния интерфейс на приоритетните опашки. Sedgewick и Wayne [61, стр. 309] дават следния изчерпателен списък на интерфейса на реални (които се ползват на практика) приоритетни опашки. Описанието е на езиковите конвенции на Java.

- `MaxPQ():` create a priority queue
- `MaxPQ(int max):` create a priority queue of initial capacity `max`
- `MaxPQ(Key[] a):` create a priority queue from the keys in `a[]`
- `void insert(Key v):` insert a key into the priority queue
- `Key max():` return the largest key
- `Key delMax():` return and remove the largest key
- `boolean isEmpty():` is the priority queue empty?
- `int size():` number of keys in the priority queue

За целите на тази лекция, интерфейсът е според учебника на Cormen и др., като INCREASE-KEY не е част от него, освен ако това не е казано изрично. Фигура 4.16 илюстрира приоритетна опашка като черна кутия плюс интерфейс.

Приложенията на приоритетните опашки са много. Всяка дейност, в която трябва да бъдат обработвани някакви неща, пристигащи асинхронно, като в даден момент може да се обработва само едно нещо, може да се моделира с приоритетна опашка. Нещата може да са,



Фигура 4.16: АТД приоритетна опашка като черна кутия. Конкретната имплементация е скрита. Отвън се “виждат” само трите функции на интерфейса.

`print jobs` в операционна система, кораби за разтоварване в пристанище, самолети, кацащи на летище, спешно отделение в болница, и така нататък. С приоритетни опашки може и да се сортира: HEAPSORT можем да разглеждаме като сортиращ алгоритъм с последователно избиране на максимален елемент (какъвто е SELECTION SORT), ползваш ефикасно реализирана приоритетна опашка.

4.4.3 Реализация на приоритетни опашки с двоични пирамиди

Както ще видим след малко, двоичните пирамиди са много подходящи за реализация на приоритетни опашки, но в никакъв случай не са единствената възможна реализация. Приоритетна опашка може да бъде реализирана и чрез следните структури данни.

- Сортиран масив. На читателя остава да измисли имплементация на трите функции от интерфейса. Тук само ще споменем, че `MAXIMUM(S)` и `EXTRACT-MAX(S)` при тази реализация имат сложност $\Theta(1)$, докато `INSERT(S, x)` има сложност $\Theta(n)$.
- Несортиран масив. И тук остава на читателя да измисли имплементация на трите функции от интерфейса. При тази реализация, `MAXIMUM(S)` и `EXTRACT-MAX(S)` имат сложност $\Theta(n)$, докато `INSERT(S, x)` има сложност $\Theta(1)$.
- Пирамида на Фибоначи. Това е сложна структура данни, даваща възможност за изключително ефикасна реализация на функциите от интерфейса. Нещо повече, тази структура данни позволява интерфейс от по-голямо множество функции, които включват изтриване на елемент и смесване (merge) на две пирамиди. За съжаление, реализацията на функциите от интерфейса е ефикасна само в асимптотичния смисъл. Големите скрити константи правят пирамидите на Фибоначи неефикасни на практика, освен за наистина много големи данни. За подробности, вижте [15, стр. 506].
- Двоични дървета, реализирани чрез списъци с дърводидна структура (по два указателя към децата за всеки запис). Подробна дискусия на предимствата, недостатъците и вариациите на този подход има в третия том на култовата поредица на Доналд Кнут “Изкуството на Компютърното Програмиране” (The Art of Computer Programming) [38, стр. 149]. От общи съображения е ясно, че този подход изисква значително—по-точно, линейно—повече памет от реализацията с масив. Измежду неговите предимства се открява възможността за бързо сливане (merge) на приоритетни опашки.

Трите функции на интерфейса може да бъдат реализирани чрез двоични пирамиди по следния начин. Входът S е масив $S[1, \dots, n]$, реализиращ пирамида. $S.size$ е размерът на пирамидата, тоест, само $S[1, \dots, S.size]$ е пирамида. Допускаме, че винаги $S.size < n$, така че

няма да се притесняваме за препълване на масива в HEAP-INSERT. Ако пирамидата е празна, както е в самото начало при създаването ѝ, $S.size = 0$.

HEAP-MAXIMUM($S[1, \dots, n]$)
 1 (* Допускаме, че $S.size \geq 1$ *)
 2 **return** $S[1]$

HEAP-EXTRACT-MAX($S[1, \dots, n]$)
 1 (* Допускаме, че $S.size \geq 1$ *)
 2 $max \leftarrow S[1]$
 3 $S[1] \leftarrow S[S.size]$
 4 $S.size \leftarrow S.size - 1$
 5 HEAPIFY($S, 1$)
 6 **return** max

HEAP-INSERT($S[1, \dots, n], x$)
 1 $S.size \leftarrow S.size + 1$
 2 $i \leftarrow S.size$
 3 $S[i] \leftarrow x$
 4 **while** $i > 1$ **and** $S[PARENT(i)] < S[i]$ **do**
 5 swap($S[PARENT(i)], S[i]$)
 6 $i \leftarrow PARENT(i)$

Коректността на тези функции е абсолютно очевидна след изложението в Секция 4.2. Сложността на HEAP-MAXIMUM е $\Theta(1)$, а на HEAP-EXTRACT-MAX и HEAP-INSERT е $\Theta(\lg(S.size))$. Ако не ползваме “ n ” за големината на масива, а за големината на самата пирамида, и изразяваме сложността чрез символа “ n ”, то двете функции имат сложност $\Theta(\lg n)$. Имайки предвид огромната, качествена разлика между нарастванията $\Theta(n)$ и $\Theta(\lg n)$, виждаме, че реализацията с двоични пирамиди е много по-добра от реализацията със сортирани или несортирани масиви.

Оттук правим важен извод, който е приложим в много по-широк контекст от ефикасното реализиране на приоритетните опашки. В някои случаи (реализиране на приоритетна опашка), оптималният подход (пирамида) е компромис (на английски, *tradeoff*) между два екстремални подхода (сортиран масив и несортиран масив). Относно две различни изисквания (бързо изваждане на максимален елемент и бързо вмъкване на нов елемент), при единият екстремален подход имаме оптимален резултат ($\Theta(1)$) по първото изискване, но изключително лош резултат ($\Theta(n)$) по второто; а при другия подход, оптимален резултат по второто изискване, но изключително лош резултат по първото. Компромисният подход (отказ от константно време за коя да е от операциите и постигане на задоволително време и за двете операции), направен интелигентно, се оказва печеливш.

4.4.4 Функцията INCREASE-KEY

Според [15], функцията от интерфейса INCREASE-KEY може да се реализира с двоична пирамида по следния начин.

HEAP-INCREASE-KEY($S[1, \dots, n], i, k$)

```

1 if k < S[i]
2   error "опит да бъде намален ключът"
3 S[i] ← k
4 while i > 1 and S[PARENT(i)] < S[i] do
5   swap(S[PARENT(i)], S[i])
6   i ← PARENT(i)

```

Очевидно, тази имплементация не може да е част от интерфейса на АТД, защото в нея е заложено, че имплементацията е именно чрез масив. В случая параметърът i е индексът на елемента, чийто ключ ще нараства. Ако имплементацията на приоритетната опашка е чрез някакви свързани списъци, този индекс няма смисъл.

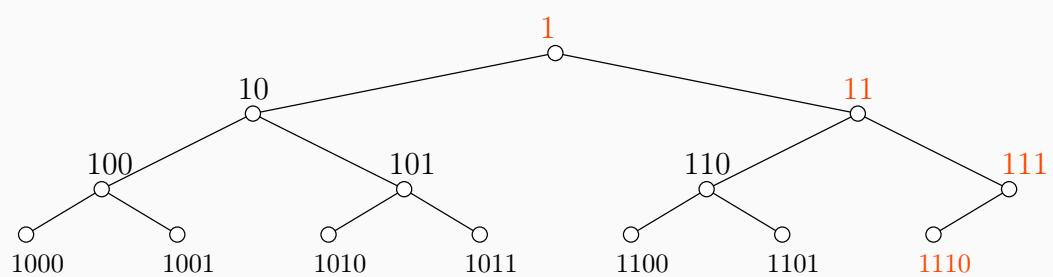
Правилното викане на функцията би било $\text{INCREASE-KEY}(S, x, k)$, където x е елемент от опашката. Но ако реализираме функцията с пирамида и дадем параметър x , който е елемент (с други думи, ако x е старата стойност на ключа, която трябва да стане k), този елемент трябва да бъде първо открит, което е операция с линейна сложност в пирамидите[†], дори ако допуснем уникални ключове. Така че, за да постигнем логаритмична сложност и на тази функция от интерфейса, се налага да "счупим" изискването за абстрактен тип данни и да заложим в нейната спецификация, че приоритетната опашка е реализирана именно чрез масив и че някак знаем индекса на елемента, чийто ключ искаме да повишим.

Допълнение 17: За броя на пирамидите на n върха

В това допълнение ще разгледаме интересния въпрос, колко от всички $n!$ перmutации на n различни числа са пирамиди, по-точно, линеаризации на пирамиди. Изложението следва изложението в [38, стр. 152].

Да разгледаме отново попълненото двоично дърво с адресите от Подсекция 4.1.2, което за удобство показваме отново на Фигура 4.17.

Фигура 4.17 : Попълнено дърво с 14 върха и техните адреси. Специалните върхове са в червено.



Дървото има 14 върха. Да запишем броевете на върховете на поддърветата, вкоренени във всички върхове на дървото, в нарастващ ред по адресите на тези върхове. Ще записваме броевете в двоична позиционна бройна система:

1110, 111, 110, 11, 11, 11, 10, 1, 1, 1, 1, 1, 1

За да е напълно ясно: цялото дърво (тоест, поддървото с корен върха, чийто адрес е 1) има четиринаесет върха, което в двоична позиционна бройна система е 1110. Поддървото, вкоренено в следващия връх (с адрес 10) има седем върха, така че записваме

[†]Примерно, най-малкият ключ може да е във всяко листо

111. И така нататък. С червен цвят са записани броевете на върховете в тези поддървета, чиито корени лежат на пътя между корена на дървото и последния връх (чийто адрес е $\text{bin}(n)$). Тези върхове са наречени в [38], *специалните върхове*, а дърветата, вкоренени в тях, *специалните поддървета*. В [38, стр. 157, упр. 20 и 21] е показано, че ако $\text{bin}(n) = b_k b_{k-1} \dots b_1 b_0$, където $k = \lfloor \lg n \rfloor$ и $b_k = 1$, то размерите^a на специалните поддървета, записани в двоична позиционна бройна система, са

$$1b_{k-1} \dots b_1 b_0, \quad 1b_{k-2} \dots b_1 b_0, \quad \dots \quad 1b_1 b_0, \quad 1b_0, \quad 1$$

а размерите на неспециалните поддървета числа от вида $2^m - 1$, защото те са съвършени двоични дървета, и броевете на неспециалните поддървета от всяка срещаща се големина са съответно

$$\begin{aligned} & \left\lfloor \frac{n-1}{2} \right\rfloor \text{ числа } 1, \\ & \left\lfloor \frac{n-2}{4} \right\rfloor \text{ числа } 3, \\ & \left\lfloor \frac{n-4}{8} \right\rfloor \text{ числа } 7, \\ & \quad \dots \\ & \left\lfloor \frac{n-2^{k-1}}{2^k} \right\rfloor \text{ числа } 2^k - 1 \end{aligned}$$

Знаяки размерите на поддърветата, можем да изчислим H_n : броя на начините да разположим ключовете $\{1, 2, \dots, n\}$ в пирамида, като използваме резултата от [38, стр. 67, упр. 20]:

$$H_n = \frac{n!}{s_1 \times s_2 \times \dots \times s_n} \tag{4.2}$$

където $\{s_1, s_2, \dots, s_n\}_M$ е мултимножеството от всички размери на поддървета (вж. Определение 27). Примерно, ако елементите са 14, каквото е дървото на Фигура 4.17, начините да сложим ключовете $\{1, 2, \dots, 14\}$, така че дървото да е пирамида, са на брой

$$H_{14} = \frac{14!}{14 \times 6 \times 2 \times 1 \times 1^6 \times 3^3 \times 7^1} = 2745600$$

Сравнете H_{14} с $14! = 87178291200$.

Редицата H_n за $n \geq 1$ е редица A056971 в онлайн енциклопедията на целочислените редици. Точният израз (4.2) не дава представа за асимптотиката на H_n . Асимптотиката на H_n се разглежда в [27]. Основният резултат е твърде сложен, но грубо приближение е следното. Ако $h_n = \log \left(\frac{n!}{H_n} \right)$, то $h_n \approx n$. Оттук—това е грубо приближение—асимптотиката на H_n е приблизително $\frac{n!}{e^n}$ за някаква константна c .

^aТова са общо $k = \lfloor \lg n \rfloor$ числа, колкото е височината на попълненото двоично дърво с n върха съгласно Лема 18.

Лекция 5

Рекурсивни Алгоритми, алгоритмична схема Разделяй-и-Владей и Рекурентни Уравнения.

Резюме: Въвеждаме понятията рекурсивен алгоритъм и рекурентно уравнение. Въвеждаме алгоритмичната схема Разделяй-и-Владей. Разискваме най-обща класификация на рекурентните уравнения. Разглеждаме шест начина за решаване на рекурентни уравнения: чрез разделяне, чрез дървото на рекурсията, по индукция, чрез Мастър теоремата, чрез Теорема 20 и чрез характеристични уравнения. Разглеждаме прости примери за рекурсивни алгоритми, конструирани по схемата Разделяй-и-Владей.

5.1 Рекурсивни алгоритми и Разделяй-и-Владей

Думата *рекурсия* идва от латинския глагол *cirrō*[†], което означава “тичам”, и префикс *re-*, който означава “назад” или “отново”. *Recurro* означава “тичам назад”, откъдето *рекурсивен* буквально означава “тичащ назад”.

Рекурсивен алгоритъм е алгоритъм, който вика себе си. Очевидно, ако детерминистичен алгоритъм извика себе си с точно същия вход, с който е бил извикан, той ще неизбежно ще вика себе си отново и отново със същия вход до безкрай, без да терминира никога. Поради това викане на себе си със същия вход е недопустимо.

По правило рекурсивните алгоритми са реализация на *алгоритмичната схема* Разделяй-и-Владей[‡]. При нея алгоритъмът се състои от три фази:

- **Разделяй** – ако входът е достатъчно голям, то той се разделя[§] на части и се преминава към следващата стъпка **Владей**; в противен случай, тоест ако входът е по-малък от някаква предварително зададена големина, задачата върху него се решава по някакъв тривиален начин или с “брутална сила” и се преминава директно към стъпка **Комбинирай**.

[†]Откъдето идва, например, и думата “курier”.

[‡]“Алгоритмична схема” означава общ шаблон, по който са конструирани множество алгоритми за множество задачи. В този курс ще разглеждаме три алгоритмични схеми: **Разделяй-и-Владей**, **Алчна схема** и **Динамично Програмиране**. В тази лекция ще се спрем на схемата Разделяй-и-Владей.

[§]Терминът “разбива” не е подходящ, защото общоприетата дефиниция на “разбива” означава, на части, които не се припокриват. При алгоритмите, изградени по схемата **Разделяй-и-Владей**, не е задължително частите на входа да не се припокриват, просто всяка трябва да е по-малка от целия вход.

- **Владей** – решаваме задачата върху всяка от по-малките части и получаваме резултатите от всяко решение.
- **Комбинирај** – използваме тези резултати, за да генерираме решение за задачата върху първоначалния вход.

Да повторим: първата фаза (на разделянето) става само докато размерът на входа не стане достатъчно малък, защото няма как да разделяме дискретен вход на по-малки части неограничено. Разделянето се случва ако и само ако размерът на входът надхвърля еди-колко; в противен случай, генерираме резултат по друг начин, например с пълно изчерпване на възможностите, който метод наричаме “с брутална сила”[†].

Както ще видим в многообразни алгоритмични примери, някои изчислителни задачи са податливи на ефикасно алгоритмично решение, което първо разделя входа на части, стига той да е достатъчно голям, решава задачата върху всяка от частите, и накрая комбинира решението. За съжаление, този подход съвсем не е универсален. За много задачи—по правило, за всички много тежки изчислителни задачи—схемата **Разделяй-и-Владей** не е приложима[‡].

Типичен пример за рекурсивен алгоритъм е следният алгоритъм, пресмятащ факториел на число.

`FACTORIAL($n \in \mathbb{N}$)`

```

1   if  $n = 0$ 
2       return 1
3   else
4       return  $n \times \text{FACTORIZATION}(n - 1)$ 
```

Да разгледаме работата му върху някакъв малък вход, например 5.

- FACTORIAL(5) проверява условието на ред 1 и понеже то е ЛЪЖА, на ред 4 изпълнява FACTORIAL(4).
- FACTORIAL(4) проверява условието на ред 1 и понеже то е ЛЪЖА, на ред 4 изпълнява FACTORIAL(3).
- FACTORIAL(3) проверява условието на ред 1 и понеже то е ЛЪЖА, на ред 4 изпълнява FACTORIAL(2).
- FACTORIAL(2) проверява условието на ред 1 и понеже то е ЛЪЖА, на ред 4 изпълнява FACTORIAL(1).
- FACTORIAL(1) проверява условието на ред 1 и понеже то е ЛЪЖА, на ред 4 изпълнява FACTORIAL(0).

[†] Терминът “Разделяй-и-Владей” (на английски, *Divide and Conquer*) идва от латинската фраза *Divide et impera*. Според наложилия се фолклор, тази максима е била ръководен принцип на древния Рим—и кралство, и републиката, и империята—който се е справял с даден силен противник, като първо е създавал вътрешни разделения в него, а после е побеждавал всеки от получените фрагменти последователно. Легендарната битка между братята Хорации и братята Куриации е спечелена именно с прилагане на разделяй-и-владей.

[‡] Както ще видим в следваща лекция, алгоритмичната схема **Динамично Програмиране** също е свързана с рекурсия, но при онези алгоритми акцентът е върху ефикасното пресмятане, което означава умело използване на вече получени резултати на общи подподзадачи.

- FACTORIAL(0) проверява условието на ред 1 и понеже то е ИСТИНА, на ред 2 връща стойност 1 на FACTORIAL(1).
- FACTORIAL(1) получава стойност 1 от викането на FACTORIAL(0), след което умножава 1×1 и връща резултата, който е 1, на FACTORIAL(2).
- FACTORIAL(2) получава стойност 1 от викането на FACTORIAL(1), след което умножава 2×1 и връща резултата, който е 2, на FACTORIAL(3).
- FACTORIAL(3) получава стойност 2 от викането на FACTORIAL(2), след което умножава 3×2 и връща резултата, който е 6, на FACTORIAL(4).
- FACTORIAL(4) получава стойност 6 от викането на FACTORIAL(3), след което умножава 4×6 и връща резултата, който е 24, на FACTORIAL(5).
- FACTORIAL(5) получава стойност 24 от викането на FACTORIAL(4), след което умножава 5×4 и връща 120.

Работата на FACTORIAL(5) се състои от два етапа. Първият етап можем да наречем, *движение надолу*. През него входът намалява, докато не бъде достигната стойността 0, определена от ред 1. Тази стойност можем да наречем, *спирачката на рекурсията*. Вторият етап да наречем, *движение нагоре*. Същинското изчисляване на факториела става във втория етап. По отношение на софтуерна имплементация на алгоритъма, първият етап е само попълване на подходяща структура данни, а именно стек, с подходящи стойности. Името “рекурсия”, което, както казахме, буквално означава “тичане назад”, без съмнение се дължи на втория етап – движението нагоре.

5.2 Рекурентни уравнения

5.2.1 Определение на рекурентно уравнение

Рекурентно уравнение е уравнение, в което вляво от знака за равенство има символ за функция, например $T(n)$ или a_k , а вдясно има математически израз, който включва една или повече появи на същия символ за функция, но с по-малки стойности на променливата. Алтернативна дефиниция е, уравнение, което изразява общия член на някаква редица от числа чрез някои или всички предишни членове. Освен това, за една или повече *начални стойности* на променливата, има други уравнения, наречени *начални условия*, които определят стойността на въпросната функция (върху началните стойности).

Можем да мислим за рекурентните уравнения като за рекурсивни алгоритми. Наистина, всяко рекурентно уравнение е рекурсивен алгоритъм, който има вход-число и изход-число. По правило, рекурентните уравнения са твърде елементарни алгоритми, защото нямат цикли и вложеност на *if*-овете. Важно е да различаваме алгоритъма-рекурентно уравнение от другия алгоритъм – онзи, чиято сложност се описва от уравнението (вижте Наблюдение 14).

Типичен пример е рекурентното уравнение на числата на Фиbonачи:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \text{ за } n \geq 2 \end{aligned} \tag{5.1}$$

Еквивалентен запис е:

$$F_n = \begin{cases} 0, & \text{ако } n = 0 \\ 1, & \text{ако } n = 1 \\ F_{n-1} + F_{n-2}, & \text{ако } n > 1 \end{cases}$$

Добре дефинирано рекурентно уравнение ще наричаме рекурентно уравнение, което задава коректен (рекурсивен) алгоритъм, който алгоритъм пресмята елементи на някаква безкрайната чисрова редица[†], да кажем

$$a_0, a_1, a_2, \dots$$

На читателя остава да съобрази, че следното не е добре дефинирано уравнение заради “недостиг” на начални условия:

$$\begin{aligned} T(0) &= 1 \\ T(n) &= 2T(n - 3) \text{ за } n \geq 1 \end{aligned}$$

Въпросът дали дадено рекурентно уравнение е добре дефинирано, дали началните условия са достатъчни и смислени, може да е нетривиален. Следното рекурентно уравнение не е добре дефинирано

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T\left(\left\lfloor \frac{n}{2} + 17 \right\rfloor\right) + n \text{ за } n > 1 \end{aligned}$$

поради това, че трансформацията $n \mapsto \left\lfloor \frac{n}{2} + 17 \right\rfloor$ [‡], започвайки от произволно естествено число n , се “установява” на $n = 33$ или $n = 34$,[§] така че началната стойност 1 не е достижима от никое число освен самото 1. С други думи, началното условие $T(1) = 1$ в случая е безполезно! Съответната програма ще “гърми” за всеки вход, различен от 1[¶].

И така, всяко добре дефинирано рекурентно уравнение определя еднозначно чисрова редица. Отношението между рекурентни уравнения и числови редици не е биективно: една и съща чисрова редица може да се определя от много рекурентни уравнения. Рекурентни уравнения, които определят една и съща чисрова редица, ще наричаме *еквивалентни*. Убедете се сами, че рекурентното уравнение (5.2) е еквивалентно на (5.1), защото и на двете съответства редицата на Фиbonачи $0, 1, 1, 2, 3, 5, 8, \dots$.

$$\begin{aligned} T(0) &= 0 \\ T(1) &= 1 \\ T(2) &= 1 \\ T(n) &= 2T(n - 1) - T(n - 3) \quad \text{за } n \geq 3 \end{aligned} \tag{5.2}$$

[†]Допускаме, че функционалният символ е “ a ” и че най-малката начална стойност на променливата е 0.

[‡]Чете се, “ n се изобразява в $\left\lfloor \frac{n}{2} + 17 \right\rfloor$ ”.

[§]Казваме, че 33 и 34 е *фиксираните точки* на тази трансформация.

[¶] Става дума за препълване на стека, който операционната система осигурява на съответния процес. Опитайте следния фрагмент на C:

```
int recfun(int n) {if (n == 1) return 1; else return 2*recfun(floor(n/(2.0)) + 17) + n;}
```

върху $n = 1$ и $n \neq 1$.

Забележка

Подчертаваме, че не трябва да се бъркат двете понятия

- рекурентно уравнение, което е понятие от синтактичното ниво
- функцията, която се реализира от някакво рекурентно уравнение – това е понятие от семантичното ниво. Числовата редица, която съответства на рекурентното уравнение, е практически същото нещо като функцията, която то реализира.

5.2.2 Обобщения на рекурентно уравнение

Има различни обобщения на понятието “рекурентно уравнение”, което току-що въведохме. Рекурентните уравнения могат да бъдат на повече от една променлива, например биномният коефициент е рекурентно уравнение на две променливи:

$$\binom{n}{0} = 1, \quad \binom{n}{n} = 1, \quad \binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Може да бъдат дадени няколко взаимно зависими рекурентни уравнения – това е проявление на *взаимна рекурсия*. Пример за такива рекурентни уравнения има в [39, стр. 9] (за краткост изпускаме началните условия):

$$\begin{aligned} T(n) &= T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) + 2T_1\left(\left\lceil \frac{n}{2} \right\rceil - 1\right) + 2T_1\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2 \\ T_1(n) &= T\left(\left\lfloor \frac{n}{2} + 1 \right\rfloor\right) + T_1\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2T_1\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2T_2\left(\left\lceil \frac{n}{2} \right\rceil - 1\right) + 2 \\ T_2(n) &= T_1\left(\left\lceil \frac{n}{2} \right\rceil\right) + T_1\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) + 2T_2\left(\left\lceil \frac{n}{2} \right\rceil - 1\right) + 2T_2\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2 \end{aligned}$$

Друг пример за взаимна рекурентност има [на тази страница](#) на специализирано списания за числа на Фиbonачи, където е дадена задачата, да се докаже, че H_{n-1} и G_n са последователни числа на Фиbonачи $\forall n \geq 1$, ако F_n са числата на Фиbonачи:

$$G_1 = 1$$

$$H_0 = 0$$

$$G_n = F_{n+1}G_{n-1} + F_nH_{n-2}, \quad n \geq 2$$

$$H_n = F_{n+1}G_n + F_nH_{n-1}, \quad n \geq 1$$

Може да се дефинират рекурентни уравнения върху рационални или реални числа. Досега допускахме, че стойността на функцията е естествено число, но това не е задължително. Например рекурентното уравнение (вж. [60, стр. 56])

$$a_0 = 1$$

$$a_n = \frac{1}{1 + a_{n-1}}$$

определя редицата от дроби $1, \frac{1}{2}, \frac{2}{3}, \frac{3}{5}, \frac{5}{8}, \dots$, която очевидно има връзка с числата на Фиbonачи. Пак в книгата на Flajolet и Sedgewick се посочва (вж. [60, стр. 46]), че методът на Newton за пресмятане на квадратен корен от дадено положително реално β е рекурентно уравнение с $a_0 = 1$:

$$a_n = \frac{1}{2} \left(a_{n-1} + \frac{\beta}{a_{n-1}} \right), \quad n > 0$$

5.2.3 Рекурентни уравнения и рекурсивни алгоритми

Връзката между рекурентните уравнения и рекурсивните алгоритми е сложна. От една страна, всяко рекурентно уравнение е рекурсивен алгоритъм, който за всеки вход-число връща изход-число след пресмятане, което включва рекурсия (викане на себе си върху по-малко число). От друга страна, сложността на рекурсивните алгоритми по правило се описва от рекурентни уравнения. Например, сложността на MERGESORT, който ще разгледаме в следваща лекция, се описва от рекурентното уравнение (както ще видим нататък, началните условия се пропускат, защото ни интересува само асимптотиката на решението):

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \quad (5.3)$$

При изследването на сложността на рекурсивни алгоритми чрез рекурентни уравнения е важно да се прави разлика между алгоритъма, чиято сложност изследваме, и рекурентното уравнение, което описва тази сложност. На свой ред то също е рекурсивен алгоритъм, но е съвършено различен обект от първоначалния алгоритъм. Като пример да разгледаме следното рекурентното уравнение, където $n \in \mathbb{N}$:

$$T(n) = \begin{cases} 1, & \text{ако } n = 0 \\ nT(n-1) + 1, & \text{в противен случай} \end{cases} \quad (5.4)$$

То е много близко до алгоритъм FACTORIAL на стр. 140. Сега да разгледаме следния алгоритъм. Той не прави нищо полезно, но ще ни послужи за илюстрация.

```
ALGX(A[1, ..., n])
1 if n = 0
2   return 1
3 else
4   s ← A[1]
5   for i ← 1 to n
6     s = s + ALGX(A[1, ..., n - 1])
7   return s
```

Сложността на ALGX се описва от (5.4). Обаче (5.4) е съвършено различен обект от ALGX. Говорейки прагматично, върху вход с размер 100 ALGX няма приключи работата си **никога**, в рамките на живота на Земята, докато $T(100)$ ще бъде пресметнато **мигновено** върху съвременен компютър[†]. Заслужава си да обособим това наблюдение по следния начин.

Наблюдение 14

Ако е даден рекурсивен алгоритъм ALGX, чиято сложност по време се описва от рекурентно уравнение Y , то Y също е рекурсивен алгоритъм. Обаче ALGX и Y са съвършено различни обекти. Най-малкото, входът на ALGX може да не е число, а масив или друг обект, докато входът на Y —това е променливата на уравнението—е число. Освен това, по правило ALGX върху вход с големина n работи несравнено по-бавно от Y с вход числото n .

[†]Стига софтуерът да може да поддържа работа с цели числа с произволна точност (на английски, *arbitrary-precision arithmetic*), за да не се стигне до препълване на променливите заради огромните числа-факториели.

5.2.4 Класифициране на рекурентните уравнения

Тук ще игнорираме началните условия и ще дискутираме “същинското” рекурентно уравнение. Рекурентните уравнения може да се класифицират по много признаки.

- **Брой на появите на функционалния символ вдясно.** Функционалният символ може да се появява точно един път вдясно, например

$$T(n) = 2T(n - 1) + 1 \quad (5.5)$$

а може да се появява много пъти с различни стойности на аргумента си, например

$$T(n) = T(n - 1) + T(n - 3) + T(n - 5) \quad (5.6)$$

или

$$T(n) = T(n - 1) + T(n - 2) + \dots + T(1) + T(0) \quad (5.7)$$

Забележка: в следното рекурентно уравнение, появите вдясно са една, а не две:

$$T(n) = T(n - 1) + T(n - 1) + 1$$

Тривиално е да се съобрази, че това рекурентно уравнение въщност е (5.5). И така, броят на появите вдясно, за който говорим, е по отношение на **различни** аргументи – както е, например, в (5.6).

Класификацията по брой појви вдясно не е от особена важност, но обикновено рекурентните уравнения с една појва вдясно се решават по-лесно в сравнение с тези с няколко појви. Мастър теоремата, за която ще говорим след малко, е приложима само за рекурентни уравнения с една појва вдясно.

- **Каква е функцията на намаляването** За всяка појва на функционалния символ вдясно, *функцията на намаляването*, неформално казано, е тази математическа операция, която е записана там. Например, в (5.4), в (5.5), в (5.6) и в (5.7), функцията на намаляването е изваждане. Функцията на намаляването може да е делене:

$$T(n) = 2T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + 4T\left(\left\lfloor \frac{n}{7} \right\rfloor\right) + 1$$

или коренуване:

$$T(n) = nT(\lfloor \sqrt{n} \rfloor) + 1$$

или логаритмуване

$$T(n) = T(\lfloor \lg n \rfloor) + 1$$

Възможностите са неограничени. Може да има и “смесен” вид намаляване, например:

$$T(n) = T(n - 2) + T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + 3T(\lfloor \sqrt{n} \rfloor) + 50T(\lfloor \lg n \rfloor) - 10$$

При изследването на сложността на “естествени” алгоритми, смесено намаляване не се среща: функцията на намаляването е или изваждане, или делене. При алгоритмите, изградени по схемата **Разделяй-и-Владей**, сложността се описва с рекурентни уравнения, при които функцията на намаляването е делене. Намаляване чрез изваждане се среща често (макар и не само там) в рекурентните уравнени, описващи сложността на алгоритми, изградени на принципа на пълното изчерпване на възможностите.

Следните определения и класификации са в сила **само за рекурентни уравнения, при които функцията на намаляването е изваждане**.

- ◆ *Ред* (на английски, *order*) на рекурентното уравнение е най-голямото число, което се изважда от n вдясно. В следните три примера, числото, което определя реда на уравнението, е в червено. И така, това рекурентно уравнение

$$T(n) = 2T(n - 1) + 1$$

е от първи ред, това

$$T(n) = 3T(n - 1) + 4T(n - 2) + n$$

е от втори ред, а това

$$T(n) = T(n - 1) + T(n - 2) + \dots + T(n - k) + n^2$$

е от k -ти ред. Прието е рекурентно уравнение от k -ти ред, в което k е константа, да се нарича *рекурентно уравнение с крайна история*. Рекурентно уравнение, в чиято дясната страна функциите имат аргументи от $n - 1$ чак до началната стойност, например

$$T(n) = T(n - 1) + T(n - 2) + \dots + T(1) + T(0)$$

се нарича *рекурентно уравнение с полна история*

Ако допуснем k да не е константа, а някаква нарастваща функция на n , то за всяко рекурентно уравнение може да твърдим, че функцията на намаляването е изваждане. Например, $T(n) = T(\lfloor \frac{n}{2} \rfloor) + 1$ може да се запише като $T(n) = T(n - \lceil \frac{n}{2} \rceil) + 1$; $S(n) = S(\lfloor \sqrt{n} \rfloor) + 1$ може да се запише като $S(n) = S(n - (n - \lfloor \sqrt{n} \rfloor)) + 1$, и така нататък. От това съображение следва, че класифицирането на рекурентните уравнения по функцията на намаляването е—говорейки теоретично—безсмислено, защото, в някакъв смисъл, тя винаги е изваждане. Но на практика това класифициране е доста удобно.

Подчертаваме, че редът на рекурентното уравнение не е същото като броят на появите на функционалния символ вдясно. Например, следното рекурентно уравнение

$$T(n) = T(n - 2) + 2T(n - 5) + 1$$

е от пети ред, мака да има само две појави вдясно[†].

- ◆ *Линейно рекурентно уравнение* е рекурентно уравнение, в което дясната страна е линейна функция на функцията на уравнението. Например, следните три рекурентни уравнения са линейни:

$$A(n) = A(n - 1) + 1$$

$$B(n) = nB(n - 2) + n^2B(n - 4) + \left\lfloor \frac{n \sin n}{\sqrt{n^3 + \lg n}} \right\rfloor$$

$$C(n) = \sum_{k=0}^{n-1} 2^k C(k)$$

[†]Можем да мислим, че $T(n) = 0T(n - 1) + 1T(n - 2) + 0T(n - 3) + 0T(n - 4) + 2T(n - 5) + 1$

докато тези четири **не са** линейни:

$$\begin{aligned} A(n) &= A(n-1)A(n-2) + 1 \\ B(n) &= \left\lceil \frac{1}{1 + B(n-1)} \right\rceil \\ C(n) &= \lfloor \sqrt{C(n-2)} \rfloor \\ D(n) &= \lfloor \lg(D(n-1) + D(n-2)) \rfloor \end{aligned}$$

Във всички примери, появите на съответните функции на уравнението вдясно са оцветени в червено.

- ◆ Различаваме рекурентни уравнения с константни коефициенти, например

$$\begin{aligned} P(n) &= 1P(n-1)P(n-2) + 3P(n-3) \\ S(n) &= \left\lceil \sum_{k=1}^{\infty} \frac{1}{k^2} \right\rceil S(n-2) + n \\ T(n) &= 2T(n-1) + 5T(n-3) \end{aligned}$$

от рекурентни уравнения с променливи коефициенти, например

$$\begin{aligned} P(n) &= \lfloor \lg n \rfloor P(n-1) \\ S(n) &= nS(n-1) \\ T(n) &= \left\lceil \frac{2+n^2}{3+n} \right\rceil T(n-2) \end{aligned}$$

В тези примери, в червено са коефициентите.

- Различаваме *хомогенни* и *нехомогенни* линейни рекурентни уравнения с крайна история и константни коефициенти. Хомогенните са от вида

$$T(n) = c_1 T(n-1) + c_2 T(n-2) + \dots + c_k T(n-k) \quad (5.8)$$

където k и c_1, \dots, c_k са константи. Нехомогенните са от вида

$$T(n) = c_1 T(n-1) + c_2 T(n-2) + \dots + c_k T(n-k) + f(n) \quad (5.9)$$

където k и c_1, \dots, c_k са константи, където $f(n)$ е функция, която зависи от n и не зависи от $T(n)$. Казваме, че $f(n)$ е *нехомогенната част*.

5.2.5 Решаване на рекурентни уравнения

Нека е дадено някакво рекурентно уравнение, в което функционалният символ е T , а името на променливата е n , тоест вляво от знака за равенство има " $T(n)$ ". Както стана ясно, то реализира някаква функция; иначе казано, определя някаква чисрова редица. *Да решим* това рекурентно уравнение означава да намерим друг, еквивалентен израз (формула) за същата функция, който обаче е затворена формула. Например, ако рекурентното уравнение е:

$$\begin{aligned} T(0) &= 0 \\ T(n) &= T(n-1) + n \text{ за } n > 0 \end{aligned} \quad (5.10)$$

то решение е $T(n) = \frac{n(n+1)}{2}$. Какво точно е затворена формула зависи от това, какви операции са допустими. Решението от примера е валидно, ако можем да ползваме събиране с единица, умножение и делене на две. Да разгледаме $T(1) = 1$, $T(n) = nT(n-1)$ за $n > 1$. Както е добре известно, решението е $T(n) = n!$, но това решение е допустимо само ако можем да ползваме факториела; ако са разрешени само събиране, изваждане, умножение, делене, коренуване и степенуване, то уравнението няма решение! При изброените ограничения, $T(n) = \prod_{i=1}^n i$ или $T(n) = n \cdot (n - 1) \cdots 2 \cdot 1$ не са решения, защото не са затворени формули. Забележете, че тези два израза използват нотации, касаещи умножение, но не са затворени формули, ако символът “ \prod ” или многоточието не са разрешени. Нещо повече, ако гледаме на решението като на алгоритъм (който, естествено, не е рекурсивен), то рекурсивният алгоритъм $T(1) = 1$, $T(n) = nT(n-1)$ за $n > 1$ и не-рекурсивният $T(n) = n!$ (който е същият като $T(n) = \prod_{i=1}^n i$) не се различават особено като сложност по време.

Тук ще допускаме, че всички “нормални” функции като изброените, плюс логаритмуването и факториелът, са допустими и че решение, което ги ползва, е валидно. Също така, решението може да съдържа биномни кофициенти, числа на Стирлинг от втори род, суми и произведения. Важното е да няма рекурсия – функционалният символ не трябва да се среща вдъясно.

Ползите от това, да решим дадено рекурентно уравнение, са поне две.

- По правило решението е по-бърз начин за изчисляване[†] на функцията, която съответства на рекурентното уравнение. Например, $\frac{n(n+1)}{2}$ е по-бърз алгоритъм за решаване на функцията, която рекурентното уравнение (5.10) реализира в сравнение със самото (5.10). Ако $n = 1\ 000\ 000\ 000$, директното изпълнение на (5.10) ще се извърши в около 2 000 000 000 стъпки[‡], докато $\frac{1\ 000\ 000\ 000(1\ 000\ 000\ 000+1)}{2}$ се изчислява с една инкрементация, едно умножение и едно делене на две.
- Ако решението е достатъчно просто, много бързо можем да преценим каква е асимптотиката на нарастването на $T(n)$. Например, в (5.10) имаме $T(n) = \Theta(n^2)$, което се вижда веднага от решението $T(n) = \frac{n(n+1)}{2}$. По отношение на числата на Фибоначи, от (5.1) е трудно да се прецени каква е асимптотиката на F_n , докато от добре известното решение

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n \quad (5.11)$$

веднага се вижда, че $F_n \asymp \left(\frac{1+\sqrt{5}}{2} \right)^n$, тоест, горе-долу, $F_n \approx 1.6^n$, понеже $\left(\frac{1-\sqrt{5}}{2} \right)$ е по-малко от единица по абсолютна стойност.

Методите за решаване на рекурентни уравнение ще разискваме в детайли нататък. Тук само ще споменем, че универсален **формален** метод за решаване на рекурентни уравнения **няма**.

5.2.6 Рекурентни уравнения в изследването на алгоритми

Рекурентните уравнения се използват широко в теорията на алгоритмите, защото сложността на даден рекурсивен алгоритъм се описва с подходящо рекурентно уравнение. Правилото

[†]С други думи, по-бърз алгоритъм. Както казахме, самото рекурентно уравнение е алгоритъм, който пресмята число. Решението е алгоритъм, който решава същата задача, но по-бързо.

[‡]Един милиард за движението надолу до достигане на началното условие и още един милиард за движението нагоре, от началното условие до $T(1\ 000\ 000\ 000)$.

за съставяне на подходящо рекурентно уравнение е просто: лявата страна е, да речем, $T(n)$ и изразява работата на алгоритъма върху вход с размер n , а дясната страна има по едно T за всяко рекурсивно викане, като аргументът е функцията, с която намалява входа при даденото рекурсивно викане, плюс израз, който отразява работата на алгоритъма извън рекурсивните викания. Ще видим достатъчно примери за конструиране на рекурентни уравнения, които описват сложността на рекурсивни алгоритми.

При изследването на алгоритмите не се интересуваме от точен израз за сложността – както се каза в минала лекция, това не е нито възможно, нито е необходимо. Интересува ни асимптотиката. Съответно няма да се опитваме да решаваме рекурентните уравнения точно, а само ще търсим асимптотиката на решението. Основно наблюдение, което оставяме без доказателство е, че асимптотиката на решението не зависи от началните условия. Разбира се, има примери за обратното – асимптотиката на

$$T(n) = T(n - 1)^{T(n-2)}$$

зависи от началните условия. Но алгоритмите, които ще разглеждаме в този курс, са такива, че съответните им рекурентни уравнения **не зависят** от началните стойности в смисъл, че асимптотиката е една и съща, каквото и да са началните стойности. Поради това отсега нататък ще изпускаме началните условия от описанията на рекурентните уравнения. Под “рекурентно уравнение” ще разбираме само “същината” – например, $F_n = F_{n-1} + F_{n-2}$ е същината на (5.1) – игнорирайки началните условия.

Да разгледаме отново рекурентното уравнение (5.3), което описва сложността по време на рекурсивен алгоритъм, който разделя входа на две равни части и прави по едно рекурсивно викане върху всяка от тях. Използването на нотациите за точна долна и горна граница е формално коректно, защото, ако n е нечетно, няма как да разделим входа на две **точно равни** части – най-близкото до разделяне на две равни части е разделянето на една част с $\lfloor \frac{n}{2} \rfloor$ елемента и друга част с $\lceil \frac{n}{2} \rceil$ елемента. Но нотациите за точна горна и долна граница правят израза по-труден за четене, без да променят асимптотиката на функцията. И така, в изложението нататък ще правим още една опростяване: в рекурентните уравнения, в които функцията на намаляването не е непременно целочисленна, няма да използваме нотация за точна горна или точна долна граница. Например, вместо (5.3) пишем:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \tag{5.12}$$

И още едно опростяване, което не променя асимптотиката на функцията, което ще покажем с пример. Ако последното рекурентно уравнение описва работата на алгоритъм, който прави две рекурсивни викания и извършва освен това линейна работа, събирамето, които отразява тази линейна работа, би трявало да е “ $\Theta(n)$ ”, така че рекурентното уравнение би трявало да е:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Но асимптотиката на решението не се променя, ако заместим “ $\Theta(n)$ ” с просто “ n ”, тоест най-простиия за записване представител на класа от функции $\Theta(n)$. И така, записът, който ще ползваме, е (5.12).

5.3 Методи за решаване на рекурентни уравнения

5.3.1 Чрез развиване

Средството, което можем да опитаме винаги, за да решим дадено рекурентно уравнение, е да започнем да го развиваме, търсейки някаква закономерност. Например, ако рекурентното уравнение е съвсем простото

$$\begin{aligned} T(0) &= 0 \\ T(n) &= T(n-1) + n \text{ за } n > 0 \end{aligned}$$

можем да разсъждаваме така. Нека n е достатъчно голямо. Тогава $T(n-1) = T((n-1)-1) + (n-1)$, тоест $T(n-1) = T(n-2) + n - 1$. Ако заместим дясната страна в (5.10), получаваме

$$T(n) = T(n-2) + (n-1) + n \quad (5.13)$$

Но за достатъчно големи n , $T(n-2) = T((n-2)-1) + (n-2)$, тоест $T(n-2) = T(n-3) + (n-2)$. Ако сега заместим дясната страна в (5.13), получаваме

$$T(n) = T(n-3) + (n-2) + (n-1) + n \quad (5.14)$$

По правило, ако след 3-4 такива последователни развивания не видим такава закономерност, няма смисъл да опитваме повече. В случая рекурентното уравнение е достатъчно просто, така че закономерност се появява. Очевидно е, че ако продължим по същия начин, след краен брой развивания ще достигнем до

$$T(n) = T(0) + 1 + 2 + \dots + (n-2) + (n-1) + n \quad (5.15)$$

В случая това наистина е очевидно, защото аргументът намалява с единица и рано или късно ще стане нула, а събирамите вдясно, ако се пишат систематично, започват от числото, с единица по-голямо от аргумента на T вдясно и завършват с n , като нарасват с единица.

Сега прилагаме два известни факта. Първо, че $T(0) = 0$, което знаем от началното условие, и второ, че $1 + 2 + \dots + n = \frac{n(n+1)}{2}$. Замествайки в (5.15), веднага получаваме

$$T(n) = \frac{n(n+1)}{2}$$

Методът с развиването (на английски, *unfolding*) не е формален, защото можем да развием началния израз не повече от константен брой пъти и неизбежно използваме интуиция, за да напишем уравнение, в което участват началните условия. Например, при прехода от (5.14) към (5.15) ползваме интуиция. Разбира се, винаги можем да докажем по индукция нашата интуиция (стига да е вярна), но това е друго нещо. Преди да ползваме индукция, трябва да знаем какво да докажем по индукция.

Методът с развиването освен това не е универсално приложим. Опитайте да решите рекурентното уравнение на числата на Фиbonacci (5.1) с развиване. Практически е невъзможно да получите (5.11), ако използвате само здрав разум без никаква теория.

Ето един значително по-сложен пример за прилагане на метода с развиването. Дадено е следното рекурентно уравнение:

$$T(n) = n^2 T\left(\frac{n}{2}\right) + 1$$

Ще го решим с развиване. И така,

$$\begin{aligned}
 T(n) &= n^2 T\left(\frac{n}{2}\right) + 1 \\
 &= n^2 \left(\frac{n^2}{4} T\left(\frac{n}{4}\right) + 1 \right) + 1 \\
 &= \frac{n^4}{2^2} T\left(\frac{n}{2^2}\right) + n^2 + 1 \\
 &= \frac{n^4}{2^2} \left(\frac{n^2}{2^4} T\left(\frac{n}{2^3}\right) + 1 \right) + n^2 + 1 \\
 &= \frac{n^6}{2^6} T\left(\frac{n}{2^3}\right) + \frac{n^4}{2^2} + n^2 + 1 \\
 &= \frac{n^6}{2^6} \left(\frac{n^2}{2^4} T\left(\frac{n}{2^4}\right) + 1 \right) + \frac{n^4}{2^2} + n^2 + 1 \\
 &= \frac{n^8}{2^{12}} T\left(\frac{n}{2^4}\right) + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
 &= \frac{n^8}{2^{12}} \left(\frac{n^2}{2^5} T\left(\frac{n}{2^5}\right) + 1 \right) + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
 &= \frac{n^{10}}{2^{20}} T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{12}} + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
 &= \frac{n^{10}}{2^{20}} T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{12}} + \frac{n^6}{2^6} + \frac{n^4}{2^2} + \frac{n^2}{2^0} + \frac{n^0}{2^0} \\
 &= \frac{n^{10}}{2^{5.4}} T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{4.3}} + \frac{n^6}{2^{3.2}} + \frac{n^4}{2^{2.1}} + \frac{n^2}{2^{1.0}} + \frac{n^0}{2^{0.(-1)}} \\
 &\dots \\
 &= \underbrace{\frac{n^{2i}}{2^{i(i-1)}} T\left(\frac{n}{2^i}\right)}_A + \underbrace{\frac{n^{2(i-1)}}{2^{(i-1)(i-2)}} + \frac{n^{2(i-2)}}{2^{(i-2)(i-3)}} + \dots + \frac{n^8}{2^{4.3}} + \frac{n^6}{2^{3.2}} + \frac{n^4}{2^{2.1}} + \frac{n^2}{2^{1.0}} + \frac{n^0}{2^{0.(-1)}}}_B: \text{сума с } i \text{ събирами}
 \end{aligned}$$

Полученият израз е общийят вид на дясната страна в процеса на развиването; с други думи, дясната страна след $i + 1$ развивания. Максималната стойност на i е $i_{\max} = \lg n^\dagger$. Изразът вдясно е сума от два израза, които означаваме с A и B . Първо ще пресметнем A с $\lg n$ заместо i , като имаме предвид, че $T(1)$ е някаква константа.

$$A = \frac{n^{2 \lg n}}{2^{\lg^2 n - \lg n}} T(1) = \frac{T(1) \cdot 2^{\lg n} n^{2 \lg n}}{2^{\lg^2 n}} = \frac{T(1) \cdot n \cdot n^{2 \lg n}}{2^{\lg^2 n}}$$

[†] i_{\max} е максималният брой пъти, който можем да делим итеративно n на две, после пак на две и така нататък, преди да достигнем някаква предварително зададена фиксирана стойност (която е началното условие). Този брой пъти е от порядъка на $\lg n$. Ето защо: ако началното условие е $T(1) = 1$, то итеративното делене на две може да бъде описано така:

$$n \mapsto \underbrace{\frac{n}{2} \mapsto \frac{n}{4} \mapsto \dots \mapsto 1}_{\text{броят на деленията е } \Theta(\lg n)}$$

Това описание е приблизително, защото не отчита факта, че може да започнат да се получават дробни числа. Ако искаме да сме по-прецизни и да работим само с цели числа, може да опишем итеративното делене на две така:

$$n \mapsto \underbrace{\left\lfloor \frac{n}{2} \right\rfloor \mapsto \left\lfloor \frac{n}{4} \right\rfloor \mapsto \dots \mapsto 1}_{\text{броят на деленията е } \Theta(\lg n)}$$

Ho

$$2^{\lg^2 n} = 2^{\lg n \cdot \lg n} = 2^{\lg(n^{\lg n})} = n^{\lg n} \quad (5.16)$$

Тогава

$$A = \frac{T(1) \cdot n \cdot n^{2 \lg n}}{n^{\lg n}} = \Theta(n^{1 + \lg n}) \quad (5.17)$$

Сега да разгледаме B. Очевидно, можем да го представим като сума по следния начин:

$$\begin{aligned} B &= \sum_{j=1}^{\lg n} \frac{n^{2((\lg n)-j)}}{2^{((\lg n)-j)((\lg n)-j-1)}} \\ &= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{n^{2 \lg n}}{2^{(\lg^2 n - j \lg n - j \lg n + j^2 - \lg n + j)}} \\ &= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{(n^{2 \lg n})(2^{2j \lg n})(2^{\lg n})}{(2^{\lg^2 n})(2^{j^2+j})} \end{aligned} \quad (5.18)$$

Ho

$$2^{2j \lg n} = 2^{\lg(n^{2j})} = n^{2j} \quad (5.19)$$

и

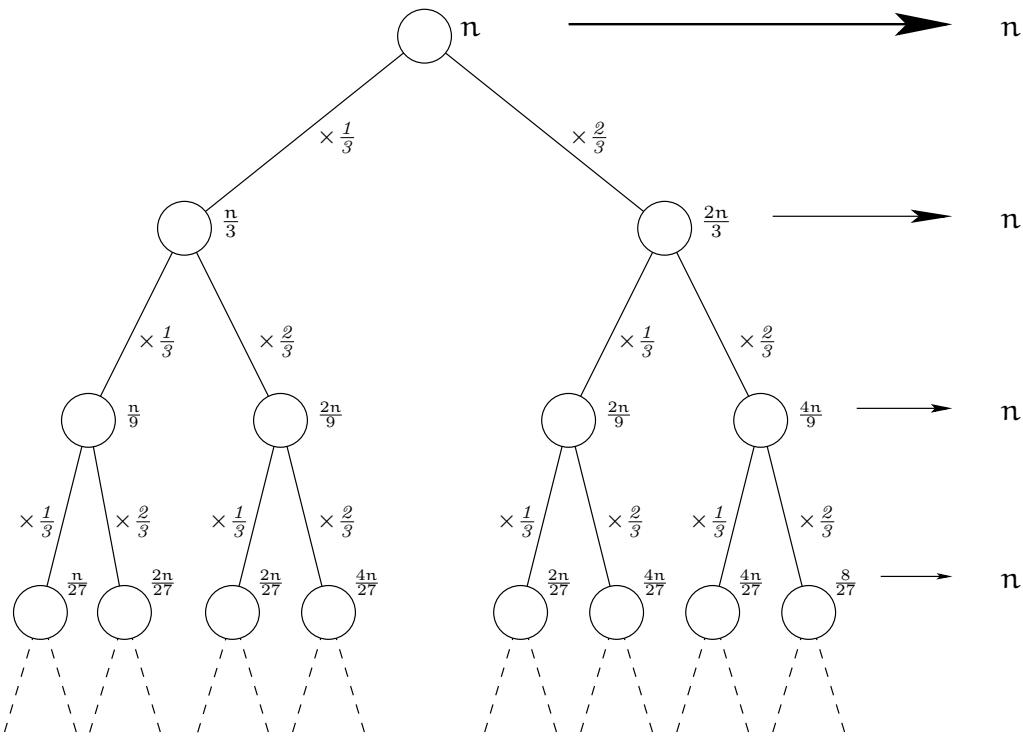
$$2^{\lg n} = n \quad (5.20)$$

Прилагаме (5.16), (5.19) и (5.20) върху (5.18) и получаваме

$$\begin{aligned} B &= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{(n^{2 \lg n})(n^{2j})(n)}{(n^{\lg n})(2^{j^2+j})} \\ &= \sum_{j=1}^{\lg n} \frac{n^{1+\lg n}}{2^{j^2+j}} \\ &= (n^{1+\lg n}) \sum_{j=1}^{\lg n} \frac{1}{2^{j^2+j}} \\ &\leq (n^{1+\lg n}) \sum_{j=1}^{\infty} \frac{1}{2^{j^2+j}} \\ &\leq n^{1+\lg n} \quad \text{тъй като знаем, че } \sum_{j=1}^{\infty} \frac{1}{2^j} = 1 \\ &= \Theta(n^{1+\lg n}) \end{aligned} \quad (5.21)$$

От (5.17) и (5.21) следва, че

$$T(n) = \Theta(n^{1+\lg n})$$



Фигура 5.1: Дървото на рекурсията, отговарящо на $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$. За всяко от изобразените нива, сумата от стойностите на върховете в него е n .

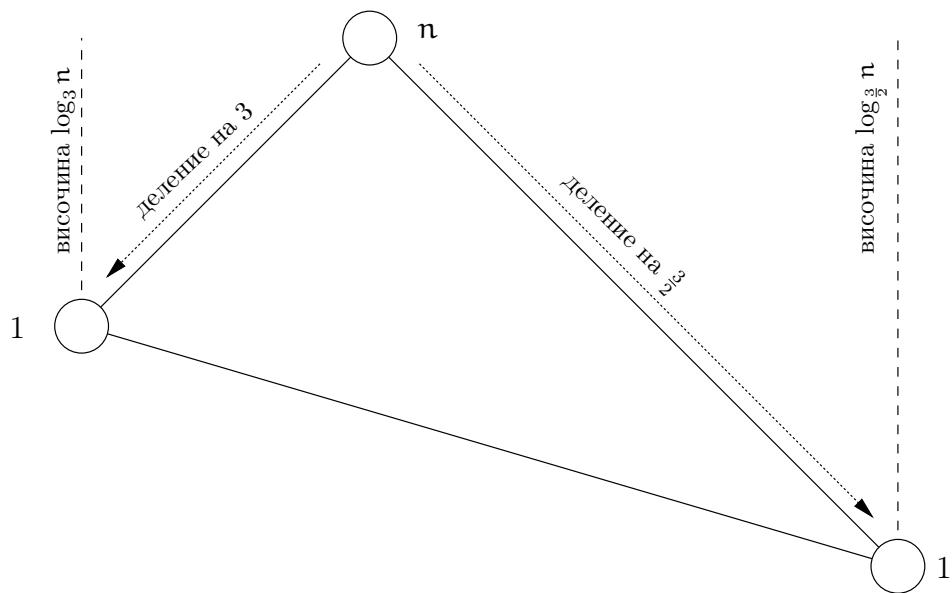
5.3.2 Чрез дървото на рекурсията

При този метод си представяме работата на рекурсивен алгоритъм, чиято сложност по време се описва от рекурентното уравнение, което изследваме. Представяме си дървото на извикванията на рекурсивния алгоритъм, съобразяваме колко работа се извършва във всеки връх от дървото (тоест, във всяко индивидуално изпълнение на рекурсивния алгоритъм), и сумираме тези количества по нива в дървото. След което търсим закономерност в сумите по нива. Виждаме, че метода с дървото е подобен на метода с развиването по това, че не е формален и за да даде резултати, трябва човекът, който го ползва, да прояви достатъчно добра интуиция. Тук ще направим илюстрация с пример. Ще решим рекурентното уравнение

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n \quad (5.22)$$

използвайки метода с дървото на рекурсията. Дървото на рекурсията—по-точно, само “горната” му част—е изобразено на Фигура 5.1. Всеки връх представлява едно индивидуално изпълнение на алгоритъма без рекурсивните викания. Двете ребра към децата му представляват рекурсивните викания, като те (ребрата) са маркирани с множителите, с които входът намалява. До всеки връх е написана работата, която алгоритъмът извършва в това индивидуално изпълнение извън рекурсивните викания. Върховете от едно и също ниво в дървото са нарисувани на едно и също ниво на фигуранта.

Коренът е първоначалното изпълнение. В него алгоритъмът извършва работа n извън двете рекурсивни викания, затова вдясно от корена е написано “ n ”. Едното рекурсивно викане се върши върху вход, който е с големина $\frac{1}{3}$ от големината на началния, а другото върху вход, който е с големина $\frac{2}{3}$ от големината на началния. Затова двете ребра, свързващи корена с децата му, са маркирани с $\frac{1}{3}$ и $\frac{2}{3}$. До всяко от двете деца е написана работата, която се извършва от съответното извикване на алгоритъма. Тази работа е същата като големината на входа за съответното викане, затова записаните стойности са $\frac{n}{3}$ и $\frac{2n}{3}$. И така нататък.



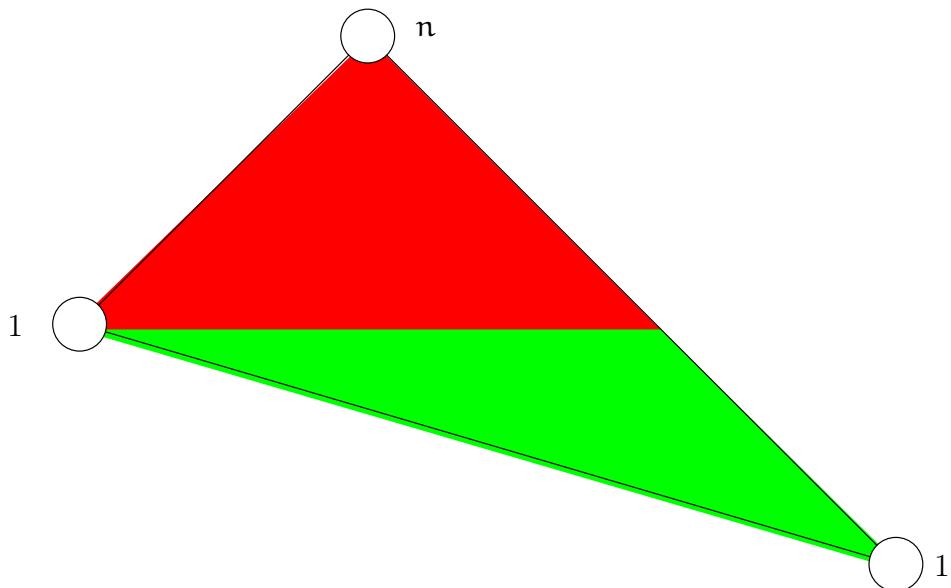
Фигура 5.2: Схема на дървото на рекурсията, отговарящо на $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$. Най-лявото листо е най-близо до корена, а най-дясното листо е най-отдалечено от корена. Височината на най-лявото листо е $\log_3 n$, а на най-дясното е $\log_3 \frac{n}{2}$.

Да мислим за дървото като за подредено дърво. Припомняме, че това означава, че различаваме ляво от дясно дете на всеки вътрешен връх. Стойността, асоциирана с лявото дете на даден връх, е стойността, асоциирана с родителя, умножена по $\frac{1}{3}$. Стойността, асоциирана с дясното дете, е стойността, асоциирана с родителя, умножена по $\frac{2}{3}$. С други думи, наляво има делене на 3, а надясно, делене на $\frac{3}{2}$. Това, че разглеждаме дървото като подредено дърво, има значение единствено за по-лесното решаване на задачата. Листата на дървото са очевидно на различни нива, така че ни трябва някаква систематика в описането на дървото, за да видим закономерност. По начина, по който разглеждаме дървото, най-близкото до корена листо е краят на пътя, който започва от корена и във всеки вътрешен връх продължава наляво, а най-отдалеченото от корена листо е краят на пътя, който започва от корена и във всеки вътрешен връх продължава надясно. Образно казано, дървото има формата, показана на Фигура 5.2. Допускаме, че началната стойност е 1, така че листата са асоциирани с единици. Дървото е нарисувано по начин, който подсказва, че с делене на 3 достигаме 1 по-бързо, отколкото с делене на $\frac{3}{2}$.

Тривиално е да се докаже, че нива с номер от 0 (коренът) до $\log_3 n$ (най-лявото дете) са пълни[†] и във всяко от тях, сумата от стойностите е n . Фигура 5.3 илюстрира пълните нива и останалите: пълните нива са в червено, останалите са в зелено. Да намерим асимптотиката на рекурентното уравнение е същото като да оценим сумата от асоциираните стойности по всички върхове на дървото. За червеното поддърво (Фигура 5.3), тази сума е височината, умножена по n , защото по всяко негово ниво, сумата е n . Височината на червеното поддърво е $\log_3 n$, така че сумата на стойностите по неговите върхове е $n \log_3 n$. Това е добра граница за сумата по всички върхове на дървото. Но очевидно $n \log_3 n$ е горна граница за същата сума, понеже \log_3 е височината на цялото дърво и никое ниво няма сума повече от n . Но $n \log_3 n \asymp n \lg n$ и $n \log_3 n \asymp n \lg n$. Щом $T(n) \leq n \lg n$ и $T(n) \geq n \lg n$, следва, че $T(n) \asymp n \lg n$.

От този пример става ясно, че методът с дървото на рекурсията е подобен на метода с развиwanето, само че тук групираме събираме по различен начин, търсейки закономерност. В последния пример изчислихме сумата от всички “некомогенни части” по всички извиквания.

[†]В смисъл, че в ниво k има 2^k върха.



Фигура 5.3: Друга схема на дървото при $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$. Червеното поддърво е съвършено, понеже нивата му са пълни. Следващите нива, нарисувани в зелено, са непълни.

И по-точно казано, не я изчислихме, а само я оценихме като асимптотика. Под “некомогенни части” имаме предвид събирами като n в (5.22). Такова събирамо отговаря на работата, която алгоритъмът извършва извън рекурсивните викания. Ако наречем такова събирамо, “некомогенната част” по аналогия с (5.9), то, прилагайки метода с дървото на рекурсията, ние въщност сумирахме некомогенните части[†]. Истинската работа, която върши алгоритъмът, е сумата от тези “некомогенни части” и броят на върховете на дървото. Но само сумата от “некомогенните части” дава асимптотиката, така че можем да игнорираме броят на върховете и да не го добавяме към сумата от “некомогенните части”. Ако обаче трябва да решаваме хомогенно рекурентно уравнение като (5.1), трябва да намерим да намерим броя на върховете, или поне неговата асимптотика.

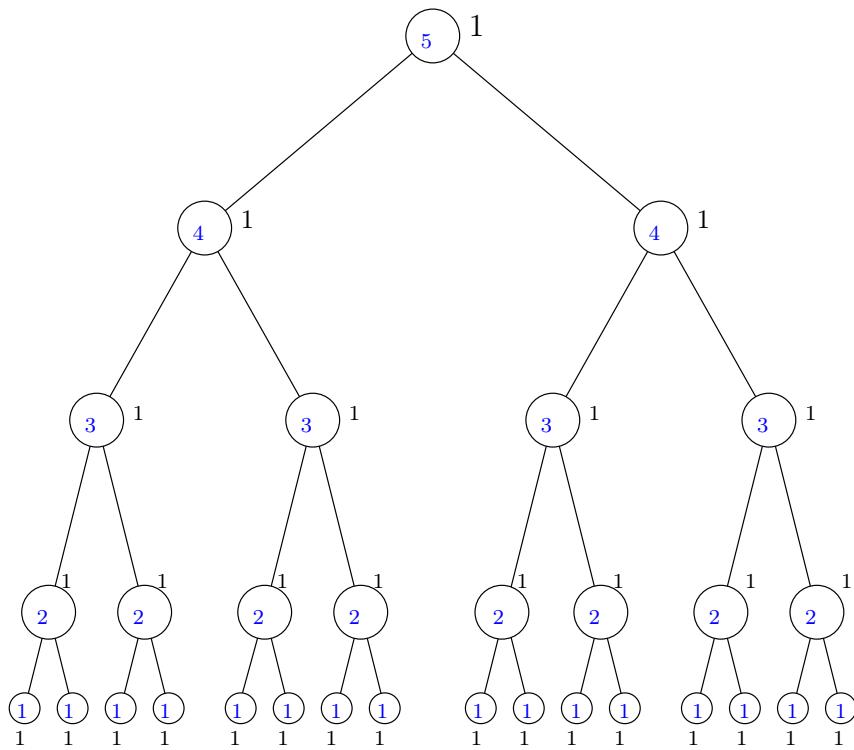
От друга страна, очевидно е, че всяко рекурентно уравнение, което описва сложност на алгоритъм, има **ненулева** некомогенна част, защото алгоритъмът няма как да не върши поне константна работа извън рекурсивните викания. Така че, по отношение на изследването на алгоритми, напълно достатъчно е да сумираме въпросните “некомогенни части”. Подобно на раздаването, този метод също е неформален и приложимостта му зависи от интуицията на човека, който го прилага. Ако човек не открие никаква закономерност в сумите, методът не е приложим.

Сега ще дадем пример за това, колко е важно да открием закономерност в сумите при този метод. Да разгледаме рекурентното уравнение

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n-1) + 1, \quad n \geq 2 \end{aligned} \tag{5.23}$$

и съответното дърво на рекурсията на Фигура 5.4 при $n = 5$. На ниво 0 сумата е 1, на ниво 1 сумата е 2, и така нататък, на ниво 4 сумата е 16. Сумата от стойностите, асоциирани с върховете, е 31. Лесно се вижда, а и лесно се доказва формално, че решението на (5.23) е $T(n) \asymp 2^n$.

[†]По-точно, сумирахме некомогенните части и началните условия, но ние игнорираме началните условия при асимптотичния анализ.



Фигура 5.4: Дървото на рекурсията при $T(1) = 1$, $T(n) = 2T(n-1) + 1$, когато $n = 5$. Във върховете, в син цвят са написани съответните стойности на n .

Сега да преобразуваме (5.23) ето така:

$$T(n) = 2T(n-1) + 1 \leftrightarrow T(n) = T(n-1) + T(n-1) + 1$$

$$\text{Очевидно } T(n-1) = 2T(n-2) + 1$$

$$\text{Тогава } T(n) = T(n-1) + 2T(n-2) + 1 + 1 = T(n-1) + 2T(n-2) + 2$$

Тъй като в случая искаме точно рекурентно уравнение с началните условия, налага се да добавим и начално условия $T(2) = 3$. Рекурентното уравнение става:

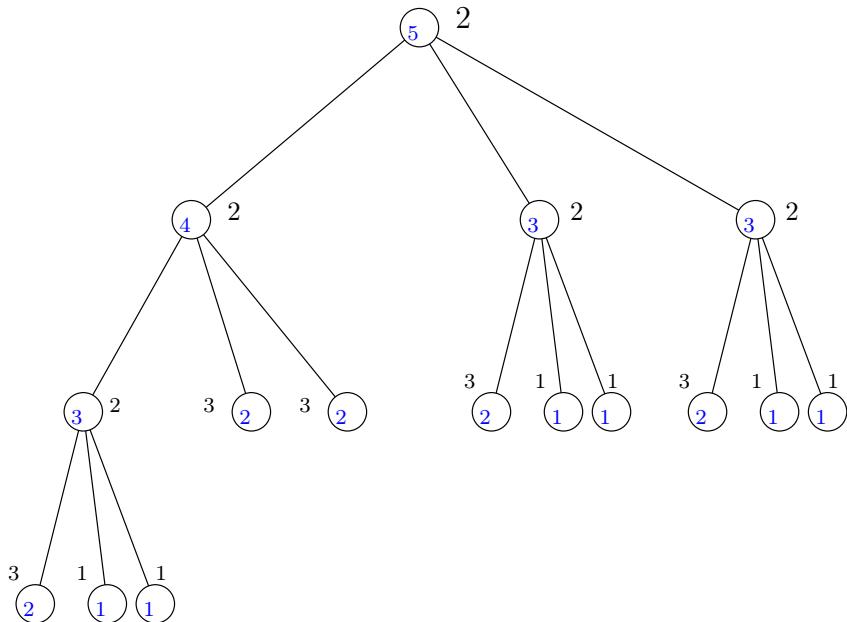
$$\begin{aligned} T(1) &= 1 \\ T(2) &= 3 \\ T(n) &= T(n-1) + 2T(n-2) + 2, \quad n \geq 3 \end{aligned} \tag{5.24}$$

Рекурентни уравнение (5.23) и (5.24) са еквивалентни. Но техните съответни дървета на рекурсията нито са еднакви, нито са изоморфни. Фигура 5.5 показва дървото на рекурсията, съответстващо на (5.24), за $n = 5$. Дървото от Фигура 5.5 е неподходящо за откриване на закономерност както за прецизното решение $T(n) = 2^n - 1$, така и за асимптотичното решение $T(n) \asymp 2^n$.

5.3.3 По индукция

Ще демонстрираме доказването на асимптотиката на нарастването на функцията от дадено рекурентно уравнение с пример. Да разгледаме:

$$T(n) = 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \tag{5.25}$$



Фигура 5.5: Дървото на рекурсията на $T(1) = 1$, $T(2) = 3$, $T(n) = T(n - 1) + 2T(n - 2) + 2$. Във върховете, в син цвят са написани съответните стойности на n .

Ще докажем, че $T(n) = \Theta(n \lg n)$ по индукция по n . За да постигнем това, ще докажем поотделно, че $T(n) = O(n \lg n)$ и $T(n) = \Omega(n \lg n)$. В тези доказателство база няма, тъй като игнорираме началните условия.

Част i: Доказателство, че $T(n) = O(n \lg n)$. По дефиниция, това е същото като да докажем, че съществуват положителна константа c и положително n_0 , такова че за всяко $n \geq n_0$:

$$T(n) \leq cn \lg n \quad (5.26)$$

Индуктивното предположение е, че

$$T\left(\left\lceil \frac{n}{2} \right\rceil\right) \leq c \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil \quad (5.27)$$

От (5.25) и (5.27):

$$\begin{aligned} T(n) &\leq 2 \cdot c \cdot \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + n \\ &\leq 2 \cdot c \cdot \left(\frac{n}{2} + 1\right) \lg \left\lceil \frac{n}{2} \right\rceil + n \end{aligned} \quad (5.28)$$

$$\leq 2 \cdot c \cdot \left(\frac{n}{2} + 1\right) \lg \left(\frac{3n}{4}\right) + n \quad \text{зашпото } \forall n \geq 2 \left(\frac{3n}{4} \geq \left\lceil \frac{n}{2} \right\rceil\right) \quad (5.29)$$

$$\begin{aligned} &= c(n + 2)(\lg n + \lg 3 - 2) + n \\ &= cn \lg n + cn(\lg 3 - 2) + 2c \lg n + 2c(\lg 3 - 2) + n \\ &\leq cn \lg n \quad \text{ако } cn(\lg 3 - 2) + 2c \lg n + 2c(\lg 3 - 2) + n \leq 0 \end{aligned}$$

Да разгледаме

$$cn(\lg 3 - 2) + 2c \lg n + 2c(\lg 3 - 2) + n = (c(\lg 3 - 2) + 1)n + 2c \lg n + 2c(\lg 3 - 2)$$

Асимптотиката на нарастването се определя от линейното събирамо. Ако константата $c(\lg 3 - 2) + 1$ е отрицателна, тогава целият израз е положителен за всички достатъчно големи стойности на n . С други думи, за по-кратко решение, няма да определяме n_0 , бидейки убедени,

че такава начална стойност има. За да бъде изпълнено $c(\lg 3 - 2) + 1 < 0$, трябва да е вярно, че $c > \frac{1}{2-\lg 3}$. Следователно, всяко c , такова че $c > \frac{1}{2-\lg 3}$, “върши работа” за нашето доказателство.

ВНИМАНИЕ! В (5.28) ние заместваме $\lceil \frac{n}{2} \rceil$ с $\frac{3n}{4}$. Бихме могли да използваме всяка друга дроб $\frac{pn}{q}$, стига да е изпълнено $\frac{1}{2} < \frac{p}{q} < 1$. Защо трябва да е изпълнено $\frac{1}{2} < \frac{p}{q}$? Ако това не е изпълнено, нямаме право да твърдим, че неравенството “ \leq ” между (5.28) и (5.29) е в сила. А защо трябва да е изпълнено $\frac{p}{q} < 1$? Да допуснем, че $\frac{p}{q} = 1$; с други думи, заместваме $\lceil \frac{n}{2} \rceil$ с n . Тогава (5.29) става:

$$c(n+2)(\lg n) + n = cn \lg n + 2c \lg n + n$$

Очевидно, това е по-голямо от $cn \lg n$ за всички достатъчно големи n , така че нямаме доказателство.

Част ii: Доказателство, че $T(n) = \Omega(n \lg n)$. По дефиниция, това е същото като да докажем, че съществуват положителна константа d и положително n_1 , такова че за всяко $n \geq n_1$:

$$T(n) \geq dn \lg n \quad (5.30)$$

Индуктивното предположение е, че

$$T\left(\left\lceil \frac{n}{2} \right\rceil\right) \geq d \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil \quad (5.31)$$

От (5.25) и (5.31):

$$\begin{aligned} T(n) &\geq 2 \cdot d \cdot \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + n \\ &\geq 2 \cdot d \cdot \left(\frac{n}{2} \right) \lg \left(\frac{n}{2} \right) + n \\ &= dn(\lg n - 1) + n \\ &= dn \lg n + (1 - d)n \\ &\geq dn \lg n \end{aligned} \quad \text{при условие, че } (1 - d)n \geq 0 \quad (5.32)$$

Очевидно всяко d , такова че $0 < d \leq 1$, “върши работа” за нашето доказателство.

С това доказателството е приключено. Преди да преминем към следващия метод, две предупреждения за потенциални грешки или проблеми при използването на индукция за доказване на асимптотиката на рекурентни уравнения.

Доказателството трябва да се извърши спрямо константата, с която е започнало. Да разгледаме отново следното рекурентно уравнение:

$$T(n) = 2T(n-1) + 1$$

Както вече знаем, решението е $T(n) = \Theta(2^n)$. Сега помислете, къде е грешката в следното доказателство, което “доказва”, че $T(n) = O(n)$. Ще “докажем”, че $T(n) \leq cn$ за някаква константа c . Индуктивното предположение е, че $T(n-1) \leq c(n-1)$. Тогава $T(n) \leq 2c(n-1) + 1$. Тогава $T(n) \leq 2cn - 2c + 1$. Но последният израз казва, че $T(n)$ е ограничена отгоре от константа, умножена по n . Тогава $T(n) = O(n)$, което и искахме да докажем.

Къде е грешката? Грешката е в това, че доказателството не се извършва по отношение на константата, с която е започнато. Твърдението, което доказваме е, че **съществува положителна константа c и стойност на аргумента n_0 , такива че за всяко $n \geq n_0$, $T(n) \leq cn$** . Естествено, това твърдение е лъжа, така че няма как да го докажем, но важното е, че това е твърдението, което доказваме. “ $T(n) = O(n)$ ” е просто кратък запис за него. Щом избраният символ за константата е c , доказателството трябва да бъде извършено по отношение на него. $2c$ е друга константа. Наистина, $2c$ е константа, но за да имаме доказателство, трябва да го направим по отношение на c .

Понякога се налага да се засили индукционното предположение. Ще докажем по индукция, че решението на

$$T(n) = 2T(n - 1) + n \quad (5.33)$$

е $T(n) = \Theta(2^n)$.

Първо ще докажем, че $T(n) = O(2^n)$. Нарочно започваме доказателството с няколко неуспешни опита.

Първи опит. Допускаме, че съществува константа c , такава че за всички достатъчно големи n е вярно, че

$$T(n) \leq c2^n$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leq 2c2^{n-1} + n \\ &= c2^n + n \\ &\leq c2^n \text{ за никоя положителна } c \end{aligned}$$

Доказателството се провали. Това не означава, че твърдението, което доказваме, непременно е грешно. Провал на доказателство **може** да означава, че твърдението не е вярно, но **може** и да означава, че твърдението е вярно, но не може да се докаже по този начин. Ще се опитаме да докажем по-силно твърдение.

Втори опит. Допускаме, че съществуват положителни константи b и c , такива че за всички достатъчно големи n е вярно, че

$$T(n) \leq c2^n - b$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - b) + n \\ &= c2^n - 2b + n \\ &\leq c2^n - b \text{ за никои положителни } b, c \end{aligned}$$

Ще засилим още твърдението.

Трети опит. Допускаме, че съществуват положителни константи b и c , такива че за всички достатъчно големи n е вярно, че

$$T(n) \leq c2^{n-b}$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leq 2(c2^{n-b-1}) + n \\ &= c2^{n-b} + n \\ &\leq c2^{n-b} \text{ за никои положителни } b, c \end{aligned}$$

Ще засилим още твърдението.

Четвърти опит. Допускаме, че съществува положителна константа c , такава че за всички достатъчно големи n е вярно, че

$$T(n) \leq c2^n - n$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - (n-1)) + n \\ &= c2^n - n + 2 \\ &\leq c2^n - n \text{ за никоя положителна } c \end{aligned}$$

Пети опит. Допускаме, че съществуват положителни константи b и c , такива че за всички достатъчно големи n е вярно, че

$$T(n) \leq c2^n - bn$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - b(n-1)) + n \\ &= c2^n - 2bn + 2b + n \\ &= c2^n - bn + (1-b)n + 2b \\ &\leq c2^n - bn \text{ за всеки избор на } c > 0 \text{ и } b > 1 \end{aligned}$$

Успех! Чак сега успяхме да формулираме индукционно предположение, което е доказуемо.

Сега ще докажем, че $T(n) = \Omega(n)$. С други думи, че съществува положителна константа d , такава че за всички достатъчно големи n е изпълнено

$$T(n) \geq d2^n$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\geq 2(d2^{n-1}) + n \\ &= d2^n + n \\ &\geq d2^n \end{aligned}$$

Успех! Видяхме, че в този случай—а това е вярно по принцип при доказателства по индукция на решения на рекурентни уравнения—засилване на индукционното предположение се иска само за едното от O и Ω доказателствата.

5.3.4 Чрез Мастър Теоремата (МТ)

Този теоретичен резултат е от 1978 г. [9]. Тук ще направим изложението по учебника на Cormen и др. [15, стр. 61].

Гръмкото име на теоремата се дължи на факта, че тя се явява обобщение на множество теореми, даващи решения на рекурентни уравнения, в които функцията на намаляване е делене. Разбира се, няма най-обща теорема, която да дава универсален метод за решаване на всякакви рекурентни уравнение. По отношение на линейните рекурентни уравнения с константни коефициенти, при които намаляването е чрез делене с константа (каквите уравнения третира Теорема 18) е известна значително по-мощна и обща теорема, известна като Теорема на Akra-Bazzi [3]. Няма да разглеждаме и няма да ползваме в тези лекции (вж. [43] за подробно обяснение), просто подчертаваме, че въпреки името си, тази теорема е просто едно средство и са известни много по-мощни средства от него.

Теорема 18: Мастър теорема, накратко МТ, [15], стр. 62

Нека $a \geq 1$ и $b > 1$ са константи и нека $f(n)$ е положителна функция. Нека

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad (5.34)$$

където $\frac{n}{b}$ има смисъл или на $\left\lfloor \frac{n}{b} \right\rfloor$, или на $\left\lceil \frac{n}{b} \right\rceil$. Тогава асимптотиката на $T(n)$ е следната:

Случай 1 Ако $f(n) = O(n^{\log_b a}/n^\epsilon)$ за някоя положителна константа ϵ , тогава $T(n) = \Theta(n^{\log_b a})$.

Случай 2 Ако $f(n) = \Theta(n^{\log_b a})$, тогава $T(n) = \Theta(n^{\log_b a} \cdot \lg n)$. С други думи, $T(n) = \Theta(f(n) \cdot \lg n)$.

Случай 3 Ако са изпълнени следните условия:

1. $f(n) = \Omega(n^{\log_b a} \cdot n^\epsilon)$ за някоя положителна константа ϵ , и
2. $a.f\left(\frac{n}{b}\right) \leq c.f(n)$ за някоя положителна константа c , такава че $0 < c < 1$ за всички достатъчно големи n ,

то $T(n) = \Theta(f(n))$.

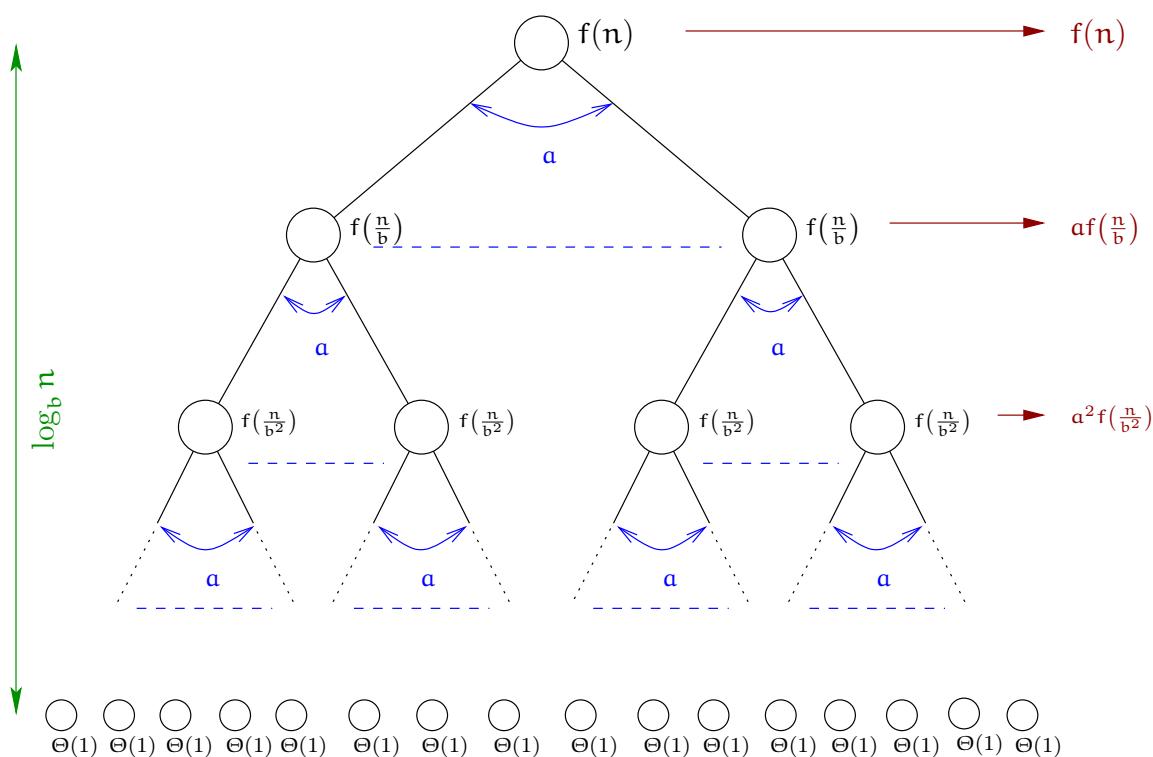
□

Условие 3.2 се нарича *условие за регулярност*.

Няма да даваме доказателство на теоремата. Читателят може да го види в [15]. Тук само ще дадем няколко пояснения.

Пояснение 1 И в трите случая на МТ сравняваме броя на листата на дървото на рекурсията с функцията $f(n)$. Да се убедим, че броят на листата на дървото наистина е $n^{\log_b a}$. Нека да мислим, че a и b са цели числа[†]. Ако a и b са цели положителни числа, като $b \geq 2$, то (5.34) отговаря на рекурсивен алгоритъм, направен по схемата **Разделяй-и-Владей**, който прави a рекурсивни извиквания, всяко върху вход с големина $\frac{n}{b}$, и освен това извършва $f(n)$ работа (като време) по разделянето на входа и комбинирането на резултатите от извикванията. Дървото на този алгоритъм е показано на Фигура 5.6. Всеки връх на дървото има

[†]В [15] е доказано, че същите съображения остават в сила дори когато a и b не са цели.



Фигура 5.6: Дървото на рекурсията на алгоритъм, чиято сложност по време се описва от $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, където a и b са цели положителни числа.

разклоненост a , и входът намалява с делене на b . Сумите по нивата на дървото на нехомогенните части са $f(n)$, $af\left(\frac{n}{b}\right)$, $a^2f\left(\frac{n}{b^2}\right)$, и така нататък. Височината на дървото е $\log_b n$. Броят на върховете от ниво[†] k очевидно е a^k , така че броят на листата е $a^{\log_b n}$. Лесно се вижда, че $a^{\log_b n} = n^{\log_b a}$.

Броят на листата е пропорционален на големината на дървото, освен ако a не е 1, така че големината на дървото е $a^{\log_b n}$; при $a = 1$ има точно едно листо, а височината продължава да е $\log_b n$. Така че при $a > 1$, МТ сравнява големината на дървото с нехомогенната част $f(n)$.

В първия случай на МТ големината на дървото доминира над $f(n)$ и тя (големината) задава асимптотиката на $T(n)$, във втория случай големината на дървото и $f(n)$ са асимптотично еквивалентни, в третия случай $f(n)$ доминира над големината на дървото и тя ($f(n)$) определя асимптотиката, ако е изпълнено и условието за регулярност[‡].

Пояснение 2 Трите случаи на МТ не са разбиване на всички възможни случаи. Както вече видяхме в Теорема 5(2.30), има функции, които не са асимптотично сравними, така че ако $n^{\log_b a}$ и $f(n)$ не са асимптотично сравними, то МТ не е приложима.

Но дори когато $n^{\log_b a}$ и $f(n)$ са асимптотично сравними, има примери за a , b и $f(n)$, в които $f(n)$ расте прекалено бързо, за да попаднем в първия случай, но прекалено бавно, за да попаднем във втория, и МТ пак не е приложима. Неформално казано, между първия и втория случай на МТ има “празнина” и ако конкретната задача е в някоя от тези “празнини”, не можем да използваме МТ.

[†]Припомняме си, че върховете от ниво k са точно върховете на разстояние k от корена.

[‡]Сравняването на големината на дървото с функцията-некомогенна част има смисъл и при други видове рекурентни уравнения. Например, рекурентното уравнение $S(n) = 2S(n - 1) + n!$ има решение $S(n) \asymp n!$, защото факториелът доминира над големината на дървото, която е само експоненциална и “бледнее” пред $n!$. В никакъв смисъл, това е аналог на третия случай на МТ.

Например, нека $a = b = 2$ и $f(n) = \frac{n}{\lg n}$. Тогава $f(n) \neq O(n^{\log_b a} / n^\epsilon)$ за всяка положителна константа ϵ , така че рекурентното уравнение не попада в първия случай. От друга страна, $f(n) \neq \Theta(n^{\log_b a})$, така че не сме и във втория случай. Бидейки попаднали в “празнината” между първия и втория случай, налага се да ползваме други методи, за да решим задачата.

Допълнение 18: Решение на $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$

Ще решим $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$. Първо да опитаме с Теорема 18. Използвайки терминологията на Теорема 18, $a = b = 2$, следователно $\log_b a$ е $\log_2 2 = 1$, следователно $n^{\log_b a}$ е $n^1 = n$. Функцията $f(n)$ е $\frac{n}{\lg n}$. Да видим дали може да класифицираме рекурентното уравнение в един от трите случая.

Случай 1 Дали $\frac{n}{\lg n} = O(n^1/n^\epsilon)$ за някоя константа $\epsilon > 0$? Не, защото $\forall \epsilon > 0 : n^\epsilon \neq O(\lg n)$. Детайлно доказателство на този факт има в сборника с решени задачи по алгоритми.

Случай 2 Дали $\frac{n}{\lg n} = \Theta(n^1)$? Не, защото $\frac{n}{\lg n} = o(n^1)$.

Случай 3 Дали $\frac{n}{\lg n} = \Omega(n^1 \cdot n^\epsilon)$ за някоя константа $\epsilon > 0$? Не, и то по същата причина: защото $\frac{n}{\lg n} = o(n^1)$.

Тогава тази задача не може да бъде решена с Теорема 18. Нещо повече: не можем да ползваме и Теорема 19 на стр. 167, защото $\forall t > 0 : \frac{n}{\lg n} \neq \Theta(n^{\log_2 2} \times \lg^t(n))$.

Ще решим задачата с развиване:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n} \\ &= 2\left(2T\left(\frac{n}{4}\right) + \frac{\frac{n}{2}}{\lg \frac{n}{2}}\right) + \frac{n}{\lg n} \\ &= 4T\left(\frac{n}{4}\right) + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\ &= 4\left(2T\left(\frac{n}{8}\right) + \frac{\frac{n}{4}}{\lg \frac{n}{4}}\right) + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\ &= 8T\left(\frac{n}{8}\right) + \frac{n}{(\lg n) - 2} + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\ &\dots \\ &= nT(1) + \frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \dots + \frac{n}{(\lg n) - 2} + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\ &= nT(1) + n \underbrace{\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{(\lg n) - 2} + \frac{1}{(\lg n) - 1} + \frac{1}{\lg n}\right)}_B \end{aligned}$$

Забелязваме, че $B = n \times H_{\lg n}$, защото в скобите се намира $(\lg n)^{\text{тата}}$ парциална сума на хармоничния ред. Добре известно е, че $H_m \asymp \lg m$, следователно $H_{\lg n} = \Theta(\lg \lg n)$, и оттам $B = \Theta(n \lg \lg n)$. От това следва, че $T(n) = \Theta(n \lg \lg n)$.

Аналогично, има примери за a, b и $f(n)$, в които $f(n)$ расте прекалено бързо, за да попаднем във втория случай, но прекалено бавно, за да попаднем в третия. С други думи, между втория и третия случай също има “празнина”, за която МТ не е приложима.

Пояснение 3 Условието $f(n) = O(n^{\log_b a} / n^\epsilon)$ в първия случай на МТ е по-силно от $f(n) = o(n^{\log_b a})$, а условието $f(n) = \Omega(n^{\log_b a} \cdot n^\epsilon)$ в третия случай е по-силно от $f(n) = \omega(n^{\log_b a})$. За удобство да направим следните дефиниции.

Определение 26: асимптотично строго по-бавно/по-бързо и полиномиално по-бавно/по-бързо

Ако $\phi(n) = o(\psi(n))$, казваме, че $\phi(n)$ *расте асимптотично строго по-бавно от $\psi(n)$* .

Ако $\phi(n) = O(\psi(n)/n^\epsilon)$ за някаква положителна константа ϵ , казваме, че $\phi(n)$ *расте полиномиално по-бавно от $\phi(n)$* .

Ако $\phi(n) = \omega(\psi(n))$, казваме, че $\phi(n)$ *расте асимптотично строго по-бързо от $\psi(n)$* .

Ако $\phi(n) = \Omega(\psi(n) \cdot n^\epsilon)$ за някаква положителна константа ϵ , казваме, че $\phi(n)$ *расте полиномиално по-бързо от $\phi(n)$* .

$f(n) = O(n^{\log_b a} / n^\epsilon)$, където ϵ е положителна константа, казва, че между $f(n)$ и $n^{\log_b a}$ може да бъде “вкарано” n^ϵ , като $f(n)$ е отдолу; с други думи, че $f(n)$ “изостава” от $n^{\log_b a}$ на една полиномиално растяща функция. Това е значително по-силно от $f(n)$ да “изостава” от $n^{\log_b a}$ на повече от константа, което е $f(n) = o(n^{\log_b a})$. Аналогично, $f(n) = \Omega(n^{\log_b a} \cdot n^\epsilon)$ казва, че между $f(n)$ и $n^{\log_b a}$ може да бъде “вкарано” n^ϵ , като $f(n)$ е отгоре; с други думи, че $f(n)$ “изпреварва” $n^{\log_b a}$ с една полиномиално растяща функция. Това е значително по-силно от $f(n)$ да “изпреварва” $n^{\log_b a}$ на повече от константа, което е $f(n) = \omega(n^{\log_b a})$.

Наблюдение 15

Ако $\phi(n)$ расте полиномиално по-бавно от $\psi(n)$, то $\phi(n)$ расте асимптотично строго по-бавно от $\psi(n)$. Конверсното не е вярно. Следователно, условието за полиномиално по-бавно нарастване е по-силно от условието за асимптотично строго по-бавно нарастване. Ако $\phi(n)$ расте полиномиално по-бързо от $\psi(n)$, то $\phi(n)$ расте асимптотично строго по-бързо от $\psi(n)$. Конверсното не е вярно. Следователно, условието за полиномиално по-бързо нарастване е по-силно от условието за асимптотично строго по-бързо нарастване.

Наблюдение 15 ни дава право да кажем следното.

$$\begin{aligned} f(n) = O(n^k / n^\epsilon) &\rightarrow f(n) = o(n^k) \\ f(n) = o(n^k) &\rightarrow f(n) = O(n^k / n^\epsilon) \\ f(n) = \Omega(n^k \cdot n^\epsilon) &\rightarrow f(n) = \omega(n^k) \\ f(n) = \omega(n^k) &\rightarrow f(n) = \Omega(n^k \cdot n^\epsilon) \end{aligned}$$

Наблюдение 16

Би било грешка да се опитаме да формулираме първия случай на МТ чрез $f(n) = o(n^{\log_b a})$ или третия случай чрез $f(n) = \omega(n^{\log_b a})$.

Пояснение 4 Условието за регулярност в третия случай е по-силно от условието $f(n) = \Omega(n^{\log_b a + \epsilon})$ в смисъл, че

$$\exists c, 0 < c < 1 : \left(a.f\left(\frac{n}{b}\right) \leq c.f(n) \right) \rightarrow \exists \epsilon (f(n) = \Omega(n^{\log_b a + \epsilon})) \quad (5.35)$$

$$\exists \epsilon (f(n) = \Omega(n^{\log_b a + \epsilon})) \Rightarrow \exists c, 0 < c < 1 : \left(a.f\left(\frac{n}{b}\right) \leq c.f(n) \right) \quad (5.36)$$

Пример за $f(n)$ като в (5.36) статията в Wikipedia за МТ: да вземем $f(n) = n(2 - \cos n)$. Доказателство на (5.35) се съдържа в Допълнение 19.

Допълнение 19: Условието за регулярност в МТ влече Случай 3

Задача 4.4-3 в [15] е: покажете, че Случай 3 на МТ съдържа излишък в смисъл, че условието за регулярност $a.f\left(\frac{n}{b}\right) \leq c.f(n)$ за някаква положителна константа $c < 1$ влече съществуването на константа $\epsilon > 0$, такава че $f(n) = \Omega(n^{\log_b a + \epsilon})$. Сега ще покажем точно това. Нека съществува константа c , такава че $0 < c < 1$ и:

$$f(n) \geq \frac{a}{c} f\left(\frac{n}{b}\right)$$

Тъй като $a \geq 1$ и $c < 1$, то $\frac{a}{c} > 1$. За краткост, ще означаваме $\frac{a}{c}$ с буквата s . Тогава в сила е:

$$f(n) \geq s f\left(\frac{n}{b}\right)$$

Тогава

$$f(n) \geq s^2 f\left(\frac{n}{b^2}\right)$$

$$f(n) \geq s^3 f\left(\frac{n}{b^3}\right)$$

...

$$f(n) \geq s^t f\left(\frac{n}{b^t}\right) \quad (5.37)$$

Съществува някаква стойност n_0 на променливата n , за която този процес ще спре. Тогава $\frac{n}{b^t} = n_0$, следователно $t = \log_b\left(\frac{n}{n_0}\right) = \log_b n - \log_b n_0$. Заместваме t и $\frac{n}{b^t}$ и с в (5.37) и получаваме:

$$f(n) \geq s^{\log_b n - \log_b n_0} \times f(n_0) = \frac{s^{\log_b n}}{s^{\log_b n_0}} \times f(n_0)$$

Забележете, че $\frac{1}{s^{\log_b n_0}} \times f(n_0)$ е константа. За краткост, да наречем тази константа β . Тогава:

$$f(n) \geq s^{\log_b n} \times \beta$$

Да се върнем към началните имена a и c , като полагаме $z = \frac{1}{c}$, за да е изпълнено $z > 1$. Имаме $s = a \times z$.

$$f(n) \geq a^{\log_b n} \times z^{\log_b n} \times \beta$$

Имайки предвид, че $a^{\log_b n} = n^{\log_b a}$ and $z^{\log_b n} = n^{\log_b z}$, извеждаме:

$$f(n) \geq \beta \times n^{\log_b a} \times n^{\log_b z}$$

Тъй като $z > 1$, то $\log_b z > 0$. Нека $\epsilon = \log_b z$. Извеждаме желания резултат: за всички достатъчно големи n и някаква константа β :

$$f(n) \geq \beta n^{\log_b a + \epsilon} \Rightarrow f(n) = \Omega(n^{\log_b a + \epsilon})$$

□

Следователно, третият случай на МТ би могъл да бъде формулиран само с условието за регуляреност. Условието $f(n) = \Omega(n^{\log_b a + \epsilon})$ е излишно. Дадената формулировка е избрана, без съмнение, за да има външна аналогия с формулировките на първите два случая. Това е и краят на поясненията върху МТ.

Ще дадем няколко примера за прилагането на МТ.

Пример 1. Ще решим

$$T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$$

Използвайки терминологията на МТ, a е 3, b е 4, така че $\log_b a$ е $\log_4 3$, което е приблизително 0.79, и $f(n)$ е $n \lg n$. Със сигурност е вярно, че $n \lg n = \Omega(n^{\log_4 3 + \epsilon})$ за някое $\epsilon > 0$, например $\epsilon = 0.1$. Но трябва да проверим и дали условието за регуляреност е в сила, за да видим дали може да приложим третия случай на МТ. Условието за регуляреност е:

$$\exists c \text{ такова че } 0 < c < 1 \text{ и } 3 \frac{n}{4} \lg \frac{n}{4} \leq cn \lg n \text{ за всички достатъчно големи } n$$

Очевидно това е вярно за, да речем, $c = \frac{3}{4}$, така че третият случай на МТ е приложим. Съгласно него, $T(n) = \Theta(n \lg n)$.

Пример 2. Ще решим

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

Преписваме условието така:

$$T(n) = 1 \cdot T\left(\frac{n}{\frac{3}{2}}\right) + 1$$

за да е по-ясно какви са a и b . Виджаме, че a е 1, b е $\frac{3}{2}$, така че $\log_b a$ е $\log_{\frac{3}{2}} 1 = 0$, следователно $n^{\log_b a}$ е $n^0 = 1$. Нехомогенната част $f(n)$ е 1. Очевидно, $1 = \Theta(n^0)$, така че вторият случай на МТ е приложим. Съгласно него, $T(n) = \Theta(1 \cdot \lg n) = \Theta(\lg n)$.

Пример 3. Ще решим

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Тук $a = 4$, $b = 2$, така че $\log_b a$ е $\log_2 4 = 2$, следователно $n^{\log_b a}$ е n^2 . Нехомогенната част $f(n)$ е n . Сравняваме по асимптотично нарастване n с n^2 . Очевидно, $n = O(n^2/n^\epsilon)$ за някое $\epsilon > 0$, така че първият случай на МТ е приложим. Съгласно него, $T(n) = n^2$.

Допълнение 20: Едно разширение на Теорема 18

Теорема 19: Разширение на МТ

При допусканията на Теорема 18, нека $\log_b a = k$ и

$$f(n) = \Theta(n^k \lg^t n) \quad (5.38)$$

за някое $t \in \mathbb{N}^+$. Тогава

$$T(n) = \Theta(n^k \lg^{t+1} n)$$

Доказателство:

Теорема 18 не е директно приложима, защото рекурентното уравнение с тази функция $f(n)$ при $t \geq 1$ не може да бъде класифицирано в нито един от трите случая. Решаваме задачата с развиване. За простота допускаме, че n е точна степен на b , тоест $n = b^m$ за някакво цяло число $m > 0$. За читателя остава да обобщи доказателството за всяко положително n .

Да допуснем, че логаритъмът в (5.38) е с основа b . Забелязваме, че всичко в Θ -нотацията в дясната страна на (5.38) може да се запише така:

$$n^k \log_b^n = n^{\log_b a} (\log_b b^m)^t = b^{(m \log_b a)} m^t = b^{(\log_b a^m)} m^t = a^m m^t \quad (5.39)$$

Тогава (5.38) е еквивалентно на

$$c_1 a^m m^t \leq f(b^m) \leq c_2 a^m m^t$$

за някакви положителни константи c_1 и c_2 и всички достатъчно големи стойности на m . За простота, ще допуснем, че

$$f(b^m) = a^m m^t \quad (5.40)$$

Не е трудно да се направи доказателството да бъде направено по същия начин и без това опростяване.

От условието на МТ имаме $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Използвайки (5.40), записваме:

$$\begin{aligned} T(b^m) &= aT\left(\frac{b^m}{b}\right) + a^m m^t \\ &= aT(b^{m-1}) + a^m m^t \leftrightarrow \\ S(m) &= aS(m-1) + a^m m^t \quad \text{замествайки } T(b^m) \text{ със } S(m) \\ &= a(aS(m-2) + a^{m-1}(m-1)^t) + a^m m^t \\ &= a^2S(m-2) + a^m(m-1)^t + a^m m^t \\ &= a^2(aS(m-3) + a^{m-2}(m-2)^t) + a^m(m-1)^t + a^m m^t \\ &= a^3S(m-3) + a^m(m-2)^t + a^m(m-1)^t + a^m m^t \\ &\dots \\ &= a^{m-1}S(1) + a^m 2^t + a^m 3^t + \dots + a^m(m-2)^t + a^m(m-1)^t + a^m m^t \\ &= a^{m-1}S(1) + a^m(2^t + 3^t + \dots + (m-2)^t + (m-1)^t + m^t) \end{aligned}$$

Но $2^t + 3^t + \dots + (m - 2)^t + (m - 1)^t + m^t = \Theta(m^{t+1})$, защото $1^k + 2^k + \dots + n^k = \Theta(n^{k+1})$; за извеждането на това вижте [21, (6.98) на стр. 288]. Тогава в сила е

$$\begin{aligned} a^{m-1}S(1) + a^m(2^t + 3^t + \dots + (m - 2)^t + (m - 1)^t + m^t) = \\ a^{m-1}S(1) + a^m\Theta(m^{t+1}) = \\ \Theta(a^m m^{t+1}) \end{aligned}$$

Изведохме:

$$S(m) = \Theta(a^m m^{t+1}) \quad (5.41)$$

Да се върнем към началните T и n . (5.41) става

$$T(n) = \Theta(a^{\log_b n} (\log_b n)^{t+1})$$

Имайки предвид, че $a^{\log_b n} = n^{\log_b a}$ и $\log_b n = \Theta(\lg n)$, заключаваме, че:

$$T(n) = \Theta(n^{\log_b a} \lg^{t+1} n)$$

□

Пример 4. Ще решим

$$T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$$

Да опитаме с МТ. Имаме $a = 2$, $b = 2$, така че $\log_b a$ е $\log_2 2 = 1$, следователно $n^{\log_b a} = n^1 = n$. Нехомогенната част е $n \lg n$. Да видим дали можем да класифицираме тази задача в някой от трите случая на МТ.

Опитваме случай 1: Дали $n \lg n = O(n^1/n^\epsilon)$ за някое $\epsilon > 0$? Не, защото $n \lg n = \omega(n^1)$.

Опитваме случай: Дали $n \lg n = \Theta(n^1)$? Не, защото $n \lg n = \omega(n^1)$.

Опитваме случай 3: Дали $n \lg n = \Omega(n^1 \cdot n^\epsilon)$ за някое $\epsilon > 0$? Не. Вижте Теорема 10.

Следователно, тази задача не може да се реши с МТ. Ще я решим чрез Теорема 19, съгласно която $T(n) = \Theta(n \lg^2 n)$.

5.3.5 Чрез Теорема 20

Теорема 20 ни позволява да решаваме рекурентни уравнения от доста ограничен вид, които обаче не могат да бъдат решени с МТ. Необходимостта от тази теорема възникна при анализа на сложността на алгоритъма SELECT в Подсекция 5.4.2. Рекурентното уравнение (5.61), което описва сложността на SELECT, е решено по индукция в [15, стр. 222]. Лесно се вижда, че уравнение (5.61) може да се обобщи до (5.42), като (5.42) има линейно решение—също както и (5.61)—който факт може да се докаже по индукция по начин, напълно аналогичен на този от [15, стр. 222].

(5.42) описва сложността на алгоритъм, изграден по схемата Разделяй-и-Владей, който прави k на брой рекурсивни викания върху входове, сумарната големина на които е по-малка от

входа му, и освен това върши поне линейна работа във фазите **Разделяй** и **Комбинирай**. Нехомогенната част $f(n)$ описва точно тази работа на алгоритъма.

Теорема 20

Нека b_1, \dots, b_k са константи, всяка от които е по-голяма от 1, и освен това $\sum_{i=1}^k \frac{1}{b_i} < 1$. Нека $f(n) = \Omega(n)$. Тогава рекурентното уравнение

$$T(n) = T\left(\frac{n}{b_1}\right) + \dots + T\left(\frac{n}{b_k}\right) + f(n) \quad (5.42)$$

има решение $T(n) \asymp f(n)$.

Доказателство: Това, че $T(n) \geq f(n)$, е очевидно. Ще докажем по индукция, че $T(n) \leq f(n)$. Това е същото като да докажем, че съществува положителна константа c , такава че за всички достатъчно големи стойности на n е вярно, че

$$T(n) \leq cf(n)$$

Допускаме, че съществува такава положителна константа c , че за всички стойности на аргумента, по-малки от n , желаното неравенство е изпълнено. В частност,

$$T\left(\frac{n}{b_1}\right) \leq c \frac{n}{b_1} \quad (5.43)$$

$$\dots$$

$$T\left(\frac{n}{b_k}\right) \leq c \frac{n}{b_k} \quad (5.44)$$

Имайки предвид (5.42), (5.43), ..., (5.44), имаме право да твърдим, че

$$T(n) = c \frac{n}{b_1} + \dots + c \frac{n}{b_k} + f(n)$$

Дали е вярно, че дясната страна не надхвърля $cf(n)$ за някоя стойност на c ? Тоест, дали има $c > 0$, такава че

$$c \frac{n}{b_1} + \dots + c \frac{n}{b_k} + f(n) \leq cf(n) \Leftrightarrow f(n) \leq c \left(f(n) - n \sum_{i=1}^k \frac{1}{b_i} \right) \Leftrightarrow c \geq \frac{f(n)}{f(n) - nB} \quad (5.45)$$

където $B = \sum_{i=1}^k \frac{1}{b_i}$. По условие, $B < 1$. Но (5.45) е същото като

$$c \geq \frac{1}{1 - B \frac{n}{f(n)}} \quad (5.46)$$

Припомняме си, че $f(n) \geq n$. Това влече, че L , такова че $0 < L < \infty$ и $\frac{n}{f(n)} < L$ за всички достатъчно големи стойности на n . Тогава очевидно такава положителна c съществува и това е краят на доказателството.

5.3.6 Чрез метода с характеристичното уравнение

Ще изложим следната теорема без доказателство съгласно учебника Увод в Дискретната Математика на Кр. Манев [45, стр. 60]. Аргументация на твърдение има, например, в книгата на Flajolet и Sedgewick [60].

Theorem 1. Нека редицата $\tilde{a} = (a_0, a_1, a_2, \dots)$ е зададена с линейното рекурентно уравнение

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_{r-1} a_{n-(r-1)} + c_r a_{n-r}, \quad n \geq r \quad (5.47)$$

Нека $\alpha_1, \alpha_2, \dots, \alpha_s$ са различните комплексни корени на характеристичното уравнение

$$x^r - c_1 x^{r-1} - c_2 x^{r-2} - \dots - c_{r-1} x - c_r = 0 \quad (5.48)$$

като α_i е с кратност k_i for $1 \leq i \leq s$ [†]. Тогава

$$a_n = P_1(n) \alpha_1^n + P_2(n) \alpha_2^n + \dots + P_s(n) \alpha_s^n \quad (5.49)$$

където $P_i(n)$ е полином на n от степен $< k_i$. Полиномите $P_1(n), P_2(n), \dots, P_s(n)$ имат общо r коефициента, които се определят еднозначно от първите r члена на редицата \tilde{a} . \square

В текущата терминология, (5.47) изглежда така:

$$T(n) = c_1 T(n-1) + c_2 T(n-2) + \dots + c_{r-1} T(n-(r-1)) + c_r T(n-r) \quad (5.50)$$

Както вече казахме, това е общ пример за линейно хомогенно рекурентно уравнение с константни коефициенти и крайна история. Такова рекурентно уравнение не може да описва сложността на рекурсивен алгоритъм, защото алгоритъмът не може да не извърши поне константна работа извън рекурсивните викания. Понеже се интересуваме от анализ на алгоритми, налага се да разгледаме по-общ вид рекурентни уравнения, а именно нехомогенни. Ще разглеждаме нехомогенни рекурентни уравнения от този вид:

$$\begin{aligned} T(n) = & c_1 T(n-1) + c_2 T(n-2) + \dots + c_{r-1} T(n-(r-1)) + c_r T(n-r) \\ & + b_1^n Q_1(n) + b_2^n Q_2(n) + \dots + b_m^n Q_m(n) \end{aligned} \quad (5.51)$$

b_1, b_2, \dots, b_m са различни положителни константи. $Q_1(n), Q_2(n), \dots, Q_m(n)$ са полиноми от съответни степени d_1, d_2, \dots, d_m .

Определение 27: Мултимножество: нотация, кратност, обединение, кардиналност

Със нотацията “ $\{\quad\}_M$ ” означаваме мултимножества. Например, $\{1, 1, 2, 3, 3, 3\}_M$ е мултимножеството, състоящо се от две единици, една двойка и две тройки. За всеки елемент a по отношение на някакво мултимножество A , с $\#(a, A)$ означаваме броя на появите на a в A , когото наричаме *кратността на a в A*. $\#(a, A) = 0$, ако a не се среща в A . Например, $\#(1, \{1, 1, 2, 3, 3, 3\}_M) = 2$. Обединението на две мултимножества A и B е:

$$A \cup B = \{x \mid (x \in A \text{ или } x \in B) \text{ и } (\#(x, A \cup B) = \#(x, A) + \#(x, B))\}_M$$

Кардиналност на мултимножеството A е сумата от броевете на появите на елементите му и се означава с $|A|$. Например, $|\{1, 1, 2, 3, 3, 3\}_M| = 6$.

Решението на (5.51) се получава по следния начин. Нека мултимножеството от корените на характеристичното уравнение е A . Очевидно, $|A| = r$. Нека B е мултимножеството от всички b_i , като $\#(b_i, B) = d_i + 1$ за $1 \leq i \leq m$. Нека $Y = A \cup B$. Очевидно, $|Y| = r + \sum_{i=1}^m (d_i + 1)$. Нека

[†]Очевидно, $k_i \geq 1$ за $1 \leq i \leq s$ и освен това, $k_1 + k_2 + \dots + k_s = r$.

преименуваме различните елементи на Y като y_1, y_2, \dots, y_t и да дефинираме, че $\#(y_i, Y) = z_i$, за $1 \leq i \leq t$. Тогава

$$\begin{aligned} T(n) &= \beta_{1,1} y_1^n + \beta_{1,2} n y_1^n + \dots + \beta_{1,z_1} n^{z_1-1} y_1^n + \\ &\quad \beta_{2,1} y_2^n + \beta_{2,2} n y_2^n + \dots + \beta_{2,z_2} n^{z_2-1} y_2^n + \\ &\quad \dots \\ &\quad \beta_{t,1} y_t^n + \beta_{t,2} n y_t^n + \dots + \beta_{t,z_t} n^{z_t-1} y_t^n \end{aligned} \tag{5.52}$$

Индексираните β -и са константи, общо $|Y|$ на брой. Понеже се интересуваме само от асимптотичното нарастване на $T(n)$, точните стойности на константите са без значение. Асимптотичното нарастване се определя от точно едно събирамо, а именно, най-голямото[†] y_i , умножено по най-голямата възможна степен на n .

Ще разгледаме два примера за решаване на рекурентни уравнения с метода с характеристичното уравнение.

Пример 1. Ще решим

$$T(n) = T(n - 1) + 1$$

Преписваме рекурентното уравнение така: $T(n) = T(n - 1) + 1^n n^0$, за да сме сигурни, че формата му удовлетворява (5.51). Характеристичното уравнение е $x - 1 = 0$ с единствен корен $x_1 = 1$. Тогава мултимножеството от корените на характеристичното уравнение е $\{1\}_M$. Използвайки конвенцията за именуване на (5.51), $m = 1$, $b_1 = 1$, и $d_1 = 0$. Така че към мултимножеството $\{1\}_M$ добавяме $b_1 = 1$ с кратност $d_1 + 1 = 1$, получавайки $\{1, 1\}_M$. Тогава $T(n) = A 1^n + B n 1^n$ за някакви константи A и B , следователно $T(n) = \Theta(n)$.

Пример 2. Ще решим

$$T(n) = 4T(n - 3) + 1 \tag{5.53}$$

Характеристичното уравнение е

$$x^3 - 4 = 0$$

Корените му са

$$\begin{aligned} x_1 &= \sqrt[3]{4} \\ x_2 &= \sqrt[3]{4} e^{i \frac{2\pi}{3}} \\ x_3 &= \sqrt[3]{4} e^{i \frac{-2\pi}{3}} \end{aligned}$$

Имайки предвид и нехомогенната част, мултимножеството от корените е

$$\{\sqrt[3]{4}, \sqrt[3]{4} e^{i \frac{2\pi}{3}}, \sqrt[3]{4} e^{i \frac{-2\pi}{3}}, 1\}_M$$

[†]Ако $T(n)$ описва сложността на алгоритъм, най-голямото по абсолютна стойност y_i задължително е положително.

Решението, спрямо някакви комплексни константи A , B , C и D е:

$$\begin{aligned} T(n) &= A \left(\sqrt[3]{4} \right)^n + B \left(\sqrt[3]{4} \right)^n e^{\frac{2\pi n i}{3}} + C \left(\sqrt[3]{4} \right)^n e^{-\frac{2\pi n i}{3}} + D \\ &= A \left(\sqrt[3]{4} \right)^n + B \left(\sqrt[3]{4} \right)^n \left(\cos \left(\frac{2\pi n}{3} \right) + i \sin \left(\frac{2\pi n}{3} \right) \right) + \\ &\quad C \left(\sqrt[3]{4} \right)^n \left(\cos \left(-\frac{2\pi n}{3} \right) + i \sin \left(-\frac{2\pi n}{3} \right) \right) + D \\ &= A \left(\sqrt[3]{4} \right)^n + \left(\sqrt[3]{4} \right)^n \cos \left(\frac{2\pi n}{3} \right) (B + C) + \\ &\quad \left(\sqrt[3]{4} \right)^n \sin \left(\frac{2\pi n}{3} \right) (B - C)i + D \end{aligned}$$

Ако вземем $B = C = \frac{1}{2}$, получаваме едно решение:

$$T_1(n) = A \left(\sqrt[3]{4} \right)^n + \left(\sqrt[3]{4} \right)^n \cos \left(\frac{2\pi n}{3} \right) + D$$

Ако вземем $B = -\frac{1}{2}i$ and $C = \frac{1}{2}i$, получаваме друго решение:

$$T_2(n) = A \left(\sqrt[3]{4} \right)^n + \left(\sqrt[3]{4} \right)^n \sin \left(\frac{2\pi n}{3} \right) + D$$

Съгласно принципа на суперпозицията[†], имаме общо решение

$$T(n) = A_1 \left(\sqrt[3]{4} \right)^n + A_2 \left(\sqrt[3]{4} \right)^n \cos \left(\frac{2\pi n}{3} \right) + A_3 \left(\sqrt[3]{4} \right)^n \sin \left(\frac{2\pi n}{3} \right) + A_4 \quad (5.54)$$

за някакви константи A_1, A_2, A_3 и A_4 . Асимптотиката на решението е $T(n) = \Theta \left(\left(\sqrt[3]{4} \right)^n \right)$.

Допълнение 21: За точното решение на $T(n) = 4T(n-3) + 1$

Точното решение на (5.53) зависи от началните условия, които не са дадени. Да кажем, че началните условия са следните:

$$\begin{aligned} T(1) &= 1 \\ T(2) &= 2 \\ T(3) &= 3 \\ T(4) &= 4 \end{aligned}$$

Щом началните условия са цели числа, то константите A_1, \dots, A_4 в (5.54) трябва да са такива, че $T(n)$ е цяло число за всяко $n \in \mathbb{N}^+$.

Първо да разгледаме общото решение (5.54) и да съобразим какви стойности вземат $\cos \left(\frac{2\pi n}{3} \right)$ и $\sin \left(\frac{2\pi n}{3} \right)$ при цели положителни стойности на n :

[†]Принципът на суперпозицията казва, че ако имаме линейно рекурентно уравнение и знаем, че някакви функции g_i , където $1 \leq i \leq k$, са решения, тогава всяка тяхна линейна комбинация също е решение. Виж [60] или [4, стр. 97].

n	1	2	3	4	5	6	7	8	9	10
$\cos\left(\frac{2\pi n}{3}\right)$	− $\frac{1}{2}$	− $\frac{1}{2}$	1	− $\frac{1}{2}$	− $\frac{1}{2}$	1	− $\frac{1}{2}$	− $\frac{1}{2}$	1	− $\frac{1}{2}$
$\sin\left(\frac{2\pi n}{3}\right)$	$\frac{\sqrt{3}}{2}$	− $\frac{\sqrt{3}}{2}$	0	$\frac{\sqrt{3}}{2}$	− $\frac{\sqrt{3}}{2}$	0	$\frac{\sqrt{3}}{2}$	− $\frac{\sqrt{3}}{2}$	0	$\frac{\sqrt{3}}{2}$

Забележете какво означава това за общото решение (5.54). Тъй като $\cos\left(\frac{2\pi n}{3}\right)$ и $\sin\left(\frac{2\pi n}{3}\right)$ задават периодични числови редици, и двете с период 3, можем да мислим за (5.54) като за **три** израза:

$$T(n) = \begin{cases} A_1 (\sqrt[3]{4})^n - A_2 \frac{(\sqrt[3]{4})^n}{2} + A_3 \frac{(\sqrt[3]{4})^n \sqrt{3}}{2} + A_4, & \text{ако } n \equiv 1 \pmod{3} \\ A_1 (\sqrt[3]{4})^n - A_2 \frac{(\sqrt[3]{4})^n}{2} - A_3 \frac{(\sqrt[3]{4})^n \sqrt{3}}{2} + A_4, & \text{ако } n \equiv 2 \pmod{3} \\ A_1 (\sqrt[3]{4})^n + A_2 (\sqrt[3]{4})^n + A_4, & \text{ако } n \equiv 0 \pmod{3} \end{cases} \quad (5.55)$$

За да намерим A_1, \dots, A_4 , в (5.54) последователно заместваме n с 1, 2, 3 и 4, като за $T(1), \dots, T(4)$ ползваме началните условия, и така получаваме следната система:

$$\begin{aligned} 1 &= A_1 4^{1/3} - A_2 \frac{4^{1/3}}{2} + A_3 \frac{4^{1/3} \sqrt{3}}{2} + A_4 \\ 2 &= A_1 4^{2/3} - A_2 \frac{4^{2/3}}{2} - A_3 \frac{4^{2/3} \sqrt{3}}{2} + A_4 \\ 3 &= A_1 + A_2 4 + A_4 \\ 4 &= A_1 4^{4/3} - A_2 2 \cdot 4^{1/3} + A_3 2\sqrt{3} 4^{1/3} + A_4 \end{aligned}$$

Решението на системата е

$$\begin{aligned} A_1 &= \frac{1}{4} + \frac{4^{2/3}}{12} + \frac{4^{1/3}}{6} \\ A_2 &= -\frac{4^{2/3}}{12} - \frac{4^{1/3}}{6} + \frac{1}{2} \\ A_3 &= \frac{4^{2/3} \sqrt{3}}{12} - \frac{4^{1/3} \sqrt{3}}{6} \\ A_4 &= 0 \end{aligned}$$

Тогава точното решение е

$$\begin{aligned} T(n) &= \left(\frac{1}{4} + \frac{4^{2/3}}{12} + \frac{4^{1/3}}{6} \right) 4^{n/3} + \\ &\quad \left(-\frac{4^{2/3}}{12} - \frac{4^{1/3}}{6} + \frac{1}{2} \right) 4^{n/3} \cos\left(\frac{2\pi n}{3}\right) + \\ &\quad \left(\frac{4^{2/3} \sqrt{3}}{12} - \frac{4^{1/3} \sqrt{3}}{6} \right) 4^{n/3} \sin\left(\frac{2\pi n}{3}\right) \end{aligned} \quad (5.56)$$

Колкото и да е странно, този израз е цяло положително число за всяко $n \geq 1$.

Имайки предвид, че $\cos\left(\frac{2\pi n}{3}\right)$ и $\sin\left(\frac{2\pi n}{3}\right)$ задават периодични числови редици, и двете с период 3, можем да мислим за (5.56) като три отделни израза, всеки с константни коефициенти – също както (5.55).

Има и по-просто, но по-непрецизно решение на (5.53). Да заменим n с $3m$, тоест казваме, че $n = 3m$. Тогава (5.53) става

$$T(3m) = 4T(3m - 3) + 1 \leftrightarrow T(3m) = 4T(3(m - 1)) + 1$$

Да направим второ полагане, този път не на променлива, а на функция. Да наречем $T(3m)$ с името $S(m)$. Тогава $T(3(m - 1))$ ще се нарича $S(m - 1)$. Рекурентното уравнение се записва като

$$S(m) = 4S(m - 1) + 1$$

Тривиално се показва чрез метода с характеристичното уравнение, че асимптотичното решение е $S(m) = \Theta(4^m)$. Това решение е елементарно, защото характеристичното уравнение е $x - 4 = 0$ с единствен корен 4. Сега се връщаме към началните имена. Първо се връщаме към началното име на функцията (което беше T):

$$T(3m) = \Theta(4^m)$$

След това се връщаме към началното име на променливата (което беше n), припомняйки си, че $m = \frac{n}{3}$:

$$T(n) = \Theta\left(4^{\frac{n}{3}}\right)$$

И пак получаваме, че $T(n) = \Theta\left(\left(\sqrt[3]{4}\right)^n\right)$. Това решение беше по-просто, но в някакъв смисъл, по-непрецизно. Замествайки n с $3m$, ако смятаме, че m е естествено, ние все едно оценяваме (5.53) само върху числата, кратни на 3, и за тях получаваме, че асимптотиката е $\left(\sqrt[3]{4}\right)^n$. Не сме доказали теоретичен резултат, който да ни позволява да твърдим, че оттук следва, че асимптотиката върху всички естествени n е същата. В това е и непрецизността на идеята за полагането. На практика обаче полагането се ползва широко за “добре държащи се функции” (каквато е 4^m), като оценяваме рекурентното уравнение (полагайки) само върху някои числа (все пак, безкрайно множество) и оттам извеждаме, че върху всички числа асимптотиката е същата.

5.4 Примери за ефикасни Разделяй-и-Владей алгоритми

Ще разгледаме два примера за задачи, които се решават ефикасно с алгоритми, конструирани по схемата Разделяй-и-Владей.

5.4.1 Най-близки елементи, 2D

Задачата е обобщение на НАЙ-БЛИЗКИ ЕЛЕМЕНТИ на стр. 83. Сега са дадени не числа, а двойки от числа, и се търси най-близка двойка. Може да мислим за задачата като за геометрична задача, където двойките са точки в Евклидовата равнина, но може да мислим и за по-общ контекст, в който точките са елементи на някакво фазово пространство.

Има качествена разлика между 1D и 2D вариантите на задачата: 1D вариантът има ефикасно решение със сортиране и премитане, което не се обобщава за повече измерения.

Оригиналната статия с решение на тази задача е от 1976 година и съдържа ефикасно решение както за двумерния случай, така и за k -мерния случай, ако k е константа [10].

Разстоянието между две двойки числа (a, b) и (c, d) най-често се дефинира като Евклидовото разстояние, което още се нарича L_2 разстоянието[†]

$$\sqrt{(a - c)^2 + (b - d)^2}$$

но може да бъде Манхатънското разстояние, което още се нарича L_1 разстоянието

$$|a - c| + |b - d|$$

или L_∞ разстоянието

$$\max \{|a - c|, |b - d|\}$$

Как точно се дефинира разстоянието не е особено важно, въпреки че някои детайли от алгоритъма, който ще разгледаме, зависят от това. Най-важното е разстоянието да може да се пресмята в $\Theta(1)$.

Изч. Задача: Най-близки Елементи, 2D, вар. 1

пример: Множество от точки $P = \{p_1, p_2, \dots, p_n\}$, където $p_i = (a_i, b_i)$, където $a_i, b_i \in \mathbb{Q}$, за $1 \leq i \leq n; n \geq 2$

решение: $\min \{\text{dist}(p_i, p_j) : 1 \leq i < j \leq n\}$

Изч. Задача: Най-близки Елементи, 2D, вар. 2

пример: Множество от точки $P = \{p_1, p_2, \dots, p_n\}$, където $p_i = (a_i, b_i)$, където $a_i, b_i \in \mathbb{Q}$, за $1 \leq i \leq n; n \geq 2$

решение: Двойка точки (p_i, p_j) , такива че $i \neq j$ и $\text{dist}(p_i, p_j)$ е минимална.

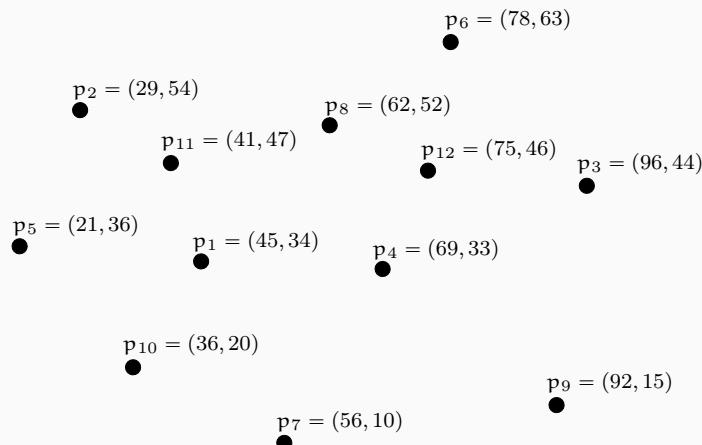
Числата a_i и b_i са произволни рационални числа. Може две точки да имат една и съща първа координата или една и съща втора координата. Може и двете им координати да съвпадат, което означава, че това е една и съща точка в равнината. Може дори всички двойки-точки да съвпадат. Това не е непременно безсмислица. За целите на това изложение смятаме, че всяка точка се идентифицира с координатите си, но в някакъв практически контекст “точките” може да са много по-сложни данни и двете координати да са само два от многото атрибути на една точка.

Ние ще решим задачата в по-лесния първи вариант, в който се търси само оптималното разстояние. Има очевидно решение с брутална сила, опитващо всички ненаредени двойки точки, но то работи във време $\Theta(n^2)$, което е прекалено бавно. Ще построим алгоритъм, работещ в $\Theta(n \lg n)$, изграден по схемата Разделяй-и-Владей. Изложението тук повтаря в основни линии изложението в [15, стр. 1039].

Ще направим пример с точките, показани на Фигура 5.7.

[†] L_m разстоянието между (a, b) и (c, d) е $(|a - c|^m + |b - d|^m)^{\frac{1}{m}}$.

Фигура 5.7 : Точките, които ще ползваме за пример.



Нотация 3

Нека е дадена точка $p \in P$. Тогава $p.x$ означава нейната първа координата, а $p.y$ означава нейната втора координата. Например, на Фигура 5.7, $p_{11}.x = 41$, $p_8.y = 52$ и така нататък.

Предварителна обработка (preprocessing). Сортираме P веднъж по първата координата и записваме резултата в масив $A[1, \dots, n]$, и после още веднъж по втората координата и записваме резултата в масив $B[1, \dots, n]$. Подчертаваме две неща.

1. A и B са масиви от точки, а не масиви от числа. В нашия пример:

$$\begin{aligned} A &= [p_5, p_2, p_{10}, p_{11}, p_1, p_7, p_8, p_4, p_{12}, p_6, p_9, p_3] \\ B &= [p_7, p_9, p_{10}, p_4, p_1, p_5, p_3, p_{12}, p_{11}, p_8, p_2, p_6] \end{aligned}$$

2. Сортирането **не е** част от Разделяй-и-Владей! Ако беше част от него, щяхме да сортираме на всяко ниво на рекурсията. Ние сортираме **само веднъж**: в предварителната обработка. По време на същинското изпълнение на рекурсивния алгоритъм не сортираме, а ползваме началната сортировка.

Това е съществено за ефикасността на алгоритъма. Ако сортираме на всяко ниво на рекурсията (в която делим входа на две равни части и правим по едно рекурсивно викане върху всяка от тях), сложността по време ще се описва от рекурентното уравнение

$$T(n) = 2T\left(\frac{n}{2}\right) + \Omega(n \lg n)$$

Причината да има нехомогенна част $\Omega(n \lg n)$ е, че сортирането изисква време поне $n \lg n$, в асимптотичния смисъл (вижте Подсекция 7.2.2). Но рекурентното уравнение $S(n) = 2S\left(\frac{n}{2}\right) + \Theta(n \lg n)$ има решение $S(n) = \Theta(n \lg^2 n)$ (вижте Теорема 19 в Допълнение 20). Тогава уравнението $T(n) = 2T\left(\frac{n}{2}\right) + \Omega(n \lg n)$ има решение $T(n) = \Omega(n \lg^2 n)$. Еrgo, ако сортираме на всяко ниво на рекурсията, няма как да “слезем” под $\Theta(n \lg^2 n)$, така че $\Theta(n \lg n)$ би била непостижима сложност за алгоритъма.

Продължаваме изложението за предварителната обработка. Заради ефикасността на алгоритъма **има смисъл всеки p_i от B да съдържа и индекса на p_i в A** . Заради това, елементите на B всъщност са наредени двойки от вида (p_i, j) , където $p_i \in P$, а j е позицията на p_i в A . Да си припомним, че в нашия пример според досегашните дефиниции имаме:

$$\begin{aligned} A &= [p_5, p_2, p_{10}, p_{11}, p_1, p_7, p_8, p_4, p_{12}, p_6, p_9, p_3] \\ B &= [p_7, p_9, p_{10}, p_4, p_1, p_5, p_3, p_{12}, p_{11}, p_8, p_2, p_6] \end{aligned}$$

Сега казваме, че всъщност елементите на B са наредени двойки от точка и индекс. В нашия пример, искаме след предварителната обработка B да изглежда така:

$$B = [(p_7, 6), (p_9, 11), (p_{10}, 3), (p_4, 8), (p_1, 5), (p_5, 1), (p_3, 12), (p_{12}, 9), (p_{11}, 4), (p_8, 7), (p_2, 2), (p_6, 10)] \quad (5.57)$$

Смисълът от това ще стане ясен нататък.

Как да изчисляваме индексите Тези индекси трябва да бъдат изчислени ефикасно. Не е ефикасна следната идея: след създаването на A и B като сортирани масиви от точки, за всеки елемент на B да търсим къде се намира той (същата точка) в A . Това би означавало предварителната обработка да има квадратична сложност, което би означавало на свой ред целият алгоритъм да има сложност $\Omega(n^2)$.

Ефикасен е следният начин за пресмятане на тези индекси. Поначало всеки от тези индекси е индексът на дадената точка в P , ако мислим за P като за масив[†]. В нашия пример, да кажем, че P е

$$P = [p_4, p_2, p_3, p_{12}, p_1, p_8, p_7, p_5, p_{10}, p_{11}, p_6, p_9]$$

Едно уточнение. “Истинските” елементи на P са наредени двойки от координати. Тези означения: “ p_4 ”, “ p_2 ” и така нататък са съкратени записи за наше удобство. “Истинският” P е (Фигура 5.7):

$$P = [(69, 33), (29, 54), (96, 44), (75, 46), (45, 34), (62, 52), (56, 10), (21, 36), (36, 20), (41, 47), (78, 63), (92, 15)]$$

Тогава B непосредствено след сортирането си е

$$B = [(p_7, 7), (p_9, 12), (p_{10}, 9), (p_4, 1), (p_1, 5), (p_5, 8), (p_3, 3), (p_{12}, 4), (p_{11}, 10), (p_8, 6), (p_2, 2), (p_6, 11)]$$

като всъщност B е

$$B = [(56, 10), 7), ((92, 15), 12), ((36, 20), 9), ((69, 33), 1), ((45, 34), 5), ((21, 36), 8), ((96, 44), 3), ((75, 46), 4), ((41, 47), 10), ((62, 52), 6), ((29, 54), 2), ((78, 63), 11)]$$

Нека елементите на P всъщност са наредени двойки (p_i, k) , като p_i е точка, а k е индексът на точката в сортирания масив A . Поначало тези индекси са недефинирани; тоест, P поначало е

$$P = [(69, 33), ?), ((29, 54), ?), ((96, 44), ?), ((75, 46), ?), ((45, 34), ?), ((62, 52), ?), ((56, 10), ?), ((21, 36), ?), ((36, 20), ?), ((41, 47), ?), ((78, 63), ?), ((92, 15), ?)]$$

[†]Дори множеството P да не е реализирано чрез масив, всеки негов елемент-точка има адрес в паметта, който адрес е известен.

Нека елементите на A всъщност са наредени двойки (p_i, s) , като p_i е точка, а s е индексът на точката p_i в P . След сортирането си A става:

$$A = [(p_5, 8), (p_2, 2), (p_{10}, 9), (p_{11}, 10), (p_1, 5), (p_7, 7), (p_8, 6), (p_4, 1), (p_{12}, 4), (p_6, 11), (p_9, 12), (p_3, 3)]$$

което всъщност е

$$A = [(21, 36), 8), ((29, 54), 2), ((36, 20), 9), ((41, 47), 10), ((45, 34), 5), ((56, 10), 7), ((62, 52), 6), ((69, 33), 1), ((75, 46), 4), ((78, 63), 11), ((92, 15), 12), ((96, 44), 3)]$$

С едно сканиране на този A отляво надясно запълваме липсващите индекси в P :

$$P = [(69, 33), 8), ((29, 54), 2), ((96, 44), 12), ((75, 46), 9), ((45, 34), 5), ((62, 52), 7), ((56, 10), 6), ((21, 36), 1), ((36, 20), 3), ((41, 47), 4), ((78, 63), 10), ((92, 15), 11)]$$

Да си припомним, че в момента B е

$$B = [(56, 10), 7), ((92, 15), 12), ((36, 20), 9), ((69, 33), 1), ((45, 34), 5), ((21, 36), 8), ((96, 44), 3), ((75, 46), 4), ((41, 47), 10), ((62, 52), 6), ((29, 54), 2), ((78, 63), 11)]$$

С едно сканиране на B отляво надясно променяме индексите на елементите му по следния начин. Нека $B[i]$ съдържа индекс j . Тогава новият индекс на $B[i]$ е индексът, който се съдържа в $P[j]$. Например, $B[1]$ има индекс 7. Тоест, $j = 7$. $P[7]$ съдържа индекс 6. $B[1]$ става $((56, 10), 6)$. Аналогично, $B[2]$ става $((92, 15), 11)$. И така нататък. Наистина, B става

$$B = [(p_7, 6), (p_9, 11), (p_{10}, 3), (p_4, 8), (p_1, 5), (p_5, 1), (p_3, 12), (p_{12}, 9), (p_{11}, 4), (p_8, 7), (p_2, 2), (p_6, 10)]$$

точно както искахме според (5.57). Очевидно е, че генерирането на тези индекси по описанния начин увеличава само с $\Theta(n)$ сложността по време на предварителната обработка. Следователно, сложността по време на предварителната обработка остава $\Theta(n \lg n)$.

След извършването на предварителната обработка започва същинският рекурсивен алгоритъм. Базата е $n \in \{2, 3\}$. Ако $n \in \{2, 3\}$, то решаваме задачата с брутална сила, опитвайки всички $\binom{n}{2}$ ненаредени двойки от числа. Заслужава си да отбележим, че при $n = 1$ задачата е безсмислена (поради което в дефиницията на задачата има изискване $n \geq 2$), така че базата не трябва да е $n = 1$.

Да допуснем, че $n > 3$.

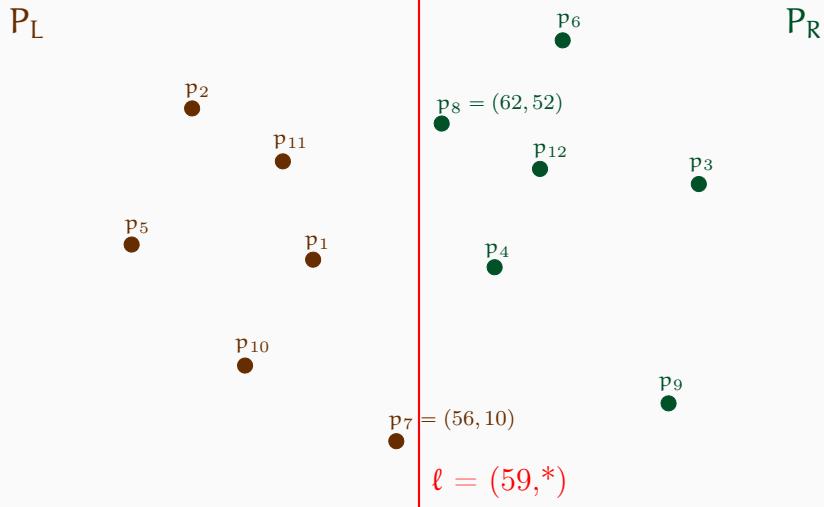
Фаза 1: разделяй. Разбиваме P на две подмножества с максимално близки мощности. Това действие ще наричаме *бисекция*. Разбиването става спрямо някаква права ℓ , като едното подмножество се състои от точките, които са в едната полуравнина спрямо ℓ , а другото подмножество, от точките в другата полуравнина. Правата ℓ се назава *бисектор*. Удачно е ℓ да се успоредна на някоя от координатните оси. Избираме да е успоредна на ординатата[†]. Подмножеството вляво от ℓ наричаме P_L , а това вдясно от ℓ наричаме P_R . Алгоритмично, бисекцията е елементарна. Ние вече имаме масива $A[1, \dots, n]$ от точките, сортирани по първа координата. Нека подмасивът $A[1, \dots, \lceil \frac{n}{2} \rceil]$ се назава A_L , а подмасивът $A[\lceil \frac{n}{2} \rceil + 1, \dots, n]$ да е A_R .

[†]Бисекторът да е вертикална права не е съществено за решението. Решение с хоризонтални бисектори има същата сложност и като описание, и като алгоритмична сложност.

Тогава P_L се състои точно от елементите на A_L , а P_R се състои точно от елементите на A_R . Бисекторът е “виртуална прока” – мисловна конструкция за по-лесно обяснение на алгоритъма. В действителност бисекторът е число: щом сме избрали бисекторът да е вертикална прока, то всички нейни точки имат една и съща първа координата и това число е действителния бисектор. Нека това число бъде средното аритметично от първите координати на средните точки (при сортиране по първа координата), а именно $\frac{1}{2}(A[\lceil \frac{n}{2} \rceil] \cdot x + A[\lceil \frac{n}{2} \rceil + 1] \cdot x)$. Не е съществено важно бисекторът да е именно средното аритметично! Ако $A[\lceil \frac{n}{2} \rceil] \cdot x$ и $A[\lceil \frac{n}{2} \rceil + 1] \cdot x$ са различни, всяко число от отворения интервал $(A[\lceil \frac{n}{2} \rceil] \cdot x, A[\lceil \frac{n}{2} \rceil + 1] \cdot x)$ би “свършило работа” като бисектор. По отношение на разбиването на P няма значение кое число от този отворен интервал ще вземем. Но за фазата **комбинирай** точната стойност на бисектора има значение, както ще видим надолу.

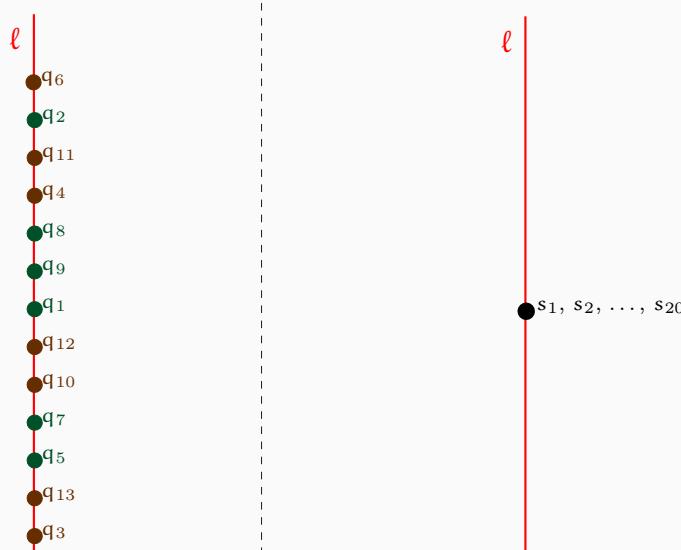
Като пример вижте Фигура 5.8. $n = 12$, така че $\lceil \frac{n}{2} \rceil = 6$ и $\lceil \frac{n}{2} \rceil + 1 = 7$, така че $A[\lceil \frac{n}{2} \rceil] \cdot x = 56$ и $A[\lceil \frac{n}{2} \rceil + 1] \cdot x = 62$, така че $\frac{1}{2}(A[\lceil \frac{n}{2} \rceil] \cdot x + A[\lceil \frac{n}{2} \rceil + 1] \cdot x) = \frac{56+62}{2} = 59$, което е и стойността на бисектора.

Фигура 5.8 : Бисекторът ℓ разбива P на P_L и P_R .



Виртуалният бисектор се състои от точките с първа координата 59. Действителният бисектор е само числото 59.

Едно уточнение. Дефиницията на задачата позволява различни точки да имат една и съща първа координата. Тъй като виртуалният бисектор е вертикална прока, при съвпадения на първите координати може бисекторът да минава през много точки, стоящи една над друга или дори съвпадащи. Може дори това да са всички точки. Ако бисекторът минава през повече от една точки, това не е истински бисектор от геометрична гледна точка, защото това коя от тези точки в кое от двете подмножества отива е произволно. Но това е само от геометрична гледна точка. От алгоритмична гледна точка, бисекторът е число и разбиването на сортирания масив A спрямо това число на A_L и A_R (с максимално близки мощности) е без проблемно. Такива дегенеративни случаи са показани на Фигура 5.9.

Фигура 5.9 : Дегенеративни случаи. ℓ си остава бисектор.

Точките имат една и съща първа координата. Визуално, бисекцията $P_L = \{q_1, q_2, q_5, q_7, q_8, q_9\}$ и $P_R = \{q_3, q_4, q_6, q_{10}, q_{11}, q_{12}, q_{13}\}$ е произволна.

Всички точки съвпадат. Бисекцията е произволна и не е показана.

Както вече казахме, алгоритмичната бисекцията става много лесно:

$$A_L = A \left[1, \dots, \left\lceil \frac{n}{2} \right\rceil \right]$$

$$A_R = A \left[\left\lceil \frac{n}{2} \right\rceil + 1, \dots, n \right]$$

P_L се състои точно от елементите на A_L , а P_R се състои точно от елементите на A_R . По отношение на примера, показан на Фигура 5.7 и Фигура 5.8:

$$A_L = [p_5, p_2, p_{10}, p_{11}, p_1, p_7]$$

$$A_R = [p_8, p_4, p_{12}, p_6, p_9, p_3]$$

Налага се обаче да разбием и B на B_L и B_R , където B_L се състои от същите точки като A_L , но сортирани по втората координата, а B_R се състои от същите точки като A_R , но сортирани по втората координата. Да видим кои множества са B_L и B_R в примера на Фигура 5.7 и Фигура 5.8, като първо си припомним B :

$$B = [p_7, p_9, p_{10}, p_4, p_1, p_5, p_3, p_{12}, p_{11}, p_8, p_2, p_6]$$

$$B_L = [p_7, p_{10}, p_1, p_5, p_{11}, p_2]$$

$$B_R = [p_9, p_4, p_3, p_{12}, p_8, p_6]$$

Очевидно B_L и B_R не се получават като първата и втората половина на масива B (за разлика от A_L и A_R , които са именно първата и втората половина на масива A). Това си заслужава да въде подчертано отново.

Наблюдение 17

В текущия контекст, в общия случай не е вярно, че $B_L = B[1, \dots, \lceil \frac{n}{2} \rceil]$ и не е вярно, че $B_R = B[\lceil \frac{n}{2} \rceil + 1, \dots, n]$. Тази разлика в конструирането на B_L и B_R спрямо A_L и A_R идва оттам, че избрахме вертикален, а не хоризонтален бисектор ℓ , с което в някакъв смисъл направихме хоризонталното направление предпочитано пред вертикалното. Естествено, можехме да изберем хоризонтален бисектор и тогава щяхме да предпочитаме вертикалното направление. Тогава B_L и B_R щяха да са съответно първата и втората половина на масива B , а A_L и A_R щяха да се конструират по-трудно.

Забележете, че не трябва да конструираме B_L като сортираме A_L по втора координата, нито да конструираме B_R като сортираме A_R по втора координата. Това би било ефективно, но неефикасно. Вече казахме, че ако искаме сложност по време $O(n \lg n)$ на алгоритъма, не трябва да сортираме на всяко ниво на рекурсията. Ще конструираме B_L и B_R от B във време $O(n)$. Да си припомним, че в предварителната обработка конструирахме масива B така, че всяка точка съдържа и индекса, който има тя самата в масива A . В нашия пример, B е

$$B = [(p_7, 6), (p_9, 11), (p_{10}, 3), (p_4, 8), (p_1, 5), (p_5, 1), \\ (p_3, 12), (p_{12}, 9), (p_{11}, 4), (p_8, 7), (p_2, 2), (p_6, 10)]$$

Слагаме в B_L точно тези елементи на B , чиито индекси в A са в множеството $\{1, \dots, \lceil \frac{n}{2} \rceil\}$, а в B_R слагаме точно тези елементи на B , чиито индекси в A са в множеството $\{\lceil \frac{n}{2} \rceil + 1, \dots, n\}$. В нашия пример, в B_L слагаме елементите на B , чиято втора компонента е в множеството $\{1, \dots, 6\}$, а в B_R слагаме тези, чиято втора компонента е в множеството $\{7, \dots, 12\}$. Забележете, че елементите на B_L са разположени взаимно по същия начин, по който са разположени в B , и елементите на B_R са разположени взаимно по същия начин, по който са разположени в B . Еrgo, генерирането на B_L и B_R можем да постигнем с едно сканиране на B отляво надясно, като слагаме съответния $B[i]$ в B_L или B_R според индекса в A . В нашия пример, $B[1]$ отива в B_L , $B[2]$ отива в B_R , $B[3]$ отива в B_L , и така нататък, като наистина получаваме

$$B_L = [p_7, p_{10}, p_1, p_5, p_{11}, p_2] \\ B_R = [p_9, p_4, p_3, p_{12}, p_8, p_6]$$

Очевидно тази фаза се изпълнява в линейно време.

Фаза 2: владей. Рекурсивно изпълняваме алгоритъма върху P_L и P_R , като от тези две викания получаваме стойности d_L и d_R . d_L е минимално разстояние между точки в P_L , а d_R е минимално разстояние между точки в P_R .

Фаза 3: комбинирай. Нека t е отсечка с минимална дължина, чиито краища са две различни точки от P . Очевидно е, че точно едно от следните е истина:

- Краищата на t са в P_L . В този случай $|t| = d_L$, тъй като допускаме, че рекурсивното викане върху P_L работи коректно.
- Краищата на t са в P_R . В този случай $|t| = d_R$, тъй като допускаме, че рекурсивното викане върху P_R работи коректно.
- Единият край на t е в P_L , а другият е в P_R . В този случай назваме, че t е *прекосяваща отсечка*. Изпълнено е $|t| = d_M$, където

$$d_M = \min \{ \text{dist}(x, y) \mid x \in P_L, y \in P_R \} \quad (5.58)$$

В началото на фазата **комбинирай** ние не разполагаме с d_M и трябва тешкото да го изчислим.

От тези съображения става ясно, че задачата се свежда до намирането на d_M , след което алгоритъмът връща

$$\min \{d_L, d_R, d_M\}$$

Изразът “ $\min \{\text{dist}(x, y) \mid x \in P_L, y \in P_R\}$ ” може да бъде превърнат директно в алгоритъм, който изчислява d_M . Този подход обаче е неефикасен. Той опитва всяка точка от P_L с всяка точка от P_R , тоест опитва всяка прекосяваща отсечка, което води до квадратична сложност по време. Естествено, ако фазата **комбинирай** работи в $\Theta(n^2)$, няма как алгоритъмът да работи в $O(n \lg n)$. Налага се да направим нещо по-умно, за да изчислим d_M . Правим две подобрения.

Първо подобрение. Нека $d = \min \{d_L, d_R\}$. Да дефинираме, че *късите прекосяващи отсечки* са тези с дължина, по-малка от d . Такива може и да няма. Ако няма къси прекосяващи отсечки, решението е d и нашият алгоритъм следва да върне d . Ако има къси прекосяващи отсечки, решението е дължината на най-къса прекосяваща отсечка и нашият алгоритъм следва да върне това число.

Наблюдение 18

Ако никакви прекосяващи отсечки не са къси, то няма смисъл да изчисяваме дължините им при изчисляването на d_M .

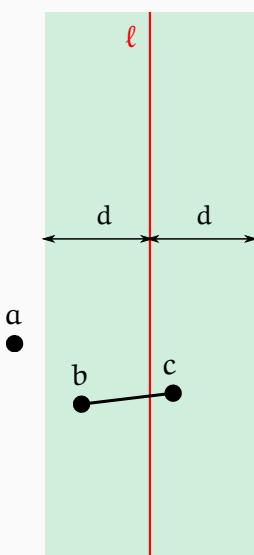
Да си представим точките от P заедно с правата-бисектор ℓ . Да дефинираме *2d-лентата* като областта от равнината, чиито точки са на разстояние не по-голямо от d от двете страни на ℓ . Иначе казано, ако поне единият край на някоя прекосяваща отсечка е извън *2d-лентата*, то нейната дължина е по-голяма от d и не ни интересува.

Наблюдение 19

Ако някоя точка лежи извън *2d-лентата*, то тя не е край на къса прекосяваща отсечка.

Това е илюстрирано на Фигура 5.10, на която *2d-лентата* е показана в светло синьо. Очевидно точка **a** не може да е край на къса прекосяваща отсечка, защото другият край на тази отсечка трябва да е от другата страна на ℓ . Обаче точка **b** се намира в *2d-лентата* и наистина тя е край на къса прекосяваща отсечка (другият ѝ край е **c**).

Фигура 5.10 : 2d-лентата.



И така, първото подобрение е следното. Съгласно Наблюдение 19 и Наблюдение 18, за изчисляването на d_M игнорираме всички точки извън 2d-лентата. Ако поне две точки от P попадат в 2d-лентата, да изчислим d_M само от точките на P , които са в 2d-лентата; а ако в 2d-лентата има нула или една точки от P , то d_M да бъде безкрайност, което е същото като да игнорираме d_M и алгоритъмът да върне d . Ако трябва да сме прецизни, разглеждайки само точките в 2d-лентата, ние не изчисляваме непременно d_M , а нещо друго, което ще наречем d'_M . Причината е, че d_M и d'_M , където d_M е вече дефинирано ((5.58) и (5.59) са еднакви), а d'_M е дефинирано от (5.60), не са непременно равни:

$$d_M = \min \{ \text{dist}(x, y) \mid x \in P_L, y \in P_R \} \quad (5.59)$$

и

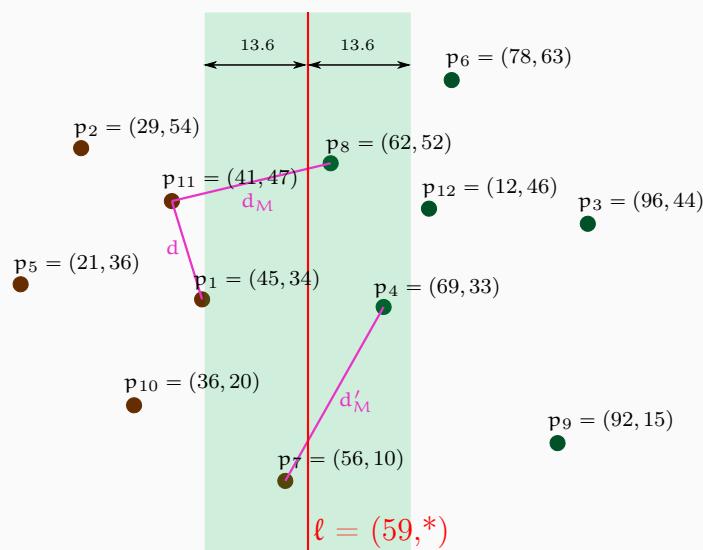
$$d'_M = \min \{ \text{dist}(x, y) \mid x \in P_L \text{ и } x \text{ е в } 2d\text{-лентата}, y \in P_R \text{ и } y \text{ е в } 2d\text{-лентата} \} \quad (5.60)$$

Забележете, че $d'_M \geq d_M$, като неравенство е възможно, а в първото подобрение става дума именно за изчисляването на d'_M . Това обаче не е проблем, защото алгоритъмът връща $\min \{d, d'_M\}$, а ако $d'_M > d_M$, то задължително $d_M > d^\dagger$; ergo, ако $d'_M > d_M$, то решението е d и алгоритъмът ще върне именно d . С други думи, невъзможно е $d'_M > d > d_M$; ако това би било възможно, то би било възможно и да “изтървем” оптимално решение d_M .

Фигура 5.11 показва 2d-лентата в примера от Фигура 5.7. Минималното разстояние между точки от P_L е между $p_1 = (45, 34)$ и $p_{11} = (41, 47)$, а именно $d = \sqrt{(45 - 41)^2 + (34 - 47)^2} \approx 13.60147051$. Тогава $d_L \approx 13.6$. Минималното разстояние между точки от P_R е между p_4 и p_{12} , както и между p_8 и p_{12} , а именно $\sqrt{(75 - 69)^2 + (46 - 33)^2} = \sqrt{(62 - 75)^2 + (52 - 46)^2} \approx 14.31782106$. Тогава $d_R \approx 14.3$ и $d \approx 13.6$. На Фигура 5.11 виждаме P_L и P_R с бисектора $\ell = (59, *)$ между тях и 2d-лентата, простираща се на около 13.6 отляво и отдясно на бисектора.

[†]Това е така, защото, ако $d'_M > d_M$, то d_M е дължина на отсечка, поне единият край на която е извън 2d-лентата и която отсечка, съгласно Наблюдение 19, не е къса, от което следва, че $d_M > d$.

Фигура 5.11 : 2d-лентата в примера от Фигура 5.7.



Само три точки от P попадат в 2d-лентата: p_7 , p_4 и p_8 (p_1 е извън 2d-лентата). Тогава $d'_M = \min \{ \text{dist}(x, y) \mid x \in \{p_7\}, y \in \{p_4, p_8\} \} = \text{dist}(p_4, p_7) = \sqrt{(56 - 69)^2 + (10 - 33)^2} \approx 26.41968963$. Виждаме, че в нашия пример е изпълнено $d_M < d'_M$, понеже d_M се реализира между p_{11} и p_8 , като $\text{dist}(p_{11}, p_8) = \sqrt{(41 - 62)^2 + (47 - 52)^2} \approx 21.58703314$. Но, както вече отбелязахме, ако $d'_M > d_M$, какъвто е този случай, то $d_M > d$, така че за крайното решение няма значение дали сравняваме 13.6 с 26.4 или с 21.5 – решението си остава 13.6.

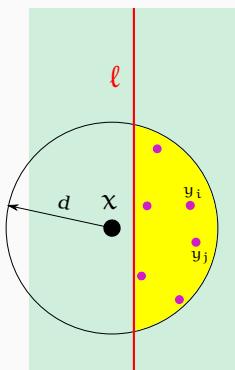
Но по отношение на най-лошия случай само първото подобрение изобщо не е подобрение. Ако всички точки от P са в 2d-лентата и търсим d_M изчерпателно, то сложността пак би била квадратична.

Второ подобрение. Забелязваме, че за всяка точка x в 2d-лентата има твърде ограничен брой други точки, такива че има смисъл да изчисляваме разстоянието между x и всяка от тях. Например, нека $x \in P_L$. Може ли да има сто точки в $y_1, y_2, \dots, y_{100} \in P_R$, такива че във фазата **комбинирай** всяка от тях да е на разстояние, по-малко от d , от точка x ? Отговорът е, че не може. За да е изпълнено това, би трябвало y_1, \dots, y_{100} да са в кръга с център x и радиус d ; по-точно казано, в сечението на този кръг и дясната половина на 2d-лентата. Но тогава биха съществували различни y_i и y_j , такива че $\text{dist}(y_i, y_j) < d$, което е невъзможно, защото $d = \min \{d_L, d_R\}$ по дефиниция.

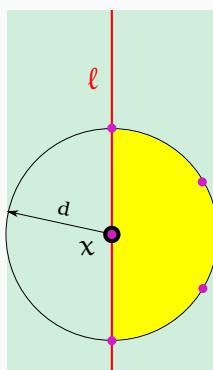
Това е илюстрирано на Фигура 5.12. Сечението на кръга с център x и радиус d с дясната половина на 2d-лентата е оцветено в жълто. Вляво разглеждаме възможността да има шест точки (в лилаво), всяка от които е “кандидат” за друг край на къса прекосяваща отсечка (единият край е x). Очевидно е, че както и да бъдат разположени шестте лилави точки в жълтия сегмент, няма как всеки две от тях да са на разстояние поне d една от друга. Например, на фигурата y_i и y_j са толкова близо една до друга, че разстоянието d_R не може да е повече от $\text{dist}(y_i, y_j)$, следователно, би трябвало $d \leq \text{dist}(y_i, y_j)$, в противоречие с показаното на фигурата.

Вдясно илюстрираме факта, че пет е точната горна граница за броя на въпросните кандидати. Едната от десните (лилавите) точки съвпада с x , което не е невъзможно. x лежи точно върху бисектора, така че сечението на кръга с диаметър d и център x с дясната страна на 2d-лентата е полуокръг. Ако d_R е минималното разстояние между точки измежду тези пет (това е възможно), и $d = d_R$ (и това е възможно), и петте лилави точки попадат в жълтия полуокръг.

Фигура 5.12 : Възможните други краища на къса прек. отсечка са малко.



Очевидно $\text{dist}(y_i, y_j) < d$. Очевидно няма как да разположим шест точки в жълтия сегмент, така че всеки две от тях да са на разстояние поне d една от друга.

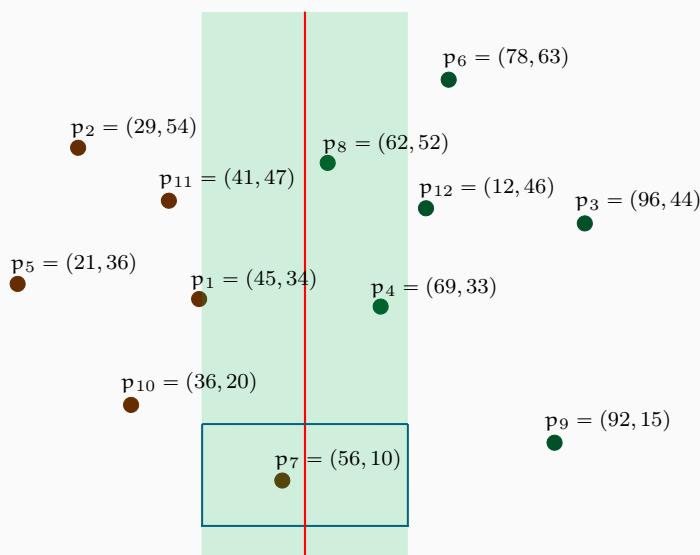


Ако жълтият сегмент е полукръг, може да разположим пет точки в него, така че всеки две да са на поне d една от друга. Едната от тези точки съвпада с x . Повече от пет точки не може.

След като се убедихме, че за всяка точка от $2d$ -лентата трябва да разглеждаме най-много константен брой други точки в изчисляването на d'_M , да видим как ще реализираме тази идея ефикасно. Досегашните разсъждения навеждат на мисълта да направим следното: за всяка точка x от лявата страна на $2d$ -лентата да намерим всички точки от дясната страна на $2d$ -лентата, които се намират в кръга с център x и радиус d , ако има такива, и да видим коя от тях е най-близо до x . Но това би довело до усложнения, които е по-добре да си спестим. Ще разгледаме подхода на [15, стр. 1040–1043], който е значително по-прост и е достатъчно ефикасен. А досегашните разсъждения за ограничения брой на точките “от другата страна”, които има смисъл да разглеждаме, са само за придобиване на по-добра интуиция.

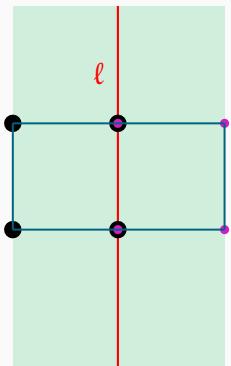
Представяме си *прозорец* – правоъгълник с размери $2d \times d$, със страни, успоредни на координатните оси, като голямата страна е успоредна на абсцисата, а центърът на прозореца лежи върху бисектора. Идеята е да плъзнем прозореца от най-долу (най-малката втора координата) до най-горе (най-голямата втора координата), като центърът му се плъзга по бисектора, а страните му остават успоредни на координатните оси. Фигура 5.13 показва прозореца, нарисуван върху примера от Фигура 5.7.

Фигура 5.13 : Прозорецът върху примера от Фигура 5.7.



На Фигура 5.13 прозорецът обхваща само една точка, а именно p_7 . Колко точки от P —това означава и от P_L , и от P_R —най-много може да обхваща прозореца? Този брой е ограничен, тъй като размерите на прозореца не са произволни. Лесно се забелязва, че най-много осем точки от P може да са в прозореца. Това е демонстрирано на Фигура 5.14. Четирите черни точки са от P_L , а четирите лилави са от P_R . Черните точки са върху върховете на квадратата—лява половина на прозореца, а лилавите са върху върховете на квадратата—дясна половина на прозореца. Две от черните точкти съвпадат с две от лилавите.

Фигура 5.14 : Прозорецът може да обхваща най-много осем точки.



Ето и второто подобрение: да плъзнем прозореца отдолу нагоре в $2d$ -лентата, да изчислим минималното разстояние между две (различни) точки, които се намират в него в кой да е момент, едната от P_L , другата от P_R , и полученото да присвоим на d'_M .

Разбира се, прозорецът е метафора. Ние не правим компютърна геометрия и не плъзгаме правоъгълници, центрирани върху вертикални прости. В действителност правим следното и това е фазата **комбинирай**.

- Първо създаваме масив C от всички точки, намиращи се в $2d$ -лентата, сортирани по втора координата. За тази цел използваме масивите A и B . Да си припомним, че истинският бисектор е число, да го наречем β , а вертикалната праща е виртуален бисектор. Еrgo, създаваме C като подмасивът на A , състоящ се от точно тези точки p_i , за които $|p_i.x - \beta| \leq d$.

Ето как изглежда това в нашия пример. Да си припомним, че

$$A = [(21, 36), (29, 54), (36, 20), (41, 47), (45, 34), (56, 10), (62, 52), (69, 33), (75, 46), (78, 63), (92, 15), (96, 44)]$$

$\beta = 59$ и $d \approx 13.6$. $59 - 13.6 = 45.4$ и $59 + 13.6 = 72.6$, така че точно три точки влизат в C , а именно $(56, 10)$, $(62, 52)$ и $(69, 33)$; нещо, което видяхме още на Фигура 5.11.

И така, използваме A , който е сортиран по първа координата, за да отделим точките, попадащи в $2d$ -лентата.

- От масива B отделяме сортирания по втора координата последователност от тези същите точки, слагайки ги в някакъв масив D в същия ред, в който са в B (тоест, сортирани по втора координата). Това можем да направим ефикасно, защото в предварителната обработка сме добавили към елементите на B техните местоположения в A , така че сега можем във време $\Theta(1)$ да определим за всеки елемент от B дали той е в C или не; с други думи, в константно време можем да определим за произволен елемент от B дали

тази точка попада в $2d$ -лентата или не. Ерго, с едно сканиране на B можем да копираме в D тези елементи, които попадат в $2d$ -лентата, като ги копираме в същия ред, в който са в B .

- В нашия пример D е много малък:

$$D = [(56, 10), (69, 33), (62, 52)]$$

но да допуснем, че D е достатъчно голям. Да кажем, че D има k елемента за някакво $k \geq 8$. Нека $m \leftarrow \infty$. Инициализираме $i \leftarrow 1$, $j \leftarrow 8$ и изпълняваме следното $k - j + 1$ пъти:

- ◆ С брутална сила намираме двойка различни елементи r, s от $D[i, \dots, j]$, които са на минимално разстояние един от друг, като единият от r, s е от P_L , а другият е от P_R . Ако има такива, $m = \min \{m, \text{dist}(r, s)\}$. Това става в константно време, защото броят на ненаредените двойки от $D[i, \dots, j]$ е $\binom{8}{2} = 28$.
- ◆ $i++, j++$.

След края на това, полученото m е точно търсеното d'_M .

Забележете, че масивът $D[i, \dots, j]$ не реализира точно прозореца $2d \times d$, за който стана дума във второто подобрение. Ако D е достатъчно голям, нашата алгоритмична реализация работи с $D[i, \dots, j]$, такъв че $j - i = 7$, тоест, винаги в нашия прозорец има осем точки. Както се вижда на Фигура 5.13, ако прозорецът е наистина $2d \times d$, той може да обхваща само една точка или дори нула точки (лесно се вижда, че има такива негови положения). С други думи, осемте точки, които разглеждаме на всяка итерация, може да бъдат прекалено далече една от друга, за да се намират в рамките на $2d \times d$ прозореца.

Това, че $D[i, \dots, j]$ обхваща винаги осем точки при достатъчно голямо D , не води до проблеми за нашия алгоритъм. Важното е, че гледайки не повече от осем последователни по вертикал точки в даден момент, не можем да изпуснем оптимална стойност на d'_M . Иначе казано, няма смисъл да търсим d'_M , разглеждайки двойка точки, които не са измеждуди осем последователни точки в D . Нашата имплементация може и да опитва да търси d'_M върху точки, които са прекалено далече една от друга, но няма да изпусне оптимално решение и е с линейна сложност.

И така, алгоритъмът връща

$$\min \{d_L, d_R, d'_M\}$$

Коректност. Коректността на алгоритъма е очевидна предвид разсъжденията, които направихме.

Сложност. Очевидно е, че фазата **Комбинирай**, реализирана по гореописания начин, работи във време $\Theta(n)$. Фазата **Разделяй** също работи в линейно време. Тогава сложността на алгоритъма след предварителната обработка се описва от рекурентното уравнение

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

което има решение $T(n) \asymp n \lg n$. Предварителната обработка има сложност $\Theta(n \lg n)$, доминирана от сортиранията; останалата работа в предварителната обработка е $\Theta(n)$. Като цяло, сложността на алгоритъма е $\Theta(n \lg n)$.

5.4.2 Избор на елемент по големина

Този нетривиален алгоритъм е открит от Blum, Floyd, Pratt, Rivest и Tarjan през 70-те [13], [14]. Изложението тук е по учебника на Cormen, Leiserson, Rivest и Stein [15], стр. 220–222].

Изч. Задача: ИЗБОР НА ЕЛЕМЕНТ ПО ГОЛЕМИНА

пример: крайно непразно множество $S \subset \mathbb{Z}$; $k \in \mathbb{N}$, такова че $0 \leq k \leq |S| - 1$

решение: Числото $x \in S$, такова че $|\{a \in S \mid a < x\}| = k$.

Една забележка: очевидно в тази изчислителна задача става дума не за k -ия, а за $(k+1)$ -ия по големина елемент, като k е броят на по-малките от него елементи. Така че формулировката на задачата чрез k —а не чрез $k+1$ —може да е леко подвеждаща, но в [15] задачата е дефинирана точно така.

Тази задача може да се реши тривиално във време $\Theta(n \lg n)$ чрез бързо сортиране на S , например чрез HEAPSORT, последвано от връщане на $(k+1)$ -ия елемент. Тук ще разгледаме по-бързо решение: линеен алгоритъм, наречен PICK в [14] и SELECT в [15].

Алгоритъмът ползва функция PARTITION-HOARE-MOD, която, както показва името, е (лека) модификация на функцията PARTITION-HOARE от алгоритъма QUICKSORT. Подробно изложение и изследване на PARTITION-HOARE ще направим в Лекция 6 и по-специално Секция 6.2.1. Входът на PARTITION-HOARE-MOD е масив $A[l, \dots, h]$, който е подмасив на някакъв масив $A[1, \dots, n]$, и елемент от $A[l, \dots, h]$, наречен pivot. А ефектът от работата на PARTITION-HOARE-MOD е следният: елементите на $A[l, \dots, h]$ биват разместени по такъв начин и върната стойност на j е такава, че:

- всички елементи, по-малки от pivot, са в подмасива $A[l, \dots, j]$;
- $A[j + 1]$ съдържа pivot;
- всички елементи, по-големи от pivot, са в подмасива $A[j + 2, \dots, h]$.

Тук допускаме, че елементите на масива са два по два различни; това допускане не е в сила в Лекция 6, но в контекста на алгоритъма за избор на елемент по големина допускаме, че няма елементи с еднакви ключове. Също така допускаме, че pivot е елемент от $A[l, \dots, h]$, което допускане може да не е в сила при някои реализации на процедурата PARTITION; естествено, при тях разбиването е на два, а не на три, подмасива, понеже $A[j + 1]$ не съдържа непременно pivot.

Подчертаваме, че алгоритъмът PARTITION-HOARE **не сортира**: в края му, нито “малките елементи” в $A[l, \dots, j]$ са непременно сортирани, нито “големите елементи” в $A[j + 2, \dots, h]$ са непременно сортирани. Следва псевдокод на PARTITION-HOARE-MOD. Функцията find-index връща индекса на елемента-първи аргумент в масива-втори аргумент, при допускането, че този елемент се среща точно веднъж в този масив.

PARTITION-HOARE-MOD($A[l, \dots, h]$: int; pivot: element from $A[l, \dots, h]$)

```

1  i ← l - 1, j ← h + 1, more ← TRUE
2  while more do
```

```

3   do
4     i++
5     while A[i] < pivot
6   do
7     j--
8     while A[j] > pivot
9     if i < j
10    swap(A[i], A[j])
11 else
12   more ← FALSE
13 i ← find-index(pivot, A[l, ..., h])
14 swap(A[i], A[j + 1])
15 return j

```

Следва псевдокод на SELECT. Да си припомним дефиницията на ИЗБОР НА ЕЛЕМЕНТ ПО ГОЛЕМИНА на предишната страница. Множеството S е реализирано чрез масив $A[1, \dots, n]$, чиито елементи са два по два различни. Вторият аргумент k е броят на елементите, по-малки от този, който връща алгоритъма. Описанието на алгоритъма, взето от [15], е на по-високо ниво от обичайния псевдокод.

```

SELECT(A[1, ..., n]: int; k ∈ {0, ..., n − 1})
1 (* A[x] ≠ A[y] за  $1 \leq x < y \leq n$  *)
2 if n = 1
3   return A[1]
4 q ←  $\lceil \frac{n}{5} \rceil$ , r ← n mod 5
5 if r = 0
6   разбий A на подмасиви  $B_1, \dots, B_q$ , всеки с 5 елемента
7 else
8   разбий A на подмасиви  $B_1, \dots, B_{q-1}$ , всеки с 5 елемента, и  $B_q$  с  $r$  елемента
9 сортирай всеки от  $B_1, \dots, B_q$  и намери съответната медиана  $m_1, \dots, m_q$ 
10 направи масив  $[m_1, \dots, m_q]$  от медианите и го наречи C
11 m ← SELECT(C,  $\lfloor \frac{q-1}{2} \rfloor$ )
12 j ← PARTITION(A, m)
13 if k = j
14   return m
15 else
16   if k < j
17     return SELECT(A[1, ..., j], k)
18   else
19     return SELECT(A[j + 2, ..., n], k - j - 1)

```

Да разгледаме малък пример. Нека $n = 25$ и елементите на A са $1, 2, \dots, 25$, като

$$A = [17, 6, 8, 22, 11, 1, 7, 4, 19, 21, 3, 20, 16, 24, 10, 25, 12, 18, 15, 14, 13, 2, 23, 9, 5]$$

Нека $k = 17$. Това е входът.

Очевидно $q = 5$. Нека A бъде разбит (ред 6) на 5 подмасива по най-естествения начин:

$$\begin{aligned}B_1 &= [17, 6, 8, 22, 11] \\B_2 &= [1, 7, 4, 19, 21] \\B_3 &= [3, 20, 16, 24, 10] \\B_4 &= [25, 12, 18, 15, 14] \\B_5 &= [13, 2, 23, 9, 5]\end{aligned}$$

След сортирането на всеки от тях (ред 9), те стават:

$$\begin{aligned}B_1 &= [6, 8, 11, 17, 22] \\B_2 &= [1, 4, 7, 19, 21] \\B_3 &= [3, 10, 16, 20, 24] \\B_4 &= [12, 14, 15, 18, 25] \\B_5 &= [2, 5, 9, 13, 23]\end{aligned}$$

Тогава $m_1 = 11$, $m_2 = 7$, $m_3 = 16$, $m_4 = 15$ и $m_5 = 9$, така че $C = [11, 7, 16, 15, 9]$ (ред 10). На ред 11, $\lfloor \frac{6-1}{2} \rfloor = 2$, така че викаме SELECT с втори аргумент 2. Това викане връща 3-ия по големина елемент на C , който е 11, така че на ред 11, m получава стойност 11.

На ред 12 викаме PARTITON върху A с втори аргумент 11. Процедурата връща $j = 10$ (което е напълно очаквано, ако елементите на A са 1, ..., 25 и pivot е 11). Самият A изглежда така след разместванията, направени от PARTITION:

$$A = [5, 6, 8, 9, 2, 1, 7, 4, 10, 3, 11, 20, 16, 24, 19, 25, 12, 18, 15, 14, 13, 21, 23, 22, 17]$$

Тъй като $k = 17$ и $j = 10$, условието на ред 13 е лъжа, условието на ред 16 също е лъжа и изпълнението отива на ред 19. $A[j + 2, \dots, n]$ е $A[12, \dots, 25]$. В случая:

$$A[12, \dots, 25] = [20, 16, 24, 19, 25, 12, 18, 15, 14, 13, 21, 23, 22, 17]$$

$k - j - 1 = 6$, така че викането на ред 19 е $\text{SELECT}(A[12, \dots, 25], 6)$. Очевидно в този пример това е коректно, понеже осемнадесетия ($17 + 1 = 18$) по големина елемент на входния A е 18, което точно съвпада със седмия ($6 + 1 = 7$) по големина елемент на

$$[20, 16, 24, 19, 25, 12, 18, 15, 14, 13, 21, 23, 22, 17]$$

С което приключва разглеждането на примера.

Коректност Сега ще докажем коректността на SELECT. Твърдим, че ако е даден масив $A[1, \dots, n]$ от две по две различни цели числа и $k \in \{0, \dots, n - 1\}$, то $\text{SELECT}(A, k)$ връща $(k + 1)$ -ия по големина елемент на A .

Спирачката на рекурсията е $n = 1$ (ред 2). Ако $n = 1$, единствената допустима стойност за k е 0, понеже $k \in \{0, \dots, n - 1\}$. Наистина, на ред 3 алгоритъмът връща $A[1]$, който наистина е $(k + 1)$ -ия по големина елемент при $k = 0$. ✓

Да допуснем, че $n \geq 2$. Ще покажем коректността на алгоритъма със силна индукция по втория аргумент. Редове 4–11 нямат значение на коректността, а само за ефикасността, така че тук няма да ги разглеждаме. За доказателството за коректност има значение единствено, че m на ред 11 е елемент от масива (очевидно) и че $\text{PARTITION}(A, m)$ на ред 12 размества A по такъв начин и връща такава стойност j , че:

- всички елементи, по-малки от m , са в подмасива $A[1, \dots, j]$;
- $A[j + 1]$ съдържа m ;
- всички елементи, по-големи от m , са в подмасива $A[j + 2, \dots, n]$.

Разглеждаме три случая.

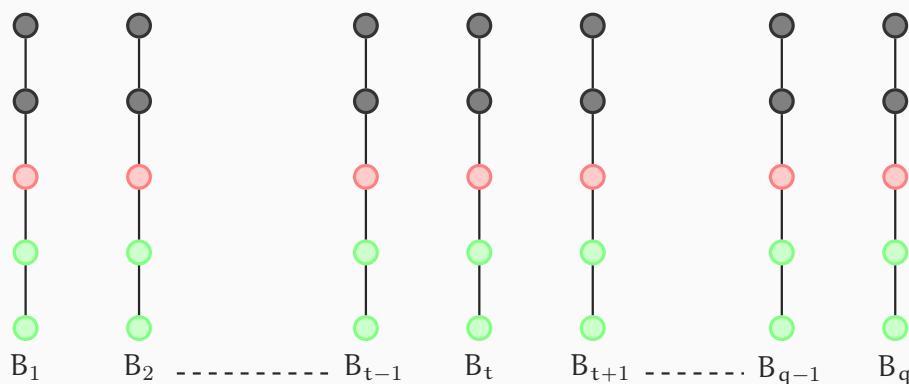
Случай 1. Ако $k = j$ (ред 13), то m е $(k + 1)$ -ият по големина елемент и алгоритъмът коректно връща m на ред 14.

Случай 2. Ако $k < j$ (ред 16), то $(k + 1)$ -ият по големина елемент на входния A е същият като $(k + 1)$ -ият по големина елемент на $A[1, \dots, j]$. За да се убедим в това, да съобразим, че всички елементи, по-малки от m , са в подмасива $A[1, \dots, j]$. Викането на процедурата $\text{SELECT}(A[1, \dots, j], k)$ на ред 17 връща $(k + 1)$ -ият по големина елемент на $A[1, \dots, j]$ съгласно индукционното предположение.

Случай 3. Ако $k > j$ (ред 19), то $(k + 1)$ -ият по големина елемент на входния A е същият като $(k - j)$ -ият по големина елемент на $A[j + 2, \dots, n]$. За да се убедим в това, да съобразим, че преди $(k + 1)$ -ията в A има точно k елемента, което означава, че преди него в $A[j + 2, \dots, n]$ има $k - (j + 1) = k - j - 1$ елемента, защото сега не броим $A[1], A[2], \dots, A[j]$ и $A[j + 1]$. Следователно, викането на $\text{SELECT}(A[j + 2, \dots, n], k - j - 1)$ на ред 19 връща желания елемент.

Сложност по време На редове 5–8 алгоритъмът разбива A на подмасиви: или всички с големина 5, или всички без един с големина 5, а изключението е с големина, по-малка от 5. Тези подмасиви биват сортирани на ред 9, след което е тривиално за всеки от тях да се намери медианата. За удобство да допуснем, че n е кратно на 5 и всички подмасиви са с по точно 5 елемента. Тогава всяка медиана измежду m_1, \dots, m_q е гарантирано по-голяма от точно два елемента и по-малка от точно два елемента от съответния подмасив измежду B_1, \dots, B_q . Фигура 5.15 показва диаграма на Hasse на множеството след ред 9 на SELECT . Всеки от подмасивите B_1, \dots, B_q е линейно наредено, петелементно множество, а за $i \neq j$, всеки елемент от B_i е несравним с всеки елемент от B_j . Във всеки B_i , двата елемента, по-малки от медианата, са изобразени в зелено, а двата елемента, по-големи от медианата, в тъмно сиво.

Фигура 5.15 : Диаграма на Hasse на данните след ред 9 на SELECT .

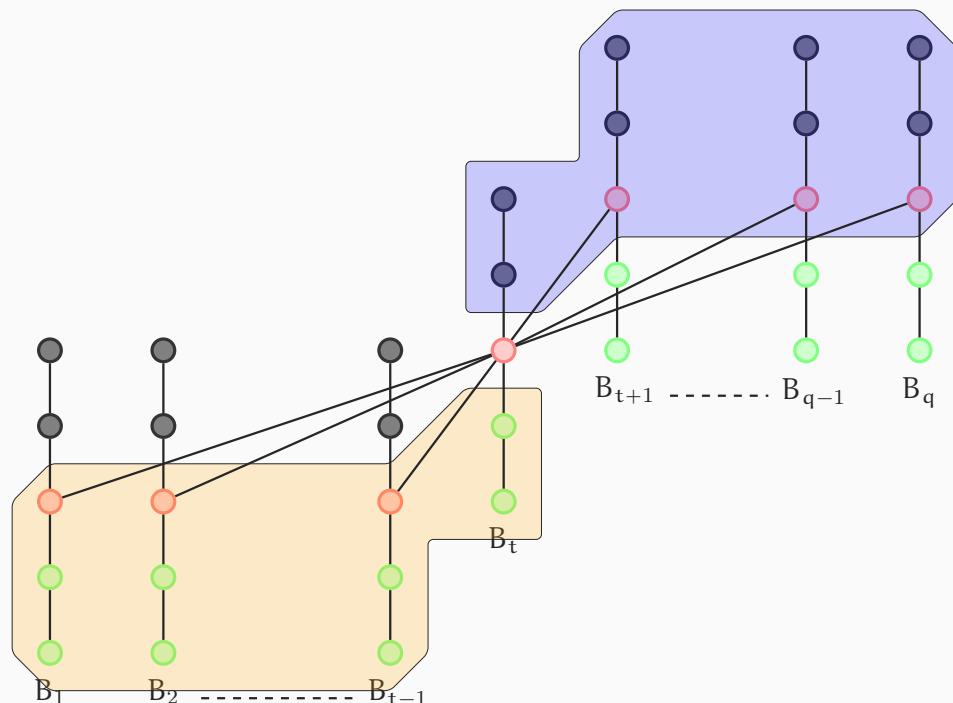


Малките елементи са в зелено, големите са в тъмно сиво, а медианите са в червено.

На ред 11 викаме SELECT върху масива от медианите $[m_1, \dots, m_q]$ с втори аргумент $\lfloor \frac{q-1}{2} \rfloor$. Съгласно индуктивното предположение (да си припомним, че правим доказателството със силна индукция), на ред 11, SELECT връща този елемент на $[m_1, \dots, m_q]$, който по големина е номер $\lfloor \frac{q-1}{2} \rfloor + 1$. Но $\lfloor \frac{q-1}{2} \rfloor + 1 = \lfloor \frac{q-1+2}{2} \rfloor = \lfloor \frac{q+1}{2} \rfloor$, а по определение елементът, който по големина е номер $\lfloor \frac{q+1}{2} \rfloor$, е медианата. Следователно, на ред 11, m получава стойността на медианата на медианите [†].

Да кажем, че медианата на медианите е медианата на B_t (Фигура 5.15), и нека без ограничение на общността медианите на B_1, \dots, B_{t-1} са по-малки от нея, а медианите на B_{t+1}, \dots, B_q са по-големи от нея [‡]. След като бъде установено всичко това, диаграмата на Hasse на данните престава да бъде като на Фигура 5.15 и става като на Фигура 5.16.

Фигура 5.16 : Диаграма на Hasse на данните след ред 11 на SELECT.



Медианата на медианите m е медианата на B_t . Елементите, гарантирано по-малки от m , са показани върху оранжев полупрозрачен фон. Елементите, гарантирано по-големи от m , са показани върху син полупрозрачен фон. Ако q е нечетно, тоест, ако m е средният по големина елемент на масива от медианите, оранжевите и сините елементи са еднакъв брой. В противен случай, сините са повече, понеже вземаме за медиана долния кандидат.

Ще намерим долна граница за броя на елементите, които са по-малки от m . На Фигура 5.16 това са елементите, показани върху оранжев полупрозрачен фон. Това са по точно три елемента от всеки от масивите B_1, \dots, B_{t-1} , плюс още два елемента от B_t . Може да има и други елементи, по-малки от m —някои от зелените елементи на B_{t+1}, \dots, B_q —но само за оранжевите е гарантирано, че са по-малки от m . Броят на оранжевите елементи е точно

$$3 \left\lfloor \frac{q-1}{2} \right\rfloor + 2 = 3 \left\lfloor \frac{\frac{n}{5}-1}{2} \right\rfloor + 2 \approx \frac{3n}{10}$$

[†]Забележете, че медианата на медианите не е непременно медианата на целия масив A . В примера, който видяхме, медианата на медианите беше 11, а медианата на A беше 13.

[‡]От това следва, че $t = \lfloor \frac{q+1}{2} \rfloor$.

Апроксимираме сложния израз с $\frac{3n}{10}$, игнорирайки адитивната константа $+2$, нотацията $\lfloor \rfloor$ и -1 в числителя. Лесно се вижда, че това не променя асимптотиката на решението, което ще получим.

Напълно аналогично, апроксимираме броя на елементите, гарантирано по-големи от m , с $\frac{3n}{10}$.

Това количество $\frac{3n}{10}$ е от решаваща важност за намиране на сложността по време. $\frac{3n}{10}$ е минималният брой елементи, които **гарантирано няма** да бъдат част от множеството, върху което се извършва рекурсивно викане на ред 17 или ред 19. С други думи, на ред 17 или ред 19 рекурсивното викане ще бъде върху вход с размер **най-много** $\frac{7n}{10}$.

Вече можем да построим израз за сложността по време. Имайки предвид, че рекурсивното викане на ред 12 е върху вход с големина приблизително $\frac{n}{5}$, а извън рекурсивните викания, SELECT извършва линейна по време работа, изразът е:

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n \quad (5.61)$$

Съгласно Теорема 20, решението на (5.61) е $T(n) \asymp n$. Това е и края на изследването на сложността по време. SELECT е линеен алгоритъм.

Лекция 6

Сортиращи алгоритми MERGESORT и QUICKSORT.

Резюме: Въвеждаме два бързи сортиращи алгоритъма, MERGESORT и QUICKSORT, и изследваме тяхната коректност, сложност по време и сложност по памет. Изследваме средната сложност по време на QUICKSORT.

6.1 MERGESORT

Сортиращият алгоритъм MERGESORT е изобретен от John von Neumann през 1945 г. [38]. Той е типичен пример за алгоритъм, изграден по схемата Разделяй-и-Владей. Във фазата **Разделяй**, MERGESORT дели входа на две равни части, всяка с размер $\frac{n}{2}$ [†], без да променя наредбата на елементите. Във фазата **Владей** прави по едно рекурсивно викане върху всяка от двете части. Във фазата **Комбинирай** всяка от двете части вече е сортирана, така че алгоритъмът слива (откъдето идва и името, *сливам* в този смисъл е *to merge* на английски) двете сортирани части в една окончателна сортирана последователност. Естествено, всичко това става, когато входът е достатъчно голям. Ако входът е с размер единица или нула, алгоритъмът не прави нищо (зашто празният масив и едноелементният масив са сортирани) – това е спирачката на рекурсията. Акцентът е върху третата фаза, където се извършва истинската работа по сортирането.

Първо ще дадем пример за сортиране с MERGESORT. Примерът нарочно използва вход с големина, която е точна степен на двойката, но кодът, който даваме нататък, работи коректно за всяка големина. И така, нека входът е:

5 2 3 1 4 8 7 6

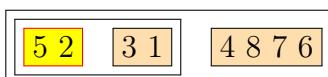
С жълт фон и червен цвят на ограждащата кутия означаваме тази част от масива, която е вход на текущо активното рекурсивно извикване. В самото начало тази част съвпада с целия масив.

Тъй като големината на входа е повече от единица, делим входа на две равни части и правим по едно рекурсивно извикване върху всяка от тях. Тези две рекурсивни извиквания не се случват едновременно, защото нашият изчислителен модел не позволява паралелизъм. Да речем, че първо извикваме рекурсивно алгоритъма върху масиваляво.

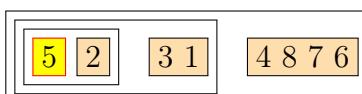
5 2 3 1	4 8 7 6
---------	---------

[†]В действителност разделянето е на една част с размер $\lfloor \frac{n}{2} \rfloor$ и друга част с размер $\lceil \frac{n}{2} \rceil$.

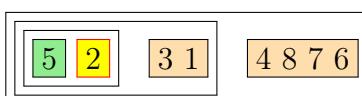
С жълт фон и червена ограждаща кутия е само тази част от началния масив, която е вход на текущото рекурсивно викане. Понеже големината на този вход е четири, което е по-голямо от едно, пак делим входа на две и правим две рекурсивни викания, първото върху частта вляво:



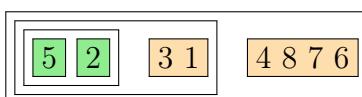
Тъй като големината на входа на текущото рекурсивно викане е две, което е по-голямо от едно, пак делим входа на две правим две рекурсивни викания, първото върху частта вляво:



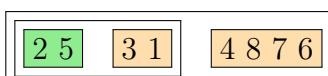
Сега вече големината на входа на текущото рекурсивно викане е едно, и алгоритъмът не прави нищо в това извикване и връща директно управлението на предното извикване, което на свой ред вика алгоритъма върху другия подмасив с големина едно:



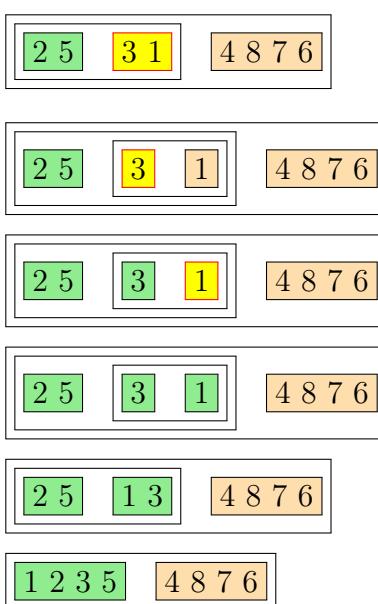
Светлозеленият фон означава, че върху този подмасив вече е достигнато дъното на рекурсията. Отново големината на текущия вход (жълт фон) е едно и алгоритъмът директно връща управлението на горното ниво, бидейки привършил и с двете викания



Сега алгоритъмът слива двета сортирани подмасива с големини единици в един сортиран подмасив с големина две:



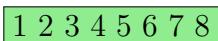
Изчислението продължава така:



Прескачаме няколко стъпки, които съответстват на второто рекурсивно викане върху 4 8 7 6, и разглеждаме масива след приключване на това второ рекурсивно викане:



Сливането на двата сортирани подмасива води до



Сега ще дадем псевдокод са функцията, която осъществява сливането. Нейната коректност е условна: тя работи коректно при условие, че двете половини на входния ѝ масив са сортирани. Символът ∞ означава число, по-голямо от всяко друго число, което може да се срещне. Този символ се нарича *пазач* (на английски, *sentinel*, по терминологията на [15]). Използването му не е задължително, но с него кодът става по-лесен за четене и верификация.

MERGE($A[1, \dots, n]$: int; l, mid, h : цели числа, такива че $1 \leq l < mid < h \leq n$)

```

1 (* подмасивите  $A[l, \dots, mid]$  и  $A[mid + 1, \dots, h]$  са сортирани *)
2  $n_1 \leftarrow mid - l + 1$ 
3  $n_2 \leftarrow h - mid$ 
4 създай  $L[1, \dots, n_1 + 1]$  и  $R[1, \dots, n_2 + 1]$ 
5  $L[1, \dots, n_1] \leftarrow A[l, \dots, mid]$ 
6  $R[1, \dots, n_2] \leftarrow A[mid + 1, \dots, h]$ 
7  $L[n_1 + 1] \leftarrow \infty$ 
8  $R[n_2 + 1] \leftarrow \infty$ 
9  $i \leftarrow 1$ 
10  $j \leftarrow 1$ 
11 for  $k \leftarrow l$  to  $h$ 
12   if  $L[i] \leq R[j]$ 
13      $A[k] \leftarrow L[i]$ 
14      $i++$ 
15   else
16      $A[k] \leftarrow R[j]$ 
17      $j++$ 
```

Лема 29

Да допуснем, че подмасивите $A[l, \dots, mid]$ и $A[mid + 1, \dots, h]$ във входа на MERGE са сортирани. След терминирането на MERGE, масивът $A[l, \dots, h]$ се състои точно от елементите на входните $A[l, \dots, mid]$ и $A[mid + 1, \dots, h]$, но в сортиран вид.

Доказателство: Приемаме за очевидно, че при първото достигане на ред 11, масивите L и R са точни копия на съответно $A[l, \dots, mid]$ и $A[mid + 1, \dots, h]$, плюс един sentinel накрая.

Инвариант 9: MERGE

част i Всеки път, когато изпълнението на MERGE е на ред 11, $A[l, \dots, k - 1]$ съдържа $k - l$ най-малки елемента на L и R , в сортиран вид.

част ii Освен това, $L[i]$ и $R[j]$ са най-малките елементи съответно в L и R , които още не са копирани в A .

База. Когато изпълнението е на ред 11 за пръв път е вярно, че $k = l$. Тогава подмасивът $A[l, \dots, k-1]$ всъщност е $A[l, \dots, l-1]$, тоест е празен. Празният масив е сортиран и съдържа $k - l = 0$ най-малки елемента на L и R, в сортиран вид. Освен това, $L[1]$ и $R[1]$ са най-малките елементи съответно в L и R, които все още не са копирани в A.

Поддръжка. Нека твърдението е в сила при някое достигане на ред 11, което не е последното. Има два алтернативни начина, по които изпълнението може да премине през тялото на цикъла. Ще ги разгледаме и двата. Преди да направим това обаче, ще докажем едно важно помошно твърдение, Лема 30. Забележете, че и L, и R съдържат стойност ∞ , така че трябва да сме сигурни, че двете стойности ∞ не биват сравнявани, понеже сравнение на две ∞ стойности не е дефинирано.

Лема 30

В сравнението на ред 12, не може едновременно $L[i]$ и $R[j]$ да са ∞ .

Доказателство: Да допуснем противното. Съгласно индуктивното предположение, $k - l$ на брой елементи са копирани в A от L и R. А щом изпълнението е на ред 12, то $k \leq h$. Еrgo, броят на копираните елементи е $\leq h - l$.

Очевидно е, че щом на ред 12 сравняваме ∞ с ∞ , всички останали елементи на L и R вече са копирани. В L и R има точно $n_1 + n_2$ елемента, които не са ∞ . Следователно, броят на копираните елементи е поне $n_1 + n_2 = mid - l + 1 + h - mid = h - l + 1 > h - l$. $\frac{1}{2}$

Да разгледаме сравнението на ред 12. Тъй като няма как и $L[i]$, и $R[j]$ да са ∞ , резултатът от сравнението е дефиниран. Първо да попуснем, че $L[i] \leq R[j]$. Очевидно, $L[i] < \infty$. Съгласно **част ii** от индуктивното предположение и допускането, че $L[i] \leq R[j]$, заключаваме, че $L[i]$ е най-малкият елемент както в L, така и в R, който все още не е копиран. Съгласно **част i** на индуктивното предположение, $L[i]$ не е по-малък от никой елемент на $A[l, \dots, k-1]$. Изпълнението отива на ред 13. Забелязваме, че $A[k]$ не е по-малък от нито един елемент на $A[l, \dots, k-1]$ и заключаваме, че $A[l, \dots, k]$ е сортиран и съдържа $k - l + 1 = (k + 1) - l$ на брой най-малки елемента на L и R. Но k бива инкрементирано при следващото достигане на ред 11. Спрямо новата стойност на k е вярно, че $A[l, \dots, k-1]$ съдържа $k - l$ най-малки елемента на L and R, в сортиран вид. И така, **част i** на инвариантата остава в сила.

Сега ще докажем и **част ii** на инвариантата. По допускане, L и R са сортирани. Преди присвояването на ред 13, $L[i]$ беше най-малък елемент от L, който все още не е копиран в A. След това присвояване, $L[i+1]$ е най-малък елемент от L, който все още не е копиран в A. Но променливата i бива инкрементирана при следващото достигане на ред 14. По отношение на новата стойност на i , $L[i]$ е най-малък елемент от L който все още не е копиран в A.

Сега да допуснем, че изпълнението все още е на ред 12 и $L[i] \leq R[j]$, тоест $L[i] > R[j]$. Доказателството е напълно аналогично на току-що направеното.

Терминация. Променливата k съдържа $h + 1$ при последното достигане на ред 11. Заместваме тази стойност в инвариантата и получаваме “подмасивът $A[l, \dots, h]$ съдържа $h - l + 1$ най-малки елемента на L и R, в сортиран вид”. Имайки предвид факта, че L и R съдържат точно елементите на входния $A[l, \dots, h]$, заключаваме, че текущият $A[l, \dots, h]$ съдържа точно елементите на входния $A[l, \dots, h]$, но в сортиран вид. \square

MERGESORT($A[1, \dots, n]$: int; l, h : индекси в A)

```

1   if  $l < h$ 
2        $mid \leftarrow \lfloor \frac{l+h}{2} \rfloor$ 
3       MERGESORT( $A, l, mid$ )

```

4 MERGESORT($A, mid + 1, h$)
 5 MERGE(A, l, mid, h)

Лема 31

Алгоритъм MERGESORT е коректен сортиращ алгоритъм, ако началното извикване е MERGESORT($A, 1, n$).

Доказателство: По индукция по разликата $h - l$ [†]. Смятаме за очевидно, че $h - l$ може да стане най-малко нула, но не и по-малко, и че базата на нашето доказателство е $h - l = 0$.

База. Нека $h - l = 0$, тоест $h = l$. Масивът $A[l, \dots, h]$ е едноелементен. От една страна, едноелементният масив $A[l]$ е тривиално сортиран. От друга страна, MERGESORT не прави нищо, когато $h = l$. Така че масивът е сортиран в края на алгоритъма.

Поддръжка. Допускаме, че MERGESORT сортира коректно подмасивите $A[l, \dots, mid]$ и $A[mid + 1, \dots, h]$ (редове 3 и 4) при всички рекурсивни викания, такива че $h > l$. От Лема 6.1 следва, че в края на работата на текущото рекурсивно извикване, целият $A[l, \dots, h]$ е сортиран.

Терминация. Когато доказваме коректност на рекурсивни алгоритми по индукция, стъпката **Терминация** се отнася до приключването на изпълнението на началното викане. В този случай, началното извикване е MERGESORT($A[1, \dots, n]$). С MERGESORT, тази стъпка е тривиална: просто забелязваме, че алгоритъмът приключи работата си, $A[1, \dots, n]$ е сортиран. \square

Сложността по време на MERGESORT се определя с рекурентното уравнение:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Решението, съгласно втория случай на Мастьър теоремата, е $T(n) \asymp n \lg n$. Сложността по памет е $\Theta(n)$, ако алгоритъмът се имплементира грамотно (имплементация, която буквально следва псевдокода горе би имала линейна сложност по време, но на практика би била твърде бавна заради честото алокиране и dealокиране на памет). MERGESORT е стабилен сортиращ алгоритъм, защото на ред 12 във функцията MERGE има нестрого неравенство. Поради това, ако биват сравнявани два елемента (от L и от R) с равни ключове, елементът от L ще окаже вляво от този от R в окончателната подредба.

Допълнение 22: Бързо намиране на броя на инверсииите в масив.

Определение 28: инверсия в масив

Нека $A[1, \dots, n]$ е масив от цели числа. *Инверсия* в A е всяка двойка индекси $\langle i, j \rangle$, такива че $1 \leq i < j \leq n$ и $A[i] > A[j]$.

Като прост пример да разгледаме $A = [5, 1, 3, 2, 4]$. Инверсии са $\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 1, 5 \rangle$ и $\langle 3, 4 \rangle$.

Очевидно максимален брой инверсии се достига тогава и само тогава, когато елементите на масива са два по два различни и масивът е сортиран обратно. В такъв случай

[†]Забележете, че това не е доказателство с инвариант на цикъла, понеже MERGESORT не е итеративен, а е рекурсивен, алгоритъм.

има точно $\frac{n(n-1)}{2}$ инверсии. Минимален брой инверсии се достига при сортиран масив. В такъв случай инверсиите са точно 0.

Забележете приликата между Определение 28 и Определение 25. Посоката на неравенството в Определение 25 е обратната: там се иска $A[i] < A[j]$, но това е, защото Определение 25 е в контекста на максимални пирамиди (вж. Определение 23). Ако ставаше дума за минимални пирамиди, неравенството в Определение 25 щеше да е $A[i] > A[j]$, също както в Определение 28. Изключвайки несъществената подробност за посоката на неравенството, забелязваме, че в никакъв смисъл Определение 25 е частен случай на Определение 28, тъй като в Определение 25 се иска $A[i]$ да е предшественик на $A[j]$, което е частен случай на изискването $i < j$ в Определение 28. И така, Наблюдение 20 е аналог на Наблюдение 12.

Наблюдение 20

Масивът от цели числа A е сортиран тогава и само тогава, когато няма нито една инверсия. □

Разглеждаме задачата, за даден масив $A[1, \dots, n]$ да се изчисли във време $O(n \lg n)$ броят на инверсиите. Наивния алгоритъм не се признава, понеже е квадратичен.

COUNT-INVERSIONS-NAIVE($A[1, \dots, n]$: int)

```

1  c ← 0
2  for i ← 1 to n - 1
3      for j ← i + 1 to n
4          if A[i] > A[j]
5              c ++
6  return c

```

Ще решим задачата с алгоритъм, изграден по схемата Разделяй-и-Владей. Ако разбирем A на два подмасива L и R , то броят на инверсиите е сумата от

- броят на инверсиите в L ,
- броят на инверсиите в R и
- броят на инверсиите $\langle i, j \rangle$, такива че i е индекс в L и j е индекс в R .

Заради ефикасността на решението искаме L и R да имат колкото се може по-блиズки големини. Алгоритъмът INVERSIONCOUNT, който предлагаме, е малка модификация на MERGESORT. За разлика от MERGESORT той връща стойност.

INVERSIONCOUNT($A[1, \dots, n]$: int; l, h : индекси в A)

```

1  if l ≥ h
2      return 0
3  else
4      mid ← ⌊ \frac{l+h}{2} ⌋
5      x ← INVERSIONCOUNT(A, l, mid)
6      y ← INVERSIONCOUNT(A, mid + 1, h)

```

```

7       $z \leftarrow \text{MERGE-MODIFIED}(A, l, mid, h)$ 
8      return  $x + y + z$ 

```

Имайки предвид горните съображения за броя на инверсиите, коректността на алгоритъма е очевидна, ако MERGE-MODIFIED връща броя на инверсиите $\langle i, j \rangle$, такива че $i \in \{l, \dots, mid\}$ и $j \in \{mid + 1, \dots, h\}$. MERGE-MODIFIED е малка модификация на MERGE на стр. 196.

```

MERGE-MODIFIED( $A[1, \dots, n]$ : int;  $l, mid, h$ :  $1 \leq l < mid < h \leq n$ )
1   (* под масивите  $A[l, \dots, mid]$  и  $A[mid + 1, \dots, h]$  са сортирани *)
2    $n_1 \leftarrow mid - l + 1$ 
3    $n_2 \leftarrow h - mid$ 
4   създай  $L[1, \dots, n_1 + 1]$  и  $R[1, \dots, n_2 + 1]$ 
5    $L[1, \dots, n_1] \leftarrow A[l, \dots, mid]$ 
6    $R[1, \dots, n_2] \leftarrow A[mid + 1, \dots, h]$ 
7    $L[n_1 + 1] \leftarrow \infty$ 
8    $R[n_2 + 1] \leftarrow \infty$ 
9    $i \leftarrow 1$ 
10   $j \leftarrow 1$ 
11   $c \leftarrow 0$ 
12  for  $k \leftarrow l$  to  $h$ 
13    if  $L[i] \leq R[j]$ 
14       $A[k] \leftarrow L[i]$ 
15       $i++$ 
16    else
17       $A[k] \leftarrow R[j]$ 
18       $c \leftarrow c + n_1 - i + 1$ 
19       $j++$ 
20  return  $c$ 

```

Обосновката на коректността на MERGE-MODIFIED е обосновката на MERGE с добавена аргументация, свързана с брояча c . Преди да направим по-формална обосновка ще дадем интуитивно обяснение. Винаги при проверката на ред 13:

- Ако $L[i] \leq R[j]$, то елементите $L[i]$ и $R[j]$ не задават инверсия (очевидно). Очевидно инверсиите между елементи от $L[1, \dots, i-1]$ и елементи от $R[1, \dots, j]$ са толкова, колкото са инверсиите между елементи от $L[1, \dots, i]$ и елементи от $R[1, \dots, j]$. Това, че инкрементираме i на ред 15, без да променяме c , е правилно.
- Ако $L[i] > R[j]$, то елементните $L[i]$ и $R[j]$ задават инверсия (очевидно). Освен това, всеки елемент от $L[i+1, \dots, n_1]$ задава инверсия с $R[j]$, понеже L е сортиран, а релацията \leq е транзитивна. Тогава всеки елемент от $L[i, \dots, n_1]$ задава инверсия с $R[j]$, а това са $n_1 - i + 1$ елемента, което означава и $n_1 - i + 1$ инверсии. Еrgo, инверсиите между елементи от $L[1, \dots, i]$ и $R[1, \dots, j]$ са с $n_1 - i + 1$ повече от инверсиите между елементи от $L[1, \dots, i-1]$ и $R[1, \dots, j]$. Това, че инкрементираме j на ред 19 и добавяме $n_1 - i + 1$ към c на ред 18, е правилно.

6.2 QUICKSORT

QUICKSORT е открит от Sir Charles Antony Richard Hoare [25] в началото на 1960-те (вж. [23], [24], [25]). В много случаи, той е най-бързият сортиращ алгоритъм за реални изчисления—а не в асимптотичния смисъл—и оттам идва и името му. Реалните имплементации са по-сложни от псевдокода, който разглеждаме тук.

QUICKSORT е типичен за схемата Разделяй-и-Владей алгоритъм, макар да е много различен от MERGESORT. Най-общо казано, идеята е да бъде избрана някаква стойност, която се нарича *pivot*, и спрямо нея масивът да бъде пренареден така, че в лявата част (може и да е празна, това зависи от *pivot*) да са само елементи, по-малки или равни на *pivot*, а вдясно от тях, само елементи, по-големи от *pivot*. Каква точно е наредбата в лявата и дясната част няма значение – не се иска при това пренареждане да се постигне сортиран масив, това би било прекалено силно изискване. Сортиране ще се получи чак след края на рекурсивните викания. На този етап искаме просто вляво да са “малките” елементи, а вдясно от тях, “големите”. Това пренареждане е фазата **Разделяй** на алгоритъма. След това алгоритъмът бива викан рекурсивно върху всяка от тези части, ако тя е достатъчно голяма. Това е фазата **Владей**. Фазата **Комбинирай** е празна: след приключването на извикванията, масивът е сортиран. Виждаме една основна разлика между QUICKSORT и MERGESORT. При MERGESORT фазата **Разделяй** е тривиална и същината на алгоритъма е във фазата **Комбинирай**. При QUICKSORT фазата **Разделяй** е същината на алгоритъма, докато фазата **Комбинирай** е празна.

Изборът на *pivot*, както ще видим, е от огромно значение за сложността по време. Не е задължително *pivot* да е елемент от масива. Простите имплементации избират за *pivot* елемент от масива, и то по прост начин, например най-левия или най-десния елемент.

Допълнение 23: За избора на *pivot* в QUICKSORT.

Говорейки не съвсем строго, в идеалния случай стойността на *pivot* е такава, че половината елементи са по-малки от него, а другата половина, по-големи от него. С други думи, ако *pivot* е елемент от масива, то идеалният *pivot* е медианата.

Нещо повече. Както ще видим след малко, много неудачен избор на *pivot*, по-точно серия от много неудачни избори на *pivot*, може да доведе до това, че сложността по време на QUICKSORT да дегенерира до $\Theta(n^2)$, което го прави драстично по-бавен от HEAPSORT и MERGESORT в най-лошия случай. Както също ще видим след малко, такава крайно неудачна серия от избори на *pivot* е малко вероятна и средната сложност по време на QUICKSORT е $\Theta(n \lg n)$.

Възниква въпросът – не може ли да изчисляваме медианата и да избираме нея за *pivot*, с което сложността по време на QUICKSORT да стане $\Theta(n \lg n)$ в най-лошия случай? Вече видяхме в Подсекция 5.4.2, че медианата може да се намери в линейно време. Говорейки **чисто теоретично** и мислейки за сложността по време **само в асимптотичния смисъл**, изчисляването на *pivot* в линейно време няма да увеличи сложността нито в най-лошия, нито в средния случай. Защо тогава не изчисляваме идеалния *pivot* – медианата?

Отговорът е, че QUICKSORT е алгоритъм от голям практически интерес. Той “бие” останалите сортиращи алгоритми, дори бързите HEAPSORT и MERGESORT. Поради това ние се интересуваме не само от асимптотичните оценки на бързодействието му, а и от реалното бързодействие на неговите софтуерни имплементации.

На практика, забавянето, до което би довело намирането на медианата, би направило

QUICKSORT напълно безсмислен. HEAPSORT и MERGESORT са бързи сортиращи алгоритми. Фолклорът казва, че QUICKSORT не е повече от, горе-долу, два пъти по-бърз от кой да е от тях (вижте тази статия на D.Abbhyankar, M.Ingle или този проект на Rashmi Raj). Ако променим QUICKSORT по такъв начин, че да първо да намираме медианата и после pivot да е медианата, той ще стане значително по-бавен от HEAPSORT или MERGESORT и ще престане да бъде от да е практически интерес, въпреки че сложността му в най-лошия случай, в асимптотичния смисъл, ще се подобри от $\Theta(n^2)$ на $\Theta(n \lg n)$.

6.2.1 Имплементация на фазата Разделяй чрез PARTITION–HOARE

Фазата **Разделяй** на QUICKSORT се реализира чрез функция, която е широко известна под името PARTITION. Ние вече видяхме една имплементация на тази функция на стр. 188, но за пълнота на изложението тук допускаме, че виждаме функцията за първи път сега.

Очевидно PARTITION може да се имплементира в линейно време, като това е и добра граница (защото всеки елемент от масива трябва да бъде “прегледан” при това пренареждане). Ключовото наблюдение е, че освен това, тя може да бъде реализирана само с **константна допълнителна памет**, тоест in-place. Известни са няколко имплементации на оригиналната PARTITION, която е открита от Hoare. Ето как изглежда изглежда функцията PARTITION в [16, стр. 154].

PARTITION–HOARE($A[1, 2, \dots, n]$: цели числа; l, h : индекси в A , такива че $l < h$)

```

1  pivot ← A[l]
2  i ← l - 1
3  j ← h + 1
4  while TRUE do
5      do
6          j ← j - 1
7      while A[j] > pivot
8      do
9          i ← i + 1
10     while A[i] < pivot
11     if i < j
12         swap(A[i], A[j])
13     else
14         return j

```

Грубо казано, i и j са индекси, които “вървят” един срещу друг, съответно отляво надясно и отдясно наляво, докато не “открият” елементи от масива X и Y , които са неправилно разположени спрямо избрания pivot в смисъл, че $X \geqslant \text{pivot}$ и $Y \leqslant \text{pivot}$, но X е вляво от Y . Тези елементи се разменят и i и j продължават да “вървят” един срещу друг, докато не се разминат, след което желаното пренареждане е постигнато.

Съгласно предложения псевдокод, ходовете на j и надолу и на i нагоре **спират**, ако достигнат елемент, равен на pivot. От общи съображения човек може да помисли, че ходовете надолу или нагоре трябва да **продължават** през елементи, равни на pivot. На пръв поглед това ще спести излишни размени на еднакви елементи. Тази “оптимизация” обаче не е добра идея, както е показано в [тази онлайн лекция](#), защото върху масив от еднакви елементи тя

води автоматично до квадратичен алгоритъм, а даденият псевдокод работи почти винаги във време $\Theta(n \lg n)$.

По-прецизно казано, по време на работата на алгоритъма, масивът е разбит на три зони (някои, но не всички, от които може да са празни):

- **Зелена зона $A[l, \dots, i]$** . Там са елементи, по-малки или равни на pivot.
- **Сива зона $A[i + 1, \dots, j - 1]$** . Това е неизследваната част от масива.
- **Жълта зона $A[j, \dots, h]$** . Там са елементи, по-големи или равни на pivot.

Изборът на имена за зоните е произволен. Освен това, оцветяваме в червено двете клетки на масива, върху които “се спират” индексите j и i , вървейки съответно отгоре надолу и отдолу нагоре. Тези червени клетки са в някакъв смисъл “нарушители”: клетката, върху която спира j , съдържа елемент, който е по-малък от pivot, а клетката, върху която спира i , съдържа елемент, който е по-голям от pivot. След извършването на swap-а на ред 12, червената клетка, която била е съседна на жълтата зона, преминава в зелената зона и вече я оцветяваме в зелено, и обратното, червената клетка, която е била съседна на зелената зона, преминава в жълтата зона и вече я оцветяваме в жълто.

Когато и двата **do ... while** цикъла приключват, и $A[i] \geqslant \text{pivot}$, и $A[j] \leqslant \text{pivot}$. В случай, че $i < j$, изпълнението отива на ред 12; след swap-а, както казахме, зелената и жълтата зона нарастват с по една клетка, и отново индексите j и i тръгват, съответно надолу и нагоре. Ако $i \neq j$, няма какво повече да се прави.

Ще демонстрираме работата на PARTITION–HOARE с пример. Нека

$A =$	3	1	8	5	7	6	2	4
-------	---	---	---	---	---	---	---	---

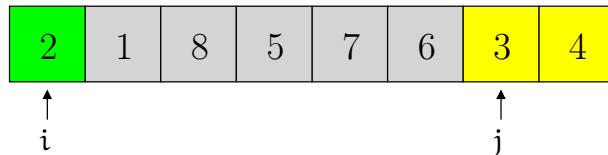
като масивът е индексиран от 1, с други думи, l е 1 и h е 8. В началото pivot е $A[l] = 3$. После i и j “застават” извън границите на масива, така че зелената и жълтата зона са празни:

3	1	8	5	7	6	2	4
↑ i						↑ j	

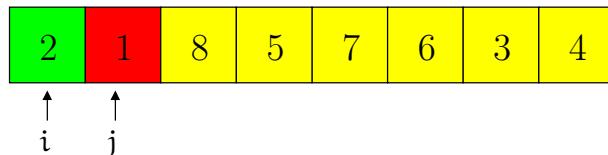
После j “слиза” надолу, първо става 8, но $A[8] \not\leqslant \text{pivot}$, така че j става 7 и изпълнението на първия **do ... while** цикъл (редове 5-6) спира, понеже $A[7] \leqslant \text{pivot}$. Започва изпълнението на втория **do ... while** цикъл (редове 8-9). Индексът i се “качва” нагоре и става 1, но $A[1] = 3$, така че $A[1] \geqslant \text{pivot}$, така че вторият **do ... while** цикъл спира. Ситуацията е следната:

3	1	8	5	7	6	2	4
↑ i						↑ j	

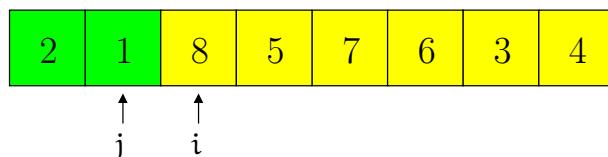
Изпълнението достига ред 11. Понеже $i < j$, изпълняваме размяната, която в случая е размяна на $A[1]$ с $A[7]$ получаваме:



После j “слиза” надолу и става 2, спирайки върху тази стойност:



После i се “качва” нагоре и става 3, спирайки върху тази стойност:



Вече не оцветяваме в червено клетките $A[j]$ и $A[i]$, защото това са елементи, чиято принадлежност към съответно жълтата и зелената зона вече установили.

Изпълнението отново е на ред 11. Сега вече $i \neq j$, така че изпълнението преминава към ред 14, като алгоритъмът връща стойност 2. И наистина, $A[1, \dots, 2]$ се състои от елементи, които са не по-големи от pivot.

Формално доказателство за коректността на PARTITION–HOARE няма да даваме, но следната инвариантна може да се докаже по познатия ни начин по индукция и от нея коректността следва. Няма нужда да се посочва, че елементите на A във всеки момент са точно същите като елементите на входния масив, защото единственото място в алгоритъма, на което се променя масива, е ред 12, а там променянето става чрез размяна на елементи.

Инвариант 10: PARTITION–HOARE

При всяко достигане на ред 4 на PARTITION–HOARE:

$$\begin{aligned} \forall x \in A[l, \dots, i] : x \leq \text{pivot} \\ \forall x \in A[j, \dots, h] : x \geq \text{pivot} \end{aligned}$$

Освен това, ако $i \leq j$:

$$\begin{aligned} \exists x \in A[l, \dots, j-1] : x \leq \text{pivot} \\ \exists x \in A[i+1, \dots, h] : x \geq \text{pivot} \end{aligned}$$

Известни са няколко вариации на PARTITION–HOARE, една от които е следната. Тя е описана в [71, стр. 16]. Тази функция ползва sentinel, който е $-\infty$ и се намира в $A[0]$. Числата, които ще се сортират, са в $A[1, \dots, n]$. В оригинала вътрешните цикли са repeat ... until. За удобство при четене, тук те са сменени с do ... while цикли.

PARTITION–HOARE–ANOTHER($A[0, 1, 2, \dots, n]$: масив; l, h : индекси в A , такива че $l < h$)

1 (* Данните за сортиране са $A[1, \dots, n]$. $A[0]$ е $-\infty$. *)

2 $\text{pivot} \leftarrow A[h]$

```

3   i ← l - 1
4   j ← h
5   while TRUE do
6     do
7       j ← j - 1
8     while A[j] > pivot
9     do
10      i ← i + 1
11    while A[i] < pivot
12    if i < j
13      swap(A[i], A[j])
14    else
15      break
16  swap(A[i], A[h])
17  return i

```

Разлика между PARTITION–HOARE и PARTITION–HOARE–ANOTHER е, че втората връща позицията на `pivot` и сме сигурни, че този елемент си е на мястото и може да правим следващите викания върху подмасиви, в които той не присъства.

6.2.2 Имплементация на фазата Разделяй чрез PARTITION–LOMUTO

Сега само ще покажем алтернативен псевдокод на функция, която пренарежда масива по желания начин. Идеята за този код е на Nico Lomuto, около 1984 г., и е публикувана в статията *Programming Pearls: Little Languages* в списанието Communications of the ACM [8].

PARTITION–LOMUTO($A[1, 2, \dots, n]$): цели числа; l, h : индекси в A , такива че $l < h$)

```

1  pivot ← A[h]
2  pp ← l
3  for i ← l to h - 1
4    if A[i] < pivot
5      swap(A[i], A[pp])
6      pp ← pp + 1
7  swap(A[pp], A[h])
8  return pp

```

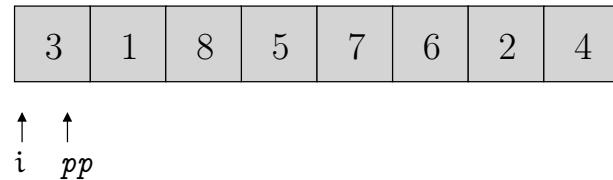
Името на променливата `pp` идва от “pivot position”.

И при PARTITION–LOMUTO можем да дефинираме зелен район (малките елементи), жълт район (големите елементи) и сив район, само че взаимното им разположение е различно, сега те са зелен, жълт и сив, отляво надясно. PARTITION–LOMUTO “търкаля” жълтия район надясно, разменяйки най-левия му елемент с текущия $A[i]$, но само в случай, че текущият $A[i]$ е по-голям или равен на `pp`.

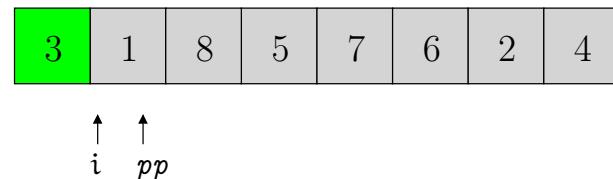
Ето пример за работата на PARTITION–LOMUTO. Нека, както и в предишния пример, A е следният масив:

$A =$	3	1	8	5	7	6	2	4
-------	---	---	---	---	---	---	---	---

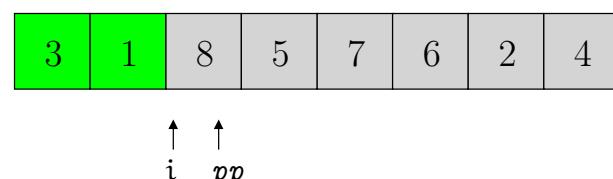
В началото `pivot` е 4. Индексите `i` и `pp` “застават” под първия елемент.



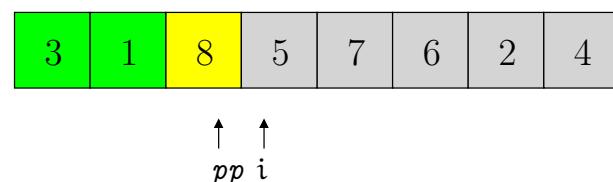
Условието на ред 4 е истина, елементът 3 се разменя със себе си, и после i и pp се оказват под втория елемент.



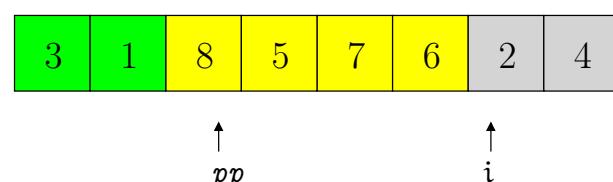
Елементът 1 се разменя със себе си и после i и pp се оказват под третия елемент.



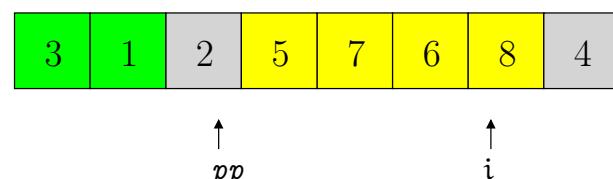
Сега вече $A[i] < pivot$ и тялото на цикъла не се изпълнява. Индексът i вече изпреварва pp и между тях се оформя районът от елементи, по-големи или равни на $pivot$.



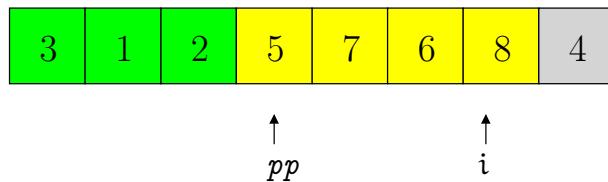
После се срещат няколко елемента, по-големи от $pivot$, които алгоритъмът “прескача”, и се стига до тази ситуация:



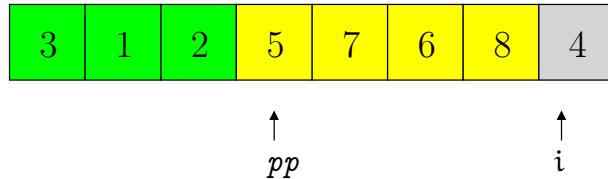
Сега условието на ред 4 е истина. Следва размяна на елементите 8 и 2:



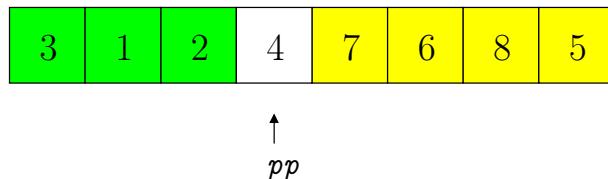
После pp пак се инкрементира:



Индексът i се инкрементира за последен път, жълтият район нараства, след което цикълът не се изпълнява повече:



После се прави размяната на ред 7:



На ред 8 алгоритъмът връща индексът на третия елемент, който е pivot-ът (четвртото). Този елемент си е на мястото. Вляво от него са само по-малки елементи. Вдясно, само по-големи или равни.

Ще докажем формално коректността на PARTITION–LOMUTO.

Лема 32

Ефектът от работата на PARTITION–LOMUTO е следният. В края на алгоритъма, спрямо някаква стойност pp , входният масив е пренареден така, че:

$$\begin{aligned} \forall x \in A[l, \dots, pp - 1] : x < A[pp] \\ \forall x \in A[pp + 1, \dots, h] : x \geq A[pp] \end{aligned}$$

Освен това, алгоритъмът връща именно pp .

Доказателство:

Следното твърдение е инвариантна на for-цикъла (редове 3–6).

Инварианта 11: PARTITION–LOMUTO

При всяко достигане на ред 3 на PARTITION–LOMUTO:

$$\begin{aligned} \forall x \in A[l, \dots, pp - 1] : x < pivot \\ \forall x \in A[pp, \dots, i - 1] : x \geq pivot \end{aligned}$$

База. Когато изпълнението е на ред 3 за първи път, $i = l$ и $pp = l$. Инвариантата е:

$$\begin{aligned} \forall x \in A[l, \dots, l - 1] : x < pivot \\ \forall x \in A[l, \dots, l - 1] : x \geq pivot \end{aligned}$$

И двете твърдения са верни, в празния смисъл. ✓

Поддръжка. Да допуснем, че инвариантата е вярна при някое достигане на ред 3, което не е последното.

Случай I $A[i] < pivot$. Ползваме индуктивното предположение и заключаваме, че:

$$\forall x \in A[l, \dots, pp-1] : x < pivot \wedge A[i] < pivot \quad (6.1)$$

Условието на ред 4 е истина и изпълнението отива на ред 5. Там се извършва размяна на $A[i]$ и $A[pp]$. Имайки предвид тази размяна, (6.1) става:

$$\forall x \in A[l, \dots, pp] : x < pivot \quad (6.2)$$

След инкрементирането на pp на 6, (6.2) става:

$$\forall x \in A[l, \dots, pp-1] : x < pivot \quad (6.3)$$

Виждаме, че първата част на инвариантата се запазва. За да се убедим, че и втората част е вярна, да “върнем лентата” назад до момента, в който изпълнението беше на ред 4. От индуктивното предположение знаем, че

$$\forall x \in A[pp, \dots, i-1] : x \geq pivot \quad (6.4)$$

След размяната на $A[i]$ и $A[pp]$, която става на ред 5, в сила е:

$$\forall x \in A[pp+1, \dots, i] : x \geq pivot \quad (6.5)$$

След инкрементирирането на pp на 6 и неявното инкрементиране на i при следващото достигане на ред 3, спрямо новите стойности на тези две променливи, (6.5) става:

$$\forall x \in A[pp, \dots, i-1] : x \geq pivot \quad (6.6)$$

Случай II $A[i] \geq pivot$. От индуктивното предположение знаем, че:

$$\forall x \in A[l, \dots, pp-1] : x < pivot$$

$$\forall x \in A[pp, \dots, i] : x \geq pivot$$

Тогава:

$$\forall x \in A[l, \dots, pp-1] : x < pivot$$

$$\forall x \in A[pp, \dots, i] : x \geq pivot$$

Но условието на ред 4 е лъжа и изпълнението отива на ред 3, като i се инкрементира неявно. Спрямо новата стойност на i е вярно, че:

$$\forall x \in A[l, \dots, pp-1] : x < pivot$$

$$\forall x \in A[pp, \dots, i-1] : x \geq pivot$$

Терминация. При последното достигане на ред 3 е вярно, че $i = h$. Тогава

$$\forall x \in A[l, \dots, pp-1] : x < pivot$$

$$\forall x \in A[pp, \dots, h-1] : x \geq pivot$$

Доказахме инвариантата и видяхме ефекта от работата на цикъла. Но доказателството за коректността все още не е готово. Изпълнението отива на ред 7, където се разменят $A[pp]$ и $A[h]$, който всъщност е $pivot$. В сила вече е

$$\forall x \in A[l, \dots, pp-1] : x < A[pp]$$

$$\forall x \in A[pp+1, \dots, h] : x \geq A[pp]$$

После алгоритъмът връща pp .

□

6.2.3 Самият QUICKSORT

Псевдокодът на самия QUICKSORT е съвсем кратък (функцията PARTITION е или PARTITION–LOMUTO, или PARTITION–HOARE–ANOTHER):

QUICKSORT($A[1, 2, \dots, n]$: цели числа; l, h : индекси в A)

```

1  if  $l < h$ 
2       $mid \leftarrow \text{PARTITION}(A, l, h)$ 
3      QUICKSORT( $A, l, mid - 1$ )
4      QUICKSORT( $A, mid + 1, h$ )

```

Ако обаче искаме да ползваме PARTITION–HOARE на стр. 202, трябва да направим рекурсивните викания върху подмасиви, които представляват разбиване на оригиналния масив; с други думи, всеки от оригиналните елементи е елемент в някой от тях.

QUICKSORT–ANOTHER($A[1, 2, \dots, n]$: цели числа; l, h : индекси в A)

```

1  (* На стр. 154 в [16] *)
2  if  $l < h$ 
3       $mid \leftarrow \text{PARTITION}(A, l, h)$ 
4      QUICKSORT( $A, l, mid$ )
5      QUICKSORT( $A, mid + 1, h$ )

```

Доказателството за коректност е напълно аналогично на доказателството за коректност на MERGESORT.

QUICKSORT не е стабилен сортиращ алгоритъм. Лесно е да се намери пример, в който PARTITION разменя взаимната наредба на два еднакви елемента. Примерът за PARTITION–LOMUTO може да се различава от примера за PARTITION–HOARE, но и за двата варианта има такива примери.

Забелязваме още една разлика между QUICKSORT и MERGESORT. Фазата **Разделяй** на QUICKSORT връща стойност, а именно индексът, който определя разделянето на масива на две части за рекурсивните викания. Това се налага съвсем естествено: за разлика от MERGESORT, сега не можем да изчислим индекса, определящ разделянето, без да сме прегледали масива.

6.2.4 Сложност по време на QUICKSORT

Сложността на QUICKSORT е изключително чувствителна към изборите на pivot. Казваме “изборите” в множествено число, защото става дума за изборите във всички рекурсивни викания. Ако във всяко отделно викане се оказва, че pivot дели съответния входен (под)масив на две равни части, сложността се описва от рекурентното уравнение

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

което, както вече видяхме при MERGESORT, има решение $T(n) \asymp n \lg n$. Ако обаче всеки избор на pivot е такъв, че този елемент дели масива на подмасив с един елемент (самият той) и друг подмасив с $n - 1$ елемента, сложността се описва от рекурентното уравнение

$$T(n) = T(n - 1) + n$$

С метода с характеристичното уравнение е лесно да се покаже, че това рекурентно уравнение има решение $T(n) \asymp n^2$.

Следователно, в най-лошия случай QUICKSORT е квадратичен сортиращ алгоритъм, с линейна сложност по памет. Ако се задоволяваме само с анализ на сложността в най-лошия случай, това е всичко, което имаме да кажем за сложността на QUICKSORT. На практика обаче QUICKSORT е по-бърз от всеки друг известен сортиращ алгоритъм, говорейки осреднено. Причината е, че лошите избори на pivot са твърде малко вероятни, ако pivot е случаен елемент от масива и големините на елементите от масива са униформно разпределени. Затова, за пръв и последен път в този курс, ще направим анализ на средната сложност на QUICKSORT. Иначе би трябвало да класифицираме най-бързия на практика сортиращ алгоритъм като бавен.

И така, анализът на средната сложност се прави с рекурентно уравнение, което обаче отразява именно средния случай. Както се убедихме, това на какви части бива разделен входът се определя от pivot елемента. Ако pivot е случаен елемент от масива, то всяко разделяне на входа на две части е равновероятно. Отчитаме само **броя на сравненията**. Неговата асимптотика е асимптотиката на самия алгоритъм. Следното рекурентно уравнение моделира казаното дотук:

$$T(n) = \frac{1}{n} \left(\sum_{k=1}^n T(k-1) + T(n-k) \right) + (n-1) \quad (6.7)$$

Променливата k е мястото **по големина**, а не по позиция, на pivot във входа. Казано по друг начин, k е “правилно място” за елемента pivot след разместяването на елементите, направено от функцията PARTITION. И така, pivot може да се окаже най-малък елемент ($k = 1$) или втори по големина ($k = 2$) или … или най-голям елемент ($k = n$). Тогава рекурентно уравнение се чете така: работата на QUICKSORT е сумата от

- сумата по всички възможности за pivot ($k = 1 \dots n$) от сумата от работата върху получени спрямо него двата подмасива. Единият подмасив е с големина $k-1$, а другият е с големина $n-k$, оттам е и изразът $T(k-1) + T(n-k)$. Сумата е умножена с $\frac{1}{n}$, като този множител отразява факта, че всички n възможности за относителната големина на pivot спрямо останалите елементи са равновероятни.
- и $n-1$, което е броят на сравненията за определяне на мястото на pivot. Очевидно pivot трябва да бъде сравнен с всеки друг елемент и това е достатъчно.

Сега ще решим (6.7).

$$\begin{aligned} T(n) &= \frac{1}{n} \left(\sum_{k=1}^n T(k-1) + T(n-k) \right) + (n-1) \\ &= \frac{1}{n} \left(\sum_{k=1}^n T(k-1) + \sum_{k=1}^n T(n-k) \right) + (n-1) \\ &= \frac{1}{n} \left(\underbrace{T(0) + T(1) + \dots + T(n-1)}_{\sum_{k=1}^n T(k-1)} + \underbrace{T(n-1) + T(n-2) + \dots + T(0)}_{\sum_{k=1}^n T(n-k)} \right) + (n-1) \end{aligned} \quad (6.8)$$

Очевидно двете подчертани суми в (6.8) са равни, така че

$$T(n) = \frac{2}{n} (T(0) + T(1) + \dots + T(n-1)) + (n-1) \quad (6.9)$$

Умножаваме двете страни на (6.9) по n и получаваме

$$nT(n) = 2(T(0) + T(1) + \dots + T(n-1)) + n(n-1) \quad (6.10)$$

Но тогава, ако n е достатъчно голямо,

$$(n-1)T(n-1) = 2(T(0) + T(1) + \dots + T(n-2)) + (n-1)(n-2) \quad (6.11)$$

Изваждаме (6.11) от (6.10) и получаваме

$$\begin{aligned} nT(n) - (n-1)T(n-1) &= 2T(n-1) + n(n-1) - (n-1)(n-2) \\ &= 2T(n-1) + 2(n-1) \quad \leftrightarrow \\ nT(n) &= (n+1)T(n-1) + 2(n-1) \end{aligned} \quad (6.12)$$

Делим двете страни на (6.12) на $n(n+1)$ и получаваме

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \quad (6.13)$$

Удобно е да направим следното полагане, за да се освободим от нотация със знаменатели: $\frac{T(n)}{n+1}$ ще бъде наричано $S(n)$. Тогава очевидно $\frac{T(n-1)}{n} = S(n-1)$ и замествайки в (6.13), получаваме следното рекурентно уравнение:

$$S(n) = S(n-1) + \frac{2(n-1)}{n(n+1)} \quad (6.14)$$

То не може да бъде решено чрез метода с характеристичното уравнение, защото нехомогенната част не е от вида, полином на n по константа на n -та степен, така че ще използваме универсалното средство – развиване.

$$\begin{aligned} S(n) &= S(n-1) + \frac{2(n-1)}{n(n+1)} \\ &= S(n-2) + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &= S(n-3) + \frac{2(n-3)}{(n-2)(n-1)} + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &\dots \\ &= S(1) + \frac{2 \cdot 1}{2 \cdot 3} + \frac{2 \cdot 2}{3 \cdot 4} + \frac{2 \cdot 3}{4 \cdot 5} + \dots + \frac{2(n-3)}{(n-2)(n-1)} + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &= S(1) + 2 \sum_{k=2}^n \frac{k-1}{k(k+1)} \end{aligned} \quad (6.15)$$

Да разгледаме (6.15). $S(1)$ е константа, това е някакво начално условие, и асимптотиката на $S(n)$ се определя от $\sum_{k=2}^n \frac{k-1}{k(k+1)}$. Но

$$\sum_{k=2}^n \frac{k-1}{k(k+1)} = \sum_{k=2}^n \frac{k}{k(k+1)} - \sum_{k=2}^n \frac{1}{k(k+1)} = \sum_{k=2}^n \frac{1}{k+1} - \sum_{k=2}^n \frac{1}{k(k+1)}$$

Както знаем от Теорема 16, $\sum_{k=1}^n \frac{1}{k} \asymp \lg n$, откъдето веднага следва, че $\sum_{k=2}^n \frac{1}{k+1} \asymp \lg n$. От друга страна, $\sum_{k=2}^n \frac{1}{k(k+1)}$ е ограничена от константа, защото редът $\sum_{k=1}^{\infty} \frac{1}{k^2}$ е сходящ със

сума $\frac{\pi^2}{6}$. Следва, че $\sum_{k=2}^n \frac{k-1}{k(k+1)} \asymp \lg n$. Тоест, $S(n) \asymp \lg n$. Връщаме се към нотацията $T(n)$ и получаваме, че

$$T(n) \asymp n \lg n$$

Доказвахме, че QUICKSORT има средна сложност по време $\Theta(n \lg n)$. Оттук и средната сложност по памет е логаритмична (а не линейна като в най-лошия случай), защото средната дълбочина на стека на рекурсията е $\Theta(\lg n)$, а на всяко ниво на рекурсията се ползва само константна допълнителна памет от функцията PARTITION.

Лекция 7

Долни граници върху сложност на задачи.

Резюме: Въвеждаме понятието долна граница върху сложност на задача. Доказваме долни граници върху броя на сравненията в две занимателни задачи. Въвеждаме понятието дърво на вземане на решение. Използвайки този модел, доказваме долна граница $\Omega(n \lg n)$ върху сложностите на СОРТИРАНЕ и УНИКАЛНОСТ НА ЕЛЕМЕНТИ, при определени допускания.

7.1 Долни граници на сложността на изчислителни задачи

Определение 29: Долна и горна граница на сложността на задача

Нека Π е произволна изчислителна задача. *Долна граница върху сложността на Π* е всяка функция $\phi(n)$, такава че всеки алгоритъм, който решава Π , работи във време $\Omega(\phi(n))$. *Горна граница върху сложността на Π* е всяка функция $\psi(n)$, такава че поне един алгоритъм, който решава Π , работи във време $O(\psi(n))$.

Забележете разликата между долна и горна граница.

- За да покажем, че дадена функция е горна граница за дадена задача, достатъчно е да покажем един алгоритъм, чиято сложност по време има такава горна граница. Например, от това, че INSERTION SORT работи във време $O(n^2)$ и това, че INSERTION SORT решава задачата СОРТИРАНЕ, следва, че n^2 е асимптотична горна граница за СОРТИРАНЕ.
- От друга страна, за да покажем, че дадена функция е долна граница за дадена задача, не е достатъчно да покажем един алгоритъм, чиято сложност по време има такава долна граница. Например, от това, че MERGESORT работи във време $\Omega(n \lg n)$ и това, че MERGESORT решава задачата СОРТИРАНЕ, по никакъв начин не следва, че $\Omega(n \lg n)$ е долна граница за СОРТИРАНЕ. Доказателство за долна граница е аргумент за всички алгоритми, а те са безброй много, които решават задачата. Всеки известен алгоритъм за СОРТИРАНЕ, който е базиран на директни сравнения, работи във време $\Omega(n \lg n)$, но **от това не следва**, че няма по-бърз алгоритъм, работещ във време, примерно, $\Theta(n \lg \lg n)$ или дори $\Theta(n)$. Както ще видим, такъв наистина няма, но това трябва да се докаже.

Доказателство за долна граница на задача обезсмисля опитите да бъдат конструирани побързи алгоритми. Долната граница е **фундаментално ограничение**, нещо като природна даденост, с която сме длъжни да се съобразяваме, независимо дали ни харесва или не. Има далечна аналогия с фундаменталните ограничения от физиката, например *невъзможността за пътуване със скорост, по-голяма от скоростта на светлината във вакуум* (“Consequently, the speed of light is a natural absolute speed limit.”), *принципът на неопределеността от квантовата механика* или *първият и втори принцип на термодинамиката*.

Доказателствата на нетривиални долни граници върху сложността на задачи са трудни. Какво е тривиална долна граница, ще обясним с пример. За СОРТИРАНЕ, тривиална долна граница е $\Omega(n)$, защото няма как да сортираме масив, без да сме “прегледали” всеки елемент[†]. Доказателството, че $\Omega(n \lg n)$ е долна граница за СОРТИРАНЕ е значително по-трудно и затова казваме, че не е тривиално.

Допълнение 24: Долни граници, задачи и изчислителни модели

Ще разгледаме една тривиална долна граница: $\Omega(n^2)$ за сортирането чрез *транспозиции*. Транспозиция е размяна на местата на два съседни елементи от масива. В нашия псевдокод това се записва като “`swap(A[i], A[i + 1])`”. Сортиране чрез транспозиции е сортиране, при което единствените достъпи до масива за преместване на елементи са транспозиции. Пример за алгоритъм, който сортира чрез транспозиции, е BUBBLESORT [15, стр. 40].

Очевидно е, че в процеса на работата на алгоритъма, всяка извършена транспозиция може да намали броя на инверсиите (Определение 28) с най-много единица. И тъй като инверсиите може да са $\frac{n(n-1)}{2}$ (при обратно сортиран масив от два по два различни елементи), то само “ликвидирането” на инверсиите налага квадратичен брой транспозиции. Масивът не може да е сортиран, ако има инверсии (Наблюдение 20), и от това веднага следва, че всеки алгоритъм, сортиращ само чрез транспозиции, има долна граница за сложността $\Omega(n^2)$.

Тази долна граница обаче не е върху самата изчислителна задача СОРТИРАНЕ, а върху СОРТИРАНЕ с никакво допълнително ограничение: да се ползват само транспозиции. Ограничението не е част от дефиницията на задачата. Да си припомним Разяснение 1: задачата е функция, изобразяваща примери в решения, а алгоритъм за нея е никаква приемливо детайлна реализация на тази функция. Ограничението да се ползват само транспозиции е ограничение не върху задачата, а върху алгоритмите за нея.

Ключовото понятие тук е “изчислителен модел”, което (бегло) споменахме в Допълнение 3. Изчислителният модел определя какво можем да правим. Очевидно, за да ограничим сортирането до такова, което ползва само транспозиции, трябва да наложим ограничения върху нашия изчислителен модел. Изследването на долни граници винаги е в контекста на даден изчислителен модел. В тези лекционни записи изчислителни модели не се разглеждат и само се споменават в допълнения, но внимателния читател ще забележи, че изследването на долни граници иска първо да е дефиниран изчислителен модел.

За ограниченияте цели на лекционните записи можем да избегнем разглеждането на изчислителни модели при долните граници, като кажем, че долната граница не е само

[†]Аналогично, както ще видим в следваща лекция, тривиална долна граница за обхождане на граф, представен със списъци на съседства, е $\Omega(n + m)$, където n е броят на върховете и m е броят на ребрата, поради това, че размерът на списъците на съседства е $\Theta(n + m)$, и няма как да обходим графа, без да сме “прегледали” всеки елемент от тези списъци.

върху дадена изчислителна задача, а върху задачата и някакъв клас алгоритми за нея; примерно, СОРТИРАНЕ и алгоритмите, ползващи само транспозиции.

И още една забележка. Това, че $\Omega(n)$ е долна граница за СОРТИРАНЕ и това, че $\Omega(n \lg n)$ е долна граница за СОРТИРАНЕ (което все още не сме доказали), не са в противоречие. Просто втората долна граница е по-висока, следователно е и по-добра (и по-интересна). Когато става дума за долни граници, интересуваме се от колкото е възможно по-високи такива. Дуално, при горните граници искаме колкото е възможно по-ниски такива: $O(n^9)$ е валидна горна граница за СОРТИРАНЕ, но тя е безинтересна предвид факта, че $O(n^2)$ също е горна граница, а на свой ред тя е безинтересна предвид това, че $O(n \lg n)$ е горна граница.

Доказателствата за долни граници, които ще разгледаме, се класифицират така.

- Елементарни доказателства. Примерно, СОРТИРАНЕ има тривиална долна граница $\Omega(n)$, защото всеки елемент трябва да бъде разгледан поне веднъж. Също така, ТЪРСЕНЕ В НЕСОРТИРАН МАСИВ има тривиална долна граница $\Omega(n)$ – по същата причина.
- Доказателства, използващи дървета за вземане на решения (на английски, *decision trees*). Примери за такива доказателства има в тази лекция: долните граници 2 за ТНЕ BALANCE PUZZLE, 3 за ТНЕ TWELVE-COIN PUZZLE, $\Omega(n \lg n)$ за СОРТИРАНЕ и $\Omega(n \lg n)$ за УНИКАЛНОСТ НА ЕЛЕМЕНТИ използват дървета за вземане на решения.
- Доказателства, използващи редукции. Примерно, долната граница $\Omega(n \lg n)$ за НАЙ-БЛИЗКА ДВОЙКА ЕЛЕМЕНТИ. Редукцията там е, УНИКАЛНОСТ НА ЕЛЕМЕНТИ се редуцира до НАЙ-БЛИЗКА ДВОЙКА ЕЛЕМЕНТИ.
- Аргументиране чрез противник (на английски, *adversary argument*).

7.2 Дървета за вземане на решения

7.2.1 Задачите ТНЕ BALANCE PUZZLE и ТНЕ TWELVE-COIN PUZZLE

Сега ще разгледаме две занимателни задачи, решението на които ще ни даде необходимата интуиция за ключовото понятие “дърво на вземане на решения”.

Задача 4: ТНЕ BALANCE PUZZLE

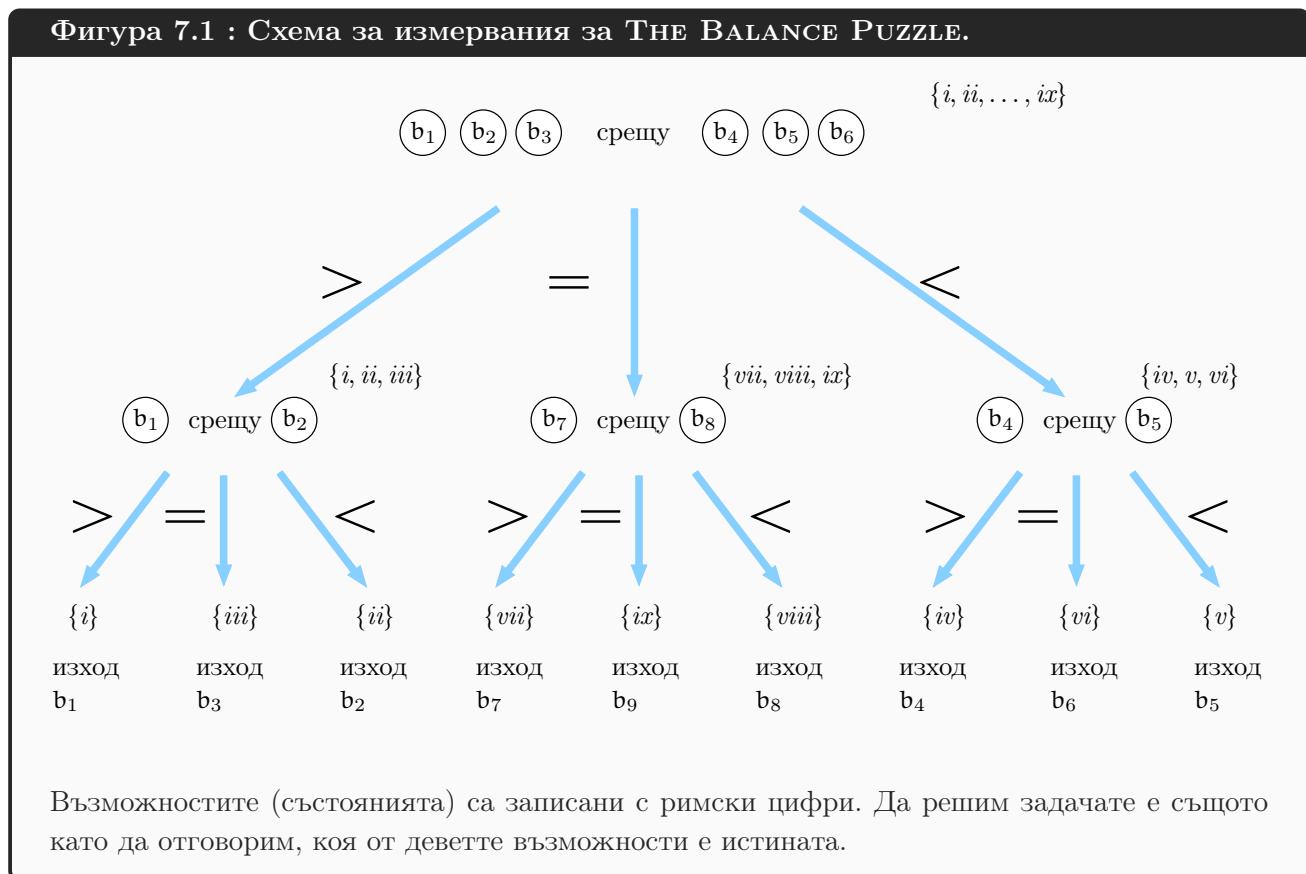
Дадени са 9 номерирани предмета, да речем топки. Осем от тях имат едно и също тегло, а една топка е по-тежка (от коя да е от осемте). Нашата цел е да идентифицираме тежката топка, като използваме везни. Везните нямат стандартни теглилки, така че можем правим единствено измервания от вида: някои топки на едното блюдо срещу други топки на другото блюдо, и да наблюдаваме резултата. Има точно три възможности за резултата: везната да се наклони наляво или везната да се наклони надясно или везната да остане балансирана. Трябва да намерим тежката топка с колкото е възможно по-малко измервания.

Предложете схема за измервания, която ползва не повече от две измервания. Докажете, че две е долна граница за броя на измерванията.

Решение: Нека деветте топки са b_1, \dots, b_9 . С едно използване на везната мерим b_1, b_2 и b_3 срещу b_4, b_5 и b_6 . Има точно три възможни изхода.

- Ако b_1 , b_2 и b_3 са по-тежки от b_4 , b_5 и b_6 , използваме везната втори път с b_1 срещу b_2 . Отново има точно три възможни изхода.
 - ◆ Ако b_1 е по-тежка от b_2 , връщаме “ b_1 е тежката топка”.
 - ◆ Ако b_2 е по-тежка от b_1 , връщаме “ b_2 е тежката топка”.
 - ◆ Ако b_1 и b_2 са еднакво тежки, връщаме “ b_3 е тежката топка”.
- Ако b_4 , b_5 и b_6 са по-тежки от b_1 , b_2 и b_3 , използваме везната втори път с b_4 срещу b_5 . Отново има точно три възможни изхода.
 - ◆ Ако b_4 е по-тежка от b_5 , връщаме “ b_4 е тежката топка”.
 - ◆ Ако b_5 е по-тежка от b_4 , връщаме “ b_5 е тежката топка”.
 - ◆ Ако b_4 и b_5 са еднакво тежки, връщаме “ b_6 е тежката топка”.
- Ако b_1 , b_2 и b_3 , от една страна, и b_4 , b_5 и b_6 , от друга страна, са еднакво тежки, използваме везната втори път с b_7 срещу b_8 . Отново има точно три възможни изхода.
 - ◆ Ако b_7 е по-тежка от b_8 , връщаме “ b_7 е тежката топка”.
 - ◆ Ако b_8 е по-тежка от b_7 , връщаме “ b_8 е тежката топка”.
 - ◆ Ако b_7 и b_8 са еднакво тежки, връщаме “ b_9 е тежката топка”.

Фигура 7.1 илюстрира тази схема.



Възможностите (състоянията) са записани с римски цифри. Да решим задачата е същото като да отговорим, коя от деветте възможности е истината.

Много полезно е да мислим в термините на възможности, или възможни състояния. Поначало има точно девет възможности: или b_1 е тежката топка, или b_2 е тежката топка, и

така нататък, или b_9 е тежката топка. Нека запишем тези девет състояния със, съответно, i, ii, \dots, ix . В началото множеството от състоянията е $\{i, ii, \dots, ix\}$. След всяко измерване, множеството от възможностите намалява. Задачата бива решена когато множеството от възможностите стане едноелементно. На Фигура 7.1 до всеки връх е записано множеството от възможностите, които са консистентни с резултатите от измерванията досега, изключвайки текущото измерване. Тоест, консистентни с резултатите от всички върхове от корена до родителя на настоящия връх.

Доказателството за долна граница е следното. Схемата трябва да е такава, че да различи една възможност от общо девет, използвайки тернарно[†] дърво на вземане на решение. Тернарно дърво с височина 1 има само три листа, така че може да различава само три възможности, което не върши работа за тази задача. Очевидно дървото трябва да има височина поне 2, за да има поне девет листа, с които да различава девет възможности.

Наблюдение 21

Броят на листата на дървото в схемата за измерване е горна граница за броят на състоянията (възможностите), които съответната схема за измерване може да различава.

Забележете, че аргументацията за долната граница няма **нищо общо** с конкретната схема, която изложихме горе. Това, че изведохме долна граница две за броя на измерванията съвсем не означава автоматично, че тази долна граница е достижима. В случая, тя е достижима, но това следва от показаната конкретна схема. Аргументът за долната граница казва само, че с по-малко измервания няма да стане. \square

Задача 5: THE TWELVE-COIN PUZZLE

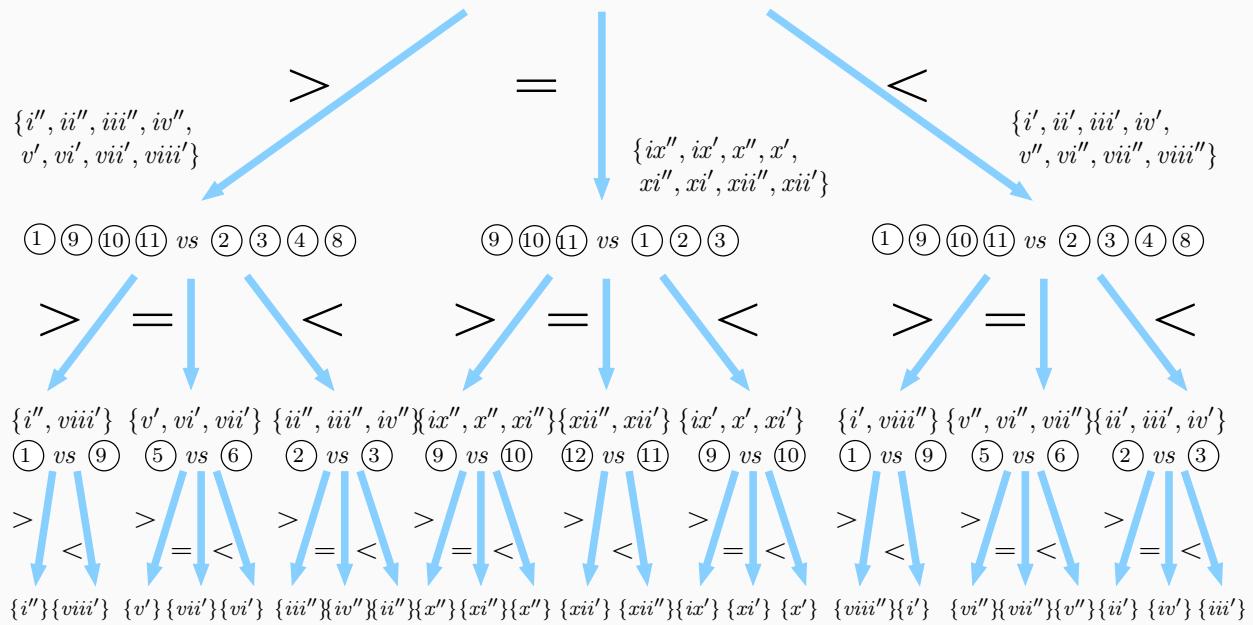
Дадени са 12 номерирани монети. Измежду тях, 11 имат едно и също тегло, а другата—да я наречем *страницата монета*—е или по-лека, или по-тежка. Нашата задача е да идентифицираме страницата монета, както и да определим дали е лека или тежка, използвайки везни като тези в Задача 4. Монетите са различими на външен вид, примерно са номерирани c_1, \dots, c_{12} .

Предложете схема за измервания, с която идентифицираме страницата монета с не повече от три измервания. Докажете, че три е доляна граница за броя на измерванията.

Решение: Тази задача е по-сложна от предишната. Броят на възможностите сега е 24: два пъти броят на монетите. Нека i' означава “ c_1 е тежка”, i'' означава “ c_1 е лека”, ii' означава “ c_2 е тежка”, ii'' означава “ c_2 е лека”, и така нататък, xii' означава “ c_{12} е тежка” и xii'' означава “ c_{12} е лека”. Фигура 7.2 показва схема от измервания с най-много три използвания на везните, която определя коя от 24-те възможности е истината.

[†]“Тернарно” кореново дърво е такова, в което всеки връх има най-много три деца.

Фигура 7.2 : Схема от измервания за THE TWELVE-COIN PUZZLE.

 $\{i'', i', ii'', ii', iii'', iii', iv'', iv', v'', v', vi'', vi', vii'', vii', viii'', viii', ix'', ix', x'', x', xi'', xi', xii'', xii', xiii'\}$
 $(1) (2) (3) (4) \text{ vs } (5) (6) (7) (8)$


Понякога само два от трите изхода са възможни. Всеки връх в дървото е асоцииран с множеството от възможностите, които са консистентни с измерванията до този момент.

Дотук доказвахме, че три измервания са достатъчни.

Сега ще докажем, че три измервания са необходими. Тъй като искаме да различим една възможност от общо 24 възможности и всяко измерване има най-много три възможни изхода, добра граница за броя на измерванията е $\lceil \log_3 24 \rceil = 3$. Защо? Защото тернарно дърво с ≥ 24 листа има височина поне три.

Забележете, че долната граница, която следва от такива съображения (височината на дървото като логаритъм от броя на листата) не винаги е достигима. В случая с THE TWELVE-COIN PUZZLE, долната граница наистина е достигима, както виждаме от схемата на Фигура 7.2. Но същата задача с 13 монети **не може** да бъде решена с най-много три измервания в най-лошия случай. Ето защо:

- Няма смисъл да мерим едно срещу друго две множества от монети с различен брой монети, защото, ако блюдото с повечето монети се наклони надолу, ние не придобиваме **никаква нова информация**, в смисъл, че възможностите не намаляват, а остават същите.
- Имайки предвид предното съображение, за всеки избор на две подмножества с една и съща кардиналност (от множеството от 13-те монети), или тези множества имат кардиналност ≥ 5 , или множеството от останалите монети има кардиналност ≥ 5 .
- Множество с кардиналност ≥ 5 има ≥ 10 възможности за странната монета.
- Няма начин да определим една възможност измежду ≥ 10 възможности, използвайки ≤ 2 измервания, защото при не повече от три измерване, две последователни

измервания могат да различат най-много 9 възможности.

Показахме долна граница четири за броя на измерванията. От друга страна, $\lceil \log_3 26 \rceil = 3$. Няма противоречие между факта, че $\lceil \log_3 26 \rceil = 3$, и факта, че THE THIRTEEN-COIN PUZZLE изиска поне четири измервания в най-лошия случай. Това са различни аргументации за долна граница, като едната аргументация дава долна граница три, а другата, четири. Три си остава валидна долна граница, просто при THE THIRTEEN-COIN PUZZLE долната граница три не е достижима; иначе казано, долната граница три не е точна. \square

7.2.2 Долна граница $\Omega(n \lg n)$ за СОРТИРАНЕ

Долната граница $\Omega(n \lg n)$, която ще докажем, не е в сила не за всеки сортиращ алгоритъм, а само за сортиращите алгоритми, базирани на директни сравнения. В [16, стр. 191] това понятие е обяснено така:

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \dots, a_n \rangle$. That is, given two elements a_i and a_j , we perform one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$ to determine their relative order. We may not inspect the values of the elements or gain order information about them in any other way.

Както ще се убедим от изложението в тази глава, тази дефиниция е ненужно ограничаваща. Ако променим, примерно, INSERTION SORT, слагайки в самото начало код, който преброява елементите от всеки вид (това може да се направи тривиално в $O(n^2)$ време и $O(n)$ памет), и оставяйки същинското сортиране същото, то това ще остане сортиращ алгоритъм, базиран на директни сравнения, въпреки че сега “инспектира” стойностите на елементите. Същественото е не дали алгоритъмът инспектира стойности, а дали пресмята изходната сортирана последователност само въз основа на директни сравнения, или не. Поради това тук правим следното определение.

Определение 30: Сортиращ алгоритъм, базиран на директни сравнения

Сортиращ алгоритъм, базиран на директни сравнения, е всеки сортиращ алгоритъм, който извършва директни сравнения от вида $a_i ? a_j$ върху входните елементи и изходът се определя еднозначно от серията извършени директни сравнения.

Образно казано, всеки алгоритъм за сортиране, базиран на директни сравнения, използва везна, подобна на везните от Подсекция 7.2.1, само че с два, а не три възможни изхода.

Всички разгледани досега сортиращи алгоритми (глави 3, 4 и 6) са базирани на директни сравнения, в контраст с COUNTING SORT от Глава 8, който не е базиран на директни сравнения, а брои колко елемента има от всеки вид, като броят на видовете е ограничен.

Сега ще се убедим с пример, че на всеки сортиращ алгоритъм, базиран на директни сравнения, съответства схема от въпроси и отговори, като въпросите са от вида $a_i ? a_j$ или $a_i \leq a_j$, а отговорите са бинарни. Да си припомним INSERTION SORT.

INSERTION SORT($A[1, \dots, n]$)

```

1  for i ← 2 to n
2      key ← A[i]
3      j ← i - 1
4      while j > 0 and key < A[j] do

```

```

5      A[j + 1] ← A[j]
6      j ← j - 1
7      A[j + 1] ← key

```

Да разгледаме работата на INSERTION SORT върху вход с големина три. Нека входът е

a_1, a_2, a_3

С малки букви, примерно “ a_1 ”, означаваме елементите от входа. Те са неизменни, в смисъл че който и да е от тях, да кажем a_1 , е едно и също нещо във всеки момент от работата на алгоритъма. С главни букви, примерно “ $A[1]$ ”, означаваме елементите от текущия масив A . Те може да се менят по време на робатата! В началото е вярно, че $A = [a_1, a_2, a_3]$, но след няколко стъпки от изпълнението може да имаме $A = [a_2, a_3, a_1]$. Да допуснем, че елементите от входа са уникални. Точно едно от следните е в сила:

$$a_1 < a_2 < a_3 \quad (7.1)$$

$$a_1 < a_3 < a_2 \quad (7.2)$$

$$a_2 < a_1 < a_3 \quad (7.3)$$

$$a_2 < a_3 < a_1 \quad (7.4)$$

$$a_3 < a_1 < a_2 \quad (7.5)$$

$$a_3 < a_2 < a_1 \quad (7.6)$$

Ще наричаме (7.1), …, (7.6), *пермутациите*[†]. Ключовото наблюдение, без което няма истинско разбиране на понятието “дърво за вземане на решение при сортиране, базирано на директни сравнения”, е следното.

Наблюдение 22

Да сортираме входа с директни сравнения в някакъв смисъл е същото като да определим, само в резултат на изходите от сравненията, коя от пермутациите е истината.

Сравненията стават само на ред 4 в $\text{key} < A[j]$. Първото сравнение е $a_2 < a_1$.

Случай I Ако $a_2 < a_1$ е TRUE, то точно тези три пермутации (от началните шест) са възможни:

$$a_2 < a_1 < a_3$$

$$a_2 < a_3 < a_1$$

$$a_3 < a_2 < a_1$$

В този случай алгоритъмът променя A по време на текущото изпълнение на **for**-цикъла, така че A става $[a_2, a_1, a_3]$. Други сравнения не се правят по време на текущото изпълнение на **for**-цикъла и започва ново негово изпълнение. Следващото сравнение е $a_3 < a_1$. Но първо да разгледаме алтернативния случай на **Случай I**.

Случай II Ако $a_2 < a_1$ е FALSE, то точно тези три пермутации са възможни:

$$a_1 < a_2 < a_3$$

$$a_1 < a_3 < a_2$$

$$a_3 < a_1 < a_2$$

[†]Всяко от (7.1), …, (7.6) е логически израз, тъй като е конюнкция от две неравенства, така че, стриктно говорейки, не е пермутация. “Пермутация” е нещо, която няма логическа интерпретация.

В този случай алгоритъмът не променя A по време на текущото изпълнение на **for**-цикъла, така че A остава $[a_1, a_2, a_3]$. Други сравнения не се правят по време на текущото изпълнение на **for**-цикъла и започва ново негово изпълнение.

Случай I.1 Връщаме се към **Случай I**. Ако $a_3 < a_1$ е TRUE, точно тези две перmutации остават възможни:

$$a_2 < a_3 < a_1$$

$$a_3 < a_2 < a_1$$

A става $[a_2, a_1, a_1]$, като стойността a_3 се съхранява в променливата `key`. Следващото сравнение е $a_3 < a_2$.

Случай I.1.a Ако $a_3 < a_2$ е TRUE, остава една единствена възможна перmutация:

$$a_3 < a_2 < a_1$$

while-цикълът се изпълнява още веднъж и A става $[a_2, a_2, a_1]$. Тогава a_3 се записва на първа позиция в A (line 7) и A става $[a_3, a_2, a_1]$. После алгоритъмът приключва работата си.

I.1.b Ако $a_3 < a_2$ е FALSE, остава една единствена възможна перmutация:

$$a_2 < a_3 < a_1$$

while-цикълът не се изпълнява повече. a_3 се записва на втора позиция в A (line 7) и A става $[a_2, a_3, a_1]$.

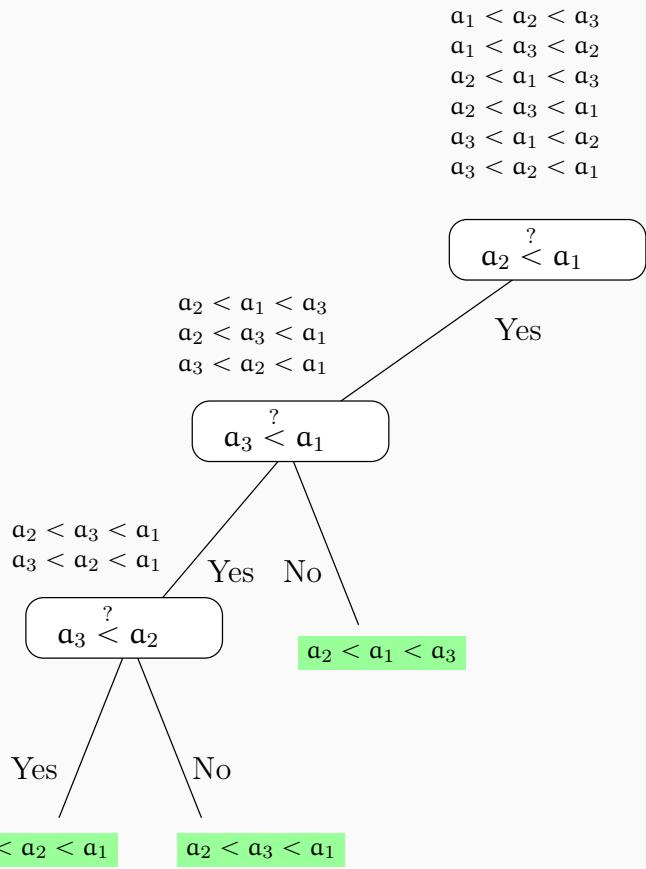
Случай I.2 Ако $a_3 < a_1$ е FALSE, остава една единствена възможна перmutация:

$$a_2 < a_1 < a_3$$

Алгоритъмът приключва работата си. A остава $[a_2, a_1, a_3]$.

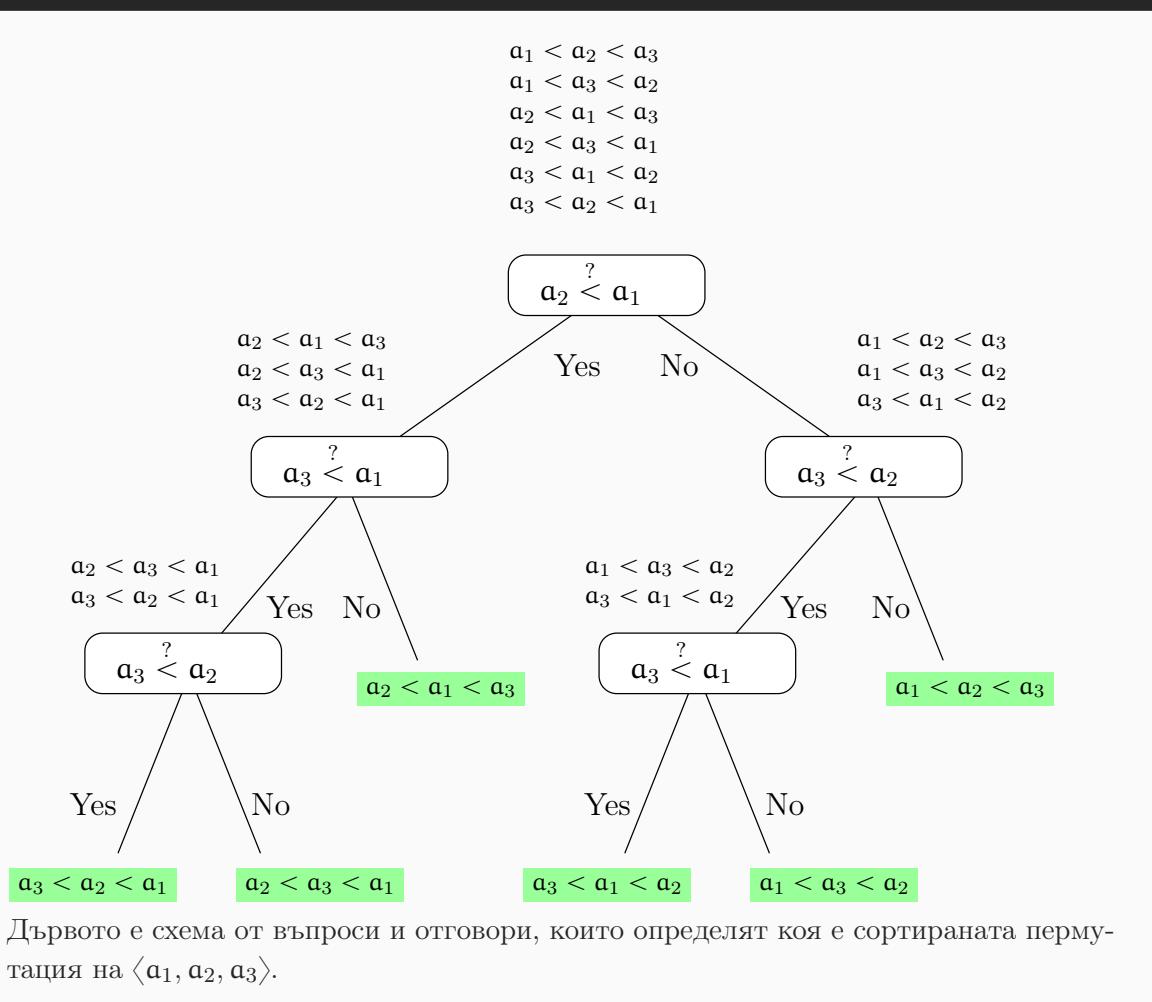
Четирите подслучай на **Случай I** може да бъдат представени чрез дърводидната структура на Фигура 7.3. Има два типа върхове—върхове-сравнения, съответстващи на въпроси от вида $a_i ? a_j$, и листа (в зелено). Над всеки връх-сравнение записваме множеството от перmutациите, които са консистентни с отговорите до този момент (преди това сравнение). Естествено, над корена са записани всички шест перmutации, които са възможни поначало. Листата отговарят на възможните изходи от алгоритъма.

Фигура 7.3 : Половината от дървото на сравненията.



Изобразени са случаите, в които $a_2 < a_1$.

Ако направим аналогичен анализ на подслучаите на **Случай II** ще получим структурата, показана на Фигура 7.4. Дървото T е двоично, понеже има два възможни отговора при всяко сравнение.

Фигура 7.4 : Дървото на сравненията на INSERTION SORT върху $\langle a_1, a_2, a_3 \rangle$.

Дървото е схема от въпроси и отговори, които определят коя е сортираната пермутация на $\langle a_1, a_2, a_3 \rangle$.

Изведохме дървовидната структура чрез подробно симулиране на работата на INSERTION SORT върху всички възможни[†] входове с размер три. Такава дървовидна схема от въпроси и отговори се нарича *дърво на вземане на решения*, на английски *decision tree*. Определението в [15, стр. 192] е:

A decision tree is a full binary tree[‡] that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored.

Дърветата на вземане на решения няма нужда да са двоични и не са ограничени до задачата СОРТИРАНЕ. Както видяхме в миналата секция, известните задачи THE BALANCE PUZZLE и THE TWELVE-COIN PUZZLE (вж. Задача 4 на стр. 215 и Задача 5 на стр. 217) са податливи на анализ чрез дървета на вземане на решение, само че тернарни.

При всяко от тези:

- THE BALANCE PUZZLE,
- THE TWELVE-COIN PUZZLE,
- INSERTION SORT върху вход с размер три

[†]Като относителни големини.

[‡]Full binary tree е двоично дърво, в което всеки вътрешен връх има точно две деца. Binary tree е дърво, в което всеки вътрешен връх има най-много две деца.

дървото на вземане на решения е едно единствено, защото става дума за една единствена големина на входа. Очевидно е, че INSERTION SORT по принцип има не едно такова дърво, а безкрайно множество от дървета, по едно за всеки размер на входа. THE BALANCE PUZZLE и THE TWELVE-COIN PUZZLE имат очевидни обобщения за вход с произволна големина. Тези обобщения също имат безкрайни множества от дървета на вземане на решение, по едно за всяка големина на входа.

Наблюдение 23

Нека A е произволен сортиращ алгоритъм, базиран на директни сравнения. A има не едно единствено дърво на вземане на решения, а **безкрайно множество** такива дървета, **по едно за всеки размер на входа**. Ако кажем просто “дървото на вземане на решения”, имаме предвид общия елемент на това множество.

Допълнение 25: Decision trees са изчислителни модели

При разсъжденията за INSERTION SORT ние изведохме дървото от алгоритъма. Но може да мислим за дървото като първичен обект. Дървото, в никакъв смисъл, сортира, само че не чрез местене на елементи, а определяйки коя е сортираната пермутация. И така, може да мислим, че дървото е първичният обект, а алгоритъмът е вторичен, като алгоритъмът мести елементи по такъв начин, че в правилният момент “на везната” да бъдат поставени правилните елементи. “Везната” в случая с INSERTION SORT е $\text{key} < A[j]$.

Дърветата за вземане на решения са изчислителни модели, защото те смятат. Ползвайки дърво за вземане на решения, можем да пресметнем коя е тежката топка или коя е сортираната пермутация. Когато разглеждаме дърветата за вземане на решения като изчислителен модел, ние ги разглеждаме именно като първични обекти.

Казваме, че дървото на вземане на решение е *неуниформен изчислителен модел*, на английски *nonuniform computational model*, защото за всяка големина на входа изчисляващото дърво е различно. Друг пример за неуниформен изчислителен модел са логическите схеми (на английски, *logic circuits*; вижте [55, стр. 16]). В контраст на това, машината с произволен достъп [55, стр. 19] е униформен изчислителен модел, защото за всяка големина на входа тази машината е една и съща. Машината на Тюринг също е униформен изчислителен модел.

Сега ще покажем, че дърветата на вземане на решение ни дават механизъм за доказване на долни граници на задачи. Първото ще разгледаме конкретното дърво T , съответстващо на INSERTION SORT с размер на входа три (Фигура 7.4). Забелязваме, че има листа с дълбочина две и листа с дълбочина три. Това означава, че върху някои входове определяме сортираната пермутация с два въпроса, а върху други, с три. Височината на дървото е максималният брой въпроси, които са ни необходими **в най-лошия случай**, по отношение на конкретната схема. Естествено, има много други възможни схеми, отговарящи на други сортиращи алгоритми. Нека читателят сам си направи дървото-схема, отговарящо на SELECTION SORT с размер на входа три, и ще се убеди, че то е различно от дървото-схема на INSERTION SORT.

Но, каквото и да е дървото-схема, за вход с размер три, то трябва да има височина поне три. Защо? Допуснете, че има дърво T' на вземане на решения за сортиране на три елемента, което има височина най-много две. Тоест, схема от въпроси δ' , която с най-много два въпроса определя коя е сортираната пермутация. Но T' също е двоично дърво, защото всеки въпрос

има два възможни отговора. Тривиално наблюдение е, че T' не може да има повече от четири листа, откъдето веднага следва, че δ' не може да различи повече от четири възможности. С други думи, δ' не може да различи повече от четири пермутации. Само че общо пермутациите са шест, следователно δ' не е коректна схема от въпроси. Полученото противоречие показва, че такова дърво T' не съществува.

Забележете, че този анализ не разглежда конкретните особености на хипотетичното дърво T' . Със сигурност бихме могли да разгледаме едно след друго всички дървета за вземане на решение за вход с размер три и да установим, че всяко от тях има височина поне три. Само че това би било твърде отегчително, и освен това този конструктивен подход не се мащабира за произволен размер на входа. Докато не-конструктивният анализ се мащабира за **произволен** размер на входа. В основата на неконструктивния анализ е следното просто наблюдение: ако листата на хипотетичното дърво са по-малко от възможностите пермутации, то, съгласно по принципа на Дирихле, поне едно листо е асоциирано с повече от една пермутация, от което следва, че това дърво не сортира.

Ето как се мащабира не-конструктивният анализ за вход с размер n . За всеки вход с размер n има схема от въпроси, която установява сортираната пермутация, като въпросите са от вида $a_i < a_j$? Знаем, че такива схеми има, защото съществуват коректни сортиращи алгоритми, базирани на директни сравнения, а на всеки сортиращ алгоритъм съответства безкрайно множество от дървета на вземане на решения, по едно за всяка големина на входа. Размерите на тези дървета растат експлозивно с нарастването на n , защото броят на листата е $\geq n!$. Следователно, не е практично да се опитваме да рисуваме такова дърво за $n > 3$. Но това не е важно. Важното е, че ако разгледаме множеството \mathfrak{T}_n от всички възможни дървета на вземане на решение за размер на входа n , за произволно n , **дървото с най-малка височина от \mathfrak{T}_n задава долна граница за броя на сравненията за задачата SORTING върху вход с големина n** . Тогава долна граница за височината на кое да е дърво от \mathfrak{T}_n е долна граница за изчислителната задача СОРТИРАНЕ, базирано на директни сравнения.

И така, всяко дърво от \mathfrak{T}_n трябва да има поне $n!$ листа, защото толкова са възможните пермутации. Тъй като дървото е бинарно, височината му е поне логаритмична при основа 2 в броя на листата. Тогава всяко дърво от \mathfrak{T}_n има височина $\geq \log_2 n!$. Както вече показвахме в Теорема 14 в $\log_2 n! = \Theta(n \lg n)$. Веднага следва, че всяко дърво от \mathfrak{T}_n има височина $\Omega(n \lg n)$, а оттук следва долната граница $\Omega(n \lg n)$ за СОРТИРАНЕ, базирано на директни сравнения.

7.2.3 Долна граница $\Omega(n \lg n)$ за УНИКАЛНОСТ НА ЕЛЕМЕНТИ

Да си припомним тази изчислителна задача, въведена в Подсекция 3.2.3 на стр. 84.

Изч. Задача: ELEMENT UNIQUENESS

пример: Числа a_1, a_2, \dots, a_n .

въпрос: Дали всички числа са уникални?

Разглеждаме задачата само като базирана на директни сравнения. Алгоритъм за задачата ELEMENT UNIQUENESS е базиран на директни сравнения, ако достъпът до елементите става само по един начин: чрез изпълнения на операция на сравнения $a_i < a_j$? с бинарен изход. Изходите от тези сравнения определят и работата на алгоритъма. Наивният, очевиден алгоритъм е сравняването на всички ненаредени двойки от входа. Сложността на този

подход очевидно е $\Theta(n^2)$. По-изтънчен подход е посочен в Подсекция 3.2.3: първо сортираме входа във време $\Theta(n \lg n)$ и след това с едно “премитане” (на английски, *linear sweep*) установяваме във време $\Theta(n)$ дали няма или има повторяния: ако има повторящи се елементи, те задължително образуват непрекъсната последователност. Сложността по време е $\Theta(n \lg n) + \Theta(n) = \Theta(n \lg n)$.

COMPARISON-BASED ELEMENT UNIQUENESS($A[1, \dots, n]$: array of integers)

```

1 Sort(A)
2 are_unique ← TRUE
3 for i ← 2 to n
4     if A[i - 1] = A[i]
5         are_unique ← FALSE
6 return are_unique

```

Ще докажем, че всеки алгоритъм за ELEMENT UNIQUENESS, базиран на директни сравнения, има сложност $\Omega(n \lg n)$. Ще направим доказателство с дърво на вземане на решения. Както вече посочихме при сортирането, всеки алгоритъм, базиран на директни сравнения, има безкрайно множество от дървета за вземане на решения, асоциирани с него, по едно за всяка големина на входа. Ние ще разгледаме общия елемент на това множество, който е дървото, асоциирано с вход n . За да докажем желаното твърдение, достатъчно е да докажем, че листата на дървото са $\geq n!$ на брой и после да приложим същите разсъждения, които направихме на предната страница за сортирането, базирано на директни сравнения.

За да докажем, че дървото има $\geq n!$ листа, достатъчно е да покажем, че то различава[†] всички перmutации – в смисъл, че всяко листо е асоциирано с не повече от една пермутация. По отношение на сортирането, това е тривиално: ако дървото има листо, асоциирано с повече от една пермутация, съответният алгоритъм не е коректен сортиращ алгоритъм. По отношение на уникалността на елементите, това не е толкова очевидно. Доказателството надолу следва доказателството от [61].

Теорема 21

Нека X е произволен алгоритъм за ELEMENT UNIQUENESS, базиран на директни сравнения. Нека неговото дърво за вземане на решения за вход с големина n е T . Всяко листо на T е асоциирано с най-много една пермутация на входните елементи.

Доказателство:

Можем да допуснем, че входните елементи са уникални, защото правим анализ на най-лошия случай, а това това ограничение се мащабира: за всяко n има вход с големина n , състоящ се от уникални елементи, и щом X е коректен алгоритъм, той трябва да работи коректно и върху този вход.

[†]Трябва да поясним в какъв смисъл алгоритъм за уникалност на елементи различава пермутации. При сортирането е ясно, защото резултатът от процеса на сортиране е пермутация, точно една на брой. При уникалността обаче резултатът е булева стойност, а не пермутация, така че може би не е очевидно в какъв смисъл алгоритъмът различава пермутации. Но припомните си, че алгоритъмът за уникалност е базиран на **директни сравнения**. В самото начало всички пермутации са възможни в смисъл, че не знаем кой елемент е най-малък и т. н., но след като започнем да правим сравнения, след всяко сравнение, множеството от пермутациите, консистентни с изходите от сравненията до момента, намалява. И така, пермутациите, асоциирани с даден връх на дървото са точно тези, които са консистентни с изходите от сравненията до момента.

Да допуснем, че най-малката стойност във входа е b_1 , втората най-малка е b_2 , и така нататък, най-голямата стойност е b_n . С други думи,

$$b_1 < b_2 < \dots < b_n$$

Лема 33

В текущия контекст, за всяко k , такова че $1 \leq k < n$, X сравнява b_k с b_{k+1} в даден момент.

Доказателство: Да допуснем противното. Тогава съществува вход $A[1, \dots, n]$ от уникални елементи, такива че X не сравнява k -ия с $(k + 1)$ -ия по големина елемент. Щом X е коректен сортиращ алгоритъм и елементите на A са уникални, $X(A[1, \dots, n])$ връща TRUE. Нека $A[i]$ е елементът от входа със стойност b_k и $A[j]$ е елементът от входа със стойност b_{k+1} . Да преобразуваме $A[1, \dots, n]$ в $A'[1, \dots, n]$, присвоявайки b_{k+1} на $A[i]$; с други думи, правим $A[i] \leftarrow A[j]$ и наричаме този масив A' . После да изпълним $X(A')$.

От една страна, A' има повтарящи се елементи и X е коректен алгоритъм, следователно $X(A')$ трябва е FALSE. От друга страна, работата на X върху A' е напълно идентична на работата му върху A , понеже по допускане X никога не сравнява $A[i]$ с $A[j]$, а резултатите от всички други сравнения са едни и същи и при вход A , и при вход A' . Но тогава $X(A[1, \dots, n])$ и $X(A'[1, \dots, n])$ връщат една и съща стойност. \diamond

Доказвахме, че X задължително сравнява всяка ненаредена двойка (има $n - 1$ такива) от елементи **със съседни стойности**.

Лема 34

Нека $a = \langle a_1, a_2, \dots, a_n \rangle$ е редица от числа, две по две различни, и нека $a' = \langle a'_1, a'_2, \dots, a'_n \rangle$ е различна редица от същите числа. Нека b_1 е най-малката стойност между числата, нека b_2 е втората най-малка, и така нататък. Тогава съществува наредена двойка $(k, k + 1)$ за някое $k \in \{1, \dots, n - 1\}$, такава че, ако a_i е елементът със стойност b_k и a_j е елементът със стойност b_{k+1} ^a, то $a'_i > a'_j$.

^aТова влече $a_i < a_j$.

Доказателство: Твърди се, че има съседни по големина елементи в a —забележете, не на съседни позиции в редицата a , а съседни по големина измежду всички числа от a —да речем a_i и a_j , като $a_i < a_j$, такива че в a' на същите тези позиции i и j има такива елементи, че този на позиция i е по-голям от този на позиция j .

Ето пример. Нека $n = 8$. Нека

$$a = \langle 4, 11, 5, 2, 14, 18, 22, 7 \rangle$$

Нека

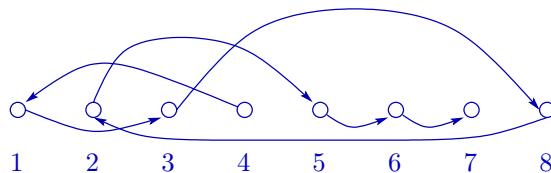
$$a' = \langle 7, 4, 11, 5, 22, 18, 2, 14 \rangle$$

Има ли в a двойка елементи a_i и a_j с желаните свойства? Да, разбира се, например $7 = a_8$ и $11 = a_2$. Съответните им големини са 4 и 5, защото 7 е четвъртият по големина в a , а 11 е петият. Очевидно $a_8 < a_2$. Забележете, че $a'_8 = 14$ и $a'_2 = 4$; ерго, в a' отношението между осмият и вторият елемент е обратното, а именно $a'_8 > a'_2$.

Сега ще извършим доказателството чрез допускане на противното. Противното е: за всеки две позиции в \mathbf{a} , съдържащи елементи със съседни по големина стойности, в \mathbf{a}' посоката на неравенството между елементите на тези позиции е същата. Но тогава следва, че $\mathbf{a} = \mathbf{a}'$. Защо? Всяка редица от n различни по двойки числа може да се опише еднозначно чрез:

- един ориентиран граф-път, чиито върхове са позициите и който има ребро от позиция i към позиция j тогава и само тогава, когато елементът на позиция i е непосредствен предшественик по големина на елемента на позиция j
- и множеството от самите числа.

Примерно, въпросният граф на редицата $\langle 4, 11, 5, 2, 14, 18, 22, 7 \rangle$ е



Ако допуснем, че твърдението, което доказваме, не е истина, директно извеждаме, че две различни редици от едни и същи числа имат един и същи граф-път. Но тогава те са една и съща редица. \square

Доказахме Лема 33 и Лема 34. Сега ще докажем теоремата. Да допуснем, че дървото на вземане на решения T на алгоритъм за уникалност на елементи X (базиран на директни сравнения) асоциира някое листо u с поне две пермутации. Съгласно Лема 34, съществуват съседни по големина елементи в първата пермутация, такива че на техните съответни позиции i и j във втората пермутация има елементи, неравенството между които е в обратната посока (спрямо неравенството между елементите на позиции i и j в първата).

Да обясним това по-подробно. Нека числата във входа са a_1, a_2, \dots, a_n . Теоремата твърди, че всяко листо е асоциирано с не повече от една тяхна пермутация. Допускаме противното: съществува листо u и поне две различни пермутации на входните числа, да кажем $\mathbf{a}' = \langle a'_1, a'_2, \dots, a'_n \rangle$ и $\mathbf{a}'' = \langle a''_1, a''_2, \dots, a''_n \rangle$, които са асоциирани с u . Това означава, че и \mathbf{a}' , и \mathbf{a}'' са консистентни с въпросите и отговорите по пътя от корена до u .

Нека b_1, b_2, \dots, b_n са стойностите от входа в нарастваща големина $b_1 < b_2 < \dots < b_n$. Лема 34 казва, че съществуват b_k и b_{k+1} , такива че, ако те са на позиции съответно i и j в \mathbf{a}' (което влече $a'_i < a'_j$), то в \mathbf{a}'' , елементите на позиции i и j (съответно a''_i и a''_j) са такива, че $a''_i > a''_j$.

Сега си припомняме Лема 33. Тя казва, че X трябва да направи сравнението $b_k ? b_{k+1}$. Само че X не задава въпроси за съседни стойности b_k и b_{k+1} ![†] X задава въпроси от вида $a_p ? a_q$, а Лема 33 гарантира, че рано или късно, a_i и a_j , които имат стойности съответно b_k и b_{k+1} , се оказват “на везната”.

Ключовото наблюдение е следното. По отношение на въпроса $a_i ? a_j$, \mathbf{a}' е консистентна с Да отговора, докато \mathbf{a}'' е консистентна с НЕ отговора. Тогава за вътрешния връх v , в който се задава въпроса $a_i ? a_j$, едното дете на v , да го наречем x , е асоциирано с \mathbf{a}' , а другото дете, да го наречем y , е асоциирано с \mathbf{a}'' . Очевидно u не може да е връх (листо) и в $T[x]$, и в $T[y]$. \square

[†] Ако знаем, че дадени елементи са съответно b_k и b_{k+1} , то е напълно безсмислено да ги сравняваме; с други думи, отговорът на въпроса $b_k ? b_{k+1}$ е предварително известен.

След като сме доказали Теорема 21, долната граница $\Omega(n \lg n)$ за задачата УНИКАЛНОСТ НА ЕЛЕМЕНТИ, базирана на директни сравнения, следва от същите съображения като тази на СОРТИРАНЕ.

7.2.4 Долна граница $\Omega(\lg n)$ за ТЪРСЕНЕ

Вече разглеждахме алгоритми за двоично търсене на стр. 83, които очевидно имат сложност по време $\Theta(\lg n)$ в най-лошия случай.

Сега твърдим, че $\Omega(\lg n)$ е асимптотична долната граница за ТЪРСЕНЕ. Забележете, че тук не говорим за двоично търсене, а за **всякакво** търсене! Ако търсенето е базирано на директни сравнения, всеки алгоритъм работи във време $\Omega(\lg n)$. Причината е много проста: има $\Theta(n)$ различни състояния по отношение на даден вход. Нека входът е масив $A[1, \dots, n]$ (сортиран или не, няма значение) и търсената стойност е key. Да допуснем, че елементите от A са два по два различни. Това не е ограничение на общността, понеже, ако получим долната граница при това допускане, в общия случай долната граница не може да е по-ниска.

Всеки алгоритъм за ТЪРСЕНЕ, базиран на директни сравнения, трябва да може да различава (поне) следните $n + 1$ състояния:

състояние 1: $\text{key} = A[1]$,

състояние 2: $\text{key} = A[2]$,

...

състояние n : $\text{key} = A[n]$,

състояние $n + 1$: $\text{key} \notin A$.

в смисъл, че това са съществено различни ситуации и дървото на вземането на решение трябва да е такова, че всяко листо да е асоциирано с не повече от едно от тези състояния. В началото на алгоритъма всички тези състояния (ситуации, ако предпочитате) са възможни, а след всяко извъшване на сравнение между key и елемент на масива, някои от тях стават невъзможни[†]. Ако в края на алгоритъма останат две или повече възможни състояния, то алгоритъмът “не си е свършил работата” и отговорът му, какъвто и да е, е невалиден. Еrgo, всеки коректен алгоритъм има такова дърво, че всяко листо е асоциирано с най-много едно от тези състояния.

Но това са $\Theta(n)$ състояния, което дава долната граница $\Omega(n)$ за броя на листата, което дава долната граница $\Omega(\lg n)$ за височината, което дава долната граница $\Omega(\lg n)$ за сложността по време на алгоритъма.

7.3 Редукции

7.3.1 Долна граница $\Omega(n \lg n)$ за НАЙ-БЛИЗКА ДВОЙКА ЕЛЕМЕНТИ

Изч. Задача: CLOSEST PAIR

пример: Числа a_1, a_2, \dots, a_n .

решение: Минимално $|a_i - a_j|$ за $1 \leq i < j \leq n$.

[†]Може дори да няма такива—които стават невъзможни след някое сравнение—ако сравнението е безсмислено. Това може да се случи, ако алгоритъмът е глупав, но с нищо не променя валидността на аргументацията.

Това е едномерният вариант на задачата: входът са числа. Съществува и 2-мерен вариант, в който входът са наредени двойки от числа и се търси минимално разстояние между две двойки числа, където разстоянието може да се дефинира по различни начини, примерно Евклидово разстояние, Манхатънско разстояние и така нататък. Тук разглеждаме едномерния вариант.

Без съмнение съществува директно доказателство на долната граница $\Omega(n \lg n)$ за тази задача с дърво на вземане на решение. Но, както видяхме при задачата УНИКАЛНОСТ НА ЕЛЕМЕНТИ, директното доказателство може да е дълго и тежко. Тук ще извършим **редукция**. А именно, ще редуцираме задачата УНИКАЛНОСТ НА ЕЛЕМЕНТИ до задачата НАЙ-БЛИЗКА ДВОЙКА ЕЛЕМЕНТИ. По този начин ще използваме наготово резултата за добра граница на УНИКАЛНОСТ НА ЕЛЕМЕНТИ. Редукцията става по следния начин.

Нека ALGCP е произволен алгоритъм за задачата НАЙ-БЛИЗКА ДВОЙКА ЕЛЕМЕНТИ (базиран на директни сравнения, естествено). Да разгледаме следния алгоритъм AlgX.

```
ALGX(Числа  $a_1, a_2, \dots, a_n$ )
1    $d \leftarrow \text{ALGCP}(a_1, a_2, \dots, a_n)$ 
2   if  $d = 0$ 
3     return FALSE
4   else
5     return TRUE
```

Очевидно ALGX решава задачата УНИКАЛНОСТ НА ЕЛЕМЕНТИ. Нещо повече, неговата сложност по време е равна, в асимптотичния смисъл, на сложността на ALGCP, защото извън извикването на ALGCP на ред 1 и последващото връщане, ALGX извършва само константна допълнителна работа (редове 2–5). Да допуснем, че не е вярно, че НАЙ-БЛИЗКА ДВОЙКА ЕЛЕМЕНТИ има добра граница $\Omega(n \lg n)$. Тогава за НАЙ-БЛИЗКА ДВОЙКА ЕЛЕМЕНТИ има един алгоритъм алгоритъм, за който $\Omega(n \lg n)$ не е добра граница. Тогава за този алгоритъм $\Theta(n \lg n)$ е горна граница.

Стриктно говорейки, такава импликация няма: от това, че $\Omega(f(n))$ не е добра граница за сложността на някакъв алгоритъм не следва, че $\Theta(f(n))$ е горна граница за нея. Виж Допълнение 26. Тук прилагаме тази импликация, понеже разглеждаме “нормално държащи се” функции като $n \lg n$.

Ключовото наблюдение е, че ако ALGCP имаше сложност по време $\Theta(n \lg n)$, то и ALGX щеше има сложност по време $\Theta(n \lg n)$, а оттук и задачата УНИКАЛНОСТ НА ЕЛЕМЕНТИ щеше да има сложност $\Theta(n \lg n)$. Само че ние вече знаем, че УНИКАЛНОСТ НА ЕЛЕМЕНТИ има добра граница $\Omega(n \lg n)$. Следователно, невъзможно е ALGCP да работи във време $\Theta(n \lg n)$. Оттук имаме и добра граница $\Omega(n \lg n)$ за задачата НАЙ-БЛИЗКА ДВОЙКА ЕЛЕМЕНТИ.

Допълнение 26: Опровергаване на $\Omega(g(n))$ не влече $\Theta(g(n))$

Какво следва от факта, че $f(n) \neq \Omega(g(n))$? С други думи, какво влече $f(n) \neq g(n)$? Читателят може да се изкуши да помисли, че тогава непременно $f(n) < g(n)$, по аналогия с импликацията $x \geq y \rightarrow x < y$ върху реалните числа. Само че \geq и $<$ са релации върху функции и при тях нещата са по-сложни. Импликация $f(n) \neq \Omega(g(n)) \rightarrow f(n) \neq g(n)$ **няма**.

Съгласно Теорема 5, има точно два случая, в $f(n) \neq g(n)$ и $f(n) < g(n)$:

1. (2.30), тоест функциите са асимптотично несравними;

2. (2.31), тоест $f(n) \leq g(n)$ и само това.

Тези случаи обаче са екзотика, що се отнася до функции, които описват алгоритмична сложност. За да бъдат функциите асимптотично несравнени или едната да изостава от, и после да догонва, другата, и така до безкрай, едната или двете функции трябва да имат “подскачащ характер”. Обикновено срещащите се в практиката функции, описващи алгоритмична сложност, като n , $n \lg n$, n^2 , 2^n и така нататък не са такива. И така, за нормално държащите се, срещащи се на практика функции е вярно, че $f(n) \neq g(n)$ влече $f(n) < g(n)$.

Казваме, че задачата УНИКАЛНОСТ НА ЕЛЕМЕНТИ се редуцира до задачата НАЙ-БЛИЗКА ДВОЙКА ЕЛЕМЕНТИ. Точна дефиниция на понятието “редукция” ще дадем в следваща лекция. Тук просто споменаваме очевидния факт, че задачата УНИКАЛНОСТ НА ЕЛЕМЕНТИ има решение, което състои в решението на задачата НАЙ-БЛИЗКА ДВОЙКА ЕЛЕМЕНТИ, плюс пренебрежимо малко допълнителна работа.

Нотация 4: Символът “ α ” е за редукции

Нека Π_1 и Π_2 са произволни изчислителни задачи. Чрез “ $\Pi_1 \alpha \Pi_2$ ” означаваме факта, че Π_1 се редуцира до Π_2 .

Грубо казано, от $\Pi_1 \alpha \Pi_2$ правим два извода:

- Ако Π_1 е трудна задача (каквото и да значи това), то Π_2 също е трудна. Няма как никаква вътрешна, иманентна сложност да “изчезне” при редукцията.
- Ако Π_2 е лесна (каквото и точно да значи това), то Π_1 също е лесна. По същата причина: не може иманентната сложност да изчезне при редукцията.

Важно е да знаем, че следните изводи не са верни.

- Ако Π_1 е лека, то Π_2 също е лека. Такъв извод няма. Винаги можем да направим нещата произволно сложни.
- Ако Π_2 е трудна, то Π_1 също е трудна. Такъв извод няма. Причината отново е, че винаги можем да направим нещата сложни.

Наблюдение 24

Трансформирайки задача в задача, можем да добавим колкото си искаем допълнителна сложност отгоре. Това, което е невъзможно, е при редукция да намалим иманентната сложност.

7.3.2 Долна граница $\Omega(n \lg n)$ за Мода

Да си припомним определението на “мода” от Подсекция 3.2.4 и Определение 27. Формално, изчислителната задача Мода е следната.

Изч. Задача: МОДА

пример: Числа a_1, a_2, \dots, a_n .

решение: Най-често срещано число измежду дадените.

Ще докажем долна граница $\Omega(n \lg n)$ за Мода. Естествено, имаме предвид намиране на мода чрез директни сравнения. Нека ALGMODE е произволен алгоритъм за задачата Мода (базиран на директни сравнения, естествено). Той връща стойността на мода. Да разгледаме следния алгоритъм AlgY.

```
ALGY(Числа  $a_1, a_2, \dots, a_n$ )
1    $m \leftarrow \text{ALGMODE}(a_1, a_2, \dots, a_n)$ 
2    $c \leftarrow 0$ 
3   for  $i \leftarrow 1$  to  $n$ 
4     if  $a_i = m$ 
5        $c \leftarrow c + 1$ 
6   if  $c = 1$ 
7     return TRUE
8   else
9     return FALSE
```

Очевидно ALGY решава задачата УНИКАЛНОСТ НА ЕЛЕМЕНТИ. Нещо повече, неговата сложност по време е равна, в асимптотичния смисъл, на сложността на ALGCP, защото извън извикването на ALGMODE на ред 1 и последващото връщане, ALGX извършва само $\Theta(n)$ допълнителна работа (редове 2–9). Ако алгоритъмът ALGMODE имаше сложност по време $O(n \lg n)$ [†], то и ALGY щеше има сложност по време $O(n \lg n)$, а оттук и задачата УНИКАЛНОСТ НА ЕЛЕМЕНТИ щеше да има сложност $O(n \lg n)$. Ние вече знаем, че УНИКАЛНОСТ НА ЕЛЕМЕНТИ има долна граница $\Omega(n \lg n)$. Следователно, невъзможно е ALGMODE да работи във време $O(n \lg n)$. Оттук имаме и долна граница $\Omega(n \lg n)$ за задачата Мода.

7.4 Аргументиране чрез противник (Adversary argument)

7.4.1 Неформално обяснение

Вие искате да решите някаква задача върху някакви данни. Вие обаче нямаете директен достъп до данните. Данните се държат от друг субект, когото наричаме условно “дявола”. Дяволът позволява да му задавате въпроси за данните, и то адаптивно: задавате въпрос, той отговаря, а какъв ще бъде следващият Ваш въпрос (може да) зависи от получения отговор. Дяволът може да знае, но може и да не знае стратегията, която следвате. Какви са въпросите, които можете да задавате, зависи от правилата на играта. Може да са въпроси с булев отговор, примерно дали измежду две числа от данните, едното е по-малко от другото. Може да не са с булев, а с числен отговор, примерно, каква е стойността на число от входа, или каква е сумата на някакви числа от входа.

Лошотията на дявола се изразява единствено в това, че той Ви кара да зададете максимално много въпроси. Всеки път, когато зададете въпрос, дяволът дава такъв отговор, който ви принуждава, по отношение на всички зададени досега въпроси и получени отговори, да зададете максимално много въпроси, за да получите решение. Дяволът е **неограничен**

[†]Съображенията на Допълнение 26 са в сила.

умен; по-академично казано, той **има неограничени изчислителни способности** и във всеки момент знае точно каква информация Ви е разкрил досега и какъв отговор да даде на следващия Ви въпрос, за да Ви затрудни максимално. Както вече казахме, дяволът може да знае стратегията, която следвате, при което той знае всеки следващ Ваш въпрос, но може и да не я знае, при което той трябва да има предвид всяка възможна поредица от следващи Ваши въпроси. Подчертаваме, че дяволът **няма магически способности** и, ако не знае Вашата стратегия за решаване на задачата, няма как да знае следващия Ви въпрос.

Дяволът трябва да е **консистентен**: той е длъжен да се ангажира с отговорите, които е дал до момента. Ако веднъж каже, примерно, че $a_5 < a_{10}$, всички негови следващи отговори трябва да са консистентни с това; той вече не може да каже $a_5 \geq a_{10}$, нито може да даде поредица отговори, от които следва, че $a_5 \geq a_{10}$. Очевидно е, че ако дяволът имаше свободата да дава неконсистентни отговори, то цялото усилие щеше да е напълно безсмислено. Забележете, че **не се твърди**, че дяволът не лъже. Това, което се твърди, че лъжите на дявола не трябва да си противоречат. Както ще стане ясно от изложението тук, дяволът дори не лъже, защото **поначало вход няма**. Дяволът създава входа в процеса на отговаряне на Вашите въпроси. Единствената му лъжа би била тази: ако в края на играта го попитате “Това ли беше входът поначало?”, той би казал “Да.”, а всъщност в началото вход изобщо не е имало. Дяволът е създал в процеса на отговаряне такъв вход, спрямо редицата от Вашите въпроси, който Ви е накарал да зададете максимален брой въпроси.

Ето пример. Нека задачата е СОРТИРАНЕ, и то базирано на директни сравнения. Вие задавате въпроси от вида $a_i ? a_j$, за да изчислите пермутацията на входа, която представлява сортирана редица. Работата на дявола е да Ви накара да зададете максимален брой въпроси. Във всеки момент дяволът знае какви отговори е дал до момента, колко пермутации на n елемента са консистентни с досега дадените отговори и за всяко следващо запитване от вида $a_i ? a_j$, дяволът отговаря **ДА** или **НЕ** въз основа на един единствен критерий – при кой от тези отговори, броят на пермутациите, които остават възможни (тоест, които са консистентни с всички досегашни отговори и този отговор) е по-голям. Ако отговор **ДА** дава по-голям брой пермутации, дяволът ще каже **ДА**. Ако отговор **НЕ** дава по-голям брой пермутации, дяволът ще каже **НЕ**. Ако отговор **ДА** оставя същият брой пермутации като отговор **НЕ**, дяволът казва **ДА** или **НЕ** произволно. В началото дяволът изобщо не си направил труда да генерира вход – вход не е имало. В края на играта, ако поискате от дявола да ви покаже входа, той ще изчисли вход, който е консистентен с всички въпроси и отговори, и ще изльже, че точно това е бил входът през цялото време.

Сега да конкретизираме примера. Нека задачата е СОРТИРАНЕ НА ТРИ ЕЛЕМЕНТА, базирано на директни сравнения. Да кажем, че трите елемента са уникатни. Нека дяволът не знае Вашата стратегия за получаване на решение. В този пример лошотията на дявола е в това, че ще Ви кара да зададете (поне) три въпроса. Очевидно, в някои случаи задачата може да се реши само с два въпроса, ако входът е подходящ за тях. Примерно, ако $a_1 < a_2 < a_3$ и въпросите са $a_1 ? a_2$ и $a_2 ? a_3$. Но при игра срещу дявола никога няма да “минете” само с два въпроса, защото дяволът ще генерира, по отношение на Вашите въпроси, вход, който е “неподходящ”. Вашият първи въпрос е или $a_1 ? a_2$, или $a_1 ? a_3$, или $a_2 ? a_3$. Кой от тези въпроси ще зададете, това дяволът не знае. Дяволът обаче знае, че има точно 6 възможни пермутации на входа и за всеки от тези въпроси, точно 3 пермутации са съвместими с положителния отговор и точно 3, с отрицателния. Така че няма значение какво ще отговори на първото запитване; да кажем, че дяволът винаги отговаря с **ДА** на първия въпрос. На втория въпрос обаче дяволът не отговаря произволно! Тъй като след отговора на първия въпрос има 3 възможни пермутации, консистентни с отговора, то, какъвто и втори въпрос да зададете, съществува отговор – **ДА** или **НЕ**, в зависимост от конкретиката – такъв че поне

две пермутации са съвместими и с първия, и с втория отговор. Е, дяволът ще даде точно този отговор на втория въпрос и по този начин ще Ви принуди да зададе и трети въпрос. Забележете, че той ще ви принуди да зададете и трети въпрос независимо от това, какви са първия и втория Ви въпрос. След като получите решението и играта приключи, дяволът лесно може да предостави такъв вход, който е консистентен с всички отговори, които е дал, така че не може да бъде уличен в лъжа и измама.

Допълнение 27: МОРСКИ ШАХ с дявола

Няколко игри се наричат МОРСКИ ШАХ на български. Тук ще разгледаме една от тях (на английски е BATTLESHIPS), която е удачна илюстрация на основната идея на игра спрещу злонамерен противник, разполагащ с неограничена изчислителна мощ.

Правилата са следните. Играят двама играчи. Всеки разполага с квадратна мрежка, да кажем 11×11 , с номерирани редове и колони, наречена *mope*:

	1	2	3	4	5	6	7	8	9	10	11
1											
2											
3											
4											
5											
6											
7											
8											
9											
10											
11											

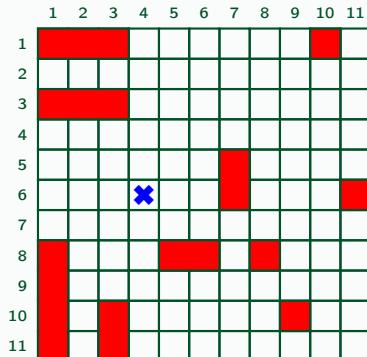
и няколко фигури, наречени *кораби*. Всеки кораб е правоъгълник, който може да бъде поставен, хоризонтално или вертикално, върху няколко съседни квадратчета от мрежата, запълвайки ги точно. Корабите са следните: един кораб 1×4 , два кораба 1×3 , три кораба 1×2 и четири кораба 1×1 . Всеки играч крие своето море от другия играч. Преди началото на играта, всеки играч разполага своите кораби в своето море както намери за добре, но скришно от другия играч. Има ограничение два кораба да не се не се припокриват и да не се докосват, нито по хоризонтал, нито по вертикал, нито по диагонал. Ето примерно разполагане на корабите на единия играч:

	1	2	3	4	5	6	7	8	9	10	11
1											
2											
3											
4											
5											
6											
7											
8											
9											
10											
11											

Самата игра се състои в редуванци се “изстрели” на всеки играч в морето на другия играч, като “изстрел” е просто казване на координатите на някое квадратче от морето на противника. Всеки играч записва изстрелите на противника си с хиксове върху

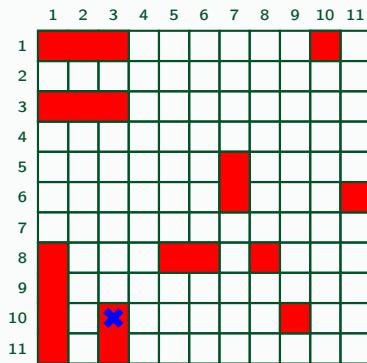
своето море. Единият играч е първи. Той или тя казва на втория някаква наредена двойка (i, j) , където $1 \leq i, j \leq 11$. Вторият играч прави следното:

- Ако на (i, j) няма кораб (в нашия пример, нека $(i, j) = (6, 4)$), вторият казва “вода”, отбелязва хикс на (i, j) в своето море:

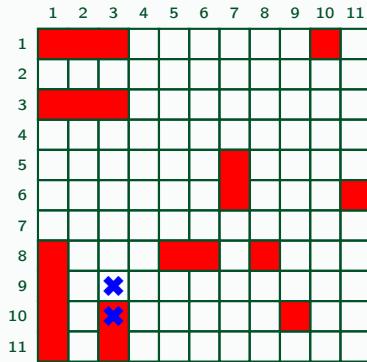


и на свой ред играе, казвайки координатите на някакво квадратче от морето на първия играч.

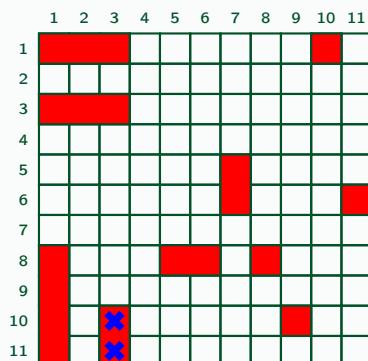
- Ако на това квадратче има кораб, който е по-голям от 1×1 и има поне едно неударено квадратче освен (i, j) (в нашия пример, нека $(i, j) = (10, 3)$), то вторият играч казва “удар” и отбелязва удара с хикс:



След това отново играе първият играч – този, който току-що нанесе удара. Първият продължава да играе, докато или не уцели вода:

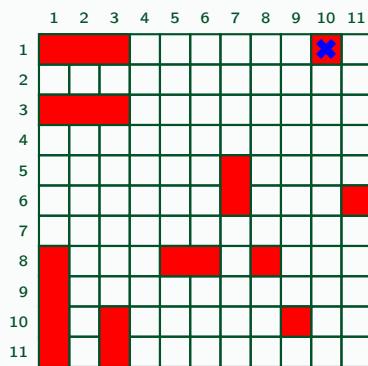


при което вече играе вторият играч, или не удари и последното досега неударено квадратче:

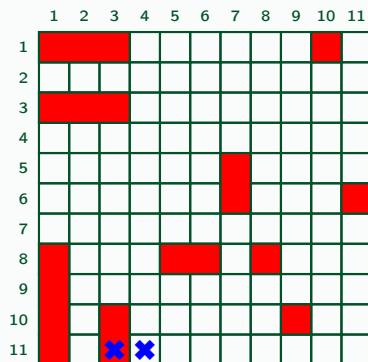


при което вторият играч казва "потопен" и първият играч продължава да играе.

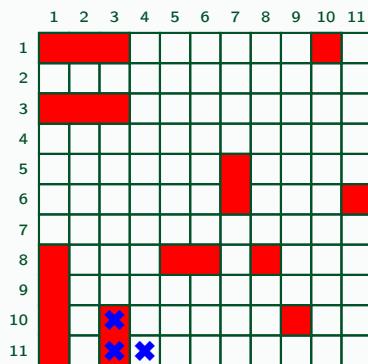
- Ако или на това квадратче има кораб 1×1 —в нашия пример, нека $(i, j) = (1, 10)$:



или (i, j) е единственото неударено квадратче на кораб, по-голям от 1×1 —в нашия пример, нека $(i, j) = (10, 3)$ и досега първият играч е направил тези два изстrela:

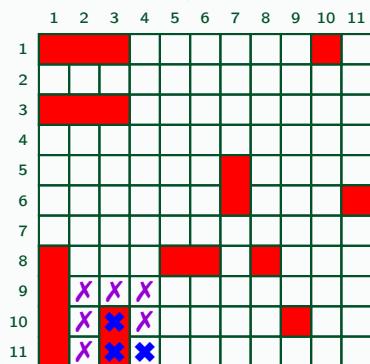


то вторият играч казва "потопен" и отбелязва мястото на удара с хикс.



После първият играч играе отново.

Печели играчът, който пръв или първа потопи всички кораби на противника си. Естествено, по време на играта няма смисъл играч да стреля върху квадратче, върху което вече е стрелял. Също така няма смисъл да стреля в района, ограждащ вече потопен кораб; ако разгледаме отново последния пример, след потапянето на кораба на $\langle(10, 3), (11, 3)\rangle$, всички квадратчета около него, в които досега не е стреляно, стават безсмислени като цели:



Това са правилата на играта. Сега да допуснем, че играете с дявола на морски шах. Почти сигурно ще загубите! Като добросъвестен и почтен човек, Вие поставяте корабите си в самото начало и после отговаряте вярно на изстрелите на противника, които улучват Вашите вече поставени кораби, с “удар” или “потопен”. Дяволът обаче не е почтен и със сигурност на Вашите начални изстрели, където и да са те, той ще отговори с “вода”. Същността на неговата стратегия е, че той изобщо не е поставил кораби. Дяволът не може да отговаря с “вода” прекалено дълго, защото съществува множество от изстрели от Ваша страна, които са така разположени, че неговите кораби нямат възможни позиции спрямо тях – но Вие с почти пълна сигурност няма да имате възможност да дадете толкова изстрели, защото ще сте загубили играта отдавна, играйки почтено.

Ако след загубата си поискате от дявола да ви покаже как са били разположени корабите му през цялото време, той ще изчисли такова разположение на корабите си, че нито един от Вашите изстрели не е попадение, и ще ви покаже него.

Имайте предвид, че дяволът е неограничено умен и ще отлага до последно първото попадение върху свой кораб. Както вече казахме, най-вероятно Вие ще сте загубили играта много преди да имате възможност да дадете толкова изстрели, че дяволът да не може повече да отлага първото попадение.

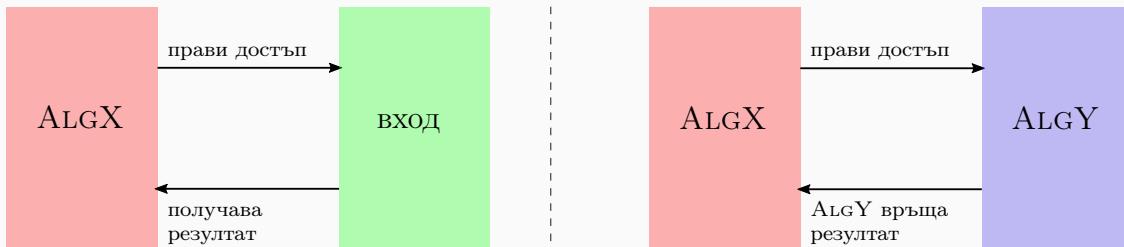
По друг начин казано: корабите общо заемат $4 + 6 + 6 + 4 = 20$ квадратчета, а морето е $11 \times 11 = 121$ квадратчета. Вероятността при първи изстрел да бъде ударен кораб, ако се играе почтено и кораби са разположени поначало, а даващия изстрел стреля по случайно квадратче, е $\frac{20}{121} \approx \frac{1}{6}$. Ако играете достатъчно дълго с дявола и се редувате за пръв изстрел, в горе-долу $\frac{1}{6}$ от игрите неговият пръв изстрел ще е попадение (удар или потапяне). Дяволът няма магически способности и не вижда корабите Ви, така че в около $\frac{5}{6}$ от игрите първият му изстрел ще е “вода”. Ще се изненадате обаче (ако не сте наясно, че играете срещу мошеник) от това, че **всеки** Ваш пръв изстрел е “вода”. Защото дяволът не слага кораби поначало. Нещо повече, Вие ще загубите преди да можете да реализирате попадение, така че **всеки** Ваш изстрел по време на **цялата** игра ще е “вода”.

Примерът с морския шах не е перфектна илюстрация за даване на долна граница на изчислителна задача, понеже при решаване на изчислителна задача играете само Вие (само Вие задавате въпроси), а дяволът само отговаря; в морския шах Вие се редувате с дявола за задаване на въпроси. Въпреки тази разлика, би трябвало този пример да даде добра интуиция за стратегията на опонент, който е максимално злонамерен и има неограничена изчислителна мощ.

7.4.2 Формално определение

Долна граница за някакъв предварително известен алгоритъм ALGX. Разглеждаме ALGX и друг алгоритъм ALGY, който играе ролята на противника (дявола). Всеки достъп на ALGX до входните данни всъщност е едно запитване (query) от страна на ALGX към ALGY. Може да си представяме, че ALGY прехваща всеки опит на ALGX за достъп до входа и той (ALGY) връща стойност, която, от гледна точка на ALGX, е легитимен резултат от достъпа да входа. Може още да си представяме, че ALGX не знае за съществуването на ALGY и няма как да разбере, че ALGY прехваща всеки опит за достъп до входа и резултатът от достъпа е това, което ALGY решава да предостави. И накрая, можем да мислим, че вход изобщо няма и ALGY връща, каквото намери за добре. Фигура 7.5 илюстрира всичко това.

Фигура 7.5 : Аргументация с противник: два алгоритъма.



Какво си мисли ALGX, че става.

Какво става в действителност.

Основните допускания са следните.

- ALGY разполага с ALGX и оттам знае каква ще е поредицата от достъпи до входа, които ще извърши ALGX.
- ALGY е неограничено мощн. По отношение на ALGY не се интересуваме от това, какви ресурси ползва като време и памет. С други думи, не ни интересува колко ефикасен е ALGY, стига да е ефективен.
- ALGY връща такава поредица от отговори на ALGX, че ALGX да направи поне някакъв брой достъпи (queries), който брой е долната граница, която доказваме.
- ALGY трябва да е консистентен. Редицата резултати \mathcal{R} , които той дава на ALGX, трябва да е такава, че да има поне един вход със съответната големина, за който редицата от резултатите е точно \mathcal{R} .

Както казахме, ALGY играе ролята на противника (дявола). Дизайнерите на ALGX обаче сме ние. Нашата цел е да докажем долна граница за работата на ALGX в най-лошия случай. За тази цел ние конструираме противник (ALGY), който е в състояние да създаде такъв вход, който да накара ALGX да извърши поне толкова достъпи, колкото е долната граница, която доказваме. Подчертаваме, че ALGY не просто генерира някакъв вход поначало, а прави нещо много по-изтънчено и перфидно: **ALGY създава входа в процеса на достъпи и връщане на резултати**, отчитайки конкретните достъпи, извършени от ALGX.

Да кажем, че ALGY кара ALGX да направи максимален брой достъпи е прекалено амбициозно. В предната подсекция се твърдеше, че дяволът, бивайки неограничено умен и абсолютно злонамерен, връща отговори, които да ни накарат да работим максимално много. Когато конструираме противник-алгоритъм, целта ни е по-скромна, но затова пък постижима. Ние конструираме противник за доказване на някаква долна граница $f(n)$, където n е големината на входа. Ако конструираме противник ALGY, който кара ALGX да извърши поне $f(n)$ достъпа за всяко n , и докажем това, ние сме си свършили работата. При това остава възможно да има друг противник ALGY', който кара ALGX да извърши дори повече достъпи; да кажем, $f'(n)$, където $f'(n) > f(n)$ за всички достатъчно големи n . Това, че може да има дори “по-лош” противник от този, който сме намерили и анализирали, не е в противоречие с нашето доказателство, че $f(n)$ е долната граница.

Наблюдение 25

Да допуснем, че сме доказали, че конструираният от нас противник кара ALGX да прави поне $f(n)$ достъпа за всяко n . Ако освен това докажем, че $f(n)$ е **точна долната граница** за броя на достъпите, то ние сме конструирали оптимален противник, който точно отговаря на дявола от Подсекция 7.4.1. В противен случай просто сме конструирали противник, който върши работа за дадено доказателство, оставяйки възможността да има дори по-лош противник.

Долна граница за изчислителна задача П. Това е сравнително по-труден за конструиране и анализиране противник. Сега вече противникът ALGY не знае нищо за конкретния алгоритъм ALGX, който решава П. От гледна точка на ALGY, ALGX е просто черна кутия, от която излизат достъпи до входа (queries). Фигура 7.5 илюстрира адекватно и тази ситуация, но сега имаме предвид, че

- ALGY не знае нищо за ALGX.
- “ALGX” не означава конкретен алгоритъм, а означава кой да е алгоритъм за задачата П.

Допусканията за неограничена мощност, “злонамереност” и консистентност на ALGY са в сила и сега.

7.4.3 Примери за доказване на долни граници чрез противник

Ще покажем няколко добре известни доказателства за долни граници на задачи чрез противник. Това не са асимптотични долни граници, каквито видяхме при задачи като СОРТИРАНЕ или УНИКАЛНОСТ НА ЕЛЕМЕНТИ. Тук ще докажем точни изрази-долни граници върху броя на сравненията за няколко лесни задачи.

$n - 1$ за намиране на максимален елемент

Задачата е да се намери чрез, и само чрез, директни сравнения максимално число измежду произволни числа a_1, a_2, \dots, a_n . Първо забелязваме, че всяко a_i трябва да участва в поне едно сравнение. Да допуснем противното: нека има коректен алгоритъм ALGMAX за тази задача, който върху поне един вход работи така, че някой a_i не участва в нито едно сравнение. Тогава противникът манипулира входа така:

- Ако ALGMAX върне a_i^{\dagger} , то противникът прави a_i по-малко от всяко друго число от входа. При това положение няма как a_i да е максимално число. Не можем да уличим противника в лъжа, защото a_i не е било сравнявано изобщо, така че противникът може да му присвои—след края на работата на ALGMAX—каквото поиска и да ни го покаже.
- Ако ALGMAX върне a_j , където $j \neq i$, противникът прави a_i по-голям от всеки друг елемент. При това положение a_i е максималното число. Отново, не можем да уличим противника в лъжа, защото a_i не е бил сравнявано изобщо.

И в двата случая ALGMAX не работи коректно, така че той не е коректен алгоритъм и допускането е опровергано.

Долната граница, която следва от съображението, че всеки елемент трябва да е бил сравнен поне веднъж с друг, е $\lceil \frac{n}{2} \rceil$. За да се убедим в това, да си представим неориентиран граф на сравненията, в който върховете са числата a_1, a_2, \dots, a_n , а ребрата са ненаредените двойки числа, които са били сравнени от алгоритъма. Твърдението, че всяко число трябва да е било сравнено поне веднъж е същото като твърдението, че графът няма изолирани върхове. Лесно се вижда, че минималният брой ребра в граф без изолирани върхове е $\lceil \frac{n}{2} \rceil$.

Но долната граница $\lceil \frac{n}{2} \rceil$ е твърде слаба. Не че всичко това не е истина, но има по-висока добра граница за броя на сравненията, която сега ще докажем. Ще докажем, че ако ALGMAX е коректен алгоритъм за задачата, то той прави поне $n - 1$ сравнения.

Първо доказателство. Да си представим отново графа на сравненията. Твърдим, че той трябва да е свързан.

Да допуснем, че графът на сравненията не е свързан. По-подробно казано, допускаме, че има вход a_1, a_2, \dots, a_n , върху който ALGMAX прави такива сравнения, че графът на сравненията има поне две свързани компоненти. Освен това допускаме, че ALGMAX работи коректно върху a_1, a_2, \dots, a_n и числото a_i , което той връща, е максимално. Но a_i е връх в някая свързана компонента G' от графа на сравненията. Тъй като този граф не е свързан, той има една друга свързана компонента G'' . Противникът манипулира входа така: прави всеки елемент-връх в G'' по-голям от всеки елемент-връх от G' , като относителната наредба между елементите-върхове от G'' се запазва. При това положение максимумът няма как да е от G' , така че ALGMAX не е работи коректно върху a_1, a_2, \dots, a_n . Противникът не може да бъде уличен в лъжа постфактум, защото той ще запълни входа по такъв начин, че елементите, които са върхове в G'' , имат такива стойности, за които изходите от сравненията са точно такива, каквито е получил ALGMAX по време на работата си.

Ключовият факт, върху който почиват тези разсъждения, е, че ако графът не е свързан, за всеки две различни свързани компоненти G' и G'' е вярно, че не може да заключим нищо за относителните големини на елемент-връх от G' и елемент-връх от G'' . Може всеки елемент от G' да е по-голям от всеки елемент от G'' , може всеки елемент от G'' да е по-голям от всеки

[†]Звучи безумно алгоритъмът да върне елемент, който изобщо не е бил “прегледан”, но тук правим теоретично доказателство и разглеждаме всички случаи.

елемент от G' , може всички елементи от G' да са с големини между два съседни по големина елементи от G'' , и така нататък – всичко е възможно за относителните им големини.

И така, доказваме, че хипотетичният ALGMAX, който върху поне един вход прави такива сравнения, че техният граф не е свързан, не е коректен. Заключаваме, че графът на сравненията е свързан. Добре известен факт от теорията на графиките е, че всеки свързан граф на n върха има поне $n - 1$ ребра. Това е и краят на доказателството на тази долна граница.

Второ доказателство. Сега да допуснем, че елементите от входа са два по два различни. Може да аргументираме долната граница $n - 1$ и със следните разсъждения. Нека казваме, че a_j е загубило от a_k , а a_k е спечелило от a_j , ако ALGMAX е направил сравнение $a_j < a_k$ и резултатът от сравнението е бил ДА. Нека ALGMAX връща като максимално число a_i .

Лема 35

В текущия контекст е вярно, че всяко число, различно от a_i , е загубило поне едно сравнение.

Доказателство: Ако допуснем противното, а именно, че за някое $j \neq i$, a_j не е губил сравнение, то противникът манипулира входа, присвоявайки на a_j стойност, която е по-голяма от стойността на всеки друг елемент. Сега вече a_j със сигурност е максималният елемент от масива, но резултатът от всяко сравнение, в което участва a_j , си остава същият – отново a_j не губи нито едно сравнение. Резултатите от сравненията, в които a_j не участва, очевидно не се променят от увеличаването на стойността на a_j . Но тогава ALGMAX продължава да връща a_i , защото резултатите от всички сравнения са същите като тези преди увеличаването на a_j .

Накратко, ако повече от един елемент не е губил сравнение, алгоритмът няма как да е коректен, защото противникът лесно ще установи това и ще направи максимум някой от тези елементи, които алгоритъмът не връща като максимален.

В това доказателство ние пак имаме предвид граф на сравненията, но сега той е **ориентиран**. Върховете пак са a_1, a_2, \dots, a_n , но всяко ребро има посока. На всяко сравнение $a_p < a_q$ съответства:

- или ориентираното ребро (a_q, a_p) при отговор ДА,
- или ориентираното ребро (a_p, a_q) при отговор НЕ.

Очевидно този граф е dag – обратното би означавало, че противникът е неконсистентен, което не е разрешено по условието[†]. Елементите, които не са загубили сравнение, са точно източниците. Е, противникът може лесно да установи дали dag-ът има повече от един източник и ако е така, да манипулира стойностите на тези източници, така че да “прецака” ALGMAX. ALGMAX ще върне някой от тези източници, а противникът ще направи друг от тях да има по-голяма стойност от този, който ALGMAX връща като максимален елемент. Няма как да уличим противника в лъжа постфактум, понеже всяка двойка източници са несравними в dag-a. □

Лема 35 ни дава желания резултат, понеже всички други елементи (освен a_i) са $n - 1$ и всеки от тях трябва да е бил сравнен поне веднъж, иначе няма как да е загубил сравнение.

[†]Цикъл в графа би имало тогава и само тогава, когато противникът е казал, че един елемент е по-голям от друг, той е по-голям от трети, и така нататък, и последният е по-голям от първият.

В току-що направеното доказателство нарушихме изискването противникът да не знае нищо за работата на ALGMAX, а само да решава какво да отговори на следващото сравнение на базата на отговорите на предните сравнения. Противникът някак вижда, че ALGMAX връща един източник на dag-а a_i и спрямо това прави друг източник на dag-а a_j по-голям от a_i . Но това не е съществена спънка. Дори да лишим противника от възможността да вижда какво прави ALGMAX, може да кажем така:

Тъй като правим анализ на най-лошия случай, самото съществуване на възможност противника да “прецака” ALGMAX е същото като противникът да “прецака” ALGMAX.

Заслужава да се отбележи, че долната граница $n - 1$ за броя на сравненията е точна добра граница. Следният прост алгоритъм намира максимален елемент във входа с $n - 1$ сравнения.

MAXELEMENT($A[1, \dots, n]$: int)

```

1 tmp ← A[1]
2 for i ← 2 to n
3   if A[i] > tmp
4     tmp ← A[i]
5 return tmp
```

$\left\lceil \frac{3n}{2} \right\rceil - 2$ за намиране на максимален и минимален елемент

Задачата е да се намери чрез, и само чрез, директни сравнения максимално число и минимално число измежду произволни числа a_1, a_2, \dots, a_n , където $n \geq 2$. Ще допуснем, че числата са две по две различни, така че ще казваме, че търсим максимума и минимума. Очевидно може да намерим максимума и минимума поотделно с общо $(n - 1) + (n - 1) = 2n - 2$ сравнения, но може да сторим това и по-бързо ето така.

Първо да допуснем, че n е четно. Групираме числата в $\frac{n}{2}$ двойки, във всяка двойка намираме максимума и минимума с едно сравнение и получаваме $\frac{n}{2}$ максимума, по един от всяка двойка, и още $\frac{n}{2}$ минимума, по един от всяка двойка. Глобалния максимум намираме от тези $\frac{n}{2}$ максимума чрез $\frac{n}{2} - 1$ сравнения. Аналогично, глобалния минимум намираме от тези $\frac{n}{2}$ минимума чрез $\frac{n}{2} - 1$ сравнения. Общийт брой на използваните сравнения е

$$\frac{n}{2} + \left(\frac{n}{2} - 1 \right) + \left(\frac{n}{2} - 1 \right) = \frac{3n}{2} - 2$$

което при четно n можем да напишем като

$$\left\lceil \frac{3n}{2} \right\rceil - 2$$

Сега да допуснем, че n е нечетно. Броят на двойките е $\left\lfloor \frac{n}{2} \right\rfloor$ и едно число a_j остава извън двойките. Намираме $\left\lfloor \frac{n}{2} \right\rfloor$ максимума, по един от всяка двойка, и още $\left\lfloor \frac{n}{2} \right\rfloor$ минимума, по един от всяка двойка. После глобалният максимум се намира измежду тези максимуми и a_j с $(\left\lfloor \frac{n}{2} \right\rfloor + 1) - 1 = \left\lfloor \frac{n}{2} \right\rfloor$ сравнения. После глобалният минимум се намира от тези минимуми и a_j (разглеждаме най-лошият случай, в който a_j не се е окказал глобален максимум и сме длъжни да проверим дали не е глобален минимум) пак с $(\left\lfloor \frac{n}{2} \right\rfloor + 1) - 1 = \left\lfloor \frac{n}{2} \right\rfloor$ сравнения. Общийт брой на сравненията е $3 \left\lfloor \frac{n}{2} \right\rfloor$. При $n = 2k + 1$, това е

$$3 \left\lfloor \frac{2k + 1}{2} \right\rfloor = 3k = (3k + 2) - 2 = \frac{6k + 4}{2} - 2 = \left\lceil \frac{6k + 3}{2} \right\rceil - 2 = \left\lceil \frac{3(2k + 1)}{2} \right\rceil - 2 = \left\lceil \frac{3n}{2} \right\rceil - 2$$

Виждаме, че и при четно, и при нечетно n тази идея води до решение с $\lceil \frac{3n}{2} \rceil - 2$ сравнения. Следният алгоритъм реализира тази идея. Алгоритъмът очевидно може да се реализира in-place, но сега сложността по памет не ни интересува, така че можем да си позволим реализация с линейна сложност по памет.

```

MAXNMIN(A[1, ..., n]: int, n ≥ 2)
1  k ← ⌊ n / 2 ⌋
2  създай масиви MAX[1, ..., k], MIN[1, ..., k]
3  for i ← 1 to k
4      if A[2i - 1] > A[2i]
5          MAX[i] ← A[2i - 1], MIN[i] ← A[2i]
6      else
7          MAX[i] ← A[2i], MIN[i] ← A[2i - 1]
8  tmpmax ← MAX[1], tmpmin ← MIN[1]
9  for i ← 2 to k
10     if MAX[i] > tmpmax
11         tmpmax ← MAX[i]
12     if MIN[i] < tmpmin
13         tmpmin ← MIN[i]
14 if isodd(n)
15     if A[n] > tmpmax
16         tmpmax ← A[n]
17     if A[n] < tmpmin
18         tmpmin ← A[n]
19 return (tmpmax, tmpmin)

```

Ще докажем, че този алгоритъм е оптимален по отношение на броя на сравненията. Ще докажем, че има противник, който може да манипулира входа така, че да накара всеки алгоритъм за задачата да направи поне $\lceil \frac{3n}{2} \rceil - 2$ сравнения.

Да си представим някакъв алгоритъм за тази задача. Както вече видяхме в доказателството на Лема 35, сравненията, направени от алгоритъма, индуцират ориентиран граф, чиито върхове са числата a_1, a_2, \dots, a_n , а ребро (a_i, a_j) съществува тогава и само тогава, когато a_i е било сравнено с a_j и резултатът е бил $a_i > a_j$. Както видяхме там, този граф е dag, който има точно един източник. Източникът отговаря на максималното число. Напълно аналогично, този dag има точно един сифон, който отговаря на минималното число. Очевидно всяко число, което не е нито максималното, нито минималното, има indegree поне единица и outdegree поне единица. Използвайки терминологията, която се ползва от Лема 35, вярно е, че всяко число без едно е загубило поне едно сравнение (най-голямото не губи никое сравнение) и всяко число без едно е спечелило поне едно сравнение (най-малкото не печели никое сравнение).

В началото нито едно число не е било сравнено. След това алгоритъмът започва да сравнява двойки числа и в края на работата му $n - 2$ числа са спечелили поне по едно сравнение и загубили поне по едно сравнение, едно число само е печелило сравнения и едно друго число само е губило сравнения. Има смисъл да въведем състояние на всяко от числата по време на работата на алгоритъма. Във всеки момент, всяко число x е в точно едно от следните четири състояния:

N , ако x още не е било сравнявано.

W , ако x е спечелило поне едно сравнение и не е загубило нито едно сравнение.

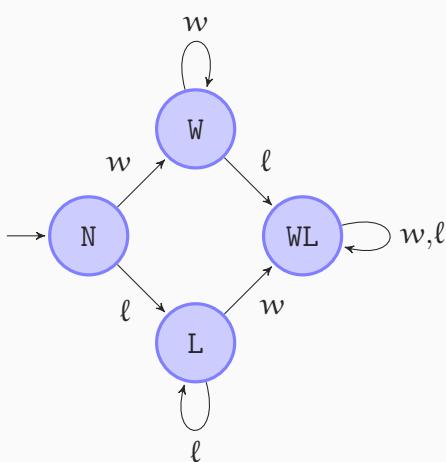
L , ако x е загубило поне едно сравнение и не е спечелило нито едно сравнение.

WL , ако x е спечелило поне едно сравнение и е загубило поне едно сравнение.

В началото всяко число е в състояние N . В края всички числа без две са с в състояние WL , точно едно е в състояние W и точно едно е в състояние L . В никакъв смисъл, работата на алгоритъма е да “вкара” числата в тези състояния – когато, и само когато, всички числа са в така описаните състояния, задачата е решена. Работата на противника е да забави колкото е възможно повече постигането на тези състояния.

Да помислим как число минава от състояние в състояние след извършване на сравнения. Началното състояние е N . WL е мъртво състояние – от него не може да се излезе. Всеки преход става след сравнение на това число с друго, което означава, че нашето число или е спечелило, или е загубило. Сравнение, в което то е спечелило, означаваме с “ w ”, а сравнение, в което то е загубило, означаваме с “ l ”. Фигура 7.6 показва диаграмата на състоянията и преходите за произволно число.

Фигура 7.6 : Диаграма на състоянията и преходите.



За достигане на WL от N са необходими поне два прехода.

Сега ще опишем стратегия на противника при сравняване на две числа, която максимизира $\min_{i,j} \{a_i < a_j\}$. Да кажем, че се извършва сравнение $a_i \stackrel{?}{<} a_j$. Всяко от a_i и a_j има състояние от $\{N, W, L, WL\}$, което е известно на противника. Да разгледаме всички възможни комбинации от състояния. Те са само $\binom{2+4-1}{2} = 10$, а не $4^2 = 16$, защото комбинациите са двуелементните мултимножества над опорното множество $\{N, W, L, WL\}$. Следната таблица описва как противникът манипулира данните във всеки възможен случай.

Съст-ние на a_i	Съст-ние на a_j	Избор на противника	Можеше ли да избере обратното?	Ново съст-ние на a_i	Ново съст-ние на a_j	Промяна в броя на съст-нията
N	N	$a_i < a_j$	да	L	W	+2
N	W	$a_i < a_j$	не	L	W	+1
N	L	$a_i > a_j$	не	W	L	+1
N	WL	$a_i < a_j$	да	L	WL	+1
W	W	$a_i < a_j$	да	WL	W	+1
W	L	$a_i > a_j$	не	W	L	0
W	WL	$a_i > a_j$	не	W	WL	0
L	L	$a_i < a_j$	да	L	WL	+1
L	WL	$a_i < a_j$	не	L	WL	0
WL	WL	$a_i < a_j$	да	WL	WL	0

От особен интерес са петте комбинации, в които противникът няма избор за своя отговор (оцветените редове). Примерно, да разгледаме втория ред. Състоянието на a_i е N, а това на a_j е W; тоест, a_i не е участвал в сравнения изобщо, а a_j е участвал в поне едно сравнение и спечелил всички сравнения, в които е участвал.

- Ако противникът отговори $a_i < a_j$ (както прави според таблицата), той променя състоянието само на a_i . Очевидно a_i не може да остане N, щом вече е участвал в едно сравнение. Но a_j си остава в W, защото продължава да е вярно, че само е печелил сравнения.
- Ако противникът отговори $a_i > a_j$ (което той няма да направи, защото е неограничено умен и знае, че би било в негов ущърб да отговори така), то a_i би се оказал в W, спечелвайки срещу a_j , но a_j би се оказал в WL, след като е отбелязал първата си загуба. В този случай и a_i , и a_j биха променили състоянията си.

От този пример трябва да е ясно каква е играта на противника: изходът от всяко сравнение е такъв, че се промяната в състоянията **на двета участника** да е минимална. Ако се сравняват N с N (първият ред), промяната е в състоянията и на двете числа, независимо от отговора на противника (съответно, при това сравнение противникът отговаря произволно). Колкото и да не иска, противникът е принуден да даде две “парчета” информация, по едно за всяко число. Естествено, сравненията N срещу N са най-много $\lfloor \frac{n}{2} \rfloor$ на брой. Във всеки друг случай (освен N срещу N), противникът може да отговори по такъв начин, че или да не промени нито едно от състоянията (например W срещу L на ред шести), или да промени само едно от състоянията.

Да въведем понятието *макросъстояние* като векторът от състоянията на всички числа. Началното макросъстояние е $\langle N, N, \dots, N \rangle$. Да кажем, че *крайно макросъстояние* е всеки от векторите с един елемент W, един елемент L и останалите $n - 2$ елемента WL. Както вече отбеляхме, имаме категоричен отговор на задачата тогава и само тогава, когато е достигнато крайно макросъстояние. За достигането на крайно макросъстояние от началното макросъстояние, трябва да бъдат извършени поне $2n - 2$ прехода, защото:

- за всяко число, което завършва в WL са необходими поне 2 прехода (вижте Фигура 7.6), което означава поне $2(n - 2) = 2n - 4$ прехода общо.
- за числото, което завършва в W е необходим поне 1 преход,
- за числото, което завършва в L е необходим поне 1 преход.

И така, трябва да бъдат извършени поне $2n - 4 + 2 = 2n - 2$ прехода общо от всички числа. Това обаче не означава $2n - 2$ сравнения! Както видяхме, има сравнения, при които **и двете** участващи числа извършват преход. Това са точно сравненията N срещу N . При всички останали сравнения, противникът може да манипулира входа така, че да има не повече от 1 преход на сравнение **и за двете** участващи числа.

Сравненията от вида N срещу N са най-много $\left\lfloor \frac{n}{2} \right\rfloor$ на брой, както вече отбелязахме. Това означава, че противникът винаги може да наложи

$$2n - 2 - \left\lfloor \frac{n}{2} \right\rfloor = 2n - 2 + \left\lceil -\frac{n}{2} \right\rceil = \left\lceil 2n + \left(-\frac{n}{2} \right) \right\rceil - 2 = \left\lceil \frac{3n}{2} \right\rceil - 2$$

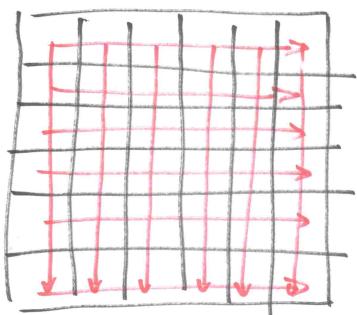
сравнения.

$\left\lceil \frac{3n-3}{2} \right\rceil$ за намиране на медиана

$n - 2 + \lceil \lg n \rceil$ за намиране на втори максимален елемент

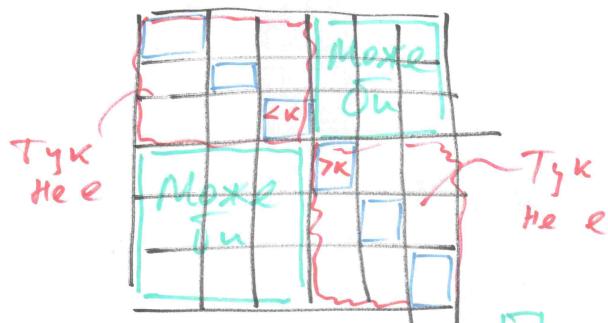
$2n - 1$ за търсене в сортирана по редове и колони, $n \times n$ матрица

Матрица от числа
 $n \times n$; число k .
Как k се среща?
Матр. е сортирана
по редове и колони.



Сложност?

(Нома) и дед
Двуетапно търсение в
Гл. диагонал, който
е сортиран \downarrow . Ако
не е там, разглежд.
4 подматрици. 2 отпадат



Рек. търсение б

Сложност $T_n = 2T_{\frac{n}{2}} + \lg n$
при идеална диаграма
 $T_n \propto n \cdot \lg n$ I MT

1 обрън и съдъ
"насупстната линий"

От Долу-надо: ако
 $a_{ij} = \text{key}$, return $(i:j)$,
иначе, ако $a_{ij} < \text{key}$,
ход надоло,
ако $a_{ij} > \text{key}$,
ход нагоре

1	2	3	4	5	40
7	8	9	10	48	56
19	21	25	45	75	78
28	29	30	65	81	88
29	40	58	70	91	95
50	55	60	74	99	100

~~75 78~~
~~50 55~~
K=52

1 злжитната на насуп-
стната линий е $2n-1$, нач-
ин, наверо.
Пак $\theta(n)$ решени

I аргумент за
линейка (\mathcal{C}_n)
долната граница

ε	ε	ε	ε	ε	$k-\varepsilon$
ε	ε	ε	ε	$k-\varepsilon$	$k+\varepsilon$
ε	ε	ε	$k-\varepsilon$	$k+\varepsilon$	∞
ε	ε	$k-\varepsilon$	$k+\varepsilon$	∞	∞
ε	$k-\varepsilon$	$k+\varepsilon$	∞	∞	∞
$k-\varepsilon$	$k+\varepsilon$	∞	∞	∞	∞

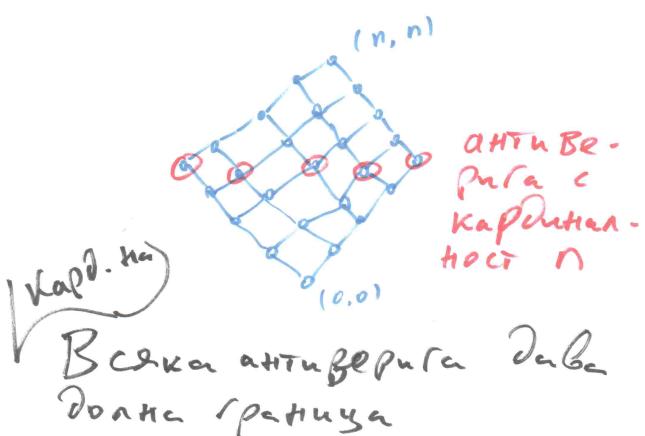
Търсим к (тъмна се!).

Ако го поставим дото
една клетка от "нагу-
ръната линия", прогнив-
никът ѝ пропадне на к.
Всеги са иначе!

Това се нарича \mathcal{C}_n .

II аргумент за
 $\Omega(n)$. Това е
 частична наредба.

Търсим в част. нар.
 Диаграма на Hasse
 на клетките:



Всяка антиверига дава
 долна граница

Лекция 8

Сортиране в линейно време. COUNTING SORT. RADIX SORT.

Резюме: Демонстрираме сортиране в линейно време върху вход, чиито елементи не са произволни, а се подчиняват на определени ограничения. Въвеждаме сортиращите алгоритми COUNTING SORT и RADIX SORT

8.1 Сортиране в линейно време

Както видяхме в предната лекция, сортирането на произволни елементи, чиито индивидуални стойности не можем четем, а само можем да сравняваме елементите един с друг, не може да става асимптотично по-бързо от $n \lg n$. Но тази долна граница не е в сила, ако има някакви ограничения възможните стойности на елементите. Съвсем прост пример е сортирането на булев масив – очевидно е достатъчно да преbroим колко са елементите от всеки вид и после да презапишем входа със съответния брой нули и единици, като нулите са преди единиците, което може да стане в $\Theta(n)$ време. Алгоритъмът, който ще разгледаме сега, се основава на идеята за пребояване на елементите от всяка големина, но както ще стане ясно, той третира елементите от входа не просто като числа, а като записи, чиито ключове са числа. Тъй като записите освен ключовете имат и някаква сателитна информация, сортирането става с местене на елементи (а не просто презаписване на входа с числа). Дори когато казваме “входът е масив от числа”, имаме предвид, че входът е масив от записи с ключове-числа.

8.2 COUNTING SORT

Входът е масив от числа $A[1, \dots, n]$. Известно е, че за някакво $k \in \mathbb{N}^+$, за всяко $A[i]$ е изпълнено

$$A[i] \in \{1, 2, \dots, k\}$$

Алгоритъмът ползва работен масив $C[0, 1, \dots, k]$ и още един масив $B[1, 2, \dots, n]$, в който се записва резултата от сортирането.

COUNTING SORT($A[1, 2, \dots, n]$): positive integers; k : a positive integer)

```

1 (* k е такова, че  $1 \leq A[i] \leq k$  за всички  $i$  *)
2 създай  $B[1, \dots, n]$  и  $C[0, 1, \dots, k]$ 
3 for  $i \leftarrow 0$  to  $k$ 
4    $C[i] \leftarrow 0$ 
5 for  $i \leftarrow 1$  to  $n$ 
6    $C[A[i]] \leftarrow C[A[i]] + 1$ 
7 for  $i \leftarrow 1$  to  $k$ 
8    $C[i] \leftarrow C[i] + C[i - 1]$ 
9 for  $i \leftarrow n$  downto 1
10   $B[C[A[i]]] \leftarrow A[i]$ 
11   $C[A[i]] \leftarrow C[A[i]] - 1$ 
```

Първо ще покажем работата на алгоритъма с пример и после ще дадем формално доказателство за коректност. Нека $n = 8$, $k = 6$ и масивът A е:

A	3	6	4	1	3	4	1	4
	1	2	3	4	5	6	7	8

Очевидно първият **for**-цикъл (редове 3–4) просто нулира масива C . Вторият **for**-цикъл (редове 5–6) преброява по колко елемента от всяко $i \in \{1, \dots, k\}$ има в A и записва това в C :

C	0	2	0	2	3	0	1
	0	1	2	3	4	5	6

Очевидно, $\sum_{i=0}^k C[i] = n$. Третият **for**-цикъл (редове 7–8) присвоява на всяко $C[i]$ броя на всички елементи на A , по-малки или равни на i :

C	0	2	2	4	7	7	8
	0	1	2	3	4	5	6

Очевидно, сега $C[k] = n$. Същината на алгоритъма е в четвъртия **for**-цикъл (редове 9–11). В началото $i = 8$, $A[8] = 4$, $C[4] = 7$. $B[7]$ става 4:

B							4	
	1	2	3	4	5	6	7	8

а C става:

C	0	2	2	4	6	7	8
	0	1	2	3	4	5	6

В червено е току-що намаленият елемент (заради ред 11) на C . После $i = 7$, $A[7] = 1$, $C[1] = 2$. $B[2]$ става 1:

B		1				4		
	1	2	3	4	5	6	7	8

а C става:

C	0	1	2	4	6	7	8
	0	1	2	3	4	5	6

После $i = 6$, $A[6] = 4$, $C[4] = 6$. $B[6]$ става 4:

B		1				4	4	
	1	2	3	4	5	6	7	8

а C става:

C	0	1	2	4	5	7	8
	0	1	2	3	4	5	6

И така нататък. В края на алгоритъма B е:

B	1	1	3	3	4	4	4	6
	1	2	3	4	5	6	7	8

а C е:

C	0	0	2	2	4	7	7
	0	1	2	3	4	5	6

В Определение 27 въведохме нотацията “ $\#(a, A)$ ” за кратността на a в мултимножеството A . Тук ще използваме нотацията “ $\#(x, Z[1, \dots, j])$ ”, за да означаваме кратността на x в мултимножеството от елементите на $Z[1, \dots, j]$, където Z е масив, j е индекс, сочещ в него, а x е елемент, чийто тип позволява да е елемент на Z .

Сега ща докажем коректността на COUNTING SORT.

Лема 36

След приключването на втория **for**-цикъл (редове 5–6), за всяко j , такова че $1 \leq j \leq k$, $C[j]$ съдържа броят на елементите в масива A , които са равни на j .

Доказателство:

Инвариант 12: Вторият **for**-цикъл на COUNTING SORT

Всеки път, когато изпълнението е на ред 5, за всеки елемент $C[j]$, където $1 \leq j \leq k$, е изпълнено $C[j] = \#(j, A[1, \dots, i - 1])$.

База. При първото достигане на ред 5, всички елементи на C са нули. От друга страна, i е 1, и така A[1, ..., i - 1] е празен. Твърдението е вярно.

Поддръжка. Да допуснем, че твърдението е в сила при дадено достигане на ред 5, което не е последното. Нека стойността на C[A[i]] е y в момента, в който изпълнението е на ред 6. По индуктивното предположение, $y = \#(A[i], A[1, \dots, i-1])$. Очевидно е, че $\#(A[i], A[1, \dots, i-1]) + 1 = \#(A[i], A[1, \dots, i])$. След изпълнението на ред 6, C[A[i]] се увеличава с единица, така че C[A[i]] става равно на $\#(A[i], A[1, \dots, i])$. Тъй като всички останали елементи на C (освен C[A[i]]) остават непроменени от текущото изпълнение на **for**-цикъла, е вярно, че:

- за всеки елемент C[j] освен C[A[i]], $C[j] = \#(j, A[1, \dots, i-1])$, а също така $C[j] = \#(j, A[1, \dots, i])$.
- $C[A[i]] = \#(A[i], A[1, \dots, i])$.

Като цяло, за всеки елемент C[j] е в сила $C[j] = \#(j, A[1, \dots, i])$. Това е преди инкрементирането на i. След инкрементирането на i, $C[j] = \#(j, A[1, \dots, i-1])$ за всяко j, такова че $1 \leq j \leq k$.

Терминация. Да разгледаме момента, в който изпълнението е на ред 5 за последен път. Очевидно i е $n+1$. Заместваме i с $n+1$ в инвариантата и получаваме “за всеки елемент C[j], където $1 \leq j \leq k$, е в сила $C[j] = \#(j, A[1, \dots, n])$.” \square

Лема 37

След приключването на третия **for**-цикъл (редове 7–8), за всяко j, такова че $1 \leq j \leq k$, C[j] съдържа броят на елементите в масива A, които са по-малки или равни на j.

Доказателство:

Инвариант 13: Третият **for**-цикъл на COUNTING SORT

Всеки път, когато изпълнението е на ред 7, за всяко j, такова че $0 \leq j \leq i-1$, $C[j] = \sum_{t=1}^j \#(t, A[1, \dots, n])$.

База. При първото достигане на ред 7, i е 1, така че твърдението е $C[0] = \sum_{t=1}^0 \#(t, A[1, \dots, n]) = 0$. Но C[0] е наистина 0, защото то се инициализира с 0 от първия **for**-цикъл и вторият **for**-цикъл не му присвоява нищо (понеже A[i] не може да е 0). \checkmark

Поддръжка. Да допуснем, че твърдението е в сила при някое достигане на ред 7, което не е последното. Елементът C[i] не е променян (засега) от изпълнението на третия **for**-цикъл, така че съгласно Лема 36, $C[i] = \#(i, A[1, \dots, n])$. По индуктивното предположение, $C[i-1] = \sum_{t=1}^{i-1} \#(t, A[1, \dots, n])$. След изпълнението на ред 8 имаме $C[i] = \sum_{t=1}^i \#(t, A[1, \dots, n])$. По отношение на новата стойност на i, за всяко j, такова че $0 \leq j \leq i-1$, $C[j] = \sum_{t=1}^j \#(t, A[1, \dots, n])$.

Терминация. Да разгледаме момента, в който изпълнението е на ред 7 за последен път. Очевидно i = $n+1$. Заместваме i с $n+1$ в инвариантата и получаваме “за всяко j, такова че $0 \leq j \leq n$, $C[j] = \sum_{t=1}^j \#(t, A[1, \dots, n])$.” \square

Определение 31

За всяко $j \in \{1, \dots, k\}$, j е съществено, ако съществува елемент на A със стойност j.

Съгласно Лема 36, ненулевите елементи на С след втория **for**-цикъл са точно елементите, чиито индекси в С са съществени.

Определение 32

За всеки x от A , правилното място на x е броят на елементите от A , които са по-малки от x , плюс броя на елементите равни на x , които са вляво от x , плюс едно.

С други думи, правилното място на x е позицията му в масива след изпълнението на стабилен сортиращ алгоритъм.

Теорема 22

COUNTING SORT е стабилен сортиращ алгоритъм.

Доказателство:

Инварианта 14: Четвъртият **for**-цикъл (редове 9–11)

Всеки път, когато изпълнението е на ред 9, за всяко j , такова че $1 \leq j \leq k$ и j е съществено, $C[j]$ съдържа правилното място на най-десния елемент от подмасива $A[1, \dots, i]$, който има стойност j . Нещо повече, всички елементи от $A[i+1, \dots, n]$ са на своите правилни места в B .

База. При първото изпълнение на ред 7, i е n . Първата част от инвариантата гласи “за всяко j , такова че $1 \leq j \leq k$ и j е съществено, $C[j]$ е правилното място на най-десния елемент от подмасива $A[1, \dots, n]$, който има стойност j ”. Забележете, че C все още не е променян от четвъртия цикъл, така че Лема 37 е в сила. За всяка стойност j , която се появява в A , правилното място на най-дясното j в A е равно на сумата от броевете на елементите със стойност $\leq j$. Според Лема 37, $C[j]$ е равно точно на тази сума.

Да разгледаме втората част от инвариантата. Подмасивът $A[i+1, \dots, n] = A[n+1, \dots, n]$ е празен, така че твърдението е в сила. ✓

Поддръжка. Да допуснем, че твърдението е в сила при някое достигане на ред 9, което не е последното. Стойността на $A[i]$ е някое съществено цяло число j между 1 и k . Нещо повече, то е най-десният елемент в $A[1, \dots, i]$ със стойност j . Съгласно индуктивното предположение, неговото правилно място е $C[A[i]]$ и алгоритъмът го копира точно там (ред 10).

Да допуснем, че има и други елементи със стойност j в $A[1, \dots, i]$. На ред 11, $C[A[i]]$ бива декрементирано, така че то вече съдържа правилното място на j в $A[1, \dots, i-1]$. След като i бъде декрементирано е вярно, че елементът на C , който току-що беше декрементиран съдържа правилното място на най-дясното j в $A[1, \dots, i]$. Тъй като всички останали елементи на C са непроменени, първата част от инвариантата е в сила.

Сега да допуснем, че в $A[1, \dots, i]$ няма други елементи със стойност j . Тогава j не е съществен по отношение на останалата част на A (която предстои да бъде сканирана от четвъртия цикъл), така че стойността на $C[A[i]]$ е без значение за алгоритъма оттук нататък. Наистина, след декрементирането на ред 11, $C[A[i]]$ сочи клетка в B , която е мястото на друг елемент (а не на j). Но, както казахме, стойността на това $C[A[i]]$ няма да бъде използвана до края на алгоритъма, понеже тази стойност на $A[i]$ няма да се среща повече. Първата част от инвариантата се запазва и в този случай.

Сега ще докажем втората част от инвариантата. Да разгледаме момента, когато изпълнението е на ред 9 в началото на текущата итерация. Съгласно индуктивното предположение, всички елементи от $A[i+1, \dots, n]$ са на своите правилни места в B . Току-що доказахме, че

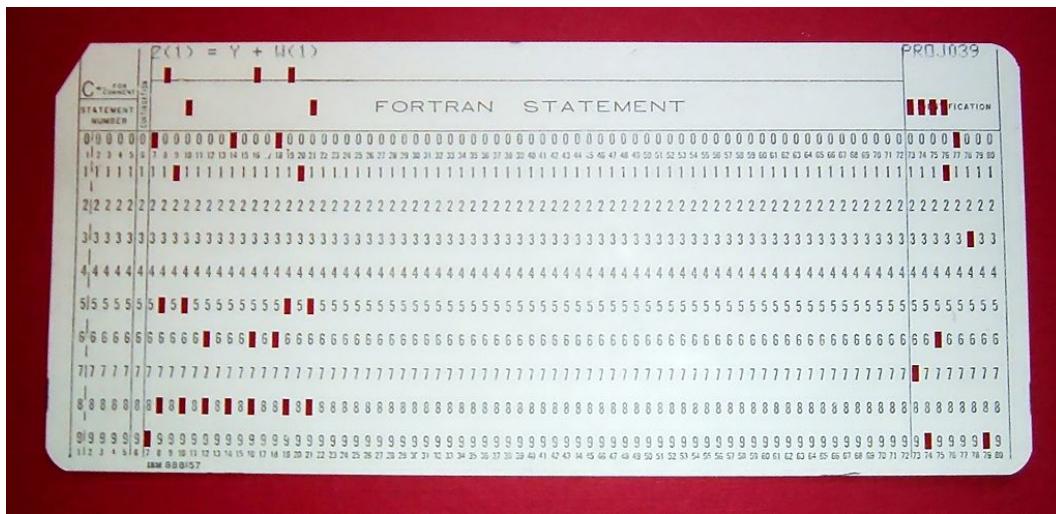
по време на това изпълнение на цикъла, елементът $A[i]$ бива копиран на правилното място. Тогава всички елементи от $A[i, \dots, n]$ са на своите правилни места в B . По отношение на новата стойност на i , вярно е, че всички елементи от $A[i + 1, \dots, n]$ са на своите правилни места в B .

Терминация. При завършването на цикъла, $i = 0$. Заместваме i с 0 във втората част на инвариантата и получаваме “всички елементи от $A[1, \dots, n]$ са на своите правилни места в B .” \square

Анализът на сложността е тривиален: COUNTING SORT работи във време $\Theta(n + k)$. Ако $k = O(n)$, както често се случва при използването на този алгоритъм, сложността по време е $\Theta(n)$. Това е значително подобрение спрямо $\Theta(n \lg n)$ на бързите сортиращи алгоритми, базирани на директни сравнения. Сложността по памет е същата: $\Theta(n + k)$, а ако $k = O(n)$, та е $\Theta(n)$. Следователно, алгоритъмът не е *in-place*.

8.3 RADIX SORT

RADIX SORT е алгоритъмът, използван за сортиране на перфокарти. Перфокарта, на английски *punched card* или *Hollerith card*, е цифров носител на информация, използван преди появата на реални компютри, и използван до 70те и дори 80те години на 20 век в компютрите. Работещ компютър, в който информацията се въвежда с перфокарти, в днешно време едво ли може да бъде намерен извън музей, но снимки на перфокарти се намират лесно по Интернет. Ето една такава снимка, взета от [wikipedia](#) с лиценз CC2.5; върху картата е написана в кодиран вид част от програма на Fortran:



Стандартната Холерит перфокарта има 80 колони, а във всяка колона може да бъде пробита дупка на точно едно измежду дванадесет възможни места. Всяка колона представлява символ. Ако символът е (десетична) цифра, дупката е на една от десетте позиции, маркирани с цифрите. Другите две позиции са за не-цифрова информация. Ако картата записва число, то може да има най-много 80 десетични разряда, колкото са колоните.

Задачата е, при дадено множество карти, които записват числа, картите да се сортират от електро-механично устройство, така че записаните числа да са в ненамаляващ ред. Електро-механичното устройство може да чете в даден момент точно един от десетичните разряди, с други думи, една от колоните, като допира вертикално разположени игли до картата и отчита коя игла потъва (през вече направената дупка), по този начин отчитайки коя е цифрата от съответния разряд.

Първата идея за сортиращ алгоритъм, който да управлява това електро-механично устройство, е да сортираме картите по най-старшия разряд, после по следващия, и така нататък, до най-младшия разряд. Недостатъкът на този подход е, че след сортирането по най-старши разряд трябва да държим картите в десет различни купчини[†] до края, след сортирането по втори най-старши разряд купчините ще станат сто[‡] и така нататък, което не е практично. Правилният подход е разрядите (колоните) да се тълкуват като ключове: старшият разряд е първичният ключ, вторият най-старши е вторичният, и така нататък, най-младшият разряд е 80-ичния ключ. Както знаем от лекцията по сортиране, ако сортираме цялата купчина (без да я разбиваме на подкупчини и подподкупчини и т. н.) първо по 80-ичния ключ, после от 79-ичния ключ, и така нататък, и най-накрая по първичния ключ, ще имаме правилно сортирани карти. Но: **при условие, че сортираме със стабилен сортиращ алгоритъм.** Стабилността гарантира, че когато сортираме по колона i , ако има няколко карти с една и съща цифра в колона i , техният взаимен порядък няма да се наруши; с други думи, вече извършените сортирания по колони $i+1, i+2$ и така нататък са определили правилното им взаимно разположение.

Ще дадем пример за работата на RADIX SORT. Ще сортираме числата 7444, 1320, 1692, 8185, 4007, 6281, 5139 и 7921. Представяме си, че са написани едно над друго в този ред, и започваме да сортираме от колоната на най-младшите разряди към колоната на най-старшите разряди. Колоната със сив фон отбелязва по кои разряди сортираме в момента.

7 4 4 4	1 3 2 0	4 0 0 7	4 0 0 7	1 3 2 0
1 3 2 0	6 2 8 1	1 3 2 0	8 1 8 5	1 6 9 2
1 6 9 2	7 9 2 1	7 9 2 1	5 1 3 9	4 0 0 7
8 1 8 5	1 6 9 2	5 1 3 9	6 2 8 1	5 1 3 9
4 0 0 7	7 4 4 4	7 4 4 4	1 3 2 0	6 2 8 1
6 2 8 1	8 1 8 5	6 2 8 1	7 4 4 4	7 4 4 4
5 1 3 9	4 0 0 7	8 1 8 5	1 6 9 2	7 9 2 1
7 9 2 1	5 1 3 9	1 6 9 2	7 9 2 1	8 1 8 5

Псевдокодът на RADIX SORT е съвсем прост (съгласно [15]):

RADIX SORT($A[1, \dots, n]$: положителни числа, записани с един и същи брой цифри; d : броят на цифрите)

- 1 (* най-младшата цифра е номер 1, най-старшата цифра е номер d *)
- 2 **for** $i \leftarrow 1$ **to** d
- 3 сортирай A по цифра i със стабилен сортиращ алгоритъм

Очевиден кандидат за стабилен сортиращ алгоритъм е COUNTING SORT. Коректността на RADIX SORT е очевидна, имайки предвид това, което знаем за стабилните сортирания. Ако използваме COUNTING SORT като стабилен сортиращ алгоритъм, сложността на RADIX SORT е $\Theta(d(n + k))$, където k е броят на възможните цифри. Ако k е константа, примерно 10, то сложността по време е $\Theta(dn)$.

Използвайки RADIX SORT, можем да решим следната задача: да се сортира в линейно време масив от n числа, всяко от което принадлежи на множеството $\{1, 2, \dots, n^2\}$. Ако опита-

[†]Купчините биха били десет, ако и десетте цифри се появяват като най-старша цифра на някоя карта.

[‡]Ще станат сто, ако и десетте цифри се появяват като и като най-старша цифра, и като втора най-старша цифра, на някоя карта. **Корекция на Емилиян Рогачев:** Купчините ще станат 100, ако всяка наредена двойка цифри се среща като начални 2 цифри.

ме директно да я решим с COUNTING SORT, решението ще работи в $\Theta(n^2)$, защото работният масив на COUNTING SORT ще бъде с размер n^2 . Но ако разгледаме записите на тези числа, примерно в двоична позиционна бройна система, ще видим, че всяко от тях се записва с $\lfloor \lg n^2 \rfloor + 1 = \lfloor 2 \lg n \rfloor + 1 \approx 2 \lg n$ бита. Ако разделим всеки бинарен запис на две части, всяка с по $\approx \lg n$ бита, може да смятаме дясната част за младши ключ, а лявата, за старши ключ. Прилагайки RADIX SORT (който на свой ред ползва COUNTING SORT) първо по младшите ключове, а после по старшите, имаме решение във време $\Theta(2n) = \Theta(n)$. Това, което ни помогна да конструираме линеен алгоритъм е, че прилагаме COUNTING SORT два пъти, но при всяко прилагане, работният масив е линеен, защото и младшите, и старшите ключове са с големина $\approx \lg n$, което означава, че могат да има най-много $\Theta(n)$ различни стойности.

Лекция 9

Въведение в алгоритмите върху графи. Обхождания на графи.

Резюме: Правим кратка рекапитулация на графите. Правим сравнителен анализ на най-популярните представления на графи. Разглеждаме основната алгоритмична задача върху графи: обхождане. Разглеждаме двата най-популярни алгоритъма за обхождане на графи: BFS и DFS.

9.1 Рекапитулация на графите

Определение 33: Граф

Граф е наредена двойка (V, E) , където V е непразно множество от *върховете*, а E е множество от *ребрата*. V е опорното множество (ground set), а E е подмножество на $\{\{x, y\} \mid \{x, y\} \subseteq V\}$.

Графите, дефинирани в Определение 33 се наричат *неориентирани*, тъй като техните ребра са множества—двуелементни множества от върхове, по-точно—а множествата нямат наредба. Ако кажем само “граф”, ще разбираме неориентиран граф.

Определение 34: Ориентиран граф

Ориентиран граф е наредена двойка (V, E) , където V е непразно множество от върховете, а E е множество от ребрата. V е опорното множество, а E е подмножество на $(V \times V) \setminus \{(v, v) \in V \times V\}$.

Определение 33 и Определение 34 не допускат *примки*; примка е ребро, на което двата края са един и същи връх. На читателя остава да прецени как да ги модифицира по такъв начин, че да се допускат примки.

Определение 35: Ориентиран мултиграф

Ориентиран мултиграф е наредена тройка (V, E, f_G) , където V е непразно множество от *върхове*, E е множество от *ребра*, а f_G е *свързващата функция*. V и E са опорни множества, а $f_G : E \rightarrow V \times V$.

Оттук насетне ще описваме мултиграфите без да споменаваме свързващата функция; примерно, $G = (V, E)$.

На читателя остава да дефинира “неориентиран мултиграф”. Когато кажем “мултиграф” без уточнения, ще разбираме ориентиран мултиграф. Под *обикновен граф* разбираме граф, който не е мултиграф.

Всички определения върху графи от курса по Дискретни Структури са в сила. Тук добавяме няколко определения и нотации, които не са изучавани по Дискретни Структури.

Нотация 5: $\text{SC}(G)$

Нека G е ориентиран граф. “ $\text{SC}(G)$ ” означава релацията на силна свързаност над $V(G)$.

Определение 36: Даг

Ако G е ориентиран ацикличен граф, ще казваме накратко, че G е *даг*.

Името идва от английския акроним “dag” (*directed acyclic graph*). Авторът на тези лекционни записи не знае общоприет български термин за “ориентиран граф без цикли”, нито харесва “огбц”, така че ще ползва “даг”.

Определение 37: арборесценция

На английски *arborescence* или *out-tree* се нарича ориентиран граф, който се получава от кореново дърво—което е неориентиран граф—с корен r , като на ребрата се даде ориентация навън от корена. Алтернативна дефиниция е, ориентиран граф G , в който съществува връх r , такъв че за всеки връх $u \in V(G)$ е вярно, че има един единствен ориентиран, не непременно прост, път от r до u .

Връх r е коренът на арборесценцията.

Думата “*arborescence*” произхожда от латинското *arbor*, което означава дърво. Авторът на тези лекционни записи не знае общоприет български термин за “ориентирано дърво с ориентация навън от корена”, така че по неволя ще използва “арборесценция”. Забележете, че всеки арборесценция е даг, докато обратното не е вярно.

Конвенция 4: Буквите m и n в контекста на графиките

Когато разглеждаме графи, неизменно буквата n ще има смисъл на броя на върховете, а m , на броя на ребрата, освен ако изрично не е казано нещо друго.

Наблюдение 26: Ограничения за m при дадено n

Ако разглеждаме обикновен неориентиран граф без примки, то $0 \leq m \leq \binom{n}{2}$, като границите са точни. Ако разглеждаме обикновен ориентиран граф без примки, то $0 \leq m \leq n(n - 1)$, като границите са точни. Ако разглеждаме обикновен ориентиран граф, който може да има примки, $0 \leq m \leq n^2$, като границите са точни. Ако разглеждаме мултиграф, няма горна граница за m , така че $0 \leq m$ е всичко, което знаем за m , ако не е дадено никакво ограничение в условието.

Основните представления на графи, които не са мултиграфи, са следните две. Нека графът е $G = (V, E)$, $V = \{u_1, \dots, u_n\}$ и $E = \{e_1, \dots, e_m\}$.

Списъци на съседство: За всеки връх е даден списък от неговите съседи, ако става дума за неориентиран граф, или списък от неговите деца, ако става дума за ориентиран граф. Списъците са точно n . Списъците **не са подредени по никакъв начин**, както един спрямо друг, така и всеки от тях вътрешно.

Матрица на съседство: Това е булева матрица $n \times n$, където клетка (i, j) съдържа 1 тогава и само тогава, когато

- има ребро **между** u_i и u_j , ако графът е неориентиран; в този случай матрицата е симетрична.
- има ребро **от** u_i **до** u_j , ако графът е ориентиран; в този случай матрицата не е непременно симетрична.

Мултиграфите се представят със списъци на съседство по гореописания начин. Има разлика в матричното представяне между обикновени графи и мултиграфи: ако искаме да представяме мултиграф с матрица, тя трябва да е не булева, а от естествени числа, като клетка (i, j) съдържа броя на ребрата от u_i до u_j .

Наблюдение 27

Ако неориентиран граф, който е мултиграф или не, е представен чрез списъци, всяко ребро (u_i, u_j) , което не е примка, се появява **два пъти**: веднъж в списъка на u_i и веднъж в списъка на u_j .

Размерът на граф е $m + n$, ако е представен чрез списъци. Това е в сила и за мултиграфи, и за графи, които не са мултиграфи. От друга страна, ако е представен чрез матрица, размерът е n^2 .

Конвенция 5

Тъй като използваме списъчното представяне по-често, по подразбиране размерът на граф е $m + n$. Следователно, по подразбиране, линеен алгоритъм върху графи е такъв, който работи във време $\Theta(m + n)$.

Нотация 6: $\text{adj}[x]$

Ако G е граф и x е връх в него, “ $\text{adj}[x]$ ” означава списъка на съседство на x в списъчното представяне на G .

Действия върху графи. Графите, които разглеждаме, понякога са статични, но понякога искаме да добавяме или да изтриваме техни елементи, в който случай ще казваме, че са **динамични**[†]. Естествено, в най-общия случай бихме искали да добавяме или изтриваме както върхове, така и ребра. На практика се оказва, че значително по-лесно е при **фиксирано** множество от върхове да се добавят или премахват ребра. Такива графи—с фиксирано множество от върхове, спрямо което добавяме или изтриваме ребра—се наричат **полудинамични**. Ние ще ограничим до действията добавяне на ребра и изтриване на ребра, което значи, че ако графиките, които разглеждаме, не са статични, те са полудинамични.

[†]Това няма **нищо общо** с динамичното програмиране, което ще разгледдаме в Лекция 13

Таблица 9.1 показва сложността в най-лошия случай на основни алгоритмични примитиви и алгоритми върху графи при матрично и при списъчно представяне. Допускаме, че е даден обикновен граф $G = (V, E)$, ориентиран или не, като $V = \{u_1, \dots, u_n\}$.

Действие	Представяне	
	чрез матрица	чрез списъци
тестване дали $(u_i, u_j) \in E$	$\Theta(1)$	$\Theta(n)$
намиране на $d(u_i)$	$\Theta(n)$	$\Theta(n)$
опит за добавяне на ребро (u_i, u_j)	$\Theta(1)$	$\Theta(n)$
опит за изтриване на ребро (u_i, u_j)	$\Theta(1)$	$\Theta(n)$
пребояване на ребрата	$\Theta(n^2)$	$\Theta(m + n)$
обхождане на графа	$\Theta(n^2)$	$\Theta(m + n)$

Таблица 9.1: Сложностите на действията върху графи при двете представяния.

Забележете, че ако графът е мултиграф, опит за добавяне на ребро отнема само $\Theta(1)$ време и при списъчното представяне, защото просто вмъкваме новото ребро в началото на списъка; в такъв случай има смисъл да говорим само за “добавяне на ребро”, а не за “опит за добавяне”, защото няма причина да не може да добавим ново ребро между/от-до съществуващи върхове. При обикновените графи такъв опит може да не е успешен, защото такова ребро може вече да има, оттам идва и сложността $\Theta(n)$ при списъчното представяне – налага се да минем през целия списък на u_i или u_j , за да се убедим, че няма ребро (u_i, u_j) .

Определение 38: Разредени графи

Нека \mathfrak{G} е безкрайно множество от графи. Казваме, че \mathfrak{G} са разредени графи, на английски sparse graphs, ако за всеки $G \in \mathfrak{G}$ имаме $|E(G)| = O(|V(G)|)$. По-подробно казано, съществува положителна константа c , такава че за всеки $G \in \mathfrak{G}$ имаме $|E(G)| \leq c|V(G)|$.

Забележете, че въпростът дали даден граф е разреден граф е **безсмислен въпрос**, говорейки формално. Съгласно Определение 38, въпростът е смислен само за безкрайни множества от графи, или *классове от графи*. Важни класове графи, които са разредени, са:

дървата. Знаям, че за всяко дърво $m = n - 1$.

планарните графи. Знаям, че за всеки планарен граф $m \leq 3n - 6$.

графите, в които максималната степен на връх е ограничена от константа. Знаям, че $\sum_{u \in V} d(u) = 2m$, което влече $\Delta(G)n \geq 2m$, а оттук $m \leq \frac{\Delta(G)}{2}n$. Да си припомним, че “ $\Delta(G)$ ” е максималната степен на връх в графа G .

Наблюдение 28: Размер на граф от клас от разредени графи

Ако \mathfrak{G} е клас от разредени графи, за всеки $G \in \mathfrak{G}$ е вярно, че размерът на G е $\Theta(n)$, ако използваме списъчно представяне. Ако използваме матрично представяне, размерът е $\Theta(n^2)$. Еrgo, списъчното представяне е особено удачно за разредени графи.

Дали неориентираните графи са всъщност ориентирани? Забележете, че за всеки неориентиран граф, представен чрез списъци, можем да мислим като за ориентиран граф, чиито ребра са “сдвоени” в смисъл, че всяко неориентирано ребро (u, v) **всъщност** е двойка ориентирани ребра (u, v) и (v, u) . Ако възприемем тази гледна точка, можем да заключим, че **всъщност** неориентирани графи няма – всички графи са само ориентирани, а понякога на тези със “сдвоените” ребра казваме “неориентирани графи”. От тази гледна точка, понятието “неориентиран граф” е само метафора, или начин за удобно изразяване, докато на ниско ниво графиките са само ориентирани. Ако гледаме само представянето със списъци, това е точно така.

Ние обаче **няма** да възприемаме тази гледна точка. Това, че представяме неориентирани графи като ориентирани със сдвоени ребра, съвсем не означава, че всъщност неориентирани графи няма. Ето някои от проблемите, до които би довело решението да отречем съществуването на неориентирани графи.

- Веднага ще следва, че няма нетривиални ациклични неориентирани графи – ако има поне едно неориентирано ребро, то всъщност е двойка ребра, които образуват ориентиран цикъл. Тогава понятието “дърво” се обезсмисля. Но интуитивно е ясно, че има гигантска разлика между “истински” неориентиран цикъл и “ерзац” цикъл от сдвоени ориентирани ребра. И че “дърво” е смислено и важно понятие.
- Алгоритми като DFS, които класифицират ребрата на графа по някакъв начин, ще класифицират двете ориентирани ребра (u, v) и (v, u) в различни класове. Но – ако става дума за неориентиран граф – бихме искали това да е само едно ребро и то да бъде класифицирано в един клас.
- Някои алгоритмични задачи, които по принцип са върху ориентирани графи, може да бъдат разгледани и върху неориентирани графи в някои случаи. Например, Обхождане на ГРАФ или Най-къс път в ГРАФ[†]. Но задачите върху неориентирани графи като Минимално покриващо дърво или Върхово покриване или Доминиращо множество са задачи именно върху неориентирани графи. Те или не може да се обобщят естествено върху ориентирани графи, или обобщенията им върху ориентирани графи имат решения със съвсем друга сложност. Например, обобщението на Минимално покриващо дърво върху ориентирани графи е NP-пълната задача Minimum Equivalent Digraph[‡] [20, стр. 65].

9.2 Обхождания на графи

Обхождането на графи е базова изчислителна задача. На английски терминът е *graph traversal*. Понякога се използва и *graph search*, но това звути прекалено близо до *graph searching*, което означава съвсем друга задача. Много неформално казано, обхождането на граф се извършва от някакво същество, което живее в графа; тръгвайки от **даден начален връх**, то иска да да се разходи из графа по такъв начин, че да обиколи максимална част (подграф) на графа.

Съществото обхожда както върхове, така и ребра. Формулировката на задачата, която казва, че биват обхождани върхове, без да се споменават ребра, е **некоректна**. Прочее, ако

[†]Както ще видим в Лекция 12, Най-къс път в ГРАФ върху неориентирани графи е безсмислена при наличие на отрицателни тегла, докато върху ориентирани графи има смисъл дори при отрицателни тегла, стига да няма отрицателни цикли.

[‡]MED е дефинирана за ориентирани графи, които не са тегловни. Ясно е, че ако добавим и тегла, задачата няма да стане по-лесна.

няма изолирани върхове, достатъчно е да кажем, че се обхождат ребрата на графа; при липса на изолирани върхове обхождането на ребрата влече и обхождане на върховете.

Задачата се формулира за ориентирани мултиграфи с възможни примки. Това, че ще я разглеждаме и отделно като задача за неориентирани мултиграфи, не променя факта, че общата ѝ формулировка е за ориентирани мултиграфи. Паралелните ребра и примките имат значение: те трябва да бъдат обходени, всяко/всяка поотделно.

Удачно е да мислим за графа като за *лабиринт*, състоящ се от *стай* и *коридори*, свързващи стаи; даден коридор може да свързва стая със себе си и такива коридори отговарят на примките в графа. Коридорите са еднопосочни, ако графът е ориентиран, или двупосочни, ако е неориентиран. Съществото иска да обиколи максимална част от лабиринта, минавайки през всеки коридор и посещавайки всяка стая, без да пропуска коридори/стай и без да "зацикли".

За да се избегне зацикляне се използва *маркировка*. Противно на житейската интуиция, според която съществото би маркирало коридорите, за да знае къде вече е било и да не зацикли, в алгоритмите-решения на тази задача се маркират **върховете**.

- В простия вариант всеки връх има две състояния, иначе казано, бива маркиран по точно един от следните два начина: **посетен** и **непосетен**. В началото всички върхове са непосетени, с изключение на стартовия връх; в неформалното обяснение с лабиринта, всички стаи в началото са непосетени с изключение на стартовата стая.

Можем да мислим, че се използват *цветове*, като непосетените стаи/върхове са бели, а посетените, черни. В началото всички стаи без стартовата са *бели*, а стартовата е *черна*. След това, когато съществото посети бяла стая, то разбира, че не е било в нея досега, и я маркира, боядисвайки я в черно, за да знае, че вече е било в нея при повторно попадане.

- В по-изтънчените решения се ползат три цвята, тоест, всяка стая има три състояния. Белите, както и в другия вариант, са непосетените. Тези, които са посетени, но с тях не е приключено, са *сиви*. Тези, с които е приключено, са черни. Да не бъде приключено с посетена стая означава, че от нея излизат коридори, по които (може би) не сме минавали. Със стая приключваме, когато сме убедени, че от нея не излизат коридори, по които не сме минавали.

Зашо маркираме върховете, а не ребрата? За да пестим памет. За да маркираме върховете ни е необходима и достатъчна $\Theta(n)$ памет, тъй като за всеки връх има $\Theta(1)$ различни възможности (бял, сив и черен, или само бял и черен в по-простия вариант). За да маркираме ребрата ни е необходима и достатъчна $\Theta(m)$ памет, дори да ползваме само един бит за ребро. Както вече видяхме, m може да е квадратично по-голямо от n при графи, които не са мултиграфи, а върху мултиграфи m може да е произволно по-голямо от n .

Аналогията с лабиринта може да е подвеждаща. За реално обхождане на физически лабиринт, реализирано по BFS или DFS идеята, че е необходимо да се измине много по-голямо разстояние (заради повтаряне на коридори), отколкото налагат нашите BFS и DFS. За пълна аналогия с обхождане на лабиринт трябва да допуснем, че съществото може да се телепортира от стая в друга стая, но само при условие, че вече е било в другата стая. Такова обхождане много точно съответства на обхождането, което следва от BFS и DFS.

В неориентиранныте графи има една особеност: всяко ребро се обхожда по точно два пъти, дори да е примка. Причината е, че маркираме не ребрата, а върховете. В аналогията с лабиринта, ходейки по някакъв коридор, ние няма как да знаем, че сме били в него. Това, че минаваме за втори път по даден коридор става ясно едва когато излезем от него и установим,

че сме попаднали в сива или черна стая. При ориентираните графи ребрата се обхождат точно по веднъж.

Какво обхождаме в неориентиран граф, стартирайки от даден връх. Ако графът, мултиграф или не, е неориентиран, при обхождане, започващо във връх u , обхождаме точно свързаната компонента, на която принадлежи u . Както знаем, всеки връх в неориентиран граф се намира в точно една свързана компонента, а обхождането няма как да “излезе” от нея. За да обходим целия граф се налага да рестартираме обхождането върху всяка свързана компонента.

Какво обхождаме в ориентиран граф, стартирайки от даден връх. Ако обаче графът е ориентиран, въпросът какво точно ще обходим, стартирайки от връх u , е по-триков. Ако е силно свързан, задължително ще го обходим целия. Ако всяка от слабо свързаните му компоненти е силно свързана, задължително ще обходим цялата слабо свързана компонента, на която принадлежи u . Ако обаче слабо свързаната компонента, на която принадлежи u , не е силно свързана, може:

- в един екстремен случай, да я обходим цялата,
- в друг екстремен случай, да обходим само u , без да може да излезем от него,
- и в общия случай, да обходим само част от нея. Естествено, не е възможно да се прехвърляме от слабо свързана компонента в друга слабо свързана компонента, понеже няма ребро от никой връх на едната от тях до никой връх на другата, но каква част от слабо свързаната компонента, на която принадлежи u , е силно зависещо от конкретиката на графа.

С извинение за тавтологичния изказ: стартирайки от u , ще обходим точно този подграф, който е достъпим от u . Следното определение прецизира това.

Определение 39: достъпим подграф

Нека G е ориентиран мултиграф с възможни примки и u е връх в него. *Подграфът на G , достъпим от u* е подграфът на G , индуциран от $\{v \in V(G) | \text{съществува път от } u \text{ до } v\}$.

9.3 Обща схема за обхождане

Ще опишем общата схема, която следват алгоритмите за обхождане, които ще разгледаме. За ограниченияте цели на тази секция, върховете имат само две състояния: необходен и обходен. Белите върхове са необходените, черните са обходените. Схемата е една и съща за всички видове графи, които разглеждаме, така че допускаме, че е даден ориентиран мултиграф с възможни примки $G = (V, E)$. Даден е и стартов връх u .

Идея за обхождане. В началото всички върхове са бели с изключение на u . Поддържаме множество от върхове S . В началото $S = \{u\}$. Обхождането е итеративно, като на всяка итерация изваждаме един връх от S и слагаме в променлива (тип връх) x . След това разглеждаме всички ребра, излизящи от x , което означава—при списъчно представяне на графа—че минаваме през списъка на x . За всяко ребро, да го наречем (x, y) :

- ако връх u е бял, правим u черен и го слагаме в S ,
- в противен случай прескачаме u .

Това приключва, когато S стане празно.

Самото обхождане на ребрата става при тези минавания през списъците. Ако беше даден лабиринт и трябваше да намерим нещо в коридорите му—изхода, някакво съкровище или [Минотавъра](#)—щяхме да го сторим по този начин.

Доказателство за коректност на идеята за обхождане. Ще покажем, не особено прецизно, че тази идея (защото това е прекалено общо описание, за да бъде алгоритъм) работи. Обхождането може да събърка по два начина: да зацикли или да пропусне. Нашето обхождане не зацикли, защото в S влизат само бели върхове, които веднага с влизането стават черни; веднъж станали черни, те остават черни. Ще покажем, че не може и да пропусне. Без ограничение на общността, нека (мулти)графът е силно свързан[†]. Да допуснем, че поне един връх v остава бял, тоест, непосетен, след края на обхождането. Но в графа има ориентиран път p от u до v . В края на алгоритъма u е черен, защото бива направен черен поначало, а v е бял по допускане. Всеки връх от p е или бял, или черен. Щом началото на p е черен връх, а краят му е бял връх, то в p има върхове a и b , такива че a е родител на b и a е черен и b е бял. Очевидно, в даден момент от обхождането, a е бил изведен от S (щом $S = \emptyset$ накрая) и всички деца на a са били направени черни и сложени в S . Тогава няма как b да е бил пропуснат и да е останал бял. \ddagger

Ако множеството S бъде реализирано чрез АТД опашка[‡], тази схема за обхождане се превръща в алгоритъма BFS, който ще разгледаме първо. Ако S бъде реализирано чрез стек, получаваме алгоритъм, близък до DFS, който ще разгледаме след BFS.

9.4 BFS

Името на този алгоритъм идва от Breadth-First Search, където “search” е синоним на “traversal”. При него, границата между посетените и непосетените върхове—образно и непрецизно казано—расте равномерно във всички посоки спрямо стартиния връх.

Отново допускаме, че е даден ориентиран мултиграф с възможни примки, а някои особености на алгоритъма върху неориентирани (мулти)графи ще разгледаме отделно.

Винаги е даден е стартичен връх, който ще наричаме s . Но ако не е вярно, че целият граф е достъпим от s , има две възможности.

1. BFS спира след като обходи подграфа, който е достъпим от s .
2. След като обходи подграфа, който е достъпим от s , BFS бива рестартиран с нов начален връх измежду непосетените досега върхове, и така нататък, докато целият граф бъде обходен.

Приемете, че върховете са числата $1, 2, \dots, n$.

[†]Ако не е, аргументът се отнася само за подграфа, достъпим от u .

[‡]Да си припомним Подсекция 4.4.1. Опашка е именно АТД, защото се дефинира чрез интерфейса си от Enqueue и Dequeue, а имплементацията остава скрита зад този интерфейс. Аналогично, стек е АТД с интерфейс push и pop.

Цветове на върховете. Всеки връх на графа във всеки момент от работата на BFS има едно от следните три състояния: *непосетен*, *посетен*, но *неприключен*, и *приключен*, със съответните цветове бял, сив и черен. Тези цветове се реализират от масив $\text{color}[1, \dots, n]$, където $\text{color}[i]$ е текущият цвят на връх i ; $\text{color}[i] \in \{\text{white}, \text{gray}, \text{black}\}$.

Дърво или гора на обхождането. Ако целият граф е достигим от s или BFS спира изчерпване на достигими от s непосетени върхове (възможност 1), BFS строи *дърво на обхождането* T . Това T е арборесценция (вижте Определение 37) с корен s , чиито върхове са точно върховете, достигими от s . В T се намират точно тези ребра на G , с които BFS открива бели върхове.

Ако BFS се рестартира, докато не обходи целия граф (възможност 2), BFS строи колекция от арборесценции T_1, \dots, T_k , по една за всяко рестартиране, като корените им са стартовите върхове на всяко от пусканията на BFS и всеки връх на графа е в точно една от T_1, \dots, T_k . Казваме, че колекцията $\{T_1, \dots, T_k\}$ е *гората на обхождането*.

В алгоритъма ни дървото или гората на обхождането се реализира с масив на предшествията $\pi[1, \dots, n]$, в който за всеки връх i , който не е корен, съответният елемент $\pi[i]$ е родителят на i ; ако i е корен, то $\pi[i] = \text{Nil}$.

Разстояния спрямо стартовия връх. Във възможност 1, BFS може да изчисли разстоянията от s до всеки друг връх. Забележете изказа с “от-до”. Става дума за разстояния в ориентирания смисъл.

Определение 40: разстояние в ориентирания смисъл

Нека G е ориентиран граф. Дали има или няма паралелни ребра или примки е без значение. За всеки два върха $u, v \in V(G)$, *разстоянието в ориентирания смисъл от u до v* , което означаваме с $\delta(u, v)$, е дължината на най-къс ориентиран път от u до v , ако такъв има, или ∞ , ако такъв няма. Ако от контекста е ясно, че разстоянието е в ориентирания смисъл, казваме кратко *разстоянието от u до v* .

Ако е даден и никакъв подграф H на G , то с $\delta_H(u, v)$ означаваме разстоянието от u до v в H . Това е дължината на най-къс ориентиран път от u до v , състоящ се само от ребра от $E(H)$. Ако такъв няма, то $\delta_H(u, v) = \infty$.

Очевидно е, че $\delta(u, v) \leq \delta_H(u, v)$.

Съществената разлика между разстоянието в ориентирания смисъл и разстоянието в неориентирания смисъл (в неориентирани графи) е, че разстоянието в ориентирания смисъл не е непременно симетрично. Тривиално е да се измислят примери, в които $\delta(u, v) \neq \delta(v, u)$, като дори може едното да е число, а другото да е ∞ . Поради това Определение 40 ползва изказа “от-до”, а не “между”, който ползваме при симетричните разстояния в неориентирания смисъл.

Нататък ще покажем, че BFS наистина изчислява разстоянията в ориентирания смисъл. Тук само отбеляваме, че BFS записва тези разстояния в масив $d[1, \dots, n]$, където $d[i]$ след терминирането е равно на $\delta(s, i)$, за всеки връх i .

Опашката от сивите върхове. Ще разгледаме итеративен BFS, в който опашка Q съдържа текущите сиви върхове, а на всяка итерация се изважда един връх x от Q с $\text{dequeue}(Q)$ — стига Q да не е празна преди това — и се обхождат всички излизачи от x ребра, като при това всеки открит бял връх y , дете на x , бива направен сив и вкаран в Q с $\text{enqueue}(Q, y)$. Алгоритъмът спира, когато Q стане празна. Очевидно интерфейсът на абстрактния тип данни опаш-

ка, за целите на нашия алгоритъм, се състои от трите функции `isempty(Q)`, `enqueue(Q, y)` и `dequeue(Q)`, като имената са индикативни за работата им.

9.4.1 Псевдокод на BFS от един стартов връх

$\text{BFS}(G = (V, E))$: ориентиран мултиграф с възможни примки, като $V = \{1, \dots, n\}; s \in V$)

```

1   for i ← 1 to n
2     color[i] ← white
3     d[i] ← ∞
4     π[i] ← Nil
5   color[s] ← gray
6   d[s] ← 0
7   създай опашка Q
8   enqueue(Q, s)
9   while not isempty(Q) do
10    x ← dequeue(Q)
11    for y ∈ adj[x]
12      if color[y] = white
13        color[y] ← gray
14        d[y] ← d[x] + 1
15        π[y] ← x
16        enqueue(Q, y)
17    color[x] ← black

```

Коректност на BFS. Това, че BFS терминира върху всеки вход, е очевидно: нови върхове влизат в опашката само ако са бели, веднъж влезли в опашката, те вече не са бели, а не-бял връх не може да стане бял; тъй като на всяка итерация се вади връх от опашката, рано или късно тя ще стане празна и BFS ще спре.

В Теорема 23 и помощните леми, нека $G = (V, E)$ е произволен ориентиран мултиграф и s е произволен връх в него. Нека G' е подграфът на G , достигим от s .

Лема 38

При терминирането на $\text{BFS}(G, s)$ няма сиви върхове.

Доказателство: Следното твърдение е инвариант за **while** цикъла (редове 9–17).

Инварианта 15: Цикълът на BFS (1)

Всеки път, когато изпълнението е на ред 9, множеството от сивите върхове е точно множеството от върховете в опашката Q .

База. При първото достигане на ред 9 твърдението е вярно, понеже на ред 7 е създадена празна опашка, а на ред 8 в нея е сложен (само) връх s . От друга страна, в този момент s е единственият сив връх.

Поддръжка. Да допуснем, че твърдението е в сила в даден момент, в който изпълнението е на ред 9 и **while** ще бъде изпълнен поне още веднъж. Последното влече, че Q не е празна.

По допускане, всички сиви върхове са в Q и в Q има само сиви върхове. Един сив връх бива изваден от Q на ред 10, но този връх става черен на ред 17. По време на изпълнение на тялото на цикъла, всяко бяло дете на този връх става сиво (ред 13), но след това то влиза в опашката на ред 16. Очевидно е, че при следващото достигане на ред 9 твърдението е вярно.

Терминация. При последното достигане на ред 9 опашката Q е празна. Това означава, че няма сиви върхове. \square

Доказателството на Лема 39 е практически същото като доказателството за коректност на идеята на обхождането на стр. 267.

Лема 39

При терминирането на $\text{BFS}(G, s)$:

- за всеки $v \in V(G')$ е вярно, че $\text{color}[v] = \text{black}$ и $d[v] < \infty$;
- за всеки $v \in V \setminus V(G')$ е вярно, че $\text{color}[v] = \text{white}$ и $d[v] = \infty$.

Доказателство: Разглеждаме момента, в който BFS терминира. От Лема 38 знаем, че при терминирането на BFS сиви върхове няма. Очевидно е, че белите върхове са точно тези, чиято d -стойност е ∞ , а черните са тези, чиято d -стойност не е ∞ . Очевидно е също така, че всички върхове от $V \setminus V(G')$ са бели. Остава да докажем, че всички върхове от $V(G')$ са черни.

Да допуснем противното. Тогава съществува връх $u \in V(G')$, който е бял в разглеждания момент. Щом u е връх в G' , съществува път p от s до u . Очевидно $u \neq s$, тъй като в този момент s е черен. Щом началото на p е черен връх, а краят му е бял връх, съществува ребро (a, b) в p , такова че a е черен и b е бял. Но щом a е черен, той е станал черен на ред 17 в някакъв момент t , в който променливата x е имала съдържание a . Преди a да стане черен, всяко негово дете y е било “прегледано” на ред 11. Но b е дете на a , щом има ребро (a, b) , така че в някой момент $\hat{t} < t$ е било вярно, че $y = b$. Тъй като цветът на $y = b$ е бил и в момента \hat{t} , булевото условие на ред 12 е било TRUE и връх b трябва да е станал сив на ред 13. Веднъж престанал да бъде бял, той няма как да се окаже бял при терминиране на BFS. \ddagger

Лема 40

При терминирането на $\text{BFS}(G, s)$, за всеки $v \in V$ е вярно, че $d[v] \geq \delta(s, v)$.

Доказателство: Ще докажем твърдението само за върховете от $V(G')$, тъй като онези от $V \setminus V(G')$ не са достижими от s , така че $\forall v \in V \setminus V(G') : d[v] = \infty$ в края на алгоритъма, а $\delta(s, v) = \infty$ по дефиниция. Следното твърдение е инвариантна за **while** цикъла (редове 9–17).

Инвариант 16: Цикълът на BFS (2)

Всеки път, когато изпълнението е на ред 9, за всеки връх $v \in V(G')$ е вярно, че $d[v] \geq \delta(s, v)$.

База. При първото достигане на ред 9 разглеждаме поотделно s и останалите върхове.

- Да разгледаме s . Тъй като $d[s] = 0$ (ред 6) и $\delta(s, s) = 0$ (от теорията на графите), неравенството $d[s] \geq \delta(s, s)$ е в сила. ✓

- Да разгледаме $V \setminus \{s\}$. За всеки $v \in V \setminus \{s\}$ е вярно, че $d[v] = \infty$ заради присвояването на ред 3. Тогава $d[v] \geq \delta(s, v)$ независимо от това дали v е досяжим от s или не; с други думи, дали $\delta(s, v)$ е различна от или равна на ∞ .

Поддръжка. Да допуснем, че твърдението е в сила в даден момент, в който изпълнението е на ред 9 и **while** ще бъде изпълнен поне още веднъж. Последното влече, че Q не е празна.

На ред 10 изваждаме връх x от непразната опашка Q . Ефектът от работата на редове 11–17 очевидно е следният:

- всяко бяло дете y на x бива “обработено” на редове 13–16. $d[y]$ получава стойност $d[x] + 1$ на ред 14. По допускане, в този момент е вярно, че

$$d[x] \geq \delta(s, x)$$

Оттук

$$d[x] + 1 \geq \delta(s, x) + 1$$

което е същото като

$$d[y] \geq \delta(s, x) + 1$$

Но $\delta(s, x) + 1 \geq \delta(s, y)$, понеже, щом има път p от s до x , има и път от s до y с дължина $|p| + 1$ [†]. Тогава

$$d[y] \geq \delta(s, y)$$

Доказвахме, че за всеки връх y , който е бил “обработен” при изпълнение на текущата итерация, $d[y] \geq \delta(s, y)$.

- Сивите и черните деца на x не биват “обработвани” и техните d -стойности не биват променяни в тялото на цикъла (редове 13–16), така че за тях неравенството остава в сила. Аналогично, неравенството остава в сила за върховете на G' , които не са деца на x .

Терминация. При последното достигане на ред 9 за всеки $v \in V(G')$ е вярно, че $d[v] \geq \delta(s, v)$, което и трябва да покажем. \square

Лема 41 казва, че във всеки момент от изпълнението на BFS, в опашката има върхове или с една и съща d -стойност, или с точно две различни, но съседни по големина, d -стойности. С други думи, опашката “обработва” върховете по d -стойности. Което, както ще видим в Теорема 23, е същото като да “обработва” върховете по ориентиранияте разстояния от s .

Лема 41: (Lemma 22.3 в [15, стр. 599])

Нека в произволен момент t от работата на $BFS(G, s)$, опашката Q съдържа следната перmutация на върхове: $\langle u_1 u_2 \cdots u_k \rangle$, като u_1 е най-отдавна сложения връх, а u_k е най-скоро сложения връх. Тогава

$$d[u_1] \leq d[u_2] \leq \cdots \leq d[u_k] \leq d[u_1] + 1$$

[†]В [15] това очевидно твърдение е Lemma 22.1 на стр. 598.

Доказателство: Ще докажем лемата по индукция по броя на достиганията на ред 9. Няма смисъл да формулираме инвариантна, понеже не се интересуваме от крайното състояние на нещата при терминирането, а разглеждаме произволен момент от изпълнението – така че формулировката на лемата е инвариантата.

Базата на индукцията е първото достигане на ред 9. В този момент опашката съдържа само s и твърдението със сигурност е вярно.

Разглеждаме произволно достижане на ред 9, което не е последното. Нека това е момент t от изпълнението на BFS. Щом това достижане на ред 9 не е последното, опашката Q не е празна в момент t . Да кажем, че Q съдържа пермутацията на върхове $\langle u_1 u_2 \dots u_k \rangle$. На ред 10 връх u_1 бива изваден от Q и сложен в променливата x . Вече опашката съдържа пермутацията $\langle u_2 u_3 \dots u_k \rangle$, така че неравенствата по отношение на върховете в опашката стават

$$d[u_2] \leq \dots \leq d[u_k] \leq d[u_1] + 1$$

Имайки предвид това, че $d[u_1] \leq d[u_2]$, имаме $d[u_1] + 1 \leq d[u_2] + 1$, откъдето следва, че

$$d[u_2] \leq \dots \leq d[u_k] \leq d[u_2] + 1$$

Еrgo, твърдението остава в сила веднага след изваждането на връх от опашката.

Да се убедим, че твърдението остава в сила след вкарането на връх в опашката във вътрешния цикъл (редове 11–16). Разглеждаме произволен връх u' , който е дете на текущия x (ред 11) и е бял. Помним, че $x = u_1$. Рано или късно, променливата y получава съдържание u' и този връх бива сложен в опашката на ред 16 в някакъв момент $\hat{t} > t$. Преди да бъде сложен в опашката, връх u' получава d -стойност на ред 14. А именно, $d[u'] \leftarrow d[u_1] + 1$. Текущият първи връх в опашката е u_2 , а вече видяхме, че $d[u_1] + 1 \leq d[u_2] + 1$. Оттук следва, че $d[u'] \leq d[u_2] + 1$. Но съдържанието на опашката в момент \hat{t} е $\langle u_2 u_3 \dots u' \rangle$, така че твърдението остава в сила.

Очевидно твърдението остава в сила дори когато не се добавят нови върхове в Q , а също така и когато Q е празна; ако Q е празна, твърдението е вярно в празния смисъл (vacuously). \square

Теорема 23: коректност на BFS (Theorem 22.5 в [15, стр. 599])

След терминирането на $BFS(G, s)$:

1. Обходеният подграф на G е точно G' .
2. За всеки връх $v \in V(G')$, $d[v] = \delta(s, v)$.
3. Дървото на обхождането, реализирано чрез $\pi[1, \dots, n]$, е дърво на най-къси пътища за G' с корен s .

Доказателство: Твърдение 1 следва директно от Лема 39.

Ще докажем 2. Да допуснем, че съществува връх $u \in V(G')$, такъв че $d[u] \neq \delta(s, u)$ след терминирането на $BFS(G, s)$. Нека v е такъв връх, като освен това стойността $\delta(s, v)$ е минимална. Очевидно $v \neq s$. Съгласно Лема 40, $d[v] \geq \delta(s, v)$. От това и допускането, че $d[v] \neq \delta(s, v)$ следва, че $d[v] > \delta(s, v)$.

Щом $v \in V(G')$, съществува път p от s до v с дължина поне единица. БОО, нека p е път от s до v с минимална дължина. Тогава $|p| = \delta(s, v)$. Нека w е родителя на v в p . Нека подпътя на p от s до w се назова q . Забележете, че $|p| = |q| + 1$.

Известно е, че най-къс път се състои от най-къси подпътища (Теорема 39). Тогава q е най-къс път от s до w , така че $|q| = \delta(s, w)$. От това следва, че $\delta(s, v) = \delta(s, w) + 1$.

Но $\delta(s, v) > \delta(s, w)$ и v е избран като връх-нарушител с минимална d -стойност. Тогава w не е “нарушител”; тоест, $d[w] = \delta(s, w)$. Изведените дотук факти изглеждат така на един ред:

$$d[v] > \delta(s, v) = \delta(s, w) + 1 = d[w] + 1 \quad (9.1)$$

Нека t е моментът от работата на BFS, в който w бива изваден от опашката (ред 10). С други думи, променливата x получава съдържание w в момент t . Следните възможности за цвета на v в момент t са изчерпателни.

color[v] = white Тогава при изпълнението на **for**-цикъла (редове 11–16) в даден момент $\hat{t} > t$, променливата y ще получи съдържание v . При присвояването на ред 14 с $y = v$ е изпълнено $d[v] = d[w] + 1$. Но това е в противоречие с (9.1). ↴

color[v] = gray В такъв случай v е бил оцветен в сиво на някоя предишна итерация на **while**-цикъла, когато върха в променливата x е бил някой $a \in V(G')$, като v е бил y и е получил d -стойността си като $d[v] \leftarrow d[a] + 1$. Но този a влиза в опашката преди w , от което следва, съгласно Лема 41, че $d[a] \leq d[w]$, което е същото като $d[a] + 1 \leq d[w] + 1$. Тогава $d[v] \leq d[w] + 1$, в противоречие с (9.1). ↴

color[v] = black Щом v е черен, той вече е бил в Q и е бил изваден оттам. Тогава v бива изваден от Q преди w , което влече, че v влиза в Q преди w . Съгласно Лема 41, $d[v] \leq d[w]$, в противоречие с (9.1). ↴

Ще докажем 3. Нека дървото на обхождането, реализирано чрез масива на родители $\pi[1, \dots, n]$, се назава T . Твърди се, че за всеки връх $v \in V(G')$, $\delta(s, v) = \delta_T(s, v)$. Тривиално е да докажем това по индукция по реда на влизане на върховете в опашката.

Базата е за $v = s$ и твърдението очевидно е вярно. Нека $v \neq s$. v получава своите d - и π -стойности на някоя итерация на **while**-цикъла, променливата y има стойност v , а променливата x има някаква стойност u , където u е родител на v в G' и е родителят на v в T . А именно, $d[v]$ става $d[u] + 1$ и $\pi[v]$ става u . Тъй като u влиза в опашката преди v , по индукционното предположение, пътят от s до u в T е най-къс път от s до u в G . Имайки предвид това и факта, че $\delta(s, v) = \delta(s, u) + 1$, виждаме, че след присвояването $\pi[v] \leftarrow u$, масивът π реализира най-къс от s до v , така че $\delta(s, v) = \delta_T(s, v)$. □

Сложност по време. Реализацията на BFS, която разглеждахме, работи във време $\Theta(n + m)$, ако мултиграфът е реализиран чрез списъци на съседство. Както в много други алгоритми върху графи, които ще разгледаме, сложността по време **НЕ СЕ ОПРЕДЕЛЯ** така (допускаме, че $G' = G$):

while-цикълът се изпълнява n пъти, по веднъж за всеки връх, а всяко изпълнение отнема време, в най-лошия случай, $\Theta(m)$, така че сложността е $\Theta(nm)$.

Това разсъждение в общия случай е **ГРЕШНО**. От това разсъждение следва асимптотична горна граница $O(nm)$, но, в общия случай, тя не е точна. Да, има безкрайни класове графи, за които това разсъждение е приложимо. Примерно, клас графи, в който всеки граф има един и същи брой ребра; тоест, $m = \Theta(1)$. Тогава наистина сметката е такава. Но в общия случай това не е така.

Прецизното разсъждение, което ни дава точна асимптотична оценка винаги, е да се игнорира **while**-цикълът и да се фокусираме върху вътрешния цикъл, тоест, върху **for**-цикъла. Но не върху всички изпълнения на **for**-цикъла в рамките на едно изпълнение на **while**-цикъла, а върху всички изпълнения на **for**-цикъла по време на цялата работа на BFS. Тогава виждаме, че променливата u минава систематично през всички елементи на всички списъци на съседство. Размерът на списъците на съседство е $\Theta(n + m)$, а алгоритъмът върши само $\Theta(1)$ работа за всеки списъчен елемент. Оттук веднага получаваме точна асимптотична оценка $\Theta(n + m)$ за сложността по време на BFS.

Ако G не е мултиграф, може да разсъждаваме така:

while-цикълът се изпълнява n пъти, по веднъж за всеки връх, а всяко изпълнение отнема време, в най-лошия случай, $\Theta(n)$, така че сложността е $\Theta(n^2)$.

Това разсъждение вече не е грешно. Наистина, най-лошият случай, в който $m = \Theta(n)$, е точно такъв. Но $\Theta(n + m)$ остава по-прецизна оценка, тъй като функцията в израза е на две променливи и това е по-информативно за общия случай.

И така, сложността по време на BFS е $\Theta(n + m)$.

9.4.2 Псевдокод на BFS, обхождащ целия граф

Следва псевдокод на BFS, който обхожда целия граф. Отново графът на входа е ориентиран мултиграф с възможни примки. Този път стартов връх няма.

Ако мултиграфът е неориентиран, променливата $count$ се инкрементира точно веднъж за всяка свързана компонента, така че алгоритъмът връща броя на свързаните компоненти. Тривиално е да се добави код, така че алгоритъмът да връща самите свързани компоненти, а не само броя им.

```
BFS-MOD( $G = (V, E)$ , като  $V = \{1, \dots, n\}$ )
1  for  $i \leftarrow 1$  to  $n$ 
2    color[i]  $\leftarrow$  white
3     $d[i] \leftarrow \infty$ 
4     $\pi[i] \leftarrow \text{Nil}$ 
5  count  $\leftarrow 0$ 
6  for  $i \leftarrow 1$  to  $n$ 
7    if color[i] = white
8      count ++
9      BFS-VISIT( $G, i$ )
10 return count
```

```
BFS-VISIT( $G = (V, E); s \in V$ )
1 color[s]  $\leftarrow$  gray
2  $d[s] \leftarrow 0$ 
3 създай опашка Q
4 enqueue(Q, s)
5 while not isempty(Q) do
6    $x \leftarrow \text{dequeue}(Q)$ 
7   for  $y \in \text{adj}[x]$ 
8     if color[y] = white
9       color[y]  $\leftarrow$  gray
10       $d[y] \leftarrow d[x] + 1$ 
11       $\pi[y] \leftarrow x$ 
12      enqueue(Q, y)
13  color[x]  $\leftarrow$  black
```

Аргументация за коректност няма да правим. Очевидно сложността е $\Theta(n + m)$.

9.4.3 Приложения на BFS

Изчисляване на двуделност на неориентиран граф Нека $G = (V, E)$ е обикновен граф. БОО, нека G е свързан. Пита се дали G е двуделен; алтернативно, дали е 2-оцветим,

което е същото или почти същото нещо, в зависимост от формалните дефиниции[†]. Знаем необходимо и достатъчно условие за това: да няма нечетни цикли. Това условие обаче не се превежда директно в ефикасен алгоритъм. Циклите на графа може да са суперекспоненциално много в n , така че, дори да имаме алгоритъм, намиращ всеки цикъл във време $\Theta(1)$ (в обикновения или амортизирания смисъл), той би бил неприемливо бавен за определяне на дувделността.

Оптимално ефикасен, в асимптотичния смисъл, е алгоритъм, базиран на BFS. Ако изобщо G е 2-оцветим[‡], да кажем в зелено и червено, то има точно две възможности:

- s е червен, неговите съседи са зелени, техните съседи без s са червени, и така нататък,
- и обратно, s е зелен, неговите съседи са червени, техните съседи без s са зелени, и така нататък.

Виждаме, че разбиването на V е едно и също и при двете схеми за оцветяване – те само разменят цветовете на двета дяла. Същественото е, че двета дяла може да дефинират по отношение на разстоянието (сега говорим за обикновено симетрично разстояние, понеже G е неориентиран) между s и всеки друг връх – върховете на четно разстояние са единият дял, примерно червените, а тези на нечетно разстояние са другия дял, в случая зелените.

Това веднага дава идея за опит за оцветяване на произволен свързан граф. Започваме BFS обхождане от произволен връх s и за всяко ребро (u, v) следим дали двета края са на различно разстояние от s , или не. Забележете, че е невъзможно разстоянието между u и s да се различават на повече от единица от разстоянието между v и s . Еrgo, тези разстояния или се различават на единица, което означава, че са с противоложна четност, което означава, че u и v са в различни цветове, или са равни, при което u и v са в един и същи цвят. Ако при обхождането се окаже, че съществува ребро (u, v) , такова че $d[u] = d[v]$, алгоритъмът връща НЕ. В противен случай връща Да.

Тривиално лесно е да се модифицира кода на BFS съгласно тази идея, така че сложността по време да остане $\Theta(n + m)$.

9.5 DFS

9.5.1 Неформално въведение и сравнение с BFS

DFS реализира, в никакъв смисъл, обратната идея на BFS обхождането. BFS обхождането е “предпазливо”. Ако си представим обхождане на лабиринт, BFS идеята е, грубо казано, такава: има стаи, в които сме били, но с които не сме приключили; ние първо преглеждаме всички коридори, излизящи от тях, преди да продължим с новите стаи, като новите стаи са тези, които сме открили чрез въпросните коридори.

DFS обхождането е “смело”. От началната стая излизаме през първия възможен коридор, от следващата също, и така нататък, като надлежно маркираме стаите, в които вече сме били, за да не зациклим. Рано или късно ще се окажем в стая, в която сме били. Тогава се връщаме колкото е възможно по-малко назад до стая, в която сме били, и опитваме същото нещо, само че излизайки от нея през врата, която не сме използвали. На английски тази идея се нарича *backtracking*[§].

[†]Някои формални дефиниции на “дувделен граф” предполагат, че графът има поне два върха.

[‡]Тези цветове нямат **нищо общо** с цветовете бял, сив и черен, които BFS използва по време на работата си!

[§]Backtracking е алгоритмична схема, също както разделяй-и-владей е алгоритмична схема. В този курс няма да разглеждаме алгоритми, използващи backtracking, освен DFS.

Допълнение 28: Разликата между BFS и DFS, илюстрирана с шах

Ще илюстрираме разликата между двете идеи—BFS и DFS—с шаха. Нека е дадена задача да открием мат в три хода в дадена позиция за белите^a. Да подходим алгоритмично.

BFS идеята. BFS подходът е да генерираме всички възможни, съгласно правила на шаха, позиции с един полуход на белите. Нека множеството от тези позиции е P_1 . За всяка от тези позиции генерираме всички позиции с един полуход на черните, получавайки множество Q_1 : това са всички позиции на един ход. Използвайки Q_1 и всички възможни полуходове на белите, генерираме множеството P_2 , а от него с всички възможни полуходове на черните генерираме Q_2 : множеството от всички позиции на два хода от началната. От Q_2 с всички възможни полуходове на белите генерираме P_3 . Ако задачата е смислена, в една или повече от позициите на P_3 има мат на черния цар, но в P_1 и P_2 няма мат на черния цар. Нещо повече. Трябва:

- да **съществува** полуход на белите в началната позиция, такъв че
- **за всеки** полуход на черните след него
- да **съществува** полуход на белите, такъв че
- **за всеки** полуход на черните след него
- да **съществува** полуход на белите, такъв че черният цар е мат.

В тази редица от полуходове се редуват квантори \exists и \forall , като \exists са на белите полуходове, а \forall , на черните. Тъй като възможните полуходове от дадена позиция може да са десетки, дори при малко фигури, говорим за дърво с разклоненост от порядъка на десетки. Височината му в полуходове е шест. Това е дървото на BFS-а, който току-що си представихме. При BFS идеята генерираме нивата на дървото систематично от корена към листата.

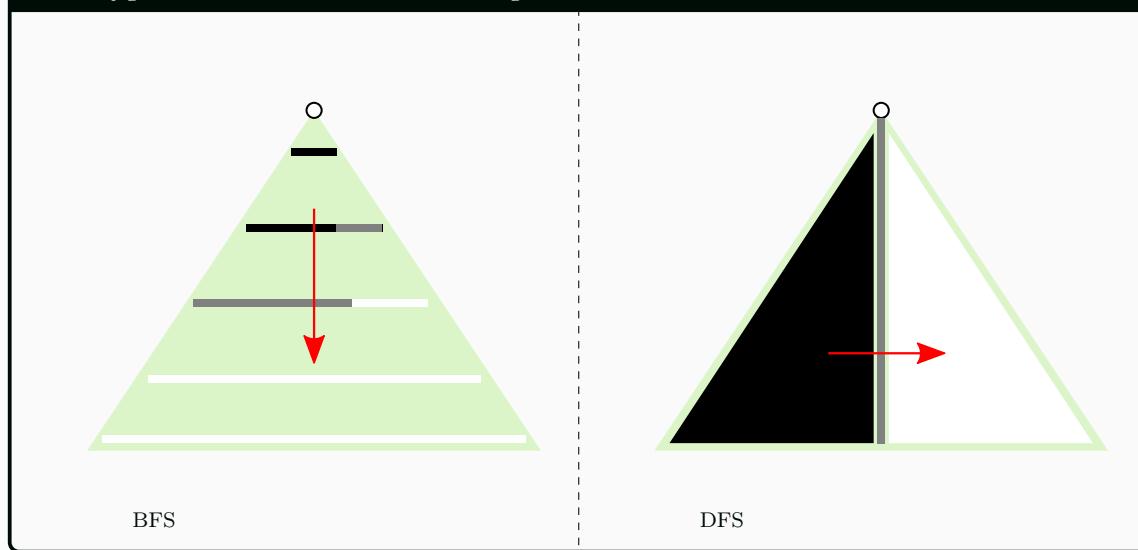
DFS идеята. DFS подходът е, да опитаме един възможен полуход на белите от началната позиция, после един полуход на черните, и така нататък, докато направим трети полуход за белите. Отчитаме дали черният цар е мат или не, и се връщаме в предната позиция, която е получена при втори полуход за черните. От нея правим следващия възможен трети полуход за белите—ако в предишния трети полуход за белите не е имало мат на черния цар—и така нататък.

След като “влезе” в позиция p , DFS генерира **една след друга** позициите, достигими от нея, като за всяка от тях, да я наречем q , “влиза” в q и се връща обратно в p чак след като е изчерпила всички възможности “до долу”, за да опита следващата след q , ако има такава. Ако p не е началната позиция, трябва да се “излезе” от нея, което става чак когато са изчерпани всички възможности за следваща позиция.

Както и при BFS, така и при DFS решението на задачата е полуход на белите, такъв че за всеки полуход на черните има полуход на белите, такъв че за всеки полуход на черните има полуход на белите, в който черният цар е мат. Съществената разлика с BFS е, че BFS генерира цялото P_1 , от него цялото Q_1 , и така нататък, докато DFS не генерира пълните нива. DFS генерира елементите на нивата, но последователно, а не всички наведнъж.

Ако си представим търсенето на мат като дърво с корен началната позиция, BFS генерира цялото дърво⁶, вървейки вертикално, отгоре надолу, ако коренът е горе. Докато DFS поддържа само един вертикален път от корена до листата, като този път се движи хоризонтално, образно казано. Във всеки момент DFS съхранява в явен вид само позициите от този път, а следващите позиции се пресмятат от текущите. Фигура 9.1 илюстрира разликата в двата подхода. На нея цветовете бял, сив и черен имат смисъла на цветовете на върховете от BFS и DFS и нямат нищо общо с цветовете на шахматните фигури.

Фигура 9.1 : Обхождане на дърво от BFS и от DFS.



^aЕто сайт с такива [шахматни задачи](#).

^bПочти цялото; може би без върхове от най-долното ниво.

Ако заменим в кода на BFS опашката със стек (съответно и функциите от интерфейса с push и pop), ще получим **нещо като** DFS, но не точно DFS. Нека читателя пробва да замени опашката със стек в BFS и да се убеди върху малък пример, че полученото обхождане е, в някакъв смисъл, междинен вариант, нито BFS, нито истински DFS обхождане.

9.5.2 Псевдокод на DFS

Следният псевдокод е буквално същият като псевдокода в [15, стр. 604]. Смисълът на цветовете е същият като при BFS. Разстоянията сега няма – DFS не пресмята разстояния в графа. В някои приложения на DFS въвеждаме нивá, които се пресмятат като d-стойностите в BFS, но при DFS нивáта нямат смисъл на разстояния.

Използваме глобална променлива `time`, чрез която пресмятаме времето на откриване на всеки връх и времето на финализиране на всеки връх. Два масива $d[1, \dots, n]$ и $f[1, \dots, n]$ съхраняват тази информация за върховете.

Разглеждаме вариант на DFS, който задължително обхожда целия граф, подобно на модификацията на BFS на стр. 274. Тривиално лесно е да направим от него DFS, който получава стартов връх като аргумент и обхожда само подграфа, който е достигим от този стартов връх.

$\text{DFS}(G = (V, E)$, като $V = \{1, \dots, n\}$)

```

1 time е глобална
2 for i ← 1 to n
3   color[i] ← white
4   π[i] ← Nil
5 time ← 0
6 for i ← 1 to n
7   if color[i] = white
8     DFS-VISIT(G, i)

```

$\text{DFS-VISIT}(G = (V, E); x \in V)$

```

1 color[x] ← gray
2 time++
3 d[x] ← time
4 foreach y ∈ adj[x]
5   if color[y] = white
6     π[y] ← x
7     DFS-VISIT(G, y)
8 color[x] ← black
9 time++
10 f[x] ← time

```

Много подробен пример за работата на DFS има в [15], стр. 605]. Забележете, че наредените двойки $(d[v], f[v])$, по всички $v \in V$, напълно определят работата на DFS върху даден граф. Конвенцията на [15] е върховете да се рисуват като елипси, а в елипсата, съответстваща на връх v , наредената двойка $(d[v], f[v])$ да се записва като " $d[v]/f[v]$ ". Например, в рисунката, показваща края на алгоритъма, връх u е маркиран с "1/8", което означава, че връх u е бил открит в момент 1 и е бил финализиран в момент 8.

Формално доказателство за коректността на DFS няма да правим тук. Аргументацията на стр. 267 за коректността на общата схема е достатъчна. Сложността по време на DFS е $\Theta(n + m)$. Това се извежда със същите разсъждения, с които се извежда линейната сложност на BFS.

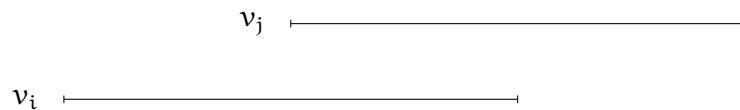
9.5.3 Свойства на DFS

Скобова структура на интервалите, генеририани от DFS. За целите на това изложение, нека върховете са v_1, \dots, v_n . Очевидно е, че променливата time , която се инициализира с 0 на ред 5, приема последователно стойностите 1, 2, ..., $2n$, тъй като бива инкрементирана два пъти за всеки връх x в DFS-VISIT. Да си представим числовата ос с тези $2n$ стойности на t . Ако групираме стойностите по двойки за отделните върхове, получаваме n интервала $[d[v_1], f[v_1]], [d[v_2], f[v_2]], \dots, [d[v_n], f[v_n]]$. Казваме, че това са *интервалите, генеририани от $\text{DFS}(G)$* . Краишата на интервалите са, две по две, различни точки, така че всеки два интервала или имат празно сечение, или имат общ подинтервал с дължина поне единица. Фигура 22.5 в [15], стр. 607] илюстрира добре интервалите на DFS.

Теорема 24: Теорема

В текущия контекст, ако два различни интервала имат непразно сечение, то единият се съдържа строго в другия: .

Доказателство: Твърди се, че е невъзможно да има такива интервали:  . Да допуснем противното. Нека има върхове v_i и v_j , такива че $d[v_i] < d[v_j] < f[v_i] < f[v_j]$:



DSF първо открива v_i , а после v_j , като v_i не е финализиран в момента на откриването на v_j . Това означава, рекурсивното викане R_i на DFS-VISIT, в което $x = v_i$, не е приключило в момента, в който в някое следващо рекурсивно викане R_j е вярно, че $x = v_j$.

Същото нещо, по-подробно казано, звучи така.

- В рекурсивното викане R_i е вярно, че $x = v_i$ и се изпълнява цикълът на редове 4–7. $d[v_i]$ получава своята стойност на ред 3. За някое дете u' на v_i е вярно, че когато y (в R_i) стане u' се прави рекурсивно викане R' на DFS-VISIT.
- В рекурсивното викане R' е вярно, че $x = u'$ и се изпълнява цикълът на редове 4–7. За някое дете u'' на u' е вярно, че когато y (в R') стане u'' се прави рекурсивно викане R'' на DFS-VISIT.
- И така нататък ...
- В рекурсивното викане R_j е вярно, че $x = v_j$.

Ключовото наблюдение е, че в момента, в който $d[v_j]$ получава своята стойност на ред 3—това става в рекурсивното викане R_j —викането R_i още не е приключило и в R_i се изпълнява цикъла на редове 4–7, така че ред 10 в R_i не е достигнат и $f[v_i]$ все още не е получила стойност. Със сигурност $f[v_i]$ ще получи стойност, но това ще е за стойност на time, по-голяма от стойността на time в момента, в който $d[v_j]$ получава своята стойност. \checkmark □

Клодифициране на ребрата на ориентиран граф от DFS. Нека $G = (V, E)$ е ориентиран мултиграф с възможни примки. Нека гората на обхождането, реализирана чрез масива π , е A . Да си спомним, че A е множество от дървета. В края на $DFS(G)$ ребрата от E се разбиват[†] на следните четири категории:

дървесни ребра. Това са точно ребрата на A . С други думи, (u, v) е ребро назад, ако u и v са върхове от едно и също дърво $T \in A$, като u е родител на v в T .

ребра назад. (u, v) е ребро назад, ако u и v са върхове от едно и също дърво $T \in A$, като v е предшественик на u в T . Тоест, в T има път от v до u . Примките в G задължително са в тази категория.

ребра напред. (u, v) е ребро напред, ако u и v са върхове от едно и също дърво $T \in A$, като u е предшественик на v в T , но (u, v) не е дървесно ребро. Тоест, (u, v) не е ребро в T .

ребра настани. (u, v) е ребро настани, ако u и v не са в отношение на предшествие в A . Това означава, че

- или u и v са върхове от едно и също дърво от A , но нито единият не е предшественик на другия,
- или са върхове от две различни дървета от A .

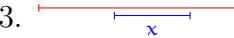
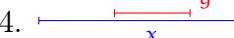
В [15, стр. 609] съответните термини са *tree edges*, *back edges*, *forward edges* и *cross edges*. Примери за граф и DFS върху него, след който се срещат и четирите вида ребра, има на Фигура 22.4 в [15, стр. 605] и на Фигура 22.5 в [15, стр. 607], като на Фигура 22.5.c е показан и графът, прерисуван като дърветата на обхождането плюс останалите ребра.

[†]Някои дялове-категории на това разбиране може да са празни, така че, формално, правилно е да се каже “се разбиват на най-много четири категории”.

Как да разберем по време на работата на DFS дадено ребро от кой вид е? Ребрата на G биват обхождани на ред 4 от DFS-VISIT. Нека реброто е (x, y) . Първо да разгледаме възможностите за цвета на y .

- Ако y е бял, то реброто (x, y) е дървесно, понеже условието на ред 5 е TRUE и изпълнението отива на ред 6, където x става родителят на y в дървото; това означава, че (x, y) става ребро от дървото.
- Ако y е сив, то y е предшественик на x в дървото. Приемаме това за очевидно.
- Ако y е черен, то или y е наследник на x и с него вече сме приключили, или y не е в отношение на предшествие с x и с него сме приключили. Тоест, (x, y) е ребро напред или ребро настрани. Ерго, само от черния цвят на y не можем да класифицираме точно (x, y) .

Сега да разгледаме възможностите за взаимните разположения на $d[x]$, $d[y]$, $f[x]$ и $f[y]$. Предвид това, че тези са две по две различни, и че $d[x] < f[x]$ и $d[y] < f[y]$, има $\binom{4}{2} = 6$ възможности от общи комбинаторни съображения. С по-подробен анализ, възможностите стават само три.

1.  y Това е невъзможно. Щом (x, y) е ребро, няма как x да бъде финализиран на ред 10 преди бялото му дете y да бъде открито на ред 5.
2.  y Това е невъзможно съгласно Теорема 24.
3.  y е предшественик на x . В такъв случай y е сив и (x, y) е ребро назад. Всички примки са такива.
4.  x е родителят на y . В такъв случай y е бял и (x, y) е дървесно ребро или y е черен и (x, y) е ребро напред.
5.  x Това е невъзможно съгласно Теорема 24.
6.  x Това е възможно. Щом y е финализиран преди откриването на x , нито x е предшественик на y , нито y е предшественик на x . В този случай y е черен и (x, y) е ребро настрани.

Сега е ясно как да различаваме дали (x, y) , при черен връх y , е ребро напред или ребро настрани. Ако $d[x] < f[y]$, то (x, y) е ребро напред, ако $f[y] < d[x]$, то (x, y) е ребро настрани.

Следната лема обобщава казаното дотук. Доказателството ѝ са току-що извършените разсъждения.

Лема 42: Възможностите за $d[x]$, $d[y]$, $f[x]$ и $f[y]$ на $(x, y) \in E$ след $DFS(G)$

Това са всички възможности след $DFS(G)$ за взаимното разположение на $d[x]$, $d[y]$, $f[x]$ и $f[y]$, където $(x, y) \in E$:

1. $d[x] < d[y] < f[y] < f[x]$ т.к. (x, y) е дървесно ребро или ребро напред,
2. $d[y] < d[x] < f[x] < f[y]$ т.к. (x, y) е ребро назад,
3. и $d[y] < f[y] < d[x] < f[x]$ т.к. (x, y) е ребро настрани.

Клодифициране на ребрата на неориентиран граф от DFS. Нека $G = (V, E)$ е неориентиран мултиграф с възможни примки. Нека гората на обхождането, реализирана чрез масива π , е A . В края на $DFS(G)$ ребрата от E се разбиват на следните две категории:

дървесни ребра. Това са точно ребрата на A .

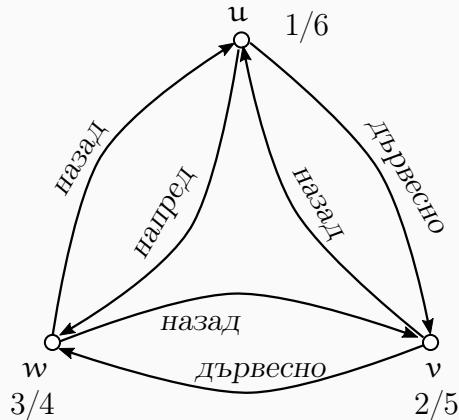
ребра назад. (u, v) е ребро назад, ако u и v са върхове от едно и също дърво $T \in A$, като v е предшественик на u в T . Примките в G задължително са в тази категория.

С други думи, в неориентираните графи няма ребра напред или ребра настрани. Общото за ребрата напред или настрани е, че чрез тях откриваме черни върхове. Дали е невъзможно DFS върху неориентирани графи да открива черни върхове? Естествено, че е възможно. Но винаги, когато DFS върху неориентиран граф открие черен връх—тоест, връх u на ред 5 е черен—е вярно, че неориентираното ребро (x, u) **вече** е било обходено от DFS и **вече** е било класифицирано като дървесно ребро или ребро напред. Същността на тази аргументация е в това, че

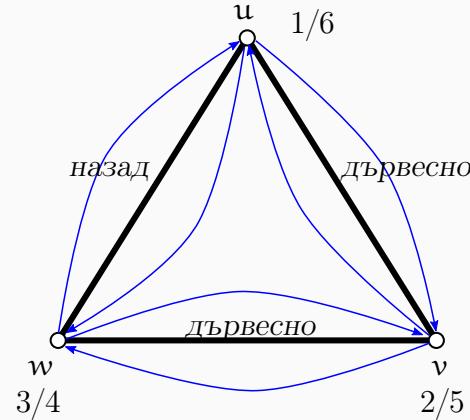
1. ако гледаме на даден граф със сдвоени (вж. дискусията на стр. 264) ориентирани ребра като на ориентиран граф, DFS може да открие ребра напред или ребра настрани, като всяко такова ребро е от двойка, другото ребро от която е дървесно ребро или ребро назад;
2. но ако гледаме на същия граф като на неориентиран граф, който само е представен като ориентиран—всяко неориентирано ребро (u, v) е представено чрез двойка ориентирани ребра (u, v) и (v, u) —то няма как да класифицираме едно и също неориентирано ребро по два начина, съответно за двете ориентирани ребра, които го представлят. И класифицирането на реброто никога не става като ребро напред или настрани.

Разгледайте Фигура 9.2. Тя сравнява резултатите от работата на DFS върху ориентиран и неориентиран граф. Двата графа имат едни и същи списъци на съседства, но графът вдясно се смята за неориентиран и всяка двойка сдвоени ориентирани ребра в представянето (в синьо) отговаря на едно неориентирано ребро на графа (в черно), докато при графа вляво всяко ориентирано ребро в представянето отговаря на точно едно ориентирано ребро на графа. И при ориентирания, и при неориентирания граф има точно две дървесни ребра след приключването на DFS. Забележете, че реброто (u, v) в ориентирания граф е дървесно, докато реброто (v, u) е ребро назад, тоест, ребрата от двойката биват класифицирани по различен начин, докато в неориентирания граф на тях отговаря само едно ребро и то бива класифицирано като дървесно, защото първият път, в който прекосяваме едното от двете отговарящи му ориентирани ребра от представянето (в синьо), откриваме бял връх (v) .

Фигура 9.2 : Ориентиран граф със сдвоени ребра и неориентиран граф.



Ориентиран граф със сдвоени ребра.

Неориентиран граф, представен чрез
ориентиран граф със сдвоени ребра.

Фигура 9.2 е добра илюстрация на факта, че в неориентиран граф не може да има ребра напред. Сравнете ориентираното ребро напред (u, w) в ориентириания граф с неориентираното ребро (u, w) в неориентирания граф – очевидно е, че при сдвоени ориентирани ребра, ребро може да бъде класифицирано като ребро напред само ако другото ребро от двойката вече е било прекосено и класифицирано като ребро назад. Ако графът е неориентиран и това са само ориентирани ребра от преставянето, очевидно е, че реброто като един обект ще бъде класифицирано съгласно първото прекояване, а именно като ребро назад. С аналогични разсъждения може да изведем, че в неориентирани графи няма ребра настрани.

Следната теорема дава формално доказателство, че при неориентирани графи ребрата са само дървесни и ребра назад.

Теорема 25: Theorem 22.10 в [15, стр. 610]

При всяка работа на DFS върху неориентиран граф G , всяко обходено ребро е дървесно или ребро назад.

Доказателство: Разглеждаме произволно ребро $(u, v) \in E(G)$. БОО, нека $d[u] < d[v]$. Тогава DFS го открива, и финализира v преди да финализира u ; това е очевидно.

Да разгледаме двете възможности за обхождане на реброто (u, v) :

1. Ако то бъде обходено в посока от u към v , то чрез него ще открием бял връх (v) и ще го класифицираме като дървесно.
2. Ако то бъде обходено в посока от v към u , то чрез него ще открием сив връх (u) и ще го класифицираме като ребро назад. \square

9.5.4 Приложения на DFS

Изследване на цикличност на графи чрез DFS. И за ориентирани, и за неориентирани графи може да отговорим ефикасно на въпроса дали графът е цикличен, използвайки DFS.

Теорема 26

Нека G е ориентиран или неориентиран граф. G е цикличен тогава и само тогава, когато произволно изпълнение на $\text{DFS}(G)$ открива поне едно ребро назад.

Доказателство: В едната посока, да допуснем, че $e = (u, v)$ е ребро назад. Ако G е неориентиран, нека обхождането става в посока от u към v . Както вече видяхме, това означава, че в момента на обхождането, u и v са сиви, като v е предшественик на u в дървото на обхождането. Тогава в дървото има път от v до u (или между v и u , в неориентириания случай), който път заедно с реброто e дава цикъл. Този аргумент е валиден дори когато e е примка; тогава въпросният път се състои от един единствен връх, а именно $u = v$.

В другата посока, нека има цикъл c . Очевидно DFS открива върховете на цикъла последователно, но не в непременно непрекъсната последователност. Нека $v \in V(c)$ е този връх на цикъла, който бива открит последен от DFS. Но v има поне един съседен връх u в c .

1. v има поне два съседни върха u, w , които са от цикъла (ако става дума за мултиграф, може $u = w$; ако c е примка, може дори $v = u = w$; всичко това е без значение за доказателството), ако G е неориентиран;
2. v има поне един съседен връх u , който е от цикъла (възможно е $u = v$, ако реброто е примка; това е без значение за доказателството), ако G е ориентиран.

Когато DFS-VISIT минава през списъка на съседство на v , открива u като сив връх и реброто (v, u) бива класифицирано като ребро назад. \square

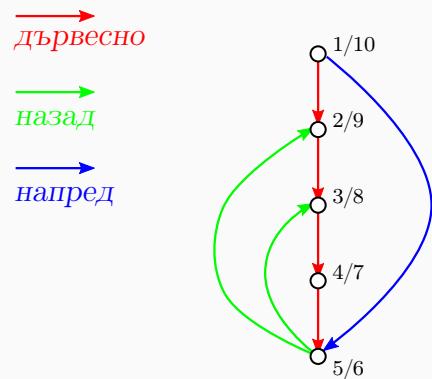
Наблюдение 29

При пускане на DFS върху G от един връх s , броят на дървесните ребра, които DFS открива, зависи единствено от G , а не от представянето му чрез списъци. Нека G' е подграфът, достъжим от s . Очевидно G' не зависи от представянето. Тогава броят на дървесните ребра, които DFS открива, е $|V(G')| - 1$.

Броят на ребрата от всички останали видове е $|E(G')| - (|V(G')| - 1) = |E(G')| - |V(G')| + 1$ и очевидно също не зависи от представянето на G . Ако G е неориентиран, останалите ребра са само ребра назад. Но ако G е ориентиран, останалите ребра са от (най-много) три вида и това, по колко има от всеки вид, може да зависи от представянето на G .

Илюстрация: Фигура 9.3 показва две различни изпълнения на DFS върху един и същи граф, от един и същи начален връх, но с различни представяния на графа (различни списъци на съседство). Броят на дървесните ребра е един и същи—винаги четири, защото върховете са пет—но останалите ребра се категоризират като ребра назад и ребра напред по различни начини и с различен брой ребра в тези категории. \square

Фигура 9.3 : Равничен брой ребра назад.

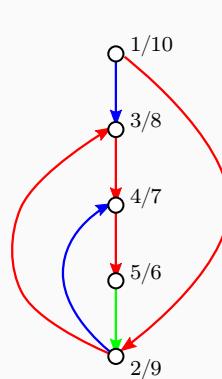


дървесно

назад

напред

Две ребра назад.



дървесно

назад

напред

Едно ребро назад.

Лекция 10

Топологическо сортиране. Намиране на силно свързани компоненти. Намиране на срязващи върхове и мостове.

Резюме: Разглеждаме два линейни алгоритъма за топологическо сортиране на дагове: алгоритъмът на Kahn и алгоритъмът на Tarjan. Разглеждаме линеен алгоритъм за намиране на силно свързани компоненти на ориентиран граф. Разглеждаме линейни алгоритми за намиране на срязващите върхове и мостовете на неориентиран граф.

10.1 Фундамент

Навсякъде в тази лекция, нека $G = (V, E)$ е ориентиран граф.

Определение 41: Топологическо сортиране

Топологическо сортиране на G е всяка биекция $h : V \rightarrow \{1, \dots, n\}$, такава че за всяко ребро $(u, v) \in E$ е вярно, че $h(u) < h(v)$.

Теорема 27: НДУ за съществуване на топологическо сортиране

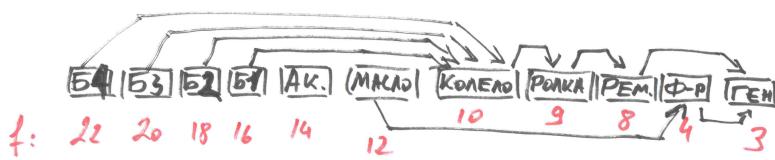
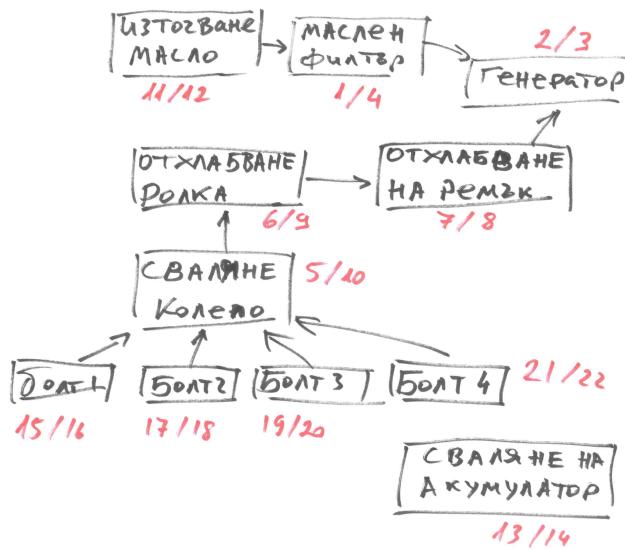
Съществува топологическо сортиране на G тогава и само тогава, когато G е даг.

Източник в G , на английски *source*, е всеки връх в G , чиято полустепен на входа е нула.
Сифон в G , на английски *sink*, е всеки връх в G , чиято полустепен на изхода е нула.

Теорема 28: Поне един източник и поне един сифон в даг

Във всеки даг има поне един източник и поне един сифон.

Пример за даг и негово топологическо сортиране е показан на следващата страница. Примерът идва от реална житейска ситуация – смяна на електрическия генератор на автомобил, като свалянето на генератора налага предварително да се извършат други дейности, които на свой ред налагат предварително да се извършат други дейности, и така нататък. Върховете на дага са дейностите, а ребрата съответстват на непосредствените предшествия между дейностите.



Допълнение 29: Произход на “топологическо сортиране”.

Името “топологическо сортиране”, без съмнение, идва от “топология” – дял на математиката, възникнал като абстракция на геометрията. По думите на Henning Makholm в тази [дискусия за произхода на името](#):

Graph theory was originally (and still sometimes is, depending on who you ask) considered a branch of topology.

This may sound strange to people with a modern education, where “topology” means more or less “the part of mathematics that deals abstractly with continuity and limits, without using real numbers” – or at least without giving the real numbers any central position in the theory. However, earlier on, “topology” appears to have been a catch-all term for “the part of mathematics that isn’t about numbers or geometric magnitudes”. (This was before algebraists stopped pretending that algebra is necessarily about numbers). Only later did a distinction between what we now call topology and discrete mathematics become common.

In this old usage, “topological sorting” simply means “the kind of sorting you can define without reference to comparison of numbers”.

Терминът “topological ordering” се появява в статията на Kahn от 1962 г. [31], където се казва:

In recent years much work has been done on the computational analysis of problems which can be formulated as networks with distinct elements. In particular, this class of problems include PERT (Program Evaluation Review Technique) which is used as management schedule tool. To a large extent the feasibility of network computations depends upon the ability to arrange the network information in topological order. A list in topological order has a special property. Simply expressed: proceeding from element to element along any path in the network, one passes through the list in one direction only. Stated another way, a list in topological order is such that no element appears in it until after all elements appearing on all paths leading to the particular element have been listed.

“PERT” е средство от 50-те години на XX век за управление на проекти, като описаното на задачите в даден проект задава нещо като даг от зависимости (задача X преди задача Y). За подробности вижте [статията в Уикипедия](#).

Допълнение 30: Команда tsort.

Типичните UNIX инсталации имат средство за топологическо сортиране: командата **tsort**. Подробна информация за нея може да се получи с **info tsort**. Ето пример за използването ѝ. Това е примерът с ремонта на автомобил на предишната страница. Елементите, които трябва да бъдат топологически сортирани, се изписват в двойки, които означават ограниченията кое преди кое да бъде. Тези двойки са ориентирани

ребра на графа. Разполагането на двойките по редове е без значение: програмата чете списък и го разбива на двойки. Известен недостатък на представянето на графа само чрез ребрата е, че няма как да се описват изолирани върхове; затова дейността сваляне на акумулатор не присъства.

```
$ tsort <<EOF
> източване-масло маслен-filtър
> маслен-filtър генератор
> отхлабване-ролка отхлабване-на-ремък
> отхлабване-на-ремък генератор
> сваляне-колело отхлабване-ролка
> болт-1 сваляне-колело
> болт-2 сваляне-колело
> болт-3 сваляне-колело
> болт-4 сваляне-колело
> EOF
болт-1
болт-2
болт-3
болт-4
източване-масло
сваляне-колело
маслен-filtър
отхлабване-ролка
отхлабване-на-ремък
генератор
$
```

10.2 Алгоритъм на Tarjan за топологическо сортиране

Ефикасен алгоритъм за топологическо сортиране на даг е показаната в тази секция модификация на DFS, предложена от Tarjan [66]. Същината на този алгоритъм е да се сортират върховете по **намаляващи f-стойности**, където $f[1, \dots, n]$ е масивът от времената на финализиране, генериирани от $\text{DFS}(G)$. Примерът на стр. 286 показва точно това. Времената на откриване и финализиране са показани в червено до върховете на дага, а след това е показана и редицата от върховете, обратно сортирани п времената на финализиране.

Сортирането на върховете по намаляващи f-стойности може да стане имплицитно в процеса на работата на DFS. Топологическата сортировка се реализира чрез масив $T[1, \dots, n]$, който е глобално видим. Масивът се пълни с върхове отдясно наляво, като върхът с най-малка f-стойност е в най-дясната позиция и така нататък. Ето и псеводокод.

TSORT(Даг $G = (V, E)$, като $V = \{1, \dots, n\}$)

```

1    $T[1, \dots, n]$ , time и  $\ell$  са глобални
2   for  $i \leftarrow 1$  to  $n$ 
3      $color[i] \leftarrow$  white
4      $\pi[i] \leftarrow$  NIL
5    $time \leftarrow 0$ ,  $\ell \leftarrow n$ 
6   for  $i \leftarrow 1$  to  $n$ 
7     if  $color[i] \leftarrow$  white
8       TSORT-REC( $G, i$ )
9   return  $T[1, \dots, n]$ 
```

TSORT-REC($G = (V, E); x \in V$)

```

1    $color[x] \leftarrow$  gray
2   time++
3    $d[x] \leftarrow$  time
4   foreach  $y \in adj[x]$ 
5     if  $color[y] =$  white
6        $\pi[y] \leftarrow x$ 
7       TSORT-REC( $G, y$ )
8    $color[x] \leftarrow$  black
9   time++
10   $f[x] \leftarrow$  time
11   $T[\ell] \leftarrow x$ 
12   $\ell--$ 
```

Интуитивна аргументация на коректността на алгоритъма е, че най-малката f -стойност задължително е на сифон—ако графът е даг, той има поне един сифон съгласно Теорема 28—така че върхът в $T[n]$ е сифон. Махайки сифона от дага, получаваме пак даг, така че следващата по големина f -стойност е на сифон в новия даг, така че $T[n - 1]$ е сифон в новия даг, и така нататък. При това не може да се появят ребра отляво наляво, по отношение на масива T , защото от сифоните не излизат ребра.

Ето и формално доказателство на коректността. В него не се използват сифони; за сифони говорихме в неформалната обосновка.

Теорема 29: Коректност на топологическото сортиране на Tarjan

Ако G е даг, пермутацията $\pi = \langle u_1 u_2 \dots u_n \rangle$ на върховете на G , такава че $f[u_1] > f[u_2] > \dots > f[u_n]$ след терминирането на $DFS(G)$, реализира топологическо сортиране.

Доказателство: Нека (u, v) е произволно ребро на графа. Съгласно Лема 42, следните възможности са изчерпателни:

- $d[u] < d[v] < f[v] < f[u]$, ако (u, v) е дървесно ребро или ребро напред. Важното в случая е, че $f[u] > f[v]$. Тое влече, че връх u е вляво от връх v в пермутацията: $\pi = \dots u \dots v \dots$, така че реброто (u, v) не противоречи на топологическото сортиране.
- $d[v] < d[u] < f[u] < f[v]$, което обаче е невъзможно в текущия контекст: това би означавало, че (u, v) е ребро назад, а съгласно Теорема 26 това би означавало, че G е цикличен, в противоречие с допускането, че G е даг.
- $d[v] < f[v] < d[u] < f[u]$, ако (u, v) е ребро настрани. Отново, имаме $f[u] > f[v]$, така че u е вляво от v в пермутацията и реброто (u, v) не противоречи на топологическото сортиране. \square

Важно наблюдение, което ще използваме в Секция 10.4 е, че този алгоритъм не дава съобщение за грешка, ако входният граф не е даг, а приключва работата си и генерира масива $T[1, \dots, n]$ от върхове. Разбира се, ако G е цикличен, T не реализира топологическо сортиране.

Сложността по време на топологическото сортиране на Tarjan е $\Theta(n + m)$. Това следва от сложността по време на DFS и факта, че топологическото сортиране на Tarjan добавя само $\Theta(1)$ работа към рекурсивните викания.

10.3 Алгоритъм на Kahn за топологическо сортиране

Този алгоритъм е описан още през 1962 г. [31]. Той работи, в никакъв смисъл, обратно на алгоритъма на Tarjan. Докато алгоритъмът на Tarjan върви отдясно наляво, тоест от сифоните назад, алгоритъмът на Kahn върви отляво надясно, тоест от източниците напред. Наричат алгоритъма на Kahn “топологическо сортиране чрез BFS”. За да се подчертава приликата с BFS тук ползваме цветовете бял, сив и черен за маркиране на различните от гледна точка на алгоритъма състояния на върховете. Не особено съществена разлика с BFS е, че сега множеството S се инициализира с всички източници в G , докато при BFS, неговото съответствие, опашката Q , се инициализира с точно един връх.

TSORT-KAHN(Даг $G = (V, E)$, като $V = \{1, \dots, n\}$)

```

1    $S \leftarrow \emptyset$ 
2   for  $i \leftarrow 1$  to  $n$ 
3      $A[i] \leftarrow 0$ 
4   for  $x \leftarrow 1$  to  $n$ 
5     foreach  $y \in adj[x]$ 
6        $A[y]++$ 
7   for  $i \leftarrow 1$  to  $n$ 
8     if  $A[i] = 0$ 
9        $S \leftarrow S \cup \{i\}$ 
10       $color[i] \leftarrow gray$ 
11    else
12       $color[i] \leftarrow white$ 
13    $k \leftarrow 0$ 
14   while  $S \neq \emptyset$  do
15      $x \leftarrow$  произволен елемент на  $S$ 
16      $S \leftarrow S \setminus \{x\}$ 
17      $k++$ 
18      $T[k] \leftarrow x$ 
19     foreach  $y \in adj[x]$ 
20        $A[y]--$ 
21       if  $A[y] = 0$ 
22          $S \leftarrow S \cup \{y\}$ 
23          $color[y] \leftarrow gray$ 
24        $color[x] \leftarrow black$ 
25     if  $k < n$ 
26       return “Грешка!  $G$  не е даг.”
27     else
28       return  $T[1, \dots, n]$ 
```

Множеството S може да бъде реализирано например чрез АТД стек или опашка. Аргументацията за коректност и сложност остава валидна и в двата случая.

Коректност. Във всеки момент от работата на алгоритъма, нека B означава множеството от черните върхове, нека A_B е множеството от клетките на A , които съдържат елементите на B , и нека $A_{\bar{B}}$ е множеството от останалите клетки на A .

Да си припомним, че “ $G - B$ ” означава графа, получен от G с изтриване на върховете от B .

Приемаме за очевиден ефектът от работата на редове 1–13, поради което го обобщаваме в следното наблюдение без доказателство.

Наблюдение 30: Инициализацията (редове 1–13) на топол. сортиране на Kahn

При първото достигане на ред 14 е вярно, че:

- $B = \emptyset$.
- S е множеството от източниците на G и се състои точно от сивите върхове.
- A съдържа полуустепените на входа на върховете на G .

Теорема 30: Коректност на топологическото сортиране на Kahn

Ако входният G е даг, алгоритъмът на Kahn го сортира топологически чрез масива T . В противен случай, алгоритъмът на Kahn връща съобщение за грешка.

Доказателство: Да допуснем, че G е даг. Следното твърдение е инвариантна за **while**-цикъла (редове 14–24).

Инвариант 17: while-цикълът на алгоритъма на Kahn

Всеки път, когато изпълнението е на ред 14,

- ① S е множеството от източниците на $G - B$ и се състои точно от сивите върхове.
- ② B е множеството $\{T[i] \mid 1 \leq i \leq k\}$.
- ③ Нито един връх от B не е дете на връх от $G - B$.
- ④ $A_{\bar{B}}$ съдържа полуустепените на входа на $G - B$.
- ⑤ $T[1, \dots, k]$ съдържа топологическа сортировка на подграфа на G , индуциран от B .

База. Разглеждаме първото достигане на ред 14.

Ще покажем ①. Според Наблюдение 30, в този момент $B = \emptyset$, така че $G - B = G$ и ① става “ S е множеството от източниците на G и се състои точно от сивите върхове”, което е част от Наблюдение 30.

Ще покажем ②. От една страна, в този момент $B = \emptyset$. От друга страна, в този момент $k = 0$ заради ред 13, така че $T[1, \dots, k]$ е празен.

Ще покажем ③. Тъй като $B = \emptyset$, твърдението е вярно в празния смисъл.

Ще покажем ④. Щом $B = \emptyset$, то $A_{\bar{B}} = A$ и ④ става “ A съдържа полуустепените на входа на върховете на G ”, което е част от Наблюдение 30.

Ще покажем ⑤. Щом $k = 0$, то $T[1, \dots, k]$ е празен. От друга страна, подграфът на G , индуциран от B , е празният граф. Ерго, ⑤ е вярно в празния смисъл. ✓

Поддръжка. Да допуснем, че инвариантата е в сила в даден момент t , в който изпълнението е на ред 14 и **while** ще бъде изпълнен поне още веднъж. Последното влече, че Q не е празна. Нека t' е моментът на следващото достигане на ред 14.

Ще покажем ①. На ред 16, x престава да е елемент на S , а на ред 24, x става черен и по този начин става елемент на B . Тогава в момента t' , x вече хем не е сив, хем не е връх на $G - B$.

Множеството от източниците в $G - B - x$ се състои точно от тези върхове, които в $G - B$ имат един единствен родител, който е x . При допускането, че ④ е в сила в момента t , това са точно тези деца y на x , за които $A[y] = 1$ преди да бъдат “обработени” от **for**-цикъла (редове 19–23). Тези деца получават A -стойност нула на ред 20, влизат в S на ред 22, а на ред 23 стават сиви. Но $G - B - x$ от момент t е $G - B$ в момент t' заради ред 24, така че ① остава в сила в t' .

Ще покажем ②. Но ② е в сила в момента t , след което, от една страна, точно един връх от сив става черен (връх x) и никой черен връх не си мени цвета, а от друга страна, k бива инкрементиран на ред 17 и, спрямо новото k , $T[k]$ получава стойност x .

Ще покажем ③. По допускане (①), в момента t , връх x е източник в $G - B$, понеже $x \in S$ в този момент. Това означава, че x не е дете на нито един връх от $G - B$ в момент t . След това към B се добавя един единствен връх, а именно x (ред 24). Очевидно в момент t' , ③ е в сила за текущото B .

Ще покажем ④. Изпълнението на тялото на **while**-цикъла засяга $A_{\bar{B}}$ по два начина. Първо, връх x , който в момент t не е в B , “влиза” в B на ред 24; ерго, в момент t , връх x е в $A_{\bar{B}}$, но в t' вече не е в $A_{\bar{B}}$. Що се отнася само до x , твърдение ④ е истина, понеже, от една страна, x престава да е връх от $G - B$, а от друга страна, $A_{\bar{B}}$ престава да съдържа съответен нему елемент.

Второ, изпълненията на ред 20 намаляват с единица полустепента на входа на всички деца на x . Твърдим, че в момент t , $\forall y \in \text{adj}[x] : A[y] > 0$. Това следва от факта, че всяко дете y на x е връх от $G - B$, който не е източник в $G - B$, и че $A[y]$ е полустепента на входа на това дете съгласно ④.

- Защо y е връх от $G - B$? Защото от ③ знаем, че в момента t , нито един черен връх не е дете на връх от $G - B$, а x в t е връх от $G - B$ и y е дете на x .
- Защо y не е източник? Защото в момент t , източниците в $G - B$ са точно върховете в S съгласно ①, а източниците не са деца по дефиниция.

И така, при всяко изпълнение на ред 20 по време на текущото изпълнение на **while**-цикъла се декрементира положително число, което след декрементирането може да е най-малко нула. Очевидно стойностите $A[y]$ на всички деца y на x в момента t' са точно полустепените на входа на тези върхове в текущия $G - B$.

Ще покажем ⑤. Очевидно $T[1, \dots, k]$ нараства с един елемент между момент t и момент t' и този елемент е връх x от ред 15. Трябва да покажем, че в момент t' няма ребро с начало $T[k]$ и край някой $T[j]$, $1 \leq j \leq k - 1$. Но това следва от това, че

- същият връх, който е $T[k]$ в момент t' , е източник в $G - B$ съгласно ①
- и това, че нито един връх от B не е дете на връх от $G - B$ в момент t .

Терминация. Приемаме за очевидно това, че при терминирането на **while**-цикъла (което е момента, в който $S = \emptyset$ съгласно ред 14), $k = n$ т.к. G е даг. При текущото допускане, че G е даг, имаме $k = n$ при достигането на ред 28. Съгласно част ⑤ от инвариантата, $T[1, \dots, n]$ съдържа топологическа сортировка на подграфа на G , индуциран от B . Но от ② знаем, че в този момент $B = \{T[i] \mid 1 \leq i \leq k\}$, което очевидно е същото като множеството $V(G)$. И така, подграфа на G , индуциран от B , е самият G , така че масивът $T[1, \dots, n]$ съдържа топологическа сортировка на G . \square

Сложност по време. Алгоритъмът на Kahn има линейна сложност по време. Намирането на полустепените на входа на редове 2–6 става във време $\Theta(n) + \Theta(n+m) = \Theta(n+m)$, защото в редове 2–6 минаваме през всички списъци на съседства с константна работа върху всеки елемент. Намирането на източниците в G на редове 7–12 става във време $\Theta(n)$. Цялото изпълнение на **while**-цикъла (редове 14–24) става във време $\Theta(n+m)$, защото пак се минава през всички списъци на съседство, с константна работа върху елемент. Като цяло, сложността по време е $\Theta(n+m)$.

10.4 Алгоритъм за намиране на силно свързаните компоненти

Задачата за намиране на силно свързани компоненти на ориентиран граф има изненадващо много приложения, например за решаване на задачата 2SAT в логиката [51, Theorem 9.1, стр. 184].

Нека навсякъде в тази секция G е ориентиран граф. Знаем, че релацията на силна свързаност $SC(G)$ (Нотация 5) е релация на еквивалентност и подграфите на G , индуцирани от класовете на еквивалентност на $SC(G)$, се наричат силно свързаните компоненти на G .

Забележете приликата между фактор-релация (Определение 14) и фактор-граф (Определение 42).

Определение 42: Фактор-граф

Нека $R \subseteq V(G) \times V(G)$ е релация на еквивалентност. Нека \mathfrak{X} е множеството от класовете на еквивалентност на R . *Фактор-графът на G спрямо R^a* е ориентираният граф G/R с множество от върхове \mathfrak{X} , като за всеки два различни класа $P, Q \in \mathfrak{X}$, ребро (P, Q) в G/R съществува тогава и само тогава, когато съществува връх $a \in P$ и съществува връх $b \in Q$, такива че $(a, b) \in E(G)$.

^aНа английски терминът е *quotient graph*.

Несъществена разлика между фактор-релация и фактор-граф е, че фактор-релацията е рефлексивна, докато фактор-графът няма примки.

В тези лекционни записи ще разглеждаме фактор-графи само спрямо релацията на силна свързаност. Когато кажем “фактор-графът на G ”, имаме предвид “фактор-графът на G спрямо $SC(G)$ ”. Фактор-графът на G се означава с “ $G/SC(G)$ ”.

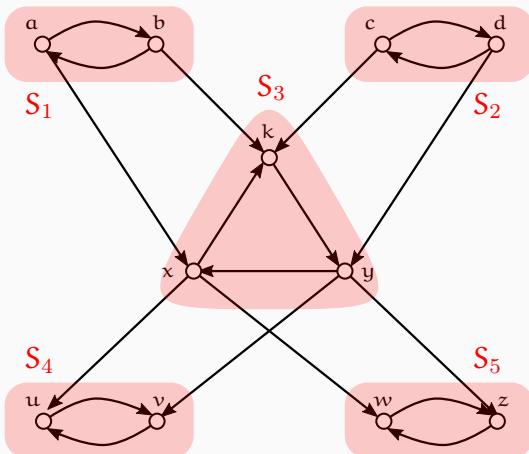
Наблюдение 31: Фактор-графът е даг.

$G/SC(G)$ е даг. Ако в $G/SC(G)$ има цикъл, за всеки два върха X и Y от цикъла—тези върхове са силно свързани компоненти в G —е вярно, че от всеки връх на X има път до всеки връх на Y в G , и от всеки връх на Y има път до всеки връх на X в G . Тогава върховете на X и Y всъщност са в една и съща силно свързана компонента на G , в противоречие с предишното допускане, според което X и Y са различни силно свързани компоненти на G .

На английски за това понятие (което нарекохме “фактор-граф”) понякога се *the condensed graph* или *the component graph* [15, стр. 617].

Фигура 10.1 показва ориентиран граф и неговият фактор-граф.

Фигура 10.1 : Ориентиран граф и неговият фактор-граф.



G и силно свърз. му компоненти.

$G/\text{SC}(G)$.

Да се опитаме да решим задачата за намиране на силно свързаните компоненти на G чрез следната малка модификация на DFS, наречена SCC-DUMMY. Както знаем, силно свързаните компоненти не са само множества от върхове, а са истински графи с върхове и ребра, но за простота генерирането на силно свързаните компоненти ще се състои в генерирането на техните множества от върхове.

SCC-DUMMY($G = (V, E); V = \{1, \dots, n\}$)

```

1    $S$  и time са глобални
2   for  $i \leftarrow 1$  to  $n$ 
3     color[ $i$ ]  $\leftarrow$  white
4   time  $\leftarrow 0$ 
5   for  $i \leftarrow 1$  to  $n$ 
6     if color[ $i$ ] = white
7        $S \leftarrow \emptyset$ 
8       SCC-DUMMY-REC( $G, i$ )
9   print  $S$ 
```

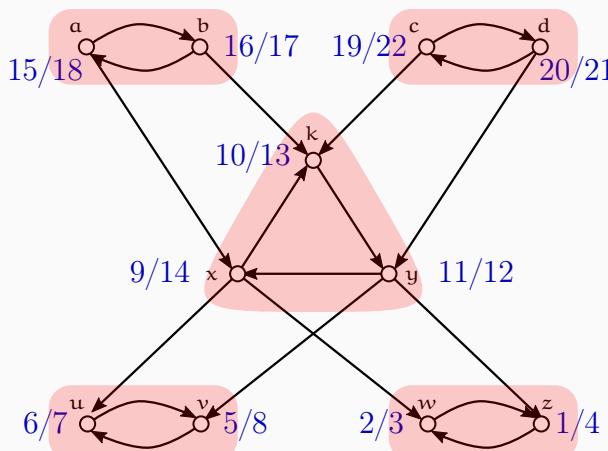
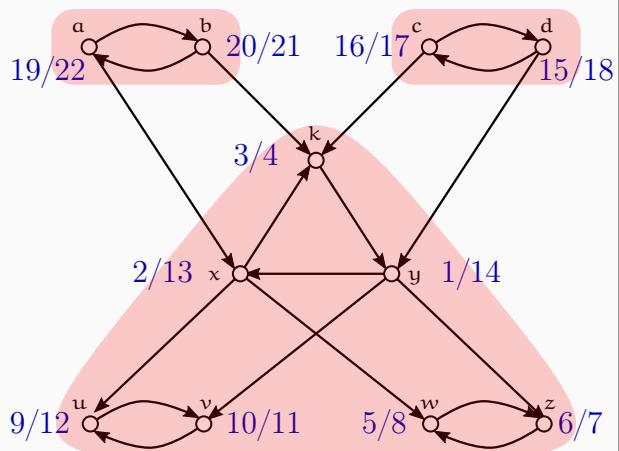
SCC-DUMMY-REC($G = (V, E); x \in V$)

```

1   color[ $x$ ]  $\leftarrow$  gray
2   time ++
3   d[ $x$ ]  $\leftarrow$  time
4    $S \leftarrow S \cup \{x\}$ 
5   foreach  $y \in \text{adj}[x]$ 
6     if color[ $y$ ] = white
7       SCC-DUMMY-REC( $G, y$ )
8   color[ $x$ ]  $\leftarrow$  black
9   time ++
10  f[ $x$ ]  $\leftarrow$  time
```

Да си представим работата на SCC-DUMMY върху графа G от Фигура 10.1. Очевидно множествата, които алгоритъмът ще принтира на ред 9, зависят от това, в какъв ред **for**-цикълът на ред 5 ще „минава“ през върховете. Това е илюстрирано на Фигура 10.2. Там е показана работата на SCC-DUMMY при две различни наредби на върховете и съответните различни разбивания. Както се вижда, при наредба 1, разбиването точно отговаря на класовете на еквивалентност на $\text{SC}(G)$, тоест, то задава силно свързаните компоненти. Но при наредба 2, разбиването е само на три класа на еквивалентност, а не на пет. Ако искаме да намираме силно свързаните компоненти, би трявало да ползваме наредба, подобна на наредба 1.

Фигура 10.2 : SCC-DUMMY при различни наредби на върховете.

наредба 1: $\langle z \ v \ u \ x \ y \ w \ k \ a \ c \ d \ b \rangle$.наредба 2: $\langle y \ w \ u \ c \ a \ b \ d \ z \ v \ x \ k \rangle$.

Лесно се вижда какъв е проблемът при наредба 2 от гледна точка на намиране на силно свързаните компоненти. Започвайки обхождането от връх y във време 1, DFS “излиза” от силно свързаната компонента на $\{k, x, y\}$ и се “прехвърля” в две други силно свързани компоненти, а именно на $\{w, z\}$ във време 5 и на $\{u, v\}$ във време 9, преди да финишира върху z във време 14. Така с едно викане на SCC-DUMMY-REC на ред 8, алгоритъмът открива три силно свързани компоненти и слага върховете им в S , което е недопустимо, ако искаме да намерим силно свързаните компоненти на G .

Наблюдение 32

За да намерим точно една силно свързана компонента при първото викане на ред 8, наредбата на ред 5 трябва да е такава, че първият връх в нея да е от силно свързана компонента, от която не може да се излезе. Наредба 1 от Фигура 10.2 е пример за такава наредба. Първият връх в нея, а именно z , е от силно свързана компонента, от която обхождането не може да излезе.

Определение 43: Благоприятна силно свързана компонента

Силно свързана компонента, от която обхождането не може да излезе, ще наричаме *благоприятна силно свързана компонента*.

Наблюдение 33

Благоприятните силно свързани компоненти на G са сифоните на фактор-графа $G/\text{SC}(G)$. Тъй като $G/\text{SC}(G)$ е даг, той има поне един сифон съгласно Теорема 28, така че G има поне една силно свързана компонента.

За да намерим силно свързаните компоненти чрез DFS, не е достатъчно първият връх в наредбата на ред 5 да е от благоприятна силно свързана компонента G_1 . След излизането от викането на ред 8 трябва следващият бял връх, за който условието на ред 6 е истина, да е от благоприятна силно свързана компонента на $G - V(G_1)$ (за да не може DFS да излезе

от нея), и така нататък до края на **for**-цикъла на ред 5. Не е необходимо върховете на силно свързаните компоненти да са непрекъснати подредици в наредбата на върховете на **for**-цикъла. Достатъчно е в тази наредба първите върхове от силно свързаните компоненти да са подредени отляво надясно така, че всеки нов тях да е от благоприятна силно свързана компонента в графа G , от който са изтрити вече обходените силно свързани компоненти.

Определение 44: Важните ребра

Важните ребра на G са точно тези ребра, чиито начало и край са от различни силно свързани компоненти.

Например, на графа G от Фигура 10.1, важните ребра са (a, x) , (b, k) , (c, k) , (d, y) , (x, u) , (y, v) , (x, w) и (y, z) , а останалите ребра не са важни.

Наблюдение 34

За да намерим силно свързаните компоненти на G чрез SCC-DUMMY, необходимо и достатъчно е наредбата на върховете по отношение на ред 5 да е такава, че за всяко важно ребро (p, q) , краят q да е преди началото p . Разположението на важните ребра в тази наредба трябва да обратното на разположението на ребрата в топологическа сортировка: в топо-сортирането се иска всички ребра да са “вчесани” отляво надясно, а сега се иска важните ребра да са вчесани “отдясно наляво”. В този смисъл, въпросната наредба, по отношение на важните ребра, трябва да е обратната на топологическата.

И така достигаме до идеята, че ако алгоритъмът получи като вход “правилна” наредба на върховете, тоест, масив от върховете, подреден по правилния от гледна точка на намиране на силно свързани компоненти начин, той ще работи коректно.

$\text{SCC}(G = (V, E); L[1, \dots, n])$: масив от върховете

```

1   S и time са глобални
2   for i ← 1 to n
3       color[i] ← white
4   time ← 0
5   for i ← 1 to n
6       if color[L[i]] = white
7           S ← ∅
8           SCC-REC(G, L[i])
9   print S

```

$\text{SCC-REC}(G = (V, E); x \in V)$

```

1   color[x] ← gray
2   time++
3   d[x] ← time
4   S ← S ∪ {x}
5   foreach y ∈ adj[x]
6       if color[y] = white
7           SCC-REC(G, y)
8   color[x] ← black
9   time++
10  f[x] ← time

```

Като пример да вземем пак Фигура 10.2. Ако $L = [z, v, u, x, y, w, k, a, c, d, b]$ (подфигурата вляво), алгоритъмът ще намери силно свързаните компоненти.

Ефикасен алгоритъм за намиране на силно свързаните компоненти на ориентиран граф. Следният алгоритъм решава ефикасно задачата за намиране на силно свързаните компоненти. Открит е от Rao Kosaraju през 1978 [1]. Необходимо ни е следното определение.

Определение 45: Транспониран граф

Транспонираният граф на G е графът $G^T = (V(G), \{(u, v) \mid (v, u) \in E(G)\})$.

На английски терминът е *the transpose of G* ([15, стр. 616]).

Алгоритъмът на Kosaraju вика алгоритъмът за Tarjan на стр. 289 за топологическо сортиране. Това, че сега G може да не е даг, **няма значение**. Забележете, че алгоритъмът на Tarjan, ако получи не-даг на входа, не установява, че графът не е даг! Той подрежда върховете по намаляващи стойности на време на финализиране, което може да бъде направено за всеки граф; ако графът не е даг, то изходът няма смисъл на топологическо сортиране на графа, но това е друго нещо.

KOSARAJU(Ориентиран граф $G = (V, E)$)

- 1 $L[1, \dots, n] \leftarrow \text{TSORT}(G)$
- 2 намери G^T
- 3 $\text{SCC}(G^T, L)$

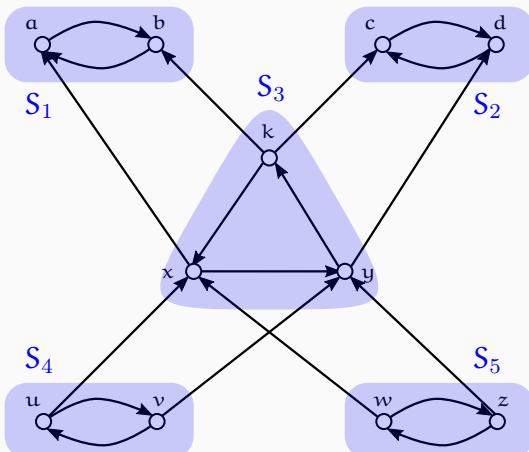
Коректност. Аргументацията за коректност се основава на следните очевидни факти.

Наблюдение 35

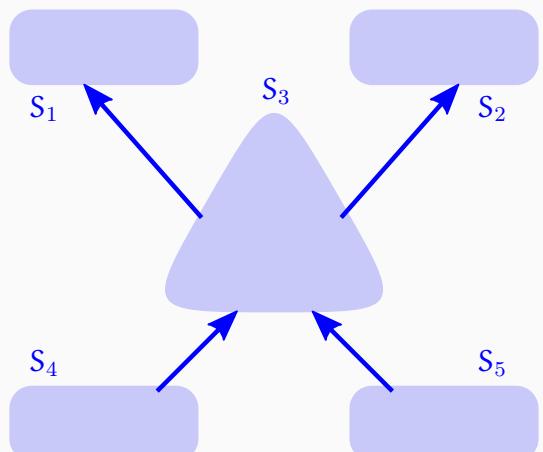
G и G^T имат практически същите силно свързани компоненти в следния смисъл: $\text{SC}(G)$ и $\text{SC}(G^T)$ съвпадат, така че разбиванията на върховете (в класове на еквивалентност на силна свързаност) в графа и в неговия транспониран граф са еднакви. Нещо повече: $(G/\text{SC}(G))^T = G^T/\text{SC}(G^T)$. Оттук следва, че източниците в $G/\text{SC}(G)$ са точно сифоните в $G^T/\text{SC}(G^T)$ и обратното, сифоните в $G/\text{SC}(G)$ са точно източниците в $G^T/\text{SC}(G^T)$.

Фигура 10.3 илюстрира това наблюдение. Тя показва транспонириания граф на графа от Фигура 10.1, както и неговия фактор-граф, който очевидно е транспонириания граф на фактор-графа от Фигура 10.1.

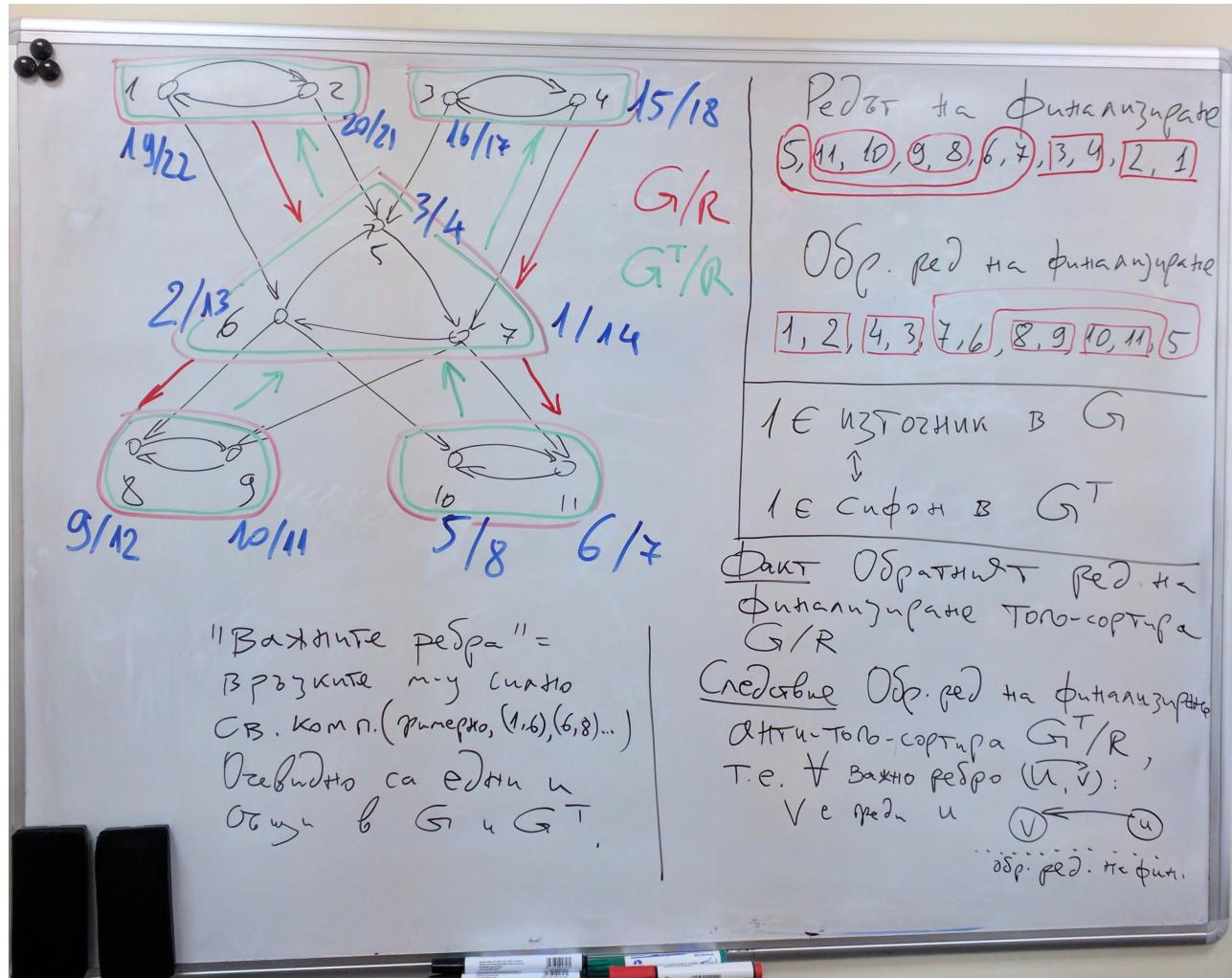
Фигура 10.3 : Транспонирианият граф на графа от Фигура 10.1.



Силно свързаните компоненти са практически същите.



Фактор-графът е транспониран.



Сложност по време. Алгоритъмът на Kosaraju има линейна сложност по време. Работата на TSORT има линейна сложност – факт, в който вече се убедихме. Това, че сега графът-вход не е непременно даг, няма значение за това. Създаването на транспонирания граф също има линейна сложност: то може да се реализира чрез едно “минаване” през списъците на съседства на G , като, щом срещнем връх v в списъка на връх u (това означава ребро (u, v) в G), слагаме u в списъка на v в представянето на G^T (това има смисъл на слагане на ребро (v, u) в G^T). И накрая, работата на SCC се извършва очевидно в линейно време.

10.5 Алгоритми за намиране на срязващите върхове и на мостовете

10.5.1 Фундамент

Сега разглеждаме само неориентирани графи. В тази секция, $G = (V, E)$ означава неориентиран свързан граф (без примки, не е мултиграф).

Определение 46: Срязващ връх и мост

Срязващ връх в G е всеки $u \in V$, такъв че $G - u$ има повече от една свързани. Срязващо ребро, още се нарича мост, е всяко $e \in E$, такова че $G - e$ има две свързани компоненти.

Определение 47: Срязващ връх и мост, алтернативно определение

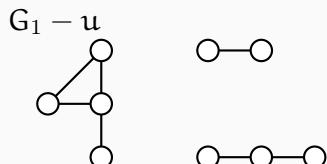
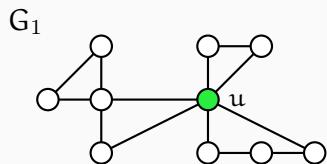
Срязващ връх в G е всеки $u \in V$, такъв че за някои различни върхове $v, w \in V$ е вярно, че всеки път между v и w съдържа u като вътрешен връх. Мост е всяко $e \in E$, такова че за някои различни върхове $v, w \in V$ е вярно, че всеки път между v и w съдържа e .

Лема 43

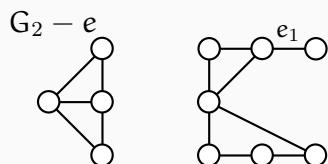
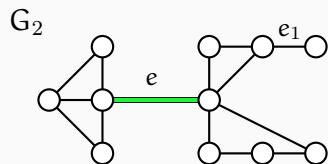
Определения 46 и 47 са еквивалентни.

Лесно се вижда, че на всеки висящ връх в свързана компонента с повече от два върха съответства точно един срязващ връх, а именно единственият му съсед. Самият висящ връх не е срязващ. Ако свързаната компонента има точно два върха, или с други думи, ако е едно единствено ребро, то в нея срязващ връх няма. На всеки висящ връх съответства точно един мост, а именно единственото ребро, инцидентно с него; това остава в сила дори свързаната компонента да се състои само от това ребро. Фигура 10.4 илюстрира Определения 46 и 47. В графа G_1 , връх u е срязващ. В графа G_2 , и e , и e_1 са мостове, но е показано само изтриването на e .

Фигура 10.4 : Срязващ връх и срязващо ребро.



u е срязващ връх.



e е срязващо ребро (e_1 също).

Приложение на срязващи върхове и ребра. Срязващите върхове и мостовете са важни понятия, ако графът моделира някаква свързаност. Например, ако моделира комуникационна мрежа, или пътна мрежа, или електрическа мрежа.

Нека графът моделира комуникационна мрежа от компютри и жични връзки между тях. Нека мрежата е свързана в смисъл, че всеки два компютъра могат да си “говорят”; може би не директно, а през други компютри, но имат връзка един с друг. Тогава срязващ връх в графа съответства на компютър, чиято повреда би довела до това, че мрежата би се разпадала на две или повече подмрежи, които не си “говорят”. В разпадната мрежа има компютри, които не могат да комуникират взаимно. А срязващо ребро отговаря на жична връзка, чиято повреда би довела до разпадане на мрежата на две подмрежи, които не си “говорят”.

И така, срязващите върхове и мостовете отговарят на някакъв критично важни компоненти на мрежата. Мрежа, чийто съответен граф има срязващи върхове или мостове е потенциално по-ненадеждна от мрежа, чийто съответен граф няма такива елементи. От гледна точка

на надеждността, очевидно предпочитаме мрежи, чито съответни графи нямат срязващи върхове или мостове.

Определение 48: Разцепване на срязващи върхове

Нека $u \in V$ е срязващ връх. Нека $G - u$ има k свързани компоненти, наречени G_1, \dots, G_k . Нека z_1, \dots, z_k са k нови върхове – такива, които не са от V . Нека, за $1 \leq i \leq k$,

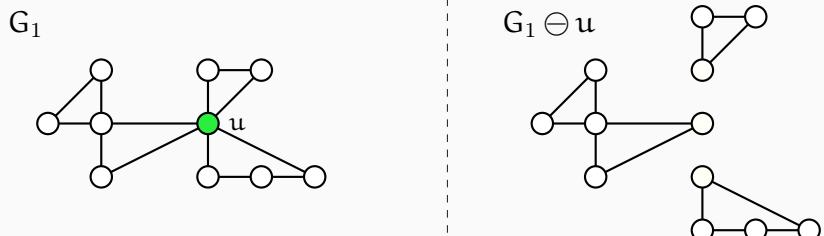
$$H_i = (V(G_i) \cup \{z_i\}, E(G_i) \cup \{(z_i, x) \mid x \in V(G_i) \cap N(u)\})$$

Да разцепим u означава да преобразуваме G в колекцията H_1, \dots, H_k . Графа–результат означаваме с $G \ominus u$.

Очевидно H_1, \dots, H_k са свързани графи.

Неформално казано, разцепването на срязващ връх u се състои в създаването на негови копия (самият u изчезва), на брой колкото са свързаните компоненти **спрямо него**, като всяко копие бива свързано с точно една от тези компоненти, с точно тези върхове в нея, с които u е бил свързан. Фигура 10.5 показва разцепване на връх – след разцепването на u се появяват три свързани компоненти.

Фигура 10.5 : Разцепване на връх.



Операцията “разцепване на срязващ връх” е асоциативна: можем да разцепваме различни върхове в каквато искаме последователност и резултатът е един и същи. Ако u_1, \dots, u_t са различни срязващи върхове в G , пишем $G \ominus u_1 \ominus \dots \ominus u_t$. Резултатът от разцепването на всички срязващи върхове е предмет на следващата дефиниция.

Определение 49: Блокове в граф

Нека $G = (V, E)$ е граф. *Блок* в G е всеки максимален по включване свързан подграф, който не съдържа срязващи върхове.

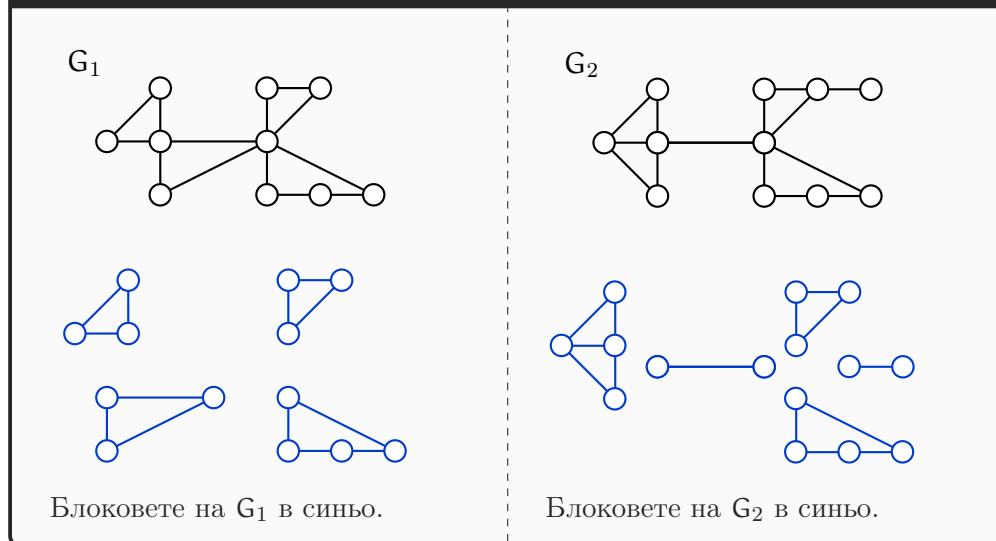
Определение 50: Блокове в граф, алтернативна дефиниция

Нека $G = (V, E)$ е граф. *Блоковете на G* са тези свързани компоненти, които се получават от разцепването на всички срязващи върхове.

Наблюдение 36

Определения 49 и 50 са еквивалентни.

Фигура 10.6 илюстрира блоковете на два графа G_1 и G_2 . Тя илюстрира и “разцепване на срязващи върхове”.

Фигура 10.6 : Блокове в графи.

Има много алгоритми върху графи—например, алгоритмите за планарност—които извършват като предварителна обработка следните декомпозиции. Първо, намират свързаните компоненти, което може да стане ефикасно с BFS или DFS. Очевидно, графът е планарен тества всяка свързана компонента е планарна. После за всяка компонента намират срязващите върхове и блоковете. Очевидно, графът е планарен тества блоковете са планарни.

Това е друго важно приложение на срязващите върхове: декомпозицията на граф на блокове.

10.5.2 Алгоритъм за ефикасно намиране на срязващите върхове

Ако конструираме алгоритъм за намиране на срязващите върхове на G направо от Определение 46, той би бил следният. За всеки връх u , изтриваме u от G и с BFS или DFS проверяваме дали G остава свързан; ако остава, то u не е срязващ връх, ако ли не, то u е срязващ връх и го добавяме към множеството от срязващите върхове, после връщаме u и продължаваме по същия начин. Това би бил твърде неефикасен алгоритъм, дори да можем да трием и връщаме върхове в $\Theta(1)$: сложността му в най-лошия случай би била $\Theta(n(n + m))$.

Ефикасният алгоритъм, който ще разгледаме сега, е базиран на [63, Section 5.9.2, pp. 173–177]. Това е DFS, модифициран за намиране на срязващи върхове.

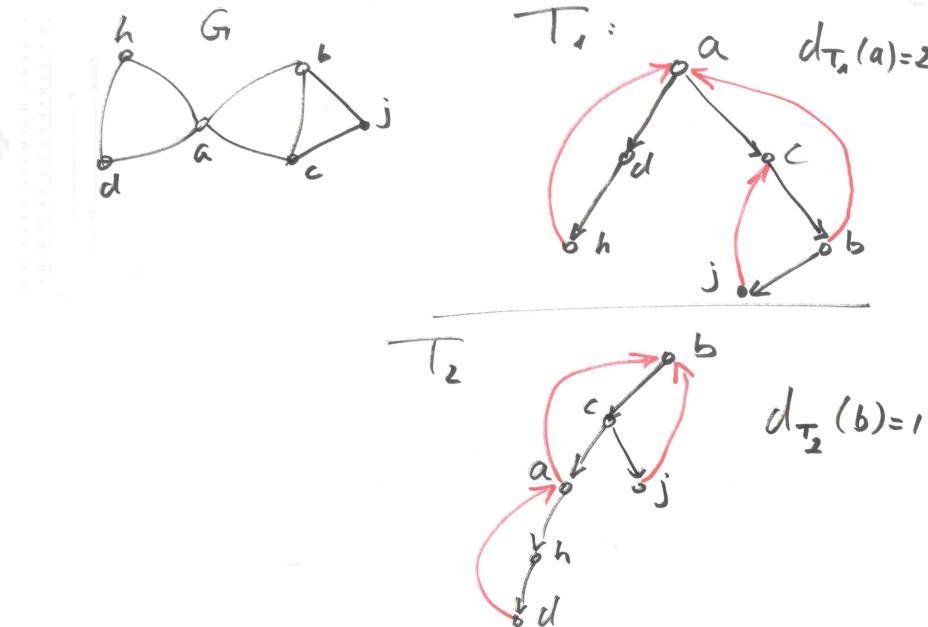
Всяко изпълнение на DFS върху свързан граф $G = (V, E)$ генерира дърво на обхождането T . Знаем, че $V(T) = V$ and $E(T) = \{e \in E \mid e \text{ е класифицирано като дървесно ребро от DFS}\}$. Разглеждаме три категории върхове на **графа** спрямо **дървото** и необходимите и достатъчни условия за това, връх от дадена категория да е срязващ.

Лема 44: Листата на дървото

В текущия контекст, нито едно листо не може да е срязващ връх. Ако v е листо на T , очевидно $G - v$ има точно една свързана компонента.

Лема 45: Коренът на дървото

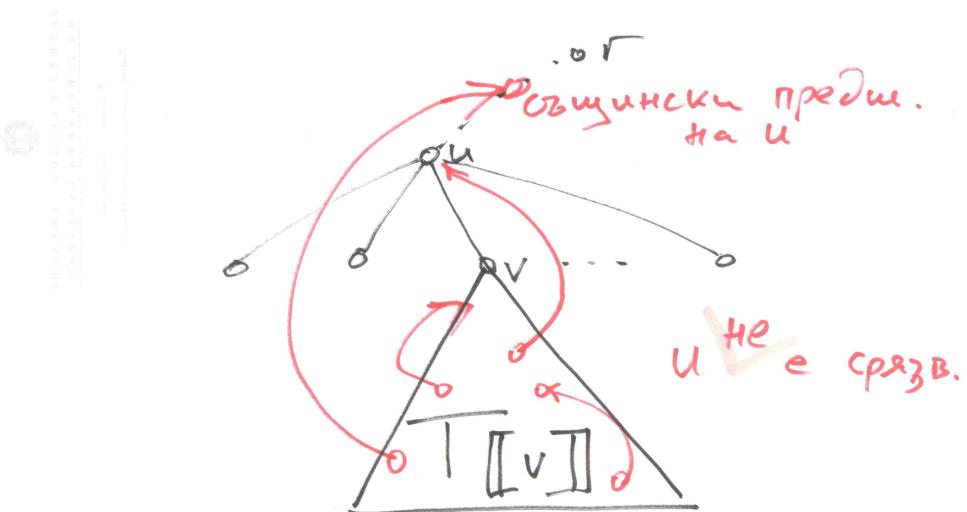
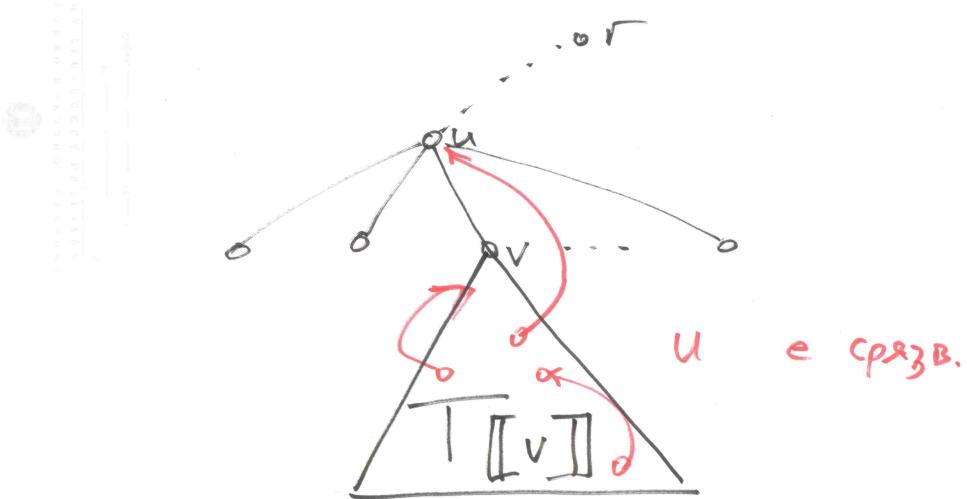
В текущия контекст, коренът е срязващ връх в G тък степента на му в дървото е поне 2.

Доказателство:**Лема 46: Останалите върхове на дървото**

В текущия контекст, всеки връх u , който не е листо или коренът, е срязващ връх тък има дете v на u в T , такова че няма нито едно ребро назад от връх на $T[v]$ към същински предшественик на u в дървото.

Доказателство: Да допуснем, че има дете v на u (в дървото), такова че няма ребро назад от никой връх от $V(T[v])$ към същински предшественик на u в дървото. Това влече, че за всеки $x \in V(T[v])$ и всеки $y \in V(G) \setminus (V(T[v]) \cup \{u\})$, за всеки път p в G между x и y е вярно, че u е вътрешен връх в p . Тогава u е срязващ връх в G .

В обратната посока, нека за всяко дете v на u (в дървото) има ребро назад между връх $x_v \in V(T[v])$ към същински предшественик y_v на u в дървото. Тогава за всеки два върха a и b в G е вярно, че в G има път $p_{a,b}$, който не съдържа u като вътрешен връх. Тогава u не е срязващ връх.



Следният алгоритъм имплементира тези идеи в модификация на DFS. В него няма масиви $d[1, \dots, n]$, $f[1, \dots, n]$ и $\pi[1, \dots, n]$, а също така няма и променлива $time$. Има масив от нивата $level[1, \dots, n]$, като $level[u]$ е нивото на връх u (разстоянието до корена) в дървото T (а не в целия G : знаем, че DFS в общия случай не изчислява разстояния в графа). Очевидно нивото на корена е 0.

FIND CUT VERTICES($G = (V, E)$): неориентиран свързан граф)

1 $level[1, \dots, n]$ е глобален

```

2   foreach  $u \in V$ 
3      $\text{color}[u] \leftarrow \text{white}$ 
4    $u$  е произволен връх от  $V$ 
5    $\text{FCV-REC}(G, u, 0);$ 

```

$\text{FCV-REC}(G = (V, E), x \in V, \ell \in \mathbb{N})$

```

1    $\text{color}[x] \leftarrow \text{gray}$ 
2    $\text{level}[x] \leftarrow \ell$ 
3    $\text{minback} \leftarrow \text{level}[x]$ 
4   if  $\text{level}[x] = 0$ 
5      $\text{isroot} \leftarrow \text{TRUE}$ 
6   else
7      $\text{isroot} \leftarrow \text{FALSE}$ 
8    $\text{count} \leftarrow 0$ 
9    $\text{iscutvertex} \leftarrow \text{FALSE}$ 
10  foreach  $y \in \text{adj}[x]$ 
11    if  $\text{color}[y] = \text{white}$  {
12       $\text{count}++$ 
13       $b \leftarrow \text{FCV-REC}(G, y, \ell + 1)$ 
14      if  $b \geq \text{level}[x]$  and (not  $\text{isroot}$ )
15         $\text{iscutvertex} \leftarrow \text{TRUE}$ 
16         $\text{minback} \leftarrow \min\{\text{minback}, b\}$  }
17    if  $\text{color}[y] = \text{gray}$  {
18      if  $\text{level}[y] < \text{minback}$  and  $\text{level}[y] \neq \text{level}[x] - 1$ 
19         $\text{minback} \leftarrow \text{level}[y]$  }
20    if  $\text{iscutvertex}$  or (isroot and  $\text{count} \geq 2$ )
21      print  $x$ 
22     $\text{color}[x] \leftarrow \text{black}$ 
23  return  $\text{minback}$ 

```

Коректност. Променливата minback , както показва и името ѝ, която FCV-REC връща, има смисъл на минимално ниво (най-близо до корена), към връх от което има ребро, другият край на което ребро е връх от $T[x]$.

Лема 44, Лема 45 и Лема 46 са достатъчна обосновка за коректността, тъй като рекурсивната функция FCV-REC работи така, че да имплементира техните необходими и достатъчно условия. Функцията очевидно не категоризира никое листо като срязващ връх, отбележава корена като срязващ връх тъкъм има поне две деца (в дървото), а за всеки друг вид връх x сравнява на ред 14 разстоянието между връха и корена $\text{level}[x]$ с минималното разстояние b между корена и най-близък до корена връх, към който има ребро назад от връх от $T[y]$, по всички деца y на x в дървото. При допускането, че рекурсивното викане на ред 13 връща стойност b с именно такъв смисъл, Лема 46 дава желаната аргументация веднага.

Има една особеност при изчисляването на minback на редове 17–19. Щом y е сив, реброто (x, y) е ребро назад. Естествено, алгоритъмът трябва да “прегледа” всички ребра назад между x и връх, който е предшественик на x в дървото, за да намери истинската стойност на minback . Но реброто между x и родителят на x в дървото не е ребро назад, а е дървесно ребро, така че трябва да бъде прескочено в това “преглеждане”. Смисълът на $\text{level}[y] \neq \text{level}[x] - 1$ на ред 18 е точно този. Забележете, че, ако x не е коренът, има един единствен негов съсед y , за който $\text{level}[y] = \text{level}[x] - 1$ и това е родителят на x в дървото.

Сложност. Очевидно сложността е $\Theta(m+n)$, което следва веднага от това, че алгоритъмът обхожда еднократно списъците на съседства и върши само $\Theta(1)$ работа за всеки елемент от тях.

10.5.3 Алгоритъм за за ефикасно намиране на мостовете

Подходът с брутална сила за намиране на мостовете би работил, в най-бруталния си вариант, в $\Theta(m(n+m))$: за всяко ребро e , изтриваме e от G и с BFS или DFS проверяваме дали G остава свързан; ако остава, то e не е мост, ако ли не, то e е мост и го добавяме към множеството от мостовете, после връщаме e и продължаваме по същия начин.

Задачата за намиране на мостовете е много подобна на задачата за намиране на срязващите върхове, откъдето е и приликата между алгоритмите FIND CUT VERTICES и FIND BRIDGES.

Лема 47

В текущия контекст, за всяко ребро (u, v) е вярно, че то е мост тък DFS(G) го категоризира като дървесно и, ако u е бащата на v в дървото на обхождането, то няма ребро назад от никой връх на $T[v]$ към u или същински предшественик на u .

Доказателство: В едната посока, да допуснем, че (u, v) е дървесно ребро, като u е бащата на v , и няма ребро назад от никой връх на $T[v]$ към u или същински предшественик на u . Тогава за всеки $x \in V(T[v])$ и всеки $y \in V(G) \setminus (V(T[v]) \cup \{u\})$, за всеки път p в G между x и y е вярно, че (u, v) е ребро в p , което означава, че (u, v) е мост.

В обратната посока, нека има ребро назад от връх $x \in V(T[v])$ към u или същински предшественик на u . Тогава за всеки два върха a и b в G е вярно, че в G има път $p_{a,b}$, който не съдържа реброто (u, v) . Тогава (u, v) не е мост. \square

За разлика от задачата за намиране на срязващите върхове, сега не е необходимо да третираме корена, листата и останалите върхове по различни начини.

FIND BRIDGES($G = (V, E)$): неориентиран свързан граф)

- 1 level[1, ..., n] е глобален
- 2 **foreach** $u \in V$
- 3 color[u] \leftarrow white
- 4 u е произволен връх от V
- 5 FBR-REC($G, u, 0$)

FBR-REC($G = (V, E)$, $x \in V$, $\ell \in \mathbb{N}$)

- 1 color[x] \leftarrow gray
- 2 level[x] $\leftarrow \ell$
- 3 minback \leftarrow level[x]
- 4 **foreach** $y \in \text{adj}[x]$
- 5 **if** color[y] = white {
- 6 b \leftarrow FBR-REC($G, y, \ell + 1$)
- 7 **if** b > level[x]
- 8 print (x, y)
- 9 **else**
- 10 minback $\leftarrow \min \{minback, b\}$

```

11   if color[y] = gray {
12       if level[y] < minback and level[y] ≠ level[x] - 1
13           minback ← level[y] }
14   color[x] ← BLACK
15   return minback

```

Коректност и сложност. Доказателството за коректност почива на Лема 47. Ако рекурсивното викане на ред 6 връща коректно минималния номер на ниво, към което има ребро назад от връх от $T[y]$, съхранявайки го в b , то, съгласно Лема 47:

- ако $b > \text{level}[x]$, тоест, ако това ниво е “под x ”, то (x, y) е мост;
- в противен случай, (x, y) не е мост. Нещо повече, в този случай има смисъл да се опитаме да подобрим (тоест, намалим) minback на ред 10. В предишния случай няма смисъл от такъв опит.

Действията на алгоритъма върху сивите съседи на x (редове 11–13) и тяхната обосновка са точно както във FIND CUT VERTICES.

Сложността по време е очевидно $\Theta(m + n)$.

Лекция 11

Намиране на минимални покриващи дървета. Алгоритми на Prim и Kruskal.

Резюме: Въвеждаме тегловни графи и задачата за намиране на минимално покриващо дърво на тегловен граф. Разглеждаме два ефикасни алгоритъма за намиране на минимални покриващи дървета: алгоритмите на Prim и Kruskal. Заради ефикасността на алгоритъма на Kruskal въвеждаме и изследваме Union-Find структури данни.

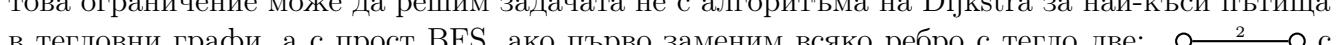
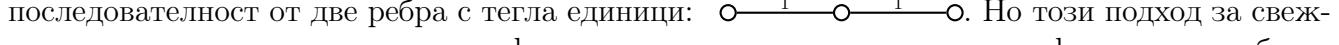
11.1 Фундамент

Определение 51: Тегловен граф

Тегловен граф е съвкупността от граф $G = (V, E)$ и тегловна функция $w : E \rightarrow \mathbb{R}^+$.

Говорим за *тегла на ребрата*[†]. На английски терминът за “тегловен граф” е *weighted graph*, откъдето идва и *edge weights*. Освен “тегловна”, тази функция понякога бива наричана *ценова функция*, откъдето може да говорим и за *цени на ребрата*, имайки предвид същото нещо като “тегла на ребрата”. Използваме термините *леките/евтините ребра* и *тежките/скъпите ребра* с очевидния смисъл.

Тегловните графи имат много по-голяма моделираща мощ от обикновените, не-тегловни графи. Очевидно е, че не-тегловните графи са частен случай на тегловните: просто си представяме, че всяко ребро има тегло единица.

Има ограничени случаи, в които задача върху тегловен граф може да бъде сведена до задача върху не-тегловен граф и да използваме алгоритъм за не-тегловния граф, за да я решим ефикасно върху тегловния. Пример за това е задачата за най-къси пътища в тегловен граф, където тегловната функция е доста ограничена, да кажем $w : E \rightarrow \{1, 2\}$. При това ограничение може да решим задачата не с алгоритъма на Dijkstra за най-къси пътища в тегловни графи, а с прост BFS, ако първо заменим всяко ребро с тегло две:  с последователност от две ребра с тегла единици:  Но този подход за свеждане на задачи върху тегловни графи към задачи върху не-тегловни графи не са машабира. Ако ограничението за теглата е $w : E \rightarrow \{1, \dots, 1\ 000\ 000\}$, този подход би налагал да заменим ребро с тегло един милион с път от един милион ребра, всяко с тегло единица. Това е все едно да записваме числата в унарна бройна система; големината на теглото става брой ребра. В общия случай, този подход би довел до експоненциално нарастване на големината на примера, ако отчитаме размера на представянето на числата, или до неограничено нарастване на

[†]Възможно е да се въведат тегла и на върховете, но ние ще разглеждаме тегла само на ребрата.

примера в големината на входа, ако игнорираме размера на представянето на числата. Да не говорим за проблемите на този подход при тегла, които не са цели числа или са отрицателни числа. И така, тегловните графи са строго по-мощно средство за моделиране на житейски задачи от не-тегловните графи.

Тегловни могат да бъдат както неориентирани, така и ориентирани графи. В тази лекция разглеждаме само неориентирани графи, тегловни или не.

Определение 52: Покриващ подграф

Нека $G = (V, E)$ е граф. *Покриващ подграф* на G е всеки подграф $G' = (V, E')$ на G . *Покриващо дърво* на G , съкратено ПД, е покриващ подграф на G , който е дърво.

На английски термините са съответно *spanning subgraph* и *spanning tree*. Ние няма да разглеждаме други видове покриващи подграфи освен покриващи дървета, но принципно има смисъл да се обобщи “покриващо дърво” до “покриващ подграф”. На читателя остава да съобрази, че покриващ цикъл е същото като Хамилтонов цикъл.

Наблюдение 37

За всеки граф G , G има поне едно покриващо дърво тстк G е свързан.

Броят на покриващите дървета е най-много n^{n-2} и тази граница е точна. Този резултат обаче е нетривиален и затова го показваме в Допълнение 31. Това, което трябва да запомним, е, че в общия случай, броят на покриващите дървета е грамаден.

Допълнение 31: За броя на покриващите дървета

Броят на покриващите дървета на граф е равен на детерминантата на една матрица, която характеризира графа. Резултатът е в сила дори за мултиграфи (наличието на примки е без значение за покриващите дървета). Този нетривиален резултат е част от алгебричната теория на графиките и е открит още през 19 век от великия физик Gustav Kirchhoff. Резултатът присъства имплицитно в негова статия [34] (превод на английски има в [33]). Подробно изследване на работата на Kirchhoff има в [32]. За подробно доказателство на резултата на Kirchhoff вижте [64, глава 9, стр. 135].

С $\kappa(G)$ означаваме броя на покриващите дървета на G .

Определение 53: Матрица на Laplace на граф

Нека $G = (\{1, \dots, n\}, E)$ е граф. *Матрицата на Laplace* на граф G е квадратна, $n \times n$, симетрична матрица L , където

$$L[i, j] = \begin{cases} d(i), & \text{ако } i = j, \\ -1, & \text{ако } i \neq j \text{ и } (i, j) \in E, \\ 0, & \text{в противен случай} \end{cases}$$

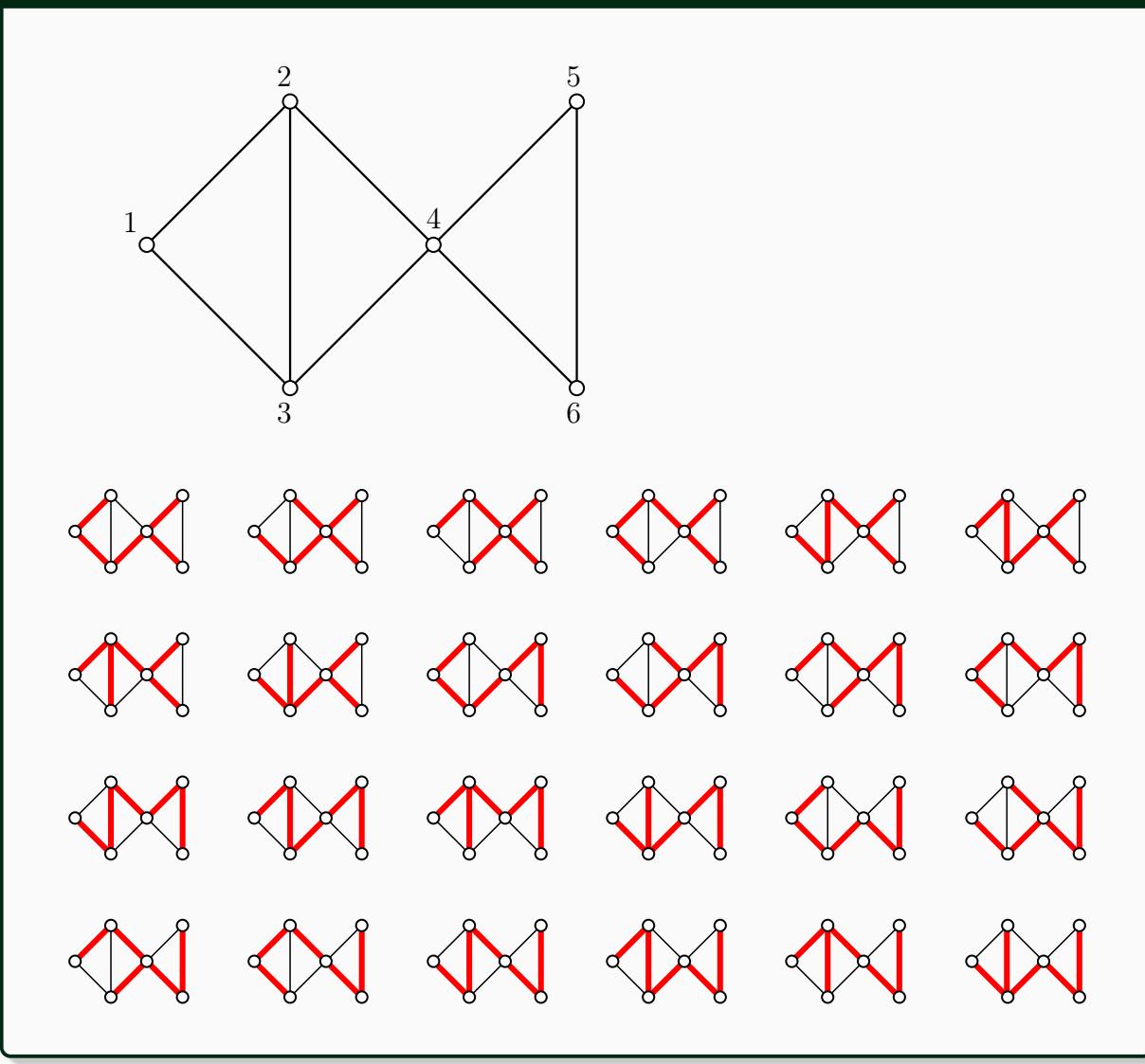
Теорема 31: The Matrix-Tree Theorem, [64, 9.8 Theorem, стр. 141]

Нека G е граф без примки с матрица на Laplace L . Нека L_0 означава L с последния ред и колона изтрити (или, по-общо казано, с i -ия ред и колона изтрити, за кое да е i). Тогава

$$\det(L_0) = \kappa(G)$$

Например, да разгледаме графа на Фигура 11.1. Той има 24 покриващи дървета, показани под него.

Фигура 11.1 : Граф и неговите покриващи дървета.



Матрицата на Laplace за този граф е

$$\begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 3 & -1 & -1 & 0 & 0 \\ -1 & -1 & 3 & -1 & 0 & 0 \\ 0 & -1 & -1 & 4 & -1 & -1 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & -1 & 2 \end{bmatrix}$$

Съгласно Теорема 31, броят на покриващите го дървета е

$$\kappa(G) = \det \begin{bmatrix} 2 & -1 & -1 & 0 & 0 \\ -1 & 3 & -1 & -1 & 0 \\ -1 & -1 & 3 & -1 & 0 \\ 0 & -1 & -1 & 4 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix} = 24$$

Теорема 32: Corollary 9.10-b на стр. 142 в [64]

Нека G е свързан, d -регулярен, граф без примки с матрица на съседство A . Нека собствените стойности на матрицата на съседство A са $\lambda_1, \dots, \lambda_{n-1}, \lambda_n$, като $\lambda_n = d$. Тогава

$$\kappa(G) = \frac{(d - \lambda_1)(d - \lambda_2) \cdots (d - \lambda_{n-1})}{n}$$

Теорема 33: Proposition 1.5 на стр. 4 в [64]

Собствените стойности на матрицата на съседство за пълния граф K_n са следните:

- -1 с кратност $n - 1$ и
- $n - 1$ с кратност 1.

Следствие 14

Броят на покриващите дървета на K_n е

$$\kappa(K_n) = \frac{(n - 1 - (-1))(n - 1 - (-1)) \cdots (n - 1 - (-1))}{n} = \frac{n^{n-1}}{n} = n^{n-2}$$

Разбира се, този резултат е в сила само ако пълният граф K_n е именуван и покриващите дървета са именувани. Броят на неименуваните покриващи дървета (неизоморфните покриващи дървета) е значително по-малък.

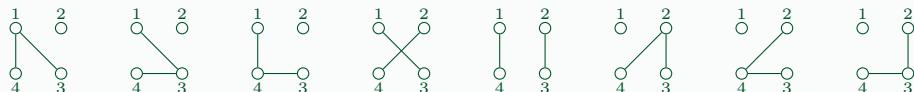
Формулата n^{n-2} за броя на покриващите дървета на K_n се нарича *формула на Cayley*. Този резултат може да се изведе и с комбинаторни, а не алгебрични средства според [2, стр. 204]. Сега мислим не за покриващите дървета на именуван K_n , а просто за възможностите да конструираме дървета над n върха.

Теорема 34: стр. 204 в [2]

Разглеждаме множество от върхове $V = \{1, \dots, n\}$. Нека A е произволно k -elementno подмножество на V . Тогава броят на именуваните гори над V , състоящи се от точно k дървета, като всеки връх от A е в различно дърво, е:

$$T_{n,k} = \begin{cases} \sum_{i=0}^{n-k} \binom{n-k}{i} T_{n-1, k-1+i}, & \text{ако } n \geq k > 1 \\ 0, & \text{ако } n \geq 1 \text{ и } k = 0 \\ 1, & \text{ако } n = 0 \text{ и } k = 0 \end{cases} \quad (11.1)$$

Първо да видим пример. Нека $n = 4$, $k = 2$ и $A = \{1, 2\}$.



Вижда се, че $T_{4,2} = 8$.

Доказателство: Началните условия са очевидни. Ще докажем рекурсивната част от формулата. Очевидно няма значение точно кои върхове са в A ; значение има само кардиналността на A . БОО, нека $A = \{1, \dots, k\}$. Фиксираме произволен връх от A . Нека това е връх 1. Индексната променлива i има смисъл на броя съседите на връх 1. Те може да са от 0 до $n - 1 - (k - 1) = n - k$, което дава и границите на сумирането в (11.1).

Изтриваме връх 1 и получаваме гора с $n - 1$ върха и с точно $k - 1 + i$ дървета, защото едно дърво изчезва, а се появяват i нови дървета. Различните гори с $n - 1$ върха и $k - 1 + i$ дървета са $T_{n-1, k-1+i}$. Връщаме връх 1, като трябва да го направим съсед на точно i върха измежду върховете от $V \setminus A$. Но $|V \setminus A| = n - k$, следователно има точно $\binom{n-k}{i}$ начина да изберем съседите на върнатия връх 1. Всяко свързване на връх 1 с i върха измежду върховете на някоя гора (общо $T_{n-1, k-1+i}$ гори) дава различна гора, което доказва и верността на (11.1).

Това, че $T_{n,n} = 1$ следва от (11.1). \square

Теорема 35: стр. 204 в [2]

В сила е

$$T_{n,k} = kn^{n-k-1} \quad (11.2)$$

Доказателство: С индукция по (11.1). Базовите случаи $n \geq 1, k = 0$ и $n = k = 0$ са очевидни.

В индуктивната стъпка имаме:

$$\begin{aligned}
 T_{n,k} &= \sum_{i=0}^{n-k} \binom{n-k}{i} T_{n-1,k-1+i} = \sum_{i=0}^{n-k} \binom{n-k}{i} (k-1+i)(n-1)^{n-1-(k-1+i)-1} \\
 &= \sum_{0 \leq i \leq n-k} \binom{n-k}{i} (k-1+i)(n-1)^{n-k-i-1} \\
 &= \sum_{0 \leq i \leq n-k} \binom{n-k}{n-k-(n-k-i)} (k-1+(n-k-i))(n-1)^{n-k-(n-k-i)-1} \\
 &= \sum_{0 \leq i \leq n-k} \binom{n-k}{i} (n-i-1)(n-1)^{i-1} \\
 &= \sum_{i=0}^{n-k} \binom{n-k}{i} (n-1)^i - \sum_{i=0}^{n-k} \binom{n-k}{i} i(n-1)^{i-1} \\
 &= \sum_{i=0}^{n-k} \binom{n-k}{i} (n-1)^i 1^{n-k-i} - \sum_{i=1}^{n-k} \binom{n-k-1}{i-1} \frac{n-k}{i} i(n-1)^{i-1} \\
 &= (n-1+1)^{n-k} - (n-k) \sum_{i=0}^{n-k-1} \binom{n-k-1}{i} (n-1)^i 1^{n-k-1-i} \\
 &= n^{n-k} - (n-k)(n-1+1)^{n-k-1} = n^{n-k} - (n-k)n^{n-k-1} \\
 &= n^{n-k} - n^{n-k} + kn^{n-k-1} = kn^{n-k-1}
 \end{aligned} \tag{11.3}$$

□

Следствие 15

В контекста на Теорема 34, $T_{n,1}$ очевидно е броят на дървата над V . Според Теорема 35, $T_{n,1} = 1n^{n-1-1} = n^{n-2}$. Тогава броят на дървата е n^{n-2} .

Определение 54: Тегло на ПД. Минимално ПД.

Нека G е свързан тегловен граф с тегловна функция w . Нека T е ПД на G . *Теглото на T* е $w(T) = \sum_{e \in E(T)} w(e)$. Минимално ПД, съкратено МПД, е такова ПД T' , че за всяко ПД T'' е вярно, че $w(T') \leq w(T'')$.

Следното определение е буквален превод на “safe edge” от [15, стр. 626].

Определение 55: Сигурно ребро.

Нека G е тегловен граф и $A \subseteq E$ е такова, че съществува МПД T , такова че $A \subseteq E(T)$. Нека e е ребро на G , което не е от A . Казваме, че e е сигурно за A , ако съществува МПД T' , такова че $A \cup \{e\} \subseteq E(T')$.

Общата идея на алгоритмите за конструиране на МПД се съдържа в следния псевдокод ([15, стр. 626]). МПД се определя напълно от своите ребра, така че да конструираме МПД е същото като да конструираме множеството от ребрата му.

Общ вид на МПД(G : граф, w : тегловна функция)

```

1    $A \leftarrow \emptyset$ 
2   while  $A$  не образува ПД do
3       намери ребро  $e$ , което е сигурно за  $A$ 
4        $A \leftarrow A \cup \{e\}$ 
5   return  $A$ 
```

Ясно е, че във всеки момент от работата на алгоритъм, изграден върху тази схема, ребрата от A дават ацикличен подграф на G ; в противен случай, e не би било сигурно. Алгоритъмът спира, когато множеството от върховете-краища на ребрата от A стане V . Веднага следва, че **while**-цикълът се изпълнява точно $n - 1$ пъти.

Цялата трудност тогава е, как да намираме сигурно ребро по отношение на множество от ребра. Теорема 36 дава достатъчна обосновка за намирането на сигурни ребра от алгоритмите, които ще разгледаме. Първо ни трябват няколко определения.

Определение 56: Срез в граф.

Срез в граф G е всяко разбиване на $V(G)$ на два дяла. Нека $S = \{V_1, V_2\}$ е срез в G . Ребро $e = (u, v)$ прекосява S , ако $u \in V_1$ и $v \in V_2$. Нека $E' \subseteq E(G)$. S е съобразен с E' , ако нито едно ребро от E' не го прекосява. Ако G е тегловен и реброто e прекосява S , казваме, че реброто e е *леко*, ако e има минимално тегло между всички ребра, които прекосяват S .

Теорема 36: Theorem 23.1 от [15, стр. 627]

Нека $G = (V, E)$ е свързан тегловен граф с тегловна функция $w : E \rightarrow \mathbb{R}^+$. Нека $A \subseteq E$ е такова, че съществува МПД T , че $A \subseteq E(T)$. Нека $S = \{V_1, V_2\}$ е срез в G , който е съобразен с A . Нека $e = (u, v)$ е произволно леко ребро, прекосяващо S . Тогава e е сигурно за A .

Доказателство: Ако $e \in E(T)$ ние сме готови с доказателството. Да допуснем, че $e \notin E(T)$. Ще покажем, че има МПД T' , такова че $A \cup \{e\} \subseteq E(T')$.

Добавяме e към T и получаваме уницикличен граф U . Нещо повече, със сигурност e е ребро от цикъла с на U . За реброто e знаем, че прекосява среза. Със сигурност съществува ребро e' в S , различно от e , което прекосява среза. Изтриваме e' от U и получаваме ПД T' , различно от T . Да сравним $w(T)$ с $w(T')$. T' се получи от T с добавяне на e и изтриване на e' . Тогава

$$w(T') = w(T) + w(e) - w(e')$$

Но e е леко по конструкция, така че $w(e) \leq w(e')$, следователно $w(e) - w(e') \leq 0$. Тогава $w(T') \leq w(T)$. Но T е МПД, така че $w(T) \leq w(T')$ по дефиниция. Тогава $w(T) = w(T')$ и T' също е МПД. Което съдържа както ребрата от A , така и e . \square

11.2 Алчни алгоритми

Предложената обща идея за намиране на МПД на тази страница заедно с Теорема 36 се конкретизира до два алгоритъма, които ще разгледаме в следващите две секции. И двата алгоритъма са изградени по *алчна схема* (на английски, *greedy strategy*). Алгоритмите, изградени по тази схема, се наричат *алчни алгоритми* (*greedy algorithms*).

По правило, изчислителните задачи, които алчните алгоритми решават, са *оптимизационни задачи*. Не всички изчислителни задачи са оптимизационни; както вече видяхме на стр. 4, има задачи за разпознаване (това са задачите с отговор Да/Не). В оптимизационните задачи, всеки пример се асоциира с множество от *допустими решения* (на английски, *feasible solutions*), което може да е и празно. Дефинирана е *целева функция* (на английски, *objective function*), която изобразява всяко допустимо решение в число. Дефинирана е и *цел*, която е или минимизация, или максимизация. Множеството от *оптималните решения* е подмножеството на допустимите решения, върху чиито елементи целевата функция има минимална стойност, ако целта е минимизация, или максимална стойност, ако целта е максимизация.

Като пример, задачата за намиране на МПД е оптимизационна. Всеки пример е наредена двойка от свързан граф и тегловна функция. Множеството от допустимите решения е множество от покриващите дървета на графа; то е задължително непразно, но това е особеност на тази задача, в други задачи може да е празно. Целевата функция изобразява допустимо решение (ПД) в число, което е сумата от теглата на ребрата му. Целта е минимизация. Оптималните решения са минималните покриващи дървета.

Друг пример е задачата за намиране на най-лек Хамилтонов цикъл в неориентиран тегловен граф. Да кажем, че теглото на цикъл е сумата от теглата на ребрата в него. Сега множеството от допустимите решения е множеството от всички Хамилтонови цикли и то може да е празно, защото не всеки граф е Хамилтонов. Целевата функция изобразява цикъл в сумата от теглата на ребрата му. Множеството от оптималните решения се състои от Хамилтоновите цикли с минимална сума от теглата на ребрата.

Оптимизационните задачи МПД и НАЙ-ЛЕК ХАМИЛТОНОВ ЦИКЪЛ се различават драстично по сложността на алгоритмите, които ги решават. Алгоритмите за МПД имат квадратична сложност (или сложност, не много по-голяма от квадратична, както ще видим в следващите секции), докато НАЙ-ЛЕК ХАМИЛТОНОВ ЦИКЪЛ е NP-трудна, което почти сигурно означава, че за нея няма ефикасен алгоритъм. Със сигурност до момента такъв не е известен, като евентуалното откриването на ефикасен алгоритъм за нея би бил събитие, за което ще се говори в новините.

Задачата МПД се решава ефикасно с алгоритми, които работят итеративно, на всяка итерация добавяйки точно едно ребро към множеството от вече добавените ребра. Критерият, по който се избира ребро за добавяне, е много прост и ясен: добавя се леко ребро, прекосяващо някакъв срез (в алгоритъма на Kruskal: стига то да не образува цикъл с досега сложените ребра). Всеки такъв избор на леко ребро е *локален*, защото не се прави с оглед на “голямата картина”. Теорема 36 гарантира, че **поредицата от локално оптимални избори** (на леки ребра) води до **глобално оптимално решение** (МПД).

Би трябвало да е ясно защо такива алгоритми се наричат алчни. Терминът “алчни” не е съвсем прецизен, понеже всеки алгоритъм, който решава оптимизационна задача, е в някакъв смисъл алчен. При алчните алгоритми става дума за някаква късогледа алчност, или локална алчност. Когато решаваме МПД, локалната алчност е печеливша стратегия. Има обаче задачи, в които проявата на локална алчност води до решение, което не е оптимално. Всички NP-трудни задачи са такива, но примерът с НАЙ-ЛЕК ХАМИЛТОНОВ ЦИКЪЛ не е особено удачен за тази илюстрация, поради което ще илюстрираме с друга задача: KNAPSACK, или задачата за раницата (Секция 13.9). Пример за това има на страници 425–427 в [15].

Подробно теоретично изложение на фундамента на алчните алгоритми има в Секция 16.2 на [15, стр. 423] и Секция 16.4 на [15, стр. 437], както и [52, стр. 280]. Това е материал, който е далече отвъд обхватата на тези лекции.

Говорейки полуформално за алчни алгоритми и алчни схеми, авторът на тези записки е на мнение, че има неизбежен субективизъм/условност в това, което наричаме “локално оптимален избор” (което дефинира алчните схеми). От някаква гледна точка, всеки избор (на

нов елемент, който да добавим в множеството) е локален. Ако искаме да прецизирате “локално оптимален избор”, трябва да говорим за максималните ресурси (време/памет), които сме склонни да отделим за него.

11.3 Алгоритъм на Prim

Алгоритъмът на Prim за намиране на МПД далечно прилича на обхождане. Той започва от един стартов връх s , който е част от входа, намира най-лекото ребро, свързващо s с друг връх, слага го в дървото, после намира най-леко ребро, свързващо някой от тези два върха с друг връх, слага го в дървото, и така нататък. В терминологията на ОбЩ вид на МПД на стр. 313, в алгоритъма на Prim, ребрата от A във всеки момент образуват едно (свързано) дърво. Естествено, то става ПД чак накрая, когато включи всички върхове, но във всеки момент от изпълнението то е едно дърво. Ето много общ псевдокод на алгоритъма на Prim.

PRIM GENERIC($G = (V, E)$): неориентиран свързан граф, w : тегловна функция върху G , s : стартов връх)

```

1    $V(T) \leftarrow \{s\}$ 
2    $A \leftarrow \emptyset$ 
3   while  $V \setminus V(T) \neq \emptyset$  do
4       намери леко ребро  $e = (x, y)$ , прекосяващо среза  $\{V(T), V \setminus V(T)\}$ 
5       нека  $x \in V(T)$ 
6        $V(T) \leftarrow V(T) \cup \{y\}$ 
7        $A \leftarrow A \cup \{e\}$ 
8   return  $A$ 
```

Коректността е очевидна и следва веднага от Теорема 36. От общи съображения обаче сложността не е добра. Очевидно **while**-цикълът се изпълнява $n - 1$ пъти. Потенциално проблематично е намирането на леко ребро (ред 4) на всяка итерация. Ако го правим с пълно изчерпване, тоест, минаваме през всички ребра на G и търсим ребро, прекосяващо среза с минимално тегло, само ред 4 би отнемал $\Theta(m)$ време на всяка итерация, поради което сложността на алгоритъма би била $\Theta(nm)$. Това е неприемливо бавно.

Известно подобрение би било на всяка итерация да не преглеждаме всички ребра на графа, а само тези, които прекосяват среза $\{V(T), V \setminus V(T)\}$. В асимптотичния смисъл обаче това не е никакво подобрение. В най-лошия случай, общият брой на прегледани ребра (за цялото изпълнение на алгоритъма) се оценява със сумата

$$\sum_{k=1}^{n-1} k(n-k) = \Theta(n^3)$$

тъй като има ребро между всеки връх от единия дял и всеки връх от другия дял, на всяка итерация (какъв график трябва да е G , за да изпълнено това?). И така, отново имаме, в най-лошия случай, кубичен алгоритъм. Трябва да направим нещо по-умно, за да избираме ефикасно леко прекосяващо ребро.

В следващите две подсекции ще разгледаме два варианта на алгоритъма на Prim. Единият не използва изтънчени структури данни и ще го наречем “базов”, а другият ползва АТД приоритетна опашка и ще го наречем “изтънчен”. Върху разредени графи изтънченият вариант е по-добър, но върху графи с брой ребра, близък до максималния, базовият алгоритъм, колкото и да е странно, е по-бърз.

11.3.1 Базов вариант на алгоритъма на Prim

Този алгоритъм следва плътно PRIM GENERIC, като намирането на леко ребро на ред 4 става във време $O(n)$, като в най-лошия случай това е $\Theta(n)$. Идеята е елементарна. За да я изложим, ще въведем класификация на върховете. Нека във всеки момент от работата на алгоритъма,

- върховете от $V(T)$ са *дървесните върхове* (на английски, *tree vertices*),
- върховете от $N(V(T))$ са *граничните върхове* (на английски, *fringe vertices*),
- върховете от $V \setminus N[V(T)]$ са *неизвестните върхове* (на английски, *unseen vertices*).

Тази класификация на практика дублира класификацията от обхождането на графи на върховете на бели, сиви и черни на стр. 265, като черните върхове сега наричаме дървесни, сивите наричаме гранични, а белите са неизвестните. Причината да въвеждаме с други имена практически същата класификация е широко приетата терминология: класификацията на бели, сиви и черни е от [15], а на tree, fringe и unseen е например от [63, стр. 485].

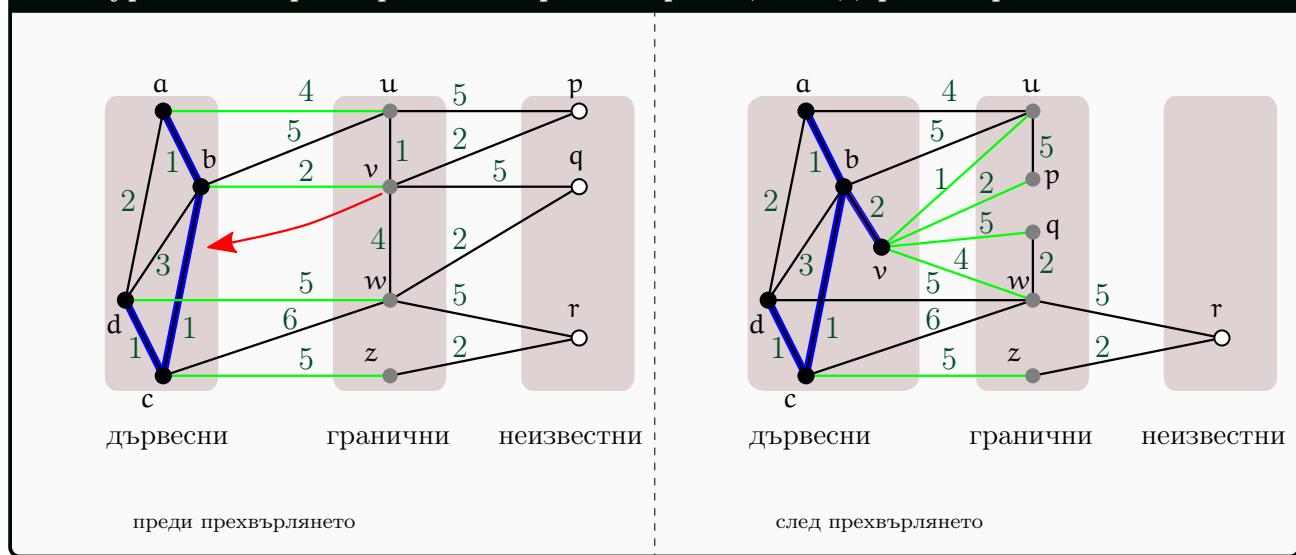
И така, сега класифицираме върховете на дървесни, гранични и неизвестни. Множеството от граничните върхове ще наричаме *границата*. На всяка итерация точно един граничен връх става дървесен. Нещо повече. Това е такъв граничен връх, който е инцидентен с леко прекосяващо ребро. В термините на PRIM GENERIC, това е връх u . За да направим избора на ред 4, достатъчно е да прегледаме всички гранични върхове; ако за всеки от тях предварително е известно теглото на най-леко инцидентно прекосяващо ребро и самото ребро, може да направим избора във време, пропорционално на размера на границата, което, в най-лошия случай е $\Theta(n)$. Иначе казано, наместо да търсим директно леко ребро, както пише на ред 4, ще търсим граничен връх, инцидентен с такова, а, когато намерим върха, в $\Theta(1)$ ще намерим и реброто, ако предварително всеки граничен връх е бил асоцииран с най-леко инцидентно прекосяващо ребро. Ако може да работим във време $\Theta(1)$ за всеки граничен връх, ще може да сведем сложността на алгоритъма до $\Theta(n^2)$.

Сега се фокусираме върху това, как да работим във време $\Theta(1)$ за всеки граничен връх. Въвеждаме понятието *кандидат-ребро*, или просто *кандидат*. По време на работата на алгоритъма, всеки граничен връх е асоцииран с точно едно кандидат-ребро и с една стойност, наречена *гранично тегло*, която е теглото на кандидат-реброто. За дървесните и неизвестните върхове кандидат-ребра и гранични тегла не се дефинират.

Всеки граничен връх може да е инцидентен с дървесни, гранични и неизвестни върхове. След избора на ред 4 прехвърляме връх от границата в дървото, като обаче това може да наложи промени в някои кандидат-ребра и гранични тегла, така че на следващата итерация, стойностите на всички кандидат-ребра и гранични тегла да са коректни и отново да можем да направим избора, извършвайки $\Theta(1)$ работа за всеки граничен връх. Това е показано на Фигура 11.2. Дървесните върхове са вляво, граничните са по средата, а неизвестните са вдясно. Ребрата в дървото са в синьо. Кандидат-ребрата са в ярко зелено. В лявата подфигура е показано състоянието на нещата в началото на някоя итерация. Граничните върхове са u , v , w и z със съответни гранични тегла 4, 2, 5 и 5. Върхът с минимално гранично тегло е v и той бива прехвърлен към дървесните върхове, както показва червената стрелка. След прехвърлянето (дясната подфигура), реброто (b, v) е част от дървото. Върховете, които са неизвестни съседи на v преди прехвърлянето (p и q) сега вече са гранични, като кандидат-ребрата им са съответно (p, v) и (q, v) . Преди прехвърлянето v е съсед на граничните върхове u и w . След прехвърлянето на v , те си остават гранични, но кандидат-ребрата им се променят на

съответно (u, v) и (w, v) , понеже и двете имат по-малко тегло от съответните им кандидат-ребра (a, u) и (d, w) преди прехвърлянето. И така, дясната подфигура показва състоянието на нещата в началото на следващата итерация.

Фигура 11.2 : Прехвърляне на връх от границата в дървото при Prim.



Ето псевдокод на базовия вариант на алгоритъма на Prim. Ползва се масив $\text{status}[1, \dots, n]$, където $\text{status}[i] \in \{\text{дървесен}, \text{граничен}, \text{неизвестен}\}$. Този масив е аналог на масива color от обхожданията.

МПД PRIM1($G = (V, E)$): неориентиран свързан граф, w : тегловна функция върху G , s : стартиращ връх

```

1   $V(T) \leftarrow \{s\}$ 
2   $A \leftarrow \emptyset$ 
3  foreach  $u \in V$ 
4       $\text{status}[u] \leftarrow \text{неизвестен}$ 
5       $\text{status}[s] \leftarrow \text{дървесен}$ 
6  foreach  $y \in \text{adj}[s]$ 
7       $\text{status}[y] \leftarrow \text{граничен}$ 
8   $x \leftarrow s$ 
9  while  $V \setminus V(T) \neq \emptyset$  do
10     (*  $x$  е последният сложен в дървото връх *)
11     foreach  $y \in \text{adj}[x]$ 
12         if  $\text{status}[y] = \text{граничен}$ 
13             ако  $w((x, y))$  е по-малко от граничното тегло на  $y$ ,
14                 смени кандидат-реброто на  $y$  с  $(x, y)$ 
15         if  $\text{status}[y] = \text{неизвестен}$ 
16              $\text{status}[y] \leftarrow \text{граничен}$ 
17             кандидат-реброто на  $y$  става  $(x, y)$ 
18         намери кандидат-ребро е с минимално тегло
19          $x \leftarrow \text{граничния връх на } e$ 
20          $\text{status}[x] \leftarrow \text{дървесен}$ 
21          $V(T) \leftarrow V(T) \cup \{x\}$ 
22          $A \leftarrow A \cup \{e\}$ 
23  return  $A$ 
```

Коректност и сложност. Подробно доказателство за коректност няма да правим. Инвариантата е, при всяко достигане на управлението на ред 9, ребрата от A задават МПД на подграфа на G , индуциран от $V(T)$. Използвайки Теорема 36, лесно показваме, че реброто е на ред 17 е сигурно за A .

Сложността по време е $\Theta(n^2)$. Намирането на най-леко кандидат-ребро на ред 17 се извършва в $\Theta(n)$ в най-лошия случай. Имайки предвид, че **while**-цикълът се “извърта” $\Theta(n)$ пъти, това дава директно $\Theta(n^2)$. Знаем, че обхождането на списъците на редове 11–16 става във време $\Theta(n+m)$ за цялата работа на алгоритъма. Тъй като $\Theta(m+n) = O(n^2)$, имаме обща асимптотична оценка за сложността $\Theta(n^2)$. Забележе, че по отношение на най-лошия случай, тази сложност по време е оптимална, защото в най-лошия случай описание на графа е с размер $\Theta(n^2)$, а за всяко n има примери за граф с n върха и тегловна функция, такива че се налага да разгледаме цалото описание на графа, за да конструираме МПД.

11.3.2 Изтънчен вариант на алгоритъма на Prim

Този вариант на алгоритъма на Prim прави бърз избор на минимално ребро, което да бъде добавено към временното дърво. Използва се АТД приоритетна опашка Q от min тип от върховете. Ключът на всеки връх в Q е минималното тегло на реброто, свързващо този връх с връх от временното дърво. Ако такова ребро няма, ключът е ∞ . В Q поначало се слагат всички върхове на графа с ключове ∞ (защото в този псевдокод поначало дърво няма), а след това на всяка итерация Q съдържа точно върховете, които **не са в дървото**; тоест, Q съдържа точно границите и неизвестните върхове. Ключовете пишем така: ако върхът е u , неговият ключ е $u.key$.

Самото МПД се реализира с масив на предшествията. За разлика от [15], този масив ще записваме като $\pi[1, \dots, n]$.

МПД PRIM2($G = (V, E)$): неориентиран свързан граф, w : тегловна функция върху G , s : стартов връх)

```

1   foreach  $u \in V$ 
2      $u.key \leftarrow \infty$ 
3      $\pi[u] \leftarrow \text{Nil}$ 
4      $s.key \leftarrow 0$ 
5     създай празна приоритетна min опашка  $Q$ 
6      $Q \leftarrow V$ 
7     while not isempty( $Q$ ) do
8        $x \leftarrow \text{EXTRACT-MIN}(Q)$ 
9       foreach  $y \in \text{adj}[x]$ 
10      if  $y \in Q$  and  $w((x, y)) < y.key$ 
11         $y.key \leftarrow w((x, y))$ 
12         $\pi[y] \leftarrow x$ 
13    return  $\pi[1, \dots, n]$ 
```

Този псевдокод следва плътно псевдокода от [15, стр. 634]. Две забележки по имплементацията. Първо, проверката $y \in Q$ на ред 10 може да бъде имплементирана, като всеки връх има булев флаг за принадлежност към опашката; в началото всички флагове са TRUE, а

при всяко вадене на връх от опашката на ред 8, правим флага му FALSE. Второ, смяната на ключа на u на ред 11 не е просто смяна на едно число с друго, както може човек да си помисли първо. Връх u е в опашката и смяната на ключа му, тоест, намаляването на ключа му, трябва да бъде съпроводено с изпълнение на DECREASE-KEY, която е огледален аналог на INCREASE-KEY от Подсекция 4.4.4. Това има значение за анализа на сложността.

Коректност.

Теорема 37: Коректност на МПД PRIM2

Масивът $\pi[1, \dots, n]$, който МПД PRIM2 връща, реализира МПД на G .

Доказателство. Първо едно помощно твърдение.

Инвариант 18: while-цикълът на МПД PRIM2

При всяко достигане на ред 7:

- ① $Q \neq \emptyset \rightarrow (\exists u \in Q : u.key < \infty)$.
- ② $\forall u \in Q \setminus \{s\} : \text{ако } u.key < \infty, \text{ то:}$
 - ❶ $\pi[u] \neq \text{Nil}$
 - ❷ $\pi[u] \notin Q$
 - ❸ $u.key$ е теглото на най-леко ребро, инцидентно с u и прекосяващо среза $\{V \setminus Q, Q\}$.
- ③ $\forall u \in Q : \text{ако } u.key = \infty, \text{ то: съществува } v \in Q, \text{ такъв че } v.key < \infty \text{ и има път между } u \text{ и } v, \text{ чиито върхове са само от } Q$.

База. Разглеждаме първото достигане на ред 7. Тогава $Q = V$ (ред 6), така че Q не е празна и антецедентът на ① е истина. Но консеквентът също е истина, понеже има връх във V (и оттам, в Q) със стойност key, по-малка от ∞ : това е връх s (ред 4). Тогава ① е в сила.

Да разгледдаме ②. Тъй като единственият връх в Q със стойност key, която не е ∞ , е s , антецедентът на импликацията е лъжа за всяко $u \in Q \setminus \{s\}$ и оттам, импликацията е истина за всяко $u \in Q \setminus \{s\}$.

Да разгледдаме ③. Истинността следва от това, че $V = Q$, $s.key < \infty$ и G е свързан. ✓

Поддръжка. Да допуснем, че твърдението е вярно за някое достигане на ред 7, което не е последното. Нека моментът на това достигане е t , а следващото достижане е в момента t' .

Ще докажем ①. Ако след изваждането на връх от Q на ред 8, Q стане празна, антецедентът на ① става лъжа, а цялата импликация, истина. Нека след ред 8 Q не е празна. Ако в момента t съществува $z \in Q, z \neq x$, такъв че $z.key < \infty$, доказателството е готово. Да допуснем, че в момента t , за всеки $z \in Q, z \neq x$, е вярно, че $z.key = \infty$. Но част ③ от индуктивното предположение влече, че x поне един съсед z' в Q . Очевидно на ред 11 $z'.key$ ще стане $w((x, z'))$, поради което в момент t' отново е вярно, че Q съдържа връх с key стойност, по-малка от ∞ .

Ще докажем ②. Нека в момента t , $Q_f = \{v \in Q | v.key < \infty\}$ и $Q_\infty = \{v \in Q | v.key = \infty\}$. Ще разгледдаме три множества: $Q_f \setminus N(x)$, $Q_f \cap N(x)$ и $Q_\infty \cap N(x)$.

Върховете от $Q_f \setminus N(x)$ не биват засегнати от изваждането на x от Q и за тях **1**, **2** и **3** остават в сила в момента t' .

Разглеждаме произволен $v \in Q_f \cup N(x)$. Ако в момента t , $w((x, v)) \geq v.key$, то булевото условие на ред [10](#) е лъжа, когато y стане v , и редове [11–12](#) не се изпълняват. Тогава в момента t' , **1**, **2** и **3** остават в сила за v от индукционното предположение. Ако в момента t , $w((x, v)) < v.key$, то булевото условие на ред [10](#) е истина, когато y стане v (ред [9](#)), и редове [11–12](#) се изпълняват. Тогава $\pi[v]$ става x (ред [12](#)) и **1** и **2** са в сила за v . На ред [11](#), $v.key$ става $w((x, v))$. Очевидно сега (x, v) е най-лекото, инцидентно с v , прекосяващо ребро, така че и **3** е в сила.

Разглеждаме произволен $v \in Q_\infty \cup N(x)$. Когато y стане v (ред [9](#)), булевото условие на ред [10](#) задължително е истина, понеже $v \in Q$ и $v.key = \infty$. После на ред [12](#), $\pi[v]$ става x , така че **1** и **2** са в сила за v . На ред [11](#), $v.key$ става $w((x, v))$. Очевидно сега (x, v) е най-лекото (и единствено), инцидентно с v , прекосяващо ребро, така че и **3** е в сила.

Ще докажем **3**. След изваждането на x от Q , за всеки $v \in Q_\infty \cap N(x)$, $v.key$ става $w((x, v))$ на ред [11](#) и v престава да е елемент на Q_∞ . От друга страна, за всеки $v \in Q_\infty \setminus N(x)$, твърдението остава в сила заради свързаността на графа. \square

Нека във всеки момент от работата на алгоритъма, $G_{\bar{Q}}$ е подграфът на G , индуциран от върховете, които не са в Q . Твърди се, че съществува МПД T на G , спрямо което Инвариантът [19](#) е в сила.

Инвариантът 19: while-цикълът на МПД PRIM2

При всяко достигане на ред [7](#), $\{(u, \pi[u]) \mid u \in V \setminus (\{s\} \cup Q)\} = E(T) \cap E(G_{\bar{Q}})$.

База. Разглеждаме първото достигане на ред [7](#). В този момент $\{s\} \cup Q = V$, така че $V \setminus (\{s\} \cup Q) = \emptyset$ и множеството в лявата страна е празно. Но множеството в дясната страна също е празно, понеже всички върхове са в Q и $E(G_{\bar{Q}})$ е празно. Със сигурност такова МПД T съществува.

Поддръжка. Да допуснем, че инвариантата е в сила в даден момент t , в който изпълнението е на ред [7](#) и **while** ще бъде изпълнен поне още веднъж, което влече, че Q не е празна. Щом Q не е празна, съгласно Инвариантът [18](#) е вярно, че $\exists u \in Q : u.key < \infty$.

Твърди се, че в момента t за всеки връх $v \in Q$, такъв че $v.key$ е минимален, е вярно, че реброто $(v, \pi[v])$ е най-леко ребро, прекосяващо среза $\{V \setminus Q, Q\}$. Но това следва директно от Инвариантът [18](#). Съгласно Теорема [36](#), в момента t за всеки връх $v \in Q$, такъв че $v.key$ е минимален, е вярно, че реброто $(v, \pi[v])$ е сигурно за множеството $E(T) \cap E(G_{\bar{Q}})$, защото това множество е съобразено със среза $\{V \setminus Q, Q\}$.

На ред [8](#) върхът, който бива изведен от опашката и съхранен в променливата x е връх, който в момента t е в Q и е с минимална key стойност – при допускането, че Extract-Min работи коректно. Заключаваме, че след изваждането на този връх от Q , множеството $\{(u, \pi[u]) \mid u \in V \setminus (\{s\} \cup Q)\}$ продължава да е равно на $E(T) \cap E(G_{\bar{Q}})$.

Терминация. При последното достигане на ред [7](#), опашката Q е празна. Тогава $E(G_{\bar{Q}}) = E(G)$, така че $E(T) \cap E(G_{\bar{Q}}) = E(T)$; освен това, $V \setminus (\{s\} \cup Q) = V \setminus \{s\}$; заключаваме, че $\{(u, \pi[u]) \mid u \in V \setminus \{s\}\} = E(T)$. С други думи, МПД PRIM2 е построил МПД чрез $\pi[1, \dots, n]$. \square

Сложност по време. Допускаме, че Q е реализирана чрез двоична пирамида. Ще приложим подхода към анализа на сложността, който ползвахме при BFS/DFS, и ще игнорираме

факта, че **while**-цикъла (редове 7–12) се изпълнява $\Theta(n)$ пъти. В израза за сложността няма множител n . Наместо да броим колко пъти се “извърта” **while**-а, ще разсъждаваме колко пъти се изпълнява **for**-цикъла на редове 9–12 за **цялото изпълнение на алгоритъма**. Както знаем, ред 9 се изпълнява $\Theta(n + m)$ пъти, защото това е просто минаване през списъците на съседство. Само че тялото на този **for** се изпълнява в най-лошия случай във време $\Theta(\lg n)$ заради имплицитното викане на DESCREASE-KEY на ред 11. Оттук имаме оценка за сложността $\Theta((n + m) \lg n)$, което при свързан граф е $\Theta(m \lg n)$. Колкото и да е странно, ако ребрата на графа са много в смисъл, че $m \asymp n^2$, изтънчения вариант на алгоритъма на Prim с приоритетна опашка е по-лош от базовия вариант, в който намираме следващото ребро с просто последователно търсене. За разредени графи обаче предимството на изтънчения вариант е сериозно: $n \lg n$ срещу n^2 .

Забележете, че в това извеждане игнорираме факта, че EXTRACT-MIN на ред 8 работи във време $\Theta(\lg n)$. Ако сумираме цялата работа на EXTRACT-MIN по време на изпълнението на алгоритъма, ще получим $\Theta(n \lg n)$. Но $\Theta(n \lg n) = O(m \lg n)$ за свързан граф, така че изразът за сложността може да бъде записан най-кратко като $\Theta(m \lg n)$.

Ако Q бъде реализирана не чрез двоична пирамида, а чрез пирамида на Fibonacci (Fibonacci heap) може сложността да се подобри, поне в асимптотичния смисъл. При пирамидите на Fibonacci, функцията DESCREASE-KEY работи в амортизирано време $\Theta(1)$. Функцията EXTRACT-MIN остава със сложност $\Theta(\lg n)$ в най-лошия случай, поради което асимптотичната сложност става $\Theta(m + n \lg n)$. Събирамето $n \lg n$ отчита работата на всички EXTRACT-MIN по време на изпълнението на алгоритъма. Piрамида на Fibonacci обаче е много сложна структура, ползването на която води до големи скрити мултипликативни константи (скрити в Θ -нотацията). Трябва данните наистина да са големи, за да има осезаема практическа полза от ползването на пирамиди на Fibonacci вместо двоични пирамиди в алгоритъма на Prim.

Пирамидите на Fibonacci имат и друго предимство пред двоичните пирамиди: те може да бъдат сливани ефикасно, в $\Theta(1)$ амортизирано време, докато двоичните пирамиди искат линейно време за сливане. Piрамидите на Fibonacci са изложени подробно в [15, стр. 505].

11.4 Алгоритъм на Kruskal

Алгоритъмът на Kruskal е ефикасен алгоритъм за намиране на МПД, който е основан на съвсем различна идея от тази на алгоритъма на Prim. Най-общо казано, първо сортираме ребрата по тегло, и после **в ненамаляваща ред** на теглата слагаме първите $n - 1$ ребра, за всяко от които, в този ред, е вярно, че не образува цикъл с вече сложените ребра. Ако обаче имплементираме тази идея буквално, ще получим кубичен алгоритъм.

- Сортирането на ребрата става във, и иска, време $\Theta(m \lg m)$, което е същото като $\Theta(n^2 \lg n)$, ако $m \asymp n^2$.
- В най-лошия случай се налага да проверим всички m ребра. За да се убедим в това, нека най-тежкото ребро е мост. Всеки мост задължително участва във всяко МПД, защото всеки мост участва във всяко ПД. Еrgo, в най-лошия случай може да се наложи да стигнем до края на сортираната редица от ребра.

За всяко ребро в сортирана редица тестваме дали образува цикъл с вече сложените ребра. Тоест, за всяко ребро тестваме дали подграфът, индуциран от вече сложените ребра плюс него е цикличен. Ако тестът се направи с BFS или DFS, той става във време

$\Theta(n)$ в най-лошия случай[†]. Така че тези тестове ще станат във време $O(nm)$, като може да се покаже, че границата е точна, тоест, $\Theta(nm)$.

При $m = n^2$, това е $\Theta(n^2 \lg n) + \Theta(n^3) = \Theta(n^3)$.

Как да подобрим сложността по време? Сортирането на ребрата отнема $\Omega(m \lg n)$ време и това е неизбежно заради долната граница на сортирането (Подсекция 7.2.2)[‡]. Еrgo, имаме добра граница $\Omega(m \lg n)$ за алгоритъма на Kruskal, независимо от това колко ефикасно имплементираме слагането на ребрата.

Слагането на ребрата става итеративно, като в най-лошия случай се изпълняват m итерации (ако най-тежкото ребро е мост). Имайки предвид това, целта ни е тестването за едно ребро да става във време $O(\lg n)$. Ако успеем да постигнем това, ще имаме $\Theta(m \lg n)$ имплементация на алгоритъма на Kruskal; дори да постигнем тестване за едно ребро във време $O(\lg n)$, пак ще имаме $\Theta(m \lg n)$ имплементация на алгоритъма на Kruskal заради сортирането в началото.

Ключовото наблюдение е, че ако имаме гора с повече от едно дърво и добавим ново ребро между два върха:

- ако те са от различни дървета на гората, няма да се образува цикъл, но тези две дървета плюс новото ребро ще станат едно дърво, а графът ще продължи да е гора;
- ако те са от едно и също дърво, ще се образува цикъл, а графът ще престане да е гора.

В светлината на това, за ефикасното тестване дали всяко ново ребро в наредбата образува цикъл е достатъчно да поддържаме разбиване на върховете на графа, като дяловете на разбиването отговарят на дърветата в гората. С други думи, за всяко дърво от гората ни трябва да знаем само кои са върховете му. Ребрата му не ни интересуват. Наистина, алгоритъмът строи множество от ребра и връща множество от ребра (чиято съвкупност е МПД), но по време на итерациите няма смисъл да поддържаме експлицитно отделните дървета в гората като върхове и ребра – интересуват ни само върховете им. Тестването дали ребро образува цикъл се свежда до това дали двата му края са в различни дялове на разбиването: ако не, това ребро се прескача, ако да, то се добавя, като обаче тези дялове трябва да станат един и същи дял.

Забележете промяната в терминологията, която направихме. Поначало говорехме за тестване дали двата края на новото ребро са в различни дървета, всяко от които си има върхове и ребра, и за сливане на тези дървета плюс новото ребро. Сега говорим за тестване дали двата края на новото ребро са в различни множества (само от върхове) и сливане на тези множества; “плюс новото ребро” сега няма.

Ето псевдокод на алгоритъма на Kruskal съгласно [63]. Алгоритъмът поддържа разбиване на множеството от върховете, като в началото създава разбиването на едноелементни множества; функцията `component(u)` връща идентификатора на дяла на разбиването, в който е връх u , а функцията `identify(component(u), component(v))` слива дяловете, в които се назират u и v , при условие, че тези дялове са различни.

МПД $KRUSKAL(G = (\{1, \dots, n\}, E))$: неориентиран свързан граф, w : тегловна функция върху G)

- 1 $A \leftarrow \emptyset$
- 2 направи разбиването $\{\{1\}, \{2\}, \dots, \{n\}\}$

[†]А не в $\Theta(n + m)$ – защо?

[‡]За теглата на ребрата няма ограничения, така че не може да ползваме COUNTING SORT.

```

3  сортирай  $E$  по тегла
4  count  $\leftarrow 0$ 
5  while count  $< n - 1$  do
6      нека  $e = (u, v)$  е следващото ребро в сортираната редица
7      if component( $u$ )  $\neq$  component( $v$ )
8           $A \leftarrow A \cup \{e\}$ 
9          identify(component( $u$ ), component( $v$ ))
10         count ++
11 return  $A$ 

```

Коректност. Коректността може да се докаже със следната инварианта: съществува МПД T , такова че при всяко достигане на ред 5 е вярно, че $A \subseteq E(T)$. Реброто e , което избираме на ред 6, има краища u и v . Ако компонентите, в които се намират u и v , да ти наречем T_u и T_v съответно, са различни, булевото условие на ред 7 е истина и изпълнението отива на ред 8. Но щом T_u и T_v са различни, то e прекосява среза $\{V(T_u), V(G) \setminus V(T_u)\}$. Тогава, съгласно Теорема 36, реброто e е сигурно за A .

Сложност по време. За да бъде ефикасен МПД KRUSKAL, трябва проверката дали $\text{component}(u) \neq \text{component}(v)$ (ред 7) и операцията $\text{identify}(\text{component}(u), \text{component}(v))$ (ред 9) да се вършат във време $O(\lg n)$. Ако успеем да постигнем това, **while**-цикълът (редове 5–10) ще се изпълнява във време $O(m \lg n)$ и алгоритъмът ще има сложност по време $\Theta(m \lg n)$ заради сортирането (ред 3).

Всяка компонента, или дял, на разбиването трябва да има идентификатор; тоест, свое име. Нека са дефинирани две примитивни функции: Find и Union. Ако u е елемент от опорното множество (в случая, връх на графа), $\text{Find}(u)$ връща името на дяла на разбиването, в който е u . Ако i и j са имената на два различни дяла, $\text{Union}(i, j)$ слива тези два дяла в един и му дава име i . Ако разполагаме с такива примитиви, можем да реализираме $\text{component}(u) \neq \text{component}(v)$ така:

$\text{component}(u) \neq \text{component}(v)$

```

1  i  $\leftarrow \text{Find}(u)$ 
2  j  $\leftarrow \text{Find}(v)$ 
3  if i  $\neq$  j
4      return TRUE
5  else
6      return FALSE

```

и да реализираме $\text{identify}(\text{component}(u), \text{component}(v))$ така:

$\text{identify}(\text{component}(u), \text{component}(v))$

```

1  i  $\leftarrow \text{Find}(u)$ 
2  j  $\leftarrow \text{Find}(v)$ 
3   $\text{Union}(i, j)$ 

```

В следващата секция ще видим как се реализират ефикасно примитивите Find и Union.

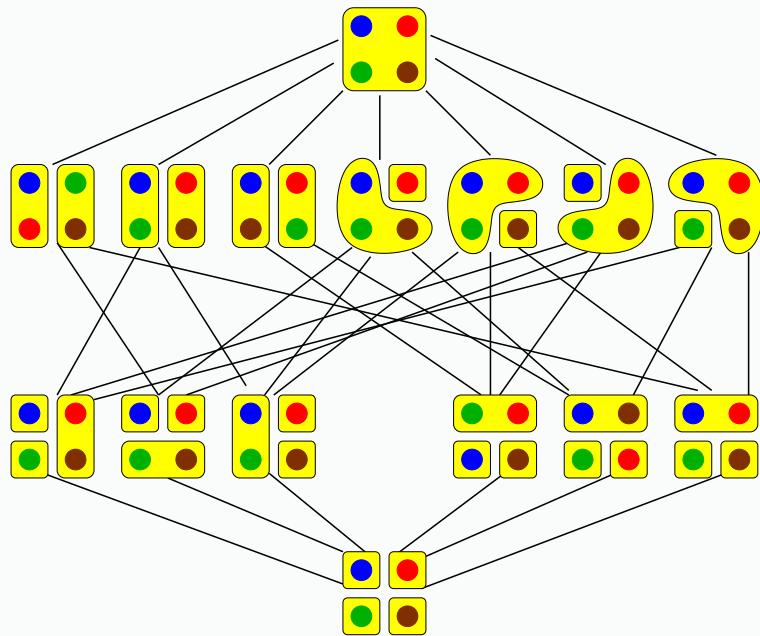
Допълнение 32: Частична наредба \sqsubseteq върху разбиванията

Нека S е множество. Множеството от всички разбивания на S ще бележим с $\Pi(S)$. За всеки $P_1, P_2 \in \Pi(S)$, P_1 рафинира P_2 , което означаваме с $P_1 \sqsubseteq P_2$, ако

$$\forall X \in P_1 \exists Y \in P_2 : X \subseteq Y$$

Конверсно, P_2 абстрактира P_1 . Примерно, ако $S = \{a, b, c, d\}$, то е вярно, че $\{\{a, b\}, \{c\}, \{d\}\} \sqsubseteq \{\{a, b\}, \{c, d\}\}$ Очевидно релацията \sqsubseteq е частична наредба.

Наредената двойка (S, \sqsubseteq) е вид решетка – понятие, което няма да дефинираме тук (вижте [11] и [53] за много подробно и изчерпателно въведение). За целите ни е достатъчно да се каже, че всяка решетка има уникален минимален елемент и уникален максимален елемент. Ето как изглежда диаграмата на Hasse на $(\{\{a, b, c, d\}, \sqsubseteq\})$ (без имената на елементите):



Минималният елемент е $\{\{a\}, \{b\}, \{c\}, \{d\}\}$: разбиването на едноелементни множества (*partition into singletons*). Максималният елемент е тривиалното разбиване $\{\{a, b, c, d\}\}$.

Връзката на всичко това с алгоритъма на Kruskal е следната. В никакъв смисъл, алгоритъмът на Kruskal се “придвижва” от минималния елемент нагоре до максималния в решетката на разбиванията, на всяка итерация генерирайки разбиване, което абстрактира предишното. При това без да се прескачат нивата в решетката, защото се сливат два дяла в един, което означава, че броят на елементите в разбиването намалява с единица.

11.5 Union-Find структури данни

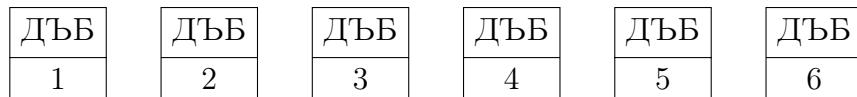
11.5.1 Два крайни подхода, които не вършат работа

Да разгледаме две екстремни възможности за реализация именуването на дяловете на разбиването. Понеже говорим за дървета, нека има дялове ДЪБ и БУК.

- Всеки елемент (връх) е запис, който има поле за името на дяла на разбиването, в който се намира.



При този начин на съхранение на имената, проверката $\text{component}(u) \neq \text{component}(v)$ става във време $\Theta(1)$. Но операцията $\text{identify}(\text{component}(u), \text{component}(v))$ е бавна. Да даваме напълно ново (което досега не е използвано) на дяла-резултат от сливането, примерно ОРЕХ, не е добра идея. Да кажем, че едното от двете използвани имена ДЪБ и БУК става името на дяла-резултат от сливането, а другото име изчезва. Да кажем, че остава името ДЪБ. Тогава след сливането имаме:



Очевидно при този начин на съхранение на имената, сливането на дялове става в линейно, а не в логаритмично време.

- Другата екстремна възможност е името да се пази на едно място, а от всеки елемент да има указател към него.



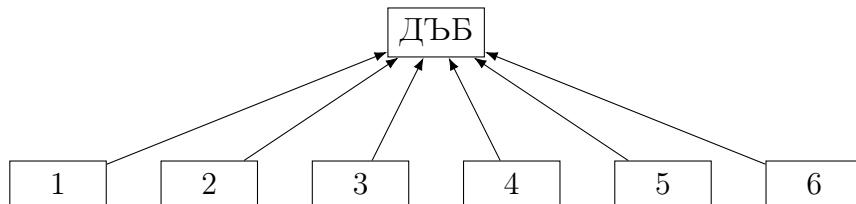
На пръв поглед, това съхранение на имената ни дава възможност да реализираме и Union, и Find в $\Theta(1)$, защото $\text{Find}(u)$, където u е елемент, се състои единствен *dereference* на указателя на u , а $\text{Union}(i, j)$ можем да реализираме, като просто сменим името j с i (в примера, БУК става ДЪБ):



Тази идея, разбира се, не работи. Показаната диаграма е “снимка” на моментно състояние на някакви компоненти, което е недостижимо: поначало компонентите имат по един елемент (защото започваме с разбиването на едноелементни дялове), така че, ако

следваме тази идея, всеки елемент ще носи името на дяла си (само че на един указател разстояние) до края и Union пак ще работи в линейно време.

Ако опитаме да премахваме едното хранилище на име, така щото всеки елемент от даден дял да сочи към само едно хранилище на име, пак ще получим Union, работещ в линейно време:



Печелившата идея е да се откажем от крайностите и да възприемем междинен подход: нито “всеки елемент има свое копие на името на компонентата си”, нито “всички хранилища на имена на компоненти са на един указател разстояние от всеки елемент, през цялото време”.

11.5.2 Реализация на разбиване чрез ориентирана гора

Ориентирана гора е множество от ориентирани дървета без общи върхове. В случая става дума за ориентирани дървета с ориентация от листата към върха (на английски, *in-trees*), точно обратно на арборесценциите от Определение 37. Върховете са еднотипни: всеки връх е запис от идентификатора на елемента в множеството и указател към родителя му. Указателите на корените сочат към себе си. Подобни дървета са дърветата на обхождането в BFS/DFS, реализирани с масив на предшествие π . Козметична разлика е, че при онези дървета указателят на корена има стойност Nil, докато тук указателят на корена е “примка” към себе си. В тази подсекция, когато кажем “дърво”, разбираме ориентирано дърво с ориентация от листата към корена и указател на корена, който е примка към себе си.

Забележете това: “върховете са еднотипни”. При компонентите ДъБ и БУК от миналата подсекция, имената на компонентите бяха съвсем различни от имената на елементите в тях. Математическият подход към описанието на дадено разбиване е точно такъв – ако е дадено множество $Z = \{z_1, \dots, z_n\}$ и разглеждаме някакво разбиване на S , типично именуване на дяловете на разбиването е, да кажем, W_1, \dots, W_k . В практическата реализация на разбиване обаче е лоша идея да се ползват отделни имена за дяловете. В практическите реализации, всеки дял на разбиването се идентифицира с точно един от елементите в него. За този елемент казваме, че е *лидер* на дяла на разбиването (в който се намира). В примера със Z от този параграф, всеки дял на разбиването би бил именуван, или идентифициран, с точно едно z_i от него.

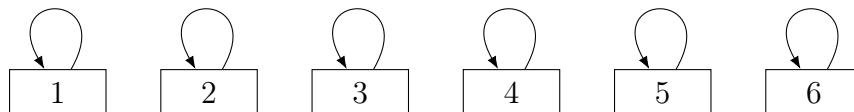
Тези лидери са важни. Функцията $\text{Find}(u)$ връща лидера на компонентата, в която се намира u , а функцията $\text{Union}(i, j)$ работи само върху лидери на компоненти.

Ето псевдокодът на Find . $\pi[u]$ е родителят на u .

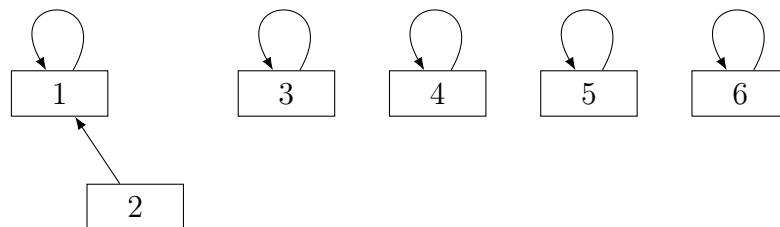
```

Find( $u$ )
1 if  $u \neq \pi[u]$ 
2   return Find( $\pi[u]$ )
3 else
4   return  $u$ 
  
```

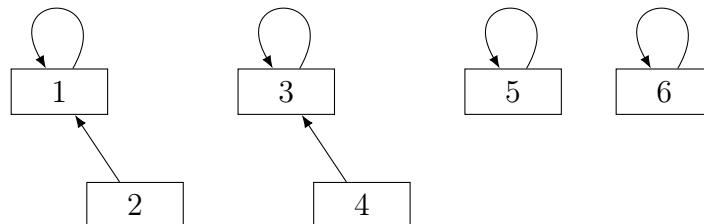
При реализация на разбиване с ориентирана гора, всеки дял се представя чрез точно едно дърво от гората, а лидерът на дяла е коренът – това е естественият избор. Началното разбиване на едноелементни множества, което се има предвид на ред 2 от МПД KRUSKAL[†] се реализира от дървета с по един връх-корен ето така:



В този момент има шест компоненти, всяка с по един елемент, който е и лидерът. Find върху който и да е елемент ще върне самия него. Да кажем, че се извърши Union(1, 2). След това вече има само пет дяла, като единият е от два елемента, а именно 1 и 2. Да кажем, че лидерът на двуелементния дял е 1. Тъй като имплементираме с ориентирани дървета, трябва указателят на 2 вече да сочи към 1:



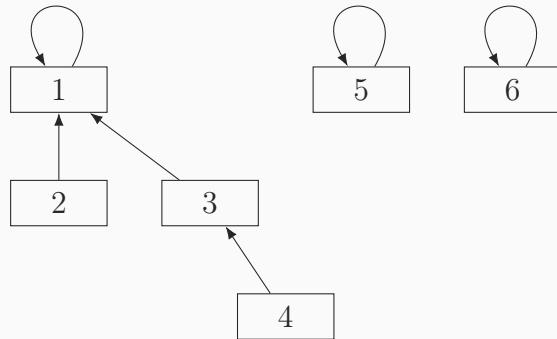
Да кажем, че следващото сливане е Union(3, 4). Резултатът изглежда така:



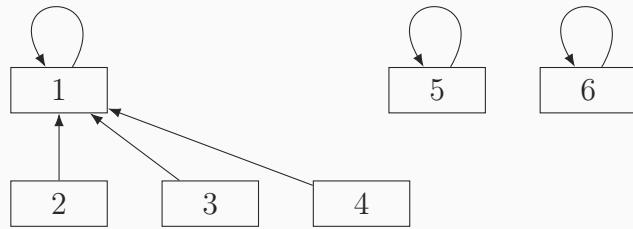
Нека следващото сливане е Union(1, 3). Викането е коректно, защото и 1, и 3 са лидери (викане Union(1, 4) би било некоректно, защото 4 не е лидер). Както се вижда на Фигура 11.3, има два начина да направим дървото на новата компонента.

[†]С други думи, минималният елемент на решетката от Допълнение 32.

Фигура 11.3 : Два варианта за Union.



Вариант 1.



Вариант 2.

Предимството на това, дървото с върхове 1, 2, 3 и 4 да е във Вариант 2 е, че височината му е само едно, за разлика от дървото с тези върхове във Вариант 1, което има височина две. Ако има бъдещо викане на $\text{Find}(4)$, във Вариант 1 ще се наложи да се направят два “скока нагоре”, за да се стигне до лидера, докато във Вариант 2, $\text{Find}(u)$ за кой да е връх $u \in \{1, 2, 3, 4\}$ иска най-много един “скок нагоре”. Предимството на ниските дървета е ясно: бърза работа на Find в най-лошия случай.

Обаче ние предпочитаме Вариант 1. Фигура 11.3 показва малки дървета, но показва ясно, че ако сливаме две произволни дървета, всяко с височина едно, и резултатът трябва да е с височина едно, в най-лошия случай Union ще работи в линейно време в броя на върховете на тези две дървета, който, на свой ред, може да е $\Theta(n)$. А ние искахме Union да работи в логаритмично време. От тези съображения, Вариант 2 отпада. Вариант 1 е изключително бърз: ако лидерите са известни, сливането на дърветата (компонентите) става във време $\Theta(1)$.

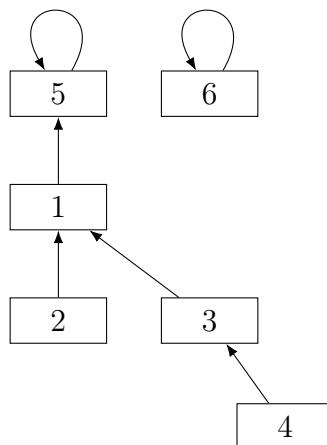
Ние обаче искахме и Union, и Find да работят в не по-лошо от логаритмично време в броя на върховете в двете компоненти, при Union, и в единствената компонента, при Find. Видяхме дори как да реализираме Union в константно време. Ако извършваме обаче Union-ите безразборно, може да получим дървета ненужно голяма височина, поради което Find да стане ненужно бавна.

Наблюдение 38

При реализацията с ориентирана гора, сложността по време на $\text{Find}(u)$ в най-лошия случай е $\Theta(h)$, където h е височината на дървото, чийто връх е u .

Като пример, да разгледаме пак дърветата на Фигура 11.3, Вариант 1. Нека извършим

Union(1,5). Ако направим връх 5 лидер на тази компонента, дървото би било с височина три:

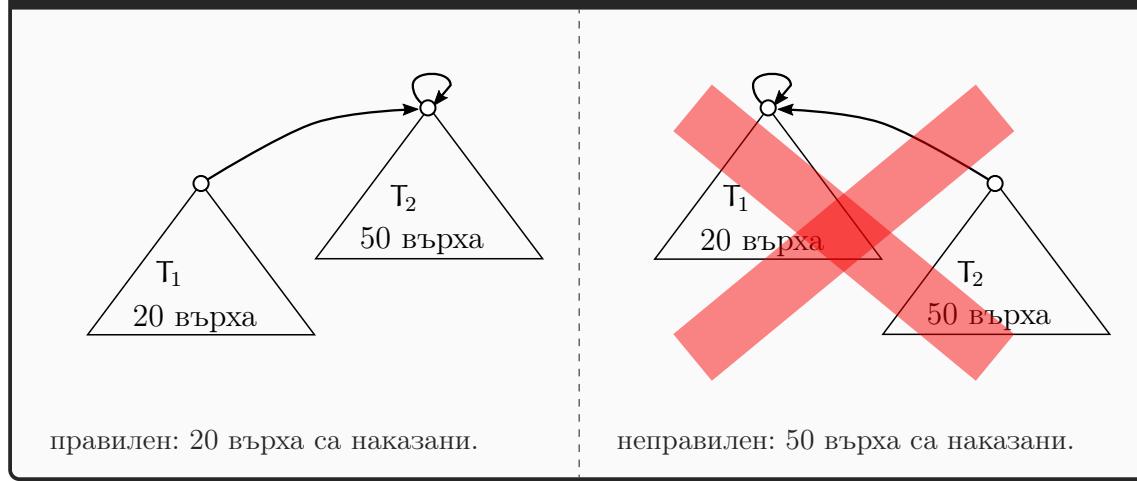


Ясно е, че изборът на 5 за нов лидер беше неуспешен. Нещо повече. Серия от такива неуспешни избори може да доведе до дървата с линейна височина, откъдето в най-лошия случай Find ще работи в линейно време. За да избегнем това, прилагаме две *евристики*[†].

Union by rank. Винаги, когато правим Union, “наказваме” върховете на едно от дърветата с увеличаване на дълбочината им с единица. Ако извършваме Union с просто пренасочване на указателя на единия корен към другия корен, това е неизбежно. Но кое от двете дървета да накажем?

Логично е да накажем дървото с по-малко върхове. В това се състои тази евристика: всяка от компонентите съдържа и информация за броя на елементите в себе си, която е *рангът* на дървото. При Union се променя указателя на корена на дървото с по-малък ранг (ако са с еднакви рангове, решението се взема произволно).

Фигура 11.4 : Union by rank: правилен и неправилен...

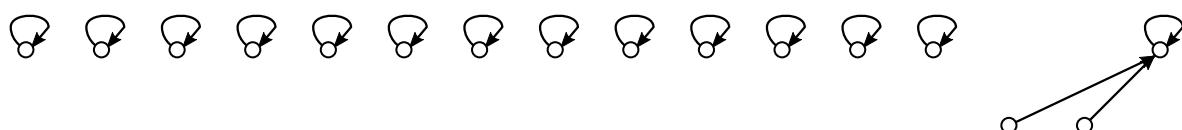


Да разгледаме примери за създаване на дървата (разбивания) чрез Union by rank. Започваме с шестнадесет върха.

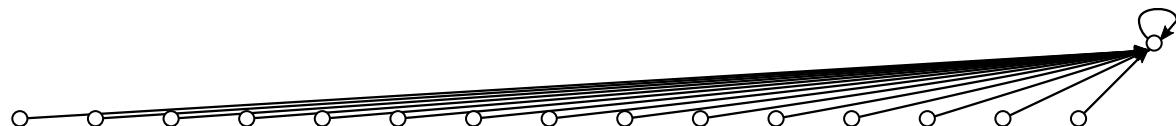
[†]На английски терминът е *heuristic*. Свободно казано, означава добра идея за решаване на някаква задача. Евристиките не гарантират оптималност на решението и дори не дават гаранция за това, колко далечно от оптималното е тяхното решение.



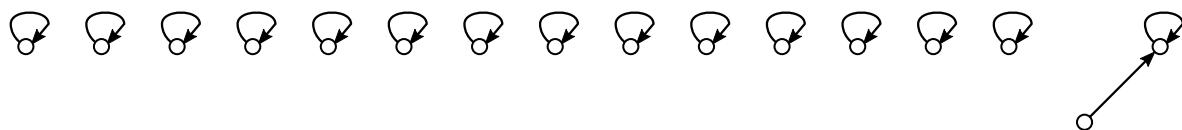
Ако сме късметлии, може сливанията да стават така:



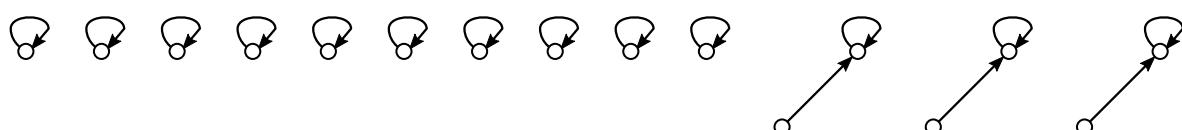
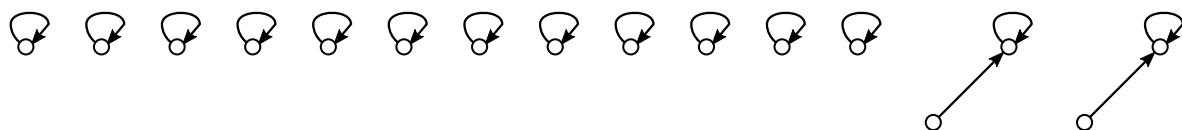
и така нататък, докато получим дърво с височина едно:



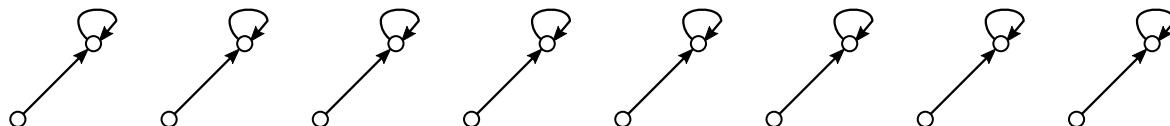
Ако не сме късметлии, може сливанията да стават по следния начин. Първото сливане отново е:



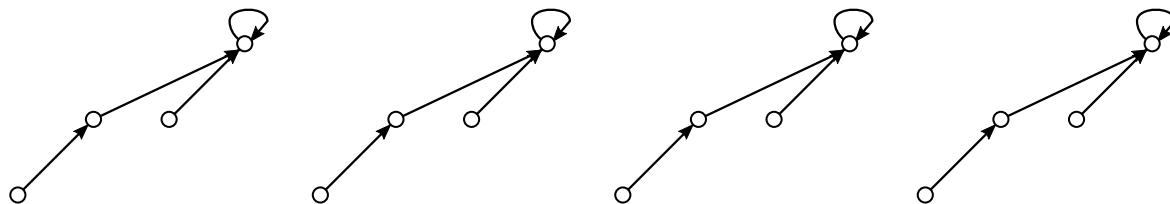
После обаче продължават така:



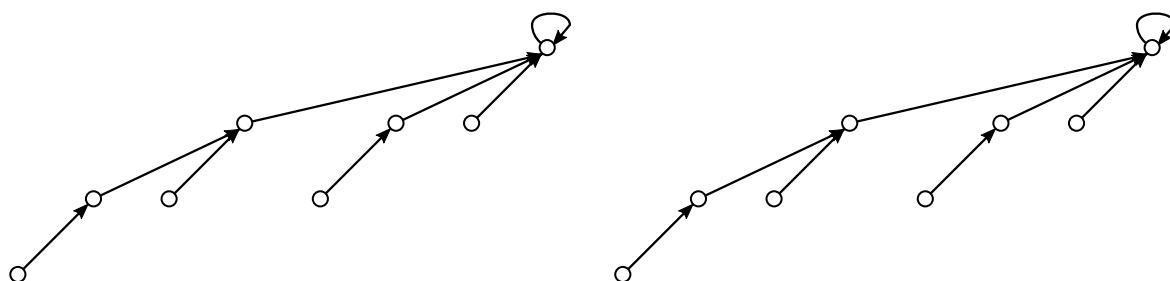
и така нататък до:



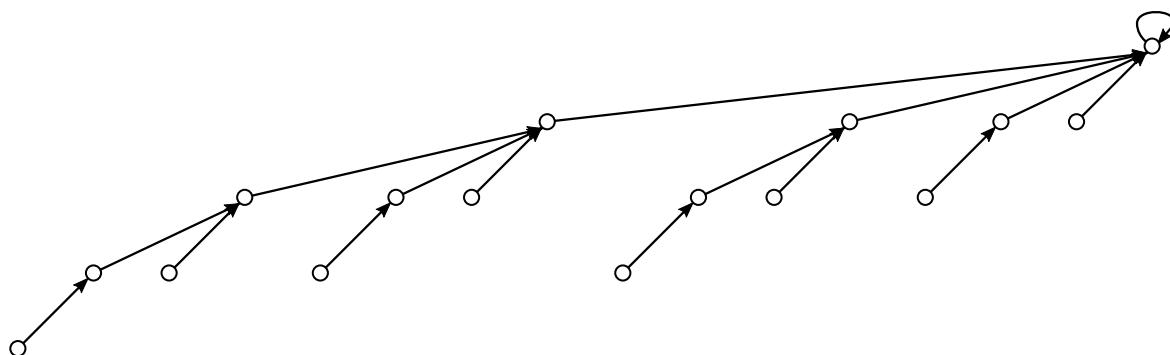
После имаме:



След това:



И накрая:



От тези илюстрации е очевидно, че чрез Union by rank може да получаваме дървета с височина и разклоненост $\lg n$. Но дали това е най-лошата (най-голямата) възможна височина? Оказва се, че да. Сега ще докажем това формално. Само че за лекота на доказателството ще вземем като ранг на дървото не броят на върховете, а височината му.

Теорема 38: Максимална височина на дърво след Union by rank

Започвайки от дървета с по един връх (разбиване на едноелементни множества), след произволна серия от изпълнения на Union by rank, за всяко дърво в колекцията от дървета е вярно, че $h = O(\lg n)$, където h е височината на дървото, а n е броят на **неговите** върхове.

Трябва да се подчертава отново, че n не е общият брой на върховете във всички дървета, а само в дървото, което разглеждаме.

Доказателство: Твърди се, че съществува константа $c > 0$, такава че $h \leq c \lg n$. Ще го докажем с индукция по h .

База $h = 0$. Дървото може да се състои от само един връх, ако височината му е нула, така че $n = 1$. Наистина, за всяка положителна константа е вярно, че $0 \leq c \cdot \lg 1$. ✓

Индукционно предположение Допускаме, че съществува константа $c > 0$, така че за някое $h \geq 0$, всяко дърво, генерирано от Union by rank, $h \leq c \lg n$.

Индукционна стъпка Разглеждаме произволно дърво с височина $h + 1$, генерирано от Union by rank, получено от две дървета с височина h . Зашто две дървета с височина h – защото искаме да видим какво става, когато минаваме към височина $h + 1$, сливайки дървета с по-малки височини, но тези по-малки височини може да са само h и h , ако искаме да получим $h + 1$. Нека тези дървета са T_1 и T_2 , съответно с n_1 и n_2 върха. Предположението е, че

$$h \leq c \lg n_1$$

$$h \leq c \lg n_2$$

БОО, нека $n_1 \geq n_2$. Тогава

$$\begin{aligned} c \lg (n_1 + n_2) &\geq c \lg (n_2 + n_2) = c \lg 2n_2 = c \lg n_2 + c \geq \\ h + c &\geq h + 1 \end{aligned} \quad // \text{съгласно инд. предположение}$$

И така, има такава константа, а именно $c = 1$. □

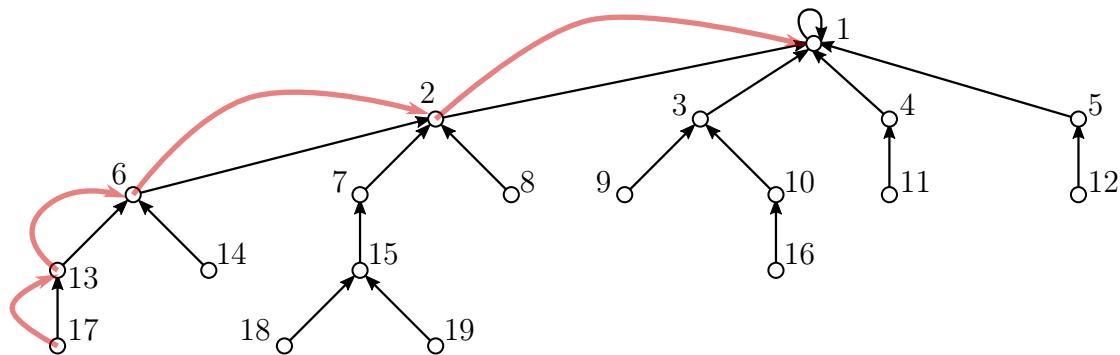
Показахме, че Union by rank генерира дървета с най-много логаритмична в броя на върховете им височина. Това означава, че Find върху такива дървета работи в не по-лошо от логаритично време. На свой ред, това означава, че в МПД KRUSKAL проверката дали $\text{component}(u) \neq \text{component}(v)$ (ред 7) и операцията $\text{identify}(\text{component}(u), \text{component}(v))$ (ред 9) се изпълняват във време $O(\lg n)$. Оттук заключаваме, че МПД KRUSKAL има сложност по време $\Theta(m \lg n)$, като само сортирането ред 3 дава добра граница $\Omega(m \lg n)$.

Path compression. Да си припомним текущия псевдокод на Find.

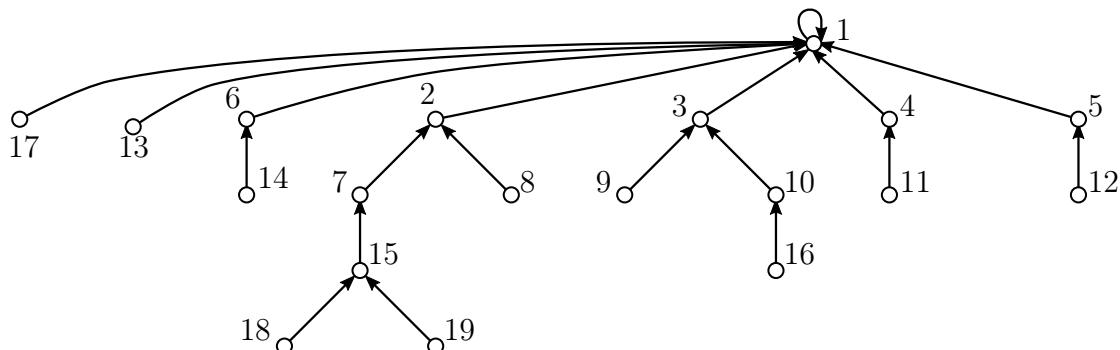
```
Find(u)
1 if u ≠ π[u]
2   return Find(π[u])
3 else
4   return u
```

Още едно подобрение (евристика), която можем да направим към Union-Find структурите, за да ги направим още по-ефикасни, е при всяко изпълнение на Find, при всяко “катерене

нагоре” в дървото за достигане до корена, върховете, през които минаваме, да ги наречем u_1, \dots, u_k , да бъдат направени деца на корена. Това няма да натовари значително Find, защото той така или иначе трябва да измине пътя до корена – защо да не направи дървото “по-сплескано”, така че при следващи изпълнения на функцията Find върху някой от u_1, \dots, u_k , тя да прави точно един “скок нагоре”, за да стигне до корена? Ето пример на дърво в Union-Find структура плюс илюстрация в червено на “скоковете” на Find(17).



Правим всеки от върховете 17, 13 и 6 дете на корена 1.



Ето и псевдокод на подобрения Find.

Find with path compression(u)

```

1 if  $u \neq \pi[u]$ 
2    $\pi[u] \leftarrow \text{Find}(\pi[u])$ 
3 return  $\pi[u]$ 
```

Анализът на сложността на Union и Find операциите при използване на union by rank и path compression евристиките е сложен и нетривиален. В [15, стр. 573] е показано, без стриктно формално доказателство, че при m операции Union или Find върху n елементно опорно множество, в каквато и да е последователност, сумарната сложност по време е $O(m\alpha(n))$, където $\alpha(n)$ е **изключително бавно растяща функция**, наречена *обратната функция на функцията на Ackermann*. Теоретично говорейки, $\alpha(n)$ е неограничено нарастваща функция на

n , но от практическа гледна точка, $\alpha(n) \leq 4$ за всяка стойност на n , която може да възникне на практика. Така че с пълно основание можем да мислим, че $\alpha(n)$ е константа на практика и че сложността на m операции е $O(m)$, което означава сложност $O(1)$ в амортизирания смисъл.

Това подобрене на сложността на Union и Find не води до подобрене на асимптотичната оценка за сложността на алгоритъма на Kruskal. Тя си остава $\Theta(m \lg n)$ заради сортирането. Но Union-Find структурите се ползват не само за реализация на гората в алгоритъма на Kruskal, а и винаги, когато трябва бързо да се установява свързаност между елементи и бързо да се сливат компоненти, така че възможността да правим всяка от тези операции в константно амортизирано време е нещо забележително и полезно.

Лекция 12

Намиране на най-къси пътища. Алгоритми на Dijkstra, Bellman-Ford и Floyd-Warshall.

Резюме: Въвеждаме задачата за намиране на най-къси пътища в ориентирани тегловни графи в няколко варианта. минимално покриващо дърво на тегловен граф. Разглеждаме проблемите, възникващи при наличие на ребра с отрицателни тегла. Разглеждаме два ефикасни алгоритъма за намиране на най-къси пътища от даден връх до всички останали: алгоритмите на Dijkstra и на Bellman-Ford. Разглеждаме ефикасни алгоритми за намиране на къси пътища от всеки до всеки връх: алгоритъм, реализиращ матрично умножение, и алгоритъмът на Floyd-Warshall.

12.1 Фундамент

12.1.1 Базови дефиниции

В тази лекция по подразбиране разглеждаме ориентирани тегловни графи. Ако в даден момент разглеждаме неориентирани графи, това ще се каже изрично. Примки не се допускат, понеже те или нямат отношение към най-късите пътища, ако теглата им са положителни, или представляват отрицателни цикли, ако теглата им са отрицателни – а отрицателните цикли са “проклятието” на задачата за най-къси пътища, както ще видим нататък. Не се допускат и паралелни ребра, тъй като от всеки спон паралелни ребра, за най-късите пътища значение има само най-леко ребро. Така че и мултиграфи не се допускат.

По подразбиране $G = (V, E)$ е ориентиран тегловен граф с тегловна функция $w : E \rightarrow \mathbb{R}$.

Конвенция 6

В тази лекция, казвайки “път”, имаме предвид път, който не е непременно прост.

Нотация 7

Нека $u, v \in V$. Нотацията “ $u \rightsquigarrow v$ ” е кратък запис за “ориентиран път от u до v ”. Нотацията “ $u \rightsquigarrow^p v$ ” е кратък запис за “ p е ориентиран път от u до v ”.

Определение 57: Отрицателен цикъл

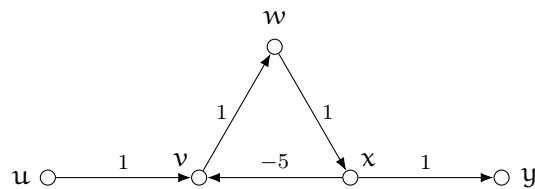
Нека G е тегловен граф. Ако G е ориентиран, *отрицателен цикъл* е всеки прост цикъл в G , който има отрицателна сума от теглата на ребрата. Ако G е неориентиран, отрицателен цикъл е всеки прост цикъл в G , който има отрицателна сума от теглата на ребрата, както и всяко ребро с отрицателно тегло.

Наблюдение 39: Проблем при отрицателните цикли.

Ако теглото на път е сумата от теглата на ребрата в него и теглата са само положителни, “най-къс път от u до v ” е добре дефинирано понятие. Обаче за всеки отрицателен цикъл c , за всеки път p , който съдържа c , съществува път p' , такъв че $w(p') < w(p)$ (и p' съдържа c повече пъти от p). Еrgo, ако поне един път от връх u до връх v съдържа поне един отрицателен цикъл, не можем да дефинираме “най-къс път от u до v ”.

Ако няма отрицателни цикли, този проблем не съществува дори при наличие на отрицателни тегла.

Като пример, да разгледаме следния граф:



Ако разглеждаме само прости пътища, най-късият път (той е и единствен) $u \rightsquigarrow y$ е пътят u, v, w, y с тегло 4. Ако обаче разглеждаме пътища, които не са непременно прости, най-къс път $u \rightsquigarrow y$ няма, защото всяко минаване през цикъла v, w, x, v добавя $1 + 1 - 5 = -3$ към теглото на пътя:

u, v, w, x, v, w, x, y има тегло 1

$u, v, w, x, v, w, x, v, w, x, y$ има тегло -2

$u, v, w, x, v, w, x, v, w, x, y$ има тегло -5

и така нататък

Наблюдение 40: Когато няма отрицателни цикли.

При липса на отрицателни цикли, за всеки два върха u и v , всеки най-къс път от u до v задължително е прост път.

Определение 58: Тегло на път

Теглото на пътя p е $w(p) = \sum_{e \in E(p)} w(e)$. Теглото на най-къс път от u до v е

$$\delta(u, v) = \begin{cases} \min \{w(p) \mid u \rightsquigarrow^p v\}, & \text{ако има поне един } u \rightsquigarrow v \\ \infty, & \text{ако няма такъв път} \end{cases}$$

Забележете, че ако съществува поне един $u \rightsquigarrow v$, който съдържа отрицателен цикъл, то, съгласно Наблюдение 39, $\min \{w(p) \mid u \xrightarrow{p} v\}$ става $-\infty$. Причината да дефинираме като ∞ теглото на най-къс път при липса на поне един път е очевидна. И така, $\delta(u, v) \in \mathbb{R} \cup \{-\infty, \infty\}$.

Има леко терминологично несъответствие: щом говорим за “тегло на път”, би трябвало да казваме “най-лек път” за път с минимално телго, а не “най-къс път”. Но както “тегло на път”, така и “най-къс път” (в тегловния смисъл) вече са широко приети както на български, така и на английски, като съответно се казва “weight of path” и “shortest path weight”, така че в тези лекции ще се съобразим с това. И така, ще се придържаме към следната езикова конвенция.

Конвенция 7

В контекста на задачата, която разглеждаме, “най-къс път” е синоним на “път с най-малко тегло”. Ако имаме предвид броя на ребрата в пътя, ще го кажем експлицитно.

Конвенция 8

Тегловните функции, които ще разглеждаме, или имат кодомейн \mathbb{R}^+ , или имат кодомейн \mathbb{R} , но в G няма отрицателни цикли. Поради това никъде няма да дефинираме пътищата като прости пътища. Те ще се оказват прости вследствие на отсъствието на отрицателни цикли, предвид Наблюдение 40.

Задачата за най-къс път е в няколко варианта. По отношение на началото и края на най-къс път вариантите на задачата са следните.

Изч. Задача: SINGLE PAIR SHORTEST PATH

пример: Ориентиран граф $G = (V, E)$, тегловна функция w , начален връх s , краен връх t .

решение: $\delta(s, t)$

Изч. Задача: SINGLE SOURCE SHORTEST PATHS

пример: Ориентиран граф $G = (V, E)$, тегловна функция w , начален връх s .

решение: $\forall v \in V : \delta(s, v)$.

Изч. Задача: SINGLE DESTINATION SHORTEST PATHS

пример: Ориентиран граф $G = (V, E)$, тегловна функция w , краен връх t .

решение: $\forall v \in V : \delta(v, t)$.

Изч. Задача: ALL PAIRS SHORTEST PATHS

пример: Ориентиран граф $G = (V, E)$, тегловна функция w .

решение: $\forall u, v \in V : \delta(u, v)$.

Първият вариант на задачата, а именно от даден връх s до даден връх t , изглежда алгоритично най-лесен. Както ще се убедим обаче, в най-лошия случай, за да изчислим теглото на

най-лек път $s \rightsquigarrow t$, се налага да изчислим и теглото на най-лек път $s \rightsquigarrow v$, за всеки $v \in V$. Еrgo, в най-лошия случай, първият вариант е труден колкото втория.

Ние ще разгледаме подробно втория вариант, като за краткост ще го наричаме SSShP.

Третият вариант има алгоритмичната трудност на втория, защото третият вариант се превръща във втория при обръщане на посоките на ребрата; тоест, на транспониране на графа.

Четвъртият вариант ще бъде разгледан подборно. За краткост ще го наричаме APShP. Очевидно APShP се редуцира лесно до SSShP, защото, ако имаме алгоритъм за SSShP, можем да го пуснем от всеки връх и да намерим решение. Но има интересни ефикасни алгоритми за APShP, които решават само APShP в смисъл, че нямат ограничена версия, която да решава SSShP по-ефикасно, и ние ще ги разгледаме и изследваме.

Ще разгледаме и друга класификация на задачата за най-къси пътища по това, какво искаме за най-къс път $u \rightsquigarrow v$.

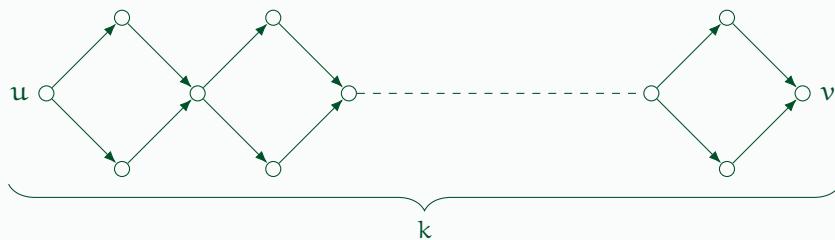
- Може да искаме само дълчината, а не самия път. Това не е много полезно на практика.
- Може да искаме дълчината плюс самия път, което в реални приложения е много полезно и информативно.
- Може да искаме броя на най-късите пътища $u \rightsquigarrow v$.
- Може да искаме множеството от всички най-къси пътища $u \rightsquigarrow v$.

Ние ще разглеждаме алгоритми, които връщат дълчината на най-къс път плюс самия път.

Да обобщим: ще решаваме задачата във вариант SSShP, като за всеки връх-край на най-къс път, ще изчисляваме както дълчината на пътя, така и самия път.

Допълнение 33: Генерирането на всички най-къси пътища е неефикасно.

В реални приложения може да е много полезно да бъдат получени всички най-къси пътища от връх до друг връх. Имайки всички най-къси пътища, може да изберем от тях път по някакъв друг критерий. За съжаление, в най-лошия случай, броят на най-късите пътища $u \rightsquigarrow v$ може да е експоненциален в n , в което може да се убедим с този прост пример (приемете, че теглата са единици):



Виждаме k на брой подграфи, да ги наречем *ромбите*, като всеки ромб е . Ромбите са “слепени” в редица. Очевидно на фигурата има 2^k пътя от u до v , всеки от тях с дължина $2k$, защото за всеки ромб, може да минем “отгоре” или “отдолу”. И тъй като е възможно $k \leq n$, в графа може да има $\Omega(2^n)$ пътя от u до v . Следователно, няма ефикасен алгоритъм, който строи всички пътища, защото само записването им отнема време $\Omega(2^n)$ в най-лошия случай.

Следният резултат е Lemma 24.1 в [15, стр. 645].

Теорема 39: Най-къс път се състои от най-къси подпътища.

Нека G е ориентиран тегловен граф, s и t са върхове в него и p е най-къс път от s до t . Нека $u, v \in V(p)$, като u предхожда v от s към t в p , ако u и v са различни^a. Нека подпътят на p от u до v се назава q :

$$p = \underbrace{s \rightsquigarrow u \rightsquigarrow \underbrace{v \rightsquigarrow t}_{q}}_p$$

Тогава q е най-къс път от u до v .

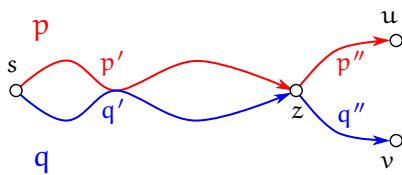
^aНе е необходимо s, t, u и v да са четири различни върха; в най-екстремния вариант, може $s = t = u = v$ и пак остава вярно, че най-къс път се състои от най-къси подпътища.

Доказателство: Ако допуснем, че има път q' , такъв че $u \rightsquigarrow v$ и $w(q') < w(q)$, веднага заключаваме, че p не е най-къс път от s до t , понеже замяната на q с q' в p намалява $w(p)$ с $w(q) - w(q')$. \square

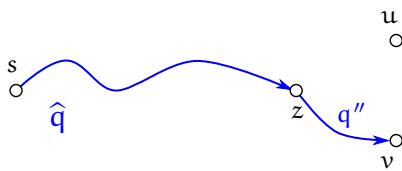
В доказателството на Теорема 39 се възползваме от това, че пътищата не са непременно прости. Ако настоявахме пътищата да са прости, то вмъкването на q' на мястото на q може да е проблематично, понеже няма гаранции, че q' и p нямат други общи върхове освен u и v ; ерго, след вмъкването целият път може да не е прост. Ако теглата са положителни, можем “да изрежем” общите части на p и q' и да получим път p'' , който е с дори още по-малко тегло, което пак ни дава желаното противоречие. Обаче при отрицателни тегла и по-специално при наличието на отрицателни цикли, това “изрязване” може да увеличи теглото на пътя, което е проблем за доказателството.

Дърво на най-късите пътища. Ако искаме да изчислим най-къс път $s \rightsquigarrow t$, паметта, която е необходима за записването на пътя е $\Theta(n)$ в най-лошия случай, защото, в най-лошия случай, дължината на пътя е $\Theta(n)$ и не можем да избегнем записването на всеки връх от него. На пръв поглед, ако искаме да запишем по един най-къс път $s \rightsquigarrow v$ за всеки $v \in V$, ще ни трябва $O(n^2)$ памет (може да се покаже, че асимптотичната горна граница $O(n^2)$ е точна, така че имаме право да пишем $\Theta(n^2)$).

Всъщност, можем “да минем” само с $\Theta(n)$ памет за всички пътища, защото можем да ги представим с ориентирано кореново дърво-арборесценция, която може да се представи с масив на предшествията $\pi[1, \dots, n]$, също както при BFS и DFS. Ще покажем, че най-къси пътища от s могат да се представят с една арборесценция с корен s . Нека p и q са най-къси пътища от s съответно до u и v , където u и v са различни върхове. Ако единственият общ връх на p и q е s , няма какво да се показва. Нека p и q имат поне един общ връх освен s . Нека z е най-отдалеченият в p и в q връх от s , който е общ за p и q . Очевидно не може $z = u = v$, понеже u и v са различни. Ако $z = u \neq v$ или $z = v \neq u$, няма какво повече да показваме; това е случаите, в които съответно p е част от q или q е част от p . Остава да разгледаме случая, в който $z \neq u$ и $z \neq v$. Нека подпътят на p от s до z е p' , а подпътят на q от s до z е q' :

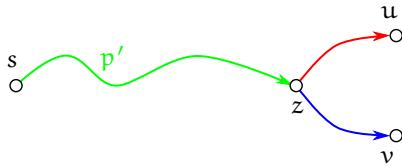


Твърдим, че $w(p') = w(q')$. Да допуснем противното. БОО, нека $w(p') < w(q')$. Тогава в q подменяме q' с p' :



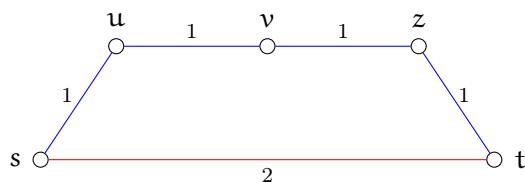
и получаваме път \hat{q} от s до v , такъв че $w(\hat{q}) < w(q)$, в противоречие с допускането, че q е най-къс път от s до v .

Щом $w(p') = w(q')$, може и в p , и в q да използваме само единият от тях, да кажем p' ; по този начин, получаваме най-къси пътища от s до u и от s до v , които се състоят от един общ подпът от s до z , след което се “разделят” и повече не се събират:



Ако си направим същото нещо за всеки два върха, достигими от s , ще получим арборесценция с корен s , което е дървото на най-късите пътища от s .

Задачата за най-късите пътища и задачата за МПД са различни. Естествено, те са задачи върху различни видове графи, МПД е върху неориентирани тегловни графи, а най-късите пътища са върху ориентирани тегловни графи, но съществената разлика не е в това. Дори да решаваме задачата за най-късите пътища върху неориентирани тегловни графи, тя остава принципно различна от задачата за МПД. Ето малък пример:



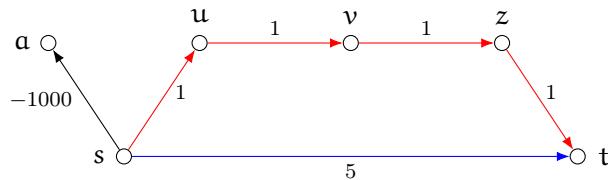
Най-късият път между s и t е реброто (s, t) с тегло 2 (в червено). Но МПД-то (то е само едно) се състои от четирите сини ребра, всяко с тегло 1. Ако си представим работата на алгоритъма на Kruskal върху този граф, сортирайки ребрата, той ще постави в началото четирите ребра с тегло 1 и след това реброто с тегло 2. Когато започне да слага ребра, той ще сложи четирите ребра с тегло 1 и ще спре, без да достигне до реброто с тегло 2, което реализира най-късия път.

Наблюдение 41

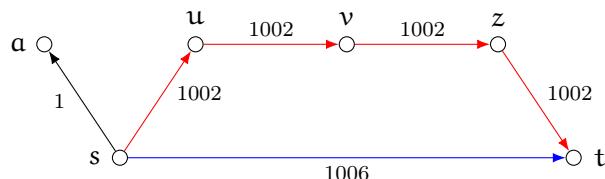
Задачите за намиране на МПД и намиране на дърво на най-късите пътища върху неориентирани тегловни графи са принципно различни. Всяко МПД, за всеки два върха s и t , дава (уникален) път p между s и t , но в общия случай p не е най-къс път в графа между s и t .

Друга важна разлика между тези две задачи е възможността да се “отървем” от отрицателни тегла. Когато дефинирахме задачата за МПД, дефинирахме теглата като реални числа, оставяйки възможността те да са отрицателни. Наистина, нищо не пречи теглата да са отрицателни в Теорема 36 или в алгоритмите на Prim и Kruskal. Ако обаче по някаква причина неискаме отрицателни тегла там, лесно може да се отървем от тях, като намерим най-малкото отрицателно тегло x и после добавим $|x| + 1$ към теглата на всички ребра. Те ще станат положителни, а МПД-тата на графа ще останат **същите**.

Този трик не работи при най-късите пътища. Ако “повдигнем” теглата достатъчно, така че всички те да станат положителни, дървото на най-късите пътища може да се промени. Ето малък пример за това, този път с ориентиран граф:



Очевидно най-късият път от s до t се състои от червените ребра и има тегло 4. Ако обаче добавим 1001 към теглото на всяко ребро, така че теглата да станат само положителни:



пътят от червените ребра става с тегло 4008, а този със синьото ребро, само 1006, така че сега той е най-късият път от s до t .

Наблюдение 42

Нека G тегловен неориентиран граф с тегловна функция $w : E \rightarrow \mathbb{R}$. Нека $w' : E \rightarrow \mathbb{R}$ е друга тегловна функция, като $\forall e \in E : w'(e) = w(e) + c$, където c е произволна реална константа. МПД-тата на G по отношение на w и по отношение на w' са едни и същи. Ако обаче G тегловен ориентиран граф с тегловна функция $w : E \rightarrow \mathbb{R}$ и $w' : E \rightarrow \mathbb{R}$ е друга тегловна функция, като $\forall e \in E : w'(e) = w(e) + c$, където c е произволна реална константа, дървото на най-късите пътища по отношение на произволен $s \in V(G)$ може да бъде различно за w и за w' .

12.1.2 Пак за отрицателните тегла

Както се убедихме, отрицателните тегла представляват значителен проблем за задачата за най-късите пътища (за разлика от задачата за МПД, където те нямат никакво значение). При наличие на отрицателни тегла ние може дори да не сме в състояние да дефинираме “най-къс път от s до t ”, ако поне един път от s до t съдържа отрицателен цикъл.

Това се дължи на факта, че разглеждаме пътища, които не са непременно прости, което влече, че всяко “завъртане” през отрицателен цикъл дава още по-къс път. Не може ли да постулираме, че разглеждаме само прости пътища, и по този начин да няма възможност да се “въртим” в циклите поначало? Отговорът е, че теоретично можем да го направим, но алгоритмично не можем да го имплементираме по ефикасен начин. Алгоритмите за най-къси пътища, примерно този на Dijkstra, не правят проверка дали изградените пътища са прости или не. Те се оказват прости (вижте Наблюдение 40; в алгоритъма на Dijkstra се иска теглата да са положителни) или, ако се окаже, че има повторение на върхове (алгоритъмът на Bellman-Ford “усеща” повторение на върхове) следва съобщение за грешка, тоест, алгоритъмът отказва да търси най-къси пътища след установяване на съществуване на отрицателен цикъл. В учебника на Skiena (вж. [63, стр. 303–304]) е показана схема, по която може да се построи алгоритъм за най-дълги пътища, който работи с прости пътища и предотвратява повтаряне на върхове. За съжаление, такъв алгоритъм би бил с експоненциална сложност по време.

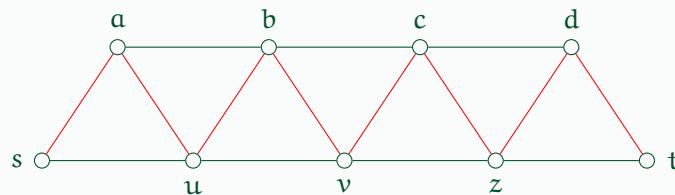
Силна улика за това, че не може да решим проблема с отрицателните цикли с елементарни средства—примерно, постулиране, че не разглеждаме други пътища освен прости—е фактът, че задачата за най-къси пътища е същата като задачата за най-дълги пътища при обръщане на знака на теглата. Иначе казано, ако G е ориентиран тегловен граф, върху който са дефинирани тегловни функции $w, w' : E \rightarrow \mathbb{R}$, като $\forall e \in E(G) : w(e) = -w'(e)$, то, за всеки $s, t \in V(G)$, p е най-къс път $s \rightsquigarrow t$ по отношение на w тогава p е най-дълъг път $s \rightsquigarrow t$ по отношение на w' . Задачата за най-дълги пътища е **NP-пълна**, което означава, че почти сигурно за нея няма ефикасен алгоритъм. Какво означава “**NP-пълна**” и защо задачата за най-дългите пътища е такава, ще видим в Лекция 14, Лекция 15 и Лекция 16.

Допълнение 34: За структурата на най-дългите пътища

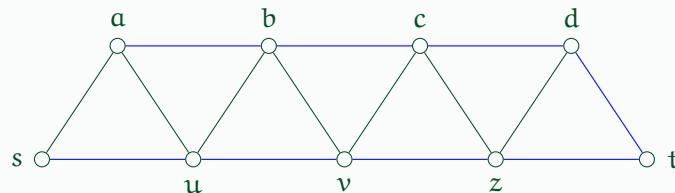
За задачата за най-дългите пътища не е вярно, че оптималната структура се състои от оптимални подструктури (сравнете с Теорема 39). Това обяснява драстичната разлика между ефикасността на най-добрите известни алгоритми за най-къси пътища и за най-дълги пътища.

Преди всичко, за да дефинираме смислено задачата за най-дългите пътища, трябва да постулираме, че разглеждаме само прости пътища. Иначе, дори в не-тегловния вариант,

задачата е лошо дефинирана, понеже съществуването на цикъл в ориентиран граф или на ребро в неориентиран граф влече съществуването на неограничено дълги, като брой ребра, пътища, които не са прости. И така, ако дефинираме задачата за най-дълги пътища по очевидния начин, настоявайки пътищата да са прости, е лесно да намерим пример, в който най-дългият път не се състои от най-дълги подпътища (допускаме, че теглата са единици):



Най-дългият път между s и t е с дължина 8 и е нарисуван с червено. В него има подпът между s и a с дължина 1. Но най-дългият път между s и a е с дължина 8, както се вижда на следната фигура (в синьо):



Очевидно не е вярно, че най-дългите пътища се състоят от най-дълги подпътища.

След всички изложени проблеми, които възникват при отрицателни тегла на ребрата, възниква въпроса, а защо не се откажем от отрицателни тегла поначало? Наистина, ако теглата моделират някакви физически величини, които по природа да положителни, примерно закъснения във времето или разстояния в реалния свят, няма как да се появят отрицателни тегла; отрицателно закъснение, например, означава пътуване назад във времето. Обаче има важни практически задачи, в които е удобно да се работи с абстракцията на отрицателните числа. Примерно, ако моделираме някакви потоци от стоки или пари, можем да кажем, че от сметката на X са прехвърлени 100 лева в сметката на Y, като кажем, че от сметката на Y са прехвърлени -100 лева в сметката на X. Така че, ако се откажем от отрицателни тегла изобщо, графиките ни ще станат със строго по-малка моделаща мош.

Допълнение 35: Намиране на арбитражи чрез отрицателни цикли

Арбитраж, на английски *arbitrage*, е термин от финансите и търговията, който означава, най-общо казано, купуване на нещо, последване от продаването на същото нещо (по всяка вероятност, не на същото място), като, естествено, купуването е на ниска цена, а продаването, на висока цена.

В случая става дума за търговия на валути. В този контекст, арбитраж се дефинира като почти едновременното купуване и продаване на ценни книжа или чуждестранна валута в различни пазари с цел печалба от разминавания в курсовете. Като пример, да разгледаме валутите евро EUR, лев BGN, унгарски форинт HUF, чешка крона CZK

и монголски тугрик MNT. Дадени са обменните курсове в квадратна матрица $M[i, j]$, като $M[i, j]$ за $i \neq j$ има смисъл на количество валута j , която можем да закупим за единица валута i . Да допуснем, че матрицата има свойството $M[i, j] = \frac{1}{M[j, i]}$ при $i \neq j$ (обратното би било арбитраж върху само две валути).

Ето примерна матрица с курсовете на тези валути от 18 май 2020 г. Курсовете са истинските с изключение на двете числа в червено.

	EUR	BGN	HUF	CZK	MNT
EUR	1	1.95583	352.604	27.6033	3 035.51
BGN	0.51129	1	180.241	14.1150	1 551.88
HUF	0.0028360	0.0055481	1	0.063403	8.60872
CZK	0.036227	0.0055481	15.77188	1	110.009
MNT	0.00032943	0.00064437	0.116161	0.0090834	1

Да кажем, че искаме да продадем и купим валути в тази последователност:

$$\text{EUR} \rightarrow \text{BGN} \rightarrow \text{MNT} \rightarrow \text{CZK} \rightarrow \text{HUF} \rightarrow \text{EUR}$$

Съгласно дадената матрица, за всяко евро, с което започваме, ще получим

$$1.95583 \times 1551.88 \times 0.0090834 \times 15.77188 \times 0.0028360 = 1.233182546$$

евро. С други думи, печалба от над 23%^a.

Задача е, при дадена матрица M , да се определи дали има редица от валути $c_{i_1}, c_{i_2}, \dots, c_{i_k}$, такава че

$$M[i_1, i_2] \times M[i_2, i_3] \times \cdots \times M[i_{k-1}, i_k] \times M[i_k, i_1] > 1 \quad (12.1)$$

Това е Problem 24.3 в [15, стр. 679]. Тази задача има добре известно решение, което я свежда до задачата за намиране на отрицателен цикъл в граф. Логаритмуваме двете страни в (12.1) и получаваме:

$$\lg(M[i_1, i_2]) + \lg(M[i_2, i_3]) + \cdots + \lg(M[i_{k-1}, i_k]) + \lg(M[i_k, i_1]) > 0 \quad (12.2)$$

което е същото като

$$-\lg(M[i_1, i_2]) - \lg(M[i_2, i_3]) - \cdots - \lg(M[i_{k-1}, i_k]) - \lg(M[i_k, i_1]) < 0 \quad (12.3)$$

Сведохме задачата за намиране на отрицателен цикъл в пълен ориентиран граф, като върховете са валутите, а теглото на реброто (i, j) е $-\lg(M[i, j])$. Алгоритъмът на Bellman-Ford може да открива наличието на отрицателни цикли и с малка модификация може да намира отрицателен цикъл.

^aЕстествено, в реалния живот не може да се правят толкова лесни печалби. Тук това става заради нагласените числа.

12.1.3 Релаксация

Става дума за задачата за най-късите пътища в SSShP варианта. Даден е някакъв стартов връх s . Според [15, стр. 647], *релаксация* е техника за подобряване на *оценката* (*estimate* на английски), по отношение на произволен връх $u \in V(G)$, на теглото на най-къс път $s \rightsquigarrow u$. Както ще видим надолу, тази оценка е всъщност горна граница, която в алгоритмите, които ще разгледаме, започва с “ ∞ ” (тоест, нищо не знаем) и става все по-добро, докато не стане точна (горна граница) и остане така до края на алгоритъма.

Тази оценка за връх u може да бъде записана като $u.d$, а може да има масив $d[1, \dots, n]$ и оценката за u да е $d[u]$. Във всеки случай, това е реално число, което има смисъл на “най-ниската горна граница за минималното тегло на $s \rightsquigarrow^p u$, която можем да дадем въз основа на наличната информация”.

Както вече казахме, ние искаме не просто дълчините на най-къси пътища, а и самите пътища, реализирани чрез масив на предшествия. На свой ред, тези стойности (идентификаторът на родителя в дървото) може да се записват като $u.\pi$, а може да има масив $\pi[1, \dots, n]$, като $\pi[u]$ е родителят на u в дървото (родителят на корена е Nil).

Оценките се инициализират от следната процедура, която ще наричаме кратко “ISS”, което идва от Initialize-Single-Source.

$\text{ISS}(G: \text{ориентиран тегловен граф}, s: \text{връх в } G)$

- 1 **foreach** $v \in V(G)$
- 2 $v.d \leftarrow \infty$
- 3 $v.\pi \leftarrow \text{Nil}$
- 4 $s.d \leftarrow 0$

Самата релаксация става така. Контекстът е, както очакваме, ориентиран тегловен граф G и тегловна функция w .

$\text{RELAX}((x, y) \in E(G))$

- 1 **if** $y.d > x.d + w((x, y))$
- 2 $y.d \leftarrow x.d + w((x, y))$
- 3 $y.\pi \leftarrow x$

Смисълът е напълно ясен: опитваме да намерим път с по-малко тегло $s \rightsquigarrow y$ от този, който е оптимален до момента. Стойността $y.d$ има смисъл на дължината на най-къс път $s \rightsquigarrow y$ според информацията до момента. Опитваме да намерим по-къс път до y , ползвайки x като предпоследен връх. Ако се окаже, че $y.d \leq x.d + w((x, y))$, то опитът е неуспешен и просто прескачаме. Ако обаче $y.d > x.d + w((x, y))$, наистина има такъв по-къс път—при допускането, че $x.d$ е смислена и π -стойностите реализират дърво на пътища—така че подменяме $y.d$ и $y.\pi$ по адекватен начин.

Изборът на името “релаксация” може ли изглежда странен. Чрез тази примитивна процедура ние всъщност **затягаме** горната граница (оценката) за дължината на най-къс път, а не я отпускаме. Според [15, стр. 647], произходът на името “релаксация” е следният. Резултатът от $\text{RELAX}((x, y))$ може да бъде разглеждан като релаксиране на ограничението (constraint) $y.d \leq x.d + w((x, y))$; съгласно неравенството на триъгълника, това ограничение трябва да е в сила, ако $y.d = \delta(s, y)$ и $x.d = \delta(s, x)$. С други думи, ако е вярно, че $y.d \leq x.d + w((x, y))$, то няма защо да се притесняваме за удовлетворяването на това ограничение, така че можем да го релаксираме.

Свойства на релаксацията В сила са следните твърдения. Всички те са от [15, стр. 650]

Лема 48: Неравенство на триъгълника

За всяко ребро $(x, y) \in E(G)$ е вярно, че $\delta(s, y) \leq \delta(s, x) + w((x, y))$.

Доказателство: Ако няма път $s \rightsquigarrow x$, то $\delta(s, x) = \infty$ и дясната страна е ∞ . Неравенството остава в сила независимо от това, дали няма път $s \rightsquigarrow y$, което води до, че лявата страна е ∞ , или има път $s \rightsquigarrow y$, което води до, че лявата страна е число.

Ако има път $s \rightsquigarrow x$, то има път $s \rightsquigarrow y$, който съдържа x като предпоследен връх и има тегло $\delta(s, x) + w((x, y))$. \square

Лема 49: Горна граница и точна горна граница

Нека G е инициализиран с $\text{ISS}(G, s)$. Тогава, за всеки връх $v \in V(G)$, $v.d \geq \delta(s, v)$, след произволна поредица от релаксации на ребрата на G . Нещо повече, при достигане на равенство, то не се променя след произволна поредица от релаксации на ребрата на G .

Сравнете Лема 49 с Лема 40.

Доказателство: Ще докажем, че $v.d \geq \delta(s, v)$ за всеки връх v , след произволна поредица от релаксации на ребра, по индукция по броя на релаксациите.

Базата е за нула релаксации, тоест, d -стойностите са както са генериирани от ISS , а именно $s.d = 0$ и $v.d = \infty$, ако $v \neq s$. Вярно е, че $s.d \geq \delta(s, s)$, защото $s.d = 0$ заради ISS , а $\delta(s, s) = 0$ или $\delta(s, s) = -\infty$ в зависимост от това, дали s не е, или е, върху отрицателен цикъл. И в двата случая е вярно, че $s.d \geq \delta(s, s)$. Нека $v \neq s$. Вярно е, че $v.d \geq \delta(s, v)$, защото $v.d = \infty$ заради ISS , а $\delta(s, v) \in \mathbb{R} \cup \{-\infty, \infty\}$.

Да допуснем, че твърдението е вярно след никаква произволна серия от релаксации. Разглеждаме $\text{RELAX}((u, v))$, за някое $(u, v) \in E(G)$, веднага след тази серия. Единствената d -стойност, която може да се промени от $\text{RELAX}((u, v))$, е $v.d$. Ако тя изобщо се промени, промяната ще е такава:

$$\begin{aligned} v.d &= u.d + w((u, v)) \quad // \text{защото RELAX работи така} \\ &\geq \delta(s, u) + w((u, v)) \quad // \text{от индуктивното предположение за } u \\ &\geq \delta(s, v) \quad // \text{от Лема 48} \end{aligned}$$

За да се убедим, че, ако веднъж се постигне равенство $v.d = \delta(s, v)$, то остава в сила във всеки момент след това, да съобразим, че

- няма как да се получи $v.d < \delta(s, v)$ заради вече доказания факт, че $v.d \geq \delta(s, v)$ винаги, и
- няма как да се получи $v.d > \delta(s, v)$ след $v.d = \delta(s, v)$, защото RELAX никога не увеличава d -стойността на края на релаксираното ребро. \square

Лема 50: При липса на пътища

Нека v е връх, такъв че няма $s \rightsquigarrow v$. Тогава, след произволна поредица от RELAX върху ребрата на графа е вярно, че $v.d = \infty$.

Доказателство: Съгласно Лема 49, $v.d \geq \delta(s, v)$, след произволна поредица от релаксации на ребрата. Но $\delta(s, v) = \infty$ съгласно Определение 58. \square

Лема 51: Конвергенция

Нека p е най-къс път $s \rightsquigarrow v$ и $u \neq v$ е предпоследният връх в p . Ако $u.d = \delta(s, u)$ в някакъв момент преди релаксирането на реброто (u, v) , то $v.d = \delta(s, v)$ във всеки момент след това релаксиране.

Доказателство: Съгласно Лема 49, ако веднъж се постигне равенство $u.d = \delta(s, u)$, то се запазва нататък. В частност, след релаксирането на (u, v) , в сила е

$$\begin{aligned} v.d &\leq u.d + w((u, v)) \quad // \text{ очевидно от кода на RELAX} \\ &= \delta(s, u) + w((u, v)) \quad // \text{ както вече казахме} \\ &= \delta(s, v) \quad // \text{ Теорема 39} \end{aligned}$$

Щом $v.d \leq u.d + w((u, v))$ и $v.d \geq u.d + w((u, v))$ (Лема 49), то $v.d = u.d + w((u, v))$ и това равенство остава в сила до края (пак Лема 49). \square

12.2 Алгоритъмът на Dijkstra в два варианта

Ще разгледаме един от най-популярните алгоритми за намиране на най-къс нът в ориентиран граф: алгоритъмът на Dijkstra. Нека тегловната функция е $w : E \rightarrow \mathbb{R}^+$. Алгоритъмът на Dijkstra не работи коректно, ако отрицателни тегла, дори да няма отрицателни цикли – намерете сами малък пример за тегловен граф с точно едно отрицателно тегло, без отрицателни цикли, и стартов връх, върху който алгоритъмът на Dijkstra “бърка”. Ще разгледаме два варианта на алгоритъма на Dijkstra, базов и изтънчен[†].

SSHP DIJKSTRA1(G : ориентиран тегловен граф, w : тегловна функция върху G , s : стартов връх)

- 1 ISS(G, s)
- 2 $S \leftarrow \emptyset$
- 3 **while** $\exists u \in V \setminus S : u.d < \infty$ **do**
- 4 избери $x \in V \setminus S$, такъв че $x.d$ е минимална
- 5 $S \leftarrow S \cup \{x\}$
- 6 **foreach** $y \in \text{adj}[x]$
- 7 RELAX((x, y))

SSHP DIJKSTRA2(G : ориентиран тегловен граф, w : тегловна функция върху G , s : стартов връх)

- 1 ISS(G, s)
- 2 $S \leftarrow \emptyset$
- 3 създай празна приоритетна min опашка Q
- 4 $Q \leftarrow V$
- 5 **while** **not** isempty(Q) **do**
- 6 $x \leftarrow \text{EXTRACT-MIN}(Q)$
- 7 $S \leftarrow S \cup \{x\}$

[†]Забележете аналогията с алгоритъма на Prim, който също беше имплементиран в два варианта, със същата разлика между тях.

```

8      foreach  $y \in \text{adj}[x]$ 
9          RELAX(( $x, y$ ))

```

Коректност и сложност. Въпреки повърхностната прилика между алгоритъма на Prim и алгоритъма на Dijkstra, вторият има значително по-трудно доказателство за коректност, защото в неговия контекст няма аналог на Теорема 36. Ще докажем коректността само по отношение на d -стойностите. Приемаме, че от това доказателство лесно следва и коректността на π -стойностите, които реализират дървото.

Допускаме ОО, че целият G е достигим от s . “SSShP DIJKSTRA” означава кой да е от двата варианта на алгоритъма.

Теорема 40: Коректност на алгоритъма на Dijkstra

За всеки ориентиран граф G , за всяка тегловна функция w над него, за всеки избор на стартов връх s е вярно, че след терминирането на SSShP DIJKSTRA(G, w, i):

$$v.d = \delta(s, v), \forall v \in V(G)$$

Доказателство: Приемаме за очевидно, че всеки връх на графа “влиза” в S на някоя итерация на **while**-а. Ще докажем, че за всеки връх $v \in V(G)$, в момента, в който v влиза в S е вярно, че $v.d = \delta(s, v)$. Това влече верността на теоремата заради Лема 49.

Да допуснем противното: за поне един $v \in V(G)$ е вярно, че $v.d \neq \delta(s, v)$ в момента, в който v влиза в S . Нещо повече: нека v е **първият** връх, който “влиза” в S с d -стойност, различна от минималното тегло на път от s до него. Очевидно $v \neq s$, защото $s.d = 0$ и, при положителни тегла, $\delta(s, s) = 0$.

Разглеждаме тази итерация, на която в S “влиза” връх v . В началото на въпросната итерация, нека t е моментът точно преди v да “влезе” в S . Щом $v \neq s$, то $S \neq \emptyset$ при тази итерация. По допускане, съществува $s \rightsquigarrow v$. Разглеждаме най-къс $s \rightsquigarrow^p v$. Щом теглата са положителни, такъв има.

В момента t е вярно, че първият връх на p (това е s) е в S , а последният връх (това е v) не е в S . Дефинираме y като най-близкият до s връх в p , който в момента t не е в S . Възможно е $y = v$, възможно е и $y \neq v$. Важното е, че y е добре дефиниран. Нека върхът преди y в p се назова x (възможно е $x = s$). Забележете, че всички върхове в p от s до x включително в момента t са в S , и y е първият връх в p , в посока s към v , който не е в S :

$$p = \underbrace{s \dots \dots \dots x}_{\text{в } S} \underbrace{y}_{\text{не е в } S} \dots \dots \dots \underbrace{v}_{\text{не е в } S}$$

Ще докажем, че $y.d = \delta(s, y)$ в t . Знаем, че $x \in S$ в t и че v е първият връх, влизащ в S , за който d -стойността не е равна на претегленото разстояние от s . Връх x е бил сложен в S на някоя предишна итерация, в никакъв момент $t' < t$. Следва, че в момента t' , $x.d = \delta(s, x)$, и съгласно Лема 49, във всеки момент след t' продължава да е вярно, че $x.d = \delta(s, x)$. Но на итерацията, в която x е сложен в S , реброто (x, y) е било релаксирано – вижте псевдокода. Прилагаме Лема 51 и заключаваме, че

$$y.d = \delta(s, y) \tag{12.4}$$

след това релаксиране, включително и в момента t .

Съгласно Теорема 39, $\delta(s, y) \leq \delta(s, v)$, тъй като теглата са положителни, а s , y и v са наредени в този ред върху най-къс път от s до v . Съгласно Лема 49, $\delta(s, v) \leq v.d$ по всяко време от работата на алгоритъма. Тогава

$$\delta(s, y) \leq \delta(s, v) \leq v.d \quad (12.5)$$

От (12.4) и (12.5) заключаваме, че в момента t :

$$y.d = \delta(s, y) \leq \delta(s, v) \leq v.d \quad (12.6)$$

Ключовото наблюдение е, че и y , и v са върхове извън S в момента t и причината алгоритъмът да избере v , а не y , като връх, който да влезе в S , е една единствена:

$$v.d \leq y.d \quad (12.7)$$

От (12.6) и (12.7) заключаваме, че

$$v.d = y.d \quad (12.8)$$

Но тогава (12.6) всъщност е верига от равенства:

$$y.d = \delta(s, y) = \delta(s, v) = v.d \quad (12.9)$$

и в частност $\delta(s, v) = v.d$. Тогава, съгласно Лема 49, това остава в сила до края на алгоритъма, в противоречие с допускането, че $\delta(s, v) + v.d$ в края. $\not\vdash$ \square

Ако не е вярно, че целият G е достигим от s , Лема 50 казва, че върховете, които не са достигими от s , са с d -стойности ∞ в края на алгоритъма. Имайки предвид Теорема 40, заключаваме, че върховете, които не са достигими от s са точно тези, които са с d -стойности ∞ в края на алгоритъма.

Сложността по време е същата, в асимптотичния смисъл, като на съответните имплементации на Prim. Базовата имплементация има сложност $\Theta(n^2)$ в най-лошия случай, а изтънчната има сложност $\Theta(m \lg n)$, ако Q е имплементирана с двоична пирамида.

12.3 Най-къси пътища в дагове

Ако G е даг, може да използваме по-ефикасен алгоритъм от алгоритъма на Dijkstra за зададената SSSHP.

DAG SSSHP(G : даг, w : тегловна функция върху G , s : стартов връх)

- 1 TOPOSORT(G)
- 2 ISS(G, s)
- 3 **foreach** $x \in V(G)$ в топологичната наредба
- 4 **foreach** $y \in \text{adj}[x]$
- 5 RELAX((x, y))

Ще покажем коректността на този алгоритъм: след терминирането, за всеки връх $v \in V(G)$ е вярно, че $v.d = \delta(s, v)$.

Инвариант

При всяко достигане на ред 3 е вярно, че $x.d = \delta(s, x)$.

Твърдението е вярно за върховете вляво от s : за всеки връх u вляво от s е вярно, че $\delta(s, u) = \infty$, понеже няма път $s \rightsquigarrow u$, а, от друга страна, във всеки момент от работата на алгоритъма е вярно, че $u.d = \infty$ заради Лема 50. ✓

Ще докажем инвариантата със силна индукция по топологичната наредба за върховете вляво от и включително s . Базата е $x = s$. Вярно е, че $\delta(s, s) = 0$ при отсъствието на отрицателни цикли (което следва от отсъствието на каквито и да е цикли), а, от друга страна, $s.d$ става 0 на ред 2 и остава така до края на алгоритъма (Лема 49). ✓

Разглеждаме произволен връх v вляво от s . Ако няма път $s \rightsquigarrow v$, то $\delta(s, v) = \infty$, а, от друга страна, във всеки момент от работата на алгоритъма е вярно, че $v.d = \infty$ заради Лема 50. Да допуснем, че v е достижим от s . Разглеждаме момента t , в който $x = v$. Нека z_1, \dots, z_k са всички родители на v . В топологичното сортиране, те са вляво от v . Тъй като използваме силна индукция, ние допускаме, че в момента t , за всеки връх u вляво от v е вярно, че $u.d = \delta(s, u)$. В частност, в момента t е вярно, че $z_i.d = \delta(s, z_i)$ за $1 \leq i \leq k$. Нещо повече: поне един z_i е достижим от s , иначе v не бил бил достижим от s , така че за поне едно i , $z_i.d < \infty$ в момента t . Забележете, че RELAX е “обработвала” връх v точно k пъти преди момента t , в смисъл че за всеки z_i , когато x е имала стойност z_i , точно веднъж y е имала стойност v . Забележете, че RELAX реализира $v.d = \min_{1 \leq i \leq k} \{z_i.d + w((z_i, v))\}$. Но $\delta(s, v) = \min_{1 \leq i \leq k} \{\delta(s, z_i) + w((z_i, v))\}$ (това е в сила дори някои z_i да не са достижими от s). Показахме, че $v.d = \delta(s, v)$ в момента t .

Показахме, че за всеки връх v , $v.d = \delta(s, v)$ в момента, в който $x = v$. Съгласно Лема 49, това равенство остава в сила до края на алгоритъма. □

Да разгледаме сложността по време. Сложността на този алгоритъм е $\Theta(n + m)$, защото и топологическото сортиране, и вложените **for**-цикъл имат такава сложност по време.

Предимствата на DAG SSSHP пред алгоритъма на Dijkstra върху даг са следните:

- DAG SSSHP е по-бърз, както на практика, така и в асимптотичния смисъл. Сложността $\Theta(n + m)$ е оптимална, в асимптотичния смисъл, и е по-добра от сложностите на DIJKSTRA, както в базовия вариант, така и в изтънчения вариант, дори приоритетната опашка да е реализирана с пирамида на Fibonacci.
- DAG SSSHP работи без проблеми дори ако има отрицателни тегла. В даговете няма цикли поначало, така че няма опасност от отрицателен цикъл. Както се каза, DIJKSTRA не е използваем при отрицателни тегла. BELLMAN-FORD “се справя” с отрицателни тегла, но той е драстично по-бавен от DAG SSSHP.
- DAG SSSHP може да намира и най-дълги пътища с една тривиална модификация: обръщане на посоката на неравенството в RELAX. Забележете, че при даговете е вярно, че най-дълъг път се състои от най-дълги подпътища (аналогът на Теорема 39): ако заменим подпът с по-дълъг подпът, ще получим по-дълъг път без опасност от повтаряне на върхове. За да се убедим в това, достатъчно е да разгледдаме дага в контекста на топологична сортировка. DIJKSTRA не може да бъде трансформиран в алгоритъм за намиране на най-дълги пътища с просто обръщане на посоката на неравенството в RELAX и на \min в \max по отношение на избора на нов връх, който да влезе в S (в изтънчената реализация това би било смяна на типа на приоритетната опашка от \min в \max), дори графът да е даг.

12.4 Алгоритъмът на Bellman-Ford

Този алгоритъм намира най-къси пътища и в графи с отрицателни тегла, стига да няма отрицателни цикли. А ако има поне един отрицателен цикъл, алгоритъмът установява това и връща съответна индикация FALSE. Ако върне TRUE, то отрицателни цикли няма. С малка модификация, алгоритъмът може да връща и някой отрицателен цикъл, ако има отрицателни цикли.

BELLMAN-FORD(G : ориентиран тегловен граф, w : тегловна функция върху G , s : стартов връх)

```

1 ISS( $G, s$ )
2 for  $i \leftarrow 1$  to  $n - 1$ 
3   foreach  $(x, y) \in E(G)$ 
4     RELAX( $((x, y))$ )
5   foreach  $(x, y) \in E(G)$ 
6     if  $y.d > x.d + w((x, y))$ 
7       return FALSE
8   return TRUE

```

Коректност и сложност. Самият алгоритъм е на редове 1–4, което го прави изключително компактен. Кодът на редове 5–7 е проверка за наличие на отрицателен цикъл.

Ще направим повърхностно доказателство за коректност. БОО, нека целият G е достъжим от s . Първо да допуснем, че отрицателни цикли няма. Тогава дърво на най-късите пътища с корен s е добре дефинирано. Нека T е такова дърво. T има височина най-много $n - 1$, което обяснява и “ $n - 1$ ” на ред 2. Инварианта на **for**-цикъла на редове 2–4 е, при всяко достигане на ред 2, за всеки връх $v \in V(G)$, който се намира на ниво $i - 1$ в T (ниво е множеството от върховете на едно и също разстояние, като брой ребра, в T), $v.d = \delta(s, v)$. Базата е при $i = 1$, тоест, $i - 1 = 0$, тоест, нулевото ниво, тоест, само връх s – очевидно твърдението е вярно. В индуктивната стъпка, ако е вярно за върховете от ниво k , прилагаме Лема 51 и заключаваме, че е вярно и за върховете от ниво $k + 1$.

Лесно се вижда, че ако G няма отрицателни цикли, така че най-къси пътища са дефинирани от s до всеки връх, ако продължим да изпълняваме цикъла на редове 3–4 и след $i = n - 1$, няма да има промяна в нито една d -стойност на връх. Иначе казано, нито едно ребро не може да бъде релаксирано с промяна (намаляване) на d -стойността на крайния му връх, понеже се е стигнало до състояние $\forall v \in V(G) : v.d = \delta(s, v)$ и това е окончателно съгласно Лема 49. Това показва и коректността на проверката за отрицателни цикли на редове 5–7: ако поне едно ребро (x, y) е “релаксирамо”, което е същото като булевото условие на ред 6 да е истина, със сигурност има отрицателен цикъл. В такъв случай, колкото и пъти да продължим да изпълняваме редове 3–4 отвъд $i = n - 1$, ще получаваме “подобрения” на d -стойността на някой връх, за който е вярно, че от s до него има път, съдържащ отрицателен цикъл. \square

Сложността по време, очевидно, е $\Theta(n(n+m))$, което означава, че BELLMAN-FORD е кубичен алгоритъм в най-лошия случай.

12.5 Най-къс път за всяка двойка върхове

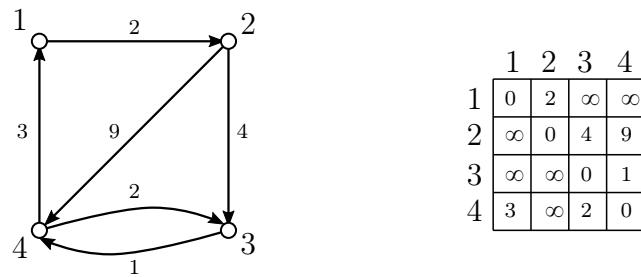
Разглеждаме задачата за най-късите пътища във варианта APShP: за всеки два върха u и v в тегловен ориентиран граф G , търсим дължината на най-къс път $u \rightsquigarrow v$, както и

един такъв път. Това можем да постигнем с алгоритъм за варианта SSShP, като например DIJKSTRA или BELLMAN-FORD (ако има отрицателни тегла), който пускаме от всеки връх. Но има интересен алгоритъм, специализиран за варианта APShP, който си заслужава да бъде разгледан подробно. Това е алгоритъмът на Floyd-Warshall, който, освен че е интересен от теоретична гледна точка, допуска отрицателни тегла и има сложност по време $\Theta(n^3)$, така че е по-бърз от n -кратното пускане на BELLMAN-FORD, което би било със сложност $\Theta(n^4)$.

В тази секция графите се представят не със списъци на съседство, а с матрици от теглата. $G = (\{1, \dots, n\}, E)$ е графът, който разглеждаме, и той е представен с квадратна матрица $n \times n$ от теглата, наречена W , като

$$W[i, j] = \begin{cases} 0, & \text{ако } i = j, \\ \text{теглото на реброто } (i, j), & \text{ако има такова и } i \neq j \\ \infty, & \text{ако няма ребро } (i, j) \text{ и } i \neq j \end{cases}$$

Теглата са произволни реални числа. Ето пример:



Разговорно казано, смисълът на $W[i, j]$ е (при липса на отрицателни цикли) “минималната цена на **директно отиване** от връх i във връх j ”: цената на отиване от i в i е нула, а цената от i в j за $i \neq j$ е, ако няма ребро (i, j) , то тя е безкрайност, иначе е цената на реброто. Става дума именно за директно отиване—без междинни върхове—а не за най-къс път. В примера, директното отиване от връх 2 във връх 4 има цена 9 и затова $W[2, 4] = 9$; това, че има път от 2 до 4 с цена 5 няма значение, тъй като този път съдържа връх 3 като вътрешен връх.

На първо време се фокусираме върху изчисляването на теглата на най-къси пътища. Когато постигнем това, лесно ще добавим код, който изчислява самите пътища. И така, изходът е квадратна матрица $n \times n$, наречена D , като $D[i, j]$ съдържа тегло на най-къс път $i \rightsquigarrow j$.

12.5.1 Алгоритъм, реализиращ матрично умножение

Според [15, стр. 706], този алгоритъм е фолклор. Това означава, че авторството му е неясно и загубено в миналото.

Това е първият алгоритъм по алгоритмичната схема Динамично Програмиране, който ще разгледаме. Тази алгоритмична схема е темата на Лекция 13, но алгоритъмът е за най-къси пътища и мястото му е в тази лекция.

Както всеки алгоритъм по схемата Динамично Програмиране, дизайна на този алгоритъм се състои от следните фази.

1. Измисляне на *характеристика на структурата на оптимално решение*; в нашия случай, на най-къс път.

2. Създаване на подходяща *рекурсивна декомпозиция* на обектите, които искаме да пресмятаме.
3. След това се решава как да се изчисляват ефикасно оптималните решения, но вече не рекурсивно, а отдолу-нагоре (bottom-up).

Първите две фази са по-скоро математически. Третата фаза е изчислителна (алгоритмична). Самият алгоритъм се съставя в третата фаза, но първите две фази са решаващи важни. Ако не разкрием смислено структурата на оптимално решение и/или не намерим смислена рекурсивна декомпозиция на оптималните решения, няма да можем да конструираме алгоритъм. Обектите, които ни интересуват (най-къси пътища в случая) може да бъдат характеризирани по различни начини, които са в основата на различни алгоритми по схемата Динамично Програмиране.

Да направим следното характеризиране на най-къс път $i \rightsquigarrow j$. Всеки път се състои от някакъв брой ребра, не повече от $n - 1$ (ако е прост път). Тогава всеки най-къс път $i \rightsquigarrow j$ има някакъв брой ребра, да кажем t .

// дотук е тривиално...

Променяме леко формулировката: за някое t , всеки най-къс път $i \rightsquigarrow j$ има не повече от t ребра.

// очевидно – е, и?

Продължаваме: всеки най-къс път $i \rightsquigarrow j$, който има не повече от t ребра, се състои от най-къс път $i \rightsquigarrow k$, който има не повече от $t - 1$ ребра, за някой връх k , плюс реброто (k, j) с тегло $W[k, j]$.

// аха, това е!

За това характеризиране на структурата ще направим рекурсивна декомпозиция: за произволни i, j , такива че $1 \leq i, j \leq n$:

- дължината на най-къс път $i \rightsquigarrow j$ с не повече от 0 ребра е 0, ако $i = j$, или ∞ , ако $i \neq j$ (това е базата на рекурсията).
- дължината на най-къс път $i \rightsquigarrow j$ с не повече от t ребра е по-малкото от тези две:
 - ◆ дължината на най-къс път $i \rightsquigarrow j$ с не повече от $t - 1$ ребра
 - ◆ минимума от сумите
 - * дължината на най-къс път $i \rightsquigarrow k$ с не повече от $t - 1$ ребра плюс
 - * теглото $W[k, j]$.
 за $k \in \{1, \dots, n\}$.

Ето схемата за изчисление, базирана на тази рекурсивна декомпозиция. За $t \in \{0, \dots, n - 1\}$, нека $D^{(t)}$ е $n \times n$ матрица от дълчините на най-къси пътищата с не повече от t ребра. Тогава $D^{(t)}[i, j]$ дължината на най-къс път с не повече от t ребра от i до j . Тогава можем да изчисляваме рекурсивно така:

$$D^{(0)}[i, j] = \begin{cases} 0, & \text{ако } i = j \\ \infty, & \text{ако } i \neq j \end{cases}$$

$$D^{(t)}[i, j] = \min \{D^{(t-1)}[i, j], \min \{D^{(t-1)}[i, k] + W[k, j] \mid 1 \leq k \leq n\}\}, \text{ за } t > 0$$

Това може да бъде опростено до

$$D^{(0)}[i, j] = \begin{cases} 0, & \text{ако } i = j \\ \infty, & \text{ако } i \neq j \end{cases} \quad (12.10)$$

$$D^{(t)}[i, j] = \min \{D^{(t-1)}[i, k] + W[k, j] \mid 1 \leq k \leq n\}, \text{ за } t > 0 \quad (12.11)$$

понеже $D^{(t-1)}[i, j]$ се среща в множеството $\{D^{(t-1)}[i, k] + W[k, j] \mid 1 \leq k \leq n\}$. За да се убедим в това, да видим, че при $k = j$, $D^{(t-1)}[i, k] + W[k, j]$ става $D^{(t-1)}[i, j] + W[j, j]$, което е $D^{(t-1)}[i, j]$, понеже $W[j, j] = 0$.

Решението е матрицата $D^{(n-1)}$, тъй като простите пътища не може да имат повече от $n - 1$ ребра.

Изразите (12.10) и (12.11) са на практика **ефективен** рекурсивен алгоритъм: за всеки i и j , $D^{(n-1)}[i, j]$ се пресмята чрез рекурсивни викания на $D^{(n-2)}[i, k]$, за $1 \leq k \leq n$, а стойностите $W[k, j]$ са част от входа. Този алгоритъм обаче е изключително **неефикасен!** Дървото на рекурсията му има разклоненост n и височина $n - 1$, което означава, че сложността му е $\Omega(n^{n-1})$. За каквото и да е реални приложения, такъв алгоритъм е безсмислен.

Но ние можем да направим друг алгоритъм, който работи съгласно (12.10) и (12.11), без ги следва буквально. Стойностите, които трябва да бъдат изчислени общо, не са необятно много. Общо всички матрици са $D^{(0)}, D^{(1)}, \dots, D^{(n-1)}$, тоест, n матрици, всяка с n^2 клетки, така че, съгласно (12.10) и (12.11), трябва да сметнем n^3 стойности. Възниква въпросът, защо има алгоритъм със сложност $\Omega(n^{n-1})$, който пресмята само n^3 неща? Както ще видим в Лекция 13, това се случва, когато **различни** рекурсивни викания пресмятат **една и съща** стойност. Тези викания са напълно отделни и “не знаят” едно за друго, поради което една стойност, веднъж пресметната, се губи и после се пресмята отново и отново.

Примерно, при $n = 300$, пресмятането на $D^{(299)}[1, 300]$ ползва стойностите $D^{(298)}[1, 2], \dots, D^{(298)}[1, 300]$ ($D^{(298)}[1, 1] = 0$). На свой ред, пресмятането на всяка от $D^{(298)}[1, 2], \dots, D^{(298)}[1, 300]$ ползва $D^{(297)}[1, 2], \dots, D^{(297)}[1, 300]$. Това означава, че за да намерим $D^{(298)}[1, 2]$, ние трябва да знаем $D^{(297)}[1, 2], \dots, D^{(297)}[1, 300]$. Дори да допуснем, че сме намерили $D^{(297)}[1, 2], \dots, D^{(297)}[1, 300]$ и после сме сметнали $D^{(298)}[1, 2]$, когато започваме да пресмятаме $D^{(298)}[1, 3]$, ще пресметнем всяко от $D^{(297)}[1, 2], \dots, D^{(297)}[1, 300]$ **отново**, ако следваме рекурсията в (12.11) буквально. Ако това разхитително пресмятане на едно и също нещо в различни върхове на дървото на рекурсията става на всяко ниво, не е чудно, че сложността на алгоритъма се оказва $\Omega(n^{n-1})$.

Печелившата идея е, веднъж пресменати, стойностите да не се губят, а се пазят и при необходимост се вземат в $\Theta(1)$ време. Така можем да направим прилично бърз алгоритъм от (12.10) и (12.11). Той вече не е рекурсивен, а итеративен, защото работи отдолу-нагоре: от $D^{(0)}$ намира $D^{(1)}$, от $D^{(1)}$ намира $D^{(2)}$, и така нататък, и накрая от $D^{(n-2)}$ намира $D^{(n-1)}$, което е решението.

Следният алгоритъм реализира тази идея. “MM” идва от “Matrix Multiplication”.

SHORTEST PATH MM(W : матрица $n \times n$, представляща тегловен ориентиран граф)

```

1  for i ← 1 to n
2      for j ← 1 to n
3          if i = j
4              D(0)[i, j] ← 0
5          else
6              D(0)[i, j] ← ∞
7  for t ← 1 to n - 1
8      for i ← 1 to n
9          for j ← 1 to n
10             D(t)[i, j] ← ∞
11             for k ← 1 to n
12                 D(t)[i, j] ← min {D(t)[i, j], D(t-1)[i, k] + W[k, j]}
13 return D(n-1)
```

Коректността му е очевидна за читател, който/която е убеден/убедена в коректността на (12.10) и (12.11), тъй като той пресмята ефикасно тази рекурсия. Сложността му по време очевидно е $\Theta(n^4)$. След малко ще видим как може да подобрим значително тази сложност. Сложността му по памет привидно е $\Theta(n^3)$, защото за всяко $i \in \{0, \dots, n - 1\}$, матрицата D^i е отделен обект. Всъщност, сложността по памет може лесно да се подобри до $\Theta(n^2)$, като използваме само една матрица и я презаписваме.

Да видим откъде идва “Matrix Multiplication”.

SHORTEST PATH MM реализира абстрактно матрично умножение Да разгледаме отново кода на SHORTEST PATH MM, с намалени отстъпи в началото на редовете, и до него да разгледаме код, реализиращ повдигането на $n \times n$ числена матрица A на $(n - 1)$ -ва степен, започвайки от матрицата-идентитет.

SHORTEST PATH MM(W)

```

for i ← 1 to n
    for j ← 1 to n
        if i = j
            D(0)[i, j] ← 0
        else
            D(0)[i, j] ← ∞
    for t ← 1 to n - 1
        for i ← 1 to n
            for j ← 1 to n
                D(t)[i, j] ← ∞
                for k ← 1 to n
                    D(t)[i, j] ← min(D(t)[i, j],
                                         D(t-1)[i, k] + W[k, j])
    return D(n-1)
```

MATRIX POWER(A)

```

for i ← 1 to n
    for j ← 1 to n
        if i = j
            A(0)[i, j] ← 1
        else
            A(0)[i, j] ← 0
    for t ← 1 to n - 1
        for i ← 1 to n
            for j ← 1 to n
                A(t)[i, j] ← 0
                for k ← 1 to n
                    A(t)[i, j] ← + (A(t)[i, j],
                                         A(t-1)[i, k] • W[k, j])
    return A(n-1)
```

Приликата между двата алгоритъма не е повърхностна, а фундаментална. SHORTEST PATH MM наистина реализира $n - 1$ матрични умножения с входната матрица, започвайки от матрицата-идентитет, само че спрямо полупръстен[†] $\mathbb{R} \cup \{\infty\}$ с операции \min (абстрактно събиране) и събиране (абстрактно умножение), като абстрактната нула (идентитетът на абстрактното събиране) е ∞ , а абстрактната единица (идентитетът на абстрактното умножение) 0. Съответствията са подчертани чрез различни цветове. Лесно може да проверим, че

1. абстрактното събиране \min е асоциативно и комутативно с идентитет ∞ ;
2. абстрактното умножение $+$ е асоциативно и комутативно с идентитет 0;
3. абстрактното умножение е ляво и дясно дистрибутивно спрямо абстрактното събиране

$$\begin{aligned} a + \min(b, c) &= \min((a + b), (a + c)) \\ \min(a, b) + c &= \min((a + c), (b + c)) \end{aligned}$$

4. абстрактното умножение $+$ с абстрактната нула ∞ дава абстрактната нула:

$$a + \infty = \infty + a = \infty$$

[†]В алгебрата, полупръстен е структура, подобна на пръстен, но без изискването за обратен елемент на абстрактното събиране. В този случай абстрактното събиране е операцията \min , която наистина няма обратен елемент.

така че това наистина е полуупръстен. Повече информация за този вид полуупръстени и техните свойства има в [26], Глава 35].

И така, SHORTEST PATH MM реализира матрично умножение, като

$$\begin{aligned} D^{(1)} &= W \\ D^{(2)} &= W^2 \\ &\dots \\ D^{(n-1)} &= W^{n-1} \end{aligned}$$

Подобряване на сложността по време до $\Theta(n^3 \lg n)$ Да кажем, че се налага да извършим умножения на матрица A със себе си, така че да получим A^m . Очевидният начин е да правим така, използвайки $\Theta(m)$ матрични умножения:

$$\begin{aligned} A^2 &\leftarrow A \cdot A \\ A^3 &\leftarrow A^2 \cdot A \\ A^4 &\leftarrow A^3 \cdot A \\ &\dots \\ A^m &\leftarrow A^{m-1} \cdot A \end{aligned}$$

Може обаче да постигнем същия резултат със само $\Theta(\lg m)$ матрични умножения:

$$\begin{aligned} A^2 &\leftarrow A \cdot A \\ A^4 &\leftarrow A^2 \cdot A^2 \\ A^8 &\leftarrow A^4 \cdot A^4 \\ &\dots \\ A^{m/2} &\leftarrow A^{m/4} \cdot A^{m/4} \\ A^m &\leftarrow A^{m/2} \cdot A^{m/2} \end{aligned}$$

Тази техника се нарича *повтаряне на повдигането на квадрат*, на английски, *repeated squaring*. На пръв поглед, това работи толкова чисто и елегантно само когато m е точна степен на двойката; в противен случай, предпоследната матрица ще има в степенния показател число, по-малко от m , а последната, число, по-голямо от m .

Ключовото наблюдение е, че по отношение на повдигането на квадрат на матрицата, съдържаща дължините на най-къси пътища с горна граница върху броя на ребрата, където под “умножение на матрици” разбираме умножението от SHORTEST PATH MM, ако входната матрица (тоест, първата степен) няма отрицателни цикли, то след краен брой умножения със себе си, резултатът ще започне да се повтаря. Причината е, че W^m има смисъл именно на дължини на най-къси пътища с не повече от m ребра. При липса на отрицателни цикли, всеки най-къс път е прост (Наблюдение 40) и съдържа най-много $n - 1$ ребра. Еrgo, може W^{n-2} да е равна или да не е равна на W^{n-1} , но със сигурност

$$W^{n-1} = W^n = W^{n+1} = W^{n+2} = \dots$$

Оттук следва, че ако искаме да получим W^{n-1} с повтаряне на повдигането на квадрат, няма да събъркаме, ако получим някоя от W^n , W^{n+1} , W^{n+2} и така нататък, защото всички те са равни на W^{n-1} .

И така, търсим най-малката точна степен на двойката, по-голяма или равна на $n - 1$, за да я сложим в степенния показател на последната матрица от серията. Тази степен на двойката е $2^{\lceil \lg(n-1) \rceil}$. Сега ще обосновем това. Тръгвайки от очевидните неравенства

$$\begin{aligned} \lceil p \rceil &\geq p, \quad \forall p \in \mathbb{R} \\ \lceil p \rceil - 1 &< p, \quad \forall p \in \mathbb{R} \end{aligned}$$

имаме за $x > 0$

$$\begin{aligned} \lceil \lg x \rceil \geq \lg x &\leftrightarrow 2^{\lceil \lg x \rceil} \geq 2^{\lg x} = x \leftrightarrow 2^{\lceil \lg x \rceil} \geq x \\ \lceil \lg x \rceil - 1 < \lg x &\leftrightarrow 2^{\lceil \lg x \rceil - 1} < 2^{\lg x} = x \leftrightarrow 2^{\lceil \lg x \rceil - 1} < x \end{aligned}$$

Замествайки x с $n - 1$, получаваме

$$\begin{aligned} 2^{\lceil \lg(n-1) \rceil} &\geq n - 1 \\ 2^{\lceil \lg(n-1) \rceil - 1} &< n - 1 \end{aligned}$$

И така, търсената степен, на която е необходимо и достатъчно да повдигнем W , за да получим желания резултат, е $2^{\lceil \lg(n-1) \rceil}$.

Всичко това ни позволява да подобрим сложността на намирането на най-къси пътища чрез матрично умножение до $\Theta(n^3 \lg n)$. Разбира се, ако искаме ефикасна реализация, няма да започваме от единичната матрица, за да получим от нея W , ние W я имаме във входа. В псевдокода на SHORTEST PATH MM на стр. 354 започнахме от единичната матрица $D^{(0)}$ от съображения за елегантност. Тук започваме от W от съображения за ефикасност. Друга малка оптимизация, която може да се направи, е да не инициализираме $D^{(t)}[i, j]$ (която сега наричаме $tmp[i, j]$) с ∞ , а директно с $W[i, 1] + W[1, j]$, и да започнем най-вътрешния цикъл от $k = 2$; отново предпочтате ефикасността пред теоретичната елегантност.

SHORTEST PATH MM, REPEATED SQUARING(W)

```

1   for t ← 1 to  $\lceil \lg(n-1) \rceil$ 
2     for i ← 1 to n
3       for j ← 1 to n
4         tmp[i, j] ← W[i, 1] + W[1, j]
5         for k ← 2 to n
6           tmp[i, j] ← min(tmp[i, j], W[i, k] + W[k, j])
7     W ← tmp
8   return W

```

Инвариантът за най-външния цикъл е, всеки път, когато изпълнението е на ред 1, ако \hat{W} означава входната матрица, то $W = \hat{W}^{2^{i-1}}$ за текущата W . Наистина, при първото достигане $i = 1$ и очевидно $W = \hat{W}^{2^{1-1}} = \hat{W}^{2^0} = \hat{W}^1$ в този момент; при всяко следващото достигане степенният показател очевидно се удвоава; при последното достигане имаме $i = \lceil \lg(n-1) \rceil + 1$, така че $W = \hat{W}^{2^{\lceil \lg(n-1) \rceil + 1 - 1}} = \hat{W}^{2^{\lceil \lg(n-1) \rceil}}$ в този момент, и алгоритъмът връща тази матрица.

Сложността по време очевидно е $\Theta(n^3 \lg n)$, понеже външният цикъл се изпълнява $\Theta(\lg n)$ пъти.

12.5.2 Алгоритъмът на Floyd-Warshall

Да характеризираме най-къс път $i \rightsquigarrow j$ по друг начин. Всеки път има множество от вътрешни върхове (което може да е и празното множество, това няма значение). Върховете на графа

са $1, \dots, n$. Тогава всеки най-къс път $i \rightsquigarrow j$ има множество от вътрешни върхове, което може да е и празното множество. // дотук е тривиално...

Тогава за всеки най-къс път $i \rightsquigarrow^p j$ има число $k \in \{0, \dots, n\}$, такова че множеството от вътрешните върхове е подмножество на $\{1, \dots, k\}$; очевидно $k = 0$ показва, че множеството от вътрешните върхове е празното множество // e, и?

Продължаваме: за всеки най-къс път $i \rightsquigarrow^p j$, множеството от вътрешните върхове на който е подмножество на $\{1, \dots, k\}$, р или съдържа k като междинен връх, или не го съдържа. В първия случай p се състои от най-къс път $i \rightsquigarrow^q k$, "слепен" в общия край k с най-къс път $k \rightsquigarrow^t j$, като q и t имат вътрешни върхове от множеството $\{1, \dots, k - 1\}$. Във втория случай множеството от вътрешните върхове на p е подмножество на $\{1, \dots, k - 1\}$. // аха, това е!

За това характеризиране на структурата ще направим рекурсивна декомпозиция. За произволни i, j , такива че $1 \leq i, j \leq n$, е вярно следното.

- Дължината на най-къс път $i \rightsquigarrow j$, чието множество от вътрешни върхове е празното множество, е или $w((i, j))$, ако $(i, j) \in E(G)$, или ∞ , ако $(i, j) \notin E(G)$. С други думи, ако няма вътрешни върхове, то дължината на най-къс път $i \rightsquigarrow j$ е точно $W[i, j]$.
- Нека $i \rightsquigarrow^p j$ с минимално тегло и вътрешните върхове на p са от $\{1, \dots, k\}$.
 - ◆ p може да съдържа връх $k \geq 1$ като вътрешен връх. Тогава $w(p)$ е сумата от дължината на най-къс път $i \rightsquigarrow k$, чиито вътрешни върхове са от $\{1, \dots, k - 1\}$, и дължината на най-къс път $k \rightsquigarrow j$, чиито вътрешни върхове са от $\{1, \dots, k - 1\}$. В този случай казваме, че k помага.
 - ◆ p може да не съдържа връх $k \geq 1$ като вътрешен връх. Тогава $w(p)$ е равна на дължината на най-къс път $i \rightsquigarrow j$, чиито вътрешни върхове са от $\{1, \dots, k - 1\}$. В този случай казваме, че k не помага.

Тогава можем да изчисляваме рекурсивно така:

$$D^{(0)} \leftarrow W \quad (12.12)$$

$$D^{(k)}[i, j] \leftarrow \min \underbrace{\{D^{(k-1)}[i, j], D^{(k-1)}[i, k] + D^{(k-1)}[k, j]\}}_{\text{k не помага}}, \text{ за } k > 0 \quad (12.13)$$

Забележете алгоритмичното предимство на (12.12) и (12.13) пред (12.10) и (12.11)! В (12.12) и (12.13) се пресмята минимум на 2 числа, докато в (12.10) и (12.11) се пресмята минимум на n числа.

Естествено, ако имплементираме (12.12) и (12.13) директно, ще получим алгоритъм с експоненциална сложност по време, защото при разклоненост 2 и височина $\Theta(n)$ на дървото на рекурсията сложността по време е $\Omega(2^n)$. Но, също както при предишния алгоритъм, ние няма да имплементираме (12.12) и (12.13) директно, а ще съобразим, че всъщност има само $\Theta(n^3)$ различни стойности, които се налага да изчислим, които са разпределени в матрици $D^{(0)}$, $D^{(1)}, \dots, D^{(n)}$, всяка от които е $n \times n$ и се пресмята от предишната. Тези матрици ще запълваме отдолу-нагоре, точно както диктува схемата Динамично Програмиране.

Всичко това води до следния кубичен алгоритъм за задачата APShP, който е публикуван от Robert Floyd [18], но се базира на теорема на Stephen Warshall [70].

FLOYD-WARSHALL(W : матрица $n \times n$, представляща тегловен ориентиран граф)

```

1    $D^{(0)} \leftarrow W$ 
2   for  $k \leftarrow 1$  to  $n$ 
```

```

3   for i ← 1 to n
4       for j ← 1 to n
5           D(k)[i, j] ← min(D(k-1)[i, j], D(k-1)[i, k] + D(k-1)[k, j])
6   return D(n)

```

Коректност и сложност по време и памет. Коректността на FLOYD-WARSHALL е очевидна, ако читателят е убеден в коректността на (12.12) и (12.13). Сложността му по време е $\Theta(n^3)$. Сложността му по памет, ако бъде реализиран по точно този начин, е $\Theta(n^3)$, но може лесно да бъде подобрена до $\Theta(n^2)$, избягвайки създаването на $n + 1$ матрици и използвайки само две матрици, една, от която четем стойности (тя отговаря на $D^{(k-1)}$) и една, в която пишем (тя отговаря на $D^{(k)}$).

Заслужава да се отбележи, че всъщност може да минем със само една матрица, тоест, можем да пишем директно върху матрицата, от която четем, и алгоритъмът ще работи коректно. Това е задача 25.2-4 от [15, стр. 699]. Да видим защо е така. Да кажем, че използваме само една матрица D , която се инициализира с W . Същността на алгоритъма е

$$D[i, j] \leftarrow \min(D[i, j], D[i, k] + D[k, j]) \quad (12.14)$$

Ако $k \neq i$ и $k \neq j$, няма причина това присвояване да води до некоректно изчисление на новото $D[i, j]$. Да кажем БОО, че $k = i$. Тогава (12.14) става

$$D[i, j] \leftarrow \min(D[i, j], D[i, i] + D[i, j]) \quad (12.15)$$

Но $D[i, i]$ е задължително 0 при отсъствие на отрицателни цикли, така че (12.16) става

$$D[i, j] \leftarrow \min(D[i, j], D[i, j]) \quad (12.16)$$

което е същото като $D[i, j]$ да не се променя. Но това е коректно, защото дължината на най-къс път $i \rightsquigarrow j$, съдържащ вътрешен връх не по-голям от i (в момента сме допуснали, че $k = i$), е равна на дължината на най-къс път $i \rightsquigarrow j$, съдържащ вътрешен връх не по-голям от $i - 1$, защото пътищата са задължително прости (Наблюдение 40) и връх i вече се появява като начало на пътя, ерго, не може да е вътрешен връх. Но $D[i, j]$ преди изпълнението на (12.15) е именно дължината на най-къс път $i \rightsquigarrow j$, съдържащ вътрешен връх не по-голям от $i - 1$. С което решаваме задача 25.2-4 от [15, стр. 699].

И още нещо за сложността по памет. Ако в началото на алгоритъма не копираме W в D , а работим **директно върху W** , ще реализираме алгоритъм с **константна сложност по памет**, иначе казано, in-place версия на **Floyd-Warshall**. Да си припомним (Подсекция 2.2.4), че при изследването на сложността по памет отчитаме само допълнителната памет, а входа игнорираме. Така че лесно може да направим версия на **Floyd-Warshall** със сложност по памет $\Theta(1)$, при условие, че сме готови да загубим входа, който ще се окаже презаписан.

Намирането на самите пътища. Дотук разглеждахме алгоритмите SHORTEST PATH MM и FLOYD-WARSHALL за APShP, но само за намиране на дълчините на пътищата. Сега ще видим как да изчисляваме и самите пътища. Очевидно във варианта APShP няма да минем с $\Theta(n)$ памет за описание на пътищата, както беше във варианта SSShP, а ще ни трябва $\Theta(n^2)$ памет, за да опишем най-къси пътища по отношение на всички двойки върхове. А именно, ще използваме матрица $n \times n$, наречена “ Π ” в [15, стр. 695], със следния смисъл на елементите: $\Pi[i, j]$ съдържа предшественика на връх j по най-къс път от връх i . Тъй като целият i -ти ред на Π описва най-къси пътища от връх i , можем да мислим за Π като за n масива на предшествие, каквито имахме в SSShP варианта, разположени в матрица. Иначе казано, i -ият ред на Π описва дърво на най-къси пътища с корен връх i .

Всъщност, алгоритъмът конструира редица от матрици $\Pi^{(0)}, \Pi^{(1)}, \dots, \Pi^{(n)}$, по една за всяка стойност на k в FLOYD-WARSHALL, а Π е последната от тези матрици, тоест $\Pi = \Pi^{(n)}$. Смисълът на елементите е следният: $\Pi^{(k)}[i, j]$ съдържа предшественика на връх j по най-къс път от връх i , който най-къс път има вътрешни върхове от $\{1, \dots, k\}$. Ще опишем тези матрици рекурсивно.

$$\begin{aligned}\Pi^{(0)}[i, j] &= \begin{cases} \text{Nil}, & \text{ако } i = j \text{ или } W[i, j] = \infty \\ i, & \text{ако } i \neq j \text{ и } W[i, j] < \infty \end{cases} \\ \Pi^{(k)}[i, j] &= \begin{cases} \Pi^{(k-1)}[i, j], & \text{ако } D^{(k-1)}[i, j] \leq D^{(k-1)}[i, k] + D^{(k-1)}[k, j] \quad // k \text{ не помага} \\ \Pi^{(k-1)}[k, j], & \text{ако } D^{(k-1)}[i, j] > D^{(k-1)}[i, k] + D^{(k-1)}[k, j] \quad // k \text{ помага} \end{cases}, \text{ за } k > 0\end{aligned}$$

Следва псевдокод на FLOYD-WARSHALL, който пресмята и пътища.

FLOYD-WARSHALL PATHS(W : матрица $n \times n$, представяща тегловен ориентиран граф)

```

1  for i ← 1 to n
2    for j ← 1 to n
3      D(0)[i, j] ← W[i, j]
4      if i = j or W[i, j] = ∞
5          Π(0)[i, j] ← Nil
6      else
7          Π(0)[i, j] ← i
8  for k ← 1 to n
9    for i ← 1 to n
10       for j ← 1 to n
11         if D(k-1)[i, j] ≤ D(k-1)[i, k] + D(k-1)[k, j]
12             D(k)[i, j] ← D(k-1)[i, j]
13             Π(k)[i, j] ← Π(k-1)[i, j]
14         else
15             D(k)[i, j] ← D(k-1)[i, k] + D(k-1)[k, j]
16             Π(k)[i, j] ← Π(k-1)[k, j]
17 return (D(n), Π(n))

```

След като веднъж разполагаме с матрицата $\Pi = \Pi^{(n)}$, ето как може да получим най-късия път от i до j , който тя описва, ако има такъв, или информация, че път от i до j няма.

PRINT APSHP(Π, i, j)

```

1  if i = j
2    print i
3  else
4    if Π[i, j] = Nil
5        print no path
6    else
7        PRINT APSHP( $\Pi, i, \Pi[i, j]$ )
8        print j

```

Пример за работата на FLOYD-WARSHALL PATHS има в [15, стр. 696]. Там са показани всички матрици $D^{(0)}, \Pi^{(0)}, \dots, D^{(n)}, \Pi^{(n)}$, които се получават при работата на алгоритъма върху графа на стр. 690.

Лекция 13

Динамично програмиране.

Резюме: Започваме с прости примери: задачата на опашката пред касата и пресмятане на числа на Fibonacci. Разглеждаме основата на алгоритмичната схема Динамично Програмиране, след което разглеждаме множество примери за нейното успешно прилагане: MATRIX-CHAIN MULTIPLICATION, MINIMUM TRIANGULATION, задачата за неасоциативното умножение, SET PARTITION, 2 EQUAL SUM SUBSETS, KNAPSACK, 2 EQUAL SUM SUBSETS, INTERVAL SCHEDULING, LONGEST COMMON SUBSEQUENCE, MINIMUM VERTEX SEPARATION ON TREES, MINIMUM DOMINATING SET ON TREES, пресмятане на биномни коефициенти и числа на Stirling. За някои от тези задачи разглеждаме ефикасни решения, основани на алчната схема. Споменаваме мемоизацията като алтернативен подход на динамичното програмиране. Накрая стигаме до ограниченията на динамичното програмиране: неефикасен алгоритъм за NP-пълната задача за намиране на най-дълъг път в графи.

13.1 Прости примери

13.1.1 Задачата за опашката пред касата

Авторът на записките научи тази задача от професор Красимир Манев. Дадена е каса за билети. Пред касата има опашка от n человека h_1, h_2, \dots, h_n , в този ред. Известно е, че h_i би се забавил(а) d_i минути на касата, пазарувайки билетите си. Известно е, че всеки двама души h_i и h_{i+1} (очевидно съседи в опашката) може да се комбинират и да си напазаруват билетите заедно, като това ще отнеме c_i минути. При дадени времена на индивидуално пазаруване на билети d_1, d_2, \dots, d_n и времена на комбинирано пазаруване c_1, c_2, \dots, c_{n-1} , да се пресметне минималното време, за което цялата опашка може да си купи билетите.

Ако нямаше възможност за комбиниране на съседи в опашката, отговорът би бил просто $\sum_{i=1}^n d_i$. Но при дадените възможности за комбиниране може да има по-бърз начин – примерно, ако $c_1 < d_1 + d_2$, има смисъл h_1 и h_2 да се комбинират, вместо да минават поотделно. Забележете, че не всички комбинирания са възможни: примерно, ако h_1 и h_2 са комбина и минат заедно през касата, възможността h_2 да се комбинира с h_3 отпада; и обратно, ако h_2 и h_3 са комбина, h_1 трябва да мине през касата сам(а). Следователно, ако $c_1 < d_1 + d_2$ и $c_2 < d_2 + d_3$, можем веднага да кажем, че никакво комбиниране сред първите трима души трябва да има, но *a priori* не е ясно какво: дали h_1 с h_2 , или h_2 с h_3 .

Задачата има решение с “груба сила”. Забелязваме, че всяко решение се определя еднозначно от това, кои съседи в опашката се комбинират; останалите хора – тези извън комбините – минават през касата индивидуално с дадените си индивидуални времена. С други думи, може да дефинираме всяко решение чрез булев (характеристичен) вектор с дължина $n - 1$, да ка-

жем $B[1, \dots, n - 1]$, като $B[i] = 1$ означава, че h_i се комбинира с h_{i+1} , а $B[i] = 0$ означава, че h_i не се комбинира с h_{i+1} . Този вектор не може да има съседни единици, тъй като всеки човек от опашката, който има избор от две комбинации (това са всички хора без първия и последния), може да участва в най-много една от тях. Следователно, всички възможни комбинирания не са 2^{n-1} , а са толкова, колкото са булевите вектори с дължина $n - 1$ без съседни единици. Известно е, че тези вектори са F_{n+1} на брой, където F_n е n -тото число на Fibonacci.

Допълнение 36: Колко са булевите вектори без съседни единици

Нека T_n е броят на всички булеви вектори с дължина n , в които няма съседни единици. Ще покажем, че $T_n = F_{n+2}$ за всяко $n \geq 1$. Очевидно $T_1 = 2 = F_3$, а $T_2 = 3 = F_4$, защото от четирите булеви вектора с дължина 2, точно 11 не отговаря на условието.

За $n > 2$, съобразяваме, че всеки такъв вектор може да започва с единица, но тогава вторият му елемент задължително е нула и следва булев вектор с дължина $n - 2$ без съседни единици, или да започва с нула, като след нея има булев вектор с дължина $n - 1$ без съседни единици. По принципа на разбиването, $T_n = T_{n-1} + T_{n-2}$. Доказвахме, че $T_n = F_{n+2}$ за всяко $n \geq 1$.

Тъй като $F_n \asymp \phi^n$, където $\phi = \frac{1+\sqrt{5}}{2} \approx 1.6$, решението с груба сила е експоненциален алгоритъм, само че по отношение на експонента, чиято основа е по-малка от 2. Въпреки че основата е по-малка от 2, все пак е число, по-голямо от 1, което прави подходът неефикасен на практика.

Ефикасно решение може да се получи със следните съображения. Да си представим, че разглеждаме последователно редиците

$$\begin{aligned}s_1 &= \langle h_1 \rangle \\s_2 &= \langle h_1, h_2 \rangle \\s_3 &= \langle h_1, h_2, h_3 \rangle \\&\dots \\s_n &= \langle h_1, h_2, \dots, h_n \rangle\end{aligned}$$

и се опитваме да получим оптимално решение за s_k чрез оптимални решения за $s_{k-1}, s_{k-2}, \dots, s_1$. Ако успеем в това, то оптималното решение за s_n ще е и решение на задачата.

Решението за s_1 е просто d_1 , защото един човек не може да се комбинира с никого. Решението за s_2 е $\min\{d_1 + d_2, c_1\}$, защото или h_1 и h_2 минават индивидуално за време $d_1 + d_2$, или се комбинират и минават за време c_1 , а ниеискаме минимума от тези времена.

Ключово е ефикасното пресмятане на решението за s_3 . Ако кажем, че решението за s_3 е $\min\{d_1 + d_2 + d_3, c_1 + d_3, d_1 + c_2\}$, защото

- може и тримата да минат индивидуално за време $d_1 + d_2 + d_3$,
- може първите двама да се комбинират, оставяйки третия сам, което означава време $c_1 + d_3$,
- може първият да мине сам, а вторият и третият да се комбинират, което означава време $d_1 + c_2$

ще съркаме. Този начин на мислене не води до ефикасен алгоритъм. Ако продължим в същия дух, решението за s_4 ще е минимума на 5 числа, за s_5 ще е минимума на 8 числа, за s_6

ще е минимума на 13 числа, и така нататък; за s_k ще търсим минимума на F_{k+1} числа. Еrgo, този начин на мислене дава ефективно решение, което обаче има експоненциална сложност по време и е неефикасно. Ефикасното решение за s_3 идва от едно съвсем просто съображение: третият човек може или да мине индивидуално, или да се комбинира с втория.

- Ако h_3 мине индивидуално, решението е d_3 плюс оптимално решение за s_2 . Когато сме стигнали до s_3 , ние вече имаме оптимално решение за s_2 . Да ли оптималното решение за s_2 е $d_1 + d_2$ или е c_1 , на този етап **не ни интересува**. Ние **вече** сме изчислили кое е по-малко от $d_1 + d_2$ и c_1 , пресмятайки решението за s_2 , така че няма смисъл да го пресмятаме отново в решението за s_3 .
- Ако h_3 се комбинира с h_2 , решението е c_2 плюс оптималното решение за s_1 , което вече имаме.

Ако продъллим по този начин, оптималното решение за s_4 ще е минимума на:

- d_4 плюс оптималното решение за s_3 и
- c_3 плюс оптималното решение за s_2 .

В същия дух, оптималното решение за s_i , $3 \leq i \leq n$, е минимум от само две числа, всяко от които е достъпно във време $\Theta(1)$. Ако искаме само числената стойност на оптималното решение, следният алгоритъм решава задачата.

Нека d и c имат смисълът, с който ги използвахме досега.

QUEUE PROCESSING($d[1, \dots, n]$, $c[1, \dots, n - 1]$)

- 1 създай масив $opt[1, \dots, n]$
- 2 $opt[1] \leftarrow d[1]$
- 3 $opt[2] \leftarrow \min(d[1] + d[2], c[1])$
- 4 **for** $i \leftarrow 3$ **to** n
- 5 $opt[i] \leftarrow \min(d[i] + opt[i - 1], c[i - 1] + opt[i - 2])$
- 6 **return** $opt[n]$

13.1.2 Числата на Fibonacci

Задачата е, по дадено n , да се изчисли F_n , където

$$F_n = \begin{cases} 0, & \text{ако } n = 0, \\ 1, & \text{ако } n = 1, \\ F_{n-1} + F_{n-2}, & \text{ако } n > 1 \end{cases}$$

При големи стойности на n бихме имали проблеми с представянето на F_n , ако ползваме стандартен език за програмиране като C, защото F_n ще препълва отредената му памет. Да игнорираме това допълнително усложнение, което няма общо с основната мисъл тук; или да допуснем, че ползваме библиотека, осигуряваща работа с цели числа с произволна точност.

Ако имплементираме това рекурентно уравнение като рекурсивна програма на C директно, получаваме фрагмент от този вид:

```
unsigned fib(unsigned n){
    if (n == 0)
        return 0;
```

```

if (n == 1)
    return 1;
return
    fib(n-1) + fib(n-2);
}

```

Ако пуснем тази програма с вход 10, изпълнението трае около 0.003 секунди, като резултатът е 55. Ако пуснем тази програма с вход 45, изпълнението трае около 8.937 секунди (това очевидно зависи от машината), като резултатът е 1 134 903 170. Ако пуснем тази програма с вход 47, изпълнението трае около 23.273 секунди (отново, зависи от машината), като резултатът е 2 971 215 073. Явно това е програма с експоненциална сложност по време.

Не е трудно да се досетим защо е така. Изпълнението на `fib(45)` вика първо `fib(44)`; на свой ред, `fib(44)` вика `fib(43)` и `fib(42)`, и така нататък. След излизането от `fib(44)` се прави викане на `fib(43)`, което обаче “не знае”, че тази стойност вече е била пресметната и започва да я смята наново, като вика `fib(42)` и `fib(41)`, и така нататък. Това ненужно повторно изчисление се случва навсякъде в дървото на рекурсията, а не само в корена. За `fib(45)`, началните условия се достигат точно 1 836 311 903 пъти, а редът `fib(n-1) + fib(n-2)`; се достига точно 1 836 311 902 пъти. С други думи, дървото на рекурсията има 1 836 311 903 листа и 1 836 311 902 вътрешни върхове. И всичко това е само за пресмятането на 46 различни стойности F_0, \dots, F_{45} .

Следният фрагмент изчислява числа на Fibonacci, като помни последните досега изчислени стойности и не върши излишни изчисления отново и отново:

```

A[0] = 0;
A[1] = 1;
for (i = 2; i <= n; i++) {
    temp = A[0] + A[1];
    A[0] = A[1];
    A[1] = temp;
}

```

Изпълнението на този фрагмент за вход 47 е светкавично: 0.002 секунди.

13.2 Фундамент

И при пресмятането на минималното време за минаване на опашката през касата, и при пресмятането на числата на Fibonacci срещаме един и същи феномен: за всеки достатъчно голям вход, решението на задачата се свежда до решения върху подзадачи, които обаче **не са независими**, а имат общи подподзадачи.

При задачата с опашката, решението за цялата опашка h_1, \dots, h_n се свежда до решението за подопашките h_1, \dots, h_{n-1} и h_1, \dots, h_{n-2} , съответно отговарящи на възможностите h_n да не се комбинира с h_{n-1} и h_n да се комбинира с h_{n-1} . Това обаче не е достатъчно, за да конструираме алгоритъм по схемата Динамично Програмиране. Само свеждането ни дава алгоритъм по схемата Разделяй-и-Владей, а, както се убедихме, това е алгоритъм с експоненциална сложност по време. Следващата стъпка към конструирането на ефикасен алгоритъм се основава на наблюдението, че решението за подопашката h_1, \dots, h_{n-1} и решението за подопашката h_1, \dots, h_{n-2} **имат общи подрешения**. Това е ключово! Въз основа на това наблюдение лесно може да измислим алгоритъм, който не е рекурсивен, а работи

отдолу-нагоре и съхранява решенията за общите подподзадачи, при което отпада преизчисляването на едни и същи решения отново и отново и отново. В това е същността на схемата Динамично Програмиране.

Допълнение 37: Не всеки Разделяй-и-Владей води до Дин. Прогр.

Има много примери, в които Разделяй-и-Владей не води до ефикасен алгоритъм, изграден по схемата Динамично Програмиране, защото подзадачите от фазата **Разделяй** нямат общи подподзадачи.

Последното е напълно вярно за MERGE SORT, защото сортиранията на двета подмасива нямат нищо общо, но MERGE SORT е ефикасен алгоритъм, така че този пример не е особено удачен. Удачен пример е задачата за местенията на Ханойските кули, ако бъде формулирана по подходящ начин: при дадени n диска да се генерира редицата от минимален брой ходове, която ги премества върху крайния прът. Известно е, че броят на тези ходове е $2^n - 1$. Може би трябва да се подчертава, че задачата не е да се изведе числото $2^n - 1$, а **редицата от ходовете**. Каква е сложността по време? Ако не отчитаме представянето на n , сложността по време не е дефинирана (иначе казано, е безкрайност). Ако n е представено в двоична позиционна бройна система, сложността е двойна експонента. Ако n е представено унарно, сложността е $\Theta(2^n)$, което е единична експонента. Да кажем, че n е представено унарно; примерно, дадени са самите дискове в началото – това на практика е същото като n да е записано в унарна бройна система. Знаем, че задачата за преместването на n диска се свежда до задачата за преместването на $n-1$ диска, но самите премествания на $n-1$ диска трябва да се случат два пъти. Тези две местения на $n-1$ диска нямат обща част в смисъл, че те **и двете** трябва да се случат в своята пълнота и никаква част от първото местене на $n-1$ диска не може да замени част от второто местене. Еrgo, подзадачите нямат общи подподзадачи. И наистина, алгоритъмът не е по схемата Динамично Програмиране, а е чист Разделяй-и-Владей алгоритъм с експоненциална сложност по време, при казаните допускания.

Заслужава да се прочете и мнението на Steve Skiena за схемата Динамично Програмиране [63, стр. 273–274]:

Once you understand it, dynamic programming is probably the easiest algorithm design technique to apply in practice. In fact, I find that dynamic programming algorithms are often easier to reinvent than to try to look up in a book. That said, until you understand dynamic programming, it seems like magic. You must figure out the trick before you can use it.

Dynamic programming is a technique for efficiently implementing a recursive algorithm by storing partial results. The trick is seeing whether the naive recursive algorithm computes the same subproblems over and over and over again. If so, storing the answer for each subproblems in a table to look up instead of recompute can lead to an efficient algorithm. Start with a recursive algorithm or definition. Only once we have a correct recursive algorithm do we worry about speeding it up by using a results matrix.

Dynamic programming is generally the right method for optimization problems on combinatorial objects that have an inherent left to right order among components. Left-to-right objects includes: character strings, rooted trees, polygons, and integer

sequences. Dynamic programming is best learned by carefully studying examples until things start to click.

Последният параграф от цитата заслужава особено внимание. Skiena казва, че Динамично Програмиране е приложимо, когато има някакви обекти, подредени в линейна наредба, и тази наредба не се мени, а е фиксирана. Читателят ще забележи, че това е в сила за всички примери за удачно ползване на Динамично Програмиране, които ще разгледаме. Ако става дума за коренови дървета (за които пише и Skiena), линейната наредба е обхождането на върховете в postorder. Ако става дума за множество, което няма иманентна наредба, наредбата е произволна линейна наредба върху това множество (задачата KNAPSACK, примерно).

И още нещо. Както казва Skiena, трябва значителна практика, за да може дизайнерът на алгоритми да интернализира тази техника и да я прилага успешно.

Допълнение 38: За произхода на “Динамично Програмиране”

Интересен страничен въпрос е, защо тази схема се назава по този начин? Има ли статично програмиране, което да различаваме от динамичното? Какво изобщо означава “програмиране” тук? Отговорът е, че името е избрано от създателя на схемата Richard Bellman през 50-те години на 20 век заради привлекателността на “динамично” и прекалено общия смисъл. Bellman описва произхода на името със следния анекдот [6, стр. 59]:

An interesting question is, “Where did the name, dynamic programming, come from?” The 1950’s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defence, and he actually had a pathological fear and hatred of the word, research. I’m not using the term lightly; I’m using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place, I was interested in planning, in decision-making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, “programming”. I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying – I thought, let’s kill two birds with one stone. Let’s take a word which has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it’s impossible to use the word, dynamic, in the pejorative sense. Try thinking of some combination which will possibly give it a pejorative meaning. It’s impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

Статично програмиране, разбира се, няма, а “програмиране” през 50-те е означавало “систематично планиране”, за разлика от съвременния смисъл.

13.3 Matrix-Chain Multiplication

Започваме с пример. Трябва да извършим следното матрично умножение, изполвайки стандартния алгоритъм за умножение на матрици:

$$\underbrace{\begin{bmatrix} 1 & -2 & 2 & -1 & 3 \\ 4 & 5 & 4 & -1 & -3 \end{bmatrix}}_A \cdot \underbrace{\begin{bmatrix} 12 & -3 \\ -1 & 1 \\ 8 & 0 \\ -6 & 9 \\ 5 & 11 \end{bmatrix}}_B \cdot \underbrace{\begin{bmatrix} 19 & -3 & 0 & 0 & -15 & 8 \\ -17 & -9 & 1 & 5 & 12 & -3 \end{bmatrix}}_C \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_D \quad (13.1)$$

Един възможен начин да извършим това е:

$$\begin{aligned} (13.1) = & \underbrace{\begin{bmatrix} 51 & 19 \\ 66 & -49 \end{bmatrix}}_{(A \cdot B)} \cdot \underbrace{\begin{bmatrix} 19 & -3 & 0 & 0 & -15 & 8 \\ -17 & -9 & 1 & 5 & 12 & -3 \end{bmatrix}}_C \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_D = \\ & \underbrace{\begin{bmatrix} 646 & -324 & 19 & 95 & -537 & 351 \\ 2087 & 243 & -49 & -245 & -1578 & 675 \end{bmatrix}}_{((A \cdot B) \cdot C)} \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_D = \underbrace{\begin{bmatrix} 466 & -17564 \\ 4724 & -19492 \end{bmatrix}}_{(((A \cdot B) \cdot C) \cdot D)} \end{aligned}$$

Да видим колко скаларни умножения са извършени при това матрично умножение. Ако умножим матрици $M_{x \times y} \cdot N_{y \times z}$, ние извършваме $x \cdot y \cdot z$ скаларни умножение. В случая, A е 2×5 , B е 5×2 , така че матричното умножение $(A \cdot B)$ "струва" $2 \cdot 5 \cdot 2 = 20$ скаларни умножения. $(A \cdot B)$ е 2×2 , а C е 2×6 , така че матричното умножение $((A \cdot B) \cdot C)$ "струва" $2 \cdot 2 \cdot 6 = 24$ скаларни умножения. $((A \cdot B) \cdot C)$ е 2×6 , а D е 6×2 , така че матричното умножение $(((A \cdot B) \cdot C) \cdot D)$ "струва" $2 \cdot 6 \cdot 2 = 24$ скаларни умножения. Като цяло, умножението на четирите матрици по този начин "струва" $20 + 24 + 24 = 68$ скаларни умножения.

Друг начин да умножим матриците е

$$\underbrace{\begin{bmatrix} 51 & 19 \\ 66 & -49 \end{bmatrix}}_{(A \cdot B)} \cdot \underbrace{\begin{bmatrix} 30 & -328 \\ -56 & -44 \end{bmatrix}}_{(C \cdot D)} = \underbrace{\begin{bmatrix} 466 & -17564 \\ 4724 & -19492 \end{bmatrix}}_{((A \cdot B) \cdot (C \cdot D))}$$

Крайният резултат е същият. Това не е изненада, понеже умножението на матрици е асоциативно. Но да видим колко скаларни умножения ни струва този начин на умножение на матриците. $(A \cdot B)$ струва, както вече видяхме, 20 скаларни умножение. $(C \cdot D)$ струва $2 \cdot 6 \cdot 2 = 24$ скаларни умножения. $((A \cdot B) \cdot (C \cdot D))$ струва $2 \cdot 2 \cdot 2 = 8$ скаларни умножения. Като цяло, броят на скаларните умножения е $20 + 24 + 8 = 52$. Което е по-малко от 68.

Трети начин да умножим матриците е:

$$\underbrace{\begin{bmatrix} 1 & -2 & 2 & -1 & 3 \\ 4 & 5 & 4 & -1 & -3 \end{bmatrix}}_A \cdot \underbrace{\begin{bmatrix} 279 & -9 & -3 & -15 & -216 & 105 \\ -36 & -6 & 1 & 5 & 27 & -11 \\ 152 & -24 & 0 & 0 & -120 & 64 \\ -267 & -63 & 9 & 45 & 198 & -75 \\ -92 & -114 & 11 & 55 & 57 & 7 \end{bmatrix}}_{(B \cdot C)} \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_D =$$

$$\underbrace{\begin{bmatrix} 646 & -324 & 19 & 95 & -537 & 351 \\ 2087 & 243 & -49 & -245 & -1578 & 675 \end{bmatrix}}_{((A \cdot (B \cdot C)))} \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_D = \underbrace{\begin{bmatrix} 466 & -17564 \\ 4724 & -19492 \end{bmatrix}}_{(((A \cdot (B \cdot C))) \cdot D)}$$

Крайният резултат отново е същият, но да видим колко скаларни умножения са извършени. Ве 5×2 и С е 2×6 , така че $(B \cdot C)$ струва $5 \cdot 2 \cdot 6 = 60$ скаларни умножения. $(A \cdot (B \cdot C))$ струва $2 \cdot 5 \cdot 6 = 60$ скаларни умножения. $((A \cdot (B \cdot C)) \cdot D)$ струва $2 \cdot 6 \cdot 2 = 24$ скаларни умножения. Като цяло, броят на скаларните умножения е $60 + 60 + 24 = 144$.

Броят на скаларните умножения дава добра представа за сложността на умножението на матриците, понеже другата операция, събирането (на скаларните произведения), е по-бърза. Заключаваме, че различните начини за извършване на верижно умножение на матрици може да се различават значително като ефикасност, въпреки че крайният резултат е един и същи заради асоциативността на умножението на матрици. Иначе казано, $((A \cdot B) \cdot C) \cdot D$, $((A \cdot B) \cdot (C \cdot D))$ и $((A \cdot (B \cdot C)) \cdot D)$ са математически неразличими, но алгоритично (като сложност) може да са доста различни. Причината е видима и в дадения пример: сложността на верижното умножение нараства, когато се появяват междуинни матрици с големи размери, като $(B \cdot C)$ в примера. Този феномен се проявява силно, когато матриците-множители са далече от квадратни. Очевидно, ако всички матрици-множители са квадратни, броят на скаларните умножения е един и същи при всеки начин на умножение.

“Начин да бъдат умножени” е същото като “начини за скобуване на веригата”. При четири матрици има пет начина за това скобуване (ние разглеждаме само три тях в примера):

- ① $((A \cdot B) \cdot C) \cdot D$
- ② $((A \cdot (B \cdot C)) \cdot D)$
- ③ $((A \cdot B) \cdot (C \cdot D))$
- ④ $(A \cdot ((B \cdot C) \cdot D))$
- ⑤ $(A \cdot (B \cdot (C \cdot D)))$

Броят на начините да бъде скобуван израз с n множители е C_{n-1} , където C_n означава n -то член на Catalan. Първите няколко числа на Catalan са $C_0 = 1$, $C_1 = 1$, $C_2 = 2$, $C_3 = 5$, $C_4 = 14$. Числата на Catalan растат много бързо, като $C_n \asymp \frac{1}{n\sqrt{n}} 4^n$. Следователно, използването на алгоритъм с груба сила за задачата, която разглеждаме, е безсмислено.

Допълнение 39: За числата на Catalan

Да дефинираме n -тото число на Catalan C_n като броят на *балансираните изрази с $2n$ скоби*. Балансиран израз с $2n$ скоби е стринг, съдържащ точно n отварящи скоби "(" и точно n затварящи скоби ")", като във всеки префикс, броят на отварящите скоби е по-голям или равен на броя на затварящите скоби.

Ето всички 5 балансиирани израза при $n = 3$:

$((()))$

$()((()$

$)()()$

$((())()$

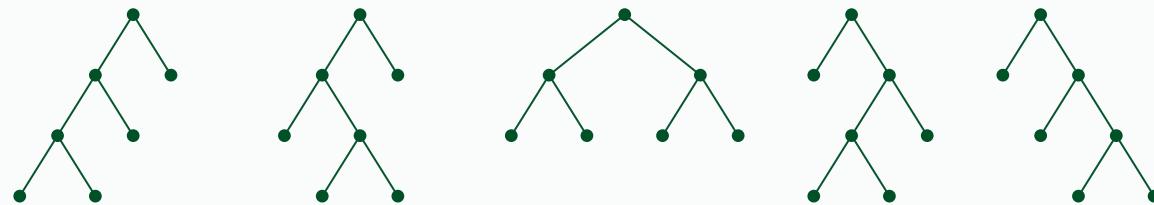
$((()()$

Забележете, че това не е същото като скобуване на израз с n множители. Тук единствените букви са скобите – множители няма.

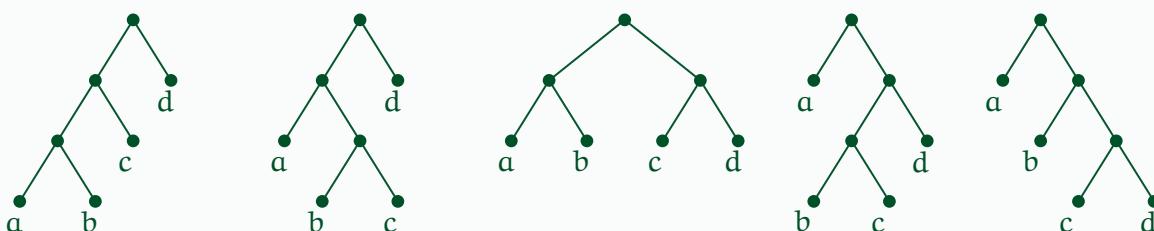
Лема 52

Съществува биекция между скобуваните изрази с $n + 1$ множители и балансираните изрази с $2n$ скоби.

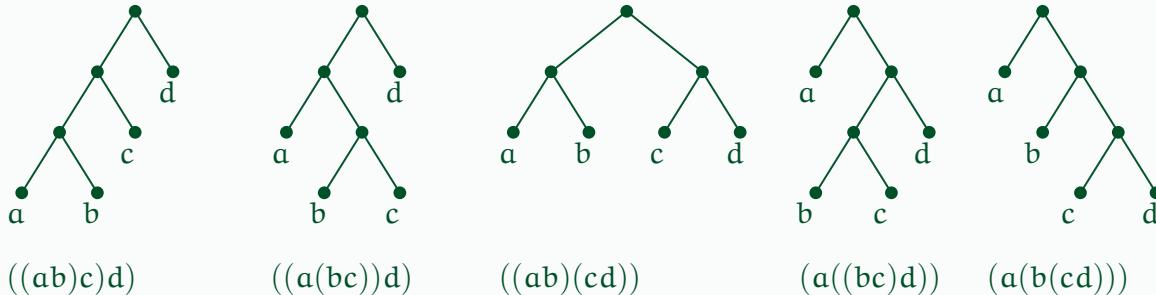
Доказателство. Първо ще покажем с пример, че съществува биекция между скобуваните изрази с $n + 1$ множители и пълните подредени двоични дървета с $n + 1$ листа. Нека $n = 3$. Тогава $n + 1 = 4$ и дърветата с анонимни върхове са тези:



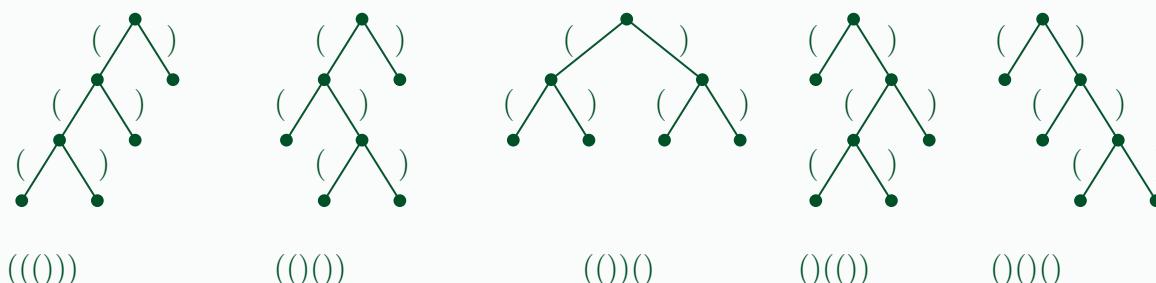
Да сложим имена на листата a , b , c и d , така че при обхождане в preorder (или postorder, няма значение), те се появяват в този ред:



Ето и съответните скобувани изрази:



Показваме с пример биекция между скобуваните изрази с $n + 1$ множители и пълните подредени двоични дървета с $n + 1$ листа. Върху същия пример ще покажем биекция между пълните подредени двоични дървета с $n + 1$ листа и балансираните изрази с $2n$ скоби. Забележете, че при $n + 1$ листа, вътрешните върхове са n на брой, така че общо върховете са $2n + 1$, откъдето следва, че ребрата са точно $2n$. Ще маркираме всяко ребро или с “(”, ако е ребро към ляво дете, или с “)”, ако е ребро към дясно дете. Балансираните изрази са това, което би принтирал DFS, който обхожда дървото и принтира ребрата в реда на откриването, като във всеки вътрешен връх първо отива към лявото дете и след това, към дясното дете.



Това е краят на доказателството на Лема 52. □

Лема 53

Редицата от числата на Catalan е решението на следното рекурентно уравнение:

$$C_n = \begin{cases} 1, & \text{ако } n = 0, \\ \sum_{i=0}^{n-1} C_i \cdot C_{n-1-i}, & \text{ако } n > 0 \end{cases} \quad (13.2)$$

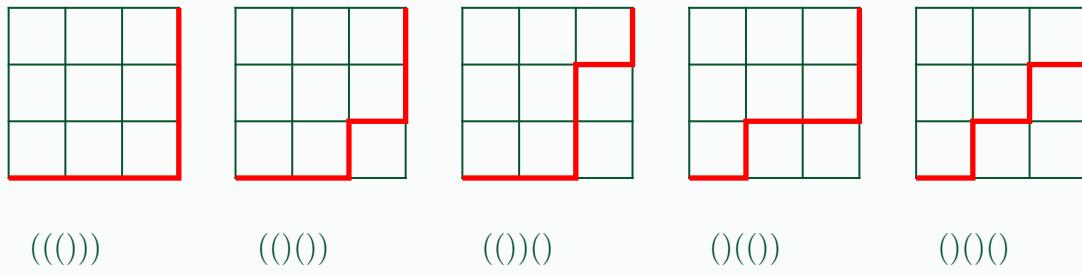
Доказателство. Ще покажем с индукция по n , че (13.2) брои скобуваните изрази с $n + 1$ множители. Но съгласно Лема 52 те са колкото балансираните изрази с $2n$ скоби, откъдето желаният резултат следва веднага.

Ако $n = 0$, (13.2) дава 1, а добре скобуваният израз с $0 + 1$ множителя е само един: (a), ако игнорираме идентичността на множителя. ✓

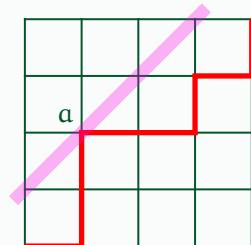
Ако $n \geq 1$, става дума за $n + 1$ множители. В линейната наредба между тях има $(n + 1) - 1 = n$ “празнини”, във всяка от които може да попада последното умножение. Същото нещо, казано в термините на пълните двоични подредени дървета, чиито листа са множителите, е: при фиксирана наредба отляво надясно на листата, има n възможности за това, кой листа са вляво от корена (което определя кой са вдясно от корена). Нека i е номерът на “празнината”, в която е последното умножение, като i взема такива стойности, че вляво от тази празнина има $i + 1$ множителя. Това означава, че вдясно има $(n + 1) - (i + 1) = n - i$ множителя. Очевидно минималната стойност на i е 0 (1 множител вляво), а максималната е $n - 1$ (n множителя вляво). За всяка позиция на празнината, умножаваме броя на скобуванията на подредицата вляво с броя на скобуванията на подредицата вдясно; съгласно индуктивното предположение, те са съответно C_i ($i + 1$ множителя вляво) и C_{n-i} ($n - i$ множителя вдясно). Накратко, за всяко i скобуванията са $C_i \cdot C_{n-i}$. Прилагаме комбинаторния принцип на разбиването, сумирайки по всички i , и получаваме израза от (13.2). \square

Рекурентното уравнение 13.2 не може да бъде решено нито с Мастьр Теоремата, нито с метода с характеристичното уравнение. Ако имаме решение, можем да го докажем по индукция, но как да получим решение?

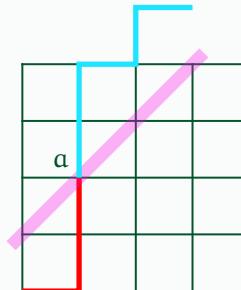
Ще изведем решение на уравнение 13.2 с елементарни комбинаторни съображения. Това уравнение брои балансираните изрази с $2n$ скоби. Ако отпадне изискването да са балансириани и остане само това, че са n скоби от вид $($ и още n скоби от вид $)$, ние знаем колко са тези изрази: $\binom{n+n}{n} = \binom{2n}{n}$. Знаем, че те отговарят биективно на разходки в правоъгълна мрежа $n \times n$, които разходки започват долу вляво и завършват горе вдясно. Нека скобата $($ е ход надясно, а $)$ е ход нагоре. Сега връщаме ограничението изразите да са балансириани. Това означава в нито един префикс броят на $)$ не е по-голям от броя на $($. Иначе казано, никоя разходка не “излиза” над вторичния диагонал от долу вляво до горе вдясно. Ето петте такива разходки при $n = 3$ със съответните балансириани изрази:



Търсим броя на разходките, които никога не излизат над вторичния диагонал, като функция на n . Този брой ще получим, като от $\binom{2n}{n}$ извадим броя на разходките-нарушители – тези, които излизат над вторичния диагонал. Следната фигура показва как броим нарушителите: всяка разходка-нарушител достига до поне една точка от диагонала над вторичния диагонал (в лилаво). От **първата** такава точка до края обръщаме разходката в смисъл, че всеки ход надясно става ход нагоре, и обратно, всеки ход нагоре става под надясно. Това ще наричаме “рефлексия след първата точка”.



разходка-нарушител



рефлексия след a

Очевидно рефлектираната разходка става разходка в правоъгълна мрежа $(n - 1) \times (n + 1)$. Не е трудно да се види, че има биекция между разходките-нарушители в оригиналната $n \times n$ мрежа и всички разходки в новата $(n - 1) \times (n + 1)$ мрежа. Оттук заключаваме, че броят на разходките, които не са нарушители, е

$$\binom{2n}{n} - \binom{n-1+n+1}{n-1} = \frac{(2n)!}{n!n!} - \frac{(2n)!}{(n-1)!(n+1)!} = \frac{(2n)!}{n!(n-1)!} \left(\frac{1}{n} - \frac{1}{n+1} \right) = \\ \frac{(2n)!}{n!(n-1)!} \left(\frac{n+1-n}{n(n+1)} \right) = \frac{(2n)!}{n!n!} \left(\frac{1}{n+1} \right) = \frac{1}{n+1} \binom{2n}{n}$$

Показахме, че $C_n = \frac{1}{n+1} \binom{2n}{n}$. Това е решението на (13.2).

Да видим асимптотиката на C_n . От Теорема 15 следва, че $\binom{2n}{n} \asymp \frac{1}{\sqrt{2n}} 2^{2n}$, тоест $\binom{2n}{n} \asymp \frac{1}{\sqrt{n}} 4^n$. Тогава $C_n \asymp \frac{1}{n\sqrt{n}} 4^n$.

И така, нека е дадено верижно умножение на n матрици

$$A_1 \cdot A_2 \cdot \dots \cdot A_n$$

като A_1 е $d_0 \times d_1$, A_2 е $d_1 \times d_2$, …, A_n е $d_{n-1} \times d_n$. Самите матрици не са дадени, защото тяхното съдържание не ни интересува в тази задача. Интересуваме се от техните размери и търсим начин да бъдат умножени, който минимизира общия брой на скаларните произведения. Задачата може да се формализира така.

Изч. Задача: MATRIX-CHAIN MULTIPLICATION

пример: Редица $\langle d_0, d_1, \dots, d_n \rangle$ от цели положителни числа, която представлява редица от n матрици $\langle A_1, A_2, \dots, A_n \rangle$, като A_i е с размери $d_{i-1} \times d_i$, за $1 \leq i \leq n$. Матриците не са дадени.

решение: Скобуване на редицата $\langle A_0, A_1, \dots, A_n \rangle$, което минимизира броя на скаларните умножения за матричното умножение $A_1 \cdot A_2 \cdot \dots \cdot A_n$.

Има ефикасен алгоритъм за MATRIX-CHAIN MULTIPLICATION, изграден по схемата Динамично Програмиране. На първо време разглеждаме опростена версия на задачата, в която

се иска само минималният брой на скаларните умножения, а самото скобуване (което води до такъв минимален брой скаларни умножения) не се иска.

Първо ще характеризираме оптимально решение на задачата. Всяко оптимально решение има място, на което се извършва последното матрично умножение; в Допълнение 39 използвахме термина “празнина”. Формално казано, има матрици A_k и A_{k+1} , за някое $k \in \{1, \dots, n-1\}$, такова че последното матрично умножение е $B \times C$, където $B = A_1 \cdot A_2 \cdot \dots \cdot A_k$ и $C = A_{k+1} \cdot A_{k+2} \cdot \dots \cdot A_n$. Визуално е по-добре да си представим последното матрично умножение така:

$$\underbrace{(A_1 \cdot A_2 \cdot \dots \cdot A_k)}_B \mid \underbrace{(A_{k+1} \cdot A_{k+2} \cdot \dots \cdot A_n)}_C$$

В контекста на тази характеризация, да направим рекурсивна декомпозиция на оптимально решение. Броят на скаларните умножения за цялото матрично умножение е сумата от следните три събирами:

- броят на скаларните умножения в матричното произведение $A_1 \cdot A_2 \cdot \dots \cdot A_k$, тоест, за конструиране на матрицата B ;
- броят на скаларните умножения в матричното произведение $A_{k+1} \cdot A_{k+2} \cdot \dots \cdot A_n$, тоест, за конструиране на матрицата C ;
- броят на скаларните умножения за матричното умножение $B \cdot C$.

Трябва да разгледаме всички възможни стойности на k , а именно от 1 включително до $n-1$ включително, и да вземем минимума от сумата на трите събирами.

Тъй като дължината на всяка от веригите за умножение $A_1 \cdot A_2 \cdot \dots \cdot A_k$ и $A_{k+1} \cdot A_{k+2} \cdot \dots \cdot A_n$ е по-малка от n , това е истинска рекурсивна декомпозиция, само че е изказана много общо и няма начални условия. Началното условие е много просто: ако веригата от матрици за умножение има дължина 1, тоест, ако $n=1$, решението е 0.

Нека осмислим добре рекурсивната декомпозиция. От написаното човек може да остане с впечатление, че при декомпозицията винаги декомпозираме на две подвериги, всяка от които “опира” в единия край на началната верига; а именно, започва с A_1 или завършва с A_n . Това, разбира се, не е вярно. Ако си представим още едно ниво на декомпозиция (означено на следната фигура със син делител), ясно се вижда, че в общия случай подверигите започват с някаква матрица A_i и завършват с някаква матрица A_j , където $1 \leq i \leq j \leq n$:

$$\underbrace{\left(\underbrace{(A_1 \cdot \dots \cdot A_\ell)}_{B_1} \mid \underbrace{(A_{\ell+1} \cdot \dots \cdot A_k)}_{B_2} \right)}_B \mid \underbrace{\left(\underbrace{(A_{k+1} \cdot \dots \cdot A_m)}_{C_1} \mid \underbrace{(A_{m+1} \cdot \dots \cdot A_n)}_{C_2} \right)}_C$$

Подробната рекурсивна декомпозиция е следната. Нека $M(i, j)$ означава минималния брой скаларни умножения за верижното матрично умножение $A_i \cdot \dots \cdot A_j$, къдено $1 \leq i \leq j \leq n$.

$$M(i, j) = \begin{cases} 0, & \text{ако } i = j, \\ \min \{M(i, k) + M(k+1, j) + d_{i-1} d_k d_j \mid i \leq k \leq j-1\}, & \text{ако } i < j \end{cases} \quad (13.3)$$

$d_{i-1} d_k d_j$ е броят на скаларните умножения в матричното умножение на матрицата-произведение $A_i \cdot \dots \cdot A_k$ (тя е с размери $(i-1) \times k$) и на матрицата-произведение $A_{k+1} \cdot \dots \cdot A_j$ (тя е с размери $k \times j$). Управляващият параметър на тази рекурсия е разликата $j - i$. Началното

условие е за стойност 0 на $j - i$, а в рекурсивната част $M(i, j)$ се пресмята чрез някакви $M(i', j')$, където $j' - i' < j - i$.

Решението е $M(1, n)$.

Ето схемата за изчисление, базирана на тази рекурсивна декомпозиция. Нека M е квадратен масив $n \times n$. Изчислителното правило е пълен аналог на (13.3):

$$M[i, j] = \begin{cases} 0, & \text{ако } i = j, \\ \min_{1 \leq k \leq j-1} (M[i, k] + M[k+1, j] + d_{i-1}d_kd_j), & \text{ако } i < j \end{cases} \quad (13.4)$$

Да съобразим как работи изчислението. Базата на рекурсията е $M[i, i] = 0$, за $1 \leq i \leq n$. С други думи, слагаме нули върху главния диагонал. Въз основа на главния диагонал и изчислителното правило от (13.4) запълваме съседния вдясно диагонал на главния. Въз основа на тези два диагонала и изчислителното правило запълваме следващия диагонал вдясно. И така нататък, като последният диагонал в тази редица от диагонали има един елемент, а именно $M[1, n]$, който е и решението. Забележете, че клетките на масива под главния диагонал не се ползват.

Ето псевдокода на алгоритъм, базиран на тази схема за изчисление.

MATRIX-CHAIN MULTIPLICATION($\langle d_0, d_1, \dots, d_n \rangle$: цели положителни)

```

1   for i ← 1 to n
2       M[i, i] ← 0
3   for diag ← 2 to n
4       for i ← 1 to n - diag + 1
5           j ← i + diag - 1
6           M[i, j] ← M[i, i] + M[i + 1, j] + d_{i-1}d_jd_j
7           for k ← i + 1 to j - 1
8               M[i, j] ← min(M[i, j], M[i, k] + M[k + 1, j] + d_{i-1}d_kd_j)
9   return M[1, n]

```

Да видим пример, взет от [15, стр. 376]. Нека $n = 6$ и $\langle d_0, \dots, d_6 \rangle = \langle 30, 35, 15, 5, 10, 20, 25 \rangle$. Двумерният масив е 6×6 . Клетките под главния диагонал не се ползват. Слагаме нули върху главния диагонал.

Да видим колко е $M[1, 2]$. Съгласно (13.4),

$$\begin{aligned} M[1, 2] &= \min_{1 \leq k \leq 1} (M[i, k] + M[k + 1, j] + d_{i-1}d_kd_j) \\ &= M[1, 1] + M[2, 2] + d_0d_1d_2 = 0 + 0 + 30 \cdot 35 \cdot 15 = 15\,750 \end{aligned}$$

	1	2	3	4	5	6
1	0	15 750				
2		0				
3			0			
4				0		
5					0	
6						0

Аналогично запълваме останалите стойности в диагонала над главния диагонал:

	1	2	3	4	5	6
1	0	15 750				
2		0	2 625			
3			0	750		
4				0	1 000	
5					0	5 000
6						0

Да видим колко е $M[1, 3]$. Съгласно (13.4),

$$\begin{aligned}
 M[1, 3] &= \min_{1 \leq k \leq 2} M[1, k] + M[k + 1, 3] + d_{1-1}d_kd_3 \\
 &= \min(M[1, 1] + M[2, 3] + d_0d_1d_3, M[1, 2] + M[3, 3] + d_0d_2d_3) \\
 &= \min(0 + 2 625 + 5 250, 15 750 + 0 + 2 250) \\
 &= \min(7 875, 18 000) = 7 875
 \end{aligned}$$

Следната фигура показва запълнената клетка $M[1, 3]$, а с цветни линии е показано кои двойки елементи от предишни диагонали участват в минимума.

	1	2	3	4	5	6
1	0	15 750	7 875			
2		0	2 625			
3			0	750		
4				0	1 000	
5					0	5 000
6						0

Да кажем, че главният диагонал е *първият диагонал*, този над него е *вторият диагонал*, а следващият—който запълваме в момента—е *третият диагонал*. Изобщо, номерът на диагонала е номерът на колоната, на която той “излиза” на първия ред. Ето запълването и на третия диагонал.

	1	2	3	4	5	6
1	0	15 750	7 875			
2		0	2 625	4 375		
3			0	750	2 500	
4				0	1 000	3 500
5					0	5 000
6						0

Ето запълването на целия масив. С цветни линии е показано кои двойки елементи от предишни диагонали участват в минимума, с който се изчислява $M[1, 6]$.

	1	2	3	4	5	6
1	0	15 750	7 875	9 375	11 875	15 125
2	0	2 625	4 375	7 125	10 500	
3		0	750	2 500	5 375	
4			0	1 000	3 500	
5				0	5 000	
6					0	

Това е и краят на примера. Да направим няколко пояснения върху псевдокода на MATRIX-CHAIN MULTIPLICATION. **for**-цикълът на редове 3–8 е с управляваща променлива, наречена не случайно “diag”. Тя има смисъл на диагонал в направление горе-ляво – долу-дясно, чийто номер отговаря на номера на колоната, в която този диагонал пресича първия ред. Диагонал 1 е главният диагонал, който запълваме с нули на редове 1–2.

Клетките $M[i, j]$ в диагонал номер $diag$ се отличават с това, че $j = i + diag - 1$. Примерно, за диагонал номер 2, разликата между j и i е 1: той се състои от $M[1, 2], M[2, 3], \dots, M[n-1, n]$. И наистина, на ред 5, на j се присвоява точно $i + diag - 1$.

Ние запълваме всеки диагонал в посока горе-ляво – долу-дясно. Не е задължително да е така. За всяка клетка от даден диагонал, стойността се изчислява чрез стойности, които се вземат само от диагоналите с по-малки номера (както и от d стойностите от входа). Еrgo, можем да запълваме даден диагонал в какъвто ред искаеме.

Тъй като сме избрали да запълваме всеки диагонал в посока горе-ляво – долу-дясно, има смисъл да инициализираме i с 1 и да го инкрементираме, докато стане $n - (diag - 1) = n - diag + 1$, защото $diag - 1$ е броят на редовете под последния ред, съдържащ елемент на диагонал номер $diag$.

Коректността на алгоритъма е очевидна за всеки, който е убеден в коректността на рекурсивната декомпозиция, основаната на нея схема за изчисление и на техническите подробности на кода. Сложността по време е $\Theta(n^3)$.

Ако искаем не само оптималната стойност, но и оптималното скобуване. Ако искаем и скобуване, такова че ако умножаваме матриците съгласно него, постигаме минимален брой скаларни умножения, добавяме масив S с размери $n \times n$ (или може да е $(n-1) \times (n-1)$, защото при него дори главният диагонал не се ползва), който запълваме заедно с M . Клетка $S[i, j]$, $i < j$, има смисъл на индекса k в подверигата от матрични умножения $A_i \cdot \dots \cdot A_j$, $i \leq k \leq j-1$, такъв че оптималното скобуване за тази подверига е

$$(A_i \cdot A_{i+1} \cdot \dots \cdot A_k) | (A_{k+1} \cdot A_{k+2} \cdot \dots \cdot A_j)$$

Псевдокодът става следният.

MATRIX-CHAIN MULTIPLICATION+($\langle d_0, d_1, \dots, d_n \rangle$: цели положителни)

1 **for** $i \leftarrow 1$ **to** n

```

2      M[i, i] ← 0
3  for diag ← 2 to n
4      for i ← 1 to n - diag + 1
5          j ← i + diag - 1
6          M[i, j] ← M[i, i] + M[i + 1, j] + di-1djdj
7          S[i, j] ← i
8          for k ← i + 1 to j - 1
9              if M[i, j] < M[i, k] + M[k + 1, j] + di-1dkdj
10             M[i, j] ← M[i, k] + M[k + 1, j] + di-1dkdj
11             S[i, j] ← k
12 return M[1, n], S

```

Следният рекурсивен алгоритъм генерира описание на оптимално скобуване.

SHOW-ORDER($S[1, \dots, n, 1, \dots, n]$, i, j)

```

1 if i = j
2     print "Ai"
3 else
4     k ← S[i, j]
5     print "("
6     SHOW-ORDER(S, i, k)
7     print "*"
8     SHOW-ORDER(S, k + 1, j)
9     print ")"

```

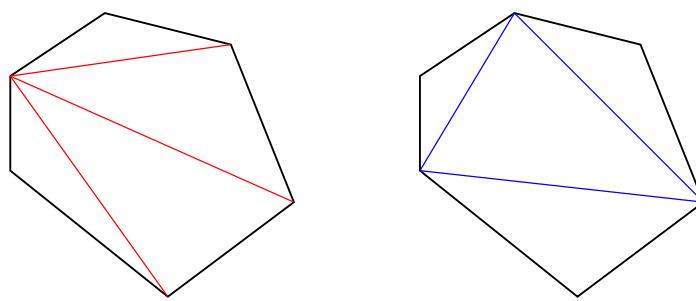
13.4 Minimum Weight Triangulation

Прост многоъгълник е множество от отсечки, поне три на брой, които не се пресичат две по две, освен може би в крайните точки, и чието обединение е приста затворена крива. Краищата на отсечките са *върховете на многоъгълника*. Съседни върхове са всеки два върха, които са краища на една от тези отсечки. Ако n е броят на отсечките (който е равен на броя на върховете), наричаме фигурата *прост n-ъгълник*.

Разглеждаме частен случай на прости многоъгълници, наречени *изпъкнали многоъгълници*. Изпъкнал многоъгълник е прост многоъгълник с вътрешни ъгли, по-малки от 180 градуса. Ако кажем само *n-ъгълник*, разбираме прост изпъкнал *n-ъгълник*.

Диагонал на прост многоъгълник е отсечка с краища върхове на многоъгълника, които не са съседни. При изпъкнатите многоъгълници всички диагонали се съдържат във вътрешността на фигурата.

Триангулация на многоъгълник е множество негови диагонали, които не се пресичат по двойки, освен може би в крайните си точки, и които разбиват вътрешността на фигураната на триъгълници. Триангулирането на многоъгълници е най-важната декомпозиция на многоъгълници с големи приложения в Компютърната геометрия, CAD и други области на Компютърните науки. Ето пример със шестъгълник и две негови различни триангулации.



Тривиално се доказва, че за всеки n -ъгълник, всяка триангулация има $n - 3$ диагонала и $n - 2$ триъгълника. Резултатът очевидно е в сила, ако $n = 3$. За целите на задачата и алгоритъма, който ще разгледаме, има смисъл да разглеждаме индивидуални отсечки като *изродени многоогълници*, двуъгълници в този случай, чито триангулации имат 0 диагонала.

Тегло на триангулация е сумата от дълчините на участващите в нея диагонали[†]. Примерно, на показаната фигура с двете триангулации, червената триангулация има тегло 127.61 mm, а синята, 116.86 mm (измерено в приложението, с което са генериирани).

Ако триангулацията се назава X , нейното тегло се бележи с $|X|$.

Изч. Задача: MINIMUM WEIGHT TRIANGULATION

пример: Редица от точки $\langle V_1, \dots, V_n \rangle$ в равнината, задаваща изпъкнал n -ъгълник P .
решение: Триангулация на P с минимално тегло.

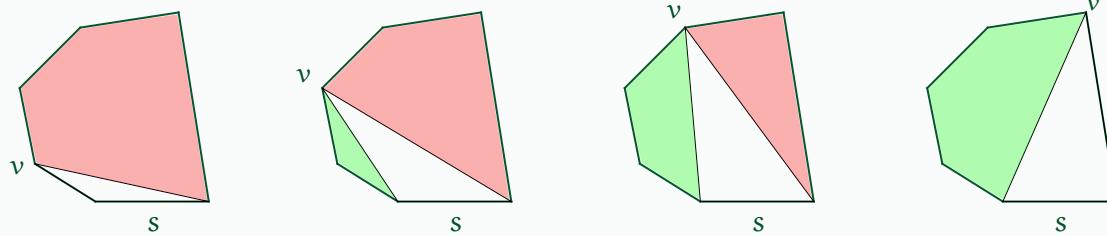
На първо време ще разгледаме по-прости варианти, в който се иска само минимално тегло.

MINIMUM WEIGHT TRIANGULATION има доста общо с MATRIX-CHAIN MULTIPLICATION. Най-малкото, броят на триангулациите на изпъкнал n -ъгълник е C_{n-2} , тоест, $(n - 2)$ -рото число на Catalan; да си припомним, че броят на скобуванията на израз с n множители е C_{n-1} .

Допълнение 40: Броят на триангулациите на изпъкнал n -ъгълник

Да си представим изпъкнал n -ъгълник за $n \geq 3$ и произволна негова триангулация. Да си представим произволна страна s на n -ъгълника. Очевидно s е страна на точно един триъгълник на триангулацията. Този триъгълник се определя еднозначно от s , която задава два от върховете му, и от третия си връх v , който не е връх от s . За v има точно $n - 2$ възможности. На следната фигура, триъгълникът е в бяло за всеки избор на v .

[†]Има и други смислени дефиниции на тегло на триангулация. Може теглото да е минималното лице на триъгълник в триангулацията или минималния ъгъл на триъгълник в триангулацията – и двете имат отношение към по-лесното визуално възприемане. При тези дефиниции на тегло, задачата става максимизационна, но това е дребна разлика.



Всеки избор на трети връх v задава един k -ъгълник (в зелено на фигурата) и един m -ъгълник (в червено на фигурата), които, заедно с триъгълника, представляват разбиване на оригиналния n -ъгълник. Забележете, че $k+m = n+1$, понеже k -ъгълникът и m -ъгълникът имат общ връх v . Минималната стойност на k е 2, при което k -ъгълникът е изроден, а m -ъгълникът е $(n-1)$ -ъгълник. Максималната стойност на k е $n-1$, при което m -ъгълникът е изроден.

За всеки избор на v , броят на триангулациите е произведението от броя на триангулациите на k -ъгълника и броя на триангулациите на m -ъгълника. Предвид това, че $m = n - k + 1$, броят T_n на триангулациите на n -ъгълника е

$$T_n = \begin{cases} 1, & \text{ако } n = 2, \\ \sum_{k=2}^{n-1} T_k \cdot T_{n-k+1}, & \text{ако } n > 2 \end{cases} \quad (13.5)$$

Да сравним (13.5) с (13.2). Веднага се вижда, че $T_n = C_{n-2}$.

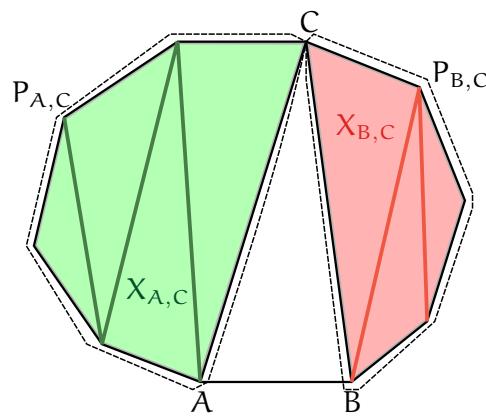
Наблюдение 43: Тегло на триангуляция, декомпозиция

Нека е даден n -ъгълник P за $n \geq 3$ и нека отсечката AB . Нека е дадена произволна триангуляция X на P . Тогава AB е страна в точно един триъгълник спрямо X . Нека C е третият връх на този триъгълник. Нека многоъгълникът, състоящ се от редицата от отсечките от C до A , които не съдържат B , плюс отсечката AC , се нарича $P_{A,C}$. Нека многоъгълникът, състоящ се от редицата от отсечките от C до B , които не съдържат A , плюс отсечката BC , се нарича $P_{B,C}$. Нека $X_{A,C}$ е триангуляцията X , ограничена до $P_{A,C}$. Нека $X_{B,C}$ е триангуляцията X , ограничена до $P_{B,C}$.

Тогава

$$|X| = |AC| + |BC| + |X_{A,C}| + |X_{B,C}| \quad (13.6)$$

Това твърдение е прекалено тривиално, за да го доказваме строго. Ето илюстрация. Имайте предвид, че теглото на триангуляция е сумата от теглата на диагоналите ѝ. Това означава, че отсечките на многоъгълника не участват в тази сума. В този пример, отсечката AC не участва в $|X_{A,C}|$ и, аналогично, BC не участва в $|X_{B,C}|$. Така че (13.6) наистина брои всички диагонали точно по един път.



Нека въходът е редица от точки $\langle V_1, \dots, V_n \rangle$ в равнината, задаваща изпъкнал n -ъгълник P . Фиксираме посока в равнината, да кажем, по часовниковата стрелка. БОО, нека редицата точки $\langle V_1, \dots, V_n \rangle$ се среща в посока по часовниковата стрелка, ако обикаляме по периферията на многоъгълника. За всеки i и j , такива че $1 \leq i < j \leq n$, $P_{i,j}$ е многоъгълникът, състоящ се от отсечките $V_iV_{i+1}, V_{i+1}V_{i+2}, \dots, V_{j-1}V_j$ и отсечката V_iV_j , която може да е диагонал на многоъгълника. Очевидно $P_{1,n} = P$, а $P_{1,2}, P_{2,3}, \dots, P_{n-1,n}$ са индивидуалните отсечки $V_1V_2, V_2V_3, \dots, V_{n-1}V_n$. Отсечката V_1V_n не може да бъде изразена по този начин, понеже, както казахме, $P_{1,n}$ е целият многоъгълник.

Рекурсивната декомпозиция, базирана на Наблюдение 43, е следната. При дадени i и j , такива че $i + 1 < j$, минималната триангулация на $P_{i,j}$ е минимума по всички точки $i + 1, \dots, j - 1$ от сумата на $|V_iV_k|, |V_kV_j|$ и теглото на минимални триангулации на $P_{i,k}$ и $P_{k,j}$. Базата на рекурсията е $j = i + 1$ [†]: отсечките $P_{i,i+1}$ са изродени многоъгълници и техните триангулации с тегло нула дават началните условия.

Схемата за изчисление е следната. Нека $T[i, j]$ е теглото на минимална триангулация на $P_{i,j}$, където $1 \leq i < j \leq n$.

$$T[i, j] = \begin{cases} 0, & \text{ако } j = i + 1 \\ \min_{i+1 \leq k \leq j-1} (T[i, k] + T[k, j] + |V_iV_k| + |V_kV_j|), & \text{ако } j > i + 1 \end{cases} \quad (13.7)$$

Има една особеност: разстоянието между точки, които са съседни (иначе казано, които са краища на някоя от отсечките, дефиниращи многоъгълника), трябва да е 0. Не се твърди, че Евклидовото разстояние между тези точки е 0, а че за целите на нашето изчисление функцията, която връща разстоянието между две точки, трябва да връща 0, ако те са съседни. Причината е, че за теглото на триангулацията ни трябват само дължини на диагонали, а съседни точки не са краища на диагонал.

Алгоритъмът е много подобен на MATRIX-CHAIN MULTIPLICATION. Дребна разлика е, че тук главният диагонал не се ползва, а следващите два диагонала вдясно от него съдържат само нули: първият заради началните условия (отсечките, дефиниращи многоъгълника), а вторият поради това, че триъгълниците имат триангулации с нулево тегло, нямайки диагонали.

MINIMUM WEIGHT TRIANGULATION($\langle V_1, V_2, \dots, V_n \rangle$): редица от точки, образуваща изпъкнал многоъгълник)

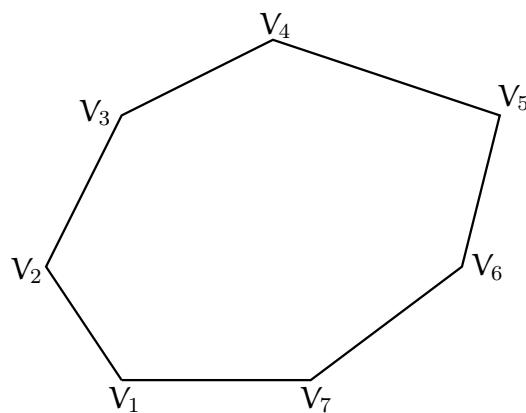
[†]A не $j = i$. $P_{i,i}$ е индивидуална точка, а чак толкова изродени многоъгълници не разглеждаме.

```

1  for i ← 1 to n - 1
2    T[i, i + 1] ← 0
3  for gap ← 2 to n - 1
4    for i ← 1 to n - gap
5      j ← i + gap
6      T[i, j] ← T[i, i + 1] + T[i + 1, j] + dist(Vi, Vi+1) + dist(Vi+1, Vj)
7      for k ← i + 2 to j - 1
8        T[i, j] ← min(T[i, j], T[i, k] + T[k, j] + dist(Vi, Vk) + dist(Vk, Vj)
9  return T[1, n]

```

Да разгледаме малък пример със седмоъгълник:



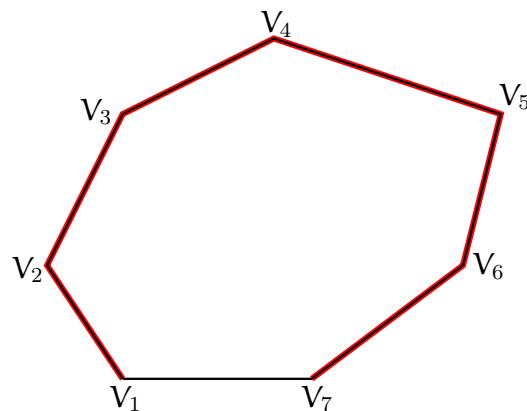
Така изглежда масивът T в началото на алгоритъма. Затъмнени са клетките, които няма да се ползват.

	1	2	3	4	5	6	7
1							
2							
3							
4							
5							
6							
7							

След изпълнението на **for**-цикъла на редове 1–2, T изглежда така:

	1	2	3	4	5	6	7
1		0					
2			0				
3				0			
4					0		
5						0	
6							0
7							

Поставянето на червените нули (това са началните условия) отговаря на триангулациите на изродените многоъгълници; тоест, страните на седмоъгълника без V_1V_7 .



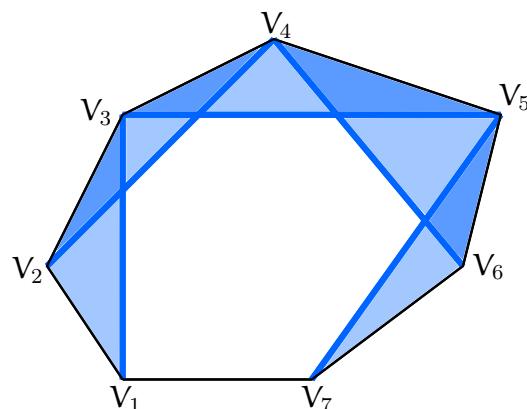
Да разгледаме запълването на следващия диагонал. Имаме $\text{gap} = 2$ от ред 3, $i = 1$ от ред 4 и $j = 3$ от ред 5. На ред 6:

$$T[1, 3] \leftarrow \underbrace{T[1, 2]}_0 + \underbrace{T[2, 3]}_0 + \underbrace{\text{dist}(V_1, V_2)}_0 + \underbrace{\text{dist}(V_2, V_3)}_0$$

така че $T[1, 3]$ става 0. Аналогично, целият диагонал се запълва с нули (в синьо):

	1	2	3	4	5	6	7
1	0	0					
2		0	0				
3			0	0			
4				0	0		
5					0	0	
6						0	
7							0

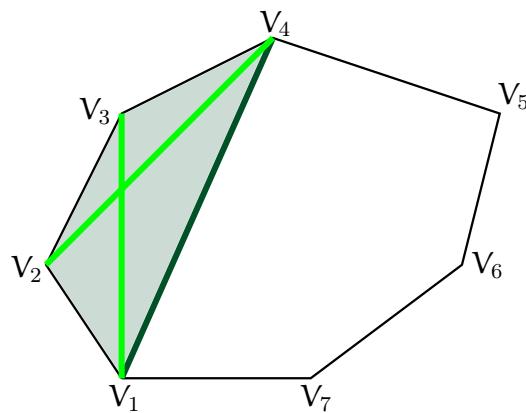
Поставянето на сините нули отговаря на триангулациите на ето тези триъгълници:



Да разгледаме запълването на следващия диагонал. Имаме gap = 3 от ред 3, i = 1 от ред 4 и j = 4 от ред 5. Редове 6–8 реализират това:

$$\begin{aligned} T[1, 4] \leftarrow \min(& \underbrace{T[1, 2]}_0 + \underbrace{T[2, 4]}_0 + \underbrace{\text{dist}(V_1, V_2)}_0 + \underbrace{\text{dist}(V_2, V_4)}_{42.43 \text{ mm}}, \\ & \underbrace{T[1, 3]}_0 + \underbrace{T[3, 4]}_0 + \underbrace{\text{dist}(V_1, V_3)}_{35 \text{ mm}} + \underbrace{\text{dist}(V_3, V_4)}_0) \end{aligned}$$

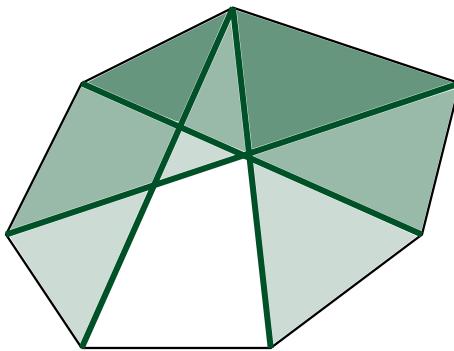
така че $T[1, 4]$ става 35 mm. Във фигурата, това изчисление отговаря на минимална триангуляция на зеления четириъгълник $P_{1,4}$, която е по-малкият от двата диагонала V_2V_4 и V_1V_3 .



В масива записваме 35 в клетка [1, 4]:

	1	2	3	4	5	6	7
1		0	0	35			
2			0	0			
3				0	0		
4					0	0	
5						0	0
6							0
7							

За да запълним този диагонал, пресмятаме минималните триангуляции (тоест, по-малкият от двата диагонала) на тези четири четириъгълника:



След запълването на зеления диагонал, масивът изглежда така:

	1	2	3	4	5	6	7
1		0	0	35			
2			0	0	42.43		
3				0	0	39.05	
4					0	0	43.01
5						0	0
6							0
7							

С това подробната симулация на алгоритъма върху примера спира. Когато gap стане 4, търсим теглото на минимална триангуляция на петоълници, три на брой; за всеки петоъгълник вземаме минимума от три стойности, отговарящи на трите междинни върха. Когато gap стане 5, търсим теглото на минимална триангуляция на два шестоъгълника; за всеки от тях вземаме минимума на четири стойности. И накрая, при gap = 6, намираме теглото на минимална триангуляция на целия седмоъгълник като минимума от пет стойности.

Коректността на алгоритъма е очевидна. Сложността му по време е $\Theta(n^3)$.

13.5 Факторизация при неасоциативно умножение

Дадена е азбука Σ и стринг $x = x_1x_2 \cdots x_n$ над Σ . Нека $|\Sigma| = k$. Дадена е бинарна операция $* : \Sigma^2 \rightarrow \Sigma$. Дадена е и буква $\sigma \in \Sigma$. Пита се: има ли скобуване на x , такова че, ако $*$ бъде приложена върху това скобуване, резултатът да бъде σ ?

Операцията $*$ да бъде приложена върху скобуване на x означава—при никакво предварително фиксирано скобуване—тя да бъде приложена върху най-вътрешната група букви $(x_i x_{i+1})$, резултатът от което е буквата $x_i * x_{i+1}$, и така нататък, докато е възможно.

Наместо строга индуктивна дефиниция ще поясним това с пример. Нека $\Sigma = \{a, b, c\}$. Нека $x = aab$, а следната таблица дефинира $*$:

	a	b	c
a	a	c	c
b	a	a	b
c	c	c	c

x има две скобувания: $((ab)b)$ и $(a(bb))$. Ако приложим $*$ съгласно първото скобуване, получаваме $((a * b) * b) = (c * b) = c$. Ако приложим $*$ съгласно второто скобуване, получаваме $(a * (b * b)) = (a * a) = a$. Причината да има разлика е, че $*$ не е асоциативна.

Ако $*$ е асоциативна, задачата се обезсмисля, защото резултатът е един и същи при всяко скобуване. Но при неассоциативна операция, както видяхме, резултатът от прилагането върху различни скобувания може да е различен.

Задачата прилича много на MATRIX-CHAIN MULTIPLICATION. В задачата MATRIX-CHAIN MULTIPLICATION матричното умножение е асоциативно, но това няма значение, защото там гледаме не каква е матрицата-результат, а с цената на колко скаларни умножения я постигаме, което вече не е асоциативно. Съществената разлика между тази задача, от една страна, и MATRIX-CHAIN MULTIPLICATION и MINIMUM WEIGHT TRIANGULATION, от друга, е че в тази задача отговорът не е число. Тази задача се решава ефикасно от алгоритъм, който външно наподобява много MATRIX-CHAIN MULTIPLICATION, пак с триъгълна таблица, която се запълва от главния диагонал навън, но сега клетките на таблицата съдържат не числа, а множества от букви. Ако таблицата се назовава T , то $T[i, j]$ има следният смисъл:

$$T[i, j] = \begin{cases} x_i, & \text{ако } i = j \\ \bigcup_{k=i}^{j-1} \{a * b \mid a \in T[i, k], b \in T[k+1, j]\}, & \text{ако } i < j \end{cases}$$

Следният алгоритъм реализира ефикасно тази схема за изчисляване.

NONASSOCIATIVE MULTIPLICATION($\Sigma, * : \Sigma^2 \rightarrow \Sigma, x = x_1 \dots x_n \in \Sigma^+, \sigma \in \Sigma$)

```

1   for i ← 1 to n
2       T[i, i] ← x_i
3   for diag ← 2 to n
4       for i ← 1 to n - diag + 1
5           j ← i + diag - 1
6           T[i, j] ← ∅
7           for k ← i to j - 1
8               foreach a ∈ T[i, k]
9                   foreach b ∈ T[k + 1, j]
10                  T[i, j] ← T[i, j] ∪ {a * b}
11   if σ ∈ T[1, n]
12       return 1
13   else
14       return 0

```

Коректността на алгоритъма е очевидна предвид това, че той реализира коректна рекурсивна декомпозиция и базирана на нея изчислителна схема. Сложността му по време е $\Theta(n^3)$, ако смятаме, че $k = \Theta(1)$. В противен случай трябва да изразим сложността като функция на n и k по този начин: $\Theta(k^2n^3)$.

13.6 Деривация в контекстно-свободна граматика

Контекстно-свободна граматика (КСГ) е наредена четворка $\Gamma = \langle \Sigma, N, S, R \rangle$, където Σ е крайно множество от терминали (Σ е азбуката), N е крайно множество от нетерминали, такова че $N \cap \Sigma = \emptyset$, $S \in N$ е фиксиран нетерминал, наречен начален нетерминал, и R е множеството от правила (още се наричат продукции) от вида $A \rightarrow X$, където A е нетерминал, а

$X \in (\Sigma \cup N)^*$. Формално, R е бинарна релация $R \subseteq N \times (\Sigma \cup N)^*$. Дефинираме бинарната релация $\Rightarrow \subseteq (\Sigma \cup N)^* \times (\Sigma \cup N)^*$ така:

$$x \Rightarrow z \stackrel{\text{def}}{\leftrightarrow} \exists A \in N \exists x_1, x_2 \in (\Sigma \cup N)^* (x = x_1 A x_2 \text{ и има правило } A \rightarrow \alpha \text{ и } z = x_1 \alpha x_2)$$

Релацията \Rightarrow^* е рефлексивното и транзитивно затваряне на \Rightarrow . Ако $x \Rightarrow^* y$, казваме, че има *деривация от x до y в Г*. За всеки $y \in \Sigma^*$ казваме, че y има *деривация в Г*, ако $S \Rightarrow^* y$.

КСГ е в Нормална Форма на Чомски (пишем кратко КСГНФЧ), ако всички правила са от вида $A \rightarrow BC$ или $A \rightarrow a$ или $S \rightarrow \epsilon$, където A, B и C са нетерминали, a е терминал, а ϵ е празният стринг.

Предложете ефикасен алгоритъм, който по дадена КСГНФЧ $\Gamma = \langle \Sigma, N, S, R \rangle$ и $y \in \Sigma^*$ връща 1, ако y има деривация в Γ , или 0 в противен случай. Дайте съвсем кратка обосновка на коректността и сложността по време на предложението от Вас алгоритъм. Приемете, че размерът на Γ е $O(1)$.

Решение: Ще построим алгоритъм по схемата динамично програмиране. Този алгоритъм е добре известен и се нарича CYK на имената на създателите си Cocke, Younger и Kasami.

Нека $y = y_1 y_2 \cdots y_n$. БОО, нека $n \geq 1$. Ако за всеки $y_i \cdots y_j$, където $1 \leq i \leq j \leq n$, намерим множеството $X_{i,j} = \{A \in N \mid A \stackrel{*}{\Rightarrow} y_i \cdots y_j\}$, то ние сме готови! Отговорът е 1 теств $S \in X_{1,n}$. Това, което позволява да направим ефикасен алгоритъм е, че при $i < j$ можем да пресмятаме ефикасно всяко $X_{i,j}$ от множествата $X_{i',j'}$, където $i \leq i' \leq j' \leq j$ и $j' - i' < j - i$.

Базата на рекурсията са множествата $X_{i,i}$ за $1 \leq i \leq n$:

$$\forall i \in \{1, 2, \dots, n\} : X_{i,i} = \{A \in N \mid \text{има правило } A \rightarrow y_i\}$$

зашото в КСГНФЧ, ако по време на деривацията в сентенциалната форма се появи нетерминал, той никога не изчезва в смисъл, че в крайния стринг ще има поне един съответстващ на него терминал; ерго, правилата от вида $A \rightarrow BC$ никога не могат да стартират деривация на стринг с една буква.

При $i < j$, $X_{i,j}$ е обединението, по всички нетривиални факторизации $(y_i \cdots y_k) \cdot (y_{k+1} \cdots y_j)$ на стринга $y_i \cdots y_j$ (тоест, по всички $k \in \{i, \dots, j-1\}$) на тези нетерминали, които са левите страни в продукции от вида $A \rightarrow BC$, такива че $B \stackrel{*}{\Rightarrow} y_i \cdots y_k$ и $C \stackrel{*}{\Rightarrow} y_{k+1} \cdots y_j$. Но тези нетерминали B и C са елементите на множествата $X_{i,k}$ и $X_{k+1,j}$. При изчисляването на $X_{i,j}$ множествата $X_{i,k}$ и $X_{k+1,j}$ са вече известни.

Самият алгоритъм може да се реализира по начин, аналогичен на реализациите на алгоритъма MATRIX-CHAIN MULTIPLICATION.

DERIVATION($\Gamma = \langle \Sigma, N, S, R \rangle$, $y \in \Sigma^*$)

```

1  n ← |y|
2  for i ← 1 to n
3      M[i, i] ← {A | (A → y_i) ∈ R}
4  for diag ← 2 to n
5      for i ← 1 to n - diag + 1
6          j ← i + diag - 1
7          M[i, j] ← ∅
8          for k ← i to j - 1
9              for B ∈ M[i, k]
10                 for C ∈ M[k + 1, j]
```

```

11      for P ∈ R
12          if P = (A → BC)
13              M[i, j] ← M[i, j] ∪ {A}
14  if S ∈ M[1, n]
15      return 1
16 else
17     return 0

```

Обосновката на коректността следва от обосновката на рекурсията. Сложността по време е $\Theta(n^3)$, ако граматиката има константна големина – тогава както множествата $M[a, b]$, така и множеството от продукциите, имат големини, ограничени от константа, от което следва, че и трите най-вътрешни цикъла “въртят” само константен брой пъти.

13.7 Set Partition

Дадено е мултимножество от цели положителни числа. Във варианта на задача за разпознаване се пита дали има подмултимножество, такова че сумата от числата в него е равна на сумата на числата извън него. Във варианта на оптимизационна задача се търси подмултимножество, такова че сумата от числата в него е равна на сумата на числата извън него.

Тази задача е частен случай на задачата KNAKSACK (вижте Секция 13.9), но решението ѝ си заслужава да бъде разгледано отделно. Ето я, формално написана, но не чрез мултимножества, а чрез обикновени множества, чито елементи има размер (*size*, оттам и “*s*” за името на функцията). Разглеждаме по-лесният вариант на задача за разпознаване.

Изч. Задача: SET PARTITION

пример: Множество $A = \{a_1, a_2, \dots, a_n\}$. Функция $s : A \rightarrow \mathbb{N}^+$.

въпрос: Дали съществува $A' \subset A$, такова че $\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a)$?

Нека $B = \sum_{a \in A} s(a)$. Очевидно е, че ако B е нечетно, отговорът е НЕ. БОО, нека B е четно. Разглеждаме множеството A като наредено с линейна наредба, а именно тази, която идва от имената на елементите: a_1 , после a_2, \dots , после a_n . Това може да изглежда странно, защото тази наредба е абсолютно произволна, а ние ще конструираме решение по отношение на нея. Всъщност няма нищо странно. Всяка от всички $n!$ линейни наредби е еднакво добра за конструиране на решение. Иска се да има **някаква** линейна наредба – точно както назва Skiena за същността на схемата Динамично Програмиране. За краткост, нека A^i означава множеството $\{a_1, \dots, a_i\}$, за всяко $i \in \{0, \dots, n\}$. Очевидно $A^0 = \emptyset$ и $A^n = A$.

Да характеризираме решение[†] на задачата. Решението (по-точно, отговорът, защото задачата е за разпознаване) е Да тества съществува подмножество на A , такова, че сумата от размерите на елементите му е точно $\frac{1}{2}B$. Такова решение се състои от елементи.

// очевидно – е, и?

Продължаваме: всяко непразно подмножество на A със suma $\frac{1}{2}B$ се състои подмножество със suma $\frac{1}{2}B - p$ и още един елемент с размер p .

// това вече е нещо.

Продължаваме: всяко $X \subseteq A$ със suma $\frac{1}{2}B$ е подмножество на някое A^i , където $i \in \{1, \dots, n\}$. При това, или $a_i \in X$, в който случай $X \setminus \{a_i\}$ е подмножество на A^{i-1} , или $a_i \notin X$, в който случай X е подмножество на A^{i-1} .

// аха, това е!

[†]Това не е оптимизационна задача, така че не говорим на “оптимално решение”, а просто за “решение”.

За това характеризиране на структурата ще направим рекурсивна декомпозиция. За краткост на записа въвеждаме следната нотация. Нека $P(i, j)$, където $1 \leq i \leq n$ и $0 \leq j \leq \frac{1}{2}B$, е множество от предикати със следния смисъл:

$$P(i, j) \stackrel{\text{def}}{=} \exists X \subseteq A^i : \sum_{a \in X} s(a) = j$$

Тяхната истинност се определя така.

- Има точно две помножества на A^1 , чиято сума е число от $\{0, \dots, \frac{1}{2}B\}$: това са празното множество със сума 0 и множеството $\{a_1\}$ със сума $s(a_1)$. Еrgo, $P(1, j)$ е истина за точно две стойности на j , а именно $j = 0$ и $j = s(a_1)$. Това означава, че за всички други стойности на j , $P(1, j)$ е лъжа.
- За $i > 1$, подмножествата на A^i , чиято сума е число от $\{0, \dots, \frac{1}{2}B\}$, са:
 - ◆ подмножествата на A^{i-1} , чиято сума е число от $\{0, \dots, \frac{1}{2}B\}$, и
 - ◆ подмножествата на A^{i-1} , чиято сума е число от $\{0, \dots, \frac{1}{2}B - s(a_i)\}$.

Тези два класа може да имат непразно сечение. Ерго, $P(1, j)$ е истина тстк $P(i-1, j)$ е истина или $P(i-1, j - s(a_i))$ е истина и $s(a_i) \leq j$. Това “или” е включващо.

Очевидно търсеният отговор е $P(n, \frac{1}{2}B)$.

Ето схемата за изчисление, базирана на тази рекурсивна декомпозиция.

$$P[1, j] = \begin{cases} 1, & \text{ако } j = 0 \text{ или } j = s(a_1) \\ 0, & \text{в противен случай} \end{cases} \quad (13.8)$$

$$P[i, j] = \begin{cases} 1, & \text{ако } P[i-1, j] \text{ или } (s(a_i) \leq j \text{ и } P[i-1, j - s(a_i)]) \\ 0, & \text{в противен случай} \end{cases}, \text{ ако } i > 1 \quad (13.9)$$

Ето алгоритъм, който извършва тези изчисления и е по схемата Динамично Програмиране.

```

SET PARTITION( $A = \{a_1, \dots, a_n\}$ ,  $s : A \rightarrow \mathbb{N}^+$ )
1   $B \leftarrow \sum_{i=1}^n s(a_i)$ 
2  if isodd( $B$ )
3    return 0
4   $B \leftarrow B/2$ 
5  for  $j \leftarrow 0$  to  $B$ 
6    if  $j = 0$  or  $j = s(a_1)$ 
7       $P[1, j] \leftarrow 1$ 
8    else
9       $P[1, j] \leftarrow 0$ 
10 for  $i \leftarrow 2$  to  $n$ 
11   for  $j \leftarrow 0$  to  $B$ 
12     if  $P[i-1, j]$  or  $(s(a_i) \leq j \text{ and } P[i-1, j - s(a_i)])$ 
13        $P[1, j] \leftarrow 1$ 
14     else
15        $P[1, j] \leftarrow 0$ 
16 return  $P[n, B]$ 

```

Пример за работата на този алгоритъм има в [20, стр. 91].

Коректността на алгоритъма е очевидна предвид (13.8) и (13.9). Интерес представлява изследването на сложността му по време. Тя е $\Theta(nB)$. Множителят n е размера на входа, при допускането, че всяко от числата $s(a_i)$ има размер $\Theta(1)$. Но B какво е? B е полусумата от **големините** на числата. Тези големини може да са произволни по отношение на n (с други думи, не са ограничени отгоре от каквато и да е функция на n), така че и B не е ограничено отгоре. Излиза, че сложността не е дефинирана (с други думи, е безкрайност). Разбира се, този парадоксален извод се появява само при допускането на нашия изчислителен модел, че числата във входа имат константни размери за всички големини; тоест, когато не отчитаме кодиранията им. В по-реалистичен модел на изчисление, който отчита и кодиранията на числата, сложността $\Theta(nB)$ е експоненциална в големината на входа.

Накратко, в изчислителния модел, който типично използваме, SET PARTITION има недефинирана (безкрайна) сложност, а в по-реалистичен модел има експоненциална сложност.

Това не прави алгоритъма безполезен. Ако числата са малки, в смисъл, че имат кодирания, чиито големини можем да смятаме за константи, алгоритъмът е полезен и смислен. Забележете, че SET PARTITION има “истинска” полиномиална сложност по време, ако числата $s(a_i)$ са кодирани в унарна бройна система! Това не прави алгоритъмът по-бърз, дори напротив, но кодирането в унарна бройна система изкуствено “раздува” големината на входа, а ние мерим сложността като функция именно на големината на входа. Следното определение се основава на този факт.

Определение 59: Псевдополиномиална сложност

Казваме, че алгоритъм A има *псевдополиномиална сложност* по време, ако A има полиномиална сложност по време при условие, че числата във входа са кодирани в унарна бройна система.

И така, SET PARTITION има псевдополиномиална сложност по време.

13.8 2 Equal Sum Subsets

Тази задача е обобщение на SET PARTITION. Сега се пита дали има две непразни подмножества на A , които имат една и съща сума от размерите на елементите. БОО, тези множества имат празно сечение, защото наличието на общи елементи в тях е без значение за въпроса, на който трябва да се отговори. Накратко ще наричаме задачата (и алгоритъма) “2ESS”.

Изч. Задача: 2ESS

пример: Множество $A = \{a_1, a_2, \dots, a_n\}$. Функция $s : A \rightarrow \mathbb{N}^+$.

въпрос: Дали съществуват непразни $A', A'' \subset A$, такива че $A' \cap A'' = \emptyset$ и $\sum_{a \in A'} s(a) = \sum_{a \in A''} s(a)$?

Задачата е **NP**-пълна [74]. 2ESS е обобщение на SET PARTITION, при която се иска и $A' \cup A'' = A$. Оттук настетне нека $B = \sum_{a \in A} s(a)$.

Наблюдение 44

Ако $B < 2^n - 1$, то отговорът е тривиално Да по принципа на Dirichlet: броят на непразните подмножества на A е точно $2^n - 1$ и ако възвожните стойности за сума на подмножество са по-малко, съществуват различни $C, D \subseteq A$, такива че $\sum_{a \in C} s(a) = \sum_{a \in D} s(a)$. Ако $C \cap D \neq \emptyset$, то $C' = C \setminus D$ и $D' = D \setminus C$ са непразни подмножества на A с празно сечение и $\sum_{a \in C'} s(a) = \sum_{a \in D'} s(a)$.

Това обаче е неконструктивен резултат. Ние знаем, че има такива подмножества C и D , но нямаме представа кои са или как да ги намерим ефикасно.

Въпреки че 2ESS е обобщение на SET PARTITION, алгоритъмът за 2ESS, който ще конструираме, не решава SET PARTITION.

Нека $\beta = \lfloor \frac{1}{2}B \rfloor$. Нека $A^i = \{a_1, \dots, a_i\}$, за $1 \leq i \leq n$. Дефинираме $n(\beta + 1)^2$ булеви променливи $Q[t, i, j]$, $1 \leq t \leq n$, $1 \leq i, j \leq \beta$, които имат следния смисъл:

$$Q[t, i, j] = 1 \text{ тстк } \exists A', A'' \subseteq A^t : \left(A' \cap A'' = \emptyset \wedge \sum_{a \in A'} s(a) = i \wedge \sum_{a \in A''} s(a) = j \right)$$

Отговорът е Да тстк съществува $k \in \{1, \dots, \beta\}$, такова че $Q[n, k, k] = 1$.

Схемата за изчисление е следната. Базата е

$$Q[1, i, j] = \begin{cases} 1, & \text{ако } i = j = 0 \vee (i = s(a_1) \wedge j = 0) \vee (i = 0 \wedge j = s(a_1)) \\ 0, & \text{в противен случай} \end{cases}$$

За $2 \leq t \leq n$:

$$Q[t, i, j] = Q[t - 1, i, j] \vee (i \geq s(a_t) \wedge Q[t - 1, i - s(a_t), j]) \vee (j \geq s(a_t) \wedge Q[t - 1, i, j - s(a_t)])$$

Очевидно е как да направим алгоритъм от тази схема за изчисление. Наместо формално обосноваване на коректността ще разгледаме малък пример. Нека $n = 4$ и $s(a_1) = 2, s(a_2) = 3, s(a_3) = 4$ и $s(a_4) = 5$. Тогава $B = 14$ и $\beta = 7$. Наместо да пишем $Q[t, i, j]$, ще пишем $Q^{(t)}[i, j]$. И ще записваме само единиците в матриците $Q^{(t)}$.

За да получим $Q^{(1)}$, слагаме 1 в клетка $[0, 0]$ и, тъй като $s(a_1) = 2$, още по една единица в $[0, 2]$ и $[2, 0]$. Тогава $Q^{(1)}$ е:

	0	1	2	3	4	5	6	7
0	1		1					
1								
2	1							
3								
4								
5								
6								
7								

За да получим $Q^{(2)}$, преписваме единиците от $Q^{(1)}$. После, тъй като $s(a_2) = 3$, слагаме по една единица на разстояние 3, по хоризонтал и вертикал, от досега сложените единици и $Q^{(2)}$ е (новите единици са в синьо):

	0	1	2	3	4	5	6	7
0	1		1	1		1		
1								
2	1			1				
3	1		1					
4								
5	1							
6								
7								

За да получим $Q^{(3)}$, преписваме единиците от $Q^{(2)}$. После, тъй като $s(a_3) = 4$, слагаме по една единица на разстояние 4, по хоризонтал и вертикал, от досега сложените единици и $Q^{(4)}$ е (новите единици са в синьо):

	0	1	2	3	4	5	6	7
0	1		1	1	1	1	1	1
1								
2	1			1	1			1
3	1		1		1		1	
4	1		1	1		1		
5	1				1			
6	1			1				
7	1		1					

Аналитично получаваме $Q^{(4)}$, като имаме предвид, че $s(a_4) = 5$:

	0	1	2	3	4	5	6	7
0	1		1	1	1	1	1	1
1								
2	1			1	1	1		1
3	1		1		1	1	1	1
4	1		1	1		1		1
5	1		1		1	1	1	1
6	1			1		1		
7	1		1	1	1	1		1

С червен фон са посочени единиците върху главния диагонал. Отговорът е Да тества има поне една единица върху главния диагонал, така че за този пример алгоритъмът трябва да върне Да.

13.9 Knapsack

Крадец иска да открадне предмети, като ги сложи в раница. Всеки предмет има положителни тегло и цена, като те не са свързани по никакъв начин. Оптималната кражба би била да

задигне всички предмети, но има максимално сумарно тегло за предметите в раницата – при по-голямо сумарно тегло тя се къса. Това максимално сумарно тегло е *капацитетът* на раницата.

В оптимизационната версия на задачата се иска да се намери подмножество от предмети с максимална сумарна цена, чието сумарно тегло не надвишава капацитета на раницата.

Ако предметите не са атомарни, а са неогранично делими, можем да кажем с известна доза условност, че е дадена *континуална* версия на задачата. Примерно, нещата за крадене са златен прах, сребърен прах и бронзов прах. Континуалната версия на задачата е решена тривиално от алчен алгоритъм, който първо пресмята относителната цена на делимите предмети – да кажем, златният прах на килограм е по-скъп от сребърния, който на свой ред е по-скъп от бронзовия на килограм – и после пълни раницата с относително най-скъпото нещо (златен прах в нашия пример), докато не достигне капацитета ѝ или това нещо не се изчерпи. Ако относително най-скъпото нещо първо се изчерпи, алчният алгоритъм продължава да пълни раницата с второто относително най-скъпо нещо, и така нататък. Очевидно този алгоритъм е ефикасен и намира оптимално решение.

Ако предметите са атомарни, да кажем, че е дадена *дискретна* версия на задачата. В дискретната версия алчният подход може да доведе до решение, което не е оптимално – вземайки предмет с най-високо отношение цена/тегло, може да се лишим от възможност да сложим други предмети, които биха били по-изгодни като цяло, въпреки че поотделно имат по-лошо отношение цена/тегло. С други думи, локалната, или късогледата, алчност сега не е печеливша стратегия; сега трябва да сме “глобално алчни”, за да напълним раницата с максимално сумарно скъпни неща, без да я скъсаме.

Дискретната версия на задачата е широко известна в поне три варианта, които ще разгледадме сега.

13.9.1 Unbounded Knapsack

Изч. Задача: UNBOUNDED KNAPSACK

пример: Множество от видове предмети $A = \{a_1, \dots, a_n\}$, като има неограничено много предмети от всеки вид. За всеки вид a_i , $1 \leq i \leq n$, са дадени неговата стойност $v(a_i) \in \mathbb{R}^+$ и неговото тегло $w(a_i) \in \mathbb{N}^+$. Даден е капацитет $C \in \mathbb{N}^+$.

решение: $(x_1, \dots, x_n) \in \mathbb{N}^n$, такава че

$$\sum_{i=1}^n x_i w(a_i) \leq C \quad (13.10)$$

$$\sum_{i=1}^n x_i v(a_i) \text{ е максимална} \quad (13.11)$$

Лесно можем да преведем това формално описание в терминологията на крадеца с раницата. С е капацитетът на раницата. Предметите за крадене са от n вида, като видовете са a_1, \dots, a_n , а от всеки вид има неограничено много предмети за крадене. Предметите от даден вид a_i са неразличими помежду си и имат една и съща стойност $v(a_i)$ и едно и също тегло $w(a_i)$. Под “неограничено” нямаме предвид безброй много, а нещо много по-реалистично: по отношение на дадената раница, крадецът може да я напълни с предмети от само един вид до достигане на капацитета; с други думи, никога няма да се изчерпят предметите от даден вид, запълвайки раницата. Наредената n -орка от естествени числа (x_1, \dots, x_n) има смисъл на това, по колко броя от всеки вид ще бъдат взети в раницата. (13.10) и (13.11) казват, че

трябва да се вземат по толкова предмети от всеки вид, че общото тегло да не надхвърли капацитета, като тази подборка трябва да има максимална сумарна стойност.

Тук ще разгледаме олекотената версия на задачата, в която изчисляваме само максималното количество $\sum_{i=1}^n x_i v(a_i)$, без да изчисляваме n -торката (x_1, \dots, x_n) , която назава по колко предмети от всеки вид да вземем.

Следното решение на задачата е добре известно. Правим рекурсивна декомпозиция по капацитета на раницата. Нека $M[j]$ е максималната стойност на предмети, които можем да сложим в раница с капацитет j , за всички $j \in \{0, \dots, C\}$. Ако можем да пресмятаме $M[j]$ чрез $M[j-1], \dots, M[0]$, ще получим решението $M[C]$. Схемата за изчисление е следната.

$$M[j] = \begin{cases} 0, & \text{ако } j = 0 \\ \max(M[j-1], \max_{i \in \{1, \dots, n\} \text{ и } w(a_i) \leq j} (M[j - w(a_i)] + v(a_i))), & \text{ако } j > 0 \end{cases}$$

Идеята е ясна. Ако раницата има нулев капацитет, в нея не можем да сложим нищо. Ако позволим капацитет $j > 0$, в нея можем да сложим най-много:

- или това, което можем да сложим при по-малкия капацитет $j-1$, с други думи не се възползваме от целия капацитет j ,
- или се възползваме от целия капацитет j , опитвайки да сложим един предмет от всеки вид, чието тегло не надхвърля j ; за да сложим предмет от вид a_i в раница с капацитет j , трябва да има свободен капацитет $j - w(a_i)$; максимумът, който можем да постигнем, слагайки предмет от вид a_i , е максимумът за раница с капацитет $j - w(a_i)$ плюс стойността $v(a_i)$ на предмета.

Ако искаме пълно решение на задачата, което ни дава не просто числен отговор, а освен това и по колко предмети от всеки вид да сложим в раницата, можем да записваме за всяко j , $1 \leq j \leq C$, дали $M[j]$ е получено като $M[j-1]$ или по другия начин. Ако е по другия начин, а именно като максимум по всички i , такива че $w(a_i) \leq j$, записваме i , за което се постига максимална стойност на $M[j - w(a_i)] + v(a_i)$. От тази информация може лесно да генерираме пълно решение във време $O(C)$.

Сложността по време е $\Theta(nC)$, а по памет е $\Theta(C)$. Това е поредният псевдополиомиален алгоритъм, който разглеждаме. Ако числата $w(a_i)$ са малки, масивът M е малък и алгоритъмът е ефикасен.

13.9.2 0-1 Knapsack

Изч. Задача: 0-1 KNAPSACK

пример: Множество от предмети $A = \{a_1, \dots, a_n\}$. За всяко a_i , $1 \leq i \leq n$, са дадени неговата стойност $v(a_i) \in \mathbb{R}^+$ и неговото тегло $w(a_i) \in \mathbb{N}^+$. Даден е капацитет $C \in \mathbb{N}^+$.

решение: Подмножество $A' \subseteq A$, такова че:

$$\sum_{a \in A'} w(a) \leq C$$

$$\sum_{a \in A'} v(a) \text{ е максимална}$$

Бихме могли да дефинираме и тази версия на задачата чрез вектор (x_1, \dots, x_n) , само че булев, а не от естествени числа. Това би бил характеристичен вектор, а знаем, че характеристичните вектори определят подмножества. Така че дори с такава формулировка, това, което се има предвид, е подмножество с максимална обща стойност и сумарен размер, не по-голям от капацитета.

В тази версия на задачата—също както и в предишната—не е добра идея да правим рекурсивна декомпозиция само по подмножествата от първите i елемента на A (има се предвид $\{a_1, \dots, a_i\}$). Читателят лесно може да измисли пример, в който оптимално решение за, да кажем, $\{a_1, \dots, a_6\}$ не ползва оптимално решение за $\{a_1, \dots, a_5\}$. Ще направим двумерна рекурсивна декомпозиция, подобна на решението на SET PARTITION. Нека $M[i, j]$ е (стойността на) оптимално решение за раница с капацитет j , ползвашо предмети a_1, \dots, a_i , където $0 \leq i \leq n$ и $0 \leq j \leq C$. Решението е $M[n, C]$. Схемата за изчисление е следната.

$$\begin{aligned} M[i, 0] &= 0, \text{ за } 1 \leq i \leq n \quad // \text{ нулев капацитет} \\ M[0, j] &= 0, \text{ за } 1 \leq j \leq C \quad // \text{ нула предмети за слагане} \\ M[i, j] &= M[i - 1, j], \text{ ако } i > 0 \text{ и } w(a_i) > j \quad // a_i \text{ не са побира при капацитет } j \\ M[i, j] &= \max \left(\underbrace{M[i - 1, j]}_{\text{не вземаме } a_i}, \underbrace{M[i - 1, j - w(a_i)] + v(a_i)}_{\text{вземаме } a_i} \right), \text{ в противен случай} \end{aligned} \quad (13.12)$$

Случай (13.12) е основата на декомпозицията и може би единственият, чиято обосновка не е напълно очевидна. В него решаваме дали да вземем предмета a_i , като стойността при такова решение ни е известна, тя е $M[i - 1, j]$, или да го вземем, при което обаче трябва да има капацитет за него $w(a_i)$, така че гледаме колко най-много можем да сложим, оставяйки си свободен капацитет $w(a_i)$, и какво ще получим при това; тази стойност се пресмята тривиално и тя е $M[i - 1, j - w(a_i)] + v(a_i)$.

Алгоритъм, реализиращ това изчисление, е следният.

```
0-1 KNAPSACK( $A = \{a_1, \dots, a_n\}$ ,  $v : A \rightarrow \mathbb{R}^+$ ,  $s : A \rightarrow \mathbb{N}^+$ ,  $C \in \mathbb{N}^+$ )
1  for  $i \leftarrow 0$  to  $n$ 
2     $M[i, 0] \leftarrow 0$ 
3  for  $j \leftarrow 0$  to  $C$ 
4     $M[0, j] \leftarrow 0$ 
5  for  $i \leftarrow 1$  to  $n$ 
6    for  $j \leftarrow 1$  to  $C$ 
7      if  $w(a_i) > j$ 
8         $M[i, j] \leftarrow M[i - 1, j]$ 
9      else
10         $M[i, j] \leftarrow \max(M[i - 1, j], M[i - 1, j - w(a_i)] + v(a_i))$ 
11 return  $M[n, C]$ 
```

Коректността на алгоритъма е очевидна. Сложността както по време, така и по памет, е $\Theta(nC)$.

13.9.3 Bounded Knapsack

Сега са дадени определен брой копия от всеки предмет. Тази версия на задачата е обобщение на 0-1 KNAPSACK, където беше дадено точно по един предмет от всеки вид. Тук ще разгледаме решение от книгата на Martello и Toth [47].

Изч. Задача: BOUNDED KNAPSACK

пример: Множество от видове предмети $A = \{a_1, \dots, a_n\}$. За всеки вид a_i , $1 \leq i \leq n$, са дадени неговата стойност $v(a_i) \in \mathbb{R}^+$, неговото тегло $w(a_i) \in \mathbb{N}^+$ и броят предмети от него $b_i \in \mathbb{N}^+$. Даден е капацитет $C \in \mathbb{N}^+$.

решение: $(x_1, \dots, x_n) \in \mathbb{N}^n$, такава че

$$\sum_{i=1}^n x_i w(a_i) \leq C \quad (13.13)$$

$$0 \leq x_i \leq b_i, \text{ за } 1 \leq i \leq n \quad (13.14)$$

$$\sum_{i=1}^n x_i v(a_i) \text{ е максимална} \quad (13.15)$$

БОО, нека $\sum_{i=1}^n b_i w(a_i) > C$, защото в противен случай имаме тривиално решение $x_i = b_i$ за $1 \leq i \leq n$.

Задачата се свежда до 0-1 KNAPSACK със следната трансформация. Нека е даден пример Π_B на BOUNDED KNAPSACK:

$$\langle \{a_1, \dots, a_n\}, (v_1, \dots, v_n), (w_1, \dots, w_n), (b_1, \dots, b_n), C \rangle$$

На него съответства пример $\Pi_{0,1}$ на 0-1 KNAPSACK:

$$\langle \{\hat{a}_1, \dots, \hat{a}_m\}, (\hat{v}_1, \dots, \hat{v}_m), (\hat{w}_1, \dots, \hat{w}_m), C \rangle$$

За всеки вид a_i , $1 \leq i \leq n$, конструираме $\lceil \log_2(b_i + 1) \rceil$ предмети в $\Pi_{0,1}$ така:

- първо, $\lfloor \log_2 b_i \rfloor$ предмети, чиито наредени двойки (стойност, тегло) са

$$(v_i, w_i), (2v_i, 2w_i), (4v_i, 4w_i), \dots, (\lfloor \log_2 b_i \rfloor v_i, \lfloor \log_2 b_i \rfloor w_i)$$

- второ, ако b_i не е число от вида “точна степен на двойката минус единица”, още един предмет със стойност

$$b_i v_i - (v_i + 2v_i + 4v_i + \dots + \lfloor \log_2 b_i \rfloor v_i)$$

и размер

$$b_i w_i - (w_i + 2w_i + 4w_i + \dots + \lfloor \log_2 b_i \rfloor w_i)$$

Очевидно броят на предметите в $\Pi_{0,1}$, съответни на a_i , е $\lceil \log_2(b_i + 1) \rceil$, като сумата от стойностите им е $b_i v_i$ и сумата от размерите им е $b_i w_i$. Общо броят на предметите в $\Pi_{0,1}$ е $\sum_{i=1}^n \lceil \log_2(b_i + 1) \rceil$. Накратко ще записваме това число като m

m е и броят на булевите променливи в решението на $\Pi_{0,1}$. Еrgo, на всяка променлива естествено число x_i от решението Π_B съответстват булеви променливи $\hat{x}_{i,1}, \dots, \hat{x}_{i,\lceil \log_2(b_i+1) \rceil}$ от решението на $\Pi_{0,1}$. Нещо повече, за $1 \leq h \leq \lceil \log_2(b_i + 1) \rceil$, булевата променлива $\hat{x}_{i,h}$ “дава” n_h предмети от Π_B , където

$$n_h = \begin{cases} 2^{h-1}, & \text{ако } h < \lceil \log_2(b_i + 1) \rceil \\ b_i - \sum_{j=1}^{\lceil \log_2(b_i+1) \rceil - 1} 2^{j-1}, & \text{ако } h = \lceil \log_2(b_i + 1) \rceil \end{cases}$$

Следователно $x_i = \sum_{h=1}^{\lceil \log_2(b_i+1) \rceil} n_h \hat{x}_{i,h}$ може да приеме всяка стойност от $\{0, \dots, b_i\}$.

Сложността на алгоритъма е сложността на трансформацията плюс сложността на 0-1 KNAPSACK върху конструирания пример. Сложността на трансформацията е $\Theta(m)$, а след това 0-1 KNAPSACK работи във време $\Theta(mC)$, така че общата сложност е $\Theta(mC)$. Да се направи подробно формално доказателство на коректността би било доста дълго, така че ще разгледаме примера за трансформацията от [47, стр. 83].

Нека е даден пример на BOUNDED KNAPSACK, в който $n = 3$.

	a_1	a_2	a_3
v	10	15	11
w	1	3	5
b	6	4	2

На око се вижда, че оптималното решение е $(x_1, x_2, x_3) = (6, 1, 0)$ с обща стойност

$$6 \cdot 10 + 1 \cdot 15 + 0 \cdot 11 = 75$$

и общо тегло

$$6 \cdot 1 + 1 \cdot 3 + 0 \cdot 5 = 9$$

Да приложим трансформацията. Имаме $\lceil \log_2(a_1+1) \rceil = 3$, $\lceil \log_2(a_2+1) \rceil = 3$ и $\lceil \log_2(a_3+1) \rceil = 2$. Нека трите предмета, съответстващи на a_1 , са a'_1 , a''_1 и a'''_1 , трите предмета, съответстващи на a_2 , са a'_2 , a''_2 и a'''_2 и двата предмета, съответстващи на a_3 , са a'_3 и a''_3 . Това са осемте предмета в примера на 0-1 KNAPSACK, който строим. Стойностите и теглата са следните.

	a'_1	a''_1	a'''_1	a'_2	a''_2	a'''_2	a'_3	a''_3
v	10	20	30	15	30	15	11	11
w	1	2	3	3	6	3	5	5

$a'_1 = 1 \cdot 10 = 10$, $a''_1 = 2 \cdot 10 = 20$, а $a'''_1 = b(a_1) \cdot v(a_1) - (a'_1 + a''_1) = 60 - 30 = 30$. Останалите стойности и теглата се получават аналогично. Ако пуснем този пример на 0-1 KNAPSACK в knapsack solver, получаваме оптимално решение 75 при

$$(x'_1, x''_1, x'''_1, x'_2, x''_2, x'''_2, x'_3, x''_3) = (11110000)$$

Имената на променливите x съответстват по очевиден начин на предметите. Решението (11110000) означава вземане на a'_1 , a''_1 и a'''_1 , тоест $1 + 2 + 3 = 6$ предмета от вида a_1 , и a'_2 , тоест 1 предмет от вида a_2 : точно както в решението на око.

13.10 Interval Scheduling

Тази задача е известна още под името “ACTIVITY-SELECTION” [15, стр. 415]. Дадено е множество отворени интервали $A = \{a_1, \dots, a_n\}$, като $a_i = (s_i, f_i)$ и $s_i < f_i$, за $1 \leq i \leq n$. Ако мислим за интервалите като за дейности във времето, дейност a_i се характеризира с начално време s_i и крайно време f_i . Два различни интервала a_i и a_j са *съвместими*, ако имат празно сечение; тоест, ако $f_i < s_j$ или $f_j < s_i$. В противен случай те са *несъвместими*. Забележете,

че при $f_i = s_j$ интервалите a_i и a_j имат празно сечение, тъй като са отворени. Също така забележете, че напълно съвпадащи—и като начално, и като крайно време—интервали не може да има, защото идентифицираме всеки интервал с наредената двойка от началното му и крайното му време, а A е множество.

Задачата е, по дадено множество от интервали A , да се намери подмножество $A' \subseteq A$, такова че всеки два интервала в A' са съвместими и освен това $|A'|$ е максимална. Тази задача още се нарича и “задачата за безработния актьор” – актьор е без работа и получава възможност да се снима в n клипа, които са интервалите. Всеки клип има начално и крайно време, които са дадени, и заплащането за всеки клип е едно и също. Задачата на актьора е да избере максимален брой клипове, които не се “застъпват” два по два във времето, защото в даден момент той може да се снима в най-много един клип, и по този начин да получи максимално заплащане.

13.10.1 Алчно решение

Ще разгледаме четири идеи, всяка от които може да е в основата на алчен алгоритъм.

- ① **Първият наличен интервал** Актьорът взема клипа, чието начало е най-рано. След това всички клипове, които са засичат с този клип, отпадат като възможности. Измежду останалите клипове отново взема този с най-ранно начално време, и така нататък.

Тази идея не работи в смисъл, че има примери, в които дава неоптимални решения. Ето контрапример с илюстрация. Вземайки първата започваща работа (червения интервал), той ще изтърве четири други, а всяка от тях му дава еднакво заплащане.



- ② **Първо най-късият интервал** Актьорът взема клипа, който е най-къс. След това всички клипове, които са засичат с този клип, отпадат като възможности. Измежду останалите клипове отново взема този, който е най-къс, и така нататък.

Тази идея също не работи. Ето контрапример с илюстрация. Вземайки най-късия във времето ангажимент (червения интервал), той ще изтърве два други.



- ③ **Първо най-малко конфликти** Актьорът взема клипа, за който има най-малко други клипове, с които се засича. После тези, с които се засича, отпадат като възможности. Измежду останалите клипове отново взема този, който се засича с най-малко клипове, и така нататък.

Тази идея също не работи. Ето контрапример с илюстрация. Вземайки клипа с най-малко засичания (червения интервал; той се засича само с два други клипа), той губи възможността да вземе оптималното решение от четири клипа (в синьо).



④ НАЙ-РАННО ПРИКЛЮЧВАНЕ Актьорът взема клипа, който приключва най-рано. После тези, с които се засича, отпадат като възможности. Измежду останалите клипове отново взема този, който приключва най-рано, и така нататък.

Тази идея работи. Читателят лесно може да се убеди, че, следвайки тази идея, в примера от ① актьорът ще вземе четири клипа, в примера от ② актьорът ще вземе два клипа и в примера от ③ актьорът ще вземе четири клипа, като всички тези решения са оптимални. Но това, че тази идея работи, трябва да се докаже формално.

Теорема 41: НАЙ-РАННО ПРИКЛЮЧВАНЕ е оптимално за INTERVAL SCHEDULING

За всеки пример A на INTERVAL SCHEDULING, подмножеството $A' \subseteq A$, конструирано съгласно стратегията НАЙ-РАННО ПРИКЛЮЧВАНЕ, е максимално множество от два по два съвместими интервали.

Доказателство: Това, че стратегията НАЙ-РАННО ПРИКЛЮЧВАНЕ конструира подмножество от интервали, между които няма несъвместима двойка, е очевидно. Ще докажем, че конструираното множество е **максимално** множество от два по два съвместими интервали. Да допуснем противното. Нека $A = \{a_1, \dots, a_n\}$ е контрапример. Да си представим работата на алчен алгоритъм ALGIS за INTERVAL SCHEDULING, изграден върху стратегията НАЙ-РАННО ПРИКЛЮЧВАНЕ, върху вход A . Изходът A' не е максимално множество от два по два съвместими интервали, съгласно допускането. Нека $A' = \{a_{i_1}, \dots, a_{i_k}\}$, където $\{i_1, \dots, i_k\} \subset \{1, \dots, n\}$, и БОО нека интервалите “влизат” в A' точно в този ред: първо a_{i_1} , после a_{i_2} , и така нататък, накрая a_{i_k} .

Да съобразим, че, слагайки интервалите по този начин в A' , алгоритъмът ALGIS греши с поне един интервал. Какво означава това? По допускане, съществува $A'' \subseteq A$, такова че се състои от два по два съвместими интервали и $|A''| > |A'|$. Ще докажем, че е невъзможно $A' \subset A''$. Ако допуснем, че $A' \subset A''$, трябва A'' да се състои от интервалите в A' и още поне един интервал a_m , който е или вляво от най-левия интервал на A' , или някъде между интервалите на A' , или вдясно от най-десния интервал на A' . И трите случая са невъзможни – лесно се вижда, че ALGIS би сложил и a_m в A' . И така, задължително има интервал в A' , който не се съдържа в A'' . За всеки такъв интервал казваме че, че с него ALGIS е *сгрешил*.

Нека a_{i_r} е първият интервал, с който ALGIS е сгрешил. Тогава интервалите, сложени от ALGIS преди a_{i_r} , а именно $a_{i_1}, \dots, a_{i_{r-1}}$, се намират и в A'' . Нека интервалът в A'' , който се намира след $a_{i_{r-1}}$, се нарича a_t , като очевидно $t \neq i_r$. Ключовото наблюдение е, че краят f_{i_r} на a_{i_r} се намира вляво от края f_t на a_t или съвпада с него, но не може да е вдясно от a_t . Това е напълно очевидно предвид факта, че ALGIS е изграден върху стратегията НАЙ-РАННО ПРИКЛЮЧВАНЕ. Следната фигура илюстрира ситуацията:



Интервалите a_{i_r} и a_t може да не са разположени точно така един спрямо друг, но със сигурност $f_{i_r} \leq f_t$, в противен случай ALGIS щеше да сложи a_t , а не a_{i_r} в A' ; забележете, че както a_{i_r} , така и a_t са съвместими с всеки от $a_{i_1}, \dots, a_{i_{r-1}}$.

Веднага се вижда, че ALGIS не може да е сгрешил с a_{i_r} , защото, ако заменим a_t с a_{i_r} в A'' , то остава валидно решение:

$$A'': \overbrace{\quad \dots \quad}^{a_{i_1}} \overbrace{\quad \dots \quad}^{a_{i_{r-1}}} \overbrace{\quad \dots \quad}^{a_r}$$

С други думи, A'' няма какво да загуби, получавайки a_{i_r} наместо a_t , тъй като $f_{i_r} \leq f_t$. \square
Следният алгоритъм решава задачата, базирайки се на НАЙ-РАННО ПРИКЛЮЧВАНЕ.

INTERVAL SCHEDULING, GENERAL($A = \{a_1, \dots, a_n\}$, където $a_i = (s_i, f_i)$ и $s_i < f_i$ за $1 \leq i \leq n$)

- 1 сортирай A по f -стойностите, резултатът е $\langle a'_1, \dots, a'_n \rangle$
- 2 $A' \leftarrow \emptyset$
- 3 **for** $i \leftarrow 1$ **to** n
- 4 ако няма несъвместимост между a'_i и елемент от A' , добави a'_i към A'
- 5 **return** A'

Ключово за ефикасността на алгоритъма е ефикасността на проверката на ред 4. Няма нужда да проверяваме за съвместимост между a_i , от една страна, и всеки интервал в текущия A' , от друга. Напълно достатъчно е да проверяваме само дали a_i е съвместим с интервала, който е добавен последно в A' . Ако a_i е съвместим с него, то той е съвместим с всеки интервал в A' заради транзитивността на релацията $<$. Ако a_i не е съвместим с него, просто го прескачаме. И така, достатъчно е да помним в някаква променлива крайното време f^* на интервала, последно сложен в A' , и да сравняваме s_i с f^* . Ето подробностите.

INTERVAL SCHEDULING, DETAILED($A = \{a_1, \dots, a_n\}$, където $n \geq 1$, $a_i = (s_i, f_i)$ и $s_i < f_i$ за $1 \leq i \leq n$)

- 1 сортирай A по f -стойностите, резултатът е $\langle a'_1, \dots, a'_n \rangle$
- 2 $A' \leftarrow \{a'_1\}$
- 3 $f^* \leftarrow f'_1$
- 4 **for** $i \leftarrow 1$ **to** n
- 5 **if** $f^* \leq s'_i$
- 6 $A' \leftarrow A' \cup \{a'_i\}$
- 7 $f^* \leftarrow f'_i$
- 8 **return** A'

Коректността се обосновава с Теорема 41. Сложността по време е $\Theta(n \lg n)$ заради сортирането в началото. а сложността по памет е $\Theta(n)$.

13.10.2 Решение по схемата Динамично Програмиране

Ще усложним задачата. Сега всеки интервал a_i има тегло w_i , който е реално положително число. В частност е възможно $w_i = f_i - s_i$, но в общия случай w_i и $f_i - s_i$ са независими. Решението е подмножество от интервали, два по два съвместими, които имат максимално сумарно тегло. Тази задача не може да се реши ефикасно с леко обобщение на алчния алгоритъм от предишната подсекция. Ще видим решение, конструирано по схемата Динамично Програмиране, което има същата сложност по време $\Theta(n \lg n)$ като алчното решение, но е несравнено по-изтъчнено. Естествено, решението с динамично програмиране решава и задачата без тегла, като дава едно и също тегло на всеки интервал.

Нека множеството от интервалите е $A = \{a_1, \dots, a_n\}$, а сортираната им редица по f -стойности е $S = \langle a'_1, \dots, a'_n \rangle$. За ефикасно решение с динамично програмиране ни е необходимо да изчисляваме бързо за всеки интервал a'_i в S , интервала с най-голям номер в S ,

който е вляво от a'_i и е съвместим с a'_i . Да наречем функцията, реализираща това изображение, ϕ , като $\phi : \{1, \dots, n\} \rightarrow \{0, \dots, n-1\}$. За всяко $i \in \{1, \dots, n\}$, нека $C(i)$ е множеството от интервалите в S , които са вляво от i и са съвместими с i . Формалната дефиниция на ϕ е

$$\forall i \in \{1, \dots, n\} : \phi(i) = \begin{cases} \max \{j \in \{1, \dots, i-1\} \mid a'_j \text{ е съвместим с } a'_i\}, & \text{ако } C(i) \neq \emptyset \\ 0, & \text{ако } C(i) = \emptyset \end{cases}$$

Рекурсивната декомпозиция е по S . Оптималното решение за първите i елемента на S или ползва a'_i , или не го ползва. Ако не го ползва, то е същото като за първите $i-1$ елемента. Ако го ползва, стойността му е сумата от w'_i и оптималната стойност за първите $\phi(i)$ елемента.

Да допуснем, че имаме всички стойности на ϕ в масив $\phi[1, \dots, n]$. Схемата за изчисление е:

$$\begin{aligned} \text{opt}[0] &= 0 \\ \text{opt}[i] &= \max (\text{opt}[i-1], w'_i + \text{opt}[\phi[i]]), \text{ ако } i > 0 \end{aligned}$$

Следният итеративен алгоритъм решава ефикасно задачата.

WEIGHTED INTERVAL SCHEDULING ($A = \{a_1, \dots, a_n\}$, където $n \geq 1$, $a_i = (s_i, f_i)$, $s_i < f_i$ и a_i има тегло w_i , за $1 \leq i \leq n$)

```

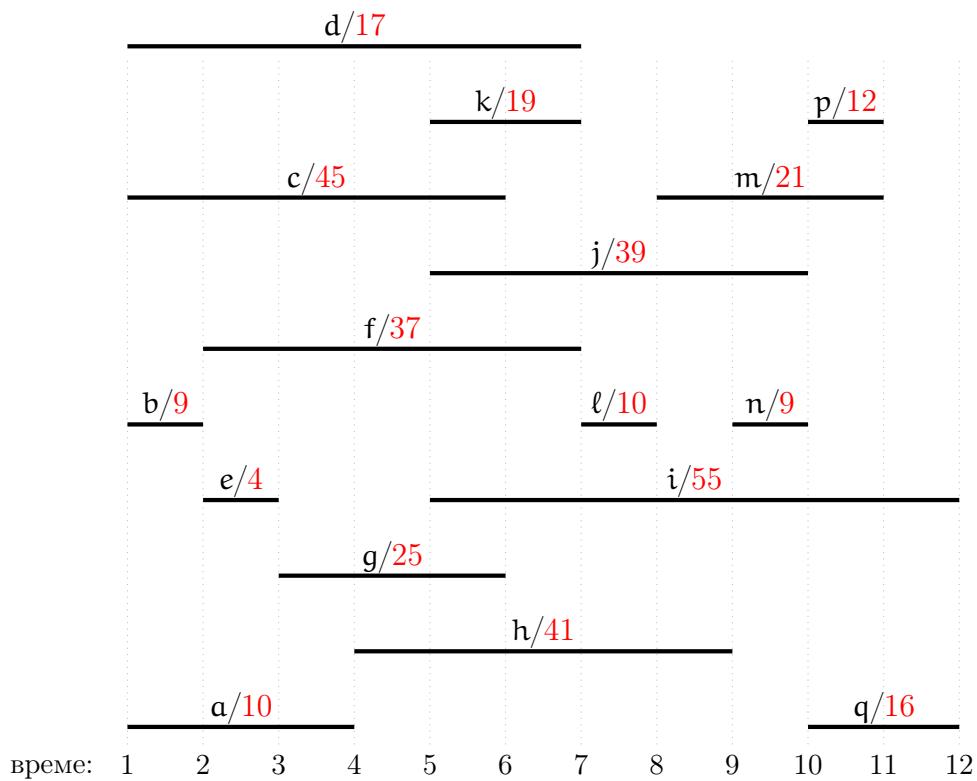
1  сортирай A по f-стойностите, резултатът е  $\langle a'_1, \dots, a'_n \rangle$ 
2  opt[0] ← 0
3  for i ← 1 to n
4      opt[i] ← max (opt[i-1],  $w'_i + \text{opt}[\phi[i]]$ )
5  return opt[n]

```

Коректността е очевидна, ако сме убедени в коректността на рекурсивната декомпозиция и схемата за изчисление. Да разгледаме сложността. Ако изчисляваме стойностите на масива $\phi[1, \dots, n]$ по наивния начин, като за всяко $\phi[i]$ се връщаме назад, може би чак до началото, за да видим кой е най-десният интервал в S , съвместим с i -ия, ще получим квадратичен алгоритъм само за пресмянето на $\phi[1, \dots, n]$ и оттам, квадратична сложност на целия алгоритъм. За щастие, стойностите на масива $\phi[1, \dots, n]$ може да се пресметнат във време $O(n \lg n)$ така: за всяко a'_i , с двоично търсене намираме максималното j , такова че $f'_j \leq s'_i$. Щом четем в момента a'_i , ние разполагаме с s'_i , а това f'_j ще намерим във време $O(\lg i)$, защото разполагаме със сортираната по f-стойности редица от интервалите.

Щом можем да построим $\phi[1, \dots, n]$ във време $O(n \lg n)$, очевидно имаме алгоритъм със сложност по време $\Theta(n \lg n)$.

Да разгледаме пример. Интервалите са именувани a, \dots, q (без име "o"), а теглата са написани до имената в червено.



Можете ли да откриете оптимално решение на око? Авторът на записките признава, че не може. Да подходим алгоритмично. Ето редицата от интервалите, сортирана по крайни времена, с индексите на интервалите в нея, написани отдолу:

b	e	a	g	c	f	k	d	l	h	n	j	m	p	q	i
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Ето масивът $\phi[1, \dots, 16]$:

x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\phi(x)$	0	1	0	2	0	1	3	0	8	3	10	3	9	12	12	3

Ето и изчисленията за $\text{opt}[16]$:

- $\text{opt}[1] = \max(0, 9 + \text{opt}[0]) = 9.$ ✓
- $\text{opt}[2] = \max(9, 4 + \text{opt}[1]) = 13.$ ✓
- $\text{opt}[3] = \max(13, 4 + \text{opt}[0]) = 13.$
- $\text{opt}[4] = \max(13, 25 + \text{opt}[2]) = 38.$ ✓
- $\text{opt}[5] = \max(38, 45 + \text{opt}[0]) = 45.$ ✓
- $\text{opt}[6] = \max(45, 37 + \text{opt}[1]) = 46.$ ✓
- $\text{opt}[7] = \max(46, 19 + \text{opt}[3]) = 46.$
- $\text{opt}[8] = \max(46, 17 + \text{opt}[0]) = 46.$
- $\text{opt}[9] = \max(46, 10 + \text{opt}[8]) = 56.$ ✓
- $\text{opt}[10] = \max(56, 41 + \text{opt}[3]) = 56.$
- $\text{opt}[11] = \max(56, 9 + \text{opt}[10]) = 65.$ ✓
- $\text{opt}[12] = \max(65, 39 + \text{opt}[3]) = 65.$
- $\text{opt}[13] = \max(65, 21 + \text{opt}[9]) = 77.$ ✓
- $\text{opt}[14] = \max(77, 12 + \text{opt}[12]) = 77.$

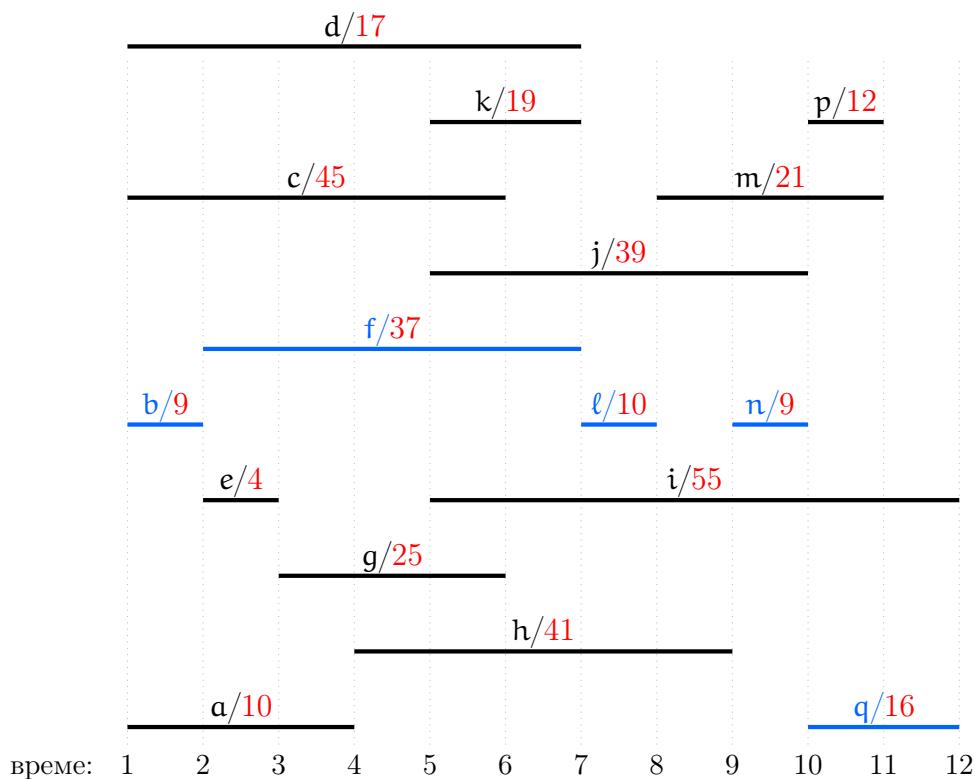
$$\text{opt}[15] = \max(77, 16 + \text{opt}[12]) = 81. \checkmark$$

$$\text{opt}[16] = \max(81, 55 + \text{opt}[3]) = 81.$$

И така, оптималното решение има стойност 81.

Ако искаме да получим и самото оптимално решение, а не само стойността му, можем да постъпим така. В изчисленията за $\text{opt}[16]$, с червен тик е отбелоязан всеки ред, в който стойността е получена от втория аргумент на \max , а именно $w'_i + \text{opt}[\phi[i]]$ на ред 5. Иначе казано, това са редовете, в които има увеличение спрямо предишния ред. Тръгвайки от последния ред, връщаме се назад до първия тик, който е на $\text{opt}[15]$. Виждаме, че q участва в оптималното решение и освен това $\text{opt}[15]$ е получен като $16 + \text{opt}[12] = 81$. Отиваме на реда на $\text{opt}[12]$. Той няма тик, така че се връщаме до първия ред назад с тик, който е реда на $\text{opt}[11]$. Тогава n участва в оптималното решение и освен това $\text{opt}[11]$ е получен като $9 + \text{opt}[10] = 65$. Отиваме на реда на $\text{opt}[10]$. Той няма тик, така че се връщаме до първия ред назад с тик, който е реда на $\text{opt}[9]$. Тогава ℓ участва в оптималното решение и освен това $\text{opt}[9]$ е получен като $10 + \text{opt}[8] = 56$. Отиваме на реда на $\text{opt}[8]$. Той няма тик, така че се връщаме до първия ред назад с тик, който е редът на $\text{opt}[6]$. Тогава f участва в оптималното решение и освен това $\text{opt}[6]$ е получен като $37 + \text{opt}[1] = 46$. Отиваме на реда на $\text{opt}[1]$, добавяме b към решението и сме готови. Решението се състои от q, n, ℓ, f и b , сумата от теглата на които наистина е $16 + 9 + 10 + 37 + 9 = 81$. Читателят лесно може да съобрази как да кодира тази идея в алгоритъма.

Ето и визуализация на оптималното решение. Интервалите от оптималното решение са в синьо.



13.11 Longest Increasing Subsequence

Дадена е числова редица $S = \langle a_1, a_2, \dots, a_n \rangle$. БОО, нека числата са две по две различни. *Поредица в S* е всяка редица $\langle a_{i_1}, a_{i_2}, \dots, a_{i_k} \rangle$, такава че $1 \leq i_1 < i_2 < \dots < i_k \leq n$. *Растяща*

поредица в S е всяка поредица $\langle a_{i_1}, a_{i_2}, \dots, a_{i_k} \rangle$, такава че $a_{i_1} < a_{i_2} < \dots < a_{i_k}$. Задачата е, при дадена чисрова редица, да се намери максимална растяща поредица; в лекия вариант, да се намери само дължината на максимална растяща поредица.

Забележете, че “поредица в S ” е редица от елементи, които **не са непременно съседни** в S . Поредица, чито елементи са съседни, ще наричаме *непрекъсната*, на английски *contiguous*. Разговорно казано, елементите на поредицата в общия случай са пръснати в S . Ето пример.

$$S = \langle 14, 9, -5, 11, -2, 0, 3, -4, 19, 6, 8, 22, 10 \rangle$$

Поредица е, примерно (в червено):

$$S = \langle 14, \textcolor{red}{9}, \textcolor{red}{-5}, 11, -2, 0, \textcolor{red}{3}, -4, \textcolor{red}{19}, 6, 8, 22, \textcolor{red}{10} \rangle$$

Следната поредица е непрекъсната (в лилаво):

$$S = \langle 14, 9, -5, \textcolor{magenta}{11}, \textcolor{magenta}{-2}, \textcolor{magenta}{0}, \textcolor{magenta}{3}, \textcolor{magenta}{-4}, 19, 6, 8, 22, 10 \rangle$$

Растяща поредица, максимална при това, е (в синьо):

$$S = \langle 14, 9, \textcolor{blue}{-5}, 11, \textcolor{blue}{-2}, \textcolor{blue}{0}, \textcolor{blue}{3}, -4, 19, \textcolor{blue}{6}, \textcolor{blue}{8}, 22, \textcolor{blue}{10} \rangle$$

Максимална непрекъсната растяща поредица е (в зелено):

$$S = \langle 14, 9, -5, 11, \textcolor{green}{-2}, \textcolor{green}{0}, \textcolor{green}{3}, -4, 19, 6, 8, 22, 10 \rangle$$

Намирането на максимална непрекъсната растяща поредица е тривиална задача. Има очевидно решение в линейно време с едно премитане, но дори решението с груба сила не е невъзможно бавно, понеже непрекъснатите поредици са $\Theta(n^2)$ на брой. Намирането на максимална растяща поредица обаче е нетривиална задача. Поредиците без ограничението за напрекъснатост са много повече—а именно, $2^n - 1$, ако не броим празната поредица—и подходът с груба сила е безнадежден. В примера със синята поредица, когато разглеждаме -5 и отиваме надясно, как да разберем, че трябва да прескочим 11, но да не прескочим -2 ?

13.11.1 Решение по схемата Дин. Прогр. със сложност $\Theta(n^2)$

Първо решаваме задачата в лекия вариант, в който се иска само стойността на оптимално решение. Изглежда смислено да характеризираме оптималното решение като поредица, завършваща с някакъв елемент a_k , където $1 \leq k \leq n$.

Спрямо тази характеризация, нека да мислим за рекурсивна декомпозиция. Нека

$$\begin{aligned} S^1 &= \langle a_1 \rangle \\ S^2 &= \langle a_1, a_2 \rangle \\ S^3 &= \langle a_1, a_2, a_3 \rangle \\ &\dots \\ S^n &= \langle a_1, a_2, \dots, a_n \rangle = S \end{aligned}$$

Нека ℓ_k е дължината на максимална растяща поредица в S^k , завършваща на a_k , за $1 \leq k \leq n$. Ако имаме тези ℓ_k , то ние сме готови! Тогава отговорът е $\max\{\ell_k \mid 1 \leq k \leq n\}$, тъй като всяка максимална растяща поредица завършва на някой елемент.

Как да получим ℓ_k от $\ell_{k-1}, \dots, \ell_1$? Само числата $\ell_{k-1}, \dots, \ell_1$ не са достатъчни, но комбинациите от ℓ_j и a_j , за $j \in \{1, \dots, k-1\}$ са достатъчни. Интересуват ни само тези a_j , за които $a_j < a_k$, ако има такива. Ако има такива, a_k е максимумът от техните ℓ -стойности, плюс едно. Ако няма такива, то a_k е единица. Нека

$$\forall k \in \{1, \dots, n\} : R_k = \{j \in \{1, \dots, k-1 \mid a_j < a_k\}$$

Тогава, за $1 \leq k \leq n$:

$$\ell_k = \begin{cases} \max \{\ell_j \mid j \in R_k\} + 1, & \text{ако } R_k \neq \emptyset \\ 1, & \text{ако } R_k = \emptyset \end{cases}$$

Превръщането на рекурсивната декомпозиция в алгоритъм е тривиално. Забележете, че $R_1 = \emptyset$, така че $\ell_1 = 1$ като начално условие.

Ще покажем как да решим по-тежкия вариант на задачата, в който се иска не просто дължината на максимална растяща поредица, а и някоя максимална растяща поредица. За всяка позиция $k \in \{1, \dots, n\}$ записваме позицията π_k на елемента—ако има такъв—който предшества a_k в максимална растяща поредица, завършваща на a_k . Ако няма такъв, нека $\pi_k = \text{nil}$. Ето алгоритъм, който реализира тази идея.

LONGEST INCREASING SUBSEQUENCE($\langle a_1, a_2, \dots, a_n \rangle$)

```

1   $\ell[1] \leftarrow 1$ 
2   $\pi[1] \leftarrow \text{nil}$ 
3  for  $k \leftarrow 2$  to  $n$ 
4     $\ell[k] \leftarrow 1$ 
5     $\pi[k] \leftarrow \text{nil}$ 
6    for  $j \leftarrow 1$  to  $k-1$ 
7      if  $\ell[k] < \ell[j] + 1$  and  $a[k] > a[j]$ 
8         $\ell[k] \leftarrow \ell[j] + 1$ 
9         $\pi[k] \leftarrow j$ 
10   return ( $\max_{1 \leq k \leq n} \{\ell[k]\}$ ,  $\pi$ )

```

Коректността е очевидна. Сложността е $\Theta(n^2)$. Както ще видим, има по-ефикасен алгоритъм за тази задача, но квадратичната сложност по време не е непоносима. Ето примерна работа на алгоритъма върху $S = \langle 14, 9, -5, 11, -2, 0, 3, -4, 19, 6, 8, 22, 10 \rangle$.

k	1	2	3	4	5	6	7	8	9	10	11	12	13
$S[k]$	14	9	-5	11	-2	0	3	-4	19	6	8	22	10
$\ell[k]$	1	1	1	2	2	3	4	2	1	5	6	1	7
$\pi[k]$	nil	nil	nil	3	3	5	6	3	nil	7	10	nil	11

13.11.2 Решение по схемата Дин. Прогр. със сложност $\Theta(n \lg n)$

13.11.3 PATIENCESORT

Лекция 14

Алгоритмична неподатливост. NP-пълнота и NP-трудност.

Лекция 15

SAT и теоремата на Cook.

Лекция 16

Основни НР-пълни задачи.

Лекция 17

Заобикаляне на неподатливостта:
апроксимиране и параметризиране.

Благодарности

Благодарности на **Добрин Цветков** за откритите и коригирани грешки. Благодарности на **Емилиян Рогачев** за детайлното четене плюс откриване и коригиране на купища грешки. Благодарности на **Теодор Грозданов** за откритата и коригирана грешка в алгоритъма DUMMY–GOLDBACH на стр. [21](#). Благодарности на **Борис Дишов** за откритата и коригирана грешка в имплементацията на QUICK SORT. Благодарности на **Йордан Петров** за откритата и коригирана грешка в доказателството за коректност на алгоритъма НАРАСТВАНЕ С ЕДИНИЦА на стр. [23](#), както и на грешки във формулировките на Лема [6](#) и Лема [9](#). Благодарности на **Костадин Гаров** за откритата грешка в доказателството на Лема [18](#). Благодарности на **Ясен Алексиев** за откритата грешка в доказателството на Теорема [17](#). Благодарности на **доктор Добромир Кралчев** за посочването на факта, че отсъствието на малко-о отношение между две функции не влече голямо-Омега отношение между тях (вж. Допълнение [26](#)). Благодарности на **Иван-Асен Чакъров** за откритите грешки в рекапитулацията на графиките. Благодарности на **Ива Петрова Караджова** за откритите грешки във формулировката на Наблюдение [29](#).

Библиография

- [1] Alfred V. Aho, John E. Hopcroft и Jeffrey Ullman. *Data Structures and Algorithms*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0201000237.
- [2] Martin Aigner и Gnter M. Ziegler. *Proofs from THE BOOK*. 4th. Springer Publishing Company, Incorporated, 2009. ISBN: 3642008550.
- [3] Mohamad Akra и Louay Bazzi. “On the Solution of Linear Recurrence Equations”. B: *Computational Optimization and Applications* 10.2 (1998), c. 195—210. ISSN: 0926-6003.
- [4] V. K. Balakrishnan. *Introductory Discrete Mathematics*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1991. ISBN: 0130399426.
- [5] Edward G. Belaga и Maurice Mignotte. “Walking Cautiously Into the Collatz Wilderness: Algorithmically, Number Theoretically, Randomly”. B: *Discrete Mathematics & Theoretical Computer Science* DMTCS Proceedings vol. AG, Fourth Colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities (ян. 2006). Достъпна онлайн на <https://dmtcs.episciences.org/3512>.
- [6] Richard Bellman. *Eye of the Hurricane: An Autobiography*. Series in modern applied mathematics. World Scientific, 1984. ISBN: 9789971966010.
- [7] C. H. Bennett. “Logical Reversibility of Computation”. B: *IBM J. Res. Dev.* 17.6 (ноем. 1973). Достъпна онлайн на https://www.math.ucsd.edu/~sbuss/CourseWeb/Math268_2013W/Bennett_Reversibility.pdf, c. 525—532. ISSN: 0018-8646. DOI: [10.1147/rd.176.0525](https://doi.org/10.1147/rd.176.0525). URL: <http://dx.doi.org/10.1147/rd.176.0525>.
- [8] Jon Bentley. “Programming Pearls: Little Languages”. B: *Commun. ACM* 29.8 (авг. 1986), c. 711—721. ISSN: 0001-0782. DOI: [10.1145/6424.315691](https://doi.org/10.1145/6424.315691). URL: <http://doi.acm.org/10.1145/6424.315691>.
- [9] Jon Louis Bentley, Dorothea Haken и James B. Saxe. “A general method for solving divide-and-conquer recurrences”. B: 12 (1980). Достъпна онлайн на <https://apps.dtic.mil/cgi-bin/tr/fulltext/u2/a064294.pdf>, c. 36—44.
- [10] Jon Louis Bentley и Michael Ian Shamos. “Divide-and-Conquer in Multidimensional Space”. B: *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*. STOC ’76. Достъпна онлайн на <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.366.9611&rep=rep1&type=pdf>. Hershey, Pennsylvania, USA: Association for Computing Machinery, 1976, c. 220—230. ISBN: 9781450374149. DOI: [10.1145/800113.803652](https://doi.org/10.1145/800113.803652). URL: <https://doi.org/10.1145/800113.803652>.
- [11] Garrett Birkhoff. *Lattice Theory*. 3rd. American Mathematical Society, 1967.
- [12] Andreas Blass и Yuri Gurevich. “Algorithms: A quest for absolute definitions”. B: *Bulletin of the European Association for Theoretical Computer Science* (2003). Достъпна онлайн на <http://research.microsoft.com/en-us/um/people/gurevich/Opera/164.pdf>.

- [13] Manuel Blum и др. “Linear Time Bounds for Median Computations”. В: *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*. STOC '72. Достъпна онлайн на <http://citeseervx.ist.psu.edu/viewdoc/download?doi=10.1.1.309.9712&rep=rep1&type=pdf>. Denver, Colorado, USA: Association for Computing Machinery, 1972, с. 119—124. ISBN: 9781450374576. DOI: [10.1145/800152.804904](https://doi.org/10.1145/800152.804904).
- [14] Manuel Blum и др. “Time bounds for selection”. В: *Journal of Computer and System Sciences* 7.4 (1973). Достъпна онлайн на <https://www.sciencedirect.com/science/article/pii/S0022000073800339>, с. 448—461. ISSN: 0022-0000. DOI: [https://doi.org/10.1016/S0022-0000\(73\)80033-9](https://doi.org/10.1016/S0022-0000(73)80033-9).
- [15] Thomas H. Cormen и др. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 9780262033848.
- [16] Thomas T. Cormen, Charles E. Leiserson и Ronald L. Rivest. *Introduction to Algorithms*. 1st edition. Cambridge, MA, USA: MIT Press, 1990. ISBN: 0-262-03141-8.
- [17] Robert W. Floyd. “Algorithm 245: Treesort”. В: *Communications of the ACM* 7.12 (дек. 1964). Достъпна онлайн на <http://doi.acm.org/10.1145/355588.365103>, с. 701—. ISSN: 0001-0782. DOI: [10.1145/355588.365103](https://doi.org/10.1145/355588.365103).
- [18] Robert W. Floyd. “Algorithm 97: Shortest Path”. В: *Commun. ACM* 5.6 (юни 1962). Достъпна онлайн на <https://dl.acm.org/doi/10.1145/367766.368168>, с. 345. ISSN: 0001-0782. DOI: [10.1145/367766.368168](https://doi.org/10.1145/367766.368168).
- [19] Lance Fortnow. “The Status of the P Versus NP Problem”. В: *Commun. ACM* 52.9 (септ. 2009). Достъпна онлайн на <http://dl.acm.org/citation.cfm?id=1562186>, с. 78—86. ISSN: 0001-0782. DOI: [10.1145/1562164.1562186](https://doi.org/10.1145/1562164.1562186). URL: <http://doi.acm.org/10.1145/1562164.1562186>.
- [20] M. R. Garey и D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. First Edition. W. H. Freeman, 1979. ISBN: 0716710455. URL: <http://www.amazon.com/Computers-Intractability-NP-Completeness-Mathematical-Sciences/dp/0716710455>.
- [21] Ronald L. Graham, Donald E. Knuth и Oren Patashnik. *Concrete Mathematics*. second. Addison-Wesley, 1994. ISBN: 0-201-55802-5.
- [22] T.L. Heath и J.L. Heiberg. *Books 3-9. The Thirteen Books of Euclid's Elements*. Достъпна онлайн на http://www.wilbourhall.org/pdfs/Heath_Euclid_II.pdf. The University Press, 1908.
- [23] C. A. R. Hoare. “Algorithm 63: Partition”. В: *Commun. ACM* 4.7 (юли 1961), с. 321—. ISSN: 0001-0782. DOI: [10.1145/366622.366642](https://doi.org/10.1145/366622.366642). URL: <http://doi.acm.org/10.1145/366622.366642>.
- [24] C. A. R. Hoare. “Algorithm 64: Quicksort”. В: *Commun. ACM* 4.7 (юли 1961), с. 321—. ISSN: 0001-0782. DOI: [10.1145/366622.366644](https://doi.org/10.1145/366622.366644). URL: <http://doi.acm.org/10.1145/366622.366644>.
- [25] C. A. R. Hoare. “Quicksort”. В: *The Computer Journal* 5.1 (1962), с. 10—16. DOI: [10.1093/comjnl/5.1.10](https://doi.org/10.1093/comjnl/5.1.10). eprint: <http://comjnl.oxfordjournals.org/cgi/reprint/5/1/10.pdf>. URL: <http://comjnl.oxfordjournals.org/cgi/content/abstract/5/1/10>.
- [26] L. Hogben. *Handbook of Linear Algebra, Second Edition*. Discrete Mathematics and Its Applications. Taylor & Francis, 2013. ISBN: 9781466507289.

- [27] Hsien-Kuei Hwang и Jean-Marc Steyaert. *On the Number of Heaps and the Cost of Heap Construction*. Достъпна онлайн на <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.9150&rep=rep1&type=pdf>. 2001.
- [28] R. Impagliazzo. “A Personal View of Average-case Complexity”. В: *Proceedings of the 10th Annual Structure in Complexity Theory Conference (SCT'95)*. SCT '95. Достъпна онлайн на <http://www.cs.ucsd.edu/users/russell/average.ps>. Washington, DC, USA: IEEE Computer Society, 1995, с. 134—. ISBN: 0-8186-7052-5. URL: <http://dl.acm.org/citation.cfm?id=829497.829786>.
- [29] Alan S. Henry John N. Crossley. “Thus spake al-Khwārizmī: A translation of the text of Cambridge University Library Ms. II.vi.5”. В: *Historia Mathematica* 17 (2 1990). Достъпна онлайн на <http://www.sciencedirect.com/science/article/pii/031508609090048I>, с. 103—131.
- [30] David S Johnson. “The NP-completeness column: An ongoing guide”. В: *Journal of Algorithms* 8.2 (1987), с. 285—303. ISSN: 0196-6774. DOI: [https://doi.org/10.1016/0196-6774\(87\)90043-5](https://doi.org/10.1016/0196-6774(87)90043-5). URL: <http://www.sciencedirect.com/science/article/pii/0196677487900435>.
- [31] A. B. Kahn. “Topological Sorting of Large Networks”. В: *Commun. ACM* 5.11 (ноем. 1962). Статията е свободно достъпна на <https://dl.acm.org/doi/pdf/10.1145/368996.369025?download=true>, с. 558—562. ISSN: 0001-0782. DOI: <10.1145/368996.369025>.
- [32] Edward C. Kirby и др. “What Kirchhoff Actually did Concerning Spanning Trees in Electrical Networks and its Relationship to Modern Graph-Theoretical Work”. В: *Croatica Chemica Acta* 89.4 (2016). Достъпна онлайн на <https://hrcak.srce.hr/file/255139>, с. 403—417. DOI: <10.5562/cca2995>.
- [33] G. Kirchhoff. “On the Solution of the Equations Obtained from the Investigation of the Linear Distribution of Galvanic Currents”. В: *IRE Transactions on Circuit Theory* 5.1 (1958), с. 4—7.
- [34] G. Kirchhoff. “Ueber die Auflösung der Gleichungen, auf welche man bei der Untersuchung der linearen Vertheilung galvanischer Ströme geführt wird”. В: *Annalen der Physik* 148.12 (1847). Достъпна чрез платен достъп или оторизирана институция на <https://onlinelibrary.wiley.com/doi/abs/10.1002/andp.18471481202>, с. 497—508. DOI: <10.1002/andp.18471481202>.
- [35] Donald E. Knuth. “Big Omicron and big Omega and big Theta”. В: *SIGACT News* 8.2 (1976). Достъпна онлайн на http://www.phil.uu.nl/datastructuren/10-11/knuth_big_omicron.pdf, с. 18—24. URL: <http://doi.acm.org/10.1145/1008328.1008329>.
- [36] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-89683-4.
- [37] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-89684-2.
- [38] Donald E. Knuth. *The Art of Computer Programming, Volume 3 (2Nd Ed.): Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0-201-89685-0.
- [39] Mark Korenblit и Vadim E. Levit. “A One-Vertex Decomposition Algorithm for Generating Algebraic Expressions of Square Rhomboids”. В: *CoRR* abs/1211.1661 (2012). Достъпна онлайн на <http://arxiv.org/pdf/1211.1661v1.pdf>.

- [40] J C Lagarias. “The 3x+1 problem: An annotated bibliography (1963–1999)”. Достъпна онлайн на <https://arxiv.org/abs/math/0608208>. 2012.
- [41] J C Lagarias. “The 3x+1 Problem: An Annotated Bibliography, II (2000-2009)”. Достъпна онлайн на <https://arxiv.org/abs/math/0309224>. 2011.
- [42] Rolf Landauer. “Irreversibility and heat generation in the computing process”. В: *IBM Journal of Research and Development* 5 (1961), с. 183–191. URL: <http://domino.watson.ibm.com/tchjr/journalindex.nsf/0/8a9d4b4e96887b8385256bfa0067fba2?OpenDocument>.
- [43] Leighton. *Note on Better Master Theorems for Divide-and-Conquer Recurrences*. Достъпна онлайн <http://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf>. 1996.
- [44] Seth Lloyd. “Ultimate physical limits to computation”. В: *Nature* 406 (6799 авг. 2000). Достъпна онлайн на <https://arxiv.org/abs/quant-ph/9908043>, с. 1047–1054. DOI: [10.1038/35023282](https://doi.org/10.1038/35023282).
- [45] Krasimir Manev. *Uvod v Diskretnata Matematika*. fifth. KLMN – Krasimir Manev, 2012.
- [46] Zohar Manna. *Mathematical Theory of Computation*. Dover Publications, Incorporated, 2003.
- [47] Silvano Martello и Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. USA: John Wiley & Sons, Inc., 1990. ISBN: 0471924202.
- [48] E.F. Moore. *The Shortest Path Through a Maze*. Bell Telephone System. Technical publications. monograph. Bell Telephone System., 1959. URL: <https://books.google.com/books?id=IVZBHAAACAAJ>.
- [49] tree (data structure). National Institute of Standards and Technology’s web site. Достъпна онлайн на <http://xlinux.nist.gov/dads/HTML/tree.html>.
- [50] k-ary tree. National Institute of Standards and Technology’s web site. Достъпна онлайн на [https://xlinux.nist.gov/dads/HTML/perfectKaryTree.html](http://xlinux.nist.gov/dads/HTML/perfectKaryTree.html).
- [51] Christos H. Papadimitriou. *Computational Complexity*. Chichester, UK: John Wiley и Sons Ltd., с. 260–265. ISBN: 0-470-86412-5.
- [52] Christos H. Papadimitriou и Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. USA: Prentice-Hall, Inc., 1982. ISBN: 0131524623.
- [53] M. Pirlot и P. Vincke. *Semiorders: Properties, Representations, Applications*. Theory and Decision Library B. Springer Netherlands, 1997. ISBN: 9780792346173. URL: <https://books.google.com/books?id=rXroQjbQHn0C>.
- [54] Dan Romik. “Stirling’s Approximation for n!: The Ultimate Short Proof?” В: *The American Mathematical Monthly* 107.6 (2000). Достъпна онлайн на <https://www.math.ucdavis.edu/~romik/data/uploads/papers/stirling.pdf>, с. 556–557.
- [55] John E. Savage. *Models of Computation: Exploring the Power of Computing*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201895390.
- [56] Byron Schmuland. *Shouting Factorials!* Достъпна онлайн на <http://www.math.ualberta.ca/pi/issue7/page10-12.pdf>.
- [57] Bruce Schneier. *Applied Cryptography (2Nd Ed.): Protocols, Algorithms, and Source Code in C*. New York, NY, USA: John Wiley & Sons, Inc., 1995. ISBN: 0-471-11709-9.
- [58] A. Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency*. Springer, 2003.

- [59] Alexander Schrijver. *On the History of the Shortest Path Problem*. Достъпна онлайн на https://www.math.uni-bielefeld.de/documenta/vol-ismp/32_schrijver-alexander-sp.pdf. 2012.
- [60] Robert Sedgewick и Philippe Flajolet. *An introduction to the analysis of algorithms*. Addison-Wesley-Longman, 1996, с. I—XV, 1—492. ISBN: 978-0-201-40009-0.
- [61] Robert Sedgewick и Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011, с. I—XII, 1—955. ISBN: 978-0-321-57351-3.
- [62] Michael Sipser. *Introduction to the theory of computation: second edition*. 2-е изд. Boston: PWS Pub., 2006. ISBN: 978-0534950972.
- [63] Steven S. Skiena. *The Algorithm Design Manual*. 2nd. Springer Publishing Company, Incorporated, 2008. ISBN: 9781848000698.
- [64] R.P. Stanley. *Algebraic Combinatorics: Walks, Trees, Tableaux, and More*. Undergraduate Texts in Mathematics. Авторът е предоставил свободно копие на книгата, без упражненията, на адрес <http://www-math.mit.edu/~rstan/algcomb/algcomb.pdf>. Springer New York, 2013. ISBN: 9781461469988.
- [65] Williard Stone. “Abacists versus Algorists”. В: *Journal of Accounting Research Journal of Accounting Research* 10.2 (1972). Достъпна онлайн на <https://www.jstor.org/stable/2490013>, с. 345—350. DOI: [10.2307/2490013](https://doi.org/10.2307/2490013).
- [66] Robert Endre Tarjan. “Edge-Disjoint Spanning Trees and Depth-First Search”. В: *Acta Informatica* 6.2 (юни 1976). По-стара версия на статията е свободно достъпна като technical report на <http://i.stanford.edu/pub/cstr/reports/cs/tr/74/455/CS-TR-74-455.pdf>, с. 171—185. ISSN: 0001-5903. DOI: [10.1007/BF00268499](https://doi.org/10.1007/BF00268499).
- [67] Alan M. Turing. “On Computable Numbers, with an application to the Entscheidungsproblem”. В: *Proceedings of the London Mathematical Society*. 2-я пор. 42 (1936). Достъпна онлайн на http://www.dna.caltech.edu/courses/cs129/caltech_restricted/Turing_1936_IBID.pdf, с. 230—265.
- [68] Hui Wang и др. “Boson Sampling with 20 Input Photons and a 60-Mode Interferometer in a 10^{14} state spaces”. В: *Physical Review Letters* 123.25 (дек. 2019). Достъпна онлайн на <https://arxiv.org/abs/1910.09930>. ISSN: 1079-7114. DOI: [10.1103/physrevlett.123.250503](https://doi.org/10.1103/physrevlett.123.250503).
- [69] Y. Wang. *The Goldbach Conjecture*. Series in pure mathematics. World Scientific, 2002. ISBN: 9789812776600. URL: <https://books.google.com/books?id=VAY9nTreXkcC>.
- [70] Stephen Warshall. “A Theorem on Boolean Matrices”. В: *J. ACM* 9.1 (ян. 1962). Достъпна онлайн на <https://dl.acm.org/doi/10.1145/321105.321107>, с. 11—12. ISSN: 0004-5411. DOI: [10.1145/321105.321107](https://doi.org/10.1145/321105.321107).
- [71] Lutz M. Wegner. *Sorting – The Turku Lectures*. Достъпна онлайн на http://tucs.fi/publications/attachment.php?fname=bWegner_LutzMx14a.full.pdf. 2014.
- [72] Andrew John Wiles. “Modular Elliptic Curves and Fermat’s Last Theorem”. В: *ANNALS OF MATH* 141 (1995). Достъпна онлайн на <http://math.stanford.edu/%7Elekheng/flt/wiles-small.pdf>, с. 141.
- [73] J. W. J. Williams. “Algorithm 232: Heapsort”. В: *Communications of the ACM* 7.6 (1964), с. 347—348.
- [74] Gerhard J. Woeginger и Zhongliang Yu. “On the Equal-Subset-Sum Problem”. В: *Inf. Process. Lett.* 42.6 (юли 1992), с. 299—302. ISSN: 0020-0190. DOI: [10.1016/0020-0190\(92\)90226-L](https://doi.org/10.1016/0020-0190(92)90226-L).