ЛЕКЦИЯ 1

ТЕМА: Введение. Алгоритмы. Сортировка вставками. Псевдокод. Анализ алгоритмов

Процесс создания компьютерной программы для решения какой-либо практической задачи состоит из нескольких этапов:

- формализация и создание ТЗ на исходную задачу;
- разработка алгоритма решения задачи;
- написание, тестирование, отладка, документирование программы;
- получение решения исходной задачи путём выполнения законченной программы.

Цель нашего курса заключается в построении решения в форме алгоритма, состоящего из конечной последовательности инструкций, каждая из которых имеет чёткий смысл и может быть выполнена с конечными вычислительными затратами и за конечное время.

Составной частью процесса компьютерного решения задач являются вычисления и оценки времени выполнения алгоритмов (временная сложность алгоритмов), поскольку при решении задач прогрессирующе больших размеров особое значение имеет временная сложность выбранного алгоритма. А не возможности новых поколений вычислительных средств.

Алгоритмы

Алгоритм (algorithm) — это формально описанная вычислительная процедура, получающая **исходные данные** (input), называемые также входом алгоритма или его аргументом, и выдающая **результат** вычислений на выход (output).

Требования к алгоритмам:

1. Должен содержать конечное количество элементарно выполнимых предписаний, то есть удовлетворять **требованию конечности записи**.
2. Должен выполнять конечное количество шагов при решении задачи. То есть удовлетворять **требованию конечности действий**.
3. Должен быть единым для любых допустимых исходных данных, то есть удовлетворять **требованию универсальности.**
4. Должен приводить к правильному по отношению к поставленной задаче решению. То есть удовлетворять **требованию правильности**.

Свойства алгоритма:

1. **Дискретность.** Это свойство состоит в том, что алгоритм должен представлять процесс решения задачи как последовательное выполнение простых шагов. При этом для выполнения алгоритма требуется конечный отрезок времени, то есть преобразование исходных данных в результат осуществляется во времени дискретно.
2. **Определенность.** Каждое действие алгоритма должно быть чётким, однозначным.
3. **Результативность.** Алгоритм должен приводить к решению за конечное число шагов.
4. **Массовость.** Алгоритм решения задачи разрабатывается в общем виде. То есть он должен быть применим для некоторого класса задач, различающихся лишь исходными данными.
5. **Правильность**. Алгоритм считается **правильным**, если на любом допустимом (для данной задачи) входе он заканчивает работу и выдаёт результат, удовлетворяющий требованиям задачи. В этом случае говорят, что алгоритм решает данную вычислительную задачу.

   Неправильный алгоритм может (для некоторого входа) вовсе не остановиться или дать неправильный результат.

Способы описания алгоритмов:

1. **Словесный**, записи на естественном языке, описание словами последовательности выполнения алгоритма.
2. **Формульно-словесный**. Аналогично словесному способу, плюс параллельная демонстрация используемых формул.
3. **Графический**, с помощью блок – схем.
4. **Программный**, тексты на языках программирования.
5. **Псевдокоды** (полуформальное описание включает словесный и программный способы).

Сортировка вставками

```
Insertion-Sort(A)
1 for j←2 to length[A]
2       do key←A[j]
3    ›добавить A[j] к отсортированной части A[1..j-1]
4          i←j-1
5          while i>0 and A[i]>key
6             do A[i+1]←A[i]
7                i←i-1
8          A[i+1]←key
```

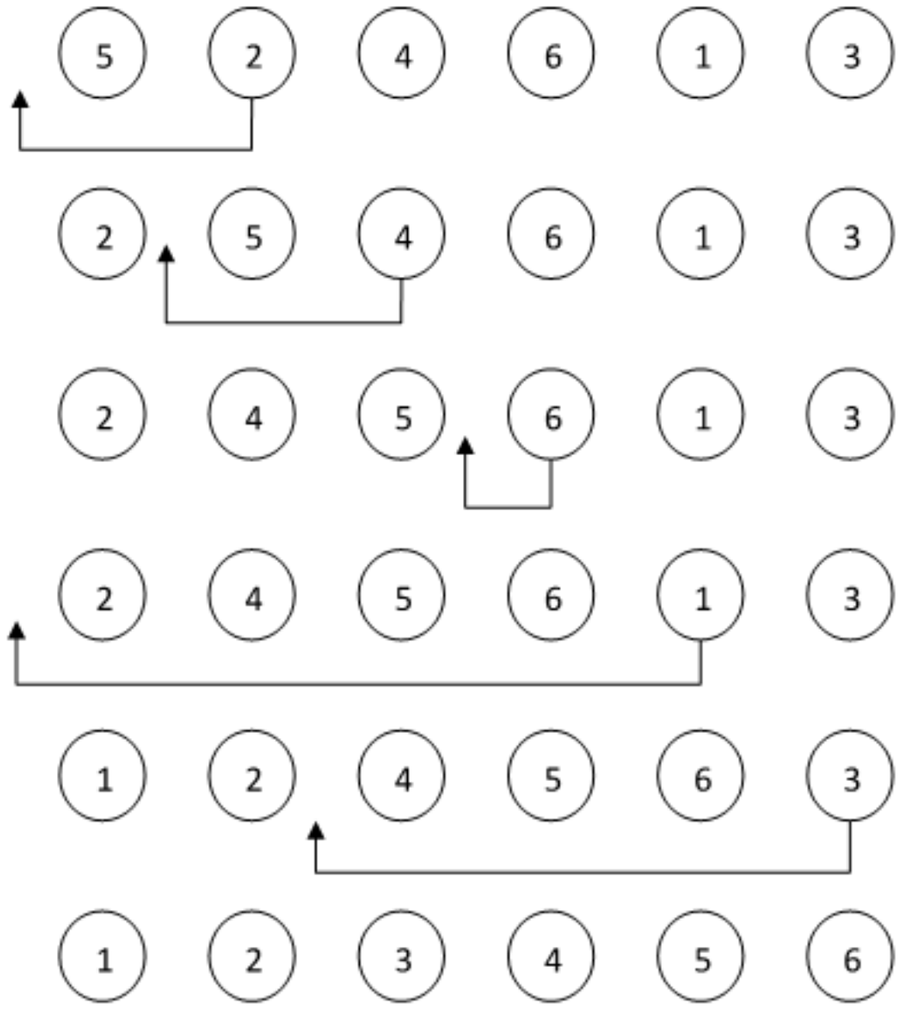


Рисунок 1.1 - Работа процедуры INSERTION-SORT для входа A = (5, 2, 4, 6, 1, 3). Позиция j показана кружком

Трассировка работы алгоритма INSERTION-SORT

Insertion-Sort(A) A = (5, 2, 4, 6, 1, 3)

1 for j←2 to 6 (i, j)

2 do key←A[2] = 2

3 ›добавить A[2] к отсортированной части A[1] = 5

4 i←j-1 = 1

5 while 1 > 0 and A[1] = 5 > key = 2

6 do A[2] ← A[1] = 5

7 i ← i - 1 = 0

5 i ≯ 0

8 A[1] ← key = 2 A = (2, 5, 4, 6, 1, 3) (i, j)

1 j = 3, 6

2 do key ← A[3] = 4

3 ›добавить A[3] к отсортированной части A[1, 2] = (2, 5)

4 i ← i - 1 = 2

5 while 2 > 0 and A[2] =5 > key = 4

6 do A[3]←A[2] = 5

7 i ← i - 1 = 1

5 while 1 > 0 and A[1] = 2 ≯ key = 4

8 A[2] ← key = 4 A = (2, 4, 5, 6, 1, 3) (i, j)

2 j = $\overline{4, 6}$

2 do key ← A[4] = 6

3 ›добавить A[3] к отсортированной части A[1, 2, 3] = (2, 4, 5)

4 i ← i - 1 = 3

5 while 3 > 0 and A[3] =5 > key = 6

8 A[4] ← key = 6 A = (2, 4, 5, 6, 1, 3)

i j

и т. д.

Анализ алгоритмов

Сортировка вставками: анализ

Insertion-Sort(A)	**стоимость**	**число раз**
1 for j←2 to length[A]	c_1	n
2 do key←A[j]	c_2	n-1
3 ›добавить A[j] к отсортированной части A[1..j-1]	0	n-1
4 i←j-1	c_4	n-1
5 while i>0 and A[i]>key	c_5	$\sum_{j=2}^{n} tj$
6 do A[i+1]←A[i]	c_6	$\sum_{j=2}^{n}(tj-1)$
7 i←i-1	c_7	$\sum_{j=2}^{n}(tj-1)$
8 A[i+1]←key	**c_8**	**n-1**

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Благоприятный случай (массив уже отсортирован): $t_j = 1$.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)\, n - (c_2 + c_4 + c_5 + c_8).$$

$T(n) = an + b.$

Массив расположен в обратном (убывающем) порядке: $t_i = j$. $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1\,, \qquad \sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2},$$

В худшем случае время работы процедуры равно:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) =$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8),$$

$T(n) = an^2 + bn + c.$ $T(n) = \Theta(n^2)$

Рассмотреть случай время работы в среднем ($t_i = j/2$).

Упражнения

1. Проиллюстрируйте работу процедуры INSERTION-SORT по сортировке массива A= {31, 41, 59, 26, 41, 58}.

2. Перепишите процедуру INSERTION-SORT для сортировки в невозрастающем порядке вместо неубывающего.

Вывод: Алгоритм с меньшим порядком роста времени работы обычно предпочтительнее: если, скажем, один алгоритм имеет время работы $\Theta(n^2)$, а другой — $\Theta(n^3)$, то первый более эффективен (по крайней мере для достаточно длинных входов; будут ли реальные входы таковыми — другой вопрос).

ЛЕКЦИЯ 2

Построение алгоритмов. Принцип «разделяй и властвуй». Анализ алгоритмов типа «разделяй и властвуй». О пользе быстрых алгоритмов

Сортировка вставками является примером алгоритма, действующего **по шагам.**

Принцип «разделяй и властвуй»

Рекурсивные алгоритмы – это алгоритмы, которые решая некоторую задачу, они вызывают самих себя для решения её подзадач.

Идея метода «разделяй и властвуй» состоит в том:

1 задача разбивается на несколько подзадач меньшего размера;

2 задачи решаются (с помощью рекурсивного вызова или непосредственно, если размер достаточно мал);

3 решения подзадач комбинируются и получается решение исходной задачи.

Для задачи сортировки эти три этапа выглядят так:

1 разбиваем массив на две половины меньшего размера;
2 сортируем каждую из половин отдельно;
3 соединяем два упорядоченных массива половинного размера в один с помощью вспомогательной процедуры MERGE (A, p, q, r).

Весь массив теперь можно отсортировать с помощью вызова MERGE-SORT (A, 1, length[A]).

```
MERGE-SORT (A, p, r)
1       if p < r
2           then q← ⌊(p + r) / 2⌋
3                   Merge-Sort (A, p, q)
4                   Merge-Sort (A, q+1, r)
5                   Merge (A, p, q, r)
```

$\lfloor x \rfloor \leq x \leq \lceil x \rceil$

$\lfloor x \rfloor -$ целую часть x (наибольшее целое число, меньшее или равное x);

$\lceil x \rceil -$ наименьшее целое число, большее или равное x.

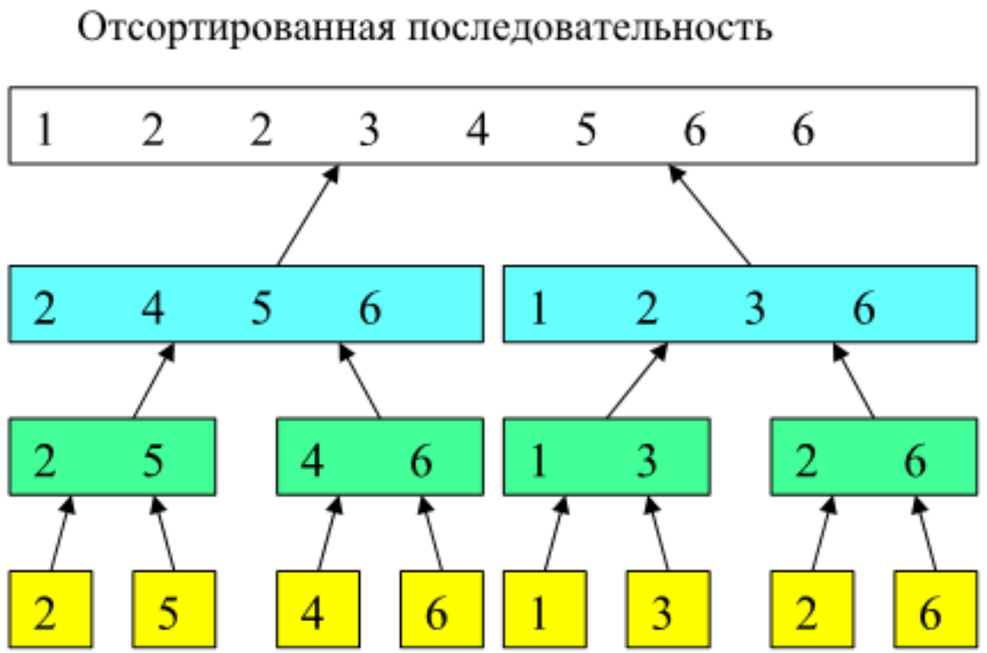


Рисунок 2.1 - Сортировка слиянием для массива A [5, 2, 4, 6, 1, 3, 2, 6]

```
MERGE (A, p, q, r)
1. n ← q – p + 1
2. m ← r – q
3. ⪢ Создаем массивы L[1 . . . n + 1] и R[1 . . . m + 1]
4. for i ← 1 to n
5.       do L[i] ← A[p + i – 1]
6. for j ← 1 to m
7.       do R[j] ← A[q + j]
8. L[n + 1] ← ∞
9. R[m + 1] ← ∞
10.   i ← 1
11.   j ← 1
12.   for k ← p to r
13.         do if L[i] ≤ R[j]
14.               then A[k] ← L[i]
15.                     i ← i +1
16.               else A[k] ← R[j]
17.                     j ← j +1
```

A - массив,

числа p, q, r - границы сливаемых участков ($p \le q < r$),

A[p..q] и A[q+1..r] - уже отсортированы,

A[p.. r] – результирующий массив

Время работы процедуры MERGE есть

$T(n) = \Theta(n)$,

где n — общая длина сливаемых участков

($n = r – p + 1$).

MERGE-SORT (A, 1, 8) A = (5, 2, 4, 6, 1, 3, 2, 6)

1 if 1 < 8
2 then q← ⌊(1 + 8) / 2⌋ = 4
3 Merge-Sort (A, 1, 4)

1 if 1 < 4
2 q← ⌊(1 + 4) / 2⌋ = 2
3 M.-S. (A, 1, 2)

1 if 1 < 2
2 q←⌊(1 + 2) / 2⌋ = 1
3 M.-S. (A, 1, 1)

1 if 1 < 1
A[1] = 5

4 M.-S. (A, 2, 2)

1 if 2 < 2
A[2] = 2

5 Merge (A, 1, 1, 2) - A[1, 2] = (2, 5)

4 M.-S. (A, 3, 4)

1 if 3 < 4
2 q←⌊(3 + 4) / 2⌋ = 3
3 M.-S. (A, 3, 3)

1 if 1 < 1
A[3] = 4

4 M.-S. (A, 4, 4)

1 if 4 < 4
A[4] = 6

5 Merge (A, 3, 3, 4) - A[3, 4] = (4, 6)

5 Merge (A, 1, 2, 4) A[1, 2, 3, 4] = (2, 4, 5, 6)

4 Merge-Sort (A, 4+1, 8)

1 if 5 < 8
2 q← ⌊(5 + 8) / 2⌋ = 6
3 M.-S. (A, 5, 6)

1 if 5 < 6
2 q←⌊(5 + 6) / 2⌋ = 5
3 M.-S. (A, 5, 5)

1 if 5 < 5
A[5] = 1

4 M.-S. (A, 6, 6) A[6] = 3

5 Merge (A, 5, 5, 6) - A[5, 6] = (1, 3)

4 M.-S. (A, 7, 8) A[7, 8] = (2, 6)

5 Merge (A, 5, 6, 8) A[5, 6, 7, 8] = (1, 2, 3, 6)

5 Merge (A, 1, 4, 8) A[1, 2, 3, 4, 5, 6, 7, 8] = (1, 2, 2, 3, 4, 5, 6, 6)

MERGE (A, 1, 2, 4) A[1, 2] = (2, 5) A[3, 4] = (4, 6)

1. n ← q – p + 1=2-1+1=2
2. m ← r – q=4 – 2=2
3. ⪢ Создаем массивы L[1 . . . n + 1] и R[1 . . . m + 1] L[1 . . . 3], R[1 . . .3]
4. for i ← 1 to n 1, 2 i = 1 i = 2
5. do L[i] ← A[p + i – 1] L[1] = A[1+1-1] = A[1]=2 L[2] = A[1+2-1] = A[2]=5
6. for j ← 1 to m=2 1, 2 j = 1 j = 2
7. do R[j] ← A[q + j] R[1] = A[2+1]=4 R[2] = A[2+2]=6
8. L[n + 1] ← ∞ L[3] = ∞
9. R[m + 1] ← ∞ R[3] = ∞
10. i ← 1
11. j ← 1
12. for k ← p to r 1, 4 k=1 k=2 k=3 k=4
13. do if L[i] ≤ R[j] L[1] ≤ R[1] 1 ≤ 4 L[2] ≤ R[1] 5 ≰ 4 L[2] ≤ R[2] 5 ≤ 6 L[3] ≤ R[2] ∞ ≤ 6
14. then A[k] ← L[i] A[1] ← L[1] = 2 A[3] ← L[2] = 5
15. i ← i +1 i ← i +1= 1+1=2 i ← 2 +1= 2+1=3
16. else A[k] ← R[j] A[2] ← R[1]=4 A[4] ← R[2]=6
17. j ← j +1 j ← 1 +1=2 j ← 2 +1=3

A[1, 2, 3, 4] = (2, 4, 5, 6)

Анализ алгоритмов типа «разделяй и властвуй»

Как оценить время работы рекурсивного алгоритма?

Предположим, что алгоритм разбивает задачу размера **n** на **a** подзадач, каждая из которых имеет в b раз меньший размер.

Будем считать, что:

- разбиение требует времени D(n),
- соединение полученных решений — времени C(n).

Тогда

$$T(n) = aT(n/b) + D(n) + C(n).$$

Это соотношение выполнено для достаточно больших n, когда задачу имеет смысл разбивать на подзадачи. Для малых b, когда такое разбиение невозможно или не нужно, применяется какой-то прямой метод решения задачи. Поскольку n ограничено, время работы тоже не превосходит некоторой константы.

Анализ сортировки слиянием

Предположим, что:

размер массива есть степень двойки.

Тогда

– разбиение на части (вычисление границы) требует времени $\Theta(1)$,

– слияние — времени $\Theta(n)$.

Получаем соотношение:

$$T(n) = \begin{cases} \Theta(1) & \text{если } n = 1 \\ 2T(n/2) + \Theta(n), & \text{если } n > 1 \end{cases}$$

Позже докажем, что

$T(n) = \Theta(n \log n)$,

(основание логарифмов не играет роли, так как приводит лишь к изменению константы).

Вывод: для больших n сортировка слиянием эффективнее сортировки вставками, требующей времени $\Theta(n^2)$.

О пользе быстрых алгоритмов

Часто разница между плохим и хорошим алгоритмом более существенна, чем между быстрым и медленным компьютером.

Пример:

Отсортировать массив из миллиона чисел.

Что быстрее — сортировать его вставками на компьютере № 1 (100 миллионов операций в секунду) или слиянием на компьютере № 2 (1 миллион операций)?

Пусть к тому же сортировка вставками написана на ассемблере чрезвычайно экономно, и для сортировки n чисел нужно, скажем, лишь $2n^2$ операций.

В то же время алгоритм слиянием написан без особой заботы об эффективности и требует $50n \log n$ операций.

Для сортировки миллиона чисел получаем:

$$\frac{2(10^6)^2 операций}{10^8 операцийвсекунду} = 20000 секунд \approx 5{,}56 часов$$

для компьютера № 1,

$$\frac{50(10^6)\, log(10^6) операций}{10^6 операцийвсекунду} = 1000 секунд \approx 17 минут$$

для компьютера № 2.

Вывод: разработка эффективных алгоритмов не менее важна, чем разработка быстрой электроники.

Упражнение

Проиллюстрируйте работу алгоритмов сортировки слиянием и быстрой для массива A = (3, 41, 52, 26, 38, 57, 9, 49).

```
QUICK – SORT (A, p, r)
1 if  p < r
2       q ←  PARTITION (A, p, r)
3       QUICK – SORT (A, p, q)
4       QUICK – SORT (A, q+1, r)

PARTITION (A, p, r)
1 x← A[p]
2 i ← p-1
3 j ← r+1
4 while true
5               do repeat
6                       j← j − 1
7               until A[j]≤ x
8               do repeat
9                       i ← i+1
10              until A[j]≥ x
11              if i < j
12                      t ← A[i]
13                      A[i] ← A[j]
14                      A[j] ← t
15              else return j
```

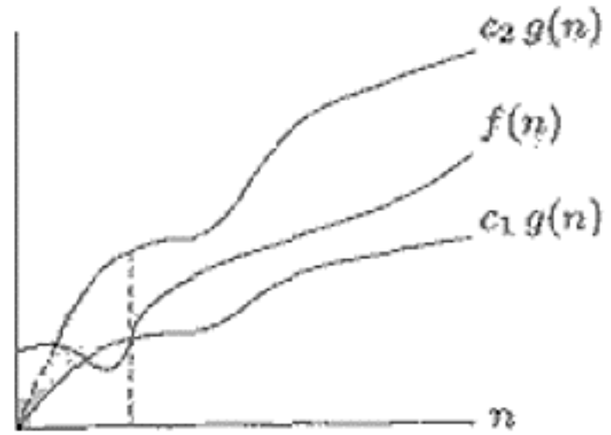
ЛЕКЦИЯ 3

Математические основы анализа алгоритмов. Скорость роста функций. Асимптотические обозначения. Анализ времени работы алгоритмов

Асимптотические обозначения

Θ-обозначение

Время **T(n)** работы алгоритма сортировки вставками на входах длины **n** есть **Θ(n^2)**.



Точный смысл этого утверждения такой: найдутся такие константы c_1, $c_2 > 0$ и такое число n_0, что $c_1n^2 \le T(n) \le c_2n^2$ при всех $n \ge n_0$. Рисунок 3.1(а))

Если функции f(n) и g(n) асимптотически неотрицательны для достаточно больших значений n, то запись

$$f(n) = \Theta(g(n))$$

означает, что найдутся такие c_1, $c_2 > 0$ и такое n_0, что $0 \le c_1g(n) \le f(n) \le c_2(n)$ для всех $n \ge n_0$ (см. рисунок 3.1(а)). (Запись $f(n) = \Theta(g(n))$ читается так: «эф от эн есть тэта от же от эн».)

Обозначение $f(n) = \Theta(g(n))$ следует употреблять с осторожностью: установив, что

$f_1(n) = \Theta(g(n))$ и $f_2(n) = \Theta(g(n))$, не следует заключать, что $f_1(n) = f_2(n)$!

Если $f(n) = \Theta(g(n))$, то говорят, что $g(n)$ является асимптотически точной оценкой для $f(n)$.

На самом деле это отношение симметрично: если

$$f(n) = \Theta(g(n)), \text{ то } g(n) = \Theta(f(n)).$$

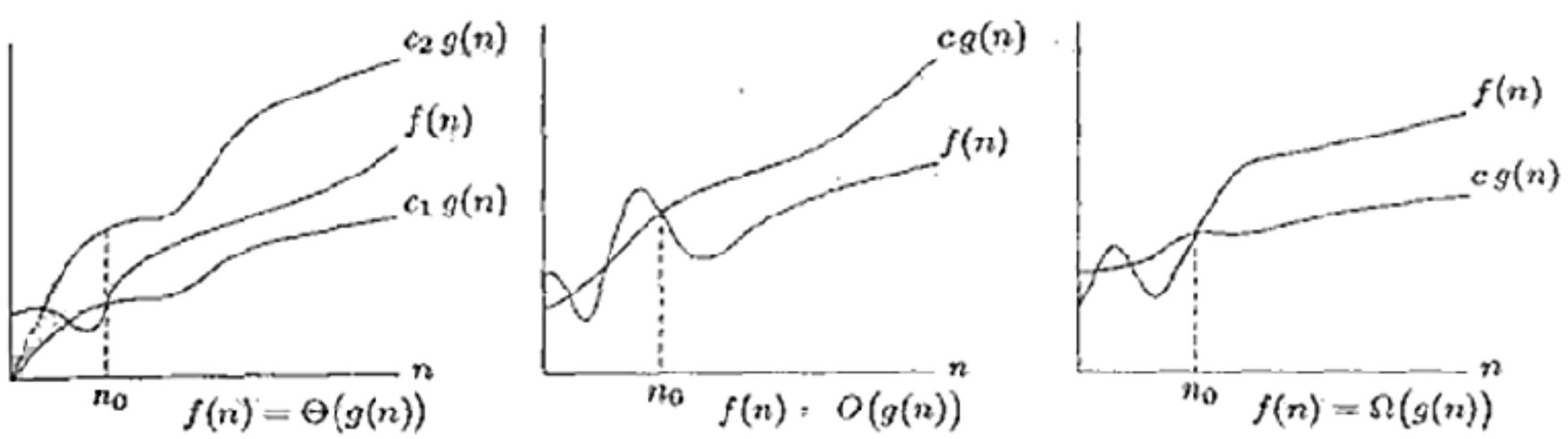


Рисунок 3.1 - Иллюстрации к определениям $f(n) = \Theta(g(n))$, $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$

Пример 1:

Проверить, что (1/2) $n^2 - 3n = \Theta(n^2)$.

Согласно определению, надо указать положительные константы $c_1 > 0$, $c_2 > 0$ и число n_0 так, чтобы неравенства

$$c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2$$

выполнялись для $\forall\ n \geq n_0$.

Разделим на n^2:

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Видно, что для выполнения второго неравенства (при $n \to \infty$) достаточно положить $c_2 = 1/2$. Первое будет выполнено, если, например, $n_0 = 7$ и $c_1 = 1/14$.

$\frac{1}{2} - \frac{3}{n} = \frac{1}{2} - \frac{3}{7} = \frac{7-6}{14} = \frac{1}{14}$.

$\Longrightarrow$ (1/2) $n^2 - 3n = \Theta(n^2)$.

Пример 2:

Например, рассмотрим квадратичную функцию

$f(n) = an^2 + bn + c,$

где a, b, c — некоторые константы и $a > 0$.

Отбрасывая члены младших порядков и коэффициент при старшем члене, находим, что $f(n) = \Theta(n^2)$.

Здесь $\Theta(n)$ обозначает некоторую функцию, про которую важно знать лишь, что она не меньше $c_1 n$ и не больше $c_2 n$ для некоторых положительных c_1 и c_2 и для всех достаточно больших n.

Для любого полинома $p(n)$ степени d с положительным старшим коэффициентом имеем $p(n) = \Theta(n^d)$.

$\Theta(1)$ обозначает ограниченную функцию, отделённую от нуля некоторый положительной константой при достаточно больших значениях аргумента.

O- и Ω-обозначения

Запись $f(n) = \Theta(g(n))$ включает в себя две оценки: **верхнюю (O)** и **нижнюю (Ω)**.

Их можно разделить.

Говорят, что $f(n) = \mathbf{O}(g(n))$, если найдётся такая константа $c > 0$ и такое число n_0, что $0 \le f(n) \le cg(n)$ для $\forall\, n \ge n_0$ (см. рисунок 3.1(б)).

Говорят, что $f(n) = \Omega(g(n))$, если найдется такая константа $c > 0$ и такое число n_0, что $0 \le cg(n) \le f(n)$ для $\forall\, n \ge n_0$ (см. рисунок 3.1(в)).

Эти записи читаются так: «эф от эн есть о большое от же от эн», «эф от эн есть омега большая от же от эн».

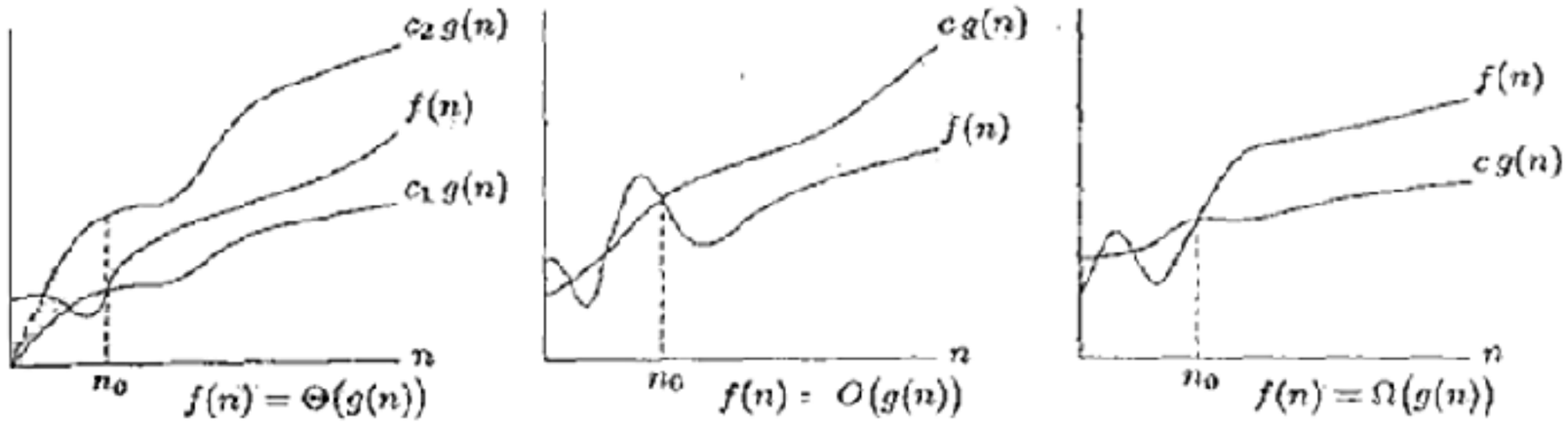


Рисунок 3.1 - Иллюстрации к определениям $f(n) = \Theta(g(n))$, $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$

Теорема 3.1. Для любых двух функций f(n) и g(n) свойство f(n) = Θ(g(n)) выполнено тогда и только тогда, когда

$$f(n) = O(g(n)) \text{ и } f(n) = \Omega(g(n)).$$

Для любых двух функций f(n) и g(n) свойства

$$f(n) = O(g(n)) \text{ и } g(n) = \Omega(f(n)) \text{ равносильны.}$$

Примеры:

1) $an^2 + bn + c = \Theta(n^2)$ (при положительных a).

   Поэтому $an^2 + bn + c = O(n^2)$.

2) при $a > 0$ можно написать $an + b = O(n^2)$.

   Докажем это

$$c_1 n^2 \leq an + b \leq c_2 n^2.$$

Разделим на n^2:

$$c_1 \leq \frac{a}{n} + \frac{b}{n^2} \leq c_2.$$

c_1 - ?

В этом случае $an + b \neq \Omega(n^2)$ и $an + b \neq \Theta(n^2)$.

Положим $c_2 = a+|b|$ ($c_2 > 0$) и $n_0 = 1$.

Поэтому можно написать $an + b = O(n^2)$.

Пример 3:

Покажем, что $6n^3 \neq \Theta(n^2)$.

В самом деле, пусть найдутся такие c_1, c_2 и n_0, что

$c_1 n^2 \leq 6n^3 \leq c_2 n^2$ для $\forall\ n \geq n_0$.

Но тогда

$c_1 \leq 6n \leq c_2$ для $\forall\ n \geq n_0 = 1$,

$c_1 = 6, c_2 - ?, \Longrightarrow\ 6n^3 \neq O(n^2), 6n^3 \neq \Theta(n^2), \Longrightarrow 6n^3 = \Omega(n^2)$

Асимптотические обозначения (Θ, Ω и Ȯ) часто употребляются внутри формул. Например, получили рекуррентное соотношение для времени работы сортировки слиянием.

$$T(n) = 2T(n/2) + \Theta(n)$$

Здесь Θ(n) обозначает некоторую функцию, про которую важно знать лишь, что она не меньше $c_1 n$ и не больше $c_2 n$ для некоторых положительных c_1 и c_2 и для всех достаточно больших n.

Часто асимптотические обозначения употребляются не вполне формально, хотя их подразумеваемый смысл обычно ясен из контекста. Например, можно написать выражение:

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

имея в виду сумму h(l) + h(2) + ... + h(n), где h(i) — некоторая функция, для которой h(i) = Ȯ (i). Легко видеть, что сама эта сумма как функция от n есть $O(n^2)$.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

Типичный пример неформального использования асимптотических обозначений - цепочка равенств наподобие

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2).$$

Второе из этих равенств

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

понимается при этом так: какова бы ни была функция $h(n) = \Theta(n)$ в левой части, сумма

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

o- и ω-обозначения

Запись **f(n) = O(g(n))** означает, что с ростом **n** отношение **f(n)/g(n)** остаётся ограниченным. Если к тому же

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

то пишем **f(n) = o(g(n))** (читается «эф от эн есть о малое от же от эн»).

Формально говоря, **f(n) = o(g(n))**, если для всякого положительного $\varepsilon > 0$ найдётся такое n_0 что

$0 \le f(n) \le \varepsilon g(n)$ при $\forall\ n \le n_0$.

(Тем самым запись f(n) = o(g(n)) предполагает, что f(n) и g(n) неотрицательны для достаточно больших n.)

Пример: $2n = o(n^2)$, но $2n^2 \neq o(n^2)$.

$c_1 n^2 \le 2n \le c_2 n^2$, $c_1 \le \frac{2}{n} \le c_2$ для $\forall\ n \ge n_0 = 1$, $c_2 = 2, c_1 - ?, \lim_{n\to\infty} \frac{2n}{n^2} = 0, \Longrightarrow 2n = o(n^2)$.

Аналогичным образом вводится **ω-обозначение**:

говорят, что **f(n) = ω(g(n))** (“эф от эн есть омега малая от же от эн”), если для всякого положительного **c** найдется такое $\mathbf{n_0}$, что

$0 \le cg(n) \le f(n)$ при $\forall\ n \ge n_0$,

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

Очевидно, **f(n) = ω(g(n))** равносильно g(n) = o(f(n)).

Пример: $n^2/2 = \omega(n)$, но $n^2/2 \neq \omega(n^2)$.

Сравнение функций

Асимптоты обладают некоторыми свойствами транзитивности, рефлексивности и симметричности.

Транзитивность:

$f(n) = \Theta(g(n))$ и $g(n) = \Theta(h(n))$ влечет $f(n) = \Theta(h(n))$,

$f(n) = O(g(n))$ и $g(n) = O(h(n))$ влечет $f(n) = O(h(n))$,

$f(n) = \Omega(g(n))$ и $g(n) = \Omega(h(n))$ влечет $f(n) = \Omega(h(n))$,

$f(n) = o(g(n))$ и $g(n) = o(h(n))$ влечет $f(n) = o(h(n))$,

$f(n) = \omega(g(n))$ и $g(n) = \omega(h(n))$ влечет $f(n) = \omega(h(n))$.

Рефлексивность:

$f(n) = \Theta(f(n))$, $f(n) = O(f(n))$, $f(n) = \Omega(f(n))$.

Симметричность:

$f(n) = \Theta(g(n))$ если $g(n) = \Theta(f(n))$.

Обращение:

$f(n) = O(g(n))$ если $g(n) = \Omega(f(n))$,

$f(n) = o(g(n))$ если $g(n) = \omega(f(n))$.

Можно пронести такую параллель: отношения между функциями f и g подобны отношениям между числами a и b:

$f(n) = O(g(n)) \approx a \leq b$,

$f(n) = \Omega(g(n)) \approx a \geq b$,

$f(n) = \Theta(g(n)) \approx a = b$,

$f(n) = o(g(n)) \approx a < b$,

$f(n) = \omega(g(n)) \approx a > b$.

ЛЕКЦИЯ № 4

Стандартные функции и обозначения. Временные оценки. Суммирование. Сумма и их свойства.

Для любого целого n

$$\lfloor n/2 \rfloor + \lceil n/2 \rceil = n.$$

Полиномы

Для заданного неотрицательного целого d полином степени d от аргумента n называется функция p(n) вида

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

где константы $a_0, a_1, \ldots, a_d$ – коэффициенты полинома и $a_d \neq 0$. Полином является асимптотически положительной функцией тогда и только тогда, когда $a_d > 0$. Для асимптотически положительных полиномов $p(n) = \Theta(n^d)$.

Говорят, что функция f(n) полиномиально ограничена, если существует такая константа k, что

$f(n) = O(n^k)$.

Соотношение между скоростями полиномов и показательных функций можно определить исходя из того факта, что для любых действительных констант a и b, таких, что $a > 1$,

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0,$$

Откуда можно заключить, что

$$n^b = o(a^n).$$

Таким образом, любая показательная функция, основание **a** которой строго больше единицы, возрастает быстрее любой полиномиальной функции.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} ,$$

Для всех вещественных X выполнено неравенство $e^x \geq 1 + x$.

При |x| ≤ 1.

$$1 + x \leq e^x \leq 1 + x + x^2 .$$

Можно сказать, что при x→ 0

$$e^x = 1 + x + \Theta(x^2) .$$

Для ∀ X имеем

$$\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x .$$

Логарифмы

$$\log n = \log_2 n \qquad \text{(бинарный логарифм)}$$
$$\ln n = \log_e n \qquad \text{(натуральный логарифм)}$$
$$\lg^k n = (\lg n)^k \qquad \text{(возведение в степень)}$$
$$\lg \lg n = \lg(\lg n) \qquad \text{(композиция)}$$

Для всех действительных $a > 0$, $b > 0$, $c > 0$ и n

$$a = b^{\log_b a}\,,$$
$$\log_c(ab) = \log_c a + \log_c b\,,$$
$$\log_b a^n = n \log_b a\,,$$
$$\log_b a = \frac{\log_c a}{\log_c b}\,,$$
$$\log_b(1/a) = -\log_b a\,,$$
$$\log_b a = \frac{1}{\log_a b}\,,$$
$$a^{\log_b c} = c^{\log_b a}\,,$$

При $b > 1$ функция от $n = \log_b n$ ($n > 0$) строго возрастает.

Для натурного логарифма есть ряд (которых сходится при |x| ≤ 1):

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \ldots$$

При x > -1 справедливы следующие неравенства:

$$\frac{x}{1+x} \leq \ln(1+x) \leq x,$$

которые обращаются в равенство при x= 0.

Любой полином растёт быстрее любого полилогарифма и для ∀ a > 0:

$\log^{b}(n) = o(n^{a})$.

Число Фибоначчи

Последовательность чисел Фибоначчи определяется рекуррентным соотношением:

$F_0 = 0$, $F_1 = 1$, $F_i = F_{i-1} + F_{i-2}$, $i \geq 2$, то есть последовательность Фибоначчи: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

Числа Фибоначчи связаны с так называемым отношением золотого сечения φ^i с сопряженным с ним числом $\hat{\varphi}$:

$$\varphi = \frac{1+\sqrt{5}}{2} = 1{,}61803\ldots,$$

$$\hat{\varphi} = \frac{1-\sqrt{5}}{2} = -0{,}61803\ldots.$$

Имеет место формула:

$$F_i = \frac{\varphi^i - \hat{\varphi}^i}{\sqrt{5}},$$

которую можно доказать по индукции.

Поскольку $|\hat{\varphi}| < 1$, слагаемое $\left|\hat{\varphi}^i/\sqrt{5}\right| < 1/\sqrt{5} < 1/2$, так что F_i = числу $\frac{\varphi^i}{\sqrt{5}}$, округленному до ближайшего целого. Числа F_i экспоненциально растут с ростом i.

Факториалы

$\mathbf{n! \leq \ n^n}$

Формула Стирлинга

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) \ ,$$

Из этой формулы следуют:

$n! = \ \sigma\ (n^n)$,

$n! = \omega(2^n)$,

$\log(n!) = \Theta(n \log n)$.

Для всех n ≥ 1

$$\sqrt{2\pi n}(\frac{n}{e})^n \leq n! \ \leq \ \sqrt{2\pi n}(\frac{n}{e})^n e^{1/12n}$$

Итерированная логарифмическая функция

Обозначим $\log^* n$ итерированным логарифмом.

Пусть $\log^{(i)} n$ представляет собой i итерацию функции f(n) = log(n), определённую так:

$$\log^{(0)} n = n \text{ и } \log^{(i)}(n) = \log(\log^{(i-1)} n)$$

$$\log^{(*)} 2 = 1$$

$$\log^{(*)} 2^2 = \log^{(*)} 4 = 2$$

$$\log^{(*)} 2^4 = \log^{(*)} 16 = 3$$

$$\log^{(*)} 2^{16} = \log^{(*)} 65536 = 4$$

$$\log^{(*)} 2^{65536} = 5$$

Это очень медленно растущая функция.

Поскольку количество атомов в видимой Вселенной оценивается примерно в 2^{80}, что на много меньше 2^{65536}, то значение n, для которых $\log^* n > 5$, вряд ли могут встретиться.

Арифметическая прогрессия

Сумма:

$$\sum_{k=1}^{n} k = 1+2+ \ldots + \mathrm{n}$$

является арифметической прогрессией.

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \Theta(n^2), \qquad \sum_{k=1}^{n+1} k = \frac{n(n+1)}{2} + n + 1 = \frac{(n+1)(n+2)}{2}.$$

Геометрическая прогрессия

При $x \neq 1$

$$\sum_{k=0}^{n} x^k = 1 + \mathrm{x} + x^2 + \ldots + x^n,$$

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1},$$

$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ при $|x| < 1$, сумма бесконечно убывающей геометрической прогрессии (*)

Гармонический ряд

1+ 1/2 +1/3+ . . . + 1/n + . . .,

его n-я частичная сумма равна:

$\mathbf{H_n = 1+ 1/2 +1/3+ \ldots + 1/n = \sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1).}$

Почленное интегрирование и дифференцирование

Дифференцируя тождество (*) и умножая на x, получим

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \qquad \sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

Сумма разностей

Для любой последовательности $a_0, a_1, \ldots, a_n$ можно записать тождество:

$$\sum_{k=1}^{n}(a_k - a_{k-1}) = a_n - a_0.$$

Аналогично,

$$\sum_{k=0}^{n-1}(a_k - a_{k+1}) = a_0 - a_n.$$

Этот приём позволяет просуммировать ряд:

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

Поскольку $\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1}$,

Получаем, что

$$\sum_{k=1}^{n-1}\frac{1}{k(k+1)} = \sum_{k=1}^{n-1}\left(\frac{1}{k} - \frac{1}{k+1}\right) = 1 - \frac{1}{n-1+1} = 1 - \frac{1}{n}.$$

Произведение

$$\log \prod_{k=1}^{n} a_k = \sum_{k=1}^{n} \log a_k$$

Оценки сумм

Рассмотрим несколько приёмов, которые позволяют найти значение суммы (или хотя бы оценить эту сумму сверху или снизу).

Индукция

Формулу для суммы легко проверить с помощью математической индукции.

Пример: докажем, что сумма арифметический прогрессии

$$S_n = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

Решение:

Проверим при n=1, выполняется.

Теперь предположим, что равенство $S_n=n(n+1)/2$ верно при некотором n и проверим его для n+1. В самом деле,

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1) = n(n+1)/2 + (n+1) = (n+1)(n+2)/2.$$

Почленные сравнения

Иногда нужно получить верхнюю оценку для суммы, заменив каждый её член больший.

Так $\sum_{k=1}^{n} k \le \sum_{k=1}^{n} \mathrm{n} = \mathrm{n}^2$.

В общем случае

$\sum_{k=1}^{n} a_k \le \mathrm{n}\ a_{\max}$.

$\sum_{k=0}^{n} a_k \le \sum_{k=0}^{\infty} a_0 r^k = a_0 \sum_{k=0}^{\infty} r^k = a_0 \frac{1}{1-r}$

где $a_{k+1}/a_k \le r$ при $r < 1$, $a_k \le a_0\ r^k$.

Применим этот метод для суммы $\sum_{k=1}^{n} k\ 3^{-k}$. Первый член = 1/3, отношение соседних членов равно:

$$\frac{(k+1)\,3^{-(k+1)}}{k3^{-k}} = \frac{1}{3}\frac{k+1}{k} = \frac{1}{3} + \frac{1}{3k} \le \frac{2}{3}, \qquad r = \frac{2}{3} \quad \text{для } \forall\, k \ge 1 \qquad a_0 = \frac{1}{3}.$$

Следовательно, каждый член суммы оценивается сверху величиной $a_0\ r^k = (1/3)(2/3)^k$, так что

$$\sum_{k=1}^{\infty} k\ 3^{-k} \le \sum_{k=1}^{\infty} \frac{1}{3}(2/3)^k = \frac{1}{3}\sum_{k=1}^{\infty}(2/3)^k = \frac{1}{3}\,\frac{1}{1-\frac{2}{3}} = 1.$$

Важно, что отношение соседних членов не просто меньше 1, а ограничено некоторой константой r<1, общей для всех членов ряда. Например, для гармонического ряда отношение (k+1) - го и k - го членов равно $\frac{k}{k+1} < 1$. Тем не менее

$$\sum_{k=1}^{\infty} \frac{1}{k} = \lim_{n\to\infty} \sum_{k=1}^{n} \frac{1}{k} = \lim_{n\to\infty} \Theta(\log n) = \infty.$$

Важные формулы

$a^n = e^{nlna}$; $3^n = e^{nln3}$; $e^x = \lim_{n\to\infty}\left(1+\frac{x}{n}\right)^n$;

Геометрическая прогрессия:

$\sum_{k=1}^{n} aq^{k-1} = \frac{a(q^n-1)}{q-1}$; $\sum_{k=1}^{n} 3^i = \frac{3^n-1}{3-1} = \frac{3^n}{2} - \frac{1}{2} = O(3^n)$;

Арифметическая прогрессия:

$\sum_{k=0}^{n-1}(a+kr) = \frac{n}{2}(2a+(n-1)r) = \frac{n}{2}(a+l)$,

где l – последний член последовательности.

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

$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} = O(n^3)$;

$\sum_{k=1}^{n} k^3 = [\frac{n(n+1)}{2}]^2 = O(n^4)$;

$\sum_{k=1}^{n} k!\,k = (n+1)! - 1$

$\sum_{k=1}^{n} k^4 = \frac{1}{30} n(n+1)(2n+1)(3n^2+3n-1) = O(n^5)$

Задачи

1 Относительный асимптотический рост. Для всех клеток следующей таблицы ответьте «да» или «нет» на вопрос о том, можно ли записать A как O, σ, Ω, ω или Θ от B ($k \geq 1$, $\varepsilon > 0$, c, $m > 1$ – некоторые константы). Приведите расчеты.

A	B	O	σ	Ω	ω	Θ
$\log^k n$	n^ε					
n^k	c^n					
$\sqrt{n}$	$n^{\sin n}$					
2^n	$2^{n/2}$					
$n^{\log m}$	$m^{\log n}$					
log (n!)	log (n^n)					

2 Сравнение скорости роста

Расположите следующие функции в порядке увеличения скорости роста. Приведите расчеты. Постройте графики этих функции для наглядности.

1. $2^{\log m}$ $(\sqrt{2})^{\log m}$
2. $2^{2^{**}n}$ $2^{2^{**}n+1}$
3. n! (n+1)!
4. log (n!) (log n)!
5. $(3/2)^n$ e^n $n*2^n$
6. n 2n n^3 n^2
7. $n^{1/\log n}$ $n^{\log\log n}$ $(\log n)^{(\log n)}$
8. ln n ln ln n
9. 4 log n $\log^2 n$ n log n
10. $\sqrt{}$ (log n) $2\sqrt{}$ (2log n)

ЛЕКЦИИ № 5

Кучи. Сохранение основного свойства кучи. Построение кучи. Сортировка с помощью кучи. Организация очереди с приоритетами

Кучи

Двоичной кучей (binary heap) называют массив с определёнными свойствами упорядоченности.

Каждая вершина дерева соответствует элементу массива. Если вершина имеет индекс i, то её родитель имеет индекс [i/2] (вершина с индексом 1 является корнем), а её дети - индексы 2i и 2i + 1.

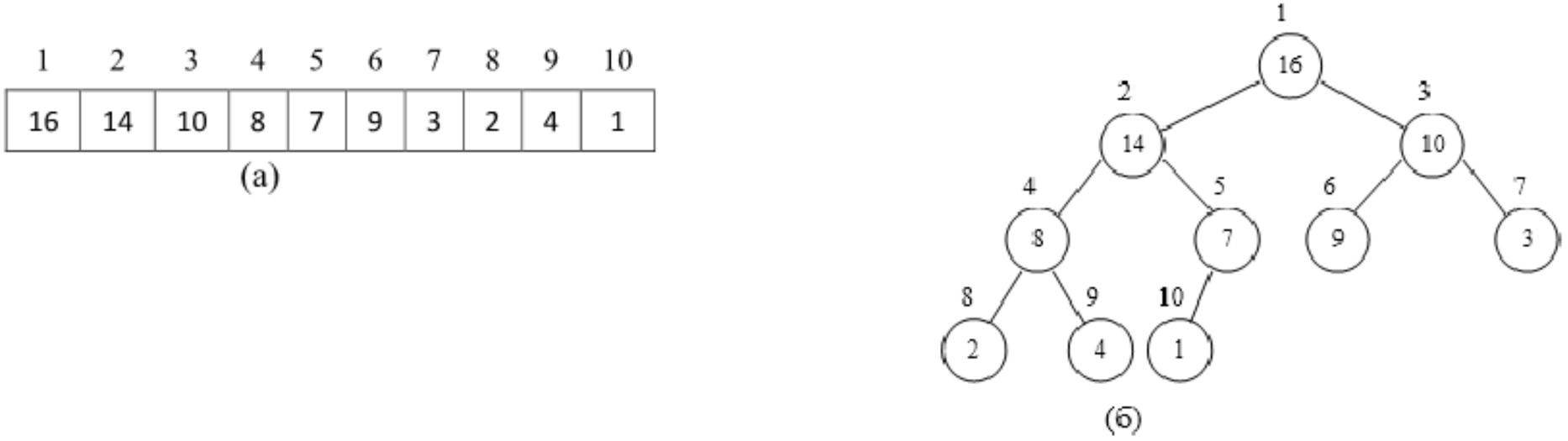


Рисунок 5.1 – Куча: как массив (а); как дерево (б). Внутри вершины показано её значение. Около вершины показан ее индекс в массиве

Куча может не занимать всего массива, и поэтому будем хранить не только массив A и его длину length[A], но также специальный параметр heap-size[A] (размер кучи), причём heap-size[A] ≤ length[A]. Куча состоит из элементов A[1],..., A[heap-size[A]]. Движение по дереву осуществляется процедурами:

Parent(i)

return[i/2]

Left (i)

return 2i

Right(i)

return 2i + 1

Элемент A[1] является корнем дерева.

Элементы, хранящиеся в куче, должны обладать основным свойством кучи (heap property): для каждой вершины i, кроме корня (т. е. при $2 \le i \le$ heap-size[A]),

$$A[PARENT(i)] \ge A[i] \qquad (5.1)$$

Отсюда следует, что значение потомка не превосходит значения предка. Таким образом, наибольший элемент дерева (или любого поддерева) находится в корневой вершине дерева (этого поддерева).

Высотой (height) вершины дерева называется высота поддерева с корнем в этой вершине (число рёбер в самом длинном пути с началом в этой вершине вниз по дереву к листу). Высота дерева, таким образом, совпадает с высотой его корня.

Высота этого дерева равна $\Theta(\log n)$, где n - число элементов в куче.

Основные операции над кучей:

- Процедура **HEAPIFY** позволяет поддерживать основное свойство

  A[PARENT(i)] ≥ A[i].

  Время работы составляет **O (log n)**.

- Процедура **BUILD-HEAP** строит кучу из исходного (неотсортированного) массива.

  Время работы **O (n)**.

- Процедура **HEAPSORT** сортирует массив, не используя дополнительной памяти.

  Время работы **O(n log n)**.

- Процедуры **HEAP-EXTRACT-MAX** (взятие наибольшего) и **HEAP-INSERT** (добавление элемента) используются при моделировании очереди с приоритетами на базе кучи.

  Время работы обеих процедур составляет **O (log n)**.

Сохранение основного свойства кучи (HEAPIFY)

Процедура HEAPIFY— важное средство работы с кучей. Её параметрами являются массив A и индекс i. Предполагается, что поддеревья с корнями LEFT(i) и RlGHT(i) уже обладают основным свойством.

Идея: если основное свойство не выполнено для вершины i, то её следует поменять с большим из её детей и т. д., пока элемент A[i] не «погрузится» до нужного места.

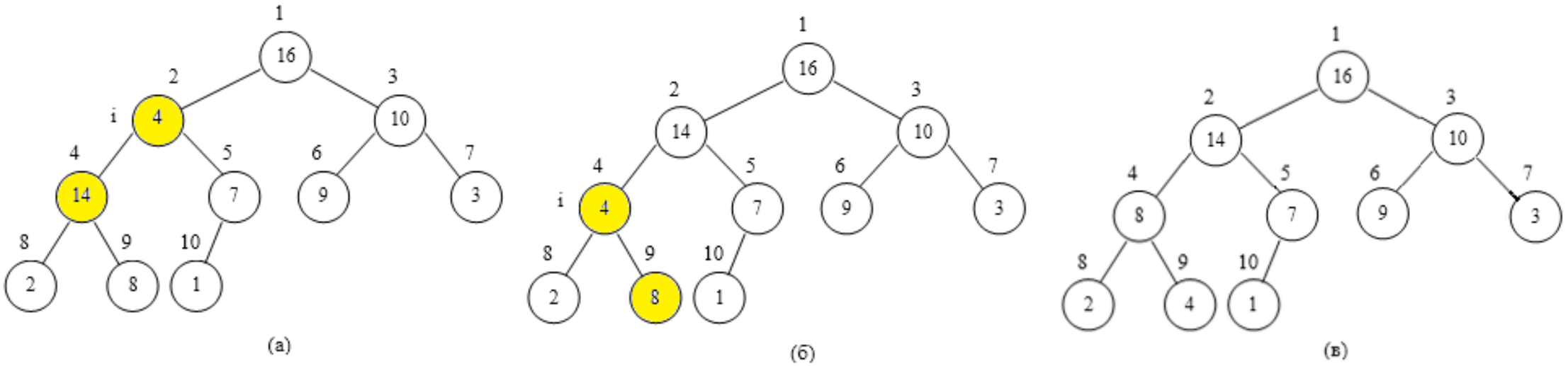


Рисунок 3.2 - Работа процедуры HEAPIFY(A, 2) при heap-size[A] = 10.
(а) Начальное состояние кучи. В вершине i=2 основное свойство нарушено. Чтобы восстановить его, необходимо поменять A[2] и A[4]. После этого (б) основное свойство нарушается в вершине с индексом 1. Рекурсивный вызов процедуры HEAPIFY(A,1) восстанавливает основное свойство и вершине с индексом 4 путём перестановки A[4] с A[9]. (в). После этого основное свойство выполнено для всех вершин, так что процедура HEAPIFY(A,9) уже ничего не делает

```
HEAPIFY (A, i)
1   l ← LEFT(i)                                  Θ(1)
2   r ← RIGHT(i)
3   if l ≤ heap-size[A] и A[l] > A[i]
4      then largest ← l
5      else largest ← i
6   if r ≤ heap-size[A] и A[r] > A[largest]
7      then largest ← r
8   if largest ≠ i
9      then обменять A[i] ↔ A[largest]
10     HEAPIFY (A, largest)
```

Время работы процедуры HEAPIFY

Пусть T(n) -время работы для поддерева, содержащего n элементов.

Поддерево с корнем состоит из n элементов, то поддеревья с корнями LEFT(i) и RlGHT(i) содержат не более чем по 2n/3 элементов каждое (наихудший случай — когда последний уровень в поддереве заполнен наполовину). Таким образом,

$$T(n) \le T(2n/3) + \Theta(1)$$

Позже докажем, что $T(n) = O(\log n)$. Эту же оценку можно получить и так: на каждом шаге мы спускаемся по дереву на один уровень, а высота дерева есть $O(\log n)$.

HEAPIFY (A, 2)

1 l ← LEFT(i)

2 r ← RIGHT(i)

3 if l ≤ heap-size[A] и A[l] > A[i]

4 then largest ← l

5 else largest ← i

6 if r ≤ heap-size[A] и A[r] > A[largest]

7 then largest ← r

8 if largest ≠ i

9 then обменять A[i] ↔ A[largest]

10 HEAPIFY (A, 4)

heap-size[A] = 10

2i = 4

2i+1 = 5

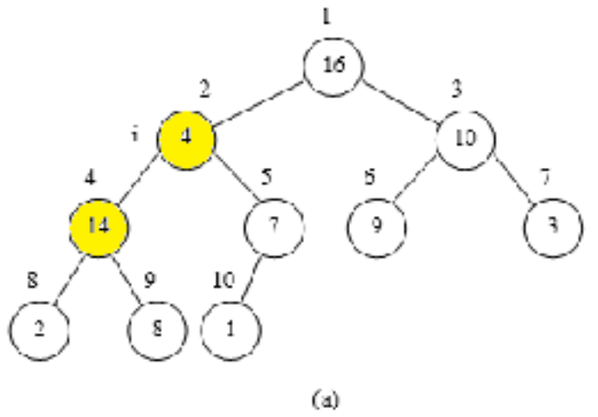
4≤10 & A[4] > A[2] (14 > 4)

largest = 4

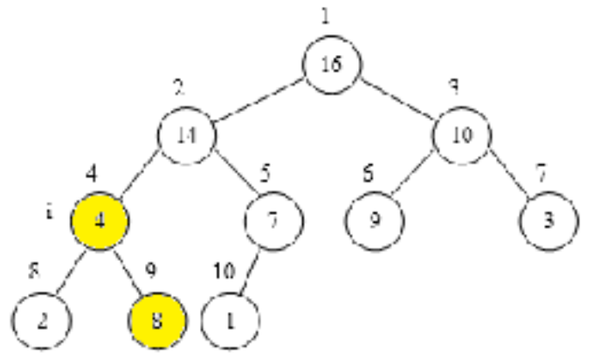
5≤10 & A[5] ≯ A[4] (7 ≯ 14)

4≠2

A[2] ↔ A[4]



(a)



(б)

HEAPIFY (A, 4)

1 l ← LEFT(i)

2 r ← RIGHT(i)

3 if l ≤ heap-size[A] и A[l] > A[i]

4 then largest ← l

5 else largest ← i

6 if r ≤ heap-size[A] и A[r] > A[largest]

7 then largest ← r

8 if largest ≠ i

9 then обменять A[i] ↔ A[largest]

10 HEAPIFY (A, 9)

heap-size[A] = 10

2i = 8

2i+1 = 9

8≤10 & A[8] > A[4] (2 > 4)

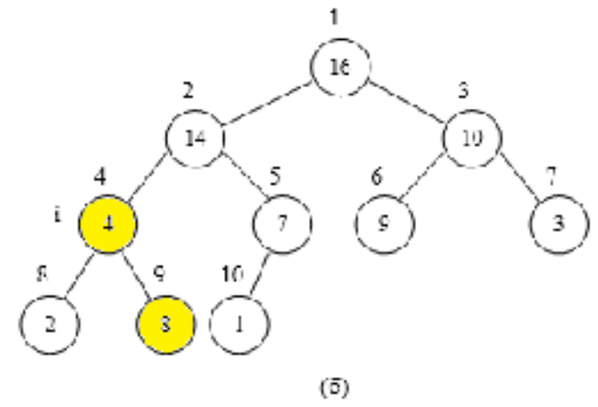
largest = 4

9≤10 & A[9] > A[4] (8 > 4)

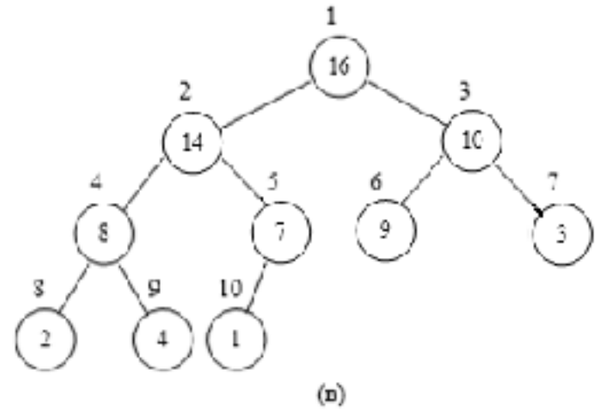
largest = 9

9≠4

A[4] ↔ A[9]



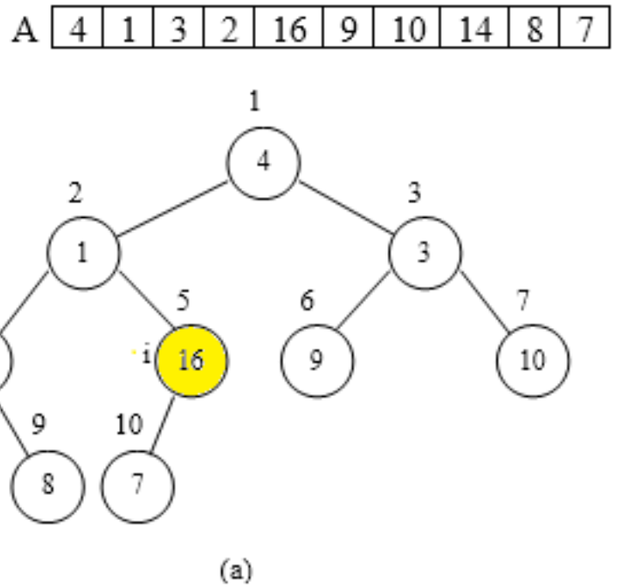
(б)



(в)

HEAPIFY (A, 9)	heap-size[A] = 10
11 l ← LEFT(i)	2i = 18
12 r ← RIGHT(i)	2i+1 = 19
13 if l ≤ heap-size[A] и A[l] > A[i]	18≤10
14 then largest ← l	
15 else largest ← i	largest = 9
16 if r ≤ heap-size[A] и A[r] > A[largest]	19≤10
17 then largest ← r	
18 if largest ≠ i	9≠9
19 then обменять A[i] ↔ A[largest]	
20 HEAPIFY (A, 9)	

Построение кучи



(а)

Пусть дан массив A [1.. n].

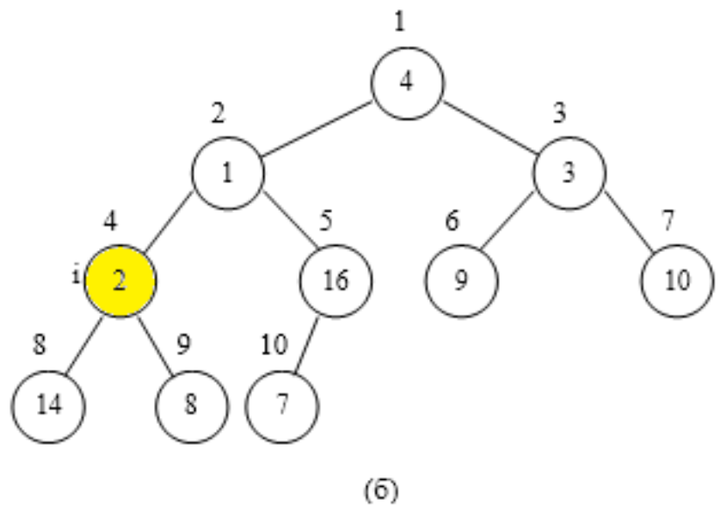
BUILD-HEAP(A)

1 heap-size[A] ← length[A]

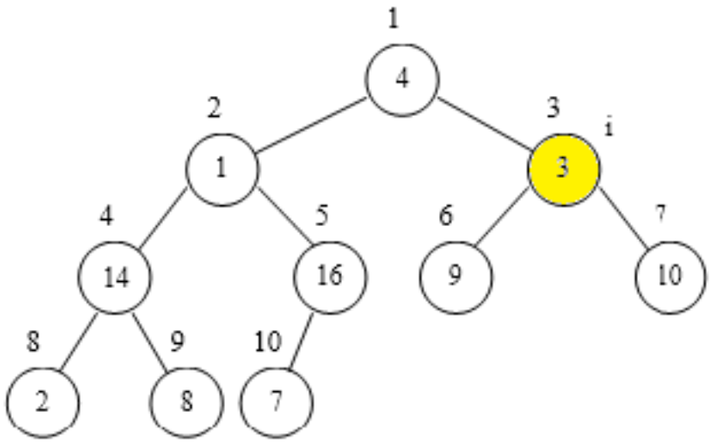
2 for i ← $\lfloor length[A]/2 \rfloor$ down to 1

3 do HEAPIFY (A, i)

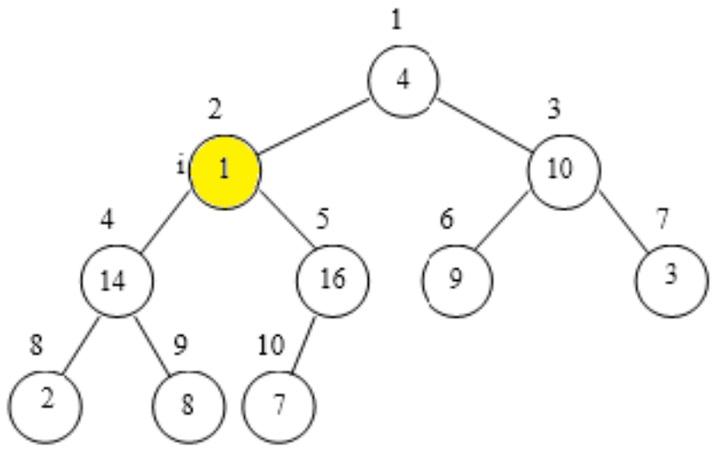
Пример работы процедуры BUILD-HEAP показан на рисунке 5.3.



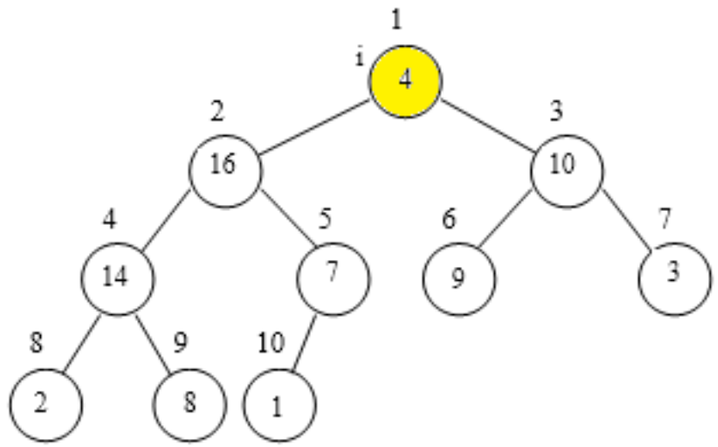
(б)



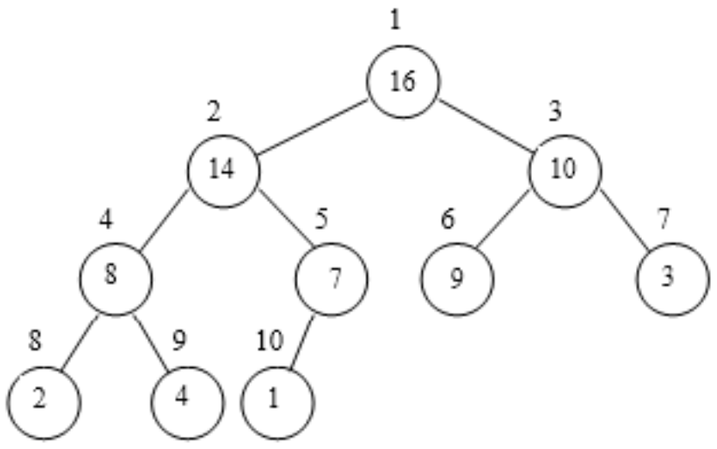
(в)



(г)



(д)



(е)

Рисунок 5.3 - Работа процедуры BUILD-HEAP. Показано состояние данных перед каждым вызовом процедуры HEAPIFY в строке 3

Время работы процедуры BUILD-HEAP не превышает $O(n \log n)$. Действительно, процедура HEAPIFY вызывается $O(n)$ раз, а каждое её выполнение требует времени $O(\log n.)$. Однако эту оценку можно улучшить.

Дело в том, что время работы процедуры HEAPIFY зависит от высоты вершины, для которой она вызывается (и пропорционально этой высоте). Поскольку число вершин высоты h в куче из n элементов не превышает $[n/2^{h+1}]$, а высота всей кучи не превышает $[\log n]$, время работы процедуры BUILD-HEAP не превышает:

$$\sum_{h=0}^{[log\, n]} \left[\frac{n}{2^{h+1}}\right] O(h) = O\left(n \sum_{h=0}^{[log\, n]} \frac{h}{2^h}\right)$$

Полагая x = 1/2, получаем верхнюю оценку для суммы в правой части:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

Таким образом, время работы процедуры BUILD-HEAP составляет:

$$O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n).$$

$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ при $|x| < 1$.

$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$

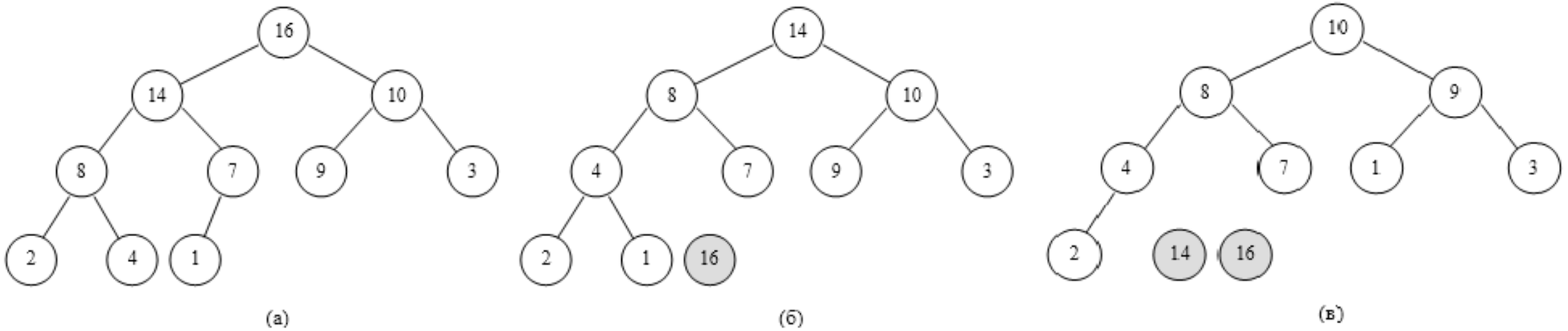
Алгоритм сортировки с помощью кучи

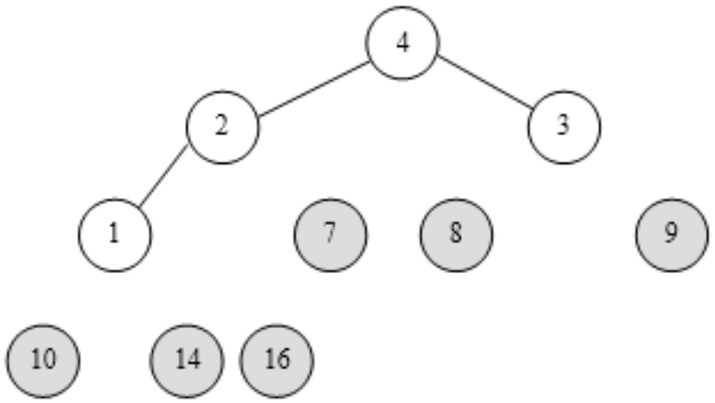
Алгоритм сортировки с помощью кучи состоит из двух частей. Сначала вызывается процедура BUILD-HEAP, после выполнения которой массив становится кучей. Идея второй части проста: максимальный элемент массива теперь находится в корне дерева (A[1]). Его следует поменять с A[n], уменьшить размер кучи на 1 и восстановить основное свойство в корневой вершине (поскольку поддеревья с корнями LEFT(l) и RlGHT(l) не утратили основного свойства кучи - это можно сделать с помощью процедуры HEAPIFY). После этого в корне будет находиться максимальный из оставшихся элементов. Так делается до тех пор, пока в куче не останется всего один элемент.

```
HEAPSORT(A)
1    BUILD-HEAP(A)
2    for i ← length[A] down to 2
3        do поменять A[1] ↔ A[i]
4            heap-size[A] ← heap-size[A] –1
5            HEAPIFY(A,1)
```

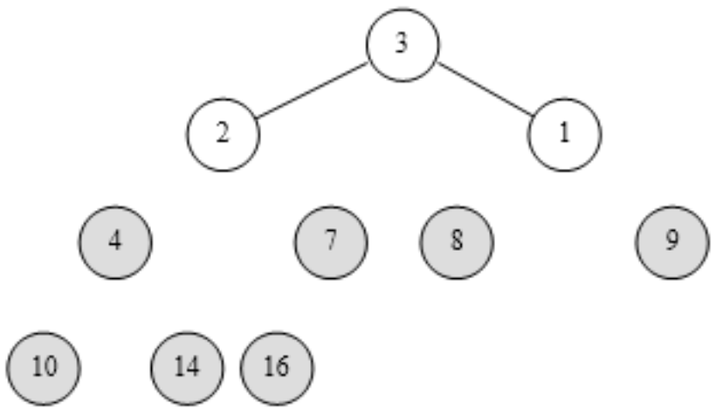
Работа второй части алгоритма показана па рисунке 5.4.

Время работы процедуры HEAPSORT составляет O(n log n). Действительно, первая часть (построение кучи) требует времени O(n), а каждое из n - 1 выполнений цикла for занимает время O(log n).

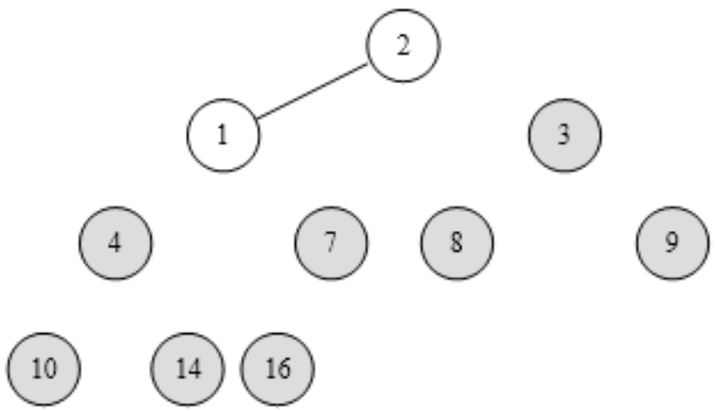




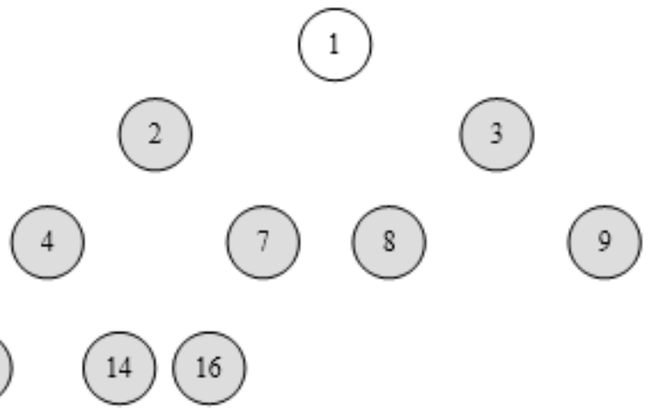
(ж)



(з)



(и)



(к)

A	1	2	3	4	7	8	9	10	14	16

Рисунок 5.4 - Работа процедуры HEAPSORT. Показано состояние массива перед каждым вызовом процедуры HEAPIFY. Зачерненные элементы уже не входят в кучу

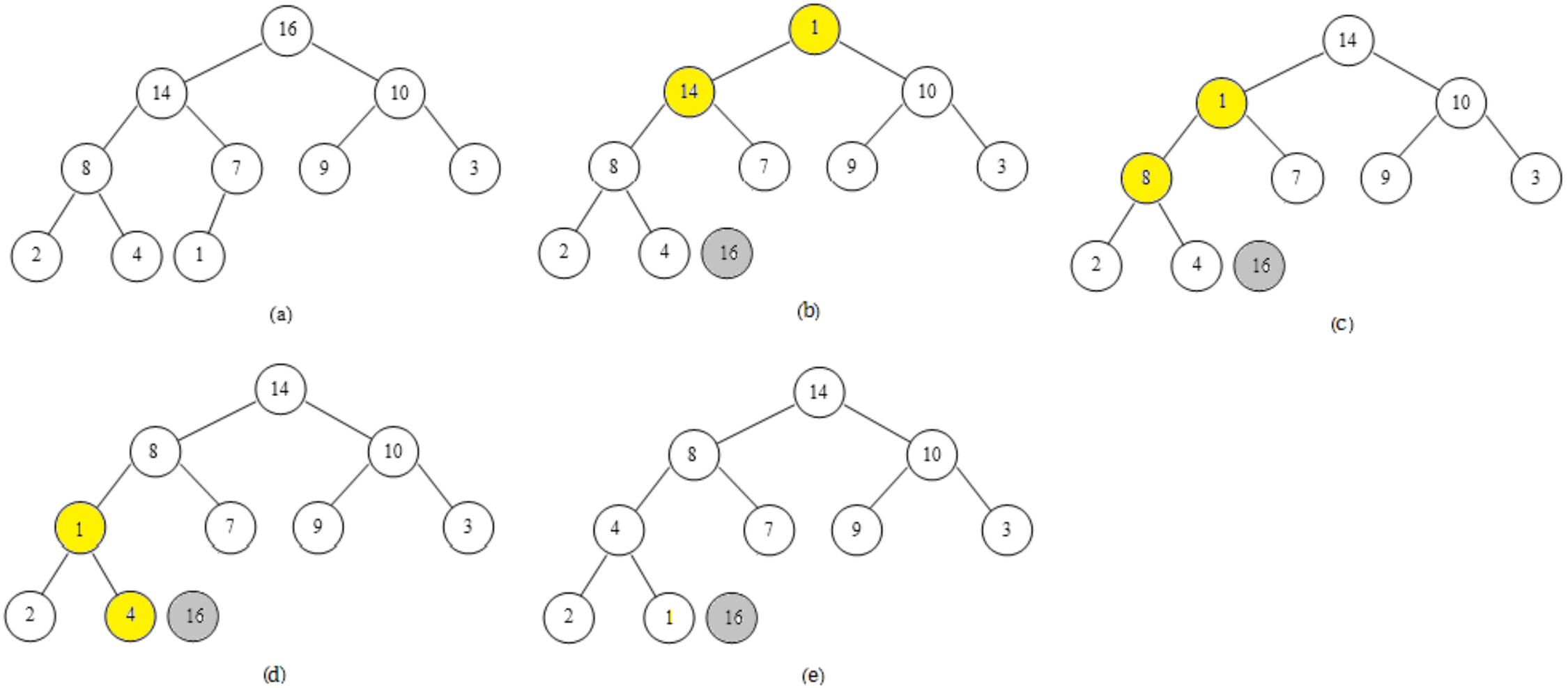


Рисунок 5.5 – Демонстрацию работы процедуры HEAPIFY(A,1)

Реализацию очереди с приоритетами

Элементы множества хранятся в виде кучи. При этом максимальный элемент находится в корне, так что операция MAXIMUM требует времени Θ(1). Чтобы изъять максимальный элемент из очереди, нужно действовать так же, как и при сортировке:

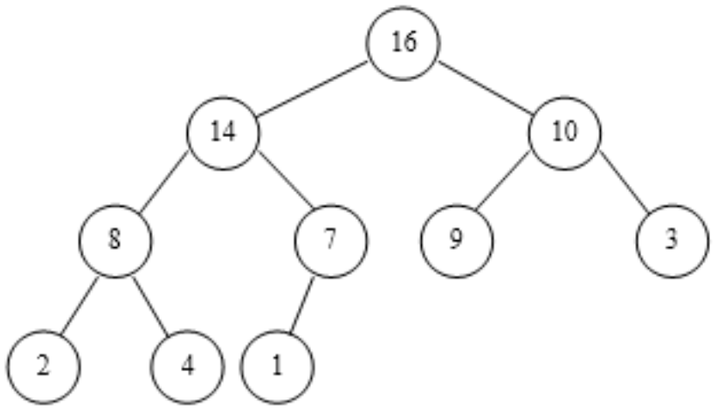
```
HEAP-EXTRACT-MAX(A)
1    if heap-size[A] < 1
2          then ошибка: «очередь пуста»
3    max ← A[1]
4    A[1] ← A[heap-size[A]]
5    heap-size[A] ← heap-size[A] –1
6    HEAPIFY(A,1)
7    return max
```

Время работы составляет O(logn) (процедура HEAPIFY вызывается один раз). Чтобы добавить элемент к очереди, его следует добавить в конец кучи (как лист), а затем дать ему «всплыть» до нужного места.

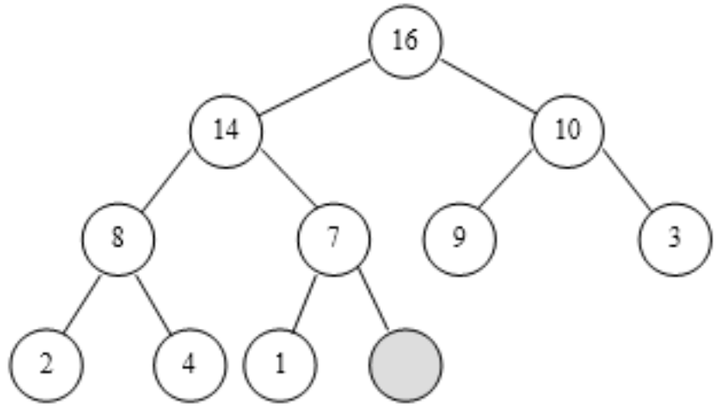
```
HEAP-INSERT (A, key)
1    heap-size[A] ← heap-size[A] + 1
2    i ← heap-size[A]
3    while i > 1 и A[PARENT(i)] < key
4          do A[i] ← A[PARENT(i)]
5          i ← PARENT(i)
6    A[i] ← key
```

Время работы составляет O(logn), поскольку «подъём» нового листа занимает не более log n шагов (индекс i после каждой итерации цикла while уменьшается по крайне мере вдвое). Итак, все операции над очередью с приоритетами из n элементов требуют времени O(log n).

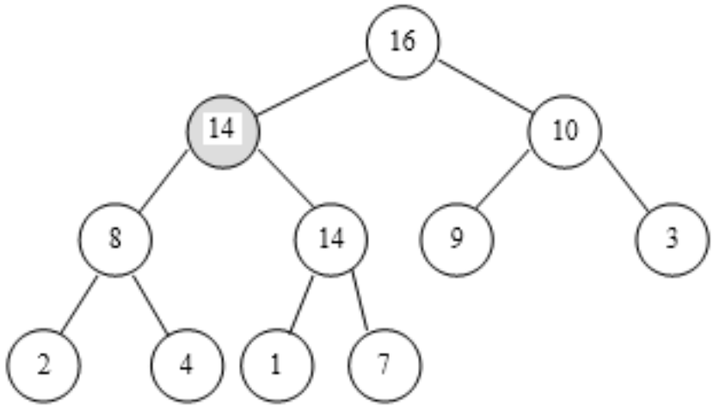
Пример работы процедуры HEAP-INSERT показан на рисунке 5.6.



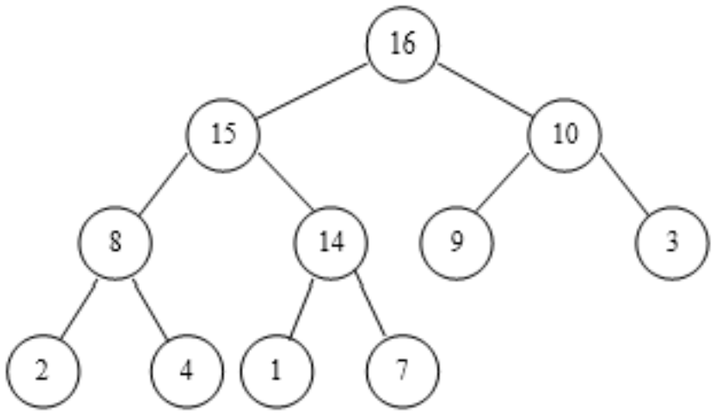
(а)



(б)



(в)



(г)

Рисунок 5.6 - Работа процедуры HEAP-INSERT. Добавляет элемент с ключевым значением 15 (темный кружок обозначает место для этого элемента)

HEAP-INSERT (A, 15)

1 heap-size[A] ← heap-size[A] + 1

2 i ← heap-size[A]

3 while i > 1 и A[PARENT(i)] < key

4 do A[i] ← A[PARENT(i)]

5 i ← PARENT(i)

6 A[i] ← key

heap-size[A] = 10 key = 15

1 heap-size[A] = 11

2 i = 11

3 11 > 1 & A[PARENT(11)] < 15 (7 < 15)

4 A[11] ← A[PARENT(5)] (A[11] = 7)

5 i = 5

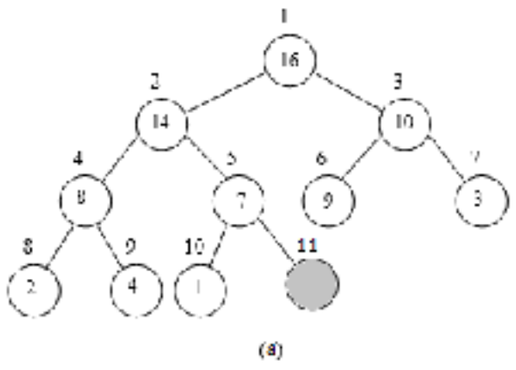
3 5 > 1 & A[PARENT(5)] < 15 (14 < 15)

4 A[5] ← A[PARENT(2)] (A[5] = 14)

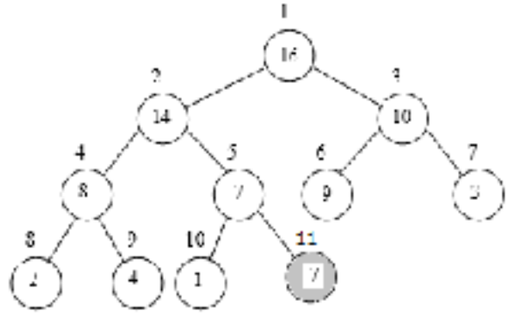
5 i = 2

3 2 > 1 & A[PARENT(2)] < 15 (16 ≮ 15)

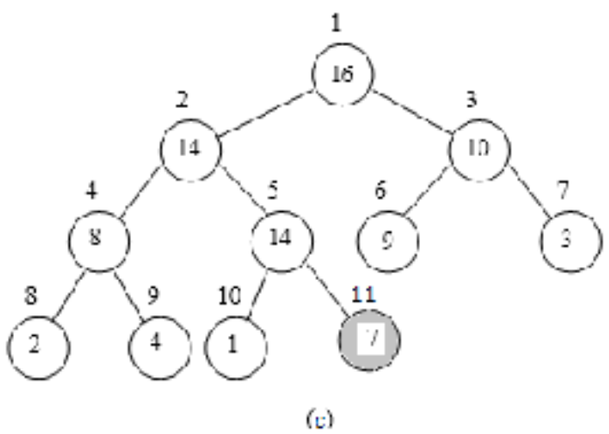
6 A[2] ← 15



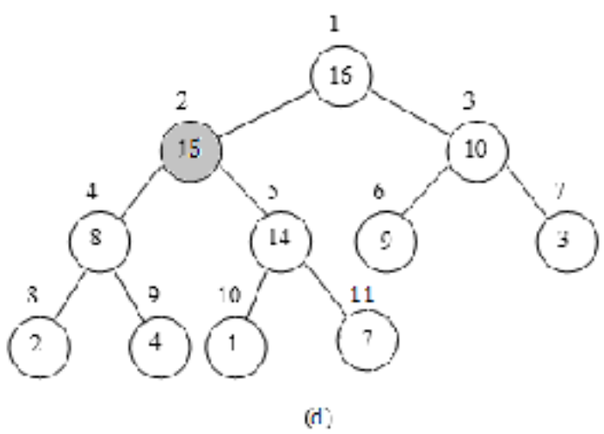
(a)



(b)



(c)



(d)

Упражнения:

1. Является ли последовательность значений (23, 17, 14, 6, 13, 10, 1, 5, 7, 12) кучей?
2. Проиллюстрируйте работу процедуры HEAPIFY (A, 3) с массивом A = (27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0).
3. Проиллюстрируйте работу процедуры BUILD-HEAP над входным массивом A = (5, 3, 17, 10, 84, 19, 6, 22, 9).
4. Проиллюстрируйте работу процедуры HEAPSORT над входным массивом A = (5, 13, 2, 25, 7, 17, 20, 8, 4).
5. Проиллюстрировать работу процедуры HEAP_EXTRACT_MAX над пирамидой A = (15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1).
6. Проиллюстрировать работу процедуры MAX-HEAP-INSERT (A, 10) над пирамидой A = (15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1).

Задача

Можно построить кучу, последовательно добавляя элементы с помощью процедуры HEAP-INSERT. Рассмотрим следующий алгоритм:

```
BUILD-HEAP'(A)
1       heap-size[A] ← 1
2       for i ← 2 to length[A]
3               do HEAP-INSERT (A, A[i])
```

1) Построить кучу над входным массивом A = (5, 3, 17, 10, 84, 19, 6, 22, 9) с помощью вставок.

2) Запустить процедуры BUILD-HEAP и BUILD-HEAP' для одного и того же массива. Всегда ли они создадут одинаковые кучи? (Докажите или приведите контрпример.)

3) Докажите, что время работы процедуры BUILD-HEAP' в худшем случае составляет O(n log n) (где n - количество элементов).

ЛЕКЦИИ № 6

Сортировка за линейное время. Нижние оценки для сортировки. Сортировка подсчетом. Цифровая сортировка. Пример

Все упомянутые алгоритмы проводят сортировку, основываясь исключительно на попарных сравнениях элементов, поэтому их иногда называют сортировками сравнением (comparison sort). Всякий алгоритм такого типа сортирует n элементов за время не меньше $\Omega(n \log n)$ в худшем случае. Тем самым алгоритмы сортировки слиянием и с помощью кучи асимптотически оптимальны: не существует алгоритма сортировки сравнением, который превосходил бы указанные алгоритмы более чем в конечное число раз.

Рассмотрим три алгоритма сортировки, работающих за линейное время:

- сортировка подсчётом;
- цифровая сортировка;
- карманная сортировка (входные числа генерируются случайным образом и равномерно распределены в интервале [0,1).

Они улучшают оценку $\Omega(n \log n)$ за счёт того, что используют не только попарные сравнения, но и внутреннюю структуру сортируемых объектов.

Сортировка за линейное время

Алгоритм сортировки подсчётом (counting sort) применим, если каждый из n элементов сортируемой последовательности — целое положительное число в известном диапазоне (не превосходящее заранее известного **k**). Если **k = O(n)**, то алгоритм сортировки подсчётом работает за время **O (n)**.

Идея этого алгоритма в том, чтобы для каждого элемента **x** предварительно подсчитать, сколько элементов входной последовательности меньше **x**, после чего записать **x** напрямую в выходной массив в соответствии с этим числом (если, скажем, **10** элементов входного массива меньше **x**, то в выходном массиве **x** должен быть записан на место номер **11**). Если в сортируемой последовательности могут присутствовать равные числа, эту схему надо слегка модифицировать, чтобы не записать несколько чисел на одно место.

В приводимом ниже псевдокоде используются:

- вспомогательный массив **C[1..k]** из **k** элементов;
- входная последовательность записана в массиве **A[1..n]**;
- отсортированная последовательность записывается в массив **B[1..n]**.

```
COUNTING-SORT (A, B, k)
1     for i ← 1 to k
2               do C[i] ← 0
3     for j ← 1 to length[A]
4               do C[A[j]] ← C[A[j]] + 1
5     ⋗ C[i] равно количеству элементов, равных i
6     for i ← 2 to k
7               do C[i] ← C[i] + C[i-1]
8     ⋗ C[i] равно количеству элементов, не превосходящих i
9     for j ← length[A] down to 1
10              do B[C[A[j]]] ← A[j]
11              C[A[j]] ← C[A[j]] -1
```

Работа алгоритма сортировки подсчётом проиллюстрирована на рисунке 6.1.

```
COUNTING-SORT (A, B, 6)
1     for i ← 1 to 6
            do C[i] ← 0
2     for j ← 1 to length[A]=8
3           do C[A[j]] ← C[A[j]] + 1
4     ⋗ C[i] равно количеству элементов, равных i
5     for i ← 2 to k
6           do C[i] ← C[i] + C[i-1]
7     ⋗ C[i] равно количеству элементов, не превосходящих i
8     for j ← length[A] down to 1
9           do B[C[A[j]]] ← A[j]
10          C[A[j]] ← C[A[j]] -1
```

$A = (3, 6, 4, 1, 3, 4, 1, 4)$
$\quad\;\; 1\;\; 2\;\; 3\;\; 4\;\; 5\;\; 6\;\; 7\;\; 8$

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

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

$C = (2, 2, 4, 7, 7, 8)$
$\quad\;\; 1\;\; 2\;\; 3\;\; 4\;\; 5\;\; 6$

for $j \leftarrow 8$ down to 1

$B[C[A[8]]] = B[C[4]] = B[7] \leftarrow A[8]=4,$

$C[A[8]] = C[4] \leftarrow C[4]-1 = 7-1 = 6$

$C = (2, 2, 4, 6, 7, 8)$
$\quad\;\; 1\;\; 2\;\; 3\;\; 4\;\; 5\;\; 6$

$B = (\qquad\qquad 4 \qquad)$
$\quad\;\; 1\;\; 2\;\; 3\;\; 4\;\; 5\;\; 6\;\; 7\;\; 8$

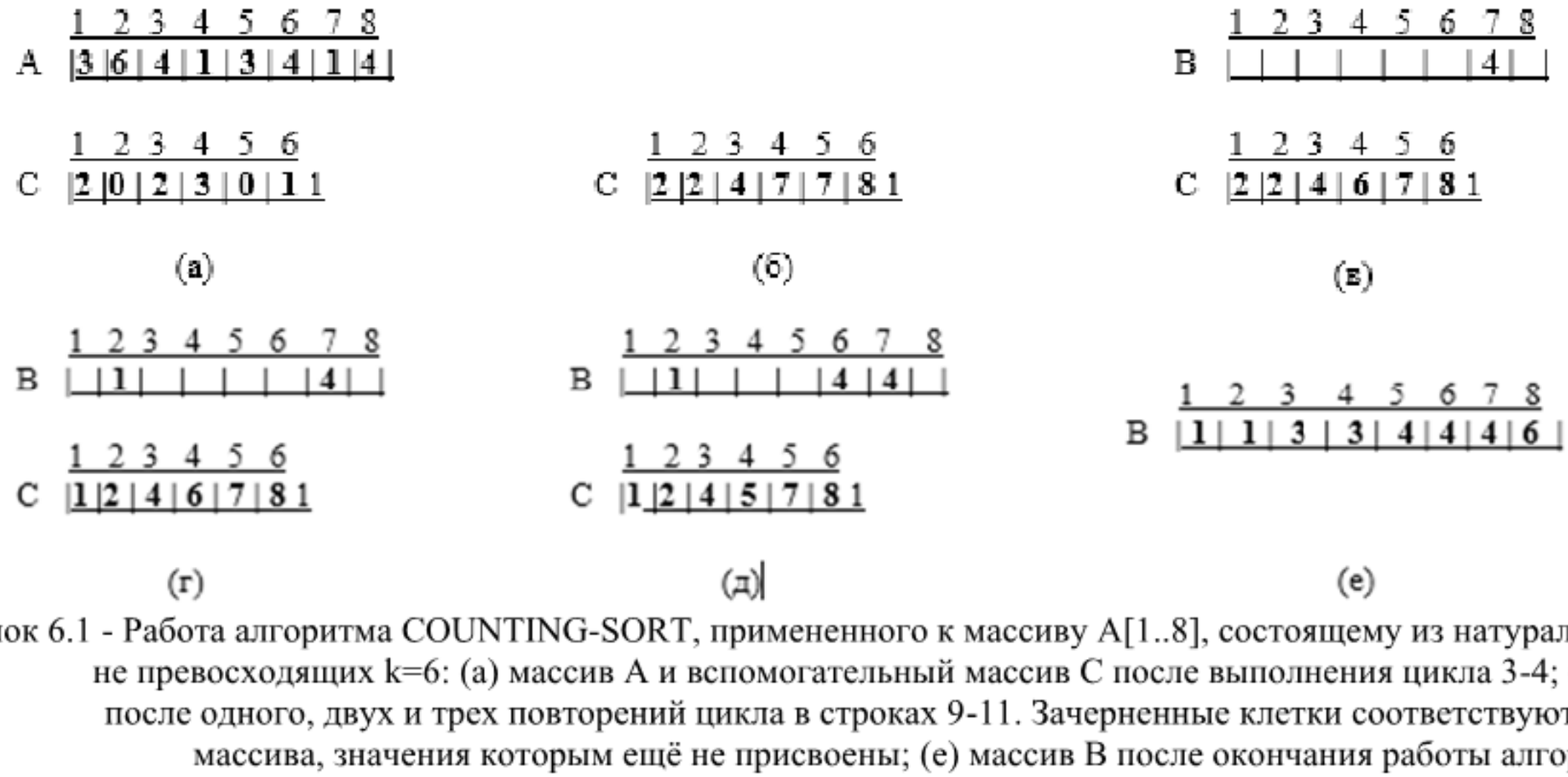


Рисунок 6.1 - Работа алгоритма COUNTING-SORT, примененного к массиву A[1..8], состоящему из натуральных чисел, не превосходящих k=6: (а) массив A и вспомогательный массив C после выполнения цикла 3-4; (б) массив C после одного, двух и трех повторений цикла в строках 9-11. Зачерненные клетки соответствуют элементам массива, значения которым ещё не присвоены; (е) массив B после окончания работы алгоритма

Алгоритм сортировки подсчётом обладает важным свойством, называемым **устойчивостью** (is stable). Именно, если во входном массиве присутствует несколько равных чисел, то в выходном массиве они стоят в том же порядке, что и во входном. Это свойство имеет смысл, только если в массиве вместе с числами записаны дополнительные данные. Более точно, представим себе, что сортируем не просто числа, а пары (t, x), где t — число от 1 до k, а x - произвольный объект, и хотим переставить их так, чтобы первые компоненты пар шли в неубывающем порядке. Алгоритм сортировки подсчётом позволяет это сделать, причём относительное расположение пар с равными первыми компонентами не меняется. Это свойство и называется **устойчивостью**.

Цифровая сортировка (поразрядная сортировка)

В компьютерах цифровая сортировка иногда используется для упорядочения данных, содержащих несколько полей. Пусть, например, нам надо отсортировать, последовательность дат.

329

457

657

839

436

720

355

Программу для цифровой сортировки попробуйте написать самостоятельно, предполагая, что каждый элемент n-элементного массива A состоит из d цифр, причем цифра номер 1 — младший разряд, а цифра номер d — старший.

Это можно сделать с помощью любого алгоритма сортировки, сравнивая даты следующим образом: сравнить годы, если годы совпадают — сравнить месяцы, если совпадают и месяцы — сравнить числа.

Вместо этого, однако, можно просто трижды отсортировать массив дат - с помощью устойчивого алгоритма: сначала по дням, потом по месяцам, потом по годам.

329	**720**	**720**	**329**
457	**355**	**329**	**355**
657	**436**	**436**	**436**
839 →	**457** →	**839** →	**457**
436	**657**	**355**	**657**
720	**329**	**457**	**720**
355	**839**	**657**	**839**
↑	↑	↑	

```
RADIX-SORT (A, d)          A = (329, 457, 657, 839, 436, 720, 355)
1     t ← 1
2     for i ← 1 to d
3         do RCSORT (A, t)
4         t ←t * 10
```

```
RCSORT (A, t)
1.  for j ← 0 to 9
2.        do C[j] ← 0
3.  for j ← 0 to length[A] - 1
4.        do C[(A[j]mod(t*10)) div t] ← C[(A[j]mod(t*10)) div t] + 1
5.  for j ← 1 to 9
6.        do C[j] ← C[j - 1] + C[j]
7.  for j ← length[A]-1 down to 0
8.        do B[C[(A[j]mod(t*10)) div t]] ← A[j]
9.        do C[(A[j]mod(t*10)) div t]] ← C[(A[j]mod(t*10)) div t] – 1
10. for j ← 0 to length[A] - 1
11.       do A[j] ← B[j]
```

RADIX-SORT (A, 3) A = (329, 457, 657, 839, 436, 720, 355)

```
1   t ← 1
2   for i ← 1 to 3                    1
3         do RXSORT (A, t)
4         t ←t * 10
```

RXSORT (A, 1)

```
1.  for j ← 0 to 9
2.        do C[j] ← 0
3.  for j ← 0 to length[A]-1=6
4.        do C[(A[0]mod(t*10)) div t] ← C[(A[0]mod(t*10)) div t] + 1
          do C[(329mod10)div1](= C[9]=0) ← C[9] +1
5.  for j ← 1 to 9
6.        do C[j] ← C[j - 1] + C[j]
7.  for j ← length[A]-1=6 down to 0
8.        do B[C[(A[6]mod(t*10)) div t]] = B[C[355mod(t*10)) div t]] =
          =B[C[5]] = B[1] ← A[6] =355
```

C = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
0 1 2 3 4 5 6 7 8 9

C = (0, 0, 0, 0, 0, 1, 1, 2, 0, 2)
0 1 2 3 4 5 6 7 8 9

C = (0, 0, 0, 0, 0, 1, 2, 4, 4, 6)
0 1 2 3 4 5 6 7 8 9

A = (329, 457, 657, 839, 436, 720, 355)
0 1 2 3 4 5 6

B = (0, 355, 0, 0, 0, 0, 0)
0 1 2 3 4 5 6

9. do C[(A[6]mod(t*10)) div t]] (=C[355mod(t*10)) div t]] = C[5]=1)←

←C[(A[6]mod(t*10)) div t](= C[5]=1) - 1

$C = (0, 0, 0, 0, 0, 0, 2, 4, 4, 6)$
$\quad\quad 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$

7. for j ← 5 down to 0

$A = (329, 457, 657, 839, 436, 720, 355)$
$\quad\quad 0\quad 1\quad 2\quad 3\quad 4\quad 5\quad 6$

8. do B[C[(A[5]mod(t*10)) div t]]= B[C[720mod(t*10)) div t]] =

=B[C[0]]= B[0] ← A[5] =720

$B = (720, 355, 0, 0, 0, 0, 0)$
$\quad\quad 0\quad 1\ 2\ 3\ 4\ 5\ 6$

9. do C[(A[5]mod(t*10)) div t]] = C[(720mod(t*10)) div t]] = C[0] ←

← C[0]- 1

$C = (-1, 0, 0, 0, 0, 0, 2, 4, 4, 6)$
$\quad\quad 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$

7. for j ← 4 down to 0

$A = (329, 457, 657, 839, 436, 720, 355)$
$\quad\quad 0\quad 1\quad 2\quad 3\quad 4\quad 5\quad 6$

8. do B[C[(A[4]mod(t*10)) div t]]= B[C[436mod(t*10)) div t]] = B[C[6]] = B[2] ←

← A[4] =436

$B = (720, 355, 436, 0, 0, 0, 0)$
$\quad\quad 0\quad 1\quad 2\quad 3\ 4\ 5\ 6$

9. do C[(A[4]mod(t*10)) div t]] = C[6] =2← C[(A[4]mod(t*10)) div t] =

= C[6]=2 – 1

$C = (-1, 0, 0, 0, 0, 0, 1, 4, 4, 6)$
$\quad\quad 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$

7. for j ← 3 down to 0

8. do B[C[(A[j]mod(t*10)) div t]]= B[C[(A[3]mod(t*10)) div t]] = B[C[839mod(t*10)) div t]] = B[C[9]] = B[6] ←

← A[3]=839

9. do C[(A[3]mod(t*10)) div t]] = C[(839mod(t*10)) div t]]= C[6] ← C[6] – 1= 5

…

$B = (720, 355, 436, 457, 657, 329, 839)$

10. for j ← 0 to length[A] - 1

11. do A[j] ← B[j]

A = (720, 355, 436, 457, 657, 329, 839)

RADIX-SORT (A, 3)

2 for i ← 2 to 3

3 do RXSORT (A, 10)

4 t ←t * 10

A = (720, 329, 436, 839, 355, 457, 657)

RADIX-SORT (A, 3)

2 for i ← 3 to 3

3 do RXSORT (A, 100)

4 t ←t * 10

A = (329, 355, 436, 457, 657, 720, 839)

digit(x,i) возвращает i-й разряд числа x.

```
radixSort(int[] A):
     for i = 1 to m
         for j = 0 to k - 1
             C[j] = 0
         for j = 0 to n - 1
             d = digit(A[j], i)
             C[d]++
         count = 0
         for j = 0 to k - 1
             tmp = C[j]
             C[j] = count
             count += tmp
         for j = 0 to n - 1
             d = digit(A[j], i)
             B[C[d]] = A[j]
             C[d]++
        A = B
```

Изначально запускаем функцию так radixSort(A, 0, A.length−1,1)

```
radixSort(int[] A, int l, int r, int d):
     if d > m or l >= r
         return
     for j = 0 to k + 1
         cnt[j] = 0
     for i = l to r
         j = digit(A[i], d)
         cnt[j + 1]++
     for j = 2 to k
         cnt[j] += cnt[j - 1]
     for i = l to r
         j = digit(A[i], d)
         c[l + cnt[j]] = A[i]
         cnt[j]--
    for i = l to r
         A[i] = c[i]
    radixSort(A, l, l + cnt[0] - 1, d + 1)
    for i = 1 to k
         radixSort(A, l + cnt[i - 1], l + cnt[i] - 1, d + 1)
```

Время работы зависит от времени работы выбранного устойчивого алгоритма. Если цифры могут принимать значения от 1 до k, где k не слишком велико, то очевидный выбор — сортировка подсчётом. Для n чисел с d знаками от 0 до k-1 каждый проход занимает время $\Theta(n+k)$ поскольку мы делаем d проходов, время работы цифровой сортировки равно $\boldsymbol{\Theta(dn + kd)}$. Если d постоянно и $k = O(n)$, то цифровая сортировка работает за линейное время.

При цифровой сортировке важно правильно выбрать основание системы счисления, поскольку от него зависит размер требуемой дополнительной памяти и время работы.

К сожалению, цифровая сортировка, опирающаяся на сортировку подсчётом, требует ещё одного массива (того же размера, что и сортируемый) для хранения промежуточных результатов, в то время как многие алгоритмы сортировки сравнением обходятся без этого. Поэтому, если надо экономить память, алгоритм быстрой сортировки может оказаться предпочтительнее.

Алгоритм	Время работы в наихудшем случае	Время работы в среднем случае/ожидаемое
Сортировка вставкой	$\Theta(n^2)$	$\Theta(n^2)$
Сортировка слиянием	Θ(nlogn)	Θ(nlogn)
Пирамидальная сортировка	O(nlogn)	—
Быстрая сортировка	$\Theta(n^2)$	Θ(nlogn) (ожидаемое)
Сортировка подсчетом	$\Theta(k+n)$	$\Theta(k+n)$
Поразрядная сортировка	$\Theta(d(n+k))$	$\Theta(d(n+k))$
Карманная сортировка	$\Theta(n^2)$	$\Theta(n)$ (в среднем случае)

Минимум и максимум

Сколько сравнений необходимо, чтобы во множестве из n чисел найти наименьшее? За n - 1 сравнении это сделать легко: надо последовательно перебирать все числа, храня значение наименьшего числа из уже просмотренных. Запишем этот алгоритм, считая, что числа заданы в виде массива A длины n.

```
MINIMUM(A)
1       min ← A[1]
2       fot i ← 2 to length[A]
3           do if min > A[i]
4               then min ← A[i]
5       return min
```

Разумеется, аналогичным образом можно найти и максимум.

Можно ли найти минимум еще быстрее? Нет, и вот почему. Рассмотрим алгоритм нахождения наименьшего числа как турнир среди n чисел, а каждое сравнение как матч, в котором меньшее число побеждает. Чтобы победитель был найден, каждое из остальных чисел должно проиграть, по крайней мере, один матч, так что меньше n-1 сравнений быть не может, и алгоритм MINIMUM оптимален по числу сравнений.

Одновременный поиск минимума и максимума

Иногда бывает нужно найти одновременно минимальный и максимальный элементы множества. Представим себе программу, которая должна уменьшить рисунок (набор точек, заданных своими координатами) так, чтобы он уместился на экране. Для этого нужно найти максимум и минимум по каждой координате.

Найдем сначала минимум, а потом максимум, затратив на каждый из них по n - 1 сравнений, то всего будет 2n-2 сравнения, что асимптотически оптимально. Можно, однако, решить эту задачу всего за 3[n/2] - 2 сравнений. Именно, будем хранить значения максимума и минимума уже просмотренных чисел, а очередные числа будем обрабатывать по два таким образом: сначала сравним два очередных числа друг с другом, а затем большее из них сравним с максимумом, а меньшее — с минимумом. При этом на обработку двух элементов затратим три сравнения вместо четырёх (кроме первой пары, где понадобится всего одно сравнение).

Упражнения:

1. Воспользовавшись в качестве образца рисунке 6.1, проиллюстрируйте работу процедуры COUNTING-SORT над входным массивом A = (6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2).
2. Воспользовавшись в качестве образца рис.6.2, проиллюстрируйте работу процедуры RADIX-SORT со следующим списком русских слов:

БАР, МАК, ЛИС, ЕГО, БУМ, ВЕС, БЫК, ВУЗ, ЕЛЬ

бар	вуз	мак	бар
мак	буй	бар	буй
лис	мак	его	бум
его	бум	вес	вес
бум	его	лис	вуз
вес	бар	ель	его
буй	лис	вуз	ель
вуз	вес	буй	лис
ель	ель	бум	мак

А б в г е з и й к л м р с у ь

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

б а р м к л и с е г о у в й з ь

списком английских слов:

BAR, EAR, TAR, COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, DIG, BIG, TEA, NOW, FOX.

ЛЕКЦИЯ № 7
Рекуррентные соотношения. Метод подстановки. Метод итераций. Общий рецепт

Как было сказано выше, если алгоритм рекуррентно вызывает сам себя, время его работы можно описать с помощью рекуррентного соотношения. **Рекуррентное соотношение** – это уравнение или неравенство, описывающее функцию с использованием её самой, но только с меньшими аргументами. Например, время работы процедуры MERGE-SORT **T(n)** в самом неблагоприятном случае описывается с помощью следующего рекуррентного соотношения:

$$T(n) = \begin{cases} \Theta(1) & \text{при } n = 1, \\ 2T(n/2) + \Theta(n) & \text{при } n > 1. \end{cases} \tag{7.1}$$

Решением этого соотношения является функция $T(n) = \Theta(n \log n)$.

Рассмотрим **4 способа**, позволяющие решить рекуррентное соотношение, то есть найти асимптотическую оценку (Θ или O):

1. **Метод подстановки**. Угадывается оценка, а затем доказывается она по индукции, подставляя угаданную формулу в правую часть соотношения.
2. **Метод итераций.** Можно «развернуть» рекуррентную формулу, получив при этом сумму, которую можно затем оценить.
3. **Применение основной теоремы о рекуррентных соотношениях**. В основном методе рекуррентные соотношения представляются в таком виде:

$$T(n) = aT(n/b) + f(n), \qquad (7.2)$$

где $a \geq 1$, $b > 1$, а функция $f(n)$ – это заданная функция; для применения этого метода необходимо запомнить четыре различных случая, но после этого определение асимптотических границ во многих простых рекуррентных соотношениях становится очень простым.

4. **Решение с помощью характеристического уравнения** (однородного и неоднородного типов).

Технические детали

В практике в процессе формулировки и решения рекуррентных соотношений определенные технические моменты опускаются. Так, например, есть одна особенность – это допущение о том, что аргумент функции является целочисленным. Обычно время работы $T(n)$ алгоритма определено лишь для целочисленных n, поскольку в большинстве алгоритмов количество входных элементов выражается целым числом. Например, рекуррентное соотношение, описывающее время работы процедуры MERGE-SORT в наихудшем случае, на самом деле записывается так:

$$T(n) = \begin{cases} \Theta(1) & \text{при } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{при } n > 1. \end{cases} \quad (7.3)$$

$$T(n) = \begin{cases} \Theta(1) & \text{при } n = 1, \\ 2T(n/2) + \Theta(n) & \text{при } n > 1. \end{cases}$$

Граничные условия – ещё один пример технических особенностей, которые обычно игнорируются. Поскольку время работы алгоритма с входными данными фиксированного размера выражается константой, то в рекуррентных соотношениях, описывающих время работы алгоритмов, для достаточно малых n обычно справедливо соотношение:

$T(n) = \Theta(1)$.

Поэтому для удобства граничные условия рекуррентных соотношений, как правило, опускаются и предполагается, что для малых **n** время работы алгоритма **T(n)** является константой. Например, рекуррентное соотношение (7.3) обычно записывается как

$$T(n) = 2T(n/2) + \Theta(n),$$

$$T(n) = \begin{cases} \Theta(1) & \text{при } n = 1, \\ 2T(n/2) + \Theta(n) & \text{при } n > 1. \end{cases}$$

без явного указания значений **T(n)** для малых **n**. Причина состоит в том, что хотя изменение значения T(1) приводит к изменению решения рекуррентного соотношения, это решение обычно изменяется не более, чем на постоянный множитель. А порядок роста остаётся неизменным.

Вывод: Формулируя и решая рекуррентные соотношения, часто опускаются граничные условия, а также тот факт, что аргументами неизвестных функций являются целые числа.

Метод подстановки

Метод подстановки, применяющийся для решения рекуррентных уравнений, состоит из двух этапов:

– делается догадка о виде решения;

– с помощью метода математической индукции определяются константы и доказывается, что решение правильное.

Метод подстановки можно применять для определения либо верхней, либо нижней границ рекуррентного соотношения. В качестве примера определим верхнюю границу рекуррентного соотношения

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \tag{7.5}$$

предполагаем, что решение имеет вид:

$$T(n) = O(n \log n).$$

Метод заключается в доказательстве того, что при подходящем выборе константы $c > 0$ выполняется неравенство

$$T(n) \leq cn \log n.$$

Предположим, что это неравенство справедливо для величины $\lfloor n/2 \rfloor$, то есть что выполняется соотношение $T(n/2) \leq c\lfloor n/2 \rfloor \log \lfloor n/2 \rfloor$.

После подстановки выражения $T(n/2) \leq c\lfloor n/2 \rfloor \log \lfloor n/2 \rfloor$ в рекуррентное соотношение

$T(n) = 2T(\lfloor n/2 \rfloor) + n$ получаем:

$$T(n) \leq 2(c\left\lfloor \frac{n}{2} \right\rfloor \log \left\lfloor \frac{n}{2} \right\rfloor) + n \leq cn \log \frac{n}{2} + n = cn \log n - cn \log 2 + n =$$

$$= cn \log n - cn + n \leq cn \log n,$$

где последнее неравенство выполняется при $c \geq 1$.

Теперь, согласно методу математической индукции, необходимо доказать. Что решение справедливо для граничных условий. В рекуррентном соотношении $T(n) = 2T(\lfloor n/2 \rfloor) + n$ необходимо доказать, что константу **c** можно выбрать достаточно большой для того, чтобы соотношение

$$T(n) \leq cn \log n$$

было справедливо и для граничных условий. Такое требование иногда приводит к проблемам.

Предположим, что $T(1) = 1$ единственное граничное условие рассматриваемого рекуррентного соотношения.

Для $n = 1$ соотношение

$$T(n) \leq cn \log n$$

даёт $T(1) \leq c * 1 * \log 1 = 0$, что противоречит условию $T(1) = 1$. Следовательно, данный базис индукции не выполняется.

Асимптотическую оценку достаточно доказать для всех **n**, начиная с некоторого. Подберем **c** так, чтобы оценка

$$T(n) \leq cn \log n$$

$$T(n) \leqslant 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \leqslant cn \log(n/2) + n = cn \log n - cn \log 2 + n =$$
$$= cn \log n - cn + n \leqslant cn \log n,$$

была верна при n = 2 и n = 3. Тогда для больших **n** можно рассуждать по индукции, и описанный случай $n = 1$ не встретится (поскольку $\lfloor n/2 \rfloor \geq 2$ при $n > 3$).

Пример

Рассмотрим рекуррентное соотношение $T(n) = 2T(\lfloor n/2 \rfloor + 15) + n$.

Дополнительное слагаемое 15 не может сильно повлиять на асимптотическое поведение решения. При достаточно больших **n** разность между $T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ и $T(\left\lfloor \frac{n}{2} \right\rfloor + 15)$ становится несущественной: оба эти числа приблизительно равны половине числа n. Следовательно, можно предложить, что $T(n) = O(n \log n)$, и проверить это с помощью метода подстановки.

Замена переменных

Часто несложная замена переменных позволяет преобразовать рекуррентное соотношение к привычному виду. Например, соотношение

$T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \log n.$

Кажется довольно сложным, однако, заменой переменных его легко упростить.

Сделав замену $n = 2^m$, $m = \log n$ получим

$T(2^m) = 2\,T(2^{m/2}) + m.$

Обозначив $T(2^m)$ через $S(m)$, приходим к соотношению:

$S(m)= 2\,S(m/2) + m,$

которое уже решали и его решение:

$S(m)= O (m \log m).$

Возвращаясь к $T(n)$ вместо $S(m)$, получим:

$T(n) = T(2^m) = S(m)= O (m \log m) = O (\log n \log\log n).$

$T(n) = 2T(\lfloor n/2 \rfloor) + n$

$T(n) = O(n \log n)$

Упражнения

1. Покажите, что Ȯ (log n) является решением рекуррентного соотношения

   $T(n) = T(\lceil n/2 \rceil) + 1.$

2. Покажите, что Ȯ (n log n) является решением рекуррентного соотношения

   $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n.$

3. Решите рекуррентное соотношение

$T(n) = 2T(\sqrt{n}) + 1$

с помощью замены переменных. Решение должно быть асимптотически точным. При решении игнорируйте тот факт, что значения являются целыми числами.

Метод итерации

Если не удаётся угадать решение? Тогда можно, интегрируя это соотношение (подставляя его само в себя), получить ряд, который можно оценивать тем или иным способом.

Пример:

Рассмотрим соотношение $T(n) = 3T(\lfloor n/4 \rfloor) + n$.

Подставим его в себя, получим:

$T(n) = n + 3T(\lfloor n/4 \rfloor) = n + 3\,(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)) = n + 3\,\lfloor n/4 \rfloor + 3(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor))) =$

$= n + 3\,\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor) = \ldots = n + 3\,\lfloor n/4 \rfloor + \ldots + 3^i T(\lfloor n/4^i \rfloor) = \ldots +$

$+3^{\log_4 n} T(1)$

$\lfloor n/4^i \rfloor = 1, i = \log_4 n.$

Заметив, что $\lfloor n/4^i \rfloor \leq n/4^i$, тогда

$T(n) \leq n + 3\,n/4 + 9\,n/16 + 27\,n/64 + \ldots + 3^{\log_4 n}\,O(1) \leq n \sum_{i=0}^{\infty} (\frac{3}{4})^i + \Theta(n^{\log_4 3}) =$

$= 4\,n + o(n) = O(n)$, заменили конечную сумму из не более чем $\log_4 n + 1$ членов на сумму бесконечного ряда. Так как $\sum_{i=0}^{\infty} (\frac{3}{4})^i = \frac{1}{1-\frac{3}{4}} = 4,\ \log_4 3 < 1.$

ЛЕКЦИЯ № 8

Тема 7: **Рекуррентные соотношения. Метод подстановки. Метод итераций. Общий рецепт**

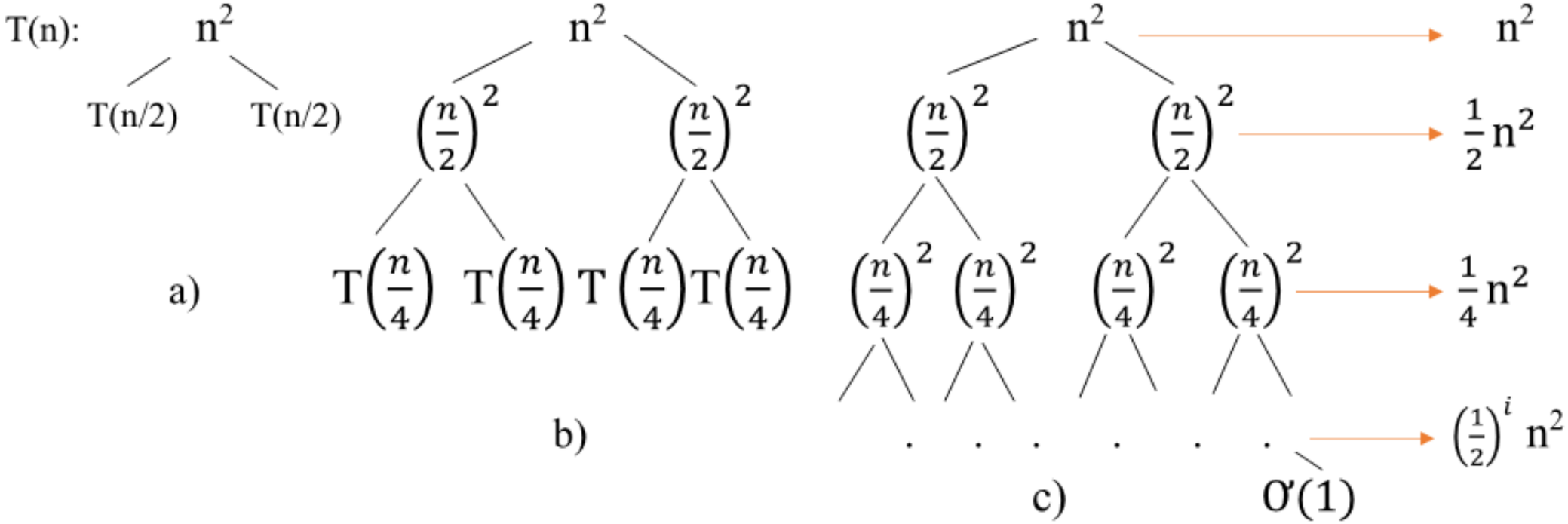
Деревья рекурсии

Процесс подстановки соотношения в себя можно изобразить в виде дерева рекурсии.

Для примера рассмотрим соотношение:

$T(n) = 2T(n/2) + n^2$.

Проиллюстрируем ниже процесс построения дерева рекурсии.

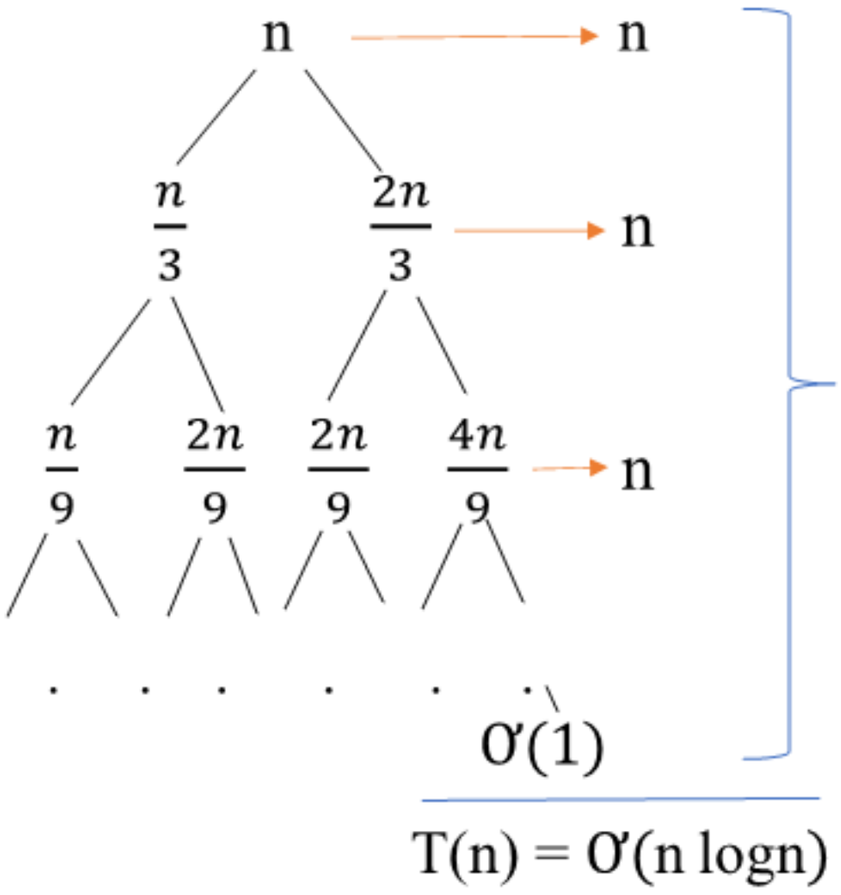


$T(n) = n^2 \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \ldots\right) = n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 2n^2 = O(n^2)$.

$T(n) = O(n^2)$

Пример 2:

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$



$$n \to \frac{2n}{3} \to \left(\frac{2}{3}\right)^2 n \to \cdots \to 1 \Longrightarrow \left(\frac{2}{3}\right)^k n = 1$$

$$k = \log_{\frac{3}{2}} n$$

$$T(n) = O(n \log n)$$

С помощью метода постановок убедимся, что сделанное выше предложение корректно. Покажем, что при подходящем выборе положительной константы c выполняется неравенство $T(n) \leq cn \log n$. Можно записать цепочку соотношений:

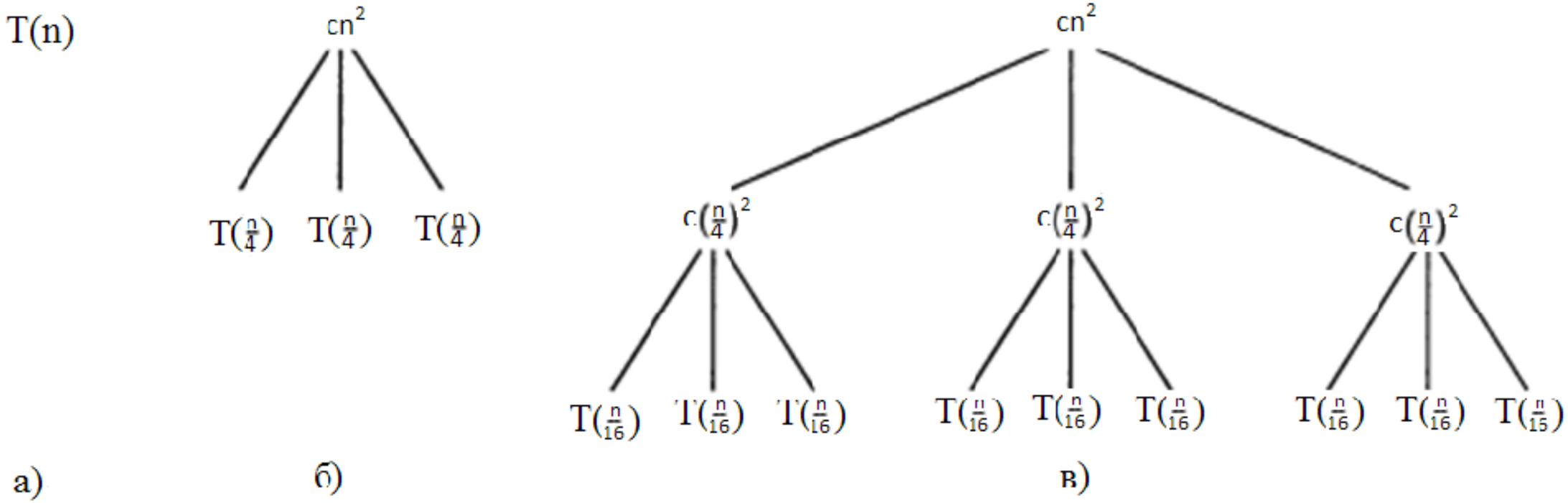
$$
\begin{aligned}
T(n) &\leqslant T(n/3) + T(2n/3) + n \leqslant \\
&\leqslant c(n/3)\log(n/3) + c(2n/3)\log(2n/3) + n = \\
&= (c(n/3)\log n - c(n/3)\log 3) + (c(2n/3)\log n - c(2n/3)\log(3/2)) + n = \\
&= cn\log n - c((n/3)\log 3 + (2n/3)\log(3/2)) + n = \\
&= cn\log n - c((n/3)\log 3 + (2n/3)\log 3 - (2n/3)\log 2) + n = \\
&= cn\log n - cn(\log 3 - 2/3) + n \leqslant \\
&\leqslant cn\log n,
\end{aligned}
$$

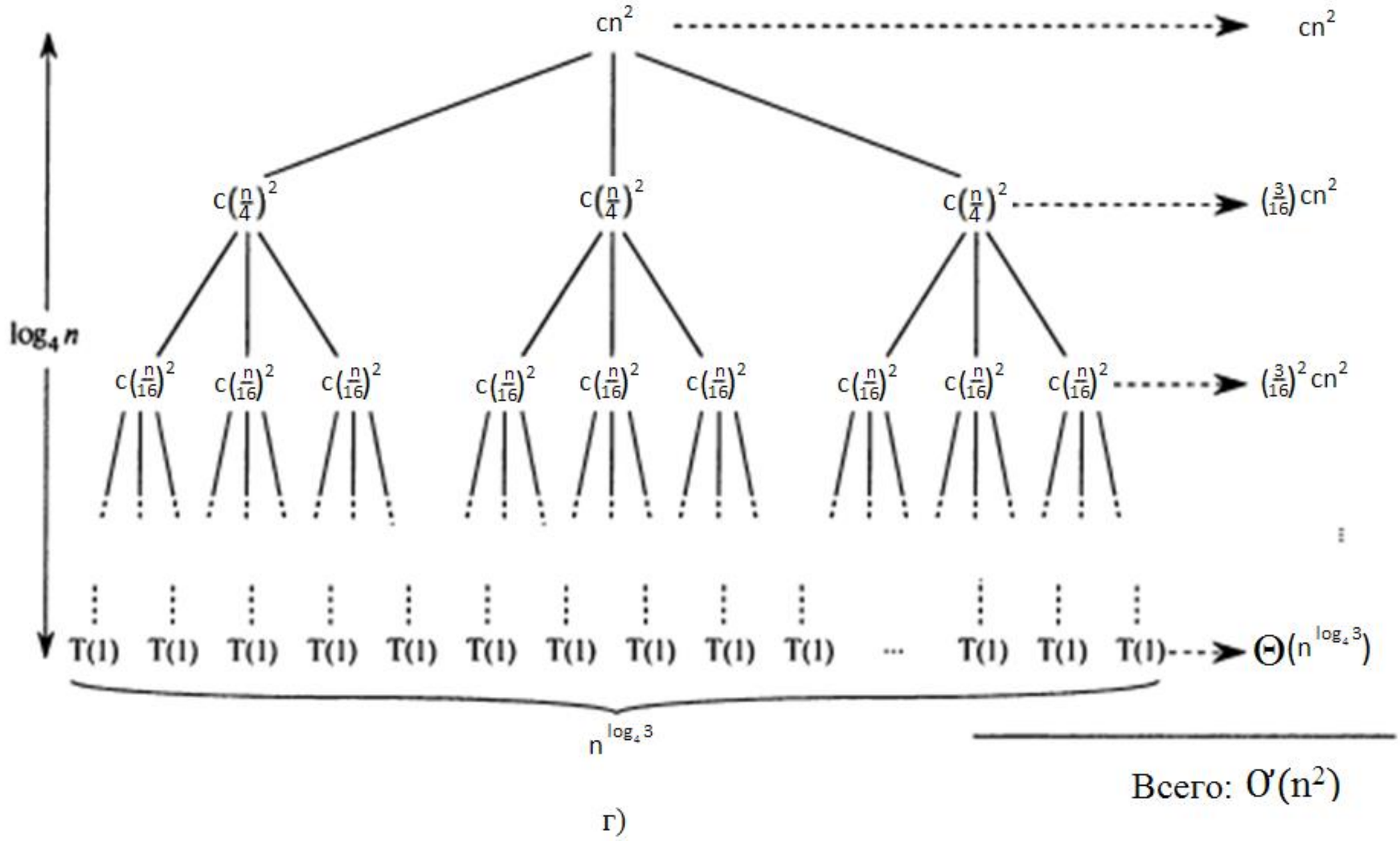
которые выполняются при $c \geq 1 / (\log 3 - 2/3)$. Таким образом, решение не делает более точный учет времени работы элементов, из которых состоит дерево рекурсии.

Пример 3:

$$T(n) = 3T(n/4) + cn^2.$$

На рисунке 7.1 проиллюстрирован процесс построения дерева рекурсии. Для удобства предположим, что n – степень четверки. В части а) показана функция T(n), которая затем все более подробно расписывается в частях б) – г) в виде эквивалентного дерева рекурсии, представляющего анализируемое рекуррентное соотношение. Член cn^2 в корне дерева представляет время верхнего уровня рекурсии, а три поддерева, берущих начало из корня, - времена выполнения подзадач размера n/4.





г)

Рисунок 7.1 – Построение дерева рекурсии для рекуррентного соотношения $T(n) = 3T(n/4) + cn^2$

Теперь суммируем времена работы всех уровней дерева и определяем время работы дерева целиком:

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) =$$

$$= \frac{1}{1-(3/16)} cn^2 + \Theta\left(n^{\log_4 3}\right) = \frac{16}{13} cn^2 + \Theta\left(n^{\log_4 3}\right) = O\left(n^2\right).$$

$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$

Таким образом, полное время работы всего дерева в основном определяется временем работы его корня.

Теперь, с помощью метода постановок проверим, что $T(n) = O(n^2)$.

Надо показать, что для некоторой константы $d > 0$ выполняется неравенство $T(n) \leq dn^2$.

Используя ту же константу $c > 0$, что и раньше, получаем:

$$T(n) \leqslant 3T(\lfloor n/4 \rfloor) + cn^2 \leqslant 3d\lfloor n/4 \rfloor^2 + cn^2 \leqslant$$
$$\leqslant 3d(n/4)^2 + cn^2 = \frac{3}{16}dn^2 + cn^2 \leqslant dn^2,$$

где последнее неравенство выполняется при $d \geq \frac{16}{13}c$.

Упражнения

1. Определите с помощью дерева рекурсии асимптотическую верхнюю границу рекуррентного соотношения

   $$T(n) = 3T(\lfloor n/2 \rfloor) + n.$$

   Проверьте ответ методом подстановок.

2. Обратившись к дереву рекурсии. Докажите. Что решение рекуррентного соотношения

   $$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn,$$

   где **c** -константа, ведет себя как $\Omega(n \log n)$.

3. Постройте дерево рекурсии для рекуррентного соотношения

   $$T(n) = 4T(\lfloor n/2 \rfloor) + cn,$$

   где **c** - константа, и найдите точную асимптотическую границу его решения. Проверьте её с помощью метода подстановок.

Основная теорема о рекуррентных соотношениях

Теорема: Пусть a ≥ 1 и b >1 – константы, f (n) – функция, T(n) определено при неотрицательных **n** формулой:

$$T(n) = a\,T(n / b) + f(n),$$

где под n/b понимается либо $\left\lceil \frac{n}{b} \right\rceil$, либо $\left\lfloor \frac{n}{b} \right\rfloor$.

Тогда

1) Если $f(n) = O(n^{\log_b a - \varepsilon})$ (зазор n^{ε}) для некоторого $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$;

2) Если $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \log n)$;

3) Если $f(n) = \Omega(n^{\log_b a + \varepsilon})$ для некоторого $\varepsilon > 0$, и если $a\,f(n / b) \le c\,f(n)$ (условие регулярности) для некоторой константы $c < 1$ и достаточно больших n, то $T(n) = \Theta(f(n))$;

4) Если $f(n) = \Theta(n^{\log_b a} \log^k n)$, то $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$; $k \ge 0$.

Применение основной теоремы

Рассмотрим соотношение:

1) $T(n) = 9\ T(n/3) + n,$

Если $f(n) - O(n^{\log_b a - \varepsilon})$ (зазор n^{ε}) для некоторого $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$;

$a = 9,\ b = 3,\ f(n) = n,$

$$n^{\log_b a} = n^{\log_3 9} = n^2 = \Theta(n^2);\ \frac{n^{\log_b a}}{f(n)} = \frac{n^2}{n} = n = n^{\varepsilon},\ \varepsilon = 1.$$

Поскольку $f(n) = O(n^{\log_b a - \varepsilon})$ для $\varepsilon = 1 > 0$, применяем 1- ое утверждение

теоремы и заключаем, что

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$$

2)T(n) = T(2n / 3) + 1,

a = 1, b = 3/2, f (n) = 1,

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1.$

Подходит 2 – ой случай, поскольку

f (n) = Θ ($n^{\log_b a}$) = Θ (1),

получаем, что

T(n) = Θ ($n^{\log_b a}$ logn) = Θ (f (n)logn) = Θ (logn);

Если f(n) = Θ ($n^{\log_b a}$), то

T(n) = Θ ($n^{\log_b a}$ $logn$);

3) $T(n) = 3\ T(n / 4) + n \log n$,

Если $f(n) = \Omega\ (n^{\log_b a+\varepsilon})$ для $\varepsilon > 0$, и если $af(n/b) \le c\ f(n)$, $c < 1$ и достаточно больших n, то $T(n) = \Theta\ (f(n))$;

$a = 3$, $b=4$, $f\ (n) = n \log n$,

$n^{\log_b a} = n^{\log_4 3} = O\ (n^{0,793})$, $\dfrac{n^{\log_b a}}{f\ (n)} = \dfrac{n^{0,793}}{n \log n} = \dfrac{1}{n^{0,207} \log n} = \dfrac{1}{n^{\varepsilon} \log n}$, $\varepsilon = 0,207.$

$f\ (n) = \Omega\ (n^{\log_b a+\varepsilon})$ (зазор с $\varepsilon \approx 0,2 > 0$).

Необходимо проверить условие регулярности.

Для достаточно большого n имеем

$a\ f(n/b) = 3(n/4) \log\left(\frac{n}{4}\right) = 3/4\ n \log n - 3/4\ n \log 4 \le 3/4\ n \log n = c\ f\ (n)$,

$c = 3/4 < 1$.

Тем самым по 3 – ему утверждению теоремы

$T(n) = \Theta\ (f\ (n)) = \Theta\ (n \log n)$;

4)T (n) = 2 T (n / 2) + n logn,

Если f(n) = Θ $(n^{\log_b a} log^k n)$, то

T(n) = Θ $(n^{\log_b a} log^{k+1} n)$; k ≥ 0.

a = 2, b = 2, f (n) = n logn,

$n^{\log_b a} = n^{\log_2 2} = n.$

Видно, что f (n) = n logn асимптотически больше, чем $n^{\log_b a}$, но зазор недостаточен:

отношение f (n) / $n^{\log_b a}$ = (n logn) / n = logn

не оценивается снизу величиной n^{ε} ни для какого $\varepsilon > 0$.

Теорему применить не удается, так как соотношение попадает в промежуток между

случаями 2 и 3;

для него можно получить ответ по формуле (утверждение 4):

T(n) = Θ (f (n) logn) = Θ (n $\log^2$n).

Упражнения

1. С помощью основной теоремы найдите точные асимптотические границы следующих рекуррентных соотношений.

   a) $T(n) = 4T(n/2) + n$,

   b) $T(n) = 4T(n/2) + n^2$,

   c) $T(n) = 4T(n/2) + n^3$

2. С помощью основной теоремы найдите точные асимптотические границы следующих рекуррентных соотношений.

   a) $T(n) = 3\,T(n/2) + n \log n$,

   b) $T(n) = 5\,T(n/5) + n \log^2 n$,

   c) $T(n) = 4\,T(n/2) + n^2\sqrt{n}$,

   d) $T(n) = 3\,T(n/3 + 5) + n/2$,

   e) $T(n) = 2\,T(n/2) + n/\log n$.

3. Можно ли применить основной метод к рекуррентному соотношению

$$T(n) = 4T(n/2) + n^2 \log n,$$

Обоснуйте ответ.

4. Найдите асимптотическую верхнюю границу решения этого рекуррентного соотношения

$$T(n) = 4T(n/2) + n^3 \log n.$$

Решение линейных однородных рекуррентных соотношений

$a_0\, t_n + a_1\, t_{n-1} + \ldots + a_k\, t_{n-k} = 0.$

Введём замену:

$t_n = x^n,$

$a_0\, x^n + a_1\, x^{n-1} + \ldots + a_k\, x^{n-k} = 0 \quad : x^{n-k} \neq 0,$

$a_0\, x^k + a_1\, x^{k-1} + \ldots + a_k = 0.$

Имеем k различных корней $r_1, r_2, \ldots, r_k$.

$t_n = \sum_{i=1}^{k} C_i r_i^n,$

где C_i – const.

При s равных корней:

$$t_n = [C_1 + C_2 n + \ldots + C_s n^{s-1}] r_1^n + C_{s+1} r_{s+1}^n + \ldots + C_k r_k^n .$$

Пример 1. Числа Фибоначчи:

$t(n) = t(n-1) + t(n-2)$ или $t_n = t_{n-1} + t_{n-2}$, $t_n - t_{n-1} - t_{n-2} = 0.$

Начальные условия: $t_0 = 0,\ t_1 = 1,\ n \geq 2.$

$t_n = x^n,$

$x^n - x^{n-1} - x^{n-2} = 0 \quad : x^{n-2} \neq 0,$

$x^2 - x - 1 = 0,$

$r_{1,2} = \frac{1 \pm \sqrt{1+4}}{2} = \frac{1 \pm \sqrt{5}}{2};$

$ax^2 + bx + c = 0,$

$x_{1,2} = \frac{-b \pm \sqrt{b^2-4ac}}{2};$

$t_n = C_1 r_1^n + C_2 r_2^n$

$r_1 = \frac{1+\sqrt{5}}{2};\ r_2 = \frac{1-\sqrt{5}}{2},$ $\quad t_n = C_1 (\frac{1+\sqrt{5}}{2})^n + C_2 (\frac{1-\sqrt{5}}{2})^n,$

$$t_n = C_1\left(\frac{1+\sqrt{5}}{2}\right)^n + C_2\left(\frac{1-\sqrt{5}}{2}\right)^n,$$

так как известны начальные условия, то найдём коэффициенты C_1 и C_2:

$$\left.\begin{array}{ll} t_0 = C_1 r_1^0 + C_2 r_2^0 = C_1 + C_2 = 0 & C_2 = -C_1 \\ t_1 = C_1 r_1^1 + C_2 r_2^1 = 1 & C_1\left(\frac{1+\sqrt{5}}{2}\right) + C_2\left(\frac{1-\sqrt{5}}{2}\right) = 1 \end{array}\right\}$$

$$C_1\left(\frac{1+\sqrt{5}}{2}\right) - C_1\left(\frac{1-\sqrt{5}}{2}\right) = 1, \quad 2\frac{\sqrt{5}}{2}C_1 = 1,\ C_1 = \frac{1}{\sqrt{5}},\ C_2 = -\frac{1}{\sqrt{5}}.$$

$$t_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right].$$

Пример 2.

Решить рекуррентное соотношение:

$t(n + 4) = 5t(n + 3) – 6\ t(n + 2) – 4\ t(n + 1) + 8\ t(n).$

$t(n) = x^n,$

$x^{n+4} – 5x^{n+3} + 6x^{n+2} + 4x^{n+1} – 8x^n = 0 \quad : x^n \neq 0,$

$x^4 – 5x^3 + 6x^2 + 4x – 8 = 0, \qquad (x + 1)\ (x – 2)^3 = 0$

решая характеристическое уравнение, получим корни:

$r_1 = 2, r_2 = 2, r_3 = 2, r_4 = -1.$

$t(n) = 2^n[C_1 + C_2 n + C_3 n^2] + C_4(-1)^n = O(2^n n^2).$

Пример 3. Решить рекуррентное соотношение:

$t_n = 5\ t_{n-1} - 8\ t_{n-2} + 4\ t_{n-3}, \qquad n \geq 3.$

Н. У.: $t_0 = 0, t_1 = 1, t_2 = 2,$

$x^n - 5x^{n-1} + 8x^{n-2} - 4x^{n-3} = 0 \quad : x^{n-3} \neq 0,$

$x^3 - 5x^2 + 8x + 4 = 0, \qquad (x - 1)(x - 2)^2 = 0,$

$r_1 = 1, r_2 = 2, r_3 = 2.$

$t_n = C_1 1^n + [C_2 + C_3 n]\ 2^n,$

$$\left.\begin{aligned} t_0 &= C_1 + C_2 + C_3 * 0 = 0, \\ t_1 &= C_1 + C_2 + 2C_3 = 1, \\ t_2 &= C_1 1^2 + [C_2 + C_3 2]\ 2^2 = 2 \end{aligned}\right\} \quad C_1 = -2, C_2 = 2, C_3 = -\frac{1}{2}.$$

$t_n = -2 * 1^n + [2 + 2^{-1} n]\ 2^n =$

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

$= O(2^n).$

Решение линейных неоднородных рекуррентных соотношений

$a_0\, t_n + a_1\, t_{n-1} + \ldots + a_k\, t_{n-k} = v_1^n P_1(n) + v_2^n P_2(n) + \ldots + v_l^n P_l(n),$

где $P_l(n)$ – полином степени **d**.

Делаем замену $t_n = x^n$,

$$(a_0\, x^k + a_1\, x^{k-1} + \ldots + a_k)(x - v_1)^{d_1+1}(x - v_2)^{d_2+1} \ldots (x - v_l)^{d_l+1} = 0.$$

Количество корней:

$$k + d_1 + 1 + d_2 + 1 + \cdots + d_l + 1.$$

Если имеем k разных корней $r_1, r_2, \ldots, r_k$.

$t_n = \sum_{i=1}^{k} C_i r_i^n$, где C_i – const;

s равных корней:

$$t_n = [C_1 + C_2 n + \ldots + C_s n^{s-1}] r_1^n + C_{s+1} r_{s+1}^n + \ldots + C_k r_k^n.$$

Пример 1: $(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x - v_1)^{d_1+1}(x - v_2)^{d_2+1} \ldots (x - v_l)^{d_l+1} = 0$

$t_n - 2t_{n-1} = 3^n,$

$t_n - 2t_{n-1} = 3^n \cdot 1,$

$t_n - 2t_{n-1} = 3^n \cdot n^0,\ n \geq 1,\ v = 3,\ d = 0.$

Делаем замену $t_n = x^n,$

$x^n - 2x^{n-1} = 0 \quad : x^{n-1} \neq 0,$

$(x - 2) \cdot (x - 3)^{0+1} = 0,$

$r_1 = 2,\ r_2 = 3.$

$t_n = C_1 r_1^n + C_2 r_2^n,$ $t_n = C_1 2^n + C_2 3^n = O(3^n).$

Пример 2: $(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x - v_1)^{d_1+1}(x - v_2)^{d_2+1} \ldots (x - v_l)^{d_l+1} = 0$

$t_n - 2t_{n-1} = 3^n n,$

$t_n - 2t_{n-1} = 3^n \cdot n^1, n \geq 1, v = 3, d = 1.$

Делаем замену $t_n = x^n,$

$x^n - 2x^{n-1} = 0 \quad : x^{n-1} \neq 0,$

$(x - 2) \cdot (x - 3)^{1+1} = 0,$

$r_1 = 2, r_{2,3} = 3.$

2 равных корня для $r_2 = r_3 = 3$:

$t_n = C_1 r_1^n + [C_2 + C_3 n] r_2^n,$ $t_n = C_1 2^n + C_2 3^n + C_3 3^n n = O(3^n n).$

Пример 3: $(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x - v_1)^{d_1+1}(x - v_2)^{d_2+1} \ldots (x - v_l)^{d_l+1} = 0$

$t_n - 2t_{n-1} = 2^n n^2 + 3^n n,$

$n \geq 1, v_1 = 2, d_1 = 2, v_2 = 3, d_2 = 1.$

Делаем замену $t_n = x^n$,

$x^n - 2x^{n-1} = 0 \quad : x^{n-1} \neq 0,$

$(x - 2) \cdot (x - 2)^{2+1} \cdot (x - 3)^{1+1} = 0,$

$r_{1,2,3,4} = 2, r_{5,6} = 3.$

равные корни $r_1 = r_2 = r_3 = r_4 = 2, r_5 = r_6 = 3$:

$t_n = [C_1 + C_2 n + C_3 n^2 + C_4 n^3]r_1^n + [C_5 + C_6 n]r_5^n,$

$t_n = [C_1 + C_2 n + C_3 n^2 + C_4 n^3]2^n + [C_5 + C_6 n]3^n = O(n^3 2^n).$

http://grafikus.ru/

$$c_1 n3^n \leq 2^n n^3 \leq c_2 n3^n$$

$$c_1 \leq \frac{2^n n^2}{3^n} \leq c_2,$$

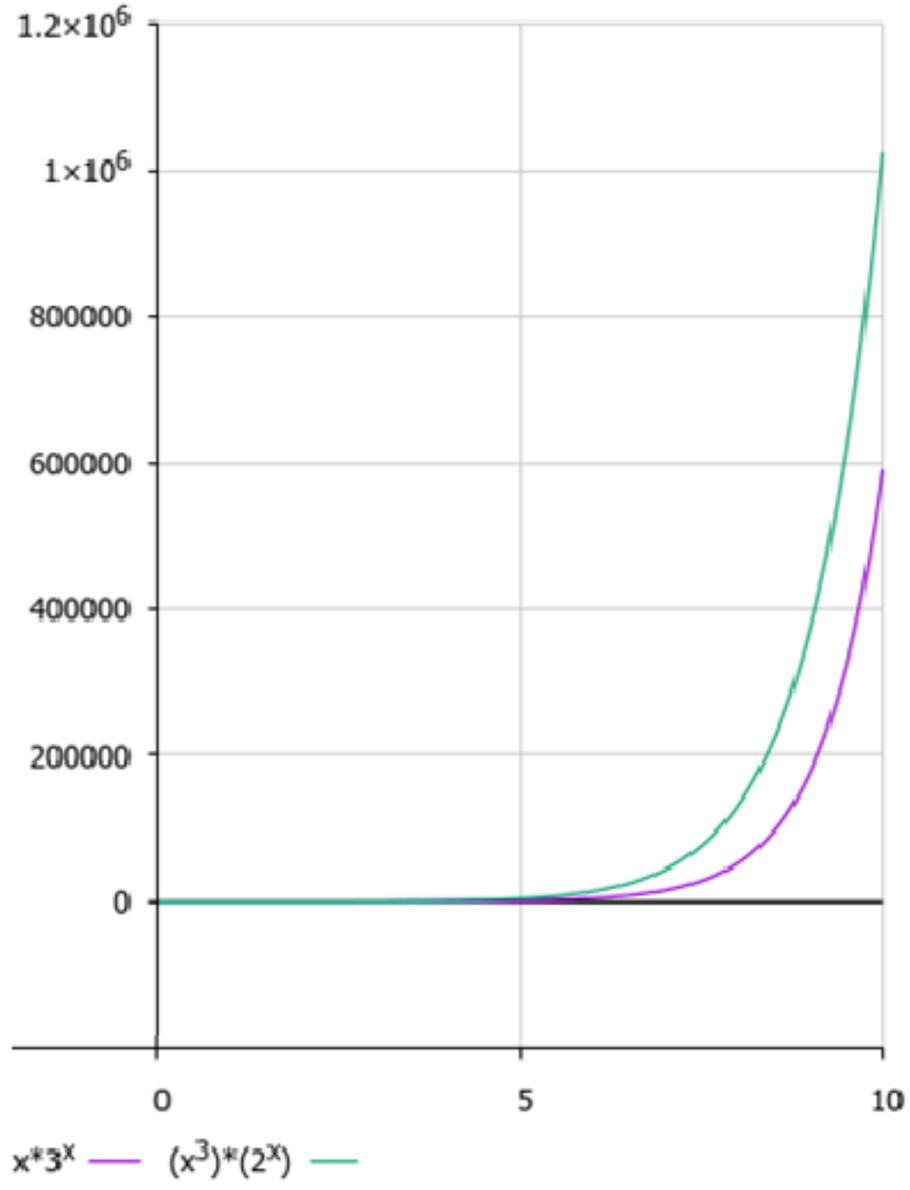
$$n_0 = 1, c_1 = \frac{2}{3}$$

$c_2 - ?$

$$n^3 2^n = \Omega(n3^n),$$

$$n3^n = O(n^3 2^n).$$

$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ при $|x| < 1$



Замена переменных $(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x - v_1)^{d_1+1}(x - v_2)^{d_2+1} \ldots (x - v_l)^{d_l+1} = 0$

T(n) = a T (n / b) + f (n),

$T(n) = 4\ T(\frac{n}{2}) + n,$

$a = 4, b = 2, n = b^k, k = \log_b n, k = \log_2 n.$

Делаем замену

$T(n) = T(2^k),$

$T(2^k) = 4\ T(\frac{2^k}{2}) + 2^k,$

$T(2^k) = 4\ T(2^{k-1}) + 2^k,$

Делаем замену

$T(2^k) = t_k,$

$t_k = 4t_{k-1} + 2^k,$

$t_k - 4t_{k-1} = 2^k,\ k \geq 1,\ v = 2,\ d = 0.$

$(x - 4)\ (x - 2)^{0+1} = 0,$

$r_1 = 4, r_2 = 2.$

$t_k = C_1 r_1^k + C_2 r_2^k;$

$t_k = C_1 4^k + C_2 2^k;$

$T(n) = C_1 4^{\log_2 n} + C_2 2^{\log_2 n};$

$T(n) = C_1 n^{\log_2 4} + C_2 n^{\log_2 2};$

$T(n) = C_1 n^2 + C_2 n = O(n^2);$

$T(n) = O(n^2).$

$T(n) = 4T(n/2) + n^2\ logn$

$a = 4,\ b = 2,\ n = b^k,\ k = \log_2 n.$

Делаем замену

$T(n) = T(2^k),$

$T(2^k) = 4\ T(\frac{2^k}{2}) + 2^{2k}\,k,$

$T(2^k) = 4\ T(2^{k-1}) + 4^k\,k,$

Делаем замену

$T(2^k) = t_k,$

$t_k = 4t_{k-1} + 4^k\,k,$

$t_k - 4t_{k-1} = 4^k\,k^1, \qquad k \geq 1,\ v = 4,\ d = 1.$

$(x - 4)\ (x - 4)^{1+1} = 0,$

$r_{1,2,3} = 4.$

$t_k = [C_1 + C_2k + C_3k^2]r_1^k;$

$t_k = [C_1 + C_2k + C_3k^2]4^k;$

$T(n) = [C_1 + C_2logn + C_3log^2n]4^{logn};$

$T(n) = [C_1 + C_2logn + C_3log^2n]n^2;$

$T(n) = O\left(n^2log^2n\right).$

Упражнения

Решить рекурентные соотношения:

a) $T(n) = T(n / 2) + T(n / 4) + T(n / 8) + n$,

b) $T(n) = T(n - 1) + 1 / n$,

c) $T(n) = 4T(n/2) + n$,

d) $T(n) = 4T(n/2) + n^2$,

e) $T(n) = 4T(n/2) + n^3$,

f) $T(n) = 4T(n/2) + n^2\ logn$,

ЛЕКЦИЯ № 9

Прямая адресация. Хеш-таблицы. Хеш-функции. Открытая адресация. Способы вычисления последовательности испробованных мест при открытой адресации

Когда нужны динамические множества, поддерживающие только словарные операции (добавление, поиск и удаление элемента), часто применяют так называемое **хеширование**, соответствующая структура данных называется «**хеш-таблица**» (или «таблица расстановки»). В худшем случае поиск в хеш-таблице может занимать столько же времени, сколько поиск в списке ($\Theta(n)$), но на практике хеширование весьма эффективно. При выполнении некоторых естественных условий математическое ожидание времени поиска элемента в хеш-таблице есть $O(1)$.

Хеш-таблицу можно рассматривать как обобщение обычного массива. Если достаточно памяти для массива, число элементов которого равно числу всех возможных ключей, для каждого возможного ключа можно отвести ячейку в этом массиве и тем самым иметь возможность добраться до любой записи за время $O(1)$. Однако, если реальное количество записей значительно меньше, чем количество возможных ключей, то эффективнее применить **хеширование**: вычислять позицию записи в массиве, исходя из ключа.

Хеширование — эффективный способ представления данных, позволяющий быстро выполнять основные словарные операции (среднее время Ȯ (1) при некоторых предположениях).

Способы хеширования:

– Прямая адресация.

– Закрытая адресация (хеш-таблицы).

– Открытая адресация.

Прямая адресация

Применима, если количество возможных ключей невелико.

Пусть возможными ключами являются числа из множества **U = {0, 1, 2, . . ., m-1}** (число **m** не очень велико). Предположим также, что ключи всех элементов различны.

Для хранения множества **U** пользуемся массивом **T [0 . . m-1]**, называемым **таблицей с прямой адресацией** (direct-address table).

Каждая позиция или ячейка (position, slot) соответствует определенному ключу из множества U (рисунок 9.1).

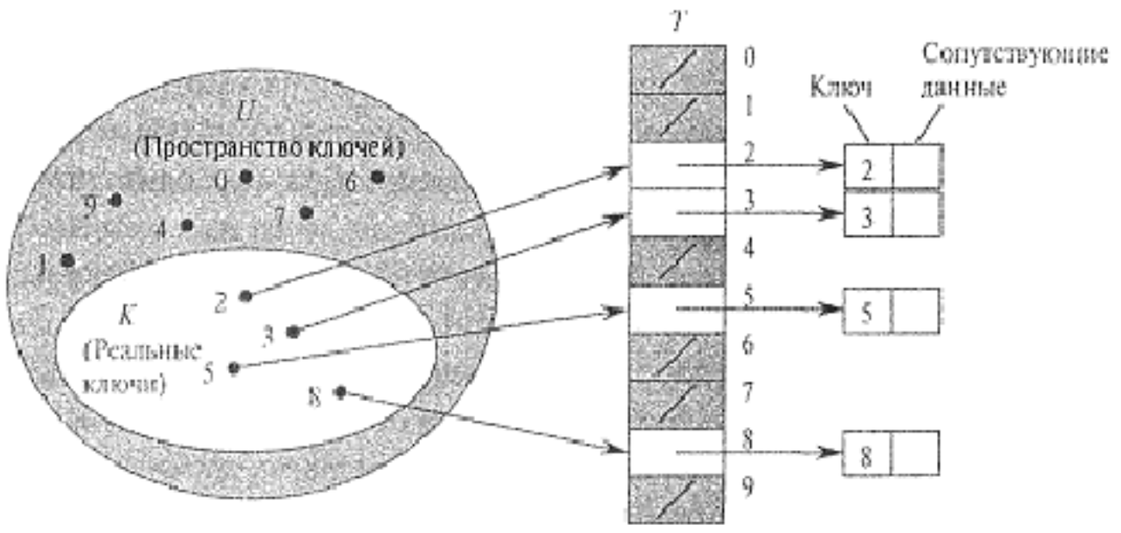


Рисунок 9.1 – Реализация динамического множества с помощью таблицы T с прямой адресацией

T[k] – место, предназначенное для записи указателя на элемент с ключом **k**; если элемента с ключом **k** в таблице нет, то **T[k] = NIL**.

Реализация словарных операций тривиальна:

DIRECT - ADDRESS – SEARCH (T, k)

Return T[k]

DIRECT - ADDRESS – INSERT (T, x)

T[key[x]] ← x

DIRECT - ADDRESS – DELETE (T, k)

T[key[x]] ← NIL

Каждая из этих операций требует времени O̓ (1).

Иногда можно сэкономить место, записывая в таблицу T не указатели на элементы множества, а сами эти элементы. Можно обойтись и без отдельного поля «ключ»: ключом служит индекс в массиве.

Закрытая адресация

Хеш – таблицы (Hash-table)

Если количество записей в таблице существенно меньше, чем количество всевозможных ключей, то хеш-таблица занимает гораздо меньше места, чем таблица с прямой адресацией. Именно, хеш-таблица требует памяти объёмом **Θ(|k|)**, где **k** – множество записей.

Элемент с ключом **k** при хешировании записывается в позицию номер **h(k)** в хеш-таблице, где **h: U = {0, 1, 2, . . ., m-1}** – некоторая функция, называемая **хеш-функцией**.

Число **h(k)** называется хеш-значением ключа **k**. Идея хеширования показана на рисунке 9.2, пользуясь массивом длины **m**, а не **|U|**, экономим память.

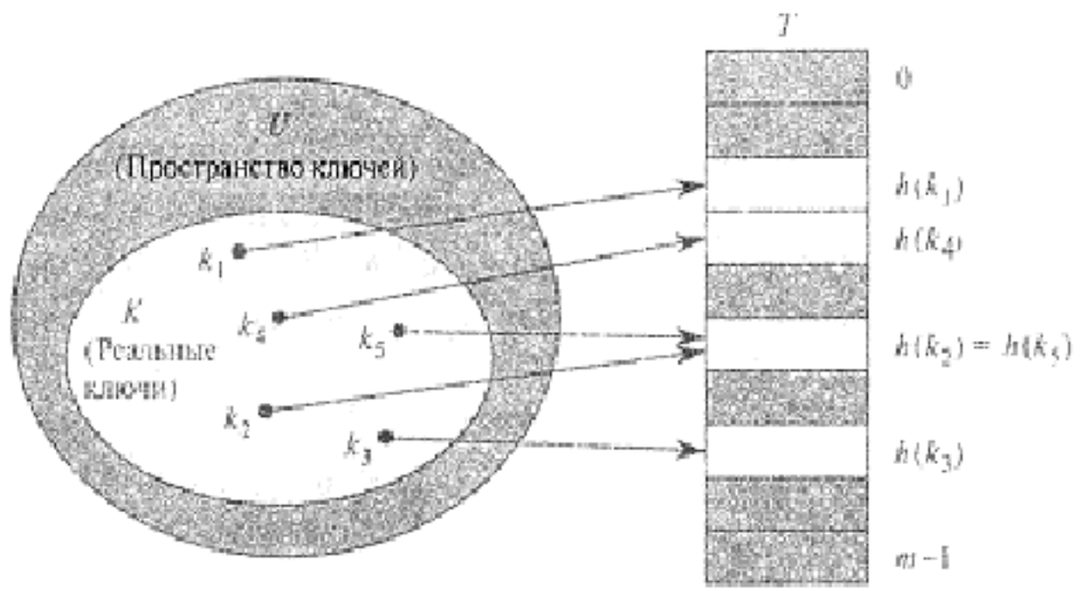


Рисунок 9.2 – Использование хеш-функции для отображения ключей в позиции хеш-таблицы. Хеш-значения ключей k_2 и k_5 совпадают- имеет место коллизия

Разрешение коллизий с помощью цепочек

Технология сцепления элементов состоит в том, что элементы множества, которым соответствует одно и то же хеш-значение связываются в **цепочку – список** (рисунок 9.3). В позиции номер **j** хранится указатель на голову списка тех элементов, у которых хеш-значение ключа равно **j**, если таких элементов в множестве нет, в позиции **j** записан NIL.

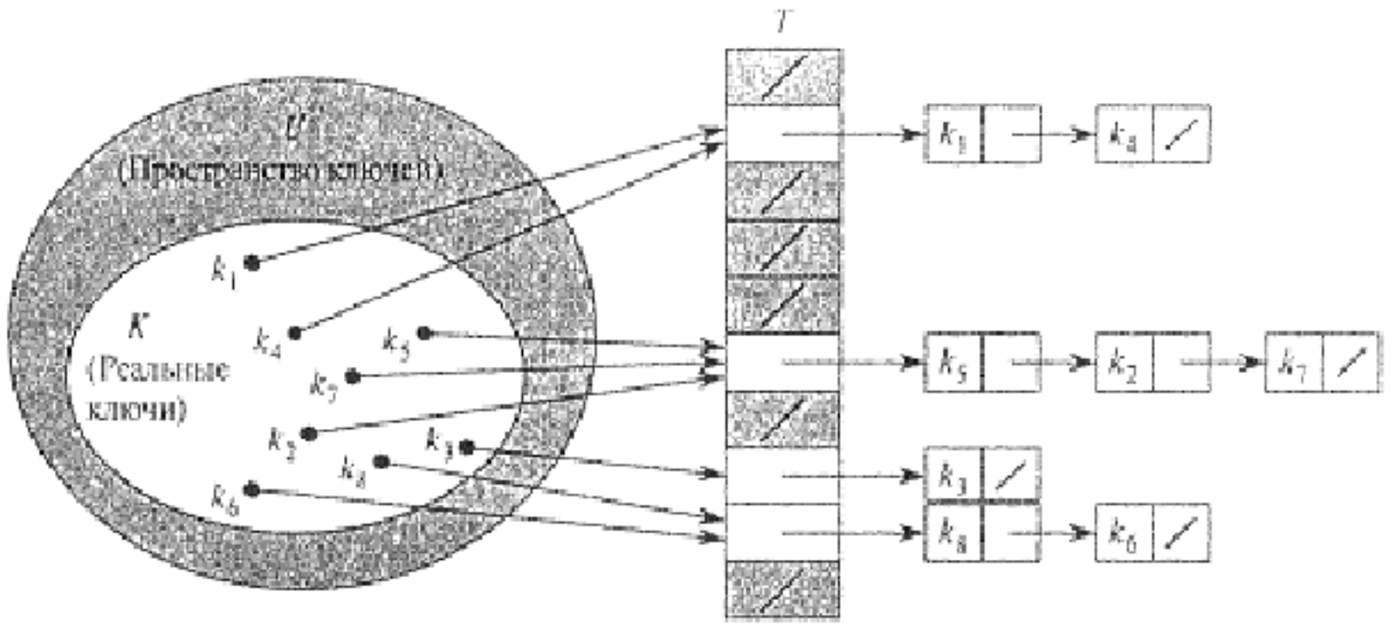


Рисунок 9.3 – Разрешение коллизии с помощью цепочек. В позиции T[j] хранится указатель на список элементов с хеш-значением j. Например, $h(k_1) = h(k_4)$ и $h(k_5) = h(k_2) = h(k_7)$

Операции добавления, поиска и удаления реализуются следующим образом:

CHAINED - HASH - INSERT (T, x)

Добавить x в голову списка T[h(key[x])]

CHAINED - HASH - SEARCH (T, k)

Найти элемент с ключом k в списке T[h(k)]

CHAINED - HASH - DELETE (T, x)

удалить x из списка T[h(key[x])]

Операция добавления работает в худшем случае за время **O (1)**. Максимальное время работы операции поиска пропорционально длине списка.

Удаление элемента можно провести за время **O (1)** – при условии, что списки двусторонне связаны (если списки связаны односторонне, то для удаления элемента **x** надо предварительно найти его предшественника, для чего необходим поиск по списку; в таком случае стоимость удаления и поиска примерно одинаковы).

Хеш-функции

Способы построения хеш-функции:

– деление с остатком;

– умножение;

– универсальное хеширование.

Хорошая хеш-функция должна (приближенно) удовлетворять предположениям равномерного хеширования: для очередного ключа все m хеш-значений должны быть равновероятны.

Обычно предполагают, что область определения хеш-функции – множество целых неотрицательных чисел.

Деление с остатком

Построение хеш-функции **методом деления с остатком** состоит в том, что ключу **k** ставится в соответствие остаток от деления **k** на **m**, где **m** – число возможных хеш-значений.

$$h(k) = k \bmod m.$$

Например, если размер хеш-таблицы равен $m = 12$ и ключ $k = 100$, то $h(100) = 100 \bmod 12 = 4$.

При этом некоторых значениях **m** стоит избегать:

– степень двойки;

– степень десятки (поскольку уже часть цифр ключа полностью определяет хеш-значение);

– $m = 2^p - 1$ (поскольку при этом одинаковые хеш-значения имеют ключи, отличающиеся лишь перестановкой «2^p-ичных цифр»).

Хорошие результаты обычно получаются, если выбрать в качестве **m** *простое число*, далеко отстоящее от степеней двойки. Пусть, например, надо поместить 2000 записей в хеш-таблицу с цепочками, причем нас не пугает возможный перебор 3-х вариантов при поиске отсутствующего в таблице элемента. Воспользуемся методом деления с остатком при длине хеш-таблицы $m = 701$. Число 701 простое, $701 \approx 2000/3$, и до степеней двойки от числа 701 тоже далеко. Таким образом, можно выбрать хеш-функцию вида: $h(k) = k \bmod 701$.

Умножение

Пусть количество хеш-функций равно **m**. Зафиксируем константу A в интервале $0 < A < 1$, и положим

$h(k) = \lfloor m(kA \bmod 1) \rfloor$,

где $kA \bmod 1$ – дробная часть kA.

Достоинство метода умножения в том, что качество хеш-функции мало зависит от выбора **m**.

Обычно в качестве **m** выбирают степень двойки, поскольку в большинстве компьютеров умножение на такое **m** реализуется как сдвиг слова.

Кнут Д. [Искусство программирования для ЭВМ. Т.3. Сортировка и поиск. - М.: Мир, 1978]. Обсуждает выбор константы A и приходит к выводу, что значение

$A \cong (\sqrt{5} - 1) / 2 = 0{,}6180339887\ldots$

является довольно удачным.

Например, если k = 123456, m =10000 A = 0,6180339887, то

$$h(k) = \lfloor 10000(123456 * 0{,}61803 \ldots \bmod 1) \rfloor = \lfloor 10000(76300{,}0041151 \ldots \bmod 1) \rfloor =$$

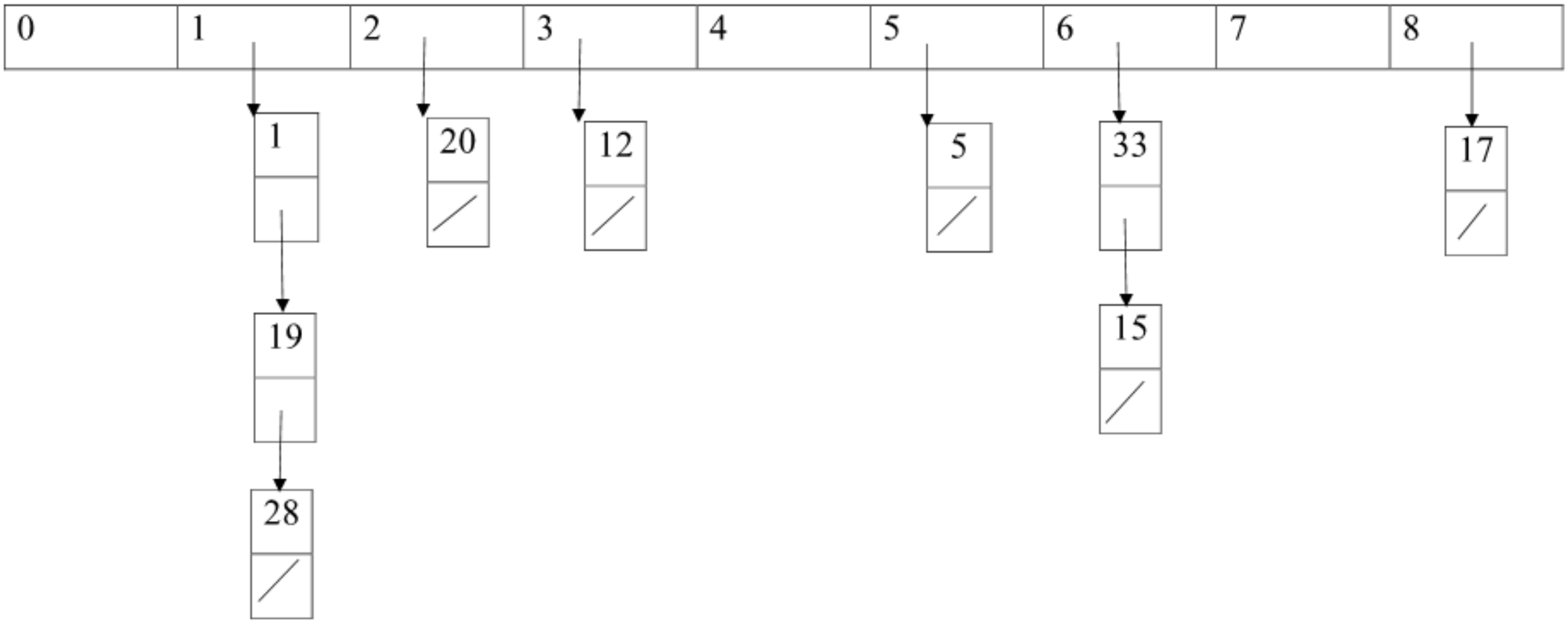
$$= \lfloor 10000 * 0{,}0041151 \ldots \rfloor = \lfloor 41{,}151 \ldots \rfloor = 41.$$

Универсальное хеширование

Выбор хеш – функции случайным образом, не зависящим от того, какие именно данные хешируем. Такой подход называется универсальным хешированием.

Упражнения

1. Как будет выглядеть хеш-таблица с цепочками после того, как в неё последовательно поместим элементы с ключами 5, 28, 19, 15, 20, 33, 12, 17, 1 (в указанном порядке)? Число позиций в таблице рано 9, хеш-функция имеет вид $h(k) = k \bmod 9$.



h(1) = 1 mod 9 = 1; h(17) = 17 mod 9 = 8; h(12) = 12 mod 9 = 3;

h(33) = 33 mod 9 = 6; h(20) = 20 mod 9 = 2; h(15) = 15 mod 9 = 6;

h(19) = 19 mod 9 = 1; h(28) = 28 mod 9 = 1; h(5) = 5 mod 9 = 5.

2. Как будет выглядеть хеш-таблица с цепочками после того, как в неё последовательно поместим элементы с ключами 78, 69, 72, 50, 98, 14, 85, 59, (в указанном порядке)? Число позиций в таблице рано 13, хеш-функция имеет вид $h(k) = k \bmod 13$.
3. Пусть размер хеш-таблицы равен m =10000. А хеш-функция имеет вид $h(k) = \lfloor m(kA \bmod 1) \rfloor$, где $A \cong (\sqrt{5} - 1) / 2$. В какие позиции попадут ключи 61, 62, 63, 64, 65?
4. Пусть размер хеш-таблицы равен m =200. А хеш-функция имеет вид $h(k) = \lfloor m(kA \bmod 1) \rfloor$, где $A \cong (\sqrt{5} - 1) / 2$. В какие позиции попадут ключи 31, 32, 33, 34, 35?
5. Пусть размер хеш-таблицы равен m =13. А хеш-функция имеет вид $h(k) = \lfloor m(kA \bmod 1) \rfloor$, где $A \cong (\sqrt{5} - 1) / 2$. В какие позиции попадут ключи 78, 69, 72, 50, 98, 14, 85, 59?
6. Как будет выглядеть хеш-таблица с цепочками после того, как в неё последовательно поместим элементы с ключами 5, 28, 19, 15, 20, 33, 12, 17, 1 (в указанном порядке)? Число позиций в таблице рано 9, хеш-функция имеет вид $h(k) = k \bmod 9$.
7. Пусть размер хеш-таблицы равен m = 9. А хеш-функция имеет вид $h(k) = \lfloor m(kA \bmod 1) \rfloor$, где $A \cong (\sqrt{5} - 1) / 2$. В какие позиции попадут ключи 5, 28, 19, 15, 20, 33, 12, 17, 1?

Открытая адресация

Здесь никаких списков нет. А все записи хранятся в самой хеш-таблице: каждая ячейка таблицы содержит либо элемент динамического множества, либо NIL.

Поиск заключается в том, что определенным образом просматриваем элементы таблицы, пока не найдем то, что ищем или не удостоверимся, что элемента с таким ключом в таблице нет. Тем самым число хранимых элементов не может быть больше размера таблицы: коэффициент заполнения не больше 1.

За счёт экономии памяти на указателях можно увеличить количество позиций в таблице. Что уменьшает число коллизий и сокращает поиск.

Чтобы добавить новый элемент в таблицу с открытой адресацией, ячейки которой занумерованы целыми числами от 0 до $m - 1$, просматриваем её пока не найдём свободное место. Если всякий раз просматривать ячейки подряд (от 0 - ой до $(m - 1)$ – ой) потребуется время $\Theta(n)$, но суть в том, что порядок просмотра таблицы зависит от ключа. Иными словами, добавляем к хеш-функции аргумент – номер попытки (нумерацию начинаем с нуля), так что хеш-функция имеет вид:

$h: U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$, где U – множество ключей.

кол-во попыток

Последовательность испробованных мест, или последовательность проб, для данного ключа k имеет вид: (h (k, 0), h (k, 1), …, h (k, m-1)); функция h должна быть такой, чтобы каждое из чисел от 0 до m-1 встретилось в этой последовательности ровно один раз.

Ниже приводится текст процедуры добавления в таблицу **T** с открытой адресацией; в нём подразумевается, что записи не содержат дополнительной информации, кроме ключа. Если ячейка таблицы пуста, в ней записан NIL (фиксированное значение, отличное от всех ключей).

```
HASH - INSERT (T, k)
1 i ← 0
5     repeat j ← h(k, i)
6             if T[j] = NIL
7                then T[j] ← k
8                        return j
9                else i← i + 1
10    until   i = m
11    error “переполнение хеш-таблицы”
```

При поиске элемента с ключом ***k*** в таблице с открытой адресацией ячейки таблицы просматриваются в том же порядке, что и при добавлении в неё элемента с ключом ***k***. Если при этом натыкаемся на ячейку, в которой записан **NIL**, то можно быть уверенным, что искомого элемента в таблице нет (иначе он был бы занесен в эту ячейку). (Внимание: предполагаем, что никакие элементы из таблицы не удаляются!)

Процедура поиска HASH-SEARCH (T, k) (если элемент с ключом **k** содержится в таблице T в позиции **j**, процедура возвращает **j**, в противном случае она возвращает **NIL**):

```
HASH – SEARCH (T, k)
1   i ← 0
2     repeat j ← h(k, i)
3           if T[j] = k
4                then return j
5                 i ← i + 1
6       until   T[j] = NIL или i = m
7   return NIL
```

Удалять элемент из таблицы с открытой адресацией не так просто. Если просто записать на его место NIL, то в дальнейшем не сможем найти те элементы, в момент добавления которых в таблицу это место было занято (и из-за этого был выбран более далёкий элемент в последовательности испробованных мест). Возможное решение записывать на место удаленного элемента не NIL, а специальное значение DELETED («удален»), и при добавлении рассматривать ячейку с записью DELETED как свободную. а при поиске - как занятую (и продолжать поиск).

Недостаток этого подхода в том, что время поиска может оказаться большим даже при низком коэффициенте заполнения. Поэтому, если требуется удалять записи из хеш-таблицы. Предпочтение обычно отдают хешированию с цепочками.

При анализе открытой адресации будем исходить из предположения, что хеширование равномерно в том смысле, что все $m!$ перестановок множества $\{0, 1, \ldots, m-1\}$ равновероятны.

Обычно применяют следующие три **способа вычисления последовательности испробованных мест**:

– линейный;

– квадратичный;

– двойное хеширование.

В каждом из этих способов последовательность $(h(k, 0), h(k, 1), \ldots, h(k, m-1))$ будет перестановкой множества $\{0, 1, \ldots, m-1\}$ при любом значении ключа k, но ни один из этих способов не является равномерным по той причине, что они дают не более m^2 перестановок из $m!$ возможных. Больше всего разных перестановок получается при двойном хешировании и на практике этот способ даёт лучшие результаты.

Линейная последовательность проб

Пусть h': $U \to \{0, 1, \ldots, m-1\}$ – обычная хеш-функция. Функция, определяющая **линейную последовательность проб**, задаётся формулой:

$h(k, i) = (h'(k) + i) \bmod m$.

Иными словами, при работе с ключом k начинают с ячейки $T[h'(k)]$, а затем перебирают ячейки таблицы подряд: $T[h'(k) +1]$, $T[h'(k) +2]$, … (после $T[m- 1]$ переходят к $T[0]$).

Открытая адресация с линейной последовательностью проб легок в реализации. Но у этого метода есть один **недостаток**: он может привести к образованию кластеров, то есть длинных последовательностей занятых ячеек, идущих подряд. Это удлиняет поиск.

Линейная последовательность проб довольно далека от равномерного хеширования.

Пример:

k = (78, 69, 72, 50, 98, 14, 85, 59), m = 13,

78				69			72	98			50	
0	1	2	3	4	5	6	7	8	9	10	11	12

h (98, 0) = 98 mod 13 = 7; h (98, 1) = (98 mod 13 + 1) mod 13 = 8

Квадратичная последовательность проб

Функция, определяющая квадратичную последовательность проб, задаётся формулой

$$h(k, i) = (h'(k) + C_1 i + C_2 i^2) \bmod m,$$

где $h'(k)$ – обычная хеш-функция,

$C_1, C_2 \neq 0$ – некоторые константы ($C_1 = 1$, $C_2 = 3$).

Пробы начинаются с ячейки номер $T[h'(k)]$, дальше ячейки просматриваются не подряд: номер пробуемой ячейки квадратично зависит от номера попытки.

Как и при линейном методе, вся последовательность проб определяется своим первым членом, так что опять получается всего m различных перестановок. Тенденции к образованию кластеров больше нет, но аналогичный эффект проявляется в (более мягкой) форме образования вторичных кластеров.

Пример:

k = (78, 69, 72, 50, 98, 14, 85, 59), m = 13,

78				69			72	98			50	
0	1	2	3	4	5	6	7	8	9	10	11	12

h (98, 0) = 98 mod 13 = 7; h (98, 1) = (98 mod 13 + 1 *1 + 3*1) mod 13 = (7 + 4) mod13 = 11;
h (98, 2) = (98 mod 13 + 1 *2 + 3*2^2) mod 13 = (7 + 14) mod13 = 21 mod13 = 8.

Двойное хеширование

Двойное хеширование – один из лучших методов открытой адресации. Перестановки индексов, возникающие при двойном хешировании, обладают многими свойствами, присущими равномерному хешированию.

При двойном хешировании функция h (k, i) имеет вид:

$h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m,$

где h_1, h_2 – обычные хеш-функции, $h_1(k) = k \bmod m$, $h_2(k) = 1 + k \bmod m'$,

$m' = m - 1 \mid m - 2$.

Пример:

$k = (78, 69, 72, 50, 98, 14, 85, 59)$, $m = 13$, $m' = m - 2 = 11$

78				69	98		72				50	
0	1	2	3	4	5	6	7	8	9	10	11	12

$h(98, 0) = (h_1(98) + i\, h_2(98)) \bmod m,$

$h_1(98) = 98 \bmod 13 = 7,$ $h_2(98) = 1 + 98 \bmod 11 = 11,$

$h(98, 1) = (h_1(98) + 1 * h_2(98)) \bmod m = (7 + 1*11) \bmod 13 = 5.$

Упражнения

1. Выполните добавление ключей 5, 28, 19, 15, 20, 33, 12, 17, 1 (в указанном порядке) в хеш-таблицу с открытой адресацией размера m = 9. Для вычисления последовательности проб используется линейный метод с h'(k) = k mod m.
2. Выполните добавление ключей 5, 28, 19, 15, 20, 33, 12, 17, 1 (в указанном порядке) в хеш-таблицу с открытой адресацией размера m = 9. Для вычисления последовательности проб используйте квадратичный метод с h'(k)=k mod m, $c_1 = 1$, $c_2 = 3$.
3. Выполните добавление ключей 5, 28, 19, 15, 20, 33, 12, 17, 1 (в указанном порядке) в хеш-таблицу с открытой адресацией размера m = 9, если используется метод двойного хеширования с $h_1(k)$=kmod m и $h_2(k) = 1 + (k \bmod (m - 1))$.
4. Рассмотрите вставку ключей 10, 22,31, 4, 15, 28, 17, 88, 59 в хеш-таблицу длины m=11 с открытой адресацией и вспомогательной хеш-функцией h'(k)=k mod m. Проиллюстрируйте результат вставки приведенного списка ключей при использовании линейного исследования, квадратичного исследования с $c_1 = 1$ и $c_2 = 3$, и двойного хеширования с $h_2(k) = 1 + (k \bmod (m - 1))$.

ЛЕКЦИЯ № 10

Динамическое программирование. (ДП). Задача об умножении последовательности матриц. Применение ДП

Подобно методу «разделяй и властвуй» динамическое программирование разбивает задачу на подзадачи и объединяет их решения. Как уже видели алгоритмы типа «разделяй и властвуй» делят задачу на подзадачи, эти подзадачи — на более мелкие подзадачи и так далее, собирают решение основной задачи «снизу вверх». Динамическое программирование применимо тогда, когда подзадачи не являются независимыми, когда у подзадач есть общие «подподзадачи». В этом случае типа «разделяй и властвуй» будет делать лишнюю работу, решая одни и те же подподзадачи по нескольку раз. Алгоритм, основанный на динамическом программировании, решает каждую из подзадач единожды и запоминает ответы в специальной таблице. Это позволяет не вычислять заново ответ к уже встречавшейся подзадаче.

В типичном случае динамическое программирование применяется **к задачам оптимизации** (optimization problems). У такой задачи может быть много возможных решений; их «качество» определяется значением какого-то параметра; и требуется выбрать оптимальное решение, при котором значение параметра будет минимальным или максимальным (в зависимости от постановки задачи). Вообще говоря, оптимум может достигаться для нескольких разных решений.

Алгоритм, основанный на динамическом программировании:

Шаг 1. Описать строение оптимальных решений.

Шаг 2. Выписать рекуррентное соотношение, связывающее оптимальные значения параметра для подзадач.

Шаг 3. Двигаясь снизу вверх, вычислить оптимальное значение параметра для подзадач.

Шаг 4. Пользуясь полученной информацией, построить оптимальное решение.

Основную часть работы составляют шаги 1-3. Если интересует только оптимальное значение параметра, шаг 4 не нужен. Если же шаг 4 необходим, для построения оптимального решения иногда приходится получать и дополнительную информацию в процессе выполнения шага 3.

Оптимизационные задачи, решаемые с помощью динамического программирования:

- как найти произведение нескольких матриц, сделав как можно меньше умножений;
- нахождение наибольшей общей подпоследовательности двух последовательностей;
- нахождение оптимальной триангуляции выпуклого многоугольника (декомпозиция многоугольника на множество треугольников, внутренние области которых попарно не пересекаются и объединение которых в совокупности составляет многоугольник).

Перемножение нескольких матриц

Найти произведение

$$A_1 \cdot A_2 \cdot \ldots \cdot A_n \qquad (10.1)$$

последовательности n матриц $(A_1, A_2, \ldots, A_n)$. Будем пользоваться стандартным алгоритмом перемножения двух матриц в качестве подпрограммы. Но прежде надо расставить скобки в (10.1), чтобы указать порядок умножений. Будем говорить, что в произведении матриц полностью расставлены скобки (product is fully parenthesized), если это произведение либо состоит из одной-единственной матрицы, либо является заключенным в скобки произведением двух произведений с полностью расставленными скобками. Поскольку умножение матриц ассоциативно, конечный результат вычислений не зависит от расстановки скобок. Например, в произведении $A_1 \cdot A_2 \cdot A_3 \cdot A_4$ можно полностью расставить скобки пятью разными способами:

$$(A_1\,(A_2\,(A_3\,A_4))),\quad (A_1\,((A_2\,A_3)\,A_4)),\quad ((A_1\,A_2)(A_3\,A_4)),$$
$$((A_1\,(A_2\,A_3))\,A_4),\quad ((A_1\,A_2)\,A_3)\,A_4)$$

во всех случаях ответ будет один и тот же.

Не влияя на ответ, способ расстановки скобок может сильно повлиять на стоимость перемножения матриц. Посмотрим сначала, сколько операций требует перемножение двух матриц. Вот стандартный алгоритм (rows и columns обозначают количество строк и столбцов матрицы соответственно):

```
MATRIX-MULTIPLY (A, B)
1 if columns[A] ≠ rows[B]
2     then error «умножить нельзя»
3     else for r ← 1 to rows [A]
4             do for j ← 1 to columns [B]
5                     do C [i, j] ← 0
6                         for k ← 1 to columns [A]
7                             do C [i, j] ← C [i, j] + A [i, k] · B [k, j]
8 return C
```

Матрицы A и B можно перемножать, только если число столбцов у **A** равно числу строк у ***B***.

Если A — это (p × q)-матрица,

B — это (q × r)-матрица,

то их произведение C = A • B является (p × r)-матрицей.

При выполнении этого алгоритма делается p×q×r умножений (строка 7) и примерно столько же сложений. Для простоты будем учитывать только умножения и принимать как данность простейший способ умножения матриц и ищем оптимум за расстановки скобок.

```
MATRIX-MULTIPLY (A, B)
1 if columns[A] ≠ rows[B]
2    then error «умножить нельзя»
3    else for r ←  1 to rows [A]      p
4          do for j ← 1 to columns [B]    r
5               do C [i, j] ← 0
6                    for k ← 1 to columns [A]    q
7                         do C [i, j] ← C [i, j] + A [i, k] · B [k, j]
8 return C
```

Пример влияния расстановки скобок на стоимость:

Рассмотрим последовательность из трех матриц $(A_1A_2A_3)$ размеров 10×100, 100×5 и 5×50 соответственно.

1) При вычислении $((A_1A_2)A_3)$:

A_1A_2 - (10×5)-матриц: 10×100×5 = 5000 умножений,

$((A_1A_2)A_3)$ - 10×5×50 = 2500 умножений.

Всего 7500 умножений.

b) При расстановке скобок $(A_1(A_2A_3))$:

A_2A_3 - (100×50)-матрица: 100×5×50 = 25000 умножений,

умножение A_1 на A_2A_3: 10×100×50 = 50000 умножений,

итого 75000 умножений.

Тем самым, первый способ расстановки скобок в 10 раз выгоднее.

Задача об умножении последовательности матриц (matrix-chain multiplication problem) может быть сформулирована следующим образом: дана последовательность из *n* матриц $(A_1, A_2, \ldots, A_n)$ заданных размеров (матрица A_i размер $p_{i-1} \times p_i$). Требуется найти такую (полную) расстановку скобок в произведении $A_1 \cdot A_2 \cdot \ldots \cdot A_n$, чтобы вычисление произведения требовало наименьшего числа умножений.

Количество расстановок скобок

Прежде чем применять динамическое программирование к задаче об умножении последовательности матриц, стоит убедиться, что простой перебор всех возможных расстановок скобок не даст эффективного алгоритма. Обозначим символом $P(n)$ количество полных расстановок скобок в произведении n матриц. Последнее умножение может происходить на границе между k-й и $(k+1)$-й матрицами. До этого отдельно вычисляем произведение первых k и остальных $n - k$ матриц. Поэтому

$$P(n) = \begin{cases} 1, & \text{если } n = 1; \\ \sum_{k=1}^{n-1} P(k)P(n-k), & \text{если } n \geq 2. \end{cases}$$

Это соотношение задаёт последовательность **чисел Каталана**

$$P(n) = \begin{cases} 1, & \text{если } n = 1; \\ \sum_{k=1}^{n-1} P(k)P(n-k), & \text{если } n \geq 2. \end{cases}$$

$$P(n) = C(n-1),$$

Числа Каталана $C(n)$ для n = 0,1,2, … образуют последовательность:

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452, …

$$C_{2n}^{n} = \frac{(2n)!}{n!\,(2n-n)!} \leq \frac{(2n)^{2n}}{n^n n^n} = 4^n; \quad n! = \sigma(n^n);$$

где

$$C(n) = \frac{C_{2n}^{n}}{n+1} \leq \frac{4^n}{n+1} \geq \frac{4^n}{n^{3/2}} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

Следовательно, число расстановок экспоненциально зависит от **n**, так что полный перебор неэффективен.

Шаг 1: строение оптимальной расстановки скобок

Если собираемся воспользоваться динамическим программированием, то для начала должны описать строение оптимальных решений. Для задачи об умножении последовательности матриц это выглядит следующим образом. Обозначим для удобства через $A_{i..j}$ матрицу, являющуюся произведением $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$. Оптимальная расстановка скобок в произведении A_1, A_2, …, A_n разрывает последовательность между A_k и A_{k+1} для некоторого k, удовлетворяющего неравенству $1 \leq k < n$. Иными словами, при вычислении произведения, диктуемом этой расстановкой скобок, сначала вычисляем произведения $A_{1..k}$ и $A_{k+1..n}$ затем перемножаем их и получаем окончательный ответ $A_{1..n}$. Стало быть, стоимость этой оптимальной расстановки равна стоимости вычисления матрицы $A_{1..k}$ плюс стоимость вычисления матрицы $A_{k+1..n}$ плюс стоимость перемножения этих двух матриц.

Чем меньше умножений нам потребуется для вычисления $A_{1..k}$ и $A_{k+1..n}$, тем меньше будет общее число умножений. Стало быть, оптимальное решение задачи о перемножении последовательности матриц содержит оптимальные решения задач о перемножении её частей.

Шаг 2: Рекуррентное соотношение

Теперь надо выразить стоимость оптимального решения задачи через стоимости оптимальных решений её подзадач. Такими подзадачами будут задачи об оптимальной расстановке скобок в произведениях $A_{i..j} = A_i\ A_{i+1}\ \ldots\ A_j$ для $1 \le i \le j \le n$. Обозначим через m[i, j] минимальное количество умножений, необходимое для вычисления матрицы $A_{i...j}$; в частности, стоимость вычисления всего произведения $A_{1..n}$ есть m [l, n].

Числа m[i, j] можно вычислить так. Если i=j, то последовательность состоит из одной матрицы $A_{i..i} = A_i$ и умножения вообще не нужны. Стало быть, m[i, j] = 0 для i = 1, *2,..., n.* Чтобы подсчитать m[i,j] для i < j, воспользуемся информацией о строении оптимального решения, полученной на шаге 1. Пусть при оптимальной расстановке скобок в произведении $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ последним идет умножение $A_i \cdot \ldots \cdot A_k$ на $A_{k+1} \cdot \ldots \cdot A_j$, где $i \le k < j$. Тогда m[i, j] равно сумме минимальных стоимостей вычисления произведений $A_{i..k}$ и $A_{k+1..j}$ плюс стоимость перемножения этих двух матриц. Поскольку для вычисления произведения $A_{i..k} \cdot A_{k+1..j}$ требуется $p_{i-1}\, p_k\, p_j$ умножений,

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}\, p_k\, p_j$$

$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$

В этом соотношении подразумевается, что оптимальное значение известно; на деле это не так. Однако число k может принимать всего лишь j - i различных значений: i, i+1,..., j-1. Поскольку одно из них оптимально, достаточно перебрать эти значения k и выбрать наилучшее. Получаем рекуррентную формулу:

$$m[i,j] = \begin{cases} 0, & \text{при } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}, & \text{при } i < j. \end{cases} \quad (10.2)$$

Числа $m[i, j]$ — стоимости оптимальных решений подзадач. Чтобы проследить за тем, как получается оптимальное решение, обозначим через оптимальное место последнего умножения, то есть такое k, что при оптимальном вычислении произведения $A_i A_{i+1} \bullet \ldots \bullet A_j$ последним идет умножение $A_i \bullet \ldots \bullet A_k$ на $A_{k+1} \bullet \ldots \bullet A_j$. Иными словами, $s[i, j]$ равно числу k, для которого

$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_k p_j$.

Шаг 3: вычисление оптимальной стоимости

Пользуясь соотношениями $m|i,j| = \begin{cases} 0, & \text{при } i = j, \\ \min\limits_{i \le k < j} \{m|i,k| + m|k+1,j| + p_{i-1}p_k p_j\}, & \text{при } i < j. \end{cases}$ (10.2), был написан рекурсивный алгоритм, определяющий минимальную стоимость вычисления произведения $A_1A_2 \ldots A_n$ (т.е. число $m[1, n]$). Однако время работы такого алгоритма экспоненциально зависит от n, так что этот алгоритм не лучше полного перебора. Настоящий выигрыш во времени получим, если воспользуемся тем, что подзадач относительно немного: по одной задаче для каждой пары, которой $1 \le i \le j \le n$, а всего $C_n^2 + n = \Theta(n^2)$. Экспоненциальное время возникает потому, что рекурсивный алгоритм решает каждую из подзадач помногу раз, на разных ветвях дерева рекурсии. Такое «перекрытие подзадач» характерный признак задач, решаемых методом динамического программирования.

Вместо рекурсии вычислим оптимальную стоимость «снизу вверх». В нижеследующей программе предполагается, что матрица A_i имеет размер $p_{i-1} \times p_i$ при $i=1,2,\ldots,n$. На вход подаётся последовательность $p = \langle p_0, p_1, \ldots p_n \rangle$, где *length[p]*=*n*+1. Программа использует вспомогательные

таблицы m[1..n, 1..n] (для хранения стоимостей m[i, j]) и s[1..n, 1..n] (в ней отмечается, что при k достигается оптимальная стоимость при вычислении m[i, j]).

$$C_n^2 = \frac{(n)!}{2!\,(n-2)!} = \frac{(n-2)!\,(n-1)n}{2!\,(n-2)!} = O(n^2)$$

```
MATRIX-CHAIN-ORDER(p)
1   n ← length[p] – 1
2   for i ←1 to n
3       do  m[i, i] ← 0
4   for l ←2 to n
5       do for i ←1  to n – l + 1
6              do j ← i + l – 1
7                 m[i, j] ← ∞
8                 for k ← i to j – 1
9                    do q← m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
10                      if q < m[i, j]
11                        then m[i, j] ← q
12                             s[i, j] ← k
13  return m, s
```

Заполняя таблицу m, этот алгоритм последовательно решает задачи об оптимальной расстановке скобок для одного, двух, ..., n сомножителей. Соотношение (10.2) показывает, что число m[i,j] — стоимость перемножения j-i+1 матриц — зависит только от стоимостей перемножения меньшего (чем j-i+1) числа матриц. Именно, для k=i, i+, ..., j-1 получается, что $A_{i..k}$ — произведение k-i+1 < j-i+1 матриц, а $A_{k+i..j}$ — произведение j-k < j-i+1 матриц.

$$m|i,j| = \begin{cases} 0, & \text{при } i = j, \\ \min\limits_{i \le k < j} \{m|i,k| + m|k+1,j| + p_{i-1}p_k p_j\}, & \text{при } i < j. \end{cases}$$

Сначала (в строках 2-3) алгоритм выполняет присваивания m[i, i]←0 для i=1,2,…,n стоимость перемножения последовательности из одной матрицы равна нулю. При первом исполнении цикла (строки 4-12) вычисляются (с-помощью соотношений (10.2)) значения m[i, i+1] для i = 1,2,..., n-1 — это минимальные стоимости для последовательностей длины 2. При втором проходе вычисляются m[i, i+2] для i = 1,2,..., n - 2 — минимальные стоимости перемножения последовательностей длины 3, и так далее. На каждом шаге значение m[i, j], вычисляемое в строках 9-12, зависит только от вычисленных ранее значений m[i,k] и m[k+1,j].

На рисунке 10.1 показано, как это происходит при $n = 6$. Поскольку определяем $m[i, j]$ только для $i \leq j$, используется часть таблицы, лежащая над главной диагональю. На рисунке таблицы повёрнуты (главная диагональ горизонтальна).

Внизу выписана последовательность матриц. Число $m[i, j]$—минимальная стоимость перемножения $A_iA_{i+1} \bullet \ldots \bullet A_j$ — находится на пересечении диагоналей, идущих вправо-вверх от матрицы A_i и влево-вверх от матрицы A_j. В каждом горизонтальном ряду собраны стоимости перемножения подпоследовательностей фиксированной длины. Для заполнения клетки $m[i, j]$ нужно знать произведения $p_{i-1}p_i p_j$ для $k = i, i+1, \ldots, j-1$ и содержимое клеток, лежащих слева-внизу и справа-внизу от $m[i, j]$.

Простая оценка показывает, что время работы алгоритма MATRIX-CHAIN-ORDER есть $O(n^3)$. В самом деле, число вложенных циклов равно трём, и каждый из индексов j, i и k принимает не более n значений.

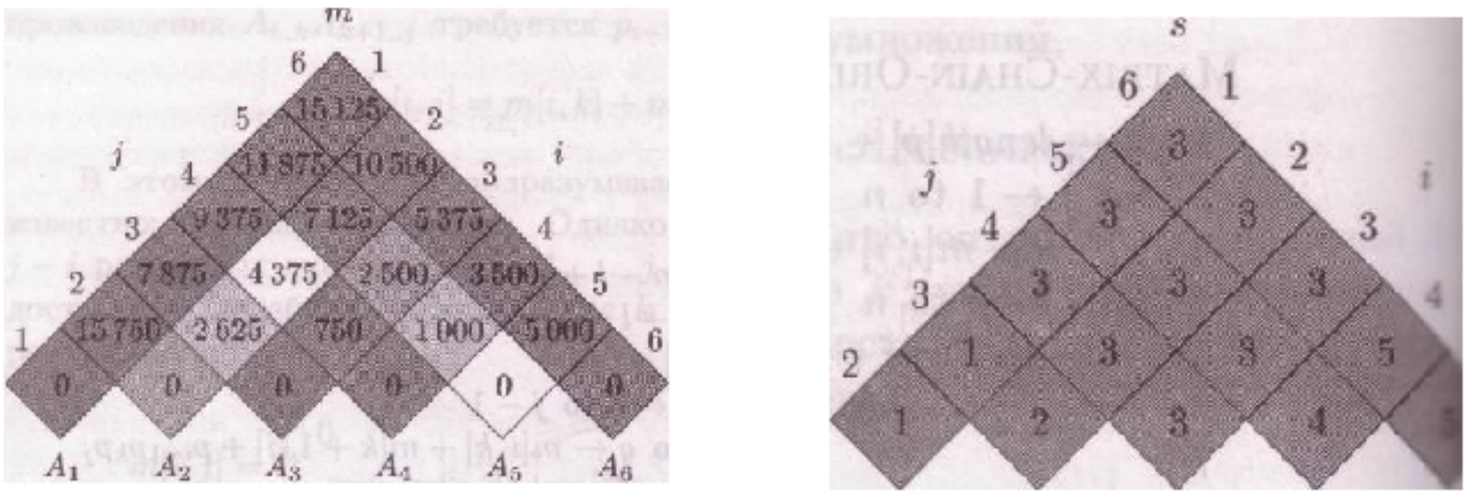


Рисунок 10.1 - Таблицы ***m*** и ***s***, вычисляемые процедурой MATRIX-CHAIN-ORDER для n=6 и матриц следующего размера:

Размер матриц:

Матрица	A_1	A_2	A_3	A_4	A_5	A_6
Размерность	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

Таблицы повёрнуты так, что главная диагональ горизонтальна. В таблице m используются только клетки, лежащие не ниже главной диагонали, в таблице s — только клетки, лежащие строго выше. Минимальное количество умножений, необходимое для перемножения все матриц, равно m[1,6]=15125. Пары клеточек, заштрихованных одинаковой светлой штриховкой, совместно входят в правую часть формулы в процессе вычисления m[2, 5] (строка 9 процедуры MATRIX-CHAIN-ORDER):

$$m[2,5] = \min_{2 \le k < 5} \begin{cases} m[i,k] + m[k+1,j] + p_{i-1}p_k p_j = m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 * 15 * 20 = 13000 \\ m[i,k] + m[k+1,j] + p_{i-1}p_k p_j = m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 * 5 * 20 = 7125 \\ m[i,k] + m[k+1,j] + p_{i-1}p_k p_j = m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 * 10 * 20 = 11375 \end{cases}$$

Время работы этого алгоритма есть $\Theta(n^3)$. Объём памяти, необходимый для хранения таблиц m и s, есть $\Theta(n^2)$. Тем самым этот алгоритм значительно эффективнее, чем требующий экспоненциального времени перебор всех расстановок.

Шаг 4: построение оптимального решения

Алгоритм matrix-chain-order находит минимальное число умножений, необходимое для перемножения последовательности матриц. Осталось выполнить расстановку скобок, приводящих к такому числу умножений.

Для этого используем таблицу $s[1..n, 1..n]$. В клетке $s[1,n]$ место последнего умножения при оптимальной расстановке скобок; другими словами, при оптимальном способе вычисления $A_{1..n}$ последним идёт умножение $\mathbf{A_{1..s[i,n]}}$ на $\mathbf{A_{s[1,n]+1..n}}$. Предшествующие умножения можно найти рекурсивно значение $s[1,s[1,n]]$ определяет последнее умножение при нахождении $A_{1..s[1,n]}$, а $s[s[1,n]+1,n]$ определяет последнее умножение при вычислении $A_{s[1,n]+1..n}$. Приведённая ниже рекурсивная процедура вычисляет произведение $A_{i...j.}$, имея следующие данные: последовательность матриц $A = (A_1,A_2,..., A_n)$, таблицу s, найденную процедурой MATRIX-CHAIN-ORDER, а также индексы i и *j*. Произведение $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ равно MATRIX-CHAIN-MULTIPLY(A, S, 1, n).

```
MATRIX- CHAIN- MULTIPLY (A, s, i, j)

1 if j > i
2    then X ← MATRIX-CHAIN-MULTIPLY (A, s,i, s[i, j])
3         У ← MATRIX-CHAIN-MULTIPLY (A, s, s[i, j] + 1, j)
4         return MATRIX-MULTIPLY (X, У)
5    else return A_i
```

В примере на рисунке 10.1 вызов MATRIX-CHAIN-MULTIPLY(A, s, 1, 6) вычислит произведение шести матриц в соответствии с расстановкой скобок

$$((A_1(A_2A_3))\ ((A_4\ A_5)A_6)). \qquad (10.3)$$

Распечатать скобки в произведении последовательности матриц:

```
PRINT_OPTIMAL_PARENS (S, i, j)

1. if i = j
2.       then print "A" i
3.       else print "("
4.       PRINT_OPTIMAL_PARENS (S, i, S[i, j])
5.       PRINT_OPTIMAL_PARENS (S, S[i, j]+1, j)
6.       PRINT ")"
```

Упражнения

1. Найдите оптимальную расстановку скобок в произведении последовательности матриц, размерности которых равны (5, 10, 3, 12, 8, 50, 6).
2. Найдите оптимальную расстановку скобок в произведении последовательности матриц, размерности которых равны (4, 10, 15, 8, 5).

Пример: Найти оптимальную расстановку скобок в произведении последовательности матриц,

размерности которых равны

p = (4, 10, 15, 8, 5), A_1 = (4 x 10); A_2 = (10 x 15);

A_3 = (15 x 8); A_4 = (8 x 5)

Дано: $p = < 4, 10, 15, 8, 5 >$, $n = 4$

p_0 p_1 p_2 p_3 p_4

Трассировка:

1 n=4

2 i= 1, 4

3 m[1, 1] = ∅, m[2, 2] = ∅, m[3, 3] = ∅, m[4, 4] = ∅,

4 l= 2, 4

5 do for i=1 to 4 – 2 + 1 = 3

6 do j=1 + 2 – 1 = 2

7 m[1, 2] ← ∞

8 for k ← 1 to 2 – 1 = 1

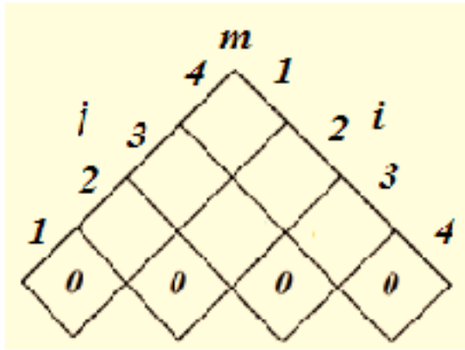
9 do q ← m[1, 1] + m[2, 2] + $p_0p_1p_2$ =

= 0 + 0 + 4·10·15 = 600

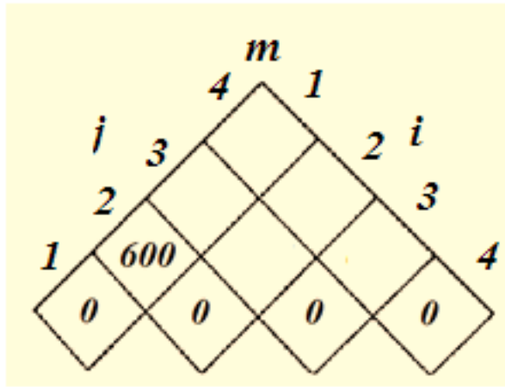
10 if 600 < ∞

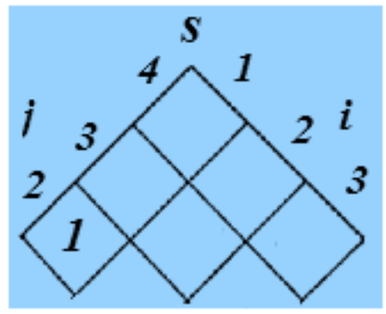
11 then m[1, 2] ← 600

12 s[1, 2] ← 1

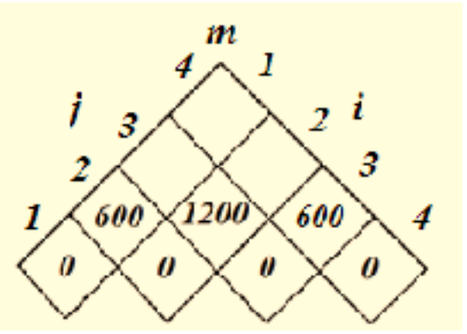



```
MATRIX-CHAIN-ORDER(p)
1  n ← length[p] - 1
2  for i ← 1 to n
3     do m[i, j] ← 0
4  for l ← 2 to n
5     do for i ← 1  to n - l + 1
6           do j ← i + l - 1
7              m[i, j] ← ∞
8              for k ← i to j - 1
9                 do q ← m[i, k] + m[k+1, j] + p_i-1 p_k p_j
10                if q < m[i, j]
11                   then m[i, j] ← q
12                        s[i, j] ← k
13 return m, s
```



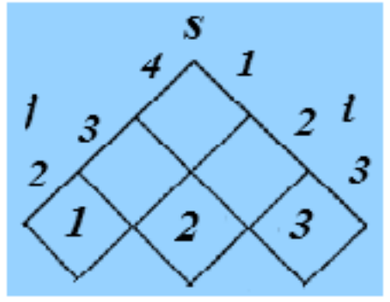


```
5   do for i=2 to 4 – 2 + 1 = 3
6           do j=2 + 2 – 1 = 3
7               m[2, 3] ← ∞
8               for k ← 2 to 3 – 1 = 2
9                   do q ← m[2, 2] + m[3, 3] + p1p2p3 = 0 + 0 +
                                                  +10·15·8 =1200
10                      if 1200 < ∞
11                          then m[2, 3] ← 1200
12                               s[2, 3] ← 2
```




```
MATRIX-CHAIN-ORDER(p)
1   n ← length[p] - 1
2   for i ← 1 to n
3   do m[i, j] ← 0
4   for l ← 2 to n
5   do for i ← 1 to n - l + 1
6          do j ← i – l – 1
7              m[i, j] ← ∞
8              for k ← i to j - 1
9                  do q ← m[i, k] + m[k + 1, j] + pi-1 pk pj
10                 if q < m[i, j]
11                     then m[i, j] ← q
12                          s[i, j] ← k
13  return m, s
```

```
5   do for i=3 to 4 – 2 + 1 = 3
6           do j=3 + 2 – 1 = 4
7               m[3, 4] ← ∞
8               for k ← 3 to 4 – 1 = 3
9                   do q ← m[3, 3] + m[4, 4] + p2p3p4 = 0 + 0 + 15·8·5 = 600
10                      if 600 < ∞
11                          then m[3, 4] ← 600
12                               s[3, 4] ← 3
```



4 $l = \overline{3, 4}$

5 do for i=1 to 4 – 3 + 1 = 2

6 do j=1 + 3 – 1 = 3

7 m[1, 3] ← ∞

8 for k ← 1 to 3 – 1 = 2

9 do q ← m[1, 1] + m[2, 3] + $p_0p_1p_3$ = 0 + 1200 + + 4·10·8 = 1520

10 if 1520 < ∞

11 then m[1, 3] ← 1520

12 s[1, 3] ← 1

```
MATRIX-CHAIN-ORDER(p)
1  n ← length[p] - 1
2  for i ← 1 to n
3  do m[i, j] ← 0
4  for l ← 2 to n
5  do for i ← 1 to n - l + 1
6        do j ← i + l - 1
7           m[i, j] ← ∞
8           for k ← i to j - 1
9              do q ← m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
10                if q < m[i, j]
11                   then m[i, j] ← q
12                        s[i, j] ← k
13 return m, s
```

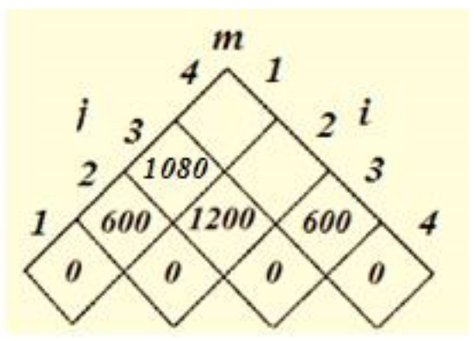
8 for k ← 2 to 3 – 1 = 2

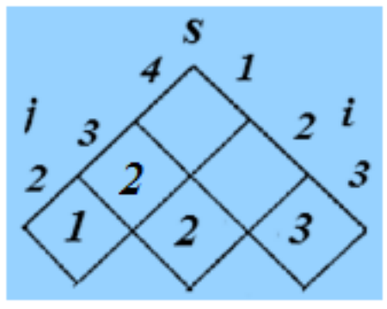
9 do q ← m[1, 2] + m[3, 3] + $p_0p_2p_3$ = 600 + 0 + 4·15·8 = 1080

10 if 1080 < 1520

11 then m[1, 3] ← 1080

12 s[1, 3] ← 2





5 do for i=2 to 4 – 3 + 1 = 2

6 do j=2 + 3 – 1 = 4

7 m[2, 4] ← ∞

8 for k ← 2 to 4 – 1 = 3

9 do q ← m[2, 2] + m[3, 4] + $p_1p_2p_4$ = 0 + 600 + +10·15·5 = 1350

10 if 1350 < ∞

11 then m[2, 4] ← 1350

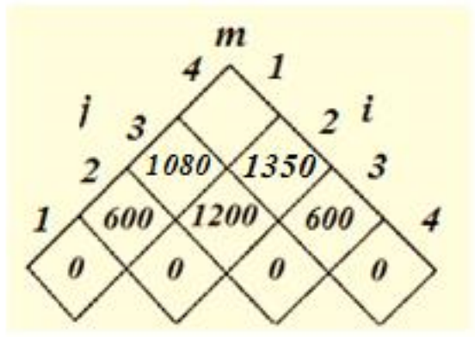
12 s[2, 4] ← 2

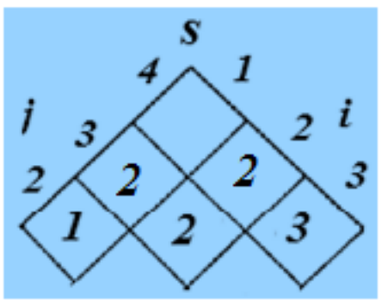
```
MATRIX-CHAIN-ORDER(p)
1  n ← length[p] - 1
2  for i ← 1 to n
3  do m[i, j] ← 0
4  for l ← 2 to n
5  do for i ← 1 to n - l + 1
6        do j ← i + l - 1
7           m[i, j] ← ∞
8           for k ← i to j - 1
9              do q ← m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
10                if q < m[i, j]
11                   then m[i, j] ← q
12                        s[i, j] ← k
13 return m, s
```

8 for k ← 3 to 4 – 1 = 3

9 do q ← m[2, 3] + m[4, 4] + $p_1p_3p_4$ = 1200 + 600 + + 10·8·5 = 2200

10 if 2200 < 1350





4 $l = \overline{4, 4}$

5 do for i=1 to 4 – 4 + 1 = 1

6 do j=1+ 4 – 1 = 4

7 m[1, 4] ← ∞

8 for k ← 1 to 4 – 1 = 3

9 do q ← m[1, 1] + m[2, 4] + $p_0p_1p_4$ = 0 + 1350 + + 4·10·5 = 1550

10 if 1550 < ∞

11 then m [1, 4] ← 1550

12 s[1, 4] ← 1

8 for k ← 2 to 4 – 1 = 3

9 do q ← m[1, 2] + m[3, 4] + $p_0p_2p_4$ = 600 + + 600 + 4·15·5 = 1500

10 if 1500 < 1550

11 then m[1, 4] ← 1500

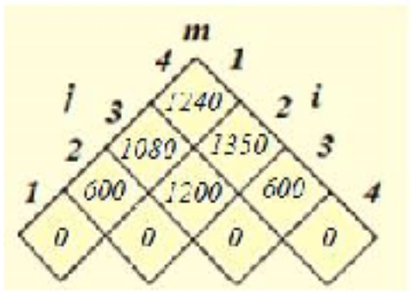
12 s[1, 4] ← 2

8 for k ← 3 to 4 – 1 = 3

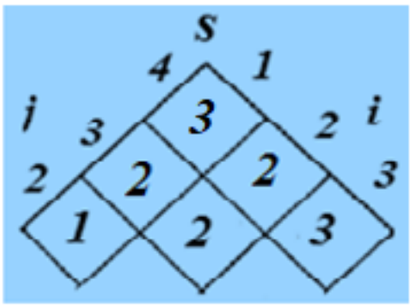
9 do q ← m[1, 3] + m[4, 4] + $p_0p_3p_4$ = 1080 + 0 + 4·8·5 = 1240

10 if 1200 < 1500

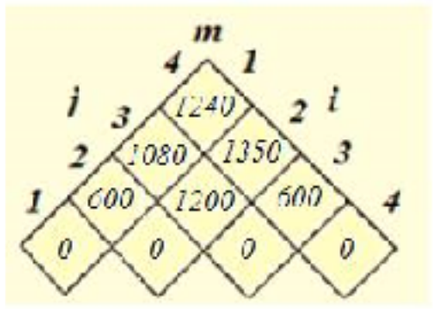
11 then m[1, 4] ← 1240

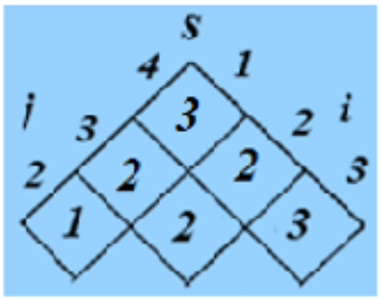



```
MATRIX-CHAIN-ORDER(p)
1  n ← length[p] - 1
2  for i ← 1 to n
3  do m[i, j] ← 0
4  for l ← 2 to n
5  do for i ← 1 to n - l + 1
6        do j ← i + l - 1
7           m[i, j] ← ∞
8           for k ← i to j - 1
9              do q ← m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
10             if q < m[i, j]
11                then m[i, j] ← q
12                     s[i, j] ← k
13 return m, s
```



```
12                              s[1, 4] ← 3
13    return m, s
```





PRINT_OPTIMAL_PARENS (S, 1, 4)

1. if 1 ≠ 4
3. else print "("
4. PRINT_OPTIMAL_PARENS (S, 1, S[1, 4]=3)

4 PRINT_OPTIMAL_PARENS (S, 1, 3)

1. if 1 ≠ 3
3. else print "("
4. PRINT_OPTIMAL_PARENS (S, 1, S[1, 3]=2)

((

```
PRINT OPTIMAL PARENS (S, i, j)

1. if i = j
2.     then print "A" i
3.     else print "("
4.     PRINT OPTIMAL PARENS (S, i, S[i, j])
5.     PRINT OPTIMAL PARENS (S, S[i, j]+1, j)
6.     PRINT ")"
```

4 PRINT_OPTIMAL_PARENS (S, 1, 2)

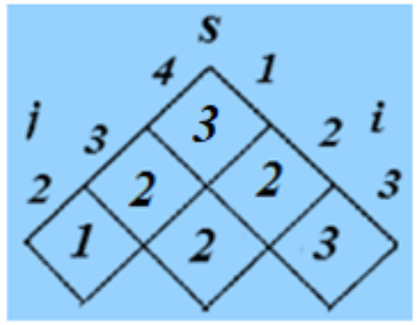
1. if 1 ≠ 2
3. else print "("
4. PRINT_OPTIMAL_PARENS (S, 1, S[1, 2]=1)

4 PRINT_OPTIMAL_PARENS (S, 1, 1)

1. if 1 = 1
2. then print "A" 1
5 PRINT_OPTIMAL_PARENS (S, 2, 2)

5 PRINT_OPTIMAL_PARENS (S, 2, 2)

1 if 2 = 2
2 then print "A" 2
6 PRINT ")"



(((A1A2)

5 PRINT_OPTIMAL_PARENS (S, 3, 3)

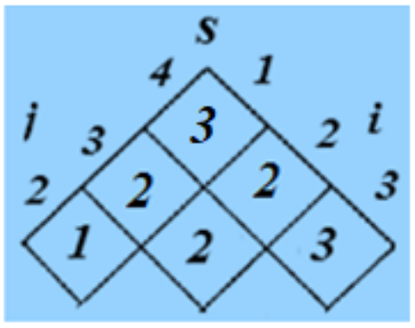
1. if 3 = 3
2. then print "A" 3
6 PRINT ")"

```
PRINT_OPTIMAL_PARENS (S, i, j)
1. if i – j
2.     then print "A" i
3.     else print "("
4.     PRINT_OPTIMAL_PARENS (S, i, S[i, j])
5.     PRINT_OPTIMAL_PARENS (S, S[i, j] + 1, j)
6.     PRINT ")"
```

5 PRINT_OPTIMAL_PARENS (S, 4, 4)

1. if 4 = 4
2. then print "A" 4
6. PRINT ")"

(((A1A2)A3)A4)



$m[1, 1] = \emptyset$, $m[2, 2] = \emptyset$, $m[3, 3] = \emptyset$, $m[4, 4] = \emptyset$;

$m[1, 2] = m[1, 1] + m[2, 2] + p_0p_1p_2 = 0 + 0 + 4 \cdot 10 \cdot 15 = 600$; $k = 1$;

$m[2, 3] = m[2, 2] + m[3, 3] + p_1p_2p_3 = 0 + 0 + 10 \cdot 15 \cdot 8 = 1200$; $k = 2$;

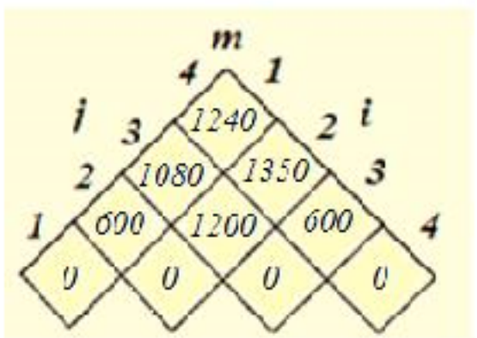
$m[3, 4] = m[3, 3] + m[4, 4] + p_2p_3p_4 = 0 + 0 + 15 \cdot 8 \cdot 5 = 600$; $k = 3$;

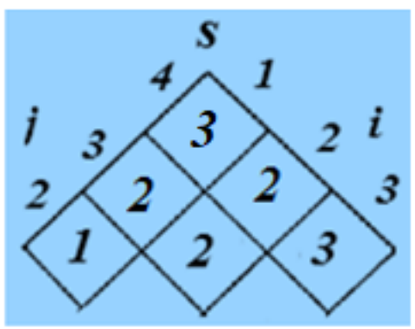
$$m[1, 3] = \min_{1 \le k < 3} \begin{cases} m[1,1] + m[2,3] + p_0p_1p_3 = 0 + 1200 + 4 \cdot 10 \cdot 8 = 1520, & k = 1; \\ m[1,2] + m[3,3] + p_0p_2p_3 = 600 + 0 + 4 \cdot 15 \cdot 8 = 1080, & k = 2; \end{cases}$$

$$m[2, 4] = \min_{1 \le k < 3} \begin{cases} m[2,2] + m[3,4] + p_1p_2p_4 = 0 + 600 + 10 \cdot 15 \cdot 5 = 1350, & k = 2; \\ m[2,3] + m[4,4] + p_1p_3p_4 = 1200 + 600 + 10 \cdot 8 \cdot 5 = 2200, & k = 3; \end{cases}$$

$$m[1, 4] = \min_{1 \le k < 4} \begin{cases} m[1,1] + m[2,4] + p_0p_1p_4 = 0 + 1350 + 4 \cdot 10 \cdot 5 = 1550, & k = 1; \\ m[1,2] + m[3,4] + p_0p_2p_4 = 600 + 600 + 4 \cdot 15 \cdot 5 = 1500, & k = 2; \\ m[1,3] + m[4,4] + p_0p_3p_4 = 1080 + 0 + 4 \cdot 8 \cdot 5 = 1240, & k = 3. \end{cases}$$

(((A1A2)A3)A4)





Упражнения:

1 Алгоритмы Дейкстра, Прима, Крускала