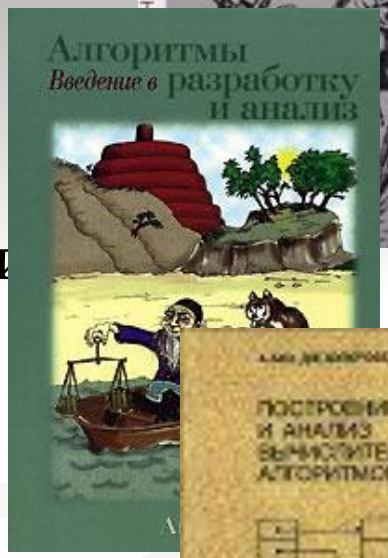
Разработка и анализ алгоритмов

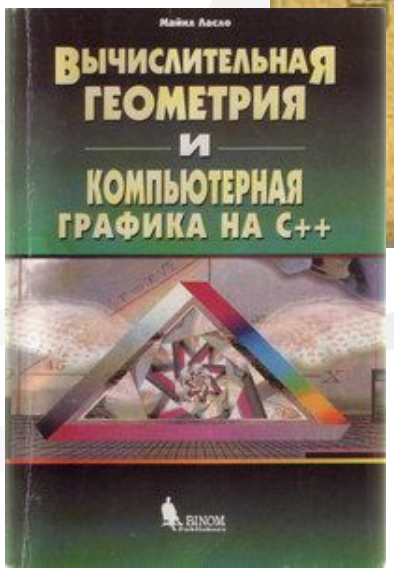
Алгоритмы
Введение в разработку и анализ

Ананий Левитин

Литература по теме

- Кормен Т., Лейзерсон Ч., Ривест Р.
  «Алгоритмы. Построение и анализ»
  (главы 1, 2, 4)
- Левитин А.
  «Алгоритмы. Введение в разработку и анализ»
  (глава 2)
- Ахо А., Хопкрофт Дж., Ульман Дж.
  «Построение и анализ вычислительных алгоритмов»
  (глава 1)
- Майкл Ласло
  «Вычислительная геометрия и компьютерная графика на C++»
  (глава 2)





Анализ вычислительной сложности алгоритмов

Алгоритм Евклида (III в. до н.э)

$$\gcd(m, n) = \gcd(n, m \bmod n).$$

Здесь выражение $(m \bmod n)$ является остатком от деления m на n. Выполнение алгоритма заканчивается, когда выражение $(m \bmod n)$ становится равным нулю. Поскольку $\gcd(m, 0) = m$ (понятно, почему?), последнее полученное значение m будет также являться НОД исходных чисел m и n.

Например, вычисление НОД пары чисел (60, 24) можно выполнить следующим образом:

$$\gcd(60, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

Псевдокод

Алгоритм $Euclid(m, n)$

// Алгоритм Евклида вычисляет значение функции $\gcd(m, n)$
// **Входные данные:** два неотрицательных целых числа m и n,
// которые не могут одновременно быть равны нулю
// **Выходные данные:** наибольший общий делитель чисел m и n
while $n \neq 0$ **do**
  $r \leftarrow m \mod n$
  $m \leftarrow n$
  $n \leftarrow r$
return m

Псевдокод

1. Отступ от левого поля указывает на уровень вложенности

2. Циклы **while**, **for**, **repeat** и условные конструкции **if**, **then**, **else** имеют тот же смысл, что и в Паскале.

3. Символ ▷ начинает комментарий (идущий до конца строки).

4. Одновременное присваивание $i \leftarrow j \leftarrow e$ (переменные i и j получают значение e) заменяет два присваивания $j \leftarrow e$ и $i \leftarrow j$ (в этом порядке).

5. Переменные (в данном случае i, j, key) локальны внутри процедуры (если не оговорено противное).

6. Индекс массива пишется в квадратных скобках: $A[i]$ есть i-й элемент в массиве A. Знак «..» выделяет часть массива: $A[1..j]$ обозначает участок массива A, включающий $A[1], A[2], \ldots, A[j]$.

Введение

Теория сложности вычислений (вычислительной сложности алгоритма) – раздел теории вычислений, **изучающий стоимость работы**, требуемой для решения вычислительной задачи.

Основная задача теории – **анализ алгоритмов с целями**:

- определения необходимых объемов **ресурсов** для решения конкретной задачи конкретным алгоритмом;
- **сравнения** нескольких алгоритмов и выбора более эффективного.

Сложность вычислений (вычислительная сложность алгоритма) = **прожорливость** алгоритма до ресурсов.

Ресурсы, расходуемые алгоритмом (вычислительные ресурсы)

Вычислительные ресурсы – возможности, обеспечиваемые компонентами вычислительной системы, расходуемые (занимаемые) в процессе её работы.

Виды вычислительных ресурсов:

- **Машинное (однопроцессорное) время** *(T)* – время работы алгоритма для решения задачи.
- **Оперативная память** *(M)* – объем памяти с произвольным доступом, необходимый алгоритму для решения поставленной задачи.
- **Долговременная память** – место на жёстком диске.
- **Пропускная способность сети** (трафик).
- **Энергия поглощаемая и выделяемая**.
- другие…

Часто удается сокращать объем потребления одного вида ресурса за счет увеличения потребления другого.

Абстрактная модель вычислений
Основные положения

1. Алгоритм рассматривается как набор **операций** и **управляющих структур.**

2. Каждому виду операций сопоставляется **временная стоимость** в абстрактных единицах времени (**шагах**).

3. Время работы алгоритма в целом равно сумме стоимостей составляющих его операций с учетом вложенности управляющих структур.

1. Алгоритм рассматривается как набор операций и управляющих структур

- **Базовые виды управляющих структур**
  - Составной оператор (begin end)
  - Условие (if then else, case)
  - Цикл (for, while, repeat)
- **Основные виды операций**
  - Логические (not, and, or, xor…)
  - Арифметические (+, -, *, /, div, mod)
  - Математические функции (sin, cos, log, exp, power..)
  - Вызов процедур и функций
  - и др.

2. Каждой операции сопоставляется временная стоимость в абстрактных единицах времени (шагах)

- **Раньше** (процессор 8088 1979 г.)

Таблица В.2. **Времена исполнения команд**

Команда	Число тактов	Число байтов
ADD регистр, регистр	3	2
ADD регистр, память	9 (13)+EA	2–4
ADD память, регистр	16 (24)+EA	2–4
MUL регистр 8	70–77	2
MUL регистр 16	118–133	2
MUL память 8	(76–83)+EA	2–4
MUL память 16	(128–143)+EA	2–4

- **Сегодня** (Intel Core 2 2006г.)

до 4-х операций за 1 такт в каждом ядре

- **Упрощенная модель вычислений** - временная стоимость всех операций считается одинаковой и равной 1 шагу.

3. Время работы алгоритма в целом равно сумме стоимостей составляющих его операций с учетом вложенности управляющих структур

№	Управляющая структура	Запись на языке Pascal	Вычислительная сложность структуры

1	Составной оператор	`begin`
		S_1;
		S_2;
		…
		`end;`
			$T = \sum_i T(S_i)$

2	Цикл	`for` ***i*** := 1 `to` ***N*** `do` ***S***; `While` ***U*** `do` ***S***; `Repeat` ***S***; `Until` ***U***;	$T = N * T(S)$, где N – число итераций цикла
3	Условие	`if` ***U*** `then` S_1 `else` S_2;	$T = T(U) + \begin{cases} T(S_1), если\ U \\ T(S_2), иначе \end{cases}$

Оценка сложности

Алгоритм $MaxElement(A[0..n-1])$

// **Входные данные:** массив вещественных чисел $A[0..n-1]$
// **Выходные данные:** возвращается значение наибольшего
// элемента массива A

$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $A[i] > maxval$
    $maxval \leftarrow A[i]$
return $maxval$

$$C(n) = \sum_{i=1}^{n-1} 1.$$

Оценка сложности

```
        g(A , n)
c1        for i=1 to n :          c1×(n+1)
c2          if A[i] > 0 :         c2×n
c3            for j=1 to i :      c3 Σ_{i=1}^{n} t_i (i+1)

c4                A[j] ← - A[j]   c4 Σ_{i=1}^{n} t_i i
```

t_i = 1 if A[i]>0 or 0 else.

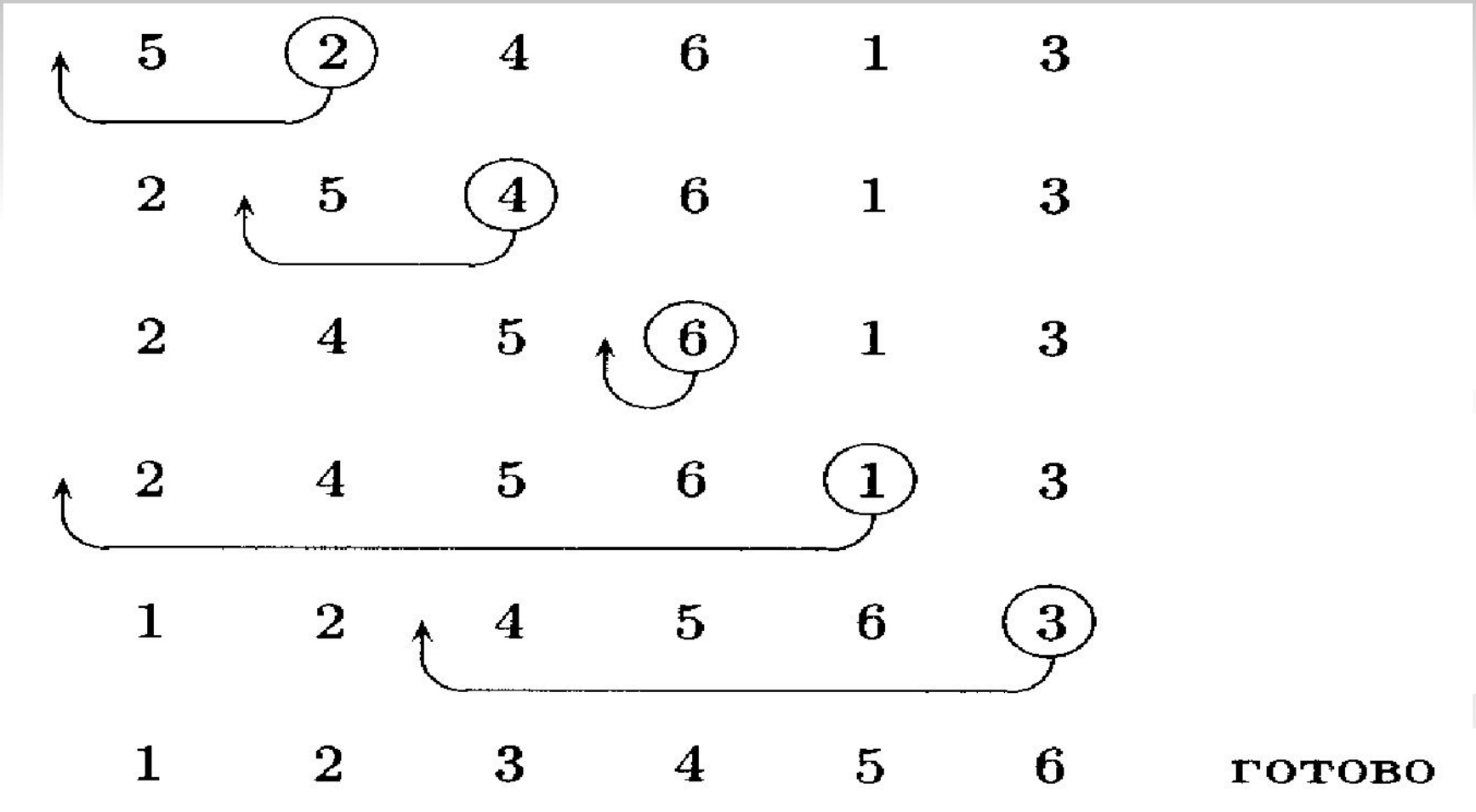
Best case: t_i =0 for all i

$$T(n)=c_1\times(n+1) + c_2\times n$$

Worst case: t_i =1 for all i

$$T(n) = c_1\times(n+1) + c_2\times n+ c_3\,((n+1)(n+2)/2 - 1)+c_4\, n\,(n+1)/2$$

Сортировка вставками



Сортировка вставками

INSERTION-SORT(A)	*стоимость*	*число раз*
1 **for** $j \leftarrow 2$ **to** $length[A]$	c_1	n
2 **do** $key \leftarrow A[j]$	c_2	$n-1$
3 ▷ добавить $A[j]$ к отсортиро-		
▷ ванной части $A[1..j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] \leftarrow key$	c_8	$n-1$

Сортировка вставками

На рис. 1.2 показана работа алгоритма при $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Индекс j указывает «очередную карту» (только что взятую со стола). Участок $A[1\,..\,j-1]$ составляют уже отсортированные карты (левая рука), а $A[j+1\,..\,n]$ — ещё не просмотренные. В цикле **for** индекс j пробегает массив слева направо. Мы берём элемент $A[j]$ (строка 2 алгоритма) и сдвигаем идущие перед ним и большие его по величине элементы (начиная с $(j-1)$-го) вправо, освобождая место для взятого элемента (строки 4–7). В строке 8 элемент $A[j]$ помещается на освобождённое место.

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

$$+ c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1).$$

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

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

Худший случай

Если же массив расположен в обратном (убывающем) порядке, время работы процедуры будет максимальным: каждый элемент $A[j]$ придётся сравнить со всеми элементами $A[1], \ldots, A[j-1]$. При этом $t_j = j$. Поскольку

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

(см. главу 3), получаем, что в худшем случае время работы процедуры равно

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

Пример анализа вычислительной сложности алгоритма

```
//Алгоритм простого последовательного поиска
      целого числа x в массиве A
function Search(A: array[1..n] of integer; x:
      integer): integer;
var i: integer;
begin
  //В цикле обход элементов массива
  for i := 1 to n do
    //Если текущий элемент равен искомому
    if A[i] = x then
    begin
      //то вернуть индекс элемента
      result := i;
      //и выйти из процедуры
      exit;
    end;
  //вернуть признак того, что элемент не найден
  result := -1;
end;
```

Пример анализа вычислительной сложности алгоритма (в упрощенной модели вычислений)

```
//Алгоритм простого последовательного поиска
      целого числа x в массиве A
function Search(A: array[1..n] of integer; x:
      integer): integer;
var i: integer;
begin //УС1
  //В цикле обход элементов массива
  for i := 1 to n do //УС2
    //Если текущий элемент равен искомому
    if A[i] = x then //УС3
    begin //УС4
      //то вернуть индекс элемента
      result := i;
      //и выйти из процедуры
      exit;
    end;
  //вернуть признак того, что элемент не найден
  result := -1;
end;
```

Вычислительная сложность управляющих структур алгоритма:

$$T_4 = 1+1 = 2$$

$$T_3 = 1+\begin{cases} T_4, если\ A[k] = x \\ 0, иначе \end{cases} = \begin{cases} 3, если\ A[k] = x \\ 1, иначе \end{cases}$$

$$T_2 = n * T_3 = \begin{cases} (k-1)*1+3, если\ A[k] = x \\ n*1, если\ x\ нет\ в\ A \end{cases} =$$

$$= \begin{cases} k+2, если\ A[k] = x \\ n, если\ x\ нет\ в\ A \end{cases}$$

$$T_{Search} = T_1 = T_2 + 1 = \begin{cases} k+2, если\ A[k] = x \\ n+1, если\ x\ нет\ в\ A \end{cases}$$

Зависимость от входных данных

Временная сложность *поиска числа в массиве* **зависит** от содержимого массива ***A***, от искомого числа ***x*** и количества элементов в массиве ***n*** :

$$T_{Search}(A, x, n) = \begin{cases} k+2, если\ A[k] = x \\ n+1, если\ x\ нет\ в\ A \end{cases}$$

Для упрощения принимается правило: временную сложность алгоритма выражать, как функцию **только размера входных данных**, НЕ зависящую от содержимого входных данных.

Размер входных данных *(N)* – **величина**, характеризующая **количество** входной информации. *(зависит от задачи)*

Упростить функцию до $T_{Search}(n)$ можно несколькими способами:

1. **Наилучший случай** – искомое число в первой ячейке $T_{Search}(n)=3$
2. **Наихудший случай** – искомое число в последней ячейке $T_{Search}(n)=n+2$
3. **Средний случай** – при равномерной распределении $T_{Search}(n)=(3+(n+2))/2$

Асимптотический анализ
вычислительной сложности алгоритмов

Асимптотический анализ – анализ поведения функции временной сложности алгоритма ***T(n)*** при $n \to \infty$ с **целью выбора ближайшей более простой** (как правило, элементарной) **функции**.

$$T_{Search}(n) = (3 + (n+2))/2 \underset{n\to\infty}{\approx} n$$

$$T_1(n) = 1000n + 200 \underset{n\to\infty}{\approx} n$$

$$T_2(n) = 100n \cdot \log_2 3n + 20 \underset{n\to\infty}{\approx} n \cdot \log_2 3n$$

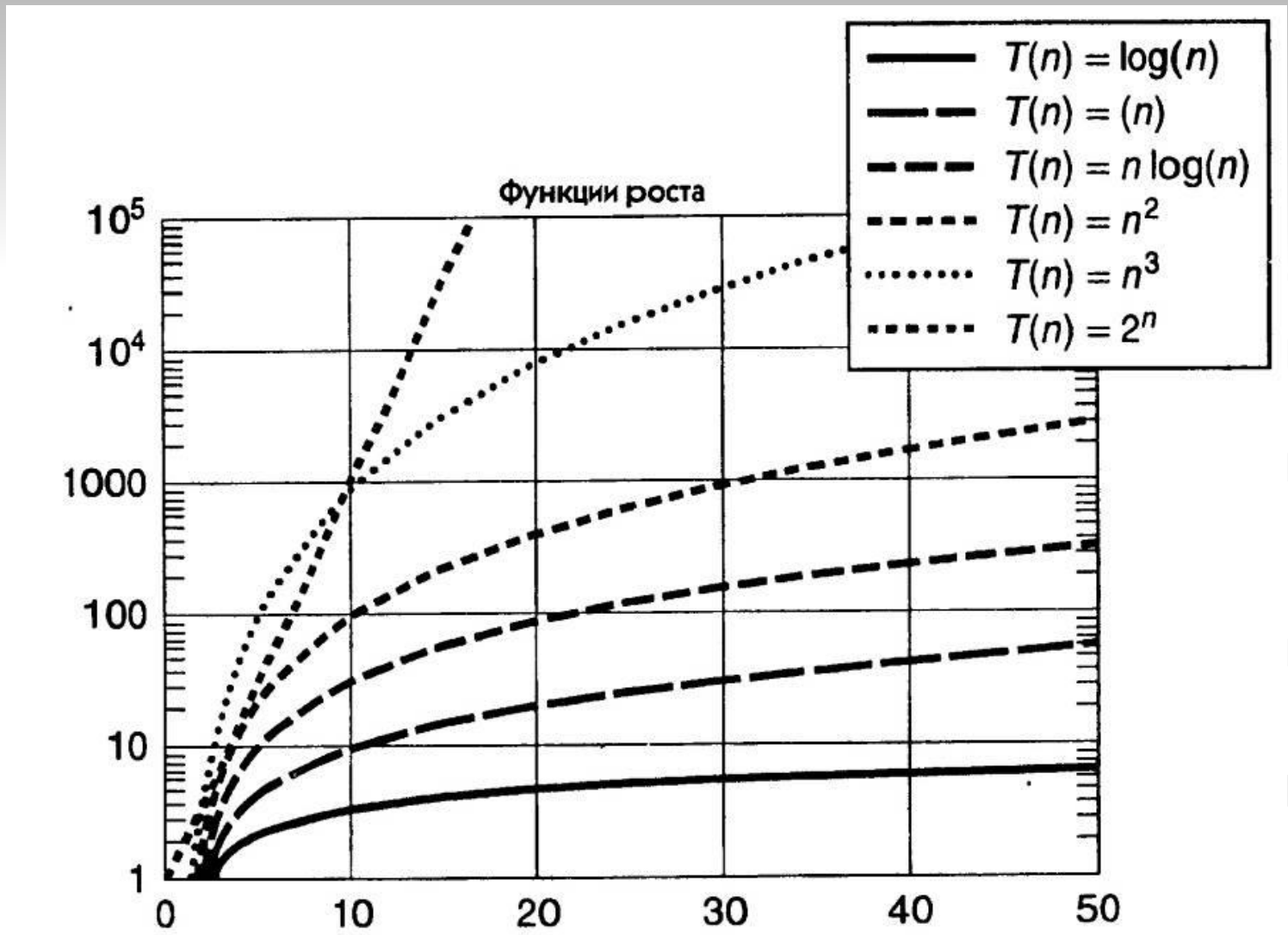
$$T_3(n) = 10n^2 + 4n + 200 \underset{n\to\infty}{\approx} n^2$$

Асимптотический анализ позволяет **оценивать** и **сравнивать скорость роста** функции временной сложности, а так же **классифицировать алгоритмы**.

Асимптотический анализ
Основные классы функции

Вид функции	Название класса функций		Примеры и комментарии
1	«Постоянное время»		Сумма двух первых чисел массива, доступ к i-му элементу массива.
$\log_2 n$ = log n	«Логарифмическое время»		Бинарный поиск элемента в отсортированном массиве.
n	«Линейное время»		Простой поиск числа в массиве, доступ к i-му элементу списка
$n \cdot \log_2 n$	«Время пропорциональное n *log n*»		Сортировка массива слиянием или быстрая сортировка в среднем случае.
n^2	«Полиномиальное время»	«Квадратичное время»	Алгоритмы, перебирающие все пары элементов исходных данных, сортировка «пузырьком» в наихудшем случае.
n^3		«Кубическое время»	Алгоритмы рассматривающие все тройки элементов исходных данных.
…		…	…
2^n	«Экспоненциальное время»		Алгоритмы, перебирающие все подмножества набора входных данных.
n!	«Факториальное время»		Алгоритмы, перебирающие все комбинации элементов набора входных данных.

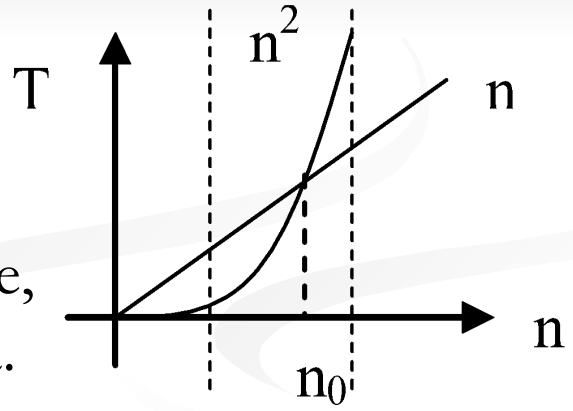
Асимптотический анализ
Графики основных классов функций



Асимптотический анализ
Область действия

- **!** Асимптотический анализ справедлив только для больших ***n.***

Для малых ***n*** бывают случаи, когда алгоритм, относящийся к более эффективному классу, работает медленнее, чем алгоритм менее эффективного класса.



Пример, метод сортировки «**пузырьком**» при малых **n** работает **быстрее** чем «**быстрая**» сортировка.

Асимптотический анализ

Цель асимптотического анализа - сравнение затрат ресурсов системы различными алгоритмами, предназначенными для решения

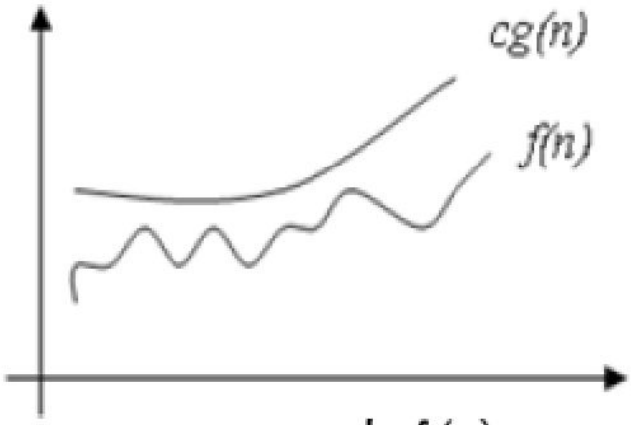
- одной и той же задачи
- при больших объемах входных данных.

Используемая в асимптотическом анализе **оценка функции трудоёмкости, называется сложностью алгоритма и** позволяет определить, как быстро растет трудоёмкость алгоритма с увеличением объема данных.

В асимптотическом анализе используются обозначения позволяющие показать скорость роста функции :

1. **Оценка Θ (тетта)**
2. **Оценка O (О большое)**
3. **Оценка Ω (Омега)**

Оценка O (O большое)



$$O(g(n)) = \left\{ \begin{array}{l} f(n): \text{ существуют положительные константы } c \text{ и } n_0 \\ \text{такие что } 0 \leqslant f(n) \leqslant cg(n) \text{ для всех } n \geqslant n_0 \end{array} \right\}$$

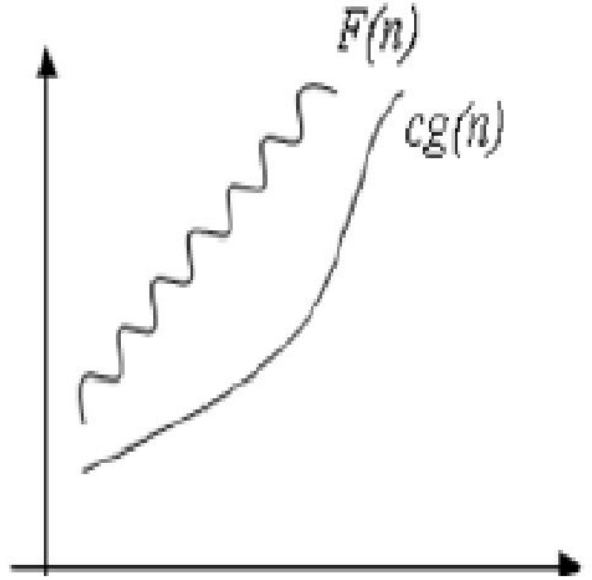
Запись O(g(n)) обозначает класс таких функций которые растут не быстрее, чем функция g(n) с точностью до постоянного множителя, поэтому иногда говорят, что g(n) **мажорирует** функцию f(n).

Например, для всех функций:

f(n)=1/n, f(n)= 12, f(n)=3n+17, f(n)=nLn(n), $f(n)=6n^2+24n+77$

будет справедлива оценка $O(n^2)$

3. Оценка Ω (Омега)



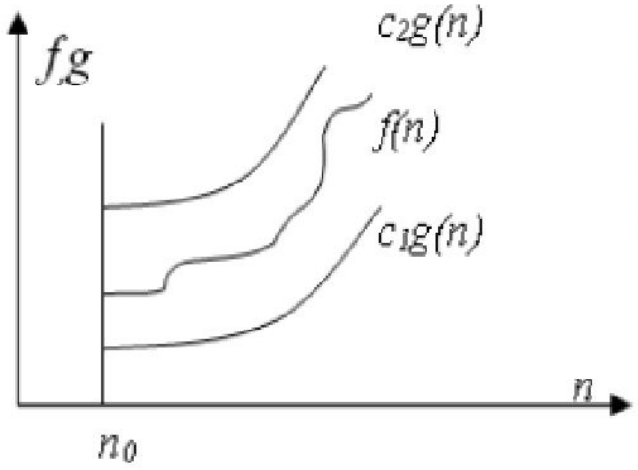
Оценка **Ω** является оценкой снизу – т.е. определяет класс функций, которые растут не медленнее, чем g(n) с точностью до постоянного множителя:

$$\Omega(g(n)) = \left\{ \begin{array}{l} f(n): \text{ существуют положительные константы } c \text{ и } n_0 \\ \text{такие что } 0 \leqslant cg(n) \leqslant f(n) \text{ для всех } n \geqslant n_0 \end{array} \right\}$$

Оценка Θ (тета)

Пусть f(n) и g(n) – положительные функции положительного аргумента, n >= 1 , тогда:

$$\Theta(g(n)) = \left\{ \begin{array}{l} f(n): \text{ существуют положительные константы } c_1, c_2 \text{ и } n_0 \\ \text{такие что } 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n) \text{ для всех } n \geqslant n_0 \end{array} \right\}$$



Обычно говорят, что при этом функция g(n) является асимптотически точной оценкой функции f(n), т.к. по определению функция f(n) не отличается от функции g(n) с точностью до постоянного множителя.

Примеры:

1) $f(n)=4n^2+n\ln(n)+174$ $f(n)=\Theta(n^2)$;

2) $f(n)=\Theta(1)$ – запись означает, что f(n) или равна константе, не равной нулю, или f(n) ограничена константой на ∞ :

$f(n) = 7+1/n = \Theta(1)$.

Теорема Для любых двух функций $f(n)$ и $g(n)$ соотношение $f(n) = \Theta(g(n))$ выполняется тогда и только тогда, когда $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$. □

Например, запись Ω(nLn(n)) обозначает класс функций, которые растут не медленнее, чем g(n) = nLn(n), в этот класс попадают все полиномы со степенью n>2 и все степенные функции с основанием большим единицы.

Не всегда для пары функций справедливо одно из асимптотических соотношений, например для $\mathbf{f(n)=n^{1+\sin(n)}}$ и **g(n)=n** не выполняется ни одно из асимптотических соотношений.

Аналогии

$$
\begin{aligned}
f(n) = O(g(n)) &\approx a \leqslant b, \\
f(n) = \Omega(g(n)) &\approx a \geqslant b, \\
f(n) = \Theta(g(n)) &\approx a = b, \\
f(n) = o(g(n)) &\approx a < b, \\
f(n) = \omega(g(n)) &\approx a > b.
\end{aligned}
$$

Примеры

Пример

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

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

Неравенства выполняются если выбрать:

$n>=7$ $c_1<=1/14$

$n\to\infty$ $c_2>=1/2$

Таким образом

$(2/7)n^2<=n^2/2-3n<=(1/2)n^2$

Пример $c_1 n^2<=a n^2+bn+c<= c_2 n^2$, $a,b,c>0$

$a n^2 <= a n^2+bn+c<= (a+b+c) n^2$

Для произвольных a,b,c

$C_1=\min\{a, a+b+c\}$, $C_2=\max\{a, a+b+c\}$

Общий план анализа эффективности нерекурсивных алгоритмов

1. Выберите параметр (или параметры), по которому будет оцениваться размер входных данных алгоритма.
2. Определите основную операцию алгоритма. (Как правило, она находится в наиболее глубоко вложенном внутреннем цикле алгоритма.)
3. Проверьте, зависит ли число выполняемых основных операций только от размера входных данных. Если оно зависит и от других факторов, рассмотрите при необходимости, как меняется эффективность алгоритма для наихудшего, среднего и наилучшего случаев.
4. Запишите сумму, выражающую количество выполняемых основных операций алгоритма[4].
5. Используя стандартные формулы и правила суммирования, упростите полученную формулу для количества основных операций алгоритма. Если это невозможно, определите хотя бы их порядок роста.

Основные формулы суммирования

будем использовать два основных правила суммирования:

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$$

и две формулы суммирования:

$$\sum_{i=l}^{u} 1 = u - l + 1,$$ где $l \leqslant u$ — целые числа, представляющие нижнюю и верхнюю границы суммы

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta\left(n^2\right)$$

АЛГОРИТМ $MatrixMultiplication\,(A\,[0..n-1, 0..n-1], B\,[0..n-1, 0..n-1])$

// Выполняется умножение двух квадратных матриц размером
// $n \times n$. Используется алгоритм, основанный на определении
// этой операции
// **Входные данные:** две квадратные $n \times n$ матрицы A и B
// **Выходные данные:** матрица $C = AB$

for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $C[i, j] \leftarrow 0.0$
    for $k \leftarrow 0$ **to** $n-1$ **do**
      $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

Общий план анализа эффективности рекурсивных алгоритмов

1. Выберите параметр (или параметры), по которому будет оцениваться размер входных данных алгоритма.
2. Определите основную операцию алгоритма.
3. Проверьте, зависит ли число выполняемых основных операций только от размера входных данных. Если оно зависит и от других факторов, рассмотрите при необходимости, как меняется эффективность алгоритма для наихудшего, среднего и наилучшего случаев.
4. Составьте рекуррентное уравнение, выражающее количество выполняемых основных операций алгоритма, и укажите соответствующие начальные условия.
5. Найдите решение рекуррентного уравнения или, если это невозможно, определите хотя бы его порядок роста.

Пример

Алгоритм $F(n)$

// Рекурсивное вычисление факториала

// **Входные данные:** Целое неотрицательное число n

// **Выходные данные:** Значение $n!$

if $n = 0$

  return 1

else

  return $F(n-1) * n$

$$M(n) = M(n-1) + 1 \quad \text{для } n > 0,$$

$$M(0) = 0.$$

Метод итераций

$$
\begin{aligned}
M(n) &= M(n-1) + 1 = \\
&= [M(n-2) + 1] + 1 = \\
&= M(n-2) + 2 = \\
&= [M(n-3) + 1] + 2 = \\
&= M(n-3) + 3.
\end{aligned}
$$

$$M(n) = M(n-i) + i.$$

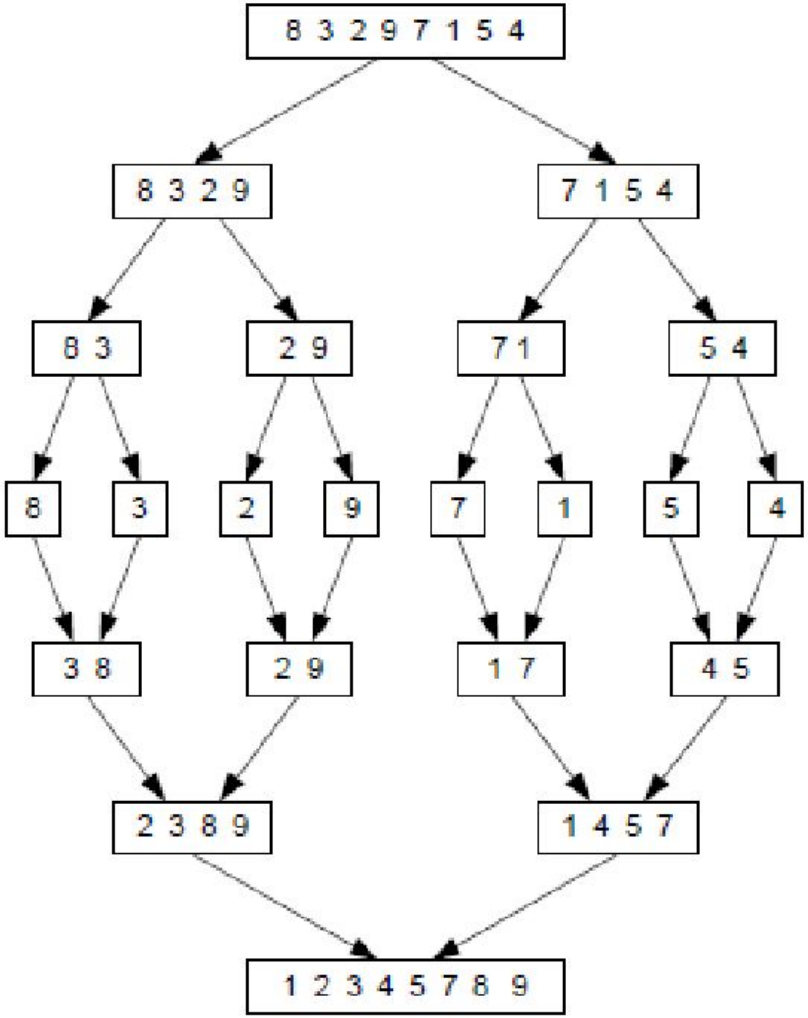
$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$

Сортировка слиянием

- Разделите массив A[0..$n-1$] на две примерно одинаковые части и скопируйте каждую часть в массивы B и C. Отсортируйте B и C тем же методом.

- Слейте отсортированные массивы B и C в массив A, сравнивая первые элементы в оставшихся частях B и C и отсылая в A наименьший из них. Когда в одном из этих массивов не остается элементов, отошлите в A элементы, оставшиеся в другом массиве.

Пример сортировки слиянием



Сортировка слиянием

```
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)
```

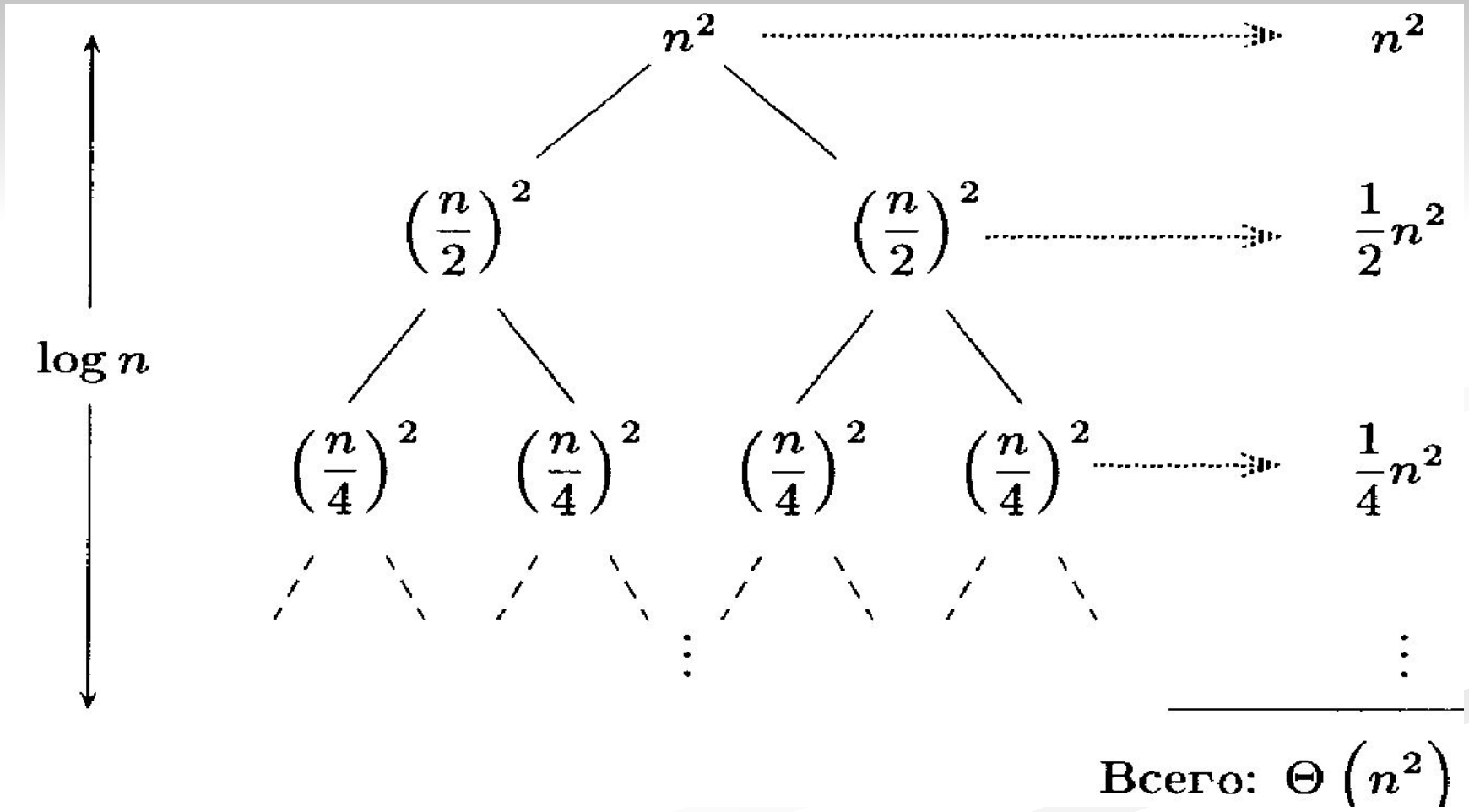
Анализ

Для простоты будем предполагать, что размер массива есть степень двойки. (Как мы увидим в главе 4, это ограничение не очень существенно.) Тогда на каждом шаге сортируемый участок делится на две равные половины. Разбиение на части (вычисление границы) требует времени $\Theta(1)$, а слияние — времени $\Theta(n)$. Получаем соотношение

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

Как мы увидим в главе 4, это соотношение влечёт $T(n) = \Theta(n \log n)$, где через

Дерево рекурсии для соотношения $T(n) = 2T(n/2) + n^2$.



Основная теорема

Теорема 5 (Основная теорема). Пусть $T(n)$ — в конечном счете неубывающая функция, удовлетворяющая рекуррентному соотношению

$$T(n) = aT(n/b) + f(n) \text{ при } n = b^k, k = 1, 2, \ldots; T(1) = c,$$

где $a \geqslant 1$, $b \geqslant 2$, $c > 0$. Если $f(n) \in \Theta\left(n^d\right)$, где $d \geqslant 0$, то

$$T(n) \in \begin{cases} \Theta\left(n^d\right) & \text{если } a < b^d, \\ \Theta\left(n^d \log n\right) & \text{если } a = b^d, \\ \Theta\left(n^{\log_b a}\right) & \text{если } a > b^d. \end{cases}$$

(Аналогичные результаты справедливы для O и Ω обозначений.)