

Теорема о сложности сортировки

Изученные ранее методы сортировки имели

порядок сложности:

$$O(n^2) \rightarrow O(n^{1,2}) \rightarrow O(n \log_2 n)$$

Вопрос:

До какого предела

МОЖНО СНИЖАТЬ

трудоемкость сортировки?

Теорема о сложности сортировки

Теорема.

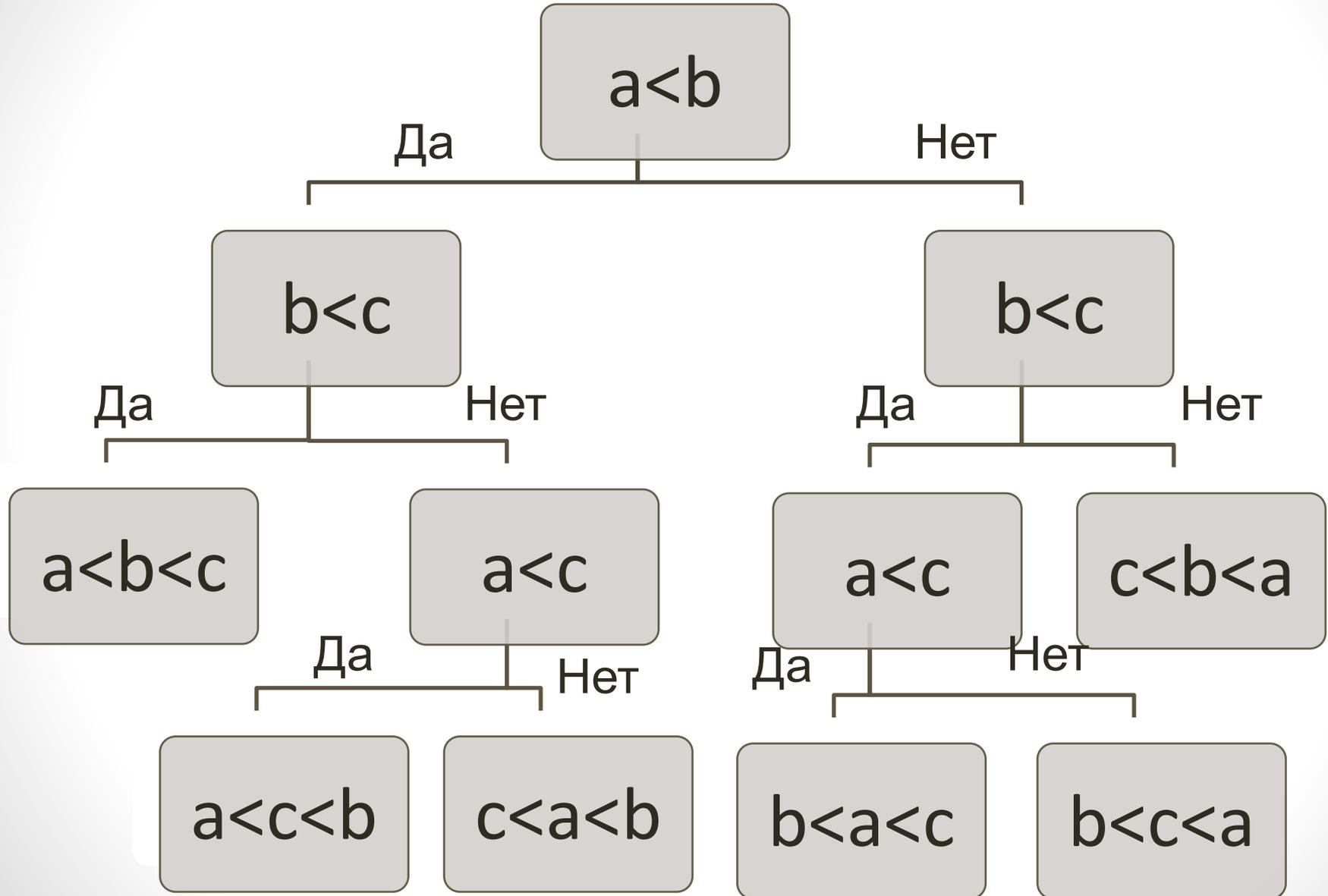
Если все перестановки из n элементов равновероятны, то любое **дерево решений**, сортирующее последовательность из n элементов, имеет **среднюю высоту** не менее

$$\log_2(n!).$$

Приведем нестрогое доказательство.

Рассмотрим **дерево решений**

для сортировки **трех** элементов **a, b, c**.



Теорема о сложности сортировки

Листья дерева - это все возможные перестановки элементов a, b, c .

Для того, чтобы узнать, какая перестановка нужна для упорядочения элементов a, b, c , достаточно сделать в некоторых случаях **2** сравнения, в других - **3** сравнения:

$$C_{\text{ср}} = \frac{2+3+3+3+3+2}{6} = \frac{\text{длина внешнего пути дерева}}{2*3 = n!}$$

Теорема о сложности

СОРТИРОВКИ

- $$C_{\text{ср}} = \frac{2+3+3+3+3+2}{6} = \frac{\text{длина внешнего пути дерева}}{2*3 = n!}$$

Из теории графов известно, что

длина внешнего пути двоичного дерева с m листьями

$$D(m) \geq m \log_2 m .$$

В нашем дереве $n!$ листьев, поэтому

$$C_{\text{ср}} \geq \frac{n! \log_2 n!}{n!}$$

$$C_{\text{ср}} \geq \log_2 n!$$

Используем **нижнюю оценку** для $\log_2 n!$:

$$\log_2 n! > n \log_2 n - n \log_2 e$$

$$C_{\text{ср}} \geq n \log_2 n - n \log_2 e$$

Получаем **следствие из теоремы**:

Не существует алгоритма сортировки n элементов, использующего в среднем менее $n \log_2 n - n \log_2 e$ операций сравнения.

Класс сложности порядка $n \log_2 n$ является

предельно достижимым для алгоритмов,

основанных на **операциях сравнения**.

Для пересылок:

если мы определили требуемую перестановку и имеем память для второй копии массива, то достаточно сделать n пересылок.

На сегодняшний день

алгоритм, имеющий

$n \log_2 n$ сравнений и n пересылок,

неизвестен.

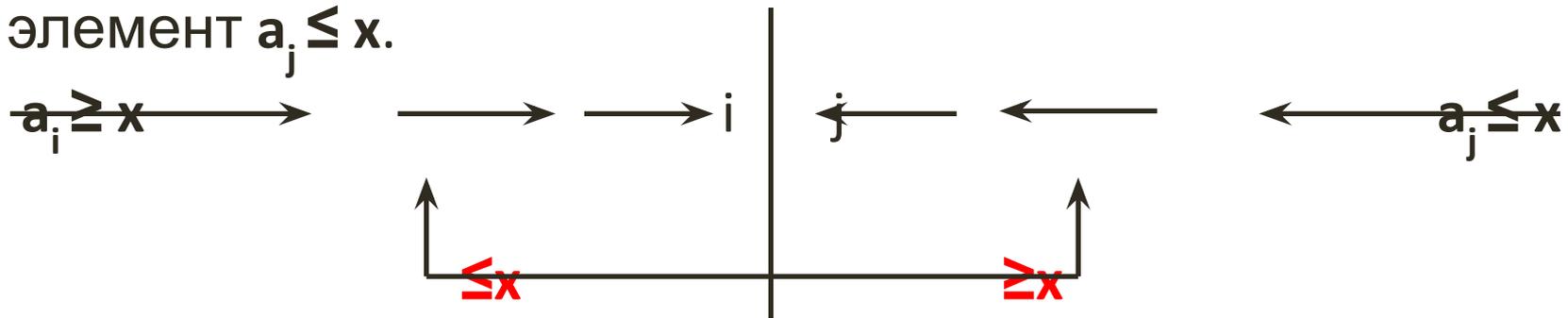
Рассмотрим метод сортировки,

самый быстрый из известных.

Метод Хоара (Hoare, 1962)

QuickSort

Возьмем произвольный элемент массива, обозначим его через X . Просматривая массив **слева**, найдем элемент $a_i \geq x$. Просматривая массив **справа**, найдем элемент $a_j \leq x$.



Поменяем местами a_i и a_j . Будем **продолжать процесс просмотра и обмена**, пока i не станет больше j .

В результате массив окажется разделенным на две части: **левую с элементами $\leq x$** , и **правую с элементами $\geq x$** .

Метод Хоара (QuickSort)

Алгоритм на псевдокоде

L, R – левая и правая границы рабочей части массива

QuickSort (L, R)

$x := a_L, i := L, j := R$

DO ($i \leq j$)

DO ($a_i < x$) $i := i+1$ OD

DO ($a_j > x$) $j := j-1$ OD

IF ($i <= j$) $a_i \leftrightarrow a_j, i := i+1, j := j-1$ FI

OD

IF ($L < j$) QuickSort (L, j) FI

IF ($i < R$) QuickSort (i, R) FI

$A \bar{A} \underline{\underline{B}} | \dot{E}$

$\textcircled{\bar{A}} \underline{A} \underline{B}$
↑ ↑

$A \dot{A} B$
↓

$A \dot{B} \underline{B} | \dot{E}$
↑ ↑

⓪̄ O P Y_K

K O | Ṗ_Y Π

• ⓪̄ | Ȯ Ⓟ̄_Y Π

Π | Ẏ P

⓪̄ P

P | Ẏ

A A B E K O Π P Y

Трудоёмкость алгоритма QuickSort существенно зависит от того, как соотносится выбираемый элемент X с остальными элементами в массиве.

Рассмотрим два крайних случая:

1) $X = \min(\max) (a_L, \dots, a_R)$

В этом случае после разделения в одной части массива будет оставаться только **один элемент**, а в другой части - **все остальные**.

$$C_1 = (n+2) + (n+1) + \dots + 2 = \frac{n^2 + 5n + 4}{2}$$

$$M_1 = 3(n-1)$$

Трудоёмкость в этом случае сравнима с методом прямого выбора: $O(n^2)$, $n \rightarrow \infty$

2) $X = \text{med}(a_L, \dots, a_R)$ – медиана

Определение.

Элемент a_m называется **медианой** для элементов a_L, \dots, a_R , если **количество элементов меньших a_m** равно количеству элементов **больших a_m** (с точностью до одного элемента).

В примере буква К – медиана для КУРАПОВАЕ.

В этом случае массив разделяется **на две равные части.**

Определим **наименьшее возможное количество сравнений.**

Возьмем для примера $n = 15$ (степень двойки), затем массив делится на 7 и 7, затем на 3,3,3,3.

Пример. $\overline{X} \overline{X} \overline{X} \overline{X} \overline{X} \overline{X} \overline{X}$

Размер части массива		Количество сравнений в части массива		Количество частей	Количество сравнений
15		16		1	
7		8		2	
3		4		4	

$$n = 15 = 2^4 - 1$$

$$n = 2^k - 1 \Rightarrow k = \log_2(n + 1)$$

Из таблицы $k = 4$, $3 = k - 1$ - количество итераций.

$$C = (k-1)2^k = (\log(n+1) - 1) 2^k = (n+1) \log(n+1) - (n+1)$$

– близкое к минимально возможному значению.

Итак, количество сравнений $C = n \log n$

Количество пересылок

зависит от расположения элементов,

но **не может быть больше одного обмена** (3 пересылки) **на два сравнения.**

Поэтому количество пересылок –

величина того же порядка, что и количество сравнений:

$$M = n \log n.$$

Средняя трудоемкость QuickSort: $O(n \log n)$, $n \rightarrow \infty$

Проблема глубины рекурсии

В теле подпрограммы доступны все объекты, описанные в основной программе, в том числе и имя самой подпрограммы. Это позволяет внутри тела подпрограммы осуществлять вызов самой подпрограммы.

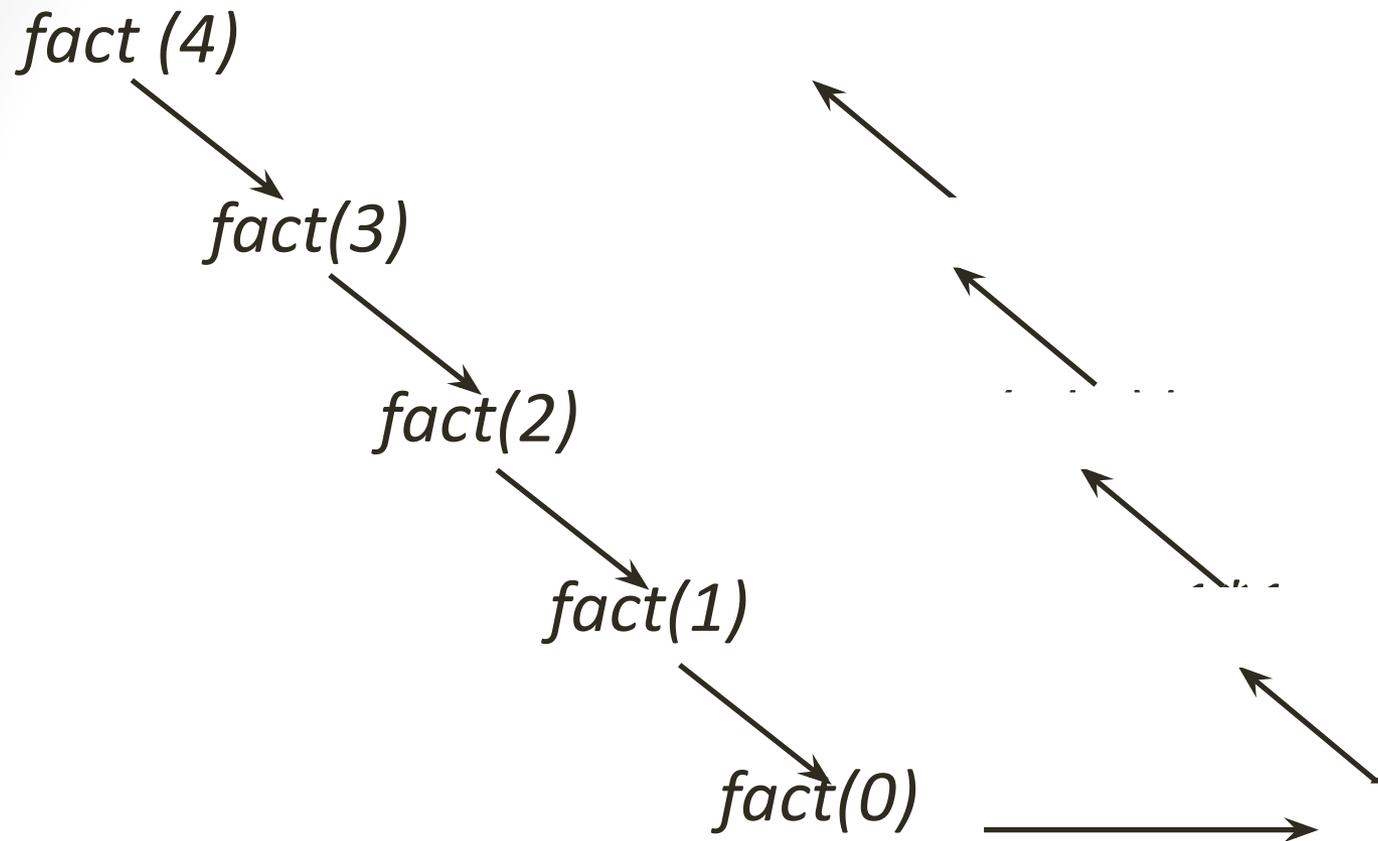
Определение. Процедуры и функции, организующие вызовы «самих себя» называются *рекурсивными*.

Многие математические алгоритмы используют рекурсию, поэтому рекурсия широко используется в программировании.

Пример: Известный **алгоритм вычисления факториала** неотрицательного целого числа: $0! = 1$, $1! = 1$, $n! = (n-1)! n$.

```
long fact (int n) {  
    if (n<0) return 0;  
    if (n==0) return 1;  
    return fact (n-1);  
}
```

Вычисление 4!



Рекурсивное выполнение программы более компактно и наглядно, но существует опасность переполнения стека.

Фрейм:

Фактические параметры
Адреса возврата
Регистры процессора
Локальные переменные

При выходе из подпрограммы эта память (фрейм) освобождается.

Но если подпрограмма вызывает другую подпрограмму или саму себя, то в дополнение к существующему фрейму создается новый.

n вложенных вызовов ->

выделение n фреймов в памяти

Quick Sort(L,R) (Первая версия)

<Разделение на две части>

IF (L < j) QuickSort (L, j) FI

IF (i < R) QuickSort (i, R) FI

Рассмотренный алгоритм может потребовать

в худшем случае n вложенных вызовов

(n - размер массива),

т.е. **глубина рекурсии может достигать n .**

Это большой **недостаток алгоритма.**

Покажем, как можно **уменьшить глубину рекурсии**
до $\log_2 n$.

Вместо **двух** рекурсивных вызовов

(для левой и правой части массива)

будем использовать только **один рекурсивный вызов**,

а другой заменим **новой итерацией цикла**.

Вопрос:

Для **какой** части массива нужно делать рекурсивный вызов,

чтобы **уменьшить глубину рекурсии**?

Ответ:

Для меньшей по размеру части массива.

!-----!-----!

L

J I

R

Алгоритм на псевдокоде

QuickSort(L,R) (вторая версия)

DO (L < R)

<Разделение на две части>

IF (j - L < R - i) QuickSort(L, j), L := i

ELSE QuickSort(i, R), R := j

FI

OD

В этом случае худший вариант,

когда обе части массива равны,

тогда **глубина рекурсии** $\log_2 n$.

Пример:

для массива из **1 млн.** элементов ($n = 1000000$)

понадобится одновременно менее **20** фреймов в памяти.

Примеры рекурсии

1. Преподаватель всегда прав.
2. Если преподаватель не прав, смотри **пункт 1**.

Бюрократия разрастается, чтобы удовлетворить нужды разрастающейся бюрократии.

Смысл жизни – в достижении её цели, цель жизни – в наполнении ее смыслом.

Если у вас украли кредитную карту, позвоните по телефону указанному на оборотной стороне кредитной карты.

Для выхода **в Интернет**, скачайте нашу программу **из Интернета**.

Keyboard not found. Press F1 to continue.

Салат : помидоры, огурцы, **салат**.

Метод	Трудоемкость	Устойчивость	Зависимость от упорядоченности
ShellSort	$O(n^{1,2})$	Не устойчив	Зависит
HeapSort	$O(n \log_2 n)$	Не устойчив	Практически не зависит
QuickSort	$O(n \log_2 n)$	Не устойчив	Зависит