Лекция 2

Разделяй и властвуй. Основной метод

Теорема и примеры. Быстрая сортировка.

1. MergeSort (A)

Сортировка слиянием массива A из n чисел

MergeSort (A[1..n])

Вход: массив A[1..n]

<u>Выход: отсортированный массив из тех же чисел</u>

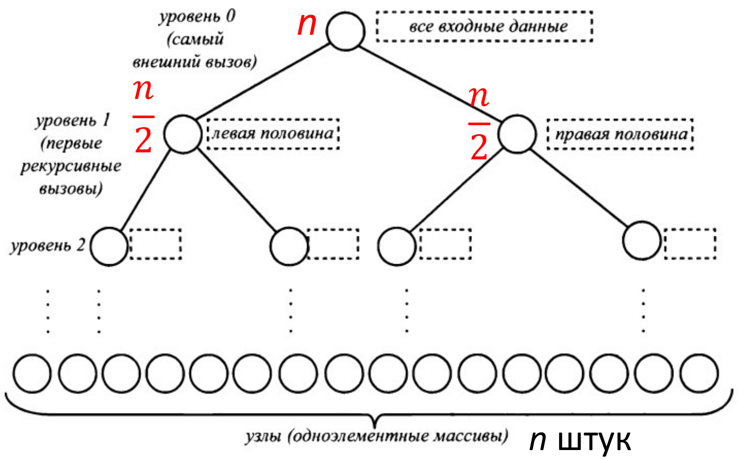
IF n>1

 THEN

  RETURN Merge (MergeSort (A[1..$\left\lceil \frac{n}{2} \right\rceil$]), MergeSort (A[$\frac{n}{2}$]+1..])

 ELSE // базовый случай

  RETURN A



T(n) - общее время работы алгоритма а n-элементном массиве A.

2 рекурсивных вызова на каждом уровне. Общее число операций на каждом уровне (не включая рекурсивные вызовы) – O(n).

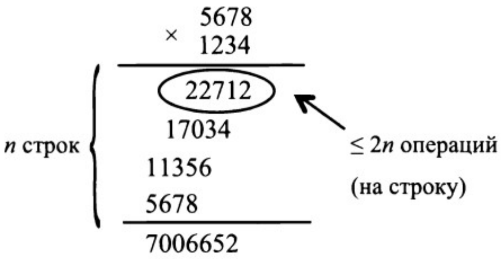
Размеры двух подмассивов

Работа вне рекурсивных вызовов

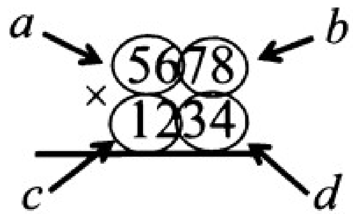
Имеем $T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + O(n)$

2 вызова

2. Умножение чисел $x \cdot y$



$O(n^2)$



$$x \times y = (10^{n/2} \times a + b) \times (10^{n/2} \times c + d) =$$
$$= 10^n \times (a \times c) + 10^{n/2} \times (a \times d + b \times c) + b \times d.$$

RECINTMULT (x, y)

Вход: два n-значных положительных целых числа, x и y.

Выход: произведение $x \times y$.

Допущение: n является степенью числа 2.

```
if n = 1 then                          // базовый случай
вычислить x × y за один шаг и выдать результат
else                                   // рекурсивный случай
a, b := первая и вторая половины x
c, d := первая и вторая половины y
рекурсивно вычислить ac := a × c, ad := a × d,
bc := b × c и bd := b × d
вычислить 10^n × ac + 10^(n/2) × (ad + bc) + bd,
используя арифметическое сложение, и выдать результат.
```

Все умножения происходят либо между парами *n/2* -значных чисел, либо связаны с возведением в степень

Умножение двух *n*-значных чисел сводится к умножению 4-х пар *n/2* -значных чисел плюс *O(n)* дополнительная работа (для добавления соответствующих нулей и школьного сложения)

Имеем $T(n) \leq 4 \cdot T\left(\frac{n}{2}\right) + O(n)$

4 вызова

3. Умножение Карацубы

$$x \times y = (10^{n/2} \times a + b) \times (10^{n/2} \times c + d) =$$
$$= 10^{n} \times (a \times c) + 10^{n/2} \times (a \times d + b \times c) + b \times d.$$

Считаем как раньше ac и bd. Но $ad + bc$ считаем иначе:

$ad + bc = (a + b)(c + d) - ac - bd$. (Всего 3 умножения, а не 4).

Имеем $T(n) \le 3 \cdot T\left(\frac{n}{2}\right) + O(n)$

Стандартное рекуррентное соотношение

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + O(n^d) \qquad (1)$$

Параметры:

- a – количество рекурсивных
- b – коэффициент сжатия размера входных данных
- d – экспонента во времени работы «шага объединения»

Теорема(основной метод):

Если $T(n)$ определяется стандартным рекуррентным соотношением (1) с параметрами $a > 1, b > 1, d \geq 0$, то

$$T(n) = \begin{cases} O(n^d \log_b n), \text{если } a = b^d & (\text{случай } 1) \\ O(n^d), \quad \text{если } a < b^d & (\text{случай } 2) \\ O(n^{\log_b a}) \quad \text{если } a > b^d & (\text{случай } 3) \end{cases}$$

Примеры

Для **MergeSort** $a = 2 = b = 2^1 = b^d \implies O(n^1 \log_2 n)$ (случай 1)

Для **RecintMult** $a = 4 > b = 2^1 = b^d \implies O\left(n^{\log_2 4}\right) = O(n^2)$ (случай 3)

Для **Карацубы** $a = 3 > b = 2^1 = b^d \implies O\left(n^{\log_2 3}\right)$ (случай 3)

4. Умножение квадратных матриц

а) Простое умножение матриц

Вход: целочисленные матрицы X, Y пор. n

Выход: $Z = X \times Y$

FOR *i*:=1 TO *n* DO

  FOR *j*:=1 TO *n* DO

    Z[*i*, *j*]:=0

      FOR *k*:=1 TO *n* DO

        Z[*i*, *j*]:= *Z*[*i*, *j*]+ *X*[*i*,*k*]· *Y*[*k*, *j*]

RETURN *Z*

Сложность $\boldsymbol{O}(\boldsymbol{n}^{3})$

б) Подход «Разделяй и властвуй»

Вход: целочисленные матрицы X, Y пор. n

Выход: $Z = X \times Y$

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \qquad Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

$$X \times Y = \begin{pmatrix} A \times E + B \times G & A \times F + B \times H \\ C \times E + D \times G & C \times F + D \times H \end{pmatrix}$$

8 умножений. $a = 8, b = 2, d = 2\ (a > b^{d}) \Rightarrow$

(случай 3) $O\left(n^{\log_b a}\right) = O\left(n^{\log_2 8}\right) = \boldsymbol{O}(\boldsymbol{n}^{3})$

в) Алгоритм Штрассена

Идея: сделать не 8, а 7 рекурсивных вызовов, вычислив матрицы:

$$
\begin{cases}
P_1 = A \times (F - H) \\
P_2 = (A + B) \times H \\
P_3 = (C + D) \times E \\
P_4 = D \times (G - E) \\
P_5 = (A + D) \times (E + H) \\
P_6 = (B - D) \times (G + H) \\
P_7 = (A - C) \times (E + F).
\end{cases}
$$

$$
X \times Y = \left(\begin{array}{c|c} A \times E + B \times G & A \times F + B \times H \\ \hline C \times E + D \times G & C \times F + D \times H \end{array}\right)
$$

$$
= \left(\begin{array}{c|c} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ \hline P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{array}\right)
$$

8 умножений. $a = 7, b = 2, d = 2$ (Случай 3)

$7 > 2^2 \Rightarrow T(n) = O\left(n^{\log_2 7}\right) \cong O(n^{2,81})$

STRASSEN (КРАЙНЕ ВЫСОКОУРОВНЕВОЕ ОПИСАНИЕ)

Вход: целочисленные матрицы размером $n \times n$, **X** и **Y**.

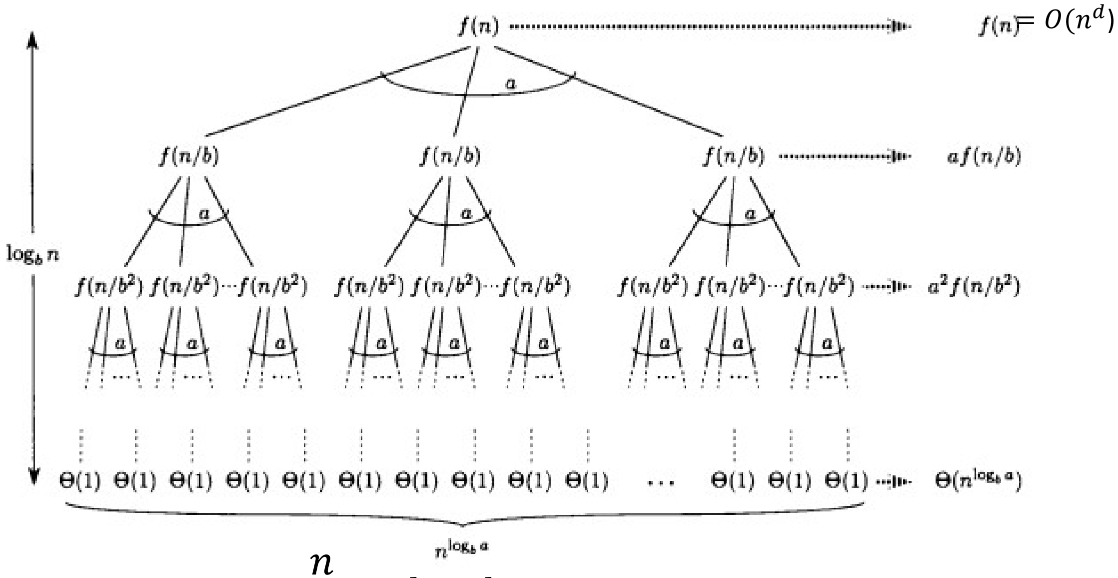
Выход: $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$.

Допущение: n является степенью числа 2.

```
if n = 1 then                              // базовый случай
   return 1×1 – матрица с элементом X[1][1] × Y[1][1]
else                                       // рекурсивный случай
   A, B, C, D := подматрицы X, как в (3.3)
   E, F, G, H := подматрицы Y, как в (3.3)
   Рекурсивно вычислить семь (предварительно отобранных)
   произведений
   С участием A, B, . . . , H
   return соответствующие (предварительно отобранные)
   сложения и вычитания матриц, вычисленных на предыдущем
   шаге
```

Доказательство теоремы (основной метод)

Будем считать, что *n* есть степень *b*.



$$1 = \frac{n}{b^k} \Rightarrow k = \log_b n$$

С увеличением уровня
размер подзадач уменьшается в *b* раз.
a – коэффициент ветвления

Уровень *j* содержит a^j подзадач размером $\frac{n}{b^j}$
Количество операций, которое производит алгоритм на уровне *j* (без рекурсивных вызовов)

$$t_j = a^j \cdot O\left(\frac{n}{b^j}\right)^d = O(n^d) \cdot \left(\frac{a}{b^d}\right)^j$$

$q = \frac{a}{b^j}$ – знаменатель геометрической прогрессии

$$T(n) = \sum_{j=0}^{k} t_j$$

Продолжение доказательства

$$t_j = a^j \cdot O\left(\frac{n}{b^j}\right)^d = O\left(n^d \cdot \left(\frac{a}{b^d}\right)^j\right)$$

$$T(n) = \sum_{j=0}^{k} t_j$$

Имеем 3 варианта $q = \frac{a}{b^d} \lesseqgtr 1$

Случай 1. $(q = 1)$ $T(n) = O(n^d \log_b n)$

Случай 2. $(q < 1)$ $T(n) = O(n^d)$, т.к. $\sum_{j=0}^{k} q^j < \frac{1}{1-q} = O(1)$

Случай 3. $(q > 1)$ $T(n) = O\left(n^d \left(\frac{a}{b^d}\right)^{\log_b n}\right) = O\left(n^{\log_b a}\right)$, т. к. $\sum_{j=0}^{k} q^j = O(q^{\log_b n})$ (старшая степень) и

$a^{\log_b n} = n^{\log_b a}, b^{d \log_b n} = n^d$

a - темп разрастания подзадач

b^d - темп сжатия работы

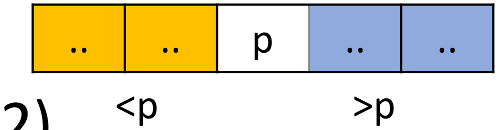
«Перетягивание каната»

Ч.т.д.

Быстрая сортировка (QuickSort)

1. Выбираем опорный элемент *p.* (Подпрограмма ChoosePivot(A).
2. Перегруппировываем элементы так, чтобы

   (подпрограмма Partition, см. д.з. к семинару 2)



3. Делаем это рекурсивно, пока длина массива станет не больше 1.

Основной метод применять нельзя!

Если в качестве опорного каждый раз выбирать медиану, то время работы $O(n\log_2 n)$.

Выбирать опорный элемент будем случайно. При равномерном распределении *p* попадает в диапазон $p > 25\%$ элементов и $p < 75\%$,

т.е. в 50% случаев находим приближенную медиану (с вероятностью P=1/2). Матожидание попадания в этот интервал (биномиальное распределение) равно $\frac{1}{P} = 2$. Значит, в среднем после 2 разбиений массива его размер уменьшится хотя бы на четверть. Имеем для выбора приближенной медианы

$$T_{med}(n) = T_{med}\left(\frac{3}{4}n\right) + O(n) \quad \text{(в среднем)} \qquad (1)$$

Время работы алгоритма QuickSort

от $O(n^2)$ (худший случай) до $O(n\log_2 n)$ (лучший медианный случай).

Медианный случай совпадает с оценкой в среднем, т.к. (интуитивное понимание)

в (1) $a = 1, b = \frac{4}{3}, d = 1 \Rightarrow T_{med}(n) = O(n)$.

Поэтому в среднем $T(n) = O(n\log_2 n)$

Замечание: это не строгое доказательство!

Примеры ситуаций, когда быстрая сортировка не эффективна:

- если она применяется для сортировки уже отсортированных массивов;
- если после обменов получается, что один из подмассивов для нового рекурсивного вызова состоит из $n-1$ одного элемента,

а другой из `1` элемента;

- если количество элементов мало `(n < 32)`.

Рекурсивные вызовы являются затратными операциями, и для небольших массивов их количество будет близким к количеству элементов. Поэтому в случае небольших объемов данных используют нерекурсивные методы сортировки, например сортировку вставками или выбором.

Для улучшения эффективности иногда используется метод «медиана из трех»

Выбираться (для больших массивов) медиана из первого, среднего и последнего элементов.

После этого найденная медиана и средний элемент массива меняются местами, при этом медиана становится опорным элементом.

В отличие от классического рекурсивного алгоритма быстрой сортировки, данный метод позволяет эффективно работать на полностью отсортированном, а также полностью обратно отсортированном массивах.

Задача – найти *i* -ый по счету минимальный элемент в массиве A
(i-ую порядковую статистику)

Пример:

6	8	9	2

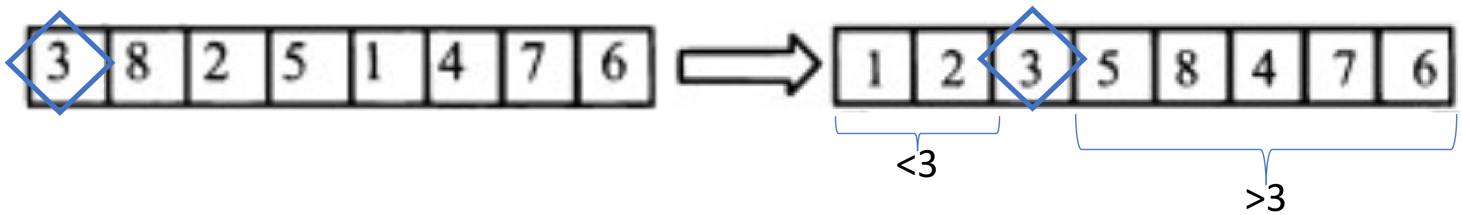
Если *i*=2, то 2-ая статистика равна 6

Если *i*=3, то 3-я статистика равна 8 и т.д.

При *i*=1 ищем min, при *i*=*n* ищем max. Для этих случаев сложность O(n).

Как насчет медианы? Можно отсортировать массив за O(n log n) и выдать любой *i*-ый элемент за O(1). Можно быстрее!

Используем идеи QuickSort (опорные элементы)



После разделения опорный элемент оказался на своем порядковом месте. Если нужна *i*=3-я порядковая статистика, то мы нашли ее. Если *i*>3, то далее обращаемся **только** к правой части, иначе – **только** к левой.

RSelect

RSELECT

Вход: массив A из $n \geq 1$ разных чисел и целое число $i \in \{1, 2, ..., n\}$.

Выход: i-я порядковая статистика массива A.

```
if n = 1 then                          // базовый случай
    return A[1]
выбрать опорный элемент p равномерно случайным образом из A
разделить A вокруг p
j := позиция p в разделенном массиве
if j = i then                          // вам посчастливилось!
    return p
else if j > i then
    return RSelect(first part of A, i)
else                                   // j < i
    return RSelect(second part of A, i - j)
```

Можно доказать, что время работы RSELECT равно в **среднем** O(n)

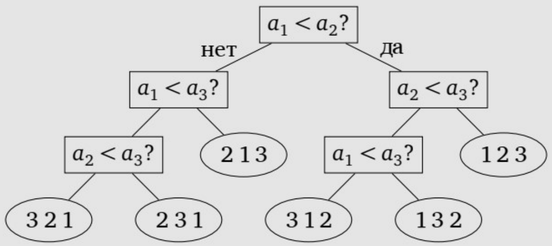
$$T_{\text{ср}}(n) \leq T_{\text{ср}}\left(\frac{n}{2}\right) + O(n) \Rightarrow O(n)$$

Нижняя оценка времени сортировки с помощью сравнений

<u>Утверждение.</u> Нижняя оценка для сортировки с помощью сравнений равна $\Omega(n \log_2 n)$.

Нижняя оценка $n \log n$ для сортировки

Работу алгоритма сортировки можно изобразить в виде дерева. Например, алгоритм на рисунке сортирует по возрастанию массив из трёх элементов a_1, a_2, a_3. Первым делом он сравнивает a_1 с a_2. Если a_1 не меньше a_2, он сравнивает его с a_3, в противном же случае сравнивает a_2 с a_3 и так далее. В конце концов мы приходим в лист, где уже известен правильный порядок трёх элементов (как перестановка чисел 1, 2, 3). Например, если $a_2 < a_1 < a_3$, то в листе будет записано «2 1 3».



Доказательство:
Рассмотрим двоичное дерево сортировки с помощью сравнений массива из *n* элементов. Пусть у него *k* уровней
Оно имеет *n!* листьев (возможных вариантов отсортированного массива). Тогда

$$2^k \geq n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

Поэтому время работы алгоритма, соответствующее количеству сравнений, оценивается глубиной дерева
$k \geq \log(n!)$

$$k \geq \frac{n}{2}\log(\frac{n}{2}) = \Omega(n \log n)$$

Более быстрые (по сравнению с $\Omega(n \log_2 n)$) сортировки при более строгих допущениях

1. Блочная сортировка BucketSort. Числовые данные равномерно распределены внутри некоторого диапазона. Например, *n* чисел в [0,…, 1]. Делим интервал на n корзин. При первом проходе раскладываем числа по корзинам $[\frac{i}{n}, \frac{i+1}{n}]$. Второй шаг – сортировка элементов внутри каждой корзины за линейное время (при малом количестве элементов в корзине). Третий шаг – объединение всех элементов в отсортированный массив за линейное время. Время $\mathrm{O}(n)$.

2. Сортировка подсчетом CountingSort. Аналогично BucketSort, но существует *k* разных значений элементов. Имеем *k* корзин. Время $\mathrm{O}(n)$.

3. Поразрядная сортировка Radix_Sort (расширенный вариант CountingSort). Обрабатывает *n*-элементные целочисленные входные массивы с достаточно большими числами. Распределяем исходные числа по корзинам в зависимости от величины младшего разряда (по возрастанию). Собираем числа в той последовательности, в которой они находятся после распределения по спискам. Повторяем эти действия для всех более старших разрядов поочередно. В итоге получаем отсортированный массив. Время $\mathrm{O}(n)$.

Пример поразрядной сортировки

Рассмотрим десятичные числа: 0, 8, 12, 56, 7, 26, 44, 97, 2, 4, 3, 3, 45, 10
Количество корзин m=10 – основанию системы счисления. Количество разрядов k=2.
Столько будет итераций.

1-ая итерация:

L0	L1	L2	L3	L4	L5	L6	L7	L8	L9
0	-	12	3	44	45	56	7	8	-
10		2	3	4		26	97		

Выписываем числа подряд
0, 10, 12, 2, 3, 3, 44, 4, 45, 56, 26, 7, 97, 8

2-ая итерация:

L0	L1	L2	L3	L4	L5	L6	L7	L8	L9
0	10	26	-	44	56	-	-	-	97
2	12			45					
3									
3									
4									
7									
8									

Выписываем числа подряд
0, 2, 3, 3, 4, 7, 8, 10, 12, 26, 44, 45, 56, 97