

Алгоритмы и структуры данных

Конспекты лекций основного потока

ЛЕКТОР: С. А. ОБЪЕДКОВ

Орлов Никита, Евсеев Борис, Рубачев Иван

НИУ ВШЭ, 2017

Содержание

Лекция 1. Асимптотика, простые алгоритмы, сортировка вставками	3
1.1 Асимптотика	4
1.2 Сортировка вставками	6
Лекция 2. Merge sort, Binary search, рекуррентные соотношения	9
2.1 Сортировка слиянием	9
2.2 Основная теорема	10
2.3 Binary Search	11
Лекция 3. Quick sort, оптимальность сортировки слиянием	13
3.1 QuickSort	13
3.2 Оптимальность сортировки слиянием	16

Лекция 1. Асимптотика, простые алгоритмы, сортировка вставками

Пусть перед нами стоит задача: найти в некотором массиве медиану. Техническое задание выглядит так: на вход программе подается массив A , на выходе хотим получить одну из медиан, неважно какую.

Напомним определение медианы m :

$$m \in A = \left\{ \begin{array}{l} |\{a \in A \mid a < m\}| \leq \frac{|A|}{2} \\ |\{a \in A \mid a > m\}| \leq \frac{|A|}{2} \end{array} \right.$$

Словами: медиана это такое число, что оно не больше половины элементов, но и не меньше половины элементов.

Легко видеть, что разных медиан в массиве может быть не больше двух, в зависимости от четности числа элементов.

Есть несколько способов решить эту задачу. Приведем несколько из них:

Алгоритм 1 Алгоритм поиска медианы

Ввод: Массив A

Вывод: Медиана m массива A

```
1: function MEDIAN( $A$ )
2:    $n := \text{len}(A)$ 
3:   for  $i := 0$  to  $(n - 1)$  do
4:      $l := 0$ 
5:      $g := 0$ 
6:     for  $j := 0$  to  $(n - 1)$  do
7:       if  $A[j] < A[i]$  then
8:          $l := l + 1$ 
9:       else if  $A[j] > A[i]$  then
10:         $g := g + 1$ 
11:    if  $l \leq n/2$  and  $g \leq n/2$  then
12:      return  $A[i]$ 
```

Посмотрим еще на один способ:

Алгоритм 2 Примитивный алгоритм поиска медианы

Ввод: Массив A

Вывод: Медиана m массива A

```
1: function MEDIAN( $A$ )
2:    $n := \text{len}(A)$ 
3:    $B := \text{sorted}(A)$ 
4:   return  $B[\lfloor \frac{n}{2} \rfloor]$ 
```

На первый взгляд это сложный подход, так как мы должны отсортировать массив и пока не знаем, как это сделать.

Асимптотика

Итак, у нас есть как минимум два способа найти медиану. Возникает абсолютно естественное желание как-нибудь выяснить, какой лучше. Оказывается, в программировании можно провести сразу несколько таких оценок по разным критериям. Два главных ресурса, которые потребляют алгоритмы, это процессорное время и память вычислительного устройства.

Определение 1.1. Время (измеренное в некой абстрактной единице), необходимое алгоритму для завершения своей работы, называется *временем работы алгоритма* и обозначается как $T(n)$, где n - длина входных данных.

Время работы можно считать в разных единицах, например в *секундах*, если реализация алгоритма и исполнитель фиксированы, или в *элементарных операциях*, если речь идет про машину Тьюринга.

Различают несколько оценок времени работы:

1. *Худший случай* - максимально возможное $T(n)$ на входе длины n . Чаще всего используется на практике, так как дает верхнюю оценку времени работы алгоритма.
2. *Средний случай* - математическое ожидание $T(n)$ на входе длины n . Используется на практике реже, чем худший случай, в силу частой неопределенности вероятностного пространства для вычисления математического ожидания.
3. *Лучший случай* - минимально возможное $T(n)$ на входе длины n . На практике не используется, так как к любому сколько угодно неэффективному алгоритму можно приписать проверку на оптимальность входных данных и выдать ответ быстрее, чем средний или худший случай. Например, в задаче про поиск медианы можно проверять, отсортирован ли массив, и, если он не отсортирован, честно запускать поиск.

Для всего зоопарка алгоритмов существует инструмент их анализа - *асимптотический анализ*. Это методология, в которой время работы и занимаемая память алгоритма ставятся в соответствие классу функций.

Для начала дадим несколько определений.

Определение 1.2. О-большим от $g(n)$ называют такое множество функций, которое удовлетворяет условию

$$\underline{O}(g(n)) = \{f(n) \mid \exists c_2 > 0, n_0 > 0 \forall n \geq n_0 : 0 \leq f(n) \leq c_2 g(n)\}$$

Иными словами, запись $f(n) \in O(g(n))$ означает, что $f(n)$ растет не быстрее, чем $g(n)$.

Определение 1.3. о-малым от $g(n)$ называют такое множество функций, которое удовлетворяет условию

$$\bar{O}(g(n)) = \{f(n) \mid \forall c_2 > 0 \exists n_0 > 0 : \forall n \geq n_0 : 0 \leq f(n) < c_2 g(n)\}$$

Определение 1.4. Ω -большим от $g(n)$ называют такое множество функций, которое удовлетворяет условию

$$f(n) \in \Omega(g(n)) \leftrightarrow g(n) = \underline{\underline{O}}(f(n))$$

Определение 1.5. ω -малым от $g(n)$ называют такое множество функций, которое удовлетворяет условию

$$f(n) \in \omega(g(n)) \leftrightarrow g(n) = \bar{\bar{o}}(f(n))$$

Определение 1.6. $\Theta(g(n))$ называется такое множество функций, которое удовлетворяет условию

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

Иными словами, $\Theta(g(n))$ растет примерно также, как и $f(n)$.

В нашем курсе мы часто будем писать что-то похожее на

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

Такая запись с точки зрения математики некорректна, но мы будем понимать знак равенства как

$$T(n) \in \Theta(f(n))$$

Например:

$$4n^2 + 12n + 12 = \Theta(n^2)$$

$$c_1 = 1, c_2 = 16, n_0 = 2$$

$$\forall n \geq n_0 : 0 \leq n^2 \leq 4n^2 + 12n + 12 \leq 16n^2$$

В общем случае верно следующее:

Лемма 1.7. Если многочлен $p(n)$ представим в виде

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

то

$$p(n) = \Theta(n^d)$$

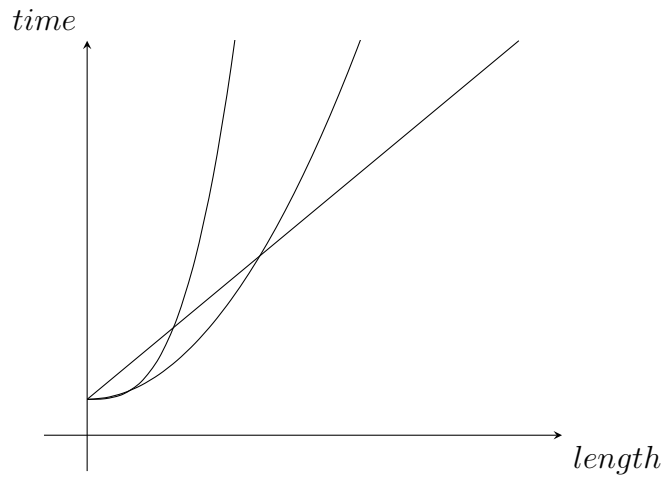
Замечание 1.8. Обычно функцию, описывающую время работы или память, занимаемую алгоритмом, называют *оценкой времени работы или памяти* алгоритма.

Замечание 1.9. По соглашению мы рассматриваем *асимптотически неотрицательные* функции, то есть такие, что

$$\exists n_0 \forall n > n_0 : f(n) \geq 0$$

Теперь поймем, что скрывается за классами функций Θ .

Пусть есть классы $\Theta(n)$, $\Theta(n^2)$, $\Theta(n^3)$. Для некоторых алгоритмов существует *оценка*, принадлежащая одному из этих классов. Помня про константу, можно сказать, что на достаточно большом объеме данных алгоритм с меньшей оценкой будет работать в среднем быстрее. Это ключевая мысль асимптотического анализа. Представить ее можно, построив графики неких линейной, квадратичной и кубической функций.



Для теоретического анализа сложности алгоритма берутся достаточно большие числа, но нужно понимать, что на практике может оказаться так, что входные данные могут быть меньше, чем n_0 .

Теперь получив матаппарат, оценим время работы алгоритмов поиска медианы.

Замечание 1.10. Будем считать, что элементарные арифметические операции, операции присваивания, копирования и тому подобные выполняются за $\Theta(1)$, иначе говоря, время их выполнения константо.

Первый алгоритм:

1. Лучший случай: медиана на первом месте. Тогда алгоритм выполнит одну итерацию внешнего цикла, n итераций внутреннего цикла, каждая из которых занимает константное время, и завершит работу. Сложность: $\Theta(1 \cdot n) = \Theta(n)$. Такая сложность считается достаточно хорошей.
2. Худший случай: медиана на последнем месте. Тогда алгоритм выполнит n итераций внешнего цикла, на каждой итерации произойдет n итераций внутреннего цикла. Сложность: $\Theta(n^2 - n) = \Theta(n^2)$.

Доказательство корректности заключается в том, что алгоритм *реализует* определение медианы. В таком случае он корректен, пока нет ошибок на уровне написания кода.

Второй алгоритм:

Второй алгоритм сложнее для оценки, так как мы не знаем, как сортируем массив. Операция взятия элемента выполняется за $\Theta(1)$. Остается сортировка, которую можно выполнить разными способами за разное время.

Сортировка вставками

Давайте возьмем простой алгоритм сортировки и оценим его сложность.

Алгоритм 3 Сортировка вставками

Ввод: Массив A с заданным на нем порядком $<$.

Вывод: Отсортированный по возрастанию массив A .

```
1: function INSERTION_SORT( $A$ )
2:    $n := \text{len}(A)$ 
3:   for  $i := 1$  to  $(n - 1)$  do
4:      $k := A[i]$ 
5:     for  $j := i - 1$  to  $0$  do
6:       if  $k < A[j]$  then
7:          $A[j + 1] := A[j]$ 
8:       else
9:         break
10:     $A[j + 1] := k$ 
11:  return  $A$ 
```

Словами: смотрим каждый i элемент и ищем его место среди первых $i - 1$ элементов.

Для начала докажем корректность алгоритма. Для этого будем использовать *инвариант* - свойство математического объекта, которое не меняется после преобразования объекта.

Теорема 1.11. Пусть есть неупорядоченный пронумерованный набор A элементов с заданным на них порядком меньше, и мы исполняем над ним алгоритм. Инвариант: элементы $A[0], \dots, A[i-1]$ являются перестановкой исходных элементов в правильном порядке.

Доказательство. Докажем по индукции. База $i = 1$ верна. Пусть инвариант верен для $i - 1$ шага. Тогда смотрим $k = A[i]$ элемент.

Возможны 3 случая:

1. k - самый большой среди первых i элементов. Тогда алгоритм пропустит эту итерацию и перейдет к следующему.
2. k - самый маленький среди первых i элементов. Тогда алгоритм передвинет его в начало, пройдя весь цикл.
3. В противном случае, мы начинаем перебирать все элементы среди первых i до тех пор, пока операция сравнения на "меньше" не вернет ложь. Это означает, что мы в отсортированном массиве нашли элемент под номером j , который меньше либо равен k :

$$A[j] \leq k \leq A[j + 1]$$

Тогда мы сдвигаем элементы с $j + 1$ до i на одну позицию вправо, и на $j + 1$ место ставим k .

$$A[j] \leq k = A[j + 1] < A[j + 2]$$

Все элементы с 0 по j позицию меньше либо равны k , а элементы с $j + 2$ по i позицию они больше k .

[:||:]

Из доказательства корректности инварианта прямо следует доказательство корректности алгоритма: когда алгоритм закончит свою работу, $i = n$, а значит инвариант верен для n элементов, а значит массив отсортирован.

Теперь можно оценить время работы сортировки вставками:

1. Лучший случай: массив уже отсортирован. Но тогда внешний цикл совершит $n - 1$ итерацию, на каждой из которых произойдет одно сравнение. Сложность получилась $\Theta(n)$.
2. Худший случай: массив отсортирован в обратном порядке. Тогда на каждой итерации число шагов внутреннего цикла будет уменьшаться на 1. Значит

$$T(n) = \sum_{i=1}^{n-1} i = \Theta(n^2)$$

Лекция 2. Merge sort, Binary search, рекуррентные соотношения

Лекция будет дополнена примерами решения рекуррентных соотношений без основной теоремы, а так же доказательством теоремы. Текущая версия вычитана лектором.

Поговорим еще немного про сортировки. Сортировка вставками имеет квадратичную сложность, что не оптимально. Есть более быстрые алгоритмы, один из них называется *сортировкой слиянием*.

Сортировка слиянием

Суть сортировки достаточно проста: если у нас есть две отсортированные последовательности, мы можем их объединить в одну отсортированную, а последовательность длины один уже отсортирована, а значит мы можем разбить наш массив на блоки одинаковой длины, и рекурсивно отсортировать.

Опишем функцию слияния двух массивов разной длины.

Алгоритм 4 Функция слияния отсортированных массивов

Ввод: Отсортированные массивы A, B .

Вывод: Отсортированный массив C

```
1: function MERGE( $A, B$ )
2:    $i := 0$ 
3:    $j := 0$ 
4:   vector  $C$ 
5:   while  $i < \text{len}(A)$  and  $j < \text{len}(B)$  do
6:     if  $A[i] \leq B[j]$  then
7:        $C.\text{push\_back}(A[i])$ 
8:        $i := i + 1$ 
9:     else
10:       $C.\text{push\_back}(B[j])$ 
11:       $j := j + 1$ 
12:    $C := C + A[i : \text{len}(A)] + B[j : \text{len}(B)]$ 
13:   return  $C$ 
```

Доказательство корректности. На входе отсортированные массивы. На каждой итерации цикла выбирается наименьший из еще не выбранных элементов. [:||:]

Время работы. Алгоритм смотрит на каждый элемент каждого массива, тогда его сложность получается $\Theta(\text{len}(A) + \text{len}(B))$. [:||:]

Теперь сам алгоритм сортировки слиянием:

Алгоритм 5 Merge Sort

Ввод: Массив A

Вывод: Отсортированный массив C

```
1: function MERGE_SORT( $A$ )
2:   if  $\text{len}(A) < 2$  then
3:     return  $A$ 
4:   else
5:      $n := \text{len}(A)$ 
6:      $A_1 := \text{merge\_sort}(A[0 : n/2])$ 
7:      $A_2 := \text{merge\_sort}(A[n/2 : n])$ 
8:     return  $\text{merge}(A_1, A_2)$ 
```

Замечание 2.1. Сортировку слиянием можно написать разными способами. Например, изначально разбить массив на куски длиной 10, каждую из них отсортировать вставками, а потом уже последовательно слить в один отсортированный массив.

Доказательство корректности. Пока корректна функция merge , весь алгоритм корректен, так как вся задача разбивается на меньшие подзадачи, а рекурсия остановится на массивах размера 1. [:|||:]

Время работы. Время работы данного алгоритма $T(n) = \Theta(n \cdot \log(n))$: У нас $\log(n)$ - глубина рекурсии, а на каждом шаге мы пройдемся в сумме по всему массиву. [:|||:]

Время работы сортировки слиянием можно записать и по-другому, с помощью *рекуррентной формулы*. В нашем случае она будет иметь вид

$$T(n) = \begin{cases} c, & n = 1; \\ 2T(\frac{n}{2}) + O(n), & n > 1; \end{cases}$$

где $2T(\frac{n}{2})$ - сложность рекурсивных вызовов, $O(n)$ - сложность слияния. Для таких соотношений хочется получить правило их раскрытия в явную формулу. О таком преобразовании говорит *основная теорема о рекуррентных соотношениях*.

Основная теорема

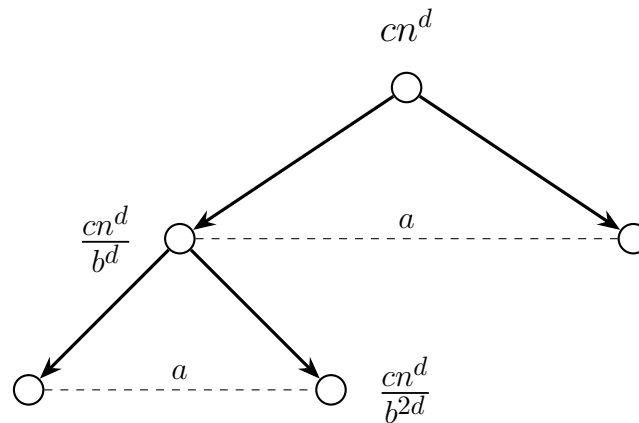
Теорема 2.2. Пусть у нас есть рекуррентное соотношение, записанное в виде

$$T(n) \leq \begin{cases} c, & n = 2 \\ aT(\frac{n}{b}) + cn^d, & n > 2 \end{cases}$$

Тогда его можно представить в следующем виде:

$$T(n) = \begin{cases} O(n^d \log(n)), & a = b^d \\ O(n^d), & a < b^d \\ O(n^{\log_b(a)}), & a > b^d \end{cases}$$

Доказательство. Рассмотрим алгоритм, который нашу задачу делит на каждом шаге на b подзадач.



На каждом уровне мы делим задачу на a подзадач, размер каждой отличается от другого уровня в b раз.

[:::]

Binary Search

Теперь давайте разберем другой рекуррентный алгоритм - *бинарный поиск*. Это алгоритм ищет в отсортированном элементе некоторый элемент и возвращает его индекс.

Алгоритм 6 Binary Search

Ввод: Отсортированный массив A , элемент x

Вывод: Индекс элемента e если он есть в массиве, -1 в противном случае.

```

1: function BINARY_SEARCH( $A, x$ )
2:    $b := 0$ 
3:    $e := \text{len}(A)$ 
4:   while  $b < e$  do
5:      $m = \text{round}(\frac{b+e}{2})$ 
6:     if  $A[m] == x$  then
7:       return  $m$ 
8:     else if  $A[m] < x$  then
9:        $b := m + 1$ 
10:    else
11:       $e := m$ 
12:  return -1

```

Словами: смотрим в середину, сравниваем, сдвигаем границы поиска в сторону, где элемент может быть, повторяем пока не нашли или пока $b \neq e$.

Время работы. Формула времени нашего алгоритма:

$$T(n) = \begin{cases} c, & n = 2 \\ T(\frac{n}{2}) + c, & n > 2 \end{cases}$$

По основной теореме

$$a = 1, \quad b = 2, \quad d = 0$$

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

[:::]

Лекция 3. Quick sort, оптимальность сортировки слиянием

CAUTION! RECURTION IS HERE!

В стратегии *разделяй и властвуй*, которая используется во многих алгоритмах обработки данных, есть 3 четко выделяемые фазы.

1. *Деление*. Задача размера n делится на несколько меньших подзадач, подобных исходной.
2. *Решение*. Каждая из подзадач рекурсивно решается тем же алгоритмом. При этом важно, чтобы рекурсия остановилась на элементарном размере подзадачи. Под элементарным подразумевается, что такую подзадачу можно решить за константное время.
3. *Слияние*. Полученные результаты сливаются в одно общее решение.

На примере Merge Sort'а это выглядит следующим образом:

Деление	$A_1 = A[: \frac{n}{2}]$ $A_2 = A[\frac{n}{2} :]$
Решение	$A'_1 = \text{mergesort}(A_1)$ $A'_2 = \text{mergesort}(A_2)$
Слияние	return $\text{merge}(A_1, A_2)$

С ним все просто и понятно. Поделили - решили - объединили. Но существуют и другие алгоритмы сортировки. Один из самых знаменитых из них - *Quick Sort*.

QuickSort

Идея алгоритма в чем-то схожа с сортировкой слиянием: мы делим массив на две половины специальным образом и применяем алгоритм к каждой из частей рекурсивно. Главное отличие - алгоритм не имеет отдельной процедуры слияния и тратит всего $O(1)$ памяти, вместо совершенно нерационального потребления Merge Sort'ом.

Делить задачу мы будем следующим образом: возьмем *опорный* элемент $p \in A$, и приведем массив в следующее состояние:

$$\begin{array}{ccc}
 A_1 & & A_2 \\
 [A[i_0] \ \dots \ A[i_k] \ p \ A[i_{k+1}] \ \dots \ A[i_n]] \\
 \leq p & & > p
 \end{array}$$

Словами: делим массив на части, в каждой из которых все элементы либо больше или равны опорного, либо меньше него.

Затем рекурсивно запускаем алгоритм от A_1 и A_2 , пока не дойдем до задачи размера 1. Утверждается, что когда рекурсия остановится, задача будет решена, и массив A будет отсортирован.

Этот алгоритм полностью зависит от процедуры выбора опорного элемента. В идеальном случае надо брать медиану на каждом шаге, тогда мы будем делить массив ровно на две части, и время работы $T(n)$ будет выражаться так:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n) = \Theta(n \log n)$$

В конце стоит $O(n)$, потому что нам нужно во время перемещения элементов посмотреть хотя бы на каждый из них.

Но если мы хотим сделать так, чтобы такая асимптотика достигалась постоянно, нужно искать медиану, а это как минимум $O(n)$, если повезет, но асимптотика тогда увеличится. Нерационально.

Самая главная часть в алгоритме называется *Partition*, именно она ответственна за разделение массива. Приведем псевдокод оптимальной по памяти версии, которая сортирует inline.

Алгоритм 7 Partition

Ввод: Массив A , границы разделяемой части l, r .

```

1: function PARTITION( $A, l, r$ )
2:    $i := l$ 
3:   for  $j := l + 1$  to  $r$  do
4:     if  $A[j] \leq A[l]$  then
5:       swap( $A[i], A[j]$ )
6:        $i += 1$ 
7:     else
8:       continue
9:   swap( $A[l], A[i]$ )

```

Эта версия алгоритма за опорный элемент берет левую границу интервала. Получается такой переход:

$$\begin{array}{ccccccc}
 [A[0] & \dots & A[l] & & \dots & \dots & A[r] & \dots & A[n]] \\
 [A[0] & \dots & A[i_0] & \dots & A[i_{\frac{k}{2}}] & A[l] & A[i_{\frac{k}{2}+1}] & \dots & A[i_k] & A[r] & \dots & A[n]]
 \end{array}$$

Время работы. Посчитаем для худшего случая - когда наш массив отсортирован в обратном порядке, значит операция Partition будет вызываться от массивов, каждый раз размером на 1 меньше. Получается

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

В лучшем случае время будет как у Merge Sort'a - $O(n \log n)$

[:::]

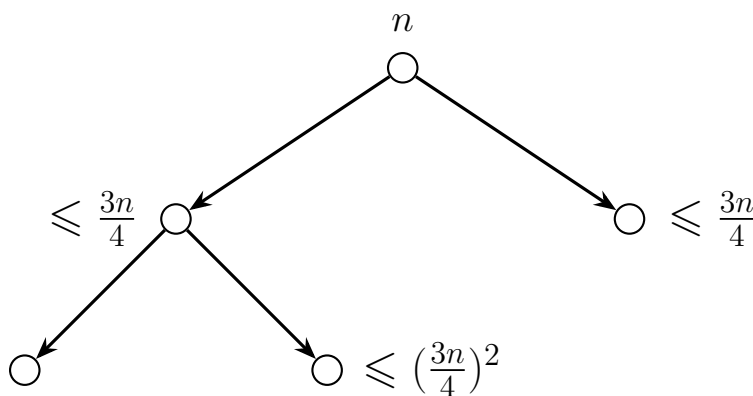
Этот алгоритм есть пример тех алгоритмов, для которых лучше рассматривать время в среднем случае.

Определение 3.1. Назовем элемент *центральным*, если в отсортированном массиве он больше первой четверти элементов и меньше последней четверти элементов.

Такой элемент оказывается очень удобным, так как его можно брать за опорный, причем он гарантированно разобьет массив на непустые части. Искать такой элемент можно *детерминированно*, или точно вычитывать по какому-то алгоритму, но сложность такого выбора будет $O(n)$. Мы же будем использовать *вероятностный* подход, или брать случайный элемент в массиве.

С этого момента Quick Sort становится *рандомизированным* алгоритмом, так как на одном и том же входе шаги алгоритма неизвестны заранее. При этом становится легко считать его сложность

Время работы. Построим дерево исполнения алгоритма для случая, когда он разбивает задачи на подзадачи меньшие, чем $\frac{3}{4}$.



На j подзадачу тратится время $O((\frac{3}{4})^j n)$, на j подзадач - $O((\frac{4}{3})^{j+1})$. В итоге получаем, что среднее время на все подзадачи

$$T(n) = O\left(\left(\frac{3}{4}\right)^j n\right) \cdot O\left(\left(\frac{4}{3}\right)^j\right)$$

Прологарифмируем и получим, что

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

[:||:]

На практике обычно Quick Sort работает быстрее, чем Merge Sort, несмотря на одинаковую асимптотику. Связано это с тем, что QS сортирует на месте, в исходном массиве, MS тратит время на работу с памятью, так как ему нужно еще $O(n)$ памяти.

Замечание 3.2. При реализации алгоритма Partition можно вместе с границами передавать опорный элемент, переписав тело цикла. Получим процедуру, которая будет делить в интервале массив на две части, которые меньше и больше опорного.

Оптимальность сортировки слиянием

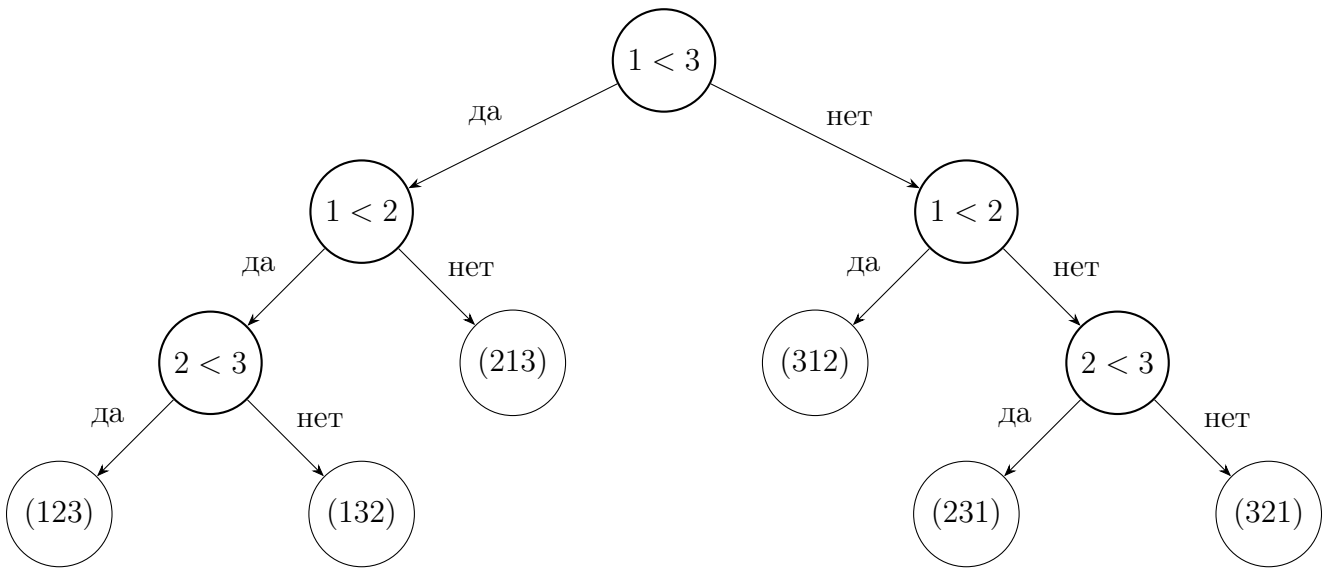
Утверждается, что время работы сортировки слиянием асимптотически самое маленькое.

Теорема 3.3. *Нельзя отсортировать массив быстрее, чем за $O(n \log n)$, если разрешено использовать только сравнение элементов и их перемещение.*

Доказательство. Рассмотрим массив A . Пусть ответом на задачу будет такая подстановка σ , что после ее применения массив окажется отсортирован.

Рассмотрим *дерево решений* некоторого абстрактного алгоритма сортировки. Пусть вершиной будет операция сравнения, ребром - ее исход, а листом - искомая перестановка. Такое дерево можно построить для каждого алгоритма.

Покажем пример такого дерева для сортировки массива длины 3:



Временем работы всего алгоритма будет высота этого дерева h . Число листьев - $n!$.

$$n! \leq 2^h$$

$$\log_2 n! \leq h$$

$$h \geq \log_2 n! \geq \log_2 \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log_2(n) - 1 = \Omega(n \log n).$$

[:::]