Лекции по предмету

Алгоритмы и структуры данных

Группа лектория ФКН ПМИ 2015-2016
Никита Попов
Тамерлан Таболов
Лёша Хачиянц

2016 год

Содержание

1 Задача о Ханойских башнях 12.01.2016

Оргмоменты

Одна контрольная — контест на реализацию какого-то алгоритма. Пользоваться своим кодом запрещено.

Задача — привести алгоритм, провести теоретический анализ (доказать его корректность, оценить время работы) и запрограммировать. Сначала сдаётся теория, потом практика.

Экзамен устный.

$$O_{\text{итоговая}} = 0.7 \cdot O_{\text{накопленная}} + 0.3 \cdot O_{\text{экзамен}}$$

$$O_{\text{накопленная}} = 0.2 \cdot O_{\text{КР}} + 0.12 \sum_{i=1}^{5} O_{\text{ДЗ i}} + 0.2 \cdot O_{\text{семинары}}$$

Списывание, как обычно, *не поощряется*. ДЗ предполагается не обсуждать.

Здесь (ссылка слева) можно найти ссылки на ДЗ и краткое содержание лекций.

Автоматов *пока* не предусмотрено.

Литература:

- Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. — «Алгоритмы. Построение и анализ»
- Дасгупта С., Пападимитриу Х., Вазирани У. — «Алгоритмы»

Лекция

Ханойские башни

Есть три стержня. На первый стержень нанизано 64 диска, от самого большого к самому маленькому. Задача: переложить все диски на второй стержень. Ограничения:

- Диски можно переносить только по одному.
- Нельзя класть диск большего диаметра на диск меньшего диаметра.

Какой может быть алгоритм?
Варианты из аудитории:

1. Полный перебор
2. Рекурсивный алгоритм.

Рассмотрим такой рекурсивный алгоритм:

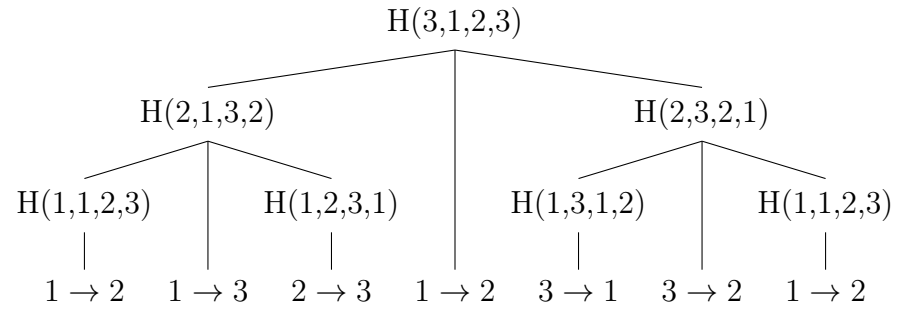
1. Переложим все диски, кроме n-ного, на третий стержень;
2. Переложим n-ный диск с первого на второй стержень;
3. Переложим все остальные с третьего стержня на второй.

Запишем этот алгоритм с помощью псевдокода:

Algorithm 1 Рекурсивный алгоритм решения задачи о Ханойской башне

```
1: function HANOI3(n, i, j, k)        ▷ n — количество дисков, i, j, k — номера стержней
2:     if n > 0 then
3:         HANOI3(n − 1, i, k, j)
4:         move i → j
5:         HANOI3(n − 1, k, j, i)
```

Нарисуем дерево операций для $n = 3$:



Алогритм, по сути, обходит это дерево в глубину и при этом слева направо, выполняя все перемещения, что встретятся.

Это дерево можно рассматривать, как полное бинарное дерево глубины n, если перемещения учитывать не в отдельных листьях, а в родительских узлах. Тогда в каждом узле мы выполняем одно действие, а в полном бинарном дереве $2^n - 1$ узлов. Следовательно, выполняется $2^n - 1$ перемещение.

Пусть число перемещений для n дисков равно $f(n)$. Тогда верно следующее:

$$f(n) = \begin{cases} 0, & n = 0 \\ 2f(n-1) + 1, & n > 0 \end{cases}$$

Свойство: $f(n) = 2^n - 1$

Доказательство. Докажем это по индукции. База верна, так как $f(0) = 0 = 2^0 - 1$. Теперь пусть предположение верно для $n-1$, то есть $f(n-1) = 2^{n-1} - 1$. Тогда $f(n) = 2f(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1$, что и требовалось доказать. □

Можно ли улучшить время работы? Оказывается, что нет.

Рассмотрим некоторый алгоритм. Он рано или поздно должен переложить наибольший диск на второй стержень. Для этого ничего не должно быть на нём и на втором, т.е. все на третьем. А как получить эту конфигурацию? Оптимальным алгоритмом на $n-1$ шаг, что приводит к нашим вычислениям и уже полученному минимальному результату в $2^n - 1$.

Утверждение. *Задачу о Ханойских башнях нельзя решить за меньшее число шагов, причём решение с таким числом шагов ровно одно.*

Доказательство. По индукции.

База ($n = 0$)**:** очевидно, решить быстрее, чем за 0 шагов нельзя и последовательность такая ровно одна.

Переход ($n - 1 \to n$)**:** предположим, что мы доказали это утверждение для $n-1$. Рассмотрим утверждение для n. Рано или поздно алгоритму понадобится освободить первые два стержня, чтобы переложить первый диск на второй стержень. Необходимо сделать это одно перекладывание и после вернуть все оптимальным алгоритмом. Итого, опираясь на предположение индукции шагов в оптимальном и единственном решении для $n-1$, нам понадобится $2(2^{n-1} - 1) + 1$ шаг, что и равно $2^n - 1$. □

Изменим задачу:

Четыре стержня

Условие в остальном ровно то же. Стала ли задача проще?

Сложнее она точно не стала, т.к. четвёртым можно просто не пользоваться.

Рассмотрев переход от двух к трём, кажется, что должно быть проще; как можно воспользоваться четвёртым?

Предложение: переложить предпоследний отдельно на четвёртый и сэкономить на перекладывании башни из $n-1$, перекладывая вместо неё башню из $n-2$

Algorithm 2 Рекурсивный алгоритм решения задачи о Ханойской башне на 4-х стержнях, версия 1

```
1: function HANOI4(n, i, j, k, l)        ▷ n — количество дисков, i, j, k, l — номера стержней
2:     if n > 0 then
3:         HANOI4(n − 1, i, l, k, j)
4:         move i → k
5:         move i → j
6:         move k → j
7:         HANOI4(n − 1, l, j, i, k)
```

Построив аналогичное дерево, получим, что в каждом узле три перемещения, а узлов $2^{\lfloor \frac{n}{2} \rfloor} - 1$ — экономия, но не очень большая.

Построим другой алгоритм:

Algorithm 3 Рекурсивный алгоритм решения задачи о Ханойской башне на 4-х стержнях, версия 2

```
1: function HANOI4(n, i, j, k, l)        ▷ n — количество дисков, i, j, k, l — номера стержней
2:     if n > 0 then
3:         HANOI4(n − m, i, l, k, j)
4:         HANOI3(m, i, j, k)
5:         HANOI4(n − m, l, j, i, k)
```

Заметим, что число шагов зависит от m. Пусть $n_m = \frac{m(m+1)}{2}$ (если n другое, то на первом шаге выберем такой m, чтобы $n - m$ было таким, а дальше на вход будет поступать число такого вида).

Построим дерево алгоритма. Оно также будет полным бинарным деревом. Заметим, что в нём m уровней, так как при каждом шаге m уменьшается на единицу. Это связано с тем, что $n_m - m = \frac{m(m+1)}{2} - m = \frac{(m-1)m}{2} = n_{m-1}$. При этом в узле на i-м уровне (нумерация с нуля) проводится $2^{m-i} - 1$ операция, так как мы пользуемся доказанным ранее алгоритмом для трёх стержней. Тогда на каждом уровне выполняется $2^m - 2^i$ операций. Тогда всего выполняется $\sum\limits_{i=0}^{m-1}(2^m - 2^i) = m2^m - \sum\limits_{i=0}^{m-1} 2^i = (m-1)2^m + 1$ операций.

Пусть число перемещений для n дисков равно $g(n)$. Тогда

$$g(n_m) = \begin{cases} 0, & m = 0 \\ 2g(n_{m-1}) + 2^m - 1, & m > 0 \end{cases}$$

Предложение: $g(n_m) = (m-1)2^m + 1$

Доказательство. По индукции. База верна, так как $g(n_0) = 0 = (0-1)2^0 + 1 = -1 + 1$. Теперь допустим, что предположение верно для n_{m-1}, то есть $g(n_{m-1}) = (m-2)2^{m-1} + 1$. Тогда $g(n_m) = 2g(n_{m-1}) + 2^m - 1 = (m-2)2^m + 2 + 2^m - 1 = (m-1)2^m + 1$. □

Заметим, что при достаточно больших n верно, что $m \approx \sqrt{2n}$. Тогда $g(n) \approx \sqrt{2n} \cdot 2^{\sqrt{2n}}$. Тогда $g(n) = \Theta(\sqrt{n} \cdot 2^{\sqrt{2n}})$. Θ означает (грубо говоря), что функция растёт примерно так же.

Попробуем обобщить этот алгоритм для любого числа стержней:

Algorithm 4 Рекурсивный алгоритм решения задачи о Ханойской башне, общий случай

```
1: function HANOI(n, i, j, P)        ▷ n — количество дисков, i, j, — основные стержни
2:                                   ▷ P —множество вспомогательных стержней
3:     if n > 0 then
4:         choose p ∈ P
5:         R := P \ p
6:         if R = ∅ then
7:             HANOI3(n, i, j, p)
8:         else
9:             HANOI(n − m, i, p, R ∪ {j})
10:            HANOI(m, i, j, R)
11:            HANOI(n − m, p, j, R ∪ {i})
```

Пусть для перемещения n дисков с помощью алгоритма на k стержнях ($k \geq 3$) нужно $h(n, k)$ операций. Тогда верно следующее:

$$h(n_m, k) = \begin{cases} 0 & n = 0 \\ 2^{n_m} - 1 & n > 0, k = 3 \\ 2h(n_{m-1}, k) + h(m, k-1) & n > 0, k > 3 \end{cases}$$

2 Алгоритмы сортировки 14.01.2016

Задача сортировки

Вход: последовательность чисел (строго говоря, может быть что угодно с полным порядком) $(a_1, a_2, \ldots, a_n)$.

Выход: $(a_{i_1}, a_{i_2}, \ldots, a_{i_n})$, где $a_{i_k} \leq a_{i_{k+1}}$. Другими словами, на выходе получается отсортированная по возрастанию последовательность.

Рассмотрим неэффективный алгоритм:

Algorithm 5 Неэффективный алгоритм сортировки

```
1: function RECURSIVE_SORT(a)                          ▷ a = (a_1, a_2, ..., a_n)
2:     n := |a|
3:     if n > 1 then
4:         RECURSIVE_SORT(a[1 : n − 1])
5:         k := a_n
6:         for i := n − 1 downto 1 do
7:             if a_i > k then
8:                 a_{i+1} := a_i
9:             else
10:                break
11:            a_{i+1} := k
```

$[6, 8, 3, 4] \to [3, 6, 8, 4] \xrightarrow{8>4} [3, 6, \ , 8] \xrightarrow{6>4} [3, \ , 6, 8] \xrightarrow{3<4} [3, 4, 6, 8]$

По сути, мы идём слева направо и каждому элементу находим место среди прошлых уже отсортированных элементов.

Теперь рассмотрим алгоритм *сортировки вставками*.

Algorithm 6 Алгоритм сортировки вставками

```
1: function RECURSIVE_SORT(a)                          ▷ a = (a_1, a_2, ..., a_n)
2:     n := |a|
3:     for j := 2 to n do
4:         k := a_j
5:         for i := j − 1 downto 1 do
6:             if a_i > k then
7:                 a_{i+1} = a_i
8:             else
9:                 break
10:            a_{i+1} := k
```

Докажем корректность алгоритма формально. Для этого найдём *инвариант*.

Инвариант: в начале каждой итерации цикла по j массив с 1 по $j - 1$ индекс уже отсортирован. При этом он состоит из тех же элементов, что и раньше.

Если это условие выполняется, то после выполнения алгоритма, весь массив (с 1-го по n-ый индексы) будет отсортирован.

Доказательство. По индукции:

База: $j = 2$ — $a[1:1]$ отсортирован

Переход Всё до j-го отсортировано; Поставим a_j на нужное место. Тогда полученный массив также будет отсортирован. □

Насколько эффективно он работает? Понятно, что это зависит от входных данных. Ясно, что чем больше элементов, тем дольше он работает. Понятно также, что если массив уже отсортирован, то работать он будет быстрее.

$T(n)$ — время работы на входе длины n в худшем случае. (1)

в среднем случае. (2)

в лучшем случае. (3)

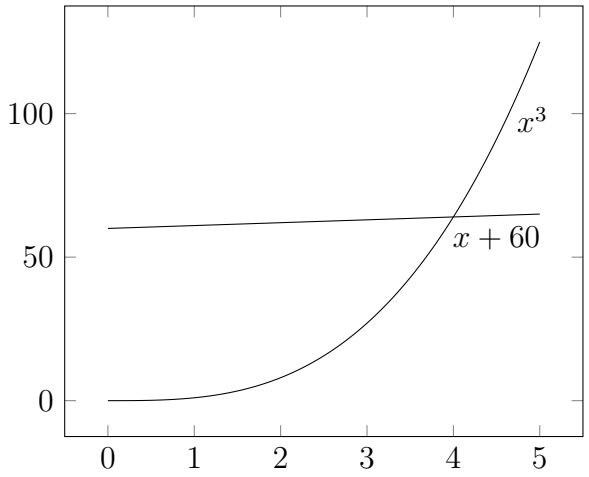
Однако оценка в лучшем случае, вообще говоря, бесполезна. Ведь любой алгоритм можно модифицировать так, чтобы в каком-то случае он работал очень быстро.

Асимптотический анализ: как меняется $T(n)$ при $n \to \infty$? Для исследования этого обычно применяют O-нотацию или Θ-нотацию.

$$\Theta(g(n)) = \{f(n) \mid \exists\, c_1, c_2 > 0 : \forall n \geqslant n_0 \implies 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\}$$

Например, пусть задана функция $f(n) = 3n^2 + 2n - 6$. Тогда $f(n) \in \Theta(n^2)$.

Асимптотика — это хорошо, но на константы тоже стоит обращать внимание: для маленьких n вполне может быть, что n^3 работает быстрее, чем n.



Оценим *худший случай* нашего алгоритма (когда на каждом шагу приходится совершать максимальное число перемещений): $T(n) = \sum\limits_{j=2}^{n} \sum\limits_{i=j-1}^{1} \Theta(1) = \sum\limits_{j=2}^{n} \Theta(j) = \Theta(n^2)$

Средний случай: Предположим, что все входы равновероятны. Тогда будет выполняться примерно половина сравнений и $T(n) = \sum \Theta(\frac{j}{2}) = \Theta(n^2)$

Лучший случай — это случай, когда массив уже отсортирован. Тогда $T(n) = \Theta(n)$.

Рассмотрим другой алгоритм — *сортировку слиянием.*

Algorithm 7 Алгоритм сортировки слиянием

```
1: function MERGE_SORT(a)                          ▷ a = (a_1, a_2, ..., a_n)
2:     n := |a|
3:     if n > 1 then
4:         b_1 := MERGE_SORT(a[1 : n/2])
5:         b_2 := MERGE_SORT(a[n/2 + 1 : n])
6:         a := MERGE(b_1, b_2)          ▷ сливаем два отсортированных массива в один
7:     return a
```

Рассмотрим, как может работать $\text{MERGE}(b_1, b_2)$ на примере. Пусть даны массивы $b_1 := [2, 5, 6, 8]$ и $b_2 := [1, 3, 7, 9]$.

Будем сливать элементы из массивов в результирующий массив b, сравнивая поочерёдно минимальные элементы, которые ещё не вошли в результирующий массив.

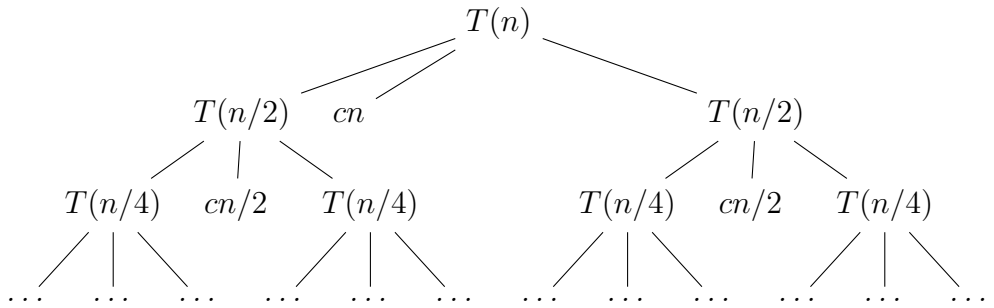
- $2 > 1$. Тогда $b[1] := b_2[1] = 1$ (будем считать, что нумерация идёт с единицы).
- $2 < 3$. Тогда $b[2] := b_1[1] = 2$.
- Аналогично продолжаем для всех остальных элементов массивов.

Очевидно, что алгоритм корректен, а его сложность — линейная, так как мы один раз проходим по массивам, то есть $\Theta(n)$.

Пусть худшее время для MERGE_SORT — $T(n)$. Тогда

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

Построим дерево рекурсии:



На каждом уровне cn работы, а высота дерева — $\log_2 n$. Общее время работы — $n\Theta(1) + cn \log n = \Theta(n \log n)$.

3 Нотация, быстрая сортировка. 19.01.2016

Повторим **нотацию:**

$$\Theta(g(n)) = \{f(n) \mid \exists c_1 > 0, c_2 > 0 \exists n_0 : \forall n \geqslant n_0 \implies 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\},$$

Θ — *асимптотическое* $=$. Например, $2n = \Theta(n)$. По определению, $c_1 n \leqslant 2n \leqslant c_2 n$. Тогда $c_1 = 1, c_2 = 2$.

$$O(g(n)) = \{f(n) \mid \exists c_2 > 0 \exists n_0 : \forall n \geqslant n_0 \implies 0 \leqslant f(n) \leqslant c_2 g(n)\}$$

O — *асимптотическое* $\leqslant$. Например, по этому определению $n = O(n \log n)$, так как при достаточно больших n_0 $\log n > 1$. Тогда $c_2 = 1$.

$$\Omega(g(n)) = \{f(n) \mid \exists c_1 > 0 \exists n_0 : \forall n \geqslant n_0 \implies 0 \leqslant c_1 g(n) \leqslant f(n)\}$$

Ω — *асимптотическое* $\geqslant$. Например, $n \log n = \Omega(n \log n)$ и $n \log n = \Omega(n)$. В обоих случаях подходит $c_1 = 1$.

$$o(g(n)) = \{f(n) \mid \forall c_2 > 0 \exists n_0 : \forall n \geqslant n_0 \implies 0 \leqslant f(n) \leqslant c_2 g(n)\}$$

o — *асимптотическое* $<$. Например, $n = o(n \log n)$. Покажем это. Пусть $n < c_2 n \log n \iff 1 < c_2 \log n \iff n > 2^{1/c_2}$. Тогда $n_0 = [2^{1/c_2} + 1]$

$$\omega(g(n)) = \{f(n) \mid \forall c_1 > 0 \exists n_0 : \forall n \geqslant n_0 \implies 0 \leqslant c_1 g(n) \leqslant f(n)\}$$

ω — *асимптотическое* $>$. Например, нельзя сказать, что $n \log n = \omega(n \log n)$. Но можно сказать, что $n \log n = \omega(n)$.

Когда мы пишем такую нотацию, мы подразумеваем функции, а не числа. Если же указывать функции явно, то это можно сделать с помощью λ-нотации:

$$\lambda n.n \in o(\lambda n.n \log_2 n)$$

Примечание: данная нотация очень похожа на лямбда-функции в Python:

$$\texttt{lambda x: x * x} \iff \lambda x.x^2$$

Заметим, что в логарифмах можно свободно менять основание: $\log_c n = \frac{\log_2 n}{\log_2 c}$. Именно поэтому не пишут основание логарифма.

Ход действий при алгоритме **"разделяй и властвуй"**:

1. Разбить задачу на подзадачи.
2. Каждую подзадачу решить рекурсивно.
3. Объединяем решения подзадач некоторым образом.

Этот алгоритм даст решение общей задачи.

Вернёмся к *сортировке слиянием*. Алгоритм состоит из трёх шагов:

1. Раделить массив напополам — $\Theta(1)$
2. Рекурсивно решить подзадачи — $2T(\frac{n}{2})$
3. Слияние уже отсортированных массивов — $\Theta(n)$

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

Быстрая сортировка

Задача та же — отсортировать массив.

Воспользуемся методом “Разделяй и властвуй”. Разобьем по-другому:

Выберем в массиве опорный элемент x (как угодно). Выбор важен, от него может много зависить. Пройдем по всем элементам и запишем те элементы, что меньше x до него, а те, что больше — после.

Две подзадачи: сортировка двух подмассивов.

Третий шаг — соединить их.

Algorithm 8 Разбитие массива на подмассивы

```
1: function PARTITION(a, p, q)   ▷ a — массив, p и q — индексы начала и конца соотвественно
2:     i := p
3:     for j := p + 1 to q do
4:         if a[j] < a[p] then
5:             i := i + 1
6:             SWAP(a[i], a[j])
7:     return i
```

Рассмотрим работу алгоритма на примере массива $\{6, 3, 8, 7, 5, 1\}$:

1. $j = 1$. Так как $6 > 3$, то запускается тело цикла. Тогда $i = 1$ и 3 остаётся на месте.

2. $j = 2$. Так как $6 < 8$, то ничего не изменяется.

3. $j = 3$. Так как $6 < 7$, то ничего не изменяется.

4. $j = 4$. Так как $6 > 5$, то запускается тело цикла. Тогда $i = 2$ и числа 5 и 8 меняются местами.

6	3	**8**	7	**5**	1	$\longrightarrow$	6	3	**5**	7	**8**	1

5. $j = 5$. Так как $6 > 1$, то запускается тело цикла. Тогда $i = 3$ и 7 и 1 меняются местами.

6	3	5	**7**	8	**1**	$\longrightarrow$	6	3	5	**1**	8	**7**

6. Последний шаг — переставить опорный элемент на место i:

6	3	5	**1**	8	7	$\longrightarrow$	**1**	3	5	**6**	8	7

Теперь рассмотрим скорость работы алгоритма.

1. Разбить задачу на подзадачи — $\Theta(n)$

2. Рекурсивно решить подзадачи. Пусть индекс опорного элемента равен r. Тогда на выполнение уйдёт $T(r-1) + T(n-r)$.

3. Объединить решения задач в одно глобальное — 0 (уже сделано).

Тогда скорость работы алгоритма задаётся следующим рекуррентным соотношением:

$$T(n) = T(r-1) + T(n-r) + \Theta(n)$$

Рассмотрим возможные случаи:

1. **Оптимальный вариант** — r всегда посередине:

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

2. **Худший случай** — r всегда минимален/максимален (массив уже «почти» отсортирован):

$$T(n) = T(n-1) + \Theta(n) \implies T(n) = \Theta(n^2)$$

3. **Средний случай.** Пусть каждый раз обе части не меньше четверти.

   Подзадачей типа j называется задача такая, что размер входного массива n' соответсвтует следующему неравенству:

$$n\left(\frac{3}{4}\right)^{j+1} < n' \leqslant n\left(\frac{3}{4}\right)^{j}$$

   Не считая рекурсии, на каждую подзадачу типа j уходит $O\left(\left(\frac{3}{4}\right)^j n\right)$.

   Стоит заметить, что подзадачи типа j не пересекаются по разбиению. При этом из них получаются подзадачи типа не меньше $j+1$.

   При этом количество подзадач типа j не больше $\left(\frac{4}{3}\right)^{j+1}$. Отсюда получаем, что на все подзадачи типа j нужно $O\left(\left(\frac{4}{3}\right)^{j+1}\left(\frac{3}{4}\right)^j n\right) = O(n)$.

   Так как максимальный тип подзадачи можно ограничить сверху $\log_{4/3} n$, то оценка работы в среднем случае равна $O(n \log n)$. При условии, что «везёт» всегда.

Как обеспечить везение?

Мы хотим, чтобы опорный элемент был близок к середине (в отсортированном массиве). Условно, в пределах средних двух четвертей. Если выбирать наугад, вероятность 50%.

Предположим, что мы выбираем случайный элемент. Распределим все прочие, и если одна из частей меньше четверти, забудем про этот элемент и выберем другой. Повторим, пока не получим хороший элемент. В среднем на это уйдёт две попытки $\left(\frac{1}{P}\right)$. На сложности алгоритма это не сказывается никак, т.к. меняется только константа. Зато теперь так не только в лучшем, но и в среднем случае.