Сортировки. Семинар 9. 4 апреля 2019 г.

Подготовил: Горбунов Э.

Источники: [Кормен 1, гл. 2]; [Кормен 2, гл. 2]; [Кормен 1, §13]

Ключевые слова: пузырёк, быстрая сортировка (Quicksort), сортировка с помощью кучи (Heapsort), сортировка слиянием (Mergesort), цифровая сортировка, нижние оценки в модели разрешающих деревьев

Задача сортировки

Перед тем, как рассматривать конкретные примеры алгоритмов сортировки и оценивать их время работы, зафиксируем постановку задачи сортировки, а также договоримся о том, как мы будем оценивать время работы алгоритма сортировки.

Задача сортировки формулируется следующим образом. На вход подаётся N элементов $a_1, a_2, \ldots, a_N$ (например, это может быть массив чисел, картинок, имён и так далее). Каждый элемент a_i состоит из двух частей: некоторой информации и ключа k_i (например, в случае массива чисел ключ просто совпадает с k_i; в случае массива картинок информацией может быть сама картинка, а ключом — некоторый id этой картинки или имя файла; когда мы имеем дело со списком имён, ключами опять-таки могут быть сами имена). На множестве ключей введено отношение порядка «$<$» так, что для любых трёх значений ключей x, y, z выполняются два свойства:

1. закон трихотомии: либо $x < y$, либо $x > y$, либо $x = y$;
2. закон транзитивности: если $x < y$ и $y < z$, то $x < z$.

Грубо говоря, мы можем сравнивать элементы массива при помощи сравнения ключей.

Задача сортировки состоит в том, чтобы на выходе получить масиив $a_{i_1}, a_{i_2}, \ldots, a_{i_N}$, у которого $k_{i_1} \leqslant k_{i_2} \leqslant \ldots \leqslant k_{i_N}$. При этом в абстрактной задаче сортировки сами ключи могут быть любой природы. Единственное, на что мы можем рассчитывать, так это на то, что мы можем сравнивать ключи. Поэтому принято оценивать временную сложность алгоритма сортировки числом попарных сравнений (считается, что время работы в таком случае пропорционально числу сравнений).

Повторение: сортировка пузырьком

На первом семинаре мы уже сталкивались с такими алгоритмами как сортировка пузырьком (Bubblesort) и сортировка слиянием (Mergesort) и даже проводили анализ времени работы этих процедур. Для полноты картины мы сейчас зафиксируем эти знания.

Начнём с более простого алгоритма — сортировки пузырьком. Сразу договоримся, что массив элементов и массив ключей мы различать не будем, и под сравнением элементов мы будем понимать сравнение соответствующих ключей. Итак, пусть нам дан массив $A[1\,..\,N]$, который мы хотим отсортировать. Опишем процедуру сортировки пузырьком.

```
1: procedure BUBBLESORT(A)
2:     for i ← 1 to N − 1 do
3:         for j ← N to i + 1 do
4:             if A[j] < A[j − 1] then
5:                 A[j] ↔ A[j − 1]
6:             end if
7:         end for
8:     end for
```

9: **end procedure**

На каждой итерации внутреннего цикла (строки 4-5) происходит ровно одно сравнение. Итоговое число сравнений равняется $(N-1)+(N-2)+\ldots 1 = \frac{N(N-1)}{2} = O(N^2)$. Таким образом, временная сложность сортировки пузырьком есть $O(N^2)$. Отметим, что можно немного видоизменить процедуру, а именно, завести некоторую переменную, которая будет обнуляться перед выполнением внутреннего цикла и ей будет присваиваться значение 1, если хотя бы одно сравнение в строке 4 было верно (такая переменная называется флагом). Кроме того, нужно перед очередным выполнением внутреннего цикла проверять, равен ли флаг единице. Если вдруг он не равен единице, то это означает, что массив уже полностью отсортирован и можно завершить процедуру. Если рассматривать видоизменённую процедуру сортировки пузырьком, то её время работы в худшем случае будет равно $O(N^2)$, а время работы в лучшем случае будет равно $O(N)$ (в том случае, когда исходный массив уже отсортирован в нужно порядке).

Кроме того, отметим, что алгоритм сортировки пузырьком требует дополнительной памяти размера $O(1)$ (размер одного элемента).

Повторение: сортировка слиянием

Теперь опишем процедуру сортировки слиянием. Неформальное описание следующее: делим массив на две почти равные части, каждую из них сортируем, вызывая процедуру рекурсивно для каждой из частей, и соединяем отсортированные части за линейное время при помощи процедуры Merge, которую мы опишем далее. Данная идея записывается формально в виде следующей процедуры, которая принимает на вход часть массива $A[p\,..\,r]$ и сортирует его.

```
 1: procedure MERGESORT(A, p, r)
 2:     if p < r then
 3:         q ← ⌊(p+r)/2⌋
 4:         MERGESORT(A, p, q)
 5:         MERGESORT(A, q + 1, r)
 6:         MERGE(A, p, q, r)
 7:     end if
 8: end procedure
 9: procedure MERGE(A, p, q, r)
10:     i ← 0
11:     j ← 0
12:     while p + i < q и q + j < r do
13:         if A[p + i] < A[q + j] then
14:             B[i + j] ← A[p + i]                ▷ массив B — это результирующий массив
15:             i ← i + 1
16:         else
17:             B[i + j] ← A[q + j]
18:             j ← j + 1
19:         end if
20:     end while                                  ▷ На этом этапе один из массивов полностью пройден
21:     while p + i < q do      ▷ Этот цикл присоединит необработанные элементы первой части к результату
22:         B[i + j] ← A[p + i]
23:         i ← i + 1
24:     end while
25:     while q + j < r do      ▷ Этот цикл присоединит необработанные элементы второй части к результату
26:         B[i + j] ← A[q + j]
27:         j ← j + 1
28:     end while
29:     for k ← 0 to i + j do                      ▷ Запишем результат в массив A
30:         A[p + k] ← B[k]
31:     end for
32: end procedure
```

Нетрудно показать, что процедура MERGE работает за время $\Theta(n)$ для массива размера n. Если обозначить через $T(n)$ временную сложность работы алгоритма MERGESORT на массиве длины n, то можно получить следующее рекуррентное соотношение:

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n).$$

Как мы уже показывали на первом семинаре, это соотношение означает, что $T(n) = \Theta(n \log n)$. Отметим, что сортировка слиянием требует дополнительной памяти размера $O(n)$.

Сортировка с помощью кучи

Итак, мы убедились, что сортировка пузырьком работает за время $O(n^2)$, но при этом использует всего лишь $O(1)$ дополнительной памяти. Сортировка слиянием работает почти на порядок быстрее, а именно за время $O(n \log n)$, однако использует $O(n)$ дополнительной памяти, что на порядок больше, чем у сортировки пузырьком.

Оказывается, что можно предложить алгоритм сортировки, который использует структуру данных, называемую *двоичной кучей*, и который работает за время $O(n \log n)$ (как сортировка слиянием), используя при этом $O(1)$ дополнительной памяти (как сортировка пузырьком). Этот способ называется *сортировкой с помощью кучи* (Heapsort).

Двоичная куча — это массив с определёнными свойствами упорядоченности, которые мы опишем далее. Чтобы описать эти свойства нам будет удобно представлять массив в виде двоичного дерева[1]. Во-первых, каждая вершина дерева соответствует элементу массива. Во-вторых, если вершина имеет индекс i, то её родитель имеет индекс $\left\lfloor \frac{i}{2} \right\rfloor$, если он определён (отсюда, например, следует, что вершина с индексом 1 является корнем), а её дети — индексы $2i$ и $2i + 1$ (что следует из первой части утверждения). В дальнейшем нам будет удобно считать, что размер кучи может быть меньше длины массива, поэтому вместе с массивом A мы будем хранить его длину length$[A]$ и размер кучи heap-size$[A]$. Движение по дереву осуществляется следующими процедурами.

```
1: procedure PARENT(i)
2:     return ⌊i/2⌋
3: end procedure
4: procedure LEFT(i)
5:     return 2i
6: end procedure
7: procedure RIGHT(i)
8:     return 2i + 1
9: end procedure
```

Наконец, должно выполняться *основное свойство кучи*: для каждой вершины i, кроме корневой,

$$A[\text{PARENT}(i)] \geqslant A[i].$$

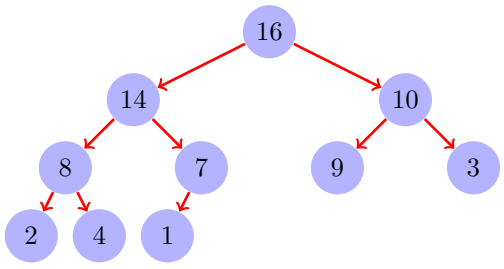

Рис. 1: Двоичная куча [16, 14, 10, 8, 7, 9, 3, 2, 4, 1], представленная в виде дерева.

[1] Это такое дерево, у каждой вершины которого существует не более двух потомков.

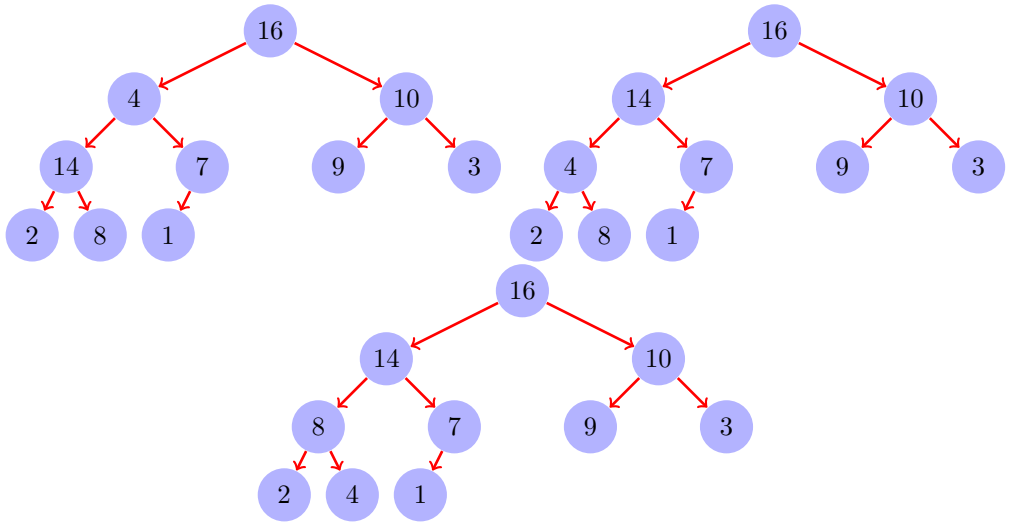

Рис. 2: Работа процедуры Heapify$(A, 2)$ при heap-size$[A] = 10$.

Высотой вершины дерева будем называть высоту поддерева, корнем которого является эта вершина (то есть число рёбер в самом длинном пути с началом в этой вершине вниз по дереву к листу). Заметим, что все уровни дерева, составляющего кучу, заполнены полностью, кроме, быть может, последнего уровня. Поэтому высота этого дерева равна $\Theta(\log n)$, где n — число элементов в куче. Как увидим далее, время работы основных операций над кучей пропорционально высоте дерева и, следовательно, составляет $O(\log n)$.

Одной из таких операций является процедура Heapify. Она позволяет сохранять основное свойство кучи. На вход эта процедура получает массив A и индекс i. Предполагается, что поддеревья с корнями Left(i) и Right(i) обладают основным свойством кучи. Идея проста: если основное свойство не выполнено для вершины i, то её следует поменять с наибольшим из её детей и так далее, пока элемент $A[i]$ не «погрузится» до нужного места.

```
1:  procedure Heapify(A, i)
2:      l ← Left(i)
3:      r ← Right(i)
4:      if l ⩽ heap-size[A] и A[l] > A[i] then
5:          largest ← l
6:      else
7:          largest ← i
8:      end if
9:      if r ⩽ heap-size[A] и A[r] > A[largest] then
10:         largest ← r
11:     end if
12:     if largest ≠ i then
13:         A[i] ↔ A[largest]
14:         Heapify(A, largest)
15:     end if
16: end procedure
```

В строках 4-10 в переменную largest помещается индекс наибольшего из элементов $A[i]$, $A[\text{Left}(i)]$ и $A[\text{Right}(i)]$. Если largest $= i$, то элемент уже «погрузился» до нужного места и работа процедуры закончена. Иначе процедура меняет местами $A[i]$ и $A[\text{largest}]$, что обеспечивает выполнение основного свойства кучи в вершине i, но, возможно, основное свойство могло нарушится в вершине largest, поэтому нужно вызвать процедуру Heapify$(A, \text{largest})$, чтобы восстановить основное свойство кучи в этой вершине.

Оценим время работы процедуры Heapify. На каждом шаге требуется совершить $\Theta(1)$ сравнений. Кроме того производится рекурсивный вызов. Если поддерево с корнем i содержит n элементов, то поддеревья Left(i) и

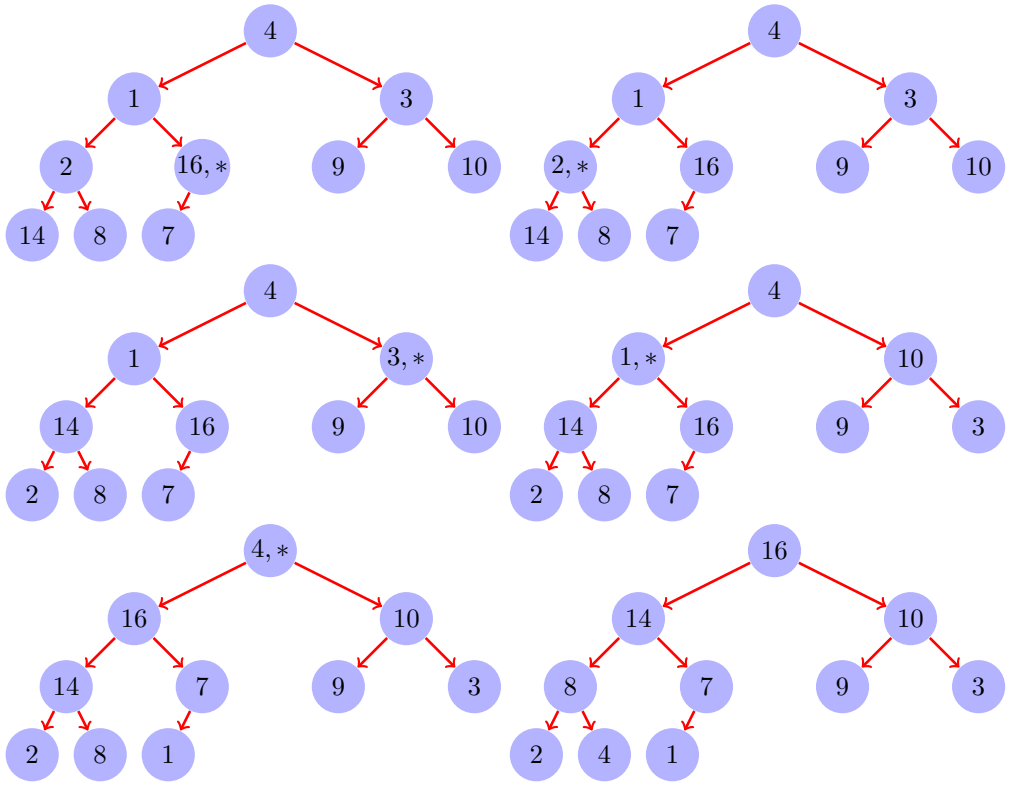

Рис. 3: Работа процедуры Build-Heap(A) для $A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$. На рисунке показано состояние данных перед каждым вызовом процедуры Heapify в строке 3. Знаком $*$ помечалась та вершина, для которой вызывалась процедура Heapify. Последняя картинка — результат работы процедуры Build-Heap.

Right(i) содержат не более $\frac{2n}{3}$ элементов (в силу того, что все уровни кучи, кроме, быть может, последнего, заполнены полностью, мы получаем, что наихудший случай — когда последний уровень для данного поддерева заполнен наполовину). Пусть $T(n)$ — время работы процедуры Heapify на поддереве размера n. Тогда

$$T(n) \leqslant T\left(\frac{2n}{3}\right) + \Theta(1).$$

Отсюда получаем, что $T(n) = O(\log n)$. Но это можно понять и из более простых соображений: на каждой итерации мы спускаемся по дерево на уровень вниз (либо завершаем работу). Поэтому итераций будет совершено не более $\log n$, то есть сравнений будет $O(\log n)$.

Теперь покажем, что из массива $A[1\,..\,n]$ можно за время $O(\log n)$ сделать двоичную кучу. Эту задачу решает процедура Build-Heap. Основная идея этой процедуры — вызывать процедуру Heapify в правильном порядке. Поскольку вершины $\left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n$ являются листьями, то поддеревья в этих вершинах по очевидным причинам удовлетворяют основному свойству кучи. Теперь для каждой из оставшихся вершин в порядке убывания индексов мы будем применять процедуру Heapify. Порядок обработки вершин гарантирует, что каждый раз условия вызова процедуры Heapify будут выполнены.

1: **procedure** Build-Heap(A)
2: heap-size[A] $\leftarrow$ length[A]
3: **for** $i \leftarrow \left\lfloor \frac{\text{length}[A]}{2} \right\rfloor$ **downto** 1 **do**
4: Heapify(A, i)
5: **end for**
6: **end procedure**

Оценим временную сложность работы процедуры Build-Heap. Во-первых, можно было бы сказать, что время работы равно $O(n \log n)$, поскольку время работы Heapify равно $O(\log n)$ и она вызывается не более n раз.

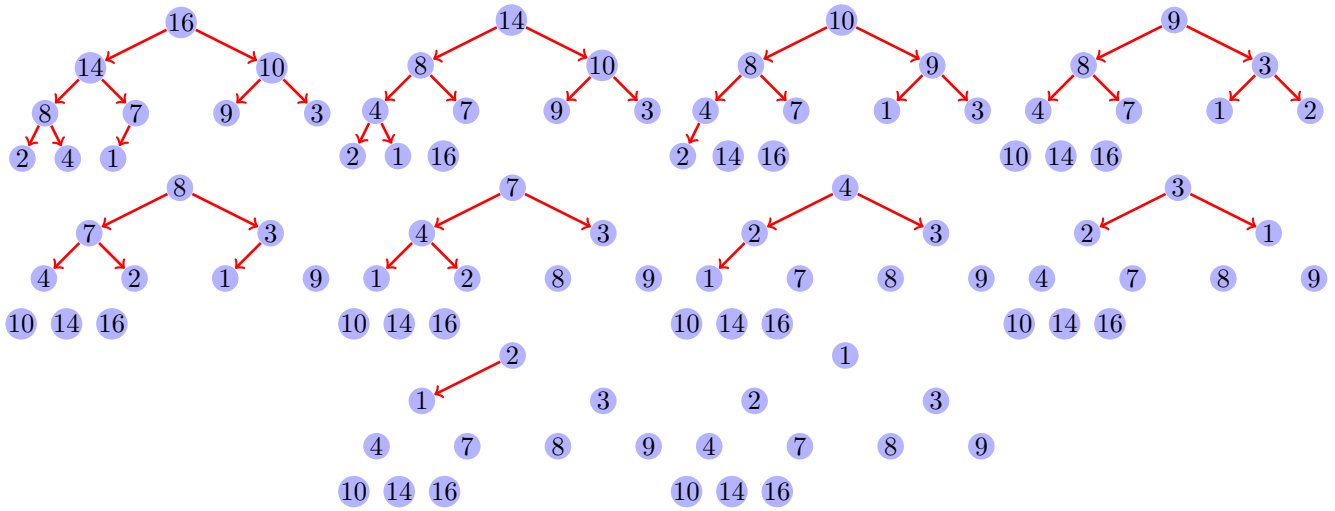

Рис. 4: Работа процедуры Heapsort(A) для $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$. На рисунке показано состояние данных перед каждым вызовом процедуры Heapify в строке 6.

Но такая оценка является на завышенной, множитель $\log n$ можно убрать.

Как мы отметили ранее, работа процедуры Heapify пропорциональна высоте дерева, для которого она вызывается. Кроме того, число вершин высоты h в двоичной куче из n элементов не превосходит $\left\lceil \frac{n}{2^{h+1}} \right\rceil$, а высота всей кучи не превосходит $\lfloor \log n \rfloor$. Поэтому время работы процедуры Build-Heap не превышает

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = O(n),$$

где последняя формула следует из

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} < \sum_{h=0}^{\infty} \frac{h}{2^h} = 2.$$

Равенство $\sum\limits_{h=0}^{\infty} \frac{h}{2^h} = 2$ было доказано на втором семинаре (см. пример вероятностного алгоритма типа «Монте-Карло»).

Теперь мы можем описать сортировку при помощи кучи (Heapsort). Она состоит из двух частей. Сначала вызывается процедура Build-Heap, чтобы сделать из массива кучу. В таком случае максимальный элемент массива будет находится в корне дерева (то есть первым). Соответственно, алгоритм меняет местами элементы $A[1]$ и $A[n]$, уменьшает размер кучи на единицу и восстанавливает основное свойство кучи в корневой вершине (так как поддеревья с корнями Left(i) и Right(i) не утратили основного свойства кучи, то это можно сделать с помощью процедуры Heapify). После этого в корне будет находиться максимальный из оставшихся элементов. Так делается до тех пор, пока в куче не останется один элемент.

```
1: procedure Heapsort(A)
2:     Build-Heap(A)
3:     for i ← length[A] downto 2 do
4:         A[1] ↔ A[i]
5:         heap-size[A] ← heap-size[A] − 1
6:         Heapify(A, 1)
7:     end for
8: end procedure
```

Время работы процедуры Heapsort равняется $O(n \log n)$: на выполнение процедуры Build-Heap требуется время $O(n)$, каждая итерация цикла требует времени $O(\log n)$, а итераций всего $n - 1$. При этом алгоритм требует лишь $O(1)$ дополнительной памяти.

Быстрая сортировка

Рассмотрим теперь вероятностный алгоритм сортировки массива под названием *быстрая сортировка* (Quicksort). Как мы увидим далее, этот алгоритм в худшем случае работает как сортировка пузырьком то есть $\Theta(n^2)$, однако математическое ожидание времени работы этого алгоритма равняется $\Theta(n \log n)$, причём множитель при $n \log n$ достаточно мал. Кроме того, быстрая сортировка требует только $O(1)$ дополнительной памяти. Поэтому этот алгоритм любят использовать на практике.

Однако начнём мы для простоты с детерминированного аналога процедуры Quicksort (и именно его будем называть Quicksort; вероятностный аналог мы будем называть Randomized-Quicksort). Быстрая сортировка основана на методе «разделяй и властвуй», а именно, сортировка участка $A[p\,..\,r]$ происходит следующим образом.

1. Элементы массива A переставляются так, чтобы любой из элементов участка $A[p\,..\,q]$ был не больше любого из элементов участка $A[q+1\,..\,r]$, где $p \leqslant q < r$. Эту операцию мы будем называть Partition. Мы с ней уже сталкивались, когда изучали поиск медианы за линейное время на первом семинаре.
2. Процедура сортировки вызывается рекурсивно для массивов $A[p\,..\,q]$ и $A[q+1\,..\,r]$.

В результате такой процедуры массив $A[p\,..\,r]$ будет отсортирован.

```
1: procedure Quicksort(A, p, r)
2:     if p < r then
3:         q ← Partition(A, p, r)
4:         Quicksort(A, p, q)
5:         Quicksort(A, q + 1, r)
6:     end if
7: end procedure
```

Чтобы отсортировать весь массив необходимо выполнить процедуру $\textsc{Quicksort}(A, 1, \text{length}[A])$.

Основной шаг данного алгоритма состоит в разбиении массива на две части. Рассмотрим следующую процедуру.

```
1: procedure Partition(A, p, r)
2:     x ← A[p]
3:     i ← p − 1
4:     j ← r + 1
5:     while True do
6:         repeat j ← j − 1
7:         until A[j] ⩽ x
8:         repeat i ← i + 1
9:         until A[i] ⩾ x
10:        if i < j then
11:            A[i] ↔ A[j]
12:        else
13:            return j
14:        end if
15:    end while
16: end procedure
```

В качестве граничного элемента берётся $x = A[p]$. Далее массив проходится с двух концов на встречу друг другу. Если при проходе слева направо встречается элемент не меньший x, то мы останавливаемся на нём. Аналогично, если при проходе справа налево встречается элемент не больший x, то мы останавливаемся на нём. Если $i < j$, то, элементы $A[i]$ и $A[j]$ нужно поменять местами (устранить коллизию). Когда все элементы будут обработаны, то индекс j будет границей левой части массива. Так как каждый элемент мы сравниваем с x не более двух раз и сравниваем числа i и j тоже не более n раз, то время работы процедуры Partition равняется $\Theta(n)$.

Время работы процедуры Quicksort зависит от того, насколько хорошо мы разделяем массив на две части. Нетрудно показать, что если разделение происходит примерно на равные части, то рекуррентное соотношение на время работы быстрой сортировки будет иметь вид

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

из которого следует, что $T(n) = O(n \log n)$. Это соответствует наилучшему разбиению. Наихудший случай наступает при условии, что при каждом вызове процедуры Partition массив делится на две части размеров 1 и $n-1$ соответственно. В таком случае рекуррентное соотношение получается следующим:

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

откуда

$$T(n) = \sum_{k=1}^{n} \Theta(k) = \Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2).$$

Более того, заметим, что если на каждом шаге массив делится на части размера αn и $(1-\alpha)n$, где $\alpha \in (0,1)$, то рекуррентное соотношение будет следующим:

$$T(n) = T(\alpha n) + T((1-\alpha)n) + \Theta(n),$$

откуда получаем, что $T(n) = \Theta(n \log n)$ (эту оценку можно показать, например, по индукции).

Мы видим, что процедура Partition на некоторых входах будет работать долго, а на некоторых — быстро. Если же мы хотим, чтобы время работы процедуры не зависело от того, как расположены элементы во входе, можно рассмотреть вероятностный аналог этой процедуры. Например, можно делать следующее: выбирать каждый раз барьерный элемент случайным образом при помощи некоторой функции Random. На самом деле это эквивалентно тому, что просто сгенерировать некоторую случайную перестановку индексов (это можно сделать за $\Theta(n)$), а затем использовать её в процессе работы. В результате получим следующие процедуры.

```
1: procedure Randomized-Partition(A, p, r)
2:     i ← Random(p, r)
3:     A[p] ↔ A[i]
4:     return Partition(A, p, r)
5: end procedure
6: procedure Randomized-Quicksort(A, p, r)
7:     if p < r then
8:         q ← Randomized-Partition(A, p, r)
9:         Randomized-Quicksort(A, p, q)
10:        Randomized-Quicksort(A, q + 1, r)
11:    end if
12: end procedure
```

В худшем случае эта процедура по прежнему работает $\Theta(n^2)$. Интуитивно понятно, что с ненулевой вероятностью мы на каждой итерации будем делить массив наиболее несбалансированно. На самом деле это можно даже строго обосновать (посмотреть, как это делается можно в книге Кормена).

Анализ среднего времени работы мы проведём в случае, когда все элементы массива различны. Тогда заметим, что значение q, которое вернёт процедура Partition зависит только от того, сколько в массиве элементов не больших $x = A[p]$ (число таких элементов будем называть рангом и обозначать через $\mathrm{rank}(x)$). Если ранг больше единицы, то левая часть разбиения будет содержать $\mathrm{rank}(x) - 1$ элементов. Если же $\mathrm{rank}(x) = 1$, то левая часть будет содержать один элемент. Так как в качестве барьерного может быть выбран любой элемент массива, то значение $\mathrm{rank}(x)$ принимает каждое из значений $1, 2, \ldots, n$ с одинаковой вероятностью $\frac{1}{n}$. Отсюда следует, что левая часть будет содержать $2, 3, \ldots, n-1$ элементов с вероятностью $\frac{1}{n}$, а с вероятностью $\frac{2}{n}$ — один элемент. Пусть $T(n)$ — это математическое ожидание времени работы процедуры Randomized-Quicksort, то есть $T(n) = \mathbb{E}[T(A)]$, где $T(A)$ — время работы Randomized-Quicksort на массиве A (это случайная величина). Если расписать $T(n)$ по формуле полного математического ожидания, то мы получим

$$T(n)\frac{1}{n}\left(2T(1) + 2T(n-1) + \sum_{q=2}^{n-1}(T(q) + T(n-q))\right) + \Theta(n).$$

Поскольку $T(1) = \Theta(1)$ и $T(n-1) = O(n^2)$, то

$$\frac{1}{n}\left(T(1) + T(n-1)\right) = \frac{1}{n}\left(\Theta(1) + O(n^2)\right) = O(n),$$

поэтому можно записать рекуррентное соотношение в следующем виде:

$$T(n) = \frac{1}{n}\sum_{q=1}^{n-1}\left(T(q) + T(n-q)\right) + \Theta(n).$$

Так как каждое слагаемое в скобке встречается ровно два раза, то

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n).$$

Покажем индукцией по n, что $T(n) \leqslant an\log n + b$, где $a > 0$ и $b > 0$ — некоторые константы, которые будут подобраны далее. При $n = 1$ утверждение верно, если взять достаточно большое b. Теперь докажем шаг индукции. При $n > 1$ имеем

$$\begin{aligned} T(n) &= \tfrac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \\ &\leqslant \tfrac{2}{n}\sum_{k=1}^{n-1}(ak\log k + b) + \Theta(n) \\ &= \tfrac{2a}{n}\sum_{k=1}^{n-1} k\log k + \tfrac{2b}{n}(n-1) + \Theta(n). \end{aligned}$$

Ниже мы покажем, что $\sum\limits_{k=1}^{n-1} k\log k \leqslant \frac{1}{2}n^2\log n - \frac{1}{8}n^2$. Используя это, получаем

$$\begin{aligned} T(n) &\leqslant \tfrac{2a}{n}\left(\tfrac{1}{2}n^2\log n - \tfrac{1}{8}n^2\right) + \tfrac{2b}{n}(n-1) + \Theta(n) \\ &\leqslant an\log n - \tfrac{a}{4}n + 2b + \Theta(n) \\ &= an\log n + b + \left(\Theta(n) + b - \tfrac{a}{4}n\right) \leqslant an\log n + b, \end{aligned}$$

где последнее неравенство верно, если взять a достаточно большим. Отсюда получаем, что $T(n) = O(n\log n)$.

Осталось показать, что $\sum\limits_{k=1}^{n-1} k\log k \leqslant \frac{1}{2}n^2\log n - \frac{1}{8}n^2$. Так как при $k < \left\lceil\frac{n}{2}\right\rceil$ имеем $\log k \leqslant \log n - 1$. Поэтому

$$\begin{aligned} \sum_{k=1}^{n-1} k\log k &\leqslant (\log n - 1)\sum_{k=1}^{\left\lceil\frac{n}{2}\right\rceil-1} k + \log n\sum_{k=\left\lceil\frac{n}{2}\right\rceil}^{n-1} k = \log n\sum_{k=1}^{n-1} k - \sum_{k=1}^{\left\lceil\frac{n}{2}\right\rceil-1} k \\ &\leqslant \tfrac{1}{2}n(n-1)\log n - \tfrac{1}{2}\left(\tfrac{n}{2} - 1\right)\tfrac{n}{2} \\ &\leqslant \tfrac{1}{2}n^2\log n - \tfrac{1}{8}n^2. \end{aligned}$$

Нижние оценки для сортировки

Итак, мы до этого рассмотрели 3 алгоритма сортировки с временной сложностью $O(n\log n)$ и один алгоритм с временной сложностью $\Theta(n^2)$. На фоне времени $O(n^2)$ время $O(n\log n)$ выглядит достаточно симпатично. Но оптимальное ли (асимптотически) это время для сортировки, основанной на сравнениях? Оказывается, что да и доказать это можно с помощью так называемых *разрешающих деревьев*.

Любой детерминированный алгоритм сортировки попарными сравнениями можно представить в виде двоичного дерева. Действительно, пусть мы сортируем n элементов $a_1, a_2, \ldots, a_n$. Каждая внутренняя вершина (то есть не листовая вершина) соответствует операции сравнения двух элементов a_i и a_j и снабжена пометкой $a_i ? a_j$. Каждый лист разрешающего дерева снабжён пометкой $(\pi(1), \pi(2), \ldots, \pi(n))$, где π — перестановка n элементов. Рёбра дерева снабжены пометками «$\leqslant$» и «$>$».

Каждому алгоритму сортировки соответствует своё разрешающее дерево (и наоборот). Чтобы по дереву получить алгоритм сортировки, нужно идти по дереву от корня к листьям и сравнивать элементы, соответствующие

текущей вершине: если в текущей вершине написано $a_i ? a_j$, то нужно сравнить a_i и a_j, и если $a_i \leqslant a_j$, то нужно пойти налево, а в противном случае — направо. Если пришли в лист, в котором записана перестановка π, то нужно вернуть результат $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$. Время работы алгоритма сортировки равно длине соответствующего пути в разрешающем дереве.

Каждая из $n!$ перестановок должна появиться хотя бы в одном листе дерева (правильный алгоритм сортировки должен предусматривать все возможные порядки). По формуле Стирлинга $n! \geqslant \left(\frac{n}{e}\right)^n$. Двоичное дерево высоты h имеет не более 2^h листьев, поэтому высота разрешающего дерева сортировки должна быть не меньше $\log(n!) \geqslant n \log n - n \log e = \Omega(n \log n)$.

Отсюда следует, что время работы $O(n \log n)$ для алгоритма сортировки, основанного на сравнениях, является асимптотически оптимальным.