Рандомизированные алгоритмы

Время работы зависит не только от самого входа, но и от рандома.

Мы в качестве $T^*(n)$ берём среднее арифметичекое по всем возможным входам с весами в виде вероятности.

То есть - информация, которая позволит оценить, сколько мы ожмдаем прождать, запустив на некоторых *случайных* данных. По факту — считаем матожидание.

$$T^*(n) = \sum_{perm} T(perm) \times probablity(perm)$$

Quick sort (алгоритм Хоара)

Идея: на каждом шагу берём некий элемент, близкий к медианному, затем все элементы, которые больше него, отправляем в одну часть, остальные - в другую.

Стандартная модификация — каждый раз берём случайный элемент.

Тогда

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

(с вероятностью $\frac{1}{3}$ мы попадём в центральную треть (в отсортированном порядке), которая даже в худшем случае позволяет нам более или менее нормально разбить массив)

Докажем по индукции, что $T^*(n) = n \log n$:

…

Но не хочется каждый раз выделять память, поэтому делаем всё “in-place”

```
sort(&a, l, r):
    if (r - l <= 1):
        return
    x = a[rand(l, r - 1)]
    m = split(l, r, x)
    sort(a, l, m)
    sort(a, m, r)
```

```
split(&a, l, r, x):
    m = l
    for i = l..r-1:
        if a[i] < x:
            swap(a[i], a[m++])
    return m
```

k-порядковая статистика

k-я порядковая статистика - k-й элемент, если отсортировать массив.

Ещё один алгоритм Хоара

Будем использовать процедуру рассечения массива элементов из алгоритма сортировки QuickSort. Пусть нам надо найти k-ую порядковую статистику, а после рассечения опорный элемент встал на позицию m

. Возможно три случая:

- **k = m**. Порядковая статистика найдена.
- **k < m**. Рекурсивно ищем k-ую статистику в первой части массива.
- **k > m**. Рекурсивно ищем $(k - m - 1)$-ую статистику во второй части массива.

Код

partition - разделяет подмассив [l, r) шкуворнем и возвращает его, шкворня, индекс.

```
int findOrderStatistic(int[] array, int k) {
  int left = 0, right = array.length;
  while (true) {
    int mid = partition(array, left, right);

    if (mid == k) {
      return array[mid];
    }
    else if (k < mid) {
      right = mid;
    }
    else {
      left = mid + 1;
    }
  }
}
```

Заметим, что он работает за $O(n)$: $T^*(n) = n + \frac{1}{3}T^*(\frac{2}{3}n) + \frac{2}{3}T^*(n)$

Алгоритм пяти мужиков: Блюма, Флойда, Пратта, Ривеста и Тарьяна

> Бывает такое, что лучший известный алгоритм для некоторой задачи работает за асимптотически лучшее время, чем лучший из известных детерминированных. Но эти ***мужики*** собрались с силами и решили исправить это для случая сортировки...

Алгоритм пытается за $O(n)$ найти нормальный k-ю порядковую статистику. Для этого что-

- Делим на блоки по 5 элементов
- Сортируем каждое по возрастанию
- Берём медиану в каждом блоке
- Найдём более или менее медианный элемент среди этих медианных элементов
- Как? Конечно же, так же - рекурсивно!

Утверждается, что мы получим хороший x.

Найдём время работы алгоритма (последний член берётся из худшего случая (мы можем утверждать только то, что разделительный элемент больше больше, чем $\frac{3}{10}n$ элементов, то есть в худшем случае мы запустимся от $\frac{7}{10}n$ элементов)):

$$T(n) \leqslant n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)$$

А это линейное время.

Тут константа похуже, чем в рандомизированом… Но зато никакого рандома! Можно ночью спать спокойно.

Почему блоки по 5? Это минимальное нечётное число, которое подходит. Подробнее - в ДЗ.

Можно ли сортировать быстрее, чем за $n \log n$

Обычно доказательство такого очень сложная. (Сложно доказать, что чего-то не может существовать).

Но тут не очень сложно.

Докажем, что нужно сделать как минимум $n \log n$ сравнений.

В зависимости от сравнения можно менять слеющие действия.

Построим дерево таким образом.

Разных данных есть $n!$ штук, и их все нужно соритровать.

Тогда высота дерева $\approx log(n!) = \sum_{i=1}^{n} \log i \geq \frac{n}{2} \log \frac{n}{2} = \Omega(n \log n)$.

Но это работает только если мы умеем только сравнивать.

Но, например, с числами можно делать более полезные вещи, например, реализовав bucket sort.

> HINT: Можно доказывать невозможность решения задачи за некоторое время, сказав, что тогда можно было бы отсортировать быстрее, чем за $n \log n$