# Алгоритмы и структуры данных

## Лекция 5. Сортировки II.

Артур Кулапин

ВШПИ МФТИ

3 октября 2023 г.

# О чем разговор?

1. Быстрая сортировка `QuickSort`
   - Классическая версия
2. Порядковые статистики
   - От сортировки к статистикам
   - Линейный алгоритм
3. Сортировки, основанные не только на сравнениях
   - Сортировка подсчетом
   - Стабильные сортировки

# Схема алгоритма

1. *Как-то* (позднее обговорим как) выбрать опорный элемент `pivot`.

# Схема алгоритма

1. *Как-то* (позднее обговорим как) выбрать опорный элемент `pivot`.
2. Произвести процедуру `Partition`: все, что меньше `pivot` левее него, а все, что больше — правее.

# Схема алгоритма

1. *Как-то* (позднее обговорим как) выбрать опорный элемент `pivot`.
2. Произвести процедуру `Partition`: все, что меньше `pivot` левее него, а все, что больше — правее.
3. Вызвать рекурсивно от правой и левой половин.

# Схема алгоритма

1. *Как-то* (позднее обговорим как) выбрать опорный элемент `pivot`.
2. Произвести процедуру `Partition`: все, что меньше `pivot` левее него, а все, что больше — правее.
3. Вызвать рекурсивно от правой и левой половин.

Использует быстрая сортировка доппамять?

# Схема алгоритма

1. *Как-то* (позднее обговорим как) выбрать опорный элемент `pivot`.
2. Произвести процедуру `Partition`: все, что меньше `pivot` левее него, а все, что больше — правее.
3. Вызвать рекурсивно от правой и левой половин.

Использует быстрая сортировка доппамять? Какое время работы?

# Схема алгоритма

1. *Как-то* (позднее обговорим как) выбрать опорный элемент `pivot`.
2. Произвести процедуру `Partition`: все, что меньше `pivot` левее него, а все, что больше — правее.
3. Вызвать рекурсивно от правой и левой половин.

Использует быстрая сортировка доппамять? Какое время работы?

**Теорема.** Среднее время работы быстрой сортировки при случайном выборе `pivot` составит $\mathcal{O}(N \log N)$, а доппамять логарифмична.

# Процедура Partition

1. Заведем два указателя на левую и правую границы диапазона $i$ и $j$ соответственно.
2. Двигаем $i$ вперед, пока не будет найден элемент, превосходящий по значению `pivot`, аналогично двигается $j$ назад.
3. Меняем местами $array[i]$ и $array[j]$, повторяем шаги 2-3, пока указатели не встретятся.

# Процедура Partition

```
pair<int, int> Partition(int[] array, int l, int r) {
  int pivot = ChoosePivot(array, l, r);
  int i = l, j = r;
  while (i <= j) {
    while (array[i] < pivot) { ++i; }
    while (array[j] > pivot) { --j; }
    if (i >= j) { break; }
    Swap(array[i], array[j]); ++i; --j;
  }
  return {j, i};
}
```

# Выбираем `Pivot`

Как уже было отмечено, в худшем случае время работы квадратично.

# Выбираем `Pivot`

Как уже было отмечено, в худшем случае время работы квадратично.

Надо научиться искать «хороший» `pivot`.

# Выбираем `Pivot`

Как уже было отмечено, в худшем случае время работы квадратично.

Надо научиться искать «хороший» `pivot`.

**Утверждение.** Пусть есть способ выбирать `pivot` так, что он делит массив каждый раз в соотношении не хуже, чем $p : q$. Тогда время работы быстрой сортировки в худшем случае $O(N \log N)$.

# Выбираем `Pivot`

Как уже было отмечено, в худшем случае время работы квадратично.

Надо научиться искать «хороший» `pivot`.

**Утверждение.** Пусть есть способ выбирать `pivot` так, что он делит массив каждый раз в соотношении не хуже, чем $p : q$. Тогда время работы быстрой сортировки в худшем случае $O(N \log N)$.

**Proof.** Надо рассмотреть дерево рекурсии и посмотреть на самую длинную потенциально возможную ветвь.

# Порядковые статистики

**Def.** $k$-я порядковая статистика массива — такой элемент, что после сортировки массива он окажется на $k$-м месте.

**Def.** Медианой массива называют его $N/2$-ю порядковую статистику.

# Порядковые статистики

**Def.** $k$-я порядковая статистика массива — такой элемент, что после сортировки массива он окажется на $k$-м месте.

**Def.** Медианой массива называют его $N/2$-ю порядковую статистику.

Получается, оптимальнее всего брать медиану в качестве `pivot`.

# Рандомизированная версия

1. Выбрать опорный элемент
2. Провести разбиение

# Рандомизированная версия

1. Выбрать опорный элемент
2. Провести разбиение
3. Если индекс опорного элемента $i$ окажется больше данного $k$, то нам надо искать слева $k$-ю порядковую, в случае равенства завершаемся, в противном случае надо искать $(i - k - 1)$-ю порядковую

# Рандомизированная версия

1. Выбрать опорный элемент
2. Провести разбиение
3. Если индекс опорного элемента $i$ окажется больше данного $k$, то нам надо искать слева $k$-ю порядковую, в случае равенства завершаемся, в противном случае надо искать $(i - k - 1)$-ю порядковую

**Теорема (б/д).** Время работы этого алгоритма при случайном выборе `pivot` линейно.

# Медиана медиан

1. Разобьем массив на пятерки элементов.

# Медиана медиан

1. Разобьем массив на пятерки элементов.
2. В каждой пятерке выберем медиану (вторую порядковую статистику) руками. Получили массив медиан.

# Медиана медиан

1. Разобьем массив на пятерки элементов.
2. В каждой пятерке выберем медиану (вторую порядковую статистику) руками. Получили массив медиан.
3. Для массива медиан найдем его медиану. Найденный элемент назовем медианой медиан.

# Медиана медиан

**Утверждение.** Медиана медиан гарантированно делит массив в соотношении не хуже чем $3 : 7$.

# Медиана медиан

**Утверждение.** Медиана медиан гарантированно делит массив в соотношении не хуже чем $3 : 7$.

**Proof.** Сначала определим нижнюю границу для количества элементов, превышающих по величине опорный элемент $x$. В общем случае как минимум половина медиан, найденных на втором шаге, больше или равны медианы медиан $x$.

# Медиана медиан

**Утверждение.** Медиана медиан гарантированно делит массив в соотношении не хуже чем 3 : 7.

**Proof.** Сначала определим нижнюю границу для количества элементов, превышающих по величине опорный элемент $x$. В общем случае как минимум половина медиан, найденных на втором шаге, больше или равны медианы медиан $x$.

Таким образом, как минимум $\frac{N}{10}$ групп содержат по 3 элемента, превышающих величину $x$, за исключением группы, в которой меньше 5 элементов и ещё одной группы, содержащей сам элемент $x$.

# Медиана медиан

**Утверждение.** Медиана медиан гарантированно делит массив в соотношении не хуже чем $3 : 7$.

**Proof.** Сначала определим нижнюю границу для количества элементов, превышающих по величине опорный элемент $x$. В общем случае как минимум половина медиан, найденных на втором шаге, больше или равны медианы медиан $x$.

Таким образом, как минимум $\frac{N}{10}$ групп содержат по 3 элемента, превышающих величину $x$, за исключением группы, в которой меньше 5 элементов и ещё одной группы, содержащей сам элемент $x$.

Таким образом получаем, что количество элементов больших $x$ не менее $\frac{3N}{10}$.

# Линейный поиск порядковой статистики

1. Найдем медиану медиан массива.

# Линейный поиск порядковой статистики

1. Найдем медиану медиан массива.
2. Возьмем ее в качестве `pivot`.
3. Проведем `Partition`.

# Линейный поиск порядковой статистики

1. Найдем медиану медиан массива.
2. Возьмем ее в качестве `pivot`.
3. Проведем `Partition`.
4. Если индекс опорного элемента $i$ окажется больше данного $k$, то нам надо искать слева $k$-ю порядковую, в случае равенства завершаемся, в противном случае надо искать $(i - k - 1)$-ю порядковую.

# Линейный поиск порядковой статистики

**Утверждение.** Данный алгоритм работает в худшем случае за $O(N)$ времени.

**Proof pt 1.** У нас есть три составляющих работы алгоритма на каждом шаге:

# Линейный поиск порядковой статистики

**Утверждение.** Данный алгоритм работает в худшем случае за $O(N)$ времени.

**Proof pt 1.** У нас есть три составляющих работы алгоритма на каждом шаге:

1. Время на разделение массива на пятерки и сортировка каждой из них: $cN$.

# Линейный поиск порядковой статистики

**Утверждение.** Данный алгоритм работает в худшем случае за $O(N)$ времени.

**Proof pt 1.** У нас есть три составляющих работы алгоритма на каждом шаге:

1. Время на разделение массива на пятерки и сортировка каждой из них: $cN$.
2. Время на поиск медианы медиан $T\left(\frac{N}{5}\right)$.

# Линейный поиск порядковой статистики

**Утверждение.** Данный алгоритм работает в худшем случае за $O(N)$ времени.

**Proof pt 1.** У нас есть три составляющих работы алгоритма на каждом шаге:

1. Время на разделение массива на пятерки и сортировка каждой из них: $cN$.
2. Время на поиск медианы медиан $T\left(\frac{N}{5}\right)$.
3. Время на поиск $k$-й порядковой не превзойдет времени его поиска в большей доле, то есть $T\left(\frac{7N}{10}\right)$.

# Линейный поиск порядковой статистики

**Утверждение.** Данный алгоритм работает в худшем случае за $O(N)$ времени.

**Proof pt 1.** У нас есть три составляющих работы алгоритма на каждом шаге:

1. Время на разделение массива на пятерки и сортировка каждой из них: $cN$.
2. Время на поиск медианы медиан $T\left(\frac{N}{5}\right)$.
3. Время на поиск $k$-й порядковой не превзойдет времени его поиска в большей доле, то есть $T\left(\frac{7N}{10}\right)$.

# Линейный поиск порядковой статистики

**Proof pt 2.** Таким образом,

$$T(N) = T\left(\frac{N}{5}\right) + T\left(\frac{7N}{10}\right) + cN$$

# Линейный поиск порядковой статистики

**Proof pt 2.** Таким образом,

$$T(N) = T\left(\frac{N}{5}\right) + T\left(\frac{7N}{10}\right) + cN$$

Покажем по индукции, что $T(N) \leq 10cN$.

# Линейный поиск порядковой статистики

**Proof pt 2.** Таким образом,

$$T(N) = T\left(\frac{N}{5}\right) + T\left(\frac{7N}{10}\right) + cN$$

Покажем по индукции, что $T(N) \leq 10cN$.

Подставим это соотношение и получим, что

$$T(N) = T\left(\frac{N}{5}\right) + T\left(\frac{7N}{10}\right) + cN \leq \frac{10cN}{5} + \frac{7 \cdot 10cN}{10} + cN = 10cN$$

# Детерминированная быстрая сортировка

1. Найдем медиану с помощью алгоритма выше.
2. Проведем по ней `Partition`.
3. Рекурсивно запустимся от каждой из частей.

# Детерминированная быстрая сортировка

1. Найдем медиану с помощью алгоритма выше.
2. Проведем по ней `Partition`.
3. Рекурсивно запустимся от каждой из частей.

**Утверждение.** Время работы такой быстрой сортировки составит $O(N \log N)$ в худшем случае.

# Итого

Мы закончили обсуждать сортировки, основанные на сравнениях.

Приведем краткое саммери

| Сортировка | Среднее время | Худшее время | Дин. память |
|---|---|---|---|
| Пирамидальная | $\mathcal{O}(N \log N)$ | $\mathcal{O}(N \log N)$ | $\mathcal{O}(1)$ |
| Быстрая | $\mathcal{O}(N \log N)$ | $\mathcal{O}(N^2)$ | $\mathcal{O}(1)$ |
| Быстрая + ММ | $\mathcal{O}(N \log N)$ | $\mathcal{O}(N \log N)$ | $\mathcal{O}(1)$ |
| Слиянием | $\mathcal{O}(N \log N)$ | $\mathcal{O}(N \log N)$ | $\mathcal{O}(N)$ |

# Counting Sort

Пусть известно, что множество объектов, массив из которых нам придется сортировать, невелико по размеру.

# Counting Sort

Пусть известно, что множество объектов, массив из которых нам придется сортировать, невелико по размеру.

Тогда давайте посчитаем, сколько раз встречается каждый элемент, а потом выведем его нужное число раз.

# Counting Sort

Пусть известно, что множество объектов, массив из которых нам придется сортировать, невелико по размеру.

Тогда давайте посчитаем, сколько раз встречается каждый элемент, а потом выведем его нужное число раз.

За сколько это работает?

# Counting Sort

Пусть известно, что множество объектов, массив из которых нам придется сортировать, невелико по размеру.

Тогда давайте посчитаем, сколько раз встречается каждый элемент, а потом выведем его нужное число раз.

За сколько это работает?

Пусть множество сортируемых элементов имеет размер $K$, тогда время равно $\mathcal{O}(N + K)$ и доппамять $\mathcal{O}(K)$.

# Стабильные сортировки

**Def.** Сортировка стабильная, если одинаковые с точки зрения компаратора элементы не меняют своего относительного расположения.

# Стабильные сортировки

**Def.** Сортировка стабильная, если одинаковые с точки зрения компаратора элементы не меняют своего относительного расположения.

Какие из уже рассмотренных сортировок стабильные?

# Стабильные сортировки

**Def.** Сортировка стабильная, если одинаковые с точки зрения компаратора элементы не меняют своего относительного расположения.

Какие из уже рассмотренных сортировок стабильные?

Ответ: только `MergeSort`.

# Stable counting sort

Пусть имеется массив $A$ длины $N$ — исходный и массив $B$ — куда будем писать ответ. Также заведем массив $P$ размера $K$, где $K$ — размер множества элементов.

1. Пройдем по массиву $A$ и запишем в $P[i]$ число объектов с ключом $i$.

# Stable counting sort

Пусть имеется массив $A$ длины $N$ — исходный и массив $B$ — куда будем писать ответ. Также заведем массив $P$ размера $K$, где $K$ — размер множества элементов.

1. Пройдем по массиву $A$ и запишем в $P[i]$ число объектов с ключом $i$.
2. Посчитаем префиксные суммы массива $P$, тогда мы знаем, начиная с какого индекса массива $B$ надо писать структуру с ключом $i$.

# Stable counting sort

Пусть имеется массив $A$ длины $N$ — исходный и массив $B$ — куда будем писать ответ. Также заведем массив $P$ размера $K$, где $K$ — размер множества элементов.

1. Пройдем по массиву $A$ и запишем в $P[i]$ число объектов с ключом $i$.
2. Посчитаем префиксные суммы массива $P$, тогда мы знаем, начиная с какого индекса массива $B$ надо писать структуру с ключом $i$.
3. Идем по массиву $A$ и пишем в нужное место, поддерживая, куда писать следующий элемент с ключом $k$.

# LSD

Алгоритм Least Significant Digit сортирует целые беззнаковые числа за линейное время.

Как мы знаем, *unsigned int* хранятся в виде строки из четырех байтов.

Тогда давайте будем сортировать побайтово данные числа как двоичные строки, сначала сортируем по убыванию последнего байта, потом стабильно по убванию второго байта и так далее. После такого получим, что массив чисел отсортирован.

# LSD

Алгоритм Least Significant Digit сортирует целые беззнаковые числа за линейное время.

Как мы знаем, *unsigned int* хранятся в виде строки из четырех байтов.

Тогда давайте будем сортировать побайтово данные числа как двоичные строки, сначала сортируем по убыванию последнего байта, потом стабильно по убванию второго байта и так далее. После такого получим, что массив чисел отсортирован.

Работает это за $\mathcal{O}(Nk)$ времени, где $k$ это число байтов или 4, то есть за $\mathcal{O}(N)$.