

# Сортировка

Никита Гаевой (Б09, Б10)

Иван Казменко (Б01)

Владислав Макаров (Б02, Б03)

Семён Петров (Б05, Б06)

Санкт-Петербургский Государственный Университет

Четверг, 29 сентября 2022 года

# Содержание

- 1 Алгоритмы сортировки
  - Постановка задачи
  - Быстрая сортировка
  - Сортировка слиянием
  - Сортировка с помощью кучи
- 2 Алгоритмы в смежных задачах
  - Мотивация
  - Порядковая статистика
  - Подсчёт количества инверсий
  - Двоичная куча
- 3 Примеры
  - Сортировка длинных чисел
  - Сортировка точек
  - Интерактивная сортировка

# Содержание

- 1 Алгоритмы сортировки
  - Постановка задачи
  - Быстрая сортировка
  - Сортировка слиянием
  - Сортировка с помощью кучи
- 2 Алгоритмы в смежных задачах
  - Мотивация
  - Порядковая статистика
  - Подсчёт количества инверсий
  - Двоичная куча
- 3 Примеры
  - Сортировка длинных чисел
  - Сортировка точек
  - Интерактивная сортировка

# Постановка задачи

## Постановка задачи:

- Есть массив из  $n$  объектов:  $a_0, a_1, \dots, a_{n-1}$ .
- К примеру, это могут быть числа, пары чисел, строки, последовательности, ...
- Для каждой пары объектов  $(x, y)$  известно, верно ли, что  $x < y$ .
- Нужно расположить эти объекты в порядке неубывания.
- Формально, нужно найти такую перестановку  $p_0, p_1, \dots, p_{n-1}$ , что  $a_{p_0} \leq a_{p_1} \leq \dots \leq a_{p_{n-1}}$ .

## Критерии качества:

- Время работы:  $O(n^2)$ ,  $O(n \log n)$ ,  $O(n)$ , ...
- Дополнительно используемая память:  $O(n)$ ,  $O(1)$ , ...
- Устойчивость: сохраняется ли исходный порядок для равных объектов.

# Постановка задачи

## Постановка задачи:

- Есть массив из  $n$  объектов:  $a_0, a_1, \dots, a_{n-1}$ .
- К примеру, это могут быть числа, пары чисел, строки, последовательности, ...
- Для каждой пары объектов  $(x, y)$  известно, верно ли, что  $x < y$ .
- Нужно расположить эти объекты в порядке неубывания.
- Формально, нужно найти такую перестановку  $p_0, p_1, \dots, p_{n-1}$ , что  $a_{p_0} \leq a_{p_1} \leq \dots \leq a_{p_{n-1}}$ .

## Критерии качества:

- Время работы:  $O(n^2)$ ,  $O(n \log n)$ ,  $O(n)$ , ...
- Дополнительно используемая память:  $O(n)$ ,  $O(1)$ , ...
- Устойчивость: сохраняется ли исходный порядок для равных объектов.

# Быстрая сортировка

Идея — алгоритм Q (QuickSort):

- Q1. Выберем один элемент  $x = a_k$ .
- Q2. Поставим  $x$  на то место, где он окажется после сортировки.
- Q3. Поставим слева от  $x$  все элементы  $\leq x$ , а справа все элементы  $\geq x$ .
- Q4. Отсортируем рекурсивно получившиеся левую и правую части.

Как выполнить пункты Q2 и Q3 — алгоритм P (Partition):

- P1. Найдём  $y$  — самый левый элемент, не меньший  $x$ .
- P2. Найдём  $z$  — самый правый элемент, не больший  $x$ .
- P3. Если  $y$  стоит левее  $z$ , поменяем местами  $y$  и  $z$  и запустим алгоритм P для отрезка массива между ними.
- P4. В противном случае работа алгоритма P завершена.

Заметим, что алгоритм P работает за линейное время от длины массива.

# Быстрая сортировка

Идея — алгоритм Q (QuickSort):

- Q1. Выберем один элемент  $x = a_k$ .
- Q2. Поставим  $x$  на то место, где он окажется после сортировки.
- Q3. Поставим слева от  $x$  все элементы  $\leq x$ , а справа все элементы  $\geq x$ .
- Q4. Отсортируем рекурсивно получившиеся левую и правую части.

Как выполнить пункты Q2 и Q3 — алгоритм P (Partition):

- P1. Найдём  $y$  — самый левый элемент, не меньший  $x$ .
- P2. Найдём  $z$  — самый правый элемент, не больший  $x$ .
- P3. Если  $y$  стоит левее  $z$ , поменяем местами  $y$  и  $z$  и запустим алгоритм P для отрезка массива между ними.
- P4. В противном случае работа алгоритма P завершена.

Заметим, что алгоритм P работает за линейное время от длины массива.

# Быстрая сортировка

Идея — алгоритм Q (QuickSort):

- Q1. Выберем один элемент  $x = a_k$ .
- Q2. Поставим  $x$  на то место, где он окажется после сортировки.
- Q3. Поставим слева от  $x$  все элементы  $\leq x$ , а справа все элементы  $\geq x$ .
- Q4. Отсортируем рекурсивно получившиеся левую и правую части.

Как выполнить пункты Q2 и Q3 — алгоритм P (Partition):

- P1. Найдём  $y$  — самый левый элемент, не меньший  $x$ .
- P2. Найдём  $z$  — самый правый элемент, не больший  $x$ .
- P3. Если  $y$  стоит левее  $z$ , поменяем местами  $y$  и  $z$  и запустим алгоритм P для отрезка массива между ними.
- P4. В противном случае работа алгоритма P завершена.

Заметим, что алгоритм P работает за линейное время от длины массива.

# Быстрая сортировка: Hoare partition

```
quick_sort (a, l, r):  
· if r - l <= 1:  
·   return  
· x = a[random (l, r)]  
· m = partition (a, l, r, x)  
· quick_sort (a, l, m)  
· quick_sort (a, m, r)
```

```
partition (a, l, r, x):  
· i = l, j = r - 1  
· while i <= j:  
·   while a[i] < x:  
·     i += 1  
·   while x < a[j]:  
·     j -= 1  
·   if i >= j:  
·     break  
·   swap (a[i], a[j])  
·   i += 1, j -= 1  
· return j
```

# Быстрая сортировка: Hoare partition

```
quick_sort (a, l, r):  
· if r - l <= 1:  
·   return  
· x = a[random (l, r)]  
· m = partition (a, l, r, x)  
· quick_sort (a, l, m)  
· quick_sort (a, m, r)
```

```
partition (a, l, r, x):  
· i = l, j = r - 1  
· while i <= j:  
·   while a[i] < x:  
·     i += 1  
·   while x < a[j]:  
·     j -= 1  
·   if i >= j:  
·     break  
·   swap (a[i], a[j])  
·   i += 1, j -= 1  
· return j
```

# Быстрая сортировка: альтернативный код

```
Q    quick_sort (a, l, r):  
    · if r - l <= 1:  
    · · return  
Q1   · x = a[random (l, r)]  
    · i = l,  j = r - 1  
P    · while i <= j:  
P1   · · while a[i] < x:  
P1   · · · i += 1  
P2   · · while x < a[j]:  
P2   · · · j -= 1  
P4   · · if i >= j:  
P4   · · · break  
P3   · · swap (a[i], a[j])  
P3   · · i += 1,  j -= 1  
Q4   · quick_sort (a, l, i)  
Q4   · quick_sort (a, j, r)
```

# Быстрая сортировка: альтернативный код

```
Q    quick_sort (a, l, r):  
    · if r - l <= 1:  
    · · return  
Q1   · x = a[random (l, r)]  
    · i = l, j = r - 1  
P    · while i <= j:  
P1   · · while a[i] < x:  
P1   · · · i += 1  
P2   · · while x < a[j]:  
P2   · · · j -= 1  
P4   · · if i >= j:  
P4   · · · break  
P3   · · swap (a[i], a[j])  
P3   · · i += 1, j -= 1  
Q4   · quick_sort (a, l, i)  
Q4   · quick_sort (a, j, r)
```

# Быстрая сортировка: Lomuto partition

```
quick_sort (a, l, r):  
· if r - l <= 1:  
·   return  
· x = a[random (l, r)]  
· ml, mr = partition (a, l, r, x)  
· quick_sort (a, l, ml)  
· quick_sort (a, mr, r)
```

```
partition (a, l, r, x):  
· i = l  
· while i < r:  
·   if a[i] < x:  
·     swap (a[l++], a[i++])  
·   else if x < a[i]:  
·     swap (a[--r], a[i])  
·   else:  
·     i += 1  
· return l, r
```

# Быстрая сортировка: анализ

Общие наблюдения:

- На каждом уровне вложенности рекурсии алгоритма  $Q$  отрезки, на которых он запускается, не имеют общих точек.
- Алгоритм  $P$  работает за линейное время от длины отрезка.
- Общее время работы —  $O(n \cdot d)$ , где  $d$  — максимальный уровень вложенности рекурсии.

# Быстрая сортировка: анализ

Общие наблюдения:

- На каждом уровне вложенности рекурсии алгоритма  $Q$  отрезки, на которых он запускается, не имеют общих точек.
- Алгоритм  $P$  работает за линейное время от длины отрезка.
- Общее время работы —  $O(n \cdot d)$ , где  $d$  — максимальный уровень вложенности рекурсии.

Лучший случай:

- Пусть каждый отрезок делится алгоритмом  $P$  поровну.
- Тогда  $d = \log_2 n$  и общее время работы —  $O(n \log n)$ .
- На практике сложно выбрать  $x$  так, чтобы поделить отрезок поровну.

# Быстрая сортировка: анализ

Общие наблюдения:

- На каждом уровне вложенности рекурсии алгоритма  $Q$  отрезки, на которых он запускается, не имеют общих точек.
- Алгоритм  $P$  работает за линейное время от длины отрезка.
- Общее время работы —  $O(n \cdot d)$ , где  $d$  — максимальный уровень вложенности рекурсии.

Худший случай:

- Пусть каждый отрезок делится алгоритмом  $P$  на часть из одного элемента и часть из всех оставшихся.
- Тогда  $d = n - 1$  и общее время работы —  $O(n^2)$ .
- Например, так будет, если все числа различны, а в качестве  $x$  на каждом отрезке выбирается либо самый маленький, либо самый большой элемент этого отрезка.

# Быстрая сортировка: анализ

Общие наблюдения:

- На каждом уровне вложенности рекурсии алгоритма  $Q$  отрезки, на которых он запускается, не имеют общих точек.
- Алгоритм  $P$  работает за линейное время от длины отрезка.
- Общее время работы —  $O(n \cdot d)$ , где  $d$  — максимальный уровень вложенности рекурсии.

Средний случай:

- Пусть выбор  $x$  случаен.
- Тогда с вероятностью  $\geq \frac{1}{2}$  в каждом отрезке длины  $n$  выбирается разделитель, попадающий в позиции  $[\frac{1}{4}n; \frac{3}{4}n]$ .
- Значит, в среднем каждое второе разделение делит отрезок в отношении не более  $3 : 1$  (бóльшая часть имеет длину  $\leq \frac{3}{4}n$ ).
- Поэтому средняя глубина рекурсии будет не больше  $2 \cdot \log_{\frac{4}{3}} n$ .

# Сортировка слиянием

Идея — алгоритм M (MergeSort):

- M1.** Поделим массив на две равные по длине части.
- M2.** Отсортируем рекурсивно левую и правую части.
- M3.** Из двух отсортированных половин получим отсортированный массив.

Как выполнить пункт M3 — алгоритм MS (Merge Segments):

- MS1.** Рассмотрим первые числа левой и правой частей.
- MS2.** Выберем из них минимальное, допишем его к ответу и удалим из соответствующей части.
- MS3.** Если одна из частей пуста, допишем к ответу другую часть целиком.
- MS4.** В противном случае перейдём к шагу MS1.

Заметим, что алгоритм MS работает за линейное время от длины массива.

# Сортировка слиянием

Идея — алгоритм М (MergeSort):

- М1. Поделим массив на две равные по длине части.
- М2. Отсортируем рекурсивно левую и правую части.
- М3. Из двух отсортированных половин получим отсортированный массив.

Как выполнить пункт М3 — алгоритм MS (Merge Segments):

- MS1. Рассмотрим первые числа левой и правой частей.
- MS2. Выберем из них минимальное, допишем его к ответу и удалим из соответствующей части.
- MS3. Если одна из частей пуста, допишем к ответу другую часть целиком.
- MS4. В противном случае перейдём к шагу MS1.

Заметим, что алгоритм MS работает за линейное время от длины массива.

# Сортировка слиянием

Идея — алгоритм M (MergeSort):

- M1.** Поделим массив на две равные по длине части.
- M2.** Отсортируем рекурсивно левую и правую части.
- M3.** Из двух отсортированных половин получим отсортированный массив.

Как выполнить пункт M3 — алгоритм MS (Merge Segments):

- MS1.** Рассмотрим первые числа левой и правой частей.
- MS2.** Выберем из них минимальное, допишем его к ответу и удалим из соответствующей части.
- MS3.** Если одна из частей пуста, допишем к ответу другую часть целиком.
- MS4.** В противном случае перейдём к шагу MS1.

Заметим, что алгоритм MS работает за линейное время от длины массива.

# Сортировка слиянием: код

```
merge_sort (a, l, r):  
· if r - l <= 1:  
· · return  
· m = (l + r) / 2  
· merge_sort (a, l, m)  
· merge_sort (a, m, r)  
· merge_segments (a, l, m, r)
```

```
merge_segments (a, l, m, r):  
· i = l, j = m, k = l  
· while i < m or j < r:  
· · if j >= r or (i < m and a[i] <= a[j]):  
· · · b[k++] = a[i++]  
· · else:  
· · · b[k++] = a[j++]  
· a[l..r] = b[l..r]
```

# Сортировка слиянием: код

```
merge_sort (a, l, r):  
· if r - l <= 1:  
· · return  
· m = (l + r) / 2  
· merge_sort (a, l, m)  
· merge_sort (a, m, r)  
· merge_segments (a, l, m, r)
```

```
merge_segments (a, l, m, r):  
· i = l, j = m, k = l  
· while i < m or j < r:  
· · if j >= r or (i < m and a[i] <= a[j]):  
· · · b[k++] = a[i++]  
· · else:  
· · · b[k++] = a[j++]  
· a[l..r] = b[l..r]
```

# Сортировка слиянием: альтернативный код

```
M    merge_sort (a, l, r):  
    · if r - l <= 1:  
    · · return  
M1   · m = (l + r) / 2  
M2   · merge_sort (a, l, m)  
M2   · merge_sort (a, m, r)  
    · i = l, j = m, k = l  
MS   · while i < m or j < r:  
MS1  · · if j >= r or (i < m and a[i] <= a[j]):  
MS2  · · · b[k++] = a[i++]  
    · · else:  
MS2  · · · b[k++] = a[j++]  
MS4  · a[l..r] = b[l..r]
```

# Сортировка слиянием: анализ

## Общие наблюдения:

- На каждом уровне вложенности рекурсии алгоритма  $M$  отрезки, на которых он запускается, не имеют общих точек.
- Алгоритм  $MS$  работает за линейное время от длины отрезка.
- Общее время работы —  $O(n \cdot d)$ , где  $d$  — максимальный уровень вложенности рекурсии.

## Специфика алгоритма:

- Каждый отрезок делится алгоритмом  $MS$  поровну.
- Значит,  $d = \log_2 n$  и общее время работы —  $O(n \log n)$ .

## Используемая память:

- Заметим, что для массива  $b$  требуется  $O(n)$  дополнительной памяти.

# Сортировка слиянием: анализ

## Общие наблюдения:

- На каждом уровне вложенности рекурсии алгоритма  $M$  отрезки, на которых он запускается, не имеют общих точек.
- Алгоритм  $MS$  работает за линейное время от длины отрезка.
- Общее время работы —  $O(n \cdot d)$ , где  $d$  — максимальный уровень вложенности рекурсии.

## Специфика алгоритма:

- Каждый отрезок делится алгоритмом  $MS$  поровну.
- Значит,  $d = \log_2 n$  и общее время работы —  $O(n \log n)$ .

## Используемая память:

- Заметим, что для массива  $b$  требуется  $O(n)$  дополнительной памяти.

# Сортировка слиянием: анализ

Общие наблюдения:

- На каждом уровне вложенности рекурсии алгоритма  $M$  отрезки, на которых он запускается, не имеют общих точек.
- Алгоритм  $MS$  работает за линейное время от длины отрезка.
- Общее время работы —  $O(n \cdot d)$ , где  $d$  — максимальный уровень вложенности рекурсии.

Специфика алгоритма:

- Каждый отрезок делится алгоритмом  $MS$  поровну.
- Значит,  $d = \log_2 n$  и общее время работы —  $O(n \log n)$ .

Используемая память:

- Заметим, что для массива  $b$  требуется  $O(n)$  дополнительной памяти.

# Сортировка с помощью кучи

Двоичная куча — это массив  $a[0..n)$ , в котором выполнены соотношения  $a[k] \geq a[2k + 1]$  и  $a[k] \geq a[2k + 2]$  (свойство кучи) для всех  $k$ , для которых существуют соответствующие пары.

Идея — решим задачу в два этапа:

Н. Сначала преобразуем массив в двоичную кучу.

НА. Затем из двоичной кучи сделаем отсортированный массив.

# Сортировка с помощью кучи

Двоичная куча — это массив  $a[0..n)$ , в котором выполнены соотношения  $a[k] \geq a[2k + 1]$  и  $a[k] \geq a[2k + 2]$  (свойство кучи) для всех  $k$ , для которых существуют соответствующие пары.

Идея — решим задачу в два этапа:

**Н.** Сначала преобразуем массив в двоичную кучу.

**НА.** Затем из двоичной кучи сделаем отсортированный массив.

# Сортировка с помощью кучи: алгоритм

Алгоритм Н (Heapify):

**Н1.** Рассматривая вершины от последней к первой, последовательно добьёмся того, чтобы для рассматриваемой вершины, а также для всех вершин с бóльшим номером, было выполнено свойство кучи.

Алгоритм НА (HeapToArray):

**НА1.** Извлечём из кучи наибольший элемент (это всегда элемент с номером 0).

**НА2.** Поменяем его местами с последним элементом кучи.

**НА3.** Уменьшим размер кучи на 1, а последний элемент объявим элементом итогового массива.

**НА4.** Восстановим для первого элемента свойство кучи.

**НА5.** Если куча ещё не пуста, перейдём к шагу НА1.

# Сортировка с помощью кучи: алгоритм

Алгоритм Н (Heapify):

**Н1.** Рассматривая вершины от последней к первой, последовательно добьёмся того, чтобы для рассматриваемой вершины, а также для всех вершин с бóльшим номером, было выполнено свойство кучи.

Алгоритм НА (HeapToArray):

**НА1.** Извлечём из кучи наибольший элемент (это всегда элемент с номером 0).

**НА2.** Поменяем его местами с последним элементом кучи.

**НА3.** Уменьшим размер кучи на 1, а последний элемент объявим элементом итогового массива.

**НА4.** Восстановим для первого элемента свойство кучи.

**НА5.** Если куча ещё не пуста, перейдём к шагу НА1.

# Сортировка с помощью кучи: подзадача

Подзадача для алгоритмов Н и НА:

- Даны массив  $a[0..n)$  и номер элемента в нём  $k$ .
- Известно, что для всех элементов с большим номером свойство кучи выполнено.
- Требуется сделать так, чтобы оно было выполнено и для  $a[k]$ .

Решение — алгоритм SD (SiftDown):

SD1. Если  $2k + 1 \geq n$ , свойство кучи выполнено автоматически.

SD2. В противном случае выберем из  $a[2k + 1]$  и  $a[2k + 2]$  наибольший элемент  $x$ .

SD3. Если  $a[k] \geq x$ , свойство кучи выполнено.

SD4. Иначе поменяем местами  $a[k]$  и  $x$ , после чего запустим алгоритм SD для номера элемента  $x$ .

# Сортировка с помощью кучи: подзадача

Подзадача для алгоритмов Н и НА:

- Даны массив  $a[0..n)$  и номер элемента в нём  $k$ .
- Известно, что для всех элементов с большим номером свойство кучи выполнено.
- Требуется сделать так, чтобы оно было выполнено и для  $a[k]$ .

Решение — алгоритм SD (SiftDown):

SD1. Если  $2k + 1 \geq n$ , свойство кучи выполнено автоматически.

SD2. В противном случае выберем из  $a[2k + 1]$  и  $a[2k + 2]$  наибольший элемент  $x$ .

SD3. Если  $a[k] \geq x$ , свойство кучи выполнено.

SD4. Иначе поменяем местами  $a[k]$  и  $x$ , после чего запустим алгоритм SD для номера элемента  $x$ .

# Сортировка с помощью кучи: код

```
def heapify (a, n):  
    · for i = (n - 1) / 2 downto 0:  
        · · sift_down (a, i, n)  
  
def sift_down (a, i, n):  
    · y = a[i]  
    · while True:  
        · · j = i * 2 + 1  
        · · if j >= n:  
            · · · break  
        · · if j + 1 < n and a[j] < a[j + 1]:  
            · · · j += 1  
        · · if y >= a[j]:  
            · · · break  
        · · a[i] = a[j]  
        · · i = j  
    · a[i] = y
```

```
def heap_to_array (a, n):  
    · for i = n - 1 downto 1:  
        · · swap (a[0], a[i])  
        · · sift_down (a, 0, i)
```

# Сортировка с помощью кучи: код

```
heapify (a, n):  
· for i = (n - 1) / 2 downto 0:  
· · sift_down (a, i, n)  
  
sift_down (a, i, n):  
· y = a[i]  
· while True:  
· · j = i * 2 + 1  
· · if j >= n:  
· · · break  
· · if j + 1 < n and a[j] < a[j + 1]:  
· · · j += 1  
· · if y >= a[j]:  
· · · break  
· · a[i] = a[j]  
· · i = j  
· a[i] = y  
  
heap_to_array (a, n):  
· for i = n - 1 downto 1:  
· · swap (a[0], a[i])  
· · sift_down (a, 0, i)
```

# Сортировка с помощью кучи: код

```
heapify (a, n):  
· for i = (n - 1) / 2 downto 0:  
· · sift_down (a, i, n)  
  
sift_down (a, i, n):  
· y = a[i]  
· while True:  
· · j = i * 2 + 1  
· · if j >= n:  
· · · break  
· · if j + 1 < n and a[j] < a[j + 1]:  
· · · j += 1  
· · if y >= a[j]:  
· · · break  
· · a[i] = a[j]  
· · i = j  
· a[i] = y  
  
heap_to_array (a, n):  
· for i = n - 1 downto 1:  
· · swap (a[0], a[i])  
· · sift_down (a, 0, i)
```

# Сортировка с помощью кучи: анализ

Общие наблюдения:

- Алгоритм SD работает за  $O(\log n)$ : на каждом шаге номер элемента увеличивается хотя бы вдвое.
- Алгоритм HA работает за  $O(n \log n)$ : он  $n - 1$  раз вызывает алгоритм SD для кучи из  $n, n - 1, \dots, 2$  элементов.
- Алгоритм H работает за  $O(n \log n)$ : он  $n/2$  раз вызывает алгоритм SD.
- Общее время работы —  $O(n \log n)$ .

На самом деле алгоритм H работает за  $O(n)$ .

# Сортировка с помощью кучи: анализ

Общие наблюдения:

- Алгоритм SD работает за  $O(\log n)$ : на каждом шаге номер элемента увеличивается хотя бы вдвое.
- Алгоритм HA работает за  $O(n \log n)$ : он  $n - 1$  раз вызывает алгоритм SD для кучи из  $n, n - 1, \dots, 2$  элементов.
- Алгоритм H работает за  $O(n \log n)$ : он  $n/2$  раз вызывает алгоритм SD.
- Общее время работы —  $O(n \log n)$ .

На самом деле алгоритм H работает за  $O(n)$ .

# Сортировка с помощью кучи: анализ

На самом деле алгоритм  $H$  работает за  $O(n)$ .

Подробный анализ алгоритма  $H$ :

- Элементов кучи, для которых алгоритм  $SD$  делает много шагов, немного.
- Для  $\frac{n}{4}$  элементов с номерами  $\frac{n}{2} - 1, \frac{n}{2} - 2, \dots, \frac{n}{4}$  он делает не более одного шага.
- Для  $\frac{n}{8}$  элементов с номерами  $\frac{n}{4} - 1, \frac{n}{4} - 2, \dots, \frac{n}{8}$  он делает не более двух шагов.
- ...
- Максимальное количество шагов ( $\log n$ ) алгоритм может сделать только для элемента с номером 0.

# Сортировка с помощью кучи: анализ

На самом деле алгоритм  $H$  работает за  $O(n)$ .

$$\begin{aligned}
 \frac{n}{4} + 2 \cdot \frac{n}{8} + 3 \cdot \frac{n}{16} + \dots &= \\
 \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \dots + &(\leq \frac{n}{2}) \\
 \frac{n}{8} + \frac{n}{16} + \dots + &(\leq \frac{n}{4}) \\
 \frac{n}{16} + \dots + &(\leq \frac{n}{8}) \\
 \dots &\leq \\
 \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots &\leq n
 \end{aligned}$$

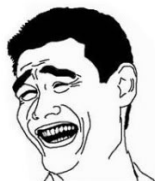
Тем не менее, вся сортировка работает за  $O(n \log n)$ .

# Содержание

- 1 Алгоритмы сортировки
  - Постановка задачи
  - Быстрая сортировка
  - Сортировка слиянием
  - Сортировка с помощью кучи
- 2 Алгоритмы в смежных задачах
  - Мотивация
  - Порядковая статистика
  - Подсчёт количества инверсий
  - Двоичная куча
- 3 Примеры
  - Сортировка длинных чисел
  - Сортировка точек
  - Интерактивная сортировка

# Мотивация

|                        |                                   |
|------------------------|-----------------------------------|
| C++:                   | <code>std::sort (a, a + n)</code> |
| Java:                  | <code>Arrays.sort (a)</code>      |
| C#, Visual Basic .NET: | <code>Array.sort (a)</code>       |
| Kotlin, Python, Rust:  | <code>a.sort ()</code>            |
| D, PascalABC .NET:     | <code>sort (a)</code>             |
| Haskell:               | <code>sort a</code>               |



```
int cmp (const void * a, const void * b)
{return * (int *) a - * (int *) b;}
```

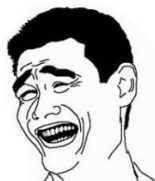
...

C: `qsort (a, n, sizeof (int), cmp)`

Можно использовать библиотечную функцию сортировки.  
Зачем знать, как она работает?

# Мотивация

|                        |                                   |
|------------------------|-----------------------------------|
| C++:                   | <code>std::sort (a, a + n)</code> |
| Java:                  | <code>Arrays.sort (a)</code>      |
| C#, Visual Basic .NET: | <code>Array.sort (a)</code>       |
| Kotlin, Python, Rust:  | <code>a.sort ()</code>            |
| D, PascalABC .NET:     | <code>sort (a)</code>             |
| Haskell:               | <code>sort a</code>               |



```
int cmp (const void * a, const void * b)
{return * (int *) a - * (int *) b;}
```

...

C: `qsort (a, n, sizeof (int), cmp)`

Можно использовать библиотечную функцию сортировки.  
Зачем знать, как она работает?

# Мотивация

Различные алгоритмы сортировки, в среднем работающие за время  $O(n \log n)$ , обладают разными достоинствами и недостатками:

- QuickSort:
  - в худшем случае работает за  $O(n^2)$ ;
  - + на практике является самым быстрым в среднем случае.
- MergeSort:
  - требует  $O(n)$  дополнительной памяти;
  - + легко может быть модифицирован для параллельных вычислений;
  - + является устойчивым.
- HeapSort:
  - плохо сочетается с кэшированием;
  - + требует  $O(n \log n)$  времени и  $O(1)$  дополнительной памяти в худшем случае.

# Мотивация

Кроме того, подходы и идеи, использованные в этих сортировках, оказываются полезны и в других задачах:

- QuickSort: нахождение  $k$ -й порядковой статистики за  $O(n)$ .
- MergeSort: подсчёт количества инверсий в перестановке за  $O(n \log n)$ .
- HeapSort: структура данных, обеспечивающая добавление элемента за  $O(\log n)$ , поиск максимума за  $O(1)$  и удаление максимума за  $O(\log n)$ .

# Порядковая статистика

Постановка задачи:

- Дан массив  $a$  размера  $n$  и число  $k$  ( $1 \leq k \leq n$ ).
- Нужно найти  $k$ -ю порядковую статистику массива  $a$ , то есть элемент, который после сортировки окажется в позиции  $k$ .

Идея — алгоритм QS (QuickSelect):

QS1. Выберем один элемент  $x = a_t$ .

QS2. Поставим  $x$  на то место, где он окажется после сортировки; пусть это оказалась позиция  $m$ .

QS3. Поставим слева от  $x$  все элементы  $\leq x$ , а справа все элементы  $\geq x$ .

QS4. Если  $k \leq m$ , найдём алгоритмом QS  $k$ -й элемент на отрезке массива от 1 до  $m$ .

QS5. В противном случае найдём алгоритмом QS  $(k - m)$ -й элемент на отрезке массива от  $m + 1$  до  $n$ .

# Порядковая статистика

Постановка задачи:

- Дан массив  $a$  размера  $n$  и число  $k$  ( $1 \leq k \leq n$ ).
- Нужно найти  $k$ -ю порядковую статистику массива  $a$ , то есть элемент, который после сортировки окажется в позиции  $k$ .

Идея — алгоритм QS (QuickSelect):

- QS1. Выберем один элемент  $x = a_t$ .
- QS2. Поставим  $x$  на то место, где он окажется после сортировки; пусть это оказалась позиция  $m$ .
- QS3. Поставим слева от  $x$  все элементы  $\leq x$ , а справа все элементы  $\geq x$ .
- QS4. Если  $k \leq m$ , найдём алгоритмом QS  $k$ -й элемент на отрезке массива от 1 до  $m$ .
- QS5. В противном случае найдём алгоритмом QS  $(k - m)$ -й элемент на отрезке массива от  $m + 1$  до  $n$ .

# Порядковая статистика

Как выполнить пункты QS2 и QS3 — алгоритм P (Partition):

- P1. Найдём  $y$  — самый левый элемент, не меньший  $x$ .
- P2. Найдём  $z$  — самый правый элемент, не больший  $x$ .
- P3. Если  $y$  стоит левее  $z$ , поменяем местами  $y$  и  $z$  и запустим алгоритм P для отрезка массива между ними.
- P4. В противном случае работа алгоритма P завершена.

Заметим, что алгоритм P работает за линейное время от длины массива.

# Порядковая статистика

Как выполнить пункты QS2 и QS3 — алгоритм P (Partition):

- P1. Найдём  $y$  — самый левый элемент, не меньший  $x$ .
- P2. Найдём  $z$  — самый правый элемент, не больший  $x$ .
- P3. Если  $y$  стоит левее  $z$ , поменяем местами  $y$  и  $z$  и запустим алгоритм P для отрезка массива между ними.
- P4. В противном случае работа алгоритма P завершена.

Заметим, что алгоритм P работает за линейное время от длины массива.

# Порядковая статистика: код

```
quick_select (a, k, l, r):  
· if r - l == 1:  
· · return l  
· x = a[random (l, r)]  
· m = partition (a, l, r, x)  
· if k < m:  
· · return quick_select (a, k, l, m)  
· else:  
· · return quick_select (a, k, m, r)
```

```
partition (a, l, r, x):  
· i = l, j = r - 1  
· while i <= j:  
· · while a[i] < x:  
· · · i += 1  
· · while x < a[j]:  
· · · j -= 1  
· · if i >= j:  
· · · break  
· · swap (a[i], a[j])  
· · i += 1, j -= 1  
· return i
```

# Порядковая статистика: код

```
quick_select (a, k, l, r):  
· if r - l == 1:  
· · return l  
· x = a[random (l, r)]  
· m = partition (a, l, r, x)  
· if k < m:  
· · return quick_select (a, k, l, m)  
· else:  
· · return quick_select (a, k, m, r)
```

```
partition (a, l, r, x):  
· i = l, j = r - 1  
· while i <= j:  
· · while a[i] < x:  
· · · i += 1  
· · while x < a[j]:  
· · · j -= 1  
· · if i >= j:  
· · · break  
· · swap (a[i], a[j])  
· · i += 1, j -= 1  
· return i
```

# Порядковая статистика: альтернативный код

```
QS    quick_select (a, k, l, r):  
    · while r - l > 1:  
QS1   · · x = a[random (l, r)]  
    · · i = l,  j = r - 1  
P     · · while i <= j:  
P1    · · · while a[i] < x:  i += 1  
P2    · · · while x < a[j]:  j -= 1  
P4    · · · if i >= j:  break  
P3    · · · swap (a[i], a[j])  
P3    · · · i += 1,  j -= 1  
QS4   · · if k < i:  
QS4   · · · r = i  
QS5   · · else:  
QS5   · · · l = i  
    · return l
```

# Порядковая статистика: альтернативный код

```
QS    quick_select (a, k, l, r):  
    · while r - l > 1:  
QS1   · · x = a[random (l, r)]  
    · · i = l,  j = r - 1  
P     · · while i <= j:  
P1    · · · while a[i] < x:  i += 1  
P2    · · · while x < a[j]:  j -= 1  
P4    · · · if i >= j:  break  
P3    · · · swap (a[i], a[j])  
P3    · · · i += 1,  j -= 1  
QS4   · · if k < i:  
QS4   · · · r = i  
QS5   · · else:  
QS5   · · · l = i  
    · return l
```

# Порядковая статистика: анализ

Общие наблюдения:

- Алгоритм QS отличается от алгоритма Q только тем, что рекурсивно запускается лишь от одного отрезка из двух — того, в котором содержится искомый элемент.
- Алгоритм P работает за линейное время от длины отрезка.
- Общее время работы пропорционально суммарной длине всех отрезков, на которых был запущен алгоритм.

# Порядковая статистика: анализ

Общие наблюдения:

- Алгоритм QS отличается от алгоритма Q только тем, что рекурсивно запускается лишь от одного отрезка из двух — того, в котором содержится искомый элемент.
- Алгоритм P работает за линейное время от длины отрезка.
- Общее время работы пропорционально суммарной длине всех отрезков, на которых был запущен алгоритм.

Лучший случай:

- Пусть каждый раз алгоритм рекурсивно запускается от меньшей из двух половин.
- Тогда сумма длин рассмотренных отрезков не превосходит  $n + \frac{n}{2} + \frac{n}{4} + \dots \leq 2n$ , поэтому общее время работы —  $O(n)$ .
- На практике сложно выбирать  $x$  так, чтобы спускаться в меньшую половину.

# Порядковая статистика: анализ

Худший случай:

- Пусть каждый отрезок делится алгоритмом  $P$  на часть из одного элемента и часть из всех оставшихся, в которую и нужно затем рекурсивно спускаться.
- Тогда общее время работы —  $O(n^2)$ .
- Например, так будет, если все числа различны, а в качестве  $x$  на каждом отрезке выбирается либо самый маленький, либо самый большой элемент этого отрезка, и при этом найти нужно тот элемент, который будет выбран последним.

# Порядковая статистика: анализ

Средний случай:

- Пусть выбор  $x$  случаен.
- Тогда с вероятностью  $\geq \frac{1}{2}$  в каждом отрезке длины  $n$  выбирается разделитель, попадающий в позиции  $[\frac{1}{4}n; \frac{3}{4}n]$ .
- Значит, в среднем каждое второе разделение делит отрезок в отношении не более  $3 : 1$  (бóльшая часть имеет длину  $\leq \frac{3}{4}n$ ).
- Поэтому, в какую бы половину мы ни спускались, в среднем за каждые два спуска мы уменьшаем длину текущего отрезка хотя бы на четверть.
- А значит, и общее время работы не превосходит  $O(n)$ , ведь  $2n + 2 \cdot \frac{3}{4}n + 2 \cdot \frac{9}{16}n + \dots \leq 2n \cdot 4$ .

# Подсчёт количества инверсий

Постановка задачи:

- Дана перестановка  $a$  размера  $n$ .
- Нужно найти количество пар индексов  $(i, j)$  таких, что  $i < j$  и  $a_i > a_j$ .

Идея — алгоритм CI (CountInversions):

- CI1. Поделим массив на две равные по длине части.
- CI2. Отсортируем рекурсивно левую и правую части и решим задачу для них.
- CI3. Получим упорядоченный массив из двух упорядоченных половин, попутно получив ответ в нашей задаче.

# Подсчёт количества инверсий

Постановка задачи:

- Дана перестановка  $a$  размера  $n$ .
- Нужно найти количество пар индексов  $(i, j)$  таких, что  $i < j$  и  $a_i > a_j$ .

Идея — алгоритм CI (CountInversions):

- CI1. Поделим массив на две равные по длине части.
- CI2. Отсортируем рекурсивно левую и правую части и решим задачу для них.
- CI3. Получим упорядоченный массив из двух упорядоченных половин, попутно получив ответ в нашей задаче.

# Подсчёт количества инверсий

Идея — алгоритм CI (CountInversions):

- CI1. Поделим массив на две равные по длине части.
- CI2. Отсортируем рекурсивно левую и правую части и решим задачу для них.
- CI3. Получим упорядоченный массив из двух упорядоченных половин, попутно получив ответ в нашей задаче.

Как выполнить пункт CI3 — алгоритм MC (Merge And Count):

- MC1. Рассмотрим первые числа левой и правой частей.
- MC2. Выберем из них минимальное, допишем его к ответу и удалим из соответствующей части.
- MC3. Если меньшее число из второй части, добавим к ответу количество чисел, оставшихся в первой части: все они больше текущего числа, но в исходном массиве стоят левее.
- MC4. Если хоть одна из частей непуста, перейдём к шагу MC1.

Алгоритм MC работает за линейное время от длины массива.

# Подсчёт количества инверсий

Идея — алгоритм CI (CountInversions):

- CI1. Поделим массив на две равные по длине части.
- CI2. Отсортируем рекурсивно левую и правую части и решим задачу для них.
- CI3. Получим упорядоченный массив из двух упорядоченных половин, попутно получив ответ в нашей задаче.

Как выполнить пункт CI3 — алгоритм MC (Merge And Count):

- MC1. Рассмотрим первые числа левой и правой частей.
- MC2. Выберем из них минимальное, допишем его к ответу и удалим из соответствующей части.
- MC3. Если меньшее число из второй части, добавим к ответу количество чисел, оставшихся в первой части: все они больше текущего числа, но в исходном массиве стоят левее.
- MC4. Если хоть одна из частей непуста, перейдём к шагу MC1.

Алгоритм MC работает за линейное время от длины массива.

# Подсчёт количества инверсий: код

```
count_inversions (a, l, r):  
· if r - l <= 1: return 0  
· m = (l + r) / 2  
· res = count_inversions (a, l, m)  
· res += count_inversions (a, m, r)  
· res += merge_and_count (a, l, m, r)  
· return res
```

```
merge_and_count (a, l, m, r):  
· i = l, j = m, k = l, res = 0  
· while i < m or j < r:  
· · if j >= r or (i < m and a[i] <= a[j]): b[k++] = a[i++]  
· · else: res += m - i, b[k++] = a[j++]  
· a[l..r] = b[l..r]  
· return res
```

# Подсчёт количества инверсий: код

```
count_inversions (a, l, r):
```

```
· if r - l <= 1: return 0
· m = (l + r) / 2
· res = count_inversions (a, l, m)
· res += count_inversions (a, m, r)
· res += merge_and_count (a, l, m, r)
· return res
```

```
merge_and_count (a, l, m, r):
```

```
· i = l, j = m, k = l, res = 0
· while i < m or j < r:
·   · if j >= r or (i < m and a[i] <= a[j]): b[k++] = a[i++]
·   · else: res += m - i, b[k++] = a[j++]
· a[l..r] = b[l..r]
· return res
```

# Подсчёт количества инверсий: альтернативный КОД

```
CI    count_inversions (a, l, r):  
    · if r - l <= 1:  
    · · return 0  
CI1   · m = (l + r) / 2  
CI2   · res = count_inversions (a, l, m)  
CI2   · res += count_inversions (a, m, r)  
    · i = l, j = m, k = l  
MC    · while i < m or j < r:  
MC1   · · if j >= r or (i < m and a[i] < a[j]):  
MC2   · · · b[k++] = a[i++]  
    · · else:  
MC3   · · · res += m - i  
MC2   · · · b[k++] = a[j++]  
    · a[l..r] = b[l..r]  
    · return res
```

# Подсчёт количества инверсий: альтернативный КОД

```
CI    count_inversions (a, l, r):  
    · if r - l <= 1:  
    · · return 0  
CI1   · m = (l + r) / 2  
CI2   · res = count_inversions (a, l, m)  
CI2   · res += count_inversions (a, m, r)  
    · i = l, j = m, k = l  
MC    · while i < m or j < r:  
MC1   · · if j >= r or (i < m and a[i] < a[j]):  
MC2   · · · b[k++] = a[i++]  
    · · else:  
MC3   · · · res += m - i  
MC2   · · · b[k++] = a[j++]  
    · a[l..r] = b[l..r]  
    · return res
```

# Подсчёт количества инверсий: анализ

## Общие наблюдения:

- Алгоритм CI отличается от алгоритма M только тем, что дополнительно вычисляет ответ на задачу.
- На каждом уровне вложенности рекурсии алгоритма CI отрезки, на которых он запускается, не имеют общих точек.
- Алгоритм MC работает за линейное время от длины отрезка.
- Общее время работы —  $O(n \cdot d)$ , где  $d$  — максимальный уровень вложенности рекурсии.

## Специфика алгоритма:

- Каждый отрезок делится алгоритмом MC поровну.
- Значит,  $d = \log_2 n$  и общее время работы —  $O(n \log n)$ .

## Используемая память:

- Заметим, что для массива  $b$  требуется  $O(n)$  дополнительной памяти.

# Подсчёт количества инверсий: анализ

Общие наблюдения:

- Алгоритм CI отличается от алгоритма M только тем, что дополнительно вычисляет ответ на задачу.
- На каждом уровне вложенности рекурсии алгоритма CI отрезки, на которых он запускается, не имеют общих точек.
- Алгоритм MC работает за линейное время от длины отрезка.
- Общее время работы —  $O(n \cdot d)$ , где  $d$  — максимальный уровень вложенности рекурсии.

Специфика алгоритма:

- Каждый отрезок делится алгоритмом MC поровну.
- Значит,  $d = \log_2 n$  и общее время работы —  $O(n \log n)$ .

Используемая память:

- Заметим, что для массива  $b$  требуется  $O(n)$  дополнительной памяти.

# Подсчёт количества инверсий: анализ

Общие наблюдения:

- Алгоритм CI отличается от алгоритма M только тем, что дополнительно вычисляет ответ на задачу.
- На каждом уровне вложенности рекурсии алгоритма CI отрезки, на которых он запускается, не имеют общих точек.
- Алгоритм MC работает за линейное время от длины отрезка.
- Общее время работы —  $O(n \cdot d)$ , где  $d$  — максимальный уровень вложенности рекурсии.

Специфика алгоритма:

- Каждый отрезок делится алгоритмом MC поровну.
- Значит,  $d = \log_2 n$  и общее время работы —  $O(n \log n)$ .

Используемая память:

- Заметим, что для массива  $b$  требуется  $O(n)$  дополнительной памяти.

# Двоичная куча

Постановка задачи — реализовать структуру данных, поддерживающую следующие операции:

- Добавление элемента за время  $O(\log n)$ .
- Поиск максимума за время  $O(1)$ .
- Удаление максимума за время  $O(\log n)$ .

Двоичная куча — это массив  $a[0..n)$ , в котором выполнены соотношения  $a[k] \geq a[2k + 1]$  и  $a[k] \geq a[2k + 2]$  (свойство кучи) для всех  $k$ , для которых существуют соответствующие пары.

# Двоичная куча

Постановка задачи — реализовать структуру данных, поддерживающую следующие операции:

- Добавление элемента за время  $O(\log n)$ .
- Поиск максимума за время  $O(1)$ .
- Удаление максимума за время  $O(\log n)$ .

Двоичная куча — это массив  $a[0..n)$ , в котором выполнены соотношения  $a[k] \geq a[2k + 1]$  и  $a[k] \geq a[2k + 2]$  (свойство кучи) для всех  $k$ , для которых существуют соответствующие пары.

# Двоичная куча: алгоритмы

- При добавлении элемента увеличиваем размер кучи на единицу и дописываем новый элемент в конец. После этого свойство кучи выполнено для всех элементов, кроме, может быть, последнего.
- При поиске максимума просто возвращаем элемент массива  $a[0]$ .
- При удалении максимума уменьшаем размер кучи на единицу и на место  $a[0]$  записываем  $a[n - 1]$ , где  $n$  — размер кучи до удаления. После этого свойство кучи выполнено для всех элементов, кроме, может быть, первого.

## Двоичная куча: подзадача

Подзадача, возникающая при добавлении и удалении элемента:

- Даны массив  $a[0..n)$  и номер элемента в нём  $k$ .
- Известно, что для всех элементов, кроме  $a[k]$ , свойство кучи выполнено.
- Требуется сделать так, чтобы оно было выполнено и для  $a[k]$ .

## Двоичная куча: подзадача

Подзадача, возникающая при добавлении и удалении элемента:

- Даны массив  $a[0..n)$  и номер элемента в нём  $k$ .
- Известно, что для всех элементов, кроме  $a[k]$ , свойство кучи выполнено.
- Требуется сделать так, чтобы оно было выполнено и для  $a[k]$ .

Решение при  $k = 0$  — алгоритм SD (SiftDown):

**SD1.** Если  $2k + 1 \geq n$ , свойство кучи выполнено автоматически.

**SD2.** В противном случае выберем из  $a[2k + 1]$  и  $a[2k + 2]$  наибольший элемент  $x$ .

**SD3.** Если  $a[k] \geq x$ , свойство кучи выполнено.

**SD4.** Иначе поменяем местами  $a[k]$  и  $x$ , после чего запустим алгоритм SD для номера элемента  $x$ .

## Двоичная куча: подзадача

Подзадача, возникающая при добавлении и удалении элемента:

- Даны массив  $a[0..n)$  и номер элемента в нём  $k$ .
- Известно, что для всех элементов, кроме  $a[k]$ , свойство кучи выполнено.
- Требуется сделать так, чтобы оно было выполнено и для  $a[k]$ .

Решение при  $k = n - 1$  — алгоритм SU (SiftUp):

- SU1. Если  $k = 0$ , свойство кучи выполнено автоматически.
- SU2. Если  $a[(k - 1)/2] \geq a[k]$ , свойство кучи также выполнено.
- SU3. Иначе поменяем местами  $a[(k - 1)/2]$  и  $a[k]$ , после чего запустим алгоритм SU для  $(k - 1)/2$ .

# Двоичная куча: код интерфейса

```
insert (a, n, x):  
· a[n] = x  
· n += 1  
· sift_up (a, n - 1, n)
```

```
get_max (a, n):  
· return a[0]
```

```
remove_max (a, n):  
· n -= 1  
· a[0] = a[n]  
· sift_down (a, 0, n)
```

# Двоичная куча: код интерфейса

```
insert (a, n, x):
```

- `a[n] = x`
- `n += 1`
- `sift_up (a, n - 1, n)`

```
get_max (a, n):
```

- `return a[0]`

```
remove_max (a, n):
```

- `n -= 1`
- `a[0] = a[n]`
- `sift_down (a, 0, n)`

# Двоичная куча: код интерфейса

```
insert (a, n, x):
```

- `a[n] = x`
- `n += 1`
- `sift_up (a, n - 1, n)`

```
get_max (a, n):
```

- `return a[0]`

```
remove_max (a, n):
```

- `n -= 1`
- `a[0] = a[n]`
- `sift_down (a, 0, n)`

# Двоичная куча: код реализации

```
sift_up (a, i, n):  
· y = a[i]  
· while i > 0:  
· · j = (i - 1) / 2  
· · if a[j] >= y:  
· · · break  
· · a[i] = a[j]  
· · i = j  
· a[i] = y
```

```
sift_down (a, i, n):  
· y = a[i]  
· while True:  
· · j = i * 2 + 1  
· · if j >= n:  
· · · break  
· · if j + 1 < n and a[j] < a[j +  
1]:  
· · · j += 1  
· · if y >= a[j]:  
· · · break  
· · a[i] = a[j]  
· · i = j  
· a[i] = y
```

# Двоичная куча: код реализации

```
sift_up (a, i, n):
```

```
· y = a[i]
· while i > 0:
· · j = (i - 1) / 2
· · if a[j] >= y:
· · · break
· · a[i] = a[j]
· · i = j
· a[i] = y
```

```
sift_down (a, i, n):
```

```
· y = a[i]
· while True:
· · j = i * 2 + 1
· · if j >= n:
· · · break
· · if j + 1 < n and a[j] < a[j + 1]:
· · · j += 1
· · if y >= a[j]:
· · · break
· · a[i] = a[j]
· · i = j
· a[i] = y
```

# Двоичная куча: анализ

Общие наблюдения:

- Алгоритм SU работает за  $O(\log n)$ : на каждом шаге номер элемента уменьшается хотя бы вдвое.
- Алгоритм SD работает за  $O(\log n)$ : на каждом шаге номер элемента увеличивается хотя бы вдвое.

# Содержание

- 1 Алгоритмы сортировки
  - Постановка задачи
  - Быстрая сортировка
  - Сортировка слиянием
  - Сортировка с помощью кучи
- 2 Алгоритмы в смежных задачах
  - Мотивация
  - Порядковая статистика
  - Подсчёт количества инверсий
  - Двоичная куча
- 3 Примеры
  - Сортировка длинных чисел
  - Сортировка точек
  - Интерактивная сортировка

# Сортировка длинных чисел

|    |                                                                          |        |
|----|--------------------------------------------------------------------------|--------|
| 1  | <code>#include &lt;algorithm&gt;</code>                                  | ВВОД:  |
| 2  | <code>#include &lt;iostream&gt;</code>                                   | 10     |
| 3  | <code>#include &lt;string&gt;</code>                                     | 20     |
| 4  | <code>#include &lt;vector&gt;</code>                                     | 999    |
| 5  | <code>using namespace std;</code>                                        | 9      |
| 6  |                                                                          | 23     |
| 7  | <code>bool lessNum (string const &amp; a, string const &amp; b) {</code> | 3      |
| 8  | <code>    if (a.size () != b.size ())</code>                             | 333    |
| 9  | <code>        return a.size () &lt; b.size ();</code>                    |        |
| 10 | <code>    return a &lt; b;</code>                                        |        |
| 11 | <code>}</code>                                                           |        |
| 12 |                                                                          | ВЫВОД: |
| 13 | <code>int main () {</code>                                               | 3      |
| 14 | <code>    vector &lt;string&gt; s;</code>                                | 9      |
| 15 | <code>    string t;</code>                                               | 10     |
| 16 | <code>    while (cin &gt;&gt; t)</code>                                  | 20     |
| 17 | <code>        s.push_back (t);</code>                                    | 23     |
| 18 | <code>    sort (s.begin (), s.end (), lessNum);</code>                   | 333    |
| 19 | <code>    for (auto e : s)</code>                                        | 999    |
| 20 | <code>        cout &lt;&lt; e &lt;&lt; endl;</code>                      |        |
| 21 | <code>    return 0;</code>                                               |        |
| 22 | <code>}</code>                                                           |        |

# Сортировка точек

```

1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  struct Point {int x, y;};
7  bool operator < (Point const & a, Point const & b) {
8      if (a.y != b.y) return a.y > b.y;
9      return a.x < b.x;
10 }
11
12 int main () {
13     int n;  cin >> n;
14     vector <Point> a (n);
15     for (auto & [x, y] : a)  // C++17
16         cin >> x >> y;
17     sort (a.begin (), a.end ());
18     for (auto [x, y] : a)  // C++17
19         cout << x << " " << y << endl;
20     return 0;
21 }

```

ВВОД:

```

9
0 1
0 -1
1 0
-1 0
0 0
1 -1
-1 1
-1 -1
1 1

```

ВЫВОД:

```

-1 1
0 1
1 1
-1 0
0 0
1 0
-1 -1
0 -1
1 -1

```

# Интерактивная сортировка

```

1  #include <algorithm>
2  #include <iostream>
3  #include <numeric>
4  #include <vector>
5  using namespace std;
6
7  bool ask (int a, int b) {
8      cout << "? " << a << " " << b << endl;
9      string s;  cin >> s;
10     return s == "<";
11 }
12
13 int main () {
14     int n;  cin >> n;
15     vector <int> p (n);
16     iota (p.begin (), p.end (), 1);
17     sort (p.begin (), p.end (), ask);
18     cout << "!";
19     for (auto i : p)
20         cout << " " << i;
21     cout << endl;
22     return 0;
23 }

```

ВВОД | ВЫВОД:

|   |  |           |
|---|--|-----------|
| 4 |  |           |
| < |  | ? 2 1     |
| > |  | ? 3 2     |
| < |  | ? 3 1     |
| > |  | ? 3 2     |
| > |  | ? 4 2     |
| > |  | ? 4 1     |
|   |  | ! 2 3 1 4 |

# Вопросы?

## Вопросы?