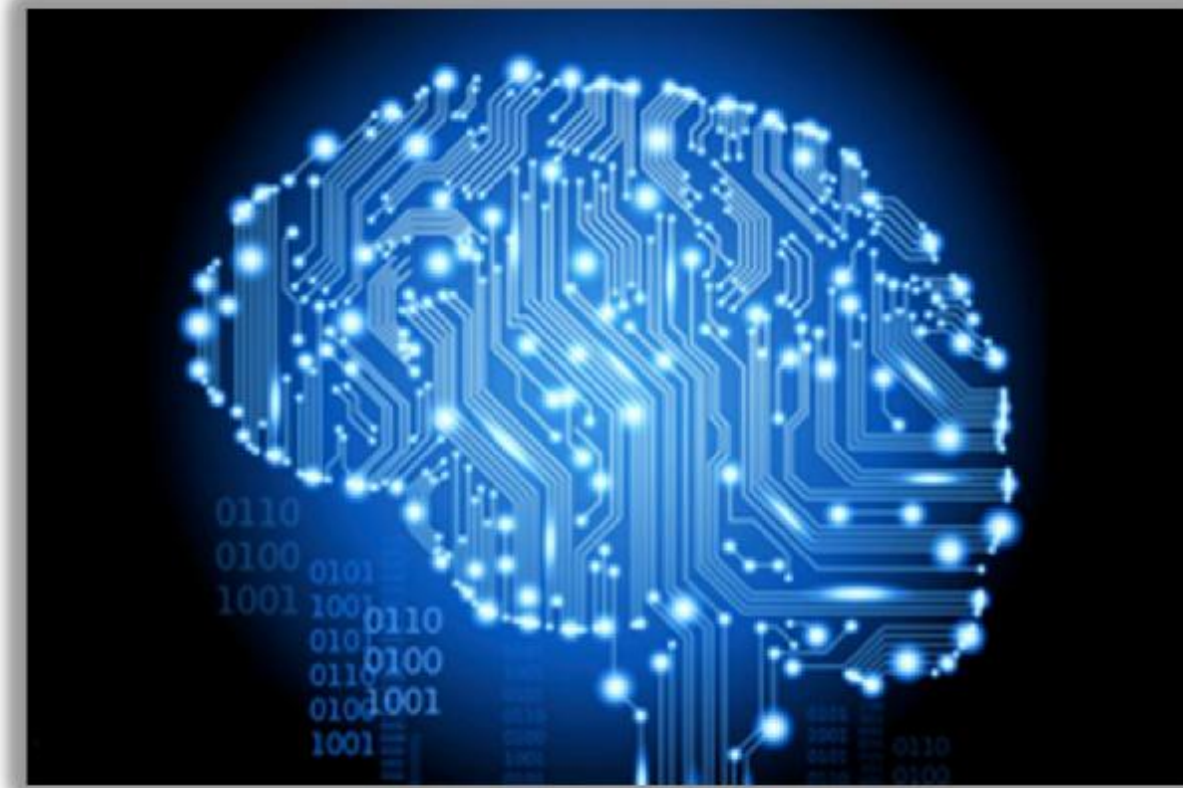
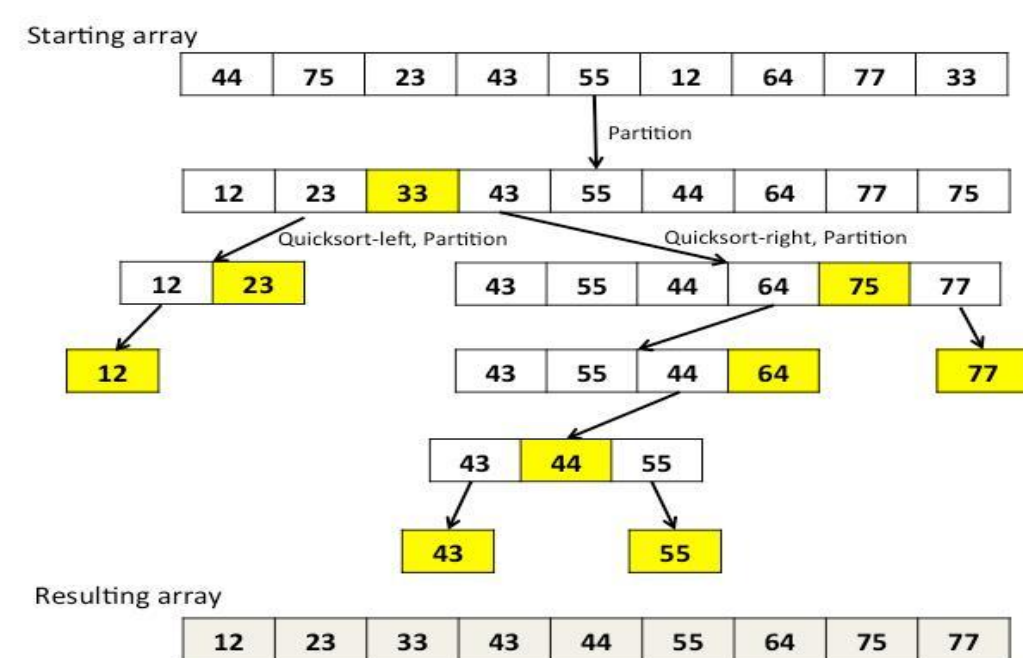


PISL05. Quick Sort, Heap Sort , Count Sort etc.



PISL05. Quick Sort, Heap Sort , Count Sort etc.

- ⌘ Quick Sort.
 - § Принцип сортировки
 - § Доказательство эффективности
 - § O-оценки
 - § Оптимизация Quick Sort
- ⌘ Selection sort
 - § Принцип сортировки
 - § O-оценки
- ⌘ Heap sort
 - § Принцип сортировки
 - § O-оценки
- ⌘ Сортировки, быстрее чем $O(n \log n)$
- ⌘ Задачи (A) (B) (C)

- ⌘ **Материалы:** <http://tinyurl.com/ei-pisl>
- ⌘ **Github:** <https://github.com/Khmelov/PISL2017-01-26>

PISL04. Quick sort

[illegible]

PISL05. Quick Sort, Heap Sort , Count Sort etc.

Функция $\text{QUICKSORT}(A, \ell, r)$

если $\ell \geq r$:
вернуть

```

 $m \leftarrow \text{PARTITION}(A, \ell, r)$ 
QUICKSORT( $A, \ell, m - 1$ )
QUICKSORT( $A, m + 1, r$ )

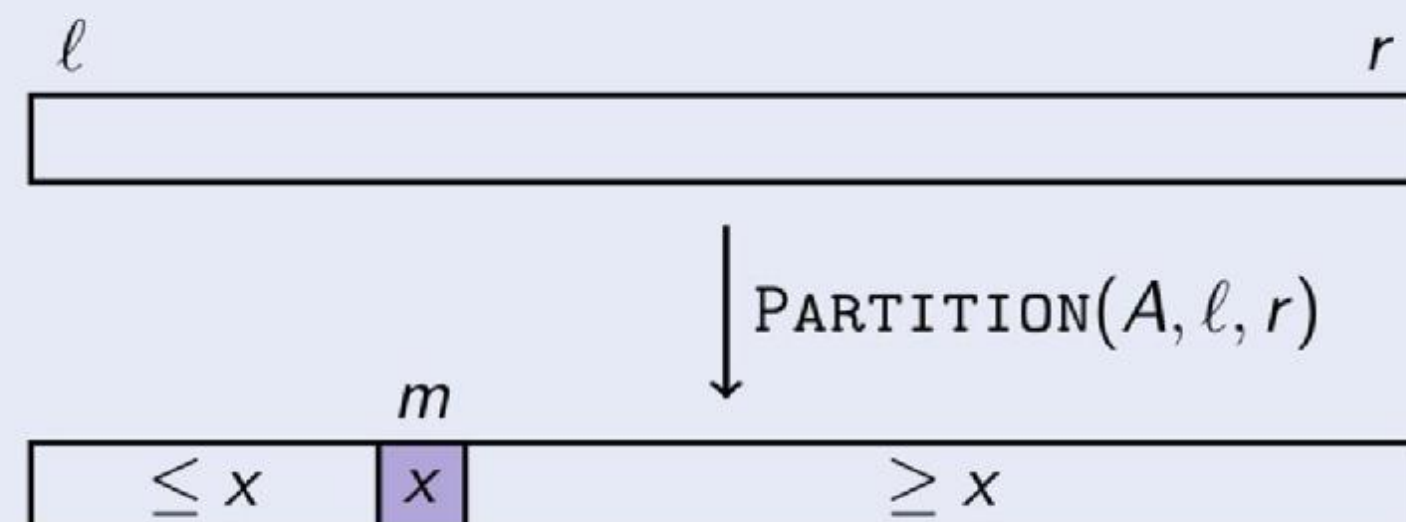
```


Функция QUICKSORT(A, ℓ, r)если $\ell \geq r$:

вернуть

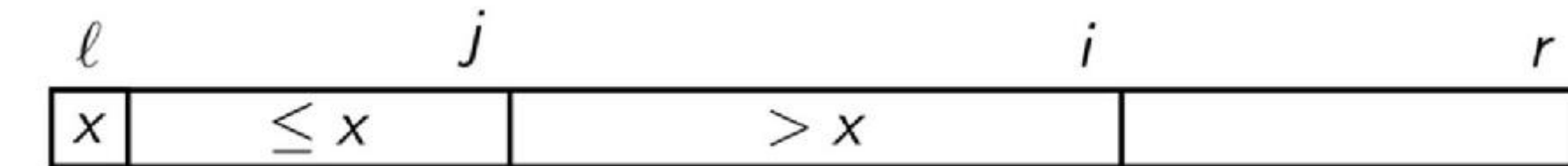
 $m \leftarrow \text{PARTITION}(A, \ell, r)$ QUICKSORT($A, \ell, m - 1$)QUICKSORT($A, m + 1, r$)

PARTITION визуально



5

Разбиение: основные идеи



- $x = A[\ell]$ — опорный элемент
- двигаем i от $\ell + 1$ до r , поддерживая следующий инвариант:
 - $A[k] \leq x$ для всех $\ell + 1 \leq k \leq j$
 - $A[k] > x$ для всех $j + 1 \leq k \leq i$
- пусть i только что увеличился; если $A[i] > x$, не делаем ничего; в противном случае меняем $A[i]$ с $A[j + 1]$ и увеличиваем j

6

Функция PARTITION(A, ℓ, r) $x \leftarrow A[\ell]$ $j \leftarrow \ell$ для i от $\ell + 1$ до r :если $A[i] \leq x$: $j \leftarrow j + 1$ обменять $A[j]$ и $A[i]$ обменять $A[\ell]$ и $A[j]$ вернуть j

7

Функция PARTITION(A, ℓ, r) $x \leftarrow A[\ell]$ $j \leftarrow \ell$ для i от $\ell + 1$ до r :если $A[i] \leq x$: $j \leftarrow j + 1$ обменять $A[j]$ и $A[i]$ обменять $A[\ell]$ и $A[j]$ вернуть j

8

Функция $\text{PARTITION}(A, \ell, r)$

```

 $x \leftarrow A[\ell]$ 
 $j \leftarrow \ell$ 
для  $i$  от  $\ell + 1$  до  $r$ :
    если  $A[i] \leq x$ :
         $j \leftarrow j + 1$ 
        обменять  $A[j]$  и  $A[i]$ 
обменять  $A[\ell]$  и  $A[j]$ 
вернуть  $j$ 

```

Время работы: $O(n)$.

Плохие и хорошие разделители

- $T(n) = T(n-1) + n$:

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

Плохие и хорошие разделители

- $T(n) = T(n-1) + n$:

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

- $T(n) = T(n-5) + T(4) + n$:

$$T(n) \geq n + (n-5) + (n-10) + \dots = \Theta(n^2)$$

Плохие и хорошие разделители

- $T(n) = T(n-1) + n$:

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

- $T(n) = T(n-5) + T(4) + n$:

$$T(n) \geq n + (n-5) + (n-10) + \dots = \Theta(n^2)$$

- $T(n) = 2T(n/2) + n$: $T(n) = O(n \log n)$

Плохие и хорошие разделители

- $T(n) = T(n-1) + n$:

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

- $T(n) = T(n-5) + T(4) + n$:

$$T(n) \geq n + (n-5) + (n-10) + \dots = \Theta(n^2)$$

- $T(n) = 2T(n/2) + n$: $T(n) = O(n \log n)$

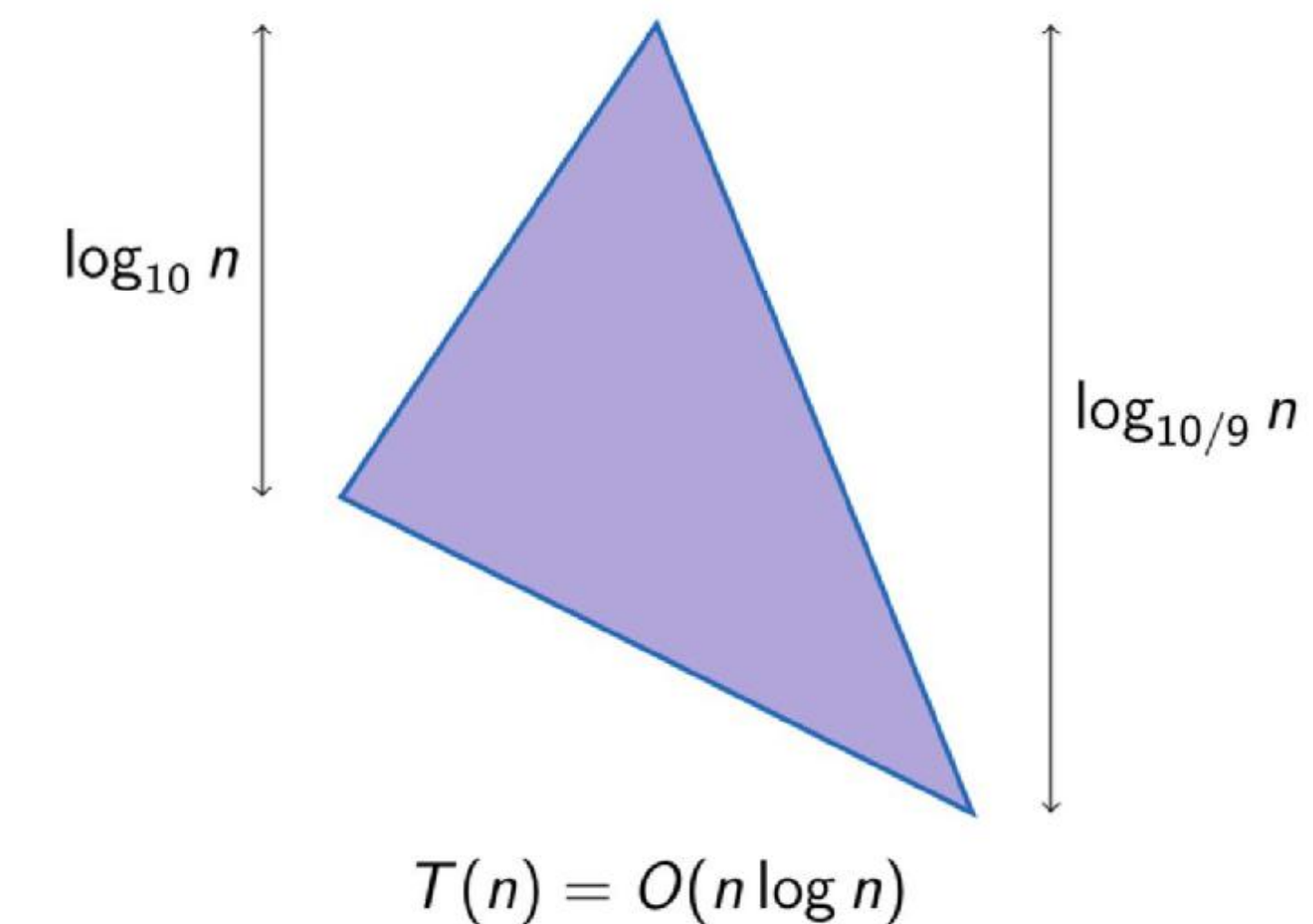
- $T(n) = T(n/10) + T(9n/10) + n$:

$$T(n) = O(n \log n)$$

13

Сбалансированные разбиения

$$T(n) = T(n/10) + T(9n/10) + O(n)$$



14

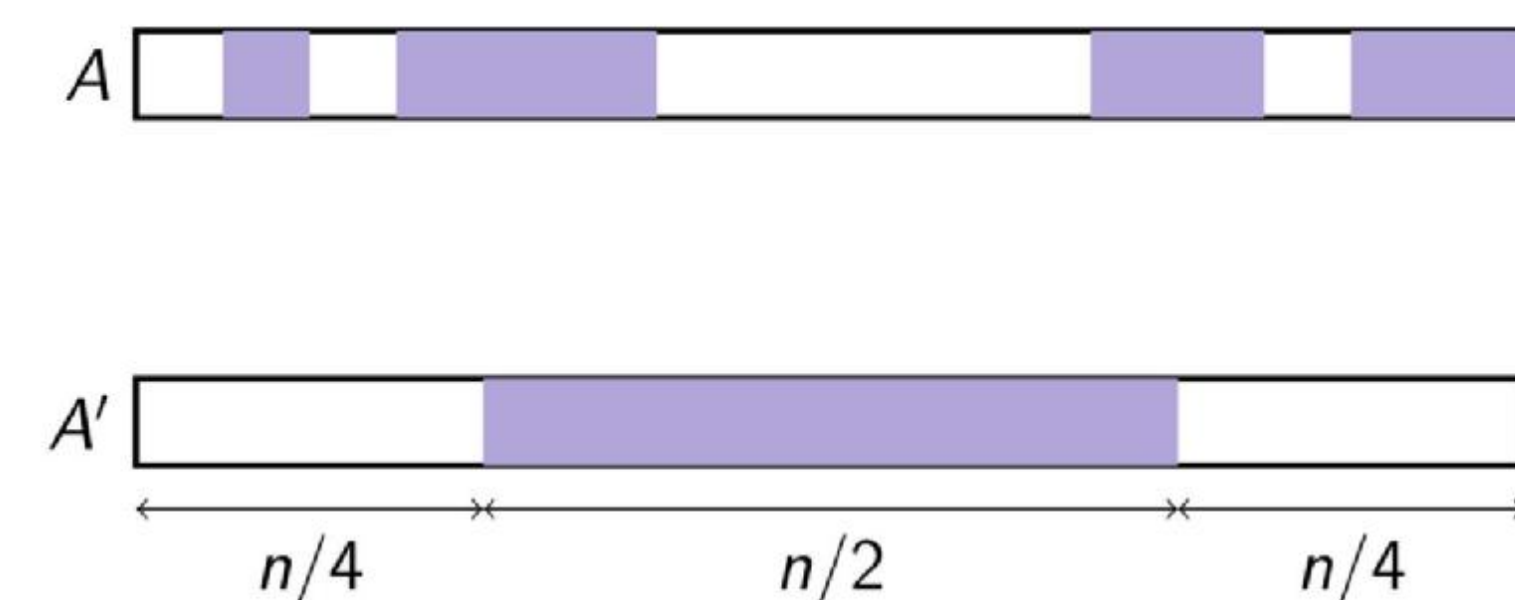
Случайный разделитель

- чтобы разбить A относительно случайного разделителя, обменяем $A[\ell]$ со случайным элементом и вызовем $\text{PARTITION}(A, \ell, r)$

15

Случайный разделитель

- чтобы разбить A относительно случайного разделителя, обменяем $A[\ell]$ со случайным элементом и вызовем $\text{PARTITION}(A, \ell, r)$
- **важное наблюдение:** половина элементов A дают сбалансированное разбиение:



16

Время работы

Теорема

Допустим, что все элементы массива $A[1 \dots n]$ различны и что разделитель всегда выбирается равномерно случайным образом. Тогда среднее время работы алгоритма $\text{QUICKSORT}(A)$ есть $O(n \log n)$, в то время как время работы в худшем случае есть $O(n^2)$.

17

Время работы

Теорема

Допустим, что все элементы массива $A[1 \dots n]$ различны и что разделитель всегда выбирается равномерно случайным образом. Тогда среднее время работы алгоритма $\text{QUICKSORT}(A)$ есть $O(n \log n)$, в то время как время работы в худшем случае есть $O(n^2)$.

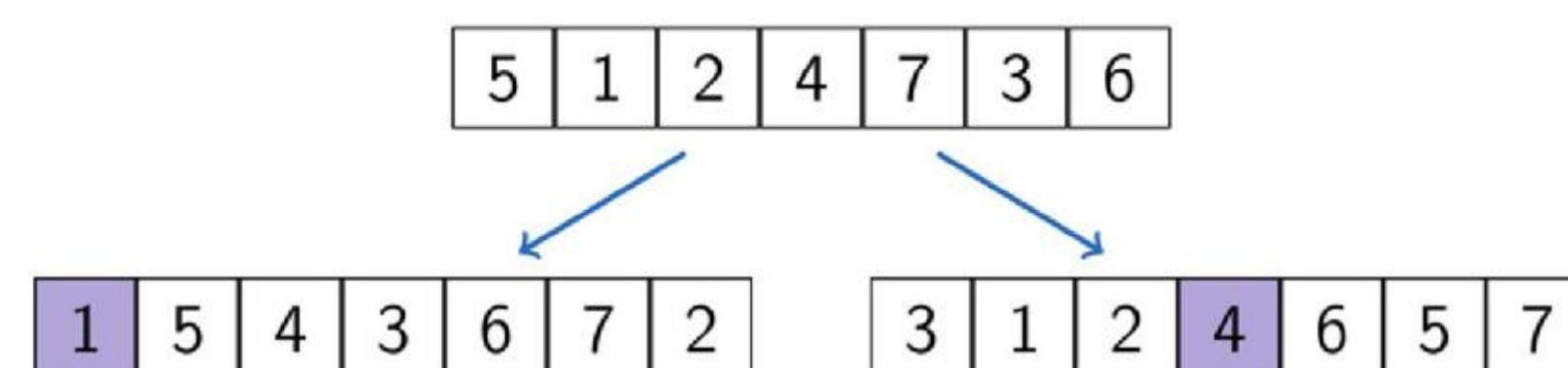
Замечание

Усреднение берётся по случайным числам алгоритма, а не по входам.

18

Идея доказательства: сравнения

- время работы пропорционально количеству сравнений
- сбалансированные разбиения лучше, потому что они лучше уменьшают количество необходимых сравнений:



узнали только, что
1 — минимум

3,1,2 уже никогда не
сравнятся с 6,5,7

19

Идеи доказательства: вероятность

A	5	1	8	9	2	4	7	3	6
A'	1	2	3	4	5	6	7	8	9

$$\text{Prob}(1 \text{ и } 9 \text{ сравнятся}) = \frac{2}{9}$$

$$\text{Prob}(3 \text{ и } 4 \text{ сравнятся}) = 1$$

20

Доказательство

- для $i < j$ положим

$$\chi_{ij} = \begin{cases} 1 & A'[i] \text{ и } A'[j] \text{ сравнились} \\ 0 & \text{в противном случае} \end{cases}$$

- для всех $i < j$ элементы $A'[i]$ и $A'[j]$ либо сравниваются ровно один раз, либо не сравниваются вообще (мы всегда сравниваем с разделителем)
- это, в частности, означает, что время работы в худшем случае не больше $O(n^2)$

21

Доказательство

- важное замечание: $\chi_{ij} = 1$, если и только если первый разделитель, выбранный из $A'[i \dots j]$, — это $A'[i]$ или $A'[j]$
- тогда $\text{Prob}(\chi_{ij}) = \frac{2}{j-i+1}$ и $E(\chi_{ij}) = \frac{2}{j-i+1}$
- тогда среднее время работы есть

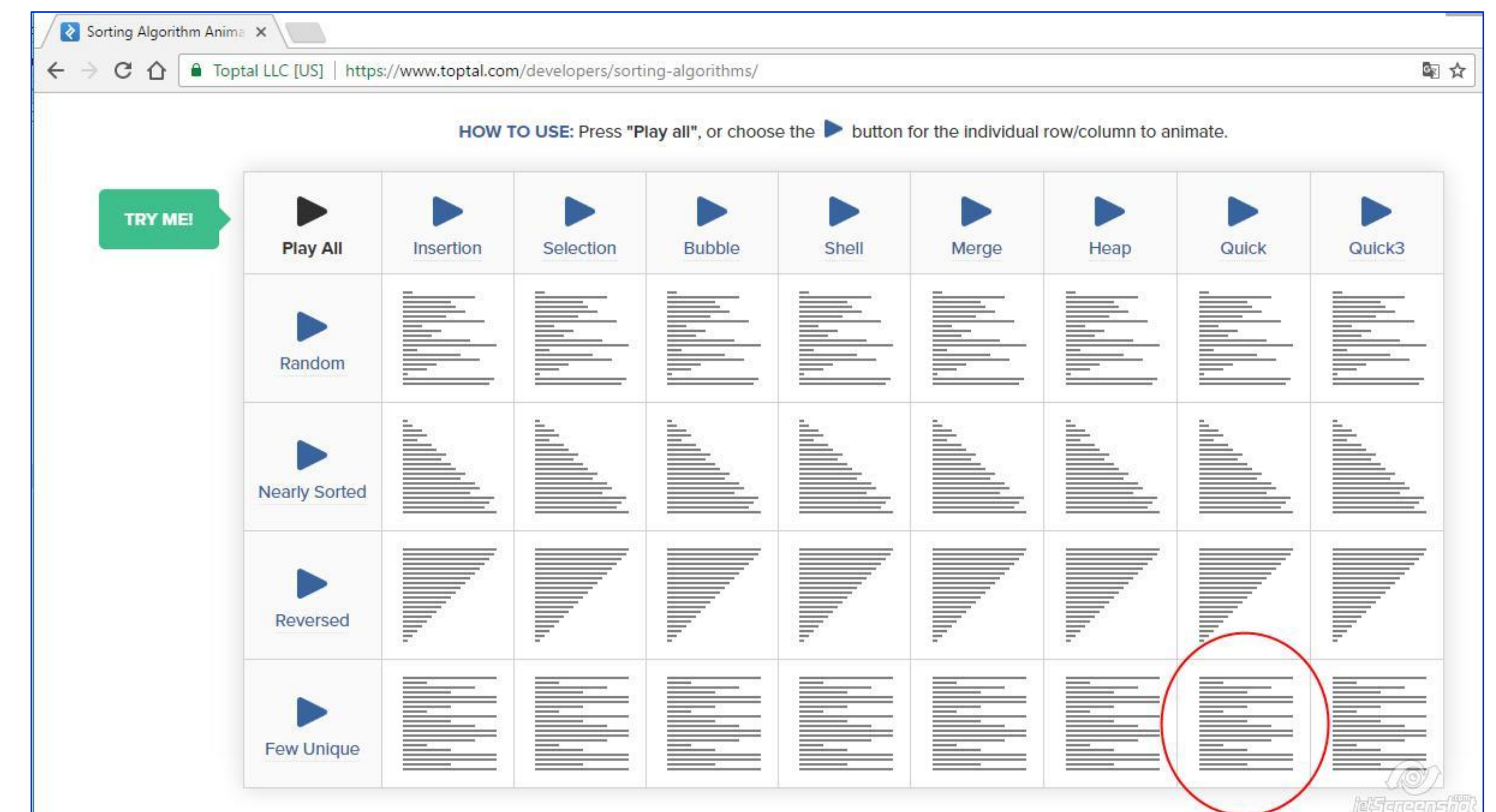
$$\begin{aligned} E \sum_{i=1}^n \sum_{j=i+1}^n \chi_{ij} &= \sum_{i=1}^n \sum_{j=i+1}^n E(\chi_{ij}) \\ &= \sum_{i < j} \frac{2}{j-i+1} \\ &\leq 2n \cdot \left(\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \right) \\ &= \Theta(n \log n) \quad \square \end{aligned}$$

22

Равные элементы

- визуализация:
www.sorting-algorithms.com/quick-sort

23



24

Равные элементы

- визуализация:
www.sorting-algorithms.com/quick-sort
- если все элементы массива равны между собой, то рассмотренная реализация алгоритма быстрой сортировки будет работать квадратичное время

25

Равные элементы

- визуализация:
www.sorting-algorithms.com/quick-sort
- если все элементы массива равны между собой, то рассмотренная реализация алгоритма быстрой сортировки будет работать квадратичное время
- чтобы обойти это препятствие, массив можно разбивать на три части вместо двух: $< x$, $= x$ и $> x$ (3-разбиение)

26

Элиминация хвостовой рекурсии

Процедура $\text{QUICKSORT}(A, \ell, r)$

пока $\ell < r$:

```

 $m \leftarrow \text{PARTITION}(A, \ell, r)$ 
 $\text{QUICKSORT}(A, \ell, m - 1)$ 
 $\ell \leftarrow m + 1$ 

```

Элиминируя рекурсивный вызов для более длинного массива, мы гарантируем, что глубина рекурсии (а значит, и дополнительная память) будет в худшем случае не более $O(\log n)$.

27

Интроспективная сортировка:
 $O(n \log n)$ в худшем случае

- запускает быструю сортировку с простой эвристикой выбора разделителя (например, медиана из первого, среднего и последнего элементов)
- если глубина рекурсии превышает порог $c \log n$, быстрая сортировка прерывается и запускается алгоритм с гарантированным временем $O(n \log n)$ в худшем случае (например, сортировка кучей)

28

Заключение

- Быстрая сортировка работает за время $O(n \log n)$ в среднем случае и за $O(n^2)$ в худшем случае.
- Усреднение берётся по случайным числам алгоритма, но не по входам.
- Очень эффективен на практике.
- Если в массиве могут быть равные числа, стоит использовать 3-разбиение вместо 2-разбиения.
- Элиминация хвостовой рекурсии позволяет сделать так, чтобы алгоритм быстрой сортировки использовал не более $O(\log n)$ дополнительной памяти.

29

Сортировка выбором (*Selection sort*) — алгоритм сортировки. Может быть как устойчивый, так и неустойчивый. На массиве из n элементов имеет время выполнения в худшем, среднем и лучшем случае $\Theta(n^2)$, предполагая что сравнения делаются за постоянное время.

Алгоритм без дополнительного выделения памяти

Шаги алгоритма:

1. находим номер минимального значения в текущем списке
2. производим обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции)
3. теперь сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы

Для реализации устойчивости алгоритма необходимо в пункте 2 минимальный элемент непосредственно вставлять в первую неотсортированную позицию, не меняя порядок остальных элементов.

Сортировка выбором

Действие алгоритма на примере сортировки случайных точек.

Предназначение: Алгоритм сортировки

Структура данных: Массив

Худшее время: $O(n^2)$

Лучшее время: $O(n^2)$

Среднее время: $O(n^2)$

30

Сортировка выбором

Процедура SELECTIONSORT($A[1 \dots n]$)

```

для  $i$  от 1 до  $n$ :
     $k \leftarrow i$ 
    для  $j$  от  $i+1$  до  $n$ :
        если  $A[j] < A[k]$ :
             $k \leftarrow j$ 
    обменять  $A[i]$  и  $A[k]$ 
    
```

31

Сортировка выбором

Процедура SELECTIONSORT($A[1 \dots n]$)

```

для  $i$  от 1 до  $n$ :
     $k \leftarrow i$ 
    для  $j$  от  $i+1$  до  $n$ :
        если  $A[j] < A[k]$ :
             $k \leftarrow j$ 
    обменять  $A[i]$  и  $A[k]$ 
    
```

Время работы: $O(n^2)$.

32

PISL04. Heap sort

PISL04. Heap sort

Википедия
Свободная энциклопедия

Главная страница
Рубрикация
Указатель А—Я
Избранные статьи
Случайная статья
Текущие события

Участие

Сообщить об ошибке
Портал сообщества
Форум
Свежие правки
Новые страницы
Справка
Пожертвовать

Инструменты

Ссылки сюда
Связанные правки
Следственные
Постоянная ссылка
Сведения о странице
Цитировать страницу

Печать/экспорт

Создать книгу
Скачать как PDF
Версия для печати

В других проектах

Викисклад
Викиданные

На других языках

Deutsch
English
Español

Статья Обсуждение

Читая

Текущая версия

Править

Править вики-текст

История

Искать в Википедии

Q

Пирамидальная сортировка

Материал из Википедии — свободной энциклопедии

[\[править\]](#) [\[править вики-текст\]](#)

Текущая версия страницы пока не проверялась опытными участниками и может значительно отличаться от версии, проверенной 15 марта 2016; проверки требуют 13 правок.

Пирамидальная сортировка (англ. *Heapsort*, «Сортировка кучей»^[1]) — алгоритм сортировки, работающий в худшем, в среднем и в лучшем случае (то есть гарантированно) за $O(n \log n)$ операций при сортировке n элементов^[2] Количество применяемой служебной памяти не зависит от размера массива (то есть, $O(1)$).

Может рассматриваться как усовершенствованная сортировка пузырьком, в которой элемент всплывает (min-heap) / тонет (max-heap) по многим путям.

Содержание [\[показать\]](#)

История создания [\[править\]](#) [\[править вики-текст\]](#)

Этот раздел не завершён.

Вы можете помочь проекту, исправив и дополнив его.

Пирамидальная сортировка была предложена Дж. Уильямсом в 1964 году.^[1]

Алгоритм [\[править\]](#) [\[править вики-текст\]](#)

Сортировка пирамидой использует бинарное сортирующее дерево. Сортирующее дерево — это такое дерево, у которого выполнены условия:

- Каждый лист имеет глубину либо d , либо $d - 1$, d — максимальная глубина дерева.
- Значение в любой вершине не меньше (другой вариант — не больше) значения её потомков.

Удобная^[*источник не указан 60 дней*] структура данных для сортирующего дерева — такой массив Array, что Array[0] — элемент в корне, а потомки элемента Array[*i*] являются Array[*2i*+1] и Array[*2i*+2].

Алгоритм сортировки будет состоять из двух основных шагов:

- Выстраиваем элементы массива в виде сортирующего дегереа^[*источник не указан 60 дней*].

Array[*i*] ≥ Array[*2i* + 1]

Array[*i*] ≥ Array[*2i* + 2]

при 0 ≤ *i* < *n*/2.

Этот шаг требует $O(n)$ операций.

Анимированная схема алгоритма ^[*?*]

Пример сортирующего дерева ^[*?*]

16 11 9 10 5 6 8 1 2 4

структура хранения данных сортирующего дерева ^[*?*]

Кафедра экономической информатики Бгуир 2017

33

Кафедра экономической информатики БГУиР 2017

PISL05. Quick Sort, Heap Sort , Count Sort etc.

Кафедра экономической информатики БГУиР 2017

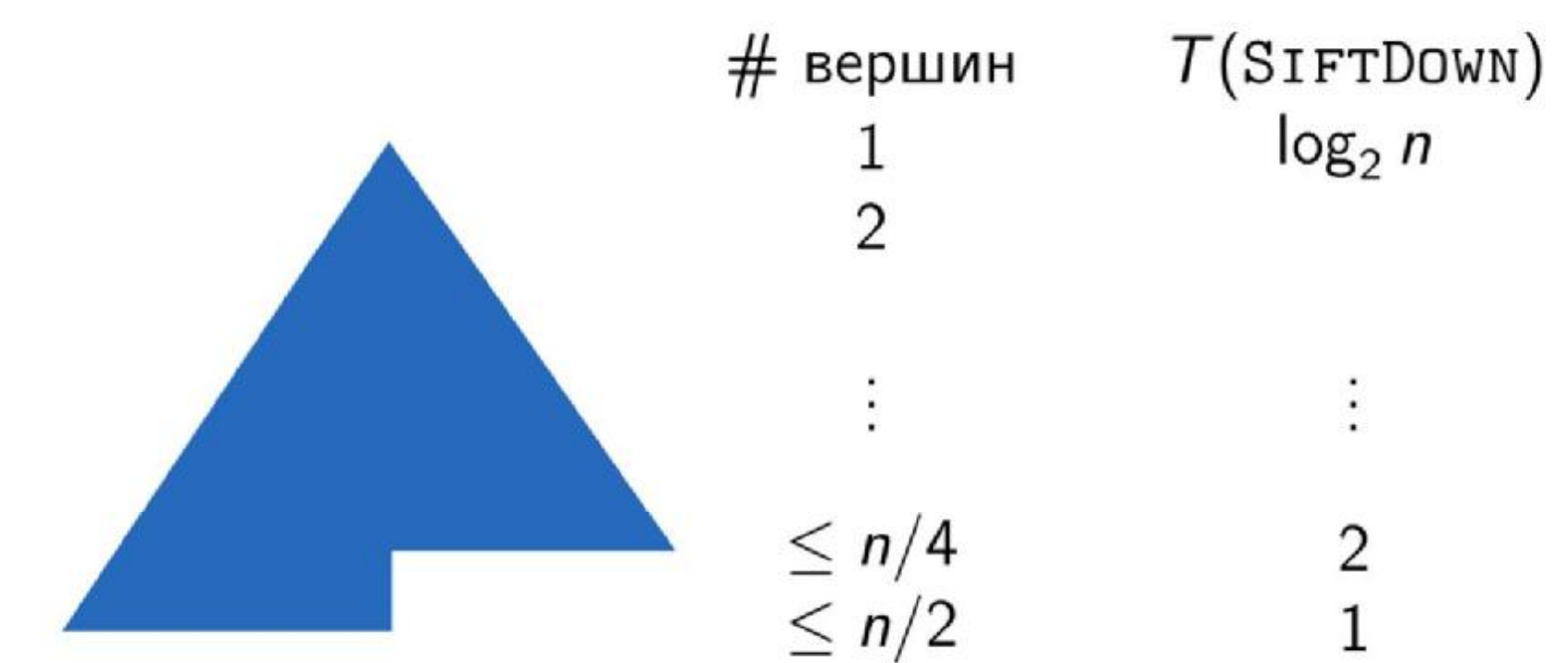
Построение кучи

Процедура BUILDMAXHEAP($A[1 \dots n]$)

для i от $\lfloor n/2 \rfloor$ до 1:
 SIFTDOWN(A, i)

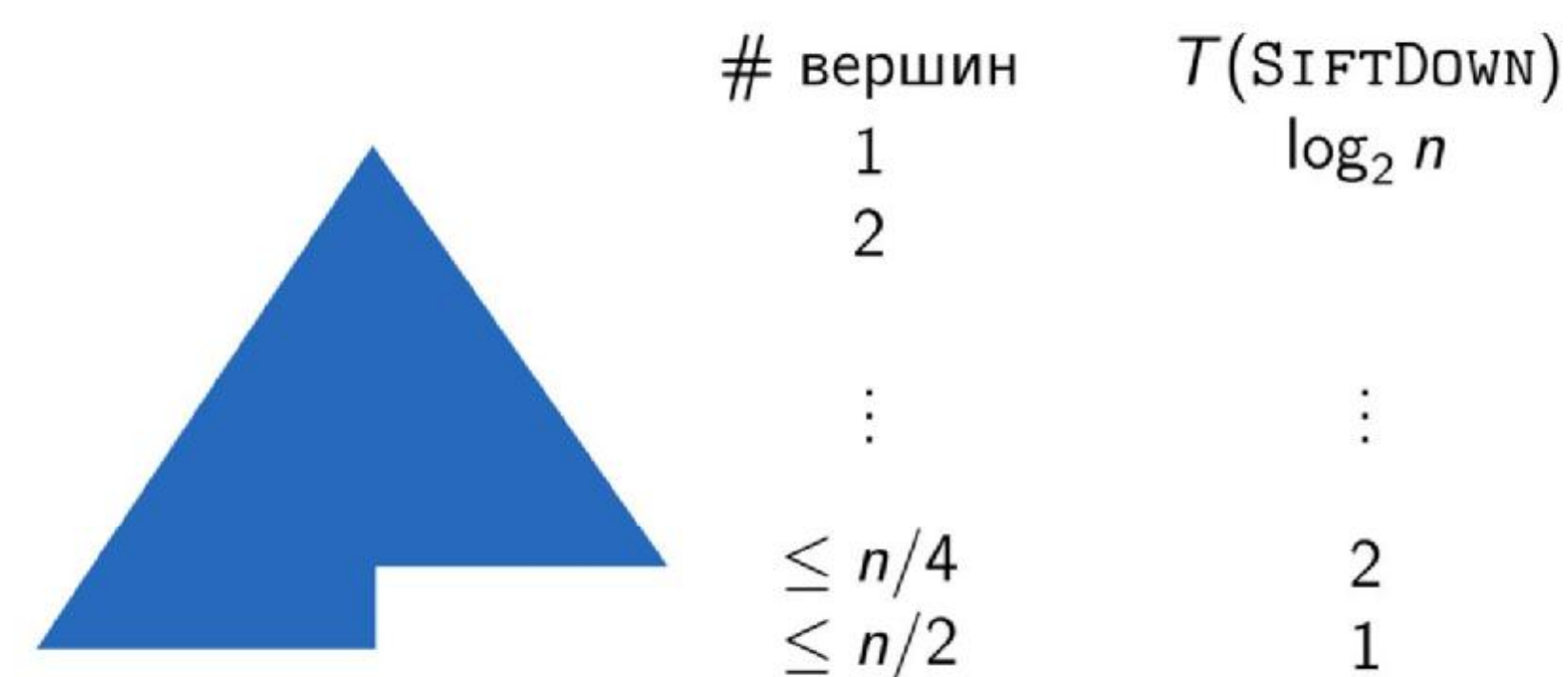
37

Время работы построения кучи



38

Время работы построения кучи

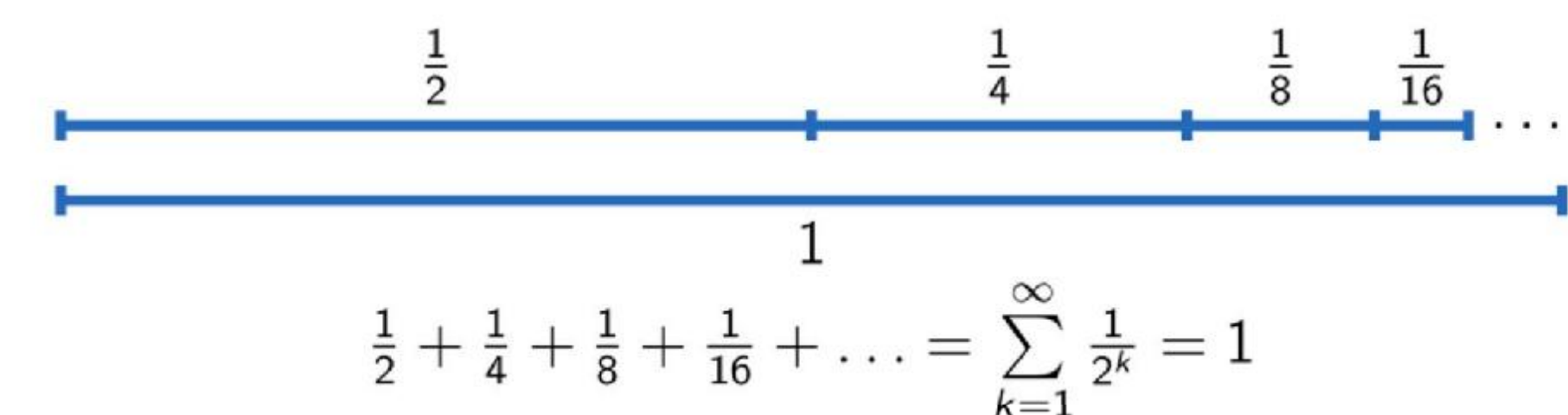


$$T(\text{BUILDHEAP}) \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \dots$$

$$\leq n \cdot \sum_{i=1}^{\infty} \frac{i}{2^i} = 2n$$

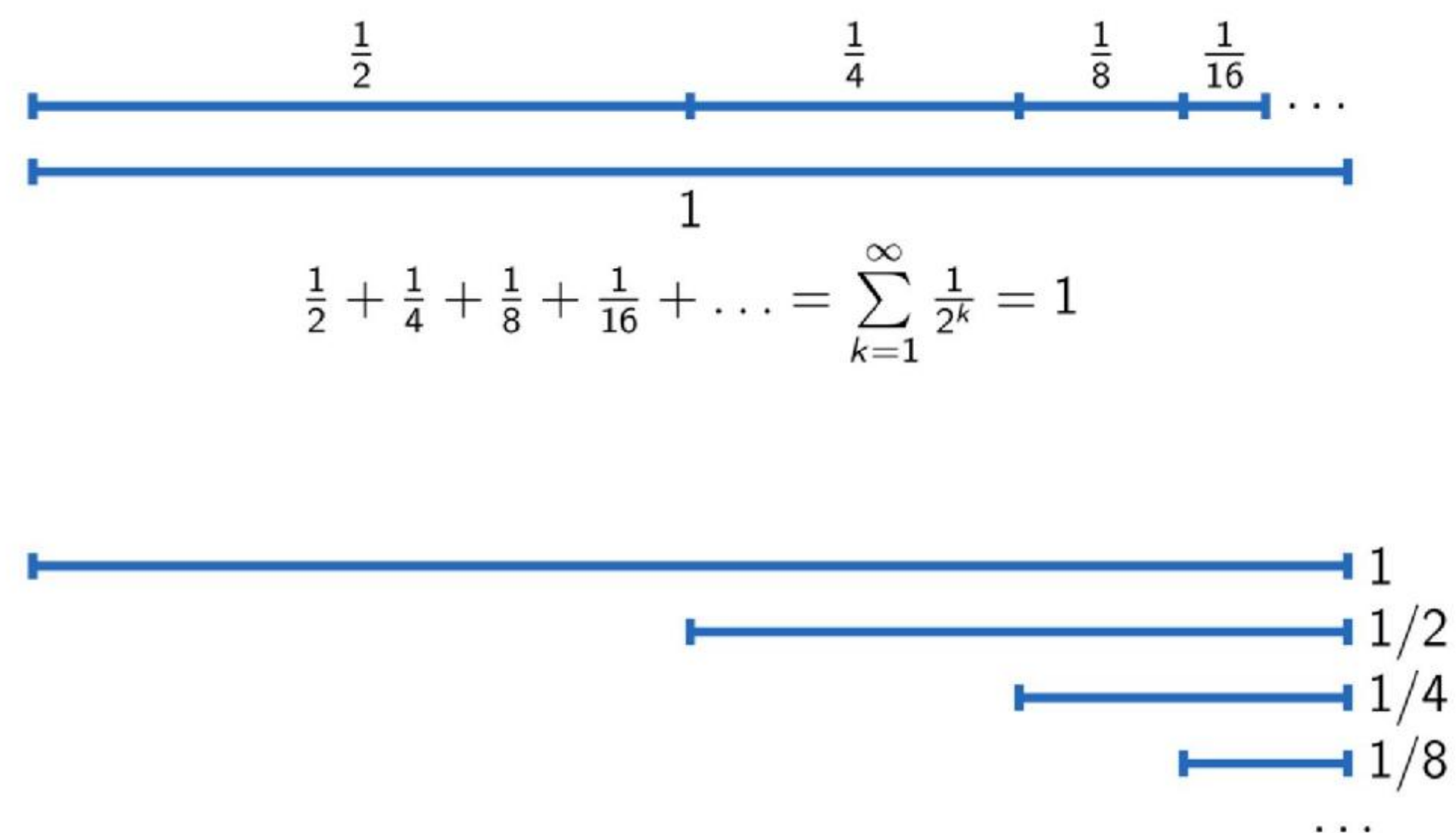
39

Оценка суммы

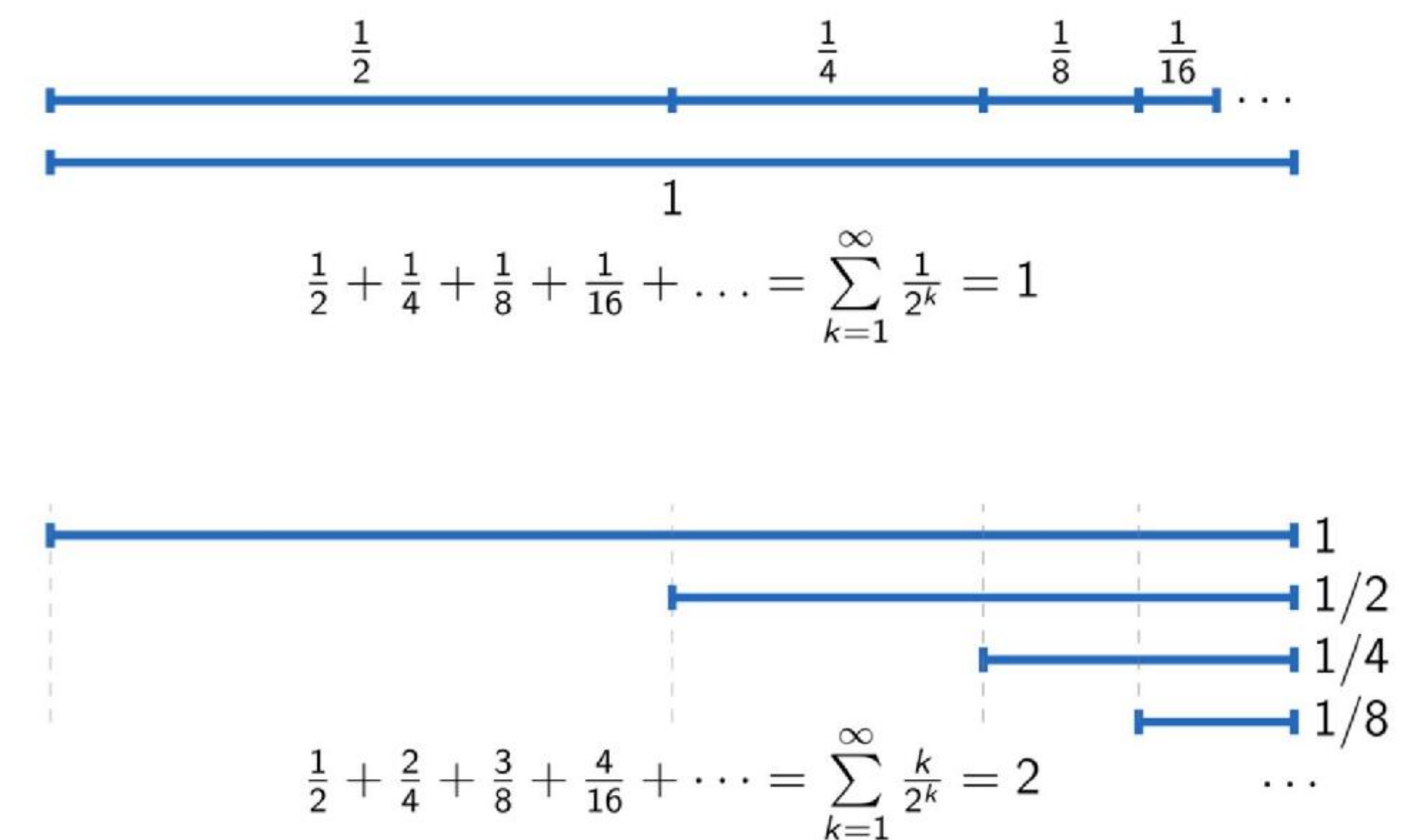


40

Оценка суммы



Оценка суммы



Заклучение


- Время работы сортировки кучей: $O(n \log n)$.

Заклучение

- Время работы сортировки кучей: $O(n \log n)$.
- Время работы построения кучи: $O(n)$.

Заклучение

- Время работы сортировки кучей: $O(n \log n)$.
- Время работы построения кучи: $O(n)$.
- Сортирует на месте.



Википедия

Свободная энциклопедия

Заглавная страница

Рубрикация

Указатель А—Я

Избранные статьи

Случайная статья

Текущие события

Участие

Сообщить об ошибке

Портал сообщества

Форум

Свежие правки

Новые страницы

Справка

Пожертвовать

Инструменты

Ссылки сюда

Связанные правки

Спецстраницы

Постоянная ссылка

Сведения о странице

Цитировать страницу

Печать/экспорт

Создать книгу

Скачать как PDF

Версия для печати

В других проектах

Викиданные

На других языках

Deutsch

English

Español

فارسی

Français

Статья

Обсуждение

Читая

Текущая версия

Править

Править вики-текст

История

Искать в Википедии

Сортировка подсчётом

Материал из Википедии — свободной энциклопедии

Текущая версия страницы пока не проверялась опытными участниками и может значительно отличаться от версии, проверенной 9 декабря 2016; проверки требуют 20 правок.

[править] [править вики-текст]

Сортировка подсчётом

(англ. *Counting sort*) — алгоритм сортировки, в котором используется диапазон чисел сортируемого массива (списка) для подсчёта совпадающих элементов. Применение сортировки подсчётом целесообразно лишь тогда, когда сортируемые числа имеют (или их можно отобразить в) диапазон возможных значений, который достаточно мал по сравнению с сортируемым множеством, например, миллион натуральных чисел меньших 1000.

Предположим, что входной массив состоит из n целых чисел в диапазоне от 0 до $k - 1$, где $k \in \mathbb{N}$. Далее алгоритм будет обобщён для произвольного целочисленного диапазона. Существует несколько модификаций сортировки подсчётом, ниже рассмотрены три линейных и одна квадратичная, которая использует другой подход, но имеет то же название.

Содержание

[показать]

Простой алгоритм

[править] [править вики-текст]

Это простейший вариант алгоритма. Создать вспомогательный массив $C[0..k - 1]$, состоящий из нулей, затем последовательно прочитать элементы входного массива A , для каждого $A[i]$ увеличить $C[A[i]]$ на единицу. Теперь достаточно пройти по массиву C , для каждого $j \in \{0, \dots, k - 1\}$ в массив A последовательно записать число j $C[j]$ раз.

SimpleCountingSort:

for i = 0 to k - 1
 C[i] = 0;
for i = 0 to n - 1
 C[A[i]] = C[A[i]] + 1;
b = 0;
for j = 0 to k - 1
 for i = 0 to C[j] - 1
 A[b] = j;
 b = b + 1;

Алгоритм со списком

[править] [править вики-текст]

Этот вариант (англ. *pigeonhole sorting, count sort*) используется, когда на вход подается массив структур данных, который следует отсортировать по ключам (*key*). Нужно создать вспомогательный массив $C[0..k - 1]$, каждый $C[i]$ в дальнейшем будет содержать список элементов из входного массива. Затем последовательно прочитать элементы входного массива A , каждый $A[i]$ добавить в список $C[A[i].key]$. В заключении пройти по массиву C , для каждого $j \in \{0, \dots, k - 1\}$ в массив A последовательно записывать элементы списка $C[j]$. Алгоритм устойчив.

Кафедра экономической информатики Бгуир 2017

46

PISL05. Quick Sort, Heap Sort , Count Sort etc.

Сортировка подсчётом

Сортировка небольших по величине целых чисел:

2	1	1	1	3	2	2	2	3	2	3	2	2	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Кафедра экономической информатики Бгуир 2017

47

PISL05. Quick Sort, Heap Sort , Count Sort etc.

Сортировка подсчётом

Сортировка небольших по величине целых чисел:

2	1	1	1	3	2	2	2	3	2	3	2	2	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

↓

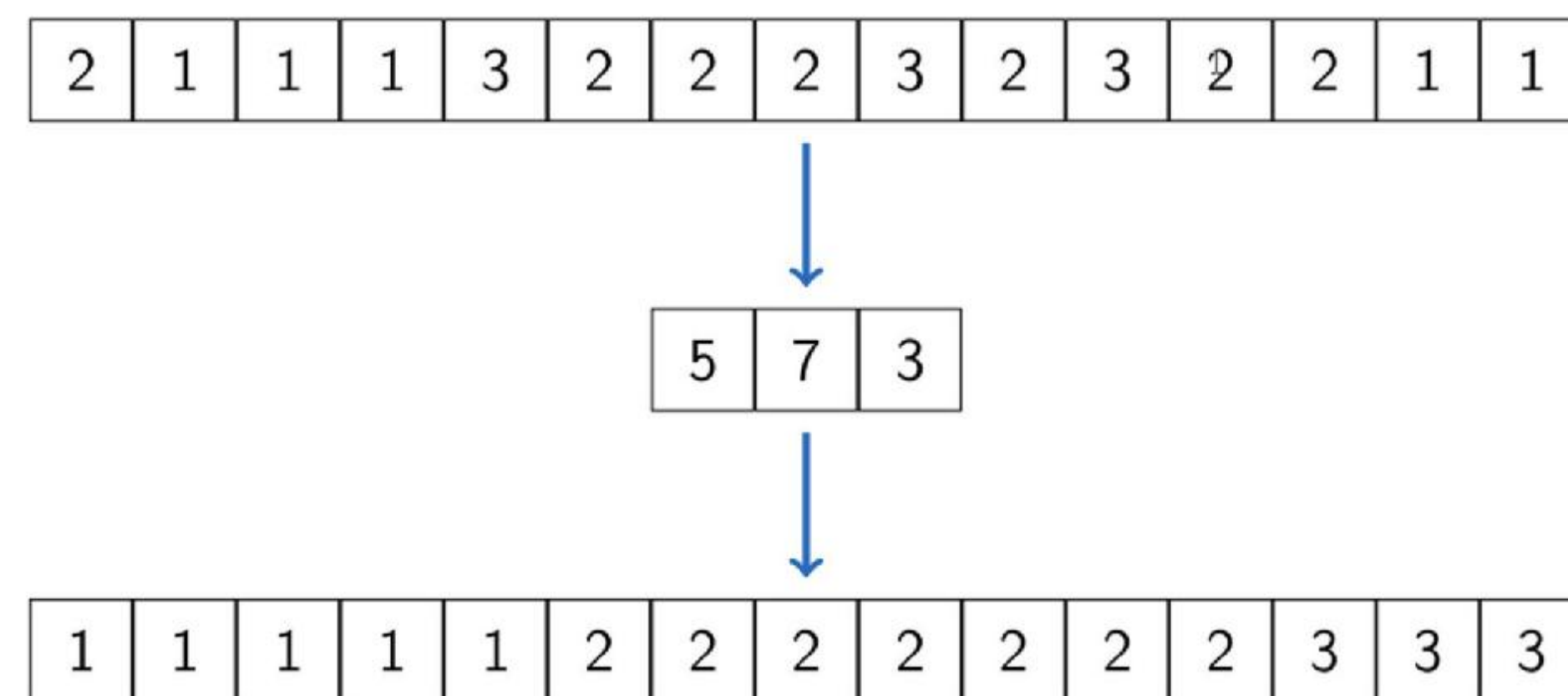
5	7	3
---	---	---

Кафедра экономической информатики Бгуир 2017

48

Сортировка подсчётом

Сортировка небольших по величине целых чисел:



49

Стабильная сортировка подсчётом

Процедура COUNTSORT($A[1 \dots n]$){массив A содержит целые числа от 1 до M }создать массив $B[1 \dots M] \leftarrow [0, 0, \dots, 0]$ для j от 1 до n : $B[A[j]] \leftarrow B[A[j]] + 1$ для i от 2 до M : $B[i] \leftarrow B[i] + B[i - 1]$ для j от n до 1: $A'[B[A[j]]] \leftarrow A[j]$ $B[A[j]] \leftarrow B[A[j]] - 1$ Время работы: $O(n + M)$.

50

```

public static int[] sort(int[] array) {
    int min, max = min = array[0];
    // тупо находим максимальное и минимальное значение
    for (int i = 1; i < array.length; i++) {
        if (array[i] < min) {
            min = array[i];
        }
        if (array[i] > max) {
            max = array[i];
        }
    }
    // понеслась
    return sort(array, min, max);
}

static int[] sort(int[] array, int min, int max) {
    // счетчик это такой массив в котором мы будем считать, как часто встречаются
    // числа в сортируемом массиве.
    // допустим массив равен = {0,2,1,5,1}, min = 0, max = 5
    // счетчик = count[0]...count[5]
    int[] count = new int[max - min + 1];
    // итак считаем...
    for (int i = 0; i < array.length; i++) {
        // подсчитываем сколько раз встречается число,
        // встретилось +1 к счетчику
        count[array[i] - min]++;
    }
    // например. count[0]=1, count[1]=2, count[3]=0, count[4]=0, count[5]=1
    int idx = 0;
    // теперь все готово
    // пробегаем по всему счетчику (от 0 до 5)
    // count[i] - показывает сколько раз встречается то или иное число
    for (int i = 0; i < count.length; i++) {
        // count[0]=1, значит array[0]=0;
        // count[1]=2, значит вставляем два раза array[1]=array[2]=1;
        // count[2]=1, опять только один раз. array[3]=2;
        // count[3]=0, значит ничего не вставляем и т.д.
        for (int j = 0; j < count[i]; j++) {
            array[idx++] = i + min;
        }
    }
    // ну собственно и всё
    return array;
}

public static void main(String[] args) {
    Random rnd = new Random();
    int arr1[] = new int[1024*1024];
    for (int i = 0; i < arr1.length; i++) {
        arr1[i] = rnd.nextInt(1000) + 123;
    }
    int arr2[] = new int[arr1.length];
    System.arraycopy(arr1, 0, arr2, 0, arr1.length);
    long t1 = System.currentTimeMillis();
    sort(arr1);
    long t2 = System.currentTimeMillis();
    Arrays.sort(arr2);
    long t3 = System.currentTimeMillis();
    System.out.printf("counting sort: %d merge sort: %d \r\n", (t2-t1), (t3-t2));
    //
    System.out.println(Arrays.equals(arr1, arr2));
}

```

<http://programador.ru/sorting-positive-int-linear-time/>

51

Цифровая сортировка

Время работы: $O(nd)$, где d — число разрядов во входных числах.

52

Заклучение

- Алгоритм сортировки подсчётом сортирует массив размера n , содержащий целые числа от 1 до M , за время $O(n + M)$.
- Нижняя оценка $\Omega(n \log n)$ не применима к алгоритму сортировки подсчётом, поскольку он основан не на сравнениях.
- Алгоритм сортировки подсчётом можно реализовать так, чтобы он был стабильным: равные элементы будут идти в том же порядке, в каком они были в исходном массиве.
- Алгоритмом цифровой сортировки можно отсортировать числа с d разрядами за время $O(nd)$.

53

Задание А.

lesson03

lesson04

A_BinaryFind

B_MergeSort

C_GetInversions

dataA.txt

dataB.txt

dataC.txt

Lesson4Test

lesson05

A_QSort

B_CountSort

C_QSortOptimized

dataA.txt

dataB.txt

dataC.txt

Lesson4Test

alesnax

apilipenka

astro_emelya

belash_ea

chatovich

du4

ededovich.lesson1

grishkevich

hustlestar

jahstreet

khadasevich

makstra

mrlokans

poznyakbogdan

rudkouski

rudzko

sardika

sazonov_valery

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

Видеорегистраторы и площадь.

На площади установлена одна или несколько камер.

Известны данные о том, когда каждая из них включалась и выключалась (отрезки работы)

Известен список событий на площади (время начала каждого события).

Вам необходимо определить для каждого события сколько камер его записали.

В первой строке задано два целых числа:

число включений камер (отрезки) $1 \leq n \leq 50000$

число событий (точки) $1 \leq m \leq 50000$.

Следующие n строк содержат по два целых числа a_i и b_i ($a_i \leq b_i$) -

координаты концов отрезков (время работы одной какой-то камеры).

Последняя строка содержит m целых чисел - координаты точек.

Все координаты не превышают 10^8 по модулю (!).

Точка считается принадлежащей отрезку, если она находится внутри него или на границе.

Для каждой точки в порядке их появления во вводе выведите,

скольким отрезкам она принадлежит.

Sample Input:

2 3

0 5

7 10

1 6 11

Sample Output:

1 0 0

54

Задание Б.

lesson02

lesson03

lesson04

A_BinaryFind

B_MergeSort

C_GetInversions

dataA.txt

dataB.txt

dataC.txt

Lesson4Test

lesson05

A_QSort

B_CountSort

C_QSortOptimized

dataA.txt

dataB.txt

dataC.txt

Lesson4Test

alesnax

apilipenka

astro_emelya

belash_ea

chatovich

du4

ededovich.lesson1

grishkevich

hustlestar

jahstreet

khadasevich

makstra

mrlokans

poznyakbogdan

rudkouski

rudzko

sardika

sazonov_valery

3

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

import ...

/*

Первая строка содержит число $1 \leq n \leq 10000$, вторая - n натуральных чисел, не превышающих 10.

Выведите упорядоченную по неубыванию последовательность этих чисел.

При сортировке реализуйте метод со сложностью $O(n)$

Пример: <https://karussell.wordpress.com/2010/03/01/fast-integer-sorting-algorithm-on/>

Вольный перевод: <http://programador.ru/sorting-positive-int-linear-time/>

*/

public class B_CountSort {

int[] countSort(InputStream stream) throws FileNotFoundException {

//подготовка к чтению данных

Scanner scanner = new Scanner(stream);

//!!!!!!!!!!!!!!!!!!!! НАЧАЛО ЗАДАЧИ !!!!!!!!!!!!!!!!!!!!!

//размер массива

int n = scanner.nextInt();

int[] points = new int[n];

//читаем точки

for (int i = 0; i < n; i++) {

points[i] = scanner.nextInt();

}

//тут реализуйте логику задачи с применением сортировки подсчетом

55

Задание С.

a_khmelov

lesson01

lesson02

lesson03

lesson04

A_BinaryFind

B_MergeSort

C_GetInversions

dataA.txt

dataB.txt

dataC.txt

Lesson4Test

lesson05

A_QSort

B_CountSort

C_QSortOptimized

dataA.txt

dataB.txt

dataC.txt

Lesson4Test

alesnax

apilipenka

astro_emelya

belash_ea

chatovich

du4

ededovich.lesson1

grishkevich

hustlestar

jahstreet

khadasevich

makstra

mrlokans

poznyakbogdan

rudkouski

rudzko

1

2

3

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

package by.it.a_khmelov.lesson05;

import ...

/*

Видеорегистраторы и площадь 2.

Условие то же что и в задаче А.

По сравнению с задачей А доработайте алгоритм так, чтобы

1) он оптимально использовал время и память:

- за стек отвечает элиминация хвостовой рекурсии,

- за сам массив отрезков - сортировка на месте

- рекурсионные вызовы должны проводиться на основе 3-разбиения

2) при поиске подходящих отрезков для точки реализуйте метод бинарного поиска,

помните при реализации, что поиск множественный

(т.е. отрезков, подходящих для точки, может быть много)

Sample Input:

2 3

0 5

7 10

1 6 11

Sample Output:

1 0 0

*/

56