Алгоритмы и структуры данных

Сортировка

д.т.н., проф. Трифонов Петр Владимирович

Содержание лекции

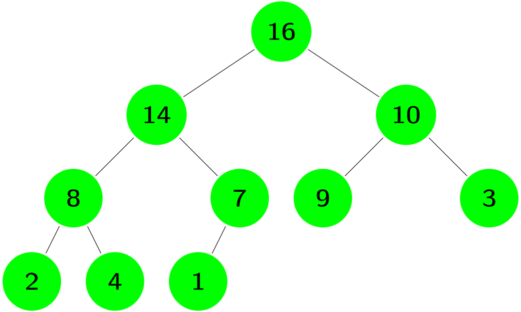
Задача сортировки

- Сортируемые объекты — структуры $(key, value)$
- На множестве ключей задано отношение строгого слабого порядка $\prec$
  - Иррефлексивное: $\forall x : x \nprec x$
  - Асимметричное: $\forall a, b : a \prec b \Rightarrow b \nprec a$
  - Транзитивное: $\forall a, b, c : a \prec b \wedge b \prec c \Rightarrow a \prec c$
  - $\forall x, y, z : ((x \nprec y \wedge y \nprec x) \wedge (y \nprec z \wedge z \nprec y)) \Rightarrow (x \nprec z \wedge z \nprec x)$
- Для простоты под $A \prec B$ будем понимать $A.key \prec B.key$
- Сложность измеряется в числе операций сравнения и обмена
- Внутренняя сортировка: весь сортируемый массив загружен в оперативную память
- Внешняя сортировка: данные записаны на медленном носителе, единовременно в памяти находится только часть сортируемых данных
- Устойчивая сортировка: если в исходном массиве $A.key = B.key$ и A расположено раньше B, то в отсортированном массиве A также расположено раньше B

Куча (пирамида)

Двоичная куча — массив $(A_0, \ldots, A_{n-1})$, для которого $A[Parent(i)] \succeq A[i], 0 < i < n$

- $Parent(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$
- $Left(i) = 2i + 1$
- $Right(i) = 2i + 2$
- $A[0]$ — корень дерева
- Высота дерева — число ребер в кратчайшем пути от корня до листа
- Высота вершины поддерева — высота поддерева с корнем в этой вершине
- Высота всего дерева — $\lfloor \log_2(n) \rfloor$



$A = (16, 14, 10, 8, 7, 9, 3, 2, 4, 1)$

Обеспечение основного свойства кучи $(A[Parent(i)] \succeq A[i], 0 < i < n)$

- Будем считать, что поддеревья с корнями $Left(i)$ и $Right(i)$ уже обладают свойством кучи
- Как переставить элементы поддерева с вершиной i, чтобы получилась куча?

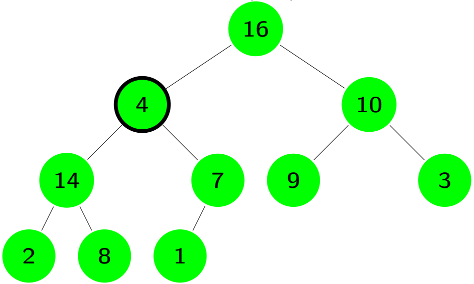
Algorithm 1: Heapify(A,i)

l=Left(i);r=right(i);
if $l < HeapSize \wedge A[l] \succ A[i]$ **then**
    largest=l
else
    largest=i
if $r < HeapSize \wedge A[r] \succ A[largest]$ **then**
    largest=r
if $largest \neq i$ **then**
    swap(A[i],A[largest]);
    Heapify(A,largest)

Обеспечение основного свойства кучи $(A[Parent(i)] \succeq A[i], 0 < i < n)$

- Будем считать, что поддеревья с корнями $Left(i)$ и $Right(i)$ уже обладают свойством кучи
- Как переставить элементы поддерева с вершиной i, чтобы получилась куча?

$Heapify(A, 1)$



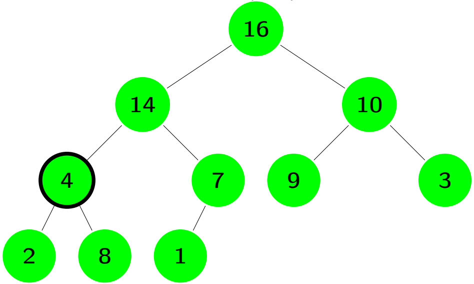
Algorithm 2: Heapify(A,i)

l=Left(i);r=right(i);
if $l < HeapSize \wedge A[l] \succ A[i]$ **then**
    largest=l
else
    largest=i
if $r < HeapSize \wedge A[r] \succ A[largest]$ **then**
    largest=r
if $largest \neq i$ **then**
    swap(A[i],A[largest]);
    Heapify(A,largest)

Обеспечение основного свойства кучи $(A[Parent(i)] \succeq A[i], 0 < i < n)$

- Будем считать, что поддеревья с корнями $Left(i)$ и $Right(i)$ уже обладают свойством кучи
- Как переставить элементы поддерева с вершиной i, чтобы получилась куча?

$Heapify(A, 3)$

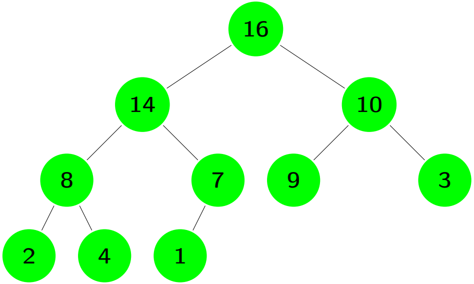


Algorithm 3: Heapify(A,i)

l=Left(i);r=right(i);
if $l < HeapSize \wedge A[l] \succ A[i]$ **then**
    largest=l
else
    largest=i
if $r < HeapSize \wedge A[r] \succ A[largest]$ **then**
    largest=r
if $largest \neq i$ **then**
    swap(A[i],A[largest]);
    Heapify(A,largest)

Обеспечение основного свойства кучи $(A[Parent(i)] \succeq A[i], 0 < i < n)$

- Будем считать, что поддеревья с корнями $Left(i)$ и $Right(i)$ уже обладают свойством кучи
- Как переставить элементы поддерева с вершиной i, чтобы получилась куча?



Algorithm 4: Heapify(A,i)

l=Left(i);r=right(i);
if $l < HeapSize \wedge A[l] \succ A[i]$ **then**
    largest=l
else
    largest=i
if $r < HeapSize \wedge A[r] \succ A[largest]$ **then**
    largest=r
if $largest \neq i$ **then**
    swap(A[i],A[largest]);
    Heapify(A,largest)

Обеспечение основного свойства кучи $(A[Parent(i)] \succeq A[i], 0 < i < n)$

- Будем считать, что поддеревья с корнями $Left(i)$ и $Right(i)$ уже обладают свойством кучи
- Как переставить элементы поддерева с вершиной i, чтобы получилась куча?
- Худший случай: левое поддерево содержит вдвое больше элементов, чем правое
- В дереве из n элементов число узлов в левом и правом поддеревьях не превосходит $2n/3$

$$T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\log(n))$$

Algorithm 5: Heapify(A,i)

l=Left(i);r=right(i);
if $l < HeapSize \wedge A[l] \succ A[i]$ **then**
    largest=l
else
    largest=i
if $r < HeapSize \wedge A[r] \succ A[largest]$ **then**
    largest=r
if $largest \neq i$ **then**
    swap(A[i],A[largest]);
    Heapify(A,largest)

Построение кучи

Algorithm 6: BuildHeap(A)

```
HeapSize=size(A);
for (i = ⌊(size(A) − 1)/2⌋ ; i ≥ 0; i − −) do
    Heapify(A,i)
```

- Для $n = size(A) \leq 3$ корректность очевидна
- Вершины $n/2, \ldots, n-1$ являются листьями, т.е. удовлетворяют свойству кучи
- Число вершин высоты h не превосходит $\lceil n/2^{h+1} \rceil$, высота кучи $\lfloor \log_2 n \rfloor$
- Время работы *Heapify* для вершины высоты h равно $O(h)$
- Время работы *BuildHeap* не превосходит

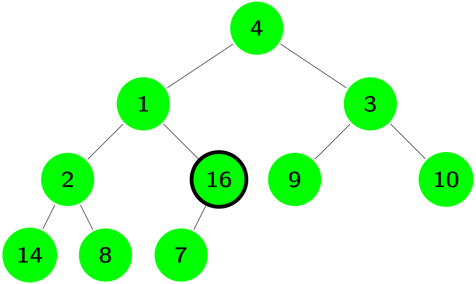
$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O\left(n \frac{1/2}{(1-1/2)^2}\right) = O(n)$$

Построение кучи

Algorithm 7: BuildHeap(A)

HeapSize=size(A);
for $(i = \lfloor (size(A) - 1)/2 \rfloor; i \geq 0; i - -)$ **do**
  Heapify(A,i)

$$A = (4, 1, 3, 2, 16, 9, 10, 14, 8, 7)$$



- Для $n = size(A) \leq 3$ корректность очевидна
- Вершины $n/2, \ldots, n-1$ являются листьями, т.е. удовлетворяют свойству кучи
- Число вершин высоты h не превосходит $\lceil n/2^{h+1} \rceil$, высота кучи $\lfloor \log_2 n \rfloor$
- Время работы *Heapify* для вершины высоты h равно $O(h)$
- Время работы *BuildHeap* не превосходит

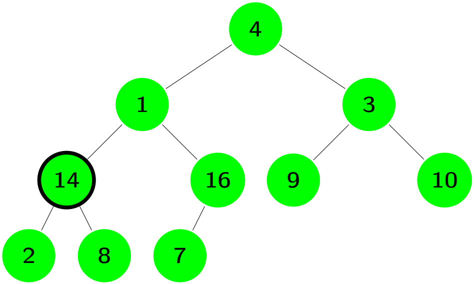
$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O\left(n \frac{1/2}{(1-1/2)^2}\right) = O(n)$$

Построение кучи

Algorithm 8: BuildHeap(A)

HeapSize=size(A);
for $(i = \lfloor (size(A) - 1)/2 \rfloor ; i \geq 0; i - -)$ **do**
  Heapify(A,i)

$$A = (4, 1, 3, 14, 16, 9, 10, 2, 8, 7)$$



- Для $n = size(A) \leq 3$ корректность очевидна
- Вершины $n/2, \dots, n-1$ являются листьями, т.е. удовлетворяют свойству кучи
- Число вершин высоты h не превосходит $\lceil n/2^{h+1} \rceil$, высота кучи $\lfloor \log_2 n \rfloor$
- Время работы *Heapify* для вершины высоты h равно $O(h)$
- Время работы *BuildHeap* не превосходит

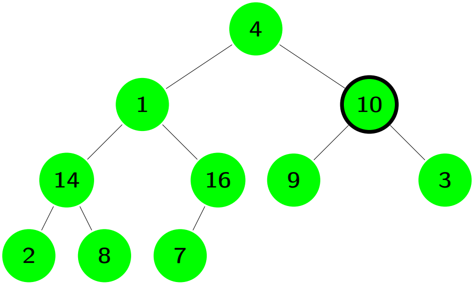
$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O\left(n \frac{1/2}{(1-1/2)^2}\right) = O(n)$$

Построение кучи

Algorithm 9: BuildHeap(A)

```
HeapSize=size(A);
for (i = ⌊(size(A) − 1)/2⌋ ; i ≥ 0; i − −) do
    Heapify(A,i)
```

$$A = (4, 1, 10, 14, 16, 9, 3, 2, 8, 7)$$



- Для $n = size(A) \leq 3$ корректность очевидна
- Вершины $n/2, \dots, n-1$ являются листьями, т.е. удовлетворяют свойству кучи
- Число вершин высоты h не превосходит $\lceil n/2^{h+1} \rceil$, высота кучи $\lfloor \log_2 n \rfloor$
- Время работы *Heapify* для вершины высоты h равно $O(h)$
- Время работы *BuildHeap* не превосходит

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) =$$

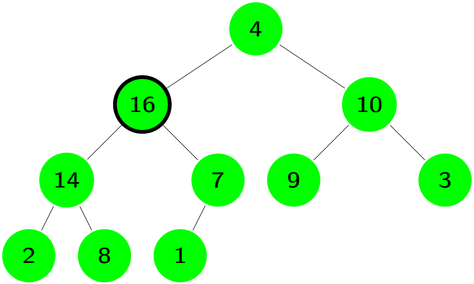
$$O\left(n \frac{1/2}{(1-1/2)^2}\right) = O(n)$$

Построение кучи

Algorithm 10: BuildHeap(A)

```
HeapSize=size(A);
for (i = ⌊(size(A) − 1)/2⌋ ; i ≥ 0; i − −) do
    Heapify(A,i)
```

$$A = (4, 16, 10, 14, 7, 9, 3, 2, 8, 1)$$



- Для $n = size(A) \le 3$ корректность очевидна
- Вершины $n/2, \ldots, n-1$ являются листьями, т.е. удовлетворяют свойству кучи
- Число вершин высоты h не превосходит $\lceil n/2^{h+1} \rceil$, высота кучи $\lfloor \log_2 n \rfloor$
- Время работы *Heapify* для вершины высоты h равно $O(h)$
- Время работы *BuildHeap* не превосходит

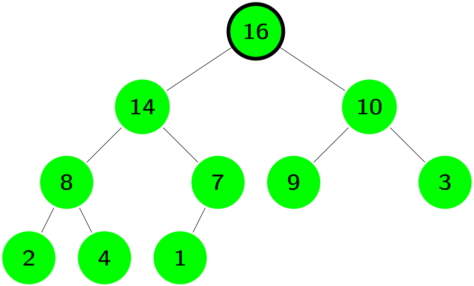
$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O\left(n \frac{1/2}{(1-1/2)^2}\right) = O(n)$$

Построение кучи

Algorithm 11: BuildHeap(A)

HeapSize=size(A);
for $(i = \lfloor (size(A) - 1)/2 \rfloor ; i \geq 0; i - -)$ **do**
    Heapify(A,i)

$$A = (16, 14, 10, 8, 7, 9, 3, 2, 4, 1)$$



- Для $n = size(A) \leq 3$ корректность очевидна
- Вершины $n/2, \ldots, n-1$ являются листьями, т.е. удовлетворяют свойству кучи
- Число вершин высоты h не превосходит $\lceil n/2^{h+1} \rceil$, высота кучи $\lfloor \log_2 n \rfloor$
- Время работы *Heapify* для вершины высоты h равно $O(h)$
- Время работы *BuildHeap* не превосходит
$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O\left(n \frac{1/2}{(1-1/2)^2}\right) = O(n)$$

Производная геометрической прогрессии

$$\sum_{i=0}^{t} a^i = \frac{a^{t+1} - 1}{a - 1}$$

$$\sum_{i=0}^{t} i a^{i-1} = \frac{d}{da} \sum_{i=0}^{t} a^i = \frac{d}{da} \frac{a^{t+1} - 1}{a - 1} = \frac{(t+1)a^t(a-1) - (a^{t+1} - 1)}{(a-1)^2}$$

Сортировка с помощью кучи

1. Построим кучу
2. Поменяем местами корень дерева (максимальный элемент) и элемент в ячейке $n-1$
3. Восстановим свойство кучи для начальных $n-1$ элементов массива
4. Повторить п. 2–3 для первых $n-1$ элементов массива

Сложность $O(n \log n)$

Algorithm 12: HeapSort(A)

```
BuildHeap(A)
for (i = n − 1; i > 0; i − −) do
    swap(A[0],A[i])
    HeapSize − −
    Heapify(A,0)
```

Приоритетная очередь

Приоритетная очередь — структура данных, предназначенная для обслуживания множества $S = \{(key, value)\}$ с определенным на ним строгим слабым порядком, поддерживающая операции

- $Insert(S, x) : S \leftarrow S \cup \{x\}$
- $Maximum(S)$: возвращает $\max S$
- $ExtractMax(S)$: возвращает $\max S$ и удаляет соответствующий элемент из S
- $IncreaseKey(S, x, k)$: увеличивает значение ключа элемента x путем его замены на k, т.е. $x.key \leftarrow k, x.key \prec k$

Реализация приоритетной очереди на основе кучи

Algorithm 13: Maximum(A)

return A[0]

Сложность $\Theta(1)$

Algorithm 14: ExtractMax(A)

if $HeapSize < 1$ **then**
    Error: очередь пуста
max=A[0];A[0]=A[HeapSize-1];
HeapSize=HeapSize-1;
Heapify(A,0);
return max

Сложность $O(\log(n))$

Algorithm 15: IncreaseKey(A,i,k)

if $k \prec A[i].key$ **then**
    Error: Новый ключ меньше предыдущего
A[i].key=k;
while $i > 0 \wedge A[Parent(i)] \prec A[i]$ **do**
    swap(A[i],A[Parent(i)]); i=Parent(i)

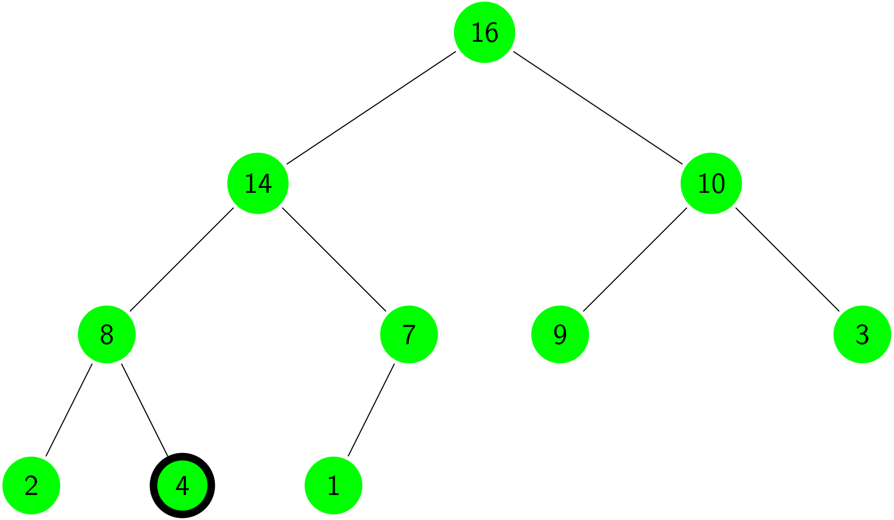
Сложность $O(\log n)$

Algorithm 16: Insert(A,(key,value))

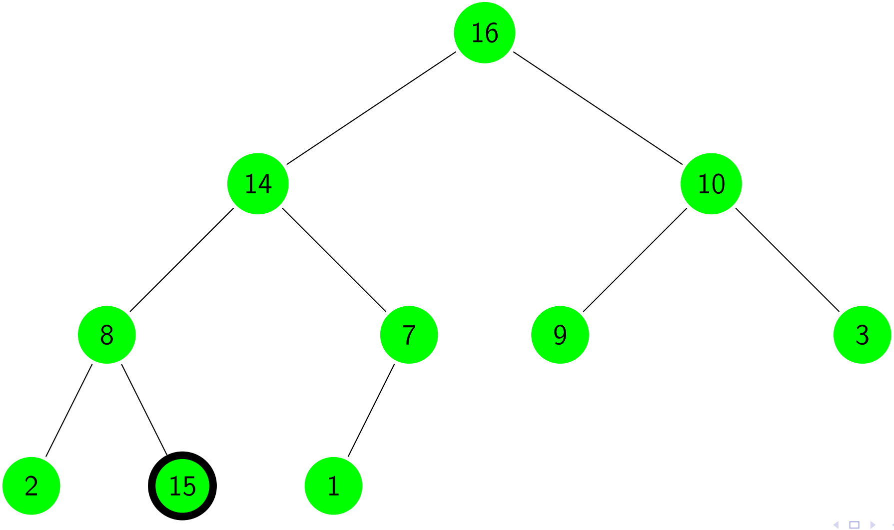
A[HeapSize]=$(-\infty, value)$;
HeapSize=HeapSize+1;
IncreaseKey(A,HeapSize-1,key)

Сложность $O(\log n)$

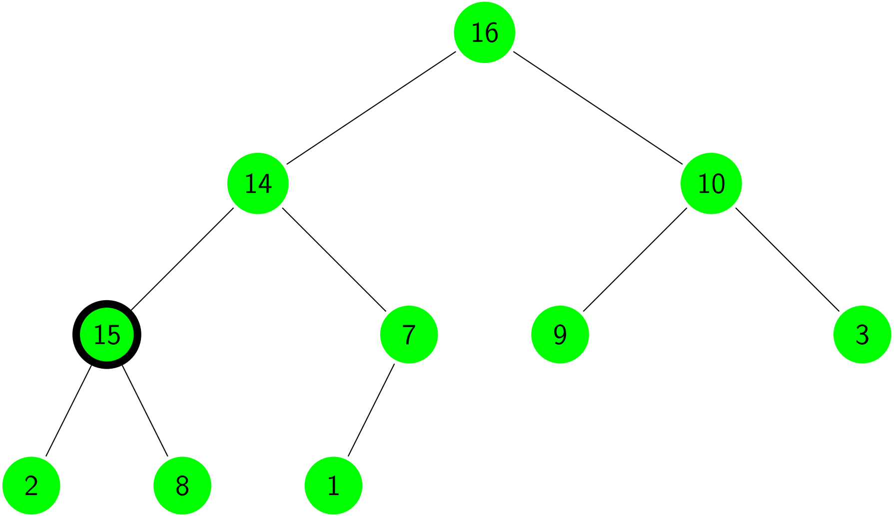
Пример работы *IncreaseKey*



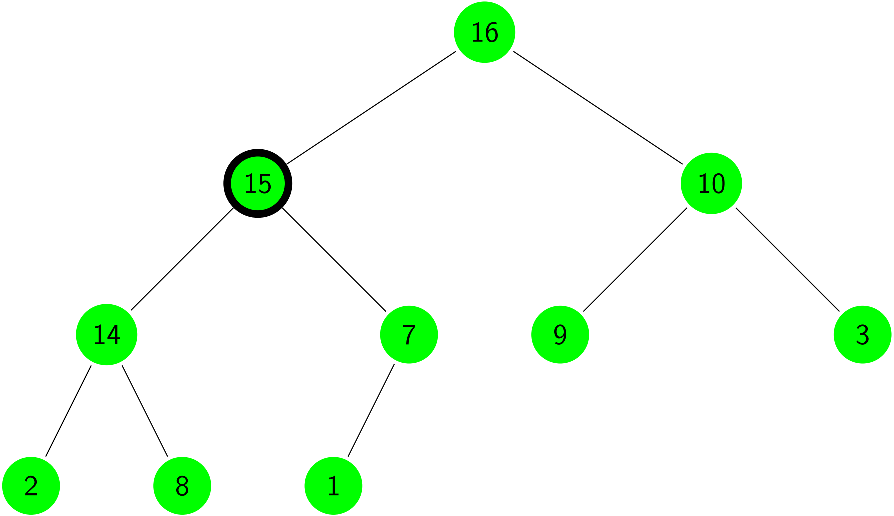
Пример работы *IncreaseKey*



Пример работы *IncreaseKey*



Пример работы *IncreaseKey*



Быстрая сортировка

Сортировка массива $(A[p], \ldots, A[r])$

- Переставим элементы массива так, чтобы $A[p], A[p+1], \ldots, A[q] \preceq A[q+1], A[q+2], \ldots, A[r]$, где $p \leq q < r$
- Вызовем рекурсивно процедуру сортировки для массивов $(A[p], \ldots, A[q])$ и $(A[q+1], \ldots, A[r])$

Algorithm 17: QuickSort(A,p,r)

```
if p < r then
    q=Partition(A,p,r);
    QuickSort(A,p,q);
    QuickSort(A,q+1,r);
```

Как выбрать q?

Разбиение массива A[p..r]

	5	3	2	6	4	1	3	7	
$\uparrow i$									$\uparrow j$
	5	3	2	6	4	1	3	7	
$\uparrow i$							$\uparrow j$		
	5	3	2	6	4	1	3	7	
	$\uparrow i$						$\uparrow j$		
	3	3	2	6	4	1	5	7	
	$\uparrow i$						$\uparrow j$		
	3	3	2	6	4	1	5	7	
	$\uparrow i$					$\uparrow j$			
	3	3	2	6	4	1	5	7	
				$\uparrow i$		$\uparrow j$			
	3	3	2	1	4	6	5	7	
				$\uparrow i$		$\uparrow j$			
	3	3	2	1	4	6	5	7	
					$\uparrow j$	$\uparrow i$			

Algorithm 18: Partition(A,p,r)

```
x=A[p];i=p-1;j=r+1;
while true do
    repeat
        j=j-1
    until A[j] ⪯ x;
    repeat
        i=i+1
    until A[i] ⪰ x;
    if i < j then
        swap(A[i],A[j])
    else
        return j
```

Сложность $\Theta(r - p + 1)$

Корректность алгоритма разбиения ($p \le r$)

- На строке 9 всегда выполняется $p \le i, j \le r$
  К первому проходу выполняется $p \le j \le r, i = p$
  В начале последующих итераций WHILE $p \le i < j$
  На каждой итерации WHILE значение $A[p]$ не увеличивается
  $\Rightarrow p \le j$ на строке 9
  После первой итерации $A[r] \succeq x \Rightarrow i \le r$ на строке 9
- Если $p < r$, то на строке 12 $j < r$
- На строке 12 $A[s] \preceq A[t], p \le s \le j, j+1 \le t \le r$

Algorithm 19: Partition(A,p,r)

```
1  x=A[p];i=p-1;j=r+1;
2  while true do
3      repeat
4          j=j-1
5      until A[j] ⪯ x;
6      repeat
7          i=i+1
8      until A[i] ⪰ x;
9      if i < j then
10         swap(A[i],A[j])
11     else
12         return j
```

Время работы: крайние случаи

- Худший случай: на каждом шаге одна часть разбиения содержит 1 элемент, вторая — $n-1$
  - $T(n) = \max_{1\leq q\leq n-1}(T(q)+T(n-q))+\Theta(n)$
  - Предположим, что
    $T(q)\leq cq^2 \Rightarrow T(n)\leq c\max\limits_{1\leq q<n}(q^2+(n-q)^2)+\Theta(n)\leq c(1+(n-1)^2)+\Theta(n)$
    $\Rightarrow T(n)\leq cn^2-2c(n-1)+\Theta(n)\leq cn^2$ (для достаточно большого c)

Время работы: крайние случаи

- Худший случай: на каждом шаге одна часть разбиения содержит 1 элемент, вторая — $n-1$
  - $T(n) = \max_{1 \leq q \leq n-1}(T(q) + T(n-q)) + \Theta(n)$
  - Предположим, что $T(q) \leq cq^2 \Rightarrow T(n) \leq c \max\limits_{1 \leq q < n} (q^2 + (n-q)^2) + \Theta(n) \leq c(1 + (n-1)^2) + \Theta(n)$ $\Rightarrow T(n) \leq cn^2 - 2c(n-1) + \Theta(n) \leq cn^2$ (для достаточно большого c)
  - $T(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \Theta(n^2)$

Время работы: крайние случаи

- Худший случай: на каждом шаге одна часть разбиения содержит 1 элемент, вторая — $n-1$
  - $T(n) = \max_{1\leq q\leq n-1}(T(q)+T(n-q))+\Theta(n)$
  - Предположим, что $T(q)\leq cq^2 \Rightarrow T(n)\leq c\max\limits_{1\leq q<n}(q^2+(n-q)^2)+\Theta(n)\leq c(1+(n-1)^2)+\Theta(n)$ $\Rightarrow T(n)\leq cn^2-2c(n-1)+\Theta(n)\leq cn^2$ (для достаточно большого c)
  - $T(n)=T(n-1)+\Theta(n)=\sum_{k=1}^n \Theta(k)=\Theta(\sum_{k=1}^n k)=\Theta(n^2)$
  - Данный случай возникает, в частности, для отсортированных массивов (сортировка вставками при этом имеет сложность $\Theta(n)$

Время работы: крайние случаи

- Худший случай: на каждом шаге одна часть разбиения содержит 1 элемент, вторая — $n-1$
  - $T(n) = \max_{1 \le q \le n-1}(T(q) + T(n-q)) + \Theta(n)$
  - Предположим, что $T(q) \le cq^2 \Rightarrow T(n) \le c \max\limits_{1 \le q < n}(q^2 + (n-q)^2) + \Theta(n) \le c(1 + (n-1)^2) + \Theta(n)$ $\Rightarrow T(n) \le cn^2 - 2c(n-1) + \Theta(n) \le cn^2$ (для достаточно большого c)
  - $T(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \Theta(n^2)$
  - Данный случай возникает, в частности, для отсортированных массивов (сортировка вставками при этом имеет сложность $\Theta(n)$
- Лучший случай: на каждом шаге разбиение на две равные части

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$$

Рандомизированный алгоритм

На практике вероятность появления отсортированных массивов существенно больше $2/n! \Rightarrow$ применяют рандомизированный алгоритм

Algorithm 20: RandPartition(A,p,r)

```
i=Random(p,r);
swap(A[p],A[i]);
return Partition(A,p,r)
```

Algorithm 21: RandQSort(A,p,r)

```
if p < r then
    q=RandPartition(A,p,r);
    RandQSort(A,p,q);
    RandQSort(A,q+1,r);
```

Анализ среднего времени работы (рандомизированный алгоритм)

- Будем считать, что все элементы $(A[0], \ldots, A[n-1])$ различны
- Значение, возвращаемое $Partition(A, 0, n-1)$, определяется числом элементов $A[i] \preceq x$ $(\text{rank}(x))$, где $x = A[p]$. $P\{\text{rank}(x) = i\} = 1/n, 0 \le i < n$
- $\text{rank}(x) = 0 \Rightarrow$ левая часть разбиения содержит 1 элемент (x)
- $\text{rank}(x) > 0 \Rightarrow$ левая часть содержит $\text{rank}(x)$ элементов
- Число элементов в левой части: $q = 1$ с вероятностью $P_1 = 2/n$, и $q : 1 < q < n$ с вероятностью $P_q = 1/n$
- Среднее время работы

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

$$= \frac{1}{n}O(n^2) + \frac{1}{n}\sum_{q=1}^{n-1}(T(q) + T(n-q)) + \Theta(n) = \frac{2}{n}\sum_{q=1}^{n-1} T(q) + \Theta(n) = O(n \log n)$$

Решение рекуррентного уравнения

Предположим, что $T(n) \le an \log n + b$. Это верно при $n = 1$ при достаточно больших a, b

$$T(n) = \frac{2}{n}\sum_{q=1}^{n-1} T(q) + \Theta(n) \le \frac{2}{n}\sum_{q=1}^{n-1}(aq\log q + b) + \Theta(n) = \frac{2a}{n}\sum_{q=1}^{n-1} q\log q + 2b\frac{n-1}{n} + \Theta(n) \le$$

$$\le \frac{2a}{n}\left(\sum_{q=1}^{\lceil \frac{n}{2}\rceil - 1} q\log q + \sum_{q=\lceil \frac{n}{2}\rceil}^{n-1} q\log q\right) + 2b + \Theta(n) \le$$

$$\le \frac{2a}{n}\left(\sum_{q=1}^{\lceil \frac{n}{2}\rceil - 1} q(\log n - 1) + \sum_{q=\lceil \frac{n}{2}\rceil}^{n-1} q\log n\right) + 2b + \Theta(n) \le$$

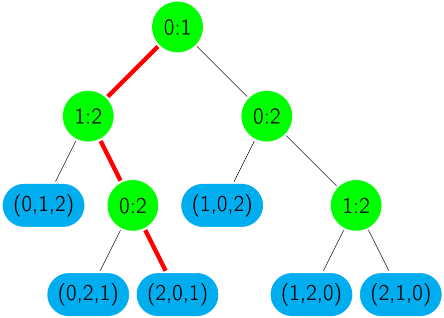
$$\le \frac{2a}{n}\left(\log n\sum_{q=1}^{n-1} q - \sum_{q=1}^{\lceil \frac{n}{2}\rceil - 1} q\right) + 2b + \Theta(n) \le \frac{2a}{n}\left(\frac{n(n-1)}{2}\log n - \frac{1}{2}(\frac{n}{2} - 1)\frac{n}{2}\right) + 2b + \Theta(n) \le$$

$$\le an\log n - an/4 + 2b + \Theta(n) = an\log n + b + (\Theta(n) + b - an/4) \le an\log n + b$$

Модель дерева решений

- Дерево решений — двоичное дерево, в котором представлены операции сравнения, выполняемые некоторым алгоритмом сортировки
- Внутренние узлы помечены $i : j, 0 \le i, j < n$. Это означает сравнение $A[i], A[j]$
  - Левое поддерево: $A[i] \preceq A[j]$
  - Правое поддерево: $A[i] \succ A[j]$
- Листья помечены перестановками $(\pi(0), \pi(1), \ldots, \pi(n-1))$, задающими всевозможные упорядочения $A[\pi(0)] \preceq A[\pi(1)] \preceq \cdots \preceq A[\pi(n-1)]$
- Выполнение алгоритма сортировки — проход от корня дерева к одному из листьев
- Корректный алгоритм сортировки способен реализовать любую из $n!$ перестановок

Дерево решений для алгоритма сортировки вставками



$(6, 8, 5) \to (5_2, 6_0, 8_1)$

Нижняя оценка сложности сортировки в худшем случае

Теорема

В наихудшем случае сложность любого алгоритма сортировки сравнением составляет $\Omega(n \log n)$ сравнений

Доказательство.

- Сортировка n элементов соответствует дереву решений высоты h с l достижимыми листьями
- Каждая из $n!$ перестановок соответствует одному из листьев $\Rightarrow n! \leq l$
- Число листьев в двоичном дереве высоты h удовлетворяет $l \leq 2^h$
- $n! \leq 2^h \Rightarrow h \geq \log(n!) = \log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right)\right) = \Theta(n \log n)$

□

Следствие

Пирамидальная сортировка (сортировка на куче) и сортировка слиянием — асимптотически

Сортировка подсчетом

- Пусть ключи сортируемых элементов $A[i].key \in \mathcal{Z}, \mathcal{Z} = \{0, 1, 2, \ldots, k\}, k = O(n)$
- Подсчитаем количество вхождений каждого числа из $\mathcal{Z}$ в сортируемый массив
- Выпишем каждое число из $\mathcal{Z}$ столько раз, сколько оно встретилось
- Сложность $\Theta(n + k)$; алгоритм устойчив

Algorithm 22: CountingSort(A,B,k)

for $(i = 0; i \leq k; i++)$ **do**
    C[i]=0
for $(j = 0; j < length(A); j++)$ **do**
    C[A[j].key]++
for $(i = 1; i \leq k; i++)$ **do**
    C[i]+=C[i-1]
for $(j = length(A) - 1; j \geq 0; j--)$ **do**
    $C[A[j]]--;$
    B[C[A[j].key]]=A[j];

$A = (2, 5, 3, 0, 2, 3, 0, 3)$
$C = (2, 2, 4, 7, 7, 8)$
$B = (0, 0, 2, 2, 3, 3, 3, 5)$

Сортировка подсчетом

- Пусть ключи сортируемых элементов $A[i].key \in \mathcal{Z}, \mathcal{Z} = \{0, 1, 2, \ldots, k\}, k = O(n)$
- Подсчитаем количество вхождений каждого числа из $\mathcal{Z}$ в сортируемый массив
- Выпишем каждое число из $\mathcal{Z}$ столько раз, сколько оно встретилось
- Сложность $\Theta(n + k)$; алгоритм устойчив
- Это не противоречит вышедоказанной теореме
  - Алгоритм не использует сравнения

Algorithm 23: CountingSort(A,B,k)

```
for (i = 0; i ≤ k; i++) do
    C[i]=0
for (j = 0; j < length(A); j++) do
    C[A[j].key]++
for (i = 1; i ≤ k; i++) do
    C[i]+=C[i-1]
for (j = length(A) − 1; j ≥ 0; j−−) do
    C[A[j]]−−;
    B[C[A[j].key]]=A[j];
```

$A = (2, 5, 3, 0, 2, 3, 0, 3)$
$C = (2, 2, 4, 7, 7, 8)$
$B = (0, 0, 2, 2, 3, 3, 3, 5)$

Сортировка подсчетом

- Пусть ключи сортируемых элементов $A[i].key \in \mathcal{Z}, \mathcal{Z} = \{0, 1, 2, \ldots, k\}, k = O(n)$
- Подсчитаем количество вхождений каждого числа из $\mathcal{Z}$ в сортируемый массив
- Выпишем каждое число из $\mathcal{Z}$ столько раз, сколько оно встретилось
- Сложность $\Theta(n+k)$; алгоритм устойчив
- Это не противоречит вышедоказанной теореме
  - Алгоритм не использует сравнения
  - Если $n > 2^h$, где h — разрядность ЭВМ, то для реализации операций с $C[i]$ понадобится длинная арифметика с числом разрядов $\eta = \lceil \log_h(n) \rceil$ и сложностью операций $\Theta(\eta) \Rightarrow$ сложность сортировки $\Theta(n \log n)$

Algorithm 24: CountingSort(A,B,k)

for $(i = 0; i \leq k; i++)$ **do**
    C[i]=0
for $(j = 0; j < length(A); j++)$ **do**
    C[A[j].key]++
for $(i = 1; i \leq k; i++)$ **do**
    C[i]+=C[i-1]
for $(j = length(A) - 1; j \geq 0; j--)$ **do**
    $C[A[j]]--;$
    B[C[A[j].key]]=A[j];

$A = (2, 5, 3, 0, 2, 3, 0, 3)$
$C = (2, 2, 4, 7, 7, 8)$
$B = (0, 0, 2, 2, 3, 3, 3, 5)$

Поразрядная сортировка

- Пусть даны n натуральных чисел $\sum_{j=0}^{d-1} a_{ij} b^j, 0 \leq a_{ij} < b$
- Отсортируем (устойчиво!) их по младшей цифре a_{i0}, объединим в массив
- Отсортируем (устойчиво!) их по a_{i1}, объединим в массив
  ...
- Корректность: доказательство индукцией по d
- Если сортировка по каждой цифре имеет сложность $\Theta(n+b)$, то алгоритм имеет сложность $\Theta(d(n+b))$

$$
\begin{array}{ccccccc}
3\,2\,9 & & 7\,2\,0 & & 7\,2\,0 & & 3\,2\,9 \\
4\,5\,7 & & 3\,5\,5 & & 3\,2\,9 & & 3\,5\,5 \\
6\,5\,7 & & 4\,3\,6 & & 4\,3\,6 & & 4\,3\,6 \\
8\,3\,9 & \rightarrow & 4\,5\,7 & \rightarrow & 8\,3\,9 & \rightarrow & 4\,5\,7 \\
4\,3\,6 & & 6\,5\,7 & & 3\,5\,5 & & 6\,5\,7 \\
7\,2\,0 & & 3\,2\,9 & & 4\,5\,7 & & 7\,2\,0 \\
3\,5\,5 & & 8\,3\,9 & & 6\,5\,7 & & 8\,3\,9
\end{array}
$$

Лемма

Пусть даны n чисел разрядности h и число $r \leq h$. Их поразрядная сортировка имеет сложность $\Theta((b/r)(n+2^r))$

Карманная сортировка (сортировка вычерпыванием)

- Рассмотрим сортировку значений x, равномерно распределенных на $[0,1)$
- Разложим сортируемые числа по карманам (ведрам, ящикам, ...), соответствующим интервалам $\left[\frac{i}{n}, \frac{i+1}{n}\right), 0 \leq i < n$
- Карман — список
- Отсортируем числа в каждом кармане
- Последовательно выгрузим содержимое всех карманов

Algorithm 25: BucketSort(A)

```
n=length(A);j=0;
for (i = 0; i < n; i++) do
    Append(B_⌊nA[i]⌋, A[i]);
for (i = 0; i < n; i++) do
    Сортировка вставками B_i;
    Переместить содержимое B_i в A[j], ..., A[j + |B_i| − 1];
    j := j + |B_i|
```

$$A = (0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68)$$

i	0	1	2	3	4	5	6	7	8	9
B_i		0.12 0.17	0.21 0.23	0.39			0.68 0.78	0.72		0.94

Сложность карманной сортировки

- Сложность сортировки кармана $O(n_i^2)$, где n_i — число элементов в B_i
- Матожидание суммарной сложности сортировки
  $M[C] = \sum_{i=0}^{n-1} O(M[n_i^2]) = O(\sum_{i=0}^{n-1} M[n_i^2])$
- Вероятность попадания $A[j]$ в $B[i]$ равна $p = 1/n$
- $P\{n_i = k\} = C_n^k p^k (1-p)^{n-k}, 0 \leq k \leq n$
- $M[n_i] = np = 1$. $D[n_i] = np(1-p) = 1 - 1/n$

$$M[n_i^2] = D[n_i] + (M[n_i])^2 = 1 - 1/n + 1 = 2 - 1/n = \Theta(1)$$

- Средняя сложность сортировки $M[C] = O(n)$

Минимум и максимум

Algorithm 26: Minimum(A)

```
min=A[0];
for (i=1;i<length(A);i++) do
    if A[i] ≺ min then
        min=A[i]
return min
```

- Нахождение минимума (максимума) требует ровно $n-1$ сравнений
- $n-1$ элементов должны “проиграть” минимальному

Algorithm 27: MinimumAndMaximum(A)

```
min = A[0]; max = A[1 − (length(A) mod 2)];
if max ≺ min then
    swap(min,max)
for (i = 2 − (length(A) mod 2); i < length(A); i+ = 2) do
    a=A[i];b=A[i+1];
    if b ≺ a then
        swap(a,b)
    if a ≺ min then
        min=a
    if max ≺ b then
        max=b
return (min,max)
```

$3\lceil n/2\rceil - 2$ сравнений

Порядковые статистики

- i-ая порядковая статистика множества (массива) A — его i-ый элемент в порядке возрастания, $0 \leq i < |A|$
  - $\mathrm{ord}_2(30, 3, 5, 10) = 10$
- Минимум — $\mathrm{ord}_0(A)$
- Максимум — $\mathrm{ord}_{|A|-1}(A)$
- Медиана: $\begin{cases} \mathrm{ord}_{(|A|-1)/2}(A), & |A| \text{ нечетная} \\ \{\underbrace{\mathrm{ord}_{|A|/2-1}(A)}_{\text{нижняя медиана}},\ \underbrace{\mathrm{ord}_{|A|/2}(A)}_{\text{верхняя медиана}}\}, & |A| \text{ четная} \end{cases}$
- Далее будем рассматривать нижнюю медиану
- Выбор i-ой порядковой статистики может быть выполнен со сложностью $O(n \log n)$ посредством сортировки, но это неэффективно

Выбор i-ой порядковой статистики с линейной сложностью

Algorithm 28: Partition(A,p,r)

```
x=A[p];i=p-1;j=r+1;
while true do
    repeat
        j=j-1
    until A[j] ⪯ x;
    repeat
        i=i+1
    until A[i] ⪰ x;
    if i < j then
        swap(A[i],A[j])
    else
        return j
```

Сложность $\Theta(r-p+1)$

Algorithm 29: RandPartition(A,p,r)

```
i=Random(p,r);
swap(A[p],A[i]);
return Partition(A,p,r)
```

Algorithm 30: RandSelect(A,p,r,i)

```
if p=r then
    return A[p]
q=RandPartition(A,p,r);
k=q-p+1;
if i=k then
    return A[q]
else if i < k then
    return RandSelect(A,p,q-1,i)
else
    return RandSelect(A,q+1,r,i-k)
```

Выбор *i*-ой порядковой статистики с линейной сложностью

- *RandPartition* разбивает массив на подмассивы $A[p..q-1] \prec A[q+1..r]$
- $A[q]$ — опорный элемент
- Строка 5: не является ли $A[q]$ искомым значением?
- Строка 8: искомое значение в подмассиве $A[p..q-1]$
- Строка 10: искомое значение в подмассиве $A[q+1..r]$

Algorithm 31: RandSelect(A,p,r,i)

```
1  if p=r then
2      return A[p]
3  q=RandPartition(A,p,r);
4  k=q-p+1;
5  if i=k then
6      return A[q]
7  else if i < k then
8      return RandSelect(A,p,q-1,i)
9  else
10     return RandSelect(A,q+1,r,i-k)
```

Выбор i-ой порядковой статистики с линейной сложностью: сложность

- Худший случай: *RandPartition* всегда возвращает номер наибольшего элемента $\Rightarrow \Theta(n^2), n = r - p + 1$
- $P\{q - p + 1 = k\} = 1/n, 1 \leq k \leq n$
- $\chi_k = \begin{cases} 1, & q - p + 1 = k \\ 0, & \text{иначе} \end{cases}$
- $M[\chi_k] = 1/n$
- Предположим, что искомый элемент всегда попадает в самую большую часть разбиения

$$T(n) \leq \sum_{k=1}^{n} \chi_k \left(T(\max(k-1, n-k)) + O(n)\right)$$

Algorithm 32: RandSelect(A,p,r,i)

if p=r **then**
    return A[p]
q=RandPartition(A,p,r);
k=q-p+1;
if i=k **then**
    return A[q]
else if $i < k$ **then**
    return *RandSelect(A,p,q-1,i)*
else
    return *RandSelect(A,q+1,r,i-k)*

Выбор i-ой порядковой статистики с линейной сложностью: сложность

$$T(n) \leq \sum_{k=1}^{n} \chi_k T(\max(k-1, n-k)) + O(n)$$

$$M[T(n)] \leq \sum_{k=1}^{n} M\left[\chi_k T(\max(k-1, n-k))\right] + O(n) = \sum_{k=1}^{n} M\left[\chi_k\right] M\left[T(\max(k-1, n-k))\right] + O(n) =$$

$$= \sum_{k=1}^{n} \frac{1}{n} M\left[T(\max(k-1, n-k))\right] + O(n)$$

- χ_k и $T(\max(k-1, n-k))$ — независимые случайные величины
- $\max(k-1, n-k) = \begin{cases} k-1, & k > \lceil n/2 \rceil \\ n-k, & k \leq \lceil n/2 \rceil \end{cases} \Rightarrow$ для четного n каждое слагаемое $T(\lceil n/2 \rceil), \ldots, T(n-1)$ появляется дважды

Выбор i-ой порядковой статистики с линейной сложностью: сложность

$$M[T(n)] \le \frac{2}{n}\sum_{k=\lfloor n/2\rfloor}^{n-1} M[T(k)] + O(n)$$

Предположим, что $M[T(n)] \le cn$, а для малых $n:\ T(n) = O(1)$

$$M[T(n)] \le \frac{2}{n}\sum_{k=\lfloor n/2\rfloor}^{n-1} ck + an = \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2\rfloor-1} k\right) + an =$$

$$= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2\rfloor - 1)\lfloor n/2\rfloor}{2}\right) + an \le \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + an =$$

$$= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{n^2/4 - 3n/2 + 2}{2}\right) + an = \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an = c(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}) + an \le$$

$$\le 3cn/4 + c/2 + an = cn - (cn/4 - c/2 - an)$$

Нужно показать, что при больших n: $cn/4 - c/2 - an \ge 0 \Leftrightarrow n(c/4 - a) \ge c/2$. $c > 4a \Rightarrow n \ge \frac{c/2}{c/4-a}$

Заключение

- Сортировка может быть выполнена со сложностью $O(n \log n)$
- Иногда сортировка может быть выполнена со сложностью $O(n)$
- Порядковые статистики могут быть найдены со сложностью $O(n)$