

§1. Сортировка вставками

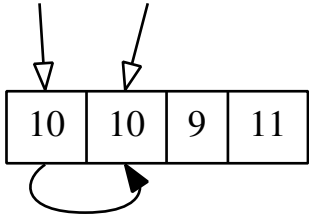
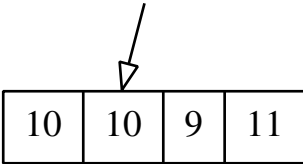
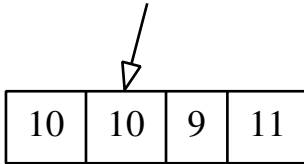
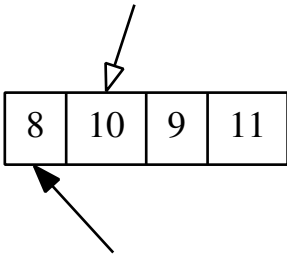
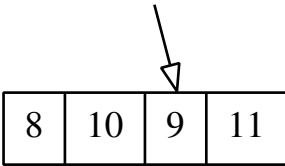
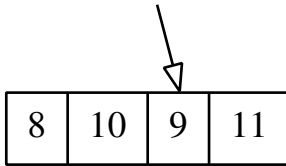
Основная идея алгоритма: сортируемая последовательность рассматривается как состоящая из двух частей – уже отсортированной и ещё неотсортированной; на каждом шаге алгоритма первый элемент неотсортированной части вставляется в нужное место отсортированной части; тем самым неотсортированная часть уменьшается до тех пор, пока не станет пустой.

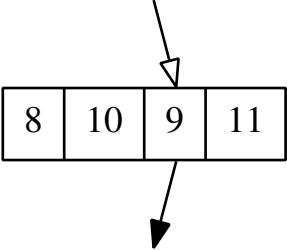
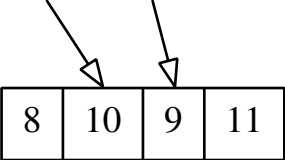
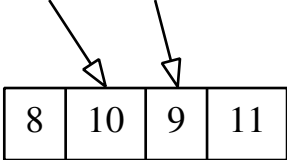
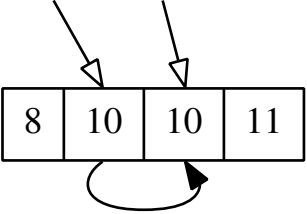
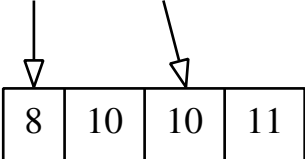
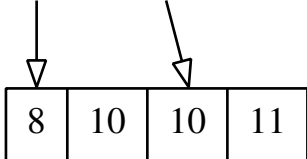
```
1 InsertSort ( in / out  $\{a[i]\}_0^{n-1}$  )  
2      $i \leftarrow 1$   
3     while  $i < n$  :  
4          $elem \leftarrow a[i]$   
5          $loc \leftarrow i - 1$   
6         while  $loc \geq 0$  and  $a[loc] > elem$  :  
7              $a[loc + 1] \leftarrow a[loc]$   
8              $loc \leftarrow loc - 1$   
9          $a[loc + 1] \leftarrow elem$   
10     $i \leftarrow i + 1$ 
```

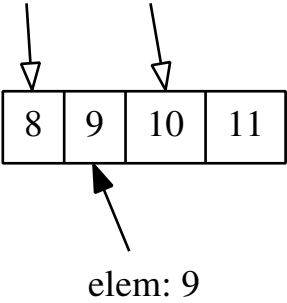
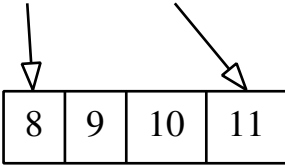
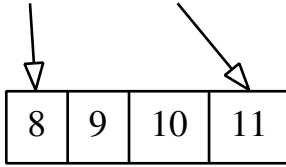
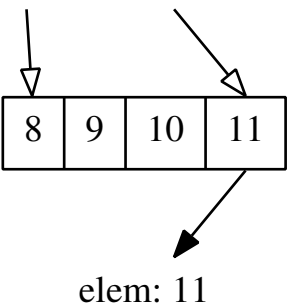
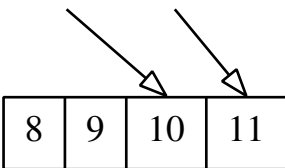
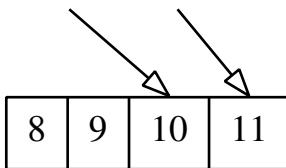
Пример.

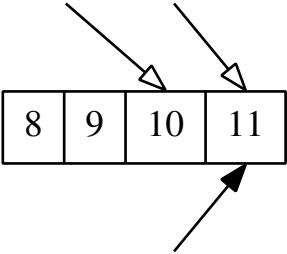
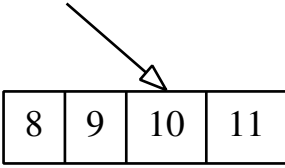
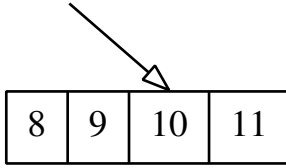
InsertSort([10, 8, 9, 11])

<p>loc: ? i: ? n: 4</p> <div><div>108911</div></div> <p>elem: ?</p>	<p>loc: ? i: 1 n: 4</p> <div><div>108911</div></div> <p>elem: ?</p>	<p>loc: ? i: 1 n: 4</p> <div><div>108911</div></div> <p>elem: ?</p>
	1. $i \leftarrow 1$	2. $i < n$?
<p>loc: ? i: 1 n: 4</p> <div><div>108911</div></div> <p>elem: 8</p>	<p>loc: 0 i: 1 n: 4</p> <div><div>108911</div></div> <p>elem: 8</p>	<p>loc: 0 i: 1 n: 4</p> <div><div>108911</div></div> <p>elem: 8</p>
3. $elem \leftarrow a[i]$	4. $loc \leftarrow i - 1$	5. $loc \geq 0$ and $a[loc] > elem$?

<p>loc: 0 i: 1 n: 4</p>  <p>elem: 8</p>	<p>loc: -1 i: 1 n: 4</p>  <p>elem: 8</p>	<p>loc: -1 i: 1 n: 4</p>  <p>elem: 8</p>
6. $a[loc + 1] \leftarrow a[loc]$	7. $loc \leftarrow loc - 1$	8. $loc \geq 0$ and $a[loc] > elem$?
<p>loc: -1 i: 1 n: 4</p>  <p>elem: 8</p>	<p>loc: -1 i: 2 n: 4</p>  <p>elem: 8</p>	<p>loc: -1 i: 2 n: 4</p>  <p>elem: 8</p>
9. $a[loc + 1] \leftarrow elem$	10. $i \leftarrow i + 1$	11. $i < n$?

<p>loc: -1 i: 2 n: 4</p>  <p>elem: 9</p>	<p>loc: 1 i: 2 n: 4</p>  <p>elem: 9</p>	<p>loc: 1 i: 2 n: 4</p>  <p>elem: 9</p>
12. $elem \leftarrow a[i]$	13. $loc \leftarrow i - 1$	14. $loc \geq 0$ and $a[loc] > elem$?
<p>loc: 1 i: 2 n: 4</p>  <p>elem: 9</p>	<p>loc: 0 i: 2 n: 4</p>  <p>elem: 9</p>	<p>loc: 0 i: 2 n: 4</p>  <p>elem: 9</p>
15. $a[loc + 1] \leftarrow elem$	16. $loc \leftarrow loc - 1$	17. $loc \geq 0$ and $a[loc] > elem$?

<p>loc: 0 i: 2 n: 4</p>  <p>elem: 9</p>	<p>loc: 0 i: 3 n: 4</p>  <p>elem: 9</p>	<p>loc: 0 i: 3 n: 4</p>  <p>elem: 9</p>
<p>18. $a[loc + 1] \leftarrow elem$</p>	<p>19. $i \leftarrow i + 1$</p>	<p>20. $i < n ?$</p>
<p>loc: 0 i: 3 n: 4</p>  <p>elem: 11</p>	<p>loc: 2 i: 3 n: 4</p>  <p>elem: 11</p>	<p>loc: 2 i: 3 n: 4</p>  <p>elem: 11</p>
<p>21. $elem \leftarrow a[i]$</p>	<p>22. $loc \leftarrow i - 1$</p>	<p>23. $loc \geq 0$ and $a[loc] > elem ?$</p>

<p>loc: 2 i: 3 n: 4</p>  <p>elem: 11</p>	<p>loc: 2 i: 4 n: 4</p>  <p>elem: 11</p>	<p>loc: 2 i: 4 n: 4</p>  <p>elem: 11</p>
<p>24. $a[loc + 1] \leftarrow elem$</p>	<p>25. $i \leftarrow i + 1$</p>	<p>26. $i < n ?$</p>

§2. Применение модели машины с произвольным доступом к памяти

Анализ алгоритма заключается в предсказании объёма требуемых для его работы вычислительных ресурсов. Чаще всего нас интересует время работы алгоритма и объём используемой оперативной памяти.

Для оценки времени работы алгоритма используют гипотетический компьютер, называемый *машиной с произвольным доступом к памяти* (Random Access Machine, или RAM) и обладающий следующими свойствами:

1. каждая простая операция ($+$, $*$, $-$, $=$, проверка условия, вызов подпрограммы) выполняется ровно за один шаг;
2. циклы и подпрограммы не считаются простыми операциями: они рассматриваются как композиции операций, составляющих их тела;
3. каждое обращение к памяти выполняется ровно за один шаг.

Пример.

```
elem := A[ i ]
```

Шаги:

1. чтение i ;
2. умножение i на размер элемента массива;
3. прибавление адреса начала массива;
4. чтение $A[i]$;
5. запись в $elem$.

Пример.

```
while i < N do
```

Шаги:

1. чтение i ;
2. чтение N ;
3. операция $<$;
4. проверка условия.

Пример.

```
while (loc >= 0) and (A[loc] > elem) do
```

Шаги:

1. чтение loc;
2. операция >=;
3. проверка условия (если «ложь», то закончить);
4. чтение loc;
5. умножение loc на размер элемента массива;
6. прибавление адреса начала массива;
7. чтение A[loc];
8. чтение elem;
9. операция >;
10. проверка условия.

Пример. Сортировка вставками

```
procedure InsertionSort(A: array of integer;  
                        n: integer)  
var i, elem, loc: integer;  
begin  
    i := 1;  
    while i < n do  
        begin  
            elem := A[i];  
            loc := i - 1;  
            while (loc >= 0) and (A[loc] > elem) do  
                begin  
                    A[loc + 1] := A[loc];  
                    loc := loc - 1;  
                end;  
            A[loc + 1] := elem;  
            i := i + 1;  
        end  
    end
```

Пример. Сортировка вставками (с указанием количества шагов)

```

procedure InsertionSort(A: array of integer ;
                        n: integer)
var i , elem , loc: integer ;
begin
(* 1 *)      i := 1 ;
(* 4 *)      while i < n do
              begin
(* 5 *)          elem := A[i] ;
(* 3 *)          loc := i - 1 ;
(* 3/10 *)     while (loc >= 0) and (A[loc] > elem) do
                begin
(* 9 *)          A[loc+1] := A[loc] ;
(* 3 *)          loc := loc - 1 ;
                end ;
(* 6 *)          A[loc+1] := elem ;
(* 3 *)          i := i + 1 ;
              end
            end
end
```

Имея набор входных данных, мы можем вычислить, за сколько шагов выполнится алгоритм. Однако, для оценки качества алгоритма требуется знать, как он поведёт себя на всех наборах входных данных.

Определение. *Сложность алгоритма в наилучшем (наихудшем) случае* – это функция, задающая зависимость минимального (максимального) количества шагов машины с произвольным доступом к памяти, выполняющей данный алгоритм, от размера входных данных.

Задание. Получить выражение для сложности алгоритма сортировки вставками в наилучшем и наихудшем случае:

$$C_{min}(n) = ?$$

$$C_{max}(n) = ?$$

Решение (наилучший случай – данные отсортированы по возрастанию)

Внутренний цикл – всегда 10 шагов.

Одна итерация внешнего цикла: $4 + 5 + 3 + 10 + 6 + 3 = 31$ шаг.

Внешний цикл выполняется $n - 1$ раз, если $n > 0$, поэтому

$$C_{min}(n) = \begin{cases} 5, & n = 0; \\ 1 + 4 + 31(n - 1) = 31n - 26, & n > 0. \end{cases}$$

Решение (наихудший случай – данные отсортированы по убыванию)

Одна итерация внутреннего цикла: $10 + 9 + 3 = 22$ шага.

Внутренний цикл выполняется i раз за время $22i + 3$, поэтому одна итерация внешнего цикла: $4 + 5 + 3 + (22i + 3) + 6 + 3 = 22i + 24$ шагов.

Внешний цикл выполняется $n - 1$ раз, если $n > 0$, поэтому

$$C_{max}(n) = \begin{cases} 5, & n = 0; \\ 1 + 4 + \sum_{i=1}^{n-1} (22i + 24) = 5 + 24(n - 1) + 22 \sum_{i=1}^{n-1} i = \\ = 24n - 19 + 22 \times \frac{n(n-1)}{2} = 11n^2 + 13n - 19, & n > 0. \end{cases}$$

Проверка. $C_{min}(1) = C_{max}(1) = 5$.

§3. Асимптотическая сложность алгоритма

В этом параграфе будем рассматривать только функции, областью определения которых являются целые неотрицательные числа. При этом эти функции не могут принимать отрицательных значений.

Определение. Функция $g(n)$ является *асимптотической верхней границей* для функции $f(n)$, если можно найти положительные константы c и n_0 такие, что для любого $n \geq n_0$ справедливо $f(n) \leq c \cdot g(n)$.

Для некоторой функции $g(n)$ обозначение $O(g(n))$ (читается «о большое от g от n ») означает множество функций, для которых $g(n)$ является асимптотической верхней границей:

$$O(g(n)) = \{f(n) \mid (\exists c, n_0 > 0) : \forall n \geq n_0, f(n) \leq c \cdot g(n)\}.$$

$f(n) \in O(g(n))$ принято записывать, как $f(n) = O(g(n))$ или $f(n) \lesssim g(n)$.

Определение. Функция $g(n)$ является асимптотической нижней границей для функции $f(n)$, если можно найти положительные константы c и n_0 такие, что для любого $n \geq n_0$ справедливо $f(n) \geq c \cdot g(n)$.

Для некоторой функции $g(n)$ обозначение $\Omega(g(n))$ (читается «омега большое от g от n ») означает множество функций, для которых $g(n)$ является асимптотической нижней границей:

$$\Omega(g(n)) = \{f(n) \mid (\exists c, n_0 > 0) : \forall n \geq n_0, f(n) \geq c \cdot g(n)\}.$$

$f(n) \in \Omega(g(n))$ принято записывать, как $f(n) = \Omega(g(n))$ или $f(n) \gtrsim g(n)$.

Замечание. $f(n) \lesssim g(n) \iff g(n) \gtrsim f(n)$.

Задание. Доказать, что $31n - 26 = \Omega(n)$.

Задание. Доказать, что $11n^2 + 13n - 19 = O(n^2)$.

Если одновременно $f(n) \lesssim g(n)$ и $f(n) \gtrsim g(n)$, то говорят, что $f(n)$ является *асимптотически точной оценкой* функции $g(n)$. При этом $g(n)$ тоже является *асимптотически точной оценкой* функции $f(n)$

Для некоторой функции $g(n)$ обозначение $\Theta(g(n))$ (читается «тета большое от g от n ») означает множество функций, для которых $g(n)$ является асимптотически точной оценкой:

$$\Theta(g(n)) = \{f(n) \mid (f(n) \lesssim g(n)) \wedge (f(n) \gtrsim g(n))\}.$$

$f(n) \in \Theta(g(n))$ принято записывать, как $f(n) = \Theta(g(n))$ или $f(n) \simeq g(n)$.

Теорема (об асимптотике многочлена)

Для любого многочлена $p(n) = \sum_{i=0}^d a_i n^i$, где a_i – константы и $a_d > 0$, справедливо: $p(n) = \Theta(n^d)$.

Константа – это полином нулевой степени, поэтому константы обозначаются как $\Theta(n^0)$ или $\Theta(1)$.

Определение. Асимптотическая сложность алгоритма в наилучшем (наихудшем) случае – это асимптотическая нижняя (верхняя) граница сложности алгоритма в наилучшем (наихудшем случае).

Пример. Асимптотическая сложность алгоритма сортировки вставками: в наилучшем случае $\Omega(n)$, в наихудшем – $O(n^2)$.

В общем случае асимптотическая сложность алгоритма в наилучшем случае доказываемается значительно сложнее, чем в наихудшем случае.

Рассмотрим примеры задач, алгоритмы решения которых имеют разную асимптотическую сложность в наихудшем случае.

Задача. Дано целое число фиксированной разрядности. Необходимо определить, является ли оно степенью двойки. Задача решается за время $O(1)$ – *константная сложность*.

Задача. Даны две строки S и T , суммарная длина которых равна n . Необходимо определить, является ли строка S подстрокой строки T . Алгоритм Кнута–Морриса–Пратта решает задачу за $O(n)$ – *линейная сложность*.

Задача. Дана отсортированная последовательность из n элементов. Нужно найти в этой последовательности некоторый элемент. Задача решается методом деления пополам за $O(\lg n)$ – *логарифмическая сложность*.

Задача. Отсортировать последовательность из n элементов.

Используя алгоритм пирамидальной сортировки или алгоритм сортировки слиянием, мы можем отсортировать любую последовательность, допускающую обращение к произвольному элементу за константное время, за $O(n \lg n)$ – *квазилинейная сложность*;

используя сортировку вставками, мы можем отсортировать любую последовательность, даже не допускающую обращение к произвольному элементу за константное время, за $O(n^2)$ – *квадратичная сложность*.

Задача. Имеется n работников и n заданий. Каждого работника можно поставить на выполнение любого задания. Однако стоимость выполнения задания различна для разных пар «работник-задание». Необходимо распределить по одному заданию на каждого работника так, чтобы суммарная стоимость была минимальна.

Задача решается так называемым «венгерским алгоритмом» за $O(n^3)$ – *кубическая сложность*.

Вообще, сложность $O(n^d)$ называется *полиномиальной*.

Задача. (Задача SAT) Имеется формула длины n в логике высказываний. Такая формула состоит из булевых переменных и логических связок (И, ИЛИ, НЕ). Требуется подобрать такие значения переменных, при которых формула истинна. Задача решается алгоритмом DPLL за время $O(\exp n)$ — экспоненциальная сложность.

§4. Размещения, перестановки и сочетания

Пусть имеется конечное множество $Q = \{q_0, q_1, \dots, q_{n-1}\}$.

Определение 4.1. *Размещение* на множестве Q по m элементов – это кортеж
 $p \in Q^m$.

Если один и тот же элемент q присутствует в размещении p более чем в одном экземпляре, то говорят, что p – *размещение с повторениями*. В противном случае, p – *размещение без повторений*.

Правило произведения. Если объект a можно выбрать x способами, и если после каждого такого выбора объект b можно выбрать y способами, то выбор пары $\langle a, b \rangle$ в указанном порядке можно осуществить xy способами.

Утверждение 4.1. Количество размещений на множестве Q по m элементов с повторениями равно $\tilde{A}_n^m = n^m$, где n – размер множества Q .

▷ База индукции: $m = 1$: $\tilde{A}_n^1 = n = n^1$.

Пусть $\tilde{A}_n^{m-1} = n^{m-1}$.

К любому размещению на множестве Q по $m-1$ элементов с повторениями можно добавить любой из n элементов.

Тогда по правилу произведения $\tilde{A}_n^m = \tilde{A}_n^{m-1} \cdot n = n^m$. ◁

Для записи алгоритма, вычисляющего размещения, нам понадобится, чтобы на множестве Q было определено отношение строгого порядка \prec , то есть чтобы

$$q_0 \prec q_1 \prec \dots \prec q_{n-1}.$$

Отношение строгого порядка может быть выбрано совершенно произвольно — нам всего лишь нужен способ всегда одинаково перечислять элементы любого подмножества множества Q .

Если $R \subseteq Q$, то цикл, перечисляющий элементы R по порядку \prec , мы будем записывать на псевдокоде как

```
1 for each  $q \in R$ 
2     что-то сделать с  $q$ 
```


Алгоритм вычисления всех размещений на множестве Q по m элементов с повторениями:

```
1 PermutRep(in Q, in m)
2     PermutRep_rec(Q, m, ⟨⟩)

4 PermutRep_rec(in Q, in m, in p)
5     if m = 0:
6         что-то сделать с p
7     else:
8         for each q ∈ Q:
9             PermutRep_rec(Q, m - 1, ⟨p, q⟩)
```

Из утверждения 4.1 следует, что алгоритм выполняется за время $\Theta(n^m)$.

Утверждение 4.2. Количество размещений на множестве Q по m элементов без повторений равно

$$A_n^m = \frac{n!}{(n-m)!}, \text{ где } n - \text{размер множества } Q.$$

▷ База индукции: $m = 1$: $A_n^1 = n = \frac{n!}{(n-1)!}$.

Пусть $A_n^{m-1} = \frac{n!}{(n-(m-1))!}$.

К любому размещению на множестве Q по $m-1$ элементов без повторений можно добавить любой из не входящих в это размещение $n - (m-1)$ элементов.

Тогда по правилу произведения $A_n^m = A_n^{m-1} \cdot (n - (m-1)) = \frac{n!}{(n-m)!}$. ◁

Алгоритм вычисления всех размещений на множестве Q по m элементов без повторений:

```
1 Permut(in  $Q$ , in  $m$ )
2     Permut_rec( $Q$ ,  $m$ ,  $\langle \rangle$ )

4 Permut_rec(in  $Q$ , in  $m$ , in  $p$ )
5     if  $m = 0$ :
6         что-то сделать с  $p$ 
7     else:
8         for each  $q \in Q$ :
9             Permut_rec( $Q \setminus \{q\}$ ,  $m - 1$ ,  $\langle p, q \rangle$ )
```

Из утверждения 4.2 следует, что алгоритм выполняется за время $\Theta\left(\frac{n!}{(n-m)!}\right)$.

Определение 4.2. *Перестановка* множества Q – это размещение на множестве Q по n элементов без повторений, где n – размер множества Q .

Количество перестановок $P_n = A_n^n = \frac{n!}{(n-n)!} = n!$

Можно дать другое определение перестановки:

Определение 4.2.1. *Перестановка* множества Q – это биекция множества Q на себя.

Перестановку $\langle q_x, q_y, \dots, q_z \rangle$ в рамках определения 4.2.1 принято записывать в виде $\mathcal{P} = \begin{pmatrix} q_0 & q_1 & \dots & q_{n-1} \\ q_x & q_y & \dots & q_z \end{pmatrix}$.

Здесь q_0 отображается в q_x , q_1 – в q_y , и т.д.

Определение 4.3. Сочетание на множестве Q по m элементов – это m -элементное подмножество множества Q .

Утверждение 4.3. Количество сочетаний на множестве Q по m элементов равно

$$C_n^m = \frac{n!}{(n-m)! \cdot m!}, \text{ где } n - \text{размер множества } Q.$$

► Очевидно, что из элементов одного сочетания по m элементов можно составить P_m различных перестановок по m элементов.

Поэтому $A_n^m = C_n^m \cdot P_m$, откуда $C_n^m = \frac{A_n^m}{P_m} = \frac{n!}{(n-m)! \cdot m!}. \triangleleft$

Алгоритм вычисления всех сочетаний на множестве Q по m элементов:

```
1 Comb(in Q, in m)
2     Comb_rec(Q, m, ∅)

4 Comb_rec(in Q, in m, in c)
5     if m = 0:
6         что-то сделать с c
7     else:
8          $Q' \leftarrow Q$ 
9         for each  $q \in Q$ :
10            if  $|Q'| < m$ :
11                break
12             $Q' \leftarrow Q' \setminus \{q\}$ 
13            Comb_rec( $Q'$ ,  $m - 1$ ,  $c \cup \{q\}$ )
```

Из утверждения 4.3 следует, что алгоритм выполняется за время

$$\Theta\left(\frac{n!}{(n-m)! \cdot m!}\right).$$

§5. Постановка задачи сортировки

Обозначения. Пусть дана последовательность натуральных чисел $\{a_i\}_0^{n-1}$, тогда:

- запись $a[i]$ обозначает обращение к i -тому элементу последовательности;
- запись $a[i : j]$ обозначает подпоследовательность, начинающуюся с i -того элемента, и заканчивающуюся j -тым элементом;
- операция обмена $a[i] \leftrightarrow a[j]$ означает, что i -тый и j -тый элементы меняются местами.

Запись Null_n обозначает последовательность нулей длины n .

Пример. Пусть даны последовательности $A \leftarrow \langle 5, 4, 6, 7, 2 \rangle$ и $B \leftarrow \langle 8, 9 \rangle$.

Тогда присваивание

$$A[1 : 2] \leftarrow B$$

превратит последовательность A в $\langle 5, 8, 9, 7, 2 \rangle$.

Если затем выполнить обмен $A[1] \leftrightarrow A[3]$, то мы получим последовательность $\langle 5, 7, 9, 8, 2 \rangle$.

Пусть множество $\mathbb{N}_n = \{0, 1, \dots, n - 1\}$.

В алгоритмах сортировки мы будем работать с перестановками множества \mathbb{N}_n , которые будем понимать как последовательности (кортежи), составленные из n различных натуральных чисел, принадлежащих множеству \mathbb{N}_n .

Пример. $\mathcal{P} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 0 & 3 \end{pmatrix}$ – перестановка множества \mathbb{N}_4 .
 $\mathcal{P}[0] = 1, \mathcal{P}[1] = 2, \mathcal{P}[2] = 0, \mathcal{P}[3] = 3$.

В алгоритмах мы будем обращаться с перестановками как с обычными последовательностями.

Обозначение. Операция инверсии перестановки \mathcal{P} записывается как $\neg\mathcal{P}$ и означает «переворот» перестановки \mathcal{P} :

$$\left(\forall i = \overline{0, n-1}\right) : \neg\mathcal{P} [\mathcal{P} [i]] = i.$$

Пример. Пусть $\mathcal{P} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 0 & 3 \end{pmatrix}$, тогда $\neg\mathcal{P} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 2 & 0 & 1 & 3 \end{pmatrix}$.

Обозначение. Мы будем использовать обозначение Id_n для перестановки множества \mathbb{N}_n такой, что

$$\left(\forall i = \overline{0, n-1}\right) : \text{Id}_n [i] = i.$$

Очевидно, что $\text{Id}_n = \neg\text{Id}_n$.

Пусть имеется совокупность из n значений, которые нужно упорядочить: r_0, r_1, \dots, r_{n-1} .

Мы будем называть эти значения *записями*.

Каждая запись r_i имеет *ключ* k_i , который управляет процессом сортировки. Ключ может быть составной частью записи, а может вычисляться на основе записи.

На множестве ключей вводится отношение строгого порядка \triangleleft (читается «*предшествует*»), которое:

1. асимметрично: $x \triangleleft y \Rightarrow y \not\triangleleft x$;
2. транзитивно: $(x \triangleleft y) \wedge (y \triangleleft z) \Rightarrow x \triangleleft z$.

(Из асимметричности следует антирефлексивность: $x \not\triangleleft x$.)

При этом из отношения \triangleleft можно построить отношение нестрогого порядка \trianglelefteq , если добавить рефлексивность: $x \trianglelefteq y \Leftrightarrow (x \triangleleft y) \vee (x = y)$.

Задача сортировки последовательности записей $\langle r_0, r_1, \dots, r_{n-1} \rangle$, имеющих ключи k_0, k_1, \dots, k_{n-1} , заключается в поиске перестановки

$\mathcal{P} = \begin{pmatrix} 0 & 1 & \dots & n-1 \\ x_0 & x_1 & \dots & x_{n-1} \end{pmatrix}$ такой, что $k_{x_0} \trianglelefteq k_{x_1} \trianglelefteq \dots \trianglelefteq k_{x_{n-1}}$.

Пример. Дана совокупность матриц:

$$r_0 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, r_1 = \begin{bmatrix} 3 & 1 \\ 0 & 5 \end{bmatrix}, r_2 = \begin{bmatrix} 0 & 1 \\ 3 & 7 \end{bmatrix} \text{ и } r_3 = \begin{bmatrix} 3 & 0 \\ 0 & 5 \end{bmatrix}.$$

Ключи – определители матриц:

$$k_0 = -2, k_1 = 15, k_2 = -3, k_3 = 15.$$

Отношение порядка на множестве ключей – порядок на множестве целых чисел.

Тогда решение задачи сортировки: $\mathcal{P} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 2 & 0 & 1 & 3 \end{pmatrix}$.

Замечание. Классическая постановка задачи сортировки подразумевает, что алгоритм сортировки ничего не знает ни о природе записей, ни о природе ключей. Алгоритм может лишь сравнивать между собой ключи, соответствующие записям.

Сортировка называется *устойчивой*, если она не меняет взаимного расположения записей, имеющих равные ключи, т.е., другими словами, для любых двух записей r_i и r_j таких, что $k_i = k_j$, справедливо, что $i < j \Rightarrow (\neg \mathcal{P})[i] < (\neg \mathcal{P})[j]$.

Пример. Пусть последовательность

$\langle 103, 12, 93, 35, 14 \rangle$

сортируется по возрастанию младшей цифры числа. Тогда устойчивый алгоритм сортировки всегда даст последовательность

$\langle 12, 103, 93, 14, 35 \rangle$,

а неустойчивый может поменять местами 103 и 93.

Пусть записи r_0, r_1, \dots, r_{n-1} имеют ключи k_0, k_1, \dots, k_{n-1} , и на множестве ключей введено отношение строгого порядка \triangleleft .

Введём отношение $\prec \subseteq \mathbb{N}_n \times \mathbb{N}_n$ (читается «меньше») следующим образом:
 $i \prec j \Leftrightarrow k_i \triangleleft k_j$.

Тогда алгоритм сортировки вставками можно переписать как

```
1 InsertSort(in  $\prec$ , in  $n$ , out  $\mathcal{P}$ )
2    $\mathcal{P} \leftarrow \mathbf{Id}_n$ 
3    $i \leftarrow 1$ 
4   while  $i < n$ :
5        $loc \leftarrow i - 1$ 
6       while  $loc \geq 0$  and  $\mathcal{P}[loc + 1] \prec \mathcal{P}[loc]$ :
7            $\mathcal{P}[loc + 1] \leftrightarrow \mathcal{P}[loc]$ 
8            $loc \leftarrow loc - 1$ 
9    $i \leftarrow i + 1$ 
```

Этот алгоритм очевидно устойчив. Дело в том, что операция обмена в строке 7 никогда не будет применена к равным записям.

§6. Пузырьковая сортировка

Пусть дана последовательность натуральных чисел $\langle 5, 4, 7, 2, 3, 6 \rangle$.

Пройдём по ней слева направо, сравнивая соседние элементы и меняя их местами, если они расположены не в порядке возрастания:

1. $\langle 4, 5, 7, 2, 3, 6 \rangle$;
2. $\langle 4, 5, 7, 2, 3, 6 \rangle$;
3. $\langle 4, 5, 2, 7, 3, 6 \rangle$;
4. $\langle 4, 5, 2, 3, 7, 6 \rangle$;
5. $\langle 4, 5, 2, 3, 6, 7 \rangle$.

В результате максимальное число 7 переместилось в крайнее правое положение, т.е. туда, где оно должно находиться в отсортированной последовательности. Кроме того, поменялись местами числа 5 и 4, что тоже сделало последовательность более упорядоченной.

Повторим проход по последовательности

$\langle 4, 5, 2, 3, 6, 7 \rangle$

ещё раз, но теперь не будем рассматривать число 7, так как его положение всё равно не изменится:

1. $\langle 4, 5, 2, 3, 6, 7 \rangle$;
2. $\langle 4, 2, 5, 3, 6, 7 \rangle$;
3. $\langle 4, 2, 3, 5, 6, 7 \rangle$;
4. $\langle 4, 2, 3, 5, 6, 7 \rangle$.

Мы убеждаемся, что число 6 занимает своё место в отсортированной последовательности.

Обратим также внимание на то, что последний обмен был совершён нами на третьем шаге (поменялись 5 и 3). Тем самым мы можем сделать вывод, что число 5 тоже находится на правильном месте, и в дальнейшем мы его уже можем не рассматривать.

Следующий проход по последовательности

$\langle 4, 2, 3, 5, 6, 7 \rangle$

не затрагивает числа 5, 6 и 7:

1. $\langle 2, 4, 3, 5, 6, 7 \rangle$;

2. $\langle 2, 3, 4, 5, 6, 7 \rangle$.

Здесь мы выясняем, что число 4 находится там, где оно должно располагаться в отсортированной последовательности.

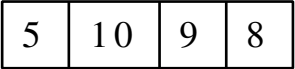
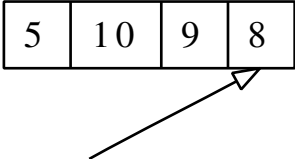
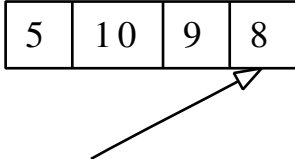
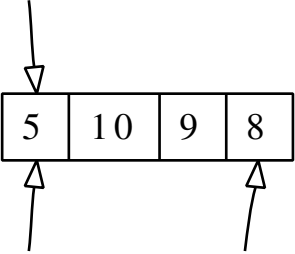
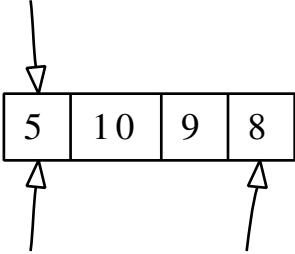
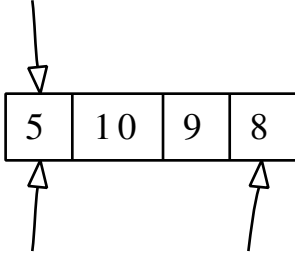
Для верности выполним ещё один проход, чтобы убедиться, что числа 2 и 3 расположены правильно.

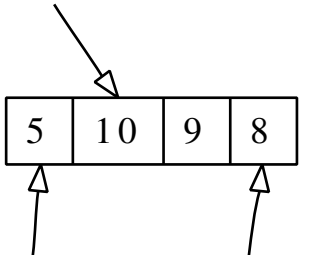
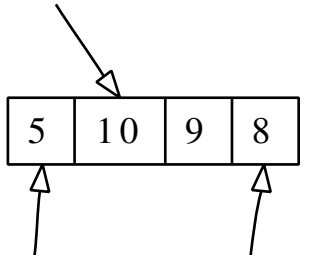
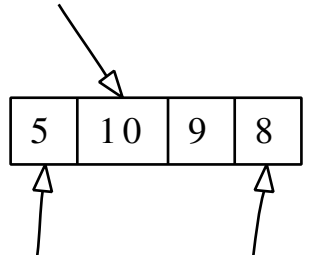
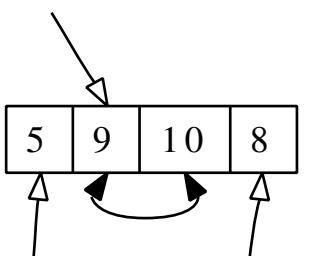
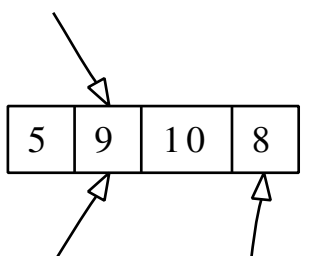
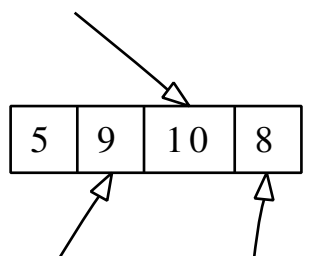
Алгоритм сортировки пузырьком:

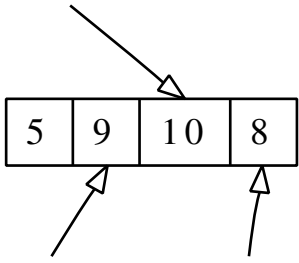
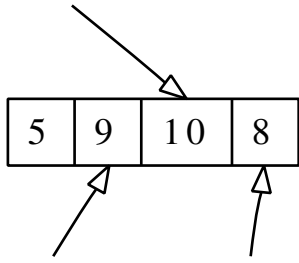
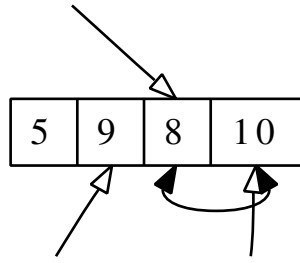
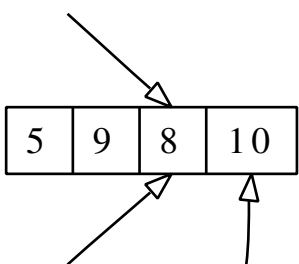
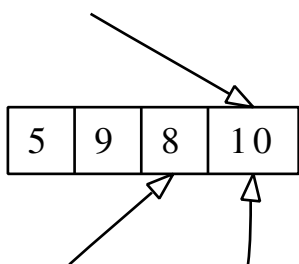
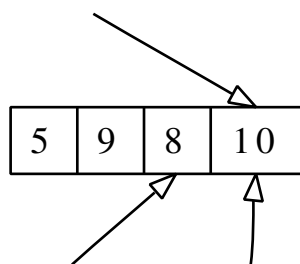
```
1 BubbleSort(in  $\prec$ , in  $n$ , out  $\mathcal{P}$ )
2    $\mathcal{P} \leftarrow \mathbf{Id}_n$ 
3    $t \leftarrow n - 1$ 
4   while  $t > 0$ :
5      $bound \leftarrow t$ 
6      $t \leftarrow 0$ 
7      $i \leftarrow 0$ 
8     while  $i < bound$ :
9       if  $\mathcal{P}[i + 1] \prec \mathcal{P}[i]$ :
10          $\mathcal{P}[i + 1] \leftrightarrow \mathcal{P}[i]$ 
11          $t \leftarrow i$ 
12      $i \leftarrow i + 1$ 
```

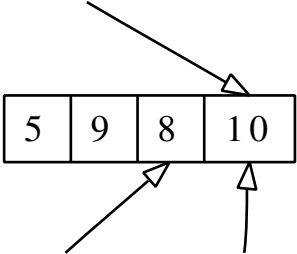
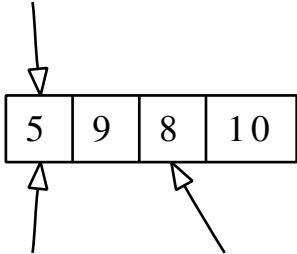
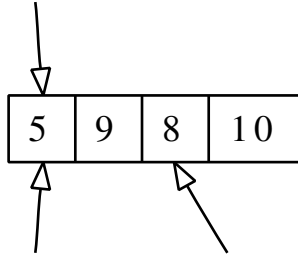
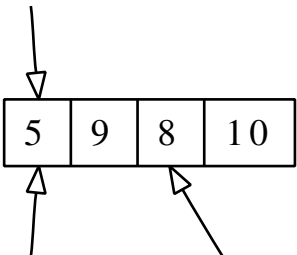
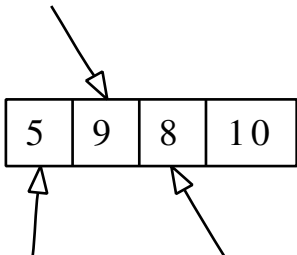
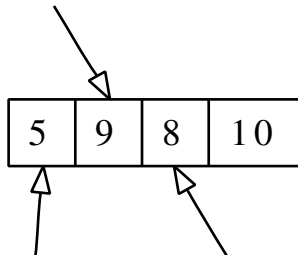
Сортируемая последовательность, как и в случае сортировки вставками, делится на две части: отсортированную и неотсортированную.

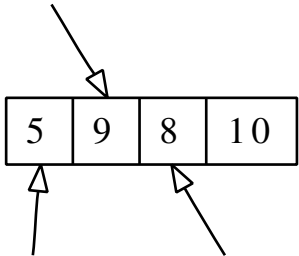
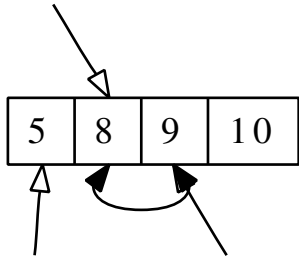
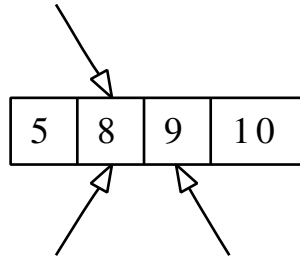
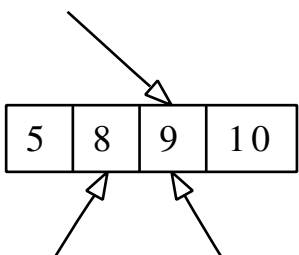
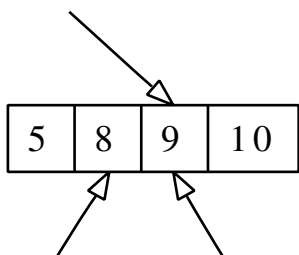
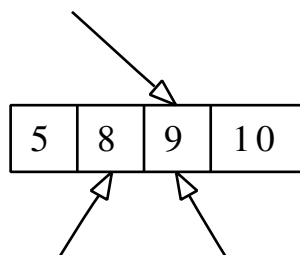
Переменная t на момент проверки условия внешнего цикла всегда содержит индекс последней записи неотсортированной части, то есть на каждой итерации цикла неотсортированная последовательность содержит записи с индексами от 0 до t .

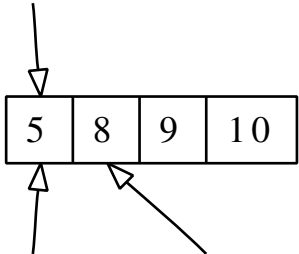
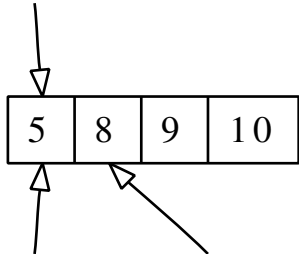
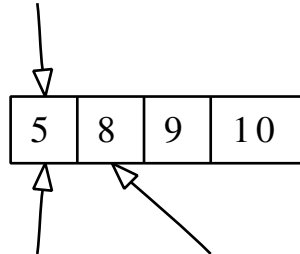
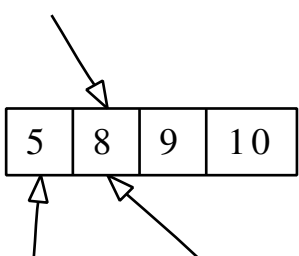
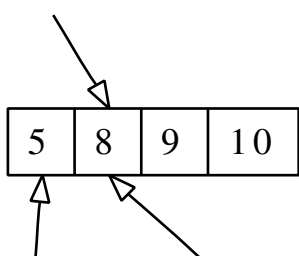
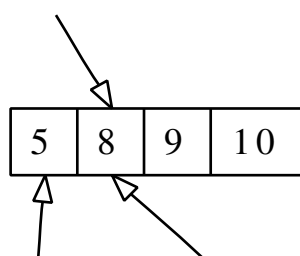
<p>$i: ?$ $n: 4$</p>  <p>$t: ?$ $\text{bound}: ?$</p>	<p>$i: ?$ $n: 4$</p>  <p>$t: 3$ $\text{bound}: ?$</p>	<p>$i: ?$ $n: 4$</p>  <p>$t: 3$ $\text{bound}: ?$</p>
	<p>1. $t \leftarrow n - 1$</p>	<p>2. $t > 0 ?$</p>
<p>$i: 0$ $n: 4$</p>  <p>$t: 0$ $\text{bound}: 3$</p>	<p>$i: 0$ $n: 4$</p>  <p>$t: 0$ $\text{bound}: 3$</p>	<p>$i: 0$ $n: 4$</p>  <p>$t: 0$ $\text{bound}: 3$</p>
<p>3. $\text{bound} \leftarrow t,$ $t \leftarrow 0,$ $i \leftarrow 0$</p>	<p>4. $i < \text{bound} ?$</p>	<p>5. $\mathcal{P}[i + 1] \prec \mathcal{P}[i] ?$</p>

<p>i: 1 n: 4</p>  <p>t: 0 bound: 3</p>	<p>i: 1 n: 4</p>  <p>t: 0 bound: 3</p>	<p>i: 1 n: 4</p>  <p>t: 0 bound: 3</p>
<p>6. $i \leftarrow i + 1$</p>	<p>7. $i < bound ?$</p>	<p>8. $\mathcal{P}[i + 1] \prec \mathcal{P}[i] ?$</p>
<p>i: 1 n: 4</p>  <p>t: 0 bound: 3</p>	<p>i: 1 n: 4</p>  <p>t: 1 bound: 3</p>	<p>i: 2 n: 4</p>  <p>t: 1 bound: 3</p>
<p>9. $\mathcal{P}[i + 1] \leftrightarrow \mathcal{P}[i]$</p>	<p>10. $t \leftarrow i$</p>	<p>11. $i \leftarrow i + 1$</p>

<p>i: 2 n: 4</p>  <p>t: 1 bound: 3</p>	<p>i: 2 n: 4</p>  <p>t: 1 bound: 3</p>	<p>i: 2 n: 4</p>  <p>t: 1 bound: 3</p>
<p>12. $i < bound$?</p>	<p>13. $\mathcal{P}[i + 1] \prec \mathcal{P}[i]$?</p>	<p>14. $\mathcal{P}[i + 1] \leftrightarrow \mathcal{P}[i]$</p>
<p>i: 2 n: 4</p>  <p>t: 2 bound: 3</p>	<p>i: 3 n: 4</p>  <p>t: 2 bound: 3</p>	<p>i: 3 n: 4</p>  <p>t: 2 bound: 3</p>
<p>15. $t \leftarrow i$</p>	<p>16. $i \leftarrow i + 1$</p>	<p>17. $i < bound$?</p>

<p>i: 3 n: 4</p>  <p>t: 2 bound: 3</p>	<p>i: 0 n: 4</p>  <p>t: 0 bound: 2</p>	<p>i: 0 n: 4</p>  <p>t: 0 bound: 2</p>
<p>18. $t > 0$?</p>	<p>19. $bound \leftarrow t,$ $t \leftarrow 0,$ $i \leftarrow 0$</p>	<p>20. $i < bound$?</p>
<p>i: 0 n: 4</p>  <p>t: 0 bound: 2</p>	<p>i: 1 n: 4</p>  <p>t: 0 bound: 2</p>	<p>i: 1 n: 4</p>  <p>t: 0 bound: 2</p>
<p>21. $\mathcal{P}[i + 1] \prec \mathcal{P}[i]$?</p>	<p>22. $i \leftarrow i + 1$</p>	<p>23. $i < bound$?</p>

<p>i: 1 n: 4</p>  <p>t: 0 bound: 2</p>	<p>i: 1 n: 4</p>  <p>t: 0 bound: 2</p>	<p>i: 1 n: 4</p>  <p>t: 1 bound: 2</p>
<p>24. $\mathcal{P}[i + 1] \prec \mathcal{P}[i]$?</p>	<p>25. $\mathcal{P}[i + 1] \leftrightarrow \mathcal{P}[i]$</p>	<p>26. $t \leftarrow i$</p>
<p>i: 2 n: 4</p>  <p>t: 1 bound: 2</p>	<p>i: 2 n: 4</p>  <p>t: 1 bound: 2</p>	<p>i: 2 n: 4</p>  <p>t: 1 bound: 2</p>
<p>27. $i \leftarrow i + 1$</p>	<p>28. $i < bound$?</p>	<p>29. $t > 0$?</p>

<p>i: 0 n: 4</p>  <p>t: 0 bound: 1</p>	<p>i: 0 n: 4</p>  <p>t: 0 bound: 1</p>	<p>i: 0 n: 4</p>  <p>t: 0 bound: 1</p>
<p>30. $bound \leftarrow t,$ $t \leftarrow 0,$ $i \leftarrow 0$</p>	<p>31. $i < bound ?$</p>	<p>32. $\mathcal{P}[i + 1] \prec \mathcal{P}[i] ?$</p>
<p>i: 1 n: 4</p>  <p>t: 0 bound: 1</p>	<p>i: 1 n: 4</p>  <p>t: 0 bound: 1</p>	<p>i: 1 n: 4</p>  <p>t: 0 bound: 1</p>
<p>33. $i \leftarrow i + 1$</p>	<p>34. $i < bound ?$</p>	<p>35. $t > 0 ?$</p>

Наилучший случай для алгоритма сортировки пузырьком – это когда данные уже отсортированы. Тогда внешний цикл имеет одну итерацию, а вложенный цикл – $(n - 1)$ итераций. Тем самым в наилучшем случае асимптотическая сложность алгоритма составляет $\Omega(n)$.

В наихудшем случае данные отсортированы в обратном порядке («по убыванию»). Тогда внешний цикл работает n раз, а вложенный цикл – от $(n-1)$ до 1 раза. Поэтому в наихудшем случае асимптотическая сложность алгоритма составляет $O(n^2)$.

Сортировка пузырьком устойчива по тем же соображениям, что и сортировка вставками.

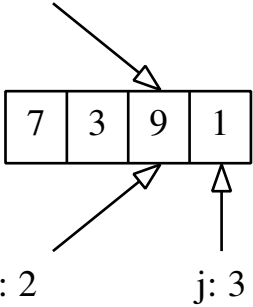
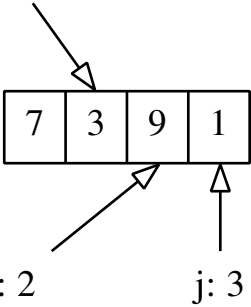
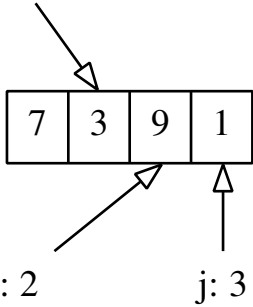
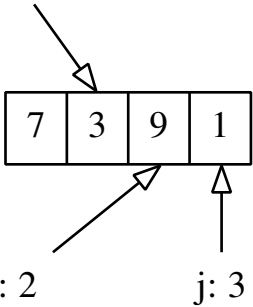
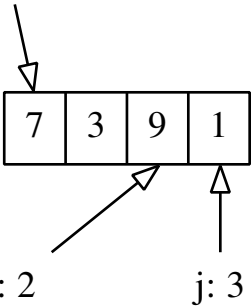
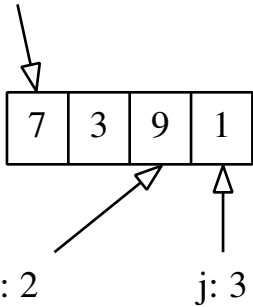
§7. Сортировка прямым выбором

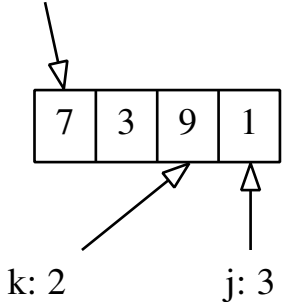
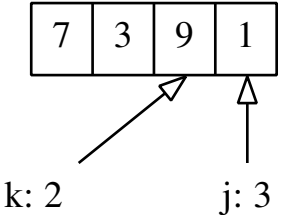
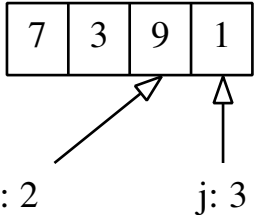
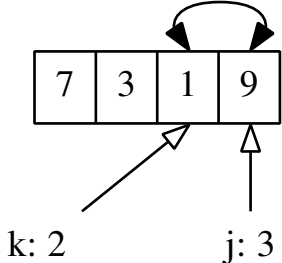
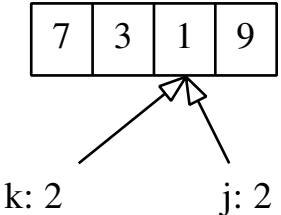
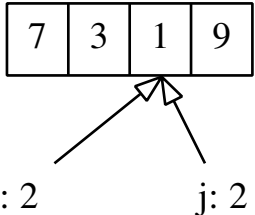
Основная идея алгоритма сортировки прямым выбором:

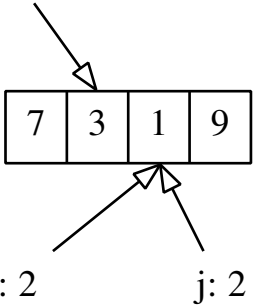
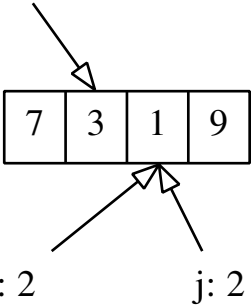
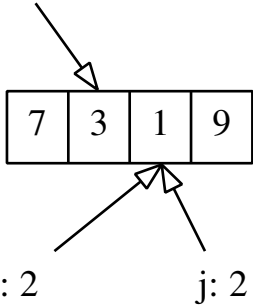
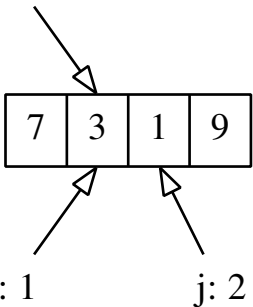
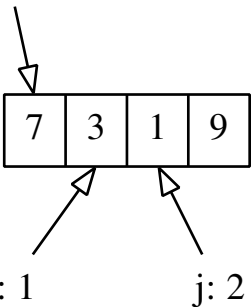
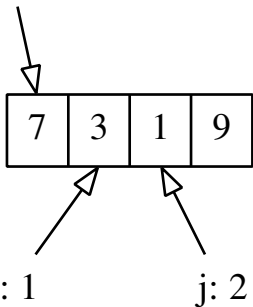
1. находим максимальную запись в сортируемой последовательности;
2. меняем её местами с последней записью, тем самым максимальная запись занимает подобающее её место в последовательности;
3. повторяем вышеприведённые шаги для оставшейся последовательности (лишённой последней записи).

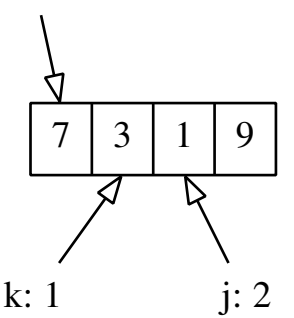
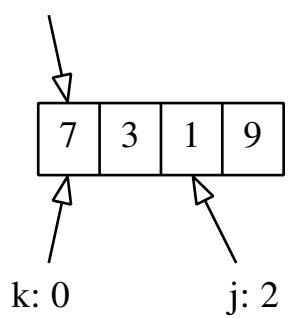
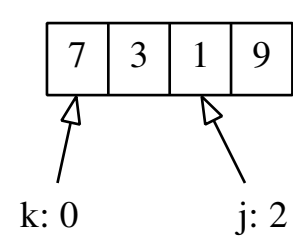
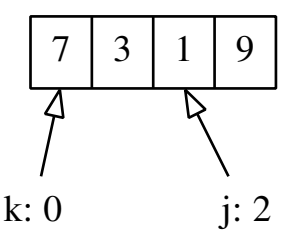
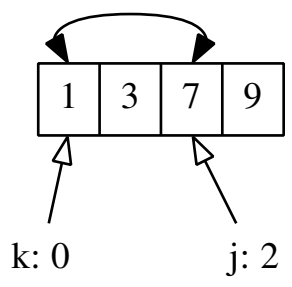
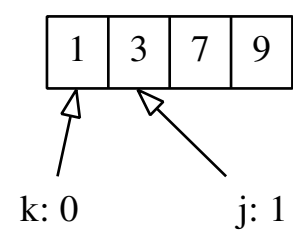
```
1 SelectSort(in  $\prec$ , in  $n$ , out  $\mathcal{P}$ )
2      $\mathcal{P} \leftarrow \mathbf{Id}_n$ 
3      $j \leftarrow n - 1$ 
4     while  $j > 0$ :
5          $k \leftarrow j$ 
6          $i \leftarrow j - 1$ 
7         while  $i \geq 0$ :
8             if  $\mathcal{P}[k] \prec \mathcal{P}[i]$ :
9                  $k \leftarrow i$ 
10             $i \leftarrow i - 1$ 
11         $\mathcal{P}[j] \leftrightarrow \mathcal{P}[k]$ 
12     $j \leftarrow j - 1$ 
```

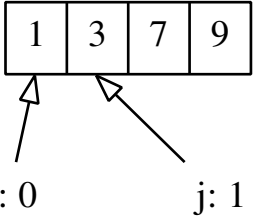
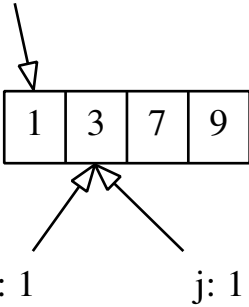
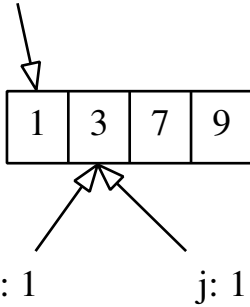
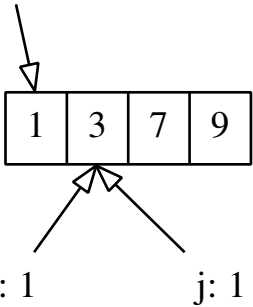
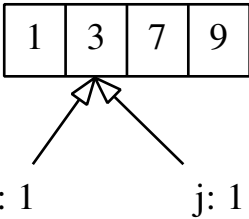
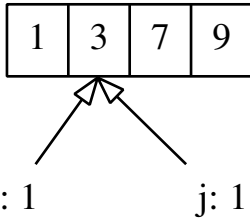
<p>i: ? n: 4</p> <div> <div>7</div><div>3</div><div>9</div><div>1</div> </div> <p>k: ? j: ?</p>	<p>i: ? n: 4</p> <div> <div>7</div><div>3</div><div>9</div><div>1</div> <div>↑</div> </div> <p>k: ? j: 3</p>	<p>i: ? n: 4</p> <div> <div>7</div><div>3</div><div>9</div><div>1</div> <div>↑</div> </div> <p>k: ? j: 3</p>
	<p>1. $j \leftarrow n - 1$</p>	<p>2. $j > 0$?</p>
<p>i: 2 n: 4</p> <div> <div>7</div><div>3</div><div>9</div><div>1</div> <div>↑</div><div>↑</div> </div> <p>k: 3 j: 3</p>	<p>i: 2 n: 4</p> <div> <div>7</div><div>3</div><div>9</div><div>1</div> <div>↑</div><div>↑</div> </div> <p>k: 3 j: 3</p>	<p>i: 2 n: 4</p> <div> <div>7</div><div>3</div><div>9</div><div>1</div> <div>↑</div><div>↑</div> </div> <p>k: 3 j: 3</p>
<p>3. $k \leftarrow j, i \leftarrow j - 1$</p>	<p>4. $i \geq 0$?</p>	<p>5. $\mathcal{P}[k] \prec \mathcal{P}[i]$?</p>

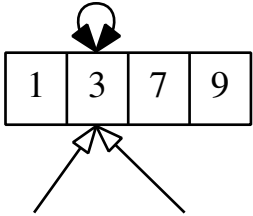
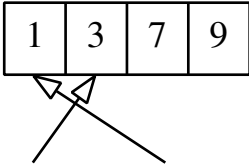
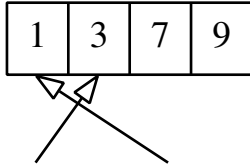
<p>i: 2 n: 4</p>  <p>k: 2 j: 3</p>	<p>i: 1 n: 4</p>  <p>k: 2 j: 3</p>	<p>i: 1 n: 4</p>  <p>k: 2 j: 3</p>
6. $k \leftarrow i$	7. $i \leftarrow i - 1$	8. $i \geq 0$?
<p>i: 1 n: 4</p>  <p>k: 2 j: 3</p>	<p>i: 0 n: 4</p>  <p>k: 2 j: 3</p>	<p>i: 0 n: 4</p>  <p>k: 2 j: 3</p>
9. $\mathcal{P}[k] < \mathcal{P}[i]$?	10. $i \leftarrow i - 1$	11. $i \geq 0$?

<p>i: 0 n: 4</p> 	<p>i: -1 n: 4</p> 	<p>i: -1 n: 4</p> 
12. $\mathcal{P}[k] < \mathcal{P}[i]$?	13. $i \leftarrow i - 1$	14. $i \geq 0$?
<p>i: -1 n: 4</p> 	<p>i: -1 n: 4</p> 	<p>i: -1 n: 4</p> 
15. $\mathcal{P}[j] \leftrightarrow \mathcal{P}[k]$	16. $j \leftarrow j - 1$	17. $j > 0$?

<p>i: 1 n: 4</p>  <p>k: 2 j: 2</p>	<p>i: 1 n: 4</p>  <p>k: 2 j: 2</p>	<p>i: 1 n: 4</p>  <p>k: 2 j: 2</p>
18. $k \leftarrow j, i \leftarrow j - 1$	19. $i \geq 0 ?$	20. $\mathcal{P}[k] \prec \mathcal{P}[i] ?$
<p>i: 1 n: 4</p>  <p>k: 1 j: 2</p>	<p>i: 0 n: 4</p>  <p>k: 1 j: 2</p>	<p>i: 0 n: 4</p>  <p>k: 1 j: 2</p>
21. $k \leftarrow i$	22. $i \leftarrow i - 1$	23. $i \geq 0 ?$

<p>i: 0 n: 4</p>  <p>k: 1 j: 2</p>	<p>i: 0 n: 4</p>  <p>k: 0 j: 2</p>	<p>i: -1 n: 4</p>  <p>k: 0 j: 2</p>
24. $\mathcal{P}[k] \prec \mathcal{P}[i]$?	25. $k \leftarrow i$	26. $i \leftarrow i - 1$
<p>i: -1 n: 4</p>  <p>k: 0 j: 2</p>	<p>i: -1 n: 4</p>  <p>k: 0 j: 2</p>	<p>i: -1 n: 4</p>  <p>k: 0 j: 1</p>
27. $i \geq 0$?	28. $\mathcal{P}[j] \leftrightarrow \mathcal{P}[k]$	29. $j \leftarrow j - 1$

<p>i: -1 n: 4</p> 	<p>i: 0 n: 4</p> 	<p>i: 0 n: 4</p> 
<p>30. $j > 0$?</p>	<p>31. $k \leftarrow j, i \leftarrow j - 1$</p>	<p>32. $i \geq 0$?</p>
<p>i: 0 n: 4</p> 	<p>i: -1 n: 4</p> 	<p>i: -1 n: 4</p> 
<p>33. $\mathcal{P}[k] < \mathcal{P}[i]$?</p>	<p>34. $i \leftarrow i - 1$</p>	<p>35. $i \geq 0$?</p>

<p>i: -1 n: 4</p>  <p>k: 1 j: 1</p>	<p>i: -1 n: 4</p>  <p>k: 1 j: 0</p>	<p>i: -1 n: 4</p>  <p>k: 1 j: 0</p>
36. $\mathcal{P}[j] \leftrightarrow \mathcal{P}[k]$	37. $j \leftarrow j - 1$	38. $j > 0 ?$

Время работы алгоритм сортировки прямым выбором практически не зависит от конкретных значений в сортируемой последовательности. Асимптотическая сложность алгоритма – $\Theta(n^2)$.

Алгоритм неустойчив.

§8. Сортировка подсчётом сравнений

Идея алгоритма сортировки подсчётом сравнений: i -тый ключ в окончательно отсортированной последовательности превышает ровно $i - 1$ остальных ключей.

Нужно сравнить попарно все ключи и для каждого отдельного ключа подсчитать, сколько ключей меньше него.

```

1 CountSort(in  $\prec$ , in  $n$ , out  $\mathcal{P}$ )
2    $count \leftarrow \mathbf{Null}_n$ 
3    $j \leftarrow 0$ 
4   while  $j < n - 1$ :
5        $i \leftarrow j + 1$ 
6       while  $i < n$ :
7           if  $i \prec j$ :
8                $count[j] \leftarrow count[j] + 1$ 
9           else :
10               $count[i] \leftarrow count[i] + 1$ 
11               $i \leftarrow i + 1$ 
12           $j \leftarrow j + 1$ 
13    $\mathcal{P} \leftarrow \neg count$ 

```

§9. Пирамидальная сортировка

Пусть дана последовательность натуральных чисел $\{a_i\}_0^{n-1}$ и задано отношение строгого порядка $\prec \subseteq \mathbb{N}_n \times \mathbb{N}_n$.

Говорят, что элемент $a[i]$ последовательности является *корнем пирамиды*, если выполняются два условия:

1. $2i + 1 < n \Rightarrow \neg(a[i] \prec a[2i + 1]) \wedge (a[2i + 1] \text{ — корень пирамиды});$
2. $2i + 2 < n \Rightarrow \neg(a[i] \prec a[2i + 2]) \wedge (a[2i + 2] \text{ — корень пирамиды}).$

Пример. ($a[2]$ — корень пирамиды, если \prec совпадает с порядком на \mathbb{N})
 $a = \langle 1, 5, \underline{7}, 2, 3, \underline{6}, \underline{7}, 9, 7, 4, 8, \underline{4}, \underline{5}, \underline{6} \rangle$.

Пирамида с корнем $a[i]$ — это подпоследовательность, в которую входит элемент $a[i]$, пирамида с корнем $a[2i + 1]$ (если $2i + 1 < n$) и пирамида с корнем $a[2i + 2]$ (если $2i + 2 < n$).

Для пирамиды с корнем $a[i]$ элементы $a[2i + 1]$ и $a[2i + 2]$ являются корнями левой и правой дочерних подпирамид, соответственно (если эти элементы существуют).

Пример. (представление пирамиды в виде дерева)

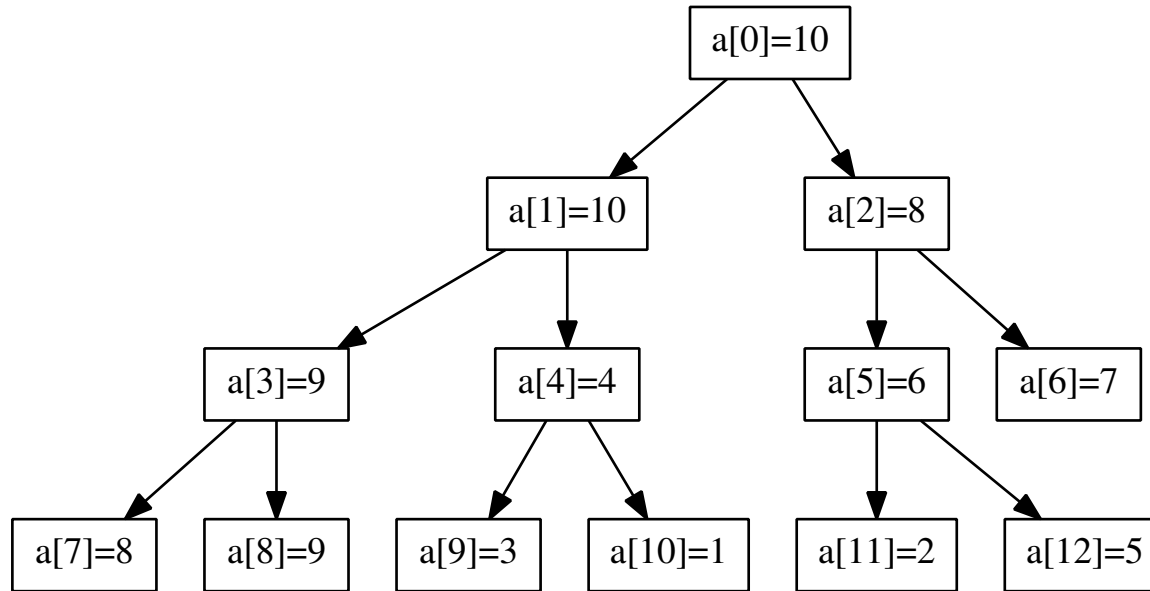
$a = \langle 10, 10, 8, 9, 4, 6, 7, 8, 9, 3, 1, 2, 5 \rangle$

$h = 3$

$h = 2$

$h = 1$

$h = 0$



Высота пирамиды, состоящей из n элементов: $\lfloor \log_2 n \rfloor$.

Количество «нелистовых» элементов пирамиды: $\lfloor n/2 \rfloor$.

На любом «этаже» высоты h находится не более $\lceil n/2^{h+1} \rceil$ элементов.

Здесь $\lfloor x \rfloor$ и $\lceil x \rceil$ обозначают округление вниз и вверх до ближайшего целого числа.

Алгоритм `Hearify` переупорядочивает элементы последовательности таким образом, чтобы её i -тый элемент являлся корнем пирамиды. Алгоритм работает при условии, что элементы $a[2i + 1]$ и $a[2i + 2]$ уже являются корнями пирамид (если эти элементы существуют).

```

1 Hearify(in  $\prec$ , in  $i$ , in  $n$ , in/out  $\mathcal{P}$ )
2     loop :
3          $l \leftarrow 2i + 1$ 
4          $r \leftarrow l + 1$ 
5          $j \leftarrow i$ 
6         if  $l < n$  and  $\mathcal{P}[i] \prec \mathcal{P}[l]$  :
7              $i \leftarrow l$ 
8         if  $r < n$  and  $\mathcal{P}[i] \prec \mathcal{P}[r]$  :
9              $i \leftarrow r$ 
10        if  $i = j$  :
11            break
12         $\mathcal{P}[i] \leftrightarrow \mathcal{P}[j]$ 

```

На каждой итерации цикла алгоритм спускается на «этаж» ниже. В худшем случае цикл завершается, когда мы оказываемся на нижнем «этаже». Отсюда сложность алгоритма `Hearify`: $O(h)$, где h — высота пирамиды с корнем в $a[i]$.

Алгоритм BuildHeap проходит по всем элементам, которые могут являться «нелистовыми» элементами пирамиды, снизу вверх и вызывает для них Heapify.

```

1 BuildHeap (in  $\prec$ , in  $n$ , in/out  $\mathcal{P}$ )
2      $i \leftarrow \lfloor n/2 \rfloor - 1$ 
3     while  $i \geq 0$ :
4         Heapify ( $\prec$ ,  $i$ ,  $n$ ,  $\mathcal{P}$ )
5          $i \leftarrow i - 1$ 

```

Верхнюю оценку сложности алгоритма BuildHeap можно записать как $\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$.

Действительно, на «этаже» высотой h находятся максимум $\lceil n/2^{h+1} \rceil$ элементов, для каждого из которых алгоритм Heapify отрабатывает за время $O(h)$. Высоты «этажей» изменяются от 0 до $\lfloor \log_2 n \rfloor$.

То, что в формуле учитываются не только «нелистовые» элементы, не должно нас смущать, так как мы даём верхнюю границу сложности.

Упростим выражение для верхней оценки сложности:

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) < \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{n}{2^{h+1}} \cdot c \cdot h \text{ для некоторого } c > 0.$$

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{n}{2^{h+1}} \cdot c \cdot h = \frac{c \cdot n}{2} \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} < \frac{c \cdot n}{2} \sum_{h=0}^{\infty} \frac{h}{2^h}.$$

Существует формула: $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$. Она выводится путём дифференцирования обеих частей формулы для суммы геометрической прогрессии.

Подставляя $x = 1/2$, получаем $\sum_{k=0}^{\infty} \frac{k}{2^k} = \frac{1/2}{(1 - 1/2)^2} = 2$.

Тем самым

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

Получается, что асимптотическая сложность алгоритма BuildHeap: $O(n)$.

Основная идея пирамидальной сортировки:

1. преобразуем сортируемую последовательность в пирамиду с помощью BuildHeap, при этом максимальный элемент становится первым;
2. переставляем первый и последний элемент, при этом максимальный элемент попадает на своё место;
3. применяем Heapify для первого элемента последовательности, лишённой последнего элемента, чтобы восстановить пирамиду;
4. повторяем предыдущие два шага до тех пор, пока длина сортируемой последовательности не уменьшится до 1.

```
1 HeapSort(in  $\prec$ , in  $n$ , out  $\mathcal{P}$ )
2      $\mathcal{P} \leftarrow \mathbf{Id}_n$ 
3     BuildHeap( $\prec$ ,  $n$ ,  $\mathcal{P}$ )
4      $i \leftarrow n - 1$ 
5     while  $i > 0$ :
6          $\mathcal{P}[0] \leftrightarrow \mathcal{P}[i]$ 
7         Heapify( $\prec$ , 0,  $i$ ,  $\mathcal{P}$ )
8          $i \leftarrow i - 1$ 
```

Сложность алгоритма: $O(n \lg n)$. Алгоритм неустойчивый.

§10. Сортировка слиянием

Пусть дана последовательность натуральных чисел $\{a_i\}_0^{n-1}$ и известно, что подпоследовательности $a[k:l]$ и $a[l+1:m]$ отсортированы в соответствии с некоторым отношением строгого порядка $\prec \subseteq \mathbb{N}_n \times \mathbb{N}_n$.

Пример. ($n = 12$, отношение \prec совпадает с порядком на \mathbb{N})

50, 40, 35, $\underbrace{3, 6, 8, 9}_{a[3:6]}$, $\underbrace{2, 3', 7}_{a[7:9]}$, 20, 15

Одна из двух троек помечена «штрихом», чтобы они различались.

Необходимо выполнить *слияние* $a[k:l]$ и $a[l+1:m]$, то есть сформировать подпоследовательность $a[k:m]$ таким образом, чтобы она была составлена из элементов $a[k:l]$ и $a[l+1:m]$ и при этом устойчиво отсортирована.

Пример. (наша последовательность после слияния)

50, 40, 35, $\underbrace{2, 3, 3', 6, 7, 8, 9}_{a[3:9]}$, 20, 15

Запишем алгоритм слияния двух подпоследовательностей. Основная идея алгоритма: мы двигаемся сразу вдоль обеих подпоследовательностей, сравниваем их текущие элементы и копируем наименьший во вспомогательную последовательность.

```

1 Merge(in  $\prec$ , in  $k$ , in  $l$ , in  $m$ , in/out  $\mathcal{P}$ )
2    $\mathcal{T}$  — вспомогательная посл-ть размера  $m - k + 1$ 
3    $i \leftarrow k$  (счётчик по  $\mathcal{P}[k : l]$ )
4    $j \leftarrow l + 1$  (счётчик по  $\mathcal{P}[l + 1 : m]$ )
5    $h \leftarrow 0$  (счётчик по  $\mathcal{T}$ )
6   while  $h < m - k + 1$ :
7       if  $j \leq m$  and  $(i = l + 1$  or  $\mathcal{P}[j] \prec \mathcal{P}[i])$ :
8            $\mathcal{T}[h] \leftarrow \mathcal{P}[j]$ 
9            $j \leftarrow j + 1$ 
10      else :
11           $\mathcal{T}[h] \leftarrow \mathcal{P}[i]$ 
12           $i \leftarrow i + 1$ 
13       $h \leftarrow h + 1$ 
14   $\mathcal{P}[k : m] \leftarrow \mathcal{T}[0 : h - 1]$ 

```

Алгоритм устойчив, так как в случае $\mathcal{P}[j] = \mathcal{P}[i]$ срабатывают строки 11..12. Асимптотическая сложность алгоритма: $\Theta(m - k + 1)$.

Алгоритм сортировки слиянием – рекурсивный. Его основная идея:

- разбиваем сортируемую последовательность напополам;
- рекурсивно вызываем алгоритм сортировки для каждой половины последовательности;
- выполняем слияние получившихся отсортированных подпоследовательностей.

```
1 MergeSort(in  $\prec$ , in  $n$ , out  $\mathcal{P}$ )
2      $\mathcal{P} \leftarrow \mathbf{Id}_n$ 
3     MergeSortRec( $\prec$ , 0,  $n - 1$ ,  $\mathcal{P}$ )

5 MergeSortRec(in  $\prec$ , in  $low$ , in  $high$ , in/out  $\mathcal{P}$ )
6     if  $low < high$ :
7          $med \leftarrow \lfloor (low + high) / 2 \rfloor$ 
8         MergeSortRec( $\prec$ ,  $low$ ,  $med$ ,  $\mathcal{P}$ )
9         MergeSortRec( $\prec$ ,  $med + 1$ ,  $high$ ,  $\mathcal{P}$ )
10        Merge( $\prec$ ,  $low$ ,  $med$ ,  $high$ ,  $\mathcal{P}$ )
```

Пусть время работы алгоритма сортировки слиянием в зависимости от длины сортируемой последовательности задаётся функцией $T(n)$. Тогда эта функция определяется *рекуррентным соотношением*

$$T(n) = \begin{cases} \Theta(1) & \text{при } n = 1, \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{при } n > 1. \end{cases}$$

Действительно, при $n = 1$ алгоритм отрабатывает за константное время, а при $n > 1$ время его работы складывается из времени двух сортировок последовательностей длины $n/2$ и времени работы алгоритма слияния, которое линейно зависит от n .

Можно доказать, что $T(n) = \Theta(n \log_2 n)$, откуда следует асимптотически точная оценка сложности алгоритма сортировки слиянием: $\Theta(n \lg n)$.

Устойчивость сортировки слиянием следует из устойчивости алгоритма Merge.

Достоинства сортировки слиянием по сравнению с пирамидальной сортировкой:

1. устойчивость;
2. хорошо распараллеливается на многоядерных процессорах;
3. лучше использует кэш процессора.

Недостаток сортировки слиянием – использование вспомогательного блока памяти размером $O(n)$.

§11. Быстрая сортировка

Рассмотрим последовательность натуральных чисел $\{a_i\}_0^{n-1}$ и некоторое отношение строгого порядка $\prec \subseteq \mathbb{N}_n \times \mathbb{N}_n$. Выберем один из элементов последовательности и назовём его *опорным* элементом.

Пусть q элементов последовательности меньше опорного элемента, тогда определим операцию *разделения* последовательности относительно выбранного опорного элемента как такое переупорядочивание элементов последовательности, что:

1. все элементы из $a[0 : q - 1]$ меньше опорного элемента;
 2. элемент $a[q]$ равен опорному элементу;
 3. опорный элемент не превышает ни один из элементов $a[q + 1 : n - 1]$.
- Число q назовём *границей разделения*.

Пример. Пусть $a = \langle 4, 5, 6, 7, 3, 9, \underline{5} \rangle$ (опорный элемент подчёркнут).

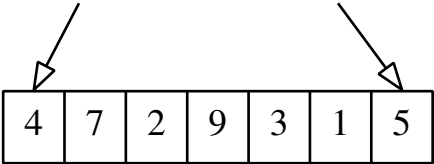
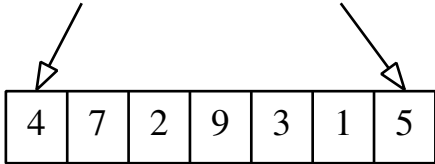
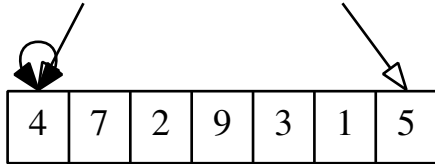
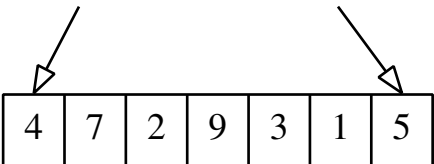
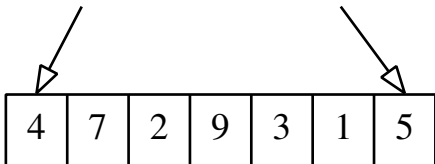
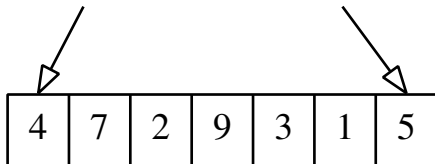
Если отношение \prec совпадает с порядком на \mathbb{N} , то результат разделения можно записать как $\langle 4, 3, \underline{5}, 6, 9, 7, 5 \rangle$. Возможны и другие варианты разделения, например: $\langle 3, 4, \underline{5}, 6, 9, 5, 7 \rangle$.

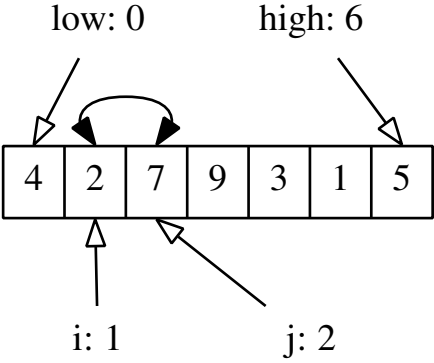
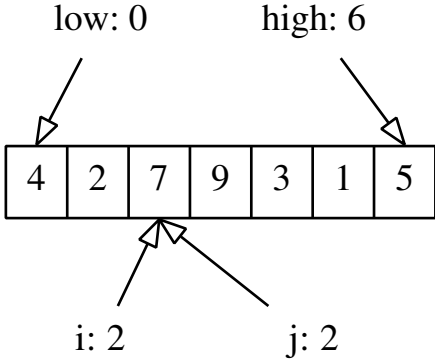
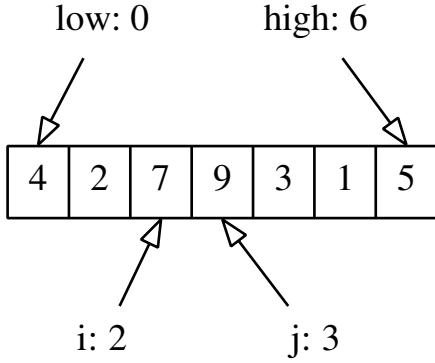
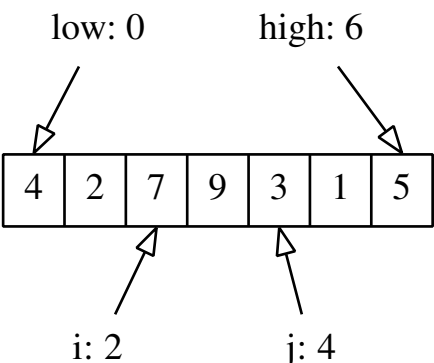
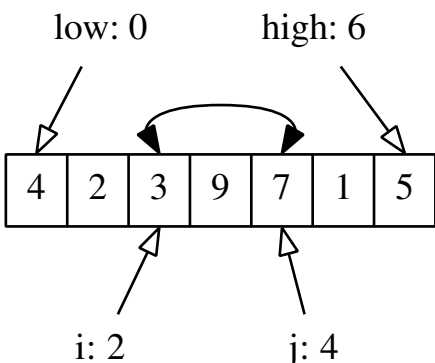
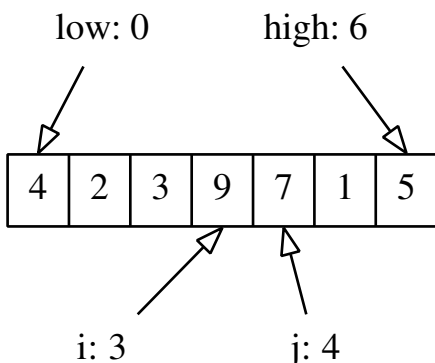
Алгоритм Partition осуществляет разделение подпоследовательности $\mathcal{P}[low : high]$ относительно элемента $\mathcal{P}[high]$. Алгоритм возвращает границу разделения.

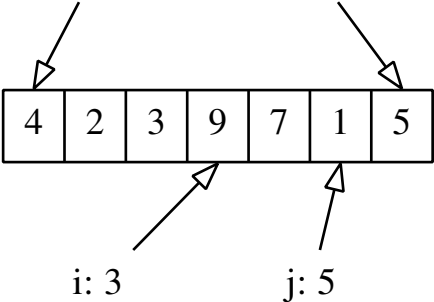
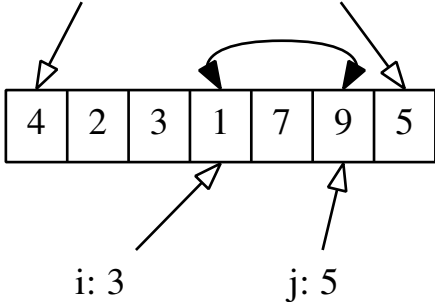
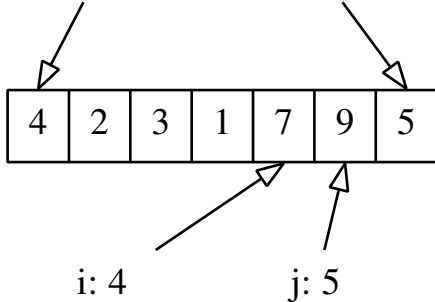
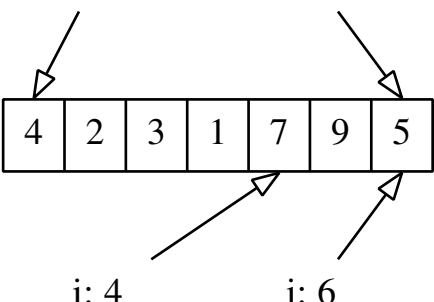
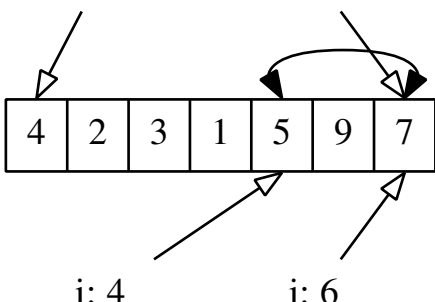
```
1 Partition (in  $\prec$ , in  $low$ , in  $high$ , in/out  $\mathcal{P}$ ):  $i$ 
2      $i \leftarrow low$ 
3      $j \leftarrow low$ 
4     while  $j < high$ :
5         if  $\mathcal{P}[j] \prec \mathcal{P}[high]$ :
6              $\mathcal{P}[i] \leftrightarrow \mathcal{P}[j]$ 
7              $i \leftarrow i + 1$ 
8          $j \leftarrow j + 1$ 
9      $\mathcal{P}[i] \leftrightarrow \mathcal{P}[high]$ 
```

В процессе работы алгоритма переменная i указывает на элемент, левее которого расположены элементы, меньшие опорного.

Сложность алгоритма составляет $O(high - low)$. Алгоритм неустойчивый.

<p>low: 0 high: 6</p>  <p>i: ? j: ?</p>	<p>low: 0 high: 6</p>  <p>i: 0 j: 0</p>	<p>low: 0 high: 6</p>  <p>i: 0 j: 0</p>
	<p>1. $i \leftarrow low, j \leftarrow low$</p>	<p>2. $\mathcal{P}[i] \leftrightarrow \mathcal{P}[j]$</p>
<p>low: 0 high: 6</p>  <p>i: 1 j: 0</p>	<p>low: 0 high: 6</p>  <p>i: 1 j: 1</p>	<p>low: 0 high: 6</p>  <p>i: 1 j: 2</p>
<p>3. $i \leftarrow i + 1$</p>	<p>4. $j \leftarrow j + 1$</p>	<p>5. $j \leftarrow j + 1$</p>

<p>low: 0 high: 6</p> 	<p>low: 0 high: 6</p> 	<p>low: 0 high: 6</p> 
<p>6. $\mathcal{P}[i] \leftrightarrow \mathcal{P}[j]$</p>	<p>7. $i \leftarrow i + 1$</p>	<p>8. $j \leftarrow j + 1$</p>
<p>low: 0 high: 6</p> 	<p>low: 0 high: 6</p> 	<p>low: 0 high: 6</p> 
<p>9. $j \leftarrow j + 1$</p>	<p>10. $\mathcal{P}[i] \leftrightarrow \mathcal{P}[j]$</p>	<p>11. $i \leftarrow i + 1$</p>

<p>low: 0 high: 6</p>  <p>i: 3 j: 5</p>	<p>low: 0 high: 6</p>  <p>i: 3 j: 5</p>	<p>low: 0 high: 6</p>  <p>i: 4 j: 5</p>
<p>12. $j \leftarrow j + 1$</p>	<p>13. $\mathcal{P}[i] \leftrightarrow \mathcal{P}[j]$</p>	<p>14. $i \leftarrow i + 1$</p>
<p>low: 0 high: 6</p>  <p>i: 4 j: 6</p>	<p>low: 0 high: 6</p>  <p>i: 4 j: 6</p>	
<p>15. $j \leftarrow j + 1$</p>	<p>16. $\mathcal{P}[i] \leftrightarrow \mathcal{P}[high]$</p>	

Алгоритм быстрой сортировки – рекурсивный. Его основная идея:

- выполняем разделение последовательности относительно последнего элемента (этот элемент оказывается на границе разделения и тем самым попадает на своё место в отсортированной последовательности);
- рекурсивно вызываем алгоритм сортировки для двух подпоследовательностей, расположенных левее и правее границы разделения.

```
1 QuickSort(in  $\prec$ , in  $n$ , out  $\mathcal{P}$ )
2      $\mathcal{P} \leftarrow \mathbf{Id}_n$ 
3     QuickSortRec( $\prec$ , 0,  $n - 1$ ,  $\mathcal{P}$ )

5 QuickSortRec(in  $\prec$ , in  $low$ , in  $high$ , in/out  $\mathcal{P}$ )
6     if  $low < high$ :
7          $q \leftarrow \text{Partition}(\prec, low, high, \mathcal{P})$ 
8         QuickSortRec( $\prec$ ,  $low$ ,  $q - 1$ ,  $\mathcal{P}$ )
9         QuickSortRec( $\prec$ ,  $q + 1$ ,  $high$ ,  $\mathcal{P}$ )
```

Наилучший случай для алгоритма быстрой сортировки – это когда в результате любого деления граница оказывается посередине разделяемой подпоследовательности. Рекуррентное соотношение для этого случая – такое же, что у сортировки слиянием. Отсюда асимптотическая нижняя граница сложности алгоритма: $\Omega(n \lg n)$.

Наихудший случай – когда в результате любого деления граница указывает на начало или конец разделяемой подпоследовательности. В этом случае осмысленные действия выполняются только в одном из двух рекурсивных вызовов. Причём на каждом запуске функции сортировки сортируемая последовательность уменьшается на один элемент. Отсюда асимптотическая верхняя граница сложности алгоритма: $O(n^2)$.

На практике оказывается, что в среднем алгоритм тяготеет к нижней границе сложности.

§12. Сортировка за линейное время

Пускай ключами записей сортируемой последовательности являются натуральные числа из \mathbb{N}_m .

Зададим функцию, отображающую множество записей R в множество ключей:

$$key : R \rightarrow \mathbb{N}_m.$$

Поставим задачу сортировки последовательности записей

$$S = \{s_i\}_0^{n-1}$$

следующим образом: требуется построить такую последовательность

$$D = \{d_i\}_0^{n-1},$$

что она составлена из элементов S и, кроме того,

$$key(d_0) \leq key(d_1) \leq \dots \leq key(d_{n-1}).$$

Замечание. Такая постановка задачи отличается от классической тем, что алгоритм сортировки имеет доступ к записям и их ключам.

Алгоритм сортировки распределением копирует записи из исходной последовательности в новую отсортированную последовательность. Идея алгоритма:

- для каждого ключа подсчитываем количество записей, которым он соответствует;
- для каждого ключа вычисляем индекс элемента в отсортированной последовательности, непосредственно перед которым может располагаться последняя запись, соответствующая этому ключу;
- копируем каждую запись исходной последовательности в нужное место результирующей последовательности.

Пример. (сортируем целые числа по возрастанию младшего разряда)

$\langle 12, 4, 51, 42, 11, 31, 54, 32, 50 \rangle$ – исходная последовательность;

$\left\langle \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1, & 3, & 3, & 0, & 2, & 0, & 0, & 0, & 0, & 0 \end{matrix} \right\rangle$ – количество записей для каждого ключа;

$\left\langle \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1, & 4, & 7, & 7, & 9, & 9, & 9, & 9, & 9, & 9 \end{matrix} \right\rangle$ – индекс для каждого ключа;

$\langle 50, 51, 11, 31, 12, 42, 32, 4, 54 \rangle$ – отсортированная последовательность.

```

1  DistributionSort(in key, in m, in S, in n): D
2      count  $\leftarrow$  Nullm
3      j  $\leftarrow$  0
4      while j < n:
5          k  $\leftarrow$  key(S[j])
6          count[k]  $\leftarrow$  count[k] + 1
7          j  $\leftarrow$  j + 1
8      i  $\leftarrow$  1
9      while i < m:
10         count[i]  $\leftarrow$  count[i] + count[i - 1]
11         i  $\leftarrow$  i + 1
12     D  $\leftarrow$  новая последовательность записей длины n
13     j  $\leftarrow$  n - 1
14     while j  $\geq$  0:
15         k  $\leftarrow$  key(S[j])
16         i  $\leftarrow$  count[k] - 1
17         count[k]  $\leftarrow$  i
18         D[i]  $\leftarrow$  S[j]
19         j  $\leftarrow$  j - 1

```

Очевидно, что сортировка распределением выполняется за время $\Theta(2n + m)$ что даёт $\Theta(n)$ в случае, когда $m \ll n$.

Размер дополнительной памяти, требуемой сортировкой распределением: $\Theta(m + n)$.

Благодаря тому, что в строках 14..19 исходная последовательность просматривается из конца в начало, и при этом диапазоны отсортированной последовательности, соответствующие одному ключу, заполняются также из конца в начало, алгоритм сортировки распределением – устойчивый.

Сортировка распределением хорошо работает только в случае, когда множество ключей невелико.

В случае, если множество ключей велико, применяют следующий приём: записывают каждый ключ в системе счисления по основанию m и подвергают последовательность нескольким сортировкам распределением по каждому разряду ключа, начиная с младшего.

Пример. ($m = 10$)

115	200	200	015
073	100	100	073
015	112	112	075
313	⇒ 073	⇒ 313	⇒ 100
200	313	115	112
100	115	015	115
075	015	073	200
112	075	075	313

Этот метод называется поразрядной сортировкой. Он работает благодаря тому, что сортировка распределением – устойчивая.

Пусть в системе счисления по основанию m каждый ключ представлен кортежем $\langle a_0, a_1, \dots, a_{q-1} \rangle$, где $a_i \in \mathbb{N}_m$.

Зададим функцию, возвращающую нужный разряд ключа:
 $keys : \mathbb{N}_q \rightarrow (R \rightarrow \mathbb{N}_m)$.

Тогда алгоритм поразрядной сортировки можно записать как

```
1 RadixSort(in keys, in q, in m, in S, in n): D
2     D ← S
3     i ← q − 1
4     while i ≥ 0:
5         D ← DistributionSort(keys(i), m, D, n)
6         i ← i − 1
```

Алгоритм, очевидно, устойчив, и выполняется за время $\Theta(n)$.

Замечание. Алгоритм можно без труда обобщить для случая, когда основание системы счисления различается у разных разрядов.

§13. Постановка задачи поиска подстроки

Строка – это последовательность, элементы которой принадлежат конечному множеству, называемому *алфавитом*.

Без потери общности можно считать, что алфавитом является конечное подмножество \mathbb{N} . Отсюда строка – последовательность натуральных чисел.

Если дана строка S , то

- её i -тый элемент мы будем обозначать как $S[i]$;
- длину строки – как $\text{len}(S)$;
- подстроку $\langle S[j], S[j+1], \dots, S[k] \rangle$ – как $S[j : k]$.

Поиск подстроки S в строке T заключается в нахождении минимального k такого, что $T[k : k + \text{len}(S) - 1] = S$ (первое вхождение S в T). В случае, если это равенство не выполняется ни для какого k , положим $k = \text{len}(T)$.

Наивный алгоритм поиска подстроки в строке:

```
1 BruteForceSubst(in S, in T, out k)
2      $k \leftarrow 0$ 
3     while  $k < \text{len}(T) - \text{len}(S) + 1$ :
4         if  $T[k : k + \text{len}(S) - 1] = S$ :
5             return
6          $k \leftarrow k + 1$ 
7      $k \leftarrow \text{len}(T)$ 
```

Так как операция сравнения подстрок работает за время $O(\text{len}(S))$, то сложность алгоритма составляет $O((\text{len}(T) - \text{len}(S) + 1) \cdot \text{len}(S))$.

Наихудший случай: $T = aaaaaa \dots a$, $S = aaaa \dots ab$.

Достоинством алгоритма является то, что в нём как «чёрный ящик» используется операция сравнения подстрок, которая допускает эффективную реализацию на ассемблере. (При реализации на языке C эта операция кодируется как вызов функции `memcmp`.)

§14. Префиксная функция

Префикс длины k строки S – это подстрока $S[0 : k - 1]$.

Суффикс длины k строки S – это подстрока $S[\text{len}(S) - k : \text{len}(S) - 1]$.

При этом, если $k < \text{len}(S)$, то префикс и суффикс – *собственные*.

В приложении к поиску подстроки в строке нас будут интересовать строки, у которых собственный префикс совпадает с суффиксом.

Пример. $\underbrace{abra}_{k=4}cad\underbrace{abra}.$

Для общности будем полагать, что у любой строки префикс нулевой длины совпадает с суффиксом.

Пусть S – строка. Обозначим через $l(S)$ самый длинный собственный префикс S , совпадающий с суффиксом S .

Утверждение. $l(l(S))$ совпадает с некоторым суффиксом S .

Данное утверждение иллюстрирует схема:

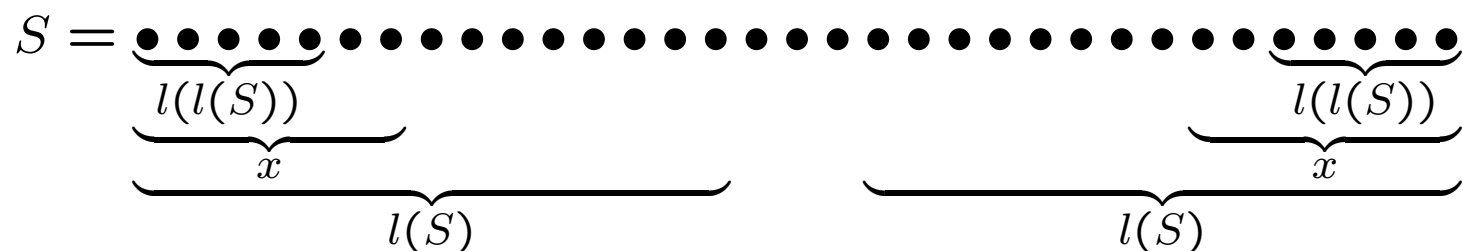
$$S = \underbrace{\overbrace{\bullet \bullet \bullet \bullet}^{l(l(S))} \bullet \bullet \bullet \overbrace{\bullet \bullet \bullet \bullet}^{l(l(S))} \bullet \bullet \bullet}_{l(S)} \cdots \underbrace{\overbrace{\bullet \bullet \bullet \bullet}^{l(l(S))} \bullet \bullet \bullet \overbrace{\bullet \bullet \bullet \bullet}^{l(l(S))} \bullet \bullet \bullet}_{l(S)}$$

Действительно, в начале и в конце строки S располагается подстрока $l(S)$, а в начале и в конце строки $l(S)$ располагается подстрока $l(l(S))$. Следовательно, $l(l(S))$ располагается в начале и в конце строки S .

Утверждение естественно распространяется и на $l(l(l(S)))$ и т.д.

Утверждение. Не существует префикса строки S , который бы совпадал с её суффиксом и был бы короче $l(S)$, но длиннее $l(l(S))$.

Предположим, что такой суффикс x существует. Получившуюся ситуацию иллюстрирует схема:



Из схемы понятно, что x располагается в начале и в конце $l(S)$, что невозможно, потому что $l(l(S))$ – самый длинный собственный префикс $l(S)$, совпадающий с её суффиксом.

Префиксная функция для строки S – это функция $\pi : \mathbb{N}_{len(S)-1} \longrightarrow \mathbb{N}_{len(S)-1}$ такая, что $\pi(i)$ равно длине $l(S[0 : i])$.

Пример.

$S =$	a	b	r	a	c	a	d	a	b	r	a
$\pi =$	0	0	0	1	0	1	0	1	2	3	4

Наивный алгоритм для вычисления префиксной функции:

```

1 BruteForcePrefix(in  $S$ ):  $\pi$ 
2    $\pi \leftarrow$  новая последовательность нат. чисел длины  $len(S)$ 
3    $\pi[0] \leftarrow 0$ 
4    $i \leftarrow 1$ 
5   while  $i < len(S)$ :
6        $k \leftarrow i$ 
7       while  $k > 0$  and  $S[0 : k - 1] \neq S[i - k + 1 : i]$ :
8            $k \leftarrow k - 1$ 
9        $\pi[i] \leftarrow k$ 
10       $i \leftarrow i + 1$ 

```

Сложность алгоритма – $O((len(S))^3)$.

Обозначения. Если A – строка, x – символ алфавита, то Ax – строка, полученная из A путём добавления к ней справа символа x .

Если A и B – строки, то AB – результат их конкатенации.

Пусть префиксная функция π построена для некоторой строки S длины n , и x – символ алфавита. Можем ли мы быстро посчитать $l(Sx)$?

Пусть $y_1 = S[\pi(n-1)]$ – первый символ строки S , следующий за префиксом $l(S)$. Очевидно, что если $y_1 = x$, то $l(Sx) = l(S)x$.

[illegible]

Если $y_1 \neq x$, то рассмотрим следующий по длине префикс строки S , совпадающий с её суффиксом: $l(l(S))$.

$$Sx = \underbrace{\overbrace{0 \bullet \bullet \bullet \bullet}^{l(l(S))}}_{l(S)}^{\pi(\pi(n-1)-1)} y_2 \dots \overset{\pi(n-1)}{\bullet} \dots \underbrace{\bullet \bullet \bullet \bullet}_{l(l(S))} x_n.$$

Если $y_2 = \pi(\pi(n-1)-1) = x$, то $l(Sx) = l(l(S))x$. (И так далее.)

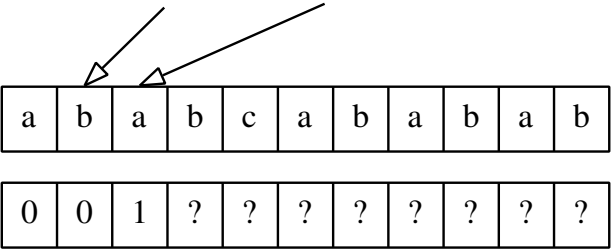
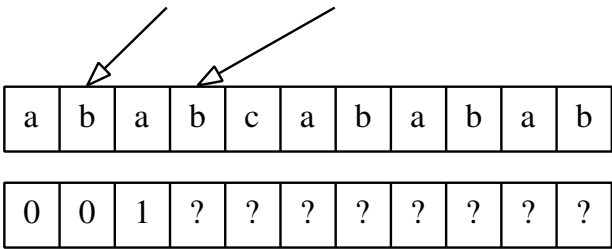
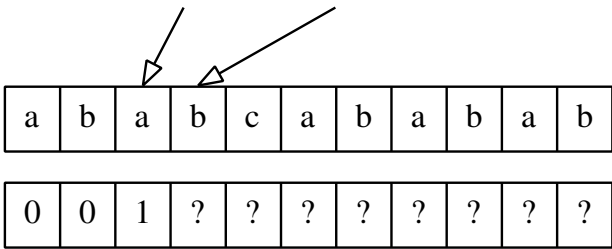
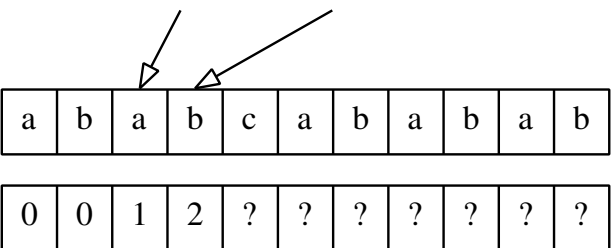
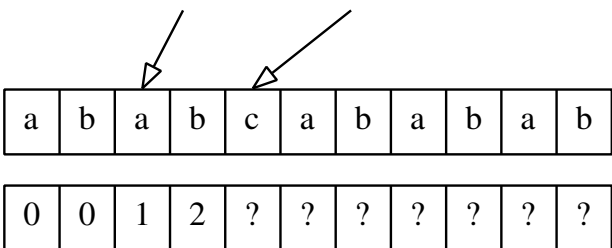
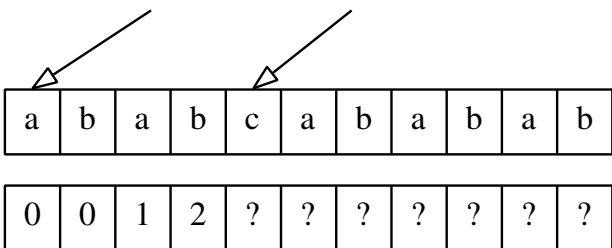
Основная идея алгоритма построения префиксной функции:

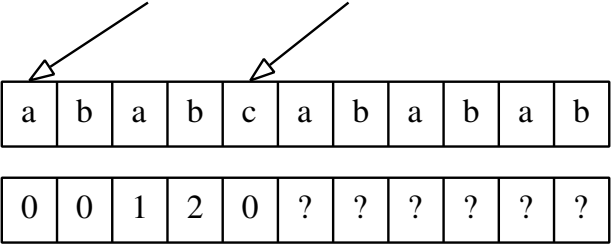
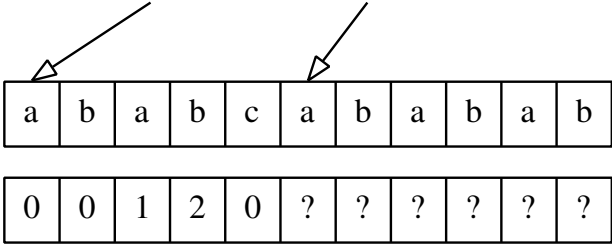
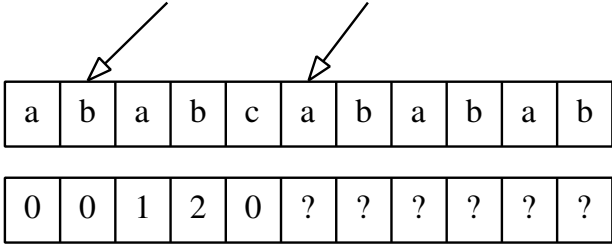
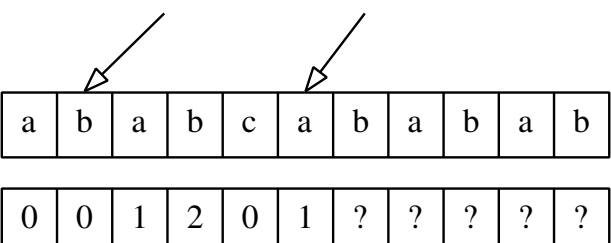
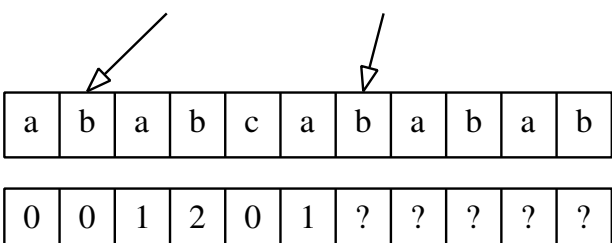
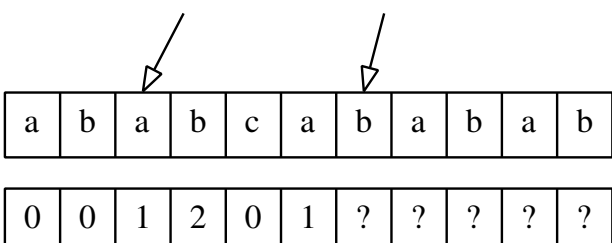
- пусть $\pi(0) = 0$, тем самым мы построили префиксную функцию для подстроки $S[0 : 0]$;
- на каждом шаге имеем построенную префиксную функцию для $S[0 : i - 1]$ и вычисляем $\pi(i)$ по приведённой на предыдущем слайде схеме.

```
1 Prefix(in S):  $\pi$ 
2    $\pi \leftarrow$  новая последовательность нат. чисел длины  $len(S)$ 
3    $\pi[0] \leftarrow t \leftarrow 0$ 
4    $i \leftarrow 1$ 
5   while  $i < len(S)$ :
6       while  $t > 0$  and  $S[t] \neq S[i]$ :
7            $t \leftarrow \pi[t - 1]$ 
8       if  $S[t] = S[i]$ :
9            $t \leftarrow t + 1$ 
10       $\pi[i] \leftarrow t$ 
11       $i \leftarrow i + 1$ 
```

Переменная t в начале каждой итерации внешнего цикла равна $\pi(i - 1)$, а в конце каждой итерации равна $\pi(i)$.

<div><div>t: ?i: ?</div><div><table><tr><td>a</td><td>b</td><td>a</td><td>b</td><td>c</td><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>b</td></tr><tr><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table></div></div>	a	b	a	b	c	a	b	a	b	a	b	?	?	?	?	?	?	?	?	?	?	?	<div><div>t: 0i: ?</div><div><div><div></div><div></div></div><table><tr><td>a</td><td>b</td><td>a</td><td>b</td><td>c</td><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>b</td></tr><tr><td>0</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table></div></div>	a	b	a	b	c	a	b	a	b	a	b	0	?	?	?	?	?	?	?	?	?	?	<div><div>t: 0i: 1</div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div><table><tr><td>a</td><td>b</td><td>a</td><td>b</td><td>c</td><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>b</td></tr><tr><td>0</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table></div></div>	a	b	a	b	c	a	b	a	b	a	b	0	?	?	?	?	?	?	?	?	?	?
a	b	a	b	c	a	b	a	b	a	b																																																										
?	?	?	?	?	?	?	?	?	?	?																																																										
a	b	a	b	c	a	b	a	b	a	b																																																										
0	?	?	?	?	?	?	?	?	?	?																																																										
a	b	a	b	c	a	b	a	b	a	b																																																										
0	?	?	?	?	?	?	?	?	?	?																																																										
	<div>1. $\pi[0] \leftarrow t \leftarrow 0$</div>	<div>2. $i \leftarrow 1$</div>																																																																		
<div><div>t: 0i: 1</div><div><div><div></div><div></div></div><div><div></div><div></div></div></div><table><tr><td>a</td><td>b</td><td>a</td><td>b</td><td>c</td><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>b</td></tr><tr><td>0</td><td>0</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table></div>	a	b	a	b	c	a	b	a	b	a	b	0	0	?	?	?	?	?	?	?	?	?	<div><div>t: 0i: 2</div><div><div><div></div><div></div></div><div><div></div><div></div></div></div><table><tr><td>a</td><td>b</td><td>a</td><td>b</td><td>c</td><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>b</td></tr><tr><td>0</td><td>0</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table></div>	a	b	a	b	c	a	b	a	b	a	b	0	0	?	?	?	?	?	?	?	?	?	<div><div>t: 1i: 2</div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div><div><div></div><div></div></div></div><table><tr><td>a</td><td>b</td><td>a</td><td>b</td><td>c</td><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>b</td></tr><tr><td>0</td><td>0</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table></div>	a	b	a	b	c	a	b	a	b	a	b	0	0	?	?	?	?	?	?	?	?	?
a	b	a	b	c	a	b	a	b	a	b																																																										
0	0	?	?	?	?	?	?	?	?	?																																																										
a	b	a	b	c	a	b	a	b	a	b																																																										
0	0	?	?	?	?	?	?	?	?	?																																																										
a	b	a	b	c	a	b	a	b	a	b																																																										
0	0	?	?	?	?	?	?	?	?	?																																																										
<div>3. $\pi[i] \leftarrow t$</div>	<div>4. $i \leftarrow i + 1$</div>	<div>5. $t \leftarrow t + 1$</div>																																																																		

<p style="text-align: center;">t: 1 i: 2</p> 	<p style="text-align: center;">t: 1 i: 3</p> 	<p style="text-align: center;">t: 2 i: 3</p> 
<p>6. $\pi[i] \leftarrow t$</p>	<p>7. $i \leftarrow i + 1$</p>	<p>8. $t \leftarrow t + 1$</p>
<p style="text-align: center;">t: 2 i: 3</p> 	<p style="text-align: center;">t: 2 i: 4</p> 	<p style="text-align: center;">t: 0 i: 4</p> 
<p>9. $\pi[i] \leftarrow t$</p>	<p>10. $i \leftarrow i + 1$</p>	<p>11. $t \leftarrow \pi[t - 1]$</p>

<p style="text-align: center;">t: 0 i: 4</p> 	<p style="text-align: center;">t: 0 i: 5</p> 	<p style="text-align: center;">t: 1 i: 5</p> 
<p>12. $\pi[i] \leftarrow t$</p>	<p>13. $i \leftarrow i + 1$</p>	<p>14. $t \leftarrow t + 1$</p>
<p style="text-align: center;">t: 1 i: 5</p> 	<p style="text-align: center;">t: 1 i: 6</p> 	<p style="text-align: center;">t: 2 i: 6</p> 
<p>15. $\pi[i] \leftarrow t$</p>	<p>16. $i \leftarrow i + 1$</p>	<p>17. $t \leftarrow t + 1$</p>

<p>t: 2 i: 6</p> <p>a b a b c a b a b a</p> <p>0 0 1 2 0 1 2 ? ? ? ?</p>	<p>t: 2 i: 7</p> <p>a b a b c a b a b a</p> <p>0 0 1 2 0 1 2 ? ? ? ?</p>	<p>t: 3 i: 7</p> <p>a b a b c a b a b a</p> <p>0 0 1 2 0 1 2 ? ? ? ?</p>
18. $\pi[i] \leftarrow t$	19. $i \leftarrow i + 1$	20. $t \leftarrow t + 1$
<p>t: 3 i: 7</p> <p>a b a b c a b a b a</p> <p>0 0 1 2 0 1 2 3 ? ? ?</p>	<p>t: 3 i: 8</p> <p>a b a b c a b a b a</p> <p>0 0 1 2 0 1 2 3 ? ? ?</p>	<p>t: 4 i: 8</p> <p>a b a b c a b a b a</p> <p>0 0 1 2 0 1 2 3 ? ? ?</p>
21. $\pi[i] \leftarrow t$	22. $i \leftarrow i + 1$	23. $t \leftarrow t + 1$

<p style="text-align: center;">t: 4 i: 8</p> <p>a b a b c a b a b a b</p> <p>0 0 1 2 0 1 2 3 4 ? ?</p>	<p style="text-align: center;">t: 4 i: 9</p> <p>a b a b c a b a b a b</p> <p>0 0 1 2 0 1 2 3 4 ? ?</p>	<p style="text-align: center;">t: 2 i: 9</p> <p>a b a b c a b a b a b</p> <p>0 0 1 2 0 1 2 3 4 ? ?</p>
<p>24. $\pi[i] \leftarrow t$</p>	<p>25. $i \leftarrow i + 1$</p>	<p>26. $t \leftarrow \pi[t - 1]$</p>
<p style="text-align: center;">t: 3 i: 9</p> <p>a b a b c a b a b a b</p> <p>0 0 1 2 0 1 2 3 4 ? ?</p>	<p style="text-align: center;">t: 3 i: 9</p> <p>a b a b c a b a b a b</p> <p>0 0 1 2 0 1 2 3 4 3 ?</p>	<p style="text-align: center;">t: 3 i: 10</p> <p>a b a b c a b a b a b</p> <p>0 0 1 2 0 1 2 3 4 3 ?</p>
<p>27. $t \leftarrow t + 1$</p>	<p>28. $\pi[i] \leftarrow t$</p>	<p>29. $i \leftarrow i + 1$</p>

<p>t: 4 i: 10</p>	<p>t: 4 i: 10</p>	<p>t: 4 i: 11</p>
30. $t \leftarrow t + 1$	31. $\pi[i] \leftarrow t$	32. $i \leftarrow i + 1$

Чтобы оценить время работы алгоритма, воспользуемся следующими соображениями:

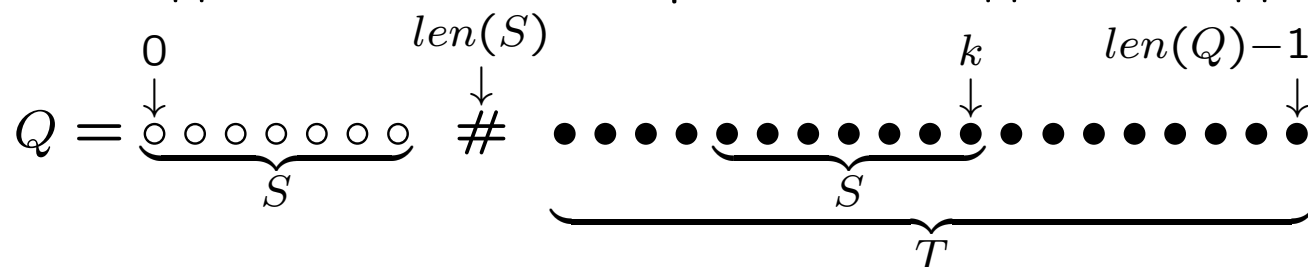
1. внешний цикл выполняется $n - 1$ раз, где $n = \text{len}(S)$;
2. изначально $t = 0$;
3. на каждой итерации внешнего цикла t может увеличиться не более, чем на единицу (строка 8), поэтому максимальное значение, которого может достигнуть t — это $n - 1$;
4. во внутреннем цикле t только уменьшается (строка 6), но не может стать меньше нуля, поэтому общее количество итераций внутреннего цикла за всё время работы алгоритма не может превышать $n - 1$.

Отсюда, сложность алгоритма: $O(n)$.

§15. Алгоритм Кнута-Морриса-Пратта

Основная идея алгоритма:

1. формируется строка $Q=S\#T$, где S – искомая подстрока, T – строка, в которой осуществляется поиск, а $\#$ – служебный символ, который гарантированно не встречается ни в S , ни в T ;
2. для строки Q выполняется построение префиксной функции π ;
3. участок T префиксной функции сканируется для обнаружения позиции k такой, что $\pi(k) = \text{len}(S)$; тем самым k будет равно индексу последнего символа первого вхождения подстроки S в строку T .



Значения префиксной функции на $Q = S\#T$ не могут превышать $\text{len}(S)$, потому что символ $\#$ – уникальный. Поэтому алгоритм Кнута-Морриса-Пратта строит только $\pi[0 : \text{len}(S) - 1]$ и вычисляет значения π на участке T , не сохраняя их в массиве:

```

1 KMPSubst(in  $S$ , in  $T$ , out  $k$ )
2      $\pi \leftarrow \text{Prefix}(S)$ 
3      $q \leftarrow 0$ 
4      $k \leftarrow 0$ 
5     while  $k < \text{len}(T)$ :
6         while  $q > 0$  and  $S[q] \neq T[k]$ :
7              $q \leftarrow \pi[q - 1]$ 
8         if  $S[q] = T[k]$ :
9              $q \leftarrow q + 1$ 
10        if  $q = \text{len}(S)$ :
11             $k \leftarrow k - \text{len}(S) + 1$ 
12            return
13         $k \leftarrow k + 1$ 

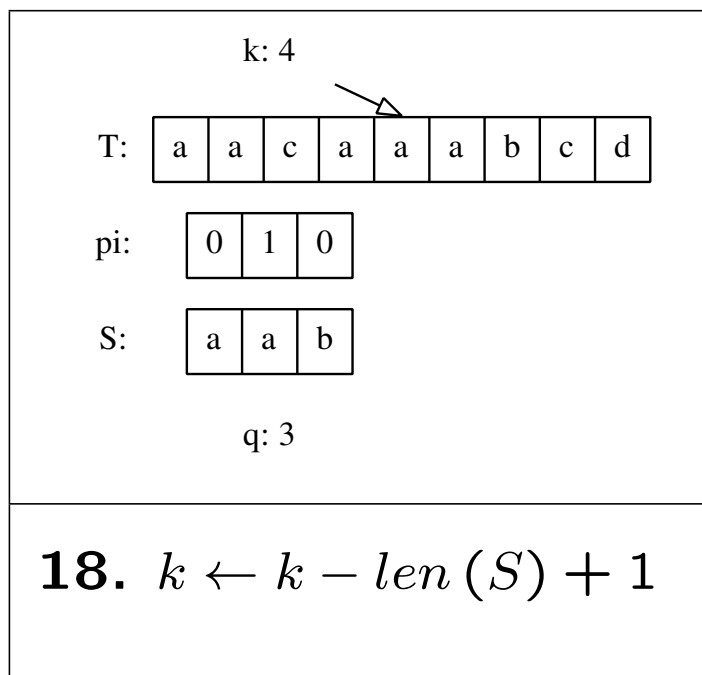
```

Так как для построения $\pi[0 : \text{len}(S) - 1]$ достаточно выполнить $\text{Prefix}(S)$, алгоритм вообще обходится без построения строки $Q = S\#T$.

<div><div>k: ?</div><div>T: <table><tr><td>a</td><td>a</td><td>c</td><td>a</td><td>a</td><td>a</td><td>b</td><td>c</td><td>d</td></tr></table></div><div>pi: <table><tr><td>?</td><td>?</td><td>?</td></tr></table></div><div>S: <table><tr><td>a</td><td>a</td><td>b</td></tr></table></div><div>q: ?</div></div>	a	a	c	a	a	a	b	c	d	?	?	?	a	a	b	<div><div>k: ?</div><div>T: <table><tr><td>a</td><td>a</td><td>c</td><td>a</td><td>a</td><td>a</td><td>b</td><td>c</td><td>d</td></tr></table></div><div>pi: <table><tr><td>0</td><td>1</td><td>0</td></tr></table></div><div>S: <table><tr><td>a</td><td>a</td><td>b</td></tr></table></div><div>q: ?</div></div>	a	a	c	a	a	a	b	c	d	0	1	0	a	a	b	<div><div>k: 0</div><div>T: <table><tr><td>a</td><td>a</td><td>c</td><td>a</td><td>a</td><td>a</td><td>b</td><td>c</td><td>d</td></tr></table></div><div>pi: <table><tr><td>0</td><td>1</td><td>0</td></tr></table></div><div>S: <table><tr><td>a</td><td>a</td><td>b</td></tr></table></div><div>q: 0</div></div>	a	a	c	a	a	a	b	c	d	0	1	0	a	a	b
a	a	c	a	a	a	b	c	d																																							
?	?	?																																													
a	a	b																																													
a	a	c	a	a	a	b	c	d																																							
0	1	0																																													
a	a	b																																													
a	a	c	a	a	a	b	c	d																																							
0	1	0																																													
a	a	b																																													
	1. $\pi \leftarrow \text{Prefix}(S)$	2. $q \leftarrow 0, k \leftarrow 0$																																													
<div><div>k: 0</div><div>T: <table><tr><td>a</td><td>a</td><td>c</td><td>a</td><td>a</td><td>a</td><td>b</td><td>c</td><td>d</td></tr></table></div><div>pi: <table><tr><td>0</td><td>1</td><td>0</td></tr></table></div><div>S: <table><tr><td>a</td><td>a</td><td>b</td></tr></table></div><div>q: 1</div></div>	a	a	c	a	a	a	b	c	d	0	1	0	a	a	b	<div><div>k: 1</div><div>T: <table><tr><td>a</td><td>a</td><td>c</td><td>a</td><td>a</td><td>a</td><td>b</td><td>c</td><td>d</td></tr></table></div><div>pi: <table><tr><td>0</td><td>1</td><td>0</td></tr></table></div><div>S: <table><tr><td>a</td><td>a</td><td>b</td></tr></table></div><div>q: 1</div></div>	a	a	c	a	a	a	b	c	d	0	1	0	a	a	b	<div><div>k: 1</div><div>T: <table><tr><td>a</td><td>a</td><td>c</td><td>a</td><td>a</td><td>a</td><td>b</td><td>c</td><td>d</td></tr></table></div><div>pi: <table><tr><td>0</td><td>1</td><td>0</td></tr></table></div><div>S: <table><tr><td>a</td><td>a</td><td>b</td></tr></table></div><div>q: 2</div></div>	a	a	c	a	a	a	b	c	d	0	1	0	a	a	b
a	a	c	a	a	a	b	c	d																																							
0	1	0																																													
a	a	b																																													
a	a	c	a	a	a	b	c	d																																							
0	1	0																																													
a	a	b																																													
a	a	c	a	a	a	b	c	d																																							
0	1	0																																													
a	a	b																																													
3. $q \leftarrow q + 1$	4. $k \leftarrow k + 1$	5. $q \leftarrow q + 1$																																													

<div><div>k: 2</div><div><div>T:</div><div><div>a</div><div>a</div><div>c</div><div>a</div><div>a</div><div>a</div><div>b</div><div>c</div><div>d</div></div></div><div><div>pi:</div><div><div>0</div><div>1</div><div>0</div></div></div><div><div>S:</div><div><div>a</div><div>a</div><div>b</div></div></div><div><div>q: 2</div></div></div>	<div><div>k: 2</div><div><div>T:</div><div><div>a</div><div>a</div><div>c</div><div>a</div><div>a</div><div>a</div><div>b</div><div>c</div><div>d</div></div></div><div><div>pi:</div><div><div>0</div><div>1</div><div>0</div></div></div><div><div>S:</div><div><div>a</div><div>a</div><div>b</div></div></div><div><div>q: 1</div></div></div>	<div><div>k: 2</div><div><div>T:</div><div><div>a</div><div>a</div><div>c</div><div>a</div><div>a</div><div>a</div><div>b</div><div>c</div><div>d</div></div></div><div><div>pi:</div><div><div>0</div><div>1</div><div>0</div></div></div><div><div>S:</div><div><div>a</div><div>a</div><div>b</div></div></div><div><div>q: 0</div></div></div>
<div>6. $k \leftarrow k + 1$</div>	<div>7. $q \leftarrow \pi [q - 1]$</div>	<div>8. $q \leftarrow \pi [q - 1]$</div>
<div><div>k: 3</div><div><div>T:</div><div><div>a</div><div>a</div><div>c</div><div>a</div><div>a</div><div>a</div><div>b</div><div>c</div><div>d</div></div></div><div><div>pi:</div><div><div>0</div><div>1</div><div>0</div></div></div><div><div>S:</div><div><div>a</div><div>a</div><div>b</div></div></div><div><div>q: 0</div></div></div>	<div><div>k: 3</div><div><div>T:</div><div><div>a</div><div>a</div><div>c</div><div>a</div><div>a</div><div>a</div><div>b</div><div>c</div><div>d</div></div></div><div><div>pi:</div><div><div>0</div><div>1</div><div>0</div></div></div><div><div>S:</div><div><div>a</div><div>a</div><div>b</div></div></div><div><div>q: 1</div></div></div>	<div><div>k: 4</div><div><div>T:</div><div><div>a</div><div>a</div><div>c</div><div>a</div><div>a</div><div>a</div><div>b</div><div>c</div><div>d</div></div></div><div><div>pi:</div><div><div>0</div><div>1</div><div>0</div></div></div><div><div>S:</div><div><div>a</div><div>a</div><div>b</div></div></div><div><div>q: 1</div></div></div>
<div>9. $k \leftarrow k + 1$</div>	<div>10. $q \leftarrow q + 1$</div>	<div>11. $k \leftarrow k + 1$</div>

<div><div>k: 4</div><div>T: <table><tr><td>a</td><td>a</td><td>c</td><td>a</td><td>a</td><td>a</td><td>b</td><td>c</td><td>d</td></tr></table></div><div>pi: <table><tr><td>0</td><td>1</td><td>0</td></tr></table></div><div>S: <table><tr><td>a</td><td>a</td><td>b</td></tr></table><div>q: 2</div></div></div>	a	a	c	a	a	a	b	c	d	0	1	0	a	a	b	<div><div>k: 5</div><div>T: <table><tr><td>a</td><td>a</td><td>c</td><td>a</td><td>a</td><td>a</td><td>b</td><td>c</td><td>d</td></tr></table></div><div>pi: <table><tr><td>0</td><td>1</td><td>0</td></tr></table></div><div>S: <table><tr><td>a</td><td>a</td><td>b</td></tr></table><div>q: 2</div></div></div>	a	a	c	a	a	a	b	c	d	0	1	0	a	a	b	<div><div>k: 5</div><div>T: <table><tr><td>a</td><td>a</td><td>c</td><td>a</td><td>a</td><td>a</td><td>b</td><td>c</td><td>d</td></tr></table></div><div>pi: <table><tr><td>0</td><td>1</td><td>0</td></tr></table></div><div>S: <table><tr><td>a</td><td>a</td><td>b</td></tr></table><div>q: 1</div></div></div>	a	a	c	a	a	a	b	c	d	0	1	0	a	a	b
a	a	c	a	a	a	b	c	d																																							
0	1	0																																													
a	a	b																																													
a	a	c	a	a	a	b	c	d																																							
0	1	0																																													
a	a	b																																													
a	a	c	a	a	a	b	c	d																																							
0	1	0																																													
a	a	b																																													
<div>12. $q \leftarrow q + 1$</div>	<div>13. $k \leftarrow k + 1$</div>	<div>14. $q \leftarrow \pi [q - 1]$</div>																																													
<div><div>k: 5</div><div>T: <table><tr><td>a</td><td>a</td><td>c</td><td>a</td><td>a</td><td>a</td><td>b</td><td>c</td><td>d</td></tr></table></div><div>pi: <table><tr><td>0</td><td>1</td><td>0</td></tr></table></div><div>S: <table><tr><td>a</td><td>a</td><td>b</td></tr></table><div>q: 2</div></div></div>	a	a	c	a	a	a	b	c	d	0	1	0	a	a	b	<div><div>k: 6</div><div>T: <table><tr><td>a</td><td>a</td><td>c</td><td>a</td><td>a</td><td>a</td><td>b</td><td>c</td><td>d</td></tr></table></div><div>pi: <table><tr><td>0</td><td>1</td><td>0</td></tr></table></div><div>S: <table><tr><td>a</td><td>a</td><td>b</td></tr></table><div>q: 2</div></div></div>	a	a	c	a	a	a	b	c	d	0	1	0	a	a	b	<div><div>k: 6</div><div>T: <table><tr><td>a</td><td>a</td><td>c</td><td>a</td><td>a</td><td>a</td><td>b</td><td>c</td><td>d</td></tr></table></div><div>pi: <table><tr><td>0</td><td>1</td><td>0</td></tr></table></div><div>S: <table><tr><td>a</td><td>a</td><td>b</td></tr></table><div>q: 3</div></div></div>	a	a	c	a	a	a	b	c	d	0	1	0	a	a	b
a	a	c	a	a	a	b	c	d																																							
0	1	0																																													
a	a	b																																													
a	a	c	a	a	a	b	c	d																																							
0	1	0																																													
a	a	b																																													
a	a	c	a	a	a	b	c	d																																							
0	1	0																																													
a	a	b																																													
<div>15. $q \leftarrow q + 1$</div>	<div>16. $k \leftarrow k + 1$</div>	<div>17. $q \leftarrow q + 1$</div>																																													



Очевидно, что сложность алгоритма Кнута-Морриса-Пратта: $O(m + n)$, где $m = \text{len}(S)$, $n = \text{len}(T)$.

§16. Упрощённый алгоритм Бойера-Мура

Работу некоторых алгоритмов поиска подстроки в строке, включая наивный алгоритм и алгоритм Бойера-Мура, можно описать как процесс *сдвига* искомой подстроки S вдоль строки T .

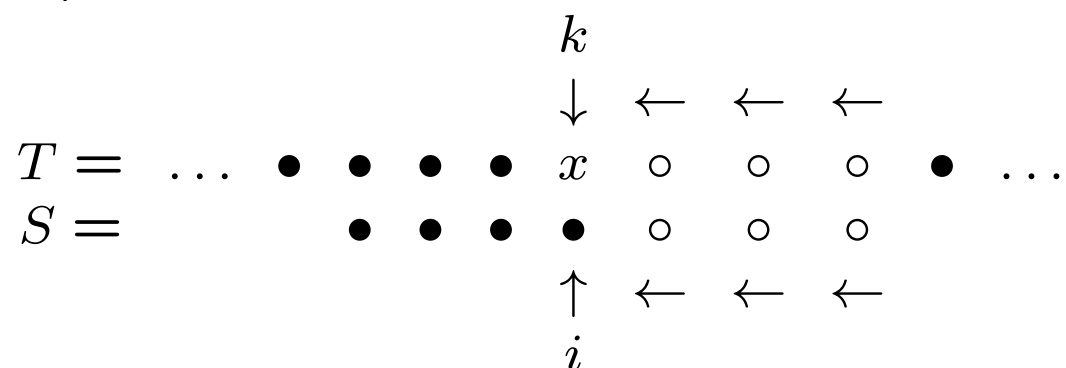
Пример. (наивный алгоритм)

	0	1	2	3	4	5	6	7	8	9	...
$T =$	a	b	b	a	d	a	b	a	c	b	...
$S =$	$\langle \mathbf{b} \rangle$	a	b	a	c						
	\rightarrow	\mathbf{b}	$\langle \mathbf{a} \rangle$	b	a	c					
		\rightarrow	\mathbf{b}	\mathbf{a}	$\langle \mathbf{b} \rangle$	a	c				
			\rightarrow	$\langle \mathbf{b} \rangle$	a	b	a	c			
				\cdot	\cdot	\cdot					

(Жирным шрифтом выделены символы, участвовавшие в сравнении; символы в скобках – места неудачных сравнений.)

В наивном алгоритме подстрока S всегда сдвигается на одну позицию вправо, и её сравнение с символами строки T выполняется слева направо.

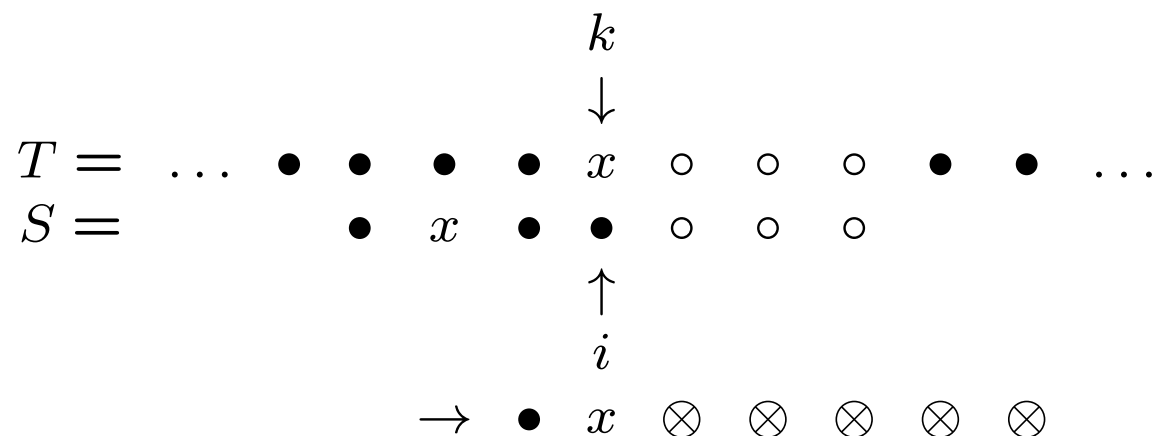
На каждом шаге алгоритма Бойера-Мура строка S сопоставляется с некоторым участком строки T и сканируется справа налево. В процессе сканирования текущий символ строки S сравнивается с текущим символом строки T .



(\circ — обозначение для символа, уже участвовавшего в сравнении, а \bullet — обозначение ещё не рассмотренного символа)

Эвристика — это приём в поиске решения задачи, позволяющий ограничить перебор. Алгоритм Бойера-Мура использует ряд *эвристик*, позволяющих сдвигать S вдоль T сразу на несколько позиций.

Эвристика стоп-символа. Встретив в строке T символ $x = T[k]$ такой, что $x \neq S[i]$, мы можем расположить строку S относительно строки T так, что последнее вхождение x в S окажется напротив $T[k]$.



(\otimes обозначает символ, гарантированно не равный x)

В частном случае, когда x вообще не входит в S , строку S можно расположить так, чтобы $S[0]$ оказался напротив $T[k+1]$.

Символ x будем называть *стоп-символом*.

Пример. ($T[3] = b$ – стоп-символ)

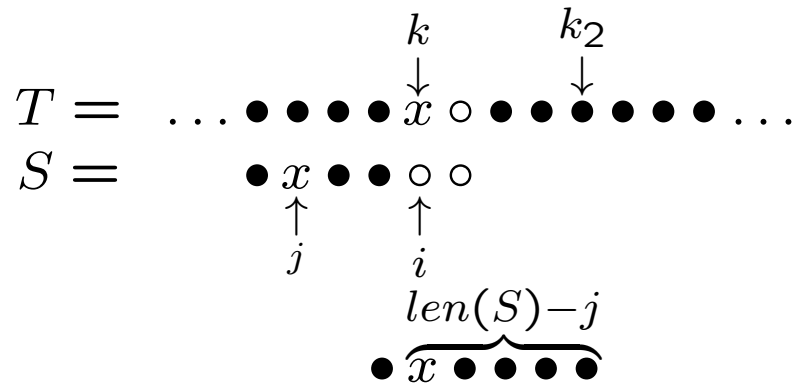
	0	1	2	3	4	5	6	7	8	9	...
$T =$	a	b	b	<u>b</u>	a	a	b	a	c	b	...
$S =$	b	b	a	$\langle c \rangle$	a						
		\rightarrow	b	<u>b</u>	a	c	a				
				.	.	.					

Пример. ($T[4] = d$ – стоп-символ)

	0	1	2	3	4	5	6	7	8	9	...
$T =$	a	b	b	a	<u>d</u>	a	b	a	c	b	...
$S =$	b	a	b	a	$\langle c \rangle$	a					
				\rightarrow	b	a	b	a	c	a	
					.	.	.				

Уже сейчас понятно, что в наилучшем случае алгоритм Бойера-Мура работает за время $O(n/m)$, где $n = \text{len}(T)$, $m = \text{len}(S)$.

Пусть $S[j] = x$ – самое правое вхождение стоп-символа $T[k] = x$ в строку S . Тогда для того чтобы расположить строку S относительно строки T так, чтобы $S[j]$ располагался напротив $T[k]$, нужно, чтобы последний символ строки S располагался напротив $T[k_2]$, где $k_2 = k + (\text{len}(S) - j) - 1$.



Отметим, что эвристика стоп-символа вовсе не всегда даёт положительный сдвиг:

Пример. ($T[2] = a$ – стоп-символ)

	0	1	2	3	4	5	6	7	8	9	...
$T =$	a	b	<u>a</u>	a	b	a	b	a	c	b	...
$S =$	c	a	$\langle b \rangle$	a	b						
	c	a	b	<u>a</u>	b	\leftarrow					

Для быстрого применения эвристики стоп-символа алгоритм Бойера-Мура использует *таблицу стоп-символов* δ_1 , которая отображает стоп-символы в величины смещения индекса k .

Алгоритм построения таблицы δ_1 :

```
1 Delta1(in  $S$ , in  $size$ ):  $\delta_1$ 
2    $\delta_1 \leftarrow$  новая последовательность размера  $size$ 
3    $a \leftarrow 0$ 
4   while  $a < size$ :
5        $\delta_1[a] \leftarrow len(S)$ 
6        $a \leftarrow a + 1$ 
7    $j \leftarrow 0$ 
8   while  $j < len(S)$ :
9        $\delta_1[S[j]] \leftarrow len(S) - j - 1$ 
10       $j \leftarrow j + 1$ 
```

Параметр $size$ задаёт размер алфавита. Мы будем считать алфавитом множество N_{size} .

Элементы таблицы стоп-символов, соответствующие символам, не входящим в S , равны $len(S)$.

Запишем упрощённый алгоритм Бойера-Мура, использующий только эвристику стоп-символа:

```
1 SimpleBMSubst(in  $S$ , in  $size$ , in  $T$ , out  $k$ )
2    $\delta_1 \leftarrow \text{Delta1}(S, size)$ 
3    $k \leftarrow \text{len}(S) - 1$ 
4   while  $k < \text{len}(T)$ :
5        $i \leftarrow \text{len}(S) - 1$ 
6       while  $T[k] = S[i]$ :
7           if  $i = 0$ :
8               return
9            $i \leftarrow i - 1$ 
10           $k \leftarrow k - 1$ 
11           $k \leftarrow k + \max(\delta_1[T[k]], \text{len}(S) - i)$ 
12   $k \leftarrow \text{len}(T)$ 
```

В настоящем алгоритме Бойера-Мура используется ещё и так называемая *эвристика совпавшего суффикса*, которую мы рассмотрим в следующем параграфе.

Пример. (работа алгоритма)

T = which ' finally ' halts ' ' ' at ' that ' point
S = at ' that

T = which ' finally ' halts ' ' ' at ' that ' point
S = at ' that

T = which ' finally ' halts ' ' ' at ' that ' point
S = at ' that

T = which ' finally ' halts ' ' ' at ' that ' point
S = at ' that

T = which ' finally ' halts ' ' ' at ' that ' point
S = at ' that

T = which ' finally ' halts ' ' ' at ' that ' point
S = at ' that

§17. Полный алгоритм Бойера-Мура

Определение префиксной функции « $\pi(i)$ равно длине $l(S[0:i])$ » можно переформулировать следующим образом:

$\pi(i)$ равно индексу элемента строки, левее которого расположен самый длинный собственный префикс строки S , равный суффиксу подстроки $S[0:i]$.

Аналогично можно дать определение *суффиксной функции*

$\sigma : \mathbb{N}_{len(S)-1} \longrightarrow \mathbb{N}_{len(S)-1}$:

$\sigma(i)$ равно индексу элемента строки, правее которого расположен самый длинный собственный суффикс строки S , равный префиксу подстроки $S[i:len(S)-1]$.

Пример.

$S =$	0 a	1 b	2 r	3 a	4 c	5 a	6 d	7 a	8 b	9 r	10 a
$\sigma =$	6	7	8	9	10	9	10	9	10	10	10

Алгоритм построения суффиксной функции аналогичен алгоритму Prefix и записывается как:

```
1 Suffix (in S):  $\sigma$ 
2    $\sigma \leftarrow$  новая последовательность нат. чисел длины  $len(S)$ 
3    $\sigma[len(S) - 1] \leftarrow t \leftarrow len(S) - 1$ 
4    $i \leftarrow len(S) - 2$ 
5   while  $i \geq 0$ :
6       while  $t < len(S) - 1$  and  $S[t] \neq S[i]$ :
7            $t \leftarrow \sigma[t + 1]$ 
8       if  $S[t] = S[i]$ :
9            $t \leftarrow t - 1$ 
10       $\sigma[i] \leftarrow t$ 
11       $i \leftarrow i - 1$ 
```


Подстрока

$$occ = S[j : j + m - 1]$$

называется *правдоподобным вхождением суффикса*

$$suf = S[len(S) - m : len(S) - 1] \text{ в } S,$$

если $occ = suf$, $j > 0$ и $S[j - 1] \neq S[len(S) - m - 1]$.

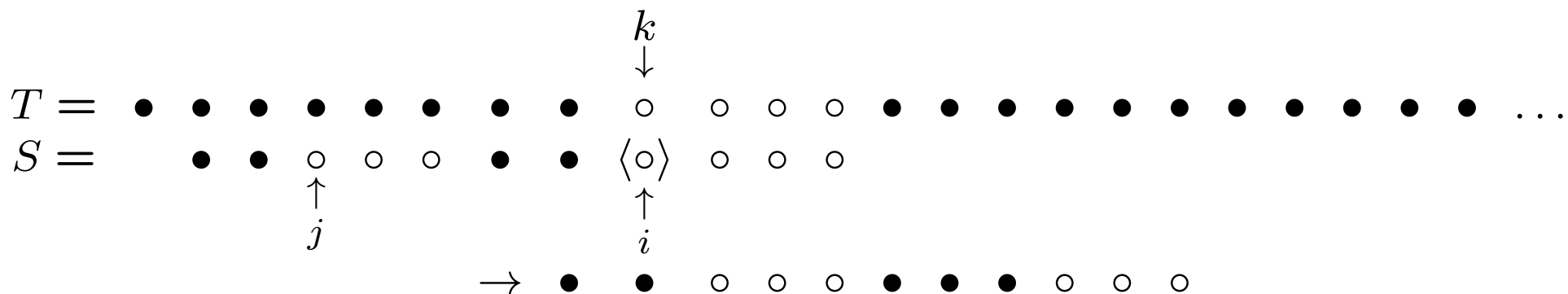
Пример. *ababab* – нет правдоподобных вхождений суффиксов.

Пример. *abc* *ab* *dab* – одно правдоподобное вхождение суффикса *ab*.

Пример. *c* *ab* *d* *ab* *ab* – несколько правдоподобных вхождений суффикса *ab*.

Эвристика совпавшего суффикса. Если при сопоставлении строки S с очередным участком строки T суффикс $S[i : \text{len}(S) - 1]$ совпал с подстрокой $T[k : k + (\text{len}(S) - i) - 1]$, то возможны два случая:

Случай 1. $S[j]$ – начало самого правого правдоподобного вхождения совпавшего суффикса в S ;



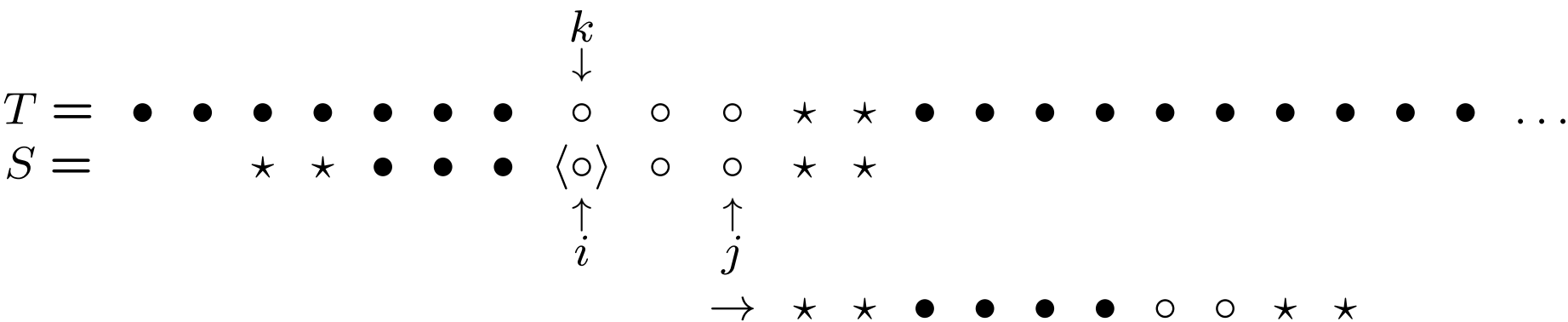
$$k \leftarrow k - j + \text{len}(S)$$

Очевидно, k увеличивается не менее, чем на $\text{len}(S) - i$.

Пример. (случай 1: $k \leftarrow 2 - 1 + 5 = 6$)

	0	1	2	3	4	5	6	7	8	9	\dots
$T =$	a	b	a	a	b	a	b	a	c	b	\dots
$S =$	c	a	$\langle b \rangle$	a	b						
		\rightarrow	c	a	b	a	b				
			\cdot	\cdot	\cdot						

Случай 2. не существует правдоподобного вхождения совпавшего суффикса в строке S , но имеется собственный суффикс $S[j + 1 : len(S) - 1]$ совпавшего суффикса, равный префиксу строки S :



$$k \leftarrow k - i + j + len(S)$$

Очевидно, k увеличивается не менее, чем на $len(S) - i$.

Пример. (случай 2: $k \leftarrow 4 - 4 + 5 + 9 = 14$)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$T =$	a	b	c	a	a	c	a	b	c	a	b	c	a	b	c	...
$S =$	a	b	c	a	$\langle b \rangle$	c	a	b	c							
						\rightarrow	a	b	c	a	b	c	a	b	c	
							.	.	.							

Для быстрого применения эвристики совпавшего суффикса алгоритм Бойера-Мура использует таблицу δ_2 , которая строится следующим алгоритмом:

```

1 Delta2 (in S):  $\delta_2$ 
2    $\delta_2 \leftarrow$  новая последовательность размера  $len(S)$ 
3    $\sigma \leftarrow \text{Suffix}(S)$ 
4    $i \leftarrow 0$ 
5    $t \leftarrow \sigma[0]$ 
6   while  $i < len(S)$ :
7       while  $t < i$ :
8            $t \leftarrow \sigma[t + 1]$ 
9            $\delta_2[i] \leftarrow -i + t + len(S)$           — случай 2
10           $i \leftarrow i + 1$ 
11   $i \leftarrow 0$ 
12  while  $i < len(S) - 1$ :
13       $t \leftarrow i$ 
14      while  $t < len(S) - 1$ :
15           $t \leftarrow \sigma[t + 1]$ 
16          if  $S[i] \neq S[t]$ :
17               $\delta_2[t] \leftarrow -(i + 1) + len(S)$   — случай 1
18           $i \leftarrow i + 1$ 

```

Замечание. Алгоритм Delta2 гарантирует, что для любого i от 0 до $len(S) - 1$ справедливо, что $\delta_2[i] \geq len(S) - i$.

Пример. (вычисление δ_2 для строки «abscabdab»)

Суффиксная функция (строка 3):

$S =$	0 a	1 b	2 c	3 a	4 b	5 d	6 a	7 b
$\sigma =$	5	6	7	5	6	7	7	7

После первого цикла (строки 6..10):

$S =$	0 a	1 b	2 c	3 a	4 b	5 d	6 a	7 b
$\delta_2 =$	13	12	11	10	9	8	9	8

После второго цикла (строки 12..18):

$S =$	0 a	1 b	2 c	3 a	4 b	5 d	6 a	7 b
$\delta_2 =$	13	12	11	10	9	5	9	1

Полный алгоритм Бойера-Мура отличается от упрощённого использованием таблицы δ_2 :

```

1 BMSubst( in  $S$  , in  $size$  , in  $T$  , out  $k$  )
2      $\delta_1 \leftarrow \text{Delta1}(S, size)$ 
3      $\delta_2 \leftarrow \text{Delta2}(S)$ 
4      $k \leftarrow \text{len}(S) - 1$ 
5     while  $k < \text{len}(T)$  :
6          $i \leftarrow \text{len}(S) - 1$ 
7         while  $T[k] = S[i]$  :
8             if  $i = 0$  :
9                 return
10                 $i \leftarrow i - 1$ 
11                 $k \leftarrow k - 1$ 
12                 $k \leftarrow k + \max(\delta_1[T[k]], \delta_2[i])$ 
13     $k \leftarrow \text{len}(T)$ 

```

В строке 12 строка S сдвигается относительно строки T не менее, чем на одну позицию вправо, так как $\delta_2[i] \geq \text{len}(S) - i$.

Пример. (работа алгоритма)

T = which ' finally ' halts ' ' ' at ' that ' point

S = at ' that

----- эвристика стоп-символа -----

T = which ' finally ' halts ' ' ' at ' that ' point

S = at ' that

----- эвристика стоп-символа -----

T = which ' finally ' halts ' ' ' at ' that ' point

S = at ' that

----- эвристика стоп-символа -----

T = which ' finally ' halts ' ' ' at ' that ' point

S = at ' that

----- эвристика совпавшего суффикса -----

T = which ' finally ' halts ' ' ' at ' that ' point

S = at ' that

§18. Дерево отрезков

Пусть дана последовательность значений $\{v_i \in V\}_0^{n-1}$ и некоторая бинарная операция $+: V \times V \longrightarrow V$, обладающая свойством ассоциативности: $(x + y) + z = x + (y + z)$.

Требуется уметь многократно вычислять значения

$$\sum_{i=l}^r v_i = v_l + v_{l+1} + \dots + v_r$$

на разных интервалах от l до r .

При этом последовательность не является фиксированной, т.е. время от времени значения её элементов могут изменяться.

Пример. Дана последовательность целых чисел. Требуется вычислять суммы чисел на разных интервалах или требуется находить максимальное число на разных интервалах.

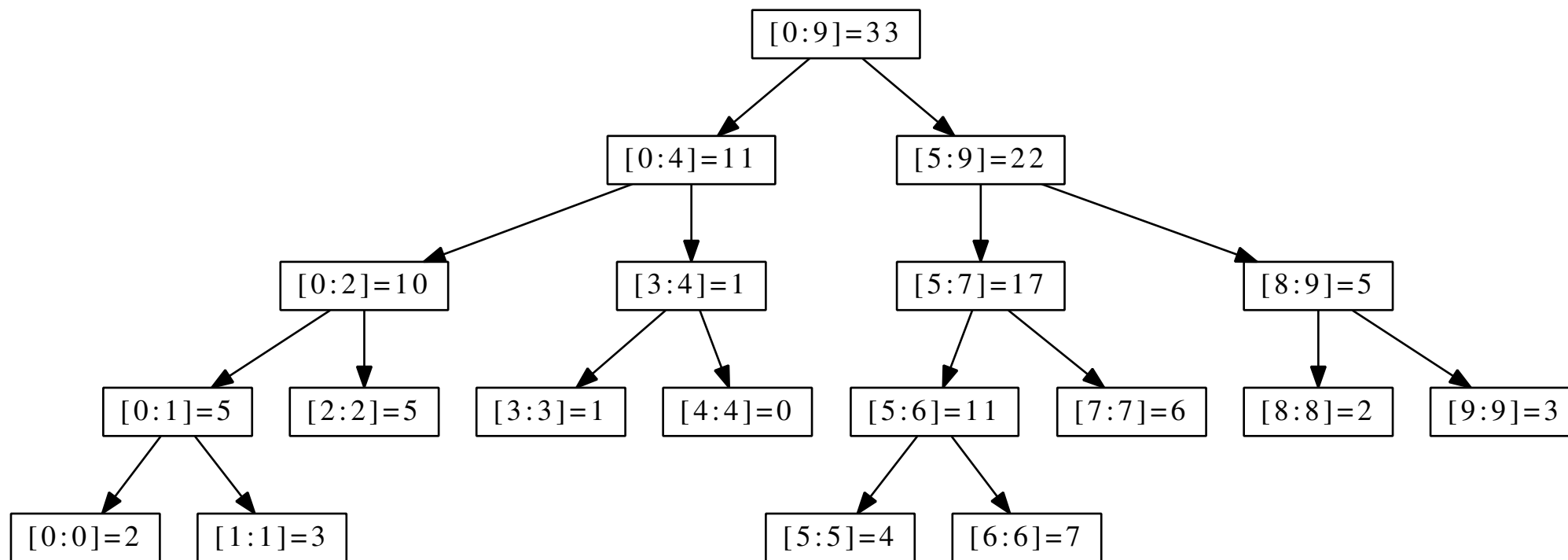
Мы будем называть *отрезком* от a до b и обозначать как $[a : b]$ значение $\sum_{i=a}^b v_i$, вычисленное на последовательности $\{v_i\}_0^{n-1}$.

Дерево отрезков – это массив T , построенный по следующим правилам:

1. в $T[0]$ содержится отрезок $[0 : n - 1]$, называемый *корнем* дерева;
2. пусть в $T[i]$ содержится отрезок $[a : b]$, причём $a \neq b$;
тогда, положив $m = \left\lfloor \frac{a + b}{2} \right\rfloor$, имеем:
в $T[2i + 1]$ находится $[a : m]$ – левый дочерний отрезок для $T[i]$,
в $T[2i + 2]$ находится $[m + 1 : b]$ – правый дочерний для $T[i]$;
3. значения остальных элементов массива T не определены и не используются.

Для представления последовательности длины n достаточно массива T размером $4n$ элементов.

Пример. (дерево отрезков для последовательности натуральных чисел $\langle 2, 3, 5, 1, 0, 4, 7, 6, 2, 3 \rangle$ с операцией сложения)



```

1 SegmentTree_Query(in  $T$ , in  $n$ , in  $l$ , in  $r$ ):  $v$ 
2      $v \leftarrow \text{query}(T, l, r, 0, 0, n-1)$ 

4 query(in  $T$ , in  $l$ , in  $r$ , in  $i$ , in  $a$ , in  $b$ ):  $v$ 
5     if  $l = a$  and  $r = b$ :
6          $v \leftarrow T[i]$ 
7     else :
8          $m \leftarrow \left\lfloor \frac{a+b}{2} \right\rfloor$ 
9         if  $r \leq m$ :
10              $v \leftarrow \text{query}(T, l, r, 2i+1, a, m)$ 
11         else if  $l > m$ :
12              $v \leftarrow \text{query}(T, l, r, 2i+2, m+1, b)$ 
13         else :
14              $v \leftarrow \text{query}(T, l, m, 2i+1, a, m) +$ 
15                  $\text{query}(T, m+1, r, 2i+2, m+1, b)$ 

```

```

1 SegmentTree_Build(in  $\{v_i\}_0^{n-1}$ , out  $T$ )
2      $T \leftarrow$  новый массив размером  $4n$ 
3     build( $\{v_i\}_0^{n-1}$ , 0, 0,  $n-1$ ,  $T$ )

5 build(in  $\{v_i\}_0^{n-1}$ , in  $i$ , in  $a$ , in  $b$ , in/out  $T$ )
6     if  $a = b$ :
7          $T[i] \leftarrow v_a$ 
8     else :
9          $m \leftarrow \left\lfloor \frac{a+b}{2} \right\rfloor$ 
10        build( $\{v_i\}_0^{n-1}$ ,  $2i+1$ ,  $a$ ,  $m$ ,  $T$ )
11        build( $\{v_i\}_0^{n-1}$ ,  $2i+2$ ,  $m+1$ ,  $b$ ,  $T$ )
12     $T[i] \leftarrow T[2i+1] + T[2i+2]$ 

```

```

1 SegmentTree_Update(in  $j$ , in  $v$ , in  $n$ , in/out  $T$ )
2     update( $j$ ,  $v$ , 0, 0,  $n-1$ ,  $T$ )

4 update(in  $j$ , in  $v$ , in  $i$ , in  $a$ , in  $b$ , in/out  $T$ )
5     if  $a = b$ :
6          $T[i] \leftarrow v$ 
7     else :
8          $m \leftarrow \left\lfloor \frac{a+b}{2} \right\rfloor$ 
9         if  $j \leq m$ :
10             update( $j$ ,  $v$ ,  $2i+1$ ,  $a$ ,  $m$ ,  $T$ )
11         else :
12             update( $j$ ,  $v$ ,  $2i+2$ ,  $m+1$ ,  $b$ ,  $T$ )
13      $T[i] \leftarrow T[2i+1] + T[2i+2]$ 

```

§19. Дерево Фенвика

Изменим постановку задачи посчёта сумм на отрезках последовательности $\{v_i \in V\}_0^{n-1}$ следующим образом: пусть операция $+$: $V \times V \longrightarrow V$ не только ассоциативна, но и коммутативна и, более того, обратима, т.е. существует операция $-$: $V \times V \longrightarrow V$ такая, то $(a + b = c) \Rightarrow (b = c - a)$.

В этом случае можно применить более компактную структуру данных, чем дерево отрезков. Она называется *бинарным индексированным деревом* или *деревом Фенвика*.

Дерево Фенвика представляет собой массив T размера n такой, что

$$T[i] = \sum_{j=f(i)}^i v_j.$$

Здесь $f(i) = i - 2^{h(i)} + 1$, где $h(i)$ – количество единиц в конце двоичной записи числа i .

Можно показать, что $f(i) = i \& (i + 1)$.

i	$h(i)$	$f(i) = i - 2^{h(i)} + 1$	V	T
$0 = 0000$	0	0	5 → 5 → → → 	5
$1 = 0001$	1	0	2 → 7 → → → 	7
$2 = 0010$	0	2	3 → 3 → → → 	3
$3 = 0011$	2	0	7 → 17 → → → 	17
$4 = 0100$	0	4	0 → 0 → → → 	0
$5 = 0101$	1	4	4 → 4 → → → 	4
$6 = 0110$	0	6	2 → 2 → → → 	2
$7 = 0111$	3	0	5 → 28 → → → 	28
$8 = 1000$	0	8	1 → 1 → → → 	1
$9 = 1001$	1	8	6 → 7 → → → 	7
$10 = 1010$	0	10	9 → 9 → → → 	9
$11 = 1011$	2	8	3 → 19 → → → 	19

Запрос к дереву Фенвика – $O(\lg n)$:

```
1 FenwickTree_Query(in  $T$ , in  $l$ , in  $r$ ):  $v$ 
2      $v \leftarrow \text{query}(T, r) - \text{query}(T, l - 1)$ 

4 query(in  $T$ , in  $i$ ):  $v$ 
5      $v \leftarrow 0$ 
6     while  $i \geq 0$ :
7          $v \leftarrow v + T[i]$ 
8          $i \leftarrow (i \& (i + 1)) - 1$ 
```

Обновление дерева Фенвика – $O(\lg n)$:

```
1 FenwickTree_Update(in  $i$ , in  $\delta$ , in  $n$ , in/out  $T$ )
2     while  $i < n$ :
3          $T[i] \leftarrow T[i] + \delta$ 
4          $i \leftarrow i | (i + 1)$ 
```


Построение дерева Фенвика – $O(n)$:

```
1 FenwickTree_Build(in  $\{v_i\}_0^{n-1}$ , out  $T$ )
2      $T \leftarrow$  новый массив размера  $n$ 
3      $r \leftarrow$  минимальная степень 2, не меньшая, чем  $n$ 
4     build( $\{v_i\}_0^{n-1}$ , 0,  $r - 1$ ,  $T$ )

6 build(in  $\{v_i\}_0^{n-1}$ , in  $l$ , in  $r$ , in/out  $T$ ):  $sum$ 
7      $sum \leftarrow 0$ 
8      $bound \leftarrow \min(r, n)$ 
9     while  $l < bound$ :
10          $m \leftarrow \left\lfloor \frac{l + r}{2} \right\rfloor$ 
11          $sum \leftarrow sum + \text{build}(\{v_i\}_0^{n-1}, l, m, T)$ 
12          $l \leftarrow m + 1$ 
13     if  $r < n$ :
14          $sum \leftarrow sum + v_r$ 
15          $T[r] \leftarrow sum$ 
```

§20. Разреженная таблица

Пусть дана последовательность значений $\{v_i \in V\}_0^{n-1}$ и некоторая бинарная операция $\min : V \times V \longrightarrow V$, обладающая двумя свойствами:

1. ассоциативность:

$$\min(\min(x, y), z) = \min(x, \min(y, z));$$

2. идемпотентность:

$$\min(x, x) = x.$$

Требуется уметь многократно вычислять значения

$$\min_{i=l}^r v_i = \min\left(v_l, \min\left(v_{l+1}, \dots, \min(v_{r-1}, v_r) \dots\right)\right)$$

на разных интервалах от l до r .

Разреженная таблица – это матрица ST размером $n \times (\lfloor \log_2 n \rfloor + 1)$ такая, что $ST[i, j] = \min_{k=i}^{i+2^j-1} v_k$.

Здесь $i = \overline{0, n-1}$, $j = \overline{0, \lfloor \log_2 n \rfloor}$, причём $i + 2^j - 1 < n$.

Другими словами, j -тый столбец таблицы хранит минимумы всех отрезков, длины которых равны 2^j . Причём в конце каждого столбца имеется $2^j - 1$ незаполненных элементов.

Построение разреженной таблицы выполняется следующим образом:

1. заполнение нулевого столбца: $ST[i, 0] = v_i$.
2. заполнение каждого следующего (j -того) столбца:
 $ST[i, j] = \min(ST[i, j-1], ST[i + 2^{j-1}, j-1])$.

Запросы к разреженной таблице выполняются за одно действие:

$$\min_{i=l}^r v_i = \min(ST[l, j], ST[r - 2^j + 1, j]), \text{ где } j = \lfloor \log_2 (r - l + 1) \rfloor.$$

Для эффективной работы разреженной таблицы требуется уметь быстро считать логарифмы. Удобнее всего заранее заполнить таблицу логарифмов для всех возможных длин отрезков:

```
1 ComputeLogarithms(in  $m$ , out  $lg$ )
2    $lg \leftarrow$  новый массив целых чисел размера  $2^m$ 
3    $i \leftarrow 1$ ,  $j \leftarrow 0$ 
4   while  $i < m$ :
5       while  $j < 2^i$ :
6            $lg[j] \leftarrow i - 1$ 
7            $j \leftarrow j + 1$ 
8    $i \leftarrow i + 1$ 
```

Запрос к разреженной таблице – $O(1)$:

```
1 SparseTable_Query(in  $ST$ , in  $l$ , in  $r$ , in  $lg$ ):  $v$ 
2    $j \leftarrow lg[r - l + 1]$ 
3    $v \leftarrow \min(ST[l, j], ST[r - 2^j + 1, j])$ 
```

Построение разреженной таблицы – $O(n \lg n)$:

```
1 SparseTable_Build(in  $\{v_i\}_0^{n-1}$ , in  $lg$ , out  $ST$ )
2      $m \leftarrow lg[n] + 1$ 
3      $ST \leftarrow$  новая матрица размером  $n \times m$ 

5      $i \leftarrow 0$ 
6     while  $i < n$  :
7          $ST[i, 0] \leftarrow v_i$ 
8          $i \leftarrow i + 1$ 

10     $j \leftarrow 1$ 
11    while  $j < m$  :
12         $i \leftarrow 0$ 
13        while  $i \leq n - 2^j$  :
14             $ST[i, j] = \min(ST[i, j - 1], ST[i + 2^{j-1}, j - 1])$ 
15             $i \leftarrow i + 1$ 
16         $j \leftarrow j + 1$ 
```

§21. Алгоритм Кадана

Пусть дана последовательность целых чисел $\{a_i\}_0^{n-1}$. Требуется найти такие индексы l и r , что сумма $\sum_{i=l}^r a_i$ — максимальна.

```
1 Kadane(in  $\{a_i\}_0^{n-1}$ , out  $l$ , out  $r$ )
2      $l \leftarrow 0$ ,  $r \leftarrow 0$ ,  $maxsum \leftarrow a_0$ 
3      $start \leftarrow 0$ ,  $sum \leftarrow 0$ 
4      $i \leftarrow 0$ 
5     while  $i < n$ :
6          $sum \leftarrow sum + a_i$ 
7         if  $sum > maxsum$ :
8              $maxsum \leftarrow sum$ 
9              $l \leftarrow start$ 
10             $r \leftarrow i$ 
11         $i \leftarrow i + 1$ 
12        if  $sum < 0$ :
13             $sum \leftarrow 0$ 
14             $start \leftarrow i$ 
```