Структуры и алгоритмы обработки данных

Лекция 1

1. Алгоритмы
2. Сложность алгоритмов
3. Асимптотические оценки

- **Алгоритм** – это точное предписание, определяющее вычислительный процесс, ведущий от варьируемых начальных данных к искомому результату.

Для этого надо уметь анализировать алгоритмы и сравнивать их между собой по тем или иным критериям.

Наиболее часто используются **следующие критерии**:

- точность решения задачи
- затраты машинного времени
- затраты оперативной памяти
- простота реализации алгоритма

Анализ алгоритмов:

- анализ корректности,
- анализ точности вычислений,
- анализ потребляемых алгоритмом ресурсов

и т.п.

Анализ сложности алгоритма - процесс исследования *эффективности* алгоритма.

Эффективность алгоритма характеризуется двумя основными параметрами:

- временем выполнения алгоритма,
- объемом потребной памяти (оперативной или внешней).

Время выполнения *программы,*

реализующей алгоритм, может быть *измерено.*

Результаты измерения будут зависеть от нескольких факторов, таких, например, как:

- быстродействие конкретного компьютера (процессора);
- тщательности разработки и степени оптимизации программной реализации алгоритма;
- типа системы программирования (компилятора и исполняющей системы);
- особенностей хронометрирования реального времени исполнения программы.

Общая функциональная зависимость времени счета от размера задачи (как быстро время счета растет с ростом размера задачи).

Размер задачи = *размер входных данных*.

Например, при обработке одномерного массива $a[1..n]$ *размер задачи* = n.

Базовые операции (например, сравнения и перестановки элементов).

Функциональная зависимость $T(n)$ времени счета от размера задачи выражаются с точностью до мультипликативной константы.

Считаем, что все «мешающие» факторы суммарно влияют только на величину константы, но не на вид функциональной зависимости $T(n)$.

$$T(n) = C\, f\,(n)$$

Приближенные значения некоторых функций, встречающихся при анализе алгоритмов

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9	$1.05 \cdot 10^{301}$	$4 \cdot 10^{2567}$
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Изменение размера задачи, решаемой за один час, при увеличении быстродействия компьютера

Функция $T(n)$	Размер задачи при решении на …			
	эталонном компьютере	компьютере в **10** раз более быстром	компьютере в **100** раз более быстром	компьютере в **1000** раз более быстром
n	N_2	$10\ N_2$	$100\ N_2$	$100\ N_2$
$n \log_2 n$	N_3	$8.7\ N_3$		
n^2	N_4	$3.16\ N_4$	$10\ N_4$	$31.6\ N_4$
n^3	N_5	$2.15\ N_5$	$4.64\ N_5$	$10\ N_5$
2^n	N_6	$N_6 + 3.3$	$N_6 + 6.64$	$N_6 + 9.97$
$n!$	N_7			

Изменение времени счета при изменении размера задачи

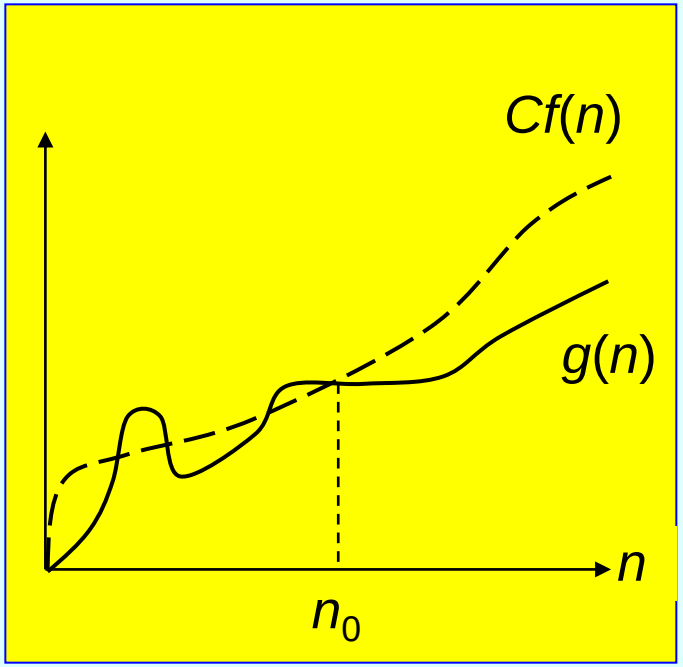
Пусть требуется решить задачу в два раза большего размера. Как увеличится при этом время решения?

- Для функции $\log_2 n$ увеличение составит всего одну единицу времени.
- Для функции n значение увеличится в 2 раза
- Для функции $n \log_2 n$ – чуть больше, чем в 2 раза
- Для функции n^2 – в 4 раза
- Для функции n^3 – в 8 раз
- Для функции 2^n время возрастет в квадрате, поскольку если $T(n) = 2^n$, то $T(2n) = 2^{2n} = (2^n)^2 = T^2(n)$.

Асимптотические оценки

Обозначение $O(f(n))$

Для указания *множества функций*, которые *не более* чем в постоянное число раз превосходят $f(n)$ при достаточно большом n, используется обозначение *О большое*: $O(f(n))$.



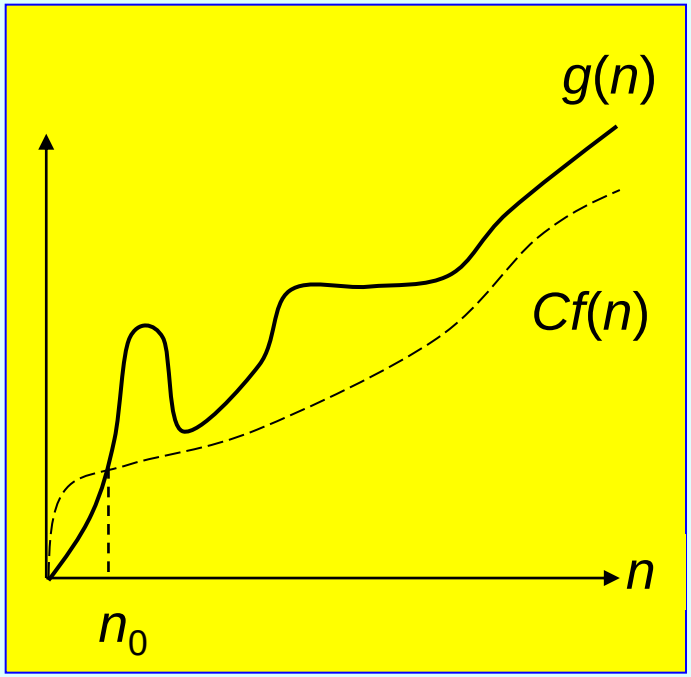
Запись $g(n) \in O(f(n))$ означает, что существуют вещественная константа $C > 0$ и натуральная константа n_0, для которых $g(n) \le C\, f(n)$ при всех $n \ge n_0$.

Асимптотические оценки

Обозначение $\Omega(f(n))$

Для указания *множества функций*, которые *не менее* чем в постоянное число раз превосходят $f(n)$ при достаточно большом n, используется обозначение

Омега большая: $\Omega(f(n))$.



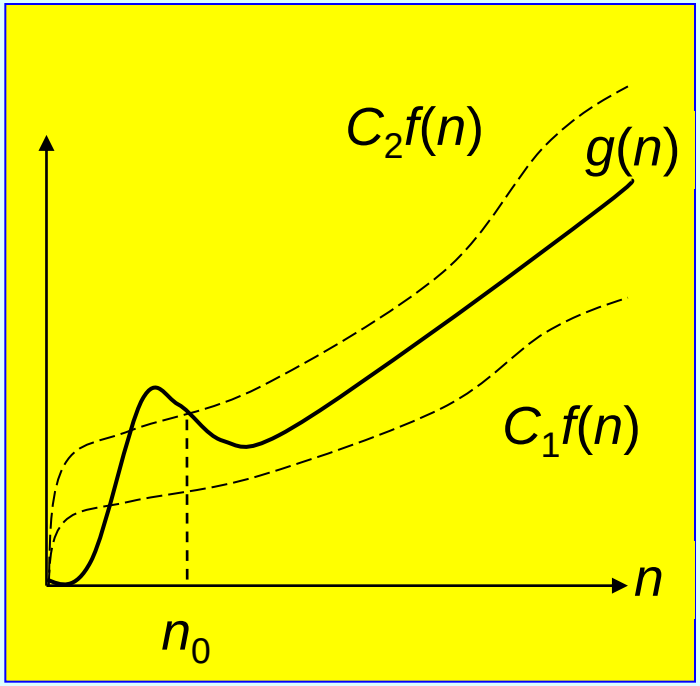
Запись $g(n) \in \Omega(f(n))$ означает, что существуют вещественная константа $C > 0$ и натуральная константа n_0, для которых $g(n) \geq C f(n)$ при всех $n \geq n_0$.

Асимптотические оценки

Обозначение $\Theta(f(n))$

Для указания *множества функций того же порядка*, что и $f(n)$ при достаточно большом n, используется обозначение

Тэта большая: $\Theta(f(n))$.



Запись $g(n) \in \Theta(f(n))$ означает, что существуют вещественные константы $C_1 > 0$ и $C_2 > 0$ и натуральная константа n_0, для которых $C_1 f(n) \le g(n) \le C_2 f(n)$ при всех $n \ge n0$.

Простые примеры

Пример 1.

$50n + 13 \in O(n^2)$.

Действительно, т. к.
$50n + 13 \le 50n + 50 = 50(n + 1) \le 50n^2$ при $n \ge 2$,

то можно положить $C = 50$ и тогда $50n + 13 \le Cn^2$ при $n \ge n_0$ и $n_0 = 2$.

Пример 2.

$50n + 13 \in O(n)$.

Действительно, т. к. $50n + 13 \le 100n$, то можно положить $C = 100$ и $n_0 = 1$.

Простые примеры (продолжение)

Пример 3.

$n^3 \in \Omega(n^2)$.

Действительно, если положить $C = 1$ и $n_0 = 1$, то $n^3 \geq Cn^2$.

Пример 4.

$n^3 \in \Omega(n^3)$.

Действительно, достаточно взять $0 < C < 1$ и $n_0 = 1$, чтобы выполнялось $n^3 \geq Cn^3$.

Простые примеры (продолжение)

Пример 5.

$$\frac{n(n-1)}{2} \in \Theta(n^2)$$

Требуется показать, что найдутся такие C_1 и C_2, что при $n \geq n_0$ будет выполнено

$$C_1 n^2 \leq \frac{n(n-1)}{2} \leq C_2 n^2$$

Действительно, во-первых,

$$\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \geq \frac{n^2}{2} - \left(\frac{n}{2}\right)^2 = \frac{n^2}{2} - \frac{n^2}{4} = \frac{n^2}{4} \geq C_1 n^2$$

при $C_1 = \frac{1}{4}$ и $n_0 = 2$.

Пример 5 (продолжение)

Во-вторых,

$$\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \le \frac{n^2}{2} \le C_2 n^2$$

при $C_1 = \frac{1}{2}$ и $n_0 = 2$.

$$\Rightarrow \quad C_1 n^2 \le \frac{n(n-1)}{2} \le C_2 n^2$$

Сложность	Описание	Форма повторения
$O(1)$	Алгоритм константной сложности. Не зависит от количества элементов данных. Выполняется за постоянную единицу времени	Присваивание. Простое выражение. Алгоритм без цикла и рекурсии
$O(n)$	Линейный. Сложность алгоритма пропорциональна размеру списка	Нахождение максимального элемента в массиве из *n* элементов

$O(n^2)$	Квадратический	`for (i=0; i<N; i++)`
		`for (j=0; j<N; j++)`
		Простые алгоритмы сортировки

$O(n^3)$	Кубический	
$O(n \log_2 n)$ $O(\log_2 n)$	Логарифмический	Возникает при неоднократном подразделении данных на подсписки. Бинарный поиск
$O(a^n)$	*NP* - сложные (неполиномиальные). Частный случай - экспоненциальная сложность $O(2^n)$	Используются при неоднократном поиске дерева решений

Способы описания алгоритмов

- **Псевдоко́д** — компактный (зачастую неформальный) язык описания алгоритмов, использующий ключевые слова императивных языков программирования, но опускающий несущественные подробности и специфический синтаксис.
- **Блок-схема** — распространенный тип схем (*графических моделей*), описывающих алгоритмы или процессы, в которых отдельные шаги изображаются в виде блоков различной формы, соединенных между собой линиями

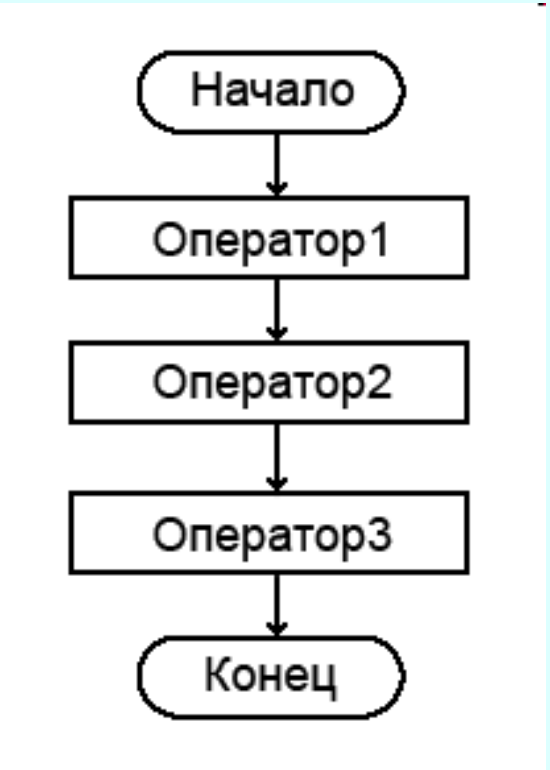
Базовые управляющие структуры псевдокода

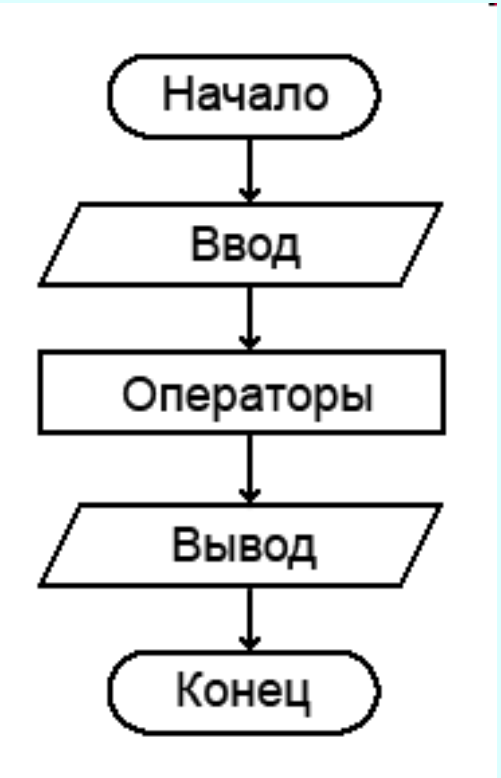
Название структуры	Псевдокод
присваивание, ввод, вывод	переменная = 0, ввод (переменная), вывод (переменная)
ветвление	если условие то (серия1 иначе серия 2)
цикл ПОКА	пока условие нц серия кц

Пример программы «Hello, world»

```
алг HELLOWORLD
нач
вывод ('Hello,World')
кон алг HELLOWORLD
```

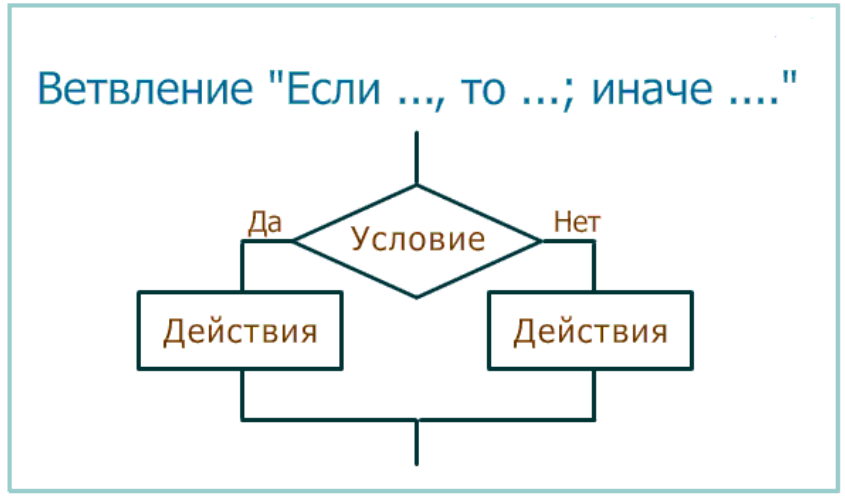
Линейный алгоритм

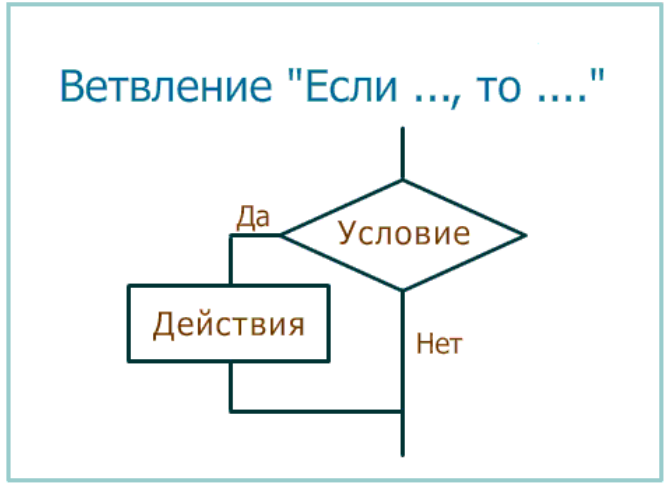




Ветвление

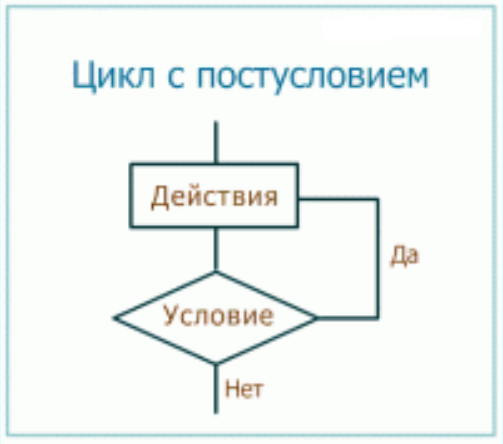

Ветвление "Если ..., то ...;
еще если ..., то ...; ...;
иначе ...."
Условие
Да
Действия
Нет
Условие
Да
Действия
Нет
Условие
Да
Действия
Нет
Действия

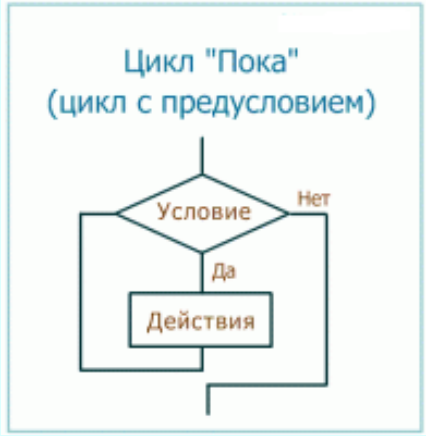

Ветвление "Если ..., то ...; иначе ...."
Да
Условие
Нет
Действия
Действия


Ветвление "Если ..., то ...."
Да
Условие
Действия
Нет

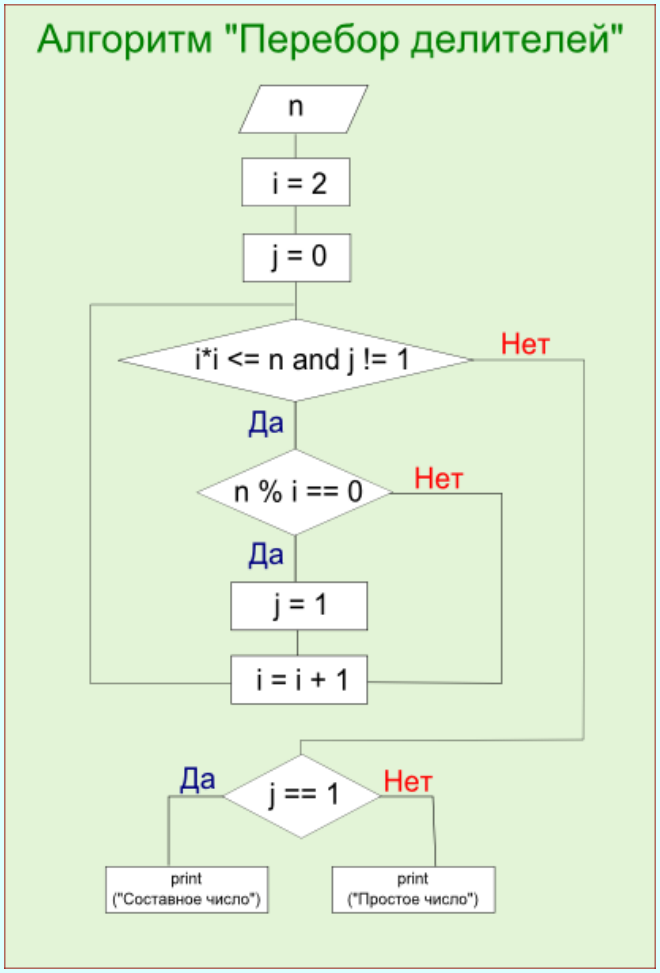
Циклы


Арифметический цикл
от, до, шаг
Действия


Цикл с постусловием
Действия
Да
Условие
Нет


Цикл "Пока"
(цикл с предусловием)
Условие
Нет
Да
Действия

Блок-схема алгоритма "Перебор делителей" (определение простоты числа)



Сортировка

1. **Постановка задачи**
2. **Простые алгоритмы сортировки**
   - **Сортировка выбором**
   - **Сортировка обменами**
   - **Сортировка вставками**

1. Постановка задачи сортировки (упорядочения)

Дан массив a: **array** [1..n] **of** Ordinal

Отрезок (сегмент) массива a[p..q] ***упорядочен***:

$$Sort\ (a, p, q) \equiv (\forall\ i: p \le i < q: a[i] \le a[i+1])$$

По возрастанию < (*строго* <, *не строго* ≤), *по неубыванию* ≤

Предусловие: a[1..n] = a0[1..n]

Постусловие: Sort (a, 1, n) &

& *Перестановка*(a[1..n], a0[1..n]),

где *Перестановка*(a[1..n], a0[1..n]) ≡

$$\equiv (\forall\ i: 1 \le i \le n: (\mathbf{N}\ j: 1 \le j \le n: a[\,i\,] = a0[\,j\,]\,) =$$

$$= (\mathbf{N}\ j: 1 \le j \le n: a[\,i\,] = a\,[\,j\,]\,)\,)$$

2. Простые алгоритмы сортировки.

2.1. Сортировка выбором

Идея. Пример. Список (удаление и вставка)

упОрядочИвание	авдеИинупОряоч
аупОрядочИвние	авдеИинОупряоч
авупОрядочИние	авдеИинОоупряч
авдупОряочИние	авдеИинОопуряч
авдеупОряочИни	авдеИинОопруяч
авдеИупОряочни	авдеИинОопруяч
авдеИйупОряочн	авдеИинОопручя

Устойчивость сортировки – сохранение относительного порядка элементов *с равными ключами*

Сортировка выбором

Пример.
Массив (обмен элементов).

Сортировка выбором

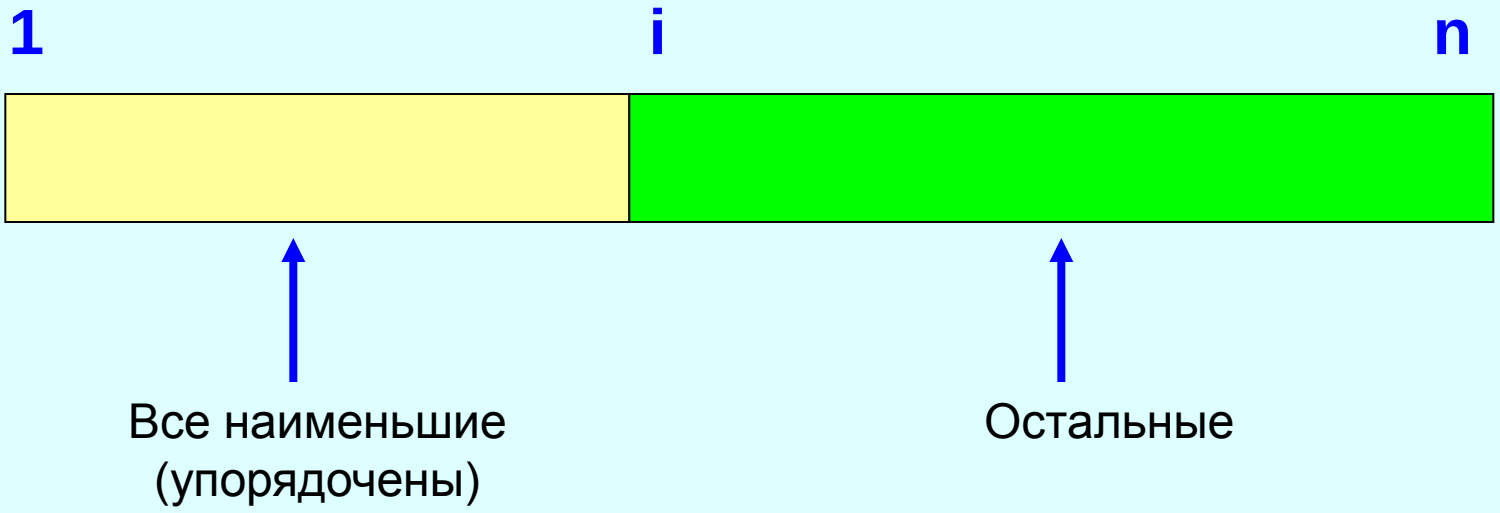
Идея сортировки выбором (*selection sort*) :

• состоит в том, чтобы создавать отсортированную последовательность путем присоединения к ней одного элемента за другим в правильном порядке. На каждой итерации находится *наименьший* из еще не упорядоченных элементов затем он перемещается в конец отсортированной части последовательности.

```
for (int i := 1; i < n – 1; i++ ){
//выбор очередного минимума и его перестановка (обмен)
ТЕЛО ЦИКЛА
}
```

Сортировка выбором

Инвариант внешнего цикла (в точке пред телом цикла):



$$(a[1..i) \le a[i..n]) \ \&\ Sort\ (a, 1, i-1) \ \&\ (1 \le i < n)$$

Если во внутреннем цикле ищем первое вхождение минимума, то сортировка в списке *устойчивая,* а в массиве *неустойчивая*

Сортировка выбором

```
template<class T>
void selectSort(T a[], long size) {
  long i, j, k;
  T x;

  for( i=0; i < size; i++) {     // i - номер текущего шага
    k=i; x=a[i];

    for( j=i+1; j < size; j++)   // цикл выбора наименьшего элемента
      if (  a[j] < x ) {
        k=j; x=a[j];             // k - индекс наименьшего элемента
      }

    a[k] = a[i]; a[i] = x;       // меняем местами наименьший с a[i]
  }
}
```

Анализ алгоритма сортировки выбором

Пусть C_i – число ***сравнений*** при выборе минимального элемента на i-ом шаге

(в сегменте из (n – i + 1) элементов).

$C_i = n - i$

Суммарно по всем шагам:

число сравнений

$$C = \sum_{i=1}^{n-1}(n-i) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \frac{(n^2-n)}{2}$$

Анализ алгоритма сортировки выбором

Пусть M_i – число ***перестановок*** на i-ом шаге, включая обновления текущего минимума (в сегменте из $(n - i + 1)$ элементов).

В ***худшем*** *случае* (обратно упорядоченный массив):

$$M_i = (n - i) + 3$$

Тогда суммарно по всем шагам ($i \in 1..n-1$):

$$M_{max} = (n^2 - n)/2 + 3(n - 1)$$

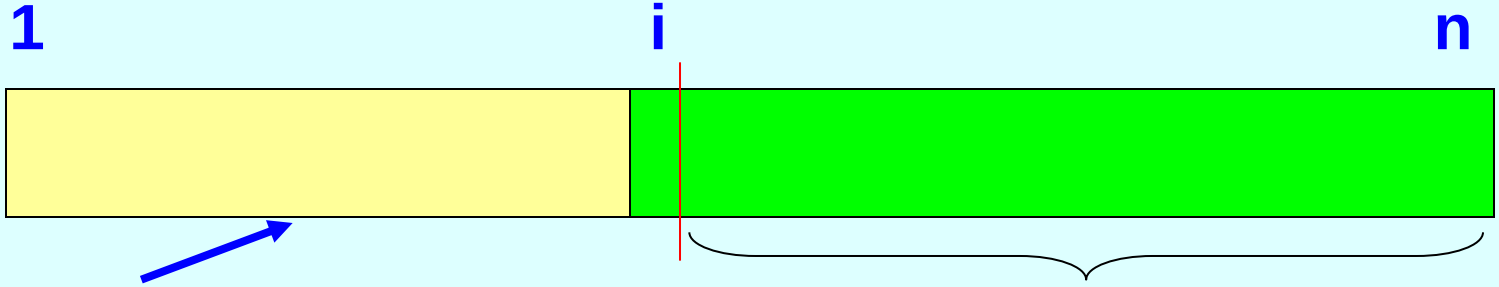
Анализ алгоритма сортировки выбором

В среднем:

Вероятность того, что произойдет обновление текущего минимума

на j-ом шаге *внутреннего цикла* $= 1/(j - i + 1)$

(любой, в том числе последний из $(j - i + 1)$ элементов - минимальный).



Все наименьшие (упорядочены)

$j \in (i + 1)..n$

На j-ом шаге обновляем (или не обновляем) текущий минимум среди $(j - i + 1)$ элементов

Анализ алгоритма сортировки выбором

В среднем (продолжение):

Т.о. за i-й шаг внешнего цикла *среднее число* перемещений (*мат.ожидание*) во внутреннем цикле

$= 1/2 + 1/3 + 1/4 + \ldots + 1/(n-i+1) = H_{n-i+1} - 1,$

где частичная сумма гармонического ряда

$H_k = 1 + 1/2 + 1/3 + \ldots + 1/k.$

Известно, что $H_k = \ln k + \gamma + 1/(2k) + \ldots$,

где γ - постоянная Эйлера.

Итак, за i-й шаг внешнего цикла среднее число присваиваний $H_{n-i+1} - 1 + 3 = H_{n-i+1} + 2 \approx \ln (n-i+1) + \gamma + 2$

Анализ сортировки выбором *в среднем*

В среднем за весь внешний цикл суммарное число перестановок есть

$$M = \sum_{i=1}^{n-1} [\ln(n-i+1) + (\gamma + 2)] =$$

$$= \sum_{i=1}^{n} \ln i + (n-1)(\gamma + 2) = ?$$

$$\sum_{i=1}^{n} \ln i \approx \int_{1}^{n} \ln x dx = (x \ln x - x)_1^n = n \ln n - n + 1$$

$$? \approx n \ln n + n(\gamma + 1)$$

Алгоритм сортировки выбором

Полезное изображение работы алгоритмов сортировки

Пусть множество значений a[1..n] ≡ {1..n}.

Примеры:

А

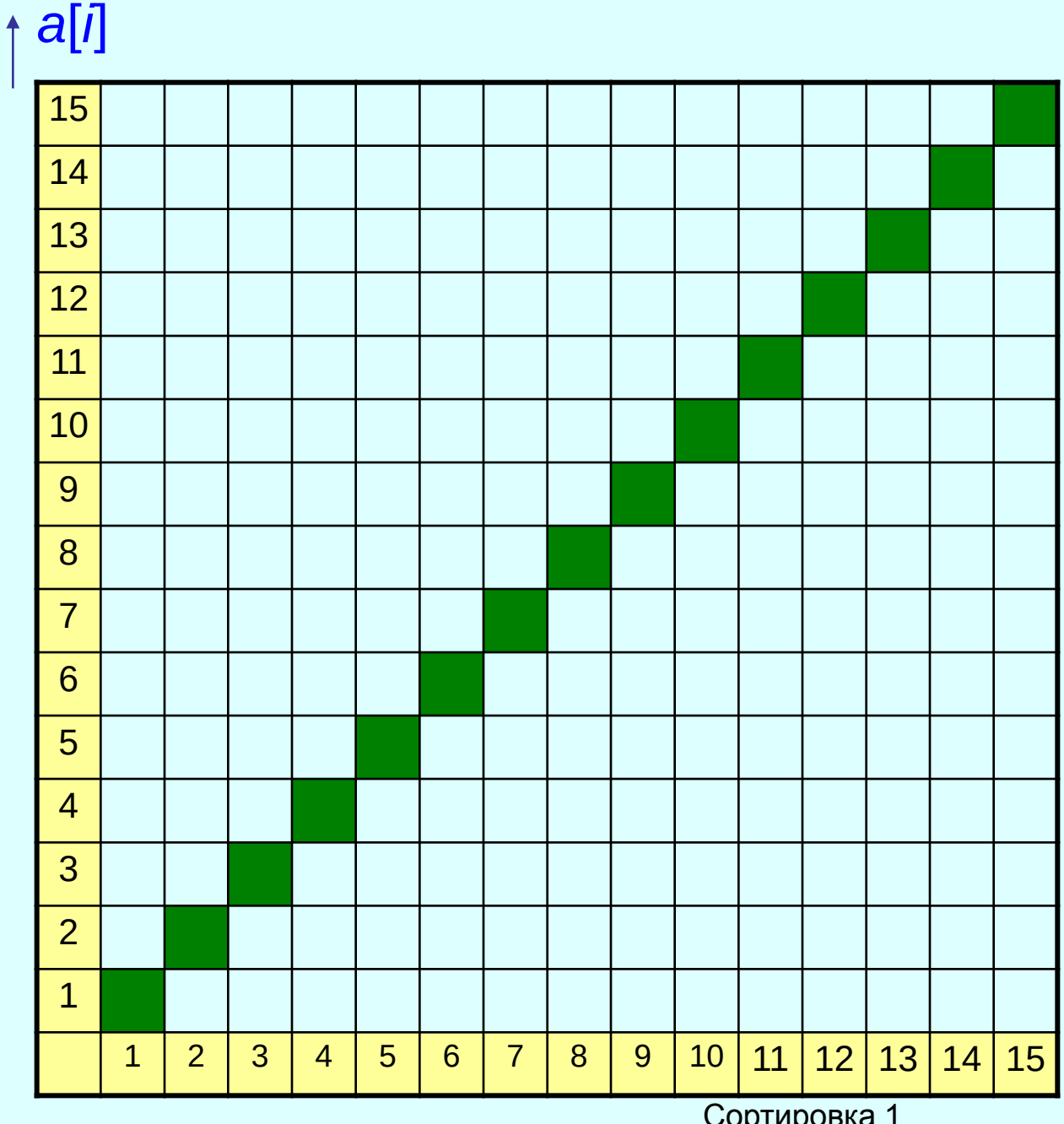
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Б

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

В

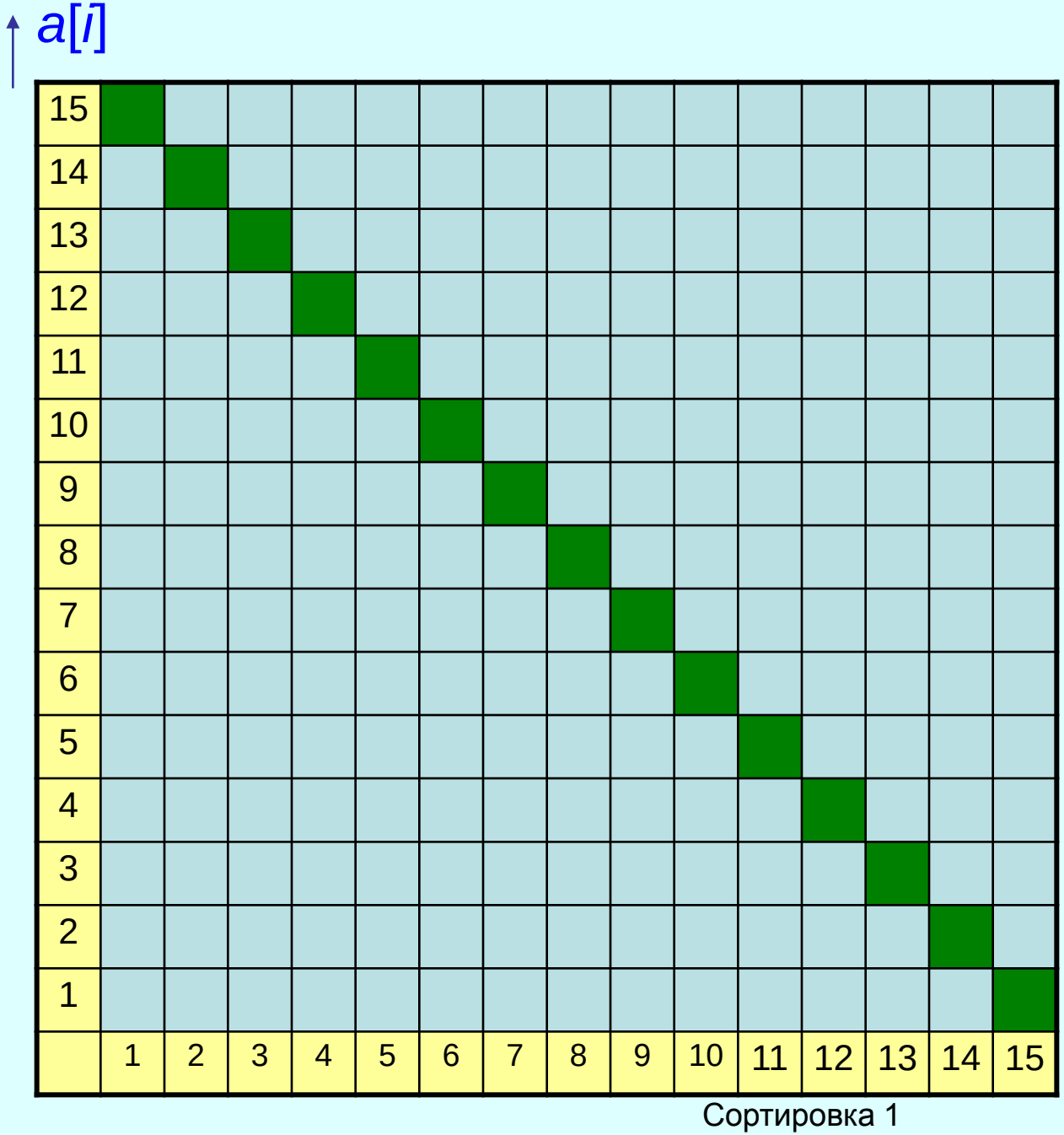
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	12	4	15	11	10	5	1	14	2	13	6	9	8	3



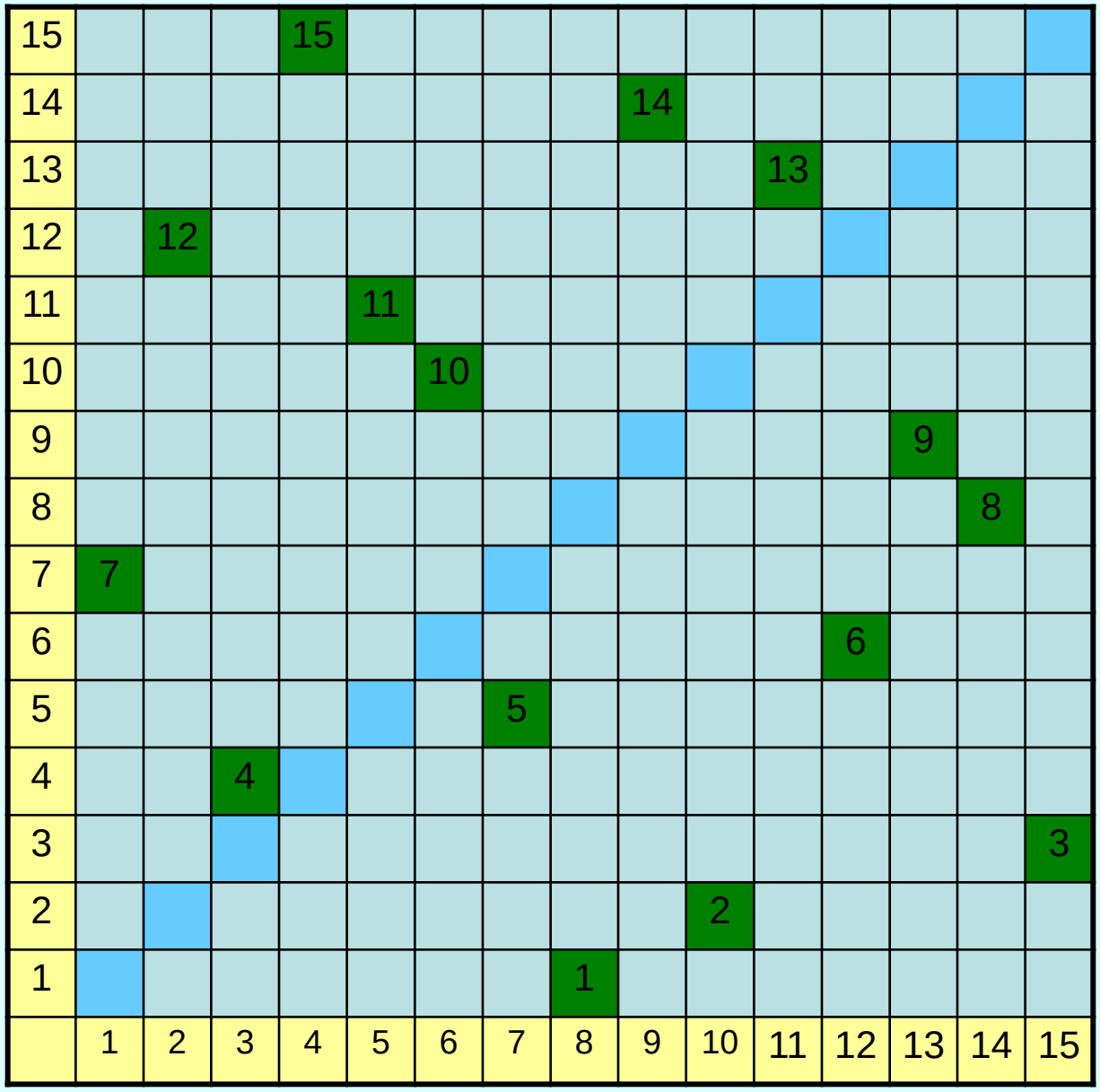
Функция («график»)

(*i*, *a*[*i*])

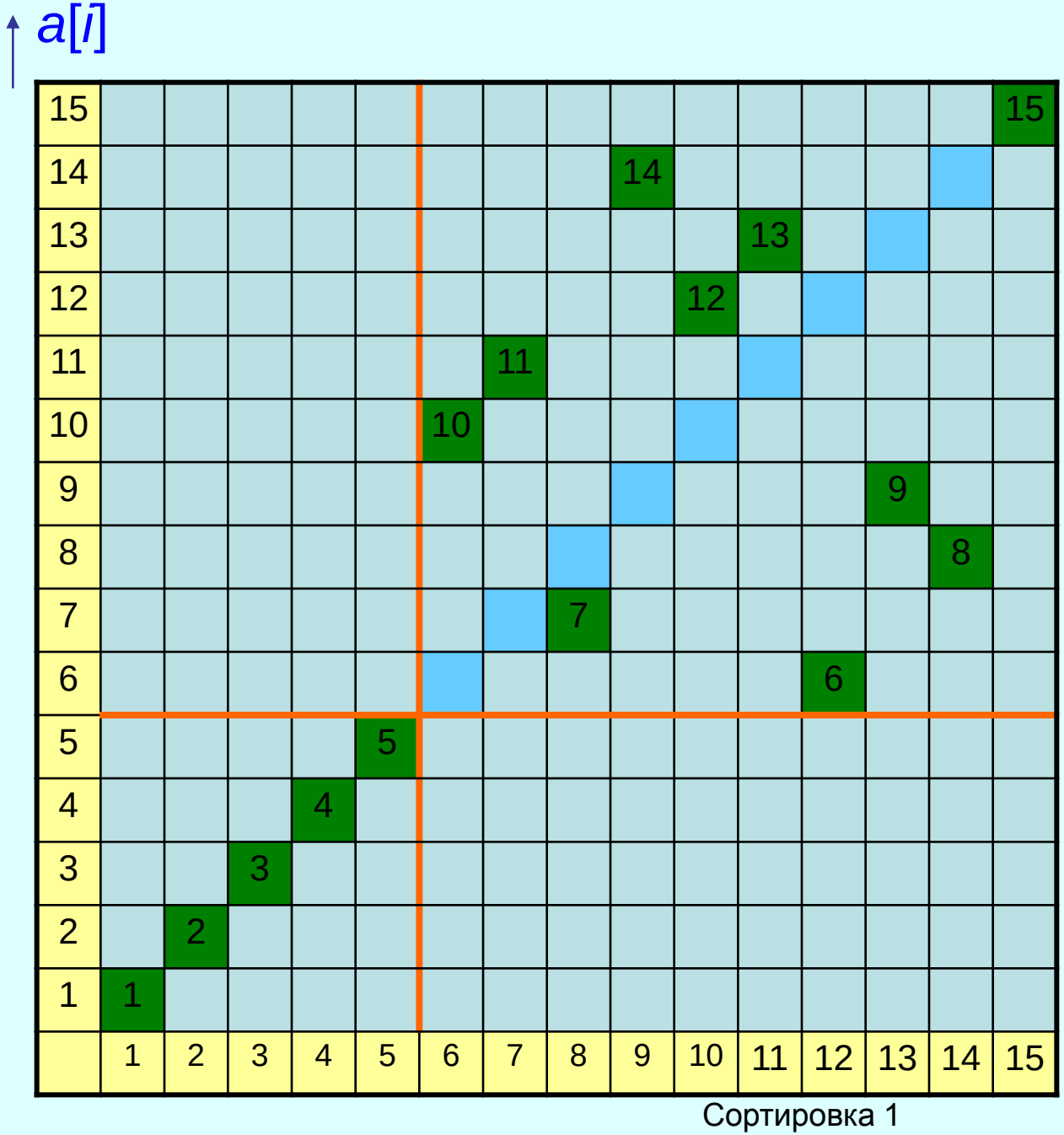
Пример А: упорядоченный массив



Пример Б:
обратно упорядоченный массив

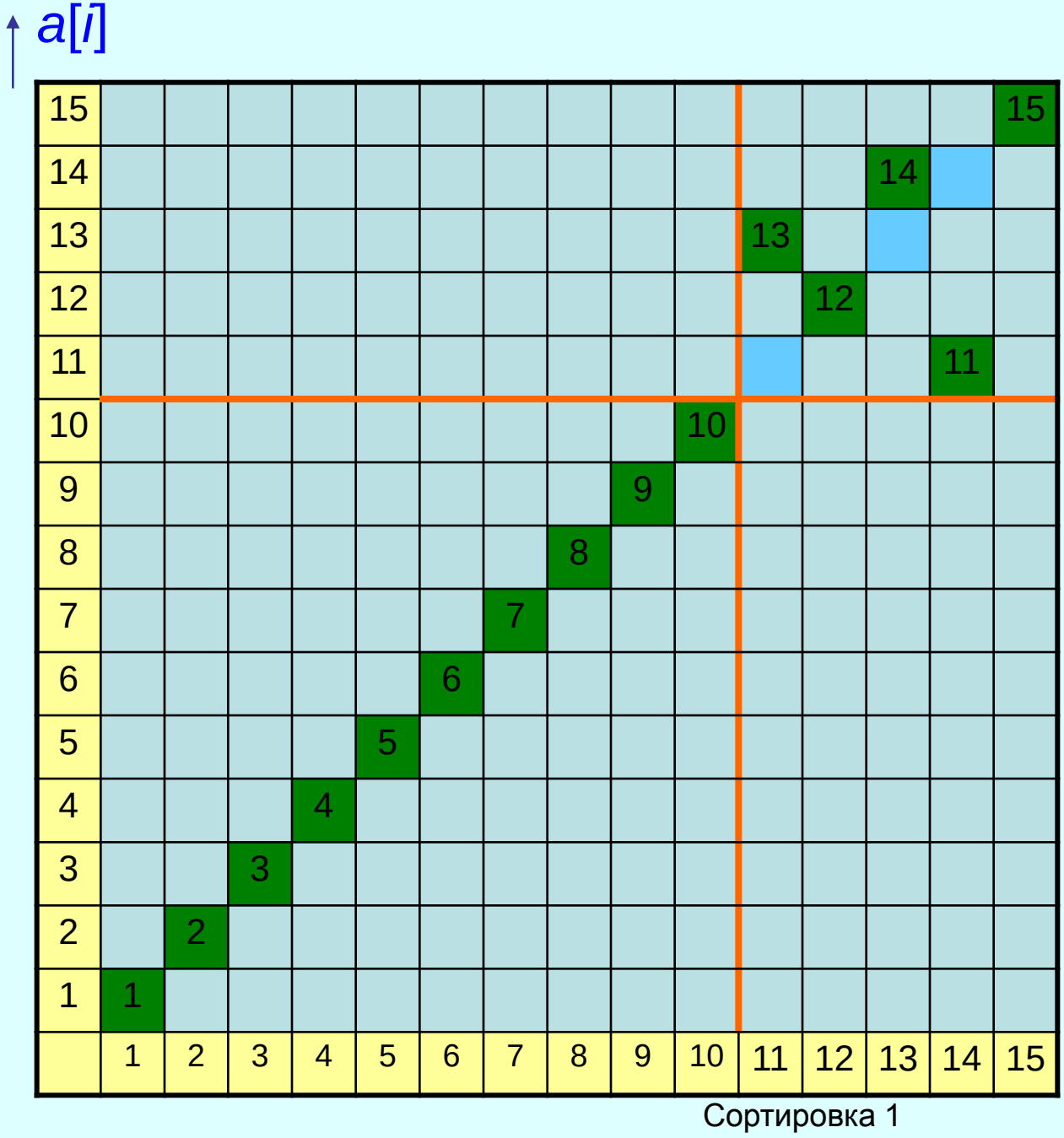


Пример B:
«случайный»
массив



Пример B:
«случайный»
массив.

Состояние
после первых 5
шагов внешнего
цикла



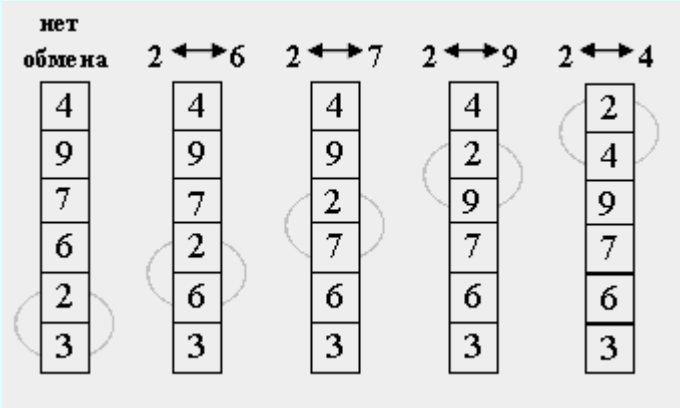
Пример B:
«случайный» массив.

Состояние после первых 10 шагов внешнего цикла

Простые алгоритмы сортировки

Сортировка обменами («пузырьковая»)

- Расположим массив сверху вниз, от нулевого элемента - к последнему.
- Идея метода: шаг сортировки состоит в проходе снизу вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами.



Простые алгоритмы сортировки

Сортировка обменами («пузырьковая»)

1 проход	*2 проход*	*3 проход*	*4 проход*
упОрядочИвание	пОрудочИваниея	ОпрдоуИваниечя	ОпдорИваниеучя
пуОрядочИвание	ОпрудочИваниея	ОпрдоуИваниечя	ОпдорИваниеучя
пОурядочИвание	ОпрудочИваниея	ОпрдоуИваниечя	ОдпорИваниеучя
пОруядочИвание	ОпрудочИваниея	ОпдроуИваниечя	ОдопрИваниеучя
пОруядочИвание	ОпрдуочИваниея	ОпдоруИваниечя	ОдопрИваниеучя
пОрудяочИвание	ОпрдоучИваниея	ОпдоруИваниечя	ОдопИрваниеучя
пОрудоячИвание	ОпрдоучИваниея	ОпдорИуваниечя	ОдопИврaниеучя
пОрудочяИвание	ОпрдоуИчваниея	ОпдорИвуаниечя	ОдопИварниеучя
пОрудочИявание	ОпрдоуИвчаниея	ОпдорИваунниечя	ОдопИванриеучя
пОрудочИвяание	ОпрдоуИвачнниея	ОпдорИвануиечя	ОдопИванireучя
пОрудочИваянние	ОпрдоуИванчиея	ОпдорИваниуечя	ОдопИваниеручя
пОрудочИваняие	ОпрдоуИваничея	ОпдорИваниеучя	
пОрудочИванияе	ОпрдоуИваниечя		
пОрудочИваниея			

Сортировка обменами («пузырьковая»)

Продолжение

5 проход	*6 проход*	*7 проход*	*8 проход*
ОдопИванieручя	дОоИванieпручя	дОИванieопручя	дИванieОопручя
дОопИванieручя	дОоИванieпручя	дИОванieопручя	дИванieОопручя
дОопИванieручя	дОоИванieпручя	дИОванieопручя	двИанieОопручя
дОопИванieручя	дОИованieпручя	дИвОанieопручя	дваИнieОопручя
дОоИпванieручя	дОИвоанieпручя	дИваОнieопручя	дваИнieОопручя
дОоИвпанieручя	дОИваонieпручя	дИванОиeопручя	дваИиneОопручя
дОоИвапнieручя	дОИваноиeпручя	дИвани Оеопручя	дваИиенОопручя
дОоИванпиeручя	дОИвании оепручя	дИванieОопручя	
дОоИвани перучя	дОИванieопручя		
дОоИванieпручя			

Сортировка обменами («пузырьковая»)

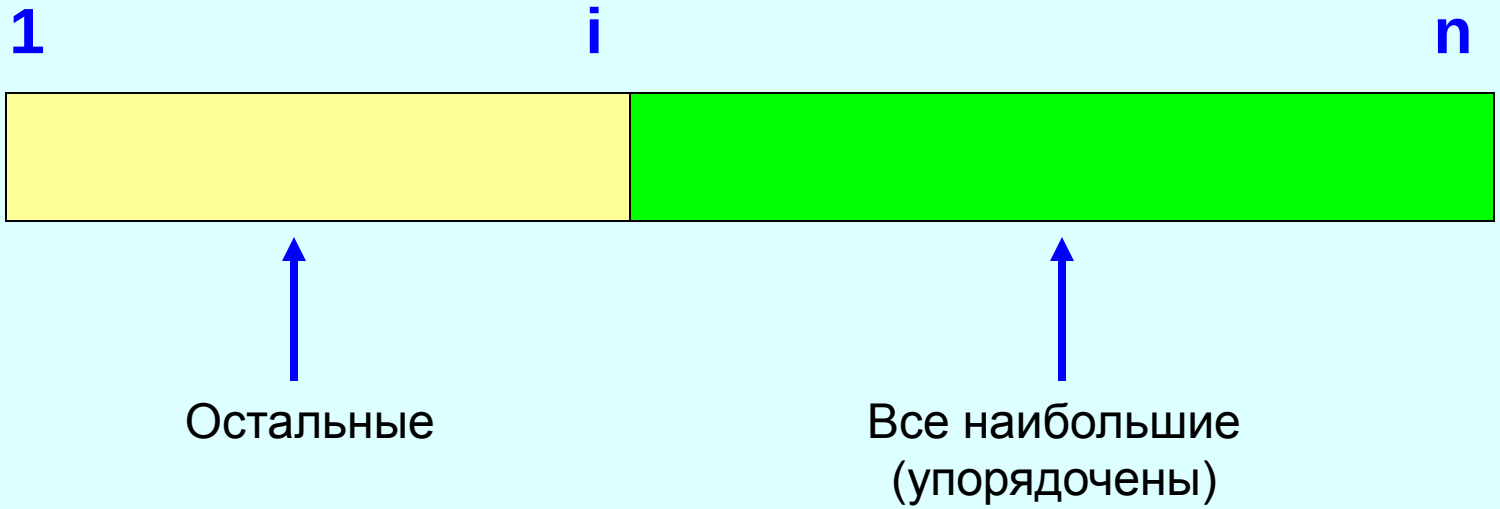
Продолжение

9 проход	*10 проход*	*11 проход*
двaИиенОопручя	вадИеинОопручя	авдеИинОопручя
вдаИиенОопручя	авдИеинОопручя	авдеИинОопручя
вадИиенОопручя	авдИеинОопручя	авдеИинОопручя
вадИиенОопручя	авдИеинОопручя	авдеИинОопручя
вадИиенОопручя	авдеИинОопручя	----------
вадИеинОопручя		
		авдеИинОопручя

Сортировка обменами

Инвариант внешнего цикла

for (int i=n; i>2; i--)



$(a[1..i] \le a(i..n]) \;\&\; Sort\,(a, i+1, n) \;\&\; (1 \le i \le n)$

Сортировка обменами (пузырьковая)

```
template<class T>
void bubbleSort(T a[], long size) {
  long i, j;
  T x;

  for( i=0; i < size; i++) {              // i - номер прохода
    for( j = size-1; j > i; j-- ) {       // внутренний цикл прохода
      if ( a[j-1] > a[j] ) {
      x=a[j-1]; a[j-1]=a[j]; a[j]=x;
    }
  }
}
```

Анализ сортировки обменами (пузырьковой)

$$C_i = i - 1; \quad C = \sum_{i=2}^{n} (i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Число сравнений

Число перестановок

В худшем случае

$$M_i = 3(i-1); \quad M = \frac{3}{2} n(n-1).$$

В среднем: $M_i = \frac{3}{2}(i-1); \quad M = \frac{3}{4} n(n-1).$

Среднее по всем перестановкам элементов массива.

Перестановки разбиваются на пары: для данной перестановки всегда есть обратная.

Действительно:

$$\begin{Bmatrix} k_1, k_2, \ldots, k_i, \ldots, k_j, \ldots, k_{n-1}, k_n \\ k_n, k_{n-1}, \ldots, k_j, \ldots, k_i, \ldots, k_2, k_1 \end{Bmatrix}$$

$k_i \leftrightarrow k_j$ один раз на 2 последовательности.

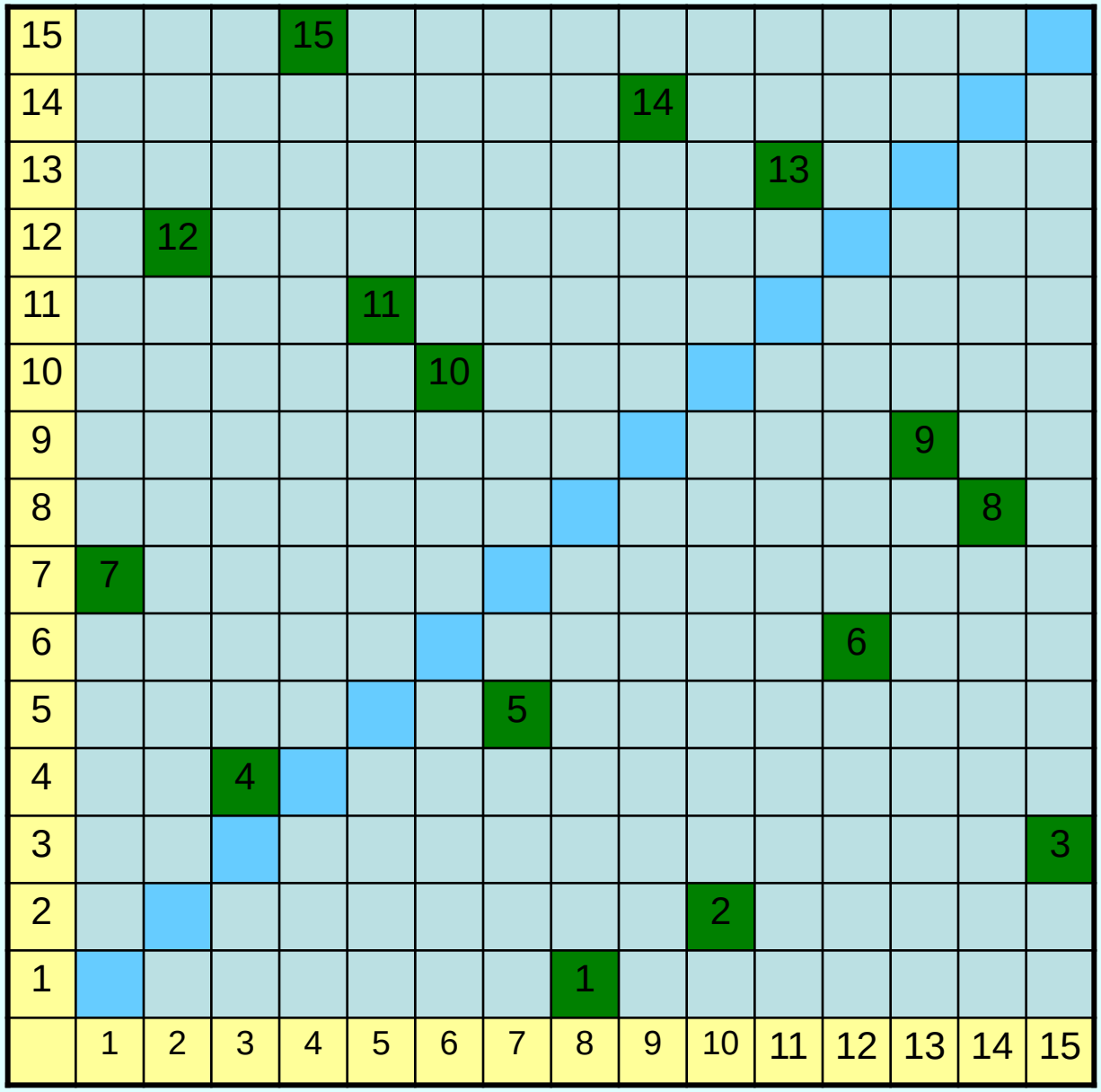
Обмен произойдет, если только $k_i < k_j$ (или $k_i > k_j$).

Всего перестановок пар $C_2^n = \frac{n(n-1)}{2}$

и всего обменов $\frac{n(n-1)}{4}$

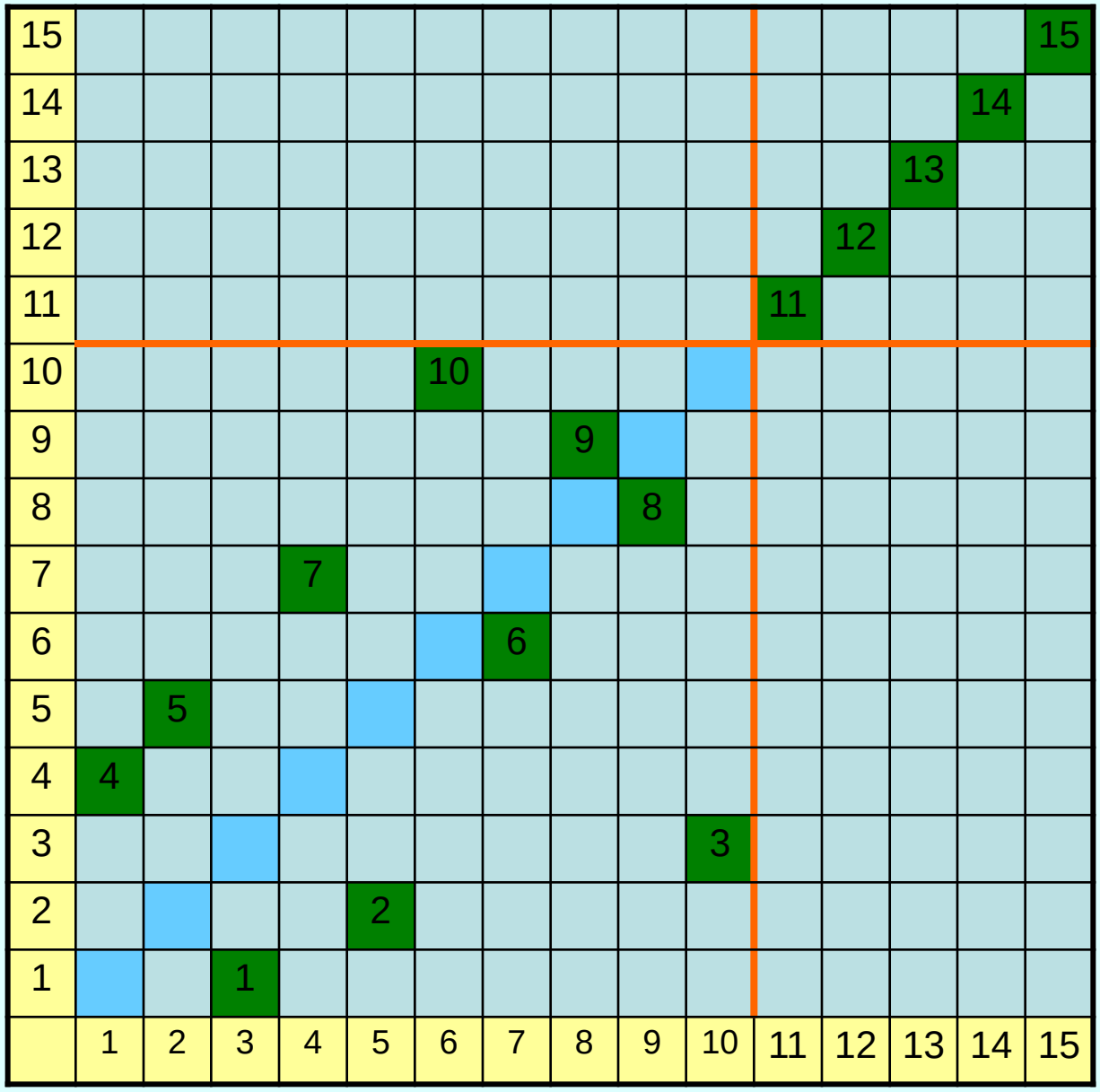
Это для любых *двух последовательностей*, а следовательно в среднем и для всех

Сортировка обменами («пузырьковая»)



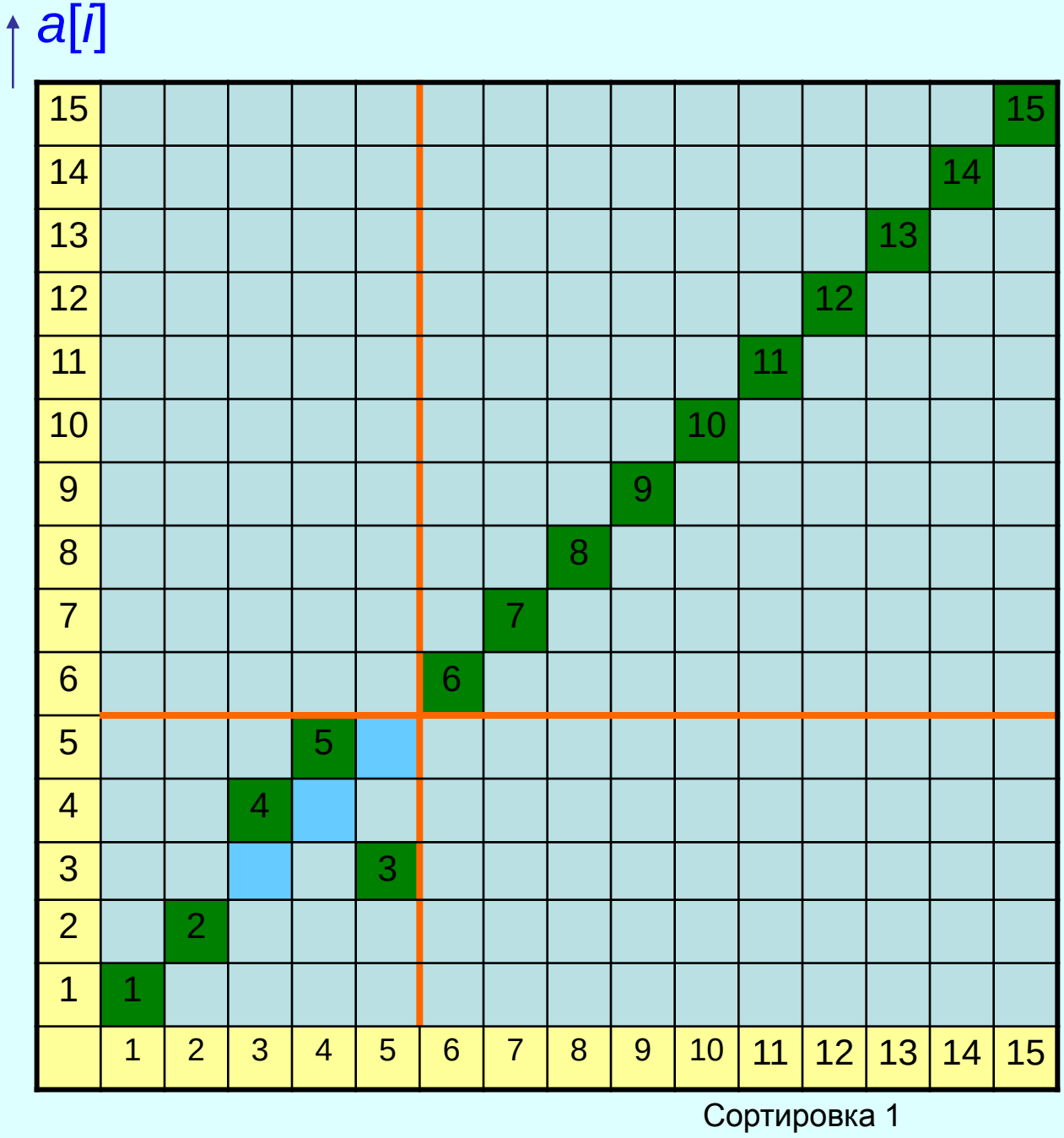
Пример В: исходный *«случайный»* массив

Сортировка обменами («пузырьковая»)



Пример В: *«случайный»* массив.

Состояние после первых 5 шагов внешнего цикла.



Сортировка обменами («пузырьковая»)

Пример B: *«случайный»* массив.

Состояние после первых 10 шагов внешнего цикла.

Простые алгоритмы сортировки

Сортировка вставками (включением)

- Будем разбирать алгоритм, рассматривая его действия на i-м шаге. Последовательность к этому моменту разделена на две части: готовую a[0]...a[i] и неупорядоченную a[i+1]...a[n].
- На следующем, (i+1)-м каждом шаге алгоритма берем a[i+1] и вставляем на нужное место в готовую часть массива. Поиск подходящего места для очередного элемента входной последовательности осуществляется путем последовательных сравнений с элементом, стоящим перед ним.

Таким образом, в процессе вставки мы "просеиваем" элемент x к началу массива, останавливаясь в случае, когда

1. Найден элемент, меньший x или
2. Достигнуто начало последовательности.

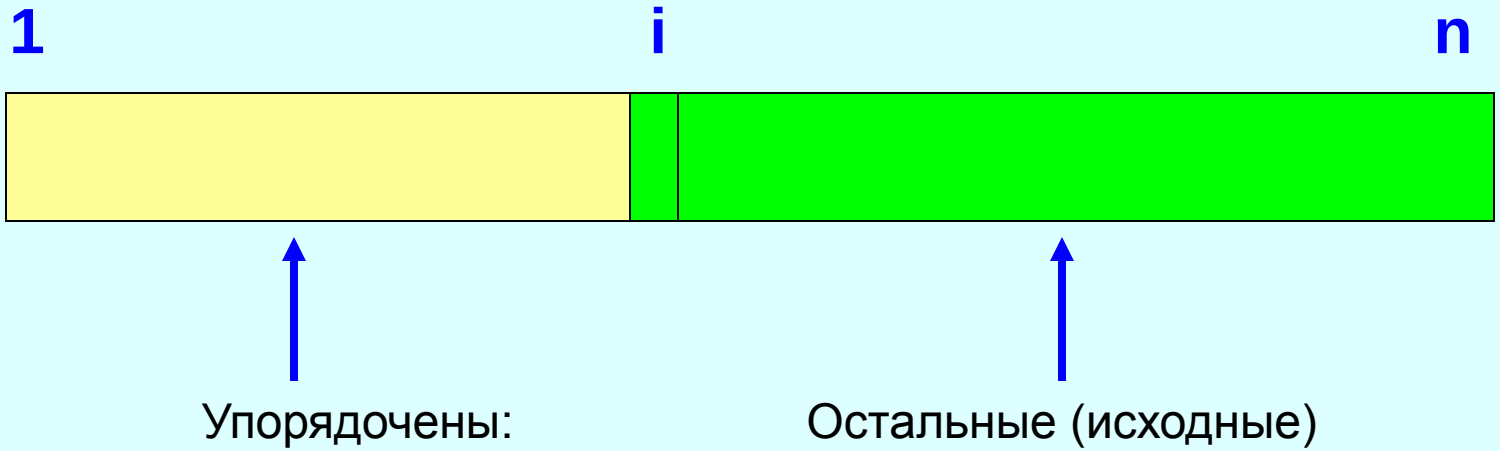
Простые алгоритмы сортировки

Сортировка вставками (включением)

упОрядочИвание	дОопручяИвание
пуОрядочИвание	дИОопручявание
ОпурядочИвание	вдИОопручяание
ОпруядочИвание	авдИОопручяние
ОпруядочИвание	авдИнОопручяие
дОпруяочИвание	авдИинОопручяе
дОопруячИвание	авдеИинОопручя

Сортировка ВСТАВКАМИ

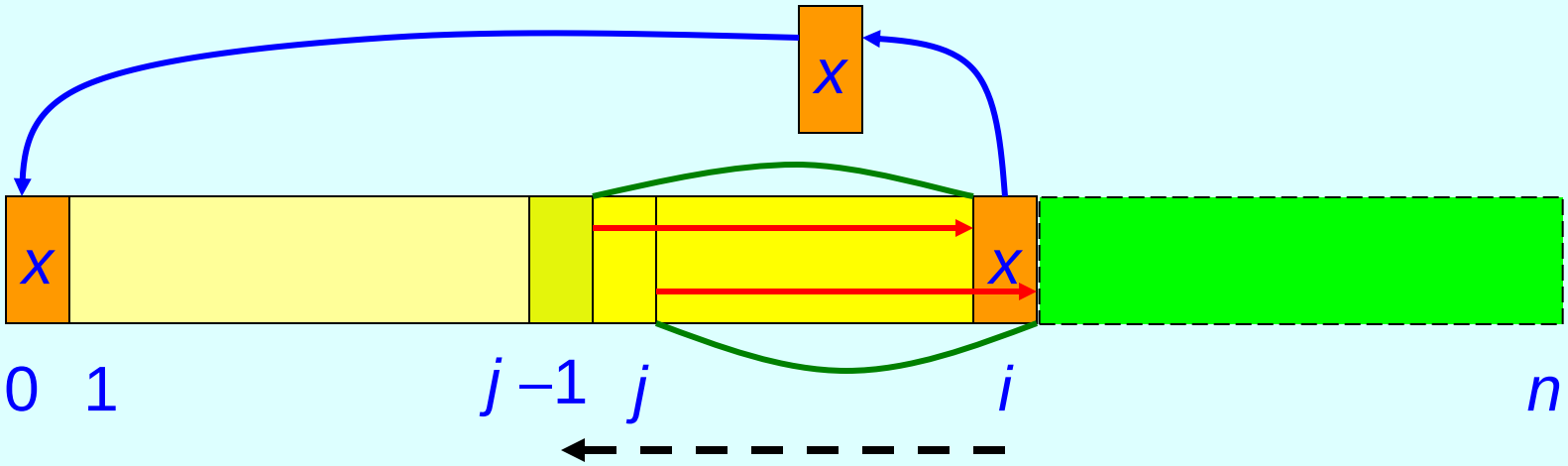
Инвариант внешнего цикла:



Sort (a, 1, i -1)

После i – го шага: Sort (a, 1, i)

Сортировка прямыми ВСТАВКАМИ



Sort (a, 1, i -1) & (x < a0[j..i-1]) & (a[j+1..i]=a0[j..i-1])

```
j := i;
while (x < a[j -1]) {
  a[j] := a[j -1];  j := j -1;
{ (x ≥ a[j -1]) & (x < a [j+1..i]) & (a[j] – «свободен»)}
a[j] := x;
```

АЛГОРИТМ *Прямые вставки*

```
template<class T>
void insertSort(T a[], long size) {
  T x;
  long i, j;

  for ( i=0; i < size; i++) {  // цикл проходов, i - номер прохода
    x = a[i];

    // поиск места элемента в готовой последовательности
    for ( j=i-1; j>=0 && a[j] > x; j--)
      a[j+1] = a[j];    // сдвигаем элемент направо, пока не дошли

    // место найдено, вставить элемент
    a[j+1] = x;
  }
}
```

Задание.

Изобразить результат работы очередного шага внешнего цикла, как функцию («график»)

(*i*, *a*[*i*])

Сортировка ПРЯМЫМИ ВСТАВКАМИ

Анализ

Число сравнений ≈ число перемещений.

Число сравнений на i –ом шаге $C_i = i/2$ в среднем.

Тогда по всем шагам $i \in 2..n$

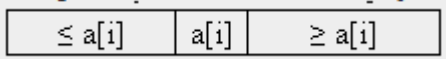
$$C = \frac{1}{2}\sum_{i=2}^{n} i = \frac{1}{2}\left(\frac{n(n+1)}{2} - 1\right)$$

$$= \frac{1}{4}(n(n+1) - 2) \approx \frac{1}{4}n^2$$

Быстрая сортировка (**QuickSort**)

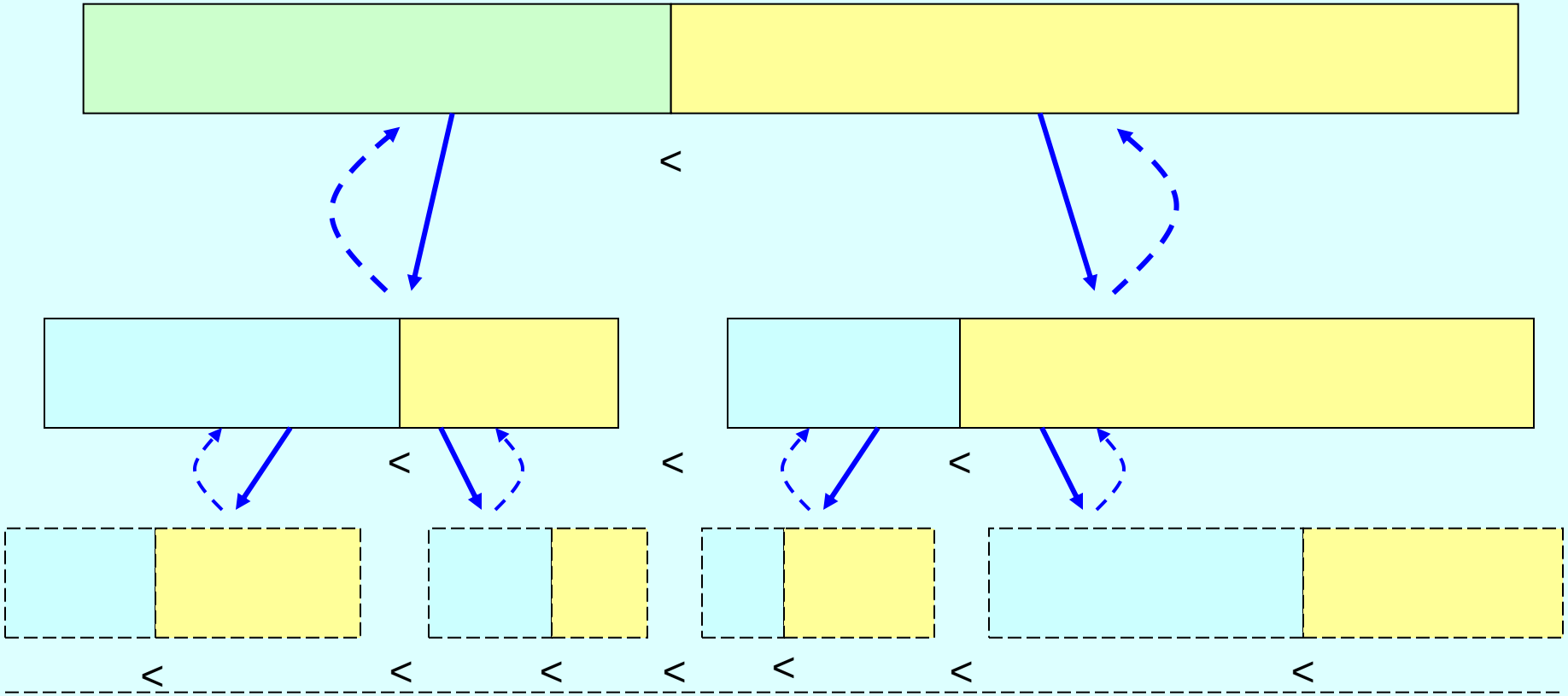
Метод основан на подходе "разделяй-и-властвуй". Общая схема такова:

1. из массива выбирается некоторый опорный элемент a[i],
2. запускается процедура разделения массива, которая перемещает все ключи, меньшие, либо равные a[i], влево от него, а все ключи, большие, либо равные a[i] - вправо,
3. теперь массив состоит из двух подмножеств, причем левое меньше, либо равно правого,



4. для обоих подмассивов: если в подмассиве более двух элементов, рекурсивно запускаем для него ту же процедуру.

Быстрая сортировка:
разделение и слияние



На входе: массив a[0]...a[N] и опорный элемент p, по которому будет производиться разделение.

1. Введем два указателя: i и j. В начале алгоритма они указывают, соответственно, на левый и правый конец последовательности.
2. Будем двигать указатель i с шагом в 1 элемент по направлению к концу массива, пока не будет найден элемент a[i] >= p. Затем аналогичным образом начнем двигать указатель j от конца массива к началу, пока не будет найден a[j] <= p.
3. Далее, если i <= j, меняем a[i] и a[j] местами и продолжаем двигать i,j по тем же правилам...
4. Повторяем шаг 3, пока i <= j.

- Рассмотрим работу функции для массива a[0]...a[6] и опорного элемента p = a[3].

Общий алгоритм

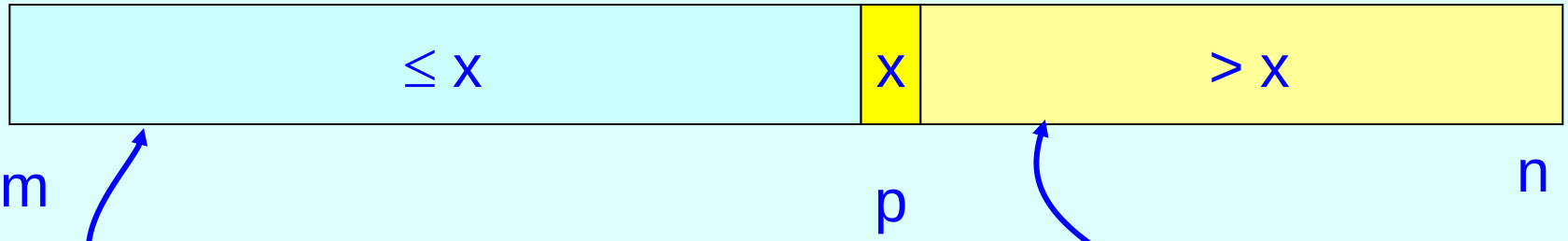
Псевдокод.

quickSort (массив a, верхняя граница N) {

- Выбрать опорный элемент p - середину массива
- Разделить массив по этому элементу
- Если подмассив слева от p содержит более одного элемента, вызвать quickSort для него.
- Если подмассив справа от p содержит более одного элемента, вызвать quickSort для него.

}

Быстрая сортировка:




```
void Sort ( mass[] a; index1 m, n; index1 k )
    index2 p, q;
  {
      Partition ( a, m, n, p );
      if (p-m>k)   Sort ( a, m, p-1, k );
       q:=p+1;
      if (n-q+1>k)  Sort ( a, q, n, k )
   { подмассив длиной <= k  не сортируем !
        1<=k<=50 , при k=1 - полная сортировка }
  };
```

Программная реализация алгоритма

```
template<class T>
void quickSortR(T* a, long N) {
// На входе – массив a[], a[N] – его последний элемент.

  long i = 0, j = N;          // поставить указатели на исходные места
  T temp, p;

  p = a[ N>>1 ];          // центральный элемент

  // процедура разделения
  do {
    while ( a[i] < p ) i++;
    while ( a[j] > p ) j--;

    if (i <= j) {
      temp = a[i]; a[i] = a[j]; a[j] = temp;
      i++; j--;
    }
  } while ( i<=j );
```

Задание.

Изобразить результат работы процедуры разделения, как функцию («график»)

(i, $a[\, i\,]$)

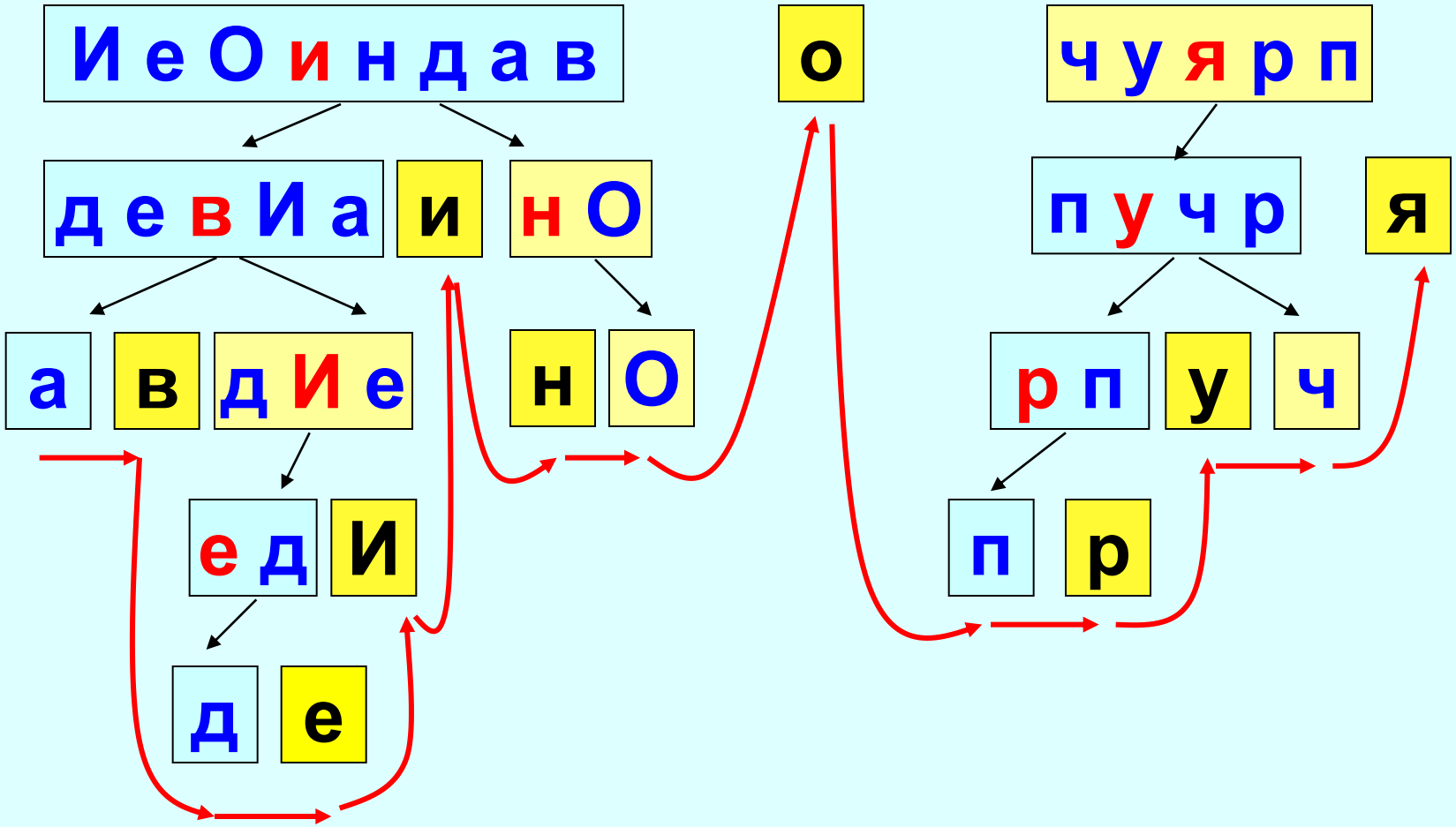
Первый вызов Partition

1	2	3	4	5	6	7	8	9	10	11	12	13	14
у	п	О	р	я	д	о	ч	И	в	а	н	и	е
о	п	О	р	я	д	у	ч	И	в	а	н	и	е
о	е	О	р	я	д	у	ч	И	в	а	н	и	п
о	е	О	р	я	д	у	ч	И	в	а	н	и	п
о	е	О	и	я	д	у	ч	И	в	а	н	р	п
о	е	О	и	н	д	у	ч	И	в	а	я	р	п
о	е	О	и	н	д	у	ч	И	в	а	я	р	п
о	е	О	и	н	д	а	ч	И	в	у	я	р	п
о	е	О	и	н	д	а	в	И	ч	у	я	р	п
И	е	О	и	н	д	а	в	о	ч	у	я	р	п

Быстрая сортировка

1 2 3 4 5 6 7 8 9 10 11 12 13 14

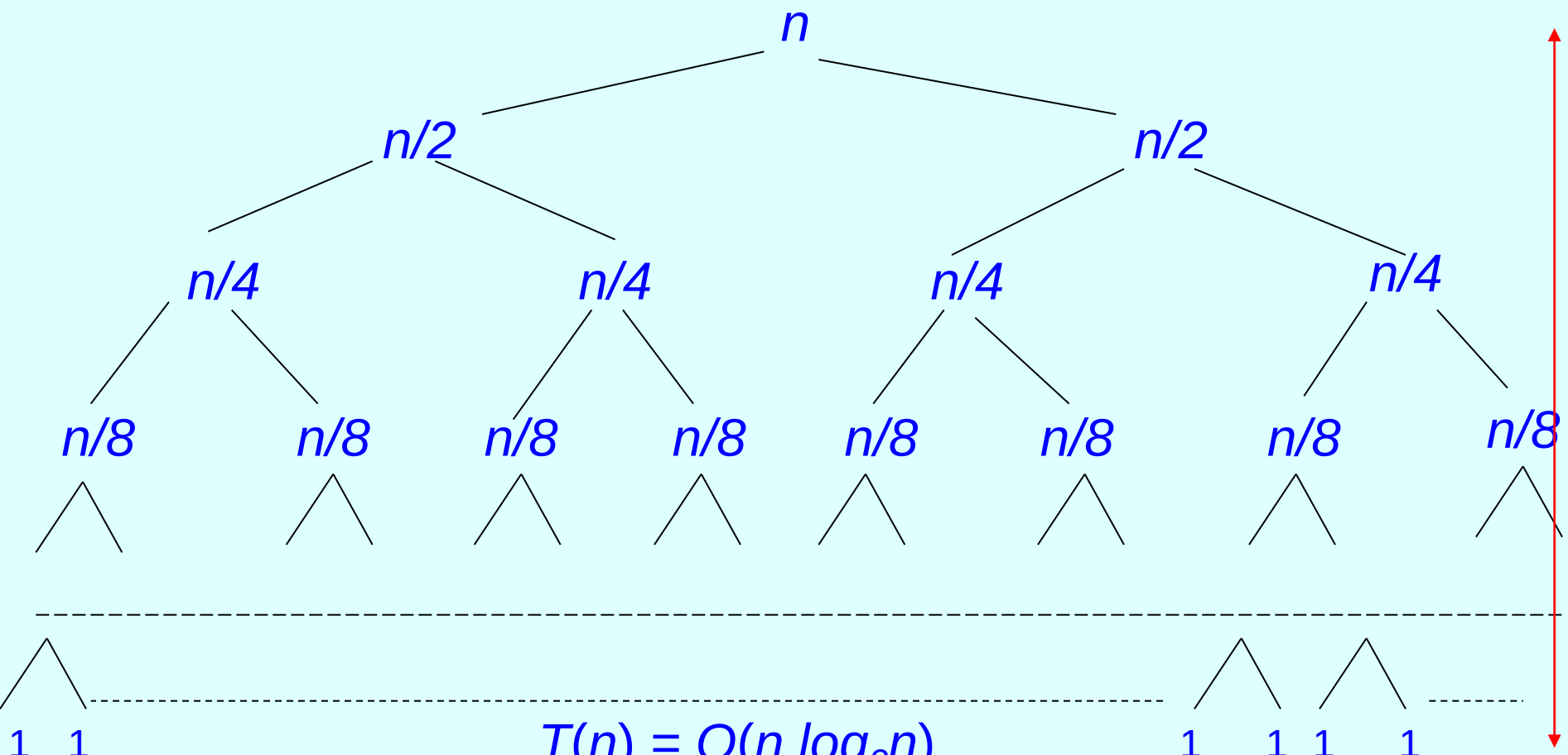
И е О и н д а в о ч у я р п



Быстрая сортировка (**QuickSort**)

Анализ. Лучший случай

$$T(n) \le 2T(n/2) + \Theta(n)$$

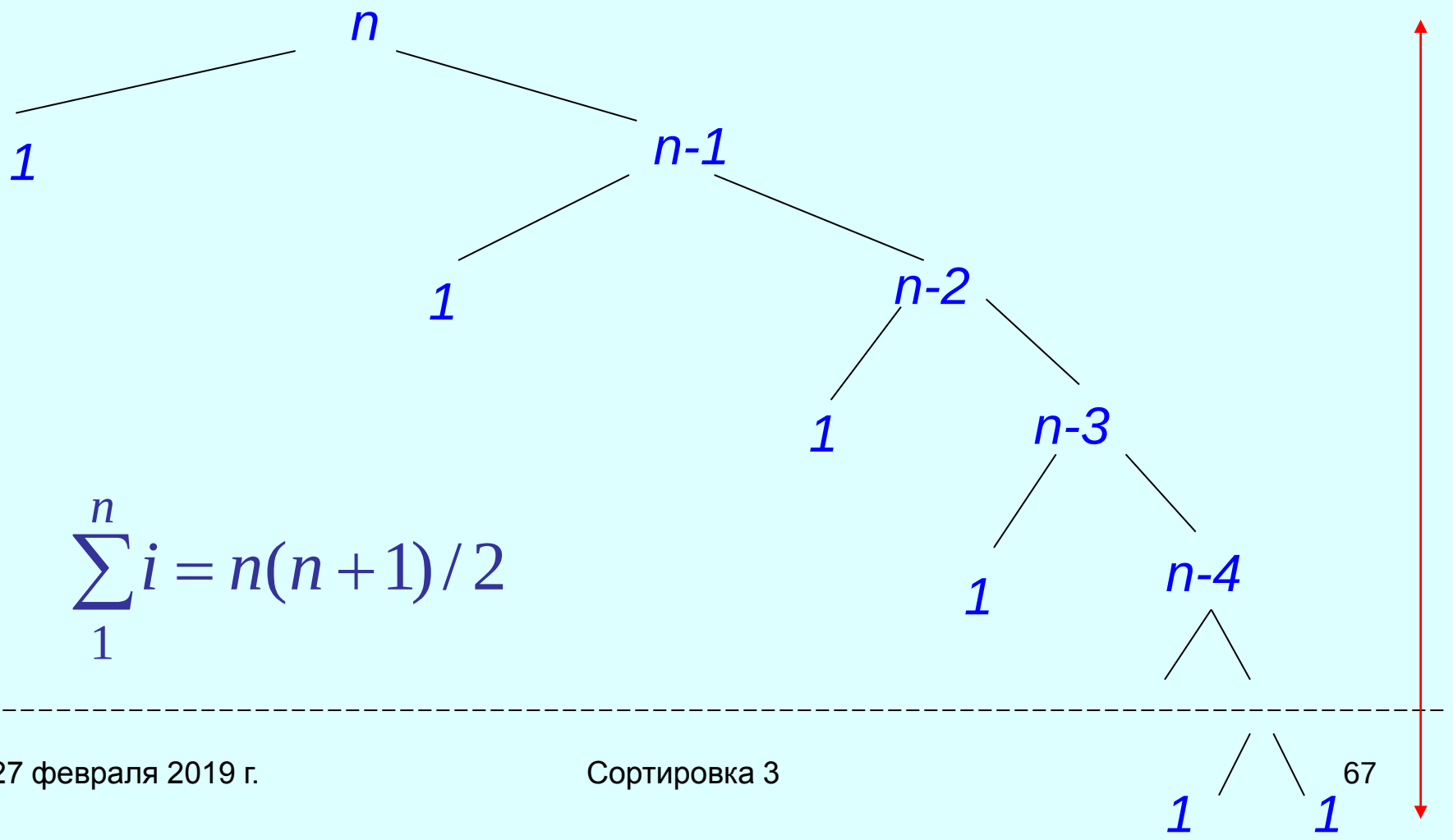


$T(n) = O(n \log_2 n)$

Быстрая сортировка (**QuickSort**)

Анализ. Худший случай

$$T(n) \le T(n-1) + \Theta(n)$$

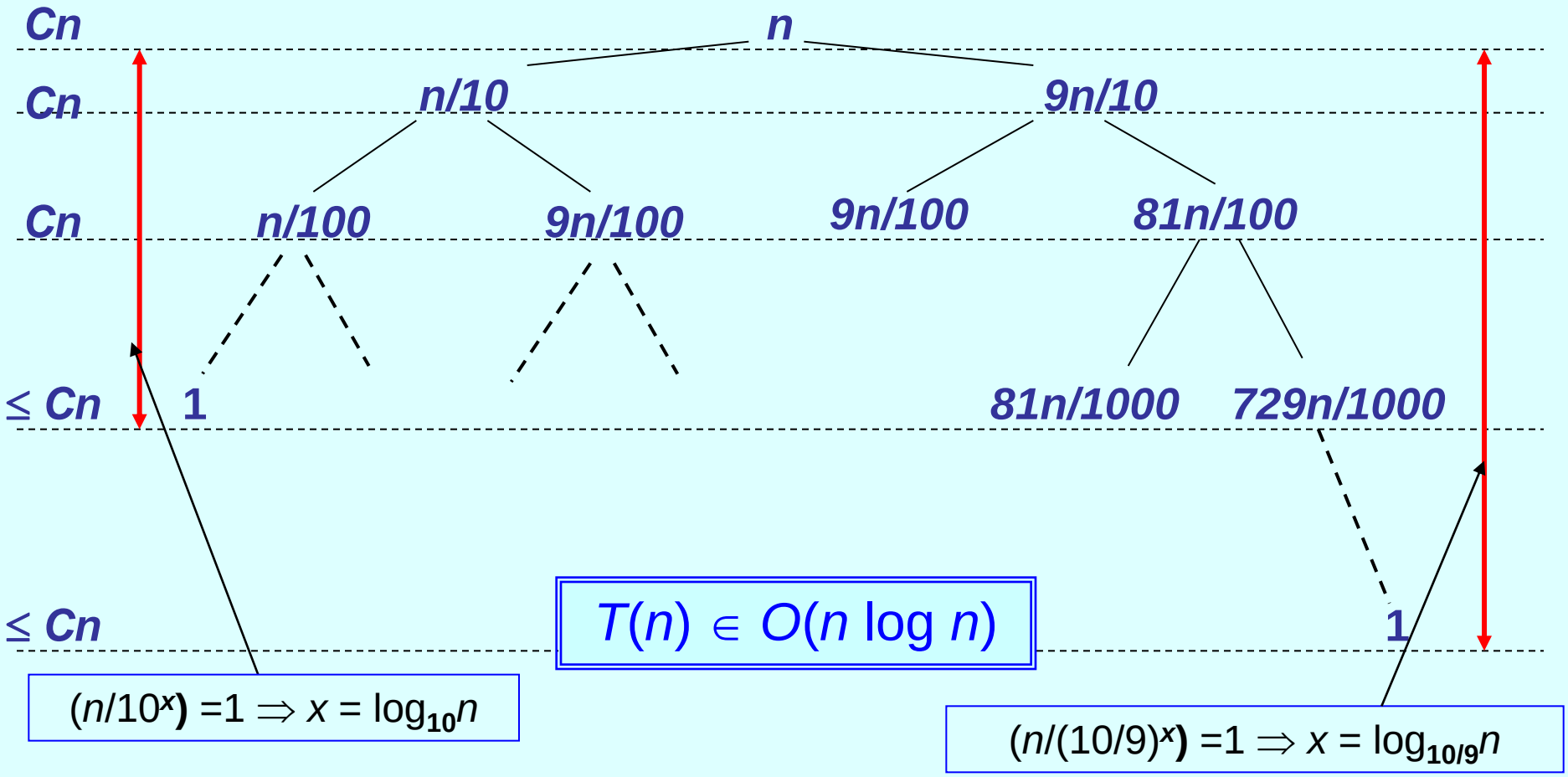


$$\sum_{1}^{n} i = n(n+1)/2$$

Быстрая сортировка (**QuickSort**)

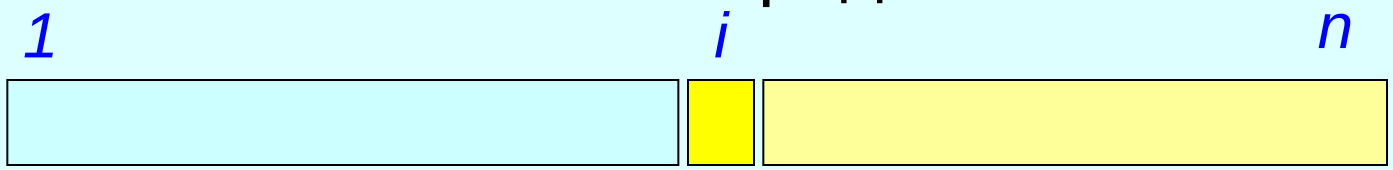
Анализ. Промежуточный случай

$$T(n) \le T(9n/10) + T(n/10) + \Theta(n)$$



Быстрая сортировка (**QuickSort**)

Анализ. В среднем



$q(n)$ – среднее число сравнений

$$q(n) \le Cn + \frac{1}{n}\sum_{i=1}^{n}[q(i-1)+q(n-i)] =$$

$$= Cn + \frac{2}{n}\sum_{i=0}^{n-1} q(i); \quad q(0) = q(1) = b;$$

$$\sum_{i=1}^{n}[q(i-1)+q(n-i)] =$$

$$= [q(0)+q(n-1)]+[q(1)+q(n-2)]+\ldots+[q(n-2)+q(1)]+[q(n-1)+q(0)] =$$

$$= 2\sum_{i=0}^{n-1} q(i);$$

Итак

$q(n)$ – среднее число сравнений

$$q(n) \le Cn + \frac{2}{n}\sum_{i=0}^{n-1} q(i); \quad q(0) = q(1) = b;$$

Докажем, что $q(n) \le k\ n \ln n, \quad k = 2(b + C)$, при $n \ge 2$

По индукции:

а) $n = 2$: $q(2) \le C \cdot 2 + (q(0) + q(1)) = 2(C + b) = k \le k \cdot 2 \ln 2$

б) пусть $q(i) \le k\ i \ln i, \quad i \in 2..(n-1)$; докажем, что $q(n) \le k\ n \ln n$

Быстрая сортировка (**QuickSort**)

Анализ. В среднем (продолжение)

$$q(n) \le Cn + \frac{2}{n}(q(0) + q(1)) + \frac{2}{n}\sum_{i=2}^{n-1} ki \ln i = ?$$

Рассмотрим

$$\sum_{i=2}^{n-1} i \ln i \le \int_2^n x \ln x dx = \frac{1}{2}(x^2 \ln x - \frac{1}{2}x^2)_2^n =$$

$$= \frac{1}{2}(n^2 \ln n - \frac{1}{2}n^2 - 4\ln 2 + \frac{1}{2}4) < \frac{1}{2}(n^2 \ln n - \frac{1}{2}n^2)$$

Быстрая сортировка (**QuickSort**)

Анализ. В среднем (продолжение)

$$? = Cn + \frac{4b}{n} + \frac{k}{n}(n^2 \ln n - \frac{1}{2}n^2) =$$

$$= kn\ln n - \frac{k}{2}n + Cn + \frac{4b}{n} = kn\ln n + \frac{4b}{n} - bn =$$

$$= kn\ln n + b(\frac{4}{n} - n) \le kn\ln n$$

$$\text{Итак} \quad q(n) \le kn\ln n, \quad \text{или} \quad q(n) = O(n\log n)$$

Идеальный код «*ПИТЕР*»

статья Джона Бентли про Quicksort

Оригинал:	**Beautiful Code**
Авторы:	Орам Энди, Уилсон Грег
Серия:	Бестселлеры O'Reilly
Тема:	Советы программистам
2009 год;	1-е издание, 2008 год, 624 стр.

2. Сортировка слиянием

Сортировка слиянием («разделяй и властвуй»)

- Общая идея метода «разделяй и властвуй»
- Сортировка слиянием (на примере списков)
- Анализ

Общая идея метода «разделяй и властвуй»

“Divide et impera”

Три стадии:

1. *Разделение* – разбиение данных на две или более составляющих (части); если размер данных меньше некоторой пороговой величины, то возвращается готовый результат.
2. *Рекурсия* – рекурсивно решается каждая из подзадач для составляющих.
3. *Слияние* – полученные результаты для составляющих объединяются (сливаются) в единый результат.

Быстрая сортировка – пример подхода «разделяй и властвуй». Основные фазы: разделение и рекурсивные вызовы. Фаза слияния тривиальна и не требует дополнительных действий.

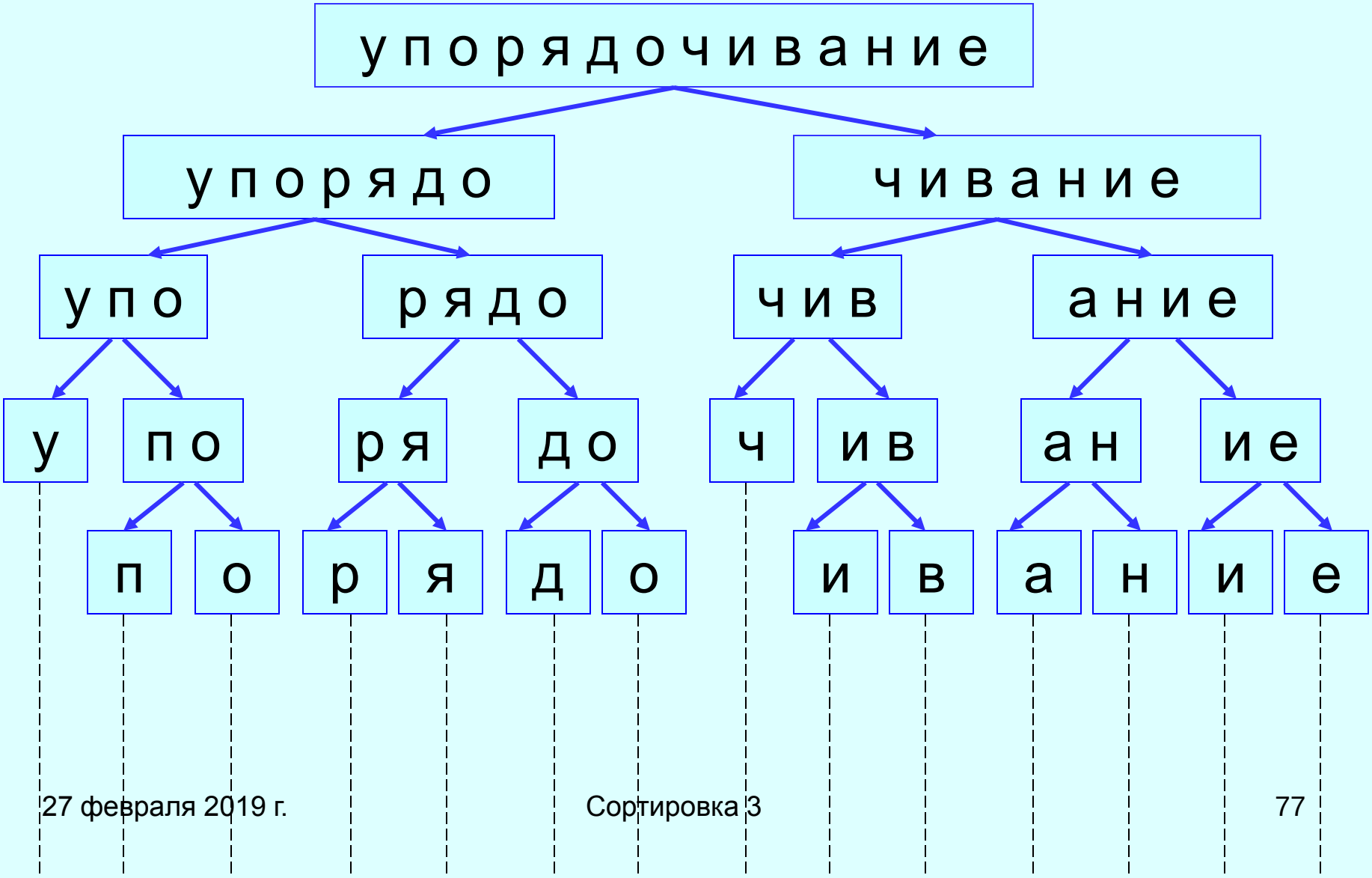
Сортировка слиянием

```
void MergeSort (array[] list; int n){

    if (n ≥ 2)
    {
      расщепить список L на два списка, выделив в
    L1        первые ⌊n/2⌋ элементов и сохранив в L
    следующие ⌈n/2⌉ элементов;
      MergeSort ( L1, ⌊n/2⌋ );
      MergeSort ( L,  ⌈n/2⌉ );
      Merge (L, ⌈n/2⌉, L1, ⌊n/2⌋)    {Слияние L := L+L1}
    }
}
```

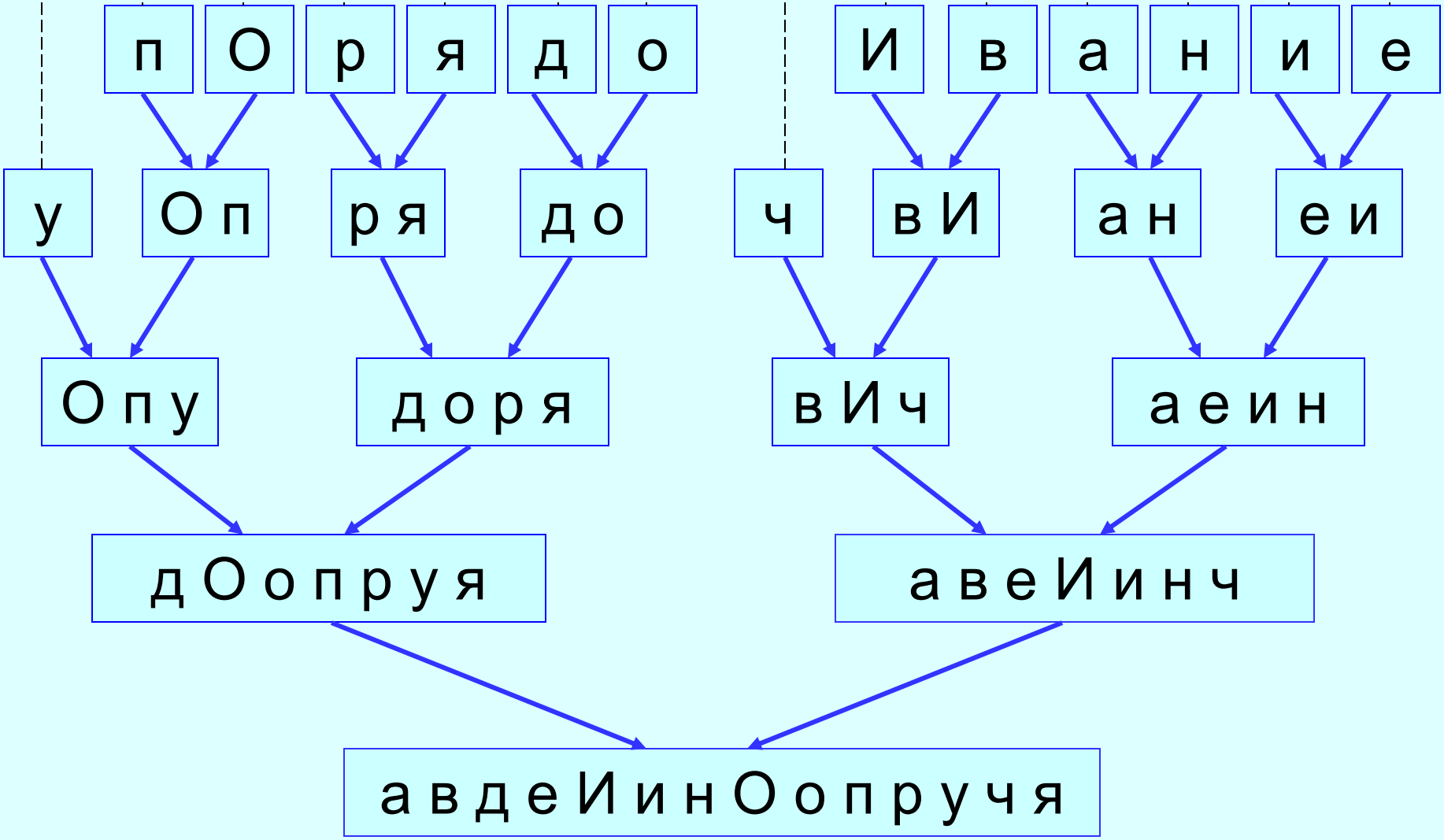
Сортировка слиянием

Пример (разделение и рекурсивные вызовы)



Сортировка слиянием

Пример (слияния и рекурсивные возвраты)



Сортировка слиянием

Анализ сложности

(1) Высота дерева $\lceil \log_2 n \rceil$.

На каждом ярусе не более Cn операций.

Всего не более $Cn\lceil \log_2 n \rceil$, т.е. $O(n \log_2 n)$.

(2) Более формально (рекуррентное соотношение)

$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + u(n); \quad T(1) = b.$

Разделение+Слияние: $u(n) \le an$;

Тогда $T(n) = O(n \log_2 n).$

Сортировка слиянием

Анализ сложности

(2a) Итеративное решение

Пусть $n = 2^q$ (для простоты).

Тогда $T(n) = 2T(n/2) + an$; $T(1) = b$.

$T(n) = 2(2T(n/2^2) + an/2) + an = 2^2T(n/2^2) + 2an =$

$= 2^2(2T(n/2^3) + an/2^2) + 2an = 2^3T(n/2^3) + 3an =$

$= \ldots = 2^kT(n/2^k) + kan;$

При $k=q$: $T(n/2^k) = T(1) = b$, $2^k = n$, $q = \log_2 n$

$T(n) = 2^kT(n/2^k) + kan = bn + an \log_2 n \le Cn \log_2 n$,

где C = a + b (например).

Т.о. $T(n) = O(n \log_2 n)$.

Сортировка слиянием

Анализ сложности

(2б) По индукции покажем, что решение

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + u(n); \quad T(1) = b$$

при $u(n) = O(n)$ есть $T(n) = O(n \log_2 n)$.

Пусть $u(n) \le an$ и $T(k) \le Ck \log_2 k$ при $k < n$.

$$T(n) \le C \lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor + C \lceil n/2 \rceil \log_2 \lceil n/2 \rceil + an \le$$

$$\lfloor x \rfloor \le x, \qquad \lceil x \rceil < x + 1$$

$$\le C \lfloor n/2 \rfloor \log_2 (n/2) + C \lceil n/2 \rceil \log_2 (n/2 + 1) + an \le$$

Анализ сложности (2б - продолжение)

$$\le\ C\lfloor n/2\rfloor \log_2(n/2)+ C\lceil n/2\rceil \log_2 (n/2)(1 + 2/n) + an \le$$

$$\le C\lfloor n/2\rfloor \log_2(n/2)+ C\lceil n/2\rceil \log_2 (n/2)+C\lceil n/2\rceil \log_2(1 + 2/n) + an \le$$

$$\log_2 (x +1) \le x,\ \lfloor n/2\rfloor + \lceil n/2\rceil = n$$

$$\le C\, n \log_2(n/2)+ C\lceil n/2\rceil (2/n) + an \le$$

$$\le C\, n \log_2(n/2)+ C\ (n/2+1)\ (2/n) + an \le$$

$$\le C\, n \log_2 n + C\ (1 + 2/n) + (a - C)n \le \underline{C\, n \log_2 n}\ \ (\text{при } C > a)$$

Сортировка слиянием

Восхоящее слияние

у п | о р | я д | о ч | и в | а н | и е

о п р у | д о ч я | а в и н | и е

д о о п р у ч я | а в е и и н

а в д е и и н о о п р у ч я

Реализация на структурах данных:

на списках, на файлах, на массивах

Пример действий для массива a[0]... a[7]

```
44   55   12   42   94   18   06   67        исходный массив
44   55   12   42   67   18   06  |94          94 <-> 67
44   55   12   42   06   18  |67   94          67 <-> 06
44   18   12   42   06  |55   67   94          55 <-> 18
06   18   12   42  |44   55   67   94          44 <-> 06
06   18   12  |42   44   55   67   94          42 <-> 42
06   12  |18   42   44   55   67   94          18 <-> 12
06  |12   18   42   44   55   67   94          12 <-> 12
```

Пирамидальная сортировка (HeapSort)

Бинарное дерево является ***пирамидой*** (Heap), если у каждого узла дерева нет сыновей с ключами большими (меньшими), чем ключ в этом узле.

Инвариант пирамиды *H*:

$$(\forall\ b \in H:$$

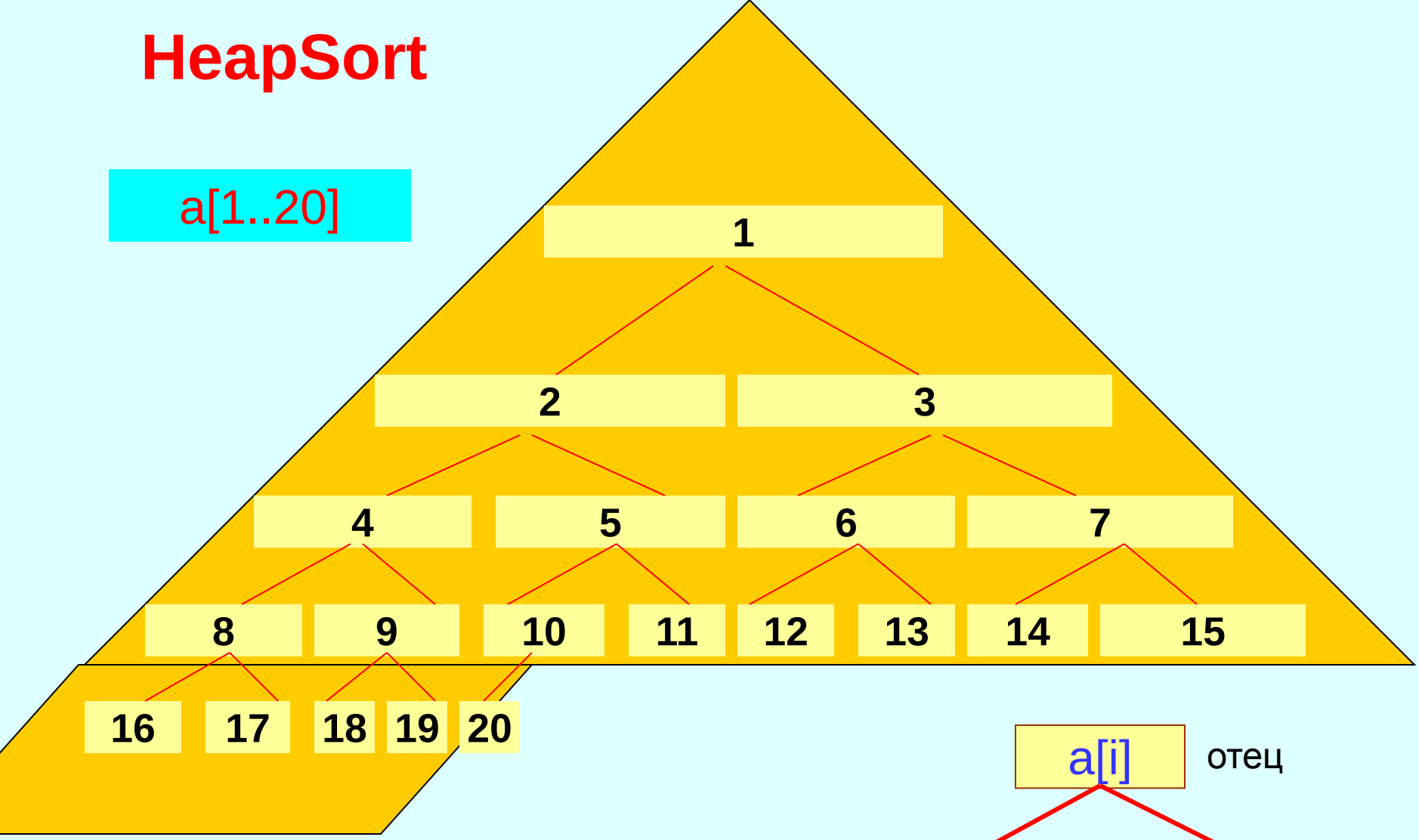
$$((\textbf{not}\ Null(Left(b)) \rightarrow (Root(Left(b)).key \leq Root(b).key\ )$$

$$\&$$

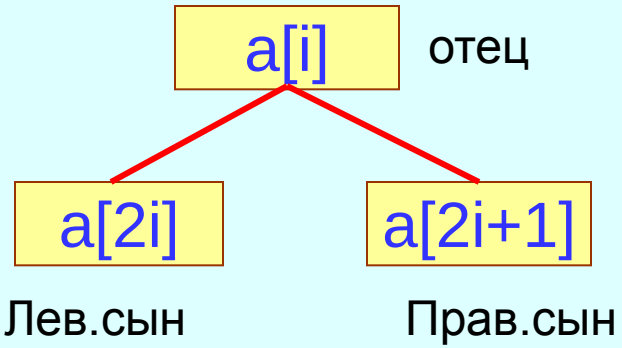
$$((\textbf{not}\ Null(Right(b)) \rightarrow (Root(Right(b)).key \leq Root(b).key\ ))$$

HeapSort

a[1..20]



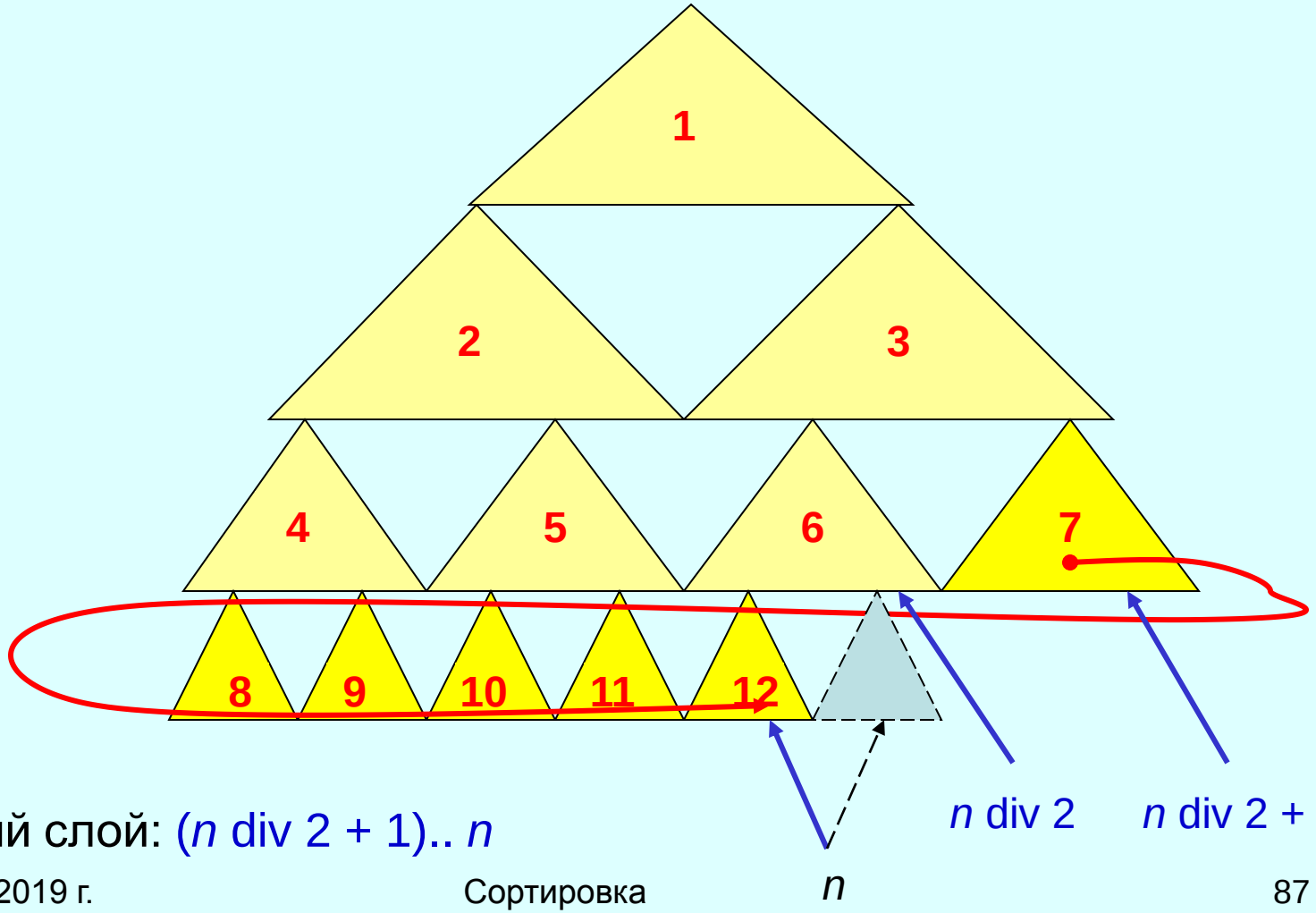
$(1 < i \le n) \rightarrow (a [i \textbf{ div } 2] \ge a [i])$



Размещение «ровной» пирамиды в массиве по ярусам

Heap Sort

Первая фаза - *построение пирамиды*

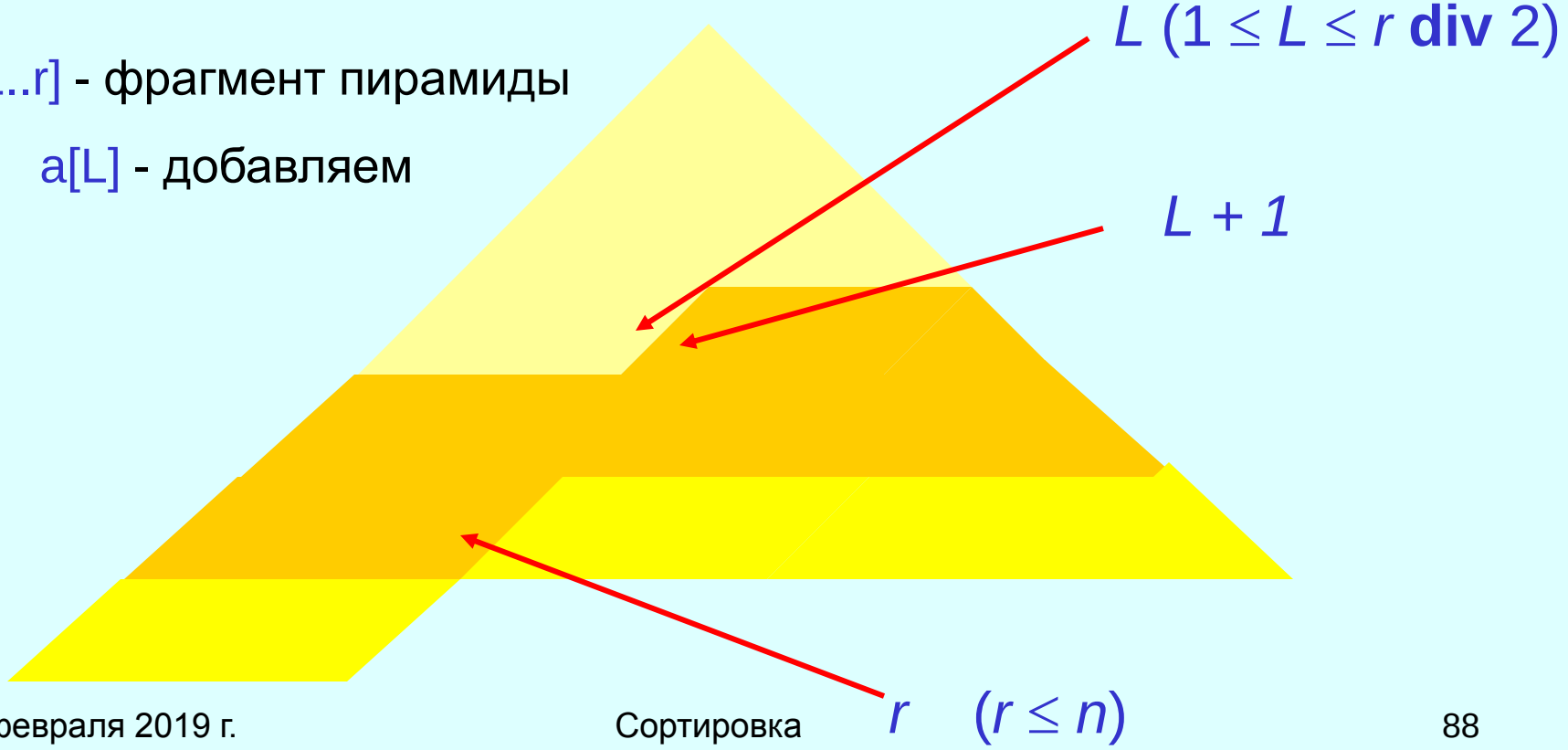


Нижний слой: (*n* div 2 + 1).. *n*

Первая фаза - *построение пирамиды*

1. Считаем, что нижний слой a[(*n* div 2 + 1).. *n*] имеется.
2. Добавляем элементы в пирамиду: **for(** *i* = *n* **div** 2; i<0; i--

Для очередного добавления



Построение пирамиды

Heap(p, q) ≡

(∀i∈p..(q **div** 2): (a[i] ≥ a[2i]) & (a[i] ≥ a[2i+1])*)

*при 2i+1 ≤ q

procedure Sift (a: mass; L, r: index1);

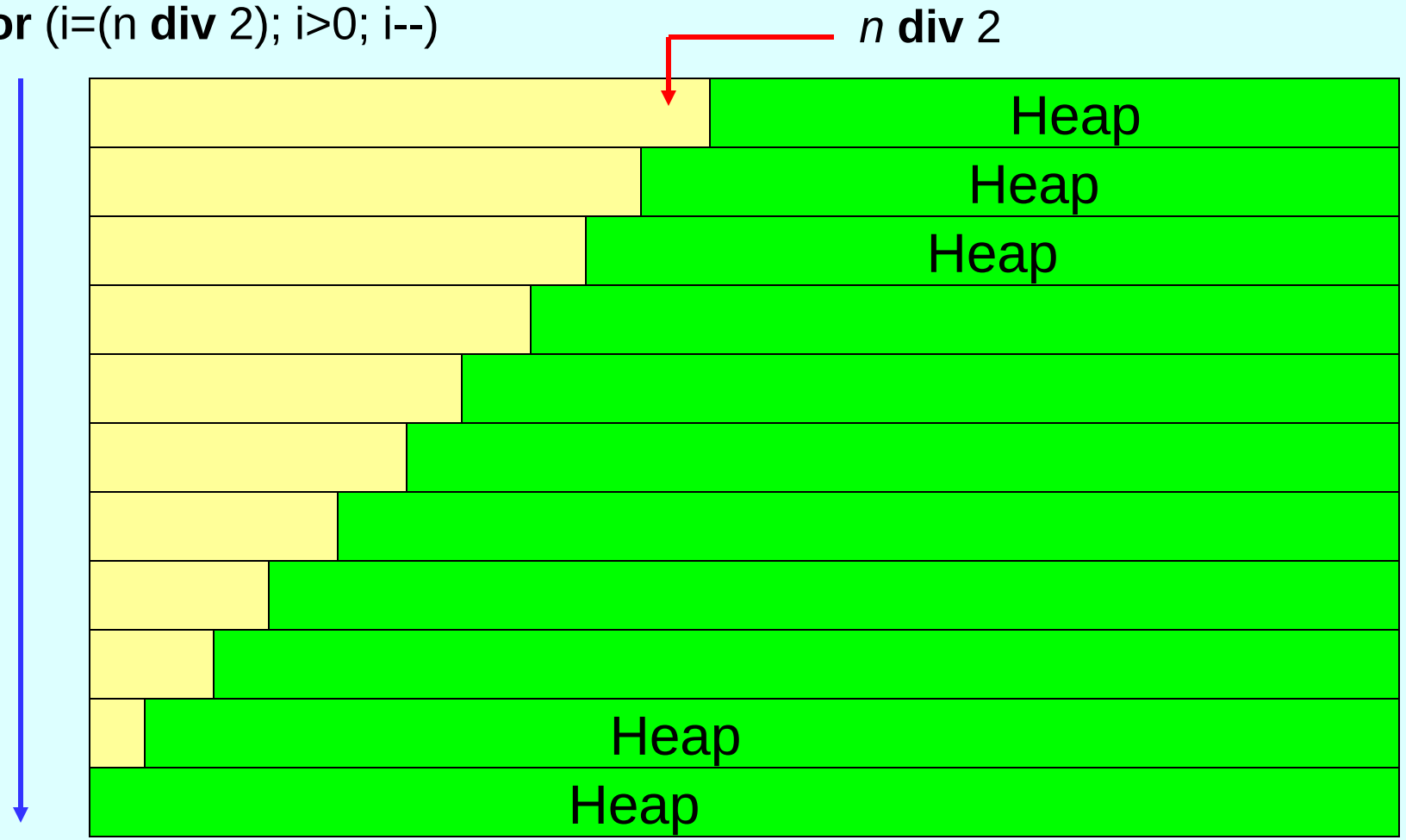
Предусловие: Heap(L+1, r)

Постусловие: Heap(L, r)

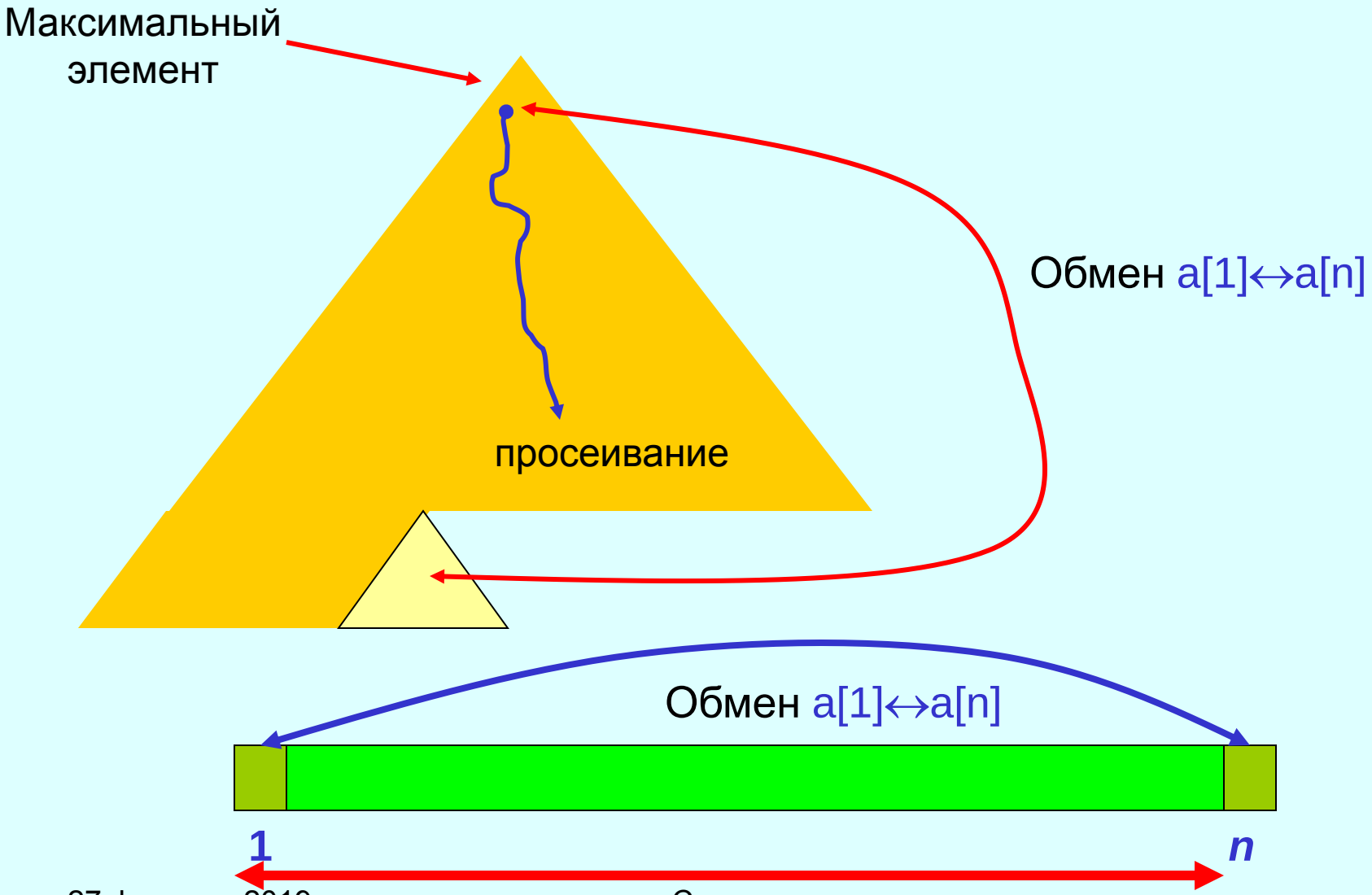
Если для a[L] свойство пирамиды не выполняется, то обменять a[L] с max(a[2L], a[2L+1]*) и т.д.

Фаза 1: формирование пирамиды

for (i=(n **div** 2); i>0; i--)

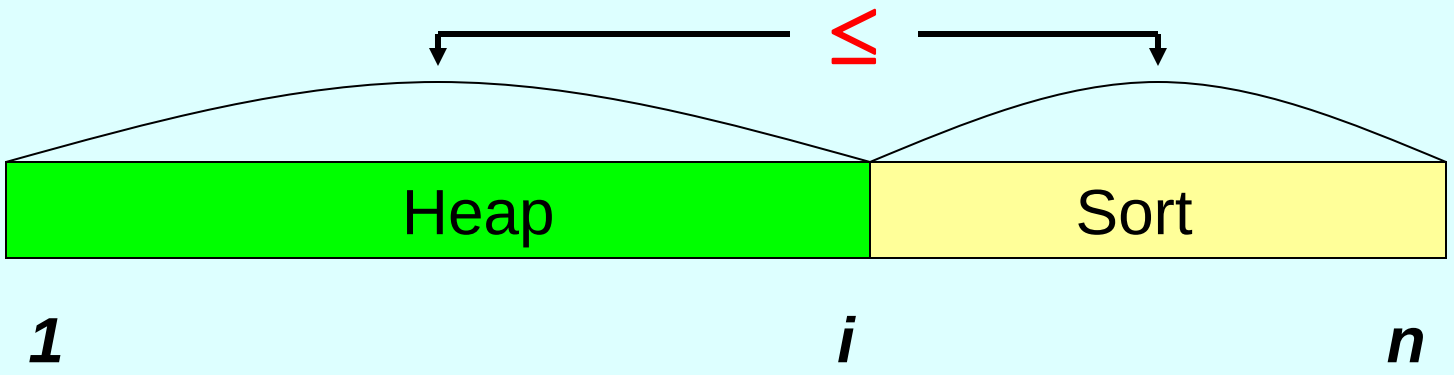


Фаза 2: полное упорядочение

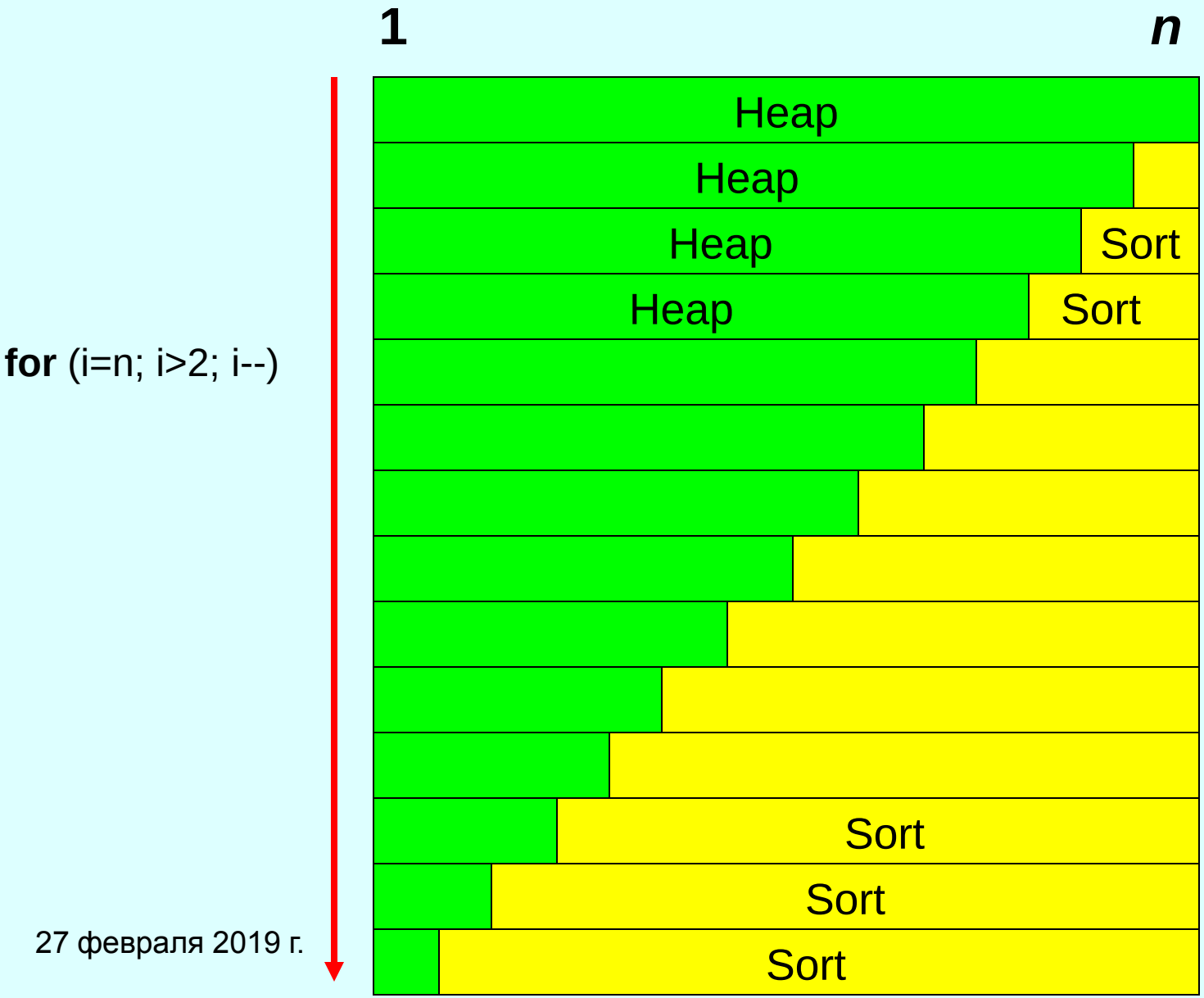


Фаза 2: полное упорядочение

```
for( i=n; i>2; i--)
 {
  x := a[1]; a[1] := a[i]; a[i] := x;
  Sift( a, 1, i-1 )
 };
```

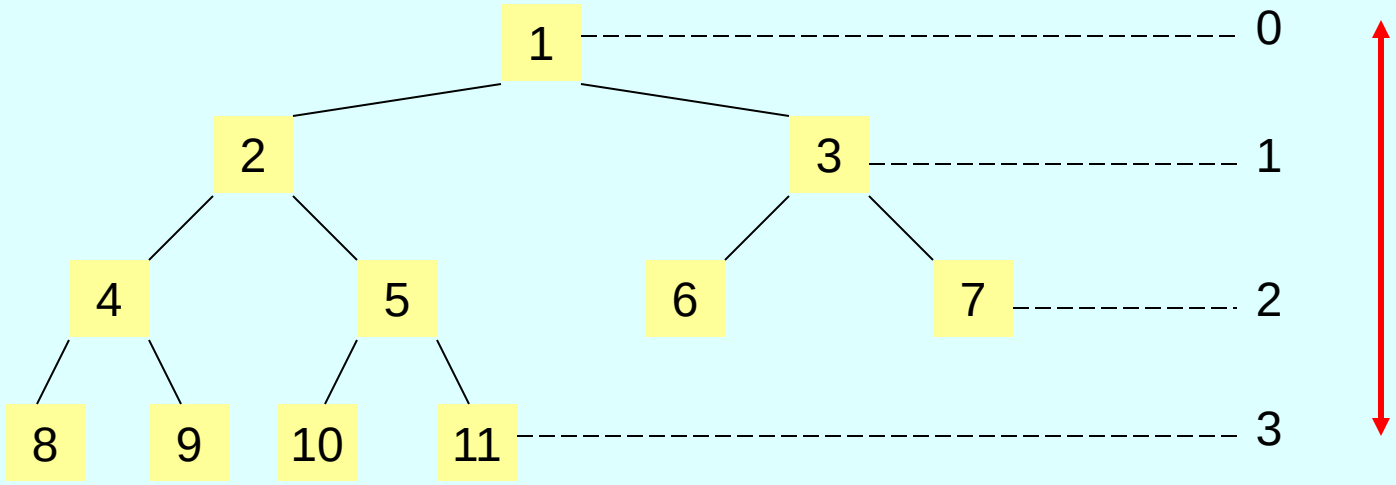


Фаза 2: полное упорядочение



Анализ алгоритма

Высота пирамиды = $\lfloor \log_2 n \rfloor$



Фаза построения пирамиды:

$$\sum_{i=1}^{\lfloor n/2 \rfloor} \lfloor \log_2 (n-i) \rfloor = O(n \log_2 n)$$

более точно $O(n)$

Оценка $O(n)$ сложности фазы построения пирамиды

Общее количество шагов внутреннего цикла (внутри процедуры Sift) ≤

(i – номер уровня, h – высота дерева, $h = \lfloor \log_2 n \rfloor$, $h \le \log_2 n$, число узлов на уровне i равно 2^i)

$$\le \sum_{i=0}^{h-1} 2^i (h-i) =$$

$$= h\sum_{i=0}^{h-1} 2^i - \sum_{i=0}^{h-1} i2^i = ?$$

$$= h(2^h - 1) - ((h-2)2^h + 2) =$$

$$= 2 \times 2^h - 2 - h \le 2n - \log_2 n - 2 =$$

$$= O(n)$$

$$\sum_{i=0}^{m} 2^i = 2^{m+1} - 1$$

$$\sum_{i=1}^{m} i2^i = (m-1)2^{m+1} + 2$$

$$2^h \le n$$

$$h \le \log_2 n$$

Анализ алгоритма

Высота пирамиды = $\lfloor \log_2 n \rfloor$

Этап (фаза) полного упорядочивания

$$\sum_{i=2}^{n} \lfloor \log_2 (i-1) \rfloor = \sum_{i=1}^{n-1} \lfloor \log_2 i \rfloor \le \sum_{i=1}^{n-1} \log_2 i \; =$$

$$= O(n \log_2 n)$$

Всего в худшем случае $O(n \log_2 n)$

Код функции сортировки

```
template<class T>
void downHeap(T a[], long k, long n) {
  //  процедура просеивания следующего элемента
  //  До процедуры: a[k+1]...a[n]  - пирамида
  //  После:  a[k]...a[n]  - пирамида
  T new_elem;
  long child;
  new_elem = a[k];

  while(k <= n/2) {          // пока у a[k] есть дети
    child = 2*k;
    //  выбираем большего сына
    if( child < n && a[child] < a[child+1] )
    child++;
    if( new_elem >= a[child] ) break;
    // иначе
    a[k] = a[child];     // переносим сына наверх
    k = child;
  }
  a[k] = new_elem;
}
```

Нижняя граница задачи сортировки

Вопрос: существует ли алгоритм сортировки, основанный на сравнениях и решающий задачу за меньшее число операций,

чем $O(n \log n)$ при любых входных данных?

Сортировка с минимальным числом сравнений

(избыточные сравнения не выполняются)

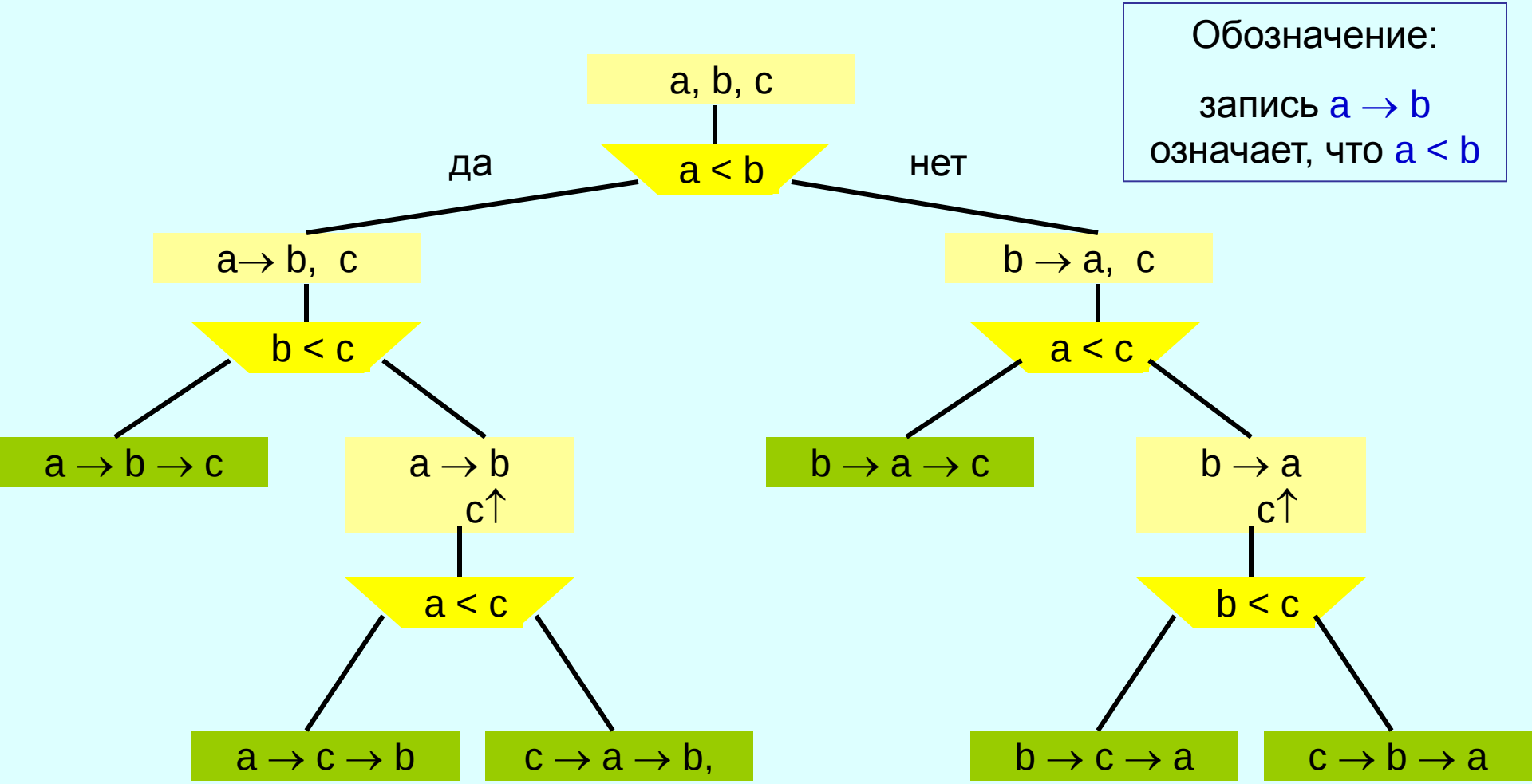
Деревья решений (сравнений)

Для простоты – все ключи различны.

Основная операция сравнения: $k_i < k_j$

Сортировка с минимальным числом сравнений

Деревья решений (сравнений)

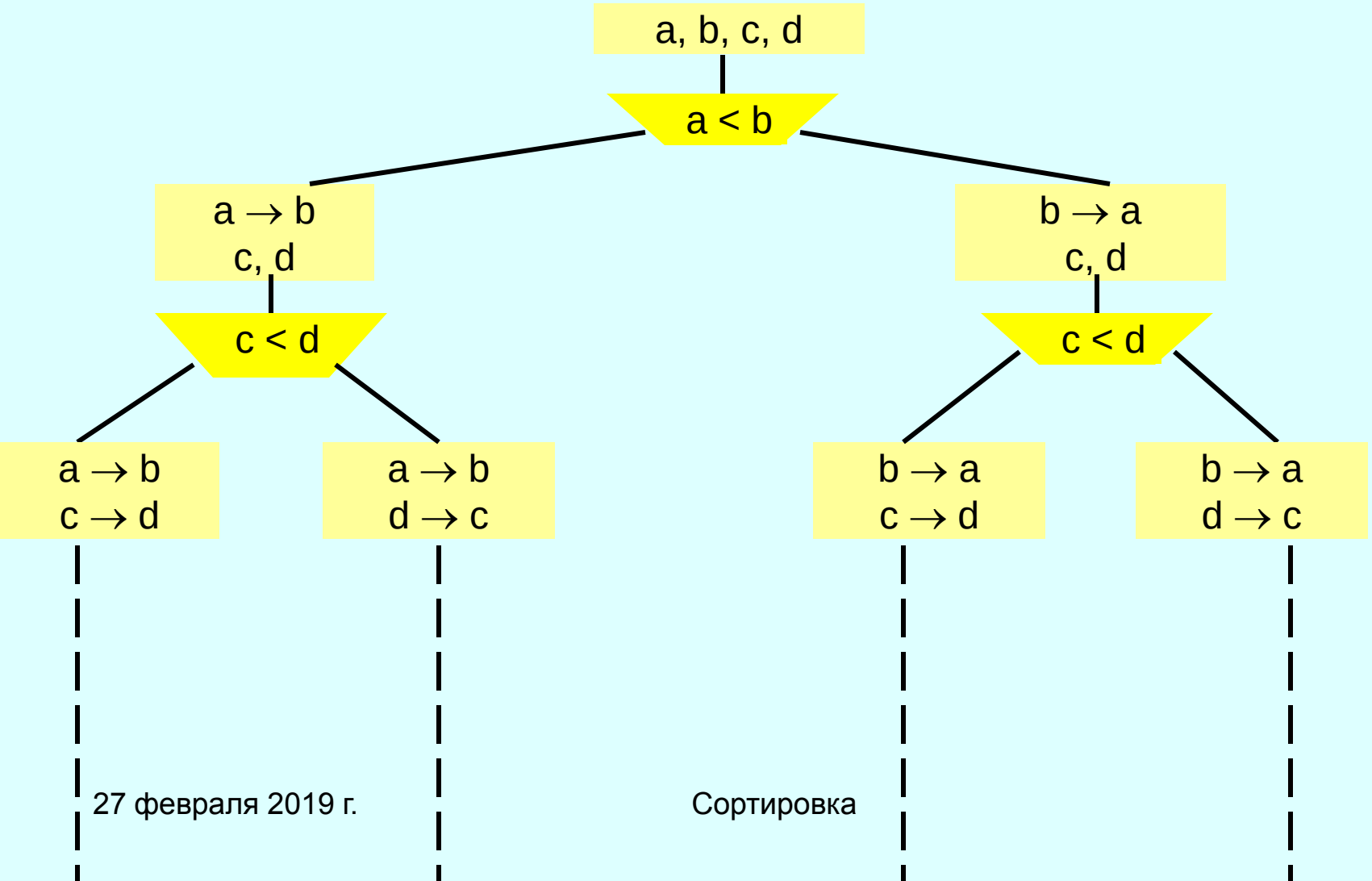


$n = 3$, число перестановок и листьев $= 3! = 6$

Сортировка с минимальным числом сравнений

Деревья решений (сравнений)

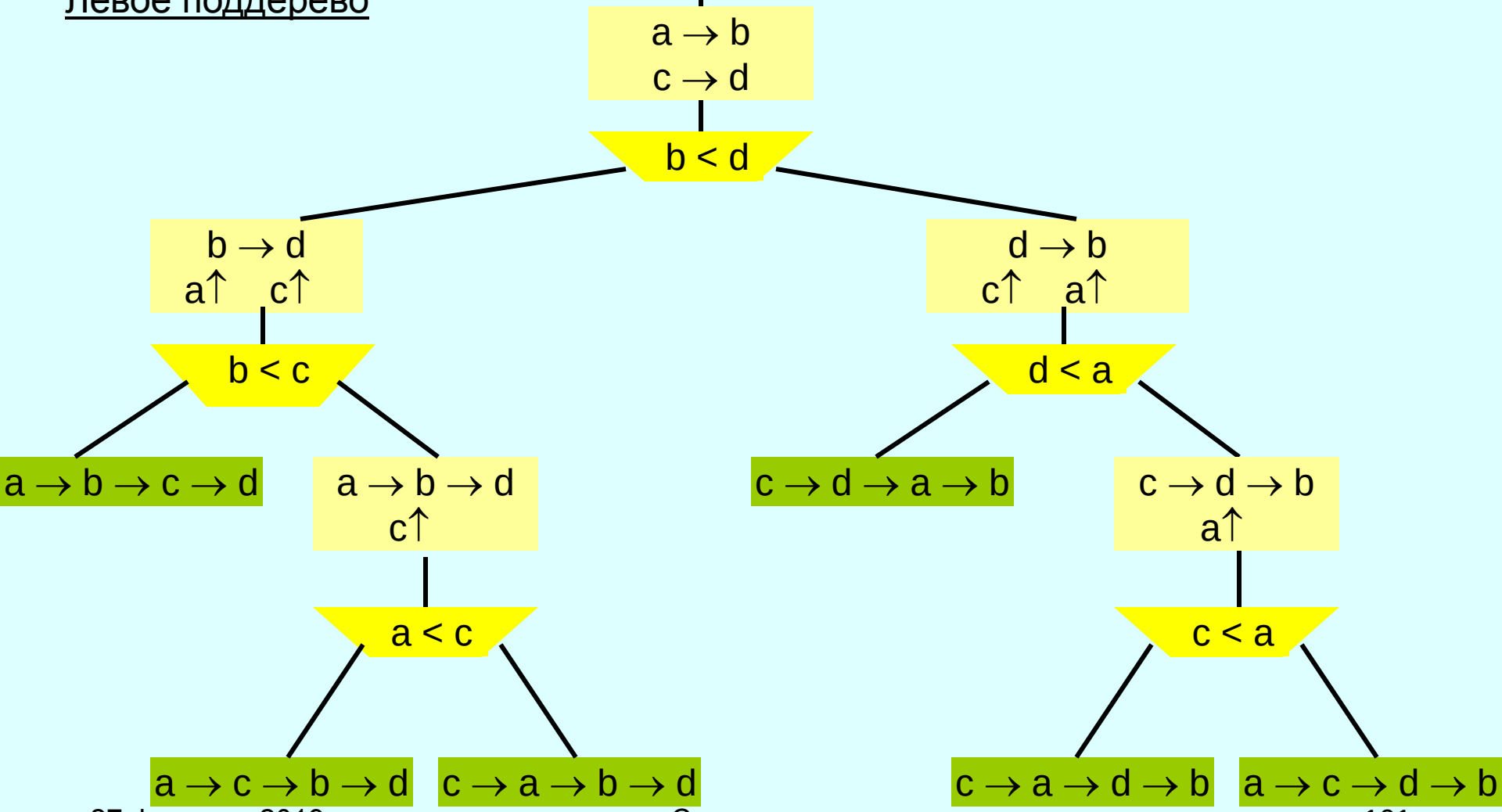
$n = 4$, число перестановок и листьев $= 4! = 24$



Деревья решений (сравнений)

$n = 4$, число перестановок и листьев $= 4! = 24$

Левое поддерево



Деревья решений (сравнений)

Пусть m высота дерева (максимальное число сравнений).

Тогда $2^m \geq$ *число листьев* $\geq n!$

(число листьев должно быть не менее числа исходов)

$$m \geq \lceil \log_2 n! \rceil$$

Итак,

утверждение:

Для любого алгоритма сортировки, основанного на сравнениях, всегда можно подобрать такие исходные данные, что применение алгоритма потребует не менее $\lceil \log_2 n! \rceil$ шагов.

Формула Стирлинга

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

$$\lceil \log_2 n! \rceil \le \log_2 n! + 1$$

$$\lceil \log_2 n! \rceil = n \log_2 n - n \log_2 e + (1/2) \log_2 n + O(1)$$

Итак, **сложность задачи** сортировки есть $\Omega(n \log_2 n)$,

а алгоритмы быстрой сортировки и пирамидальной сортировки решают задачу за время $\Theta(n \log_2 n)$

Оптимальная сортировка

(оптимальная по числу сравнений и перемещений)

Бинарные вставки: $B(n) = \sum_{k=1}^{n} \lceil \log_2 k \rceil = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$

n	2	3	4	5	6	7	8	9	12	16
$\lceil \log_2 n! \rceil$	1	3	5	7	10	13	16	19	29	45
B(n)	1	3	5	8	11	14	17	21	33	49

Пусть $S(n)$ – минимальное число сравнений, достаточное для сортировки. $S(n) \geq \lceil \log_2 n! \rceil$. ? $S(n) = \lceil \log_2 n! \rceil$?

Например, $S(5) = 7$ (см.далее), а $S(12) = 30$ (!)

Оптимальная сортировка

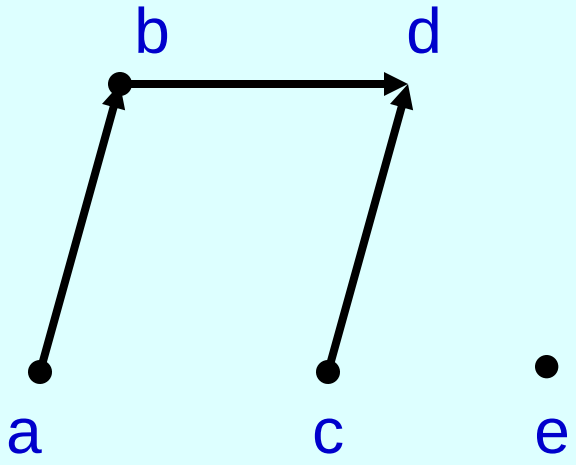
Пример: $S(5) = 7$?

k_1, k_2, k_3, k_4, k_5

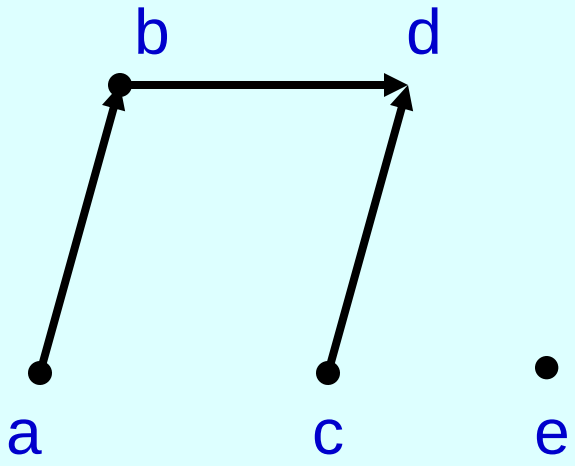
1-е сравнение: $k_1 : k_2 \rightarrow \max(k_1, k_2)$

2-е сравнение: $k_3 : k_4 \rightarrow \max(k_3, k_4)$

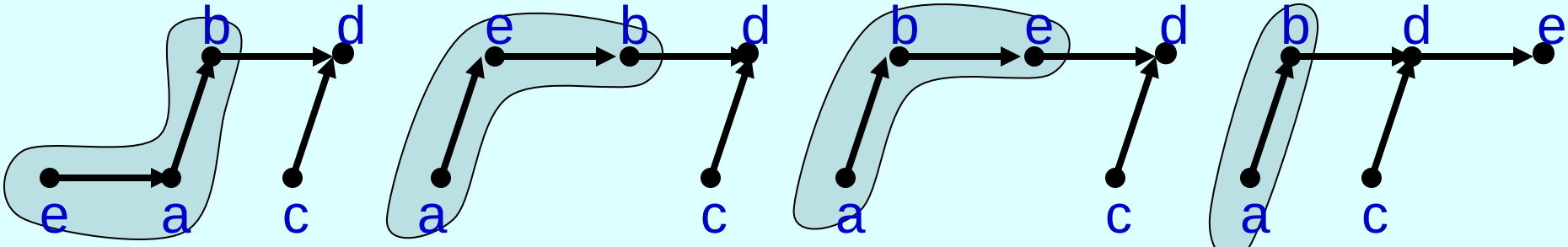
3-е сравнение: $\max(k_1, k_2) : \max(k_3, k_4)$



Оптимальная сортировка



4 и 5-е сравнения: e среди a<b<d



Поразрядная сортировка

(Распределяющая сортировка)

Пример 1 (с картами).

Карта = (масть, достоинство)

***Масть*: ♣ - трефа, ♦ - бубна, ♥ - черва, ♠ - пика**

***Достоинство*: 2, 3, 4, 5, 6, 7, 8, 9, 10, В, Д, К, Т**

Порядок по масти: ♣ < ♦ < ♥ < ♠.

Порядок по достоинству: 2<3<4<5<6<7<8<9<10<В<Д<К<Т

Порядок карт *лексикографический*:

♣2<♣3<♣4<♣5<♣6<♣7<♣8<♣9<♣10<♣В<♣Д<♣К<♣Т<

♦2<♦3<♦4<♦5<♦6<♦7<♦8<♦9<♦10<♦В<♦Д<♦К<♦Т<

♥2<♥3<♥4<♥5<♥6<♥7<♥8<♥9<♥10<♥В<♥Д<♥К<♥Т<

♠2<♠3<♠4<♠5<♠6<♠7<♠8<♠9<♠10<♠В<♠Д<♠К<♠Т

Поразрядная сортировка

Сортировка колоды карт в лексикографическом порядке:

1 *шаг* – распределяем по достоинству (раскладываем) на кучки («лицом» вверх)

2	3	4	5	6	7	8	9	10	В	Д	К	Т

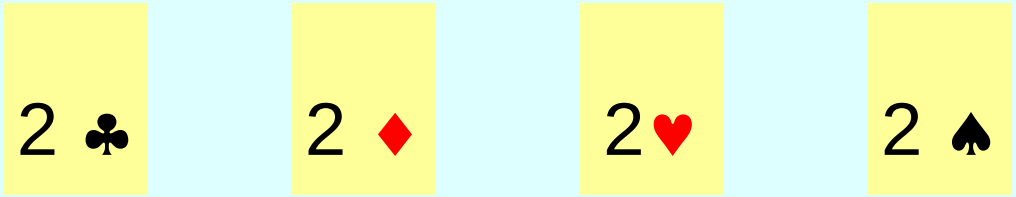
Затем складываем кучки друг на друга (последовательно кучки большего достоинства сверху).

2 шаг – держим колоду рубашкой вниз и распределяем на кучки по масти (карты кладем лицом вверх).

При этом в каждой «кучке по масти» карты лягут в порядке возрастания достоинства вниз кучки.

Поразрядная сортировка

В итоге получим 4 кучки с верхними картами: ♣ < ♦ < ♥ < ♠



Заключительный шаг – последовательно складываем, начиная справа, левую кучку на правую.

Задание: найти колоду карт и поупражняться

Поразрядная сортировка

Пример 2 (с числами).

Рассмотрим целые (положительные) *трехразрядные* числа

в позиционной ***шестиричной*** системе счисления,

т.е. каждый разряд содержит цифры 0, 1, 2, 3, 4, 5

Например, пусть дана последовательность из 9 чисел

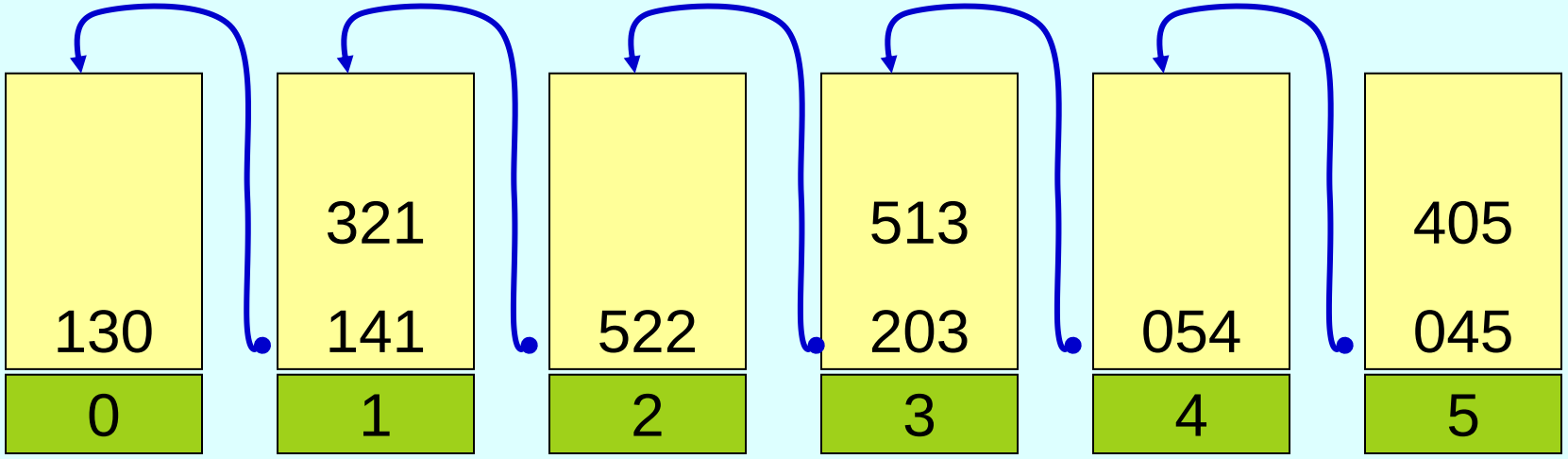
203, 045, 141, 405, 321, 522, 130, 054, 513

1 *шаг* – распределяем числа последовательности сначала по младшей цифре по ящикам с «именами»: «0», «1», «2», «3», «4», «5».

Ящики заполняются снизу вверх.

Поразрядная сортировка

203, 045, 141, 405, 321, 522, 130, 054, 513



Сцепляем последовательно слева направо содержимое ящиков в одну последовательность, так, что нижнее число правого ящика следует за верхним числом левого соседнего ящика (внутри ящиков последовательность расположена снизу вверх). Таким образом, получим последовательность

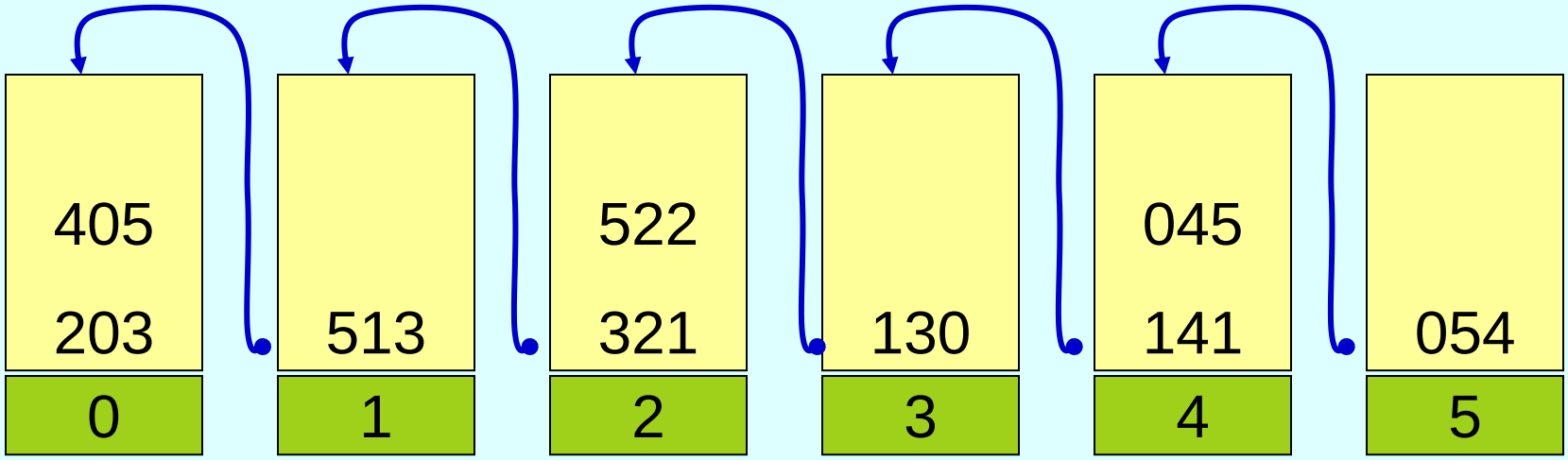
130, 141, 321, 522, 203, 513, 054, 045, 405.

Заметим, что младшие цифры этой последовательности упорядочены по неубыванию.

Поразрядная сортировка

130, 141, 321, 522, 203, 513, 054, 045, 405

2 шаг – аналогичным образом распределяем по ящикам числа полученной последовательности *по средней цифре.*



Повторяя сцепление содержимого ящиков, получим последовательность

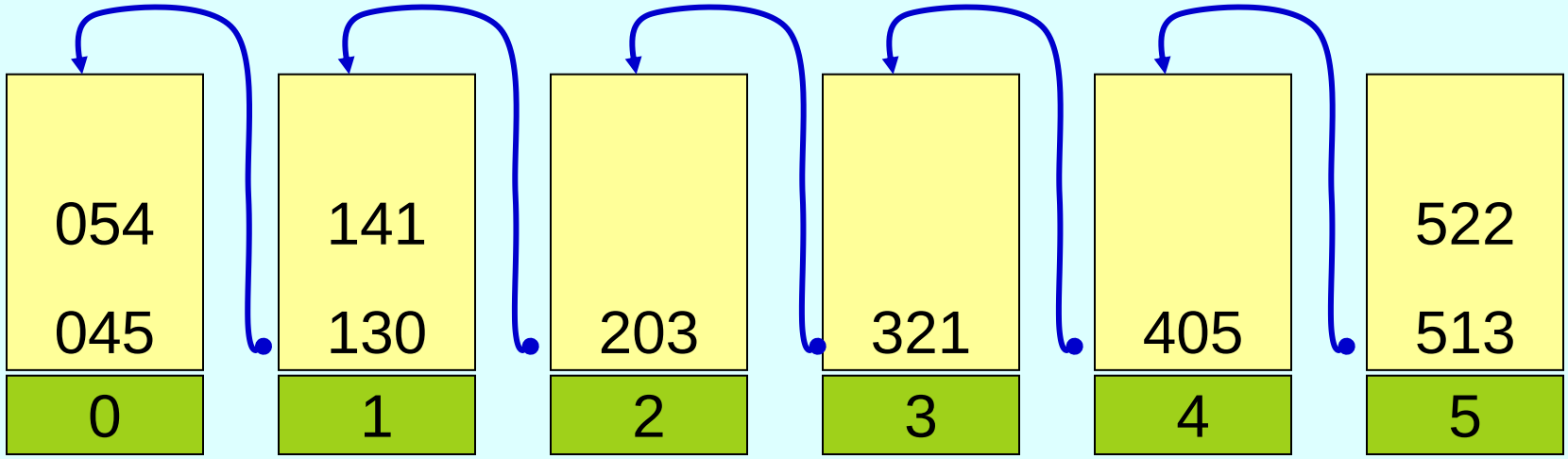
2<u>03</u>, 4<u>05</u>, 5<u>13</u>, 3<u>21</u>, 5<u>22</u>, 1<u>30</u>, 1<u>41</u>, 0<u>45</u>, 0<u>54</u>,

упорядоченную по двум младшим цифрам.

Поразрядная сортировка

203, 405, 513, 321, 522, 130, 141, 045, 054

3 шаг – аналогичным образом распределяем по ящикам числа полученной последовательности *по старшей цифре.*



Повторяя сцепление содержимого ящиков, получим упорядоченную последовательность

045, 054, 130, 141, 203, 321, 405, 513, 522.

Алгоритм поразрядной сортировки

В общем случае даны числа (ключи):

$$x_1, x_2, x_3, \ldots, x_n.$$

Каждый ключ состоит из p разрядов:

$$(\forall\, i \in 1..n: x_i = (x_i(p), x_i(p-1), \ldots, x_i(1))),$$

а каждый q-й разряд представлен цифрой $x_i(q) \in 0..r-1$.

Тогда

$$(x_i < x_j) \leftrightarrow$$

$$(\exists\, t \in 1..p: (\forall\, q \in t+1..p: x_i(q) = x_j(q)) \,\&\, (x_i(t) < x_j(t))).$$

Алгоритм поразрядной сортировки

Для представления «ящиков» и их сцепления в новую последовательность используем ***очереди***:

- Q – общая очередь (исходная, промежуточные и результирующая последовательности);
- $q[0], q[1], \ldots, q[r-1]$ – очереди по каждой r-ичной цифре («ящики»).

Алгоритм поразрядной сортировки

Алгоритм:

Алгоритм поразрядной сортировки

Подсчитаем количество операций.

Внешний цикл по разрядам выполняется p раз.

Каждая *итерация* этого цикла для каждого из n чисел выполняет извлечение и запись в очередь, т.е. всего $2n$ таких операций.

Кроме того, на каждой итерации происходит $r - 1$ сцепление очередей.

Таким образом на одной итерации требуется $2n + r - 1$ операций,

а всего по всем разрядам $O(p\,n + p\,r)$ операций.

Сравнение с $O(n \log n)$!...

Определения

- **Алфавит** – конечное множество символов
- **Строка (слово)** – это последовательность символов из некоторого алфавита. *Длина строки* – количество символов в строке
- Строка X называется **подстрокой** строки Y, если найдутся такие строки Z1 и Z2, что Y=Z1XZ2.
  При этом Z1 называют *левым*, а Z2 – *правым* крылом подстроки. Подстрокой может быть и сама строка. Иногда при этом строку X называют *вхождением* в строку Y.
  Например, строки hrf и fhr является подстроками строки abhrfhr.

Определения(продолжение)

- Подстрока X называется **префиксом** строки Y, если есть такая подстрока Z, что Y=XZ. Причем сама строка является префиксом для себя самой (так как найдется нулевая строка L, что X=XL).
  Например, подстрока ab является префиксом строки abcfa.
- Подстрока X называется **суффиксом** строки Y, если есть такая подстрока Z, что Y=ZX. Аналогично, строка является суффиксом себя самой.
  Например, подстрока bfg является суффиксом строки vsenfbfg

Постановка задачи

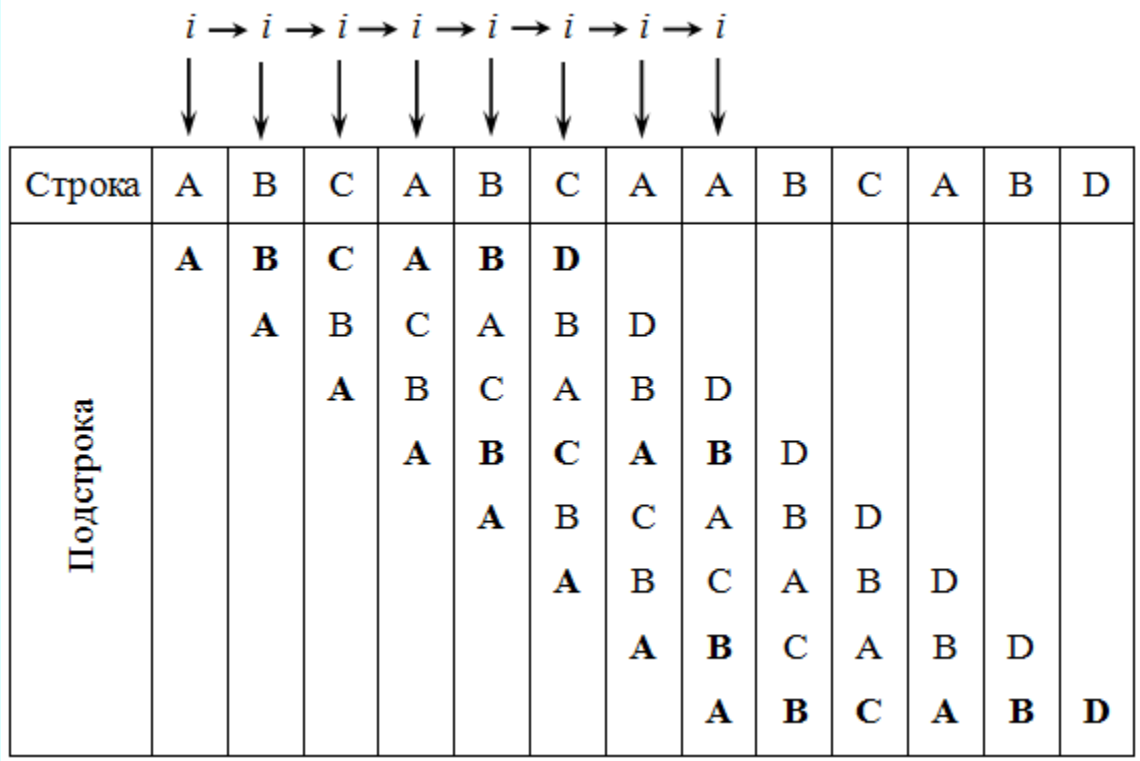
- Проверить, входит ли заданная подстрока в данную строку.
- Если входит, то найдем номер, начиная с какого символа строки, то есть, определим первое вхождение заданной подстроки в исходной строке.

Прямой поиск

Основная идея алгоритма прямым поиском заключается в посимвольном сравнении строки с подстрокой.

- В начальный момент происходит сравнение первого символа строки с первым символом подстроки, второго символа строки со вторым символом подстроки и т. д. Если произошло совпадение всех символов, то фиксируется факт нахождения подстроки. В противном случае производится сдвиг подстроки на одну позицию вправо и повторяется посимвольное сравнение

Прямой поиск(продолжение)



Прямой поиск(реализация)

```
//Описание функции прямого поиска подстроки в строке
int DirectSearch(char *string, char *substring){
  int sl, ssl;
  int res = -1;
  sl = strlen(string);
  ssl = strlen(substring);
  if ( sl == 0 )
    cout << "Неверно задана строка\n";
  else if ( ssl == 0 )
    cout << "Неверно задана подстрока\n";
  else
    for (int i = 0; i < sl - ssl + 1; i++)
      for (int j = 0; j < ssl; j++)
        if ( substring[j] != string[i+j] )
          break;
        else if ( j == ssl - 1 ){
          res = i;
          break;
        }
  return res;
}
```

Прямой поиск(итоги)

Данный алгоритм является малозатратным и не нуждается в предварительной обработке и в дополнительном пространстве. Большинство сравнений алгоритма прямого поиска являются лишними.

Поэтому в худшем случае алгоритм будет малоэффективен, так как его сложность будет пропорциональна $O((n-m+1)*m)$, где n и m – длины строки и подстроки соответственно.

Алгоритм Кнута, Морриса и Пратта

Данный алгоритм является малозатратным и не нуждается в предварительной обработке и в дополнительном пространстве. Большинство сравнений алгоритма прямого поиска являются лишними.

Поэтому в худшем случае алгоритм будет малоэффективен, так как его сложность будет пропорциональна $O((n-m+1)*m)$, где n и m – длины строки и подстроки соответственно.

КМП алгоритм

- алгоритм основывается на том, что после частичного совпадения начальной части подстроки с соответствующими символами строки фактически известна пройденная часть строки и можно, вычислить некоторые сведения, с помощью которых затем быстро продвинуться по строке.

КМП алгоритм(продолжение)

Основное отличие:

- сдвиг подстроки выполняется не на один символ, а на некоторое переменное количество символов.
- Для повышения эффективности алгоритма необходимо, чтобы сдвиг на каждом шаге был бы как можно большим

КМП алгоритм(продолжение)

Основное отличие:

- сдвиг подстроки выполняется не на один символ, а на некоторое переменное количество символов.
- Для повышения эффективности алгоритма необходимо, чтобы сдвиг на каждом шаге был бы как можно большим

КМП алгоритм(продолжение)

Если для произвольной подстроки определить все ее начала, одновременно являющиеся ее концами, и выбрать из них самую длинную (не считая, конечно, саму строку), то такую процедуру принято называть префикс-функцией.

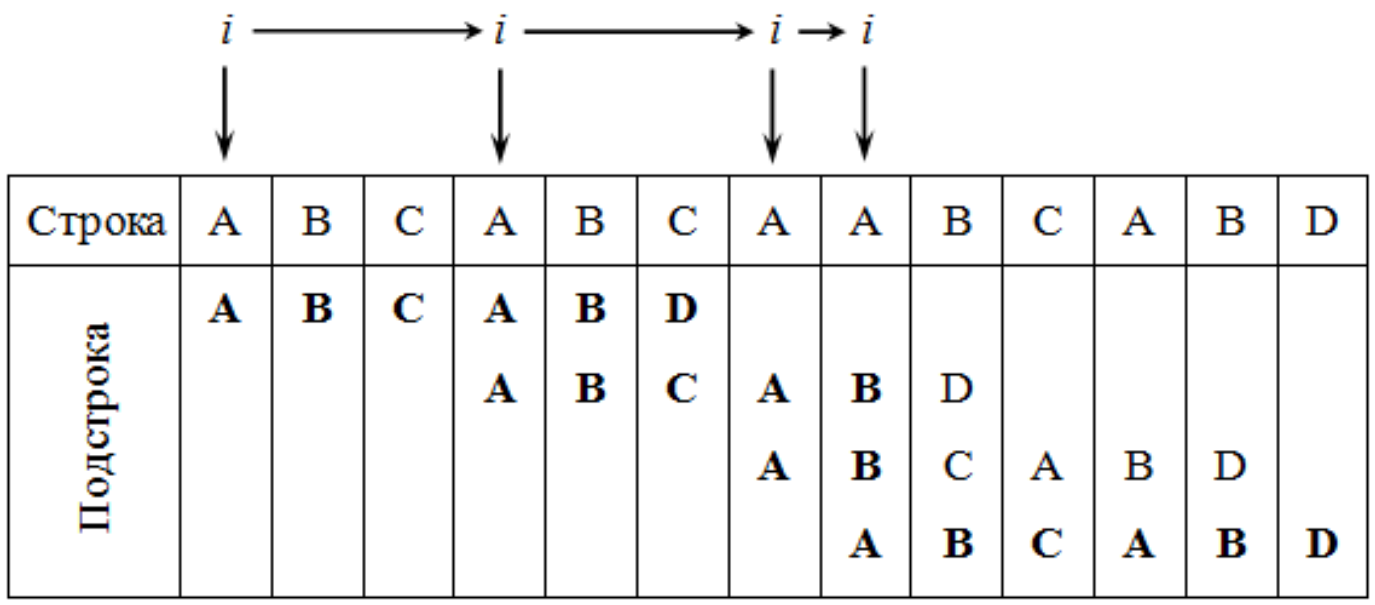
В реализации алгоритма КМП используется предобработка искомой подстроки, которая заключается в создании префикс-функции на ее основе.

При этом используется следующая идея: если префикс (он же суффикс) строки длиной i длиннее одного символа, то он одновременно и префикс подстроки длиной i-1.

Таким образом, проверяем префикс предыдущей подстроки, если же тот не подходит, то префикс ее префикса, и т.д.

находим наибольший искомый префикс!!!!

КМП алгоритм(продолжение)



Строка	A	B	C	A	B	C	A	A	B	C	A	B	D
Подстрока	**A**	**B**	**C**	**A**	**B**	**D**							
				A	**B**	**C**	**A**	**B**	D				
							A	**B**	C	A	B	D	
								A	**B**	**C**	**A**	**B**	**D**

КМП алгоритм(реализация)

```
//описание функции алгоритма Кнута, Морриса и Пратта
int KMPSearch(char *string, char *substring){
  int  sl, ssl;     int res = -1;
  sl = strlen(string);
  ssl = strlen(substring);
  if ( sl == 0 )
    cout << "Неверно задана строка\n";
  else if ( ssl == 0 )
    cout << "Неверно задана подстрока\n";
  else {
    int  i, j = 0, k = -1;   int  *d;
    d = new int[1000];       d[0] = -1;
    while ( j < ssl - 1 ) {
      while ( k >= 0 && substring[j] != substring[k] )
        k = d[k];
      j++;   k++;
      if ( substring[j] == substring[k] )
        d[j] = d[k];
      else
        d[j] = k;
    }
    i = 0;     j = 0;
    while ( j < ssl && i < sl ){
      while ( j >= 0 && string[i] != substring[j] )
        j = d[j];
      i++;  j++;
    }
    delete [] d;
    res =  j == ssl ? i - ssl : -1;
  }
  return res;
}
```

КМП алгоритм(итоги)

- Точный анализ рассматриваемого алгоритма весьма сложен.
- Д. Кнут, Д. Моррис и В. Пратт доказывают, что для данного алгоритма требуется порядка O(m+n) сравнений символов (где n и m – длины строки и подстроки соответственно), что значительно лучше, чем при прямом поиске.

Алгоритм Бойера и Мура

- Наиболее быстрый из алгоритмов
- Преимущество этого алгоритма в том, что необходимо сделать некоторые предварительные вычисления над подстрокой, чтобы сравнение подстроки с исходной строкой осуществлять не во всех позициях – часть проверок пропускаются как заведомо не дающие результата.

Алгоритм Бойера и Мура(продолжение)

- Первоначально строится таблица смещений для искомой подстроки.
- Далее идет совмещение начала строки и подстроки и начинается проверка с последнего символа подстроки.
- Если последний символ подстроки и соответствующий ему при наложении символ строки не совпадают, подстрока сдвигается относительно строки на величину, полученную из таблицы смещений,
  и снова проводится сравнение, начиная с последнего символа подстроки.
- Если же символы совпадают, производится сравнение предпоследнего символа подстроки и т.д.

Алгоритм Бойера и Мура(продолжение)

- Если какой-то (не последний) символ подстроки не совпадает с соответствующим символом строки, далее производим сдвиг подстроки на один символ вправо и снова начинаем проверку с последнего символа.
- Весь алгоритм выполняется до тех пор, пока либо не будет найдено вхождение искомой подстроки, либо не будет достигнут конец строки

Алгоритм Бойера и Мура(продолжение)

- Величина смещения для каждого символа подстроки зависит только от порядка символов в подстроке, поэтому смещения удобно вычислить заранее и хранить в виде одномерного массива, где каждому символу алфавита соответствует смещение относительно последнего символа подстроки

Алгоритм Бойера и Мура(продолжение)

Строка	A	B	C	A	F	D	F	A	B	C	A	B	D
Подстрока	A	B	C	A	**B**	**D** A	B	C **A**	A **B**	B **C**	**D** **A**	**B**	**D**

Алгоритм Бойера и Мура(продолжение)

- Алгоритм Бойера и Мура на хороших данных очень быстр, а вероятность появления плохих данных крайне мала. Поэтому он оптимален в большинстве случаев, когда нет возможности провести предварительную обработку текста, в котором проводится поиск.
  Таким образом, данный алгоритм является наиболее эффективным в обычных ситуациях, а его быстродействие повышается при увеличении подстроки или алфавита. В наихудшем случае трудоемкость рассматриваемого алгоритма $O(m+n)$.
- Существуют попытки совместить присущую алгоритму Кнута, Морриса и Пратта эффективность в "плохих" случаях и скорость алгоритма Бойера и Мура в "хороших" – например, турбо-алгоритм, обратный алгоритм Колусси и другие

Итоги

- Задачи поиска слова в тексте используются в криптографии, различных разделах физики, сжатии данных, распознавании речи и других сферах человеческой деятельности.
- Основная идея алгоритма прямым поиском заключается в посимвольном сравнении строки с подстрокой.
- Алгоритм прямого поиска является малозатратным и не нуждается в предварительной обработке и в дополнительном пространстве.
- Алгоритм Кнута, Морриса и Пратта основывается на том, что после частичного совпадения начальной части подстроки с соответствующими символами строки можно, вычислить сведения, с помощью которых быстро продвинуться по строке.
- Трудоемкость алгоритма Кнута, Морриса и Пратта лучше, чем трудоемкость алгоритма прямого поиска.
- Особенность алгоритма Бойера и Мура заключается в предварительных вычислениях над подстрокой с целью сравнения подстроки с исходной строкой, осуществляемой не во всех позициях.
- Алгоритм Бойера и Мура оптимален в большинстве случаев, когда нет возможности провести предварительную обработку текста, в котором проводится поиск.

Абстрактные типы данных

- Список
- Стек
- Очередь
- Ассоциативный массив
- Очередь с приоритетом

Список

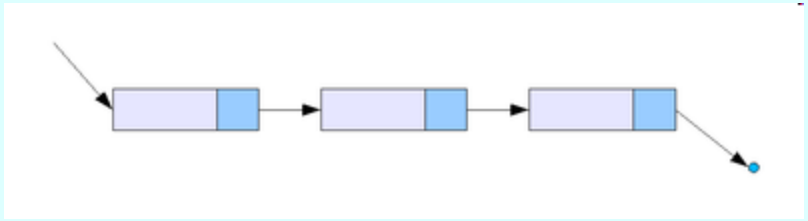
это абстрактный тип данных, представляющий собой упорядоченный набор значений, в котором некоторое значение может встречаться более одного раза.

Экземпляр списка является компьютерной реализацией математического понятия конечной последовательности — кортежа. Экземпляры значений, находящихся в списке, называются элементами списка (англ. item, entry либо element); если значение встречается несколько раз, каждое вхождение считается отдельным элементом.

Связный список

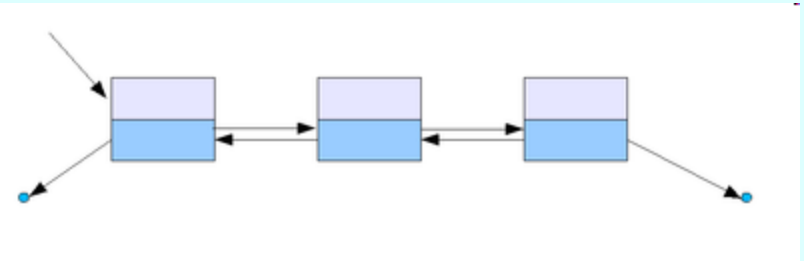
структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки («связки») на следующий и/или предыдущий узел списка

Односвязный список (Однонаправленный связный список)



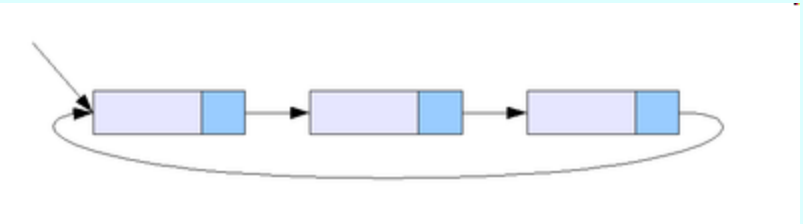
В односвязном списке можно передвигаться только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла невозможно.

Двусвязный список (Двунаправленный связный список)



По двусвязному списку можно передвигаться в любом направлении

Кольцевой связный список



В каждом кольцевом списке есть указатель на первый элемент. В этом списке константы NULL не существует.

Свойства связных списков

- **Размер списка** — количество элементов в нём, исключая последний «нулевой» элемент, являющийся по определению пустым списком.
- **Тип элементов** — тот самый тип , над которым строится список; все элементы в списке должны быть этого типа.
- **Отсортированность** — список может быть отсортирован в соответствии с некоторыми критериями сортировки (например, по возрастанию целочисленных значений, если список состоит из целых чисел).
- **Возможности доступа** — некоторые списки в зависимости от реализации могут обеспечивать программиста селекторами для доступа непосредственно к заданному по номеру элементу.
- **Сравниваемость** — списки можно сравнивать друг с другом на соответствие, причём в зависимости от реализации операция сравнения списков может использовать разные технологии.

Недостатки

- сложность определения адреса элемента по его индексу (номеру) в списке
- на поля-указатели (указатели на следующий и предыдущий элемент) расходуется дополнительная память (в массивах, например, указатели не нужны)
- работа со списком медленнее, чем с массивами, так как к любому элементу списка можно обратиться, только пройдя все предшествующие ему элементы
- элементы списка могут быть расположены в памяти разреженно, что окажет негативный эффект на кэширование процессора
- над связными списками гораздо труднее (хотя и в принципе возможно) производить параллельные векторные операции, такие как вычисление суммы

Достоинства

- лёгкость добавления и удаления элементов
- размер ограничен только объёмом памяти и разрядностью указателей
- динамическое добавление и удаление элементов

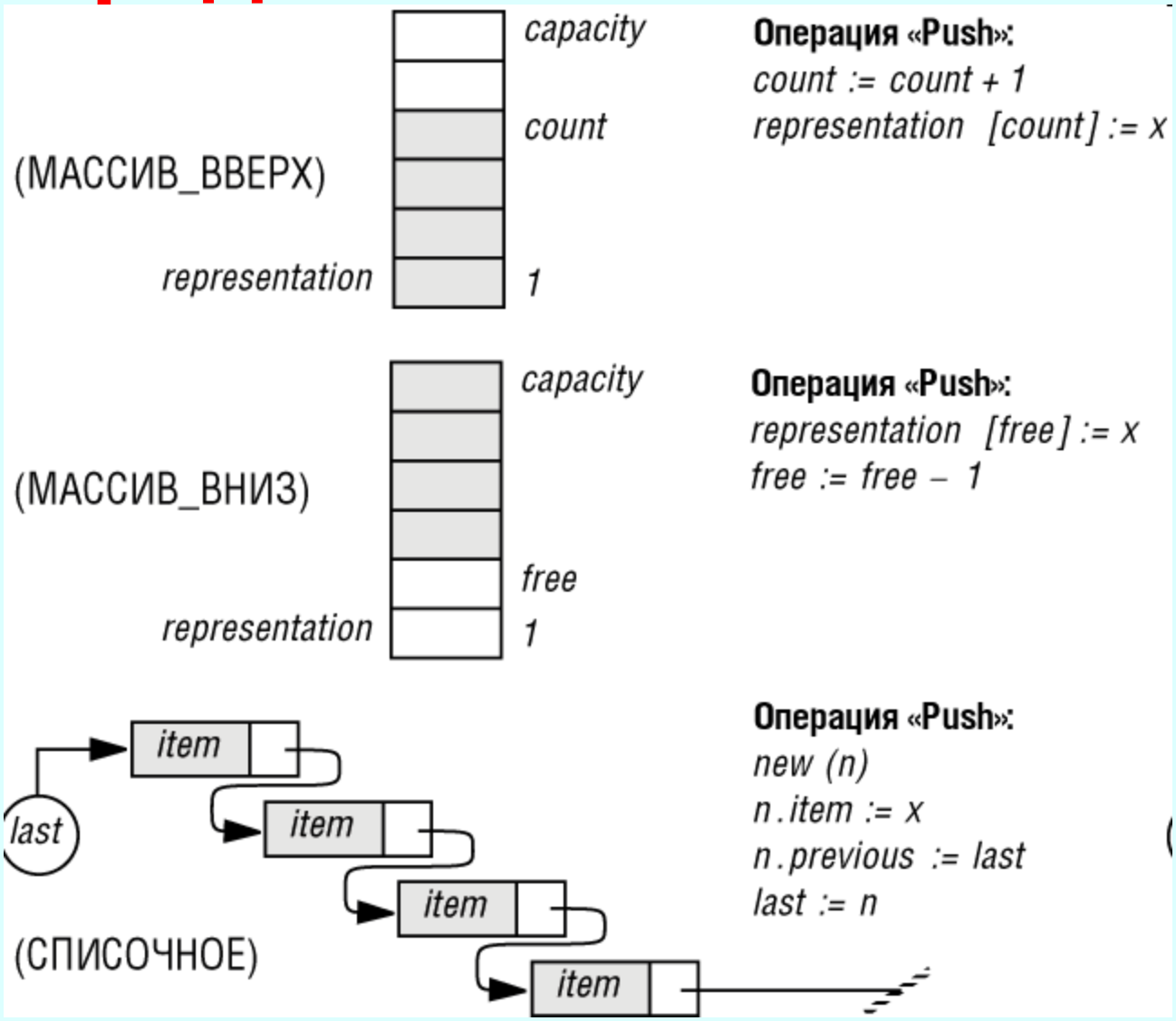
Стек

Стек (англ. stack — стопка)

— структура данных, в которой доступ к элементам организован по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

Добавление элемента, называемое также проталкиванием (*push*), возможно только в вершину стека (добавленный элемент становится первым сверху). Удаление элемента, называемое также выталкиванием (*pop*), тоже возможно только из вершины стека, при этом второй сверху элемент становится верхним

Представления стэков



- МАССИВ_ВВЕРХ: представляет стек посредством массива representation и целого числа count, с диапазоном значений от 0 (для пустого стека) до capacity - размера массива representation, элементы стека хранятся в массиве и индексируются от 1 до count.
- МАССИВ_ВНИЗ: похож на МАССИВ_ВВЕРХ, но элементы помещаются в конец стека, а не в начало. Здесь число, называемое free, является индексом верхней свободной позиции в стеке или 0, если все позиции в массиве заняты и изменяется в диапазоне от capacity для пустого стека до 0 для заполненного. Элементы стека хранятся в массиве и индексируются от capacity до free +1.
- СПИСОЧНОЕ: при списочном представлении каждый элемент стека хранится в ячейке с двумя полями: item, содержащем сам элемент, и previous, содержащем указатель на ячейку с предыдущим элементом. Для этого представления нужен также указатель last на ячейку, содержащую вершину стека.

Очередь

Очередь — структура данных с дисциплиной доступа к элементам «первый пришёл — первый вышел» (FIFO, First In — First Out). Добавление элемента (принято обозначать словом enqueue — поставить в очередь) возможно лишь в конец очереди, выборка — только из начала очереди (что принято называть словом dequeue — убрать из очереди), при этом выбранный элемент из очереди удаляется.

Способы реализации очереди

- **Массив** Первый способ представляет очередь в виде массива и двух целочисленных переменных start и end.
- **Связный список**
- **Реализация на двух стеках**

Ассоциативный массив

Ассоциативный массив (**словарь**) — абстрактный тип данных (интерфейс к хранилищу данных), позволяющий хранить пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу

Реализации ассоциативного массива

- Самая простая реализация может быть основана на обычном массиве, элементами которого являются пары (ключ, значение). Для ускорения операции поиска можно упорядочить элементы этого массива по ключу и осуществлять нахождение методом бинарного поиска. Но это увеличит время выполнения операции добавления новой пары, так как необходимо будет «раздвигать» элементы массива, чтобы в образовавшуюся пустую ячейку поместить новую запись

Очередь с приоритетом

(**priority queue**) — абстрактный тип данных в программировании поддерживающий три операции:

InsertWithPriority: добавить в очередь элемент с назначенным приоритетом

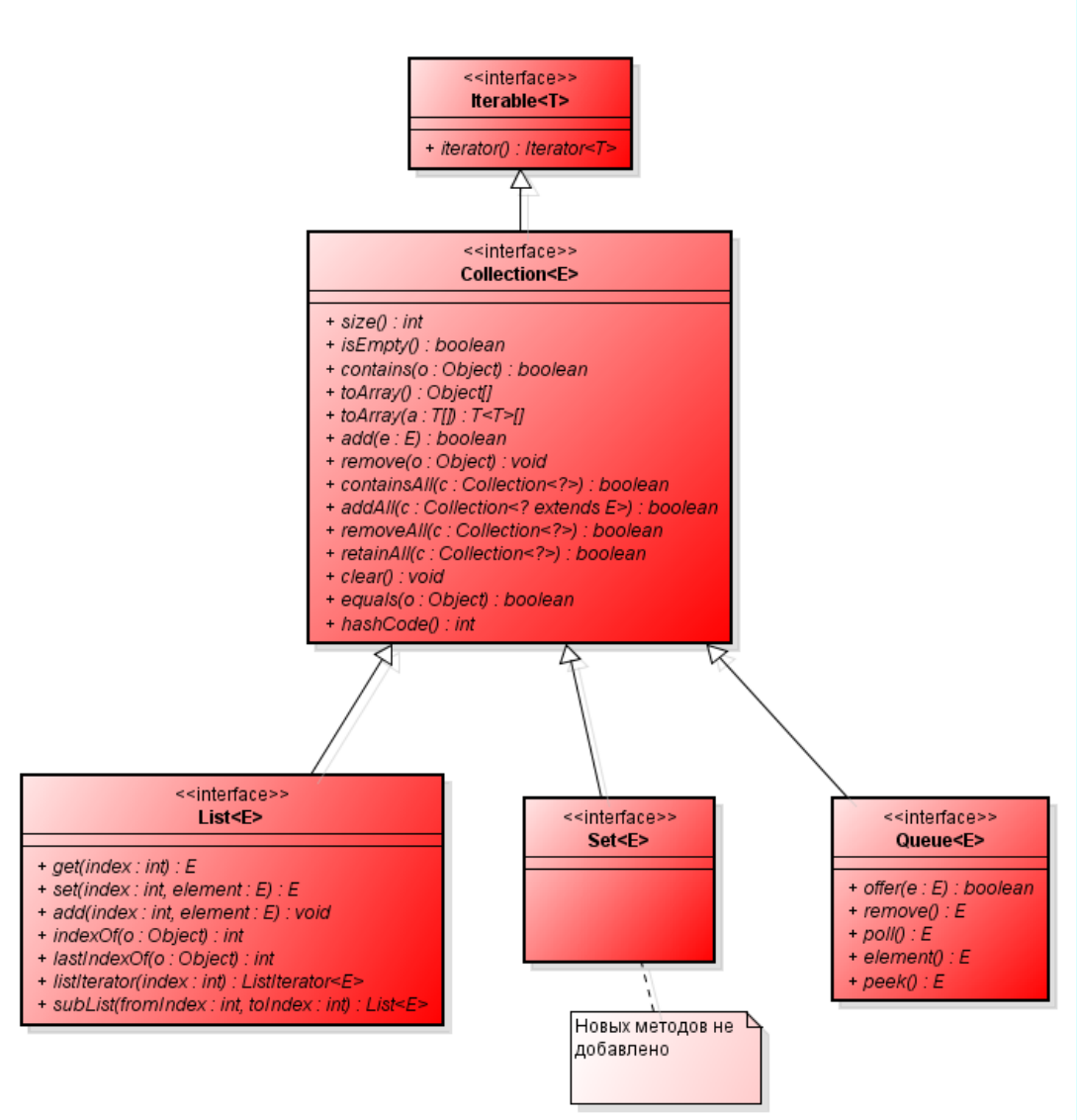
GetNext: извлечь из очереди и вернуть элемент с минимальным приоритетом (другие названия "PopElement(Off)" или "GetMinimum")

PeekAtNext (необязательная операция): просмотреть элемент с наивысшим приоритетом без извлечения

Коллекции объектов Java

- реализованы различными классами пакета java.util
- Коллекции обладают одним важным свойством — их размер не ограничен

Интерфейс Collection



- **Итератор** - объект, абстрагирующийся за единым интерфейсом доступ к элементам коллекции. Итератор это паттерн позволяющий получить доступ к элементам любой коллекции без вникания в суть ее реализации.
- **List** - Представляет собой неупорядоченную коллекцию, в которой допустимы дублирующие значения. Иногда их называют последовательностями (sequence). Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу.
- **Set** - описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Это соответствует математическому понятию множества (set).
- **Queue** - очередь. Сразу запоминаем как правильно произносится: Queue - Это коллекция, предназначенная для хранения элементов в порядке, нужном для их обработки. В дополнение к базовым операциям интерфейса Collection, очередь предоставляет дополнительные операции вставки, получения и контроля.

Реализации интерфейса List

- **ArrayList** - пожалуй самая часто используемая коллекция. ArrayList инкапсулирует в себе обычный массив, длина которого автоматически увеличивается при добавлении новых элементов.
  Так как ArrayList использует массив, то время доступа к элементу по индексу минимально (В отличии от LinkedList). При удалении произвольного элемента из списка, все элементы находящиеся «правее» смещаются на одну ячейку влево, при этом реальный размер массива (его емкость, capacity) не изменяется. Если при добавлении элемента, оказывается, что массив полностью заполнен, будет создан новый массив размером (n * 3) / 2 + 1, в него будут помещены все элементы из старого массива + новый, добавляемый элемент.
- **LinkedList** - Двусвязный список. Это структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и две ссылки («связки») на следующий и предыдущий узел списка. Доступ к произвольному элементу осуществляется за линейное время (но доступ к первому и последнему элементу списка всегда осуществляется за константное время — ссылки постоянно хранятся на первый и последний, так что добавление элемента в конец списка вовсе не значит, что придется перебирать весь список в поисках последнего элемента). В целом же, LinkedList в абсолютных величинах проигрывает ArrayList и по потребляемой памяти и по скорости выполнения операций.

Реализации интерфейса Set

- **HashSet** - коллекция, не позволяющая хранить одинаковые объекты(как и любой Set). **HashSet** инкапсулирует в себе объект **HashMap** (то-есть использует для хранения хэш-таблицу).
  Хеш-таблица хранит информацию, используя так называемый механизм хеширования, в котором содержимое ключа используется для определения уникального значения, называемого хеш-кодом. Этот хеш-код затем применяется в качестве индекса, с которым ассоциируются данные, доступные по этому ключу. Преобразование ключа в хеш-код выполняется автоматически — вы никогда не увидите самого хеш-кода. Также ваш код не может напрямую индексировать хеш-таблицу. Выгода от хеширования состоит в том, что оно обеспечивает константное время выполнения методов add(), contains(), **remove()** и **size()** , даже для больших наборов.

- **LinkedHashSet** - поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор. То есть, когда идет перебор объекта класса LinkedHashSet с применением итератора, элементы извлекаются в том порядке, в каком они были добавлены.
- **TreeSet** - коллекция, которая хранит свои элементы в виде упорядоченного по значениям дерева. TreeSet инкапсулирует в себе TreeMap, который в свою очередь использует сбалансированное бинарное красно-черное дерево для хранения элементов. TreeSet хорош тем, что для операций add, remove и contains потребуется гарантированное время log(n).

Реализации интерфейса Queue

- **PriorityQueue** - единственная прямая реализация интерфейса **Queue** (не считая **LinkedList**, который больше является списком, чем очередью).
  Эта очередь упорядочивает элементы либо по их натуральному порядку (используя интерфейс **Comparable**), либо с помощью интерфейса **Comparator**, полученному в конструкторе.

Реализации интерфейса Map

- Интерфейс **Map** соотносит уникальные ключи со значениями. Ключ — это объект, который используется для последующего извлечения данных. Задавая ключ и значение, можно помещать значения в объект карты. После того как это значение сохранено, можно получить его по ключу.

- HashMap — основан на хэш-таблицах, реализует интерфейс Map (что подразумевает хранение данных в виде пар ключ/значение). Ключи и значения могут быть любых типов, в том числе и null. Данная реализация не дает гарантий относительно порядка элементов с течением времени.
- LinkedHashMap - расширяет класс HashMap. Он создает связный список элементов в карте, расположенных в том порядке, в котором они вставлялись. Это позволяет организовать перебор карты в порядке вставки. То есть, когда происходит итерация по коллекционному представлению объекта класса LinkedHashMap, элементы будут возвращаться в том порядке, в котором они вставлялись. Вы также можете создать объект класса LinkedHashMap, возвращающий свои элементы в том порядке, в котором к ним в последний раз осуществлялся доступ.
- TreeMap - расширяет класс AbstractMap и реализует интерфейс NavigatebleMap. Он создает коллекцию, которая для хранения элементов применяет дерево. Объекты сохраняются в отсортированном порядке по возрастанию. Время доступа и извлечения элементов достаточно мало, что делает класс TreeMap блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена.
- WeakHashMap - коллекция, использующая слабые ссылки для ключей (а не значений). Слабая ссылка (англ. weak reference) — специфический вид ссылок на динамически создаваемые объекты в системах со сборкой мусора. Отличается от обычных ссылок тем, что не учитывается сборщиком мусора при выявлении объектов, подлежащих удалению. Ссылки, не являющиеся слабыми, также иногда именуют «сильными».

```
Iterator<ItemInstance> allItemsIt = InventoryManager.allItemsIterator(this);
while (allItemsIt.hasNext()) {
    ItemInstance itemInstance = allItemsIt.next();
    if (itemInstance.getItem() instanceof Weapon) {
        if (!itemInstance.initModuleSystem()) {
            continue;
        }
```

```
Collection<ItemInstance> inventoryItems = inventory.getItems();
for (ItemInstance itemInstance : inventoryItems) {
    TIItem template = itemInstance.getItem();
    if (template instanceof Module) {
        continue;
    }
```

```
private void markEquippedItemsToRestoreFree(PremiumAccountRecord premiumAccountRecord) {
    Map<Integer, Integer> acceptedItems = new HashMap<>();
    for (ItemInstance ii :pet.getInventory().getItems()) {
        if (premiumAccountRecord.isItemAllowedToRestoreFree(ii)) {
            if (acceptedItems.containsKey(ii.getItemId())) {
                acceptedItems.put(ii.getItemId(), ii.getCount() + acceptedItems.get(ii.getItemId()));
            } else {
                acceptedItems.put(ii.getItemId(), ii.getCount());
            }
        }
    }
```

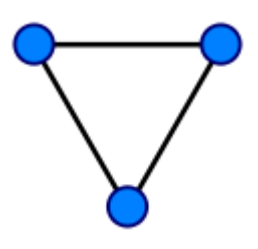
Раздел: Алгоритмы на графах

Тема лекции:

Поиск по графу

Понятие графа. Способы представления графа.

- Граф *G = (V, E)* задается парой конечных множеств *V* и *E.* Элементы первого множества *V1, v2,..., vM* называются **вершинами** графа (при графическом представлении им соответствуют точки). Элементы второго множества *e1, e2, ...,eN* называют **ребрами.**



- Вершины и рёбра графа называются также **элементами** графа, число вершин в графе
- $|V|$ — **порядком**, число рёбер
- $|E|$ — **размером** графа.

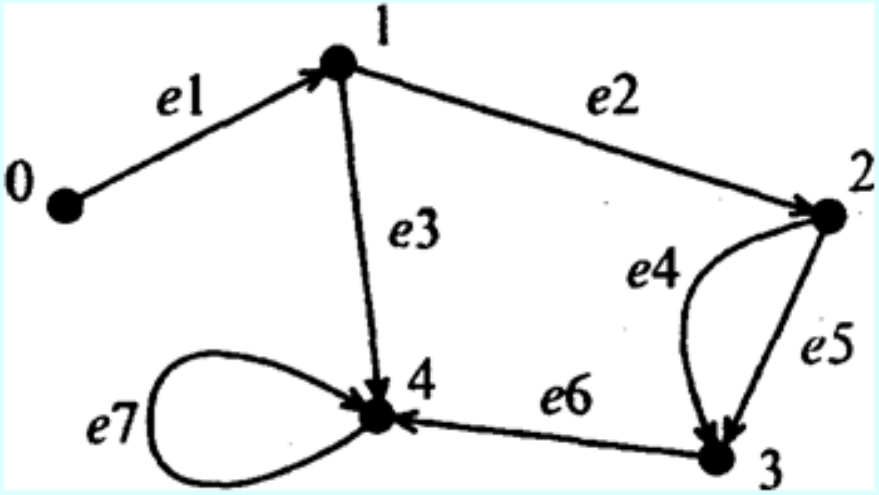
- Вершины U и V называются **концевыми** вершинами (или просто **концами**) ребра $e = \{u, v\}$.
- Ребро, в свою очередь, **соединяет** эти вершины. Две концевые вершины одного и того же ребра называются **соседними**.
- Два ребра называются **смежными**, если они имеют общую концевую вершину.

- Два ребра называются **кратными**, если множества их концевых вершин совпадают.

- Ребро называется **петлёй**, если его концы совпадают, то есть .

$$e = \{v, v\}$$

- Если ребра графа определяются упорядоченными парами вершин, то такой граф называют **ориентированным**



Представления графов: **Матрица смежности**

- Если вершины графа G помечены метками *v1, v2,..., vn*, то элементы матрицы смежности *A(G)* размера *V, xV* определяются следующим образом: *A(i.j)* = 1, если *vi* смежна с *vj; A(ij)* = 0 в противном случае

A(G):

	1	2	3	4	5
1	0	1	0	1	1
2	1	0	1	0	1
3	0	1	0	1	0
4	1	0	1	0	1
5	1	1	0	1	0

Представления графов: **Матрица инцидентности**

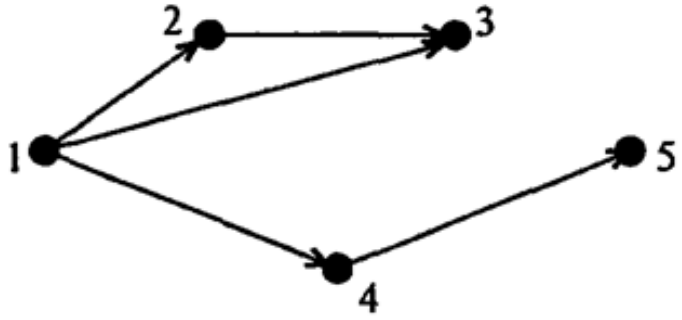
- Если вершины графа *G* помечены метками *v1, v2,..., vm,* а ребра - метками *e1, e2,..., eп,* то элементы **матрицы инцидентности** *I(G)* размера *M* x *N* определяются правилом: *B(ij)* = 1, если *vi* инцидентна *ej; B(iJ)* = 0 в противном случае

I(G):

	a	b	c	d	e	f	g
1	1	1	0	0	0	1	0
2	0	1	1	0	0	0	1
3	0	0	1	1	0	0	0
4	0	0	0	1	1	1	0
5	1	0	0	0	1	0	1

Представления графов: **Матрица достижимости**

- Для ориентированного графа *G*, имеющего *N* вершин можно рассмотреть матрицу **достижимости** *C(G)* размера *N* x *N*, элементы которой определяются так: C(*I*, *J)* = 1, если вершина *vj* достижима из *vi; C(I, J)* = 0 в противном случае



C(G):

	1	2	3	4	5
1	0	1	1	1	1
2	0	0	1	0	0
3	0	0	0	0	0
4	0	0	0	0	1
5	0	0	0	0	0

Поиск в графе

- **Маршрут, или путь** - это последовательность ребер в неориентированном графе, в котором конец каждого ребра совпадает с началом следующего ребра. Число ребер маршрута называется его длиной

Поиск кратчайшего пути – важная задача

- возникающая в разных областях науки и техники:
- –в экономике ,
- •при оптимизации перевозок
- –в робототехнике,
- •при поиске роботом оптимального маршрута
- –в компьютерных играх
- •при перемещении отрядов в лабиринте в стратегиях реального времени

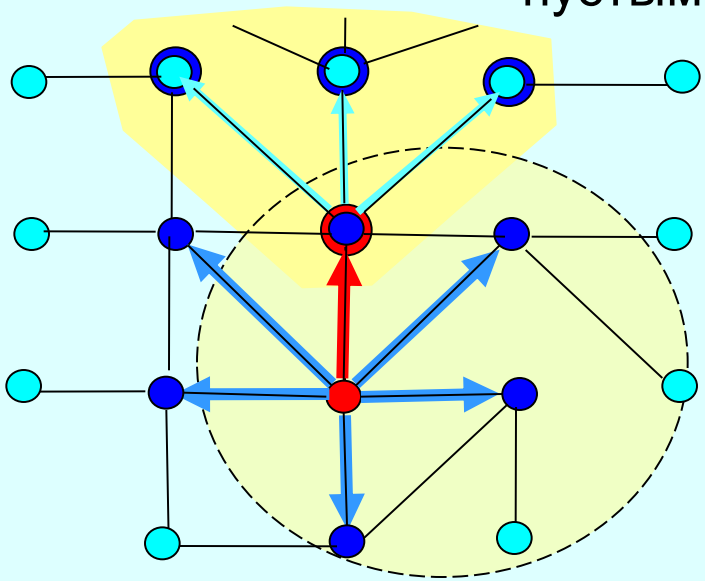
четыре задачи поиска кратчайших путей на графе

- Выделяют задачи:
- –между двумя вершинами (задача 1);
- –от заданной вершины ко всем вершинам (задача 2);
- –от всех вершин до заданной вершины (задача 3);
- –от всех вершин ко всем вершинам (задача 4).

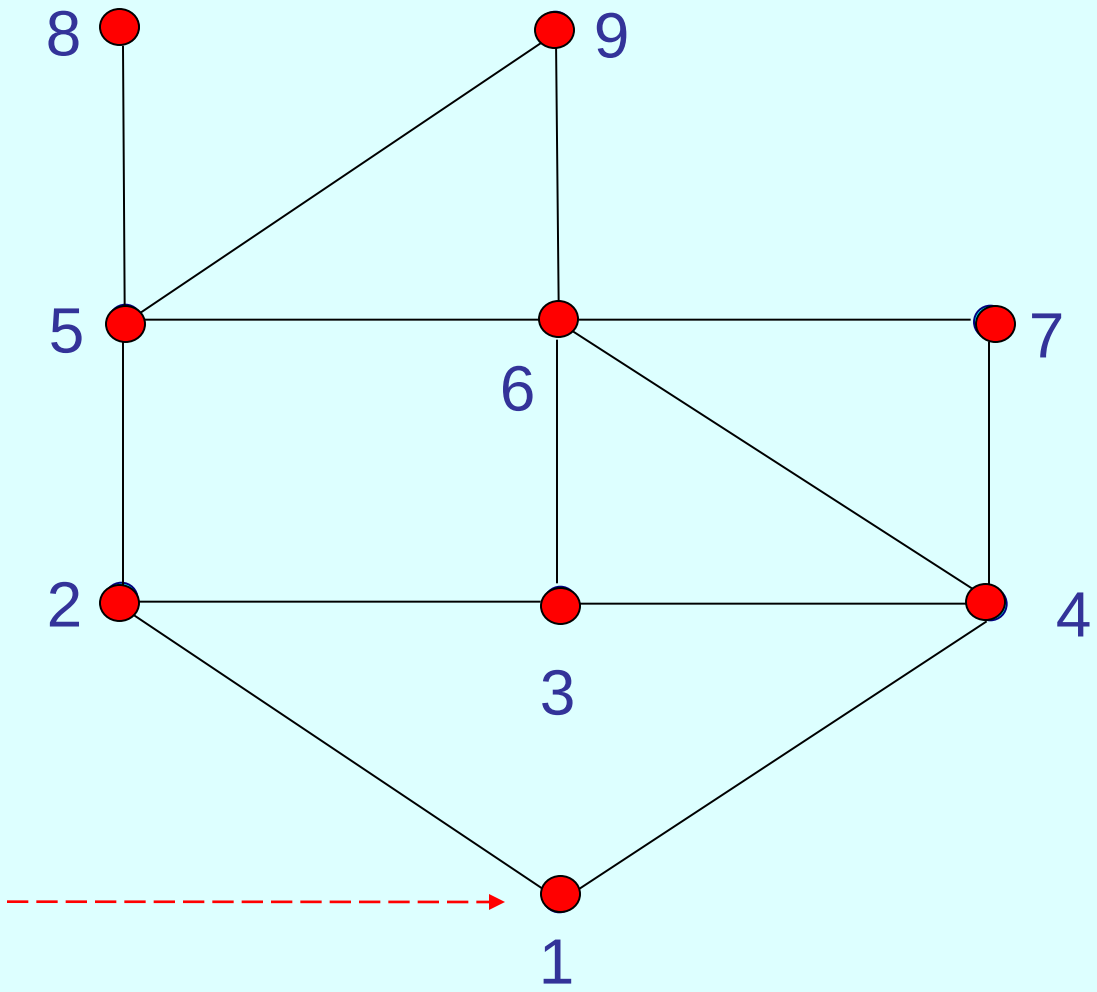
Поиск по графу. Алгоритм пометок

Обход графа:

- из *текущей вершины* доступны смежные; они *помечаются* и заносятся в специальное множество;
- из множества извлекается *очередная* вершина, она *посещается* (обрабатывается), и процесс повторяется, пока множество не станет пустым.



Поиск по графу. Алгоритм пометок



Структура смежности:

★	1	2, 4
★	2	1, 3, 5
★	3	2, 4, 6
★	4	1, 3, 6, 7
★	5	2, 6, 8, 9
★	6	3, 4, 5, 7, 9
★	7	4, 6
★	8	5
★	9	5, 6

Множество: 1 2 4 3 5 6 8 9 7

Обход закончен!

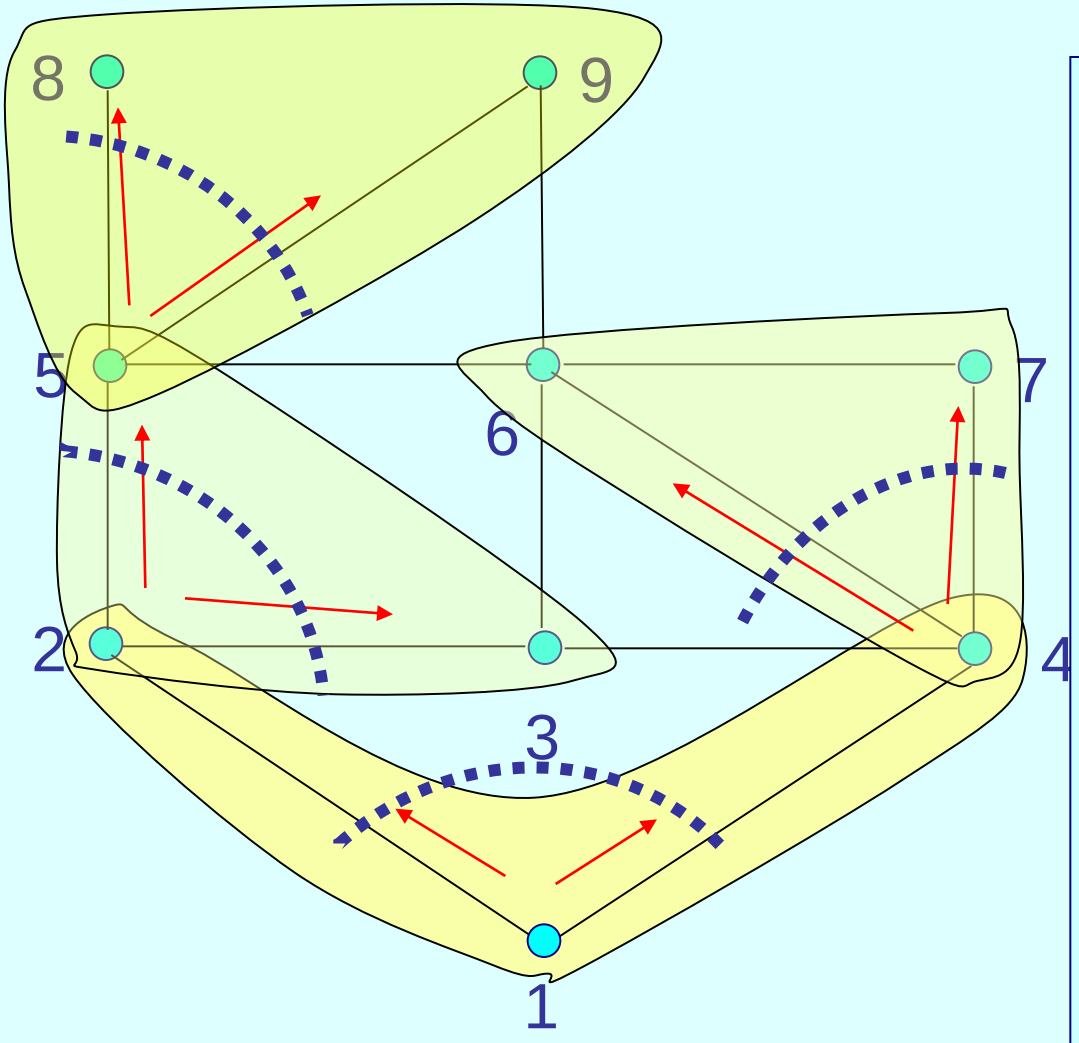
Поиск по графу: волновой алгоритм

- Дано: непустой граф G=(V,E). Требуется найти путь между вершинами s и t графа (s не совпадает с t), содержащий минимальное количество промежуточных вершин (ребер).

Поиск по графу: волновой алгоритм

1. каждой вершине vi приписывается целое число T(vi) - волновая метка (начальное значение T(vi)=-1);
2. заводятся два списка OldFront и NewFront (старый и новый "фронт волны"), а также переменная T (текущее время);
3. OldFront:={s}; NewFront:={}; T(s):=0; T:=0;
4. для каждой из вершин, входящих в OldFront, просматриваются инцидентные (смежные) ей вершины uj, и если T(uj) = -1, то T(uj):=T+1, NewFront:=NewFront + {uj};
5. если NewFront = {}, то ВЫХОД("нет решения");
6. если t O NewFront (т.е. одна из вершин uj совпадает t), то найден кратчайший путь между s и t с T(t)=T+1 промежуточными ребрами; ВЫХОД("решение найдено");
7. OldFront:=NewFront; NewFront:={}; T:=T+1; goto (4).

Волновой алгоритм



Свойство BFS-остова:

⌉ $G = (V, E)$ и

(V, T) – BFS-остов графа G.

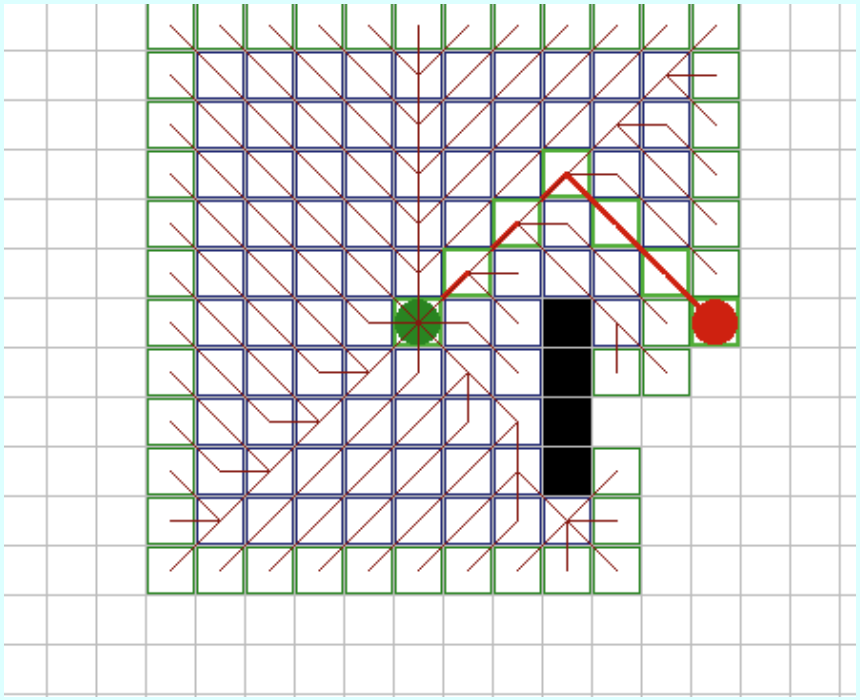
Тогда путь в (V, T) из $v \in V$ до корня остова r является кратчайшим путем из v в r в графе G.

Док-во по индукции (по этапам волны или по порциям записи в очередь).

ПОИСК В ШИРИНУ

(Breadth First Search - BFS)

Начиная со стартового узла, этот алгоритм сначала определяет все непосредственно соседние узлы, затем все узлы в двух шагах, затем в трех, и так далее, пока цель не достигнута.



ПОИСК В ШИРИНУ

Типичным является то, что для каждого узла его непроверенные соседи помещаются в список Open, который обычно является FIFO очередью.

```
ПоискВширь
   узел n, n', s
      s.родитель = null    // s - стартовый узел
      положить s в Open
      пока очередь Open не пуста
         извлечь n из Open
         если n целевой узел
            сконструировать путь
            выйти с кодом "успешное завершение"
         для каждого наследника n' узла n
            если n' уже посещен, то
               пропустить n'
            n'.родитель = n
            положить n' в Open
      выйти с кодом "путь не найден"
```

из заданной стартовой вершины a.

$V(x)$ обозначает множество всех вершин, смежных с вершиной x

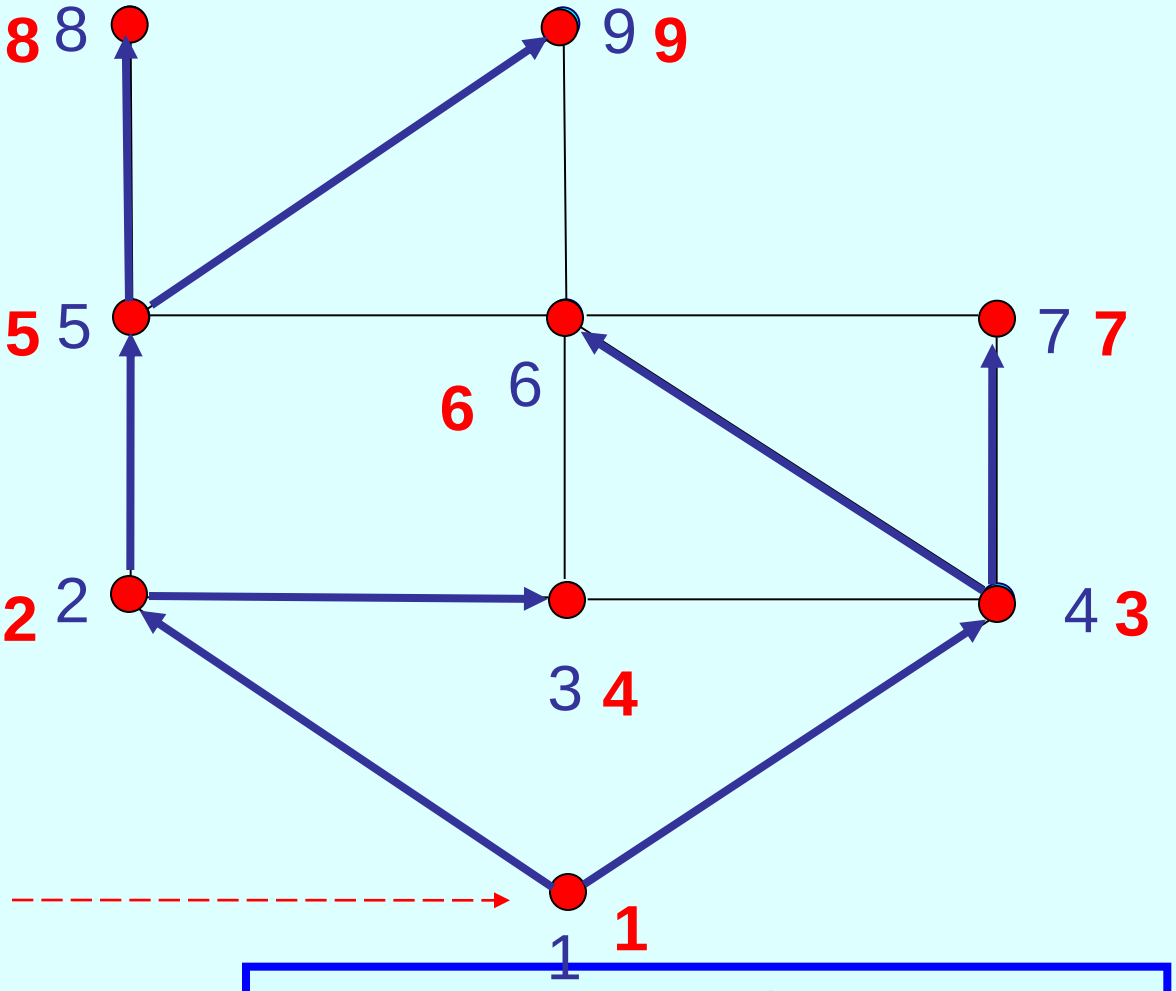
Q - очередь открытых вершин.

Procedure BFS(a)

1. посетить вершину a
2. $a \Rightarrow Q$
3. **while** $Q \neq \varnothing$ **do**
4. $x \Leftarrow Q$
5. **for** $y \in V(x)$ **do**
6. исследовать ребро (x, y)
7. **if** вершина y новая
8. **then** посетить вершину y
9. $y \Rightarrow Q$

ПОИСК В ШИРИНУ

Пример



★	1	2, 4
★	2	1, 3, 5
★	3	2, 4, 6
★	4	1, 3, 6, 7
★	5	2, 6, 8, 9
★	6	3, 4, 5, 7, 9
★	7	4, 6
★	8	5
★	9	5, 6

Очередь: 1 2 4 3 5 6 7 8 9

Обход закончен!

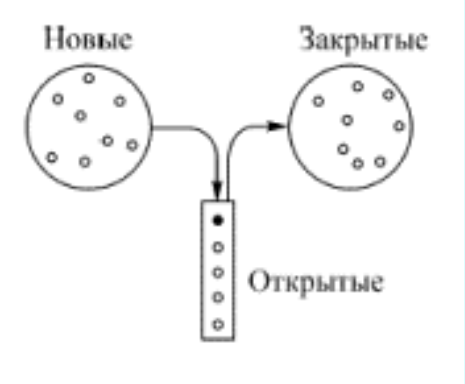
ПОИСК В ГЛУБИНУ

(Depth First Search - DFS)

Идея этого метода - идти вперед в неисследованную область, пока это возможно, если же вокруг все исследовано, отступить на шаг назад и искать новые возможности для продвижения вперед. Метод поиска в глубину известен под разными названиями, например, "бэктрекинг", "поиск с возвращением"

отличие от поиска в ширину

- при поиске в глубину в качестве активной выбирается та из открытых вершин, которая была посещена последней. Для реализации такого правила выбора наиболее удобной структурой хранения множества открытых вершин является стек: открываемые вершины складываются в стек в том порядке, в каком они открываются, а в качестве активной выбирается последняя вершина.



ПОИСК В ГЛУБИНУ
(Depth First Search - DFS)

Обход начинается с посещения заданной стартовой вершины a, которая становится активной и единственной открытой вершиной. Затем выбирается инцидентное вершине a ребро (a, y) и посещается вершина y. Она становится открытой и активной. Заметим, что при поиске в ширину вершина a оставалась активной до тех пор, пока не были исследованы все инцидентные ей ребра. В дальнейшем, как и при *поиске в ширину*, каждый очередной шаг начинается с выбора активной вершины из множества открытых вершин. Если все ребра, инцидентные активной вершине x, уже исследованы, она превращается в закрытую. В противном случае выбирается одно из неисследованных ребер (x, y), это ребро исследуется. Если вершина y новая, то она посещается и превращается в открытую.

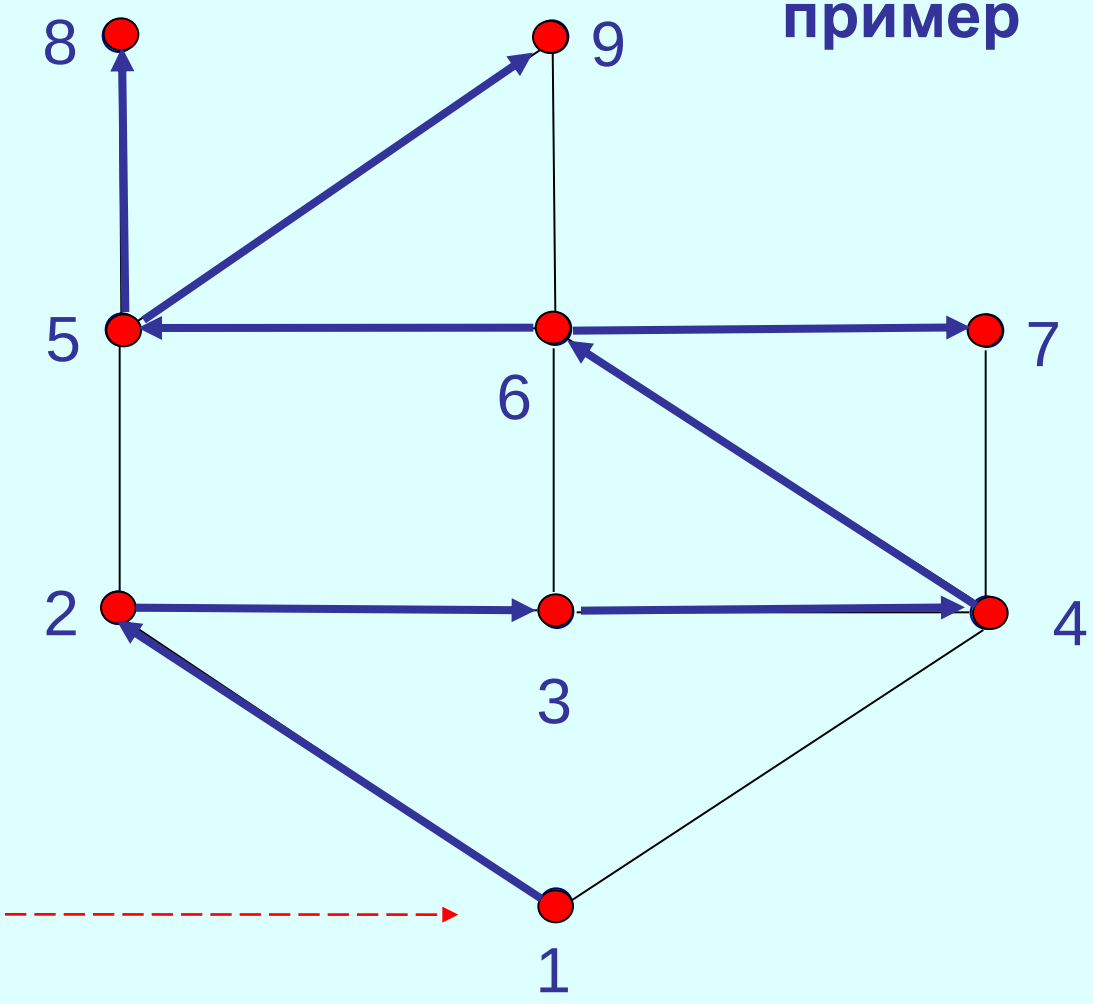
- Обозначим S стек для открытых вершин через , остальные обозначения сохраняют тот же смысл, что и в предыдущем разделе. Через top(S) обозначается верхний элемент стека (т.е. последний элемент, добавленный к стеку)

Procedure *DFS*(a)

1. посетить вершину a
2. $a \Rightarrow S$
3. **while** $S \neq \varnothing$ **do**
4. $x := \mathrm{top}(S)$
5. **if** имеется неисследованное ребро (x, y)
6. **then** исследовать ребро (x, y)
7. **if** вершина y новая
8. **then** посетить вершину y
9. $y \Rightarrow S$
10. **else** удалить x из S

ПОИСК В ГЛУБИНУ

пример



★	1	2, 4
★	2	1, 3, 5
★	3	2, 4, 6
★	4	1, 3, 6, 7
★	5	2, 6, 8, 9
★	6	3, 4, 5, 7, 9
★	7	4, 6
★	8	5
★	9	5, 6

Алгоритм Дейкстры

- Находит кратчайшее расстояние от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса. Алгоритм широко применяется в программировании и технологиях, например, его использует протокол OSPF для устранения кольцевых маршрутов.

Формулировка задачи
алгоритм Дейкстры

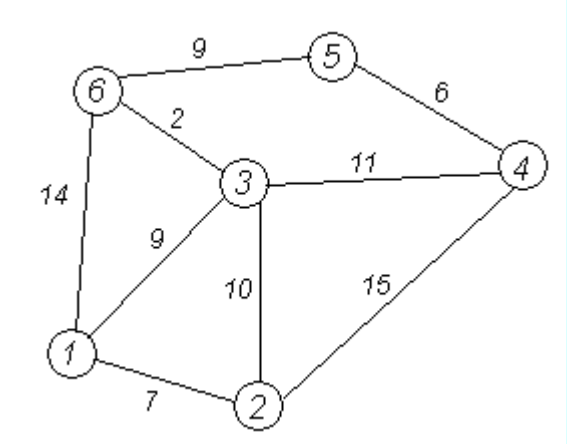
- *Вариант 1.* Дана сеть автомобильных дорог, соединяющих города Московской области. Некоторые дороги односторонние. Найти кратчайшие пути от города Москва до каждого города области (если двигаться можно только по дорогам).
- *Вариант 2.* Имеется некоторое количество авиарейсов между городами мира, для каждого известна стоимость. Стоимость перелёта из А в В может быть не равна стоимости перелёта из В в А. Найти маршрут минимальной стоимости (возможно, с пересадками) от Копенгагена до Барнаула.

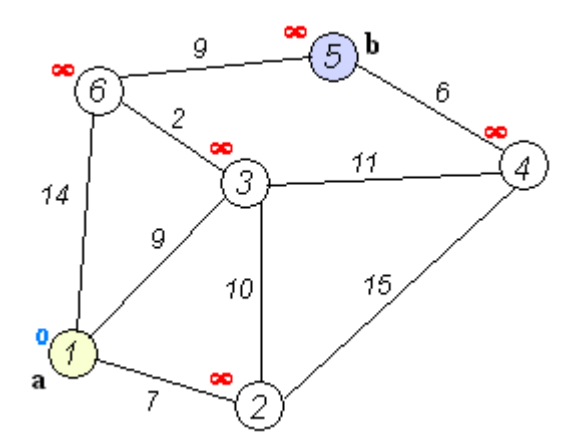
Неформальное объяснение

- Каждой вершине из *V* сопоставим метку — минимальное известное расстояние от этой вершины до *a*. Алгоритм работает пошагово — на каждом шаге он «посещает» одну вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены.
- **Инициализация**. Метка самой вершины *a* полагается равной 0, метки остальных вершин — бесконечности. Это отражает то, что расстояния от *a* до других вершин пока неизвестны. Все вершины графа помечаются как непосещённые.
- **Шаг алгоритма**. Если все вершины посещены, алгоритм завершается. В противном случае, из ещё не посещённых вершин выбирается вершина *u*, имеющая минимальную метку. Мы рассматриваем всевозможные маршруты, в которых *u* является предпоследним пунктом. Вершины, в которые ведут рёбра из *u*, назовем *соседями* этой вершины. Для каждого соседа вершины *u*, кроме отмеченных как посещённые, рассмотрим новую длину пути, равную сумме значений текущей метки *u* и длины ребра, соединяющего *u* с этим соседом. Если полученное значение длины меньше значения метки соседа, заменим значение метки полученным значением длины. Рассмотрев всех соседей, пометим вершину *u* как посещенную и повторим шаг алгоритма.

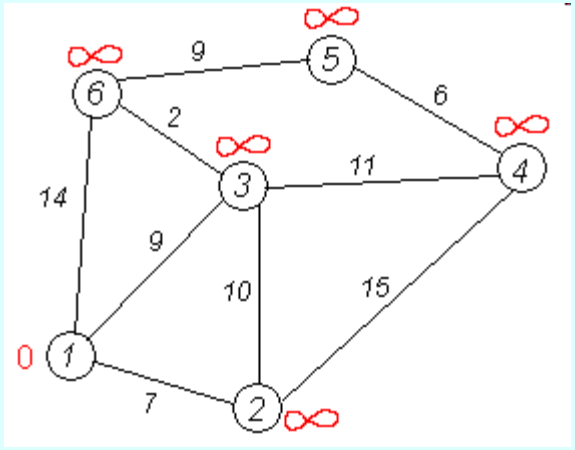
Пример

- Рассмотрим выполнение алгоритма на примере графа, показанного на рисунке. Пусть требуется найти кратчайшие расстояния от 1-й вершины до всех остальных

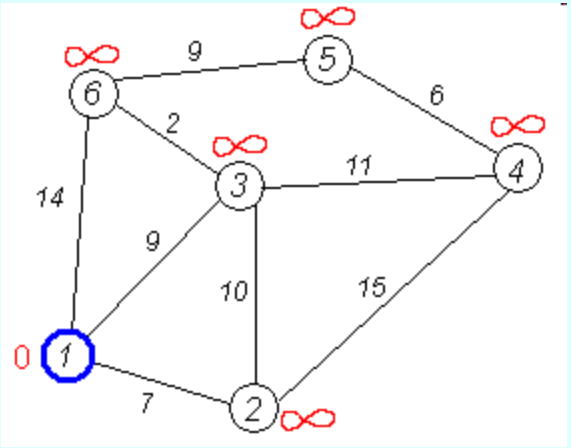




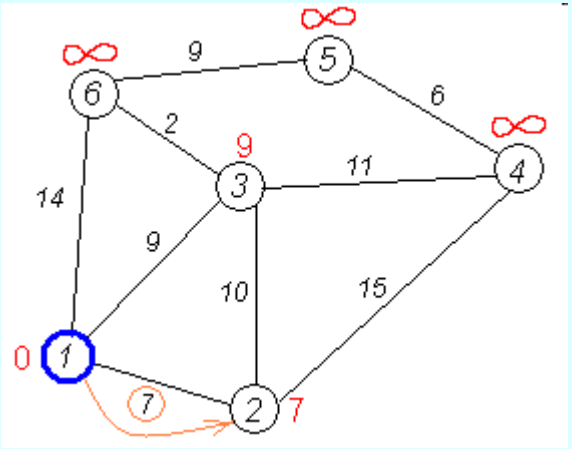
- Кружками обозначены вершины, линиями — пути между ними (ребра графа). В кружках обозначены номера вершин, над ребрами обозначена их «цена» — длина пути. Рядом с каждой вершиной красным обозначена метка — длина кратчайшего пути в эту вершину из вершины 1



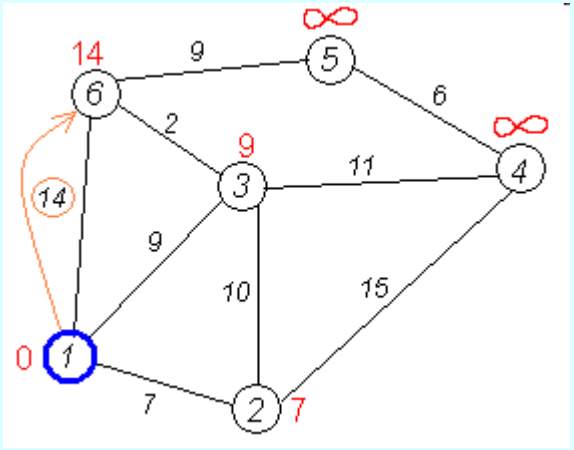
- **Первый шаг**. Рассмотрим шаг алгоритма Дейкстры для нашего примера. Минимальную метку имеет вершина 1. Её соседями являются вершины 2, 3 и 6.



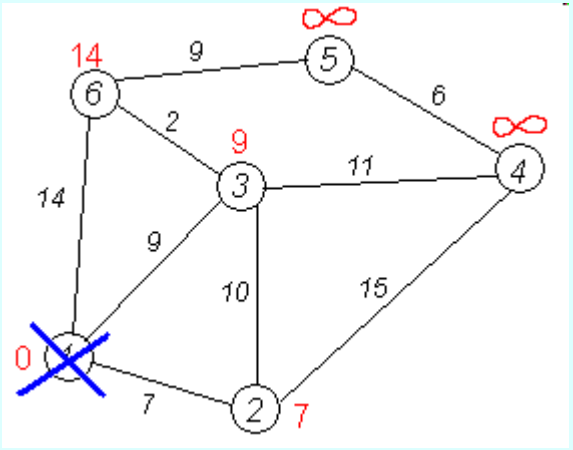
- Первый по очереди сосед вершины 1 — вершина 2, потому что длина пути до неё минимальна. Длина пути в неё через вершину 1 равна сумме кратчайшего расстояния до вершины 1, значению её метки, и длины ребра, идущего из 1-й в 2-ю, то есть 0 + 7 = 7. Это меньше текущей метки вершины 2, бесконечности, поэтому новая метка 2-й вершины равна 7.



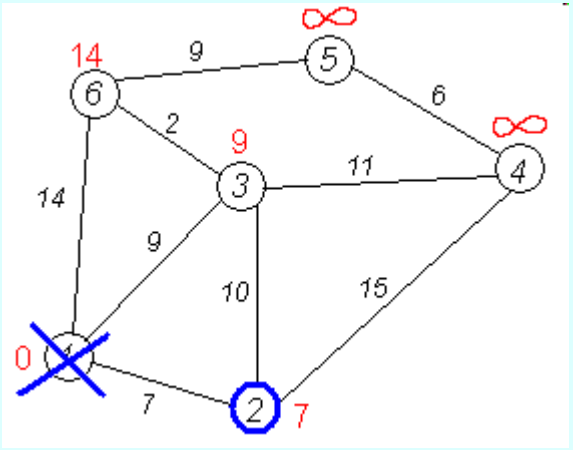
- Аналогичную операцию проделываем с двумя другими соседями 1-й вершины — 3-й и 6-й.



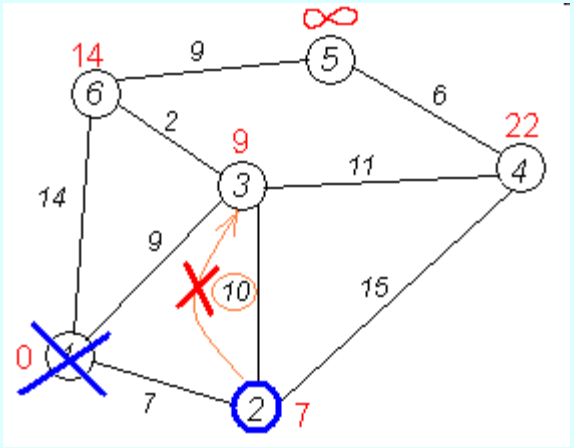
- Все соседи вершины 1 проверены. Текущее минимальное расстояние до вершины 1 считается окончательным и пересмотру не подлежит (то, что это действительно так, впервые доказал Э. Дейкстра). Вычеркнем её из графа, чтобы отметить, что эта вершина посещена.



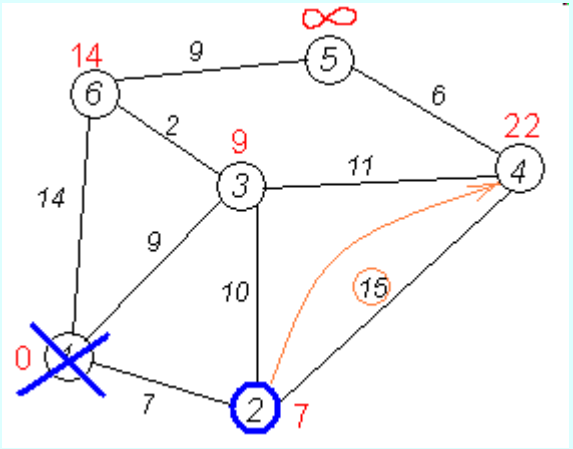
- **Второй шаг**. Шаг алгоритма повторяется. Снова находим «ближайшую» из непосещенных вершин. Это вершина 2 с меткой 7.



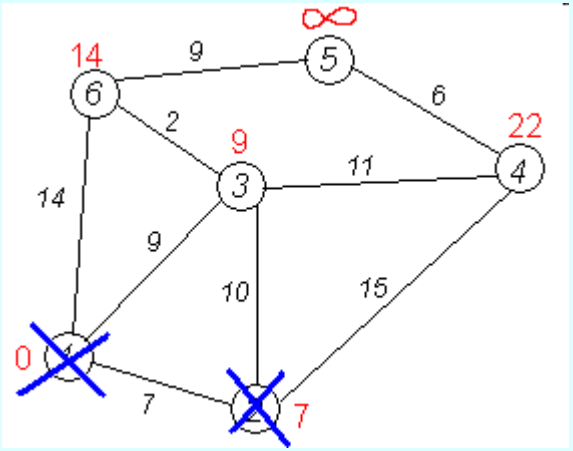
- Снова пытаемся уменьшить метки соседей выбранной вершины, пытаясь пройти в них через 2-ю вершину. Соседями вершины 2 являются вершины 1, 3 и 4.
- Первый (по порядку) сосед вершины 2 — вершина 1. Но она уже посещена, поэтому с 1-й вершиной ничего не делаем.
- Следующий сосед вершины 2 — вершина 3, так как имеет минимальную метку из вершин, отмеченных как не посещённые. Если идти в неё через 2, то длина такого пути будет равна 17 (7 + 10 = 17). Но текущая метка третьей вершины равна 9<17, поэтому метка не меняется.



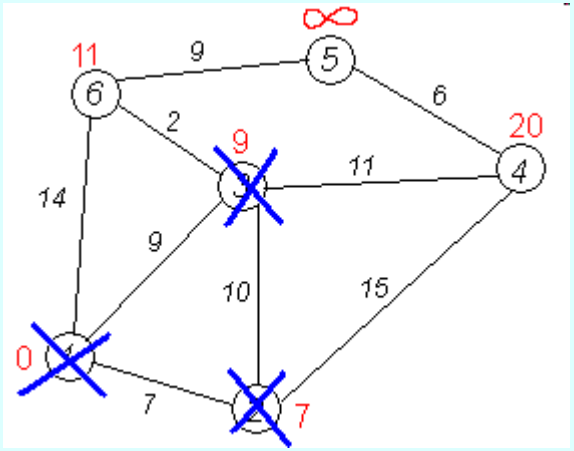
- Ещё один сосед вершины 2 — вершина 4. Если идти в неё через 2-ю, то длина такого пути будет равна сумме кратчайшего расстояния до 2-й вершины и расстояния между вершинами 2 и 4, то есть 22 (7 + 15 = 22). Поскольку 22< , устанавливаем метку вершины 4 равной 22



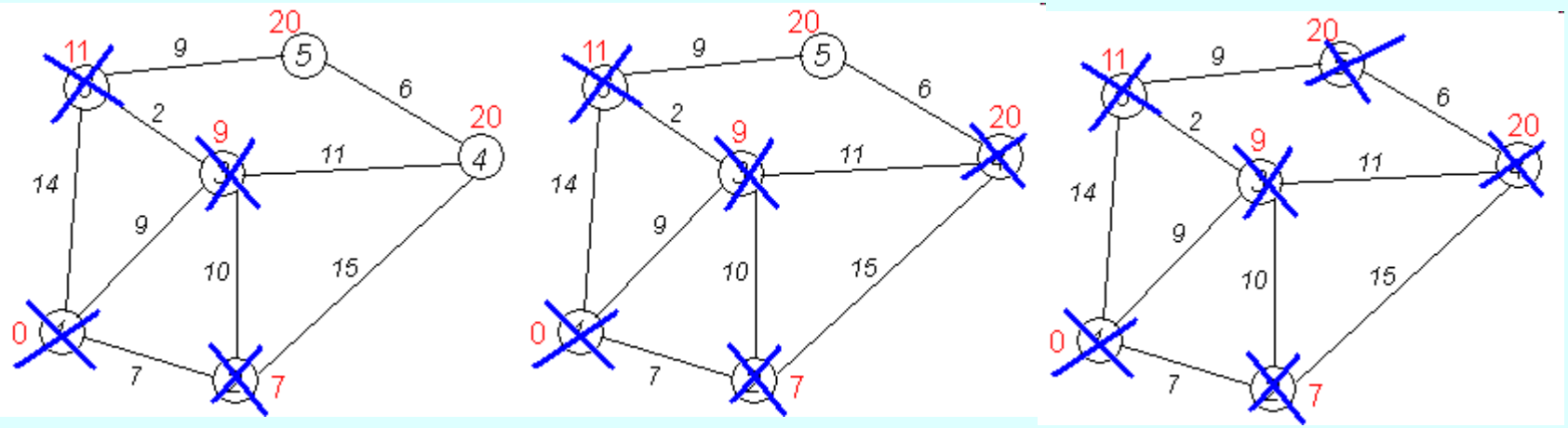
- Все соседи вершины 2 просмотрены, замораживаем расстояние до неё и помечаем её как посещенную.



- **Третий шаг**. Повторяем шаг алгоритма, выбрав вершину 3. После её «обработки» получим такие результаты:



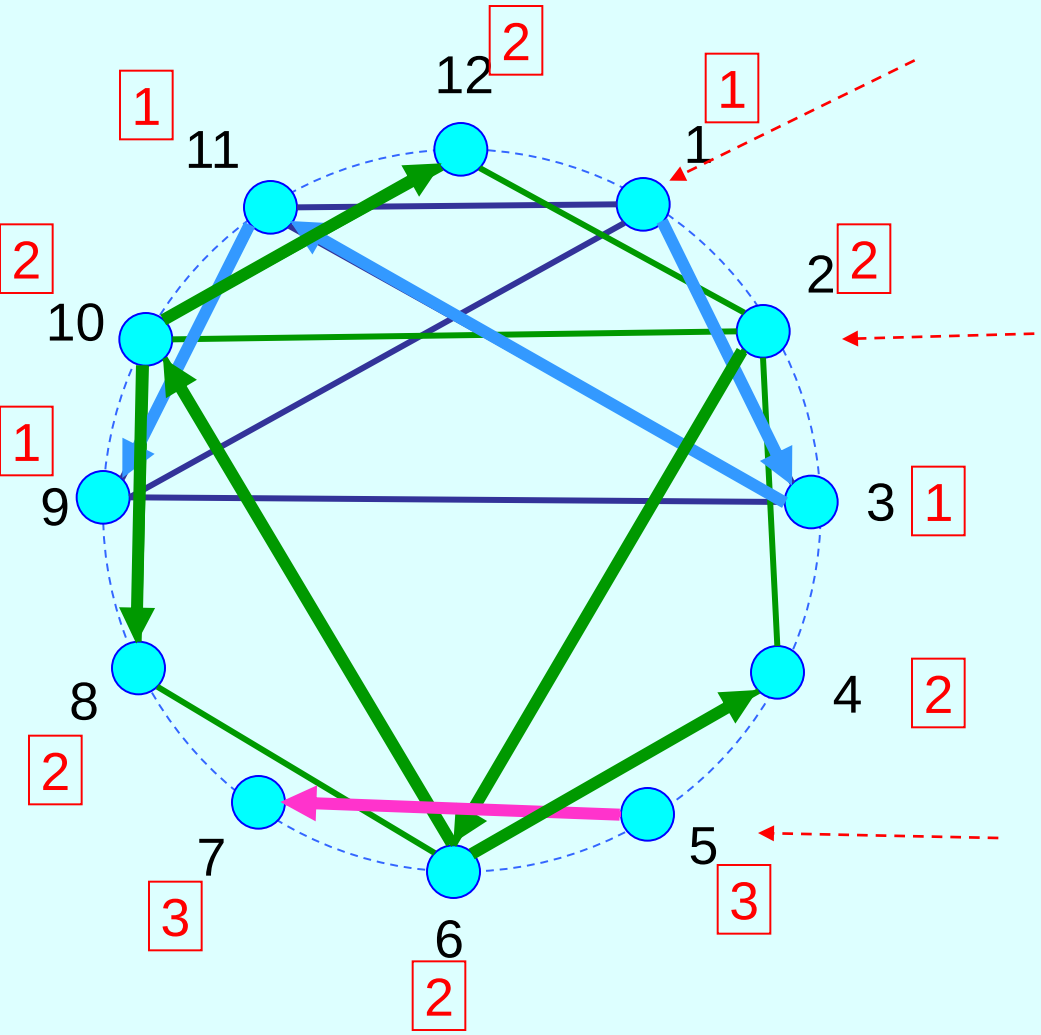
- **Дальнейшие шаги**. Повторяем шаг алгоритма для оставшихся вершин. Это будут вершины 6, 4 и 5, соответственно порядку.



СВЯЗНЫЕ КОМПОНЕНТЫ (Поиск в глубину)

Рекурсивная процедура нахождения связных компонент

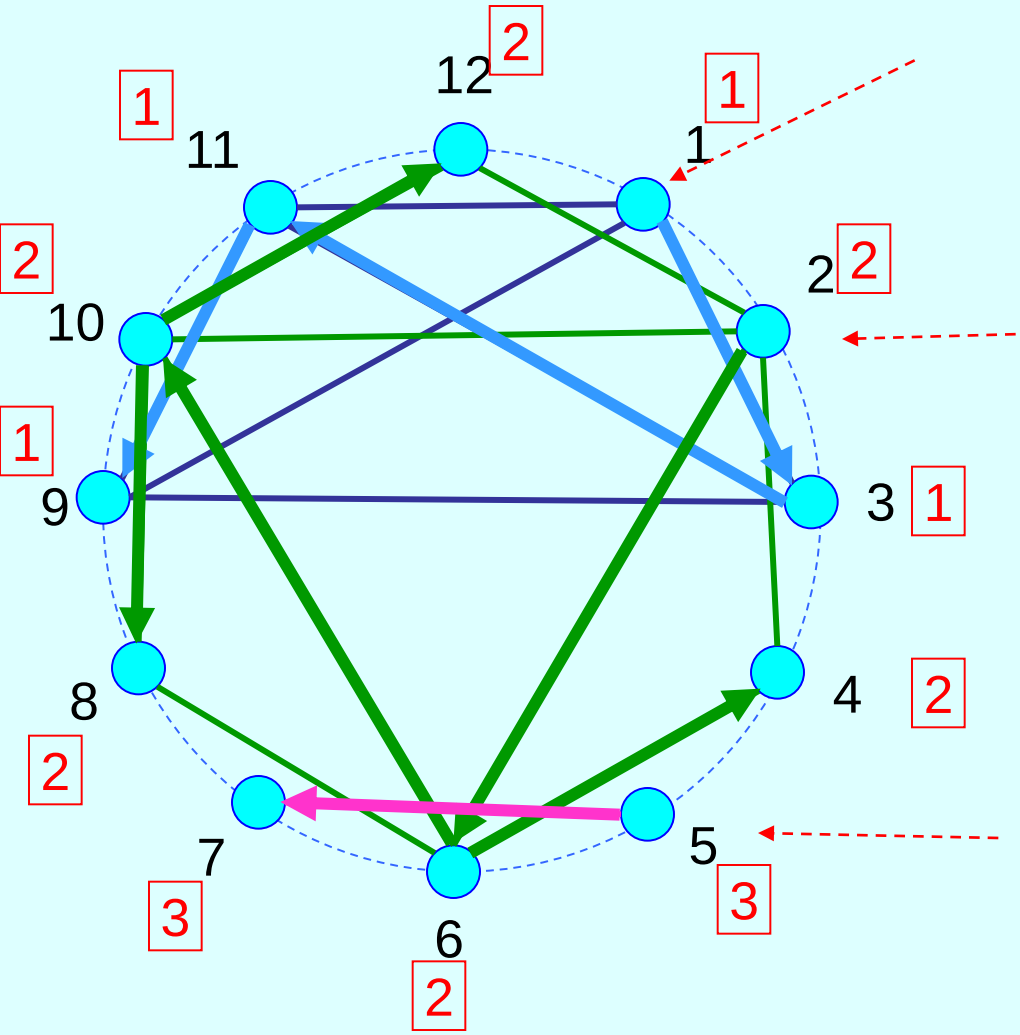
Пример (нахождение связных компонент)



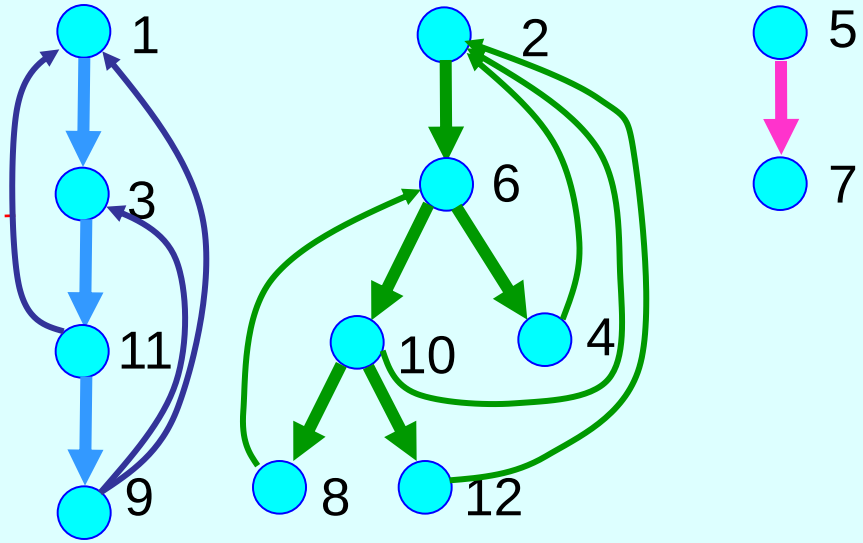
1	3, 9, 11
2	6, 4, 10 , 12
3	1, 11, 9
4	2, 6
5	7
6	10, 8, 4, 2
7	5
8	10, 6
9	3, 1, 11
10	8, 6, 2, 12
11	1, 3, 9
12	2, 10

Все вершины помечены. Обход закончен!

Пример (нахождение связных компонент)



Лес деревьев (связных компонент)

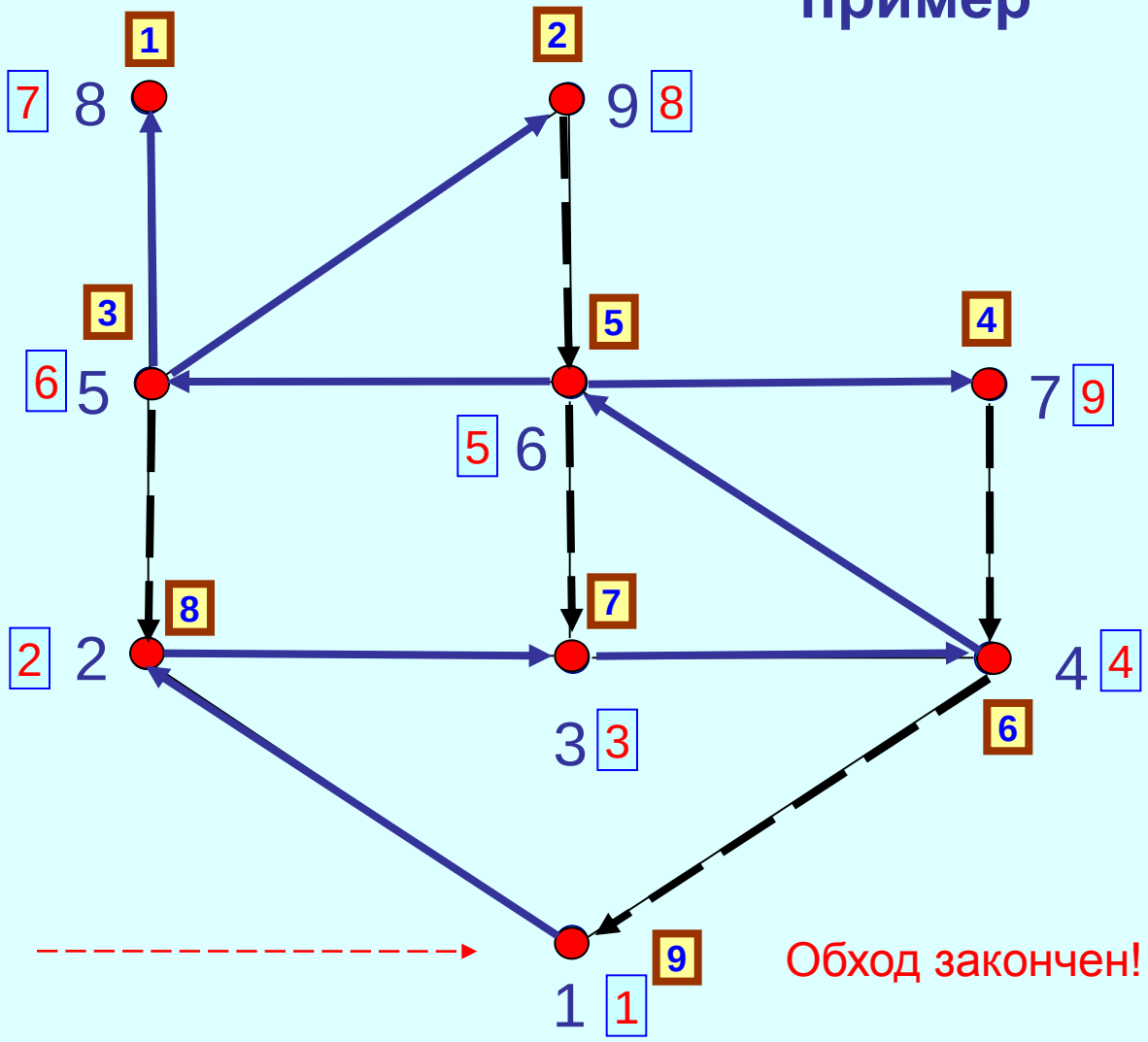


Ребра графа:

1. ***Прямые*** (древесные)
2. ***Обратные*** (к ранее пройденным вершинам)

Построение остовного дерева и множества обратных ребер при поиске в глубину (Tree of Depth First Search -TDFS)

Построение остовного дерева
пример



★	1	2, 4
★	2	3, 5
★	3	2, 4, 6
★	4	1, 3, 6, 7
★	5	2, 6, 8,9
★	6	3, 4, 5, 7, 9
★	7	4, 6
★	8	5
★	9	5, 6

1 - Номер *посещения*

1 - Номер *использования*

Построение остовного дерева
пример

Задание:

Выполнить поиск в глубину для этого же графа, в т. ч. построить остов и обратные ребра

- Начиная с любой другой вершины
- Изменив списки смежности (порядок элементов в списке)

Свойство DFS-остова (глубинного остовного дерева)

Пусть (V, T) – DFS-остов графа G = (V, E) и пусть {u, v} ∈ E. Тогда либо u потомок v (в дереве), либо v потомок u.

Доказательство.

Пусть v посещена раньше, чем u. По завершении TDFS(v), будет NumVert[v] < NumVert[u], т.к. {u,v} ∈ E. Но это значит, что в TDFS добавились ребра, содержащие путь из v в u. Отсюда следует, что v лежит на пути от корня в u, т.е. u потомок v.

Аналогично рассуждаем в случае, если u посещена раньше, чем v.

Свойство **TDFS-остова (глубинного остовного дерева)**

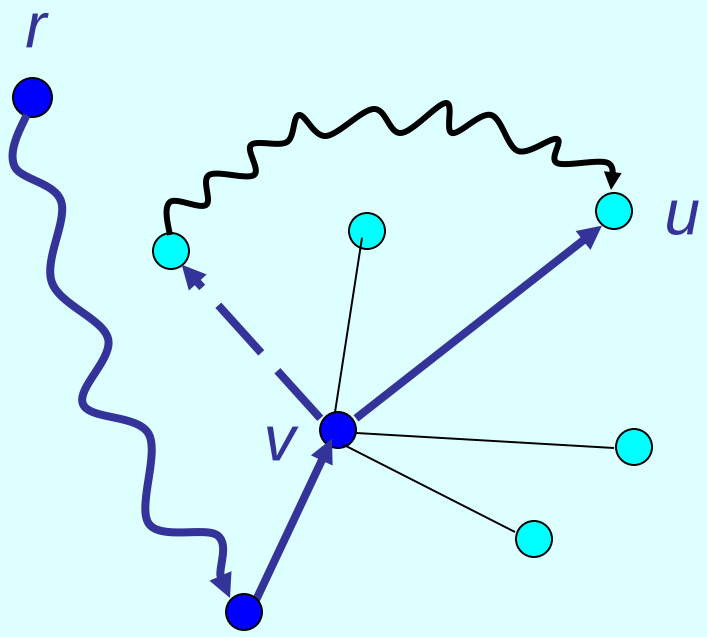


Иллюстрация к доказательству:

- $\{u,v\} \in E$
- v посещена раньше, чем u
- NumVert[v] < NumVert[u]
- r – корень TDFS

Алгоритм Борувки построения МОД

O(m*log n)

Вход : V - множество вершин заданного графа G=(V,E)

E - множество ребер заданного графа G=(V,E)

D [1..n,1..n] - матрица весов (или ф-ия D(.,.))

Выход: T - множество ребер МОД

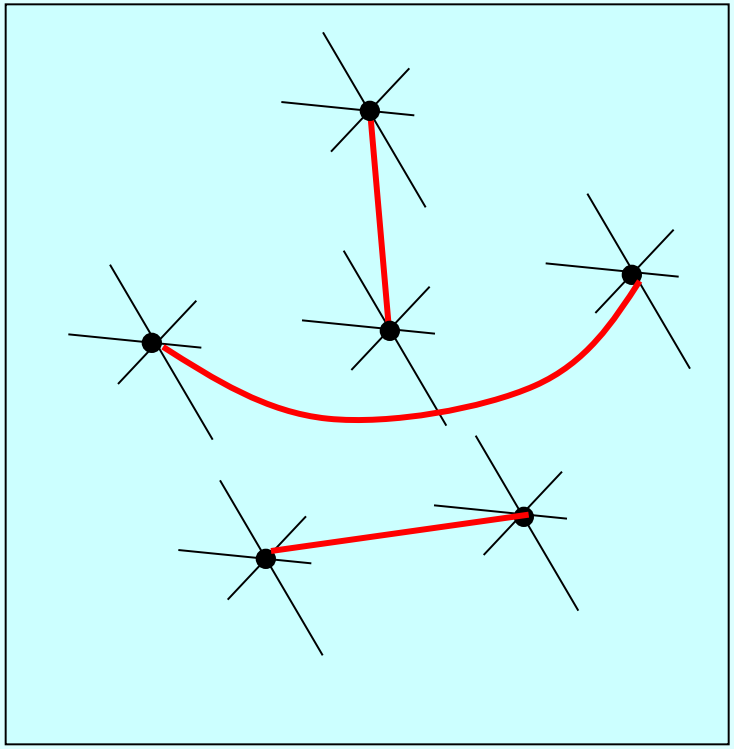
Рабочие: C - множество связных компонент графа (V,T), т.е. леса;

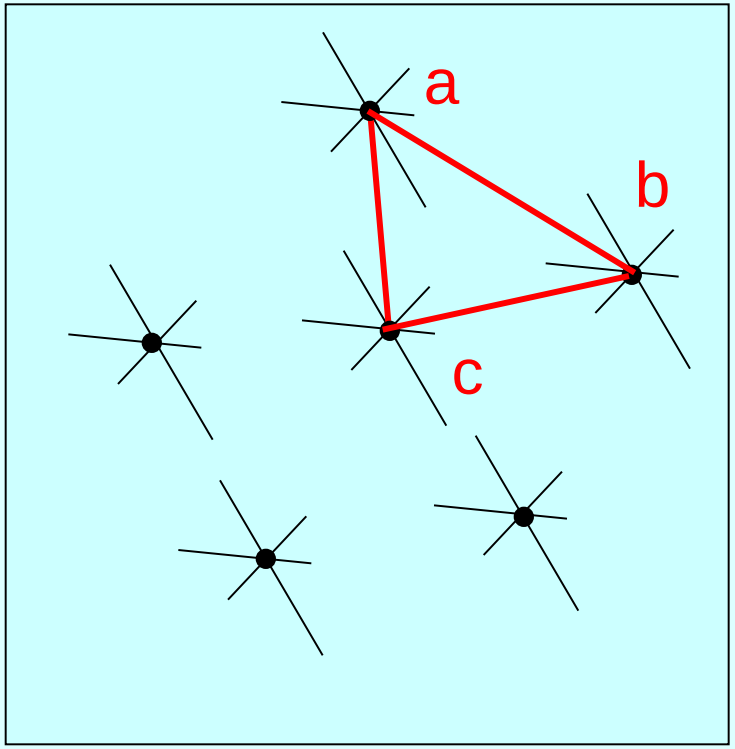
C = $\{S_1,...,S_k\}$, где S_j – связная компонента леса, т.е. дерево

Кратч[1..n] – массив ребер; Кратч[i] - ребро, кратчайшее из всех ребер, имеющих ровно один конец (вершину) в дереве (в связной компоненте) $S_i=(V_i,T_i)$;

Min[1..n] - вспомогательный массив для определения Кратч[i]

Идея алгоритма





$W[a, b] = \min \{W[u, v]: u \in V(a), v \notin V(a)\}$

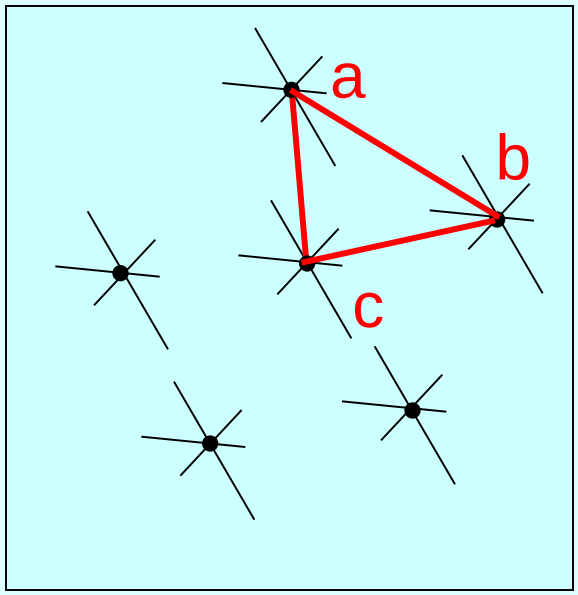
$W[b, c] = \min \{W[u, v]: u \in V(b), v \notin V(b)\}$

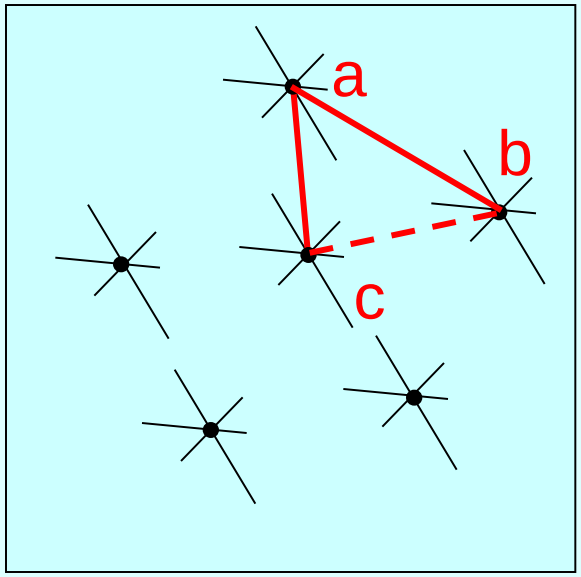
$W[c, a] = \min \{W[u, v]: u \in V(c), v \notin V(c)\}$

$(W[b, c] < W[a, b]) \&$
$(W[c, a] < W[b, c]) \rightarrow$
$(W[c, a] < W[a, b])$

!?!?

Это в случае, когда все веса различны





Случай равных весов

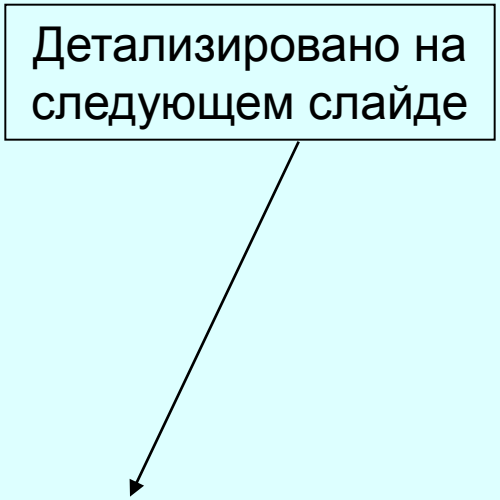
(W[b, c] = W[a, b]) &
(W[c, a] = W[b, c]) &
(W[c, a] = W[a, b])
Цикл !?!?

Разрыв цикла

При выборе минимума в случае равных весов выбирать ребро, соединяющее с компонентой, имеющей меньший номер.

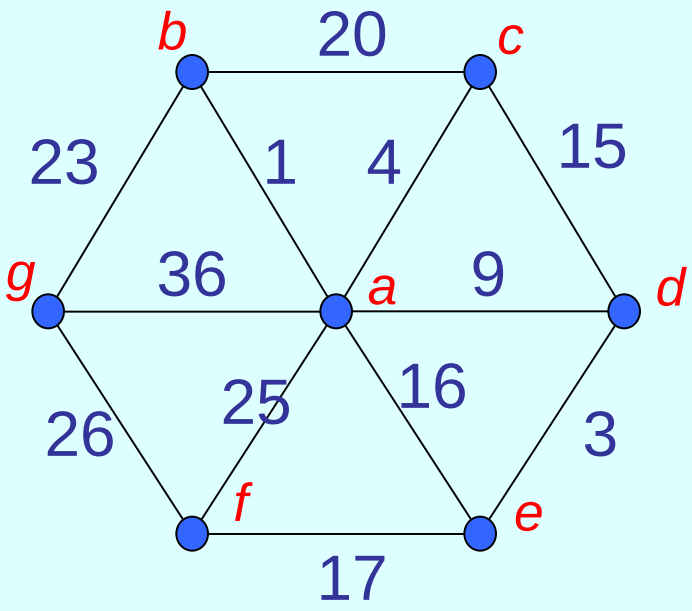
Например, a < b < c.

Тогда компонента с максимальным номером (c) не будет выбрана (при равных весах)

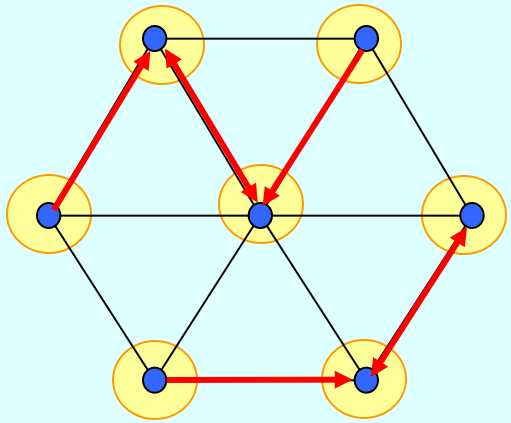

Детализировано на
следующем слайде

Алгоритм Борувки построения МОД

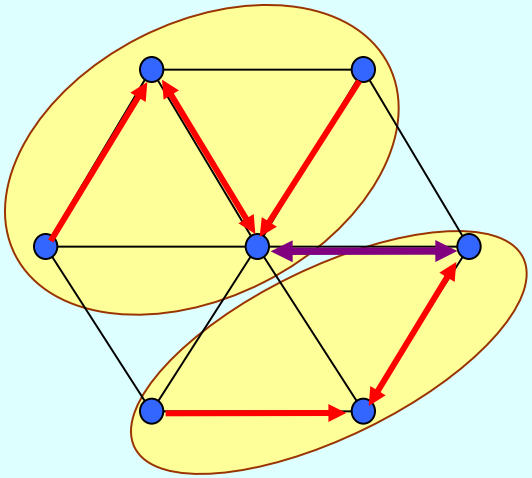
Пример



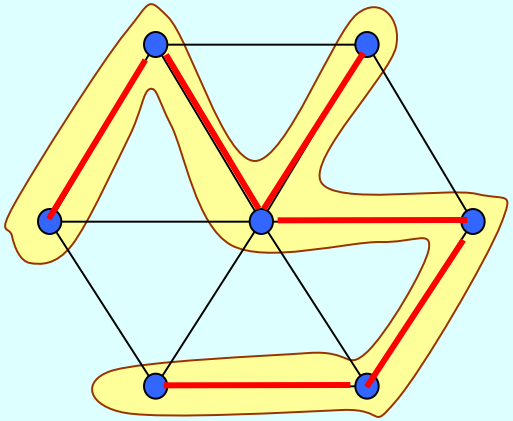
1 этап



2 этап



Результат



Сложность алгоритма

На каждом новом этапе число деревьев уменьшается *не менее, чем вдвое* (!).

Т. о. всего не более чем $\log_2 n$ этапов.

Каждый этап имеет стоимость $O(m)$.

Общая сложность $O(m \log_2 n)$.