

Задачи, наиболее часто встающих перед программистами, – это задачи **сортировки** и **поиска**.

Данные задачи применяются как сами по себе, так и входят как подзадачи в состав более сложных задач.

Например, дан массив  $N$  элементов, из которого надо удалить все дублирующиеся элементы. Решение сравнения каждого элемента с остальными потребует  $T(N^2)$  времени. Однако если предварительно отсортировать массив (на что, как увидим позже, требуется  $T(N \cdot \log_2(N))$  времени), то найти все дубли можно за  $T(N)$  времени, сравнивая только соседние элементы, так что общее время решения задачи –  $T(N \cdot \log_2(N))$ .

Здесь задача сортировки вошла в другую задачу в качестве подзадачи.

Задача сортировки формулируется следующим образом:

На вход алгоритма подается последовательность из  $n$  элементов  $a_1, a_2, \dots, a_n$ ; на выходе требуется получить некоторую перестановку входной последовательности  $a'_1, a'_2, \dots, a'_n$  такую, что  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

Алгоритмы сортировки можно разделить на алгоритмы внутренней сортировки для сортировки данных, хранящихся во внутренней оперативной памяти компьютера, и внешней сортировки – для сортировки больших объемов данных, хранящихся в файлах внешней (например, дисковой) памяти. В данном учебном курсе будут рассматриваться только алгоритмы внутренней сортировки.

**Пузырьковая сортировка** по возрастанию – проходит по массиву снизу вверх (*от последнего элемента к первому*), сравнивая каждый элемент массива с расположенным выше, и если верхний больше, то меняет их местами. При этом проходе наименьший элемент – "всплывет" наверх. Операция продолжается пока наименьший элемент не станет первым.

Затем операция повторяется над подмножеством массива с номерами (индексами) элементов от 2 до N, затем над подмножеством от 3 до N и так до подмножества N-1, N. То есть, до тех пор пока массив не будет отсортирован по возрастанию элементов.

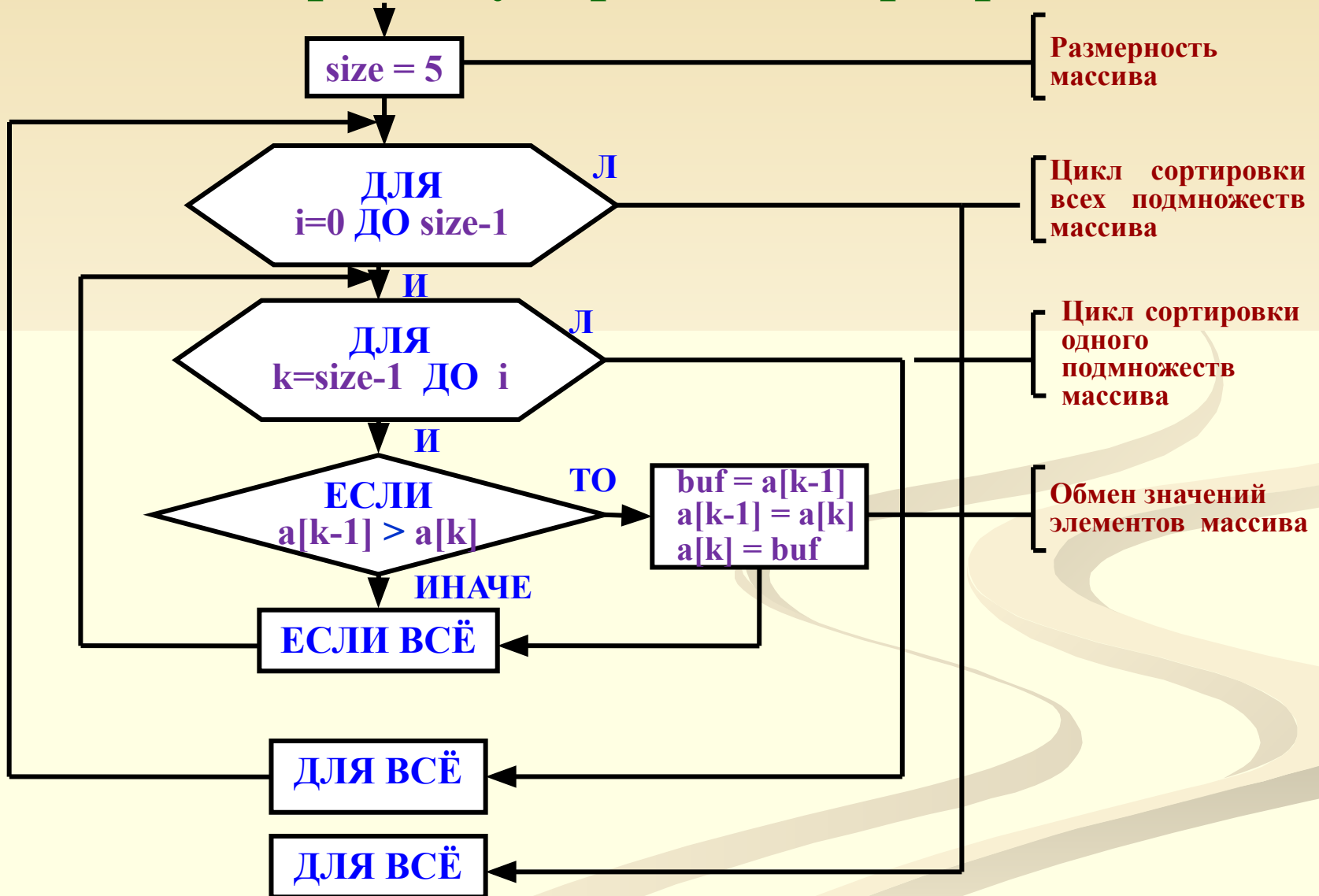
*(При формировании условия сравнения "наибольший наверх" будет происходить сортировка по убыванию элементов массива).*

Индекс	Исходный массив	1 шаг	2 шаг	3 шаг
1	6	2	2	2
2	4	6	3	3
3	3	4	6	4
4	2	3	4	6

На каждом шаге происходит три перестановки значений элементов.

# Сортировка

## Алгоритм пузырьковой сортировки



Написать программу пузырьковой сортировки на С.

## Пузырьковая сортировка

C \ C++

### Практическое занятие

```
// Сортировка массива целых чисел методом "пузырька" – по возрастанию
#include <stdio.h>
#include <conio.h>
#define sz 5 // размерность массива
void main ()
{ int a[sz]; /*массив целых чисел*/
  int i; //счетчик циклов сортировки
  int buf; // буфер, исп. при обмене элементов массива
  int k; // текущий индекс элемента массива
  printf ("\nВведите в одной строке %i", sz);
  printf (" целых чисел и нажмите Enter\n");
  printf ("-> ");
  for (k = 0; k < sz; k++) scanf ("%i", &a[k]);
  // Сортировка
  for (i = 0; i < sz-1; i++)
  {
    for (k = sz-1; k > i; k--)
    {
      if (a[k-1] > a[k])
      // Меняем местами k-тый и k-1 элементы
      buf = a[k-1]; a[k-1] = a[k]; a[k] = buf;
    }
  }
  // Цикл сортировки закончен
  // Вывод отсортированного массива
  printf ("Отсортированный массив\n");
  for (k = 0; k < sz; k++) printf ("%i ", a[k]);
}
```

# Сортировка

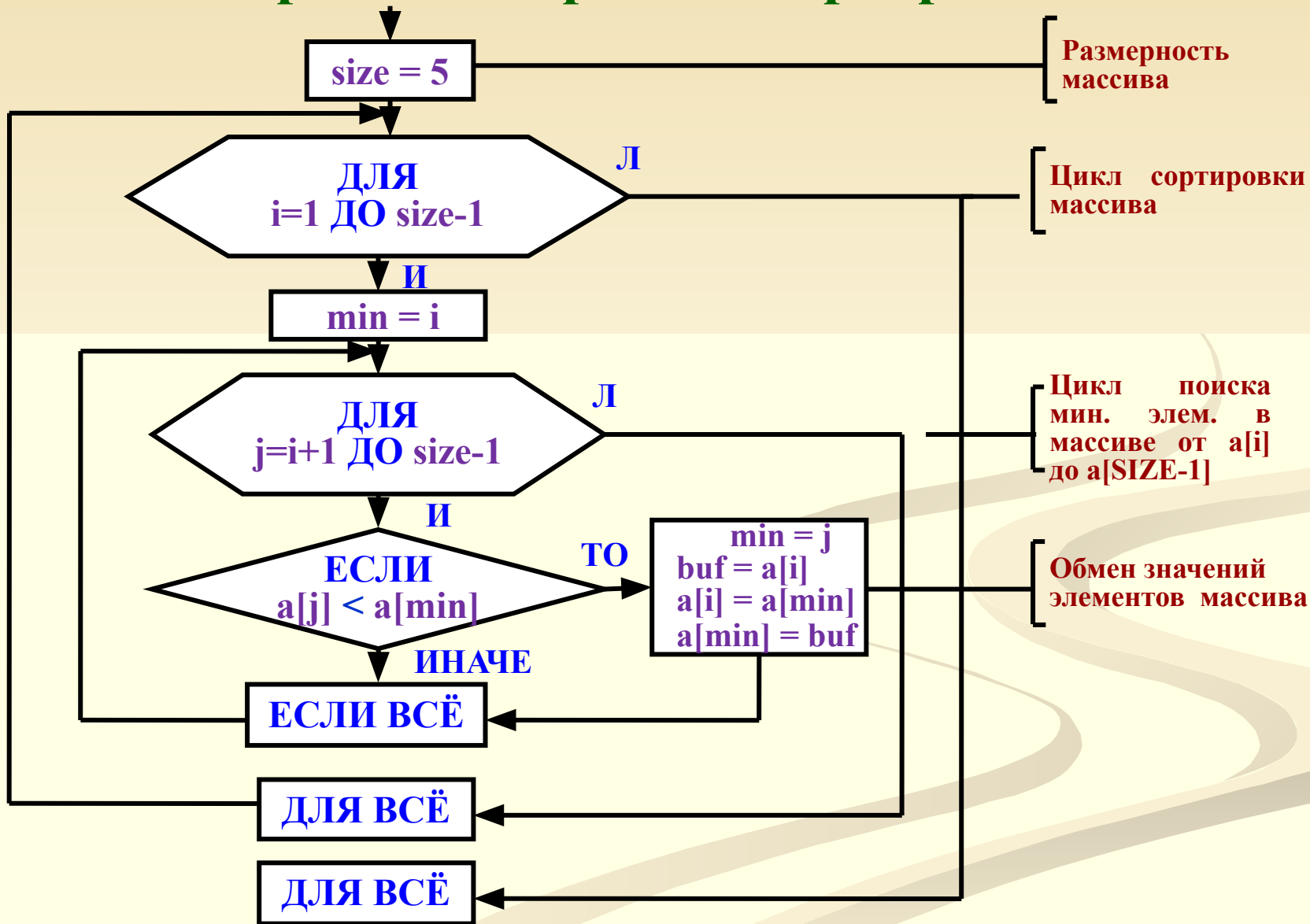
Главный недостаток пузырьковой сортировки – большое количество перестановок элементов. Алгоритм *выборочной сортировки* устраняет этот недостаток, здесь элемент сразу занимает свою конечную позицию.

**Выборочная сортировка** – происходит следующим образом:

- 1.** Просматривается весь первичный массив, определяется наименьший (наибольший) элемент массива и затем осуществляется единственный обмен в текущем массиве.
- 2.** Потом просматривается массив-подмножество без наименьшего (наибольшего) элемента, определяется наименьший (наибольший) элемент подмножества и снова осуществляется единственный обмен в текущем подмножестве массива.
- 3.** Шаг **2** повторяется пока весь массив не будет отсортирован.

# Сортировка

## Алгоритм выборочной сортировки



Написать программу выборочной сортировки на С.

# Сортировка

## Выборочная сортировка

C \ C++

### Практическое занятие

```

// Сортировка мас. целых чисел выборочн. методом
#include <stdio.h>
#include <conio.h>
#define sz 5 // размерность массива
void main ()
{ int a[sz]; // массив целых чисел
  int i; // № элем., от которого ведется поиск мин. элем.
  int min; // № мин. элем. в части мас. от i до конца мас.
  int j; // № элемента сравниваемого с мин.
  int buf; // буфер, исп. при обмене элементов массива
  int k; // индекс для ввода и вывода
  printf ("\nВведите в одной строке %i", sz);
  printf (" целых чисел и нажмите Enter\n");

  for (k=0; k<sz; k++) scanf ("%i", &a[k]);
  // Сортировка
  for (i = 0; i < sz-1; i++)
  { // Поиск мин. элем. в части мас. от a[i] до a[sz]
    min = i; for (j = i+1; j < sz; j++)
      if (a[j] < a[min]) min = j;
  // Меняем местами a[min] и a[i]
    buf = a[i]; a[i] = a[min]; a[min] = buf;
  }
  // Цикл сортировки закончен
  // Вывод отсортированного массива
  printf ("Отсортированный массив\n");
  for (k = 0; k<sz; k++) printf ("%i ", a[k]);
}

```

# Сортировка

Тем не менее оба метода и пузырьковая и выборочная сортировка сравнительно неэффективны.

Среднее время работы этих алгоритмов пропорционально  $N^2$   
 Существуют более быстрые методы сортировки: быстрая сортировка (Quicksort) и сортировка слиянием (метод Шелла).

Среднее время работы этих методов пропорционально  $N \cdot \log_2(N)$

Зависимость времени сортировки от количества элементов массива ( $N$ ) и

$N$	$N^2$	$N \cdot \log_2(N)$	Зависимость времени сортировки от мощности алгоритма [раз] $N^2 / N \cdot \log_2(N)$	Зависимость времени сортировки от количества элементов массива [раз]
$10^2$	$10^4$	$10^2 \cdot \log_2(10^2) = 6,64 \cdot 10^2$	$10^4 / (6,64 \cdot 10^2) = 1,5 \cdot 10^1$	-
$10^3$	$10^6$	$10^3 \cdot \log_2(10^3) = 9,97 \cdot 10^3$	$10^6 / (9,97 \cdot 10^3) = 1,0 \cdot 10^2$	$1,0 \cdot 10^2 / 1,5 \cdot 10^1 = 6,7$
$10^6$	$10^{12}$	$10^6 \cdot \log_2(10^6) = 19,93 \cdot 10^6$	$10^{12} / (19,93 \cdot 10^6) = 5,0 \cdot 10^4$	$5,0 \cdot 10^4 / 1,0 \cdot 10^2 = 500$



# Сортировка

**Быстрая сортировка** (автор Чарльз Хоар, в 1962г.)

– Quicksort – **Метод сортировки разделением:**

Из массива выбирается **опорный элемент P**.

- Сравнивая элементы массива с **P** и разделяем (сортируем) массив на 2-а подмассива (подмножества). Слева от **P** элементы меньше и равные **P**, а справа – большие или равные.
- Для обоих подмножеств, если в них больше 1-го элемента, делается та же процедура.
- Процесс повторяется для каждой части массива пока он не будет отсортирован.

Опорный элемент выбирается или случайным образом, или как среднее некоторого количества значений массива (например, первого и последнего).

# Сортировка

## Алгоритм быстрой сортировки

### Быстрая сортировка (разделением):

- 1) Берем массив  $M[N]$ . Назначаем индексы  $I$  и  $J$ .
- 2) Устанавливаем начальные значения индексов  $I=1$  и  $J=N$ .
- 3) Выбираем **опорный элемент**  $P = M[K]$ , где  $K = (I + J) / 2$ .
- 4) Сравниваем  $M[I] \leq P$ , если ДА, то увеличиваем  $I$  ( $I=I+1$ ).
- 5) Затем сравниваем  $M[J] \geq P$ , если ДА, то уменьшаем  $J$  ( $J=J-1$ ).
- 6) Если НЕТ и  $I \leq J$ , то меняем местами  $M[I]$  и  $M[J]$ .
- 7) Повторяем шаги 4-6 пока  $I \leq J$ .
- 8) В результате массив разделяется на 2-е части, слева от  $P$  элементы меньше или равны  $P$ , а справа – больше или равны.
- 9) Над каждой частью (подмножеством) массива повторяем шаги 2-7. Получаем 4-е подмножества.
- 10) Над каждым подмножеством повторяем шаги 2-7. Выполняем эти операции пока массив не будет отсортирован.

# Сортировка

## Алгоритм быстрой сортировки

1. Массив  $M[N]$ . Назнач.  $I$  и  $J$ .
2. Уст. нач. знач.  $I=1$  и  $J=N$ .
3. Выб. **Опор. элем.**  $P = M[(M[1]+M[N])/2]$ .
4. Сравн.  $M[I] \leq P$ , если ДА, то  $I=I+1$ .
5. Сравн.  $M[J] \geq P$ , если ДА, то  $J=J-1$ .
6. Если НЕТ и  $I \leq J$ , то меняем местами  $M[I]$  и  $M[J]$ .
7. Повторяем шаги 2-6 пока  $I \leq J$ .
8. Массив раздел. на 2-е части, слева от  $P$  элементы  $\leq P$ , а справа  $\geq P$ .
9. Над кажд. подмнож. мас. повт. шаги 2-7. Получ. 4 подмнож.
10. Над кажд. подмнож. повт. шаги 2-7. Вып. эти операции пока массив не будет отсортирован.

Пример:

1-2. {5 2 7 2 13 3 8 15 19}

3.  $P=13$

4-7. {5 2 7 2 13 3 8 15 19} -

меняем местами 13 и 8 =

{5 2 7 2 8 3 13 15 19}

8. Массив разделен на две части по 13.

9-10. Сортируем подмножество

{5 2 7 2 8 3 13 15 19}

2-7.  $P=7 \rightarrow$  {5 2 7 2 8 3 13 15 19} =

{5 2 3 2 8 7 13 15 19} -

$P=2 \rightarrow$  {5 2 3 2 8 7 13 15 19} =

{2 2 3 5 8 7 13 15 19} -

$P=8 \rightarrow$  {2 2 3 5 8 7 13 15 19} =

{2 2 3 5 7 8 13 15 19}

Нарисовать алгоритм быстрой сортировки.

# Сортировка

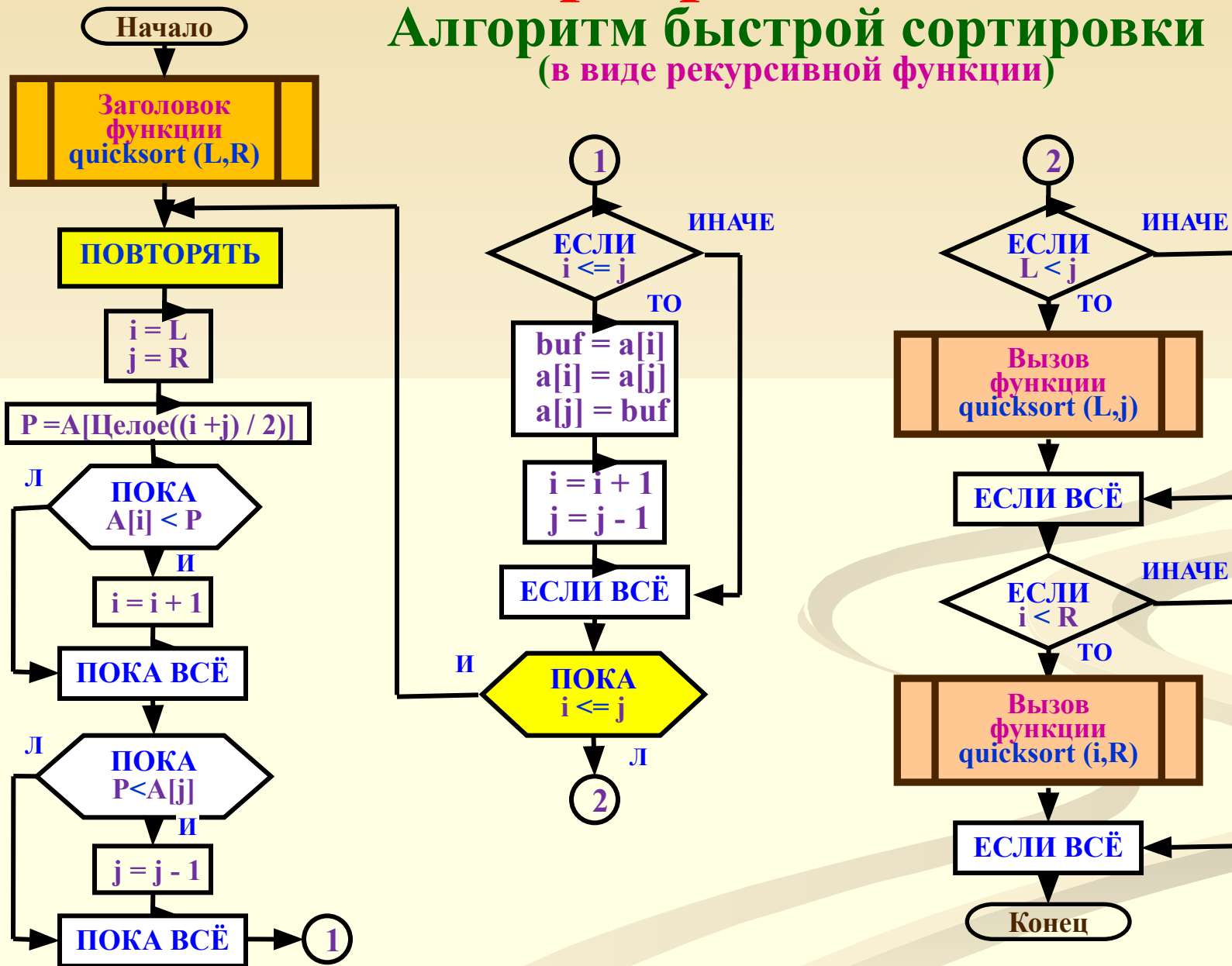
## Алгоритм быстрой сортировки



Алгоритм быстрой сортировки повторяется для каждого подмножества – прямой смысл реализовать эту сортировку в виде рекурсивной функции 12

# Сортировка

## Алгоритм быстрой сортировки (в виде рекурсивной функции)



Написать программу быстрой сортировки на С.

# Сортировка

C / C++

## Быстрая сортировка

C / C++

```

void quicksort (long High, long Low)
// Функция быстрой сортировки
{ long i, j;  int p, temp;
// Инициализация нижней границы
  i = Low;
// Инициализация верхней границы
  j = High;
// опорный элемент
  p = array[(int) (Low+High)/2];
do { while (array[i] < p) i++;
      while (array[j] > p) j--;
      if (i<=j) // Если верно, то обмен
      { temp = array[i]; array[i] = array[j];
        array[j] = temp; i++; j--;  }  }
while (i<=j); // пока индексы не пересекутся

  if (j > Low) quicksort (j, Low);
/* Если подмассив [j, Low] более одного
элемента, он сортируется функцией quicksort */
  if (High > i) quicksort (High, i);
// Аналогично для [High, i]
}

```

```

#include <iostream.h>
#include <conio.h>
int array[10000]; // Объявление массива
/* Функция - Быстрая сортировка
..... */
main() // Главная функция
{ int i; int size; // количества элементов
  cout << "\n Введите количество элементов
сортируемого массива size = ";
  cin >> size;
  for (i=0; i<size; i++)  cin >> array[i];
// Чтение очередного элемента массива
  for (i=0; i<size; i++)
    cout << array[i] << " ";
  quicksort (size-1, 0);
// Вывод отсортированного массива
  cout << "\n
Отсортированный массив\n ";
  for (i=0; i<size; i++)
    cout << array[i] << " ";
  return 0;
}

```

# Сортировка

## Сортировка методом Шелла

Суть этого метода заключается в сравнении элементов массива, разделенных одинаковым расстоянием, таким образом, чтобы элементы были упорядочены по расстоянию. Затем это расстояние делится пополам и процесс повторяется.

### Алгоритм сортировки Шелла:

В этом методе первоначально рассматриваются элементы отстоящие друг от друга на расстояние  $d = \lfloor n/2 \rfloor$ , где  $\lfloor \ ]$  - операция взятия целой части, и  $n$  - количество элементов исходного массива.

На следующих шагах  $d$  меняется по закону  $d = \lfloor d/2 \rfloor$ , при  $d = 1$  метод Шелла вырождается в метод стандартного обмена ("Метод Пузырка")

# Сортировка

## Сортировка методом Шелла

### Рассмотрим пример:

Дано множество {6,3,4,8,2,9} ->

$$d = \lfloor n/2 \rfloor = \lfloor 6/2 \rfloor = 3 \rightarrow$$

{6,3,4,8,2,9} - сравниваем 6 и 8 ->

{6,2,4,8,3,9} - сравниваем 3 и 2, переставляем ->

{6,3,4,8,2,9} - сравниваем 4 и 9 ->

далее  $d = \lfloor d/2 \rfloor = \lfloor 3/2 \rfloor = 1$ .

И алгоритм выродился в метод "Пузырька"

В этом примере не очень видна эффективность метода, но представьте, что вы сортируете **1000** элементов. Этот метод обеспечивает более быстрое перебрасывание больших элементов **вправо** и **меньших влево**, чем метод "Пузырька" и этим обеспечивает большее быстроедействие.



C \ C++

# Сортировка

## Сортировка методом Шелла

```

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#define size 20

int mass[size];

// сортировка методом Шелла
void ShellSort(int n, int mass[])
{
    int i, j, step, tmp;
    for (step = n / 2; step > 0; step /= 2)
        for (i = step; i < n; i++)
            { tmp = mass[i];
              for (j = i; j >= step)
                  { if (tmp < mass[j - step])
                      mass[j] = mass[j - step];
                    else break;
                  }
              mass[j] = tmp; }
}

```

*см. продолжение*

*продолжение*

```

int main()
{
    srand(time(NULL));
    // ввод элементов массива
    for (int i = 0; i < size; i++)
        {
            mass[i]=rand() % 100;
            printf ("%i ", mass[i]);
        }
    // сортировка методом Шелла
    ShellSort(size, mass);
    // вывод отсортированного массива на экран
    printf("отсортированный массив:\n");
    for (int i = 0; i < size; i++)
        printf("%d ", mass[i]);
    return 0;
}

```

# Поиск

**Поиск** необходимой компоненты структуры данных – одна из важнейших задач обработки данных.

Для решения задачи поиска необходимо, чтобы данные в памяти ЭВМ были организованы определенным образом. Основные способы организации данных: массивы элементов одинакового типа, структуры данных, линейные списки, деревья, произвольные графы.

Алгоритмы поиска существенно зависят от способа организации данных.

Рассмотрим алгоритмы поиска в МАССИВАХ:

**а) последовательный (линейный) поиск** -- простейший метод поиска элемента, находящегося в неупорядоченном массиве данных, это **последовательный** просмотр каждого элемента массива, продолжающийся до тех пор, пока не будет найден желаемый элемент. Если просмотрен весь массив, но элемент не найден – значит он отсутствует в массиве. Для последовательного поиска в среднем требуется  $(N+1)/2$  сравнений, а в худшем  $N$ . Линейный поиск может применяться и для упорядоченных (отсортированных) массивов, **НО эффективнее использовать:**

**б) бинарный (двоичный, дихотомический, логарифмический) поиск** – он состоит в разделении **упорядоченного** массива пополам, определении в какой половине находится искомый элемент, затем это половина снова разделяется пополам и так пока полученное подмножество из одного элемента не станет равным искомому. Для бинарного поиска в худшем случае требуется  $\log_2(N)$  сравнений.

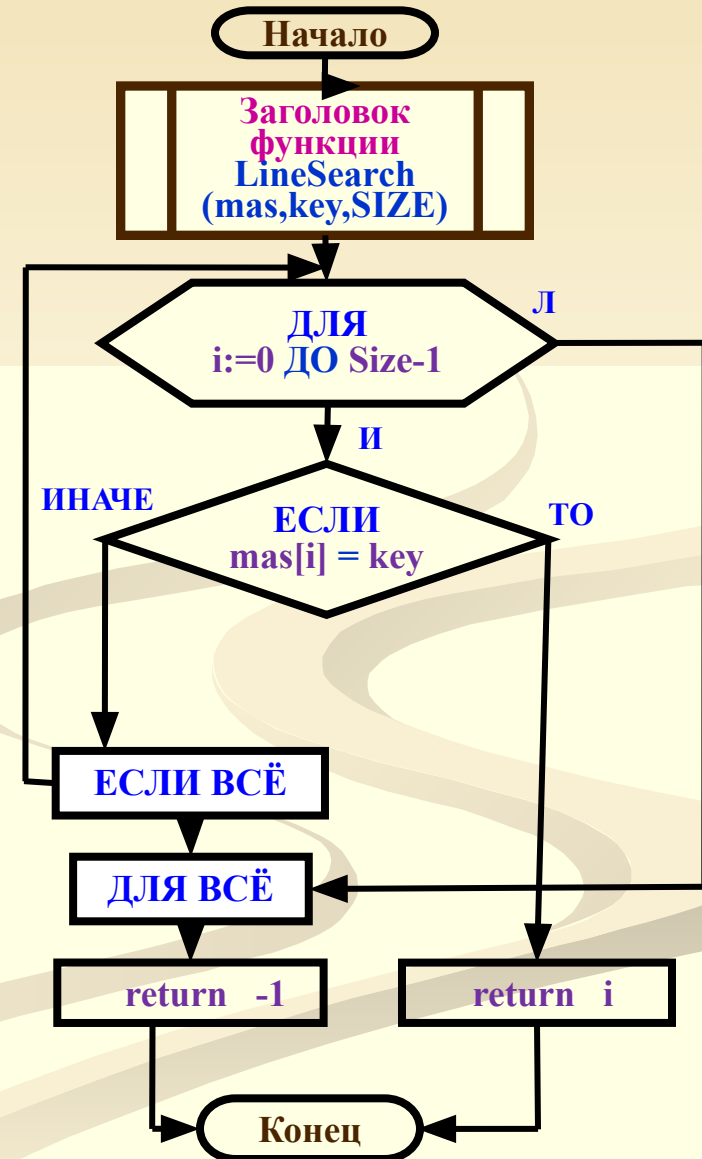
C / C++

# Поиск

## Алгоритм и функция последовательного поиска

### Алгоритм функции

```
// Последовательный поиск
#include <iostream.h>
#include <stdio.h>
#define size 5 // Размерность массива
/* Функция последовательного поиска,
   возвращает индекс искомого элемента массива */
int seq_search (int items[], int count, char key)
{ int t;
  for (t=0; t < count; ++t)
    if (key == items[t])
      return t; // элемент найден
  return -1; // элемент не найден
}
main()
{ int array[size], N; // массив
  int k, i; // k-искомый элемент
  cout << "\n Введите " << size << " элемента(ов), после
           ввода каждого элемента -> Enter\n";
  for (i=0; i<size; i++) cin >> array[i]; // Ввод элемента
  cout << "Введенный массив\n";
  for (i=0; i<size; i++) cout << array[i] << " ";
  cout << "\n Введите искомый элемент массива - ";
  cin >> k;
  N=seq_search (array, size, k); // Вызов функции
  if (N==-1) cout << "Такого элемента в массиве нет";
  else
  cout << "\nНомер искомого элемента в массиве--"<<N+1;
  return 0;
}
```



C / C++

# Поиск

## Написать функцию бинарного поиска

Алгоритм функции

```
// Бинарный поиск
#include <iostream.h>
#define size 5 // Размерность массива
/* Функция Бинарного поиска,
   возвращает индекс искомого элемента массива */
int bin_search (int ar[], int size, int key);
{ int low, high, mid;
  low = 0; high = size- 1;
  while(low <= high)
  { mid = (int) (low + high)/2;
    if (key < ar[mid]) high = mid - 1;
    else if(key > ar[mid]) low = mid + 1;
    else return mid; };
  return -1; };
main()
{ int array[size], N, k, i;
  cout << "\n Введите " << size << " элемента(ов), \
  после ввода каждого элемента -> Enter\n";
  for (i=0; i<size; i++) cin >>array[i]; // Ввод элемента
  cout << "Введенный массив\n";
  for (i=0; i<size; i++) cout << array[i] << " ";
  cout << "\n Введите искомый элемент массива - ";
  cin >> k;
  N=bin_search (array, size, k); // Вызов функции
  if (N==-1) cout << "Такого элемента в массиве нет";
  else
  cout << "\nНомер искомого элемента в массиве--"<<N+1;
  return 0;
}
```

