



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное  
учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

---

ФАКУЛЬТЕТ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА «ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ И ИНФОРМАЦИОННЫЕ  
ТЕХНОЛОГИИ» (ИУ7)

НАПРАВЛЕНИЕ ПОДГОТОВКИ 09.03.01 Информатика и вычислительная техника

**О Т Ч Е Т**

по лабораторной работе № 3

Название: Трудоёмкость алгоритмов сортировки

Дисциплина: Анализ алгоритмов

Студент

ИУ7-52Б

(Группа)

\_\_\_\_\_  
(Подпись, дата)

Е.В. Брянская

(И.О. Фамилия)

Преподаватель

Л.Л. Волкова

\_\_\_\_\_  
(Подпись, дата)

(И.О. Фамилия)

Москва, 2020

# Оглавление

<b>Введение</b>	<b>3</b>
<b>1 Аналитическая часть</b>	<b>4</b>
1.1 Пузырьковая сортировка . . . . .	4
1.2 Сортировка вставками . . . . .	4
1.3 Поразрядная сортировка . . . . .	5
<b>2 Конструкторская часть</b>	<b>6</b>
2.1 Сортировка пузырьком . . . . .	6
2.2 Сортировка вставками . . . . .	7
2.3 Поразрядная сортировка . . . . .	7
2.4 Требования к ПО . . . . .	7
2.5 Заготовки тестов . . . . .	8
<b>3 Технологическая часть</b>	<b>12</b>
3.1 Выбранный язык программирования . . . . .	12
3.2 Листинг кода . . . . .	12
3.3 Результаты тестов . . . . .	16
3.4 Оценка трудоёмкости . . . . .	20
3.5 Оценка времени . . . . .	20
<b>4 Исследовательская часть</b>	<b>21</b>
<b>Заключение</b>	<b>22</b>
<b>Список литературы</b>	<b>23</b>

# Введение

**Трудоёмкость алгоритма** - это зависимость стоимости операций от линейного(ых) размера(ов) входа(ов) [1].

Модель вычислений трудоёмкости должна учитывать следующие оценки.

- 1) Стоимость базовых операций. К ним относятся: =, +, -, \*, /, ==, !=, <, <=, >, >=, %, +=, -=, \*=, /=, [ ], < <, > >. Каждая из операций имеет стоимость равную 1.
- 2) Оценка цикла. Она складывается из стоимости тела, инкремента и сравнения.
- 3) Оценка условного оператора if. Положим, что стоимость перехода к одной из веток равной 0. В таком случае, общая стоимость складывается из подсчета условия и рассмотрения худшего и лучшего случаев.

Оценка характера трудоёмкости даётся по наиболее быстрорастущему слагаемому.

**Сортировка** - процесс перегруппировки заданного множества объектов в некотором определенном порядке. Сортировка предпринимается для того, чтобы облегчить последующий поиск элементов в отсортированном множестве.

В этой лабораторной работе будет оцениваться трудоёмкость алгоритмов сортировки. Будут рассмотрены следующие алгоритмы:

- 1) сортировка пузырьком;
- 2) сортировка вставками;
- 3) поразрядная сортировка.

# 1. Аналитическая часть

**Цель** данной работы – оценить трудоёмкость алгоритмов сортировки.

Для достижения поставленной цели необходимо решить ряд следующих **задач**:

- 1) дать математическое описание;
- 2) описать алгоритмы сортировки;
- 3) дать теоретическую оценку трудоёмкости алгоритмов;
- 4) реализовать эти алгоритмы ;
- 5) провести замеры процессорного времени работы алгоритмов на материале серии экспериментов;
- 6) провести сравнительный анализ алгоритмов.

**Задача сортировки** состоит в перестановке (переупорядочивании) входной последовательности из  $n$  чисел  $a_1, a_2, \dots, a_n$ , так, чтобы выполнялось условие

$$a'_1 \leq a'_2 \leq \dots \leq a'_n \quad (1.1)$$

(в случае сортировки по неубыванию). Аналогичным образом осуществляется сортировка по невозрастанию.

## 1.1. Пузырьковая сортировка

Представляет собой популярный, но не эффективный алгоритм сортировки. В его основе лежит многократная перестановка соседних элементов, нарушающих порядок сортировки [2].

## 1.2. Сортировка вставками

Считается простым алгоритмом сортировки. На каждом шаге алгоритма для очередного элемента находится подходящая позиция в уже отсортированной части массива и осуществляется вставка этого элемента.

### 1.3. Поразрядная сортировка

Также называется LSD-сортировкой (Least Significant Digit - по младшей цифре). В этом алгоритме массив несколько раз перебирается и элементы группируются в зависимости от того, какая цифра находится в определённом разряде.

Сначала значения сортируются по единицам, затем по десяткам, сохраняя отсортированность по единицам внутри десятков, затем по сотням, сохраняя отсортированность по десяткам и единицам внутри сотен и так далее.

После обработки всех разрядов массив становится упорядоченным.

## 2. Конструкторская часть

Рассмотрим выбранные алгоритмы сортировки. Для упрощения задачи будем сортировать последовательность по неубыванию.

### 2.1. Сортировка пузырьком

Осуществляется проход по массиву от начала до конца, в процессе меняя местами неотсортированные соседние элементы.

В результате первого прохода на последнем месте окажется максимальный элемент. Далее снова делается проход по неотсортированной части массива (от первого до предпоследнего) и так же меняются неупорядоченные соседние элементы. Таким образом, на предпоследнее место будет помещён второй по величине элемент.

Действия повторяются до тех пор, пока не обработается вся неотсортированная часть.

Схема алгоритма представлена на Рис.2.1.

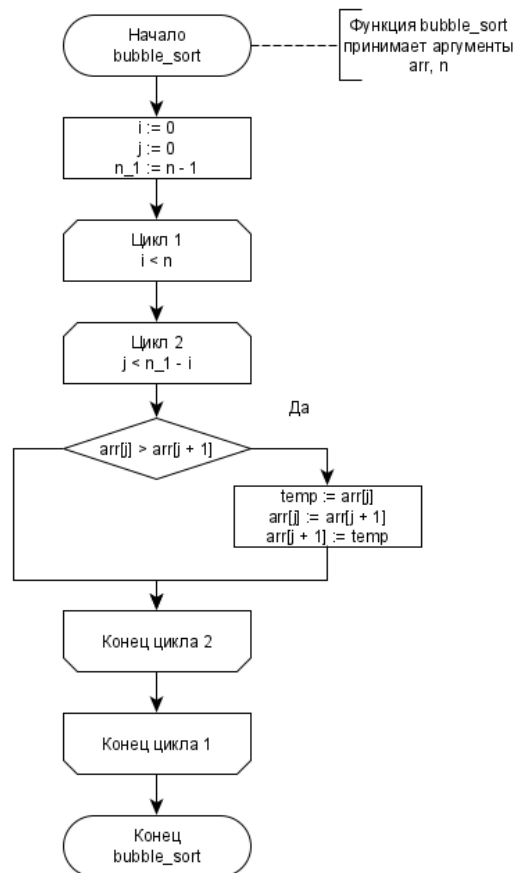


Рис. 2.1 — Сортировка пузырьком

## 2.2. Сортировка вставками

В этом алгоритме рассматриваемый массив условно делится на две части: отсортированная и нет.

В начале работы отсортированной частью считается нулевой элемент. Далее берётся каждый следующий и сравнивается с уже отсортированной частью. Находится подходящая для текущего элемента позиция в ней, осуществляется сдвиг уже отсортированных элементов, но больших по величине, чем рассматриваемый. И затем рассматриваемый элемент помещается на найденную позицию.

И так до тех пор, пока не просмотрится вся неотсортированная часть.

Схема алгоритма представлена на Рис.2.2.

## 2.3. Поразрядная сортировка

Производится сортировка массива целых положительных чисел, а также чисел, которые можно преобразовать в неотрицательные, путём увеличения всех элементов на величину, по модулю равную минимальному элементу.

Перед началом работы алгоритма находится максимальное количество разрядов  $k$  среди рассматриваемых чисел.

Далее сравнение производится поразрядно: сначала рассматриваются значения одного крайнего разряда, и формируются группы элементов по результатам этого сравнения, затем сравниваются значения следующего соседнего разряда, и происходит переупорядочивание элементов по результатам текущего сравнения (с сохранением относительного порядка, который был достигнут ранее). Сравнения продолжаются до тех пор, пока не обработается  $k$ ый разряд.

Схема алгоритма представлена на Рис.2.3 и 2.4.

## 2.4. Требования к ПО

Для корректной работы алгоритмов и проведения тестов необходимо выполнить следующее.

- Обеспечить возможность ввода массива через консоль и выбора алгоритма сортировки.

- В случае ввода некорректных данных вывести соответствующее сообщение. Программа не должна аварийно завершаться.
- Программа должна отсортировать массив и вывести результат на экран.
- Реализовать функцию замера процессорного времени, которое выбранный метод затрачивает на вычисление результатов. Для этого дать возможность ввода размера тестируемого массива. Вывести результаты замеров на экран.

## 2.5. Заготовки тестов

При проверке на корректность работы реализованных функций необходимо провести следующие тесты:

- отсортировать массив размером в один элемент;
- простой массив ненулевой длины;
- упорядоченный по невозрастанию массив;
- упорядоченный по неубыванию массив;
- массив, состоящий из одинаковых элементов.



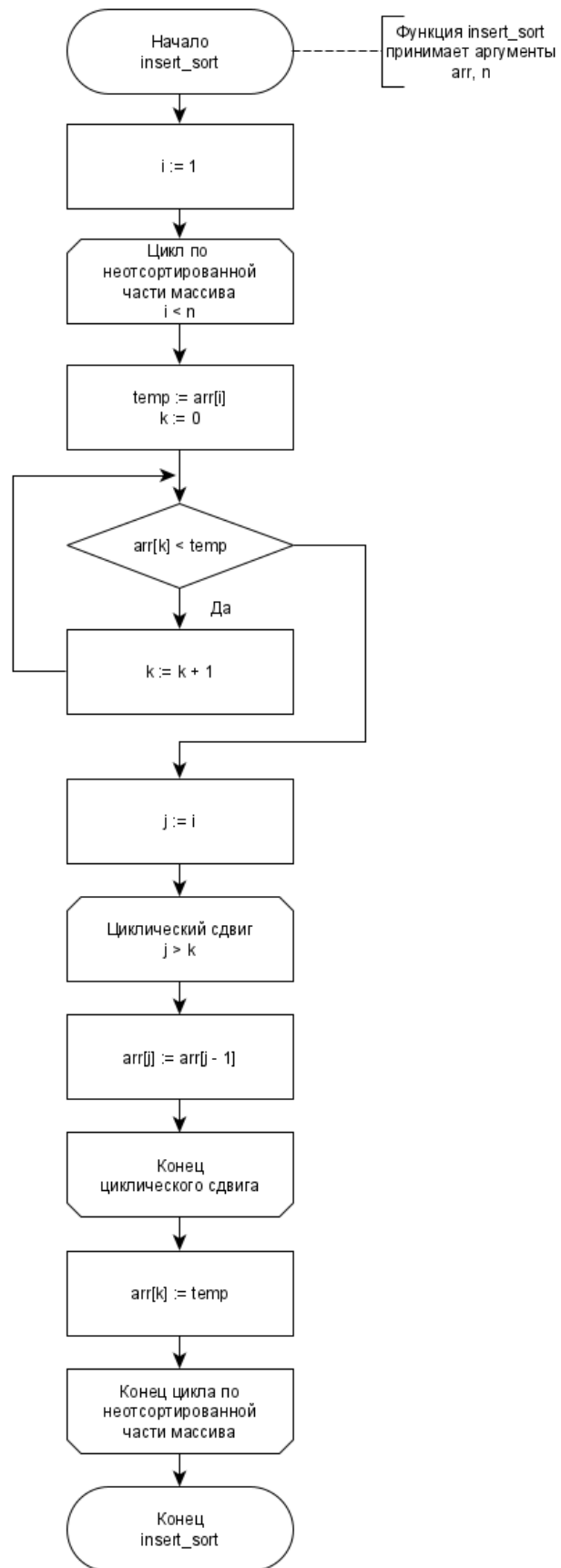


Рис. 2.2 — Сортировка вставками

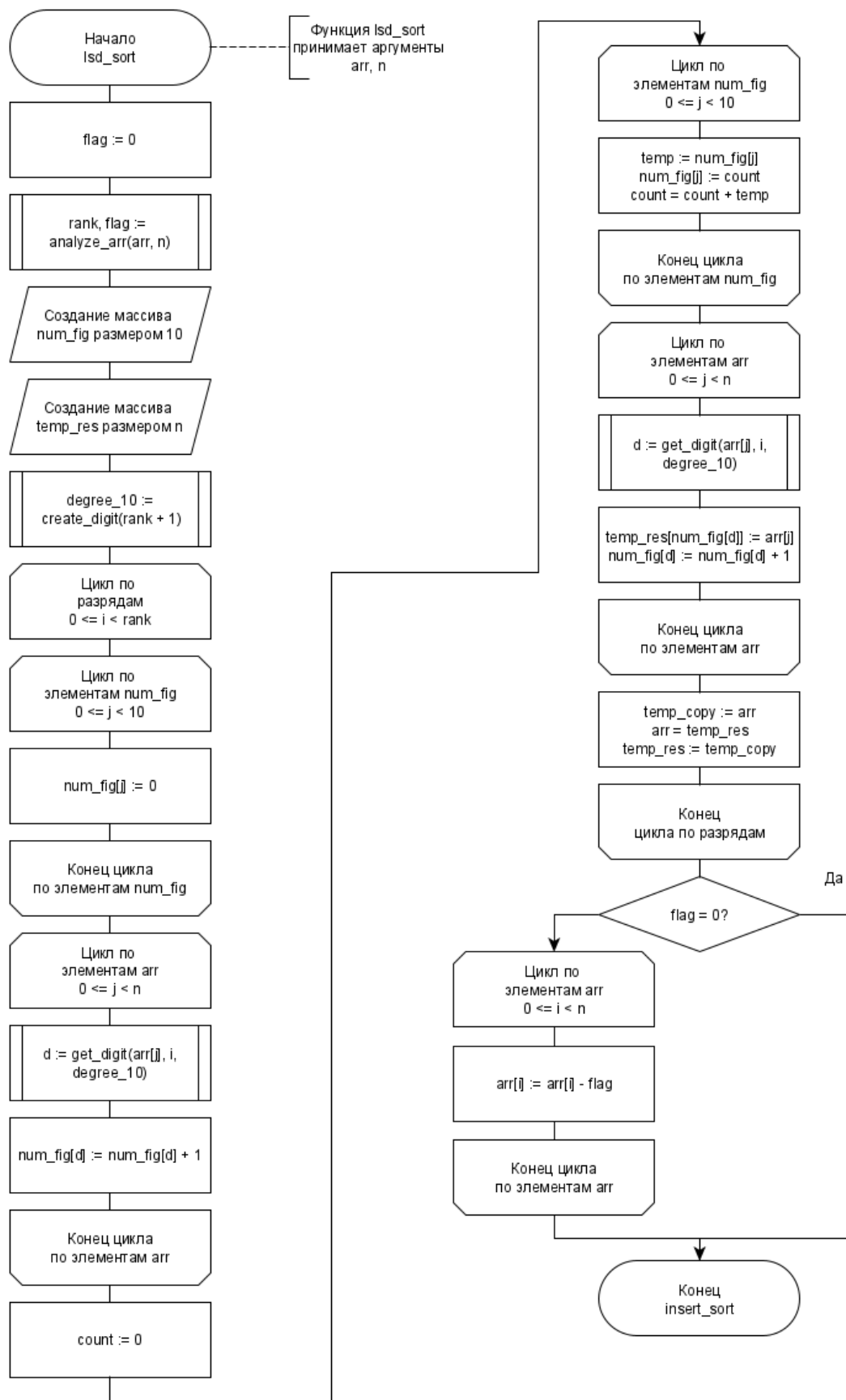


Рис. 2.3 — Поразрядная сортировка

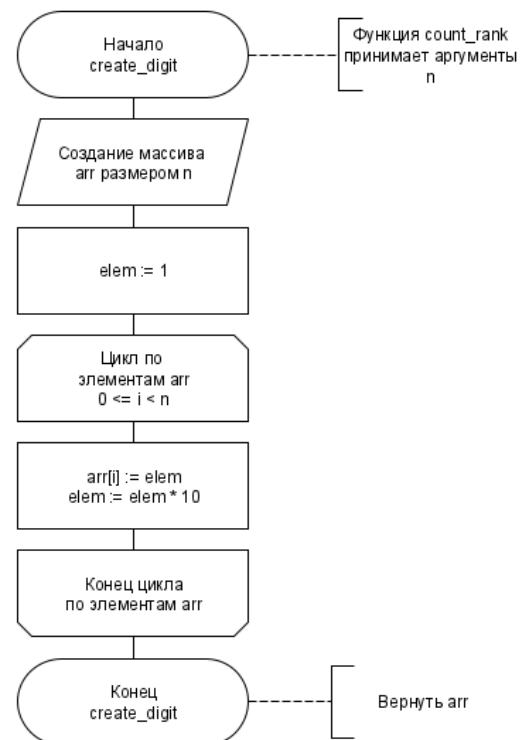
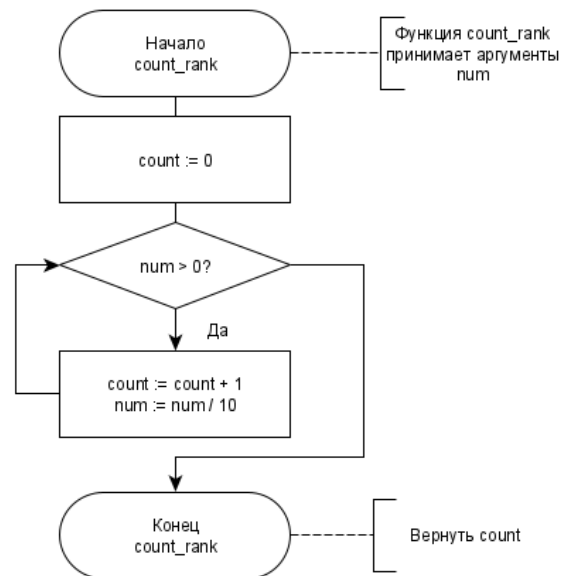
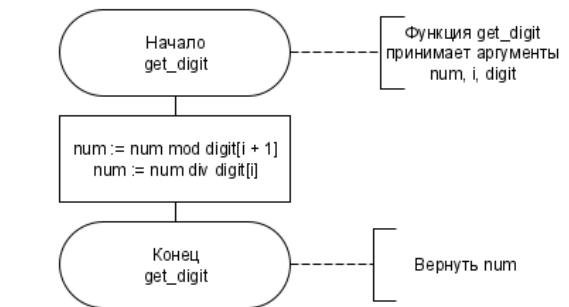
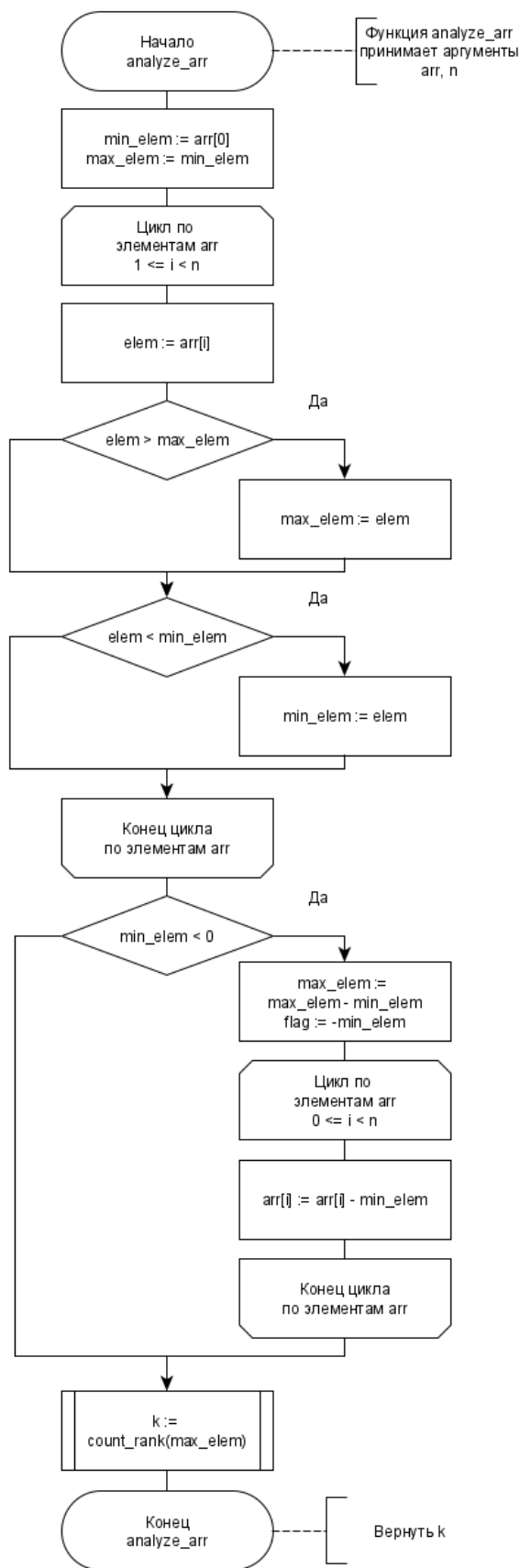


Рис. 2.4 — Поразрядная сортировка (продолжение)

## 3. Технологическая часть

### 3.1. Выбранный язык программирования

Для выполнения этой лабораторной работы был выбран язык программирования C++, так как есть большой навык работы с ним и с подключаемыми библиотеками, которые также использовались для проведения тестирования и замеров.

Использованная среда разработки - Visual Studio.

### 3.2. Листинг кода

Ниже представлены Листинги 3.1 - 3.3 функций, реализующих сортировку массива.

Листинг 3.1 — Сортировка пузырьком

```
1 #pragma optimize("", off)
2
3 void bubble_sort(array_t& arr, int n)
4 {
5     double temp;
6     int n_1 = n - 1;
7
8     for (int i = 0; i < n; i++)
9         for (int j = 0; j < n_1 - i; j++)
10             if (arr[j] > arr[j + 1])
11                 {
12                     temp = arr[j];
13                     arr[j] = arr[j + 1];
14                     arr[j + 1] = temp;
15                 }
16 }
17
18 #pragma optimize("", on)
```

Листинг 3.2 — Сортировка вставками

```
1 #pragma optimize("", off)
2
3 void insert_sort(array_t& arr, int n)
4 {
5     double temp;
```

```

6  int k;
7
8  for (int i = 1; i < n; i++)
9  {
10     temp = arr[i];
11     k = 0;
12
13     while (arr[k] < temp)
14         k += 1;
15
16     for (int j = i; j > k; j--)
17         arr[j] = arr[j - 1];
18
19     arr[k] = temp;
20 }
21 }
22 #pragma optimize("", on)

```

Листинг 3.3 — Поразрядная сортировка

```

1 #pragma optimize("", off)
2
3 int* create_digit(int n)
4 {
5     int* arr = create_array(n);
6     int elem = 1;
7
8     for (int i = 0; i < n; i++, elem *= 10)
9         arr[i] = elem;
10
11     return arr;
12 }
13
14 int count_rank(int num)
15 {
16     int count = 0;
17
18     while (num > 0)
19     {
20         count++;
21         num /= 10;

```

```

22     }
23
24     return count;
25 }
26
27 int analyze_arr(array_t arr, int n, int& flag)
28 {
29     int min_elem = arr[0], max_elem = min_elem, elem;
30
31     for (int i = 1; i < n; i++)
32     {
33         elem = arr[i];
34         if (elem > max_elem)
35             max_elem = elem;
36         if (elem < min_elem)
37             min_elem = elem;
38     }
39
40     if (min_elem < 0)
41     {
42         max_elem -= min_elem;
43         flag = -min_elem;
44
45         for (int i = 0; i < n; i++)
46             arr[i] -= min_elem;
47     }
48
49     return count_rank(max_elem);
50 }
51
52 int get_digit(int num, int i, int* digit)
53 {
54     num %= digit[i + 1];
55     num /= digit[i];
56
57     return num;
58 }
59
60 void lsd_sort(array_t& arr, int n)
61 {

```

```

62  int flag = 0, count, temp, d;
63  int rank = analyze_arr(arr, n, flag);
64  int* degree_10 = create_digit(rank + 1);
65  int num_fig[10];
66  array_t temp_res = create_array(n), temp_copy;
67
68  for (int i = 0; i < rank; i++)
69  {
70      for (int j = 0; j < 10; j++)
71          num_fig[j] = 0;
72
73      for (int j = 0; j < n; j++)
74          num_fig[get_digit(arr[j], i, degree_10)]++;
75
76      count = 0;
77
78      for (int j = 0; j < 10; j++)
79      {
80          temp = num_fig[j];
81          num_fig[j] = count;
82          count += temp;
83      }
84
85      for (int j = 0; j < n; j++)
86      {
87          d = get_digit(arr[j], i, degree_10);
88          temp_res[num_fig[d]] = arr[j];
89          num_fig[d]++;
90      }
91
92      temp_copy = arr;
93      arr = temp_res;
94      temp_res = temp_copy;
95  }
96
97  free_array(&degree_10);
98  free_array(&temp_res);
99
100  if (flag)
101      for (int i = 0; i < n; i++)

```

```

102     arr[i] == flag;
103 }
104
105 #pragma optimize("", on)

```

### 3.3. Результаты тестов

Для тестирования были написаны функции, проверяющие, согласно заготовкам выше, случаи. Выводы о корректности работы делаются на основе сравнения результатов.

**Все тесты пройдены успешно.** Сами тесты представлены ниже (Листинг 3.4).

Листинг 3.4 — Тесты

```

1 bool sort_cmp(array_t a, int n)
2 {
3     bool res = true;
4     void(*func_arr[])(array_t&, int) = { bubble_sort, insert_sort,
5         lsd_sort };
6
7     array_t c = copy_array(a, n);
8     sort(c, c + n);
9
10    for (int i = 0; i < 3 && res; i++)
11    {
12        array_t temp_arr = copy_array(a, n);
13
14        (*func_arr[i])(temp_arr, n);
15
16        res = cmp_array(c, temp_arr, n);
17
18        free_array(&temp_arr);
19    }
20
21    free_array(&c);
22
23    return res;
24 }
25 // Размер массива равен 1

```



```

26 void test_size_1()
27 {
28     int n = 1;
29
30     for (int i = 0; i < 3; i++)
31     {
32         array_t a = random_fill_array(n);
33
34         if (!sort_cmp(a, n))
35         {
36             cout << endl << __FUNCTION__ << " FAILED" << endl;
37             free_array(&a);
38             return;
39         }
40
41         free_array(&a);
42     }
43
44     cout << endl << __FUNCTION__ << " OK" << endl;
45 }
46
47 // Произвольные массивы различной длин
48 void test_std()
49 {
50     int n[] = { 3, 5, 8, 10, 12 };
51
52     for (int i = 0; i < sizeof(n)/sizeof(n[0]); i++)
53     {
54         array_t a = random_fill_array(n[i]);
55
56         if (!sort_cmp(a, n[i]))
57         {
58             cout << endl << __FUNCTION__ << " FAILED" << endl;
59             free_array(&a);
60             return;
61         }
62
63         free_array(&a);
64     }
65

```

```

66     cout << endl << __FUNCTION__ << " OK" << endl;
67 }
68
69 // Уже отсортированные по неубыванию массивы
70 void test_sorted()
71 {
72     int n = 30;
73
74     array_t a = random_fill_array(n);
75
76     sort(a, a + n);
77
78     if (!sort_cmp(a, n))
79     {
80         cout << endl << __FUNCTION__ << " FAILED" << endl;
81         free_array(&a);
82         return;
83     }
84
85     free_array(&a);
86
87     cout << endl << __FUNCTION__ << " OK" << endl;
88 }
89
90 // Уже отсортированные по невозрастанию массивы
91 void test_reverse_sorted()
92 {
93     int n = 30;
94
95     array_t a = random_fill_array(n);
96
97     sort(a, a + n);
98     reverse(a, a + n);
99
100    if (!sort_cmp(a, n))
101    {
102        cout << endl << __FUNCTION__ << " FAILED" << endl;
103        free_array(&a);
104        return;
105    }

```

```

106
107     free_array(&a);
108
109     cout << endl << __FUNCTION__ << " OK" << endl;
110 }
111
112 // Массив одинаковых элементов
113 void test_same_elements()
114 {
115     int n = 30;
116     array_t a = create_array(n);
117
118     for (int i = 0; i < n; i++)
119         a[i] = 102;
120
121     if (!sort_cmp(a, n))
122     {
123         cout << endl << __FUNCTION__ << " FAILED" << endl;
124         free_array(&a);
125         return;
126     }
127
128     free_array(&a);
129
130     cout << endl << __FUNCTION__ << " OK" << endl;
131 }
132
133 void run_tests()
134 {
135     test_size_1();
136     test_std();
137     test_sorted();
138     test_reverse_sorted();
139     test_same_elements();
140 }

```

### 3.4. Оценка трудоёмкости

Произведём оценку трудоёмкости приведённых алгоритмов. Рассмотрим сортировку массива  $A[N]$ .

### 3.5. Оценка времени

Процессорное время измеряется с помощью функции `QueryPerformanceCounter` библиотеки `windows.h` [3]. Осуществление замеров показано ниже (Листинг 3.5).

Листинг 3.5 — Замеры процессорного времени

1



## 4. Исследовательская часть

## Заключение

## Список литературы

1. Трудоёмкость алгоритмов и временные оценки [Электронный ресурс]. Режим доступа: <http://techn.sstu.ru/kafedri/подразделения/1/MetMat/shaturn/theoralg/5.htm>, свободный (дата обращения: 29.09.20).
2. Кормен, Томас Х. и др Алгоритмы: построение и анализ, 3-е изд. : Пер. с англ. - М. : ООО "И.Д. Вильямс 2018. - 1328 с. : ил. - Парал. тит. англ. - ISBN 978-5-8459-2016-4 (рус.).
3. QueryPerformanceCounter function [Электронный ресурс]. Режим доступа: <https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter>, свободный (дата обращения: 01.10.2020).