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 _____ Е.В. Брянская
(Группа) (Подпись, дата) (И.О. Фамилия)

Преподаватель _____ Л.Л. Волкова
(Подпись, дата) (И.О. Фамилия)

Москва, 2020

Оглавление

Введение **3**

1 Аналитическая часть **4**

1.1 Пузырьковая сортировка 4

1.2 Сортировка вставками 4

1.3 Поразрядная сортировка 5

2 Конструкторская часть **6**

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

3 Технологическая часть **12**

3.1 Выбранный язык программирования 12

3.2 Листинг кода 12

3.3 Результаты тестов 16

3.4 Оценка трудоёмкости 20

3.4.1 Сортировка пузырьком 20

3.4.2 Сортировка вставками 20

3.4.3 Поразрядная сортировка 20

3.5 Оценка времени 21

4 Исследовательская часть **24**

4.1 Характеристики ПО 24

4.2 Измерения 24

Заключение **26**

Список литературы **27**

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 перестеновке (переупорядочивании) входной последовательности из n чисел $a_1, a_2, ..., a_n$, так, чтобы выполнялось условие

$$a'_1 <= a'_2 <= ... <= a'_n \tag{1.1}$$

(в случае сортировки по неубыванию). Аналогичным образом осуществляется сортировка по невозрастанию [2].

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 отстортированной 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, сохраненяя 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.

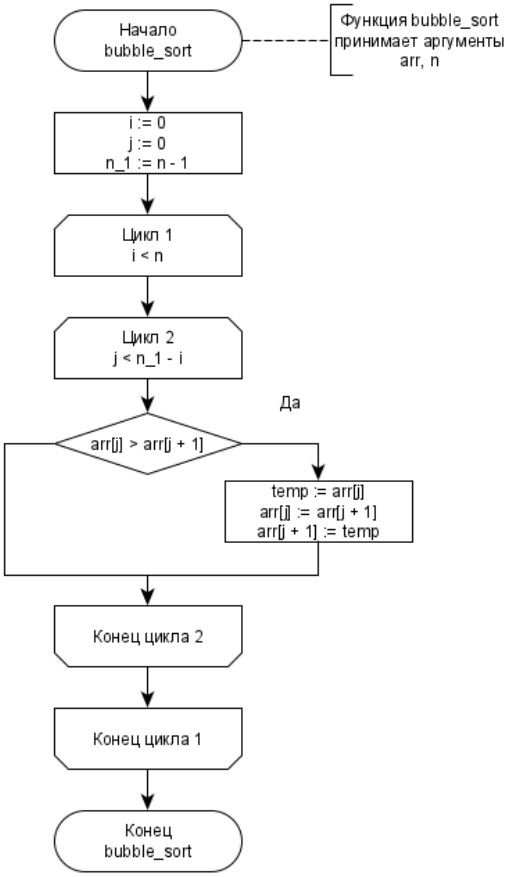

Рис. 2.1 — Сортировка пузырьком

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 kый 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. 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.

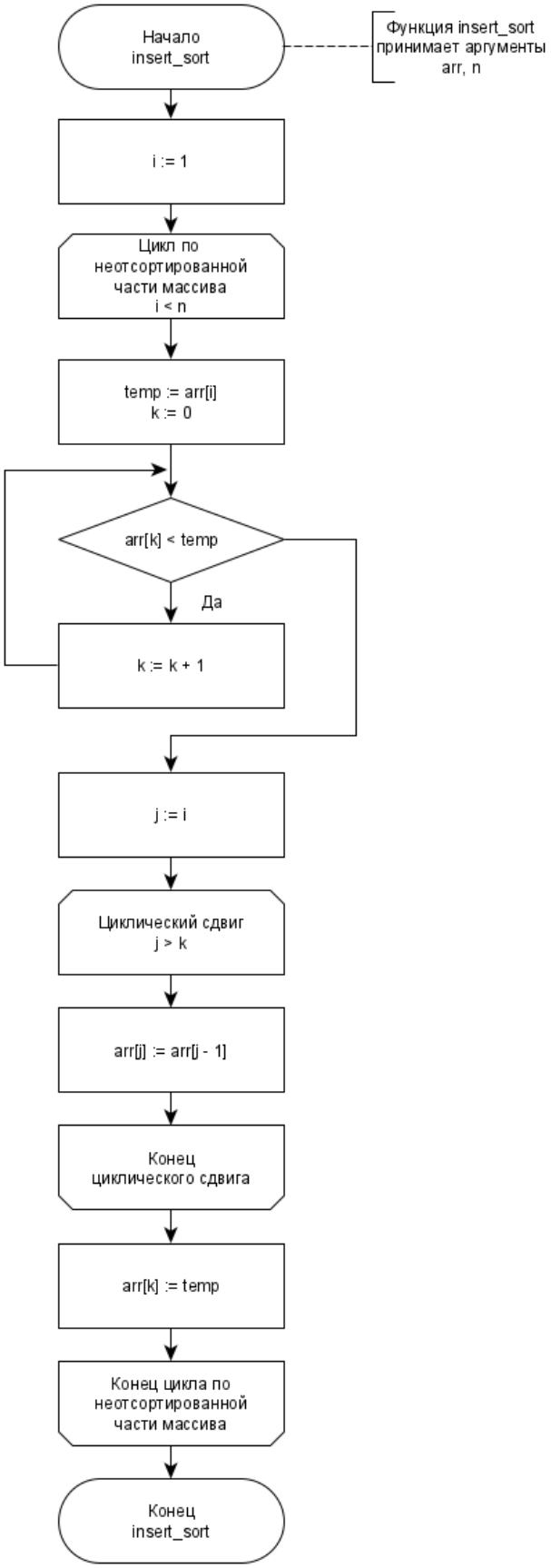

Рис. 2.2 — Сортировка вставками

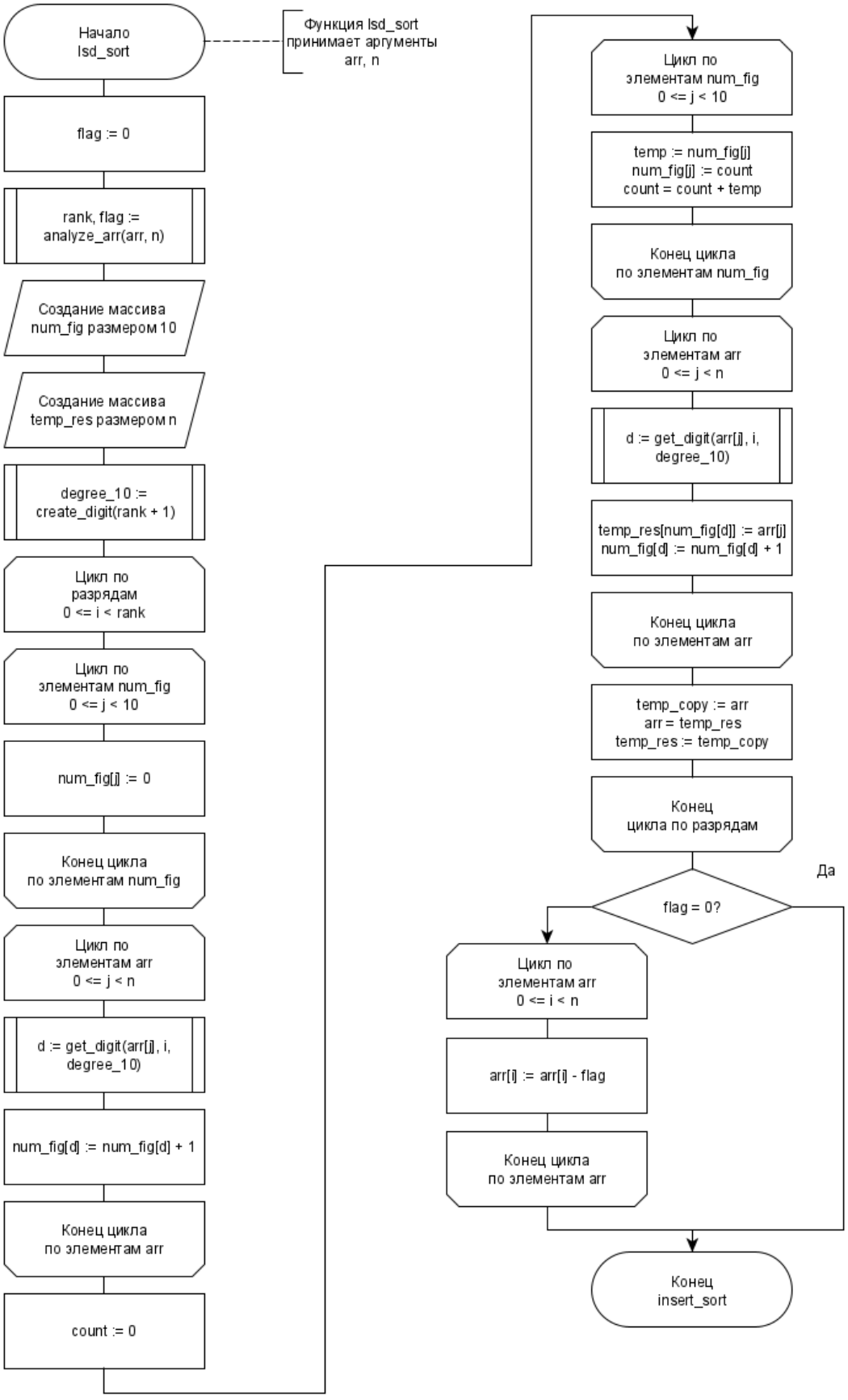

Рис. 2.3 — Поразрядная сортировка

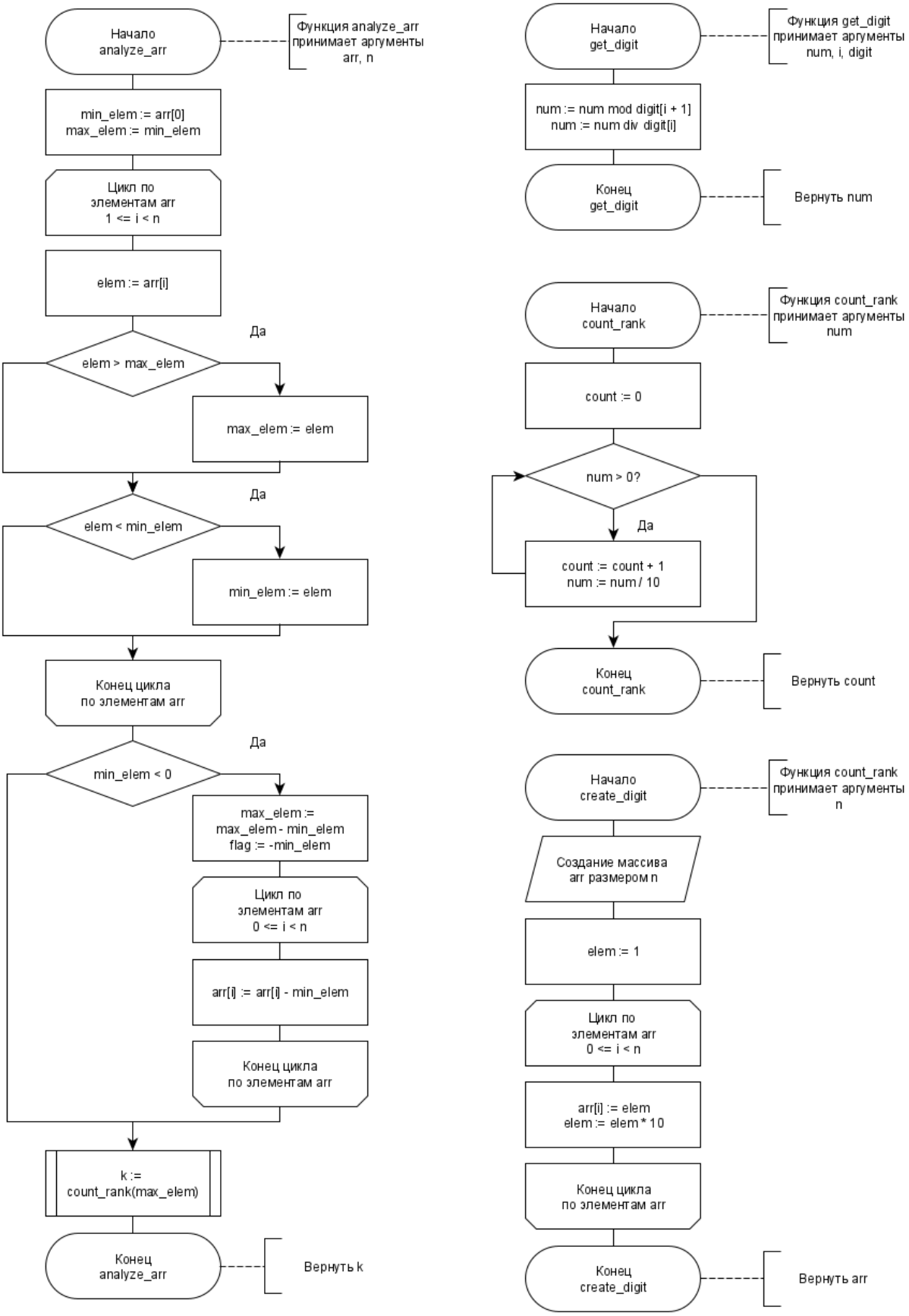

Рис. 2.4 — Поразрядная сортировка (продолжение)

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. Тестирование реализованных функций производилось путём сравнения полученного результата с результатом обработки массива методом из стандартной библиотеки algorithm [5].

Использованная среда разработки - Visual Studio [4].

3.2. Листинг кода

Ниже представлены Листиги 3.1 - 3.3 функций, реализующих сортировки массива.

Листинг 3.1 — Сортировка пузырьком

```
#pragma optimize("", off)

void bubble_sort(array_t& arr, int n)
{
  double temp;
  int n_1 = n -1;

  for (int i = 0; i < n; i++)
    for (int j = 0; j < n_1 - i; j++)
      if (arr[j] > arr[j + 1])
      {
        temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
}

#pragma optimize("", on)
```

Листинг 3.2 — Сортировка вставками

```
#pragma optimize("", off)

void insert_sort(array_t& arr, int n)
{
```

```
  double temp;
  int k;

  for (int i = 1; i < n; i++)
  {
    temp = arr[i];
    k = 0;

    while (arr[k] < temp)
      k += 1;

    for (int j = i; j > k; j--)
      arr[j] = arr[j - 1];

    arr[k] = temp;
  }
}
#pragma optimize("", on)
```

Листинг 3.3 — Поразрядная сортировка

```
#pragma optimize("", off)

int* create_digit(int n)
{
  int* arr = create_array(n);
  int elem = 1;

  for (int i = 0; i < n; i++, elem *= 10)
    arr[i] = elem;

  return arr;
}

int count_rank(int num)
{
  int count = 0;

  while (num > 0)
  {
    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,
       lsd_sort };
5
6    array_t c = copy_array(a, n);
7    sort(c, c + n);
8
9    for (int i = 0; i < 3 && res; i++)
10   {
11     array_t temp_arr = copy_array(a, n);
12
13     (*func_arr[i])(temp_arr, n);
14
15     res = cmp_array(c, temp_arr, n);
16
17     free_array(&temp_arr);
18   }
19
20   free_array(&c);
21
22   return res;
23 }
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 различный 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 $A[N]$, где N - размер массива. Пусть R - максимальное количество разрядов числа, которое есть в этом массиве.

3.4.1 Сортировка пузырьком

$$f_{bubble} = 2 + 2 + N(2 + 3 + \frac{N-1}{2}(3 + 4 + \begin{bmatrix} 0, & \text{л.с.} \\ 2 + 4 + 3, & \text{х.с.} \end{bmatrix}))$$

$$f_{bubble} = 4 + N(5 + \frac{N-1}{2}(7 + \begin{bmatrix} 0, & \text{л.с.} \\ 9, & \text{х.с.} \end{bmatrix}))$$

Лучший случай (отсортированный массив):

$f_{bubble} = 4 + N(5 + 3.5(N-1)) = 3.5N^2 + 1.5N + 4$

Худший случай (отсортированный в обратном порядке массив):

$f_{bubble} = 4 + N(5 + 8(N-1)) = 8N^2 - 3N + 4$

3.4.2 Сортировка вставками

$$f_{insert} = 2 + N(2 + 2 + 1 + \begin{bmatrix} 2, & \text{обратно отсортированный массив} \\ 2 + (\frac{N+1}{2})3, & \text{отсортированный массив} \end{bmatrix} + 2 +$$
$$+ \begin{bmatrix} \frac{N+1}{2} + 6, & \text{обратно отсортированный массив} \\ 0, & \text{отсортированный массив} \end{bmatrix} + 2)$$

Первый случай:

$f_{insert} = 2 + N(17 + \frac{N+1}{2}) = 0.5N^2 + 17.5N + 2$

Второй случай:

$f_{insert} = 2 + N(11 + \frac{N+1}{2}3) = 1.5N^2 + 12.5N + 2$

3.4.3 Поразрядная сортировка

Функция получения значения конкретного разряда:

$f_{digit} = 3 + 2 = 5$

Функция создания массива степеней 10:

$f_{create} = 2 + 2 + (R + 1)(3 + 2) = 4 + 5(R + 1) = 9 + 5R$

Функция посчёта количества разрядов числа:

$f_{count} = 1 + 1 + R(1 + 1) = 2 + 2R$

Функция, нормализующая массив:

$$f_{analyze} = 2{+}1{+}2{+}N(2{+}2{+}1{+}\begin{bmatrix} 0, & \text{л.с.} \\ 1, & \text{х.с.} \end{bmatrix} {+}1{+}\begin{bmatrix} 0, & \text{л.с.} \\ 1, & \text{х.с.} \end{bmatrix}){+}1{+}\begin{bmatrix} 0, & \text{л.с.} \\ 1 + 1 + 2 + N(2 + 2), & \text{х.с.} \end{bmatrix} +$$

$$+f_{count}$$

$$f_{analyze} = 8 + 2R + N(6{+}\begin{bmatrix} 0, & \text{л.с.} \\ 1, & \text{х.с.} \end{bmatrix} {+}\begin{bmatrix} 0, & \text{л.с.} \\ 1, & \text{х.с.} \end{bmatrix}){+}\begin{bmatrix} 0, & \text{л.с.} \\ 4 + 4N, & \text{х.с.} \end{bmatrix}$$

Функция, реализующая алгоритм поразрядной сортировки:

$$f_{lsd} = 1 + 1 + f_{analyze} + 1 + 1 + f_{create} + 2 + R(2 + 2 + 10(2 + 2) + 2 + N(2 + 3 + f_{digit}) + 1 +$$

$$2 + 10(2 + 5) + 2 + N(2 + 2 + f_{digit} + 4 + 2) + 3) + 1{+}\begin{bmatrix} 0, & \text{л.с.} \\ 2 + N(2 + 2), & \text{х.с.} \end{bmatrix}$$

$$f_{lsd} = 24{+}131R{+}25RN{+}\begin{bmatrix} 0, & \text{л.с.} \\ 2 + 4N, & \text{х.с.} \end{bmatrix} {+}N(6{+}\begin{bmatrix} 0, & \text{л.с.} \\ 1, & \text{х.с.} \end{bmatrix} {+}\begin{bmatrix} 0, & \text{л.с.} \\ 1, & \text{х.с.} \end{bmatrix}){+}\begin{bmatrix} 0, & \text{л.с.} \\ 4 + 4N, & \text{х.с.} \end{bmatrix}$$

Лучший случай (массив состоит из неотрицательных элементов):

$f_{lsd} = 24 + 131R + 25RN + 6N$

Худший случай (в массиве есть отрицательные элементы):

$f_{lsd} = 24 + 131R + 25RN + 2 + 4N + N(6 + 1 + 1) + 4 + 4N = 30 + 131R + 25RN + 16N$

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

```
void test_range(vector<int>& n)
{
  for (int key : n)
  {
    cout << endl << endl << "Размер тестируемого массива: " << key << endl;

    cout << endl << "Сортировка пузырьком" << endl;
```

```
8       test_time(bubble_sort, key);
9       cout << endl << "Сортировка вставками" << endl;
10      test_time(insert_sort, key);
11      cout << endl << "Поразрядная сортировка" << endl;
12      test_time(lsd_sort, key);
13    }
14  }
15
16  void start_measuring()
17  {
18    LARGE_INTEGER li;
19    QueryPerformanceFrequency(&li);
20
21    PCFreq = double(li.QuadPart) / 1000;
22
23    QueryPerformanceCounter(&li);
24    CounterStart = li.QuadPart;
25  }
26
27  double get_measured()
28  {
29    LARGE_INTEGER li;
30    QueryPerformanceCounter(&li);
31
32    return double(li.QuadPart - CounterStart) / PCFreq;
33  }
34
35
36  void test_time(void(*f)(array_t&, int), int n)
37  {
38    array_t a = random_fill_array(n);
39
40    int num = 0;
41    start_measuring();
42
43    while (get_measured() < 3 * 1000)
44    {
45      f(a, n);
46      num++;
47    }
```

```
48
49    double t = get_measured() / 1000;
50    cout << "Выполнено " << num << " операций за " << t << " секунд" << endl;
51    cout << "Время: " << t / num << endl;
52
53    free_array(&a);
54 }
```

4. Исследовательская часть

4.1. Характеристики ПО

При проведении замеров времени использовался компьютер, имеющий следующие характеристики:

- ОС - Windows 10 Pro
- Процессор - Inter Core i7 10510U (1800 МГц)
- Объём ОЗУ - 16 Гб

4.2. Измерения

Для проведения замеров процессорного времени использовались массивы длин N. $N \in \{10, 50, 100, 500, 1000, 3000, 7000, 10000, 50000, 100000\}$. Их содержимое генерируется случайным образом.

Каждый замер проводится 5 раз для получения более точного среднего результата.

В таблице 4.1 и таблице 4.2 представлены результаты замеров процессорного времени работы реализаций алгоритмов (в сек).

Таблица 4.1 — Результаты измерений на размерах до 100 элементов

Размер n / Алгоритм	10	50	100
Сортировка пузырьком	$1.762 \cdot 10^{-7}$	$2.38 \cdot 10^{-6}$	$9.066 \cdot 10^{-6}$
Сортировка вставками	$1.56 \cdot 10^{-7}$	$1.687 \cdot 10^{-6}$	$6.387 \cdot 10^{-6}$
Поразрядная сортировка	$1.875 \cdot 10^{-6}$	$6.883 \cdot 10^{-6}$	$1.328 \cdot 10^{-5}$

Таблица 4.2 — Результаты измерений на размерах до 100 000 элементов

Размер n / Алгоритм	500	1 000	3 000	7 000	10 000	50 000	100 000
Сортировка пузырьком	$2.58 \cdot 10^{-4}$	$8.53 \cdot 10^{-4}$	0.0077	0.043	0.089	5.924	23.705
Сортировка вставками	$1.66 \cdot 10^{-4}$	$6.07 \cdot 10^{-4}$	0.0054	0.0295	0.060	1.68	7.417
Поразрядная сортировка	$6.452 \cdot 10^{-5}$	$1.28 \cdot 10^{-4}$	$3.8 \cdot 10^{-4}$	$8.9 \cdot 10^{-4}$	0.0013	0.0064	0.0127

Согласно полученным результатам можно сделать следующие **выводы**:

- на массивах малого размера (до 100 элементов) поразрядная сортировка показывает результаты по времени примерно на порядок хуже, чем сортировки пузырьком и вставками;
- что касается алгоритма сортировки вставками на таких массивах, то среди рассматриваемых трёх алгоритмов у него лучшие показатели по времени;
- на массивах большего размера поразрядная сортировка эффективнее, по сравнению с двумя другими алгоритмами, на несколько порядков;
- на таких массивах хуже всего показатели по времени у сортировки пузырьком;
- сортировка вставками обрабатывает подобные массивы лучше, чем сортировка пузырьком, но значительно уступает по времени поразрядному алгоритму.

Заключение

В ходе лабораторной работы была достигнута поставленная цель, а именно, оценена трудоёмкость трёх алгоритмов сортировки массивов (сортировка Пузырьком, Вставками и Поразрядная сортировка).

В процессе выполнения были решены все задачи. Описаны все рассматриваемые алгоритмы, дана теоретическая оценка трудоёмкости каждого. Все проработанные алгоритмы реализованы, кроме того, были проведены замеры процессорного времени работы на материале серии экспериментов и проведён сравнительный анализ, сделаны выводы.

По результатам замеров процессорного времени сделаны следующие заключения.

- Применение поразрядной сортировки эффективно на массивах большого размера (от 100 элементов), в то время, как на меньших размерах этот алгоритм требует значительных временных затрат по сравнению со сравниваемыми алгоритмами.
- Чем больше размер массива, тем лучше показатели по времени демонстрирует поразрядная сортировка по сравнению с двумя другими алгоритмами.
- Сортировка пузырьком, как и ожидалось, уступает сортировке вставками на любых размерностях исследуемых массивов.
- Среди трёх рассматриваемых алгоритмов сортировка вставками имеет лучшие временные показатели на массивах до 100 элементов, также может быть применима и на массивах, большего размера.

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: 01.10.20).

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: 03.10.2020).

4. Документация по Visual Studio 2019 [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/visualstudio/windows/?view=vs-2019, свободный (дата обращения: 05.10.2020)

5. Документация по Стандартной библиотекн языка C++ algorithm [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/cpp/standard-library/algorithm?view=vs-2019, свободный (дата обращения 12.10.2020)