

*Государственное образовательное учреждение высшего
профессионального образования*

**«Московский государственный технический
университет имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)**

ЛАБОРАТОРНАЯ РАБОТА №2
ПО КУРСУ «АНАЛИЗ АЛГОРИТМОВ»

Сортировка массивов

Выполнил: Сорокин А.П., гр. ИУ7-52Б

Преподаватели: Волкова Л.Л., Строганов Ю.В.

Москва, 2019 г.

Оглавление

Введение	2
1 Аналитическая часть	3
1.1 Задачи	3
1.2 Описание алгоритмов	3
1.2.1 Сортировка вставками	3
1.2.2 Сортировка слиянием	3
1.2.3 Быстрая сортировка	3
1.2.4 Модель вычислений	3
2 Конструкторская часть	4
2.1 Схемы алгоритмов	4
2.2 Оценка трудоёмкости	7
2.2.1 Сортировка вставками	7
2.2.2 Сортировка слиянием	7
2.2.3 Быстрая сортировка	7
2.3 Замер используемой памяти	7
3 Технологическая часть	8
3.1 Требования к программному обеспечению	8
3.2 Средства реализации	8
3.3 Реализации алгоритмов	8
3.4 Тесты	10
4 Экспериментальная часть	11
4.1 Примеры работы	11
4.2 Сравнение времени работы алгоритмов	11
Заключение	13
Литература	14

Введение

...

1. Аналитическая часть

1.1 Задачи

Цель лабораторной работы - изучение трех алгоритмов сортировки массивов. Были выбраны следующие алгоритмы:

- сортировка вставками;
- сортировка слиянием;
- быстрая сортировка.

Для того чтобы добиться поставленной цели, были поставлены следующие задачи:

- изучить и реализовать алгоритмы сортировок массивов;
- оценить трудоёмкости алгоритмов;
- выполнить сравнительный анализ алгоритмов на основе трудоёмкости и используемой памяти;
- оценить и сравнить эффективности алгоритмов по времени.

1.2 Описание алгоритмов

1.2.1 Сортировка вставками

1.2.2 Сортировка слиянием

1.2.3 Быстрая сортировка

1.2.4 Модель вычислений

В рамках данной работы используется следующая модель вычислений:

- операции, имеющие трудоемкость 1: $<$, $>$, $=$, $<=$, $=>$, $==$, $!=$, $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$, $[]$;
- оператор условного перехода имеет трудоёмкость, равную трудоёмкости операторов тела условия;
- оператор цикла `for` имеет трудоемкость:

$$F_{for} = F_{init} + F_{check} + N * (F_{body} + F_{inc} + F_{check}), \quad (1.1)$$

где F_{init} – трудоёмкость инициализации, F_{check} – трудоёмкость проверки условия, F_{inc} – трудоёмкость инкремента аргумента, F_{body} – трудоёмкость операций в теле цикла, N – число повторений. [1]

2. Конструкторская часть

2.1 Схемы алгоритмов

На рисунках 2.1, 2.2, 2.3 представлены схемы алгоритмов трёх алгоритмов сортировки массивов. Сортировка выполняется по возрастанию.

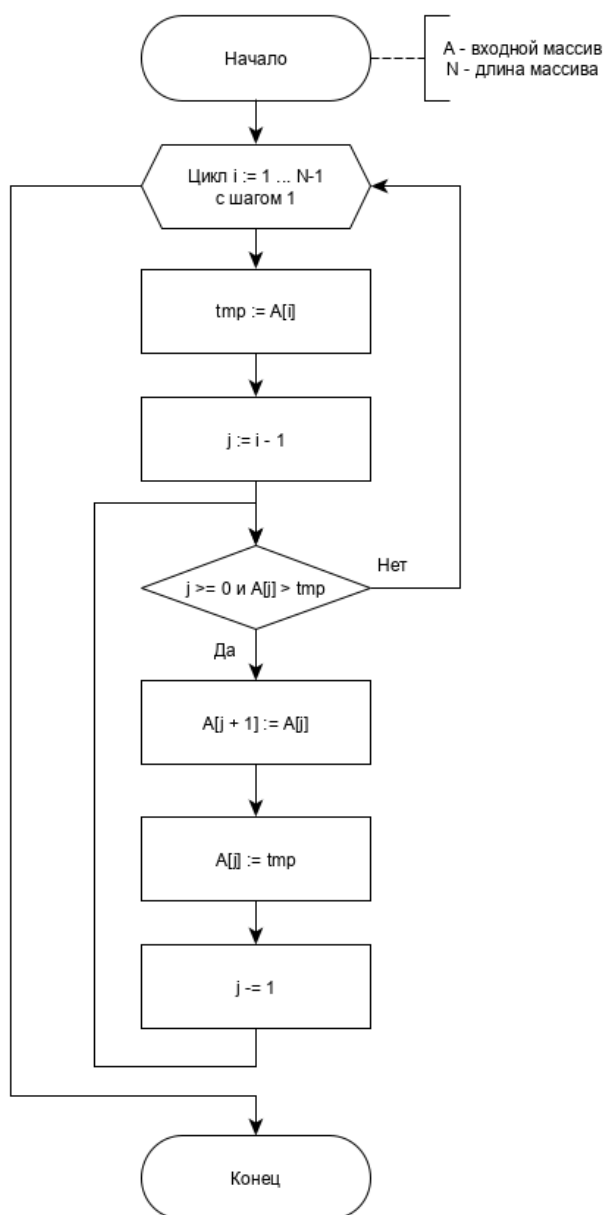


Рис. 2.1: Сортировка вставками

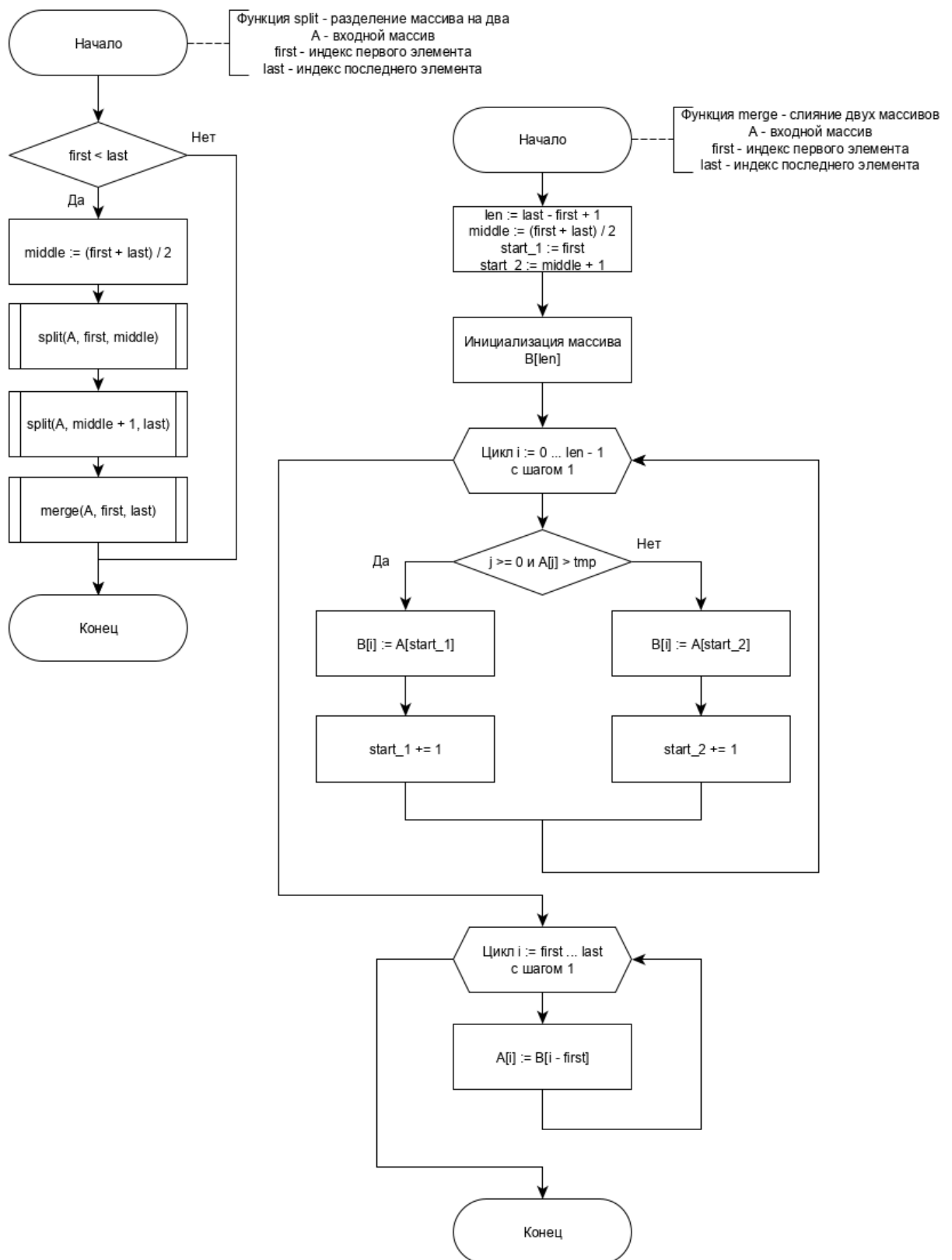


Рис. 2.2: Сортировка слиянием

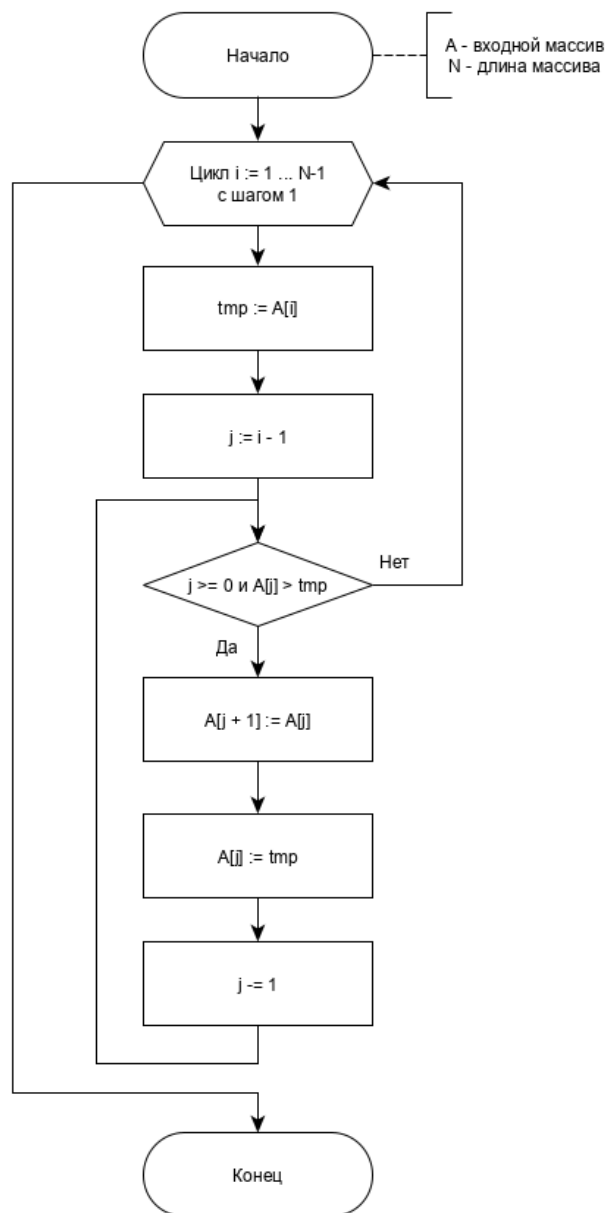


Рис. 2.3: Быстрая сортировка

2.2 Оценка трудоёмкости

Пусть дан массив A длиной N . Рассмотрим трудоёмкость трёх алгоритмов сортировки.

2.2.1 Сортировка вставками

2.2.2 Сортировка слиянием

2.2.3 Быстрая сортировка

2.3 Замер используемой памяти

Пусть дан массив A длиной N и для хранения целого числа требуется 4 байта памяти. В каждом из алгоритмов требуется хранить исходный массив A . Таким образом, под хранение массива требуется $4N$ байт памяти.

3. Технологическая часть

3.1 Требования к программному обеспечению

На вход подаются размер массива и сам массив. На выход программа выдаёт три массива, которые являются результатами работы трёх различных алгоритмов сортировки. Сортировка выполняется по возрастанию.

3.2 Средства реализации

Для реализации программы был использован язык C++ [2]. Для замера процессорного времени была использована функция `rdtsc()` из библиотеки `stdrin.h`.

3.3 Реализации алгоритмов

В листингах 3.1, 3.2, 3.3 представлены коды реализации алгоритмов сортировки массивов.

Листинг 3.1: Сортировка вставками

```
1 void array_sort_insert(int * const arr, size_t n)
2 {
3     for (size_t i = 1; i < n; i++)
4     {
5         int tmp = arr[i];
6         int j = i - 1;
7         while (j >= 0 && arr[j] > tmp)
8         {
9             arr[j + 1] = arr[j];
10            arr[j] = tmp;
11            j--;
12        }
13    }
14 }
```

Листинг 3.2: Сортировка слиянием

```
1 void array_merge(int * const arr, size_t first, size_t last)
2 {
3     size_t len = last - first + 1;
4     size_t middle = (first + last) / 2;
5     size_t start_1 = first;
6     size_t start_2 = middle + 1;
7
8     int *arr_tmp = new int[len];
9 }
```

```

10  for (size_t i = 0; i < len; i++)
11      if ((start_1 <= middle) && ((start_2 > last) ||
12          (arr[start_1] < arr[start_2])))
13      {
14          arr_tmp[i] = arr[start_1];
15          start_1++;
16      }
17      else
18      {
19          arr_tmp[i] = arr[start_2];
20          start_2++;
21      }
22
23  for (size_t i = first; i <= last; i++)
24      arr[i] = arr_tmp[i - first];
25  delete [] arr_tmp;
26 }
27
28 void array_sort_merge(int * const arr, size_t first, size_t last)
29 {
30     if (first < last)
31     {
32         size_t middle = (first + last) / 2;
33         array_sort_merge(arr, first, middle);
34         array_sort_merge(arr, middle + 1, last);
35         array_merge(arr, first, last);
36     }
37 }

```

Листинг 3.3: Быстрая сортировка

```

1  void array_sort_quick(int * const arr, size_t first, size_t last)
2  {
3      size_t l_hold = first;
4      size_t r_hold = last;
5      int pivot = arr[first];
6
7      while (first < last)
8      {
9          while ((arr[last] >= pivot) && (first < last))
10             last--;
11             if (first != last)
12             {
13                 arr[first] = arr[last];
14                 first++;
15             }
16             while ((arr[first] <= pivot) && (first < last))
17                 first++;
18             if (first != last)
19             {
20                 arr[last] = arr[first];
21                 last--;
22             }
23     }

```

```

24
25     size_t middle = first;
26     arr[middle] = pivot;
27     if (l_hold < middle)
28         array_sort_quick(arr, l_hold, middle - 1);
29     if (r_hold > middle)
30         array_sort_quick(arr, middle + 1, r_hold);
31 }

```

3.4 Тесты

Для проверки корректности работы были подготовлены функциональные тесты, представленные в таблице 3.1. Все алгоритмы должны выдавать на выходе одинаковые результаты. Сортировка выполняется по возрастанию.

Таблица 3.1: Функциональные тесты

Массив											Ожидание										
[1]											[1]										
[0	1	2	3	4	5	6	7	8	9]	[0	1	2	3	4	5	6	7	8	9]		
[9	8	7	6	5	4	3	2	1	0]	[0	1	2	3	4	5	6	7	8	9]		
[1	1	1	1	1	1	1	1	1	1]	[1	1	1	1	1	1	1	1	1	1]		
					5	4	0	2	-1	4]						-1	0	2	4	4	5]
					5	4	0	2	-1	3]						-1	0	2	3	4	5]

В результате проверки реализации всех алгоритмов сортировки прошли все поставленные функциональные тесты.

4. Экспериментальная часть

4.1 Примеры работы

На рисунке 4.1 представлен пример работы программы, демонстрирующий корректную работу алгоритмов.

Рис. 4.1: Пример работы программы

4.2 Сравнение времени работы алгоритмов

Для сравнения времени работы алгоритмов сортировки массивов были использованы массивы длиной от 100 до 1000 с шагом 50. Эксперимент для более точного результата повторялся 1000 раз. Итоговый результат рассчитывался как средний из полученных результатов. Результаты измерений показаны в таблице 4.1 и на рисунке 4.2.

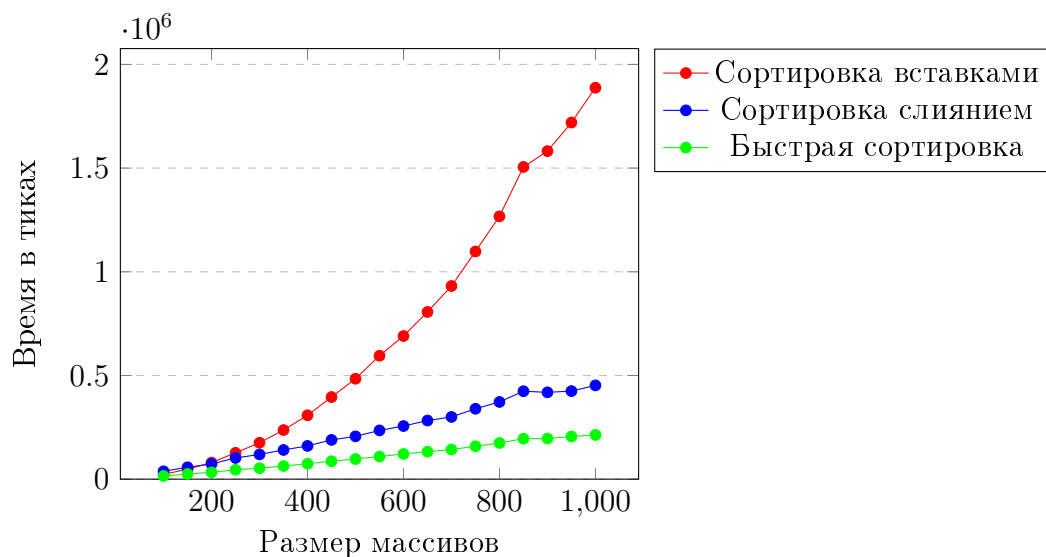


Рис. 4.2: График времени работы алгоритмов сортировки массивов

Таблица 4.1: Время работы алгоритмов сортировки массивов в тактах процессора

Размер массива	Сортировка вставками	Сортировка слиянием	Быстрая сортировка
100	23020	37936	15425
150	48231	57332	24830
200	79756	73360	32727
250	126912	102182	45246
300	175714	119256	53000
350	236744	140888	63348
400	307970	160337	74571
450	396018	189314	86469
500	484205	206362	97275
550	595318	235035	108988
600	690243	256135	121582
650	806436	282743	132505
700	931462	300124	143085
750	1098020	339428	158600
800	1267320	372270	174627
850	1505559	424077	195101
900	1581934	418612	195729
950	1719698	424659	205611
1000	1887598	452219	213460

Заключение

...

Литература

- [1] Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р.М. Ривест: – МЦНТО, 1999.
- [2] <https://cppreference.com/> [Электронный ресурс]