МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического занятия 4

Тема: Определение эффективного алгоритма сортировки

Дисциплина: Структуры и алгоритмы обработки данных

Выполнил студент Антонов А.Д.

Группа ИКБО-01-20

Москва 2021

Содержание

Задание 1. Определение эффективного алгоритма в среднем случае

Вариант 2.

1.1. Сортировка простого обмена с условием Айверсона

Постановка задачи

Разработать алгоритм сортировки простого обмена (пузырька) с условием Айверсона, провести анализ вычислительной и емкостной сложности алгоритма на массивах, заполненных случайно.

Описание алгоритма сортировки

При переборе массива попарно сравниваются соседние элементы. Если порядок их следования не соответствует заданному критерию упорядоченности, то элементы меняются местами. В результате одного такого просмотра при сортировке по возрастанию элемент с самым большим значением ключа переместится («всплывет») на последнее место массива. При следующем проходена свое место «всплывет» второй по величине элемент и т.д. Отсутствие перестановок на какой-либо итерации означает упорядоченность массива (условие Айверсона).

Алгоритм сортировки

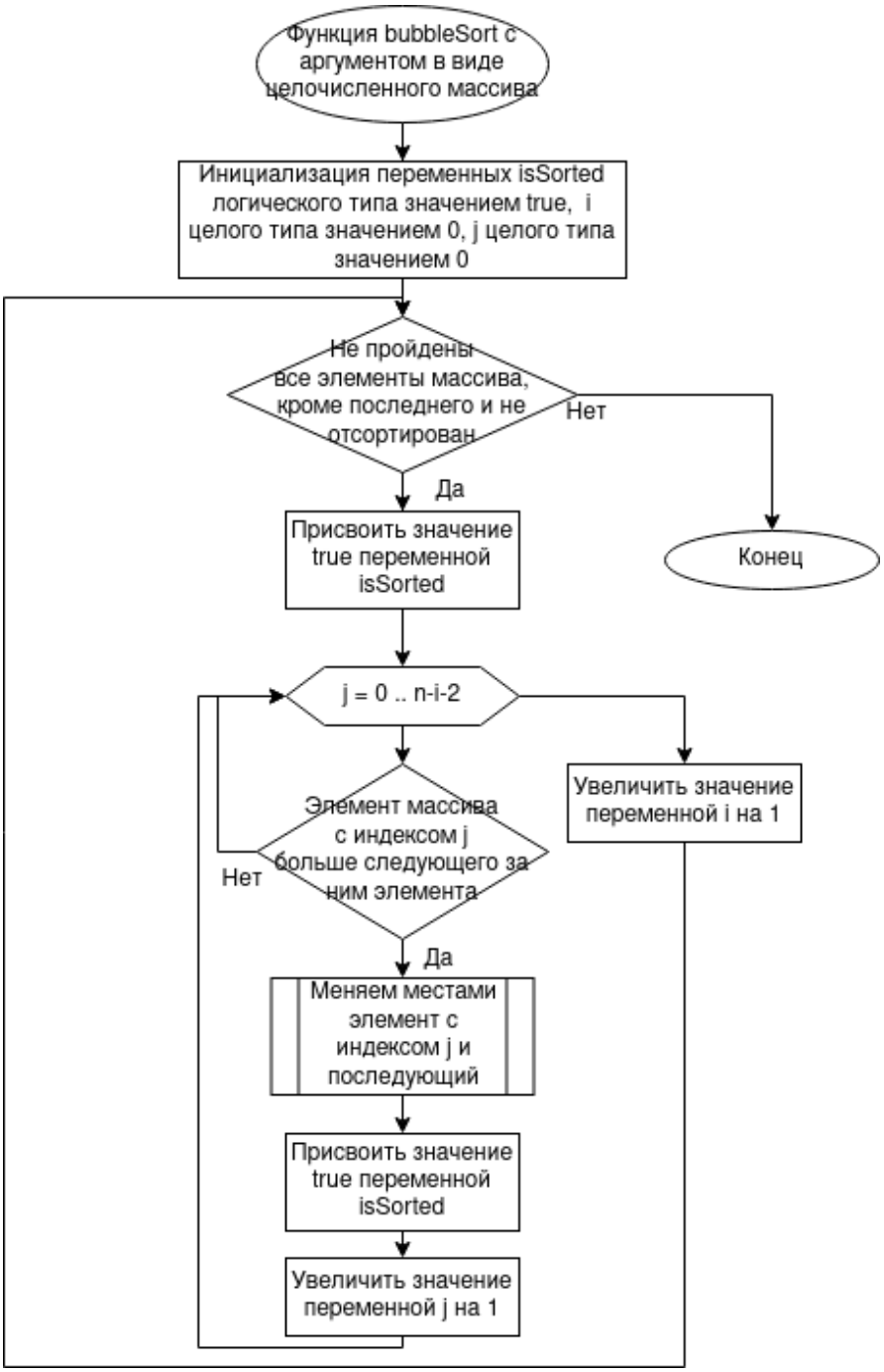


Рис. 1 - Блок-схема сортировки пузырьком с условием Айверсона

Оценка функции роста скорости выполнения алгоритма сортировки

Определим теоретическую сложность алгоритма при помощи таблицы операторов.

Таблица 1 - Подсчет количества операторов в алгоритме сортировки простого обмена

Номер оператора	Оператор	Время выполнения одного оператора	Кол-во выполнений оператора в строке
1	bool isSorted = false;	C1	1 раз
2	for (int i = 0; !isSorted && i < n-1; ++i) {	C2	n раз
3	isSorted = true;	C3	n-1 раз
4	for (int j = 0; j < n - i - 1; ++j) {	C4	n(n-1) раз
5	if (v[j] > v[j+1]) {	C5	n(n-1) - 1 раз
6	std::swap(v[j], v[j+1]);	C6	n(n-1) - 1 раз
7	isSorted = false;}}}	C7	n(n-1) - 1 раз

Из таблицы 1 получим функцию роста выполнения алгоритма сортировки простого обмена. Пусть $T(n)$ - время выполнения алгоритма, зависящее от n. Тогда

$$T(n) = C_1 + C_2 \cdot n + C_3 \cdot (n-1) + C_4 \cdot (n^2 - n) + C_5 \cdot (n^2 - n - 1) + C_6 \cdot (n^2 - n - 1) + C_7 \cdot (n^2 - n - 1)$$

После упрощения получаем

$$T(n) = C_1 + C_2 \cdot n + C_3 \cdot n - C_3 + C_4 \cdot n^2 - C_4 \cdot n + C_5 \cdot n^2 - C_5 \cdot n - C_5 + C_6 \cdot n^2 - C_6 \cdot n - C_6. + C_7 \cdot n^2 - C_7 \cdot n - C_7$$

Подведя подобные, получаем

$$T(n) = An^2 + Bn + C.$$

Оставляя справа только доминирующую функцию, получаем порядок роста $T(n) = O(n^2)$, где n - размер массива.

Емкостная сложность сортировки

Емкостная сложность алгоритма порядка $O(n)$ т.к. используется только исходный массив размера n.

Код функции сортировки

```
4   void bubbleSortAiv (std::vector<int> &v, int n) {
5       bool isSorted = false;
6       for (int i = 0; !isSorted && i < n-1; ++i) {
7           isSorted = true;
8           for (int j = 0; j < n-i-1; ++j) {
9               if (v[j] > v[j+1]) {
10                  std::swap ( & v[j], & v[j+1]);
11                  isSorted = false;
12              }
13          }
14      }
15  }
```

Рис. 2 - Код сортировки пузырьком

Тестирование функции сортировки

```
Length: 10
Generated array:
4 8 8 2 4 5 5 1 7 1
Sorted array:
1 1 2 4 4 5 5 7 8 8
Length: 20
Generated array:
11 15 2 7 16 11 4 2 13 12 2 1 16 18 15 7 6 11 18 9
Sorted array:
1 2 2 2 4 6 7 7 9 11 11 11 12 13 15 15 16 16 18 18
```

Рис. 3 - Результаты тестирования на работоспособность сортировки пузырьком

Сводная таблица тестирования

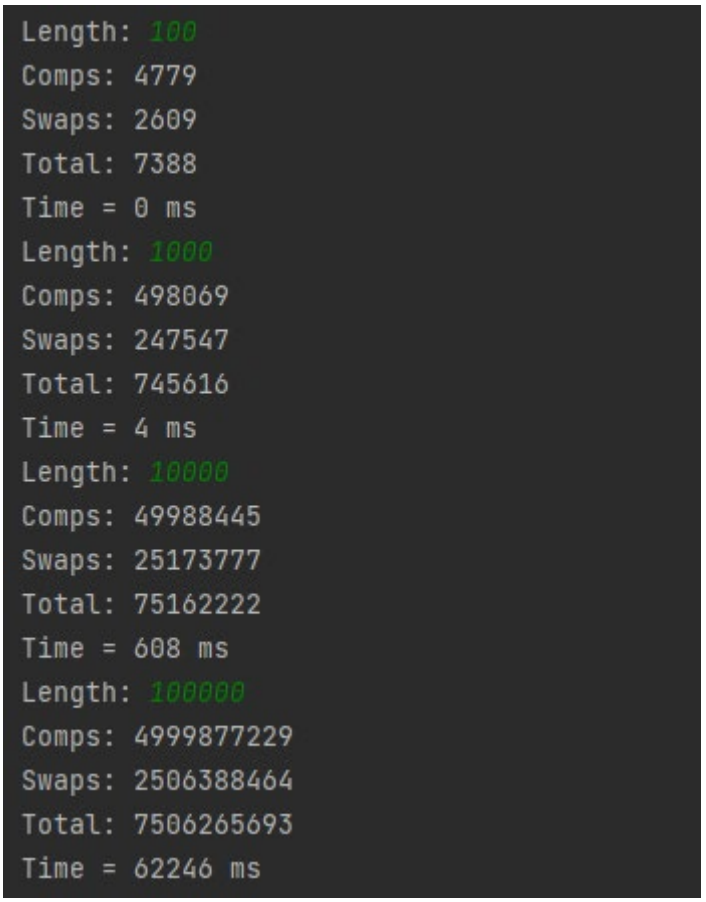


Рис. 4 - Результаты тестирования сортировки пузырьком

Таблица 2 - Сводная таблица тестирования сортировки пузырьком

n	$T(n)$	$T_{\text{т}} = f(C+M)$	$T_{\text{п}} = C_{\text{ф}} + M_{\text{ф}}$
100	<1 мс	$O(n^2)$	7388
1000	4 мс		745616
10000	608 мс		75162222
100000	62246 мс		7506265693
1000000	-		-

Для тестирования была создана версия функции сортировки со встроенной отладкой. (рисунок 5).

```
18  void bubbleSortAivLog (std::vector<int> &v, int n) {
19      bool isSorted = false;
20      long long comps = 0, swaps = 0;
21      for (int i = 0; !isSorted && i < n-1; ++i) {
22          isSorted = true;
23          for (int j = 0; j < n-i-1; ++j) {
24              ++comps;
25              if (v[j] > v[j+1]) {
26                  ++swaps;
27                  std::swap ( &v[j],  &v[j+1]);
28                  isSorted = false;
29              }
30          }
31      }
32      std::cout << "Comps: " << comps << std::endl;
33      std::cout << "Swaps: " << swaps << std::endl;
34      std::cout << "Total: " << comps + swaps << std::endl;
35  }
```

Рис. 5 - Функция bubbleSortAivLog

1.2. Шейкерная сортировка

Постановка задачи

Разработать алгоритм шейкерной сортировки (двусторонний пузырек), провести анализ вычислительной и емкостной сложности алгоритма на массивах, заполненных случайно.

Описание сортировки

Является улучшенной версией сортировки пузырьком. На первом проходе мы задвигаем максимальный элемент в конец массива, потом же идем в обратном направлении и двигаем минимум в начало. Отсортированные крайние области массива увеличиваются после каждой итерации.

Алгоритм сортировки

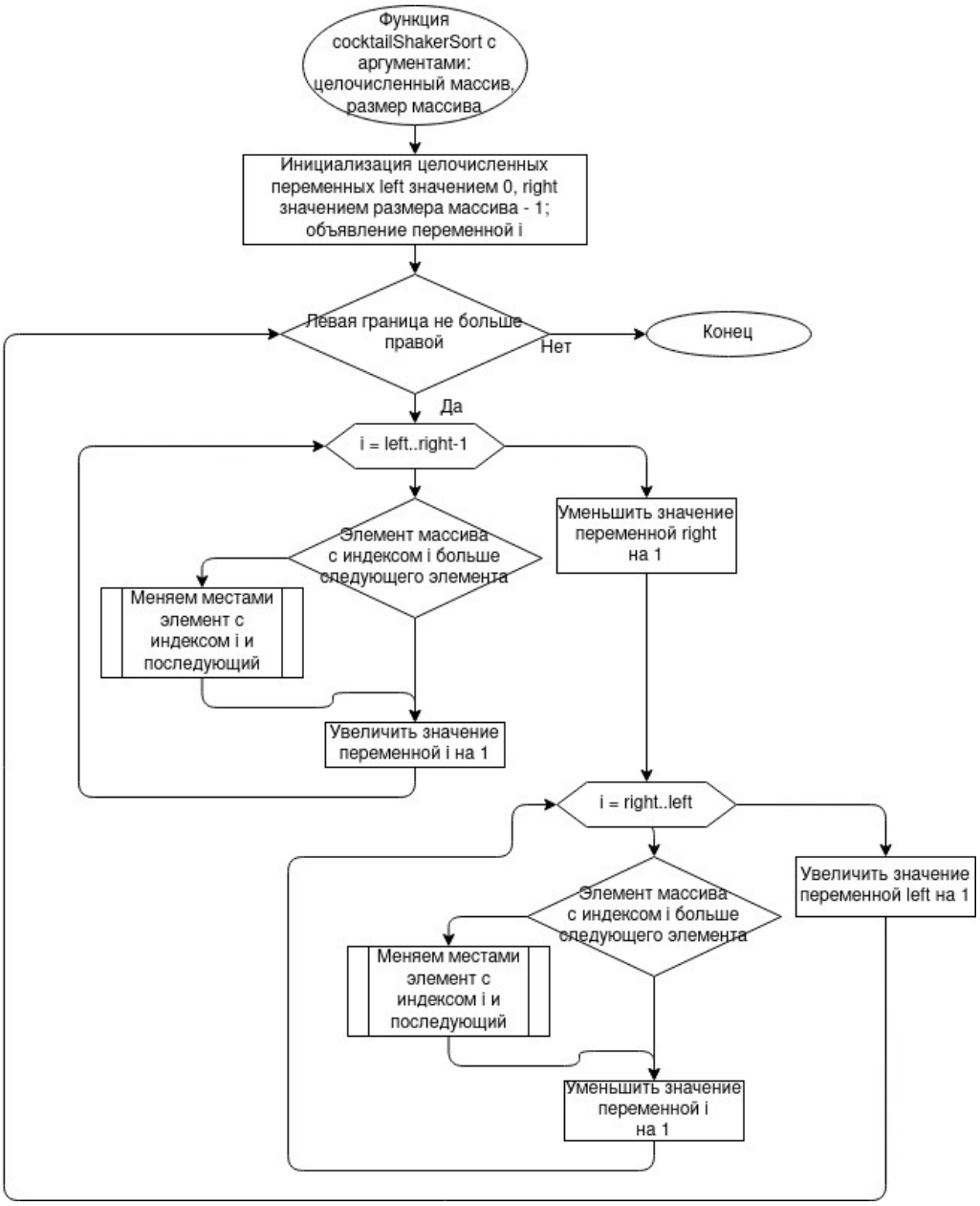


Рис. 6 - Блок-схема шейкерной сортировки

Оценка функции роста скорости выполнения алгоритма сортировки

Определим теоретическую сложность алгоритма при помощи таблицы операторов.

Таблица 3 - Подсчет количества операторов в алгоритме шейкерной сортировки

Номер оператора	Оператор	Время выполнения одного оператора	Кол-во выполнений оператора в строке
1	int left = 0; int right = n - 1;	C1	1 раз
2	while (left ≤ right)	C2	n раз
3	for (int i = left; i < right; ++i) {	C3	n(n-1) раз
4	if (v[i] > v[i+1]) {	C4	n(n-1)-1 раз
5	std::swap(v[i], v[i+1])}}	C5	n(n-1)-1 раз
6	--right;	C6	n-1 раз
7	for (int i = right-1; i ≥ left; --i) {	C7	n(n-1) раз
8	if (v[i] > v[i+1]) {	C8	n(n-1)-1 раз
9	std::swap(v[i], v[i+1])}}	C5	n(n-1)-1 раз
10	++left;}	C10	n-1 раз

Из таблицы 3 получим функцию роста выполнения алгоритма сортировки простого обмена. Пусть $T(n)$ - время выполнения алгоритма, зависящее от n. Тогда

$$T(n) = C_1 + C_2 \cdot n + C_3 \cdot (n^2 - n) + C_4 \cdot (n^2 - n - 1)$$
$$+ C_5 \cdot (n^2 - n - 1) + C_6 \cdot (n - 1) + C_7 \cdot (n^2 - n).$$
$$+ C_8 \cdot (n^2 - n - 1) + C_9 \cdot (n^2 - n - 1) + C_{10} \cdot (n - 1)$$

После упрощения и подведя подобные получаем

$$T(n) = An^2 + Bn + C.$$

Оставляя справа только доминирующую функцию, получаем порядок роста $T(n) = O(n^2)$, где n - размер массива.

Емкостная сложность сортировки

Емкостная сложность алгоритма порядка $O(n)$ т.к. используется только исходный массив размера n.

Код функции сортировки

```
void cocktailShakerSort (std::vector<int> &v, int n) {
    int left = 0, right = n - 1;
    while (left <= right) {
        for (int i = left; i < right; i++) {
            if (v[i] > v[i+1]) {
                std::swap( & v[i],  & v[i+1]);
            }
        }
        --right;
        for (int i = right; i >= left; --i) {
            if (v[i] > v[i+1]) {
                std::swap( & v[i],  & v[i+1]);
            }
        }
        ++left;
    }
}
```

Рис. 7 - Код шейкерной сортировки

Тестирование функции сортировки

```
Length: 10
Generated array:
1 7 4 0 9 4 8 8 2 4
Sorted array:
0 1 2 4 4 4 7 8 8 9
Length: 20
Generated array:
5 5 1 7 1 11 15 2 7 16 11 4 2 13 12 2 1 16 18 15
Sorted array:
1 1 1 2 2 2 4 5 5 7 7 11 11 12 13 15 15 16 16 18
```

Рис. 8 - Результаты тестирования на работоспособность шейкерной сортировки

Сводная таблица тестирования

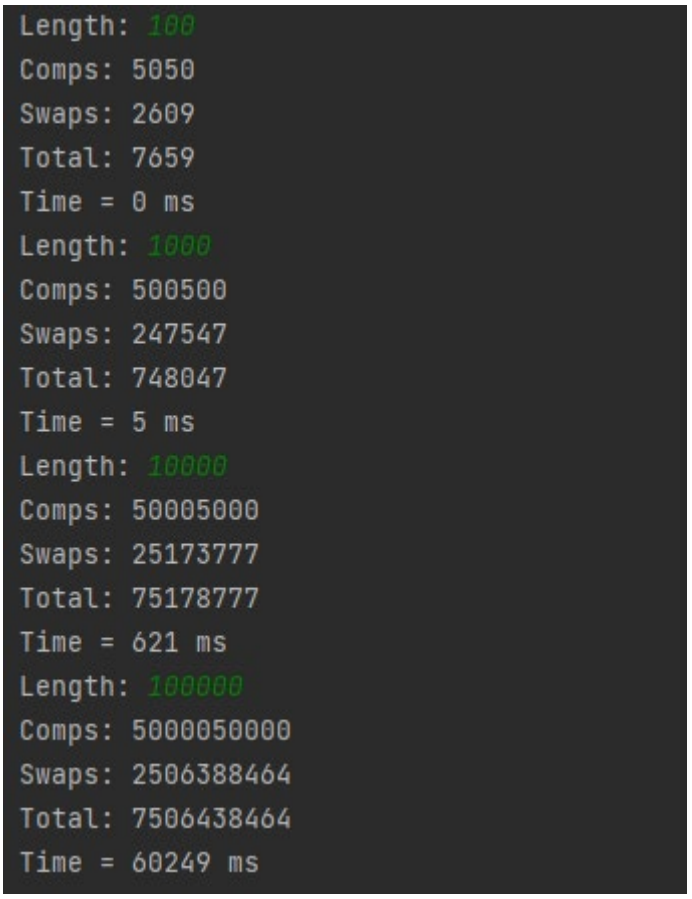


Рис. 9 - Результаты тестирования шейкерной сортировки

Таблица 4 - Сводная таблица тестирования шейкерной сортировки

n	$T(n)$	$T_{\text{Т}} = f(C+M)$	$T_{\text{П}} = C_{\text{Ф}} + M_{\text{Ф}}$
100	<1 мс	$O(n^2)$	7659
1000	5 мс		748047
10000	621 мс		75178777
100000	60249 мс		7506438464
1000000	-		-

Для тестирования была создана версия функции сортировки со встроенной отладкой (рисунок 10).

```cpp
void cocktailShakerSortLog (std::vector<int> &v, int n) {
    int left = 0, right = n - 1;
    long long comps = 0, swaps = 0;
    while (left <= right) {
        ++comps;
        for (int i = left; i < right; i++) {
            ++comps;
            if (v[i] > v[i+1]) {
                ++swaps;
                std::swap( & v[i],  & v[i+1]);
            }
        }
        --right;
        for (int i = right; i >= left; --i) {
            ++comps;
            if (v[i] > v[i+1]) {
                ++swaps;
                std::swap( & v[i],  & v[i+1]);
            }
        }
        ++left;
    }
    std::cout << "Comps: " << comps << std::endl;
    std::cout << "Swaps: " << swaps << std::endl;
    std::cout << "Total: " << comps + swaps << std::endl;
}
```

Рис. 10 - Функция cocktailShakerSortLog

1.3. Анализ результатов 1-й и 2-й сортировок

По таблицам 2 и 4 непросто заметить разницу в скорости выполнения сортировки. Несмотря на одинаковую асимптотическую сложность, шейкерная сортировка все же в среднем имеет намного меньше операций перестановки по сравнению с пузырьковой, что можно заметить по практической вычислительной сложности алгоритма и скорости выполнения программы. Отсюда следует, что алгоритм шейкерной сортировки в среднем случае эффективнее алгоритма сортировки пузырьком с условием Айверсона.

1.4. Графики зависимостей практических вычислительных сложностей 1-й и 2-й сортировок

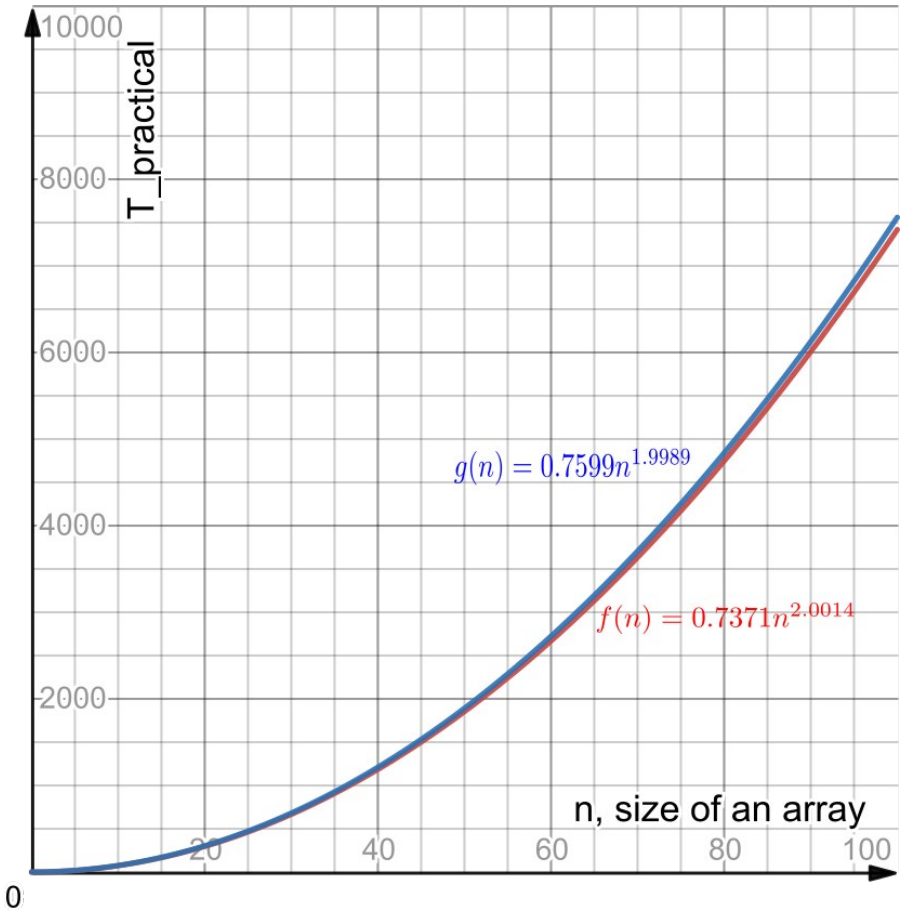


Рис. 11 - Сравнение скоростей сортировок пузырьком и шейкерной

На рис. 11 приведены графики зависимостей практических вычислительных сложностей алгоритмов сортировки пузырком с условием Айверсона $f(n) = 0.7371n^{2.0014}$ и шейкерной сортировки от размера n массива $g(n) = 0.7599n^{1.9989}$.По графикам можно заметить разницу в росте времени работы алгоритмов – время работы шейкерной сортировки растет медленнее с увеличением размера массива.

1.5. Сортировка слиянием

Постановка задачи

Разработать алгоритм сортировки простым слиянием. Сформировать таблицу результатов для массива, заполненного случайными числами. Определить емкостную сложность алгоритма. Определить асимптотическую сложность алгоритма.

Описание алгоритма сортировки

Сортировка слиянием состоит из двух главных действий:

1. Разделить неотсортированный массив на n подмассивов, каждый из которых содержит один элемент (т.е. массив считается отсортированным).
2. Повторно производить слияние подмассивов для создания больших по размеру сортированных подмассивов, пока не останется единственный подмассив, который и будет нашим отсортированным массивом.

Алгоритм сортировки

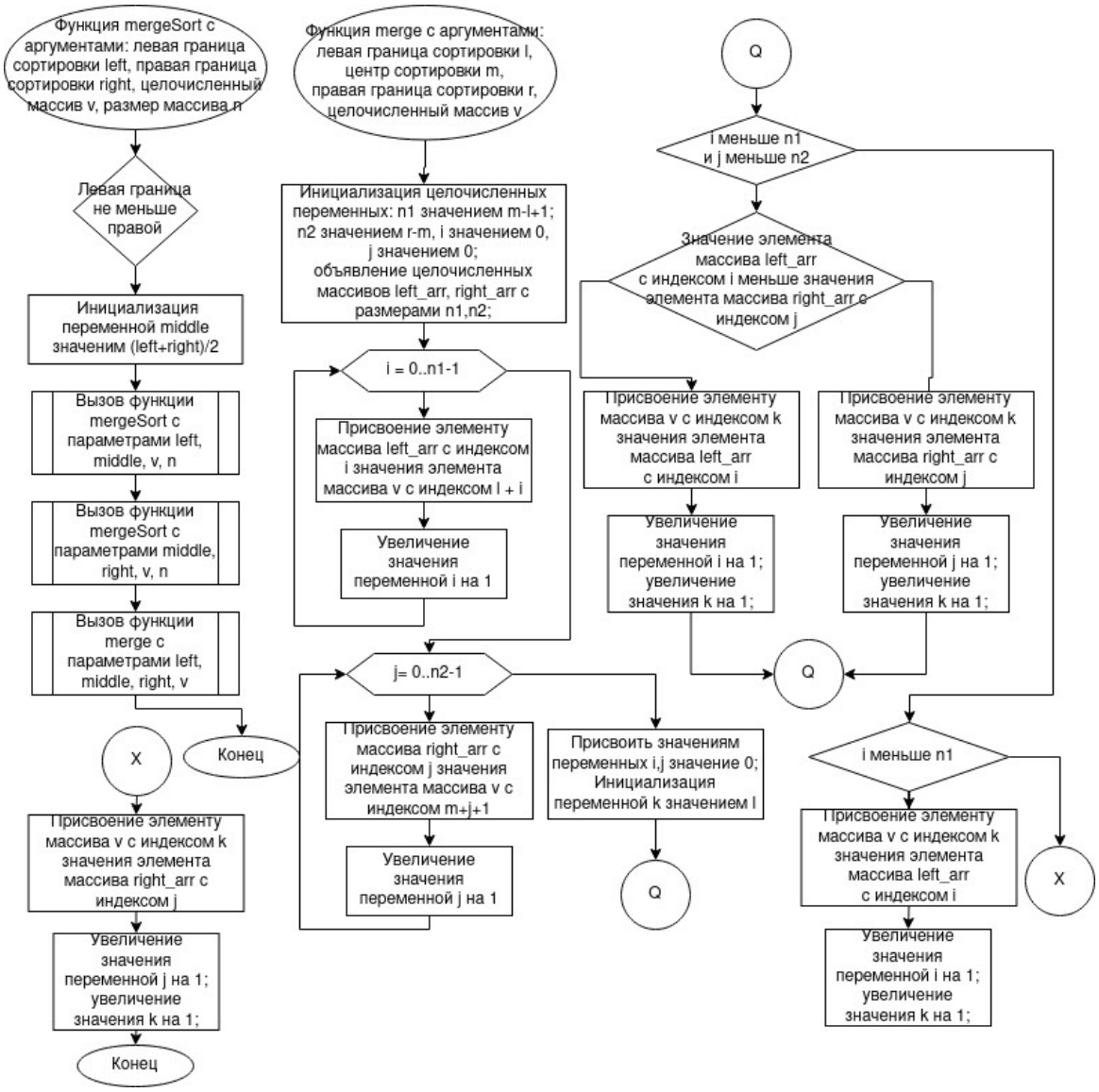


Рис. 12 - Блок-схема сортировки простым слиянием

Оценка функции роста скорости выполнения алгоритма сортировки

Временная сложность функции merge = $\Theta(n)$ т.к. в функции нет вложенных циклов и не происходят операции со скоростью меньше чем $\Theta(n)$. Для оценки скорости выполнения рекурсивного алгоритма mergeSort сначала запишем его в рекуррентом виде

$$T(n) = aT(\frac{n}{b}) + f(n), \text{где } a \geq 1, b \geq 1 \quad (1)$$

где n - размер задачи, a - количество задач в подрекурсии, $\frac{n}{b}$ - размер каждой подзадачи, $f(n)$ - оценка сложности работы, производимой алгоритмом вне рекурсивных вызовов. Для сортировки слиянием: $f(n) = \Theta(n)$, т.к. кроме рекурсивных вызовов происходят только вызова функций со сложностью $\Theta(n)$; $a = 2$т.к. мы вызываем две подзадачи; $b = 2$ т.к. мы разбиваем текущий массив на два подмассива. Отсюда мы получаем

$$T(n) = 2T(\frac{n}{2}) + \Theta(n).$$

По Мастер теореме: если $f(n) = \Theta(n)$, то тогда $T(n) = \Theta(n \log n)$.

Емкостная сложность алгоритма

Т.к. в данной задаче используются дополнительные массивы размера входного массива n, то емкостная сложности алгоритма равняется $O(n)$ (также память расходуется на рекурсивные вызовы, но ими можно пренебречь по сравнению с памятью на создание дополнительных массивов).

Код функции сортировки

```
void merge (int l, int m, int r, std::vector<int> &v) {
    int n1 = m - l + 1;
    int n2 = r - m;
    std::vector<int> left_arr(n1), right_arr(n2);
    for (int i = 0; i < n1; ++i) {
        left_arr[i] = v[l + i];
    }
    for (int j = 0; j < n2; ++j) {
        right_arr[j] = v[m + 1 + j];
    }
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (left_arr[i] < right_arr[j]) {
            v[k++] = left_arr[i++];
        }
        else {
            v[k++] = right_arr[j++];
        }
    }
    while (i < n1) {
        v[k++] = left_arr[i++];
    }
    while (j < n2) {
        v[k++] = right_arr[j++];
    }
}

void mergeSort (int left, int right, std::vector<int> &v) {
    if (left >= right) {
        return;
    }
    int middle = (left + right) / 2;
    mergeSort(left, middle, &v);
    mergeSort( left: middle + 1, right, &v);
    merge(left, middle, right, &v);
}
```

Рис. 13 - Код сортировки слиянием

Тестирование функции сортировки

```
Length: 10
Generated array:
1 7 4 0 9 4 8 8 2 4
Sorted array:
0 1 2 4 4 4 7 8 8 9
Length: 20
Generated array:
5 5 1 7 1 11 15 2 7 16 11 4 2 13 12 2 1 16 18 15
Sorted array:
1 1 1 2 2 2 4 5 5 7 7 11 11 12 13 15 15 16 16 18
```

Рис. 14 - Результаты тестирования на работоспособность сортировки простым слиянием

Сводная таблица тестирования

```
Length: 100
Comps: 1416
Moves: 1344
Total: 2760
Time = 0 ms
Length: 1000
Comps: 22094
Moves: 21296
Total: 43390
Time = 0 ms
Length: 10000
Comps: 296138
Moves: 288528
Total: 584666
Time = 4 ms
Length: 100000
Comps: 3701241
Moves: 3626384
Total: 7327625
Time = 73 ms
Length: 1000000
Comps: 44326887
Moves: 43529232
Total: 87856119
Time = 731 ms
```

Рис. 15 - Результаты тестирования сортировки слиянием

Таблица 5 - Сводная таблица тестирования сортировки слиянием

n	$T(n)$	$T_T = f(C+M)$	$T_п = C_ф + M_ф$
100	<1 мс	$\Theta(n\log n)$	2760
1000	<1 мс		43390
10000	4 мс		584666
100000	73 мс		7327625
1000000	731 мс		87856119

Для тестирования была создана версия функции сортировки со встроенной отладкой. (рисунок 16).

```
55  void mergeLog (int l, int m, int r, std::vector<int> &v, long long &comps, long long &moves) {
56      int n1 = m - l + 1, n2 = r - m;
57      std::vector<int> left_arr(n1), right_arr(n2);
58      for (int i = 0; i < n1; ++i)
59          { left_arr[i] = v[l + i]; ++moves; }
60      for (int j = 0; j < n2; ++j)
61          { right_arr[j] = v[m + 1 + j]; ++moves; }
62      int i = 0, j = 0, k = l;
63      while (i < n1 && j < n2) {
64          comps += 2;
65          if (left_arr[i] < right_arr[j])
66              { v[k++] = left_arr[i++]; ++moves; }
67          else
68              { v[k++] = right_arr[j++]; ++moves; }
69      }
70      while (i < n1)
71          { v[k++] = left_arr[i++]; ++comps; ++moves; }
72      while (j < n2)
73          { v[k++] = right_arr[j++]; ++comps; ++moves; }
74  }
75
76  void mergeSortLog (int left, int right, std::vector<int> &v, long long &comps, long long &moves) {
77      ++comps;
78      if (left >= right) { return; }
79      int middle = (left + right) / 2;
80      mergeSortLog(left, middle,  &v,  &comps,  &moves);
81      mergeSortLog( left: middle + 1, right,  &v,  &comps,  &moves);
82      mergeLog(left, middle, right,  &v,  &comps,  &moves);
83  }
```

Рис. 16 - Функция mergeSortLog

1.6. Анализ результатов 2-й и 3-й сортировок

Для сравнения шейкерной сортировки и сортировки простого слияния сравним результаты из таблиц 4 и 5. Из таблиц вычислительная сложность обоих алгоритмов повторяет найденную теоретически, при этом сортировка слиянием с асимптотической сложностью $\Theta(n \log n)$ превосходит в скорости работы шейкерную сортировку со сложностью $O(n^2)$. Таким образом, алгоритм сортировки простым слиянием эффективнее алгоритма шейкерной сортировки повременной сложности в среднем случае.

1.7. Графики зависимостей практических вычислительных сложностей 2-й и 3-й сортировок

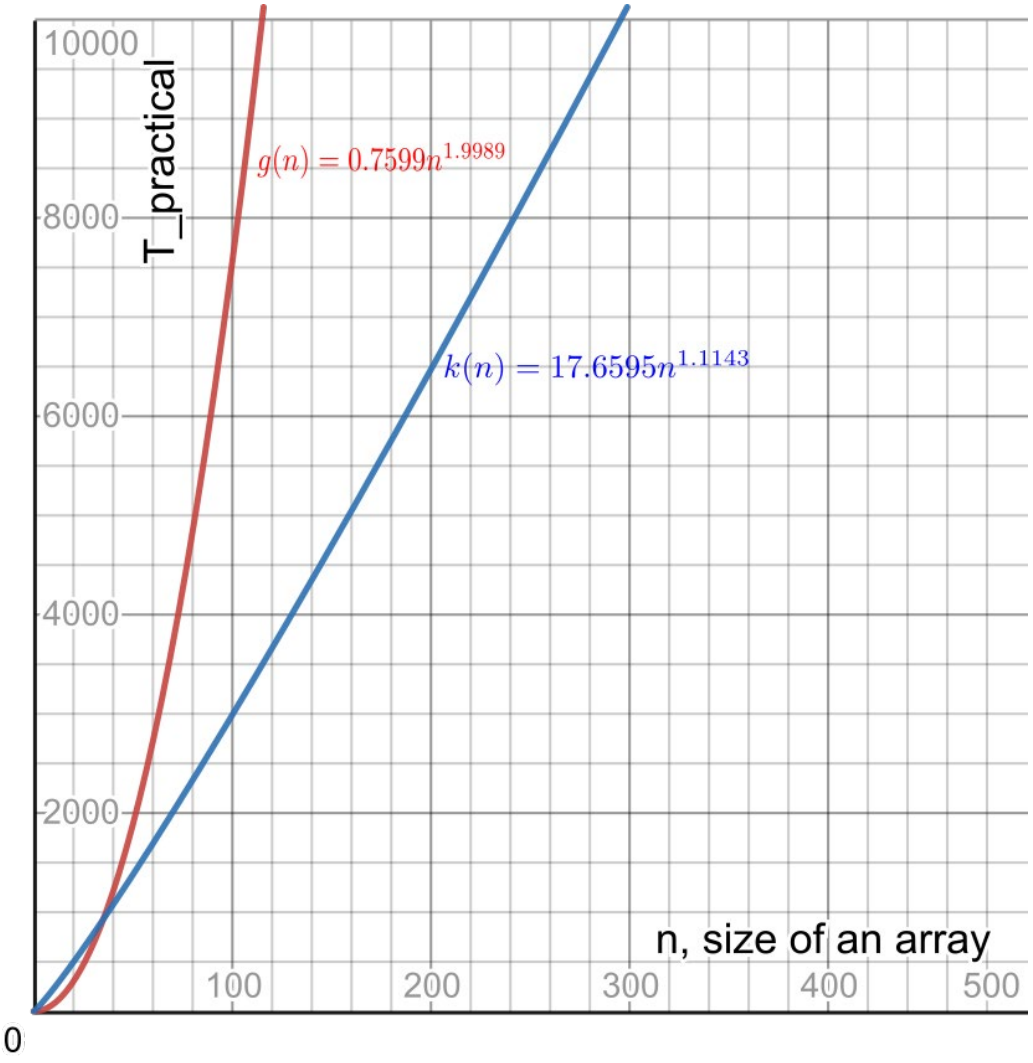


Рис. 17 - Сравнение скоростей шейкерной и простого слияния сортировок

На рис. 17 приведены графики зависимостей практических вычислительных сложностей алгоритмов шейкерной сортировки $g(n) = 0.7599n^{1.9989}$ и сортировки слиянием $k(n) = 17.6595n^{1.1143}$. По графикам можно заметить, что практическая сложность алгоритма сортировки слиянием растет значительно медленней сложности шейкерной сортировки.

Задание 2. Определение эффективного из алгоритмов для наихудшего и наилучшего случаев

2.1. Результаты тестирования алгоритмов на упорядоченных массивах

```
Length: 100
Comps: 99
Swaps: 0
Total: 99
Time = 14 mcs
Length: 1000
Comps: 999
Swaps: 0
Total: 999
Time = 14 mcs
Length: 10000
Comps: 9999
Swaps: 0
Total: 9999
Time = 66 mcs
Length: 100000
Comps: 99999
Swaps: 0
Total: 99999
Time = 484 mcs
Length: 1000000
Comps: 999999
Swaps: 0
Total: 999999
Time = 3387 mcs
```

Рис. 18 - Результаты тестирования сортировки пузырьком с условием Айверсона в лучшем случае

Таблица 6 - Сводная таблица тестирования в лучшем случае

n	$T(n)$	$T_{\text{Т}} = f(C+M)$	$T_{\text{п}} = C_{\text{ф}} + M_{\text{ф}}$
100	0.014 мс	$O(n)$	99
1000	0.014 мс		999
10000	0.066 мс		9999
100000	0.484 мс		99999
1000000	3.387 мс		999999

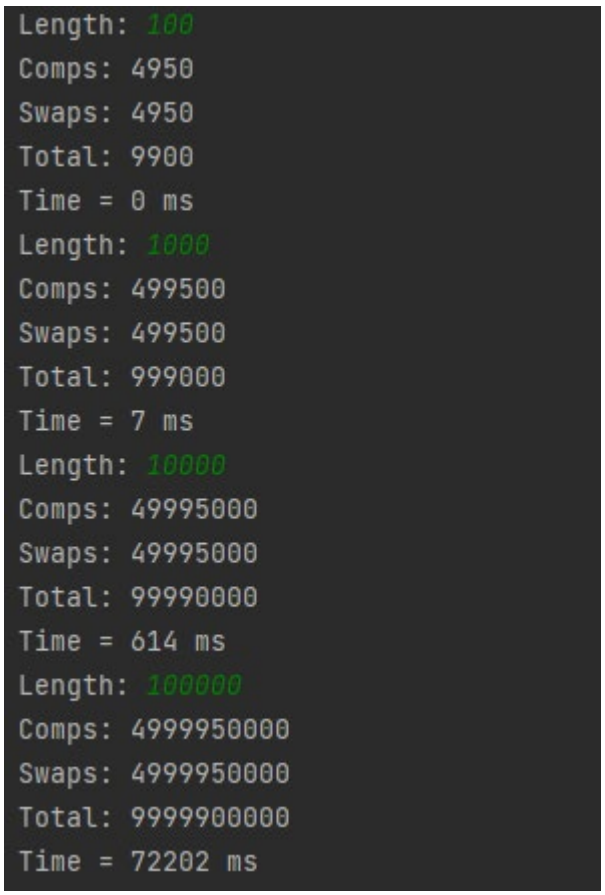


Рис. 19 - Результаты тестирования сортировки пузырьком с условием Айверсона в худшем случае

Таблица 7 - Сводная таблица тестирования в худшем случае

n	$T(n)$	$T_{\text{Т}} = f(C+M)$	$T_{\text{П}} = C_{\text{ф}} + M_{\text{ф}}$
100	<1 мс	$O(n^2)$	9900
1000	7 мс		999000
10000	614 мс		99990000
100000	72202 мс		9999900000
1000000	-		-

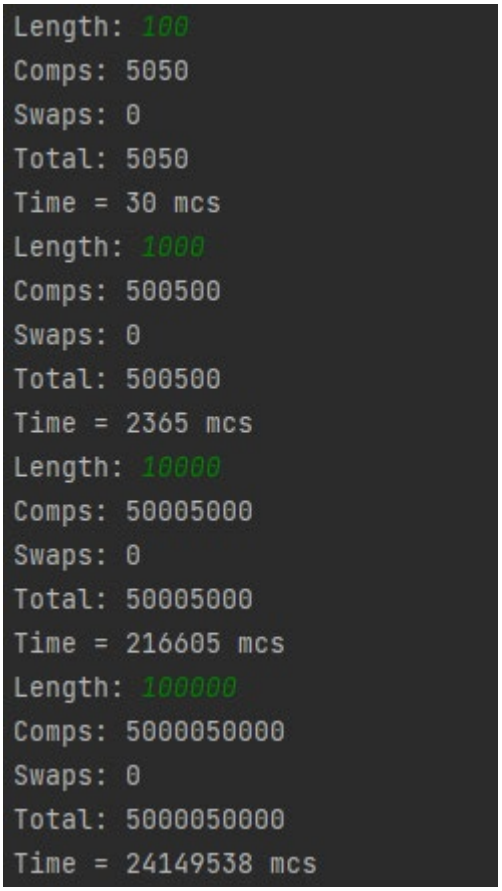


Рис. 20 - Результаты тестирования шейкерной сортировки в лучшем случае

Таблица 8 - Сводная таблица тестирования в лучшем случае

n	$T(n)$	$T_{\text{Т}} = f(C+M)$	$T_{\text{п}} = C_{\text{ф}} + M_{\text{ф}}$
100	0.030 мс	$O(n^2)$	5050
1000	2.365 мс		500500
10000	216 мс		50005000
100000	24149 мс		5000050000
1000000	-		-

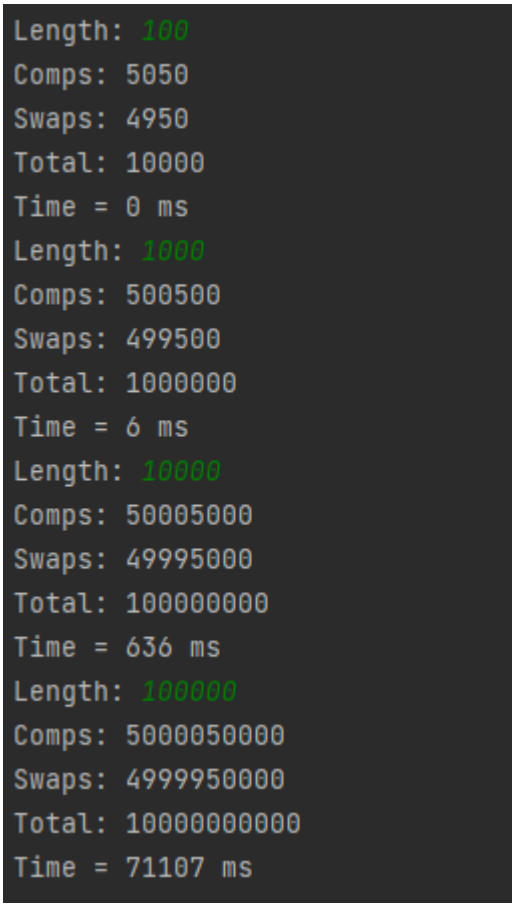


Рис. 21 - Результаты тестирования шейкерной сортировки в худшем случае

Таблица 9 - Сводная таблица тестирования в худшем случае

n	$T(n)$	$T_{\text{Т}} = f(C+M)$	$T_{\text{п}} = C_{\text{ф}} + M_{\text{ф}}$
100	<1 мс	$O(n^2)$	10000
1000	6 мс		1000000
10000	636 мс		100000000
100000	71107 мс		10000000000
1000000	-		-

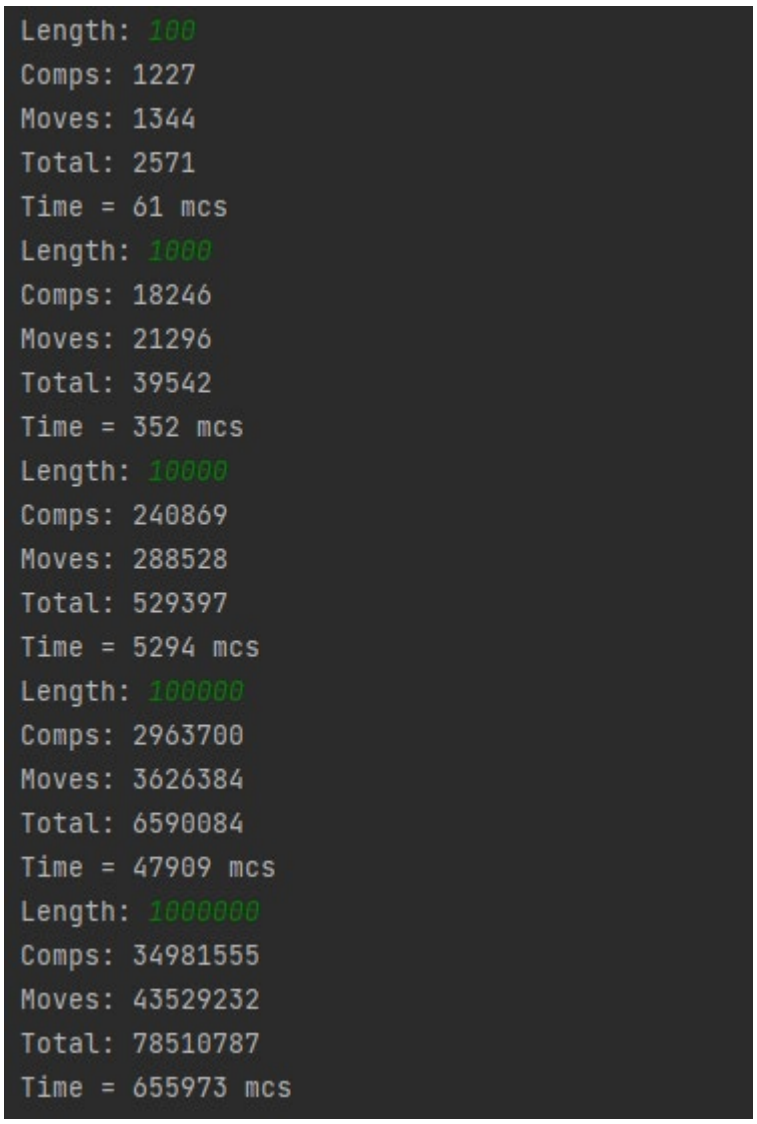


Рис. 22 - Результаты тестирования сортировки слиянием в лучшем случае

Таблица 10 - Сводная таблица тестирования в лучшем случае

n	$T(n)$	$T_{\text{т}} = f(C+M)$	$T_{\text{п}} = C_{\text{ф}} + M_{\text{ф}}$
100	0.061 мс	$\Theta(n\log n)$	2571
1000	0.352 мс		39542
10000	5.294 мс		529397
100000	48 мс		6590084
1000000	656 мс		78510787

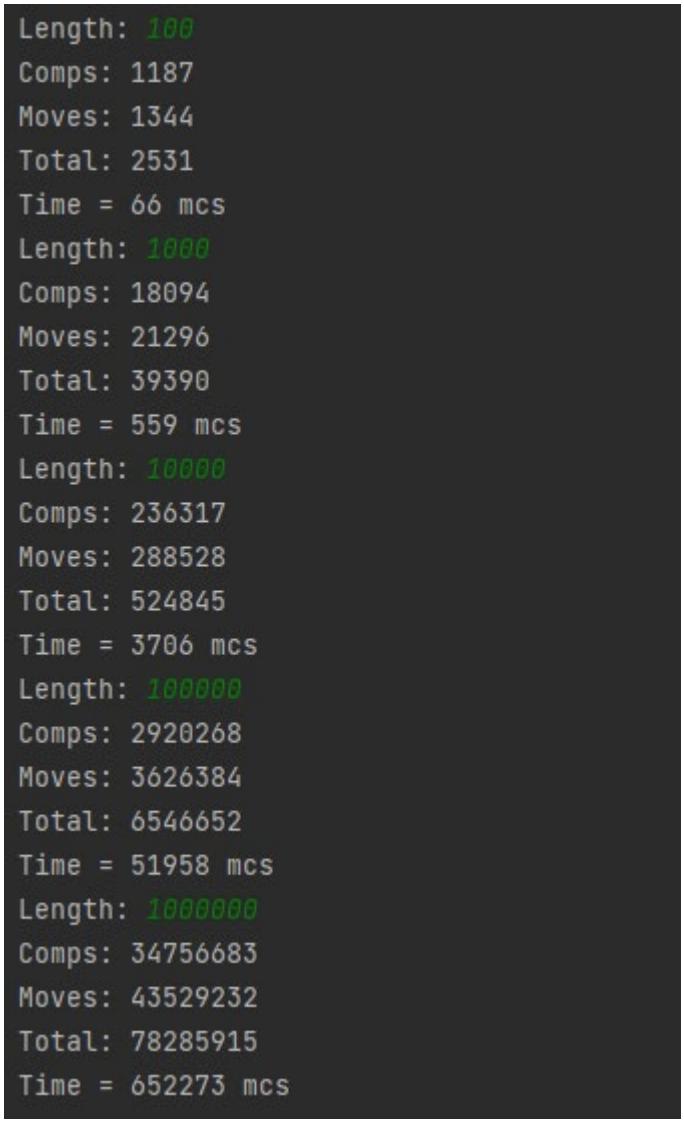
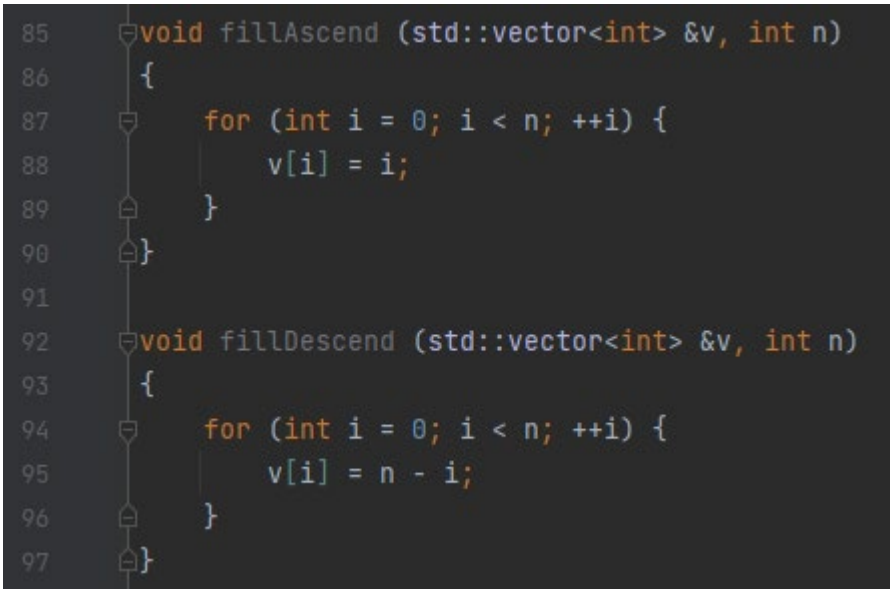


Рис. 23 - Результаты тестирования сортировки слиянием в худшем случае

Таблица 11 - Сводная таблица тестирования в худшем случае

n	$T(n)$	$T_{\text{Т}} = f(C+M)$	$T_{\text{п}} = C_{\text{ф}} + M_{\text{ф}}$
100	0.066 мс	$\Theta(n\log n)$	2531
1000	0.559 мс		39390
10000	3.706 мс		524845
100000	51 мс		6546652
1000000	652 мс		78285915

Функции для тестирования сортировки в лучшем и худшем случаях представлены на рисунке 24.



```
85  void fillAscend (std::vector<int> &v, int n)
86  {
87      for (int i = 0; i < n; ++i) {
88          v[i] = i;
89      }
90  }
91
92  void fillDescend (std::vector<int> &v, int n)
93  {
94      for (int i = 0; i < n; ++i) {
95          v[i] = n - i;
96      }
97  }
```

Рис. 24 - Код функций для заполнения массивов по возрастанию и убыванию

2.2. Асимптотическая сложность алгоритмов в лучшем и худшем случаях

Алгоритм сортировки пузырьком с условием Айверсона: в лучшем случае имеет асимптотическую сложность $O(n)$, в худшем - $O(n^2)$. Алгоритм шейкерной сортировки: в лучшем и худшем случаях имеет асимптотическую сложность $O(n^2)$.

Алгоритм сортировки слиянием: в лучшем и худшем случаях имеет асимптотическую сложность $\Theta(n \log n)$.

Из вышерасмотренных алгоритмов самым эффективным является алгоритм сортировки слиянием, имея значительное преимущество во временной асимптотической сложности.

2.3. Таблица асимптотической сложности алгоритмов

Таблица 12 - Асимптотическая сложность алгоритмов, рассмотренных в данной работе

	Асимптотическая сложность алгоритма			
Алгоритм	Наихудший случай	Наилучший случай	Средний случай	Емкостная сложность
Сортировка пузырьком с условием Айверсона	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$
Шейкерная сортировка	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n)$
Сортировка простым слиянием	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	$O(n)$, $O(n)$ дополнительно

Выводы

В ходе практической работы были разработаны алгоритмы сортировки простого обмена с условием Айверсона, шейкерной сортировки и сортировки простого слияния, приобретены навыки по анализу вычислительной сложности алгоритмов сортировки и определен наиболее эффективный алгоритм (сортировка простым слиянием).

Список используемой литературы

1. Thomas H. Cormen, Clifford Stein и другие: Introduction to Algorithms, 3rd Edition. Сентябрь 2009. The MIT Press.

2. B. Strousrup: A Tour of C++ (2nd Edition). Июль 2018. Addison-Wesley.

3. Merge sort // Wikipedia
   [Электронный ресурс]. URL:
   https://en.wikipedia.org/wiki/Merge_sort (Дата обращения: 18.04.2021)

4. Курс Algorithms, part 1 // Coursera [Электронный ресурс]. URL:
   https://www.coursera.org/learn/algorithms-part1 (Дата обращения: 18.04.2021)