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 Хвостов В. В.

Группа ИКБО-01-20

Москва 2021

Содержание

Задание 1. Определение эффективного алгоритма в среднем случае

2 Вариант.

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 проходе на свое место «всплывет» второй по величине элемент и т.д. Отсутствие перестановок на какой-либо итерации означает упорядоченность массива (условие Айверсона).

Алгоритм сортировки

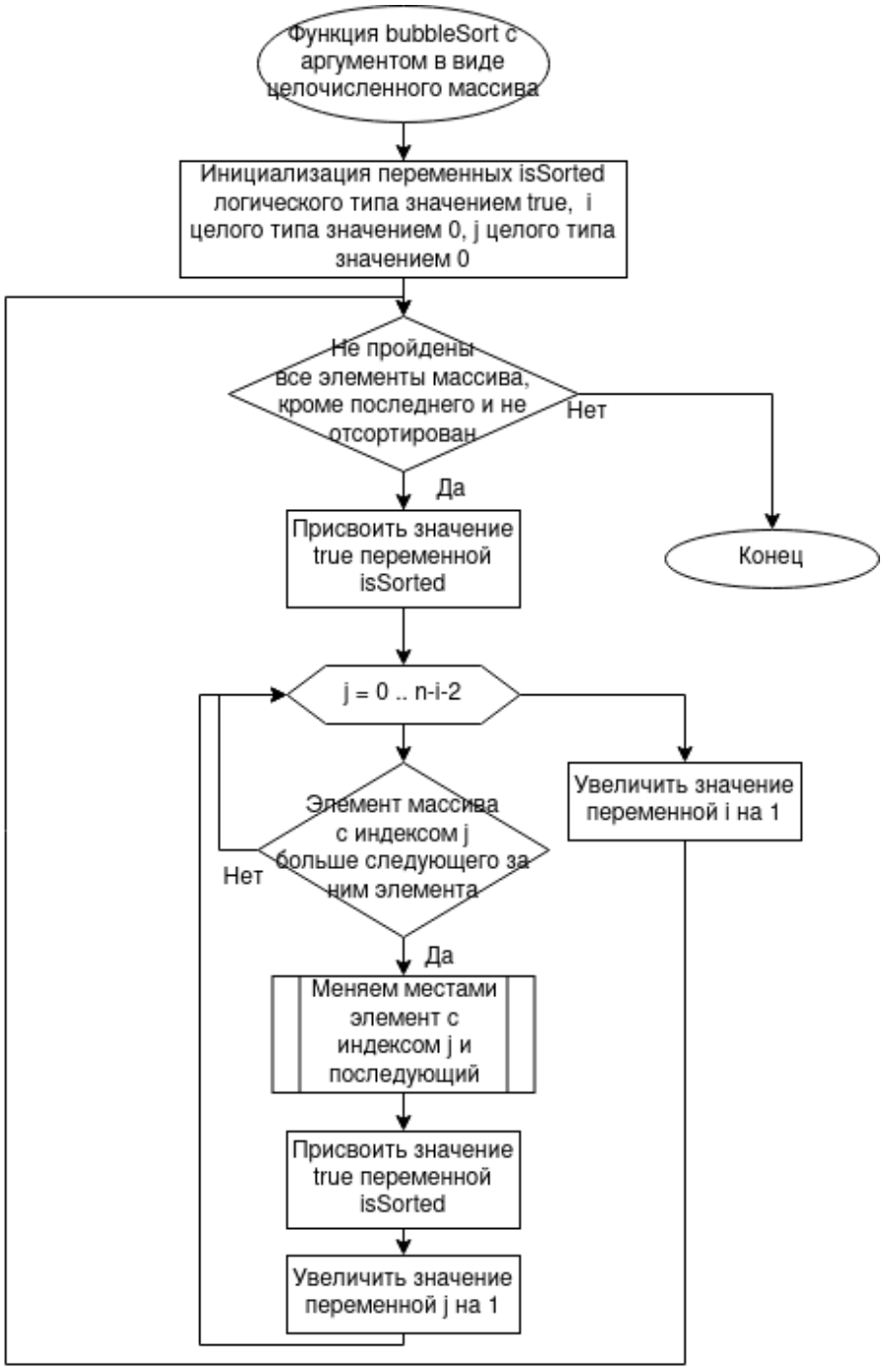

Рис. 1 - Блок-схема сортировки пузырьком с условием Айверсона

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

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

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

```
void bubbleSortAiv(std::vector<int>& v, int n) {
  bool isSorted = false;
  for (int i = 0; !isSorted && i < n - 1; ++i) {
    isSorted = true;
    for (int j = 0; j < n - i - 1; ++j) {
      if (v[j] > v[j+1]) {
        std::swap(v[j], v[j+1]);
        isSorted = false;
      }
    }
  }
}
```

Рис. 2 - Код сортировки пузырьком

Тестирование функции сортировки

```
Enter the size of an array: 5
1 3 4 3 2
1 2 3 3 4
vitalir@swiftly:~/Documents/education/latex/univers
Enter the size of an array: 10
1 9 7 8 7 9 3 5 1 4
1 1 3 4 5 7 7 8 9 9
vitalir@swiftly:~/Documents/education/latex/univers
Enter the size of an array: 20
1 15 15 13 16 6 17 14 17 1 2 5 11 20 1 1 5 18 19 4
1 1 1 1 2 4 5 5 6 11 13 14 15 15 16 17 17 18 19 20
```

Рис. 3 - Результаты тестирования на работоспособность сортировки пузырьком

Сводная таблица тестирования

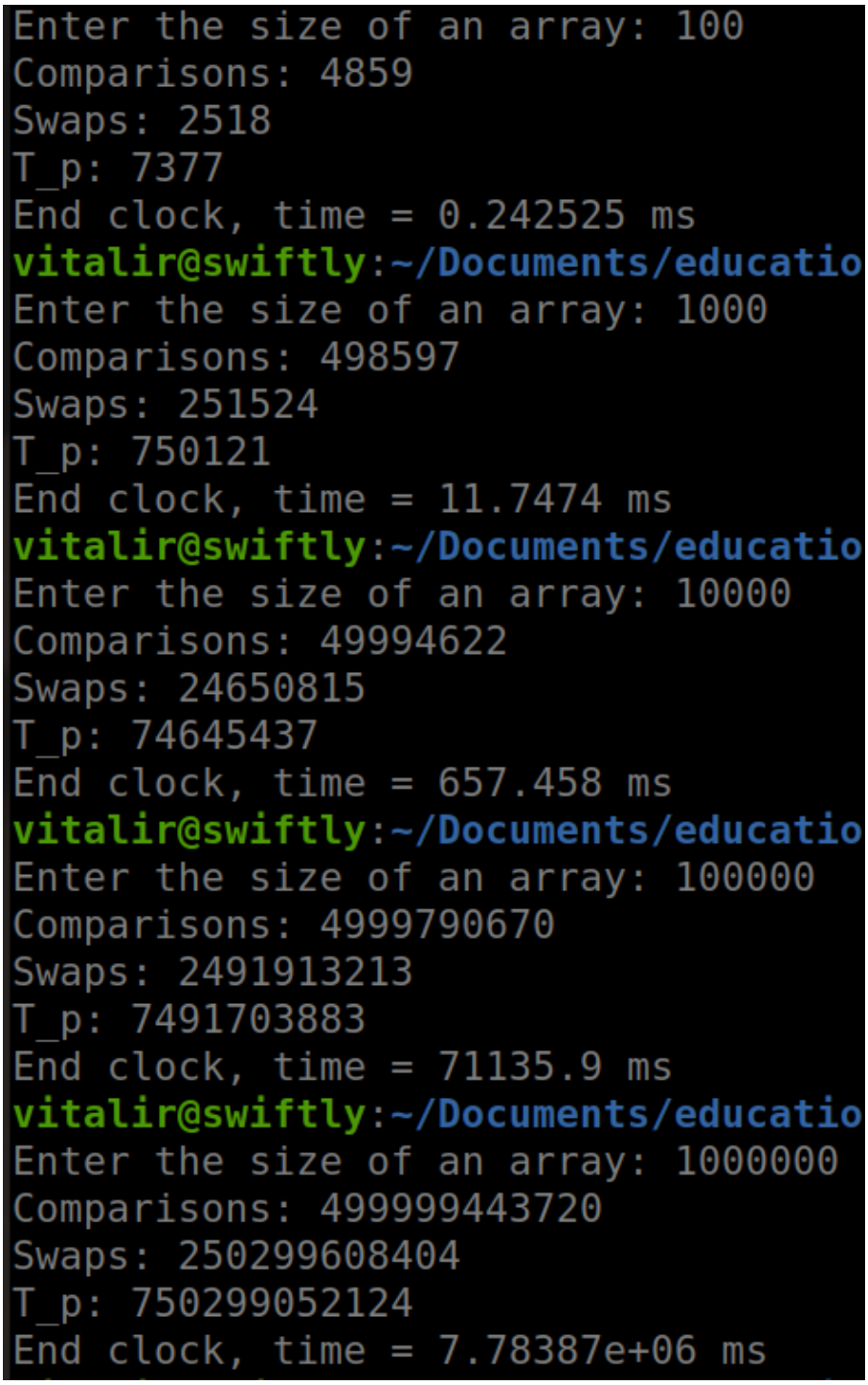

Рис. 4 - Результаты тестирования сортировки пузырьком

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	0.242525 мс	$O(n^2)$	7377
1000	11.7474 мс		750121
10000	657.458 мс		74645458
100000	71135.9 мс		7491703883
1000000	7783870 мс		750299052124

Для тестирования был создан класс для подсчета времени TimeCounter; также была создана версия функции сортировки со встроенной отладкой. (рисунки 5 и 6).

```
class TimeCounter {
public:
  TimeCounter() {
//    std::clog << "Start clock\n";
    start = std::chrono::steady_clock::now();
  }
  ~TimeCounter() {
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double, std::milli> diff = end - start;
    std::clog << "End clock, time = " << diff.count() << " ms" << std::endl;
  }
private:
  std::chrono::time_point<std::chrono::steady_clock> start;

};
```

Рис. 5 - Класс TimeCounter

```cpp
void bubbleSortAivLog(std::vector<int>& v, int n) {
  bool isSorted = false;
  int64_t comparisons = 0, swaps = 0;
  for (int i = 0; !isSorted && i < n - 1; ++i) {
    isSorted = true;
    for (int j = 0; j < n - i - 1; ++j) {
    ++comparisons;
      if (v[j] > v[j+1]) {
        ++swaps;
        std::swap(v[j], v[j+1]);
        isSorted = false;
      }
    }
  }
  std::clog << "Comparisons: " << comparisons;
  std::clog << "\nSwaps: " << swaps << '\n';
  std::clog << "T_p: " << comparisons + swaps << '\n';
}
```

Рис. 6 - 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

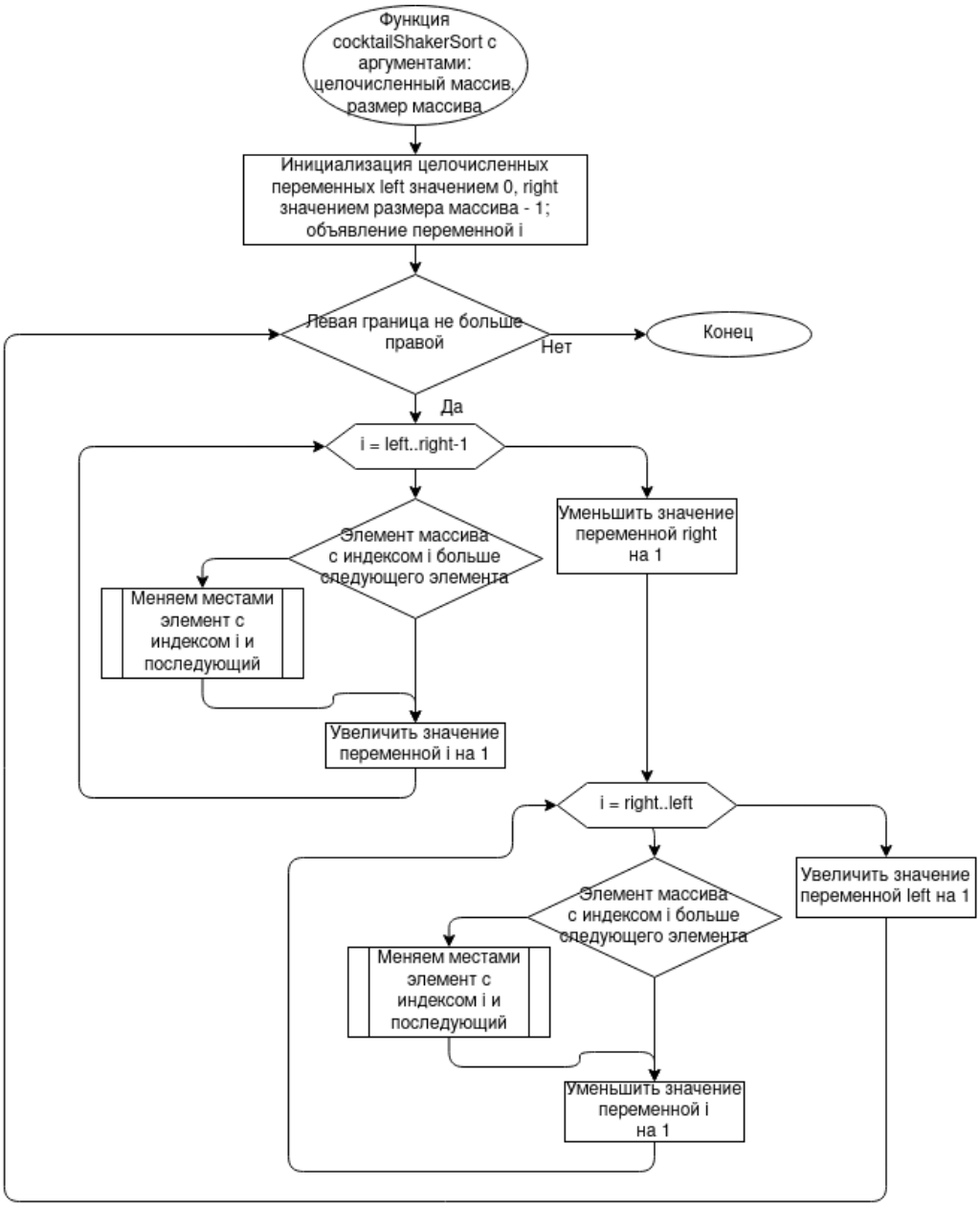

Рис. 7 - Блок-схема шейкерной сортировки

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 $\leq$ 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 $\geq$ 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

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

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;
  int 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. 8 - Код шейкерной сортировки

Тестирование функции сортировки

```
Enter the size of an array: 5
3 5 3 4 1
1 3 3 4 5
vitalir@swiftly:~/Documents/education/latex/univers
Enter the size of an array: 10
4 7 2 4 9 10 8 3 9 10
2 3 4 4 7 8 9 9 10 10
vitalir@swiftly:~/Documents/education/latex/univers
Enter the size of an array: 20
1 16 1 16 10 1 2 15 10 11 7 13 19 17 13 13 3 7 7 7
1 1 1 2 3 7 7 7 7 10 10 11 13 13 13 15 16 16 17 19
```

Рис. 9 - Результаты тестирования на работоспособность шейкерной сортировки

Сводная таблица тестирования

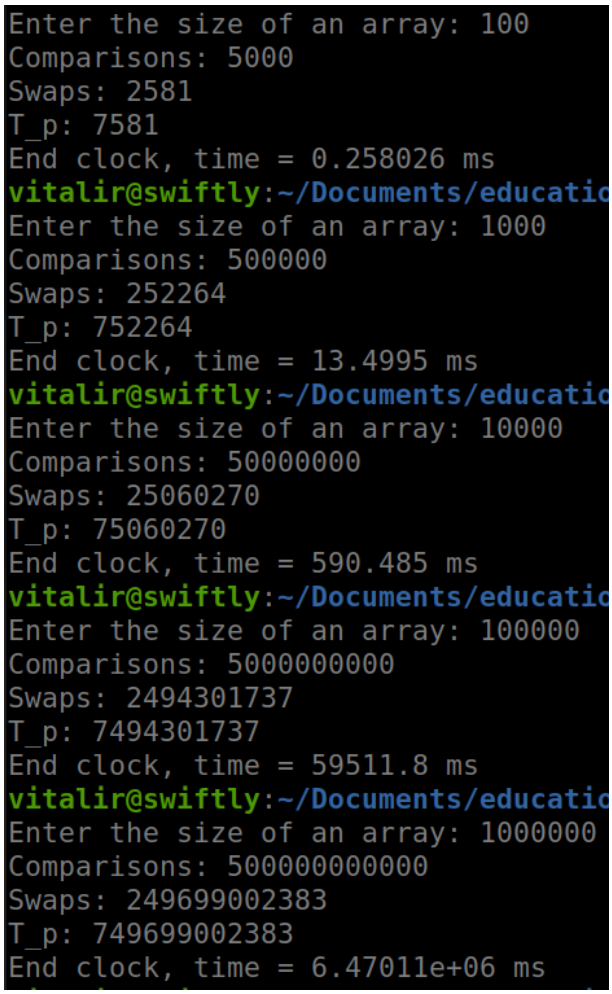

Рис. 10 - Результаты тестирования шейкерной сортировки

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	0.258026 мс	$O(n^2)$	7581
1000	13.4995 мс		752264
10000	590.485 мс		75060270
100000	59511.8 мс		7494301737
1000000	6470110 мс		749699002383

Для тестирования был использован класс для подсчета времени TimeCounter; также была создана версия функции сортировки со встроенной отладкой. (рисунки 5 и 11).

```
void cocktailShakerSortLog(std::vector<int>& v, int n) {
  int left = 0;
  int right = n - 1;
  int64_t comparisons = 0, swaps = 0;
  while (left <= right) {
    ++comparisons;
    for (int i = left; i < right; ++i) {
      ++comparisons;
      if (v[i] > v[i+1]) {
        ++swaps;
        std::swap(v[i], v[i+1]);
      }
    }
    --right;
    for (int i = right - 1; i >= left; --i) {
      ++comparisons;
      if (v[i] > v[i+1]) {
        ++swaps;
        std::swap(v[i], v[i+1]);
      }
    }
    ++left;
  }
  std::clog << "Comparisons: " << comparisons;
  std::clog << "\nSwaps: " << swaps << '\n';
  std::clog << "T_p: " << comparisons + swaps << '\n';
}
```

Рис. 11 - 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

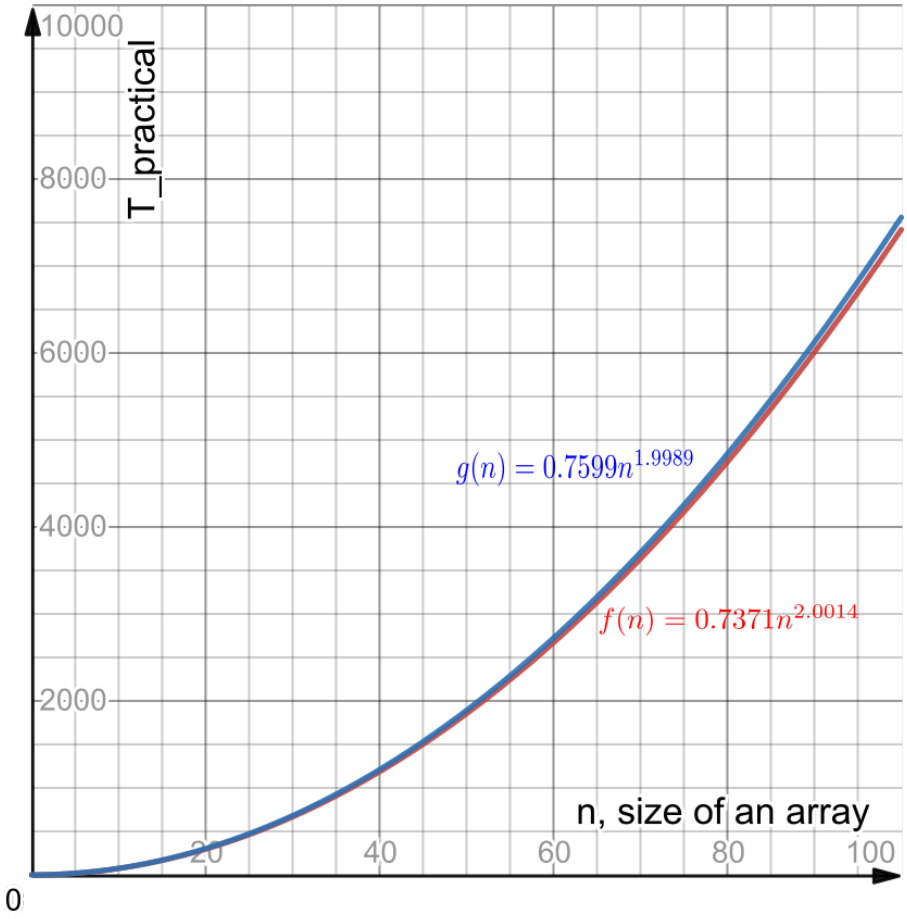

Рис. 12 - Сравнение скоростей сортировок пузырьком и шейкерной

На рис. 12 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

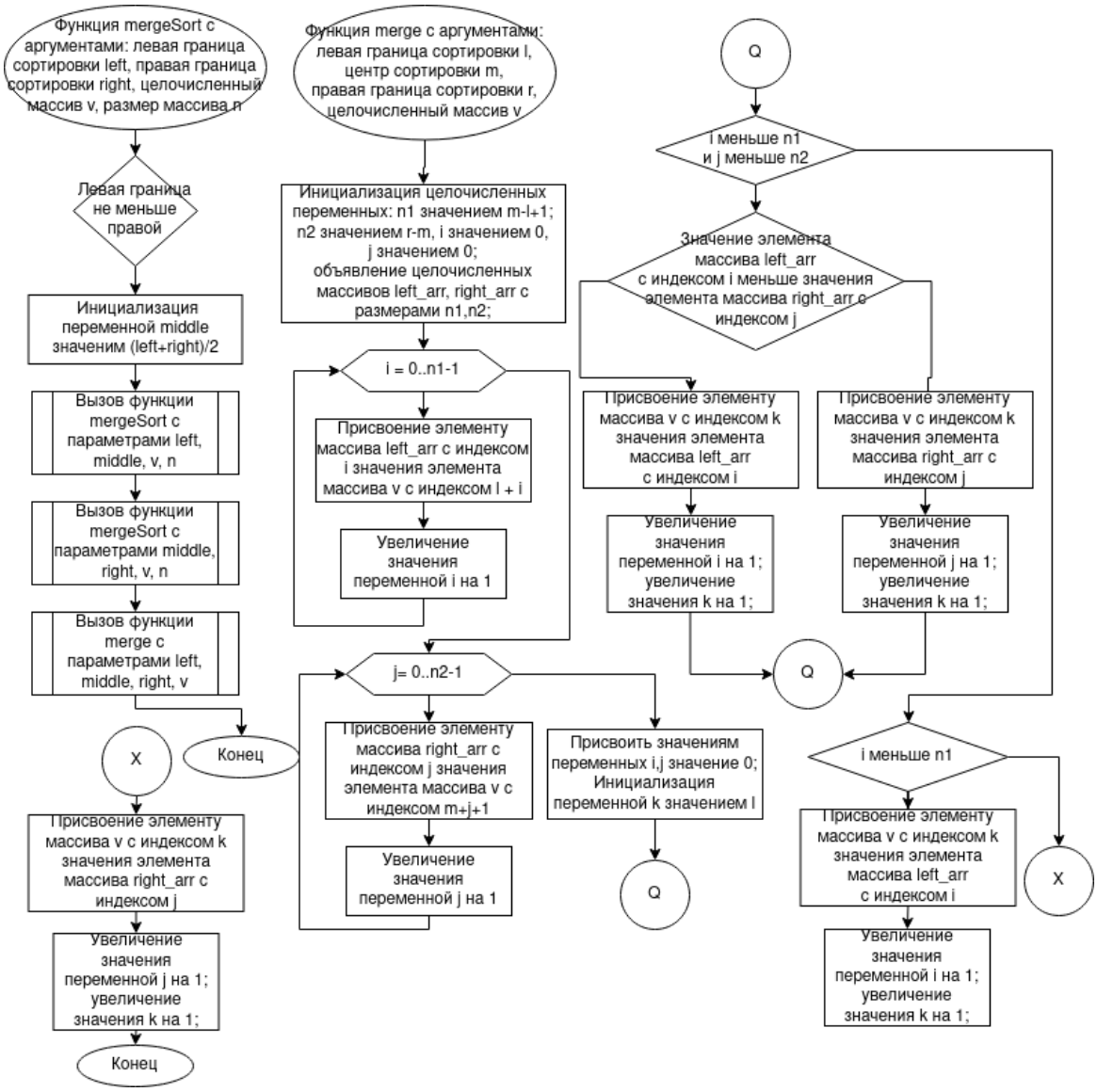

Рис. 13 - Блок-схема сортировки простым слиянием

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

void merge(int l, int m, int r, std::vector<int>& v) {
  int n1 = m - l + 1, 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++];
  }
}
```

Рис. 14 - Код сортировки слиянием

Тестирование функции сортировки

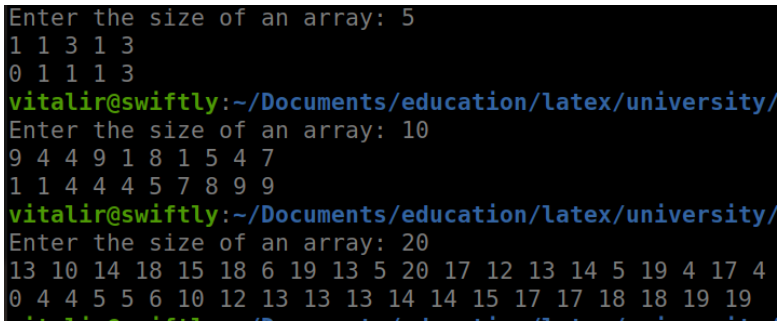

Рис. 15 - Результаты тестирования на работоспособность сортировки простым слиянием

Сводная таблица тестирования

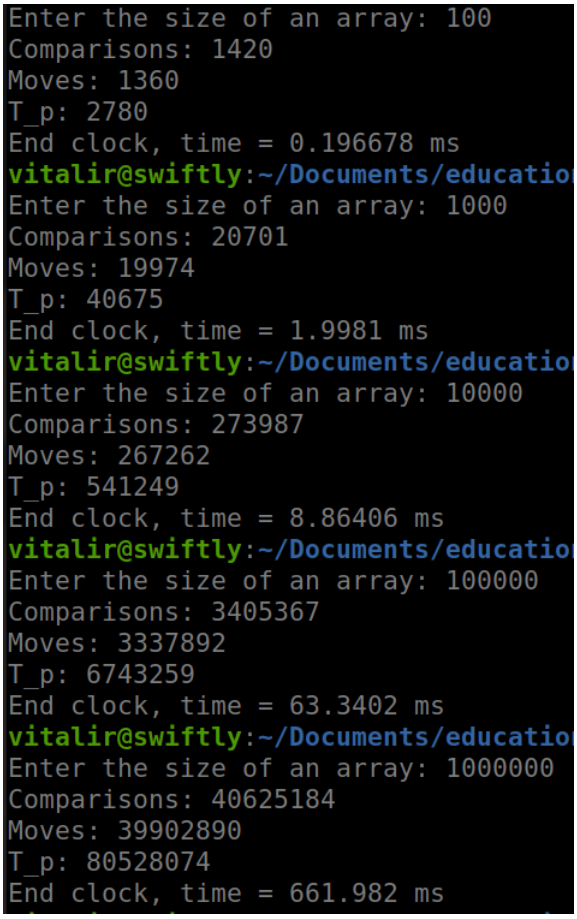

Рис. 16 - Результаты тестирования сортировки слиянием

Таблица 5 - Сводная таблица тестирования сортировки слиянием

n	$T(n)$	$T_{\text{т}} = f(C+M)$	$T_{\text{п}} = C_{\text{ф}} + M_{\text{ф}}$
100	0.196678 мс	$\Theta(n \log n)$	2780
1000	1.9981 мс		40675
10000	8.86406 мс		541249
100000	63.3402 мс		6743259
1000000	661.982 мс		80528074

Для тестирования был использован класс для подсчета времени TimeCounter; также была создана версия функции сортировки со встроенной отладкой. (рисунки 5 и 17).

```
void mergeSortLog(int left, int right, std::vector<int>& v, int64_t& comparisons, int64_t& moves) {
  ++comparisons;
  if (left >= right) { return; }
  int middle = (left + right) / 2;
  mergeSortLog(left, middle, v, comparisons, moves);
  mergeSortLog(middle+1, right, v, comparisons, moves);
  mergeLog(left, middle, right, v, comparisons, moves);
}
void mergeLog(int l, int m, int r, std::vector<int>& v, int64_t& comparisons, int64_t& moves) {
  int n1 = m - l + 1, 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]; ++moves;
  }
  for (int j = 0; j < n2; j++) {
      right_arr[j] = v[m + 1 + j]; ++moves;
  }
  int i = 0, j = 0, k = l;
  while (i < n1 && j < n2) {
    comparisons += 2;
    if (left_arr[i] < right_arr[j]) {
      v[k++] = left_arr[i++]; ++moves;
    } else {
      v[k++] = right_arr[j++]; ++moves;
    }
  }
  while (i < n1) {
      ++comparisons; ++moves;
    v[k++] = left_arr[i++];
  }
  while (j < n2) {
      ++comparisons; ++moves;
    v[k++] = right_arr[j++];
  }
}
```

Рис. 17 - 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 сор-

тировки простым слиянием эффективнее алгоритма шейкерной сортировки по временной сложности в среднем случае.

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

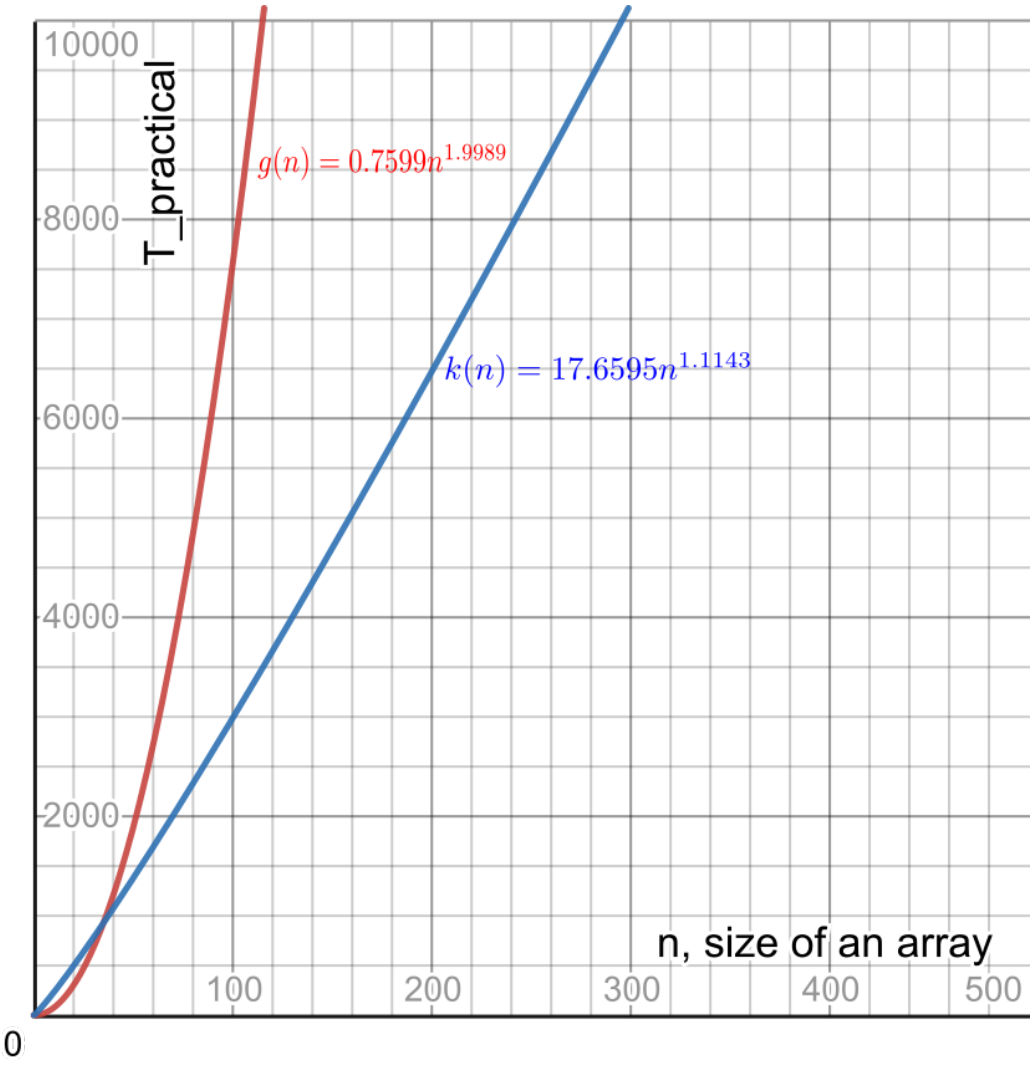

Рис. 18 - Сравнение скоростей шейкерной и простого слияния сортировок

На рис. 18 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

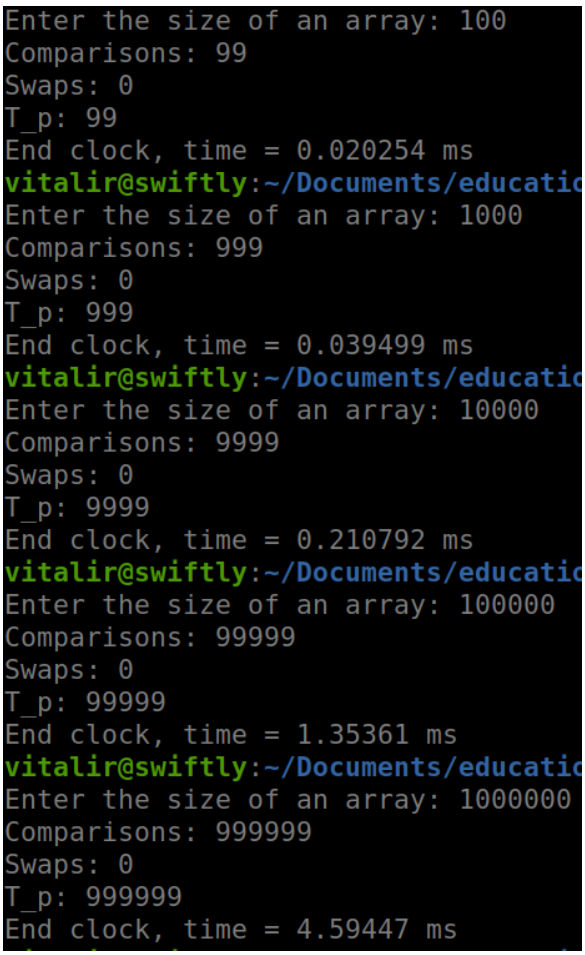

Рис. 19 - Результаты тестирования сортировки пузырьком с условием Айверсона в лучшем случае

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	0.020254 мс	$O(n)$	99
1000	0.039499 мс		999
10000	0.210792 мс		9999
100000	1.35361 мс		99999
1000000	4.59447 мс		999999

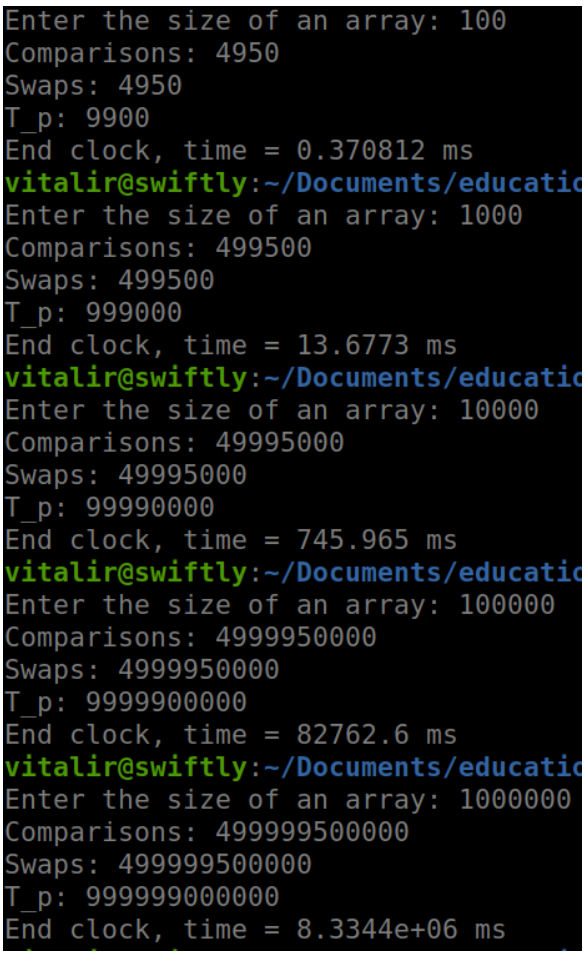

Рис. 20 - Результаты тестирования сортировки пузырьком с условием Айверсона в худшем случае

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	0.370012 мс	$O(n^2)$	9900
1000	13.6673 мс		999000
10000	745.965 мс		99990000
100000	82762.6 мс		9999900000
1000000	833440 мс		999999000000

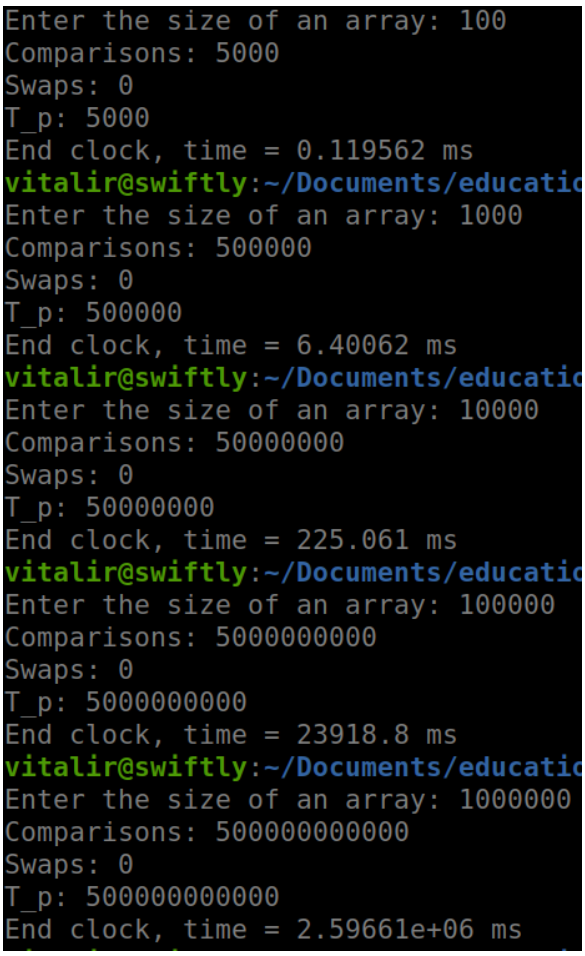

Рис. 21 - Результаты тестирования шейкерной сортировки в лучшем случае

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	0.119562 мс	$O(n^2)$	5000
1000	6.40062 мс		500000
10000	225.061 мс		50000000
100000	23918.8 мс		5000000000
1000000	2596610 мс		500000000000

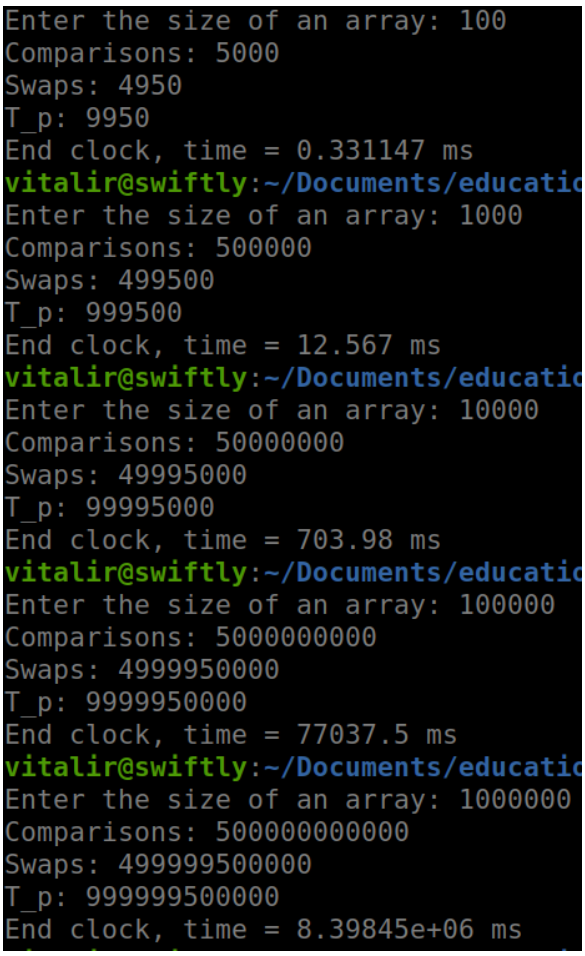

Рис. 22 - Результаты тестирования шейкерной сортировки в худшем случае

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	0.331147 мс	$O(n^2)$	9950
1000	12.567 мс		999500
10000	703.98 мс		99995000
100000	77037.5 мс		9999950000
1000000	8398450 мс		999999500000

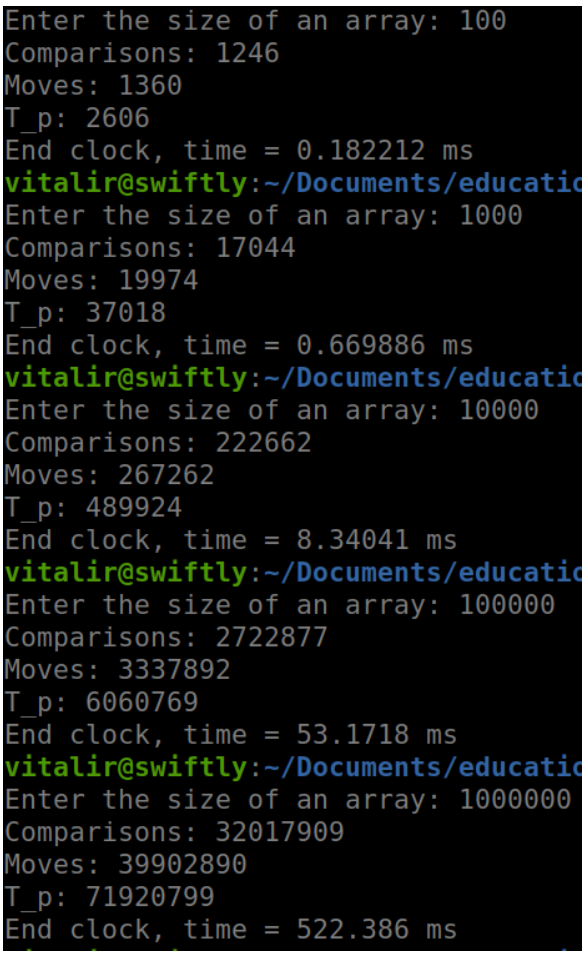

Рис. 23 - Результаты тестирования сортировки слиянием в лучшем случае

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	0.182212 мс	$\Theta(n \log n)$	2606
1000	0.669886 мс		37018
10000	8.34041 мс		489924
100000	53.1718 мс		6060769
1000000	522.386 мс		71920799

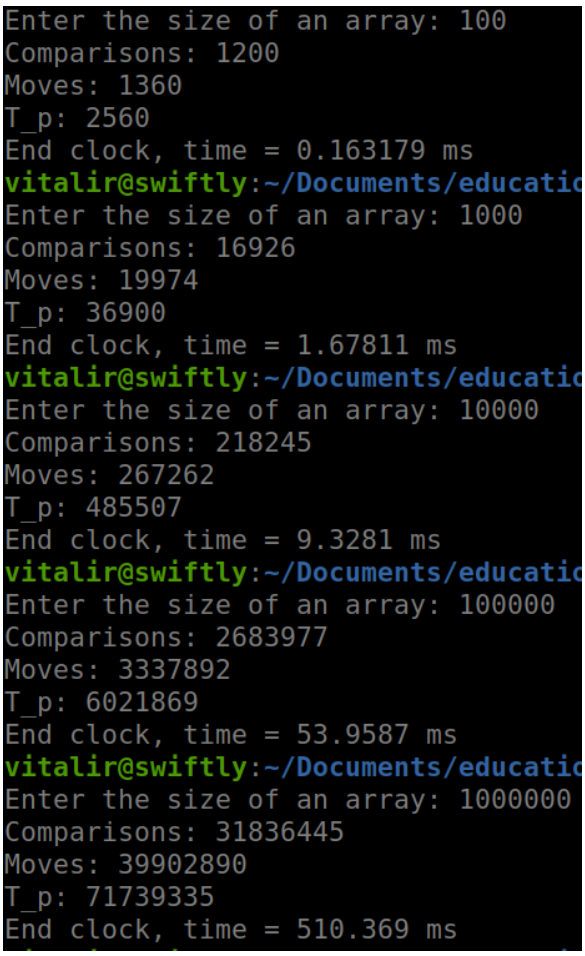

Рис. 24 - Результаты тестирования сортировки слиянием в худшем случае

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	0.163179 мс	$\Theta(n \log n)$	2560
1000	1.67811 мс		36900
10000	9.3281 мс		485507
100000	53.9587 мс		6021869
1000000	510.369 мс		71739335

Функции для тестирования сортировки в лучшем и худшем случаях представлены на рисунке 25.

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 $O(n^2)$.

```cpp
void fillAscendVector(std::vector<int>& v) {
  int i = 0;
  for (int& elem: v) {
    elem = ++i;
  }
}

void fillDescendVector(std::vector<int>& v) {
  int i = std::numeric_limits<int>::max();
  for (int& elem: v) {
    elem = --i;
  }
}
```

Рис. 25 - Код функций для заполнения массивов по возрастанию и убыванию

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)