МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического занятия 3

Тема: Эмпирический анализ алгоритмов сортировки

Дисциплина: Структуры и алгоритмы обработки данных

Выполнил студент Хвостов В. В.

Группа ИКБО-01-20

Москва 2021

Содержание

Первое задание

1.1. Постановка задачи

Оценить зависимость времени выполнения алгоритма простой сортировки на массиве, заполненном случайными числами (в соответствии с персональным вариантом).

Вариант 2. (сортировка простого обмена или пузырьком)

1.2. Алгоритм сортировки

Схема алгоритма представлена на рисунке 1.

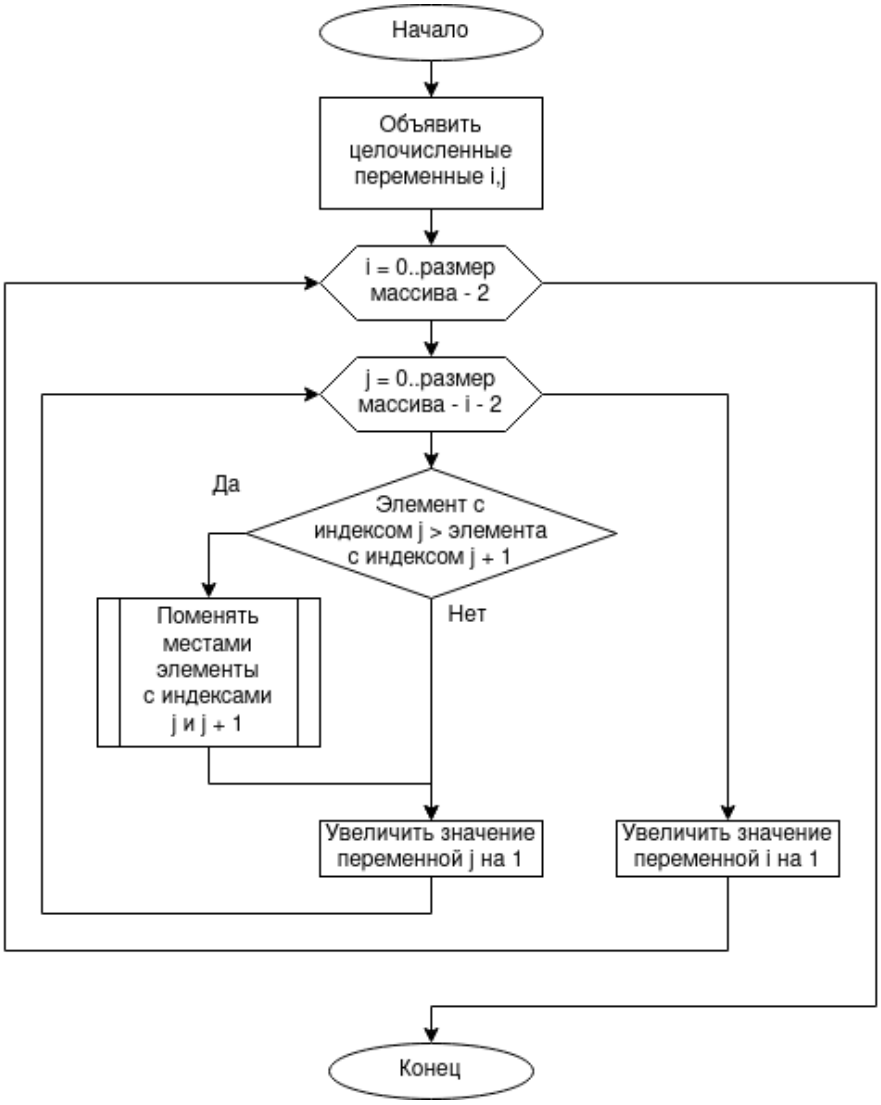

Рис. 1 - Блок-схема первого алгоритма

Результаты тестирования алгоритма на заполненном с клавиатуры массиве (рис. 2), а также на случайно заполненном массиве (рис. 3 и 4).

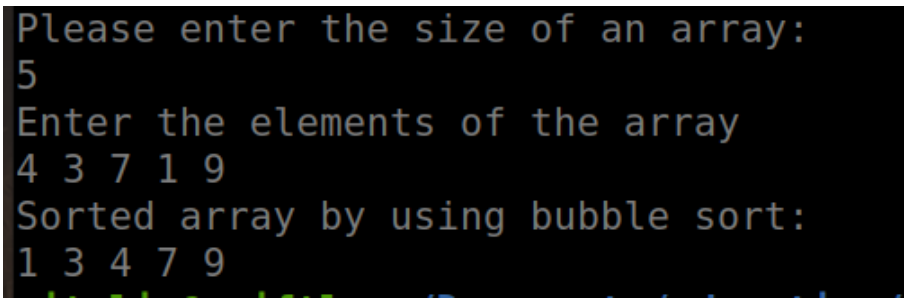

Рис. 2 - Результаты тестирования алгоритма на введенном пользователем массиве

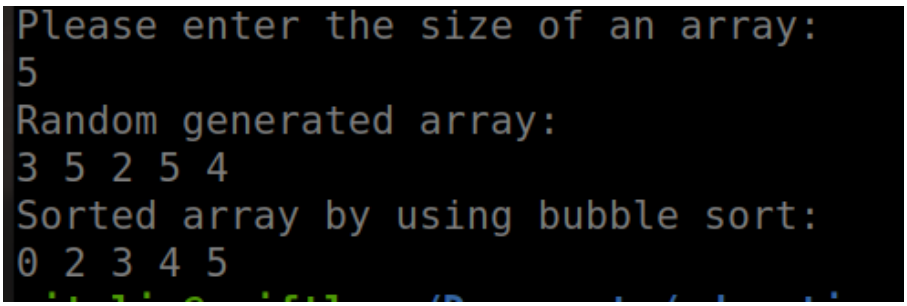

Рис. 3 - Результаты тестирования алгоритма на случайном массиве размером 5

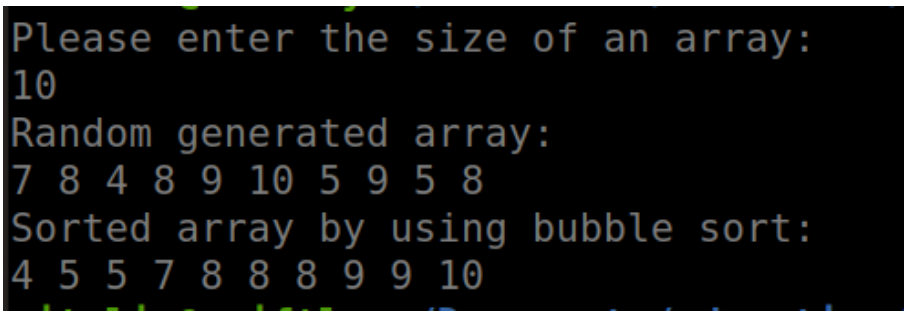

Рис. 4 - Результаты тестирования алгоритма на случайном массиве размером 10

1.3. Оценка функции роста выполнения алгоритма сортировки пузырьком

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	int n = v.size();	C1	1 раз
2	for (int i = 0; i < n-1; ++i) {	C2	n раз
3	for (int j = 0; j < n - i - 1; ++j) {	C3	n(n-1) раз
4	if (v[j] > v[j+1]) {	C4	n(n-1) - 1 раз
5	std::swap(v[j], v[j+1]);}}}	C5	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

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

После упрощения получаем

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

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.

1.4. Результаты выполнения сортировки

Представлены на рисунках 5, 6 и на таблице 2.

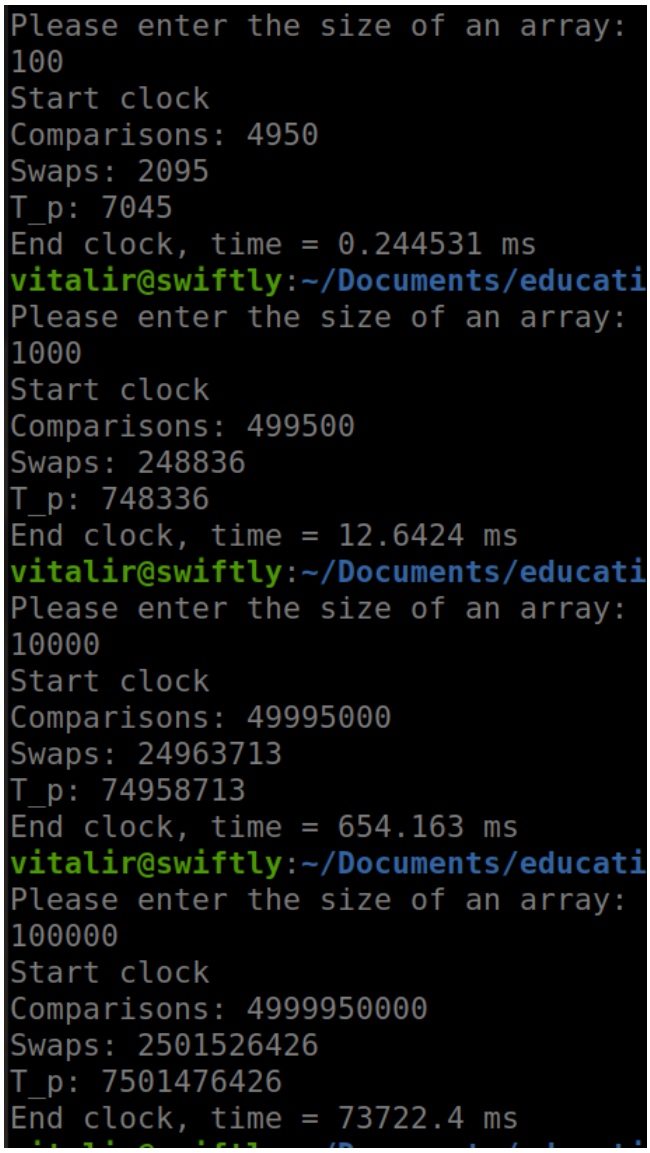

Рис. 5 - Результаты выполнения программы задания 1, 1 часть

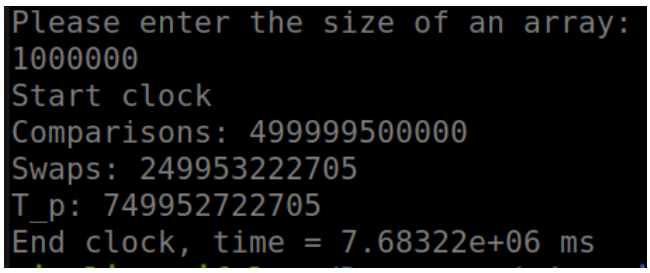

Рис. 6 - Результаты выполнения программы задания 1, 2 часть

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	0.244351 мс	$O(n^2)$	7045
1000	12.6424 мс		748336
10000	654.163 мс		74958713
100000	73722.4 мс		7501476426
1000000	7683220 мс		749952722705

1.5. Код алгоритма и основной программы

Алгоритм сортировки простым обменом представлен на рисунке 7.

```
void bubbleSortVector(std::vector<int>& v) {
  int n = v.size();
  for (int i = 0; i < n - 1; ++i) {
    for (int j = 0; j < n - i - 1; ++j) {
      if (v[j] > v[j+1]) {
        std::swap(v[j], v[j+1]);
      }
    }
  }
}
```

Рис. 7 - Код функции сортировка простым обменом

Для отладки были реализованы функции генерации случайного массива, класс для замерения времени и отдельная функция сортировки простым обменом со встроенным подсчетом практической сложности (представлены, соответственно, на рисунках 8, 9, 10).

```
void generateRandomVector(std::vector<int>& v, int upper_bound) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(1, upper_bound);
    for (int& elem: v) {
      elem = distrib(gen);
    }
}
```

Рис. 8 - Код функции генерации случайного массива

```
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. 9 - Код класса для замера времени выполнения

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

Рис. 10 - Код функции сортировки простым обменом с подсчетом операций

Главная функция программы представлена на рисунке 11.

```cpp
int main() {
  std::ios_base::sync_with_stdio(false);
  std::cout << "Please enter the size of an array:\n";
  int n;
  std::cin >> n;
  std::vector<int> arr(n);
  generateRandomVector(arr, n);
//  std::cout << "Random generated array:\n" << arr << '\n';
  {
    TimeCounter tc;
    bubbleSortVectorLog(arr);
  }
 // std::cout << "Sorted array by using bubble sort:\n" << arr << '\n';
  return 0;
}
```

Рис. 11 - Главная функция программы задания 1

1.6. Графики зависимостей теоретической и практической сложности

Для вычисления зависимости практической вычислительной сложности алгоритма от размера n массива можно воспользоваться калькулятором методов регрессии для аппроксимации функции одной переменной. Наименьшая ошибка аппроксимации приходится на степенную модель регрессии с показателем, приблизительно равным 2.

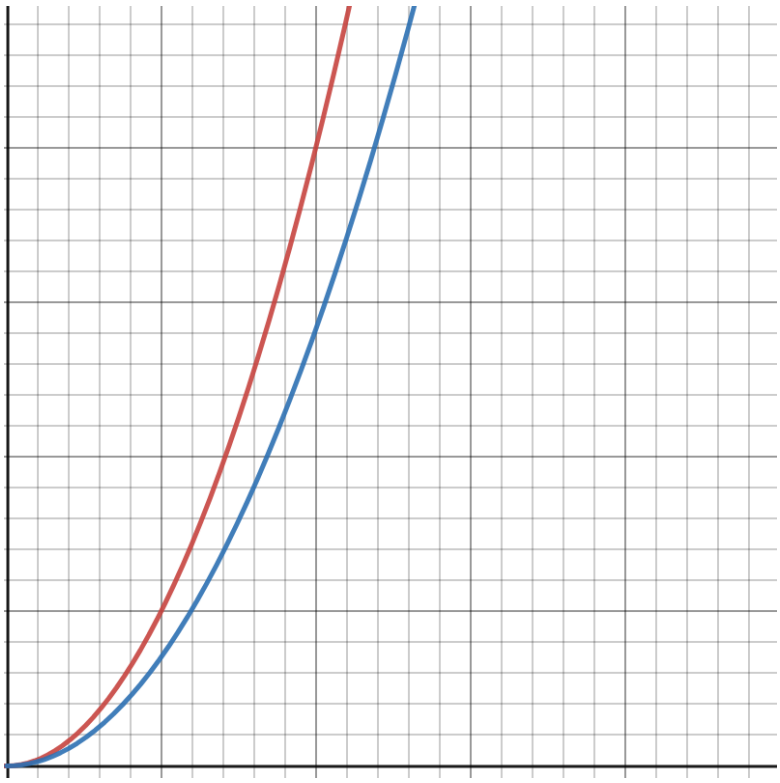

Рис. 12 - График зависимости теоретической и практической сложности алгоритма сортировки простым обменом

На рисунке 12 показаны графики теоретической зависимости $f(n) = n^2$ (красным) и практической зависимости $g(n) = 0.7035x^{2.0055}$ (синим) вычислительной сложности алгоритма от размера массива.

1.7. Анализ результатов

Из графика видно, что эмпирически полученные данные не противоречат теоретическим расчетам вычислительной сложности алгоритма: сложность алгоритма имеет квадратичную зависимость от размера входных данных.

Задание 2

2.1. Постановка задачи

Оценить вычислительную сложность алгоритма простой сортировки (в соответствии с персональным вариантом) в наилучшем и наихудшем случаях.

Вариант 2.

2.2. Результаты тестирования на массиве, отсортированном строго по возрастанию

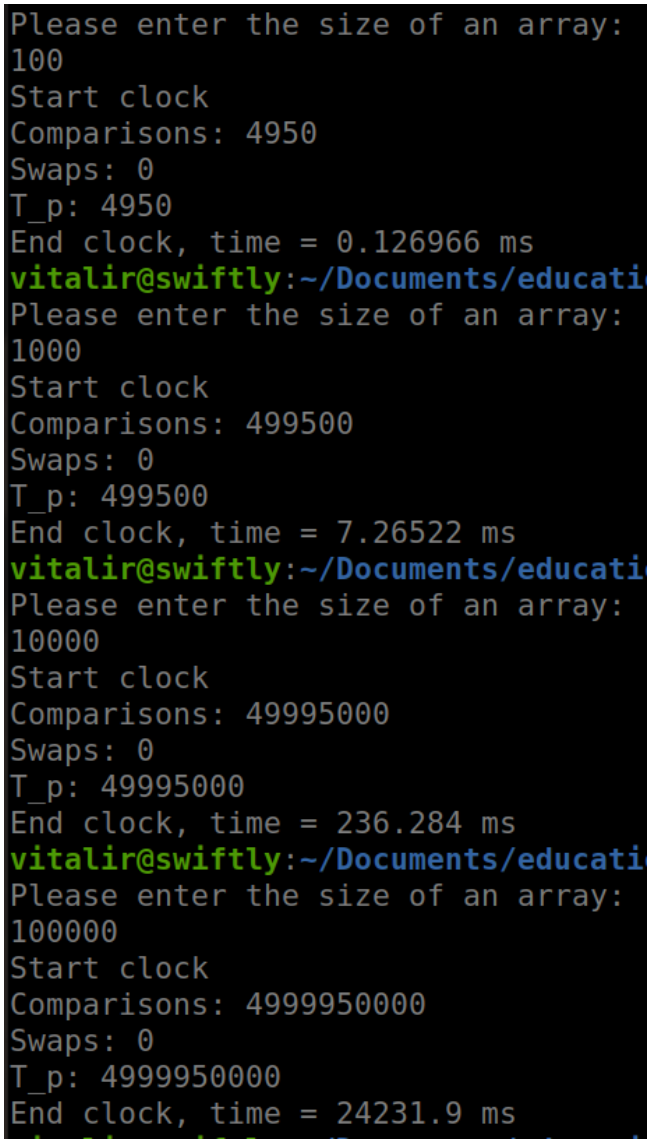

Рис. 13 - Результаты выполнения программы задания 2, лучший случай, 1 часть

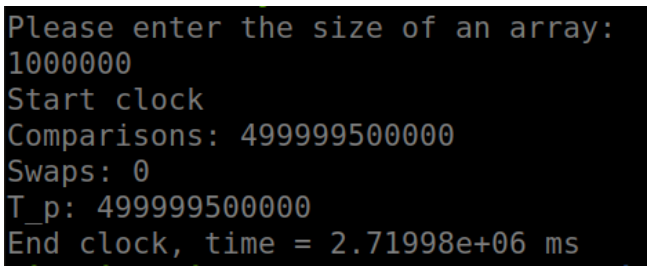

Рис. 14 - Результаты выполнения программы задания 2, лучший случай, 2 часть

Таблица 3 - 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	0.126966 мс	$O(n^2)$	4950
1000	7.26552 мс		499500
10000	236.284 мс		49995000
100000	24231.9 мс		4999950000
1000000	7683220 мс		749952722705

Отсюда $T(n) = \frac{n^2-n}{2}$.

2.3. Результаты тестирования на массиве, отсортированном строго по убыванию

```
Please enter the size of an array:
100
Start clock
Comparisons: 4950
Swaps: 4950
T_p: 9900
End clock, time = 0.324369 ms
vitalir@swiftly:~/Documents/educati
Please enter the size of an array:
1000
Start clock
Comparisons: 499500
Swaps: 499500
T_p: 999000
End clock, time = 11.9698 ms
vitalir@swiftly:~/Documents/educati
Please enter the size of an array:
10000
Start clock
Comparisons: 49995000
Swaps: 49995000
T_p: 99990000
End clock, time = 720.841 ms
vitalir@swiftly:~/Documents/educati
Please enter the size of an array:
100000
Start clock
Comparisons: 4999950000
Swaps: 4999950000
T_p: 9999900000
End clock, time = 77201.8 ms
```

Рис. 15 - Результаты выполнения программы задания 2, худший случай, 1 часть

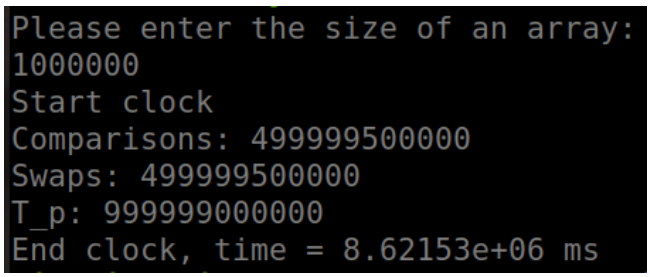

Рис. 16 - Результаты выполнения программы задания 2, худший случай, 2 часть

Таблица 4 - 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	0.324369 мс	$O(n^2)$	9900
1000	11.9698 мс		999000
10000	720.841 мс		99990000
100000	77201.8 мс		9999900000
1000000	8621530 мс		999999000000

Отсюда $T(n) = n^2 - n$.

2.4. Код тестирующей программы

Для тестирования лучших и худших случаев сортировки были созданы функции для заполнения массива в порядке возрастания и убывания, а также немного изменена главная функция программы (рис. 17).

```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;
  }
}


int main() {
  std::ios_base::sync_with_stdio(false);
  std::cout << "Please enter the size of an array:\n";
  int n;
  std::cin >> n;
  std::vector<int> arr(n);
 // fillAscendVector(arr); // 1st case
  fillDescendVector(arr); // 2nd case
  {
    TimeCounter tc;
    bubbleSortVectorLog(arr);
  }
  return 0;
}
```

Рис. 17 - Код программы задания 2

2.5. График зависимости теоретической и практической вычислительной сложности алгоритма для трех рассмотренных случаев

На рис. 18 представлены графики зависимостей теоретической (красный) и практической (синий для табл. 1, зеленый для табл. 3 , темно-фиолетовый для табл. 4) вычислительных сложностей алгоритма. При этом квадратичная зависимость для теоретической сложности сохраняется во всех рассмотренных случаев, исходя из выведенных формул функции роста времени для худшего и лучшего случаев. Графики практических сложностей для упорядоченного массива являются квадратичными зависимостями с формулами $T(n) = \frac{n^2-n}{2}$ для лучшего случая и $T(n) = n^2 - n$ для худшего случая.

2.6. Емкостная сложность алгоритма от n

Алгоритм сортировки методом простого обмена имеет линейную емкостную сложность $O(n)$, т.к. в процессе сортировки обрабатывается только исходный массив размера n.

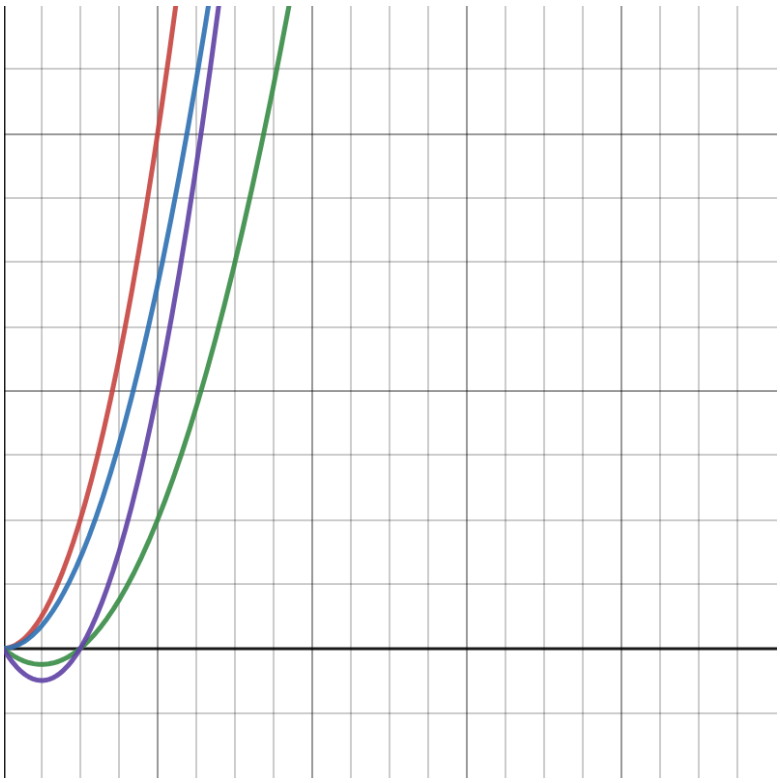

Рис. 18 - Графики зависимостей вычислительной сложности сортировки пузырьком от n

2.7. Анализ результатов

Установленная эмпирическим путем зависимость вычислительной сложности алгоритма подтверждает найденную ранее теоретическую сложность. Для лучшего случая в алгоритме количество операций сравнения при увеличении размера массива в 10 раз увеличивается в 100, а операции перестановки не выполняются совсем. Для худшего же случая сохраняется квадратичная зависимость критических операций от размера массива.

Задание 3

3.1. Постановка задачи

Выполнить разработку алгоритма и программную реализацию алгоритма задания 3. Сформировать таблицу в соответствии с форматом таблицы 2 на тех же массивах, что и в задании 1. Выполнить сравнительный анализ полученных результатов контрольных прогонов и построением соответствующих графиков. Определить емкостную сложность алгоритма от n.

Вариант 2 (сортировка простыми вставками)

3.2. Алгоритм сортировки

Схема алгоритма представлена на рисунке 19.

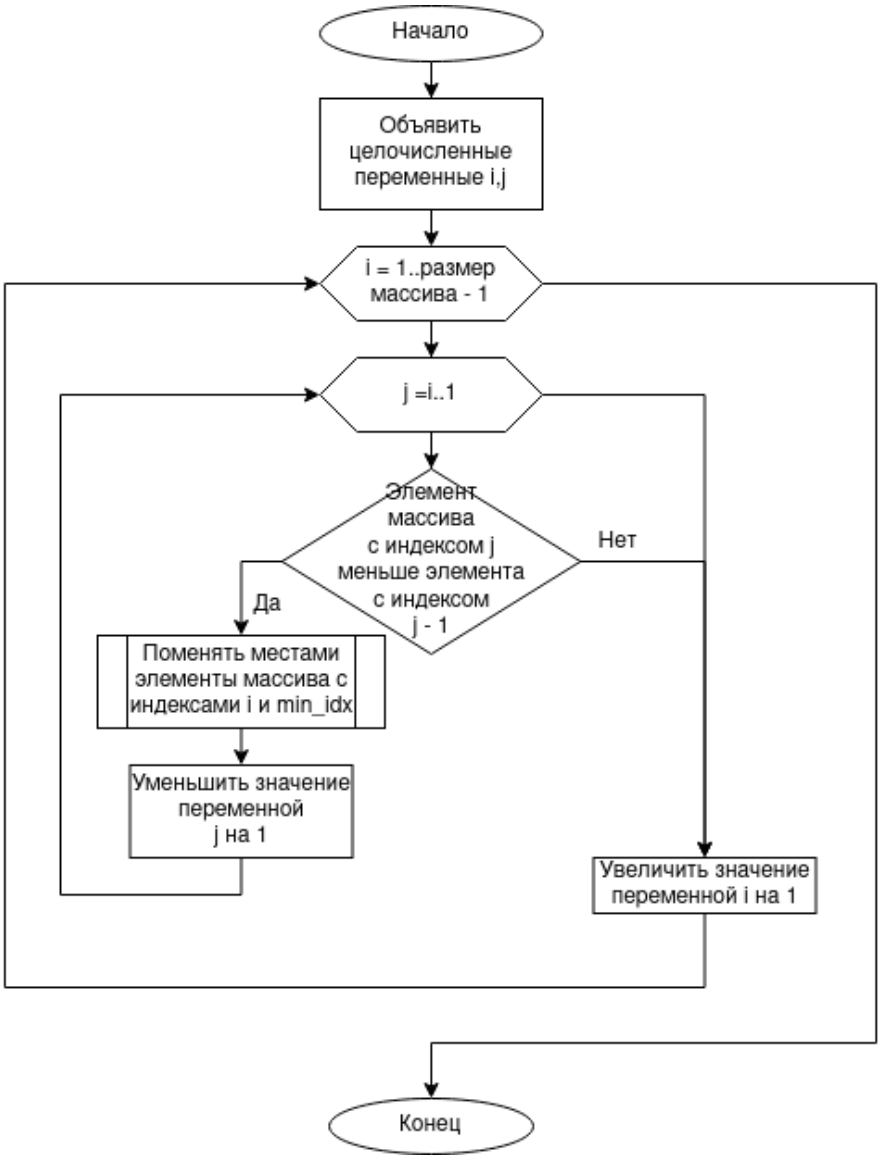

Рис. 19 - Блок-схема второго алгоритма

3.3. Оценка функции роста выполнения алгоритма сортировки вставками

Определим теоретическую сложность алгоритма при помощи таблицы операторов.

Таблица 5 - Подсчет количества операторов в алгоритме сортировки простых вставок

Номер оператора	Оператор	Время выполнения одного оператора	Кол-во выполнений оператора в строке
1	for (int i = 1; i < n; ++i) {	C1	n раз
2	for (int j = i; j > 0 && v[j] < v[j-1]; --j) {	C2	n(n-1) раз
3	std::swap(v[j], v[j-1]);}}	C3	n(n-1)-1 раз

Из таблицы 5 получим функцию роста выполнения алгоритма сортировки простыми вставками. Пусть $T(n)$ - время выполнения алгоритма, зависящее от n. Тогда

$$T(n) = C_1 \cdot n + C_2 \cdot (n^2 - n) + C_3 \cdot (n^2 - n - 1).$$

После упрощения получаем

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

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.

3.4. Результаты выполнения сортировки

Для произвольного массива

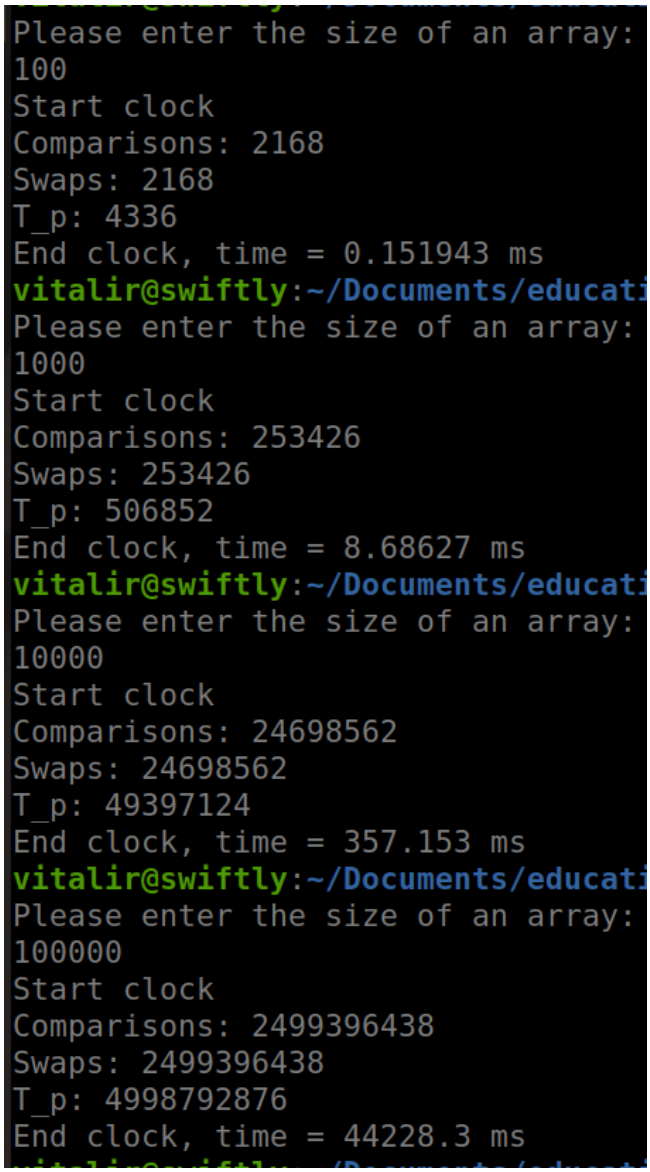

Рис. 20 - Результаты выполнения программы задания 3, случайный массив, 1 часть

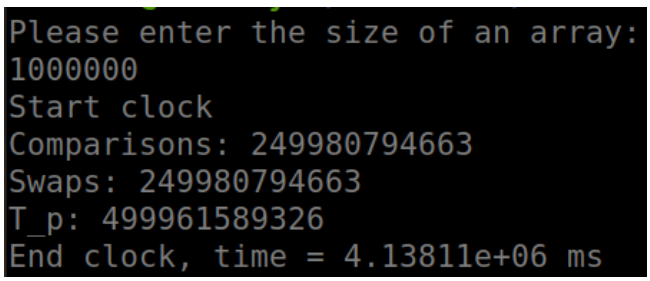

Рис. 21 - Результаты выполнения программы задания 3, случайный массив, 2 часть

Таблица 6 - Сводная таблица тестирования сортировки вставками

n	$T(n)$	$T_{\text{т}} = f(C+M)$	$T_{\text{п}} = C_{\text{ф}} + M_{\text{ф}}$
100	0.151943 мс	$O(n^2)$	4336
1000	8.68627 мс		506852
10000	357.153 мс		49397124
100000	44228.3 мс		4998792876
1000000	4138110 мс		499961589326

3.5. Код программы

```
void insertionSortVectorLog(std::vector<int>& v) {
  int64_t comparisons = 0, swaps = 0;
  for (size_t i = 1; i < v.size(); ++i) {
    for (size_t j = i; j > 0 && v[j] < v[j-1]; --j) {
      ++comparisons, ++swaps;
      std::swap(v[j], v[j-1]);
    }
  }
  std::clog << "Comparisons: " << comparisons;
  std::clog << "\nSwaps: " << swaps << '\n';
  std::clog << "T_p: " << comparisons + swaps << '\n';
}

int main() {
  std::ios_base::sync_with_stdio(false);
  std::cout << "Please enter the size of an array:\n";
  int n;
  std::cin >> n;
  std::vector<int> arr(n);
  fillDescendVector(arr);
  {
    TimeCounter tc;
    bubbleSortVectorLog(arr);
  }
  return 0;
}
```

Рис. 22 - Код программы тестирования сортировки вставками

Для тестирования программы был разработан код функции сортировки простыми вставками с отладкой и немного изменен код главной функции (см. рис. 22).

3.6. График зависимости вычислительных сложностей для 2 случаев

На рис. 23 отображена зависимость теоретической (синий) и практической (красный для алгоритма простого обмена, зеленый для алгоритма простой вставки с формулой $y = 0.4361^{2.0118}$). вычислительных сложностей. Теоретическая сложность у обоих алгоритмов квадратично зависит от размера массива.

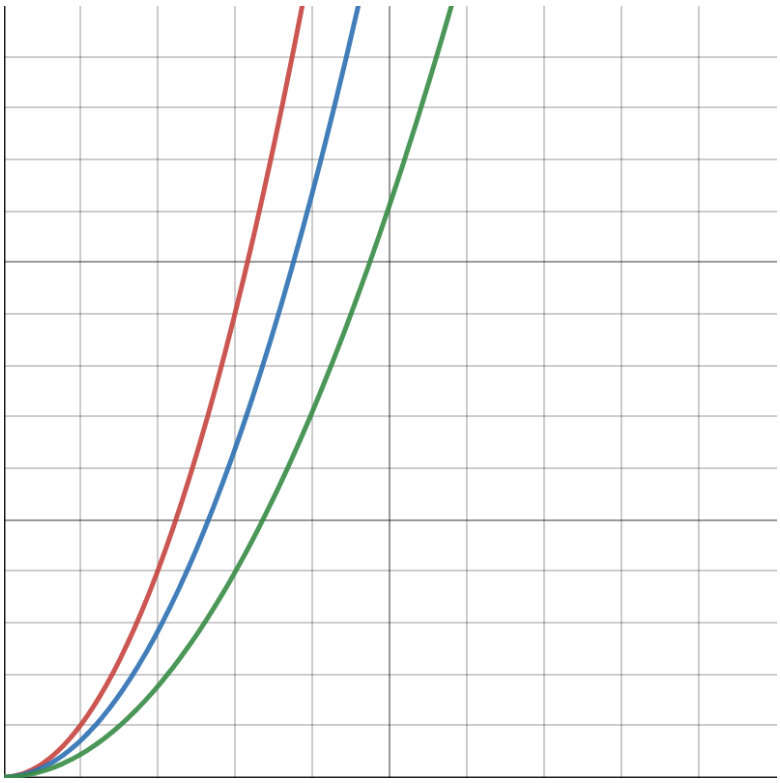

Рис. 23 - Сравнение вычислительных сложностей сортировок пузырьком и вставкой

3.7. Определение эффективности алгоритма

Эффективность алгоритма — это свойство алгоритма, которое связано с вычислительными ресурсами, используемыми алгоритмом. Алгоритм считается эффективным, если потребляемый им ресурс (или стоимость ресурса) являются наиболее оптимальными. Оценка эффективности алгоритма состоит в определении времени его выполнения и объема потребляемой им памяти. Алгоритм считается эффективнее другого, если время его работы в наихудшем случае имеет более низкий порядок роста.

3.8. Сравнение эффективности двух алгоритмов из заданий 1 и 3

Сравнивая алгоритмы в среднем, лучшем и худшем случаях, можно сделать вывыд, что сортировка вставками работает быстрее сортировки пузырьком (по графикам практических зависимостей, а также в лучшем случае сортировка вставками имеет линейную зависимость, в отличии от сортировки пузырьком).

Ответы на дополнительные вопросы

Вопрос 7

Применим условие Айверсона к алгоритму сортировки простого обмена, для этого изменив код функции (см рис. 24).

```
void bubbleSortVectorAivLog(std::vector<int>& v) {
  bool isSorted = false;
  int n = v.size();
  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';
}

int main() {
  std::ios_base::sync_with_stdio(false);
  std::cout << "Please enter the size of an array:\n";
  int n;
  std::cin >> n;
  std::vector<int> arr(n);
  generateRandomVector(arr, n);
  {
    TimeCounter tc;
    bubbleSortVectorAivLog(arr);
  }
  return 0;
}
```

Рис. 24 - Код измененного фрагмента программы сортировки пузырьком с использованием условия Айверсона

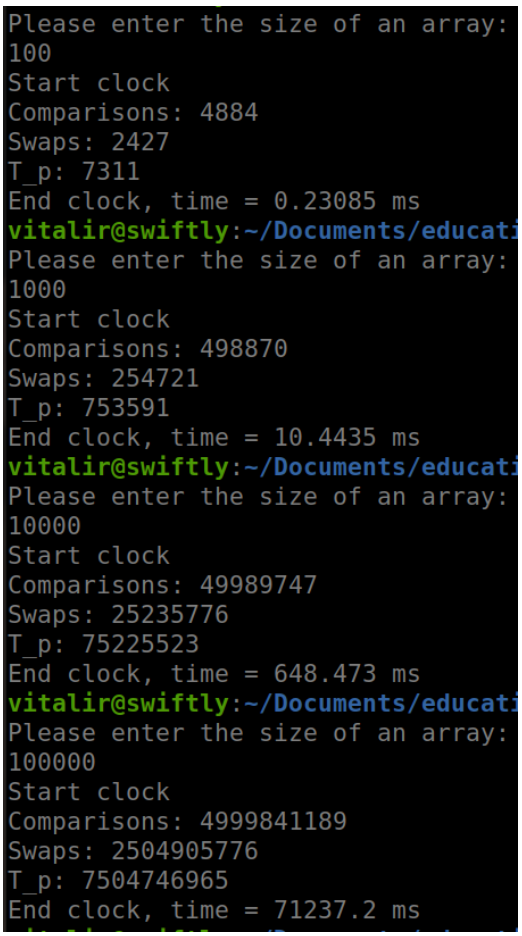

Рис. 25 - Результаты выполнения программы вопроса 7, случайный массив, 1 часть

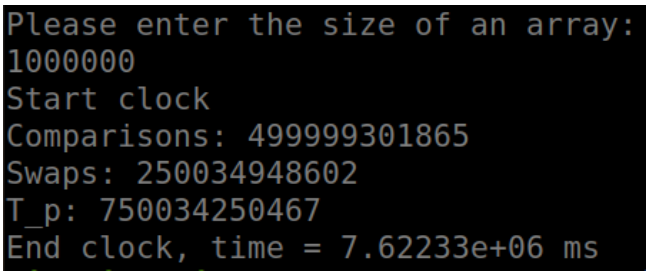

Рис. 26 - Результаты выполнения программы вопроса 7, случайный массив, 2 часть

Таблица 7 - Сводная таблица тестирования сортировки обмена с условием Айверсона

n	$T(n)$	$T_{\text{т}} = f(C+M)$	$T_{\text{п}} = C_{\text{ф}} + M_{\text{ф}}$
100	0.23085 мс	$O(n^2)$	7311
1000	10.4435 мс		753591
10000	648.473 мс		75225523
100000	71237.2 мс		7504746965
1000000	7622330 мс		750034250467

Сравнивая таблицы 1 и 7, можно сделать вывод, что условие Айверсона особо не влияет на расчет эффективности алгоритма сортировки простого обмена.

Вопрос 8

Сортировка массива [5 6 1 2 3] с данными шагами является сортировкой простой вставки, суть которой заключается в переборе массива, при котором на каждом шаге элемент перемещается на свое конечное место, начиная с начала массива.

Вопрос 9

В лучшем случае алгоритм имеет порядок роста времени $O(n)$, в худшем $O(n^2)$ операций сравнения и перестановки. Емкостная сложность составляет $O(n)$, так как используется только один массив данных.

Выводы

В ходе выполнения практической работы были изучены методы оценки времени выполнения алгоритмов простых сортировок (сортировки пузырьком и простыми вставками) на случайно заполненных массивах. Данные алгоритмы были описаны на языке блок-схем, реализованы на языке C++, была определена теоретическая сложность в наилучшем и наихудшем случаях, выполнен прогон алгоритмов на тестовых данных для среднего, наилучшего и наихудшего случаев, определена практическая сложность алгоритмов. Из результатов тестирования, теоретические предположения совпали с полученными эмпирически данными. Основываясь на этих данных, алгоритм пузырьковой сортировки почти в два раза менее эффективнее по времени чем алгоритм сортировки вставками.

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. Difference between bubblesort & insertion sort // Pediaa [Электронный ресурс]. URL: https://pediaa.com/what-is-the-difference-between-bubble-sort-and-insertion-sort/ (Дата обращения: 05.04.2021)

4. Курс Algorithms, part 1 // Coursera [Электронный ресурс]. URL: https://www.coursera.org/learn/algorithms-part1 (Дата обращения: 05.04.2021)