МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Отчет по выполнению практического задания № 2
Тема:
«Эмпирический анализ сложности простых алгоритмов сортировки»
Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнил студент: Зернов Н.А.

Группа: ИКБО-74-23

Москва – 2024

СОДЕРЖАНИЕ

1 ЦЕЛЬ

Актуализация знаний и приобретение практических умений по эмпирическому определению вычислительной сложности алгоритмов.

2 ЗАДАНИЕ №1

2.1 Формулировка задачи

Оценить эмпирически вычислительную сложность алгоритма простой сортировки на массиве, заполненном случайными числами (средний случай).

1. Составить функцию простой сортировки одномерного целочисленного массива A[n], используя алгоритм простой вставки. Провести тестирование программы на исходном массиве n=10.

2. Используя теоретический подход, определить для алгоритма:

a. Что будет ситуациями лучшего, среднего и худшего случаев.

b. Функции роста времени работы алгоритма от объёма входа для лучшего и худшего случаев.

3. Провести контрольные прогоны программы массивов случайных чисел при n = 100, 1000, 10000, 100000 и 1000000 элементов с вычислением времени выполнения T(n) – (в миллисекундах/секундах). Полученные результаты свести в сводную таблицу 2.

4. Провести эмпирическую оценку вычислительной сложности алгоритма, для чего предусмотреть в программе подсчет фактического количества критических операций $T_п$ как сумму сравнений $C_п$ и перемещений $M_п$. Полученные результаты вставить в сводную таблицу 2.

5. Построить график функции роста $T_п$ этого алгоритма от размера массива n.

6. Определить ёмкостную сложность алгоритма.

7. Сделать вывод об эмпирической вычислительной сложности алгоритма на основе скорости роста функции роста.

2.2 Математическая модель решения алгоритма

2.2.1 Описание выполнения и блок-схема алгоритма сортировки простыми вставками

Последовательное добавление элементов из неотсортированной к уже отсортированной части массива с сохранением в ней упорядоченности.

Эта сортировка обладает естественным поведением, т.е. алгоритм работает быстрее для частично упорядоченного массива. Алгоритм устойчив – элементы с одинаковыми ключами не переставляются.

Отсортированной частью массива по умолчанию считаем начальный элемент. На первом шаге следующий второй элемент (он же первый элемент неотсортированной части) сравнивается с первым (крайним справа элементом отсортированной части).

Если он больше первого, то остаётся на своём месте и просто включается в отсортированную часть, а остальная (неотсортированная) часть массива остаётся без изменений.

Если же он меньше первого, то последовательно сравнивается с элементами в упорядоченной части (начиная с наибольшего – крайнего справа), пока не встретится элемент меньший. После чего происходит вставка на место первого, большего его в отсортированной части, со сдвигом всех элементов правее в отсортированной части на одну позицию вправо.

Этот процесс повторяется аналогично для всех последующих элементов неотсортированной части исходного массива.

Реализация данного описания выполнения алгоритма представлена в виде блок-схемы (рис.1).

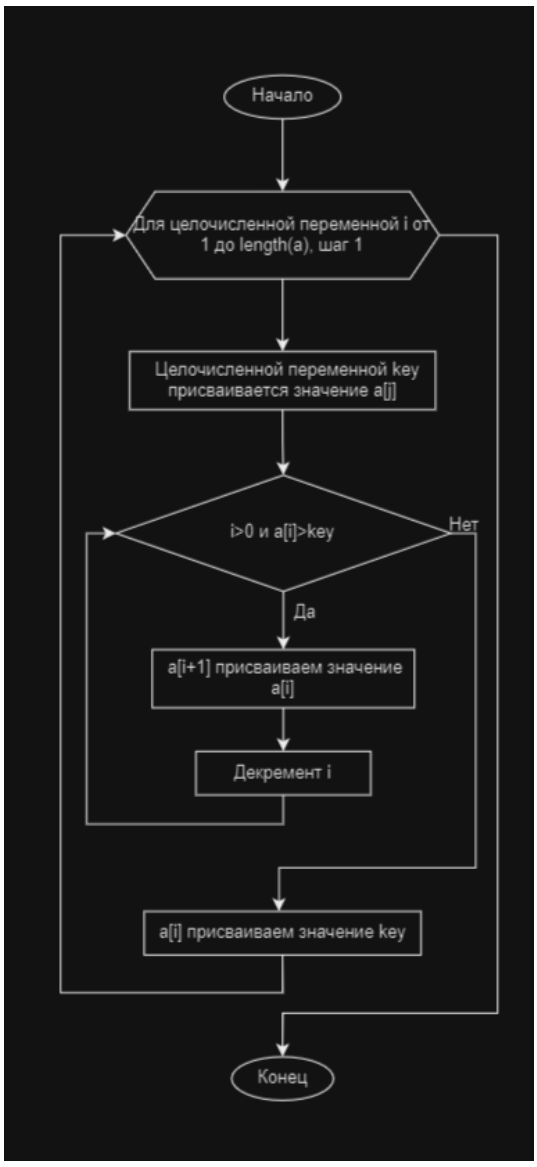


Рисунок 1 – Блок-схема алгоритма сортировки простыми вставками

2.2.2 Доказательство корректности циклов алгоритма сортировки простыми вставками

Инвариант для внешнего цикла: значение переменной i всегда меньше length(a).

Инвариант для внутреннего цикла: значение переменной i всегда больше 0 и a[i] больше key.

Докажем конечность циклов. Внешний цикл for проходит через все элементы массива начиная со второго. На каждом повторении внешнего цикла текущий элемент key перемещается влево в отсортированную часть массива до

тех пор, пока не встретит элемент, меньший или равный ему, или не дойдет до начала массива. Внутренний цикл while сдвигает все элементы отсортированной части массива, которые больше key, вправо, пока не найдет место для x или не дойдет до начала массива. Затем key помещается на свое место. Таким образом, циклы не могут быть бесконечны.

Из доказательства можно сделать вывод, что все циклы данного алгоритма корректны.

2.2.3 Определение ситуаций лучшего, среднего и худшего случая и функции роста времени работы алгоритма сортировки простыми вставками

Таблица 1-Псевдокод и анализ алгоритма сортировки вставками

№	Алгоритм, записанный на псевдокоде	Количество выполнений оператора
1	InsertionSort(a,n){	
2	for i←2 to length(a) do	n
3	key←a[j]	n-1
4	while i>0 и a[i]>key do	n-1
5	a[i+1]←a[i]	$\sum^{n}_{j=2}(t_j - 1)$
6	i←i-1	$\sum^{n}_{j=2}(t_j - 1)$
7	od	
8	a[i]←key	n-1
9	od	
10	}	

a. Лучший случай - массив уже отсортирован. В этом случае количество операций сравнения и перемещения будет минимальным и будет составлять O(n).

Средний случай - массив заполнен случайными числами. В этом случае алгоритм будет иметь сложность $O(n^2)$.

Худший случай - массив отсортирован в обратном порядке. В этом случае количество операций также будет $O(n^2)$.

b. Функции роста времени:

Лучший случай: O(n).

Худший случай: $O(n^2)$.

Для данного метода сортировки, время исполнения в худшем случае увеличивается квадратично с ростом размера входного массива. Следовательно, можно использовать квадратичную функцию для описания функции роста данного сортировочного метода. Время исполнения в лучшем случае увеличивается линейно с ростом размера входного массива.

Ёмкостная сложность алгоритма будет равна O(1).

2.3 Реализация алгоритма на языке C++, проведение тестирования и построение графика

2.3.1 Реализация алгоритма сортировки простыми вставками на языке C++

Реализуем данный алгоритм на языке C++(рис.2,3). Для реализации понадобятся такие библиотеки, как iostream, vector и random, chrono. Iostream — это заголовочный файл с классами, функциями и переменными для организации потока ввода и вывода в языке программирования C++. Vector — это шаблон класса для контейнеров последовательности. Вектор хранит элементы заданного типа в линейном расположении и обеспечивает быстрый случайный доступ к любому элементу. Random - позволяет генерировать случайные числа в диапазоне. В данной программе задан диапазон от 1 до 10. Chrono позволяет реализовать такие концепции, как: интервалы времени, моменты времени, таймеры. Для подсчёта количество операций присваивания или сравнения введём переменную operations_counter_counter, которая представляет собой целое число в диапазоне от -9223372036854775807 до 9223372036854775807 и занимает 8 байта в памяти.

```cpp
#include <iostream>
#include <random>
#include <vector>
#include <chrono>
using namespace std;

long long int insertion_sort(vector<int>& arr) {
    long long int operations_counter = 0;
    for (long long int i = 0; i < arr.size(); ++i) {
        long long int tmp = arr[i];
        long long int j = i;
        while (j > 0 && arr[j - 1] > tmp) {
            arr[j] = arr[j - 1];
            --j;
            operations_counter += 3; // Сравнение в цикле, присвоение в теле цикла, декремент
        }
        arr[j] = tmp;
        operations_counter += 5; // Сравние в цикле при входе, 3 присвоение, 2 из которых вышее цикла while, сравнение в цикле при выходе
    }
    ++operations_counter; // Сравнение при выходе из цикла for
    return operations_counter;
}

```

Рисунок 2 – Программа алгоритма сортировки простыми вставками

```cpp
int main() {
    setlocale(LC_ALL, "Rus");
    long long int n;
    cout << "Введите размер массива: ";
    cin >> n;
    vector<int> array;

    cout << "Выберите тип заполнения массива" << endl;
    cout << "a - рандом" << endl;
    cout << "b - с одним заданым значением" << endl;
    cout << "c - ручной ввод массива" << endl;
    char ty;
    cin >> ty;
    switch (ty) { //Конструкция switch case default для выбор типа заполения массива
    case 'a': {
        mt19937 gen(random_device{}()); // mt19937 gen - генератор случайных чисел
        uniform_int_distribution<int> dist(-10, 10); // Инструмент, позволяющий генерировать случайное целое число в заданном диапазоне

        for (long long int i = 0; i < n; i++)
            array.push_back(dist(gen));
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
                cout << array[i] << " ";
        cout << endl;
    }
            break;
    case 'b': {
        cout << "Введите значение: ";
        long long int x;
        cin >> x;
        for (long long int i = 0; i < n; i++)
            array.push_back(x);
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
            cout << array[i] << " ";
        cout << endl;
    }
            break;
```

Рисунок 3 – Функция main для алгоритма сортировки простыми вставками

```
60      case 'c': {
61          // Ручной ввдо массива
62          for (long long int i = 0; i < n; i++) {
63              long long int tmp;
64              cin >> tmp;
65              array.push_back(tmp);
66          }
67          for (int i = 0; i < n; i++) // Вывод сгенерированого массива
68              cout << array[i] << " ";
69          cout << endl;
70      }
71              break;
72      default: {
73          cout << "Допущенны ошибки" << endl;
74          return -1; // Завершение программы при некоректных данных
75      }
76      }
77
78      auto start = chrono::high_resolution_clock::now(); // Начало отсчета время сорировки
79      long long int operation_counter = insertion_sort(array); // Вызов функции сортировки массива с воращением значения подсчета операций
80      auto end = chrono::high_resolution_clock::now(); // Окончание отсчета время сорировки
81      cout << "Вывод отсортированного массива" << endl;
82      for (auto& it : array) // Вывод массива
83          cout << it << " ";
84      cout << endl;
85      cout << "Количество операций срванения, присоения алгоритма сортировки: " << operation_counter << endl;
86      cout << "Время работы алгоритма сортировки: " << chrono::duration_cast<chrono::microseconds>(end - start).count() << endl;
87      return 0;
88  }
```

Рисунок 4 – Функция main для алгоритма сортировки простыми вставками

2.3.2 Тестирование

Стоит задача протестировать программу с заданным размером массива n = 10 (рис.5), n = 100, n =1 000, n =1 0000, n = 100000, n = 1000000. Чтобы провести данной тестирование понадобился ввод с случайной генерацией числа. Результаты тестирования от n = 100 до n = 1000000 будут продемонстрированы в таблице 2. Воспользуемся структурой high_resolution_clock для подсчёта затраченного времени на сортировку. Для более точных результатов в программе будем рассматривать микросекунды, которые в дальнейшем, для заполнения таблицы, переведем в миллисекунды.

```
Введите размер массива: 10
Выберите тип заполнения массива
a - рандом
b - с одним заданым значением
c - ручной ввод массива
a
-5 -2 -5 3 4 -4 6 -1 7 -10
Вывод отсортированного массива
-10 -5 -5 -4 -2 -1 3 4 6 7
Количество операций срванения, присоения алгоритма сортировки: 99
Время работы алгоритма сортировки: 2
```

Рисунок 5 - Тестирование программы при n = 10

Таблица 2. Сводная таблица результатов

n	T(n), мс	$T_т$=C+M	$T_п$=$C_п$+$M_п$
100	0.024	-	7779
1000	3.3061	-	706185
10000	376.4537	-	67114215
100000	31820.1538	-	6823051368
1000000	2493639.7998	-	682414374213

2.3.3 Построение графика

На основе полученных данных, продемонстрированных в таблице 2 построим график функции роста $T_п$ этого алгоритма от размера массива n(рис.5).

Рисунок 6 - График функции роста $T_п$ этого алгоритма от размера массива n

2.4 Вывод по заданию №1

На основе полученных данных тестирования и вычислительной сложности алгоритма, можно сделать вывод, что алгоритм сортировки простыми вставками

имеет квадратичную вычислительную сложность, что означает, что время выполнения будет расти с увеличением размера массива квадратично.

3 ЗАДАНИЕ №2

3.1 Формулировка задачи

Оценить вычислительную сложность алгоритма простой сортировки вставками в наихудшем и наилучшем случаях.

1. Провести дополнительные прогоны программы на массивах при n = 100, 1000, 10000, 100000 и 1000000 элементов, отсортированных:

a. строго в убывающем порядке значений, результаты представить в сводной таблице по формату Таблицы 2;

b. строго в возрастающем порядке значений, результаты представить в сводной таблице по формату Таблицы 2;

2. Сделать вывод о зависимости (или независимости) алгоритма сортировки от исходной упорядоченности массива.

3.2 Тестирование программы

Дополнительное тестирование программы на массивах при n = 100, 1000, 10000, 100000 и 1000000 элементов.

3.2.1 Массив упорядоченный по убыванию

Будет проведено тестирование программы на массивах при n = 100, 1000, 10000, 100000 и 1000000 элементов, которые отсортированы в строго убывающем порядке. Изменим программу в функции main, чтобы значения элементов массива сортировались в убывающем порядке (рис.7) и продемонстрируем работу программы при n = 10 (рис.8). Для сортировки значений добавим библиотеку algorithm (рис.6). Algorithm - стандартная библиотека C++, которая обрабатывает диапазоны итератора, которые обычно определяются их начальными или конечными позициями. Алгоритм сортировки простыми вставками не изменяется и соответствует продемонстрированному на рисунке 2.

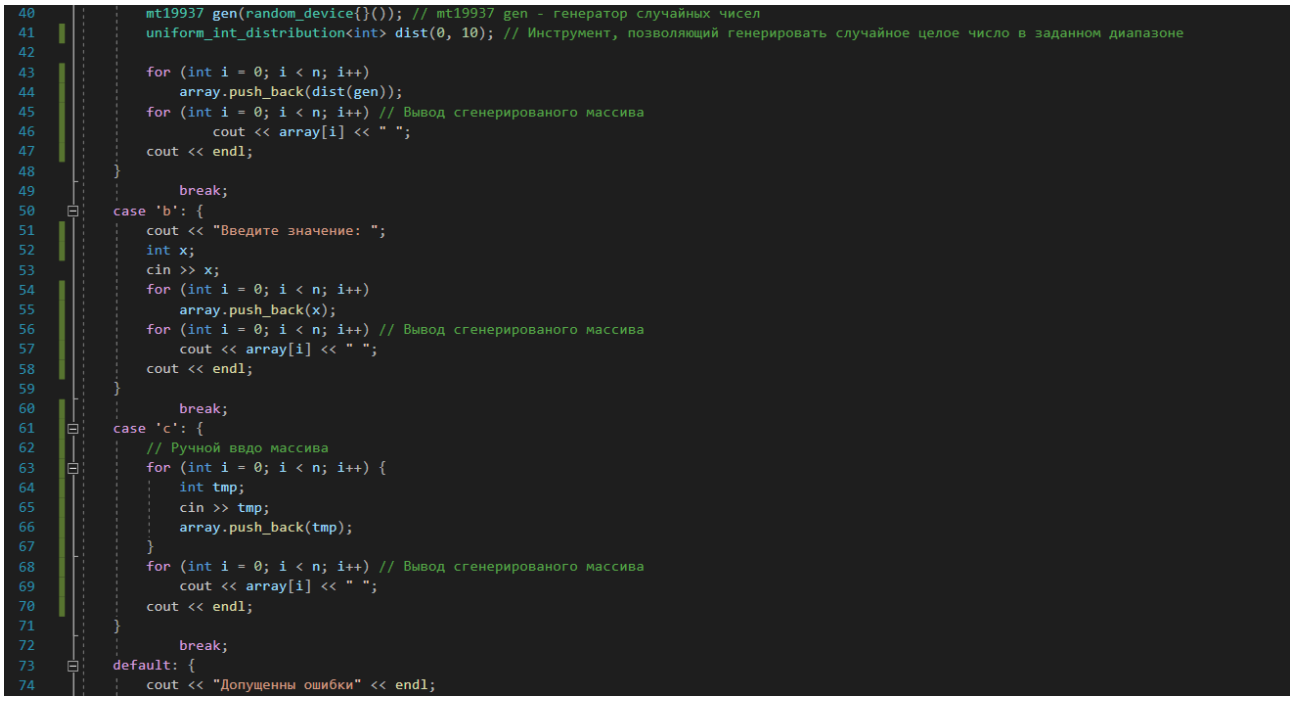
```cpp
#include <iostream>
#include <random>
#include <vector>
#include <chrono>
#include <algorithm>
using namespace std;

long long int insertion_sort(vector<int>& arr) {
    long long int operations_counter = 0;
    for (int i = 0; i < arr.size(); ++i) {
        int tmp = arr[i];
        int j = i;
        while (j > 0 && arr[j - 1] > tmp) {
            arr[j] = arr[j - 1];
            --j;
            operations_counter += 3; // Сравнение в цикле, присвоение в теле цикла, декремент
        }
        arr[j] = tmp;
        operations_counter += 5; // Сравние в цикле при входе, 3 присвоение, 2 из которых вышее цикла while, сравнение в цикле при выходе
    }
    ++operations_counter; // Сравнение при выходе из цикла for
    return operations_counter;
}

int main() {
    setlocale(LC_ALL, "Rus");
    int n;
    cout << "Введите размер массива: ";
    cin >> n;
    vector<int> array;

    cout << "Выберите тип заполнения массива" << endl;
    cout << "a - рандом" << endl;
    cout << "b - с одним заданым значением" << endl;
    cout << "c - ручной ввод массива" << endl;
    char ty;
    cin >> ty;
    switch (ty) { //Конструкция switch case default для выбор типа заполения массива
    case 'a': {
```

Рисунок 7 – Программа алгоритма сортировки простыми вставками и main функция



```cpp
        mt19937 gen(random_device{}()); // mt19937 gen - генератор случайных чисел
        uniform_int_distribution<int> dist(0, 10); // Инструмент, позволяющий генерировать случайное целое число в заданном диапазоне

        for (int i = 0; i < n; i++)
            array.push_back(dist(gen));
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
                cout << array[i] << " ";
        cout << endl;
    }
            break;
    case 'b': {
        cout << "Введите значение: ";
        int x;
        cin >> x;
        for (int i = 0; i < n; i++)
            array.push_back(x);
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
            cout << array[i] << " ";
        cout << endl;
    }
            break;
    case 'c': {
        // Ручной ввдо массива
        for (int i = 0; i < n; i++) {
            int tmp;
            cin >> tmp;
            array.push_back(tmp);
        }
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
            cout << array[i] << " ";
        cout << endl;
    }
            break;
    default: {
        cout << "Допущенны ошибки" << endl;
```

Рисунок 8 – main функция

```
        mt19937 gen(random_device{}()); // mt19937 gen - генератор случайных чисел
        uniform_int_distribution<int> dist(0, 10); // Инструмент, позволяющий генерировать случайное целое число в заданном диапазоне

        for (int i = 0; i < n; i++)
            array.push_back(dist(gen));
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
                cout << array[i] << " ";
        cout << endl;
    }
            break;
    case 'b': {
        cout << "Введите значение: ";
        int x;
        cin >> x;
        for (int i = 0; i < n; i++)
            array.push_back(x);
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
            cout << array[i] << " ";
        cout << endl;
    }
            break;
    case 'c': {
        // Ручной ввдо массива
        for (int i = 0; i < n; i++) {
            int tmp;
            cin >> tmp;
            array.push_back(tmp);
        }
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
            cout << array[i] << " ";
        cout << endl;
    }
            break;
    default: {
        cout << "Допущенны ошибки" << endl;
```

Рисунок 9 – main функция

```
            return -1; // Завершение программы при некоректных данных
        }
        }
        sort(array.begin(), array.end(), greater<int>()); // Массив отсортированный по убыванию
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
            cout << array[i] << " ";
        cout << endl;
        auto start = chrono::high_resolution_clock::now(); // Начало отсчета время сорировки
        long long int operation_counter = insertion_sort(array); // Вызов функции сортировки массива с воращением значения подсчета операций
        auto end = chrono::high_resolution_clock::now(); // Окончание отсчета время сорировки
        cout << "Вывод отсортированного массива" << endl;
        for (auto& it : array) // Вывод массива
            cout << it << " ";
        cout << endl;
        cout << "Количество операций срванения, присоения алгоритма сортировки: " << operation_counter << endl;
        cout << "Время работы алгоритма сортировки: " << chrono::duration_cast<chrono::nanoseconds>(end - start).count() << endl;
        return 0;
    }
```

Рисунок 10 – main функция

```
Введите размер массива: 10
Выберите тип заполнения массива
a - рандом
b - с одним заданым значением
c - ручной ввод массива
a
4 9 8 4 3 9 2 5 8 8
9 9 8 8 8 5 4 4 3 2
Вывод отсортированного массива
2 3 4 4 5 8 8 8 9 9
Количество операций срванения, присоения алгоритма сортировки: 171
Время работы алгоритма сортировки: 4100
```

Рисунок 11 – Результаты тестирования программы при n = 10 и с отсортированными значениями по убыванию

Так как значения идут в строго убывающем порядке, то можно сделать вывод, что данная ситуация являться худшим случаем, следовательно имеет сложность $O(n^2)$. Следовательно, в худшем случае алгоритм является квадратичным. Результаты тестирования будут приведены в таблице 3.

Таблица 3. Сводная таблица результатов

n	**T(n), мс**	**$T_т$=C+M**	**$T_п$=$C_п$+$M_п$**
100	0.1318	-	14019
1000	7.0928	-	1367406
10000	844.9592	-	136407252
100000	65928.3491	-	13636746549
1000000	4519260.135	-	136367465590

На основе полученных данных, продемонстрированных в таблице 3, построим график функции роста $T_п$ алгоритма сортировки простыми вставками с отсортированными значениями по убыванию от размера массива n (рис.9).

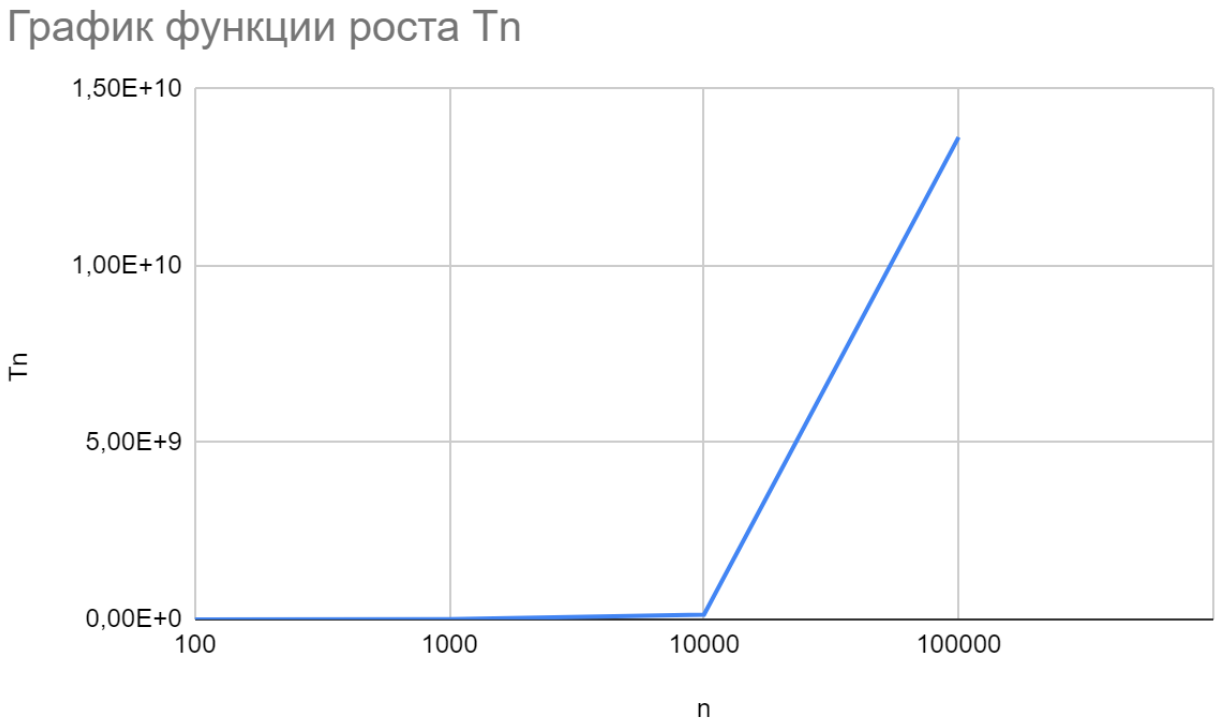


Рисунок 12 - График функции роста T_n алгоритма сортировки простыми вставками с отсортированными значениями по убыванию от размера массива n

3.2.2 Массив упорядоченный по возрастанию

Будет проведено тестирование программы на массивах при n = 100, 1000, 10000, 100000 и 1000000 элементов, которые отсортированы в строго возрастающем порядке. Изменим программу в функции main, чтобы значения элементов массива сортировались в возрастающем порядке (рис. 10) и продемонстрируем работу программы при n = 10 (рис.11). Для сортировки значений добавим библиотеку algorithm (рис.6). Algorithm - стандартная библиотека C++, которая обрабатывает диапазоны итератора, которые обычно определяются их начальными или конечными позициями. Алгоритм сортировки простыми вставками не изменяется и соответствует продемонстрированному на рисунке 2.

```cpp
int main() {
    setlocale(LC_ALL, "Rus");
    int n;
    cout << "Введите размер массива: ";
    cin >> n;
    vector<int> array;

    cout << "Выберите тип заполнения массива" << endl;
    cout << "a - рандом" << endl;
    cout << "b - с одним заданым значением" << endl;
    cout << "c - ручной ввод массива" << endl;
    char ty;
    cin >> ty;
    switch (ty) { //Конструкция switch case default для выбор типа заполения массива
    case 'a': {
        mt19937 gen(random_device{}()); // mt19937 gen - генератор случайных чисел
        uniform_int_distribution<int> dist(0, 10); // Инструмент, позволяющий генерировать случайное целое число в заданном диапазоне

        for (int i = 0; i < n; i++)
            array.push_back(dist(gen));
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
                cout << array[i] << " ";
        cout << endl;
    }
            break;
    case 'b': {
        cout << "Введите значение: ";
        int x;
        cin >> x;
        for (int i = 0; i < n; i++)
            array.push_back(x);
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
            cout << array[i] << " ";
        cout << endl;
    }
            break;
    case 'c': {
        // Ручной ввдо массива
        for (int i = 0; i < n; i++) {
            int tmp;
            cin >> tmp;
            array.push_back(tmp);
        }
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
            cout << array[i] << " ";
        cout << endl;
    }
            break;
```

Рисунок 13 – Тестирование программы при n=10 и с отсортированными значениями по возрастанию

```cpp
    default: {
        cout << "Допущенны ошибки" << endl;
        return -1; // Завершение программы при некоректных данных
    }
    }
    sort(array.begin(), array.end()); // Массив отсортированный не по убыванию
    for (int i = 0; i < n; i++) // Вывод сгенерированого массива
        cout << array[i] << " ";
    cout << endl;
    auto start = chrono::high_resolution_clock::now(); // Начало отсчета время сорировки
    long long int operation_counter = insertion_sort(array); // Вызов функции сортировки массива с воращением значения подсчета операций
    auto end = chrono::high_resolution_clock::now(); // Окончание отсчета время сорировки
    cout << "Вывод отсортированного массива" << endl;
    for (auto& it : array) // Вывод массива
        cout << it << " ";
    cout << endl;
    cout << "Количество операций срванения, присоения алгоритма сортировки: " << operation_counter << endl;
    cout << "Время работы алгоритма сортировки: " << chrono::duration_cast<chrono::nanoseconds>(end - start).count() << " наносекунд" << endl;
    return 0;
}
```

Рисунок 14 – Тестирование программы при n=10 и с отсортированными значениями по возрастанию

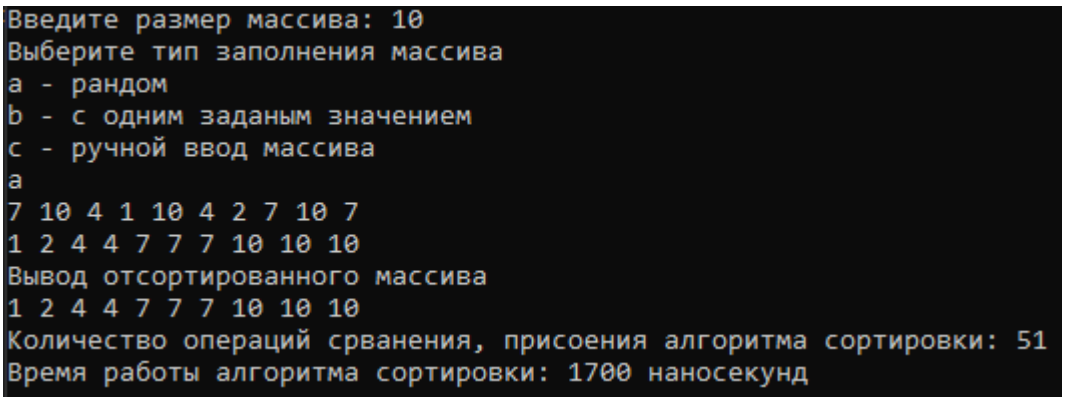


Рисунок 15 – Результаты тестирования программы при n = 10 и с отсортированными значениями по возрастанию

Так как значения элементов массива идут в строго возрастающем порядке, то можно сделать вывод, что данная ситуация будет являться лучшим случаем, так как нет необходимости сдвигать элементы массива, а следовательно сложность алгоритма равна O(n). Следовательно, в лучшем случае алгоритм является линейным. Результаты тестирования будут приведены в таблице 4.

Таблица 4. Сводная таблица результатов

n	**T(n), мс**	**$T_т$=C+M**	**$T_п$=$C_п$+$M_п$**
100	0.0092	-	501
1000	0.0894	-	5001
10000	0.8748	-	50001
100000	8.8268	-	500001
1000000	88.9373	-	5000001

На основе полученных данных, продемонстрированных в таблице 4, построим график функции роста $T_п$ этого алгоритма от размера массива n с отсортированными значениями по возрастанию (рис.12).

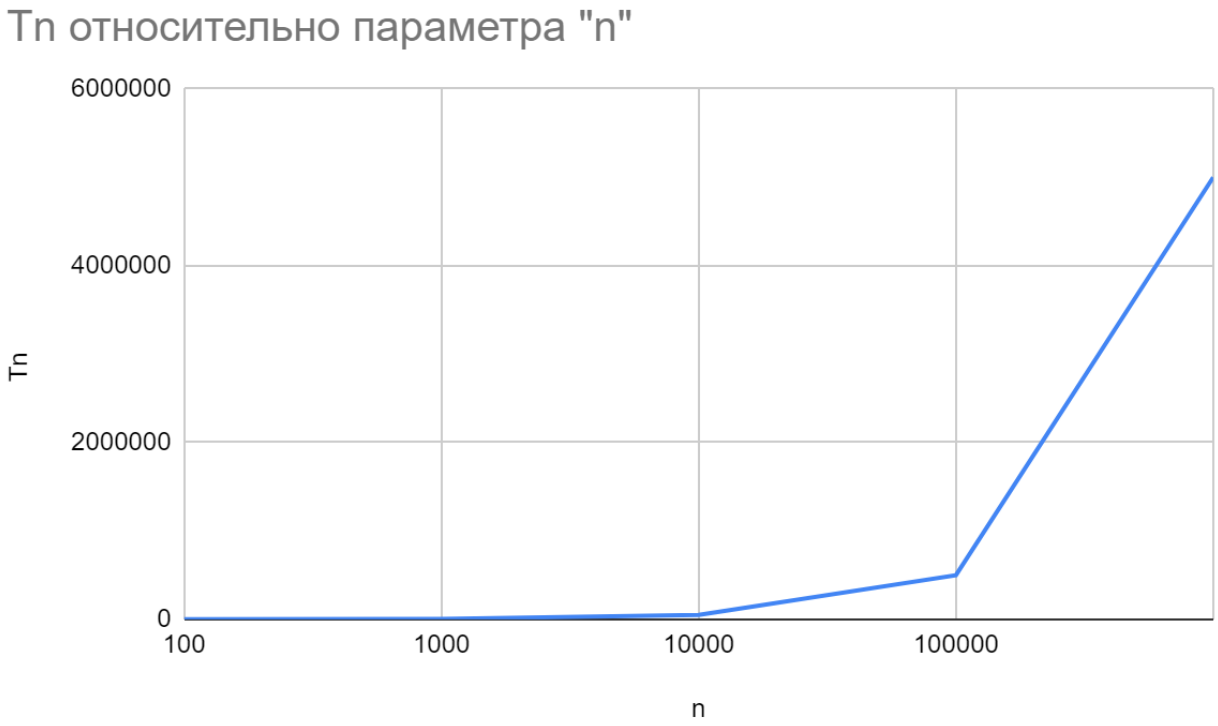


Рисунок 16 - График функции роста $T_п$ алгоритма сортировки простыми вставками с отсортированными значениями по возрастанию от размера массива n

3.3 Вывод по заданию №2

Алгоритм сортировки вставкой зависим от исходной упорядоченности массива. Время выполнения алгоритма сортировки простой вставкой зависит от первоначальной упорядоченности массива. Если значения элементов массива строго возрастают, алгоритму потребуется меньше действий, так как нет необходимости сдвигать элементы.

Если значения элементов массива строго убывают, алгоритму сортировки простой вставкой потребуется большее количество шагов, так как для каждого элемента потребуется больше сдвигов, чтобы найти подходящее место для вставки.

Таким образом, время выполнения алгоритма сортировки простой вставкой непосредственно зависит от начальной степени упорядоченности массива.

4 ЗАДАНИЕ №3

4.1 Формулировка задания

Сравнить эффективность алгоритмов простых сортировок

1. Выполнить разработку и программную реализацию алгоритма простого выбора.

2. Аналогично заданиям 1 и 2 сформировать таблицы с результатами эмпирического исследования второго алгоритма в среднем, лучшем и худшем случаях в соответствии с форматом Таблицы 2 (на тех же массивах, что и в заданиях 1 и 2).

3. Определить ёмкостную сложность алгоритма от n.

4. На одном сравнительном графике отобразить функции $T_п(n)$ двух алгоритмов сортировки в худшем случае.

5. Аналогично на другом общем графике отобразить функции $T_п(n)$ двух алгоритмов сортировки для лучшего случая.

6. Выполнить сравнительный анализ полученных результатов для двух алгоритмов.

4.2 Математическая модель решения алгоритма

4.2.1 Описание выполнения и блок-схема алгоритма сортировки простым выбором

На первом шаге неупорядоченная часть – весь массив длиной n. В неупорядоченной части выбирается элемент с минимальным значением, который обменивается местами с первым элементом этой части массива и исключается из дальнейшей сортировки (включается в уже упорядоченную часть массива).

Затем выбирается минимальный элемент среди оставшихся n-1 элементов исходного массива, обменивается местами начальным элементом неупорядоченной части и также исключается из сортировки.

Процесс продолжается, пока в неотсортированной части массива не окажется единственный элемент, который считается упорядоченным.

Реализация данного описания выполнения алгоритма представлена в виде блок-схемы (рис.13).

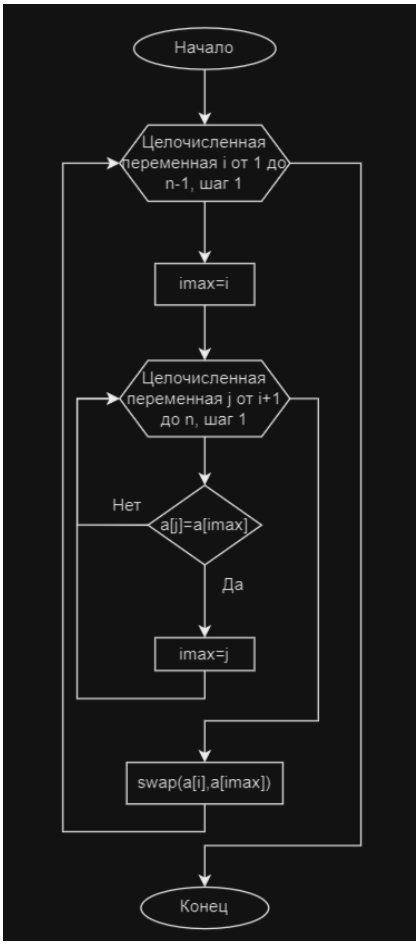


Рисунок 17 – Блок-схема алгоритма сортировки простым выбором

4.2.2 Доказательство корректности циклов алгоритма сортировки простым выбором

Инвариант для внешнего цикла: значение переменной i всегда меньше n-1.

Инвариант для внутреннего цикла: значение переменной j всегда больше i и меньше n.

Докажем конечность циклов. Внешний цикл for проходит через все элементы массива начиная со второго, так как первые 0 элементов в массиве уже отсортированы по определению. Внутренний цикл находит минимум среди

неотсортированных элементов. Этот минимум меняется местами с текущим элементом, тем самым осуществляя сортировку. Этот процесс повторяется, пока не будут рассмотрены все элементы массива. Таким образом циклы алгоритма завершаются за конечное число повторений.

Из доказательства можно сделать вывод, что все циклы данного алгоритма корректны.

4.2.3 Определение ёмкостной сложности, ситуации лучшего, среднего и худшего случая и функции роста времени работы алгоритма сортировки простым выбором

Таблица 5. Псевдокод и анализ алгоритма сортировки выбором

№	Алгоритм, записанный на псевдокоде	Количество выполнений оператора
1	SelectionSort(a,n){	
2	for i←1 to n-1 do	n
3	imax←i	n-1
4	for j←i+1 to n do	$(n^2-n)/2$
5	if a[j]←a[imax] then	$(n^2-n)/2-(n-1)$
6	imax←j	$(n^2-n)/2-(n-1)$
7	Endlf od	
8	swap(a[i],a[imax])	3*(n-1)
9	od	
10	}	

a. Лучший случай - массив уже отсортирован. В этом случае алгоритм будет иметь сложность $O(n^2)$.

Средний случай - массив заполнен случайными числами. В этом случае алгоритм будет иметь сложность $O(n^2)$.

Худший случай - массив отсортирован в обратном порядке. В этом случае количество операций также будет $O(n^2)$.

b. Функции роста времени:

Лучший случай: $O(n^2)$.

Худший случай: $O(n^2)$.

Для данного метода сортировки, время исполнения постоянно увеличивается квадратично с ростом размера входного массива. Следовательно, можно использовать квадратичную функцию для описания функции роста данного сортировочного метода.

Ёмкостная сложность алгоритма будет равна O(1).

4.3 Реализация алгоритма на языке C++, проведение тестирования и построение графика

4.3.1 Реализация алгоритма сортировки простым выбором на языке C++

Реализуем данный алгоритм на языке C++(рис.14,15). Для реализации понадобятся такие библиотеки, как iostream, vector, random, chrono, algorithm. Iostream — это заголовочный файл с классами, функциями и переменными для организации ввода-вывода в языке программирования C++. Vector — это шаблон класса для контейнеров последовательности. Вектор хранит элементы заданного типа в линейном расположении и обеспечивает быстрый случайный доступ к любому элементу. Для сортировки значений добавим библиотеку algorithm. Algorithm - стандартная библиотека C++, которая обрабатывает диапазоны итератора, которые обычно определяются их начальными или конечными позициями. Random - позволяет генерировать случайные числа в диапазоне. В данной программе задан диапазон от 1 до 10. Chrono позволяет реализовать такие концепции, как: интервалы времени, моменты времени, таймеры. Для подсчёта количество операций присваивания или сравнения введём переменную operations_counter, которая представляет собой целое число в диапазоне от -9223372036854775807 до 9223372036854775807 и занимает 8 байта в памяти.

```cpp
#include <iostream>
#include <random>
#include <vector>
#include <chrono>
using namespace std;

long long int selection_sort(vector<int>& arr) {
    long long int operations_counter = 0;
    for (int i = 0; i < arr.size(); ++i) {
        int min_ind = i;
        for (int j = i + 1; j < arr.size(); ++j) {
            if (arr[j] < arr[min_ind]) {
                min_ind = j; //  Теперь минимальный индекс равен текущему j-ому
                ++operations_counter; // Присвоение
            }
            ++operations_counter; // Сравнение
        }
        swap(arr[i], arr[min_ind]);
        operations_counter += 4; // Присвоение сравнение при входе и выходе из цикла
    }
    ++operations_counter; // Выход из цикла
    return operations_counter;
}

```

Рисунок 18 – Алгоритм алгоритма сортировки простым выбором с подключением библиотек

```cpp
int main() {
    setlocale(LC_ALL, "Rus");
    int n;
    cout << "Введите размер массива: ";
    cin >> n;
    vector<int> array;

    cout << "Выберите тип заполнения массива" << endl;
    cout << "a - рандом" << endl;
    cout << "b - с одним заданым значением" << endl;
    cout << "c - ручной ввод массива" << endl;
    char ty;
    cin >> ty;
    switch (ty) { //Конструкция switch case default для выбор типа заполения массива
    case 'a': {
        mt19937 gen(random_device{}()); // mt19937 gen - генератор случайных чисел
        uniform_int_distribution<int> dist(0, 10); // Инструмент, позволяющий генерировать случайное целое число в заданном диапазоне

        for (int i = 0; i < n; i++)
            array.push_back(dist(gen));
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
                cout << array[i] << " ";
        cout << endl;
    }
            break;
    case 'b': {
        cout << "Введите значение: ";
        int x;
        cin >> x;
        for (int i = 0; i < n; i++)
            array.push_back(x);
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
            cout << array[i] << " ";
        cout << endl;
    }
            break;
    case 'c': {
        // Ручной ввдо массива
        for (int i = 0; i < n; i++) {
```

Рисунок 19 – Функция main для алгоритма сортировки простым выбором

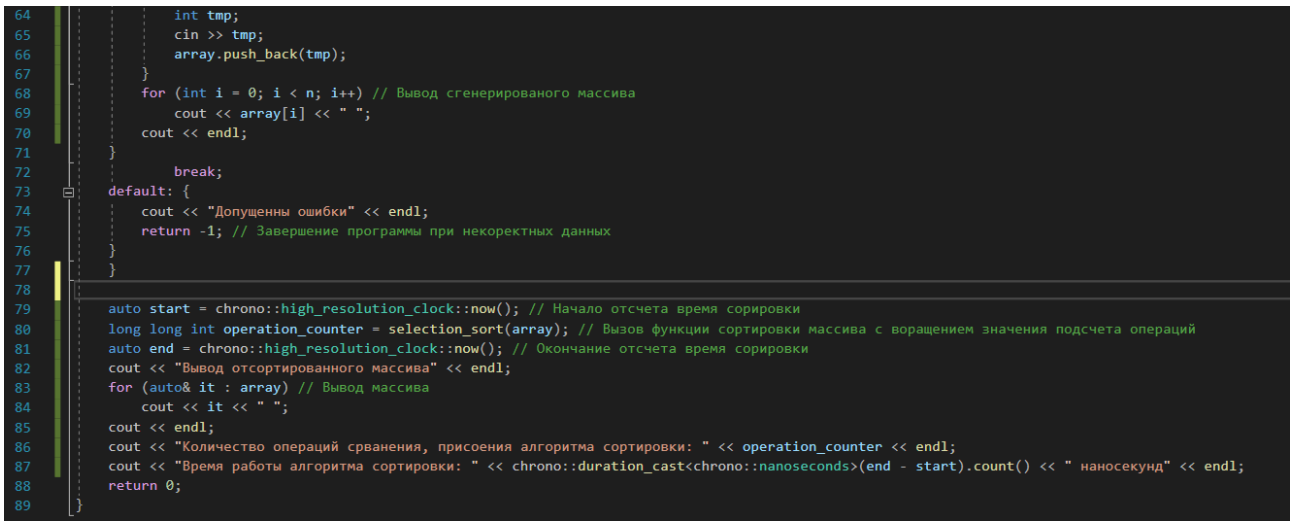


Рисунок 20 – Функция main для алгоритма сортировки простым выбором

4.3.2 Тестирование при случайном заполнении массива

Стоит задача протестировать программу с заданным размером массива n = 10 (рис.16), n = 100, n = 1000, n = 10000, n = 100000, n = 1000000. Чтобы провести данной тестирование понадобился ввод с случайной генерацией числа. Результаты тестирования от n = 100 до n = 1000000 будут продемонстрированы в таблице 6. Воспользуемся структурой high_resolution_clock для подсчёта затраченного времени на сортировку. Для более точных результатов в программе будем рассматривать микросекунды, которые в дальнейшем, для заполнения таблицы, переведем в миллисекунды.

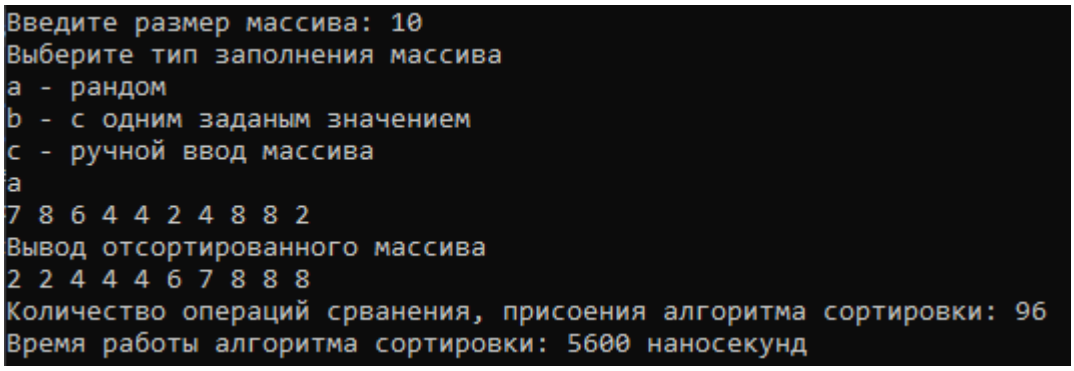


Рисунок 20 - Тестирование программы при n=10

Таблица 6. Сводная таблица результатов

n	T(n), мс	$T_т$=C+M	$T_п$=$C_п$+$M_п$
100	0.3425	-	5519
1000	33.2711	-	505401
10000	3287.6379	-	50054360
100000	36577.144653	-	5000542269
1000000	3953527.144653	-	50000543619

4.3.3 Построение графика

На основе полученных данных, продемонстрированных в таблице 6 построим график функции роста $T_п$ этого алгоритма от размера массива n(рис.17).

Рисунок 18 - График функции роста $T_п$ этого алгоритма от размера массива n

4.3.4 Тестирование при упорядоченном по убыванию элементов массива и построение графика

Будет проведено тестирование программы на массивах при n = 100, 1000, 10000, 100000 и 1000000 элементов, которые отсортированы в строго

убывающем порядке. Добавим в программу алгоритм, в котором проведем сортировку массива по убыванию (рис.20). В функцию main добавим вызов функции сортировки по убыванию (рис.21) и продемонстрируем работу программы при n = 10 (рис.22).

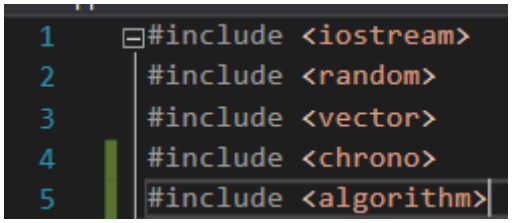
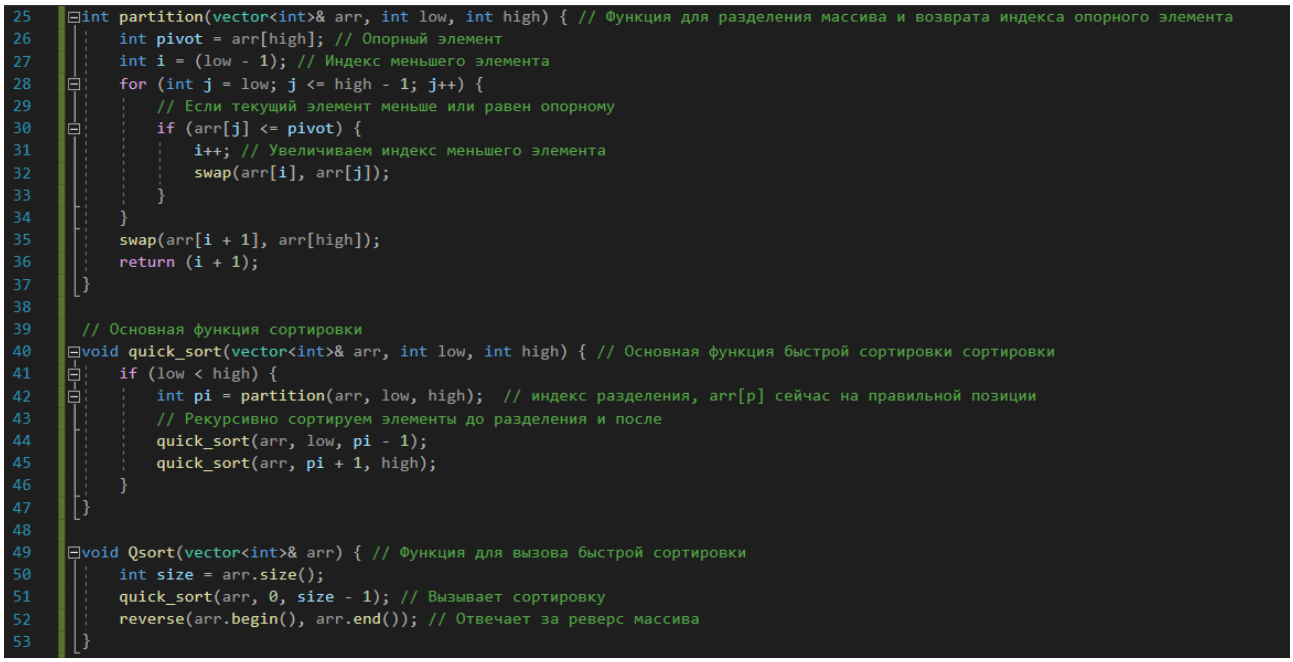


Рисунок 19 – Подключенные библиотеки

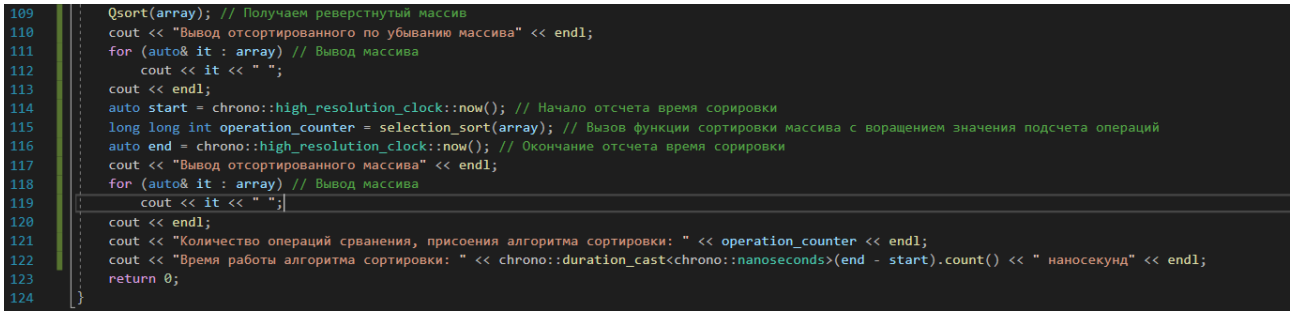


```
int partition(vector<int>& arr, int low, int high) { // Функция для разделения массива и возврата индекса опорного элемента
    int pivot = arr[high]; // Опорный элемент
    int i = (low - 1); // Индекс меньшего элемента
    for (int j = low; j <= high - 1; j++) {
        // Если текущий элемент меньше или равен опорному
        if (arr[j] <= pivot) {
            i++; // Увеличиваем индекс меньшего элемента
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

// Основная функция сортировки
void quick_sort(vector<int>& arr, int low, int high) { // Основная функция быстрой сортировки сортировки
    if (low < high) {
        int pi = partition(arr, low, high);  // индекс разделения, arr[p] сейчас на правильной позиции
        // Рекурсивно сортируем элементы до разделения и после
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}

void Qsort(vector<int>& arr) { // Функция для вызова быстрой сортировки
    int size = arr.size();
    quick_sort(arr, 0, size - 1); // Вызывает сортировку
    reverse(arr.begin(), arr.end()); // Отвечает за реверс массива
}
```

Рисунок 20 – Алгоритм сортировки по убыванию на основании QSort и последующим шагом перевернутого массива



```
    Qsort(array); // Получаем реверстнутый массив
    cout << "Вывод отсортированного по убыванию массива" << endl;
    for (auto& it : array) // Вывод массива
        cout << it << " ";
    cout << endl;
    auto start = chrono::high_resolution_clock::now(); // Начало отсчета время сорировки
    long long int operation_counter = selection_sort(array); // Вызов функции сортировки массива с воращением значения подсчета операций
    auto end = chrono::high_resolution_clock::now(); // Окончание отсчета время сорировки
    cout << "Вывод отсортированного массива" << endl;
    for (auto& it : array) // Вывод массива
        cout << it << " ";
    cout << endl;
    cout << "Количество операций срванения, присоения алгоритма сортировки: " << operation_counter << endl;
    cout << "Время работы алгоритма сортировки: " << chrono::duration_cast<chrono::nanoseconds>(end - start).count() << " наносекунд" << endl;
    return 0;
}
```

Рисунок 21 – Тестирование программы при n = 10 и с отсортированными значениями по убыванию, изменения в функции main

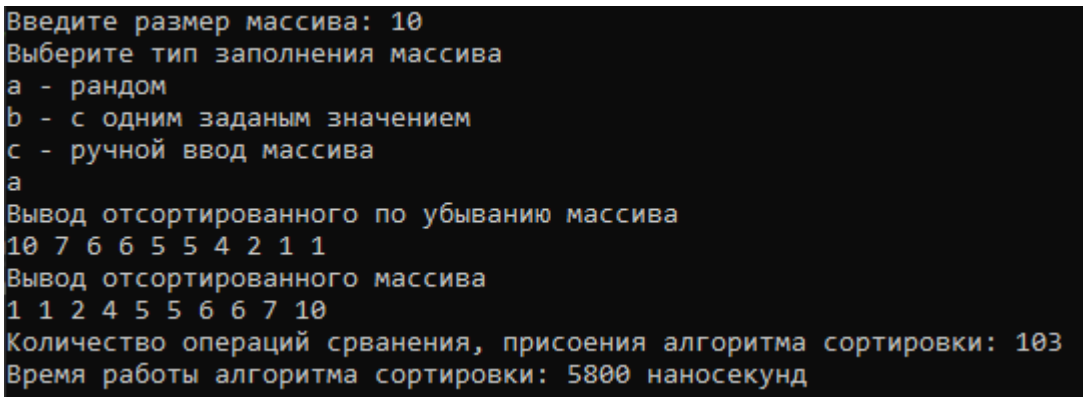


Рисунок 22 – Результаты тестирования программы при n = 10 и с отсортированными значениями по убыванию

Так как значения идут в строго убывающем порядке, то можно сделать вывод, что данная ситуация будет являться худшим случаем, а следовательно сложность алгоритма равна $O(n^2)$. Следовательно, в худшем случае алгоритм имеет квадратичную сложность алгоритма. Результаты тестирования будут приведены в таблице 7.

Таблица 7. Сводная таблица результатов

n	**T(n), мс**	**$T_т$=C+M**	**$T_п$=$C_п$+$M_п$**
100	2.76614	-	50610
1000	3.951119	-	506270
10000	311.236091	-	50063647
100000	33372,651975	-	5000619487
1000000	3333472,16236	-	50000629799

На основе полученных данных, продемонстрированных в таблице 7, построим график функции роста $T_п$ алгоритма сортировки простым выбором с отсортированными значениями по убыванию от размера массива n (рис.22).

Рисунок 23 - График функции роста $T_п$ алгоритма сортировки простым выбором с отсортированными значениями по убыванию от размера массива n

4.3.5 Массив упорядоченный по возрастанию

Будет проведено тестирование программы на массивах при n = 100, 1000, 10000, 100000 и 1000000 элементов, которые отсортированы в строго возрастающем порядке. Добавим в программу алгоритм, в которой проведем сортировку массива по возрастанию(рис.23). В функцию main добавим вызов функции сортировки по возрастанию(рис.24) и продемонстрируем работу программы при n=10 (рис.25).

```cpp
int partition(vector<int>& arr, int low, int high) { // Функция для разделения массива и возврата индекса опорного элемента
    int pivot = arr[high]; // Опорный элемент
    int i = (low - 1); // Индекс меньшего элемента
    for (int j = low; j <= high - 1; j++) {
        // Если текущий элемент меньше или равен опорному
        if (arr[j] <= pivot) {
            i++; // Увеличиваем индекс меньшего элемента
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

// Основная функция сортировки
void quick_sort(vector<int>& arr, int low, int high) { // Основная функция быстрой сортировки сортировки
    if (low < high) {
        int pi = partition(arr, low, high);  // индекс разделения, arr[p] сейчас на правильной позиции
        // Рекурсивно сортируем элементы до разделения и после
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}

void Qsort(vector<int>& arr) { // Функция для вызова быстрой сортировки
    int size = arr.size();
    quick_sort(arr, 0, size - 1); // Вызывает сортировку
}
```

Рисунок 24 - Функция сортировки массива по возрастанию

```cpp

int main() {
    setlocale(LC_ALL, "Rus");
    int n;
    cout << "Введите размер массива: ";
    cin >> n;
    vector<int> array;

    cout << "Выберите тип заполнения массива" << endl;
    cout << "a - рандом" << endl;
    cout << "b - с одним заданым значением" << endl;
    cout << "c - ручной ввод массива" << endl;
    char ty;
    cin >> ty;
    switch (ty) { //Конструкция switch case default для выбор типа заполения массива
    case 'a': {
        mt19937 gen(random_device{}()); // mt19937 gen - генератор случайных чисел
        uniform_int_distribution<int> dist(1, 10); // Инструмент, позволяющий генерировать случайное целое число в заданном диапазоне

        for (int i = 0; i < n; i++)
            array.push_back(dist(gen));
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
                cout << array[i] << " ";
        cout << endl;
    }
            break;
    case 'b': {
        cout << "Введите значение: ";
        int x;
        cin >> x;
        for (int i = 0; i < n; i++)
            array.push_back(x);
        for (int i = 0; i < n; i++) // Вывод сгенерированого массива
            cout << array[i] << " ";
        cout << endl;
    }
            break;
```

Рисунок 25 – Тестирование программы при n = 10 и с отсортированными значениями по возрастанию

```
88          cout << endl;
89      }
90              break;
91      case 'c': {
92          // Ручной ввдо массива
93          for (int i = 0; i < n; i++) {
94              int tmp;
95              cin >> tmp;
96              array.push_back(tmp);
97          }
98          for (int i = 0; i < n; i++) // Вывод сгенерированого массива
99              cout << array[i] << " ";
100         cout << endl;
101     }
102             break;
103     default: {
104         cout << "Допущенны ошибки" << endl;
105         return -1; // Завершение программы при некоректных данных
106     }
107     }
108
109     Qsort(array); // Получаем реверстнутый массив
110     cout << "Вывод отсортированного массива" << endl;
111     for (auto& it : array) // Вывод массива
112         cout << it << " ";
113     cout << endl;
114     auto start = chrono::high_resolution_clock::now(); // Начало отсчета время сорировки
115     long long int operation_counter = selection_sort(array); // Вызов функции сортировки массива с воращением значения подсчета операций
116     auto end = chrono::high_resolution_clock::now(); // Окончание отсчета время сорировки
117     cout << "Вывод отсортированного массива" << endl;
118     for (auto& it : array) // Вывод массива
119         cout << it << " ";
120     cout << endl;
121     cout << "Количество операций срванения, присоения алгоритма сортировки: " << operation_counter << endl;
122     cout << "Время работы алгоритма сортировки: " << chrono::duration_cast<chrono::nanoseconds>(end - start).count() << " наносекунд" << endl;
123     return 0;
124 }
```

Рисунок 26 – Тестирование программы при n = 10 и с отсортированными значениями по возрастанию

```
Введите размер массива: 10
Выберите тип заполнения массива
a - рандом
b - с одним заданым значением
c - ручной ввод массива
a
4 8 10 3 2 10 10 6 5 6
Вывод отсортированного массива
2 3 4 5 6 6 8 10 10 10
Вывод отсортированного массива
2 3 4 5 6 6 8 10 10 10
Количество операций срванения, присоения алгоритма сортировки: 86
Время работы алгоритма сортировки: 5800 наносекунд
```

Рисунок 27 – Результаты тестирования программы при n = 10 и с отсортированными значениями по возрастанию

Так как значения элементов массива идут в строго возрастающем порядке, то можно сделать вывод, что данная ситуация будет являться лучшим случаем, так как нет необходимости совершать обмен значениями, а следовательно сложность алгоритма равна $O(n^2)$. Следовательно, в лучшем случае алгоритм

имеет квадратичную сложность алгоритма. Результаты тестирования будут приведены в таблице 8.

Таблица 8. Сводная таблица результатов

n	**T(n), мс**	**$T_т$=C+M**	**$T_п$=$C_п$+$M_п$**
100	0.04509	-	5351
1000	2.96678	-	503501
10000	646.677288	-	50035001
100000	33426.102422	-	5000350001
1000000	3393258,87498	-	500003500001

На основе полученных данных, продемонстрированных в таблице 8, построим график функции роста $T_п$ этого алгоритма от размера массива n с отсортированными значениями по возрастанию (рис.28).

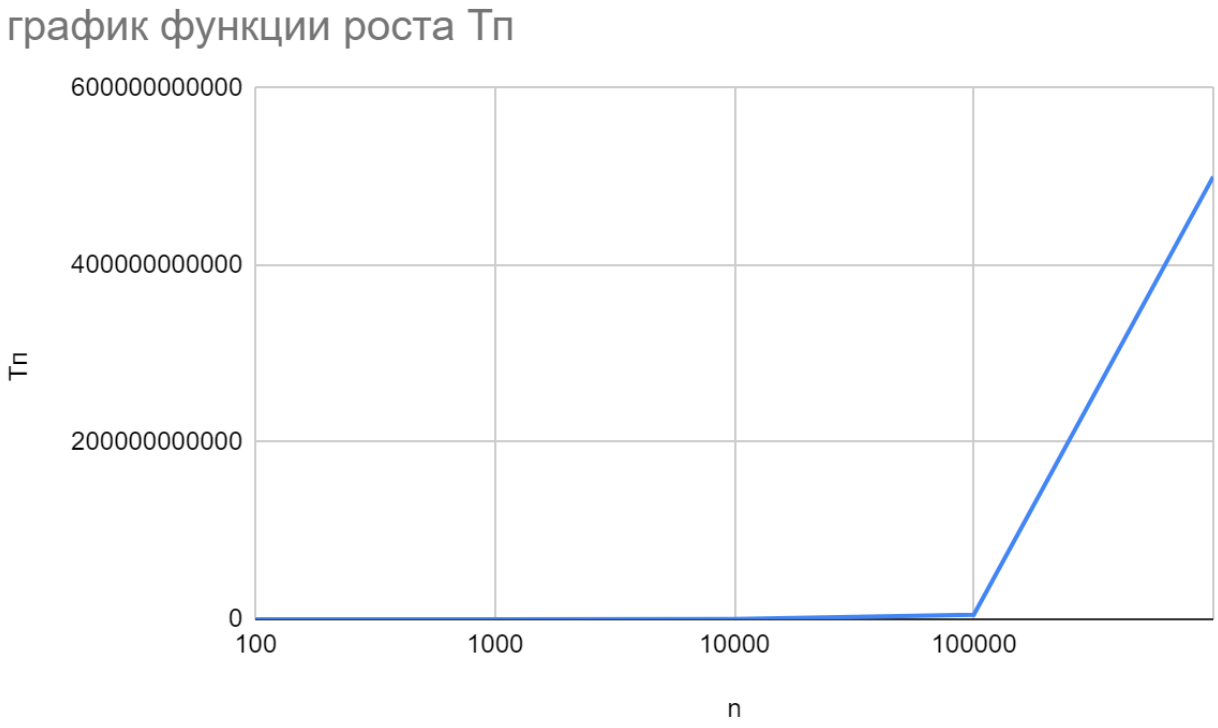


Рисунок 28 - График функции роста $T_п$ алгоритма сортировки простым выбором с отсортированными значениями по возрастанию от размера массива n

4.4 Сравнение графиков двух алгоритмов сортировки из задания 1 и 3

4.4.1 Отображение функции $T_п(n)$ двух алгоритмов сортировки в худшем случае

На основании данных из таблицы 3 и графика алгоритма сортировки простыми вставками в худшем случае(рис.12), и таблицы 6 и графика алгоритма сортировки простым выбором в худшем случае(рис.18), мы создадим новый график для сравнения роста графиков(рис.29).

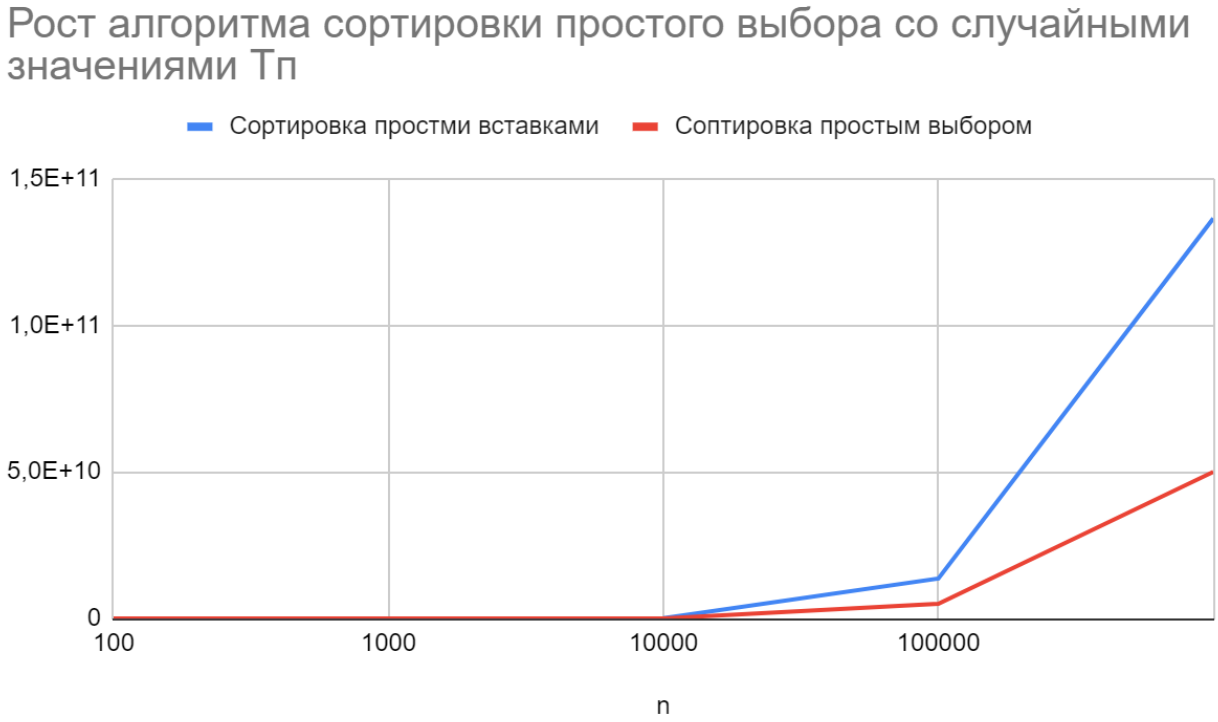


Рисунок 29 – График двух алгоритмов в худшем случае

На основе таблиц 3 и 7 и графика(рис.29), можно сделать вывод, что в худшем случае алгоритм сортировки простой вставкой менее эффективный, чем алгоритм сортировки простого выбора.

4.4.2 Отображение функции $T_п(n)$ двух алгоритмов сортировки в лучшем случае

На основании данных из таблицы 4 и графика алгоритма сортировки простыми вставками в лучшем случае(рис.16), и таблицы 8 и графика алгоритма сортировки простым выбором в лучшем случае(рис.28), мы создадим новый график для сравнения роста графиков(рис.30).

Рисунок 27 – График двух алгоритмов в лучшем случае

На основе таблиц 4 и 8 и графика(рис.30), можно сделать вывод, что в лучшем случае алгоритм сортировки простой вставкой более эффективный, чем алгоритм сортировки простого выбора.

4.8 Выводы по заданию №3

Алгоритм сортировки простыми вставками имеет линейную вычислительную сложность в лучшем случае O(n), в среднем и в худшем случае $O(n^2)$. Этот алгоритм эффективен для небольших наборов данных, но может стать очень медленным на больших массивах.

Алгоритм сортировки простым выбором также имеет квадратичную вычислительную сложность в худшем случае $O(n^2)$. В лучшем случае алгоритм простого выбора также демонстрирует квадратичную вычислительную сложность, так как у данной сортировки нет возможности преобразоваться для

лучшего случая. Этот алгоритм может быть очень медленным на больших массивах данных.

Таким образом, выбор алгоритма сортировки зависит от конкретной задачи и данных, с которыми он будет работать. В некоторых случаях сортировка простым выбором может быть более эффективной, а в других случаях - сортировка простой вставкой. Поэтому при выборе алгоритма необходимо учитывать размер и последовательность входных данных.

4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие сортировки называют простыми?

Простыми сортировками являются сортировка пузырьком, сортировка вставками и сортировка выбором.

2. Что означает понятие «внутренняя сортировка»?

Методы сортировки можно разделить на внутренние и внешние. Внутренняя сортировка – это упорядочение последовательности элементов, когда она целиком находится в оперативной памяти. Такие алгоритмы применяются к относительно небольшим по своему объёму последовательностям. Алгоритмам внутренней сортировки достаточно для решения задачи ресурса внутренней (оперативной) памяти.

3. Какие операции считаются основными при оценке сложности алгоритма сортировки?

Критическими (основными) называют операции, которые выполняются наиболее часто и время выполнения которых составляет основную часть общего времени выполнения алгоритма.

В ходе анализа эффективности алгоритма следует предварительно выявить эти критические операции (группы операций).

4. Какие характеристики сложности алгоритма используются при оценке эффективности алгоритма?

Критериями эффективности алгоритма являются скорость (время выполнения или, то же, время работы исполнителя – процессора ЭВМ) и расход памяти (внутренней в первую очередь) и/или других ресурсов.

5. Какая вычислительная и емкостная сложность алгоритма: простого обмена, простой вставки, простого выбора?

1. Простой обмен:

- Вычислительная сложность: $O(n^2)$

- Емкостная сложность: O(1)

2. Простая вставка:

- Вычислительная сложность: $O(n^2)$ в худшем случае, O(n) в лучшем случае
- Емкостная сложность: O(1)

3. Простой выбор:

- Вычислительная сложность: $O(n^2)$
- Емкостная сложность: O(1)

6. Какую роль в сортировке обменом играет условие Айверсона?

Условие Айверсона: если в очередном проходе сортировки при сравнении элементов не было сделано ни одной перестановки, то множество считается упорядоченным.

Это условие позволяет оптимизировать работу алгоритма для случаев, когда массив уже отсортирован или требует минимального количества обменов. При выполнении условия Айверсона - алгоритм прекращает дальнейшую сортировку, так как массив уже отсортирован.

7. Определите, каким алгоритмом, рассмотренным в этом задании, сортировался исходный массив 5 6 1 2 3. Шаги выполнения сортировки:

1) 1 5 6 2 3
2) 1 2 5 6 3
3) 1 2 3 5 6

Исходный массив сортировался алгоритмом сортировки простым выбором.

8. Какова вычислительная теоретическая сложность алгоритма сортировки, рассмотренного в вопросе 7.

Лучший случай - массив уже отсортирован. В этом случае алгоритм будет иметь сложность $O(n^2)$.

Средний случай - массив заполнен случайными числами. В этом случае алгоритм будет иметь сложность $O(n^2)$.

Худший случай - массив отсортирован в обратном порядке. В этом случае количество операций также будет $O(n^2)$.

5 ВЫВОДЫ

В ходе практической работы были выполнены следующие задачи:

- Актуализированы знания и приобретены умения по эмпирическому определению вычислительной сложности;

- Проведён анализ алгоритмов простой сортировки вставками и выбором;

- Были реализованы программы для алгоритмов простой сортировки вставками и выбором;

- Проведённое тестирование программ для алгоритмов простой сортировки вставками и выбором;

- Построены графики функции роста $T_п$ алгоритмов простой сортировки вставками и выбором от размера массива n.

- Произведено сравнение алгоритмов простой сортировки вставками и выбором на основе анализа, результатов тестирования и графиков.

Таким образом, главную цель практической работы, а именно актуализация знаний и приобретение практических умений по эмпирическому определению вычислительной сложности алгоритмов, можно считать выполненной.

6 ЛИТЕРАТУРА

1. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. – СПб: Питер, 2017. – 288 с.

2. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985. – 406 с.

3. Кнут Д.Э. Искусство программирования, том 3. Сортировка и поиск, 2-е изд. – М.: ООО «И.Д. Вильямс», 2018. – 832 с.

4. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. – К.: Издательство «Диасофт», 2001. – 688 с.

5. AlgoList – алгоритмы, методы, исходники [Электронный ресурс]. URL: http://algolist.manual.ru/ (дата обращения 15.03.2022).

6. Алгоритмы – всё об алгоритмах / Хабр [Электронный ресурс]. URL: https://habr.com/ru/hub/algorithms/ (дата обращения 15.03.2022).

7. НОУ ИНТУИТ | Технопарк Mail.ru Group: Алгоритмы и структуры данных [Электронный ресурс]. URL: https://intuit.ru/studies/courses/3496/738/info (дата обращения 15.03.2022).