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: Величко В.Д.

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 простого обмена. 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 простым обменом

Сортировка простым обменом (Exchange Sort, пузырьковая) – очень известный алгоритм, который изучают ещё в рамках школьного курса информатики.

В цикле, с начала массива до границы рассматриваемой области массива (в первом проходе до n-1, уменьшаясь с каждым разом на 1) соседние значения попарно сравниваются между собой и в случае, если предшествующий элемент больше последующего, меняются местами (для случая сортировки по возрастанию).

После первого прохода всего массива элемент с максимальным значением займет место последнего элемента – свою окончательную позицию, потому исключается из дальнейшего рассмотрения.

Алгоритм завершится, когда: а) рассматриваемая область станет пустой, или б) когда на очередном проходе по массиву не будет произведено ни одной перестановки (условие Айверсона).

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

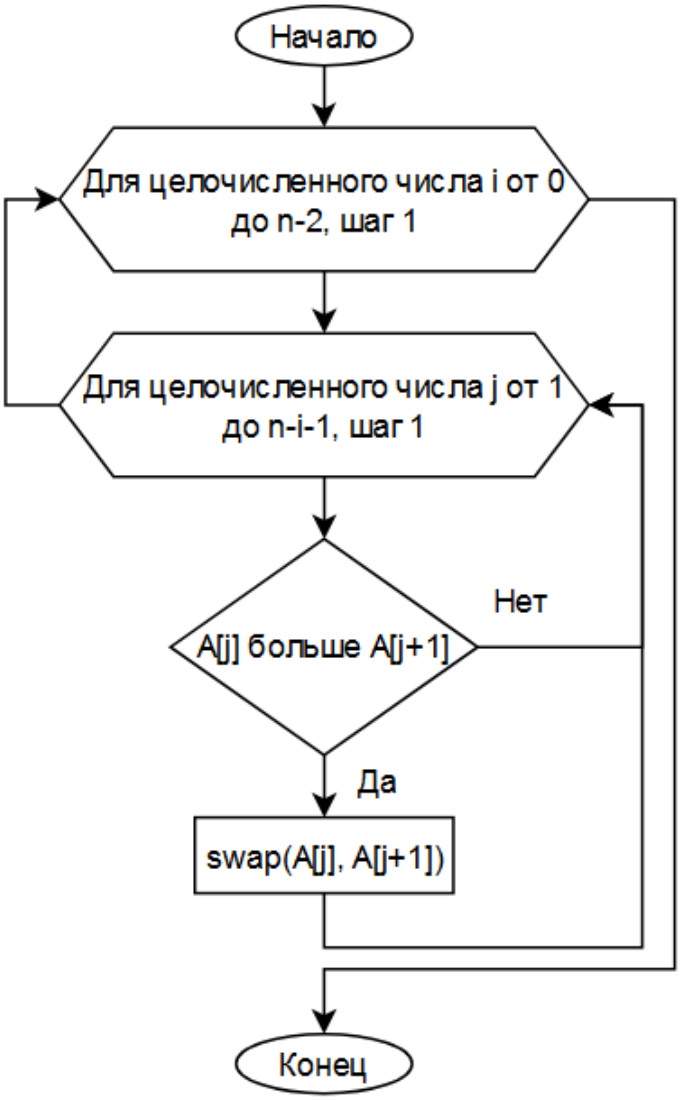


Рисунок 1 – Блок-схема алгоритма сортировки простыми вставками

2.2.2 Доказательство корректности циклов алгоритма сортировки простым обменом

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

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

Докажем конечность циклов:

Внешний цикл for проходит через все элементы массива начиная с первого и проходя n-1 раз. Внутренний цикл for проходит n/2 раз. Соседние значения

попарно сравниваются между собой и в случае, если предшествующий элемент больше последующего, меняются местами. После первого прохода всего массива элемент с максимальным значением займет место последнего элемента Следовательно, циклы данного алгоритма конечны.

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 простым обменом

Таблица 1-Псевдокод и анализ алгоритма сортировки вставками

№	Алгоритм, записанный на псевдокоде	Количество выполнений оператора
1	function ExchangeSort(a):	
2	for i ← 0 to (n - 2) do	n
3	for j ← 1 to (n - 1 - i) do	$\sum^{n-1}_{j=1} t_j = n + n - 1 + \cdots + 2 = 0.5n^2 - 1.5n + 1$
4	if (a[j] > a[j + 1]) then	$\sum^{n-1}_{j=1} t_j - 1 = 0.5n^2 - 2.5n + 2$
5	swap(a[j], a[j + 1])	$3*\sum^{n-1}_{j=1}(t_j - 1) = 1.5n^2 - 7.5n + 6$
6	endif	
7	od	
8	od	
9	}	

а. 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 простым обменом на языке C++

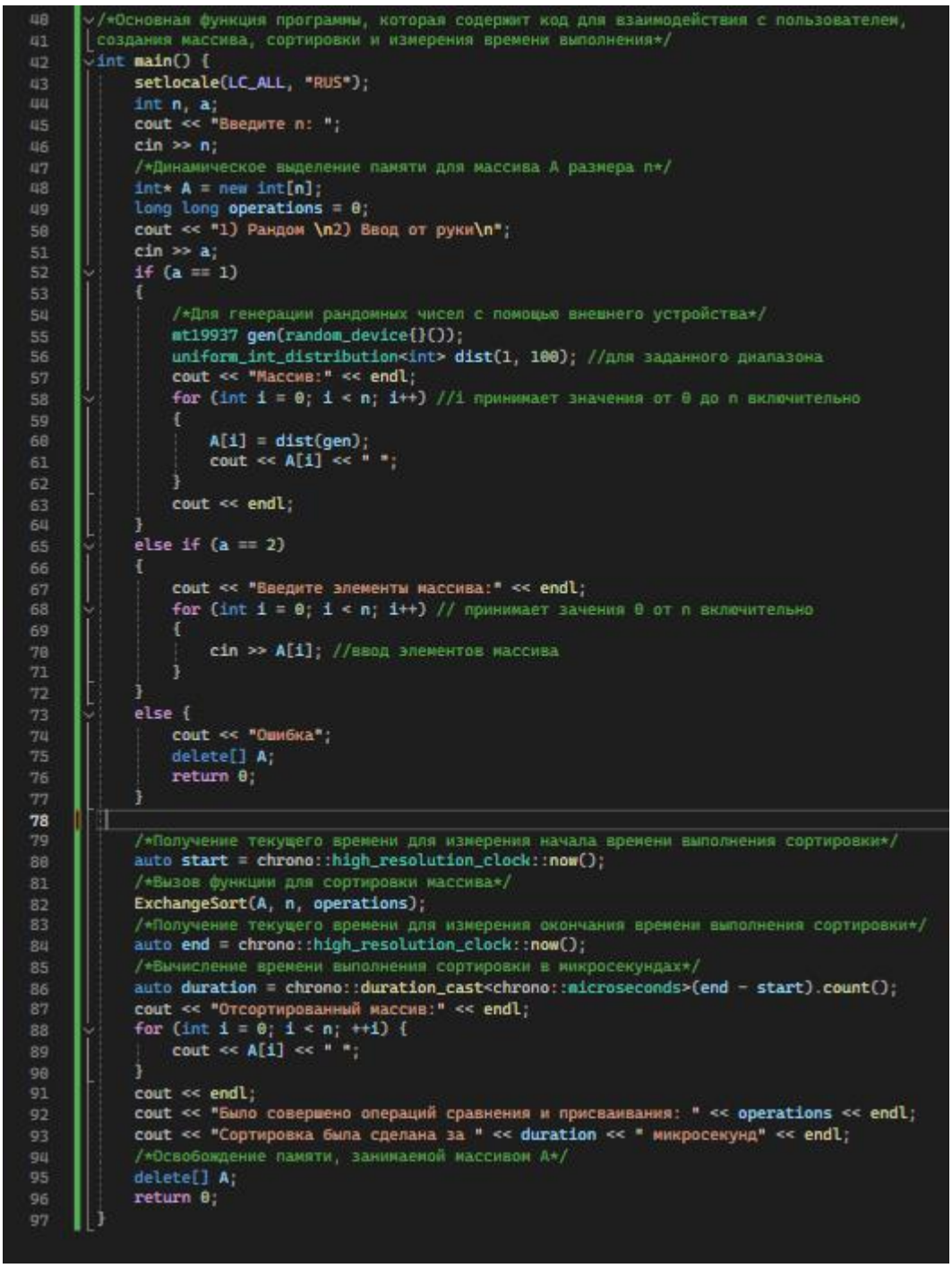
Реализуем данный алгоритм на языке C++(рис.2,3). Для реализации понадобятся такие библиотеки, как iostream, random, chrono. Iostream — это заголовочный файл с классами, функциями и переменными для организации ввода-вывода в языке программирования C++.Random - позволяет генерировать случайные числа в 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, которая представляет собой целое число в диапазоне от −9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт в памяти.

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


/*Объявление функции swap, которая меняет местами значения двух переменных типа int*/
void swap(int& a, int& b)
{
    setlocale(LC_ALL, "RUS");
    int temp = a;
    a = b;
    b = temp;
}


/*Объявление функции ExchangeSort, реализующей сортировку обменом (пузырьковую сортировку)*/
/* В функции ExchangeSort реализуется алгоритм пузырьковой сортировки,
который сравнивает и меняет местами соседние элементы массива*/
void ExchangeSort(int A[], int n, long long& operation) //A - массив, n - размер массива
{
    for (int i = 0; i < n - 1; ++i)
    {
        /*Вложенный цикл, который проходит по элементам массива
        до текущего конца отсортированной части*/
        for (int j = 0; j < n - i - 1; ++j)
        {
            if (A[j] > A[j + 1]) //сравнение элементов массива
            {
                swap(A[j], A[j + 1]); //перестановка элементов массмва
                operation++;
            }
            operation += 2;
        }
        operation += 2;
    }
    operation++;
}
```

Рисунок 2 – Программа алгоритма сортировки простым обменом



```cpp
/*Основная функция программы, которая содержит код для взаимодействия с пользователем,
создания массива, сортировки и измерения времени выполнения*/
int main() {
    setlocale(LC_ALL, "RUS");
    int n, a;
    cout << "Введите n: ";
    cin >> n;
    /*Динамическое выделение памяти для массива A размера n*/
    int* A = new int[n];
    long long operations = 0;
    cout << "1) Рандом \n2) Ввод от руки\n";
    cin >> a;
    if (a == 1)
    {
        /*Для генерации рандомных чисел с помощью внешнего устройства*/
        mt19937 gen(random_device{}());
        uniform_int_distribution<int> dist(1, 100); //для заданного диапазона
        cout << "Массив:" << endl;
        for (int i = 0; i < n; i++) //i принимает значения от 0 до n включительно
        {
            A[i] = dist(gen);
            cout << A[i] << " ";
        }
        cout << endl;
    }
    else if (a == 2)
    {
        cout << "Введите элементы массива:" << endl;
        for (int i = 0; i < n; i++) // принимает зачения 0 от n включительно
        {
            cin >> A[i]; //ввод элементов массива
        }
    }
    else {
        cout << "Ошибка";
        delete[] A;
        return 0;
    }

    /*Получение текущего времени для измерения начала времени выполнения сортировки*/
    auto start = chrono::high_resolution_clock::now();
    /*Вызов функции для сортировки массива*/
    ExchangeSort(A, n, operations);
    /*Получение текущего времени для измерения окончания времени выполнения сортировки*/
    auto end = chrono::high_resolution_clock::now();
    /*Вычисление времени выполнения сортировки в микросекундах*/
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start).count();
    cout << "Отсортированный массив:" << endl;
    for (int i = 0; i < n; ++i) {
        cout << A[i] << " ";
    }
    cout << endl;
    cout << "Было совершено операций сравнения и присваивания: " << operations << endl;
    cout << "Сортировка была сделана за " << duration << " микросекунд" << endl;
    /*Освобождение памяти, занимаемой массивом A*/
    delete[] A;
    return 0;
}
```

Рисунок 3 – Функция main для алгоритма сортировки простым обменом

2.3.2 Тестирование

Стоит задача протестировать программу с заданным размером массива n=10 (рис.4), 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 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.

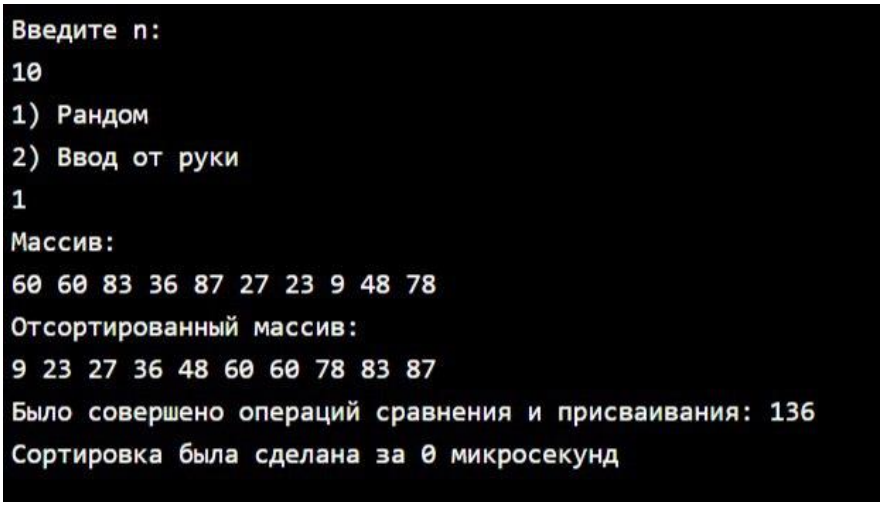


Рисунок 4 - Тестирование программы при n=10

Таблица 2. Сводная таблица результатов

n	**T(n), мс**	**$T_т$=C+M**	**$T_п$=$C_п$+$M_п$**
100	0.05	-	12236
1000	4.526	-	1234682
10000	394.969	-	122374417
100000	33726.936	-	12254127887
1000000	4215867.28	-	1270495346447

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).

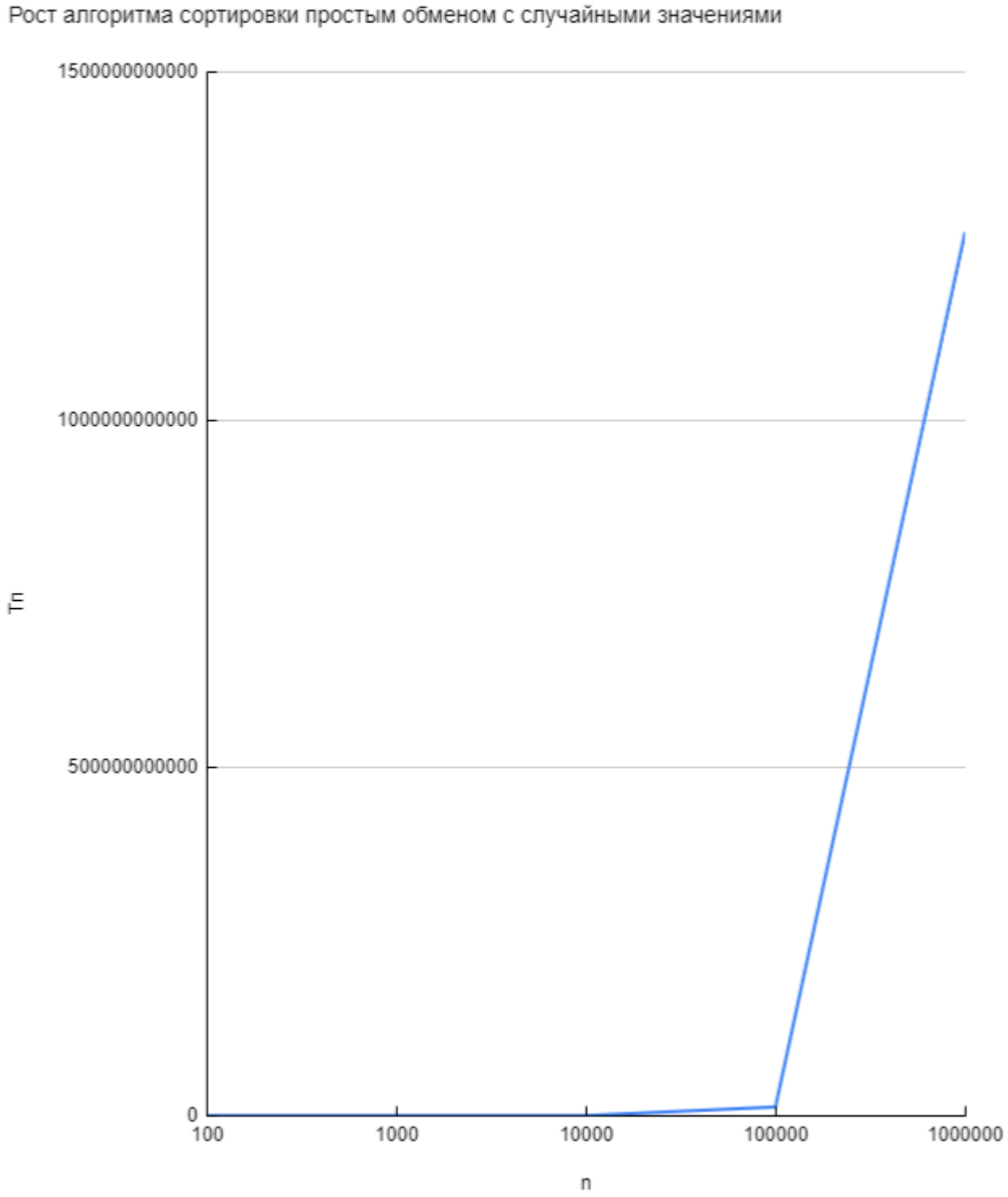


Рисунок 5 - График функции роста $T_п$ этого алгоритма от размера массива n

2.4 Вывод по заданию №1

На основе полученных данных тестирования и вычислительной сложности алгоритма, можно сделать вывод, что алгоритм сортировки простым обменом 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 обменом 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. Добавим в программу функцию, в которой проведем сортировку массива по убыванию (рис.6). В функцию main добавим вызов функции сортировки по убыванию (рис.7) и продемонстрируем работу программы при n=10 (рис.8). Алгоритм сортировки простым обменом не изменяется и соответствует продемонстрированному на рисунке 2.

```
/*По убыванию*/
void ReversedSort(int A[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (A[i] < A[j])
            {
                swap(A[i], A[j]);
            }
        }
    }
}
```

Рисунок 6 – Функция сортировки по убыванию

```
/*Основная функция программы, которая содержит код для взаимодействия с пользователем,
создания массива, сортировки и измерения времени выполнения*/
int main() {
    setlocale(LC_ALL, "RUS");
    int n, a;
    cout << "Введите n: ";
    cin >> n;
    /*Динамическое выделение памяти для массива A размера n*/
    int* A = new int[n];
    long long operations = 0;
    cout << "1) Рандом \n2) Ввод от руки\n";
    cin >> a;
    if (a == 1)
    {
        /*Для генерации рандомных чисел с помощью внешнего устройства*/
        mt19937 gen(random_device{}());
        uniform_int_distribution<int> dist(1, 100); //для заданного диапазона
        cout << "Массив:" << endl;
        for (int i = 0; i < n; i++) //i принимает значения от 0 до n включительно
        {
            A[i] = dist(gen);
            cout << A[i] << " ";
        }
        cout << endl;
    }
    else if (a == 2)
    {
        cout << "Введите элементы массива:" << endl;
        for (int i = 0; i < n; i++) // принимает зачения 0 от n включительно
        {
            cin >> A[i]; //ввод элементов массива
        }
    }
    else {
        cout << "Ошибка";
        delete[] A;
        return 0;
    }
    ReversedSort(A, n);
    /*Получение текущего времени для измерения начала времени выполнения сортировки*/
    auto start = chrono::high_resolution_clock::now();
    /*Вызов функции для сортировки массива*/
    ExchangeSort(A, n, operations);
    /*Получение текущего времени для измерения окончания времени выполнения сортировки*/
    auto end = chrono::high_resolution_clock::now();
    /*Вычисление времени выполнения сортировки в микросекундах*/
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start).count();
    cout << "Отсортированный массив:" << endl;
    for (int i = 0; i < n; ++i) {
        cout << A[i] << " ";
    }
    cout << endl;
    cout << "Было совершено операций сравнения и присваивания: " << operations << endl;
    cout << "Сортировка была сделана за " << duration << " микросекунд" << endl;
    /*Освобождение памяти, занимаемой массивом A*/
    delete[] A;
    return 0;
}
```

Рисунок 7 – Тестирование программы при n=10 и с отсортированными значениями по убыванию

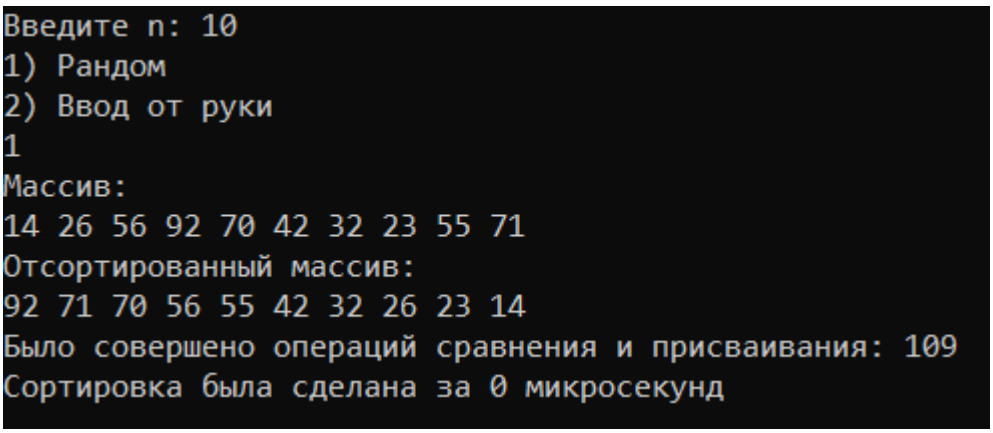


Рисунок 8 – Результаты тестирования программы при n=10 и с отсортированными значениями по убыванию

Так как значения идут в строго убывающем порядке, то можно сделать вывод, что данная ситуация являться худшим случаем, а 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	0.058	-	14571
1000	3.308	-	1450808
10000	428.769	-	145003739
100000	29225.886	-	14500056019
1000000	3215857.298	-	1450000500879

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

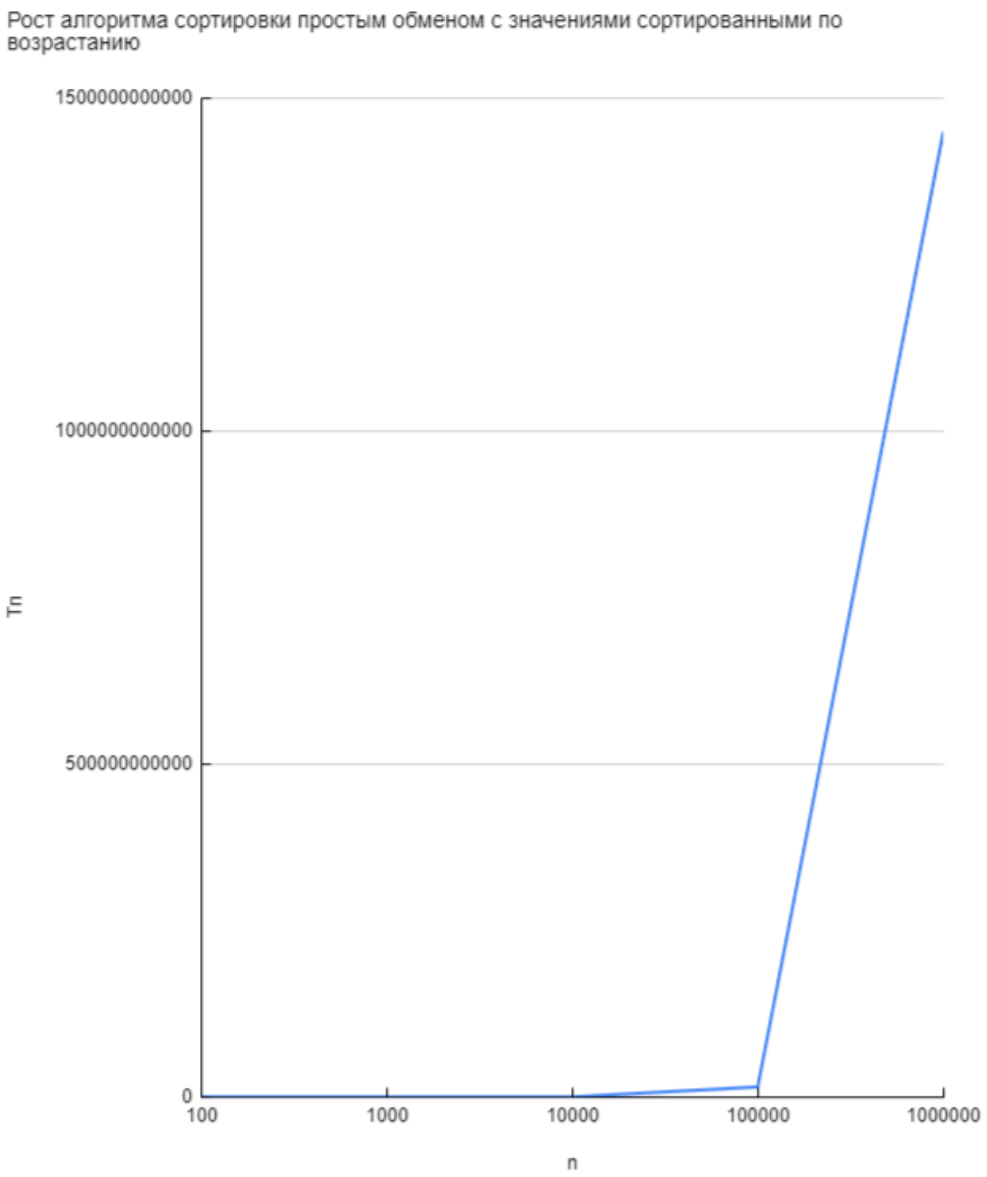


Рисунок 9 - График функции роста $T_п$ алгоритма сортировки простыми вставками с отсортированными значениями по убыванию от размера массива n

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 убывающем порядке. Добавим в программу функцию, в которой проведем сортировку массива по возрастанию(рис.10). В функцию main добавим вызов функции сортировки по убыванию (рис.11) и продемонстрируем работу программы при n=10 (рис.12). Алгоритм сортировки простым обменом не изменяется и соответствует продемонстрированному на рисунке 2.

```
16  void Sort(int A[], int n)
17  {
18      for (int i = 0; i < n - 1; i++)
19      {
20          for (int j = i + 1; j < n; j++)
21          {
22              if (A[i] > A[j])
23              {
24                  swap(A[i], A[j]);
25              }
26          }
27      }
28  }
```

Рисунок 10 – Функция сортировки по убыванию

```cpp
/*Основная функция программы, которая содержит код для взаимодействия с пользователем,
создания массива, сортировки и измерения времени выполнения*/
int main() {
    setlocale(LC_ALL, "RUS");
    int n, a;
    cout << "Введите n: ";
    cin >> n;
    /*Динамическое выделение памяти для массива A размера n*/
    int* A = new int[n];
    long long operations = 0;
    cout << "1) Рандом \n2) Ввод от руки\n";
    cin >> a;
    if (a == 1)
    {
        /*Для генерации рандомных чисел с помощью внешнего устройства*/
        mt19937 gen(random_device{}());
        uniform_int_distribution<int> dist(1, 100); //для заданного диапазона
        cout << "Массив:" << endl;
        for (int i = 0; i < n; i++) //i принимает значения от 0 до n включительно
        {
            A[i] = dist(gen);
            cout << A[i] << " ";
        }
        cout << endl;
    }
    else if (a == 2)
    {
        cout << "Введите элементы массива:" << endl;
        for (int i = 0; i < n; i++) // принимает зачения 0 от n включительно
        {
            cin >> A[i]; //ввод элементов массива
        }
    }
    else {
        cout << "Ошибка";
        delete[] A;
        return 0;
    }
    Sort(A, n);
    /*Получение текущего времени для измерения начала времени выполнения сортировки*/
    auto start = chrono::high_resolution_clock::now();
    /*Вызов функции для сортировки массива*/
    ExchangeSort(A, n, operations);
    /*Получение текущего времени для измерения окончания времени выполнения сортировки*/
    auto end = chrono::high_resolution_clock::now();
    /*Вычисление времени выполнения сортировки в микросекундах*/
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start).count();
    cout << "Отсортированный массив:" << endl;
    for (int i = 0; i < n; ++i) {
        cout << A[i] << " ";
    }
    cout << endl;
    cout << "Было совершено операций сравнения и присваивания: " << operations << endl;
    cout << "Сортировка была сделана за " << duration << " микросекунд" << endl;
    /*Освобождение памяти, занимаемой массивом A*/
    delete[] A;
    return 0;
}
```

Рисунок 11 – Тестирование программы при n=10 и с отсортированными значениями по возрастанию

```
Введите n: 10
1) Рандом
2) Ввод от руки
1
Массив:
100 9 73 74 89 18 82 92 1 2
Отсортированный массив:
1 2 9 18 73 74 82 89 92 100
Было совершено операций сравнения и присваивания: 109
Сортировка была сделана за 0 микросекунд
```

Рисунок 12 – 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, 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	0.016	-	10099
1000	1.684	-	1000999
10000	708.394	-	100009999
100000	15378.269	-	10000099999
1000000	7219328.269	-	1000000999999

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.13).

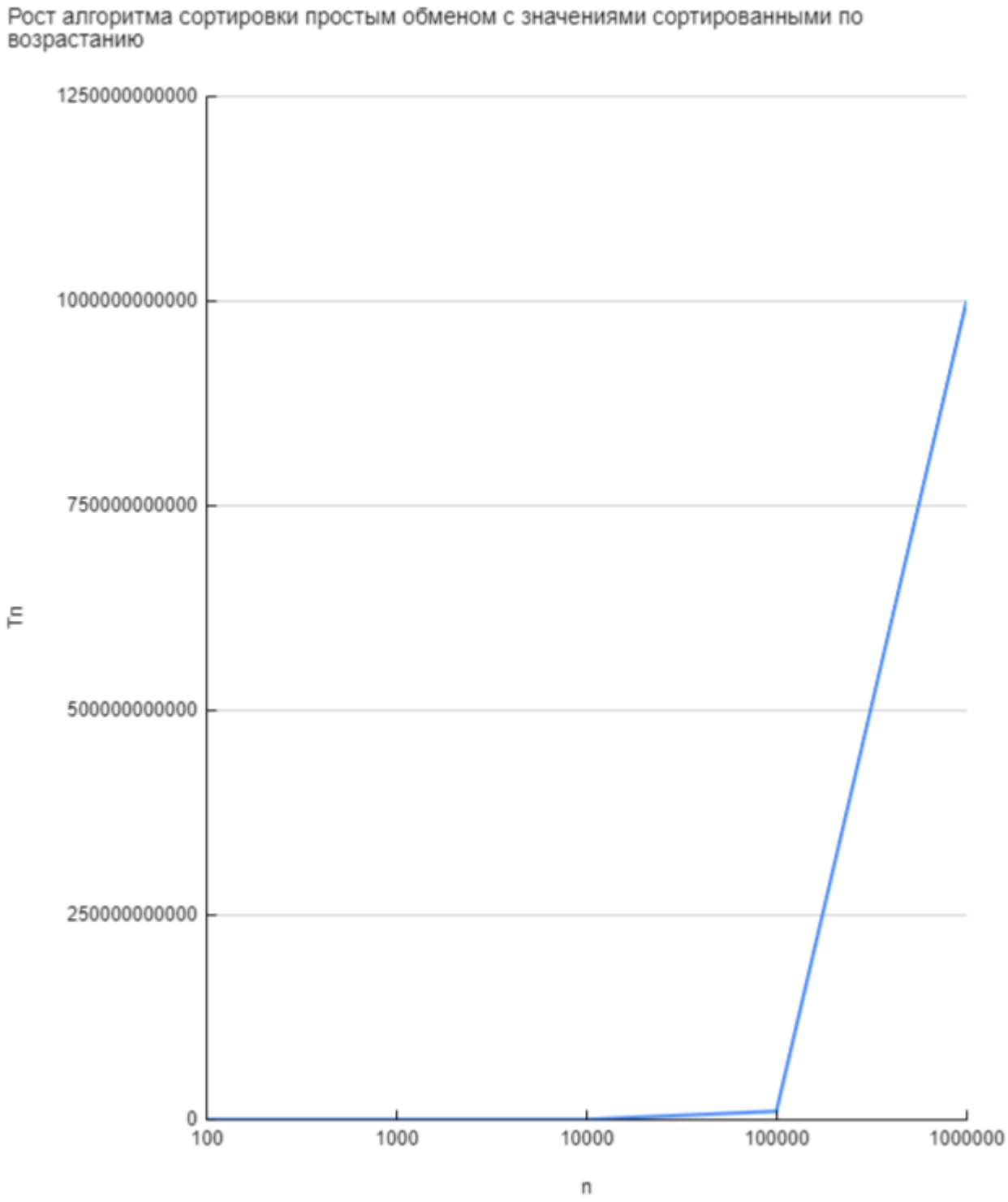


Рисунок 13 - График функции роста $T_п$ алгоритма сортировки простыми вставками с отсортированными значениями по возрастанию от размера массива n

3.3 Вывод по заданию №2

Алгоритм сортировки простым обменом (также известный как пузырьковая сортировка) зависит от исходной упорядоченности массива. Если массив уже отсортирован, то время выполнения алгоритма будет минимальным, так как элементы не будут менять свои позиции. Однако, если массив отсортирован в обратном порядке, то алгоритм будет работать наихудшим образом, так как придется делать максимальное количество обменов.

Таким образом, можно сделать вывод, что алгоритм сортировки простым обменом зависит от исходной упорядоченности массива и его эффективность может значительно изменяться в зависимости от этого фактора.

4 ЗАДАНИЕ №3

4.1 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.

4.2 Математическая модель решения алгоритма

4.2.1 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.14).

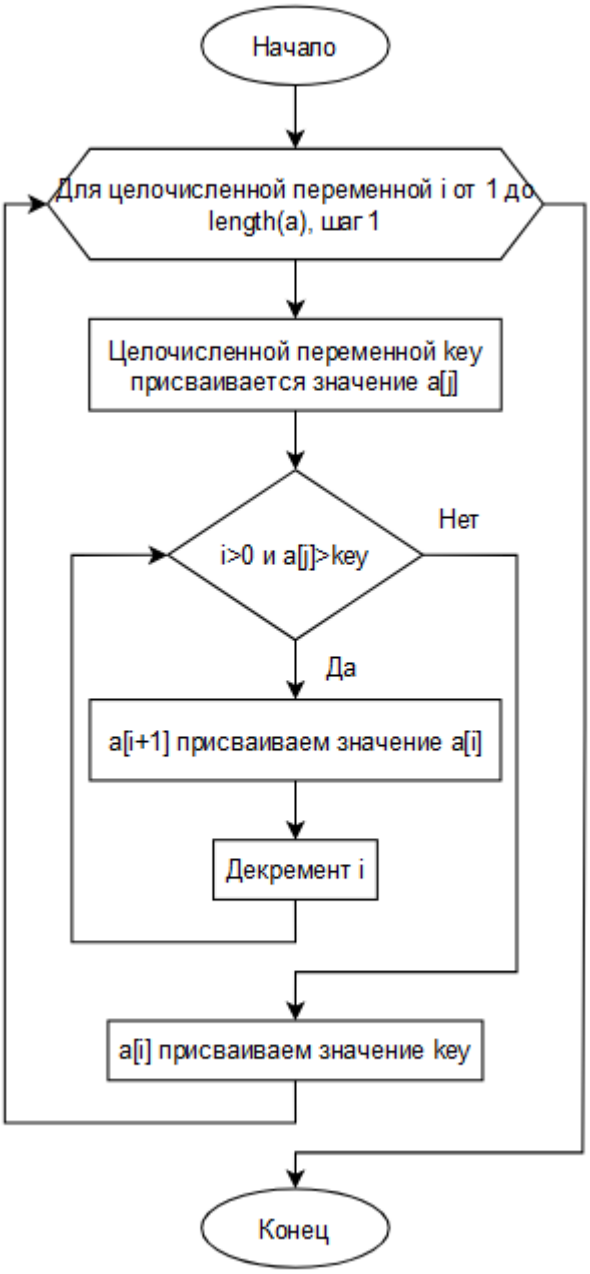


Рисунок 14 – Блок-схема алгоритма сортировки простыми вставками

4.2.2 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.

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

Таблица 5-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).

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

4.3.1 Реализация алгоритма сортировки простыми вставками на языке C++

Реализуем данный алгоритм на языке C++(рис.15,16). Для реализации понадобятся такие библиотеки, как iostream, vector и random, chrono. Iostream — библиотека iostream, которая позволяет вводить и выводить данные. Vector — это шаблон класса для контейнеров последовательности. Вектор хранит элементы заданного типа в линейном расположении и обеспечивает быстрый случайный доступ к любому элементу. Random - библиотека random, которая предоставляет генераторы случайных чисел. Chrono - библиотека chrono, которая предоставляет возможности для измерения времени. Для подсчёта количество операций присваивания или сравнения введём переменную operations, которая представляет собой целое число в диапазоне от -2 147 483 648 до 2 147 483 648 и занимает 4 байта в памяти.

```cpp
#include <iostream>
#include <vector>
#include <random>
#include <chrono>
using namespace std;
void InsertionSort(vector<int>& values, long& operations)
{
    setlocale(LC_ALL, "RUS");
    for (size_t i = 1; i < values.size(); ++i) {
        int x = values[i];
        size_t j = i;
        while (j > 0 && values[j - 1] > x) {
            values[j] = values[j - 1];
            --j;
            operations += 3; // увеличиваем операции на 3 (сравнение при входе в цикл,присваивание и декремент)
        }
        values[j] = x;
        operations += 5; // увеличиваем операции на 5 (сравнение при входе в цикл, присваивание 3 раза, сравнение при выходе из цикла)
    }
    ++operations; // увеличиваем операции на 1 (сравнение при выходе из цикла)
}
```

Рисунок 15 – Программа алгоритма сортировки простыми вставками

```cpp
int main()
{
    setlocale(LC_ALL, "RUS");
    long operations = 0;
    int n, a;
    cout << "Введите n: ";
    cin >> n;
    vector<int> A(n);
    cout << "1) Рандом \n2) От руки\n";
    cin >> a;
    if (a == 1)
    {
        /*Для генерации рандомных чисел с помощью внешнего устройства*/
        mt19937 gen(random_device{}());
        uniform_int_distribution<int> dist(1, 10);
        cout << "Массив:" << endl;
        for (int i = 0; i < n; i++) { // i принимает значения от 0 до n включительно
            A[i] = dist(gen);
            cout << A[i] << " ";
        } cout << endl;
    }
    else if (a == 2) {// i принимает значения от 0 до n включительно
        cout << "Введите элементы массива:" << endl;
        for (int i = 0; i < n; i++) {
            cin >> A[i];
            cout << A[i] << " ";
        }cout << endl;
    }
    else {
        cout << "Ошибка";
        return 0;
    }
    /*Получение текущего времени для измерения начала времени выполнения сортировки*/
    auto start = chrono::high_resolution_clock::now();
    /*Вызов функции для сортировки массива*/
    InsertionSort(A, operations);
    /*Получение текущего времени для измерения окончания времени выполнения сортировки*/
    auto end = chrono::high_resolution_clock::now();
    /*Вычисление времени выполнения сортировки в микросекундах*/
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start).count();
    cout << "Отсортированный массив:\n";//вывод массива
    for (int i = 0; i < n; ++i) {
        cout << A[i] << " ";
    } cout << endl;
    cout << "Было совершено операций сравнения и присваивания: " << operations << endl;
    cout << "Сортировка была сделана за " << duration << " микросекунд" << endl;
    return 0;
}
```

Рисунок 16 – Функция main для алгоритма сортировки простыми вставками

4.3.2 Тестирование

Стоит задача протестировать программу с заданным размером массива n=10 (рис.17), 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

времени, которое было потрачено на сортировку. Для более точных результатов в программе будем рассматривать микросекунды, которые потом, для заполнения таблицы, будут переведены в миллисекунды.

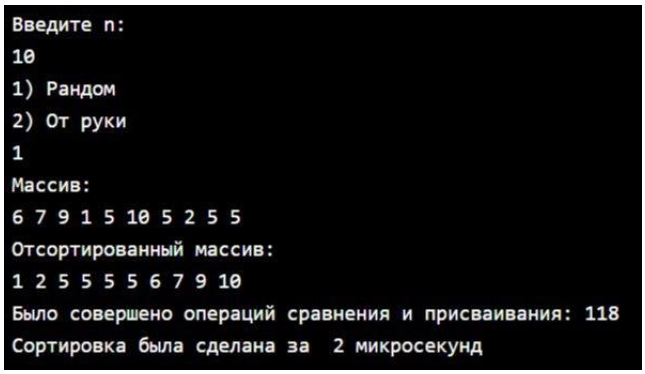


Рисунок 17 - Тестирование программы при n=10

Таблица 6. Сводная таблица результатов

n	**T(n), мс**	**$T_т$=C+M**	**$T_п$=$C_п$+$M_п$**
100	0.03	-	6382
1000	1.86	-	694861
10000	197.22	-	67770571
100000	14275.56	-	6752008879
1000000	1033321.23	-	668448879021

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.18).

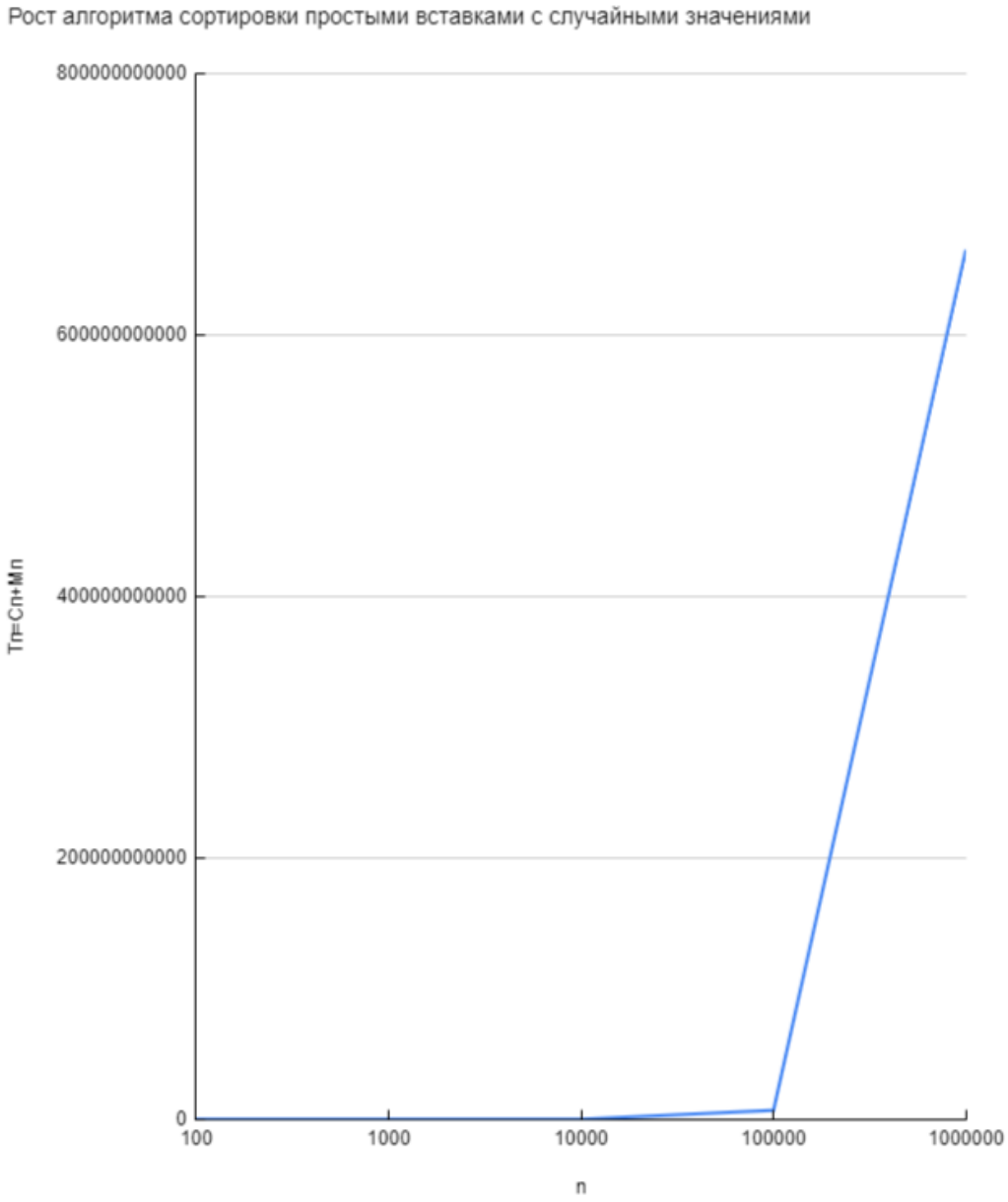


Рисунок 18 - График функции роста $T_п$ этого алгоритма от размера массива n

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 убывающем

порядке. Изменим программу в функции main, чтобы значения элементов массива сортировались в убывающем порядке (рис.20) и продемонстрируем работу программы при n=10 (рис.21). Для сортировки значений добавим библиотеку algorithm (рис.19). 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 15.

```cpp
#include <iostream>
#include <vector>
#include <random>
#include <chrono>
#include <algorithm>
```

Рисунок 19 – Использованные библиотеки

```cpp
int main()
{
    setlocale(LC_ALL, "RUS");
    long operations = 0;
    int n, a;
    cout << "Введите n: ";
    cin >> n;
    vector<int> A(n);
    cout << "1) Рандом \n2) От руки\n";
    cin >> a;
    if (a == 1)
    {
        /*Для генерации рандомных чисел с помощью внешнего устройства*/
        mt19937 gen(random_device{}());
        uniform_int_distribution<int> dist(1, 10);
        cout << "Массив:" << endl;
        for (int i = 0; i < n; i++) { // i принимает значения от 0 до n включительно
            A[i] = dist(gen);
            cout << A[i] << " ";
        } cout << endl;
    }
    else if (a == 2) {// i принимает значения от 0 до n включительно
        cout << "Введите элементы массива:" << endl;
        for (int i = 0; i < n; i++) {
            cin >> A[i];
            cout << A[i] << " ";
        }cout << endl;
    }
    else {
        cout << "Ошибка";
        return 0;
    }
    sort(A.begin(), A.end(), greater<int>()); //по убыванию
    /*Получение текущего времени для измерения начала времени выполнения сортировки*/
    auto start = chrono::high_resolution_clock::now();
    /*Вызов функции для сортировки массива*/
    InsertionSort(A, operations);
    /*Получение текущего времени для измерения окончания времени выполнения сортировки*/
    auto end = chrono::high_resolution_clock::now();
    /*Вычисление времени выполнения сортировки в микросекундах*/
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start).count();
    cout << "Отсортированный массив:\n";//вывод массива
    for (int i = 0; i < n; ++i) {
        cout << A[i] << " ";
    } cout << endl;
    cout << "Было совершено операций сравнения и присваивания: " << operations << endl;
    cout << "Сортировка была сделана за " << duration << " микросекунд" << endl;
    return 0;
}
```

Рисунок 20 – Тестирование программы при n=10 и с отсортированными значениями по убыванию

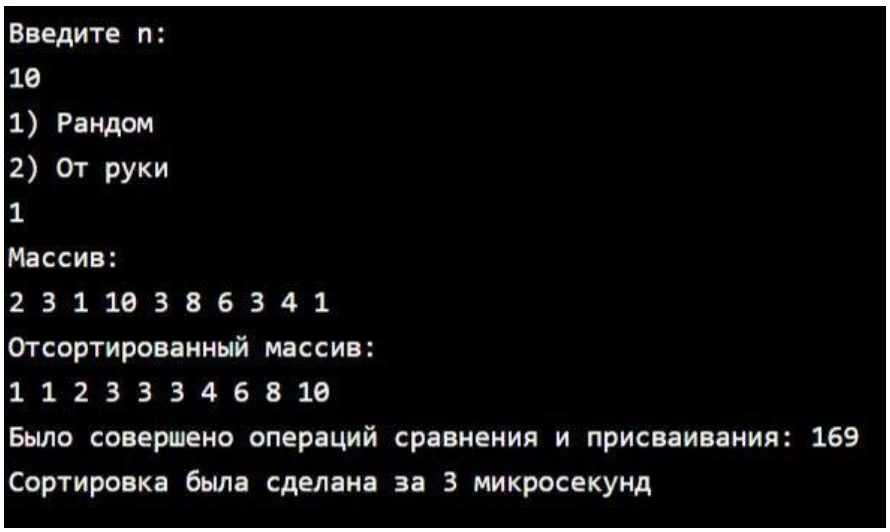


Рисунок 21 – Результаты тестирования программы при n=10 и с отсортированными значениями по убыванию

Так как значения идут в строго убывающем порядке, то можно сделать вывод, что данная ситуация являться худшим случаем, а следовательно имеет сложность $O(n^2)$. Значит, в худшем случае алгоритм является квадратичным. 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	0.055	-	13885
1000	6.2	-	1353475
10000	284.459	-	135043384
100000	28022.498	-	13500400477
1000000	2760539.811	-	1349646370321

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

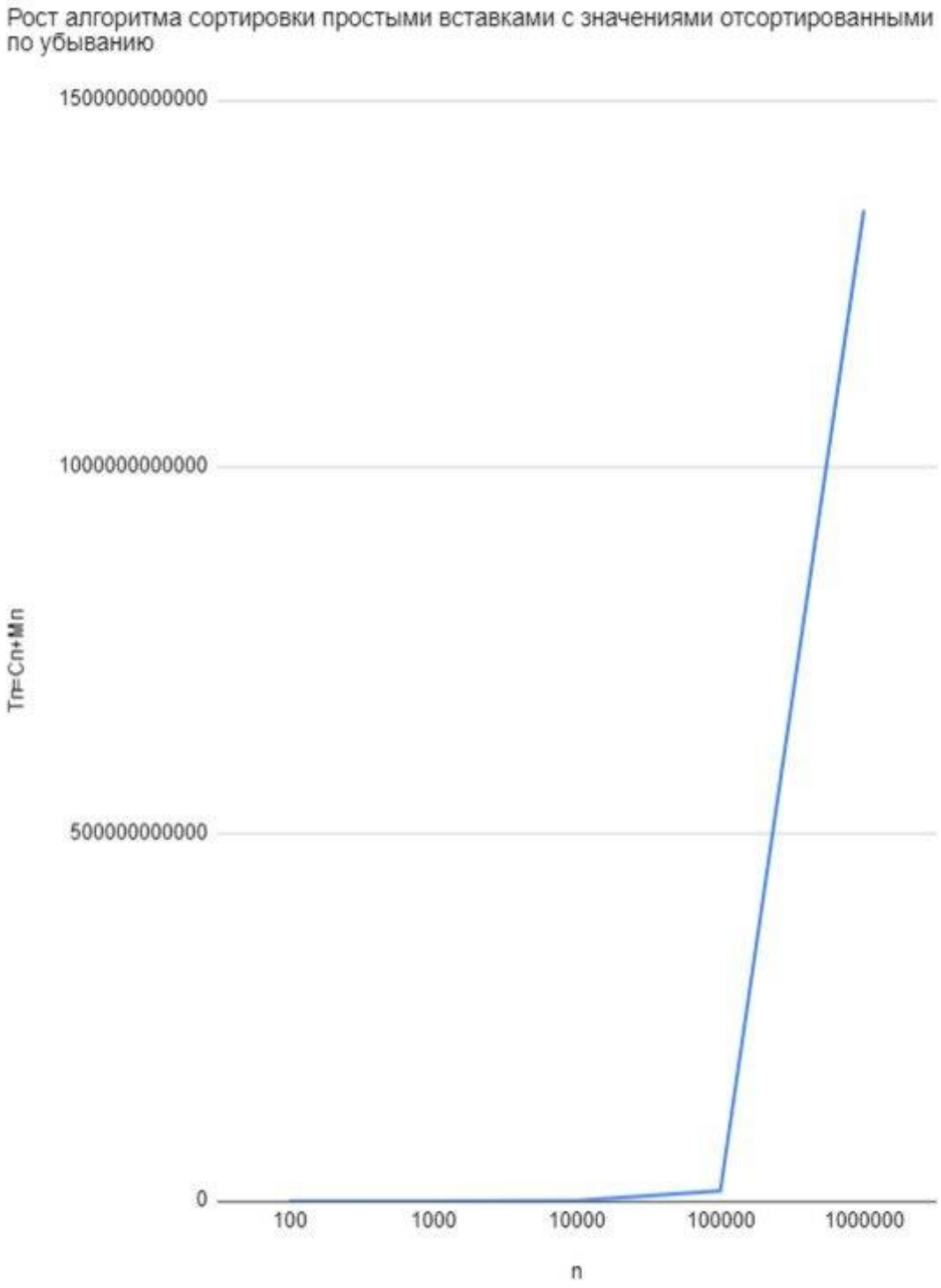


Рисунок 22 - График функции роста $T_п$ алгоритма сортировки простыми вставками с отсортированными значениями по убыванию от размера массива n

4.3.5 Тестирование при упорядоченном по возрастанию элементов массива и построение графика

Будет проведено тестирование программы на массивах при n = 100, 1000, 10000, 100000 и 1000000 элементов, которые отсортированы в возрастающем порядке. Изменим программу в функции main, чтобы значения элементов массива сортировались в возрастающем порядке (рис. 23) и продемонстрируем работу программы при n=10 (рис.24). Для сортировки значений добавим библиотеку algorithm (рис.19). 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 15.

```
int main()
{
    setlocale(LC_ALL, "RUS");
    long operations = 0;
    int n, a;
    cout << "Введите n: ";
    cin >> n;
    vector<int> A(n);
    cout << "1) Рандом \n2) От руки\n";
    cin >> a;
    if (a == 1)
    {
        /*Для генерации рандомных чисел с помощью внешнего устройства*/
        mt19937 gen(random_device{}());
        uniform_int_distribution<int> dist(1, 10);
        cout << "Массив:" << endl;
        for (int i = 0; i < n; i++) { // i принимает значения от 0 до n включительно
            A[i] = dist(gen);
            cout << A[i] << " ";
        } cout << endl;
    }
    else if (a == 2) {// i принимает значения от 0 до n включительно
        cout << "Введите элементы массива:" << endl;
        for (int i = 0; i < n; i++) {
            cin >> A[i];
            cout << A[i] << " ";
        }cout << endl;
    }
    else {
        cout << "Ошибка";
        return 0;
    }
    sort(A.begin(), A.end()); //по убыванию
    /*Получение текущего времени для измерения начала времени выполнения сортировки*/
    auto start = chrono::high_resolution_clock::now();
    /*Вызов функции для сортировки массива*/
    InsertionSort(A, operations);
    /*Получение текущего времени для измерения окончания времени выполнения сортировки*/
    auto end = chrono::high_resolution_clock::now();
    /*Вычисление времени выполнения сортировки в микросекундах*/
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start).count();
    cout << "Отсортированный массив:\n";//вывод массива
    for (int i = 0; i < n; ++i) {
        cout << A[i] << " ";
    } cout << endl;
    cout << "Было совершено операций сравнения и присваивания: " << operations << endl;
    cout << "Сортировка была сделана за " << duration << " микросекунд" << endl;
    return 0;
}
```

Рисунок 23 – Тестирование программы при n=10 и с отсортированными значениями по возрастанию

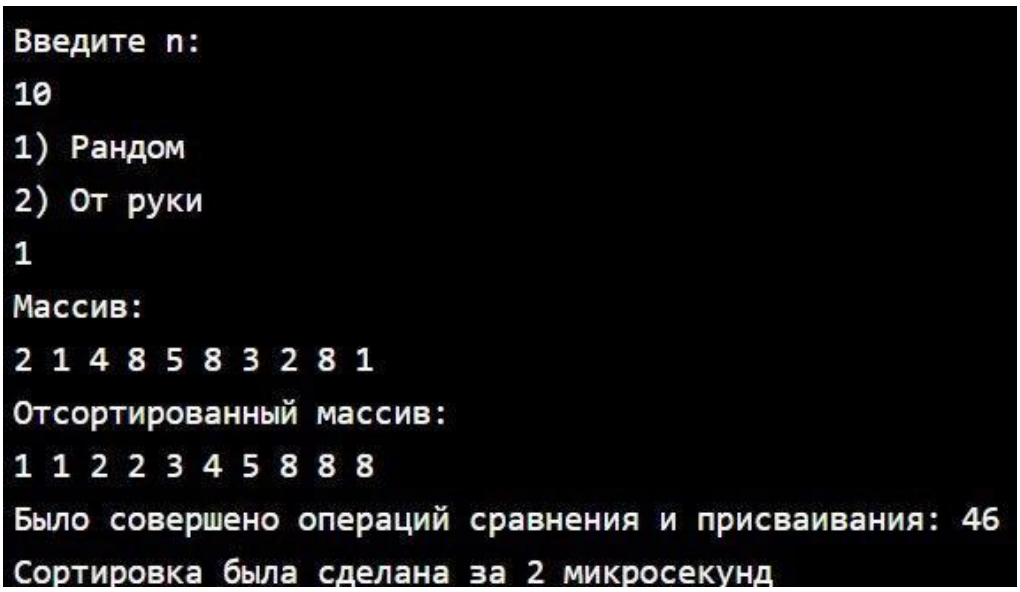


Рисунок 24 – Результаты тестирования программы при n=10 и с отсортированными значениями по возрастанию

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). Значит, в лучшем случае алгоритм будет линейным. 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	0.008	-	496
1000	0.021	-	4996
10000	0.096	-	49996
100000	0.951	-	499996
1000000	9.421	-	4999996

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.25).

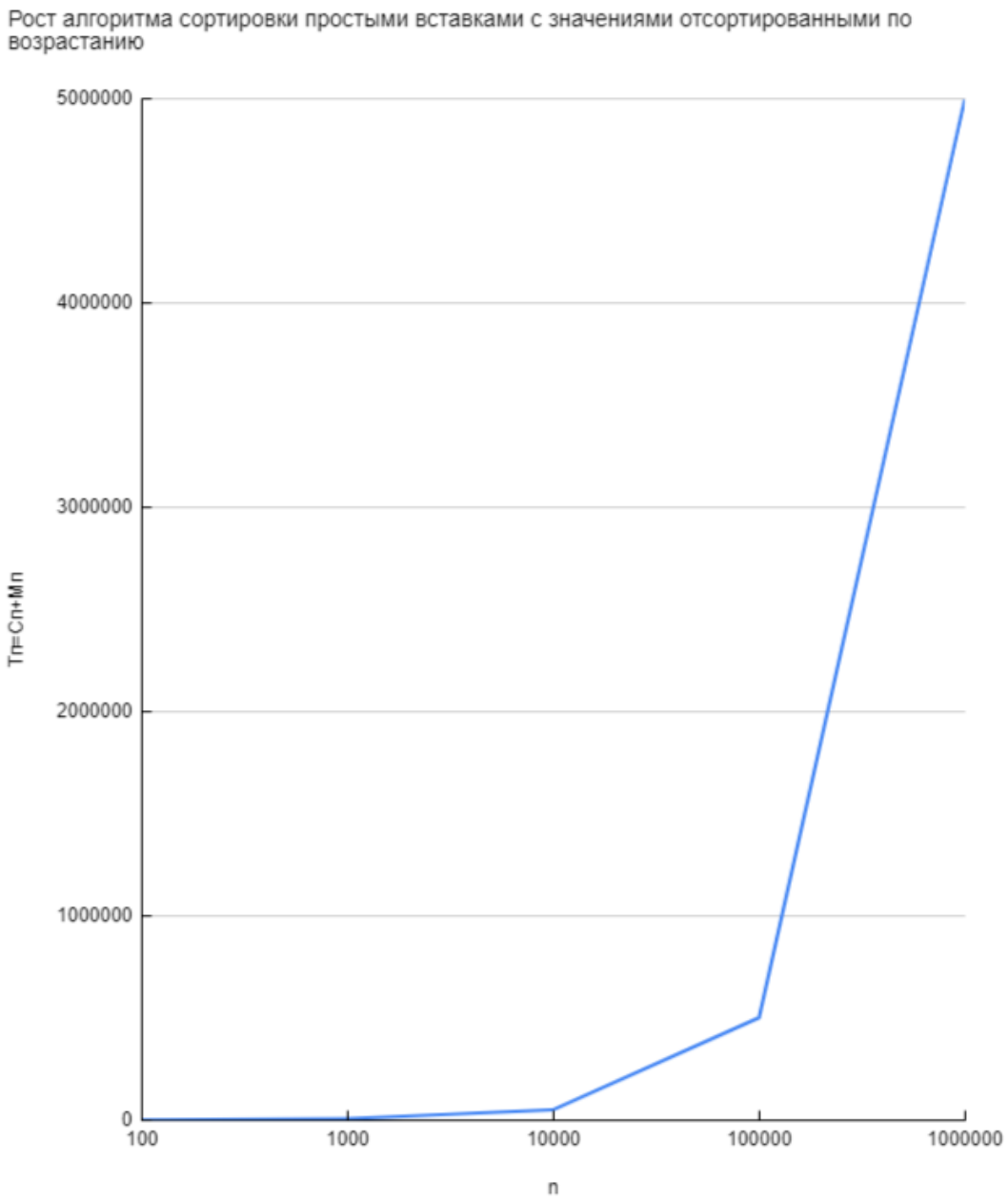


Рисунок 25 - График функции роста $T_п$ алгоритма сортировки простыми вставками с отсортированными значениями по возрастанию от размера массива n

4.4 Сравнение графиков двух алгоритмов сортировки из задания 1 и 3

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

На основании данных из таблицы 3 и графика алгоритма сортировки простым обменом в худшем случае(рис.9), и таблицы 7 и графика алгоритма сортировки простыми вставками в худшем случае(рис.22), мы создадим новый график для сравнения роста графиков(рис.26).

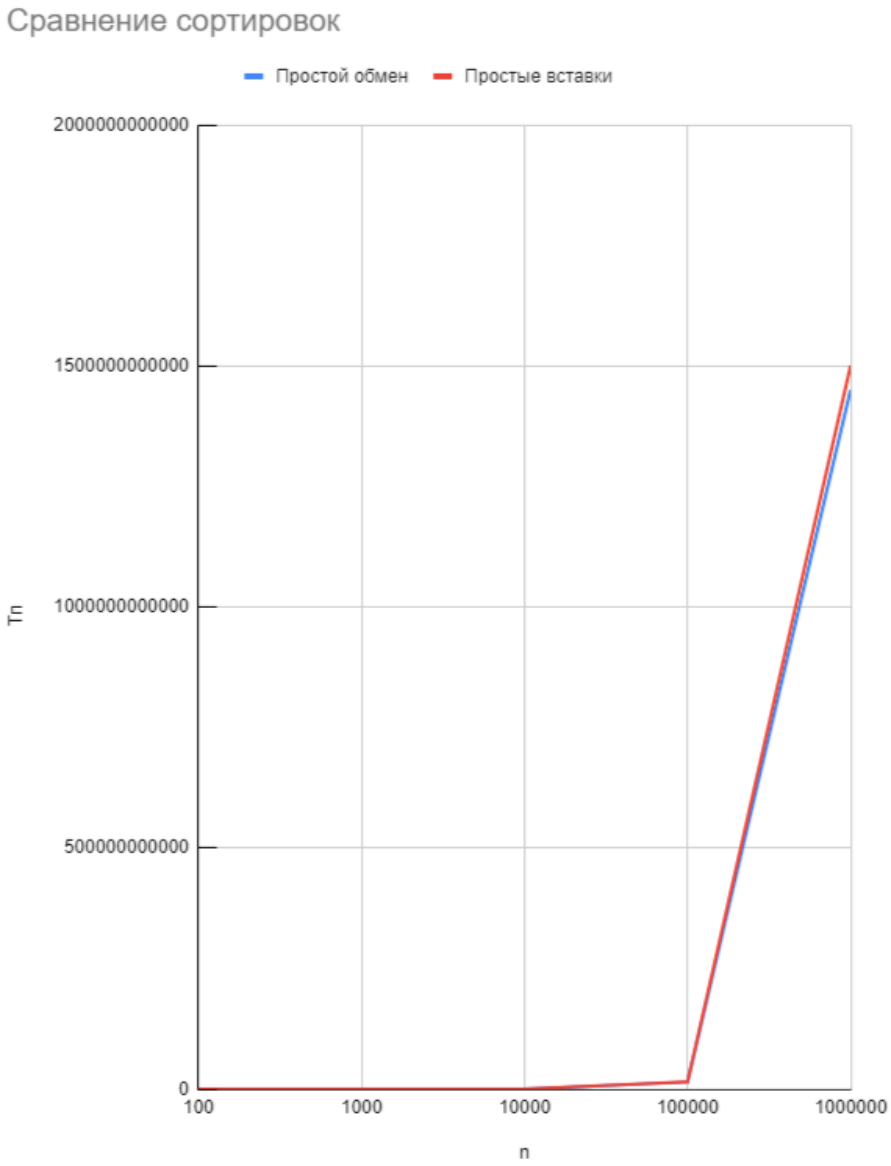


Рисунок 26 – График двух алгоритмов в худшем случае

На основе таблиц 3 и 7 и графика(рис.26), 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.12), и таблицы 8 и графика алгоритма сортировки простым обменом в лучшем случае(рис.25), мы создадим новый график для сравнения роста графиков(рис.27).

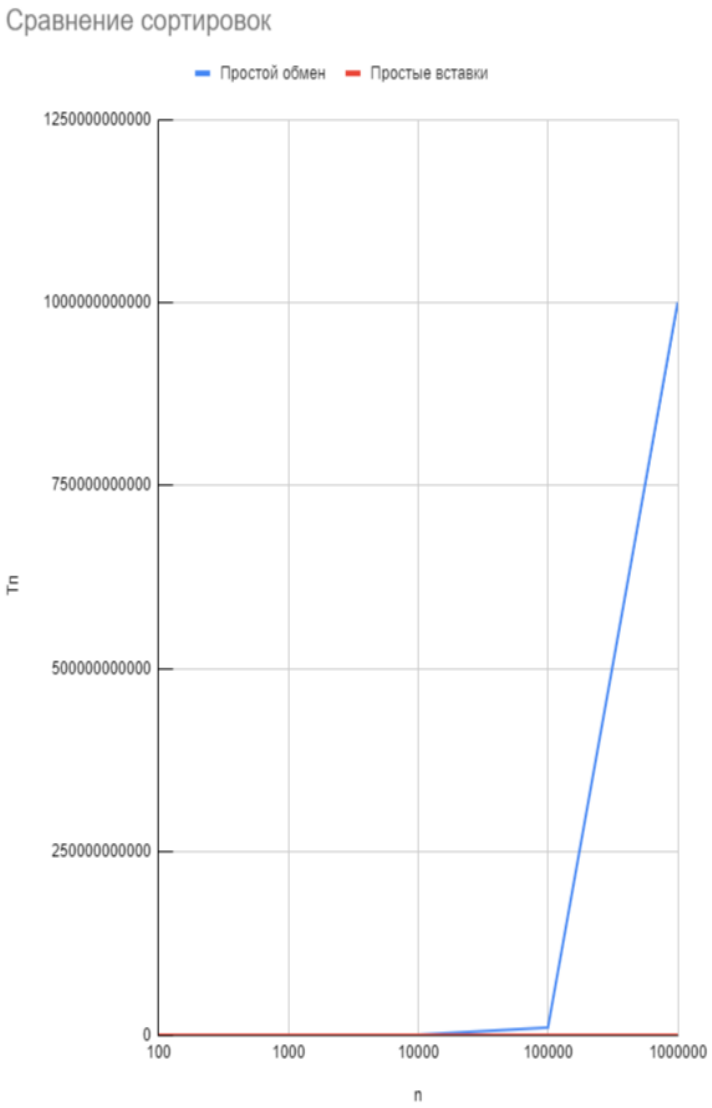


Рисунок 27 – График двух алгоритмов в лучшем случае

На основе таблиц 4 и 8 и графика(рис.27), можно сделать вывод, что в лучшем случае алгоритм сортировки простой вставкой более эффективный, чем алгоритм сортировки простого обмена.

4.5 Выводы по заданию №3

Алгоритм сортировки простым обменом имеет линейную временную сложность в лучшем случае O(n), но квадратичную сложность в среднем и худшем случае $O(n^2)$. Этот метод подходит для небольших объемов данных, но может быть медленным на больших наборах. Алгоритм сортировки простыми вставками также имеет квадратичную сложность $O(n^2)$ в худшем случае, но в лучшем случае она равна линейной O(n). Он также может быть медленным на больших массивах данных. Поэтому выбор конкретного алгоритма сортировки зависит от размера и структуры входных данных. В некоторых случаях сортировка простым обменом может быть эффективнее, в других - сортировка простыми вставками.

4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие сортировки называют простыми?

Простыми сортировками обычно называют такие методы сортировки, как сортировка пузырьком, сортировка вставками и сортировка выбором. Эти методы отличаются своей простотой и легкостью в реализации, но при этом имеют квадратичную сложность времени выполнения. Это означает, что время выполнения этих сортировок увеличивается квадратично с увеличением размера входных данных. Несмотря на их простоту, простые сортировки могут быть полезны для небольших массивов данных или в качестве базовых алгоритмов для дальнейших улучшений.

2. Что означает понятие «внутренняя сортировка»?

"Внутренняя сортировка" относится к сортировке элементов, находящихся в оперативной памяти компьютера. Это означает, что все данные, которые нужно отсортировать, уже находятся в доступной для алгоритма сортировки памяти. В отличие от внешней сортировки, которая предполагает использование внешних устройств хранения данных (например, жестких дисков), внутренняя сортировка работает исключительно с данными, которые могут быть загружены непосредственно в оперативную память. Основной преимуществом внутренней сортировки является скорость выполнения, так как доступ к данным происходит намного быстрее, чем при работе с внешними устройствами хранения. Таким образом, внутренние сортировки обычно используются для обработки небольших массивов данных или ситуаций, когда данные помещаются в оперативную память компьютера.

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

При оценке сложности алгоритма сортировки обычно выделяют несколько основных операций:

1) Сравнения: количество сравнений элементов, которые необходимо выполнить, чтобы правильно упорядочить данные. Сравнения являются одной из основных операций в большинстве алгоритмов сортировки.
2) Перестановки (обмены): количество операций обмена элементов местами, необходимых для упорядочивания данных. Это может включать перемещение элементов в массиве или списке.
3) Дополнительная память (потребление памяти): некоторые алгоритмы сортировки требуют дополнительной памяти для временного хранения данных или вспомогательных структур данных. Потребление дополнительной памяти может быть важным фактором при выборе алгоритма сортировки.
4) Время выполнения: общее время, необходимое для сортировки данных. Это также может включать в себя количество операций сравнения и обмена. Оценка сложности алгоритма сортировки учитывает сочетание этих операций для определения времени выполнения и затрат по памяти на конкретный алгоритм.

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

При оценке эффективности алгоритма сортировки используются следующие характеристики сложности:

1) Время выполнения: это основная характеристика, которая определяет скорость работы алгоритма сортировки. Время выполнения определяется количеством операций, необходимых для завершения сортировки данных, такими как сравнения и перестановки.

2)Относительная сложность: сравнение времени выполнения алгоритмов сортировки для разного размера входных данных или различных типов данных. Это помогает определить, какой алгоритм будет более эффективным в конкретной ситуации.

3)Потребление дополнительной памяти: некоторые алгоритмы сортировки требуют дополнительной памяти для временного хранения данных или вспомогательных структур. Потребление памяти может быть важным фактором при оценке эффективности и выборе алгоритма.

4) Устойчивость: некоторые алгоритмы сортировки сохраняют относительный порядок равных элементов, что называется устойчивостью сортировки. Устойчивость может быть важным критерием при работе с данными, в которых необходимо сохранить порядок элементов с одинаковым значением.

Оценка эффективности алгоритма сортировки включает анализ вышеперечисленных характеристик для выбора подходящего алгоритма в зависимости от задачи и ситуации.

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

1) Простой обмен (сортировка пузырьком):

- - Вычислительная сложность: $O(n^2)$ в среднем и худшем случаях, где n - количество элементов в массиве.
- - Емкостная сложность: O(1), поскольку алгоритм не требует дополнительной памяти для сортировки.

2) Простая вставка:

- - Вычислительная сложность: В среднем случае $O(n^2)$, в лучшем случае O(n), где n - количество элементов в массиве.
- - Емкостная сложность: O(1), поскольку не требуется дополнительной памяти для сортировки.

3) Простой выбор:

- - Вычислительная сложность: O(n^2) в среднем и худшем случаях, где n - количество элементов в массиве.

- - Емкостная сложность: O(1), поскольку алгоритм не требует дополнительной памяти для сортировки.

6. Какую роль в сортировке обменом играет условие Айверсона?

Условие Айверсона в сортировке пузырьком указывает на завершение сортировки, если на текущем проходе не было перестановок, что помогает ускорить алгоритм. 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.

Вычислительная теоретическая сложность алгоритма сортировки простым выбором в худшем и среднем случаях составляет $O(n^2)$, где n - количество элементов в массиве.

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

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

Худший случай - массив отсортирован в обратном порядке. В этом случае сложность $O(n^2)$/

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 обменом и вставками;

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

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

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

- Произведено сравнение алгоритмов простой сортировки обменом и вставками 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).