|  |
| --- |
| 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: Елисеев И.А. |
|  |  |
|  | Группа: ИКБО-74-23 |

Москва – 2024

СОДЕРЖАНИЕ

[1 ЦЕЛЬ 4](#_gjdgxs)

[2 ЗАДАНИЕ №1 5](#_30j0zll)

[2.1 Формулировка задачи (Вариант 3) 5](#_1fob9te)

[2.2 Математическая модель решения алгоритма 5](#_3znysh7)

[2.2.1 Описание выполнения и блок-схема алгоритма сортировки простым выбором 6](#_2et92p0)

[2.2.2 Доказательство корректности циклов алгоритма сортировки простым выбором 6](#_tyjcwt)

[2.2.3 Определение ситуаций лучшего, среднего и худшего случая и функции роста времени работы алгоритма сортировки простого выбора 7](#)

[2.3 Реализация алгоритма на языке C++, проведение тестирования и построение графика 8](#)

[2.3.1 Реализация алгоритма сортировки простого выбора на языке C++ 8](#_4d34og8)

[2.3.2 Тестирование 10](#_2s8eyo1)

[2.3.3 Построение графика 11](#)

[2.4 Вывод по заданию №1 11](#_3rdcrjn)

[3 ЗАДАНИЕ №2 13](#)

[3.1 Формулировка задачи 13](#_26in1rg)

[3.2 Тестирование программы 13](#_lnxbz9)

[3.2.1 Массив упорядоченный по убыванию 13](#_35nkun2)

[3.2.2 Массив упорядоченный по возрастанию 17](#)

[3.3 Вывод по заданию №2 20](#_1ksv4uv)

[4 ЗАДАНИЕ №3 20](#_44sinio)

[4.1 Формулировка задания 21](#_2jxsxqh)

[4.2 Математическая модель решения алгоритма 21](#_z337ya)

[4.2.1 Описание выполнения и блок-схема алгоритма сортировки простым выбором 21](#_3j2qqm3)

[4.2.2 Доказательство корректности циклов алгоритма сортировки простым обменом 22](#_1y810tw)

[4.2.3 Определение ёмкостной сложности, ситуации лучшего, среднего и худшего случая и функции роста времени работы алгоритма сортировки простым обменом 22](#_4i7ojhp)

[4.3 Реализация алгоритма на языке C++, проведение тестирования и построение графика 24](#_2xcytpi)

[4.3.1 Реализация алгоритма сортировки простым обменом на языке C++ 24](#_1ci93xb)

[4.3.2 Тестирование при случайном заполнении массива 26](#_3whwml4)

[4.3.3 Построение графика 26](#_2bn6wsx)

[4.3.4 Массив упорядоченный по убыванию 27](#_qsh70q)

[4.3.5 Массив упорядоченный по возрастанию 31](#)

[4.4 Сравнение графиков двух алгоритмов сортировки из задания 1 и 3 35](#_2p2csry)

[4.5 Выводы по заданию №3 37](#_147n2zr)

[5 КОНТРОЛЬНЫЕ ВОПРОСЫ 39](#_3o7alnk)

[6 ВЫВОДЫ 42](#_23ckvvd)

[7 ЛИТЕРАТУРА 43](#_ihv636)

# 1 ЦЕЛЬ

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

# 2 ЗАДАНИЕ №1

## **2.1 Формулировка задачи (Вариант 3)**

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 Тп как сумму сравнений Сп и перемещений Мп. Полученные результаты вставить в сводную таблицу 2.

5. Построить график функции роста Тп 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 выбором**

Алгоритм сортировки выбором работает следующим образом: изначально в массиве имеется неупорядоченная часть, в которой поочередно выбирается элемент с наименьшим значением. Этот элемент затем обменивается с первым элементом неупорядоченной части, и становится частью упорядоченной части массива. Процесс повторяется для оставшихся элементов неупорядоченной части. Таким образом, на каждом шаге алгоритма увеличивается упорядоченная часть массива, а неупорядоченная часть сокращается. Этот процесс продолжается до тех пор, пока в неупорядоченной части не останется только один элемент.

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

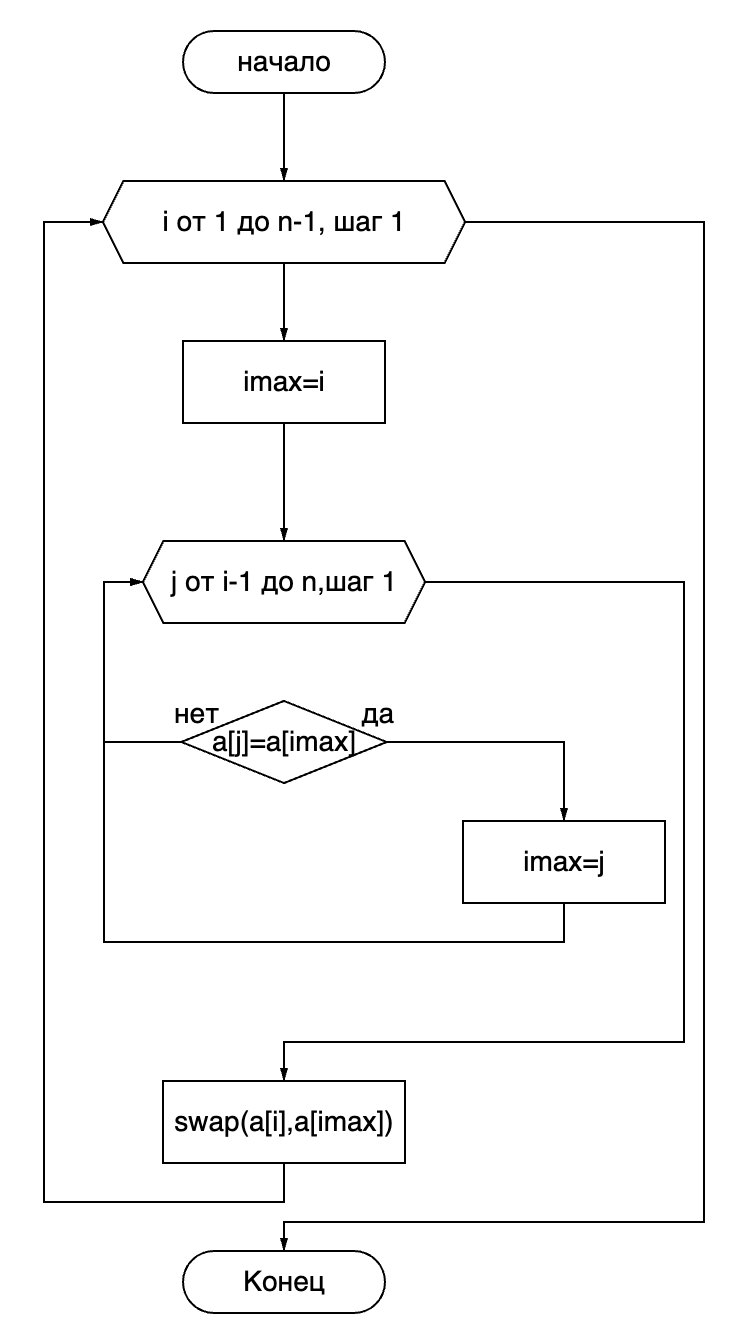


Рисунок 1 – Блок-схема алгоритма сортировки простым выбором

### **2.2.2 Доказательство корректности циклов алгоритма сортировки простым выбором**

Мы можем утверждать, что внешний цикл алгоритма сортировки выбором всегда завершается, так как его инвариант заключается в том, что переменная i всегда меньше n−1. Это означает, что количество итераций внешнего цикла ограничено сверху значением n−1, что гарантирует его завершение.

Аналогично, внутренний цикл завершается, так как его инвариант утверждает, что переменная j всегда больше i и меньше n. Это гарантирует, что цикл будет выполняться только в пределах корректного диапазона значений.

Мы доказали конечность циклов, исходя из того, что на каждой итерации внешнего цикла один элемент из неупорядоченной части массива переходит в упорядоченную часть. Поскольку каждая итерация уменьшает размер неупорядоченной части на 1 элемент, алгоритм завершится, когда неупорядоченная часть будет содержать только один элемент. Таким образом, конечность циклов обусловлена постепенным переходом элементов из неупорядоченной части в упорядоченную, что гарантирует корректное завершение алгоритма.

### **2.2.3 Определение ситуаций лучшего, среднего и худшего случая и функции роста времени работы алгоритма сортировки простого выбора**

Таблица 1-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 | (n2-n)/2 |
| 5 | if a[j]←a[imax] then | (n2-n)/2-(n-1) |
| 6 | imax←j | (n2-n)/2-(n-1) |
| 7 | Endlf  оd |  |
| 8 | swap(a[i],a[imax]) | 3\*(n-1) |
| 9 | od |  |
| 10 | } |  |

a. В случае, когда массив уже отсортирован, механизм сортировки выбором имеет временную сложность O(n2). Для массива, содержащего случайные числа, временная сложность остается на уровне O(n2). В худшем случае, когда массив упорядочен в обратном порядке, количество операций также оценивается как O(n2).

b. Функции роста времени:

- В лучшем случае: O(n2)

- В худшем случае: O(n2)

Для данного метода сортировки, время выполнения растет квадратично по размеру входного массива. Поэтому можно использовать квадратичную функцию для описания функции роста этого метода сортировки. Ёмкостная сложность алгоритма остается на уровне O(1).

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

### **2.3.1 Реализация алгоритма сортировки простого выбора на языке C++**

Для реализации алгоритма на языке C++ (листинг 1) потребуются следующие библиотеки: iostream, random, chrono.

| **#include <iostream>** **#include <random>** **#include <chrono>** using namespace std; void **swap**(int& a, int& b) {*// Функция для обмена значений двух элементов*  int temp = a;  a = b;  b = temp; } void **selectionsort**(int A[], int n, long& operations) {*// Функция сортировки выбором*  for (int i = 0; i < n - 1; i++) {*// Цикл для перебора массива от первого до предпоследнего элеиента*  int m = i;*//переменная для хранения индекса минимального элемента*  for (int j = i + 1; j < n; j++) {*//поиска минимального элемента в массиве*  if (A[j] < A[m]) {  m = j;*//обновление индекса минимального элемента*  operations++;*//присвоение*  }  operations+=2;  }  swap(A[i], A[m]);*// Обмен значениями*  operations+=4;*//сравнение при входе/выходе в цикл, присваивание*  }  operations++; } int **main**() {  int n,a;  cout << "Введите n: ";  cin >> n;  int A[n];  long operations;  cout << "Введите 1 (для случайной генерации чисел) или 2 (для ручного ввода) : ";  cin >> a;  if (a==1){  mt19937 **gen**(random\_device{}()); *//mt19937 gen-генератор случайных чисел. random\_device{}- класс, формирует случайную последовательность с помощью внешнего устройства.*  uniform\_int\_distribution<int> dist(1, 10); *//инструмент, позволяющий генерировать случайное целое число в заданном диапазоне*  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++) { *// i принимает значения от 0 до n включительно.*   cin >> A[i];*// Ввод элементов массива*  }  }  else {  cout<<"Error";  return 0;  }  auto start = chrono::high\_resolution\_clock::now();*// Замер времени начала сортировки*  selectionsort(A, n, operations);  auto end = chrono::high\_resolution\_clock::now();*// Замер времени окончания сортировки*  auto duration = chrono::duration\_cast<chrono::microseconds>(end - start).count();*//подсчёт затраченного превени сортировки*  cout << "Сортированный массив:\n";  for (int i = 0; i < n; ++i) {  cout << A[i] << " ";  }  cout << endl;  cout << "Было совершено операций сравнения и присваивания: " << operations << endl;  cout << "Сортировка заняла " << duration << " микросекунд" << endl;*// Вывод времени сортировки (в микросекундах)*   return 0; } |
| --- |

Листинг 1 – Программа алгоритма сортировки простым выбором

### **2.3.2 Тестирование**

Необходимо провести тестирование программы с массивами различного размера: n=10(рис.2), n=100, n=1000, n=10000, n=100000, n=1000000. Для этого будем использовать случайную генерацию чисел. Результаты тестирования для массивов размером от 100 до 1000000 будут представлены в таблице 2. Будем измерять время в наносекундах для получение максимально точных результатов, а затем для заполнения таблицы переведем данные в миллисекунды.

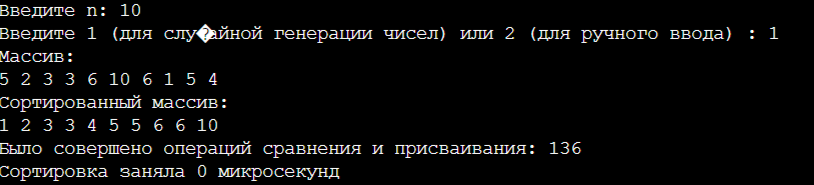


Рисунок 2 - Тестирование программы при n=10

Таблица 2. Сводная таблица результатов

| **n** | **T(n), мс** | **Тт=C+M** | **Тп=Cп+Mп** |
| --- | --- | --- | --- |
| 100 | 0,023 | - | 10490 |
| 1000 | 1,32 | - | 1004967 |
| 10000 | 193,46 | - | 100049023 |
| 100000 | 119357,53 | - | 10000493053 |
| 1000000 | 3069895,14 | - | 1000004902412 |

### 

### **2.3.3 Построение графика**

На основе данных, представленных в таблице 2, будет построен график зависимости времени выполнения алгоритма от размера массива n (рис.3).

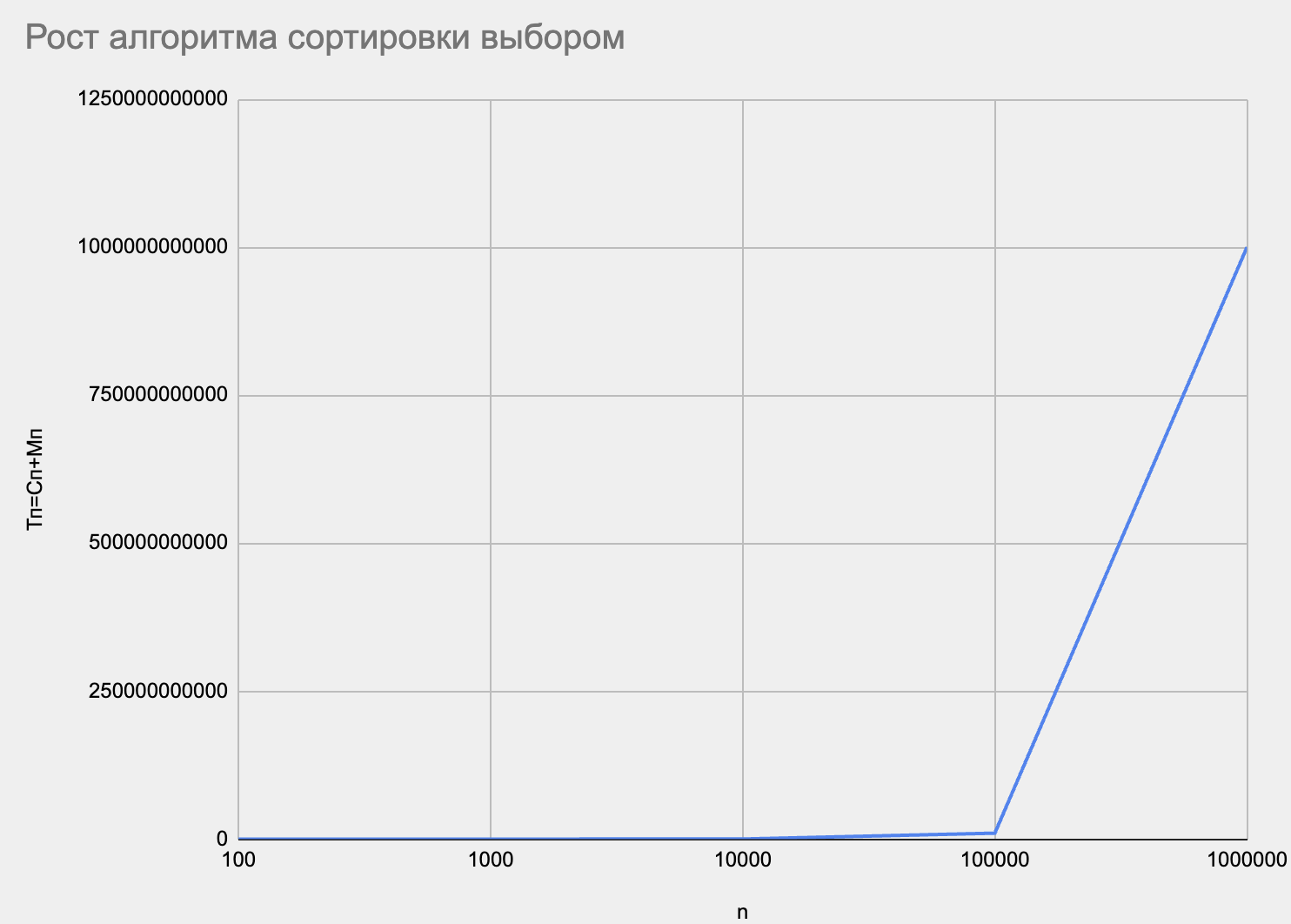


Рисунок 3 - График функции роста Тп этого алгоритма от размера массива n

## **2.4 Вывод по заданию №1**

Исходя из результатов тестирования и анализа вычислительной сложности, можно сделать вывод о квадратичной вычислительной сложности алгоритма сортировки простым выбором. Это означает, что время выполнения алгоритма будет увеличиваться квадратично с увеличением размера массива.

# 3 ЗАДАНИЕ №2

## **3.1 Формулировка задачи**

Оценить вычислительную сложность алгоритма простой сортировки выбором 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;

b. строго в возрастающем порядке значений, результаты представить в сводной таблице;

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

## **3.2 Тестирование программы**

### **3.2.1 Массив упорядоченный по убыванию**

Проведется тестирование программы на массивах различного размера: 100, 1000, 10000, 100000 и 1000000 элементов, отсортированных в порядке убывания. Реализация приведена в листинге 2.Продемонстрируем исполнение программы при n=10 (рис.4).

| **#include <iostream>** **#include <random>** **#include <chrono>** using namespace std; void **swap**(int& a, int& b) {*// Функция для обмена значений двух элементов*  int temp = a;  a = b;  b = temp; } void **ReversedSort**(int A[],int n){*//сортировка по убыванию*  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]);  }  }  } } void **selectionsort**(int A[], int n, long& operations) {*// Функция сортировки выбором*  for (int i = 0; i < n - 1; i++) {*// Цикл для перебора массива от первого до предпоследнего элеиента*  int m = i;*//переменная для хранения индекса минимального элемента*  for (int j = i + 1; j < n; j++) {*//поиска минимального элемента в массиве*  if (A[j] < A[m]) {  m = j;*//обновление индекса минимального элемента*  operations++;*//присвоение*  }  operations+=2;  }  swap(A[i], A[m]);*// Обмен значениями*  operations+=4;*//сравнение при входе/выходе в цикл, присваивание*  }  operations++; } int **main**() {  int n,a;  cout << "Введите n: ";  cin >> n;  int A[n];  long operations;  cout << "Введите 1 (для случайной генерации чисел) или 2 (для ручного ввода) : ";  cin >> a;  if (a==1){  mt19937 **gen**(random\_device{}()); *//mt19937 gen-генератор случайных чисел. random\_device{}- класс, формирует случайную последовательность с помощью внешнего устройства.*  uniform\_int\_distribution<int> dist(1, 10); *//инструмент, позволяющий генерировать случайное целое число в заданном диапазоне*  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++) { *// i принимает значения от 0 до n включительно.*   cin >> A[i];*// Ввод элементов массива*  }  }  else {  cout<<"Error";  return 0;  }  ReversedSort(A,n);*//сортировка массива по убыванию*  auto start = chrono::high\_resolution\_clock::now();*// Замер времени начала сортировки*  selectionsort(A, n, operations);  auto end = chrono::high\_resolution\_clock::now();*// Замер времени окончания сортировки*  auto duration = chrono::duration\_cast<chrono::microseconds>(end - start).count();*//подсчёт затраченного превени сортировки*  cout << "Сортированный массив:\n";  for (int i = 0; i < n; ++i) {  cout << A[i] << " ";  }  cout << endl;  cout << "Было совершено операций сравнения и присваивания: " << operations << endl;  cout << "Сортировка заняла " << duration << " микросекунд" << endl;*// Вывод времени сортировки (в микросекундах)*   return 0; } |
| --- |

Листинг 2 – Алгоритм сортировки простым выбором

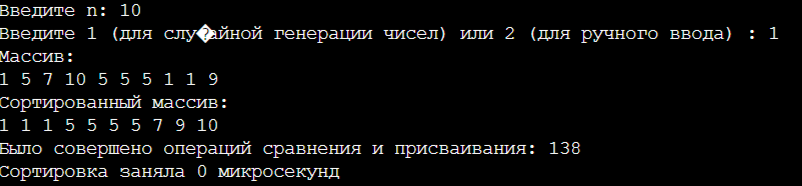


Рисунок 4 – Результаты тестирования программы при n=10 и с отсортированными значениями по убыванию

Поскольку значения отсортированы в строго убывающем порядке, можно сделать вывод о том, что это представляет собой наихудший случай сортировки, и, следовательно, сложность алгоритма равна O(n2). Таким образом, в наихудшем случае алгоритм имеет квадратичную вычислительную сложность. Результаты тестирования будут занесены в таблицу 3.

Таблица 3. Сводная таблица результатов

| **n** | **T(n), мс** | **Тт=C+M** | **Тп=Cп+Mп** |
| --- | --- | --- | --- |
| 100 | 0,037 | - | 10534 |
| 1000 | 1,34 | - | 1005553 |
| 10000 | 159,48 | - | 100055547 |
| 100000 | 18340,574 | - | 1000055578 |
| 1000000 | 7335234,943 | - | 1000005555734 |

На основе данных, представленных в таблице 3, будет построен график зависимости времени выполнения алгоритма от размера массива n (рис.5).

### 

Рисунок 5 - График функции роста Тп алгоритма сортировки простым выбором с отсортированными значениями по убыванию от размера массива n

### **3.2.2 Массив упорядоченный по возрастанию**

Проведется тестирование программы на массивах различного размера: 100, 1000, 10000, 100000 и 1000000 элементов, отсортированных в порядке убывания (листинг 3). Продемонстрируем исполнение программы при n=10 (рис.6).

| **#include <iostream>** **#include <random>** **#include <chrono>** using namespace std; void **swap**(int& a, int& b) {*// Функция для обмена значений двух элементов*  int temp = a;  a = b;  b = temp; } void **Sort**(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]);  }  }  } } void **selectionsort**(int A[], int n, long& operations) {*// Функция сортировки выбором*  for (int i = 0; i < n - 1; i++) {*// Цикл для перебора массива от первого до предпоследнего элеиента*  int m = i;*//переменная для хранения индекса минимального элемента*  for (int j = i + 1; j < n; j++) {*//поиска минимального элемента в массиве*  if (A[j] < A[m]) {  m = j;*//обновление индекса минимального элемента*  operations++;*//присвоение*  }  operations+=2;  }  swap(A[i], A[m]);*// Обмен значениями*  operations+=4;*//сравнение при входе/выходе в цикл, присваивание*  }  operations++; } int **main**() {  int n,a;  cout << "Введите n: ";  cin >> n;  int A[n];  long operations;  cout << "Введите 1 (для случайной генерации чисел) или 2 (для ручного ввода) : ";  cin >> a;  if (a==1){  mt19937 **gen**(random\_device{}()); *//mt19937 gen-генератор случайных чисел. random\_device{}- класс, формирует случайную последовательность с помощью внешнего устройства.*  uniform\_int\_distribution<int> dist(1, 10); *//инструмент, позволяющий генерировать случайное целое число в заданном диапазоне*  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++) { *// i принимает значения от 0 до n включительно.*   cin >> A[i];*// Ввод элементов массива*  }  }  else {  cout<<"Error";  return 0;  }  Sort(A,n);*//сортировка массива по убыванию*  for (int i = 0; i < n; ++i) {  cout << A[i] << " ";  }  cout << endl;  auto start = chrono::high\_resolution\_clock::now();*// Замер времени начала сортировки*  selectionsort(A, n, operations);  auto end = chrono::high\_resolution\_clock::now();*// Замер времени окончания сортировки*  auto duration = chrono::duration\_cast<chrono::microseconds>(end - start).count();*//подсчёт затраченного превени сортировки*  cout << "Сортированный массив:\n";  for (int i = 0; i < n; ++i) {  cout << A[i] << " ";  }  cout << endl;  cout << "Было совершено операций сравнения и присваивания: " << operations << endl;  cout << "Сортировка заняла " << duration << " микросекунд" << endl;*// Вывод времени сортировки (в микросекундах)*   return 0; } |
| --- |

Листинг 3 – Алгоритм сортировки простым выбором

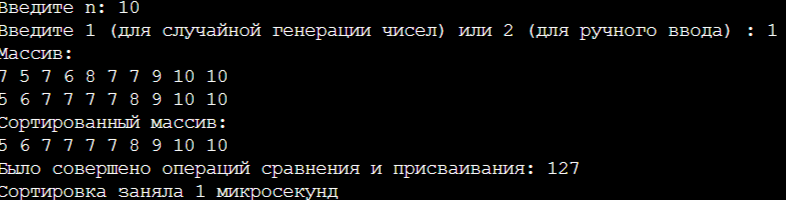


Рисунок 6 – Результаты тестирования программы при n=10 и с отсортированными значениями по возрастанию

Поскольку значения отсортированы в строго возрастающем порядке, можно сделать вывод о том, что это представляет собой лучший случай сортировки, и, следовательно, сложность алгоритма равна O(n2). Таким образом, в лучшем случае алгоритм имеет квадратичную вычислительную сложность. Результаты тестирования будут занесены в таблицу 4.

Таблица 4. Сводная таблица результатов

| **n** | **T(n), мс** | **Тт=C+M** | **Тп=Cп+Mп** |
| --- | --- | --- | --- |
| 100 | 0,053 | - | 10297 |
| 1000 | 2,12 | - | 1002997 |
| 10000 | 174,64 | - | 100029997 |
| 100000 | 20358,35 | - | 10000299997 |
| 1000000 | 2371358,324 | - | 1000002999997 |

На основе полученных данных, продемонстрированных в таблице 4, построим график функции роста Тп этого алгоритма от размера массива n с отсортированными значениями по возрастанию (рис.7).

### 

Рисунок 7 - График функции роста Тп алгоритма сортировки простым выбором с отсортированными значениями по возрастанию от размера массива n

## **3.3 Вывод по заданию №2**

Сортировка выбором не зависит от исходного порядка элементов в массиве. Независимо от того, отсортирован ли массив или нет, алгоритм всегда будет просматривать каждый элемент, чтобы найти минимальный или максимальный. Это означает, что процесс сортировки будет выполняться таким же образом независимо от исходного порядка элементов. Таким образом, алгоритм сортировки выбором всегда требует полного просмотра массива для корректной сортировки, что делает его независимым от начальной упорядоченности массива.

# 4 ЗАДАНИЕ №3

## **4.1 Формулировка задания**

Сравнить эффективность алгоритмов простых сортировок

1. Выполнить разработку и программную реализацию алгоритма простого обмена.

2. Аналогично заданиям 1 и 2 сформировать таблицы с результатами эмпирического исследования второго алгоритма в среднем, лучшем и худшем случаях (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 Тп(n) двух алгоритмов сортировки в худшем случае.

5. Аналогично на другом общем графике отобразить функции Тп(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**

Пузырьковая сортировка, также известная как сортировка простым обменом, - один из наиболее широко распространенных алгоритмов сортировки, который обычно изучается еще в школьных курсах информатики. В этом алгоритме элементы массива сравниваются попарно, и, если необходимо, меняются местами, чтобы упорядочить их по возрастанию. После каждого прохода по массиву наибольший элемент занимает свое окончательное место в конце массива. Алгоритм завершается, когда либо вся область массива отсортирована, либо при очередном проходе не было совершено ни одной перестановки (по критерию Айверсона).

Реализация данного описания выполнения алгоритма представлена в виде блок-схемы (рис.8).

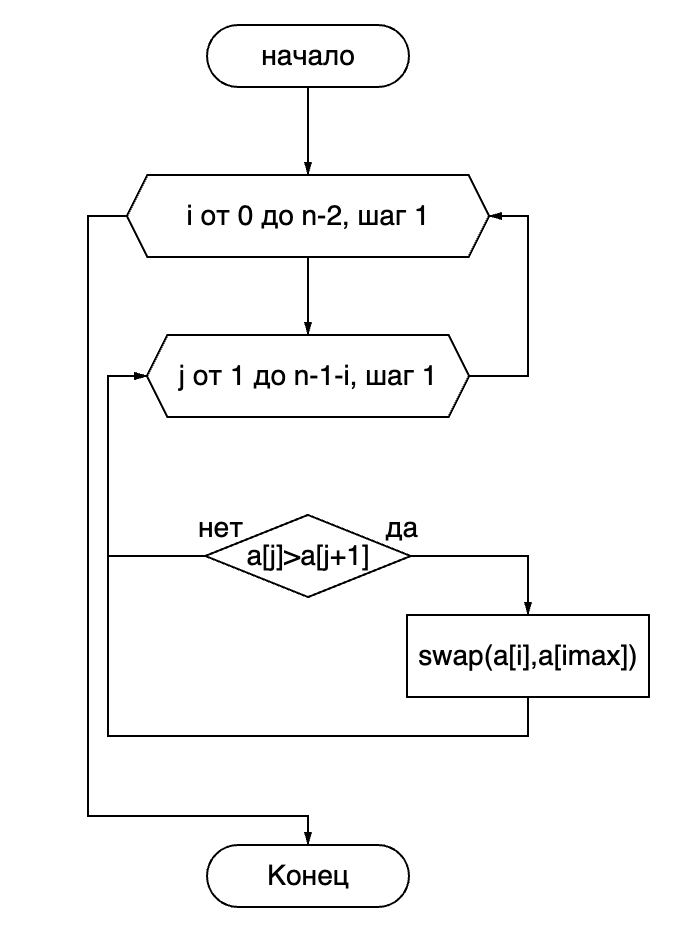


Рисунок 8 – Блок-схема алгоритма сортировки простым выбором

### **4.2.2 Доказательство корректности циклов алгоритма сортировки простым обменом**

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

Таблица 5. Псевдокод и анализ алгоритма сортировки выбором

| № | 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 | tj = 𝑛 + 𝑛 − 1 + ⋯ + 2=0.5𝑛2 − 1.5𝑛 + 1 |
| 4 | if (a[j] > a[j + 1]) then | tj - 1= 0.5𝑛2 − 2.5𝑛 + 2 |
| 5 | swap(a[j], a[j + 1]) | 3\*(tj - 1) = 1.5𝑛2 − 7.5𝑛 + 6 |
| 6 | endif |  |
| 7 | оd |  |
| 8 | od |  |
| 9 | } |  |

В лучшем случае, когда массив уже отсортирован, количество операций сравнения и перемещения минимизируется до O(n), что означает линейную сложность. В среднем случае, когда массив заполнен случайными числами, алгоритм имеет квадратичную сложность O(n2). В худшем случае, когда массив отсортирован в обратном порядке, количество операций также будет O(n2).

Для данного метода сортировки, время выполнения увеличивается квадратично с ростом размера входного массива. Поэтому можно использовать квадратичную функцию для описания функции роста данного сортировочного метода. Время выполнения в лучшем случае увеличивается линейно с ростом размера входного массива. Ёмкостная сложность алгоритма остается постоянной и равной O(1).

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

### **4.3.1 Реализация алгоритма сортировки простым обменом на языке C++**

Для реализации алгоритма на языке C++ (листинг 4) потребуются следующие библиотеки: iostream, random, chrono.

| **#include <iostream>** **#include <random>** **#include <chrono>** using namespace std; void **swap**(int& a, int& b) {*// Функция для обмена значений двух элементов*  int temp = a;  a = b;  b = temp; } void **ExchangeSort**(int A[], int n, long long& operation){*//A- массив, n-размер массива*   for (int i=0; i<n-1;++i){  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 массива*  operation++;  }  operation+=2;  }  operation+=2;  }  operation++; } int **main**() {  int n,a;  cout << "Enter n: ";  cin >> n;  int A[n];  long long operations;  cout << "Enter 1 (random) or 2 (manual input): ";  cin >> a;  if (a==1){  mt19937 **gen**(random\_device{}()); *//mt19937 gen-генератор случайных чисел. random\_device{}- класс, формирует случайную последовательность с помощью внешнего устройства.*  uniform\_int\_distribution<int> dist(1, 100); *//инструмент, позволяющий генерировать случайное целое число в заданном диапазоне*  cout << "Array:" << 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 << "Enter the array elements:" << endl;  for (int i = 0; i < n; i++) { *// i принимает значения от 0 до n включительно.*   cin >> A[i];*// Ввод элементов массива*  }  }  else {  cout<<"Error";  return 0;  }  auto start = chrono::high\_resolution\_clock::now();*// Замер времени начала сортировки*  ExchangeSort(A, n, operations);  auto end = chrono::high\_resolution\_clock::now();*// Замер времени окончания сортировки*  auto duration = chrono::duration\_cast<chrono::microseconds>(end - start).count();*//подсчёт затраченного превени сортировки*  cout << "Sorted array:\n";  for (int i = 0; i < n; ++i) {  cout << A[i] << " ";  }  cout << endl;  cout << "There were no comparatoin or assigment operations: " << operations << endl;  cout << "Sorting took " << duration << " microseconds" << endl;*// Вывод времени сортировки (в микросекундах)*   return 0; } |
| --- |

Листинг 4 – Программа алгоритма сортировки простым обменом

### **4.3.2 Тестирование при случайном заполнении массива**

Необходимо провести тестирование программы с массивами различного размера: n=10(рис.9), n=100, n=1000, n=10000, n=100000, n=1000000. Для этого будем использовать случайную генерацию чисел. Результаты тестирования для массивов размером от 100 до 1000000 будут представлены в таблице 6. Будем измерять время в наносекундах для получение максимально точных результатов, а затем для заполнения таблицы переведем данные в миллисекунды.

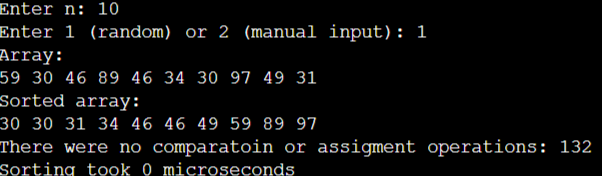


Рисунок 9 - Тестирование программы при n=10

Таблица 6. Сводная таблица результатов

| **n** | **T(n), мс** | **Тт=C+M** | **Тп=Cп+Mп** |
| --- | --- | --- | --- |
| 100 | 0,046 |  | 12468 |
| 1000 | 2,93 |  | 1229357 |
| 10000 | 290,35 |  | 12227235 |
| 100000 | 36453,64 |  | 12253586358 |
| 1000000 | 5215557,35 |  | 1220453346358 |

### **4.3.3 Построение графика**

На основе данных, представленных в таблице 6, будет построен график зависимости времени выполнения алгоритма от размера массива n (рис.10).

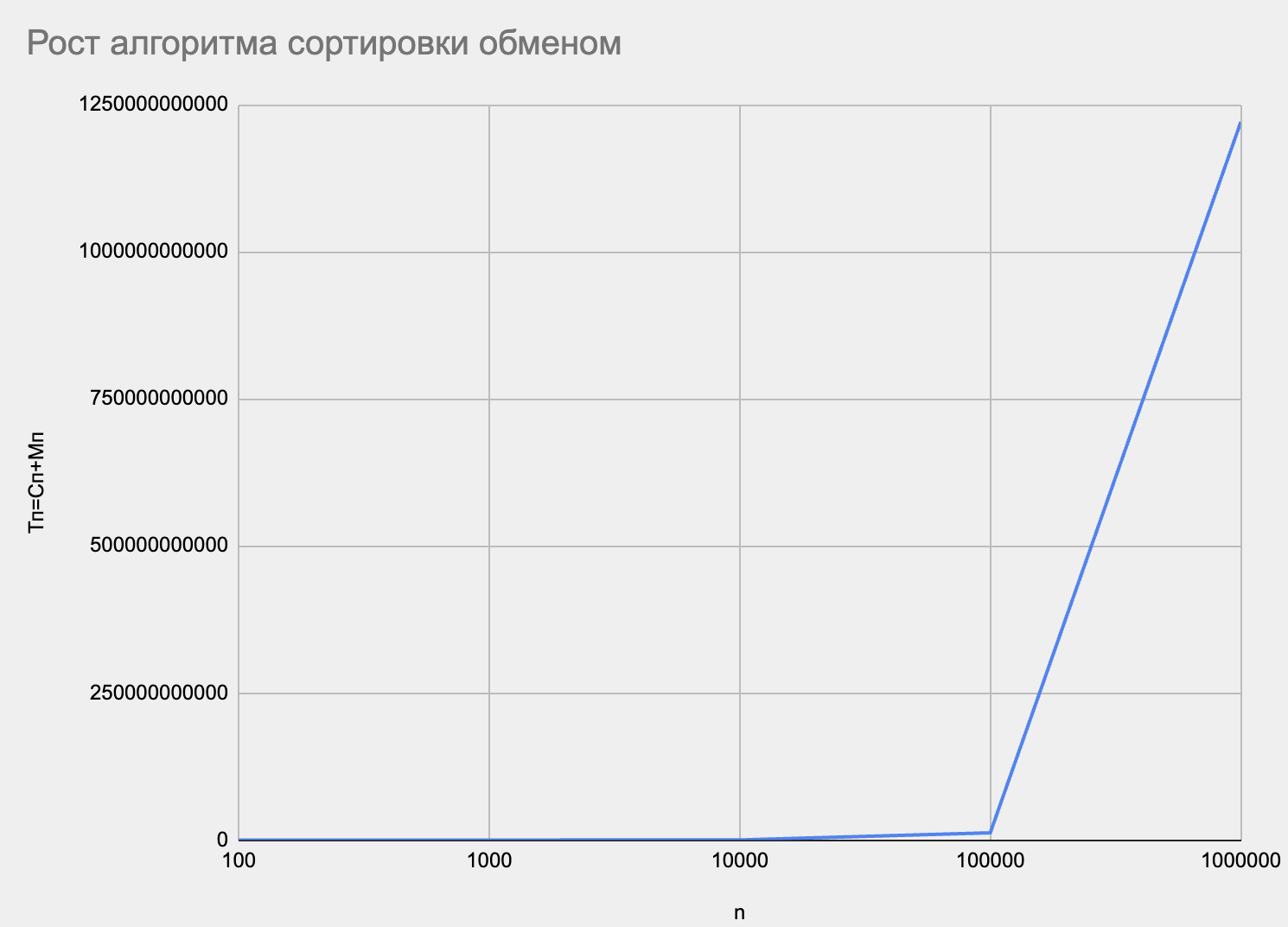


Рисунок 10 - График функции роста Тп этого алгоритма от размера массива n

### **4.3.4 Массив упорядоченный по убыванию**

Проведется тестирование программы на массивах различного размера: 100, 1000, 10000, 100000 и 1000000 элементов, отсортированных в порядке убывания(листинг 5). Продемонстрируем исполнение программы при n=10 (рис.11).

| **#include <iostream>** **#include <random>** **#include <chrono>** using namespace std; void **swap**(int& a, int& b) {*// Функция для обмена значений двух элементов*  int temp = a;  a = b;  b = temp; } 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]);  }  }  } } void **ExchangeSort**(int A[], int n, long long& operation){*//A- массив, n-размер массива*   for (int i=0; i<n-1;++i){  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 массива*  operation++;  }  operation+=2;  }  operation+=2;  }  operation++; } int **main**() {  int n,a;  cout << "Enter n: ";  cin >> n;  int A[n];  long long operations;  cout << "Enter 1 (random) or 2 (manual input): ";  cin >> a;  if (a==1){  mt19937 **gen**(random\_device{}()); *//mt19937 gen-генератор случайных чисел. random\_device{}- класс, формирует случайную последовательность с помощью внешнего устройства.*  uniform\_int\_distribution<int> dist(1, 100); *//инструмент, позволяющий генерировать случайное целое число в заданном диапазоне*  cout << "Array:" << 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 << "Enter the array elements:" << endl;  for (int i = 0; i < n; i++) { *// i принимает значения от 0 до n включительно.*   cin >> A[i];*// Ввод элементов массива*  }  }  else {  cout<<"Error";  return 0;  }  ReversedSort(A,n);*//сортировка массива по убыванию*  auto start = chrono::high\_resolution\_clock::now();*// Замер времени начала сортировки*  ExchangeSort(A, n, operations);  auto end = chrono::high\_resolution\_clock::now();*// Замер времени окончания сортировки*  auto duration = chrono::duration\_cast<chrono::microseconds>(end - start).count();*//подсчёт затраченного превени сортировки*  cout << "Sorted array:\n";  for (int i = 0; i < n; ++i) {  cout << A[i] << " ";  }  cout << endl;  cout << "There were no comparatoin or assigment operations: " << operations << endl;  cout << "Sorting took " << duration << " microseconds" << endl;*// Вывод времени сортировки (в микросекундах)*  return 0; } |
| --- |

Листинг 5 – Алгоритм сортировки простым обменом

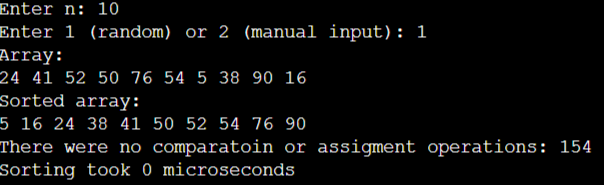


Рисунок 11 – Результаты тестирования программы при n=10 и с отсортированными значениями по убыванию

Поскольку значения отсортированы в строго убывающем порядке, можно сделать вывод о том, что это представляет собой наихудший случай сортировки, и, следовательно, сложность алгоритма равна O(n2). Таким образом, в наихудшем случае алгоритм имеет квадратичную вычислительную сложность. Результаты тестирования будут занесены в таблицу 7.

Таблица 7. Сводная таблица результатов

| **n** | **T(n), мс** | **Тт=C+M** | **Тп=Cп+Mп** |
| --- | --- | --- | --- |
| 100 | 0,053 |  | 14534 |
| 1000 | 3,12 |  | 1450464 |
| 10000 | 335,16 |  | 145007213 |
| 100000 | 33582,35 |  | 14500018935 |
| 1000000 | 42535837,31 |  | 1450000074635 |

На основе данных, представленных в таблице 7, будет построен график зависимости времени выполнения алгоритма от размера массива n с отсортированными значениями по убыванию (рис.12).

### 

Рисунок 12 - График функции роста Тп алгоритма сортировки простым выбором с отсортированными значениями по убыванию от размера массива n

### **4.3.5 Массив упорядоченный по возрастанию**

Проведется тестирование программы на массивах различного размера: 100, 1000, 10000, 100000 и 1000000 элементов, отсортированных в порядке убывания (листинг 6).Продемонстрируем исполнение программы при n=10 (рис.13).

| **#include <iostream>** **#include <random>** **#include <chrono>** using namespace std; void **swap**(int& a, int& b) {*// Функция для обмена значений двух элементов*  int temp = a;  a = b;  b = temp; } void **Sort**(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]);  }  }  } } void **ExchangeSort**(int A[], int n, long long& operation){*//A- массив, n-размер массива*   for (int i=0; i<n-1;++i){  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 массива*  operation++;  }  operation+=2;  }  operation+=2;  }  operation++; } int **main**() {  int n,a;  cout << "Enter n: ";  cin >> n;  int A[n];  long long operations;  cout << "Enter 1 (random) or 2 (manual input): ";  cin >> a;  if (a==1){  mt19937 **gen**(random\_device{}()); *//mt19937 gen-генератор случайных чисел. random\_device{}- класс, формирует случайную последовательность с помощью внешнего устройства.*  uniform\_int\_distribution<int> dist(1, 100); *//инструмент, позволяющий генерировать случайное целое число в заданном диапазоне*  cout << "Array:" << 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 << "Enter the array elements:" << endl;  for (int i = 0; i < n; i++) { *// i принимает значения от 0 до n включительно.*   cin >> A[i];*// Ввод элементов массива*  }  }  else {  cout<<"Error";  return 0;  }  Sort(A,n);*//сортировка массива по возрастанию*  auto start = chrono::high\_resolution\_clock::now();*// Замер времени начала сортировки*  ExchangeSort(A, n, operations);  auto end = chrono::high\_resolution\_clock::now();*// Замер времени окончания сортировки*  auto duration = chrono::duration\_cast<chrono::microseconds>(end - start).count();*//подсчёт затраченного превени сортировки*  cout << "Sorted array:\n";  for (int i = 0; i < n; ++i) {  cout << A[i] << " ";  }  cout << endl;  cout << "There were no comparatoin or assigment operations: " << operations << endl;  cout << "Sorting took " << duration << " microseconds" << endl;*// Вывод времени сортировки (в микросекундах)*  return 0; } |
| --- |

Листинг 6 - Алгоритм сортировки простым обменом

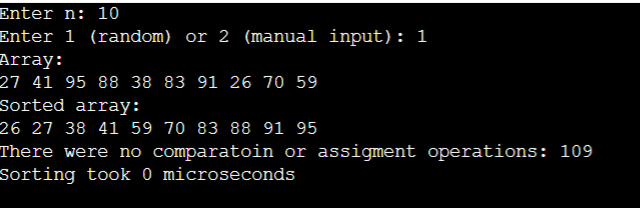


Рисунок 13 – Результаты тестирования программы при n=10 и с отсортированными значениями по возрастанию

Поскольку значения отсортированы в строго возрастающем порядке, можно сделать вывод о том, что это представляет собой лучший случай сортировки, и, следовательно, сложность алгоритма равна O(n). Таким образом, в лучшем случае алгоритм имеет линейную вычислительную сложность. Результаты тестирования будут занесены в таблицу 8.

Таблица 8. Сводная таблица результатов

| **n** | **T(n), мс** | **Тт=C+M** | **Тп=Cп+Mп** |
| --- | --- | --- | --- |
| 100 | 0,06 | - | 10099 |
| 1000 | 1,23 | - | 1000999 |
| 10000 | 223,35 | - | 100009999 |
| 100000 | 19747,53 | - | 10000099999 |
| 1000000 | 5213578,935 | - | 1000000999999 |

На основе данных, представленных в таблице 8, будет построен график зависимости времени выполнения алгоритма от размера массива n с отсортированными значениями по возрастанию (рис.14).

### 

Рисунок 14 - График функции роста Тп 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 Тп(n) двух алгоритмов сортировки в худшем случае**

На основе данных, представленных в таблице 3 и 7, будет построен график зависимости времени выполнения алгоритма от размера массива n для сравнения роста алгоритмов сортировок(рис.15).

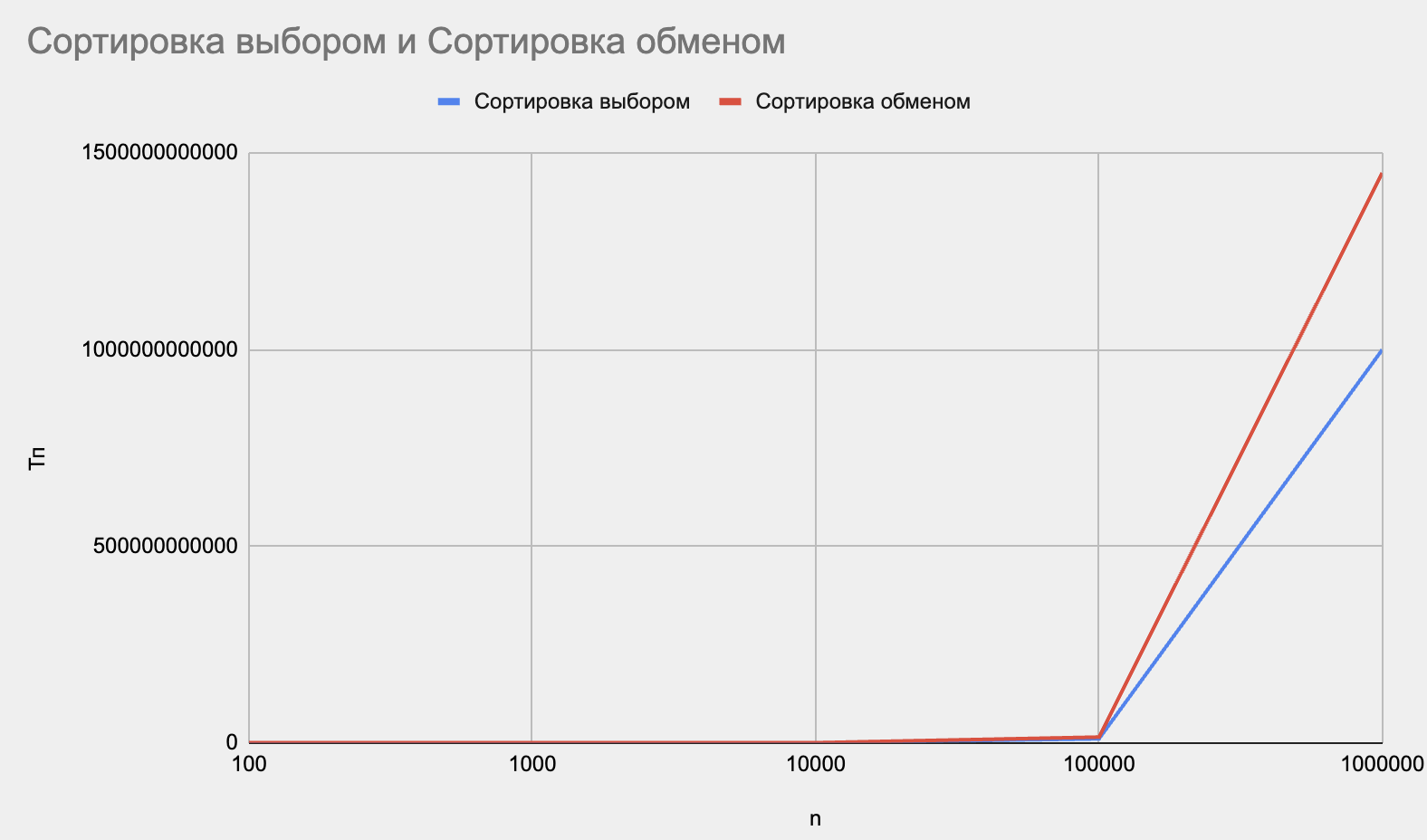


Рисунок 15 – График двух алгоритмов в худшем случае

Можно сделать вывод, что в худшем случае алгоритм сортировки простом обменом менее эффективный, чем алгоритм сортировки простого выбора.

**4.4.2 Отображение функции Тп(n) двух алгоритмов сортировки в лучшем случае**

На основе данных, представленных в таблице 4 и 8, будет построен график зависимости времени выполнения алгоритма от размера массива n для сравнения роста алгоритмов сортировок(рис.16).

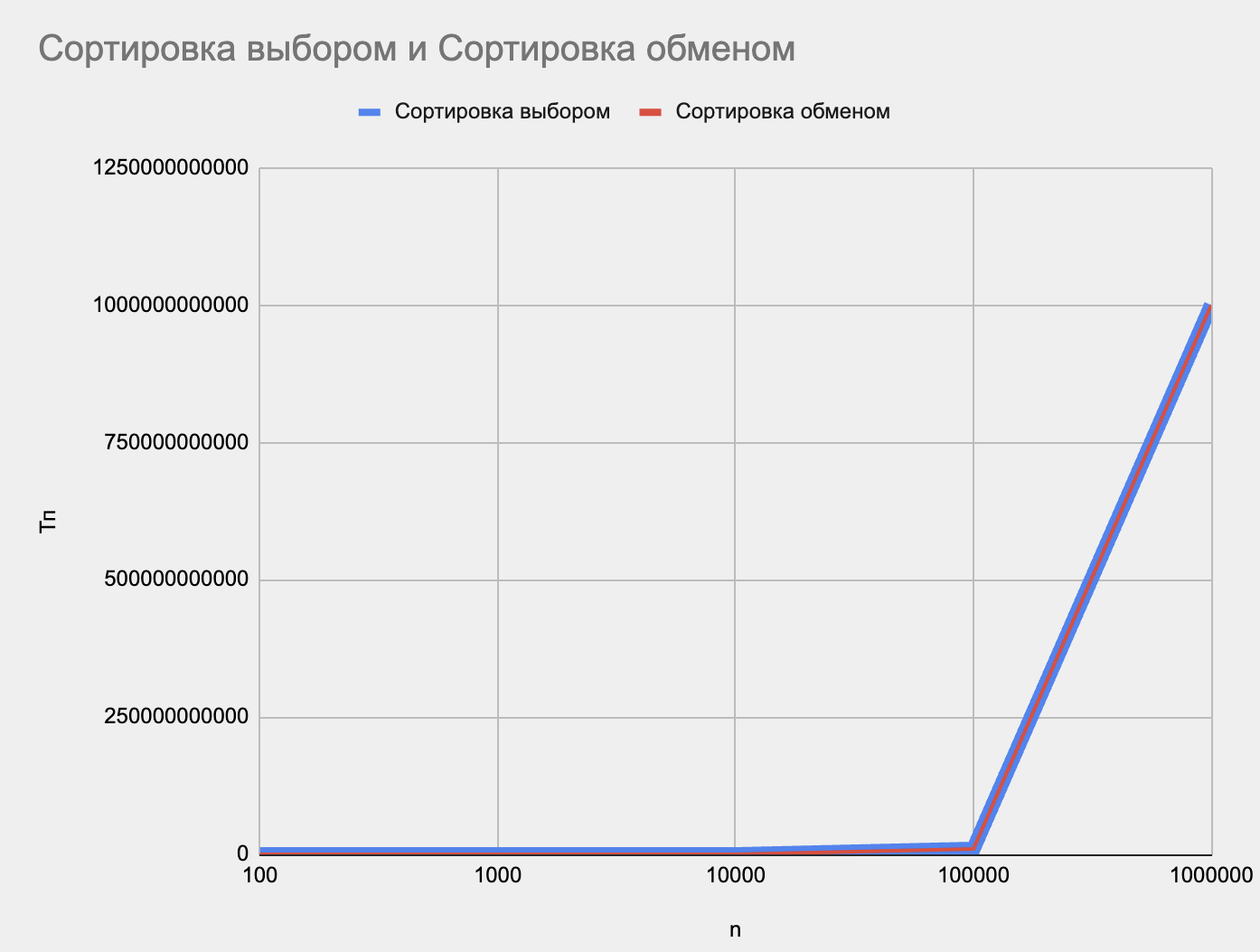


Рисунок 16 – График двух алгоритмов в лучшем случае

Можно сделать вывод, что в лучшем случае алгоритм сортировки простом обменом и сортировка простым выбором почти совпадает. Но всё же более эффективной будет сортировка простым обменом, чем сортировка простым выбором.

## **4.5 Выводы по заданию №3**

Сравнение сортировки пузырьком и сортировки выбором позволяет выявить различия в их эффективности и производительности.

Сортировка пузырьком, известная также как сортировка простым обменом, осуществляется путем сравнения и обмена соседних элементов массива. Она имеет временную сложность O(n2) и становится неэффективной для больших наборов данных из-за большого количества обменов.

В отличие от этого, сортировка выбором работает путем нахождения минимального элемента в массиве и перемещения его в начало. В худшем случае у нее также временная сложность O(n2), но, в отличие от сортировки пузырьком, она требует меньше обменов.

Таким образом, сортировка выбором может быть более эффективной в некоторых случаях использования, особенно при работе с небольшими наборами данных. Однако обе сортировки показывают низкую производительность при сортировке больших массивов данных и не являются оптимальным выбором для работы с большими объемами данных.

# 5 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие сортировки называют простыми?

Простыми сортировками обычно называют сортировки с простыми и понятными шагами исполнения, например, сортировка пузырьком, сортировка вставками, сортировка выбором и др.

2. Что означает понятие «внутренняя сортировка»?

Понятие "внутренняя сортировка" означает, что все операции сортировки происходят непосредственно в памяти компьютера, где хранятся сортируемые данные, без необходимости использования внешних ресурсов, таких как дополнительные диски.

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

Основные операции, которые учитываются при оценке сложности алгоритма сортировки, включают в себя сравнения элементов массива и перемещения элементов для их правильной упорядоченности.

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

При оценке эффективности алгоритма сортировки используются следующие характеристики сложности:

* Временная сложность (сколько операций требуется для выполнения алгоритма в зависимости от размера входных данных);
* Пространственная сложность (сколько памяти потребуется для выполнения алгоритма);
* Стабильность (сохраняется ли порядок элементов при равных значениях);
* Стабильность наихудшего случая (как поведет себя алгоритм в самом неэффективном случае).

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

Простой обмен (сортировка пузырьком):

• Вычислительная сложность: O(n2)

• Емкостная сложность: O(1)

Простая вставка:

• Вычислительная сложность: О(n2) в худшем случае, O(n) в лучшем случае

• Емкостная сложность: O(1)

Простой выбор:

• Вычислительная сложность: O(n2)

• Емкостная сложность: O(1)

6. Какую роль в сортировке обменом играет условие Айверсона?

Условие Айверсона определяет порядок сортировки элементов в алгоритме сортировки простым обменом (сортировка пузырьком). Оно позволяет уменьшить количество сравнений элементов, так как при каждом проходе алгоритма самый большой элемент "всплывает" на правильное место в конце массива.

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

По шагам выполнения сортировки исходного массива 5 6 1 2 3 можно сказать, что использовалась сортировка пузырьком (простой обмен). Последний шаг показывает отсортированный массив, что является результатом сортировки обменом.

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

Вычислительная теоретическая сложность алгоритма сортировки простого обмена в худшем случае составляет O(n2), что означает квадратичную зависимость от размера входных данных.

# 6 ВЫВОДЫ

В ходе практической работы были выполнены следующие задачи:

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

- Проведён анализ алгоритмов простой сортировки выбором и обменом;

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

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

- Построены графики функции роста Тп алгоритмов простой сортировки выбором и обменом от размера массива n.

- Произведено сравнение алгоритмов простой сортировки выбором и обменом 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.

# 7 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).