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

| **Отчет по выполнению практического задания № 3** | |
| --- | --- |
| **Тема:** | |
| **«Определение эффективного алгоритма сортировки на основе эмпирического и асимптотического методов анализа»** | |
| Дисциплина: «Структуры и алгоритмы обработки данных» | |
|  | Выполнил студент: Гендриксон А. А. |
|  | Группа: ИКБО-74-23 |

Москва – 2024

СОДЕРЖАНИЕ

[1 ЦЕЛЬ 4](#)

[2 ЗАДАНИЕ №1 5](#_30j0zll)

[2.1 Формулировка задачи (Вариант 6, в списке 6) 5](#_1fob9te)

[2.2 Математическая модель решения алгоритма 6](#_2et92p0)

[2.2.1 Описание выполнения и блок-схема алгоритма Шелла со смещением Седжвика 6](#_tyjcwt)

[2.2.2 Определение ситуаций лучшего, среднего и худшего случая и функции роста времени работы алгоритма шейкерной сортировки 8](#_1t3h5sf)

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

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

[2.3.2 Тестирование и построение графика 10](#)

[2.4 Математическая модель решения алгоритма 11](#_17dp8vu)

[2.4.1 Описание выполнения и блок-схема алгоритма простого слияния 11](#_ok2lxnxllljb)

[2.4.2 Определение ситуаций лучшего, среднего и худшего случая и функции роста времени работы алгоритма простого слияния 13](#)

[2.5 Реализация алгоритма на языке C++ и проведение тестирования 13](#)

[2.5.1 Реализация алгоритма простого слияния на языке C++ 13](#)

[2.5.2 Тестирование 15](#)

[2.6 Сортировка простым обменом 16](#_bwovjebkury)

[2.7 Сравнение трёх алгоритмов на графике 16](#)

[2.8 Тестирование программ для алгоритмов Шелла со смещением Седжвика и простого слияния. 18](#_3rdcrjn)

[2.8.1 Тестирование при упорядоченном по убыванию элементов массива и построение графика для алгоритма Шелла со смещением Седжвика 18](#_26in1rg)

[2.8.2 Тестирование при упорядоченном по возрастанию элементов массива и построение графика для алгоритма Шелла со смещением Седжвика 20](#)

[2.8.3 Тестирование при упорядоченном по убыванию элементов массива и построение графика для алгоритма простого слияния 23](#_lnxbz9)

[2.8.4 Тестирование при упорядоченном по возрастанию элементов массива и построение графика для алгоритма простого слияния 25](#)

[2.9 Вывод по заданию №1 28](#_1ksv4uv)

[3 ЗАДАНИЕ №2 29](#_2jxsxqh)

[3.1 Формулировка задачи 29](#_z337ya)

[3.2 Формулы функции роста алгоритма сортировки простым обменом в худшем и лучшем случае 29](#_3j2qqm3)

[3.3 Асимптотическая оценка вычислительной сложности простого алгоритма сортировки обменом 30](#)

[3.4 Графическое представление функции роста и полученных асимптотических оценок сверху и снизу 30](#_1y810tw)

[3.5 Справочная информация о вычислительной сложности алгоритма Шелла со смещением Седжвика и простого слияния 31](#_4i7ojhp)

[3.5.1 Справочная информация о вычислительной сложности алгоритма Шелла со смещением Седжвика 31](#_2xcytpi)

[3.5.2 Справочная информация о вычислительной сложности алгоритма простого слияния 32](#)

[3.6 Таблица асимптотической сложности трёх алгоритмов 32](#_1ci93xb)

[5 ВЫВОДЫ 34](#_3whwml4)

[6 ЛИТЕРАТУРА 35](#_2bn6wsx)

# 

# 1 ЦЕЛЬ

Получить навыки по анализу вычислительной сложности алгоритмов сортировки и определению наиболее эффективного алгоритма.

# 2 ЗАДАНИЕ №1

## **2.1 Формулировка задачи (Вариант 6, в списке 6)**

Эмпирическая оценка эффективности алгоритмов.

1. Разработать алгоритм Шелла со смещениями Седжвика, реализовать код на языке С++. Сформировать таблицу 1.1 результатов эмпирической оценки сложности сортировки по формату табл. 1 для массива, заполненного случайными числами.

2. Определить ёмкостную сложность алгоритма Шелла со смещениями Седжвика.

3. Разработать алгоритм быстрой сортировки (Хоара), реализовать код на языке С++. Сформировать таблицу 1.2 результатов эмпирической оценки сортировки по формату табл. 1 для массива, заполненного случайными числами.

4. Определить ёмкостную сложность алгоритма быстрой сортировки (Хоара).

5. Добавьте в отчёт данные по работе любого из алгоритмов простой сортировки в среднем случае, полученные в предыдущей практической работе (в отчёте – таблица 1.3).

6. Представить на общем сравнительном графике зависимости Тп(n)=Cф+Mф для трёх анализируемых алгоритмов. График должен быть подписан, на нём – обозначены оси.

7. На основе сравнения полученных данных определите наиболее эффективный из алгоритмов в среднем случае (отдельно для небольших массивов при n до 1000 и для больших массивов с n>1000).

8. Провести дополнительные прогоны программ ускоренной и быстрой сортировок на массивах, отсортированных а) строго в убывающем и б) строго возрастающем порядке значений элементов. Заполнить по этим данным соответствующие таблицы 1.4, 1.5, 1.6 и 1.7 для каждого алгоритма по формату табл. 1.

9. Сделайте вывод о зависимости (или независимости) алгоритмов сортировок от исходной упорядоченности массива на основе результатов, представленных в таблицах.

## **2.2 Математическая модель решения алгоритма**

### **2.2.1 Описание выполнения и блок-схема алгоритма Шелла со смещением Седжвика**

Сортировка Шелла является довольно интересной модификацией алгоритма сортировки простыми вставками.

Рассмотрим следующий алгоритм сортировки массива a[0].. a[15].

1. Вначале сортируем простыми вставками каждые 8 групп из 2-х элементов (a[0], a[8[), (a[1], a[9]), ... , (a[7], a[15]).

2. Потом сортируем каждую из четырех групп по 4 элемента (a[0], a[4], a[8], a[12]), ..., (a[3], a[7], a[11], a[15]).

В нулевой группе будут элементы 4, 12, 13, 18, в первой - 3, 5, 8, 9 и т.п.

3. Далее сортируем 2 группы по 8 элементов, начиная с (a[0], a[2], a[4], a[6], a[8], a[10], a[12], a[14]).

4. В конце сортируем вставками все 16 элементов.

Единственной характеристикой сортировки Шелла является *приращение* - расстояние между сортируемыми элементами, в зависимости от прохода. В конце приращение всегда равно единице - метод завершается обычной сортировкой вставками, но именно последовательность приращений определяет рост эффективности.

Использованный в примере набор ..., 8, 4, 2, 1 - неплохой выбор, особенно, когда количество элементов - степень двойки. Однако гораздо лучший вариант предложил Р.Седжвик. Его последовательность имеет вид



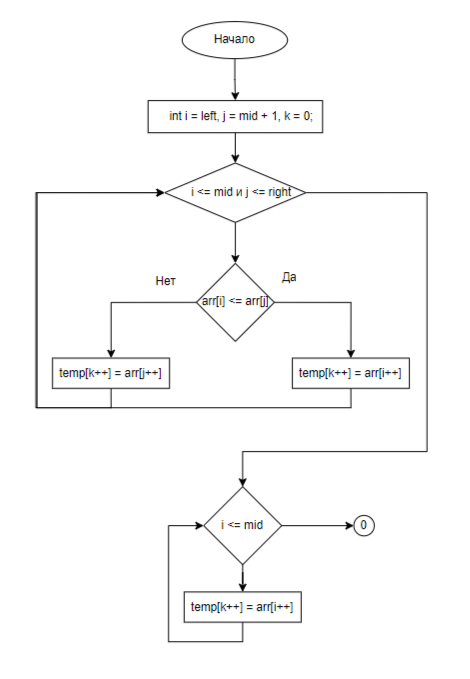


Рисунок 1 – Блок-схема алгоритма сортировки Шелла

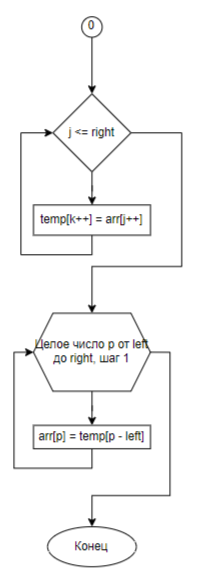


Рисунок 2 – Блок-схема алгоритма сортировки Шелла

### **2.2.2 Определение ситуаций лучшего, среднего и худшего случая и функции роста времени работы алгоритма шейкерной сортировки**

Лучший случай: лучший случай будет, если исходный массив уже почти отсортирован. В этом случае на каждой итерации смещение будет равно 1, и алгоритм будет завершаться после финальной сортировки вставками за время O(n), где n - размер массива.

Средний случай: при использовании смещения Седжвика алгоритм сортировки Шелла обычно достигает хороших результатов на разного вида данных. Сложность среднего случая зависит от выбранной последовательности смещений перед сужением до 1. Обычно сложность времени работы алгоритма в среднем случае составляет примерно O(n log n).

Худший случай: худший случай происходит, когда последовательность элементов в массиве распределена в таком порядке, который делает процесс сортировки максимально затратным. Например, если исходный массив отсортирован в обратном порядке, то сложность худшего случая может быть близка к O(n2).

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 шейкерной сортировки на языке C++**

Для реализации алгоритма сортировки Шелла с использованием смещения Седжвика на языке C++(блок кода 1), понадобятся следующие библиотеки: iostream, random, chrono. Iostream используется для работы с вводом-выводом в C++, random - для генерации случайных чисел в определенном диапазоне, а chrono - для работы с интервалами времени и таймерами.

| #include <iostream> #include <vector> #include <chrono> #include <random> using namespace std;  void shellSort(vector<int>& arr, long& compCount) { //функция сортировки Шелла  int n = arr.size(); // Размер массива  int gaps[] = {701, 301, 132, 57, 23, 10, 4, 1}; // Массив смещений Р. Седжвика   for (int gap : gaps) { // Цикл по каждому смещению  compCount+=2;  for (int i = gap; i < n; i++) { // Цикл по каждому элементу массива, начиная с текущего смещения  int temp = arr[i]; // Сохранение текущего элемента во временной переменной  compCount+=3;  int j; // Объявление индекса для внутреннего цикла  for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) { // Внутренний цикл для сравнения и  перемещения элементов  arr[j] = arr[j - gap]; // Перемещение элемента на позицию смещений  compCount+=2; // Увеличение счетчика сравнений  }   arr[j] = temp; // Вставка сохраненного элемента на правильное место  compCount+=2; // Увеличение счетчика присвоений  }  } compCount+=3; } int main() {  int n;  cout << "Введите размер массива: ";  cin >> n;   vector<int> arr(n);  default\_random\_engine generator; // генератор случайных чисел  uniform\_int\_distribution<int> distribution(0,9);  cout << "Сгенерированный массив: " << endl;  for (int i = 0; i < n; i++) {  arr[i] = distribution(generator); // Генерация случайного числа и присвоение его элементу массива  cout << arr[i] << " ";  }  cout << endl; // Вывод новой строки   long compCount = 0; // Объявление счетчиков для операций  auto start = chrono::high\_resolution\_clock::now(); // Запуск таймера  shellSort(arr, compCount); // Вызов функции сортировки Шелла  auto stop = chrono::high\_resolution\_clock::now(); // Остановка таймера   cout << "Отсортированный массив: " << endl; // Вывод сообщения о сортировке массива  for (int i = 0; i < n; i++) { // Цикл по каждому элементу массива  cout << arr[i] << " "; // Вывод отсортированного числа  }  cout << endl; // Вывод новой строки  auto duration = chrono::duration\_cast<chrono::microseconds>(stop - start); // Вычисление времени сортировки  cout << "Время выполнения сортировки: " << duration.count() << " микросекунд" << endl;  времени выполнения сортировки  cout << "Количество операций сравнения: " << compCount << endl; // Вывод количества операций  сравнения  return 0; } |
| --- |

Блок кода 1 - Шелл со смещением Седжвика

### **2.3.2 Тестирование и построение графика**

Задача программы - протестировать алгоритм сортировки на разных размерах массивов: n=10(рис.3), 100, 1000, 10000, 100000, 1000000. Тестирование проводится с помощью случайной генерации чисел. Результаты тестирования для размеров массивов от 100 до 1000000 будут отображены в таблице 1.1.

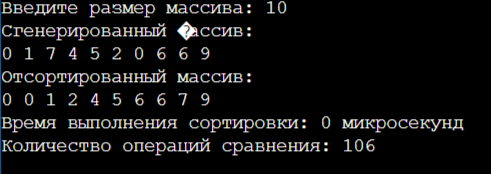


Рисунок 3 - Тестирование программы при n=10

Таблица 1.1. Сводная таблица результатов

| **n** | **T(n), мс** | **Тп(n)=Cф+Mф** |
| --- | --- | --- |
| 100 | 0.012 | 2434 |
| 1000 | 0.164 | 36426 |
| 10000 | 1.223 | 485382 |
| 100000 | 13.15 | 5965462 |
| 1000000 | 235.186 | 70536735 |

## **2.4 Математическая модель решения алгоритма**

### **2.4.1 Описание выполнения и блок-схема алгоритма простого слияния**

Алгоритм простого слияния (также известный как сортировка слиянием) - это эффективный алгоритм сортировки данных, который работает по принципу "разделяй и властвуй". Вот общее описание выполнения алгоритма простого слияния:

Разделение:

Исходный массив данных разделяется на две равные или близкие по размеру части.

Этот процесс разделения продолжается рекурсивно до тех пор, пока каждая подгруппа не будет содержать только один элемент. Такие подгруппы являются уже отсортированными.

Слияние:

После разделения происходит процесс слияния, при котором отсортированные подгруппы объединяются в один отсортированный массив.

Каждый раз при слиянии двух подгрупп сравниваются их элементы, и меньший элемент перемещается в результирующий массив.

Этот процесс продолжается до тех пор, пока все элементы не будут объединены в один упорядоченный массив.

Завершение:

После того как все подгруппы были успешно объединены, алгоритм простого слияния завершается, и на выходе получается отсортированный массив.

Преимущества алгоритма простого слияния включают его стабильность, эффективность на больших объемах данных и возможность эффективного параллельного выполнения.

Реализация данного описания выполнения алгоритма представлена в виде блок-схемы (рис.4).

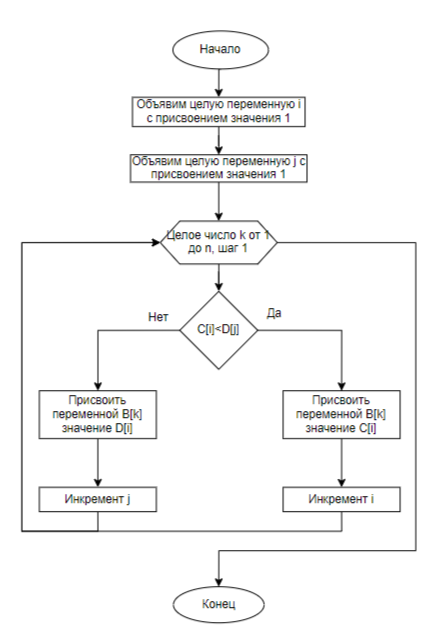


Рисунок 4 – Блок-схема алгоритма простого слияния

### **2.4.2 Определение ситуаций лучшего, среднего и худшего случая и функции роста времени работы алгоритма простого слияния**

a. Лучший случай представляет собой отсортированный массив, что минимизирует количество операций сравнения и перемещения до O(n log2n). Средний случай возникает при случайных числах в массиве и имеет сложность O(n log2n). Худший случай наступает, когда массив отсортирован в обратном порядке, в этом случае количество операций составит O(n2).

b. Временные характеристики:

Лучший случай: O(n log2 n).

Худший случай: O(n2).

В случае данной сортировки, время выполнения в худшем случае увеличивается квадратично по размеру входного массива. Следовательно, можно использовать квадратичную функцию для описания эволюции времени выполнения данного метода сортировки. Время выполнения в лучшем случае увеличивается близким к линейному способом по размеру входного массива.

Сложность по памяти алгоритма будет составлять O(log2n).

## **2.5 Реализация алгоритма на языке C++ и проведение тестирования**

### **2.5.1 Реализация алгоритма простого слияния на языке C++**

Реализуем алгоритм сортировки прямого слияния на языке C++(блок кода 2)понадобятся следующие библиотеки: iostream, random, chrono, vector.

| #include <iostream> #include <vector> #include <random> #include <chrono> using namespace std;  void merge(vector<int>& arr, int left, int mid, int right,long& compCount) {  vector<int> temp(right - left + 1);  int i = left, j = mid + 1, k = 0;  compCount+=7;  while (i <= mid && j <= right) {  if (arr[i] <= arr[j]) {  temp[k++] = arr[i++];  } else {  temp[k++] = arr[j++];  }  compCount+=3;  }   while (i <= mid) {  temp[k++] = arr[i++];  compCount+=2;  }   while (j <= right) {  temp[k++] = arr[j++];  compCount+=2;  }   for (int p = left; p <= right; p++) {  arr[p] = temp[p - left];  compCount+=2;  } }  void merge\_sort(vector<int>& arr, int left, int right,long& compCount) {  if (left < right) {  int mid = left + (right - left) / 2;  merge\_sort(arr, left, mid, compCount);  merge\_sort(arr, mid + 1, right, compCount);  merge(arr, left, mid, right, compCount);  compCount++;  }  compCount++;  } int main() {  int n;  cout << "Введите размер массива: ";   cin >> n;   vector<int> arr(n);   default\_random\_engine generator; // генератор случайных чисел  uniform\_int\_distribution<int> distribution(0,9);  cout << "Сгенерированный массив: " << endl;   for (int i = 0; i < n; i++) {  arr[i] = distribution(generator); // Генерация случайного числа и присвоение его элементу массива  cout << arr[i] << " ";  }  cout << endl; // Вывод новой строки   long compCount = 0; // Объявление счетчиков для операций   auto start = chrono::high\_resolution\_clock::now(); // Запуск таймера  merge\_sort(arr, 0, arr.size() - 1, compCount); // Вызов функции сортировки Шелла  auto stop = chrono::high\_resolution\_clock::now(); // Остановка таймера   cout << "Отсортированный массив: " << endl; // Вывод сообщения о сортировке массива  for (int i = 0; i < n; i++) { // Цикл по каждому элементу массива  cout << arr[i] << " "; // Вывод отсортированного числа  }  cout << endl; // Вывод новой строки  auto duration = chrono::duration\_cast<chrono::microseconds>(stop - start); // Вычисление времени сортировки  cout << "Время выполнения сортировки: " << duration.count() << " микросекунд" << endl;   cout << "Количество операций сравнения: " << compCount << endl; // Вывод количества операций   return 0; } |
| --- |

Блок кода 2 - Простое слияние

### **2.5.2 Тестирование**

Протестируем программу с заданным размером массива n=10 (рис.5), n=100, n=1000, n=10000, n=100000, n=1000000. Продемонстрируем результаты тестирования от n=100 до n=1000000 в таблице 1.2. Воспользуемся библиотекой Chrono для подсчёта затраченного времени на сортировку.

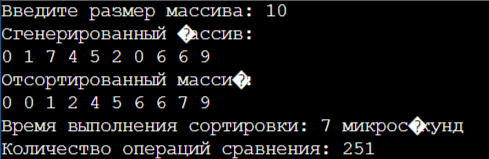


Рисунок 5 - Тестирование программы при n=10

Таблица 1.2. Сводная таблица результатов

| **n** | **T(n), мс** | **Тп(n)=Cф+Mф** |
| --- | --- | --- |
| 100 | 0.034 | 4223 |
| 1000 | 0.351 | 58538 |
| 10000 | 5.642 | 752394 |
| 100000 | 66.24 | 9162394 |
| 1000000 | 486.421 | 10773593 |

## **2.6 Сортировка простым обменом**

Добавим из предыдущей работы таблицу результатов тестирования простой сортировки обменом в среднем случае(табл.1.3).

Таблица 1.3. Сводная таблица результатов

| **n** | **T(n), мс** | **Тп(n)=Cф+Mф** |
| --- | --- | --- |
| 100 | 0.05 | 12236 |
| 1000 | 4.526 | 1234682 |
| 10000 | 394.969 | 122374417 |
| 100000 | 33726.936 | 12254127887 |
| 1000000 | 4215867.28 | 1270495346447 |

## 

## **2.7 Сравнение трёх алгоритмов на графике**

На основе полученных данных, продемонстрированных в таблицах 1.1, 1.2 и 1.3, построим график функции роста Тп алгоритма сортировки Шелла со смещением по Седжвику, простого слияния и сортировки простым обменом в среднем случае от размера массива n. Для наглядности сравнения построим два графика. Первый будет построен на значениях до 1000(рис.6), а второй от 10000 и до 1000000(рис.7). Это позволит нам сделать более точное сравнение.

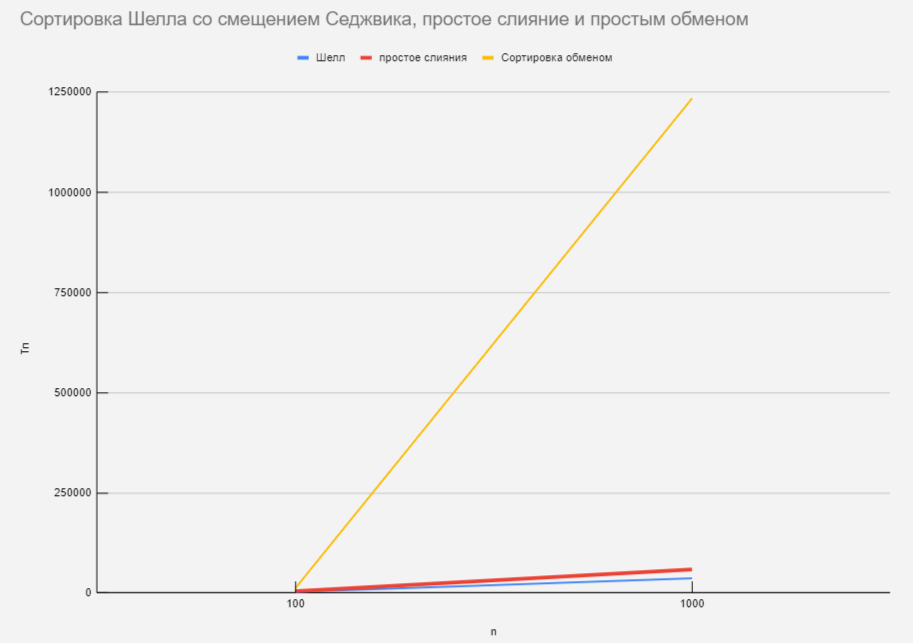


Рисунок 6 - График сравнения трёх сортировок в среднем случае при n до 1000

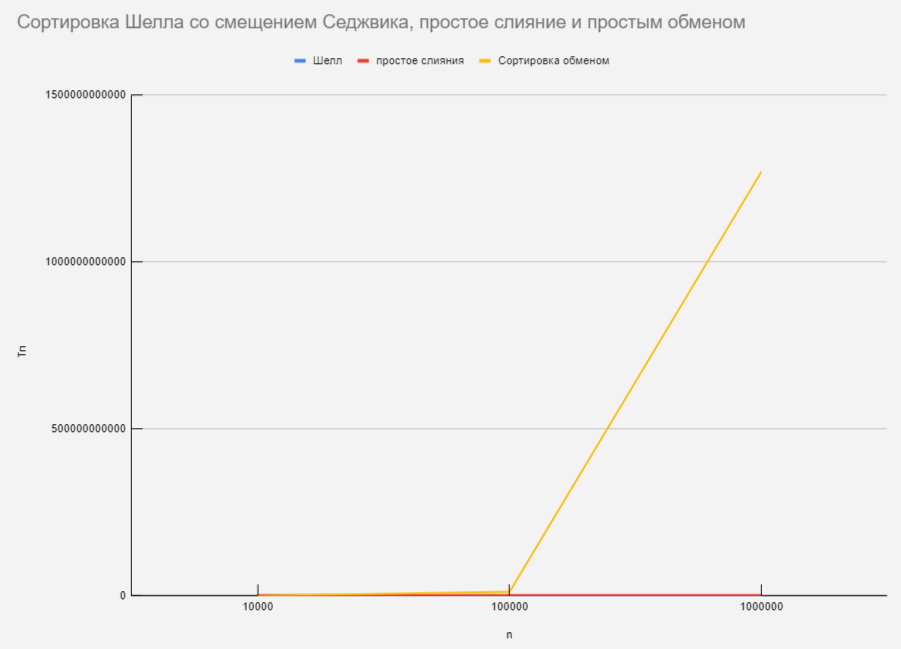


Рисунок 7 - График сравнения трёх сортировок в среднем случае при n от 10000 до 1000000

На основе таблиц 1.1, 1.2 и 1.3, можно сделать вывод, что в среднем случае алгоритм сортировки простого обмена самый неэффективный, алгоритм сортировки простого слияния второй по эффективности, а алгоритм Шелла со смещением по Седжвику самый эффективный.

## **2.8 Тестирование программ для алгоритмов Шелла со смещением Седжвика и простого слияния.**

### **2.8.1 Тестирование при упорядоченном по убыванию элементов массива и построение графика для алгоритма Шелла со смещением Седжвика**

Адаптируем программу к заполнению массива элементами в убывающем порядке (блок кода 3). Программа будет протестирована на массивах с элементами, упорядоченными в строго убывающем порядке для значений n = 100, 1000, 10000, 100000 и 1000000. Продемонстрируем работу программы при n = 10(рис.8). На основе данных, представленных в таблице 1.4.

| #include <iostream> #include <vector> #include <random> #include <chrono> #include <algorithm> using namespace std;  void shellSort(vector<int>& arr,long& compCount) {  // Расчет последовательности Шелла для шага  int n = arr.size();  int gap = 1;    // Начало сортировки  while (gap < n / 3) {  gap = gap \* 3 + 1;  compCount+=2;  }  // Сортировка элементов на расстоянии шага gap  while (gap > 0) {  for (int i = gap; i < n; i++) {  int temp = arr[i];  int j = i;   while (j >= gap && arr[j - gap] > temp) {  arr[j] = arr[j - gap];  j -= gap;  compCount+=3;  }  // Вставка элементов на нужное место  arr[j] = temp;  compCount+=5;  }  // Уменьшаем шаг для следующей итерации  gap = (gap - 1) / 3;  compCount+=3;  }  compCount+=4; } int main() {  int n;  cout << "Введите размер массива: ";   cin >> n;   vector<int> arr(n);   default\_random\_engine generator; // генератор случайных чисел  uniform\_int\_distribution<int> distribution(0,9);  cout << "Сгенерированный массив: " << endl;   for (int i = 0; i < n; i++) {  arr[i] = distribution(generator); // Генерация случайного числа и присвоение его элементу массива  cout << arr[i] << " ";  }  cout << endl; // Вывод новой строки  sort(arr.begin(), arr.end(), greater<int>());  long compCount = 0; // Объявление счетчиков для операций   auto start = chrono::high\_resolution\_clock::now(); // Запуск таймера  shellSort(arr, compCount); // Вызов функции сортировки Шелла  auto stop = chrono::high\_resolution\_clock::now(); // Остановка таймера   cout << "Отсортированный массив: " << endl; // Вывод сообщения о сортировке массива  for (int i = 0; i < n; i++) { // Цикл по каждому элементу массива  cout << arr[i] << " "; // Вывод отсортированного числа  }  cout << endl; // Вывод новой строки  auto duration = chrono::duration\_cast<chrono::microseconds>(stop - start); // Вычисление времени сортировки  cout << "Время выполнения сортировки: " << duration.count() << " микросекунд" << endl;   cout << "Количество операций сравнения: " << compCount << endl; // Вывод количества операций   return 0; } |
| --- |

Блок кода 3 - алгоритм Шелла со смещением Седжвика по убыванию

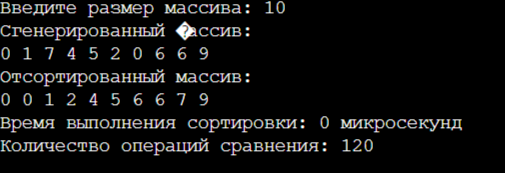


Рисунок 8 – Результат программы при n=10 и с отсортированными значениями по убыванию

Таблица 1.4. Сводная таблица результатов

| **n** | **T(n), мс** | **Тп(n)=Cф+Mф** |
| --- | --- | --- |
| 100 | 0.005 | 2538 |
| 1000 | 0.064 | 32538 |
| 10000 | 0. 586 | 454539 |
| 100000 | 13.684 | 5241353 |
| 1000000 | 135.583 | 63954594 |

### **2.8.2 Тестирование при упорядоченном по возрастанию элементов массива и построение графика для алгоритма Шелла со смещением Седжвика**

Адаптируем программу к заполнению массива элементами в возрастающем порядке (блок кода 4). Программа будет протестирована на массивах с элементами, упорядоченными в строго возрастающем порядке для значений n = 100, 1000, 10000, 100000 и 1000000. Продемонстрируем работу программы при n = 10(рис.9). На основе данных, представленных в таблице 1.5.

| #include <iostream> #include <vector> #include <random> #include <chrono> #include <algorithm> using namespace std;  void shellSort(vector<int>& arr,long& compCount) {  // Расчет последовательности Шелла для шага  int n = arr.size();  int gap = 1;    // Начало сортировки  while (gap < n / 3) {  gap = gap \* 3 + 1;  compCount+=2;  }  // Сортировка элементов на расстоянии шага gap  while (gap > 0) {  for (int i = gap; i < n; i++) {  int temp = arr[i];  int j = i;   while (j >= gap && arr[j - gap] > temp) {  arr[j] = arr[j - gap];  j -= gap;  compCount+=3;  }  // Вставка элементов на нужное место  arr[j] = temp;  compCount+=5;  }  // Уменьшаем шаг для следующей итерации  gap = (gap - 1) / 3;  compCount+=3;  }  compCount+=4; } int main() {  int n;  cout << "Введите размер массива: ";   cin >> n;   vector<int> arr(n);   default\_random\_engine generator; // генератор случайных чисел  uniform\_int\_distribution<int> distribution(0,9);  cout << "Сгенерированный массив: " << endl;   for (int i = 0; i < n; i++) {  arr[i] = distribution(generator); // Генерация случайного числа и присвоение его элементу массива  cout << arr[i] << " ";  }  cout << endl; // Вывод новой строки  sort(arr.begin(), arr.end());  long compCount = 0; // Объявление счетчиков для операций   auto start = chrono::high\_resolution\_clock::now(); // Запуск таймера  shellSort(arr, compCount); // Вызов функции сортировки Шелла  auto stop = chrono::high\_resolution\_clock::now(); // Остановка таймера   cout << "Отсортированный массив: " << endl; // Вывод сообщения о сортировке массива  for (int i = 0; i < n; i++) { // Цикл по каждому элементу массива  cout << arr[i] << " "; // Вывод отсортированного числа  }  cout << endl; // Вывод новой строки  auto duration = chrono::duration\_cast<chrono::microseconds>(stop - start); // Вычисление времени сортировки  cout << "Время выполнения сортировки: " << duration.count() << " микросекунд" << endl;   cout << "Количество операций сравнения: " << compCount << endl; // Вывод количества операций   return 0; } |
| --- |

Блок кода 4 - Шелл со смещением Седжвика по возрастанию

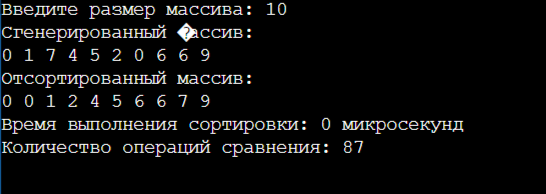


Рисунок 9 – Результат тестирования при n=10 и с отсортированными значениями по возрастанию

Таблица 1.5. Сводная таблица результатов

| **n** | **T(n), мс** | **Тп=Cп+Mп** |
| --- | --- | --- |
| 100 | 0.003 | 1741 |
| 1000 | 0.054 | 27593 |
| 10000 | 0.458 | 374395 |
| 100000 | 5.539 | 4833575 |
| 1000000 | 82.248 | 590239485 |

### **2.8.3 Тестирование при упорядоченном по убыванию элементов массива и построение графика для алгоритма простого слияния**

Адаптируем программу к заполнению массива элементами в убывающем порядке (блок кода 5). Программа будет протестирована на массивах с элементами, упорядоченными в строго убывающем порядке для значений n = 100, 1000, 10000, 100000 и 1000000. Продемонстрируем работу программы при n = 10(рис.10). На основе данных, представленных в таблице 1.6.

| #include <iostream> #include <vector> #include <random> #include <chrono> #include <algorithm> using namespace std;  void merge(vector<int>& arr, int left, int mid, int right,long& compCount) {  vector<int> temp(right - left + 1);  int i = left, j = mid + 1, k = 0;  compCount+=7;  while (i <= mid && j <= right) {  if (arr[i] <= arr[j]) {  temp[k++] = arr[i++];  } else {  temp[k++] = arr[j++];  }  compCount+=3;  }   while (i <= mid) {  temp[k++] = arr[i++];  compCount+=2;  }   while (j <= right) {  temp[k++] = arr[j++];  compCount+=2;  }   for (int p = left; p <= right; p++) {  arr[p] = temp[p - left];  compCount+=2;  } }  void merge\_sort(vector<int>& arr, int left, int right,long& compCount) {  if (left < right) {  int mid = left + (right - left) / 2;  merge\_sort(arr, left, mid, compCount);  merge\_sort(arr, mid + 1, right, compCount);  merge(arr, left, mid, right, compCount);  compCount++;  }  compCount++;  } int main() {  int n;  cout << "Введите размер массива: ";   cin >> n;   vector<int> arr(n);   default\_random\_engine generator; // генератор случайных чисел  uniform\_int\_distribution<int> distribution(0,9);  cout << "Сгенерированный массив: " << endl;   for (int i = 0; i < n; i++) {  arr[i] = distribution(generator); // Генерация случайного числа и присвоение его элементу массива  cout << arr[i] << " ";  }  cout << endl; // Вывод новой строки  sort(arr.begin(), arr.end(), greater<int>());  long compCount = 0; // Объявление счетчиков для операций   auto start = chrono::high\_resolution\_clock::now(); // Запуск таймера  merge\_sort(arr, 0, arr.size() - 1, compCount); // Вызов функции сортировки Шелла  auto stop = chrono::high\_resolution\_clock::now(); // Остановка таймера   cout << "Отсортированный массив: " << endl; // Вывод сообщения о сортировке массива  for (int i = 0; i < n; i++) { // Цикл по каждому элементу массива  cout << arr[i] << " "; // Вывод отсортированного числа  }  cout << endl; // Вывод новой строки  auto duration = chrono::duration\_cast<chrono::microseconds>(stop - start); // Вычисление времени сортировки  cout << "Время выполнения сортировки: " << duration.count() << " микросекунд" << endl;   cout << "Количество операций сравнения: " << compCount << endl; // Вывод количества операций   return 0; } |
| --- |

Блок кода 5 - Простое слияние по убыванию

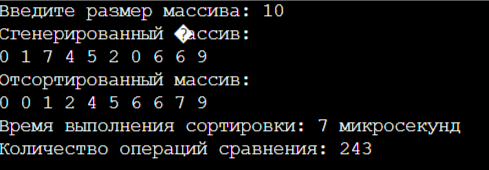


Рисунок 10 – Результат программы при n=10 и с отсортированными значениями по убыванию

Таблица 1.6. Сводная таблица результатов

| **n** | **T(n), мс** | **Тп(n)=Cф+Mф** |
| --- | --- | --- |
| 100 | 0.002 | 1835 |
| 1000 | 0.02 | 26353 |
| 10000 | 0.532 | 341247 |
| 100000 | 10.247 | 4536830 |
| 1000000 | 153.547 | 554775839 |

### **2.8.4 Тестирование при упорядоченном по возрастанию элементов массива и построение графика для алгоритма простого слияния**

Адаптируем программу к заполнению массива элементами в возрастающем порядке (блок кода 6). Программа будет протестирована на массивах с элементами, упорядоченными в строго возрастающем порядке для значений n = 100, 1000, 10000, 100000 и 1000000. Продемонстрируем работу программы при n = 10(рис.11). На основе данных, представленных в таблице 1.7.

| #include <iostream> #include <vector> #include <random> #include <chrono> #include <algorithm> using namespace std;  void merge(vector<int>& arr, int left, int mid, int right,long& compCount) {  vector<int> temp(right - left + 1);  int i = left, j = mid + 1, k = 0;  compCount+=7;  while (i <= mid && j <= right) {  if (arr[i] <= arr[j]) {  temp[k++] = arr[i++];  } else {  temp[k++] = arr[j++];  }  compCount+=3;  }   while (i <= mid) {  temp[k++] = arr[i++];  compCount+=2;  }   while (j <= right) {  temp[k++] = arr[j++];  compCount+=2;  }   for (int p = left; p <= right; p++) {  arr[p] = temp[p - left];  compCount+=2;  } }  void merge\_sort(vector<int>& arr, int left, int right,long& compCount) {  if (left < right) {  int mid = left + (right - left) / 2;  merge\_sort(arr, left, mid, compCount);  merge\_sort(arr, mid + 1, right, compCount);  merge(arr, left, mid, right, compCount);  compCount++;  }  compCount++;  } int main() {  int n;  cout << "Введите размер массива: ";   cin >> n;   vector<int> arr(n);   default\_random\_engine generator; // генератор случайных чисел  uniform\_int\_distribution<int> distribution(0,9);  cout << "Сгенерированный массив: " << endl;   for (int i = 0; i < n; i++) {  arr[i] = distribution(generator); // Генерация случайного числа и присвоение его элементу массива  cout << arr[i] << " ";  }  cout << endl; // Вывод новой строки  sort(arr.begin(), arr.end());  long compCount = 0; // Объявление счетчиков для операций   auto start = chrono::high\_resolution\_clock::now(); // Запуск таймера  merge\_sort(arr, 0, arr.size() - 1, compCount); // Вызов функции сортировки Шелла  auto stop = chrono::high\_resolution\_clock::now(); // Остановка таймера   cout << "Отсортированный массив: " << endl; // Вывод сообщения о сортировке массива  for (int i = 0; i < n; i++) { // Цикл по каждому элементу массива  cout << arr[i] << " "; // Вывод отсортированного числа  }  cout << endl; // Вывод новой строки  auto duration = chrono::duration\_cast<chrono::microseconds>(stop - start); // Вычисление времени сортировки  cout << "Время выполнения сортировки: " << duration.count() << " микросекунд" << endl;   cout << "Количество операций сравнения: " << compCount << endl; // Вывод количества операций   return 0; } |
| --- |

Блок кода 6 - Простое слияние по возрастанию

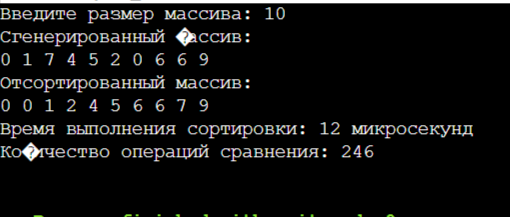


Рисунок 11 – Результат тестирования при n=10 и с отсортированными значениями по возрастанию

Таблица 1.7. Сводная таблица результатов

| **n** | **T(n), мс** | **Тп(n)=Cф+Mф** |
| --- | --- | --- |
| 100 | 0.003 | 1832 |
| 1000 | 0.053 | 26234 |
| 10000 | 0.539 | 335386 |
| 100000 | 18.539 | 4533542 |
| 1000000 | 102.530 | 52450539 |

## **2.9 Вывод по заданию №1**

Исходя из результатов, можно сделать вывод, что алгоритмы сортировки Шелла со смещением Седжвика и простого слияния не зависят от исходной упорядоченности массива. Оба алгоритма работают эффективно как на упорядоченных, так и на неупорядоченных массивах, сохраняя свою скорость и производительность. Таким образом, можно использовать эти алгоритмы независимо от того, как упорядочен исходный массив.

# 

# 3 ЗАДАНИЕ №2

## **3.1 Формулировка задачи**

Асимптотический анализ сложности алгоритмов

Требования по выполнению задания

1. Из материалов предыдущей практической работы приведите в отчёте формулы Тт(n) функций роста алгоритма простой сортировки обменом в лучшем и худшем случае.

2. На основе определений соответствующих нотаций получите асимптотическую оценку вычислительной сложности простого алгоритма сортировки обменом:

- в О-нотации (оценка сверху) для анализа худшего случая;

- в Ω-нотации (оценка снизу) для анализа лучшего случая.

3. Получите (если это возможно) асимптотически точную оценку вычислительной сложности алгоритма в нотации θ.

4. Реализуйте графическое представление функции роста и полученных асимптотических оценок сверху и снизу.

5. Привести справочную информацию о вычислительной сложности алгоритмов Шелла со смещением Седжвика и простого слияния.

6. Общие результаты свести в табл. 2.

7. Сделать вывод о наиболее эффективном алгоритме из трёх.

## **3.2 Формулы функции роста алгоритма сортировки простым обменом в худшем и лучшем случае**

В лучшем случае, когда массив уже отсортирован, количество операций будет минимальным и составит Тт(n)=n, что означает, что алгоритм будет работать за оптимальное время.

В среднем случае, когда массив заполнен случайными числами, алгоритм будет иметь сложность Тт(n)=(n2-n)/2. Это означает, что алгоритм будет требовать больше операций по сравнению с лучшим случаем, но все равно будет работать эффективно.

В худшем случае, когда массив отсортирован в обратном порядке, количество операций также будет Тт(n)=3\*(n2-n)/2. Это означает, что алгоритм будет работать медленнее в этом случае из-за большего количества операций.

Ёмкостная сложность алгоритма будет равна O(1), что означает, что он будет потреблять постоянное количество памяти независимо от размера входных данных.

## **3.3 Асимптотическая оценка вычислительной сложности простого алгоритма сортировки обменом**

Асимптотическая оценка вычислительной сложности простого алгоритма сортировки обменом для худшего случая в О-нотации (оценка сверху) будет равна О(n2).

Асимптотическая оценка вычислительной сложности простого алгоритма сортировки обменом для лучшего случая в Ω-нотации (оценка снизу) будет равна Ω(n).

Асимптотическая оценка вычислительной сложности простого алгоритма сортировки обменом для среднего случая в θ-нотации будет равна θ(n2).

Ёмкостная сложность алгоритма простой сортировки обменом O(1).

## **3.4 Графическое представление функции роста и полученных асимптотических оценок сверху и снизу**

На полученных данных в пунктах 3.2 и 3.3, мы можем сделать графическое представление роста и полученных асимптотических оценок сверху и снизу(рис.).

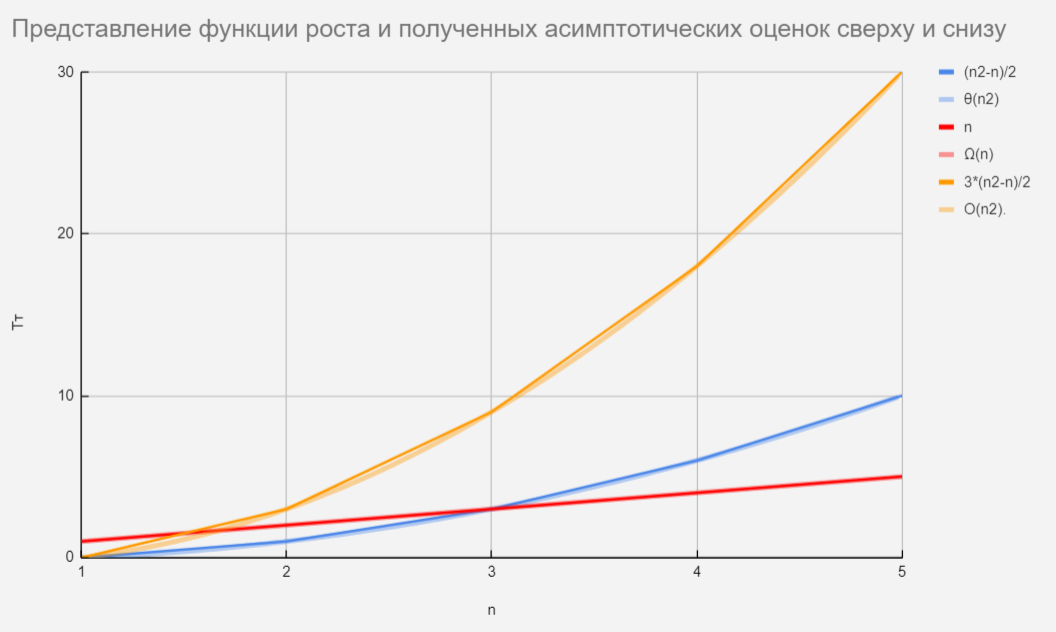


Рисунок - Графическое представление функции роста и полученных асимптотических оценок сверху и снизу

## **3.5 Справочная информация о вычислительной сложности алгоритма Шелла со смещением Седжвика и простого слияния**

### **3.5.1 Справочная информация о вычислительной сложности алгоритма Шелла со смещением Седжвика**

В худшем случае алгоритм шейкерной сортировки имеет оценку сложности O(n3/2), что указывает на возможность значительного увеличения времени выполнения при больших объемах данных. В лучшем случае сложность данного алгоритма составляет Ω(nlog(n)), что говорит о более эффективной работе при оптимальных условиях. Для среднего случая оценка сложности шейкерной сортировки в θ-нотации равна θ(n1.3), что подчеркивает его нестабильность при обработке различных входных данных.

Ёмкостная сложность алгоритма шейкерной сортировки оценивается как O(1), что говорит о небольшом объеме дополнительной памяти, необходимой для его работы.

### **3.5.2 Справочная информация о вычислительной сложности алгоритма простого слияния**

Асимптотическая оценка вычислительной сложности алгоритма простого слияния для худшего случая равна О(nlog2n).

Асимптотическая оценка вычислительной сложности алгоритма простого слияния для лучшего случая равна Ω(nlog2n).

Асимптотическая оценка вычислительной сложности простого слияния для среднего случая равна θ(nlog2n).

Ёмкостная сложность алгоритма простого слияния равна О(n).

## **3.6 Таблица асимптотической сложности трёх алгоритмов**

На основе данных из пункта 3.3 и 3.5 заполним таблицу 2 асимптотической сложности алгоритма для алгоритмов сортировки простого обмена, Шелла со смещением Седжвика и простым слиянием. А также укажем ёмкостную сложность данных алгоритмов сортировок.

Таблица 2. Сводная таблица результатов

| Алгоритм | Асимптотическая сложность алгоритма | | | |
| --- | --- | --- | --- | --- |
| Наихудший случай (сверху) | Наилучший случай (снизу) | Средний случай (точная оценка) | Ёмкостная сложность |
| Простой обмен | О(n2) | Ω(n) | θ(n2) | О(1) |
| Шелла со смещением Седжвика | О(n3/2) | Ω (nlog2n) | θ(n1.3) | О(1) |
| Простым слиянием | О(nlog2n) | Ω(nlog2n) | θ(nlog2n) | О(n) |

**3.7 Выводы по заданию №2**

Исходя из выше обсужденного, можно сделать вывод о том, что наиболее эффективным алгоритмом из простого слияния, Шелла со смещением Седжвика и простого обмена будет Шелл с смещением Седжвика.

Шелл с смещением Седжвика обладает следующими преимуществами:

1. Он основан на алгоритме Шелла, который обычно показывает хорошую производительность на практике.

2. Использование смещения Седжвика позволяет улучшить эффективность алгоритма, уменьшая количество сравнений и перемещений элементов.

3. Сложность Шелла с смещением Седжвика составляет примерно O(n log n), что позволяет ему эффективно сортировать массивы любого размера.

Таким образом, Шелл с смещением Седжвика будет наиболее эффективным выбором среди рассмотренных алгоритмов сортировки.

# 5 ВЫВОДЫ

В ходе выполнения практической работы были осуществлены следующие задачи:

- Получены навыки по анализу вычислительной сложности алгоритмов сортировки и выбору наиболее эффективного метода;

- Проанализированы алгоритмы простого слияния и сортировки Шелла с последовательностью Седжвика;

- Написаны программы для алгоритмов простого слияния и Шелла с последовательностью Седжвика;

- Проведено тестирование программ для упомянутых алгоритмов сортировки;

- Построены графики функций роста времени для алгоритмов простого слияния и сортировки Шелла с последовательностью Седжвика;

- Осуществлено сравнение алгоритмов простой обмена, простого слияния и сортировки Шелла с последовательностью Седжвика;

- Проведен анализ асимптотической сложности алгоритмов простой обмена, простого слияния и сортировки Шелла с последовательностью Седжвика;

- Выполнено сравнение асимптотической сложности упомянутых алгоритмов сортировки;

- Определено наиболее эффективный алгоритм. Таким образом, главную цель практической работы - получение навыков по анализу вычислительной сложности алгоритмов сортировки и выбору наиболее эффективного метода, можно считать достигнутой.

# 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. Кораблин Ю.П. Структуры и алгоритмы обработки данных: учебно-методическое пособие / Ю.П. Кораблин, В.П. Сыромятников, Л.А. Скворцова. – М.: РТУ МИРЭА, 2020. — 219 с.

5. Кормен Т.Х. и др. Алгоритмы: построение и анализ, 3-е изд. – М.: ООО «И.Д.Вильямс», 2013. – 1328 с.

6. Макконнелл Дж. Основы современных алгоритмов. Активный обучающий метод. 3-е доп. изд., - М.: Техносфера, 2018. – 416 с.

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

8. Скиена С. Алгоритмы. Руководство по разработке, - 2-е изд. – СПб: БХВ-Петербург, 2011. – 720 с.

9. Хайнеман Д. и др. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд. – СПб: ООО «Альфа-книга», 2017. – 432 с.

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

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