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: Величко В.Д.

Группа: ИКБО-74-23

Москва – 2024

СОДЕРЖАНИЕ

1 ЦЕЛЬ

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

2 ЗАДАНИЕ №1

2.1 Формулировка задачи (Вариант 5, в списке 5)

Эмпирическая оценка эффективности алгоритмов.

1. Разработать алгоритм Шелла со смещениями Д. Кнута вторым способом, реализовать код на языке C++. 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 Д. Кнута вторым способом.

3. Разработать алгоритм простого слияния, реализовать код на языке C++. 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 $T_п(n)=C_ф+M_ф$ 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 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 сортировки Шелла со смещениями Д.Кнута вторым способом

Сортировка Шелла (англ. Shell sort) — алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. Иными словами — это сортировка вставками с предварительными «грубыми» проходами.

Алгоритм Шелла с смещениями Д. Кнута (второй способ) является улучшенной версией классического алгоритма Шелла. Он основан на идее использования последовательности чисел, которая строится на основе чисел Фибоначчи.

Алгоритм начинается с определения последовательности смещений, которая будет использоваться для сортировки элементов. Затем происходит проход по этой последовательности, начиная с самого большого смещения и заканчивая самым маленьким. Каждый проход по последовательности осуществляется сортировкой подмассива элементов с использованием сортировки вставками.

Таким образом, используя последовательность смещений, алгоритм Шелла с смещениями Д. Кнута улучшает производительность сортировки за счет более эффективного уменьшения количества инверсий в массиве. Этот подход позволяет сократить время сортировки и повысить производительность алгоритма

Реализация данного описания выполнения алгоритма представлена в виде блок-схемы (рис.1,2).

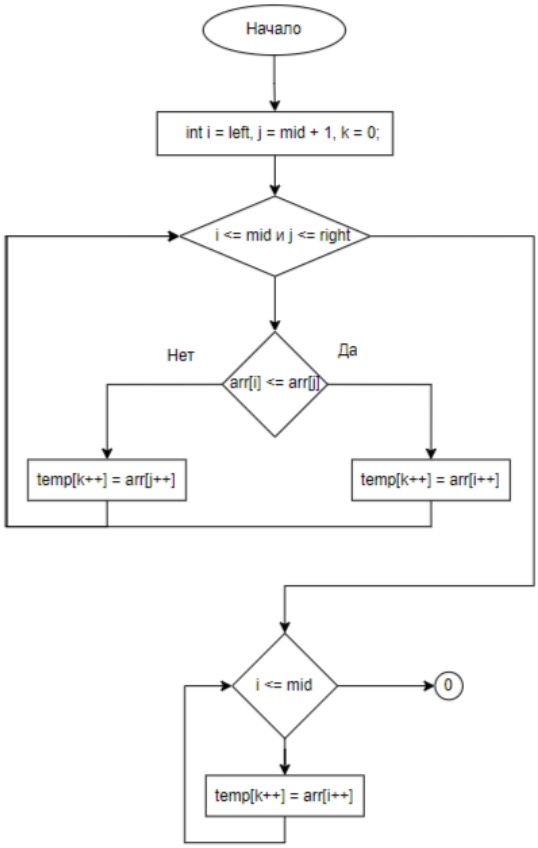


Рисунок 1 – Блок-схема алгоритма сортировки Шелла

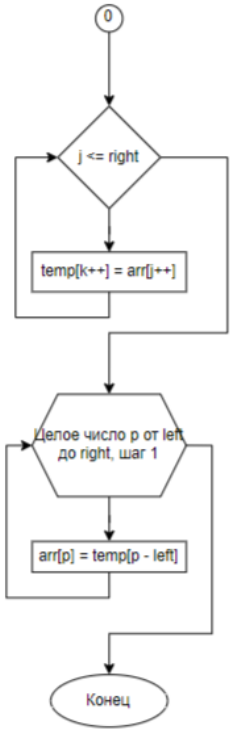


Рисунок 2 – Блок-схема алгоритма сортировки Шелла

2.2.2 Доказательство корректности циклов алгоритма сортировки Шелла со смещениями Д.Кнута вторым способом

Инвариант для цикла: значение переменной i всегда меньше или равна mid и j всегда меньше или равна right.

Инвариант для цикла: значение переменной i всегда меньше или равна mid.

Инвариант для цикла: значение переменной j всегда меньше или равна right.

Инвариант для цикла: значение переменной p всегда меньше right.

Докажем конечность циклов. Пусть дан массив из n элементов. После определения последовательности шагов и начала выполнения сортировки для каждого шага, происходит перемещение элементов массива на заданный шаг влево и сравнение их с элементом на этом же шаге слева от них. Когда мы достигаем шага 1, это означает сравнение и обмен элементов соседних позиций. Таким образом, с каждым проходом по массиву уменьшается количество инверсий и элементы приближаются к своему правильному месту. По мере уменьшения шага сортировки, количество инверсий также уменьшается. Таким образом, поскольку количество инверсий в массиве ограничено и уменьшается с каждым проходом по нему сортировке Шелла, циклы в алгоритме сортировки Шелла будут иметь конечную длину и завершатся в конечном количестве итераций.

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 Шелла со смещениями Д.Кнута вторым способом

В лучшем случае массив уже отсортирован по возрастанию. Тогда количество операций сравнений и перемещений будет составлять $O(n \log_2 n)$.

В среднем случае массив заполнен случайными числами. Тогда количество операций сравнений и перемещений будет составлять $O(n \log_2 n^2)$.

В худшем случае массив отсортирован по убыванию. Тогда количество операций сравнений и перемещений будет составлять $O(n \log_2 n^2)$.

Функции роста времени: лучший случай: $O(n \log_2 n)$, худший случай:$O(n \log_2 n^2)$.

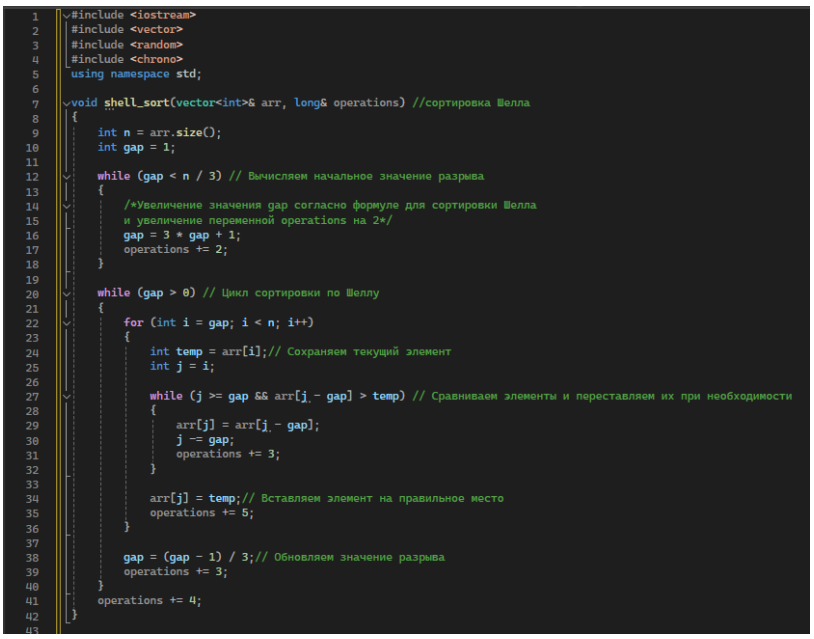
Время исполнения в худшем случае увеличивается квадратично с ростом размера входного массива. Время исполнения в лучшем случае увеличивается линейно с ростом размера входного массива.

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 Шелла со смещениями Д.Кнута вторым способом сортировки на языке C++

Реализуем алгоритм сортировки Шелла со смещением вторым способом на языке C++(рис.3,4). Используем встроенные библиотеки iostream, random, chrono и vector. Vector — это шаблон класса для контейнеров последовательности. Вектор хранит элементы заданного типа в линейном расположении и обеспечивает быстрый случайный доступ к любому элементу. Iostream — это заголовочный файл с классами, функциями и переменными для организации ввода-вывода в языке программирования C++. Random - позволяет генерировать случайные числа в диапазоне.



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

void shell_sort(vector<int>& arr, long& operations) //сортировка Шелла
{
    int n = arr.size();
    int gap = 1;

    while (gap < n / 3) // Вычисляем начальное значение разрыва
    {
        /*Увеличение значения gap согласно формуле для сортировки Шелла
        и увеличение переменной operations на 2*/
        gap = 3 * gap + 1;
        operations += 2;
    }

    while (gap > 0) // Цикл сортировки по Шеллу
    {
        for (int i = gap; i < n; i++)
        {
            int temp = arr[i];// Сохраняем текущий элемент
            int j = i;

            while (j >= gap && arr[j - gap] > temp) // Сравниваем элементы и переставляем их при необходимости
            {
                arr[j] = arr[j - gap];
                j -= gap;
                operations += 3;
            }

            arr[j] = temp;// Вставляем элемент на правильное место
            operations += 5;
        }

        gap = (gap - 1) / 3;// Обновляем значение разрыва
        operations += 3;
    }
    operations += 4;
}
```

Рисунок 3 – Программа алгоритма сортировки Шелла со смещениями Д.Кнута вторым способом

```cpp
int main() {
    setlocale(LC_ALL, "RUS");
    long operations = 0;
    int n, a;
    cout << "Введите n: ";
    cin >> n;
    vector<int> arr(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); //Для заданного диапазона
        cout << "Массив:" << endl;
        for (int i = 0; i < n; i++)
        {
            arr[i] = dist(gen);
            cout << arr[i] << " ";
        }
        cout << endl;
    }
    else if (a == 2)
    {
        cout << "Введите элементы массива:" << endl;
        for (int i = 0; i < n; i++)
        {
            cin >> arr[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*/
    shell_sort(arr, 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 << "Сортированный массив:\n";
    for (int i = 0; i < n; ++i)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "Было совершено операций сравнения и присваивания: " << operations << endl;
    cout << "Сортировка заняла " << duration << " микросекунд" << endl;
    return 0;
}
```

Рисунок 4 – Функция main для алгоритма сортировки Шелла со смещениями Д.Кнута вторым способом

2.3.2 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 1.1. Воспользуемся библиотекой Chrono для подсчёта затраченного времени на сортировку.

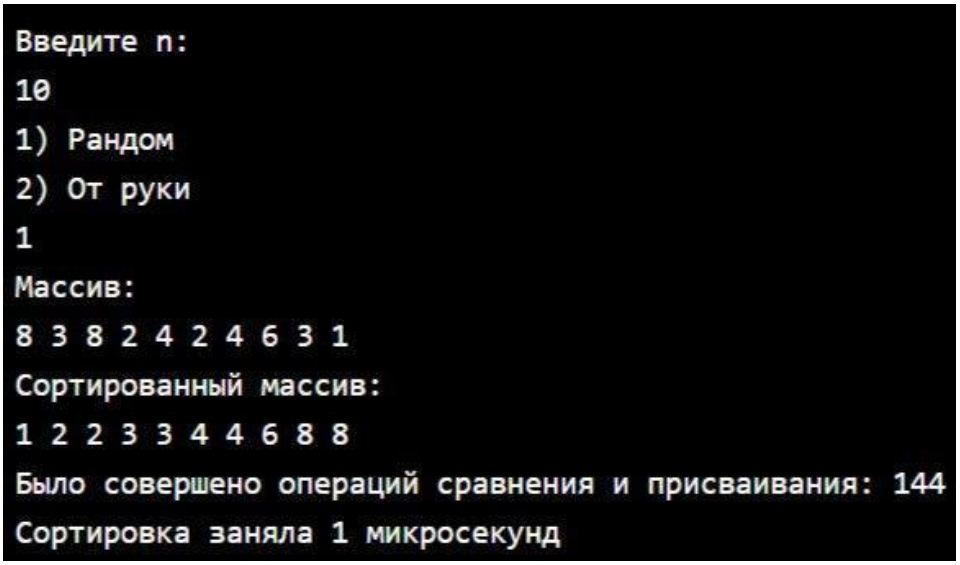


Рисунок 5 - Тестирование программы при n=10

Таблица 1.1. Сводная таблица результатов

n	**T(n), мс**	**$T_п(n)=C_ф+M_ф$**
100	0.01	2608
1000	0.112	36335
10000	1.634	489515
100000	14.317	594966
1000000	165.343	70607821

2.3.3 Построение графика

По таблице 1.1, построим график функции роста $T_п$ алгоритма сортировки Шелла со смещениями вторым способом в среднем случае от размера массива n (рис.6).

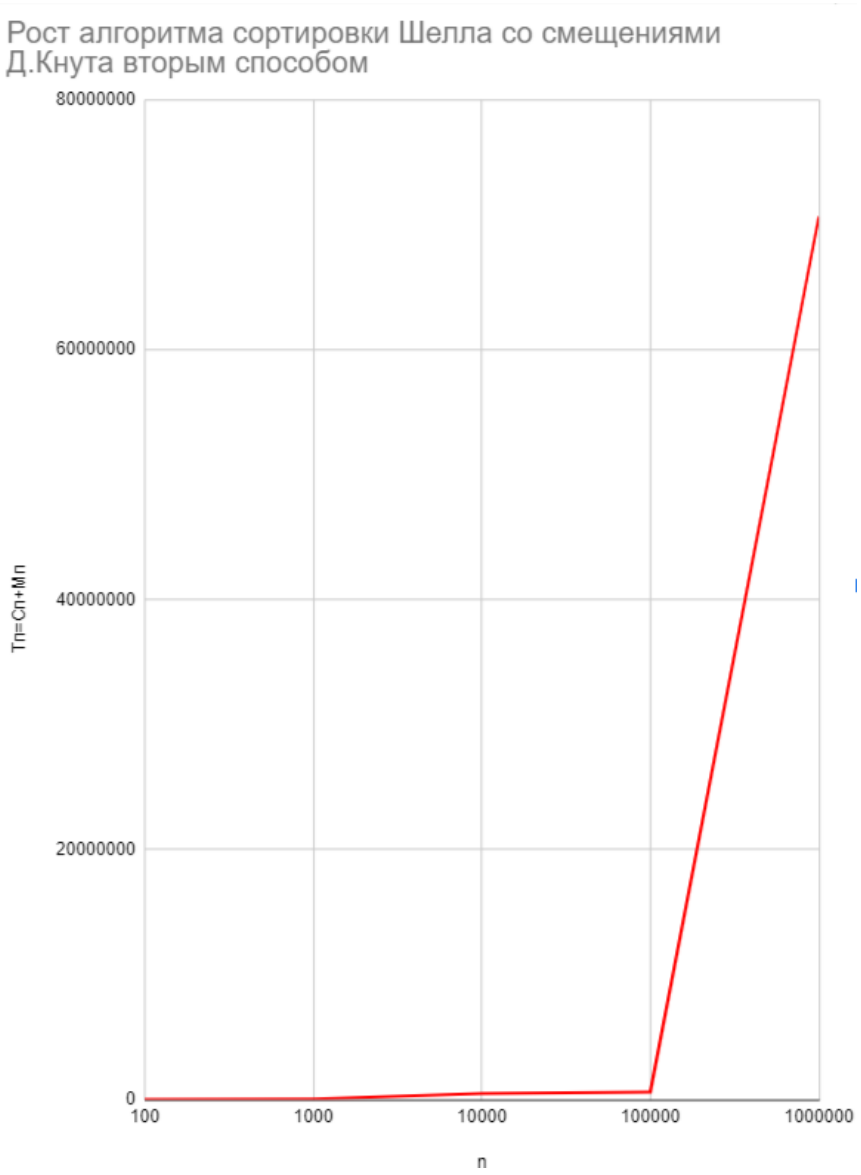


Рисунок 6 - График функции роста $T_п$ алгоритма алгоритма сортировки Шелла со смещениями вторым способом от размера массива n

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.7).

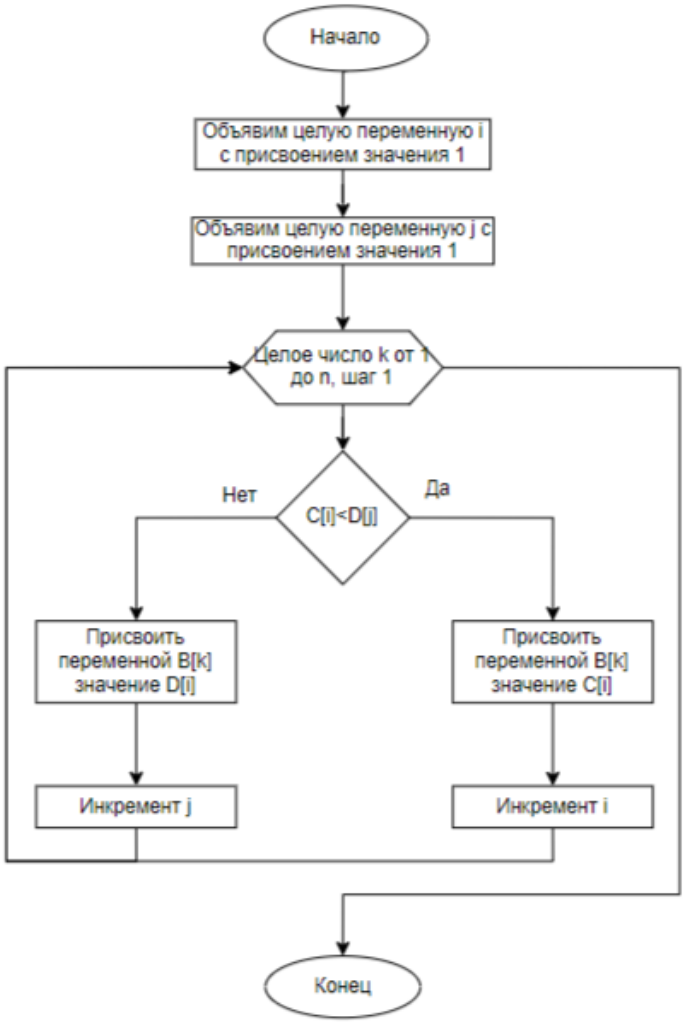


Рисунок 7 – Блок-схема алгоритма простого слияния

2.4.2 Доказательство корректности циклов алгоритма простого слияния

Инвариант для цикла: значение переменной k меньше n.

На каждом шаге алгоритма исходный массив разбивается на меньшие подмассивы размером в два раза меньше. Этот процесс повторяется рекурсивно до тех пор, пока размер подмассива не достигнет 1 элемента. Таким образом, количество разбиений исходного массива конечно и зависит от его исходного размера. Слияние: После разбиения и сортировки подмассивов, происходит их последовательное слияние попарно. При этом каждый следующий шаг алгоритма уменьшает количество подмассивов в два раза. Таким образом,

количество шагов слияния также ограничено и зависит от начального размера массива.

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

2.4.3 Определение ситуаций лучшего, среднего и худшего случая и функции роста времени работы алгоритма простого слияния

a. Лучший случай - массив уже отсортирован. В этом случае количество операций сравнения и перемещения будет минимальным и будет составлять $O(n\log_2 n)$.

Средний случай - массив заполнен случайными числами. В этом случае алгоритм будет иметь сложность $O(n\log_2 n)$.

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

b. Функции роста времени:

Лучший случай: $O(n\log_2 n)$.

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 квазилинейным ростом размера входного массива.

Ёмкостная сложность алгоритма будет равна $O(\log_2 n)$.

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

2.5.1 Реализация алгоритма простого слияния на языке C++

Реализуем алгоритм сортировки Шелла со смещением вторым способом на языке C++(рис.3,4). Используем встроенные библиотеки iostream, random, chrono

и vector. Vector — это шаблон класса для контейнеров последовательности. Вектор хранит элементы заданного типа в линейном расположении и обеспечивает быстрый случайный доступ к любому элементу. Iostream — это заголовочный файл с классами, функциями и переменными для организации ввода-вывода в языке программирования C++. Random - позволяет генерировать случайные числа в диапазоне.

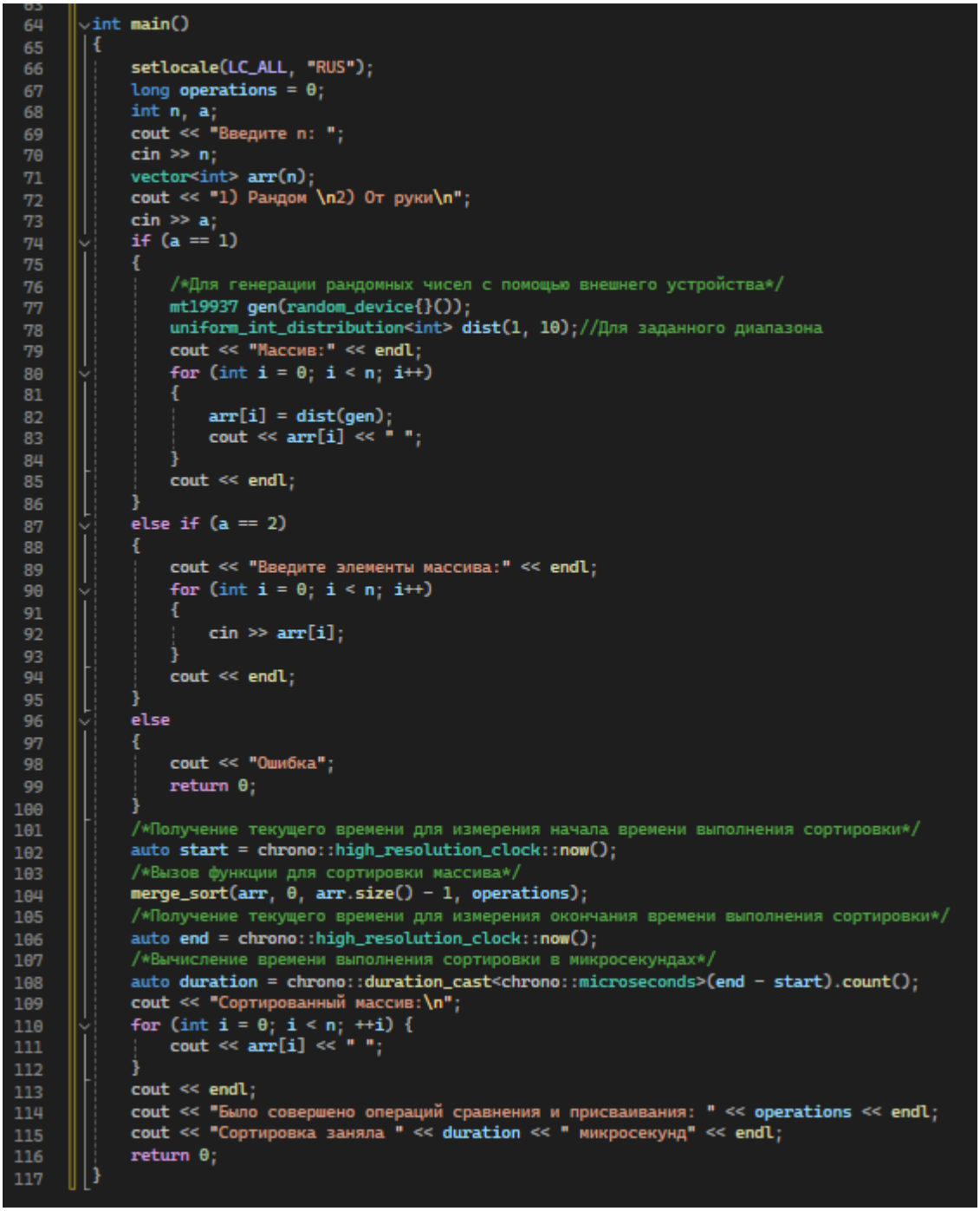
```cpp
#include <iostream>
#include <vector>
#include <random>
#include <chrono>
#include <algorithm>
using namespace std;

/* Объявление функции merge для слияния двух частей массива в отсортированный массив*/
void merge(vector<int>& arr, int left, int mid, int right, long& operations)
{
    vector<int> temp(right - left + 1);
    /*Инициализация переменных i, j, k для перемещения по частям массива и временного массива.*/
    int i = left, j = mid + 1, k = 0;
    operations += 7;
    /*Цикл while для слияния двух отсортированных частей массива*/
    while (i <= mid && j <= right)
    {
        if (arr[i] <= arr[j])
        {
            temp[k++] = arr[i++];
        }
        else
        {
            temp[k++] = arr[j++];
        }
        operations += 3;
    }
    /*Циклы while для добавления оставшихся элементов из частей массива во временный массив*/
    while (i <= mid)
    {
        temp[k++] = arr[i++];
        operations += 2;
    }

    while (j <= right)
    {
        temp[k++] = arr[j++];
        operations += 2;
    }

    for (int p = left; p <= right; p++)
    {
        arr[p] = temp[p - left];
        operations += 2;
    }
}
/*Объявление функции merge_sort для рекурсивного сортировки массива методом слияния*/
void merge_sort(vector<int>& arr, int left, int right, long& operations)
{
    /*Условие проверки, что левая граница массива меньше правой*/
    if (left < right)
    {
        /*Вычисление среднего элемента для разделения массива на две части*/
        int mid = left + (right - left) / 2;
        /*Рекурсивный вызов merge_sort для левой и правой частей массива и вызов функции merge для слияния*/
        merge_sort(arr, left, mid, operations);
        merge_sort(arr, mid + 1, right, operations);
        merge(arr, left, mid, right, operations);
        operations++;
    }
    operations++;
}
```

Рисунок 8 – Программа алгоритма простого слияния



```
int main()
{
    setlocale(LC_ALL, "RUS");
    long operations = 0;
    int n, a;
    cout << "Введите n: ";
    cin >> n;
    vector<int> arr(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);//Для заданного диапазона
        cout << "Массив:" << endl;
        for (int i = 0; i < n; i++)
        {
            arr[i] = dist(gen);
            cout << arr[i] << " ";
        }
        cout << endl;
    }
    else if (a == 2)
    {
        cout << "Введите элементы массива:" << endl;
        for (int i = 0; i < n; i++)
        {
            cin >> arr[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*/
    merge_sort(arr, 0, arr.size() - 1, 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 << "Сортированный массив:\n";
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "Было совершено операций сравнения и присваивания: " << operations << endl;
    cout << "Сортировка заняла " << duration << " микросекунд" << endl;
    return 0;
}
```

Рисунок 9 – Функция main для алгоритма простого слияния

2.5.2 Тестирование

Протестируем программу с заданным размером массива n=10 (рис.10), 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.

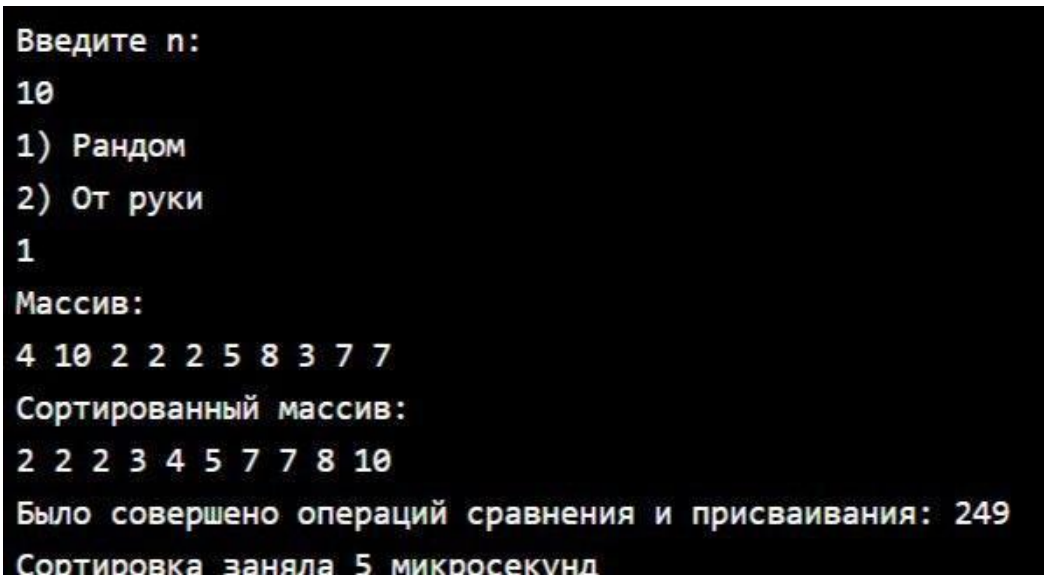


Рисунок 10 - Тестирование программы при n=10

Таблица 1.2. Сводная таблица результатов

n	**T(n), мс**	**$T_п(n)=C_ф+M_ф$**
100	0.045	4225
1000	0.494	58421
10000	3.893	751215
100000	61.41	9160076
1000000	480.337	107773401

2.5.3 Построение графика

По таблице 1.2, построим график функции роста $T_п$ алгоритма сортировки простым слиянием в среднем случае от размера массива n (рис.11).

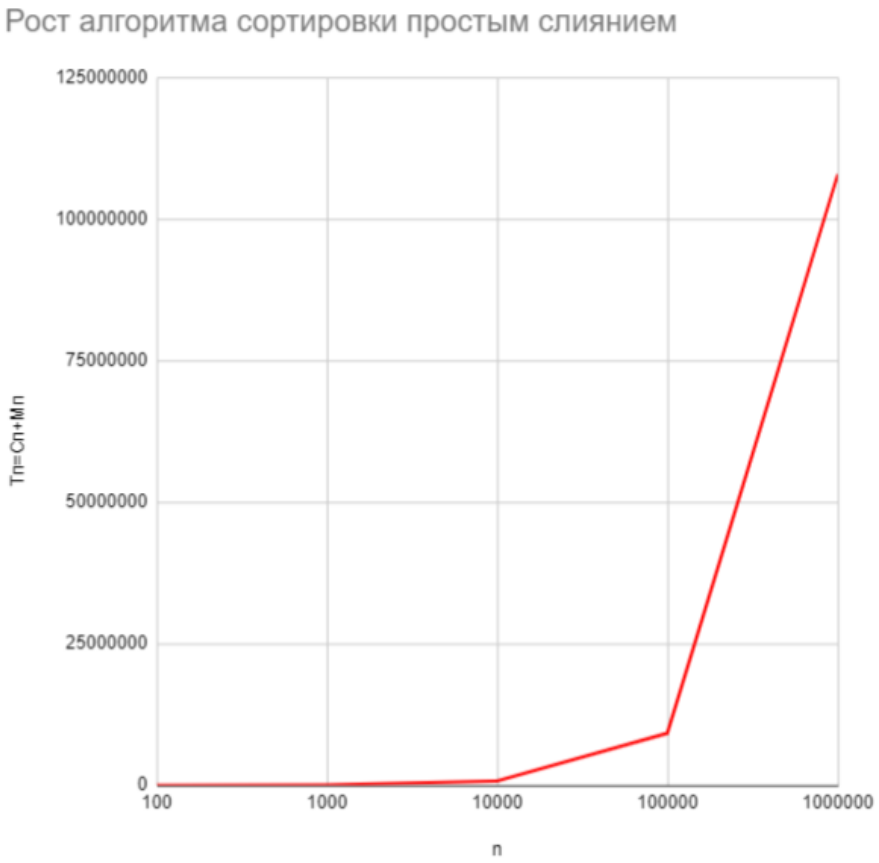


Рисунок 11 - График функции роста $T_п$ алгоритма быстрой сортировки от размера массива n

2.6 Сортировка простыми вставками

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**	**$T_п(n)=C_ф+M_ф$**
100	0.03	6382
1000	1.86	694861
10000	197.22	67770571
100000	14275.56	6752008879
1000000	1033321.23	668448879021

2.7 Сравнение трёх алгоритмов на графике

Построим график функции роста $T_п$ алгоритма сортировки простого слияния, Шелла со смещением Д.Кнута вторым способом и простыми вставками в среднем случае от размера массива n, на данных результата тестирования, продемонстрированных в таблицах 1.1, 1.2 и 1.3. Построим два графика. Первый будет построен на значениях до 1000(рис.12), а второй от 10000 и до 1000000(рис.13). Это позволит нам сделать более точное сравнение.

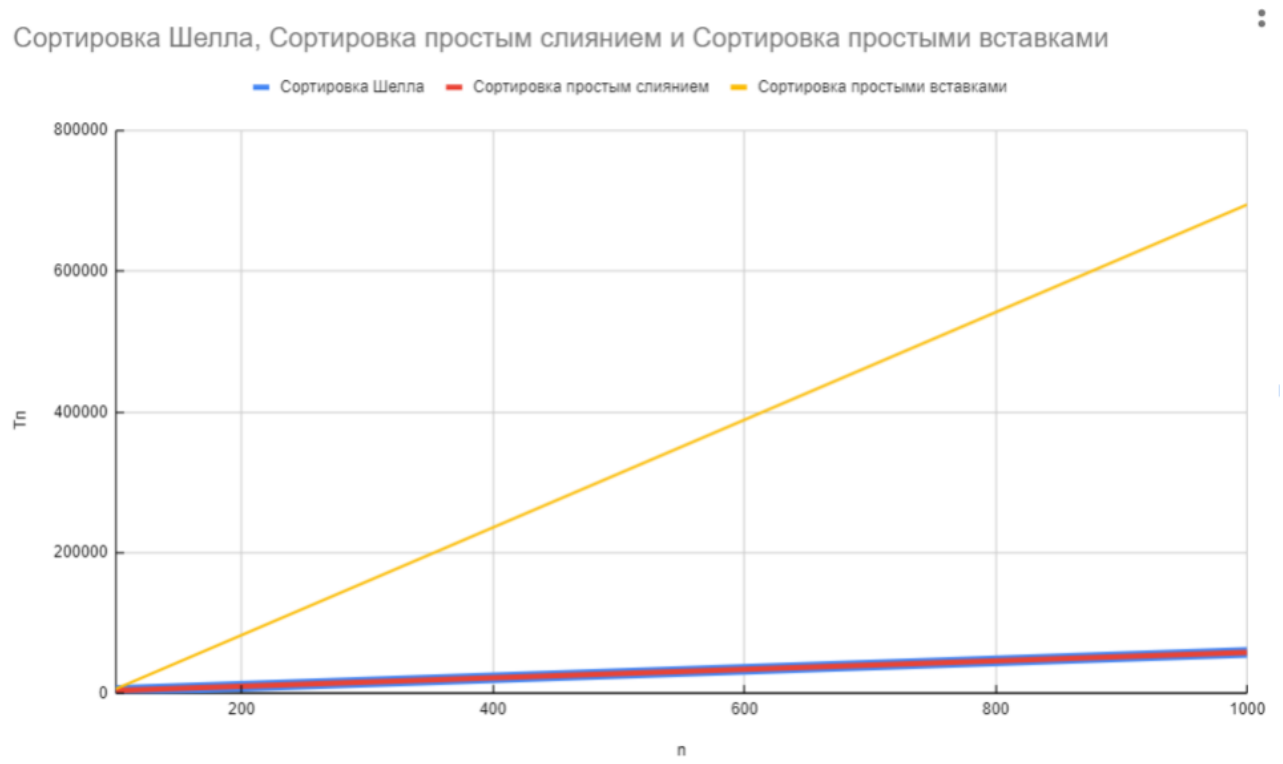


Рисунок 12 - График сравнения трёх сортировок в среднем случае при n до 1000

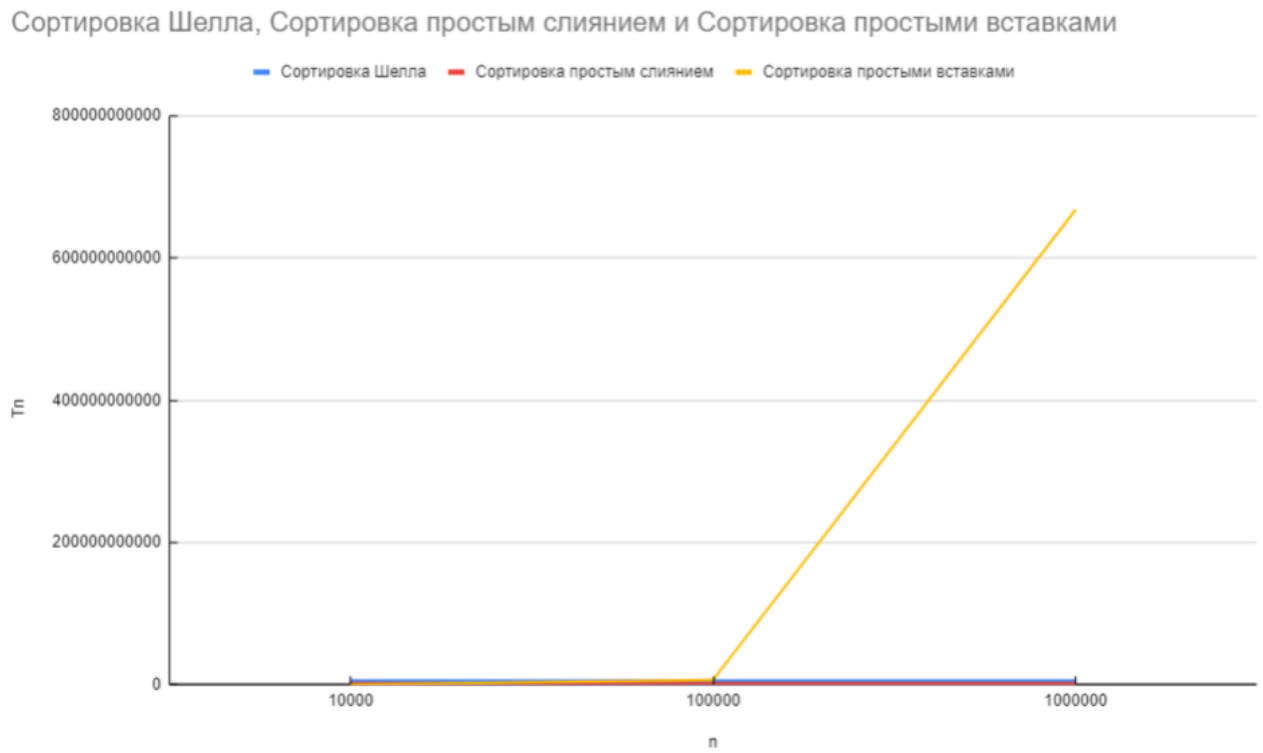


Рисунок 13 - График сравнения трёх сортировок в среднем случае при n от 10000 до 1000000

Анализируя графики, можно сделать вывод, что в среднем случае алгоритм сортировки простой вставкой самый неэффективный, алгоритм простого слияния второй по эффективности, а алгоритм сортировки Шелла со смещением Д.Кнута вторым способом самый эффективный.

2.8 Тестирование программы алгоритма сортировки Шелла со смещением Д.Кнута вторым способом в лучшем и худшем случае

2.8.1 Тестирование программы алгоритма сортировки Шелла со смещением Д.Кнута вторым способом в худшем случае

Протестируем программу с заданным размером массива n=10 (рис.16), n=100, n=1000, n=10000, n=100000, n=1000000 и отсортированными значениями по убыванию. Продемонстрируем результаты тестирования от n=100 до n=1000000 в таблице 1.4. Воспользуемся библиотекой Chrono для подсчёта затраченного времени на сортировку. Воспользуемся библиотекой Algorithm для сортировки по убыванию. Алгоритм сортировки Шелла со смещением Д.Кнута

вторым способом не изменяется и соответствует продемонстрированному на рисунке 3.

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

Рисунок 14 – Используемые библиотеки

```
int main() {
    setlocale(LC_ALL, "RUS");
    long operations = 0;
    int n, a;
    cout << "Введите n: ";
    cin >> n;
    vector<int> arr(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); //Для заданного диапазона
        cout << "Массив:" << endl;
        for (int i = 0; i < n; i++)
        {
            arr[i] = dist(gen);
            cout << arr[i] << " ";
        }
        cout << endl;
    }
    else if (a == 2)
    {
        cout << "Введите элементы массива:" << endl;
        for (int i = 0; i < n; i++)
        {
            cin >> arr[i];
        }
        cout << endl;
    }
    else
    {
        cout << "Ошибка";
        return 0;
    }
    sort(arr.begin(), arr.end(), greater<int>());
    /*Получение текущего времени для измерения начала времени выполнения сортировки*/
    auto start = chrono::high_resolution_clock::now();
    /*Вызов функции для сортировки массива*/
    shell_sort(arr, 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 << "Сортированный массив:\n";
    for (int i = 0; i < n; ++i)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "Было совершено операций сравнения и присваивания: " << operations << endl;
    cout << "Сортировка заняла " << duration << " микросекунд" << endl;
    return 0;
}
```

Рисунок 15 – Функция main с сортировкой по убыванию

```
Введите n:
10
1) Рандом
2) От руки
1
Массив:
1 10 6 9 7 5 4 9 2 2
Сортированный массив:
1 2 2 4 5 6 7 9 9 10
Было совершено операций сравнения и присваивания: 126
Сортировка заняла 1 микросекунд
```

Рисунок 16 – Результаты тестирования программы при n=10 и с отсортированными значениями по убыванию

2.8.2 Тестирование программы алгоритма сортировки Шелла со смещением Д.Кнута вторым способом в лучшем случае

Протестируем программу с заданным размером массива n=10 (рис.19), n=100, n=1000, n=10000, n=100000, n=1000000 и отсортированными значениями по возрастанию. Продемонстрируем результаты тестирования от n=100 до n=1000000 в таблице 1.4. Воспользуемся библиотекой Chrono для подсчёта затраченного времени на сортировку. Воспользуемся библиотекой Algorithm для сортировки по возрастанию. Алгоритм сортировки Шелла со смещением Д.Кнута вторым способом не изменяется и соответствует продемонстрированному на рисунке 3.

```
1  #include <iostream>
2  #include <vector>
3  #include <random>
4  #include <chrono>
5  #include <algorithm>
```

Рисунок 17 – Используемые библиотеки

```cpp
int main() {
    setlocale(LC_ALL, "RUS");
    long operations = 0;
    int n, a;
    cout << "Введите n: ";
    cin >> n;
    vector<int> arr(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); //Для заданного диапазона
        cout << "Массив:" << endl;
        for (int i = 0; i < n; i++)
        {
            arr[i] = dist(gen);
            cout << arr[i] << " ";
        }
        cout << endl;
    }
    else if (a == 2)
    {
        cout << "Введите элементы массива:" << endl;
        for (int i = 0; i < n; i++)
        {
            cin >> arr[i];
        }
        cout << endl;
    }
    else
    {
        cout << "Ошибка";
        return 0;
    }
    sort(arr.begin(), arr.end());
    /*Получение текущего времени для измерения начала времени выполнения сортировки*/
    auto start = chrono::high_resolution_clock::now();
    /*Вызов функции для сортировки массива*/
    shell_sort(arr, 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 << "Сортированный массив:\n";
    for (int i = 0; i < n; ++i)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "Было совершено операций сравнения и присваивания: " << operations << endl;
    cout << "Сортировка заняла " << duration << " микросекунд" << endl;
    return 0;
}
```

Рисунок 18 – Функция main с сортировкой по возрастанию

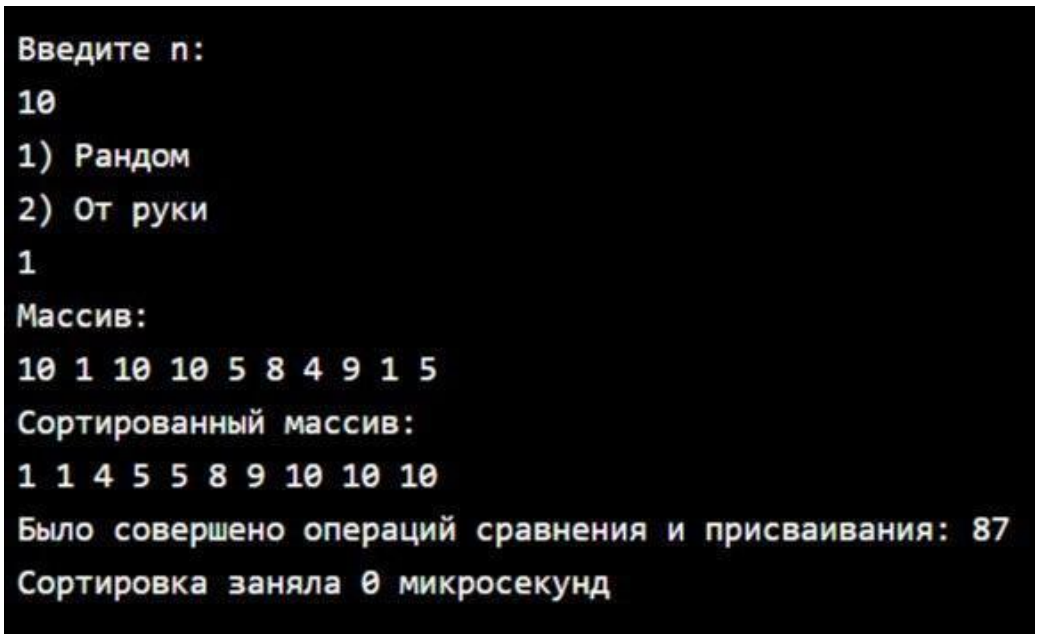


Рисунок 19 – Результаты тестирования программы при n=10 и с отсортированными значениями по возрастанию

2.8.2 Заполнение таблицы для алгоритма сортировки Шелла со смещением Д.Кнута вторым способом в лучшем и худшем случае

Заполним таблицу 1.4 данными, полученными в результате тестирования программ.

Таблица 1.4. Сводная таблица результатов

n	T(n), мс	$T_п(n)=C_ф+M_ф$ в худшем случае	T(n), мс	$T_п(n)=C_ф+M_ф$ в лучшем случае
100	0.006	2122	0.005	1732
1000	0.111	32618	0.055	27317
10000	1.085	446513	0.78	376262
100000	7.849	5175219	9.844	4835787
1000000	394.724	62940478	88.35	59021392

2.8.3 Построение графика для алгоритма сортировки Шелла со смещением Д.Кнута вторым способом в лучшем и худшем случае

По таблице 1.4, построим график функции роста $T_п$ алгоритма сортировки Шелла со смещением Д.Кнута вторым способом от размера массива n в худшем(рис.20) и лучшем(рис.21) случае.

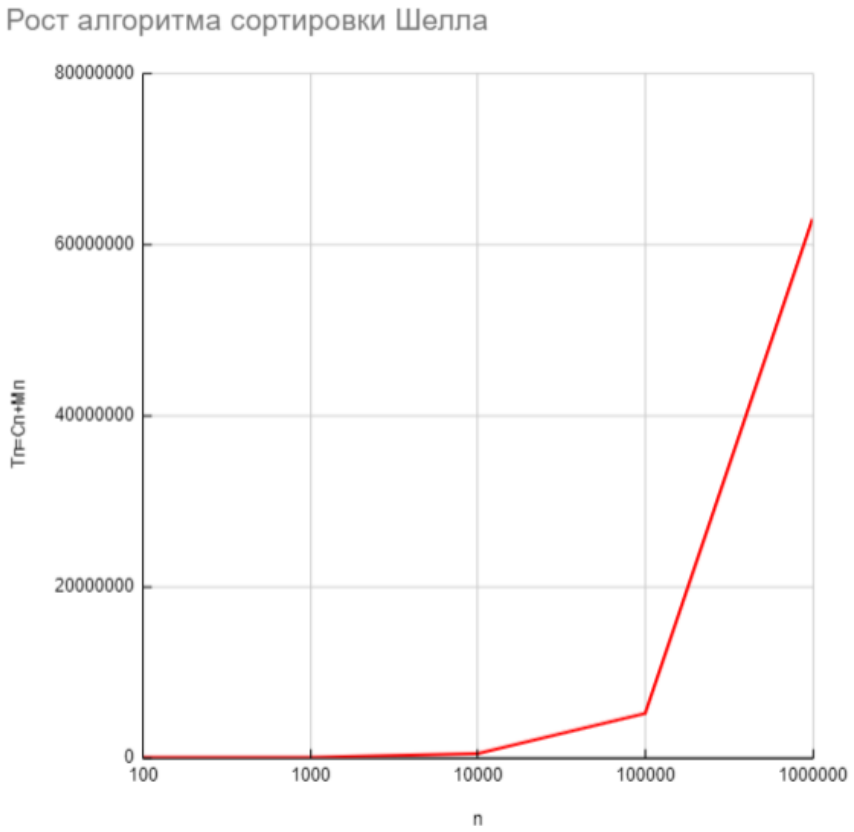


Рисунок 20 - График функции роста $T_п$ алгоритма сортировки Шелла со смещением Д.Кнута вторым способом от размера массива n в худшем случае

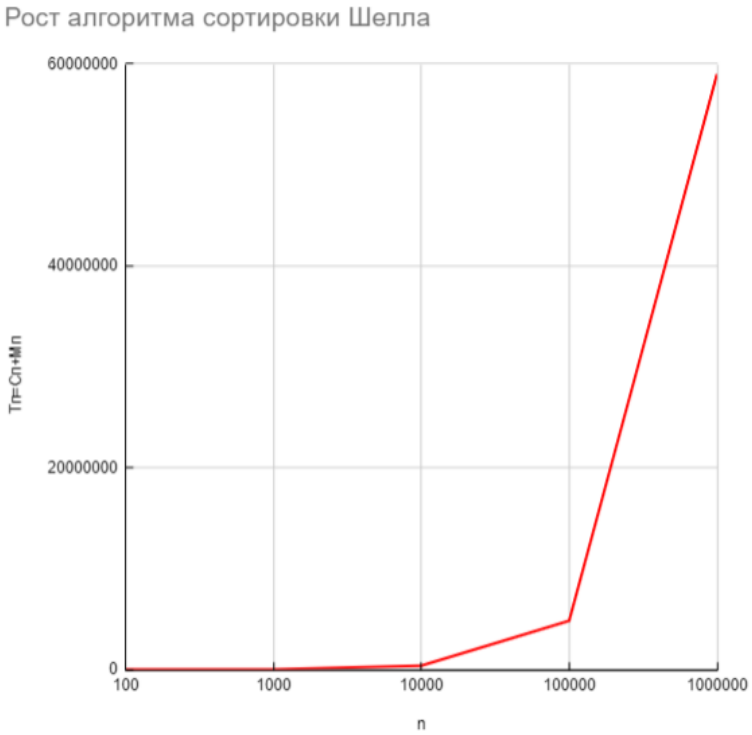


Рисунок 21 - График функции роста $T_п$ алгоритма сортировки Шелла со смещением Д.Кнута вторым способом от размера массива n в лучшем случае

2.9 Тестирование программы алгоритма сортировки простым слиянием в лучшем и худшем случае

2.9.1 Тестирование программы алгоритма сортировки простым слиянием в худшем случае

Протестируем программу с заданным размером массива n=10 (рис.24), n=100, n=1000, n=10000, n=100000, n=1000000 и отсортированными значениями по убыванию. Продемонстрируем результаты тестирования от n=100 до n=1000000 в таблице 1.5. Воспользуемся библиотекой Chrono для подсчёта затраченного времени на сортировку. Воспользуемся библиотекой Algorithm для сортировки по убыванию. Алгоритм простого слияния не изменяется и соответствует продемонстрированному на рисунке 8.

```
1  #include <iostream>
2  #include <vector>
3  #include <random>
4  #include <chrono>
5  #include <algorithm>
```

Рисунок 22 – Используемые библиотеки

```cpp
int main()
{
    setlocale(LC_ALL, "RUS");
    long operations = 0;
    int n, a;
    cout << "Введите n: ";
    cin >> n;
    vector<int> arr(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);//Для заданного диапазона
        cout << "Массив:" << endl;
        for (int i = 0; i < n; i++)
        {
            arr[i] = dist(gen);
            cout << arr[i] << " ";
        }
        cout << endl;
    }
    else if (a == 2)
    {
        cout << "Введите элементы массива:" << endl;
        for (int i = 0; i < n; i++)
        {
            cin >> arr[i];
        }
        cout << endl;
    }
    else
    {
        cout << "Ошибка";
        return 0;
    }
    sort(arr.begin(), arr.end(), greater<int>());
    /*Получение текущего времени для измерения начала времени выполнения сортировки*/
    auto start = chrono::high_resolution_clock::now();
    /*Вызов функции для сортировки массива*/
    merge_sort(arr, 0, arr.size() - 1, 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 << "Сортированный массив:\n";
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "Было совершено операций сравнения и присваивания: " << operations << endl;
    cout << "Сортировка заняла " << duration << " микросекунд" << endl;
    return 0;
}
```

Рисунок 23 – Функция main с сортировкой по убыванию

```
Введите n:
10
1) Рандом
2) От руки
1
Массив:
10 7 2 6 5 8 1 5 8 8
Сортированный массив:
1 2 5 5 6 7 8 8 8 10
Было совершено операций сравнения и присваивания: 246
Сортировка заняла 5 микросекунд
```

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

2.9.2 Тестирование программы алгоритма сортировки простым слиянием в лучшем случае

Протестируем программу с заданным размером массива n=10 (рис.27), n=100, n=1000, n=10000, n=100000, n=1000000 и отсортированными значениями по возрастанию. Продемонстрируем результаты тестирования от n=100 до n=1000000 в таблице 1.5. Воспользуемся библиотекой Chrono для подсчёта затраченного времени на сортировку. Воспользуемся библиотекой Algorithm для сортировки по возрастанию. Алгоритм простого слияния не изменяется и соответствует продемонстрированному на рисунке 8.

```
1  #include <iostream>
2  #include <vector>
3  #include <random>
4  #include <chrono>
5  #include <algorithm>
```

Рисунок 25 – Используемые библиотеки

```cpp
int main()
{
    setlocale(LC_ALL, "RUS");
    long operations = 0;
    int n, a;
    cout << "Введите n: ";
    cin >> n;
    vector<int> arr(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);//Для заданного диапазона
        cout << "Массив:" << endl;
        for (int i = 0; i < n; i++)
        {
            arr[i] = dist(gen);
            cout << arr[i] << " ";
        }
        cout << endl;
    }
    else if (a == 2)
    {
        cout << "Введите элементы массива:" << endl;
        for (int i = 0; i < n; i++)
        {
            cin >> arr[i];
        }
        cout << endl;
    }
    else
    {
        cout << "Ошибка";
        return 0;
    }
    sort(arr.begin(), arr.end());
    /*Получение текущего времени для измерения начала времени выполнения сортировки*/
    auto start = chrono::high_resolution_clock::now();
    /*Вызов функции для сортировки массива*/
    merge_sort(arr, 0, arr.size() - 1, 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 << "Сортированный массив:\n";
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "Было совершено операций сравнения и присваивания: " << operations << endl;
    cout << "Сортировка заняла " << duration << " микросекунд" << endl;
    return 0;
}
```

Рисунок 26 – Функция main с сортировкой по возрастанию

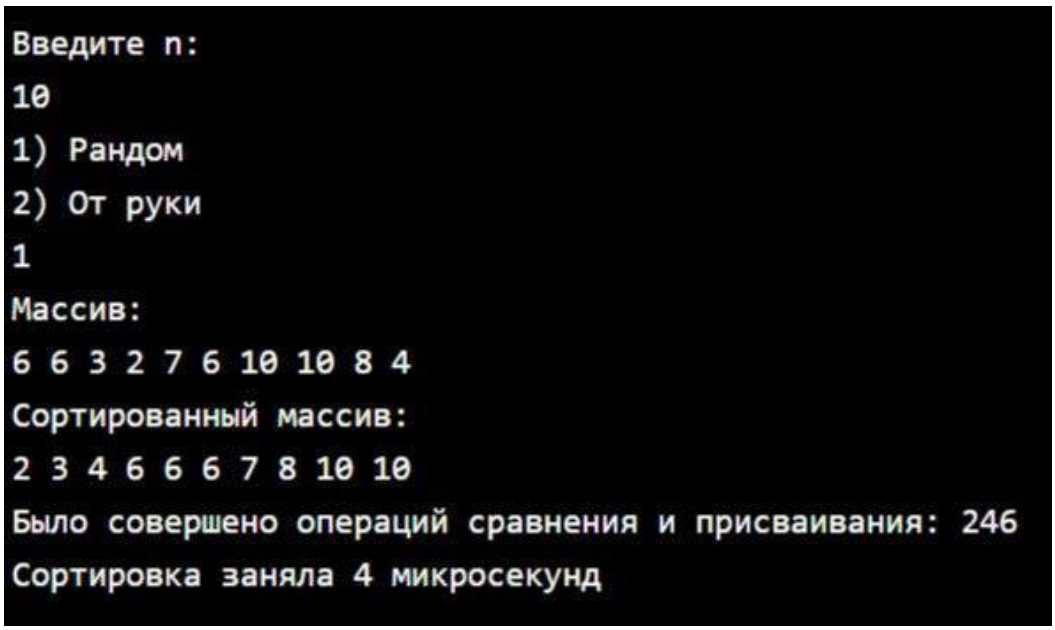


Рисунок 27 – Результаты тестирования программы при n=10 и с отсортированными значениями по возрастанию

2.9.3 Заполнение таблицы для алгоритма сортировки простым слиянием в лучшем и худшем случае

Заполним таблицу 1.5 данными, полученными в результате тестирования программ.

Таблица 1.5. Сводная таблица результатов

n	**T(n), мс**	**$T_п(n)=C_ф+M_ф$ в худшем случае**	**T(n), мс**	**$T_п(n)=C_ф+M_ф$ в лучшем случае**
100	0.047	4096	0.044	4035
1000	0.296	55896	0.295	54939
10000	5.061	712775	4.876	703463
100000	394.407	8629263	55.858	8529607
1000000	600.149	100871855	601.909	99872119

2.9.4 Построение графика для алгоритма сортировки простым слиянием в лучшем и худшем случае

По таблице 1.5, построим график функции роста $T_п$ алгоритма сортировки простым слиянием в худшем(рис.28) и лучшем(рис.29) случае от размера массива n.

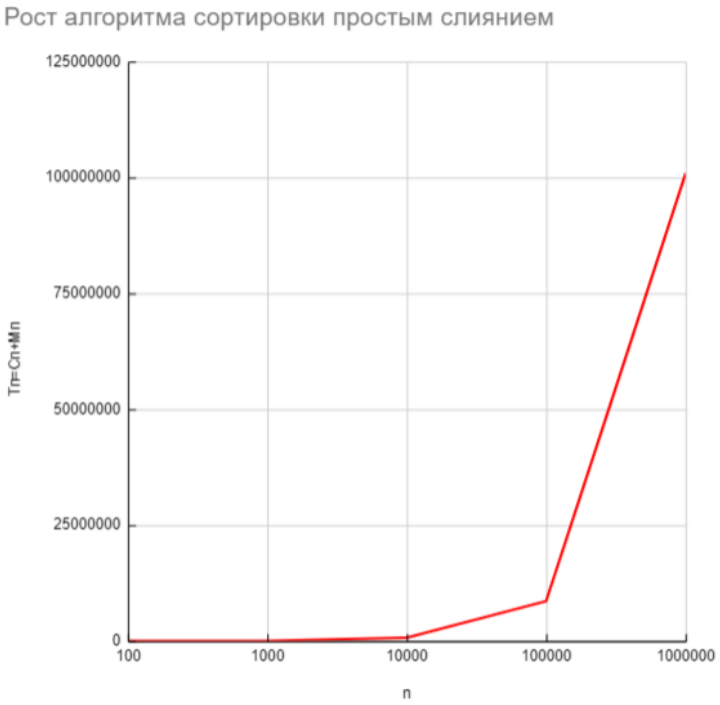


Рисунок 28 - График функции роста $T_п$ алгоритма простого слияния в худшем случае от размера массива n

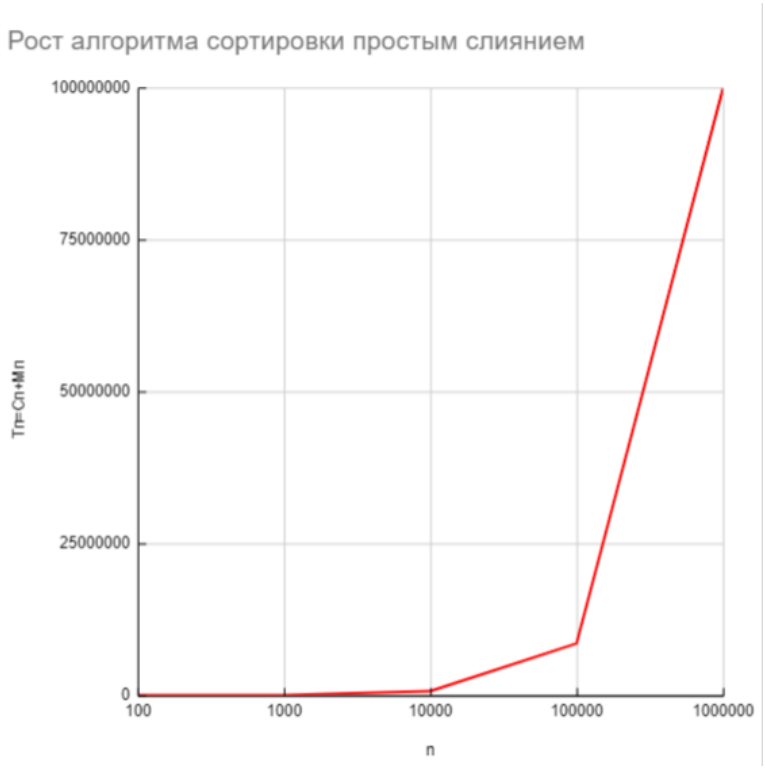


Рисунок 29 - График функции роста $T_п$ алгоритма простого слияния в лучшем случае от размера массива n

2.10 Вывод по заданию №1

Из результатов сравнения алгоритмов сортировки Шелла смещением Д.Кнута и простого слияния можно сделать вывод, что оба алгоритма показывают эффективность и хорошую производительность независимо от исходной упорядоченности массива.

Алгоритм сортировки Шелла смещением Д.Кнута вторым способом показывает хорошие результаты на различных типах данных, включая случайные, упорядоченные и обратно упорядоченные массивы. Он обладает хорошей скоростью сходимости и обычно работает быстрее, чем простое слияние.

С другой стороны, алгоритм простого слияния проявляет стабильную производительность на разных типах данных, но может показать более высокую производительность на частично упорядоченных массивах.

Таким образом, можно сделать вывод, что оба алгоритма показывают хорошие результаты независимо от исходной упорядоченности массива, но

могут показывать различную производительность в зависимости от характеристик данных.

3 ЗАДАНИЕ №2

3.1 Формулировка задачи (Вариант 5, в списке 5)

Асимптотический анализ сложности алгоритмов

Требования по выполнению задания

1. Из материалов предыдущей практической работы приведите в отчёте формулы $T_т(n)$ 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 O-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 шейкерной сортировки и быстрой сортировки (Хоара).

6. Общие результаты свести в табл. 2.

7. Сделать вывод о наиболее эффективном алгоритме из трёх.

3.2 Формулы функции роста алгоритма сортировки простой вставкой в худшем и лучшем случае

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

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

Средний случай - массив отсортирован в обратном порядке. В этом случае количество операций также будет $T_т(n)=n^2/4$

3.3 Асимптотическая оценка вычислительной сложности простого алгоритма сортировки вставкой

Асимптотическая оценка вычислительной сложности простого алгоритма сортировки вставкой для худшего случая равна $O(n^2)$.

Асимптотическая оценка вычислительной сложности простого алгоритма сортировки вставкой для лучшего случая равна $\Omega(n)$.

Асимптотическая оценка вычислительной сложности простого алгоритма сортировки вставкой для среднего случая равна $\theta(n^2)$.

Ёмкостная сложность алгоритма простой сортировки вставкой O(1).

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

Построим график на основе пунктов 3.2 и 3.3(рис.30).

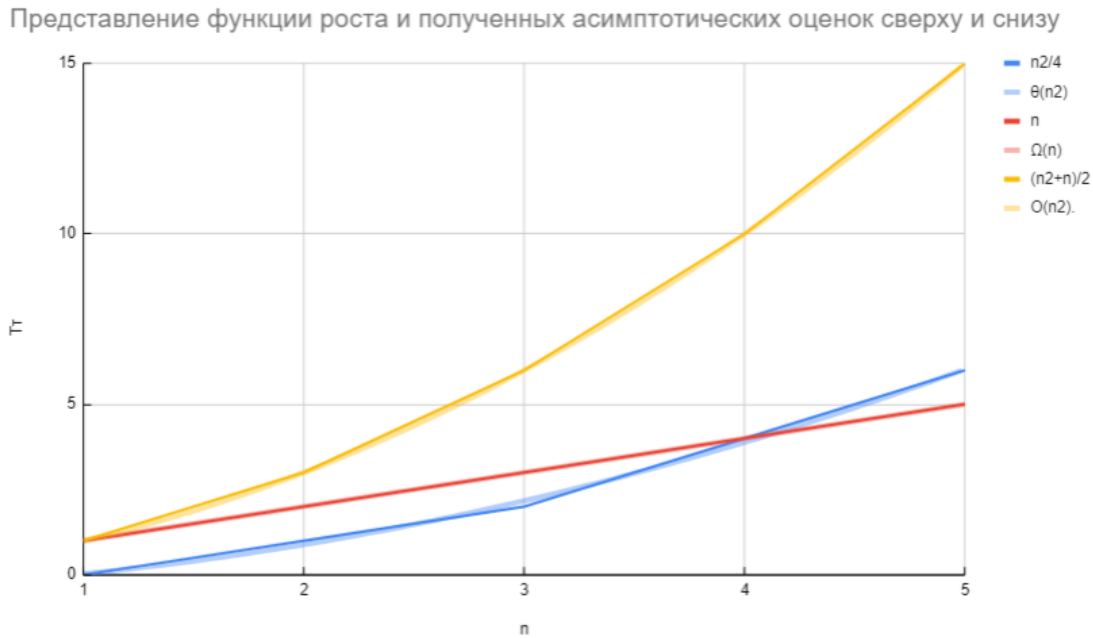


Рисунок 30 - Графическое представление функции роста и полученных асимптотических оценок сверху и снизу

3.5 Справочная информация о вычислительной сложности алгоритмов сортировки Шелла со смещением Д.Кнута вторым способом и простым слиянием

Асимптотическая оценка вычислительной сложности алгоритма сортировки Шелла со смещением Д.Кнута вторым способом для худшего случая равна $O(n\log^2_2 n)$

Асимптотическая оценка вычислительной сложности алгоритма сортировки Шелла со смещением Д.Кнута вторым способом для лучшего случая равна $\Omega(n)$.

Асимптотическая оценка вычислительной сложности алгоритма сортировки Шелла со смещением Д.Кнута вторым способом для среднего случая равна $\theta(n^{1.25})$.

Ёмкостная сложность алгоритма сортировки Шелла со смещением Д.Кнута вторым способом равна $O(1)$.

Асимптотическая оценка вычислительной сложности алгоритма простого слияния для худшего случая равна $O(n\log_2 n)$.

Асимптотическая оценка вычислительной сложности алгоритма простого слияния для лучшего случая равна $\Omega(n\log_2 n)$.

Асимптотическая оценка вычислительной сложности простого слияния для среднего случая равна $\theta(n\log_2 n)$.

Ёмкостная сложность алгоритма простого слияния равна $O(n)$.

3.6 Таблица асимптотической сложности трёх алгоритмов

Заполним таблицу 2 асимптотической сложности алгоритма для алгоритмов сортировки простой вставки, Шелла со смещением Д.Кнута вторым способом 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
Простая вставка	$O(n^2)$	$\Omega(n)$	$\theta(n^2)$	$O(1)$
Сортировка Шелла со смещением Д.Кнута вторым способом	$O(n\log^2_2 n)$	$\Omega(n)$	$\theta(n^{1.25})$	$O(1)$
Простым слиянием	$O(n\log_2 n)$	$\Omega(n\log_2 n)$	$\theta(n\log_2 n)$	$O(n)$

3.7 Выводы по заданию №2

Сортировка Шелла с смещением Д.Кнута вторым способом наиболее эффективна. Она обладает более высокой скоростью сортировки и эффективностью в сравнении с простым слиянием и простой вставкой. Сортировка Шелла в общем случае имеет асимптотическую сложность $O(n \log^2_2 n)$, что делает её эффективным алгоритмом на практике, особенно для больших массивов данных.

Простое слияние, хотя и более простой в реализации, требует больше операций сравнения и перемещения элементов, что приводит к более длительному времени выполнения в сравнении с сортировкой Шелла. Простое слияние имеет асимптотическую сложность $O(n\log_2 n)$, что тоже является эффективным результатом. Однако, сортировка Шелла обычно работает быстрее на практике, особенно при сортировке больших массивов.

Простая вставка также имеет хорошую эффективность, но она не так эффективна на больших массивах данных, как сортировка Шелла. Простая вставка имеет асимптотическую сложность $O(n^2)$, что делает её менее эффективной по сравнению с двумя другими алгоритмами.

Таким образом, можно сделать вывод, что сортировка Шелла с смещением Д.Кнута вторым способом является наиболее эффективным алгоритмом среди перечисленных для сортировки массивов данных.

5 ВОПРОСЫ

1) В чём заключается улучшение сортировки?

Улучшение сортировки заключается в улучшении её производительности, эффективности и скорости работы. Это может быть достигнуто путем оптимизации алгоритма сортировки, выбора наиболее подходящего алгоритма для конкретной задачи, уменьшения количества операций сравнения и перемещения элементов, а также параллельной или распределенной сортировки данных. Улучшение сортировки также может включать в себя использование специализированных структур данных, оптимизацию памяти и учет особенностей данных для ускорения процесса сортировки.

2) В чём отличия шейкерной сортировки от простой сортировки обменом?

Шейкерная сортировка (также известная как сортировка перемешиванием или сортировка коктейльной) является модификацией простой сортировки обменом. Основные отличия между шейкерной и простой сортировкой обменом заключаются в направлении прохода и условиях завершения сортировки.

1. Направление прохода:

- Простая сортировка обменом всегда проходит данные слева направо, сравнивая и обменивая соседние элементы до тех пор, пока массив не будет отсортирован.

- Шейкерная сортировка движется в обоих направлениях: сначала слева направо (как при простой сортировке), затем обратно справа налево. Она меняет направление прохода после каждого прохода данных.

2. Условия завершения:

- В простой сортировке обменом сортировка завершается, когда не было совершено ни одной перестановки элементов за проход даннных, что означает, что массив отсортирован.

- В шейкерной сортировке сортировка завершается, когда в течение прохода не было совершено ни одной перестановки элементов как слева направо, так и справа налево.

Таким образом, шейкерная сортировка, дополнительно к изменению направления прохода, может ускорить сортировку в некоторых случаях за счет движения элементов в обоих направлениях. Однако в общем случае эффективность шейкерной сортировки и простой сортировки обменом примерно одинакова, но шейкерная сортировка может быть более эффективной в некоторых сценариях.

3) Насколько повышается эффективность в случае шейкерной сортировки?

Эффективность шейкерной сортировки по сравнению с простой сортировкой обменом может быть незначительно выше в определенных сценариях, но общее увеличение скорости выполнения не всегда гарантировано.

В целом, необходимо учитывать множество факторов, таких как размеры сортируемых данных, начальное состояние массива, структура данных и многое другое, чтобы оценить и сравнить эффективность шейкерной и простой сортировки обменом. Кроме того, для оптимальной эффективности сортировки рекомендуется также рассмотреть другие более эффективные алгоритмы сортировки, такие как квиксорт или сортировка слиянием.

4) Как работает сортировка Шелла?

Сортировка Шелла (или сортировка методом уменьшения интервала) является усовершенствованием сортировки вставками. Основное отличие

заключается в том, что сортировка Шелла сначала упорядочивает данные с небольшими интервалами, а затем постепенно сокращает интервалы до одного (что эквивалентно обычной сортировке вставками).

Основные шаги работы сортировки Шелла:

1. Выбор интервала: сначала выбирается последовательность значений интервалов, которая будет использоваться для сортировки. Эта последовательность значений обычно выбирается так, чтобы начальные интервалы были большими, а затем они уменьшались по мере продвижения к конечному интервалу 1.

2. Сортировка с интервалами: Данные сортируются, используя сортировку вставками, но для каждого шага сравнения элементов используется выбранный интервал вместо шага в одну позицию.

3. Уменьшение интервала: после сортировки данных с текущим интервалом (данные некорректно упорядочены), интервал уменьшается. Алгоритм продолжает проходить по данным и сортировать их с новым уменьшенным интервалом.

4. Последняя фаза с интервалом 1: наконец, алгоритм доводит интервал до 1 и завершает сортировку, применяя обычную сортировку вставками.

Сортировка Шелла является эффективной для сортировки больших массивов данных. Она позволяет ускорить сортировку за счет предварительной сортировки с более крупными интервалами, что способствует более эффективному перемещению элементов к их окончательным позициям. Однако эффективность сортировки Шелла сильно зависит от выбора последовательности интервалов.

5) Почему затраты времени на частичную сортировку оправдывают себя?

Затраты времени на частичную сортировку могут оправдать себя в определенных ситуациях, когда необходимо быстро получить отсортированные данные или когда есть возможность сократить количество операций обработки данных. Ниже приведены некоторые причины, почему частичная сортировка может оправдать затраты времени: улучшение производительности, уменьшение времени на поиск, построение приоритетной очереди, повышение эффективности циклического доступа.

6) От чего зависит эффективность сортировки Шелла в среднем случае?

Эффективность сортировки Шелла в среднем случае зависит от выбора последовательности интервалов, которая определяет, какие элементы будут сравниваться и перемещаться во время сортировки.

7) Опишите вариант использования смещения Седжвика в сортировке Шелла.

Смещение Седжвика — это один из вариантов последовательности интервалов, которая используется в сортировке Шелла. Этот вариант последовательности интервалов был предложен Дональдом Седжвиком и обладает хорошей производительностью для сортировки Шелла.

Смещение Седжвика определяется следующим образом:

1. Начальное значение интервала вычисляется как 1, 4, 10, 23, 57, 132, 301, 701, и т.д., используя формулу:

h = 9 2^k - 9 2^(k/2) + 1, где k - порядковое число интервала.

2. Значения интервалов уменьшаются в соответствии с формулой:

h = (h - 1) / 3

3. Когда значение интервала становится равным 1, сортировка Шелла завершается с использованием обычной сортировки вставками.

Использование смещения Седжвика в сортировке Шелла обеспечивает хорошее улучшение производительности за счет хорошо подобранной последовательности интервалов. Этот способ позволяет ускорить процесс сортировки за счет выбора оптимальных значений интервалов, что позволяет эффективно перемещать элементы в массиве и сокращать интервалы до единицы, обеспечивая оптимальную сортировку.

8) Опишите вариант использования смещений Кнута в сортировке Шелла.

Смещения Кнута — это еще один вариант последовательности интервалов, который можно использовать в сортировке Шелла. Этот вариант был предложен известным ученым Дональдом Кнутом и также обладает хорошей производительностью при сортировке с использованием метода Шелла.

Смещения Кнута определяются следующим образом:

1. Начальное значение интервала вычисляется как 1, 4, 13, 40, 121, 364 и т.д., используя формулу:

h = 3h + 1, где h - предыдущее значение интервала.

2. Значения интервалов уменьшаются в соответствии с формулой:

h = (h - 1) / 3

3. Когда значение интервала становится равным 1, сортировка Шелла завершается с использованием обычной сортировки вставками.

Использование смещений Кнута в сортировке Шелла также обеспечивает эффективное улучшение производительности за счет оптимально подобранной последовательности интервалов. Выбор оптимальных значений интервалов позволяет сортировке эффективно перемещать элементы в массиве и после сокращения интервалов до единицы выполнить оптимальную сортировку.

Обе последовательности интервалов, предложенные Седжвиком и Кнутом, являются популярными и широко используются для сортировки Шелла из-за их хорошей производительности. Каждый из вариантов имеет свои особенности, которые могут подходить для разных типов данных и размеров массивов.

9) Назовите автора алгоритма сортировки простым слиянием. Каким был исторический контекст этой разработки?

Автором алгоритма сортировки простым слиянием является Джон фон Нейман (John von Neumann), выдающийся математик, физик и инженер, который сделал огромный вклад в различные области науки, включая компьютерные науки.

Исторический контекст разработки алгоритма сортировки простым слиянием связан с ранними шагами в развитии компьютерных наук и информатики. В период после Второй мировой войны были созданы первые компьютеры, и появилась необходимость разработки эффективных алгоритмов сортировки для обработки и анализа данных.

10) Объясните идею алгоритма сортировки простым слиянием.

Идея алгоритма сортировки простым слиянием состоит в разделении массива на две части путем рекурсивного деления до тех пор, пока каждая часть

не будет содержать только один элемент (базовый случай). Затем происходит слияние этих частей в отсортированный массив.

Шаги алгоритма сортировки простым слиянием можно описать следующим образом:

1. Разделение массива: Входной массив делится на две части примерно одинакового размера (или близкого к этому), путем нахождения середины массива. Этот процесс повторяется для каждой из двух частей массива до тех пор, пока каждая часть не будет содержать только один элемент.

2. Слияние: Отсортированные подмассивы сливаются в один отсортированный массив. При слиянии элементы из двух отсортированных подмассивов сравниваются, и наименьший элемент помещается в результирующий массив. Этот процесс продолжается до тех пор, пока все элементы не будут помещены в результирующий массив.

3. Возврат: Полученный отсортированный массив возвращается в качестве результата работы алгоритма.

11) Как оценивается асимптотическая сложность алгоритма простого слияния?

Асимптотическая сложность алгоритма сортировки простым слиянием оценивается как O(n*log(n)), где n - количество элементов в массиве.

12) В чём отличие алгоритма естественного слияния от простого?

Алгоритм естественного слияния (Natural Merge Sort) и алгоритм простого слияния (Merge Sort) являются оба методами сортировки, основанными на слиянии отсортированных подмассивов. Вот основные отличия между ними:

1. Структура входного массива:

- Простой алгоритм слияния требует, чтобы массив был разделен на две равные части в начале процесса сортировки. Затем эти две части сортируются отдельно и объединяются.

- В алгоритме естественного слияния массив не делится на две равные части в начале. Вместо этого алгоритм сначала находит участки отсортированных элементов в массиве (возрастающие подмассивы), затем эти участки объединяются, что уменьшает количество сравнений и оптимизирует производительность в том случае, если части массива уже отсортированы.

2. Эффективность:

- Алгоритм естественного слияния более эффективен и быстрее в худшем случае, когда массив состоит из отсортированных блоков, так как не требуется деление массива на две равные части и последующая сортировка каждой части отдельно.

- Алгоритм простого слияния обычно выполняется чуть медленнее в худшем случае, но он более общий и может быть эффективно применен к любому массиву.

3. Сложность:

- Сложность алгоритма естественного слияния в худшем случае составляет O(n*log(n)), как и у простого алгоритма слияния.

- Однако в среднем случае алгоритм естественного слияния может быть более быстрым за счет оптимизации процесса слияния отсортированных блоков.

Каждый из этих алгоритмов имеет свои преимущества и недостатки, и выбор между ними зависит от особенностей входных данных и требуемой производительности.

13) Какой вариант сортировки слиянием более эффективен для работы с внешними данными?

Для работы с внешними данными, то есть когда данные не помещаются полностью в оперативную память, более эффективным вариантом сортировки слиянием является алгоритм естественного слияния (Natural Merge Sort). Этот алгоритм специально разработан для работы с внешними данными и имеет ряд преимуществ:

1. Минимизация операций чтения и записи данных:

Алгоритм естественного слияния позволяет объединять уже отсортированные подмассивы без предварительной сортировки, что позволяет избежать множества операций чтения и записи на диск.

2. Эффективное слияние блоков данных:

Алгоритм естественного слияния эффективен при слиянии блоков данных, которые были разделены на участки отсортированных элементов. Это позволяет минимизировать количество операций ввода-вывода и улучшить производительность при работе с внешними данными.

3. Меньшее использование дополнительной памяти:

При работе с внешними данными важно учитывать ограниченность оперативной памяти. Алгоритм естественного слияния требует минимального использования дополнительной памяти, так как данные считываются и записываются напрямую с диска без необходимости хранения больших объемов данных в оперативной памяти.

4. Устойчивость к отсутствию дополнительной памяти:

В случае, если оперативная память недоступна или её объем существенно ограничен, алгоритм естественного слияния может эффективнее обрабатывать внешние данные, так как он не требует хранения всего массива данных в памяти для сортировки.

Таким образом, для работы с внешними данными более предпочтительным вариантом сортировки слиянием является алгоритм естественного слияния, который оптимизирован для эффективной работы с ограниченными ресурсами оперативной памяти и внешними носителями данных.

14) Как работает быстрая сортировка?

Этот метод сортировки, называемый быстрая сортировка, представляет собой улучшенную версию сортировки пузырьком. Основные шаги алгоритма:

1. Выбор опорного элемента.

2. Разделение массива: все элементы, меньшие опорного, перемещаются перед ним, а все элементы, большие или равные опорному, перемещаются после него.

3. Рекурсивное применение первых двух шагов к двум подмассивам слева и справа от опорного элемента (если в подмассиве больше одного элемента).

4. Комбинирование не требуется, так как подмассивы сортируются на месте. В результате весь массив оказывается отсортированным.

15) Как оценивается асимптотическая сложность алгоритма быстрой сортировки? От чего она зависит?

Сложность алгоритма быстрой сортировки оценивается как $O(n\log_2 n)$, где n - размер входных данных. Эта оценка зависит от логарифма размера входных данных и самого размера данных.

16) Какие есть варианты выбора опорного элемента?

Есть несколько способов выбора опорного элемента в алгоритме быстрой сортировки. Один из них - использовать средний элемент. Другой вариант - выбрать минимальный элемент. Можно также выбрать опорным случайный элемент.

17) В чём отличие схемы Хоара в алгоритме быстрой сортировки?

Схема Хоара — это метод выбора опорного элемента в алгоритме быстрой сортировки. Он заключается в том, что выбирается элемент с максимальным значением, затем он меняется местами с элементом, находящимся в середине

массива. Это позволяет повысить эффективность алгоритма, поскольку уменьшается количество сравнений и перестановок элементов.

18) Что такое скорость роста функции?

Скорость роста функции описывает, насколько быстро алгоритм выполняется при увеличении размера входных данных. Например, если функция имеет асимптотическую сложность $O(n^2)$, это указывает на то, что алгоритм будет работать в квадратичной зависимости от размера входных данных.

19) Что такое асимптотический анализ сложности алгоритмов?

Асимптотический анализ сложности алгоритмов - это метод оценки производительности алгоритмов, который позволяет более общим и устойчивым способом оценивать, как быстро или эффективно алгоритм выполняется при увеличении размера входных данных. Он основан на теоретическом исследовании поведения алгоритмов при стремлении размера входных данных к бесконечности. Асимптотический анализ позволяет выявлять основные характеристики производительности алгоритма и сравнивать их между собой, не углубляясь в конкретные детали реализации или специфические характеристики входных данных.

20) Объясните смысл асимптотических оценок времени работы алгоритмов в нотациях θ, O, Ω, o, ω.

Асимптотические оценки времени работы алгоритмов в нотациях θ, O, Ω, o, ω используются для определения верхней и нижней границ временной сложности алгоритмов в зависимости от размера входных данных.

1. O-нотация (O): Это указывает на верхнюю границу временной сложности алгоритма. Если мы говорим, что алгоритм имеет временную сложность $O(f(n))$, это означает, что для всех значений размера входных данных n, время выполнения алгоритма ограничивается функцией $f(n)$ с точностью до постоянного множителя.

2. θ-нотация (θ): Это указывает на точную границу временной сложности алгоритма. Если алгоритм имеет временную сложность θ(f(n)), это означает, что временная сложность алгоритма равномерно оценивается сверху и снизу функцией f(n).

3. Ω-нотация (Ω): Это указывает на нижнюю границу временной сложности алгоритма. Если алгоритм имеет временную сложность Ω(f(n)), то это означает, что для всех значений размера входных данных n, время выполнения алгоритма ограничивается снизу функцией f(n) с точностью до постоянного множителя.

4. o-нотация (o): Это указывает на строгую верхнюю границу временной сложности алгоритма. Если алгоритм имеет временную сложность o(f(n)), это означает, что время выполнения алгоритма строго меньше функции f(n).

5. ω-нотация (ω): Это указывает на строгую нижнюю границу временной сложности алгоритма. Если алгоритм имеет временную сложность ω(f(n)), это означает, что время выполнения алгоритма строго больше функции f(n).

Эти нотации помогают оценить производительность алгоритма и сравнить его эффективность при увеличении размера входных данных.

21) В чём отличие нотаций O и o, а также Ω и ω?

Отличие между нотациями O и o, а также Ω и ω заключается в том, что в каждой паре первая нотация обозначает верхнюю или нижнюю оценку временной сложности алгоритма, а вторая - строгую верхнюю или нижнюю границу.

1. O-нотация (O) и o-нотация (o):

- O-нотация (O) указывает на верхнюю границу временной сложности алгоритма. Это оценка, которая гарантирует, что время выполнения

алгоритма не превосходит функции f(n) с постоянным множителем при увеличении размера входных данных.

- O-нотация (o), с другой стороны, указывает на строгую верхнюю границу временной сложности. Это оценка, при которой время выполнения алгоритма строго меньше функции f(n) с постоянным множителем при увеличении размера входных данных.

2. Ω-нотация (Ω) и ω-нотация (ω):

- Ω-нотация (Ω) указывает на нижнюю границу временной сложности алгоритма. Это оценка, которая гарантирует, что время выполнения алгоритма не уходит ниже функции f(n) с постоянным множителем при увеличении размера входных данных.

- ω-нотация (ω), с другой стороны, указывает на строгую нижнюю границу временной сложности. Это оценка, при которой время выполнения алгоритма строго превосходит функцию f(n) с постоянным множителем при увеличении размера входных данных.

Таким образом, важно понимать разницу между O и o, а также между Ω и ω, чтобы точно определить, как оценивать эффективность алгоритма в зависимости от его временной сложности.

22) Какие нотации чаще всего используют в практике асимптотической оценки сложности алгоритмов?

O, Ω, Θ, ω, o

23) Что такое асимптотически точная оценка сложности?

Оценка сложности алгоритма с асимптотически точной нотацией (Θ) позволяет оценить скорость работы алгоритма при увеличении размера входных данных. Это помогает определить, как быстро или медленно выполняется алгоритм, а также выбрать наиболее эффективный алгоритм для решения определенной задачи.

24) Назовите основные правила подсчёта асимптотической сложности.

Правило сложения: если f(n) и g(n) являются функциями сложности, то их сумма f(n)+g(n) также является функцией сложности. Правило умножения: если f(n) и g(n) являются функциями сложности, то их произведение f(n)*g(n) также является функцией сложности. Правило деления: если f(n) является функцией сложности, то функция 1/f(n) также является функцией сложности.

25) Назовите основные классы сложности алгоритмов.

1, log2n, n, n* log2n, n2, n3, 2n, n!

26) Перечислите свойства асимптотических сравнений.

- Транзитивность
- Рефлексивность
- Симметричность
- Перестановочная симметрия

6 ВЫВОДЫ

В ходе практической работы были выполнены следующие задачи:

- Получены навыки по анализу вычислительной сложности алгоритмов сортировки и определению наиболее эффективного алгоритма;

- Проведён анализ алгоритмов сортировки простым слиянием и Шелла со смещением Д.Кнута вторым способом;

- Были реализованы программы алгоритмов сортировки простым слиянием и Шелла со смещением Д.Кнута вторым способом;

- Проведено тестирование программ для алгоритмов сортировки простым слиянием и Шелла со смещением Д.Кнута вторым способом;

- Построены графики функции роста $T_п$ алгоритмов сортировки простым слиянием и Шелла со смещением Д.Кнута вторым способом;

- Произведено сравнение алгоритмов простой сортировки вставками, сортировки простым слиянием и Шелла со смещением Д.Кнута вторым способом;

- Проведен анализ асимптотической сложности алгоритмов сортировки простыми вставками, сортировки простым слиянием и Шелла со смещением Д.Кнута вторым способом;

-Произведено сравнение асимптотической сложности алгоритмов сортировки простыми вставками, сортировки простым слиянием и Шелла со смещением Д.Кнута вторым способом;

- Проведено определение наиболее эффективного алгоритма.

Таким образом, главную цель практической работы, а именно получение навыков по анализу вычислительной сложности алгоритмов сортировки и определению наиболее эффективного алгоритма, 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. 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).

12. 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).