



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: Борзов Дмитрий Олегович  
Фамилия И.О

Группа: ИКБО-12-22  
Номер группы

Москва - 2023

## **1 ЦЕЛЬ РАБОТЫ**

### **1.1 Цель:**

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

## 2 ХОД РАБОТЫ

### 2.1 Задание 1

#### 2.1.1 Формулировка задачи:

Эмпирическая оценка эффективности алгоритмов

Требования по выполнению задания:

1. Разработать алгоритм ускоренной сортировки, определенной в варианте (приложение 1), 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 ускоренной сортировки.

3. Разработать алгоритм быстрой сортировки, определенной в варианте (приложение 1), 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.

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

$T_n(n) = C_f + M_f$  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 и 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.

### 2.1.2 Алгоритм ускоренной сортировки Шелла со смещениями Д. Кнута.

Алгоритм сортировки Шелла со смещениями Д. Кнута в общем виде можно описать следующим образом:

Вычисляем последовательность промежутков длиной  $d[i]$ , используя формулу  $d[i-1]=3*d[i]+1$ , где  $d_0 = 1$ ,  $i=1,2,...,t$ , и  $t = \log_3(n - 1)$ . Последовательность промежутков должна заканчиваться промежутком, равным 1.

Для каждого промежутка  $d[i]$  выполняем сортировку вставками элементов массива с шагом  $d[i]$ . То есть, сравниваем и меняем местами элементы массива, находящиеся на расстоянии  $d[i]$  друг от друга.

Повторяем шаг 2 для всех промежутков, начиная с последнего и заканчивая промежутком, равным 1.

Когда все промежутки будут обработаны, массив будет отсортирован по возрастанию.

Например, для массива [8, 3, 6, 4, 9, 2, 1, 5, 7] последовательность промежутков будет следующей: 4, 1. После сортировки вставками с шагом 4 массив будет выглядеть так: [9, 2, 6, 4, 8, 3, 1, 5, 7]. Затем, после сортировки вставками с шагом 1 массив будет отсортирован и иметь вид: [1, 2, 3, 4, 5, 6, 7, 8, 9]. Блок схема алгоритма (рис. 1).

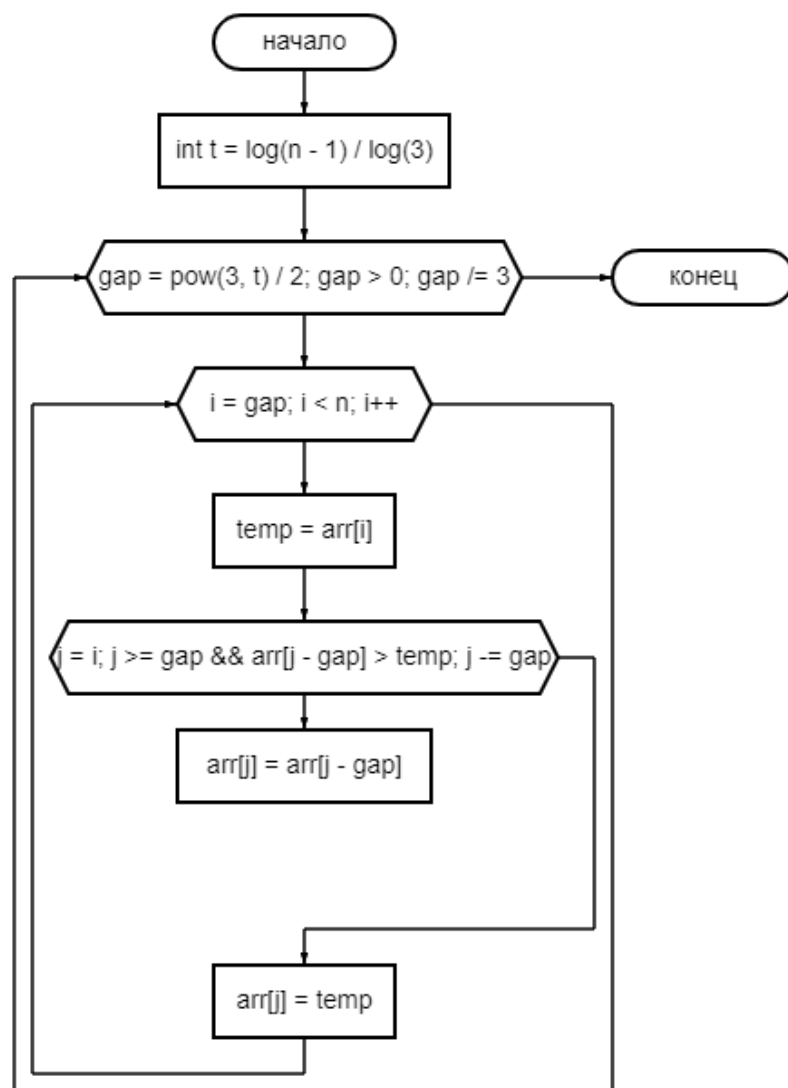


Рисунок 1 – Блок-схема алгоритма

### 2.1.3 Код программы

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono>
using namespace std;

// Функция реализующая алгоритм сортировки Шелла со смещениями Д.Кнута
void shell_sort_knuth(int arr[], int n)
{
    chrono::steady_clock::time_point begin = chrono::steady_clock::now();
    unsigned long long cnt = 0;
    int gap, i, j, temp;

    int t = log(n - 1) / log(3); // вычисляем количество смещений t
    for (gap = pow(3, t) / 2; gap > 0; gap /= 3) { // вычисляем промежутки
        for (i = gap; i < n; i++) { // сортировка вставками с шагом gap
            temp = arr[i]; // сохраняем текущий элемент во временной переменной
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap]; // сдвигаем элементы на промежуток gap вправо
            }
            arr[j] = temp; // вставляем сохраненный элемент на нужное место
        }
    }

    chrono::steady_clock::time_point end = chrono::steady_clock::now();
    cout << "\ncnt:" << cnt << endl;
    cout << "Time difference = " << chrono::duration_cast<chrono::microseconds>(end
- begin).count() << "[ms]" << endl;
    cout << "Time difference = " << chrono::duration_cast<chrono::nanoseconds>(end
- begin).count() << "[ns]\n" << endl;
}

int main() // Основная функция
{
    srand(time(NULL)); // Инициализируем генератор случайных чисел
    int n = 100; // Размер массива
    int* arr = new int[n]; // Создаем массив нужного размера

    // Заполняем массив случайными числами
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100; // Генерируем случайное число от 0 до 99
    }

    shell_sort_knuth(arr, n);

    return 0;
}
```

### 2.1.4 Тестирование

Результат работы программы для массива размерности 10 (рис. 2).

```
28 41 0 39 26 57 41 31 56 81  
  
Time difference = 10[ms]  
Time difference = 10500[ns]  
  
Sorted array: 0 26 28 31 39 41 41 56 57 81
```

Рисунок 2 – Тест программы на массиве  $n = 10$

### 2.1.5 Эмпирическая оценка вычислительной сложности алгоритма

Таблица 1 – Сводная таблица результатов

| <b>n</b> | <b>T(n), мс</b> | <b><math>T_{\Pi}(n)</math></b> |
|----------|-----------------|--------------------------------|
| 100      | 0.017           | 2286                           |
| 1000     | 0.069           | 35976                          |
| 10000    | 0.746           | 486810                         |
| 100000   | 8.996           | 6259518                        |
| 1000000  | 103.691         | 73825810                       |

### 2.1.6 Определение ёмкостной сложности

В нашем случае, мы используем дополнительную память только для хранения временного значения `temp`, которое используется для вставки элементов на нужное место в массиве. Таким образом, требуемая дополнительная память для выполнения алгоритма будет пропорциональна размеру `temp`, то есть константна и не зависит от размера входного массива.

### 2.1.7 Алгоритм быстрой сортировки (Хоара)

Быстрая сортировка Хоара (англ. QuickSort) - это эффективный алгоритм сортировки, который широко применяется в практике программирования. Он использует подход "разделяй и властвуй", основанный на разбиении массива на меньшие подмассивы и их последующей сортировке.



Алгоритм быстрой сортировки Хоара состоит из следующих шагов:

1. Выбрать опорный элемент из массива. Опорный элемент может быть выбран случайным образом или же по определенному критерию, например, как первый элемент, последний элемент или средний элемент массива.
2. Разбить массив на две части так, чтобы элементы, меньшие опорного, находились слева от него, а элементы, большие опорного, находились справа от него. Этот процесс называется разбиением массива.
3. Рекурсивно вызвать функцию быстрой сортировки для левой и правой части массива, пока не будет достигнут базовый случай, когда размер части массива равен 1 или 0.

Блок схема алгоритма (рис. 3).

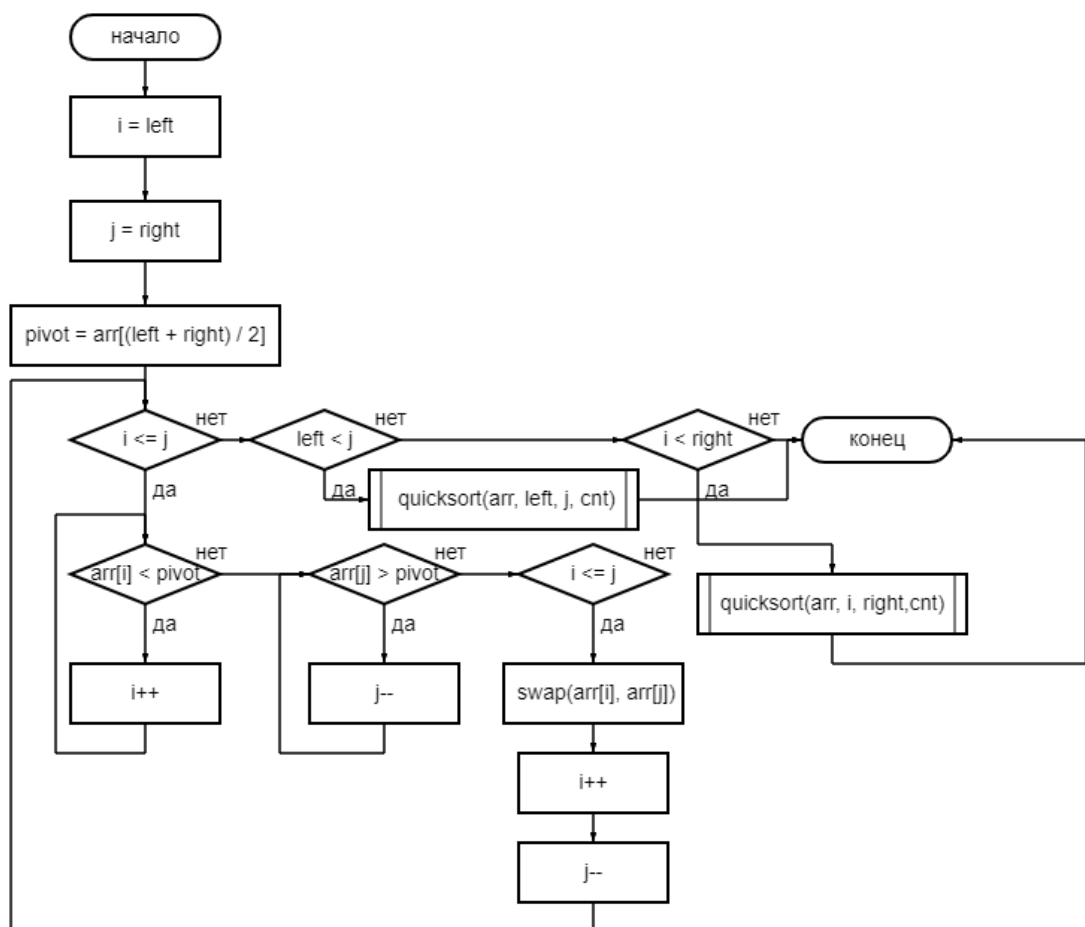


Рисунок 3 – Блок-схема алгоритма Хоара

## 2.1.8 Код алгоритма

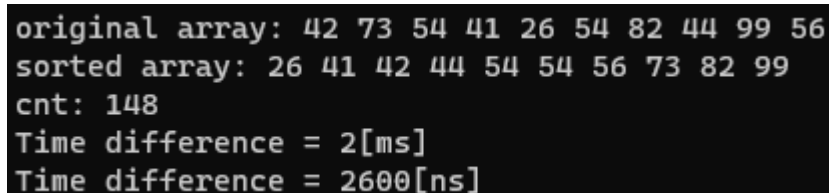
```
// объявляем функцию быстрой сортировки (Хоара), принимающую указатель на массив,
левую и правую границы массива
void quicksort(int* arr, int left, int right, unsigned long long& cnt) {
    int i = left; cnt++; // инициализируем левый индекс как left
    int j = right; cnt++; // инициализируем правый индекс как right
    int pivot = arr[(left + right) / 2]; cnt++; // выбираем опорный элемент из
среднего значения левого и правого индексов

    while (i <= j) { // запускаем цикл, пока левый индекс меньше или равен правому
        cnt++;
        while (arr[i] < pivot) { // ищем первый элемент слева, который больше или
равен опорному
            i++; cnt++;
        }
        cnt++;
        while (arr[j] > pivot) { // ищем первый элемент справа, который меньше или
равен опорному
            j--; cnt++;
        }
        cnt++;
        if (i <= j) { // если индекс слева не больше индекса справа, то меняем
элементы местами
            swap(arr[i], arr[j]); cnt++;
            i++; cnt++;
            j--; cnt++;
        }
        cnt++;
    }
    cnt++;

    if (left < j) { // если левый индекс меньше индекса j, то рекурсивно вызываем
функцию quicksort для левой половины массива
        quicksort(arr, left, j, cnt); cnt++;
    }
    cnt++;
    if (i < right) { // если индекс i меньше правого индекса, то рекурсивно вызываем
функцию quicksort для правой половины массива
        quicksort(arr, i, right, cnt); cnt++;
    }
    cnt++;
}
```

## 2.1.9 Тестирование

Результат работы программы для массива размерности 10 (рис. 4).



```
original array: 42 73 54 41 26 54 82 44 99 56
sorted array: 26 41 42 44 54 54 56 73 82 99
cnt: 148
Time difference = 2[ms]
Time difference = 2600[ns]
```

Рисунок 4 – Тест программы на массиве  $n = 10$

### 2.1.10 Эмпирическая оценка вычислительной сложности алгоритма

Таблица 2 – Сводная таблица результатов

| <b>n</b> | <b>T(n), мс</b> | <b>T<sub>п</sub>(n)</b> |
|----------|-----------------|-------------------------|
| 100      | 0.013           | 2935                    |
| 1000     | 0.149           | 41225                   |
| 10000    | 1.523           | 500689                  |
| 100000   | 17.188          | 5952310                 |
| 1000000  | 223.011         | 73053554                |

### 2.1.11 Определение ёмкостной сложности

Пространственная сложность алгоритма Хоара относительно используемой дополнительной памяти линейно зависит от размера входных данных. Это связано с тем, что алгоритм Хоара сортирует массив "на месте", то есть не создает новый массив для хранения отсортированных элементов, а изменяет порядок элементов в исходном массиве. Таким образом, для выполнения алгоритма требуется только константный объем дополнительной памяти для хранения указателей на границы подмассивов и выбранного элемента в качестве опорного. В связи с этим, алгоритм Хоара является эффективным в использовании памяти и может использоваться для сортировки массивов большого размера.

### 2.1.12 Сравнение трёх алгоритмов

Таблица 3 – Сводная таблица результатов insertionSort

| <b>n</b> | <b>T(n), мс</b> | <b>T<sub>п</sub></b> |
|----------|-----------------|----------------------|
| 100      | 0.007           | 7966                 |
| 1000     | 0.567           | 745855               |
| 10000    | 43.630          | 73641304             |
| 100000   | 4442.693        | 7417950757           |
| 1000000  | 449465.956      | 742713072145         |

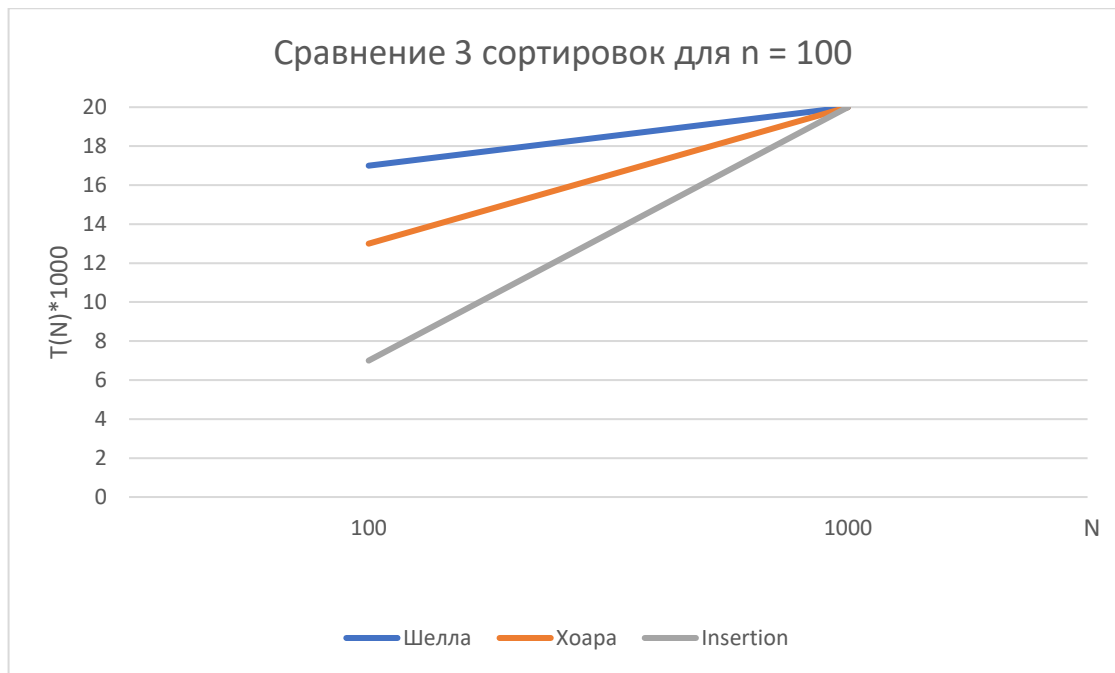


График 1



График 2

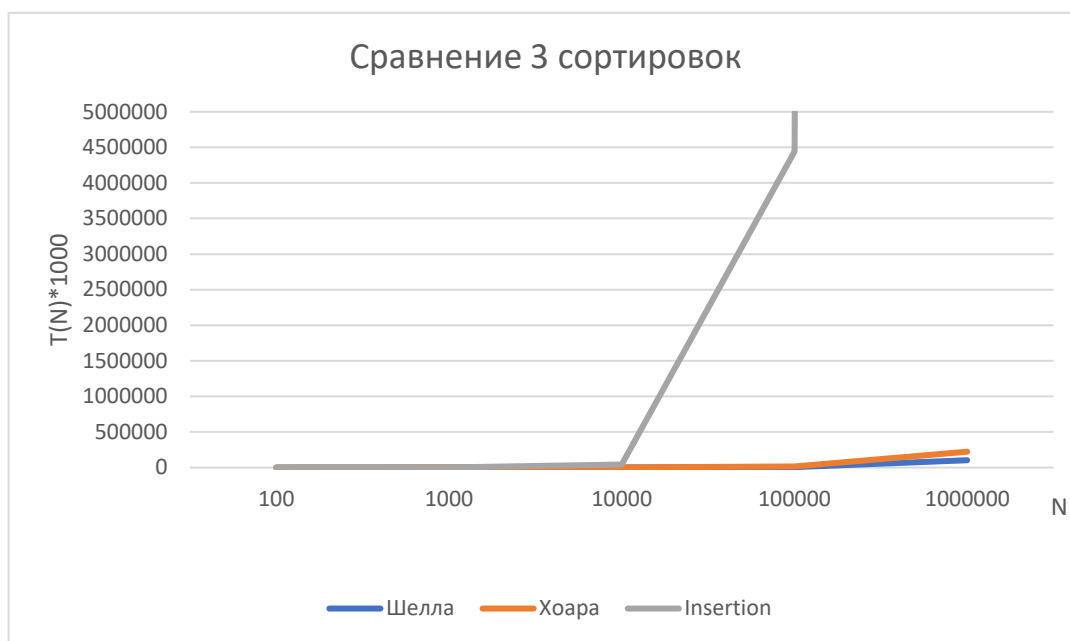


График 3

Для небольших массивов с  $n$  до 100, наиболее эффективным алгоритмом в среднем случае является сортировка вставками (Insertion Sort). В таблице видно, что для малых размеров массивов (например,  $n=100$ ), сортировка вставками работает быстрее, чем сортировка Шелла и Хоара. Это связано с тем, что сортировка вставками имеет меньшую константу времени выполнения, что делает ее более эффективной для маленьких массивов.

Для больших массивов с  $n > 100$ , наиболее эффективным алгоритмом в среднем случае является сортировка Шелла. Это видно из таблицы, где сортировка Шелла работает быстрее, чем сортировка Хоара (Quicksort) для всех размеров массивов, начиная от  $n=100$ .

Таким образом, в среднем случае, для небольших массивов с  $n$  до 1000, наиболее эффективным алгоритмом является сортировка вставками, а для больших массивов с  $n > 100$  - сортировка Шелла.

### 2.1.13 Дополнительные прогоны программ

Таблица 4 – Сводная таблица результатов Шелла (убывающий массив)

| <b>n</b> | <b>T(n), мс</b> | <b>T<sub>п</sub>(n)</b> |
|----------|-----------------|-------------------------|
| 100      | 0.014           | 1838                    |
| 1000     | 0.036           | 29682                   |
| 10000    | 0.361           | 412298                  |
| 100000   | 4.244           | 5073474                 |
| 1000000  | 56.175          | 60188314                |

Таблица 5 – Сводная таблица результатов Шелла (возрастающий массив)

| <b>n</b> | <b>T(n), мс</b> | <b>T<sub>п</sub>(n)</b> |
|----------|-----------------|-------------------------|
| 100      | 0.011           | 1378                    |
| 1000     | 0.030           | 21842                   |
| 10000    | 0.260           | 300354                  |
| 100000   | 2.266           | 3822898                 |
| 1000000  | 33.015          | 46405730                |

Таблица 6 – Сводная таблица результатов Хоара (убывающий массив)

| <b>n</b> | <b>T(n), мс</b> | <b>T<sub>п</sub>(n)</b> |
|----------|-----------------|-------------------------|
| 100      | 0.008           | 1996                    |
| 1000     | 0.057           | 24638                   |
| 10000    | 0.619           | 324950                  |
| 100000   | 5.620           | 4005406                 |
| 1000000  | 59.148          | 44742938                |

Таблица 7 – Сводная таблица результатов Хоара (возрастающий массив)

| <b>n</b> | <b>T(n), мс</b> | <b>T<sub>п</sub>(n)</b> |
|----------|-----------------|-------------------------|
| 100      | 0.008           | 1841                    |
| 1000     | 0.050           | 23127                   |
| 10000    | 0.518           | 309917                  |
| 100000   | 4.826           | 3855381                 |
| 1000000  | 49.921          | 43242907                |

Для обоих алгоритмов время сортировки возрастает при уменьшении упорядоченности массива (т.е. при переходе от возрастающего массива к убывающему), но оба алгоритма показывают лучшую производительность на возрастающих массивах.

Таким образом, можно сделать вывод, что производительность алгоритмов сортировки зависит от исходной упорядоченности массива, и для определенных типов упорядоченности (возрастающий или убывающий).

## **2.2 Задание 2**

### **2.2.1 Формулировка задачи:**

Асимптотический анализ сложности алгоритмов

Требования по выполнению задания:

1. Из материалов предыдущей практической работы приведите в отчёте формулы  $T(n)$  функций роста алгоритма простой сортировки в лучшем и худшем случае (того же алгоритма, что и в задании 1).
2. На основе определений соответствующих нотаций получите асимптотическую оценку вычислительной сложности простого алгоритма сортировки:
  - в  $O$ -нотации (оценка сверху) для анализа худшего случая;
  - в  $\Omega$ -нотации (оценка снизу) для анализа лучшего случая.
3. Получите (если это возможно) асимптотически точную оценку вычислительной сложности алгоритма в нотации  $\theta$ .
4. Реализуйте графическое представление функции роста и полученных асимптотических оценок сверху и снизу.

5. Привести справочную информацию о вычислительной сложности усовершенствованного и быстрого алгоритмов сортировки, заданных в вашем варианте.

6. Общие результаты свести в табл. 2.

### 2.2.3 Анализ алгоритма insertionSort

Лучший случай для сортировки методом Простой вставки возникает, когда массив уже отсортирован. В этом случае внутренний цикл не выполняется, а функция роста  $T(n) = 5n - 4$

Худший случай возникает, когда массив отсортирован в обратном порядке. В этом случае внутренний цикл будет выполнен максимальное количество раз для каждого элемента, а функция роста  $T(n) = n^2 + 4n - 4 + \frac{n^2 - n}{2}$

#### 2.2.3.1 Асимптотическая оценка

В О-нотации для худшего случая алгоритм сортировки вставками имеет асимптотическую оценку  $O(n^2)$ , так как в этом случае внутренний цикл будет выполнен максимальное количество раз для каждого элемента, что приводит к квадратичному росту функции  $T(n) = n^2 + 4n - 4 + \frac{n^2 - n}{2}$ .

Докажем:  $T(n) = n^2 + 4n - 4 + \frac{n^2 - n}{2}$ .

Для этого определим константы  $c_1$ ,  $c_2$  и  $n_0$ , для которых справедливо неравенство:  $c_1 * n^2 \leq n^2 + 4n - 4 + \frac{n^2 - n}{2} \leq c_2 * n^2$  для всех  $n \geq n_0$ .

Разделив неравенство на  $n^2$ , получим:  $c_1 \leq \frac{3n^2 + 7n - 8}{2n^2} \leq c_2$ .

$$\lim_{n \rightarrow \infty} \left( \frac{3n^2 + 7n - 8}{2n^2} \right) = \frac{3}{2}$$

Правая часть  $\frac{3n^2 + 7n - 8}{2n^2} \leq c_2$  выполнится для всех  $n \geq 1$ , если выбрать  $c_2 \geq \frac{3}{2}$



В  $\Omega$ -нотации для лучшего случая алгоритм сортировки вставками имеет асимптотическую оценку  $\Omega(n)$ , так как в этом случае массив уже отсортирован, и внутренний цикл не будет выполняться, что приведет к линейному росту функции  $T(n) = 5n-4$ .

Докажем:  $T(n) = 5n-4$ .

Для этого определим константы  $c_3$ ,  $c_4$  и  $n_0$ , для которых справедливо неравенство:  $c_3 * n \leq 5n-4 \leq c_4 * n$  для всех  $n \geq n_0$ .

Разделив неравенство на  $n$ , получим:  $c_3 \leq 1 \leq c_4$ .

Получаем:  $c_4 \geq 1$

### **2.2.3.2 Асимптотически точная оценка**

Для сортировки вставками точной асимптотической оценки не существует, так как она имеет различную вычислительную сложность в зависимости от исходных данных. В лучшем случае, когда массив уже отсортирован, сложность алгоритма будет  $\Omega(n)$ , а в худшем случае, когда массив отсортирован в обратном порядке, сложность будет  $O(n^2)$ .

Это связано с тем, что алгоритм вставок просто проходит по элементам массива и вставляет каждый элемент в правильное место, если оно найдено. В лучшем случае все элементы уже находятся в правильном порядке, и вставки не требуются. В худшем случае каждый новый элемент приходится вставлять на первую позицию, что требует выполнения  $n$  операций вставки для каждого элемента.

Таким образом, в зависимости от конкретных исходных данных, время выполнения сортировки вставками может быть как  $\Omega(n)$ , так и  $O(n^2)$ , и точной асимптотической оценки не существует.

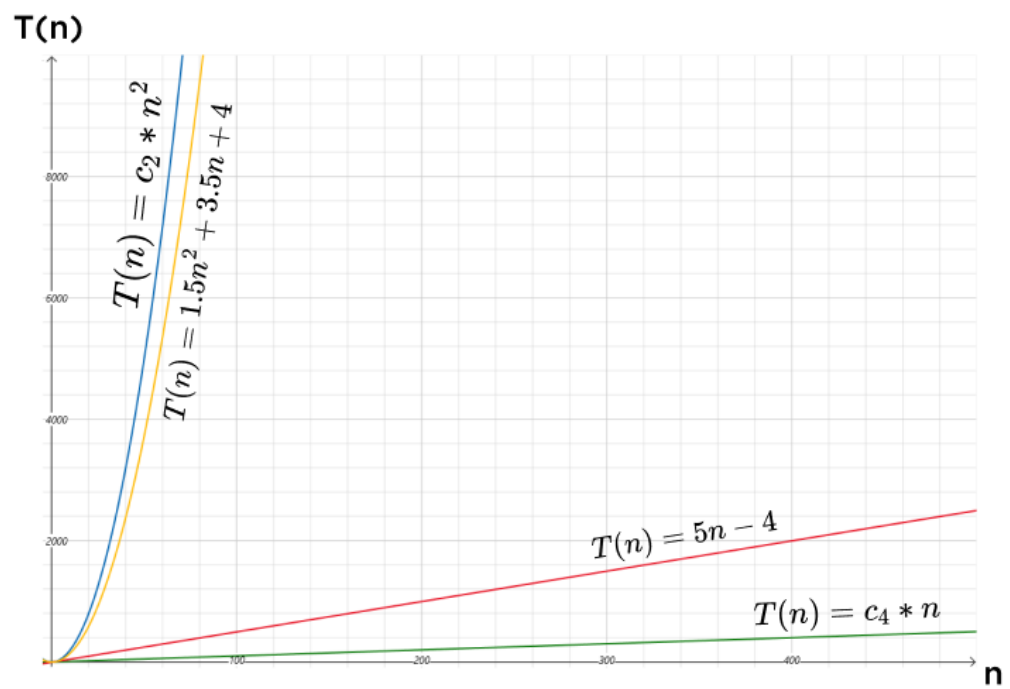


Рисунок 5 – Графическое представление функции роста и асимптотической нотации

## 2.2.4 Справочная информация вычислительной сложности для усовершенствованного и быстрого алгоритмов сортировки

Таблица 8 – Сводная таблица результатов

| Алгоритм            | Асимптотическая сложность алгоритма |                          |                                |                     |
|---------------------|-------------------------------------|--------------------------|--------------------------------|---------------------|
|                     | Наихудший случай (сверху)           | наилучший случай (снизу) | Средний случай (точная оценка) | Ёмкостная сложность |
| Простой             | $O(n^2)$                            | $\Omega(n)$              | -                              | $O(1)$              |
| Усовершенствованный | $O(n^2)$                            | $\Omega(n \cdot \log n)$ | $\Theta(n \cdot \log n)$       | $O(1)$              |
| Быстрый             | $O(n^2)$                            | $\Omega(n \cdot \log n)$ | $\Theta(n \cdot \log n)$       | $O(\log n)$         |

## 2.2.6 Вывод об эффективности алгоритмов

Наиболее эффективным алгоритмом из трёх представленных является алгоритм сортировки Шелла. Этот алгоритм демонстрирует лучшую производительность на всех размерах массивов, за исключением массивов размером  $\leq 100$ , где алгоритм сортировки вставками (Insertion Sort) оказался быстрее.

Хотя алгоритм Хоара (Quicksort) тоже показал неплохие результаты, но он уступил по производительности алгоритму Шелла (Shell Sort).

### **3 ВЫВОД**

В результате данной практической работы были получены навыки анализа вычислительной сложности алгоритмов сортировки и определения наиболее эффективного алгоритма. Были изучены различные алгоритмы сортировки и проведен анализ их времени выполнения и используемых ресурсов.

Было выяснено, что выбор наиболее эффективного алгоритма зависит от конкретной задачи и условий ее решения. Для каждой задачи необходимо проводить анализ вычислительной сложности и выбирать наиболее подходящий алгоритм.

Основной целью анализа вычислительной сложности является определение скорости выполнения алгоритмов и количества ресурсов, которые они затрачивают. Это позволяет сравнить различные алгоритмы и выбрать наиболее оптимальный для решения конкретной задачи.

Полученные навыки по анализу вычислительной сложности алгоритмов сортировки будут полезны для решения многих задач в области программирования, анализа данных и других сферах, где требуется эффективная обработка больших объемов информации.

#### 4 СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Рысин М.Л. и др. Введение в структуры и алгоритмы обработки данных. Ч. 1 - учебное пособие, 2022.pdf
2. ПР-1.1 (Теоретическая сложность алгоритма).
3. Лекционные материалы - СиАОД-1Файл
6. Оценка сложности алгоритмов <https://habr.co/>
7. Бхаргава А. Грожаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. – СПб: Питер, 2017. – 288 с.
8. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985. – 406 с.
9. Кнут Д.Э. Искусство программирования, том 3. Сортировка и поиск, 2-е изд. – М.: ООО «И.Д. Вильямс», 2018. – 832 с.
10. Седжвик Р. Фундаментальные алгоритмы на C++.  
Анализ/Структуры данных/Сортировка/Поиск. – К.: Издательство «Диасофт», 2001. – 688 с.
11. <https://www.mycplus.com/featured-articles/shell-sort-algorithm/#shell-sort-complexity>
12. [https://neerc.ifmo.ru/wiki/index.php?title=Быстрая\\_сортировка](https://neerc.ifmo.ru/wiki/index.php?title=Быстрая_сортировка)