МИНОБРНАУКИ РОСИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания №1.3

Тема:

Определение эффективного алгоритма сортировки на основе эмпирического и асимптотического методов анализа

Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнил студент: Павлов Н.С.

Группа: ИКБО-30-23

Москва 2024

СОДЕРЖАНИЕ

ЦЕЛЬ РАБОТЫ

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

1 ЗАДАНИЕ №1

1.1 ФОРМУЛИРОВКА ЗАДАЧИ

Эмпирическая оценка эффективности алгоритмов

Персональный вариант:

Усовершенствованный алгоритм: Сортировка Шелла со смещениями Д. Кнута. Способ 1

Быстрый алгоритм: Быстрая сортировка (Хоара)

1.2 РАБОТА С АЛГОРИТМОМ

1.2.1 Математическая модель решения Shell sort

Идея сортировки методом Шелла состоит в том, чтобы сортировать элементы, отстоящие друг от друга на некотором расстоянии d. Затем сортировка повторяется при меньших значениях d, и в конце процесс сортировки Шелла завершается при d = 1 (а именно обычной сортировкой вставками).

Подробный порядок действий алгоритма обозначен на блок-схеме (рис.1).

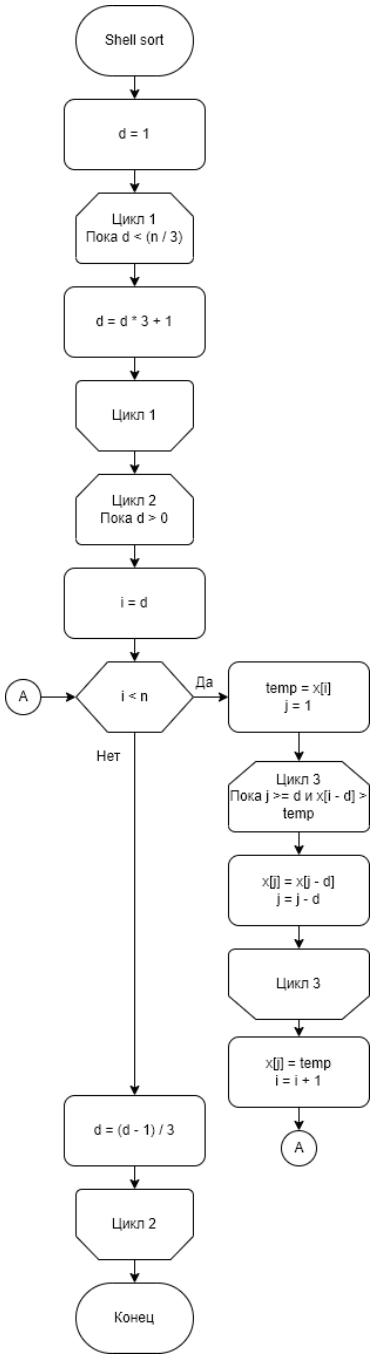

Рисунок 1 – Блок-схема алгоритма Shell sort

1.2.2 Реализация Shell sort

Составим функцию сортировки Шелла со смещениями Д. Кнута (первым способом) (рисунок 2) и проведём тестирование программы при n = 10 (рисунок 3).

```
void shell(int* x, long n, long long& sravn, long long& del)
{
    int d = 1;
    while (d < n / 3) {
        d = d * 3 + 1;
    }
    while (d > 0) {
        sravn++;
        for (int i = d; i < n; i++) {
            sravn++;
            int temp = x[i];
            del++;
            int j = i;
            while (j >= d && x[j - d] > temp) {
                sravn += 2;
                x[j] = x[j - d];
                del++;
                j -= d;
            }
            sravn += 2;
            x[j] = temp;
            del++;
        }
        sravn++;
        d = (d - 1) / 3;
    }
    sravn++;
}
```

Рисунок 2 – Реализация Shell sort

```
Введите размер массива: 10
Исходный массив: 3 0 5 0 7 2 7 3 7 2
Отсортированный массив: 0 0 2 2 3 3 5 7 7 7
Количество сравнений: 80
Количество смещений: 45
Время работы: 0 ms
```

Рисунок 3 – Тестирование Shell sort при n = 10

1.2.3 Контрольные тесты Shell sort

Проведем контрольные на массиве случайных чисел длинной n = 100, 1000, 10000, 100000, 1000000 элементов с вычислением времени выполнения T(n) – (в миллисекундах) (рисунки 4-8).

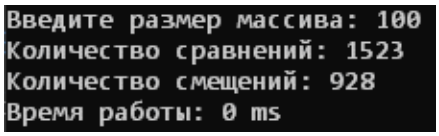

Рисунок 4 – Тестирование алгоритма при n = 100

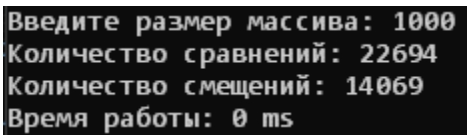

Рисунок 5 – Тестирование алгоритма при n = 1000

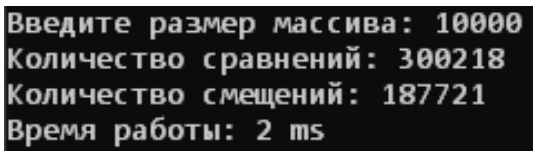

Рисунок 6 – Тестирование алгоритма при n = 10000

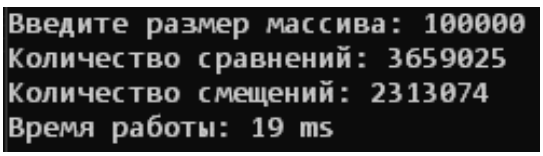

Рисунок 7 – Тестирование алгоритма при n = 100000

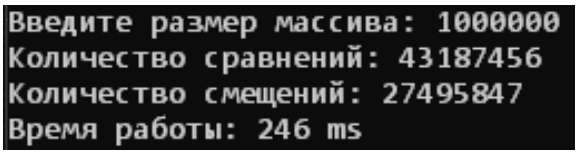

Рисунок 8 – Тестирование алгоритма при n = 1000000

Результаты эмпирической оценки вычислительной сложности алгоритма представлены в таблице 1.

Таблица 1 – Сводная таблица результатов

n	**T(n), мс**	**$T_п = C_п + M_п$**
100	0	2451
1000	0	36763
10000	2	487939
100000	19	5972099
1000000	246	70683303

1.2.4 Оценка емкостной сложности Shell sort

Определим количество выделяемой памяти на каждый тип переменных с помощью функции sizeof() для конкретной системы (рис. 9).

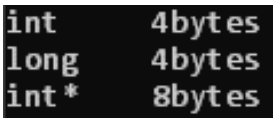

Рисунок 9 – количество памяти, занимаемой переменными в конкретной системе

Перечислим все переменные, одновременно существующие в алгоритме, и размер в памяти, занимаемый ими (табл. 3).

Таблица 2 – Таблица определения размера переменных

Имя переменной	Тип переменной	Занимаемая память в байтах
i	int	4
j	int	4
d	int	4
temp	int	4
n	long	4
x	int*	8
x[1]…x[n]	Массив int	4*n

Сложив значения третьего столбца, получаем, что емкостная сложность алгоритма: 4*5+8+4*n = 28+4n байт

1.2.5 Математическая модель решения quicksort

Рассматриваемый прием работает быстро и реализуется в несколько этапов:

1. Выбирается опора. Необходимо определить опорный элемент массива. Чаще всего им выступает самый левый или самый правый компонент множества.

2. Происходит разделение элементов. Переупорядочивание массива и его компонентов осуществляется так, чтобы все составляющие меньше опорного располагались перед ним, а все элементы, которые больше «опоры» – после.

Равные значения допускаются в любых направлениях. Стержень после описанных манипуляций будет занимать свое конечное положение.

3. Повторение. Здесь осуществляется рекурсия описанных ранее шагов для подмассива элементов с меньшими значениями, чем у опорного. Отдельно необходимо использовать метод к подмножеству с компонентами, значения которых превосходят «опору».

Подробный порядок действий алгоритма обозначен на блок-схеме (рис.10).

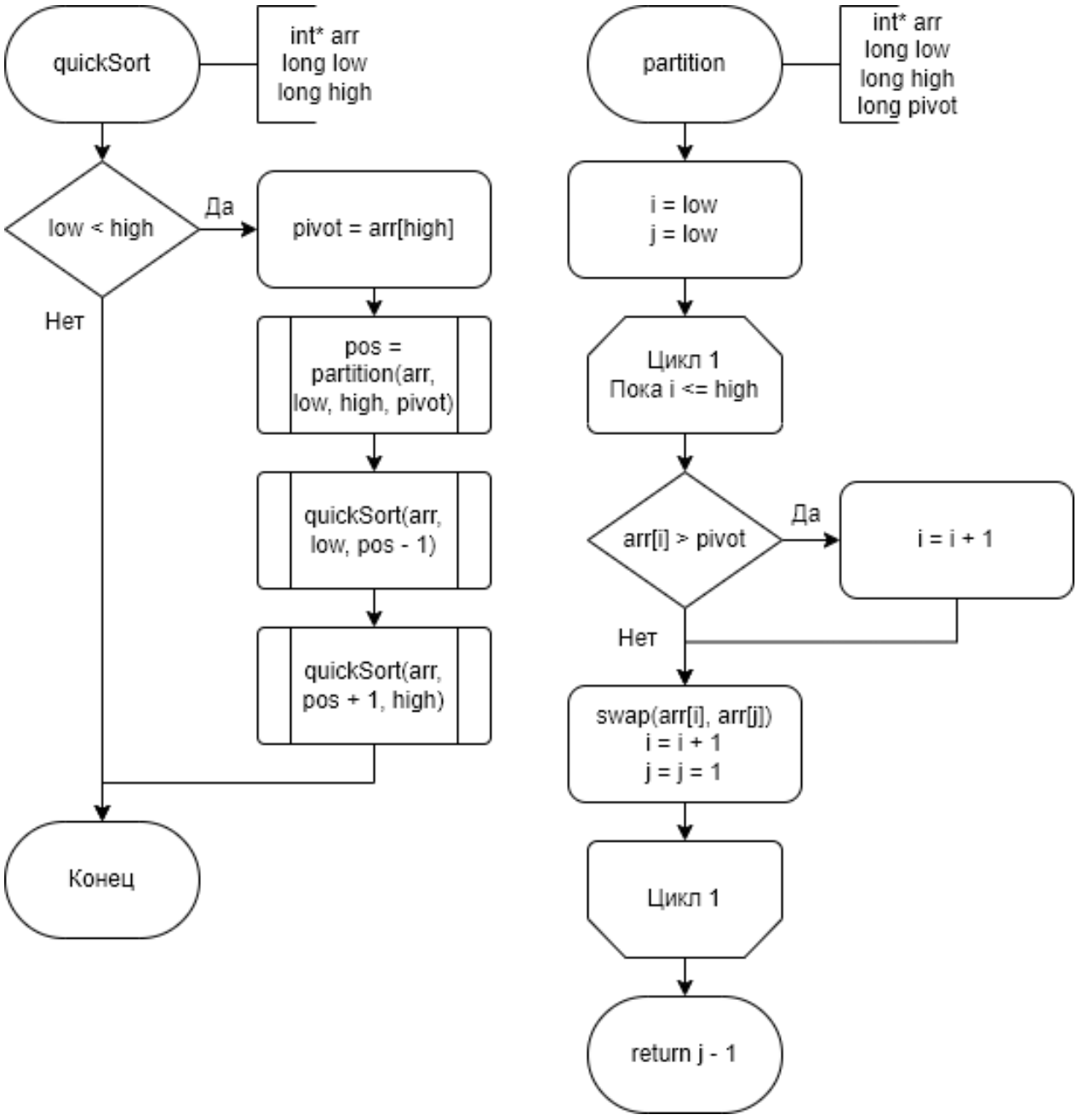

Рисунок 10 – Блок-схема алгоритма quicksort

1.2.6 Реализация quicksort

Составим функцию быстрой сортировки Хоара (рисунок 11) и проведём тестирование программы при n = 10 (рисунок 12).

```
int partition(int* arr, long low, long high, long pivot, long long& sravn, long long& del) {
    long i = low;
    long j = low;
    while (i <= high) {
        sravn++;
        if (arr[i] > pivot) {
            i++;
        }
        else {
            long z = arr[i];
            arr[i] = arr[j];
            arr[j] = z;
            del += 3;
            i++;
            j++;
        }
        sravn++;
    }
    sravn++;
    return j - 1;
}

void quickSort(int* arr, long low, long high, long long& sravn, long long& del) {
    if (low < high) {
        sravn++;
        long pivot = arr[high];
        long pos = partition(arr, low, high, pivot, sravn, del);
        quickSort(arr, low, pos - 1, sravn, del);
        quickSort(arr, pos + 1, high, sravn, del);
    }
    sravn++;
}
```

Рисунок 11 – Реализация quicksort

```
Введите размер массива: 10
Исходный массив: 4 3 2 0 9 0 6 9 6 8
Отсортированный массив: 0 0 2 3 4 6 6 8 9 9
Количество сравнений: 91
Количество смещений: 81
Время работы: 0 ms
```

Рисунок 12 – Тестирование quicksort при n = 10

1.2.7 Контрольные тесты quicksort

Проведем контрольные на массиве случайных чисел длинной n = 100, 1000, 10000, 100000, 1000000 элементов с вычислением времени выполнения T(n) – (в миллисекундах) (рисунки 13-17).

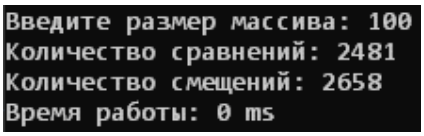

Рисунок 13 – Тестирование алгоритма при n = 100

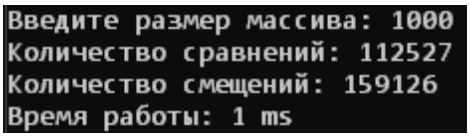

Рисунок 14 – Тестирование алгоритма при n = 1000

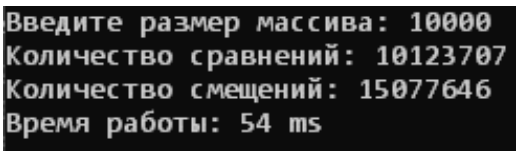

Рисунок 15 – Тестирование алгоритма при n = 10000

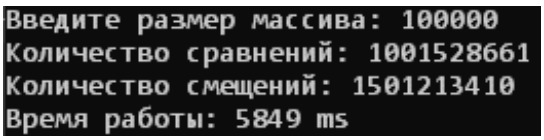

Рисунок 16 – Тестирование алгоритма при n = 100000

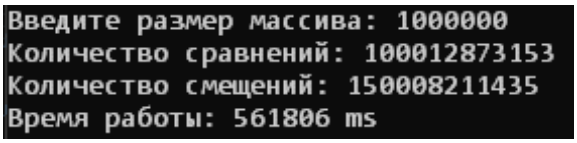

Рисунок 17 – Тестирование алгоритма при n = 1000000

Результаты эмпирической оценки вычислительной сложности алгоритма представлены в таблице 3.

Таблица 3 – Сводная таблица результатов

n	**T(n), мс**	**$T_п = C_п + M_п$**
100	0	5139
1000	1	271653
10000	54	25201353
100000	5849	2502742071
1000000	561806	250021084588

1.2.8 Оценка емкостной сложности quicksort

Определим количество выделяемой памяти на каждый тип переменных с помощью функции sizeof() для конкретной системы (рис. 18).

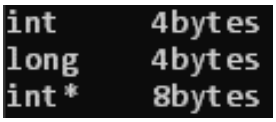

Рисунок 18 – количество памяти, занимаемой переменными в конкретной системе

Перечислим все переменные, одновременно существующие в алгоритме, и размер в памяти, занимаемый ими (табл. 4).

Таблица 4 – Таблица определения размера переменных

Имя переменной	Тип переменной	Занимаемая память в байтах
i	long	4
j	long	4
pos	long	4
pivot	long	4
high	long	4
low	long	4
n	long	4
arr	int*	8
arr[1]…arr[n]	Массив int	4*n

Сложив значения третьего столбца, получаем, что емкостная сложность алгоритма: 4*7+8+4*n = 36+4n байт

1.2.9 Импорт данных об Insertion sort

Проведем контрольные тесты упорядоченном по убыванию массиве длинной n = 100, 1000, 10000, 100000 элементов с вычислением времени выполнения T(n) – (в миллисекундах) (рисунки 19-22).

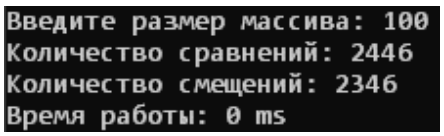

Рисунок 19 – Тестирование алгоритма при n = 100

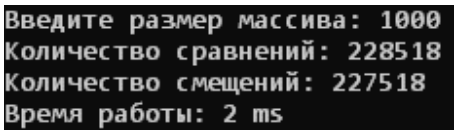

Рисунок 20 – Тестирование алгоритма при n = 1000

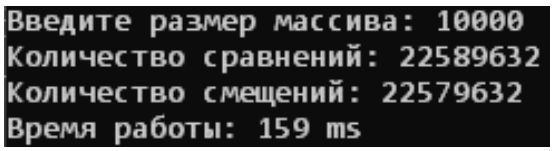

Рисунок 21 – Тестирование алгоритма при n = 10000

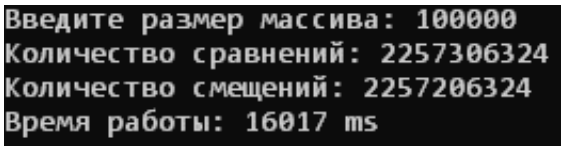

Рисунок 22 – Тестирование алгоритма при n = 100000

Результаты эмпирической оценки вычислительной сложности алгоритма представлены в таблице 5.

Таблица 5 – Сводная таблица результатов

n	**T(n), мс**	**$T_п = C_п + M_п$**
100	0	4792
1000	2	456036
10000	159	45169264
100000	16017	4514512648

1.2.10 Сравнительные графики

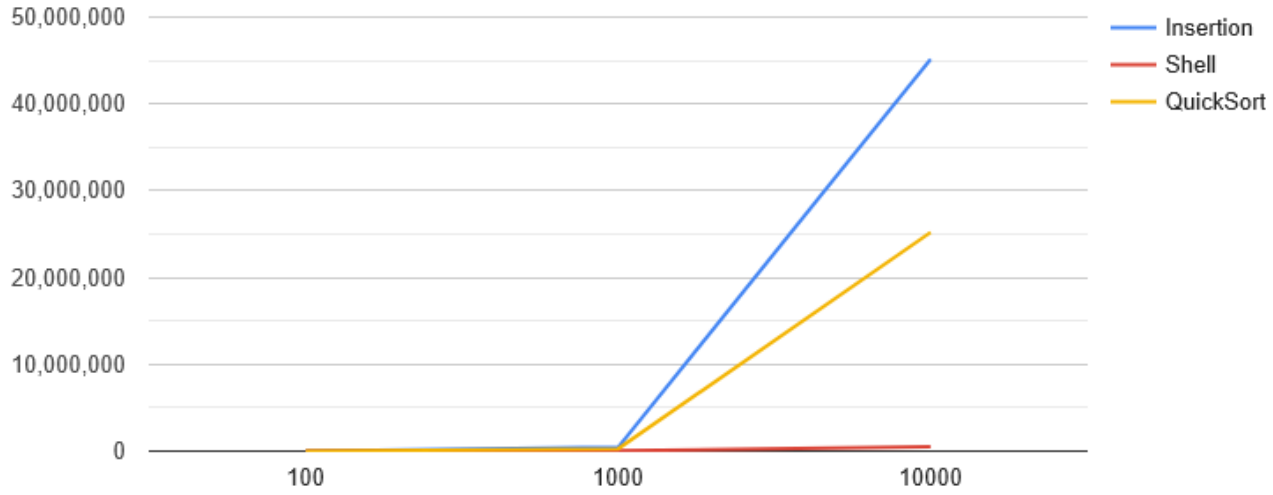

Рисунок 23 – Сравнение на малых массивах

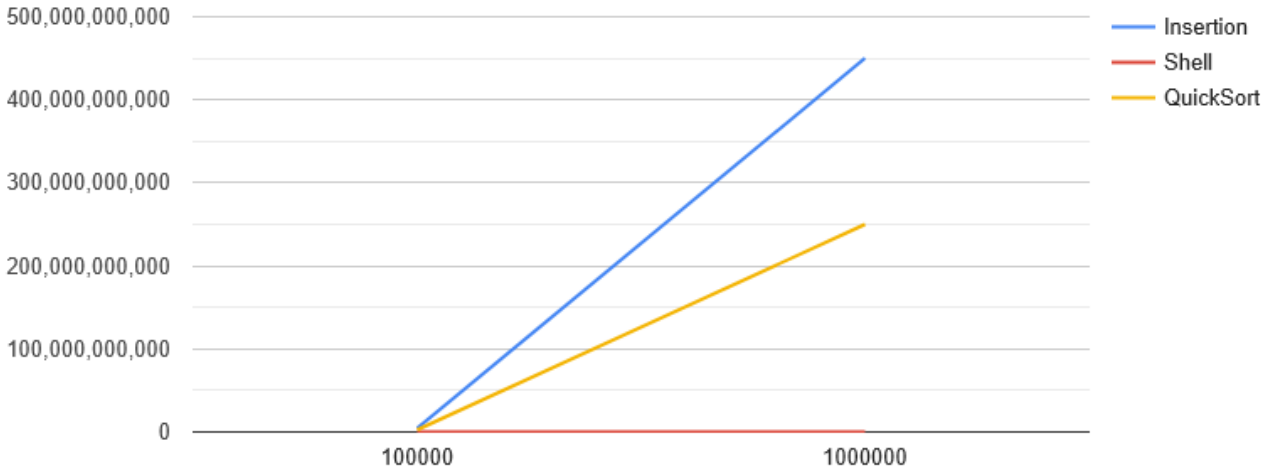

Рисунок 24 – Сравнение на больших массивах

Анализируя три таблицы полученных результатов для малых n, можно сделать вывод, что при n <1000 самая эффективная сортировка Insertion. Несмотря на то, что у нее наибольшее количество операций, она завершает свою работу быстрее чем быстрая и ускоренная сортировка. Однако при больших n сортировка Шелла эффективнее, чем другие две и это можно заметить, если сравнить время работы при одинаковых n.

1.3 ДОПОЛНИТЕЛЬНОЕ ТЕСТИРОВАНИЕ

1.3.1 Тестирование и составление таблицы для лучшего случая (Shell)

Проведем контрольные тесты на упорядоченном по возрастанию массиве длинной n = 100, 1000, 10000, 100000 элементов с вычислением времени выполнения T(n) – (в миллисекундах) (рисунки 25-28).

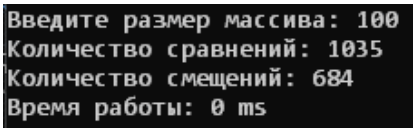

Рисунок 25 – Тестирование алгоритма при n = 100

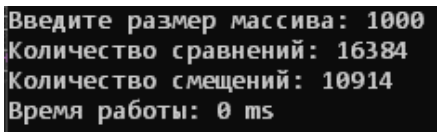

Рисунок 26 – Тестирование алгоритма при n = 1000

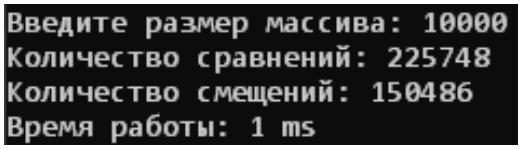

Рисунок 27 – Тестирование алгоритма при n = 10000

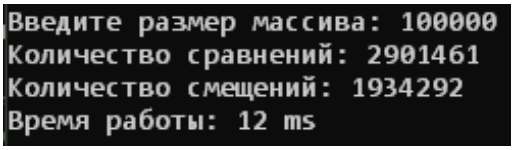

Рисунок 28 – Тестирование алгоритма при n = 100000

Результаты эмпирической оценки вычислительной сложности алгоритма представлены в таблице 6.

Таблица 6 – Сводная таблица результатов

n	**T(n), мс**	$\mathbf{T_п = C_п + M_п}$
100	0	1719
1000	0	27298
10000	1	376234
100000	12	4835753

1.3.2 Тестирование и составление таблицы для худшего случая (Shell)

Проведем контрольные тесты на упорядоченном по убыванию массиве длинной n = 100, 1000, 10000, 100000 элементов с вычислением времени выполнения T(n) – (в миллисекундах) (рисунки 29-32).

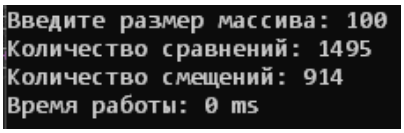

Рисунок 29 – Тестирование алгоритма при n = 100

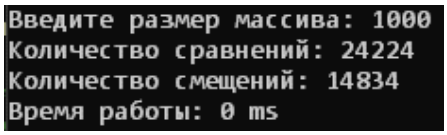

Рисунок 30 – Тестирование алгоритма при n = 1000

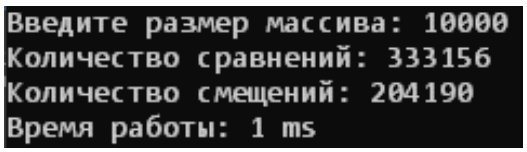

Рисунок 31 – Тестирование алгоритма при n = 10000

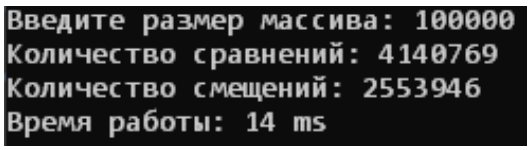

Рисунок 32– Тестирование алгоритма при n = 100000

Результаты эмпирической оценки вычислительной сложности алгоритма представлены в таблице 7.

Таблица 7 – Сводная таблица результатов

n	**T(n), мс**	**$T_п = C_п + M_п$**
100	0	2409
1000	0	39058
10000	1	537346
100000	14	6694715

1.3.3 Тестирование и составление таблицы для лучшего случая (Quick)

Проведем контрольные тесты на упорядоченном по возрастанию массиве длинной n = 100, 1000, 10000, 100000 элементов с вычислением времени выполнения T(n) – (в миллисекундах) (рисунки 33-36).

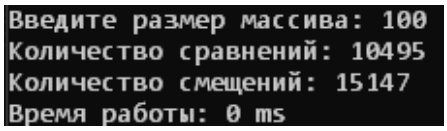

Рисунок 33 – Тестирование алгоритма при n = 100

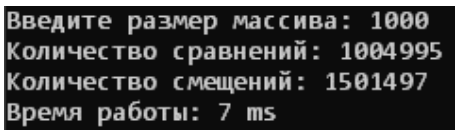

Рисунок 34 – Тестирование алгоритма при n = 1000

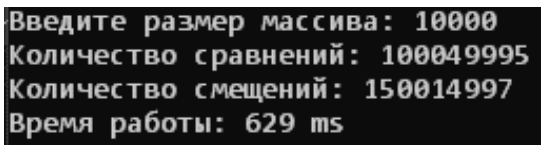

Рисунок 35 – Тестирование алгоритма при n = 10000

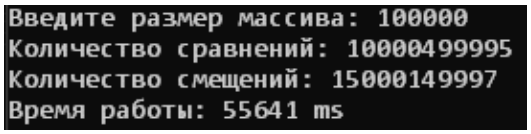

Рисунок 36 – Тестирование алгоритма при n = 100000

Результаты эмпирической оценки вычислительной сложности алгоритма представлены в таблице 8.

Таблица 8 – Сводная таблица результатов

n	**T(n), мс**	**$T_п = C_п + M_п$**
100	0	25642
1000	7	2506492
10000	629	250064992
100000	55641	25000649992

1.3.4 Тестирование и составление таблицы для худшего случая (Quick)

Проведем контрольные тесты на упорядоченном по убыванию массиве длинной n = 100, 1000, 10000, 100000 элементов с вычислением времени выполнения T(n) – (в миллисекундах) (рисунки 37-40).

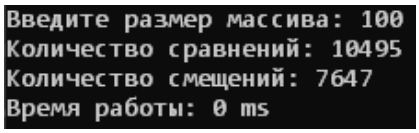

Рисунок 37 – Тестирование алгоритма при n = 100

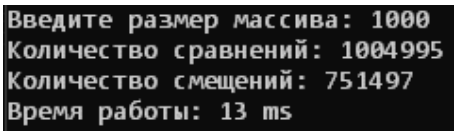

Рисунок 38 – Тестирование алгоритма при n = 1000

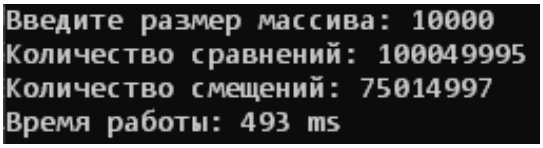

Рисунок 39 – Тестирование алгоритма при n = 10000

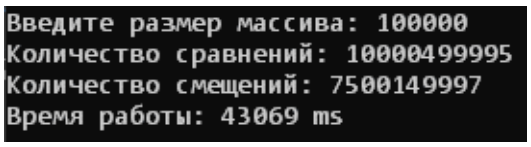

Рисунок 40– Тестирование алгоритма при n = 100000

Результаты эмпирической оценки вычислительной сложности алгоритма представлены в таблице 9.

Таблица 9 – Сводная таблица результатов

n	**T(n), мс**	$\mathbf{T_п = C_п + M_п}$
100	0	18142
1000	13	1756492
10000	493	175064992
100000	43069	17500649992

1.4 ВЫВОД

На основе результатов таблиц можно сделать вывод, что скорость выполнения алгоритма быстрой сортировки не зависит от упорядоченности массива, а вот уже сортировка Шелла зависит от нее. При этом чем ближе расположены элементы в массиве к худшему случаю, тем эффективнее становится быстрая сортировка. А вот с сортировкой Шелла наоборот.

2 ЗАДАНИЕ №2

2.1 ПОСТАНОВКА ЗАДАЧИ

Асимптотический анализ сложности алгоритмов.

2.2 РАБОТА С АЛГОРИТМОМ

2.2.1 Функции роста простой сортировки

Импортируем данные о функциях роста Insertion sort из практической работы №2

Худший случай:

$$T(n) = n + (n-1) + (n-1) + (n+1)\left(\frac{n}{2}+1\right) + \frac{n}{2}(n-1) + (n-1) \quad (1)$$
$$= n^2 + 5n - 2$$

Лучший случай:

$$T(n) = n + (n-1) + (n-1) + (n+1) + (n-1) = 5n - 2 \quad (2)$$

2.2.2 Асимптотический анализ

В O-нотации (оценка сверху) для анализа худшего случая Insertion sort: для T(n) подберем такую простую g(n) и константу C, так что C*g(n) превышает T(n), по мере того как n значительно растет. Получаем, что T(n) имеет порядок роста O(g(n)), если имеется константа c и счетчик n_0, такие что 0 <T(n) ≤ C·g(n), для n>=n_0. В нашем случае g(n) = n^2, C = 2, а n_0 = 2. Следовательно, O(n^2).

В Ω-нотации (оценка снизу) для анализа лучшего случая сортировки обменом. Найдем такую константу c такую, что для бесконечного числа значений n> n_0 выполняется неравенство T(n) ≥ c·k(n). Получим, что k(n) = n, c = 5, а n_0 = 1. Следовательно: Ω(n)

Для данного алгоритма возможно получить асимптотически точную оценку вычислительной сложности алгоритма в нотации θ. Мы получим, что T(n) = θ(n^2). Докажем, что это действительно так. Для этого определим константы c_1, c_2 и n_0, для которых справедливо:

$$c_1 * n^2 \le n^2 + 5*n - 2 \le c_2 * n^2 \text{ для всех } n \ge n_0$$

Разделив неравенство на n^2, получим:

$$c_1 \le 1 + 5/n - 2/n^2 \le c_2$$

Правая часть 1 +5/n – 2/ n^2≤ c_2 выполнится для всех n ≥ 1, если выбрать c_2 ≥ 4 (при n→∞ 5/n→0 и 2/ n^2).

Аналогично левая часть $c_1 \leq 1 + 5/n - 2/n^2$ выполнится для всех $n \geq 1$, если выбрать $c_1 \leq 1$ (при $n \to \infty$ $5/n \to 0$ и $2/n^2$).

Тогда найдены $c_1 = 1$, $c_2 = 4$ и $n_0 = 4$, а, значит, по определению, $T(n) = \theta(n^2)$, ч.т.д.

2.2.3 Графическое отображение

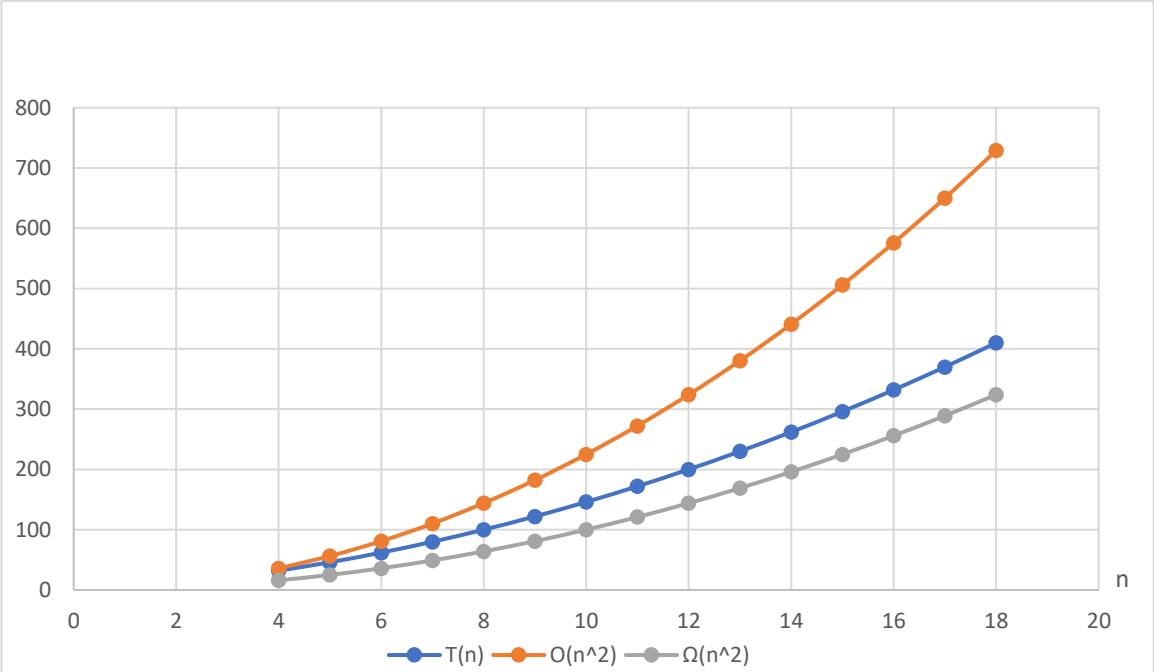

Рисунок 41 – Сравнение функции роста и полученных оценок сверху и снизу

2.2.4 Справочная информация

Сортировка Шелла со смещениями Д. Кнута. Способ 1 в худшем случае имеет сложность $O(n^2)$, в лучшем случае Ω (n*log^2(n)), в среднем θ(n*log(n)).

Сортировка слиянием в худшем случае имеет сложность $O(n^2)$, а в лучшем и среднем случае имеет одинаковую сложность, а именно: Ω (n*log(n)) и θ (n*log(n)).

Таблица 10 – Сводная таблица результатов

Алгоритм	Асимптотическая сложность алгоритма			
	Наихудший случай (сверху)	Наилучший случай (снизу)	Средний случай (точная оценка)	Ёмкостная сложность
Insertion sort	O (n^2)	Ω (n)	θ (n^2)	O (n)
Сортировка Шелла со Смещениями Д. Кнута	O (n^2)	Ω (n*log^2(n))	θ(n*log(n))	O (n)
Хоара	O (n^2)	Ω (n*log(n))	θ (n*log(n))	O (n)

2.3 ВЫВОД

Из трех сортировок, исследуемых мной, самой эффективной для большего количества элементов, является сортировка Шелла со смещениями Д. Кнута. Способ 1

СПИСОК ИНФОРМАЦТОННЫХ ИСТОЧНИКОВ

1. Рысин, М. Л. Введение в структуры и алгоритмы обработки данных. Часть 1. Сложность алгоритмов. Сортировки. Линейные структуры данных. Поиск в таблице. : учеб. пособие / М. Л. Рысин, М. В. Сартаков, М. Б. Туманова ; МИРЭА – Российский технологический университет, 2022. – 109 с. – ISBN 978-5-7339-1612-5.

2. ГОСТ 19.701-90. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения : межгосударственный стандарт : дата введения 1992-01-01 / Федеральное агентство по техническому регулированию. – Изд. официальное. – Москва : Стандартинформ, 2010. – 23 с.