МИНОБРНАУКИ РОССИИ САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

ОТЧЕТ

по лабораторной работе № 1

дисциплина «Алгоритмы и Структуры данных»

Тема: «Алгоритмы сортировки сравнением»

Студент гр. 3351 _____Фабер К.А.

Преподаватель _____ Пестерев Д.О.

Санкт-Петербург

2024

Цель лабораторной работы: реализация алгоритмов сортировки сравнением и исследование их временной сложности.

Даны следующие алгоритмы сортировки сравнением: сортировка выбором, сортировка вставками, сортировка пузырьком, сортировка слиянием, сортировка Шелла (последовательность Шелла, Хиббарда, Пратта) быстрая сортировка, пирамидальная сортировка.

Теоретическая часть.

Алгоритм	Асимптотическая временная сложность			Асимптотическая пространственная сложность		
	Лучший случай	Средний случай	Худший случай	Лучший случай	Средний случай	Худший случай
Сортировка выбором	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$	$\theta(1)$	$\theta(1)$	$\theta(1)$
Сортировка вставками	$\theta(n)$	$\theta(n^2)$	$\theta(n^2)$	$\theta(1)$	$\theta(1)$	$\theta(1)$
Сортировка пузырьком	$\theta(n)$	$\theta(n^2)$	$\theta(n^2)$	$\theta(1)$	$\theta(1)$	$\theta(1)$
Сортировка слиянием	$\theta(n\log n)$	$\theta(n\log n)$	$\theta(n\log n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Сортировка Шелла	$\theta(n^{1.667})$	$\theta\left(n^{\frac{3}{2}}\right)$	$\theta(n^2)$	$\theta(1)$	$\theta(1)$	$\theta(1)$
Сортировка Шелла по Хиббарду	$\theta(n\log n)$	$\theta\left(n^{\frac{5}{4}}\right)$	$\theta\left(n^{\frac{3}{2}}\right)$	$\theta(1)$	$\theta(1)$	$\theta(1)$
Сортировка Шелла по Пратту	$\theta(n)$	$\theta(n\,(\log n)^2)$	$\theta(n\,(\log n)^2)$	$\theta(n\log n)$	$\theta(n\log n)$	$\theta(n\log n)$
Быстрая сортировка	$\theta(n\log n)$	$\theta(n\log n)$	$\theta(n^2)$	$\theta(n\log n)$	$\theta(n\log n)$	$\theta(n)$
Пирамидальная сортировка	$\theta(n\log n)$	$\theta(n\log n)$	$\theta(n\log n)$	$\theta(1)$	$\theta(1)$	$\theta(1)$

Сортировка выбором.

Суть алгоритма заключается в том, чтобы в каждой итерации цикла по элементам массива, находить минимальный элемент в неотсортированной части и производить обмен значениями с текущим элементом массива. Сортировка является неустойчивой.

Функция временной сложности и её асимптотическая оценка:

Худший случай:

$$T_{SS}^{w}(n) = 1 + 2n + n(3 + 5(n-1) + 3) = 5n^2 + 3n + 1 = \ \theta(n^2)$$

Лучший случай:

$$T_{SS}^{b}(n) = 1 + 2n + n(2 + 3(n-1) + 3) = 3n^2 + 4n + 1 = \theta(n^2)$$

Средний случай:

$$T_{SS}^{a} = 1 + 2n + n(1 + 2 + 1 + 2n + n + 0.5n(1+1) + 1)$$
$$= 1 + 2n + n + 2n + n + 2n^2 + n^2 + n^2 + n = 4n^2 + 7n + 1$$
$$= \theta(n^2)$$

Функция пространственной сложности и её асимтотическая оценка:

Худший случай:

$$S_{SS}^{w}(n) = 4 + 4 + 4 + 4 + 4 + 4 = 24 = \theta(1)$$

Лучший случай:

$$S_{SS}^{b}(n) = 4 + 4 + 4 + 4 + 4 + 4 = 24 = \theta(1)$$

Средний случай:

$$S_{SS}^{a}(n) = 4 + 4 + 4 + 4 + 4 + 4 = 24 = \theta(1)$$

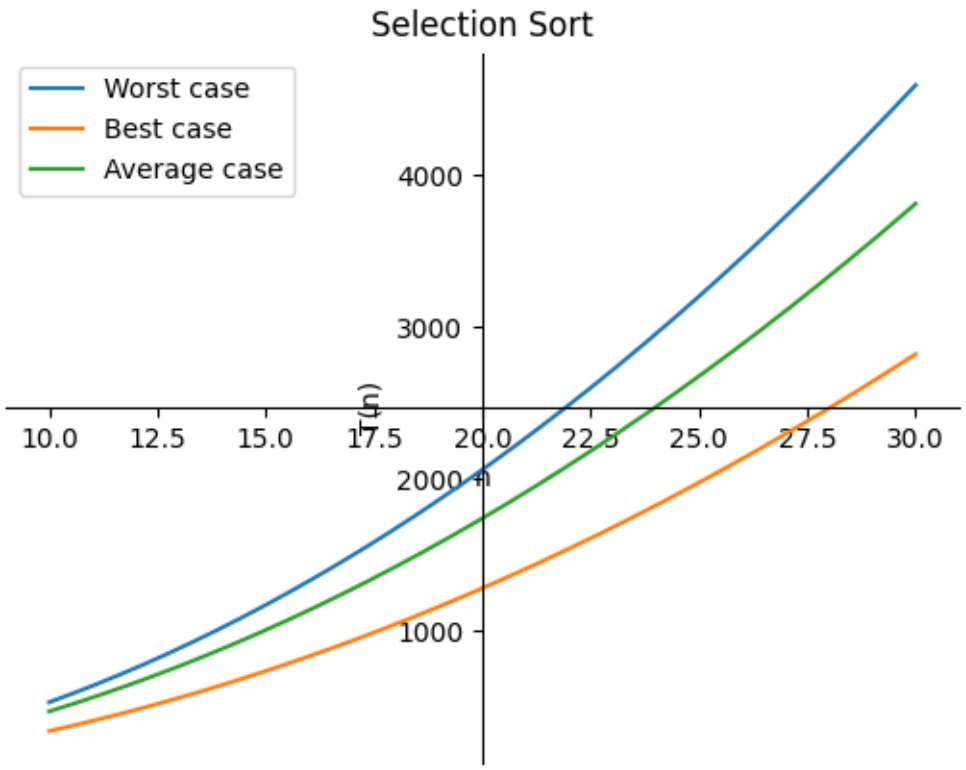

Рисунок 1 – График функции временной сложности сортировки выбором

Сортировка вставками.

Алгоритм делит массив на отсортированную и неотсортированную части. Начиная с первого элемента, и по мере перебора элементов из неотсортированной части, они вставляются в правильное место в отсортированной части. Сортировка вставками является устойчивой.

Функция временной сложности и её асимптотическая оценка:

Худший случай:

$$T_{IS}^{w}(n) = 1 + 2n + n(1 + 1 + 3n + 2n) = 3n^2 + 7n + 1 = \ \theta(n^2)$$

Лучший случай:

$$T_{IS}^{b}(n) = 1 + 2n + n + 3n = 6n + 1 = \ \theta(n)$$

Средний случай:

$$T_{IS}^{a}(n) = 1 + 2n + n + n + n\big(0.5n(1 + 1) + 0.5n(1 + 1 + 2)\big)$$
$$= 1 + 4n + n(n + 2n) = 3n^2 + 4n + 1 = \ \theta(n^2)$$

Функция пространственной сложности и её асимтотическая оценка:

Худший случай:

$$S_{IS}^{w}(n) = 4 + 4 + 4 = 12 = \ \theta(1)$$

Лучший случай:

$$S_{IS}^{b}(n) = 4 + 4 + 4 = 12 = \ \theta(1)$$

Средний случай:

$$S_{IS}^{a}(n) = 4 + 4 + 4 = 12 = \theta(1)$$

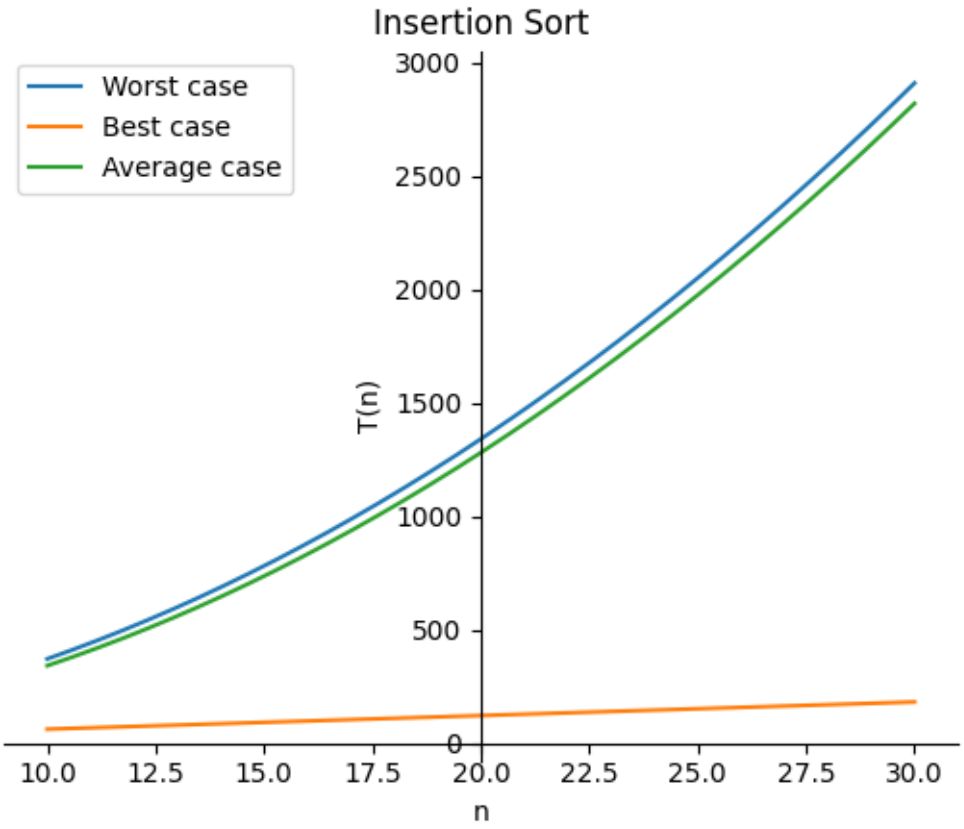

Рисунок 2 – График функции временной сложности сортировки вставками

Сортировка пузырьком.

Алгоритм проходит по массиву несколько раз, сравнивая соседние элементы и меняя их местами, если они расположены в неправильном порядке. Процесс повторяется, пока не будет сделан полный проход без перестановок, что означает, что массив отсортирован. Сортировка пузырьком является устойчивой.

Функция временной сложности и её асимптотическая оценка:

Худший случай:

$$T_{BS}^{w}(n) = 1 + 1 + 3(n-1) + (n-1)\big(1 + 2n(n-1) + 5(n-1)\big)$$
$$= 2 + 3n - 3 + n - 1 + 2(n-1)^2 + 5(n-1)^2 = 7n^2 - 10n + 5$$
$$= \theta(n^2)$$

Лучший случай:

$$T_{BS}^{b}(n) = 1 + 3 + 1 + 1 + 2(n-1) + n - 1 = 3n + 3 = \theta(n)$$

Средний случай:

$$T_{BS}^{a}(n) = 1 + 1 + (n-1)\big(1 + 2 + 1 + 0.5n(2+1) + 0.5n(2+1+4)\big)$$
$$= 2 + (n-1)\left(4 + \frac{3}{2}n + \frac{7}{2}n\right) = 2 + (n-1)(4+5n)$$
$$= 2 + 4n - 4 + 5n^2 - 5n = 5n^2 - n - 2 = \theta(n^2)$$

Функция пространственной сложности и её асимтотическая оценка:

Худший случай:

$S_{BS}^{w}(n) = 1 + 4 + 4 + 4 = 13 = \ \theta(1)$

Лучший случай:

$S_{BS}^{b}(n) = 1 + 4 + 4 = 12 = \ \theta(1)$

Средний случай:

$S_{BS}^{a}(n) = 1 + 4 + 4 + 4 = 13 = \theta(1)$

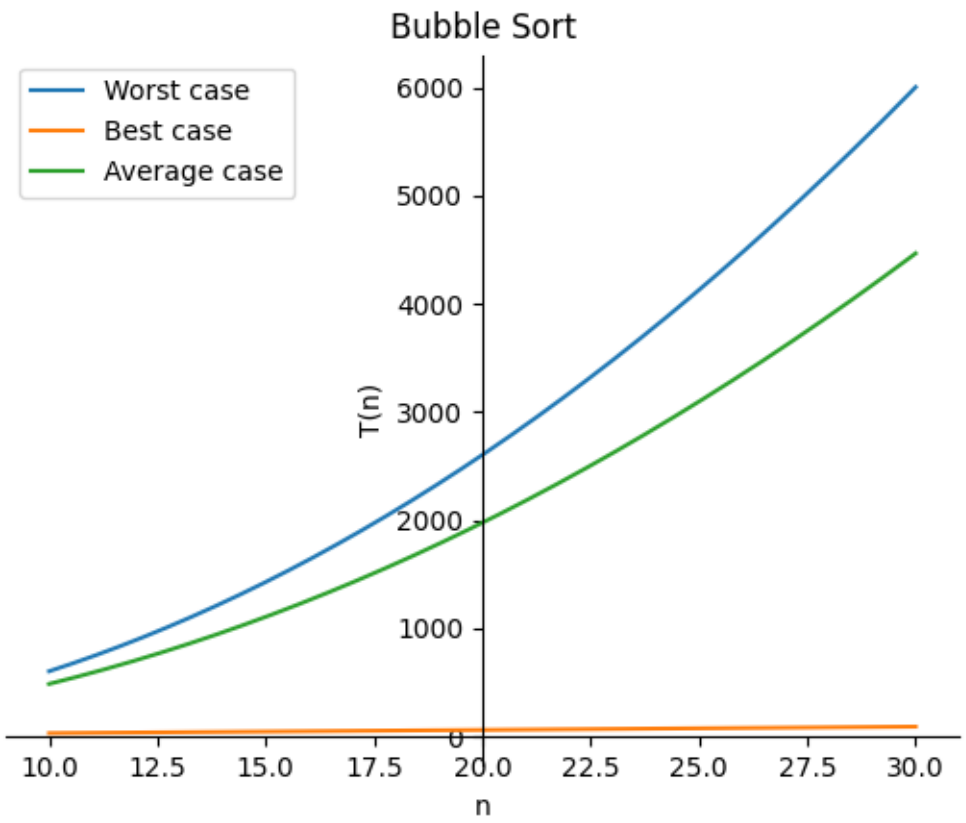

Рисунок 3 – График функции временной сложности сортировки пузырьком

Сортировка слиянием.

Алгоритм делит массив на две половины, рекурсивно сортирует каждую половину, а затем объединяет их обратно в отсортированный массив. Сортировка слиянием является устойчивой.

Функция временной сложности и её асимптотическая оценка:

Для нахождения функции временной сложности, используется мастер теорема:

Худший случай:

$$T_{MS}^{w}(n) = 2 \cdot T_{MS}^{w}\left(\frac{n}{2}\right) + f(n)$$

Найдем $f(n)$ – функцию временной сложности слияния подмассивов:

$$f(n) = 1 + 3 + 2n + n + 3n + 1 = 6n + 5 = \theta(n)$$

Тогда,

$$\begin{aligned} T_{MS}^{w} &= 2 \cdot T_{MS}^{w}\left(\frac{n}{2}\right) + 6n + 5 = 2 \cdot \left(2 \cdot T_{MS}^{w}\left(\frac{n}{4}\right) + \frac{6n}{2} + 5\right) \\ &= 2 \cdot \left(2 \cdot \left(2 \cdot T_{MS}^{w}\left(\frac{n}{4}\right) + \frac{6n}{4} + 5\right)\right) = \cdots \\ &= 2^{k} \cdot T_{MS}^{w}\left(\frac{n}{2^{k}}\right) + k \cdot (6n + 5) \end{aligned}$$

Когда $\frac{n}{2^k} = 1$ рекурсия прекращается, поэтому $k = \log_2 n$ и на этом уровне $T(1) = 2 = \theta(1)$, подставим k:

$$\begin{aligned} T_{MS}^{w}(n) &= n \cdot T_{MS}^{w}(1) + \log_2 n \cdot (6n + 5) = 2n + \log_2 n \cdot (6n + 5) \\ &= \theta(n \log n) \end{aligned}$$

Лучший случай:

Аналогично:

$$T_{MS}^{b}(n) = 2n + \log_2 n \cdot (6n + 5) = \theta(n \log n)$$

Средний случай:

$$T_{MS}^{a}(n) = 2n + \log_2 n \cdot (6n + 5) = \theta(n \log n)$$

Функция пространственной сложности и её асимтотическая оценка:

Худший случай:

$$\begin{aligned} S_{MS}^{w}(n) &= S_{MS}^{w}\left(\frac{n}{2}\right) + 2n + 1 = S_{MS}^{w}\left(\frac{n}{4}\right) + \frac{2n}{2} + 1 + 2n + 1 \\ &= S_{MS}^{w}\left(\frac{n}{8}\right) + \frac{2n}{4} + 1 + \frac{2n}{2} + 1 + 2n + 1 \\ &= 2n \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) + c = \theta(n) \end{aligned}$$

Лучший случай:

Аналогично:

$S_{MS}^{b}(n) = \theta(n)$

Средний случай:

$S_{MS}^{a}(n) = \theta(n)$

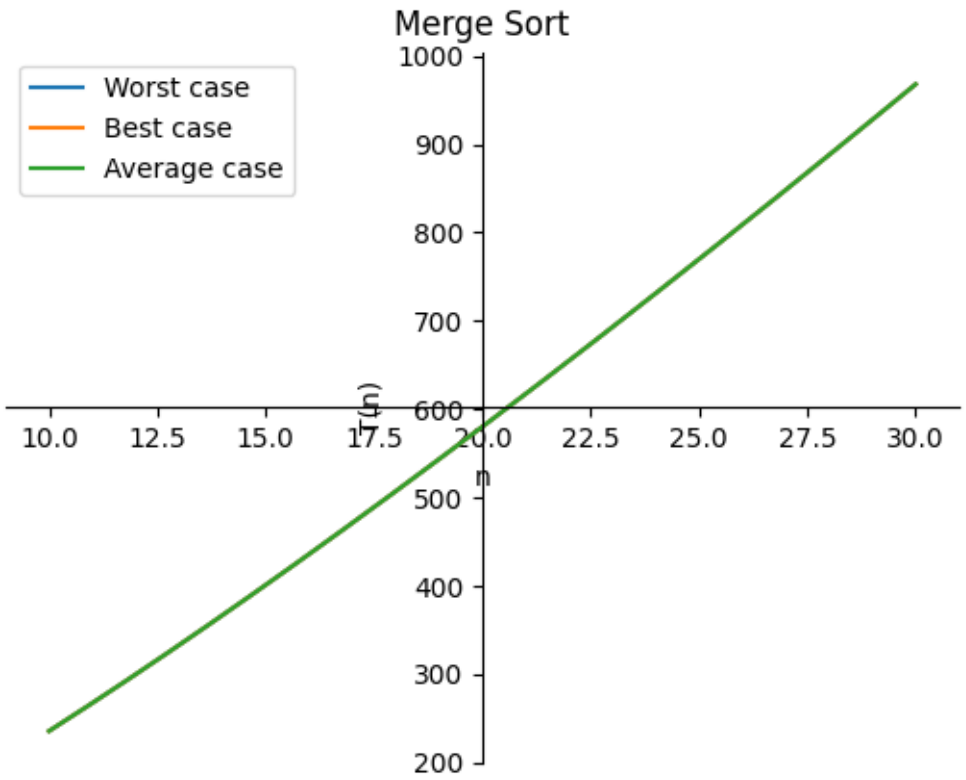

Рисунок 4 – График функции временной сложности сортировки слиянием

Сортировка Шелла.

Это обобщение сортировки вставками. Элементы сравниваются и сортируются на заданном расстоянии (или шаге), который с каждой итерацией уменьшается в 2 раза. Сортировка является неустойчивой.

Функция временной сложности и её асимптотическая оценка:

Худший случай:

$T_{SHS}^{w}(n) = \theta(n^2)$

Лучший случай:

$T^b_{SHS}(n) = \theta\left(n^{\frac{3}{2}}\right)$

Средний случай:

$T^a_{SHS} = \theta(n^{1.667})$

Функция пространственной сложности и её асимтотическая оценка:

Худший случай:

$S^w_{SHS}(n) = \theta(1)$

Лучший случай:

$S^b_{SHS}(n) = \theta(1)$

Средний случай:

$S^a_{SHS}(n) = \theta(1)$

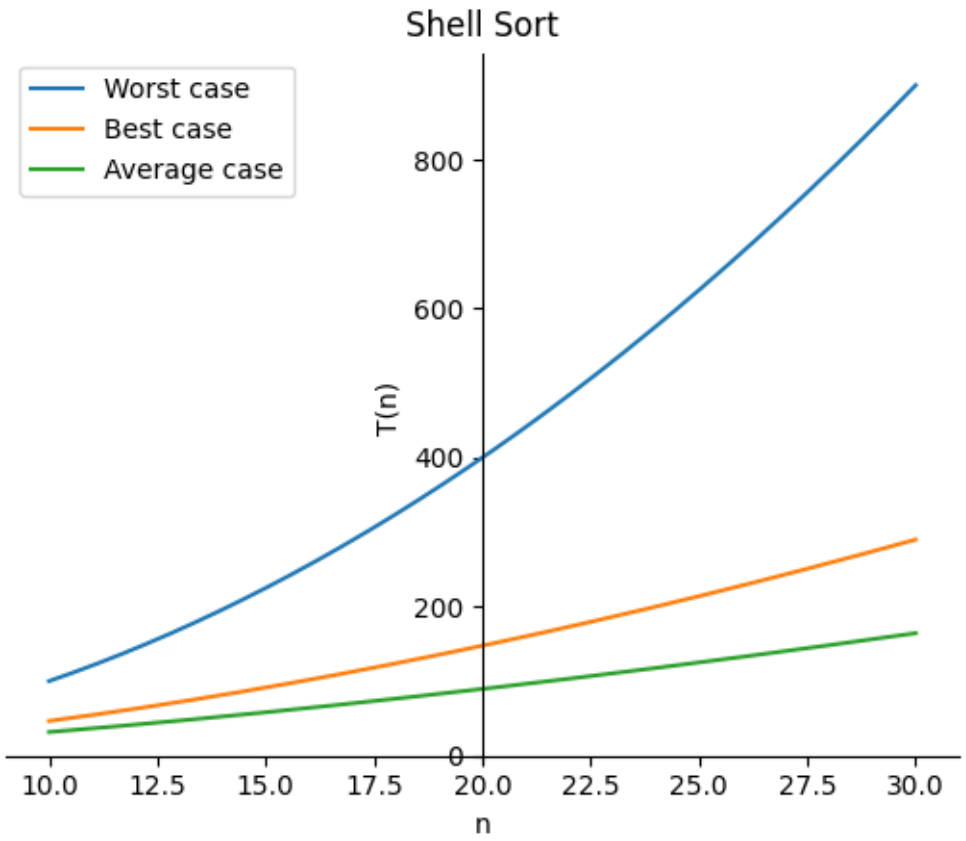

Рисунок 5 – График функции временной сложности сортировки Шелла

Сортировка Шелла (последовательность Хиббарда).

Вариант сортировки Шелла, где используется последовательность шагов $h_k = 2^k - 1$. Шаги уменьшаются по этой последовательности. Сортировка является неустойчивой.

Функция временной сложности и её асимптотическая оценка:

Худший случай:

$$T^w_{SHHS}(n) = \theta\left(n^{\frac{3}{2}}\right)$$

Лучший случай:

$$T^b_{SHHS}(n) = \theta(n \log n)$$

Средний случай:

$$T^a_{SHHS} = \theta\left(n^{\frac{5}{4}}\right)$$

Функция пространственной сложности и её асимтотическая оценка:

Худший случай:

$$S^w_{SHHS}(n) = \theta(1)$$

Лучший случай:

$$S^b_{SHHS}(n) = \theta(1)$$

Средний случай:

$$S^a_{SHHS}(n) = \theta(1)$$

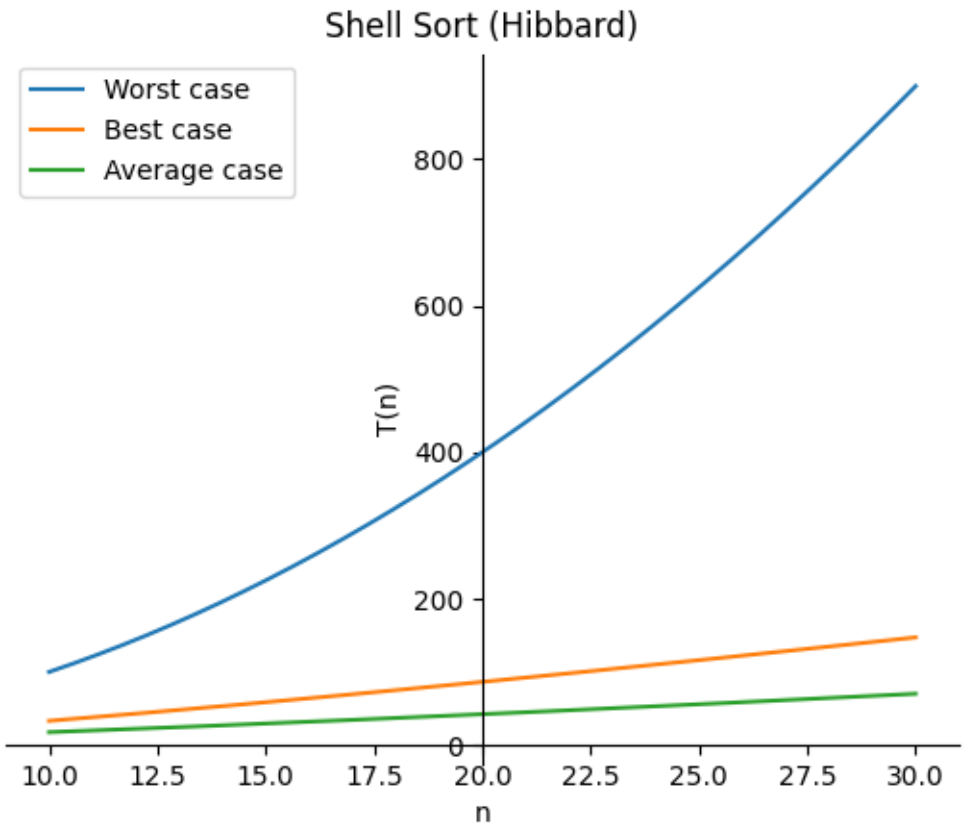

Рисунок 6 – График функции временной сложности сортировки Шелла (последовательность Хиббарда)

Сортировка Шелла (последовательность Пратта).

Вариант сортировки Шелла, где для шагов используется последовательность чисел, которая получается как комбинация степеней двойки и тройки: $h_k = 2^i \cdot 3^j$, где i и j — неотрицательные целые числа. Сортировка является неустойчивой.

Функция временной сложности и её асимптотическая оценка:

Худший случай:

$T^w_{SHPS}(n) = \theta(n\,(\log n)^2)$

Лучший случай:

$T^b_{SHPS}(n) = \ \theta(n)$

Средний случай:

$T^a_{SHPS} = \ \theta(n\,(\log n)^2)$

Функция пространственной сложности и её асимтотическая оценка:

Худший случай:

$S^{w}_{SHPS}(n) = \theta(n \log n)$

Лучший случай:

$S^{b}_{SHPS}(n) = \theta(n \log n)$

Средний случай:

$S^{a}_{SHPS}(n) = \theta(n \log n)$

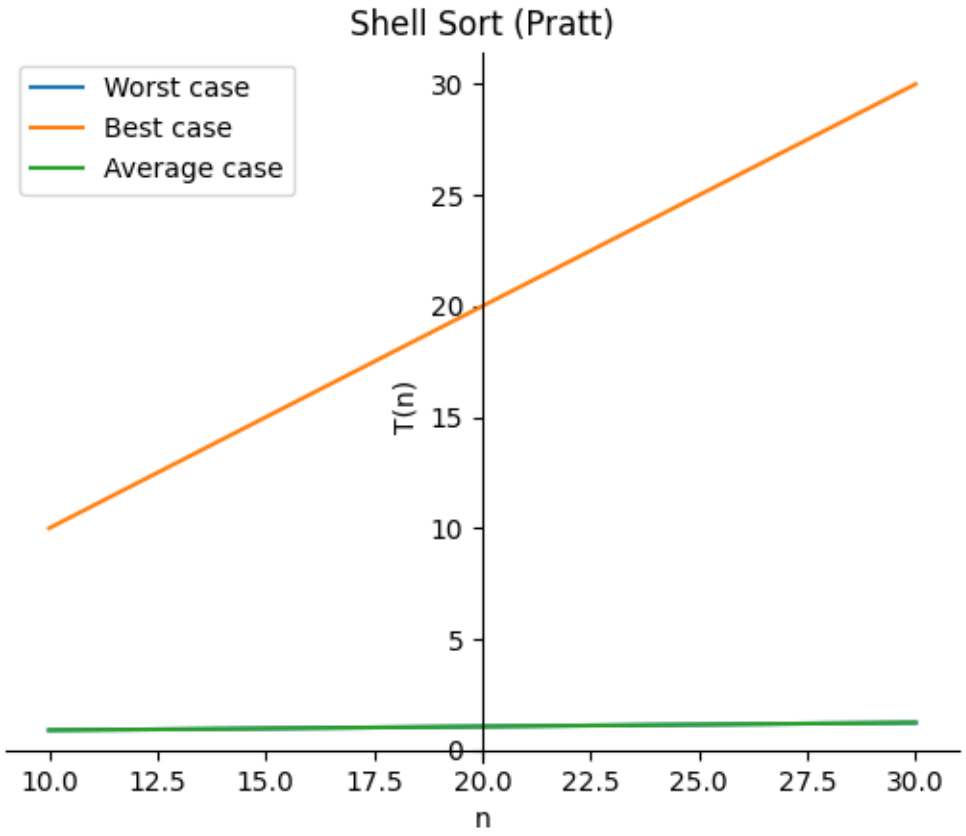

Рисунок 7 – График функции временной сложности сортировки Шелла(последовательность Пратта)

Быстрая сортировка.

Алгоритм выбирает опорный элемент и разделяет массив на элементы меньше и больше опорного, затем рекурсивно сортирует обе части. Сортировка является неустойчивой.

Функция временной сложности и её асимптотическая оценка:

Худший случай:

$$T_{QS}^{w}(n) = T_{QS}^{w}(n-1) + 2n - 1 = T_{QS}^{w}(n-2) + 2(n-1) - 1$$
$$= T_{QS}^{w}(n-3) + 2(n-2) - 1 =$$

$$= T_{QS}^{w}(1) + \sum_{i=2}^{n}(2i-1)$$
$$= 1 + \sum_{i=2}^{n}(2i-1) = 1 + 2\cdot 2 - 1 + 2\cdot 3 - 1 + 2\cdot 4 - 1 + \cdots$$
$$+ (2n-1) = 1 + 3 + 5 + 7 + \cdots + (2n-1)$$
$$= 1 + \frac{3 + 2n - 1}{2}(n-1) = 1 + (1+n)(n-1) =$$
$$= 1 + n + n^2 - 1 + n = n^2 + 2n = \theta(n^2)$$

Лучший случай:

$$T_{QS}^{b}(n) = 2\cdot T_{QS}^{b}\left(\frac{n}{2}\right) + 2n - 1 = 2^k \cdot T_{QS}^{b}\left(\frac{n}{2^k}\right) + k\cdot(2n-1)$$
$$= n\cdot T_{QS}^{b}(1) + \log_2 n\cdot(2n-1)$$
$$= n + 2n\log_2 n - \log_2 n = \theta(n\log n)$$

Средний случай:

$$T_{QS}^{a}(n) = T_{QS}^{a}\left(\frac{n}{3}\right) + T_{QS}^{a}\left(\frac{2n}{3}\right) + 2n - 1$$
$$= \left(T_{QS}^{a}\left(\frac{n}{9}\right) + T_{QS}^{a}\left(\frac{2n}{9}\right) + \frac{2n}{3} - 1\right) + (T_{QS}^{a}\left(\frac{2n}{9}\right) + T_{QS}^{a}\left(\frac{4n}{9}\right) + \frac{4n}{3}$$
$$- 1)$$
$$= T_{QS}^{a}\left(\frac{n}{9}\right) + T_{QS}^{a}\left(\frac{2n}{9}\right) + T_{QS}^{a}\left(\frac{2n}{9}\right) + T_{QS}^{a}\left(\frac{4n}{9}\right) + \frac{2n}{3} - 1 + \frac{4n}{3} - 1$$
$$= \cdots = T_{QS}^{a}\left(\frac{n}{3^k}\right) + T_{QS}^{a}\left(\frac{2n}{3^k}\right) + \sum_{i=1}^{\log_3 n - 1}\frac{2n}{3^i} + n\log_3 n$$
$$= 2 + \sum_{i=1}^{\log_3 n - 1}\frac{2n}{3^i} + n\log_3 n = 2 + 2n\frac{1 - \left(\frac{1}{3}\right)^{\log_3 n}}{1 - \frac{1}{3}} + n\log_3 n$$
$$= 2 + 2n\frac{1 - \frac{1}{n}}{\frac{2}{3}} + n\log_3 n = 2 + 3n + 3n\log_3 n = \theta(n\log n)$$

Функция пространственной сложности и её асимтотическая оценка:

Худший случай:

$S_{QS}^{w}(n) = \theta(n)$

Лучший случай:

$S_{QS}^{b}(n) = \theta(n \log n)$

Средний случай:

$S_{QS}^{a}(n) = \theta(n \log n)$

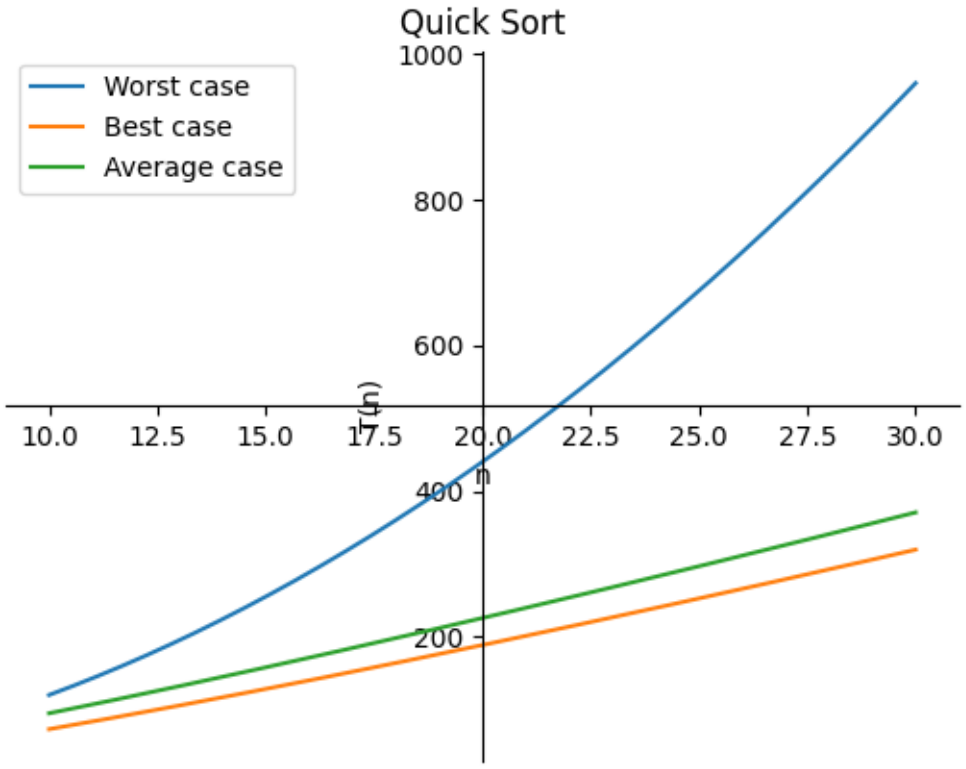

Рисунок 8 – График функции временной сложности быстрой сортировки

Пирамидальная сортировка.

Сначала строится структура данных в виде пирамиды (кучи), затем на каждом шаге извлекается максимальный элемент и восстанавливается структура кучи. Сортировка является неустойчивой.

Функция временной сложности и её асимптотическая оценка:

Худший случай:

$T_{HS}^{w}(n) = \theta(n \log n)$

Лучший случай:

$T_{HS}^{b}(n) = \theta(n \log n)$

Средний случай:

$T_{HS}^{a} = \theta(n \log n)$

Функция пространственной сложности и её асимтотическая оценка:

Худший случай:

$S_{HS}^{w}(n) = \theta(1)$

Лучший случай:

$S_{HS}^{b}(n) = \theta(1)$

Средний случай:

$S_{HS}^{a}(n) = \theta(1)$

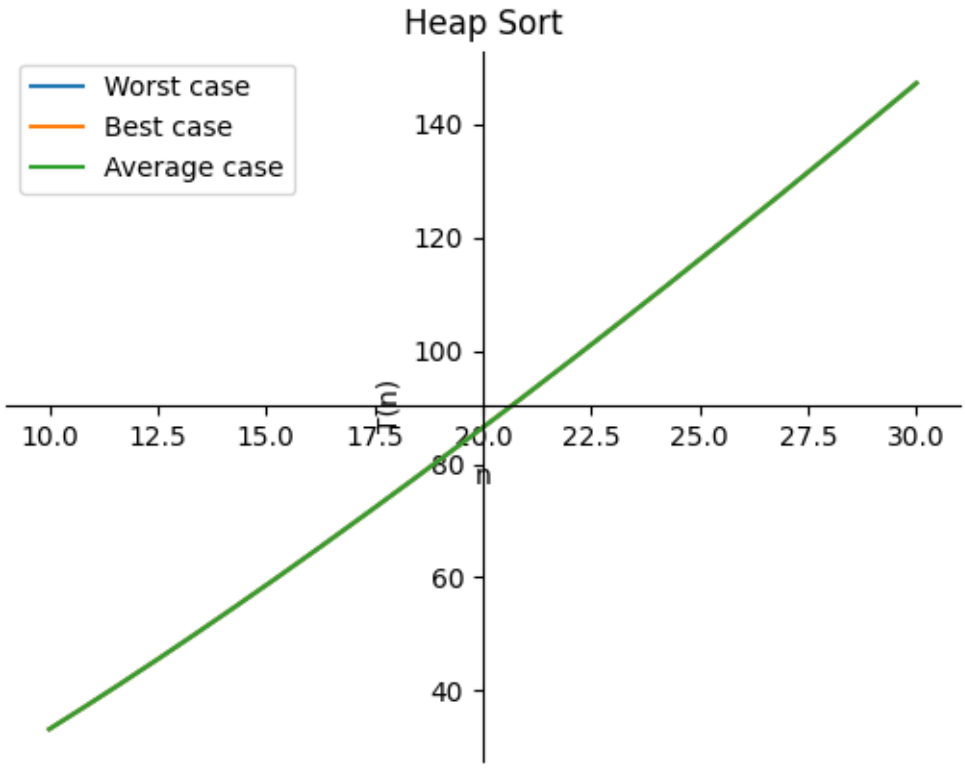

Рисунок 9 – График функции временной сложности пирамидальной сортировки

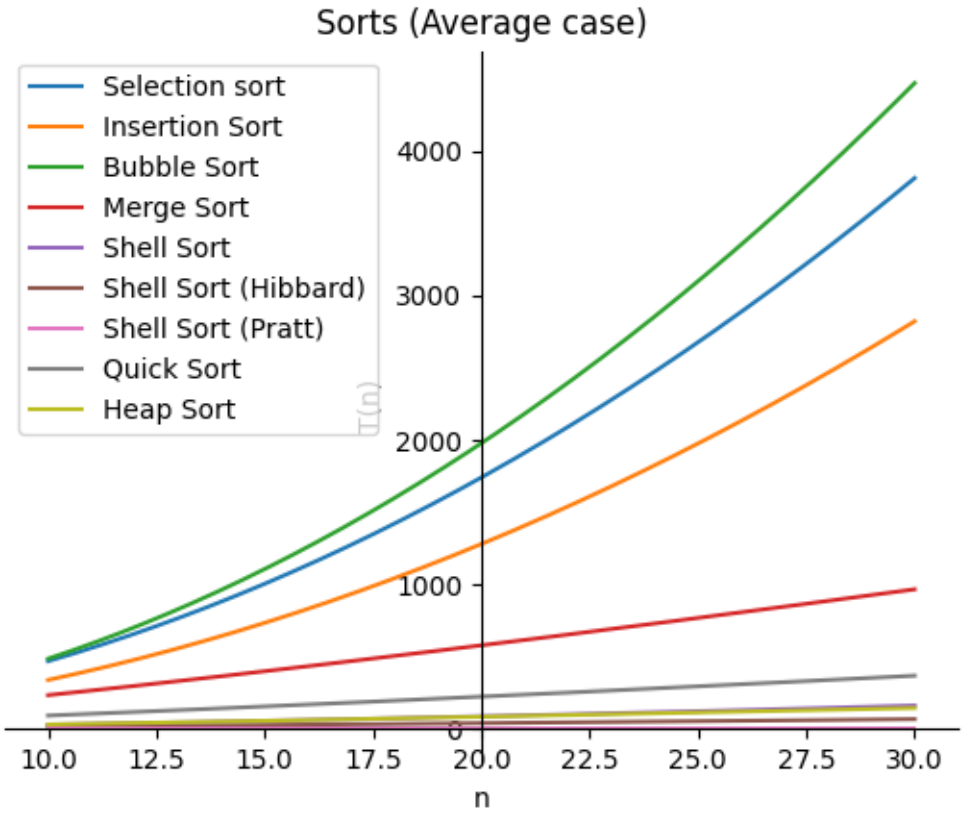

Рисунок 10 – Графики функции временной сложности всех сортировок в среднем случае

Исходя из теоретических данных, можно предположить что самой быстрой сортировой в среднем случае при размере массива > 100 000 будет сортировка Шелла с последовательностью Пратта.

Практическая часть.

Для выполнения практической части использовался язык C++, где были написаны все сортировки и функции заполнения, а также язык Python для отрисовки графиков. Для каждой сортировки и для каждого заполнения считалось время работы функции на определенном интервале элементов, результаты были представлены в таблице и на графиках.

Результаты эксперимента:

Сортировка выбором.

Отсортированный массив

Количество элементов	Время сортировки(сек)
20000	2.66567
30000	4.66885
40000	6.39783
50000	8.94413
60000	11.4693
70000	15.6198
80000	20.381
90000	25.8344
100000	31.8396
110000	38.7964

Таблица 1 – Время работы сортировки выбором (отсортированный массив)

Почти отсортированный массив

Количество элементов	Время сортировки(сек)
20000	1.2919
30000	2.91443
40000	5.21007
50000	8.15979
60000	11.7337
70000	15.9747
80000	20.8411
90000	26.414
100000	32.608
110000	39.4108

Таблица 2 – Время работы сортировки выбором (почти отсортированный массив)

Отсортированный в сторону убывания массив

Количество элементов	Время сортировки(сек)
20000	1.62244
30000	3.66446
40000	6.5082
50000	10.1608
60000	14.6623
70000	19.9365
80000	26.0528

90000	33.2684
100000	40.6719
110000	49.1756

Таблица 3 – Время работы сортировки выбором (массив отсортированный в сторону убывания)

Случайный массив

Количество элементов	Время сортировки(сек)
20000	1.26856
30000	2.87571
40000	5.11884
50000	7.98469
60000	11.4606
70000	15.633
80000	20.4218
90000	25.8532
100000	31.8553
110000	38.6172

Таблица 4 – Время работы сортировки выбором (случайный массив)

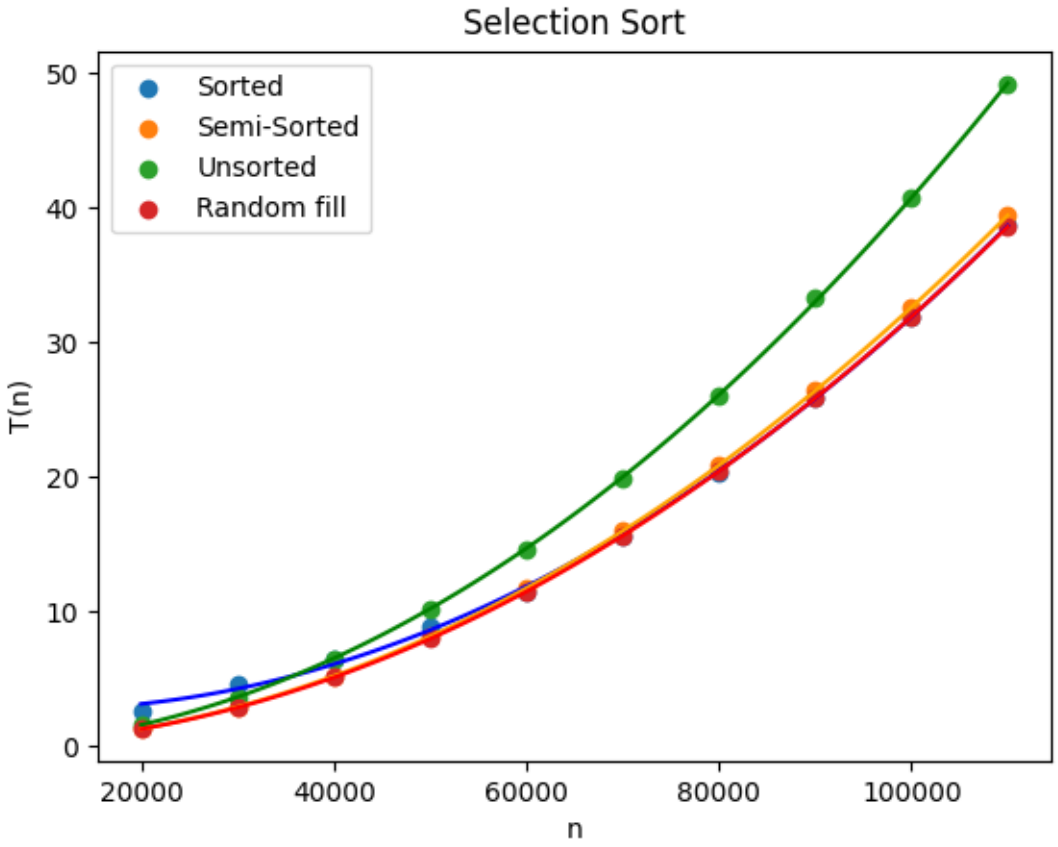

Рисунок 11 – График сортировки выбором для всех перечисленных случаев

Сортировка вставками.

Отсортированный массив

Количество элементов	Время сортировки(сек)
20000	0.0002023
30000	0.0002938
40000	0.0003852
50000	0.0005148
60000	0.0006166
70000	0.0006631
80000	0.0008135
90000	0.0008764
100000	0.0009646
110000	0.0010791

Таблица 5 – Время работы сортировки вставками (отсортированный массив)

Почти отсортированный массив

Количество элементов	Время сортировки(сек)
20000	0.496464
30000	1.12121

40000	1.99586
50000	3.10445
60000	4.46802
70000	6.08987
80000	7.95875
90000	10.0981
100000	12.4482
110000	14.9828

Таблица 6 – Время работы сортировки вставками (почти отсортированный массив)

Отсортированный в сторону убывания массив

Количество элементов	Время сортировки(сек)
20000	2.77427
30000	6.21777
40000	11.0594
50000	17.3814
60000	24.8324
70000	33.8663
80000	44.2537
90000	55.8205
100000	68.9507
110000	83.4295

Таблица 7 – Время работы сортировки вставками (массив отсортированный в сторону убывания)

Случайный массив

Количество элементов	Время сортировки(сек)
20000	1.36942
30000	3.10041
40000	5.54315
50000	8.62329
60000	12.4247
70000	16.9918
80000	22.2321
90000	28.0488
100000	34.6441
110000	41.7512

Таблица 8 – Время работы сортировки вставками (случайный массив)

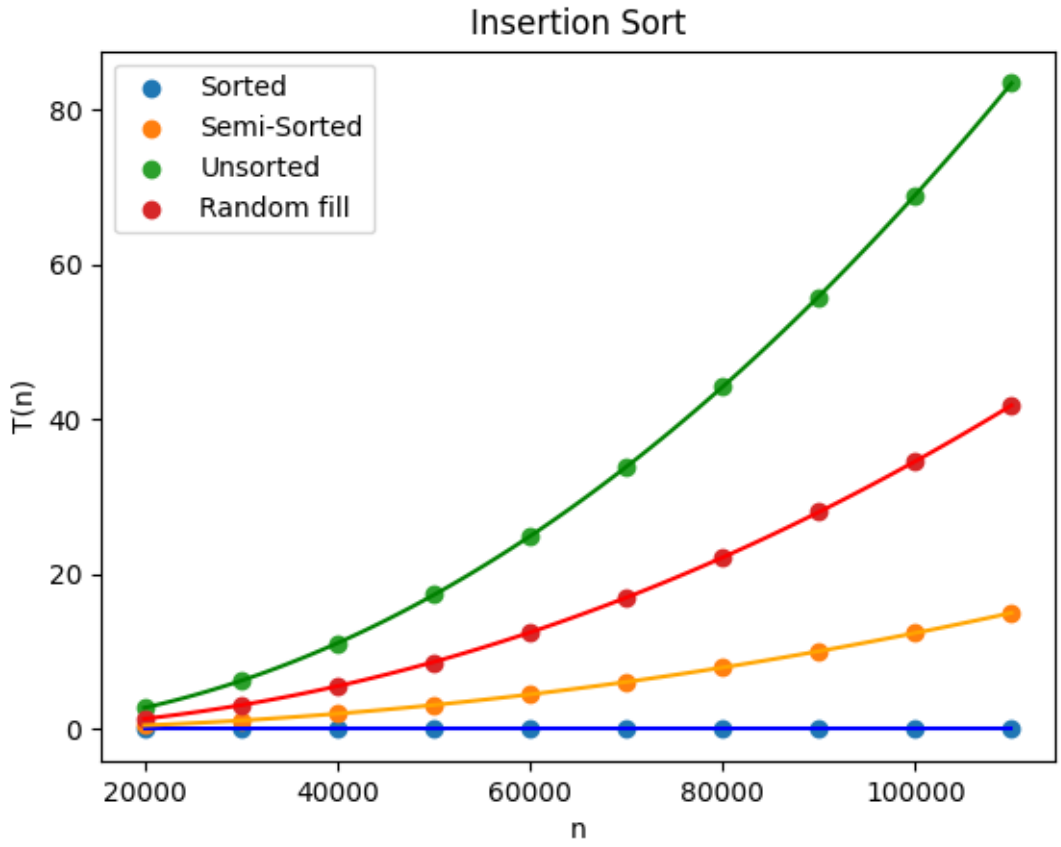

Рисунок 12 – График сортировки вставками для всех перечисленных случаев

Сортировка пузырьком.

Отсортированный массив

Количество элементов	Время сортировки(сек)
20000	0.0001824
30000	0.0002716
40000	0.0003632
50000	0.0004531
60000	0.0005587
70000	0.0006304
80000	0.0007265
90000	0.0008174
100000	0.0009131
110000	0.0010541

Таблица 9 – Время работы сортировки пузырьком (отсортированный массив)

Почти отсортированный массив

Количество элементов	Время сортировки(сек)
20000	2.42518
30000	5.47569
40000	9.69952
50000	15.1398

60000	21.9849
70000	29.7326
80000	38.733
90000	49.2102
100000	60.6163
110000	73.242

Таблица 10 – Время работы сортировки пузырьком (почти отсортированный массив)

Отсортированный в сторону убывания массив

Количество элементов	Время сортировки(сек)
20000	4.73915
30000	10.6544
40000	18.9611
50000	29.8437
60000	42.8215
70000	58.3135
80000	76.1058
90000	96.4762
100000	118.597
110000	143.577

Таблица 11 – Время работы сортировки пузырьком (массив отсортированный в сторону убывания)

Случайный массив

Количество элементов	Время сортировки(сек)
20000	3.89553
30000	8.7496
40000	15.5626
50000	24.3244
60000	34.961
70000	47.7517
80000	62.5312
90000	78.6228
100000	97.0987
110000	117.903

Таблица 12 – Время работы сортировки пузырьком (случайный массив)

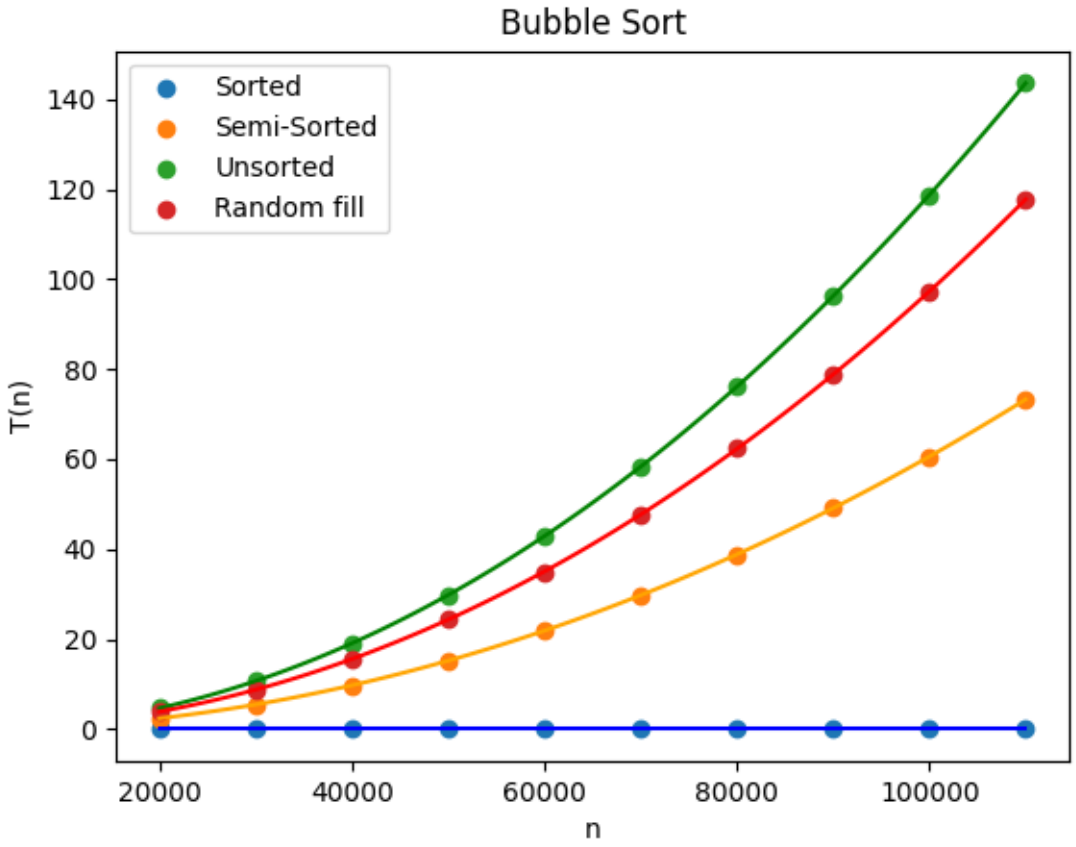

Рисунок 13 – График сортировки пузырьком для всех перечисленных случаев

Сортировка слиянием.

Отсортированный массив

Количество элементов	Время сортировки(сек)
20000	0.119119
30000	0.181034
40000	0.241704
50000	0.302832
60000	0.367303
70000	0.419884
80000	0.486093
90000	0.544194
100000	0.609719
110000	0.670313

Таблица 13 – Время работы сортировки слиянием (отсортированный массив)

Почти отсортированный массив

Количество элементов	Время сортировки(сек)
20000	0.119785
30000	0.184277
40000	0.238275
50000	0.303816

60000	0.362658
70000	0.423836
80000	0.488236
90000	0.549497
100000	0.610446
110000	0.664803

Таблица 14 – Время работы сортировки слиянием (почти отсортированный массив)

Отсортированный в сторону убывания массив

Количество элементов	Время сортировки(сек)
20000	0.11924
30000	0.181112
40000	0.239262
50000	0.300966
60000	0.36414
70000	0.423604
80000	0.491852
90000	0.552898
100000	0.611672
110000	0.679772

Таблица 15 – Время работы сортировки слиянием (массив отсортированный в сторону убывания)

Случайный массив

Количество элементов	Время сортировки(сек)
20000	0.120628
30000	0.184345
40000	0.243965
50000	0.30647
60000	0.369017
70000	0.430038
80000	0.500635
90000	0.565148
100000	0.625679
110000	0.686129

Таблица 16 – Время работы сортировки слиянием (случайный массив)

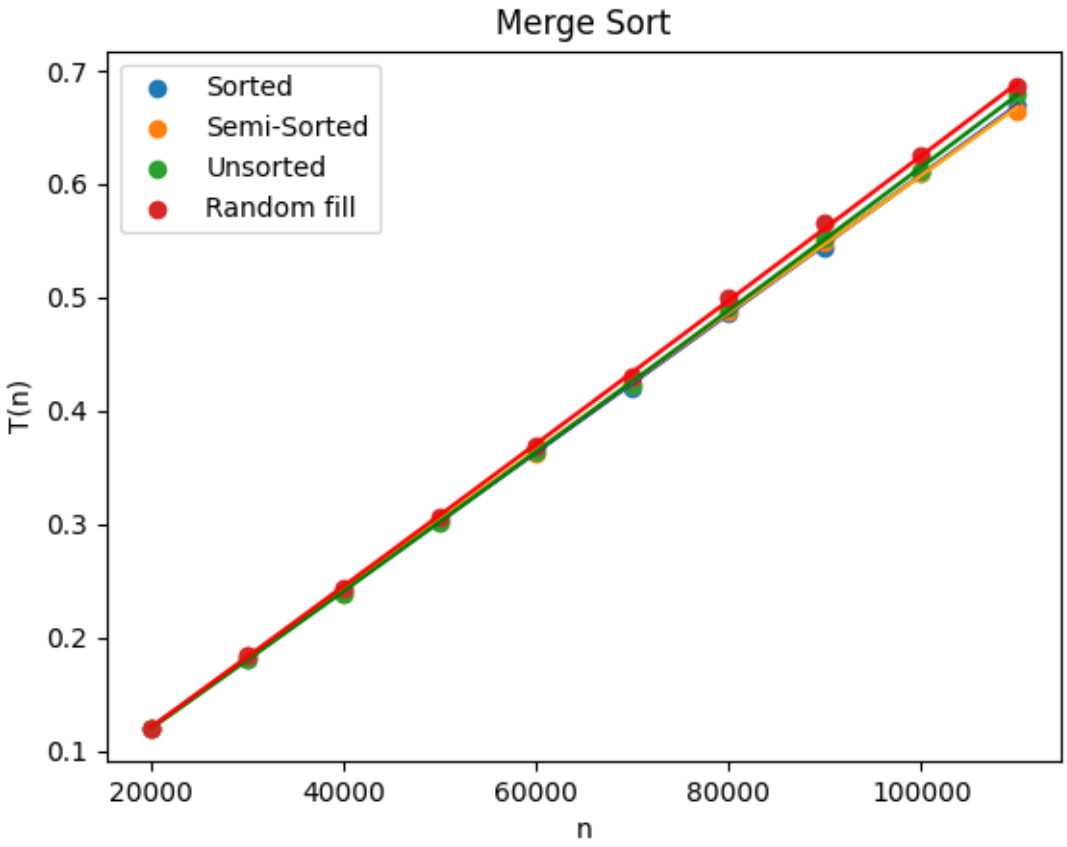

Рисунок 14 – График сортировки слиянием для всех перечисленных случаев

Сортировка Шелла.

Отсортированный массив

Количество элементов	Время сортировки(сек)
20000	0.0025692
30000	0.0040266
40000	0.0058938
50000	0.0070289
60000	0.0099503
70000	0.0109661
80000	0.0122907
90000	0.0141399
100000	0.0165586
110000	0.0173711

Таблица 17 – Время работы сортировки Шелла (отсортированный массив)

Почти отсортированный массив

Количество элементов	Время сортировки(сек)
20000	0.0046973
30000	0.0071715
40000	0.0098534
50000	0.0122118

60000	0.0159593
70000	0.0180952
80000	0.0214234
90000	0.0241003
100000	0.027351
110000	0.0295887

Таблица 18 – Время работы сортировки Шелла (почти отсортированный массив)

Отсортированный в сторону убывания массив

Количество элементов	Время сортировки(сек)
20000	0.0062428
30000	0.0086927
40000	0.0124227
50000	0.0175604
60000	0.0198182
70000	0.0220982
80000	0.0264077
90000	0.0296526
100000	0.0344315
110000	0.0355962

Таблица 19 – Время работы сортировки Шелла (массив отсортированный в сторону убывания)

Случайный массив

Количество элементов	Время сортировки(сек)
20000	0.0133668
30000	0.0196815
40000	0.0292328
50000	0.0364038
60000	0.0448961
70000	0.0542839
80000	0.0696328
90000	0.0741254
100000	0.0842963
110000	0.0930418

Таблица 20 – Время работы сортировки Шелла (случайный массив)

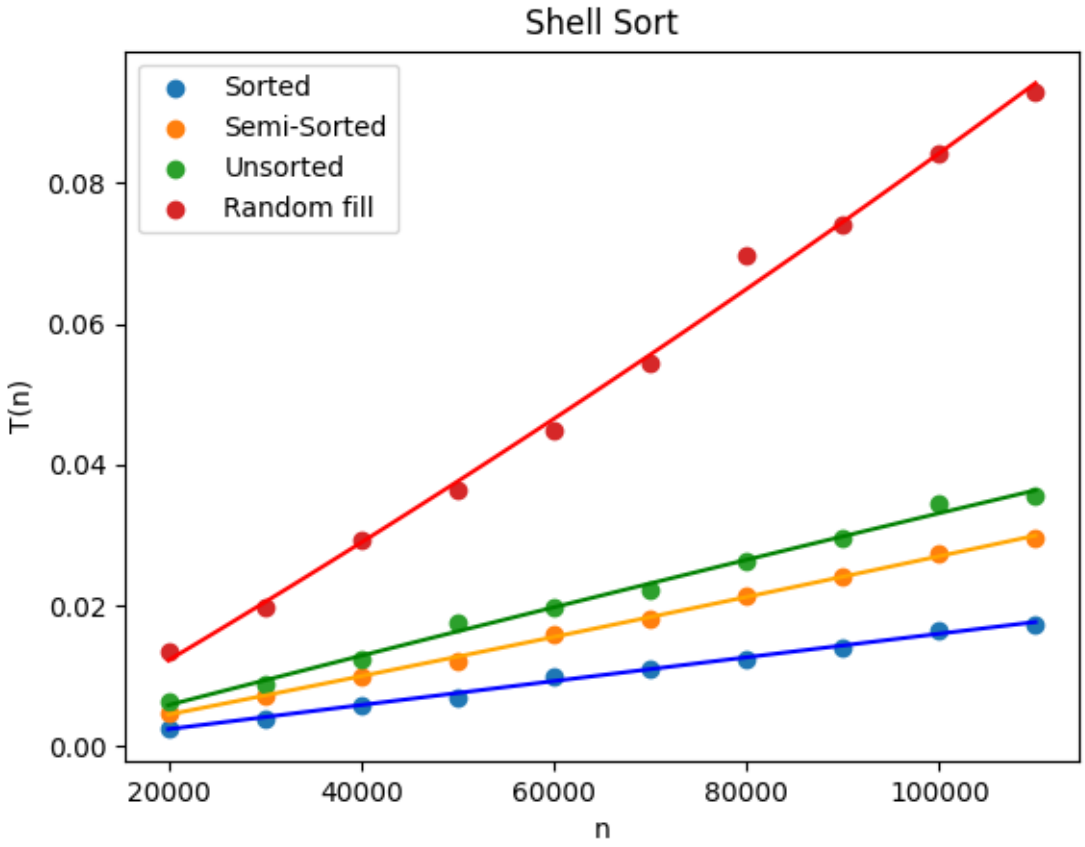

Рисунок 15 – График сортировки Шелла для всех перечисленных случаев

Сортировка Шелла (последовательность Хиббарда).

Отсортированный массив

Количество элементов	Время сортировки(сек)
20000	0.0026784
30000	0.004077
40000	0.0057306
50000	0.00707
60000	0.0097337
70000	0.0118632
80000	0.0125999
90000	0.0140554
100000	0.0153364
110000	0.0172229

Таблица 21 – Время работы сортировки Шелла с последовательностью Хиббарда (отсортированный массив)

Почти отсортированный массив

Количество элементов	Время сортировки(сек)
20000	0.349606
30000	0.77396
40000	1.36914

50000	2.13966
60000	3.08466
70000	4.18427
80000	5.47437
90000	6.91487
100000	8.5398
110000	10.35

Таблица 22 – Время работы сортировки Шелла с последовательностью Хиббарда (почти отсортированный массив)

Отсортированный в сторону убывания массив

Количество элементов	Время сортировки(сек)
20000	1.3813
30000	3.08412
40000	5.48161
50000	8.51795
60000	12.2955
70000	16.794
80000	21.9399
90000	27.7016
100000	34.0814
110000	41.3485

Таблица 23 – Время работы сортировки Шелла с последовательностью Хиббарда (массив отсортированный в сторону убывания)

Случайный массив

Количество элементов	Время сортировки(сек)
20000	0.793968
30000	1.76007
40000	3.14114
50000	4.94465
60000	7.07201
70000	9.64302
80000	12.536
90000	15.8554
100000	19.5989
110000	23.8815

Таблица 24 – Время работы сортировки Шелла с последовательностью Хиббарда (случайный массив)

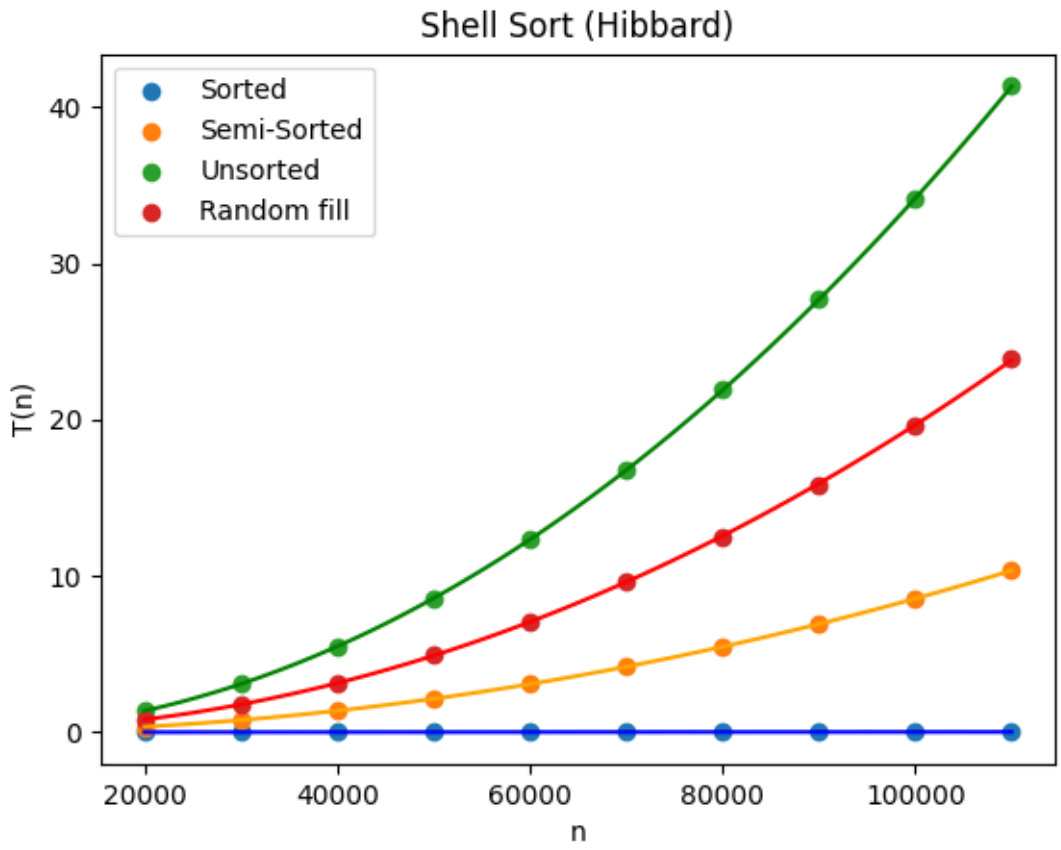

Рисунок 16 – График сортировки Шелла с последовательностью Хиббарда для всех перечисленных случаев

Сортировка Шелла (последовательность Пратта).

Отсортированный массив

Количество элементов	Время сортировки(сек)
20000	0.0147609
30000	0.0240634
40000	0.0338253
50000	0.0449055
60000	0.0549618
70000	0.0665212
80000	0.0777683
90000	0.0898924
100000	0.101281
110000	0.112626

Таблица 25 – Время работы сортировки Шелла с последовательностью Пратта (отсортированный массив)

Почти отсортированный массив

Количество элементов	Время сортировки(сек)
20000	0.401001

30000	0.905097
40000	1.58703
50000	2.46637
60000	3.56336
70000	4.81731
80000	6.30777
90000	7.94743
100000	9.81106
110000	11.8715

Таблица 26 – Время работы сортировки Шелла с последовательностью Пратта (почти отсортированный массив)

Отсортированный в сторону убывания массив

Количество элементов	Время сортировки(сек)
20000	2.17904
30000	4.86932
40000	8.64304
50000	13.5119
60000	19.526
70000	26.5711
80000	34.6417
90000	43.9067
100000	55.3354
110000	65.4773

Таблица 27 – Время работы сортировки Шелла с последовательностью Пратта (массив отсортированный в сторону убывания)

Случайный массив

Количество элементов	Время сортировки(сек)
20000	1.10081
30000	2.46073
40000	4.35058
50000	6.76874
60000	9.75255
70000	13.3869
80000	17.4461
90000	22.0458
100000	27.2873
110000	32.8196

Таблица 28 – Время работы сортировки Шелла с последовательностью Пратта (случайный массив)

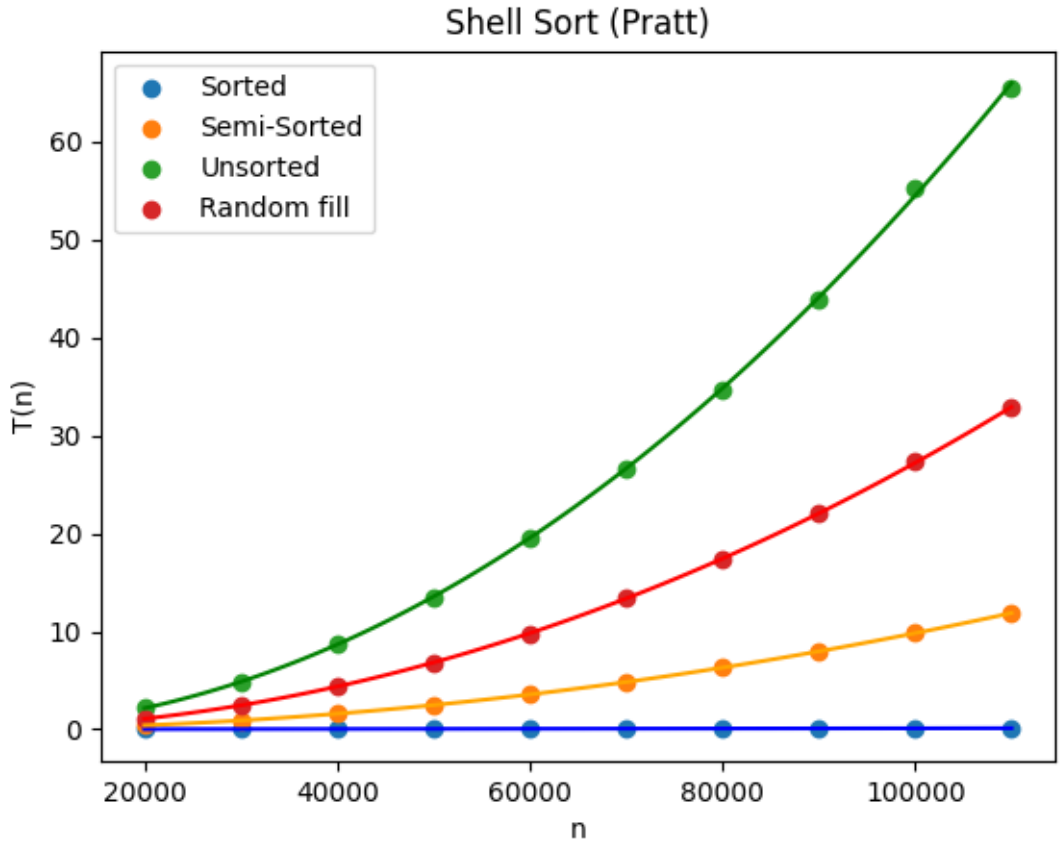

Рисунок 17 – График сортировки Шелла с последовательностью Пратта для всех перечисленных случаев

Быстрая сортировка.

Отсортированный массив

Количество элементов	Время сортировки(сек)
20000	3.66786
30000	8.24835
40000	14.696
50000	22.916
60000	33.0002
70000	44.8831
80000	58.6188
90000	74.3403
100000	91.5042
110000	110.956

Таблица 29 – Время работы быстрой сортировки (отсортированный массив)

Почти отсортированный массив

Количество элементов	Время сортировки(сек)
20000	0.285622
30000	0.68641
40000	1.13633

50000	1.94769
60000	2.72248
70000	3.59794
80000	4.53337
90000	6.35956
100000	7.77131
110000	9.38359

Таблица 30 – Время работы быстрой сортировки (почти отсортированный массив)

Отсортированный в сторону убывания массив

Количество элементов	Время сортировки(сек)
20000	2.23928
30000	5.02467
40000	8.96705
50000	13.982
60000	20.1267
70000	27.4946
80000	35.7306
90000	45.2961
100000	55.7968
110000	67.5421

Таблица 31 – Время работы быстрой сортировки (массив отсортированный в сторону убывания)

Случайный массив

Количество элементов	Время сортировки(сек)
20000	0.0052804
30000	0.0078259
40000	0.0106528
50000	0.0143539
60000	0.0173637
70000	0.0223593
80000	0.0263657
90000	0.0267479
100000	0.0336292
110000	0.0348456

Таблица 32 – Время работы быстрой сортировки (случайный массив)

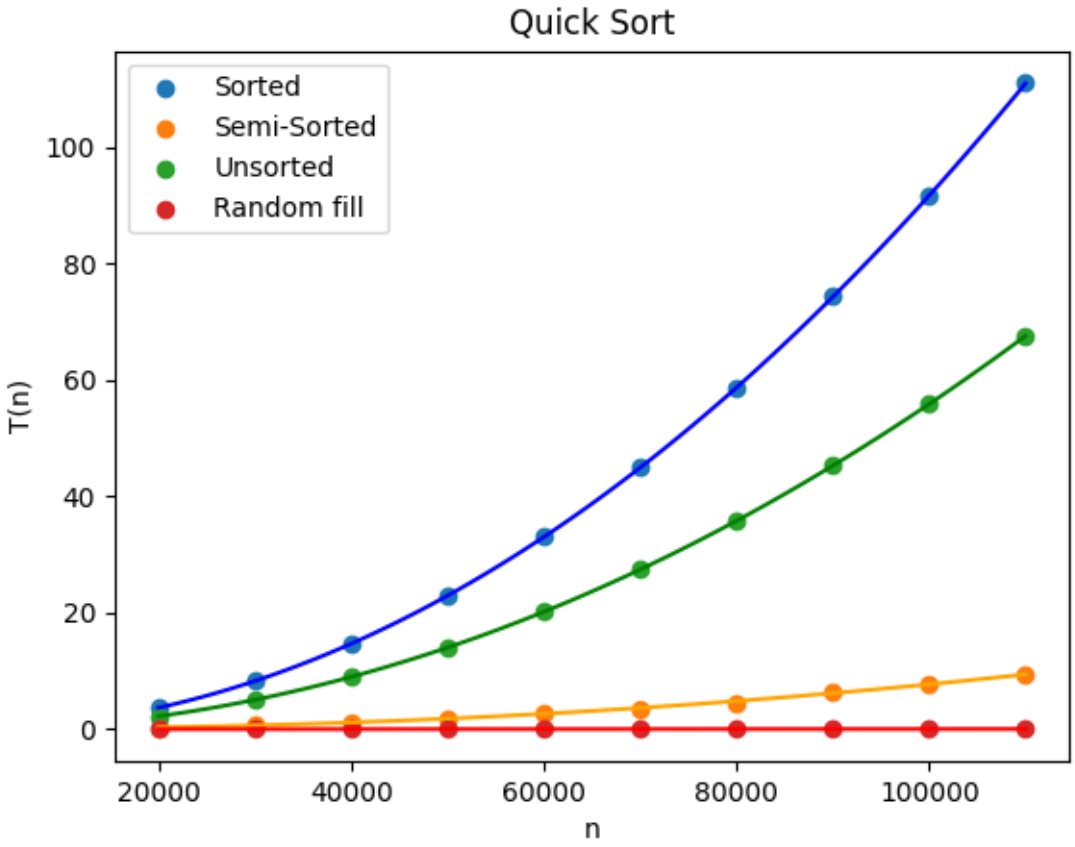

Рисунок 18 – График быстрой сортировки для всех перечисленных случаев

Пирамидальная Сортировка.

Отсортированный массив

Количество элементов	Время сортировки(сек)
20000	0.0092998
30000	0.0155491
40000	0.0201777
50000	0.0273056
60000	0.0324229
70000	0.0377784
80000	0.0446009
90000	0.0513214
100000	0.0562167
110000	0.063118

Таблица 33 – Время работы пирамидальной сортировки (отсортированный массив)

Почти отсортированный массив

Количество элементов	Время сортировки(сек)
20000	0.0091366
30000	0.0142432
40000	0.0201044

50000	0.0258544
60000	0.0328031
70000	0.0367901
80000	0.043337
90000	0.0494247
100000	0.0564179
110000	0.0606181

Таблица 34 – Время работы пирамидальной сортировки (почти отсортированный массив)

Отсортированный в сторону убывания массив

Количество элементов	Время сортировки(сек)
20000	0.0085953
30000	0.0138777
40000	0.0185811
50000	0.0252764
60000	0.0308708
70000	0.0351187
80000	0.0414403
90000	0.0462746
100000	0.0531185
110000	0.0595631

Таблица 35 – Время работы пирамидальной сортировки (массив отсортированный в сторону убывания)

Случайный массив

Количество элементов	Время сортировки(сек)
20000	0.0094293
30000	0.01591
40000	0.020359
50000	0.0261803
60000	0.032276
70000	0.0381665
80000	0.045444
90000	0.0523038
100000	0.0568849
110000	0.0643849

Таблица 36 – Время работы пирамидальной сортировки (случайный массив)

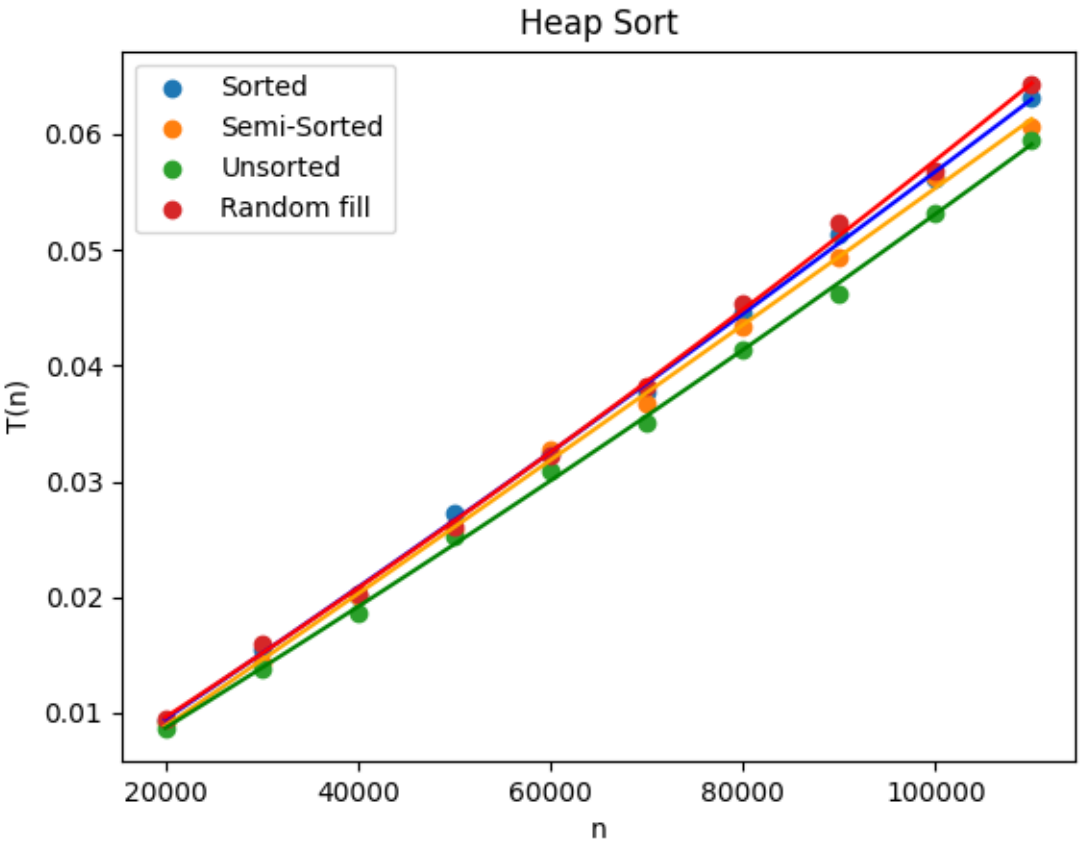

Рисунок 19 – График пирамидальной сортировки для всех перечисленных случаев

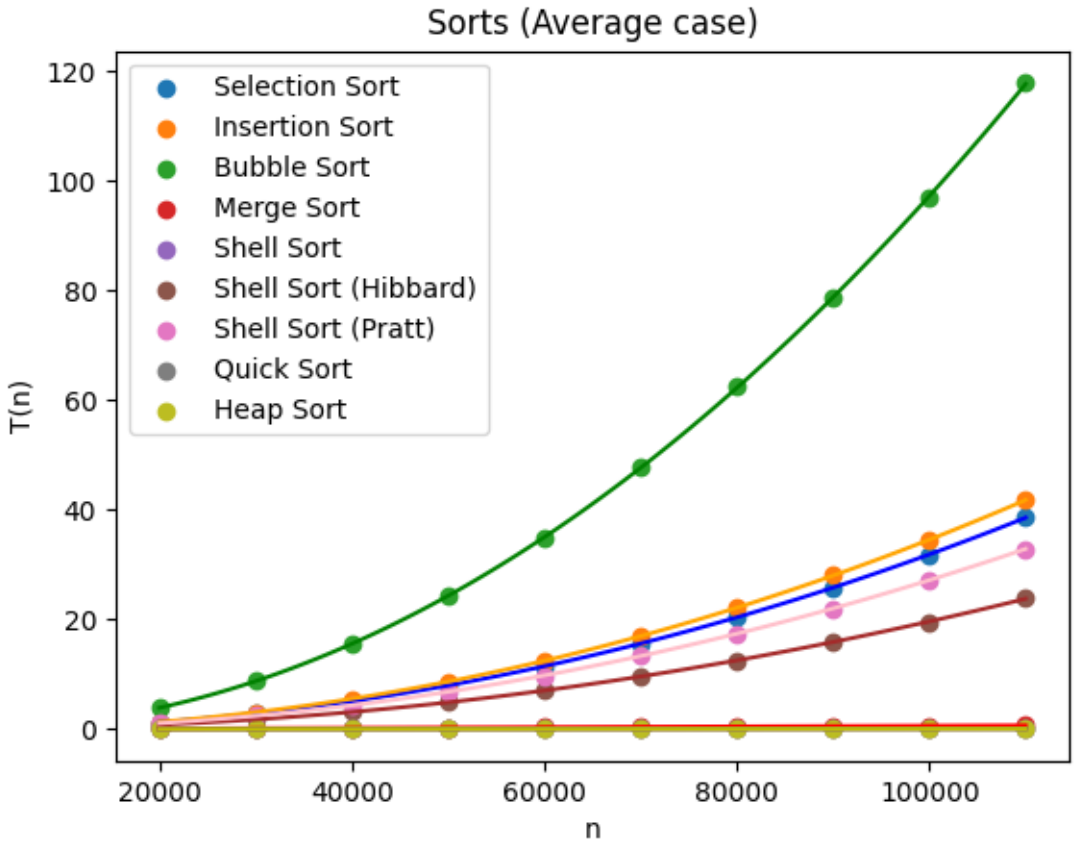

Рисунок 20 – График всех сортировок при случайном заполнении массива

Вывод:

Сравнивая теоретические и практические графики можно сделать вывод, что в большинстве случаев они согласовны и демонстрируют ожидаемую асимптотику, за исключением некоторых отклонений. Также, самой быстрой сортировкой в среднем случае при размере массива > 100 000 оказалась быстрая сортировка, что не совпадает с предположением из теоретической части.

Ссылки:

Репозиторий GitHub: https://github.com/KIRILLFABER/AICDLAB1

Код:

Сортировки и заполнение таблицы (C++):

```
#include <iostream>
#include <vector>
#include <ctime>
#include <fstream>
#include <string>
#include <chrono>

using namespace std;

const int RANGE = 30000; // максимальное значение элемента для случайного заполнения
const int FROM = 20000, TO = 120000, STEP = 10000; //



// Прототипы функций
void selectionSort(vector<int>& arr);
void insertionSort(vector<int>& arr);
int findMin(vector<int>& arr, int a);
void bubbleSort(vector<int>& arr);
vector<int> merge(vector<int> left, vector<int> right);
vector<int> mergeSort(vector<int> arr);
void shellSort(vector<int>& arr);
void shellSortHibb(vector<int>& arr);
void shellSortPratt(vector<int>& arr);
int partition(vector<int>& arr, int low, int high);
void quickSort(vector<int>& arr, int low, int high);
void heapify(vector<int>& arr, int n, int i);
void heapSort(vector<int>& arr);
```

```
void printArr(vector<int>& arr);
bool isSorted(vector<int>& arr);
void checkSorts();


// SORTS


// Функция сортировки выбором
void selectionSort(vector<int>& arr) {
    for (int i = 0; i < arr.size(); i++) {
        int j = findMin(arr, i); // Найти индекс минимального элемента
        // Обмен значений
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
}

// Функция для нахождения минимального элемента
int findMin(vector<int>& arr, int a) {
    int min = arr[a], min_index = a;
    for (int i = a + 1; i < arr.size(); i++) {
        if (min > arr[i]) {
            min = arr[i];
            min_index = i; // Запоминаем индекс минимального элемента
        }
    }
    return min_index;
}

// Функция сортировки вставками
void insertionSort(vector<int>& arr) {
```

```
    for (int i = 0; i < arr.size(); i++) {
        int el = arr[i]; // Сохраняем текущий элемент
        // Сдвигаем элементы, которые больше текущего
        for (int j = i; j > 0 && arr[j - 1] > el; j--) {
            arr[j] = arr[j - 1];
            arr[j - 1] = el; // Вставляем текущий элемент на правильную позицию

        }
    }
}


// Функция сортировки пузырьком
void bubbleSort(vector<int>& arr) {

    bool swapped = true; // флаг для отслеживания перестановок
    for (int i = 0; i < arr.size() - 1 && swapped; i++) {
        swapped = false;
        for (int j = 0; j < arr.size() - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Обмен значений
                int tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
                swapped = true;
            }
        }
    }

}

// Функция для слияния двух отсортированных массивов
vector<int> merge(vector<int> left, vector<int> right) {
    // Массив для слияния подмассивов
```

```
    vector<int> result((left.size() + right.size()));
    for (int i = 0, j = 0, k = 0; k < result.size(); k++) {
        // Заполнение массива
        result[k] = i < left.size() && (j == right.size() || left[i] < right[j]) ? left[i++] : right[j++];
    }
    return result;
}

// Функция сортировки слиянием
vector<int> mergeSort(vector<int> arr) {
    // Базовый случай: если массив содержит 1 элемент, он уже отсортирован
    if (arr.size() == 1) return arr;

    vector<int> left;  // Левый подмассив
    vector<int> right; // Правый подмассив
    // Делим массив на два подмассива
    for (int i = 0; i < arr.size(); i++) {
        if (i < arr.size() / 2)
            left.push_back(arr[i]); // Добавляем элемент в левый подмассив
        else
            right.push_back(arr[i]); // Добавляем элемент в правый подмассив
    }

    // Рекурсивно сортируем левый и правый подмассивы, затем сливаем их
    return merge(mergeSort(left), mergeSort(right));
}

// Функция сортировки Шелла
void shellSort(vector<int>& arr) {

    for (int s = arr.size() / 2; s > 0; s /= 2) { // Цикл по шагам s
        for (int i = s; i < arr.size(); ++i) {
            // Сравниваем текущий элемент с элементами, находящимися на расстоянии s
            for (int j = i - s; j >= 0 && arr[j] > arr[j + s]; j -= s) {
```

```
            // Обмен значений
            int temp = arr[j];
            arr[j] = arr[j + s];
            arr[j + s] = temp;
        }
    }
  }
}


void shellSortHibb(vector<int>& arr) {
    int i, j, k, increment, temp;
    int val;
    // Вычисление начального шага
    val = (int)log((float)arr.size() + 1) / log((float)2);
    increment = pow((float)2, val) - 1;
    while (increment > 0)
    {
        for (i = 0; i < increment; i++)
        {
            for (j = 0; j < arr.size(); j += increment)
            {
                temp = arr[j];
                for (k = j - increment; k >= 0 && temp < arr[k]; k -= increment)
                {
                    arr[k + increment] = arr[k]; // Сдвигаем элементы вправо
                }
                arr[k + increment] = temp; // Вставляем элемент temp в правильное место
            }
        }
        val--; // Уменьшаем val для следующего шага

        // Вычисление нового значения шага
        if (increment != 1)
```

```
            increment = pow((float)2, val) - 1;
        else
            increment = 0;
    }
}

void shellSortPratt(vector<int>& arr) {
    vector<int> gaps;
    int size = arr.size();

    // Генерация последовательности Пратта
    for (int i = 1; i < size; i *= 2)
    {
        for (int j = i; j < size; j *= 3)
        {
            gaps.push_back(j);
        }
    }

    // Сортировка шагов по убыванию
    insertionSort(gaps);

    // Сортировка массива по каждому шагу
    for (int gap : gaps)
    {
        for (int i = gap; i < size; i++)
        {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap){
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
```

```
    }

}
```

```
// Функция для разбиения массива на две части
int partition(vector<int>& arr, int low, int high) {

    int pivot = arr[high]; // Определяем опорный элемент как последний элемент массива


    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {


        if (arr[j] <= pivot) {
            i++; // Увеличиваем индекс меньшего элемента
            // Обмен значений
            int tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;

        }
    }
```

```
    // Меняем местами опорный элемент с элементом на позиции i + 1
    int tmp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = tmp;

    return (i + 1); // Возвращаем индекс опорного элемента
}

// Функция быстрой сортировки
void quickSort(vector<int>& arr, int low, int high) {

    if (low < high) {

        int pi = partition(arr, low, high); // Вычисление индекса опорного элемента
        quickSort(arr, low, pi - 1); // Быстрая сортировка для элементов до опорного
        quickSort(arr, pi + 1, high); // Быстрая сортировка для элементов после опорного
    }
}



void heapify(vector<int>& arr, int n, int i) {
    int largest = i;       // Инициализируем наибольший элемент как корень
    int left = 2 * i + 1;  // Левый дочерний элемент
    int right = 2 * i + 2; // Правый дочерний элемент

    // Если левый дочерний элемент больше корня
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // Если правый дочерний элемент больше, чем самый большой элемент на данный момент
    if (right < n && arr[right] > arr[largest])
        largest = right;
```

```
    // Если самый большой элемент не корень
    if (largest != i) {
        int tmp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = tmp;


        // Рекурсивно преобразуем затронутое поддерево
        heapify(arr, n, largest);
    }
}

// Функция пирамидальной сортировки
void heapSort(vector<int>& arr) {
    int n = arr.size();
    // Построение max-heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Извлечение элементов из кучи по одному
    for (int i = n - 1; i >= 0; i--) {
        // Перемещаем текущий корень в конец
        int tmp = arr[0];
        arr[0] = arr[i];
        arr[i] = tmp;

        // Вызываем heapify на уменьшенной куче
        heapify(arr, i, 0);
    }
}
```

```
// FILL


// Функция заполнения массива случайными числами
void randomFill(std::vector<int>& arr, int n) {
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        arr.push_back(rand() % RANGE);
    }
}

// Функция заполнения отсортированного массива
void sortedFill(std::vector<int>& arr, int n) {
    for (int i = 0; i < n; i++) {
        arr.push_back(i);
    }
}

// Функция заполнения почти отсортированного массива (90/10)
void semiSortedFill(std::vector<int>& arr, int n) {
    int sorted = n * 0.9;
    int unsorted = n - sorted;
    for (int i = 0; i < sorted - 1; i++) {
        arr.push_back(i);
    }
    for (int i = unsorted; i >= 0; i--) {
        arr.push_back(i);
    }
}

// Функция заполнения отсортированного массива в обратную сторону
```

```
void unSortedFill(std::vector<int>& arr, int n) {
    for (int i = n; i > 0; i--) {
        arr.push_back(i);
    }
}


// DATA


// Функция для заполнения файла с данными
void fillDataFile() {
    // Создание файла и его открытие
    ofstream data_file;
    data_file.open("DATA.csv");
    // Проверка файла на открытие
    if (!data_file.is_open()) {
        cout << "ERROR\n";
        return;
    }
    // Заполнение шапки таблицы
    data_file << "sort;fill;n;T(n)\n";

    // Заполнение таблицы

    // SELECTION SORT
    for (int n = FROM; n < TO; n += STEP) {
        vector<int> arr;
        sortedFill(arr, n);
        auto start = chrono::high_resolution_clock::now();
        selectionSort(arr);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<float> duration = end - start;
        float T = duration.count();
```

```
    data_file << "SS;S;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    semiSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    selectionSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SS;SS;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    unSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    selectionSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SS;US;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    randomFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    selectionSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SS;RS;" << n << ";" << T << endl;
}

// INSERTION SORT
```

```
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    sortedFill(arr, n);

    auto start = chrono::high_resolution_clock::now();
    insertionSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "IS;S;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    semiSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    insertionSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "IS;SS;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    unSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    insertionSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "IS;US;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    randomFill(arr, n);
```

```
    auto start = chrono::high_resolution_clock::now();
    insertionSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "IS;RS;" << n << ";" << T << endl;
}

// BUBBLE SORT
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    sortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    bubbleSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "BS;S;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    semiSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    bubbleSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "BS;SS;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    unSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    bubbleSort(arr);
```

```
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "BS;US;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    randomFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    bubbleSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "BS;RS;" << n << ";" << T << endl;
}

// MERGE SORT
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    sortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    mergeSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "MS;S;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    semiSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    mergeSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
```

```
    float T = duration.count();
    data_file << "MS;SS;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    unSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    mergeSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "MS;US;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    randomFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    mergeSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "MS;RS;" << n << ";" << T << endl;
}

// SHELL SORT
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    sortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    shellSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SHS;S;" << n << ";" << T << endl;
```

```
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    semiSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    shellSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SHS;SS;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    unSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    shellSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SHS;US;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    randomFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    shellSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SHS;RS;" << n << ";" << T << endl;
}

// SHELL SORT (HIBBARD)
```

```
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    sortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    shellSortHibb(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SHHS;S;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    semiSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    shellSortHibb(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SHHS;SS;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    unSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    shellSortHibb(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SHHS;US;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    randomFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
```

```
    shellSortHibb(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SHHS;RS;" << n << ";" << T << endl;
}

// SHELL SORT (PRATT)

for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    sortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    shellSortPratt(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SHPS;S;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    semiSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    shellSortPratt(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SHPS;SS;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    unSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    shellSortPratt(arr);
```

```
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SHPS;US;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    randomFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    shellSortPratt(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "SHPS;RS;" << n << ";" << T << endl;
}

// QUICK SORT
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    sortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    quickSort(arr, 0, arr.size() - 1);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "QS;S;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    semiSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    quickSort(arr, 0, arr.size() - 1);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
```

```
    float T = duration.count();
    data_file << "QS;SS;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    unSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    quickSort(arr, 0, arr.size() - 1);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "QS;US;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    randomFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    quickSort(arr, 0, arr.size() - 1);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "QS;RS;" << n << ";" << T << endl;
}

// HEAP SORT
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    sortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    heapSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "HS;S;" << n << ";" << T << endl;
```

```
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    semiSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    heapSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "HS;SS;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    unSortedFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    heapSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "HS;US;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    randomFill(arr, n);
    auto start = chrono::high_resolution_clock::now();
    heapSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = end - start;
    float T = duration.count();
    data_file << "HS;RS;" << n << ";" << T << endl;
}


data_file.close(); // Закрытие файла
```

```
}



// Функция для печати массива
void printArr(vector<int>& arr) {
    for (int i = 0; i < arr.size(); i++) {
        cout << arr[i] << ' '; // Вывод i-го элемента
    }
    cout << endl;
}

// Функция для проверки, отсортирован ли массив
bool isSorted(vector<int>& arr) {
    for (int i = 1; i < arr.size(); i++) {
        if (arr[i - 1] > arr[i]) return false;
    }
    return true;
}

// Функция для проверки всех сортировок
void checkSorts() {
    int size = 10000;
    vector<int> arr;



    randomFill(arr, size);
    // Sellection Sort
    vector<int> selectionArr = arr;
    selectionSort(selectionArr);
    cout << "Selection Sort: " << (isSorted(selectionArr) ? "Sorted" : "Not sorted")
        << ", Size: " << arr.size() << " -> " << selectionArr.size() << endl;
```

```
// Insertion Sort
vector<int> insertionArr = arr;
insertionSort(insertionArr);
cout << "Insertion Sort: " << (isSorted(insertionArr) ? "Sorted" : "Not sorted")
   << ", Size: " << arr.size() << " -> " << insertionArr.size() << endl;

// Bubble Sort
vector<int> bubbleArr = arr;
bubbleSort(bubbleArr);
cout << "Bubble Sort: " << (isSorted(bubbleArr) ? "Sorted" : "Not sorted")
   << ", Size: " << arr.size() << " -> " << bubbleArr.size() << endl;

// Merge Sort
vector<int> mergeArr = arr;
mergeArr = mergeSort(mergeArr);
cout << "Merge Sort: " << (isSorted(mergeArr) ? "Sorted" : "Not sorted")
   << ", Size: " << arr.size() << " -> " << mergeArr.size() << endl;

// Shell Sort
vector<int> shellArr = arr;
shellSort(shellArr);
cout << "Shell Sort: " << (isSorted(shellArr) ? "Sorted" : "Not sorted")
   << ", Size: " << arr.size() << " -> " << shellArr.size() << endl;

// Shell Sort (Hibbard)
vector<int> shellHibbArr = arr;
shellSortHibb(shellHibbArr);
cout << "Shell Sort Hibbard: " << (isSorted(shellHibbArr) ? "Sorted" : "Not sorted")
   << ", Size: " << arr.size() << " -> " << shellHibbArr.size() << endl;

// Shell Sort (Pratt)
vector<int> shellPrattArr = arr;
shellSortPratt(shellPrattArr);
```

```
    cout << "Shell Sort Pratt: " << (isSorted(shellPrattArr) ? "Sorted" : "Not sorted")
        << ", Size: " << arr.size() << " -> " << shellPrattArr.size() << endl;

    // Quick Sort
    vector<int> quickArr = arr;
    quickSort(quickArr, 0, quickArr.size() - 1);
    cout << "Quick Sort: " << (isSorted(quickArr) ? "Sorted" : "Not sorted")
        << ", Size: " << arr.size() << " -> " << quickArr.size() << endl;

    // Heap Sort
    vector<int> heapArr = arr;
    heapSort(heapArr);
    cout << "Heap Sort: " << (isSorted(heapArr) ? "Sorted" : "Not sorted")
        << ", Size: " << arr.size() << " -> " << heapArr.size() << endl;
}



int main() {

    //checkSorts();

    fillDataFile();

    return 0;
}
```

Обработка таблицы и создание графиков (Python):

```
import csv
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
from sympy import *
```

```
FROM = 10
TO = 30

def createTable():
    with open("TABLE.csv", "w") as w_file:
        names = ["Сортировка", "Заполнение", "Количество элементов", "Время сортировки(сек)"]
        writer = csv.DictWriter(w_file, delimiter = ";", lineterminator="\r", fieldnames=names)
        writer.writeheader()



        sorts = ["SS", "IS", "BS", "MS", "SHS", "SHHS", "SHPS", "QS", "HS"] # Список сортировок
        fills = ["S", "SS", "US", "RS"] # Список видов заполнения


        for sort in sorts:
            for fill in fills:
                # Запись заголовков в таблицу
                writer.writerow({"Сортировка": sort, "Заполнение": fill})
                writer.writerow({"Количество элементов": "Количество элементов", "Время
сортировки(сек)": "Время сортировки(сек)"})
                n = []  # Список для количества элементов
                T = []  # Список для времени сортировки
                readData(n, T, sort, fill)  # Чтение данных из файла
                for i in range(len(n)):
                    writer.writerow({"Количество элементов": n[i], "Время сортировки(сек)": T[i]})
                # Добавление пустых строк для удобства чтения
                for i in range(5):
                    writer.writerow({"Количество элементов": ' ', "Время сортировки(сек)": ' '})




def reg(n, T, col):
    # Находим коэффициенты регрессии 2-й степени
```

```
    coefficients = np.polyfit(n, T, 2)
    polynomial_regression = np.poly1d(coefficients)

    # Генерация значений для кривой регрессии
    x_reg = np.linspace(min(n), max(n), 100)
    y_reg = polynomial_regression(x_reg)

    # Построение кривой регрессии
    plt.plot(x_reg, y_reg, color=col)

def printGraphics(data_n, data_T, title, filename):
    plt.clf() # Очистка
    # Построение графика с точками
    plt.scatter(data_n[0], data_T[0], label='Sorted')
    plt.scatter(data_n[1], data_T[1], label='Semi-Sorted')
    plt.scatter(data_n[2], data_T[2], label='Unsorted')
    plt.scatter(data_n[3], data_T[3], label='Random fill')

    # Построение регрессионных кривых
    reg(data_n[0], data_T[0], 'blue')
    reg(data_n[1], data_T[1], 'orange')
    reg(data_n[2], data_T[2], 'green')
    reg(data_n[3], data_T[3], 'red')

    # Добавление подписей
    plt.xlabel('n')
    plt.ylabel('T(n)')
    plt.title(title)
    plt.legend()

    # Сохранение графика
    plt.savefig(filename)
    #plt.show()
```

```
def printGraphicsAverage(data_n, data_T, title, filename):
    plt.clf() # Очистка
    # Построение графика с точками

    plt.scatter(data_n[0], data_T[0], label='Selection Sort')
    plt.scatter(data_n[1], data_T[1], label='Insertion Sort')
    plt.scatter(data_n[2], data_T[2], label='Bubble Sort')
    plt.scatter(data_n[3], data_T[3], label='Merge Sort')
    plt.scatter(data_n[4], data_T[4], label='Shell Sort')
    plt.scatter(data_n[5], data_T[5], label='Shell Sort (Hibbard)')
    plt.scatter(data_n[6], data_T[6], label='Shell Sort (Pratt)')
    plt.scatter(data_n[7], data_T[7], label='Quick Sort')
    plt.scatter(data_n[8], data_T[8], label='Heap Sort')

    # Построение регрессионных кривых
    reg(data_n[0], data_T[0], 'blue')
    reg(data_n[1], data_T[1], 'orange')
    reg(data_n[2], data_T[2], 'green')
    reg(data_n[3], data_T[3], 'red')
    reg(data_n[4], data_T[4], 'purple')
    reg(data_n[5], data_T[5], 'brown')
    reg(data_n[6], data_T[6], 'pink')
    reg(data_n[7], data_T[7], 'gray')
    reg(data_n[8], data_T[8], 'y')

    # Добавление подписей
    plt.xlabel('n')
    plt.ylabel('T(n)')
    plt.title(title)
    plt.legend()

    # Сохранение графика
    plt.savefig(filename)
    #plt.show()
```

```
def readData(n, T, sort, fill):
    n.clear()  # Очистка списка n
    T.clear()  # Очистка списка T
    with open("DATA.csv", "r") as r_file:
        reader = csv.DictReader(r_file, delimiter=";")  # Чтение данных из DATA
        for row in reader:

            if(row["sort"] == sort and row["fill"] == fill):
                n.append(int(row["n"]))  # Количество элементов
                T.append(float(row["T(n)"]))  # Время сортировки

# Очистка данных
def clearData():
    global n_s, T_s, n_ss, T_ss, n_us, T_us, n_rs, T_rs
    n_s.clear()
    T_s.clear()
    n_ss.clear()
    T_ss.clear()
    n_us.clear()
    T_us.clear()
    n_rs.clear()
    T_rs.clear()

def plotSort(sort, name, filename):
    clearData()  # Очищаем данные перед новой отрисовкой
    # Чтение данных
```

```
    readData(n_s, T_s, sort, "S")
    readData(n_ss, T_ss, sort, "SS")
    readData(n_us, T_us, sort, "US")
    readData(n_rs, T_rs, sort, "RS")
    data_n = [n_s, n_ss, n_us, n_rs]
    data_T = [T_s, T_ss, T_us, T_rs]
    # Отрисовка графика
    printGraphics(data_n, data_T, name, filename)


def plotSortAverage(filename):
    # Списки для времени и количества элементов каждой сортировки
    T_1 = []
    T_2 = []
    T_3 = []
    T_4 = []
    T_5 = []
    T_6 = []
    T_7 = []
    T_8 = []
    T_9 = []
    n_1 = []
    n_2 = []
    n_3 = []
    n_4 = []
    n_5 = []
    n_6 = []
    n_7 = []
    n_8 = []
    n_9 = []

    # Чтение данных
    readData(n_1, T_1, "SS", "RS")
    readData(n_2, T_2, "IS", "RS")
```

```
    readData(n_3, T_3, "BS", "RS")
    readData(n_4, T_4, "MS", "RS")
    readData(n_5, T_5, "SHS", "RS")
    readData(n_6, T_6, "SHHS", "RS")
    readData(n_7, T_7, "SHPS", "RS")
    readData(n_8, T_8, "QS", "RS")
    readData(n_9, T_9, "HS", "RS")

    data_T = [T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9]
    data_n = [n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9]
    # Отрисовка графика
    printGraphicsAverage(data_n, data_T, "Sorts (Average case)", filename)



# Графики для теории
n = Symbol('n')

# Графики для каждой сортировки (все случаи на одном изображении)
# Selection Sort
title = "Selection Sort"
T_w = 5 * n ** 2 + 3 * n + 1
T_b = 3 * n ** 2 + 4 * n + 1
T_a = 4 * n ** 2 + 7 * n + 1
plot1 = plot(T_w, (n, FROM, TO), show = False, label = "Worst case", xlabel = "n", ylabel = "T(n)", title
= title)
plot2 = plot(T_b, (n, FROM, TO), show = False, label = "Best case")
plot3 = plot(T_a, (n, FROM, TO), show = False, label = "Average case")

plot1.extend(plot2)
plot1.extend(plot3)
plot1.legend = True


plot1.save('TheorGraphics\SS.png')
```

```
# Insertion Sort

title = "Insertion Sort"


T_w = 3 * n ** 2 + 7 * n + 1

T_b = 6 * n + 1

T_a = 3 * n ** 2 + 4 * n + 1

plot1 = plot(T_w, (n, FROM, TO), show = False, label = "Worst case", xlabel = "n", ylabel = "T(n)", title = title)

plot2 = plot(T_b, (n, FROM, TO), show = False, label = "Best case")

plot3 = plot(T_a, (n, FROM, TO), show = False, label = "Average case")


plot1.extend(plot2)

plot1.extend(plot3)

plot1.legend = True



plot1.save('TheorGraphics\IS.png')

# Bubble Sort

title = "Bubble Sort"


T_w = 7 * n ** 2 - 10 * n + 5

T_b = 3 * n + 3

T_a = 5 * n ** 2 - n - 2

plot1 = plot(T_w, (n, FROM, TO), show = False, label = "Worst case", xlabel = "n", ylabel = "T(n)", title = title)

plot2 = plot(T_b, (n, FROM, TO), show = False, label = "Best case")

plot3 = plot(T_a, (n, FROM, TO), show = False, label = "Average case")


plot1.extend(plot2)

plot1.extend(plot3)

plot1.legend = True



plot1.save('TheorGraphics\BS.png')
```

```
# Merge Sort

title = "Merge Sort"


T_w = 2 * n + log(n, 2) * (6 * n + 5)

T_b = 2 * n + log(n, 2) * (6 * n + 5)

T_a = 2 * n + log(n, 2) * (6 * n + 5)

plot1 = plot(T_w, (n, FROM, TO), show = False, label = "Worst case", xlabel = "n", ylabel = "T(n)", title = title)

plot2 = plot(T_b, (n, FROM, TO), show = False, label = "Best case")

plot3 = plot(T_a, (n, FROM, TO), show = False, label = "Average case")


plot1.extend(plot2)

plot1.extend(plot3)

plot1.legend = True



plot1.save('TheorGraphics\MS.png')

# Shell Sort

title = "Shell Sort"


T_w = n ** 2

T_b = n ** 1.667

T_a = n ** 1.5

plot1 = plot(T_w, (n, FROM, TO), show = False, label = "Worst case", xlabel = "n", ylabel = "T(n)", title = title)

plot2 = plot(T_b, (n, FROM, TO), show = False, label = "Best case")

plot3 = plot(T_a, (n, FROM, TO), show = False, label = "Average case")


plot1.extend(plot2)

plot1.extend(plot3)

plot1.legend = True



plot1.save('TheorGraphics\SHS.png')

# Shell Sort (Hibbard)
```

```
title = "Shell Sort (Hibbard)"


T_w = n ** 2

T_b = n * log(n, 2)

T_a = n ** 1.25

plot1 = plot(T_w, (n, FROM, TO), show = False, label = "Worst case", xlabel = "n", ylabel = "T(n)", title = title)

plot2 = plot(T_b, (n, FROM, TO), show = False, label = "Best case")

plot3 = plot(T_a, (n, FROM, TO), show = False, label = "Average case")


plot1.extend(plot2)

plot1.extend(plot3)

plot1.legend = True



plot1.save('TheorGraphics\SHHS.png')

# Shell Sort (Pratt)

title = "Shell Sort (Pratt)"


T_w = n * log(2, n) ** 2

T_b = n

T_a = n * log(2, n) ** 2

plot1 = plot(T_w, (n, FROM, TO), show = False, label = "Worst case", xlabel = "n", ylabel = "T(n)", title = title)

plot2 = plot(T_b, (n, FROM, TO), show = False, label = "Best case")

plot3 = plot(T_a, (n, FROM, TO), show = False, label = "Average case")


plot1.extend(plot2)

plot1.extend(plot3)

plot1.legend = True



plot1.save('TheorGraphics\SHPS.png')


# Quick Sort
```

```
title = "Quick Sort"


T_w = n ** 2 + 2 * n

T_b = n + 2 * n * log(n, 2) - log(n, 2)

T_a = 3 * n + 3 * n * log(n, 3) + 2

plot1 = plot(T_w, (n, FROM, TO), show = False, label = "Worst case", xlabel = "n", ylabel = "T(n)", title
= title)

plot2 = plot(T_b, (n, FROM, TO), show = False, label = "Best case")

plot3 = plot(T_a, (n, FROM, TO), show = False, label = "Average case")


plot1.extend(plot2)

plot1.extend(plot3)

plot1.legend = True



plot1.save('TheorGraphics\QS.png')

# Heap Sort

title = "Heap Sort"

T_w = n * log(n, 2)

T_b = n * log(n, 2)

T_a = n * log(n, 2)

plot1 = plot(T_w, (n, FROM, TO), show = False, label = "Worst case", xlabel = "n", ylabel = "T(n)", title
= title)

plot2 = plot(T_b, (n, FROM, TO), show = False, label = "Best case")

plot3 = plot(T_a, (n, FROM, TO), show = False, label = "Average case")


plot1.extend(plot2)

plot1.extend(plot3)

plot1.legend = True



plot1.save('TheorGraphics\HS.png')



# График всех сортировок в среднем случае
```

```
title = "Sorts (Average case)"

T_1 = 4 * n ** 2 + 7 * n + 1

T_2 = 3 * n ** 2 + 4 * n + 1

T_3 = 5 * n ** 2 - n - 2

T_4 = 2 * n + log(n, 2) * (6 * n + 5) ###

T_5 = n ** 1.5

T_6 = n ** 1.25

T_7 = n * log(2, n) ** 2

T_8 = 3 * n + 3 * n * log(n, 3) + 2

T_9 = n * log(n, 2)


plot1 = plot(T_1, (n, FROM, TO), show = False, label = "Selection sort", xlabel = "n", ylabel = "T(n)",
title = title)

plot2 = plot(T_2, (n, FROM, TO), show = False, label = "Insertion Sort")

plot3 = plot(T_3, (n, FROM, TO), show = False, label = "Bubble Sort")

plot4 = plot(T_4, (n, FROM, TO), show = False, label = "Merge Sort")

plot5 = plot(T_5, (n, FROM, TO), show = False, label = "Shell Sort")

plot6 = plot(T_6, (n, FROM, TO), show = False, label = "Shell Sort (Hibbard)")

plot7 = plot(T_7, (n, FROM, TO), show = False, label = "Shell Sort (Pratt)")

plot8 = plot(T_8, (n, FROM, TO), show = False, label = "Quick Sort")

plot9 = plot(T_9, (n, FROM, TO), show = False, label = "Heap Sort")


plot1.extend(plot2)

plot1.extend(plot3)

plot1.extend(plot4)

plot1.extend(plot5)

plot1.extend(plot6)

plot1.extend(plot7)

plot1.extend(plot8)

plot1.extend(plot9)

plot1.legend = True


plot1.save('TheorGraphics\ALL.png')
```

```
# Графики для практики


# Графики для каждой сортировки (каждое заполнение)
n_s = []
T_s = []
n_ss = []
T_ss = []
n_us = []
T_us = []
n_rs = []
T_rs = []

plotSort("SS", "Selection Sort", "PracGraphics\SS.png")
plotSort("IS", "Insertion Sort", "PracGraphics\IS.png")
plotSort("BS", "Bubble Sort", "PracGraphics\BS.png")
plotSort("MS", "Merge Sort", "PracGraphics\MS.png")
plotSort("SHHS", "Shell Sort (Hibbard)", "PracGraphics\SHHS.png")
plotSort("SHPS", "Shell Sort (Pratt)", "PracGraphics\SHPS.png")
plotSort("SHS", "Shell Sort", "PracGraphics\SHS.png")
plotSort("QS", "Quick Sort", "PracGraphics\QS.png")
plotSort("HS", "Heap Sort", "PracGraphics\HS.png")
plotSortAverage("PracGraphics\ALL.png")

createTable() # Создание отдельной таблицы
```