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:

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

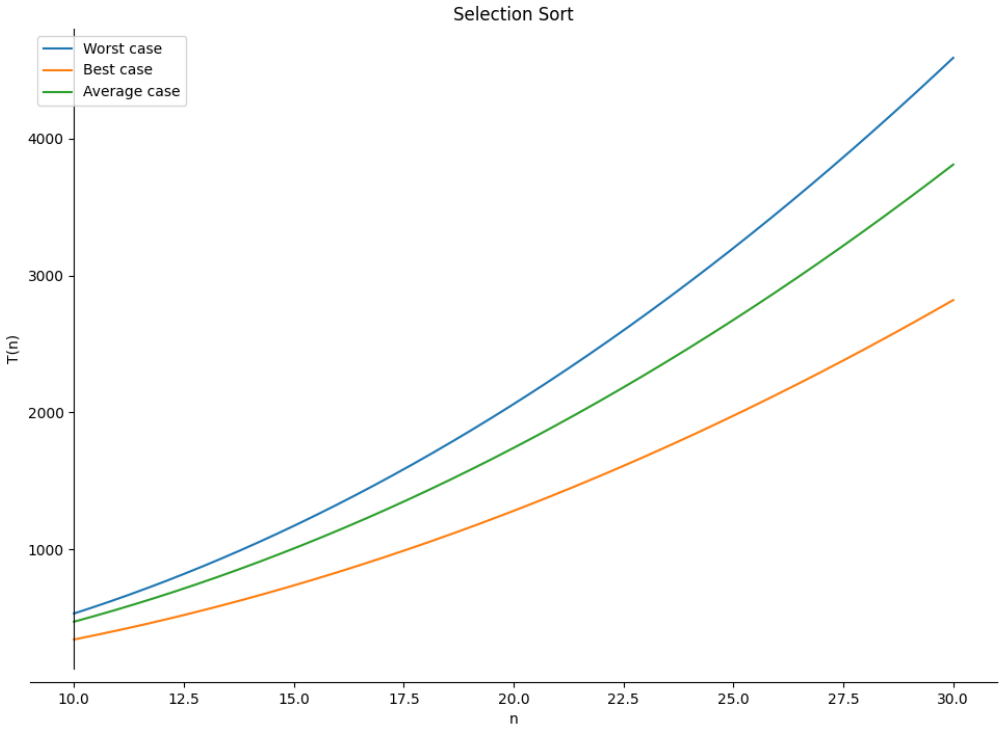

Рисунок 1 – График функции временной сложности сортировки выбором

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:

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

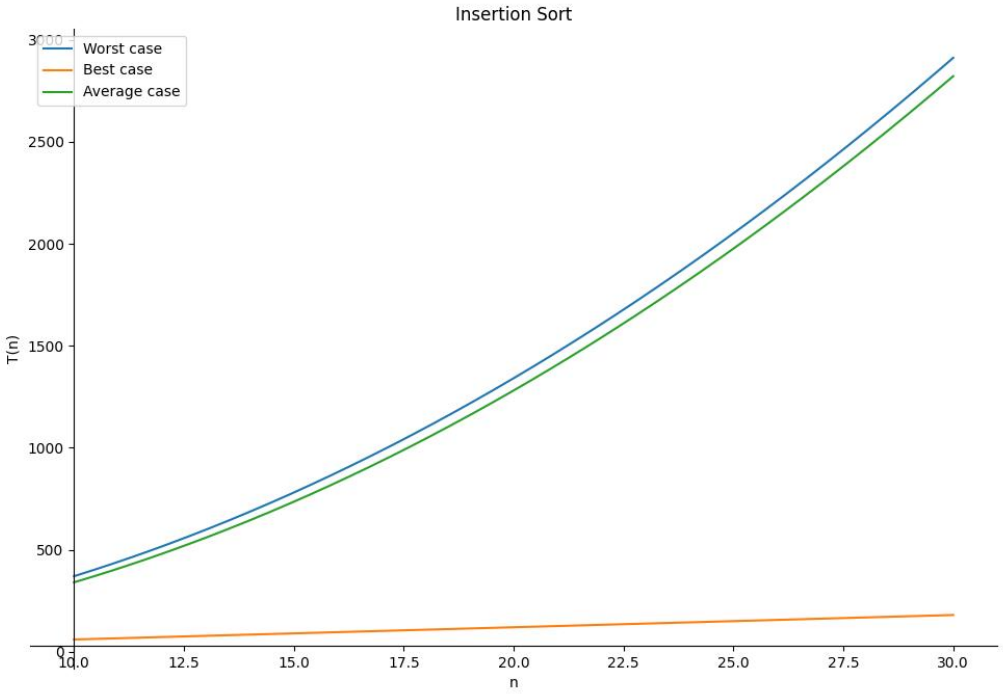

Рисунок 2 – График функции временной сложности сортировки вставками

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:

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

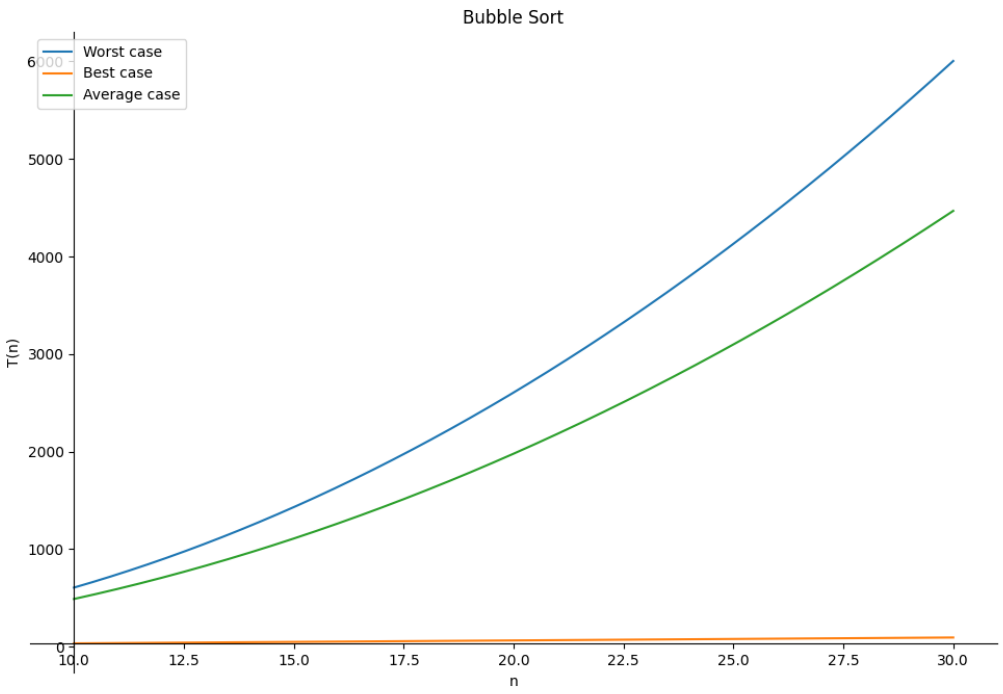

Рисунок 3 – График функции временной сложности сортировки пузырьком

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,

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

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

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

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:

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

Лучший случай:

Аналогично:

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

Средний случай:

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

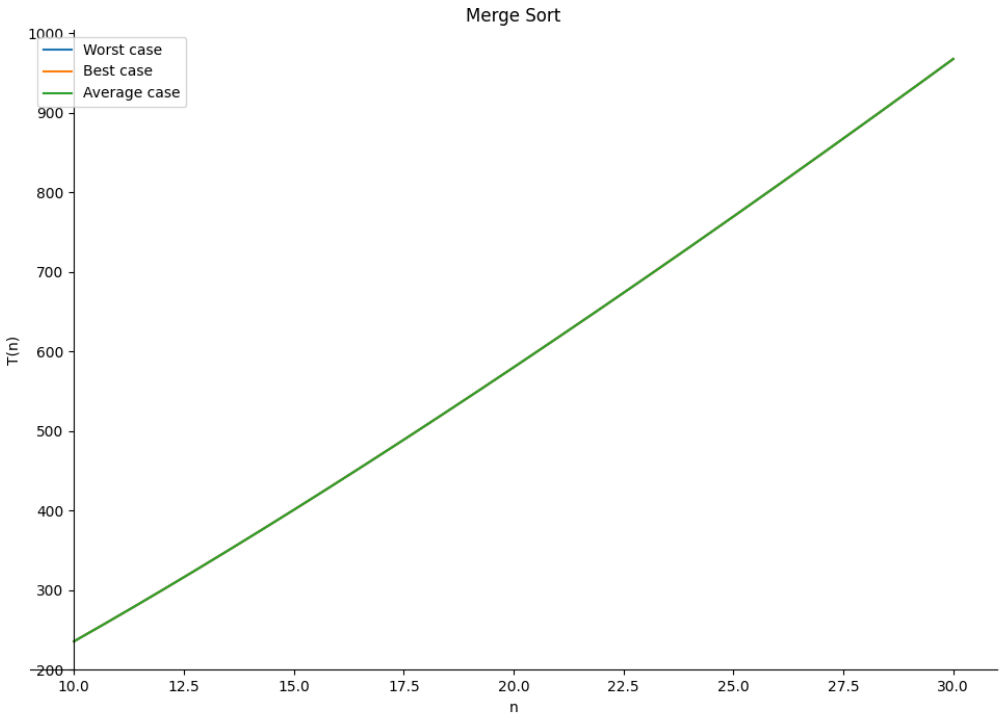

Рисунок 4 – График функции временной сложности сортировки слиянием

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:

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

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:

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

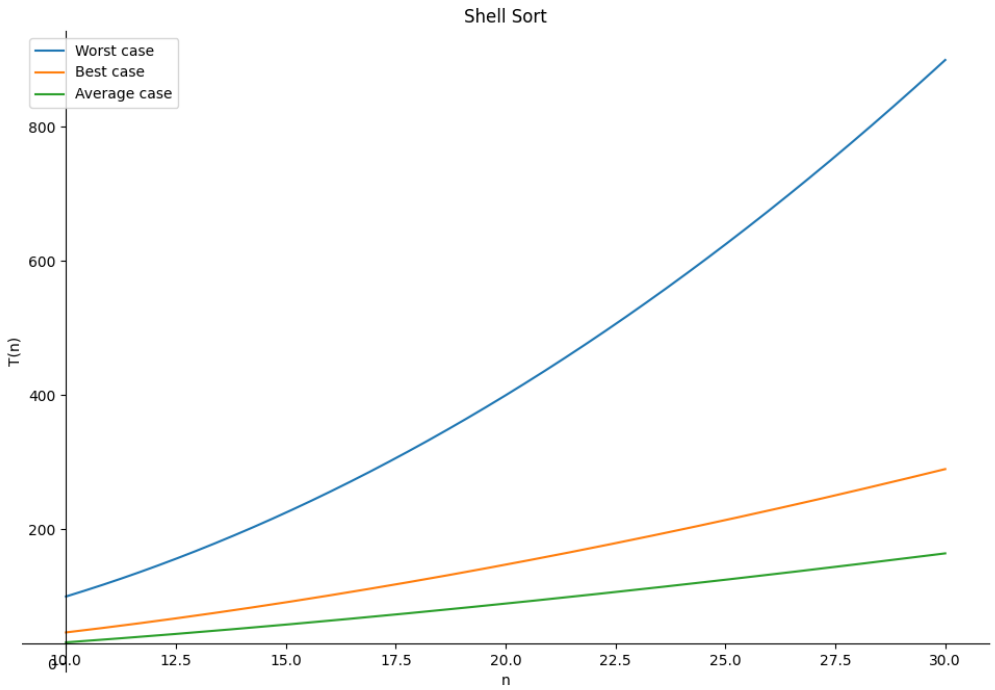

Рисунок 5 – График функции временной сложности сортировки Шелла

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:

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

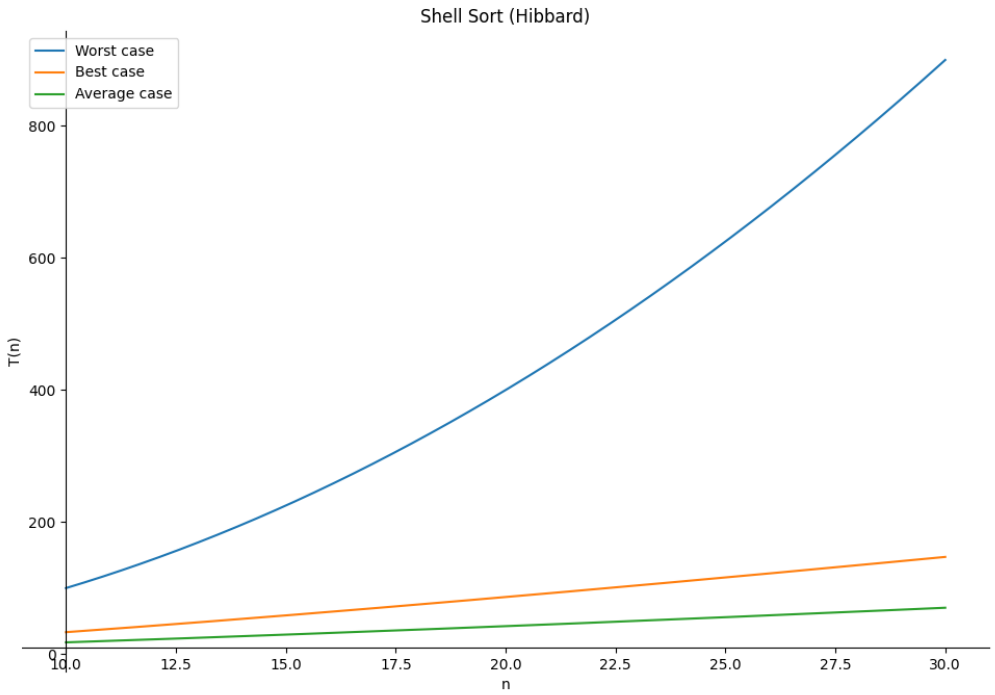

Рисунок 6 – График функции временной сложности сортировки Шелла (последовательность Хиббарда)

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:

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

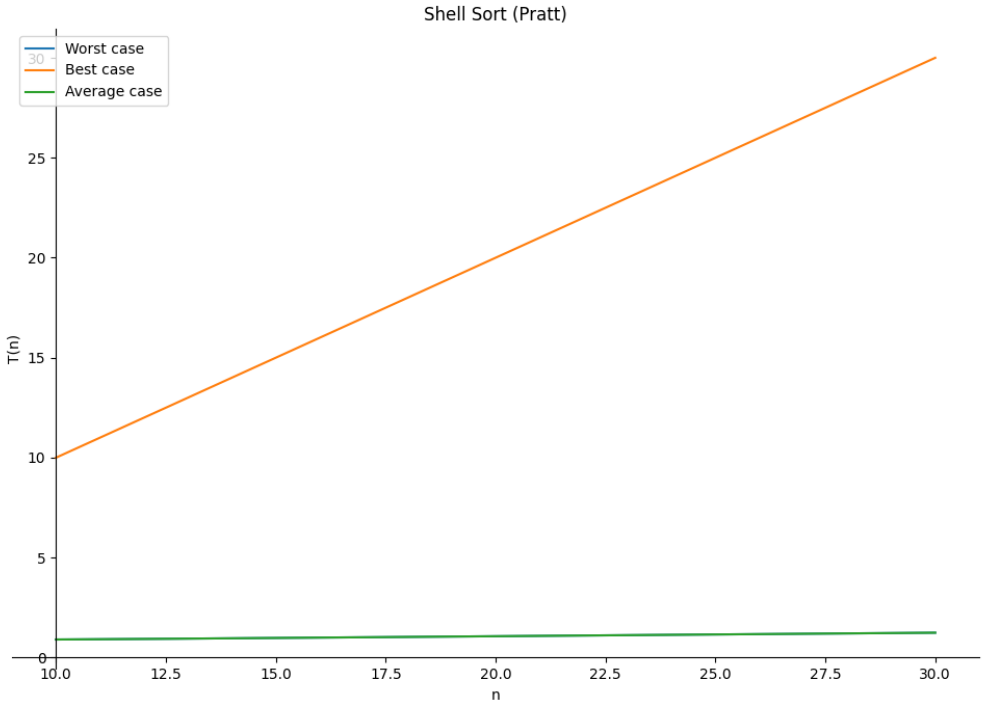

Рисунок 7 – График функции временной сложности сортировки Шелла(последовательность Пратта)

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:

$$T_{QS}^{a}(n) = T_{QS}^{a}\left(\frac{n}{3}\right) + T_{QS}^{a}\left(\frac{2n}{3}\right) + 2n - 1$$
$$= \left(T_{QS}^{a}\left(\frac{n}{9}\right) + T_{QS}^{a}\left(\frac{2n}{9}\right) + \frac{2n}{3} - 1\right) + (T_{QS}^{a}\left(\frac{2n}{9}\right) + T_{QS}^{a}\left(\frac{4n}{9}\right) + \frac{4n}{3}$$
$$- 1) + 2n - 1 =$$
$$= T_{QS}^{a}\left(\frac{n}{9}\right) + T_{QS}^{a}\left(\frac{2n}{9}\right) + T_{QS}^{a}\left(\frac{2n}{9}\right) + T_{QS}^{a}\left(\frac{4n}{9}\right) + \frac{2n}{3} - 1 + \frac{4n}{3} - 1$$
$$+ 2n - 1$$
$$= (T_{QS}^{a}\left(\frac{n}{27}\right) + T_{QS}^{a}\left(\frac{2n}{27}\right) + \frac{2n}{9} - 1) + 2(T_{QS}^{a}\left(\frac{2n}{27}\right) + T_{QS}^{a}\left(\frac{4n}{27}\right) + \frac{4n}{9}$$
$$- 1) + (T_{QS}^{a}\left(\frac{4n}{27}\right) + T_{QS}^{a}\left(\frac{8n}{27}\right) + \frac{8n}{9} - 1) + \frac{2n}{3} - 1 + \frac{4n}{3} - 1 + 2n$$
$$- 1 = \cdots = T_{QS}^{a}\left(\frac{n}{3^k}\right) + T_{QS}^{a}\left(\frac{2n}{3^k}\right) + \sum_{i=1}^{k}(\frac{2n}{3^i} - 1)$$
$$= 2 + \sum_{i=1}^{\log_3 n - 1}\frac{2n}{3^i} + \mathrm{n}\log_3 n = 2 + 2n\frac{1 - \left(\frac{1}{3}\right)^{\log_3 n}}{1 - \frac{1}{3}} + n\log_3 n$$
$$= 2 + 2n\frac{1 - \frac{1}{n}}{\frac{2}{3}} + n\log_3 n = 2 + 3n + 3\,\mathrm{n}\log_3 n = \theta(n\log n)$$

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:

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

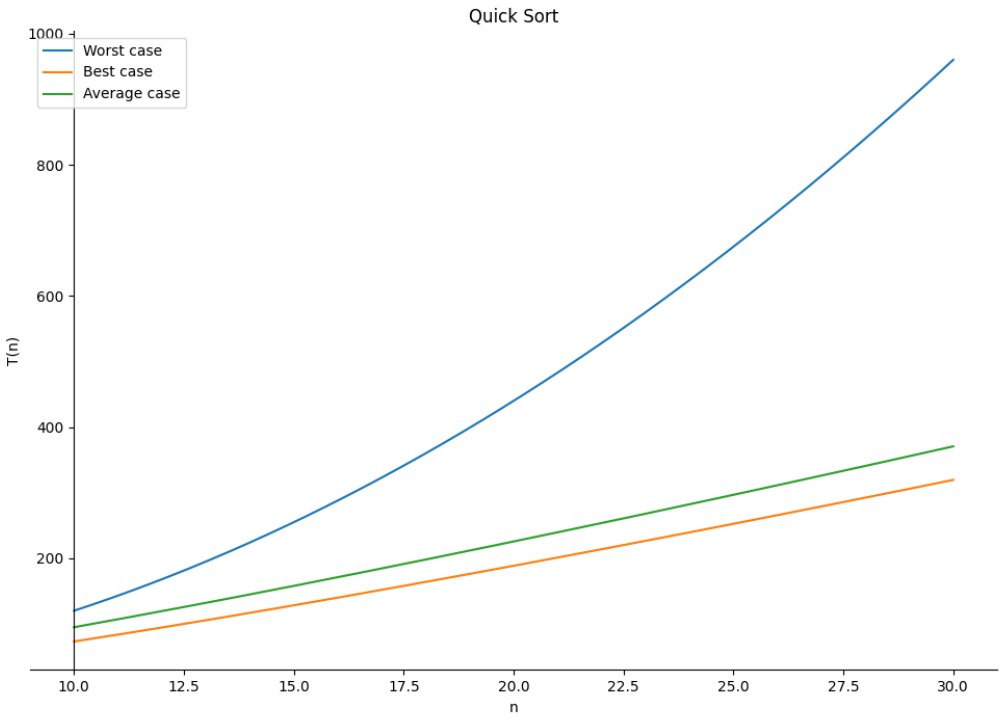

Рисунок 8 – График функции временной сложности быстрой сортировки

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:

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

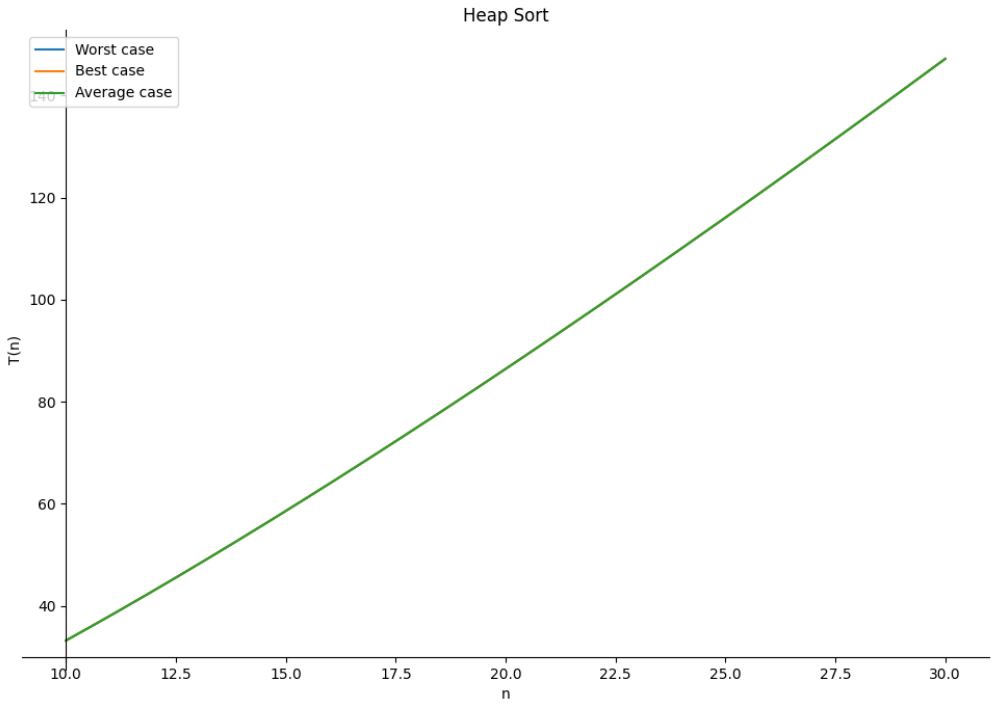

Рисунок 9 – График функции временной сложности пирамидальной сортировки

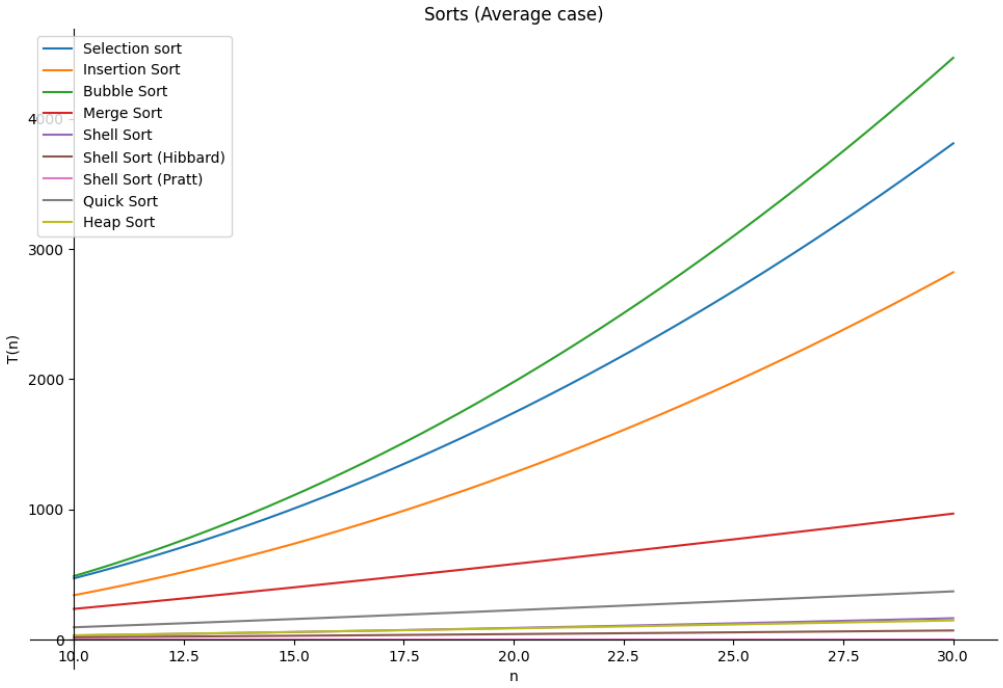

Рисунок 10 – Графики функции временной сложности всех сортировок в среднем случае

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.

Обозначения в таблице:

Сортировки:

SS – Сортировка выбором

IS – Сортировка вставками

BS – Сортировка пузырьком

MS – Сортировка слиянием

SHS – Сортировка Шелла

SHHS – Сортировка Шелла с последовательностью Хиббарда

SHPS – Сортировка Шелла с последовательностью Пратта

QS – Быстрая сортировка

HS – Пирамидальная сортировка

Исходный массив:

S – Отсортированный массив

SS – Почти отсортированный массив (90/10)

US – Отсортированный в обратную сторону массив

RS – Массив, заполненный случайными числами

Сортировка	Заполнение	Количество элементов	Время сортировки(сек)
SS	S		
		10000	0.432192
		20000	1.29126
		30000	2.75478
		40000	4.89201
		50000	7.65748
		60000	10.9937
		70000	15.0872
		80000	19.5353
		90000	24.8854
		100000	31.0504
		110000	37.1149
SS	SS		
		10000	0.309458
		20000	1.25082
		30000	2.82078
		40000	5.08329
		50000	7.82753
		60000	11.2627
		70000	15.3235
		80000	20.0707
		90000	25.6294
		100000	31.3773
		110000	37.928
SS	US		
		10000	0.381257
		20000	1.53081

		30000	3.44977
		40000	6.12647
		50000	9.5624
		60000	13.7782
		70000	18.7806
		80000	24.5653
		90000	31.0834
		100000	38.3282
		110000	46.2528
SS	RS		
		10000	0.307281
		20000	1.22746
		30000	2.74793
		40000	5.23586
		50000	8.44214
		60000	13.7763
		70000	18.6002
		80000	24.9288
		90000	27.5836
		100000	31.7841
		110000	36.9191

Таблица 1 – Время работы сортировки выбором для разных случаев

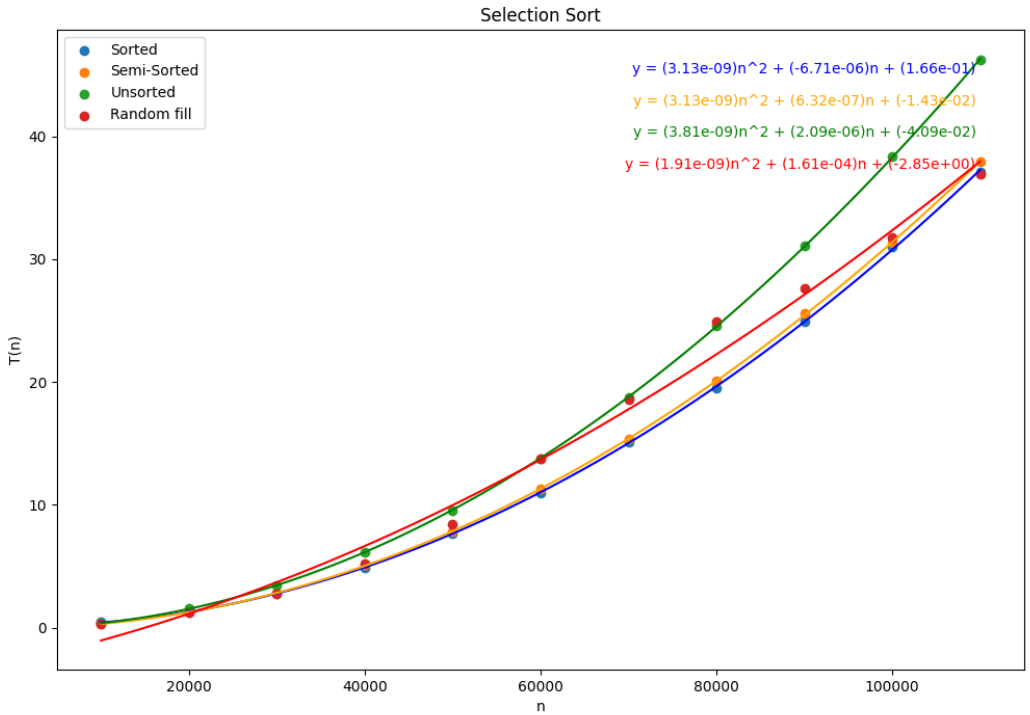

Рисунок 11 – График сортировки выбором для всех перечисленных случаев

Сортировка вставками.

Сортировка	Заполнение	Количество элементов	Время сортировки(сек)
IS	S		
		10000	8.58e-05
		20000	0.0001778
		30000	0.0002751
		40000	0.0003745
		50000	0.0004594
		60000	0.0005308
		70000	0.0006685
		80000	0.0007347
		90000	0.0008532
		100000	0.0010105
		110000	0.0010275
IS	SS		
		10000	0.118842
		20000	0.473401
		30000	1.05393
		40000	1.87296
		50000	2.93918
		60000	4.24972
		70000	5.76355
		80000	7.51071
		90000	9.55902
		100000	14.2675
		110000	15.6239
IS	US		
		10000	0.652456
		20000	2.62258
		30000	5.99825
		40000	10.6908
		50000	16.5368
		60000	23.7153
		70000	32.0017
		80000	42.0567
		90000	52.9515
		100000	65.5407
		110000	79.0832
IS	RS		
		10000	0.333994
		20000	1.32284
		30000	2.94169
		40000	5.21082
		50000	8.1482
		60000	11.8219
		70000	15.9767
		80000	23.8598

	90000	27.5079
	100000	37.0094
	110000	39.5525

Таблица 2 – Время работы сортировки вставками для разных случаев

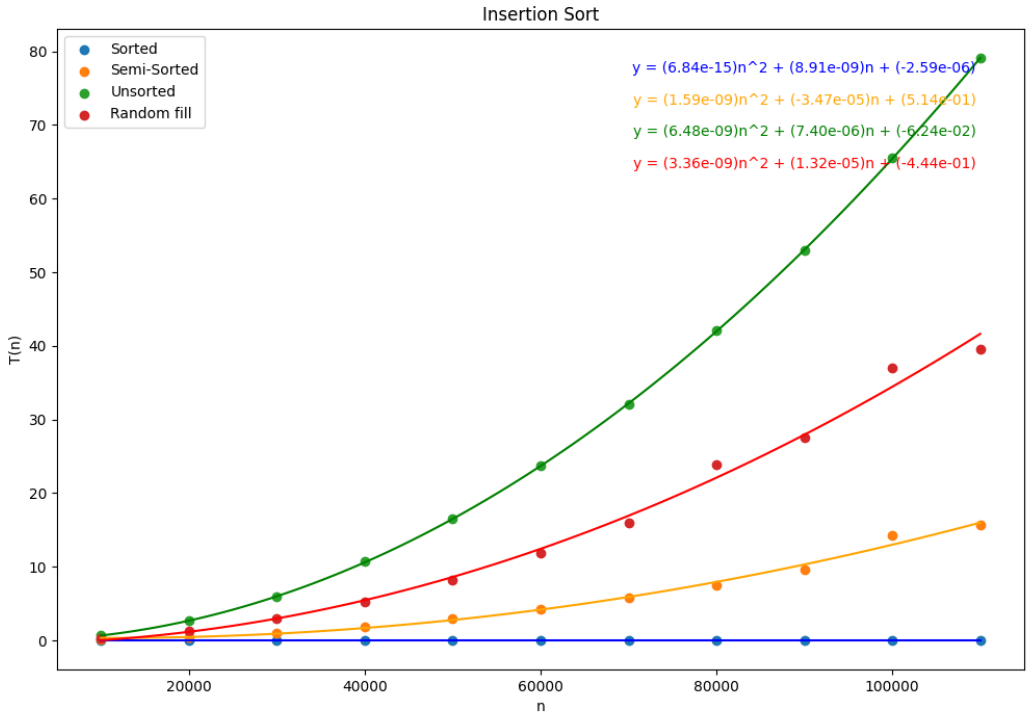

Рисунок 12 – График сортировки вставками для всех перечисленных случаев

Сортировка пузырьком.

Сортировка	Заполнение	Количество элементов	Время сортировки(сек)
BS	S		
		10000	8.69e-05
		20000	0.0001829
		30000	0.0002656
		40000	0.0003667
		50000	0.0004604
		60000	0.0005536
		70000	0.0006487
		80000	0.0007196
		90000	0.0008253
		100000	0.0008984
		110000	0.0010268
BS	SS		
		10000	0.590011
		20000	2.35306
		30000	5.26908
		40000	9.37429

		50000	14.6774
		60000	21.1202
		70000	28.8029
		80000	37.4752
		90000	47.4852
		100000	58.4973
		110000	70.8491
BS	US		
		10000	1.09592
		20000	4.39606
		30000	9.86374
		40000	17.5572
		50000	27.8516
		60000	39.5838
		70000	53.6352
		80000	70.1138
		90000	89.6466
		100000	109.596
		110000	132.712
BS	RS		
		10000	0.932384
		20000	3.70602
		30000	8.32765
		40000	14.8092
		50000	23.213
		60000	33.5204
		70000	45.3819
		80000	62.1944
		90000	75.4081
		100000	97.5033
		110000	117.504

Таблица 3 – Время работы сортировки пузырьком для разных случаев

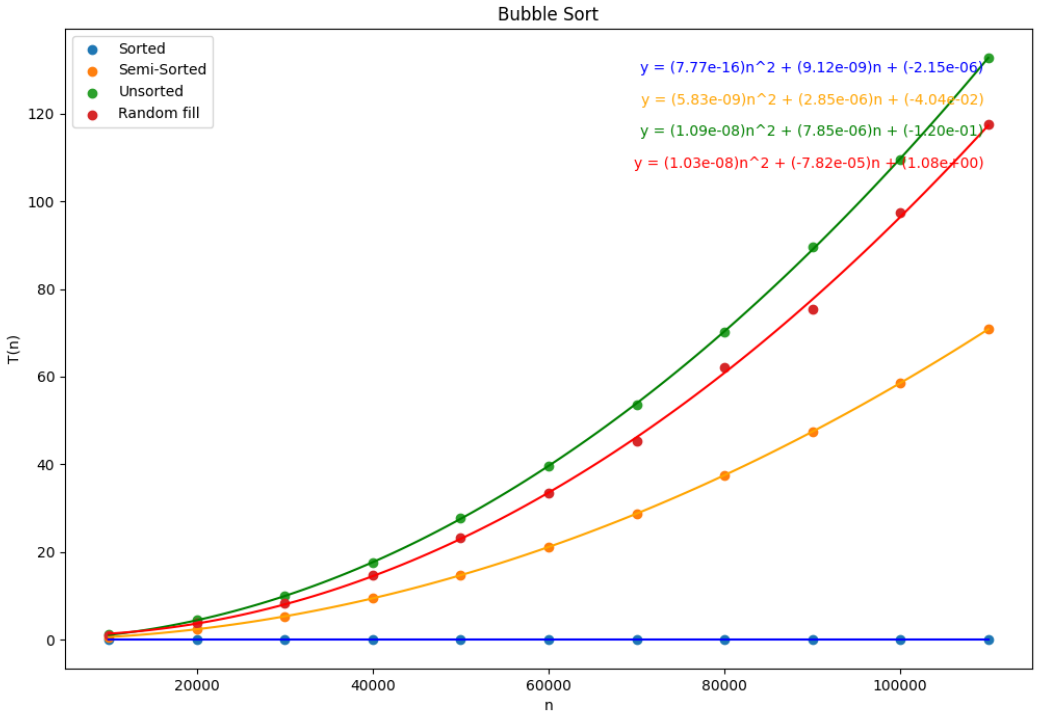

Рисунок 13 – График сортировки пузырьком для всех перечисленных случаев

Сортировка слиянием.

Сортировка	Заполнение	Количество элементов	Время сортировки(сек)
MS	S		
		10000	0.0600646
		20000	0.117101
		30000	0.169873
		40000	0.230453
		50000	0.284562
		60000	0.346617
		70000	0.405674
		80000	0.468342
		90000	0.528692
		100000	0.578752
		110000	0.644183
MS	SS		
		10000	0.0557126
		20000	0.120674
		30000	0.172239
		40000	0.228718
		50000	0.285598
		60000	0.351868
		70000	0.402663
		80000	0.470206
		90000	0.523566

		100000	0.58072
		110000	0.650315
MS	US		
		10000	0.0565053
		20000	0.112931
		30000	0.169492
		40000	0.229905
		50000	0.285248
		60000	0.351411
		70000	0.402939
		80000	0.480513
		90000	0.524759
		100000	0.579664
		110000	0.647454
MS	RS		
		10000	0.0615326
		20000	0.114147
		30000	0.175867
		40000	0.234358
		50000	0.293434
		60000	0.353408
		70000	0.41278
		80000	0.497548
		90000	0.554838
		100000	0.593705
		110000	0.656557

Таблица 4 – Время работы сортировки слиянием для разных случаев

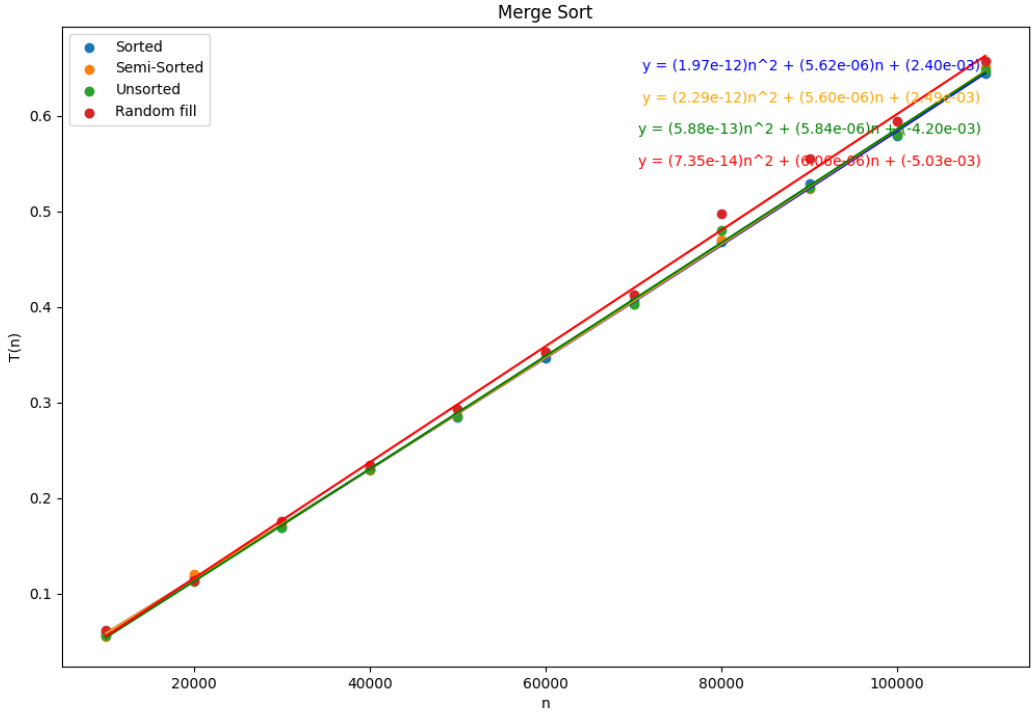

Рисунок 14 – График сортировки слиянием для всех перечисленных случаев

Сортировка Шелла.

Сортировка	Заполнение	Количество элементов	Время сортировки(сек)
SHS	S		
		10000	0.0010946
		20000	0.0025289
		30000	0.0036951
		40000	0.0053029
		50000	0.0067562
		60000	0.0080853
		70000	0.0100571
		80000	0.011635
		90000	0.0128724
		100000	0.0143145
		110000	0.0157939
SHS	SS		
		10000	0.0020111
		20000	0.0040461
		30000	0.0062479
		40000	0.0132477
		50000	0.0109628
		60000	0.0136345
		70000	0.016535
		80000	0.0194976
		90000	0.021571

		100000	0.0237768
		110000	0.0265395
SHS	US		
		10000	0.002418
		20000	0.0052173
		30000	0.00785
		40000	0.0112777
		50000	0.0143152
		60000	0.0168609
		70000	0.0197327
		80000	0.0239638
		90000	0.0298033
		100000	0.0310234
		110000	0.0322673
SHS	RS		
		10000	0.0049247
		20000	0.0112556
		30000	0.018936
		40000	0.0253778
		50000	0.0351102
		60000	0.0400203
		70000	0.0516336
		80000	0.0635513
		90000	0.0738362
		100000	0.0866484
		110000	0.0807297

Таблица 5 – Время работы сортировки Шелла для разных случаев

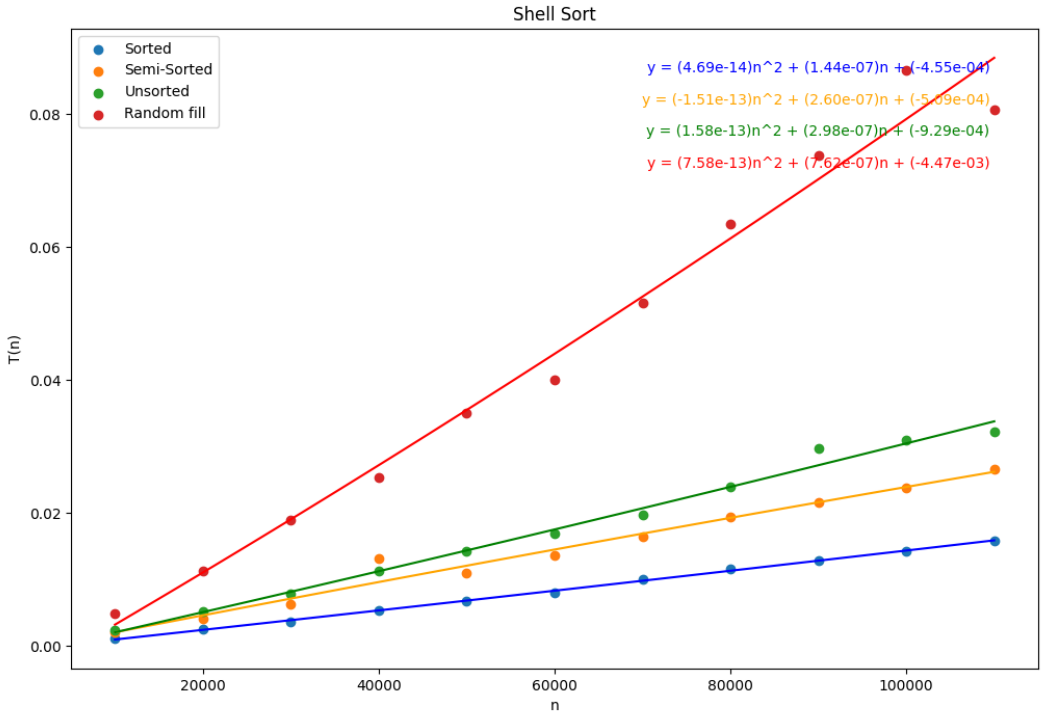

Рисунок 15 – График сортировки Шелла для всех перечисленных случаев

Сортировка Шелла (последовательность Хиббарда).

Сортировка	Заполнение	Количество элементов	Время сортировки(сек)
SHHS	S		
		10000	0.0052104
		20000	0.0022827
		30000	0.0035908
		40000	0.0054116
		50000	0.0066678
		60000	0.007797
		70000	0.0093467
		80000	0.0137566
		90000	0.0126079
		100000	0.0138994
		110000	0.0155902
SHHS	SS		
		10000	0.0018209
		20000	0.0037862
		30000	0.006346
		40000	0.008094
		50000	0.0113957
		60000	0.0132931
		70000	0.0164222
		80000	0.0191894
		90000	0.0216392

		100000	0.0235448
		110000	0.0261194
SHHS	US		
		10000	0.001821
		20000	0.0041413
		30000	0.0060275
		40000	0.0088236
		50000	0.0105209
		60000	0.0131699
		70000	0.0155768
		80000	0.0208821
		90000	0.020896
		100000	0.0234037
		110000	0.024611
SHHS	RS		
		10000	0.0044726
		20000	0.0105552
		30000	0.0183035
		40000	0.0245012
		50000	0.0320854
		60000	0.042821
		70000	0.0483647
		80000	0.0599577
		90000	0.0715037
		100000	0.0760597
		110000	0.0959209

Таблица 6 – Время работы сортировки Шелла (последовательность Хиббарда) для разных случаев

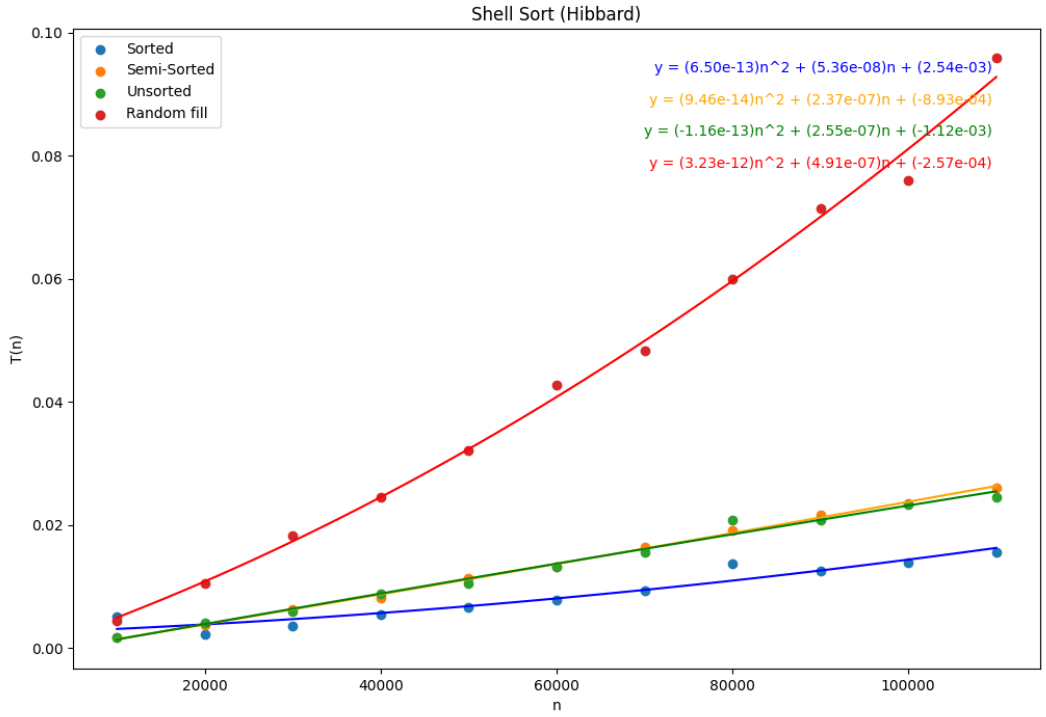

Рисунок 16 – График сортировки Шелла с последовательностью Хиббарда для всех перечисленных случаев

Сортировка Шелла (последовательность Пратта).

Сортировка	Заполнение	Количество элементов	Время сортировки(сек)
SHPS	S		
		10000	0.005883
		20000	0.0141163
		30000	0.0201087
		40000	0.0282886
		50000	0.036787
		60000	0.0454853
		70000	0.0541505
		80000	0.0642339
		90000	0.0766015
		100000	0.0815875
		110000	0.0915751
SHPS	SS		
		10000	0.0105126
		20000	0.019432
		30000	0.0358395
		40000	0.0309903
		50000	0.0422195
		60000	0.0499852
		70000	0.0649284
		80000	0.0733537

		90000	0.0850483
		100000	0.0910673
		110000	0.103484
SHPS	US		
		10000	0.0063484
		20000	0.0140099
		30000	0.0234081
		40000	0.033566
		50000	0.0410904
		60000	0.0506922
		70000	0.0615249
		80000	0.0728277
		90000	0.082572
		100000	0.095157
		110000	0.101741
SHPS	RS		
		10000	0.0086841
		20000	0.0197145
		30000	0.0316899
		40000	0.0442949
		50000	0.0579507
		60000	0.0719247
		70000	0.0891173
		80000	0.104795
		90000	0.119169
		100000	0.129355
		110000	0.148095

Таблица 7 – Время работы сортировки Шелла (последовательность Пратта) для разных случаев

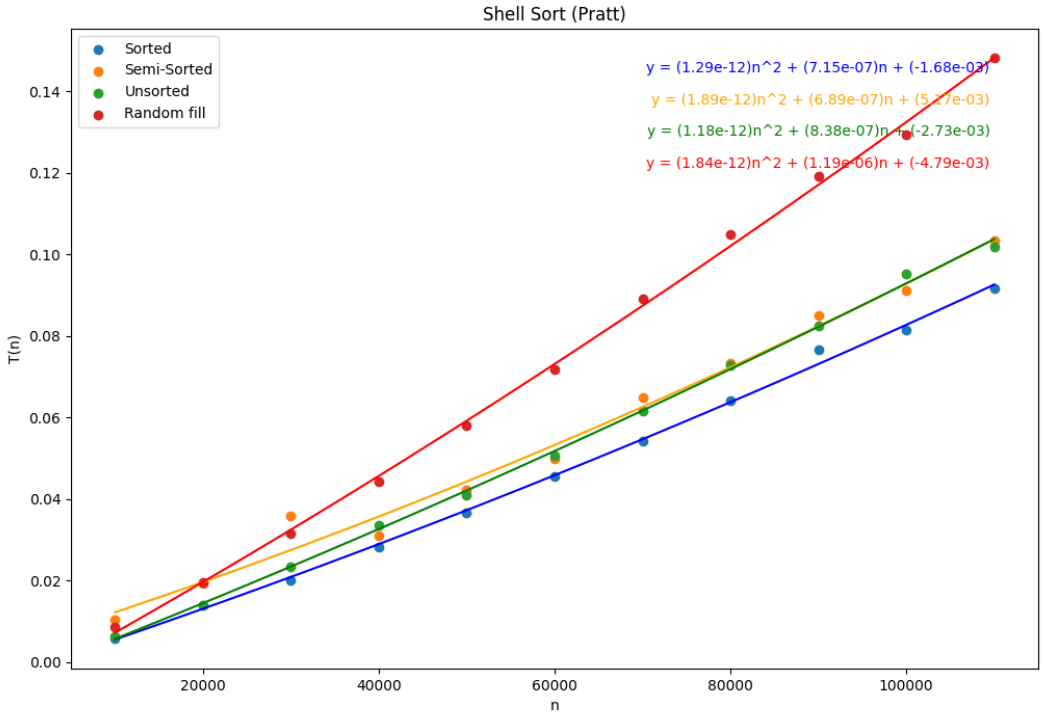

Рисунок 17 – График сортировки Шелла с последовательностью Пратта для всех перечисленных случаев

Быстрая сортировка.

Сортировка	Заполнение	Количество элементов	Время сортировки(сек)
QS	S		
		10000	0.833779
		20000	3.29521
		30000	7.46445
		40000	13.2497
		50000	20.734
		60000	29.9926
		70000	40.6216
		80000	54.2066
		90000	68.1006
		100000	82.8507
		110000	100.209
QS	SS		
		10000	0.0665388
		20000	0.264039
		30000	0.615486
		40000	1.0242
		50000	1.75992
		60000	2.49076
		70000	3.26494
		80000	4.08371

		90000	5.71823
		100000	7.0169
		110000	8.52244
QS	US		
		10000	0.504688
		20000	2.06962
		30000	4.52497
		40000	8.64818
		50000	14.6994
		60000	19.0018
		70000	25.8095
		80000	33.0761
		90000	41.1406
		100000	50.1722
		110000	60.8276
QS	RS		
		10000	0.0024275
		20000	0.0047602
		30000	0.007547
		40000	0.0108023
		50000	0.0132496
		60000	0.0163542
		70000	0.0194485
		80000	0.0231103
		90000	0.0289324
		100000	0.0275727
		110000	0.0327522

Таблица 8 – Время работы быстрой сортировки для разных случа

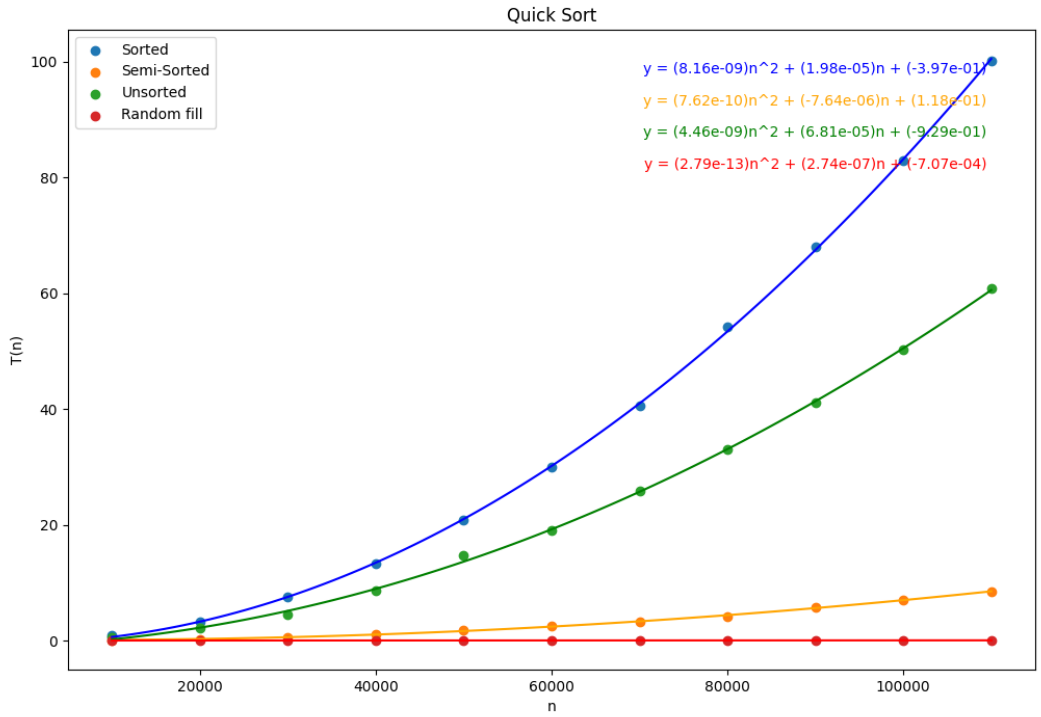

Рисунок 18 – График быстрой сортировки для всех перечисленных случаев

Пирамидальная Сортировка.

Сортировка	Заполнение	Количество элементов	Время сортировки(сек)
HS	S		
		10000	0.0041545
		20000	0.0090528
		30000	0.0141363
		40000	0.0197981
		50000	0.0259565
		60000	0.0309966
		70000	0.0360152
		80000	0.0416219
		90000	0.0479294
		100000	0.0539569
		110000	0.0594437
HS	SS		
		10000	0.0040673
		20000	0.0091211
		30000	0.0139878
		40000	0.0192672
		50000	0.0251499
		60000	0.0310341
		70000	0.0369142
		80000	0.0427841

		90000	0.0485486
		100000	0.0542335
		110000	0.0623578
HS	US		
		10000	0.0038648
		20000	0.0083953
		30000	0.0131598
		40000	0.0180874
		50000	0.0240375
		60000	0.0345999
		70000	0.033934
		80000	0.0395112
		90000	0.0449066
		100000	0.0501492
		110000	0.0583241
HS	RS		
		10000	0.0045808
		20000	0.0092505
		30000	0.0148281
		40000	0.0201365
		50000	0.026656
		60000	0.0321864
		70000	0.0366771
		80000	0.0427078
		90000	0.0494382
		100000	0.0551015
		110000	0.0613788

Таблица 9 – Время работы пирамидальной сортировки для разных случаев

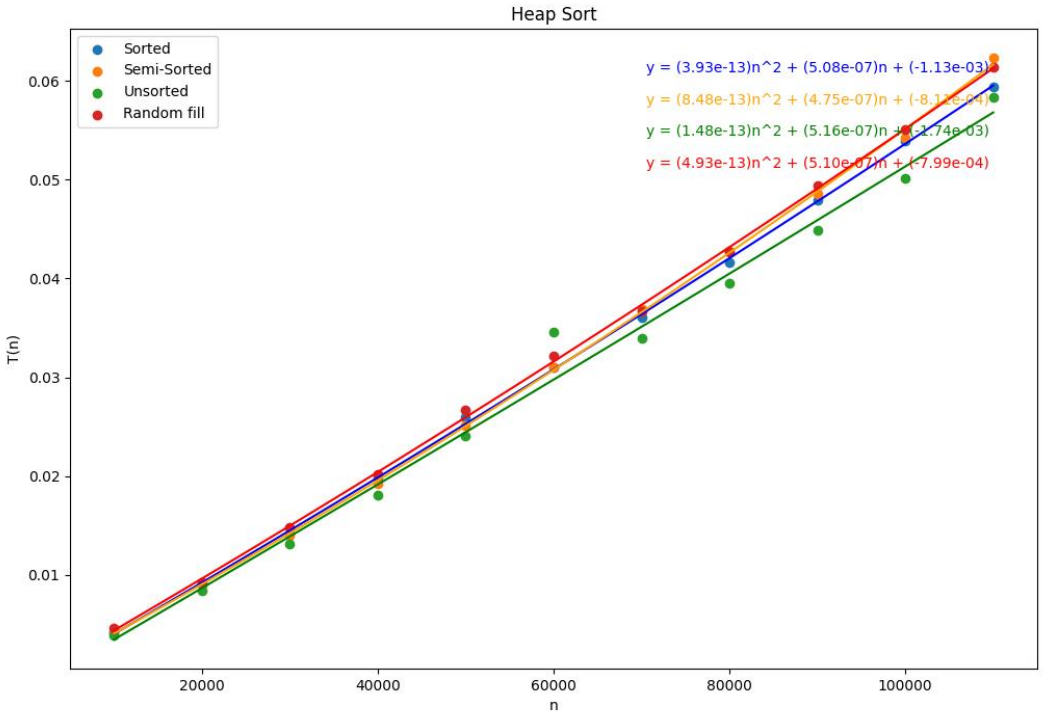

Рисунок 19 – График пирамидальной сортировки для всех перечисленных случаев

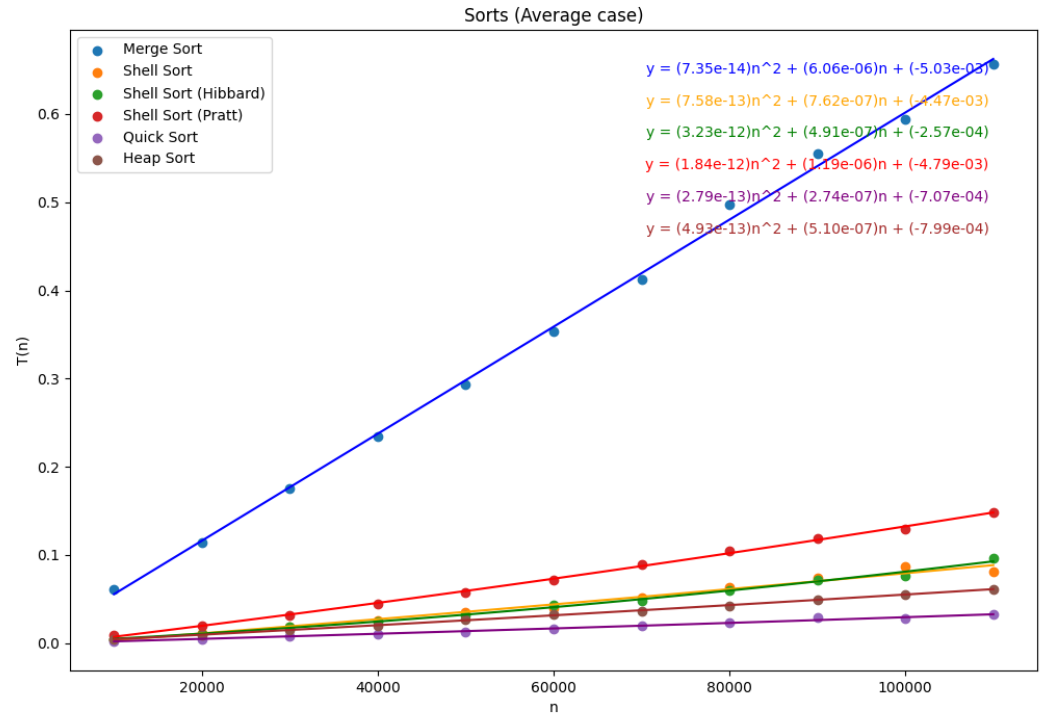

Рисунок 20 – График всех сортировок с линейной сложностью при случайном заполнении массива

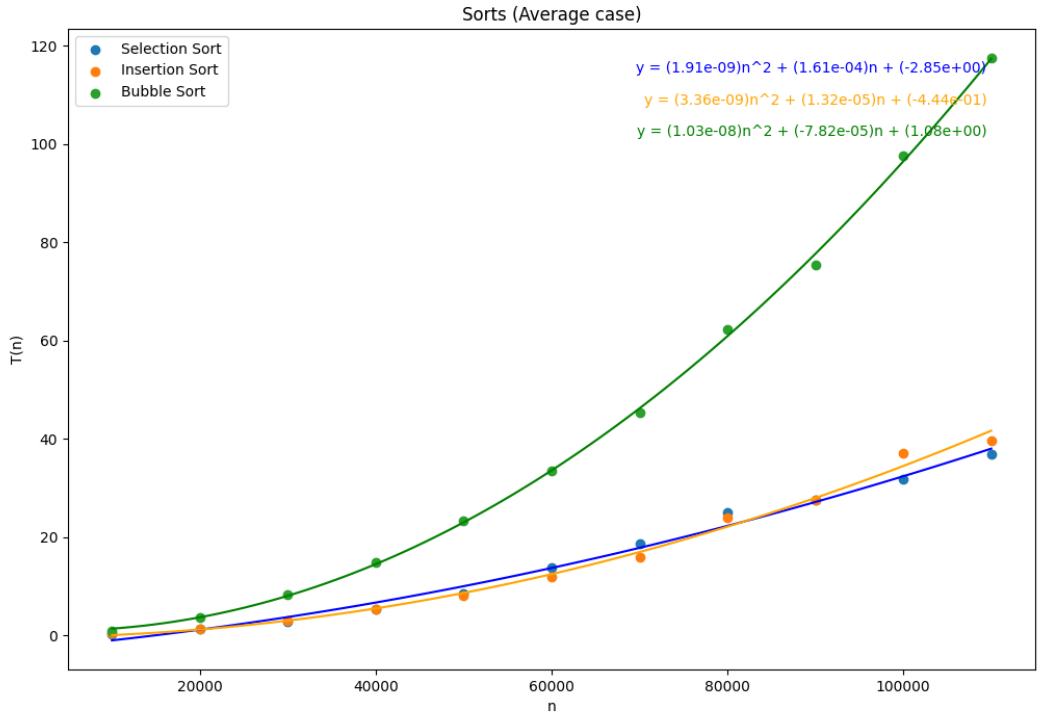

Рисунок 21 – График всех сортировок с квадратичной сложностью при случайном заполнении массива

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>
#include <random>

using namespace std;

const int RANGE = 200000; // максимальное значение элемента для случайного заполнения
const int FROM = 10000, 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) {
    for (int k = (int)(log(arr.size() + 1) / log(2)); k > 0; k--) { // Цикл по шагам s
        int s = pow(2, k) - 1;
        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 shellSortPratt(vector<int>& arr) {
    vector<int> gaps;

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

```
            gaps.push_back(j);
        }
    }
    mergeSort(gaps);
    int s = 0;
    for (int k = gaps.size() - 1; k >= 0; k--) { // Цикл по шагам s
        s = gaps[k];
        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;
            }
        }
    }
}




// 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++) {
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<> dis(0, RANGE);
        arr.push_back(dis(gen));

    }
}

// 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;
}
cout << "Selection Sort: Done\n";
// 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;
}
cout << "Insertion Sort: Done\n";
// 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;
}
cout << "Bubble Sort: Done\n";
// 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;
}
cout << "Merge Sort: Done\n";
// 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;
}
cout << "Shell Sort: Done\n";
// 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;
}
cout << "Shell Sort (Hibbard): Done\n";
// 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;
}
cout << "Shell Sort (Pratt): Done\n";
// 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;
}
cout << "Quick Sort: Done\n";
// 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;
    }
    cout << "Heap Sort: Done\n";

    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 numpy as np
import matplotlib.pyplot as plt
from sympy import *


# Границы для теоретических графиков
FROM = 10
TO = 30


#Размеры окна
X = 12
Y = 8


def plotTheor(T_w, T_b, T_a, n, title, filename, CX = FROM, CY = 30):


    p1 = plot(T_w, T_b, T_a, (n, FROM, TO), title = title, xlabel = "n", ylabel = "T(n)", size = (X, Y), label = ("Worst case", "Best case", "Average case"), axis_center = (CX, CY), show = False, legend = True)
    p1.save(filename)









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})
                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)": ' '})
        writer.writerow({"Сортировка": "Сортировка", "Заполнение": "Заполнение", "Количество
элементов" : "Количество элементов", "Время сортировки(сек)": "Время сортировки(сек)"})




def reg(n, T, col, index):
    # 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)

    a, b, c = coefficients


    # Выводим уравнение в правом верхнем углу с небольшим смещением
    # Сдвигаем уравнение по вертикали с учетом порядка кривой
    plt.text(0.95,  0.95 - 0.05 * index, f"y = ({a:.2e})n^2 + ({b:.2e})n + ({c:.2e})",
transform=plt.gca().transAxes,
             fontsize=10, color=col, ha='right', va='top')


def printGraphics(data_n, data_T, title, filename):
    plt.clf() # Очистка
    plt.figure(figsize=(X, Y))
    # 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', 0)
    reg(data_n[1], data_T[1], 'orange', 1)
    reg(data_n[2], data_T[2], 'green', 2)
    reg(data_n[3], data_T[3], 'red', 3)

    # 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
    plt.figure(figsize=(X, Y))
    # 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', 0)
    reg(data_n[1], data_T[1], 'orange' , 1)
    reg(data_n[2], data_T[2], 'green', 2)
    reg(data_n[3], data_T[3], 'red', 3)
    reg(data_n[4], data_T[4], 'purple', 4)
    reg(data_n[5], data_T[5], 'brown', 5)
    reg(data_n[6], data_T[6], 'pink', 6)
    reg(data_n[7], data_T[7], 'gray', 7)
    reg(data_n[8], data_T[8], 'y', 8)
```

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

# Сохранение графика
plt.savefig(filename)



# Отдельный вывод линейных сортировок

plt.clf() # Очистка



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')

reg(data_n[3], data_T[3], 'blue', 0)
reg(data_n[4], data_T[4], 'orange' , 1)
reg(data_n[5], data_T[5], 'green', 2)
reg(data_n[6], data_T[6], 'red', 3)
reg(data_n[7], data_T[7], 'purple', 4)
reg(data_n[8], data_T[8], 'brown', 5)

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

```
    # Сохранение графика
    plt.savefig("PracGraphics\ALL(LIN).png")

    # Отдельный вывод квадратичных сортировок
    plt.clf() # Очистка


    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')

    reg(data_n[0], data_T[0], 'blue', 0)
    reg(data_n[1], data_T[1], 'orange' , 1)
    reg(data_n[2], data_T[2], 'green', 2)


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

    # Сохранение графика
    plt.savefig("PracGraphics\ALL(S).png")




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

plotTheor(T_w, T_b, T_a, n, title, 'TheorGraphics\SS.png')



# 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
plotTheor(T_w, T_b, T_a, n, title, '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
plotTheor(T_w, T_b, T_a, n, title, '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)
plotTheor(T_w, T_b, T_a, n, title, 'TheorGraphics\MS.png', CY = 200)
# Shell Sort
title = "Shell Sort"

T_w = n ** 2
T_b = n ** 1.667
T_a = n ** 1.5
plotTheor(T_w, T_b, T_a, n, title, '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
plotTheor(T_w, T_b, T_a, n, title, 'TheorGraphics\SHHS.png', CY = 10)
# Shell Sort (Pratt)
title = "Shell Sort (Pratt)"

T_w = n * log(2, n) ** 2
T_b = n
T_a = n * log(2, n) ** 2
plotTheor(T_w, T_b, T_a, n, title, 'TheorGraphics\SHPS.png', CY = 0)

# 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


plotTheor(T_w, T_b, T_a, n, title, '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)

plotTheor(T_w, T_b, T_a, n, title, '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, axis_center = (FROM, 0), size = (X, Y))

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 более удобной таблицы для прочтения
```