**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №1**

**по дисциплине «Алгоритмы и структуры данных»**

Тема: **Алгоритмы сортировки сравнением**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3354 |  | Жаворонкова Е.С. |
| Преподаватель |  | Пестерев Д.О. |

Санкт-Петербург

2024

**Цель работы.**

Реализация алгоритмов сортировки сравнением и исследование их временной сложности.

**Основные теоретические положения.**

* **Сортировка выбором — Selection Sort**

Сортировка выбором заключается в обходе сортируемого массива, нахождения минимального элемента и вставки его в конец отсортированной части. Отсортированная часть представляет собой некоторое количество отсортированных в ходе работы элементов массива и начинается с первого элемента массива. При запуске алгоритма отсортированная часть не содержит элементов, поэтому первый минимальный элемент будет вставлен на первое место массива.

Сортировка выбором является неустойчивой сортировкой.

**Асимптотическая оценка временной сложности.** Предположим, что в массиве n элементов. Тогда для нахождения минимального элемента из всего массива необходимо пройтись по элементам всего массива — совершить n шагов. После этого этот элемент помещается в конец отсортированной части и начинается поиск нового минимального элемента, для чего необходимо (n – 1) шагов. Таким образом, сумма всех шагов будет равна: . Поскольку алгоритм будет всегда обходить весь массив, чтобы вновь найти новый элемент, то функция временной сложности для лучшего, среднего и худшего случаев Tworst case = Taverage case = Tbest case = Ɵ(n2). В случае, если массив полностью отсортирован, будет сокращаться время перестановки элементов в отсортированную часть массива, но поскольку временная сложность такого действия равна Ɵ(1), то это в значительной мере не сказывается на основной функции.

**Функция временной сложности.**

**Вывод функции пространственной сложности и ее асимптотическая оценка.** Поскольку объем дополнительного пространства памяти необходимый алгоритму остается постоянным, независимо от размера сортируемого входного массива, пространственная сложность сортировки выбором Vmax = O(1).

**График функции временной сложности для лучшего, худшего и среднего случаев.**

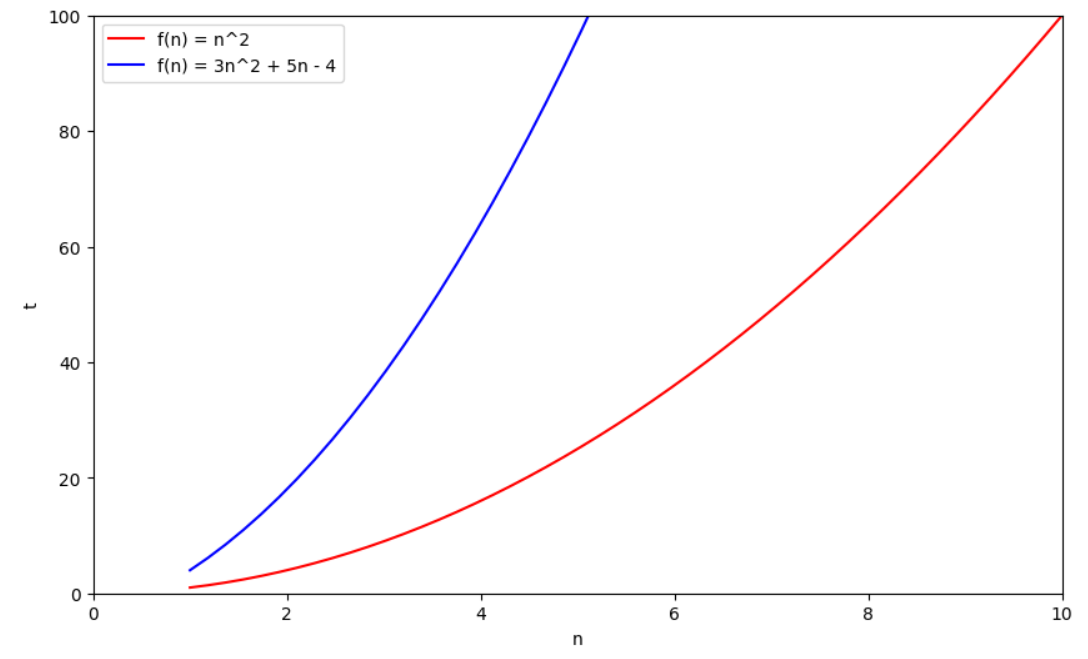


Рис. 1 — График функции временной сложности для сортировки выбором.

На рисунке 1 показаны графики временной сложности для асимптотической оценки (на рисунке обозначена красным цветом) и для функции временной сложности (на рисунке обозначена синим цветом).

* **Сортировка вставками —Insertion Sort**

Сортировка вставками заключается в сравнении первого элемента неотсортированной части поочередно (справа налево) с элементами отсортированной части и сдвигом отсортированных элементов вправо. Если сравниваемый элемент становится больше элемента слева и меньше элемента справа, то сравнение заканчивается и элемент вставляется на свою позицию. При запуске алгоритма отсортированная часть содержит первый элемент массива, так как один элемент всегда отсортирован.

Сортировка вставками является устойчивой сортировкой.

**Асимптотическая оценка временной сложности.** Предположим, что в массиве n элементов и все они расположены по убыванию. В таком случае каждому элементу из неотсортированной части придется пройти сравнение с каждым элементом из отсортированной части и встать в начало этой части. Тогда сумма всех шагов будет равна: , где i — количество сравнений в отсортированной части для рассматриваемого элемента из неотсортированной. Таким образом, время работы алгоритма в худшем случае Tworst case = Ɵ(n2). Теперь предположим, что в массиве также n элементов, только они как-то равно вероятно перемешаны. В таком случае элемент может встать на одну из следующих позиций: 0, 1, 2, …, i (где i — количество элементов в отсортированной части), а количество действий, совершенных для поиска этой позиции, будет равно: i, i-1, i-2, …, 1, 0. Следовательно, вероятность того, что элемент станет на определенную позицию равна . Тогда среднее количество действий на i-ом шаге равно: . Просуммируем полученный результат: Таким образом, время работы алгоритма в среднем Taverage case = Ɵ(n2). Для лучшего случая, а именно, когда массив отсортирован, время работы алгоритма Tbest case = Ɵ(n), поскольку в таком случае алгоритм один раз пройдется по всему массиву, каждый раз увеличивая отсортированную часть без перестановок.

**Функция временной сложности.**

Худший и средний случаи:

Лучший случай:

**Вывод функции пространственной сложности и ее асимптотическая оценка.** Поскольку объем дополнительного пространства памяти необходимый алгоритму остается постоянным, независимо от размера сортируемого входного массива, пространственная сложность сортировки вставками Vmax = O(1).

**График функции временной сложности для лучшего, худшего и среднего случаев.**

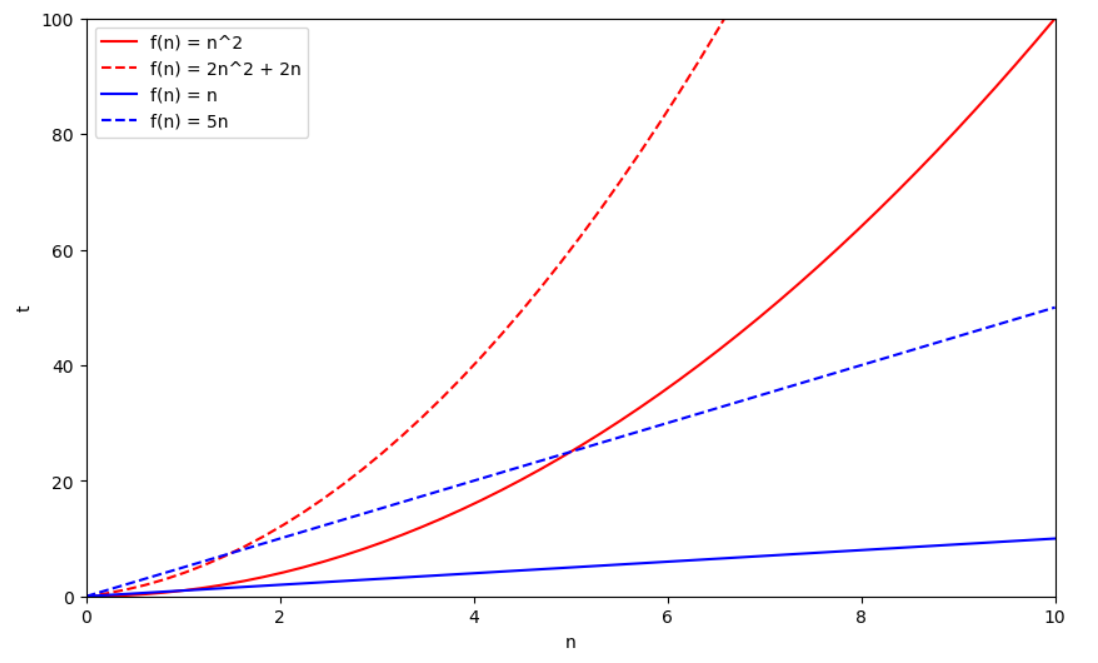


Рис. 2 — График функции временной сложности для сортировки вставками.

На рисунке 2 показаны графики временной сложности для асимптотической оценки (на рисунке обозначена линией): для худшего и среднего случаев (красный цвет) и для лучшего случая (синий цвет). А также для функции временной сложности для каждого из случаев (обозначено пунктиром, цвета соответствуют).

* **Сортировка пузырьком — Bubble Sort**

Сортировка пузырьком заключается в обходе массива и сравнении соседних элементов; если элементы стоят неправильно относительно друг друга, то они меняются местами.

Сортировка пузырьком является устойчивой сортировкой.

**Асимптотическая оценка временной сложности.** Предположим, что в массиве n элементов. Тогда количество действий равно: (n – 1), а количество сравнений равно: (n – 2). Следовательно, сумма всех шагов будет равна: . Данная оценка будет верна для худшего и среднего случаев массива, так как алгоритм будет проходить по всему массиву, пока массив не станет отсортированным. Однако, если в ходе обхода массива не было произведено ни одной перестановки, то функция временной сложности будет равна Ɵ(n) (один раз обойдется весь массив). Таким образом, функция временной сложности для среднего и худшего случаев Tworst case = Taverage case = Ɵ(n2), для лучшего случая — Tbest case = Ɵ(n).

**Функция временной сложности.**

Худший и средний случаи:

Лучший случай:

**Вывод функции пространственной сложности и ее асимптотическая оценка.** Поскольку объем дополнительного пространства памяти необходимый алгоритму остается постоянным, независимо от размера сортируемого входного массива, пространственная сложность сортировки пузырьком Vmax = O(1).

**График функции временной сложности для лучшего, худшего и среднего случаев.**

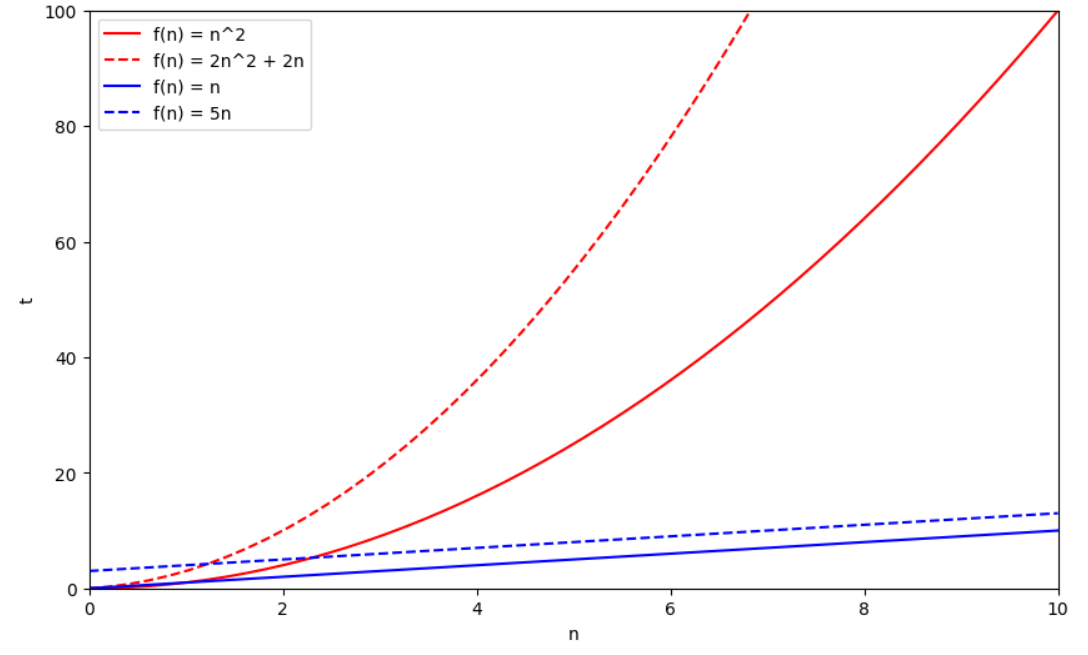


Рис. 3 — График функции временной сложности для сортировки пузырьком.

На рисунке 3 показаны графики временной сложности для асимптотической оценки (на рисунке обозначена линией): для худшего и среднего случаев (красный цвет) и для лучшего случая (синий цвет). А также для функции временной сложности для каждого из случаев (обозначено пунктиром, цвета соответствуют).

* **Сортировка слиянием — Merge Sort**

Сортировка слиянием заключается в том, что сортируемая последовательность, состоящая из n элементов, разбивается на две меньшие последовательности, каждая из которых содержит n/2 элементов. Затем каждая из частей снова сортируется методом слияния. В конце сортировки происходит слияние двух отсортированных последовательностей для получения окончательного результата. Рекурсия достигает своего нижнего предела, когда длина сортируемой последовательности становится равной 1. В этом случае вся работа уже сделана, поскольку любую такую последовательность можно считать упорядоченной.

Сортировка слиянием является устойчивой сортировкой.

**Асимптотическая оценка временной сложности.** Сортировка одного элемента методом слияния длится в течение фиксированного времени. Если n > 1, время работы распределяется таким образом:

* В ходе разбиения определяется, где находится средина подмассива. Эта операция длится фиксированное время, поэтому D (n) = Ɵ(1).
* Рекурсивно решаются две подзадачи, объем каждой из которых составляет n/2. Время решения этих подзадач равно 2T (n/2).
* Процедура MERGE в n-элементном подмассиве выполняется в течение времени Ɵ(n), поэтому C(n) = Ɵ(n).

Сложив функции D (n) и C (n), получим сумму величин Ɵ(n) и Ɵ(1) — Ɵ(n). Прибавляя к этой величине слагаемое 2T(n/2), получим рекуррентное соотношение для времени работы T(n) алгоритма сортировки по методу слияния в наихудшем случае: Для решения рекуррентного соотношения воспользуемся основной теоремой (это второй случай из основной теоремы) — . Тогда решением будет: .

**Функция временной сложности.**

\* — данное выражение отражает вспомогательную функцию слияния

Для решения данного выражения развернем рекурсию:

**Вывод функции пространственной сложности и ее асимптотическая оценка.** Поскольку для объединения отсортированных половин входного массива используется вспомогательный массив размером n, пространственная сложность сортировки слиянием Vmax = O(n).

**График функции временной сложности для лучшего, худшего и среднего случаев.**

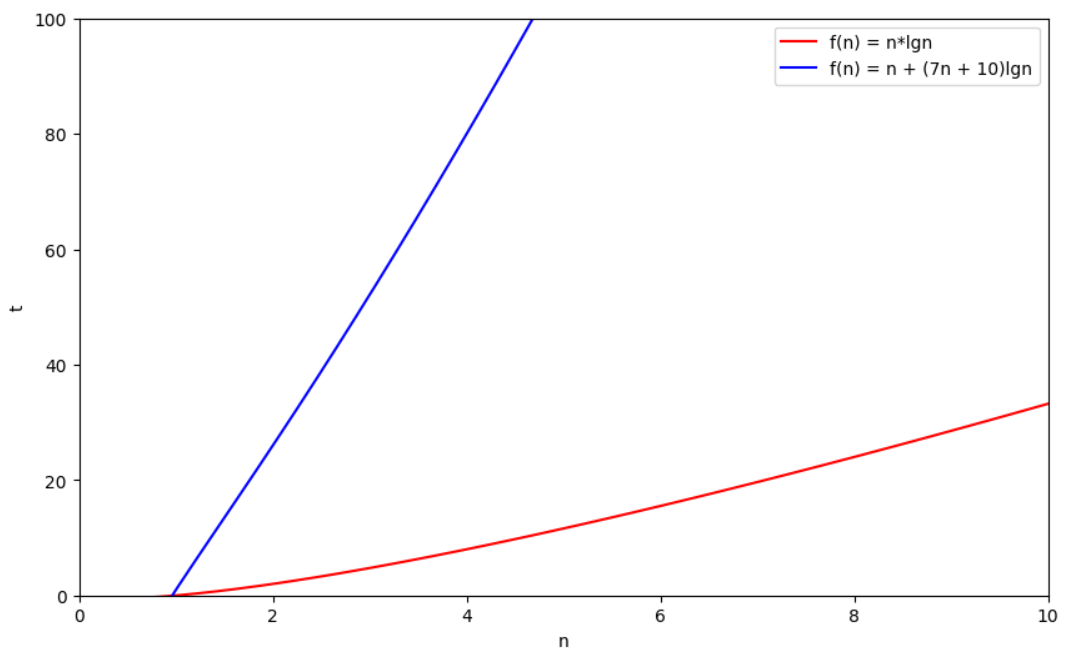


Рис. 4 — График функции временной сложности для сортировки слиянием.

На рисунке 4 показаны графики временной сложности для асимптотической оценки (на рисунке обозначена красным цветом) и для функции временной сложности.

* **Сортировка Шелла — Shell Sort**

Сортировка Шелла основана на сортировке вставками и также заключается в последовательном сравнении элементов из неотсортированной части с элементами из отсортированной части. Только в отличие от сортировки вставками сортировка Шелла имеет шаг обхода — определенное количество элементов, на расстоянии которых происходит сравнение. Для последовательности Шелла такое расстояние равно половине массива (далее этот шаг с каждым проходом уменьшается до нуля), для последовательности Хиббарда — 2k – 1, где k — индекс текущего шага, для последовательности Пратта — 2p3q.

Сортировка Шелла является неустойчивой сортировкой.

**Асимптотическая оценка и функция временной сложности.**

Последовательность Шелла:

* Худший случай — .
* Средний случай — .
* Лучший случай — .

Последовательность Хиббарда:

* Худший случай — .
* Средний случай — .
* Лучший случай — .

Последовательность Пратта:

* Худший случай — .
* Средний случай — .
* Лучший случай — .

**Вывод функции пространственной сложности и ее асимптотическая оценка.** Поскольку объем дополнительного пространства памяти необходимый алгоритму остается постоянным, независимо от размера сортируемого входного массива, пространственная сложность сортировки Шелла Vmax = O(1).

**График функции временной сложности для лучшего, худшего и среднего случаев.**

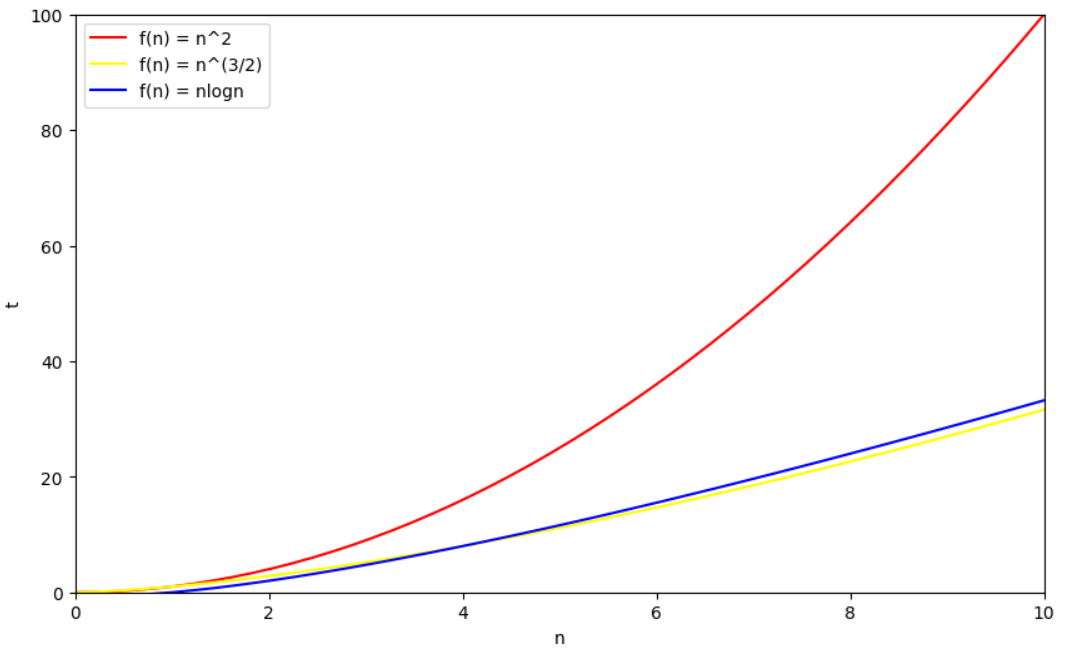


Рис. 5 — График функции временной сложности для сортировки Шелла для последовательностей Шелла.

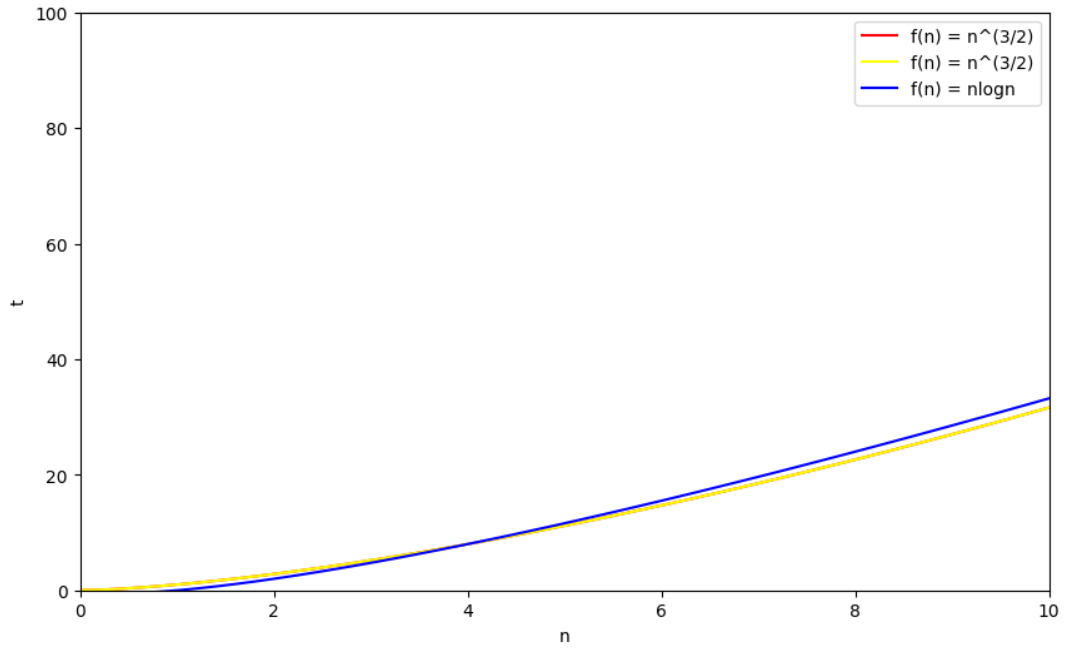


Рис. 6 — График функции временной сложности для сортировки Шелла для последовательностей Хиббарда.

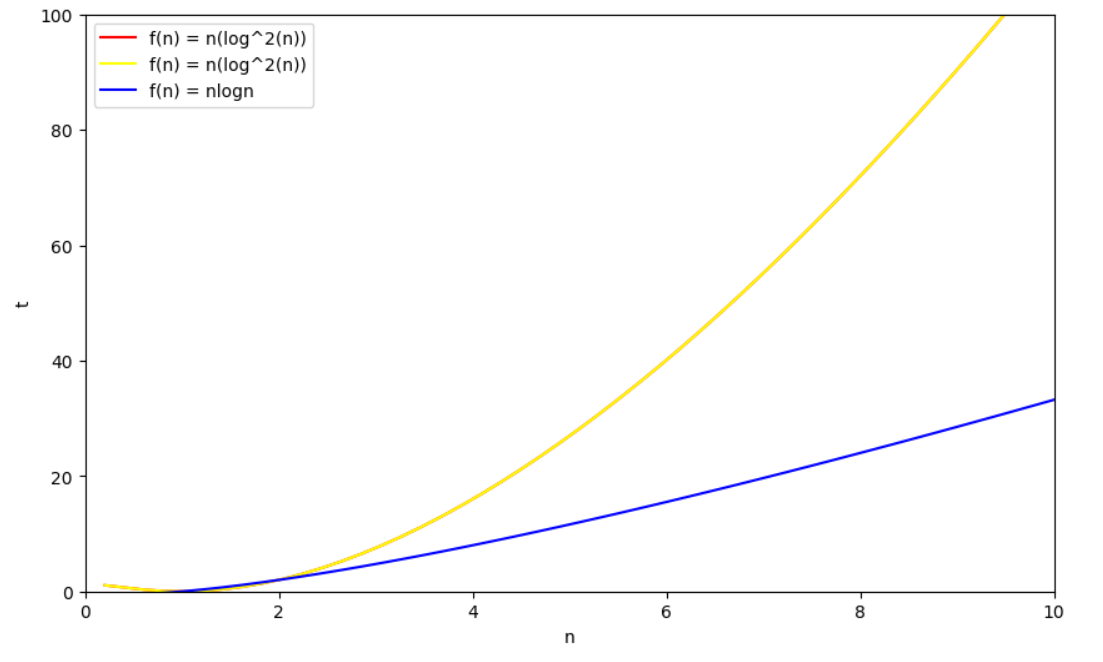


Рис. 7 — График функции временной сложности для сортировки Шелла для последовательности Пратта.

На рисунках 5-7 показаны графики временной сложности для асимптотической оценки алгоритма сортировки Шелла для лучшего (красный цвет), среднего (желтый цвет) и худшего случаев (синий цвет).

* **Быстрая сортировка — Quick Sort**

Быстрая сортировка заключается в том, чтобы поменять местами элементы массива таким образом, чтобы слева от опорного элемента стояли все элементы меньше его, а справа — большие. Таким образом, опорный элемент будет стоять на своем месте, а для подмассивов будет применен данный алгоритм.

Быстрая сортировка является неустойчивой сортировкой.

**Асимптотическая оценка временной сложности.**

1. **Для худшего случая.**

Худший случай для быстрой сортировки происходит, когда программа, выполняющая разбиение, порождает одну подзадачу с (n – 1) элементом, а вторую — с 0 элементов. Такое может произойти, если в качестве опорного элемента на каждом этапе будет выбран элемент либо наименьший, либо наибольший из всех обрабатываемых. При простейшем выборе опорного элемента — первого или последнего в массиве — такое разбиение каждый раз будет давать уже отсортированный (в прямом или обратном порядке) массив.

Для выполнения разбиения требуется время Ɵ(n). Поскольку рекурсивный вызов процедуры разбиения, на вход которой подается массив размера 0, приводит к возврату из этой процедуры без выполнения каких-либо операций, то T(0) = Ɵ(1). Таким образом, рекуррентное соотношение, описывающее время работы этой процедуры: T(n) = T(n − 1) + T(0) + Ɵ(n) = T(n − 1) + Ɵ(n) = Ɵ(n2).

1. **Для лучшего случая.**

Лучший случай для быстрой сортировки происходит, когда программа, выполняющая разбиение, порождает одну подзадачу с размером , а вторую — с размером . Тогда рекуррентное соотношение, описывающее время работы этой процедуры: . Для решения этого рекуррентного соотношения воспользуемся основной теоремой (это второй случай из основной теоремы) — . Тогда решением будет: .

1. **Для среднего случая.**

Любое разбиение, кроме описанного в худшем случае, характеризующееся конечной константой пропорциональности, приводит к образованию рекурсивного дерева высотой и временем работы каждого уровня O(n). Таким образом, при любой постоянной величине пропорции полное время выполнения составит .

**Функция временной сложности.**

Для решения рекуррентного соотношения воспользуемся основной теоремой (это второй случай из основной теоремы) — . Тогда решением будет: .

**Вывод функции пространственной сложности и ее асимптотическая оценка.** Для получения функции пространственной сложности и асимптотической оценки быстрой сортировки можно рассмотреть два случая:

* Если в ходе работы алгоритма будет происходить несбалансированное разбиение массива, как в худшем случае, то дерево рекурсии будет перекошенным, требующем стека вызовов размера O(n).
* Если в ходе работы алгоритма будет происходить сбалансированное разбиение массива, то дерево рекурсии будет сбалансированным и высота дерева (стек вызовов) будет равна .

**График функции временной сложности для лучшего, худшего и среднего случаев.**

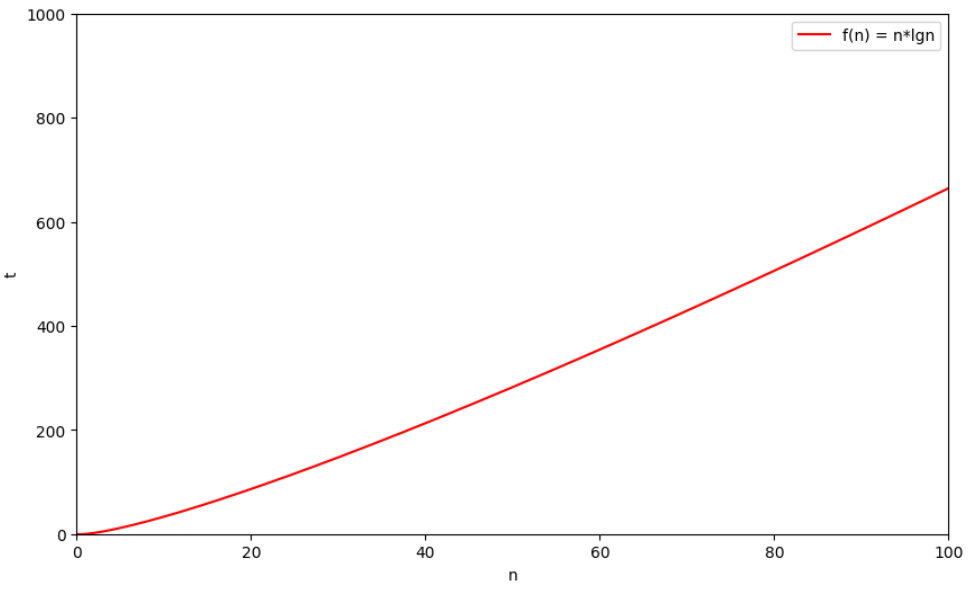


Рис. 8 — График функции временной сложности для быстрой сортировки.

На рисунке 8 показаны графики временной сложности для асимптотической оценки.

* **Пирамидальная сортировка — Heap Sort**

Пирамидальная сортировка заключается в построении почти полного бинарного дерева (пирамиды), расположив элементы на котором массив будет отсортирован. Индексация дерева идет слева на право последовательно от корня к листьям. Для реализации данной сортировки используются вспомогательные функции: для поддержания свойств пирамиды и для создания самой пирамиды.

Пирамидальная является неустойчивой сортировкой.

**Асимптотическая оценка временной сложности.**

Высота пирамиды определяется как высота его корня. Поскольку n-элементная пирамида строится по принципу полного бинарного дерева, то ее высота равна . Время выполнения основных операций в пирамиде приблизительно пропорционально высоте дерева, и, таким образом, эти операции требуют для работы время .

Время работы на поддереве размера n с корнем в заданном узле i вычисляется как время , необходимое для исправления отношений между элементами A [i], A [Lef t(i)] или A [Right(i)], плюс время работы этой процедуры с поддеревом, корень которого находится в одном из дочерних узлов узла i. Размер каждого из таких дочерних поддеревьев не превышает величину 2n/3, причем наихудший случай — это когда последний уровень заполнен наполовину. Таким образом, время работы данной процедуры описывается следующим рекуррентным соотношением: . Для решения этого рекуррентного соотношения воспользуемся основной теоремой (это второй случай из основной теоремы) — . Тогда решением будет: .

Верхнюю оценку времени работы функции для построения дерева можно получить следующим простым способом. Каждый вызов функции для поддержания свойств пирамиды (описано в предыдущем абзаце) занимает время , и всего имеется таких вызовов. Таким образом, время работы алгоритма равно .

Таким образом, время работы пирамидальной сортировки Tworst case = Taverage case = Tbest case = .

**Вывод функции пространственной сложности и ее асимптотическая оценка.** Поскольку объем дополнительного пространства памяти необходимый алгоритму остается постоянным, независимо от размера сортируемого входного массива, пространственная сложность пирамидной сортировки Vmax = O(1).

**График функции временной сложности для лучшего, худшего и среднего случаев.**

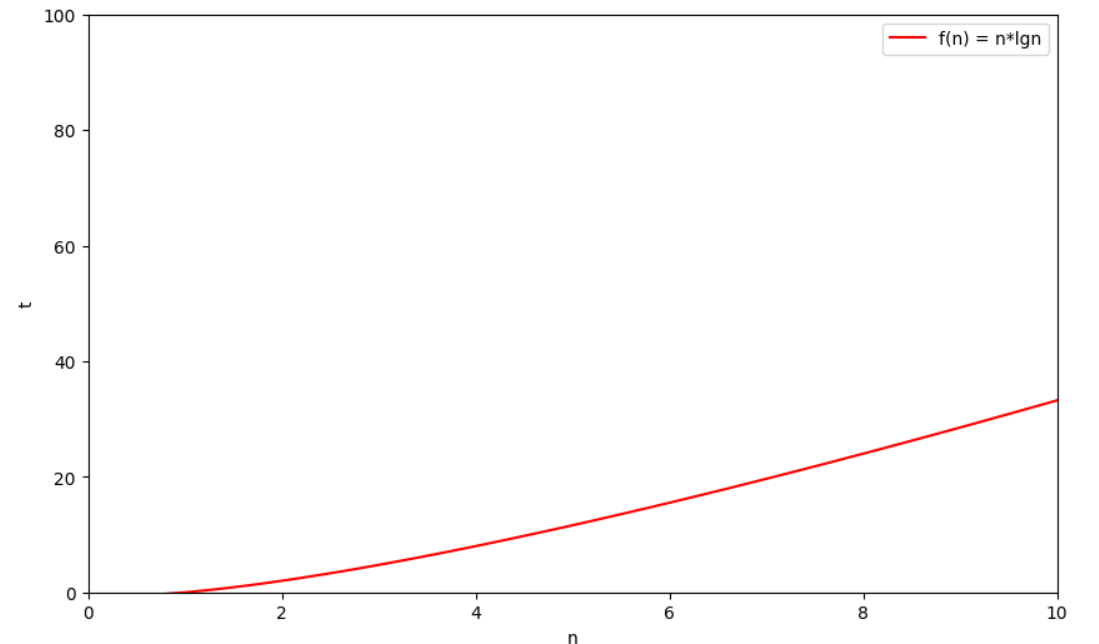


Рис. 9 — График функции временной сложности для пирамидальной сортировки.

На рисунке 9 показан график временной сложности для асимптотической оценки.

**Асимптотическая временная и пространственная сложности для каждой из сортировок.**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Сортировка** | | | **Tworst case** | | **Taverage case** | | **Tbest case** | | **Vmax** | |
| Выбором | | |  | | | | | |  | |
| Вставками | | |  | | | |  | |  | |
| Пузырьком | | |  | | | |  | |  | |
| Слиянием | | |  | | | | | |  | |
| Шелла | Последовательность Шелла |  | |  | |  | |  | |
| Последовательность Хиббарда |  | | | |  | |
| Последовательность Пратта |  | | | |  | |
| Быстрая | | |  | |  | | | |  | |
| Пирамидальная | | |  | | | | | |  | |

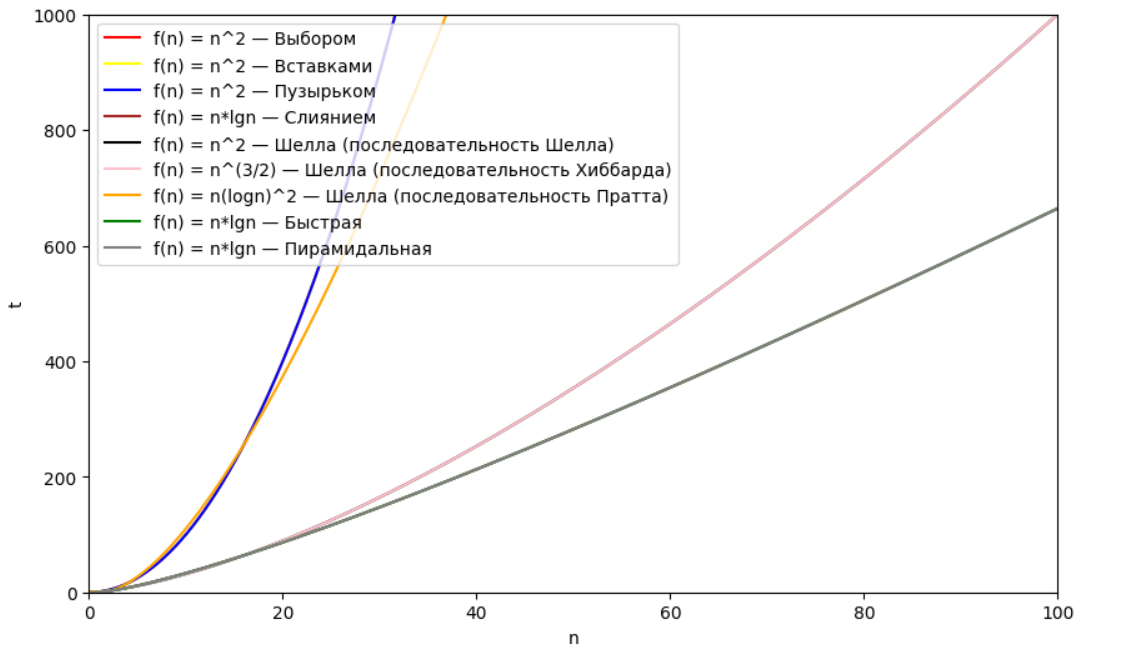
****

Рис. 10 — График всех рассматриваемых сортировок для средних случаев.

**Предположение о наибыстрейшей сортировке в среднем случае при размере массива более 100 000 элементов.** Как видно из графика на рисунке 10, самой быстрой сортировкой в среднем случае на большом массиве являются сортировки: пирамидальная, быстрая и слиянием.

**Экспериментальные результаты.**

* **Экспериментальная оценка временной сложности для отсортированного, почти отсортированного, отсортированного в сторону убывания и случайного массива.**

**Сортировка выбором.**

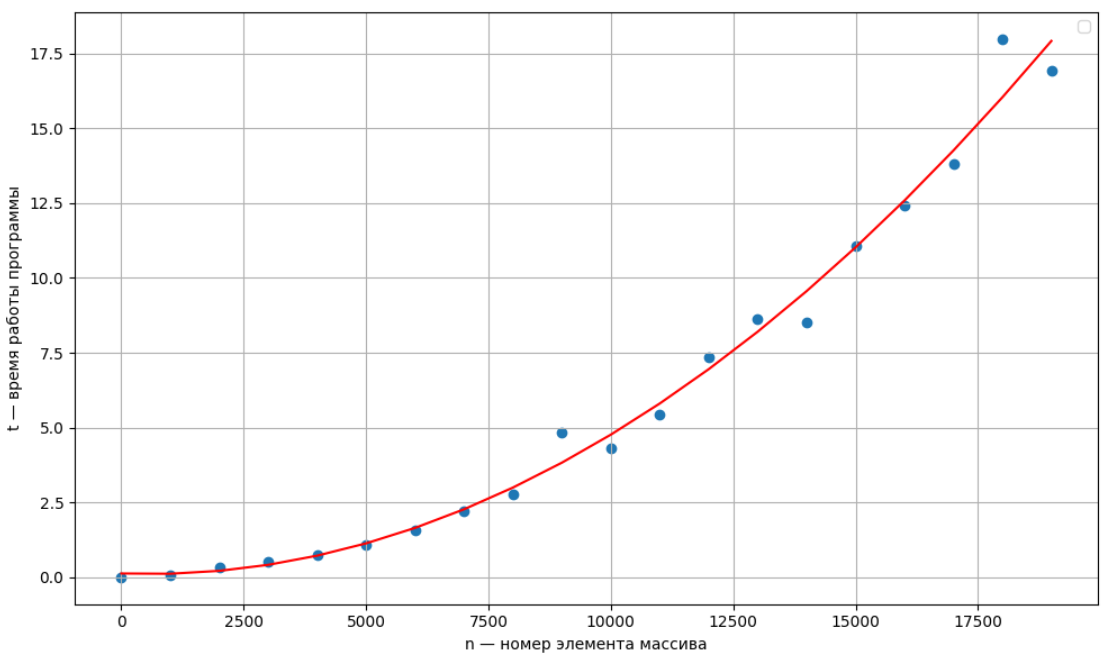


Рис. 11 — График экспериментальных данных для отсортированного массива и регрессионная кривая (4.797e-08x2 + 1.02e-05x - 0.004455).

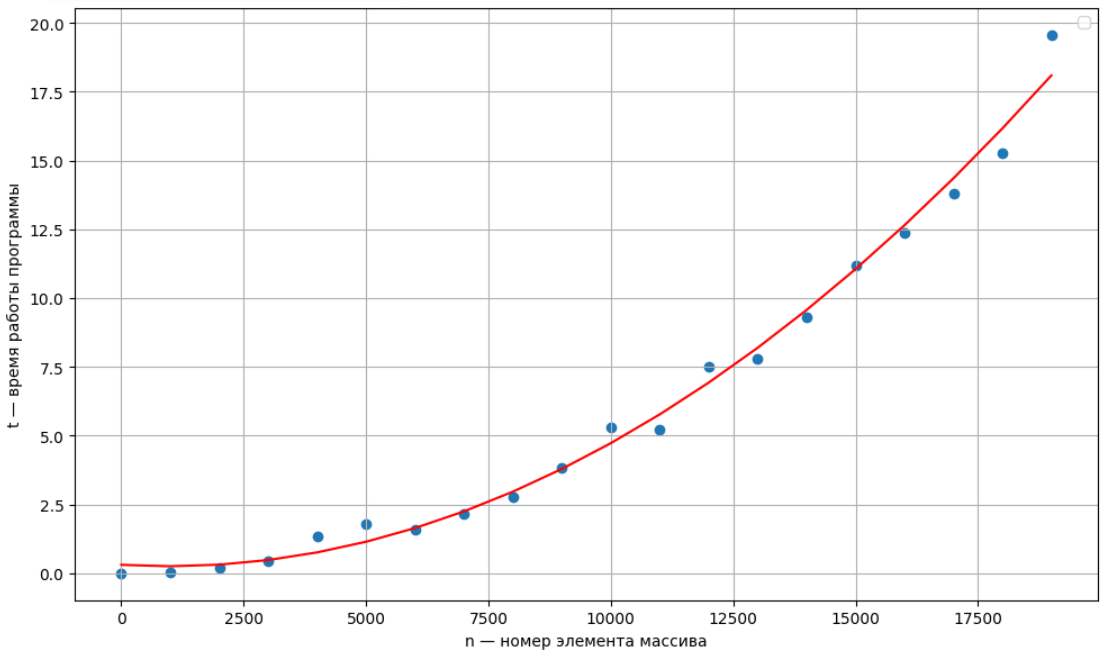


Рис. 12 — График экспериментальных данных для почти отсортированного массива и регрессионная кривая (5.126e-08x2 - 4.287e-05x + 0.1013).

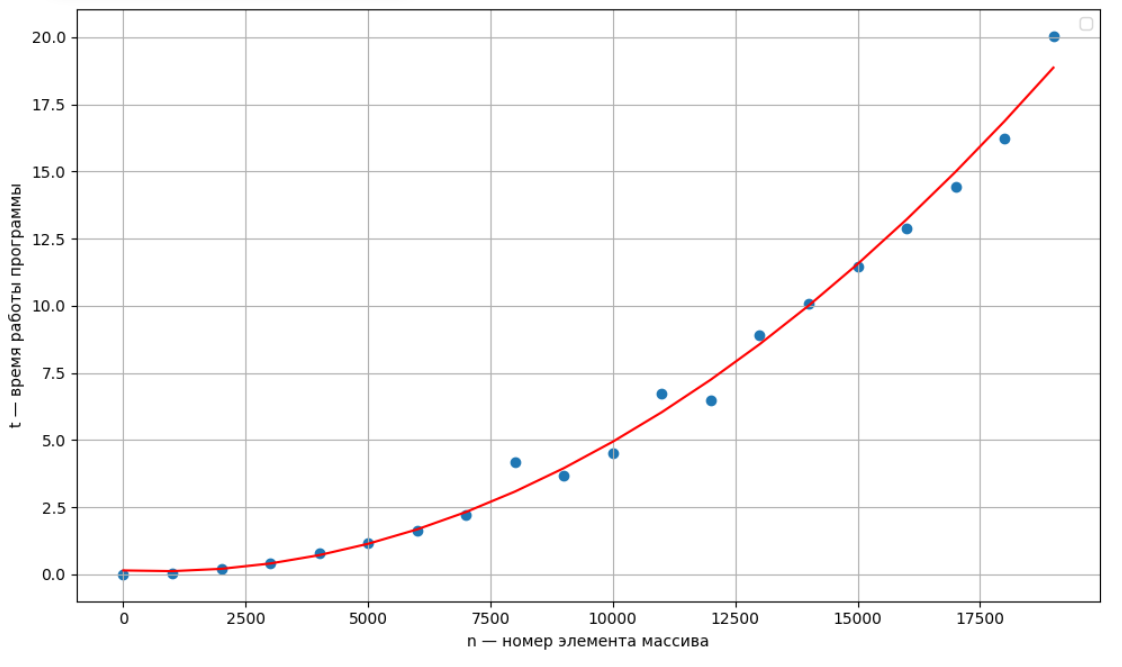


Рис. 13 — График экспериментальных данных для отсортированного в сторону убывания массива и регрессионная кривая (5.375e-08x2 - 7.476e-05x + 0.3129).

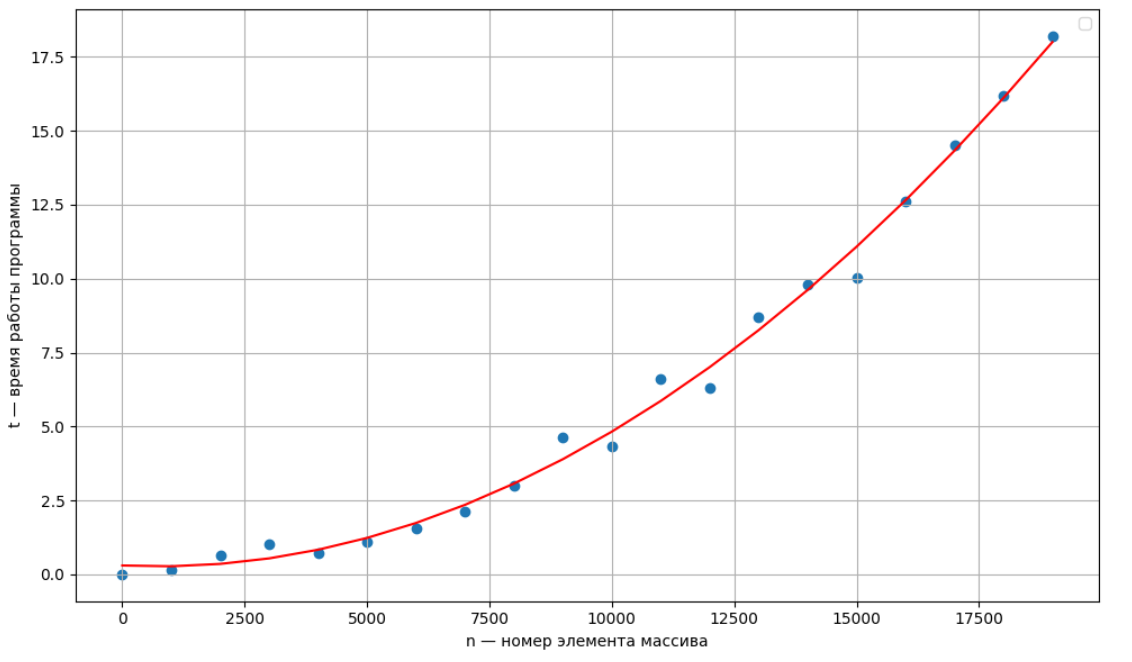


Рис. 14 — График экспериментальных данных для случайного массива и регрессионная кривая (4.942e-08x2 - 5.519e-07x - 0.005412).

**Таблица результатов измерений для сортировки выбором.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов в массиве | Отсортированный массив | Почти отсортированный массив | Отсортированный по убыванию массив | Случайный массив |
| 1 000 | 0.08016 | 0.04832 | 0.04589 | 0.14526 |
| 5 000 | 1.09429 | 1.77381 | 1.14218 | 1.10196 |
| 10 000 | 4.32557 | 5.28150 | 4.51067 | 4.34062 |
| 15 000 | 11.05996 | 11.19048 | 11.46145 | 10.03695 |
| 20 000 | 16.91212 | 19.56069 | 20.03704 | 21.47550 |

По приведенным выше примерам можно сделать вывод о том, что в сортировке выбором вне зависимости от отсортированности входного массива время работы остается приблизительно одинаковым. При увеличении количества элементов массива, возрастает время выполнения сортировки. Данный вывод говорит о том, что теоретическое описание подтверждается практической частью.

**Сортировка вставками.**

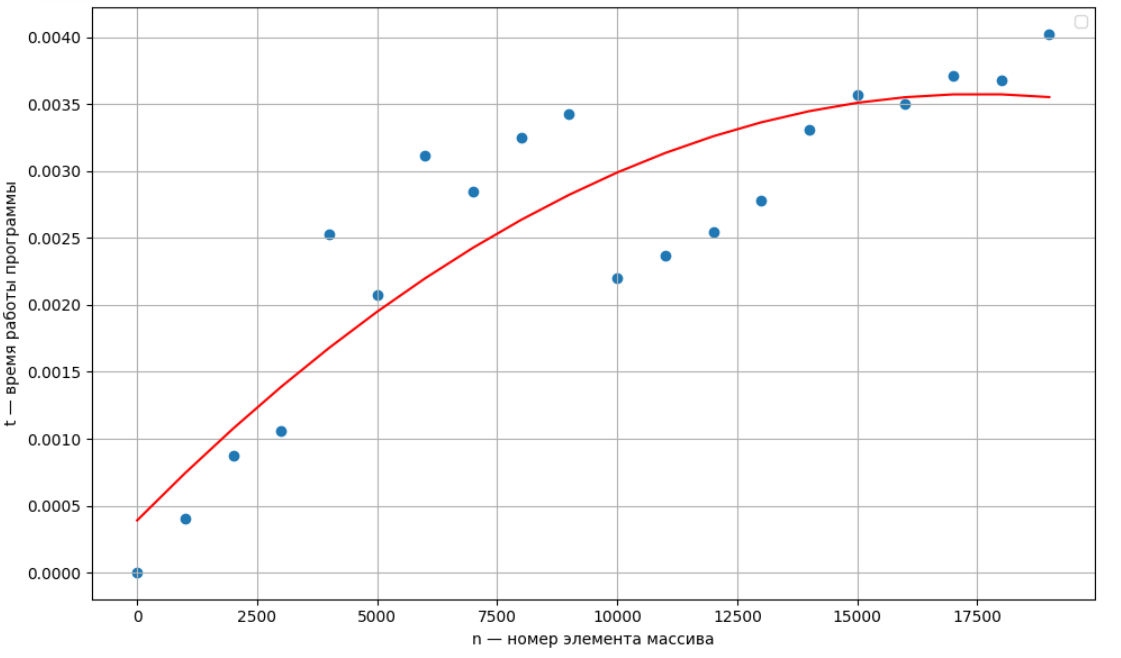


Рис. 15 — График экспериментальных данных для отсортированного массива и регрессионная кривая (5.07e-12x2 + 9.123e-08x + 0.0007764).

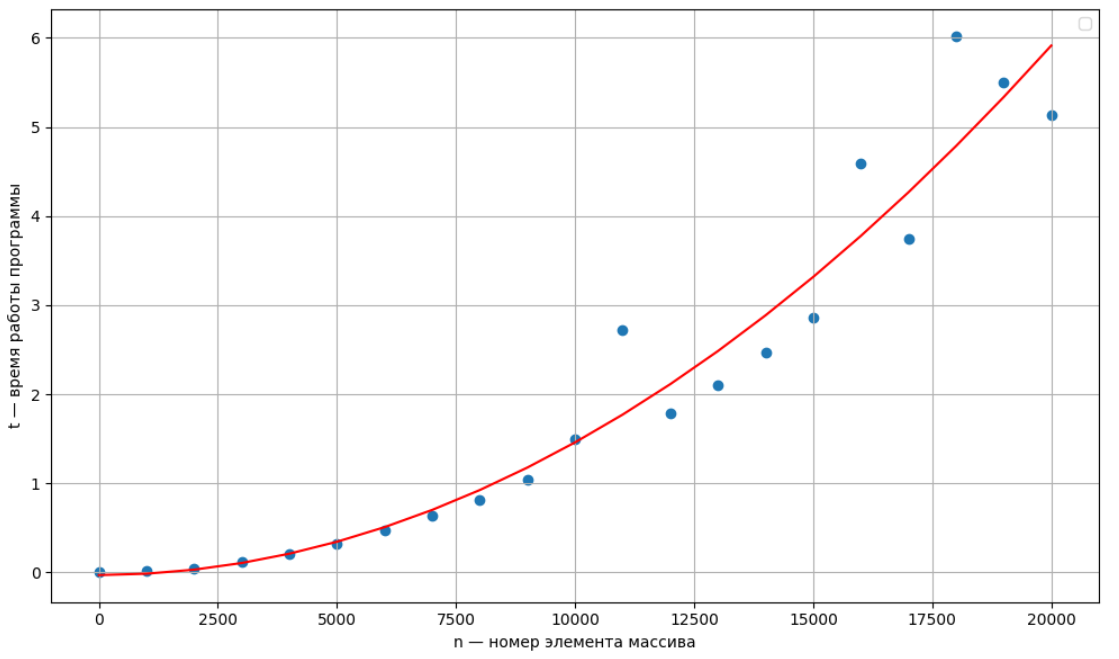


Рис. 16 — График экспериментальных данных для почти отсортированного массива и регрессионная кривая (1.612e-08x2 - 4.533e-05x + 0.1532).

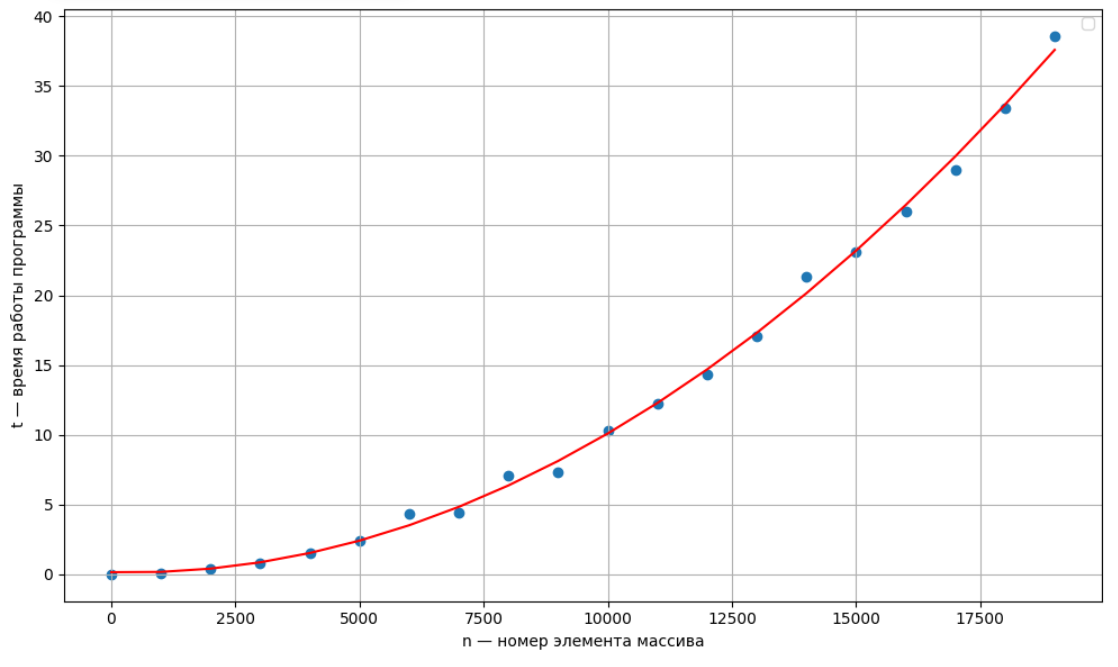


Рис. 17 — График экспериментальных данных для отсортированного в сторону убывания массива и регрессионная кривая (1.086e-07x2 - 0.0001378x + 0.6094).

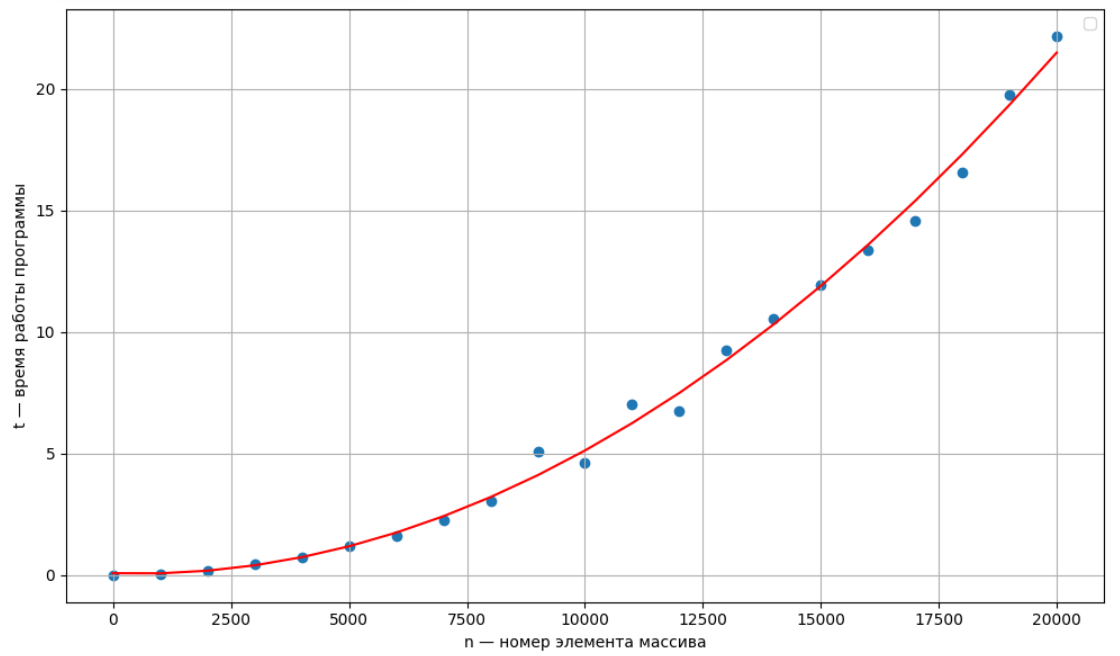


Рис. 18 — График экспериментальных данных для случайного массива и регрессионная кривая (5.39e-08x2 - 6.061e-05x + 0.2364).

**Таблица результатов измерений для сортировки вставками.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов в массиве | Отсортированный массив | Почти отсортированный массив | Отсортированный по убыванию массив | Случайный массив |
| 1 000 | 0.00040 | 0.01506 | 0.08584 | 0.04917 |
| 5 000 | 0.00207 | 0.32454 | 2.42784 | 1.18087 |
| 10 000 | 0.00220 | 1.49208 | 10.28489 | 4.63800 |
| 15 000 | 0.00357 | 2.85613 | 23.12821 | 11.93658 |
| 20 000 | 0.00402 | 5.13079 | 38.57710 | 22.15831 |

По приведенным выше примерам можно сделать вывод о том, что в сортировке вставками при полностью отсортированном массиве время, затраченное на обработку, значительно ниже, чем при других рассмотренных случаях отсортированности массива. При увеличении количества элементов массива, возрастает время выполнения сортировки. Данный вывод говорит о том, что теоретическое описание подтверждается практической частью.

**Сортировка пузырьком.**

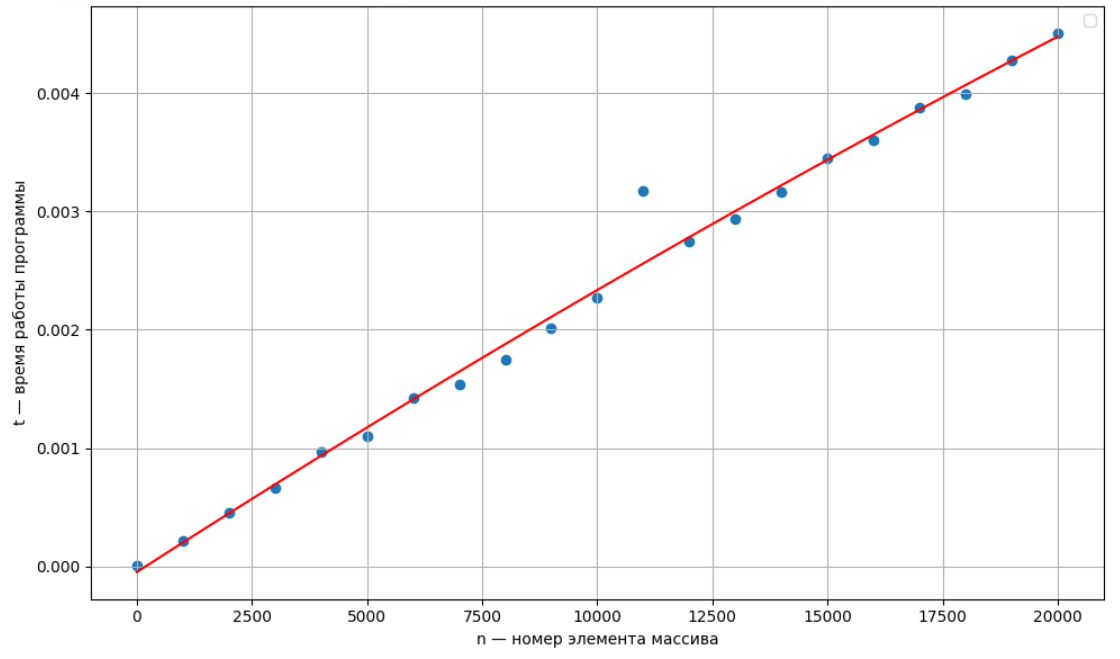


Рис. 19 — График экспериментальных данных для отсортированного массива и регрессионная кривая (3.199e-12x2 + 6.828e-08 x + 0.0003363).

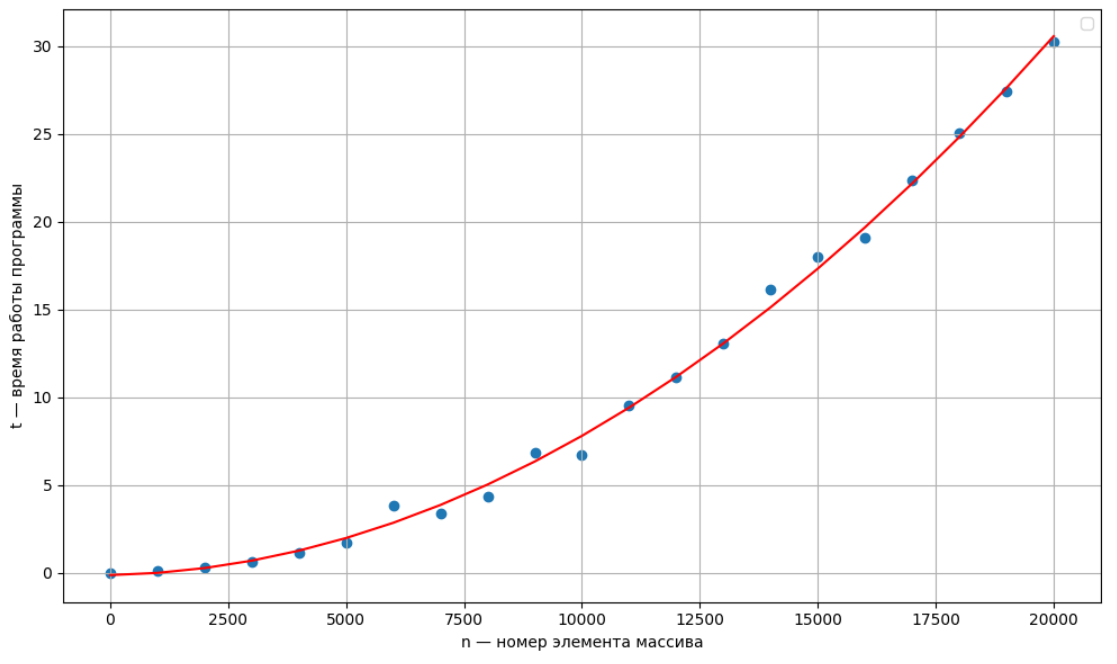


Рис. 20 — График экспериментальных данных для почти отсортированного массива и регрессионная кривая (8.145e-08x2 - 9.255e-05x + 0.3575).

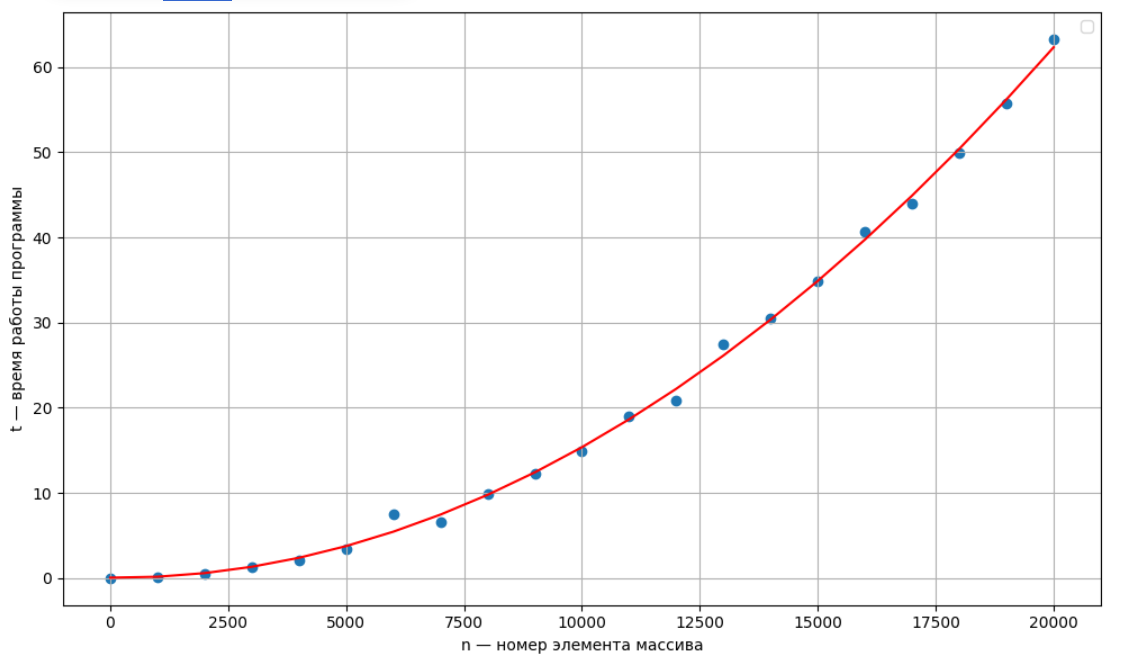


Рис. 21 — График экспериментальных данных для отсортированного в сторону убывания массива и регрессионная кривая (1.33e-07x2 + 0.0007304x - 2.724).

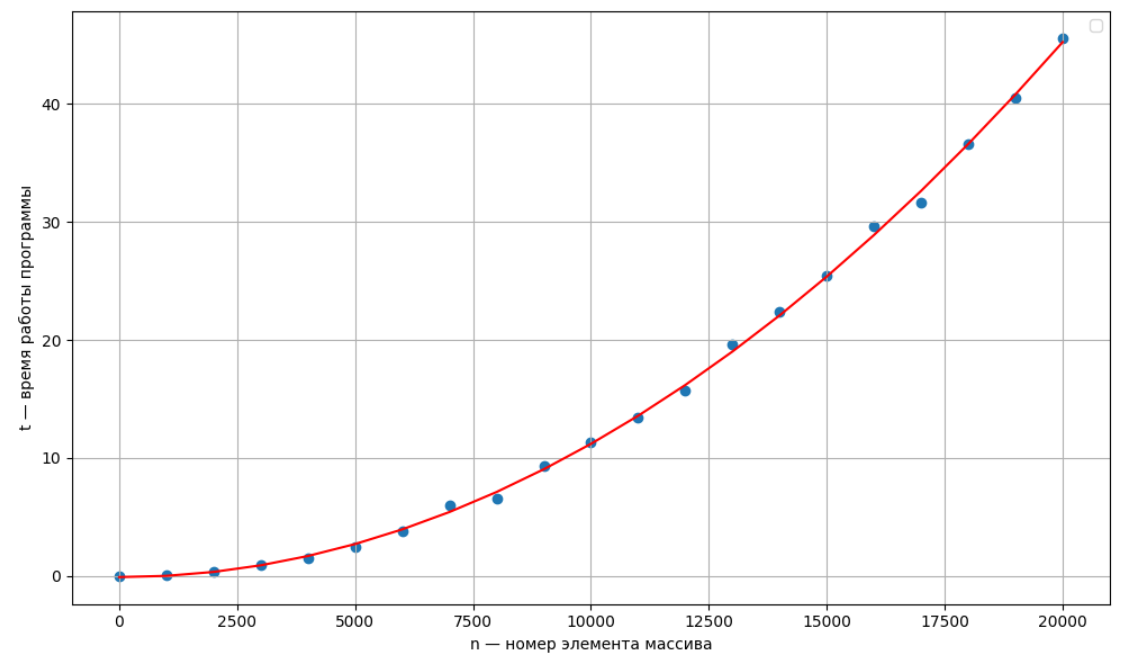


Рис. 22 — График экспериментальных данных для случайного массива и регрессионная кривая (1.133e-07x2 + 7.846e-06x - 0.08525).

**Таблица результатов измерений для сортировки пузырьком.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов в массиве | Отсортированный массив | Почти отсортированный массив | Отсортированный по убыванию массив | Случайный массив |
| 1 000 | 0.00021 | 0.11300 | 0.12433 | 0.09431 |
| 5 000 | 0.00109 | 1.69624 | 3.37536 | 2.45159 |
| 10 000 | 0.00227 | 6.70952 | 14.90345 | 11.30563 |
| 15 000 | 0.00344 | 18.00013 | 34.80971 | 25.41137 |
| 20 000 | 0.00450 | 30.25337 | 63.27321 | 45.54581 |

По приведенным выше примерам можно сделать вывод о том, что в сортировке пузырьком при полностью отсортированном массиве время, затраченное на обработку, значительно ниже, чем при других рассмотренных случаях отсортированности массива. При увеличении количества элементов массива, возрастает время выполнения сортировки. Данный вывод говорит о том, что теоретическое описание подтверждается практической частью.

**Сортировка слиянием.**

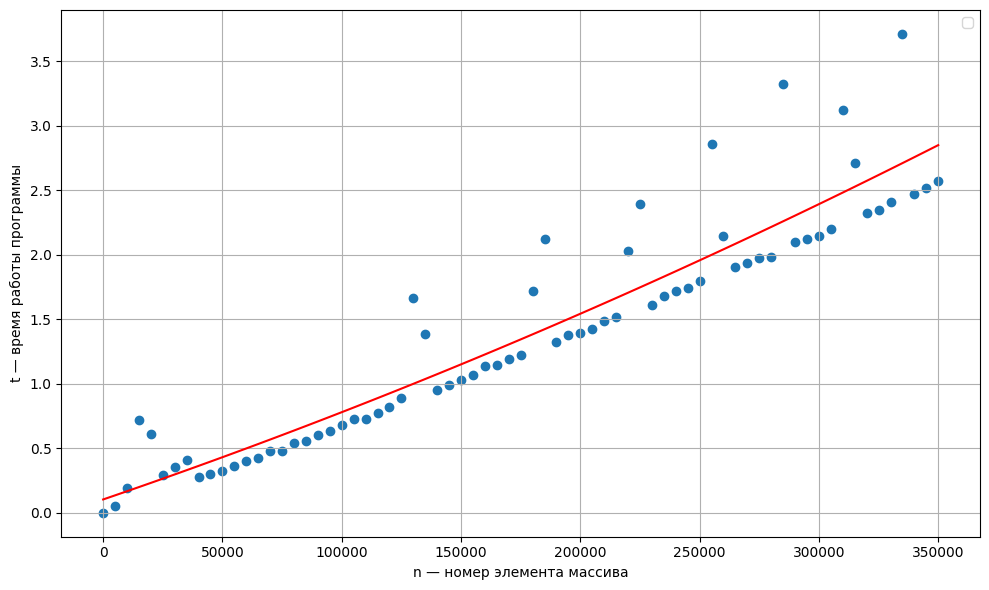


Рис. 23 — График экспериментальных данных для отсортированного массива и регрессионная кривая (4.333e-12x2 + 6.335e-06x + 0.1033).

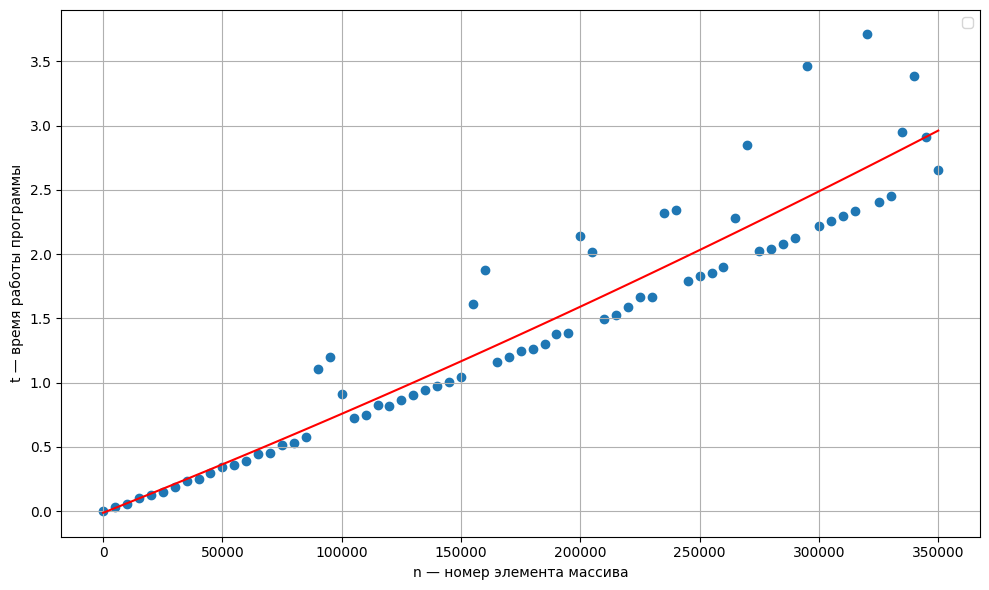


Рис. 24 — График экспериментальных данных для почти отсортированного массива и регрессионная кривая (3.186e-12x2 + 7.381e-06x - 0.013).

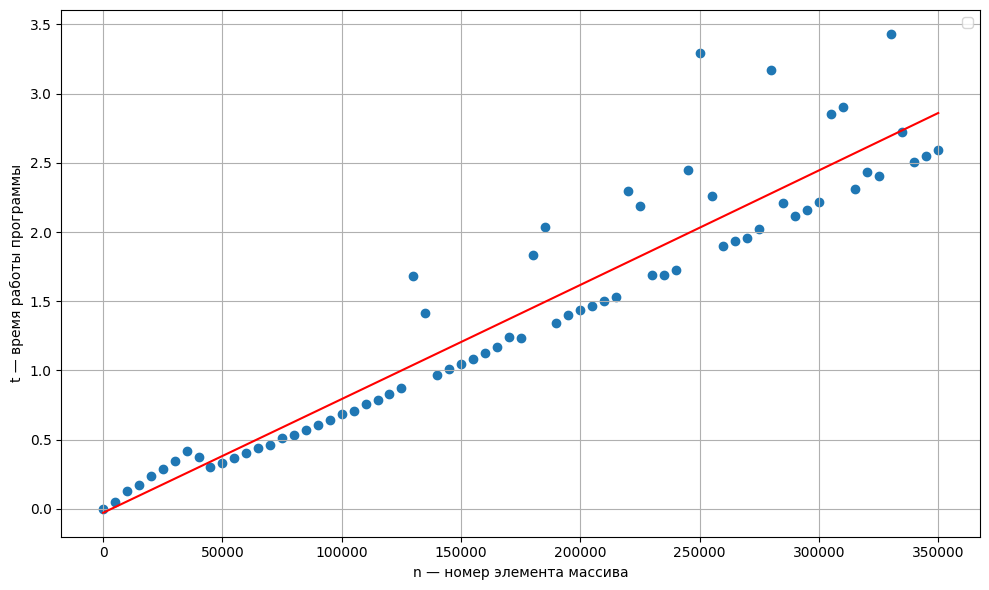


Рис. 25 — График экспериментальных данных для отсортированного в сторону убывания массива и регрессионная кривая (1.41e-13x2 + 8.203e-06x - 0.02847).

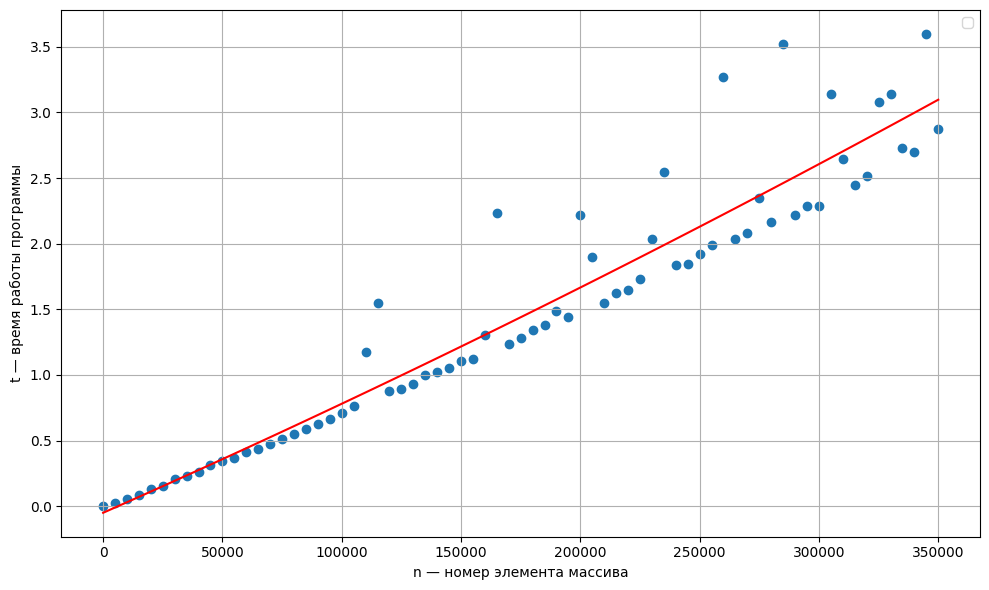


Рис. 26 — График экспериментальных данных для случайного массива и регрессионная кривая (2.737e-12x2 + 8.033e-06x - 0.04979).

**Таблица результатов измерений для сортировки слиянием.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов в массиве | Отсортированный массив | Почти отсортированный массив | Отсортированный по убыванию массив | Случайный массив |
| 5 000 | 0.04871 | 0.03503 | 0.05090 | 0.02630 |
| 10 000 | 0.19417 | 0.05665 | 0.12584 | 0.05465 |
| 15 000 | 0.71849 | 0.09915 | 0.17185 | 0.08618 |
| 20 000 | 0.60885 | 0.12474 | 0.23522 | 0.13226 |
| 25 000 | 0.29265 | 0.15040 | 0.29012 | 0.15198 |
| 50 000 | 0.32107 | 0.34041 | 0.32802 | 0.34201 |
| 100 000 | 0.68055 | 0.90938 | 0.68334 | 0.70997 |
| 200 000 | 1.39610 | 2.14276 | 1.43672 | 2.21789 |
| 350 000 | 2.57658 | 2.65581 | 2.59042 | 2.87777 |

По приведенным выше примерам можно сделать вывод о том, что в сортировке слиянием вне зависимости от отсортированности входного массива время работы остается приблизительно одинаковым. При увеличении количества элементов массива, возрастает время выполнения сортировки. Данный вывод говорит о том, что теоретическое описание подтверждается практической частью.

**Сортировка Шелла (последовательность Шелла).**

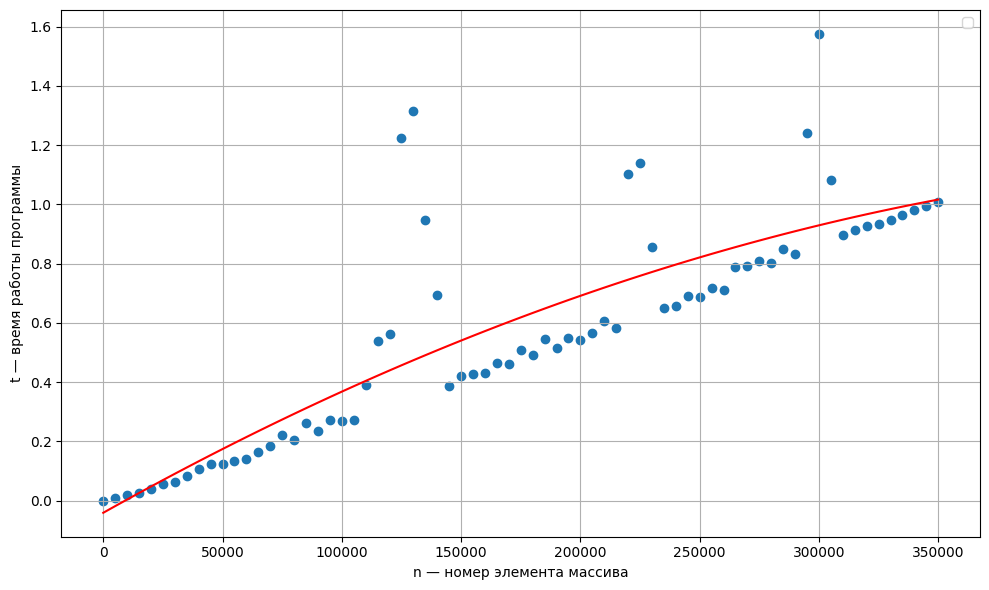


Рис. 27 — График экспериментальных данных для отсортированного массива и регрессионная кривая (-4.273e-12x2 + 4.516e-06x - 0.04067).

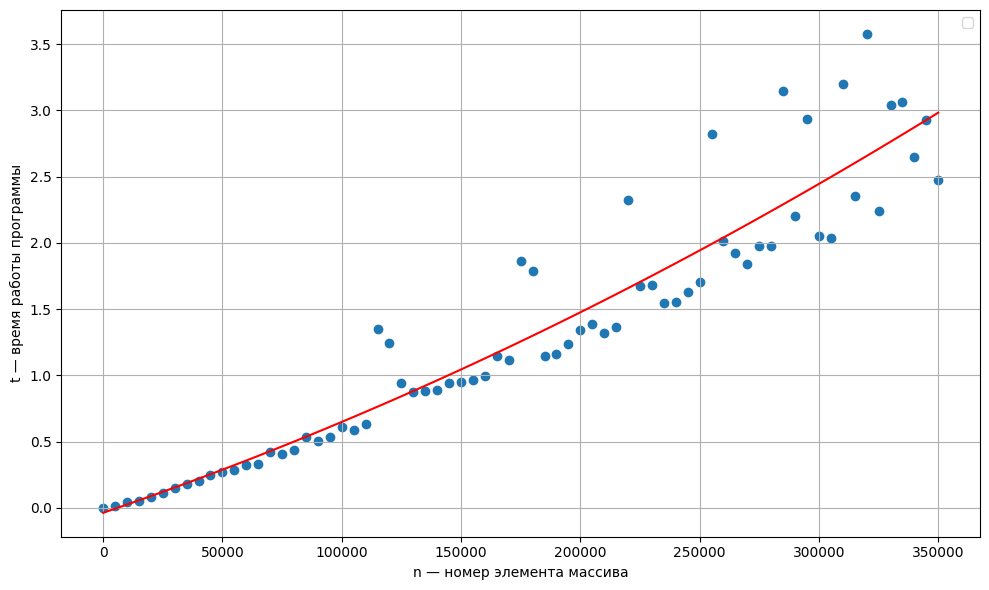


Рис. 28 — График экспериментальных данных для почти отсортированного массива и регрессионная кривая (7.088e-12x2 + 6.152e-06x - 0.0378).

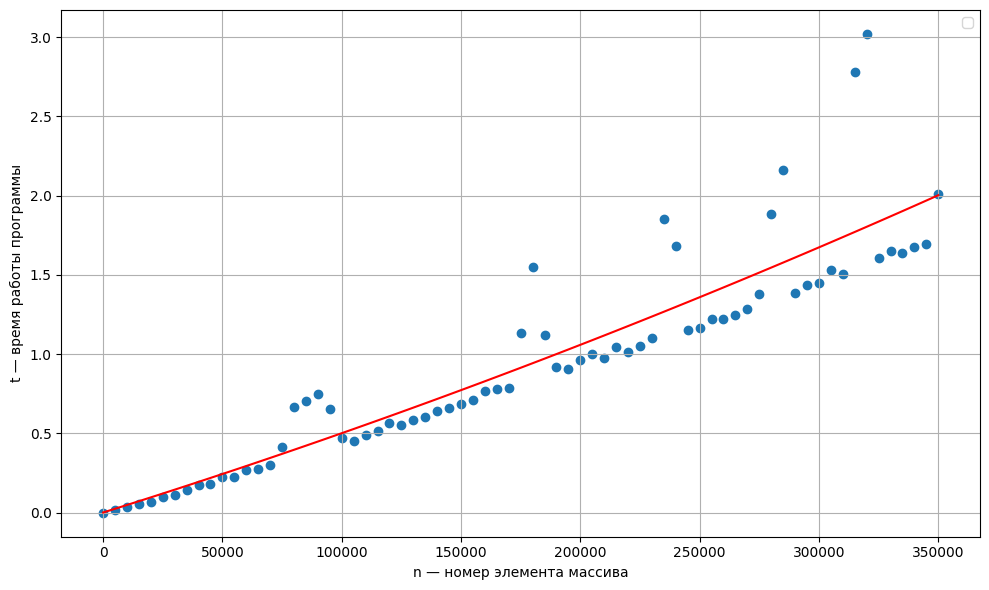


Рис. 29 — График экспериментальных данных для отсортированного в сторону убывания массива и регрессионная кривая (2.86e-12x2 + 4.715e-06x + 0.001765).

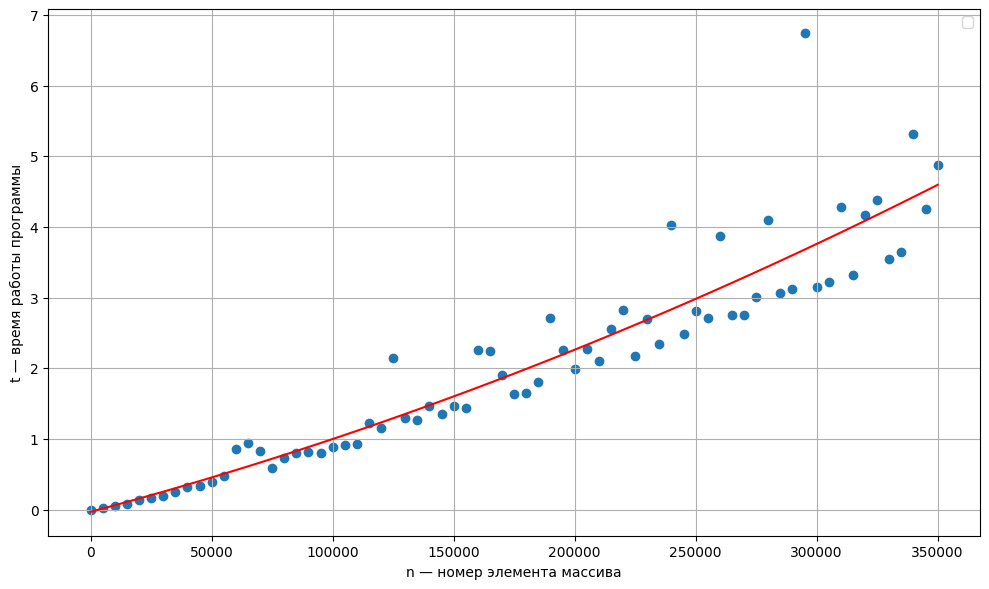


Рис. 30 — График экспериментальных данных для случайного массива и регрессионная кривая (1.169e-11x2 + 9.136e-06x - 0.03232).

**Таблица результатов измерений для сортировки Шелла (последовательность Шелла).**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов в массиве | Отсортированный массив | Почти отсортированный массив | Отсортированный по убыванию массив | Случайный массив |
| 5 000 | 0.00993 | 0.01608 | 0.01650 | 0.02259 |
| 10 000 | 0.01847 | 0.04116 | 0.03414 | 0.05041 |
| 15 000 | 0.02748 | 0.05338 | 0.05762 | 0.08175 |
| 20 000 | 0.03913 | 0.07812 | 0.06995 | 0.12899 |
| 25 000 | 0.05786 | 0.11293 | 0.09692 | 0.16618 |
| 50 000 | 0.12302 | 0.27205 | 0.22272 | 0.39648 |
| 100 000 | 0.27024 | 0.60993 | 0.46891 | 0.87883 |
| 200 000 | 0.54310 | 1.34030 | 0.96490 | 1.99622 |
| 350 000 | 1.00717 | 2.47517 | 2.00859 | 4.88111 |

По приведенным выше примерам можно сделать вывод о том, что в сортировке Шелла (последовательность Шелла) при полностью отсортированном массиве время, затраченное на обработку, меньше, чем при других рассмотренных случаях отсортированности массива; причем, самое большое время для сортировки оказалось у массива из случайно расположенных элементов. При увеличении количества элементов массива, возрастает время выполнения сортировки. Данный вывод говорит о том, что теоретическое описание подтверждается практической частью.

**Сортировка Шелла (последовательность Хиббарда).**

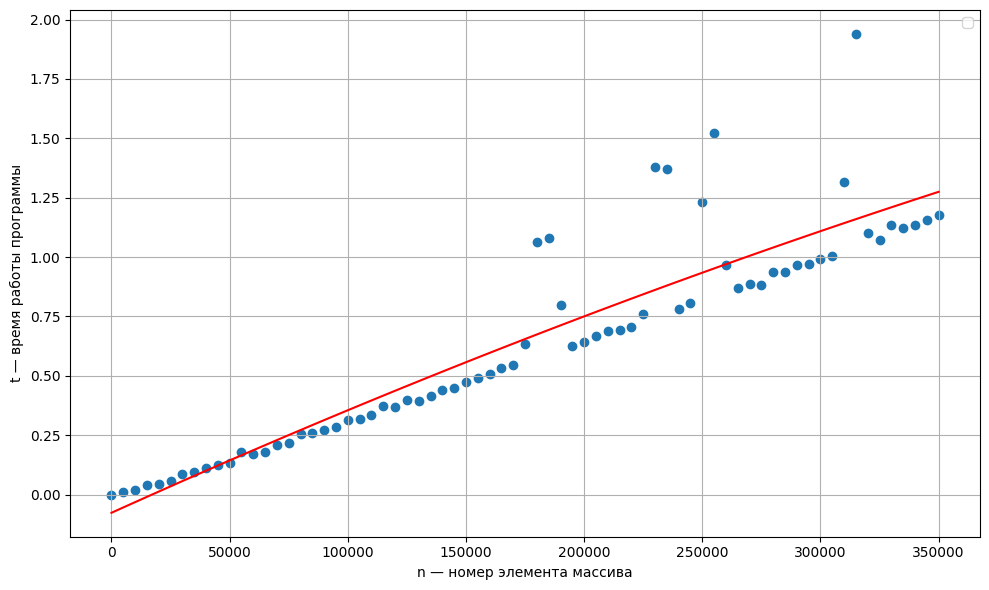


Рис. 31 — График экспериментальных данных для отсортированного массива и регрессионная кривая (-1.824e-12x2 + 4.499e-06x - 0.07662).

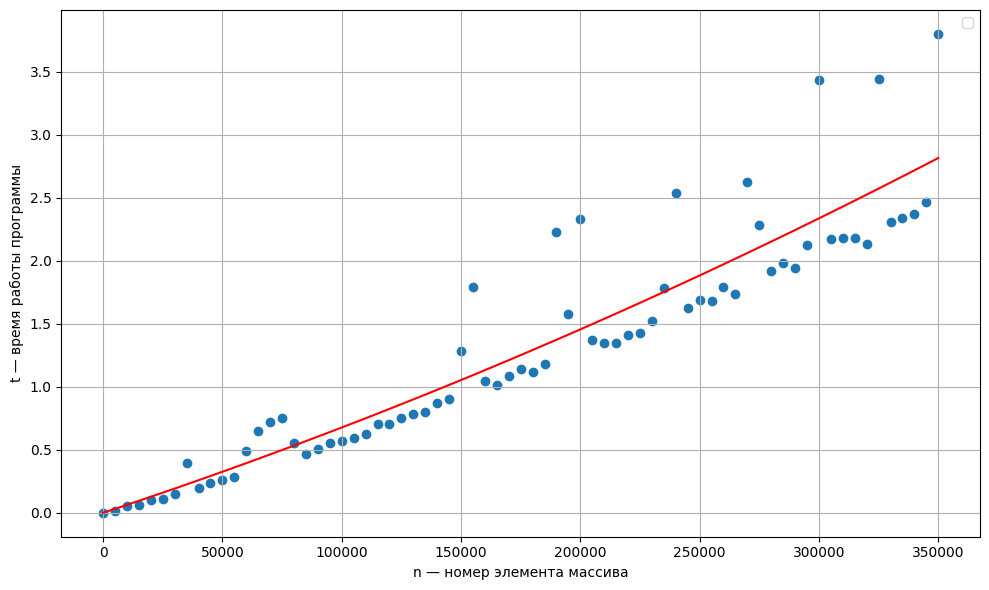


Рис. 32 — График экспериментальных данных для почти отсортированного массива и регрессионная кривая (5.103e-12x2 + 6.258e-06x - 0.0007571).

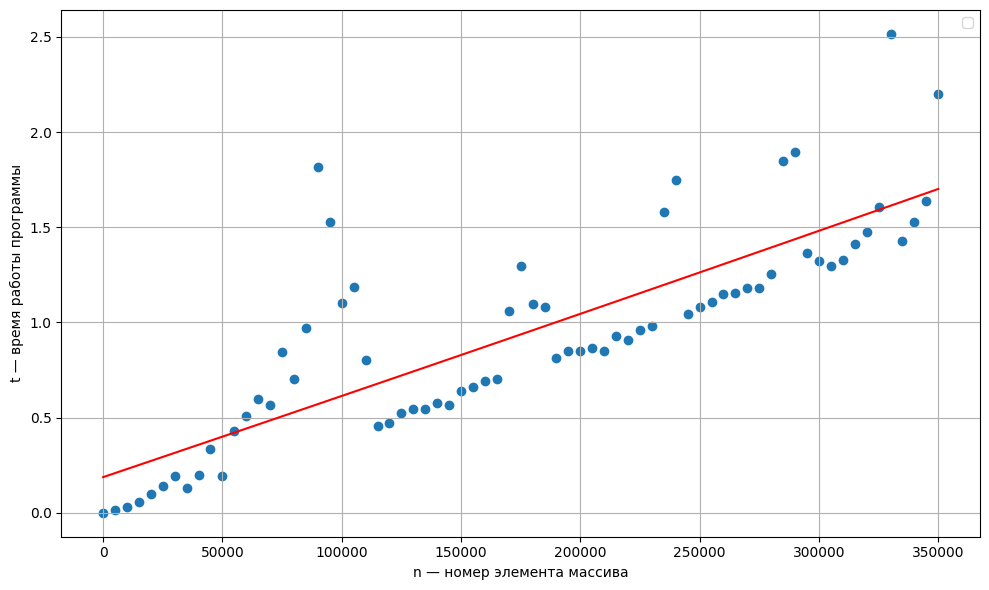


Рис. 33 — График экспериментальных данных для отсортированного в сторону убывания массива и регрессионная кривая (2.409e-13x2 + 4.241e-06x + 0.1873).

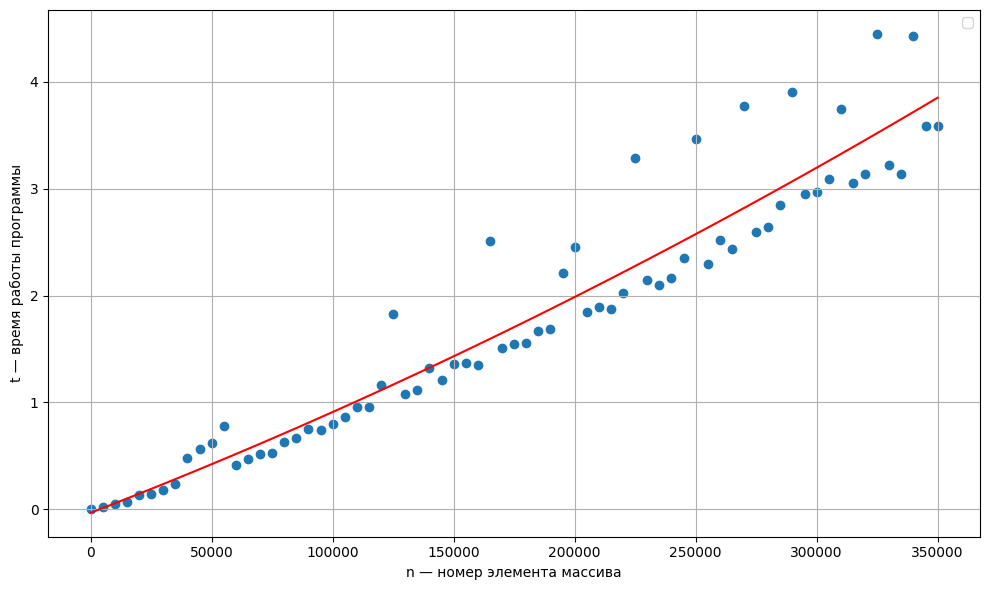


Рис. 34 — График экспериментальных данных для случайного массива и регрессионная кривая (6.677e-12x2 + 8.753e-06x - 0.03141).

**Таблица результатов измерений для сортировки Шелла (последовательность Хиббарда).**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов в массиве | Отсортированный массив | Почти отсортированный массив | Отсортированный по убыванию массив | Случайный массив |
| 5 000 | 0.01012 | 0.01603 | 0.01507 | 0.02537 |
| 10 000 | 0.02083 | 0.04943 | 0.03241 | 0.04627 |
| 15 000 | 0.04066 | 0.06363 | 0.05895 | 0.07402 |
| 20 000 | 0.04552 | 0.09984 | 0.09637 | 0.13121 |
| 25 000 | 0.05925 | 0.10738 | 0.13860 | 0.13992 |
| 50 000 | 0.13428 | 0.26239 | 0.19062 | 0.62031 |
| 100 000 | 0.31375 | 0.56800 | 1.10462 | 0.79983 |
| 200 000 | 0.64400 | 2.33422 | 0.85215 | 2.45157 |
| 350 000 | 1.17775 | 3.79963 | 2.19755 | 3.58289 |

По приведенным выше примерам можно сделать вывод о том, что в сортировке Шелла (последовательность Хиббарда) при случайном расположении элементов в массиве время, затраченное на обработку, больше, чем при других рассмотренных случаях отсортированности массива; причем, минимальное время для сортировки оказалось у полностью отсортированного массива. При увеличении количества элементов массива, возрастает время выполнения сортировки. Данный вывод говорит о том, что теоретическое описание подтверждается практической частью.

**Сортировка Шелла (последовательность Пратта).**

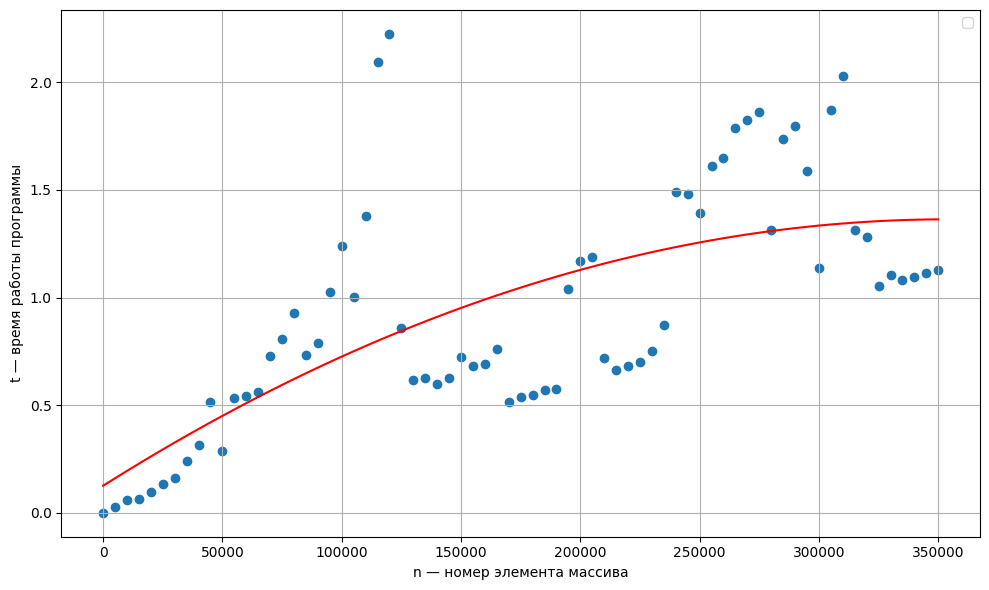


Рис. 35 — График экспериментальных данных для отсортированного массива и регрессионная кривая (-9.854e-12x2 + 6.986e-06x + 0.1257).

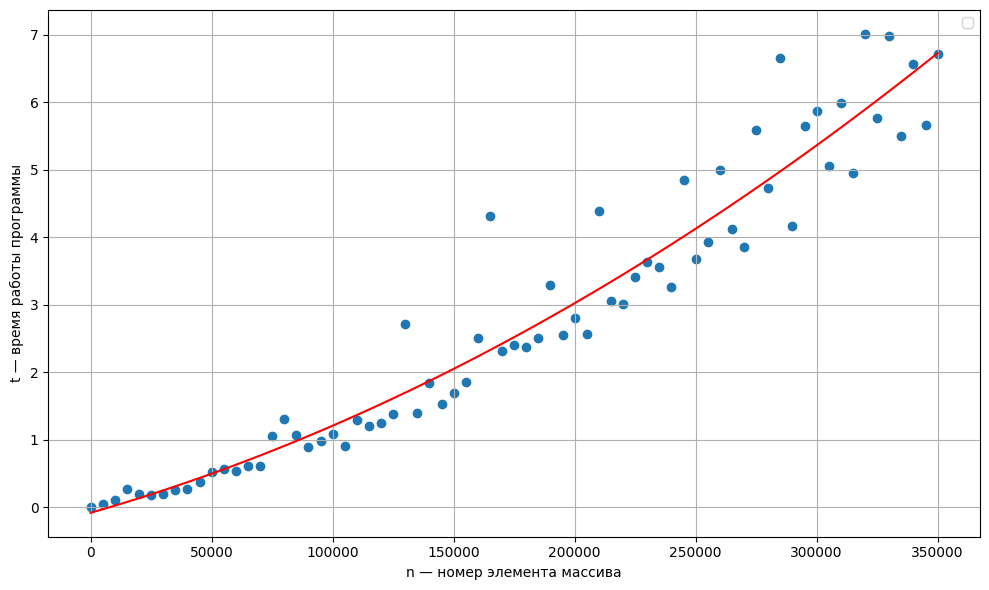


Рис. 36 — График экспериментальных данных для почти отсортированного массива и регрессионная кривая (2.624e-11x2 + 1.029e-05x - 0.08466).

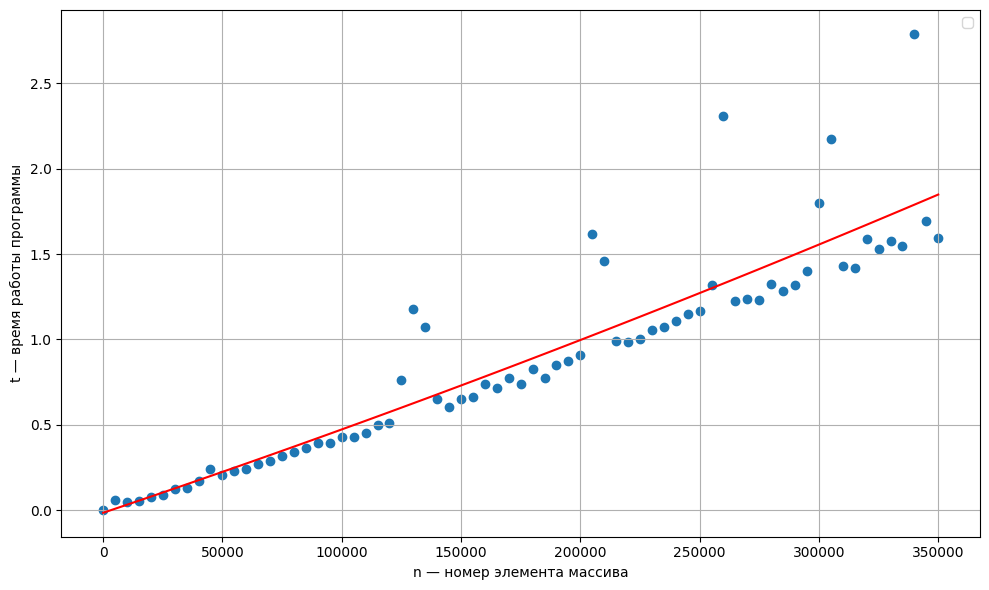


Рис. 37 — График экспериментальных данных для отсортированного в сторону убывания массива и регрессионная кривая (1.776e-12x2 + 4.703e-06x - 0.01514).

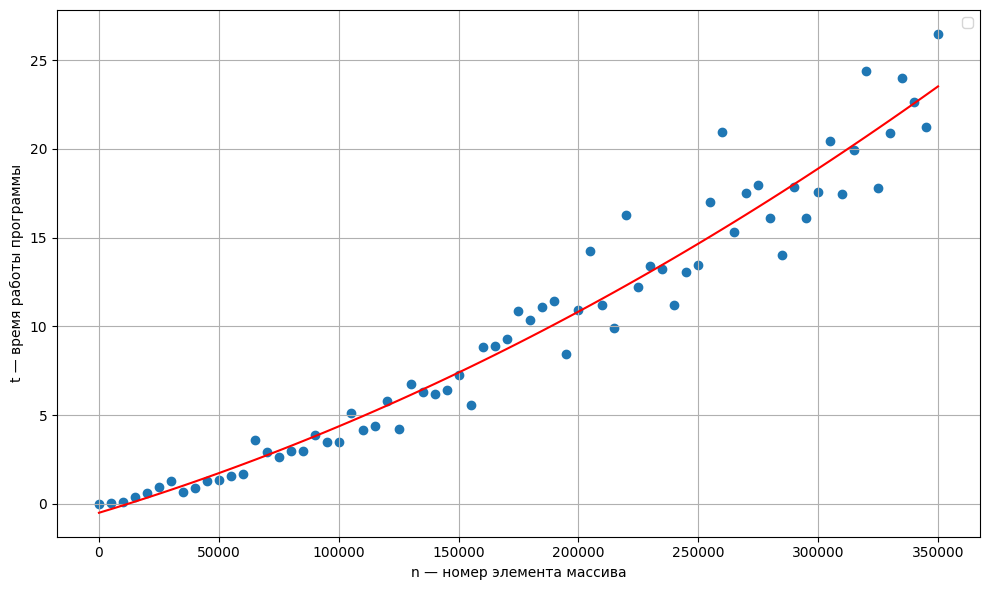


Рис. 38 — График экспериментальных данных для случайного массива и регрессионная кривая (7.989e-11x2 + 4.07e-05x - 0.5103).

**Таблица результатов измерений для сортировки Шелла (последовательность Пратта).**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов в массиве | Отсортированный массив | Почти отсортированный массив | Отсортированный по убыванию массив | Случайный массив |
| 5 000 | 0.02908 | 0.04799 | 0.06175 | 0.04674 |
| 10 000 | 0.05813 | 0.10821 | 0.04670 | 0.11574 |
| 15 000 | 0.06508 | 0.26408 | 0.05169 | 0.36632 |
| 20 000 | 0.09669 | 0.19799 | 0.07939 | 0.63092 |
| 25 000 | 0.13235 | 0.17796 | 0.08969 | 0.95515 |
| 50 000 | 0.28854 | 0.51603 | 0.20330 | 1.31494 |
| 100 000 | 1.23774 | 1.08282 | 0.42973 | 3.46216 |
| 200 000 | 1.16924 | 2.80856 | 0.90873 | 10.91501 |
| 350 000 | 1.12969 | 6.71164 | 1.59175 | 26.47688 |

По приведенным выше примерам можно сделать вывод о том, что в сортировке Шелла (последовательность Пратта) при отсортированном массиве время, затраченное на обработку, меньше, чем при других рассмотренных случаях отсортированности массива; причем, самое большое время для сортировки оказалось у массива из случайно расположенных элементов. При увеличении количества элементов массива, возрастает время выполнения сортировки. Данный вывод говорит о том, что теоретическое описание подтверждается практической частью.

**Общие графики для рассматриваемых последовательностей сортировки Шелла.**

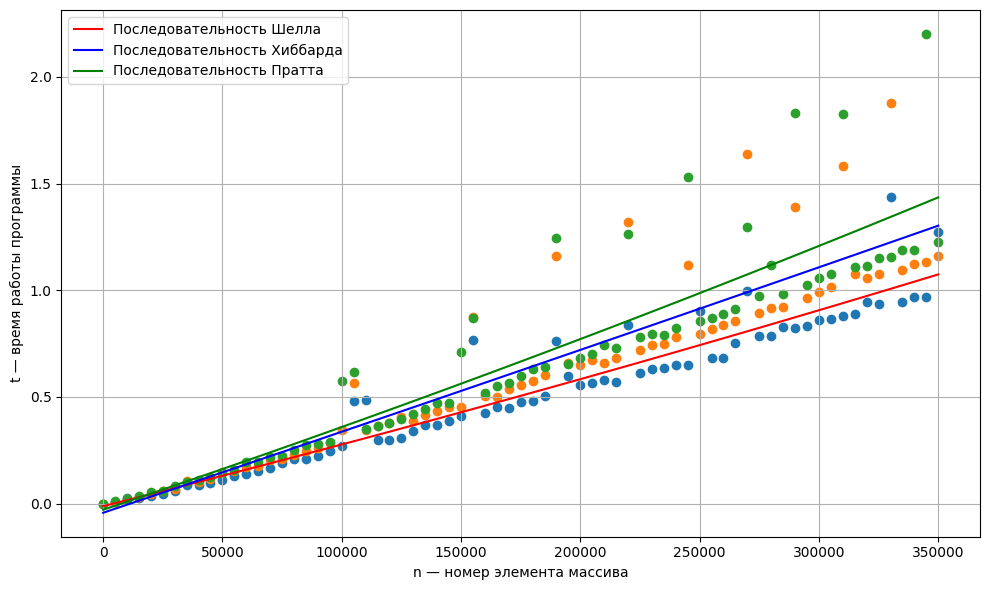


Рис. 39 — Графики экспериментальных данных для отсортированного массива и регрессионные кривые.

На рисунке 39 представлены графики функций для рассматриваемых последовательностей для полностью отсортированного массива. Уравнения регрессионных кривых: для последовательности Шелла — 8.442e-13x2 + 2.808e-06x - 0.01197, для последовательности Хиббарда — 1.851e-13x2 + 3.779e-06x - 0.04286, для последовательности Пратта — 1.22e-12x2 + 3.755e-06x - 0.02869. Из рисунка видно, что в данном примере сортировка Шелла работает быстрее при использовании последовательности Шелла.

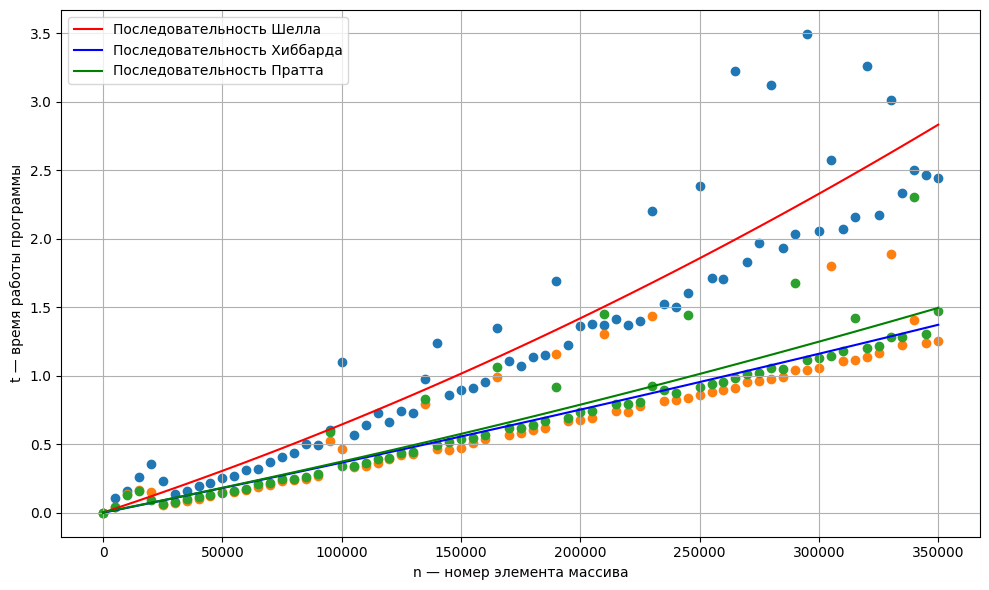


Рис. 40 — Графики экспериментальных данных для почти отсортированного массива и регрессионные кривые.

На рисунке 40 представлены графики функций для рассматриваемых последовательностей для почти отсортированного массива. Уравнения регрессионных кривых: для последовательности Шелла — 6.659e-12x2 + 5.756e-06x + 0.001923, для последовательности Хиббарда — 1.044e-12x2 + 3.554e-06x + 0.0003034, для последовательности Пратта — 2.227e-12x2 + 3.484e-06 x + 0.002932. Из рисунка видно, что в данном примере сортировка Шелла работает быстрее при использовании последовательности Хиббарда.

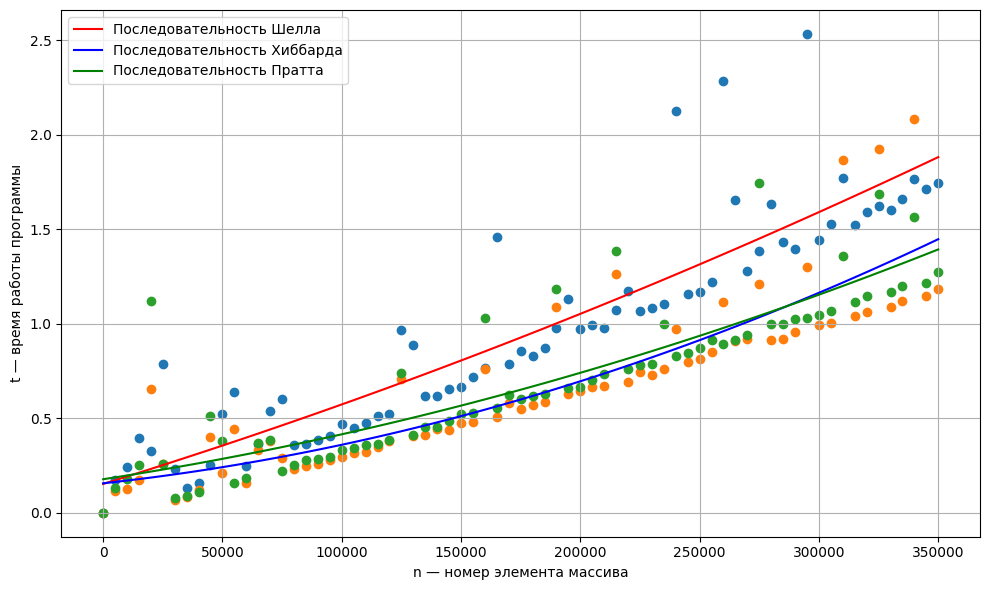


Рис. 41 — Графики экспериментальных данных для отсортированного в сторону убывания массива и регрессионные кривые.

На рисунке 41 представлены графики функций для рассматриваемых последовательностей для отсортированного в сторону убывания массива. Уравнения регрессионных кривых: для последовательности Шелла — 2.935e-12x2 + 3.911e-06x + 0.1525, для последовательности Хиббарда — 6.605e-12x2 + 1.376e-06x + 0.156, для последовательности Пратта — 4.387e-12x2 + 1.937e-06x + 0.1774. Из рисунка видно, что в данном примере сортировка Шелла работает быстрее при использовании последовательности Пратта.

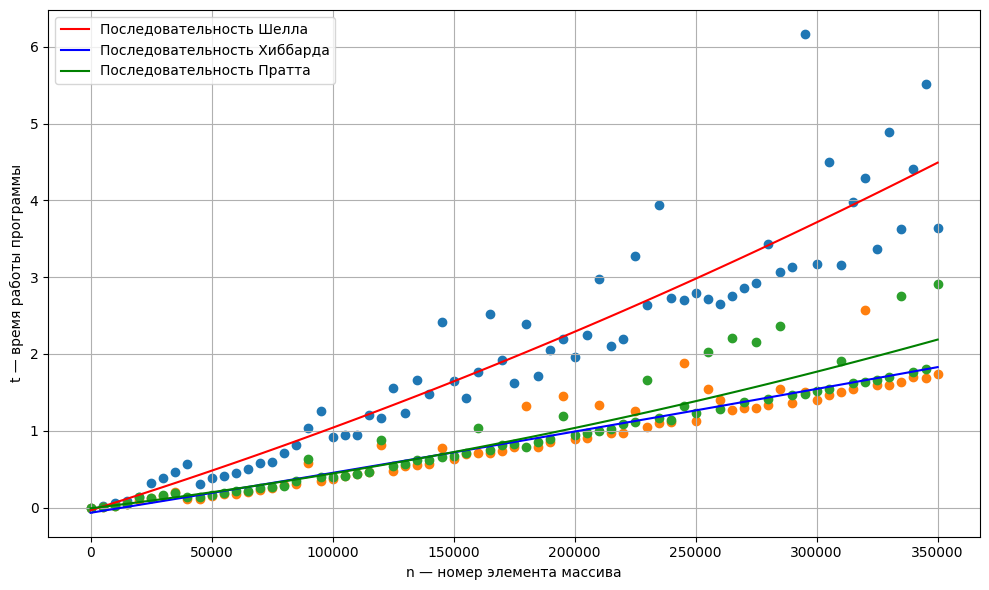


Рис. 42 — Графики экспериментальных данных для случайного массива и регрессионные кривые.

На рисунке 42 представлены графики функций для рассматриваемых последовательностей для массива из случайно расположенных элементов. Уравнения регрессионных кривых: для последовательности Шелла — 8.726e-12x2 + 9.883e-06x - 0.03685, для последовательности Хиббарда — 8.726e-13x2 + 5.115e-06x - 0.06876, для последовательности Пратта — 6.982e-12x2 + 3.834e-06x - 0.01043. Из рисунка видно, что в данном примере сортировка Шелла работает быстрее при использовании последовательности Хиббарда.

**Быстрая сортировка.**

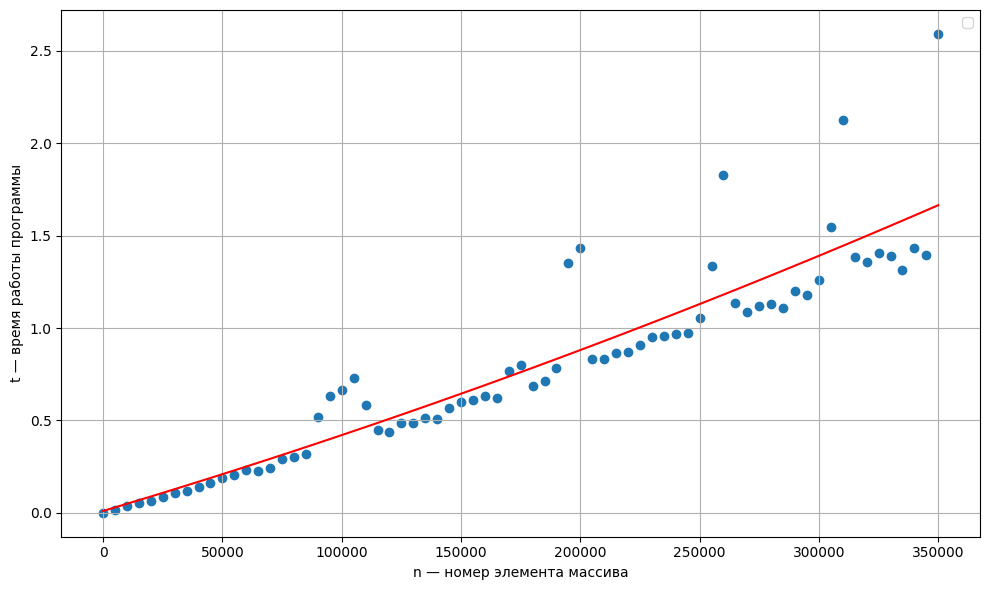


Рис. 43 — График экспериментальных данных для отсортированного массива и регрессионная кривая (2.49e-12x2 + 3.857e-06x + 0.009942).

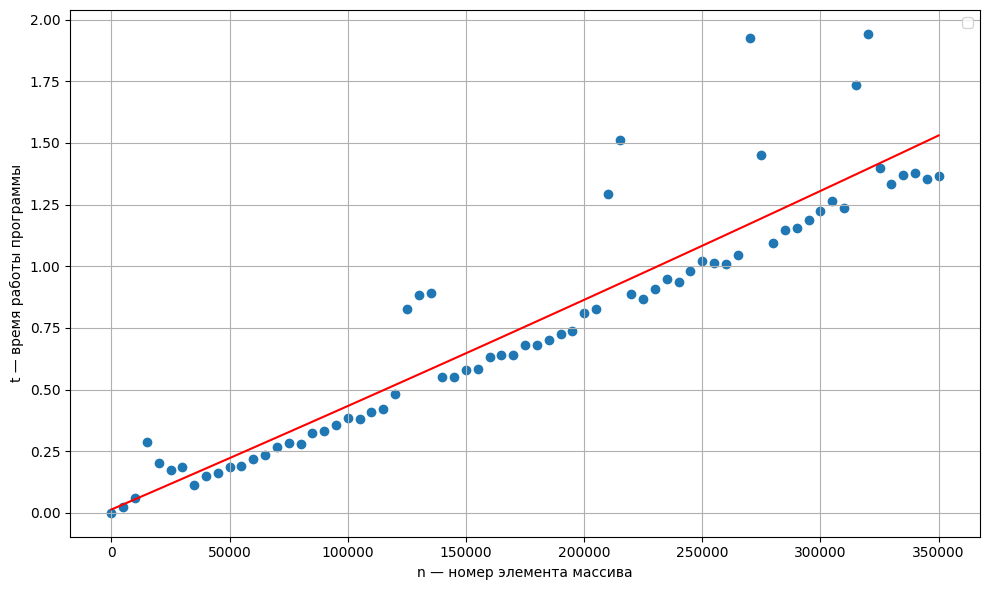


Рис. 44 — График экспериментальных данных для почти отсортированного массива и регрессионная кривая (5.51e-13x2 + 4.141e-06x + 0.01343).

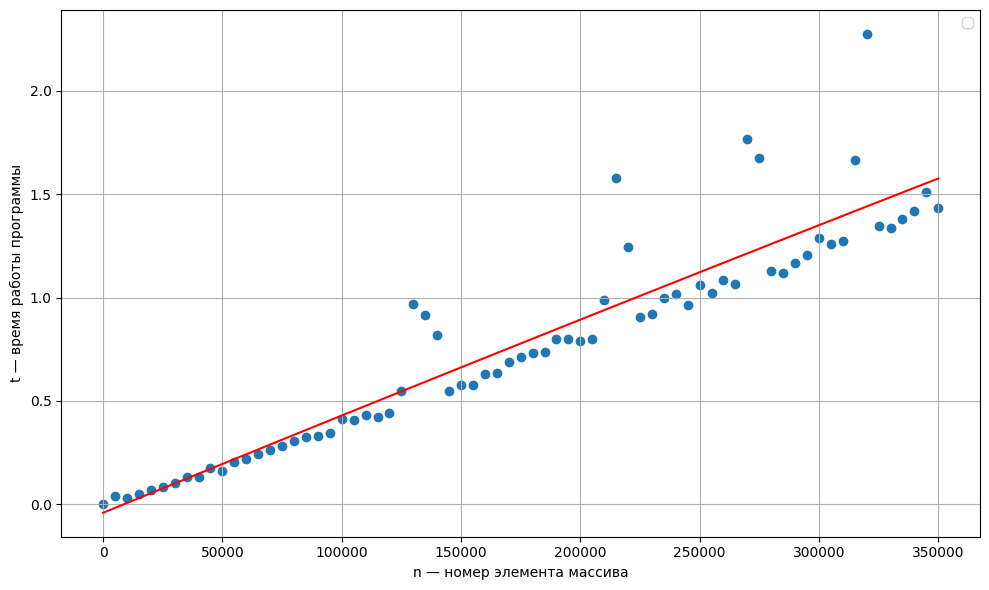


Рис. 45 — График экспериментальных данных для отсортированного в сторону убывания массива и регрессионная кривая (-3.383e-13x2 + 4.739e-06x - 0.04077).

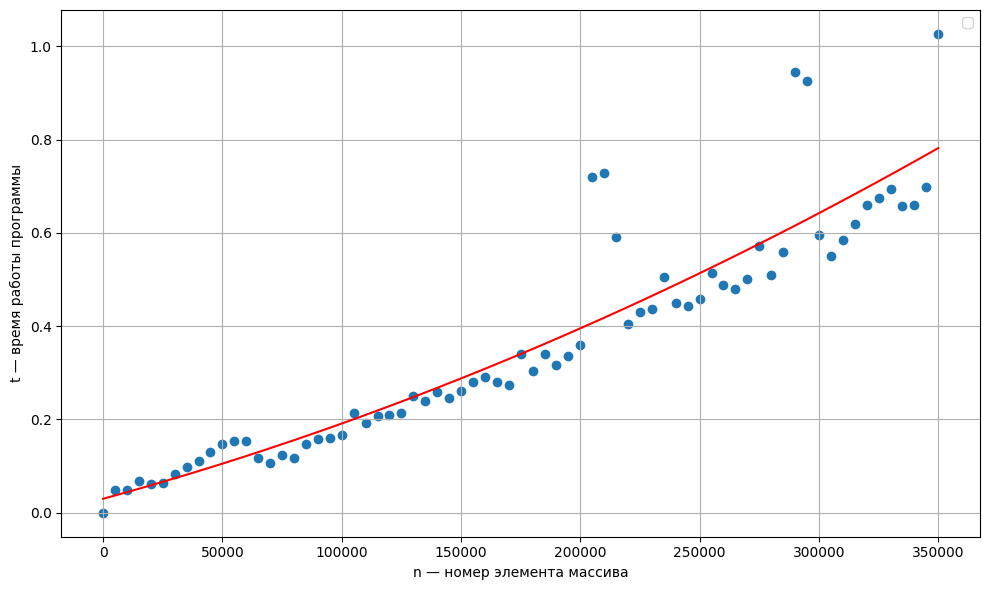


Рис. 46 — График экспериментальных данных для случайного массива и регрессионная кривая (2.14e-12x2 + 1.399e-06x + 0.03008).

**Таблица результатов измерений для быстрой сортировки.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов в массиве | Отсортированный массив | Почти отсортированный массив | Отсортированный по убыванию массив | Случайный массив |
| 5 000 | 0.01439 | 0.02386 | 0.03819 | 0.04784 |
| 10 000 | 0.03784 | 0.05929 | 0.03236 | 0.04777 |
| 15 000 | 0.05055 | 0.28775 | 0.05120 | 0.06862 |
| 20 000 | 0.06220 | 0.20201 | 0.06813 | 0.06087 |
| 25 000 | 0.08701 | 0.17317 | 0.08302 | 0.06341 |
| 50 000 | 0.18892 | 0.18454 | 0.16375 | 0.14728 |
| 100 000 | 0.66352 | 0.38643 | 0.41343 | 0.16656 |
| 200 000 | 1.43054 | 0.81140 | 0.79107 | 0.35946 |
| 350 000 | 2.59191 | 1.36762 | 1.43503 | 1.02639 |

По приведенным выше примерам можно сделать вывод о том, что в быстрой сортировке при отсортированном массиве время, затраченное на обработку, больше, чем при других рассмотренных случаях отсортированности массива; причем, наименьшее время для сортировки оказалось у массива из случайно расположенных элементов. При увеличении количества элементов массива, возрастает время выполнения сортировки. Данный вывод говорит о том, что теоретическое описание подтверждается практической частью.

**Пирамидальная сортировка.**

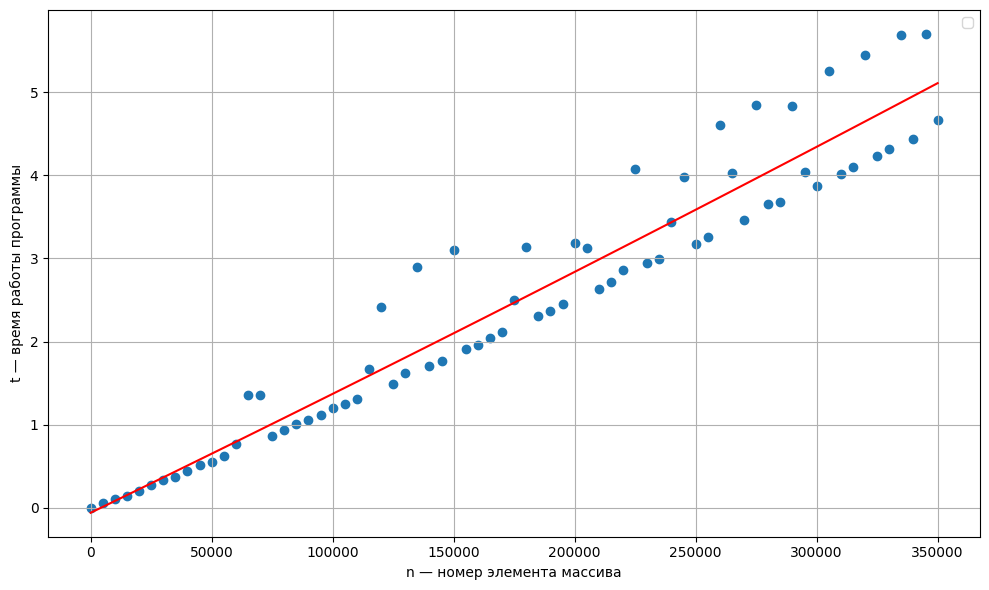


Рис. 47 — График экспериментальных данных для отсортированного массива и регрессионная кривая (1.851e-12x2 + 1.412e-05x - 0.06001).

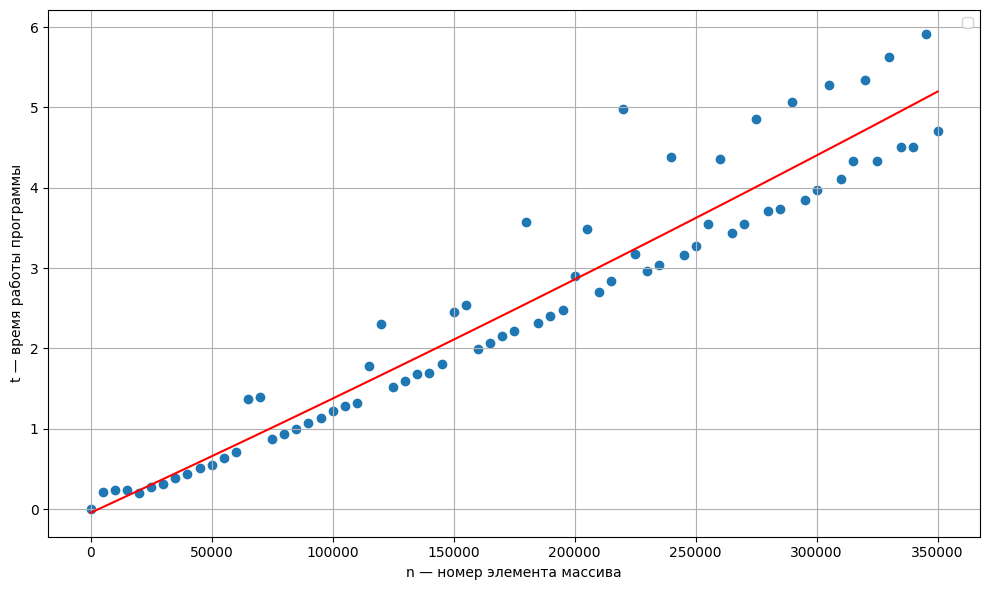


Рис. 48 — График экспериментальных данных для почти отсортированного массива и регрессионная кривая (3.064e-12x2 + 1.391e-05x - 0.04654).

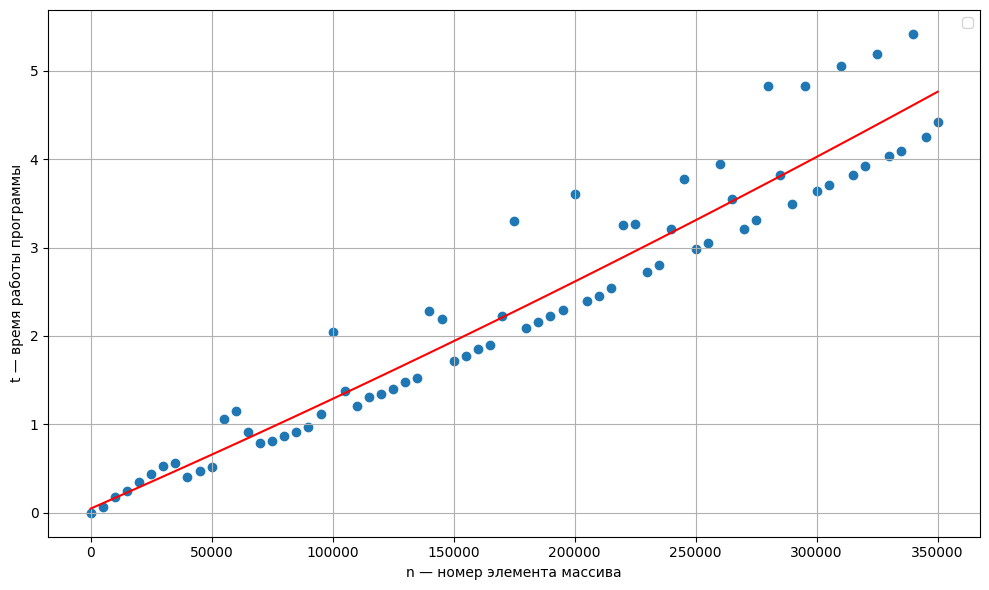


Рис. 49 — График экспериментальных данных для отсортированного в сторону убывания массива и регрессионная кривая (4.231e-12x2 + 1.199e-05x + 0.04703).

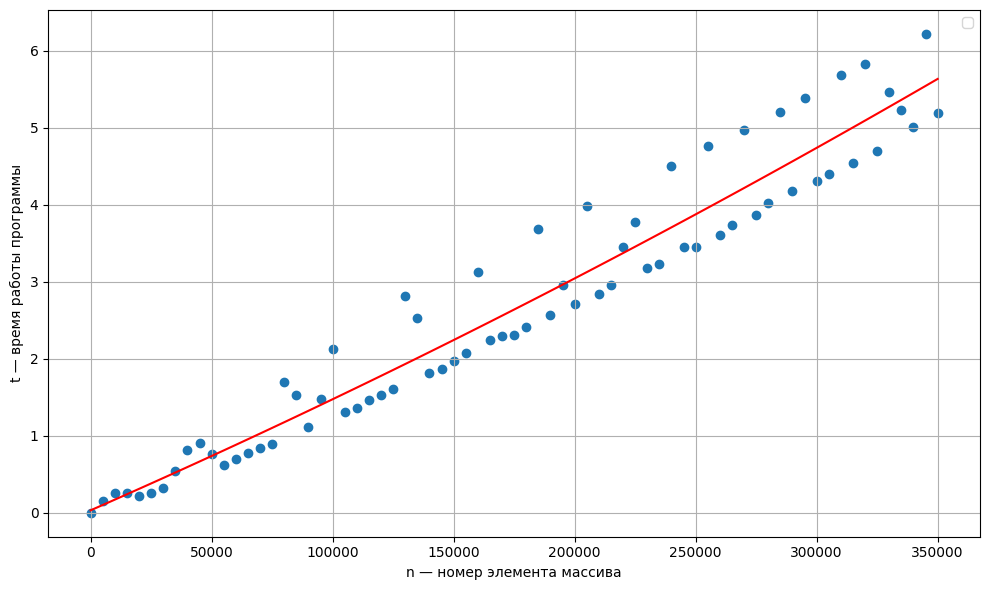


Рис. 50 — График экспериментальных данных для случайного массива и регрессионная кривая (6.328e-12x2 + 1.379e-05x + 0.032).

**Таблица результатов измерений для пирамидальной сортировки.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов в массиве | Отсортированный массив | Почти отсортированный массив | Отсортированный по убыванию массив | Случайный массив |
| 5 000 | 0.05612 | 0.21545 | 0.07013 | 0.15887 |
| 10 000 | 0.09996 | 0.23299 | 0.17329 | 0.26145 |
| 15 000 | 0.14414 | 0.24101 | 0.24282 | 0.25363 |
| 20 000 | 0.20574 | 0.20364 | 0.34455 | 0.22213 |
| 25 000 | 0.26814 | 0.27726 | 0.43801 | 0.25412 |
| 50 000 | 0.55447 | 0.54833 | 0.51902 | 0.76426 |
| 100 000 | 1.19614 | 1.21438 | 2.04393 | 2.13097 |
| 200 000 | 3.18144 | 2.89948 | 3.60746 | 2.70622 |
| 350 000 | 4.66594 | 4.70105 | 4.42313 | 5.18700 |

По приведенным выше примерам можно сделать вывод о том, что в пирамидальной сортировке вне зависимости от отсортированности входного массива время работы остается приблизительно одинаковым. При увеличении количества элементов массива, возрастает время выполнения сортировки. Данный вывод говорит о том, что теоретическое описание подтверждается практической частью.

**Обработка результатов эксперимента.**

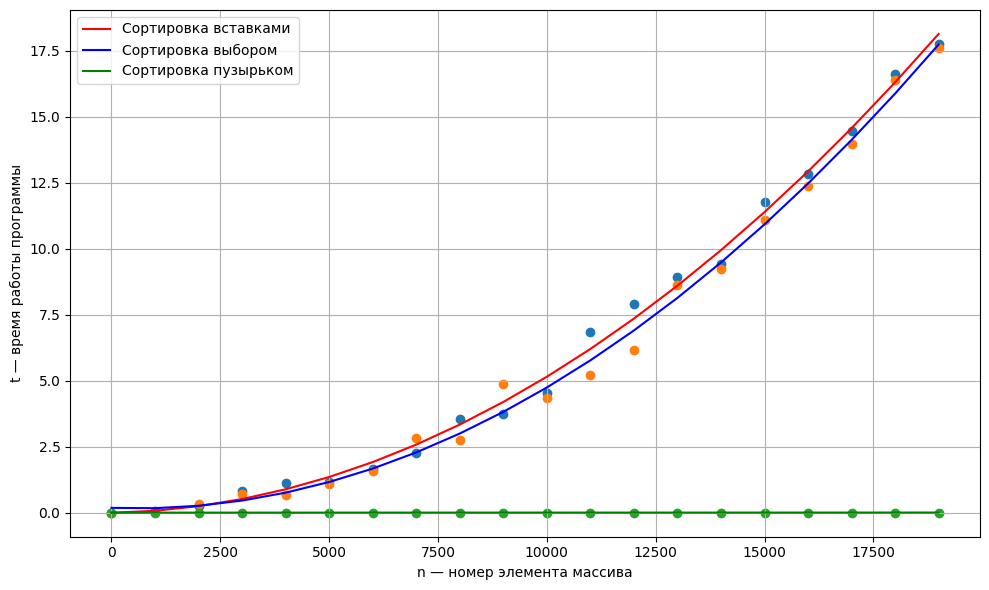


Рис. 51 — График экспериментальных данных для квадратичных сортировок.

Регрессионные кривые для квадратичных сортировок на рисунке 51: сортировка вставками — 4.885e-08x2 + 2.705e-05x - 0.005362, сортировка выбором — 5.202e-08x2 - 6.4e-05x + 0.1806, сортировка пузырьком — 6.736e-12x2 + 3.183e-08x + 0.0004261.

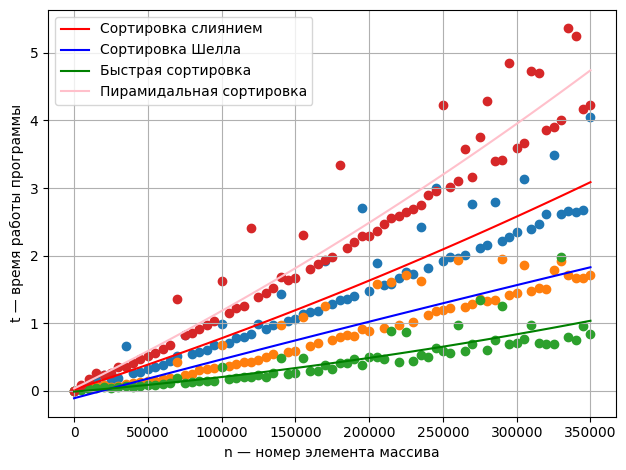


Рис. 52 — График экспериментальных данных для не квадратичных сортировок.

Из рисунка 52 можно сказать, что самой быстрой сортировкой является быстрая сортировка. Этот же вывод можно получить, сравнив данные таблиц, полученные при разборе каждой сортировки выше. Регрессионные кривые для каждого графика: сортировка слиянием — 4.723e-12x2 + 7.116e-06x + 0.01751, сортировка Шелла (последовательность Хиббарда) — -8.347e-13x2 + 5.833e-06x - 0.1123, быстрая сортировка — 3.341e-12x2 + 1.829e-06x - 0.0146, пирамидальная сортировка — 7.912e-12x2 + 1.069e-05x + 0.0319.

**Код программы для сортировки массивов.**

from random import randint

import random

# функция для сортировки вставками

def Insertion\_Sort(A):

for j in range(1, len(A)):

key = A[j]

i = j - 1

while i >= 0 and A[i] > key:

A[i + 1] = A[i]

i = i - 1

A[i + 1] = key

return A

#функция для быстрой сортировки

def Quick\_Sort(A):

if len(A) <= 1:

return A

else:

q = random.choice(A)

l\_nums = [n for n in A if n < q]

el\_nums = [q] \* A.count(q)

r\_nums = [n for n in A if n > q]

return Quick\_Sort(l\_nums) + el\_nums + Quick\_Sort(r\_nums)

# функция для сортировки выбором

def Selection\_Sort(A):

index = 0

index\_min = 0

for i in range(index\_min, len(A) - 1):

for j in range(i + 1, len(A)):

if A[j] < A[index\_min]:

index\_min = j

A[index], A[index\_min] = A[index\_min], A[index]

index += 1

index\_min = index

return A

#функция сортировки пузырьком

def Bubble\_Sort(A):

for i in range(len(arr)):

flag = False

for j in range(0, len(A) - i - 1):

if A[j] > A[j + 1]:

A[j], A[j + 1] = A[j + 1], A[j]

flag = True

if not flag:

return A

#функция сортировки слиянием

def Merge\_Sort(A, p, r):

if p < r:

q = int((p + r)/2)

Merge\_Sort(A, p, q)

Merge\_Sort(A, q + 1, r)

Merge(A, p, q, r)

return A

def Merge(A, p, q, r):

n1 = q - p + 1

n2 = r - q

L = []

R = []

for i in range(n1):

L.append(A[p + i])

for j in range(n2):

R.append(A[q + j + 1])

L.append(float('inf'))

R.append(float('inf'))

i = j = 0

for k in range(p, r + 1):

if(L[i] <= R[j]):

A[k] = L[i]

i += 1

else:

A[k] = R[j]

j += 1

return A

#функция сортировки Шелла — последовательность Шелла

def Shell\_Sort\_Shell(A):

step = len(A)//2

while step > 0:

for i in range(step, len(A)):

value = A[i]

while i >= step and A[i - step] > value:

A[i] = A[i - step]

i -= step

A[i] = value

step //= 2

return A

#функция сортировки Шелла — последовательность Хиббарда

def Shell\_Sort\_Hib(A):

n = len(A)

k = 0

step = 1

while step < n:

step = 2 \*\* k - 1

k += 1

while step > 0:

for i in range(step, n):

value = A[i]

j = i

while j >= step and A[j - step] > value:

A[j] = A[j - step]

j -= step

A[j] = value

k -= 1

step = 2 \*\* k - 1 if k > 0 else 0

return A

#функция сортировки Шелла — последовательность Пратта

def generate\_pratt(n):

sequence = []

i, j = 0, 0

while True:

value = (2 \*\* i) \* (3 \*\* j)

if value > n: break

sequence.append(value)

if i < j: j += 1

else: i += 1

return sequence

def Shell\_Sort\_Pratt(A):

n = len(A)

pratt\_sequence = generate\_pratt(n)

pratt\_sequence.reverse()

for step in pratt\_sequence:

for i in range(step, n):

value = A[i]

j = i

while j >= step and A[j - step] > value:

A[j] = A[j - step]

j -= step

A[j] = value

return A

#функции для пирамидальной сортировки

def Parent(i): return (i - 1) // 2

def Left(i): return 2 \* i + 1

def Right(i): return 2 \* i + 2

def Max\_Heapify(A, i, heap\_size):

l = Left(i)

r = Right(i)

largest = i

if l < heap\_size and A[l] > A[largest]:

largest = l

if r < heap\_size and A[r] > A[largest]:

largest = r

if largest != i:

A[i], A[largest] = A[largest], A[i]

Max\_Heapify(A, largest, heap\_size)

def Build\_Max\_Heap(A):

heap\_size = len(A)

for i in range(heap\_size // 2 - 1, -1, -1):

Max\_Heapify(A, i, heap\_size)

def Heap\_Sort(A):

Build\_Max\_Heap(A)

heap\_size = len(A)

for i in range(heap\_size - 1, 0, -1):

A[0], A[i] = A[i], A[0]

Max\_Heapify(A, 0, i)

return A

N = 100

#arr = [randint(1, 20000) for i in range(N)]

arr = [42, 7, 19, 86, 33, 58, 14, 91, 27, 65, 3, 74, 12, 50, 88, 88, 88]

#arr = [5, 2, 4, 7, 1, 3, 2, 6]

arr\_sort = arr.copy()

arr\_sort.sort()

print(arr)

arr = Quick\_Sort(arr)

print(arr\_sort)

#print(Insertion\_Sort(arr))

print(arr)

#print(Selection\_Sort(arr))

#print(shuffle(arr))

#print(Bubble\_Sort(arr))

#print(Merge\_Sort(arr, 0, len(arr) - 1))

#print(Shell\_Sort(arr))

#print(Heap\_Sort(arr))

print(arr==arr\_sort)

**Код программы для построения графиков.**

from time import \*

import numpy as np

import matplotlib.pyplot as plt

import random

from random import randint

from math import \*

def Insertion\_Sort(A):

    for j in range(1, len(A)):

        key = A[j]

        i = j - 1

        while i >= 0 and A[i] > key:

            A[i + 1] = A[i]

            i = i - 1

        A[i + 1] = key

    return A

def Selection\_Sort(A):

    index = 0

    index\_min = 0

    for i in range(index\_min, len(A) - 1):

        for j in range(i + 1, len(A)):

            if A[j] < A[index\_min]:

                index\_min = j

        A[index], A[index\_min] = A[index\_min], A[index]

        index += 1

        index\_min = index

    return A

def Bubble\_Sort(A):

    for i in range(len(arr)):

        flag = False

        for j in range(0, len(A) - i - 1):

            if A[j] > A[j + 1]:

                A[j], A[j + 1] = A[j + 1], A[j]

                flag = True

        if not flag:

            return A

def Merge\_Sort(A, p, r):

    if p < r:

        q = int((p + r)/2)

        Merge\_Sort(A, p, q)

        Merge\_Sort(A, q + 1, r)

        Merge(A, p, q, r)

    return A

def Merge(A, p, q, r):

    n1 = q - p + 1

    n2 = r - q

    L = []

    R = []

    for i in range(n1):

        L.append(A[p + i])

    for j in range(n2):

        R.append(A[q + j + 1])

    L.append(float('inf'))

    R.append(float('inf'))

    i = j = 0

    for k in range(p, r + 1):

        if(L[i] <= R[j]):

            A[k] = L[i]

            i += 1

        else:

            A[k] = R[j]

            j += 1

    return A

def Shell\_Sort(A):

    n = len(A)

    k = 0

    step = 1

    while step < n:

        step = 2 \*\* k - 1

        k += 1

    while step > 0:

        for i in range(step, n):

            value = A[i]

            j = i

            while j >= step and A[j - step] > value:

                A[j] = A[j - step]

                j -= step

            A[j] = value

        k -= 1

        step = 2 \*\* k - 1 if k > 0 else 0

    return A

def Quick\_Sort(A):

    if len(A) <= 1:

        return A

    else:

        q = random.choice(A)

    l\_nums = [n for n in A if n < q]

    el\_nums = [q] \* A.count(q)

    r\_nums = [n for n in A if n > q]

    return Quick\_Sort(l\_nums) + el\_nums + Quick\_Sort(r\_nums)

def Parent(i): return (i - 1) // 2

def Left(i): return 2 \* i + 1

def Right(i): return 2 \* i + 2

def Max\_Heapify(A, i, heap\_size):

    l = Left(i)

    r = Right(i)

    largest = i

    if l < heap\_size and A[l] > A[largest]:

        largest = l

    if r < heap\_size and A[r] > A[largest]:

        largest = r

    if largest != i:

        A[i], A[largest] = A[largest], A[i]

        Max\_Heapify(A, largest, heap\_size)

def Build\_Max\_Heap(A):

    heap\_size = len(A)

    for i in range(heap\_size // 2 - 1, -1, -1):

        Max\_Heapify(A, i, heap\_size)

def Heap\_Sort(A):

    Build\_Max\_Heap(A)

    heap\_size = len(A)

    for i in range(heap\_size - 1, 0, -1):

        A[0], A[i] = A[i], A[0]

        Max\_Heapify(A, 0, i)

    return A

time\_arr\_Insertion = []

time\_arr\_Selection = []

time\_arr\_Bubble = []

time\_arr\_Merge = []

time\_arr\_Shell = []

time\_arr\_Quick = []

time\_arr\_Heap = []

x\_arr\_Insertion = []

x\_arr\_Selection = []

x\_arr\_Bubble = []

x\_arr\_Merge = []

x\_arr\_Shell = []

x\_arr\_Quick = []

x\_arr\_Heap = []

for i in range(0, 350001, 5000):

    arr = [randint(1, 1000) for \_ in range(i)]

    save\_arr\_Invertion = arr

    save\_arr\_Selection = arr

    save\_arr\_Bubble = arr

    save\_arr\_Merge = arr

    save\_arr\_Shell = arr

    save\_arr\_Quick = arr

    save\_arr\_Heap = arr

    '''start\_time = time()

    Insertion\_Sort(save\_arr\_Invertion)

    finish\_time = time()

    time\_arr\_Insertion.append(finish\_time-start\_time)

    x\_arr\_Insertion.append(i)

    start\_time = time()

    Selection\_Sort(save\_arr\_Selection)

    finish\_time = time()

    time\_arr\_Selection.append(finish\_time-start\_time)

    x\_arr\_Selection.append(i)

    start\_time = time()

    Bubble\_Sort(save\_arr\_Bubble)

    finish\_time = time()

    time\_arr\_Bubble.append(finish\_time-start\_time)

    x\_arr\_Bubble.append(i)'''

    start\_time = time()

    Merge\_Sort(save\_arr\_Merge, 0, len(save\_arr\_Merge)-1)

    finish\_time = time()

    time\_arr\_Merge.append(finish\_time-start\_time)

    x\_arr\_Merge.append(i)

    start\_time = time()

    Shell\_Sort(save\_arr\_Shell)

    finish\_time = time()

    time\_arr\_Shell.append(finish\_time-start\_time)

    x\_arr\_Shell.append(i)

    start\_time = time()

    Quick\_Sort(save\_arr\_Quick)

    finish\_time = time()

    time\_arr\_Quick.append(finish\_time-start\_time)

    x\_arr\_Quick.append(i)

    start\_time = time()

    Heap\_Sort(save\_arr\_Bubble)

    finish\_time = time()

    time\_arr\_Heap.append(finish\_time-start\_time)

    x\_arr\_Heap.append(i)

'''p\_Insertion = np.polyfit(x\_arr\_Insertion, time\_arr\_Insertion, 2)

polinom = np.poly1d(p\_Insertion)

print("Уравнение регрессионной кривой для вставками — ", polinom)

p\_Selection = np.polyfit(x\_arr\_Selection, time\_arr\_Selection, 2)

polinom = np.poly1d(p\_Selection)

print("Уравнение регрессионной кривой для выбором — ", polinom)

p\_Bubble = np.polyfit(x\_arr\_Bubble, time\_arr\_Bubble, 2)

polinom = np.poly1d(p\_Bubble)

print("Уравнение регрессионной кривой для сортировки пузырьком — ", polinom)'''

p\_Merge = np.polyfit(x\_arr\_Merge, time\_arr\_Merge, 2)

polinom = np.poly1d(p\_Merge)

print("Уравнение регрессионной кривой для сортировки слиянием — ", polinom)

p\_Shell = np.polyfit(x\_arr\_Shell, time\_arr\_Shell, 2)

polinom = np.poly1d(p\_Shell)

print("Уравнение регрессионной кривой для сортировки Шелла — ", polinom)

p\_Quick = np.polyfit(x\_arr\_Quick, time\_arr\_Quick, 2)

polinom = np.poly1d(p\_Quick)

print("Уравнение регрессионной кривой для быстрой сортировки — ", polinom)

p\_Heap = np.polyfit(x\_arr\_Heap, time\_arr\_Heap, 2)

polinom = np.poly1d(p\_Heap)

print("Уравнение регрессионной кривой для сортировки кучей — ", polinom)

'''plt.figure(figsize=(10, 6))

plt.scatter(x\_arr\_Insertion,time\_arr\_Insertion)

plt.scatter(x\_arr\_Selection,time\_arr\_Selection)

plt.scatter(x\_arr\_Bubble,time\_arr\_Bubble)'''

plt.scatter(x\_arr\_Merge,time\_arr\_Merge)

plt.scatter(x\_arr\_Shell,time\_arr\_Shell)

plt.scatter(x\_arr\_Quick,time\_arr\_Quick)

plt.scatter(x\_arr\_Heap,time\_arr\_Heap)

plt.xlabel('n — номер элемента массива')

plt.ylabel('t — время работы программы')

plt.legend()  # Добавляем легенду

plt.grid()    # Добавляем сетку

'''plt.plot(x\_arr\_Insertion, np.polyval(p\_Insertion, x\_arr\_Insertion), label='Сортировка вставками', color='red')

plt.plot(x\_arr\_Selection, np.polyval(p\_Selection, x\_arr\_Selection), label='Сортировка выбором', color='blue')

plt.plot(x\_arr\_Bubble, np.polyval(p\_Bubble, x\_arr\_Bubble), label='Сортировка пузырьком', color='green')'''

plt.plot(x\_arr\_Merge, np.polyval(p\_Merge, x\_arr\_Merge), label='Сортировка слиянием', color='red')

plt.plot(x\_arr\_Shell, np.polyval(p\_Shell, x\_arr\_Shell), label='Сортировка Шелла', color='blue')

plt.plot(x\_arr\_Quick, np.polyval(p\_Quick, x\_arr\_Quick), label='Быстрая сортировка', color='green')

plt.plot(x\_arr\_Heap, np.polyval(p\_Heap, x\_arr\_Heap), label='Пирамидальная сортировка', color='pink')

plt.tight\_layout()

plt.legend()

plt.show()

*Ссылка на GitHub*: https://github.com/KatyukaPinched/algorithms\_ds

**Выводы.**

В ходе работы были проанализированы квадратичные сортировки (выбором, вставками, пузырьком) и более быстрые (слиянием, Шелла, быстрая, пирамидальная). Были произведены теоретические и практические сравнения данных сортировок, в ходе которых стало очевидным, что самой быстрой сортировкой из рассмотренных является быстрая сортировка.