**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра систем автоматизированного проектирования**

отчет

**по лабораторной работе №1**

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

**Тема: «Алгоритмы сортировки сравнением»**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 3352 |  | Гультяев А.С. |
| Преподаватель |  | Пестерев Д.О. |

Санкт-Петербург

2024

**Оглавление**

[**Введение**](#_Введение)**3**

1. [Теоретическая часть](#_Теоретическая_часть)4
   1. [Сортировка вставками](#_Сортировка_вставками.)4
   2. [Сортировка выбором6](#_Сортировка_выбором)
   3. [Сортировка пузырьком](#_Сортировка_пузырьком)8
   4. [Сортировка слиянием](#_Сортировка_слиянием)11
   5. [Быстрая сортировка](#_Быстрая_сортировка)14
   6. [Пирамидальная сортировка](#_Пирамидальная_сортировка)17
   7. [Сортировка Шелла с последовательностью Шелла](#_Сортировка_Шелла_с)19
   8. [Сортировка Шелла с последовательностью Пратта](#_Сортировка_Шелла_с_1)20
   9. [Сортировка Шелла с последовательностью Хиббарда2](#_Сортировка_Шелла_с_2)1
   10. [Общий график23](#_Общий_график)
2. [**Практическая часть24**](#_Практическая_часть.)
   1. [Сортировка вставками2](#_Сортировка_вставками)4
   2. [Сортировка выбором](#_Сортировка_выбором_1)25
   3. [Сортировка пузырьком2](#_Сортировка_пузырьком_1)7
   4. [Сортировка слиянием2](#_Сортировка_слиянием_1)9
   5. [Быстрая сортировка](#_Быстрая_сортировка_1)31
   6. [Пирамидальная сортировка](#_Пирамидальная_сортировка_1)33
   7. [Сортировка Шелла с последовательностью Шелла](#_Сортировка_Шелла_с_3)34
   8. [Сортировка Шелла с последовательностью Пратта](#_Сортировка_Шелла_с_4)36
   9. [Сортировка Шелла с последовательностью Хиббарда](#_Сортировка_Шелла_с_5)38

[**Код** **программы**](#_Код_программы)**40**

[**Заключение**](#_Заключение)**52**

**[Ссылка на репозиторий GitHub52](#_Ссылка_на_репозиторий)**

# Введение

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

**Задачи:** для каждого алгоритма сортировки:

1. Рассчитать функцию временной сложности для лучшего, худшего и среднего случаев;
2. Проанализировать устойчивость сортировок;
3. Получить асимптотическую оценку временной сложности для лучшего, худшего, среднего случаев;
4. Получить функцию пространственной сложности и ее асимптотическую оценку;
5. Построить графики функции временной сложности для лучшего, худшего, среднего случаев;
6. Получить экспериментальную оценку временной сложности для отсортированного, почти отсортированного, обратно отсортированного и случайного массивов;
7. Оценить асимптотику полученных данных с помощью регрессионной кривой.

# Теоретическая часть

# Сортировка вставками.

1. Описание алгоритма:

Сначала мы проверяем, больше ли размер массива единицы. Если он меньше или равен одному, то просто возвращаем исходный массив. Если размер массива больше одного, мы начинаем цикл, который проходит от второго элемента до последнего. В этом цикле мы сохраняем текущий элемент и начинаем проверку от него к началу массива (в уже отсортированную часть). Если текущий элемент меньше элемента, с которым мы сравниваем, мы сдвигаем этот элемент вправо.

Этот процесс продолжается до тех пор, пока текущий элемент не окажется на своем месте.

1. Устойчивость сортировки:

Сортировка является устойчивой, поскольку элементы передвигаются правее лишь в том случае, если элемент, который ищет свое место является строго меньше, то есть, если у нас в массиве содержится два элемента с одинаковым значением, то местами они не поменяются.

1. Расчет функции временной сложности и ее асимптотическая оценка:

Для лучшего случая –

Для худшего случая –

Для среднего случая –

1. Расчет функции пространственной сложности и ее асимптотическая оценка:

Для лучшего случая – , поскольку никаких дополнительных рекурсивных вызовов или инициализаций массива не происходит.

Для худшего случая – , поскольку никаких дополнительных рекурсивных вызовов или инициализаций массива не происходит.

Для среднего случая – , поскольку никаких дополнительных рекурсивных вызовов или инициализаций массива не происходит.

1. Таблица значений:

Таблица 1 – Оценка временной и пространственной сложности для сортировки вставками

|  |  |  |  |
| --- | --- | --- | --- |
| Сложность | Лучшая | Худшая | Средняя |
| Временная |  |  |  |
| Пространственная |  |  |  |

1. Графики функции временной сложности для разных случаев:

Рисунок 1 – График функции лучшего случая сортировки вставками

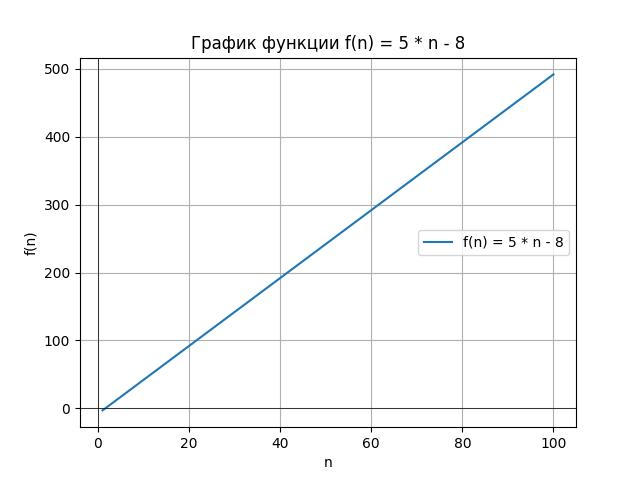


Рисунок 2 – График функции худшего случая сортировки вставками

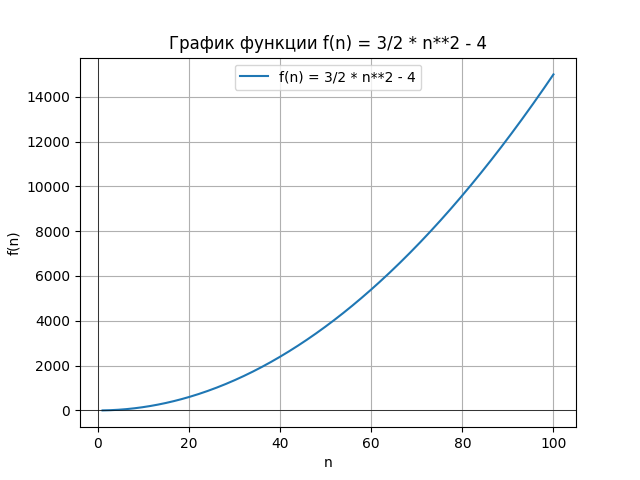
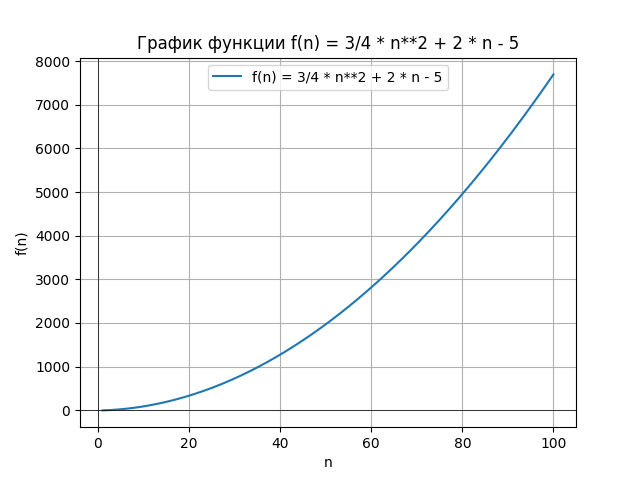


Рисунок 3 – График функции среднего случая сортировки вставками



# Сортировка выбором

1. Описание алгоритма:

Сначала проверяем, больше ли размер массива единицы. Если меньше или равен одному, возвращаем массив. Иначе начинаем цикл, который проходит по массиву. Для каждого элемента находим минимальный элемент в неотсортированной части массива и меняем его местами с текущим элементом. Так продолжается до тех пор, пока все элементы не будут отсортированы.

1. Устойчивость сортировки:

Сортировка является устойчивой, поскольку индекс наименьшего элемента меняется лишь в том случае, если новый элемент строго меньше прошлого. Таким образом, если в массиве будет два и более элементов с одинаковым значением, то местами они не поменяются.

1. Расчет функции временной сложности и ее асимптотическая оценка:

Для лучшего случая –

Для худшего случая –

Для среднего случая –

1. Расчет функции пространственной сложности и ее асимптотическая оценка:

Для лучшего случая – , поскольку никаких дополнительных рекурсивных вызовов или инициализаций массива не происходит.

Для худшего случая – , поскольку никаких дополнительных рекурсивных вызовов или инициализаций массива не происходит.

Для среднего случая – , поскольку никаких дополнительных рекурсивных вызовов или инициализаций массива не происходит.

1. Таблица значений:

Таблица 2 – Оценка временной и пространственной сложности для сортировки выбором

|  |  |  |  |
| --- | --- | --- | --- |
| Сложность | Лучшая | Худшая | Средняя |
| Временная |  |  |  |
| Пространственная |  |  |  |

1. График функции временной сложности для разных случаев:

Рисунок 4 – График функции лучшего случая сортировки выбором

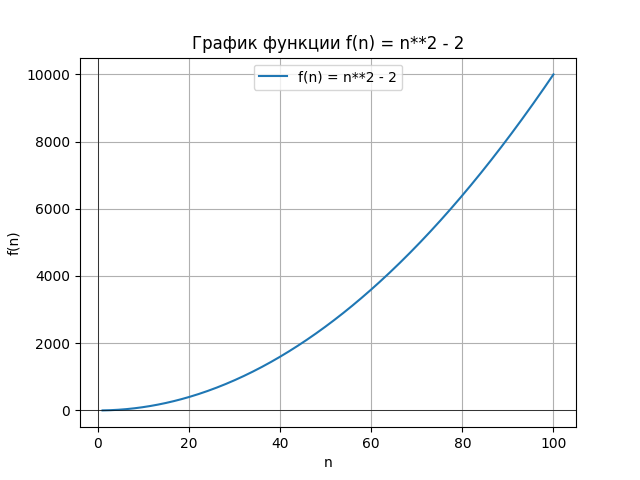


Рисунок 5 – График функции худшего случая сортировки выбором

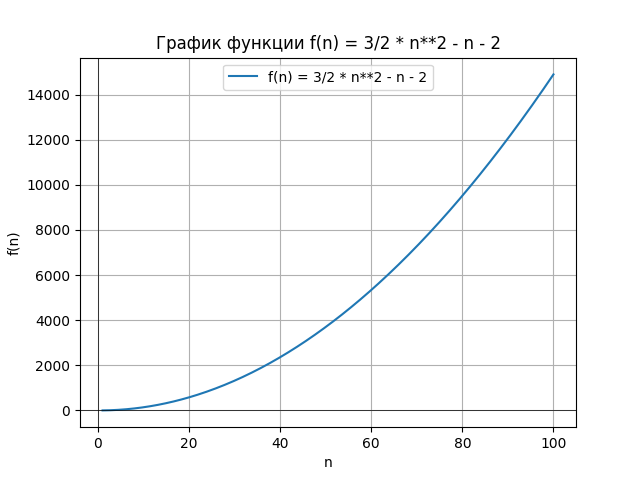
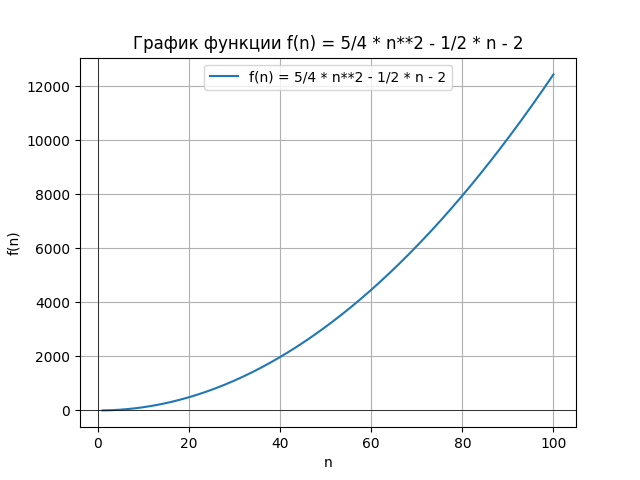


Рисунок 6 – График функции среднего случая сортировки выбором



# Сортировка пузырьком

1. Описание алгоритма:

Проверяем, больше ли размер массива единицы. Если меньше или равен одному, возвращаем массив. Иначе начинаем цикл, который на каждой итерации сравнивает соседние элементы и меняет их местами, если они стоят в неправильном порядке. Если на одной из итераций обменов не произошло, значит массив уже отсортирован, и мы завершаем работу.

1. Устойчивость сортировки:

Сортировка пузырьком является устойчивой, поскольку сравнение элементов для смены местами проводится строго, то есть два элемента с одинаковым значением сдвинуты не будут.

1. Расчет функции временной сложности и ее асимптотическая оценка:

Для лучшего случая –

Для худшего случая –

Для среднего случая –

1. Расчет функции пространственной сложности и ее асимптотическая оценка:

Для лучшего случая – , поскольку никаких дополнительных рекурсивных вызовов или инициализаций массива не происходит.

Для худшего случая – , поскольку никаких дополнительных рекурсивных вызовов или инициализаций массива не происходит.

Для среднего случая – , поскольку никаких дополнительных рекурсивных вызовов или инициализаций массива не происходит.

1. Таблица значений:

Таблица 3 – Оценка временной и пространственной сложности для сортировки пузырьком

|  |  |  |  |
| --- | --- | --- | --- |
| Сложность | Лучшая | Худшая | Средняя |
| Временная |  |  |  |
| Пространственная |  |  |  |

1. График функции временной сложности для разных случаев:

Рисунок 7 – График функции лучшего случая сортировки пузырьком

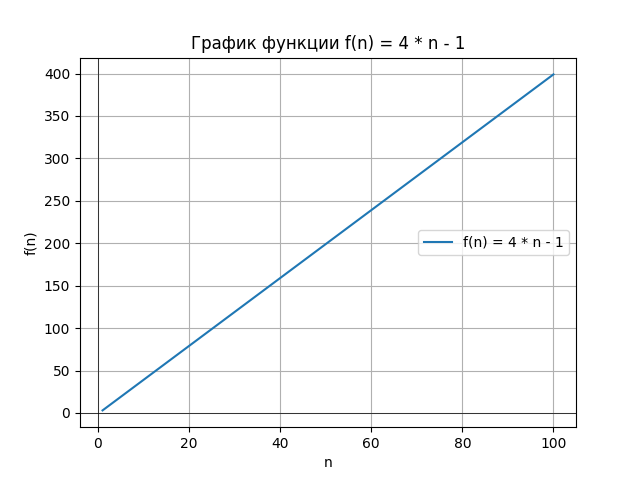


Рисунок 8 – График функции худшего случая сортировки пузырьком

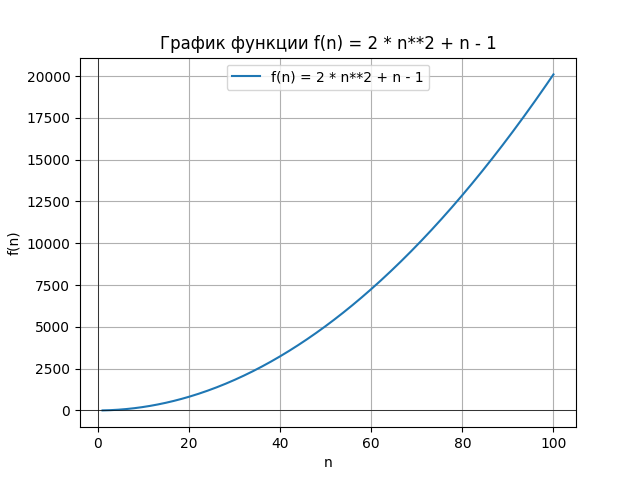
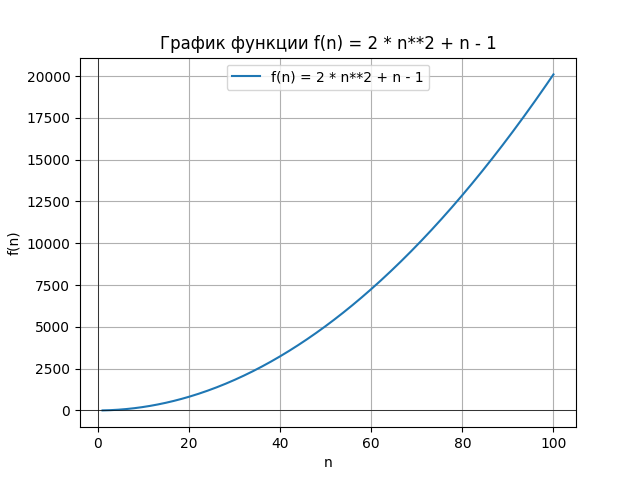


Рисунок 9 – График функции среднего случая сортировки пузырьком



# Сортировка слиянием

1. Описание алгоритма:

Функция проверяет, является ли размер массива больше одного. Если размер равен единице или меньше, то массив уже отсортирован, и функция возвращает его. Если размер больше одного, массив делится на две части: левую и правую. Эти две части сортируются рекурсивно с помощью той же функции. После этого происходит слияние двух отсортированных половин в один новый массив. В процессе слияния элементы из каждой части сравниваются и поочередно вставляются в новый массив. После завершения слияния новый массив копируется обратно в исходный, а временные массивы освобождаются из памяти.

1. Устойчивость сортировки:

Сортировка слиянием также будет являться устойчивой, поскольку все сравнения производятся строгие. То есть два элемента с одинаковыми значениями останутся в таком же порядке, в каком были изначально.

1. Расчет функции временной сложности и ее асимптотическая оценка:

Для лучшего случая – . В данном случае нам понадобится применение мастер теоремы. В нашем случае , тогда , а . Так как мастер теорема имеет вид: , то нам подходит второй случай: . Таким образом получаем, что для лучшего случая

Для худшего случая – , так как функция и алгоритм никак не отличается от лучшего случая

Для среднего случая – , так как функция в целом никак не отличается от двух предыдущих случаев

1. Расчет функции пространственной сложности и ее асимптотическая оценка:

Для лучшего случая – , мы получили сложность для создания и инициализации переменных и массивов, теперь рассчитаем затраты памяти на рекурсию: , поскольку вызов для рекурсии для каждого вызова уменьшается вдвое, мы можем записать ее значение как . Тогда итоговая сложность будет , так как каждый массив у нас используется только внутри одной рекурсии.

Для худшего случая – , поскольку худший случай ничем не отличается от лучшего.

Для среднего случая – , поскольку средний случай ничем не отличается от лучшего и худшего.

1. Таблица значений:

Таблица 4 – Оценка временной и пространственной сложности для сортировки слиянием

|  |  |  |  |
| --- | --- | --- | --- |
| Сложность | Лучшая | Худшая | Средняя |
| Временная |  |  |  |
| Пространственная |  |  |  |

1. График функции временной сложности для разных случаев:

Рисунок 10 – График функции лучшего случая сортировки слиянием

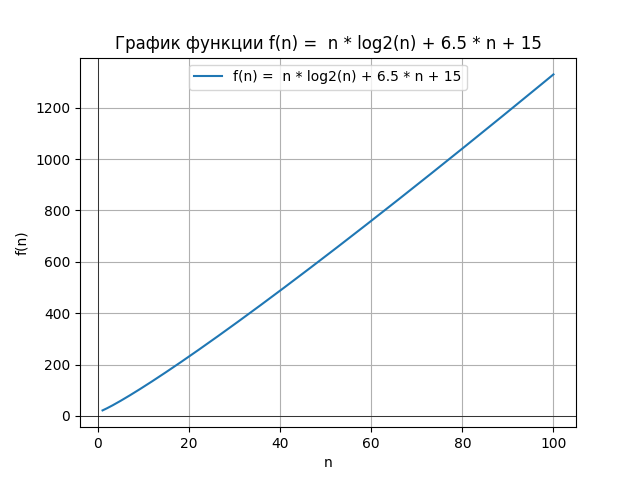


Рисунок 11 – График функции худшего случая сортировки слиянием

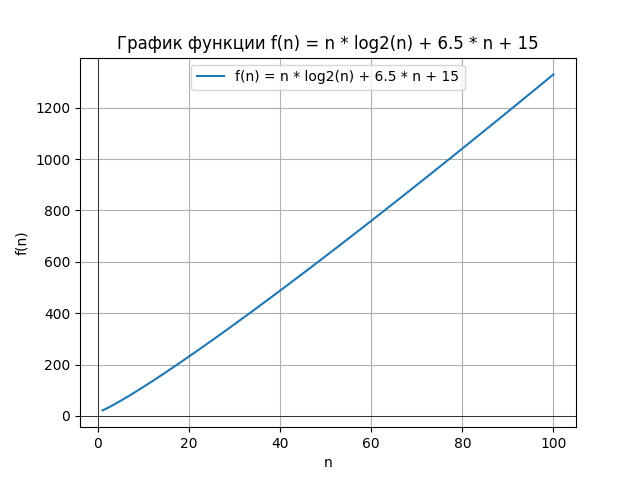
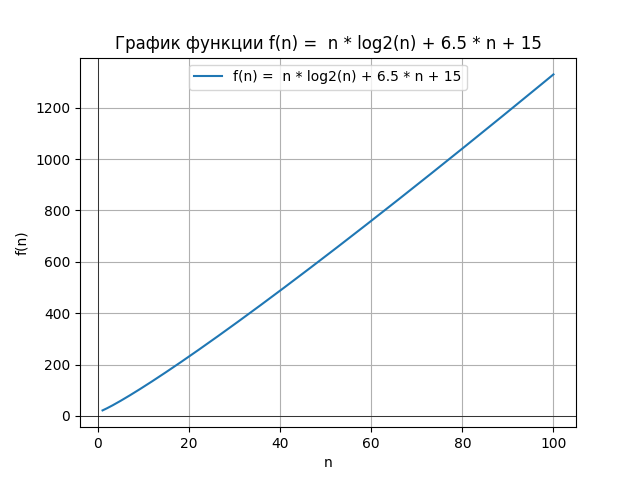


Рисунок 12 – График функции среднего случая сортировки слиянием



# Быстрая сортировка

1. Описание алгоритма:

Если размер массива меньше или равен единице, он уже отсортирован. В противном случае выбирается случайный опорный элемент. Массив делится на три части: элементы, которые меньше опорного, равные ему и больше него. Затем левая и правая части сортируются рекурсивно. После этого отсортированные части объединяются в один массив, и возвращается результат. В конце все временные массивы освобождаются из памяти.

1. Устойчивость сортировки:

Быстрая сортировка в ее начальном виде не является устойчивой, но поскольку в моей сортировке я делаю разбиение на три массива вместо двух, то при каждом рекурсивном вызове так или иначе одинаковые символы рано или поздно останутся вдвоем, после чего запишутся в один массив в их начальном порядке. Таким образом быстрая сортировка в моем случае является устойчивой.

1. Расчет функции временной сложности и ее асимптотическая оценка:

Для лучшего случая – . В данном случае нам понадобится применение мастер теоремы. В нашем случае , тогда , а . Так как мастер теорема имеет вид: , то нам подходит третий случай: . Таким образом получаем, что для лучшего случая

Для худшего случая – , так как функция имеет в себе рекуррентный вызов , то применим метод разложения: и так далее. Тогда заметим, что в конечном итоге будет вызываться раз. Итоговая сложность для худшего случая:

Для среднего случая – , так как функция имеет в себе рекурсивный вызов, то нам надо снова использовать мастер теорему, в этот раз , что дает нам , тогда как . Получается, что это второй случай мастер теоремы: . Следовательно, для среднего случая .

1. Расчет функции пространственной сложности и ее асимптотическая оценка:

Для лучшего случая – , мы получили сложность для создания и инициализации переменных и массивов, теперь рассчитаем затраты памяти на рекурсию: , поскольку вызов для рекурсии для каждого вызова уменьшается втрое, мы можем записать ее значение как , и по свойству логарифма . Тогда итоговая сложность будет .

Для худшего случая – , теперь рассчитаем затраты памяти на рекурсию: . Тогда итоговая сложность будет , так как каждый массив у нас будет инициализироваться.

Для среднего случая – , поскольку средний случай практически ничем не отличается от лучшего, кроме того, что количество рекуррентных вызовов будет не . То есть нам просто не нужно будет приводить логарифм к нужному виду.

1. Таблица значений:

Таблица 5 – Оценка временной и пространственной сложности для быстрой сортировки

|  |  |  |  |
| --- | --- | --- | --- |
| Сложность | Лучшая | Худшая | Средняя |
| Временная |  |  |  |
| Пространственная |  |  |  |

1. График функции временной сложности для разных случаев:

Рисунок 13 – График функции лучшего случая быстрой сортировки

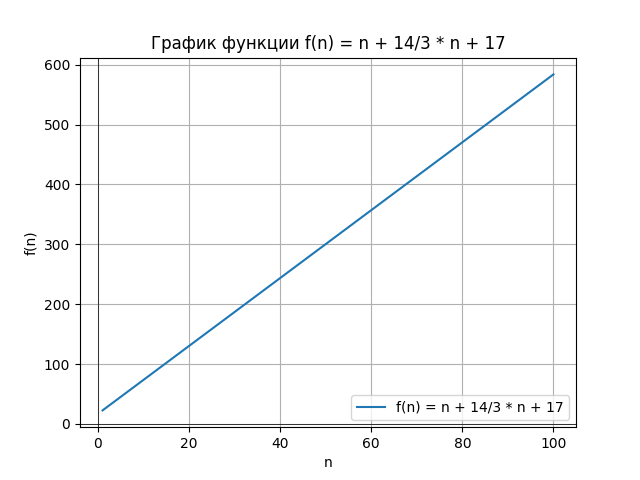


Рисунок 14 – График функции худшего случая быстрой сортировки

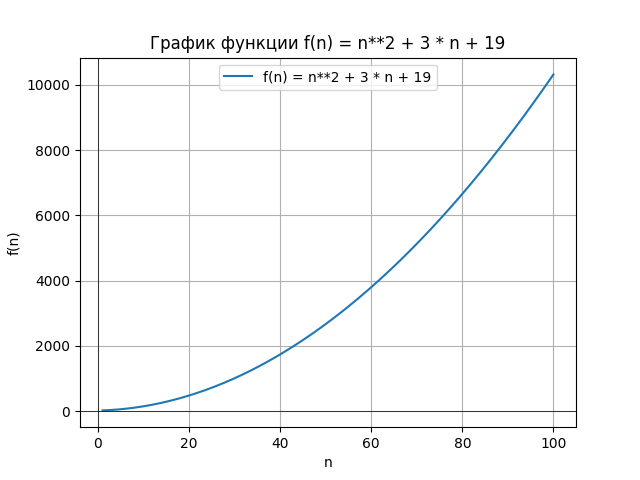
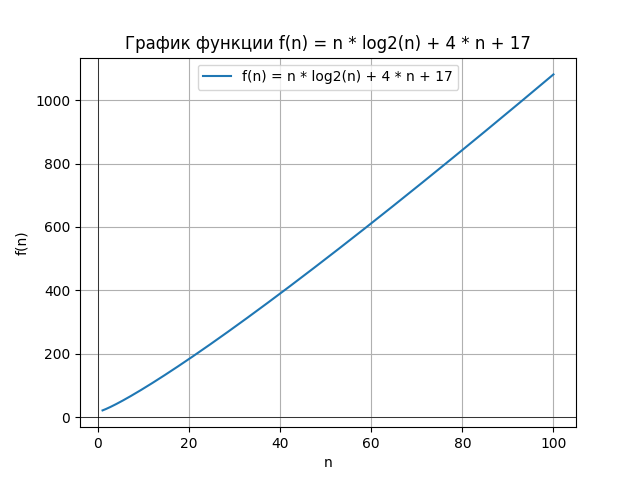


Рисунок 15 – График функции среднего случая быстрой сортировки



# Пирамидальная сортировка

1. Описание алгоритма:

Если размер массива меньше или равен единице, он уже отсортирован. В противном случае алгоритм сначала строит структуру данных, называемую кучей, чтобы упорядочить элементы. Начинается с последнего элемента и постепенно перестраивает массив так, чтобы каждый родительский элемент был больше своих детей.

После того как куча создана, максимальный элемент, находящийся в корне, перемещается в конец массива. Затем структура снова перестраивается, чтобы сохранить порядок. Этот процесс повторяется, пока все элементы не окажутся на своих местах. В итоге возвращается отсортированный массив.

1. Устойчивость сортировки:

Пирамидальная, или другими словами, сортировка кучей не является устойчивой, поскольку если элемент будет являться наибольшем в массиве и будет иметь еще несколько своих копий, то они в конечном счете скорее всего поменяются местами, поскольку начальный наибольший элемент уйдет в конец. То есть сортировка кучей не будет устойчива.

1. Расчет функции временной сложности и ее асимптотическая оценка:

Для лучшего случая –

Для худшего случая –

Для среднего случая –

1. Расчет функции пространственной сложности и ее асимптотическая оценка:

Для лучшего случая – , поскольку ни одной рекурсии совершено не будет в лучшем случае.

Для худшего случая – , поскольку будет совершено рекурсивных вызовов функции.

Для среднего случая – , поскольку будет совершено вызовов, поскольку в среднем случае перемещение элементов будет длится до половины дерева.

1. Таблица значений:

Таблица 6 – Оценка временной и пространственной сложности для пирамидальной сортировки

|  |  |  |  |
| --- | --- | --- | --- |
| Сложность | Лучшая | Худшая | Средняя |
| Временная |  |  |  |
| Пространственная |  |  |  |

1. График функции временной сложности для разных случаев:

Графики отсутствуют.

# Сортировка Шелла с последовательностью Шелла

1. Описание алгоритма:

Если размер массива меньше или равен единице, то он уже отсортирован. Если же нет, то алгоритм начинает проход по массиву, начиная с элемента с индексом шага. После чего идет такой же алгоритм, как и в сортировке вставками, то есть копируется элемент под текущем индексом, после чего идет сдвиг вправо элементов левее текущего, пока начальный элемент не найдет свое место. Единственная разница сортировки Шелла от сортировки вставками в том, что сортировка Шелла с последовательностью Шелла идет не каждый элемент, а с определенным шагом, начиная от до .

1. Устойчивость сортировки:

Сортировка Шелла не является устойчивой, поскольку из-за разных шагов она может поменять местами элементы с одинаковыми значениями, из-за чего и нарушается устойчивость.

1. Расчет функции временной сложности и ее асимптотическая оценка:

Для лучшего случая –

Для худшего случая –

Для среднего случая –

1. Расчет функции пространственной сложности и ее асимптотическая оценка:

Для лучшего случая – , поскольку никаких переменных и массивов, зависящих от тут нет, а также ни одной рекурсии совершено не будет.

Для худшего случая – , поскольку никаких переменных и массивов, зависящих от тут нет, а также ни одной рекурсии совершено не будет.

Для среднего случая – , поскольку никаких переменных и массивов, зависящих от тут нет, а также ни одной рекурсии совершено не будет.

1. Таблица значений:

Таблица 7 – Оценка временной и пространственной сложности для сортировки Шелла с последовательностью Шелла

|  |  |  |  |
| --- | --- | --- | --- |
| Сложность | Лучшая | Худшая | Средняя |
| Временная |  |  |  |
| Пространственная |  |  |  |

1. График функции временной сложности для разных случаев:

Графики отсутствуют.

# Сортировка Шелла с последовательностью Пратта

1. Описание алгоритма:

Если размер массива меньше или равен единице, то он уже отсортирован. Если же нет, то алгоритм начинает проход по массиву, начиная с элемента с индексом шага. После чего идет такой же алгоритм, как и в сортировке вставками, то есть копируется элемент под текущем индексом, после чего идет сдвиг вправо элементов левее текущего, пока начальный элемент не найдет свое место. Единственная разница сортировки Шелла от сортировки вставками в том, что сортировка Шелла с последовательностью Шелла идет не каждый элемент, а с определенным шагом, начиная от до .

1. Устойчивость сортировки:

Сортировка Шелла не является устойчивой, поскольку из-за разных шагов она может поменять местами элементы с одинаковыми значениями, из-за чего и нарушается устойчивость.

1. Расчет функции временной сложности и ее асимптотическая оценка:

Для лучшего случая –

Для худшего случая –

Для среднего случая –

1. Расчет функции пространственной сложности и ее асимптотическая оценка:

Для лучшего случая – , поскольку никаких переменных и массивов, зависящих от тут нет, а также ни одной рекурсии совершено не будет.

Для худшего случая – , поскольку никаких переменных и массивов, зависящих от тут нет, а также ни одной рекурсии совершено не будет.

Для среднего случая – , поскольку никаких переменных и массивов, зависящих от тут нет, а также ни одной рекурсии совершено не будет.

1. Таблица значений:

Таблица 8 – Оценка временной и пространственной сложности для сортировки Шелла с последовательностью Пратта

|  |  |  |  |
| --- | --- | --- | --- |
| Сложность | Лучшая | Худшая | Средняя |
| Временная |  |  |  |
| Пространственная |  |  |  |

1. График функции временной сложности для разных случаев:

Графики отсутствуют.

# Сортировка Шелла с последовательностью Хиббарда

1. Описание алгоритма:

Если размер массива меньше или равен единице, то он уже отсортирован. Если же нет, то алгоритм начинает проход по массиву, начиная с элемента с индексом шага. После чего идет такой же алгоритм, как и в сортировке вставками, то есть копируется элемент под текущем индексом, после чего идет сдвиг вправо элементов левее текущего, пока начальный элемент не найдет свое место. Единственная разница сортировки Шелла от сортировки вставками в том, что сортировка Шелла с последовательностью Шелла идет не каждый элемент, а с определенным шагом, начиная от до .

1. Устойчивость сортировки:

Сортировка Шелла не является устойчивой, поскольку из-за разных шагов она может поменять местами элементы с одинаковыми значениями, из-за чего и нарушается устойчивость.

1. Расчет функции временной сложности и ее асимптотическая оценка:

Для лучшего случая –

Для худшего случая –

Для среднего случая –

1. Расчет функции пространственной сложности и ее асимптотическая оценка:

Для лучшего случая – , поскольку никаких переменных и массивов, зависящих от тут нет, а также ни одной рекурсии совершено не будет.

Для худшего случая – , поскольку никаких переменных и массивов, зависящих от тут нет, а также ни одной рекурсии совершено не будет.

Для среднего случая – , поскольку никаких переменных и массивов, зависящих от тут нет, а также ни одной рекурсии совершено не будет.

1. Таблица значений:

Таблица 9 – Оценка временной и пространственной сложности для сортировки Шелла с последовательностью Хиббарда

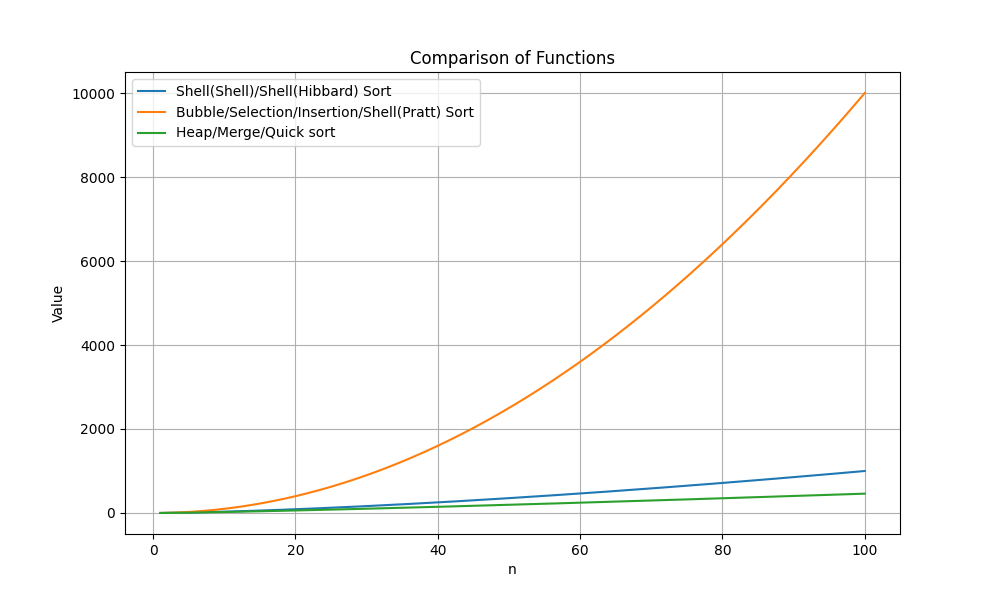
|  |  |  |  |
| --- | --- | --- | --- |
| Сложность | Лучшая | Худшая | Средняя |
| Временная |  |  |  |
| Пространственная |  |  |  |

1. График функции временной сложности для разных случаев:

Графики отсутствуют.

# Общий график

Рисунок 16 – Общий график для всех сортировок в среднем случае



Смотря на график всех сортировок в среднем случае, можно сделать вывод, что быстрее всего на 100000 и более элементах будет работать одна из следующих сортировок: пирамидальная, слиянием, быстрая. Проанализировав то, какие ситуации требуются для лучшего и худшего случая каждой из сортировок, а так же оценив их временную сложность, можно попытаться предположить, что в среднем, на массивах размером более 100000 элементов быстрее всего будет работать быстрая сортировка, поскольку в лучшем случае у нее лучшая временная сложность, а достичь худшего случая очень тяжело.

# Практическая часть.

# Сортировка вставками

Таблица 10 – Экспериментально полученные значения для сортировки вставками

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов | Время выполнения (в секундах) | | | |
| Sorted | Reversed | 90% sorted | Random |
| 2500 | 7.8e-06 | 0.0053418 | 5.96667e-06 | 0.00353873 |
| 5000 | 1.76667e-05 | 0.0222097 | 1.24e-05 | 0.0116738 |
| 7500 | 1.99e-05 | 0.0503482 | 1.75667e-05 | 0.0254371 |
| 10000 | 2.33e-05 | 0.0868919 | 3.45e-05 | 0.0435003 |
| 12500 | 2.99333e-05 | 0.134105 | 3.62333e-05 | 0.0684211 |
| 15000 | 3.92e-05 | 0.201411 | 4.04e-05 | 0.100557 |
| 17500 | 4.32e-05 | 0.266344 | 5.78e-05 | 0.134399 |
| 20000 | 4.91667e-05 | 0.250462 | 8.16333e-05 | 0.173278 |
| 22500 | 5.25333e-05 | 0.349503 | 9.81e-05 | 0.22206 |
| 25000 | 5.83e-05 | 0.520923 | 9.43333e-05 | 0.200145 |
| 30000 | 7.34333e-05 | 0.780344 | 0.0001455 | 0.372772 |
| 35000 | 8.51667e-05 | 0.786663 | 0.0001668 | 0.509417 |
| 40000 | 0.000106033 | 1.34124 | 0.000223133 | 0.453354 |
| 45000 | 0.000105567 | 1.49652 | 0.0002577 | 0.580024 |
| 50000 | 0.000119967 | 2.00886 | 0.000332433 | 0.71425 |
| 60000 | 0.000153067 | 3.09895 | 0.0004108 | 0.985397 |
| 70000 | 0.000176867 | 3.46001 | 0.0007079 | 1.98902 |
| 80000 | 0.000199067 | 4.60548 | 0.000773267 | 2.1418 |
| 90000 | 0.000233667 | 5.93559 | 0.000816433 | 3.44139 |
| 100000 | 0.000256033 | 7.02813 | 0.0013205 | 4.33916 |
| 110000 | 0.000298767 | 8.31763 | 0.00138843 | 4.02578 |
| 120000 | 0.0003857 | 10.2062 | 0.00121923 | 5.98088 |
| 130000 | 0.000414933 | 12.904 | 0.00178957 | 5.74784 |
| 140000 | 0.000650167 | 16.8879 | 0.00209867 | 6.39245 |
| 150000 | 0.0005723 | 15.698 | 0.00283153 | 7.16721 |

Рисунок 16 – График экспериментальных значений для сортировки вставками

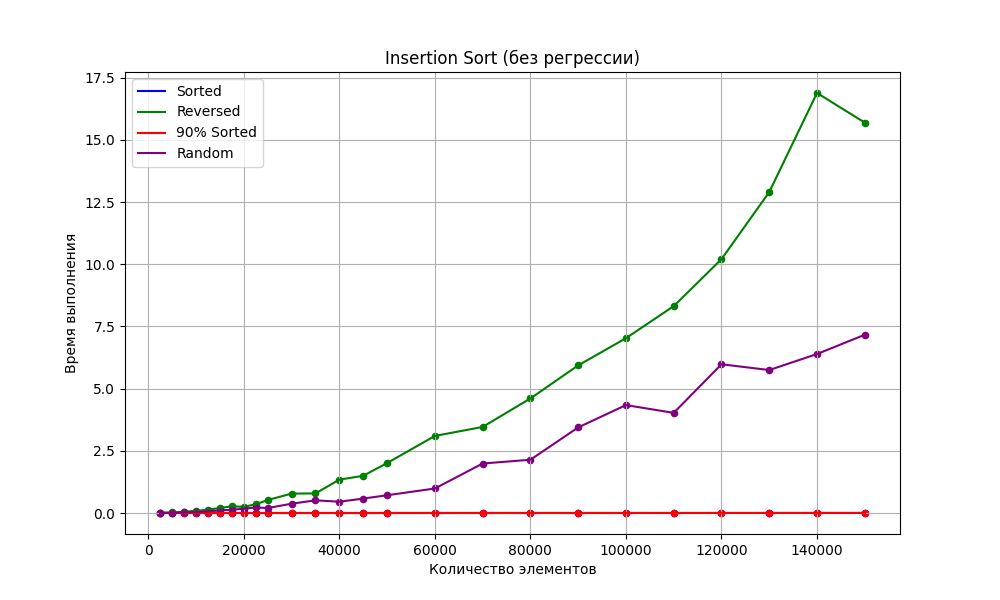
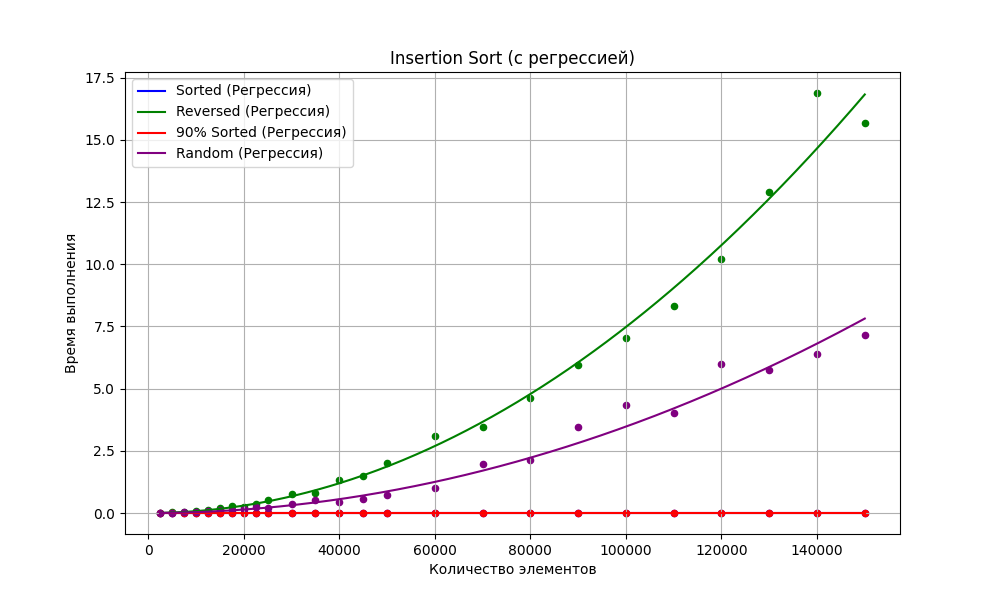


Рисунок 17 – График экспериментальных значений с регрессией для сортировки вставками



Проанализировав график полученных экспериментальных значений, мы можем сказать, что при построении графика с регрессионной кривой наш график соответствует высчитанной асимптотической оценке, описанной в теоретической части. Конечно, есть некоторые скачки, которые зависят от процессора и параллельной работы с сортировкой, поэтому в целом такие погрешности – нормальное явление и не сильно влияют на общую функцию.

# Сортировка выбором

Таблица 11 – Экспериментально полученные значения для сортировки выбором

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов | Время выполнения (в секундах) | | | |
| Sorted | Reversed | 90% sorted | Random |
| 2500 | 0.00552953 | 0.00346937 | 0.0064429 | 0.00309813 |
| 5000 | 0.0188636 | 0.0263267 | 0.02035 | 0.0113649 |
| 7500 | 0.022102 | 0.0693671 | 0.0444665 | 0.0217437 |
| 10000 | 0.0374313 | 0.137217 | 0.0808887 | 0.0378638 |
| 12500 | 0.0579741 | 0.211447 | 0.122247 | 0.0591419 |
| 15000 | 0.100279 | 0.352858 | 0.179921 | 0.0855371 |
| 17500 | 0.119016 | 0.460489 | 0.239464 | 0.116852 |
| 20000 | 0.235146 | 0.412158 | 0.318912 | 0.153841 |
| 22500 | 0.407697 | 0.942698 | 0.397749 | 0.19818 |
| 25000 | 0.265429 | 1.20115 | 0.298103 | 0.245094 |
| 30000 | 0.69486 | 1.72883 | 0.727746 | 0.353359 |
| 35000 | 0.701241 | 2.03813 | 0.902571 | 0.475537 |
| 40000 | 0.833513 | 2.127 | 0.616036 | 0.619499 |
| 45000 | 0.850729 | 2.79345 | 0.768755 | 0.783055 |
| 50000 | 1.32359 | 3.29973 | 0.953867 | 0.968131 |
| 60000 | 1.54177 | 4.32484 | 1.40583 | 1.40527 |
| 70000 | 2.89636 | 6.38915 | 2.04094 | 1.89644 |
| 80000 | 4.18744 | 7.58283 | 4.73258 | 2.47787 |
| 90000 | 4.79747 | 9.61287 | 4.69984 | 3.07072 |
| 100000 | 6.62845 | 10.6359 | 6.13557 | 3.80179 |
| 110000 | 5.92054 | 12.8441 | 9.22324 | 4.58517 |
| 120000 | 5.46097 | 10.5943 | 8.31787 | 5.53007 |
| 130000 | 6.38412 | 12.3088 | 8.86065 | 6.50435 |
| 140000 | 7.5734 | 15.8991 | 12.0061 | 7.50912 |
| 150000 | 11.5902 | 21.1032 | 12.328 | 8.76669 |

Рисунок 18 – График экспериментальных значений для сортировки выбором

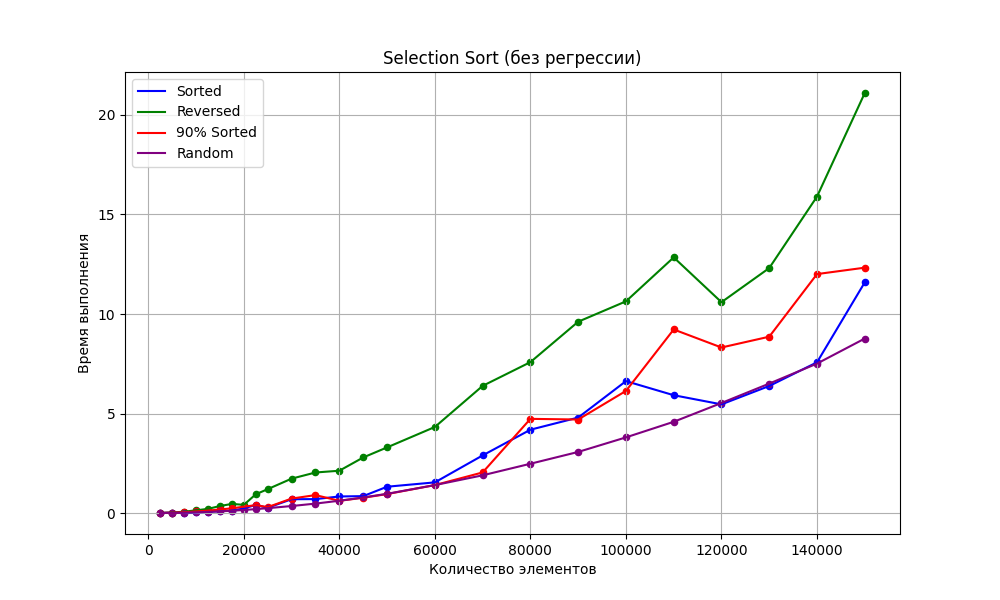
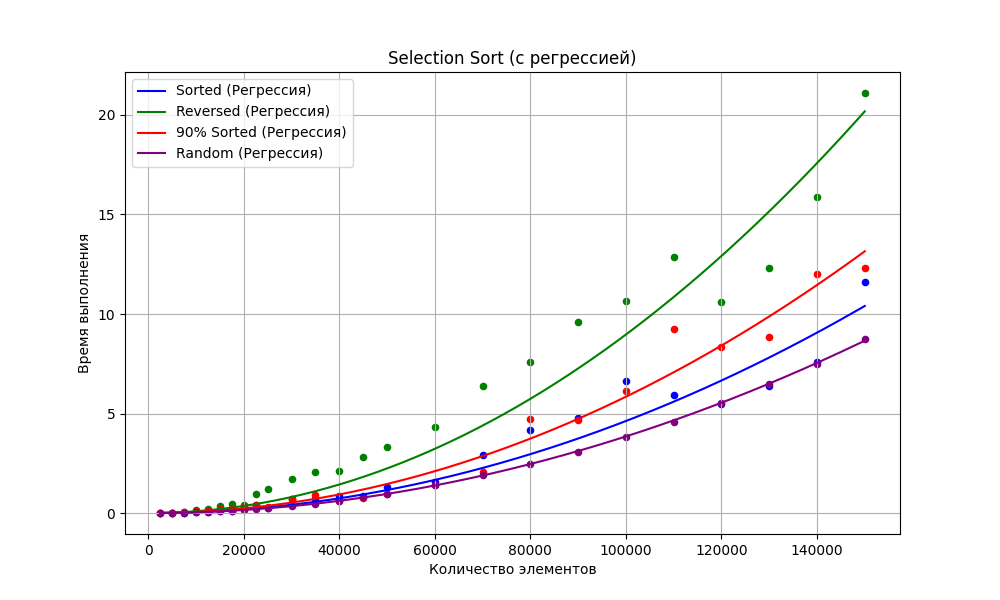


Рисунок 19 – График экспериментальных значений с регрессией для сортировки выбором



Проанализировав графики полученных экспериментальных значений, мы можем наблюдать некоторые отклонения и скачки в точках. Это может быть связано с тем, что процессор был нагружен какими-то другими процессами, помимо сортировок, но общая функция, пусть и имеет отклонения, но по асимптотике она соответствует рассчитанным значениям.

# Сортировка пузырьком

Таблица 12 – Экспериментально полученные значения для сортировки пузырьком

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов | Время выполнения (в секундах) | | | |
| Sorted | Reversed | 90% sorted | Random |
| 2500 | 2.43333e-06 | 0.00907353 | 5.76667e-06 | 0.00818443 |
| 5000 | 4.43333e-06 | 0.0357816 | 1.11333e-05 | 0.0340396 |
| 7500 | 6.3e-06 | 0.0816999 | 1.67e-05 | 0.0856795 |
| 10000 | 9.73333e-06 | 0.144741 | 0.123456 | 0.16265 |
| 12500 | 1.11667e-05 | 0.227057 | 0.182963 | 0.266267 |
| 15000 | 1.41667e-05 | 0.324827 | 0.275202 | 0.400601 |
| 17500 | 1.53667e-05 | 0.444401 | 0.209108 | 0.586246 |
| 20000 | 1.99333e-05 | 0.583175 | 0.388429 | 0.790265 |
| 22500 | 1.95e-05 | 0.744058 | 0.427592 | 0.982628 |
| 25000 | 2.17333e-05 | 0.901649 | 0.507888 | 1.23887 |
| 30000 | 2.56333e-05 | 1.35063 | 0.693369 | 1.81812 |
| 35000 | 2.97333e-05 | 1.83129 | 0.870862 | 2.52068 |
| 40000 | 3.46333e-05 | 2.3161 | 0.743782 | 3.33763 |
| 45000 | 3.62667e-05 | 2.98559 | 2.0107 | 4.23211 |
| 50000 | 4.08667e-05 | 3.68477 | 0.95861 | 5.25952 |
| 60000 | 4.89333e-05 | 5.22311 | 1.46301 | 7.6888 |
| 70000 | 6.19667e-05 | 7.20563 | 1.9685 | 10.4019 |
| 80000 | 6.72333e-05 | 9.3994 | 2.61831 | 13.7157 |
| 90000 | 8.16667e-05 | 20.8556 | 6.21875 | 22.6254 |
| 100000 | 9.04667e-05 | 24.6721 | 6.92409 | 31.6097 |
| 110000 | 9.40667e-05 | 30.1707 | 8.09619 | 32.5089 |
| 120000 | 0.000109 | 41.5159 | 10.2404 | 31.8375 |
| 130000 | 0.000113033 | 26.6398 | 12.8395 | 36.7798 |
| 140000 | 0.0001226 | 28.2599 | 8.00047 | 43.4807 |
| 150000 | 0.0001348 | 37.4205 | 9.09859 | 49.0221 |

Рисунок 20 – График экспериментальных значений для сортировки пузырьком

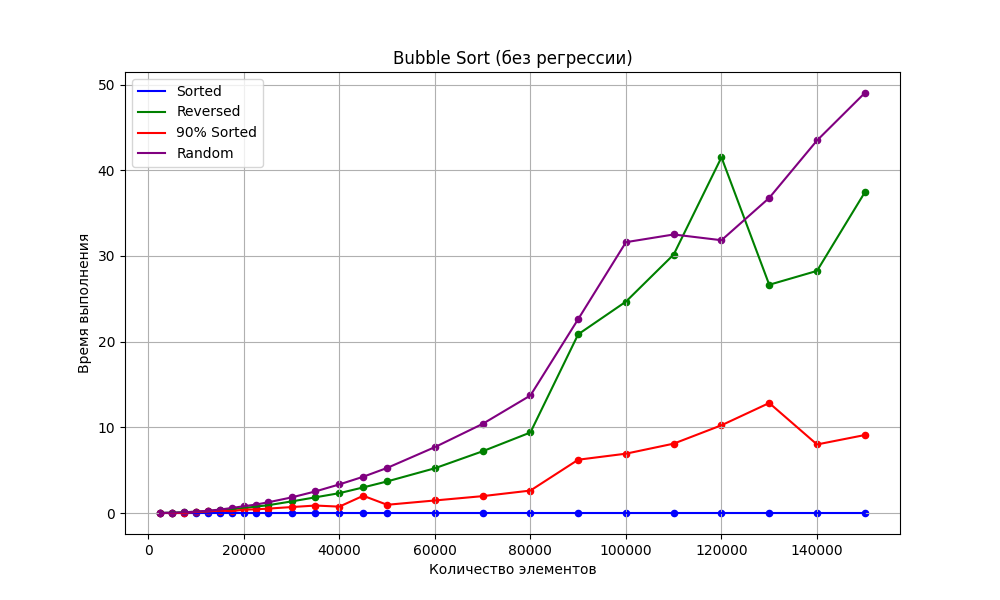
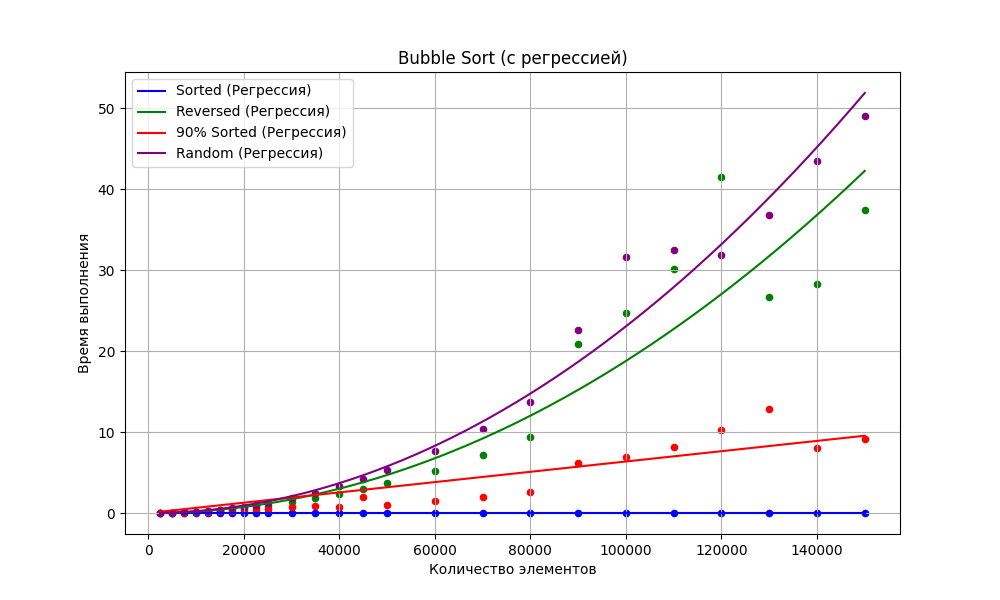


Рисунок 21 – График экспериментальных значений с регрессией для сортировки пузырьком



Сравнив график полученных экспериментальных значений, мы можем заметить, что у нас некоторое несоответствие, худший случай выполняется быстрее, чем массив со случайными элементами. Это не очень хорошо, но не критично, поскольку это зависит только от нагрузки и скорости процессора, скорее всего во время выполнения сортировки пузырьком, был неожиданно запущен какой-то процесс, который и произвел данное несоответствие. Но в целом построив график с использованием регрессии, мы можем понять, что он, хоть и с небольшими отклонениями, соответствует выраженной ранее сложности.

# Сортировка слиянием

Таблица 13 – Экспериментально полученные значения для сортировки слиянием

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов | Время выполнения (в секундах) | | | |
| Sorted | Reversed | 90% sorted | Random |
| 2500 | 0.0010081 | 0.0009986 | 0.000980367 | 0.00141967 |
| 5000 | 0.00200223 | 0.00203393 | 0.00204873 | 0.00262047 |
| 7500 | 0.0030201 | 0.0032351 | 0.00304573 | 0.00357497 |
| 10000 | 0.00416613 | 0.00419477 | 0.004413 | 0.0048978 |
| 12500 | 0.00523843 | 0.00534697 | 0.0053505 | 0.00623803 |
| 15000 | 0.006155 | 0.00623883 | 0.0067365 | 0.0076039 |
| 17500 | 0.00726027 | 0.0072749 | 0.0078486 | 0.00872223 |
| 20000 | 0.00820887 | 0.0089602 | 0.0088615 | 0.0102519 |
| 22500 | 0.0128045 | 0.00963677 | 0.0099708 | 0.011011 |
| 25000 | 0.011838 | 0.011057 | 0.0111798 | 0.0138023 |
| 30000 | 0.012709 | 0.0127285 | 0.0131416 | 0.0147128 |
| 35000 | 0.0149173 | 0.0148773 | 0.0153146 | 0.0173194 |
| 40000 | 0.0170361 | 0.0167529 | 0.0173133 | 0.0196434 |
| 45000 | 0.01911 | 0.0187605 | 0.019132 | 0.0221626 |
| 50000 | 0.0215324 | 0.0215814 | 0.0215394 | 0.0244826 |
| 60000 | 0.0253864 | 0.0257695 | 0.0259553 | 0.0292694 |
| 70000 | 0.0299041 | 0.0303706 | 0.0311594 | 0.0349527 |
| 80000 | 0.0359597 | 0.0331256 | 0.0352301 | 0.0398314 |
| 90000 | 0.0397994 | 0.0383518 | 0.0409825 | 0.0450375 |
| 100000 | 0.0448984 | 0.0428203 | 0.0465098 | 0.0499018 |
| 115000 | 0.0520202 | 0.0511027 | 0.0503704 | 0.0604699 |
| 130000 | 0.0645996 | 0.0557545 | 0.0557254 | 0.065814 |
| 145000 | 0.0775268 | 0.0642607 | 0.0638608 | 0.0762942 |
| 160000 | 0.0719981 | 0.0673522 | 0.0720138 | 0.0831885 |
| 175000 | 0.0793337 | 0.0755936 | 0.080161 | 0.0884969 |
| 190000 | 0.082737 | 0.08528 | 0.0881423 | 0.0991009 |
| 205000 | 0.0916077 | 0.0920574 | 0.0944673 | 0.109461 |
| 220000 | 0.0978029 | 0.0952131 | 0.101232 | 0.13312 |
| 235000 | 0.102902 | 0.102909 | 0.104413 | 0.215847 |
| 250000 | 0.108254 | 0.106347 | 0.114644 | 0.124673 |
| 265000 | 0.115896 | 0.115747 | 0.120859 | 0.134676 |
| 280000 | 0.123255 | 0.120575 | 0.126443 | 0.145397 |
| 295000 | 0.13006 | 0.12904 | 0.133998 | 0.153555 |

Рисунок 22 – График экспериментальных значений для сортировки слиянием

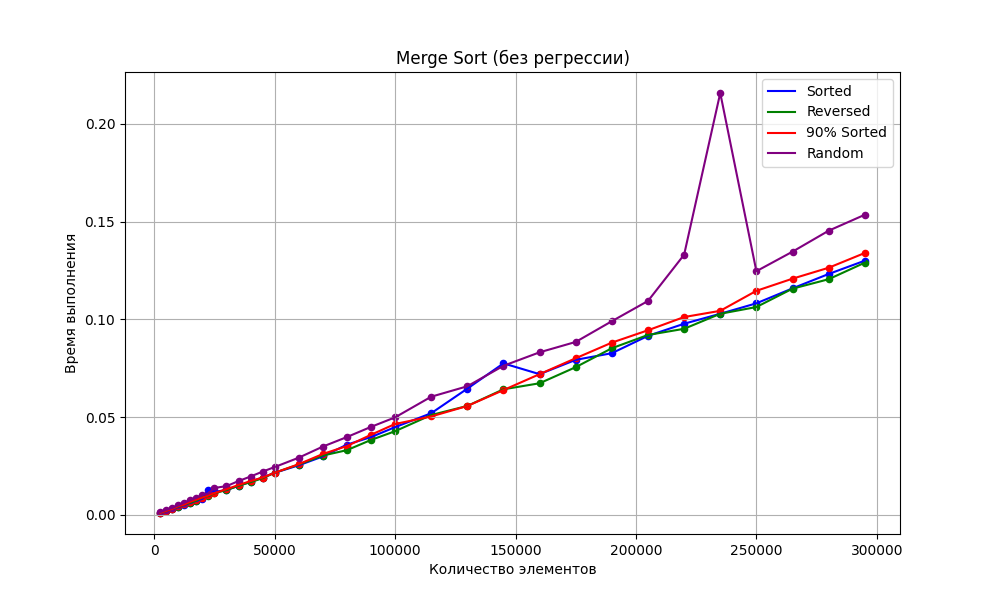
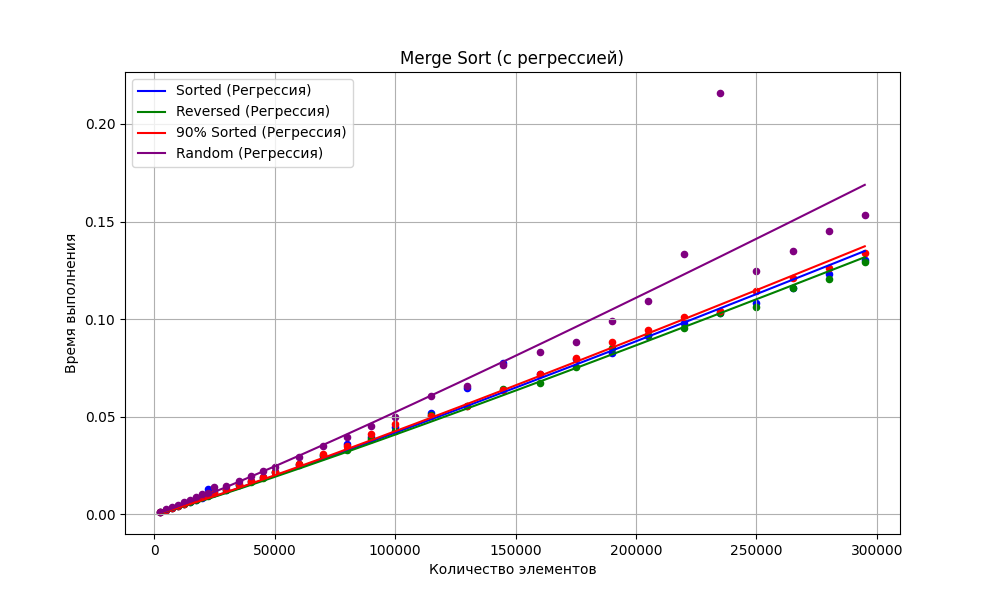


Рисунок 23 – График экспериментальных значений с регрессией для сортировки слиянием



Проведя анализ графиков экспериментальных значений для сортировки слиянием, мы можем заметить, что пусть тут и есть небольшой скачок для случайного массива, но в основном все остальные точки соответствуют ожиданиям, что и видно на график с регрессией.

# Быстрая сортировка

Таблица 14 – Экспериментально полученные значения для быстрой сортировки

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов | Время выполнения (в секундах) | | | |
| Sorted | Reversed | 90% sorted | Random |
| 2500 | 0.00152313 | 0.001471 | 0.0012201 | 0.00140307 |
| 5000 | 0.00270167 | 0.00290183 | 0.00304603 | 0.00349583 |
| 7500 | 0.00412083 | 0.00430593 | 0.00402393 | 0.00477363 |
| 10000 | 0.00490823 | 0.00567517 | 0.00607167 | 0.00642383 |
| 12500 | 0.0057087 | 0.00714807 | 0.00717553 | 0.0077702 |
| 15000 | 0.0065244 | 0.00867037 | 0.00834603 | 0.0089115 |
| 17500 | 0.00783177 | 0.0098951 | 0.00939317 | 0.0103744 |
| 20000 | 0.00851627 | 0.009274 | 0.0108591 | 0.0115069 |
| 22500 | 0.0097884 | 0.00947387 | 0.0135793 | 0.0123735 |
| 25000 | 0.0101494 | 0.0117065 | 0.0127145 | 0.013332 |
| 30000 | 0.0120926 | 0.0118685 | 0.0158539 | 0.0157861 |
| 35000 | 0.0158274 | 0.0138106 | 0.0183801 | 0.0177017 |
| 40000 | 0.0135779 | 0.0153772 | 0.0166683 | 0.0186091 |
| 45000 | 0.0146425 | 0.0189799 | 0.019113 | 0.0206449 |
| 50000 | 0.0161717 | 0.0206767 | 0.0215271 | 0.0223617 |
| 60000 | 0.0181241 | 0.0240486 | 0.0248804 | 0.0247004 |
| 70000 | 0.0187296 | 0.0241593 | 0.0259592 | 0.0255735 |
| 80000 | 0.0231898 | 0.0245912 | 0.0240035 | 0.0296451 |
| 90000 | 0.0222875 | 0.0285953 | 0.024993 | 0.0323485 |
| 100000 | 0.0231927 | 0.0304066 | 0.0263961 | 0.034797 |
| 115000 | 0.0243764 | 0.033574 | 0.0280562 | 0.0372083 |
| 130000 | 0.0279344 | 0.0319072 | 0.0306353 | 0.0401611 |
| 145000 | 0.0285767 | 0.0290128 | 0.0316516 | 0.0431349 |
| 160000 | 0.0343878 | 0.0312355 | 0.0337974 | 0.0454353 |
| 175000 | 0.0341664 | 0.0350346 | 0.034934 | 0.0495687 |
| 190000 | 0.039799 | 0.0354935 | 0.0381667 | 0.0511849 |
| 205000 | 0.0409496 | 0.116416 | 0.0420664 | 0.0492924 |
| 220000 | 0.0425463 | 0.0451784 | 0.0545194 | 0.0502809 |
| 235000 | 0.044334 | 0.0457544 | 0.0497699 | 0.0551559 |
| 250000 | 0.0450475 | 0.0454084 | 0.0580923 | 0.0559414 |
| 265000 | 0.0484991 | 0.0685452 | 0.0527253 | 0.0613315 |
| 280000 | 0.0481409 | 0.0528291 | 0.0522074 | 0.0622943 |
| 295000 | 0.0498294 | 0.049012 | 0.0773461 | 0.0632955 |

Рисунок 24 – График экспериментальных значений для быстрой сортировки

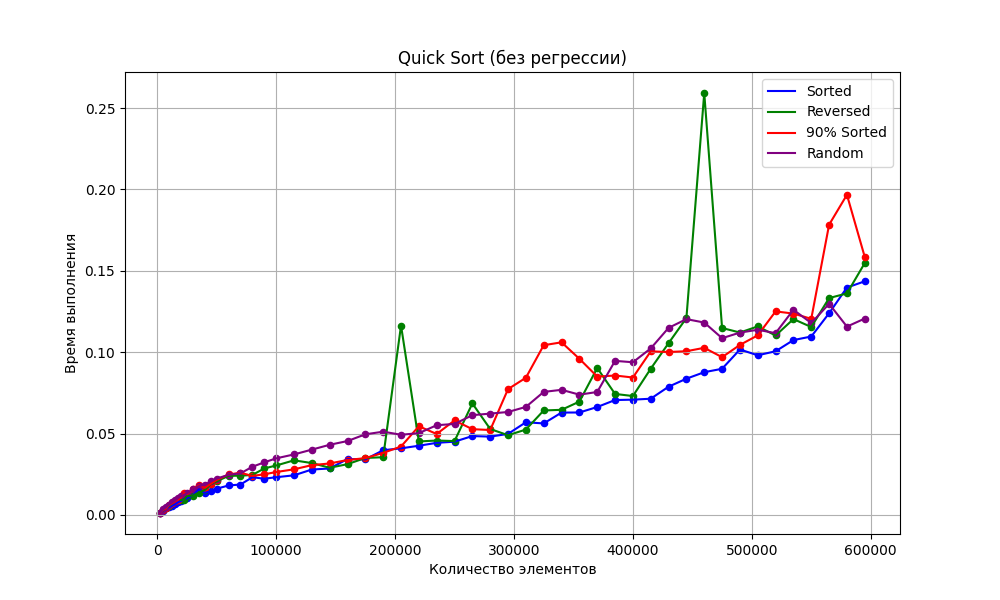
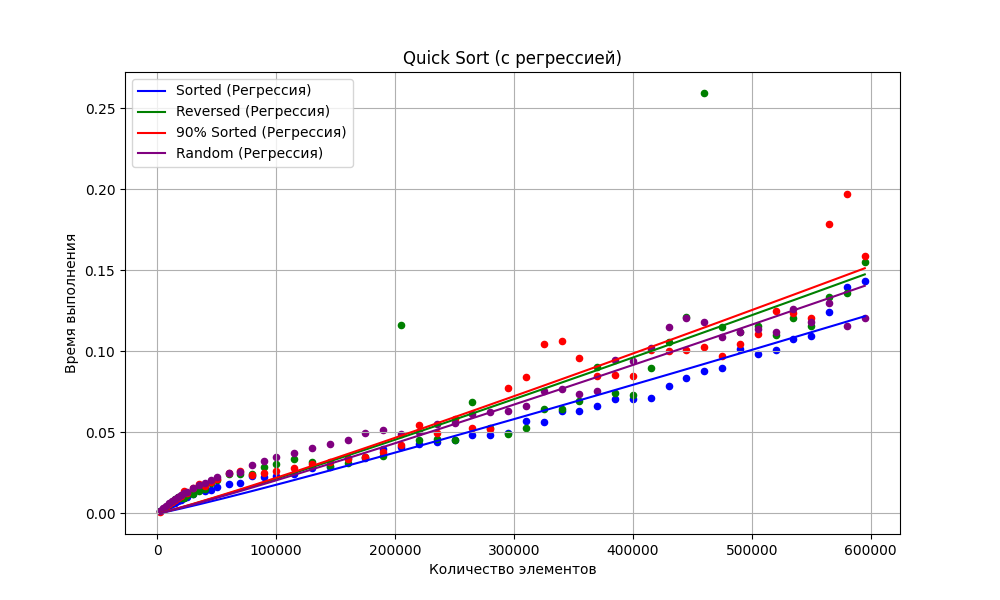


Рисунок 25 – График экспериментальных значений с регрессией для быстрой сортировки



Проведя экспериментальный анализ, мы можем сказать, что в целом ожидания соответствуют ожиданиям, пусть и с отклонениями. Данные отклонения могли появиться из-за того, что для быстрой сортировки очень тяжело сгенерировать лучший и худшие случаи, поэтому каждый из случаев обрабатывался как средний. При этом не исключено, что в сортировки, которые мы асимптотически изобразили с регрессией для среднего случая, являлись худшим или лучшим случаем. Вследствие этого, мы можем сказать, что наши предположения по асимптотике подтвердились.

# Пирамидальная сортировка

Таблица 15 – Экспериментально полученные значения для пирамидальной сортировки

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов | Время выполнения (в секундах) | | | |
| Sorted | Reversed | 90% sorted | Random |
| 2500 | 0.0002596 | 0.000249267 | 0.000306367 | 0.000367967 |
| 5000 | 0.0005606 | 0.000538567 | 0.000564667 | 0.0007226 |
| 7500 | 0.000860067 | 0.0009046 | 0.000932433 | 0.00104057 |
| 10000 | 0.0012027 | 0.00138617 | 0.00123683 | 0.0014634 |
| 12500 | 0.00152983 | 0.00143863 | 0.001586 | 0.00187333 |
| 15000 | 0.00177863 | 0.0018012 | 0.00186313 | 0.0025384 |
| 17500 | 0.00219287 | 0.00211653 | 0.00249897 | 0.00274713 |
| 20000 | 0.00248087 | 0.00246847 | 0.00275943 | 0.00374147 |
| 22500 | 0.00286877 | 0.0027315 | 0.00297617 | 0.00366507 |
| 25000 | 0.0031412 | 0.00308177 | 0.00338127 | 0.0042647 |
| 30000 | 0.00391907 | 0.0038327 | 0.0047611 | 0.0050565 |
| 35000 | 0.0046455 | 0.00440997 | 0.0047137 | 0.00611447 |
| 40000 | 0.00528137 | 0.00516443 | 0.0053981 | 0.0068636 |
| 45000 | 0.0061577 | 0.0059684 | 0.0061287 | 0.0080867 |
| 50000 | 0.00692867 | 0.00686277 | 0.0068133 | 0.00917067 |
| 60000 | 0.00822877 | 0.0080907 | 0.00832067 | 0.0108694 |
| 70000 | 0.00965153 | 0.00924967 | 0.00983407 | 0.012931 |
| 80000 | 0.011425 | 0.0114693 | 0.0112096 | 0.0148923 |
| 90000 | 0.0130457 | 0.0123698 | 0.0132877 | 0.0174402 |
| 100000 | 0.0145494 | 0.0137927 | 0.015261 | 0.019026 |
| 115000 | 0.0168341 | 0.0159505 | 0.0172708 | 0.022084 |
| 130000 | 0.0190406 | 0.0184791 | 0.018861 | 0.0258181 |
| 145000 | 0.021507 | 0.0211858 | 0.0213012 | 0.029728 |
| 160000 | 0.0239686 | 0.0242223 | 0.0240713 | 0.0317811 |
| 175000 | 0.0263667 | 0.025969 | 0.0263045 | 0.0355724 |
| 190000 | 0.0292712 | 0.028041 | 0.0296238 | 0.0388935 |
| 205000 | 0.0316473 | 0.0306751 | 0.0316082 | 0.0434803 |
| 220000 | 0.0334107 | 0.033266 | 0.0342047 | 0.0467289 |
| 235000 | 0.0359883 | 0.0367918 | 0.0356186 | 0.0485179 |
| 250000 | 0.0391967 | 0.0370166 | 0.0381046 | 0.0525739 |
| 265000 | 0.0405771 | 0.0402126 | 0.041161 | 0.05708 |
| 280000 | 0.0425245 | 0.0414352 | 0.0443925 | 0.0592343 |
| 295000 | 0.0459611 | 0.0454325 | 0.0459727 | 0.0650927 |

Рисунок 26 – График экспериментальных значений для пирамидальной сортировки

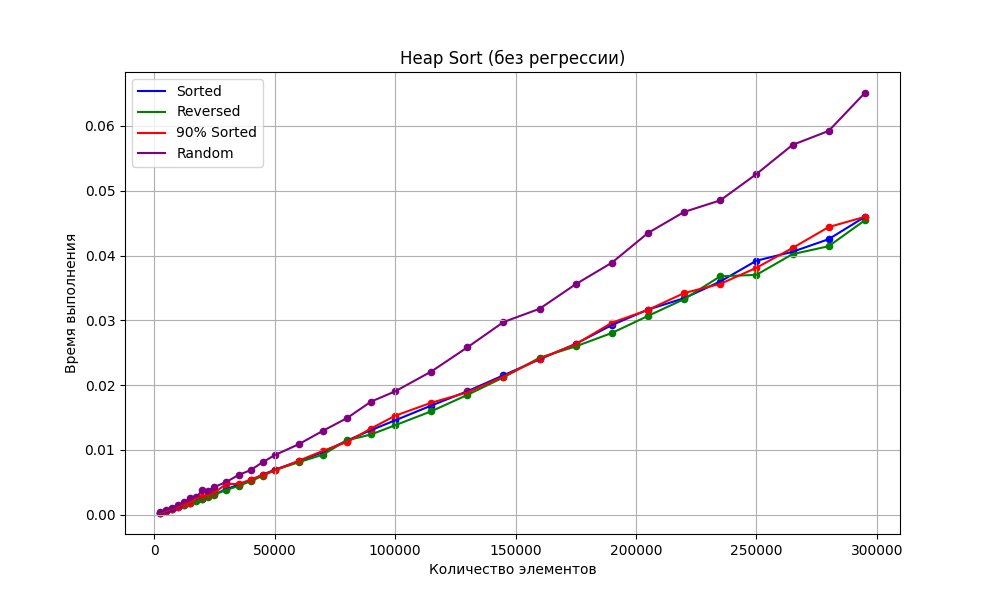
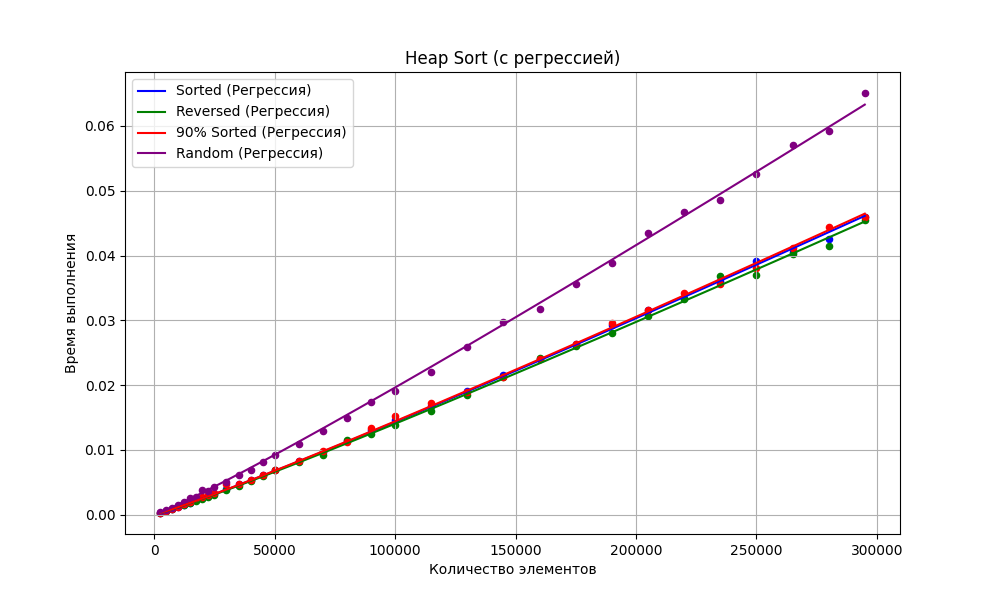


Рисунок 27 – График экспериментальных значений с регрессией для пирамидальной сортировки



Пирамидальная сортировка полностью соответствует нашим расчетам, что и видно на графике с использованием регрессии.

# Сортировка Шелла с последовательностью Шелла

Таблица 16 – Экспериментально полученные значения для сортировки Шелла с последовательностью Шелла

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов | Время выполнения (в секундах) | | | |
| Sorted | Reversed | 90% sorted | Random |
| 2500 | 9.76667e-06 | 0.0029449 | 1.70333e-05 | 0.00239107 |
| 5000 | 1.85333e-05 | 0.0118438 | 2.96e-05 | 0.0064209 |
| 7500 | 2.31e-05 | 0.0314345 | 3.37333e-05 | 0.0175708 |
| 10000 | 6.00333e-05 | 0.043101 | 0.000114 | 0.0419289 |
| 12500 | 4.38333e-05 | 0.0814892 | 6.93e-05 | 0.077091 |
| 15000 | 5.30667e-05 | 0.106809 | 0.000100733 | 0.104791 |
| 17500 | 7.24333e-05 | 0.136107 | 0.000118967 | 0.13654 |
| 20000 | 0.000227967 | 0.16434 | 0.000175467 | 0.164369 |
| 22500 | 8.88e-05 | 0.206678 | 0.0001976 | 0.206988 |
| 25000 | 8.28333e-05 | 0.320723 | 0.000171767 | 0.307099 |
| 30000 | 0.000105267 | 0.415848 | 0.000258067 | 0.240517 |
| 35000 | 0.000142967 | 0.543102 | 0.0003422 | 0.334159 |
| 40000 | 0.000154233 | 0.656186 | 0.000516633 | 0.653266 |
| 45000 | 0.000183233 | 0.825732 | 0.000527767 | 0.814704 |
| 50000 | 0.000177967 | 1.28759 | 0.000573667 | 1.00317 |
| 60000 | 0.000217267 | 1.6754 | 0.000824967 | 1.24876 |
| 70000 | 0.000310333 | 2.23065 | 0.000882233 | 1.94958 |
| 80000 | 0.0003399 | 3.00072 | 0.00105227 | 2.40066 |
| 90000 | 0.000386933 | 4.93393 | 0.0013678 | 3.10757 |
| 100000 | 0.0003707 | 7.301 | 0.00174643 | 4.38085 |
| 110000 | 0.0004075 | 9.24897 | 0.00218223 | 3.55423 |
| 120000 | 0.0004617 | 9.538 | 0.00177793 | 6.05091 |
| 130000 | 0.000477633 | 11.108 | 0.00242117 | 6.87155 |
| 140000 | 0.000712133 | 11.5881 | 0.00228597 | 6.58058 |
| 150000 | 0.000686867 | 13.937 | 0.00323503 | 6.75355 |

Рисунок 28 – График экспериментальных значений для сортировки Шелла с последовательностью Шелла

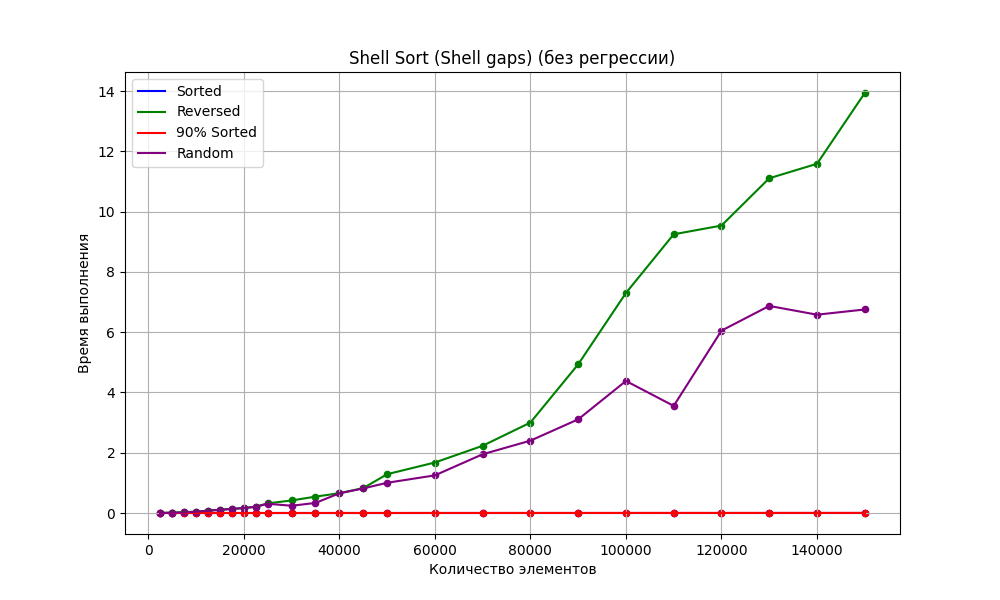
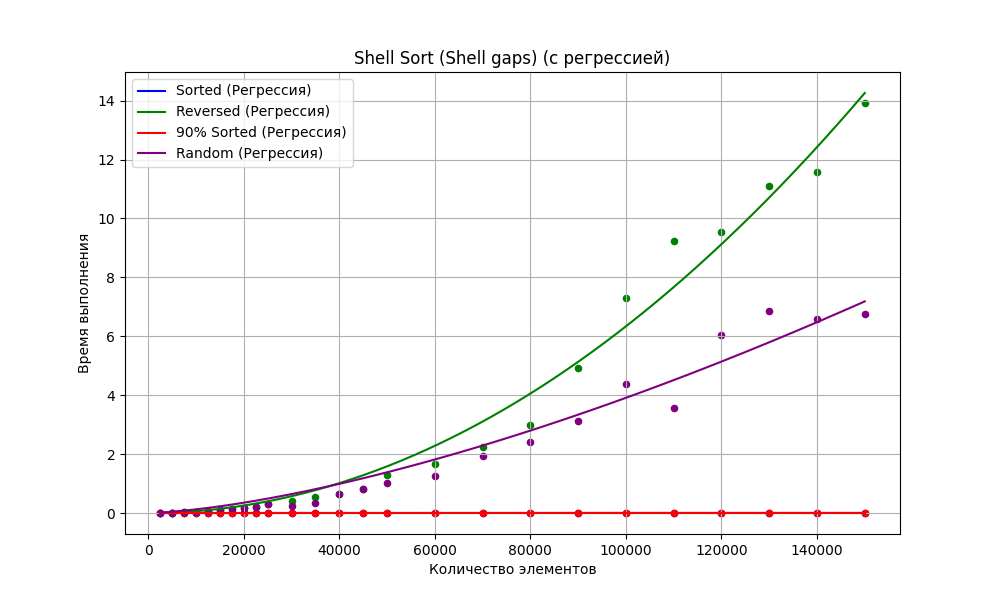


Рисунок 29 – График экспериментальных значений с регрессией для сортировки Шелла с последовательностью Шелла



Сортировка Шелла с последовательностью Шелла показывает хорошие результат, хоть и видно, что имеются некоторые отклонения от нормы, но в целом на графике с регрессией мы видим, что график практически идеально сходится, так что мы можем пренебречь данными отклонениями.

# Сортировка Шелла с последовательностью Пратта

Таблица 17 – Экспериментально полученные значения для сортировки Шелла с последовательностью Пратта

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов | Время выполнения (в секундах) | | | |
| Sorted | Reversed | 90% sorted | Random |
| 2500 | 1.4e-05 | 0.00312827 | 2.12333e-05 | 0.00264563 |
| 5000 | 2.69667e-05 | 0.0157846 | 4.18333e-05 | 0.0098202 |
| 7500 | 4.19333e-05 | 0.0343633 | 6.28333e-05 | 0.0220443 |
| 10000 | 5.95667e-05 | 0.0614112 | 0.000102567 | 0.0369155 |
| 12500 | 7.05667e-05 | 0.0953913 | 0.0001559 | 0.0582727 |
| 15000 | 0.0005935 | 0.08384 | 0.000144833 | 0.0845258 |
| 17500 | 0.000154533 | 0.114574 | 0.000195333 | 0.112797 |
| 20000 | 0.0001734 | 0.145217 | 0.000258167 | 0.149063 |
| 22500 | 0.000196767 | 0.180434 | 0.0001798 | 0.200214 |
| 25000 | 0.000222167 | 0.223952 | 0.000289067 | 0.199065 |
| 30000 | 0.000264333 | 0.321295 | 0.000335167 | 0.197005 |
| 35000 | 0.000304767 | 0.533742 | 0.0004173 | 0.278733 |
| 40000 | 0.000350033 | 0.962976 | 0.0004402 | 0.475287 |
| 45000 | 0.000394267 | 1.21966 | 0.0004842 | 0.758406 |
| 50000 | 0.000444967 | 1.50747 | 0.000532867 | 0.921031 |
| 60000 | 0.000546 | 1.86128 | 0.0007667 | 1.09637 |
| 70000 | 0.0006117 | 2.87489 | 0.00122903 | 1.65729 |
| 80000 | 0.000521133 | 3.38793 | 0.00117547 | 2.35436 |
| 90000 | 0.000667633 | 4.47101 | 0.00135147 | 2.55547 |
| 100000 | 0.000745033 | 5.916 | 0.00141257 | 2.86879 |
| 110000 | 0.00122973 | 7.20319 | 0.001582 | 3.35629 |
| 120000 | 0.001858 | 7.53013 | 0.00396373 | 4.58007 |
| 130000 | 0.00113647 | 9.03095 | 0.00293667 | 5.65225 |
| 140000 | 0.0013095 | 9.44055 | 0.00412413 | 6.42173 |
| 150000 | 0.00118893 | 11.109 | 0.00395747 | 7.34628 |

Рисунок 30 – График экспериментальных значений для сортировки Шелла с последовательностью Пратта

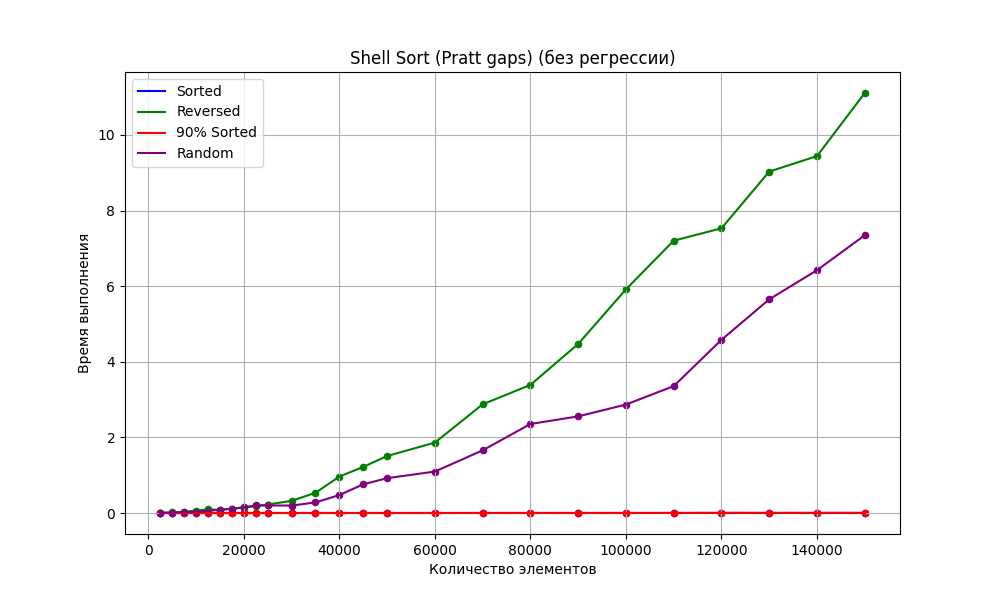
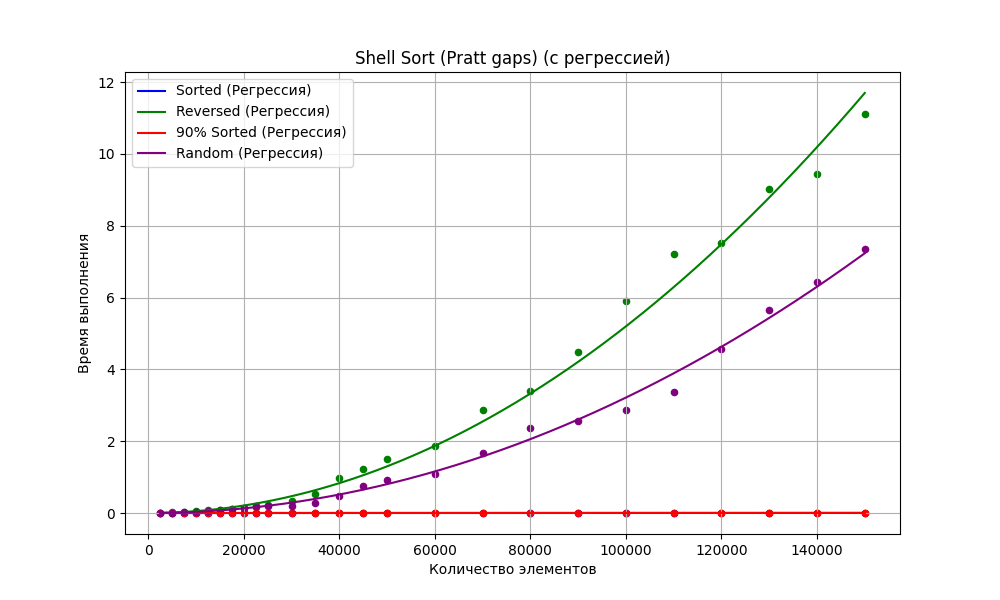


Рисунок 31 – График экспериментальных значений с регрессией для сортировки Шелла с последовательностью Пратта



Сортировка Шелла с последовательностью Пратта показывает чуть более стабильные экспериментальные результаты, что и видно на графике. На графике с использованием регрессии видно, что наши теоретические предположения сходятся с практическими.

# Сортировка Шелла с последовательностью Хиббарда

Таблица 18 – Экспериментально полученные значения для сортировки Шелла с последовательностью Хиббарда

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов | Время выполнения (в секундах) | | | |
| Sorted | Reversed | 90% sorted | Random |
| 2500 | 1.14667e-05 | 0.00495703 | 7.9e-06 | 0.00345463 |
| 5000 | 2.25e-05 | 0.0194866 | 2.02667e-05 | 0.014972 |
| 7500 | 3.58333e-05 | 0.0430764 | 3.37333e-05 | 0.0260555 |
| 10000 | 4.52333e-05 | 0.0780166 | 5.18e-05 | 0.0443592 |
| 12500 | 5.66667e-05 | 0.122678 | 7.23e-05 | 0.0554535 |
| 15000 | 7.03e-05 | 0.160882 | 8.79e-05 | 0.0994931 |
| 17500 | 8.17e-05 | 0.20647 | 0.0001239 | 0.134185 |
| 20000 | 9.32e-05 | 0.191915 | 0.000142867 | 0.175642 |
| 22500 | 0.000105333 | 0.390394 | 0.000171233 | 0.222568 |
| 25000 | 0.000113033 | 0.481189 | 0.000182633 | 0.235139 |
| 30000 | 0.0001419 | 0.695147 | 0.0011002 | 0.396235 |
| 35000 | 0.000161967 | 0.848416 | 0.000324733 | 0.48216 |
| 40000 | 0.000185533 | 1.23381 | 0.0006432 | 0.589618 |
| 45000 | 0.000211833 | 1.53258 | 0.000511533 | 0.794368 |
| 50000 | 0.0002317 | 1.52148 | 0.000537067 | 0.978246 |
| 60000 | 0.000226333 | 2.09823 | 0.0008112 | 1.05187 |
| 70000 | 0.000281533 | 2.79849 | 0.00100047 | 1.59277 |
| 80000 | 0.000520233 | 3.58888 | 0.00121857 | 3.07724 |
| 90000 | 0.000420033 | 4.79983 | 0.001368 | 3.10898 |
| 100000 | 0.000477067 | 5.9041 | 0.0017208 | 4.11535 |
| 110000 | 0.0005143 | 6.60889 | 0.00172847 | 4.7048 |
| 120000 | 0.000593933 | 8.72021 | 0.00218173 | 5.38415 |
| 130000 | 0.000707233 | 11.7224 | 0.00288343 | 6.46233 |
| 140000 | 0.000756267 | 11.796 | 0.0023281 | 8.09812 |
| 150000 | 0.000835667 | 12.2643 | 0.0032553 | 9.72955 |

Рисунок 32 – График экспериментальных значений для сортировки Шелла с последовательностью Хиббарда

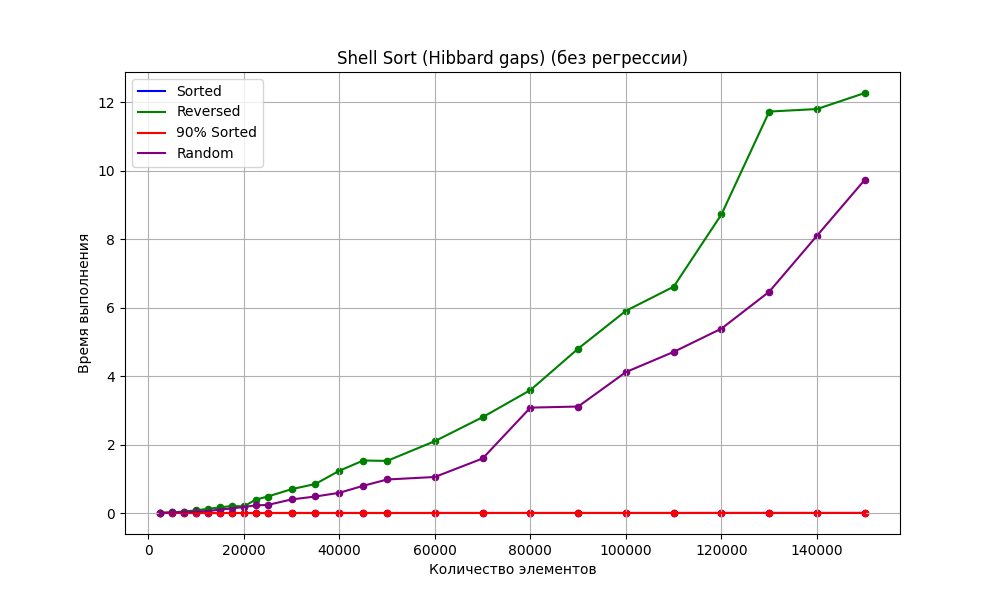
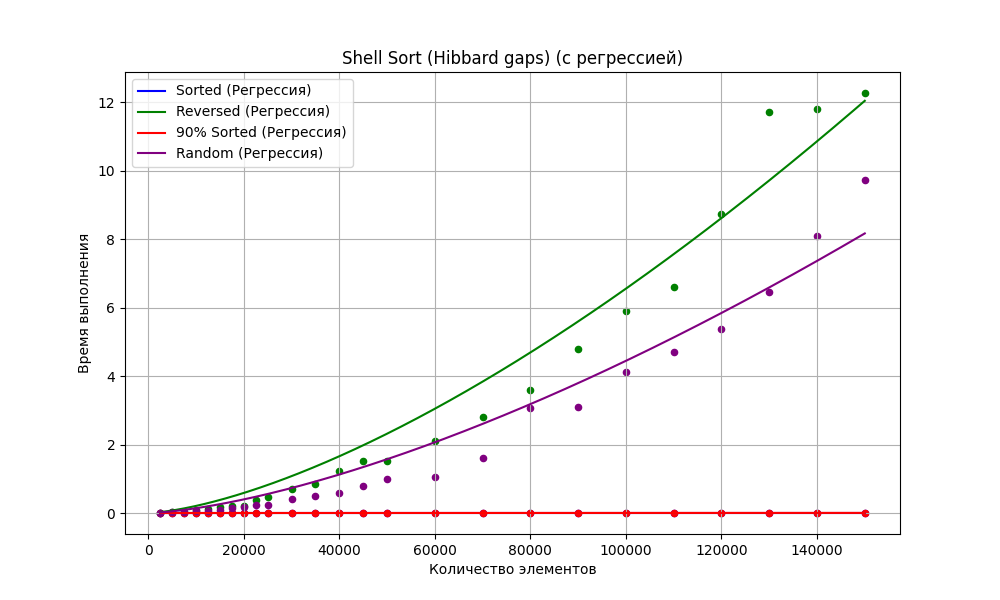


Рисунок 33 – График экспериментальных значений с регрессией для сортировки Шелла с последовательностью Хиббарда



Сортировка Шелла с последовательностью Хиббарда не так хорошо показала себя на практике, как сортировка Шелла с последовательностью Пратта, но в целом отклонения находятся в допустимых значениях, поскольку данные отклонения могли быть вызваны какими-то дополнительными нагрузками на процессор во время проведения тестов. Если сравнивать теоретическую оценку с практической, мы можем сказать, что теория сходится с практикой.

# Код программы

main.cpp – файл с функциями для генерации массивов и проведения тестов.

#include <iostream>

#include <locale.h>

#include <ctime>

#include <chrono>

#include <fstream>

#include "sorts.h"

using namespace std;

const int N = 100;

bool is\_sorted(int\* arr, int size) {

for (int i = 0; i < size - 1; i++) {

if (arr[i] > arr[i + 1]) return false;

}

return true;

}

int\* generate\_array(int size, int min\_value, int max\_value) {

int\* array = new int[size];

for (int i = 0; i < size; ++i) {

array[i] = rand() % (max\_value - min\_value + 1) + min\_value;

}

return array;

}

void test\_for\_lab(ofstream& ofs, ofstream& ofs1, int check\_up=0) {

std::cout << "\n\tНачало тестов для лабы.\n\n";

int test\_now = 0;

int test\_counts = 3;

int gap\_size = 0;

int\* gap = nullptr;

int check\_gap = 1;

int num\_of\_sort = 1;

int size\_step = 15000;

int size\_max = 300000;

if (!check\_up) {

std::cout << "\n\n\tВведи через клавишу Enter:\n1 - количество тестов\n2 - максимальное количество элементов\n3 - шаг количества элементов\n- ";

std::cin >> test\_counts;

std::cout << "- ";

std::cin >> size\_max;

std::cout << "- ";

std::cin >> size\_step;

std::cout << "\n\n";

}

int start\_size\_max = size\_max;

int start\_size\_step = size\_step;

while (num\_of\_sort != 8) {

if (check\_up && (num\_of\_sort == 2 || num\_of\_sort == 3 || num\_of\_sort == 4 || num\_of\_sort == 1)) {

size\_max = 150000;

size\_step = 10000;

}

else if (check\_up) {

size\_max = start\_size\_max;

size\_step = start\_size\_step;

}

cout << num\_of\_sort << "\n";

if (!check\_up) {

std::cout << "\tВведи номер сортировки, которую хочешь использовать:\n";

std::cout << "1 - ShellSort\n2 - InsertionSort\n3 - SelectionSort\n4 - BubbleSort\n";

std::cout << "5 - HeapSort\n6 - MergeSort\n7 - QuickSort\n\n";

std::cout << "Ввод - ";

std::cin >> num\_of\_sort;

}

if (num\_of\_sort == 1) {

ofs << "\tНачало тестов для ShellSort";

if (!check\_up) {

ofs << "\n\n";

std::cout << "\n\nВы выбрали ShellSort, поэтому стоит выбрать определенный шаг:\n1 - ShellGap\n2 - PrattGap\n3 - HibbardGap\n4 - MyGap(test)\nВвод - ";

std::cin >> check\_gap;

}

else {

if (check\_gap == 1) ofs << " - Shell Gaps\n\n";

else if (check\_gap == 2) ofs << " - Pratt Gaps\n\n";

else if (check\_gap == 3) ofs << " - Hibbard Gaps\n\n";

else {

ofs << " - My Gaps(test)\n\n";

}

}

}

else if (num\_of\_sort == 2) ofs << "\tНачало тестов для InsertionSort\n\n";

else if (num\_of\_sort == 3) ofs << "\tНачало тестов для SelectionSort\n\n";

else if (num\_of\_sort == 4) ofs << "\tНачало тестов для BubbleSort\n\n";

else if (num\_of\_sort == 5) ofs << "\tНачало тестов для HeapSort\n\n";

else if (num\_of\_sort == 6) ofs << "\tНачало тестов для MergeSort\n\n";

else ofs << "\tНачало тестов для QuickSort\n\n";

sorts sorter;

gaps gap\_generator;

for (int k = 0; k < 4; k++) {

int i = 0;

for (check\_up ? i = 0 : i = 0; i <= size\_max;) {

if (check\_up) {

if (i < 25000) i += 2500;

else if (i < 50000) i += 5000;

else if (i < 100000) i += 10000;

else i += size\_step;

}

else i += size\_step;

//i += size\_step;

if (i > size\_max) break;

chrono::duration<double>\* average\_time = new chrono::duration<double>[test\_counts];

chrono::duration<double> a\_time = chrono::duration<double>();

for (int test\_no = 0; test\_no < test\_counts; ++test\_no) {

int\* arr = new int[1];

if (k == 0) {

arr = generate\_array(i, -10000, 10000);

arr = sorter.quick\_sort(arr, i);

}

else if (k == 1) {

arr = generate\_array(i, -10000, 10000);

arr = sorter.reverse\_quick\_sort(arr, i);

}

else if (k == 2) {

int st = 0;

int\* new\_arr = new int[i - (i / size\_step)];

new\_arr = sorter.quick\_sort(generate\_array(i - (i / size\_step), -10000, 10000), i - (i / size\_step));

int\* random\_elements = new int[i / size\_step];

random\_elements = generate\_array(i / size\_step, -10000, 10000);

arr = new int[i];

for (int z = 0; z < i - (i / size\_step); z++) {

arr[z] = new\_arr[z];

}

for (int z = i - (i / size\_step); z < i; z++) {

arr[z] = random\_elements[st++];

}

}

else {

arr = generate\_array(i, -10000, 10000);

}

if (arr == nullptr) {

std::cerr << "Ошибка выделения памяти для массива!\n";

return;

}

if (check\_gap) {

switch (check\_gap) {

case 1:

gap = gap\_generator.shell\_gaps(i, gap\_size);

break;

case 2:

gap = gap\_generator.prat\_gaps(i, gap\_size);

break;

case 3:

gap = gap\_generator.hib\_gaps(i, gap\_size);

break;

case 4:

gap = gap\_generator.my\_simple\_gaps(i, gap\_size);

break;

default:

std::cerr << "Неверный выбор промежутка!\n";

delete[] arr;

return;

}

if (gap == nullptr) {

std::cerr << "Ошибка выделения памяти для промежутков!\n";

delete[] arr;

return;

}

}

auto start = chrono::high\_resolution\_clock::now();

switch (num\_of\_sort) {

case 1: arr = sorter.shell\_sort(arr, gap, i, gap\_size); break;

case 2: arr = sorter.insertion\_sort(arr, i); break;

case 3: arr = sorter.selection\_sort(arr, i); break;

case 4: arr = sorter.bubble\_sort(arr, i); break;

case 5: arr = sorter.heap\_sort(arr, i); break;

case 6: arr = sorter.merge\_sort(arr, i); break;

case 7: arr = sorter.quick\_sort(arr, i); break;

default: std::cerr << "Неверный номер сортировки!" << std::endl;

delete[] arr;

return;

}

auto end = chrono::high\_resolution\_clock::now();

chrono::duration<double> duration = end - start;

average\_time[test\_no] = duration;

delete[] arr;

if (check\_gap) delete[] gap;

}

for (int j = 0; j < test\_counts; j++) {

a\_time += average\_time[j];

}

a\_time /= test\_counts;

if (k == 0) {

ofs << "Тест #" << i / size\_step << " для полностью ОТСОРТИРОВАННОГО массива размера " << i

<< ": " << "УСПЕШНО"

<< " - Среднее время: " << a\_time.count() << " секунд" << std::endl;

cout << "Тест #" << i / size\_step << " для полностью ОТСОРТИРОВАННОГО массива размера " << i

<< ": " << "УСПЕШНО"

<< " - Среднее время: " << a\_time.count() << " секунд" << std::endl;

}

else if (k == 1) {

ofs << "Тест #" << i / size\_step << " для ОБРАТНО отсортированного массива размера " << i

<< ": " << "УСПЕШНО"

<< " - Среднее время: " << a\_time.count() << " секунд" << std::endl;

cout << "Тест #" << i / size\_step << " для ОБРАТНО отсортированного массива размера " << i

<< ": " << "УСПЕШНО"

<< " - Среднее время: " << a\_time.count() << " секунд" << std::endl;

}

else if (k == 2) {

ofs << "Тест #" << i / size\_step << " для ЧАСТИЧНО отсортированного массива размера " << i

<< ": " << "УСПЕШНО"

<< " - Среднее время: " << a\_time.count() << " секунд" << std::endl;

cout << "Тест #" << i / size\_step << " для ЧАСТИЧНО отсортированного массива размера " << i

<< ": " << "УСПЕШНО"

<< " - Среднее время: " << a\_time.count() << " секунд" << std::endl;

}

else {

ofs << "Тест #" << i / size\_step << " для СЛУЧАЙНОГО массива размера " << i

<< ": " << "УСПЕШНО"

<< " - Среднее время: " << a\_time.count() << " секунд" << std::endl;

cout << "Тест #" << i / size\_step << " для СЛУЧАЙНОГО массива размера " << i

<< ": " << "УСПЕШНО"

<< " - Среднее время: " << a\_time.count() << " секунд" << std::endl;

}

ofs1 << i << " " << a\_time.count() << endl;

delete[] average\_time;

}

ofs << endl;

ofs1 << endl;

cout << endl;

}

if (!check\_up) break;

if (num\_of\_sort == 1 && check\_gap < 3) {

num\_of\_sort--;

check\_gap++;

}

else {

check\_gap = 0;

}

num\_of\_sort++;

}

std::cout << "Конец тестов для лабы.\n";

}

int main() {

setlocale(LC\_ALL, "Ru");

srand(time(0));

ofstream ofs("result.txt");

ofstream ofs1("graph.txt");

test\_for\_lab(ofs, ofs1, 1);

return 0;

}

sorts.h – файл с объявлением класса для сортировок

#pragma once

#include <iostream>

#include <cmath>

#include <random>

class sorts

{

public:

int\* shell\_sort(int\* arr, int\* gaps, int size, int gaps\_size);

int\* quick\_sort(int\* arr, int size);

int\* merge\_sort(int\* arr, int size);

int\* insertion\_sort(int\* arr, int size);

int\* selection\_sort(int\* arr, int size);

int\* bubble\_sort(int\* arr, int size);

int\* heap\_sort(int\* arr, int size);

int\* reverse\_quick\_sort(int\* arr, int size);

private:

void heapify(int\* arr, int size, int root\_idx);

};

struct gaps {

public:

bool is\_prime(int num);

int\* my\_simple\_gaps(int n, int& gaps\_size);

int\* shell\_gaps(int n, int& gaps\_size);

int\* hib\_gaps(int n, int& gaps\_size);

int\* prat\_gaps(int n, int& gaps\_size);

};

void swap(int& n1, int& n2);

sorts.cpp – файл с функциями для сортировок

#include "sorts.h"

#include <list>

void swap(int& n1, int& n2)

{

int tmp = n1;

n1 = n2;

n2 = tmp;

}

int\* sorts::shell\_sort(int\* arr, int\* gaps, int size, int gaps\_size) {

if (size <= 1) {

return arr;

}

else {

for (int g = 0; g < gaps\_size; g++) {

int gap = gaps[g];

for (int i = gap; i < size; i += gap) {

int key = arr[i];

int j = i;

while (j >= gap && arr[j - gap] > key) {

arr[j] = arr[j - gap];

j -= gap;

}

arr[j] = key;

}

}

}

return arr;

}

int\* sorts::quick\_sort(int\* arr, int size)

{

if (size <= 1) return arr;

int pivot = arr[rand() % size];

int l\_i = 0, m\_i = 0, r\_i = 0;

int\* l\_arr = new int[size];

int\* m\_arr = new int[size];

int\* r\_arr = new int[size];

for (int i = 0; i < size; i++) {

if (arr[i] < pivot) {

l\_arr[l\_i++] = arr[i];

}

else if (arr[i] > pivot) {

r\_arr[r\_i++] = arr[i];

}

else {

m\_arr[m\_i++] = arr[i];

}

}

int\* sorted\_left = quick\_sort(l\_arr, l\_i);

int\* sorted\_right = quick\_sort(r\_arr, r\_i);

int\* result = new int[size];

std::copy(sorted\_left, sorted\_left + l\_i, result);

std::copy(m\_arr, m\_arr + m\_i, result + l\_i);

std::copy(sorted\_right, sorted\_right + r\_i, result + l\_i + m\_i);

delete[] l\_arr;

delete[] m\_arr;

delete[] r\_arr;

return result;

}

int\* sorts::merge\_sort(int\* arr, int size)

{

if (size <= 1) return arr;

int mid = size / 2;

int r\_size = size - mid;

int\* l\_fake = new int[mid];

int\* r\_fake = new int[r\_size];

for (int i = 0; i < mid; i++) {

l\_fake[i] = arr[i];

}

for (int i = 0; i < r\_size; i++) {

r\_fake[i] = arr[mid + i];

}

merge\_sort(l\_fake, mid);

merge\_sort(r\_fake, r\_size);

int i = 0, j = 0, k = 0;

int\* new\_arr = new int[size];

while (i < mid && j < r\_size) {

if (l\_fake[i] < r\_fake[j]) new\_arr[k++] = l\_fake[i++];

else new\_arr[k++] = r\_fake[j++];

}

while (i < mid) new\_arr[k++] = l\_fake[i++];

while (j < r\_size) new\_arr[k++] = r\_fake[j++];

for (int i1 = 0; i1 < size; i1++) {

arr[i1] = new\_arr[i1];

}

if (l\_fake) delete[] l\_fake;

if (r\_fake) delete[] r\_fake;

if (new\_arr) delete[] new\_arr;

return arr;

}

int\* sorts::insertion\_sort(int\* arr, int size)

{

if (size <= 1) return arr;

for (int i = 1; i < size; i++) {

int elem = arr[i];

int j = i;

while (j >= 1 && arr[j - 1] > elem) {

arr[j] = arr[j - 1];

j--;

}

arr[j] = elem;

}

return arr;

}

int\* sorts::selection\_sort(int\* arr, int size)

{

if (size <= 1) return arr;

for (int i = 0; i < size - 1; i++) {

int min\_idx = i;

for (int j = i + 1; j < size; j++) {

if (arr[j] < arr[min\_idx]) {

min\_idx = j;

}

}

swap(arr[i], arr[min\_idx]);

}

return arr;

}

int\* sorts::bubble\_sort(int\* arr, int size)

{

if (size <= 1) return arr;

for (int i = 0; i < size; i++) {

bool flag = 1;

for (int j = 0; j < size - (i + 1); j++) {

if (arr[j] > arr[j + 1]) {

flag = 0;

swap(arr[j], arr[j + 1]);

}

}

if (flag) return arr;

}

return arr;

}

int\* sorts::heap\_sort(int\* arr, int size)

{

if (size <= 1) return arr;

for (int i = size - 1; i >= 0; i--) {

heapify(arr, size, i);

}

for (int i = size - 1; i > 0; i--) {

swap(arr[i], arr[0]);

heapify(arr, i, 0);

}

return arr;

}

void sorts::heapify(int\* arr, int size, int root\_idx)

{

int largest = root\_idx;

int l\_kid = (2 \* root\_idx) + 1;

int r\_kid = (2 \* root\_idx) + 2;

if (l\_kid < size && arr[l\_kid] > arr[largest]) largest = l\_kid;

if (r\_kid < size && arr[r\_kid] > arr[largest]) largest = r\_kid;

if (largest != root\_idx) {

swap(arr[root\_idx], arr[largest]);

heapify(arr, size, largest);

}

}

int\* sorts::reverse\_quick\_sort(int\* arr, int size) {

if (size <= 1) return arr;

int pivot = arr[rand() % size];

int l\_i = 0, m\_i = 0, r\_i = 0;

int\* l\_arr = new int[size];

int\* m\_arr = new int[size];

int\* r\_arr = new int[size];

for (int i = 0; i < size; i++) {

if (arr[i] < pivot) {

l\_arr[l\_i++] = arr[i];

}

else if (arr[i] > pivot) {

r\_arr[r\_i++] = arr[i];

}

else {

m\_arr[m\_i++] = arr[i];

}

}

int\* sorted\_left = reverse\_quick\_sort(l\_arr, l\_i);

int\* sorted\_right = reverse\_quick\_sort(r\_arr, r\_i);

int\* result = new int[size];

std::copy(sorted\_right, sorted\_right + r\_i, result);

std::copy(m\_arr, m\_arr + m\_i, result + r\_i);

std::copy(sorted\_left, sorted\_left + l\_i, result + r\_i + m\_i);

delete[] l\_arr;

delete[] m\_arr;

delete[] r\_arr;

return result;

}

bool gaps::is\_prime(int num)

{

if (num < 2) {

return false;

}

for (int x = 2; x <= static\_cast<int>(sqrt(num)) + 1; ++x) {

if (num % x == 0) {

return false;

}

}

return true;

}

int\* gaps::my\_simple\_gaps(int n, int& gaps\_size)

{

int\* temp = new int[n + 1];

int count = 2;

int size = 1;

temp[0] = 1;

while (count <= n) {

if (is\_prime(count)) {

temp[size] = count;

size++;

count \*= static\_cast<int>(1.5 \* exp((count - 1) / static\_cast<double>(count)));

}

else {

++count;

}

}

int\* gaps\_1 = new int[size];

for (int i = 0; i < size; i++) {

gaps\_1[i] = temp[i];

}

for (int i = 0; i < size / 2; ++i) {

swap(gaps\_1[i], gaps\_1[size - i - 1]);

}

gaps\_size = size;

return gaps\_1;

}

int\* gaps::shell\_gaps(int n, int& gaps\_size)

{

int\* gaps = new int[log2(n)];

gaps\_size = 0;

for (int i = n / 2; i > 1; i /= 2) {

gaps[gaps\_size] = i;

gaps\_size++;

}

int\* real\_gaps = new int[++gaps\_size];

for (int i = 0; i < gaps\_size - 1; i++) {

real\_gaps[i] = gaps[i];

}

real\_gaps[gaps\_size - 1] = 1;

if (gaps) delete[] gaps;

return real\_gaps;

}

int\* gaps::hib\_gaps(int n, int& gaps\_size)

{

int max\_size = log2(n) + 1;

int\* gaps = new int[max\_size];

int res = 0;

gaps\_size = 0;

for (int i = 0; res <= n; i++) {

res = pow(2, i + 1) - 1;

if (res > n) break;

gaps[gaps\_size++] = res;

}

int\* real\_gaps = new int[gaps\_size];

for (int i = 0; i < gaps\_size; i++) {

real\_gaps[i] = gaps[gaps\_size - 1 - i];

}

delete[] gaps;

return real\_gaps;

}

int\* gaps::prat\_gaps(int n, int& gaps\_size)

{

std::list<int> unique\_sequence;

for (int i = 0; (1 << i) <= n; ++i) {

for (int j = 0; (1 << i) \* static\_cast<int>(pow(3, j)) <= n; ++j) {

int value = (1 << i) \* static\_cast<int>(pow(3, j));

if (value <= n) {

if (std::find(unique\_sequence.begin(), unique\_sequence.end(), value) == unique\_sequence.end()) {

unique\_sequence.push\_back(value);

}

}

}

}

std::vector<int> sorted\_vector(unique\_sequence.begin(), unique\_sequence.end());

std::sort(sorted\_vector.begin(), sorted\_vector.end(), std::greater<int>());

gaps\_size = sorted\_vector.size();

int\* result\_array = new int[gaps\_size];

std::copy(sorted\_vector.begin(), sorted\_vector.end(), result\_array);

return result\_array;

}

# Заключение

В ходе выполнения лабораторной работы, мы изучили множество различных алгоритмов сортировок сравнением, а именно были рассчитаны их временная и пространственная сложности, выведена функция временной сложности и построены графики по этой функции. Помимо всего прочего, было проведено практическое тестирование каждой из сортировок, что дало нам понять, что все практические значения сошлись с рассчитанными в теоретической части, что можно увидеть на графиках, которые были построенные с использованием регрессии, а также без нее.

Помимо прочего, наше предположение о том, что быстрейшая сортировка на массивах от 100000 элементов – быстрая сортировка, оправдалась, что так же было проверено экспериментальным путем.

# Ссылка на репозиторий GitHub

//тут ссылка