МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«Челябинский государственный университет»**

**(ФГБОУ ВО «ЧелГУ»)**

Институт информационных технологий

ОТЧЕТ

по лабораторной работе №1

Авторы отчета М.Е. Садунов ПрИ-201

подпись инициалы, фамилия группа

Н.С. Мельников ПрИ-201

подпись инициалы, фамилия группа

А.О. Марциохо ПрИ-201

подпись инициалы, фамилия группа

М.С. Захаров ПрИ-201

подпись инициалы, фамилия группа

Отчет защищен \_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_

дата оценка

Челябинск 2024 г.

**Цель работы:** Эмпирический анализ временной сложности алгоритмов.

**Задание.**

I. Для каждого n от 1 до 2000 произведите для пяти запусков замер среднего машинного времени исполнения программ, реализующих нижеуказанные алгоритмы и функции. Изобразите на графике полученные данные, отражающие зависимость среднего времени исполнения от n. Проведите теоретический анализ временной сложности рассматриваемых алгоритмов и сравните эмпирическую и теоретическую временные сложности. I. Сгенерируйте n-мерный случайный вектор v = [v1, v2, . . . , vn] с неотрицательными элементами. Для полученного вектора v осуществите подсчет функций и реализацию алгоритмов:

1. (постоянная функция);
2. (сумма элементов);
3. (произведение элементов)
4. полагая, что элементы – коэффициенты многочлена степени , вычислите значение путем прямого (наивного) вычисления для (т.е. оценивая каждый член по одному) и методом Горнера представление полинома: ;
5. алгоритм сортировки пузырьком (Bubble sort) элементов ;
6. алгоритм быстрой сортировки (Quick sort) элементов ;
7. гибридный алгоритм сортировки Timsort элементов ;
8. Алгоритмы возведения в степень

II. Сгенерируйте случайные матрицы и размером с неотрицательными элементами. Найдите обычное матричное произведение матриц и .

III. На каждого члена команды найти алгоритм не ниже линейного класса сложности и провести с ним эксперимент.

Стек технологий:   
1. C#  
2. .NET Multi-Platform App UI

Подробнее о каждом стеке:  
1. C#-- объектно-ориентированный язык общего назначения.

2. .NET Multi-Platform App UI (.NET MAUI) — это кросс-платформенная платформа для создания собственных мобильных и классических приложений с помощью C# и XAML.

Рис 0 Общая структура проекта

**Задание I.**

На вход подавались массивы размеров от 1 до N элементов. Замер времени происходил для каждого массива. Подсчеты проводились с помощью параллельных вычислений.

1. F(v) = 1

* Временная сложность: O(1)
* N = 50000
* Ср. знач. на основе тестов: 100

Присвоение новому элементу, не из массива, длины массива умноженной на 2.

Замеры времени от количества элементов(смотри рис 1.1.1):

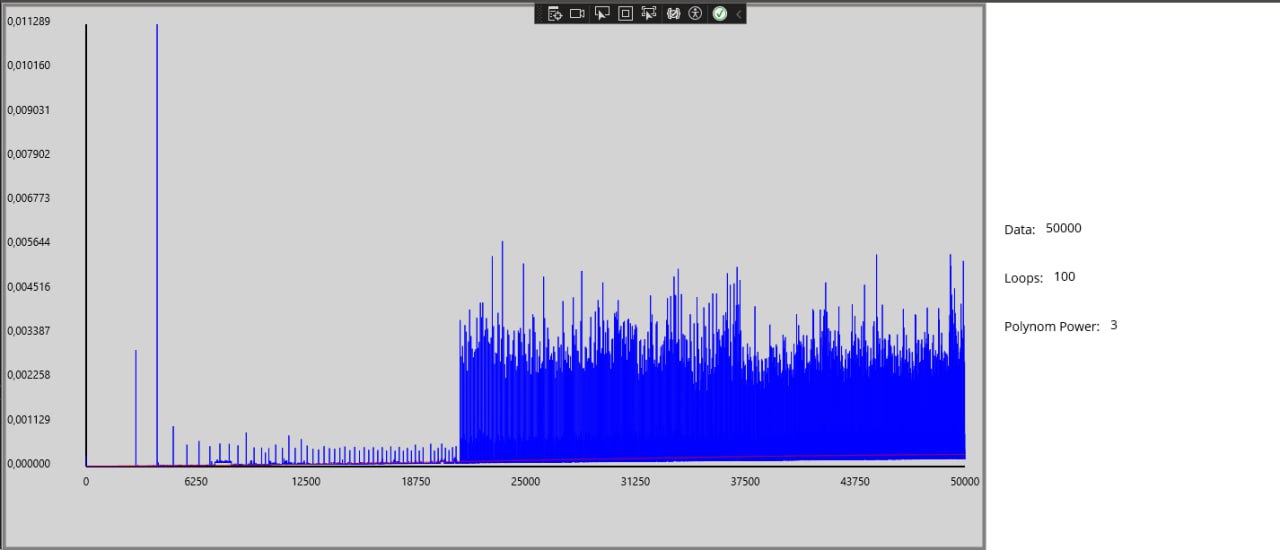


Рис. 1.1.1 Const алгоритм

Код изображен на Рис. 1.1.2:



Рис. 1.1.2 Код алгоритма

1. F(v) = (сумма элементов)

* Временная сложность: O(n)
* N=50000
* Ср. знач. на основе тестов: 100

Алгоритм суммирует все i-тые элементы массива.

По графику (см. Рис. 1.2.1) данного алгоритма мы видим, что у нас

получилась линейная зависимость времени от количества входных данных,

необходимых для обработки:

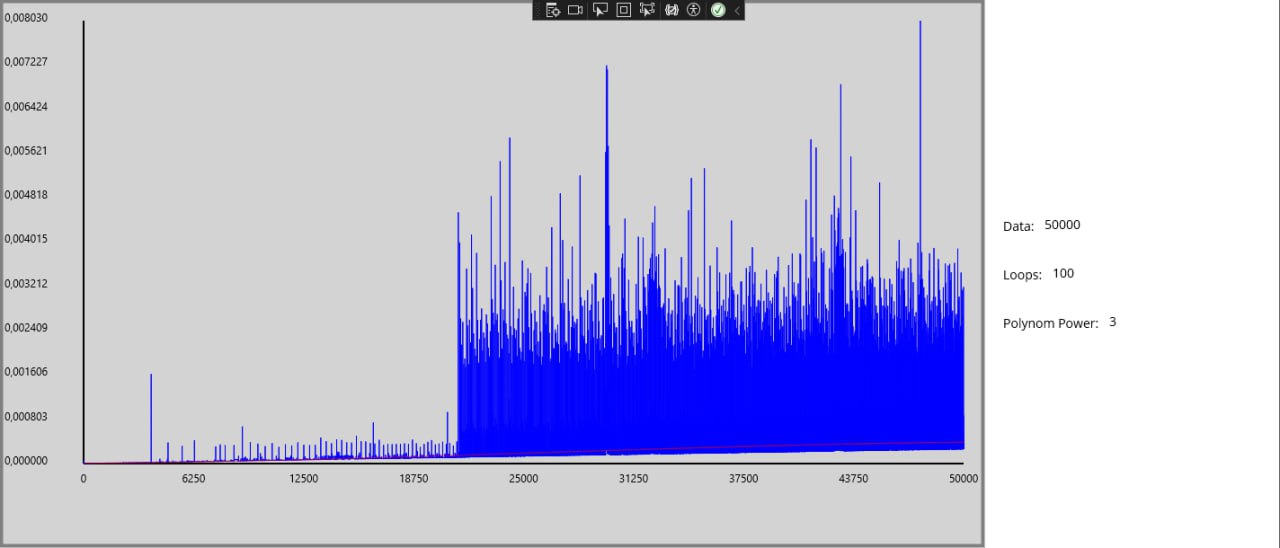
****

Рис. 1.2.1 Алгоритм суммы элементов

Код изображен на Рис. 1.2.2

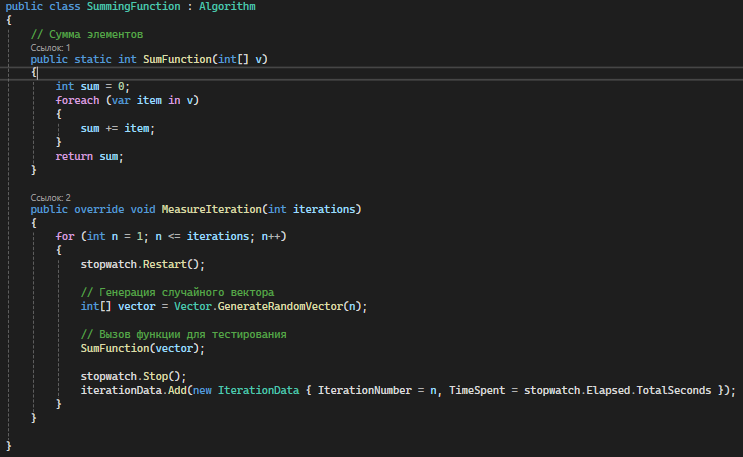


Рис. 1.2.2 Код алгоритма

1. (произведение элементов)

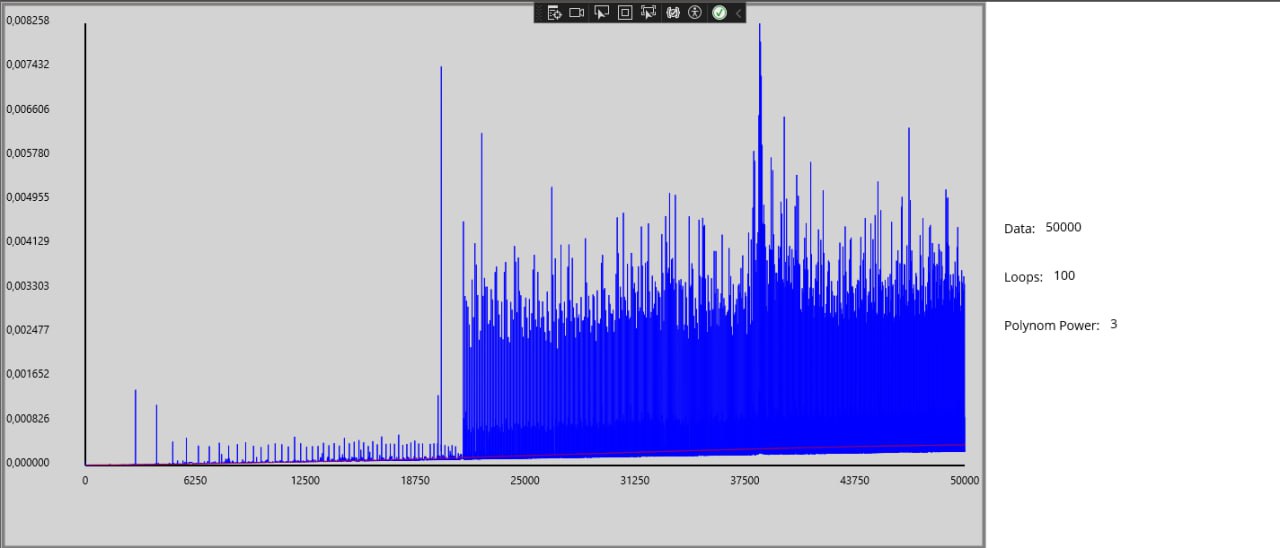
* Временная сложность: O(n)
* N=50000
* Ср. знач. на основе тестов: 100

Алгоритм выполняет операцию произведение над i-тыми элементами массива.

По графику (см. Рис. 1.3.1) данного алгоритма мы видим, что у нас получилась

линейная зависимость времени от количества входных данных необходимых на

обработку:

****Рис. 1.3.1 Алгоритм суммы элементов

Код изображен на Рис. 1.3.2

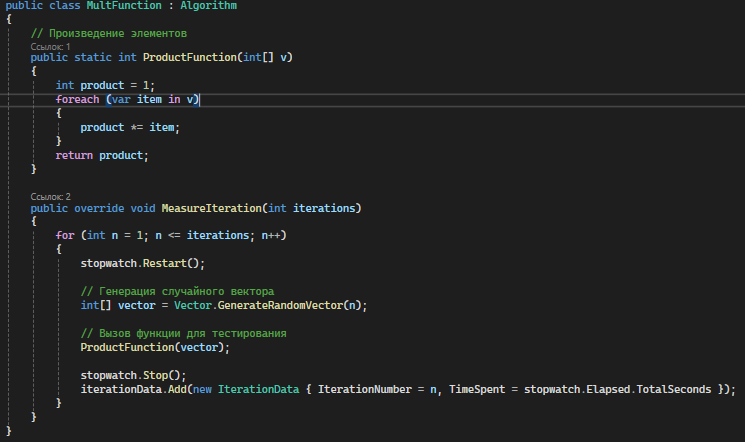


Рис. 1.3.2 Код алгоритма

* Временная сложность:
* T = O(n)
* N1=50000
* N2=50000
* Ср. знач. на основе тестов: 100

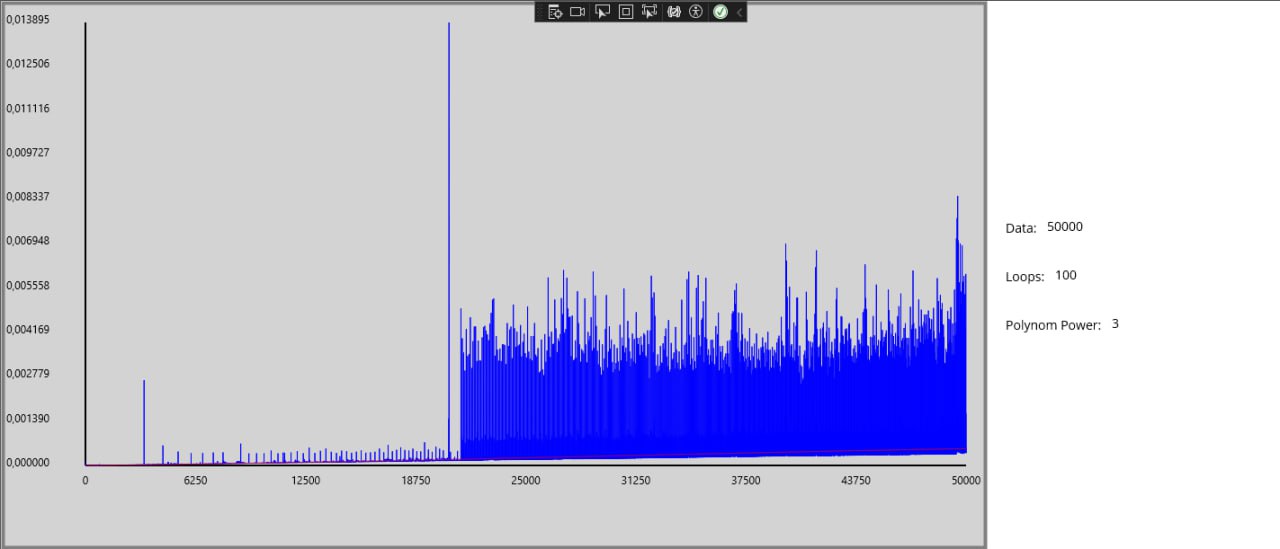
Алгоритм Горнера

Этот алгоритм позволяет вычислять многочлен степени n только с помощью

n умножения и n сложения. Это оптимально, поскольку существуют многочлены степени n, которые не могут быть вычислены меньшим количеством арифметических операций.

В качестве альтернативы, метод Хорнера также относится к методу аппроксимации корней многочленов, описанному Хорнером в 1819 году. Это вариант метода Ньютона–Рафсона, который стал более эффективным для ручных вычислений благодаря применению правила Хорнера. Он широко использовался до тех пор, пока компьютеры не вошли во всеобщее употребление примерно в 1970 году.

График зависимости времени выполнения алгоритма методом Горнера от количества элементов массива(см. Рис. 1.4.1.1).

****Рис. 1.4.1.1 Алгоритм метода Горнера.

Код изображен на Рис. 1.4.1.2

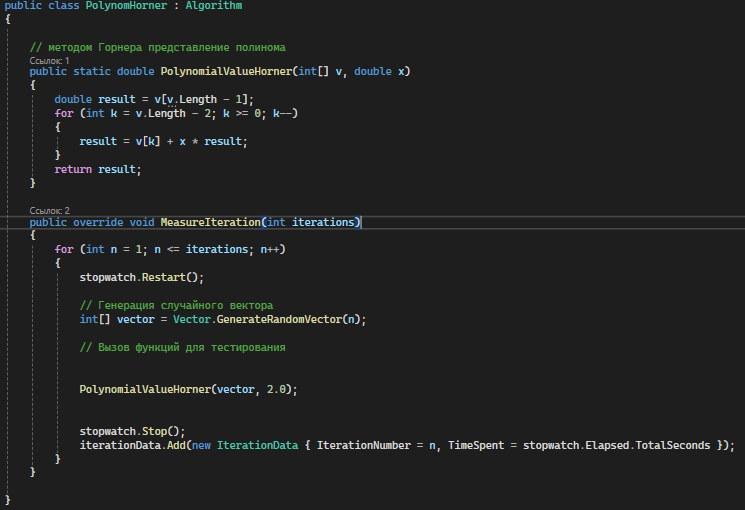
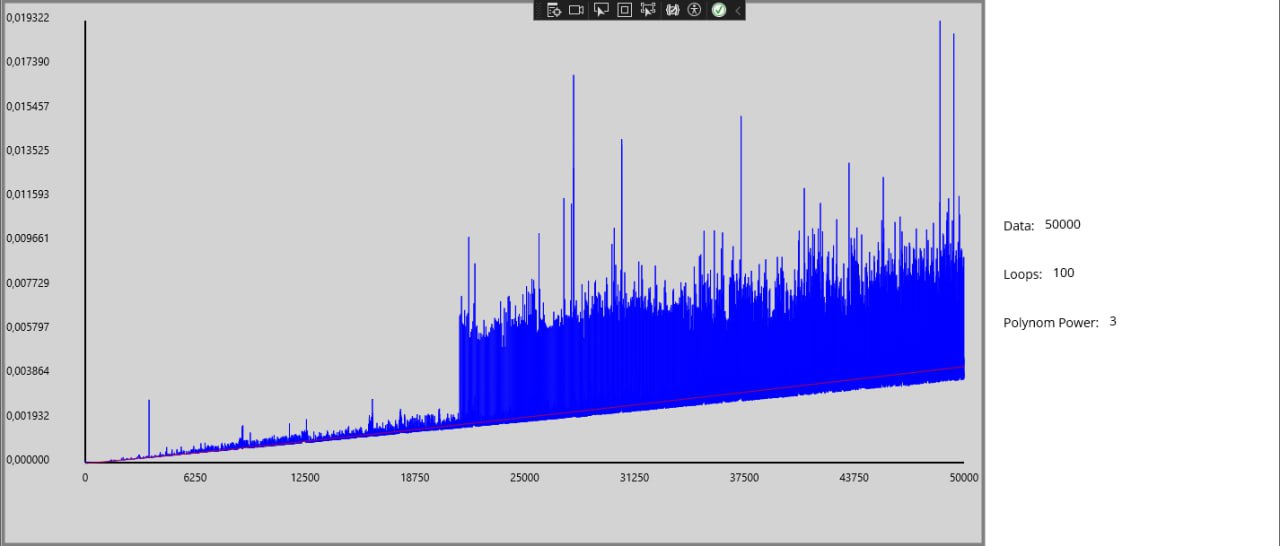


Рис. 1.4.1.2 Код метода Горнера.

График зависимости времени выполнения алгоритма Прямым(Straigth) методом от количества элементов массива(см. Рис. 1.4.2.1):

****Рис. 1.4.2.1 Алгоритм метода Горнера.

Код изображен на Рис. 1.4.2.2

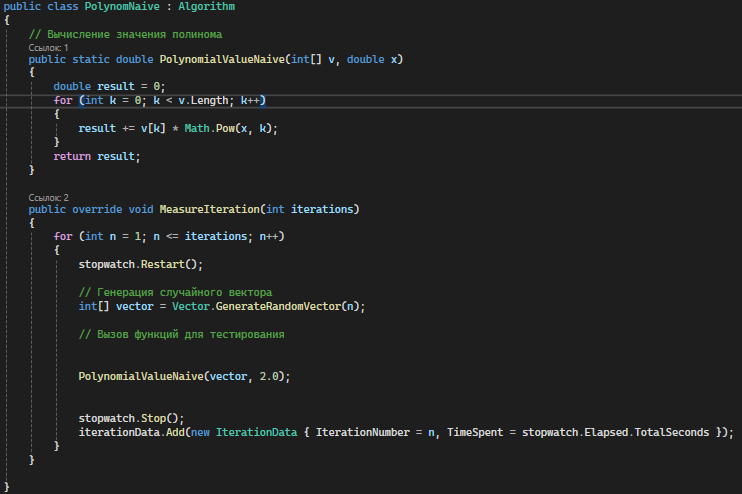


Рис. 1.4.2.2 Код метода Горнера.

Проанализировав графики (см. Рис. 1.4.1.1 и Рис. 1.4.2.1) можно сделать вывод, что метод Горнера работает в 8-9 раз быстрее, чем алгоритм прямого вычисления многочлена.

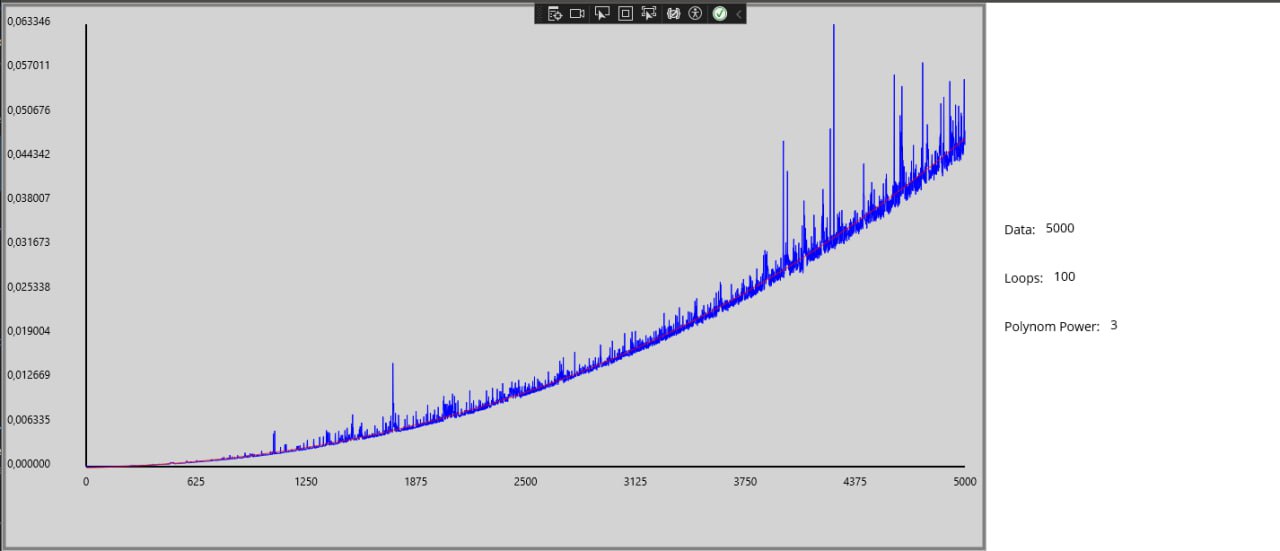
1. Алгоритм сортировки пузырьком (Bubble sort) элементов

* Временная сложность: O(n2)
* N=5000
* Ср. знач. на основе тестов: 100

Примечание: алгоритм итерируется по коллекции, меняя соседние элементы местами, если они не отсортированы между собой.

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. Нужно последовательно сравнивать значения соседних элементов и менять числа местами, если предыдущее оказывается больше последующего. Таким образом элементы с большими значениями оказываются в конце списка, а с меньшими остаются в начале. Алгоритм простейший в понимании и реализации, но эффективен он лишь для небольших массивов.

График зависимости времени выполнения сортировки пузырьком от объёма данных (см.Рис. 1.5.1):

****Рис. 1.5.1 Алгоритм сортировки пузырьком

Код изображен на Рис. 1.5.2

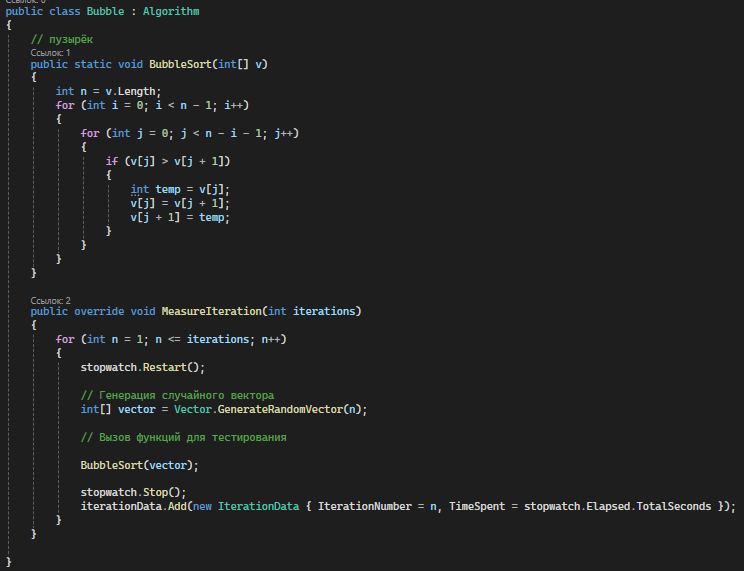


Рис. 1.5.2 Код сортировки пузырьком

1. Алгоритм быстрой сортировки (Quick sort) элементов

* Временная сложность: O(N \* log(N))
* N = 50000
* Ср. знач. на основе тестов: 100

Примечание: улучшенная сортировка пузырьком, перестановки производятся

не только между соседними элементами, после каждого прохода коллекция

рекурсивно делится на две независимых.

QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «Пузырьковая сортировка» и «Шейкерная сортировка»), известного в том числе своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы (таким образом улучшение самого неэффективного прямого метода сортировки дало в результате один из наиболее эффективных улучшенных методов).

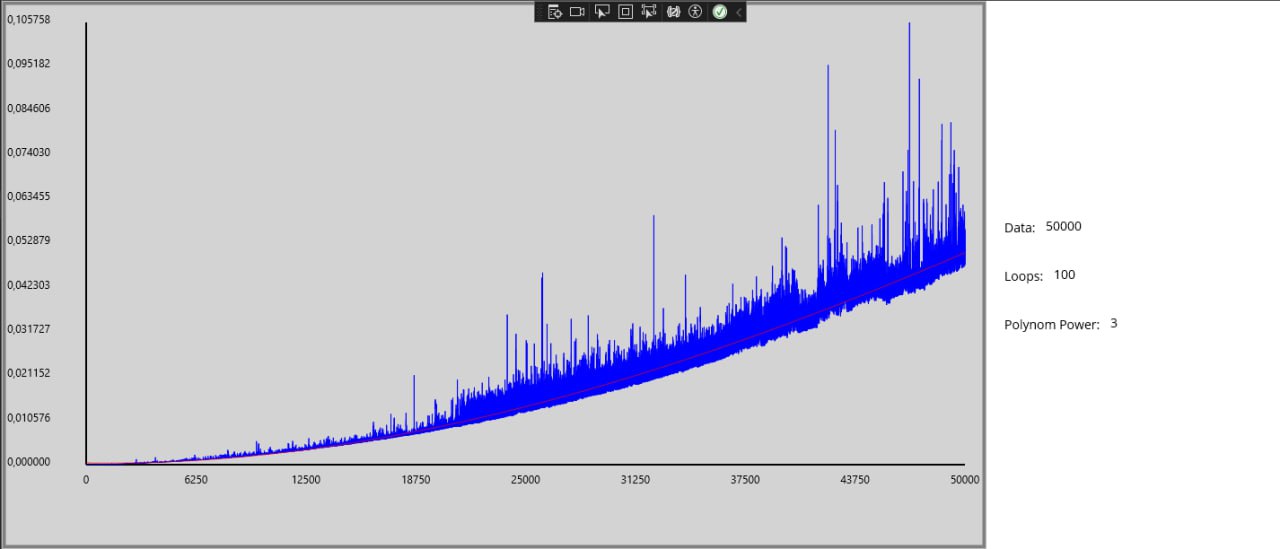
Общая идея алгоритма состоит в следующем:

Задать выбранный из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность (см. ниже).

Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного», «равные» и «большие».

Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

График зависимости времени выполнения быстрой сортировки от объёма данных(см.Рис. 1.6.1):

****Рис. 1.6.1 Алгоритм быстрой сортировки

Код изображен на Рис. 1.6.2:

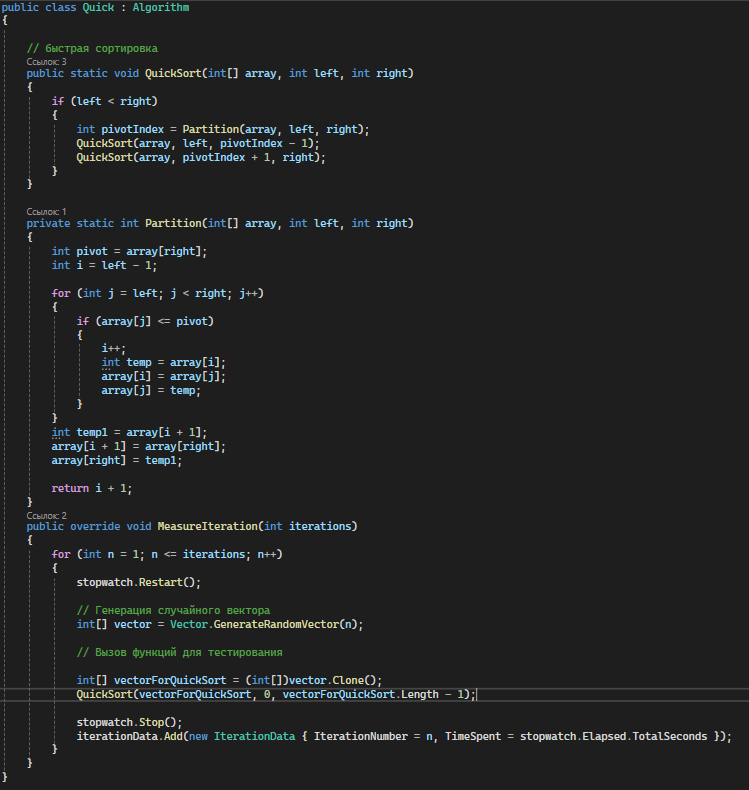


Рис 1.6.2 Код быстрой сортировки

1. Гибридный алгоритм сортировки Timsort элементов

Временная сложность: O(N \* log(N))

* N = 50 000
* Ср. знач. на основе тестов: 100

Примечание: алгоритм разбивает коллекцию на упорядоченные, сортирует их вставками и объединяет сортировкой слиянием.

Основная идея алгоритма заключается в использовании наблюдения, согласно которому на практике сортируемые (упорядочиваемые) массивы данных часто содержат отсортированные (упорядоченные) подмассивы. На таких данных алгоритм timsort сравнительно быстрее некоторых алгоритмов сортировки.

График зависимости времени выполнения Timsort от обёма даннных(см.Рис 1.7.1):

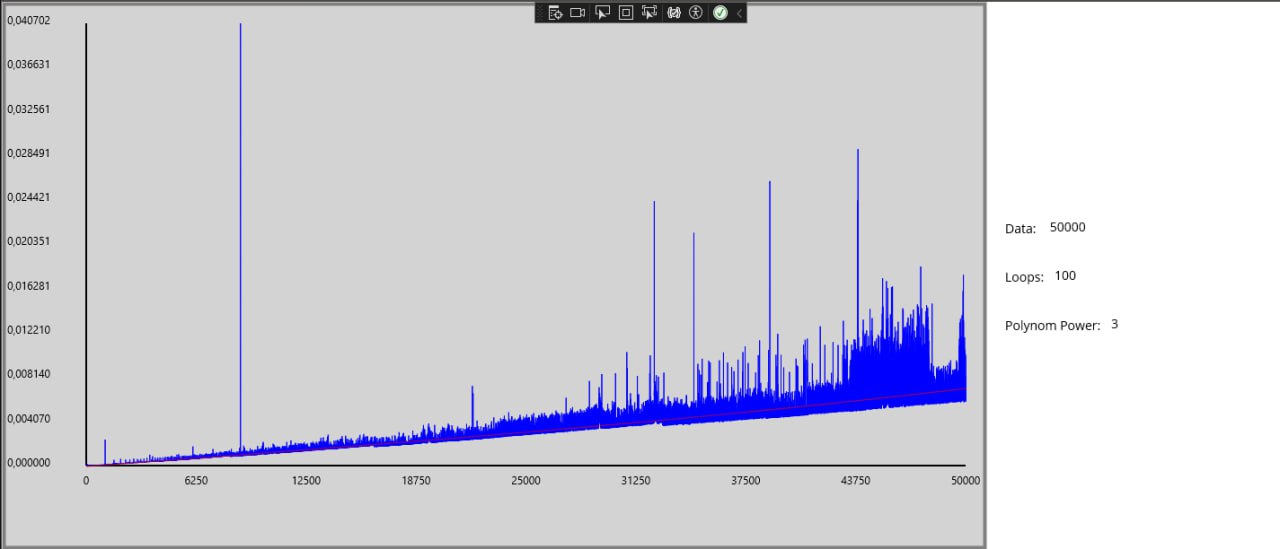


Рис. 1.7.1 Алгоритм гибридной сортировки

Код изображен на Рис. 1.7.2-1.7.4:

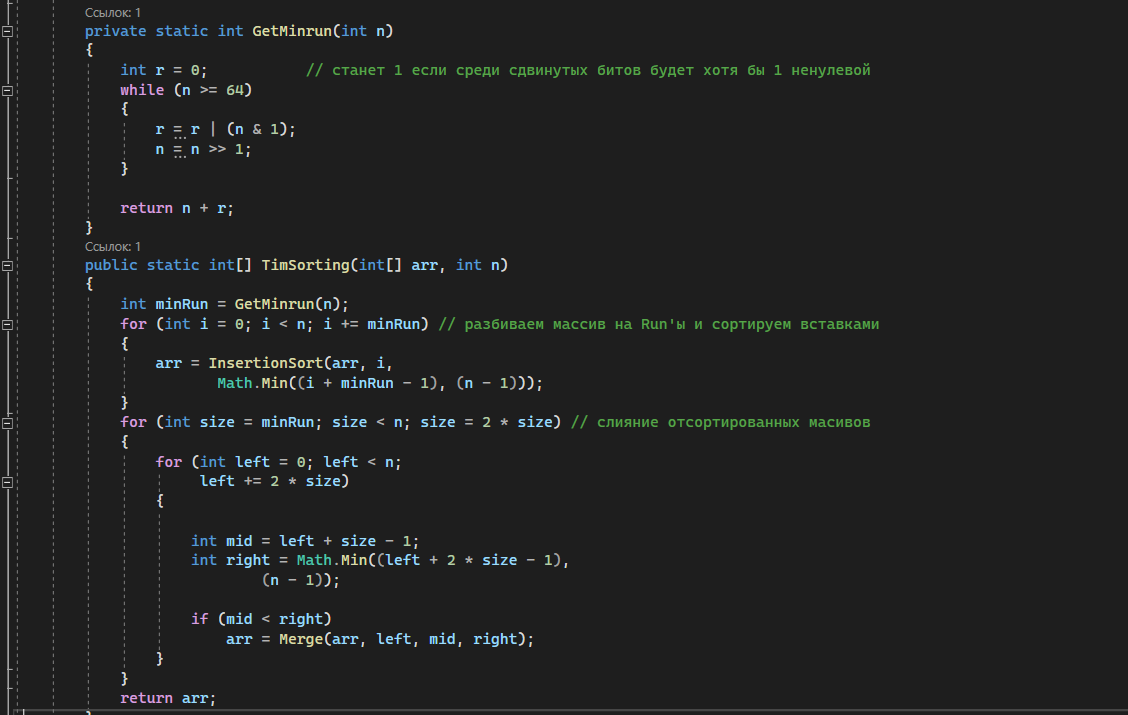


Рис 1.7.2 Код гибридной сортировки(ч.1)

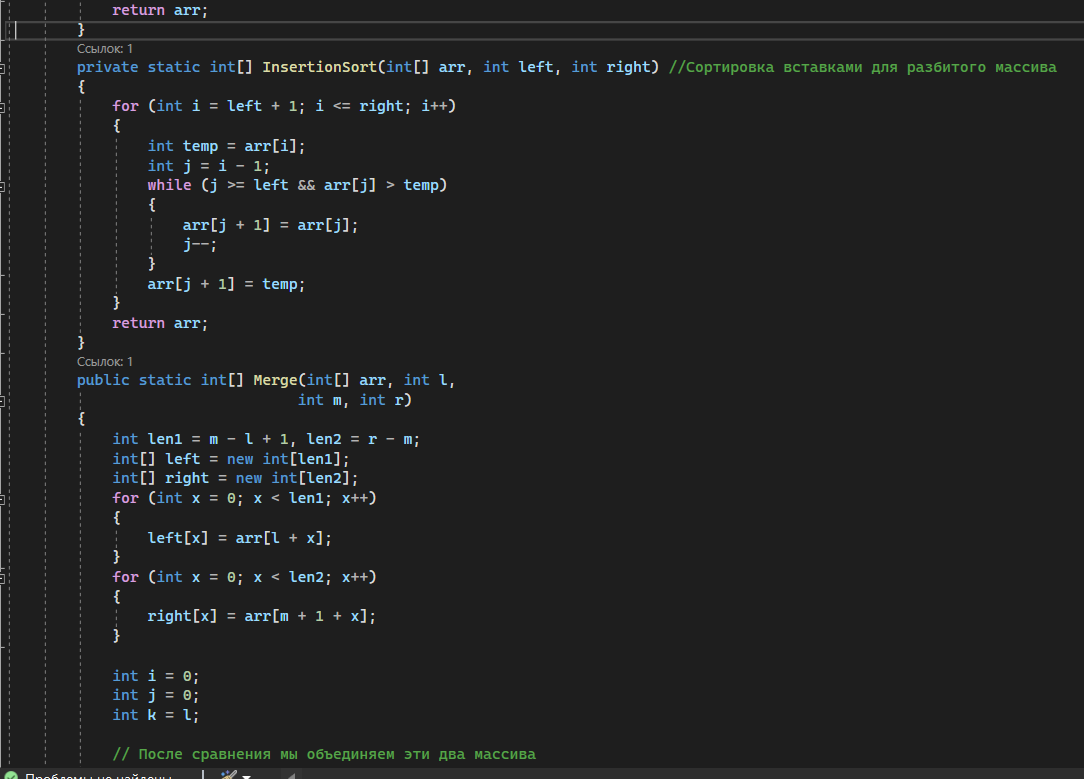


Рис 1.7.3 Код гибридной сортировки(ч.2)

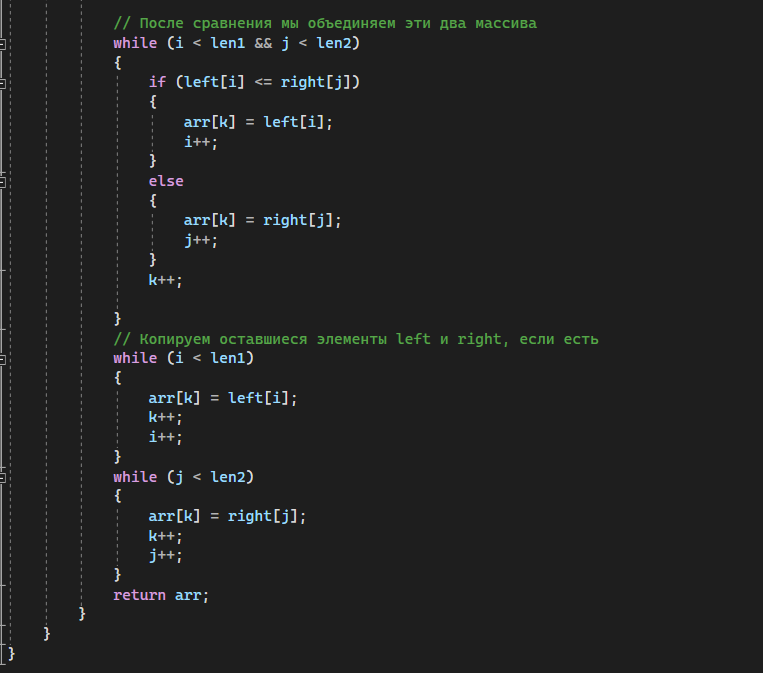


Рис 1.7.4 Код гибридной сортировки(ч.3)

1. Алгоритмы возведения в степень

8.1.Классический(см. Рис. 1.8.1.1):

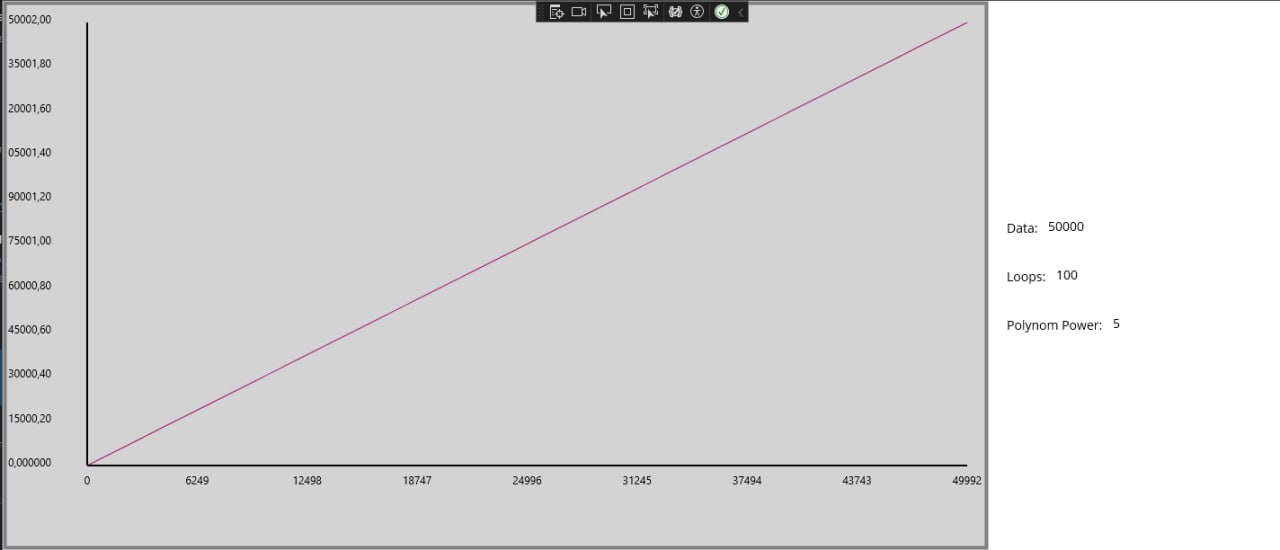


Рис. 1.8.1.1 Возведение в степень (классический алгоритм)

На данном графике видно, что кривая не имеет отклонений от линии тренда

И мы имеем достаточно большое количество шагов, что является на наш взгляд не очень эффективным.

Код изображен на Рис. 1.8.1.2 и 1.8.1.3:

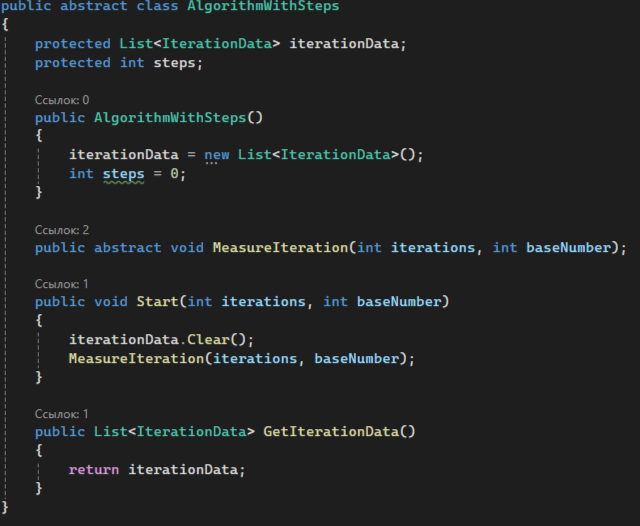


Рис.1.8.1.2 Класс AlgorithmWithSteps

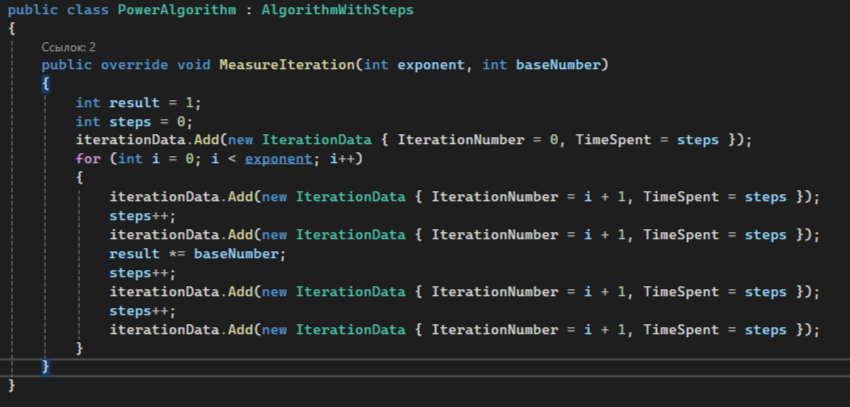


Рис. 1.8.1.3 Возведение в степень (классический алгоритм)

Класс PowerAlgorithm(см. Рис. 1.8.1.3) реализует классический алгоритм возведения в степень и наследуется от класса AlgorithmWithSteps(см. Рис. 1.8.1.2) для подсчета шагов при анализе сложности данного алгоритма.

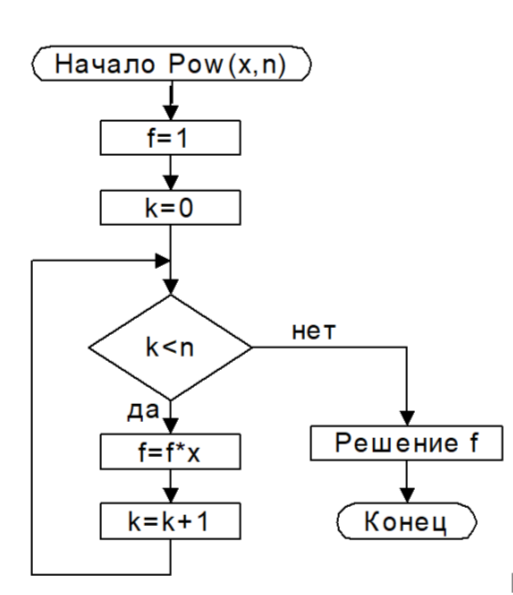


Рис. 1.8.1.4 Блок-схема Возведение в степень (классический алгоритм)

Рекурсивный(см Рис 1.8.2.1):

Рис. 1.8.2.1 Возведение в степень (рекурсивный алгоритм)

Данный график существенно отличается от предыдущего (см. Рис. 1.8.1.1) количеством шагов. Так же он имеет резкие и неравномерные скачки количества шагов.

Код изображен на Рис. 1.8.2.3:



Рис. 1.8.2.2 Вызов PowerRecursive в MeasureIteration

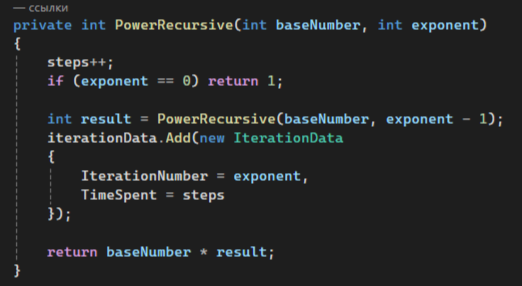


Рис. 1.8.2.3 Возведение в степень (рекурсивный алгоритм)

Метод PowerRecursive(рис. 1.8.2.3) описывает в целом реализацию алгоритма, а метод MeasureIteration(рис. 1.8.2.2) переопределяется в данном классе для подсчета шагов при анализе сложности.

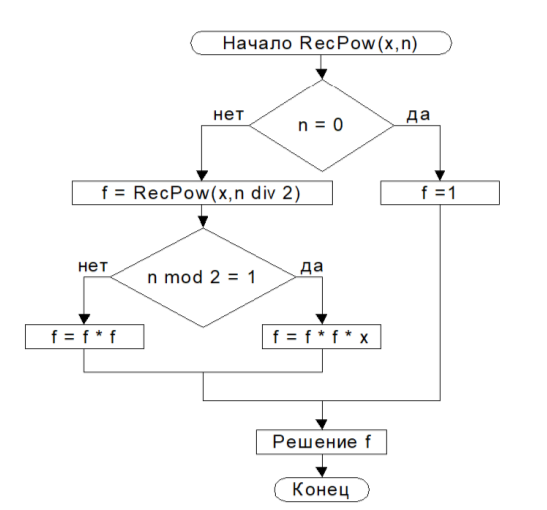
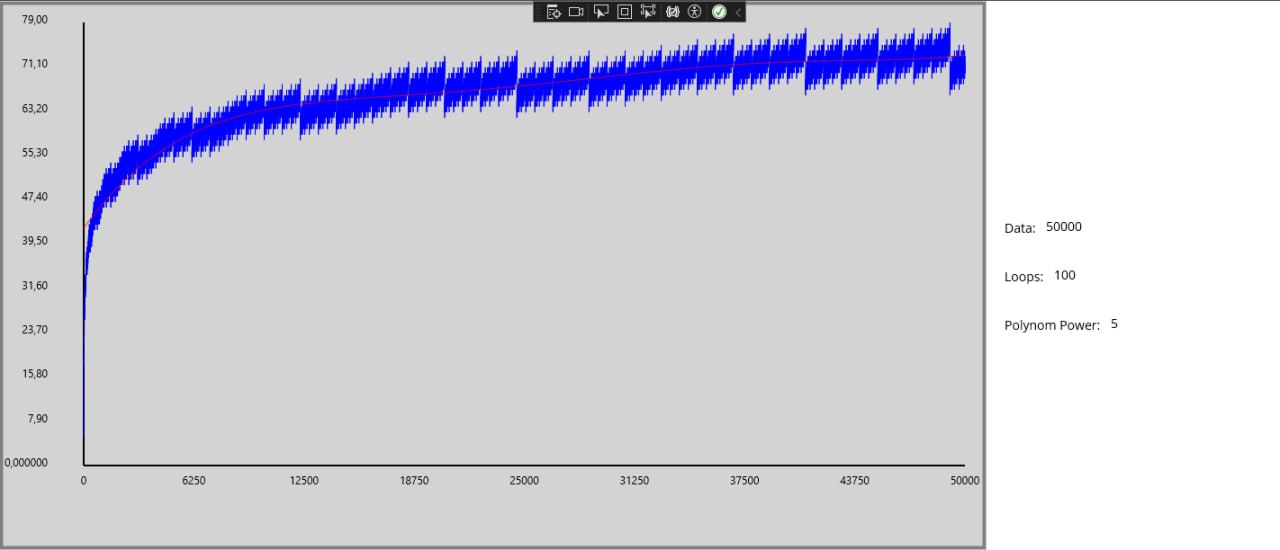


Рис. 1.8.2.4 Блок-схема возведение в степень (рекурсивный алгоритм)

Быстрый(см Рис. 1.8.3.1):

Рис. 1.8.3.1 Возведение в степень (быстрый алгоритм)

На данном графике видны существенные отклонения от теоретической кривой. Но он имеет значительно меньше шагов чем 1 алгоритм (см. Рис 1.8.1.1).

Код изображен на Рис. 1.8.3.2:

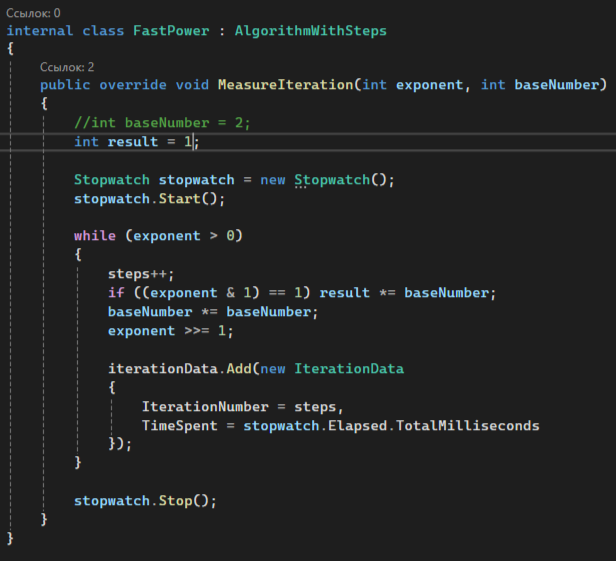


Рис. 1.8.3.2 Возведение в степень (быстрый алгоритм)

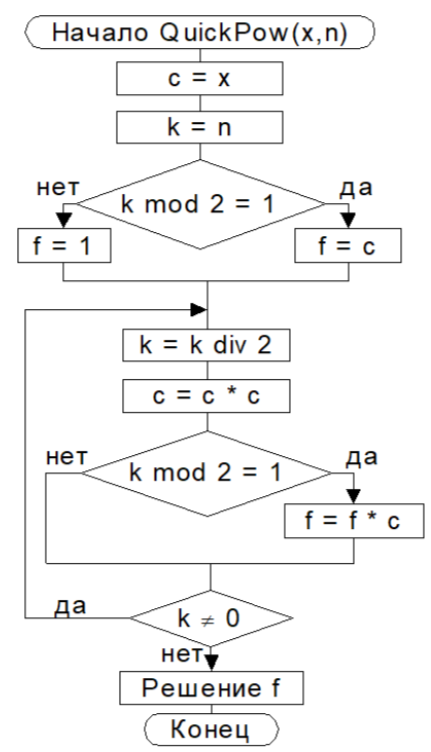


Рис. 1.8.3.3 Блок-схема возведение в степень (быстрый алгоритм)

**Задание II.**

Сгенерируйте случайные матрицы A и B размером n x n с неотрицательными

элементами. Найдите обычное матричное произведение матриц A и B.

* Временная сложность: O(N3 )
* N = 1000
* Кол-во проходов = 100

Примечание: генерировались квадратные матрицы размером N\*N, умножение

производилось прямым методом.

График зависимости времени выполнения перемножения двух заданных матриц от объёма данных(см.Рис. 2.1):

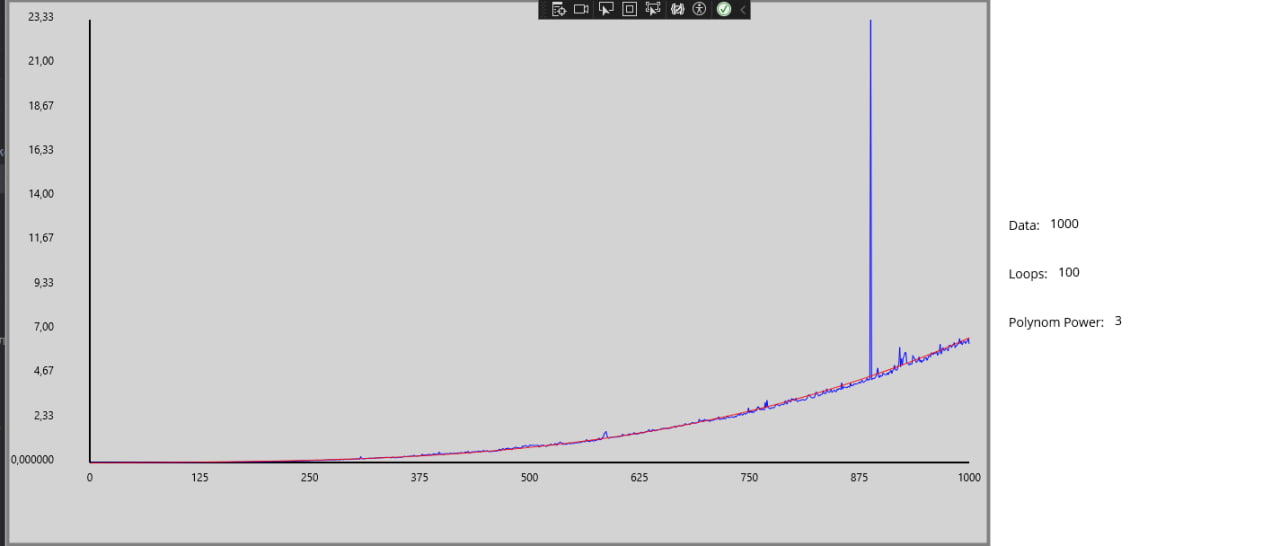


Рис. 2.1 Алгоритм перемножение двух матриц

Код изображен на Рис. 2.2 – 2.3:

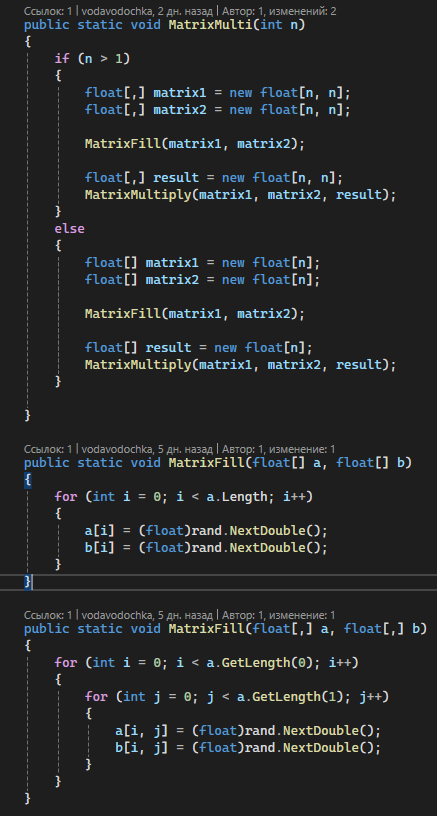


Рис. 2.2 Код для матриц (ч.1)

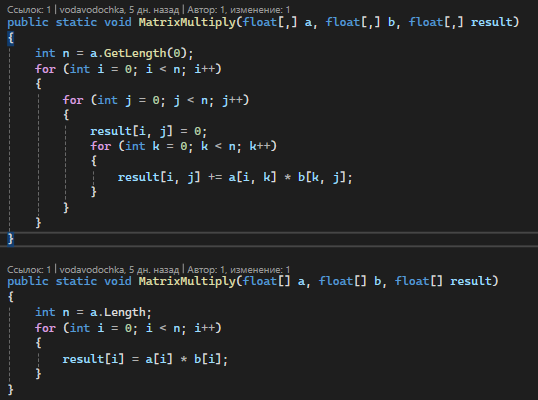


Рис. 2.3 Код для матриц (ч.2)

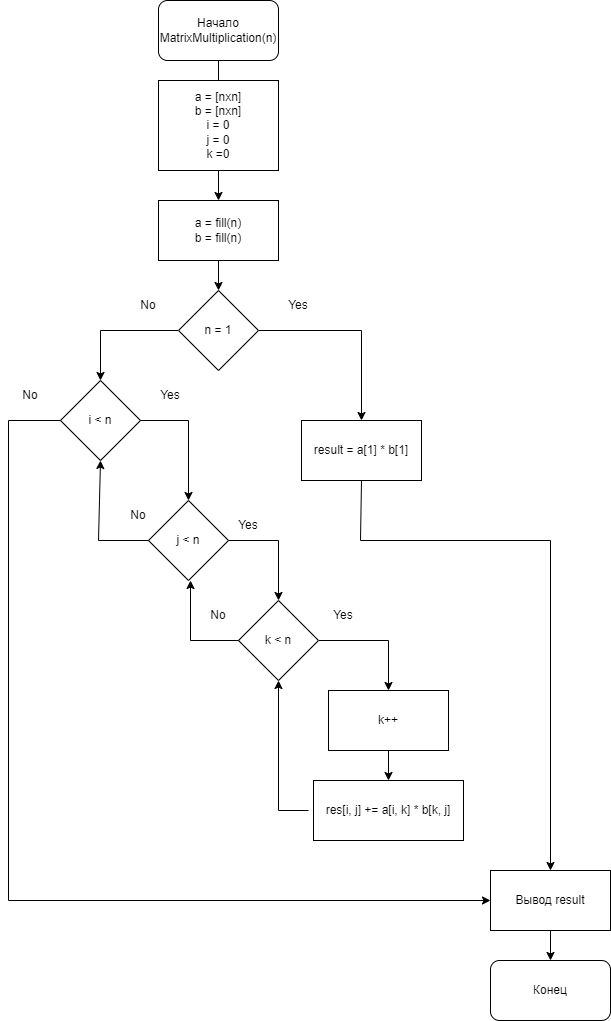
Умножение матриц на блок схемах Рис. 2.4:  


Рис. 2.4 Блок-схема для матриц

**Задание III.**

**1. RadixSort**

* Временная сложность:
* N = 50000
* Ср. знач. на основе тестов: 100

RadixSort — это алгоритм сортировки, который сортирует числа по разрядам. Он работает следующим образом:

* Сортировка по разрядам: Начинаем с наименее значимого разряда и продвигаемся к наиболее значимому.
* Использование стабильной сортировки: На каждом шаге используется стабильная сортировка (например, CountingSort) для сортировки чисел по текущему разряду.
* Повторение: Процесс повторяется для каждого разряда, пока не будут обработаны все разряды.

График времени выполнения RadixSort(см.Рис. 3.1.1):

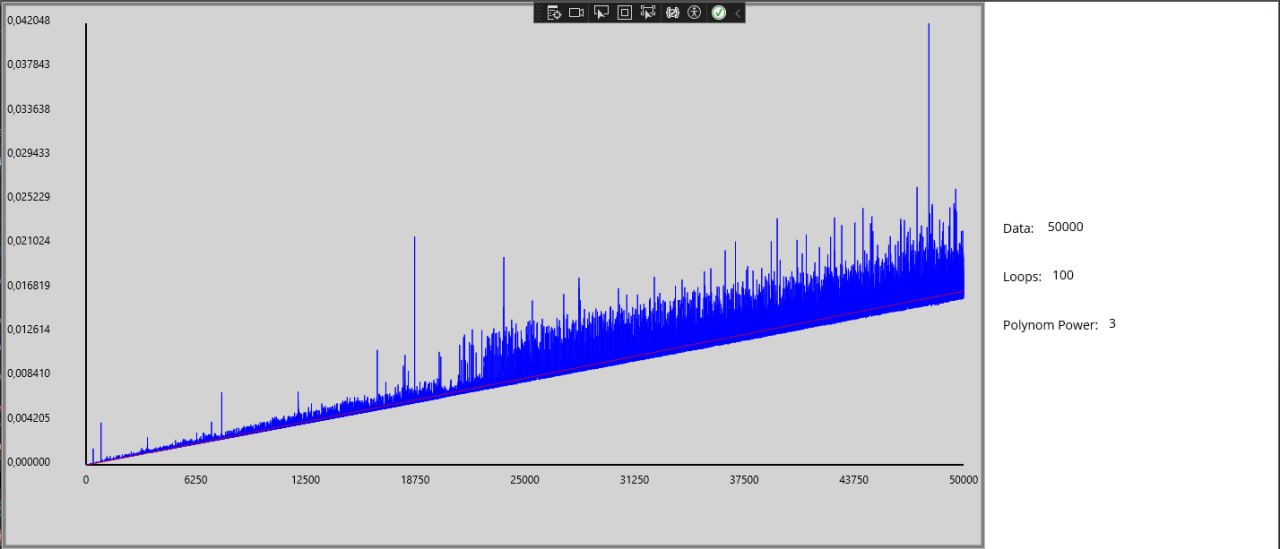


Рис. 3.1.1 алгоритм RadixSort

Код алгоритма( см. Рис. 3.1.2):

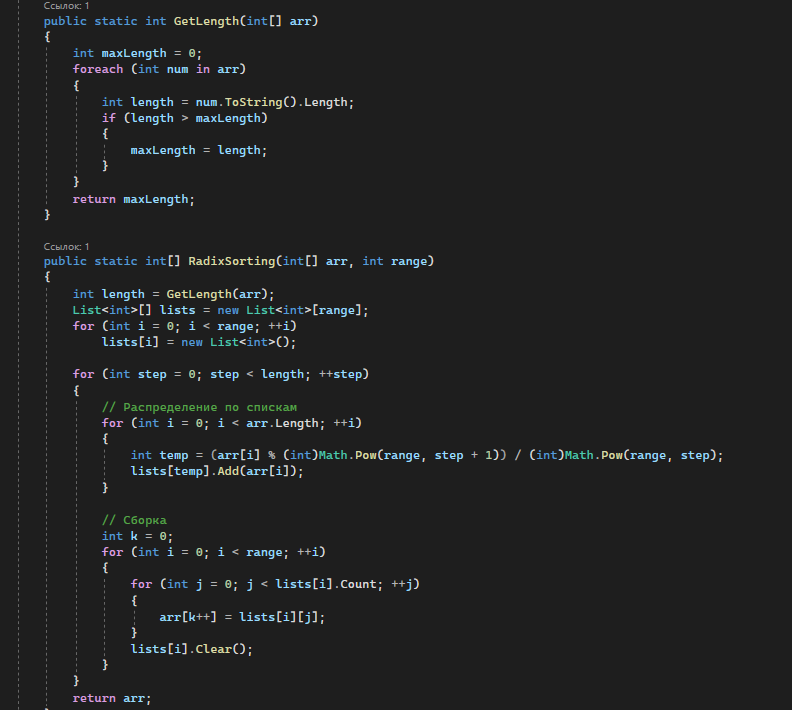
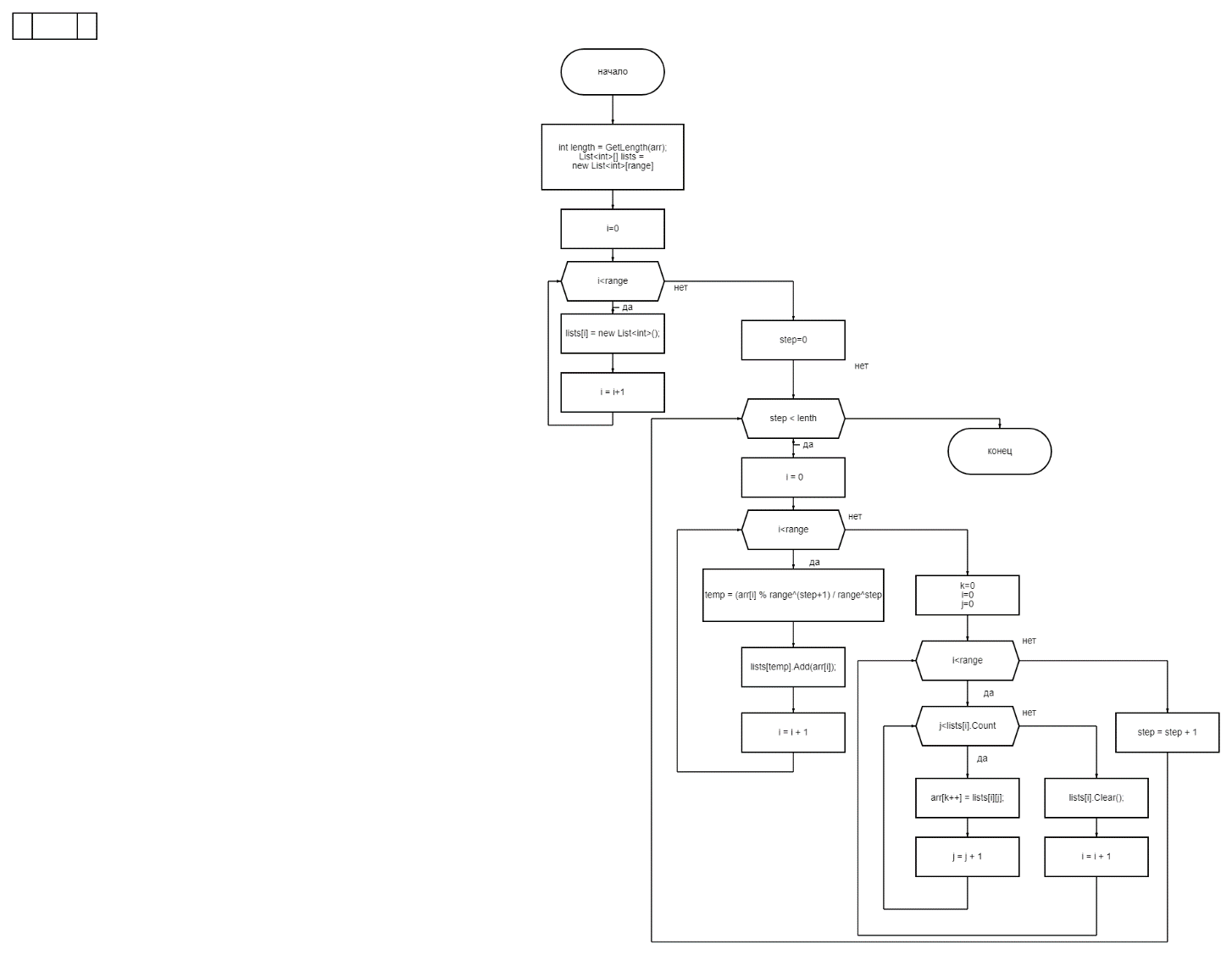


Рис. 3.1.2 код RadixSort

Рис. 3.1.3 Блок схема алгоритма RadixSort

**2**. **Longest Increasing Subsequence**

* Временная сложность: O(N2 )
* N = 10000
* Кол-во проходов = 100

Алгоритм Longest Increasing Subsequence, LIS заключается в следующем:

1. Инициализация:

* Создайте массив dp, где dp[i] будет содержать длину наибольшей возрастающей подпоследовательности, заканчивающейся на элементе arr[i].
* Инициализируйте все элементы массива dp единицами, так как каждый элемент сам по себе является подпоследовательностью длины 1.

1. Заполнение массива dp:

* Проходите по каждому элементу массива arr и для каждого элемента arr[i] проверяйте все предыдущие элементы arr[j] (где j < i).
* Если arr[j] < arr[i], то обновляйте dp[i] как dp[i] = max(dp[i], dp[j] + 1).

1. Нахождение результата:

* Максимальное значение в массиве dp будет длиной наибольшей возрастающей подпоследовательности.

На рис 3.2.1 показана реализация LIS на псевдокоде

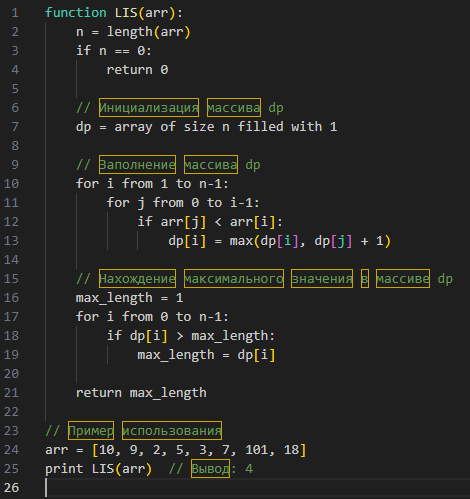


Рис. 3.2.1 Псевдокод LIS

График зависимости времени поиска наибольшей возрастающей подпоследовательности от объёма данных (см. Рис. 3.2):

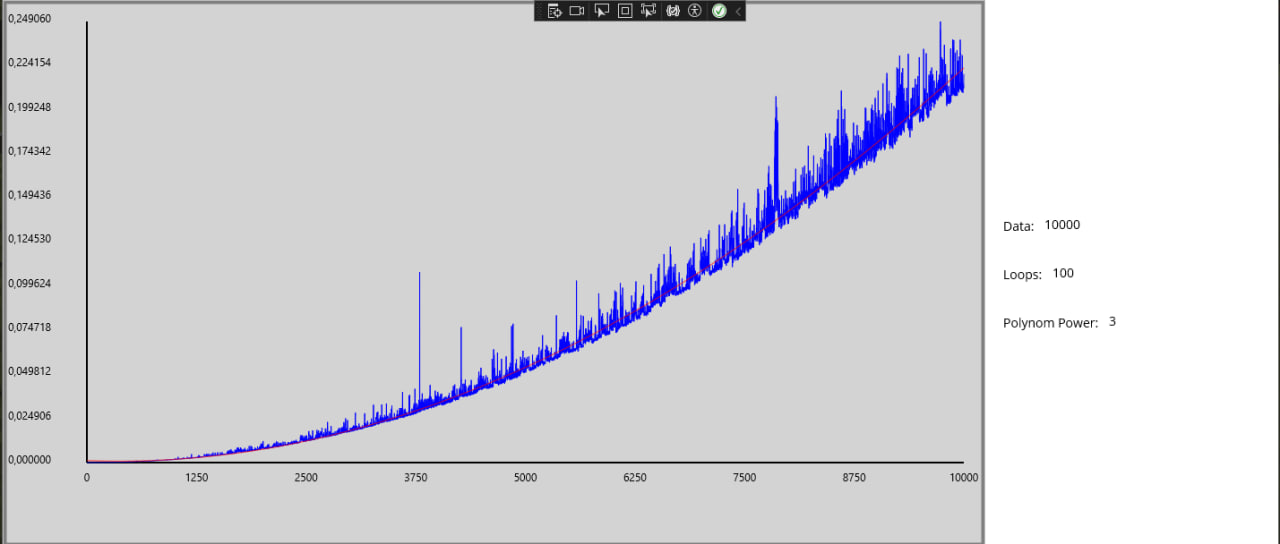


Рис. 3.2.2 Алгоритм LIS

Код изображен на Рис. 3.3:

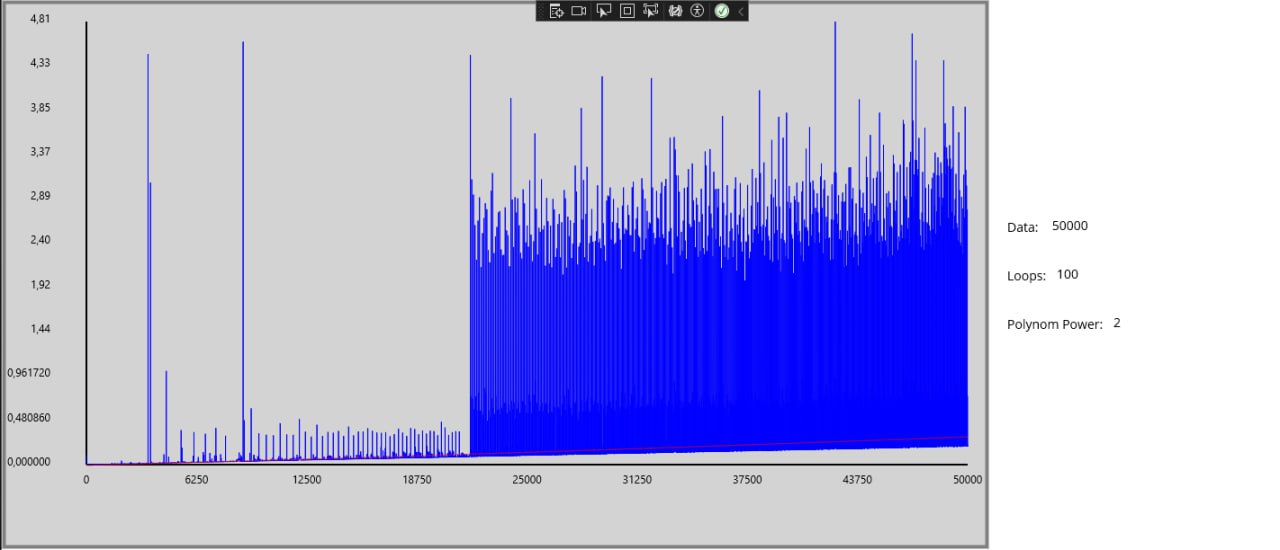
**3.Алгоритм линейного поиска LinearSearch:**

* Временная сложность: O(n)
* N = 50000
* Ср. знач. На основе тестов: 100

Алгоритм LinearSearch заключается в следующем:

* это набор инструкций для обхода данного списка/массива и проверки каждого элемента в списке/массиве, пока мы не найдем тот элемент, который ищем.
* алгоритм переходит от самого левого (или начального) элемента и продолжает поиск, пока не найдет желаемый элемент или не пройдется по всем элементам в списке.
* если элемент найден, мы вернем позицию (или index) элемента.

График оценки временной сложности алгоритма LinearSearch:

Рис. 3.3.1 График алгоритма LinearSearch

Код алгоритма (см. Рис. 3.3.2):

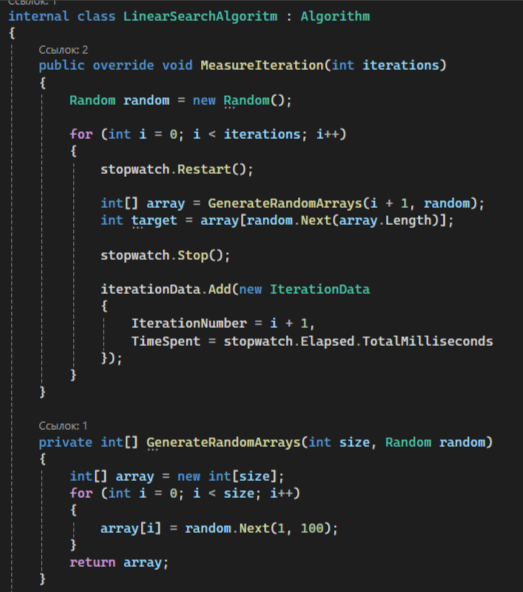


Рис. 3.3.2 Код алгоритма LinearSearch

В данном коде (см. Рис. 3.1.2) переопределяем метод MeasureIteration для оценки временной сложности алгоритма и в этом же методе прописываем сам алгоритм LinearSearch, используя метод GenerateRandomArray, который заполняет нам массив в котором мы ищем целевой элемент.

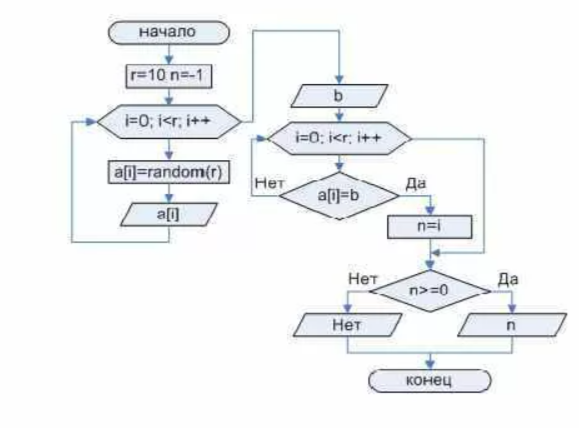
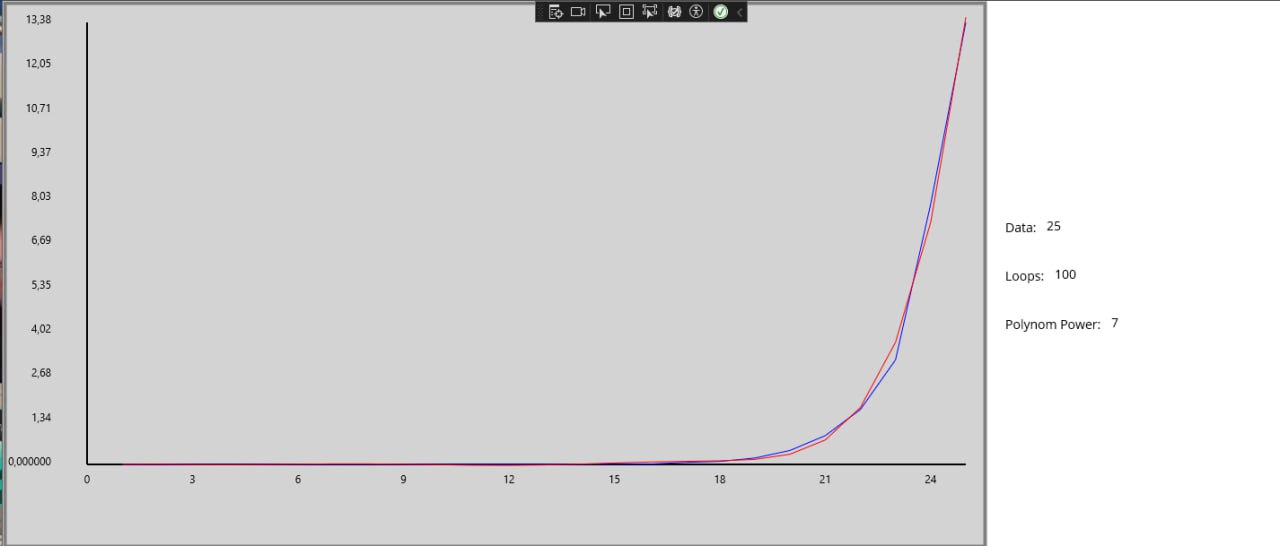


Рис. 3.3.3 Блок схема алгоритма LinearSearch

4. **Генерация всех подмножеств (Power Set)**

* Временная сложность: O(2^n)
* N = 25
* Ср. знач. на основе тестов: 100
* Сфера применения: Этот алгоритм используется в комбинаторике и теории множеств. Практические применения включают генерацию всех возможных комбинаций объектов (например, в задачах, связанных с выбором товаров или созданием наборов для тестирования).

****Рис. 3.4.1 Power Set алгоритм

Код изображен на Рис. 3.4.2:

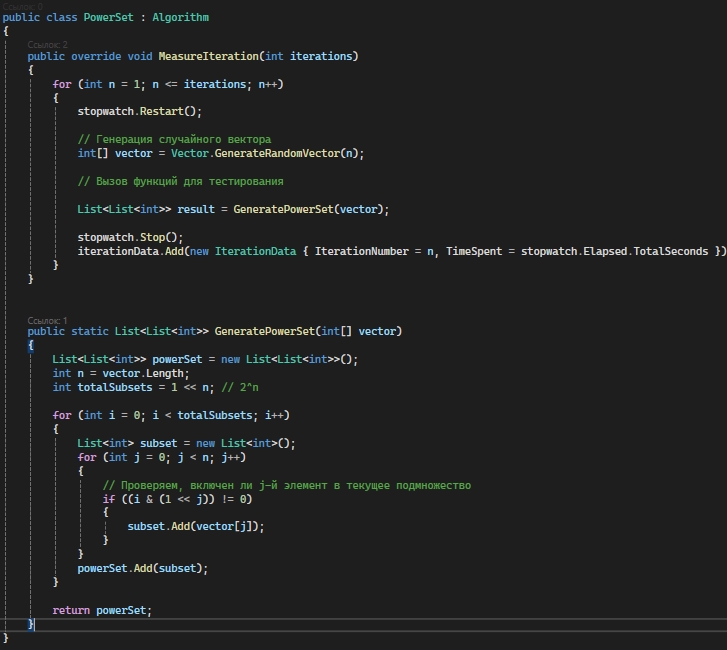


Рис. 3.4.2 Код алгоритма

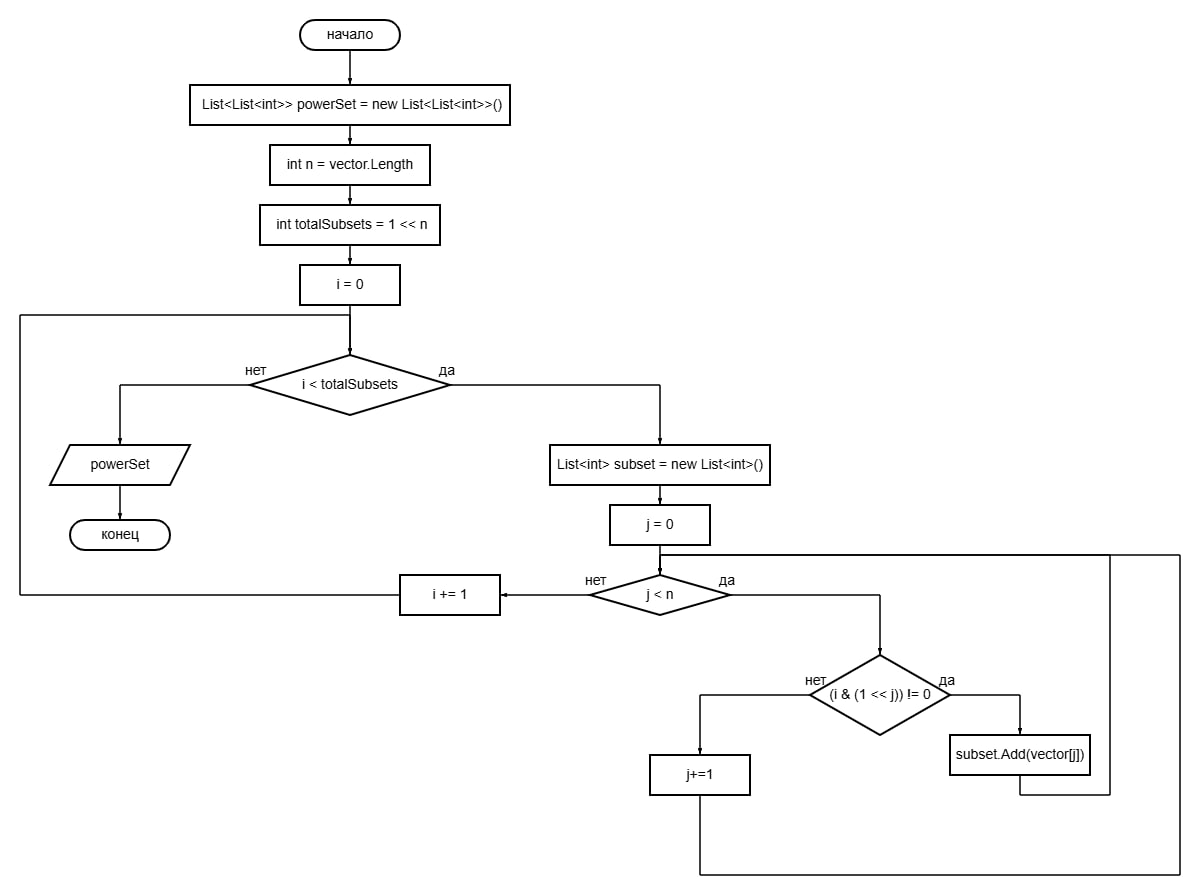
Блок-схема алгоритма   


Рис. 3.4.3 Блок схема PowerSet