МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«Челябинский государственный университет»**

**(ФГБОУ ВО «ЧелГУ»)**

Институт информационных технологий

Кафедра информационных технологий и экономической информатики

ОТЧЕТ

по лабораторной работе №1

Выполнил:

Студент группы ПрИ-202 Приходько Даниил Александрович

Студент группы ПрИ-202 Саламатин Алексей Юрьевич

Студент группы ПрИ-202 Скоробогатов Максим Дмитриевич

Принял:

Преподаватель ИИТ Николаев Иван Евгеньевич

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

дата оценка

Челябинск 2024 г.

Содержание

Введение………………………………………………………………………….…3

Блок 1

Постоянная функция………………………………………………………….……4

Сумма элементов…………………………………………………………………...5

Произведение элементов ………………………………………………………….7

Алгоритм вычисления полинома………………………………………………….9

Метод Горнера……………………………………………………………………..11

Алгоритм сортировки пузырьком………………………………………………...13

Алгоритм быстрой сортировки…………………………………………………...15

Возведение в степень……………………………………………………..……….18

Алгоритм Тимсорт………………………………………………………………...26

Блок 2

Матричное произведение…………………………………………………...……30

Блок 3

Слияние…………………………………………………………………………….33

Алгоритм разбиения множества на два подмножества с равной суммой……..36

Алгоритм Дейкстры……………………………………………………………….38

Модель проекта……………………………………………………………………41

Введение

Цель работы

Цель данной лабораторной работы — провести эмпирический анализ временной сложности алгоритмов. В рамках работы вводится понятие временной сложности алгоритма, рассматривается математический аппарат для оценки временной сложности и правила применения этого аппарата.

Задание

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.

Теоретическая часть

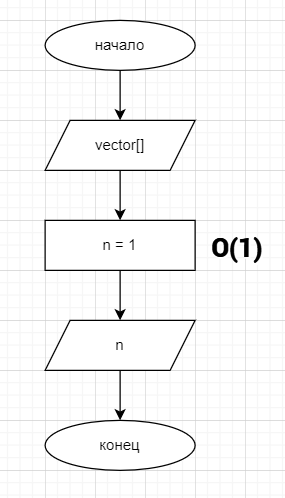
На деле был произведен уход от начально задания, так как объем входных данных был недостаточен для наглядной оценки временной сложности работы предоставленных алгоритмов. Длина входного вектора была изменена и теперь пользователь выбирает количество подаваемых на вход программе чисел в пределе от 1 до 100000, для того чтобы увидеть прогрессию функции на большом размере входного вектора. Помимо этого также и количество запусков выбирается пользователем и варьируется от 1 до 10. Также пользователю была предоставлена возможность самостоятельно выбирать максимальный размер случайного числа, в размере до 1000. Для алгоритмов возведения в степень, есть возможность выбирать степени в пределах от 2 до 200. В алгоритме перемножения матриц можно выбрать размер стороны квадратной матрицы от 1 до 5000. А в алгоритме Дейкстры предоставляется возможность указать количество вершин графа также от 1 до 7000.

Блок 1

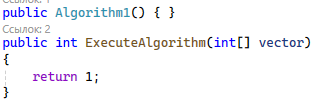
1) Постоянная функция – функция которая независимо от количества и размера входных данных возвращает 1.

Теоретическая сложность O(1) , т.к не зависит от n (рис.1.1)

Код алгоритма представлен на рисунке 1.2

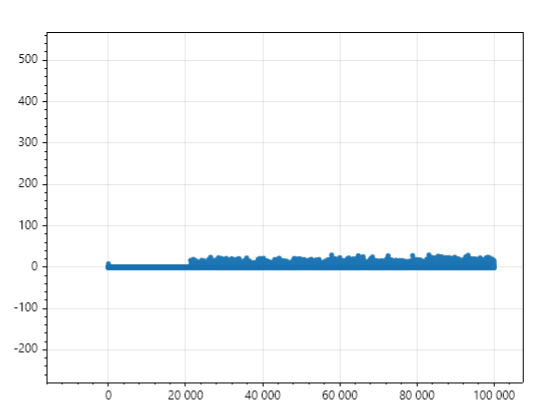


**рис.1.1 Постоянная функция, блок-схема и сложность**



**рис.1.2 Постоянная функция, код**

После обработки массива с данными программа вывела график (рис.1.3)

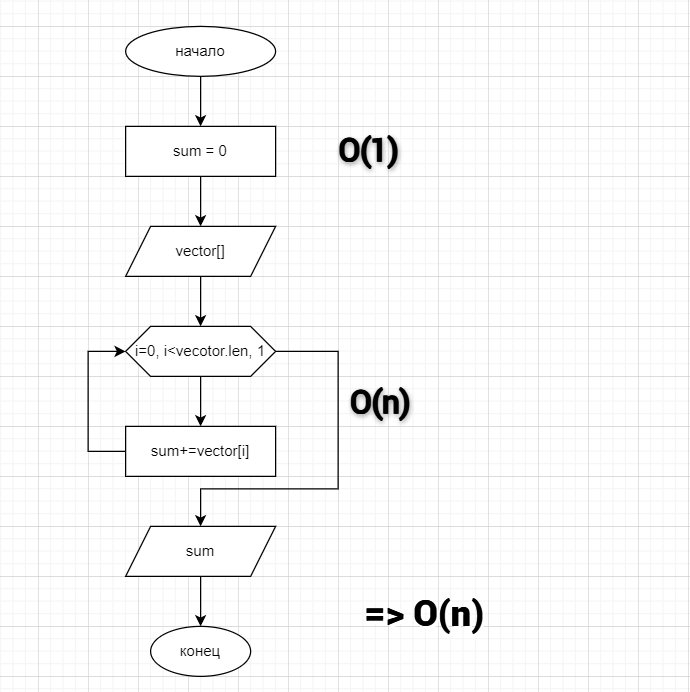


**рис.1.3 Постоянная функция, график**

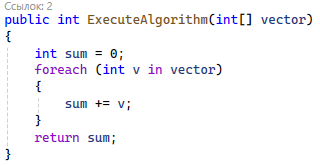
2) Сумма элементов – алгоритм, который перебором последовательно складывает все элементы вектора и возвращает их сумму

Теоретическая сложность O(n) (рис.2.1)

Код алгоритма представлен на рисунке 2.2

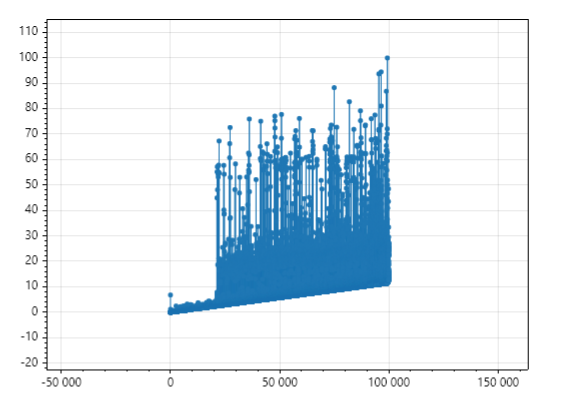


**рис.2.1 Сумма элементов, блока-схема и сложность**



**Рис.2.2 Сумма элементов, код**

После обработки массива с данными программа вывела график (рис.2.3)

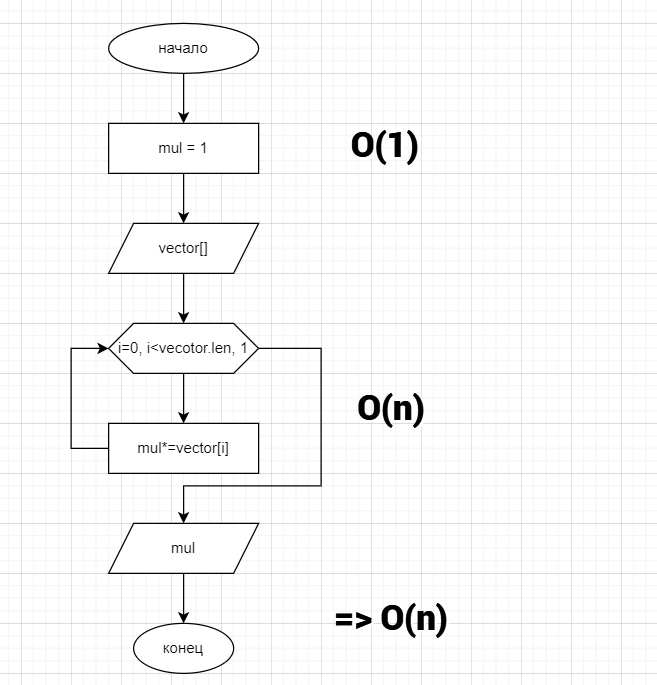


**рис.2.3 Сумма элементов, график**

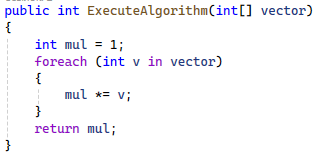
3) Произведение элементов - алгоритм, который перебором последовательно пееремножает все элементы вектора и возвращает их произведение

Теоретическая сложность O(n) (рис.3.1)

Код алгоритма представлен на рисунке 3.2

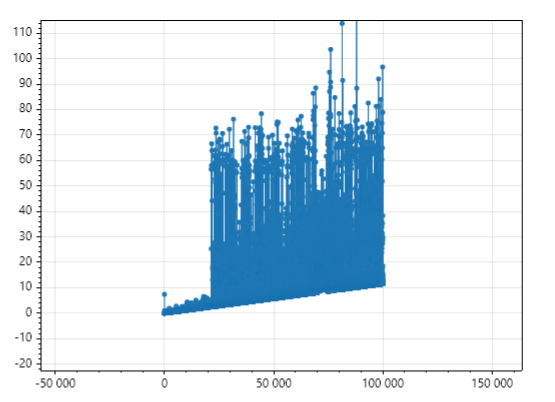


**рис.4.1 Произведение элементов, блок-схема и сложность**



**рис.3.2 Произведение элементов, код**

После обработки массива с данными программа вывела график (рис.3.3)

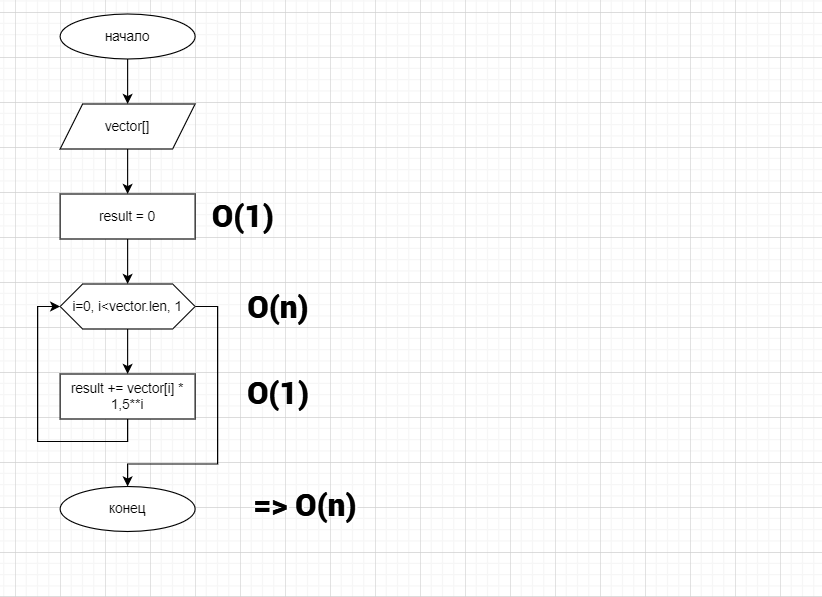


**рис.3.3 Произведение элементов, график**

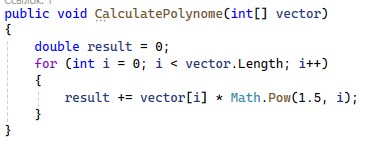
4) Алгоритм вычисления полинома - это последовательность шагов, которая позволяет вычислить значение полинома для заданных значений переменных. Обычно полином представляется в виде суммы слагаемых, каждое из которых содержит переменные, возведенные в некоторую степень. Для вычисления значения полинома нужно подставить заданные значения переменных вместо переменных в каждом слагаемом, произвести вычисления и сложить полученные результаты. Таким образом, алгоритм вычисления полинома помогает выяснить, какое значение полинома будет при конкретных значениях переменных.

Теоретическая сложность O(n) (рис.4.1)

Код алгоритма представлен на рисунке 4.2

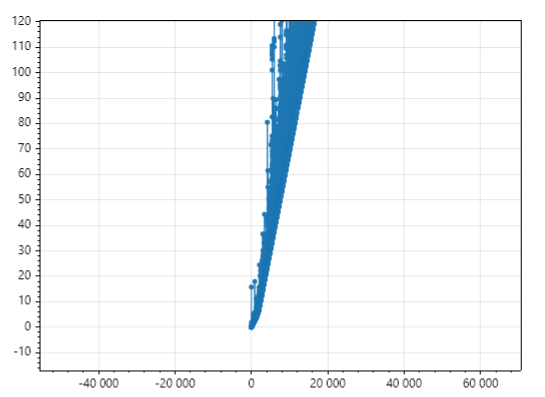


**рис.5.1 Вычисление полинома, блок-схема и сложность**



**рис.4.2 Вычисление полинома, код**

После обработки массива с данными программа вывела график (рис.4.3)

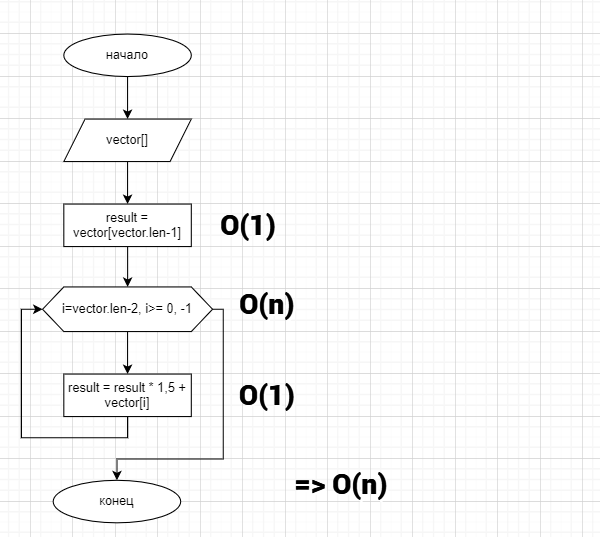


**рис.4.3 Вычисление полинома, график**

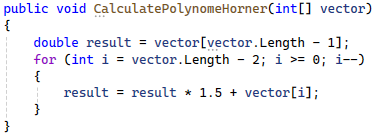
5) Метод Горнера - это метод вычисления значения полинома в точке, который позволяет уменьшить количество операций умножения при вычислении значений многочлена. Вместо того, чтобы вычислять каждое слагаемое по отдельности и складывать их все, используется более эффективный подход, который позволяет вычислить значение полинома за меньшее количество шагов.

Теоретическая сложность O(n) (рис.5.1)

Код алгоритма представлен на рисунке 5.2

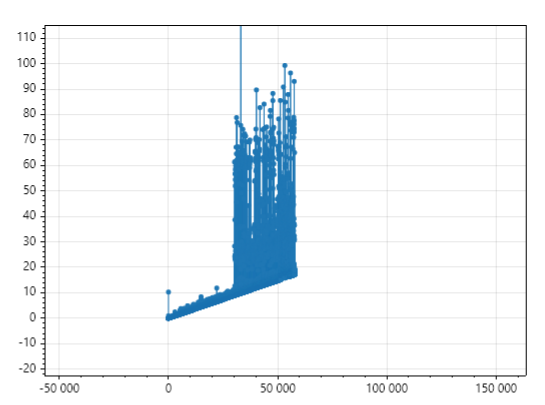


**рис.6.1 Метод Горнера, блок-схема и сложность**



**рис.5.2 Метод Горнера, код**

После обработки массива с данными программа вывела график (рис.5.3)

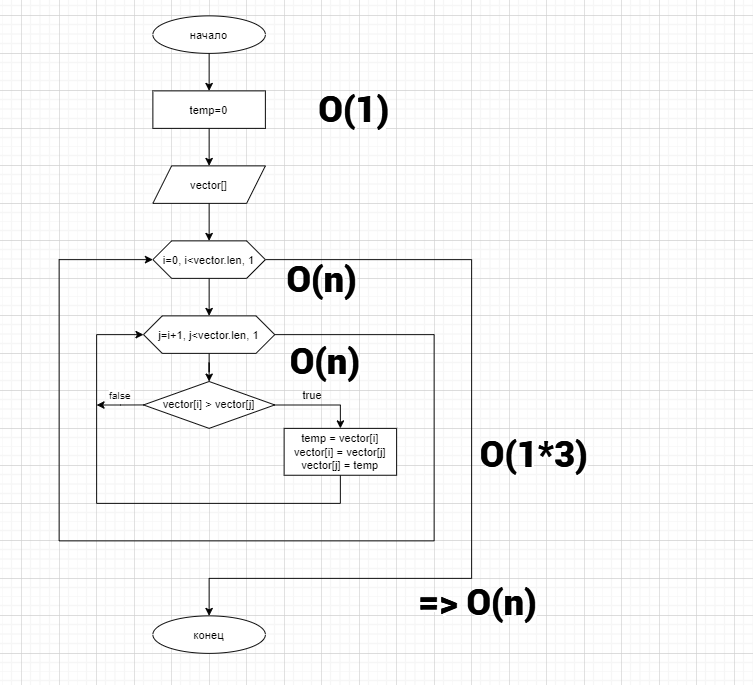


**рис.5.3 Метод Горнера, график**

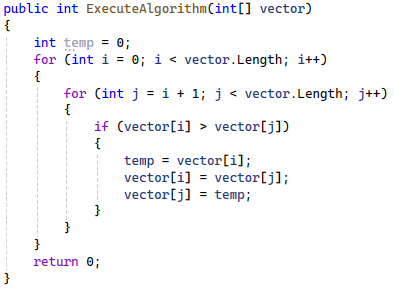
6) Алгоритм сортировки пузырьком - это простой алгоритм сортировки, который последовательно проходит по списку значений, сравнивая соседние элементы и меняя их местами, если они находятся в неправильном порядке. При каждом проходе по списку самый большой (или самый маленький, в зависимости от порядка сортировки) элемент "всплывает" на своё место, подобно пузырьку в воде, отсюда и название алгоритма. Процесс повторяется до тех пор, пока список не будет полностью отсортирован. Сортировка пузырьком неэффективна на больших наборах данных, но благодаря своей простоте она часто используется в образовательных целях или для сортировки небольших наборов данных.

Теоретическая сложность O(n^2) (рис.6.1)

Код алгоритма представлен на рисунке 6.2

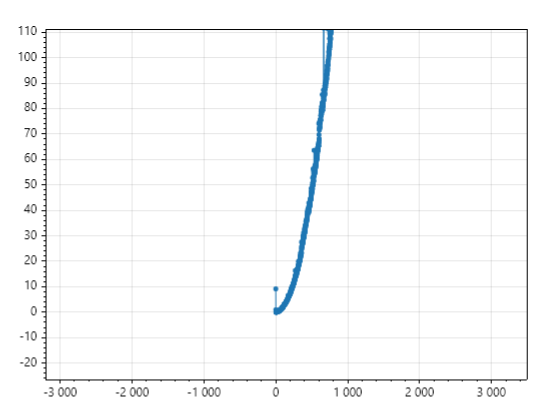


**рис.7.1 Сортировка пузырьком, блок-схема и сложность**



**рис.6.2 Сортировка пузырьком, код**

После обработки массива с данными программа вывела график (рис.6.3)



**рис.6.3 Сортировка пузырьком, график**

7) Алгоритм быстрой сортировки (Quick Sort) - это эффективный алгоритм сортировки, который использует метод "разделяй и властвуй". Он работает следующим образом:

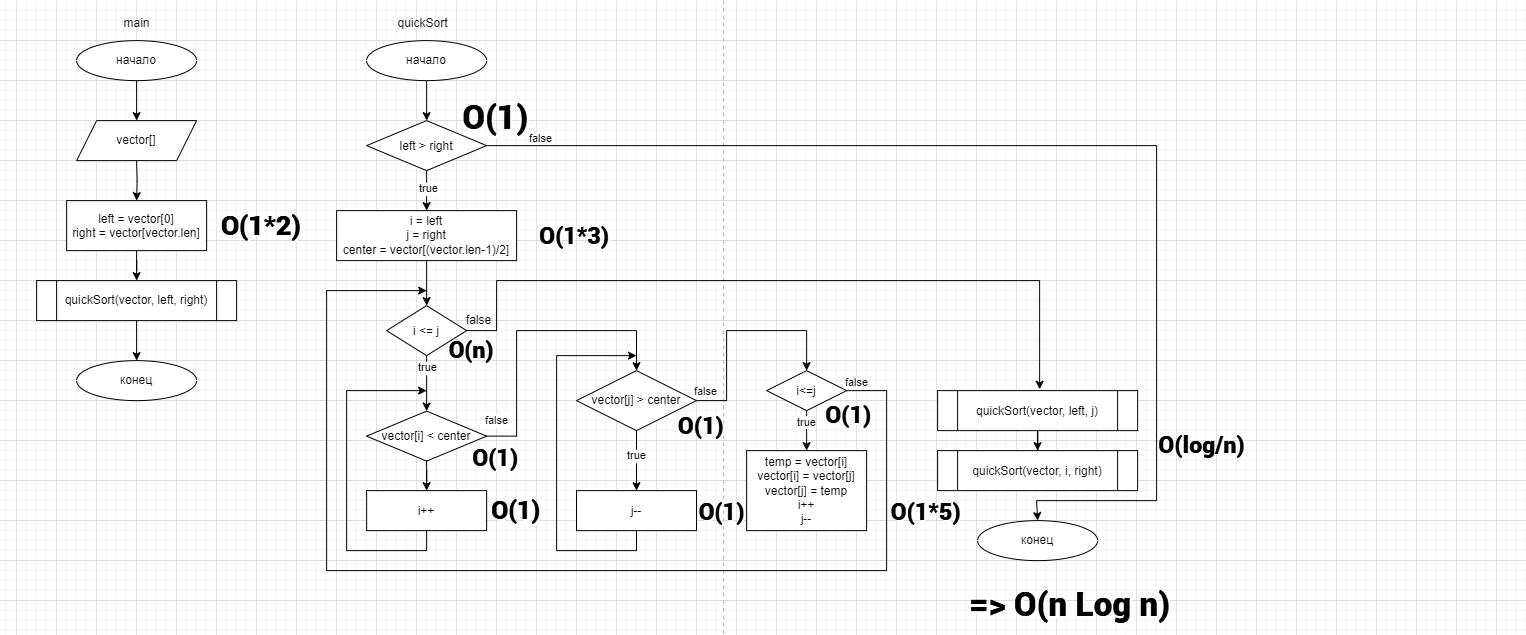
1. **Выбор опорного элемента**: Из массива выбирается один элемент, который называют опорным. Опорный элемент можно выбрать разными способами, например, взять первый, последний элемент или элемент, стоящий в середине массива.
2. **Разделение**: Массив делится на две части. Все элементы, которые меньше опорного, перемещаются влево от него, а все элементы, которые больше - вправо. Это называется операцией "разделение".
3. **Рекурсивная сортировка подмассивов**: Алгоритм рекурсивно применяется к двум получившимся частям массива - левому и правому подмассиву по отношению к опорному элементу.
4. **Слияние**: Обе части объединяются, хотя фактически в алгоритме Quick Sort само слияние не требуется, так как элементы уже находятся на своих окончательных местах.

Алгоритм завершает работу, когда все массивы, которые нужно сортировать, становятся единичными или пустыми, что, безусловно, означает, что они отсортированы.

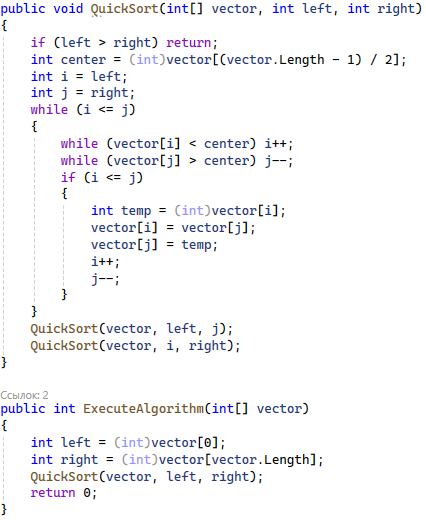
Quick Sort популярен из-за своей эффективности и обычно работает быстрее, чем другие алгоритмы сортировки, такие как сортировка пузырьком или сортировка выбором.

Теоретическая сложность O(n log n) (рис.7.1)

Код алгоритма представлен на рисунке 7.2

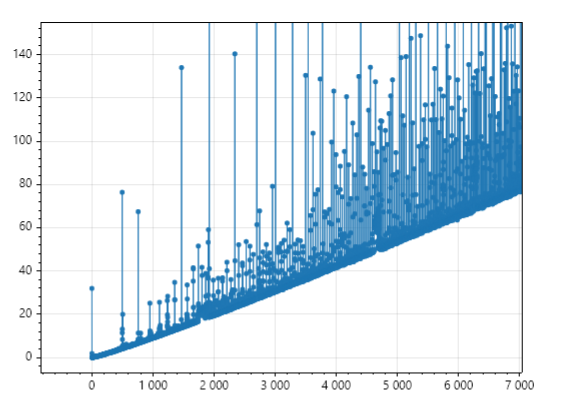


**рис.8.1 QuickSort, блок-схема и сложность**



**рис.7.2 QuickSort, код**

После обработки массива с данными программа вывела график (рис.7.3)



**рис.7.3 QuickSort, график**

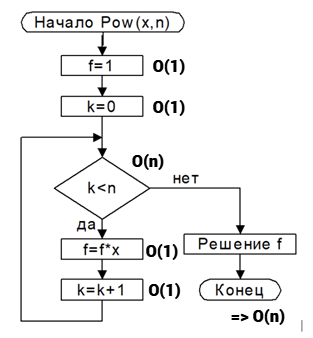
8) Возведение в степень

8.1. Классическое возведение в степень.

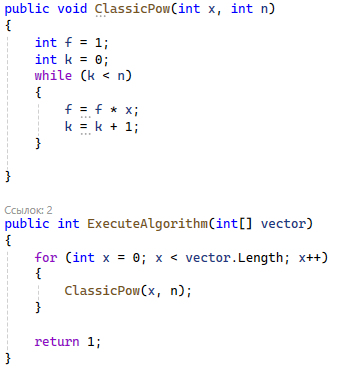
Алгоритм получает на вход основание и степень. С помощью цикла он вычисляет значение и выводит его.

Теоретическая сложность O(n) (рис.8.1.1)

Код алгоритма представлен на рисунке 8.1.2

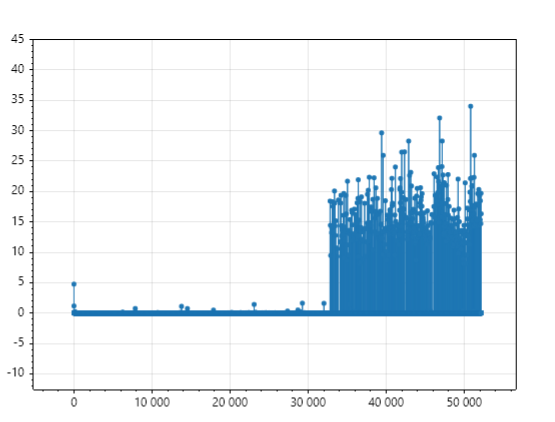


**рис.8.1.1 Классическое возведение, блок-схема и сложность**



**рис.8.1.2 Классическое возведение, код**

После обработки массива с данными программа вывела график (рис.8.1.3)



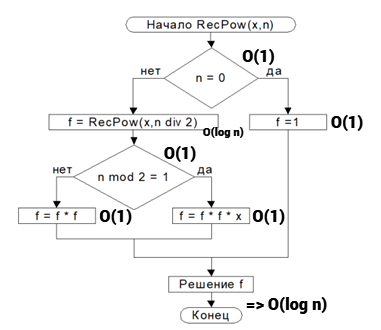
**рис.8.1.3 Классическое возведение, график**

8.2. Рекурсивное возведение в степень

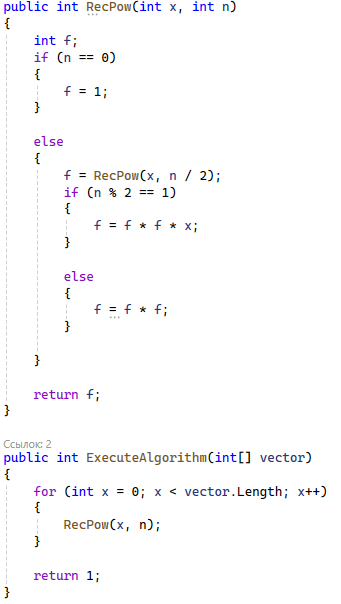
Алгоритм "ныряет" всё глубже, уменьшая степень вдвое на каждом шаге, пока не достигнет базового случая (n = 0). Затем, возвращаясь "назад", он комбинирует результаты подзадач, чтобы получить окончательный ответ.

Теоретическая сложность O(log n)

Код алгоритма представлен на рисунке 8.2.2

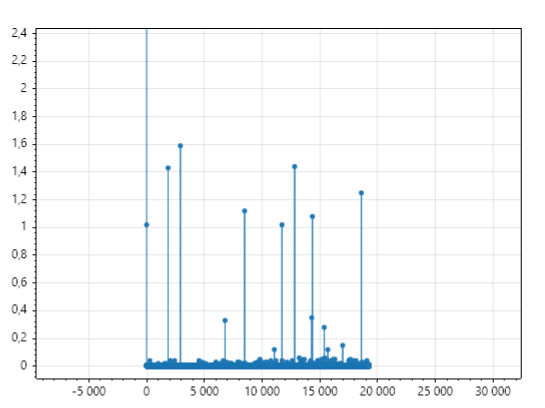


**рис.9.2.1 Рекурсивное возведение, блок-схема и сложность**



**рис.8.2.2 Рекурсиваное возведение, код**

После обработки массива с данными программа вывела график (рис.8.2.3)

****

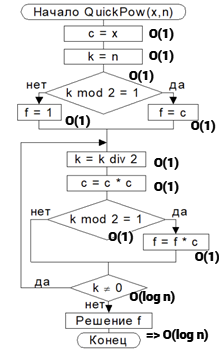
**рис.8.2.3 Рекурсивное возведение, график**

8.3. Быстрый алгоритм

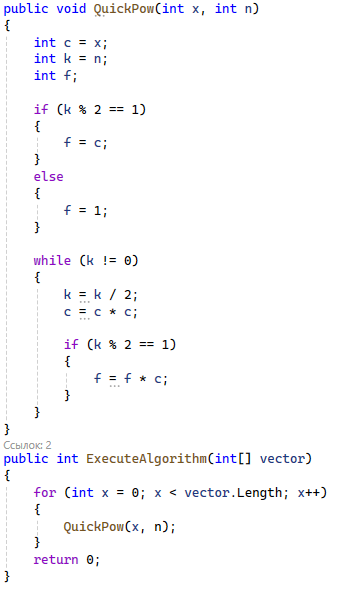
Алгоритм быстро возводит число в степень, сокращая количество умножений. Он делает это, деля степень пополам на каждом шаге и одновременно возводя основание в квадрат. Если степень нечётная, результат умножается на текущее основание. Этот процесс повторяется, пока степень не станет равна нулю, а результат будет содержать искомое значение

Теоретическая сложность O(log n)

Код алгоритма представлен на рисунке 8.3.2

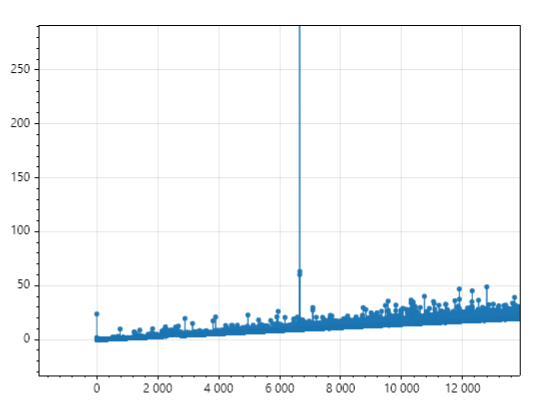


**рис.8.3.1 Быстрый алгоритм, блок-схема и сложность**



**рис.8.3.2 Быстрый алгоритм, код**

После обработки массива с данными программа вывела график (рис.8.3.3)



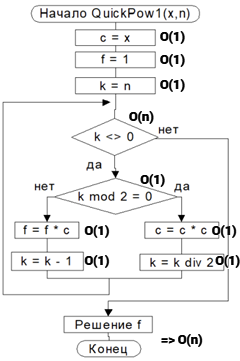
**рис.8.3.3 Быстрый алгоритм, график**

8.4. Классический быстрый алгоитм

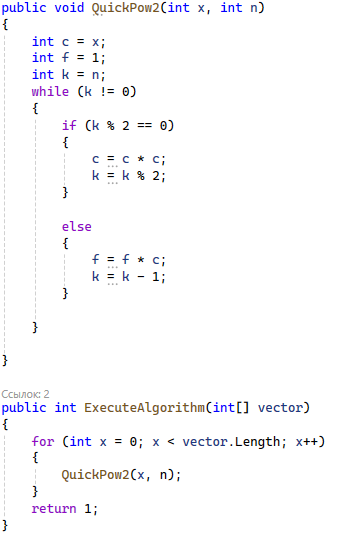
Алгоритм QuickPow1 быстро вычисляет степень числа, даже если она отрицательная. Он использует цикл, в котором на каждом шаге проверяет чётность степени. Если степень чётная, то основание возводится в квадрат, а степень делится на два. Если степень нечётная, то результат умножается на текущее основание, а степень уменьшается на единицу. Этот процесс повторяется до тех пор, пока степень не станет равной нулю.

Теоретическая сложность O(n)

Код алгоритма представлен на рисунке 8.4.2

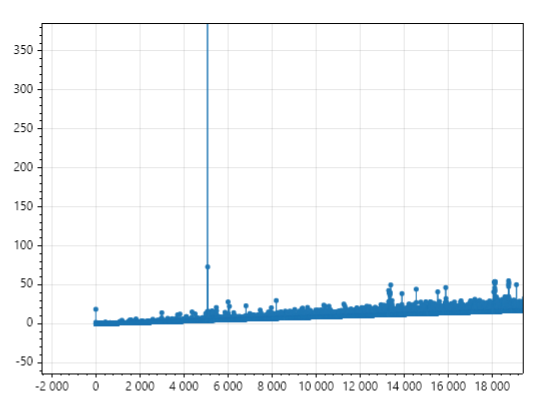


**рис.8.4.1 Классический быстрый алгоритм, блок-схемма и сложность**



**рис.8.4.2 Классический быстрый алгоритм, код**

После обработки массива с данными программа вывела график (рис.8.4.3)

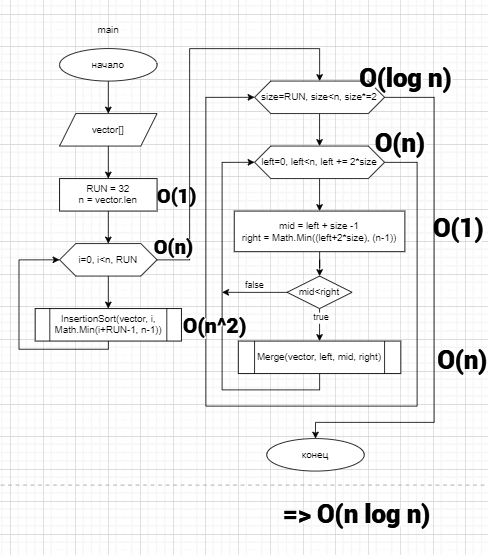


**рис.8.4.3 Классический быстрый алгоритм, график**

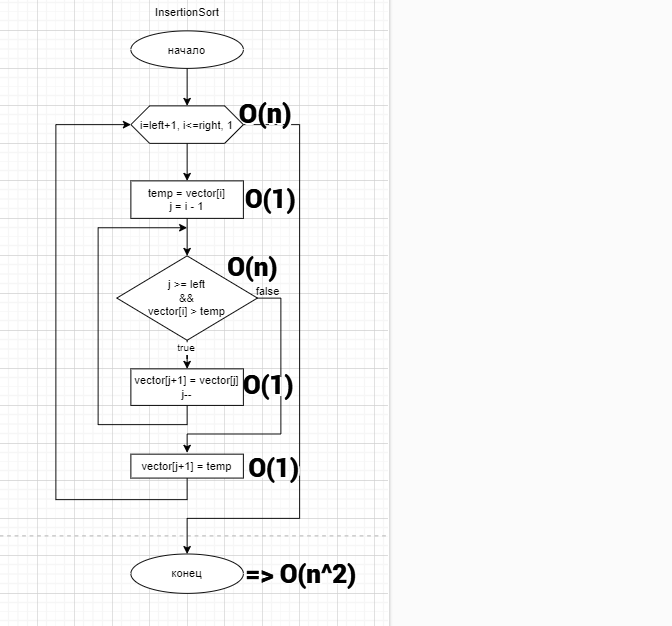
9) Алгоритм Тимсорт - это сортировочный алгоритм, который объединяет в себе принципы сортировки вставками и слиянием. Основная идея алгоритма Тимсорт состоит в том, чтобы сначала разделить список на относительно упорядоченные блоки (путем использования сортировки вставками (рис.9.2)), а затем объединить эти блоки с использованием сортировки слиянием (рис.9.3)

Теоретическая сложность O(n log n) (рис.9.1, рис.9.2, рис.9.3)

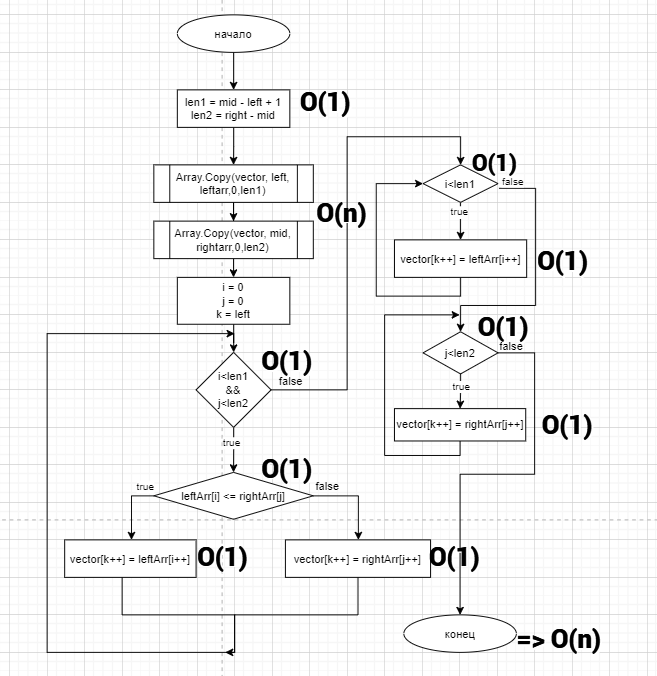
Код алгоритма представлен на рисунке 9.4



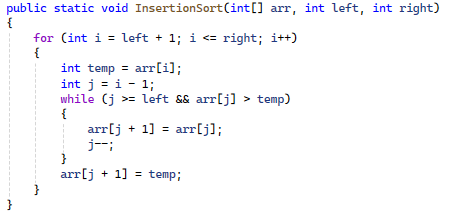
**рис.11.1 TimSort Main, блок-схема и сложность**

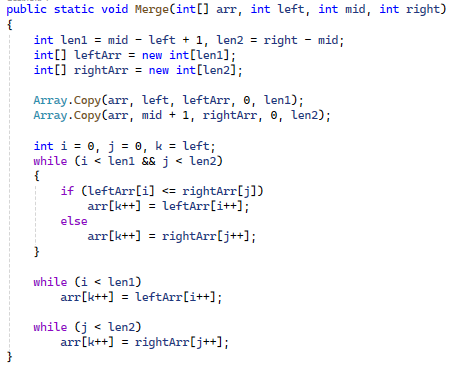


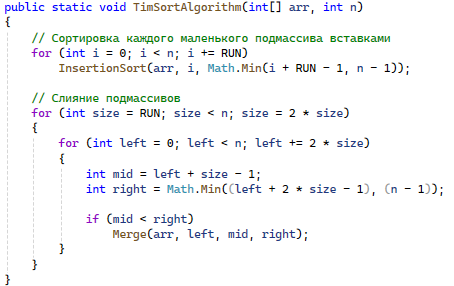
**рис.9.2 TimSort InsertionSort, блок-схема и сложность**



**рис.9.3 TimSort Merge, блок-схема и сложность**

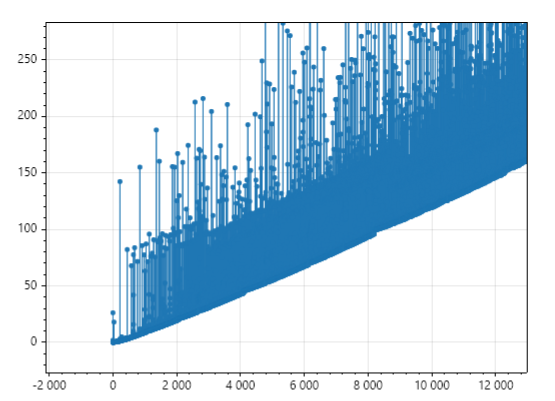






**рис.9.4 TimSort, код**

После обработки массива с данными программа вывела график (рис.9.5)



**рис.9.6 TimSort, график**

Блок 2

В данном блоке требуется перемножить две квадратные матрицы размера n.

Чтобы перемножить квадратные матрицы, нужно **умножить элементы в строках первой матрицы на элементы в столбцах второй матрицы и сложить полученные значения**.

Умножать можно только **согласованные матрицы**, где количество столбцов первой матрицы равно количеству строк второй.

**Алгоритм умножения:**

**1)** Перемножаем числовые значения первой строки на значения из первого столбца:

1.1) умножаем первый элемент первой строки на соответствующий элемент из первого столбца;

1.2) находим произведение второго элемента первой строки и второго элемента, который берём из столбца №1.

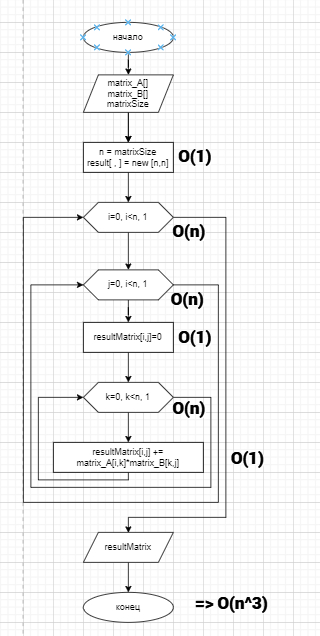
2) Проделываем такие же действия со всеми элементами, пока не дойдём до конца первой строки матрицы.

3) Вычисленные произведения необходимо сложить между собой. Вычисленный результат будет равняться элементу для первой строки.

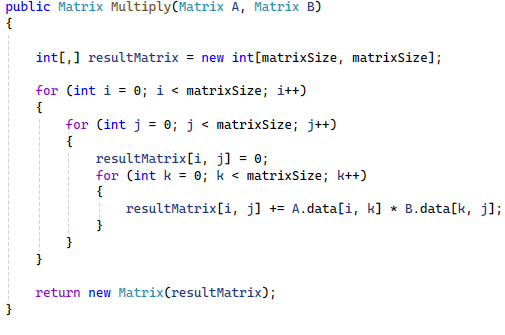
4) Аналогичные действия нужно проводить с каждой строкой вычисляемой матрицы. Вычисления проводятся до тех пор, пока все строчки новой матрицы не будут заполнены значениями.

Теоретическая сложность O(n^3) (рис.10.1)

Код алгоритма представлен на рисунке 10.4

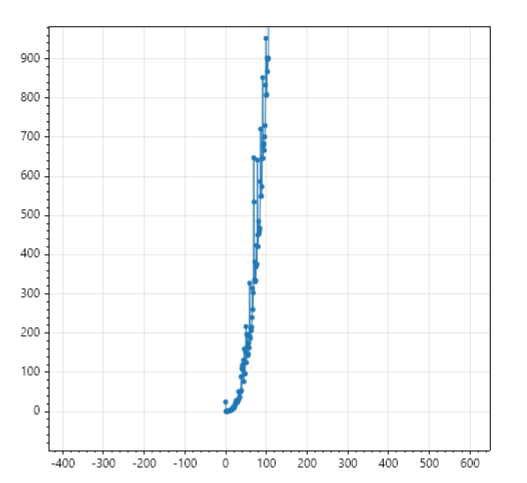


**рис.12.1 Матричное произведение, блок-схема и сложность**



**рис.10.2 Матричное произведение, код**

После обработки массива с данными программа вывела график (рис.10.3)

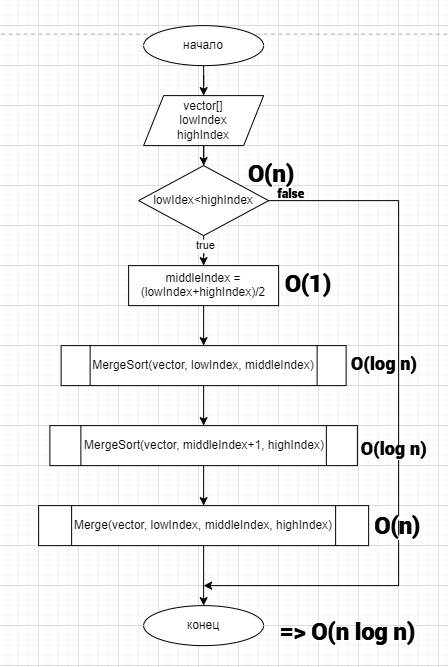


**рис.10.4 Матричное произведение, график**

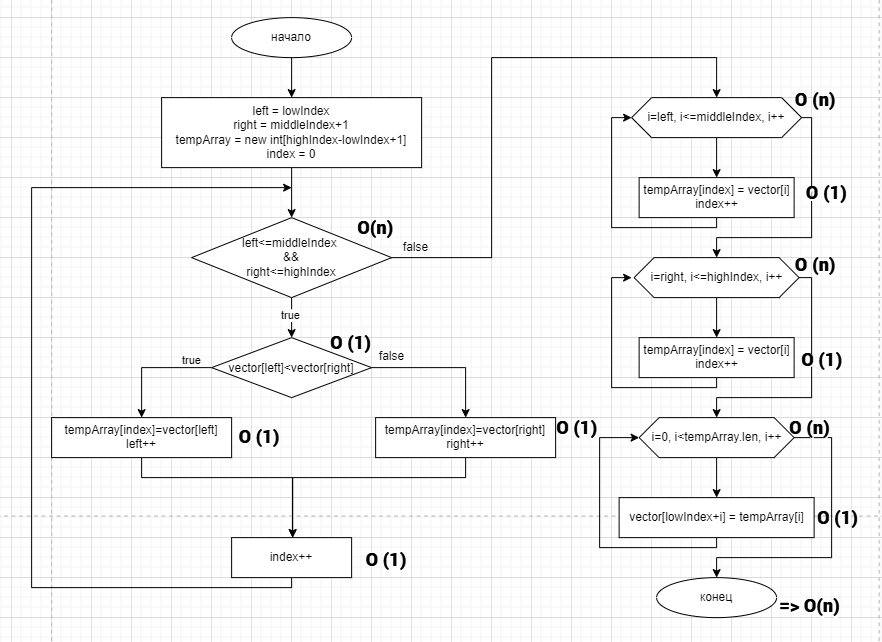
Блок 3

1) Алгоритм слияния — это алгоритм, используемый для объединения двух отсортированных массивов в один отсортированный массив. Он использует два указателя, по одному на каждый массив, сравнивает элементы, на которые указывают указатели, и добавляет меньший элемент в новый массив. Указатель на меньший элемент перемещается, и процесс повторяется до тех пор, пока не будут обработаны все элементы обоих массивов.

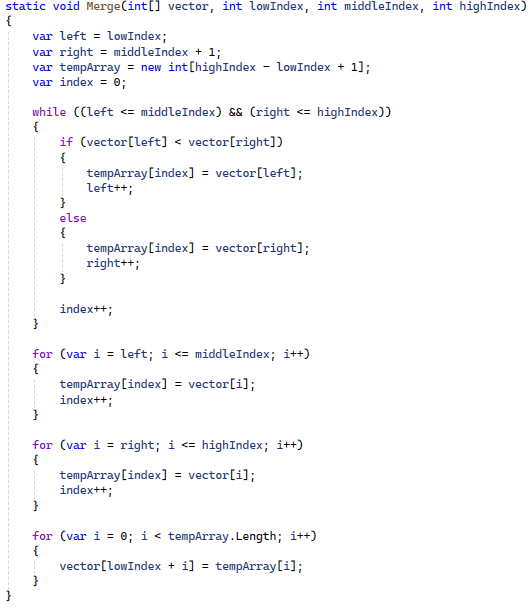
Теоретическая сложность O(n log n) (рис.11.1, рис.11.2)

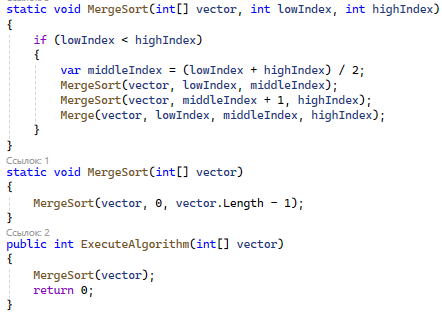
Код алгоритма представлен на рисунке 11.3

**рис.13.1 Алгоритм слияния MergeSort, блок-схема и сложность**



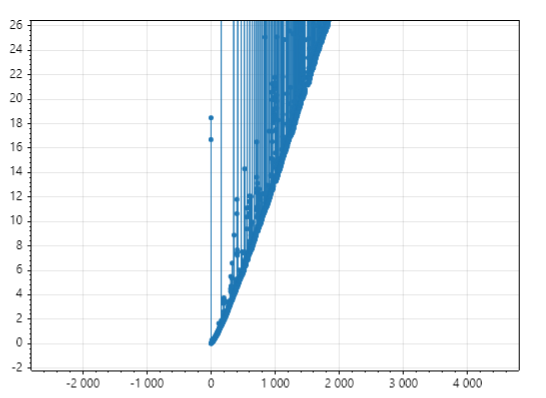
**рис.11.2 Алгоритм слияния Merge, блок-схема и сложность**





**рис.11.3 Алгоритм слияние, код**

После обработки массива с данными программа вывела график (рис.11.4)

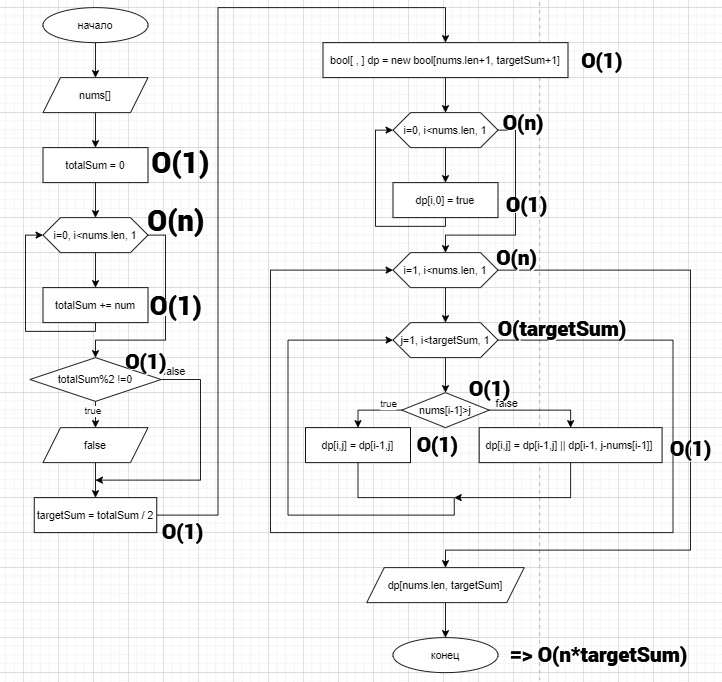


**рис.11.4 Алгоритм слияние, график**

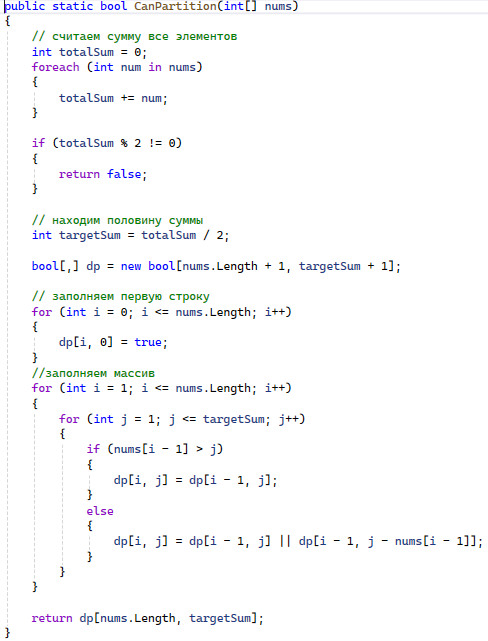
2) Алгоритм разбиения множества на два подмножества с равной суммой проверяет возможность разделения множества на два подмножества с одинаковой суммой элементов. Он использует динамическое программирование, чтобы проверить все возможные суммы, которые можно получить из подмножеств, и определяет, достижима ли половина общей суммы элементов. Если половина суммы достижима, значит, множество можно разделить на два подмножества с равными суммами.

Теоретическая сложность O(n\*targetSum) (рис.12.1)

Код алгоритма представлен на рисунке 12.2

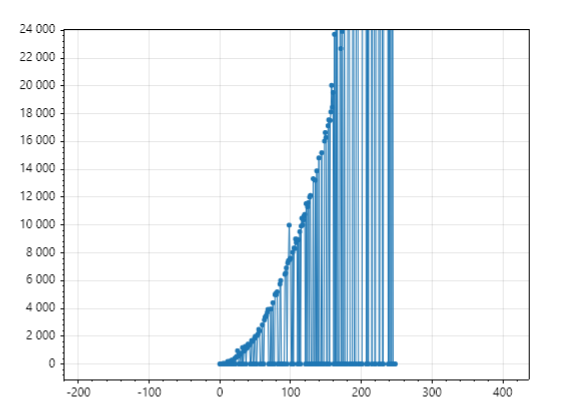


**рис.14.1 Разбиение множества, блок-схема и сложность**



**рис.12.2 Разбиение множества, код**

После обработки массива с данными программа вывела график (рис.12.3)

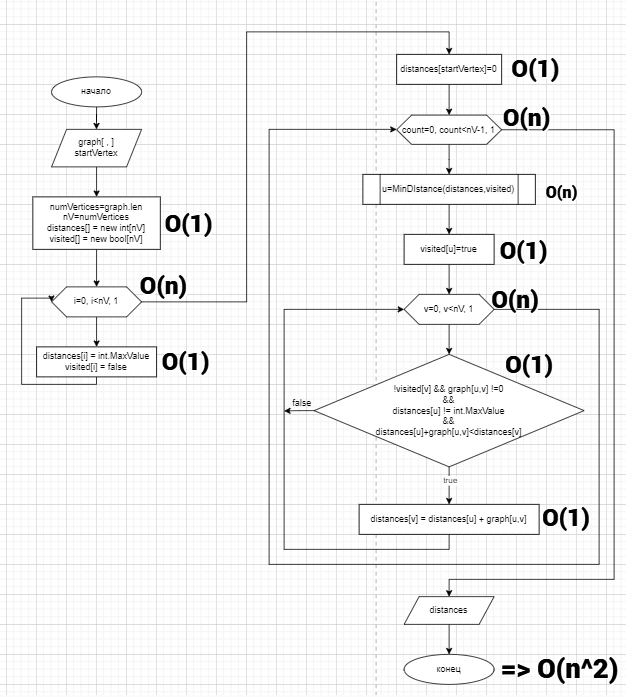


**рис.12.3 Разбиение множества, график**

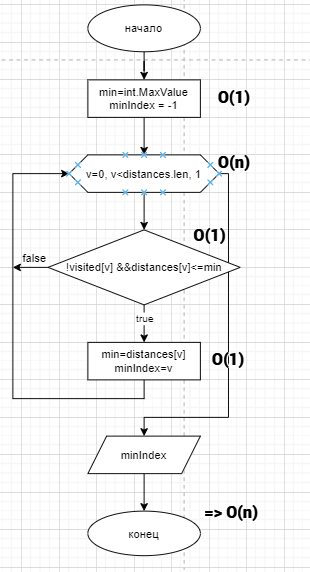
3) Алгоритм Дейкстры находит кратчайшие пути от одной вершины графа до всех остальных. Он постепенно исследует граф, начиная с начальной вершины, и обновляет расстояния до соседних вершин, если находит более короткий путь. Алгоритм завершается, когда посещены все вершины или найдена целевая вершина, гарантируя при этом нахождение кратчайших путей для графов с неотрицательными весами ребер.

Теоретическая сложность O() (рис.13.1, рис.13.2)

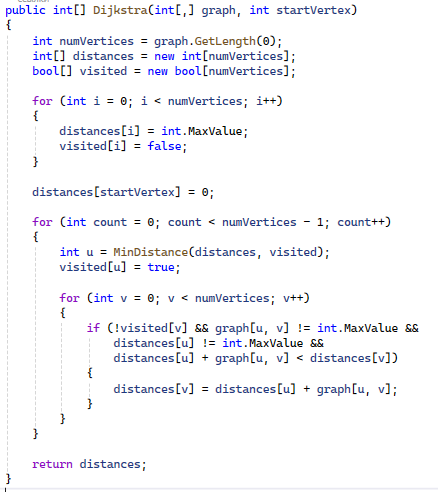
Код алгоритма представлен на рисунке 13.3

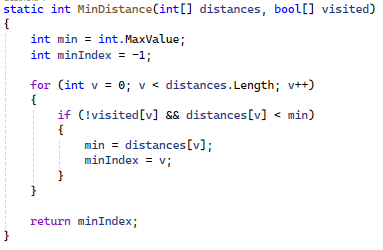


**рис.15.1 Дейкстра Main, блок-схема и сложность**



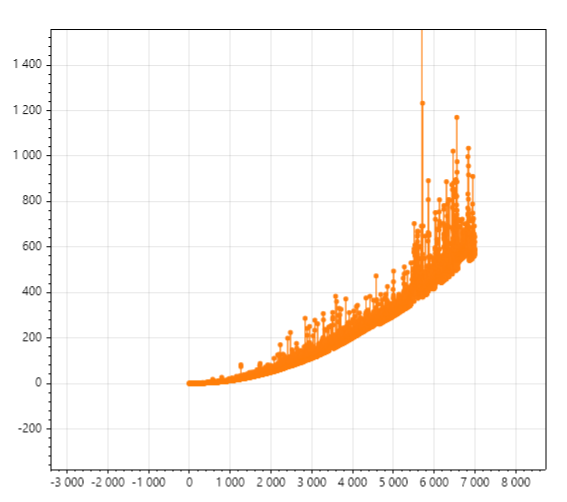
**рис.13.2 Дейкстра minDistance блок-схема и сложность**





**рис.13.3 Дейкстра, код**

После обработки массива с данными программа вывела график (рис.13.4)



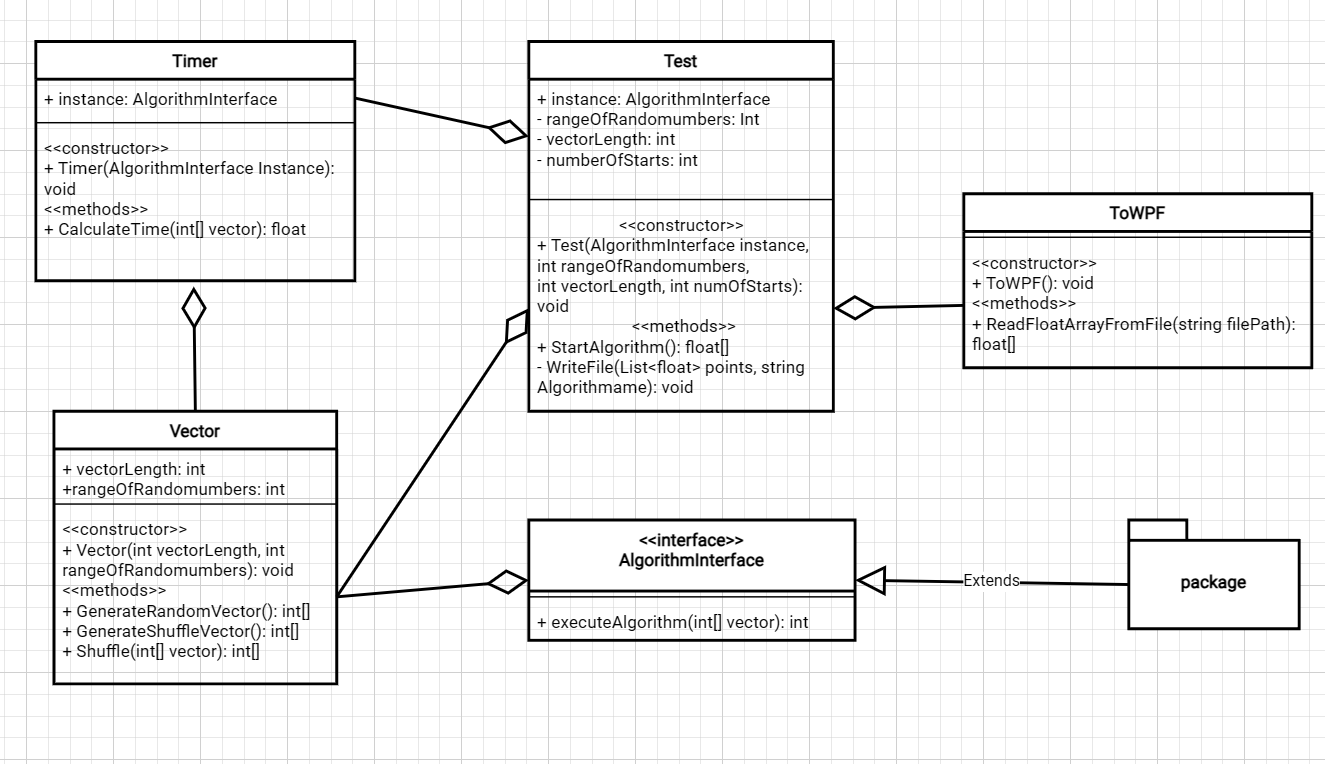
**рис.13.4 Дейкстра, график**

Модель проекта

Все приложение работает на языке C#, для содания внешней оболочки использован WPF, для отрисовки графиков – ScottPlot.

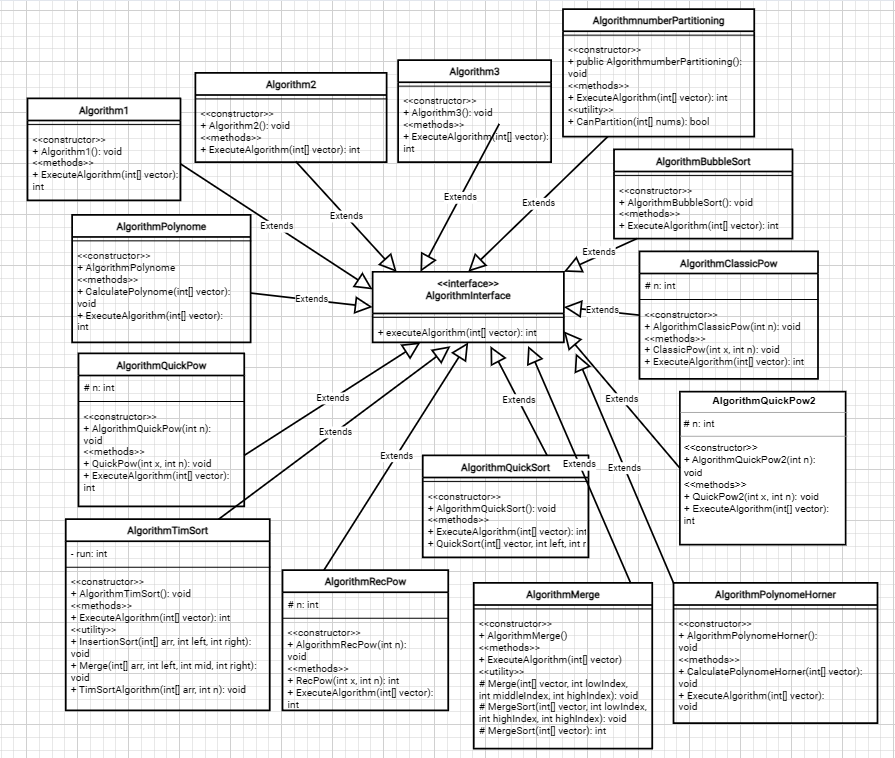
Приложение поделено на 2 части – фронтенд и бэкенд. В основе бэкенда лежит три созданных библиотеки:

1) AlgLogic – необходима для работы с алгоритмами которым на вход подается вектор (рис.14)



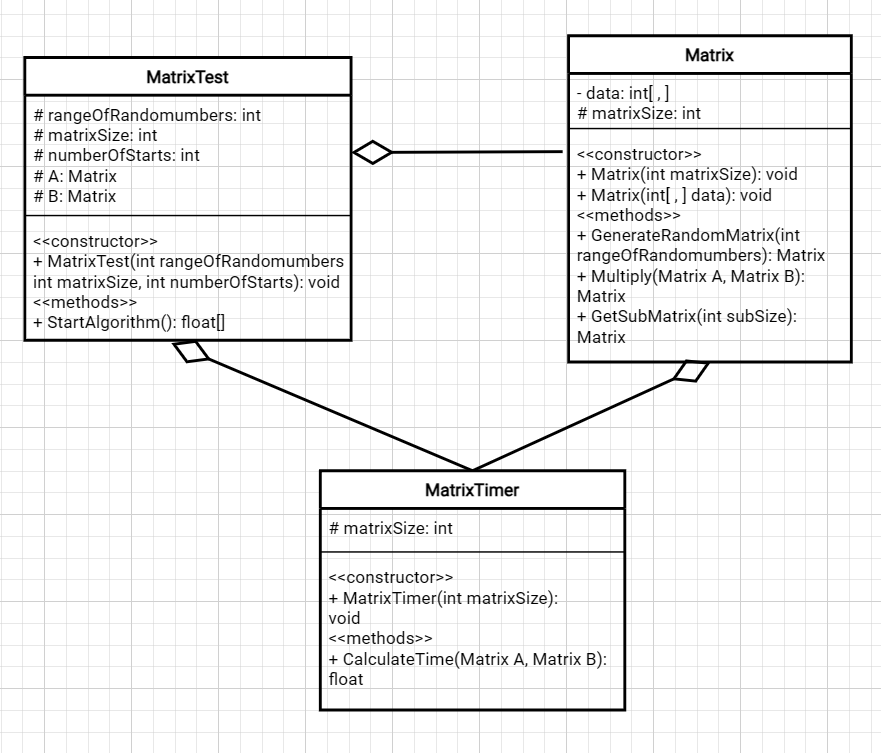
**рис.16 UML AlgLogic**

Здесь под пакетом имеются все классы-наследники интерфейса. (рис.15)



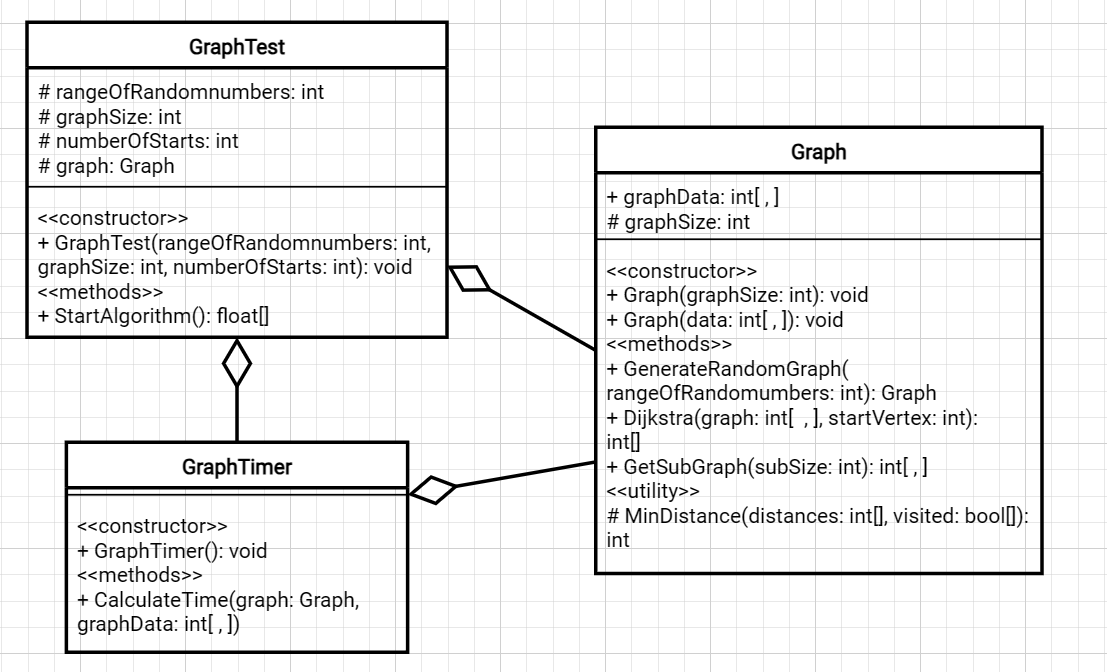
**рис.17 UML интерфейсы**

2) MatrixEntities – необходима для выполнения операций над матрицами (рис.16)



**рис.18 UML MatrixExtends**

3) DijkstraAlgorithm – необходима для работы с графами и выполнения алгоритма Дейкстры. (рис.17)



**рис.19 UML DijkstraAlgoritm**

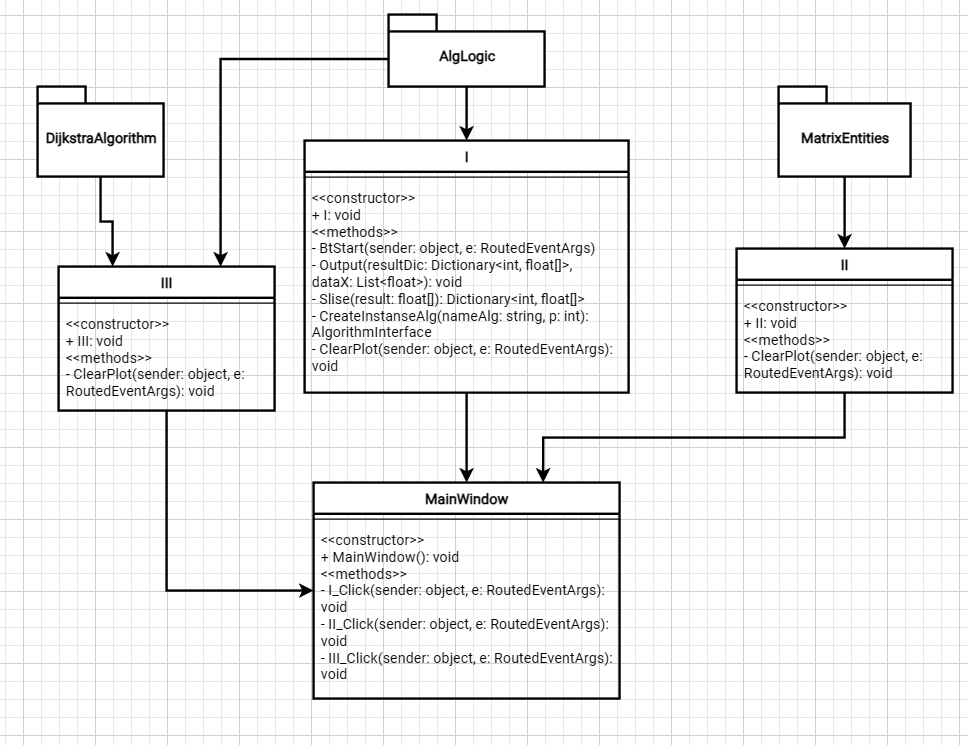
Фронтенд состоит из одной библиотеки (рис.18) в которой содержатся четыре файла:

1) MainWindow.xaml + MainWindow.cs – эти классы отвечают за начальный экран приложения, на котором расположены три кнопки отвечающие за переходы в следующие части программы

2) I.xaml + I.cs – отвечает за вывод результатов первого блока лабораторной работы

3) II.xaml + II.cs – отвечает за вывод результатов второго блока лабораторной работы (матрицы)

4) III.xaml + III.cs – отвечает за вывод результатов третьего блока лабораторной работы (собственные алгоритмы)



**рис.20 UML фронтенд**