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

Авторы отчета А.В. Сакулин ПрИ-202

подпись инициалы, фамилия группа

Д.А. Захаров ПрИ-202

подпись инициалы, фамилия группа

Д.А. Луценко ПрИ-202

подпись инициалы, фамилия группа

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

дата оценка

Челябинск 2024 г.

**Цель работы:** Эмпирический анализ временной сложности алгоритмов. Вводиться понятие временной сложности алгоритма, рассматривается математический аппарат для оценки временной сложности и правила применения этого аппарата.

**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.

Лабораторную выполняли с применением полиморфизма, т.е. у нас есть чертёж базового класса, который наследуют другие классы с соблюдений условий заданых базовым классом.  
  
Стек технологий:   
1. C#  
2. WPF

3. OxyPlot

4. Accord.Math

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

2. WPF позволяет работать с большими объемами данных. Имеет множество полезных библиотек, которые упрощают реализацию различных проектов. Достаточно просто работать с графиками, имея необходимые для их построения данные.

3. Библиотека OxyPlot. Использовали для построения и визуализации графиков.

4. Accord.Math — использовался для аппроксимации входных данных

**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: 5

Замеры времени от количества элементов(смотри рис 1.1.1):

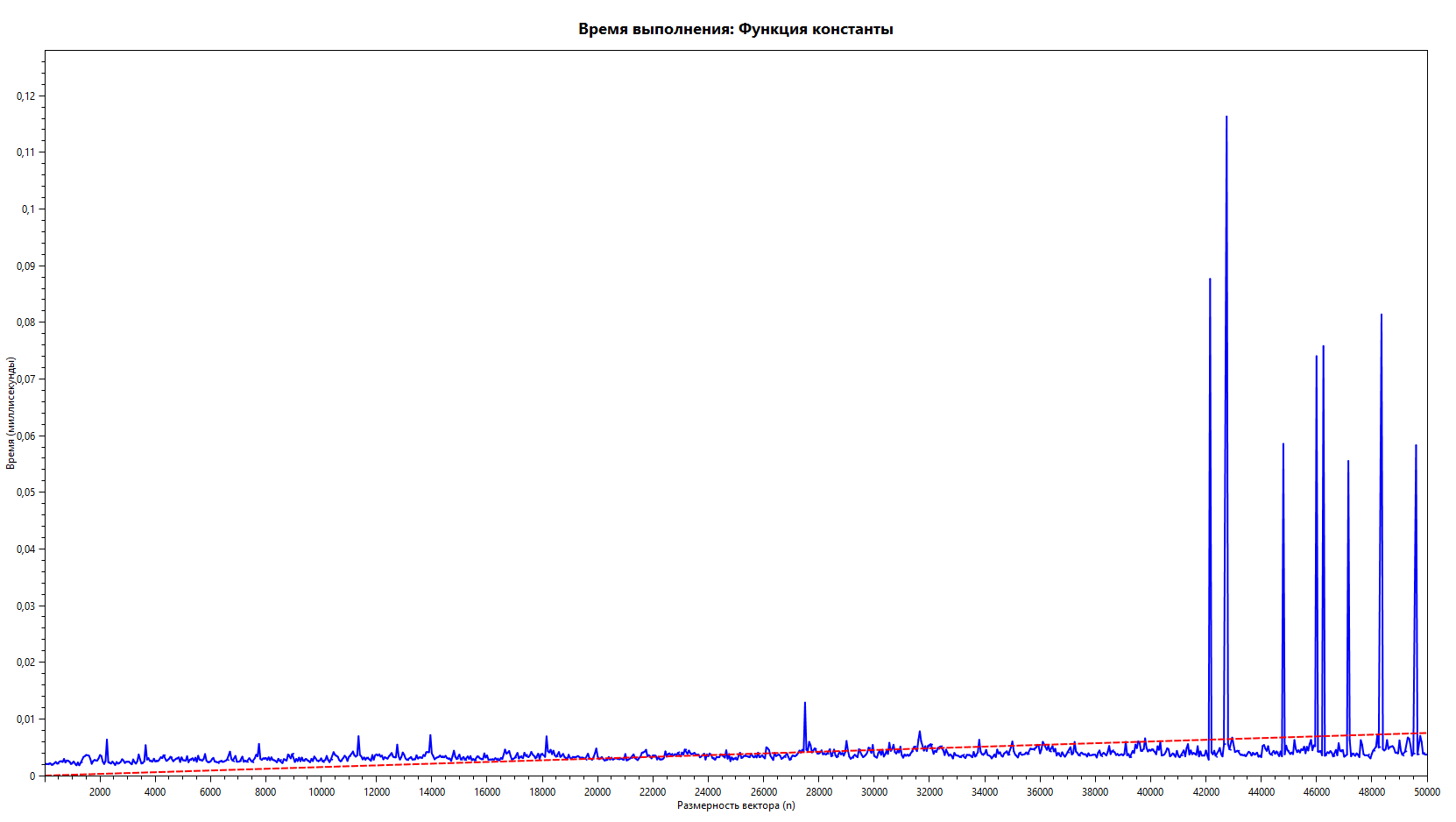


Рис. 1.1.1 Const алгоритм

Код изображен на Рис. 1.1.2:

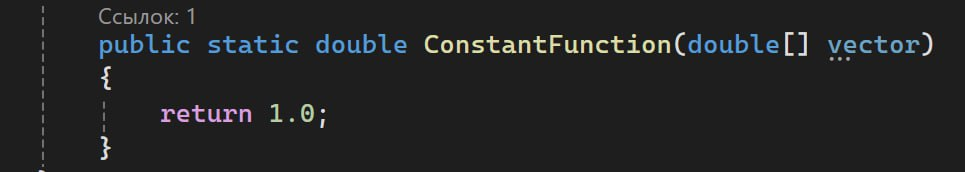


Рис. 1.1.2 Код алгоритма

1. F(v) = (сумма элементов)

* Временная сложность: O(n)
* N=50000
* Ср. знач. на основе тестов: 5

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

получилась линейная зависимость времени от количества входных данных:

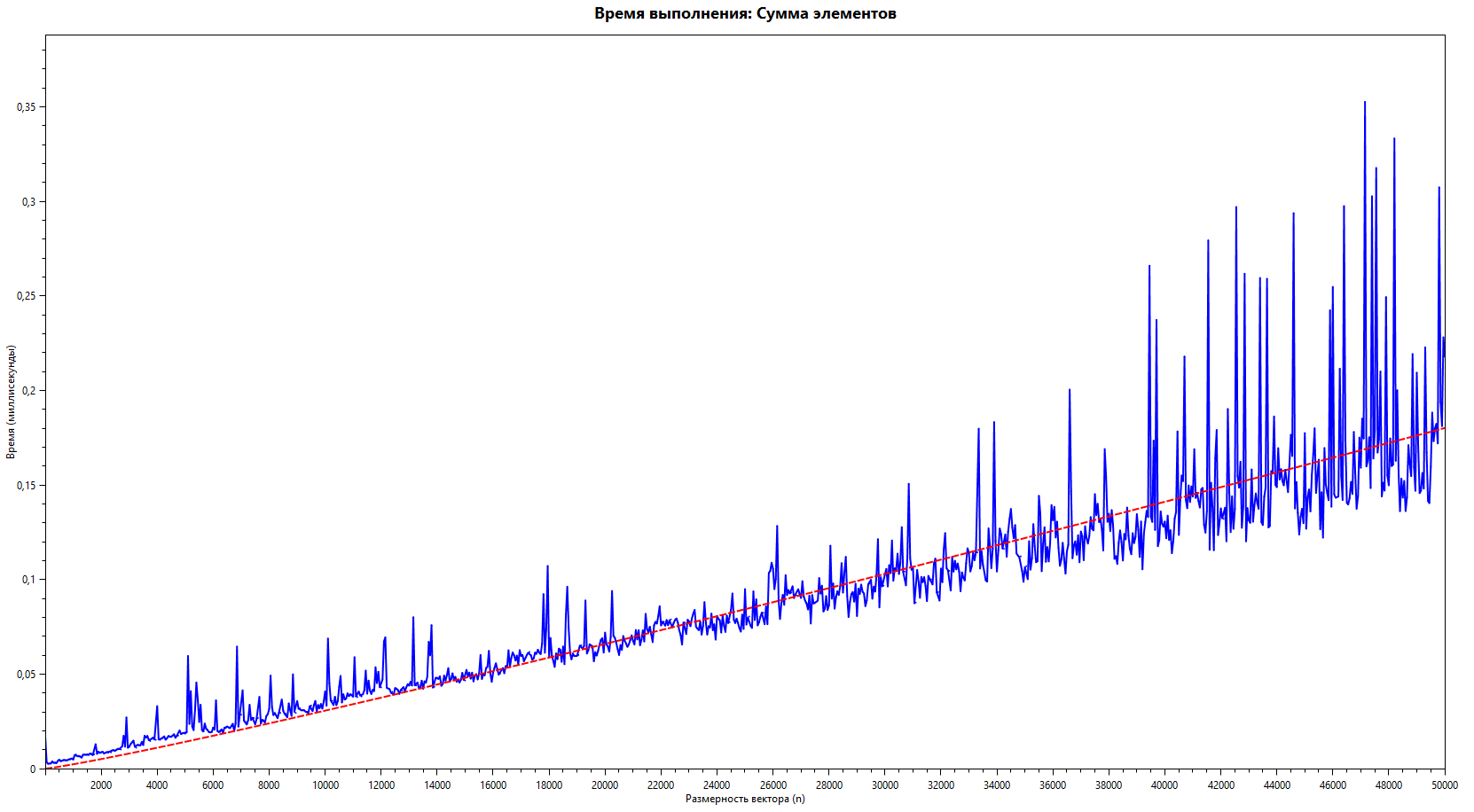


Рис. 1.2.1 Алгоритм суммы элементов

Код изображен на Рис. 1.2.2

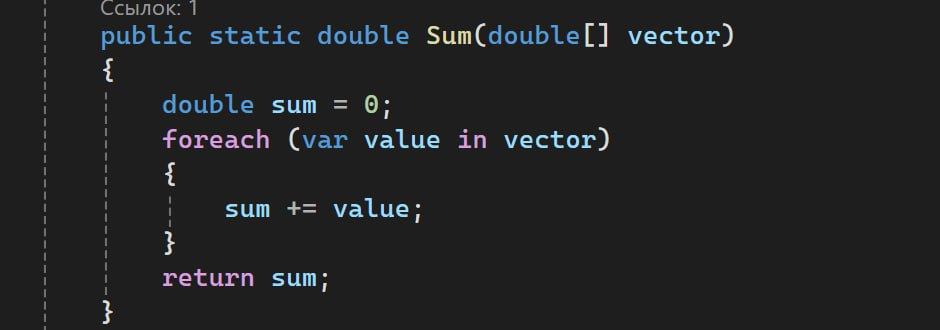


Рис. 1.2.2 Код алгоритма

1. (произведение элементов)

* Временная сложность: O(n)
* N=50000
* Ср. знач. на основе тестов: 5

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:

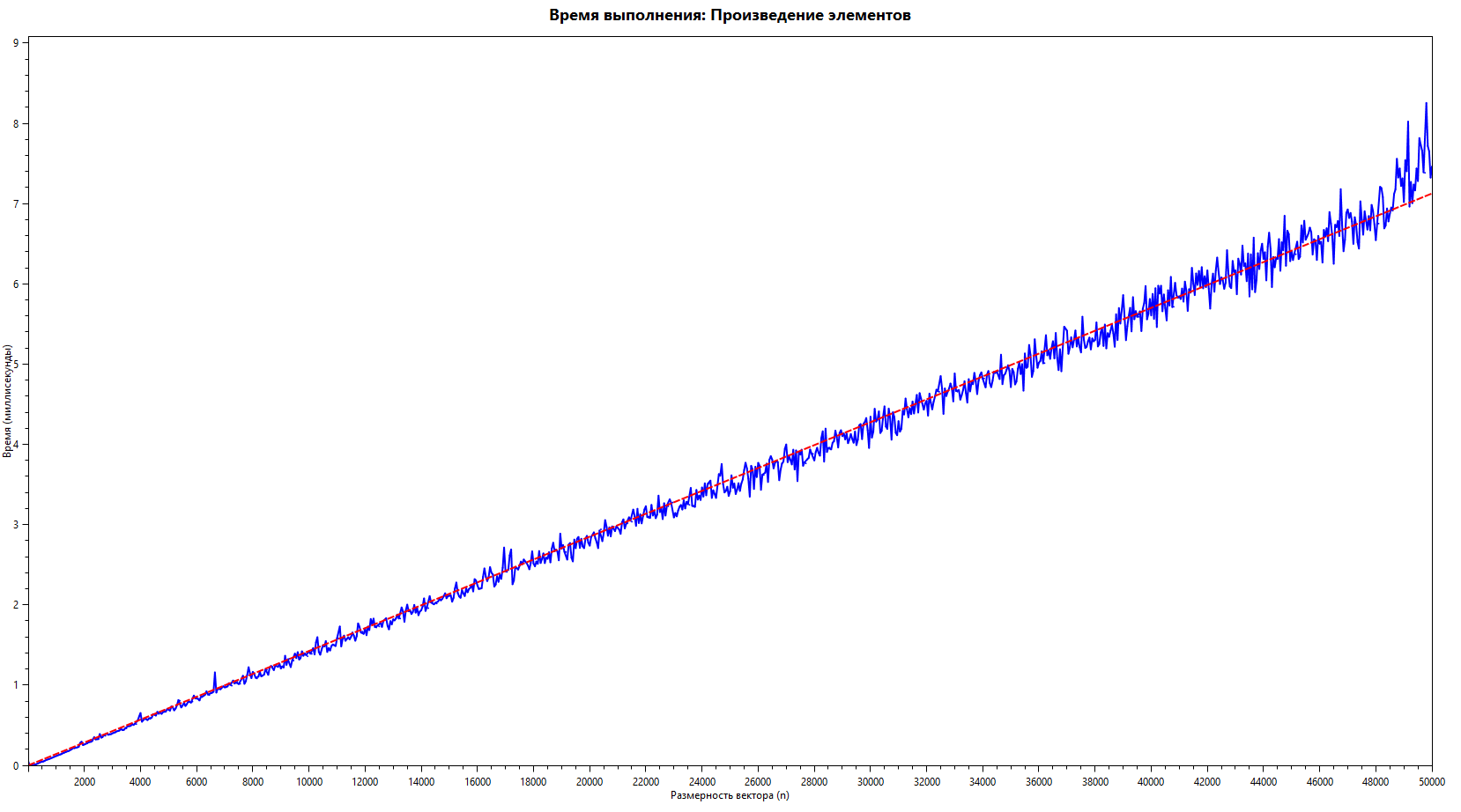


Рис. 1.3.1 Алгоритм произведения элементов

Код изображен на Рис. 1.3.2

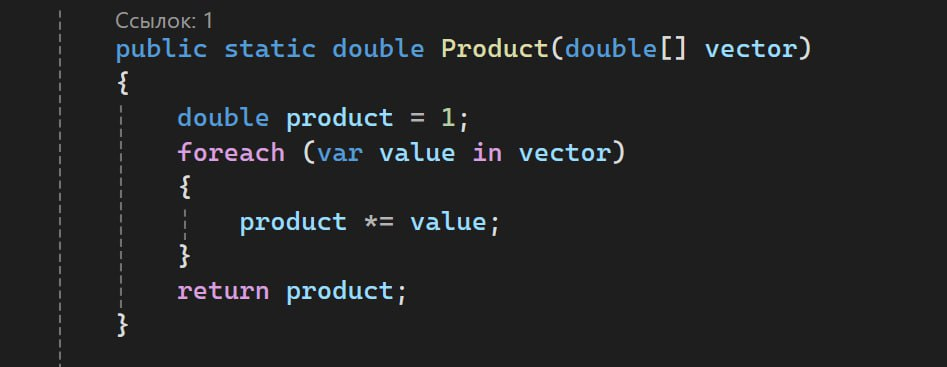


Рис. 1.3.2 Код алгоритма

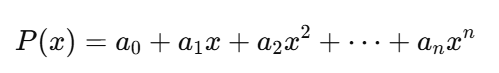
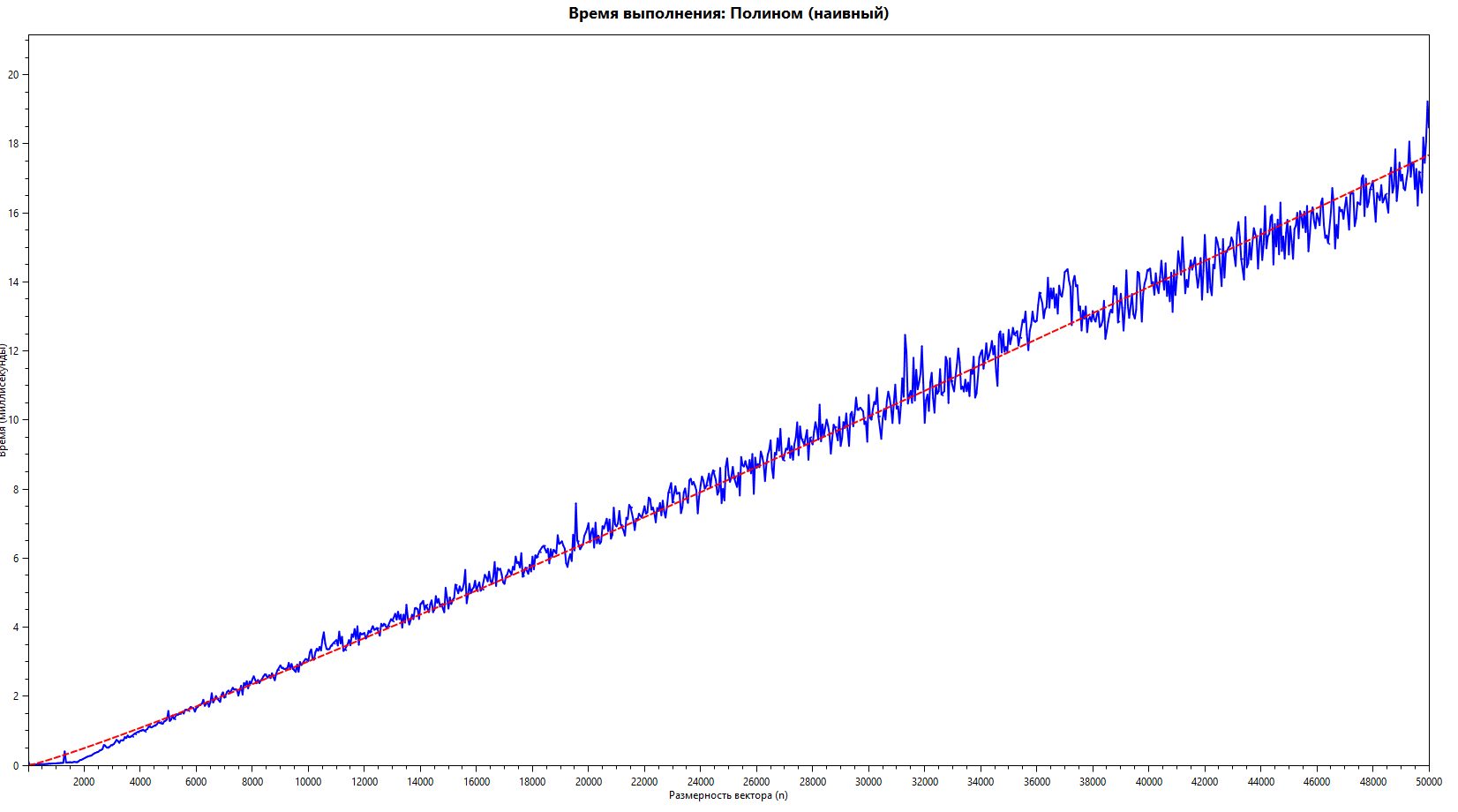
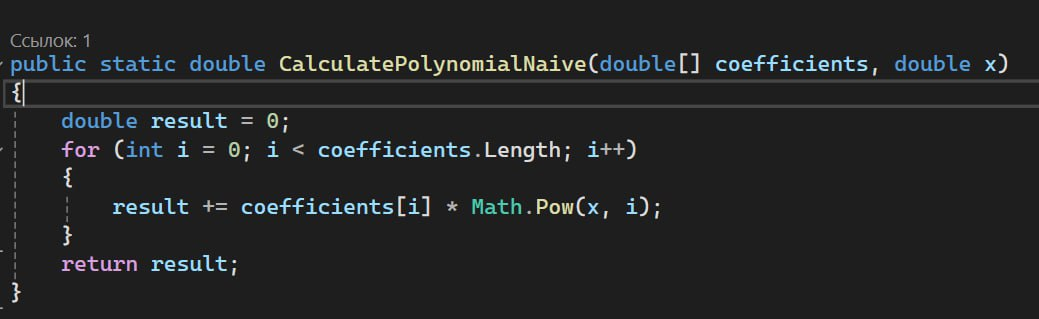
* (1) Наивный полином  
  Этот алгоритм реализует классический способ вычисления полинома вида: 
* Для каждого коэффициента ai он вычисляет x^i, используя функцию Math.Pow, и умножает результат на соответствующий коэффициент.
* Затем все результаты складываются, и итоговое значение возвращается.   
    
  Для полиномов большой степени этот алгоритм может оказаться медленным из-за необходимости повторного возведения в степень.
* Временная Сложность:
* T = O(n)
* N= 50000
* Ср. значение на основе тестов = 100

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

* **Наивный алгоритм** использует n сложений и до n(n+1)/2 операций возведения в степень, что увеличивает его временную сложность до O(n), если учитывать стоимость возведения в степень.
* Код изображен на Рис. 1.4.1.2
* 

(2) Полином Горнера :

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

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

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

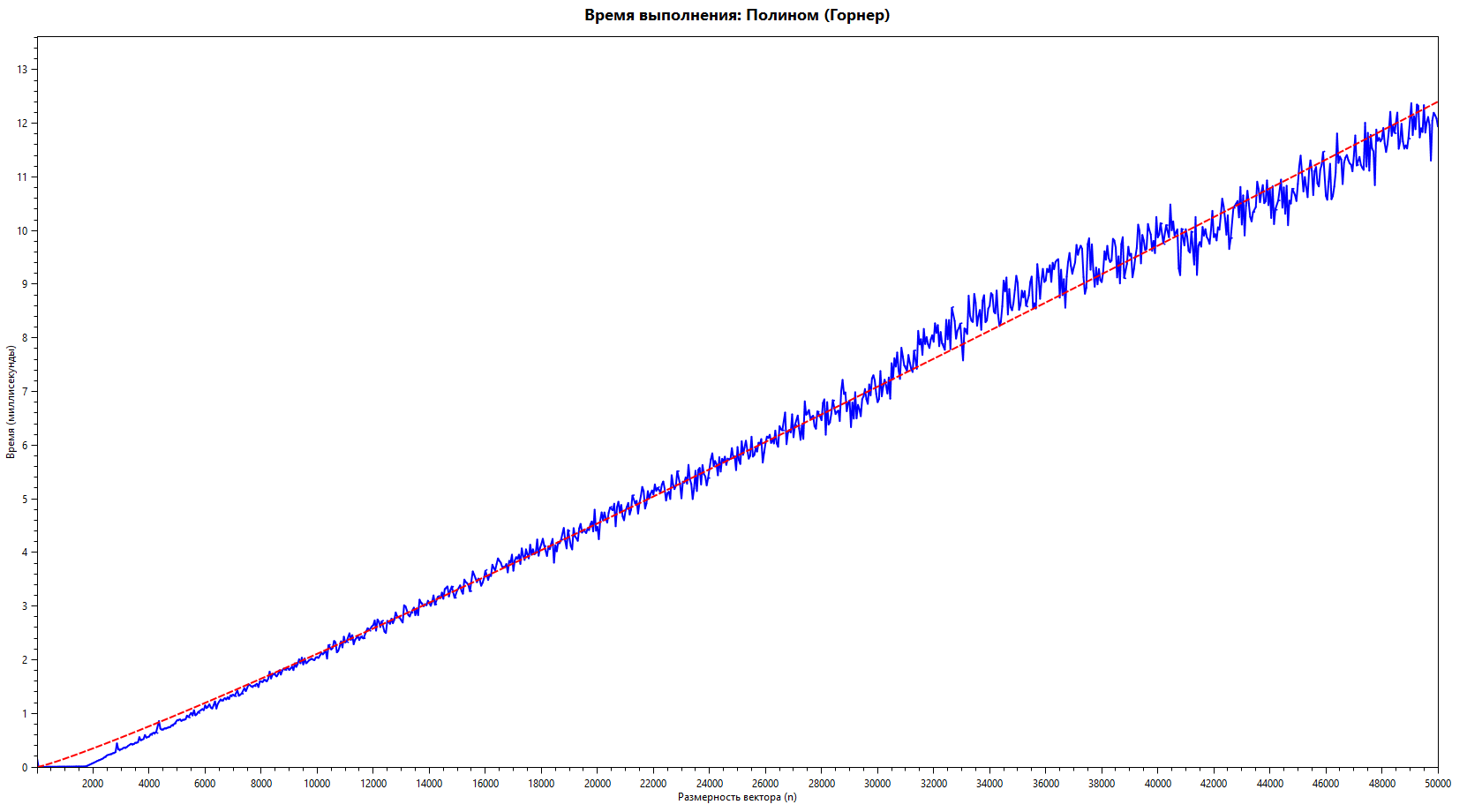


Рис. 1.4.1.1 Алгоритм метода Наивный Полином

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

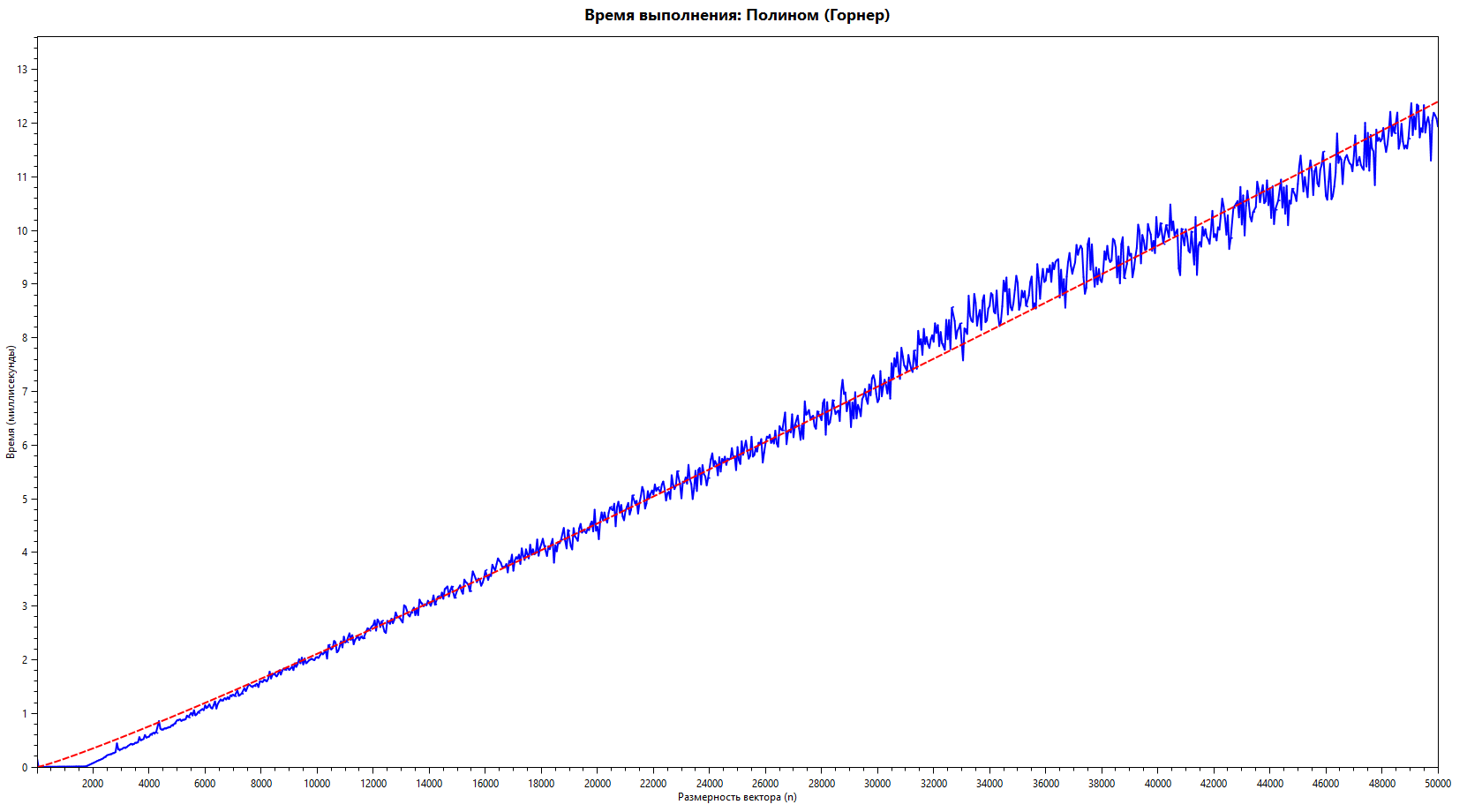


Рис. 1.4.2.1 Алгоритм метода Горнера.

Код изображен на Рис. 1.4.2.2

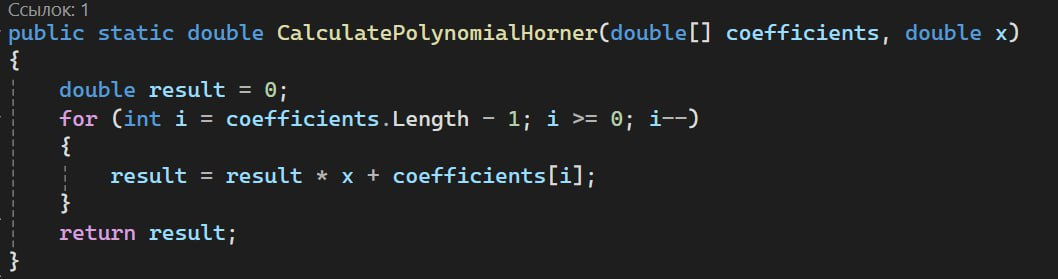


Рис. 1.4.2.2 Код метода Горнера.

Проанализировав графики (см. Рис. 1.4.1.1 и Рис. 1.4.2.1) можно сделать вывод, что метод Горнера работает в несколько 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: 5

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):

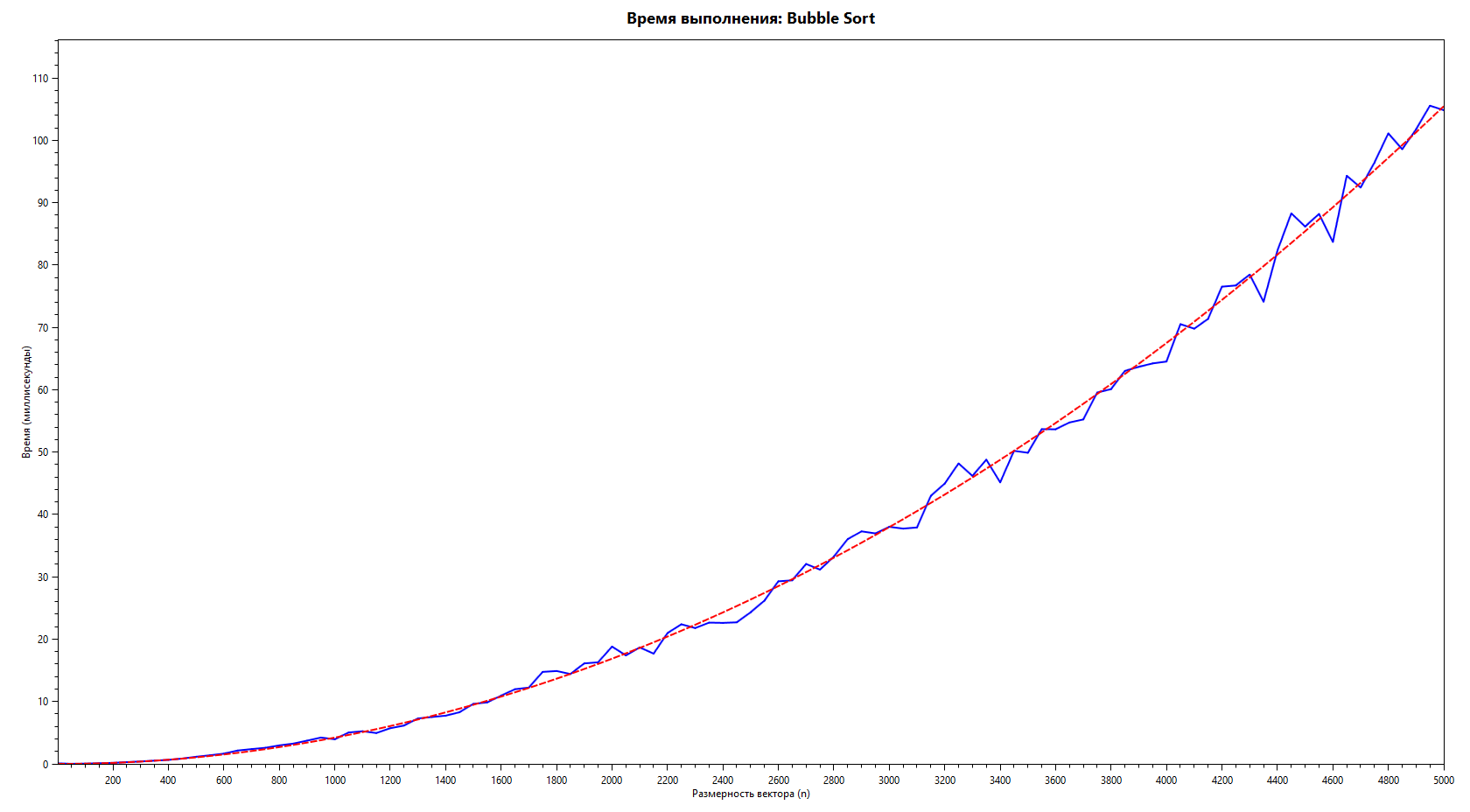


Рис. 5.1 Алгоритм сортировки пузырьком

Код изображен на Рис. 1.5.2

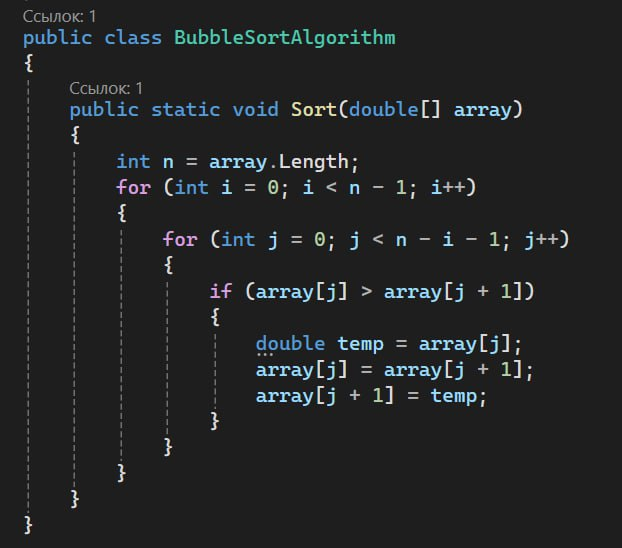


Рис. 1.5.2 Код сортировки пузырьком

1. Алгоритм быстрой сортировки (Quick sort) элементов

* Временная сложность: O(N \* log(N))
* N = 100000
* Ср. знач. на основе тестов: 5

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):

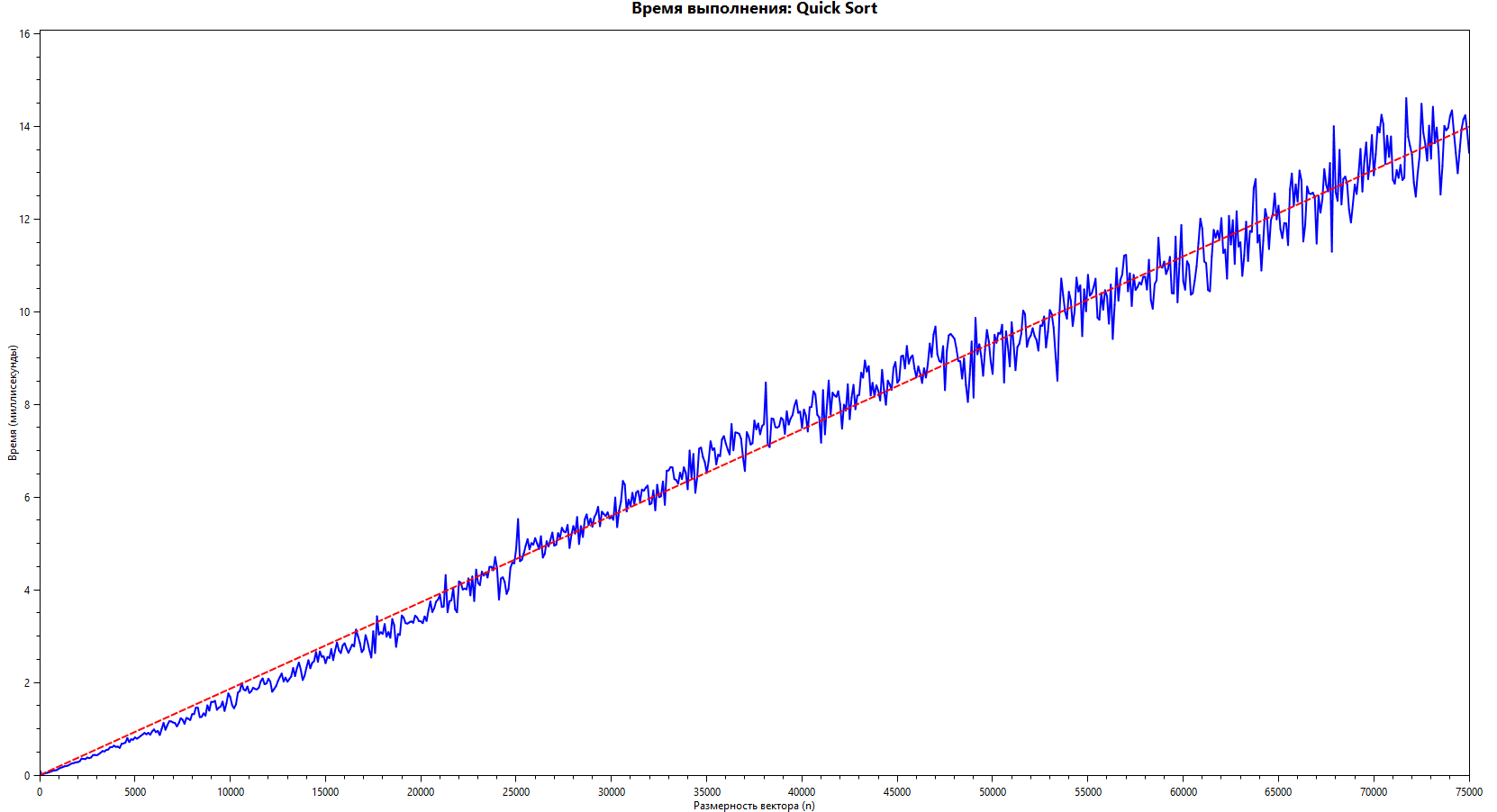


Рис. 1.6.1 Алгоритм быстрой сортировки

Код изображен на Рис. 1.6.2:

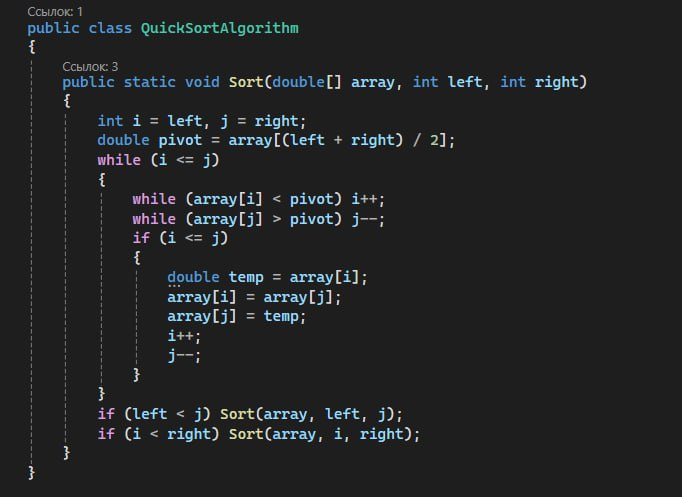


Рис 1.6.2 Код быстрой сортировки

1. Гибридный алгоритм сортировки Timsort элементов

Временная сложность: O(N \* log(N))

* N = 50000
* Ср. знач. на основе тестов: 5

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

Алгоритм построен на идеи, что сортируемые массивы данных часто содержат в себе упорядоченные подмассивы. Timsort разбивает входной массив на подмассивы (“RUN’ы”) и сортирует их, используя сортировку вставками. После сортировки вставками объединяет подмассивы попарно сортировкой слиянием и возвращает отсортированный массив.

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

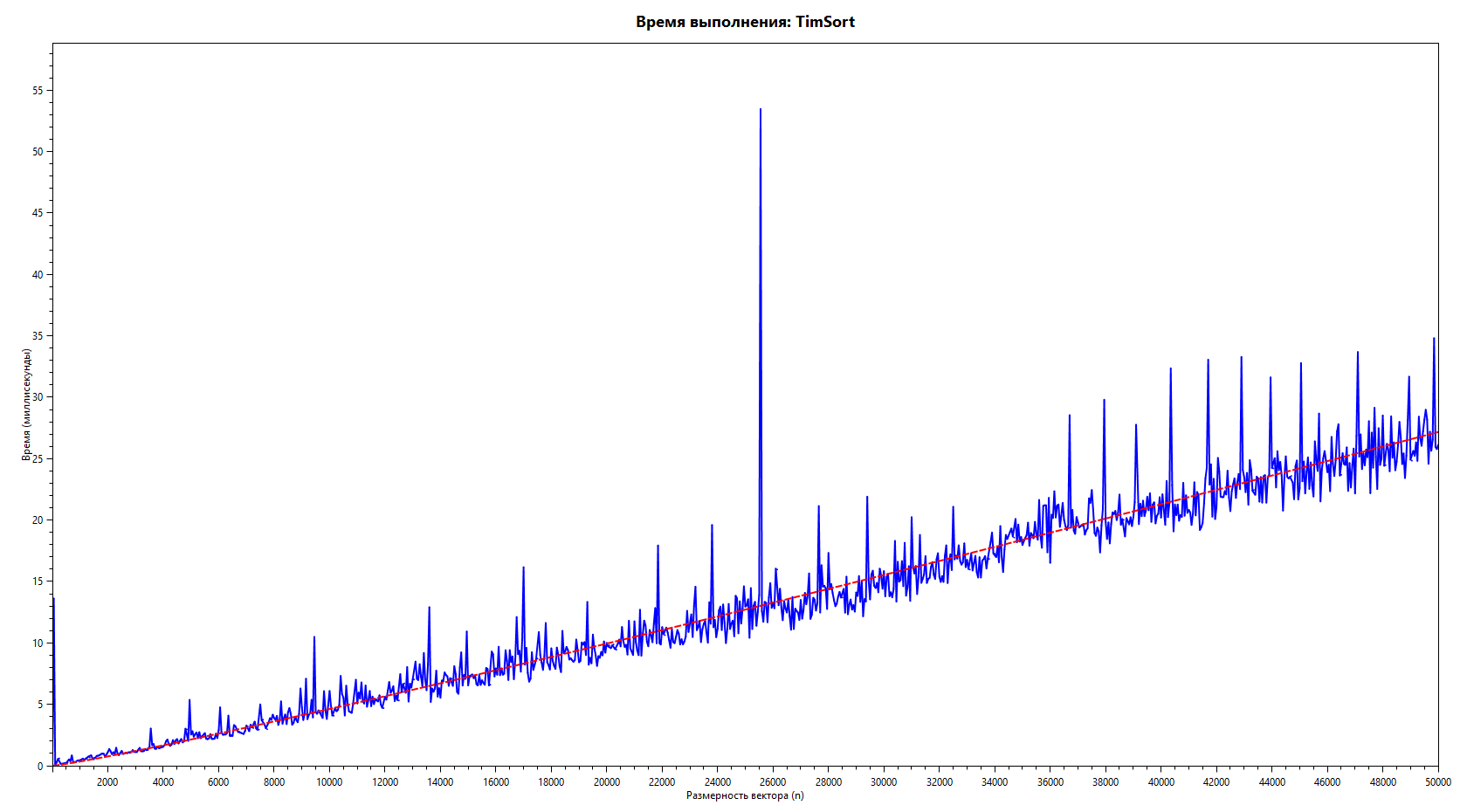


Рис. 1.7.1 Алгоритм гибридной сортировки

Код изображен на Рис. 1.7.2-1.7.8:

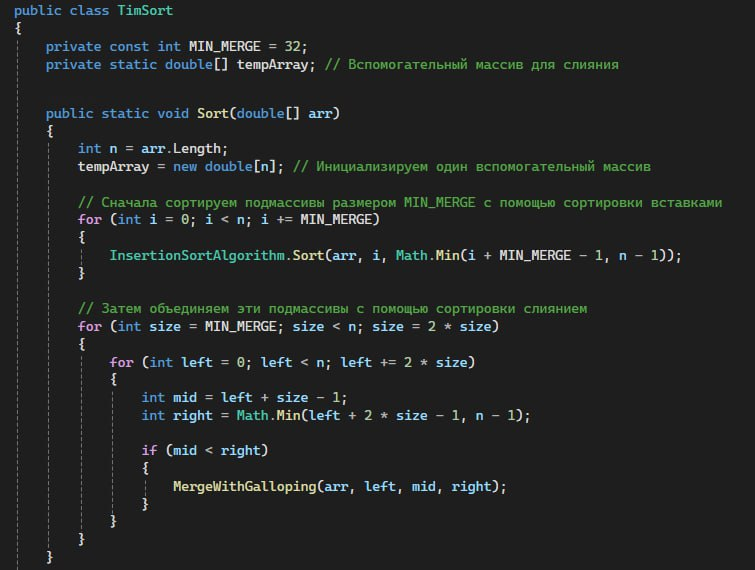


Рис 1.7.2 Код гибридной сортировки(ч.1)

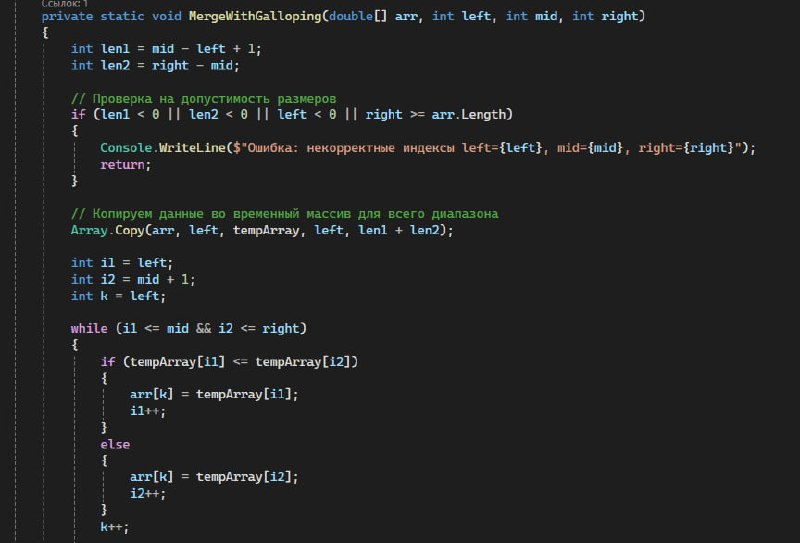


Рис 1.7.3 Код гибридной сортировки(ч.2)

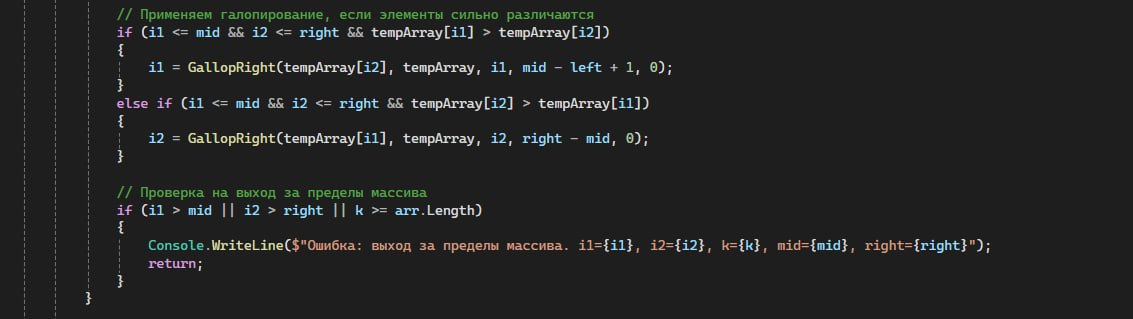


Рис 1.7.4 Код гибридной сортировки(ч.3)

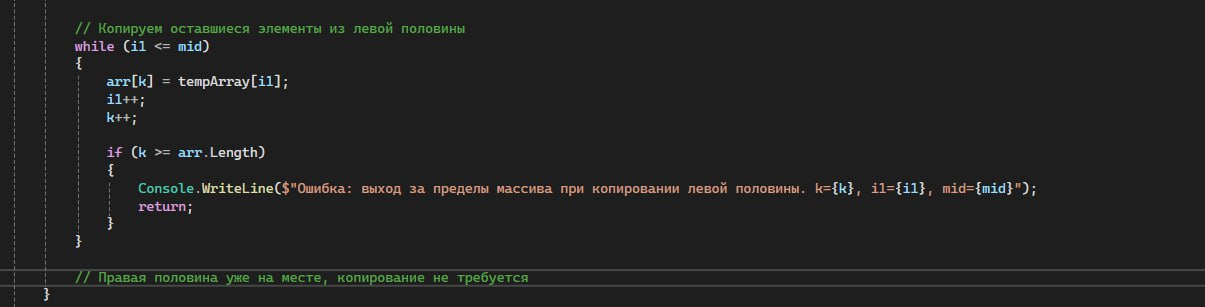


Рис 1.7.5 Код гибридной сортировки(ч.4)

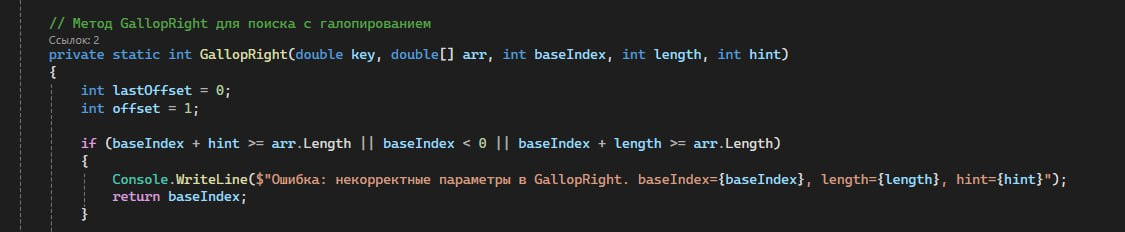


Рис 1.7.6 Код гибридной сортировки(ч.5)

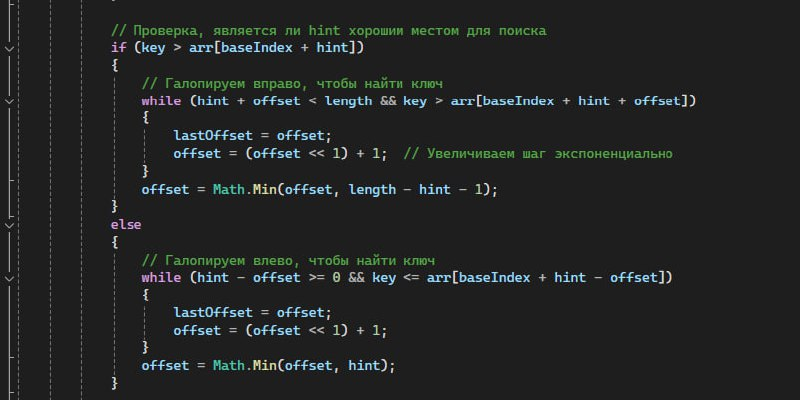


Рис 1.7.7 Код гибридной сортировки(ч.6)

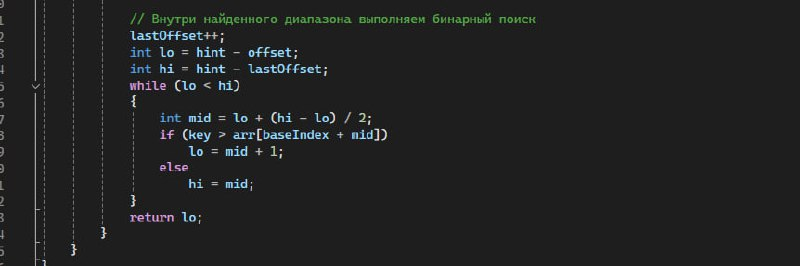


Рис 1.7.8 Код гибридной сортировки(ч.7)

1. Алгоритмы возведения в степень

7.1.Классический(см. Рис. 1.8.1.1):

• Временная сложность: O(N)

• N = 50000

• Ср. знач. на основе тестов: 5

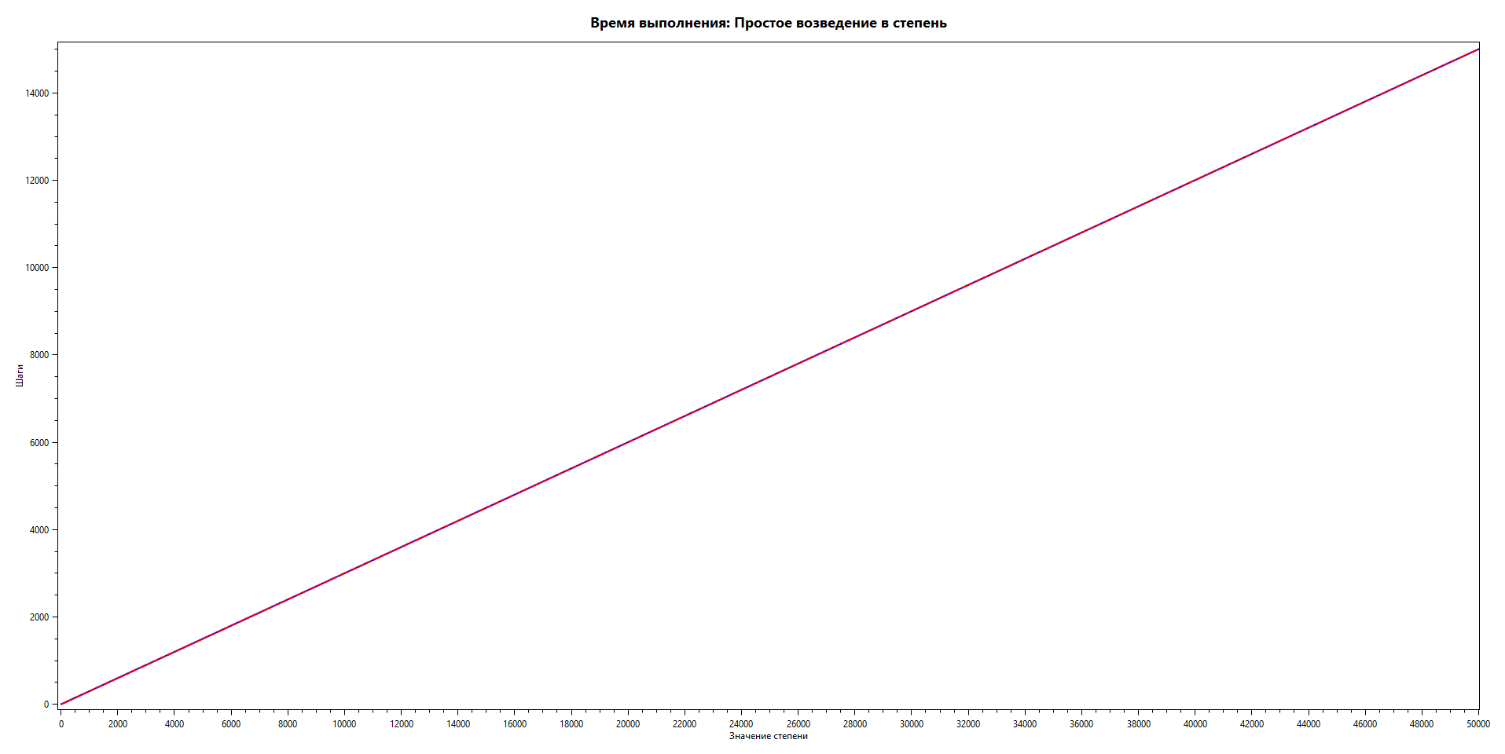


Рис. 1.8.1.1 Возведение в степень (классический алгоритм)

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:

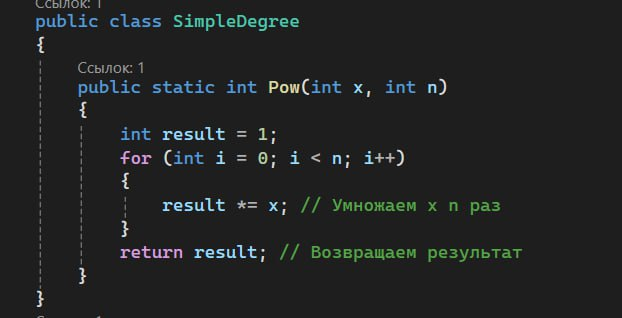
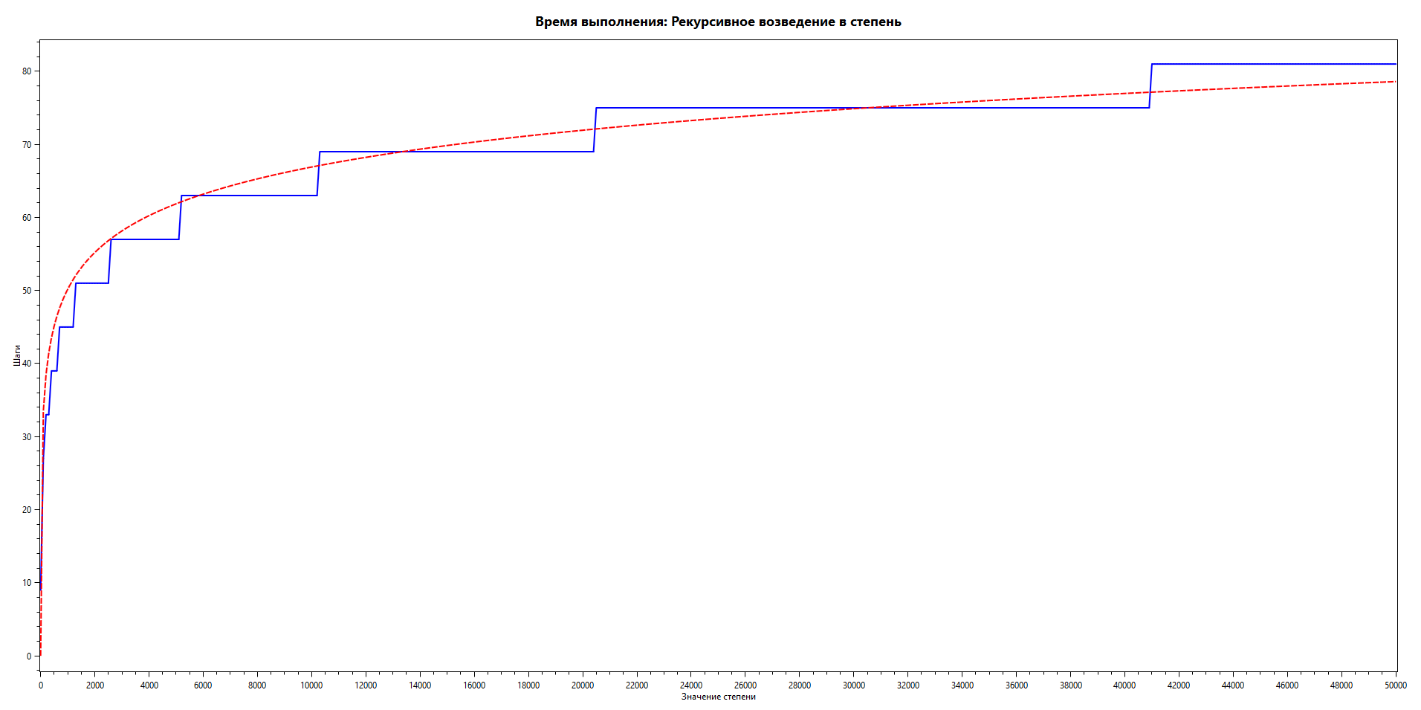


Рис. 1.8.1.2 Возведение в степень (классический алгоритм)

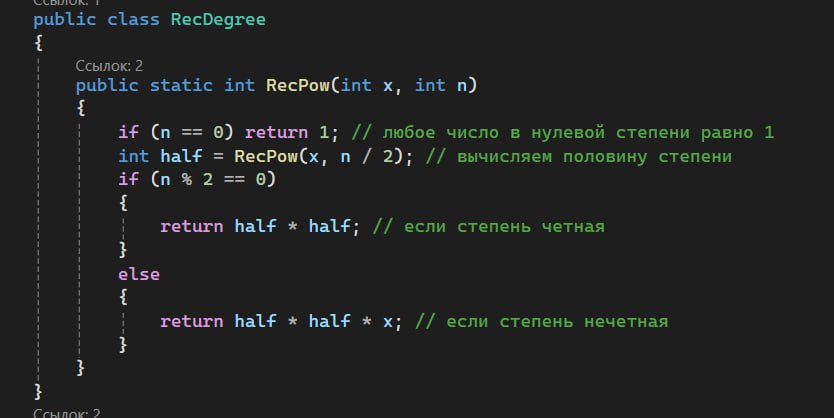
Рекурсивный(см Рис 1.8.2.1):

* Временная сложность: O(log n)
* N = 50000
* Ср. знач. на основе тестов: 5

Рис. 1.8.2.1 Возведение в степень (рекурсивный алгоритм)

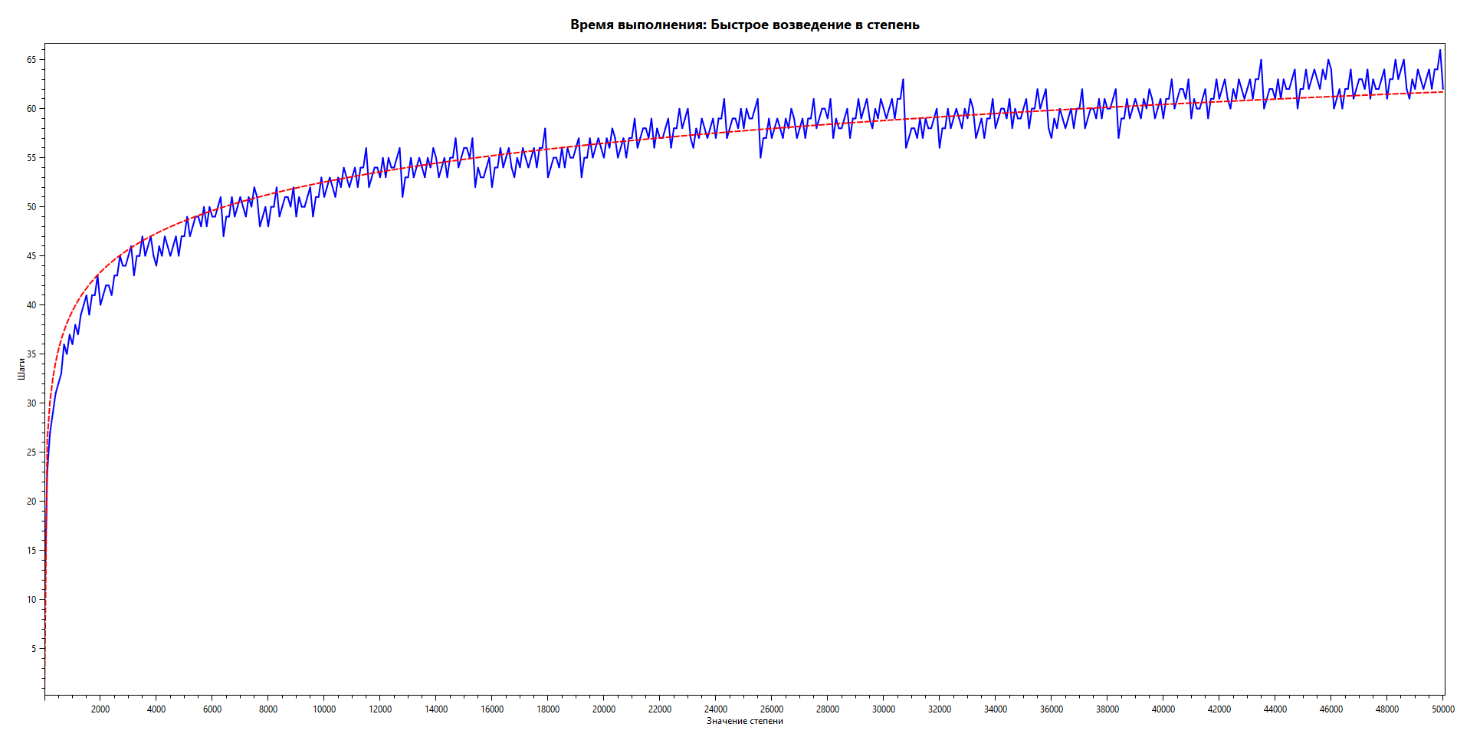
Данный график существенно отличается от предыдущего (см. Рис. 8.1.1) количеством шагов. Так же он имеет резкие и неравномерные скачки количества шагов.

Код изображен на Рис. 1.8.2.2:

  
Рис. 1.8.2.2 Возведение в степень (рекурсивный алгоритм)

Быстрый(см Рис. 1.8.3.1):

* Временная сложность: O(log n)
* N = 50000
* Ср. знач. на основе тестов: 5

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:

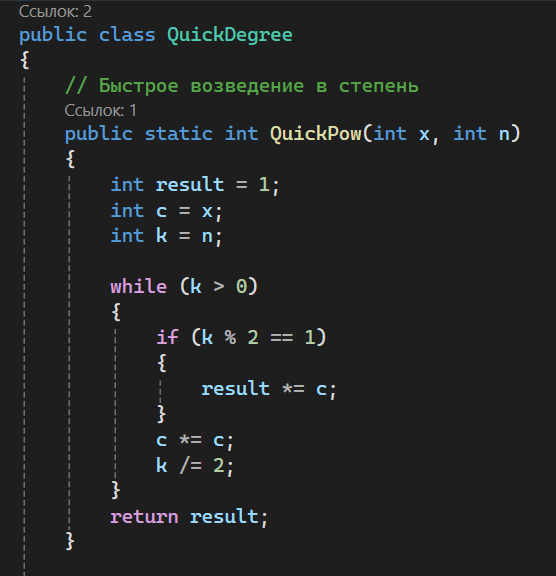
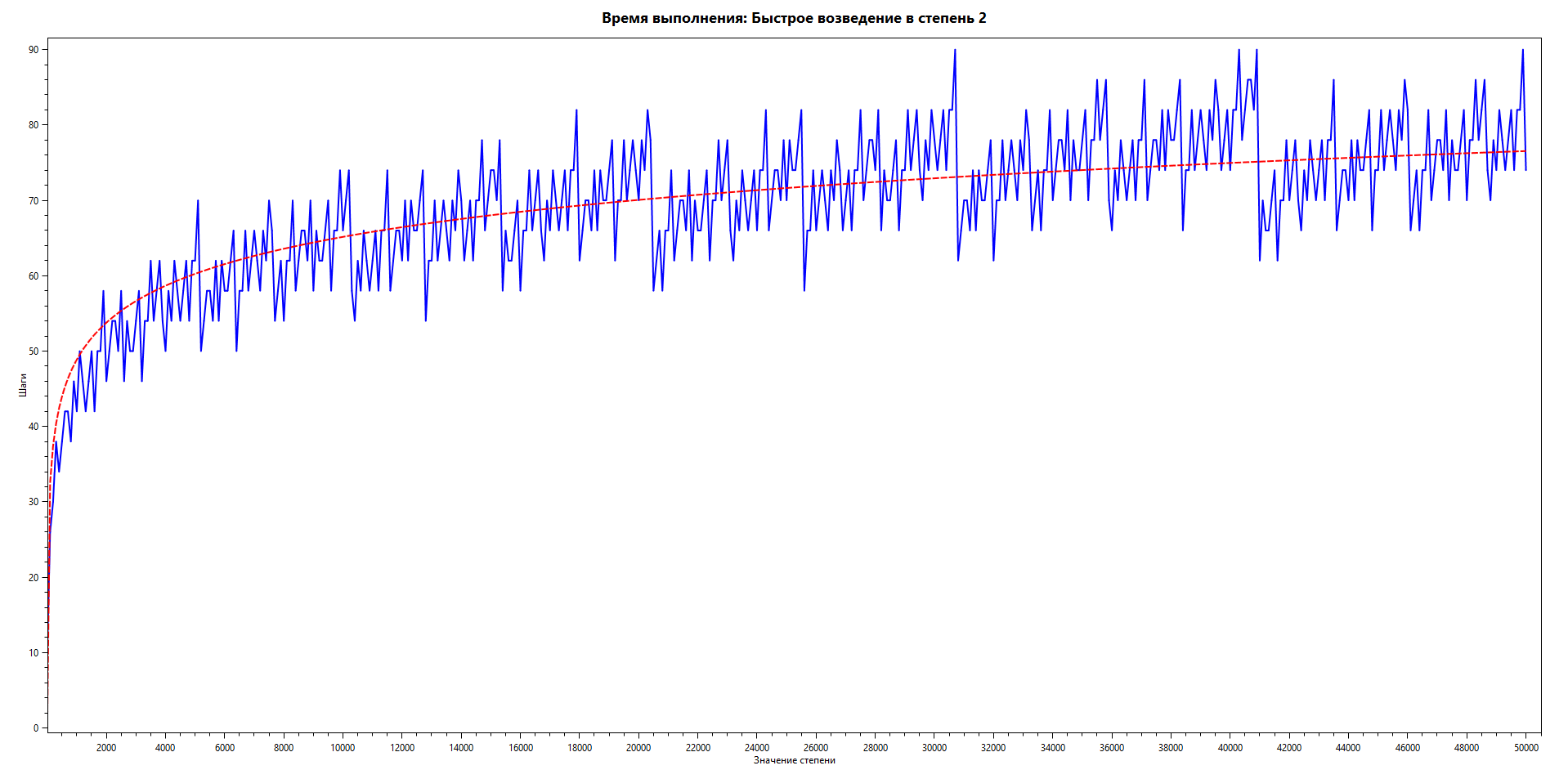


Рис. 1.8.3.2 Возведение в степень (быстрый алгоритм)

Классический быстрый(см. Рис. 1.8.4.1):

* Временная сложность: O(log n)
* N = 50000
* Ср. знач. на основе тестов: 5

Рис. 1.8.4.1 Возведение в степень (классический быстрый алгоритм)

Примечание:

Было замечено, что при больших степенях и параллельных вычислений, рекурсивный алгоритм вызывает переполнение стека. Быстрые алгоритмы для низкий значений показывают почти константную сложность.

Код изображен на Рис. 1.8.4.2:

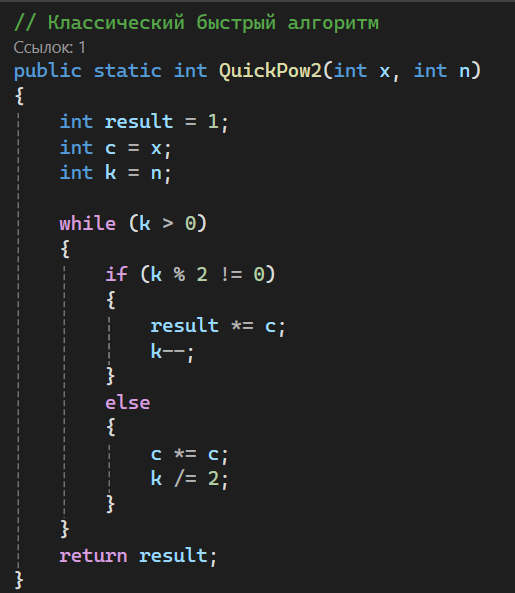


Рис. 1.8.4.2 Возведение в степень (классический быстрый алгоритм)

**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 = 1500
* Ср. знач. на основе тестов: 5

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):

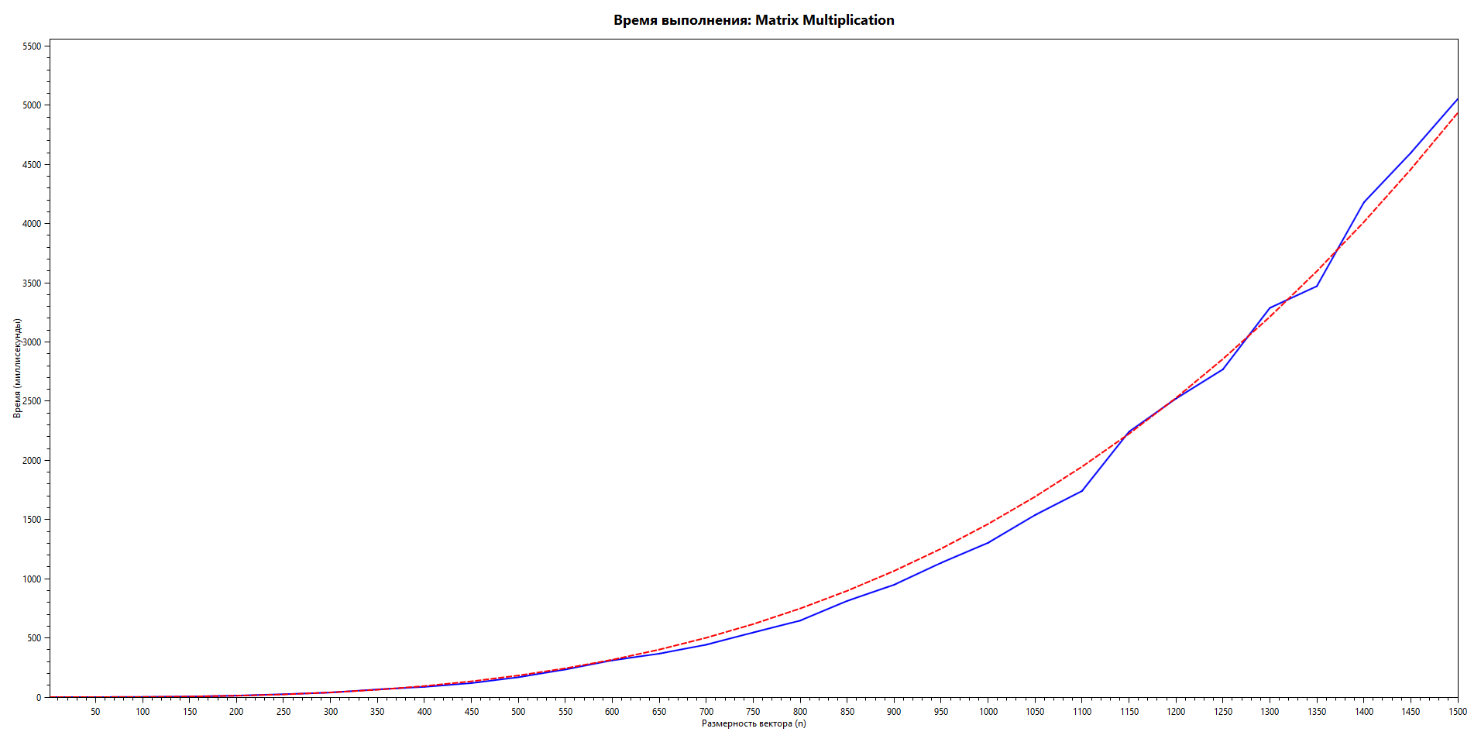


Рис. 2.1 Алгоритм перемножение двух матриц

Код изображен на Рис. 2.2:

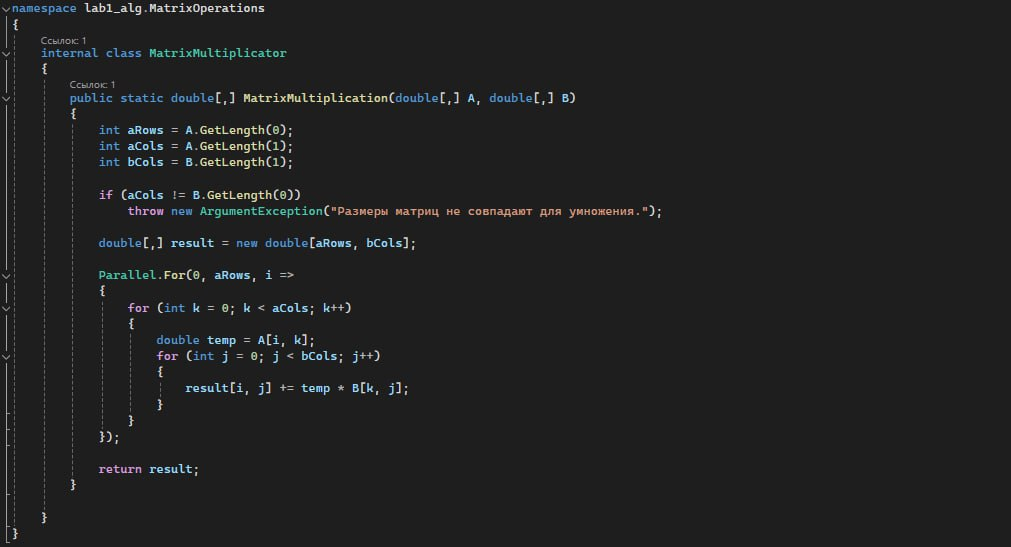


Рис. 2.2 Код для матриц

**Задание III.**

**1. Insertion sort**

* Временная сложность:
* N = 10000
* Ср. знач. на основе тестов: 5

Алгоритм Insertion sort заключается в следующем:

1. Начало: Начинаем с первого элемента массива, который считается отсортированной частью.

2. Итерация: Проходим по каждому элементу массива, начиная со второго. Для каждого элемента:

- Сохраняем его значение в переменной (например, key).

- Сравниваем key с элементами отсортированной части (то есть с элементами слева от него).

- Сдвигаем элементы отсортированной части вправо, пока не найдем позицию, куда можно вставить key.

3. Вставка: Вставляем key на найденную позицию.

4. Повторение: Повторяем шаги 2 и 3 для всех элементов массива, пока не пройдем все элементы.

5. Конец: В результате массив будет отсортирован.

Применяется для сортировки небольших массивов данных, сортировки данных, поступающих по частям, сортировки массивов с ограниченным количеством изменений и сортировки списков, когда элементы вставляются по одному.

График зависимости времени выполнения Insertion sort массива от объёма данных(см. Рис. 3.1.1):

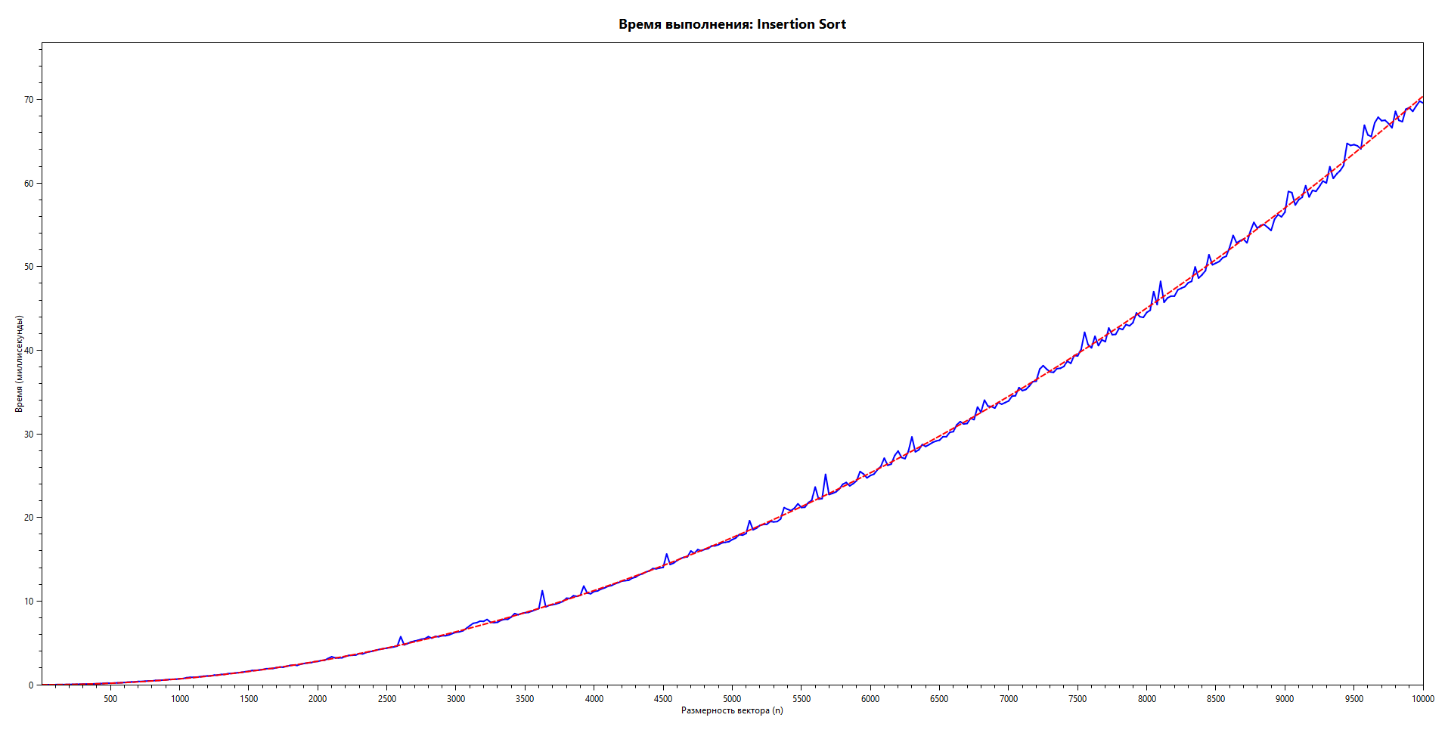


Рис. 3.1.1 алгоритм Insertion sort

Код алгоритма( см. Рис. 3.1.2):

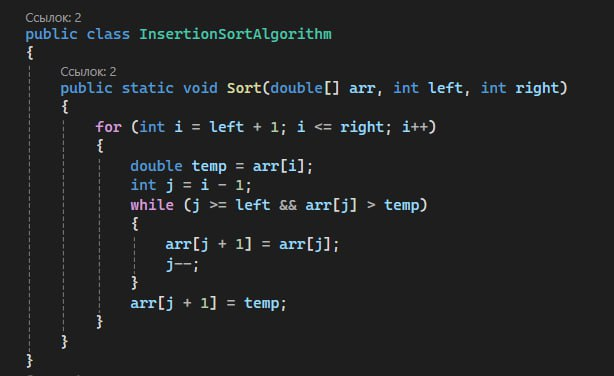


Рис. 3.1.2 код Insertion sort

**2. Shell Sort**

* Временная сложность:
* N = 20000
* Ср. знач. на основе тестов: 5

Основные этапы алгоритма сортировки Шелла:

1. Определение начального шага: Начинаем с шага `gap`, равного примерно половине размера массива.

2. Сортировка с шагом: Проводим сортировку вставками для элементов, находящихся на расстоянии `gap` друг от друга.

3. Сокращение шага: Уменьшаем значение `gap` (например, делим его на 2) и повторяем шаг 2.

4. Повторение: Продолжаем повторять шаги 2 и 3 до тех пор, пока `gap` не станет равным 1.

Применяется для сортировки средних и больших массивов данных, сортировки массивов с неравномерным распределением и сортировки массивов, где требуется быстрое получение отсортированных данных

График зависимости времени выполнения Shell sort массива от объёма данных(см. Рис. 3.2.1):

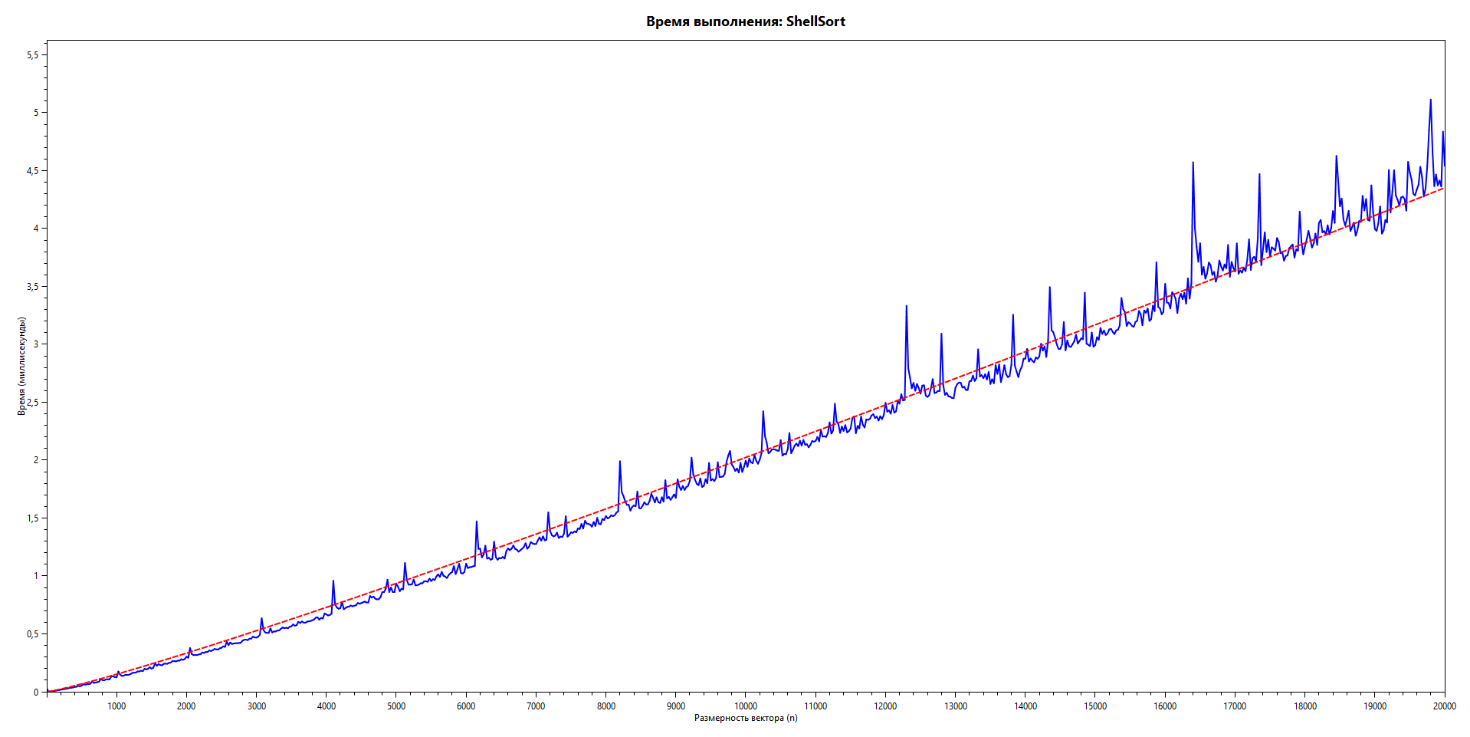


Рис. 3.2.1 алгоритм Shell sort

Код алгоритма( см. Рис. 3.2.2):

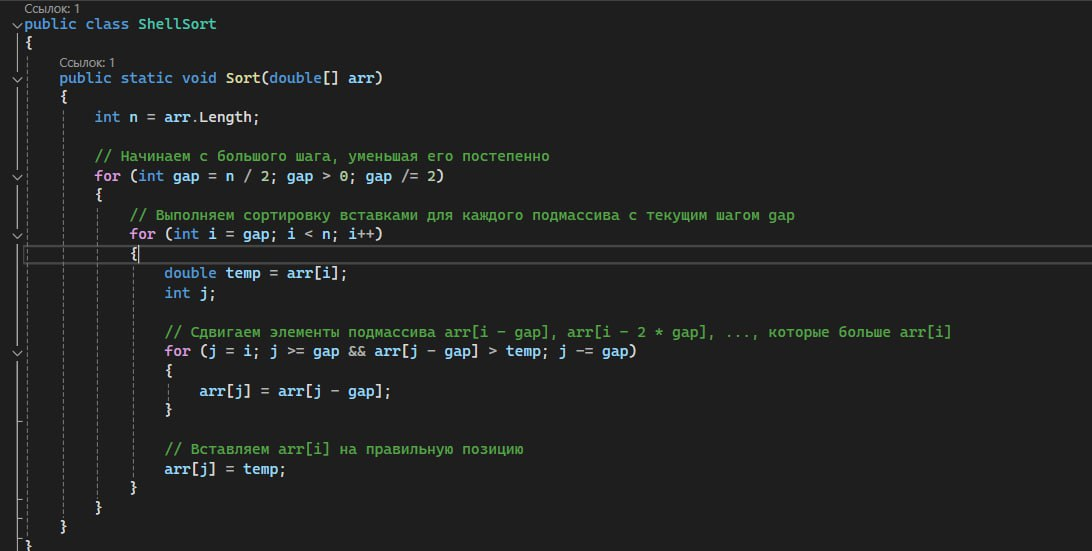


Рис. 3.2.2 код Shell sort

**3. Coctail Sort**

* Временная сложность:
* N = 10000
* Ср. знач. на основе тестов: 5

Основные этапы алгоритма коктейльной сортировки:

1. Инициализация: Устанавливаем флаг `swapped` в значение `False`, чтобы начать сортировку.

2. Прямой проход:

- Проходим от начала массива к концу и сравниваем соседние элементы.

- Если текущий элемент больше следующего, меняем их местами.

- Если было выполнено хотя бы одно изменение (т.е. `swapped` становится `True`), продолжаем проход до конца массива.

3. Обратный проход:

- Если во время прямого прохода не было изменений (т.е. `swapped` = `False`), это означает, что массив отсортирован, и сортировку можно завершить.

- Если изменения были, меняем направление и берем проход в обратную сторону (с конца к началу массива).

- Сравниваем и меняем соседние элементы так же, как и на прямом проходе.

4. Повторение:

- Продолжаем прямые и обратные проходы, пока флаг `swapped` остается `True`, то есть пока происходят изменения.

Применяется для сортировки небольших массивов данных, визуализации сортировки и сортировки массивов, где требуется интерактивная визуализация

График зависимости времени выполнения Coctail sort массива от объёма данных(см. Рис. 3.3.1):

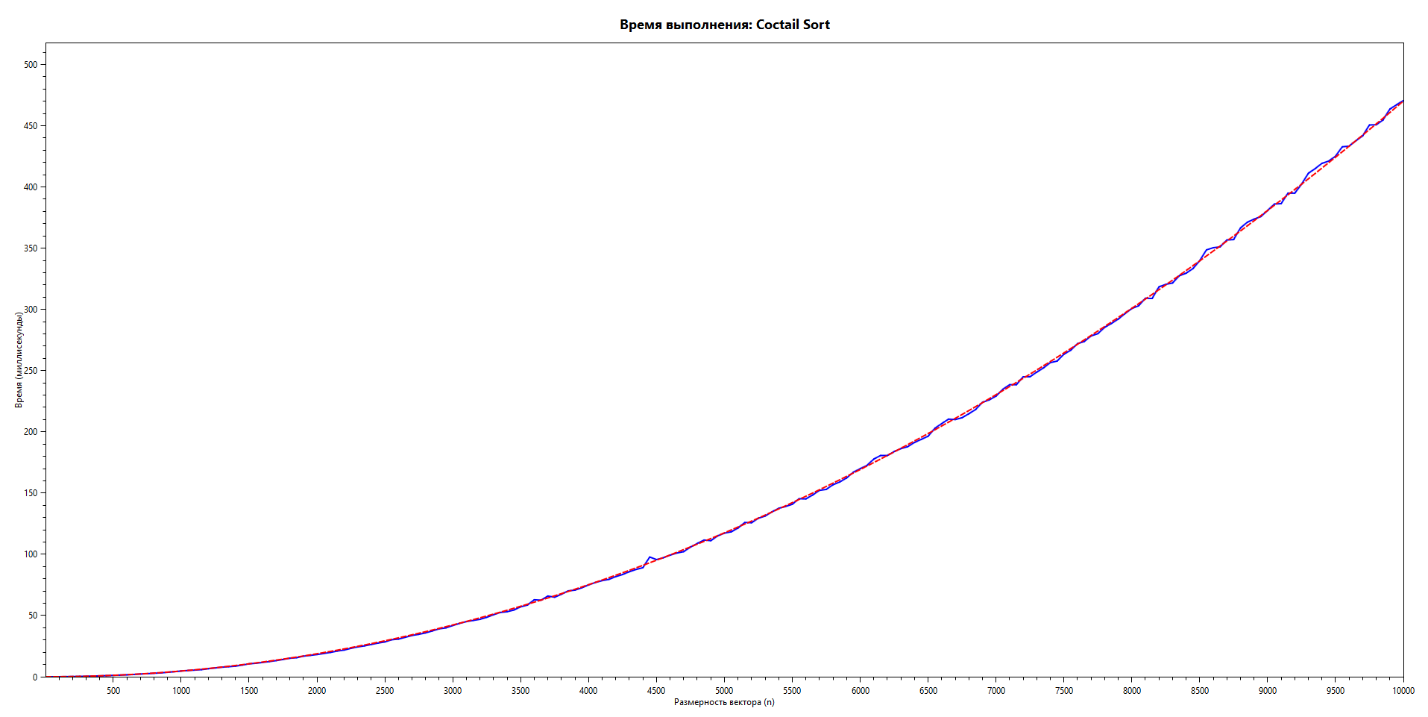


Рис. 3.3.1 алгоритм Coctail sort

Код алгоритма( см. Рис. 3.3.2 — 3.3.4):

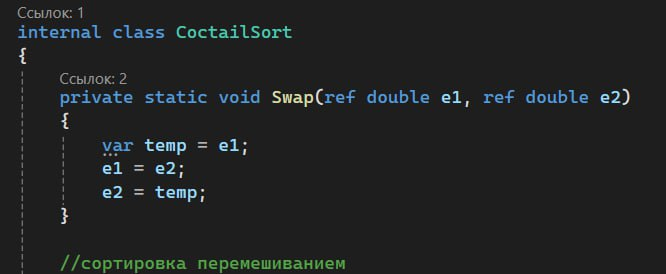


Рис. 3.3.2 код Coctail sort(ч.1)

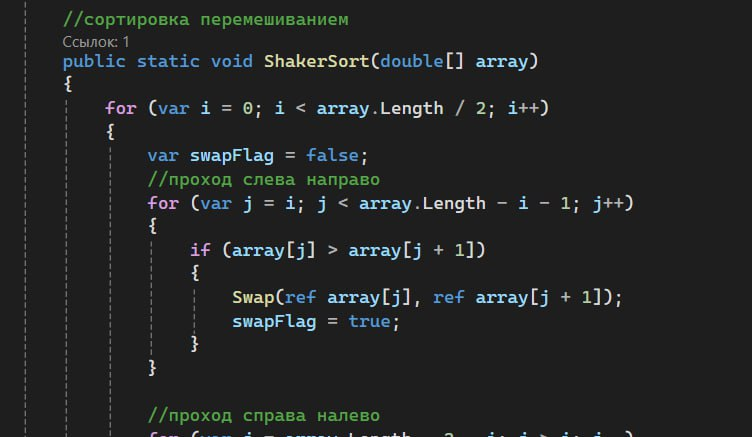


Рис. 3.3.3 код Coctail sort(ч.2)

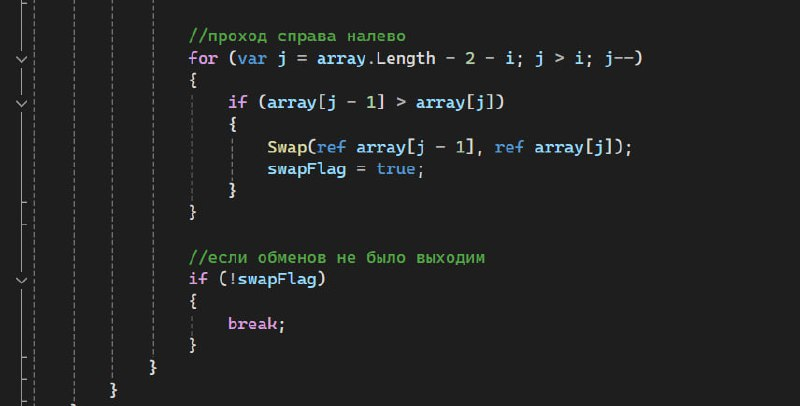
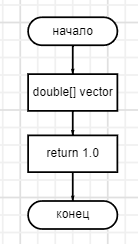


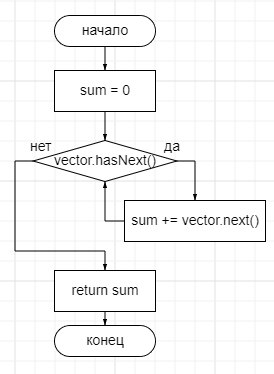
Рис. 3.3.4 код Coctail sort(ч.3)

Примечание

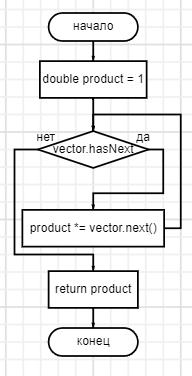
1. Константа



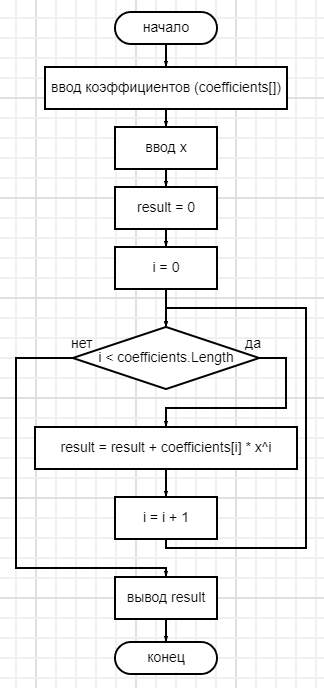
2. Сумма



3. Произведение



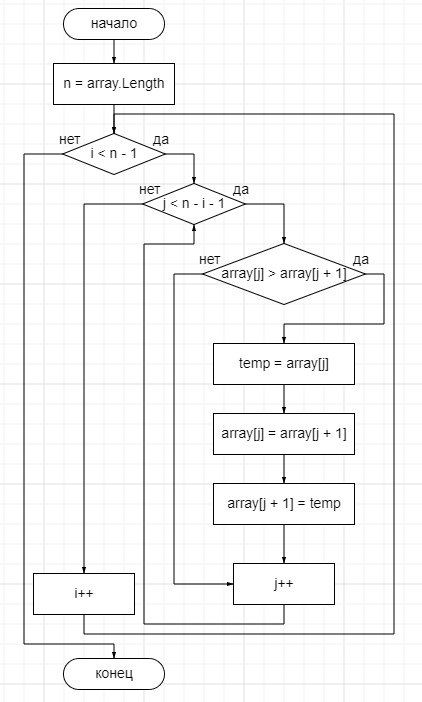
4. Наивное



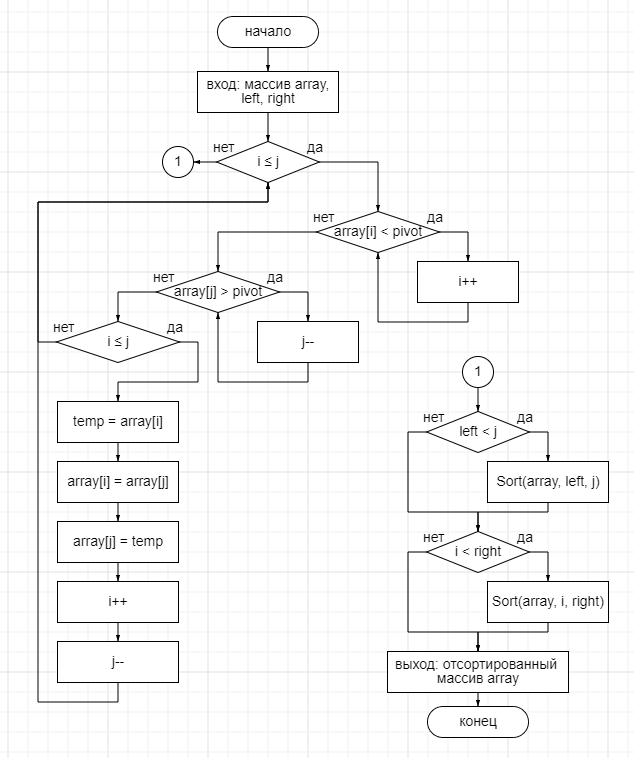
5. Горнер



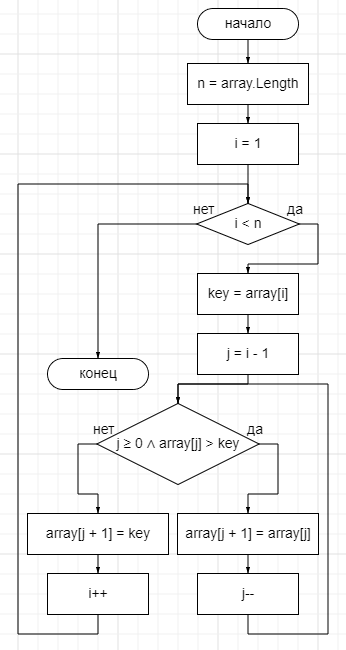
6. Bubble sort



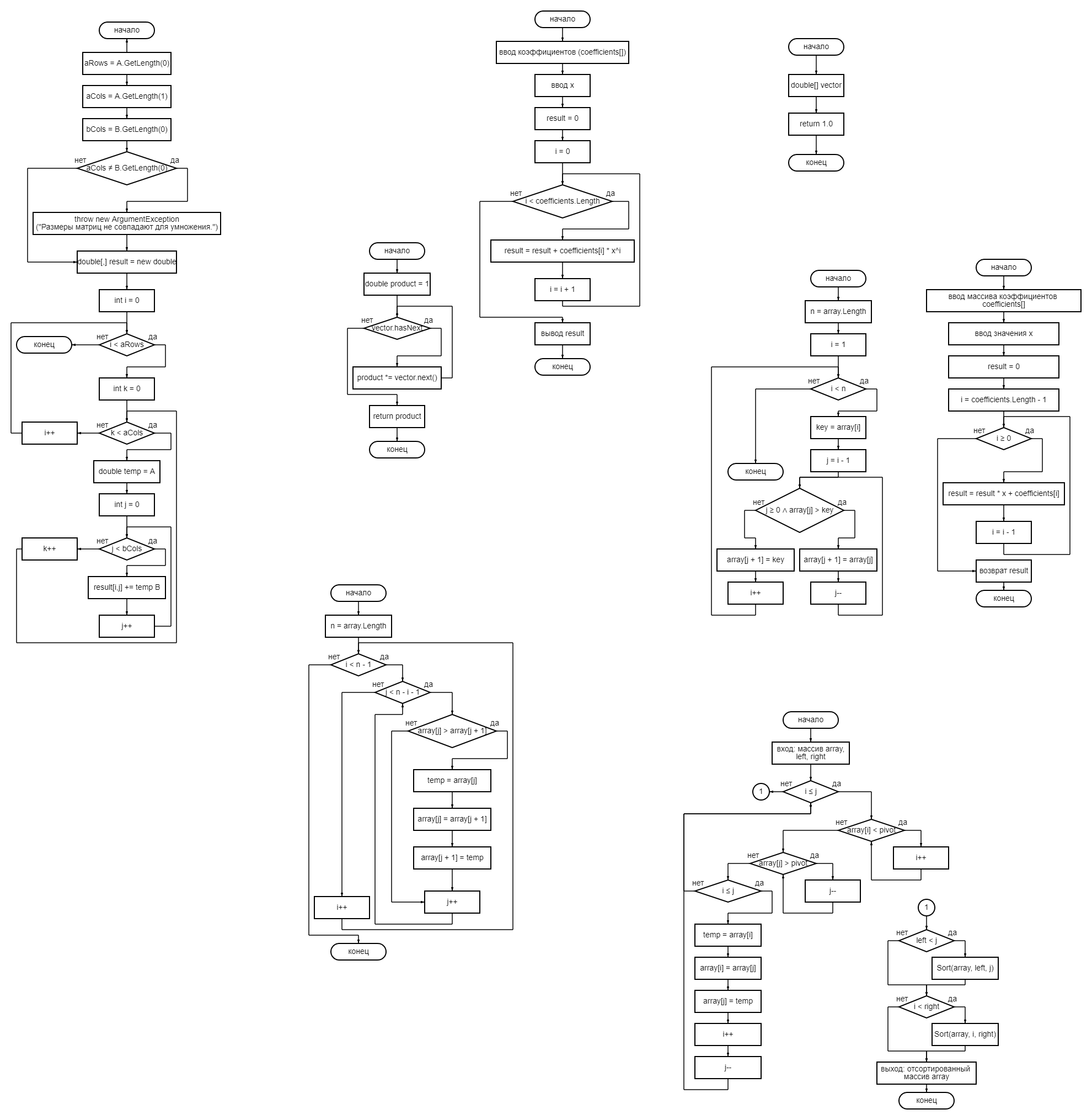
7. Quick sort



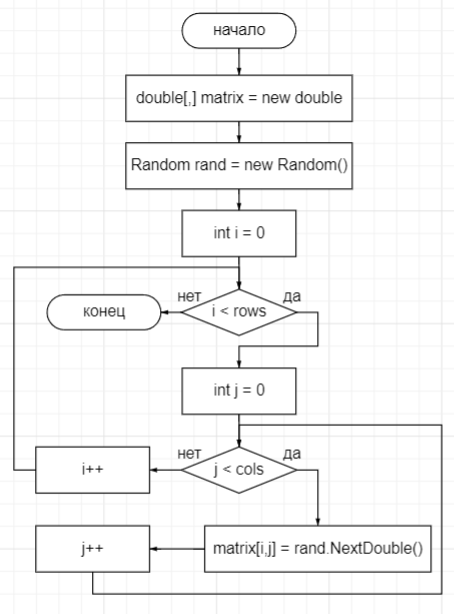
8. Insertion sort



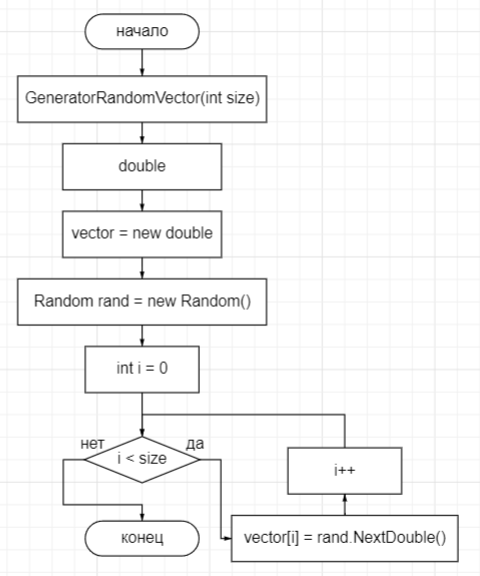
9. Перемножение матриц



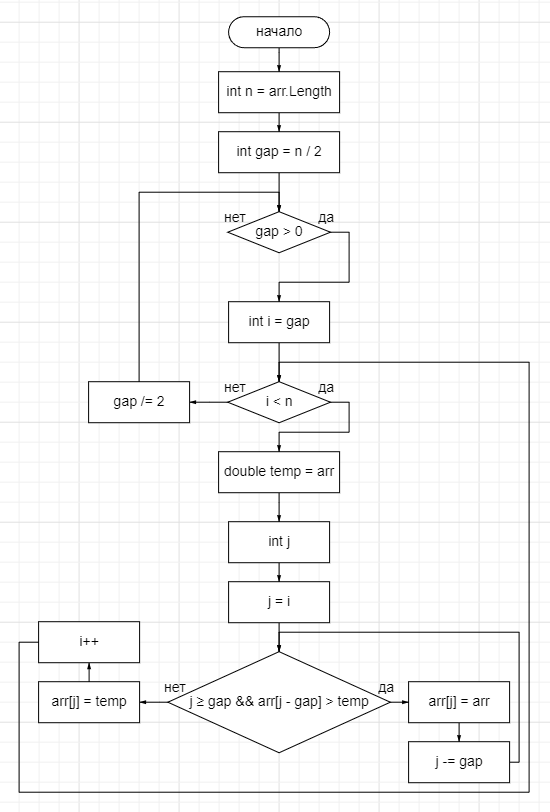
10. Генерация матриц



11. Генерация векторов



12. Shell sort



13. Cocktail sort

