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 С.М. Панов ПрИ-201

подпись инициалы, фамилия группа

А.Д. Казбеков ПрИ-201

подпись инициалы, фамилия группа

В.И. Кочетков ПрИ-201

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.

Лабораторная работа была выполнена с использованием 4 классов: MatrixOperations для логики работы с матрицами, SortingAlgoritms для логики работы с алгоритмами сортировки, а также VectorOperations для работы с векторами и класс Program для использования всей логики. Так же в проекте использовалось Wpf.

Стек технологий:

1. C#

***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 = 100
* Ср. знач. на основе тестов: 1

Каждому новому элементу, не входящему в массив, присваивается длина массива, умноженная на 2.

Замер времени от кол-ва элементов: (см. рис. 1.1.1):

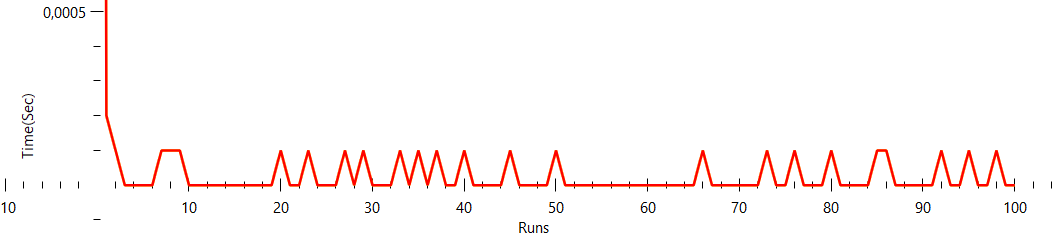


Рис. 1.1.1 Const алгоритм

Код алгоритма изображен на: (см. рис. 1.1.2):

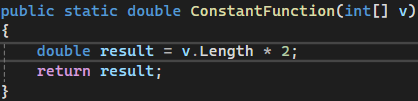


Рис. 1.1.2 Код алгоритма

1. **F(v) = (сумма элементов)**

* Временная сложность: O(n)
* N = 50 000
* Ср. знач. на основе тестов: 50

Алгоритм суммирует все элементы(i-тые) массива.

Замер времени от кол-ва элементов указывает на линейную зависимость от кол-ва данных на входе: (см. рис. 1.2.1):

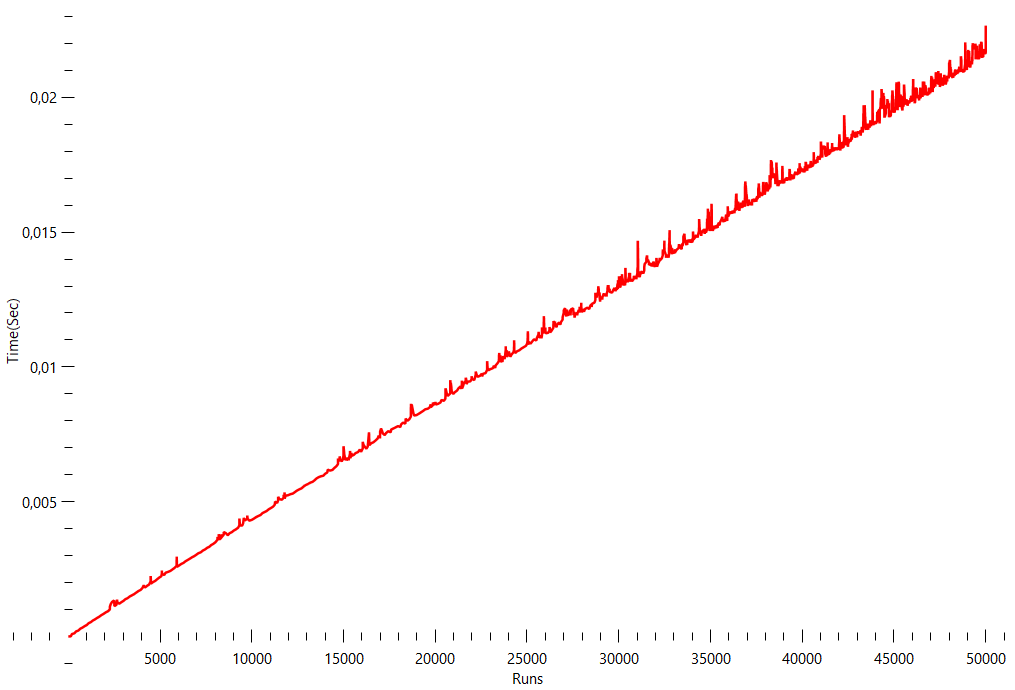


Рис. 1.2.1 Усреднённый график алгоритма суммы элементов

Код алгоритма изображен на: (см. рис. 1.2.2):

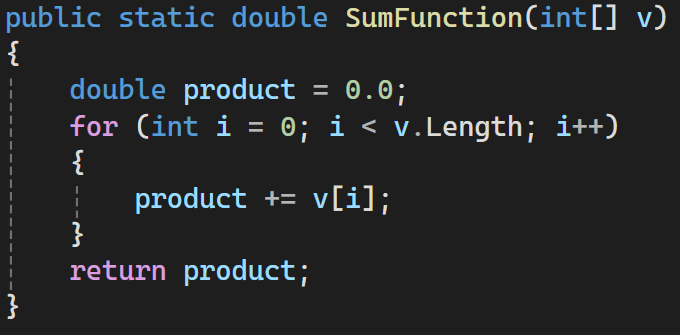


Рис. 1.2.2 Код алгоритма суммы элементов

1. **(произведение элементов)**

* Временная сложность: O(n)
* N = 50 000
* Ср. знач. на основе тестов: 50

Алгоритм выполняет операцию умножения каждого элемента(i-того) массива. При анализе получившегося графика (см. рис. 1.3.1) получилась линейная зависимость.

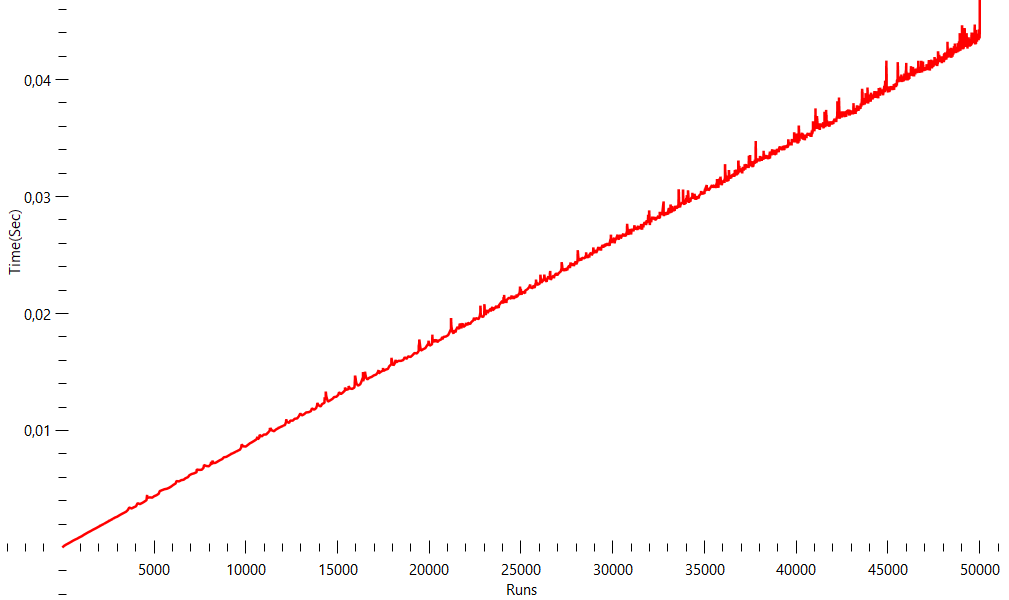


Рис. 1.3.1 Усреднённый график алгоритма произведения элементов

Код алгоритма изображен на: (см. рис. 1.3.2):

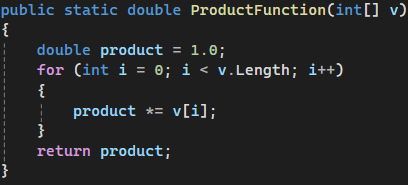


Рис. 1.3.2 Код алгоритма произведения элементов

* Временная сложность:
* T = O(n)
* N1 = 5 000
* N2 = 1.5
* Ср. знач. на основе тестов: 10

Алгоритм Горнера - вычисления значения многочлена, записанного в виде суммы мономов (одночленов), при заданном значении переменной.

График зависимости времени выполнения алгоритма методом Горнера от кол-ва элементов массива и коэффициенте x = 1.5. Получилась линейная зависимость (см. рис. 1.4.1)

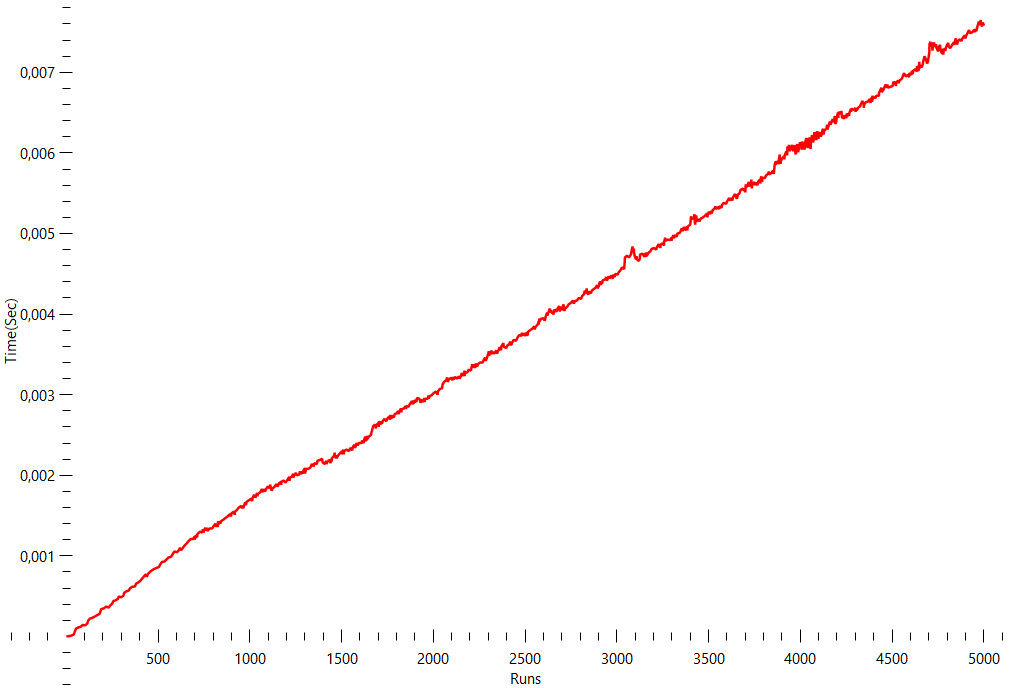


Рис. 1.4.1 Усреднённый график Метода Горнера

Код алгоритма изображен на: (см. рис. 1.4.2):

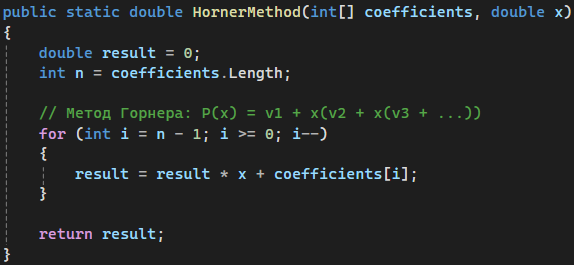


Рис. 1.4.2 Код алгоритма Горнера

График зависимости времени выполнения выполнения алгоритма Прямым (Straigth) методом от количества элементов массива (см. рис. 1.4.3)

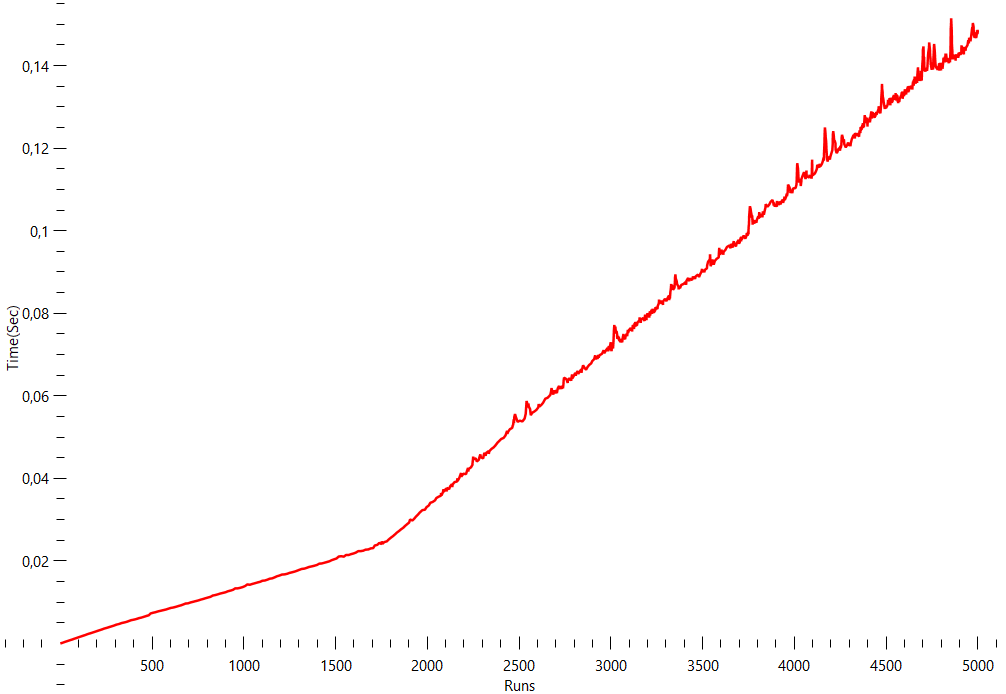


Рис. 1.4.3 Усреднённый график метода прямого вычисления

Код изображен на: (см. рис. 1.4.4):

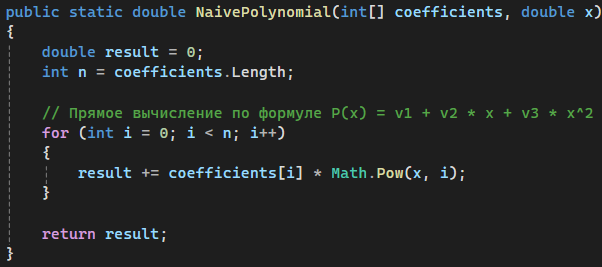


Рис. 1.4.4 Код алгоритма прямого вычисления

Примечание: Сравнив графики (см. рис. 1.4.1 и рис. 1.4.4) можно сделать вывод, что метод Горнера работает примерно в 8-10 раз быстрее, чем алгоритм прямого вычисления многочлена.

1. **Алгоритм сортировки пузырьком (Bubble sort) элементов**

* Временная сложность: O(n2)
* N = 6000
* Ср. знач. на основе тестов: 3

Примечание: последовательно сравниваются значения соседних элементов и меняются местами, если предыдущее оказывается больше последующего.

Таким образом элементы с большими значениями оказываются в конце списка, а с меньшими остаются в начале.

Алгоритм прост в понимании, однако, эффективен только для небольших массивов данных.

График зависимости времени выполнения от объема данных (см. рис. 1.5.1)

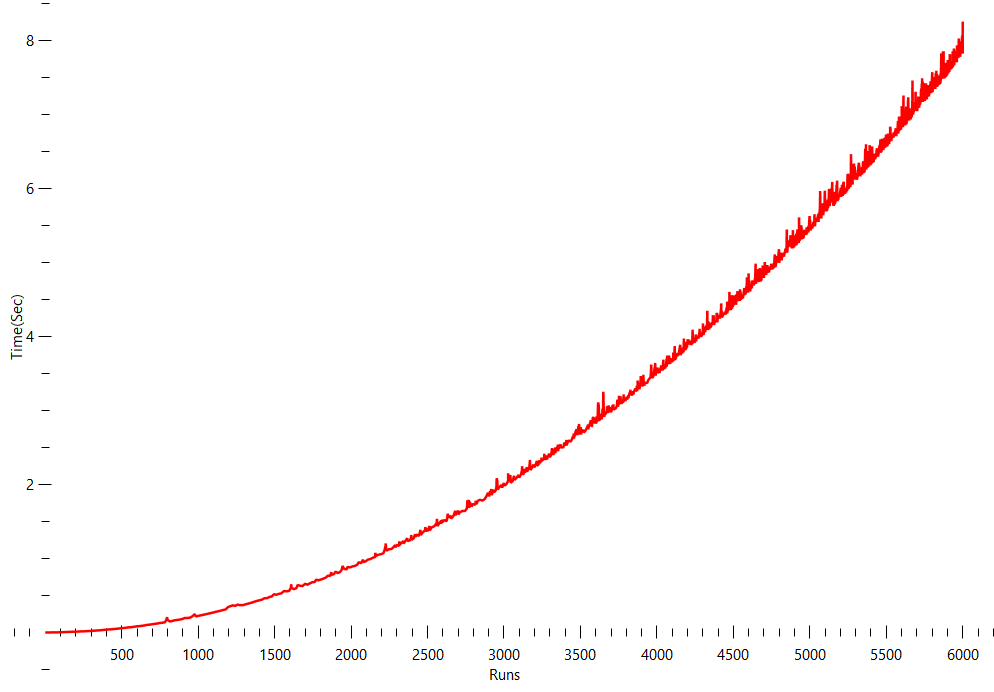


Рис. 1.5.1 Усреднённый график алгоритма сортировки пузырьком

Код алгоритма изображен на: (см. рис. 1.5.2):

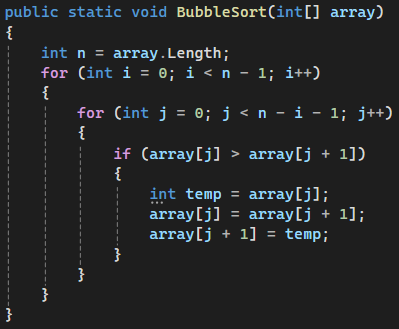


Рис. 1.5.2 Код алгоритма сортировки пузырьком

1. **Алгоритм быстрой сортировки (Quick sort) элементов**

* Временная сложность: O(N \* log(N))
* N = 6000
* Ср. знач. на основе тестов: 3

Быстрая сортировка - улучшенная сортировка пузырьком, перестановки производятся не только между соседними элементами, после каждого прохода коллекция рекурсивно делится на две независимых.

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) (см. рис. 1.6.1)

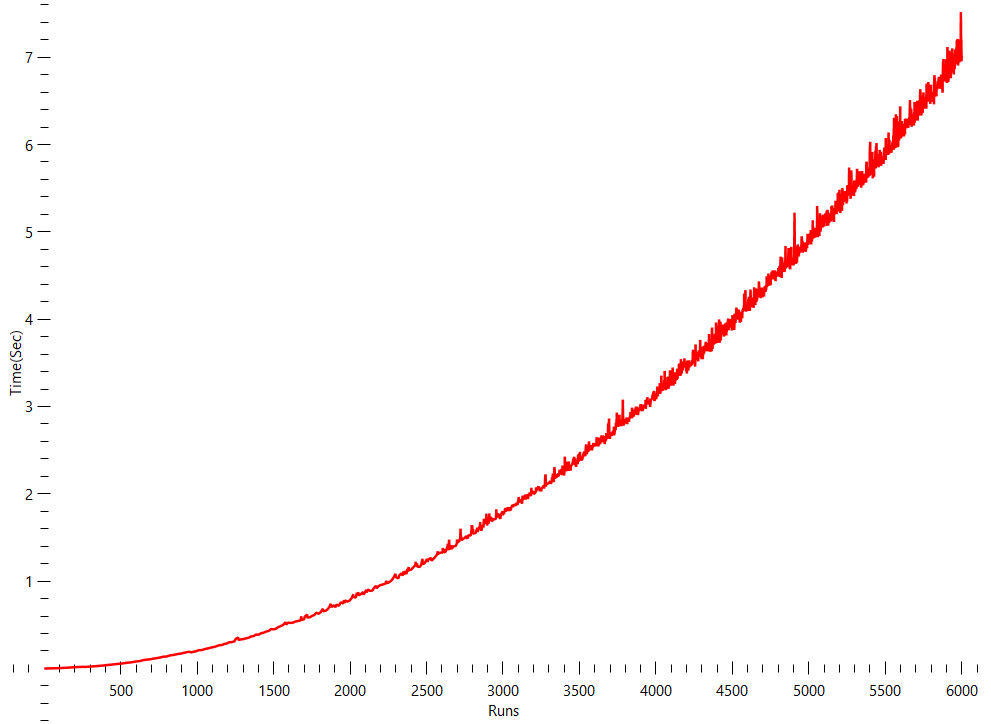


Рис. 1.6.1 Усреднённый график алгоритма быстрой сортировки

Код алгоритма изображен на: (см. рис. 1.6.2):

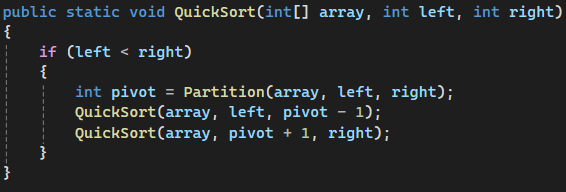


Рис. 1.6.2 Алгоритм быстрой сортировки

Примечание: для реализации был использован дополнительный метод Partition (см. рис 1.6.3)

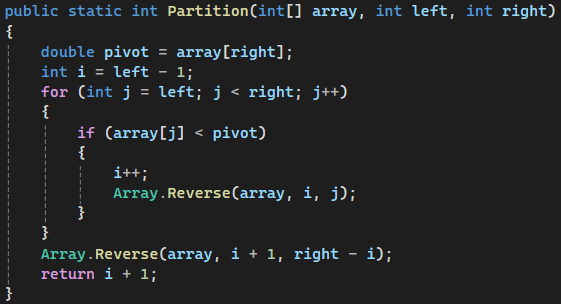


Рис. 1.6.3 Вспомогательный метод

1. **Гибридный алгоритм сортировки Timsort элементов**

* Временная сложность: O(N \* log(N))
* N = 10 000
* Ср. знач. на основе тестов: 25

Примечание: алгоритм TimSort - гибридный алгоритм сортировки, сочетающий сортировку вставками и сортировку слиянием. То есть, алгоритм разбивает коллекцию на упорядоченные, сортирует их вставками и объединяет сортировкой слиянием.

После сортировки вставками объединяет подмассивы попарно сортировкой слиянием и возвращает отсортированный массив. График зависимости времени выполнения TimSort от объема данных (см. рис. 1.7.1)

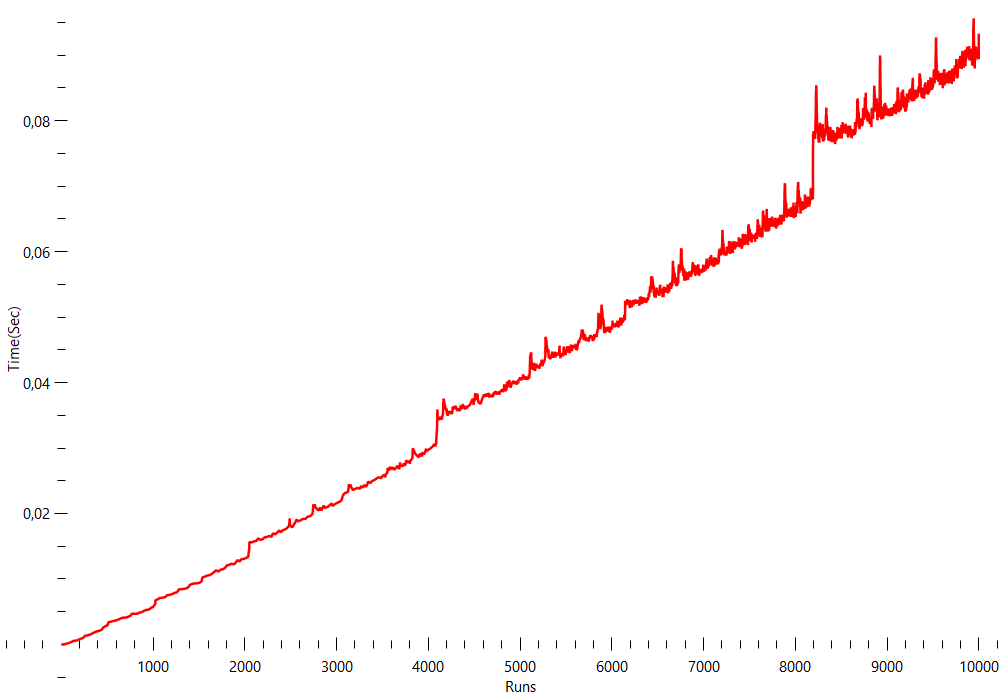


Рис. 1.7.1 Усреднённый график алгоритма сортировки TimSort

Примечание: можно заметить, насколько быстро этот алгоритм сортирует данные, в сравнении с быстрой сортировкой (см. рис 1.6.1), разница колоссальна.

Код алгоритма изображен на: (см. рис. 1.7.2):

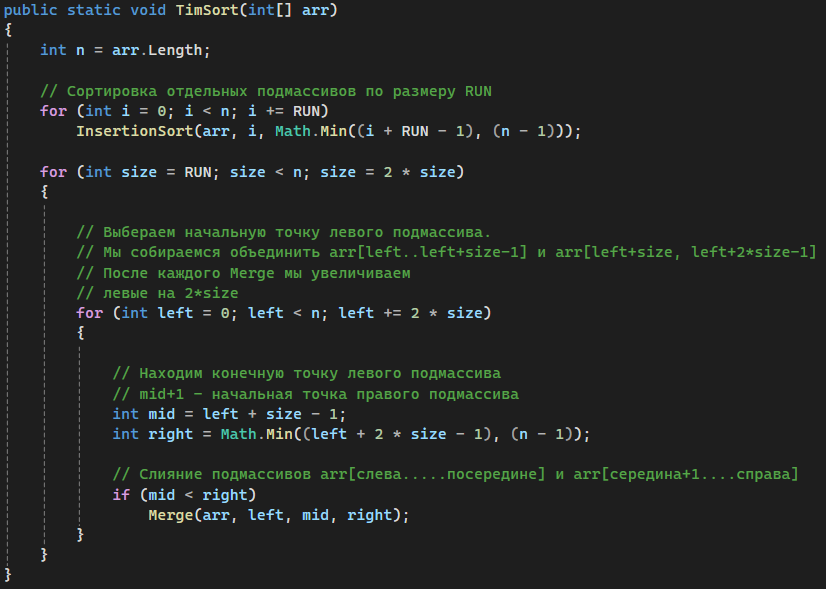


Рис. 1.7.2 Код алгоритма TimSort

Примечание: стоит заметить, что в данном алгоритме используется сортировка слиянием (см. рис. 1.7.3), а также сортировка вставками (см. рис. 1.7.4)

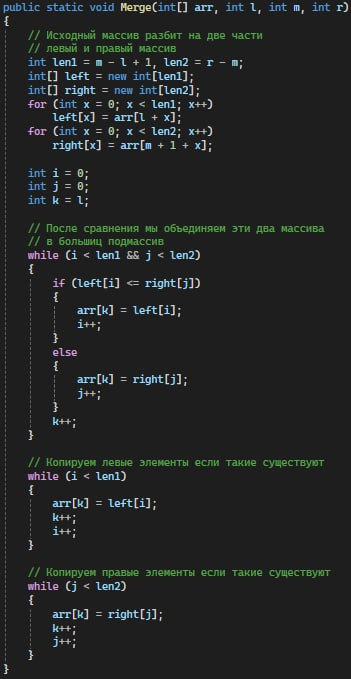


Рис. 1.7.3 Алгоритм сортировки слиянием

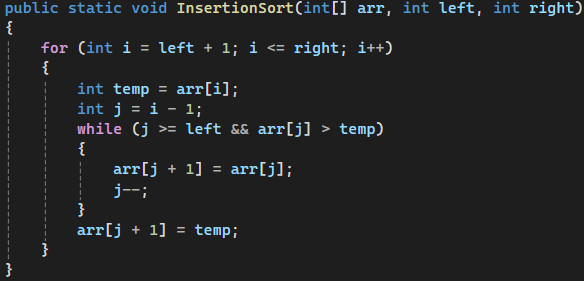


Рис. 1.7.4 Алгоритм сортировки вставками

1. **Алгоритмы возведения в степень**

**8.1)** (Pow) Классический алгоритм возведения в степень (см. Рис. 1.8.1.1)

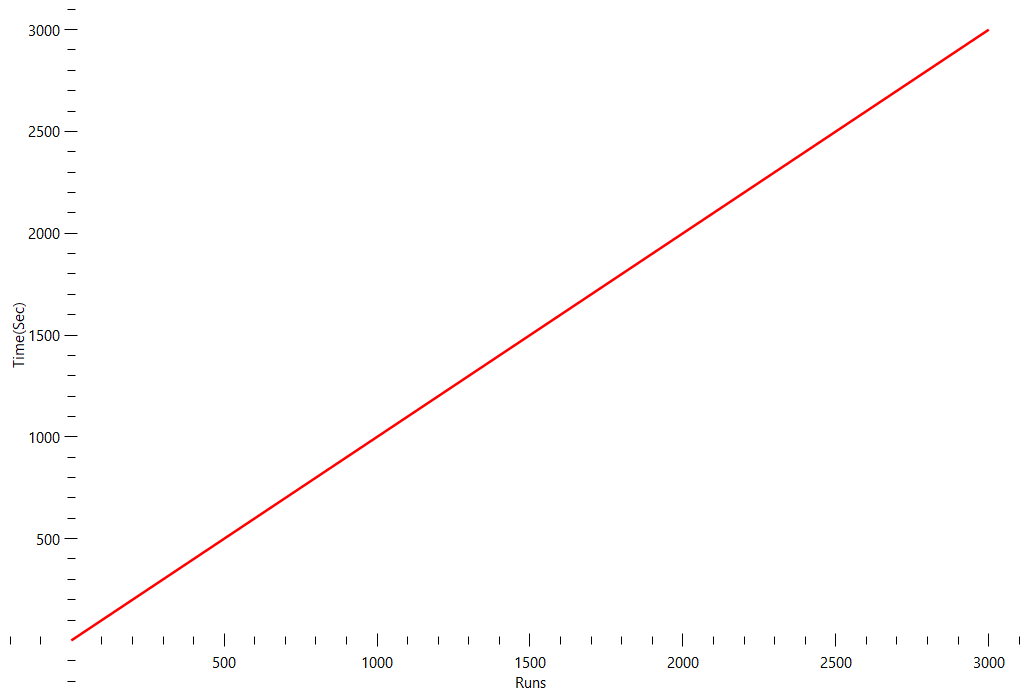


Рис. 1.8.1.1 График алгоритма классического возведения в степнь

Данный график не имеет отклонений, и кол-во шагов достаточно большое

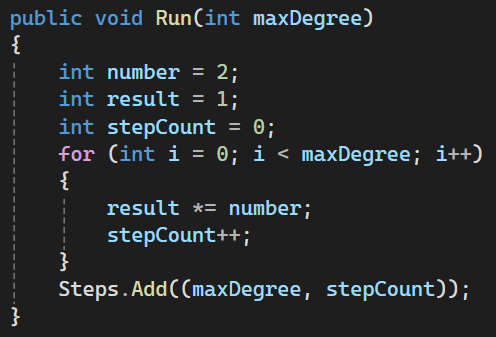


Рис 1.8.1.2 Код алгоритма классического возведения в степень

**8.2)** (RecPow) Рекурсивный алгоритм возведения в степень (см. Рис. 1.8.2.1)

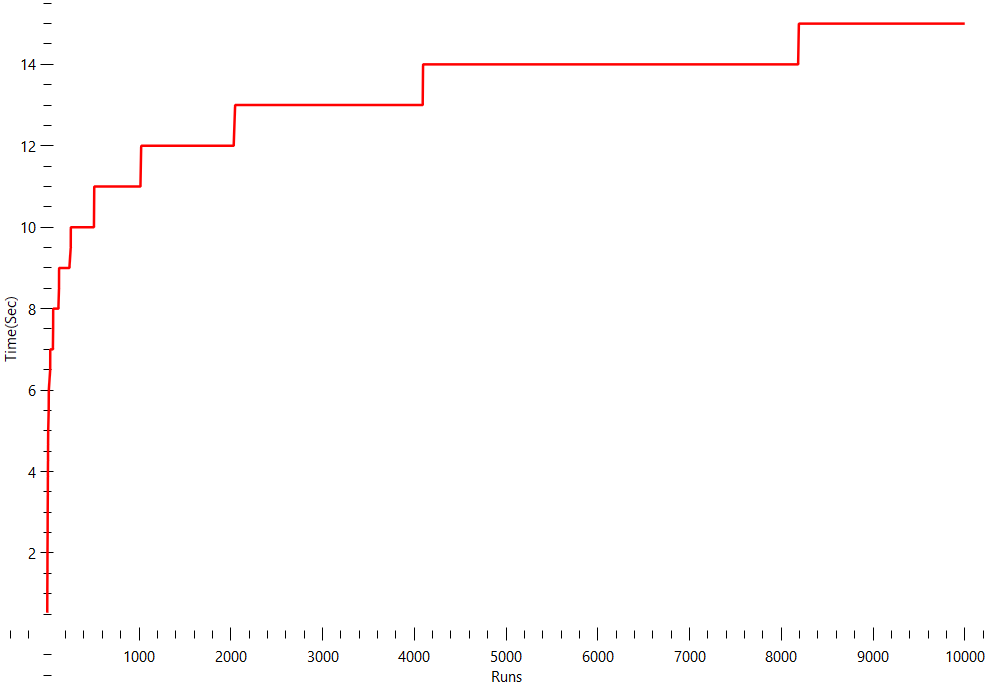


Рис 1.8.2.1 График рекурсивного алгоритма возведения в степень

Этот график сильно отличается от прошлого, он имеет резкие скачки шагов

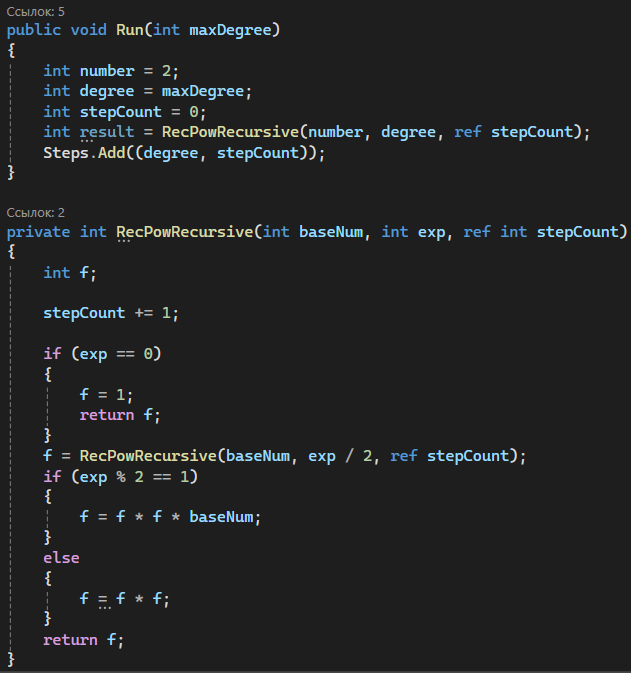


Рис 1.8.2.2 Методы рекурсивного алгоритма возведения в степень

**8.3)** (QuickPow) Быстрый алгоритм возведения в степень (см. Рис. 1.8.3.1)

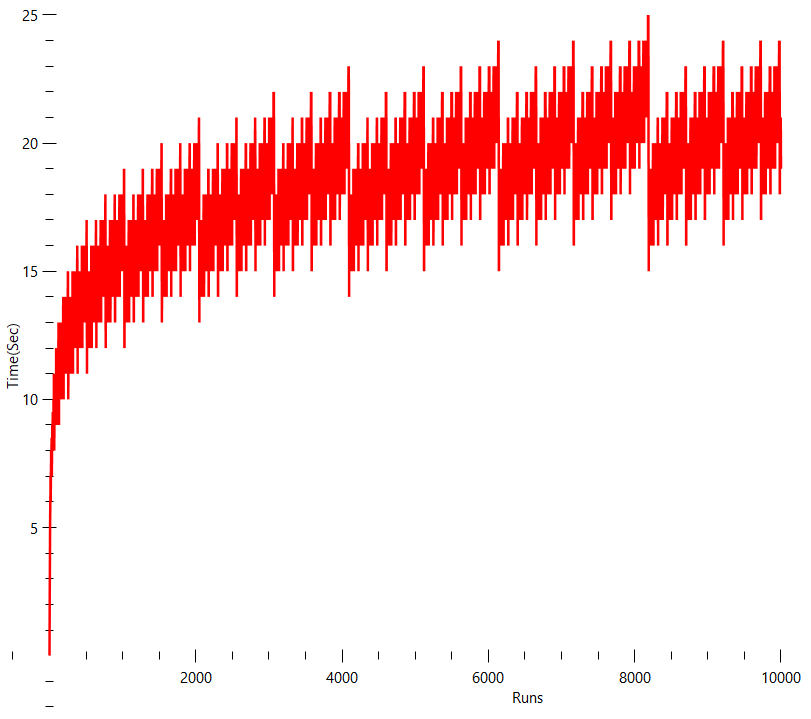


Рис 1.8.3.1 График быстрого алгоритма возведения в степень

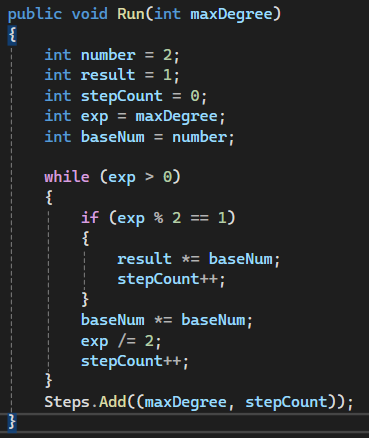


Рис 1.8.3.2 Код быстрого алгоритма возведения в степень

**8.4)** (QuickPow1) Классический быстрый алгоритм возведения в степень

(см. Рис. 1.8.4.1)

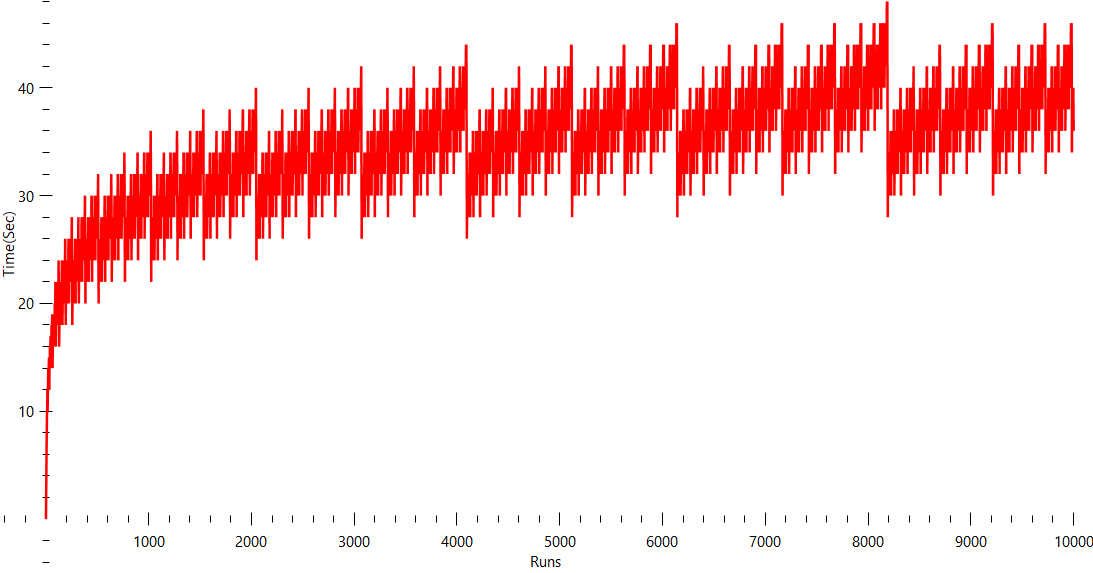


Рис 1.8.4.1 График классического быстрого алгоритма возведения в степень

График схож с предыдущем, но шагов требуется больше

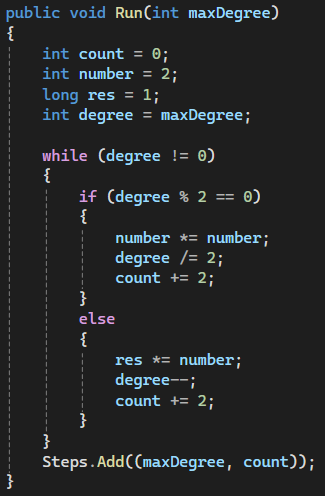


Рис 1.8.4.2 Код классического быстрого алгоритма сортировки

***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 = 300
* Ср. знач. на основе тестов: 3

Примечание: генерировались квадратные матрицы размером M \* M, умножение производилось прямым методом.

График зависимости времени выполнения перемножения двух заданных матриц от объема данных (см. рис. 2.1)

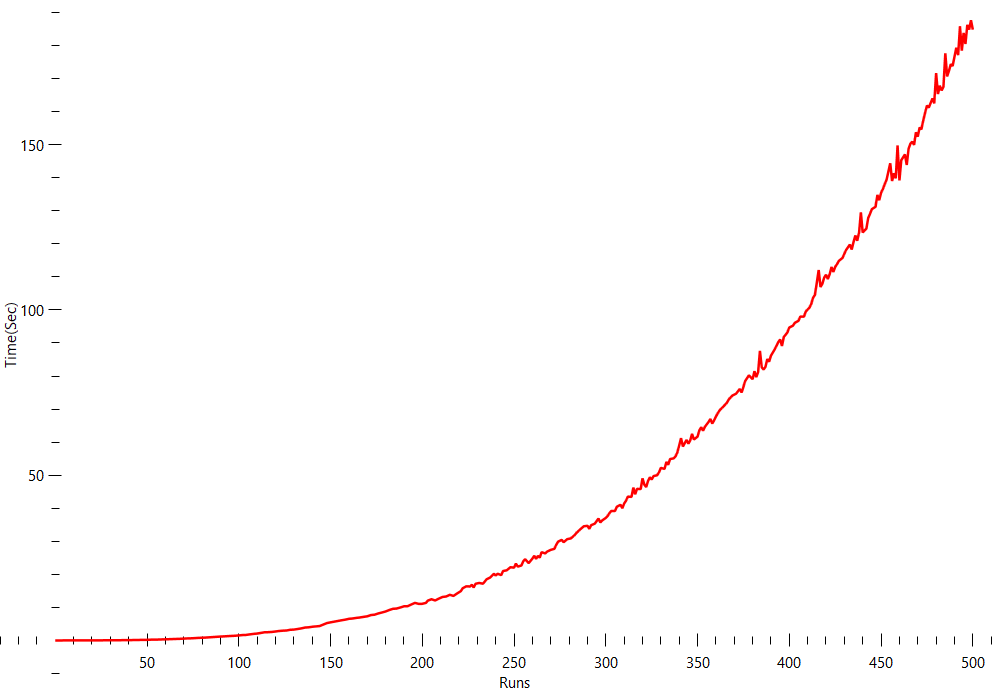


Рис. 2.1 Усреднённый график перемножения двух квадратных матриц

Код изображен на: (см. рис. 2.2)

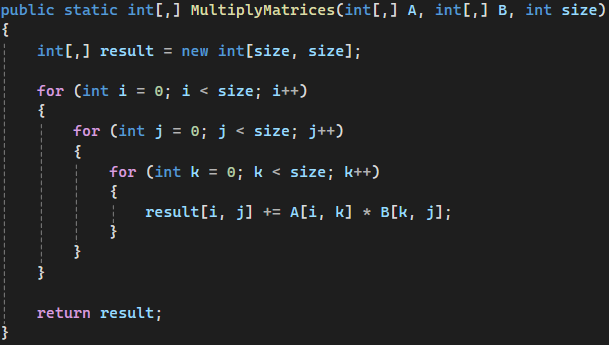


Рис. 2.2

Примечание: для генерации матриц использовался генератор рандомных матриц (см. рис. 2.3)

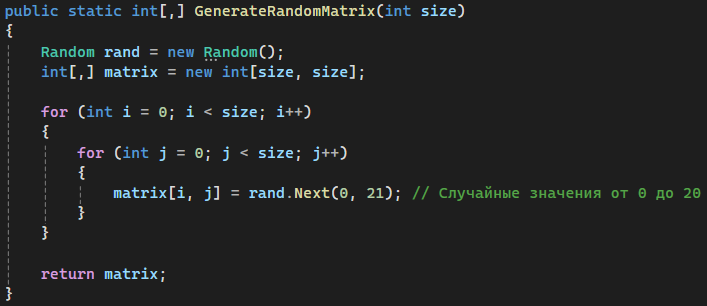


Рис. 2.3 Генератор рандомных матриц

***Задание III***

1. **OddEvenSort (Чет-нечет сортировка)**

* Временная сложность:

Худшее: O(n2)

Среднее: O(n2)

Лучшее: O(n)

* N = 5000
* Ср. знач. на основе тестов: 5

Производится многократный прогон по массиву, соседние элементы сравниваются и, в случае необходимости, меняются местами. В отличие от пузырьковой сортировки шаг по массиву равен двум, а не единице.

Сначала элементы с нечётными индексами сравниваются/обмениваются с элементами с чётными индексами (1-й со 2-м, 3-й с 4-м, 5-й с 6-м и т.д.). Затем элементы с чётными индексами сравниваются/обмениваются с соседними элементами с нечётными индексами (2-й с 3-м, 4-й с 5-м, 6-й с 7-м и т.д.). Затем снова нечётные сравниваются с чётными, потом снова чётные с нечётными и т.д.

Процесс завершается если в результате двух прогонов не происходило обменов, значит массив упорядочен.

График зависимости времени выполнения алгоритма от объема данных (см. рис. 3.1)

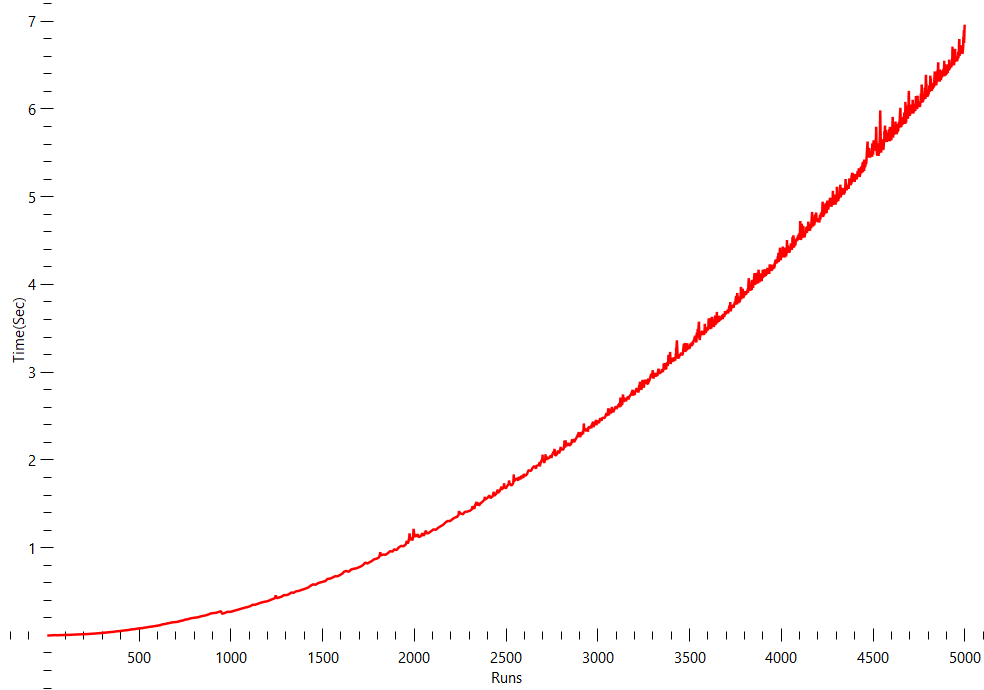


Рис. 3.1 Усреднённый график алгоритма OddEvenSort

Код алгоритма изображен на: (см. рис. 3.2)

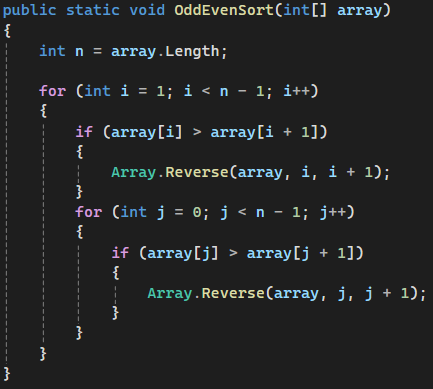


Рис. 3.2 Алгоритм OddEvenSort

1. **CombSort (Сортировка расческой)**

* Временная сложность: O(nlog2n)
* N = 5000
* Ср. знач. на основе тестов: 50

Comb sort (сортировка расческой) – идея работы алгоритма крайне похожа на сортировку обменом, но главным отличием является то, что сравниваются не два соседних элемента, а элементы на промежутке, к примеру, в пять элементов.

Это обеспечивает от избавления мелких значений в конце, что способствует ускорению сортировки в крупных массивах. Первая итерация совершается с шагом, рассчитанным по формуле (размер массива)/(фактор уменьшения), где фактор уменьшения равен приблизительно 1,2473…., или округлено до 1,3. Вторая и последующие итерации будут проходить с шагом (текущий шаг)/(фактор уменьшения) и будут происходить до тех пор, пока шаг не будет равен единице.

График зависимости времени выполнения от объема данных (см. рис. 3.3)

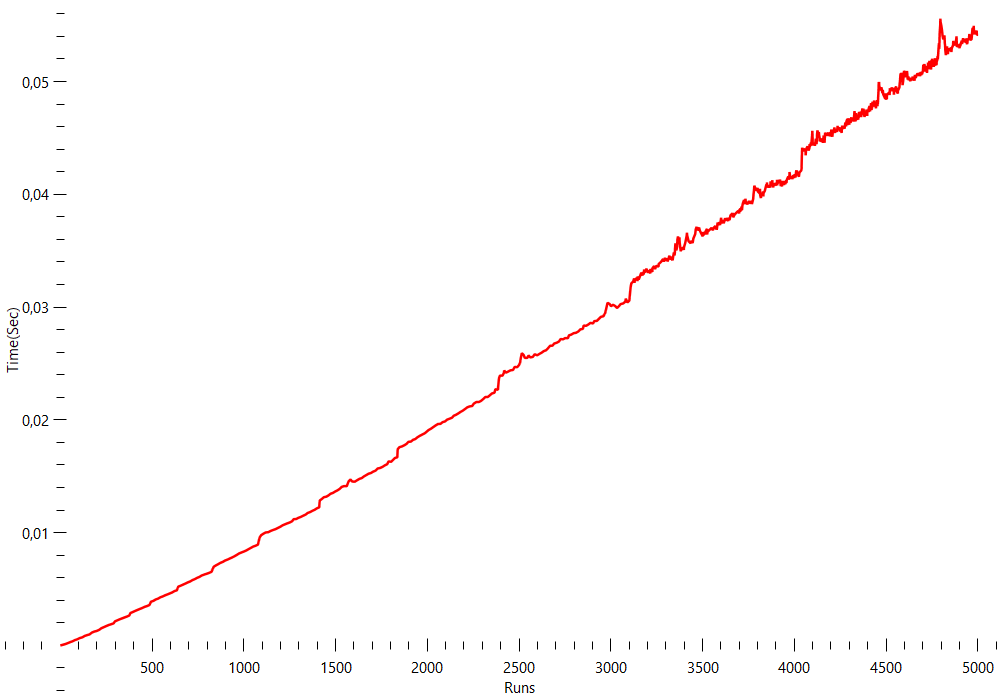


Рис. 3.3 Усреднённый график алгоритма CombSort

Код алгоритма изображен на: (см. рис. 3.4)

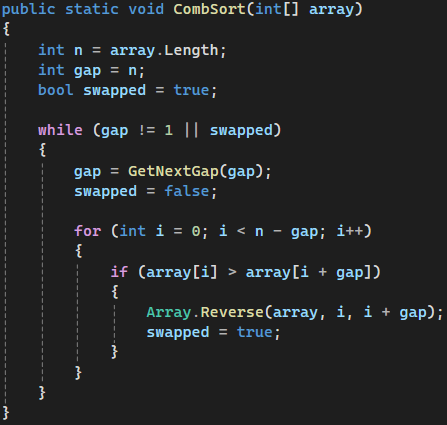


Рис. 3.4 Код алгоритма CombSort

Примечание: в сортировке используется вспомогательный метод GetNextGap (см. рис. 3.5)

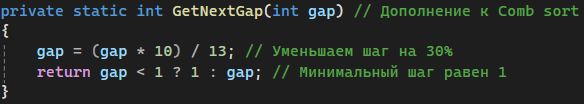


Рис. 3.5 вспомогательный метод GetNextGap

1. **SelectionSort (Сортировка выбором)**

* Временная сложность: O(n^2)
* N = 5000
* Ср. знач. на основе тестов: 5

Сортировка выбором (Selection Sort) – проходим по массиву в поисках минимального элемента. Найденный минимум меняем местами с первым элементом. Неотсортированная часть массива уменьшилась на один элемент (не включает первый элемент, куда мы переставили найденный минимум). К этой неотсортированной части применяем те же действия — находим минимум и ставим его на первое место в неотсортированной части массива. И так продолжаем до тех пор, пока неотсортированная часть массива не уменьшится до одного элемента.

График зависимости времени выполнения от объема данных (см. рис. 3.6)

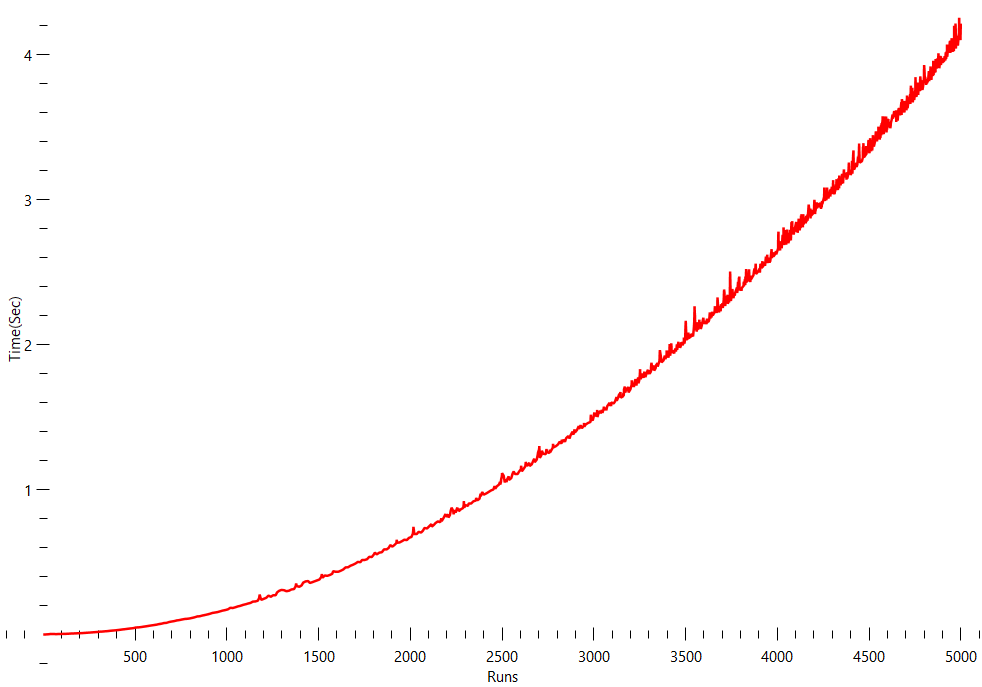


Рис 3.6 Усреднённый график алгоритма SelectionSort

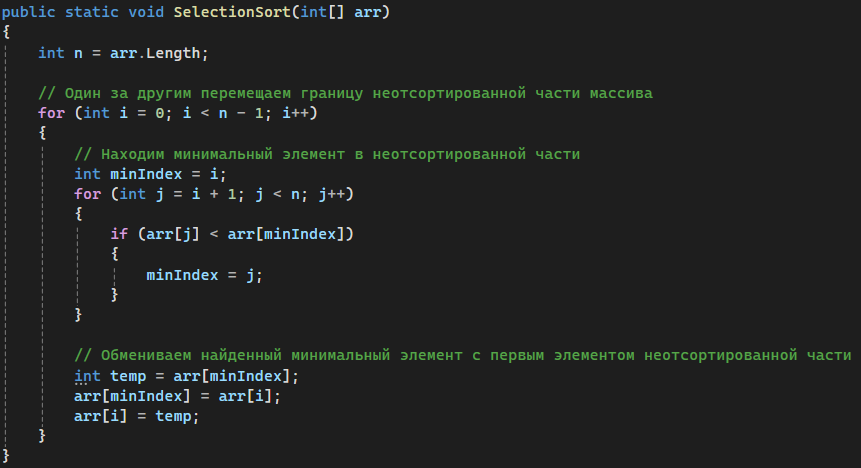


Рис. 3.7 Код алгоритма SelectionSort