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 Костюк С.А. При-201

подпись инициалы, фамилия группа

Дударов Д. При-201

подпись инициалы, фамилия группа

Хотенов Е. При-201

подпись инициалы, фамилия группа

Отчет защищен \_\_\_\_\_\_\_02.10.24\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

дата оценка

Челябинск 20\_24\_ г.

**Алгоритм 1: f(u)= 1 (постоянная функция)**

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 = 100 000
* Ср. знач. на основе тестов: 30

Вычисление факториала 100000. На практике видим, что время выполнения не зависите от вводимых данных.

Замеры времени от количества элементов (смотри рис 1.1):

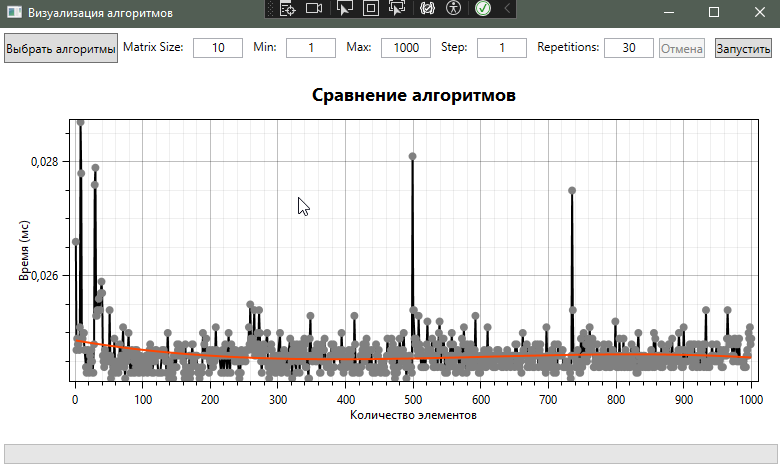


Рис. 1.1 Const алгоритм

Код изображен на Рис. 1.2:

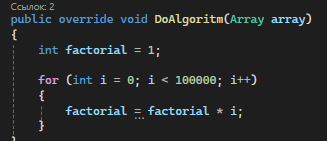


Рис. 1.2 Код алгоритма

**Алгоритм 2: f(v) = (сумма элементов)**

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

* f(v) = (сумма элементов)
* Временная сложность: O(n)
* n от 1 до 50000
* Шаг 10
* Ср. знач. на основе тестов: 5

Алгоритм суммирует все элементы массива. На практике видим линейную зависимость от размера массива.

Замеры времени от количества элементов (смотри рис 2.1):

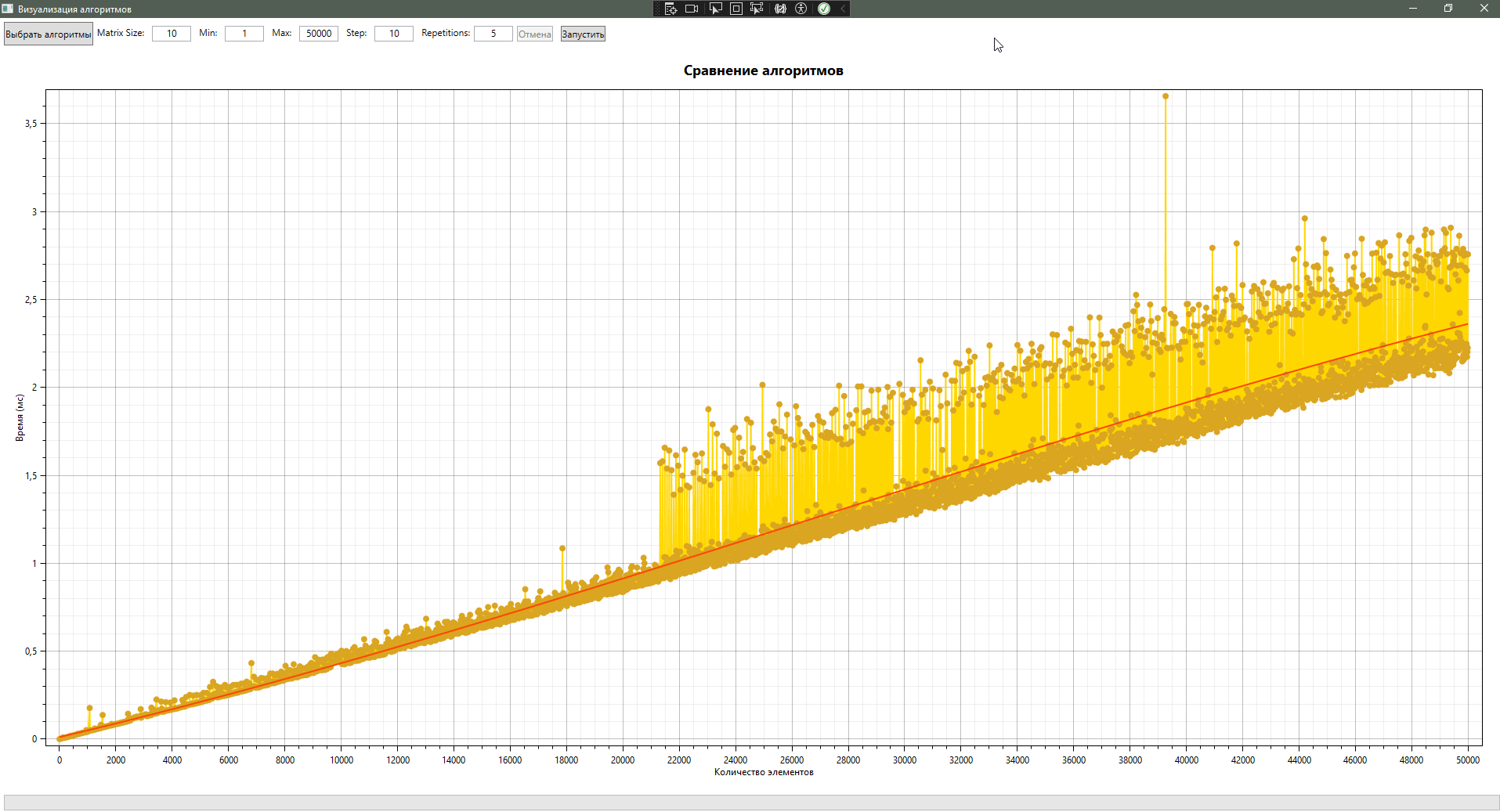


Рис. 2.1 Алгоритм суммы элементов

Код изображен на Рис. 2.2:

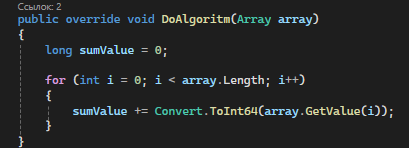


Рис. 2.2 Код алгоритма

**Алгоритм 3: (произведение элементов)**

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

* (произведение элементов)
* Временная сложность: O(n)
* n от 1 до 50000
* Шаг 10
* Ср. знач. на основе тестов: 5

Алгоритм перемножает все элементы массива. На практике видим линейную зависимость от размера массива.

Замеры времени от количества элементов (смотри рис 3.1):

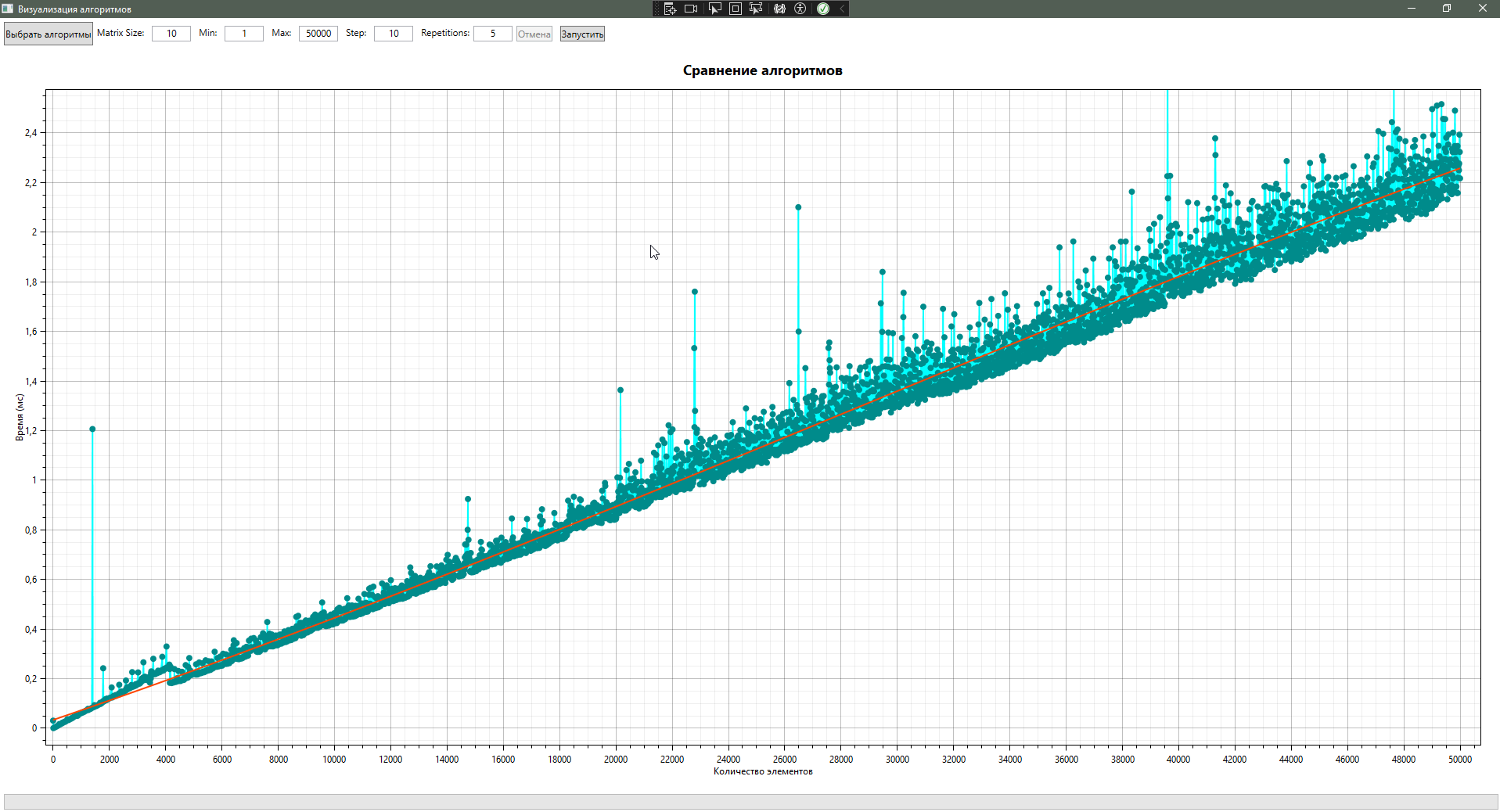


Рис. 3.1 Алгоритм произведения элементов

Код изображен на Рис. 3.2:

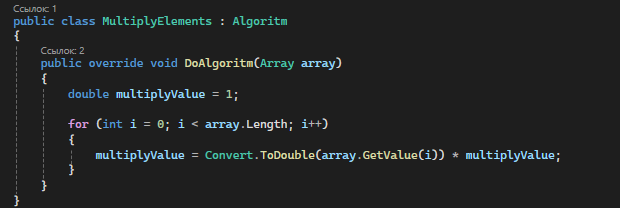


Рис. 2.2 Код алгоритма

**Сравнение алгоритмов постоянной функции, суммы и умножения элементов.**

На графике (смотри рис 3.3) проведена визуализация сравнения алгоритмов рассмотренных ранее. Отчётлива видна разница между алгоритмами временной сложности О(n) и алгоритмом временной сложности O(1).

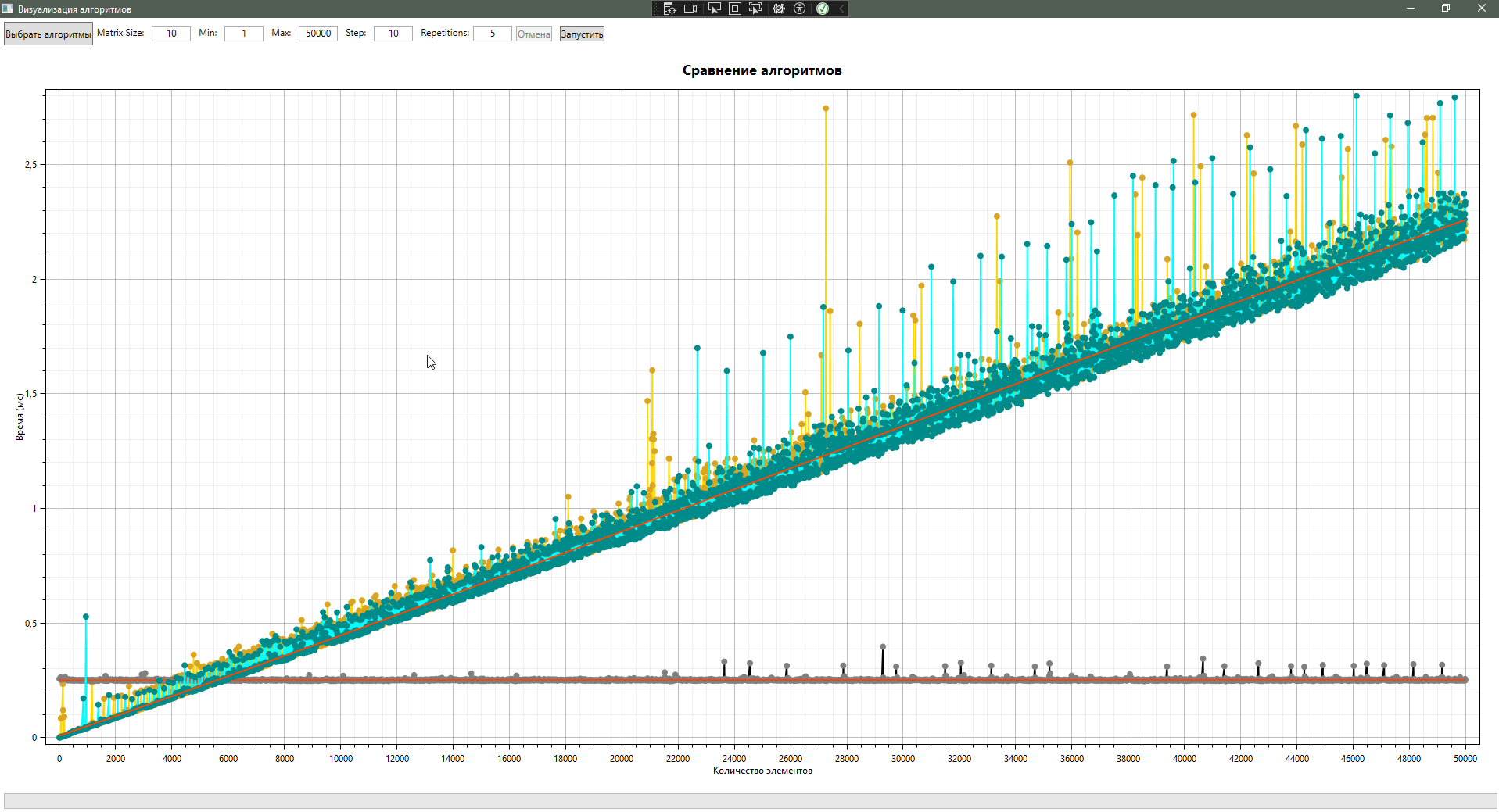


Рис. 3.3 сравнение линейных и постоянного алгоритмов

**Алгоритм 5: Сортировка пузырьком**

Этот алгоритм считается учебным и почти не применяется на практике из-за низкой эффективности: он медленно работает на тестах, в которых маленькие элементы (их называют «черепахами») стоят в конце массива.

Однако на нём основаны многие другие методы, например, шейкерная сортировка и сортировка расчёской.

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

* BubbleSort
* Временная сложность: O(n^2)
* n от 100 до 1000
* Шаг 10
* Ср. знач. на основе тестов: 5

Выполняется некоторое количество проходов по массиву — начиная от начала массива, перебираются пары соседних элементов массива. Если 1-й элемент пары больше 2-го, элементы переставляются (выполняется обмен).

Пары элементов массива перебираются (проходы по массиву повторяются) либо (n-1) раз, либо до тех пор, пока на очередном проходе не обнаружится, что более не требуется выполнять перестановки (обмены) (массив отсортирован).

При каждом проходе алгоритма по внутреннему циклу очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива (как бы «всплывает» до нужной позиции, как пузырёк в воде — откуда и название алгоритма).

Замеры времени от количества элементов (смотри рис 5.1):

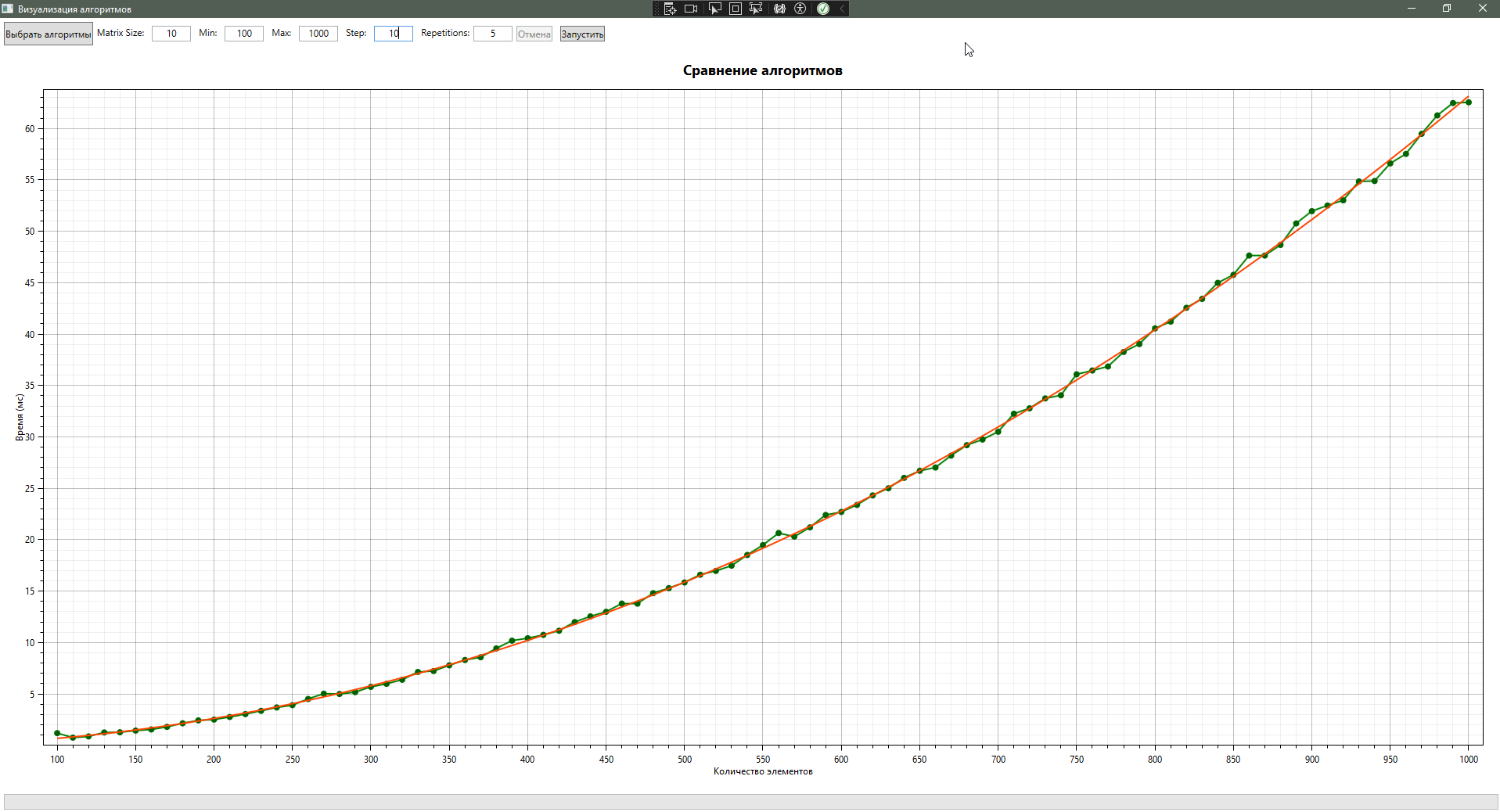


Рис. 5.1 Алгоритм сортировки пузырьком

Код изображен на Рис. 5.2:

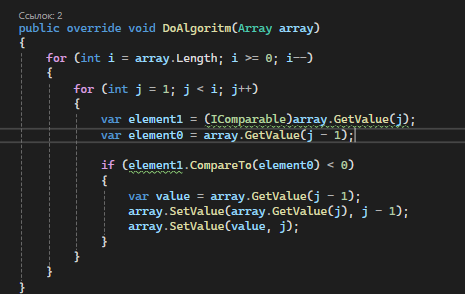


Рис. 5.2 Код алгоритма

**Алгоритм 7: Сортировка TimSort**

Timsort, в отличии от всяких там «пузырьков» и «вставок», штука относительно новая — изобретен был в 2002 году Тимом Петерсом (в честь него и назван). С тех пор он уже стал стандартным алгоритмом сортировки в Python, OpenJDK 7 и Android JDK 1.5.

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

* TimSort
* Временная сложность: O(n\*log(n))
* n от 100 до 10000
* Шаг 10
* Ср. знач. на основе тестов: 5

Работу алгоритма можно разделить на следующие шаги:

1. определение минимального размера подмассива массива;
2. деление входного массива на подмассивы с использованием специального алгоритма;
3. сортировка каждого подмассива с использованием алгоритма сортировки вставками;
4. объединение отсортированных подмассивов в массив с использованием изменённого алгоритма сортировки слиянием.

Особенности алгоритма заключаются в особенностях алгоритма деления массива на подмассивы и в особенностях изменённого алгоритма сортировки слиянием.

Замеры времени от количества элементов (смотри рис 7.1):

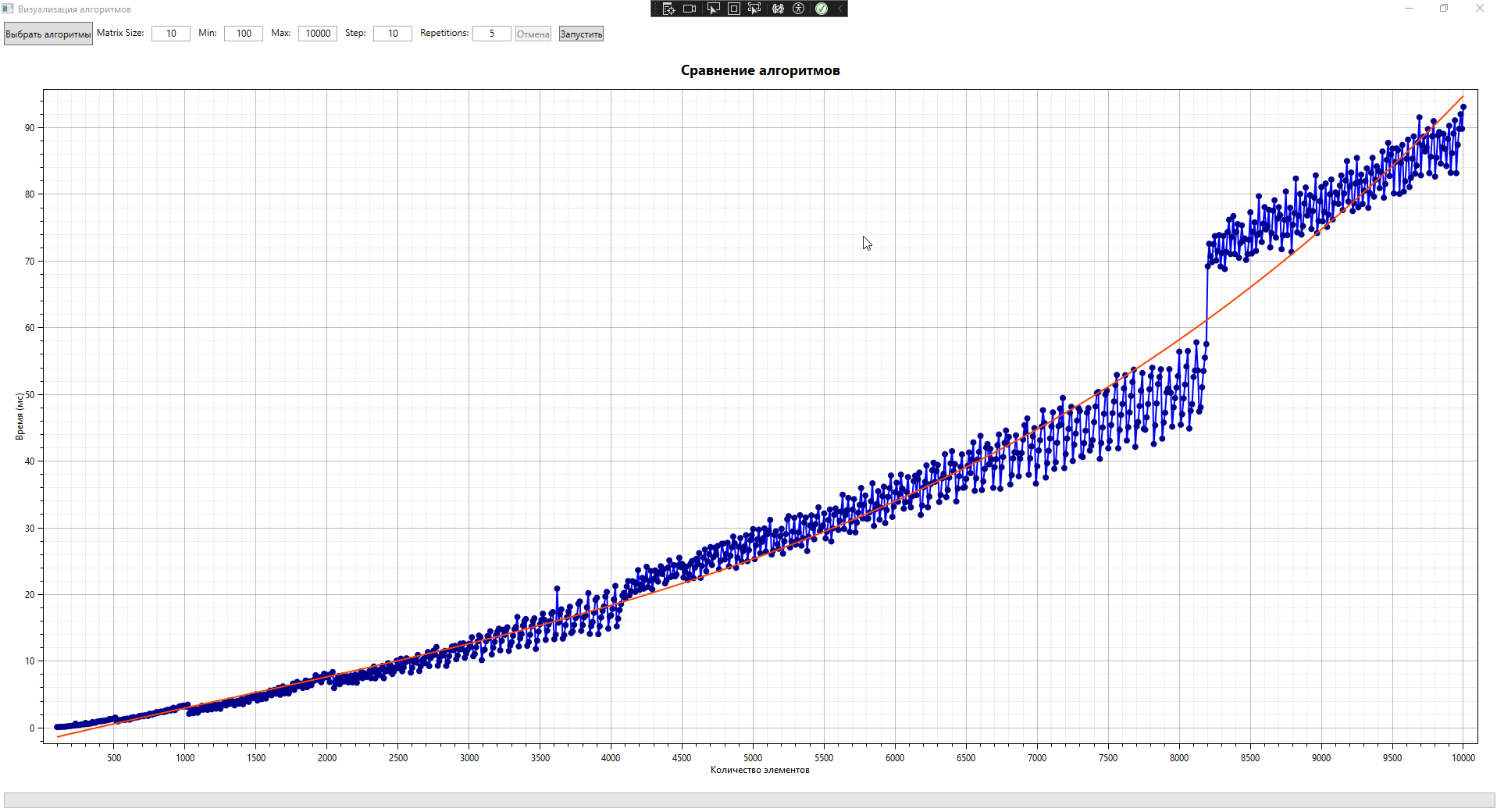


Рис. 7.1 Алгоритм сортировки TimSort

Рассмотрим подробнее шаги алгоритма.

Шаг 1. Вычисление minrun.

Число minrun определяется на основе n исходя из следующих принципов:

Оно не должно быть слишком большим, поскольку к подмассиву размера minrun будет в дальнейшем применена сортировка вставками, а она эффективна только на небольших массивах

Оно не должно быть слишком маленьким, поскольку чем меньше подмассив — тем больше итераций слияния подмассивов придётся выполнить на последнем шаге алгоритма.

Хорошо бы, чтобы n \ minrun было степенью числа 2 (или близким к нему). Это требование обусловлено тем, что алгоритм слияния подмассивов наиболее эффективно работает на подмассивах примерно равного размера.

В этом месте автор алгоритма ссылается на собственные эксперименты, показавшие, что при minrun> 256 нарушается пункт 1, при minrun < 8 — пункт 2 и наиболее эффективно использовать значения из диапазона (32;65). Исключение — если N < 64, тогда minrun = n и timsort превращается в простую сортировку вставкой. В данный момент алгоритм расчёта minrun просто до безобразия: берём старшие 6 бит из N и добавляем единицу, если в оставшихся младших битах есть хотя бы один ненулевой.

Код выглядит так (смотри рис 7.2):

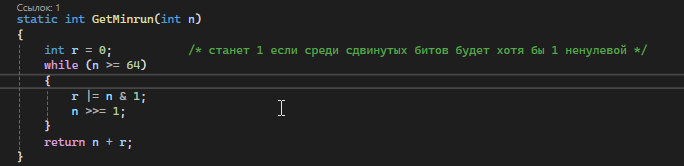


Рис. 7.2 Код вычисления minrun

Шаг 2. Разбиение на подмассивы и их сортировка.

Итак, на данном этапе у нас есть входной массив, его размер N и вычисленное число minrun. Алгоритм работы этого шага:

1. Ставим указатель текущего элемента в начало входного массива.
2. Начиная с текущего элемента, ищем во входном массиве run (упорядоченный подмассив). По определению, в этот run однозначно войдет текущий элемент и следующий за ним, а вот дальше — уже как повезет. Если получившийся подмассив упорядочен по убыванию — переставляем элементы так, чтобы они шли по возрастанию (это простой линейный алгоритм, просто идём с обоих концов к середине, меняя элементы местами).
3. Если размер текущего run'а меньше чем minrun — берём следующие за найденным run-ом элементы в количестве minrun — size(run). Таким образом, на выходе у нас получается подмассив размером minrun или больше, часть которого (а в идеале — он весь) упорядочена.
4. Применяем к данному подмассиву сортировку вставками (смотри рис 7.4). Так как размер подмассива невелик и часть его уже упорядочена — сортировка работает быстро и эффективно.
5. Ставим указатель текущего элемента на следующий за подмассивом элемент.
6. Если конец входного массива не достигнут — переход к пункту 2, иначе — конец данного шага.

Основной код алгоритма:

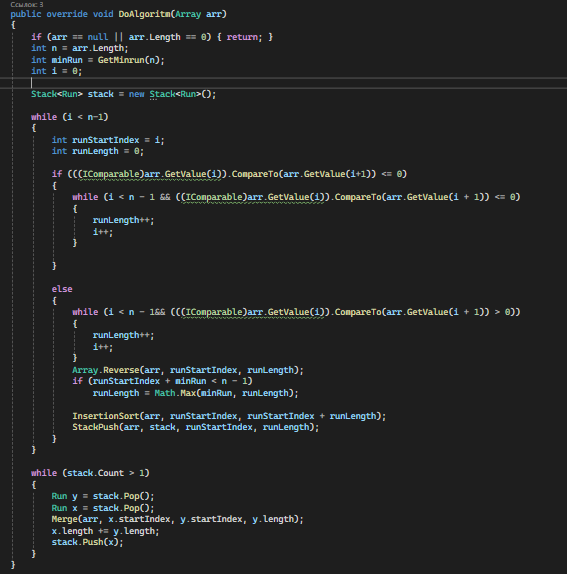


Рис. 7.3 Основной код алгоритма

Шаг 3 сортировка вставками:

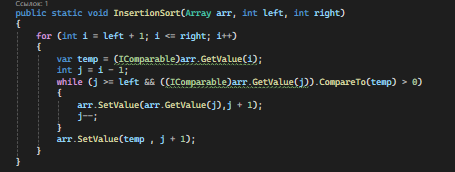


Рис. 7.4 Алгоритм сортировки вставками

Шаг 4. Слияние.

На данном этапе у нас имеется входной массив, разбитый на подмассивы, каждый из которых упорядочен. Если данные входного массива были близки к случайным — размер упорядоченных подмассивов близок к minrun, если в данных были упорядоченные диапазоны (а исходя из рекомендаций по применению алгоритма, у нас есть основания на это надеяться) — упорядоченные подмассивы имеют размер, превышающий minrun.

Теперь нам нужно объединить эти подмассивы для получения результирующего, полностью упорядоченного массива. Причём по ходу этого объединения нужно выполнить 2 требования:

Объединять подмассивы примерно равного размера (так получается эффективнее).

Сохранить стабильность алгоритма — т.е. не делать бессмысленных перестановок (например, не менять два последовательно стоящих одинаковых числа местами).

Достигается это таким образом.

Создаем пустой стек пар <индекс начала подмассива>-<размер подмассива>. Берём первый упорядоченный подмассив.

Добавляем в стек пару данных <индекс начала>-<размер> для текущего подмассива.

Определяем, нужно ли выполнять процедуру слияния текущего подмассива с предыдущими. Для этого проверяется выполнение 2 правил (пусть X, Y и Z — размеры трёх верхних в стеке подмассивов):

X > Y + Z

Y > Z

Если одно из правил нарушается — массив Y сливается с меньшим из массивов X и Z. Повторяется до выполнения обоих правил или полного упорядочивания данных.

Если еще остались не рассмотренные подмассивы — берём следующий и переходим к пункту 2. Иначе — конец.

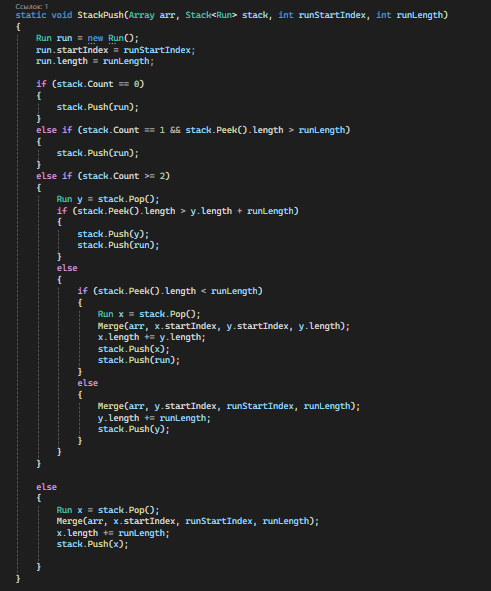
Код работы со стеком: 

Рис. 7.5 Код стека

Процедура слияния подмассивов

Как Вы помните, на втором шаге алгоритма мы занимаемся слиянием двух подмассивов в один упорядоченный. Мы всегда соединяем 2 последовательных подмассива. Для их слияния используется дополнительная память.

1. Создаём временный массивы в размере ранов.
2. Копируем раны в временный массив.
3. Ставим указатели текущей позиции на первые элементы временных массивов.
4. На каждом следующем шаге рассматриваем значение текущих элементов в временных массивах, берём меньший из них и копируем его в новый отсортированный массив. Перемещаем указатель текущего элемента в массиве, из которого был взят элемент.
5. Повторяем 4, пока один из массивов не закончится.
6. Добавляем все элементы оставшегося массива в конец нового массива.

Код слияния подмассивов:

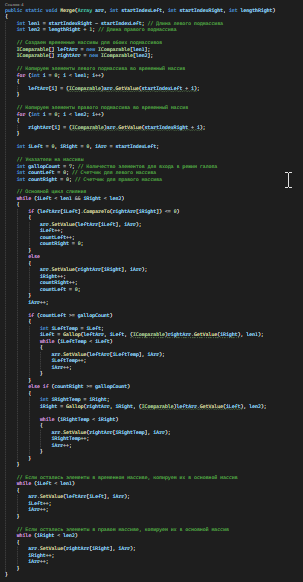


Рис. 7.6 Код слияния подмассивов.

Модификация Галоп. Суть в следующем:

1. Начинаем процедуру слияния, как было показано выше.
2. На каждой операции копирования элемента из временного или большего подмассива в результирующий запоминаем, из какого именно подмассива был элемент.
3. Если уже некоторое количество элементов (в данной реализации алгоритма это число жестко равно 7) было взято из одного и того же массива — предполагаем, что и дальше нам придётся брать данные из него. Чтобы подтвердить эту идею, мы переходим в режим «галопа», т.е. бежим по массиву-претенденту на поставку следующей большой порции данных бинарным поиском (мы помним, что массив упорядочен и мы имеем полное право на бинарный поиск) текущего элемента из второго соединяемого массива. Бинарный поиск эффективнее линейного, а потому операций поиска будет намного меньше.
4. Найдя, наконец, момент, когда данные из текущего массива-поставщика нам больше не подходят (или дойдя до конца массива), мы можем, наконец, скопировать их все разом (что может быть эффективнее копирования одиночных элементов).

Код Галлоа:

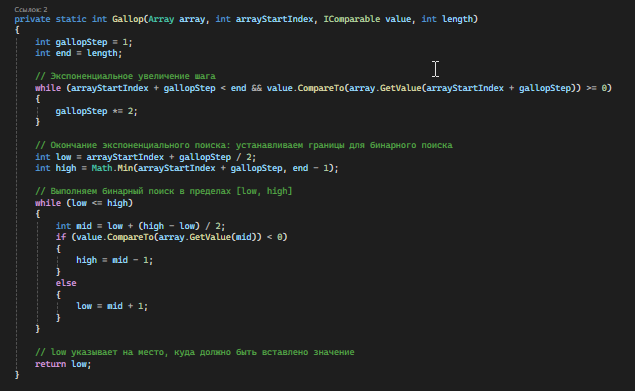


Рис. 7.7 Код галопа.

**Сравнение алгоритмов сортировки.**

На графике (смотри рис 7.8) проведена визуализация сравнения алгоритмов сортировки рассмотренных ранее. Отчётлива видна разница между алгоритмами временной сложности О(n^2) и алгоритмом временной сложности O(n\* log(n)).

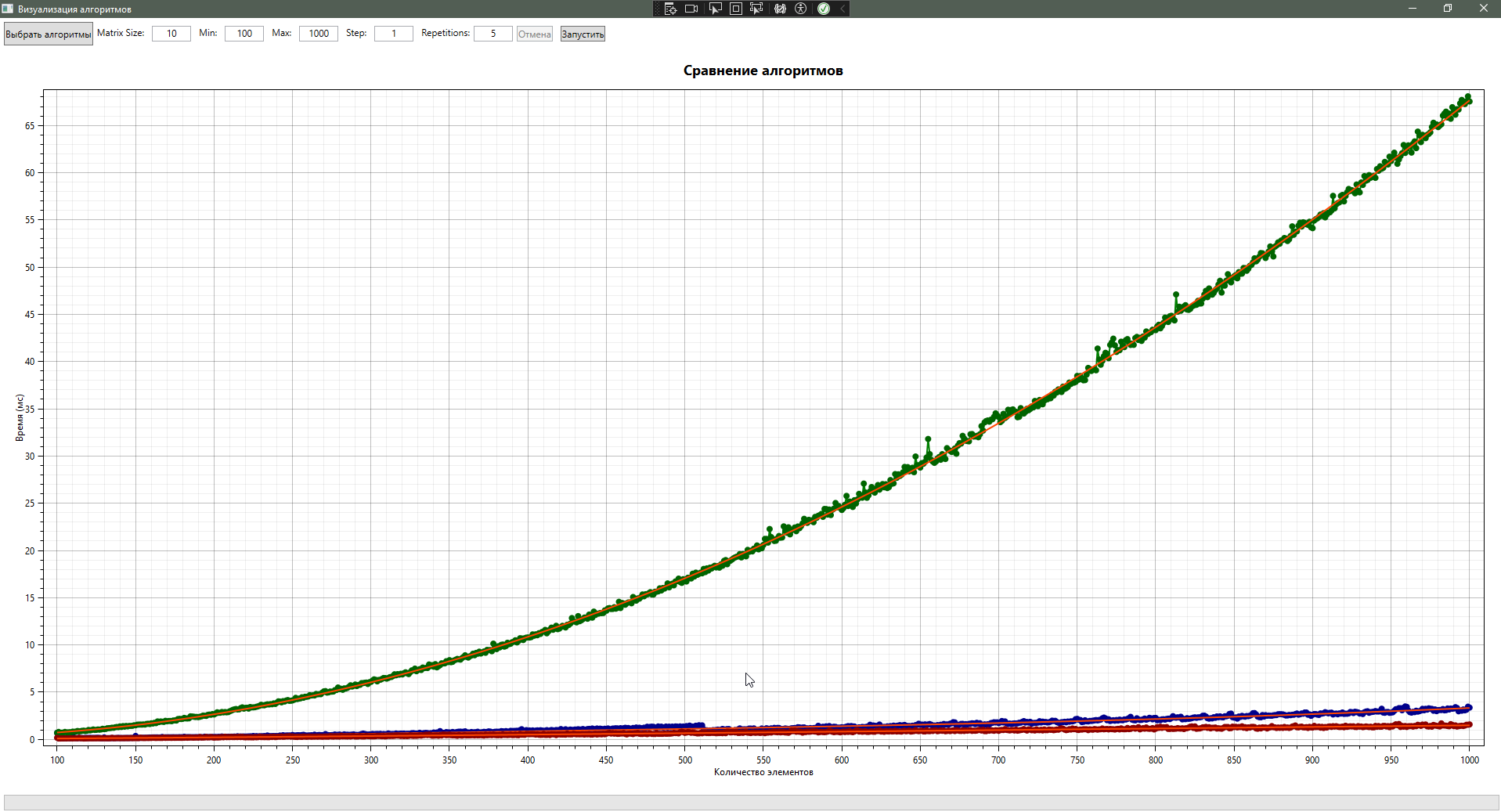


Рис. 7.8 Сравнение алгоритмов сортировки.