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.В. При-201

подпись инициалы, фамилия группа

Хотенов Е.С. 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):

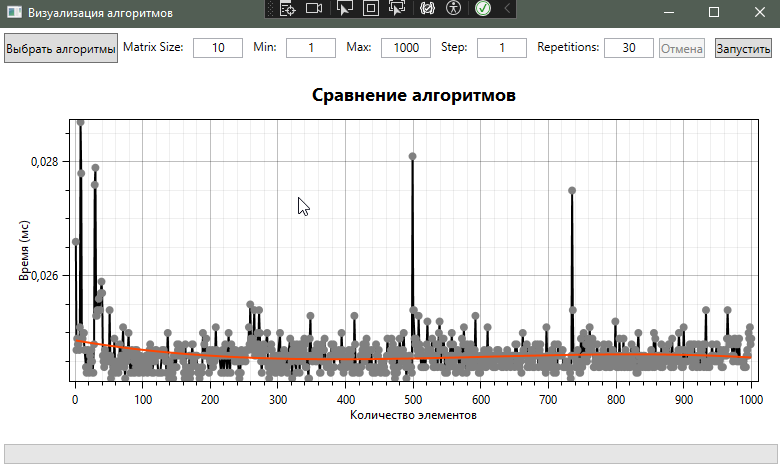


Рис. 1.1 Const алгоритм

Код изображен на Рис. 1.2:

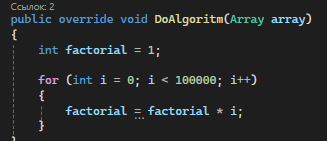


Рис. 1.2 Код алгоритма

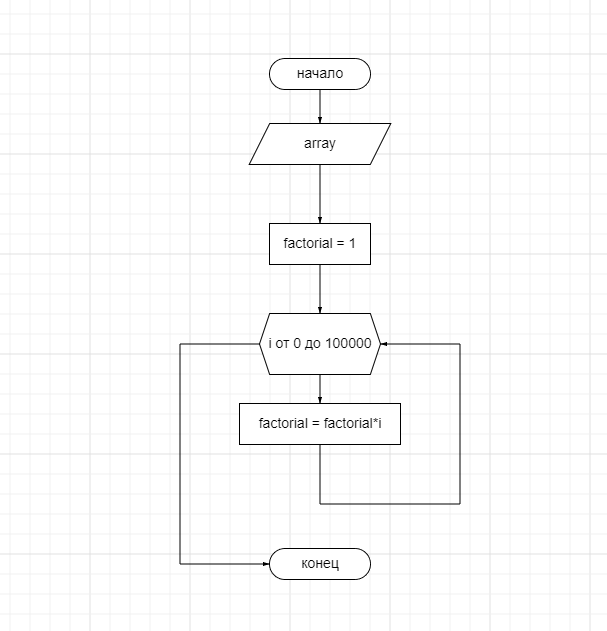


Рис. 1.3 БСА

**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):

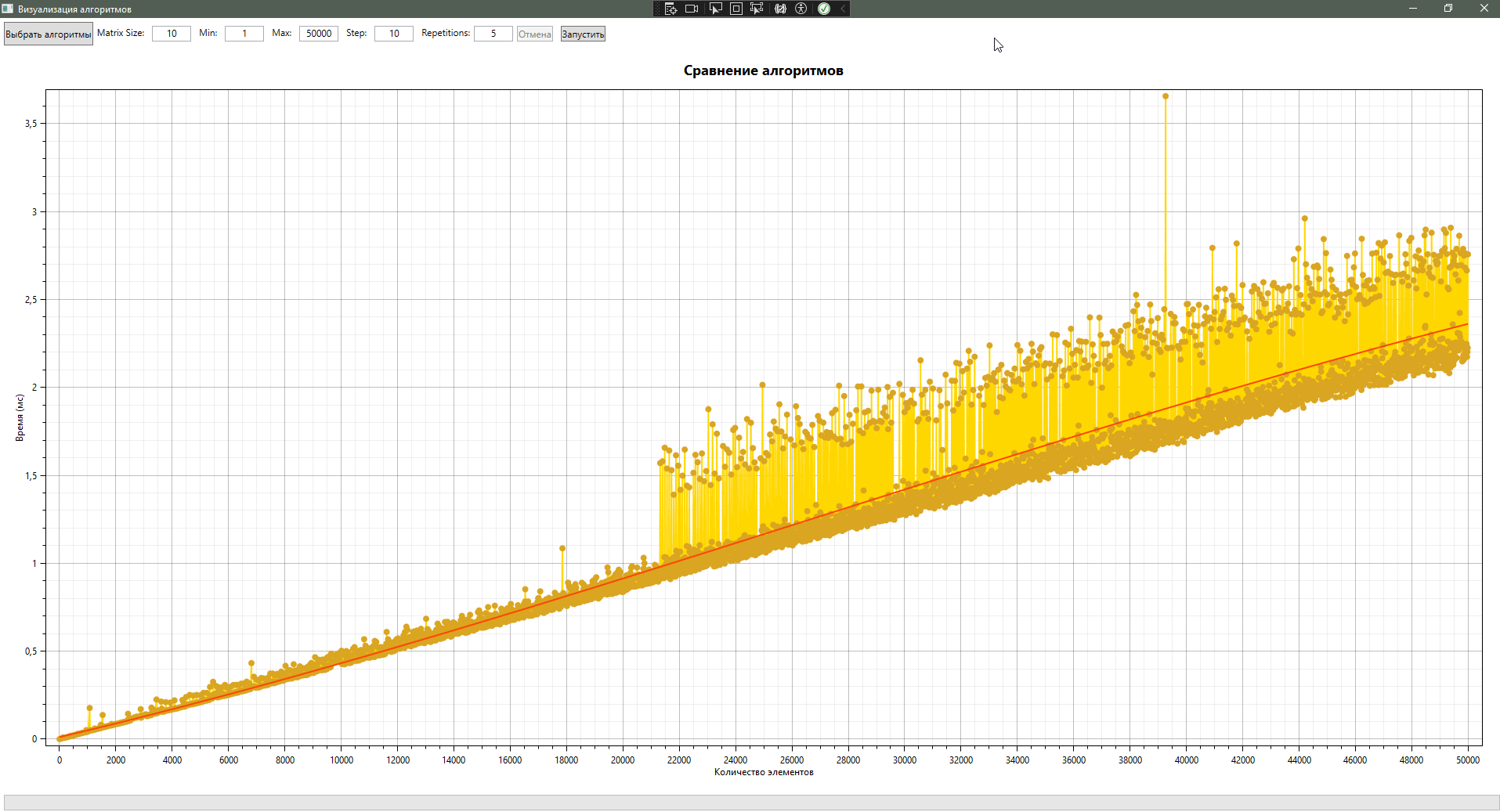


Рис. 2.1 Алгоритм суммы элементов

Код изображен на Рис. 2.2:

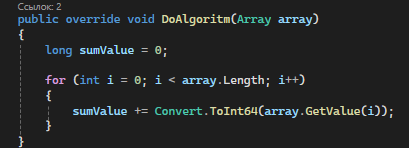


Рис. 2.2 Код алгоритма

Блок-схема алгоритма изображена на рисунке 2.3

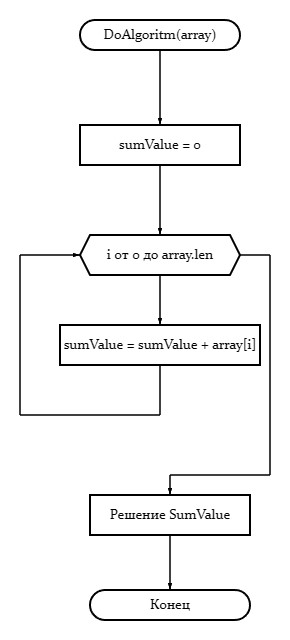


Рис. 2.3 Блок-схема алгоритма

**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):

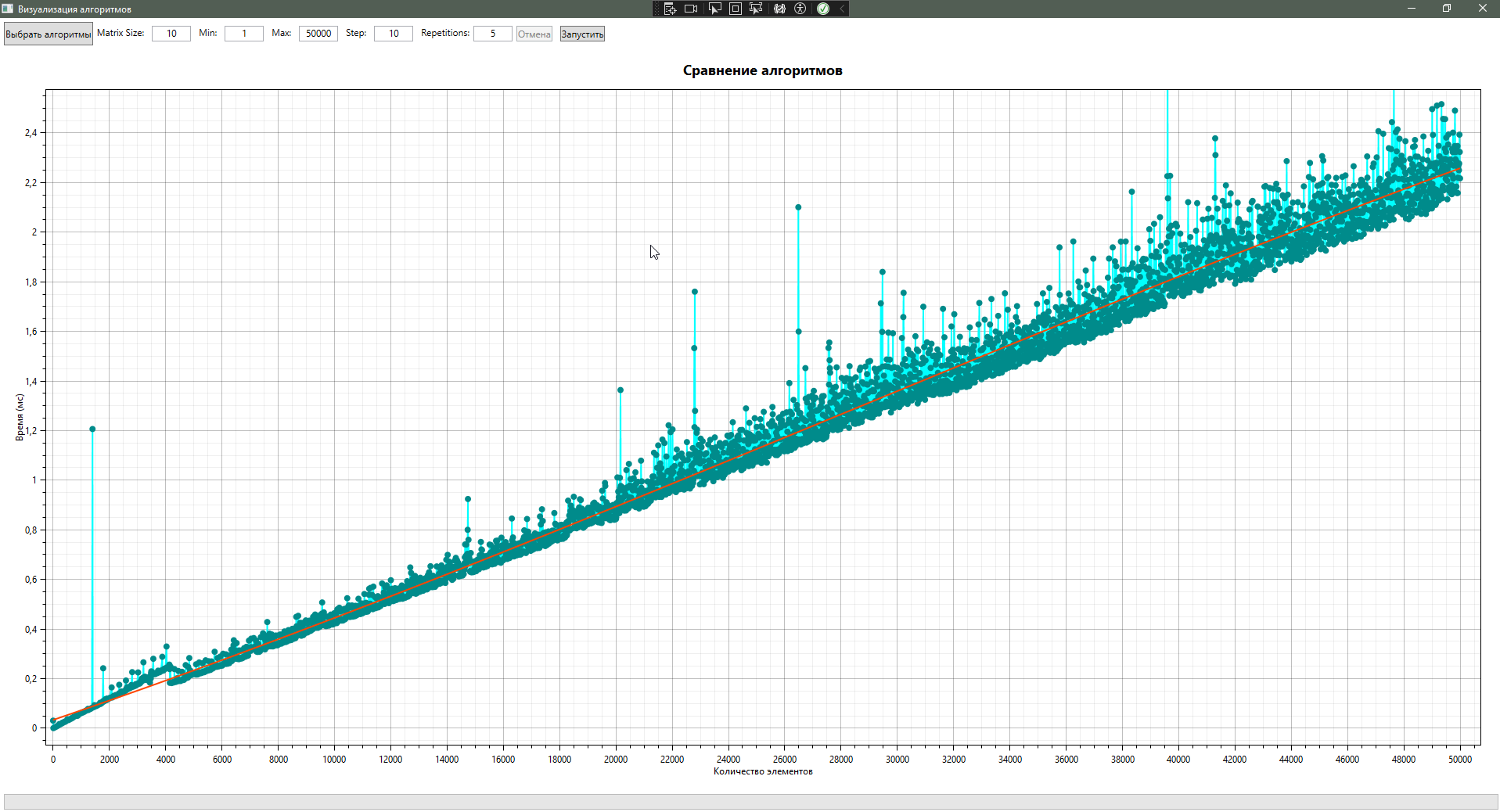


Рис. 3.1 Алгоритм произведения элементов

Код изображен на Рис. 3.2:

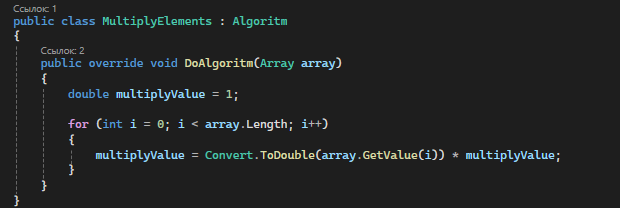


Рис. 3.2 Код алгоритма

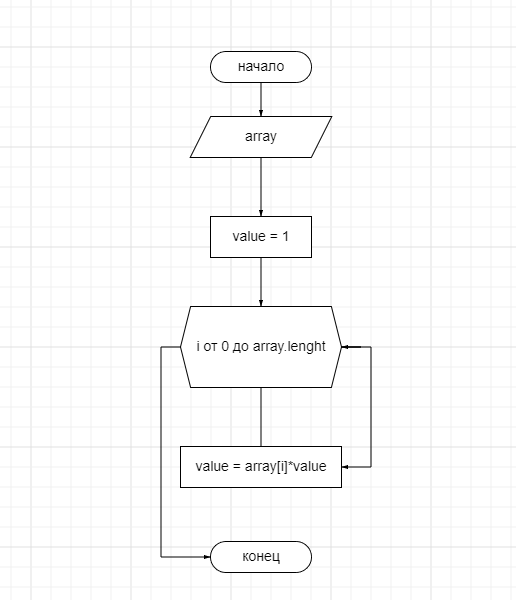


Рис. 3.3 Код алгоритма

**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).

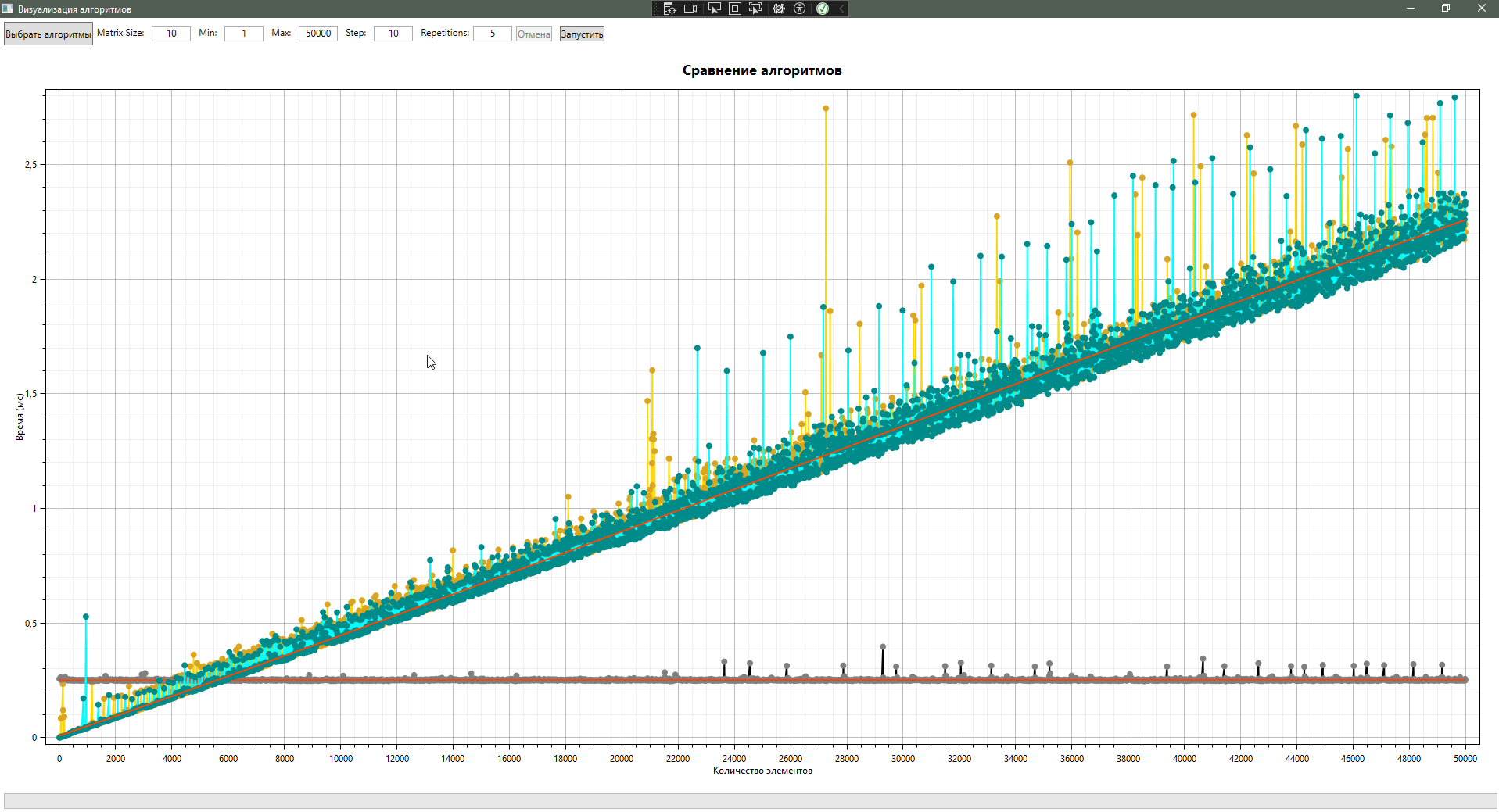


Рис. 3.4 сравнение линейных и постоянного алгоритмов

**Алгоритм 4.1: Прямое вычисление многочлена P степени n – 1 для x = 1,5**

Данный алгоритм считается довольно медленным из-за своей оценки каждого элемента по отдельности, он низкоэффективен и требует большей вычислительной мощности, но при этом является простым в реализации и достаточно наглядным. Суть алгоритма заключается в вычислении каждого члена: для каждого коэффициента вычисляется произведение этого коэффициента и соответствующей степени переменной, суммировании членов: все вычисленные члены суммируются, чтобы получить значение полинома в заданной точке.

* Прямое вычисление многочлена
* Временная сложность O()
* n от 100 до 1000
* Шаг 10
* Ср. знач. на основе тестов: 5

Замеры времени от количества элементов. (смотри рис. 4.1.1)

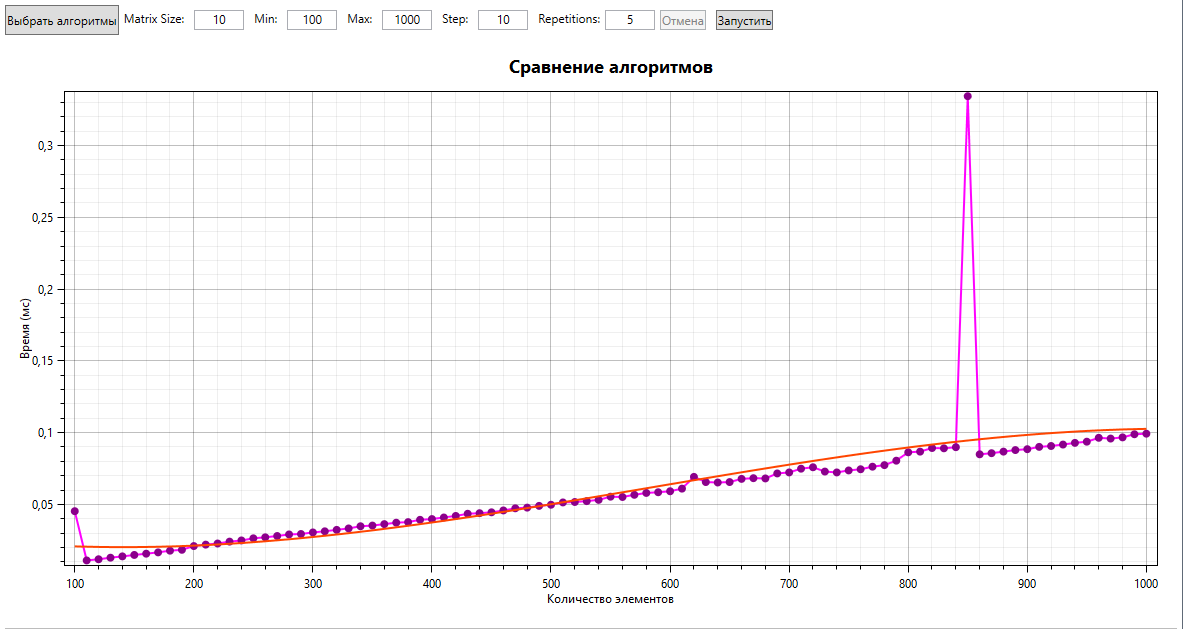


Рис. 4.1.1 Прямое вычисление многочлена

Код изображен на рисунке 4.1.2

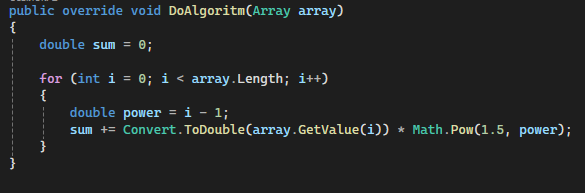


Рис. 4.1.2 Код алгоритма

Блок-схема алгоритма изображена на рисунке 4.1.3

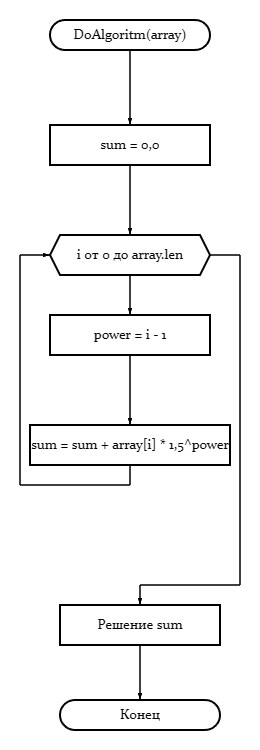


Рис. 4.1.3 блок-схема алгоритма

**Алгоритм 4.2: Вычисление многочлена P степени n-1 методом Горнера для х = 1,5**

Метод Горнера для вычисления полинома в заданной точке является

более эффективным, так как позволяет минимизировать количество операций умножения и сложения, но при этом метод Горнера сложнее в реализации. **Описание метода Горнера:**

1. Нахождение старшего коэффициента

2. Для каждого следующего коэффициента: умножаем текущее значение на переменную х, прибавляем следующий коэффициент

3. Повторяем шаги 1-2 для всех коэффициентов, начиная от старшего и заканчивая младшим.

В конце данного алгоритма мы получим значение полинома в заданной точке.

* Вычисление многочлена методом Горнера
* Временная сложность O(n)
* n от 100 до 1000
* Шаг 10
* Ср. знач. на основе тестов: 5

Замеры времени от количества элементов. (смотри рис. 4.2.1)

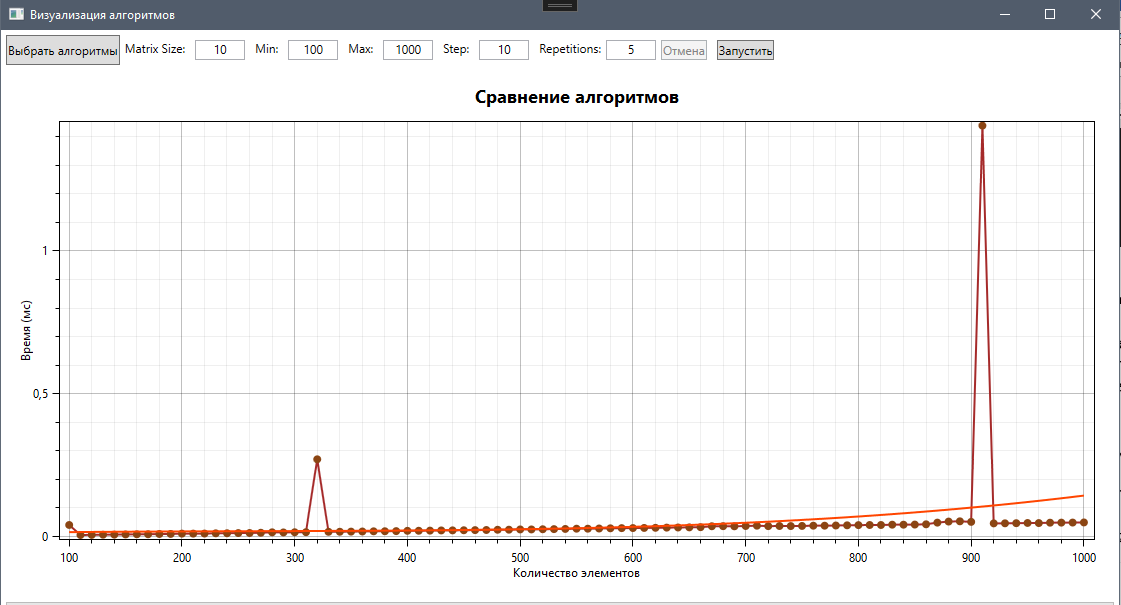


Рис. 4.2.1 Метод горнера

Код изображен на рисунке 4.2.2

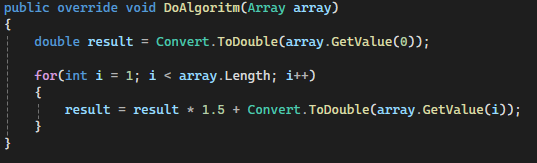


Рис. 4.2.2

Блок-схема алгоритма изображена на рисунке 4.2.3

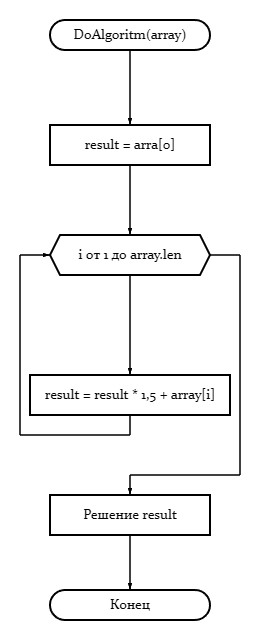


Рис. 4.2.3 Блок-схема алгоритма

**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):

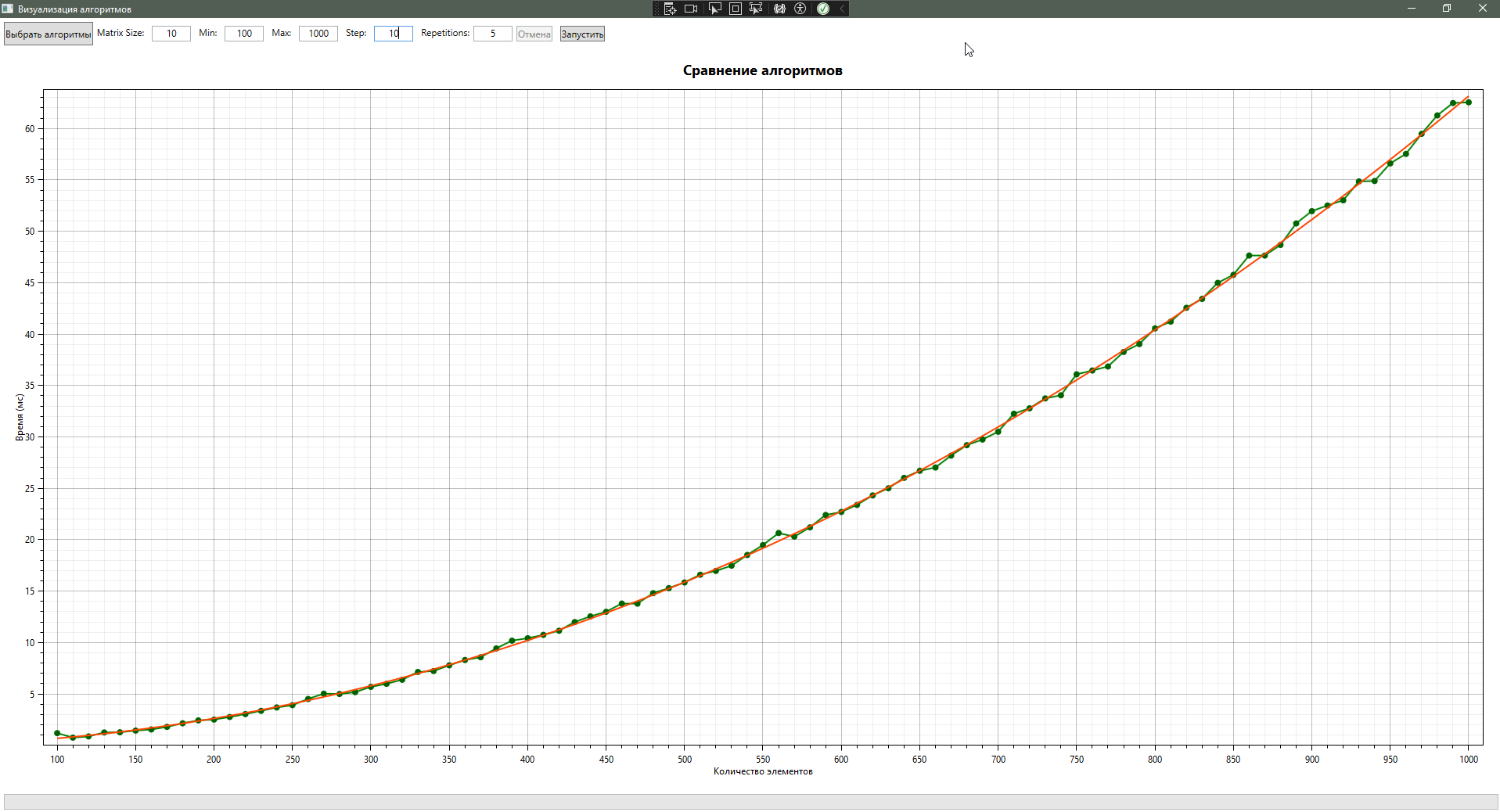


Рис. 5.1 Алгоритм сортировки пузырьком

Код изображен на Рис. 5.2:

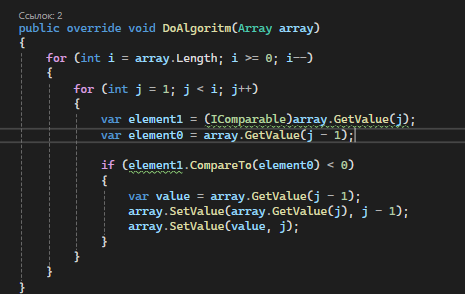


Рис. 5.2 Код алгоритма

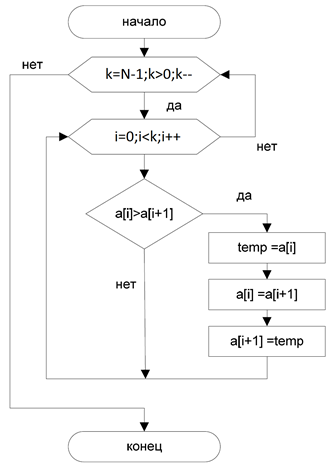


Рис. 5.3 БСА

**Алгоритм 6: Сортировка QuickSort**

Quick sort - это один из самых популярных и эффективных алгоритмов сортировки, который использует стратегию "разделяй и властвуй". Он работает путем разделения массива на подмассивы, которые затем сортируются отдельно.

**Описание работы QuickSort.**

1. Выбор опорного элемента (pivot): выбирается опорный элемент из массива. Это может быть первый элемент, последний элемент, случайный элемент или элемент, выбранный по какому-либо другому правилу.

2. Разделение массива: Массив разделяется на два подмассива: один с элементами, меньшими опорного, и другой с элементами, большими опорного. Опорный элемент занимает свое окончательное место в отсортированном массиве.

3. Рекурсивная сортировка: Процесс повторяется рекурсивно для каждого подмассива, пока не будут отсортированы все элементы.

* QuickSort
* Временная сложность: O(n \* log(n))
* n от 100 до 1000
* Шаг 10
* Ср. знач. на основе тестов: 5

Замеры времени от количества элементов (смотри рис. 6.1)

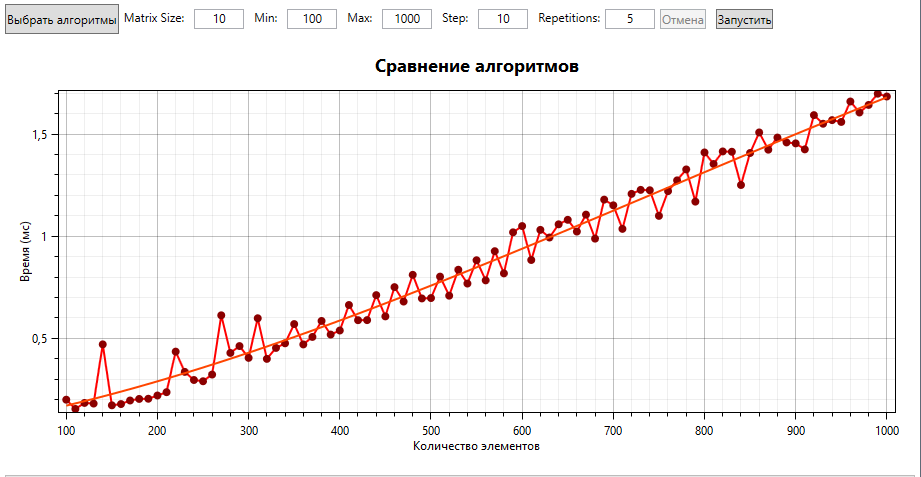


Рис. 6.1 QuckSort

Код изображен на рисунках 6.2 – 6.4

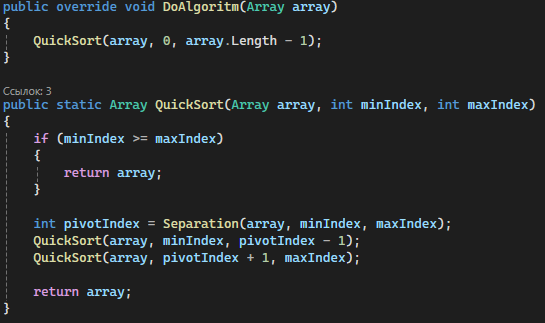


Рис. 6.2 Код алгоритма

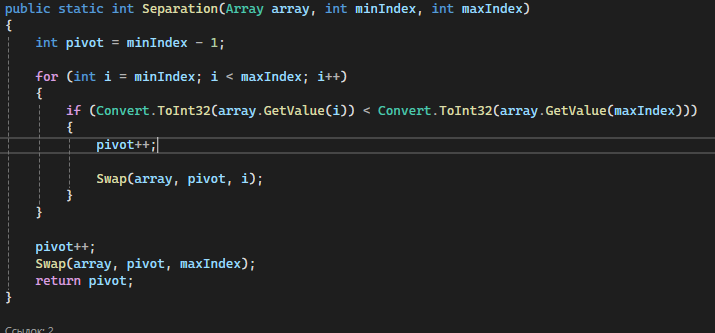


Рис. 6.3 Метод разделения

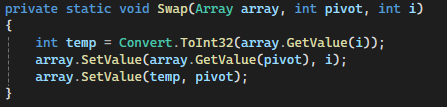


Рис. 6.4 Метод для перестановки элементов

Блок схемы алгоритма и вспомогательных методов изображены на рисунках 6.5-6.7

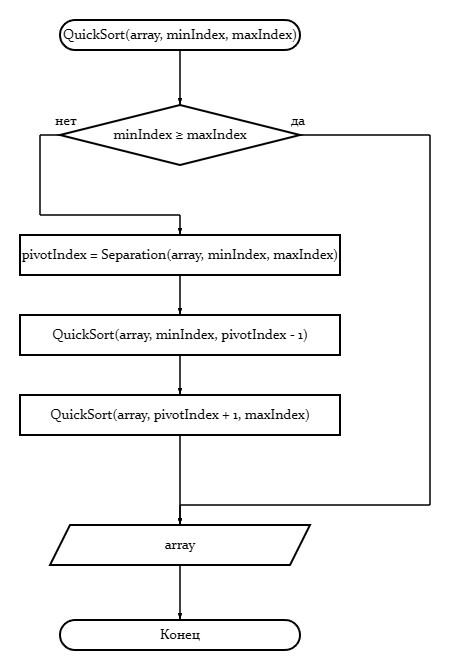


Рис.6.5 Блок-схема алгоритма

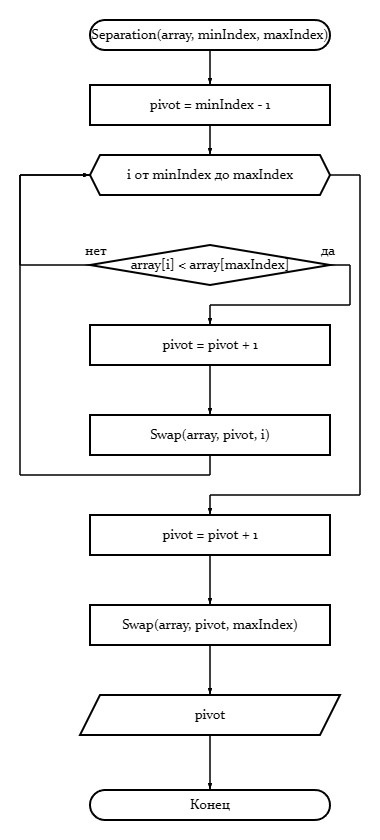


Рис 6.6 Блок-схема вспомогательного метода Separation

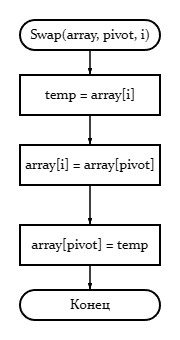


Рис.6.7 Блок-схема вспомогательного метода Swap

**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):

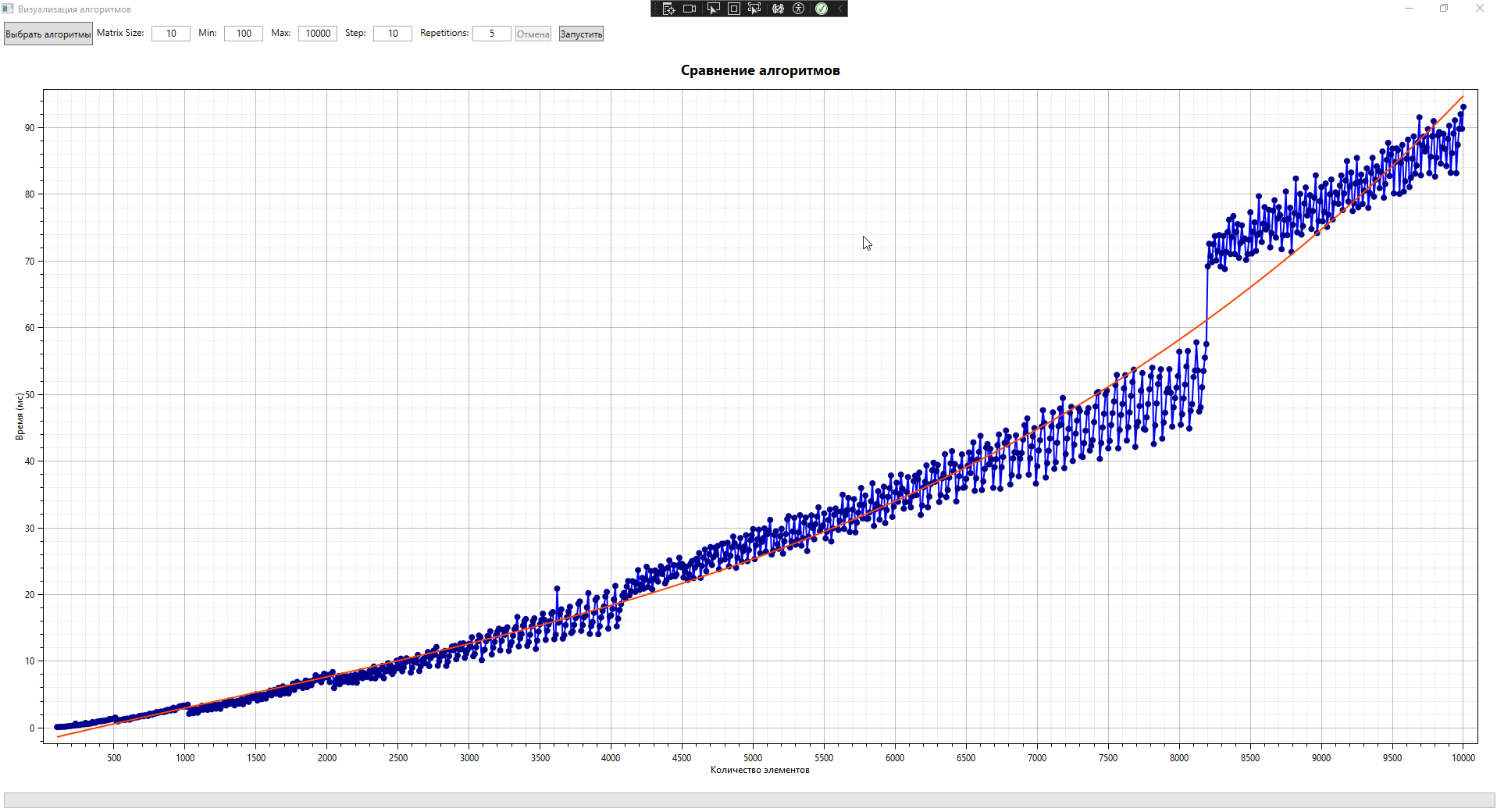


Рис. 7.1 Алгоритм сортировки TimSort

Рассмотрим подробнее шаги алгоритма.

Шаг 1. Вычисление minrun.

Число minrun определяется на основе n исходя из следующих принципов:

1. Оно не должно быть слишком большим, поскольку к подмассиву размера minrun будет в дальнейшем применена сортировка вставками, а она эффективна только на небольших массивах
2. Оно не должно быть слишком маленьким, поскольку чем меньше подмассив — тем больше итераций слияния подмассивов придётся выполнить на последнем шаге алгоритма.
3. 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. Алгоритм расчёта minrun: 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):

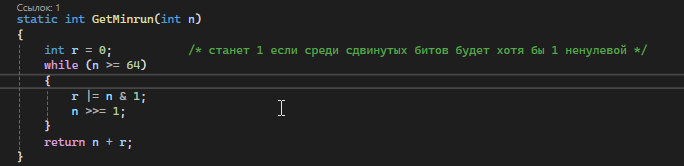


Рис. 7.2 Код вычисления minrun

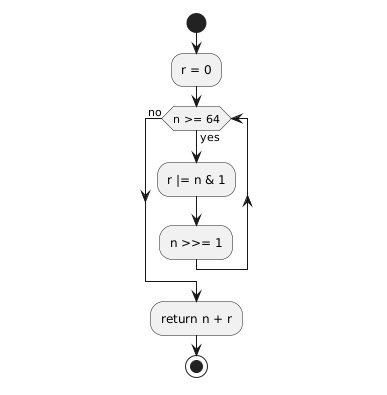


Рис. 7.3 БСА метода GetMinrun

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:

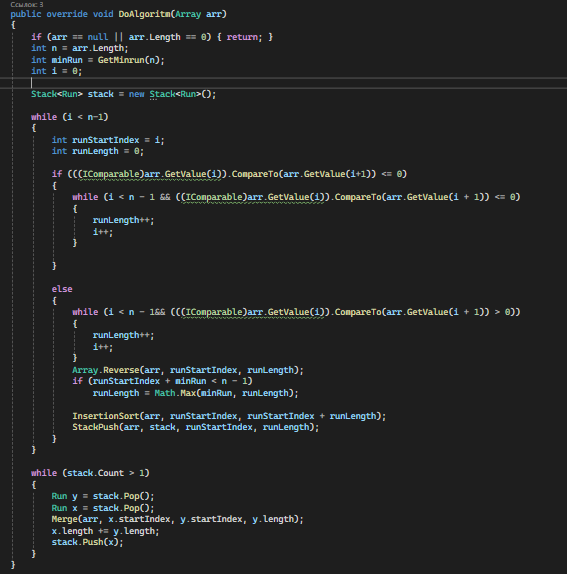
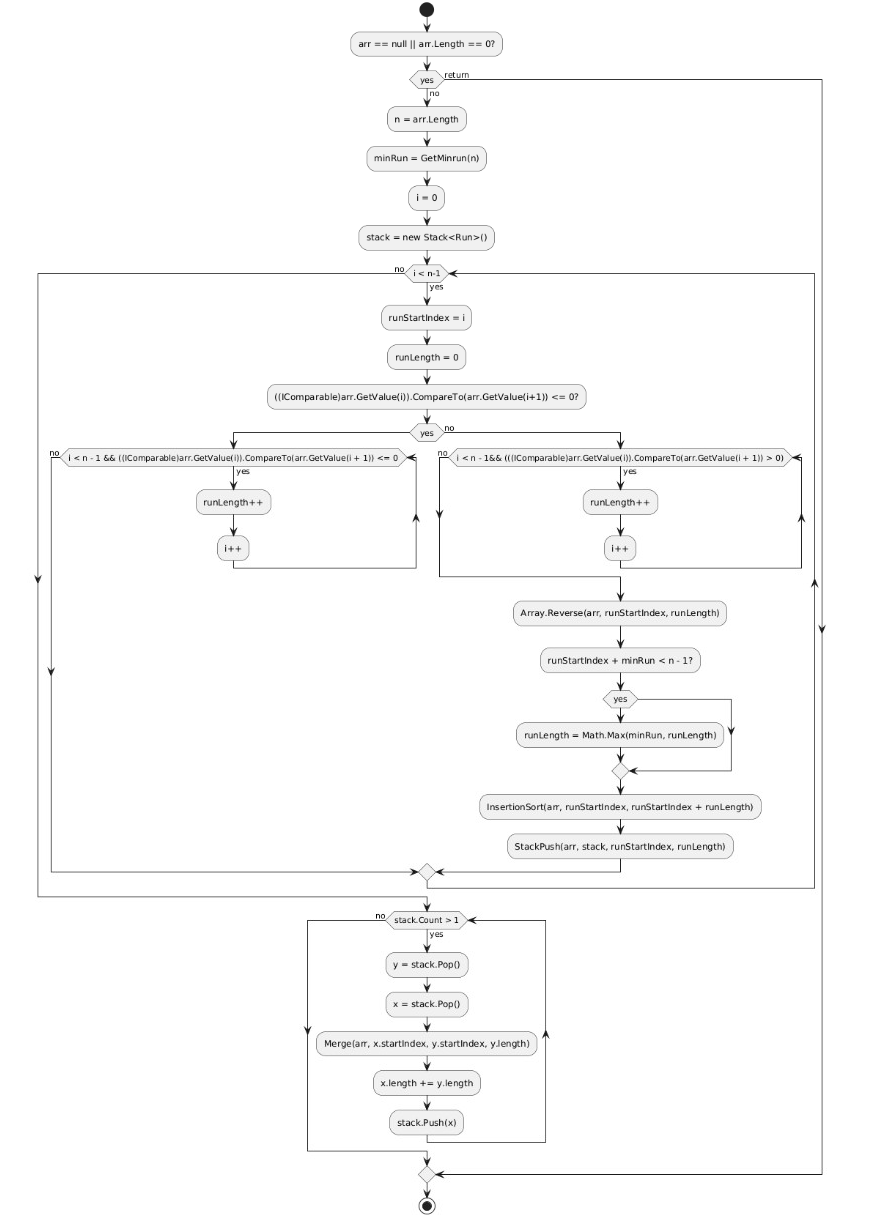


Рис. 7.4 Основной код алгоритма

 Рис. 7.4 БСА основной части алгоритма

Шаг 3 сортировка вставками:

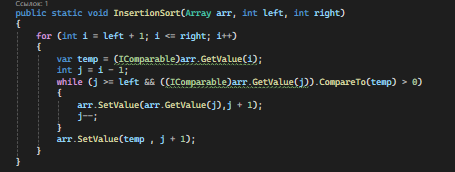


Рис. 7.5 Алгоритм сортировки вставками

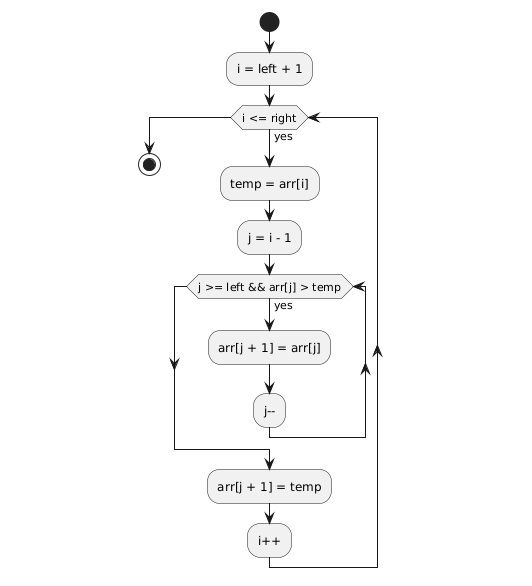


Рис. 7.6 БСА сортировки вставками

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.

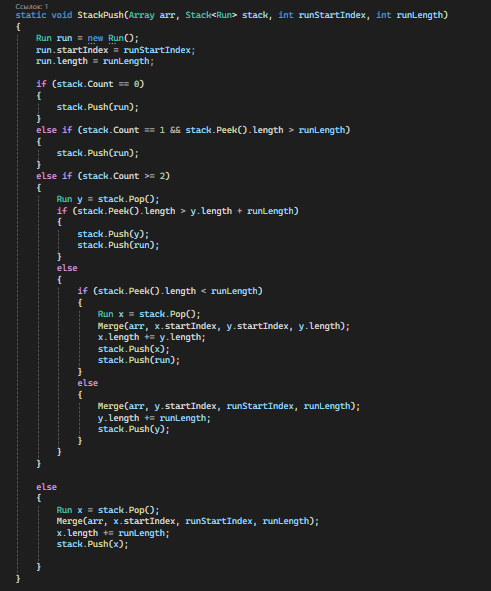
Код работы со стеком: 

Рис. 7.7 Код стека

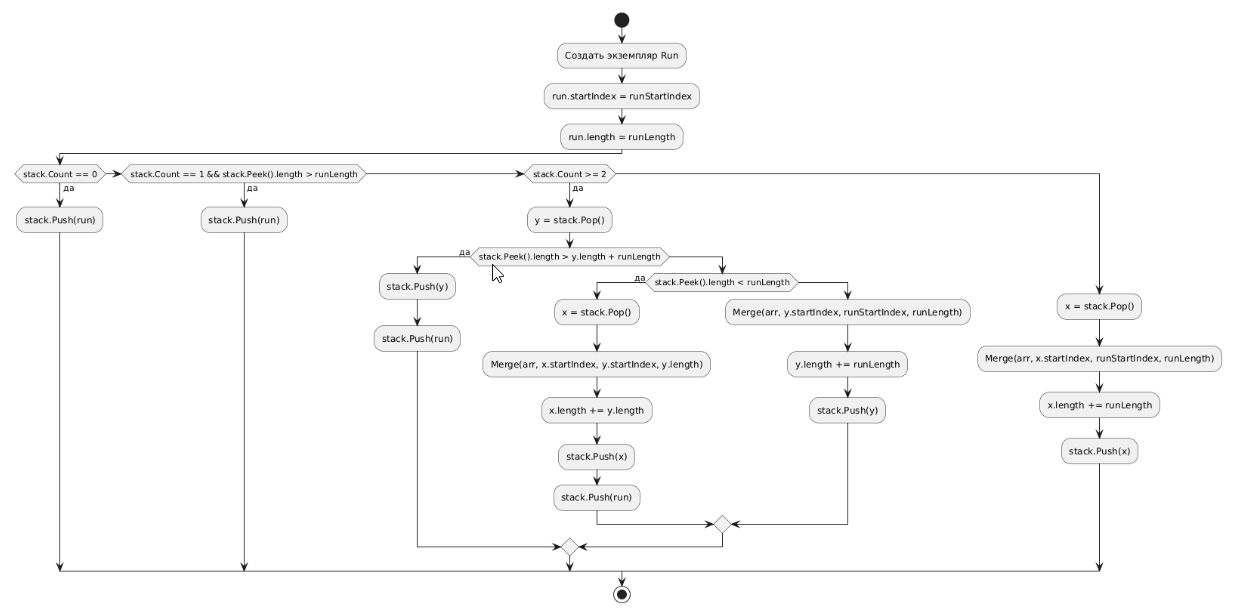


Рис. 7.8 БСА работы со стеком

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:

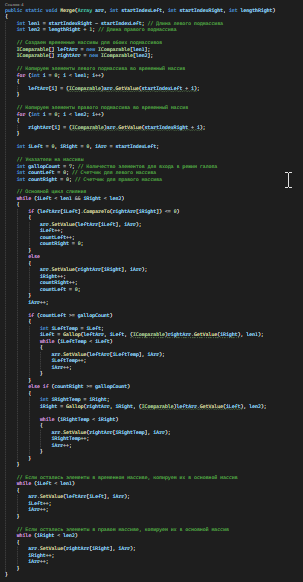


Рис. 7.9 Код слияния подмассивов.

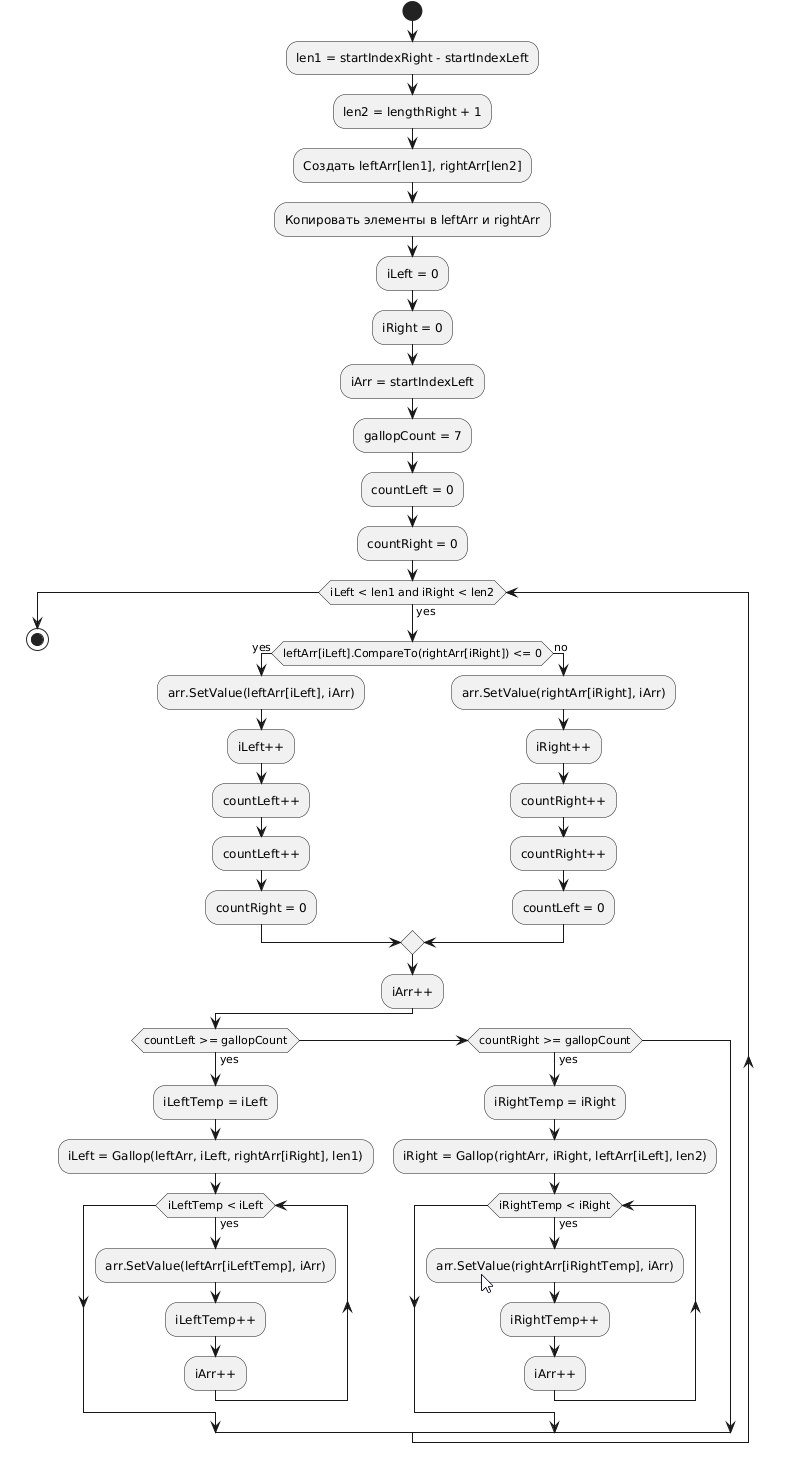


Рис. 7.10 БСА слияния подмассивов.

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 Галопа:

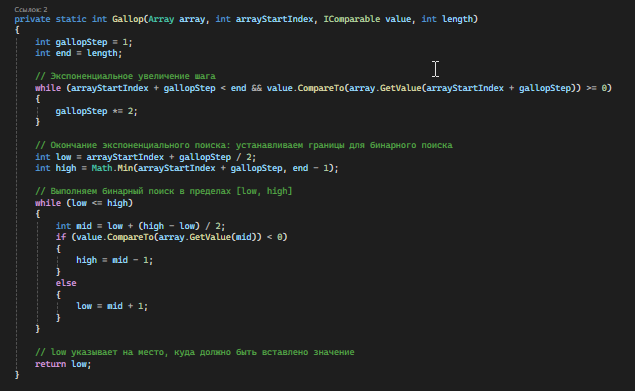


Рис. 7.11 Код галопа.

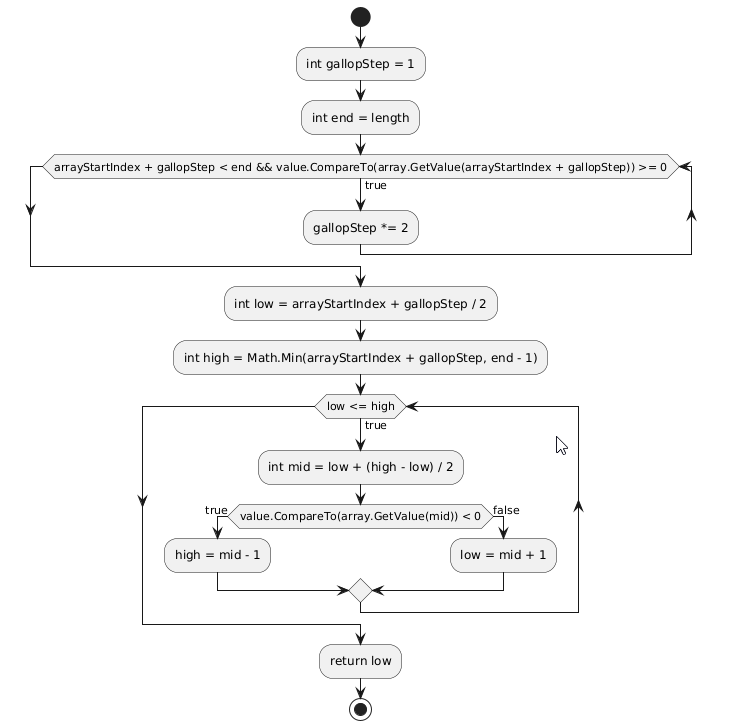


Рис. 7.12 БСА галопа.

**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)).

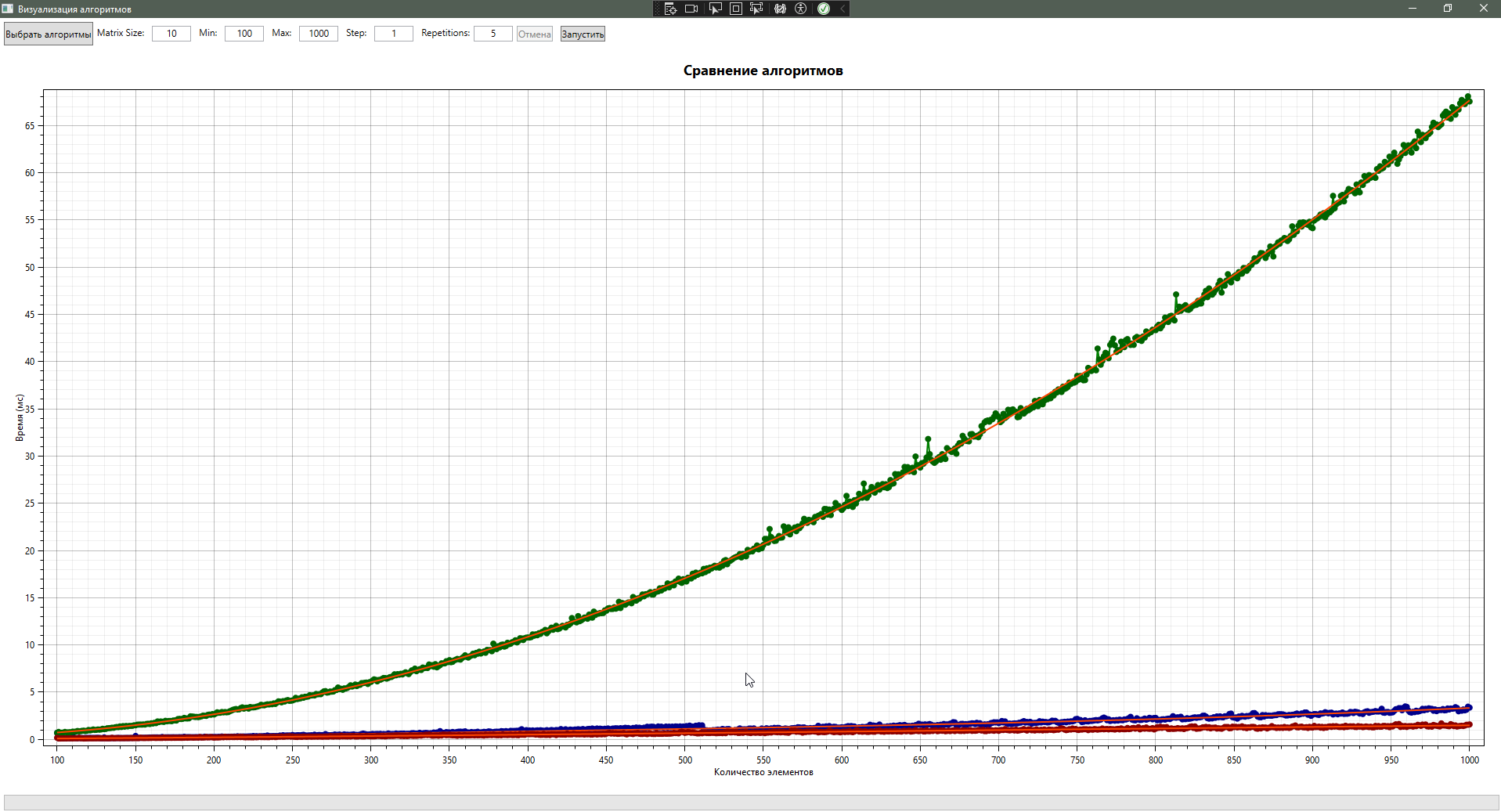


Рис. 7.13 Сравнение алгоритмов сортировки.

**Алгоритм 8. Возведение в степень по определению**

Алгоритм заключается в последовательном умножении числа на само себя n-ое количество раз, где n – степень. Данный алгоритм является наиболее простым для понимания, ведь исходит из определения, но неэффективным и ресурсозатратным.

* Возведение в степень по определению
* Временная сложность: O(n)
* n от 100 до 1000
* Шаг 10
* Ср. знач. на основе тестов: 5

Замеры времени от количества элементов (смотри рис. 8.1)

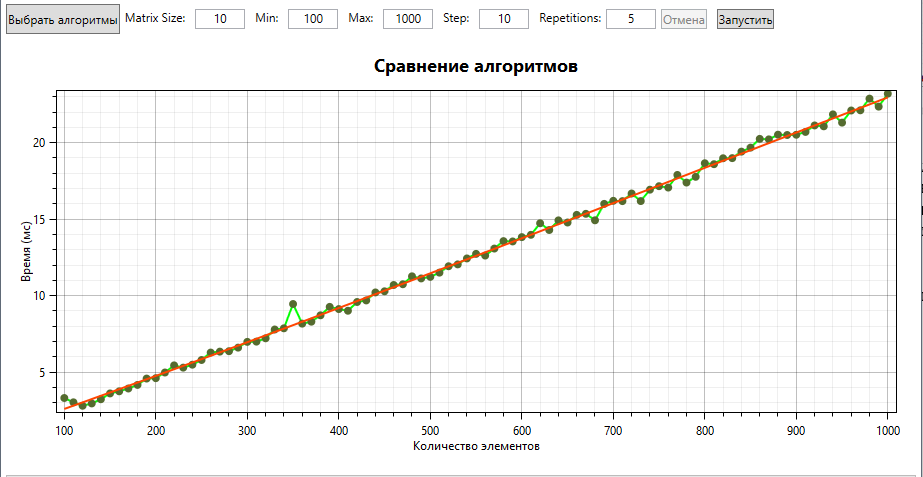


Рис. 8.1 Возведение в степень по определению

Код алгоритма изображен на рисунке 8.2

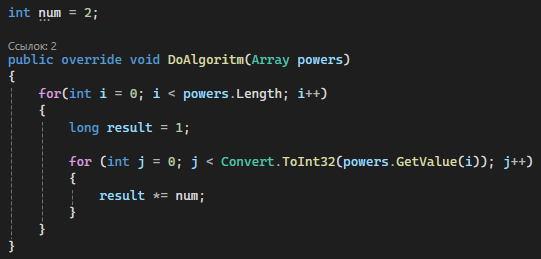


Рис 8.2 Код алгоритма

Блок-схема алгоритма изображена на рисунке 8.3

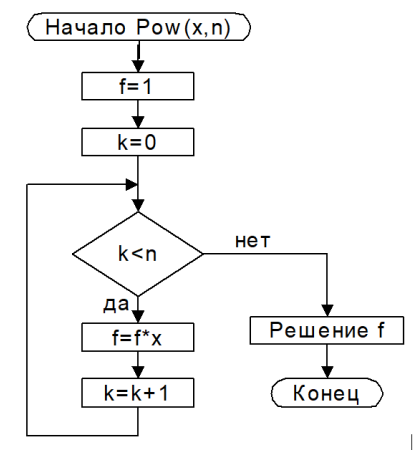


Рис.8.3 Блок-схема алгоритма

В данном и последующих алгоритмах возведения в степень мы принимаем массив степеней, в то время как число является фиксированным. Это связано с тем, что время работы алгоритма напрямую зависит от степени возведения числа.

**Алгоритм 9. Рекурсивное возведение в степень**

Рекурсивный алгоритм возведения в степень — это метод, который использует рекурсию для вычисления степени числа. Он отличается большей скоростью по сравнению с возведением по определению.

* Рекурсивное возведение в степень
* Временная сложность: O(n)
* n от 100 до 1000
* Шаг 10
* Ср. знач. на основе тестов: 5

**Описание работы алгоритма:**

1. Проверка степени на ноль. Если степень будет равна нулю, алгоритм сразу вернет результат равный единице, так как любое число в нулевой степени равно единице.

2. Рекурсивный вызов алгоритма. На этом шаге в переменную результата будет записываться возвращенное значение от рекурсивного вызова этого же метода, но в этом рекурсивном вызове мы передаем параметр степени, как целую часть параметра деленного на 2.

3. Проверка четности числа

А) Если число четно, то результат умножается сам на себя

Б) Если число нечетно, то результат умножается сам на себя и на число, которое было передано.

4. Возвращение переменной результата

Замеры времени от количества элементов (смотри рис. 9.1)

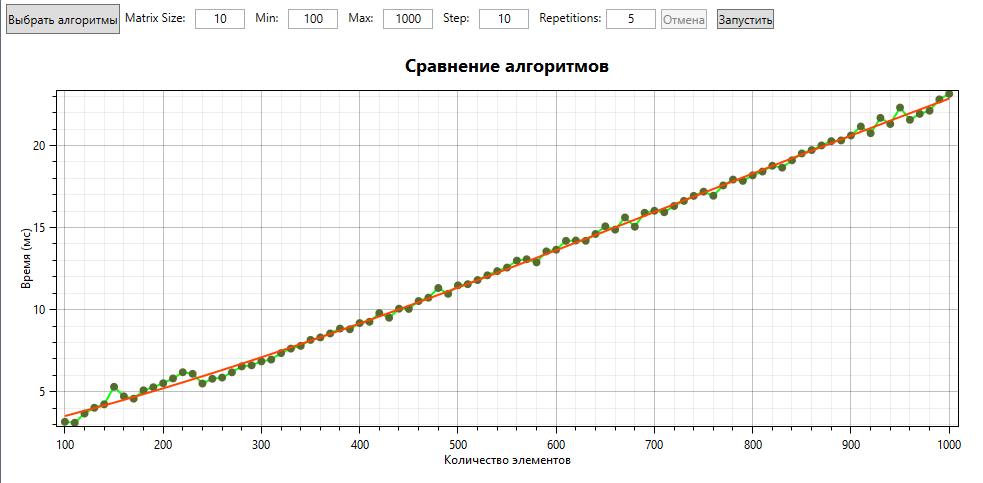


Рис. 9.1 Рекурсивное возведение в степень

Код алгоритма изображен на рисунке 9.2

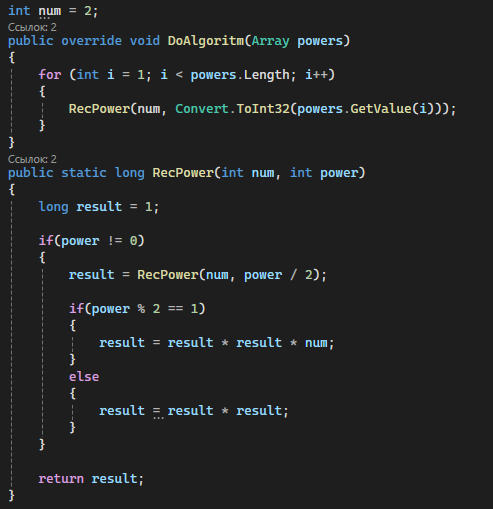


Рис. 9.2 Код Алгоритма

Блок-схема рекурсивного возведения в степень изображена на рисунке 9.3

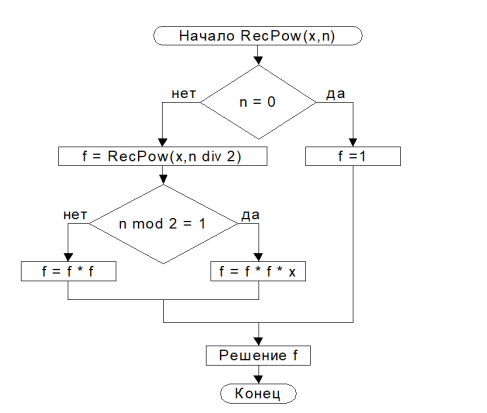


Рис. 9.3 Блок-схема алгоритма

**Алгоритм 10. Быстрое возведение в степень**

Алгоритм быстрого возведения в степень – это алгоритм, выделяющийся на фоне других своей скоростью, но являющийся при этом более трудным к пониманию.

* Быстрое возведение в степень
* Временная сложность: O(n \* log(n))
* n от 100 до 1000
* Шаг 10
* Ср. знач. на основе тестов: 5

**Описание работы алгоритма:**

1. Проверка четности степени.

А) Если степень четная, то результат равен единице

Б) Если степень нечетная, то результат равен передаваемому числу

2. Пока степень не равна нулю, выполняются пункты 3-5

3. Берем целую часть от деления степени на 2

4. Умножаем число само на себя

5. Если степень нечетная, то результат умножаем на число

6. Как только степень стала равна нулю, возвращаем результат

Замеры времени от количества элементов (смотри рис. 10.1)

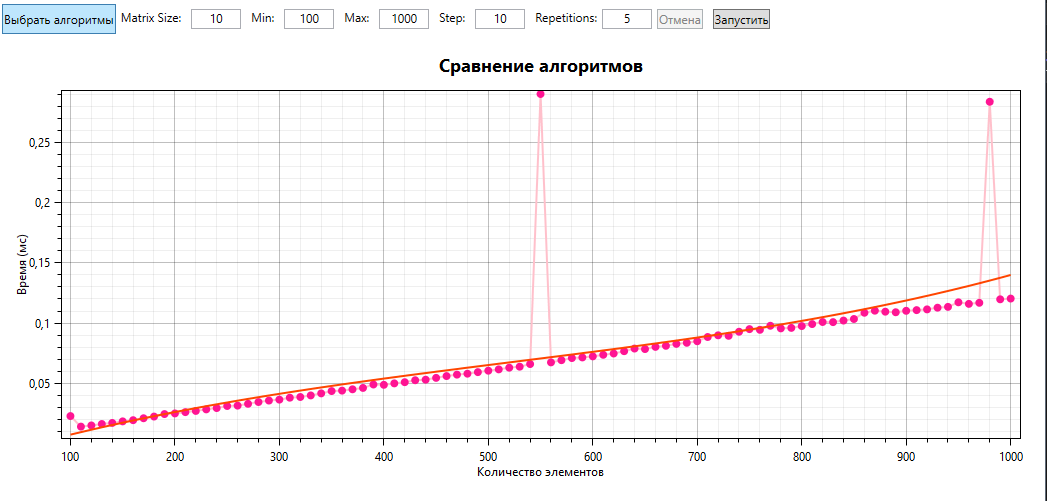


Рис. 10.1 Быстрое возведение в степень

Код алгоритма изображен на рисунке 10.2

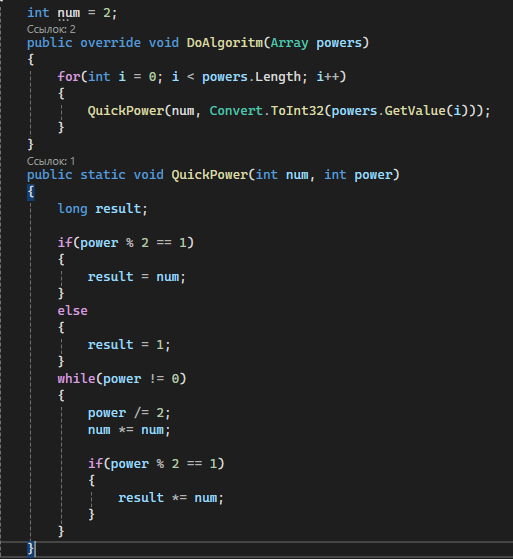


Рис. 10.2 Код алгоритма

Блок-схема алгоритма быстрого возведения в степень изображена на рисунке 10.3

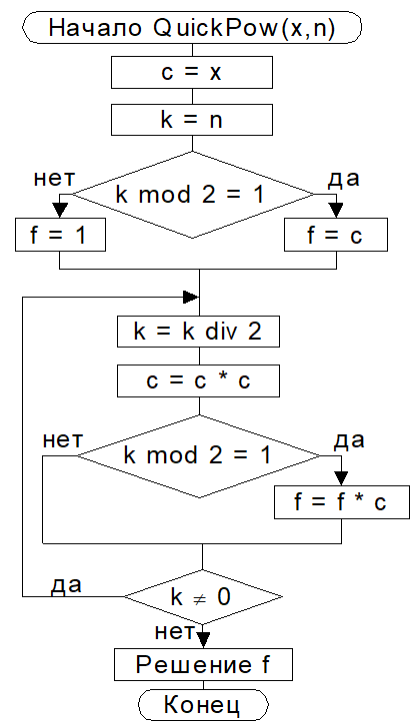


Рис. 10.3 Блок-схема алгоритма

**Алгоритм 11.** **Классический быстрый алгоритм возведения в степень**

Классический быстрый алгоритм возведения в степень – это более устоявшаяся вариация алгоритма быстрого возведения в степень.

**Описание работы алгоритма:**

1. Приравниваем переменную результата к единице

2. Пока степень не равна нулю, выполняем шаг 3

3. Проверка четности степени

А) Если степень четная, число умножаем само на себя, а степень приравниваем к целой части от деления степени на 2

Б) Если степень нечетная, результат умножаем на число, а степень уменьшаем на 1

4. Как только степень стала равной нулю, возвращаем результат.

Замеры времени от количества элементов (смотри рис. 11.1)

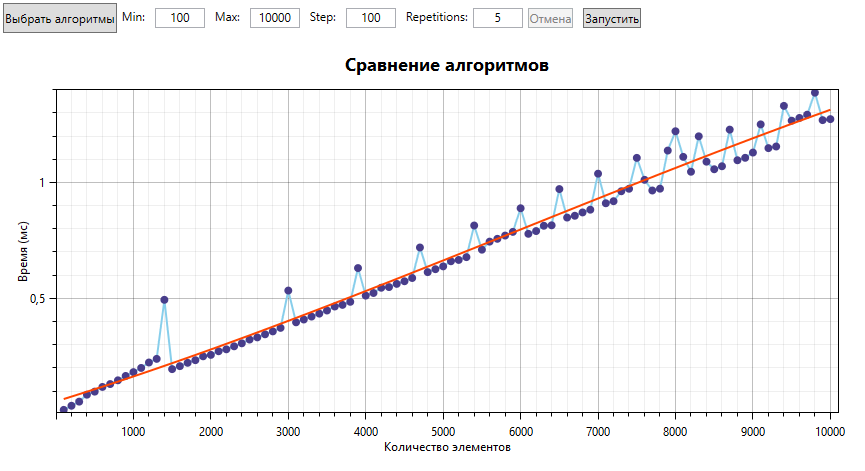


Рис. 11.1 Классический алгоритм быстрого возведения в степень

Код алгоритма изображен на рисунке 11.2

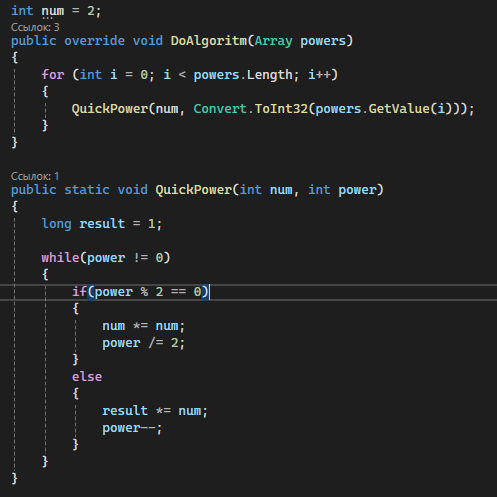


Рис. 11.2 Код алгоритма

Блок-схема классического быстрого возведения в степень изображена на рисунке 11.3

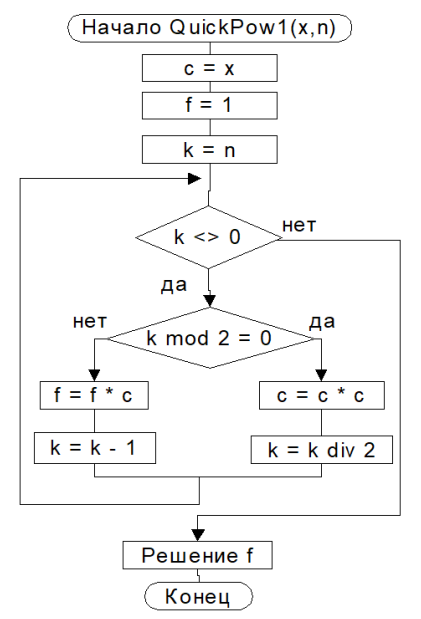


Рис. 11.3 Блок-схема алгоритма

**Алгоритм 12. Сортировка кучей (Heap Sort)**

Heap Sort — это эффективный алгоритм сортировки, который использует структуру данных, называемую кучей. Куча — это специальное двоичное дерево, которое удовлетворяет свойству кучи: в максимальной куче каждый родительский узел больше или равен своим дочерним узлам, а в минимальной куче каждый родительский узел меньше или равен своим дочерним узлам.

**Достоинства Heap Sort:**

1. Эффективность. Алгоритм работает за О(n log n)

2. Алгоритм не требует дополнительной памяти для сортировки, за исключением небольшого количества памяти для рекурсивных вызовов

**Описание работы алгоритма:**

1. Построение максимальной кучи. Начиная с последнего нелистового узла, "просеиваем" каждый узел вниз, чтобы удовлетворить свойству максимальной кучи. Это делается путем сравнения узла с его дочерними узлами и обмена их местами, если необходимо.

2. Обмениваем корень кучи (максимальный элемент) с последним элементом массива

3. Уменьшаем размер кучи на один

4. Преобразовываем полученное дерево в максимальную кучу с новым корнем

5. Повторяем вышестоящие шаги, размер кучи больше 1

* Сортировка кучей
* Временная сложность: O(n \* log(n))
* n от 100 до 1000
* Шаг 10
* Ср. знач. на основе тестов: 5

Замеры времени от количества элементов (смотри рис. 12.1)

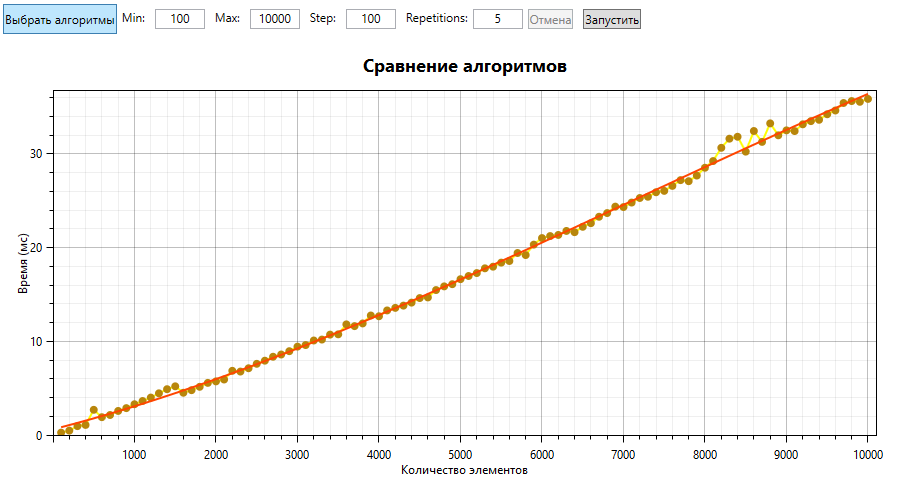


Рис. 12.1 Сортировка кучей

Код алгоритма показан на рисунках 12.2 – 12.3

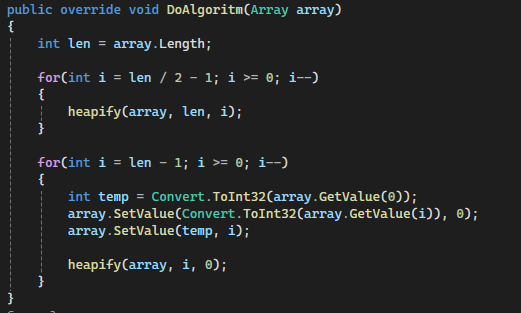


Рис. 12.2 Основной код алгоритма

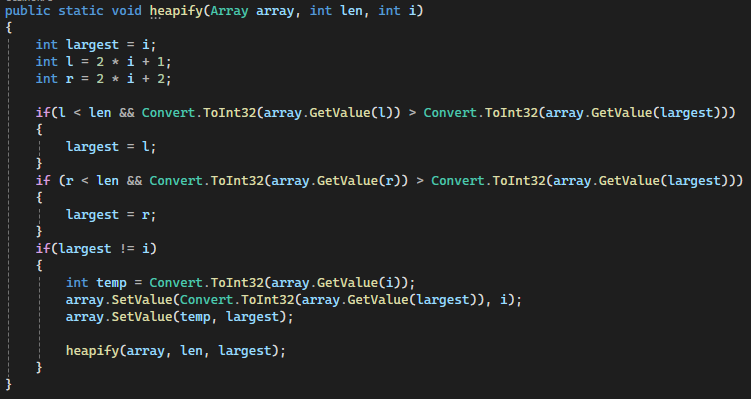


Рис. 12.3 Метод построения кучи

Блок-схемы алгоритма и вспомогательного метода построения кучи изображены на рисунках 12.4 – 12.5

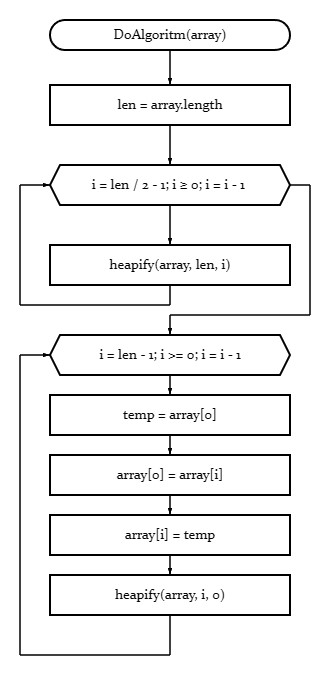


Рис. 12.4 Блок-схема алгоритма

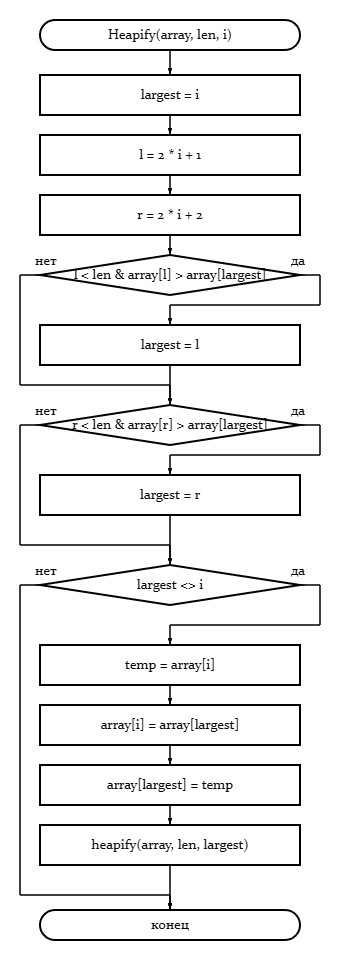


Рис. 12.5 Блок-схема вспомогательного метода heapify

**Алгоритм 13. BingoSort**

Это усовершенствованная разновидность сортировки выбором. для сортировки выбором частичная или реверсная упорядоченность массива роли не играет — она обработает его примерно с той же скоростью что и обычный рандом. Также для классической сортировки выбором неважно, состоит ли массив из уникальных или повторяющихся элементов — на скорость это практически не влияет. Но бинго-сортировка учитывает, если массив состоит из повторяющихся элементов. Смысл в том, что в неупорядоченной части запоминается не только максимальный элемент, но и определяется максимум для следующей итерации. Это позволяет при повторяющихся максимумах не искать их заново каждый раз, а ставить на своё место сразу как только этот максимум в очередной раз встретили в массиве.

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

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

Алгоритмическая сложность осталась та же. Но если массив состоит из повторяющихся чисел, то бинго-сортировка справится в десятки раз быстрее, чем обычная сортировка выбором.

Замеры времени от количества элементов (смотри рис. 13.1)

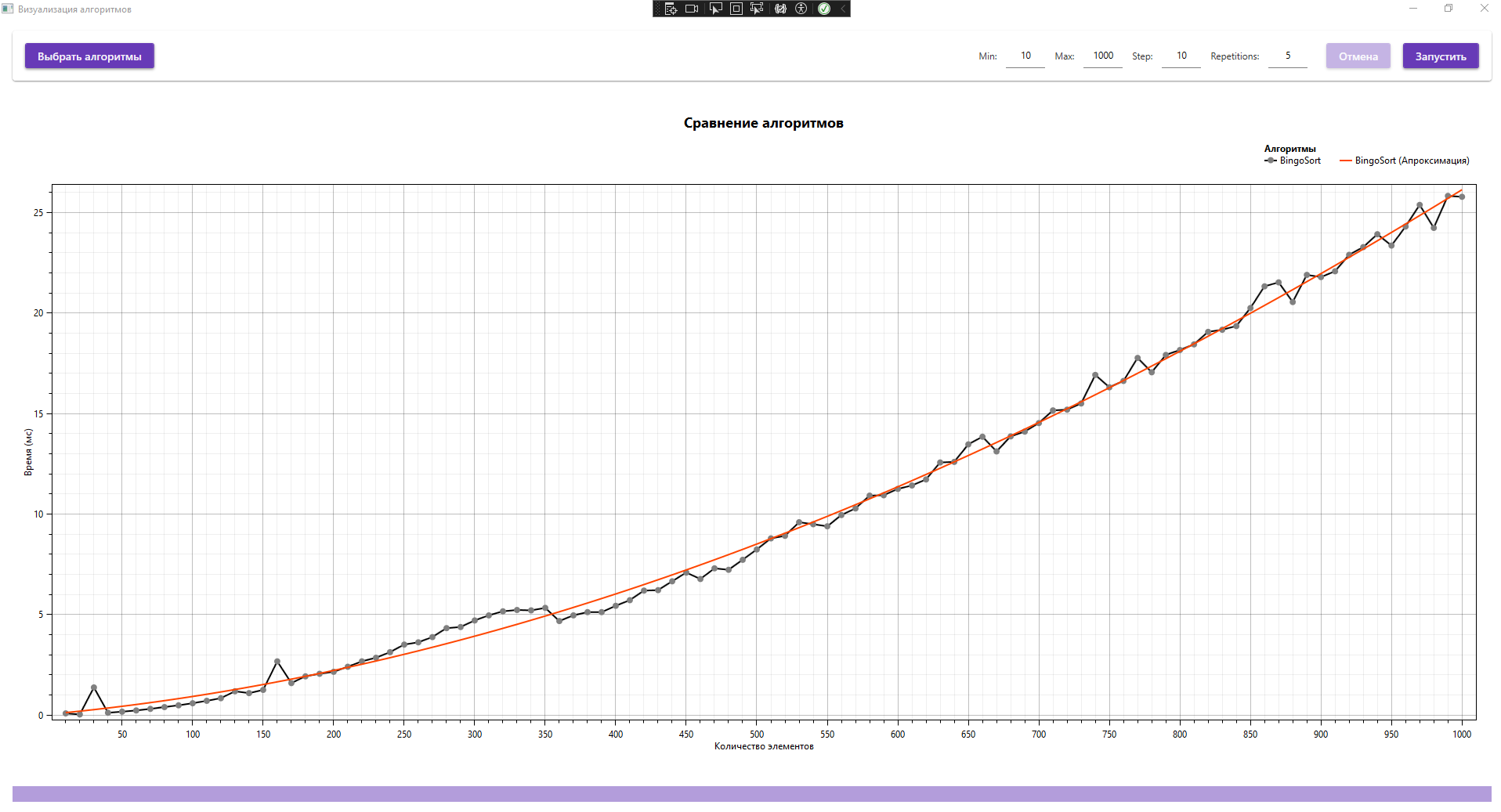


Рис. 13.1 бинго-сортировка

Код алгоритма показан на рисунках 13.2 – 13.3

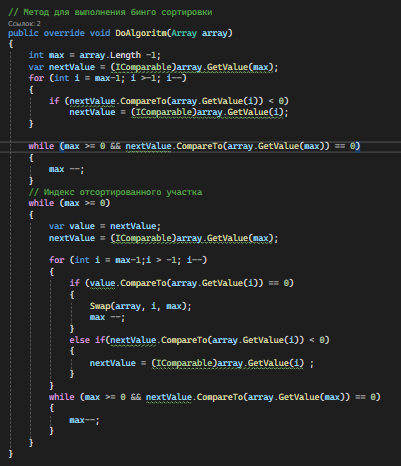


Рис. 13.2.1 Основной код алгоритма

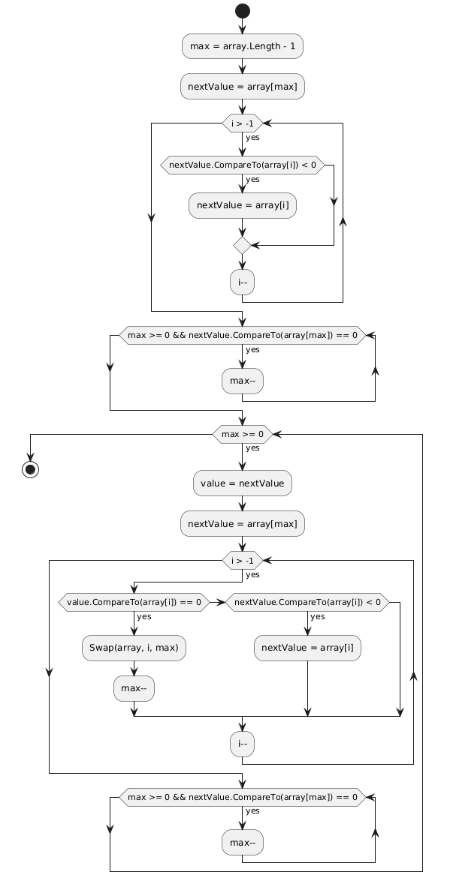


Рис. 13.2.2 БСА алгоритма

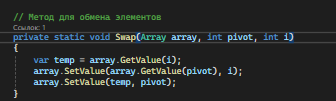


Рис. 13.3 метод перестановки элементов

**Алгоритм 14: Gnome Sort (Гномья сортировка)**

Gnome Sort - это простой алгоритм сортировки, который работает по принципу садового гнома, упорядочивающего горшки с цветами. Алгоритм проходит по массиву, сравнивая соседние элементы. Если элементы расположены в неправильном порядке, они меняются местами. Затем алгоритм делает шаг назад, чтобы проверить, не нарушился ли порядок в результате обмена.

**Описание работы Gnome Sort:**

1. Начинаем с первого элемента массива (индекс 0).
2. Сравниваем текущий элемент со следующим.
   * Если текущий элемент больше следующего, меняем их местами и уменьшаем индекс на 1 (делаем шаг назад).
   * Если текущий элемент меньше или равен следующему, увеличиваем индекс на 1 (делаем шаг вперед).
3. Если индекс становится меньше 0, значит, мы дошли до начала массива и нужно увеличить индекс на 1.
4. Повторяем шаги 2-3, пока не дойдем до конца массива.

* Временная сложность: O(n^2)
* n от 100 до 10000
* Шаг 10
* Ср. знач. на основе тестов: 5

Замеры времени от количества элементов (смотри рис. 14.1)

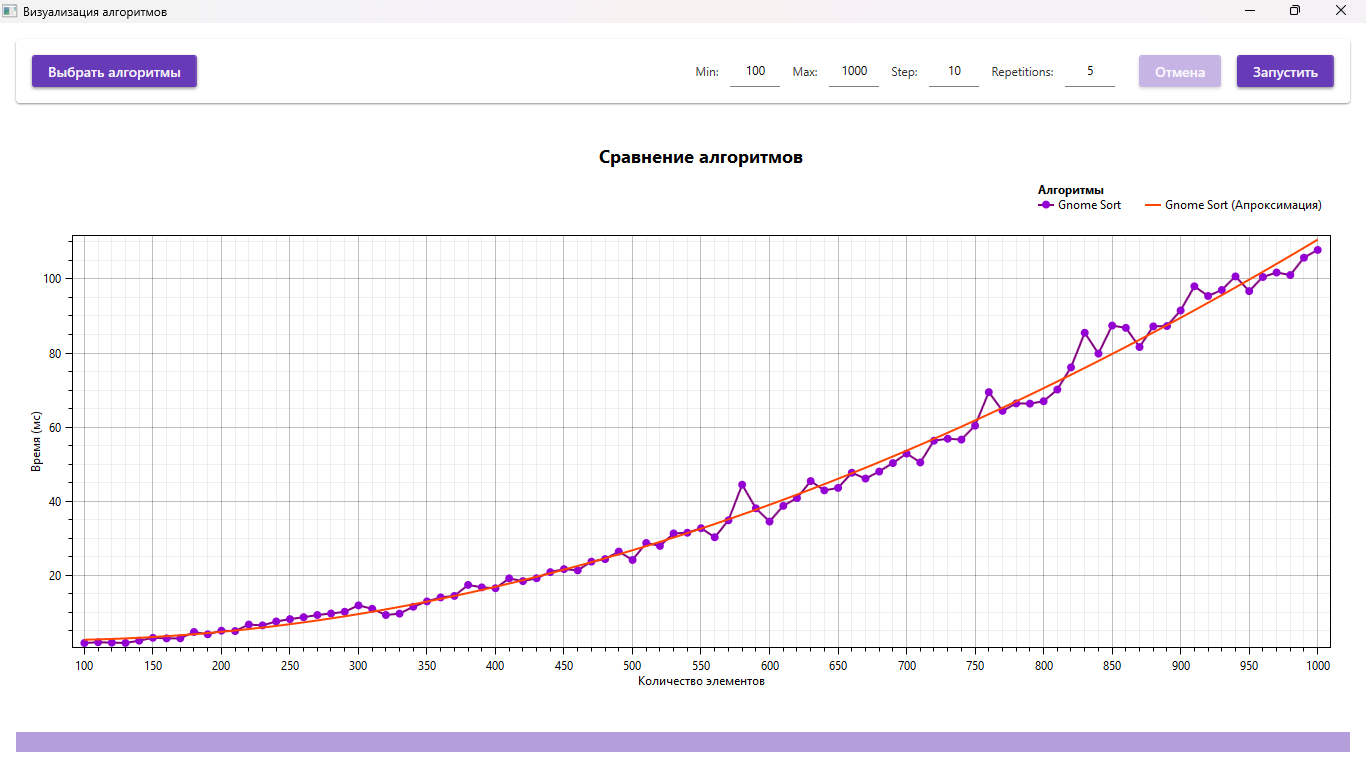


Рис. 14.1 Гномья сортировка

Код алгоритма показан на рисунках 14.2 – 14.3

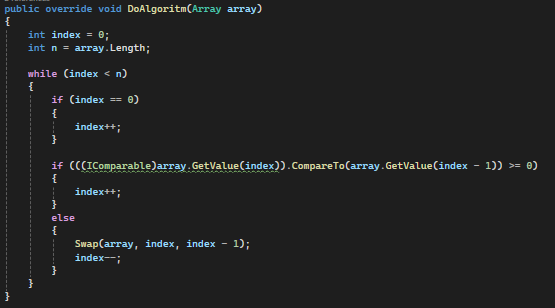


Рис. 14.2.1 Основной код алгоритма

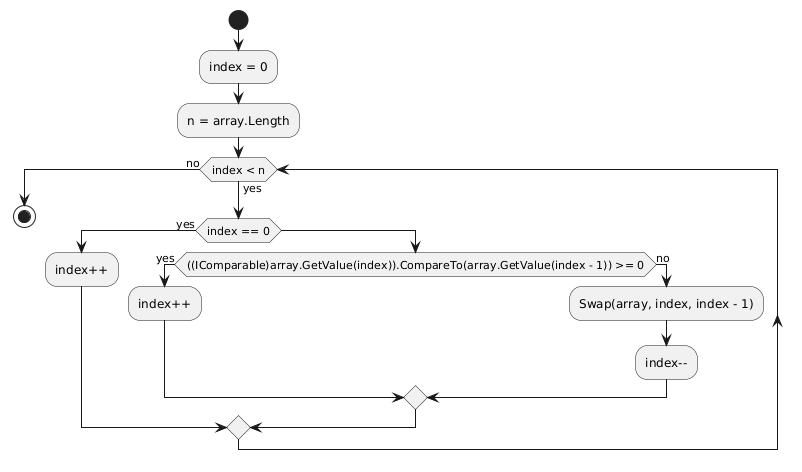


Рис. 14.2.2 БСА Основного алгоритма

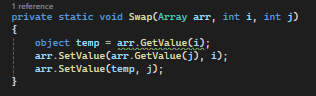


Рис. 14.3 Метод перестановок

**Алгоритм 15: Умножение матриц**

Алгоритм умножения матриц — это фундаментальная операция в линейной алгебре. Он используется во многих областях, включая компьютерную графику, машинное обучение и научные вычисления.

**Описание работы алгоритма:**

Для умножения двух матриц A (размерности m x n) и B (размерности n x p) результирующая матрица C (размерности m x p) вычисляется следующим образом:

C[i][j] = сумма(A[i][k] \* B[k][j] для k от 0 до n-1)

* Временная сложность: Классический алгоритм умножения матриц имеет временную сложность O(n^3).
* Размер матрицы от 10 до 100
* Шаг 1
* Ср. знач. на основе тестов: 5

Замеры времени от количества элементов (смотри рис. 15.1)

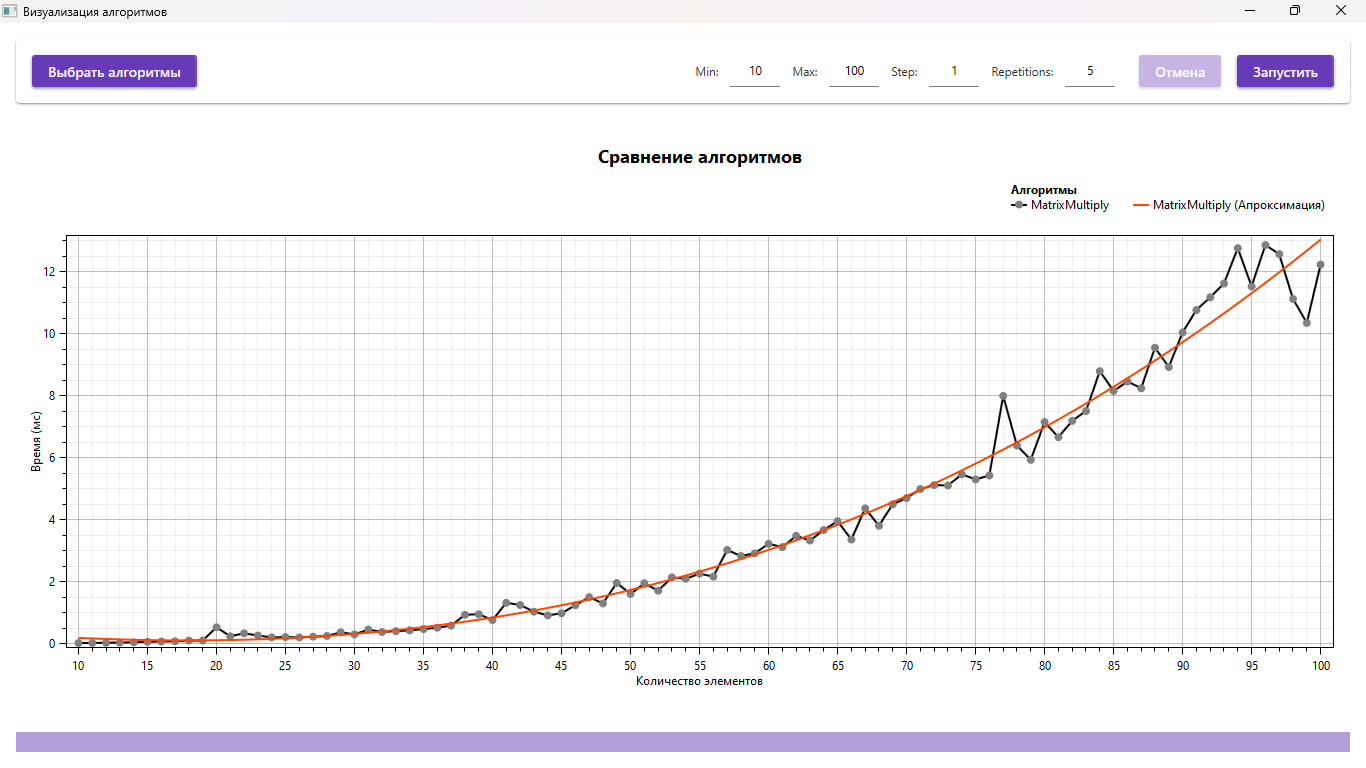


Рис. 15.1 Умножение матриц

Код алгоритма показан на рисунке 15.2

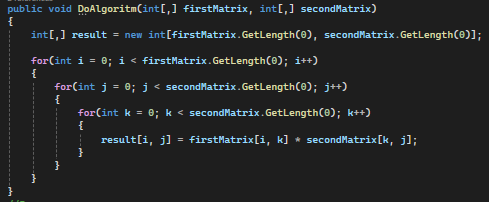


Рис. 15.2 Основной код алгоритма

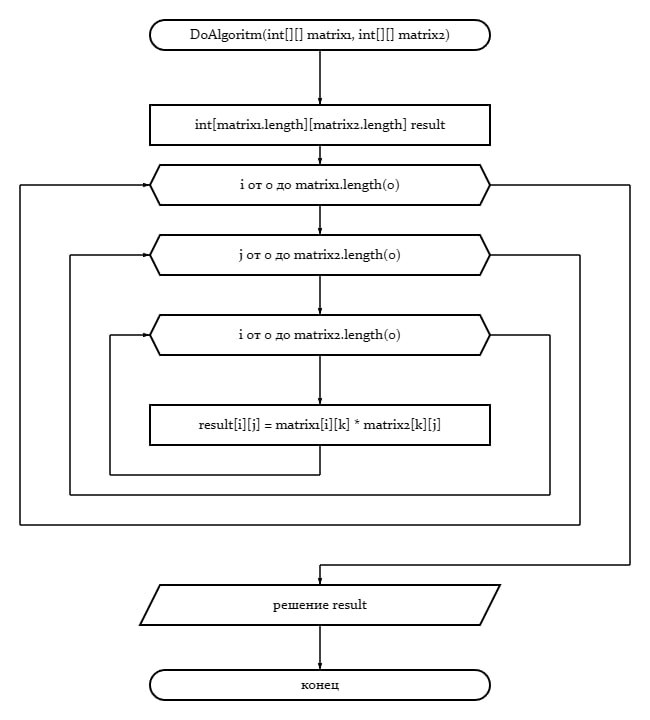


Рис. 15.3 БСА основного алгоритма

**UML диаграмма архитектуры проекта**

