Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

НАПРАВЛЕНИЕ ПОДГОТОВКИ «09.03.04 Программная инженерия»

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №3

Название: Алгоритмы сортировки

Дисциплина: Анализ алгоритмов

Студент	ИУ7-56Б		Т. А. Казаева
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель			Л. Л. Волкова
		Подпись, дата	И. О. Фамилия

Москва, 2021 г.

Оглавление

Страница

1 | Введение

Переразмещение элементов в порядке возрастания или убывания – один из самых частых вопросов, возникающих в программировании. От порядка, в котором хранятся элементы в памяти компьютера, зависит множество аспектов работы компьютерных приложений – от удобства пользователя при работе с большими последовательностями до скорости работы этих приложений. Алгоритмы сортировки, рассматриваемые в данной лабораторной работе, являются, скорее, алгоритмами упорядочивания элементов последовательности в порядке возрастания или убывания. Ниже представлены некоторые области применения сортировочных алгоритмов:

- Решение задач группирования;
- Поиск общих элементов в двух или более последовательностях;
- Поиск информации по значению ключей.

Характеристики алгоритма сортировки – это время его работы и память, которую алгоритм использует. Эти параметры определяются как зависимость от длины сортируемой последовательности. Также алгоритм может быть устойчивым или неустойчивым. Устойчивая сортировка не меняет взаимного расположения элементов с одинаковыми ключами[1].

В рамках данной лабораторной работы будут рассмотрены три алгоритма сортировки, проанализированы время их работы.

2 | Аналитический раздел

2.1 Сортировка вставками

Сортировка вставками (англ. *Insertion sort*) – алгоритм сортировки, основанный на следующей идее: перед рассмотрением записи x_i предполагается, что записи x_1, x_{i-1} уже упорядочены и x_i вставлена в соответствующее ей место[2].

Пусть $1 < i \leq N$ и записи $x_1 \dots x_{i-1}$ расположены так, что расположение их ключей K отвечает формуле 2.1:

$$K_1 \leq K_2 \leq \dots \leq K_{i-1} \tag{2.1}$$

Ключ K_i сравнивается с ключами K_{i-1}, K_{i-2}, ... до тех пор, пока не обнаруживается, что запись x_i вставляется между x_j и x_{j-1}, и тогда записи $x_{j+1}, \dots x_{i-1}$ сдвигаются на одну позицию вверх и рассматриваемая запись помещается в ее место.

2.2 Сортировка выбором

Существует семейство сортировок, основанное на идее многократного выбора[3]. Простейшая сортировка, основанная на этой идее, сводится к следующим шагам:

1. Найти больший ключ, переслать в соответствующую область;
2. Повторить шаг 1 для наименьшего из оставшихся ключей;
3. Повторять шаг 1 пока не будет проанализирована вся последовательность.

В реализации, которая будет рассмотрена в рамках этой лабораторной работы, на каждой итерации область обработки последовательности будет уменьшена на один элемент.

2.3 Пирамидальная сортировка

Пирамидальная сортировка, или сортировка кучей (англ. *Heap sort*) рассматривается как усовершенствованная сортировка пузырьком[4]. Идея сортировки основывается на работе с бинарным сортирующим деревом, называемым пирамидой или двоичной кучей. Дерево можно называть пирамидой, если каждый лист имеет глубину либо d, либо $d-1$ (d — максимальная глубина дерева) и значение в любой вершине не меньше (другой вариант — не больше) значения её потомков.

Алгоритм работает следующим образом. После преобразования неотсортированной части последовательности в сортирующее дерево, в корне окажется наибольший элемент. Максимум обменивается с последним ключом неотсортированной подпоследовательности. К этой части заново применяется процедура преобразования в сортирующее дерево, найденный максимум переставляется в конец. Процедура повторяется до тех пор, пока в неотсортированной последовательности не останется один элемент.

Сортировка кучей неустойчива – для обеспечения устойчивости нужно расширять ключ. Также алгоритм плохо сочетается с кешированием, поскольку на одном шаге выборку приходится делать хаотично по всей длине массива.

2.4 Вывод

В данном разделе были проанализированы три алгоритма сортировки, для каждой из сортировок были выделены принципы работы с последовательностями. Было получено достаточно знаний для разработки и визуализации соответствующих алгоритмов.

3 | Конструкторский раздел

3.1 Схемы алгоритмов

Последовательность элементов наиболее удобно хранить в массиве. На схемах алгоритмов массив представлен как A, а элемент с индексом i представлен как A_i.

На рисунке 3.1 представлена схема алгоритма сортировки вставками.

На рисунке 3.2 представлена схема алгоритма сортировки выбором.

На рисунке 3.3 представлена схема алгоритма пирамидальной сортировки.

На рисунке 3.4 представлена схема алгоритма конструирования бинарного дерева для пирамидальной сортировки.

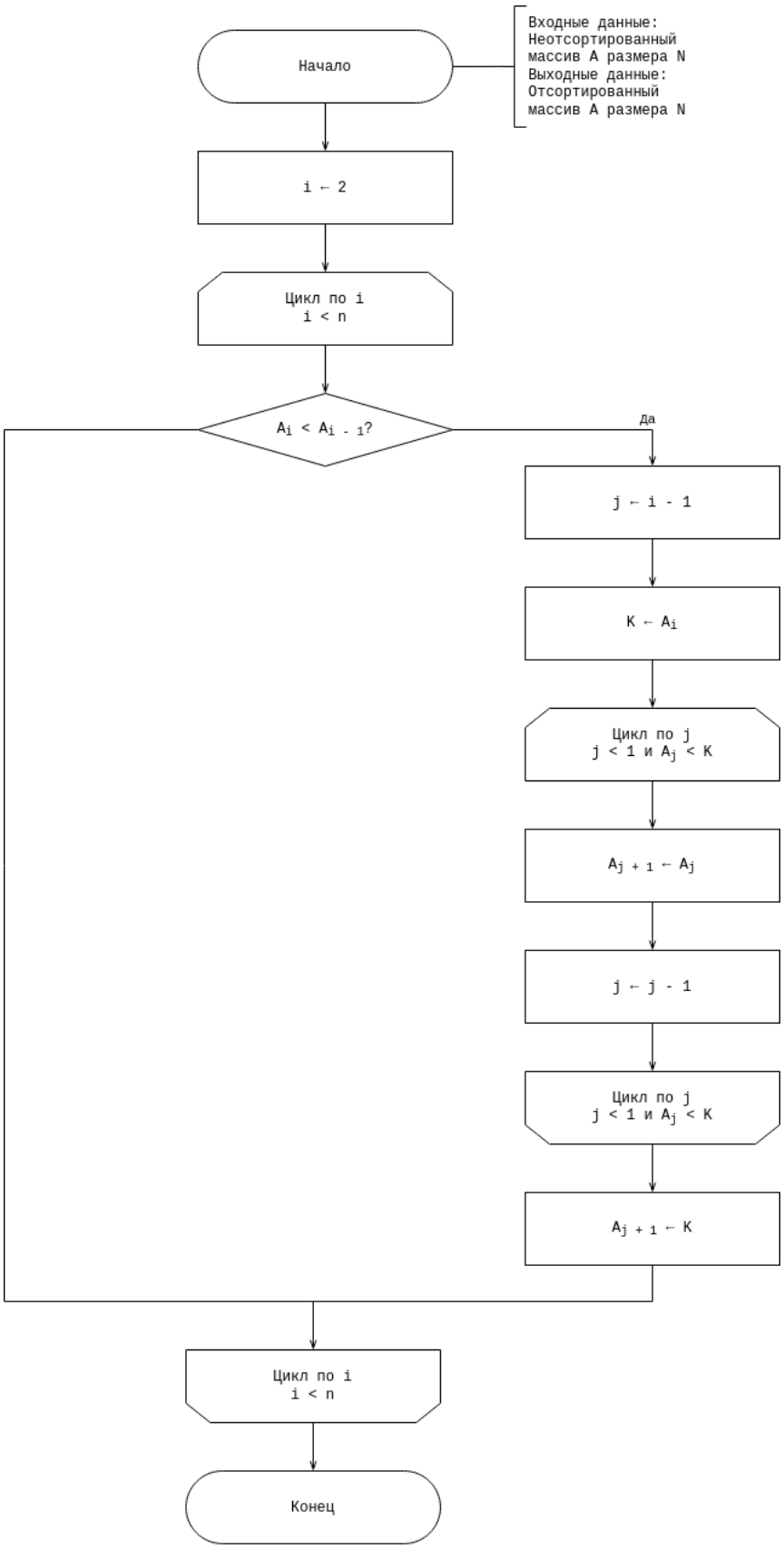

Рисунок 3.1 – Схема алгоритма сортировки вставками

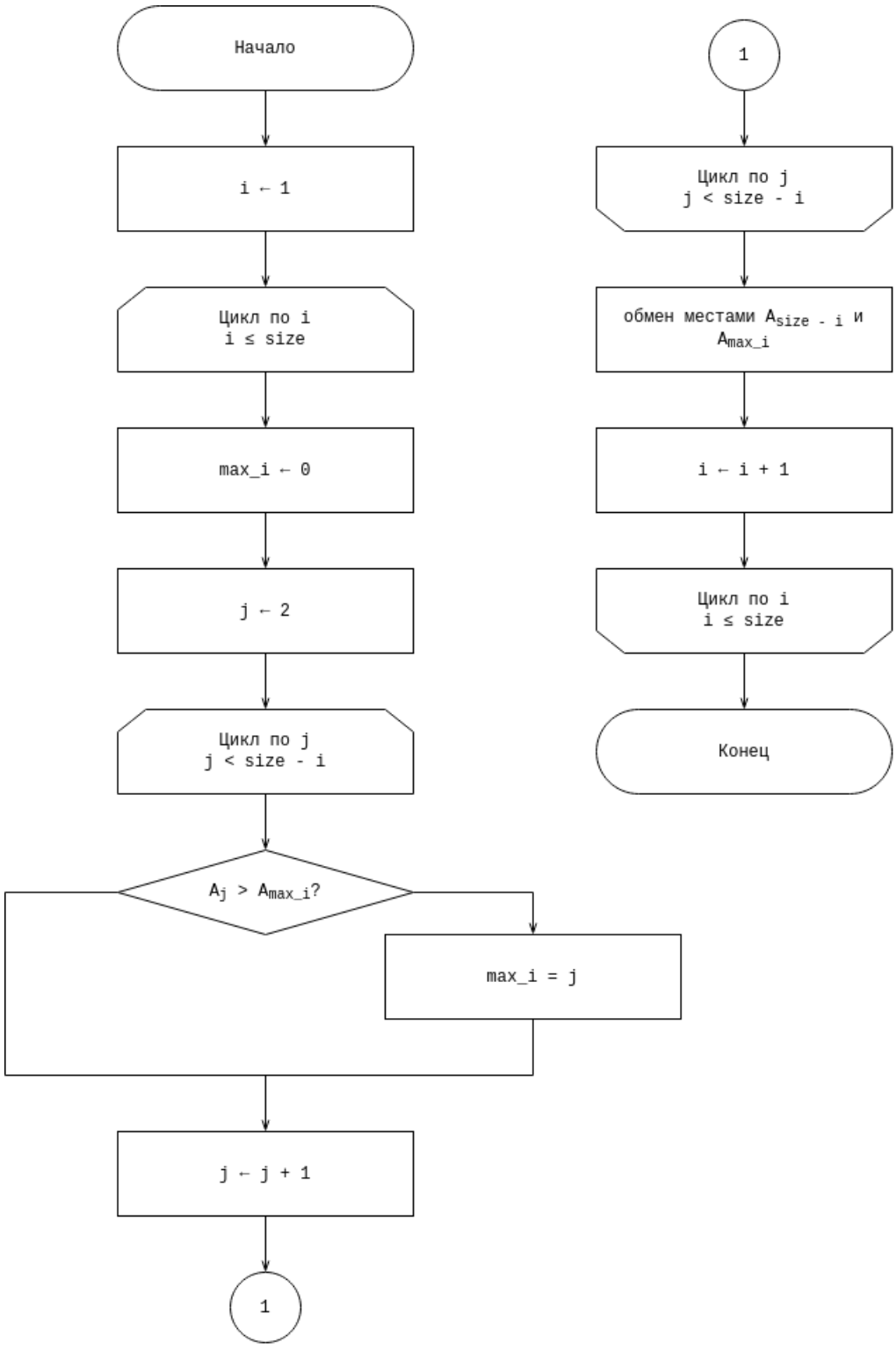

Рисунок 3.2 – Схема алгоритма сортировки выбором

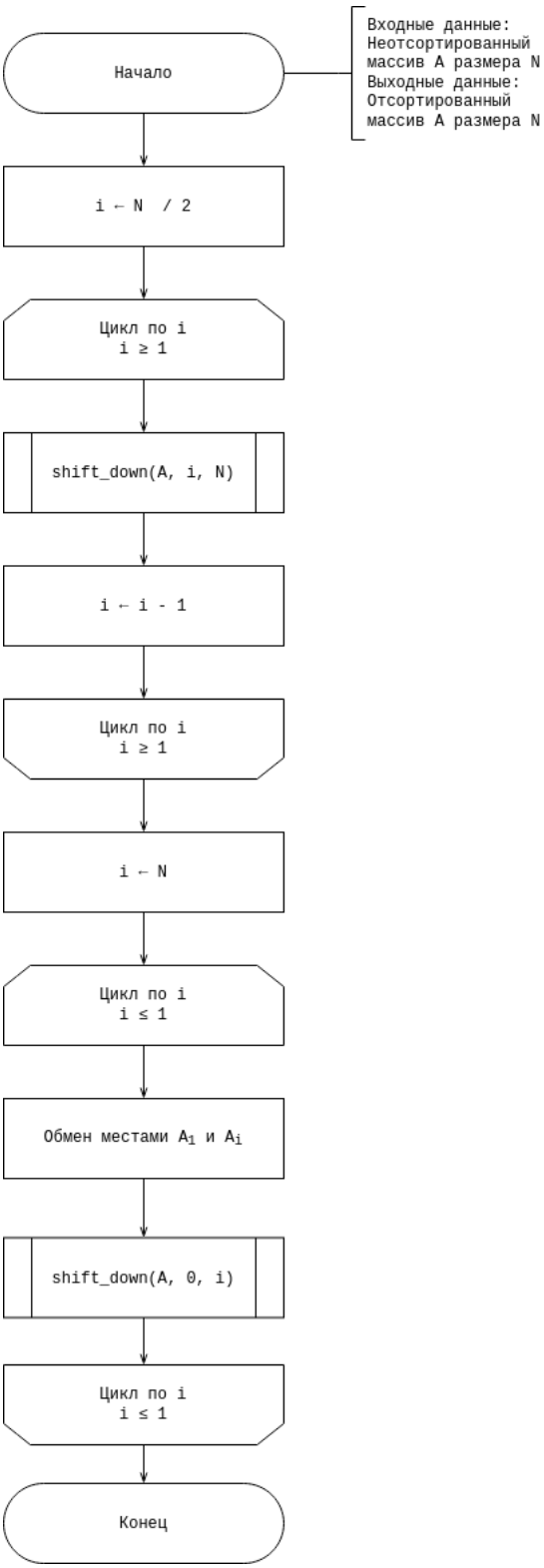

Рисунок 3.3 – Схема алгоритма пирамидальной сортировки

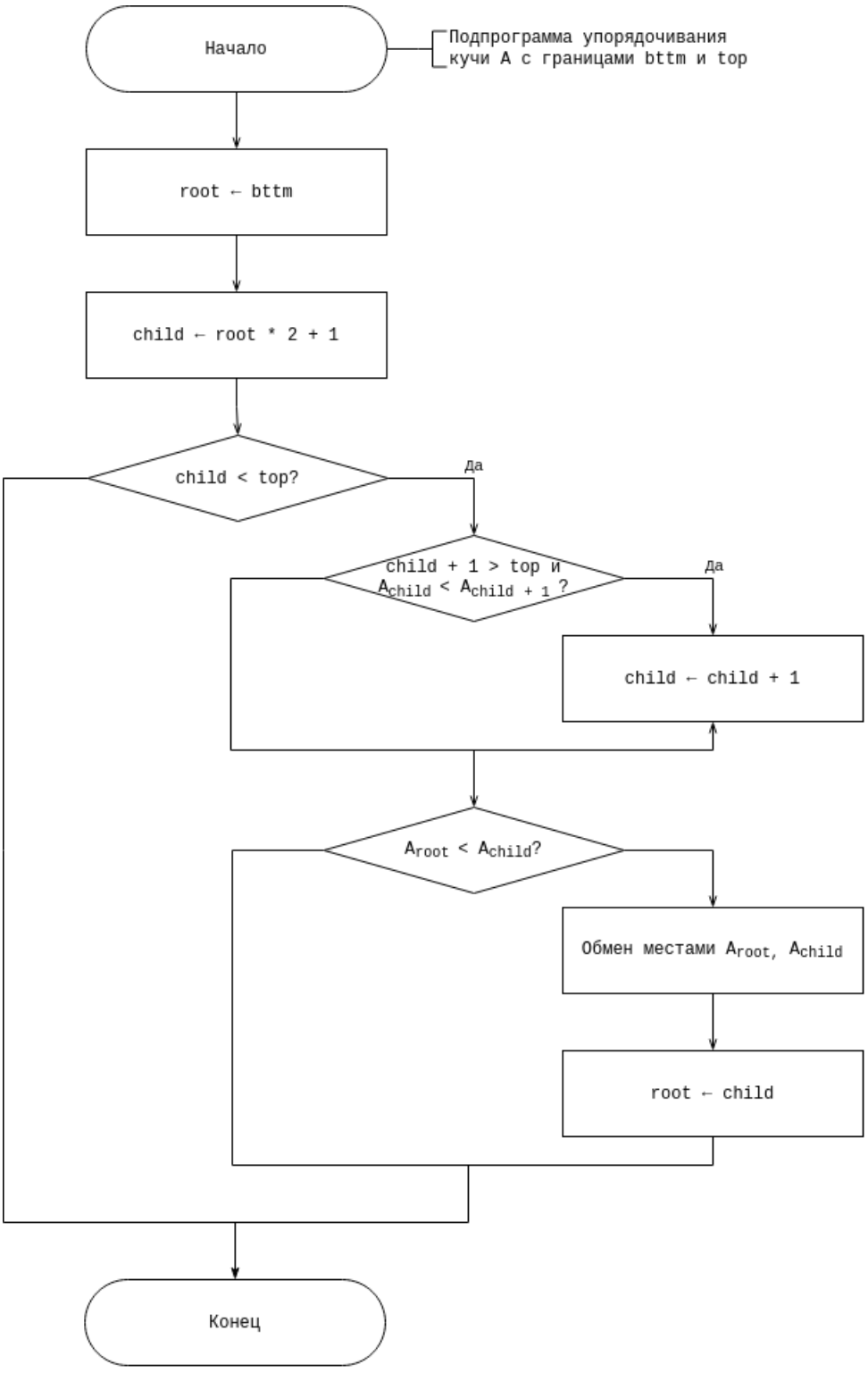

Рисунок 3.4 – Схема алгоритма конструирования бинарного дерева для пирамидальной сортировки

3.2 Модель оценки трудоемкости

Трудоемкость алгоритмов

Для получения функции трудоемкости алгоритма необходимо ввести модель оценки трудоемкости.
Трудоемкость "элементарных"операций оценивается следующим образом:

1. Трудоемкость 1 имеют следующие операции(3.1):

$$\begin{array}{l} +, -, =, <, > \\ <=, >=, ==, + =, - = \\ ++, --, [], \&\&, || \\ >>, << \end{array} \tag{3.1}$$

2. Трудоемкость 2 имеют следующие операции(3.2):

$$*, /, \backslash, \% \tag{3.2}$$

3. Трудоемкость конструкции ветвления определяется согласно формуле 3.3:

$$f_{if} = f_{\text{condition}} + \begin{bmatrix} min(f_{true}, f_{false}) & \text{в лучшем случае,} \\ max(f_{true}, f_{false}) & \text{в худшем случае.} \end{bmatrix} \tag{3.3}$$

4. Трудоемкость цикла расчитывается по формуле 3.4:

$$f_{cycle} = f_{init} + f_{cmp} + N(f_{body} + f_{inc} + f_{cmp}), \tag{3.4}$$

где
f_{init} – трудоемкость инициализации,
f_{body} – трудоемкость тела цикла,
f_{iter} - трудоемкость инкремента,
f_{cmp} - трудоемкость сравнения,
N – количество повторов.

Трудоемкость сортировки вставками

Трудоемкость сортировки вставками рассчитывается по формуле 3.5:

$$T_{ins} = f_{outer} + f_{inner} \tag{3.5}$$

где f_{outer} – трудоемкость внешнего цикла, f_{inner} – трудоемкость внутреннего. Трудоемкость внутреннего цикла вычисляется соответственно формуле 3.6:

$$f_{inner} = f_{init} + f_{cmp} + N(f_{body} + f_{inc} + f_{cmp}) \tag{3.6}$$

После подстановки значений стоимостей операций из списка 3.2, выражение для вычисления трудоемкости внутреннего цикла примет вид 3.7:

$$f_{inner} = 0 + 4 + N(5 + 1 + 4) \tag{3.7}$$

Лучшим случаем для сортировки является отсортированный массив, в этом случае N в формуле 3.7 принимает нулевое значение. В худшем случае (массив отсортирован в обратном порядке) цикл будет содержать $N - 1$ итераций. В общем случае выражение 3.6 принимает вид(3.8):

$$f_{inner} = 4 + \begin{bmatrix} 0 & \text{л. с.}, \\ 10(N-1) & \text{х. с.}. \end{bmatrix} \tag{3.8}$$

Трудоемкость внешнего цикла вычисляется аналогично 3.6. Однако, тело цикла содержит только оператор ветвления, соответственно, представить цикл можно следующей формулой(3.9):

$$f_{outer} = f_{init} + f_{cmp} + N(f_{if} + f_{inc} + f_{cmp}) \tag{3.9}$$

Тело внутреннего цикла после подстановки значений, согласно формуле стоимости ветвления из списка 3.2, имеет следующий вид(3.10):

$$f_{if} = 4 + \begin{bmatrix} min(7 + 4 + 10N, 0) & \text{л. с}, \\ max(7 + 4 + 10N, 0) & \text{х. с.} \end{bmatrix} \tag{3.10}$$

В любом случае у внешнего цикла будет $n-1$ прогонов. Значит, трудоемкость внешнего цикла будет равна (3.11)

$$f_{outer} = 2 + (n-1)(2 + f_{if}) \tag{3.11}$$

В результате:
В лучшем случае время работы составит 3.12:

$$T_{ins} = 4 + 0 + 2 + (n-1)(2+0) = 4 + 2n \approx 2n = O(n) \tag{3.12}$$

В худшем случае 3.13:

$$\begin{aligned} &T_{ins} = 4 + 10n + 2 + (n-1)(11 + 10n) = \\ &6 + (n-1)(11 + 10(n-1)) = 10n^2 + n + 5 \approx 10n^2 = O(n^2) \end{aligned} \tag{3.13}$$

Трудоемкость сортировки выбором

Трудоемкость сортировки выбором можно описать в виде формулы 3.14:

$$T_{sel} = 1 + f_{outer} \tag{3.14}$$

Внешний цикл f_{outer} в любом случае будет содержать n итераций. Его трудоемкость равна:

$$f_{outer} = 2 + N(f_{search} + c_{exch} + 3) \tag{3.15}$$

Здесь f_{search} – цикл поиска максимального ключа. Он будет содержать $n-1$ итераций в любом случае. Трудоемкость можно определить аналогично предыдущим случаям. После подстановки значений формула примет вид 3.16:

$$f_{search} = 2 + N(2 + f_{body}) \tag{3.16}$$

Здесь тело цикла представляет собой оператор ветвления, поэтому f_{body} можно представить как 3.17:

$$f_{if} = 3 + \begin{cases} 0 & \text{л. с.}, \\ 1 & \text{х. с.}. \end{cases} \tag{3.17}$$

c_{exch} – стоимость операции обмена, которая численно равна 8. В итоге: Для лучшего случая формула примет вид 3.18:

$$1 + 2 + n(2 + (n-1)(2+3+0)) = 5n^2 - 3n + 3 \approx 5n^2 = O(n^2) \tag{3.18}$$

Для худшего случая(3.19):

$$1 + 2 + n(2 + (n-1)(2+3+1)) = 6n^2 - 4n + 3 \approx 5n^2 = O(n^2) \tag{3.19}$$

3.3 Вывод

Для алгоритма сортировки вставками время работы линейно зависит от размера входных данных в лучшем случае и является квадратичной функцией от размера входных данных в худшем. В случае сортировки выбором время сортировки растет квадратично относительно количества элементов.

Алгоритмы были проанализированы с точки зрения временных затрат, были построены схемы алгоритмов.

4 | Технологический раздел

4.1 Требования к ПО

Программа должна отвечать следующим требованиям:

- ПО корректно реагирует на любые действия пользователя;
- ПО возвращает полученный массив;
- Время отклика программы на любое действие пользователя должно быть приемлемым.

4.2 Средства реализации

Для реализации ПО был выбран компилируемый многопоточный язык программирования Golang, поскольку язык отличается легкой и быстрой сборкой программ, автоматическим управлением памяти и понятным синтаксисом. В качестве среды разработки была выбрана среда VS Code, написание сценариев осуществлялось утилитой make.

4.3 Листинги кода

Реализация алгоритмов

Утилиты, требуемые для решения задачи продемонстрированы на листингах 4.1, 4.2.

Листинг 4.1 – объявление типа "целочисленный срез"

```
1    type IArray []int
```

Листинг 4.2 – Программный код утилиты для генерации срезов

```
    func Slice(size int, sorted bool) []int {
        slice := make([]int, size, size)
        rand.Seed(time.Now().UnixNano())
        for i := 0; i < size; i++ {
            slice[i] = rand.Intn(999) - rand.Intn(999)
        }
        if sorted {
            sort.Ints(slice)
        }
        return slice
    }
```

Листинги 4.3, 4.4, 4.5 демонстрируют программный код проанализированных сортировок.

Листинг 4.3 – Программный код сортировки вставками

```
func InsertionSort(items IArray) IArray {
    for i := 1; i < len(items); i++ {
        if items[i] < items[i-1] {
            j := i - 1
            key := items[i]
            for j >= 0 && items[j] > key {
                items[j+1] = items[j]
                j--
            }
            items[j+1] = key
        }
    }
    return items
}
```

Листинг 4.4 – Программный код сортировки выбором

```
func SelectionSort(items IArray) IArray {
    size := len(items)
    for i := 0; i < size; i++ {
        max_i := 0
        for j := 1; j < size - i; j++ {
            if items[j] > items[max_i] {
                max_i = j
            }
        }
        items[size - i - 1], items[max_i] =
                items[max_i], items[size - i - 1]
    }
    return items
}
```

Листинг 4.5 – Программный код пирамидальной сортировки

```
func HeapSort(items IArray) IArray {

    for i := (len(items) - 1) / 2; i >= 0; i-- {
        _siftDown(items, i, len(items))
    }

    for i := len(items) - 1; i > 0; i-- {
        items[0], items[i] = items[i], items[0]
        _siftDown(items, 0, i)
    }
    return items
}
```

Листинг 4.6 – вспомогательная программа для сортировки кучей.

Листинг 4.6 – Программный код функции получения сортировочного дерева

```
func _siftDown(heap []int, bttm, top int) {
    root := bttm
    for {
        child := root*2 + 1
        if child >= top {
            break
        }
        if child + 1 < top \&\& heap[child] < heap[child + 1] {
            child++
        }
        if heap[root] < heap[child] {
            heap[root], heap[child] = heap[child], heap[root]
            root = child
        } else {
            break
        }
    }
}
```

Тестирование ПО

Таблица 4.1 – Тестовые данные

№	Входной массив	Результат	
1.	«»	Вставки	«»
		Выбор	«»
		Куча	«»
2.	-26 -652 -328 -373 332	Вставки	-652 -373 -328 -26 332
		Выбор	-652 -373 -328 -26 332
		Куча	-652 -373 -328 -26 332
3.	-5 163 243 501 736	Вставки	-5 163 243 501 736
		Выбор	-5 163 243 501 736
		Куча	-5 163 243 501 736
4.	705 338 -226 -447 -519	Вставки	-519 -447 -226 338 705
		Выбор	-519 -447 -226 338 705
		Куча	-519 -447 -226 338 705

4.4 Вывод

На основе схем из конструкторского раздела были разработаны программные реализации требуемых алгоритмов.

5 | Исследовательская часть

5.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими техническими характеристиками:

- Операционная система Ubuntu 20.04.1 LTS;
- Память 7 GiB
- Процессор Intel(R) Core(TM) i3-8145U CPU @ 2.10GHz

Во время тестирования устройство было подключено к блоку питания и не нагружено никакими приложениями, кроме встроенных приложений окружения, окружением и системой тестирования. Процессорное время замерялось с помощью вставки на языке C (5.1).

Листинг 5.1 – Вставка языка C для замера времени и ее использование

```
#include <pthread.h>
#include <time.h>
#include <stdio.h>

static double GetCpuTime() {
    struct timespec time;
    if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time)) {
        perror("clock_gettime");
        return 0;
    }

    long seconds = time.tv_sec;
    long nanoseconds = time.tv_nsec;
    double elapsed = seconds + nanoseconds*1e-9;

    return elapsed;
}
```

5.2 Таблица времени выполнения алгоритмов

Таблица 5.1 – Временные замеры работы алгоритмов

N	Random, ns $\cdot 10^{-7}$			Best, ns $\cdot 10-7$			Worst, ns $\cdot 10^{-7}$		
	Insert	Select	Heap	Insert	Select	Heap	Insert	Select	Heap
50	7.99	16.60	9.94	5.00	18.90	10.10	14.10	16.60	9.95
100	22.20	60.10	18.00	5.45	53.90	18.60	39.30	58.60	16.50
150	46.70	131.00	26.30	7.00	109.00	27.20	76.70	121.00	24.60
200	73.80	230.00	34.70	6.86	192.00	38.10	131.00	219.00	32.20
250	107.00	333.00	45.10	6.79	282.00	46.30	190.00	310.00	41.00
300	155.00	468.00	54.20	8.69	398.00	59.00	269.00	447.00	52.40
350	198.00	641.00	71.50	8.54	551.00	78.50	373.00	614.00	63.90
400	257.00	810.00	102.00	8.75	694.00	111.00	465.00	767.00	86.60
450	308.00	1010.00	139.00	10.20	868.00	140.00	584.00	960.00	116.00
500	382.00	1230.00	178.00	10.70	1070.00	168.00	717.00	1180.00	148.00
500	397.00	1240.00	177.00	11.70	1070.00	168.00	717.00	1180.00	147.00
600	546.00	1770.00	240.00	11.80	1570.00	222.00	1060.00	1720.00	204.00
700	738.00	2360.00	313.00	13.30	2140.00	275.00	1430.00	2360.00	257.00
800	957.00	3030.00	378.00	14.20	2750.00	332.00	1830.00	2980.00	308.00
900	1170.00	3760.00	447.00	16.40	3430.00	380.00	2310.00	3780.00	360.00
1000	1430.00	4580.00	520.00	16.10	4220.00	434.00	2820.00	4600.00	415.00
1100	1730.00	5550.00	602.00	16.90	5160.00	515.00	3490.00	5680.00	472.00
1200	2080.00	6540.00	685.00	18.40	6100.00	563.00	4070.00	6610.00	526.00
1300	2470.00	7580.00	736.00	20.10	7100.00	631.00	4760.00	7790.00	584.00
1400	2840.00	8730.00	844.00	22.40	8220.00	693.00	5590.00	8990.00	653.00
1500	3270.00	9950.00	894.00	23.20	9420.00	742.00	6420.00	10300.00	691.00
1600	3700.00	11300.00	996.00	23.70	10700.00	811.00	7210.00	11800.00	758.00
1700	4170.00	12700.00	1050.00	25.20	12000.00	856.00	8100.00	13300.00	808.00
1800	4700.00	14200.00	1130.00	27.20	13500.00	947.00	9100.00	14900.00	888.00
1900	5180.00	15800.00	1200.00	27.10	15000.00	987.00	10100.00	16800.00	929.00
2000	5780.00	17400.00	1270.00	30.20	16600.00	1040.00	11100.00	18500.00	985.00
2100	6480.00	19200.00	1330.00	31.80	18300.00	1100.00	12300.00	20300.00	1030.00
2200	7070.00	21000.00	1420.00	31.20	20100.00	1160.00	13500.00	22600.00	1110.00
2300	7590.00	22800.00	1480.00	33.30	21900.00	1210.00	14700.00	25300.00	1130.00
2400	8050.00	24800.00	1560.00	33.90	23900.00	1260.00	16000.00	27200.00	1200.00
2500	8820.00	26900.00	1630.00	36.40	25900.00	1310.00	17300.00	30600.00	1270.00

Результаты замеров приведены в таблице 5.1. Замеры проводились 10000 раз, время усреднялось.

5.3 Графики функций

Теоретически было установлено, что как в худшем случае(3.19), так и в лучшем(3.18) сортировка выбором имеет трудоемкость $O(n^2)$. Экспериментальное подтверждение этому продемонстрировано на графике 5.1.

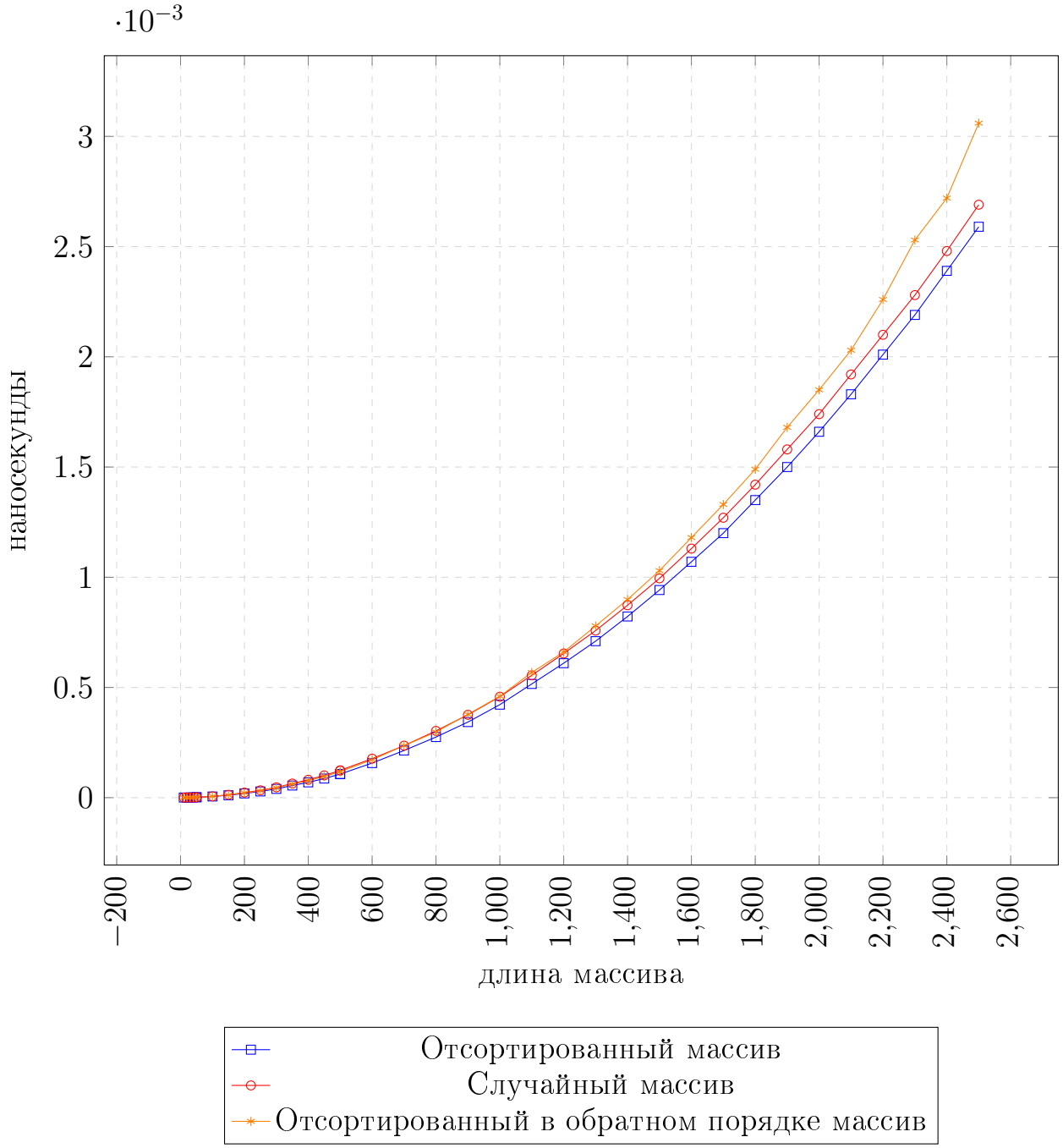

Рисунок 5.1 – Зависимость времени сортировки выбором от размера массива

Зависимости $O(n)$ и $O(n^2)$, экспериментально доказанная для вставок (3.12, 3.13) продемонстрирована графически на рисунке 5.2.

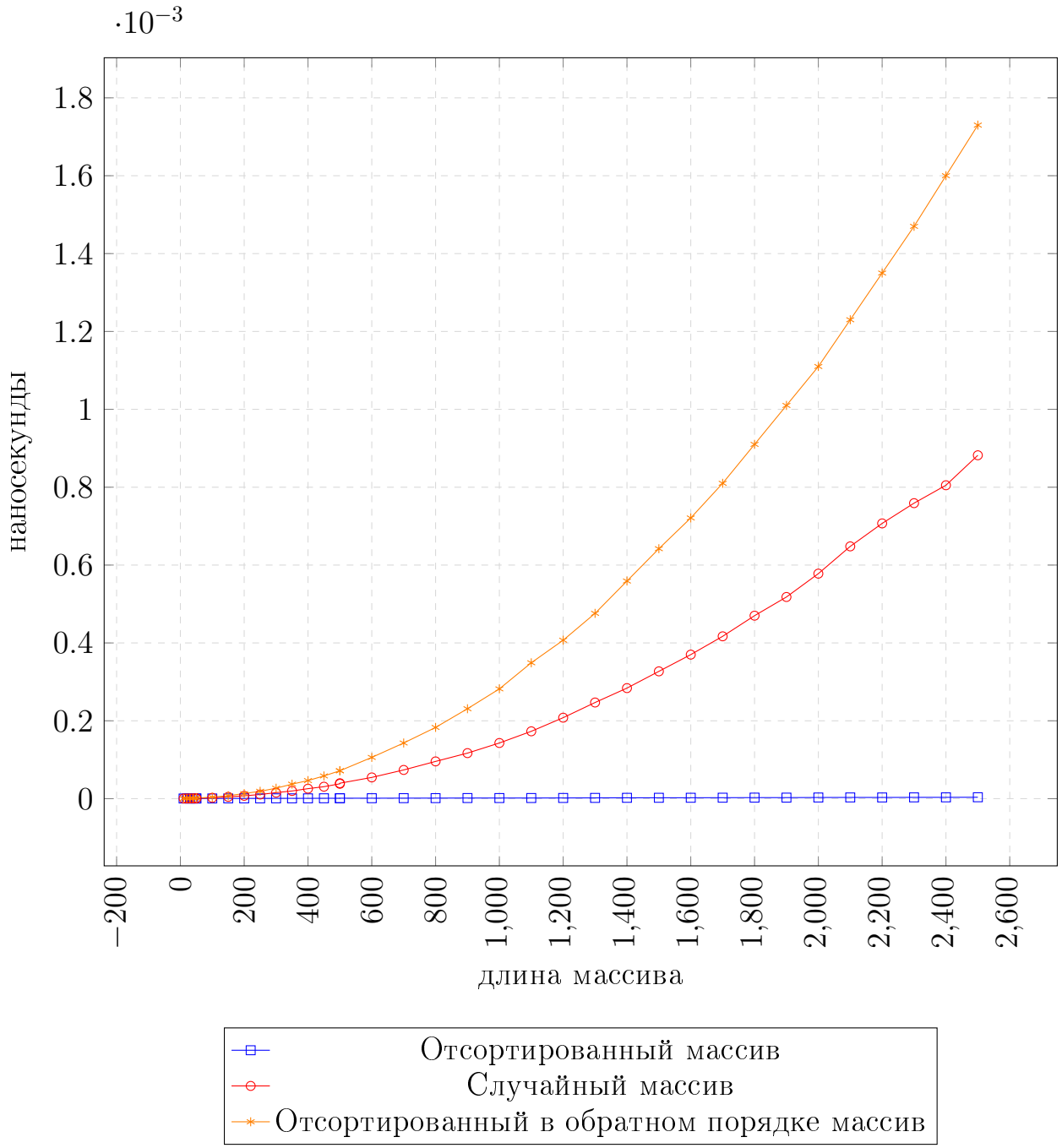

Рисунок 5.2 – Зависимость времени сортировки вставками от размера массива

Здесь следует рассмотреть отсортированный массив более детально (5.3) для получения наглядной картины для зависимости $O(n)$:

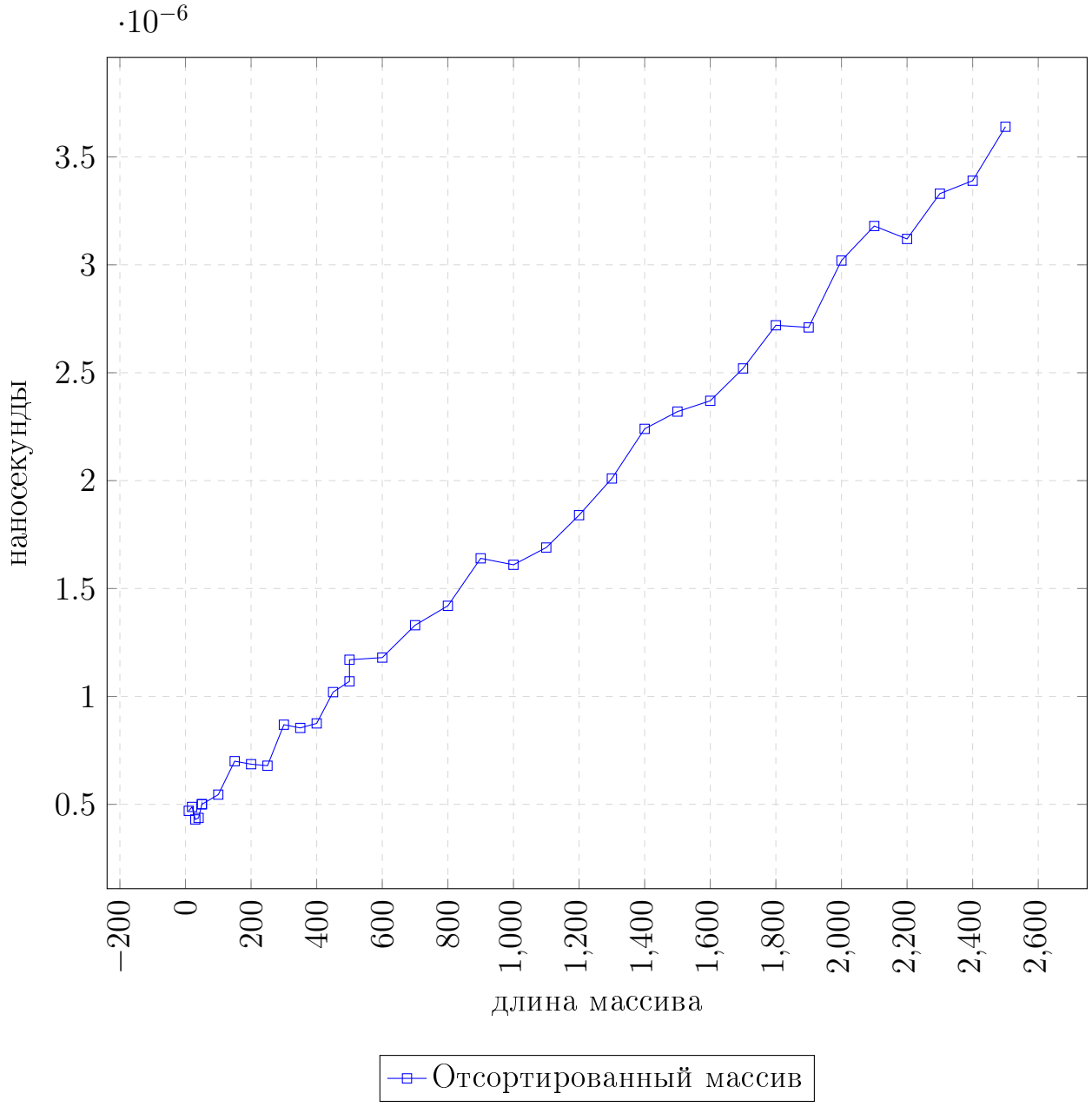

Рисунок 5.3 – Зависимость времени работы сортировки вставками от длины массива на отсортированных данных

Для пирамидальной сортировки, трудоемкость которой в лучшем случае и в худшем $O(n \cdot \log n)$, продемонстрирована на рисунке 5.4.

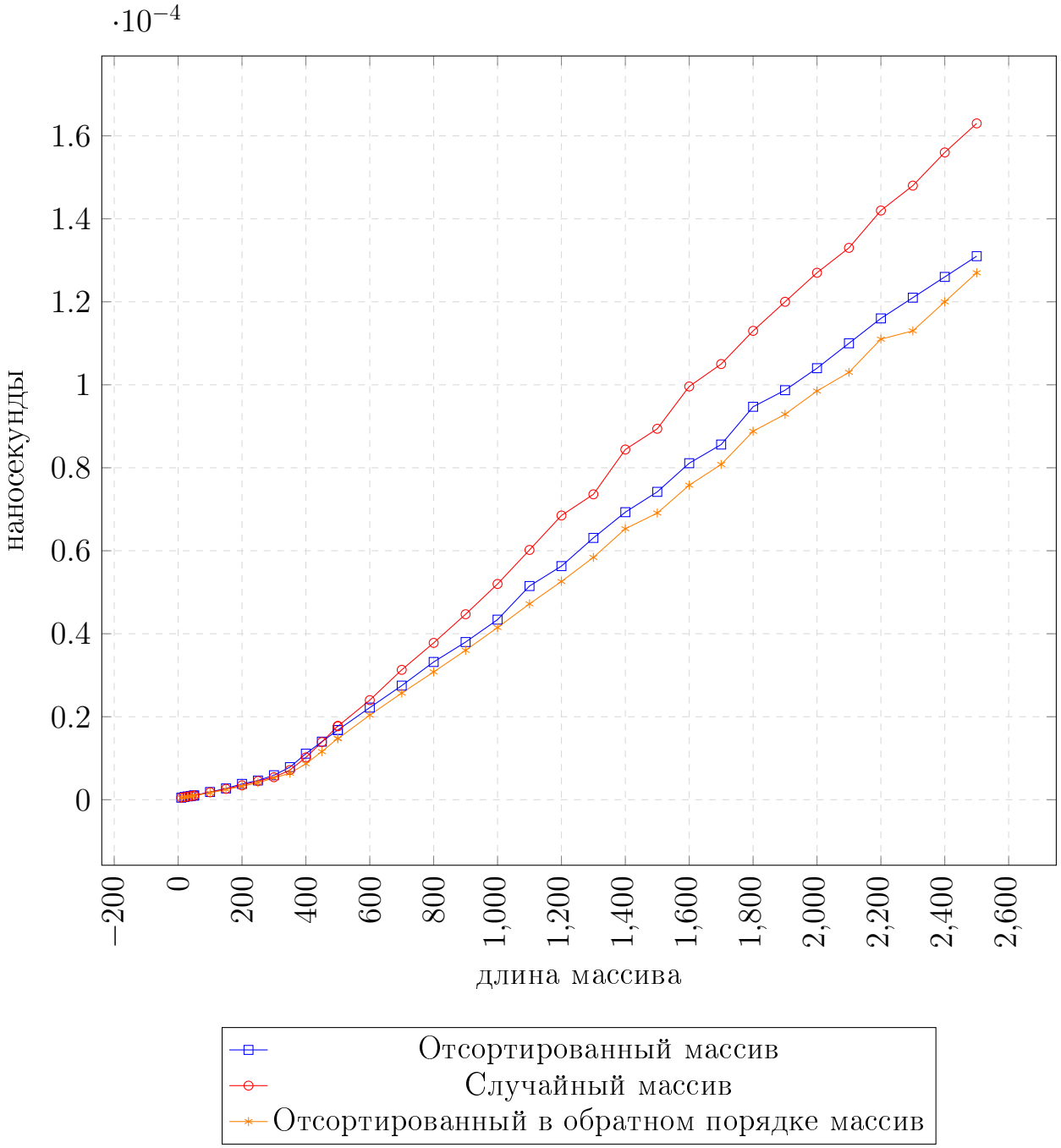

Рисунок 5.4 – Зависимость времени сортировки вставками от размера массива

Экспериментальное сравнение работы алгоритмов сортировки друг с другом показало следующие результаты:

1. Сортировка выбором во всех случаях(5.5, 5.6, 5.7) работает более медленно, чем остальные рассматриваемые. Особенно это видно на больших размерах массива – на массиве в 2000 случайных чисел она работает на 66% медленнее, чем вставки и на 92% медленнее, чем куча.

2. Куча, однако, работает намного быстрее (5.5) – например, на массиве из 1600 случайных чисел пирамидальная сортировка работает на 93% быстрее, чем сортировка выбором и на 82% быстрее, чем сортировка вставками.

3. Сортировка вставками работает намного быстрее на отсортированных данных (5.6), чем на обратно отсортированных (5.7), но даже в этом случае она превосходит выбор: на отсортированных данных в 1900 элементов она превосходит выбор на 99%, а на обратно отсортированных – на 39%.

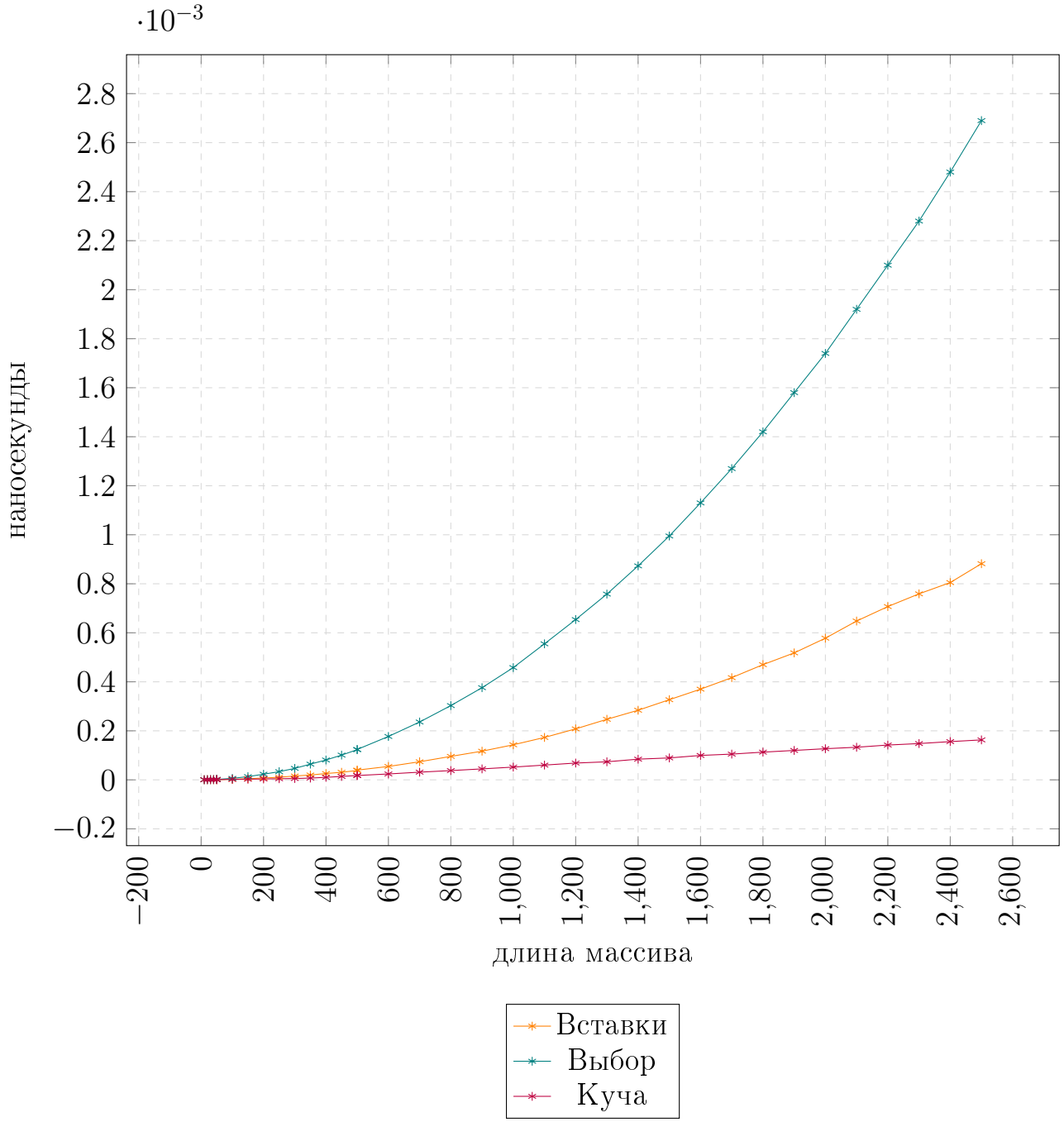

Рисунок 5.5 – Зависимость времени сортировок на случайных данных от размера массива

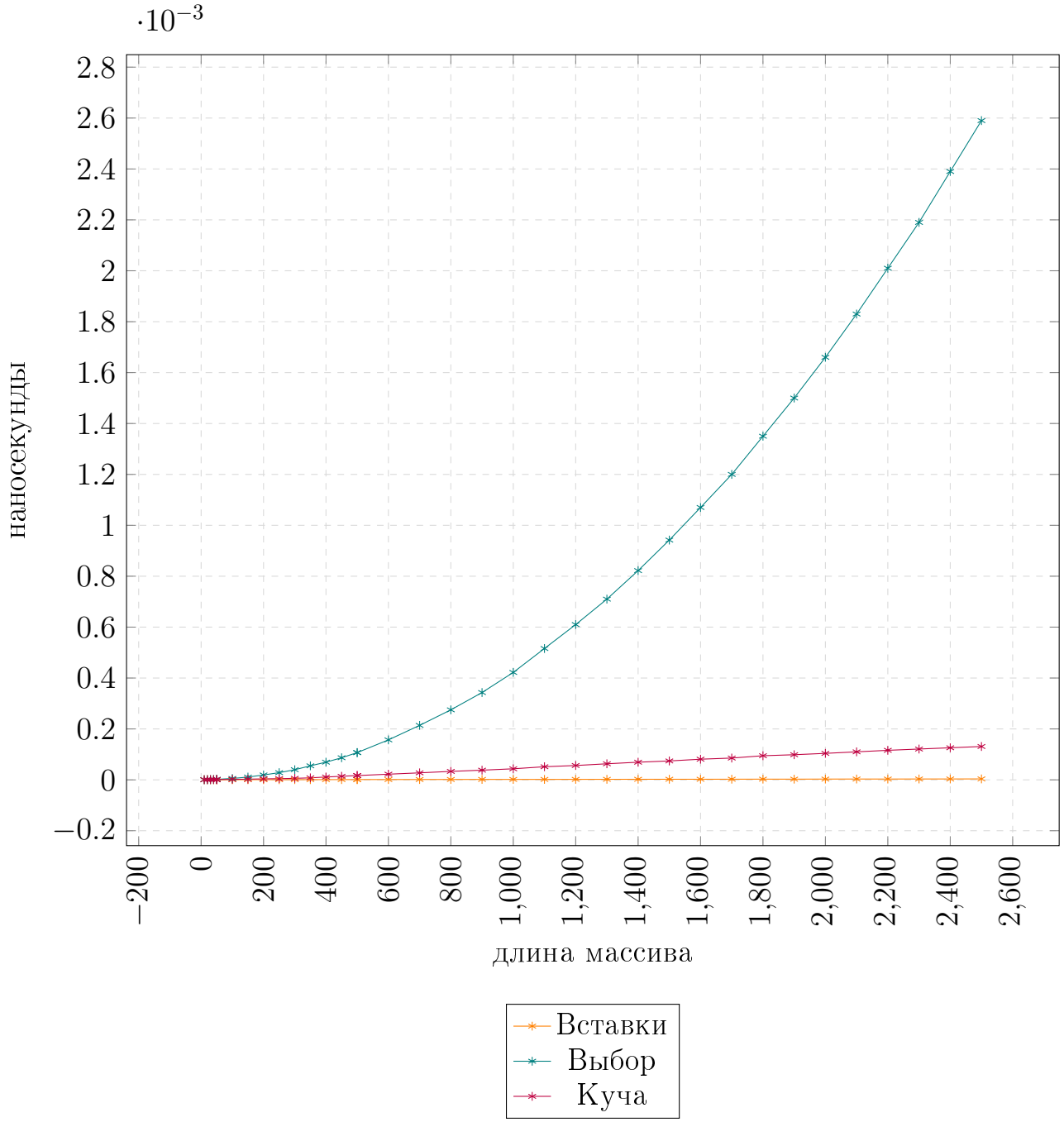

Рисунок 5.6 – Зависимость времени сортировок на отсортированных данных от размера массива

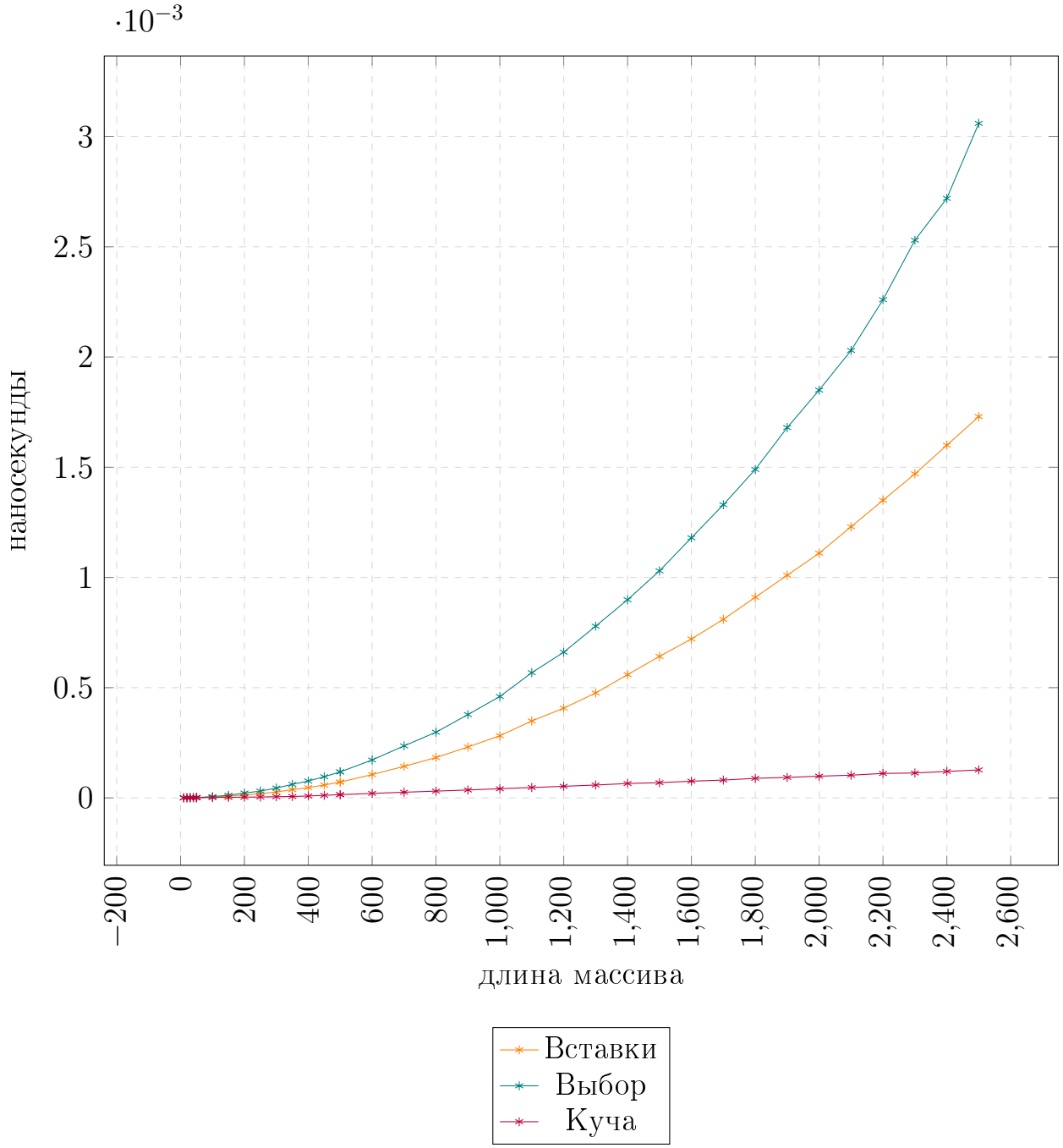

Рисунок 5.7 – Зависимость времени сортировок на обратно отсортированных данных от размера массива

Вывод

Сортировки были протестированы в лучшем, среднем и худшем случаях. Как было уже замечено, куча работает быстрее остальных сортировок – сортировка работает на 93% быстрее, чем сортировка выбором и на 82%

быстрее, чем сортировка вставками. Сортировка выбором работает медленнее остальных во всех рассмотренных случаях. Однако, с ростом количества элементов на обратно отсортированных данных сортировка вставками работает практически так же, как и сортировка выбором – на массиве в 5000 элементов их разница составляет всего лишь 43%.

6 | Заключение

Выбор сортировочного алгоритма – такая же трудная задача для программиста, как и сама сортировка. В данной лабораторной работе сортировки были проанализированы по их временным характеристикам, а также выделены их преимущества и недостатки. Сортировка выбором оказалась самой медленной из всех сортируемых. Пирамидальная сортировка сильно выигрывает по времени. Однако, она неустойчива и на почти отсортированных массивах работает столь же долго, как и на случайных данных.

Хотя сортировка вставками и явно проигрывает пирамидальной, она, в отличие от двух других рассматриваемых сортировок является устойчивой. Однако, ее временные показатели сильно ухудшаются с ростом количества сортируемых элементов – на массиве размера 1000 элементов она работает на 99,43% медленнее, если данные отсортированы в обратном порядке.

7 | Список использованных источников

[1] Дональд Эрвин Кнут. “Искусство программирования: Сортировка и поиск, Том 3”. в: под ред. В. Т. Тертышный (гл. 5) и И. В. Красиков (гл. 6). Москва: Вильямс, 2007, с. 832.

[2] Дональд Эрвин Кнут. “Искусство программирования: Сортировка и поиск, Том 3”. в: под ред. В. Т. Тертышный (гл. 5) и И. В. Красиков (гл. 6). Москва: Вильямс, 2007. гл. 5.2.1.

[3] Дональд Эрвин Кнут. “Искусство программирования: Сортировка и поиск, Том 3”. в: под ред. В. Т. Тертышный (гл. 5) и И. В. Красиков (гл. 6). Москва: Вильямс, 2007. гл. 5.2,3, с. 143.

[4] Sedgewick R. Schaffer R. “The Analysis of Heapsort”. в: *Journal of Algorithms* (1993).