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»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №3

Группа ИУ7-51Б

Тема работы **Алгоритмы поиска в массиве**

Студент Баранов Николай Алексеевич

Преподаватель Волкова Лилия Леонидовна

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **4**

1 Аналитическая часть **5**

1.1 Алгоритм линейного поиска 5

1.2 Алгоритм бинарного поиска 5

1.3 Алгоритм сортировки 6

1.3.1 Быстрая сортировка 6

1.3.2 Сортировка слиянием 6

1.3.3 Пирамидальная сортировка 7

1.3.4 Выбор алгоритма сортировки 7

1.4 Вывод 7

2 Конструкторская часть **8**

2.1 Описание алгоритмов 8

2.2 Выводы 10

3 Технологическая часть **11**

3.1 Средства реализации 11

3.2 Реализация алгоритмов 11

3.3 Функциональные тесты 13

3.4 Вывод 13

4 Исследовательская часть **14**

4.1 Замеры количества сравнений 14

4.2 Вывод 16

ЗАКЛЮЧЕНИЕ **17**

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ **18**

ВВЕДЕНИЕ

В данной лабораторной работе рассматриваются алгоритм поиска в массиве полным перебором, а также алгоритм бинарного поиска.

Цель работы: исследование алгоритма линейного поиска в массиве, а также алгоритма бинарного поиска.

1) описать алгоритмы поиска в массиве и выбрать алгоритм сортировки массива;
2) написать программу, реализующую эти алгоритмы;
3) провести анализ затрат реализаций алгоритмов в терминах числа сравнений.

1 Аналитическая часть

1.1 Алгоритм линейного поиска

Поиск нужного элемента сводится к проходу с начала массива либо до совпадения, либо до конца массива [1].

Пусть дан массив длины N с индексами от 0 до $N-1$. При расположении элемента на позиции с индексом i потребуется $i+1$ сравнение. В случае отсутствия элемента в массиве потребуется N сравнений. Лучшим случаем для данного алгоритма будет нахождение нужного элемента на позиции 0, так как потребуется 1 сравнение. Худших случаев 2 - отсутствие элемента в массиве и расположение на позиции $N-1$. Для них потребуется N сравнений. Трудоёмкость в среднем рассчитывается по формуле $\frac{\sum_{i=1}^{N} i+N}{N+1} = \frac{N}{2} + \frac{N}{N+1}$.

1.2 Алгоритм бинарного поиска

Данный алгоритм работает только для отсортированного массива [1]. Поиск начинается со среднего элемента. Возможны 2 варианта алгоритма.

В 1 варианте поиск останавливается при совпадении. В противном случае происходит ещё одно сравнение элемента с текущим, чтобы отбросить половину массива, что возможно благодаря тому, что массив отсортирован, так как элементы с меньшими индексами будут меньше текущего, а с большими – больше. Далее поиск повторяется либо до нахождения позиции, либо до сокращения размера диапазона поиска до 0.

Пусть дан массив длины N. Лучшим случаем для данного алгоритма будет нахождение нужного элемента при первом сравнении. Отсутствие элемента в массиве является худшим случаем и требует $2 * log_2 N$ сравнений, что при больших N меньше, чем трудоёмкость в среднем у линейного поиска.

Данный вариант возвращает первую найденную позицию. Это означает, что при наличии нескольких элементов с одинаковыми значениями неизвестно, какой из них вернется в качестве результата.

В 2 варианте поиск не останавливается при совпадении. Вместо этого

каждый раз происходит только 1 сравнение, чтобы отбросить половину массива, и так до тех пор, пока размер диапазона поиска не уменьшится до 1. После этого происходит ещё одно сравнение для определения правильности поиска. Из-за этого во всех случаях количество сравнений будет $log_2 N + 1$.

В данном варианте в зависимости от способа сравнения будет возвращаться либо наименьший индекс элемента, либо наибольший.

Для данной лабораторной работы выбран 1 вариант, так как по условию все числа в массиве различные.

1.3 Алгоритм сортировки

1.3.1 Быстрая сортировка

На каждом шаге алгоритма выбирается опорный элемент, после чего массив делится на 2 части [2]. В одной из частей элементы больше опорного, в другой – меньше. Далее операция повторяется с частями, пока их размер не достигнет 1.

В худшем случае алгоритм работает за $O(N^2)$, однако в среднем выходит $O(Nlog_2 N)$.

1.3.2 Сортировка слиянием

В данном алгоритме массив сначала делится на 2 равные части [2]. Эта операция повторяется для частей до тех пор, пока их размер не станет равным 1. Затем части массива объединяются в порядке возрастания элементов. Благодаря тому, что объединение начинается с частей размером 1, все части при слиянии уже будут отсортированными.

Во всех случаях алгоритм работает за $O(Nlog_2 N)$, но при этом требует память под ещё один массив.

1.3.3 Пирамидальная сортировка

Для данного алгоритма используется структура данных двоичная куча [2]. Сначала на массиве строится двоичная куча для поиска максимума. Затем в цикле с последнего элемента вершина кучи обменивается с текущим элементом, после чего размер кучи уменьшается на 1 и восстанавливается свойство кучи – каждый родитель не меньше своих потомков.

Во всех случаях алгоритм работает за $O(Nlog_2N)$.

1.3.4 Выбор алгоритма сортировки

В таблице 1.1 представлены результаты сравнения алгоритмов сортировки:

Таблица 1.1 — Результаты сравнения алгоритмов сортировки

№	Название	л. с.	ср. с.	х. с.	Не нужен дополнительный массив
1	Быстрая сортировка	$O(Nlog_2N)$	$O(Nlog_2N)$	$O(N^2)$	+
2	Сортировка слиянием	$O(Nlog_2N)$	$O(Nlog_2N)$	$O(Nlog_2N)$	-
3	Пирамидальная сортировка	$O(Nlog_2N)$	$O(Nlog_2N)$	$O(Nlog_2N)$	+

В результате сравнения была выбрана пирамидальная сортировка.

1.4 Вывод

В данном разделе были описаны алгоритмы поиска в массиве, а также выбран алгоритм сортировки.

2 Конструкторская часть

2.1 Описание алгоритмов

Алгоритмы поиска (рисунки 2.1 и 2.2) в массиве получают на вход массив arr, размер массива n и значение value.

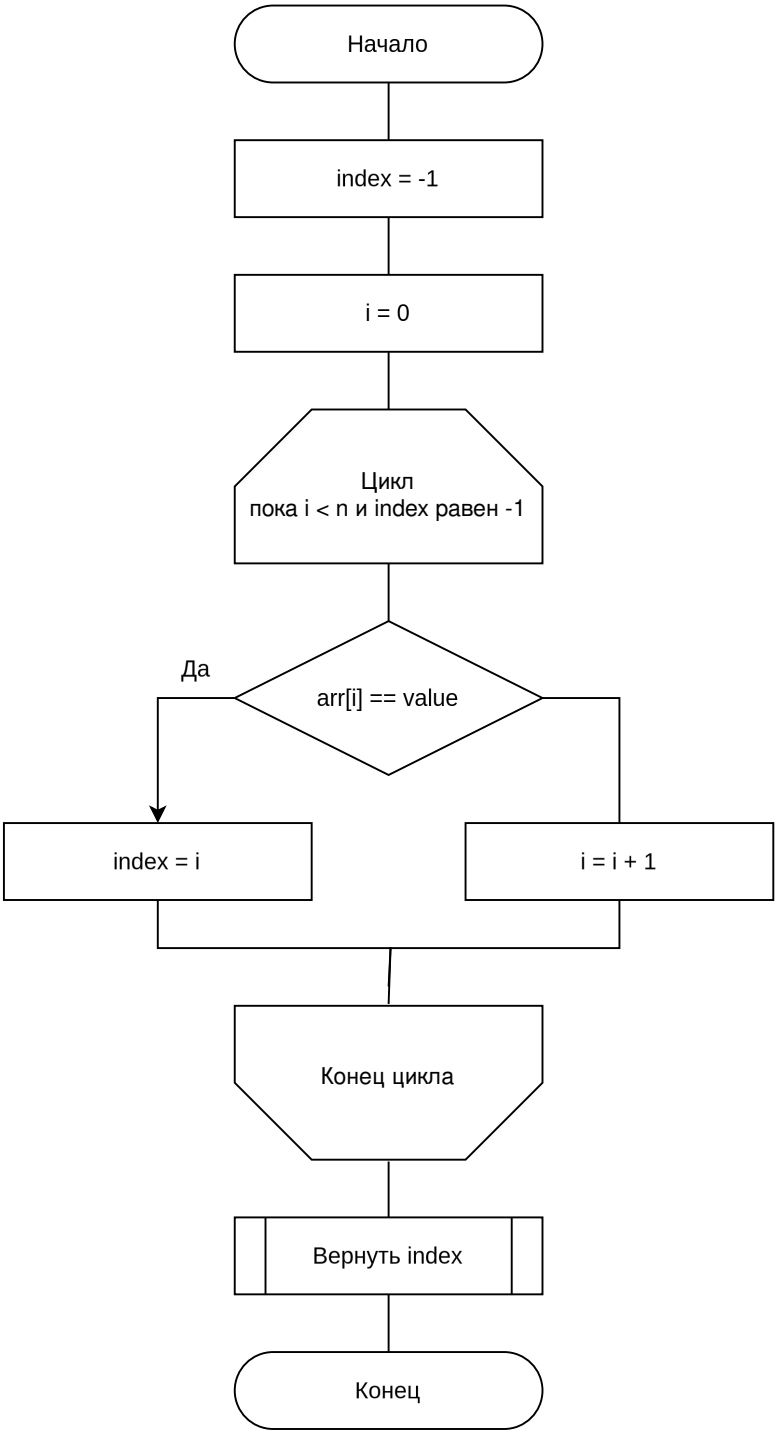

Рисунок 2.1 — Схема алгоритма линейного поиска в массиве

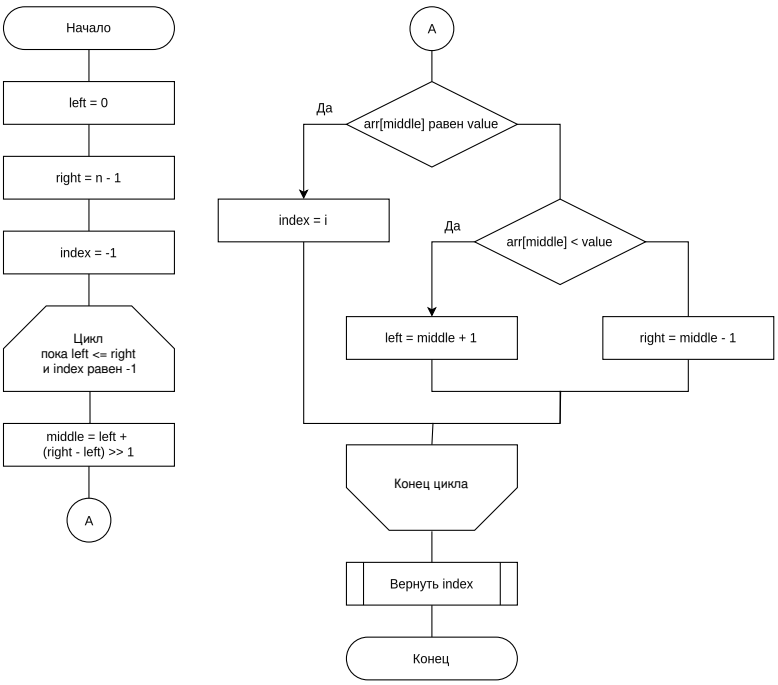

Рисунок 2.2 — Схема алгоритма бинарного поиска в массиве

Алгоритм пирамидальной сортировки (рисунок 2.3) получает на вход массив arr и размер массива n. В нём swap – функция, меняющая местами переменные, а heapify – функция восстановления свойства кучи.

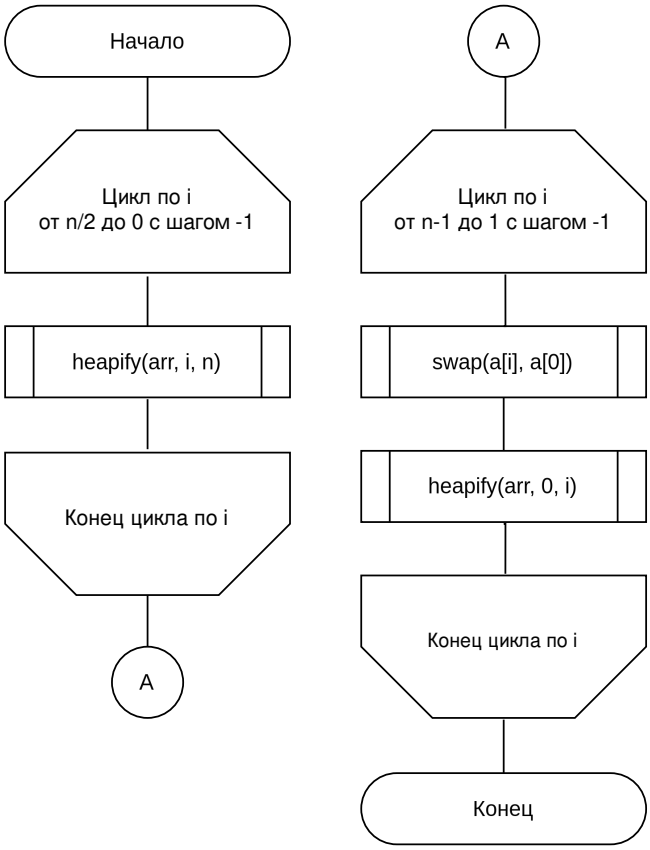

Рисунок 2.3 — Схема алгоритма пирамидальной сортировки

Алгоритм восстановления свойства кучи (рисунок 2.4) в вершине получает на вход массив arr, размер массива size и индекс index. В нём swap – функция, меняющая местами переменные.

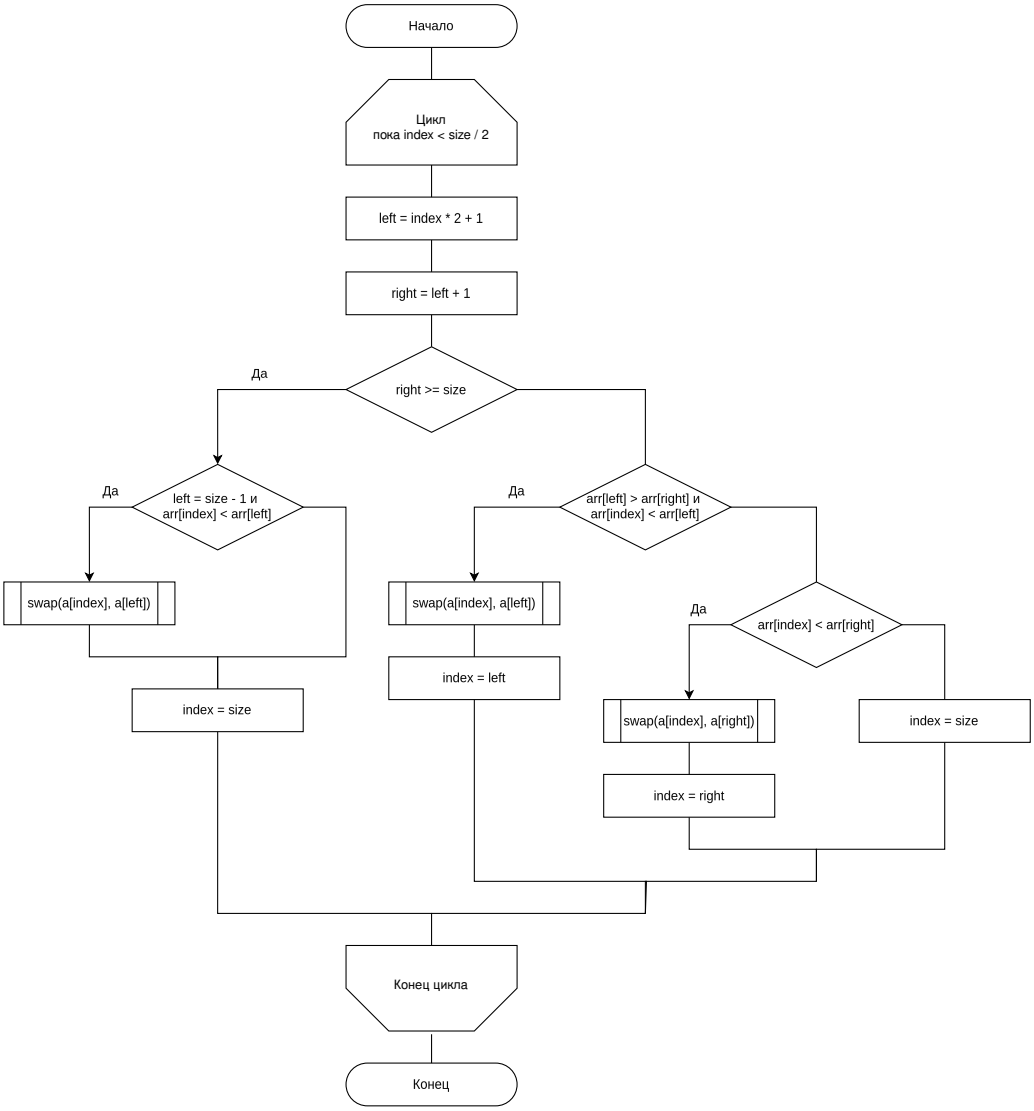

Рисунок 2.4 — Схема алгоритма восстановления свойства кучи в вершине

2.2 Выводы

В данном разделе были построены схемы алгоритмов и выбраны структуры данных.

3 Технологическая часть

3.1 Средства реализации

Для реализации данной лабораторной работы был выбран язык Java, так как он содержит необходимые средства для реализации алгоритмов.

3.2 Реализация алгоритмов

В листингах 3.1 и 3.2 представлены реализации алгоритмов поиска в массиве. В листинге 3.3 представлена реализация алгоритма пирамидальной сортировки.

Листинг 3.1 — Алгоритм линейного поиска в массиве

```
public int search(int value) {
    var result = -1;
    for (var i = 0; result == -1 && i < array.length; ++i) {
        if (array[i] == value) {
            result = i;
        }
    }
    return returnsComprasionCount ? result == -1 ? array.length :
        result + 1 : result;
}
```

Листинг 3.2 — Алгоритм бинарного поиска в массиве

```
public int binarySearch(int value) {
    var left = 0;
    var right = array.length - 1;
    var result = -1;
    var iterations = 0;
    while (result == -1 && left <= right) {
        var middle = left + ((right - left) >> 1);
        ++iterations;
        if (array[middle] == value) {
            result = middle;
        } else if (value < array[middle]) {
```

```
            ++iterations;
            right = middle - 1;
        } else {
            ++iterations;
            left = middle + 1;
        }
    }
    return returnsComprasionCount ? iterations : result;
}
```

Листинг 3.3 — Алгоритм пирамидальной сортировки

```
public static void sort(int[] array) {
    for (var i = array.length >> 1; i >= 0; --i) {
        heapify(array, i, array.length);
    }
    for (var i = array.length - 1; i > 0; --i) {
        swap(array, 0, i);
        heapify(array, 0, i);
    }
}

private static void heapify(int[] array, int index, int size) {
    while (index < (size >> 1)) {
        var left = (index << 1) + 1;
        var right = left + 1;
        if (right >= size) {
            if (left == size - 1 && array[index] < array[left]) {
                swap(array, index, left);
            }
            break;
        } else if (array[index] < array[left] && array[left] >
           array[right]) {
            swap(array, index, left);
            index = left;
        } else if (array[index] < array[right]) {
            swap(array, index, right);
            index = right;
        } else {
            break;
        }
```

```
    }
}

private static void swap(int[] array, int i, int j) {
    array[i] ^= array[j];
    array[j] ^= array[i];
    array[i] ^= array[j];
}
```

3.3 Функциональные тесты

В таблице 3.1 представлены функциональные тесты. Тесты выполнялись для массива $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$. Первое число в результате показывает результат выполнения линейного поиска, второе – бинарного.

Таблица 3.1 — Результаты выполнения функциональных тестов

№	Число	Ожидаемый вывод	Полученный вывод	Описание теста
1	1	0 0	0 0	Первое число
2	10	9 9	9 9	Последнее число
3	7	6 6	6 6	Иное число
4	11	-1 -1	-1 -1	Не найдено

3.4 Вывод

В данном разделе был выбран язык программирования для написания программы, были рализованы все ранее описанные алгоритмы и описаны функциональные тесты.

4 Исследовательская часть

4.1 Замеры количества сравнений

Замеры проводились на массивах длины 1091. Каждый элемент массива во время замеров равен его индексу. Во время замеров для каждого элемента в массиве вызывался его поиск. Также был поиск элемента, отсутствующего в массиве. При установке в переменную returnsComprasionCount значения true все функции возвращают число сравнений вместо результата. Результаты представлены на рисунках 4.1, 4.2 и 4.1.

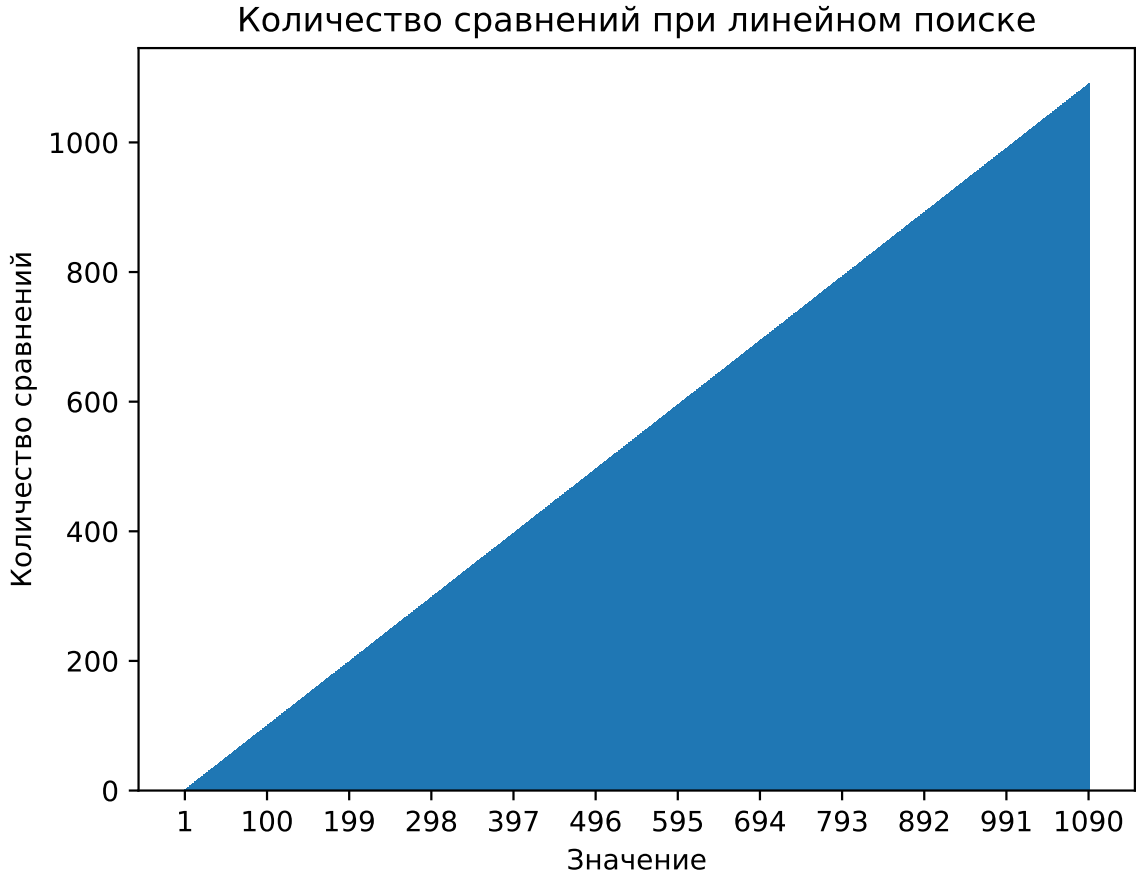

Рисунок 4.1 — Результаты замера числа сравнений для алгоритма линейного поиска

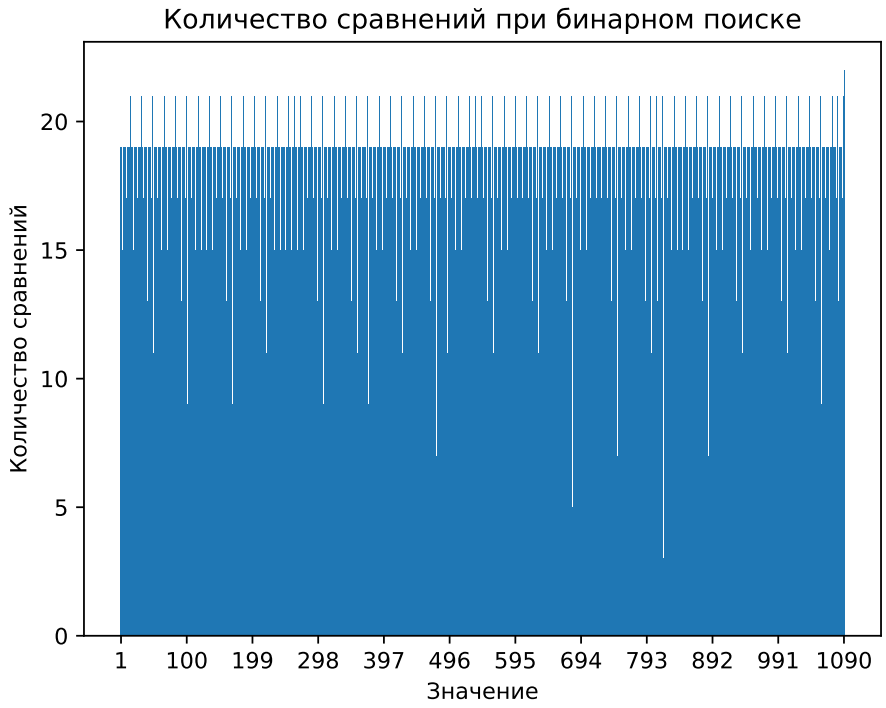

Рисунок 4.2 — Результаты замера числа сравнений для алгоритма бинарного поиска

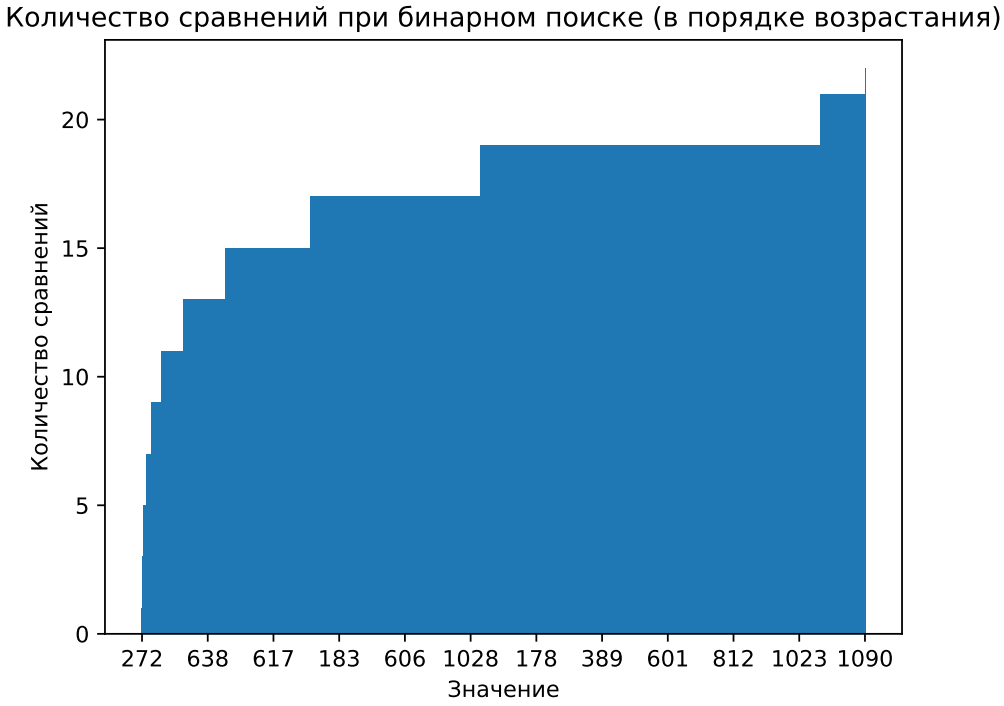

Рисунок 4.3 — Отсортированные результаты замера числа сравнений для алгоритма бинарного поиска

Результаты совпали с расчетами, алгоритм линейного поиска в худшем случае сделал больше 1000 сравнений, в то время как алгоритм бинарного поиска сделал меньше 25 сравнений.

4.2 Вывод

В данном разделе были проведены замеры числа сравнений. Алгоритм бинарного поиска оказался эффективнее алгоритма линейного поиска по числу сравнений.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы было выявлено, что алгоритм бинарного поиска в массиве эффективнее алгоритма линейного поиска по числу сравнений, однако требует, чтобы массив был отсортирован. Цель работы была достигнута, для чего были описаны алгоритмы поиска в массиве, выбран алгоритм сортировки массива, написана программу, реализующую эти алгоритмы и проведён анализ затрат реализаций алгоритмов в терминах числа сравнений.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. С.С. Ахматова. Алгоритмы поиска данных. Современные наукоемкие технологии №3, 2007. — С. 11–14.

2. В.В. Ландовский. Алгоритмы обработки данных. Новосибирск: Издательство НГТУ, 2018. — С. 11–17.