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

по курсу «Анализы Алгоритмов»

на тему: «Трудоемкость сортировок»

Студент группы ИУ7-56Б _____ (Подпись, дата) Чупахин М. Д. (Фамилия И.О.)

Преподаватель _____ (Подпись, дата) Волкова Л. Л. (Фамилия И.О.)

Москва — 2023 г.

Содержание

Введение

В данной лабораторной работе будет проведен анализ сортировок.

Сортировка - это переупорядочивание некой последовательности или кортежа в определенном порядке. Эта операция является важной частью обработки структурированных данных. Упорядоченное расположение элементов позволяет более эффективно работать с данными, особенно при поиске нужных элементов.

Существует множество алгоритмов сортировки, но каждый из них включает в себя следующие основные элементы:

— сравнение элементов, определяющее их порядок;

— перестановка для изменения местоположения элементов;

— алгоритм, который использует сравнение и перестановку для сортировки данных.

Каждый алгоритм обладает своими преимуществами, и его эффективность оценивается на основе ответов на следующие вопросы:

— какая средняя скорость сортировки этим алгоритмом;

— каковы лучший и худший случаи сортировки;

— проявляется ли "естественное"поведение алгоритма, т.е. увеличивается ли скорость сортировки с увеличением упорядоченности массива;

— является ли алгоритм стабильным, то есть сохраняет ли он порядок элементов с одинаковыми значениями.

Цель данной лабораторной работы заключается в описании и исследовании трудоемкости алгоритмов сортировки.

Ниже представлены задачи, которые необходимо выполнить для достижения поставленной цели.

1) Разработать программное обеспечение, реализующее следующие алгоритмы сортировки:

   — Блинная;

- Поразрядная;
- Бинарным деревом.

2) Оценить трудоемкость этих алгоритмов сортировки.

3) Измерить время выполнения алгоритмов.

4) Проанализировать затраты времени работы программы и выявить их зависимость от различных характеристик.

1 Аналитическая часть

В данном разделе мы рассмотрим три алгоритма сортировок: блинная, поразрядная, бинарным деревом.

1.1 Блинная сортировка

Блинная сортировка (англ. pancake sorting) [1] – это алгоритм сортировки, который использует операции переворачивания элементов в массиве для достижения упорядочивания элементов в правильном порядке. Суть этого алгоритма заключается в пошаговом перемещении наибольшего элемента массива в правильное положение, а затем уменьшении размера обрабатываемой части массива и повторении этого процесса до тех пор, пока весь массив не будет упорядочен.

Ниже представлены основные шаги блинной сортировки.

1) Начать с полного массива, который нужно отсортировать.

2) Найти индекс максимального элемента в текущей части массива.

3) Перевернуть массив так, чтобы максимальный элемент переместился на верхнюю позицию (максимальный элемент становится первым).

4) Перевернуть массив снова, чтобы максимальный элемент оказался в правильном положении (он становится последним элементом).

5) Уменьшить размер текущей части массива, и перейти к следующему наибольшему элементу, повторяя шаги 2-4, пока весь массив не будет упорядочен.

Блинная сортировка обычно рассматривается как не самый эффективный алгоритм сортировки из-за своей высокой временной сложности в худшем случае $O(N^3)$, но он обладает простой и наглядной логикой. Этот алгоритм иногда используется для обучения и демонстрации основных понятий сортировки и манипуляции с элементами массива.

1.2 Поразрядная сортировка

Поразрядная сортировка (англ. Radix Sort) [2] – это алгоритм сортировки, который основывается на разрядах (цифрах) чисел. Он работает для целых чисел или других данных, которые можно разделить на разряды. Алгоритм выполняет сортировку путем поочередного рассмотрения чисел по разрядам, начиная с самого младшего разряда и двигаясь к старшему разряду. Внутри каждого прохода по разрядам используется какой-либо стабильный алгоритм сортировки, обычно сортировка подсчетом (counting sort) или сортировка вставками (insertion sort).

Ниже представлены основные шаги поразрядной сортировки.

1) Определить максимальное количество разрядов в числах, которые нужно отсортировать.

2) Начать с самого младшего разряда и перейти к старшим разрядам поочередно.

3) Для каждого разряда (начиная с младшего) выполнить сортировку элементов с учетом этого разряда, используя стабильный сортировочный алгоритм. Например, можно применить сортировку подсчетом или сортировку вставками для элементов в текущем разряде.

4) Повторять шаг 3 для каждого разряда, двигаясь от младшего к старшему.

5) После обработки всех разрядов массив будет отсортирован.

Поразрядная сортировка может быть эффективной, особенно когда числа имеют ограниченное количество разрядов, и все разряды равнозначны. Однако она может быть не так эффективной, если числа имеют разное количество разрядов, и в худшем случае она имеет временную сложность $O(n * k)$, где n - количество элементов, а k - количество разрядов.

Этот алгоритм часто используется для сортировки целых чисел, строк и других данных, которые можно представить как последовательность разрядов.

1.3 Сортировка бинарным деревом

Сортировка бинарным деревом [3], также известная как сортировка двоичным деревом поиска (Binary Search Tree Sort), это алгоритм сортировки, который использует бинарное дерево поиска для упорядочивания элементов. Этот алгоритм начинает с пустого бинарного дерева [3] и поочередно вставляет элементы в дерево так, чтобы сохранить порядок сортировки. Затем, когда все элементы вставлены в дерево, они извлекаются в упорядоченной последовательности с помощью обхода в порядке возрастания (in-order traversal) бинарного дерева.

Ниже представлены основные шаги сортировки бинарным деревом.

1) Начать с пустого бинарного дерева.

2) Последовательно вставить все элементы из исходного массива в бинарное дерево. При вставке элемента следует учитывать его значение и соблюдать порядок сортировки. Это означает, что элементы с меньшими значениями должны быть помещены в левое поддерево, а элементы с большими значениями - в правое поддерево.

3) После вставки всех элементов в бинарное дерево выполнить обход в порядке возрастания (in-order traversal) этого дерева. Обход возвращает элементы в упорядоченной последовательности.

4) Элементы, извлеченные в результате обхода, образуют упорядоченный массив, который представляет собой отсортированную версию исходного массива.

Сортировка бинарным деревом имеет среднюю временную сложность $O(n*log(n))$, где n - количество элементов, однако в худшем случае (если элементы добавляются в дерево в уже упорядоченном порядке) может иметь временную сложность $O(n^2)$, что делает ее менее эффективной, чем некоторые другие алгоритмы сортировки. Кроме того, сортировка бинарным деревом требует дополнительной памяти для хранения структуры дерева.

Вывод

В данном разделе мы рассмотрели алгоритмы сортировки: блинную, поразрядную, бинарным деревом. Рассмотренные сортировки следует реализовать, изучить трудоемкость и проанализировать время и память для выполнения.

2 Конструкторская часть

2.1 Введение

В данном разделе представлены схемы алгоритмов сортировок: блинной, поразрядной, бинарным деревом, а также их теоретическая оценка трудоемкости.

2.2 Разработка алгоритмов

В данном разделе представлены схемы алгоритмов для блинной сортировки (рисуноки 2.1, 2.2 и 2.3), алгоритма поразрядной сортировки (рисунки 2.4 и 2.5) и алгоритма сортировки бинарным деревом (рисунки 2.6 и 2.7).

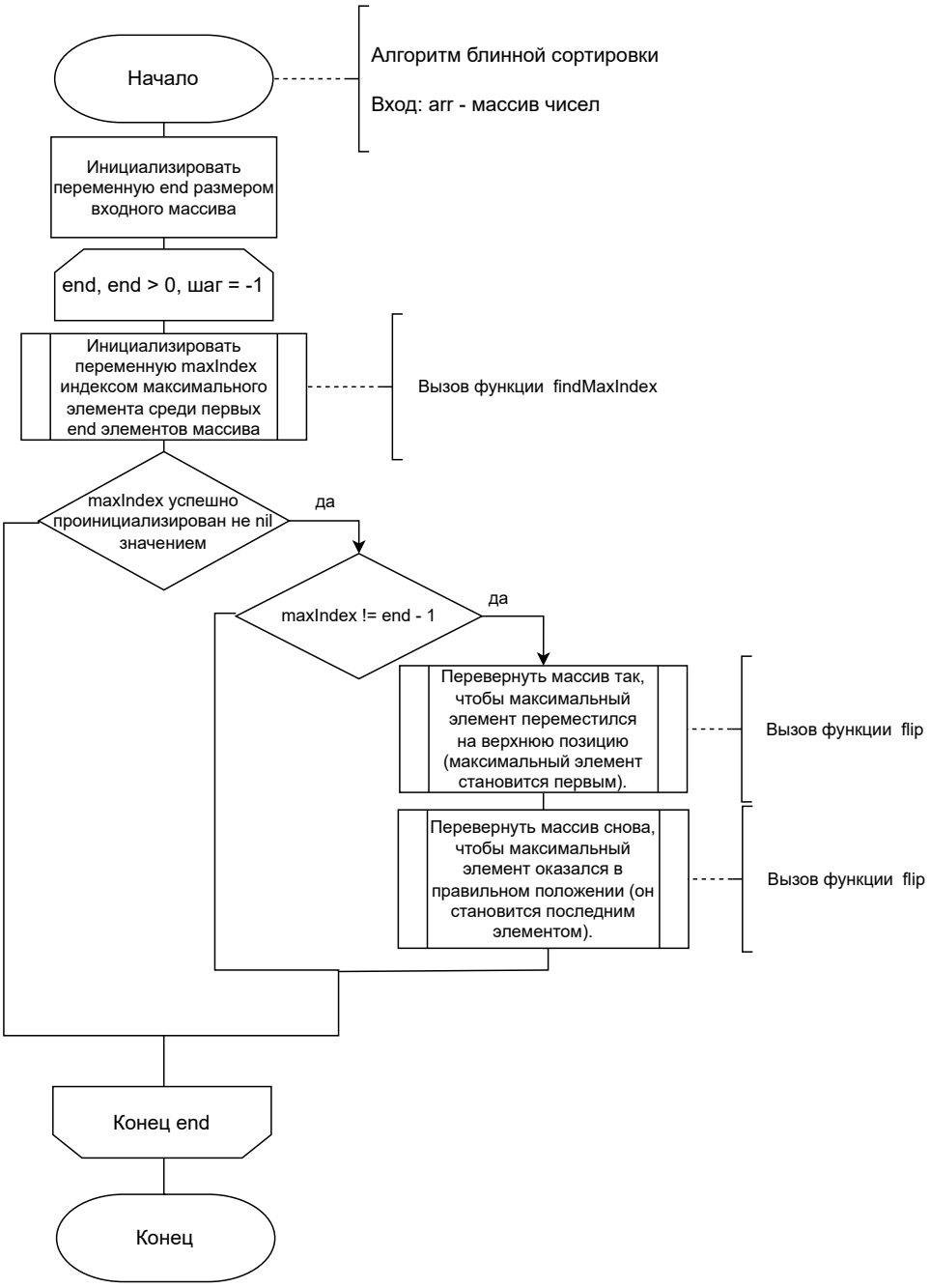

Рисунок 2.1 – Схема блинного алгоритма сортировки

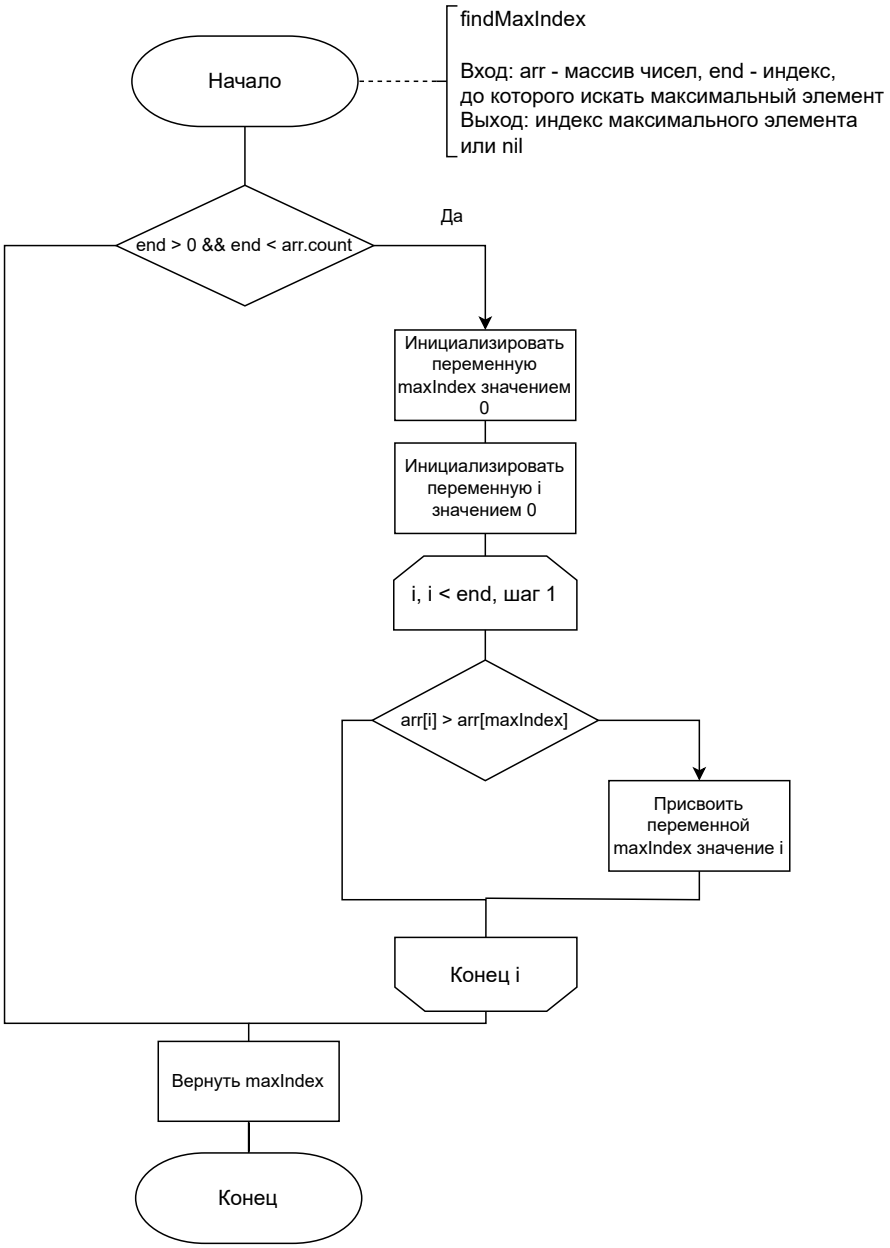

Рисунок 2.2 – Схема алгоритма поиска индекса максимального элемента

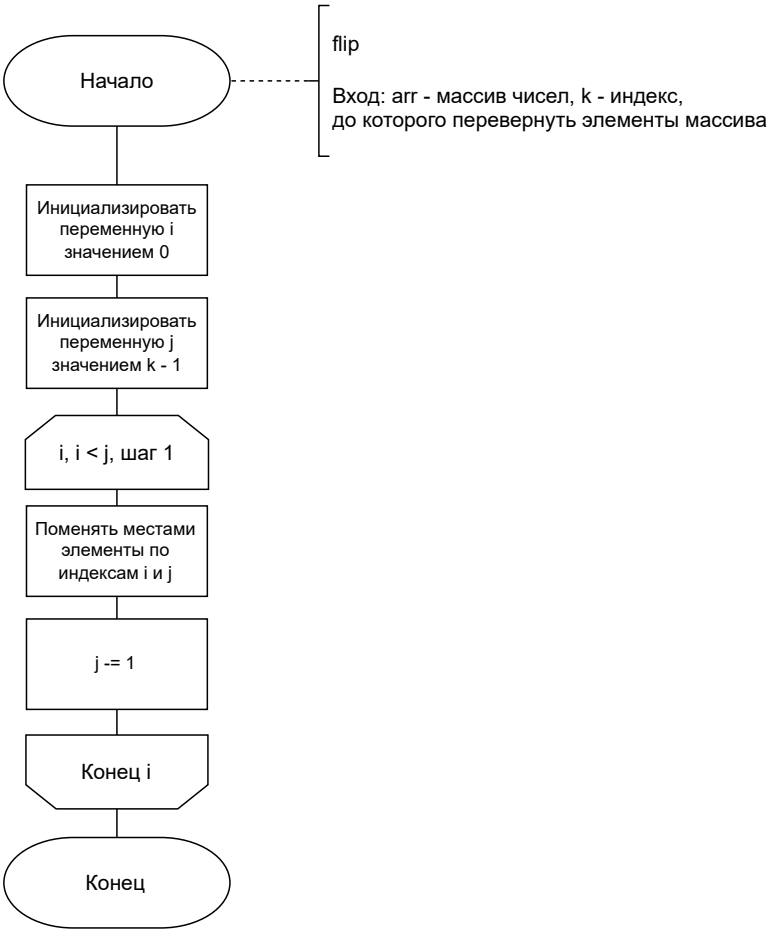

Рисунок 2.3 – Схема алгоритма переворачивания массива

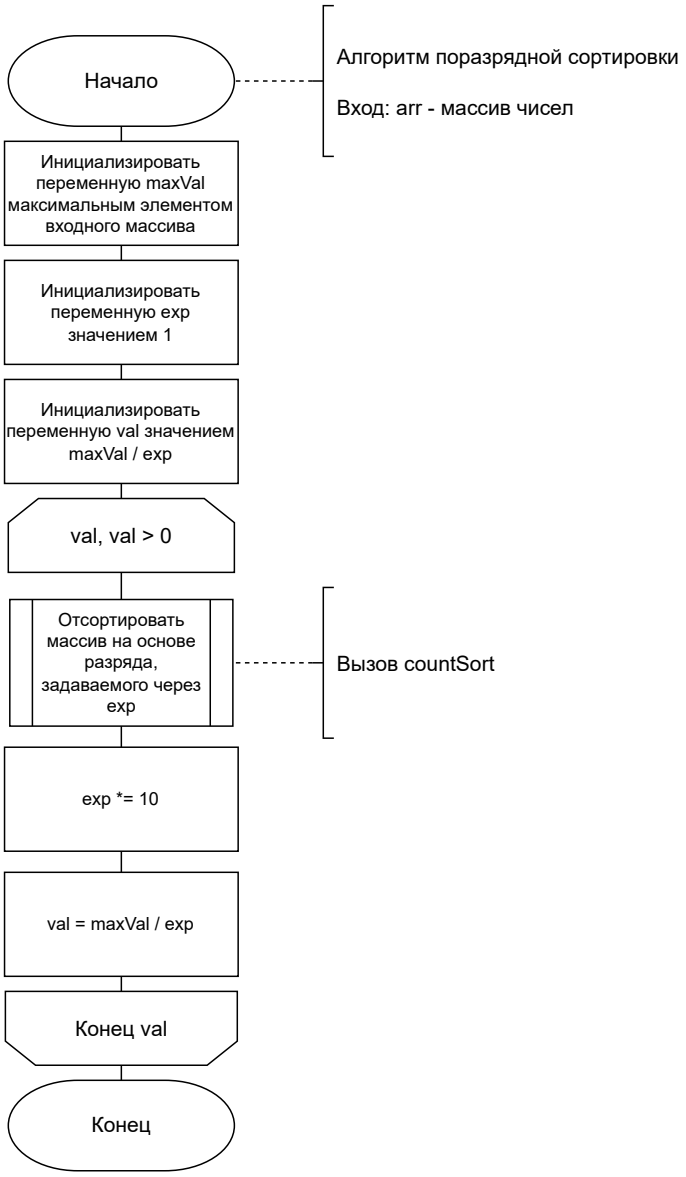

Рисунок 2.4 – Схема поразрядного алгоритма сортировки

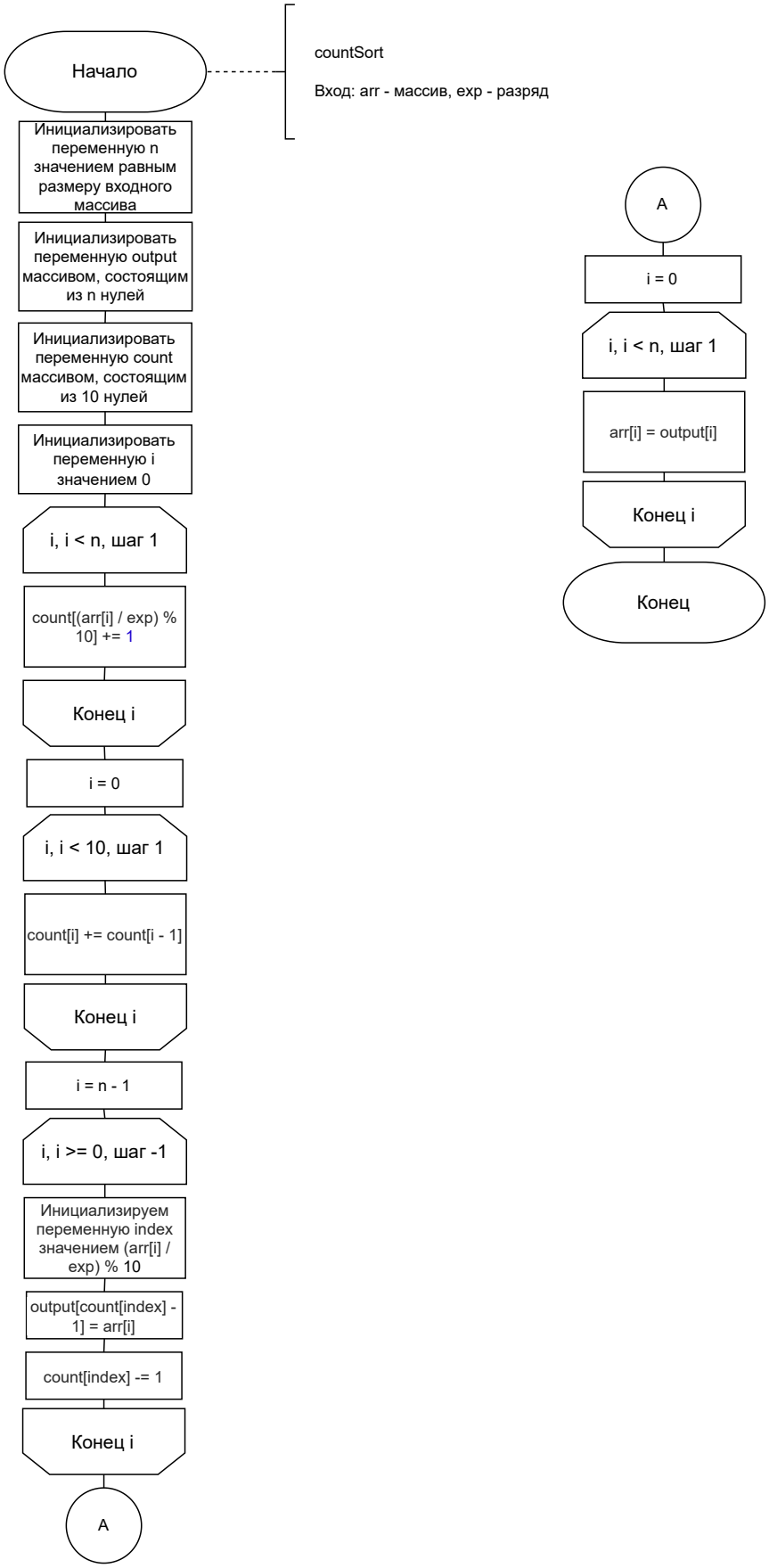

Рисунок 2.5 – Схема алгоритма сортировки подсчетом

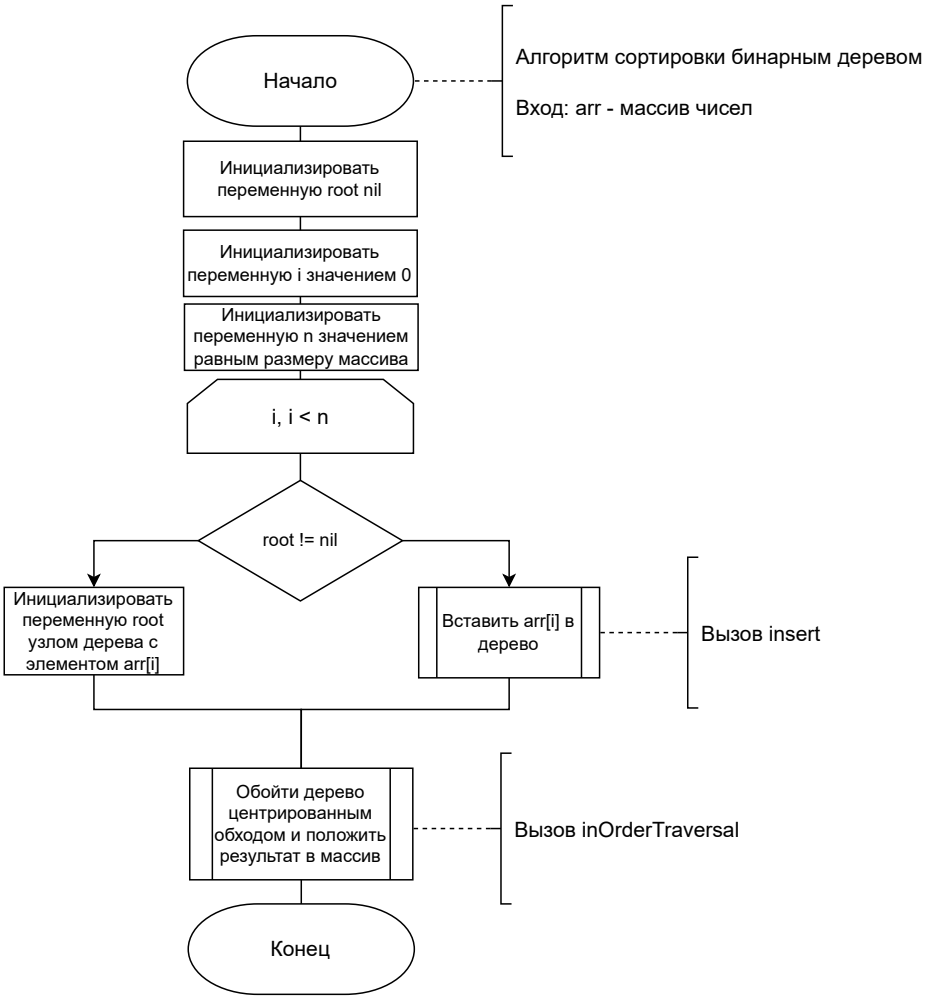

Рисунок 2.6 – Схема алгоритма сортировки бинарным деревом

Рисунок 2.7 – Схема методов работы с деревом

2.3 Модель вычислений для проведения оценки трудоемкости

Введем модель вычислений, которая потребуется для определения трудоемкости каждого отдельного взятого алгоритма умножения матриц.

1) Трудоемкость базовых операций имеет:

   — равную 1:

$$+, -, =, + =, - =, ==, ! =, <, >, <=, >=, [], ++, --, \\ \&\&, >>, <<, ||, \&, | \tag{2.1}$$

   — равную 2:

$$*, /, \%, * =, / =, \% = \tag{2.2}$$

2) Трудоемкость условного оператора:

$$f_{if} = f_{\text{условия}} + \begin{cases} min(f_1, f_2), & \text{лучший случай} \\ max(f_1, f_2), & \text{худший случай} \end{cases} \tag{2.3}$$

3) Трудоемкость цикла:

$$f_{for} = f_{\text{инициализация}} + f_{\text{сравнения}} + M_{\text{итераций}} \cdot (f_{\text{тело}} + \\ + f_{\text{инкремент}} + f_{\text{сравнения}}) \tag{2.4}$$

4) Трудоемкость передачи параметра в функции и возврат из функции равны 0.

2.4 Трудоемкость алгоритмов

Была рассчитана трудоемкость алгоритмов сортировки.

2.4.1 Блинная сортировка

Для алгоритма блинной сортировки трудоемкость будет равна трудоемкости цикла по $end \in [1 \dots N]$.

Трудоемкость цикла указана в формуле (2.5).

$$f_{body} = 2 + N \cdot (f_{findMaxIndex} + \begin{cases} 2, \\ 2 + 2 \cdot f_{flip} \end{cases} + 1) \tag{2.5}$$

Вычислим трудоемкость функции findMaxIndex. Результат представлен в формуле (2.6).

$$f_{findMaxIndex} = 3 + 1 + 2 + end \cdot (2 + \begin{cases} 3, \\ 3 + 1 \end{cases}) \tag{2.6}$$

Трудоемкость функции flip представлена в формуле (2.7).

$$f_{flip} = 1 + 2 + 1 + end \cdot (1 + 3 + 2) = 4 + 6 \cdot end \tag{2.7}$$

Таким образом, трудоемкость блинной сортировки в лучшем случае равна, формула (2.8).

$$f_{best} = 2 + N \cdot 9 + \frac{1 + N}{2} \cdot N \cdot 5 = 2.5N^2 + 11.5N + 2 \approx 2.5N^2 \tag{2.8}$$

Трудоемкость в худшем случае представлена в формуле (2.9).

$$f_{worst} = 2 + N \cdot 17 + \frac{1 + N}{2} \cdot N \cdot (6 + 2 \cdot 6) = 9N^2 + 26N + 2 \approx 9N^2 \tag{2.9}$$

2.4.2 Поразрядная сортировка

Чтобы вычислить трудоемкость алгоритма поразрядной сортировки, нужно учесть следующее:

— функцию получения максимального элемента массива, трудоемкость которой указана в формуле (2.10);

$$f_{getMax} = 2 + 2 + N \cdot (2 + \begin{cases} 1, & \\ 2 & \end{cases}) \tag{2.10}$$

— сортировку подсчетом, которая учитывает разряд, трудоемкость которой указана в формуле (2.11);

$$\begin{gathered} f_{countSort} = 2 + N + 10 + 2 + N \cdot (1 + 2 + 2 + 1 + 1 + 2) + \\ +2 + 9 \cdot 6 + 2 + N \cdot (1 + 1 + 2 + 2 + 3 + 2 + 2 + 2) + \\ +2 + N \cdot 5 = N \cdot (9 + 15 + 5) + 14 + 58 + 2 = 29N + 74 \end{gathered} \tag{2.11}$$

— внешний цикл, в теле которого сортировка подсчетом, трудоемкость которого указана в формуле (2.12);

$$\begin{gathered} f_{while} = 2 + K \cdot (f_{countSort} + 2 + 2) = 2 + K \cdot (29N + 78) = \\ 29KN + 78K + 2 \end{gathered} \tag{2.12}$$

В итоге, трудоемкость поразрядной сортировки равна:

$$f_{radixSort} = 1 + 4 + 4N + 29KN + 78K + 2 = 29KN + 78K + 4N + 7 \tag{2.13}$$

2.4.3 Сортировка бинарным деревом

Трудоёмкость данного алгоритма посчитаем следующим образом: сортировка – преобразование массива или списка в бинарное дерево поиска посредством операции вставки нового элемента в бинарное дерево. Операция

вставки в бинарное дерево имеет сложность $\log_2(size)$, где $size$ - количество элементов в дереве. Для преобразования массива или списка размером $size$ потребуется использовать операцию вставки в биннарное дерево $size$ раз, таким образом, итоговая трудоёмкость данной сортировки будет равна (2.14):

$$f_{radix} = size \cdot \log_2(size) \tag{2.14}$$

Вывод

В данном разделе были представлены схемы алгоритмов сортировки и проведена теоретическая оценка трудоемкости алгоритмов. Результаты показывают, что поразрядная сортировка наиболее эффективна, а блинная сортировка - наименее эффективна.

3 Технологическая часть

В данной главе представлены требования к программному обеспечению, описаны средства реализации, приведены листинги кода и функциональные тесты.

3.1 Требования к программному обеспечению

Программное обеспечение должно удовлетворять следующим функциональным требованиям: на входе – массив, на выходе – остортированный массив.

Программное обеспечение также должно соответствовать следующим требованиям:

— наличие пользовательского интерфейса для выбора действий;

— вывод результата сортировки

— предоставление функционала для измерения времени выполнения алгоритмов сортировки.

3.2 Средства реализации

Для разработки данной лабораторной работы был выбран язык программирования Swift [4]. Этот выбор обусловлен возможностью измерения процессорного времени [5] и соответствием с выдвинутыми техническими требованиям.

Измерение времени выполнения алгоритмов производится с использованием функции *clock_gettime()* [5].

3.3 Сведения о модулях программы

Программа разбита на следующие модули:

— `main.swift` — точка входа в программу, где происходит вызов алгоритмов через интерфейс;

— `Algorithms.swift` — содержит реализации алгоритмов сортировки (блинная, поразрядная, бинарным деревом);

— `CPUTimeMeasure.swift` — измеряет время работы алгоритмов с учетом заданного количества повторений;

— `GraphRenderer.swift` — Строит графики для каждого из алгоритмов с учетом заданного количества повторений для каждого алгоритма;

3.4 Реализация алгоритмов

В листингах 3.1 – 3.8 приведены реализации алгоритмов сортировки: блинная, поразрядная, бинарным деревом.

В листинге 3.9 приведена реализации ввода массива.

Листинг 3.1 – Блинная сортировка

```
1     private static func pancakeSort(_ arr: inout [Int]) {
2         for end in stride(from: arr.count, to: 1, by: -1) {
3             if let maxIndex = findMaxIndex(arr, end) {
4                 if maxIndex != end - 1 {
5                     flip(&arr, maxIndex + 1)
6                     flip(&arr, end)
7                 }
8             }
9         }
10    }
```

Листинг 3.2 – Функция поиска индекса максимального элемента

```
1     private static func findMaxIndex(_ arr: [Int], _ end: Int)
        -> Int? {
2         if end <= 0 || end > arr.count {
3             return nil
4         }
5
6         var maxIndex = 0
7         for i in 0..<end {
8             if arr[i] > arr[maxIndex] {
9                 maxIndex = i
10            }
11        }
12
13        return maxIndex
14    }
```

Листинг 3.3 – Функция переворачивания массива

```
1     private static func flip(_ arr: inout [Int], _ k: Int) {
2         var i = 0
3         var j = k - 1
4         while i < j {
5             arr.swapAt(i, j)
6             i += 1
7             j -= 1
8         }
9     }
```

Листинг 3.4 – Поразрядная сортировка

```
1     private static func radixSort(_ arr: inout [Int]) {
2         let maxVal = getMax(arr)
3
4         var exp = 1
5         while maxVal / exp > 0 {
6             countSort(&arr, exp)
7             exp *= 10
8         }
9     }
```

Листинг 3.5 – Функция поиска максимума

```
1     private static func getMax(_ arr: [Int]) -> Int {
2         var maxVal = arr[0]
3         for value in arr {
4             if value > maxVal {
5                 maxVal = value
6             }
7         }
8         return maxVal
9     }
```

Листинг 3.6 – Сортировка подсчетом

```
1     private static func countSort(_ arr: inout [Int], _ exp:
        Int) {
2         let n = arr.count
3         var output = Array(repeating: 0, count: n)
4         var count = Array(repeating: 0, count: 10)
5
6         for i in 0..<n {
7             count[(arr[i] / exp) % 10] += 1
8         }
9
10        for i in 1..<10 {
11            count[i] += count[i - 1]
12        }
13
14        for i in stride(from: n - 1, through: 0, by: -1) {
15            let index = (arr[i] / exp) % 10
16            output[count[index] - 1] = arr[i]
17            count[index] -= 1
18        }
19
20        for i in 0..<n {
21            arr[i] = output[i]
22        }
23    }
```

Листинг 3.7 – Сортировка бинарным деревом

```
1     private static func binaryTreeSort(_ arr: inout [Int]) {
2         var root: TreeNode?
3         for element in arr {
4             if let existingRoot = root {
5                 existingRoot.insert(element)
6             } else {
7                 root = TreeNode(value: element)
8             }
9         }
10
11        var sortedArray = [Int]()
12        root?.inOrderTraversal(&sortedArray)
13
14        arr = sortedArray
15    }
```

Листинг 3.8 – Бинарное дерево

```
fileprivate class TreeNode {
    var value: Int
    var left: TreeNode?
    var right: TreeNode?

    fileprivate init(value: Int) {
        self.value = value
    }

    fileprivate func insert(_ value: Int) {
        if value < self.value {
            if let left = left {
                left.insert(value)
            } else {
                left = TreeNode(value: value)
            }
        } else {
            if let right = right {
                right.insert(value)
            } else {
                right = TreeNode(value: value)
            }
        }
    }

    fileprivate func inOrderTraversal(_ result: inout [Int]) {
        left?.inOrderTraversal(&result)
        result.append(value)
        right?.inOrderTraversal(&result)
    }
}
```

Листинг 3.9 – Функция ввода массива

```
1     private static func inputArray() -> [Int] {
2         var numbers: [Int] = []
3
4         print(Welcome.inputArray)
5
6         if let input = readLine() {
7             let numberStrings = input.split(separator: " ")
8
9             for numberString in numberStrings {
10                if let number = Int(numberString) {
11                    numbers.append(number)
12                } else {
13                    print(Errors.invalidNumberInput)
14                }
15            }
16        }
```

3.5 Функциональные тесты

В таблице приведены функциональные тесты для алгоритмов сортировки. Все тесты были успешно пройдены.

Таблица 3.1 – Тесты алгоритмов сортировки

Входной массив	**Ожидаемый результат**
[1, 2, 3, 4, 5, 6, 7]	[1, 2, 3, 4, 5, 6, 7]
[7, 6, 5, 4, 3, 2, 1]	[1, 2, 3, 4, 5, 6, 7]
[1, 5, 3, 7, 6, 2, 4]	[1, 2, 3, 4, 5, 6, 7]
[1, 1, 1, 1, 5, 5, 1, 2, 2]	[1, 1, 1, 1, 1, 2 ,2, 5, 5]

Вывод

В данной главе были представлены требования к программному обеспечению, описаны средства реализации, приведены листинги кода алгоритмов и функциональные тесты, подтверждающие корректность работы алгоритмов.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени:

— Процессор: Apple M1 Pro [6]

— Оперативная память: 32 ГБайт.

— Операционная система: macOS Ventura 13.5.2. [7]

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

На изображении 4.1 представлена иллюстрация работы разработанного программного продукта. Конкретно, демонстрируются результаты выполнения алгоритмов сортировки.

```
Menu:
1. Sort array
2. Measure CPU time
3. Draw graphs
0. Exit

Choose Menu option: 1
Enter a list of integers (separated by spaces):
4 2 1 5 6 4 2 4
Performing sort:
Pancake sort
[1, 2, 2, 4, 4, 4, 5, 6]
Radix sort
[1, 2, 2, 4, 4, 4, 5, 6]
Binary tree sort
[1, 2, 2, 4, 4, 4, 5, 6]
```

Рисунок 4.1 – Демонстрация работы программы

4.3 Анализ временных характеристик

В данном разделе представлены результаты экспериментов, в которых измерялось время выполнения сортировок. Данные результаты представлены в таблицах 4.1, 4.2 и 4.3.

Таблица 4.1 содержит результаты замеров времени выполнения алгоритмов сортировки для отсортированного входного массива размером в диапазоне от 1 до 901 с шагом 10.

Таблица 4.1 – Результаты замеров времени (массив уже отсортирован)

Размер массива	**Время, мкс**		
	Блинная	**Поразрядная**	**Бинарным деревом**
1.0	0.48	0.36	0.58
101.0	1287.46	183.84	128.58
201.0	4501.62	334.02	457.14
301.0	9992.08	497.56	1010.18
401.0	17600.5	674.88	1788.92
501.0	27532.06	839.58	2843.38
601.0	39279.7	980.44	4186.78
701.0	53423.48	1149.32	5822.16
801.0	69950.26	1336.72	7700.92
901.0	88903.92	1497.94	9837.62

На основе данных из таблицы 4.1 был построен график (см. рисунок 4.2).

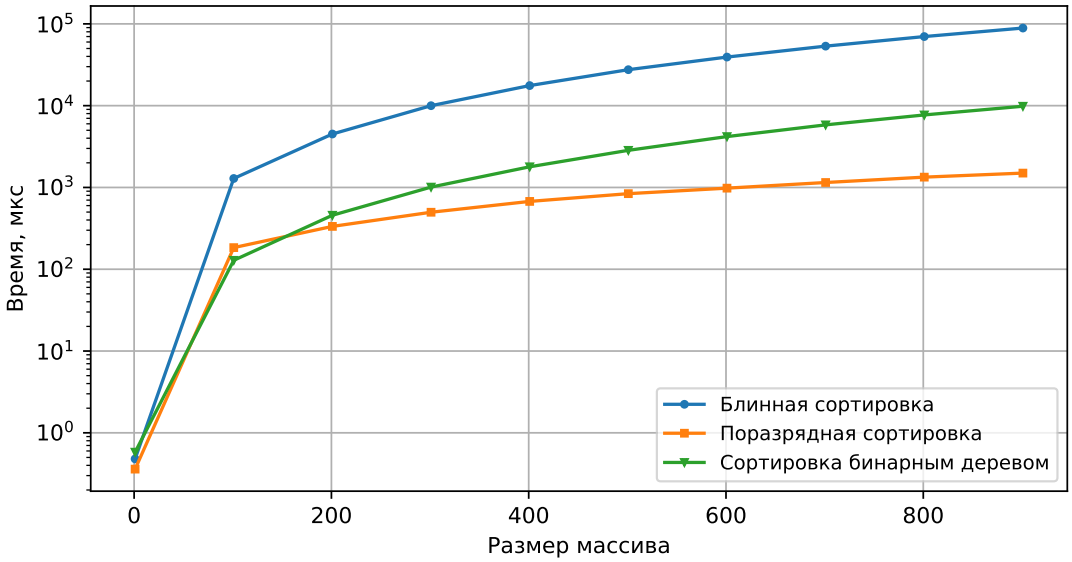

Рисунок 4.2 – Сравнение времени выполнения алгоритмов сортировки для отсортированного массива

Таблица 4.2 содержит результаты замеров времени выполнения алгоритмов сортировки для массивов, отсортированных в обратном порядке.

Таблица 4.2 – Результаты замеров времени (массив отсортирован в обратном порядке)

Размер массива	**Время, мкс**		
	Блинная	**Поразрядная**	**Бинарным деревом**
1.0	0.46	3.84	0.58
101.0	1181.88	175.24	122.38
201.0	4495.76	339.94	456.54
301.0	9928.06	502.8	1007.6
401.0	17643.26	662.86	1781.7
501.0	27468.38	861.06	2877.56
601.0	39573.02	999.8	4169.6
701.0	53783.08	1158.86	5786.46
801.0	70181.04	1346.86	7659.62
901.0	88974.4	1488.44	9791.12

На основе данных из таблицы 4.2 был построен график (см. рисунок 4.3).

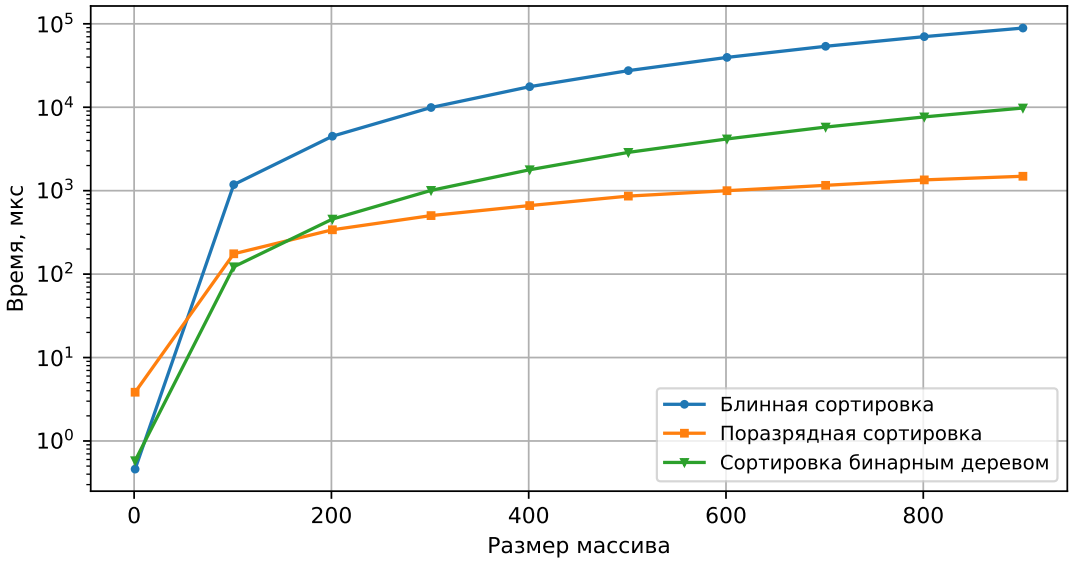

Рисунок 4.3 – Сравнение времени выполнения алгоритмов сортировки для массивов, отсортированных в обратном порядке

Таблица 4.3 содержит результаты замеров времени выполнения алгоритмов сортировки для массивов, содержащих случайный набор данных.

Таблица 4.3 – Результаты замеров времени (случайный набор данных)

Размер массива	**Время, мкс**		
	Блинная	**Поразрядная**	**Бинарным деревом**
1.0	2.44	39.3	2.22
101.0	2236.28	171.0	31.48
201.0	5914.7	433.28	72.94
301.0	13109.72	488.0	102.8
401.0	23333.1	645.26	148.22
501.0	36552.6	812.34	191.26
601.0	52732.48	1280.16	232.36
701.0	71384.34	1484.68	278.68
801.0	92454.4	1291.88	313.58
901.0	116855.18	1439.56	360.94

На основе данных из таблицы 4.3 был построен график (см. рисунок 4.4).

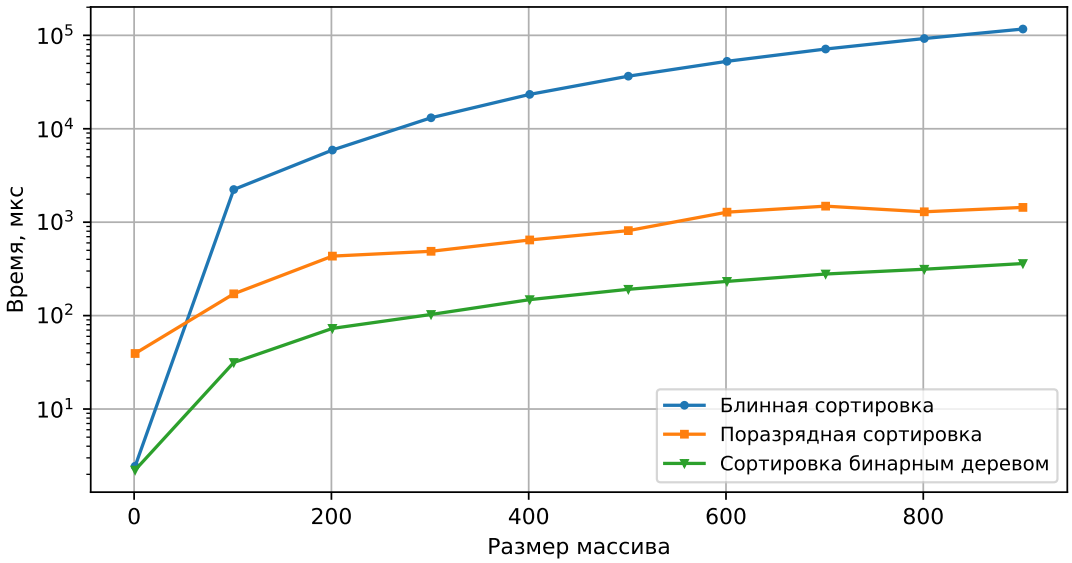

Рисунок 4.4 – Сравнение времени выполнения алгоритмов сортировки для массивов, содержащих случайные наборы данных

4.4 Выводы

Исходя из полученных результатов, сортировка бинарным деревом на отсортированных массивах и блинная сортировка на случайном массиве показывают наименьшую эффективность (на длине массива в 800 элементов примерно в 71 раз дольше, чем поразрядная сортировка), при этом поразрядная сортировка показала себя лучше всех на любых данных. Можно сделать вывод, что использование сортировки бинарным деревом показывает наилучший результат при случайных, никак не отсортированных данных, т.к. при отсортированных данных обычное бинарное дерево вырождается в связный список, из-за чего вырастает высота дерева. Поразрядная сортировка же эффективнее в том случае, когда заранее известно максимальное количество разрядов в сортируемых данных.

Теоретические результаты оценки трудоёмкости и полученные практическим образом результаты замеров совпадают.

Заключение

Результаты данного исследования позволяют сделать несколько важных выводов. Среди всех реализаций алгоритмов наилучшие показатели по времени демонстрирует поразрядная сортировка. Также стоит отметить, что наименее эффективным алгоритмом оказался алгоритм блинной сортировки.

Основной целью данной лабораторной работы было изучение и описание особенностей алгоритмов сортировки, а именно блинной, поразрядной и бинарным деревом. Ниже представлены выполненные задачи.

1) Проведено подробное описание алгоритмов сортировки.

2) Разработано программное обеспечение, реализующее следующие алгоритмы:

   — Блинная сортировка.

   — Поразрядная сортировка.

   — Сортировка бинарным деревом.

3) Выбраны инструменты для измерения процессорного времени выполнения реализаций алгоритмов.

4) Проведен анализ временных затрат программы, чтобы выявить влияющие на них характеристики и факторы.

Цели и задачи исследования были успешно достигнуты, и полученные результаты позволяют лучше понять и оценить эффективность различных алгоритмов сортировки.

Список использованных источников

1 William H. G. Bounds for sorting by prefix reversal. – Department of Electrical Engineering, University of California, BerkeleyNorth-Holland Publishhing Company, 1978. Pp. 47–57.

2 Левитин А. В. Алгоритмы: введение в разработку и анализ. – М.: Издательский дом “Вильямс“, 2006. – 576 с.

3 Акопов Р. Двоичные деревья поиска. – М.: Оптим, RSDN Magazine [Электронный ресурс], 2004. URL: Режим доступа: `http://rsdn.org/article/alg/binstree.xml` (дата обращения: 12.10.2023).

4 Документация к Swift [Электронный ресурс]. — Режим доступа: `https://developer.apple.com/documentation/swift` (дата обращения: 12.09.2023).

5 Измерение процессорного времени [Электронный ресурс]. — Режим доступа: `https://man7.org/linux/man-pages/man3/clock_gettime.3.html` (дата обращения: 14.09.2023).

6 Процессоры M1 [Электронный ресурс]. — Режим доступа: `https://developer.apple.com/documentation/apple-silicon` (дата обращения: 15.09.2023).

7 Операционная система macOS. [Электронный ресурс]. — Режим доступа: `https://developer.apple.com/macos/` (дата обращения: 15.09.2023).