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 дисциплине "Анализ алгоритмов"

Тема Трудоемкость сортировок

Студент Егорова П.А.

Группа ИУ7-54Б

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2022 г.

Оглавление

Введение

Сортировкой называют процесс перегруппировки заданной последовательности объектов в некотором определенном порядке. Определенный порядок (например, упорядочение последовательности целых чисел по возрастанию) в последовательности объектов необходим для удобства работы с этим объектом. Одной из целей сортировки является упрощение дальнейшего поиска элементов в отсортированном множестве. Существует множество различных методов сортировки данных. Однако любой алгоритм сортировки можно разбить на три основные части:

- сравнение, определяющее упорядоченность пары элементов;
- перестановка, меняющая местами пару элементов;
- сортирующий алгоритм, который осуществляет сравнение и перестановку элементов данных до тех пор, пока все эти элементы не будут упорядочены.

Целью работы работы является изучение и реализация сортировки, вычисление трудоёмкости этих алгоритмов. В данной лабораторной работе рассматривается блочная сортировка, сортировка перемешиванием и бинарным деревом.

Для достижения цели ставятся следующие задачи:

- изучить и реализовать 3 алгоритма сортировки: блочная, перемешиванием и бинарным деревом;
- провести сравнительный анализ трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
- провести сравнительный анализ времени работы алгоритмов на основе экспериментальных данных и памяти, используемой ими.

1 Аналитическая часть

1.1 Блочная сортировка

Блочная сортировка [1] – это алгоритм сортировки, который разделяет несортированные элементы массива на несколько групп, называемых блоками или корзинами. Затем каждая корзина сортируется с использованием любого из подходящих алгоритмов сортировки или рекурсивного применения того же алгоритма блочной сортировки: мной был выбран алгоритм сортировки вставками в силу быстроты работы с почти упорядоченными массивами малых размеров.

Мной были найдены реализации данной сортировки только для положительных чисел, поэтому этап поиска интервала – размера блока был усовершенствован, дабы алгоритм был применим и для отрицательных чисел.

В итоге отсортированные сегменты объединяются для формирования окончательного отсортированного массива.

1.2 Сортировка перемешиванием

Сортировка перемешиванием [2] – это разновидность сортировки пузырьком. Отличие в том, что данная сортировка в рамках одной итерации проходит по массиву в обоих направлениях (слева направо и справа налево), тогда как сортировка пузырьком – только в одном направлении (слева направо).

Общие идеи алгоритма:

- обход массива слева направо, аналогично пузырьковой – сравнение соседних элементов, меняя их местами, если левое значение больше правого;
- обход массива в обратном направлении (справа налево), начиная с элемента, который находится перед последним отсортированным, то есть на этом этапе элементы также сравниваются между собой и меняются местами, чтобы наименьшее значение всегда было слева.

1.3 Сортировка бинарным деревом

Из элементов массива формируется бинарное дерево поиска [3]. Первый элемент - корень дерева, остальные добавляются по следующему методу: начиная с корня дерева, элемент сравнивается с узлами. Если элемент меньше чем узел – спускаемся по левой ветке, иначе – по правой. Спустившись до конца, сам элемент становится узлом.

Построенное таким образом дерево можно легко обойти так, чтобы двигаться от узлов с меньшими значениями к узлам с большими значениями. При этом получаем все элементы в возрастающем порядке.

1.4 Вывод

В данном разделе были рассмотрены и описаны три алгоритма сортировок: блочная, перемешиванием, бинарным деревом.

2 Конструкторская часть

2.1 Схемы алгоритмов

На рисунках 2.1 – 2.4 представлены схемы алгоритмов

2.2 Модель вычислений

Для последующего вычисления трудоемкости введём модель вычислений:

1. Операции из списка (2.1) имеют трудоемкость 1.

$$+, -, /, \%, ==, !=, <, >, <=, >=, [], ++, -- \tag{2.1}$$

2. Трудоемкость оператора выбора if условие then A else B рассчитывается, как (2.2).

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \tag{2.2}$$

3. Трудоемкость цикла рассчитывается, как (2.3).

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}) \tag{2.3}$$

4. Трудоемкость вызова функции равна 0.

2.3 Трудоёмкость алгоритмов

2.3.1 Алгоритм блочной сортировки

2.3.2 Алгоритм сортировки перемешиванием

- Трудоёмкость сравнения внешнего цикла $while$, которая равна (2.4):

$$f_{outer} = 1 + 2 \cdot (N - 1) \tag{2.4}$$

- Суммарная трудоёмкость внутренних циклов, количество итераций которых меняется в промежутке $[1..N-1]$, которая равна (2.5):

$$f_{inner} = 5(N - 1) + \frac{2 \cdot (N - 1)}{2} \cdot (3 + f_{if}) \tag{2.5}$$

- Трудоёмкость условия во внутреннем цикле, которая равна (2.6):

$$f_{if} = 4 + \begin{cases} 0, & \text{л.с.} \\ 9, & \text{х.с.} \end{cases} \tag{2.6}$$

Трудоёмкость в лучшем случае (2.7):

$$f_{best} = -3 + \frac{3}{2}N + \approx \frac{3}{2}N = O(N) \tag{2.7}$$

Трудоёмкость в худшем случае (2.8):

$$f_{worst} = -3 - 8N + 8N^2 \approx 8N^2 = O(N^2) \tag{2.8}$$

2.3.3 Алгоритм сортировки бинарным деревом

2.4 Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы всех алгоритмов сортировки массивов, а также описана модель вычислений и, исходя из нее, оценена трудоемкость для каждого алгоритма.

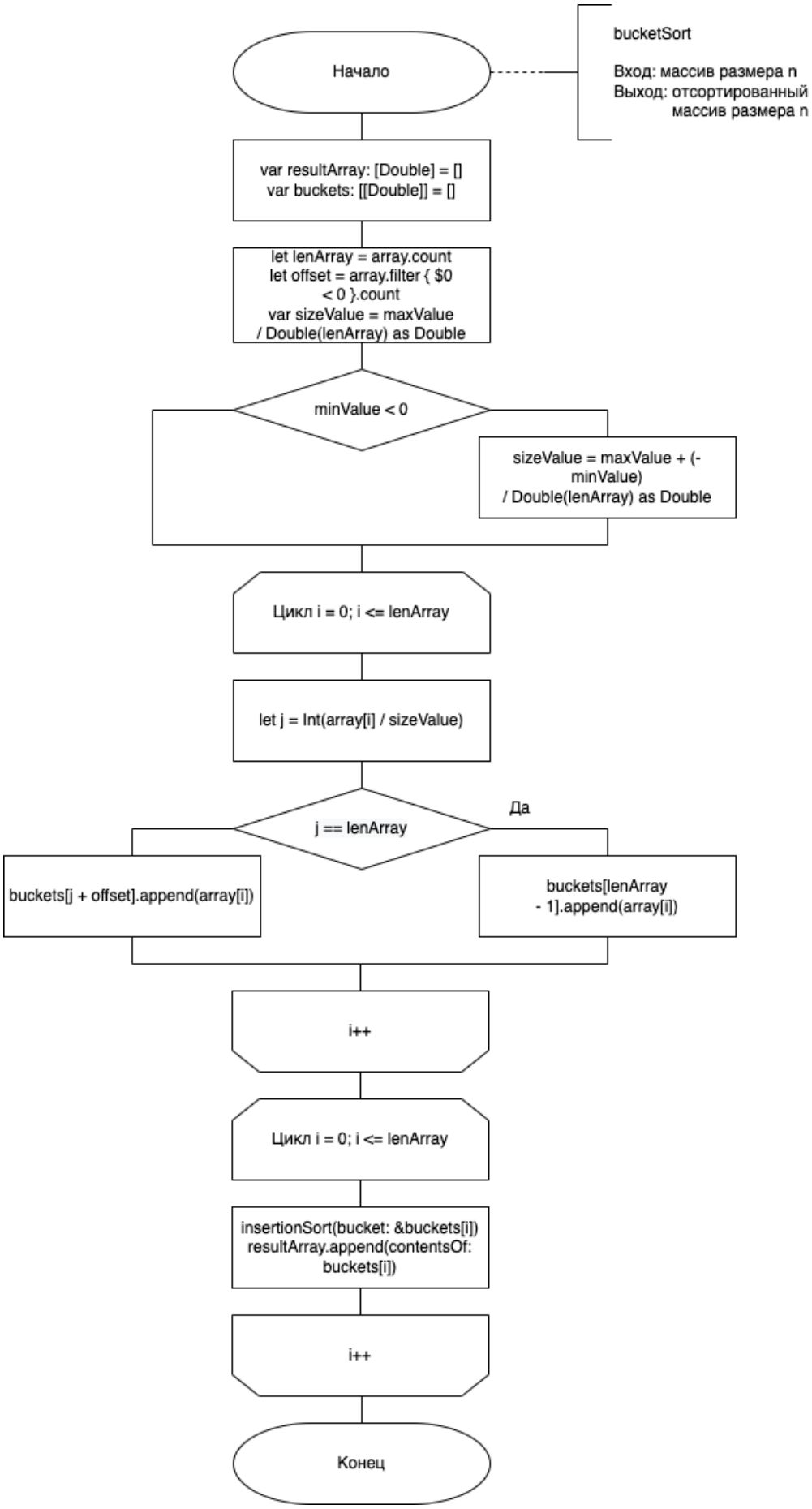

Рис. 2.1: Схема алгоритма блочной сортировки

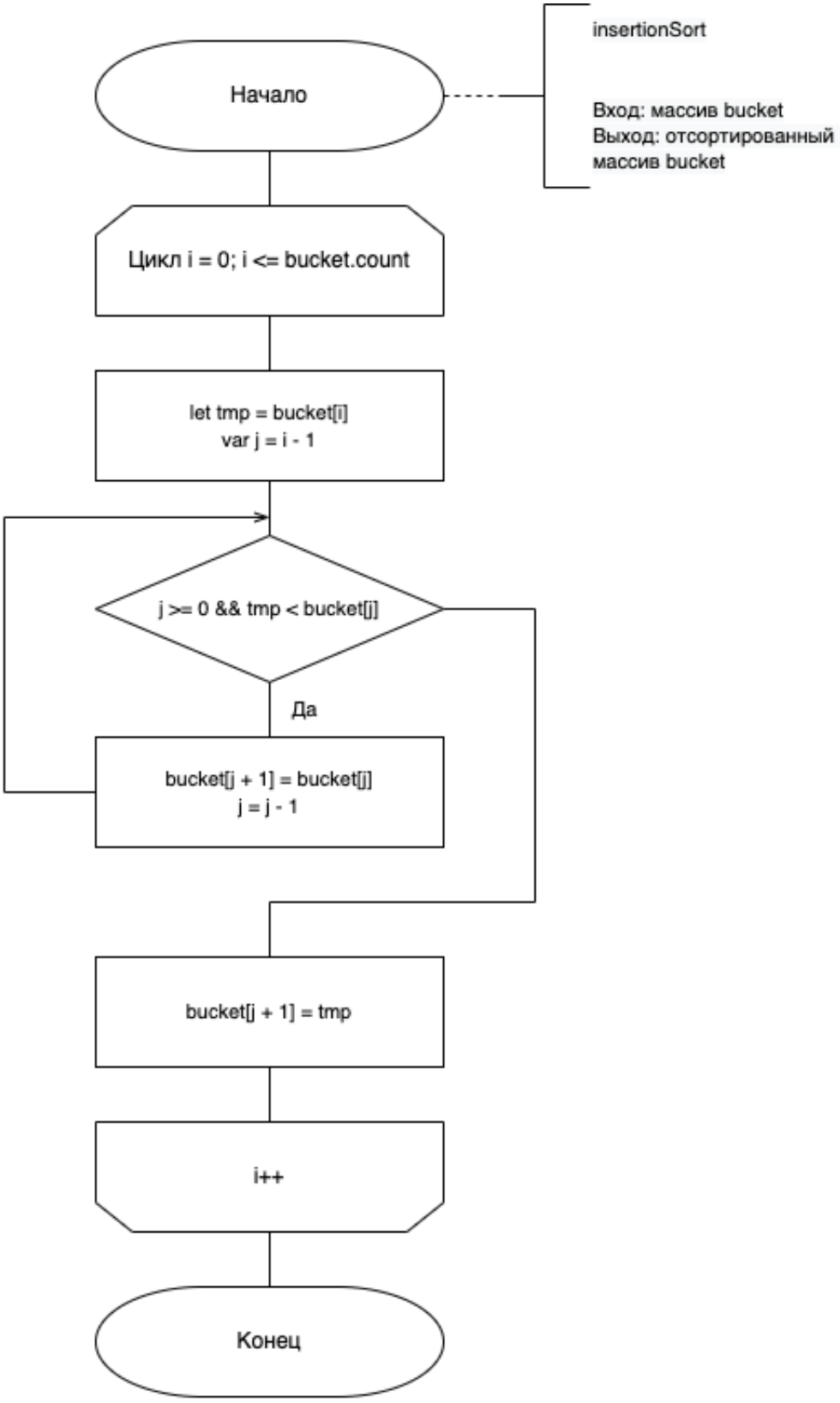

Рис. 2.2: Схема алгоритма сортировки перемешиванием

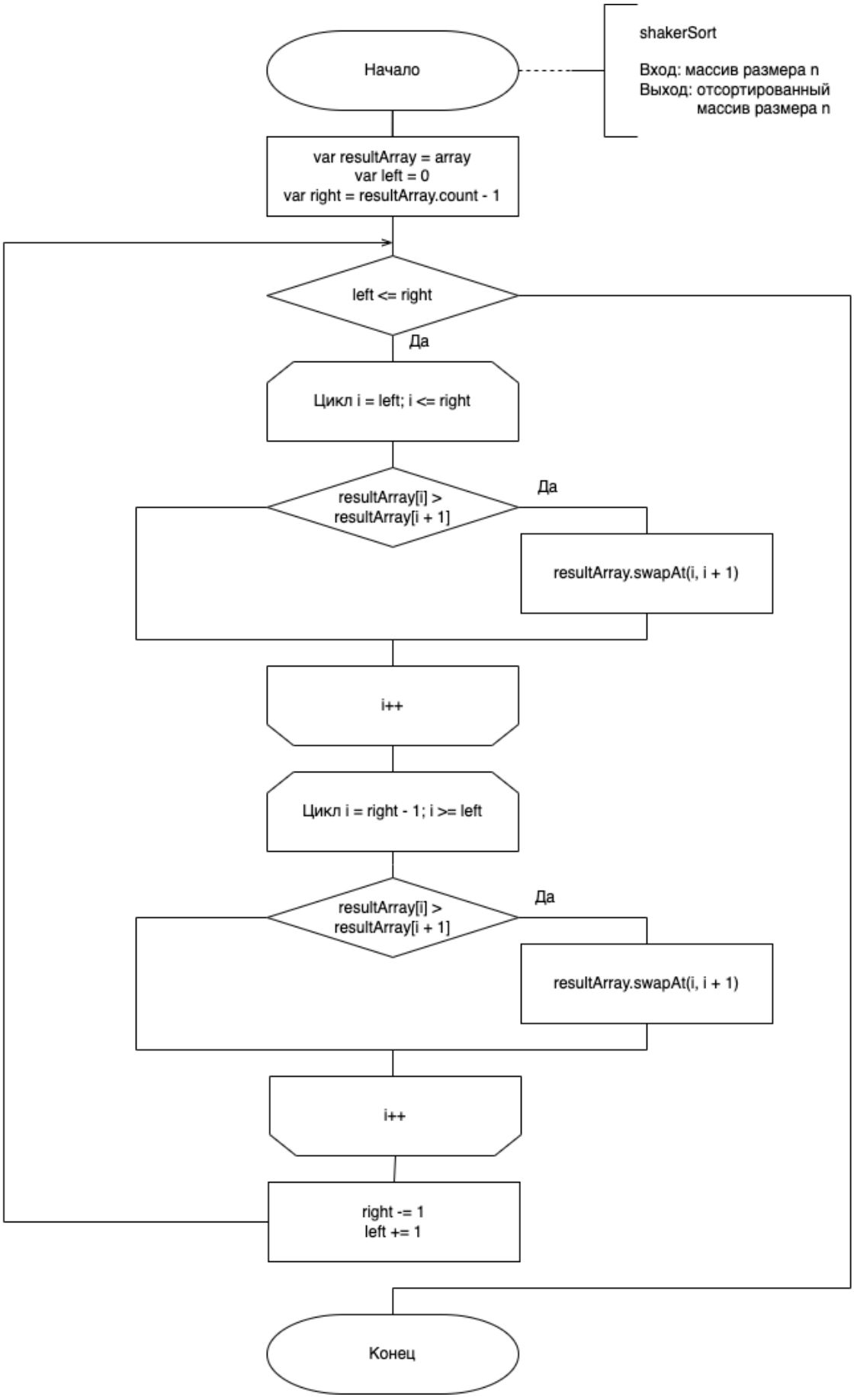

Рис. 2.3: Схема алгоритма сортировки перемешиванием

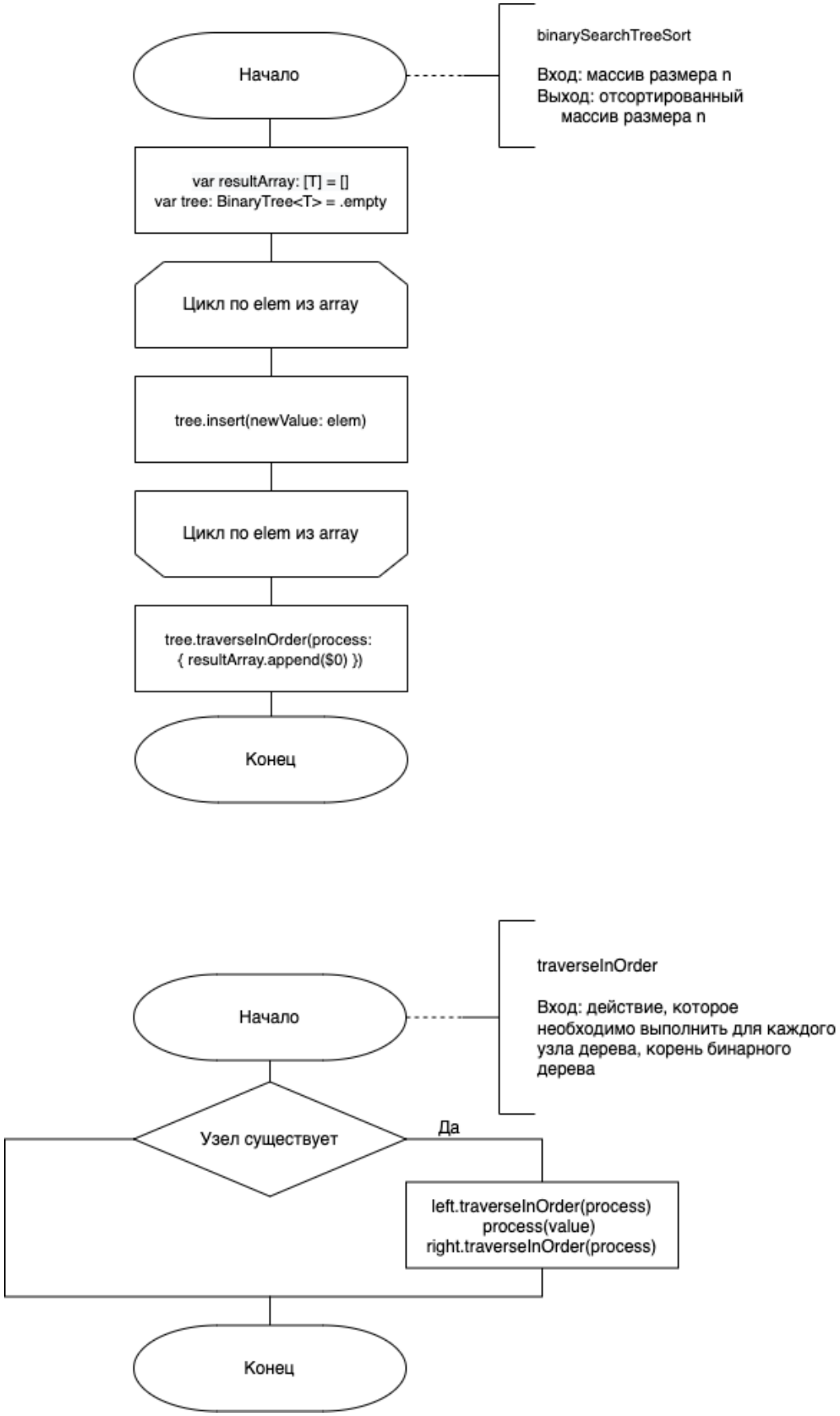

Рис. 2.4: Схема алгоритма сортировки бинарным деревом

3 Технологическая часть

В данном разделе производится выбор средств реализации, а также приводятся требования к программному обеспечению (ПО), листинги реализованных алгоритмов.

3.1 Требования к ПО

На вход программе подается массив сравниваемых элементов, на выходе должен быть получен отсортированный массив. Результирующий массив вычислен с помощью каждого из реализованных алгоритмов сортировки: блочной, перемешиванием, бинарным деревом. Также необходимо вывести затраченное каждым алгоритмом процессорное время и занимаемую память.

3.2 Выбор средств реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык Swift [4]. Данный язык создан компаний Apple для разработки программного обеспечения для macOS, iOS, watchOS и tvOS. Выбор обоснован желанием расширить знания в области применения данного языка, а также возможностью создать на основе написанного кода приложение.

Кроме того, в Swift есть фреймворк CoreFoundation [5], который предоставляет функции для замера процессорного времени. В качестве среды разработки выбран XCode [6].

3.3 Листинги кода

В листингах 3.1 – 3.3 представлены реализации рассматриваемых алгоритмов.

3.4 Листинги кода

В Листинге 3.1 показана реализация алгоритма сортировки бинарным деревом.

```
enum BinaryTree<T: Comparable> {
    case empty
    indirect case node(BinaryTree, T, BinaryTree)

    func newTreeWithInsertedValue(newValue: T) -> BinaryTree {
        switch self {
        case .empty:
            return .node(.empty, newValue, .empty)
        case let .node(left, value, right):
            if newValue < value {
                return .node(left.newTreeWithInsertedValue(newValue:
    newValue), value, right)
            } else { return .node(left, value, right.
    newTreeWithInsertedValue(newValue: newValue)) }
        }
    }
    mutating func insert(newValue: T) {
        self = newTreeWithInsertedValue(newValue: newValue)
    }
    func traverseInOrder(process: (T) -> ()) {
        switch self {
        case .empty:
            return
        case let .node(left, value, right):
            left.traverseInOrder(process: process)
            process(value)
            right.traverseInOrder(process: process)
        }
    }
}

func binarySearchTreeSort<T: Comparable>(_ array: [T]) -> [T] {
    var resultArray: [T] = []
    var tree: BinaryTree<T> = .empty
    for elem in array { tree.insert(newValue: elem) }
    tree.traverseInOrder(process: { resultArray.append($0) })

    return resultArray
}
```

Листинг 3.1: Функция алгоритма сортировки бинарным деревом

В Листинге 3.3 показана реализация реализация алгоритма сортировки перемешиванием.

```
func shakerSort<T: Comparable>(_ array: [T]) -> [T] {
    var resultArray = array
    var left = 0
    var right = resultArray.count - 1
    while left <= right {
        for i in left..<right { if resultArray[i] > resultArray[i + 1] {
            resultArray.swapAt(i, i + 1) } }
        right -= 1
        for i in stride(from: right, to: left, by: -1) { if resultArray[i
    - 1] > resultArray[i] {
            resultArray.swapAt(i, i - 1) } }
        left += 1
    }
    return resultArray
}
```

Листинг 3.2: Функция алгоритма сортировки перемешиванием

В Листинге 3.2 показана реализация оптимизированного алгоритма блочной сортировки.

```
func bucketSort(_ array: [Double]) -> [Double]  {
    var resultArray: [Double] = []
    let maxValue = array.max()!
    let minValue = array.min()!
    let lenArray = array.count
    let offset = array.filter { $0 < 0 }.count
    var sizeValue = maxValue / Double(lenArray) as Double
    if minValue < 0 { sizeValue = maxValue + (-minValue) / Double(lenArray
    ) as Double }
    var buckets: [[Double]] = []
    for _ in 0..<lenArray { buckets.append([]) }
    for i in 0..<lenArray {
        let j = Int(array[i] / sizeValue)
        if j != lenArray {
            buckets[j + offset].append(array[i])
        } else { buckets[lenArray - 1].append(array[i]) }
    }
    for i in 0..<lenArray {
        insertionSort(bucket: &buckets[i])
        resultArray.append(contentsOf: buckets[i])
    }
    return resultArray
}
```

Листинг 3.3: Функция алгоритма блочной сортировки

3.5 Функциональные тесты

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы сортировки. Все тесты пройдены успешно.

Входной массив	Результат	Ожидаемый результат
$[15, 25, 35, 45]$	$[15, 25, 35, 45]$	$[15, 25, 35, 45]$
$[55, 45, 35, 25]$	$[25, 35, 45, 55]$	$[25, 35, 45, 55]$
$[-10, -20, -30, -25]$	$[-30, -25, -20, -10]$	$[-30, -25, -20, -10]$
$[40, -10, -30, 75]$	$[-30, -10, 40, 75]$	$[-30, -10, 40, 75]$
$[100]$	$[100]$	$[100]$
$[-20]$	$[-20]$	$[-20]$
$[1.1, 2.2, 3.3, 4.4]$	$[1.1, 2.2, 3.3, 4.4]$	$[1.1, 2.2, 3.3, 4.4]$
$[1.1, -2.2, 3.3, -4.4]$	$[-4.4, -2.2, 1.1, 3.3]$	$[-4.4, -2.2, 1.1, 3.3]$
$[-1.1, -2.2, -3.3, -4.4]$	$[-4.4, 3.3, -2.2, -1.1]$	$[-4.4, -3.3, -2.2, -1.1]$
$[10, 10]$	$[10, 10]$	$[10, 10]$

Таблица 3.1: Тестирование функций

3.6 Вывод

В данном разделе были разработаны исходные коды трёх алгоритмов сортировки: блочной, перемешиванием и бинарным деревом. Работа реализаций протестирована на различных наборах данных.

4 Исследовательская часть

Технические характеристики устройства, на котором было произведено тестирование разработанного ПО:

- операционная система: macOS Big Sur 11.6.4;
- оперативная память: 8 Гб;
- 2,4 GHz 2-ядерный процессор Intel Core i5;

Во время тестирования ноутбук был включен в сеть питания.

4.1 Время выполнения реализаций алгоритмов

В таблицах 4.1 – 4.3 представлены замеры времени работы для каждого из алгоритмов для отсортированных массивов, массивов из случайных чисел и неотсортированных массивов. Здесь и далее: БС — блочная сортировка, СП – сортировка перемешиванием, СБД – сортировка бинарным деревом. Высчитывалось среднее время в секундах работы алгоритмов при количестве повторений равном пятидесяти.

Таблица 4.1: Результаты замеров времени алгоритмов на отсортированных массивах

Размер	БС	СП	СБД
500	0.0033	0.0944	0.0458
750	0.0052	0.2581	0.1061
1000	0.0066	0.4021	0.2084
1250	0.0098	0.6588	0.3221
1500	0.0109	1.1394	0.4929
1750	0.0116	1.3223	0.5063
2000	0.0135	1.7849	0.8165

На рисунке 4.4 изображены графики зависимости времени работы алгоритмов сортировок от количества элементов массивов.

Таблица 4.2: Результаты замеров времени алгоритмов на массивах из случайных чисел

Размер	БС	СП	СБД
500	0.0069	0.1313	0.0021
750	0.0121	0.3305	0.0038
1000	0.0204	0.5272	0.0048
1250	0.0286	0.8101	0.0061
1500	0.0383	1.1723	0.0072
1750	0.0511	1.7372	0.0082
2000	0.0656	2.5861	0.0106

Таблица 4.3: Результаты замеров времени алгоритмов на отсортированных в обратном порядке массивах

Размер	БС	СП	СБД
500	0.0036	0.2136	0.1041
750	0.0051	0.5488	0.1399
1000	0.0099	0.8875	0.1849
1250	0.0081	1.2612	0.3215
1500	0.0103	1.6534	0.4063
1750	0.0117	2.3841	0.4865
2000	0.0135	2.7928	0.6994

4.2 Используемая память

Пусть дан массив N элементов сравниваемого типа Type. Тогда затраты по памяти для алгоритмов будут следующие:

- блочная сортировка:
  - длина массива, смещение – $2 \cdot MemoryLayout < Int > .size$
  - минимальный и максимальный элементы, размер блока – $3 \cdot MemoryLayout < Type > .size$
  - массив – $N \cdot MemoryLayout < Type > .size$
- сортировка перемешиванием:
  - переменные right, left – $2 \cdot MemoryLayout < Int > .size$

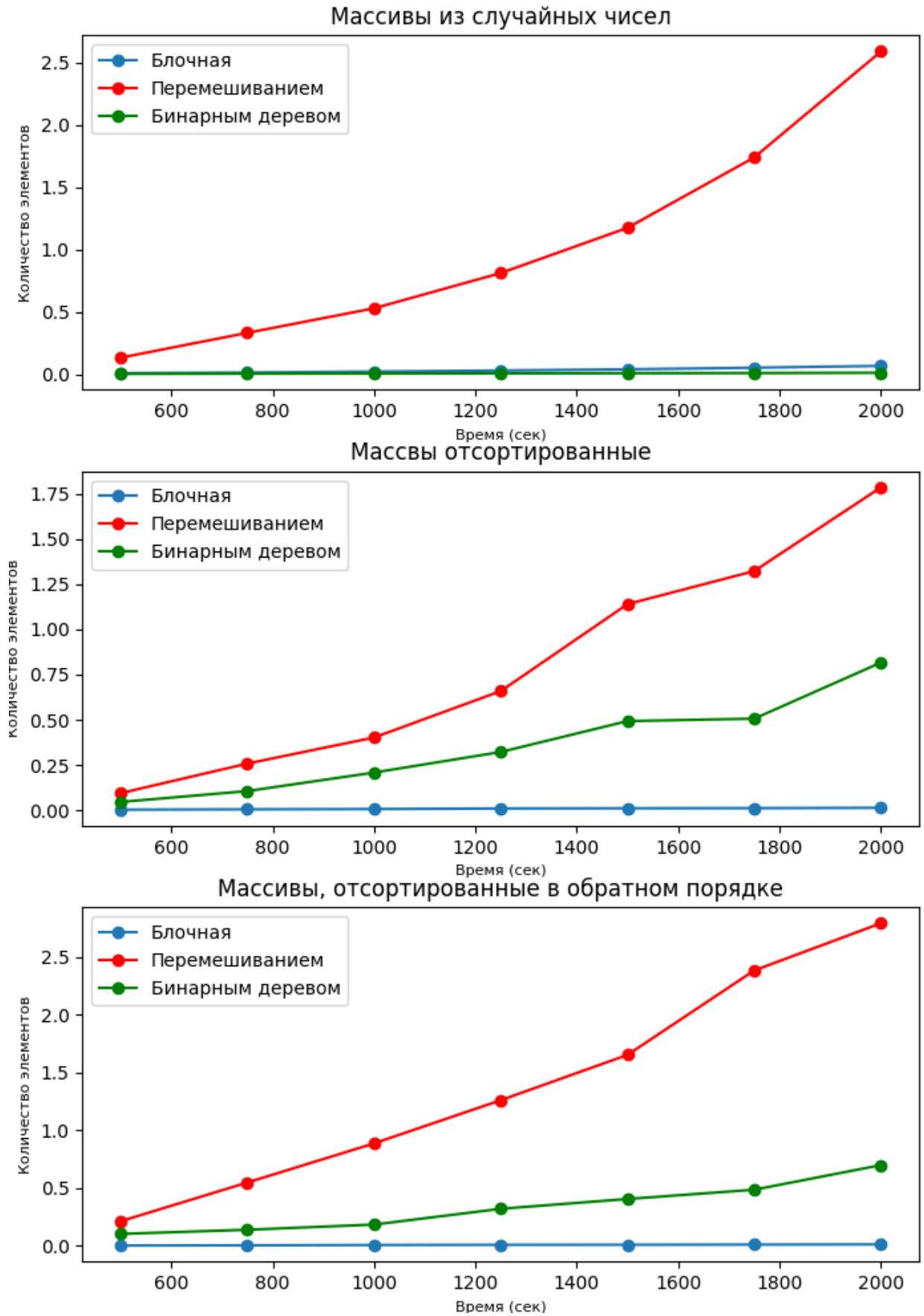

Рис. 4.1: Время работы сортировок

- массив – $N \cdot MemoryLayout < Type > .size$
- переменная-буфер для обмена элементов местами – $1 \cdot MemoryLayout < Type > .size$

- сортировка бинарным деревом:
  - массив – $N \cdot MemoryLayout < Type > .size$
  - бинарное дерево – $N \cdot (3 \cdot MemoryLayout < Type > .size)$

  Нетрудно заметить, что самой затратной по памяти является сортировка бинарным деревом из-за необходимости хранить не только массив и несколько вспомогательных переменных, как в двух других реализациях сортировок, но и структуру бинарного дерева. Размеры же памяти, занимаемой сортировкой перемешиванием и блочной, разнятся не сильно.

Вывод

Экспериментально получено, что при любом наборе входных данных самой медленной оказывается сортировка перемешиванием, а самой независимой к характеристикам входных данных (отсортированный массив, неотсортированный массив, массив случайных чисел) является блочная сортировка. Использование реализации с бинарным деревом эффективно с массивом случайных чисел, однако ее показатели не разнятся при работе с отсортированными массивами и массивами, отсортированными в обратном порядке. В среднем для работы с массивами, состоящими из неупорядоченных данных, время работы бинарной и блочной сортировок практически не отличается и растет как линейная функция, а то время как сортировка перемешиванием уступает им в 1.5 раза и изменяется как функция квадратичная. Самой же неэффективной по памяти является сортировка бинарным деревом из-за необходимости хранить структуру бинарного дерева.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- были изучены и реализованы 3 алгоритма сортировки массивов: блочный, перемешиванием, бинарным деревом;
- был произведен анализ трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
- оценена эффективность алгоритмов по времени на основе экспериментальных данных: выявлено, что сортировка перемешиванием проигрывает по памяти конкурентным реализациям при любом наборе входных данных: отсортированном массива, неотсортированном и массиве случайных чисел. Самым быстрым оказался блочный алгоритм, хоть данная реализация и незначительно медленнее сортировки бинарным деревом.
- оценена эффективность алгоритмов по объему занимаемой памяти: выявлено, что сортировка бинарным деревом проигрывает по этому параметру двум другим в виду использования дополнительной памяти для хранения структуры бинарного дерева. Сортировка перемешиванием и блочная не имеют больших различий по рассматриваемому параметру.

Литература

[1] Блочная сортировка [Электронный ресурс]. Режим доступа: https://www.programiz.com/dsa/bucket-sort

[2] Сортировка перемешиванием [Электронный ресурс]. Режим доступа: https://alley-science.ru/domains_data/files/january-2018/ANALIZ%20ShEYKER-SORTIROVKI%20V%20MASSIVAH.pdf

[3] Сортировка бинарным деревом [Электронный ресурс]. Режим доступа: https://www.raywenderlich.com/990-swift-algorithm-club-swift-binary-search-tree-data-structure

[4] Swift: Swift | Мультипарадигмальный компилируемый язык [Электронный ресурс]. Режим доступа: https://www.apple.com/swift/

[5] Core Foundation: Core Foundation | Time Utilities [Электронный ресурс]. Режим доступа: https://developer.apple.com/documentation/corefoundation/time_utilities

[6] Xcode: Xcode [Электронный ресурс]. Режим доступа: https://developer.apple.com/xcode/

[7] AppCode: AppCode | Smart Swift/Objective-C IDE for iOS [Электронный ресурс]. Режим доступа: https://www.jetbrains.com/objc/