**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** Алгоритмы сортировки

**Студент** Алькина А.Р.

**Группа** ИУ7-54Б

**Оценка (баллы)** _____

**Преподаватели** Волкова Л. Л., Строганов Д. В.

Москва — 2023 г.

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

**Алгоритм сортировки** — это алгоритм упорядочивания элементов в коллекции. Поле, служащее критерием порядка, называется **ключом сортировки**.

Обычно алгоритмы сортировки классифицируют по времени работы, количеству требуемой дополнительной памяти, устойчивости, количеству обменов и детерминированности.

Алгоритмы сортировки являются фундаментальными инструментами компьютерной науки, которые позволяют упорядочить набор данных в определенной последовательности. Они играют важную роль в решении различных задач, от поиска элементов до оптимизации работы с данными.

Цель лабораторной работы — изучение, реализация и сравнение алгоритмов сортировки. В данной лабораторной работе рассматриваются блосная сортировка, порязрядная сортировка и сортировка слиянием.

Задачи лабораторной работы:

— исследовать алгоритмы сортировки;

— применить методы динамического программирования для реализации алгоритмов сортировки;

— оценить и сравнить трудоёмкость алгоритмов сортировки;

— провести сравнительный анализ по времени и памяти на основе экспериментальных данных;

— подготовить отчет по лабораторной работе.

# 1 Аналитическая часть

## 1.1. Блочная сортировка

Блочная сортировка (корзинная сортировка) — алгоритм сортировки, в котором сортируемые элементы распределяются между конечным числом отдельных блоков (карманов, корзин) так, чтобы все элементы в каждом следующем по порядку блоке были всегда больше (или меньше), чем в предыдущем [1]. Каждый блок затем сортируется отдельно либо рекурсивно тем же методом либо другим. Затем элементы помещаются обратно в массив.

Данный тип сортировки требует знания об устройстве входных данных. В случае их равномерного распределения алгоритм блочной сортировки может обладать линейным временем исполнения.

## 1.2. Поразрядная сортировка

Суть алгоритма поразрядной сортировки заключается в том, что массив несколько раз перебирается по разрядам и элементы перегруппировываются в зависимости от того, какая цифра находится в определённом разряде [2]. При этом разряды могут обрабатываться в противоположных направлениях — от младших к старшим или наоборот.

После каждой сортировки по текущему разряду массив становится отсортирован по этому разряду. Для определения количества итераций необходимо предвычислять максимальное количество разрядов среди элементов массива.

## 1.3. Сортировка слиянием

Метод сортировки слияниями был предложен в 1945 году одним из величайших математиков XX века Джоном фон Нейманом. Алгоритм основан на многократном слиянии уже упорядоченных и рядом расположенных

групп элементов массива [3].

Принцип «разделяй и властвуй» нашёл отражение в данной сортировке: массив разбивается на подмассивы меньшего размера, которые рекурсивно разбиваются на подмассивы. Процесс происходит до тех пор, пока в массиве не останется одного элемента — такой случай тривиален, так как массив из одного элемента уже отсортирован. Затем происходит возврат из рекурсивных вызовов с объединением отсорированных частей массива.

# Вывод

Были рассмотрены алгоритмы алгоритмы сортировки: блочная сортировка, поразрядная сортировка, сортировка слиянием. Данные алгоритмы используют различные подходы к сортировке элементов массива. Блочная и поразрядная сортировки работают только с определённым типом входных данных, сортировка слиянием может работать с большим диапазоном типов входных данных.

# 2 Конструкторская часть

## 2.1. Разработка алгоритма блочной сортировки

На рисунках 2.1 — 2.2 приведена схема алгоритма блочной сортировки.

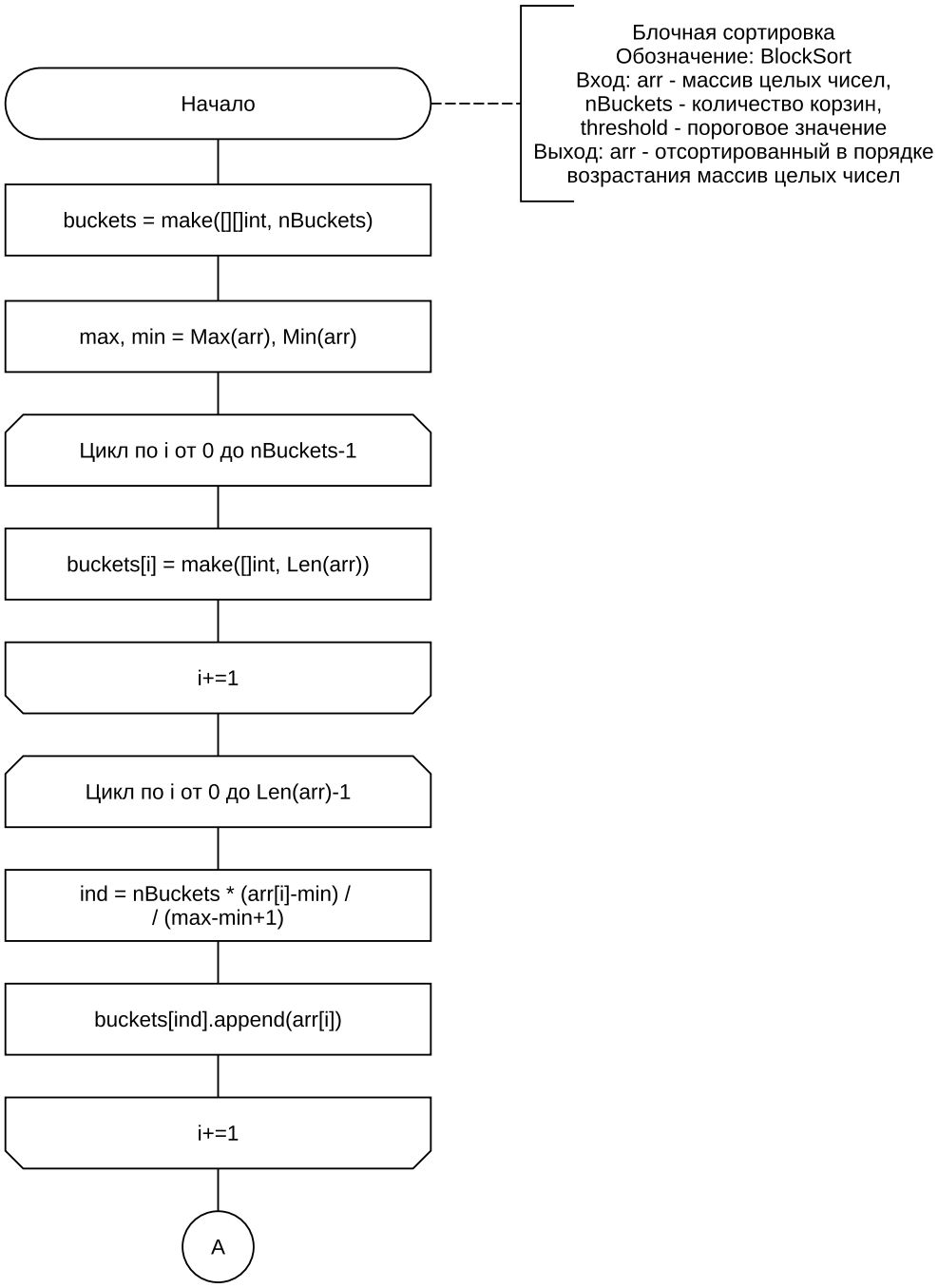


Рисунок 2.1 – Схема алгоритма блочной сортировки: начальная инициализация корзин и распределение чисел по корзинам

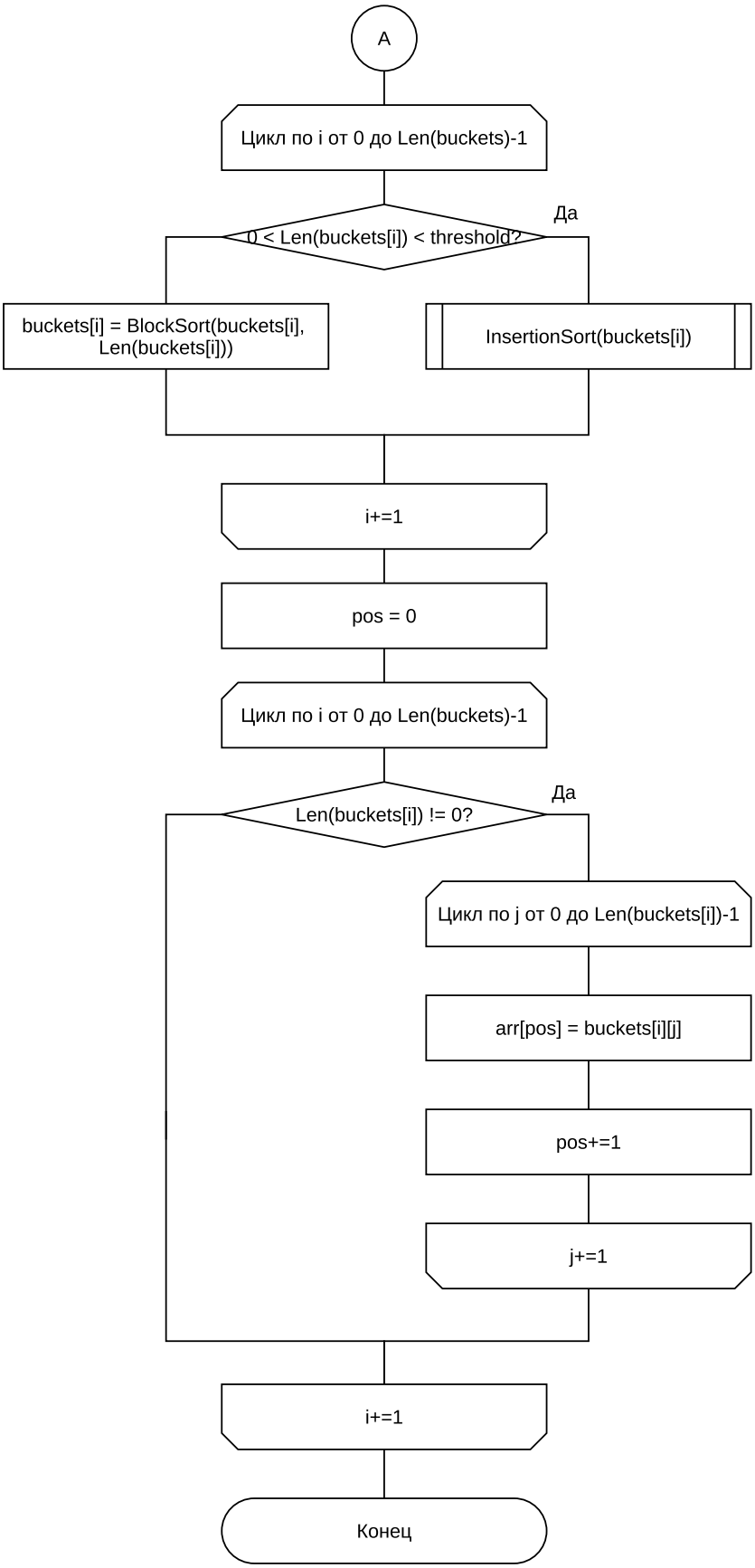


Рисунок 2.2 – Схема алгоритма блочной сортировки: рекурсивная часть и создание результирующего массива

На рисунке 2.3 приведена схема алгоритма сортировки вставками, используемого в алгоритме блочной сортировки.

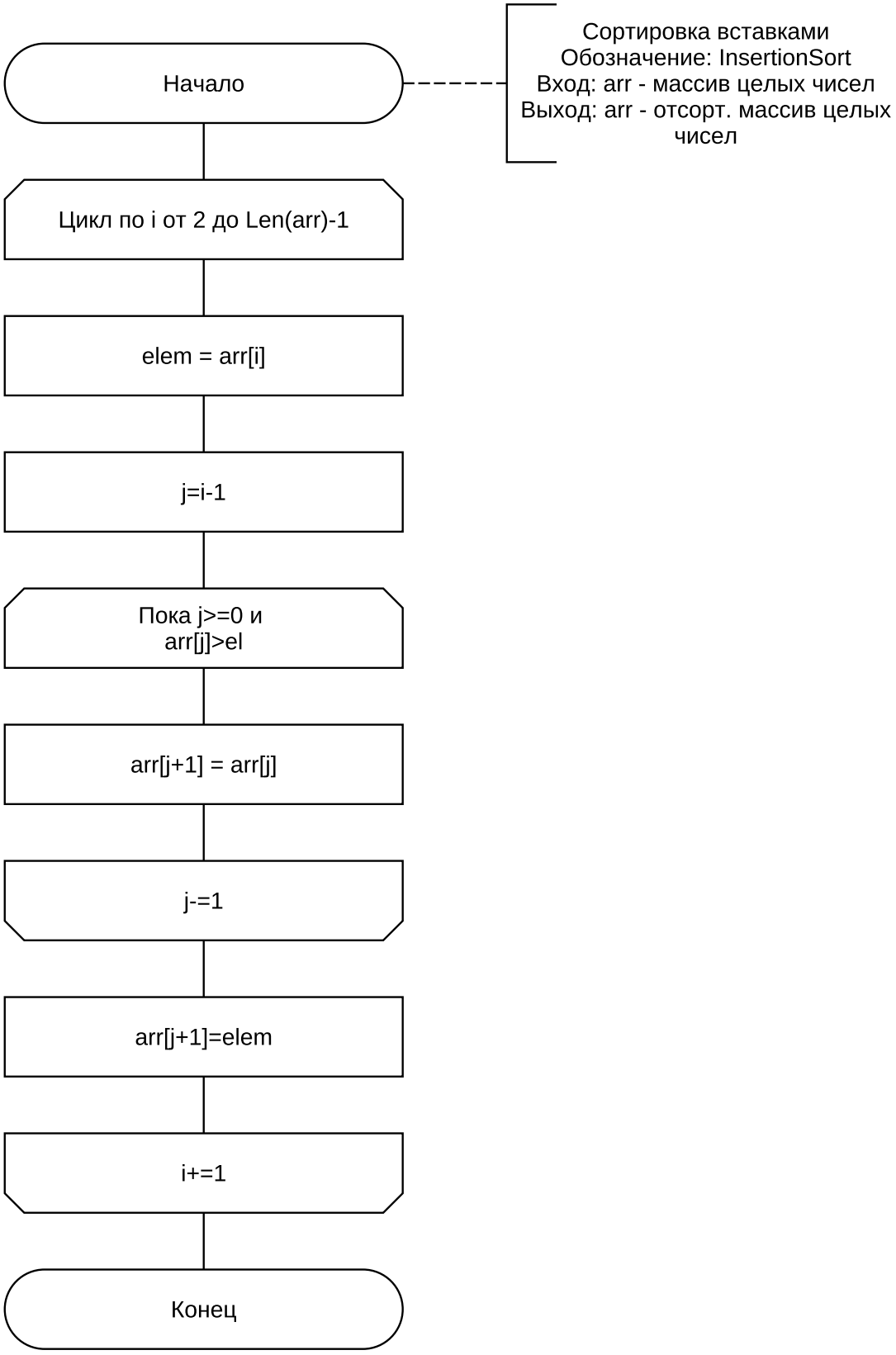


Рисунок 2.3 – Схема алгоритма сортировки вставками

## 2.2. Разработка алгоритма поразрядной сортировки

На рисунках 2.4 — 2.6 приведена схема алгоритма поразрядной сортировки.

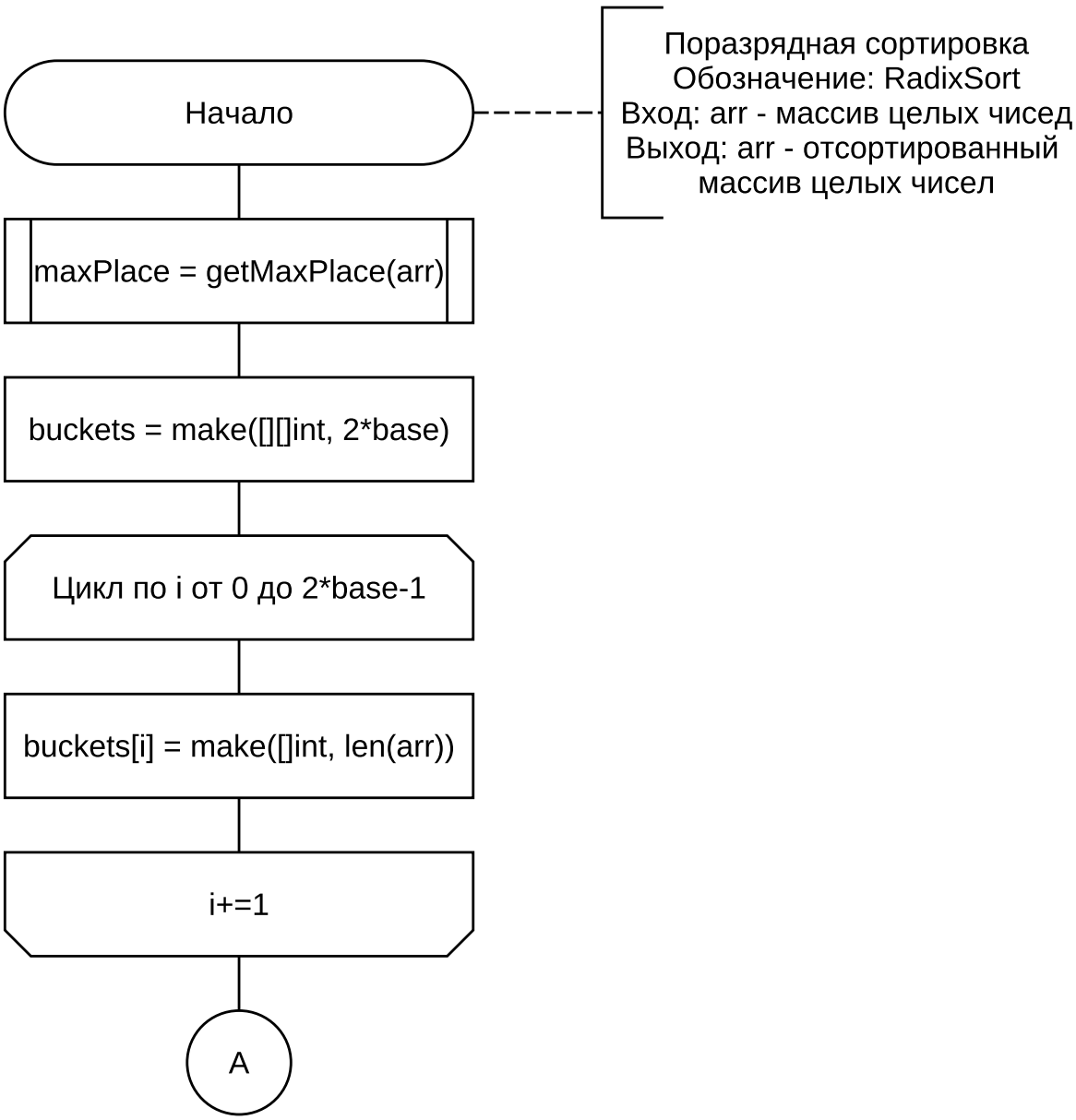


Рисунок 2.4 – Схема алгоритма поразрядной сортировки: начальная инициализация массивов и получение максимального количества разрядов

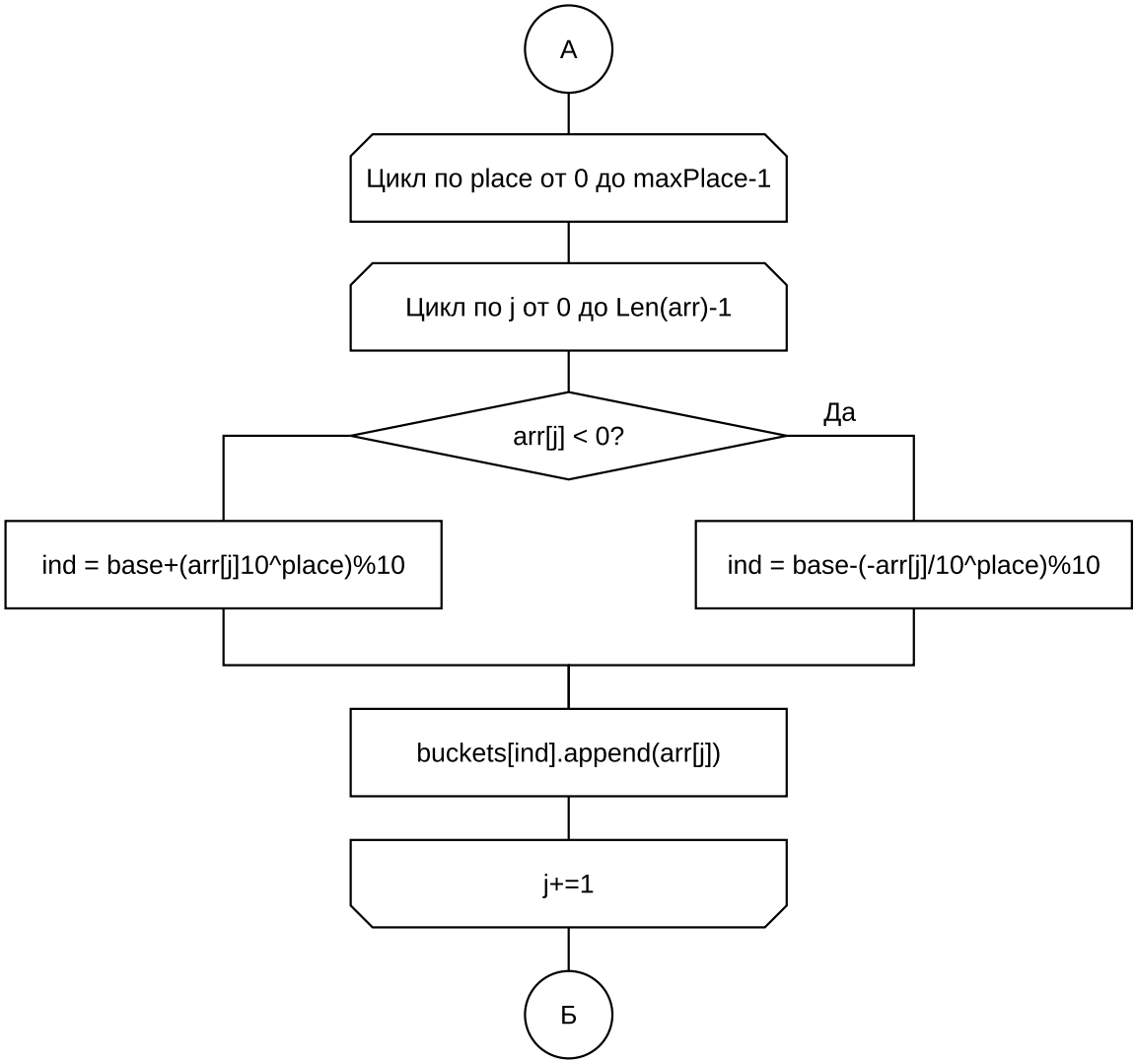


Рисунок 2.5 – Схема алгоритма поразрядной сортировки: цикл по разрядам (распределение чисел по текущему разряду)

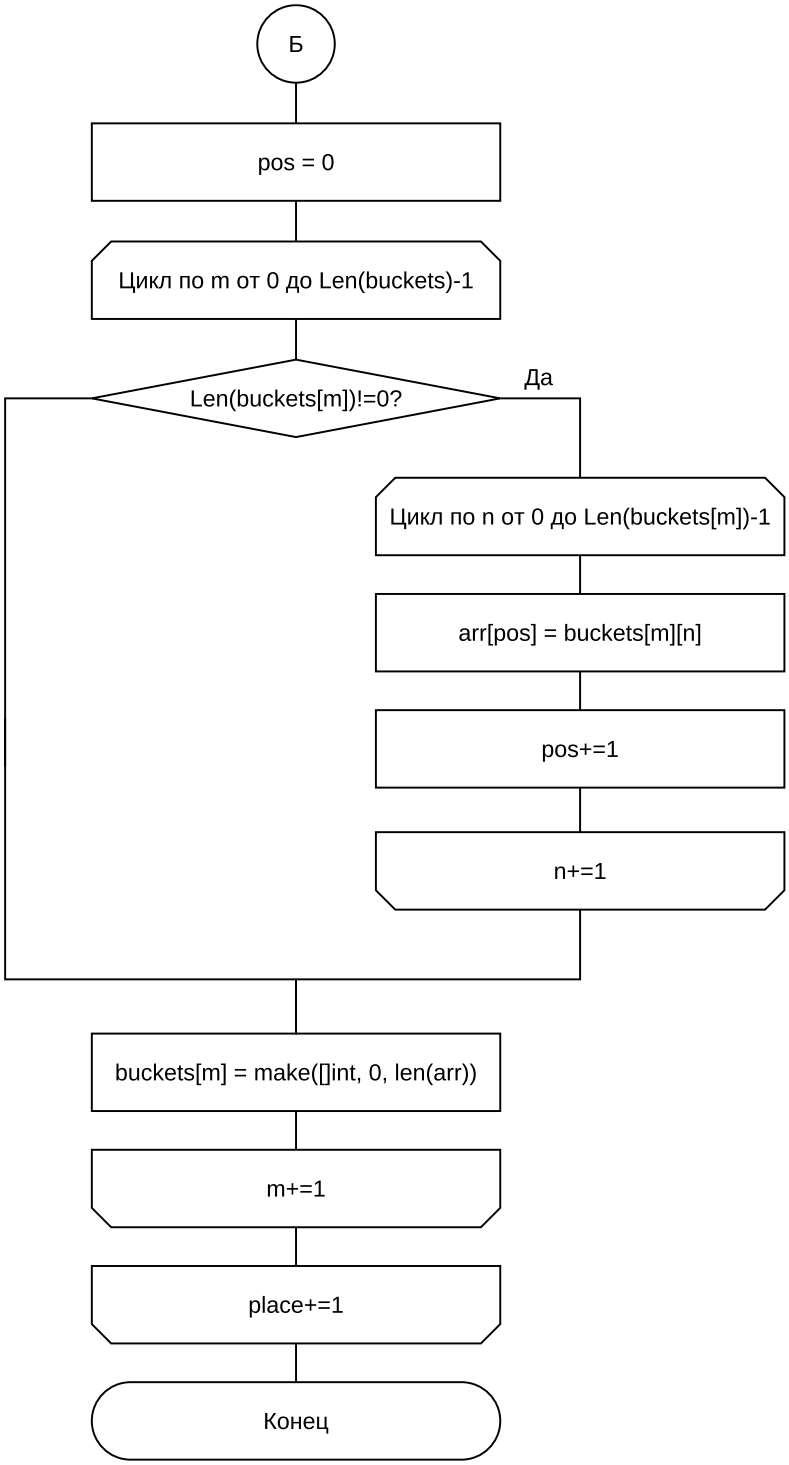


Рисунок 2.6 – Схема алгоритма поразрядной сортировки: цикл по разрядам (объединение отсортированных частей в результирующий массив)

На рисунке 2.7 приведена схема алгоритма получения максимального количества разрядов числа в массиве, используемого в алгоритме поразрядной сортировки.

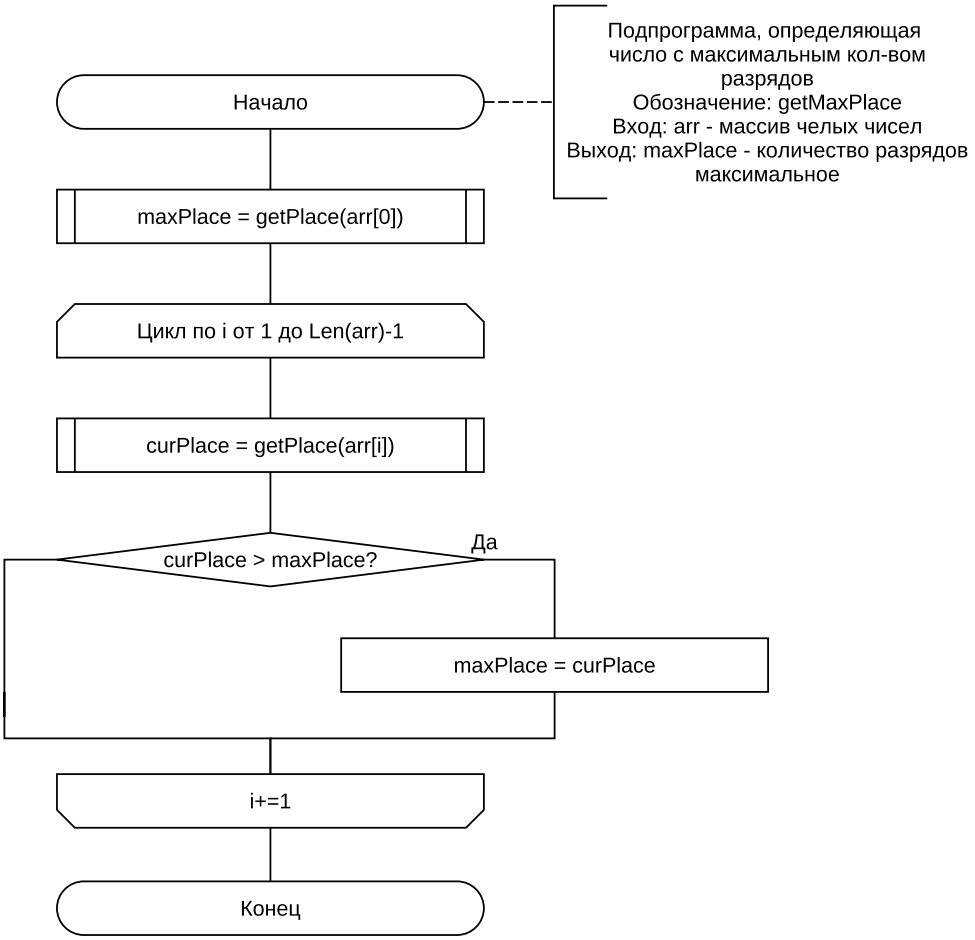


Рисунок 2.7 – Схема алгоритма получения максимального количества разрядов числа в массиве

На рисунке 2.8 приведена схема алгоритма подпрограммы получения количества разрядов числа, используемого в алгоритме получения максимального количества разрядов числа в массиве.

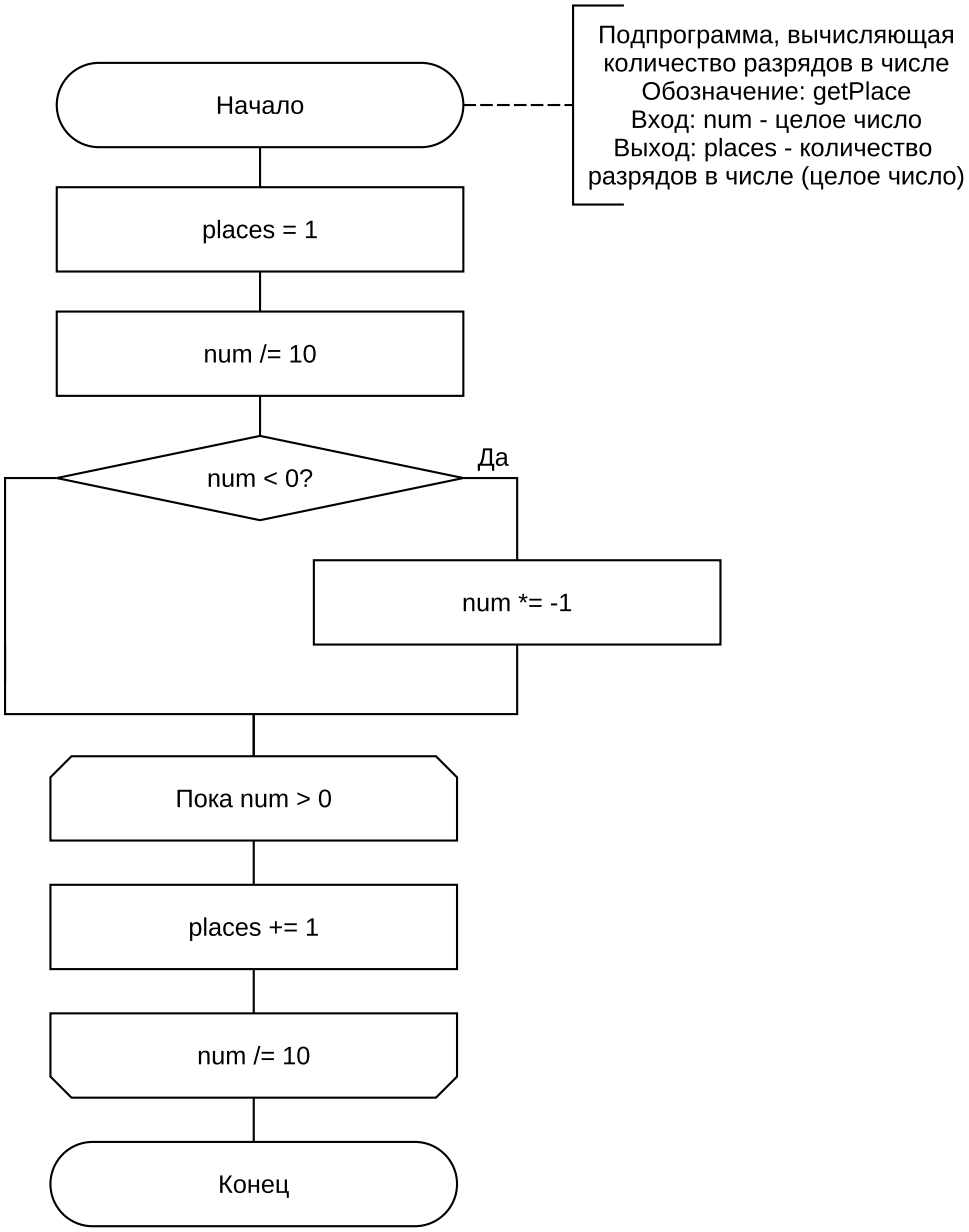


Рисунок 2.8 – Схема алгоритма получения количества разрядов числа

## 2.3. Разработка алгоритма сортировки слиянием

На рисунке 2.9 приведена схема алгоритма сортировки слиянием.

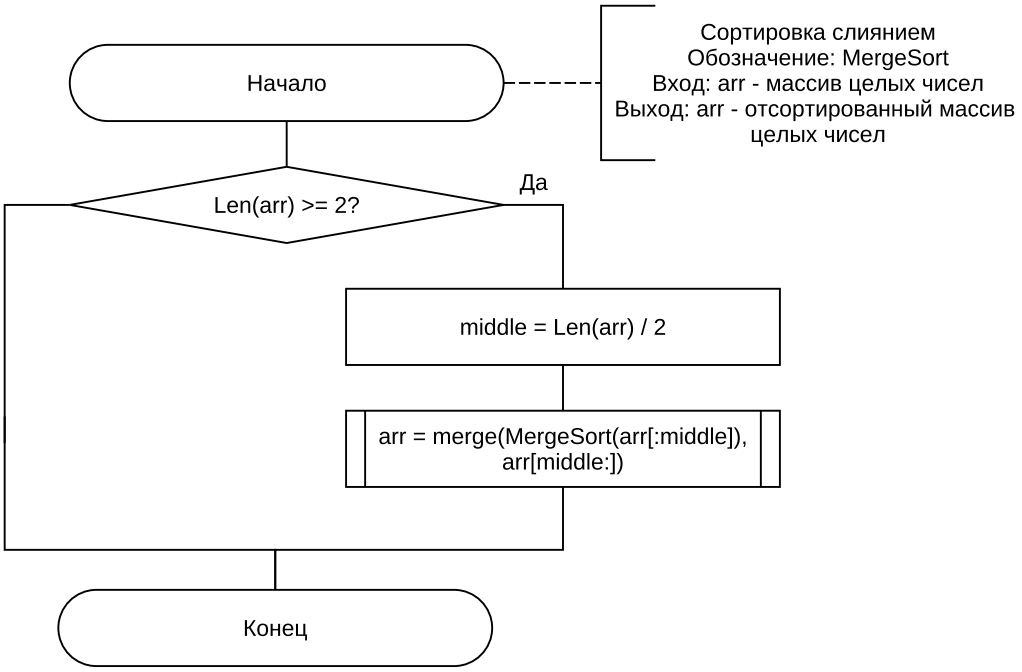


Рисунок 2.9 – Схема алгоритма сортировки слиянием

На рисунке 2.10 приведена схема алгоритма подпрограммы слияния массивов, используемого в сортировке слиянием.

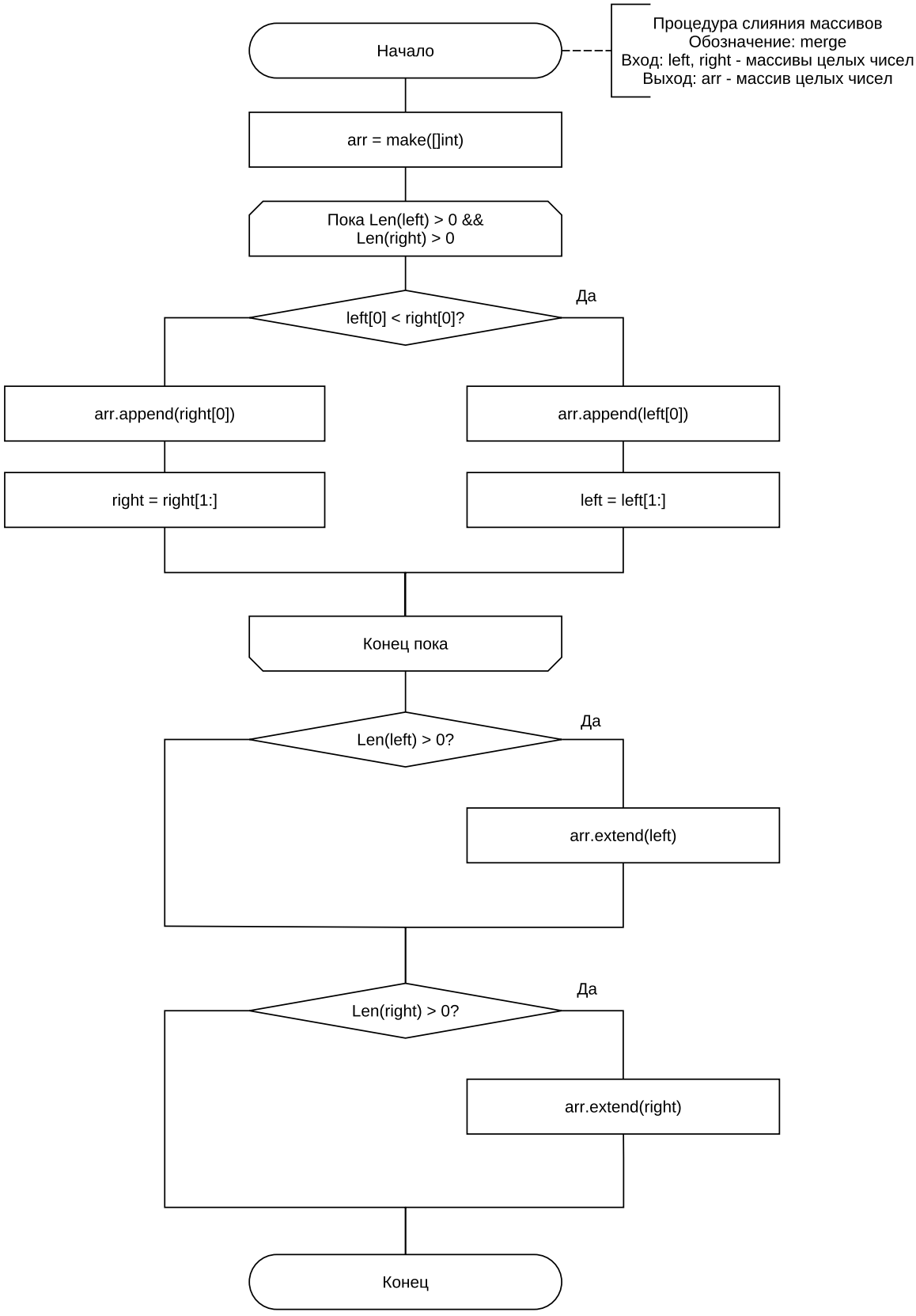


Рисунок 2.10 – Схема алгоритма подпрограммы слияния массивов

## 2.4. Трудоемкость алгоритмов

### 2.4.1 Модель вычислений

Чтобы провести вычисление трудоемкости алгоритмов сортировки, введем модель вычислений:

1. операции из списка (2.1) имеют трудоемкость 1;

$$+, -, ==, !=, <, >, <=, >=, [], ++, --, +=, -=, =, \&\&, || \quad (2.1)$$

2. операции из списка (2.2) имеют трудоемкость 2;

$$*, /, *=, /=, \% \quad (2.2)$$

3. трудоемкость оператора выбора `if условие then A else B` рассчитывается, как (2.3);

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \quad (2.3)$$

4. трудоемкость цикла рассчитывается, как (2.4);

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}) \quad (2.4)$$

5. трудоемкость вызова функции равна 0.

### 2.4.2 Трудоёмкость алгоритма блочной сортировки

Обозначим за $N$ длину массива.

Для алгоритма блочной сортировки трудоемкость будет слагаться из:

— инициализаций минимума, максимума и массива корзин, суммарная трудоёмкость которых в худшем случае: $f = 2+4+1+1+N \cdot (1+1+$

$2+2) = 8+6N$, в лучшем случае: $f = 2+4+1+1+N \cdot (1+1+1+1) = 8+4N$;

— инициализации корзин, трудоёмкость которой: $f = 1+1+N \cdot (1+1+9+4) = 2+15N$;

— цикла по $i \in [0..N-1]$, трудоёмкость которого в случае выполнения условия: $f = 1+1+N \cdot (1+1+2+3+f_{insertionsort})$, в случае невыполнения условия: $f = 1+1+N \cdot (1+1+2+4+f_{blocksort})$;

— инициализации переменной pos, трудоёмкость которой: $f = 1$;

— заполнения результирующего массива из корзин, трудоёмкость которого в случае выполнения условия: $f = 1+1+N \cdot (1+1+2+(1+1+N \cdot (1+1+5))) = 7N^2+6N+2$, в случае невыполнения условия: $f = 1+1+N \cdot (1+1+2) = 4N+2$.

Трудоёмкость алгоритма сортировки вставками $f_{insertionsort}$, используемого в блочной сортировке складывается из:

— цикла по $i \in [0..N-1]$, трудоёмкость которого в лучшем случае (на уже отсортированном масиве): $f = 1+1+N \cdot (1+1+2+2+3) = 9N+2$, в худшем случае (на отсортированном в обратном порядке массиве): $f = 1+1+N \cdot (1+1+2+2+(1+1+16N)+3) = 16N^2+11N+2$.

**Лучший случай** для реализуемого алгоритма блочной сортировки наступает, когда на вход подаётся массив размером меньше threshold (константа, отвечающая за выбор дальнейшего разбиения или применения алгоритма сортировки вставками) и данный массив уже отсортирован. В таком случае трудоёмкость алгоритма блочной сортировки составит:

$f = 8+4N+2+15N+1+1+1 \cdot (1+1+2+3+9N+2)+1+7+6+2 = 28N+37$

Оценка трудоёмкости по самому быстрорастущему слагаемому из зависимости от входных данных: $O(n)$.

**Худший случай** для реализуемого алгоритма блочной сортировки наступает, когда на вход подаётся массив размером больше threshold (константа, отвечающая за выбор дальнейшего разбиения или применения алгоритма сортировки вставками) и данный массив отсортирован в обратном

порядке, а также элементы равномерно распределяются по всем корзинам. В таком случае трудоёмкость алгоритма блочной сортировки составит:

$f = R \cdot (8 + 6N + 2 + 15N + 1 + 1 + N \cdot (1 + 1 + 2 + 3 + 8 + 6N + 2 + 15N + 1 + 1 + N \cdot (1 + 1 + 2 + 3 + 16N^2 + 11N + 2)) + 1 + 7N^2 + 6N + 2) = 16N^4R + 11N^3R + 37N^2R + 46NR + 15R$

Оценка трудоёмкости по самому быстрорастущему слагаемому из зависимости от входных данных: $O(n^4)$.

### 2.4.3 Трудоёмкость алгоритма поразрядной сортировки

Обозначим за $K$ максимальное количество разрядов, среди чисел массива, за $N$ — количество чисел в массиве.

Для алгоритма поразрядной сортировки трудоемкость будет слагаться из:

- поиска максимального количества разрядов, трудоёмкость которого в лучшем случае: $f = 2 + 3 + (1 + K \cdot (1 + 1 + 1)) + 1 + 1 + (N - 2) \cdot (2 + 3K + 4 + 1) + 2 + 3K + 4 + 2 = 3KN + 7N + 2$, в худшем случае: $f = 2 + 4 + (1 + K \cdot (1 + 1 + 1)) + 1 + 1 + (N - 1) \cdot (2 + 3K + 5 + 2) = 3KN + 9N - 1$;

- создания и заполнения двумерного массива для разрядов, трудоёмкость которых: $f = 3 + 4 + 20 \cdot (4 + 2) = 127$;

- основного цикла сортировки по разрядам по $place \in [0..K - 1]$, трудоёмкость которого в лучшем случае: $f = 1 + 1 + K \cdot (1 + 1 + (1 + 1 + N \cdot (1 + 1 + 13)) + 1 + (1 + 1 + (1 + 1 + 1 + 1 + 1 + N \cdot (1 + 1 + 5) + 19 \cdot (1 + 1 + 1)) + 40)) = 22KN + 109K + 2$, в худшем случае: $1 + 1 + K \cdot (1 + 1 + (1 + 1 + N \cdot (1 + 1 + 13)) + 1 + (1 + 1 + 20 \cdot (1 + 1 + 1 + 1 + 1 + \frac{N}{20} \cdot (1 + 1 + 5) + 1))) = 22KN + 110K + 2$.

И **худший** ($f = 25KN + 110K + 9N + 128$), и **лучший случай** ($f = 25KN + 109K + 7N + 131$) существенно друг от друга не отличаются и характеризуются линейной оценкой трудоёмкости: $O(n)$

### 2.4.4 Трудоёмкость алгоритма сортировки слиянием

Для алгоритма сортировки слиянием трудоемкость будет слагаться из трудоёмкости собственно основной функции сортировки, осуществляющей рекурсивные вызовы, и из трудоёмкости функции слияния отсортированных частей.

Трудоёмкость функции слияния отсортированных частей для массивов длиной $N_1$ и $N_2$ складывается из:

- инициализации результирующего массива: $f = 1$;
- цикла слияния массивов: $f = 3 + N_1 \cdot (3 + 3 + 4))$;
- проверки двух условий для дозаполнения массива: $f = 1 + 1 + N_2 - N_1 = N_2 - N_1 + 2$.

Итого, трудоёмкость операции слияния массивов составляет: $f = 9N_1 + N_2 + 5$. Если массивы имеют одинаковую длину, трудоёмкость составит: $f = 10N + 5$, где $N$ — длина результирующего массива.

Алгоритм, получая на входе массив из $N$ элементов, делит его пополам, поэтому будет рассмотрен случай, когда $N = 2^k, k = \log_2 N$.

В этом случае при рекурсивных вызовах строится полное дерево рекурсивных вызовов глубиной $k$, содержащее $N$ листьев. Если количество вершин дерева обозначить через $V$, то его можно выразить следующим образом:

$$V = N + \frac{N}{2} + \frac{N}{4} + \ldots + 1 = n \cdot (1 + \frac{1}{2} + \frac{1}{4} + \ldots + 1) = 2N - 1 = 2^{k+1} - 1 \quad (2.5)$$

Из $2^{k+1} - 1$ все внутренние вершины порождают рекурсию, количество таких вершин — $V_r = N - 1$, остальные $N$ вершин — это вершины, в которых рассматривается только один элемент массива, что не вызывает дальнейших рекурсивных вызовов.

Таким образом, для $N$ вершин суммарная трудоёмкость составит: $f = 1 \cdot N = N$.

Суммарная трудоёмкость в любом случае составит: $f = (4+10N+5)+(4+10\cdot\frac{N}{2}+5)+(4+10\cdot\frac{N}{4}+5)+\ldots = 9+N+(k-1)\cdot(10N+5) = 9+N+(\log_2 N-1)\cdot(10N+5) = 10N\log_2 N+5\log_2 N-9N+4$.

Оценка трудоёмкости по самому быстрорастущему слагаемому из зависимости от входных данных: $O(n\log n)$.

# Вывод

В данном разделе представлены схемы алгоритмов сортировки массивов целых чисел: алгоритма блочной сортировки, алгоритма поразрядной сортировки, алгоритма сортировки слиянием, также оценена их трудоёмкость.

По итогам расчётов алгоритм блочной сортировки в худшем случае имеет наибольшую трудоёмкость $O(n^4)$. Данный алгоритм сильно зависит от выбора функции распределения элементов по корзинам, а также от природы входных данных. Однако в лучшем случае имеет линейную оценку трудоёмкости $O(n)$.

Наименее трудоёмким является алгоритм поразрядной сортировки: он имеет линейную оценку трудоёмкости $O(n)$ на всех наборах, с которыми может работать. Однако данный алгоритм сортировки сильно ограничен в типах элементов наборов, что делает его неприменимым во многих задачах.

Алгоритм сортировки слиянием во всех случаях имеет оценку трудоёмкости $O(n\log n)$, что делает его применимым в задачах, в которых неизвестно ничего о внутреннем устройстве входных данных.

# 3 Технологическая часть

В данном разделе представлены требования к программному обеспечению, а также рассматриваются средства реализации и приводятся листинги кода.

## 3.1. Требования к программному обеспечению

К программе предъявляется ряд требований: на вход подаётся массив, содержащий целые числа, на выходе — отсортированный массив, содержащий целые числа.

## 3.2. Средства реализации

В качестве языка программирования для реализации лабораторной работы был выбран многопоточный язык Go [4]. Выбор был сделан в пользу данного языка программирования, вследствие наличия пакетов для тестирования программного обеспечения.

## 3.3. Реализация

В листингах 3.1–3.2 приведена реализация алгоритма блочной сортировки.

Листинг 3.1 – Алгоритм блочной сортировки

```
func InsertionSort(arr []int) []int {
    for i := 1; i < len(arr); i++ {
        el := arr[i]
        j := i - 1
        for ; j >= 0 && arr[j] > el; j-- {
            arr[j+1] = arr[j]
        }
        arr[j+1] = el
    }

    return arr
}

func findMinAndMax(arr []int) (min int, max int) {
    min = arr[0]
    max = arr[0]
    for _, value := range arr {
        if value < min {
            min = value
        }
        if value > max {
            max = value
        }
    }
    return min, max
}

func BlockSort(arr []int, nBuckets int) []int {
    buckets := make([][]int, nBuckets)
    min, max := findMinAndMax(arr)

    for i := 0; i < nBuckets; i++ {
        buckets[i] = make([]int, 0, len(arr))
    }

    for i := range arr {
        ind := nBuckets * (arr[i] - min) / (max - min + 1)
        buckets[ind] = append(buckets[ind], arr[i])
    }

    for i := range buckets {
        if len(buckets[i]) < threshold && len(buckets[i]) > 0 {
            buckets[i] = InsertionSort(buckets[i])
        } else if len(buckets[i]) >= threshold {
            buckets[i] = BlockSort(buckets[i], len(buckets[i]))
        }
    }
```

Листинг 3.2 – Продолжение листинга 3.1

```
    pos := 0
    for i := range buckets {
        if len(buckets[i]) != 0 {
            for j := 0; j < len(buckets[i]); j++ {
                arr[pos] = buckets[i][j]
                pos++
            }
        }
    }

    return arr
}
```

В листингах 3.3–3.4 приведена реализация алгоритма поразрядной сортировки.

Листинг 3.3 – Алгоритм поразрядной сортировки

```
func getPlace(num int) int {
    places := 1
    num /= 10

    if num < 0 {
        num *= -1
    }

    for ; num > 0; num /= 10 {
        places++
    }
    return places
}

func getMaxPlace(arr []int) int {
    maxPlace := getPlace(arr[0])

    for i := 1; i < len(arr); i++ {
        curPlace := getPlace(arr[i])
        if curPlace > maxPlace {
            maxPlace = curPlace
        }
    }

    return maxPlace
}

func RadixSort(arr []int) []int {
    var ind int
```

Листинг 3.4 – Продолжение листинга 3.3

```
    maxPlace := getMaxPlace(arr)

    buckets := make([][]int, 2*base)
    for i := 0; i < 2*base; i++ {
        buckets[i] = make([]int, 0, len(arr))
    }

    for place := 0; place < maxPlace; place++ {
        for j := range arr {
            if arr[j] < 0 {
                ind = base - (-arr[j]/int(math.Pow10(place)))%base
            } else {
                ind = base + (arr[j]/int(math.Pow10(place)))%base
            }
            buckets[ind] = append(buckets[ind], arr[j])
        }
        pos := 0
        for m := range buckets {
            if len(buckets[m]) != 0 {
                for n := range buckets[m] {
                    arr[pos] = buckets[m][n]
                    pos++
                }
            }
            buckets[m] = make([]int, 0, len(arr))
        }
    }

    return arr
}
```

В листингах 3.5–3.6 приведена реализация алгоритма сортировки слиянием.

Листинг 3.5 – Алгоритм сортировки слиянием

```
func merge(left, right []int) []int {
    arr := make([]int, 0)
    for len(left) > 0 && len(right) > 0 {
        if left[0] < right[0] {
            arr = append(arr, left[0])
            left = left[1:]
        } else {
            arr = append(arr, right[0])
            right = right[1:]
        }
    }
```

Листинг 3.6 – Продолжение листинга 3.5

```
    if len(left) > 0 {
        arr = append(arr, left...)
    }
    if len(right) > 0 {
        arr = append(arr, right...)
    }

    return arr
}

func MergeSort(arr []int) []int {
    if len(arr) >= 2 {
        middle := len(arr) / 2
        arr = merge(MergeSort(arr[:middle]), MergeSort(arr[middle:]))
    }

    return arr
}
```

Листинги со служебным кодом находятся в приложении.

## 3.4. Функциональное тестирование

В таблице 3.1 приведены функциональные тесты для алгоритмов сортировки массивов целых чисел: алгоритма блочной сортировки, алгоритма поразрядной сортировки, алгоритма сортировки слиянием. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

| **Массив** | **Ожидаемый результат** |
|---|---|
| 1 2 3 4 5 | 1 2 3 4 5 |
| 5 4 3 2 1 | 1 2 3 4 5 |
| 1 1 1 1 1 | 1 1 1 1 1 |
| 1 23 −3 94 −517 | −517 −3 1 23 94 |
| 1 | 1 |

## 3.5. Пример работы

Демонстрация работы программы приведена на рисунке 3.1.

```
Добро пожаловать в главное меню!

        1. Ввести размерность массива и заполнить его случайными числами (неотсорт.)
        2. Ввести размерность массива и заполнить его случайным числом
        3. Уже отсортированный массив, заполненный случайными числами
        4. Отсортированный в обратном порядке массив, заполненный случайными числами
        5. Ввести массив вручную
        6. Вывести массив

        Отсортировать массив:
        7. блочной сортировкой
        8. поразрядной сортировкой
        9. сортировкой слиянием

        10. Замеры времени

        11. Вывести меню
        0. Выход

Выберите опцию: 1

Введите количество элементов массива: 5

Выберите опцию: 6

Массив:
[272 94 -713 229 196]

Выберите опцию: 7

Отсортированный массив:
[-713 94 196 229 272]

Выберите опцию: 0
```

Рисунок 3.1 – Демонстрация работы программы

# Вывод

В результате, были разработаны и протестированы следующие алгоритмы: алгоритм блочной сортировки, алгоритм поразрядной сортировки, алгоритм сортировки слиянием.

# 4 Исследовательская часть

## 4.1. Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

— операционная система: Ubuntu [5] 22.04.3 LTS;

— оперативная память: 15 ГБ;

— процессор: 12 ядер, AMD Ryzen 5 4600ГЦ with Radeon Graphics [6].

## 4.2. Время выполнения алгоритмов

Все замеры были проведены в одинаковых условиях.

Замеры проводились с помощью разработанной функции GetCPU, которая использует модуль syscall [7]. syscall.Rusage является структурой в пакете syscall, предназначенной для хранения информации о ресурсах, используемых процессом или потоком. Эта структура содержит различные поля, такие как Utime (время использования ЦПУ в пользовательском режиме), Stime (время использования ЦПУ в режиме ядра) и так далее.

Для получения времени использования ЦПУ, извлекаем атрибуты структуры Utime и Stime.

Выражение usage.Utime.Nano() + usage.Stime.Nano() возвращает сумму времени использования ЦПУ в наносекундах для процесса или потока. Это значение может быть использовано для измерения общего времени использования ЦПУ в пределах процесса или потока.

По устройству входных данных можно выделить четыре случая:

— неупорядоченный массив, заполненный случайными числами;

— уже отсортированный массив, заполненный случайными числами;

— отсортированный в обратном порядке массив, заполненный случайными числами;

— массив, заполненный одинаковыми числами.

Обозначим:

— 1 — алгоритм блочной сортировки;

— 2 — алгоритм поразрядной сортировки;

— 3 — алгоритм сортировки слиянием.

Для случая 1 (неупорядоченный массив, заполненный случайными числами) результаты замеров приведены в таблице 4.1.

Таблица 4.1 – Замер времени для массивов размером от 60 до 960: неупорядоченный массив, заполненный случайными числами

| **Размер массива** | **Время, с** | | |
|---|---|---|---|
| | **1** | **2** | **3** |
| 60 | 0.0000230 | 0.0000269 | 0.0000207 |
| 110 | 0.0000737 | 0.0000469 | 0.0000409 |
| 160 | 0.0001593 | 0.0000670 | 0.0000653 |
| 210 | 0.0002166 | 0.0000818 | 0.0000761 |
| 260 | 0.0003162 | 0.0001067 | 0.0001056 |
| 310 | 0.0005103 | 0.0001417 | 0.0001405 |
| 360 | 0.0005355 | 0.0001293 | 0.0001543 |
| 410 | 0.0007342 | 0.0001404 | 0.0001752 |
| 460 | 0.0013891 | 0.0002050 | 0.0001768 |
| 510 | 0.0017596 | 0.0002082 | 0.0001959 |
| 560 | 0.0014765 | 0.0001999 | 0.0002651 |
| 610 | 0.0019370 | 0.0002371 | 0.0002815 |
| 660 | 0.0022556 | 0.0002334 | 0.0003008 |
| 710 | 0.0028482 | 0.0002455 | 0.0003156 |
| 760 | 0.0029105 | 0.0002523 | 0.0003345 |
| 810 | 0.0030107 | 0.0002591 | 0.0003471 |
| 860 | 0.0032621 | 0.0002710 | 0.0003997 |
| 910 | 0.0055645 | 0.0004126 | 0.0004037 |
| 960 | 0.0057715 | 0.0004102 | 0.0003897 |

Для случая 2 (уже отсортированный массив, заполненный случайными числами) результаты замеров приведены в таблице 4.2.

Таблица 4.2 – Замер времени для массивов размером от 60 до 960: уже отсортированный массив, заполненный случайными числами

| **Размер массива** | **Время, с** | | |
|---|---|---|---|
| | **1** | **2** | **3** |
| 60 | 0.0000178 | 0.0000287 | 0.0000193 |
| 110 | 0.0000598 | 0.0000491 | 0.0000406 |
| 160 | 0.0001274 | 0.0000718 | 0.0000662 |
| 210 | 0.0001824 | 0.0000843 | 0.0000777 |
| 260 | 0.0003170 | 0.0001089 | 0.0001067 |
| 310 | 0.0005138 | 0.0001419 | 0.0001370 |
| 360 | 0.0005472 | 0.0001283 | 0.0001564 |
| 410 | 0.0007269 | 0.0001433 | 0.0001739 |
| 460 | 0.0014211 | 0.0002042 | 0.0001727 |
| 510 | 0.0017514 | 0.0002036 | 0.0001940 |
| 560 | 0.0014893 | 0.0001990 | 0.0002540 |
| 610 | 0.0019461 | 0.0002346 | 0.0002816 |
| 660 | 0.0022517 | 0.0002390 | 0.0003061 |
| 710 | 0.0028303 | 0.0002445 | 0.0003299 |
| 760 | 0.0029102 | 0.0002551 | 0.0003318 |
| 810 | 0.0029992 | 0.0002696 | 0.0003537 |
| 860 | 0.0032282 | 0.0002703 | 0.0003719 |
| 910 | 0.0055146 | 0.0004111 | 0.0004084 |
| 960 | 0.0057964 | 0.0004153 | 0.0003864 |

Для случая 3 (отсортированный в обратном порядке массив, заполненный случайными числами) результаты замеров приведены в таблице 4.3.

Таблица 4.3 – Замер времени для массивов размером от 60 до 960: отсортированный в обратном порядке массив, заполненный случайными числами

| **Размер массива** | **Время, с** | | |
|---|---|---|---|
| | **1** | **2** | **3** |
| 60 | 0.0000179 | 0.0000281 | 0.0000206 |
| 110 | 0.0000593 | 0.0000457 | 0.0000417 |
| 160 | 0.0001234 | 0.0000720 | 0.0000666 |
| 210 | 0.0001961 | 0.0000800 | 0.0000825 |
| 260 | 0.0003161 | 0.0001059 | 0.0001092 |
| 310 | 0.0005213 | 0.0001426 | 0.0001362 |
| 360 | 0.0005417 | 0.0001277 | 0.0001584 |
| 410 | 0.0007436 | 0.0001457 | 0.0001720 |
| 460 | 0.0013978 | 0.0002042 | 0.0001879 |
| 510 | 0.0017501 | 0.0002058 | 0.0001842 |
| 560 | 0.0014740 | 0.0002034 | 0.0002674 |
| 610 | 0.0019411 | 0.0002331 | 0.0002846 |
| 660 | 0.0022418 | 0.0002357 | 0.0002903 |
| 710 | 0.0028399 | 0.0002522 | 0.0003306 |
| 760 | 0.0029163 | 0.0002517 | 0.0003486 |
| 810 | 0.0030151 | 0.0002579 | 0.0003780 |
| 860 | 0.0032179 | 0.0002698 | 0.0004011 |
| 910 | 0.0055837 | 0.0004122 | 0.0004173 |
| 960 | 0.0058539 | 0.0004111 | 0.0004389 |

Для случая 4 (массив, заполненный одинаковыми числами) результаты замеров приведены в таблице 4.4.

Таблица 4.4 – Замер времени для массивов размером от 60 до 960: массив, заполненный одинаковыми числами

| **Размер массива** | **Время, с** | | |
|---|---|---|---|
| | **1** | **2** | **3** |
| 60 | 0.0000194 | 0.0000221 | 0.0000202 |
| 110 | 0.0000604 | 0.0000361 | 0.0000400 |
| 160 | 0.0001269 | 0.0000529 | 0.0000667 |
| 210 | 0.0001931 | 0.0000648 | 0.0000834 |
| 260 | 0.0003144 | 0.0000799 | 0.0001075 |
| 310 | 0.0005135 | 0.0001110 | 0.0001379 |
| 360 | 0.0005368 | 0.0001231 | 0.0001529 |
| 410 | 0.0007376 | 0.0001456 | 0.0001668 |
| 460 | 0.0013911 | 0.0002007 | 0.0001866 |
| 510 | 0.0017591 | 0.0002042 | 0.0001964 |
| 560 | 0.0014809 | 0.0001976 | 0.0002676 |
| 610 | 0.0021222 | 0.0002349 | 0.0002858 |
| 660 | 0.0023415 | 0.0002367 | 0.0002959 |
| 710 | 0.0028345 | 0.0002505 | 0.0003315 |
| 760 | 0.0028885 | 0.0002531 | 0.0003485 |
| 810 | 0.0029731 | 0.0002612 | 0.0003597 |
| 860 | 0.0031907 | 0.0002680 | 0.0003998 |
| 910 | 0.0053525 | 0.0004040 | 0.0003987 |
| 960 | 0.0048988 | 0.0004073 | 0.0004264 |

На следующем графике представлена зависимость времени работы алгоритмов сортировки от размера массива в случае 1 (неупорядоченный массив, заполненный случайными числами).

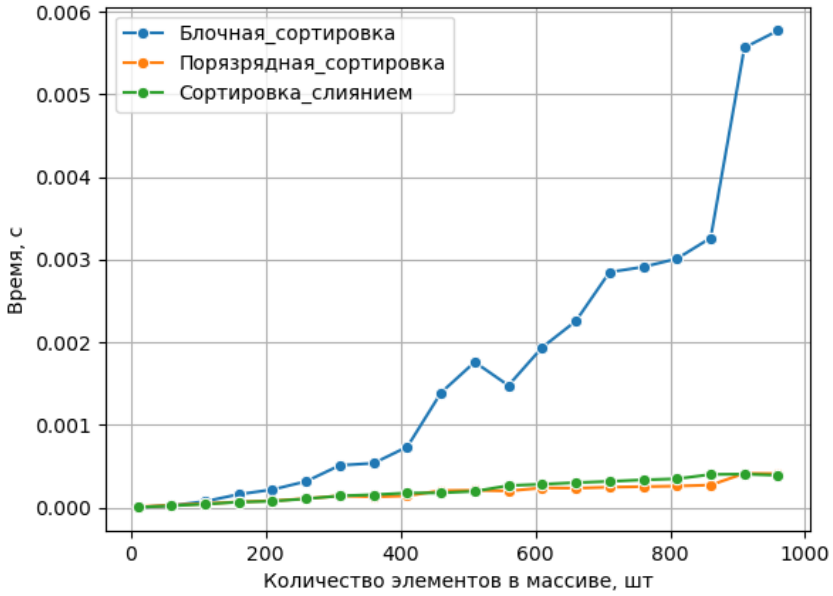


Рисунок 4.1 – Время работы реализаций алгоритмов сортировки на неупорядоченном массиве

На следующем графике представлена зависимость времени работы алгоритмов сортировки от размера массива в случае 2 (уже отсортированный массив, заполненный случайными числами).

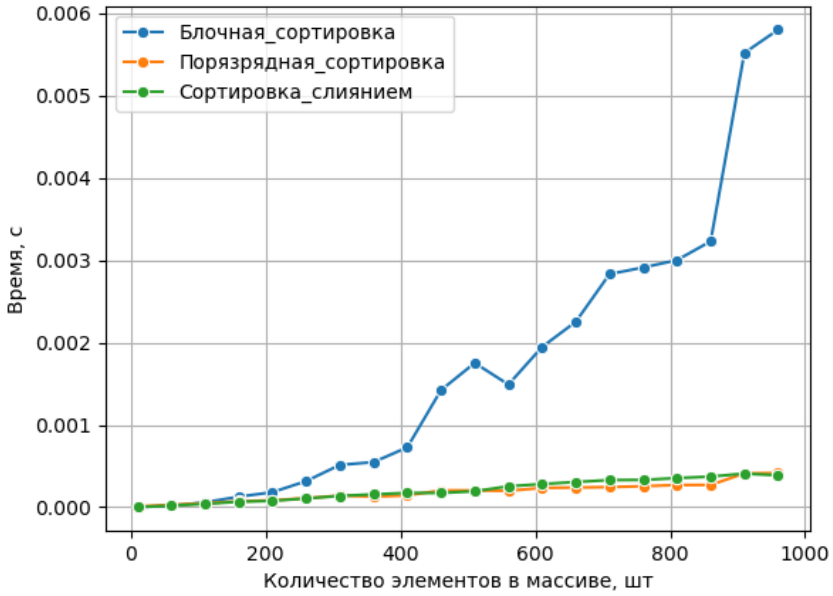


Рисунок 4.2 – Время работы реализаций алгоритмов сортировки на отсортированном массиве

На следующем графике представлена зависимость времени работы алгоритмов сортировки от размера массива в случае 3 (отсортированный в обратном порядке массив, заполненный случайными числами).

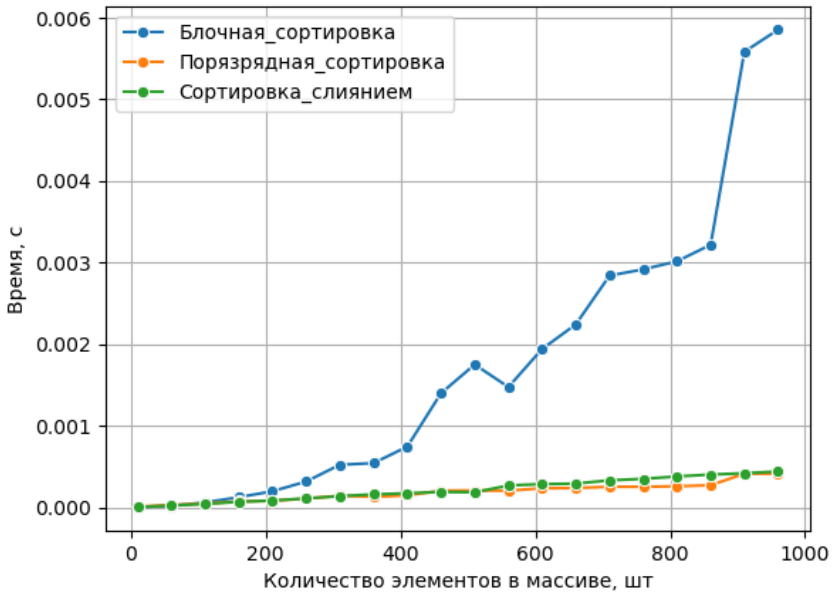


Рисунок 4.3 – Время работы реализаций алгоритмов сортировки на отсортированном в обратном порядке массиве

На следующем графике представлена зависимость времени работы алгоритмов сортировки от размера массива в случае 4 (массив, заполненный одинаковыми числамии).

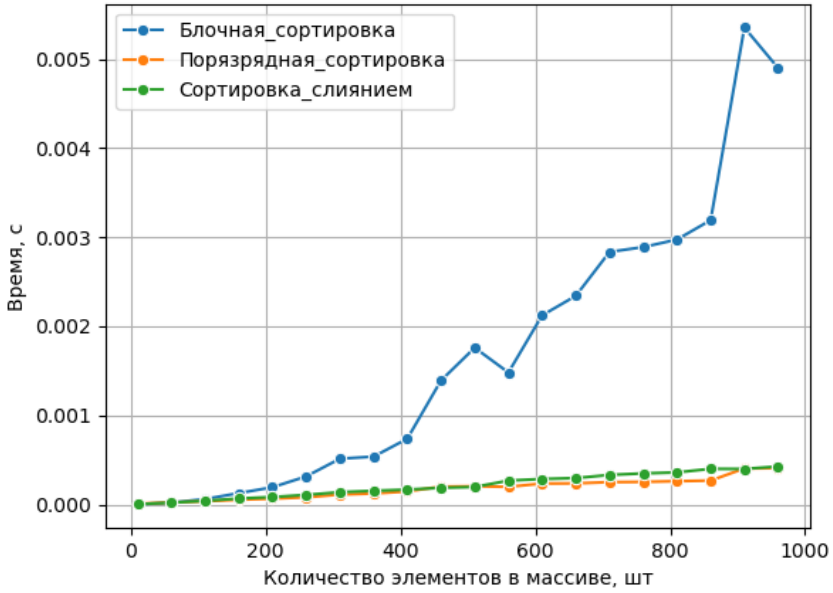


Рисунок 4.4 – Время работы реализаций алгоритмов сортировки на массиве, заполненном одинаковыми числами

## 4.3. Использование памяти

Выделение памяти при работе реализаций алгоритмов указано в листинге 4.1. Получено при помощи команды: `go test -bench . -benchmem`.

Листинг 4.1 – Использование памяти

```
goos: linux
goarch: amd64
pkg: lab03/algs
cpu: AMD Ryzen 5 4600H with Radeon Graphics
BenchmarkBlock100-12 27331 41861 ns/op 93184 B/op 102 allocs/op
BenchmarkBlock200-12 10185 116865 ns/op 365058 B/op 202 allocs/op
BenchmarkBlock500-12 1305 974032 ns/op 2064404 B/op 502 allocs/op
BenchmarkBlock700-12 691 1651312 ns/op 4325408 B/op 702 allocs/op
BenchmarkBlock1000-12 357 3564946 ns/op 8224808 B/op 1002 allocs/op
BenchmarkRadix100-12 38264 31286 ns/op 72576 B/op 81 allocs/op
BenchmarkRadix200-12 22082 54853 ns/op 145152 B/op 81 allocs/op
BenchmarkRadix500-12 8226 134449 ns/op 331778 B/op 81 allocs/op
BenchmarkRadix700-12 6502 165200 ns/op 497668 B/op 81 allocs/op
BenchmarkRadix1000-12 4345 266150 ns/op 663559 B/op 81 allocs/op
BenchmarkMerge100-12 34336 35820 ns/op 13256 B/op 307 allocs/op
BenchmarkMerge200-12 15704 77325 ns/op 30720 B/op 627 allocs/op
BenchmarkMerge500-12 5532 191907 ns/op 73425 B/op 1500 allocs/op
BenchmarkMerge700-12 3526 309925 ns/op 148762 B/op 2280 allocs/op
BenchmarkMerge1000-12 2809 433071 ns/op 171939 B/op 3006 allocs/op
PASS
ok lab03/algs 21.680s
```

На рисунке 4.5 представлено сравнение потребления памяти реализациями алгоритмов сортировки.

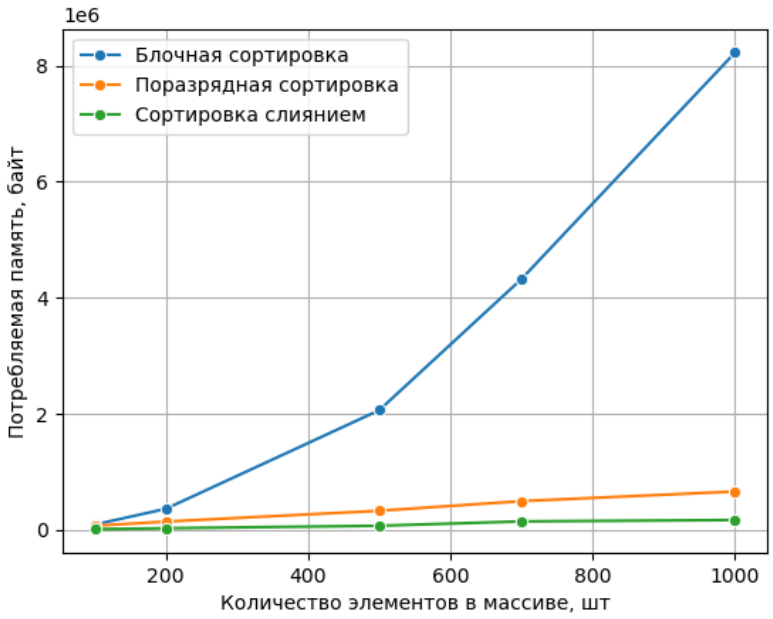


Рисунок 4.5 – Память, потребляемая реализациями алгоритмов сортировки

Реализация алгоритма блочной сортировки потребляет больше всего памяти, так как в ней выделяется множество блоков. Реализация алгоритма сортировки слиянием потребляет наименьшее количество памяти.

# Вывод

Реализация алгоритма блочной сортировки потребляет больше всего памяти и является наименее эффективной по времени. Это соотносится с оценкой трудоёмкости алгоритмов.

Реализации алгоритмов сортировки слиянием и поразрядной сортировки имеют примерно одинаковое время исполнения. Реализация алгоритма сортировки слиянием потребляет меньше всего памяти.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были выполнены все поставленные задачи:

— были исследованы алгоритмы сортировки;

— были применины методы динамического программирования для реализации алгоритмов сортировки;

— была произведена оценка и сравнение трудоёмкости алгоритмов сортировки;

— был проведён сравнительный анализ по времени и памяти на основе экспериментальных данных;

— был подготовлен отчет по лабораторной работе.

Исследование позволило выявить различия в производительности различных алгоритмов сортировки. В частности, реализация алгоритма поразрядной сортировки оказалась самой эффективной по времени исполнения наряду с реализацией алгоритма сортировки слиянием, который даёт примерно такие же результаты. Данные реализации алгоритмов работают примерно в 8 раз быстрее реализации алгоритма блочной сортировки на массиве длиной 610.

Также была проведена оценка трудоёмкости данных алгоритмов на заданной модели вычислений. В результате наиболее трудоёмким в среднем оказался алгоритм блочной сортировки. Алгоритм поразрядной сортировки имеет наименьшую трудоёмкость.

По потребляемой памяти наиболее выигрышным оказалась реализация алгоритма сортировки слиянием. Реализация алгоритма блочной сортировки потребляет наибольшее количество памяти вследствие выделения памяти под большое количество корзин, предусматриваемое алгоритмом.

В целом, исследование позволило получить практические и теоретические результаты, подтверждающие различия в производительности и использовании памяти различных алгоритмов сортировки. Эти результаты могут служить основой для выбора наиболее эффективного алгоритма в зависимости от конкретных требований и ограничений проекта.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Меркулова О. О. Никитина А. Б. Фёдоров О. А. Избранные вопросы теории алгоритмов. Учебно-методическое пособие. — Южно-Сахалинск : СахГУ, 2018.

[2] С. Абакумова А. Разработка алгоритмов сортировки данных большого объёма с использованием параллельных технологий. — НИУ БелГУ, 2017.

[3] К. Сахаров А. Модифицированный метод сортировки слиянием. — Алтайский государственный педагогический университет, 2015.

[4] The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/` (дата обращения: 17.10.2023).

[5] Linux — Официальная документация [Электронный ресурс]. Режим доступа: `https://www.linux.org/` (дата обращения: 17.10.2023).

[6] AMD Processors [Электронный ресурс]. Режим доступа: `https://www.amd.com/en.html` (дата обращения: 17.10.2023).

[7] Пакет syscall [Электронный ресурс]. Режим доступа: `https://pkg.go.dev/syscall` (дата обращения: 17.10.2023).

# ПРИЛОЖЕНИЕ А

В листинге А.1 приведен модуль для замеров процессорного времени выполнения программы.

Листинг А.1 — Измерение времени

```
package time_measure

import (
    "fmt"
    "lab03/algs"
    "lab03/inp_out"
    "syscall"
)

const N int = 10000
const nAlgs int = 3
const nTypesData int = 4

func GetCPU() int64 {
    usage := new(syscall.Rusage)
    syscall.Getrusage(syscall.RUSAGE_SELF, usage)
    return usage.Utime.Nano() + usage.Stime.Nano()
}

func blockSortTimeMeasurement(arr []int) (float32, []int) {
    var sum float32

    var startTime, finishTime int64
    var result []int

    for i := 0; i < N; i++ {
        result = make([]int, len(arr))
        copy(result, arr)
        startTime = GetCPU()
        result = algs.BlockSort(result, len(result))
        finishTime = GetCPU()
        sum += float32(finishTime - startTime)
    }

    result[0] += 1

    return (sum / float32(N)) / 1e+9, result
}

func radixSortTimeMeasurement(arr []int) (float32, []int) {
    var sum float32
```

```
42
43      var startTime, finishTime int64
44      var result []int
45
46      for i := 0; i < N; i++ {
47          result = make([]int, len(arr))
48          copy(result, arr)
49          startTime = GetCPU()
50          result = algs.RadixSort(result)
51          finishTime = GetCPU()
52          sum += float32(finishTime - startTime)
53      }
54
55      result[0] += 1
56
57      return (sum / float32(N)) / 1e+9, result
58  }
59
60  func mergeSortTimeMeasurement(arr []int) (float32, []int) {
61      var sum float32
62
63      var startTime, finishTime int64
64      var result []int
65
66      for i := 0; i < N; i++ {
67          result = make([]int, len(arr))
68          copy(result, arr)
69          startTime = GetCPU()
70          result = algs.MergeSort(result)
71          finishTime = GetCPU()
72          sum += float32(finishTime - startTime)
73      }
74
75      result[0] += 1
76
77      return (sum / float32(N)) / 1e+9, result
78  }
79
80  func MeasureTime(begin int, end int, step int) ([][][]float32, []int) {
81      seconds := make([][][]float32, nTypesData, nTypesData)
82      number := (end-begin)/step + 1
83      for i := range seconds {
84          seconds[i] = make([][]float32, nAlgs)
85          for j := range seconds[i] {
86              seconds[i][j] = make([]float32, number, number)
87          }
88      }
89
```

```
    sizes := make([]int, number)
    var arr, res []int

    func_pointers := [nAlgs]func([]int) (float32, []int){
        blockSortTimeMeasurement,
        radixSortTimeMeasurement,
        mergeSortTimeMeasurement,
    }

    for k := 0; k < nTypesData; k++ {
        for i := 0; i < nAlgs; i++ {
            for j := 0; j < number; j++ {
                arr = inp_out.GenerateArrayForTest(false, begin+j*step)
                seconds[k][i][j], res = func_pointers[i](arr)
                sizes[j] = begin + step*j
                fmt.Println(i, sizes[j])
            }
        }
    }

    res[0] *= 1

    return seconds, sizes
}
```