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 «Анализ алгоритмов»

на тему: «Алгоритмы сортировки»

Студент ИУ7-53Б _____ В. Марченко
(Группа) (Подпись, дата) (И. О. Фамилия)

Преподаватель _____ Ю. В. Строганов
(Подпись, дата) (И. О. Фамилия)

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

Москва — 2022 г.

Оглавление

Введение **4**

1 Аналитическая часть **5**

1.1 Цели и задачи 5

1.2 Блочная сортировка 5

1.3 Поразрядная сортировка 6

1.4 Сортировка перемешиванием 6

2 Конструкторская часть **7**

2.1 Описание алгоритмов 7

2.2 Модель вычислений 23

2.3 Трудоемкость алгоритмов 23

2.3.1 Алгоритм сортировки перемешиванием 23

2.3.2 Алгоритм поразрядной сортировки 24

2.3.3 Алгоритм блочной сортировки 26

3 Технологическая часть **29**

3.1 Требования к программному обеспечению 29

3.2 Средства реализации 29

3.3 Реализация алгоритмов 29

3.3.1 Вспомогательные функции 29

3.3.2 Алгоритм сортировки перемешиванием 30

3.3.3 Алгоритм поразрядной сортировки 31

3.3.4 Алгоритм блочной сортировки 34

3.4 Тестовые данные 36

4 Исследовательская часть **38**

4.1 Технические характеристики устройства 38

4.2 Время работы алгоритмов 38

Заключение **42**

Список использованных источников **43**

Введение

Люди встречаются с отсортированными объектами в телефонных книгах, в списках подоходных налогов, в оглавлениях книг, в библиотеках, в словарях, на складах —- почти везде, где нужно искать хранимые объекты. Сортировку следует понимать как процесс перегруппировки заданного множества объектов в некотором определенном порядке. Цель сортировки -— облегчить последующий поиск элементов в таком упорядоченном множестве [1].

Задача сортировки формально определяется следующим образом.

Вход: последовательность из n чисел $\langle a_1, a_2, ..., a_n \rangle$.

Выход: перестановка (изменение порядка) $\langle a'_1, a'_2, ..., a'_n \rangle$ входной последовательности таким образом, что для ее членов выполняется соотношение $a'_1 \leq a'_2 \leq ... \leq a'_n$ [2].

Существует множество различных алгоритмов сортировки. Для их сравнения вводится понятие трудоемкости. Под трудоемкостью алгоритма для данного конкретного входа в данной модели вычислений понимается количество «элементарных» операций, совершаемых алгоритмом.

1 Аналитическая часть

1.1 Цели и задачи

Цель работы: получить навыки оценки трудоемкости и временной эффективности на материале алгоритмов сортировки.

Задачи текущей лабораторной работы:

1) изучить и реализовать три алгоритма сортировки — перемешиванием, поразрядную и блочную;

2) провести сравнительный анализ трудоемкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;

3) провести сравнительный анализ алгоритмов на основе экспериментальных данных.

1.2 Блочная сортировка

Блочная сортировка (англ. bucket sort) — это алгоритм сортировки, который работает путем распределения элементов массива по нескольким сегментам («ведрам», блокам). Каждый блок сортируется индивидуально с использованием любого другого алгоритма сортировки. Очевидно, что трудоемкость блочной сортировки зависит от алгоритма, используемого для сортировки каждого сегмента.

Блочная сортировка работает следующим образом.

1. Инициализируется массив изначально пустых сегментов.

2. Происходит обход исходного массива, и каждый элемент помещается в соответствующий блок.

3. Элементы в каждом непустом сегменте сортируются.

4. Происходит обход блоков, и все элементы копируются в исходный массив.

1.3 Поразрядная сортировка

Поразрядная сортировка (англ. radix sort) — алгоритм сортировки, который выполняется за линейное время. Он позволяет избежать сравнений, создавая и распределяя элементы по «корзинам» в соответствии с их разрядом. Для элементов с более чем одной значащей цифрой этот процесс группирования повторяется для каждой цифры, сохраняя порядок предыдущего шага, пока не будут учтены все цифры.

Поразрядная сортировка может применяться к данным, которые можно отсортировать лексикографически, будь то целые числа, слова, перфокарты, игральные карты или почта [3].

1.4 Сортировка перемешиванием

Сортировка перемешиванием (англ. cocktail sort, shaker sort, shuffle sort) — разновидность пузырьковой сортировки. Анализируя метод пузырьковой сортировки, можно отметить два обстоятельства. Во-первых, если при обходе части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, ее можно исключить из рассмотрения. Во-вторых, при обходе от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо. Эти две идеи приводят к следующим модификациям в методе пузырьковой сортировки: границы рабочей части массива (обход которой происходит в текущий момент) устанавливаются в месте последнего обмена на каждой итерации и массив обходится поочередно справа налево и слева направо.

Вывод

В текущем разделе были рассмотрены три алгоритма сортировки: перемешиванием, поразрядная и блочная.

2 Конструкторская часть

2.1 Описание алгоритмов

На рисунках 2.1–2.3 показаны схемы алгоритмов вспомогательных функций, которые используются в рассматриваемых алгоритмах сортировки. На рисунках 2.4–2.15 показаны схемы трех алгоритмов сортировки — перемешиванием, поразрядной и блочной.

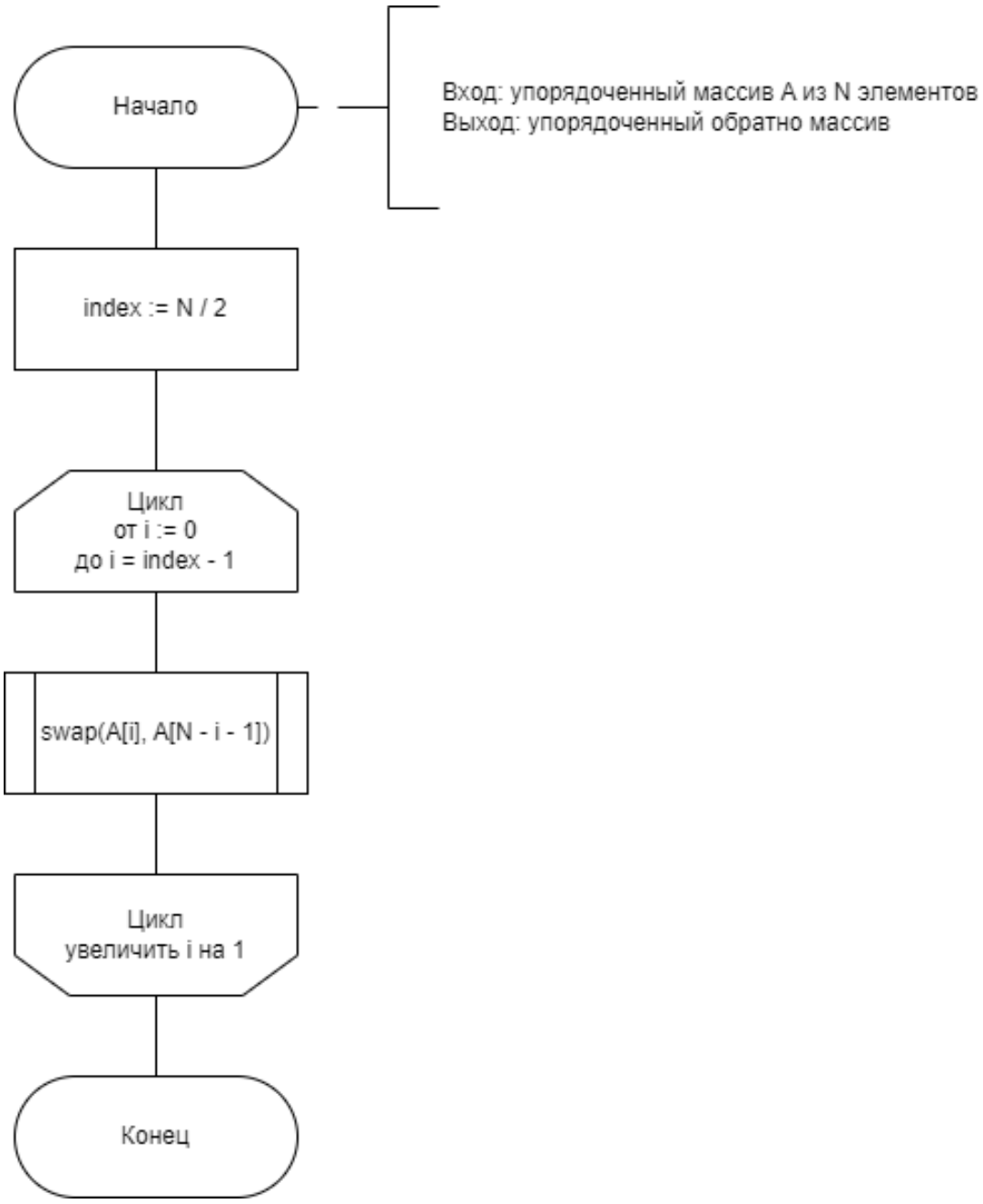

Рисунок 2.1 – Алгоритм получения массива, в котором элементы находятся в порядке, обратном исходному

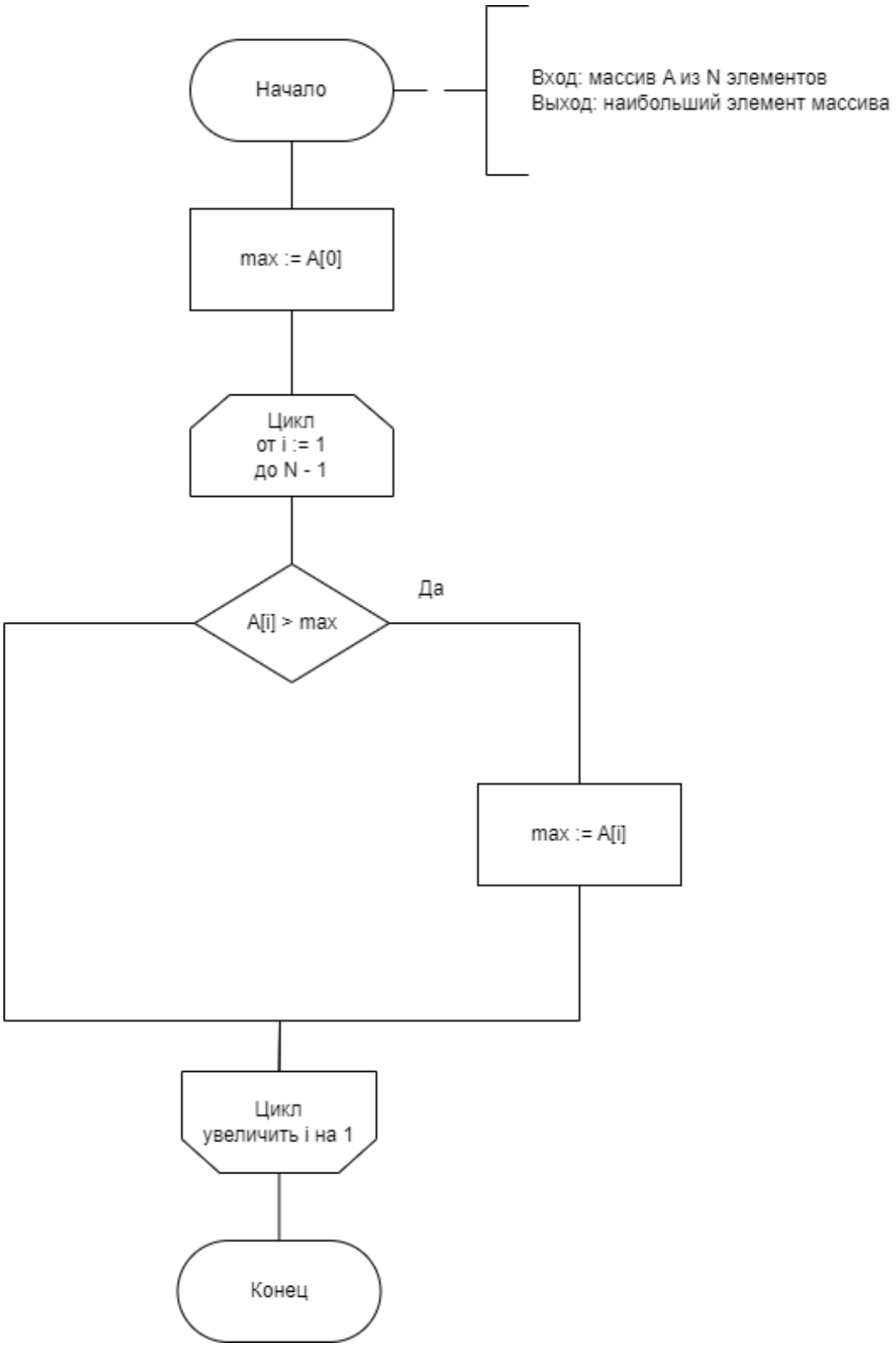

Рисунок 2.2 – Алгоритм поиска наибольшего элемента в массиве

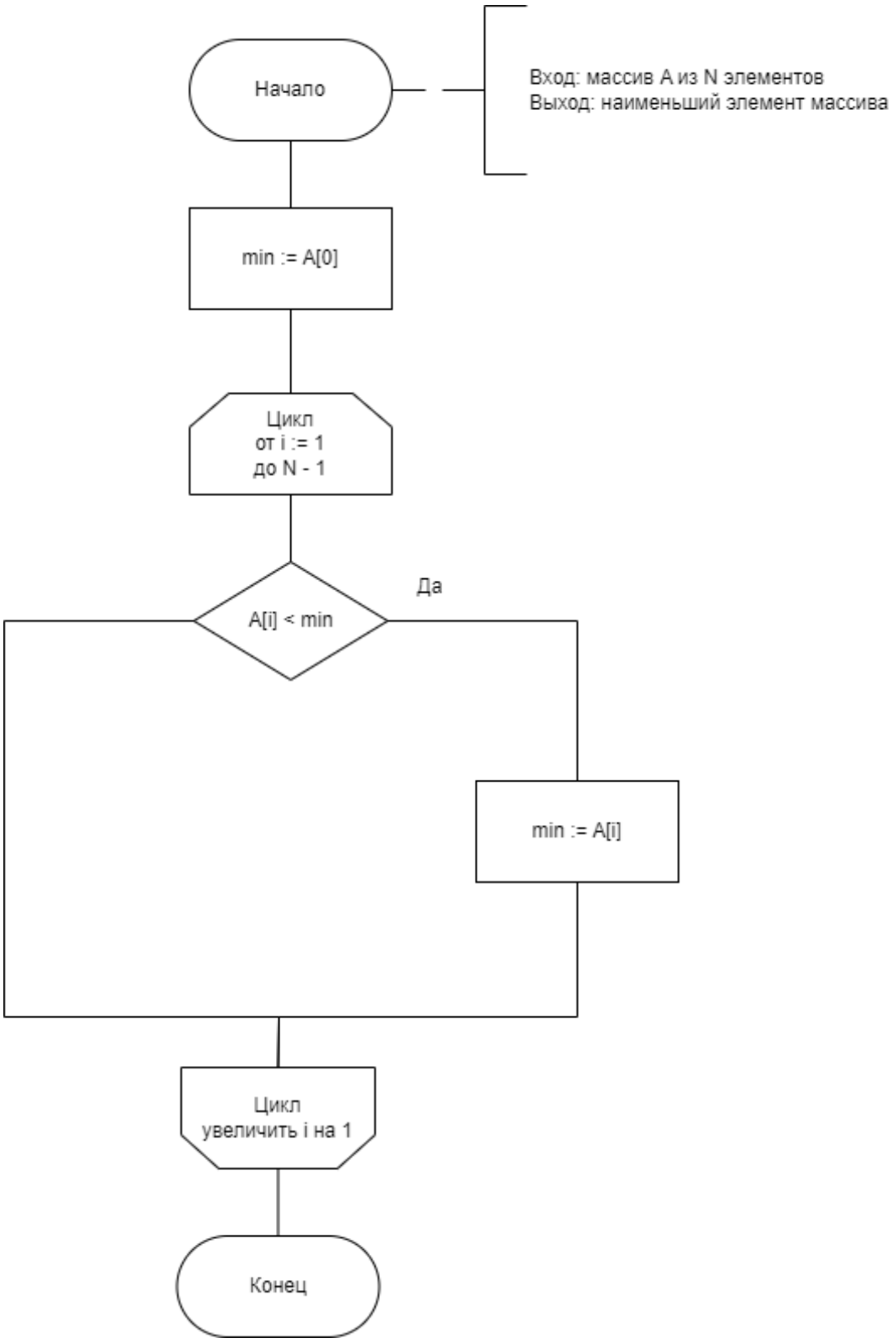

Рисунок 2.3 – Алгоритм поиска наименьшего элемента в массиве

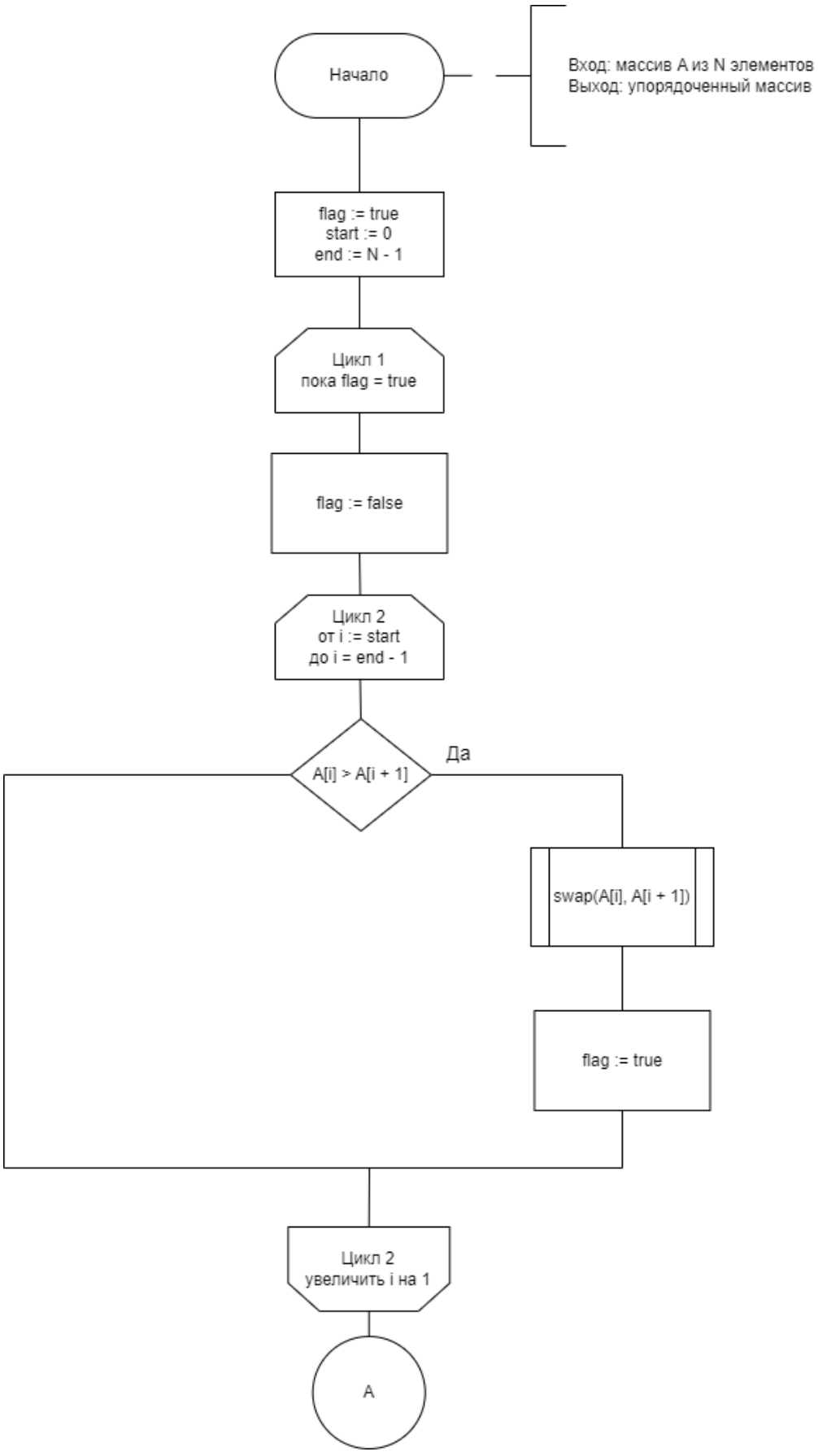

Рисунок 2.4 – Алгоритм сортировки перемешиванием — ч. 1

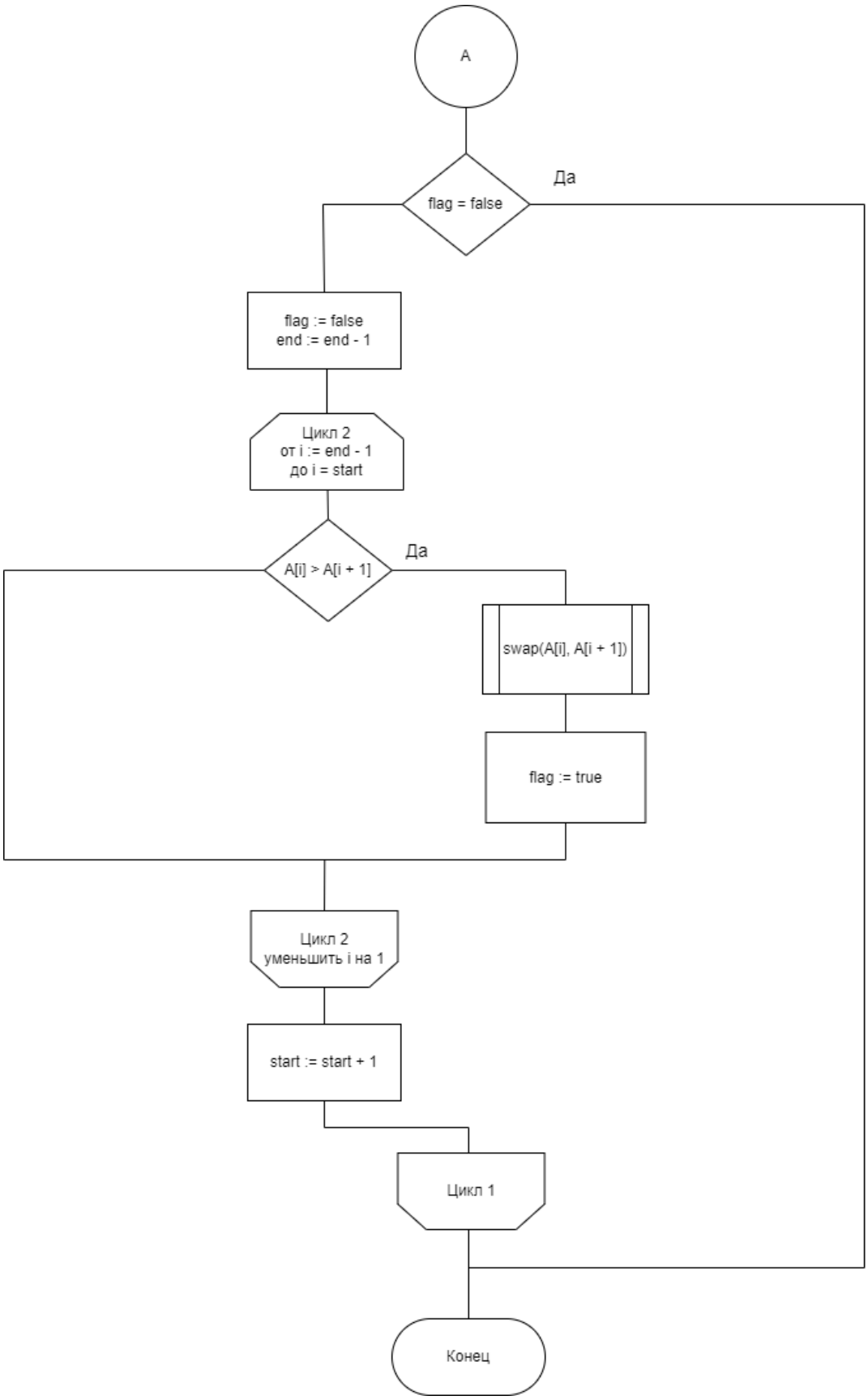

Рисунок 2.5 – Алгоритм сортировки перемешиванием — ч. 2

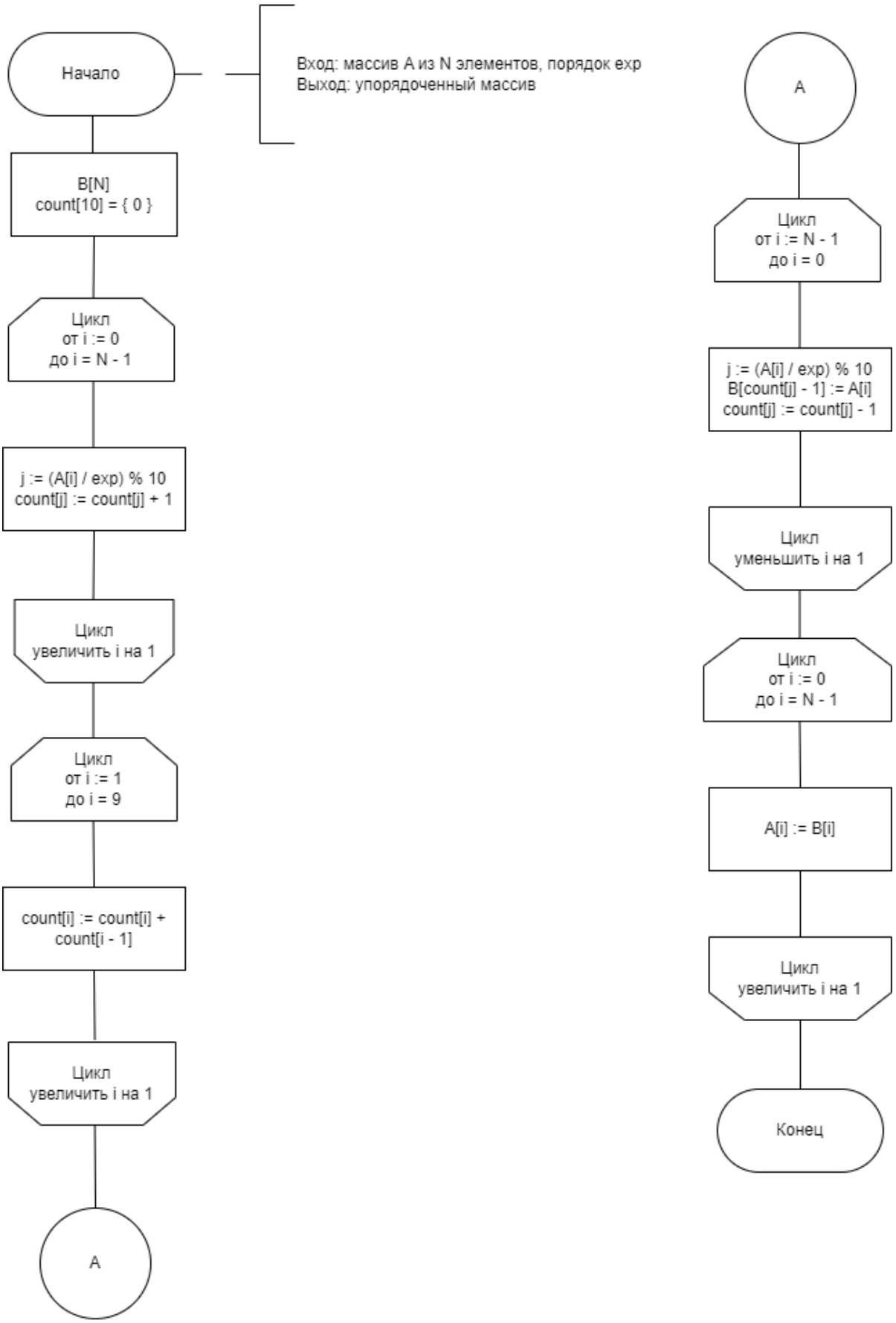

Рисунок 2.6 – Алгоритм сортировки подсчетом

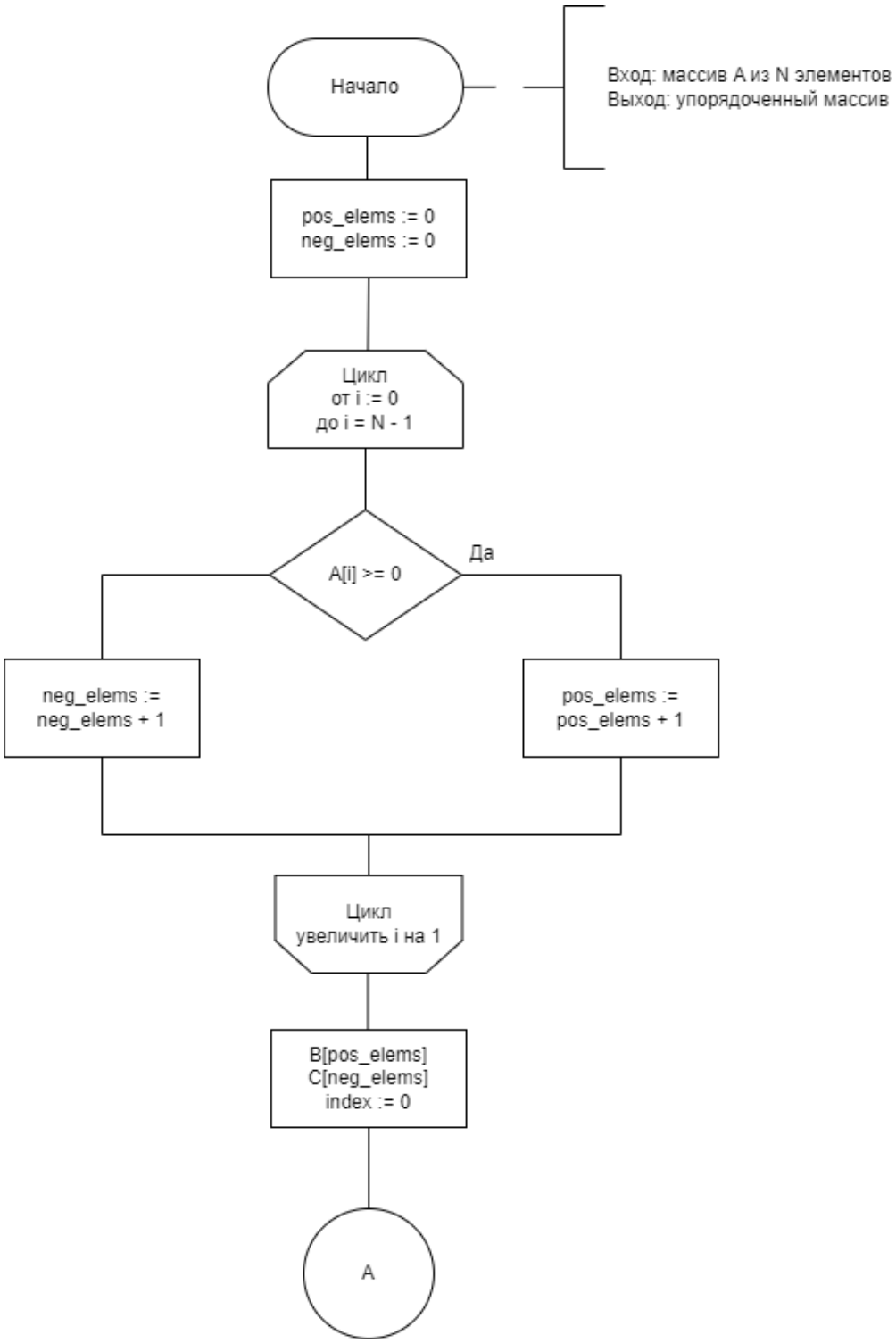

Рисунок 2.7 – Алгоритм поразрядной сортировки — ч. 1

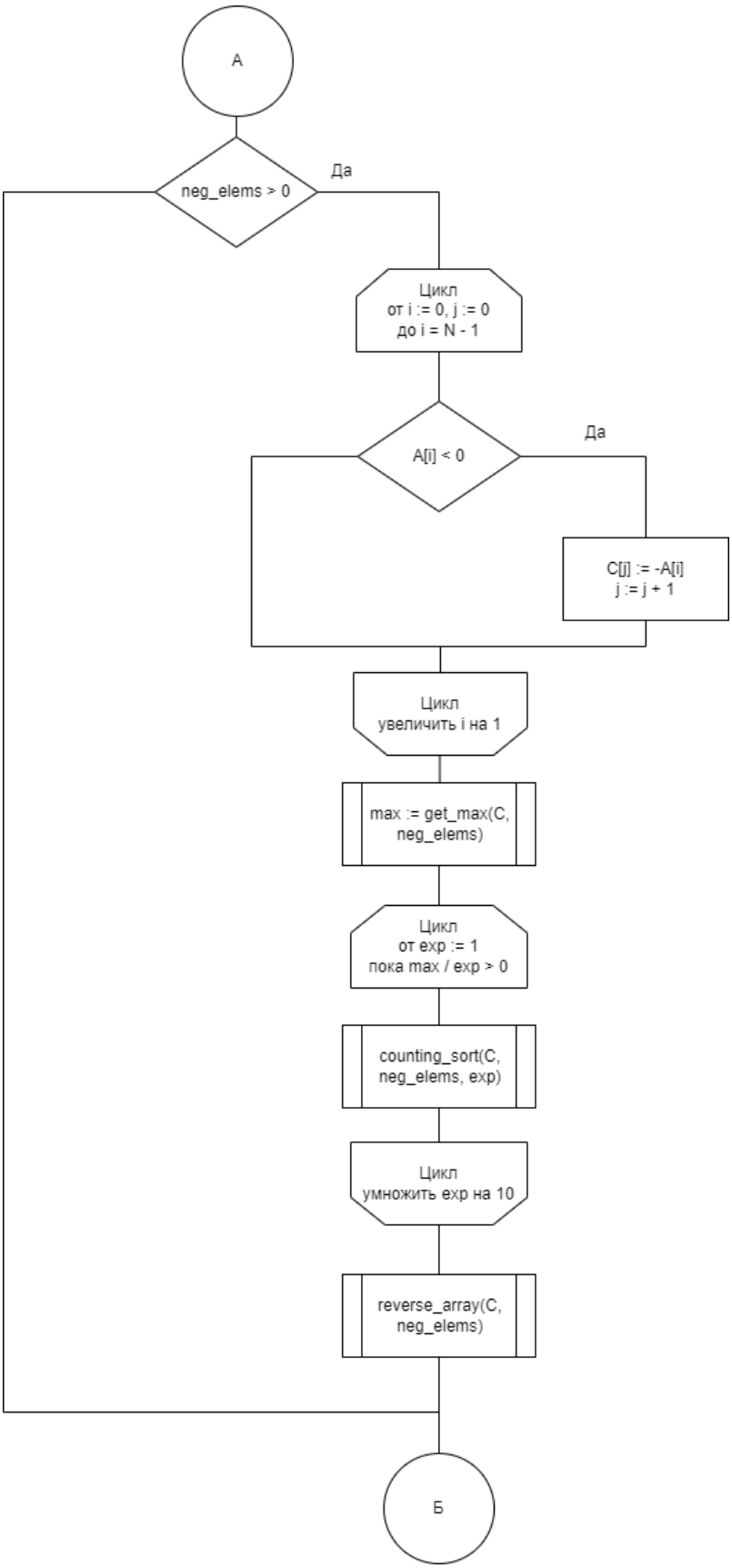

Рисунок 2.8 – Алгоритм поразрядной сортировки — ч. 2

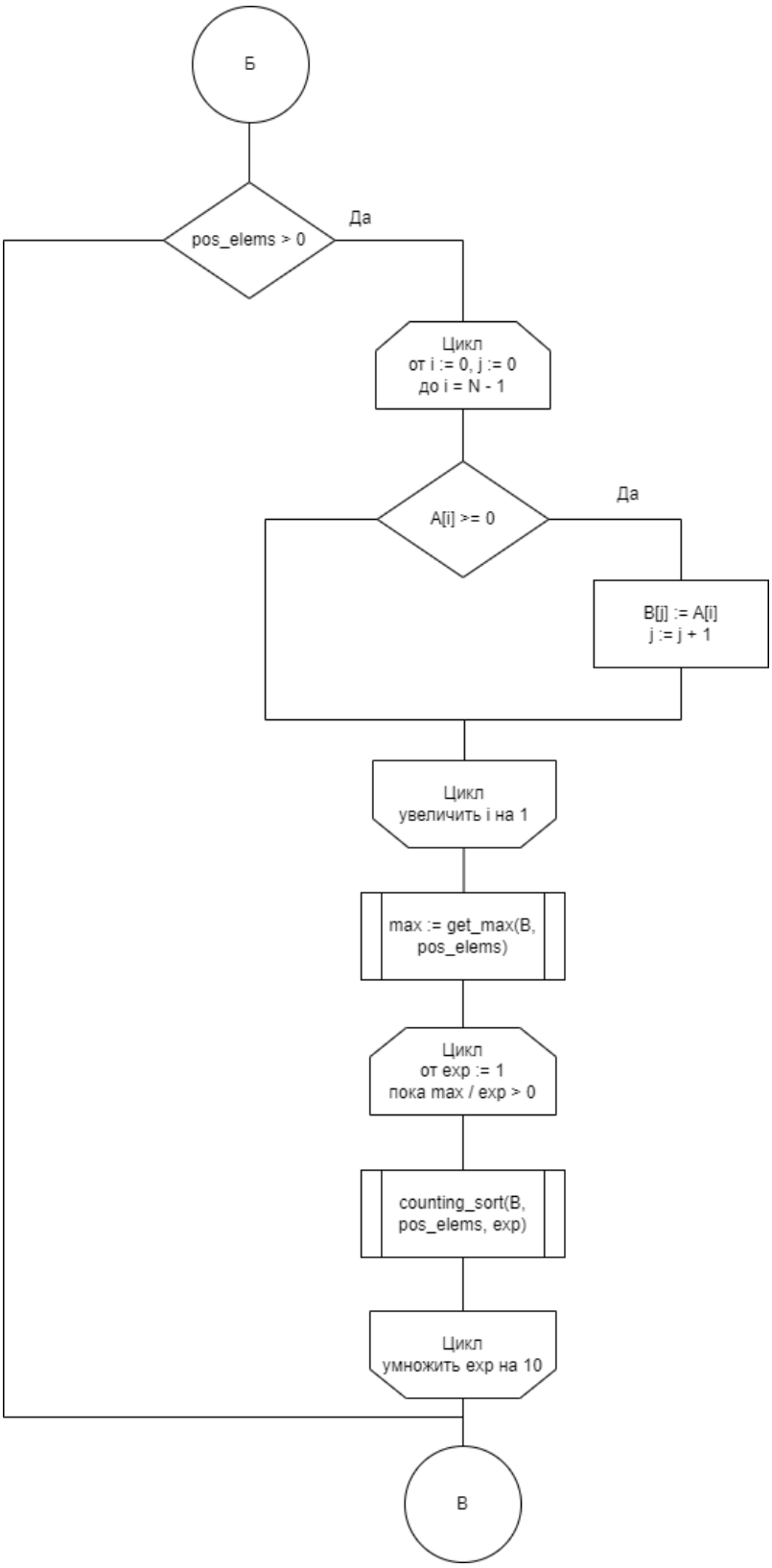

Рисунок 2.9 – Алгоритм поразрядной сортировки — ч. 3

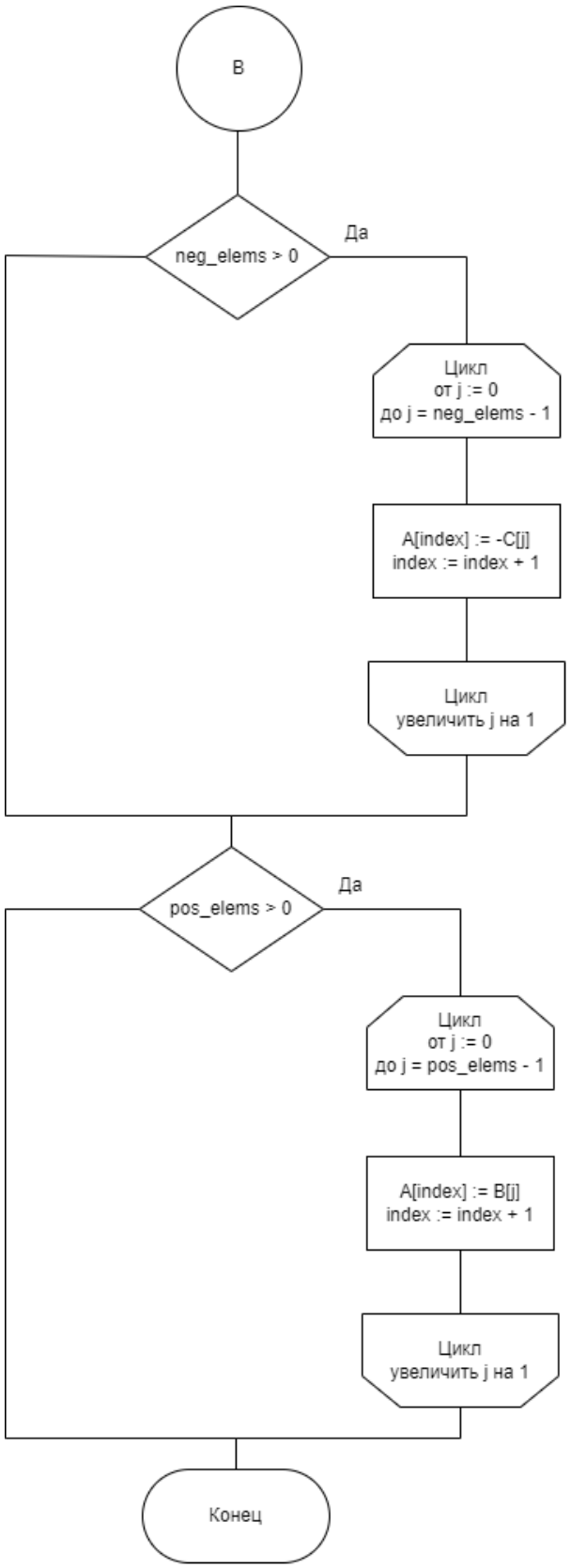

Рисунок 2.10 – Алгоритм поразрядной сортировки — ч. 4

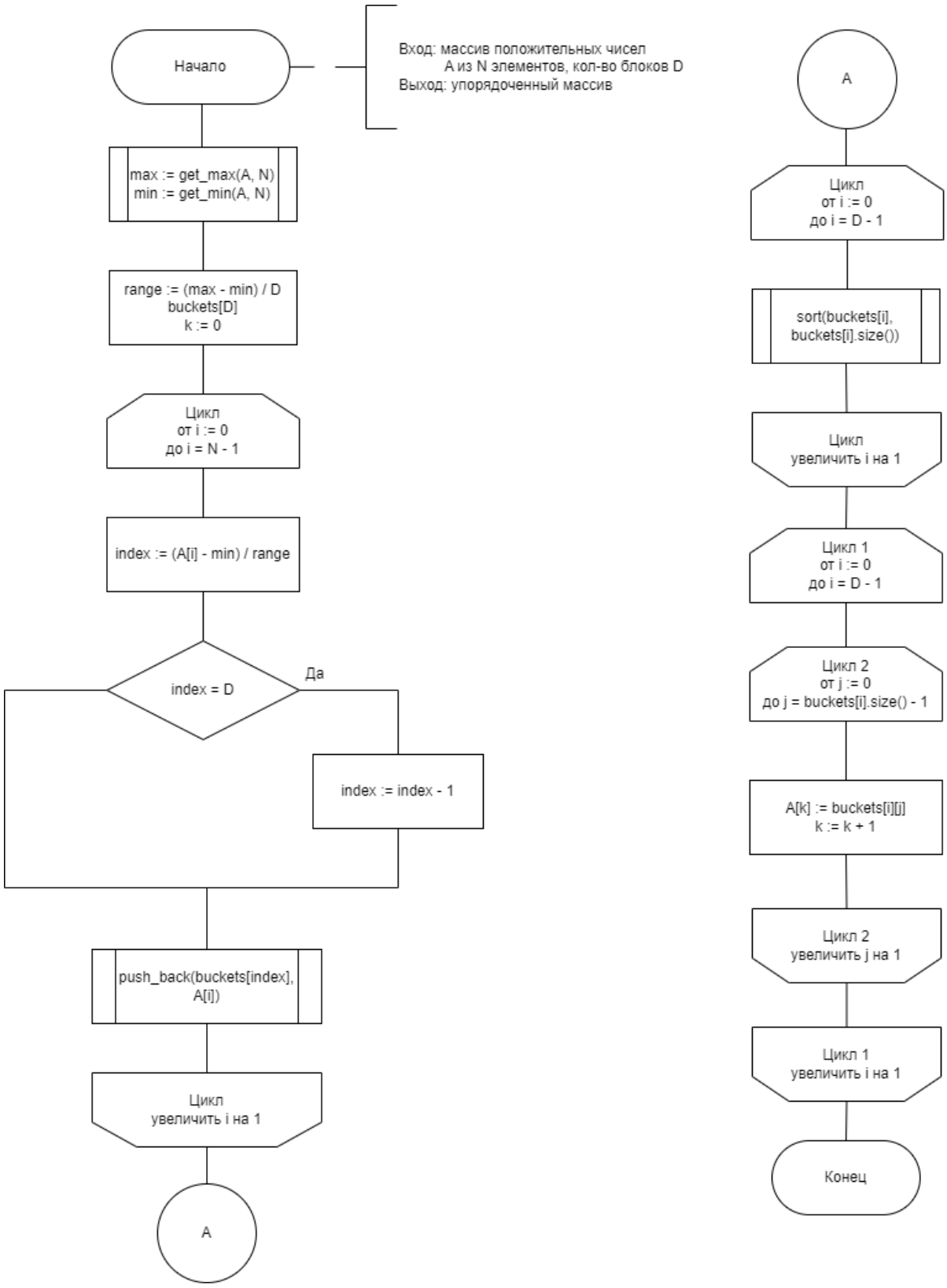

Рисунок 2.11 – Алгоритм блочной сортировки для неотрицательных чисел

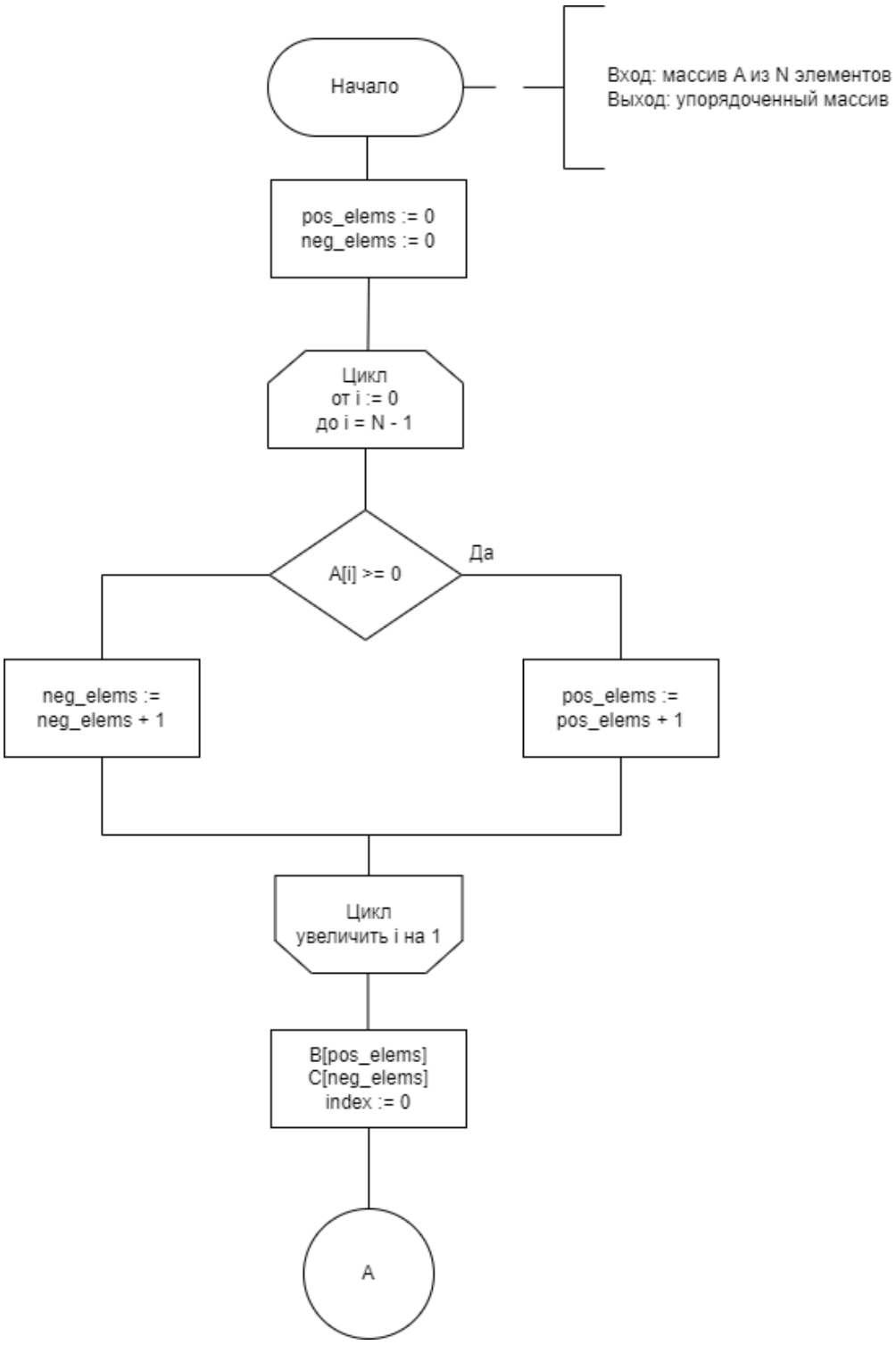

Рисунок 2.12 – Алгоритм блочной сортировки — ч. 1

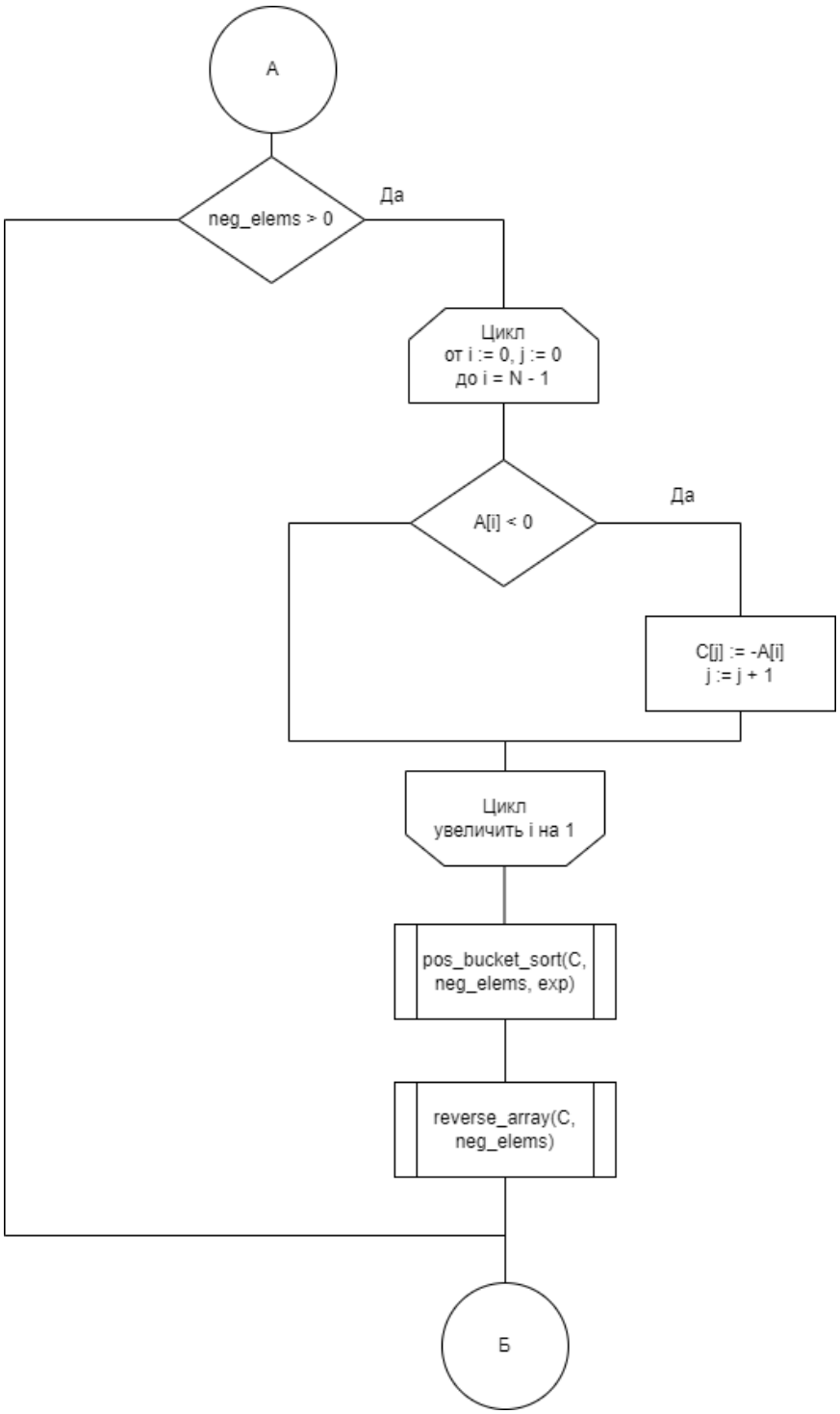

Рисунок 2.13 – Алгоритм блочной сортировки — ч. 2

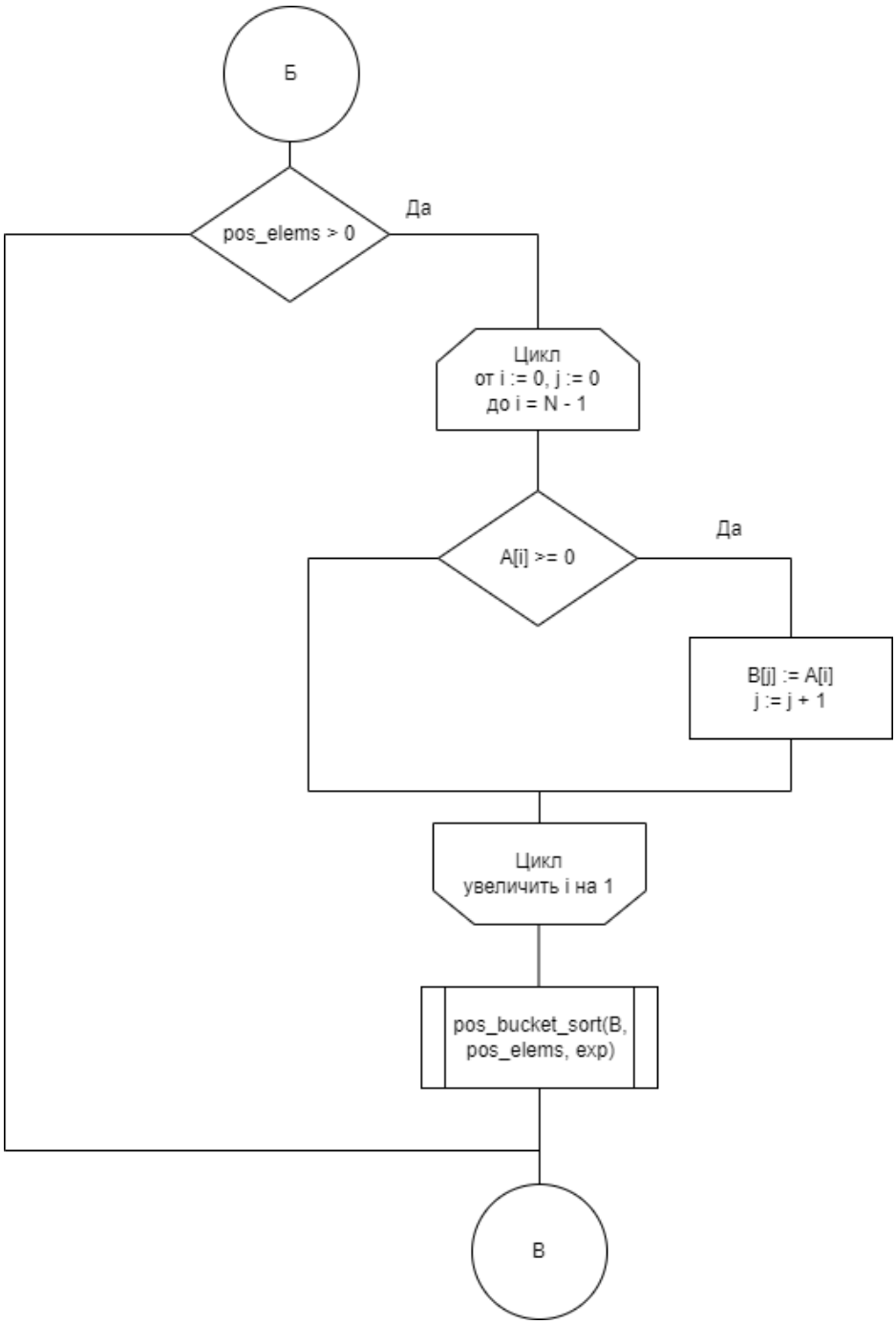

Рисунок 2.14 – Алгоритм блочной сортировки — ч. 3

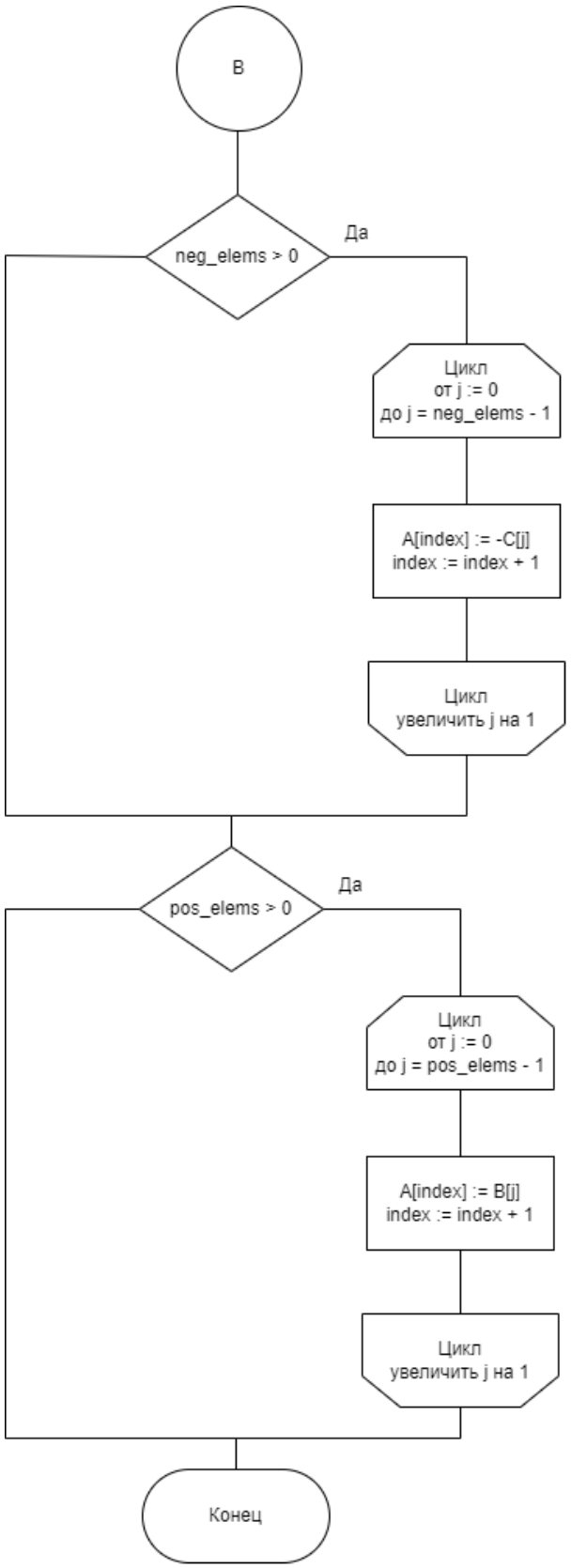

Рисунок 2.15 – Алгоритм блочной сортировки — ч. 4

2.2 Модель вычислений

Обозначим размер массива во всех вычислениях через N.
Для вычисления трудоемкости введена следующая модель вычислений.

1. Операции +, –, *, /, %, ==, !=, <, <=, >, >= [ ], ++, – –, +=, –= имеют трудоемкость 1.

2. Трудоемкость условного оператора рассчитывается следующим образом:

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, \text{ если условие выполняется,} \\ f_B, \text{ иначе.} \end{cases} \tag{2.1}$$

3. Трудоемкость цикла рассчитывается как:

$$f_{loop} = f_{\text{инициализации}} + f_{\text{сравнения}} + N \cdot (f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}). \tag{2.2}$$

4. Трудоемкость вызова функции равна 0.

2.3 Трудоемкость алгоритмов

2.3.1 Алгоритм сортировки перемешиванием

Сравнение для проверки количества элементов в массиве:

$$f_1 = 1. \tag{2.3}$$

Блок инициализации:

$$f_2 = 4. \tag{2.4}$$

Основной цикл:

$$\begin{gathered} f_3 = 1 + N \cdot ((1 + 2 + N \cdot (5 + 4 + 1 + 2) + 1 + 2) + \\ +3 + N \cdot (5 + 4 + 1) + 1)) = \\ = 1 + N \cdot ((6 + 12 \cdot N) + 4 + 10 \cdot N) = \\ = 1 + N \cdot (10 + 22 \cdot N) = \\ = 1 + 10 \cdot N + 22 \cdot N^2. \end{gathered} \tag{2.5}$$

Таким образом, трудоемкость алгоритма сортировки перемешиванием равна:

$$\begin{gathered} f = f_1 + f_2 + f_3 = 1 + 4 + 1 + 10 \cdot N + 22 \cdot N^2 = \\ = 6 + 10 \cdot N + 22 \cdot N^2 \approx O(N^2). \end{gathered} \tag{2.6}$$

2.3.2 Алгоритм поразрядной сортировки

Обозначим через M количество десятичных разрядов наибольшего элемента в массиве.

Сравнение для проверки количества элементов в массиве:

$$f_1 = 1. \tag{2.7}$$

Блок инициализации:

$$f_2 = 2. \tag{2.8}$$

Первый цикл:

$$f_3 = 2 + 4 \cdot N. \tag{2.9}$$

Второй блок инициализации:

$$f_4 = 3. \tag{2.10}$$

Функция, возвращающая массив, элементы которого находятся в порядке, обратном исходному:

$$f_5 = 1 + 2 + 2 + 4 \cdot \frac{N}{4} = 5 + N. \tag{2.11}$$

Цикл для работы с отрицательными элементами входного массива:

$$\begin{gathered} f_6 = 2 + 3 + 8 \cdot N + 1 + 2 + 2 + 6 \cdot (N - 1) + \\ +3 + M \cdot (1 + 2 + 2 + 8 \cdot \frac{N}{2} + 2 + 9 \cdot 7 + 3 + \\ +11 \cdot \frac{N}{2} + 2 + 5 \cdot \frac{N}{2} + 3) + 5 + N = \\ = 12 + 15 \cdot N + M \cdot (78 + 12 \cdot N) = \\ = 12 + 15 \cdot N + 78 \cdot M + 12 \cdot N \cdot M. \end{gathered} \tag{2.12}$$

Цикл для работы с неотрицательными элементами входного массива:

$$\begin{gathered} f_7 = 2 + 3 + 7 \cdot N + 1 + 2 + 2 + 6 \cdot (N - 1) + \\ +3 + M \cdot (1 + 2 + 2 + 8 \cdot \frac{N}{2} + 2 + 9 \cdot 7 + 3 + \\ +11 \cdot \frac{N}{2} + 2 + 5 \cdot \frac{N}{2} + 3) = \\ = 12 + 14 \cdot N + 78 \cdot M + 12 \cdot N \cdot M. \end{gathered} \tag{2.13}$$

Цикл для записи отрицательных элементов в исходный массив:

$$f_8 = 3 + 7 \cdot \frac{N}{2}. \tag{2.14}$$

Цикл для записи неотрицательных элементов в исходный массив:

$$f_9 = 3 + 6 \cdot \frac{N}{2}. \tag{2.15}$$

Таким образом, трудоемкость алгоритма поразрядной сортировки равна:

$$\begin{gathered} f = f_1 + f_2 + f_3 + f_4 + f_5 + f_6 + f_7 + f_8 + f_9 = \\ = 43 + 40.5 \cdot N + 156 \cdot M + 24 \cdot N \cdot M \approx \\ \approx O(N \cdot M). \end{gathered} \tag{2.16}$$

2.3.3 Алгоритм блочной сортировки

Обозначим через K количество сегментов, используемых для сортировки элементов.

Сравнение для проверки количества элементов в массиве:

$$f_1 = 1. \tag{2.17}$$

Блок инициализации:

$$f_2 = 2. \tag{2.18}$$

Подсчет количества отрицательных и неотрицательных элементов в исходном массиве:

$$f_3 = 2 + 5 \cdot N. \tag{2.19}$$

Второй блок инициализации:

$$f_4 = 3. \tag{2.20}$$

Функция для поиска наибольшего элемента в массиве:

$$f_5 = 2 + 2 + 6 \cdot (\frac{N}{2} - 1) = 4 + 6 \cdot \frac{N}{2} - 6 = 3 \cdot N - 2. \tag{2.21}$$

Функция для поиска наименьшего элемента в массиве:

$$f_6 = 2 + 2 + 6 \cdot (\frac{N}{2} - 1) = 4 + 6 \cdot \frac{N}{2} - 6 = 3 \cdot N - 2. \tag{2.22}$$

Функция, возвращающая массив, элементы которого находятся в порядке, обратном исходному:

$$f_7 = 3 + 2 + 6 \cdot \frac{N}{4} = 5 + 3 \cdot \frac{N}{2} = 5 + 1.5 \cdot N. \tag{2.23}$$

Функция, реализующая алгоритм сортировки перемешиванием (значение получено в пункте 2.3.1):

$$f_8 = N^2. \tag{2.24}$$

Функция для сортировки массива, состоящего из неотрицательных элементов:

$$\begin{gathered} f_9 = 14 + f_5 + f_6 + 9 \cdot \frac{N}{2} + (5 + f_8) \cdot K + (4 + 9 \cdot \frac{N}{2 \cdot K}) \cdot K = \\ = 14 + 3 \cdot N - 2 + 3 \cdot N - 2 + 4.5 \cdot N + 5 \cdot K + \\ + 18 \cdot K - 4 \cdot N \cdot K + 2 \cdot N^2 \cdot K + 4 \cdot K + 4.5 \cdot N = \\ = 22 \cdot K + 4.5 \cdot N - 4 \cdot N \cdot K + 2 \cdot N^2 \cdot K. \end{gathered} \tag{2.25}$$

Сортировка массива, состоящего из отрицательных элементов:

$$f_{10} = 6 + 9 \cdot \frac{N}{2} + f_9 + f_7 = 6 + 4.5 \cdot N + f_7 + f_9. \tag{2.26}$$

Сортировка массива, состоящего из неотрицательных элементов:

$$f_{11} = 6 + 8 \cdot \frac{N}{2} + f_9 = 6 + 4 \cdot N + f_9. \tag{2.27}$$

Цикл для записи отрицательных элементов в исходный массив:

$$f_{12} = 3 + 7 \cdot \frac{N}{2} = 3 + 3.5 \cdot N. \tag{2.28}$$

Цикл для записи неотрицательных элементов в исходный массив:

$$f_{13} = 3 + 6 \cdot \frac{N}{2} = 3 + 3 \cdot N. \tag{2.29}$$

Таким образом, трудоемкость алгоритма блочной сортировки равна:

$$\begin{gathered} f = f_1 + f_2 + f_3 + f_4 + f_{10} + f_{11} + f_{12} + f_{13} = \\ = 8 + 5 \cdot N + 6 + 4.5 \cdot N + f_7 + f_9 + 6 + 4 \cdot N + \\ + f_9 + 3 + 3.5 \cdot N + 3 + 3 \cdot N = \\ = 26 + 20 \cdot N + f_7 + 2 \cdot f_9 = \\ = 26 + 20 \cdot N + 5 + 1.5 \cdot N + 2 \cdot (22 \cdot K + \\ + 4.5 \cdot N - 4 \cdot N \cdot K + 2 \cdot N^2 \cdot K) = \\ = 75 + 30.5 \cdot N - 8 \cdot N \cdot K + 2 \cdot N^2 \cdot K \approx \\ \approx O(N^2 \cdot K). \end{gathered} \tag{2.30}$$

Вывод

В текущем разделе на основе теоретических данных, полученных из аналитического раздела, были построены схемы трех алгоритмов сортировки, а также оценены их трудоемкости.

3 Технологическая часть

В текущем разделе приведены средства реализации алгоритмов сортировки и листинги кода.

3.1 Требования к программному обеспечению

Программа должна запрашивать размер N массива A, а затем сами элементы вектора (все элементы должны быть целыми числами). Далее программа должна запросить у пользователя порядковый номер одного из трех алгоритмов сортировки. После этого массив должен быть упорядочен по возрастанию с помощью выбранного алгоритма сортировки и выведен на экран.

Программа должна обрабатывать ошибки (например, попытку ввода отрицательного размера массива) и завершать работу с выводом информации об ошибке на экран.

3.2 Средства реализации

Для реализации программного обеспечения был выбран язык **C++** ввиду следующих причин:

1) в библиотеке стандартных шаблонов имеется контейнер **std::vector**;
2) для **std::vector** реализован метод **size()**, который возвращает размер вектора;
3) для считывания данных и вывода их на экран в стандартной библиотеке существуют соответственно функции **scanf()** и **printf()**.

Таким образом, с помощью языка **C++** можно реализовать программное обеспечение, которое соответствует перечисленным выше требованиям.

3.3 Реализация алгоритмов

3.3.1 Вспомогательные функции

В листинге 3.1 показана реализация вспомогательных функций, которые нужны для реализации алгоритмов сортировки.

Листинг 3.1 – Вспомогательные функции для реализации алгоритмов сортировки

```
static void reverse_array(std::vector<int> &array, const
    std::size_t size)
{
    if (size < 2)
        return;
    std::size_t last_elem_index = size / 2;
    for (std::size_t i = 0; i < last_elem_index; i++)
        std::swap(array[i], array[size - i - 1]);
}

static int get_max(const std::vector<int> &array, const
    std::size_t size)
{
    int max = array[0];
    for (std::size_t i = 1; i < size; i++)
        if (array[i] > max)
            max = array[i];
    return max;
}

static int get_min(const std::vector<int> &array, const
    std::size_t size)
{
    int min = array[0];
    for (std::size_t i = 1; i < size; i++)
        if (array[i] < min)
            min = array[i];
    return min;
}
```

3.3.2 Алгоритм сортировки перемешиванием

В листинге 3.2 показана реализация алгоритма сортировки перемешиванием.

Листинг 3.2 – Реализация сортировки перемешиванием

```
void cocktail_sort(std::vector<int> &array, const std::size_t size)
{
```

```
3       if (size < 2)
4           return;
5       bool flag = true;
6       int start = 0, end = size - 1;
7       while (flag)
8       {
9           flag = false;
10          for (int i = start; i < end; ++i)
11          {
12              if (array[i] > array[i + 1])
13              {
14                  std::swap(array[i], array[i + 1]);
15                  flag = true;
16              }
17          }
18          if (!flag)
19              break;
20          flag = false;
21          --end;
22          for (int i = end - 1; i >= start; --i)
23          {
24              if (array[i] > array[i + 1])
25              {
26                  std::swap(array[i], array[i + 1]);
27                  flag = true;
28              }
29          }
30          ++start;
31      }
32  }
```

3.3.3 Алгоритм поразрядной сортировки

В листинге 3.3 показана реализация алгоритма поразрядной сортировки с использованием сортировки подсчетом.

Листинг 3.3 – Реализация поразрядной сортировки

```
1  static void counting_sort(std::vector<int> &array, const
       std::size_t size, int exp)
```

```
{
    if (size < 2)
        return;
    std::vector<int> output_array(size, 0);
    std::vector<int> count(10, 0);
    for (std::size_t i = 0; i < size; i++)
    {
        int index = (array[i] / exp) % 10;
        count[index]++;
    }
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];
    for (int i = size - 1; i >= 0; i--)
    {
        int index = (array[i] / exp) % 10;
        output_array[count[index] - 1] = array[i];
        count[index]--;
    }
    for (std::size_t i = 0; i < size; i++)
        array[i] = output_array[i];
}

void radix_sort(std::vector<int> &array, const std::size_t size)
{
    if (size < 2)
        return;
    int pos_elements_count = 0, neg_elements_count = 0;
    for (std::size_t i = 0; i < size; i++)
    {
        if (array[i] >= 0)
            pos_elements_count++;
        else
            neg_elements_count++;
    }
    std::vector<int> pos_array;
    std::vector<int> neg_array;
    int index = 0;
    if (neg_elements_count > 0)
    {
```

```
41          for (std::size_t i = 0; i < size; i++)
42              if (array[i] < 0)
43                  neg_array.push_back(-array[i]);
44          int max = get_max(neg_array, neg_array.size());
45          for (int exp = 1; max / exp > 0; exp *= 10)
46              counting_sort(neg_array, neg_array.size(), exp);
47          reverse_array(neg_array, neg_array.size());
48      }
49      if (pos_elements_count > 0)
50      {
51          for (std::size_t i = 0; i < size; i++)
52              if (array[i] >= 0)
53                  pos_array.push_back(array[i]);
54          int max = get_max(pos_array, pos_array.size());
55          for (int exp = 1; max / exp > 0; exp *= 10)
56              counting_sort(pos_array, pos_array.size(), exp);
57      }
58      if (neg_elements_count > 0)
59      {
60          for (std::size_t i = 0; i < neg_array.size(); i++)
61          {
62              array[index] = -neg_array[i];
63              index++;
64          }
65      }
66      if (pos_elements_count > 0)
67      {
68          for (std::size_t i = 0; i < pos_array.size(); i++)
69          {
70              array[index] = pos_array[i];
71              index++;
72          }
73      }
74  }
```

3.3.4 Алгоритм блочной сортировки

В листинге 3.4 показана реализация алгоритма блочной сортировки с использованием сортировки перемешиванием.

Листинг 3.4 – Реализация блочной сортировки

```
static void pos_bucket_sort(std::vector<int> &array, const
   std::size_t size)
{
    if (size < 2)
        return;
    double max = get_max(array, size);
    double min = get_min(array, size);
    int range = std::ceil((max - min) / NUM_OF_BUCKETS);
    std::vector<int> buckets[NUM_OF_BUCKETS];
    for (std::size_t i = 0; i < size; i++)
    {
        int bucket_index = std::floor((array[i] - min) / range);
        if (bucket_index == NUM_OF_BUCKETS)
            bucket_index--;
        buckets[bucket_index].push_back(array[i]);
    }
    for (int i = 0; i < NUM_OF_BUCKETS; i++)
        cocktail_sort(buckets[i], buckets[i].size());
    int index = 0;
    for (int i = 0; i < NUM_OF_BUCKETS; i++)
    {
        for (std::size_t j = 0; j < buckets[i].size(); j++)
        {
            array[index] = buckets[i][j];
            index++;
        }
    }
}

void bucket_sort(std::vector<int> &array, const std::size_t size)
{
    if (size < 2)
        return;
```

```
    int pos_elements_count = 0, neg_elements_count = 0;
    for (std::size_t i = 0; i < size; i++)
    {
        if (array[i] >= 0)
            pos_elements_count++;
        else
            neg_elements_count++;
    }
    std::vector<int> pos_array;
    std::vector<int> neg_array;
    int index = 0;
    if (neg_elements_count > 0)
    {
        for (std::size_t i = 0; i < size; i++)
            if (array[i] < 0)
                neg_array.push_back(-array[i]);
        pos_bucket_sort(neg_array, neg_array.size());
        reverse_array(neg_array, neg_array.size());
    }
    if (pos_elements_count > 0)
    {
        for (std::size_t i = 0; i < size; i++)
            if (array[i] >= 0)
                pos_array.push_back(array[i]);
        pos_bucket_sort(pos_array, pos_array.size());
    }
    if (neg_elements_count > 0)
    {
        for (std::size_t i = 0; i < neg_array.size(); i++)
        {
            array[index] = -neg_array[i];
            index++;
        }
    }
    if (pos_elements_count > 0)
    {
        for (std::size_t i = 0; i < pos_array.size(); i++)
        {
            array[index] = pos_array[i];
```

```
72                 index++;
73             }
74         }
75 }
```

3.4 Тестовые данные

В таблице 3.1 приведены тестовые данные и их описание для трех функций, реализующих алгоритмы сортировки. Тесты выполнялись по методологии черного ящика (модульное тестирование). Все тесты пройдены успешно.

Таблица 3.1 – Тестовые данные

Описание теста	**Входной массив**	**Ожидаемый результат**	**Выходной масив**
Пустой массив	NULL	NULL	NULL
Один отрицательный элемент	-25	-25	-25
Один положительный элемент	76	76	76
Упорядоченный по возрастанию массив	-282, -50, 32, 54, 76, 108	-282, -50, 32, 54, 76, 108	-282, -50, 32, 54, 76, 108
Упорядоченный по убыванию массив	982, 654, 54, 3, -19, -89, -320	-320, -89, -19, 3, 54, 654, 982	-320, -89, -19, 3, 54, 654, 982
Случайно упорядоченный массив	10, -20, 32, -89, -76, -10, 89, 35, 197	-89, -76, -20, -10, 10, 32, 35, 89, 197	-89, -76, -20, -10, 10, 32, 35, 89, 197

Вывод

В данном разделе был написан исходный код трех алгоритмов сортировки — перемешиванием, поразрядной и блочной. Описаны тесты и приведены результаты тестирования.

4 Исследовательская часть

4.1 Технические характеристики устройства

Технические характеристики устройства, на котором было проведено измерение времени работы алгоритмов сортировки:

1) операционная система Windows 10 Pro x64;
2) оперативная память 16 ГБ;
3) процессор Intel® Core™ i7-4790K @ 4.00 ГГц.

4.2 Время работы алгоритмов

В таблицах 4.1–4.3 приведено время работы в тиках трех функций, реализующих алгоритмы сортировки при различных размерах входного массива. На рисунках 4.1–4.3 изображены зависимости времени работы в тиках алгоритмов сортировки от размера входного массива.

Время работы функций замерены с помощью ассемблерной инструкции rdtsc, которая читает счетчик Time Stamp Counter и возвращает 64-битное количество тиков с момента последнего сброса процессора.

Таблица 4.1 – Время работы функций, реализующих алгоритмы сортировки, в зависимости от размеров входного упорядоченного по возрастанию массива

Размер массива (элементы)	**Блочная сортировка (тики)**	**Поразрядная сортировка (тики)**	**Сортировка перемешиванием (тики)**
10	30126	9762	469
50	43202	17131	842
100	44411	18125	886
250	76321	59982	2187
500	113912	110664	3974

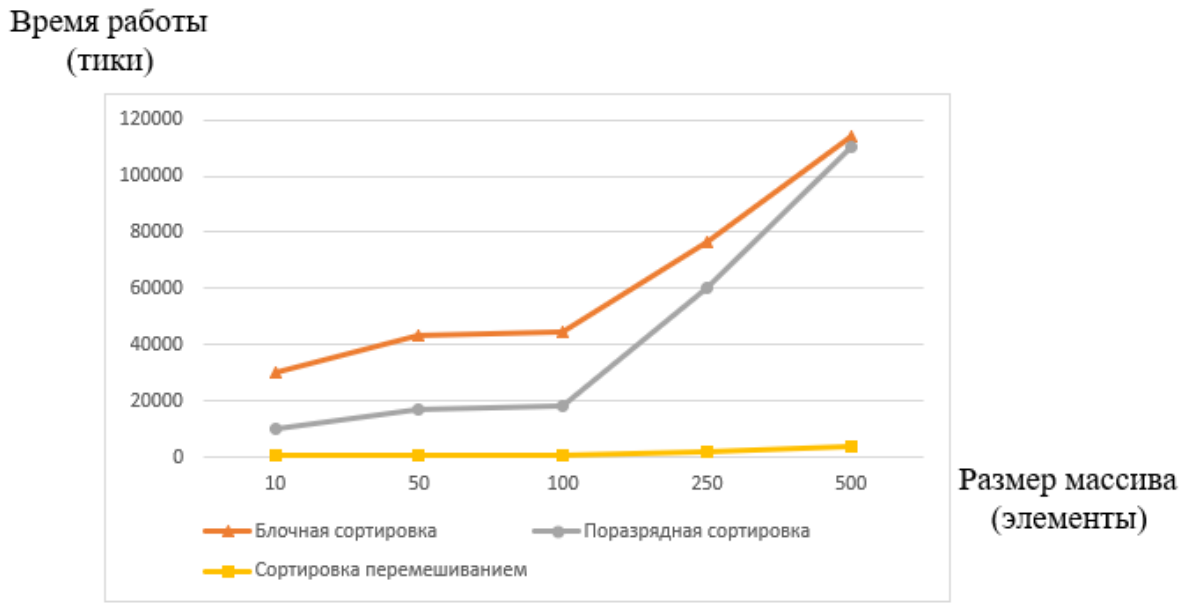

Рисунок 4.1 – Зависимость времени работы функций, реализующих алгоритмы сортировки, в тиках от размеров входного упорядоченного по возрастанию массива

Таблица 4.2 – Время работы функций, реализующих алгоритмы сортировки, в зависимости от размеров входного упорядоченного по убыванию массива

Размер массива (элементы)	**Блочная сортировка (тики)**	**Поразрядная сортировка (тики)**	**Сортировка перемешиванием (тики)**
10	34547	9886	38376
50	64888	16952	145939
100	64895	18070	147250
250	220321	56900	1021819
500	1844674	109577	3575462

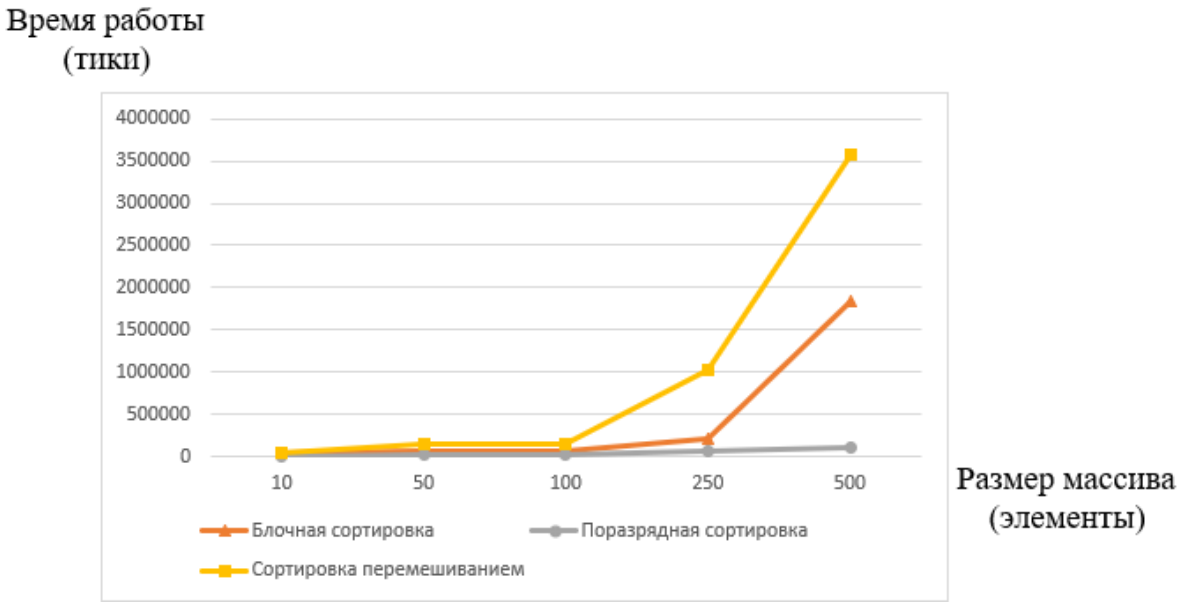

Рисунок 4.2 – Зависимость времени работы функций, реализующих алгоритмы сортировки, в тиках от размеров входного упорядоченного по убыванию массива

Таблица 4.3 – Время работы функций, реализующих алгоритмы сортировки, в зависимости от размеров входного случайно упорядоченного массива

Размер массива (элементы)	**Блочная сортировка (тики)**	**Поразрядная сортировка (тики)**	**Сортировка перемешиванием (тики)**
10	36580	48053	31330
50	61390	86500	109334
100	62672	88012	109417
250	156227	200412	612988
500	428196	396996	2410021

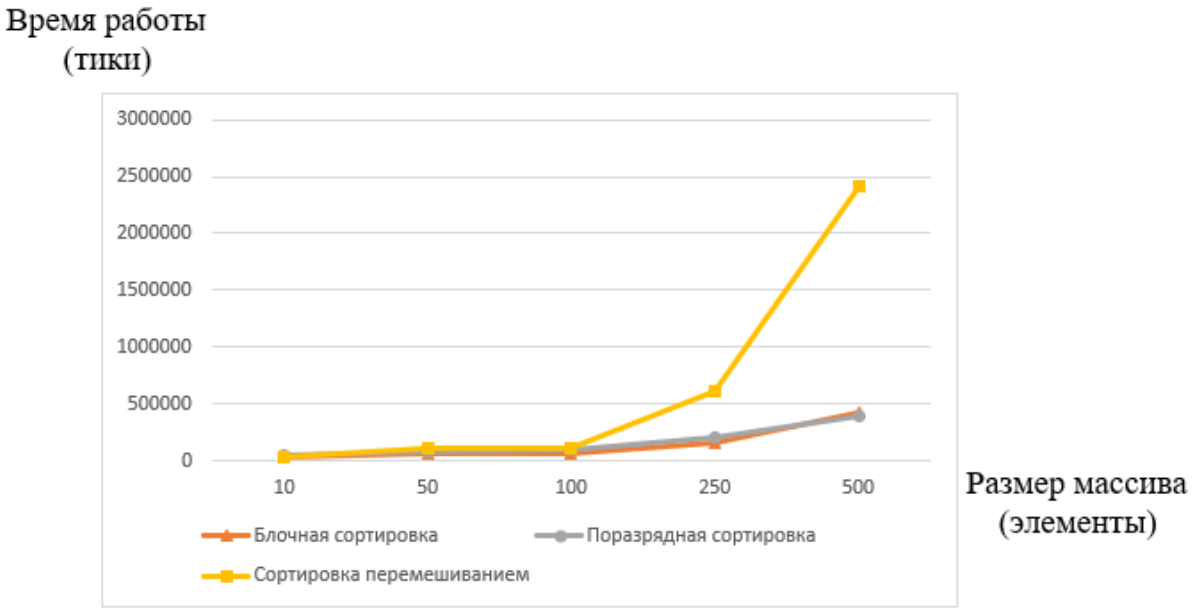

Рисунок 4.3 – Зависимость времени работы функций, реализующих алгоритмы сортировки, в тиках от размеров входного случайно упорядоченного массива

Вывод

Сортировка перемешиванием наименее затратна по времени, если на вход подается упорядоченный по возрастанию вектор. Если же массив заполнен случайными числами, то время работы сортировки перемешиванием растет значительно быстрее с увеличением размера вектора. В этом случае наиболее эффективно себя показывают блочная и подразрядная сортировки. Если вектор упорядочен по убыванию, быстрее других алгоритмов работает поразрядная сортировка.

Согласно полученным при проведении эксперимента данным, наиболее эффективной можно считать поразрядную сортировку, которая показала хорошие результаты относительно других алгоритмов во всех трех случаях. К тому же, она является наиболее эффективной в случае, когда на вход подается упорядоченный по убыванию массив.

Заключение

В рамках данной лабораторной работы была достигнута поставленная цель: были получены навыки оценки трудоемкости и временной эффективности на материале алгоритмов сортировки.

Решены все поставленные задачи:

1) были изучены и реализованы три алгоритма сортировки — перемешиванием, поразрядная и блочная;

2) был проведен сравнительный анализ трудоемкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;

3) был проведен сравнительный анализ алгоритмов на основе экспериментальных данных.

Согласно полученным при проведении эксперимента данным, наиболее эффективной является поразрядная сортировка, которая показала хорошие результаты относительно других алгоритмов во всех трех случаях.

Список использованных источников

1. *Вирт Н.* Алгоритмы и структуры данных // М.: Мир. — 1989. — С. 360.
2. *Кормен Т.*, *Лейзерсон Ч.* Алгоритмы: построение и анализ // Вильямс. — 2011. — С. 46—47.
3. *R.Sinha*, *Zobel J.* Efficient Trie-Based Sorting of Large Sets of Strings. — 2003.