Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Вятский государственный университет»

Колледж ВятГУ

**ОТЧЕТ**

**ПО ДОМАШНЕЙ КОНТРОЛЬНОЙ РАБОТЕ №5**

**«Исследование алгоритмов сортировки»**

**ПО ДИСЦИПЛИНЕ «Основы алгоритмизации и программирование»**

Выполнил: студент учебной группы

ИСПк-203-52-00

Четвериков Д. А.

Преподаватель:

Сергеева Елизавета Григорьевна

Киров

2024

**1. Цель работы:** получить базовые сведения о наиболее известных алгоритмах сортировки, изучить принципы работы с текстовыми файлами.

**2. Формулировка задания:** Вариант 20

Задание:

1. Реализовать сортировку данных с помощью алгоритма подсчетом.

2. Реализовать сортировку данных с помощью пирамидального алгоритма.

3. В обоих случаях необходимо предусмотреть возможность изменения компаратора (реализация компаратора в виде передаваемой в подпрограмму функции).

4. Считывание и вывод данных необходимо производить из текстового файла.

5. Для демонстрации работы программных реализаций самостоятельно подготовить варианты входных данных (при этом объем тестовых файлов должен позволять оценить скорость работы программ).

**Описание алгоритма:**

**▎1. Процедура CountingSort**

Описание:

Сортировка подсчетом — это не сравнивающий алгоритм сортировки, который работает за линейное время. Он полезен, когда диапазон значений ограничен.

Алгоритм:

1. Проверка на пустой массив: Если количество элементов (count) равно 0, процедура завершает выполнение.

2. Поиск минимального и максимального значения:

• Инициализируем MinValue и MaxValue первым элементом массива.

• Проходим по массиву data, обновляя MinValue и MaxValue, если находим элементы меньше или больше текущих значений.

3. Определение диапазона значений:

• Рассчитываем диапазон как Range = MaxValue - MinValue + 1.

4. Инициализация массива подсчета:

• Создаем массив CountArray размером Range и инициализируем все его элементы нулями.

5. Подсчет вхождений каждого элемента:

• Проходим по массиву data и для каждого элемента увеличиваем соответствующий элемент в CountArray. Индекс в CountArray определяется как data[i] - MinValue.

6. Заполнение исходного массива отсортированными элементами:

• Инициализируем индекс j для записи отсортированных значений.

• Проходим по CountArray, и для каждого индекса, где значение больше 0, записываем соответствующее значение в data и уменьшаем счетчик в CountArray.

**▎2. Процедура Heapify**

Описание:

Процедура Heapify поддерживает свойства кучи (max-heap) для указанного узла.

Алгоритм:

1. Инициализируем переменную largest как текущий индекс i.

2. Определяем индексы левого (left) и правого (right) дочерних узлов.

3. Сравниваем текущий элемент с левым и правым дочерними узлами с помощью функции сравнения:

• Если левый дочерний узел больше текущего, обновляем largest.

• Если правый дочерний узел больше текущего (или левого), также обновляем largest.

4. Если largest изменился (то есть один из дочерних узлов больше текущего), меняем местами элементы на позициях i и largest, а затем рекурсивно вызываем Heapify для поддерева.

**▎3. Процедура HeapSort**

Описание:

Пирамидальная сортировка (Heap Sort) строит кучу из данных и затем сортирует их.

Алгоритм:

1. Строим кучу, начиная с последнего родительского узла и двигаясь вверх к корню:

• Для каждого узла вызываем процедуру Heapify.

2. Извлекаем элементы из кучи:

• Меняем местами корень (максимальный элемент) с последним элементом массива.

• Уменьшаем размер кучи и восстанавливаем свойства кучи для корня, вызывая Heapify.

**▎4. Функция CompareAscending**

Описание:

Эта функция определяет порядок сортировки (возрастающий).

Алгоритм:

• Возвращает разницу между двух целых чисел, что позволяет использовать ее для сравнения в других процедурах.

**▎5. Процедура ReadDataFromFile**

Описание:

Считывает данные из файла и загружает их в массив.

Алгоритм:

1. Открывает файл для чтения.

2. Инициализирует счетчик элементов (count) как 0.

3. В цикле считывает числа из файла до конца файла (Eof):

• Каждое считанное число добавляется в массив data, а счетчик увеличивается.

• Увеличиваем размер массива на 1 для следующего числа.

4. Закрывает файл после завершения чтения.

**▎6. Процедура WriteDataToFile**

Описание:

Записывает отсортированные данные в файл.

Алгоритм:

1. Открывает файл для записи.

2. В цикле записывает каждый элемент массива в файл.

3. Закрывает файл после завершения записи.

**▎7. Основная программа**

Описание:

Главная часть программы, которая связывает все процедуры и функции.

Алгоритм:

1. Инициализирует массивы для входных данных, данных для сортировки подсчетом и данных для пирамидальной сортировки.

2. Считывает данные из файла с помощью ReadDataFromFile.

3. Запускает сортировку подсчетом:

• Измеряет время выполнения перед и после вызова CountingSort.

• Записывает отсортированные данные в файл через WriteDataToFile.

4. Запускает пирамидальную сортировку:

• Аналогично измеряет время выполнения перед и после вызова HeapSort.

• Записывает отсортированные данные в другой файл.

5. Вычисляет время выполнения каждой сортировки и выводит его на экран.

Схема алгоритма:

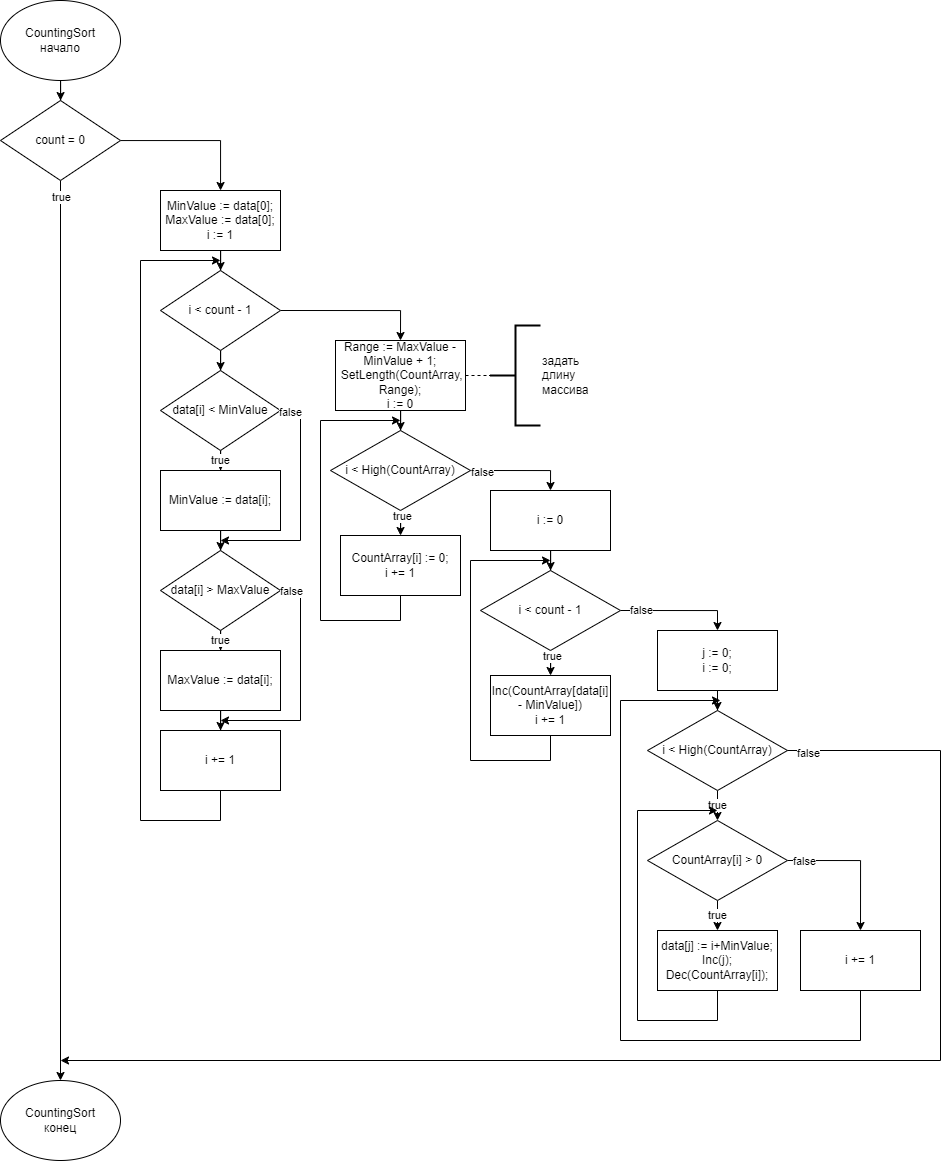


Рисунок 1 – схема алгоритма для функции Counting sort

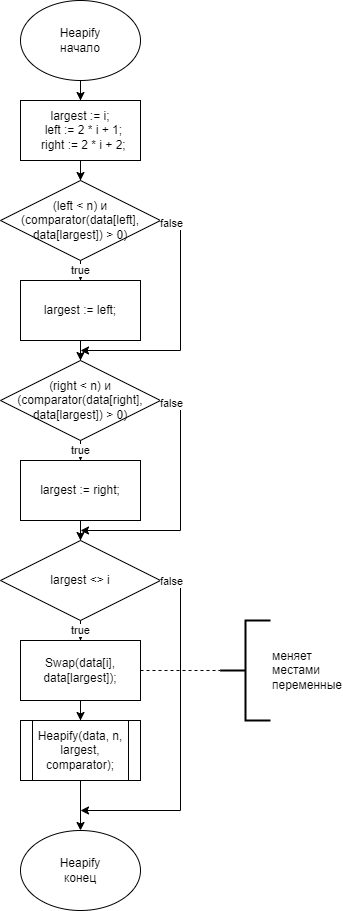


Рисунок 2 – схема алгоритма для функции Heapify

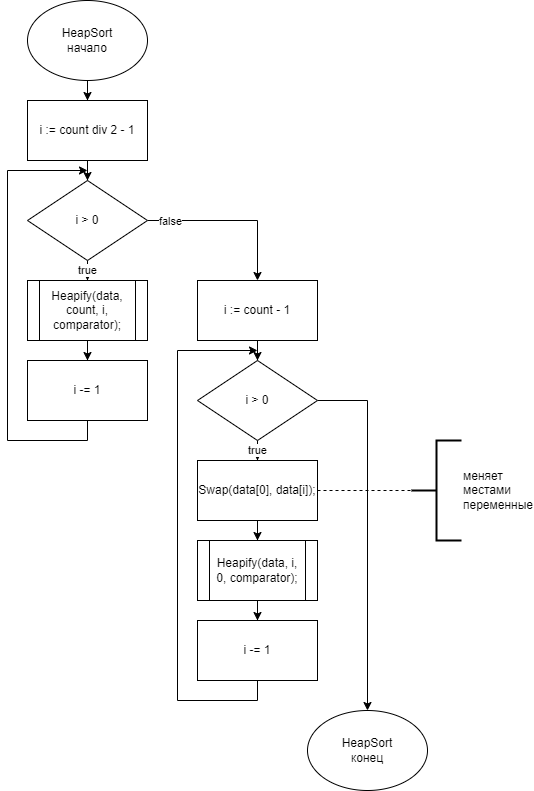


Рисунок 3 – схема алгоритма для функции HeapSort

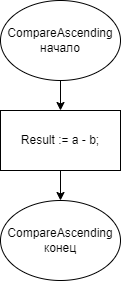


Рисунок 4 – схема алгоритма для функции CompareAscending

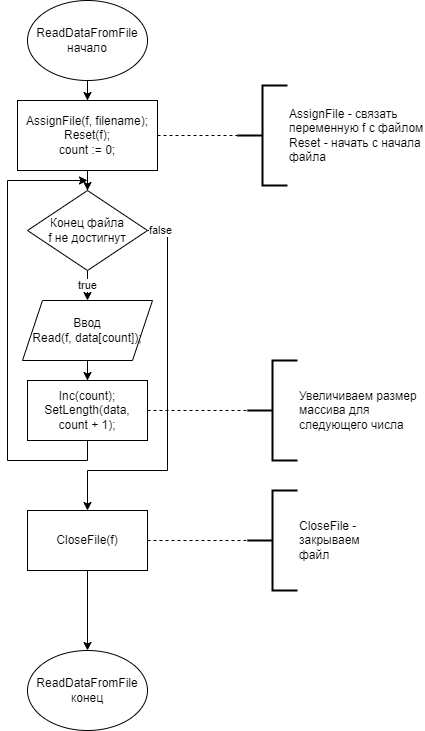


Рисунок 5 – схема алгоритма для функции ReadDataFromFile

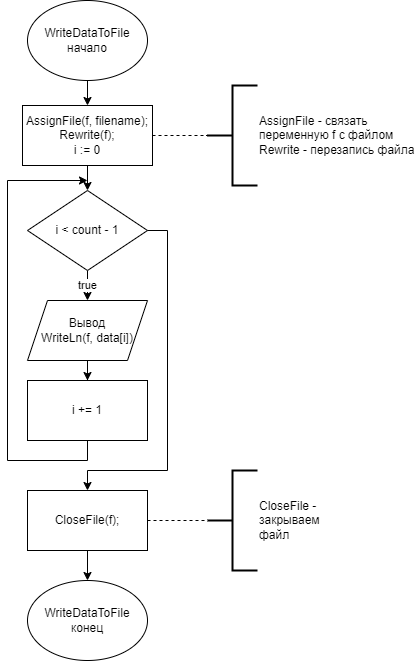


Рисунок 5 – схема алгоритма для функции WriteDataFromFile

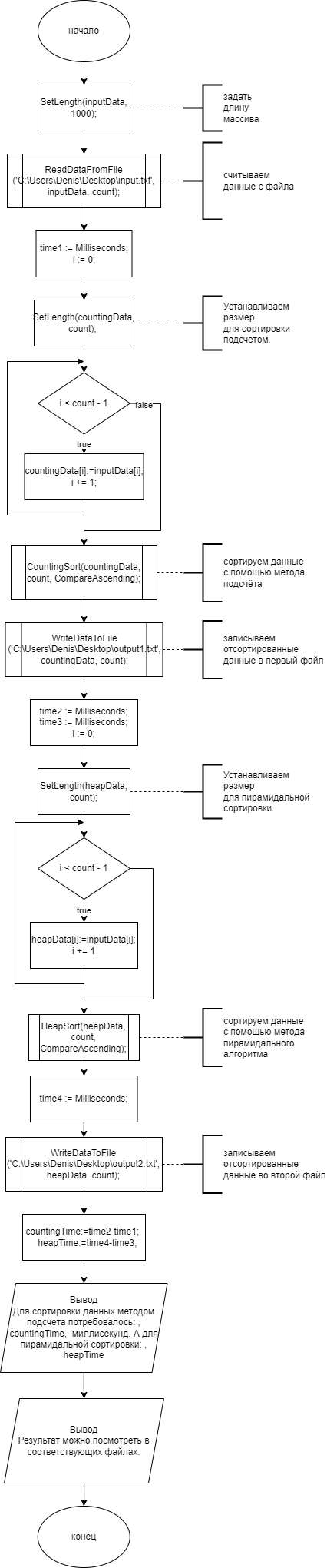


Рисунок 6 – схема алгоритма ввода и вывода результата

**Код программы:**

**procedure** CountingSort(**var** data: **array of** Integer; count: Integer; comparator: **function**(a, b: Integer): Integer);

**var**

i, j, Range, MinValue, MaxValue: Integer;

CountArray: **array of** Integer;

**begin**

**if** count = 0 **then Exit**;

MinValue := data[0];

MaxValue := data[0];

// Находим минимальное и максимальное значение

**for** i := 1 **to** count - 1 **do**

**begin**

**if** data[i] < MinValue **then** MinValue := data[i];

**if** data[i] > MaxValue **then** MaxValue := data[i];

**end**;

Range := MaxValue - MinValue + 1;

SetLength(CountArray, Range);

// Инициализация массива подсчета

**for** i := 0 **to** High(CountArray) **do**

CountArray[i] := 0;

// Подсчет количества вхождений каждого элемента

**for** i := 0 **to** count - 1 **do**

Inc(CountArray[data[i] - MinValue]);

// Заполнение исходного массива отсортированными элементами

j := 0;

**for** i := 0 **to** High(CountArray) **do**

**while** CountArray[i] > 0 **do**

**begin**

data[j] := i + MinValue;

Inc(j);

Dec(CountArray[i]);

**end**;

**end**;

**procedure** Heapify(**var** data: **array of** Integer; n, i: Integer; comparator: **function**(a, b: Integer): Integer);

**var**

largest, left, right: Integer;

**begin**

largest := i;

left := 2 \* i + 1;

right := 2 \* i + 2;

**if** (left < n) **and** (comparator(data[left], data[largest]) > 0) **then**

largest := left;

**if** (right < n) **and** (comparator(data[right], data[largest]) > 0) **then**

largest := right;

**if** largest <> i **then**

**begin**

Swap(data[i], data[largest]);

Heapify(data, n, largest, comparator);

**end**;

**end**;

**procedure** HeapSort(**var** data: **array of** Integer; count: Integer; comparator: **function**(a, b: Integer): Integer);

**var**

i: Integer;

**begin**

**for** i := count **div** 2 - 1 **downto** 0 **do**

Heapify(data, count, i, comparator);

**for** i := count - 1 **downto** 1 **do**

**begin**

Swap(data[0], data[i]);

Heapify(data, i, 0, comparator);

**end**;

**end**;

**function** CompareAscending(a, b: Integer): Integer;

**begin**

Result := a - b;

**end**;

**procedure** ReadDataFromFile(**const** filename: string; **var** data: **array of** Integer; **var** count: Integer);

**var**

f: TextFile;

**begin**

AssignFile(f, filename);

Reset(f);

count := 0;

**while not** Eof(f) **do**

**begin**

Read(f, data[count]);

Inc(count);

SetLength(data, count + 1); // Увеличиваем размер массива для следующего числа.

**end**;

CloseFile(f);

**end**;

**procedure** WriteDataToFile(**const** filename: string; **const** data: **array of** Integer; count: Integer);

**var**

f: TextFile;

**begin**

AssignFile(f, filename);

Rewrite(f);

**for var** i := 0 **to** count - 1 **do**

WriteLn(f, data[i]);

CloseFile(f);

**end**;

**var**

inputData: **array of** Integer;

countingData: **array of** Integer;

heapData: **array of** Integer;

count, i, time1, time2, time3, time4, countingTime, heapTime: Integer;

**begin**

SetLength(inputData, 1000);

ReadDataFromFile('C:\Users\Denis\Desktop\input.txt', inputData, count);

// Сортировка подсчетом

time1 := Milliseconds;

SetLength(countingData, count); // Устанавливаем размер для сортировки подсчетом.

**for** i := 0 **to** count - 1 **do**

countingData[i] := inputData[i];

CountingSort(countingData, count, CompareAscending);

WriteDataToFile('C:\Users\Denis\Desktop\output1.txt', countingData, count);

time2 := Milliseconds;

// Пирамидальная сортировка (Heap Sort)

time3 := Milliseconds;

SetLength(heapData, count); // Устанавливаем размер для пирамидальной сортировки.

**for** i := 0 **to** count - 1 **do**

heapData[i] := inputData[i];

HeapSort(heapData, count, CompareAscending);

time4 := Milliseconds;

WriteDataToFile('C:\Users\Denis\Desktop\output2.txt', heapData, count);

countingTime := time2 - time1;

heapTime := time4 - time3;

WriteLn('Для сортировки данных методом подсчета потребовалось: ', countingTime, ' миллисекунд. А для пирамидальной сортировки: ', heapTime, '.');

WriteLn('Результат можно посмотреть в соответствующих файлах.');

**end**.

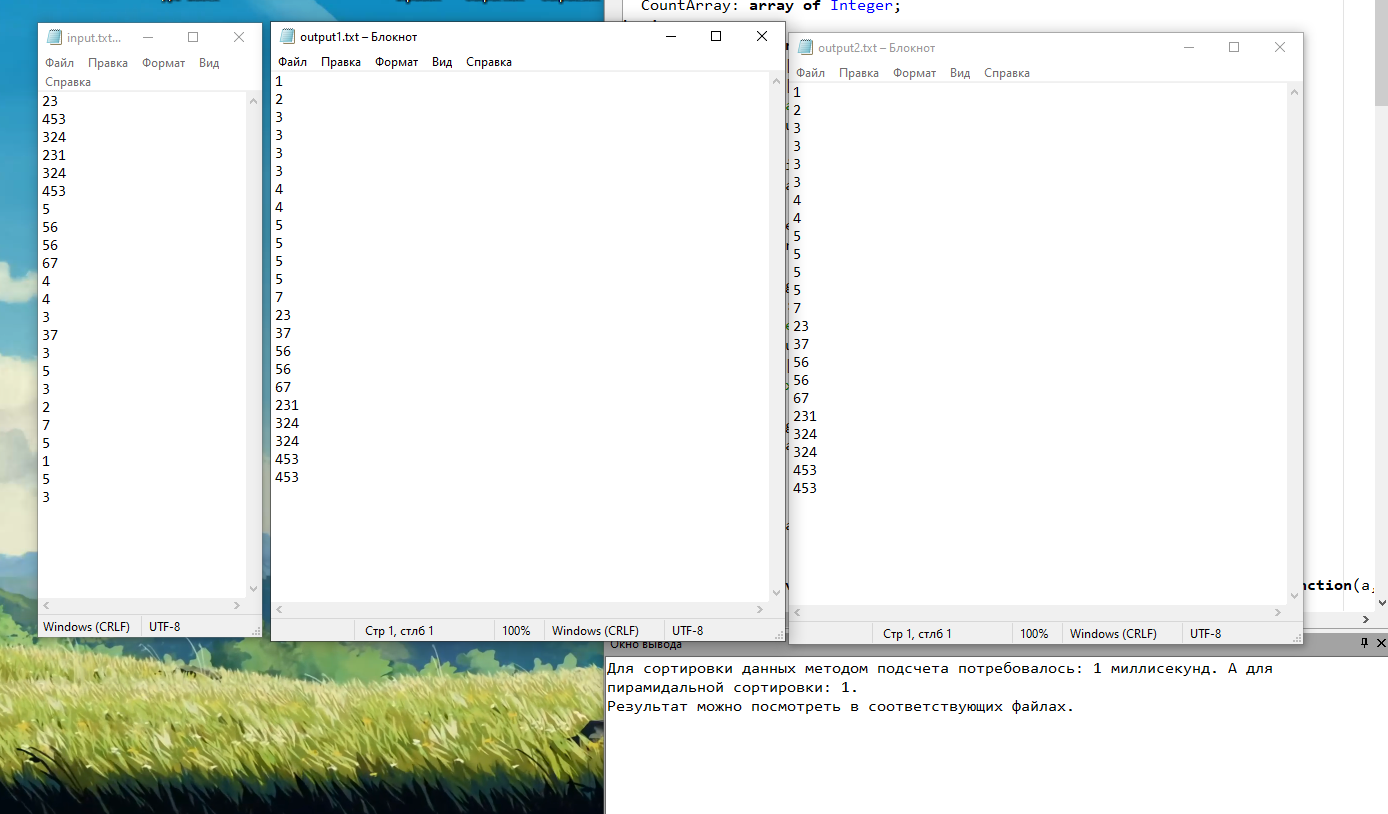


Рисунок 7 – результат выполнения программы

**Вывод:** в ходе данной работы мы провели исследование основных алгоритмов сортировки, таких как сортировка вставками, пузырьковая сортировка, быстрая сортировка и сортировка слиянием. Мы рассмотрели принципы их работы, сильные и слабые стороны, а также случаи, в которых каждый из этих алгоритмов может быть наиболее эффективным.

Сравнительный анализ алгоритмов сортировки показал, что выбор конкретного алгоритма зависит от объема данных, требуемой стабильности сортировки и ее сложности. Так, например, сортировка вставками хорошо подходит для почти отсортированных массивов, а быстрая сортировка является одним из самых эффективных подходов для больших массивов данных, несмотря на свою сложность в реализации.

Также мы изучили принципы работы с текстовыми файлами в языке Pascal. Создание, чтение и запись данных в файлы были проиллюстрированы с помощью примеров. Мы узнали о важности правильного управления ресурсами и обработки ошибок при работе с файлами. Использование текстовых файлов для сохранения исходных и отсортированных данных позволяет создавать более гибкие приложения и обрабатывать результаты без необходимости повторной сортировки.

Таким образом, сочетание знаний об алгоритмах сортировки и умений работать с текстовыми файлами открывает новые горизонты для разработки программного обеспечения и обработки данных. Мы уверены, что полученные навыки будут полезны в будущем как в изучении более сложных алгоритмов, так и в практической разработке программ.