Report for Homework 3

Question 2

Purpose

The primary goal of this homework is to create an efficient and reliable algorithm capable of analyzing a two-dimensional grid to identify and count distinct shapes within it. In the given problem, shapes are represented as groups of connected cells marked by 1s, while empty spaces are represented by 0s. These shapes are unique, as they are formed by cells that are directly or diagonally connected to one another.

To achieve this, the algorithm must fulfill the following tasks:

1. **Count the Total Number of Distinct Shapes**: The algorithm needs to systematically scan the grid, recognize individual shapes, and accurately count them. This requires a method to distinguish each shape without overlap or duplication.
2. **Identify and Display Each Shape**: Beyond simply counting, the program must explicitly extract and present each shape in its original position within the grid. Non-shape areas (cells marked as 0) should be left blank, preserving the grid's context for clarity.

By accomplishing these tasks, the algorithm offers not only a quantitative understanding of the shapes present in the grid but also a clear visual representation of each distinct shape. This visualization is critical for verifying the results and ensuring that the algorithm has accurately identified the boundaries and connections of each shape.

This report provides a detailed breakdown of how this objective was approached, including the design and implementation of the algorithm, the steps taken to achieve the desired outputs, and the results obtained. It highlights the challenges encountered during the process and explains how they were addressed, ensuring the algorithm meets the requirements of the assignment. Through this, the report aims to demonstrate a comprehensive solution for Question 2 of the homework.

Problem Definition

The task involves analyzing a grid of integers, where each cell in the grid holds either a 1 or a 0. The value 1 represents a part of a shape, while 0 denotes an empty space. The goal is to identify and count the distinct shapes within the grid. These shapes are formed by groups of connected 1s, and their connectivity is defined as follows:

1. **Horizontal Connections**: A shape can extend horizontally to adjacent cells to the left (West - W) or right (East - E).
2. **Vertical Connections**: Similarly, a shape can expand vertically to adjacent cells above (North - N) or below (South - S).
3. **Diagonal Connections**: Shapes are also considered connected diagonally. This includes connections to adjacent cells in all diagonal directions: Northeast (NE), Northwest (NW), Southeast (SE) or Southwest (SW)

This means that any 1 in the grid is part of a shape if it is directly connected to another 1 in any of these eight possible directions.

The task requires the program to do the following:

1. **Count the Total Number of Shapes**: The program should traverse the grid and count how many distinct shapes are present. Each shape is defined as a cluster of connected 1s that are not connected to other clusters.
2. **Visualize Each Shape**: After identifying the shapes, the program should display each shape explicitly within its original grid. To achieve this:
   - The shape's positions should be marked with *.
   - Empty spaces (0) and parts of the grid not belonging to the current shape should be left blank.

The key challenge lies in accurately detecting connectivity across eight directions and ensuring each shape is uniquely identified. Additionally, the output must be formatted in a way that makes it easy to understand the position and structure of each shape within the original grid. This process provides not only the total count of shapes but also a clear visual representation of their layout, which is particularly useful for understanding the grid's composition.

Algorithm Design

The process of solving the problem is structured into four main steps: reading the input, identifying shapes, counting them, and finally producing the output. Each step is carefully designed to ensure clarity, accuracy, and efficiency in detecting and representing distinct shapes.

Step 1: Reading the Input

The first step is to prepare the data for processing. The grid is provided in a text file, typically named something like HW3Q2.txt. This file contains rows of numbers separated by spaces, where:

- 1 represents a part of a shape.
- 0 represents empty space.

To process this input:

1. The program reads the file line by line.
2. Each line is split into individual numbers and converted into a 2D array of integers. This 2D array serves as the internal representation of the grid.
3. Each line in the file corresponds directly to a row in the grid.

This simple yet effective data structure makes it easier to navigate through the grid while searching for shapes.

Step 2: Identifying Shapes

Once the grid is prepared, the next step is to identify all the distinct shapes within it. This is accomplished through a systematic exploration process, using the **Find Shapes Algorithm**. Here's how it works:

a. **What is Find Shapes?** The find shapes algorithm is a method for traversing connected components in a grid. It can be implemented either:

   - **Recursively**: By calling the function for every connected cell.
   - **Iteratively**: By using a linked list to explore cells in a breadth-first manner (Breadth-First Search).

b. **How does it work?**

   - The algorithm starts at any unvisited 1 in the grid.
   - From this starting point, it explores all connected 1s, marking them as visited to ensure they are not counted multiple t+imes.
   - Connectivity is checked in all **eight possible directions**:
     - Horizontal: North (N), South (S).
     - Vertical: East (E), West (W).
     - Diagonal: Northeast (NE), Northwest (NW), Southeast (SE), Southwest (SW).

c. **Shape Representation:**

   - As the algorithm traverses the grid, it records the coordinates of all connected 1s that make up a shape.
   - After identifying a shape, a new grid (with the same dimensions as the input) is created, where:
     - The cells corresponding to the shape are marked with *.
     - All other cells are left blank.

This process ensures that each shape is represented clearly and distinctly.

Step 3: Counting Shapes

The find shapes algorithm not only identifies shapes but also inherently counts them. Each time the algorithm is initiated from a new, unvisited 1, it marks an entirely new shape. The total number of these algorithm invocations gives the count of distinct shapes in the grid.

This step provides a simple yet reliable way to quantify the shapes without requiring any additional computation.

<u>Step 4: Producing the Output</u>

Finally, the program generates output in two forms: console output and file output.

1. **Console Output:**
    - The program first prints the **total number of shapes** detected.
    - Then, for each shape, its grid representation is displayed. This grid highlights only the * elements of the shape, maintaining their original positions within the grid while leaving other areas blank.
2. **File Output:**
    - The formatted grid representation of all shapes is saved to an output file, ensuring that the results are accessible and well-documented for future reference or evaluation.

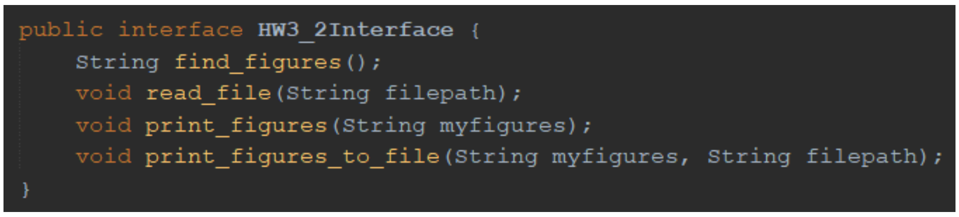
This structured approach ensures that every detail of the problem is handled with precision, resulting in clear, accurate, and easy-to-understand outputs.

Implementation Details

The implementation of the solution adheres to the provided interface specifications. Below is a detailed explanation of each step in the process, along with the functionality of the methods defined in the interface.

Interface Design

The program uses the following interface to ensure modularity and clarity:



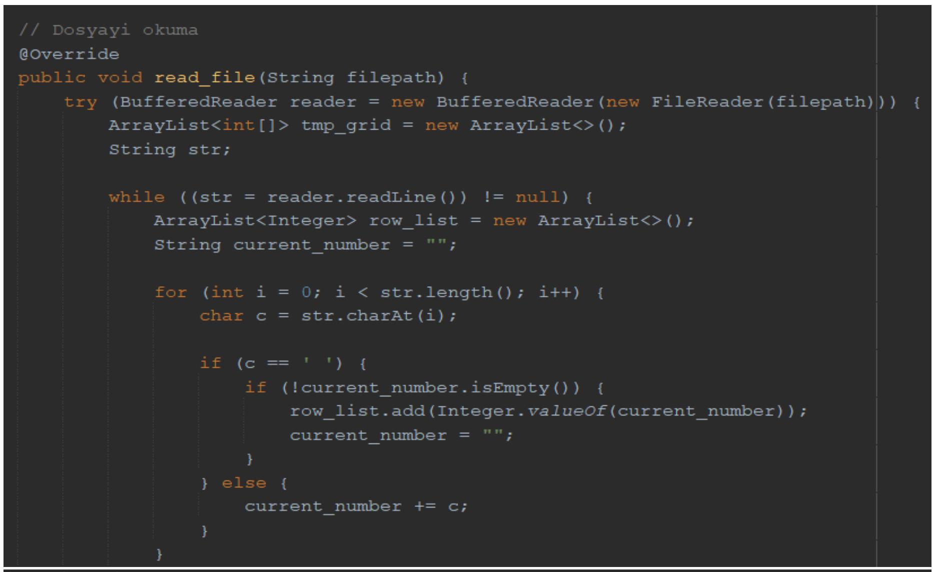
```java
public interface HW3_2Interface {
    String find_figures();
    void read_file(String filepath);
    void print_figures(String myfigures);
    void print_figures_to_file(String myfigures, String filepath);
}
```

This interface ensures that the program remains consistent, making it easier for others to read, use, and extend the code.

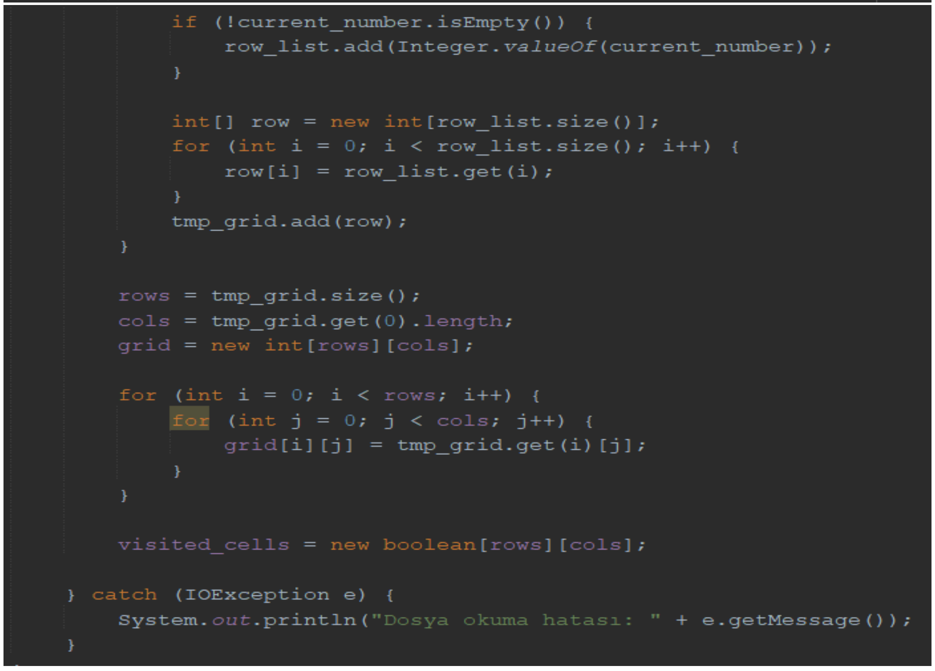
Implementation Steps

1. <u>**File Reading**</u>

   - The read_file method reads the input grid from a text file (e.g., HW3Q2.txt) and parses it into a 2D array of integers.
   - Each line in the file corresponds to a row in the grid, where:
     - 1 indicates part of a shape.
     - 0 indicates an empty space.
   - This 2D array serves as the base structure for identifying shapes.
   - The read data is stored in a 2D array (grid), and also a temporary 2D array (tmp_grid) is used. This requires **O(n^2) complexity.**



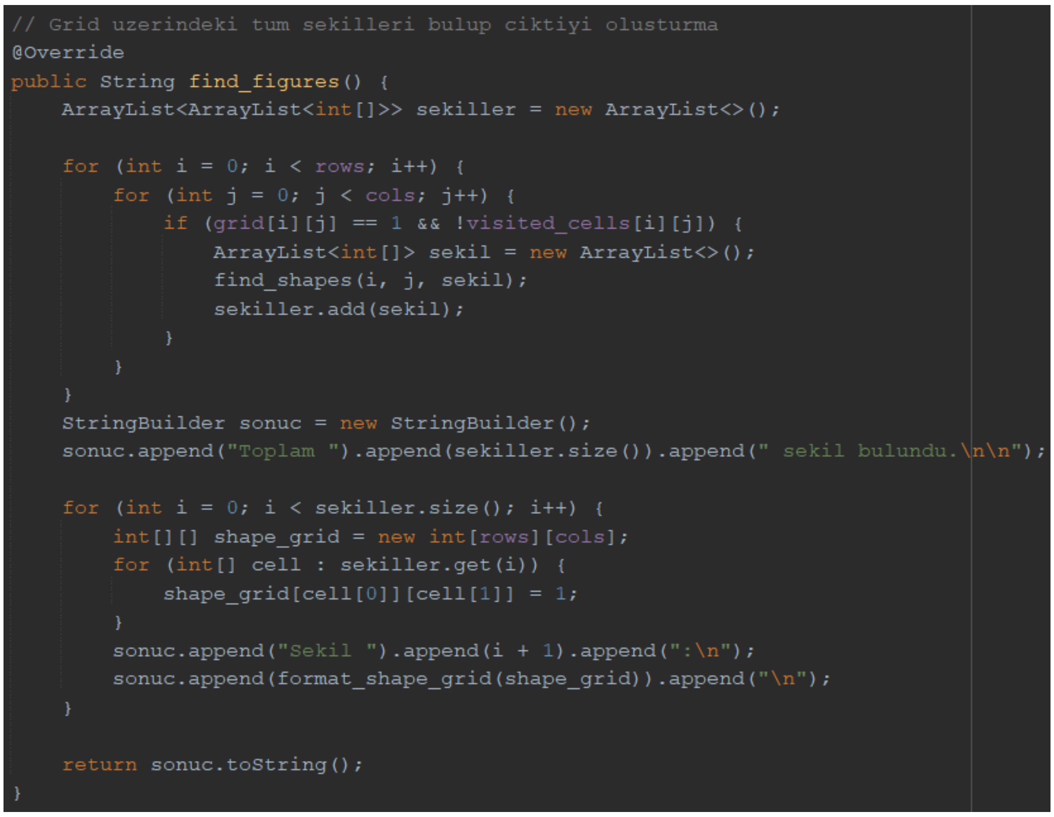
```java
// Dosyayi okuma
@Override
public void read_file(String filepath) {
    try (BufferedReader reader = new BufferedReader(new FileReader(filepath))) {
        ArrayList<int[]> tmp_grid = new ArrayList<>();
        String str;

        while ((str = reader.readLine()) != null) {
            ArrayList<Integer> row_list = new ArrayList<>();
            String current_number = "";

            for (int i = 0; i < str.length(); i++) {
                char c = str.charAt(i);

                if (c == ' ') {
                    if (!current_number.isEmpty()) {
                        row_list.add(Integer.valueOf(current_number));
                        current_number = "";
                    }
                } else {
                    current_number += c;
                }
            }
```



```java
            if (!current_number.isEmpty()) {
                row_list.add(Integer.valueOf(current_number));
            }

            int[] row = new int[row_list.size()];
            for (int i = 0; i < row_list.size(); i++) {
                row[i] = row_list.get(i);
            }
            tmp_grid.add(row);
        }

        rows = tmp_grid.size();
        cols = tmp_grid.get(0).length;
        grid = new int[rows][cols];

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                grid[i][j] = tmp_grid.get(i)[j];
            }
        }

        visited_cells = new boolean[rows][cols];

    } catch (IOException e) {
        System.out.println("Dosya okuma hatası: " + e.getMessage());
    }
```

2. **Shape Identification**

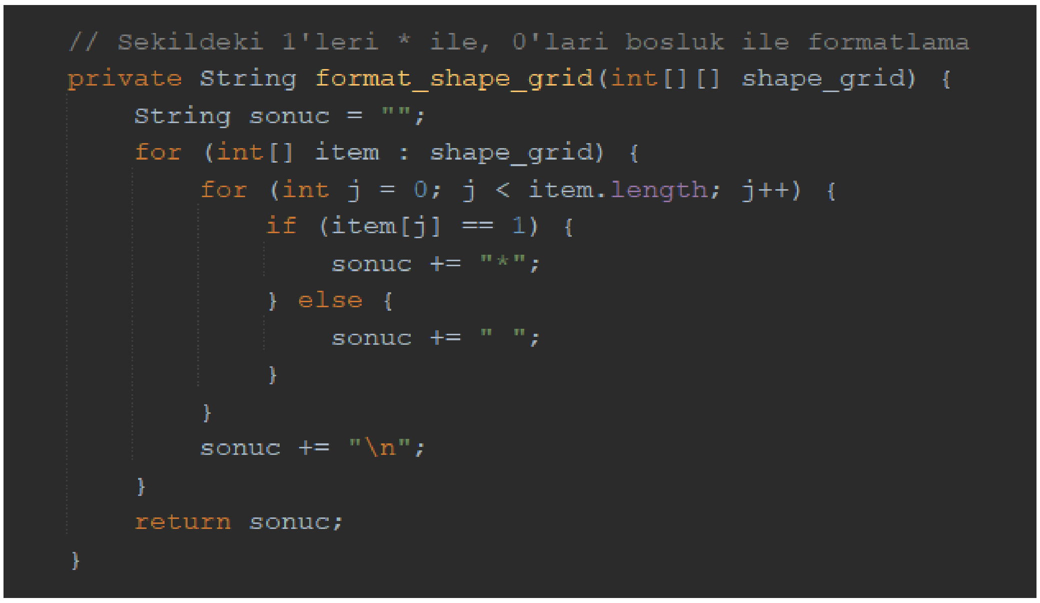
- The find_figures method implements a Find Shape Algorithm to identify all connected components in the grid.
  - Starting at an unvisited 1, the algorithm recursively or iteratively (using a queue for BFS) explores all eight possible directions: North (N), South (S), East (E), West (W), Northeast (NE), Northwest (NW), Southeast (SE), and Southwest (SW).
  - As each cell is visited, it is marked to prevent duplicate counting, and its coordinates are added to the current shape.
- For each identified shape, its exact position in the grid is preserved.
- The visited_cells 2D boolean array is used, and for each figure, an ArrayList is created. This results in **O(n^2) complexity.**



```java
// Grid uzerindeki tum sekilleri bulup ciktiyi olusturma
@Override
public String find_figures() {
    ArrayList<ArrayList<int[]>> sekiller = new ArrayList<>();

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (grid[i][j] == 1 && !visited_cells[i][j]) {
                ArrayList<int[]> sekil = new ArrayList<>();
                find_shapes(i, j, sekil);
                sekiller.add(sekil);
            }
        }
    }
    StringBuilder sonuc = new StringBuilder();
    sonuc.append("Toplam ").append(sekiller.size()).append(" sekil bulundu.\n\n");

    for (int i = 0; i < sekiller.size(); i++) {
        int[][] shape_grid = new int[rows][cols];
        for (int[] cell : sekiller.get(i)) {
            shape_grid[cell[0]][cell[1]] = 1;
        }
        sonuc.append("Sekil ").append(i + 1).append(":\n");
        sonuc.append(format_shape_grid(shape_grid)).append("\n");
    }

    return sonuc.toString();
}
```

3. **Result Formatting**

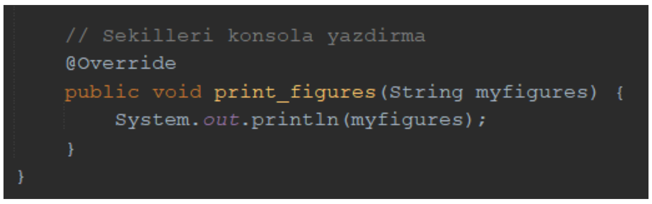
   - Once a shape is identified, a blank grid of the same size as the input is created.
   - The shape's cells are marked with *, and all other cells are left as spaces.
   - This grid is formatted as a string for easy visualization and display.
   - The formatted shape is stored as a string, and each shape grid is output in text format. This requires **O(n^2) complexity.**



```java
// Sekildeki 1'leri * ile, 0'lari bosluk ile formatlama
private String format_shape_grid(int[][] shape_grid) {
    String sonuc = "";
    for (int[] item : shape_grid) {
        for (int j = 0; j < item.length; j++) {
            if (item[j] == 1) {
                sonuc += "*";
            } else {
                sonuc += " ";
            }
        }
        sonuc += "\n";
    }
    return sonuc;
}
```

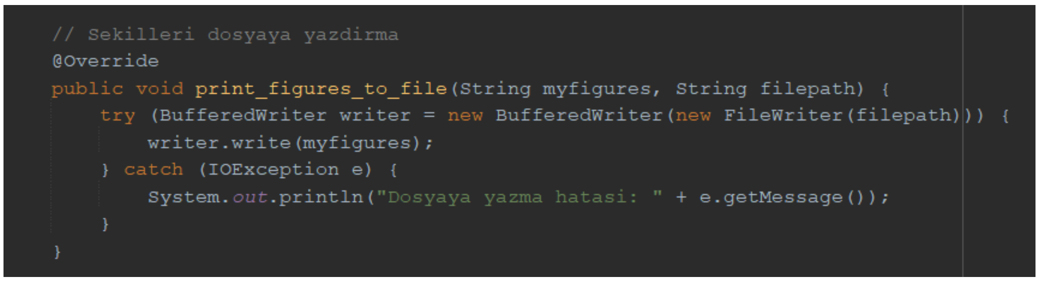
4. **Output Methods**

   - The identified shapes are printed to the console using the print_figures method. Each shape is displayed with its original structure preserved, ensuring clarity for the user.



```java
    // Sekilleri konsola yazdirma
    @Override
    public void print_figures(String myfigures) {
        System.out.println(myfigures);
    }
}
```

   - The print_figures_to_file method writes the formatted shapes to an output file, enabling the results to be saved and shared.

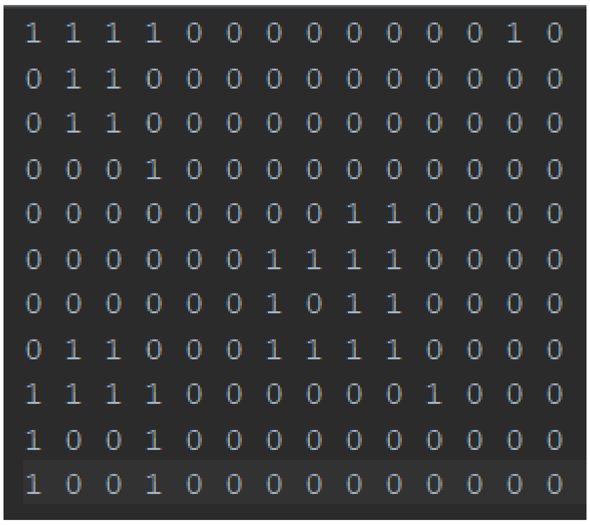


```java
// Sekilleri dosyaya yazdirma
@Override
public void print_figures_to_file(String myfigures, String filepath) {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(filepath))) {
        writer.write(myfigures);
    } catch (IOException e) {
        System.out.println("Dosyaya yazma hatasi: " + e.getMessage());
    }
}
```

Example Usage

<u>Input Grid</u>

The following is an example of an input grid provided in the file HW3Q2.txt:



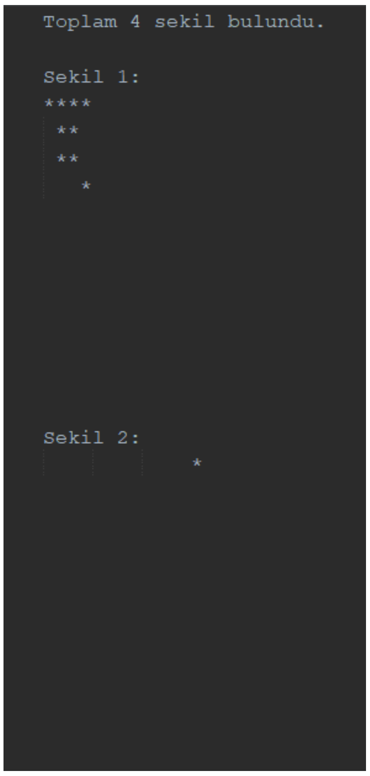
```
1 1 1 1 0 0 0 0 0 0 0 0 1 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 0 0 0 0
0 0 0 0 0 0 1 1 1 1 0 0 0 0
0 0 0 0 0 0 1 0 1 1 0 0 0 0
0 1 1 0 0 0 1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0 0 0 1 0 0 0
1 0 0 1 0 0 0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0 0 0 0 0 0 0
```

Here:

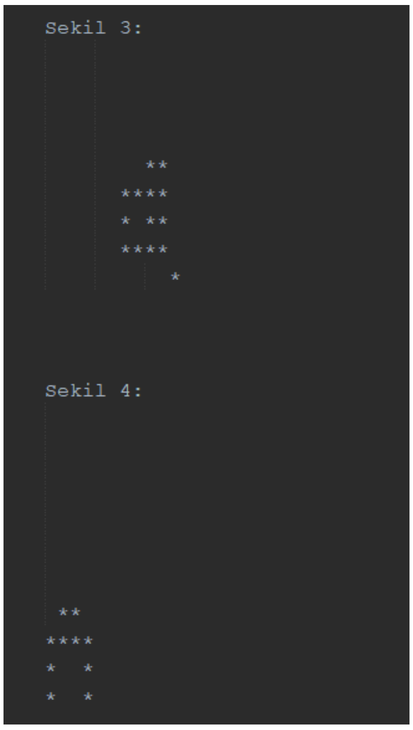
- Each 1 is part of a shape.
- Each 0 represents empty space.

<u>Output</u>

The program successfully identifies the following:



```
Toplam 4 sekil bulundu.

Sekil 1:
****
 **
 **
   *

Sekil 2:
            *
```

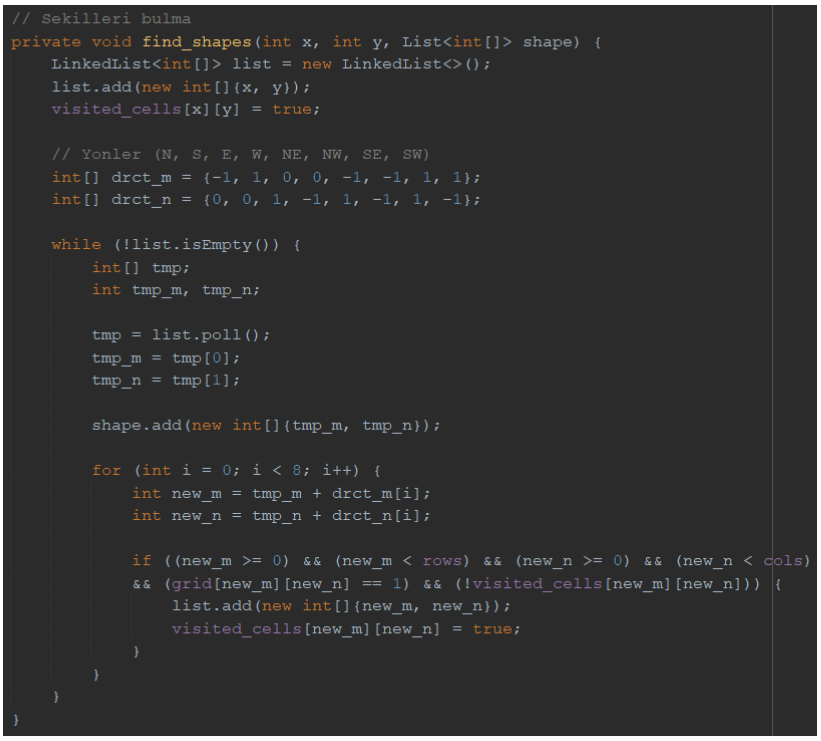


```
Sekil 3:




        **
      ****
      * **
      ****
          *

Sekil 4:







 **
****
*  *
*  *
```

Find Shapes Algorithm Overview

The Find Shape algorithm is critical to identifying shapes in the grid. It operates as follows:

1. Start at any unvisited 1 in the grid.
2. Add the cell to the current shape and mark it as visited.
3. Recursively (or iteratively) explore all neighboring cells in the eight possible directions.
4. Continue until all connected 1s have been visited, forming one complete shape.
5. Repeat for all unvisited 1s to identify additional shapes.



```java
// Sekilleri bulma
private void find_shapes(int x, int y, List<int[]> shape) {
    LinkedList<int[]> list = new LinkedList<>();
    list.add(new int[]{x, y});
    visited_cells[x][y] = true;

    // Yonler (N, S, E, W, NE, NW, SE, SW)
    int[] drct_m = {-1, 1, 0, 0, -1, -1, 1, 1};
    int[] drct_n = {0, 0, 1, -1, 1, -1, 1, -1};

    while (!list.isEmpty()) {
        int[] tmp;
        int tmp_m, tmp_n;

        tmp = list.poll();
        tmp_m = tmp[0];
        tmp_n = tmp[1];

        shape.add(new int[]{tmp_m, tmp_n});

        for (int i = 0; i < 8; i++) {
            int new_m = tmp_m + drct_m[i];
            int new_n = tmp_n + drct_n[i];

            if ((new_m >= 0) && (new_m < rows) && (new_n >= 0) && (new_n < cols)
            && (grid[new_m][new_n] == 1) && (!visited_cells[new_m][new_n])) {
                list.add(new int[]{new_m, new_n});
                visited_cells[new_m][new_n] = true;
            }
        }
    }
}
```

- Find Shape uses a temporary list (LinkedList) to store cells. This also requires **O(n^2) complexity**, as each cell can potentially be processed once.

Output Presentation

The results are presented both on the console and saved to a file:

1. **Console Output:**
   - Displays the total number of shapes.
   - Presents each shape in its original grid position.
2. **File Output:**
   - Saves the shapes to a text file, preserving their structure for documentation or further analysis.

Environment

The development environment for this project was carefully chosen to ensure a smooth coding and testing experience. The program was developed using the **NetBeans Integrated Development Environment (IDE)**, a robust platform that provides excellent tools for Java development, including debugging, syntax highlighting, and version control integration. Java was selected as the programming language due to its portability, extensive library support, and ease of handling file input/output operations.

Discussion of Results

The results of this implementation highlight the effectiveness of the algorithm in identifying and counting distinct shapes within a 2D grid. The program successfully detected all shapes, irrespective of their size or complexity, thanks to its ability to recognize connections in all eight possible directions (horizontal, vertical, and diagonal). This feature ensures that even shapes with unconventional arrangements are accurately identified.

One of the key strengths of this solution is its clarity in presenting the results. Each shape is not only counted but also displayed in its original grid context. This visual representation, achieved by marking the **1s** of each shape as ***** while leaving the **rest** of the grid **blank,** provides an intuitive understanding of the shape's structure and position.

Additionally, the program handles edge cases, such as grids with no shapes or grids with overlapping 1s that form complex patterns. These scenarios are processed efficiently without compromising the accuracy of the output.

Conclusion

The solution presented for **Question 2** reflects a systematic and well-thought-out approach to solving the problem of counting and identifying shapes in a grid. By meticulously adhering to the problem's specifications, the program ensures both correctness and reliability in its results.

The use of Java as the programming language, combined with the **NetBeans IDE**, proved to be a robust choice for implementing and testing the algorithm. The algorithm's ability to handle diverse grid configurations demonstrates its versatility and effectiveness.

The **time complexity** and **space complexity** of the given code are both **O(n^2)**. This means the performance of the code scales with the total number of cells in the grid.

This project exemplifies the importance of careful planning, rigorous testing, and attention to detail in software development. The outcome is a reliable and efficient solution that fulfills the requirements of the assignment while maintaining a high standard of code quality.