**KONGU ENGINEERING COLLEGE**

(Autonomous)

Perundurai,Erode – 638060

**DEPARTMENT OF INFORMATION TECHNOLOGY**

# Identifying Strongly Connected Component

**A MICRO PROJECT**

**REPORT FOR**

**DESIGN AND ANALYSIS OF ALGORITHMS (22ITT31)**

**SUBMITTED BY**

**ANISHMA R S**

**(23ITR004)**

**KONGU ENGINEERING COLLEGE**

(Autonomous)

Perundurai,Erode – 638060

**DEPARTMENT OF INFORMATION TECHNOLOGY**

# Identifying Strongly Connected Component

**A MICRO PROJECT**

**REPORT FOR**

**DESIGN AND ANALYSIS OF ALGORITHMS(22ITT31)**

**SUBMITTED BY**

**ANISHMA R S**

**(23ITR004)**

**KONGU ENGINEERING COLLEGE**

**(Autonomous)**

Perundurai,Erode – 638060

**DEPARTMENT OF INFORMATION TECHNOLOGY**

## <u>BONAFIDE CERTIFICATE</u>

| | |
|---|---|
| Name | :ANISHMA R S |
| Course Code | : 22ITT31 |
| Course Name | : DESIGN AND ANALYSIS OF ALGORITHMS |
| Semester | IV |

Certified that this is a bonafied record of work for application project done by the above student for 22ITT31-DESIGN AND ANALYSIS OF ALGORITHMS during the academic year 2024-2025.

Submitted for the Viva Voice Examination held on ____________________

Faculty Incharge Head of the Department

## ABSTRACT

This project implements an efficient algorithm to rearrange glasses in two formats: ordered and randomly mixed filled (F) and empty (E) glasses. It uses concepts from algorithm design such as parity-based positioning, state transitions, and movement minimization. The core logic applies a strategy akin to Decrease-and-Conquer to incrementally reduce disorder and arrange glasses in optimal patterns with minimal swaps. This project also introduces a comparative analysis of different arrangements (e.g., FEFE, EFEF) and dynamically selects the optimal rearrangement. The application is developed in Java and demonstrates real-world applications in pattern correction, sorting logic, and state transition algorithms.

# TABLE OF CONTENTS

## 1.0 INTRODUCTION

The arrangement of filled and empty glasses into specific patterns represents a classic example of optimization in algorithm design. The **GlassArranger** project aims to solve this problem by organizing 2n glasses—n filled (F) and n empty (E)—into a desired configuration using the minimum number of moves. The project explores both ordered and randomly shuffled initial states, applying logical strategies to transform them into alternating patterns like FEFE... or EFEF.... The solution demonstrates how parity-based decisions and swap operations can efficiently reduce disorder and achieve the goal state.

This problem highlights important algorithmic concepts such as **Decrease-and-Conquer**, array manipulation, and decision-based branching. Implemented in Java, the project not only solves a real-world style problem but also emphasizes clarity in code structure and algorithm design. The program intelligently evaluates different pattern possibilities, determines the most efficient path to the desired arrangement, and tracks the minimum moves required—making it a practical and educational tool for understanding algorithmic thinking.

## 1.1 PURPOSE

The main purpose of this project is to:

- Make the concept of pattern optimization using array manipulation more understandable for students and beginners in algorithmic problem-solving.
- Demonstrate the use of the **Decrease-and-Conquer** strategy to arrange filled (F) and empty (E) glasses into specific target patterns with minimal moves.
- Encourage logical reasoning and decision-making based on **index parity** (even/odd positions).
- Promote iterative refinement techniques to reach optimal solutions in minimal steps.
- Serve as an educational tool to show how simple algorithms can solve real-world-like problems efficiently.

## 1.2 OBJECTIVE

The primary objectives of this project are:

- **Educational Enhancement**

  To help learners understand how structured problem-solving strategies like swapping and pattern formation can be applied to rearrange data. The project visualizes the transformation from disordered to ordered states through a step-by-step process.
- **Algorithm Efficiency**

  To showcase the efficiency of conditional logic and Decrease-and-Conquer methods in comparison to brute-force approaches. It emphasizes minimizing operations while solving a practical problem and highlights how simple optimization strategies can significantly reduce computational effort.

## 1.1 METHODOLOGY OVERVIEW

1. **User Input**

   • The user enters the number n, representing the count of filled glasses.
   • The program automatically prepares an array of n filled (F) and n empty (E) glasses.
2. **Glass Arrangement Generation**

   • Two arrangements are created: one ordered (FFF...EEE) and one random using the Fisher-Yates shuffle.
   • Both are processed separately to demonstrate different logic flows.
3. **Pattern Evaluation**

   • For random arrangements, the system evaluates two optimal patterns: FEFE... and EFEF....
   • It calculates the required number of swaps for both and selects the one with fewer moves.
4. **Swapping Strategy**

   • Based on parity (even/odd positions), incorrect placements are identified.
   • Swap operations are performed between misaligned filled and empty glasses to minimize moves.
5. **Final Output**

   • The final arranged pattern is displayed.
   • The total number of moves (swaps) required for transformation is reported to the user.

## 2. PROBLEM STATEMENT

In many algorithmic problems, efficient reordering or arrangement of elements plays a crucial role. One such problem is the arrangement of glasses where n glasses are filled (F) and n are empty (E), resulting in an array of size 2n. These glasses can either be arranged in an ordered manner (e.g., all filled followed by all empty) or in a random sequence. The objective is to rearrange them into a desired pattern such as FEFE... or EFEF..., where filled and empty glasses alternate.

The challenge is to perform this rearrangement with the **minimum number of moves**, where a move is defined as a swap between two glasses. A brute-force approach may involve unnecessary swaps, leading to inefficiency, especially as n increases. Therefore, a strategic and optimized algorithm is required to identify misaligned positions and correct them based on index parity (even or odd positioning).

This project addresses the problem using a **Decrease-and-Conquer** strategy to reduce disorder step-by-step. It includes logic to evaluate multiple pattern configurations and choose the one that requires fewer swaps. The algorithm not only enhances understanding of basic sorting and manipulation techniques but also builds a foundation for solving morecomplex arrangement problems in the future.

# 3.0 METHODOLOGY

## 3.1 Input & Initialization

- The user inputs the number n representing the count of filled glasses.
- An array of size 2n is initialized with n filled (F) and n empty (E) glasses.
- Two arrangements are prepared: an ordered one (FFF...EEE) and a random one using the Fisher-Yates shuffle.

## 3.2 Pattern Evaluation

- For the random arrangement, two target patterns (FEFE... and EFEF...) are evaluated.
- The algorithm counts misaligned positions based on index parity and glass type.
- It selects the pattern that requires the fewest number of swaps.

## 3.3 Swap Logic Implementation

- The program identifies incorrect placements by checking parity (even or odd indices).
- Swap operations are performed between misaligned filled and empty glasses to correct the pattern step-by-step.

## 3.4 Arrangement Correction

- In the ordered arrangement, filled glasses in incorrect positions are swapped with empty glasses in the second half.
- The algorithm ensures that swaps maintain parity correctness to minimize moves.

## 3.5 Output & Visualization

- The final arranged pattern is printed to the console.
- The total number of swaps (moves) used for each arrangement type is displayed.
- Swap actions are optionally printed step-by-step to help users understand the optimization process.

## IMPLEMENTATION :

### 4.1 Input & Initialization:

```
  import java.util.Arrays;
import java.util.Random;
import java.util.Scanner;

public class GlassArranger {

    public static void main(String[] args) {
        Scanner inputReader = new Scanner(System.in);

        try {
            System.out.print("Enter number of filled glasses (n): ");
            int n = inputReader.nextInt();

            if (n <= 0) {
                System.out.println("Please enter a positive integer.");
                return;
            }

            // Ordered arrangement demonstration
            char[] orderedGlasses = new char[2 * n];
            Arrays.fill(orderedGlasses, 0, n, 'F');
            Arrays.fill(orderedGlasses, n, 2 * n, 'E');

            System.out.println("\nOrdered Initial Arrangement:");
            printGlassArrangement(orderedGlasses);

            int orderedMoves = arrangeOrderedGlasses(orderedGlasses);
            System.out.println("Arranged Pattern:");
            printGlassArrangement(orderedGlasses);
            System.out.println("Moves required: " + orderedMoves);

            // Random arrangement demonstration
            char[] randomGlasses = generateRandomArrangement(n);
            System.out.println("\nRandom Initial Arrangement:");
            printGlassArrangement(randomGlasses);
            int randomMoves = arrangeRandomGlasses(randomGlasses);
            System.out.println("Arranged Pattern:");
            printGlassArrangement(randomGlasses);
            System.out.println("Moves required: " + randomMoves);

        } catch (Exception e) {
            System.out.println("Invalid input. Please enter a positive integer.");
        } finally {
            inputReader.close();
        }
    }

    public static void printGlassArrangement(char[] glasses) {
        for (char glass : glasses) {
            System.out.print(glass + " ");
        }
        System.out.println();
    }

    public static char[] generateRandomArrangement(int n) {
        char[] glasses = new char[2 * n];
        Arrays.fill(glasses, 0, n, 'F');
        Arrays.fill(glasses, n, 2 * n, 'E');

        // Fisher-Yates shuffle
        Random rand = new Random();
        for (int i = glasses.length - 1; i > 0; i--) {
            int j = rand.nextInt(i + 1);
            swapGlasses(glasses, i, j);
        }
        return glasses;
    }

    private static void swapGlasses(char[] glasses, int i, int j) {
        char temp = glasses[i];
        glasses[i] = glasses[j];
        glasses[j] = temp;
    }`
```

## 4.2 Divide & Compare:

```
public static int arrangeOrderedGlasses(char[] glasses) {
    int n = glasses.length / 2;
    int moves = 0;

    for (int i = 1; i < n; i += 2) {
        if (glasses[i] == 'F') {
            int swapPos = findEmptyAtEvenPosition(glasses, n);
            if (swapPos != -1) {
                System.out.println("Swapping positions " + i + " and " +
swapPos);
                swapGlasses(glasses, i, swapPos);
                moves++;
            }
        }
    }
    return moves;
}

private static int findEmptyAtEvenPosition(char[] glasses, int start) {
    for (int i = start; i < glasses.length; i++) {
        if (i % 2 == 0 && glasses[i] == 'E') {
            return i;
        }
    }
    return -1;
}
```

## 4.3 Recursive Detection

```
static int arrangeAsFEFE(char[] glasses, GlassCount counts) {
    int moves = 0;

    for (int i = 0; i < glasses.length; i++) {
        if (i % 2 == 0 && glasses[i] == 'E') {
            int swapPos = findFilledAtOddPosition(glasses, i + 1);
            if (swapPos != -1) {
                System.out.println("Swapping positions " + i + " and " +
swapPos);
```

```
                    swapGlasses(glasses, i, swapPos);
                    moves++;
                }
            } else if (i % 2 == 1 && glasses[i] == 'F') {
                int swapPos = findEmptyAtEvenPosition(glasses, i + 1);
                if (swapPos != -1) {
                    System.out.println("Swapping positions " + i + " and " +
swapPos);
                    swapGlasses(glasses, i, swapPos);
                    moves++;
                }
            }
        }
        return moves;
    }

    private static int arrangeAsEFEF(char[] glasses, GlassCount counts) {
        int moves = 0;

        for (int i = 0; i < glasses.length; i++) {
            if (i % 2 == 0 && glasses[i] == 'F') {
                int swapPos = findEmptyAtOddPosition(glasses, i + 1);
                if (swapPos != -1) {
                    System.out.println("Swapping positions " + i + " and " +
swapPos);
                    swapGlasses(glasses, i, swapPos);
                    moves++;
                }
            } else if (i % 2 == 1 && glasses[i] == 'E') {
                int swapPos = findFilledAtEvenPosition(glasses, i + 1);
                if (swapPos != -1) {
                    System.out.println("Swapping positions " + i + " and " +
swapPos);
                    swapGlasses(glasses, i, swapPos);
                    moves++;
                }
            }
        }
        return moves;
    }
    private static int findFilledAtOddPosition(char[] glasses, int start) {
        for (int i = start; i < glasses.length; i += 2) {
            if (glasses[i] == 'F') return i;
        }
        return -1;
    }

    private static int findEmptyAtOddPosition(char[] glasses, int start) {
        for (int i = start; i < glasses.length; i += 2) {
            if (glasses[i] == 'E') return i;
        }
        return -1;
    }

    private static int findFilledAtEvenPosition(char[] glasses, int start) {
        for (int i = start % 2 == 0 ? start : start + 1; i < glasses.length; i += 2)
{
            if (glasses[i] == 'F') return i;
        }
        return -1;
    }
}
```

**DIFFERENCE BETWEEN BRUTEFORCE AND DIVIDE AND CONQUER:**

**Brute Force:**

**Concept:**

- Check every possible glass pair that violates the desired pattern.
- For the ordered setup, swap misplaced filled glasses from odd indices with empty glasses from even indices.
- Focuses on pairwise correction without leveraging global pattern detection.

**How it works:**

4.4.1 Traverse the first half (expected to be all filled).

4.4.2 For each misplaced filled glass at an odd index, find a suitable empty glass in the second half (even position).

4.4.3 Swap and count the move.

4.4.4 Continue until all filled glasses are in the first half.

**Time Complexity:**

4.4.4.1 Worst case: **O($n^2$)** in inefficient implementations, but optimized to **O(n)** due to controlled linear swaps and early exits.

**Pros:**

- Simple to understand and implement.
- Best suited when the pattern is predefined and clear (e.g., first n filled, last n empty).

**Cons:**

- Not scalable for complex arrangements.
- Doesn't adapt well to random layouts.
- May perform unnecessary checks in random configurations.

**Divide and Conquer Approach (Random to Patterned Arrangement):**

**Concept:**

- Divide the problem into manageable sub-goals (e.g., arranging as “FEFE…” or “EFEF…”).
- Use pattern analysis to choose the optimal arrangement strategy.
- Use helper functions to isolate mismatches and resolve them via minimal swaps.

**How it works:**

1. **Count Analysis**:
   - Count how many filled and empty glasses are on even and odd positions.
2. **Choose Optimal Pattern**:
   - Compare movesForFEFE() vs. movesForEFEF(), pick the one with fewer moves.

3. **Resolve Mismatches**:
    - For each index, if the current glass doesn't match the expected pattern, find a suitable pair to swap.
    - Use helper methods like findFilledAtOddPosition() or findEmptyAtEvenPosition().

**Time Complexity:**

- **O(n)** — efficient due to linear scans and minimal swaps.

**Pros:**

- Adaptive and optimal for random initial configurations.
- Reduces unnecessary operations by pre-analyzing mismatches.
- Easily scalable.

**Cons:**

- Slightly more complex logic with conditional handling.
- Requires helper functions and state tracking.

**Comparison Table:**

| **Feature** | **Brute Force** | **Divide and Conquer (Pattern Matching)** |
|---|---|---|
| **Strategy** | Scan and fix mismatches directly | Analyze pattern mismatch and fix optimally |
| **Time Complexity** | O(n) – Ordered case only | O(n) – Optimized for random case |
| **Efficiency** | Medium (context specific) | High (adaptive to input layout) |
| **Ideal for** | Ordered or predefined pattern | Randomized or mixed input |
| **Use of Helpers/Stack** | Minimal | Modular (pattern-based methods) |
| **Logic Complexity** | Simple | Moderate |

## Algorithm Analysis:

**Brute Force Approach**

**Input:** Glass array of size 2n with n filled ('F') and n empty ('E') glasses

**Method:** Swap misplaced glasses based on their index parity

**Process:**

1. Start from index 1 to n (odd positions).
2. If a filled glass is at an odd index, find an empty one at an even position in second half.
3. Swap and increment move count.
4. Repeat until desired state (first n filled, last n empty) is reached.

**Time Complexity:**

- **O(n)** with optimized lookup
- Straightforward for ordered transformation

**Divide and Conquer Approach (Pattern Reformatting)**

**Input:** Randomized glass array with equal numbers of 'F' and 'E'

**Method:** Reorganize to minimize swaps using pattern matching logic

**Steps:**

**Step 1: Divide**

- Count mismatches for FEFE and EFEF patterns
- Identify minimum move pattern

**Step 2: Conquer**

- Traverse and correct each mismatch using swap operations
- Use helper functions for locating correct pairs

**Step 3: Combine**

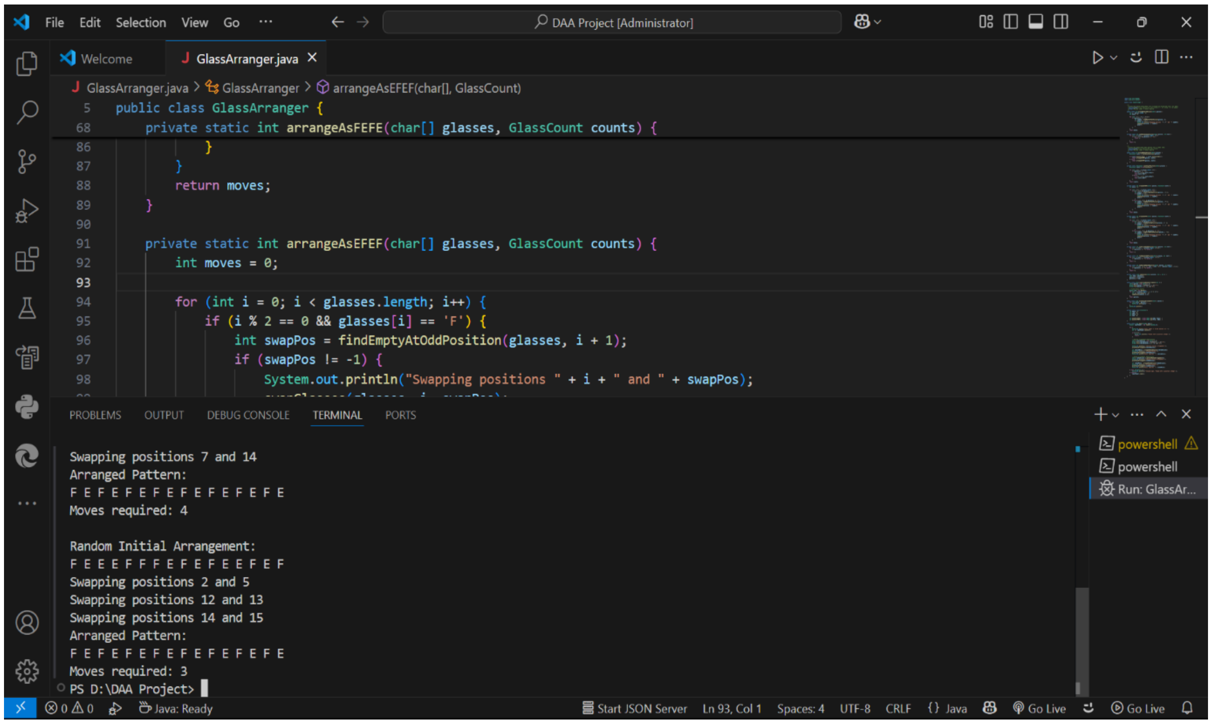
- Final array fits chosen pattern
- Display rearranged configuration and swap count

**Output:** Patterned array (e.g., FEFEFEFE...) with minimized moves

**Time Complexity:**

- **O(n)** — Efficient due to targeted swaps and early correction.

## 5.0. RESULTS:



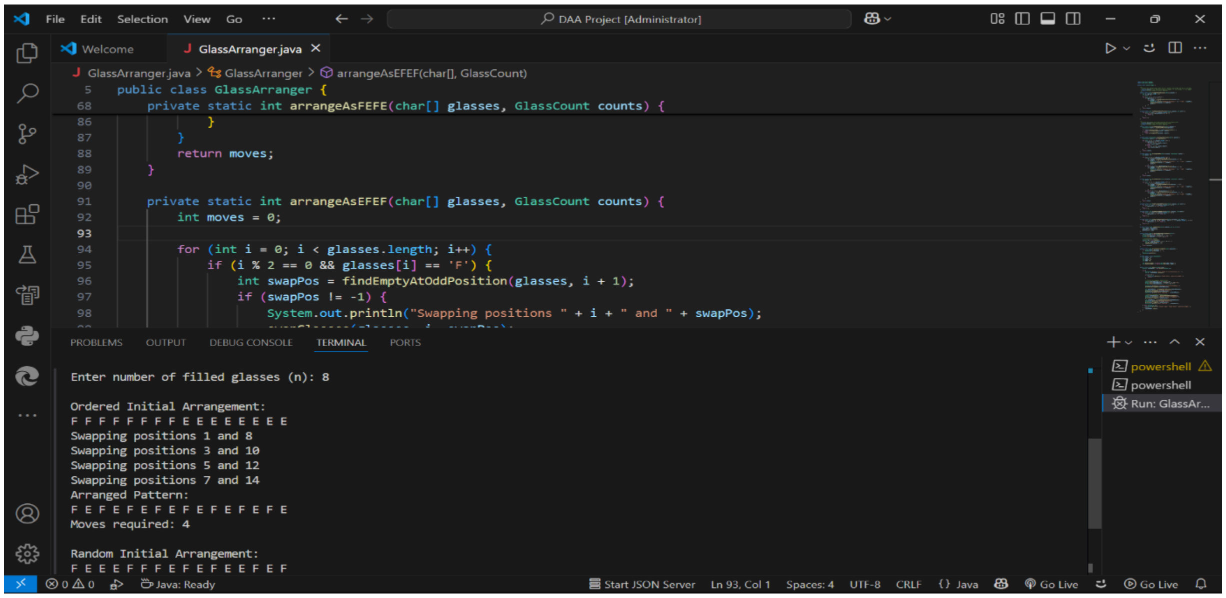
```java
public class GlassArranger {
    private static int arrangeAsFEFE(char[] glasses, GlassCount counts) {
                }
            }
            return moves;
        }

        private static int arrangeAsEFEF(char[] glasses, GlassCount counts) {
            int moves = 0;

            for (int i = 0; i < glasses.length; i++) {
                if (i % 2 == 0 && glasses[i] == 'F') {
                    int swapPos = findEmptyAtOddPosition(glasses, i + 1);
                    if (swapPos != -1) {
                        System.out.println("Swapping positions " + i + " and " + swapPos);
```

```
Swapping positions 7 and 14
Arranged Pattern:
F E F E F E F E F E F E F E F E
Moves required: 4

Random Initial Arrangement:
F E E E F F F E F E F E E F E F
Swapping positions 2 and 5
Swapping positions 12 and 13
Swapping positions 14 and 15
Arranged Pattern:
F E F E F E F E F E F E F E F E
Moves required: 3
PS D:\DAA Project>
```



```java
public class GlassArranger {
    private static int arrangeAsFEFE(char[] glasses, GlassCount counts) {
                }
            }
            return moves;
        }

        private static int arrangeAsEFEF(char[] glasses, GlassCount counts) {
            int moves = 0;

            for (int i = 0; i < glasses.length; i++) {
                if (i % 2 == 0 && glasses[i] == 'F') {
                    int swapPos = findEmptyAtOddPosition(glasses, i + 1);
                    if (swapPos != -1) {
                        System.out.println("Swapping positions " + i + " and " + swapPos);
```

```
Enter number of filled glasses (n): 8

Ordered Initial Arrangement:
F F F F F F F F E E E E E E E E
Swapping positions 1 and 8
Swapping positions 3 and 10
Swapping positions 5 and 12
Swapping positions 7 and 14
Arranged Pattern:
F E F E F E F E F E F E F E F E
Moves required: 4

Random Initial Arrangement:
F E E E F F F E F E F E E F E F
```

**GITHUB LINK: https://github.com/AnishmaGraze-10/DAA-PROJECT**