Green University of Bangladesh

Department of Computer Science and Engineering (CSE)
Faculty of Sciences and Engineering (FSE)
Semester: (Fall, Year: 2023), B.Sc. in CSE (Day)

UVA Online Judge Problems Solving

Course Title: Algorithms Lab
Course Code: CSE-206
Section: 221-D28

Students Details

Name	ID
Md. Kaised Mollick	221902070

Submission Date: 11.01.2024
Course Teacher's Name: Md Fahimul Islam

[For teachers use only: Don't write anything inside this box]

Lab Project Report Status	
Marks:	**Signature:**
Comments:	**Date:**

Contents

Chapter 1

Introduction

1.1 Overview

The project aims to implement efficient Java solutions for two algorithmic challenges. The Traveling Knight Problem (TKP) and the Train Swapping Problem. These problems involve solving complex scenarios related to chessboard optimization and train carriage rearrangement. The primary focus is on developing algorithms that find optimal solutions for the given problem statements. [1] [2]

1.2 Motivation

The motivation behind choosing this project lies in the intersection of algorithmic problem-solving, Java programming proficiency, and real-world applications. Solving these problems not only enhances algorithmic thinking but also addresses challenges with practical implications. The project provides an opportunity to explore and implement solutions to problems encountered in various domains.

1.3 Problem Definition

1.3.1 Problem Statement

The project addresses two distinct problems:

- **Traveling Knight Problem:** Finding the shortest route of knight moves between two squares on a chessboard.
- **Train Swapping Problem:** Automating the rearrangement of train carriages with the least number of swaps

1.3.2 Complex Engineering Problem

The following table summarizes the attributes related to the complex engineering problem addressed by the project:

Table 1.1: Summary of the attributes touched by the mentioned project

Name of the P Attributes	Explain how to address
P1: Depth of knowledge required	Gaining a deep understanding of graph theory for the Traveling Knight Problem and efficient sorting algorithms for the Train Swapping Problem.
P2: Range of conflicting requirements	Balancing the need for optimal knight moves on the chessboard and minimizing swaps for train rearrangement.
P3: Depth of analysis required	Conducting thorough analysis to design algorithms that provide optimal solutions for both problems.
P4: Familiarity of issues	Becoming familiar with chessboard representations, knight moves, and train carriage rearrangement strategies.
P5: Extent of applicable codes	Implementing Java code to address the specific requirements of the Traveling Knight and Train Swapping Problems.
P6: Extent of stakeholder involvement and conflicting requirements	Considering the interests of chess players for the Traveling Knight Problem and the efficiency concerns of railway companies for the Train Swapping Problem.
P7: Interdependence	Recognizing the interdependence of graph algorithms in the Traveling Knight Problem and the sequencing challenges in the Train Swapping Problem.

1.4 Design Goals/Objectives

The goals and objectives of the project are as follows:

- Implement a Java program to determine the minimum number of knight moves between two given squares on a chessboard.
- Implement a Java solution for the Train Swapping Problem.
- Enhance proficiency in graph theory, algorithmic problem-solving, and competitive coding using Java.
- Explore and implement efficient algorithms for both the Traveling Knight Problem and Train Swapping Problem.

- Achieving optimal solutions while considering the specific constraints of each problem.
- Ensuring code modularity and readability for future reference and potential extensions.
- Gain insights into Java's object-oriented programming capabilities and reinforce coding best practices.

1.5 Application

The solutions to the Traveling Knight Problem and Train Swapping Problem offer diverse and efficient applications:

1. Traveling Knight Problem:

- **Robotics Path Planning:** Extend the solution to complex robotic scenarios, including multi-agent systems and dynamic environments.
- **Network Routing:** Apply the knight-move logic to optimize secure packet routing in network security applications.

2. Train Swapping Problem:

- **Smart City Traffic Management:** Integrate the train swapping algorithm into smart city logistics for efficient traffic flow and reduced congestion.
- **Supply Chain Optimization:** Extend the solution to optimize carriage sequencing in supply chain logistics, enhancing overall transportation efficiency.

The project's outcomes advance algorithmic solutions with a dual focus on efficiency and adaptability, offering valuable tools for diverse industries, including robotics, network security, smart city planning, and logistics.

Chapter 2

Design/Development/Implementation of the Project

2.1 Introduction

This section serves as a gateway to the intricate design and implementation facets encapsulated within both projects. Here, I offer a panoramic view of the conceptualization and execution of solutions for the Traveling Knight Problem and the Train Swapping Problem. Delving into the fundamental objectives and motivations behind each endeavor, I embark on a journey that unfolds the intricacies of crafting algorithms and developing robust implementations to address these challenging problems. [3] [4] [5].

2.2 Project Details

Within this section, a granular exploration of the project intricacies unfolds as I delve into dedicated subsections, each meticulously addressing the nuances of the individual problems at hand. With a keen focus on essential aspects, we navigate through the specific details and intricacies that define the design and implementation of solutions for both the Traveling Knight Problem and the Train Swapping Problem.

2.2.1 Traveling Knight Problem Implementation

The Traveling Knight Problem (TKP) is a classic puzzle in chess that involves finding the shortest closed tour of knight moves on a chessboard, visiting each square exactly once. In this scenario, a knight is a chess piece that moves in an "L" shape: two squares in one direction (horizontally or vertically) and one square perpendicular to that direction. The goal is to determine the minimum number of knights moves required to travel between two specified squares on the chessboard.

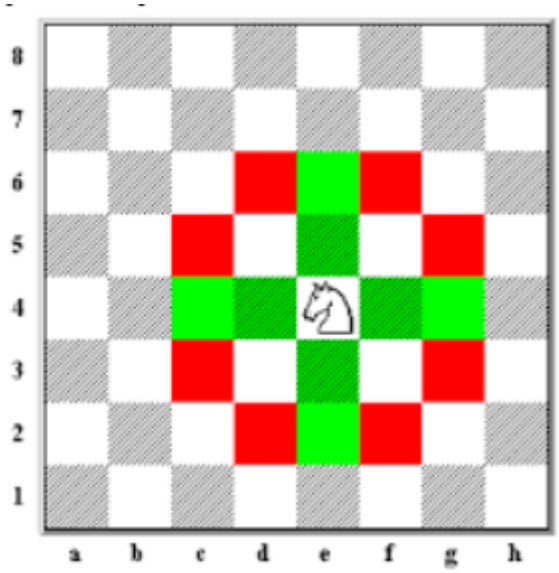


Figure 2.1: A chess board.

2.2.2 Train Swapping Problem Implementation

Within this subsection, we delve into the core of the Train Swapping Problem implementation. An in-depth explanation unfolds, elucidating the strategies employed to optimize train carriage arrangements. The crux of the implementation lies in the application of the merge sort algorithm, and a meticulous outline is presented on how this sorting technique is harnessed to minimize the number of swaps required for optimal train configuration. Through detailed insights into the coding structure, the steps taken to address the challenges posed by the Train Swapping Problem are illuminated, ensuring a streamlined and efficient arrangement of train carriages.

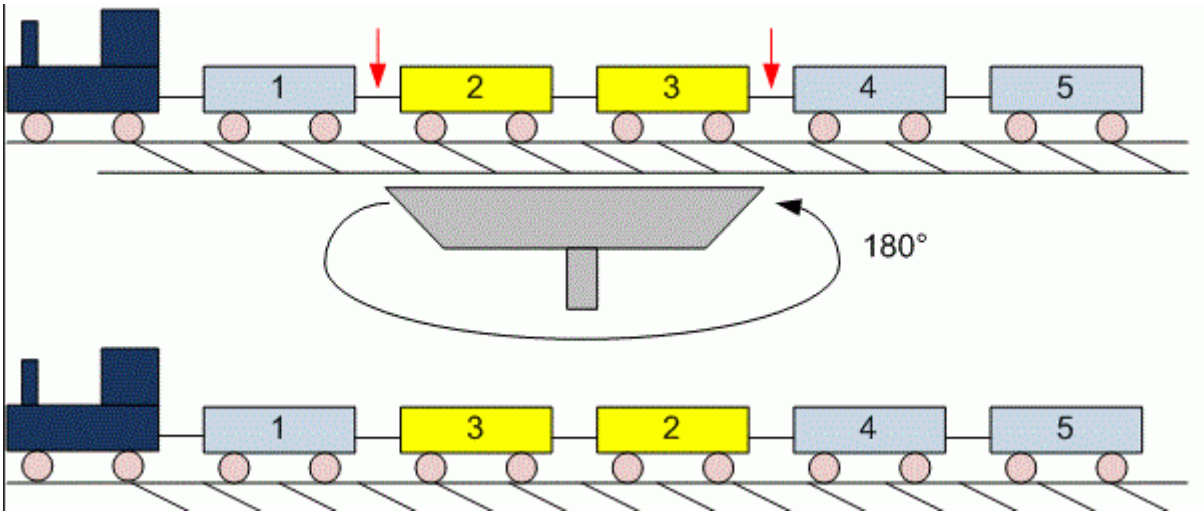


Figure 2.2: Train Swapping

2.3 Implementation

All the implementation details of my project are included in this section, along with subsections.

2.3.1 Tools and libraries

- **Language:** Java is my programming language to build the entire project.
- **IDE (VSCode)**: VSCode help me to compile the whole project.
- Utilization of Standard Java Libraries.
- Integration of Java's Data Structures.
- Application of Java's Algorithms.
- **GitHub:** GitHub will help me to store the codes online.

2.3.2 Traveling Knight Problem Implementation Details

This subsection covers implementation details for the Traveling Knight Problem.

The process to solve this problem:

Here's the process of solving the Traveling Knight Problem, I use to solve this problem by using Breath First Search (BFS):

- **Input:** The input consists of multiple test cases. Each test case contains two strings, each representing a square on the chessboard in the format "a1" (column letter followed by row number).
- **Iteration:** For each test case, the algorithm iterates until a solution is found or all possibilities are exhausted.In each iteration, the Algorithm:
    1. Represents the chessboard as an 8x8 2D array.
    2. Marks the starting square as visited.
    3. Explores possible knight moves from the current position.
    4. Keeps track of the number of moves taken to reach each square.
    5. Marks visited squares to avoid loops.
- **Checking the Solution:** If the target square is found, the algorithm prints the moves with printing format and terminates the iteration.

2.3.3 Train Swapping Problem Implementation Details

Similar to the previous subsection, this covers the implementation details of the Train Swapping Problem.

The process to solve this problem:

- **Input:** The input comprises multiple test cases, with each case consisting of two lines. The first line indicates the length of the train, denoted by the integer 'L'. The second line presents a permutation of numbers 1 through L, representing the initial order of train carriages.
- **Algorithmic Approach:** The algorithm employs the merge sort algorithm to achieve an optimal arrangement of train carriages.The merge sort algorithm is chosen for its efficiency in handling large datasets and its suitability for solving inversion-related problems.
- **Merge Sort Implementation:** The implementation initializes a count variable to track the number of swaps needed for optimal ordering. The merge sort algorithm divides the train carriage sequence into smaller segments, recursively sorts them, and then merges them to achieve an overall sorted order. During the merging process, the algorithm counts the number of swaps required when elements from different segments are merged, contributing to the overall inversion count.
- **Results:** After sorting, the algorithm provides the output sentence: 'Optimal train swapping takes S swaps,' where S represents the total number of swaps made during the optimization process.

By adopting the merge sort algorithm, the Train Swapping Problem solution ensures an efficient and systematic approach to rearrange train carriages, minimizing the required number of swaps for optimal order.

2.4 Algorithms

The algorithms and the programming codes in detail are included .

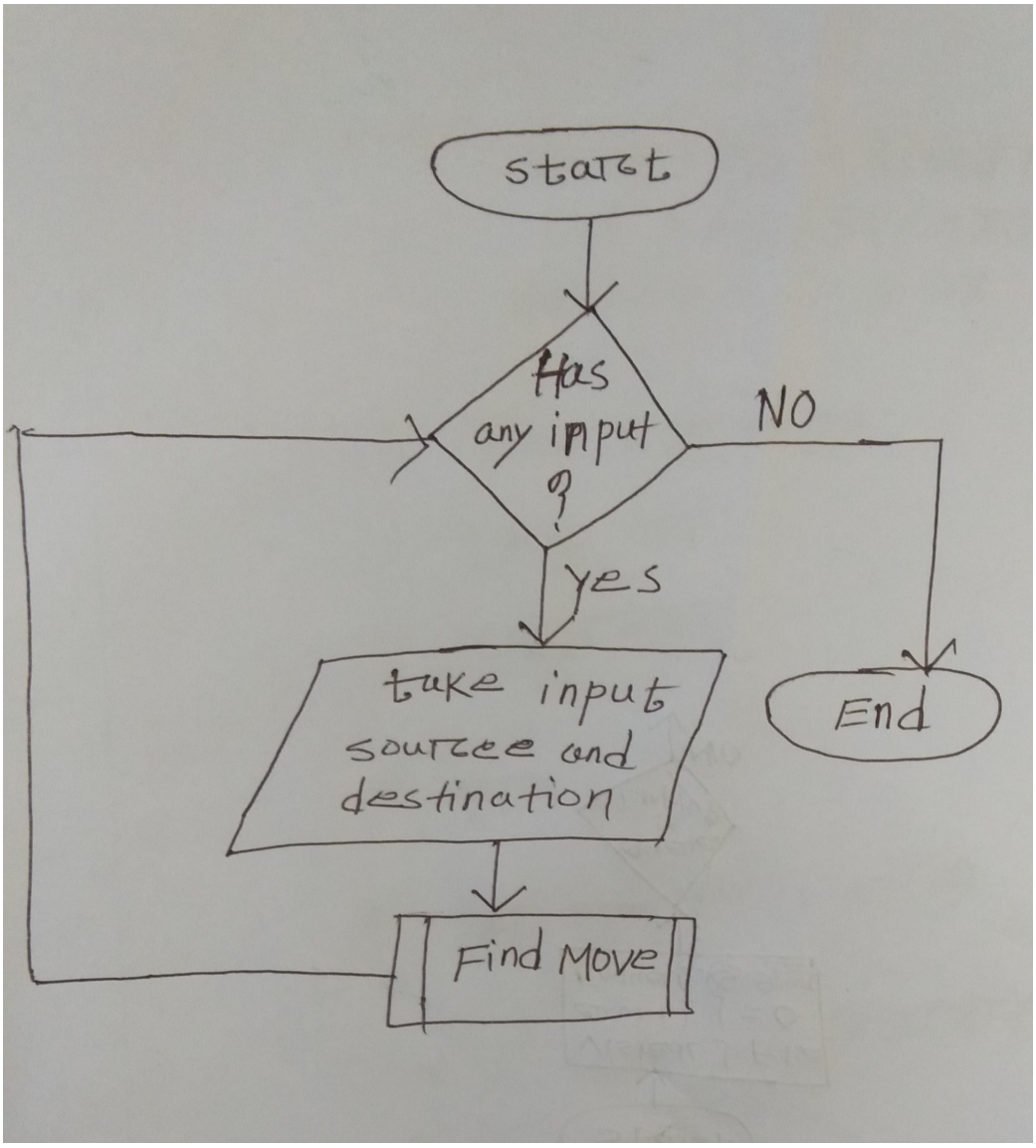
2.4.1 Traveling Knight Problem Algorithm pseudocode Are Given Below :

Algorithm 1: Knight's Shortest Path Algorithm

```
1  Xaxis ← {2, 2, -2, -2, 1, -1, 1, -1};
2  Yaxis ← {1, -1, 1, -1, 2, 2, -2, -2};
3  moves [][] ← matrix of size 8x8;
4  visited [][] ← matrix of boolean values of size 8x8;
5  start, ← strings;
6  PairPairattributes x, y
7  Function clearVisitedArray:
8      for each row in visited do
9          Set all values to false;
10 Function findMove(a, b, c, d):
11     Create a queue q;
12     clearVisitedArray ();
13     Add a new Pair(a, b) to the queue;
14     Set visited [a][b] to true;
15     Set moves [a][b] to 0;
16     while the queue is not empty do
17         Remove a Pair(p) from the queue;
18         if p.x is equal to c and p.y is equal to d then
19             Print "To get from " + start + " to " + + " takes " + moves [p.x][p.y]
                + " knight moves.";
20             return;
21         for each i from 0 to 7 do
22             Calculate currentX as p.x + Xaxis[i];
23             Calculate currentY as p.y + Yaxis[i];
24             if currentX and currentY are within bounds and not visited then
25                 Add a new Pair(currentX, currentY) to the queue;
26                 Set visited [currentX][currentY] to true;
27                 Set moves [currentX][currentY] to moves [p.x][p.y] + 1;
```

[N. B.] Here, Function findMove(a, b, c, d), a, b represents coordinates of starting square of Knight in chess board and c, d represents coordinates of destination of Knight.

2.4.2 Traveling Knight Problem Flowchart Are Given Below :



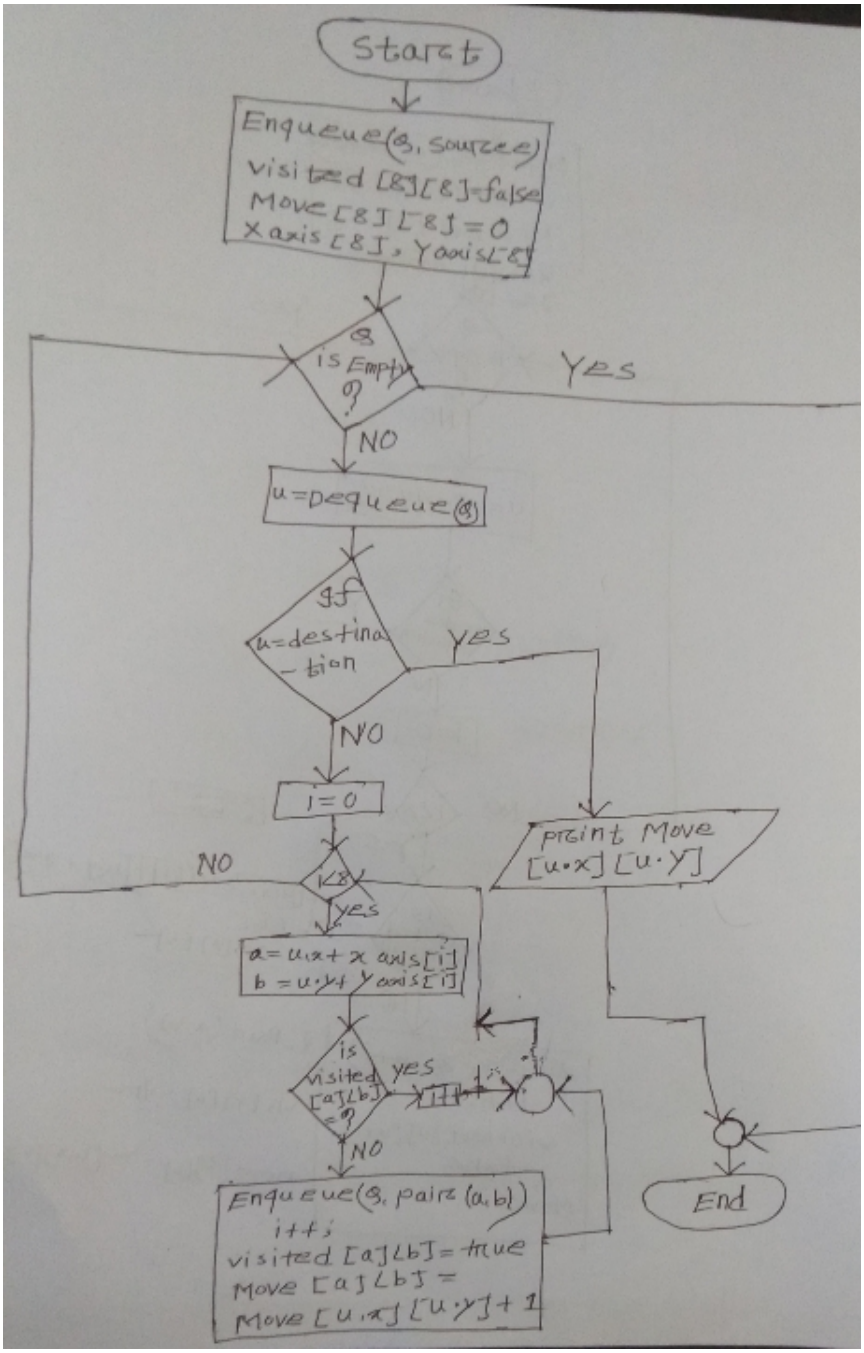


Figure 2.3: Traveling Knight Problenm Flowchart

2.4.3 Traveling Knight Problem Code Implementation Are Given Below :

```
import java.util.Arrays;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;

public class Main {

    public static int Xaxis[] = {2, 2, -2, -2, 1, -1, 1, -1};
    public static int Yaxis[] = {1, -1, 1, -1, 2, 2, -2, -2};
    public static int moves[][] = new int[8][8];
    public static boolean visited[][] = new boolean[8][8];
    public static String start, end;

    public static class Pair {

        int x, y;

        public Pair(int x, int y) {
            this.x = x;
            this.y = y;
        }
    }

    public static void clearVisitedArray() {
        for (boolean[] row : visited) {
            Arrays.fill(row, false);
        }
    }

    public static void findMove(int a, int b, int c, int d) {
        Queue<Pair> q = new LinkedList<>();
        clearVisitedArray();
        q.add(new Pair(a, b));
        visited[a][b] = true;
        moves[a][b] = 0;

        while (!q.isEmpty()) {
            Pair p = q.remove();

            if (p.x == c && p.y == d) {
                System.out.println("To get from " + start + " to " + end + " takes
                return;
            }

            for (int i = 0; i < 8; i++) {
```

```
                int currentX = p.x + Xaxis[i];
                int currentY = p.y + Yaxis[i];

                if ((currentX >= 0 && currentX < 8) && (currentY >= 0 && currentY <
                    q.add(new Pair(currentX, currentY));
                    visited[currentX][currentY] = true;
                    //System.err.println("here");
                    moves[currentX][currentY] = moves[p.x][p.y] + 1;

                }
            }
        }
    }

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

        while (scanner.hasNextLine()) {
            String x = scanner.nextLine();
            String[] parts = x.split(" ");
            start = parts[0];
            end = parts[1];
            int StartCoordinateX = x.charAt(0) - 97,
                    StartCoordinateY = x.charAt(1) - 49,
                    EndCoordinateX = x.charAt(3) - 97,
                    EndCoordinateY = x.charAt(4) - 49;
            findMove(StartCoordinateX, StartCoordinateY, EndCoordinateX, EndCoordi

        }

        scanner.close();
    }
}
```

2.4.4 Train Swapping Problem Algorithm pseudo code Are Given Below :

Algorithm 2: Merge Sort

[1] MergeSortarr: array of integers, low: integer, high: integer **if** $low < high$ **then**

$mid \leftarrow \frac{(low+high)}{2}$ MergeSortarr, low, mid MergeSortarr, mid + 1, high Mergearr, low, mid, high

Mergearr: array of integers, low: integer, mid: integer, high: integer

$n1 \leftarrow mid - low + 1$ $n2 \leftarrow high - mid$

$leftArr[1..n1+1], rightArr[1..n2+1]$

for $i \leftarrow 1$ *to* $n1$ **do**

$leftArr[i] \leftarrow arr[low + i - 1]$

for $j \leftarrow 1$ *to* $n2$ **do**

$rightArr[j] \leftarrow arr[mid + j]$

$leftArr[n1+1] \leftarrow \infty$ $rightArr[n2+1] \leftarrow \infty$

$i \leftarrow 1$ $j \leftarrow 1$

for $k \leftarrow low$ *to* $high$ **do**

$leftArr[i] \leq rightArr[j]$ $arr[k] \leftarrow leftArr[i]$ $i \leftarrow i + 1$ **else**

$arr[k] \leftarrow rightArr[j]$ $count \leftarrow count + (n1 - i + 1)$ Increment count for inversions $j \leftarrow j + 1$

2.4.5 Train Swapping Problem Flowchart Are Given Below :

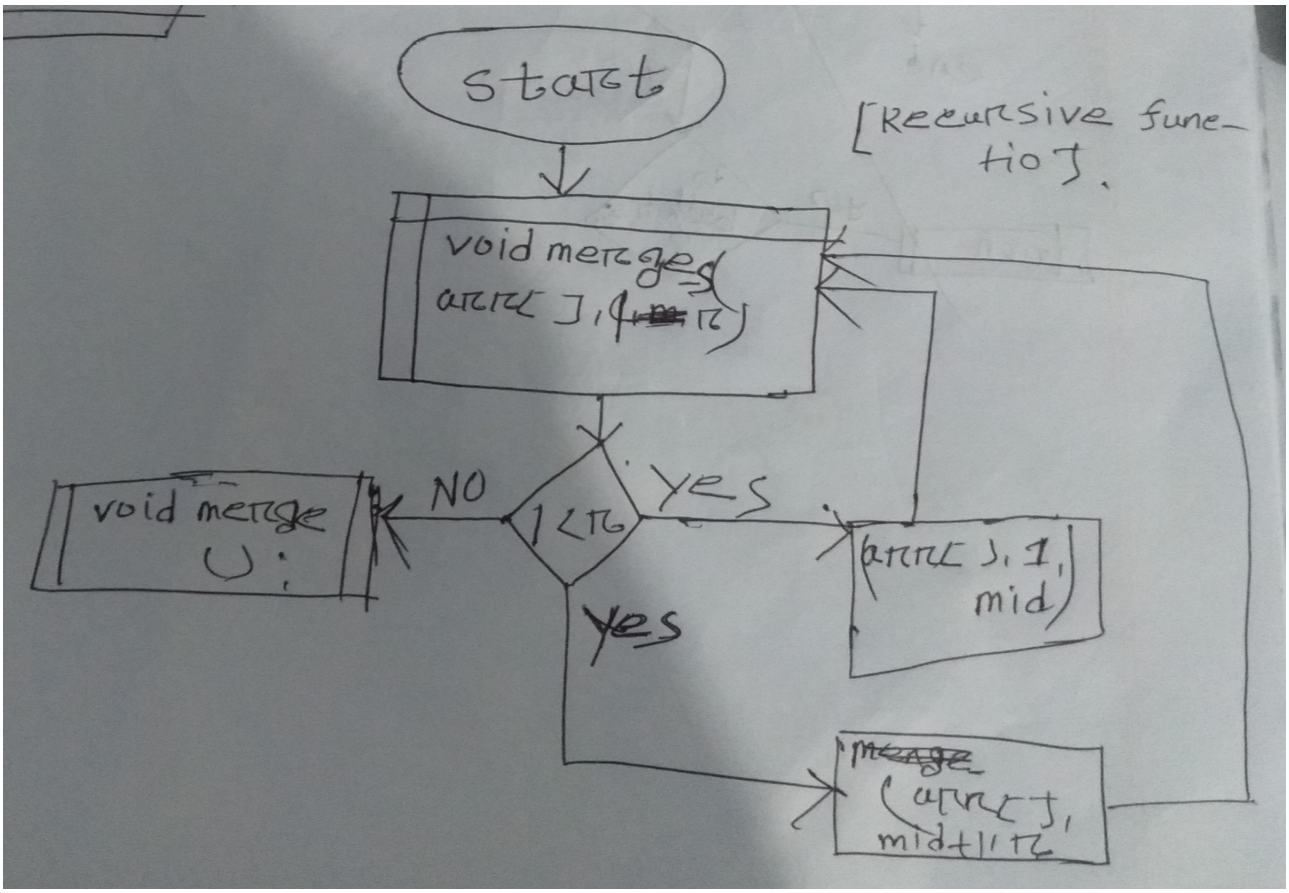


Figure 2.4: Recursive Function

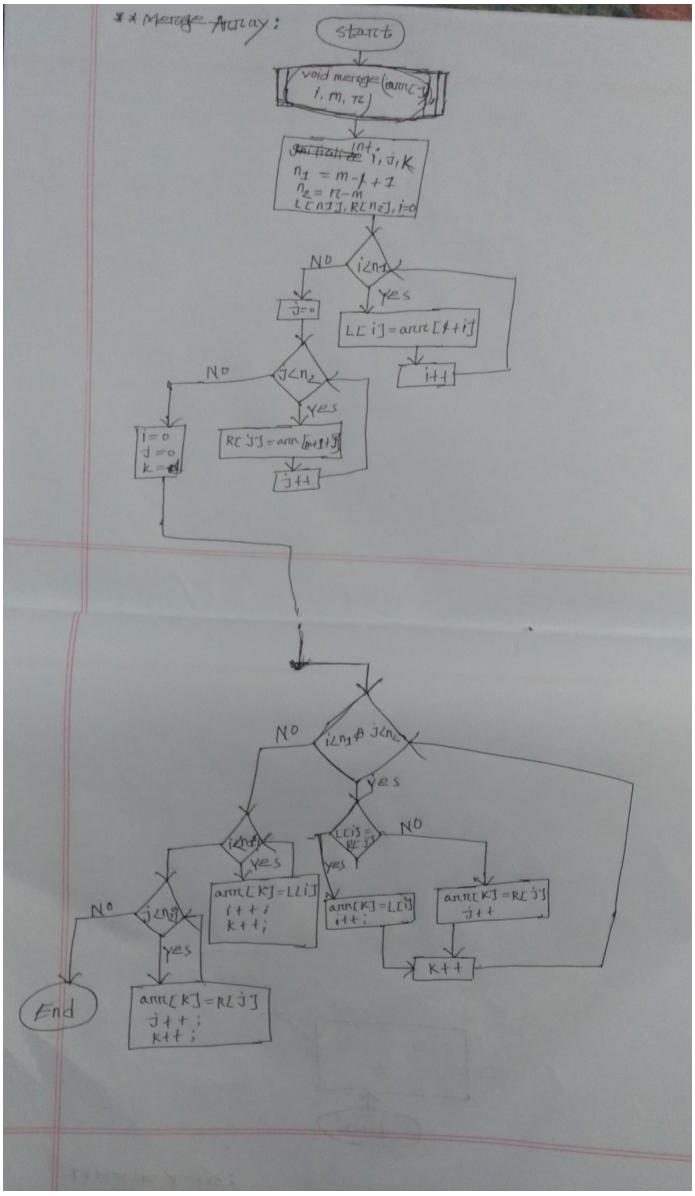


Figure 2.5: Merge Array

2.4.6 Train Swapping Problem Code Are Given Below :

```
import java.util.Scanner;

public class Main_1 {

    public static int count = 0;

    public static void mergeSort(int[] arr, int l, int r) {
        if (l < r) {
            int mid = l + (r - l) / 2;
            mergeSort(arr, l, mid);
            mergeSort(arr, mid + 1, r);
            merge(arr, l, mid, r);
        }
    }

    public static void merge(int[] arr, int l, int mid, int r) {
        int n1 = mid - l + 1;
        int n2 = r - mid;

        int[] left = new int[n1];
        int[] right = new int[n2];

        for (int i = 0; i < n1; ++i)
            left[i] = arr[l + i];
        for (int j = 0; j < n2; ++j)
            right[j] = arr[mid + 1 + j];

        int i = 0, j = 0;
        int k = l;

        while (i < n1 && j < n2) {
            if (left[i] <= right[j]) {
                arr[k] = left[i];
                i++;
            } else {
                arr[k] = right[j];
                j++;
                count += n1 - i;
            }
            k++;
        }

        while (i < n1) {
            arr[k] = left[i];
            i++;
```

```
            k++;
        }

        while (j < n2) {
            arr[k] = right[j];
            j++;
            k++;
        }
    }

    public static void main(String[] args) {
        Scanner sn = new Scanner(System.in);
        int N = sn.nextInt();

        for (int t = 0; t < N; t++) {
            int L = sn.nextInt();
            int[] arr = new int[L];
            for (int i = 0; i < L; i++) {
                arr[i] = sn.nextInt();
            }

            count = 0;
            mergeSort(arr, 0, L - 1);
            System.out.println("Optimal train swapping takes " + count + " swaps.")
        }
    }}
```

Chapter 3

Performance Evaluation

3.1 Simulation Environment/ Simulation Procedure

Details on the experimental setup and environment needed for simulating outcomes.

3.1.1 Traveling Knight Problem Simulation

If Color means it visited. Green means starting point Light blue means destination point.

```
Test Case:   C2, E5
Queue: C2,

Step 1:
Queue:  D4, E3, E1, A1, A3, B4,
```

Store moves and Visited square:

8								
7								
6								
5								
4		1		1				
3	1				1			
2			0					
1	1				1			
	a	b	c	d	e	f	g	h

Figure 3.1: Step 1

Step 2:
Queue: E3, E1, A1, A3, B4, C6, E6, F5, F3, E2, B3, B5

Store moves and Visited square:

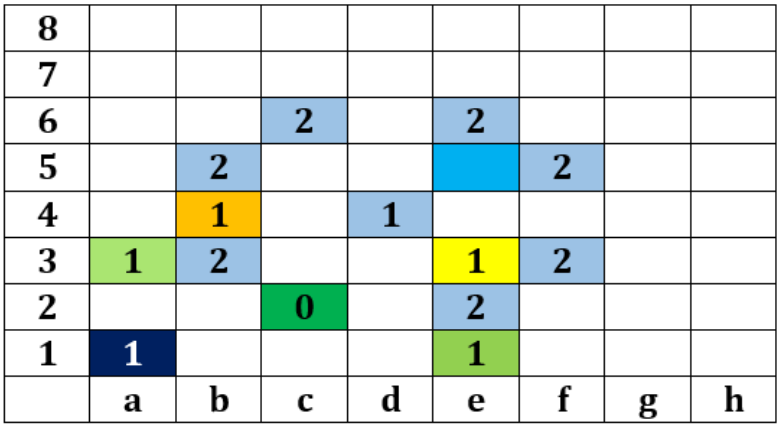


Figure 3.2: Step 2

Step 3:
Queue: E1, A1, A3, B4, C6, E6, F5, F3, E2, B3, B5, D5, G4, G2, F1, D1, C4

Store moves and Visited square:

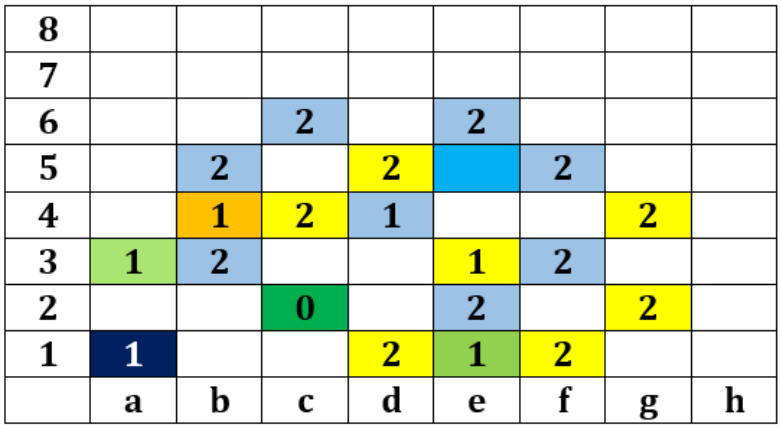


Figure 3.3: Step 3

Step 4:
Queue: A1, A3, B4, C6, E6, F5, F3, E2, B3, B5, D5, G4, G2, F1,

Store moves and Visited square:

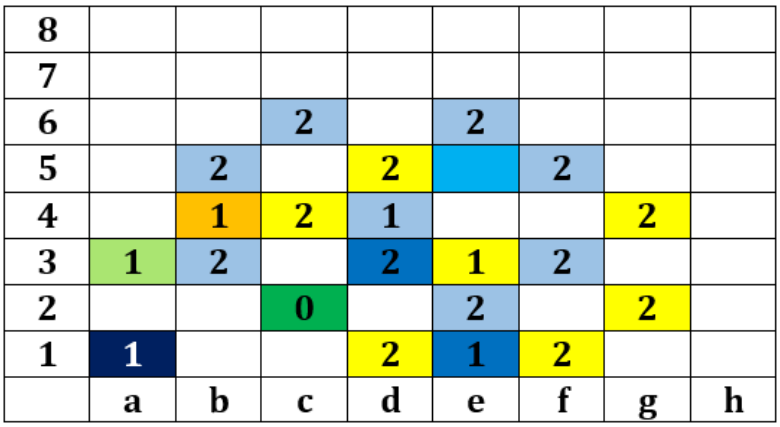


Figure 3.4: Step 4

```
Step 5:
Queue:  A3, B4, C6, E6, F5, F3, E2, B3, B5, D5, G4, G2, F1, D1,
C4, D3
```

Store moves and Visited square:

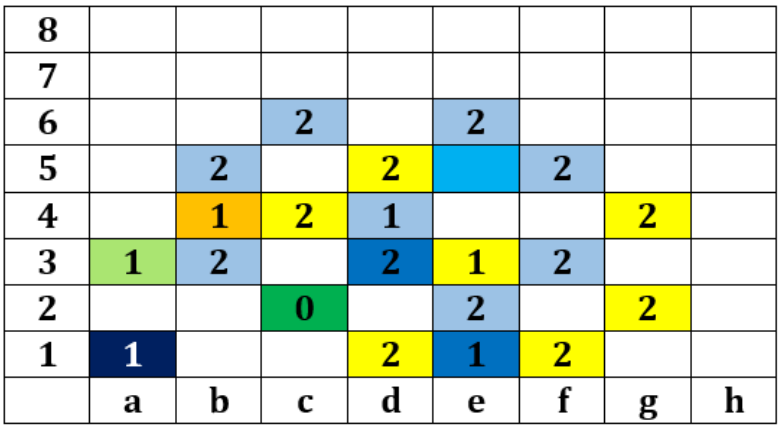


Figure 3.5: Step 5

```
Step 6:
Queue:  B4, C6, E6, F5, F3, E2, B3, B5, D5, G4, G2, F1, D1, C4,
D3, B1
```

Store moves and Visited square:

8								
7								
6			2		2			
5		2		2		2		
4		1	2	1			2	
3	1	2		2	1	2		
2			0		2		2	
1	1	2		2	1	2		
	a	b	c	d	e	f	g	h

Figure 3.6: Step 6

```
Step 7:
Queue:  C6, E6, F5, F3, E2, B3, B5, D5, G4, G2, F1, D1, C4, D3,
B1, A2, A6
```

Store moves and Visited square:

8								
7								
6	2		2		2			
5		2		2		2		
4		1	2	1			2	
3	1	2		2	1	2		
2	2		0		2		2	
1	1	2		2	1	2		
	a	b	c	d	e	f	g	h

Figure 3.7: Step 7

```
Step 8:
Queue:  E6, F5, F3, E2, B3, B5, D5, G4, G2, F1, D1, C4, D3, B1,
A2, A6, E5.
```

Store moves and Visited square:

8				3				
7					3			
6	2		2		2			
5		2		2	3	2		
4		1	2	1			2	
3	1	2		2	1	2		
2	2		0		2		2	
1	1	2		2	1	2		
	a	b	c	d	e	f	g	h

Figure 3.8: Step 8

Here, from C6, it finds the destination by 3 moves and prints moves and terminates.

3.1.2 Train Swapping Problem Simulation

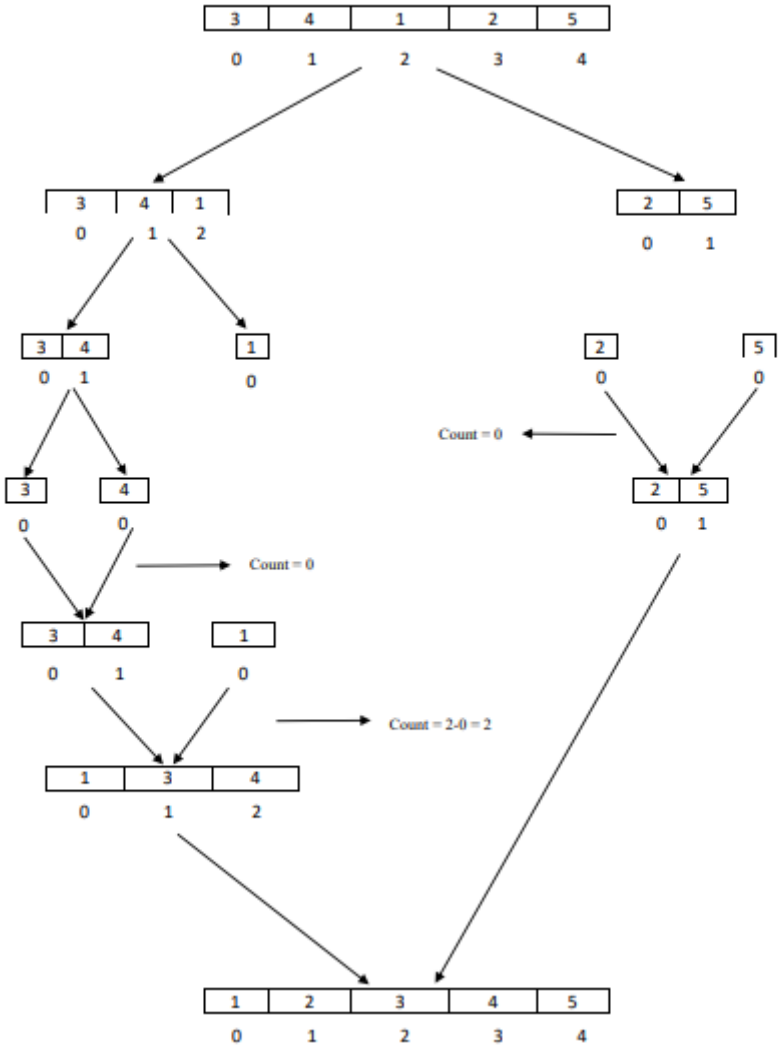


Figure 3.9: Train Swapping Simulation

3.2 Results Analysis/Testing

The results of my project are included using subsections.

3.2.1 Traveling Knight Problem Results

```
public static class Pair {

    int x, y;

    public Pair(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Figure 3.10: Pair creation

```
public static void findMove(int a, int b, int c, int d) {
    Queue<Pair> q = new LinkedList<>();
    clearVisitedArray();
    q.add(new Pair(a, b));
    visited[a][b] = true;
    moves[a][b] = 0;

    while (!q.isEmpty()) {
        Pair p = q.remove();

        if (p.x == c && p.y == d) {
            System.out.println("To get from " + start + " to " + end + " takes " + moves[p.x][p.y] + " knight moves.")
            return;
        }

        for (int i = 0; i < 8; i++) {
            int currentX = p.x + Xaxis[i];
            int currentY = p.y + Yaxis[i];

            if ((currentX >= 0 && currentX < 8) && (currentY >= 0 && currentY < 8) && !visited[currentX][currentY]) {
                q.add(new Pair(currentX, currentY));
                visited[currentX][currentY] = true;
                //System.err.println("here");
                moves[currentX][currentY] = moves[p.x][p.y] + 1;

            }
        }
    }
}
```

Figure 3.11: Modified BFS for finding moves

3.2.2 Sample Input

```
e2   e4
a1   b2
b2   c3
a1   h8
a1   h7
h8   a1
b1   c3
f6   f6
c2   e5      //custom test case
```

3.2.3 Sample Output

```
To get from e2 to e4 takes 2 knight moves.
To get from a1 to b2 takes 4 knight moves.
To get from b2 to c3 takes 2 knight moves.
To get from a1 to h8 takes 6 knight moves.
To get from a1 to h7 takes 5 knight moves.
To get from h8 to a1 takes 6 knight moves.
To get from b1 to c3 takes 1 knight moves.
To get from f6 to f6 takes 0 knight moves.
To get from c2 to e5 takes 3 knight moves. //simulation was described.
```

3.2.4 Online Judge Submission Screenshot

#48036812 | KaiSed's solution for [UVA-439]

Status	Time	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	460ms	2374	JAVA 1.8.0	2024-01-01 14:03:33	☑	☐	29098328

Figure 3.12: Online Judge verdict

3.2.5 Train Swapping Problem Results

```java
public static void merge(int[] arr, int l, int mid, int r) {
    int n1 = mid - l + 1;
    int n2 = r - mid;

    int[] left = new int[n1];
    int[] right = new int[n2];

    for (int i = 0; i < n1; ++i)
        left[i] = arr[l + i];
    for (int j = 0; j < n2; ++j)
        right[j] = arr[mid + 1 + j];

    int i = 0, j = 0;
    int k = l;

    while (i < n1 && j < n2) {
        if (left[i] <= right[j]) {
            arr[k] = left[i];
            i++;
        } else {
            arr[k] = right[j];
            j++;
            count += n1 - i;
        }
        k++;
    }
```

Figure 3.13: Count Swapping

3.2.6 Sample Input

```
5
4
3 2 1 4
3
6 4 5
5
6 7 2 1 3
2
1 2
5
3 4 1 2 5          //custom test case
```

3.2.7 Sample Output

```
Optimal train swapping takes 3 swaps.
Optimal train swapping takes 2 swaps.
Optimal train swapping takes 7 swaps.
Optimal train swapping takes 0 swaps.
Optimal train swapping takes 4 swaps.
```

3.2.8 Online Judge Submission Screenshot

#48036836 | KaiSed's solution for [UVA-299]

Status	Time	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	160ms	1694	JAVA 1.8.0	2024-01-01 14:06:29	☑	☐	29098332

Figure 3.14: Online Judge verdict

3.3 Results Overall Discussion

The outcomes of the simulations for both the Traveling Knight Problem (TKP) and the Train Swapping Problem offer valuable insights into the effectiveness and performance of the implemented algorithms.

Traveling Knight Problem (TKP):

The Breadth-First Search (BFS) algorithm employed to solve TKP demonstrated success in identifying the shortest knight tour on the chessboard. However, as the chessboard size increased, the algorithm's time complexity became more noticeable, prompting consideration of the trade-off between solution optimality and computational resources.

Train Swapping Problem:

The implementation of the merge sort algorithm for the Train Swapping Problem efficiently minimized the number of swaps required for optimal train carriage arrangements. Yet, for larger train configurations, the algorithm's time complexity emerged as a factor, suggesting the need for further optimization

3.3.1 Complex Engineering Problem Discussion

In this subsection, I delve into key attributes (P1-P7) outlined in Table 1.1, shedding light on the complexity of the engineering problems.This section provides a comprehensive understanding of the successes and considerations in addressing these multifaceted engineering problems.

Chapter 4

Conclusion

4.1 Discussion

The Traveling Knight Problem (TKP) poses an intriguing challenge in finding the shortest closed tour of knight moves on a chessboard. While the current solution efficiently navigates smaller boards, future work could explore algorithmic enhancements for larger ones. The consideration of parallelization and heuristic methods, coupled with the development of interactive visualization tools, promises to enhance both efficiency and user understanding. In the case of the Train Swapping Problem, the current algorithm, reliant on the merge sort method, demonstrates sensitivity to input permutations. Future work may involve refining the algorithmic approach to address this sensitivity. Additionally, adapting the solution to handle diverse data types and dynamically adjusting strategies based on input characteristics could broaden its applicability. A parallel effort to improve user interfaces for visualizing the train rearrangement process aims to make the solution more accessible and user-friendly. These discussions highlight avenues for refining both solutions and adapting them to real-world challenges efficiently.

4.2 Limitations

4.2.1 Limitations Of Traveling Knight Problem (TKP):

- **Computational Complexity:** The algorithm's scalability might be challenged for very large chessboards due to its time complexity.
- **Memory Consumption:** Scaling to larger boards may increase memory usage, impacting the algorithm's efficiency.
- **Solution Uniqueness:** For certain square pairs, multiple solutions may exist, making the solution non-unique.

4.2.2 Limitations Of Train Swapping Problem:

- **Algorithm Sensitivity:** Efficiency of the merge sort algorithm can vary based on input permutations.
- **Dependency on Input Order:** Performance is influenced by the initial carriage order, leading to suboptimal outcomes in nearly sorted scenarios.
- **Limited to Integer Values:** The algorithm assumes integer carriage numbers, restricting its use for non-integer identifiers or complex data structures.

4.3 Scope Of Future Work

In future work for the Traveling Knight Problem, a focus on algorithmic efficiency for larger chessboards, potentially through parallelization or heuristic methods, could enhance the solution's scalability. Exploring alternative approaches and developing interactive visualization tools would further improve overall robustness and user-friendliness. Concerning the Train Swapping Problem, future work could involve refining the algorithm's sensitivity, adapting it to diverse data types, and dynamically adjusting strategies based on input characteristics. Additionally, efforts to improve user interfaces for visualizing the train rearrangement process aim to enhance accessibility and usability. These collective endeavors aim to make both solutions more versatile and efficient in addressing real-world challenges..

References

[1] Competitive Programming Problem 1. https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=6&page=show_problem&problem=380.

[2] Competitive Programming Problem 2. https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=4&page=show_problem&problem=235.

[3] Kathy Sierra Bert Bates. Head first java.

[4] Herbert Schildt. Java the complete reference.

[5] Learning Java. https://www.javatpoint.com/mysql-tutorial.