Name: Vaishali Ramesh Kale

Email ID: kalevaishalir16@gmail.com

Day 12:

Task 1: Bit Manipulation Basics Create a function that counts the number of set bits (1s) in the binary representation of an integer. Extend this to count the total number of set bits in all integers from 1 to n.

Solution::::

Explanation

1. countSetBits(int number):
- This function takes an integer and counts the number of 1s in its binary representation.
- It uses a loop that continues until the number becomes 0.
- Inside the loop, it adds the least significant bit of the number to the count and then right shifts the number.

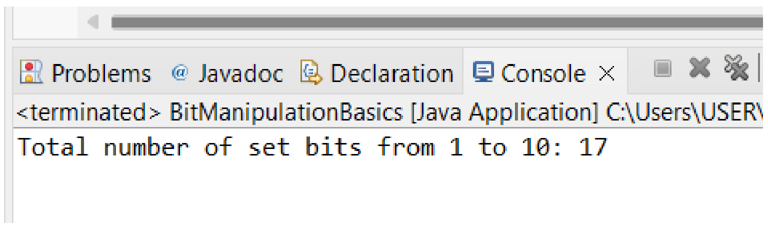
2. countTotalSetBits(int n):
- This function takes an integer n
- n and counts the total number of set bits in all integers from 1 to n
- n.It uses a loop to iterate from 1 to n
- n, calling countSetBits for each number and summing the results.

CODE::::

```
package com.wipro.java;

public class BitManipulationBasics {

// Method to count the number of set bits in an integer
public static int countSetBits(int number) {
int count = 0;
while (number > 0) {
count += number & 1;
number >>= 1;
}
return count;
}
```

```java
// Method to count the total number of set bits in all integers from 1 to n
public static int countTotalSetBits(int n) {
int totalCount = 0;
for (int i = 1; i <= n; i++) {
totalCount += countSetBits(i);
}
return totalCount;
}

public static void main(String[] args) {
int n = 10; // Example range
int totalSetBitsCount = countTotalSetBits(n);
System.out.println("Total number of set bits from 1 to " + n + ": " + totalSetBitsCount);
}
}
```

OUTPUT::::



```
<terminated> BitManipulationBasics [Java Application] C:\Users\USER\
Total number of set bits from 1 to 10: 17
```

Task 2: Unique Elements Identification Given an array of integers where every element appears twice except for two, write a function that efficiently finds these two non-repeating elements using bitwise XOR operations.

Solution::::

Explanation

1. XOR of All Elements:

- XOR-ing all elements will cancel out all the elements that appear twice, leaving the XOR of the two unique elements.

2. Finding a Set Bit:

- The expression xor & -xor isolates the rightmost set bit in the XOR result. This bit is guaranteed to be different between the two unique elements.

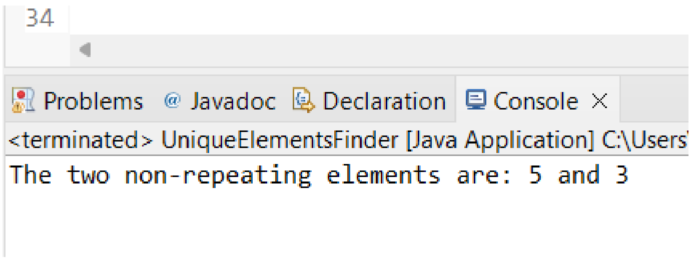
3. Partition and XOR:

- By partitioning the numbers based on the isolated bit, we ensure that the two unique numbers fall into different groups. XOR-ing the elements in each group will cancel out the duplicates and leave us with the two unique numbers.

CODE::::

```
package com.wipro.java;

public class UniqueElementsFinder {

public static int[] findUniqueElements(int[] arr) {
// Step 1: XOR all the elements to get the XOR of the two unique numbers
int xor = 0;
for (int num : arr) {
xor ^= num;
}

// Step 2: Find a set bit in the XOR result
int setBit = xor & -xor; // This isolates the rightmost set bit

// Step 3: Partition the array into two groups and XOR the elements within each group
int x = 0, y = 0;
for (int num : arr) {
if ((num & setBit) == 0) {
x ^= num;
} else {
y ^= num;
}
}

return new int[]{x, y};
}

public static void main(String[] args) {
int[] arr = {4, 1, 2, 1, 2, 3, 4, 5}; // Example array
int[] uniqueElements = findUniqueElements(arr);
System.out.println("The two non-repeating elements are: " + uniqueElements[0] + " and " +
uniqueElements[1]);
}
}
```

OUTPUT::::



```
<terminated> UniqueElementsFinder [Java Application] C:\Users
The two non-repeating elements are: 5 and 3
```

****

Day 13 and 14:

Task 1: Tower of Hanoi Solver Create a program that solves the Tower of Hanoi puzzle for n disks. The solution should use recursion to move disks between three pegs (source, auxiliary, and destination) according to the game's rules. The program should print out each move required to solve the puzzle.

Solution::::

Explanation

1. Main Method:
- The main method initializes the number of disks n to 3 and calls the hanoi method with the rod labels "A", "B", and "C".

2. hanoi Method:
- This is a recursive method to solve the Tower of Hanoi puzzle.
- Base Case: When n is 1, it directly moves the disk from the rodFrom to the rodTo and prints the move.

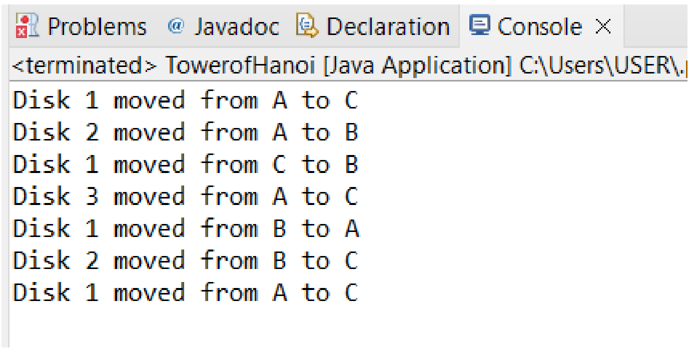
3. Recursive Case:
- It first moves n-1 disks from rodFrom to rodAux using rodTo as an auxiliary rod.
- Then, it moves the nth disk from rodFrom to rodTo.
- Finally, it moves the n-1 disks from rodAux to rodTo using rodFrom as an auxiliary rod.

CODE::::

```
package com.wipro.computalgo;

public class TowerofHanoi {
```

```
public static void main(String[] args) {
// Number of disks
int n = 3;

// Calling the hanoi function with n disks
hanoi(n, "A", "B", "C");
}

/**
* Recursive method to solve Tower of Hanoi puzzle
*
* @param n        Number of disks
* @param rodFrom  The starting rod
* @param rodAux   The auxiliary rod
* @param rodTo    The destination rod
*/
private static void hanoi(int n, String rodFrom, String rodAux, String rodTo) {
if (n == 1) {
System.out.println("Disk 1 moved from " + rodFrom + " to " + rodTo);
return;
}

// Move n-1 disks from rodFrom to rodAux using rodTo as auxiliary
hanoi(n - 1, rodFrom, rodTo, rodAux);

// Move the nth disk from rodFrom to rodTo
System.out.println("Disk " + n + " moved from " + rodFrom + " to " + rodTo);

// Move the n-1 disks from rodAux to rodTo using rodFrom as auxiliary
hanoi(n - 1, rodAux, rodFrom, rodTo);
}
}
```

OUTPUT::::



```
<terminated> TowerofHanoi [Java Application] C:\Users\USER\.
Disk 1 moved from A to C
Disk 2 moved from A to B
Disk 1 moved from C to B
Disk 3 moved from A to C
Disk 1 moved from B to A
Disk 2 moved from B to C
Disk 1 moved from A to C
```

Task 2: Traveling Salesman Problem Create a function int FindMinCost(int[,] graph) that takes a 2D array representing the graph where graph[i][j] is the cost to travel from city i to city j. The function should return the minimum cost to visit all cities and return to the starting city. Use dynamic programming for this solution.

Solution::::

1. Initialization:
- We initialize the dp table with Long.MAX_VALUE to represent infinite costs initially.
- We set dp[0][1] to 0 because starting at city 0 incurs no cost.

2. Dynamic Programming Loop:
- For each possible subset of cities (mask), and for each city u in this subset, we try to add another city v not in the subset.
- If dp[u][mask] is not Long.MAX_VALUE (meaning it's a valid path) and graph[u][v] is not zero (meaning there's a valid path from u to v), we update dp[v][mask | (1 << v)] to the minimum cost of reaching v by extending the current subset.

3. Debugging:

- The code prints the dp table for debugging purposes. This helps verify if the DP table is being populated correctly.

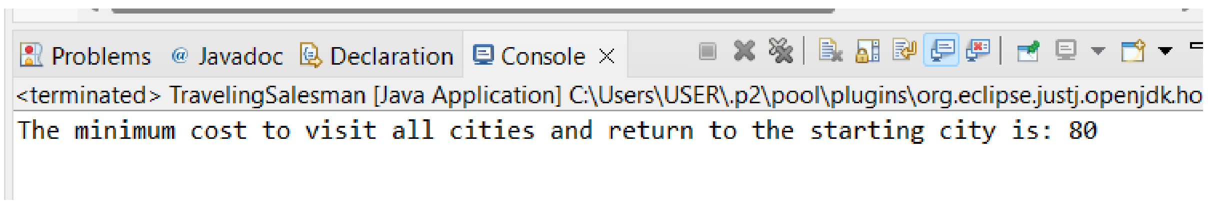
4. Result Calculation:
- The minimum cost to visit all cities and return to the starting city is found by checking the costs in the DP table for all cities and adding the cost to return to the starting city.

CODE::::

```
package com.wipro.computalgo;

public class TravelingSalesman {

// Function to find the minimum cost to visit all cities and return to the starting city
public static long findMinCost(int[][] graph) {
int n = graph.length;
long[][] dp = new long[n][(1 << n)];

// Initialize the dp array with a large value
for (int i = 0; i < n; i++) {
for (int j = 0; j < (1 << n); j++) {
```

```
dp[i][j] = Long.MAX_VALUE;
}
}

// Starting point, the cost to visit starting point is 0
dp[0][1] = 0;

// Iterate over all subsets of nodes
for (int mask = 1; mask < (1 << n); mask++) {
for (int u = 0; u < n; u++) {
if ((mask & (1 << u)) != 0) { // if u is in the subset represented by mask
for (int v = 0; v < n; v++) {
if ((mask & (1 << v)) == 0) { // if v is not in the subset
if (dp[u][mask] != Long.MAX_VALUE && graph[u][v] != 0) {
dp[v][mask | (1 << v)] = Math.min(dp[v][mask | (1 << v)], dp[u][mask] + graph[u][v]);
}
}
}
}
}
}

// Find the minimum cost to return to the starting point
long minCost = Long.MAX_VALUE;
for (int i = 1; i < n; i++) {
if (dp[i][(1 << n) - 1] != Long.MAX_VALUE && graph[i][0] != 0) {
minCost = Math.min(minCost, dp[i][(1 << n) - 1] + graph[i][0]);
}
}

return minCost;
}

public static void main(String[] args) {
int[][] graph = {
{ 0, 10, 15, 20 },
{ 10, 0, 35, 25 },
{ 15, 35, 0, 30 },
{ 20, 25, 30, 0 }
};

long minCost = findMinCost(graph);
System.out.println("The minimum cost to visit all cities and return to the starting city is: " +
minCost);
}
}
```

OUTPUT::::



```
<terminated> TravelingSalesman [Java Application] C:\Users\USER\.p2\pool\plugins\org.eclipse.justj.openjdk.ho
The minimum cost to visit all cities and return to the starting city is: 80
```

Task 3: Job Sequencing Problem Define a class Job with properties int Id, int Deadline, and int Profit. Then implement a function List<Job> JobSequencing(List<Job> jobs) that takes a list of jobs and returns the maximum profit sequence of jobs that can be done before the deadlines. Use the greedy method to solve this problem.

Solution::::

Explanation:

1. Job Class: The Job class has three properties: id, deadline, and profit, with a constructor to initialize these properties.

2. Job Sequencing Function:

- The function sorts the list of jobs by their profit in descending order.
- It then finds the maximum deadline to determine the size of the result and slot arrays.
- It iterates through the sorted jobs and attempts to schedule each job in the latest available slot before its deadline.
- It adds successfully scheduled jobs to the result array and keeps track of filled slots using a boolean array.

3. Main Method:

- Initializes a list of jobs and calls the jobSequencing function.
- Prints the job sequence and the total profit.

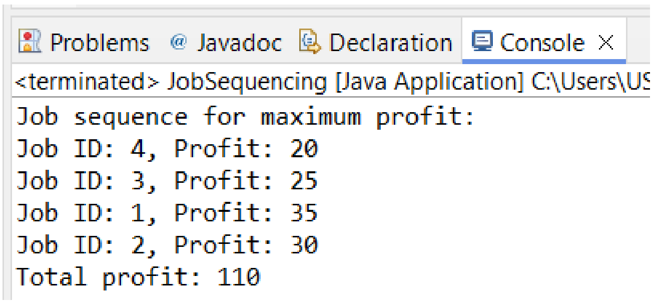
CODE::::

```
package com.wipro.computalgo;
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Job {
int id;
int deadline;
int profit;

public Job(int id, int deadline, int profit) {
this.id = id;
this.deadline = deadline;
this.profit = profit;
}
}

public class JobSequencing {

public static List<Job> jobSequencing(List<Job> jobs) {
// Sort jobs by decreasing profit
Collections.sort(jobs, (a, b) -> b.profit - a.profit);

int maxDeadline = 0;
for (Job job : jobs) {
maxDeadline = Math.max(maxDeadline, job.deadline);
}

// Create an array to keep track of free time slots and initialize it to null
Job[] result = new Job[maxDeadline];
```

```
boolean[] slot = new boolean[maxDeadline];

// Iterate through all given jobs
for (Job job : jobs) {
// Find a free slot for this job (starting from the last possible slot)
for (int j = Math.min(maxDeadline - 1, job.deadline - 1); j >= 0; j--) {
if (!slot[j]) {
slot[j] = true;
result[j] = job;
break;
}
}
}

// Collect all jobs that were successfully scheduled
List<Job> jobSequence = new ArrayList<>();
for (Job job : result) {
if (job != null) {
jobSequence.add(job);
}
}

return jobSequence;
}

public static void main(String[] args) {
List<Job> jobs = new ArrayList<>();
jobs.add(new Job(1, 3, 35));
jobs.add(new Job(2, 4, 30));
jobs.add(new Job(3, 4, 25));

jobs.add(new Job(4, 2, 20));

jobs.add(new Job(5, 3, 15));

jobs.add(new Job(6, 1, 12));


List<Job> jobSequence = jobSequencing(jobs);


System.out.println("Job sequence for maximum profit:");

for (Job job : jobSequence) {

System.out.println("Job ID: " + job.id + ", Profit: " + job.profit);

}


int totalProfit = jobSequence.stream().mapToInt(job -> job.profit).sum();

System.out.println("Total profit: " + totalProfit);

}

}
```

OUTPUT::::



```
<terminated> JobSequencing [Java Application] C:\Users\US
Job sequence for maximum profit:
Job ID: 4, Profit: 20
Job ID: 3, Profit: 25
Job ID: 1, Profit: 35
Job ID: 2, Profit: 30
Total profit: 110
```

****
****

Day 15 and 16:

Task 1: Knapsack Problem Write a function int Knapsack(int W, int[] weights, int[] values) in C# that determines the maximum value of items that can fit into a knapsack with a capacity W. The function should handle up to 100 items. Find the optimal way to fill the knapsack with the given items to achieve the maximum total value. You must consider that you cannot break items, but have to include them whole.

Solution::::

Explanation:

1. Main Method:
- Defines the capacity of the knapsack and the weights and values of the items.
- Calls the knapsack method to find the maximum value that can be obtained.
- Prints the maximum value and the items included in the knapsack.

2. knapsack Method:
- Uses a 2D array dp to store the maximum value that can be obtained with a given capacity and a subset of items.
- Iterates through the items and capacities to fill the array based on whether the current item is included or not.
- Calls the findItemsIncluded method to find which items are included in the optimal solution.

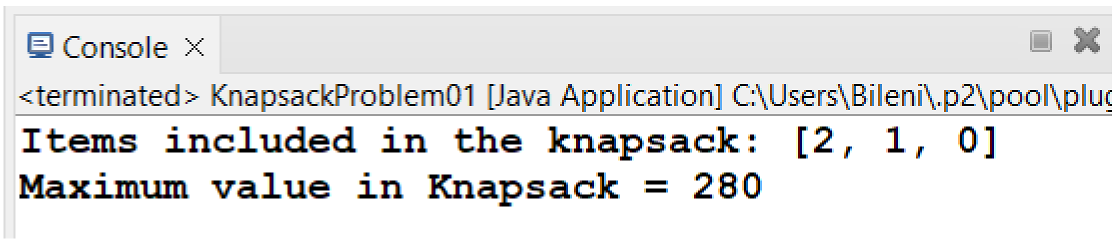
3. findItemsIncluded Method:
- Traces back through the 2D array dp to find which items are included in the optimal solution.
- Returns a list of the indices of the included items.

CODE::::

```
package com.wipro.dynamicprog;

import java.util.ArrayList;
import java.util.List;

public class KnapsackProblem01 {

public static int knapsack(int W, int[] weights, int[] values) {
int n = weights.length;
int[][] dp = new int[n + 1][W + 1];

// Fill dp array
for (int i = 0; i <= n; i++) {
```

```
for (int w = 0; w <= W; w++) {
if (i == 0 || w == 0) {
dp[i][w] = 0;
} else if (weights[i - 1] <= w) {
dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
} else {
dp[i][w] = dp[i - 1][w];
}
}
}

// Backtrack to find the items included
List<Integer> itemsIncluded = new ArrayList<>();
int w = W;
for (int i = n; i > 0 && w > 0; i--) {
if (dp[i][w] != dp[i - 1][w]) {
itemsIncluded.add(i - 1);
w -= weights[i - 1];
}
}

// Print the items included
System.out.println("Items included in the knapsack: " + itemsIncluded);

return dp[n][W];
}

public static void main(String[] args) {
int W = 100;
int[] weights = { 10, 20, 30 };
int[] values = { 60, 100, 120 };

System.out.println("Maximum value in Knapsack = " + knapsack(W, weights, values));
}
}
```

OUTPUT::::



```
<terminated> KnapsackProblem01 [Java Application] C:\Users\Bileni\.p2\pool\plug
Items included in the knapsack: [2, 1, 0]
Maximum value in Knapsack = 280
```

Task 2: Longest Common Subsequence Implement int LCS(string text1, string text2) to find the length of the longest common subsequence between two strings.

Solution::::

Explanation:

1. Initialization:

- dp[i][j] is a 2D array where dp[i][j] represents the length of the longest common subsequence of text1[0..i-1] and text2[0..j-1].
- If either string is empty (i == 0 or j == 0), the LCS length is 0.

2. Filling the dp Table:

- If the characters of text1 and text2 at the current indices match (text1.charAt(i - 1) == text2.charAt(j - 1)), the LCS length is 1 + dp[i - 1][j - 1].
- Otherwise, the LCS length is the maximum of the LCS lengths by either excluding the current character of text1 or text2.

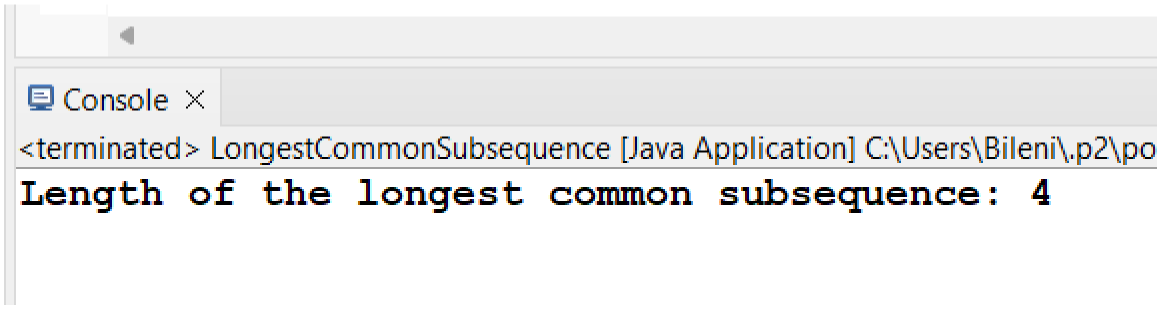
3. Result:

- The length of the LCS of text1 and text2 is stored in dp[m][n].

CODE::::

```
package com.wipro.dynamicprog;

public class LongestCommonSubsequence {

public static void main(String[] args) {
String text1 = "babbab";
String text2 = "abaaba";

int length = LCS(text1, text2);
System.out.println("Length of the longest common subsequence: " + length);
}

private static int LCS(String text1, String text2) {
int m = text1.length();
int n = text2.length();
int[][] dp = new int[m + 1][n + 1];
for (int i = 0; i <= m; i++) {
for (int j = 0; j <= n; j++) {
if (i == 0 || j == 0) {
dp[i][j] = 0;
} else if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
dp[i][j] = 1 + dp[i - 1][j - 1];
} else {
dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
}
}
}
return dp[m][n];
}
}
```

OUTPUT::::



```
<terminated> LongestCommonSubsequence [Java Application] C:\Users\Bileni\.p2\po
Length of the longest common subsequence: 4
```

****

****

Day 16 and 17:

Task 1: The Knight's Tour Problem Create a function bool SolveKnightsTour(int[,] board, int moveX, int moveY, int moveCount, int[] xMove, int[] yMove) that attempts to solve the Knight's Tour problem using backtracking. The function should return true if a solution exists and false otherwise. The board represents the chessboard, moveX and moveY are the current coordinates of the knight, moveCount is the current move count, and xMove[], yMove[] are the possible next moves for the knight. Fill the chessboard such that the knight visits every square exactly once. Keep the chessboard size to 8x8.

Solution::::

Explanation:

- The code structure and functionality are identical to the C# version. Here's a breakdown.

1. solveKnightsTour Function:

- Takes the board (2D integer array representing the chessboard), current knight position (moveX, moveY), current move count (moveCount), and possible knight moves (xMove, yMove arrays) as input.
- Implements the backtracking logic to find a solution.
- Returns true if a solution exists, false otherwise.

2. isValidMove Function:

- Checks if a given position (x, y) is within the board boundaries and is an unvisited square (marked as 0 in the board array).
- Returns true if the move is valid, false otherwise.

3. Usage:

- Define the chessboard (board) as an 8x8 integer array.
- Define the possible knight moves (xMove and yMove arrays) as {2, 2, 1, 1, -1, -1, -2, -2} and {-1, 1, -2, 2, -2, 2, -1, 1} respectively.
- Call the solveKnightsTour function with the starting position (e.g., moveX = 0, moveY = 0), initial move count (moveCount = 1), and the defined move arrays.
- If solveKnightsTour returns true, a solution exists, and you can print the visited squares from the board array. Otherwise, no solution is found for the given starting position.

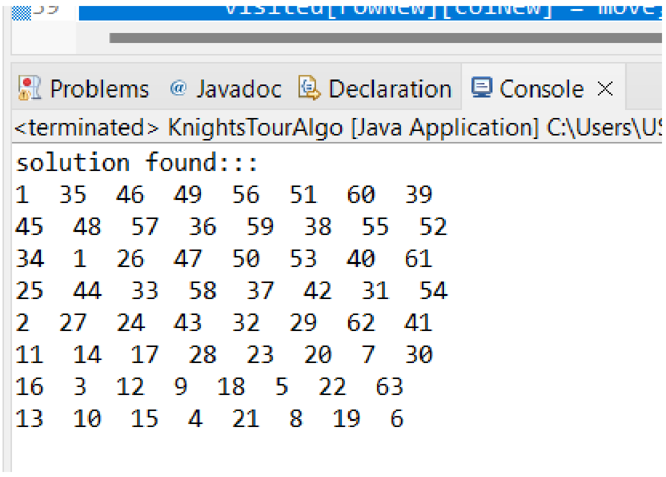
CODE::::

```
package com.wipro.backtrackingalgo;

public class KnightsTourAlgo {

// Possible moves of a Knight
```

```
int[] pathRow = {2, 2, 1, 1, -1, -1, -2, -2};
int[] pathCol = {-1, 1, -2, 2, -2, 2, -1, 1};

public static void main(String[] args) {
KnightsTourAlgo knightTour = new KnightsTourAlgo();
int[][] visited = new int[8][8];
visited[0][0] = 1;  // Mark the starting position (0, 0) as visited

if (!knightTour.findKnightTour(visited, 0, 0, 1)) {
System.out.println("Solution Not Available :(");
}
}

private boolean findKnightTour(int[][] visited, int row, int col, int move) {
if (move == 64) {
System.out.println("solution found:::");
// Print the solution (visited array)
for (int i = 0; i < 8; i++) {
for (int j = 0; j < 8; j++) {
System.out.print(visited[i][j] +"  ");
}
System.out.println();
}

return true;
} else {
// Try all possible moves from current position
for (int i = 0; i < 8; i++) {
int rowNew = row + pathRow[i];
int colNew = col + pathCol[i];

if (isValidMove(visited, rowNew, colNew)) {
visited[rowNew][colNew] = move; // Mark the new position as visited with current move
number
if (findKnightTour(visited, rowNew, colNew, move + 1)) {
return true;  // If a solution is found from the new position, return true
} else {
visited[rowNew][colNew] = 0;  // Backtrack: unmark the new position if solution not found
}
}
}
// No valid move found from current position, backtrack
return false;
}
}

private boolean isValidMove(int[][] visited, int rowNew, int colNew) {
```

```
if (rowNew >= 0 && rowNew < 8 && colNew >= 0 && colNew < 8 &&
visited[rowNew][colNew] == 0) {
return true;
}
return false;
}
}
```

OUTPUT::::



```
<terminated> KnightsTourAlgo [Java Application] C:\Users\US
solution found:::
1   35   46   49   56   51   60   39
45   48   57   36   59   38   55   52
34   1   26   47   50   53   40   61
25   44   33   58   37   42   31   54
2   27   24   43   32   29   62   41
11   14   17   28   23   20   7   30
16   3   12   9   18   5   22   63
13   10   15   4   21   8   19   6
```

Task 2: Rat in a Maze mplement a function bool SolveMaze(int[,] maze) that uses backtracking to find a path from the top left corner to the bottom right corner of a maze. The maze is represented by a 2D array where 1s are paths and 0s are walls. Find a rat's path through the maze. The maze size is 6x6.

Solution::::

Explanation:

1. findPathInMaze Method:

- **Recursively tries to find the path through the maze.**
- **Checks if the current cell (row, col) is the destination. If so, it prints the solution.**
- **Iterates through possible moves (right, left, down, up) and checks if the move is valid.**
- **If the move is valid, it marks the cell as visited and recurses to the next cell.**
- **If the move does not lead to a solution, it backtracks by unmarking the cell.**

2. isValidMove Method:

- Checks if the move to (rowNew, colNew) is within bounds, on a path (maze[rowNew][colNew] == 1), and not yet visited.

3. printSolution Method:

- Prints the maze with the path taken by the rat.

4. main Method:

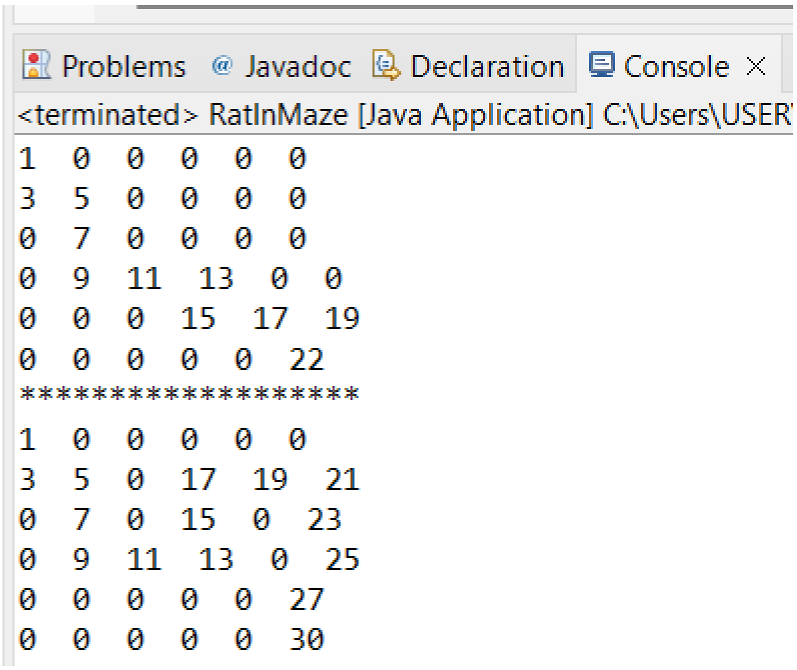
- Initializes a 6x6 maze and the visited array.
- Calls findPathInMaze to find and print the path.

CODE::::

```
package com.wipro.backtrackingalgo;

public class RatInMaze {


int[] pathRow = { 0, 0, 1, -1 };

int[] pathCol = { 1, -1, 0, 0 };


private void findAllPathsInMaze(int[][] maze, int[][] visited, int row, int col, int destRow, int destCol, int move) {

if (row == destRow && col == destCol) {

visited[row][col] = move;

printSolution(visited);

visited[row][col] = 0; // Reset for finding other paths

return;

} else {

for (int index = 0; index < pathRow.length; index++) {

int rowNew = row + pathRow[index];

int colNew = col + pathCol[index];


if (isValidMove(maze, visited, rowNew, colNew)) {
```

```
move++;

visited[rowNew][colNew] = move;

findAllPathsInMaze(maze, visited, rowNew, colNew, destRow, destCol, move + 1);


move--;

visited[rowNew][colNew] = 0; // Backtrack

}

}

}

}


private boolean isValidMove(int[][] maze, int[][] visited, int rowNew, int colNew) {

return (rowNew >= 0 && rowNew < maze.length && colNew >= 0 && colNew <
maze[0].length && maze[rowNew][colNew] == 1 && visited[rowNew][colNew] == 0);

}


private void printSolution(int[][] visited) {

for (int i = 0; i < visited.length; i++) {

for (int j = 0; j < visited[0].length; j++) {

System.out.print(visited[i][j] + "  ");

}

System.out.println();

}

System.out.println("*******");

}


public static void main(String[] args) {

int[][] maze = {

{1, 0, 0, 0, 0, 0},

{1, 1, 0, 1, 1, 1},
{0, 1, 0, 1, 0, 1},

{0, 1, 1, 1, 0, 1},

{1, 0, 0, 1, 1, 1},

{1, 1, 1, 0, 0, 1}

};

int[][] visited = new int[6][6];

visited[0][0] = 1;


RatInMaze ratInMaze = new RatInMaze();

ratInMaze.findAllPathsInMaze(maze, visited, 0, 0, 5, 5, 2);

}

}
```

OUTPUT::::



```
<terminated> RatInMaze [Java Application] C:\Users\USER
1  0  0  0  0  0
3  5  0  0  0  0
0  7  0  0  0  0
0  9  11  13  0  0
0  0  0  15  17  19
0  0  0  0  0  22
*******************
1  0  0  0  0  0
3  5  0  17  19  21
0  7  0  15  0  23
0  9  11  13  0  25
0  0  0  0  0  27
0  0  0  0  0  30
```

Task 3: N Queen Problem Write a function bool SolveNQueen(int[,] board, int col) in C# that places N queens on an N x N chessboard so that no two queens attack each other using backtracking. Place N queens on the board such that no two queens can attack each other. Use a standard 8x8 chessboard.

Solution::::

Explanation:

1. printSolution Method:

•Prints the chessboard with queens placed.

2. isSafe Method:

•Checks if placing a queen at position (row, col) is safe, i.e., no other queen can attack it.

3. solveNQueen Method:

• Tries to place queens column by column using backtracking.

• Calls isSafe to check if a queen can be placed in the current row of the current column.

•If it is safe, the queen is placed, and the function recurses to place the next queen.

•If placing the queen does not lead to a solution, it backtracks by removing the queen and trying the next row.

4. main Method:

• Initializes the board and calls the solveNQueen method.

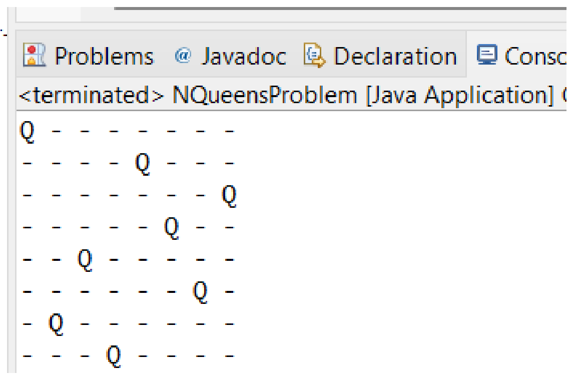
• Prints the solution if it exists; otherwise, it prints that no solution exists.

CODE::::

```
package com.wipro.backtrackingalgo;

public class NQueensProblem {

public static void main(String[] args) {

int size = 8;

boolean[][] board = new boolean[size][size];

NQueensProblem nQueensProblem = new NQueensProblem();

if (!nQueensProblem.nQueen(board, size, 0)) {

System.out.println("No solution found :( ");

}

}
```

```
private boolean nQueen(boolean[][] board, int size, int row) {
if (row == size) {
printBoard(board, size);
return true;
} else {
for (int col = 0; col < size; col++) {
if (isValidCell(board, size, row, col)) {
board[row][col] = true; // Place the queen

if (nQueen(board, size, row + 1)) {
return true;
}
board[row][col] = false; // Backtrack
}
}
}
return false;

private boolean isValidCell(boolean[][] board, int size, int row, int col) {
// check column
for (int i = 0; i < row; i++) {
if (board[i][col]) {
return false;
}
}
// check upper left diagonal
for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
if (board[i][j]) {
return false;
}
}

// check upper right diagonal
for (int i = row, j = col; i >= 0 && j < size; i--, j++) {
if (board[i][j]) {
return false;
}
}
return true;
}
private void printBoard(boolean[][] board, int size) {
for (int i = 0; i < size; i++) {
for (int j = 0; j < size; j++) {
System.out.print(board[i][j] ? "Q " : "- ");
}
System.out.println();
}
}
}
```

OUTPUT::::



```
<terminated> NQueensProblem [Java Application]
Q - - - - - - -
- - - - Q - - -
- - - - - - - Q
- - - - - Q - -
- - Q - - - - -
- - - - - - Q -
- Q - - - - - -
- - - Q - - - -
```

****
****