DS Assignment

Task 4: Rabin-Karp Substring Search

Implement the Rabin-Karp algorithm for substring search using a rolling hash. Discuss the impact of hash collisions on the algorithm's performance and how to handle them.

```
public class RabinKarp {
    public static int prime = 101;
    public static int base = 256;
    public static List<Integer> search(String text, String pattern) {
        List<Integer> occurrences = new ArrayList<>();
        int n = text.length();
        int m = pattern.length();



        long patternHash = 0;
        for (int i = 0; i < m; i++) {
            patternHash = (patternHash * base + pattern.charAt(i)) %
prime;
        }



        long textHash = 0;
        for (int i = 0; i < m; i++) {
            textHash = (textHash * base + text.charAt(i)) % prime;
        }



        for (int i = 0; i <= n - m; i++) {
            if (patternHash == textHash) {

                if (text.substring(i, i + m).equals(pattern)) {
                    occurrences.add(i);
                }
            }



            if (i < n - m) {
                textHash = ((textHash - text.charAt(i) * Math.pow(base, m -
1) % prime) * base + text.charAt(i + m)) % prime;
                if (textHash < 0) {
                    textHash += prime;
                }
            }
        }

        return occurrences;
    }
    public static void main(String[] args) {
        String text = "ABCCDDAEFG";
        String pattern = "CDD";
        List<Integer> occurrences = search(text, pattern);
        System.out.println("Pattern found at positions: " + occurrences);
    }
}
```

The impact of hash collisions on the algorithm's performance is that it can lead to false positives, where the algorithm identifies a match even though the pattern is not actually present in the text. This is because the hash function may map different strings to the same hash value, leading to a collision.

To handle hash collisions, the Rabin-Karp algorithm verifies each potential match by comparing the characters in the text and pattern one by one. This step ensures that the algorithm only reports true matches, even in the presence of hash collisions. However, this verification step can slow down the algorithm's performance, especially if there are many hash collisions.

To improve the algorithm's performance and reduce the impact of hash collisions, you can choose a better hash function that minimizes the likelihood of collisions. Some strategies include:

- Using a larger prime number
- Increasing the base
- Using a more complex hash function
- Implementing a Bloom filter

Task 5: Boyer-Moore Algorithm Application

Use the Boyer-Moore algorithm to write a function that finds the last occurrence of a substring in a given string and returns its index. Explain why this algorithm can outperform others in certain scenarios.

```
public class BoyerMoore {
    public static int lastOccurrence(String text, String pattern) {
        int n = text.length();
        int m = pattern.length();



        int[] lps = new int[m];
        int j = 0;
        for (int i = 1; i < m; i++) {
            while (j > 0 && pattern.charAt(i) != pattern.charAt(j)) {
                j = lps[j - 1];
            }
            if (pattern.charAt(i) == pattern.charAt(j)) {
                j++;
            }
            lps[i] = j;
        }
        int i = n - 1;

        int j = m - 1;



        while (i >= 0) {

            if (pattern.charAt(j) == text.charAt(i)) {

                if (j == 0) {

                    return i;

                }

                i--;

                j--;

            } else {


                if (j > 0) {

                    j = lps[j - 1];

                } else {

                    i--;

                }

            }

        }


        return -1;


    public static void main(String[] args) {
        String text = "ABCCDDAEFG";
        String pattern = "CDD";
        int index = lastOccurrence(text, pattern);
        System.out.println("Last occurrence of pattern at index: " + index);
    }
}
```

The Boyer-Moore algorithm can outperform other algorithms in certain scenarios for several reasons:

- Efficient Use of Preprocessing: The algorithm preprocesses the pattern to create a lookup table (lps) that stores the longest proper prefix which is also a proper suffix for each prefix of the pattern.
- Reduced Comparisons: The algorithm uses the lookup table to skip characters in the text that cannot match the pattern.
- Improved Performance: The algorithm's performance is improved because it avoids unnecessary comparisons by skipping characters in the text that cannot match the pattern.
- Handling Long Patterns: The Boyer-Moore algorithm is particularly effective when dealing with long patterns.
- Handling Large Texts: The algorithm can handle large texts efficiently by using the lookup table to skip characters in the text that cannot match the pattern.

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.

```
public class BitManipulation {
  public static int countSetBits(int n) {
      int count = 0;
      while (n != 0) {
          count += n & 1;
          n >>= 1;
      }
      return count;
   }



   public static long countTotalSetBits(int n) {
      long totalSetBits = 0;
      for (int i = 1; i <= n; i++) {
          totalSetBits += countSetBits(i);
      }
      return totalSetBits;
   }

   public static void main(String[] args) {
      int n = 10;
```

```
        System.out.println("Number of set bits in " + n + ": " +
countSetBits(n));
        System.out.println("Total number of set bits in all integers from 1
to " + n + ": " + countTotalSetBits(n));
    }
}
```

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.

```
public class UniqueElements {
    public static int[] findUniqueElements(int[] arr) {
        int xor = 0;
        for (int i : arr) {
            xor ^= i;
        }
        int rightmostSetBit = xor & -xor;

        int x = 0;
        int y = 0;
        for (int i : arr) {
            if ((i & rightmostSetBit) != 0) {
                x ^= i;
            } else {
                y ^= i;
            }
        }

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

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 2, 1, 4, 3};
        int[] uniqueElements = findUniqueElements(arr);
        System.out.println("Unique elements: " + uniqueElements[0] + ", " + uniqueElements[1]);
    }
}
```

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.

```
public class TowerOfHanoi {
    public static void solveTowerOfHanoi(int n, char source, char auxiliary, char destination) {
        if (n == 1) {
            System.out.println("Move disk 1 from " + source + " to " +
destination);
        } else {
            solveTowerOfHanoi(n - 1, source, destination, auxiliary);
            System.out.println("Move disk " + n + " from " + source + " to "
+ destination);
            solveTowerOfHanoi(n - 1, auxiliary, source, destination);
        }
    }

    public static void main(String[] args) {
        int n = 3;
        solveTowerOfHanoi(n, 'A', 'B', '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.

```
public class TravelingSalesman {
    public static int findMinCost(int[][] graph) {
```

```
int n = graph.length;
int[] dp = new int[1 << n];
int[] parent = new int[1 << n];



for (int i = 0; i < (1 << n); i++) {
    dp[i] = Integer.MAX_VALUE;
}
dp[0] = 0;
for (int i = 1; i < (1 << n); i++) {
    for (int j = 0; j < n; j++) {

        if (((i >> j) & 1) == 1) {

            int subsetCost = dp[i ^ (1 << j)] + graph[j][0];

            if (subsetCost < dp[i]) {
                dp[i] = subsetCost;
                parent[i] = j;
            }
        }
    }
}
```

```
    int minCost = dp[(1 << n) - 1];
    int current = (1 << n) - 1;
    StringBuilder path = new StringBuilder();



    while (current != 0) {
        int city = parent[current];
        path.insert(0, city);
        current ^= (1 << city);
    }

    System.out.println("Path: " + path);
    System.out.println("Minimum cost: " + minCost);

    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}
        };

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

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.

```
import java.util.*;

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) {

        jobs.sort((a, b) -> a.deadline - b.deadline);



        List<Job> result = new ArrayList<>();



        int currentTime = 0;



        for (Job job : jobs) {

            if (currentTime + 1 <= job.deadline) {

                result.add(job);

                currentTime++;
            }
        }


        return result;
    }


    public static void main(String[] args) {
        List<Job> jobs = new ArrayList<>();
        jobs.add(new Job(1, 2, 50));
        jobs.add(new Job(2, 1, 20));
        jobs.add(new Job(3, 2, 100));
        jobs.add(new Job(4, 1, 30));
        jobs.add(new Job(5, 3, 200));


        List<Job> result = jobSequencing(jobs);


        System.out.println("Maximum profit sequence of jobs:");
        for (Job job : result) {
            System.out.println("Job ID: " + job.id + ", Deadline: " + job.deadline + ", Profit: " + job.profit);
        }
    }
}
```

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.

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



        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= W; j++) {
                dp[i][j] = 0;
            }
        }



        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= W; j++) {
                if (weights[i - 1] <= j) {
                    dp[i][j] = Math.max(values[i - 1] + dp[i - 1][j - weights[i - 1]],
dp[i - 1][j]);
```

```
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }

        return dp[n][W];
    }

    public static void main(String[] args) {
        int W = 10;
        int[] weights = {2, 3, 4, 5, 7};
        int[] values = {10, 20, 30, 40, 50};

        int maxValue = knapsack(W, weights, values);
        System.out.println("Maximum value: " + maxValue);
    }
}
```

Task 2: Longest Common Subsequence

Implement int LCS(string text1, string text2) to find the length of the longest common subsequence between two strings.

```
public class LongestCommonSubsequence {
    public 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 = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        return dp[m][n];
    }

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

        int lcsLength = lcs(text1, text2);
        System.out.println("Length of LCS: " + lcsLength);
    }
}
```

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.

```
public class KnightsTour {
    public static boolean solveKnightsTour(int[][] board, int moveX, int
moveY, int moveCount, int[] xMove, int[] yMove) {
        int n = board.length;
        boolean[][] visited = new boolean[n][n];


        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                visited[i][j] = false;
            }
        }


        visited[moveX][moveY] = true;


        if (!backtrackKnightsTour(board, moveX, moveY, moveCount,
xMove, yMove, visited)) {
            return false;
        }

        return true;
    }

    public static boolean backtrackKnightsTour(int[][] board, int moveX,
int moveY, int moveCount, int[] xMove, int[] yMove, boolean[][]
visited) {
        int n = board.length;
        int[] x = new int[n];
        int[] y = new int[n];

        for (int i = 0; i < n; i++) {
            x[i] = xMove[i];
            y[i] = yMove[i];
        }


        while (true) {

            int nextMove = -1;
            for (int i = 0; i < n; i++) {
                if (x[i] != -1 && y[i] != -1) {
                    if (nextMove == -1 || distance(moveX, moveY, x[i], y[i]) <
distance(moveX, moveY, x[nextMove], y[nextMove])) {
                        nextMove = i;
                    }
                }
            }


            if (nextMove == -1) {
                return false;
            }


            int newX = x[nextMove];
            int newY = y[nextMove];
```

```
visited[moveX][moveY] = true;

moveX = newX;
moveY = newY;

moveCount++;

if (moveCount == n * n) {
    return true;
}

if (!backtrackKnightsTour(board, moveX, moveY, moveCount, x, 
y, visited)) {
    return false;
}

return false;
```

```
        }
    }

    public static int distance(int x1, int y1, int x2, int y2) {
        return Math.abs(x1 - x2) + Math.abs(y1 - y2);
    }

    public static void main(String[] args) {
        int[][] board = new int[8][8];
        int moveX = 0;
        int moveY = 0;
        int moveCount = 1;
        int[] xMove = new int[]{2, 1, -1, -2, -2, -1, 1, 2};
        int[] yMove = new int[]{1, 2, 2, 1, -1, -2, -2, -1};

        if (solveKnightsTour(board, moveX, moveY, moveCount, xMove,
yMove)) {
            System.out.println("Solution exists");
        } else {
            System.out.println("No solution exists");
        }
    }
}
```

Task 2: Rat in a Maze

Implement 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.

```
public class RatInAMaze {
    public static boolean solveMaze(int[][] maze) {
        int n = maze.length;
        boolean[][] visited = new boolean[n][n];



        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                visited[i][j] = false;
            }
        }



        if (!backtrackMaze(maze, 0, 0, visited)) {
            return false;
        }

        return true;
```

```
  }

  public static boolean backtrackMaze(int[][] maze, int x, int y,
boolean[][] visited) {
      int n = maze.length;



      if (x < 0 || x >= n || y < 0 || y >= n || maze[x][y] == 0 ||
visited[x][y]) {
          return false;
      }



      if (x == n - 1 && y == n - 1) {
          return true;
      }



      visited[x][y] = true;



      if (backtrackMaze(maze, x + 1, y, visited) ||
          backtrackMaze(maze, x - 1, y, visited) ||
          backtrackMaze(maze, x, y + 1, visited) ||
        backtrackMaze(maze, x, y - 1, visited)) {
        return true;
    }



    visited[x][y] = false;


    return false;
}


public static void main(String[] args) {
    int[][] maze = {
        {1, 0, 0, 0, 0, 0},
        {1, 1, 1, 1, 1, 0},
        {0, 0, 0, 0, 1, 0},
        {0, 1, 1, 1, 1, 1},
        {0, 0, 0, 0, 1, 0},
        {0, 0, 0, 0, 1, 1}
    };


    if (solveMaze(maze)) {
        System.out.println("Path found");
    } else {
        System.out.println("No path found");
        }
    }
}
```

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.

```
public class NQueen {
    public static boolean solveNQueen(int[][] board, int col) {
        int n = board.length;


        if (col >= n) {
            return true;
        }



        for (int i = 0; i < n; i++) {

            if (isSafe(board, i, col)) {
```

```
            board[i][col] = 1;



            if (solveNQueen(board, col + 1)) {
                return true;
            }



            board[i][col] = 0;
        }
    }

    return false;
}

public static boolean isSafe(int[][] board, int row, int col) {
    int n = board.length;



    for (int i = 0; i < n; i++) {
        if (board[i][col] == 1) {
            return false;
        }
    }
```

```
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 1) {
                return false;
            }
        }

        for (int i = row + 1, j = col - 1; i < board.length && j >= 0; i++, j--) {
            if (board[i][j] == 1) {
                return false;
            }
        }

        for (int i = row - 1, j = col + 1; i >= 0 && j < board.length; i--, j++) {
            if (board[i][j] == 1) {
                return false;
            }
        }

        for (int i = row + 1, j = col + 1; i < board.length && j < board.length;
i++, j++) {
            if (board[i][j] == 1) {
                return false;
            }
        }

        return true;
    }

    public static void main(String[] args) {
        int[][] board = new int[8][8];



        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                board[i][j] = 0;
            }
        }

       if (solveNQueen(board, 0)) {
            System.out.println("Solution found");
        } else {
            System.out.println("No solution found");
        }
    }
}
```