Ain-Shams University

Faculty of Engineering

Department: Computer and Systems Engineering Senior - 1

Course Name: Design and analysis of Algorithms – spring 2023

Course Code: CSE332s

Member Name	Member Faculty ID
Mostafa Mahmoud Ali Ahmed	1900034
Abdelrahman Ali Mohamed Ali	1900913
Aliaa Nabil Mahmoud Mohamed	1900949
Ahmed Mohamed Mahmoud Mohamed	1900185
Farah Mohamed Yasser Mohamed	1900192
Malek Abdelrhman Hassan	1901104
Muhamed hassan mohamed	1900819
Salma Ahmed Desouky Mohamed Ibrahim	1901517
Reem Abdelbary Hussien Abdelbary	1901047
Jumana Emad Eldin Saleh Mohamed Elhak	1900980
Ahmed Essam El-Din Mohammed	1900731
Omar Mohamed Moustafa Mohamed	1900953

Table of Contents

Table of Figures

Task 1

Problem:

Devise an algorithm for the following task given a $2^n \times 2^n$ ($n > 1$) board with one missing square, tile it with right trominoes of only three colors so that no pair of trominoes that share an edge have the same color. Recall that the right tromino is an L-shaped tile formed by three adjacent squares.
Use dynamic programing to solve this problem.

Problem Description:

Given n by n board where n is of form 2^k where k >1 (Basically n is a power of 2 with minimum value as 2). The board has one missing cell. Fill the board using L-shaped tiles (trominoes) which consist of 3 squares. But we must make sure that no two trominoes adjacent to each other have the same size knowing that we have only 3 colors for the trominoes.

Solution Description:

The problem can be solved by the following recursive algorithm. If n = 2,
divide the board into four 2 × 2 boards and place one gray tromino to cover the three central squares that are not in the 2 × 2 board with the missing square. Then place one black tromino in the upper left 2 × 2 board, one white tromino in the upper right 2 × 2 board, one black tromino in the lower right 2 × 2 board, and one white tromino in the lower left 2 × 2 board. [1]
We divide the board into sub-quads and use these smaller quads into building the original large board.
Note that for any location of the missing square, the following properties hold:
1- Going left to right, the upper edge of the board is covered by an alternating sequence of two black squares followed by two white squares, with the missing square possibly replacing one square in this sequence.

2- Going down, the right edge of the board is covered by an alternating sequence of two white squares followed by two black squares, with the missing square possibly replacing one square in this sequence.

3- Going right to left, the lower edge of the board is covered by an alternating sequence of two black squares followed by two white squares, with the missing square possibly replacing one square in this sequence.

4- Going up, the left edge of the board is covered by an alternating sequence of two white squares followed by two black squares, with the missing square possibly replacing one square in this sequence.
If n > 2, divide the board into four $2n-1 \times 2n-1$ boards and place one gray tromino to cover the three central squares that are not in the $2n-1 \times 2n-1$ board with the missing square. Then tile each of the three $2n-1 \times 2n-1$
boards recursively by the same algorithm.

Problem Pseudocode:

```
function int[][] tileBoard(int[][] board,int rowxg , int colxg)
    int n = board.length
    int[][] tiling = board

    // base when input n =1
    if (n == 2) {
       if(rowxg ==0 AND colxg==0)
           for (int i <-- 0; i < n; i++)
                for (int j <-- 0; j < n; j++)
                    if (i ==0 AND j==0)
                       continue

                    tiling[i][j]=1

           if(rowxg ==1 AND colxg==0)
               for (int i <-- 0; i < n; i++)
                    for (int j<-- 0; j < n; j++)
                        if (i==1 AND j==0)
                           continue

                       tiling[i][j]=0


       if(rowxg ==0 AND colxg==1)
             for (int i <-- 0; i < n; i++)
                  for (int j <-- 0; j < n; j++)
                     if (i==0 AND j==1)
                        continue

                     tiling[i][j]=0

        if(rowxg ==1 AND colxg==1)
             for (int i <-- 0; i < n; i++)
                for (int j <-- 0; j < n; j++)
                     if (i==1 AND j==1)
                              continue

                           tiling[i][j]=1

    else
                    int[][] quad1 = new int[n/2][n/2]
            int[][] quad2 = new int[n/2][n/2]
            int[][] quad3 = new int[n/2][n/2]
            int[][] quad4 = new int[n/2][n/2]

            f1 = false
            f2 = false
            f3 = false
            f4 = false

            int[][] tiling1= new int[n/2][n/2]
            int[][] tiling2= new int[n/2][n/2]
            int[][] tiling3= new int[n/2][n/2]
            int[][] tiling4= new int[n/2][n/2]

            for (int i <-- 0; i < n; i++)
                for (int j <-- 0; j < n; j++)
                    if (i < n/2 AND j < n/2)
                        quad1[i][j] = board[i][j]
                     else if (i < n/2 AND j >= n/2)
                        quad2[i][j-n/2] = board[i][j]
                     else if (i >= n/2 AND j < n/2)
                        quad3[i-n/2][j] = board[i][j]
                    else
                        quad4[i-n/2][j-n/2] = board[i][j]

                 t = quad2.length

                for(int i <-- 0;i<t ; i++)
                    for(int j<-- 0;j<t;j++)
                        if(quad1[i][j]==9 ||quad1[i][j]==2)
                            f1 <--true
                            break
                for(int i <--0;i<t ; i++)
                    for(int j<--0;j<t;j++)
                        if(quad2[i][j]==9 ||quad2[i][j]==2)
                            f2 <--true
                            break
                for(int i <--0;i<t ; i++)
                    for(int j=0;j<t;j++)
                        if(quad3[i][j]==9 ||quad3[i][j]==2)
                            f3 <-- true
                            break
                for(int i <--0;i<t ; i++)
                    for(int j=0;j<t;j++)
                        if(quad4[i][j]==9 ||quad4[i][j]==2)
                            f4=true
                            break
```

```
                if(f1)
                            quad2[t-1][0]=2
                            quad3[0][t-1]=2
                            quad4[0][0]=2
                            if(rowxg > t/2)
                                rowxg -= t
                            if(colxg > t/2)
                                colxg -= t
                            tiling1 = tileBoard(quad1,rowxg,colxg)
                            tiling2 = tileBoard(quad2,t-1,0)
                            tiling3 = tileBoard(quad3,0,t-1)
                            tiling4 = tileBoard(quad4,0,0)
                        if(f2)
                              quad1[t-1][t-1]=2
                              quad3[0][t-1]=2
                              quad4[0][0]=2
                              tiling1 = tileBoard(quad1,t-1,t-1)
                              if(rowxg > t/2)
                                  rowxg -= t

                              if(colxg > t/2)
                                  colxg -= t

                              tiling2 = tileBoard(quad2,rowxg,colxg)
                              tiling3 = tileBoard(quad3,0,t-1)
                              tiling4 = tileBoard(quad4,0,0)

                         if(f3)
                                quad1[t-1][t-1]=2
                                quad2[t-1][0]=2
                                quad4[0][0]=2
                                tiling1 = tileBoard(quad1,t-1,t-1)
                                tiling2 = tileBoard(quad2,t-1,0)
                                if(rowxg > t/2)
                                    rowxg -= t

                                if(colxg > t/2)
                                    colxg -= t

                                tiling3 = tileBoard(quad3,rowxg,colxg)
                                tiling4 = tileBoard(quad4,0,0)

            if(f4)
                                quad1[t-1][t-1]=2
                                quad2[t-1][0]=2
                                quad3[0][t-1]=2
                                tiling1 = tileBoard(quad1,t-1,t-1)
                                tiling2 = tileBoard(quad2,t-1,0)
                                tiling3 = tileBoard(quad3,0,t-1)
                                if(rowxg > t/2)
                                    rowxg -= t
                                if(colxg > t/2)
                                    colxg -= t
                                tiling4 = tileBoard(quad4,rowxg,colxg)

            // Merge the solutions of the quadrants
            for (int i <-- 0; i < n; i++)
                for (int j <-- 0; j < n; j++)
                    if (i < n/2 AND j < n/2)
                        tiling[i][j] = tiling1[i][j]
                     else if (i < n/2 AND j >= n/2)
                        tiling[i][j] = tiling2[i][j-n/2]
                     else if (i >= n/2 AND j < n/2)
                        tiling[i][j] = tiling3[i-n/2][j]
                     else
                        tiling[i][j] = tiling4[i-n/2][j-n/2]
        return tiling
```

Problem Java Code

```
package Project;

public class NewClass1 {
    public static final int WHITE = 0;
    public static final int BLACK = 1;
    public static final int GRAY = 2;



    public static int[][] tileBoard(int[][] board,int rowxg , int colxg) {
        int n = board.length;
        int[][] tiling = board;

        // base when input n =1
        if (n == 2) {
            if(rowxg ==0 && colxg==0) {
                for (int i = 0; i < n; i++) {
                    for (int j = 0; j < n; j++) {
                        if (i==0 && j==0) {
                            continue;
                        }
                        tiling[i][j]=1;
                    }
                }
            }

            if(rowxg ==1 && colxg==0) {
                for (int i = 0; i < n; i++) {
                    for (int j = 0; j < n; j++) {
                        if (i==1 && j==0) {
                            continue;
                        }
                        tiling[i][j]=0;
                    }
                }
            }
            if(rowxg ==0 && colxg==1) {
                for (int i = 0; i < n; i++) {
                    for (int j = 0; j < n; j++) {
                        if (i==0 && j==1) {
                            continue;
                        }
                        tiling[i][j]=0;
                    }
                }
            }
            if(rowxg ==1 && colxg==1) {
                for (int i = 0; i < n; i++) {
                    for (int j = 0; j < n; j++) {
                        if (i==1 && j==1) {
                            continue;
                        }
                        tiling[i][j]=1;
                    }
                }
            }

        }
```

```
        else {
            int[][] quad1 = new int[n/2][n/2];
            int[][] quad2 = new int[n/2][n/2];
            int[][] quad3 = new int[n/2][n/2];
            int[][] quad4 = new int[n/2][n/2];

            boolean f1 = false;
            boolean f2 = false;
            boolean f3 = false;
            boolean f4 = false;

            int[][] tiling1= new int[n/2][n/2];
            int[][] tiling2= new int[n/2][n/2];
            int[][] tiling3= new int[n/2][n/2];
            int[][] tiling4= new int[n/2][n/2];

            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (i < n/2 && j < n/2) {
                        quad1[i][j] = board[i][j];
                    } else if (i < n/2 && j >= n/2) {
                        quad2[i][j-n/2] = board[i][j];
                    } else if (i >= n/2 && j < n/2) {
                        quad3[i-n/2][j] = board[i][j];
                    } else {
                        quad4[i-n/2][j-n/2] = board[i][j];
                    }
                }
            }

            int t = quad2.length;

            for(int i =0;i<t ; i++) {
                for(int j=0;j<t;j++) {
                    if(quad1[i][j]==9 ||quad1[i][j]==2) {
                        f1=true;
                        break;
                    }
                }
            }
            for(int i =0;i<t ; i++) {
                for(int j=0;j<t;j++) {
                    if(quad2[i][j]==9 ||quad2[i][j]==2) {
                        f2=true;
                        break;
                    }
                }
            }
            for(int i =0;i<t ; i++) {
                for(int j=0;j<t;j++) {
                    if(quad3[i][j]==9 ||quad3[i][j]==2) {
                        f3=true;
                        break;
                    }
                }
            }
            for(int i =0;i<t ; i++) {
                for(int j=0;j<t;j++) {
                    if(quad4[i][j]==9 ||quad4[i][j]==2) {
                        f4=true;
                        break;
                    }
```

```
        }
    }
    if(f1) {
        quad2[t-1][0]=2;
        quad3[0][t-1]=2;
        quad4[0][0]=2;
        if(rowxg > t/2){
            rowxg -= t;
        }
        if(colxg > t/2){
            colxg -= t;
        }

        tiling1 = tileBoard(quad1,rowxg,colxg);
        tiling2 = tileBoard(quad2,t-1,0);
        tiling3 = tileBoard(quad3,0,t-1);
        tiling4 = tileBoard(quad4,0,0);
    }

    if(f2) {
        quad1[t-1][t-1]=2;
        quad3[0][t-1]=2;
        quad4[0][0]=2;
        tiling1 = tileBoard(quad1,t-1,t-1);
        if(rowxg > t/2){
            rowxg -= t;
        }
        if(colxg > t/2){
            colxg -= t;
        }
        tiling2 = tileBoard(quad2,rowxg,colxg);
        tiling3 = tileBoard(quad3,0,t-1);
        tiling4 = tileBoard(quad4,0,0);
    }

    if(f3) {
        quad1[t-1][t-1]=2;
        quad2[t-1][0]=2;
        quad4[0][0]=2;
        tiling1 = tileBoard(quad1,t-1,t-1);
        tiling2 = tileBoard(quad2,t-1,0);
        if(rowxg > t/2){
            rowxg -= t;
        }
        if(colxg > t/2){
            colxg -= t;
        }
        tiling3 = tileBoard(quad3,rowxg,colxg);
        tiling4 = tileBoard(quad4,0,0);
    }


    if(f4) {
        quad1[t-1][t-1]=2;
        quad2[t-1][0]=2;
        quad3[0][t-1]=2;
        tiling1 = tileBoard(quad1,t-1,t-1);
        tiling2 = tileBoard(quad2,t-1,0);
        tiling3 = tileBoard(quad3,0,t-1);
        if(rowxg > t/2){
            rowxg -= t;
        }
```

```
                if(colxg > t/2){
                    colxg -= t;
                }
                tiling4 = tileBoard(quad4,rowxg,colxg);
            }

            // Merge the solutions of the quadrants
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (i < n/2 && j < n/2) {
                        tiling[i][j] = tiling1[i][j];
                    } else if (i < n/2 && j >= n/2) {
                        tiling[i][j] = tiling2[i][j-n/2];
                    } else if (i >= n/2 && j < n/2) {
                        tiling[i][j] = tiling3[i-n/2][j];
                    } else {
                        tiling[i][j] = tiling4[i-n/2][j-n/2];
                    }
                }
            }
        }
        return tiling;
    }

}
```

```
package Project;

import static Project.NewClass1.tileBoard;

public class maiiiin {

    public static void main(String[] args) {

        int n =4;

        int[][] board = new int[(int) Math.pow(2, n)][(int) Math.pow(2, n)];

        board[1][0] = 9;

        int[][] tiling = tileBoard(board,1,0);


        for (int[] row : board) {


            for (int cell: row) {


                if (cell == 9) {


                    System.out.print("X ");


                } else if (cell == 1) {


                    System.out.print("B ");
```

```
                } else if (cell == 0) {
                    System.out.print("W ");
                } else if (cell == 2) {
                    System.out.print("G ");
                }
            }
            System.out.println();
        }
    }
}
```

Problem Complexity Analysis:

The time complexity of this code is O(N^2)where the size of the board is 2^N.

Problem Output screenshots:

At n=4 and the missing square at (1,0):

```
run:
W W W W B B W W B B W W B B W W
X W G W B G G W B G G W B G G W
W G G B W W G B W G B B W W G B
W W B B G W B B W W B G G W B B
B B W G G B W W B B W W G B W W
B G W W B B G W B G G W B B G W
W G G B W G G B W W G B W G G B
W W B B W W B B G W B B W W B B
B B W W B B W G G B W W B B W W
B G G W B G W W B B G W B G G W
W G B B W G G B W G G B W W G B
W W B G W W B B W W B B G W B B
B B W G G B W W B B W G G B W W
B G W W B B G W B G W W B B G W
W G G B W G G B W G G B W G G B
W W B B W W B B W W B B W W B B
BUILD SUCCESSFUL (total time: 0 seconds)
```

At n=4 and the missing square at (0,0):

```
run:
X B W W B B W W
B B G W B G G W
W G G B W W G B
W W B B G W B B
B B W G G B W W
B G W W B B G W
W G G B W G G B
W W B B W W B B
BUILD SUCCESSFUL
```

Comparison Between Algorithms

This problem can be solved using Divide and Conquer. Below is the recursive algorithm:
1) Base case: n = 2, A 2 x 2 square with one cell missing is nothing but a tile and can be filled with a single tile.

2) Place a L-shaped tile at the center such that it does not cover the n/2 * n/2 subsquare that has a missing square. Now all four subsquares of size n/2 x n/2 have a missing cell (a cell that doesn't need to be filled).

3) Solve the problem recursively for following four. Let p1, p2, p3 and
p4 be positions of the 4 missing cells in 4 squares.
a) Tile(n/2, p1)
b) Tile(n/2, p2)
c) Tile(n/2, p3)
d) Tile(n/2, p3)

Comparison between algorithms:

Dynamic programming Algorithm:[2]
The dynamic programming method is generally the preferred approach for solving this problem as it provides an optimal solution efficiently by breaking down the problem into smaller subproblems and avoiding duplicate computations.

- Time Complexity: O(n^2)

Divide and Conquer Algorithm:[3]
The function `tile()` is called recursively on quadrants of size n/2 until n equals 2, at which point a constant time operation is performed. Therefore, the depth of the recursion tree is log(n), and each level performs O(n^2) operations. Hence, the total time complexity is O(n^2 * log(n)).

- Time Complexity: O(n^2 * log(n)).

Pseudocode:

```
    static int size_of_grid, b, a, cnt = 0
    static int[][] arr = new int[128][128]
    static int[] colors = {1, 2, 3} // Three colors to use

    // Placing tile at the given coordinates
   function place(int x1, int y1, int x2, int y2, int x3, int y3) {
        cnt++
        arr[x1][y1] = cnt
        arr[x2][y2] = cnt
        arr[x3][y3] = cnt
    }

    // Quadrant names
    // 1 2
    // 3 4

    // Function based on divide and conquer
    function tile(int n, int x, int y, int color) {
        int r = 0, c = 0;
        if (n == 2) {

                cnt++


            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (arr[x + i][y + j] == 0) {
                        arr[x + i][y + j] = cnt
                    }
                }
            }
            return 0
        }

        // finding hole location
        for (int i = x; i < x + n; i++) {
            for (int j = y; j < y + n; j++) {
                if (arr[i][j] != 0) {
                    r = i
                    c = j
                }
            }
        }

        // Choose a color for the tromino that doesn't match any of the adjacent trominoes
        int[] adjacentColors = {0, 0, 0}; // Colors of the adjacent trominoes
        if (r >= 1 && arr[r - 1][c] != 0) {
            adjacentColors[0] = arr[r - 1][c]
        }
        if (c < arr.length - 1 && arr[r][c + 1] != 0) {
            adjacentColors[1] = arr[r][c + 1]
        }
        if (r < arr.length - 1 && arr[r + 1][c] != 0) {
            adjacentColors[2] = arr[r + 1][c]
        }
        int trominoColor = 0;
        for (int i = 0; i < colors.length; i++) {
            if (colors[i] != adjacentColors[0] && colors[i] != adjacentColors[1] && colors[i] !=
adjacentColors[2]) {
                trominoColor = colors[i];
                break;
            }
        }
```

```
        // If missing tile is 1st quadrant
        if (r < x + n / 2 && c < y + n / 2) {
            place(x + n / 2, y + (n / 2) - 1, x + n / 2, y + n / 2, x + n / 2 - 1, y + n / 2)
            tile(n / 2, x, y + n / 2, trominoColor)
            tile(n / 2, x, y, trominoColor)
            tile(n / 2, x + n / 2, y, trominoColor)
            tile(n / 2, x + n / 2, y + n / 2, trominoColor)
        }

        // If missing Tile is in 3rd quadrant
        else if (r >= x + n / 2 && c < y + n / 2) {
            place(x + (n / 2) - 1, y + (n / 2), x + (n / 2), y + n / 2, x + (n / 2) - 1, y + (n / 2)
- 1)
            tile(n / 2, x, y + n / 2, trominoColor)
            tile(n / 2, x, y, trominoColor)
            tile(n / 2, x + n / 2, y, trominoColor)
            tile(n / 2, x + n / 2, y + n / 2, trominoColor)
        }

        // If missing Tile is in 2nd quadrant
        else if (r < x + n / 2 && c >= y + n / 2) {
            place(x + n / 2, y + (n / 2) - 1, x + n / 2, y + n / 2, x + n / 2 - 1, y + n / 2 - 1)
            tile(n / 2, x, y + n / 2, trominoColor)
            tile(n / 2, x, y, trominoColor)
            tile(n / 2, x + n / 2, y, trominoColor)
            tile(n / 2, x + n / 2, y + n / 2, trominoColor)
        }

        // If missing Tile is in 4th quadrant
        else if (r >= x + n / 2 && c >= y + n / 2) {
            place(x + (n / 2) - 1, y + (n / 2), x + (n / 2), y + (n / 2) - 1, x + (n / 2) - 1, y + (n
/ 2) - 1)
            tile(n / 2, x, y + n / 2, trominoColor)
            tile(n / 2, x, y, trominoColor)
            tile(n / 2, x + n / 2, y, trominoColor)
            tile(n / 2, x + n / 2, y + n / 2, trominoColor)
        }
        return 0
    }


    function  main() {

        // size of box
        size_of_grid = 8

        // Coordinates which will be marked
        a = 0
        b = 0

        // Here tile can not be placed
        arr[a][b] = -1
        tile(size_of_grid, 0, 0, 0)

        // The grid is
        for (int i <-- 0; i < size_of_grid; i++) do
            for (int j <-- 0; j < size_of_grid; j++) do
                System.out.print(arr[i][j] + " ")
            end for
     end for
            System.out.println()
        }
    }
```

Conclusion:

Dynamic programming and divide and conquer are both powerful techniques for solving complex problems, but they differ in their approach and efficiency. Dynamic programming can be more efficient than divide and conquer in some cases because it avoids redundant computations by storing the solutions to subproblems. However, the efficiency of dynamic programming depends on the problem being solved and the specific implementation.

Task 2

Problem:

Is it possible for a chess knight to visit all the cells of an 8 × 8 chessboard exactly once, ending at a cell one knight's move away from the starting cell? (Such a tour is called closed or re-entrant. Note that a cell is considered visited only when the knight lands on it, not just passes over it on its move.)
If it is possible design a greedy algorithm to find the minimum number of moves the chess knight needs.

Problem Description:

The problem asks if a knight can move around the chessboard and visit every square exactly once, ending at one knight's move away from the starting point. The knight is a chess piece that moves in an L-shape, two squares horizontally and one square vertically, or two squares vertically and one horizontally. The knight can move to any square on the board as long as it is not already occupied.

Solution Description:

Here are the Steps to solve this problem:

1-The solution uses the Warnsdorff heuristic to pick the next move for the knight. This heuristic chooses the move that leads to the square with the fewest accessible squares (i.e., the lowest degree) to avoid getting stuck in a dead end.
2-The program starts by initializing an 8x8 chessboard with -1's and selecting a starting position for the knight.
3-Then, it repeatedly applies the Warnsdorff heuristic to find the next move until the knight has visited all squares.
4-Finally, the program checks whether the tour is closed (i.e., the knight can end up at the starting position) and prints the resulting chessboard.
5-The BoardCell class is used to store the x and y coordinates of the knight's current position.
6-The limits method checks whether the knight is still within the 8x8 chessboard
7-The isempty method checks whether a square is empty.
8-The getDegree method returns the number of empty squares adjacent to the current position.
9-The nextMove method applies the Warnsdorff heuristic to select the next move for the knight.
10-The print method prints the resulting chessboard.
11-The neighbour method checks whether the knight ends up at a square that is one knight's move from the starting position.
12-Finally, the findClosedTour method generates the legal moves using Warnsdorff's heuristic and checks whether the tour is closed
13-The program runs in a loop until a closed tour is found.[1,4]

Problem Pseudocode :

```
// Define the size of the chessboard
const N = 8;

// Define the move pattern for the knight
const xcoordinate = [1, 1, 2, 2, -1, -1, -2, -2];
const ycoordinate = [2, -2, 1, -1, 2, -2, 1, -1];

// Check if the knight can move to the given (x, y) coordinates
function limits(x, y):
    return x >= 0 && y >= 0 && x < N && y < N

// Check if the given square is empty
function isempty(a, x, y):
    return limits(x, y) && a[y * N + x] < 0

// Count the number of empty squares adjacent to the given (x, y) coordinates
function getDegree(a, x, y):
    count = 0
    for i in [0, N):
        if isempty(a, x + xcoordinate[i], y + ycoordinate[i]):
            count += 1
    return count

// Use Warnsdorff's heuristic to pick the next move
function nextMove(a, cell):
    min_degree_idx = -1
    min_degree = N + 1
    start = random number between 0 and N-1
    for count in [0, N):
        i = (start + count) % N
        nextx = cell.x + xcoordinate[i]
        nexty = cell.y + ycoordinate[i]
        if isempty(a, nextx, nexty) && getDegree(a, nextx, nexty) <
min_degree: min_degree_idx = i
            min_degree = getDegree(a, nextx, nexty)
    if min_degree_idx == -1:
        return null
    nextx = cell.x + xcoordinate[min_degree_idx]
    nexty = cell.y + ycoordinate[min_degree_idx]
    a[nexty * N + nextx] = a[cell.y * N + cell.x] + 1
    cell.x = nextx
    cell.y = nexty
    return cell

// Check if the given (x, y) coordinates are adjacent to the starting point
function neighbour(x, y, xx, yy):
    for i in [0, N):
        if x + xcoordinate[i] == xx && y + ycoordinate[i] == yy:
            return true
    return false

// Find a closed Knight's Tour using Warnsdorff's heuristic
function findClosedTour():
    // Initialize the chessboard with -1s
    a = array of size N*N filled with -1
    // Set the initial position
    startx = 1
    starty = 6
    cell = BoardCell(startx, starty)
    a[cell.y * N + cell.x] = 1 // Mark the first move
    // Use Warnsdorff's heuristic to find the next moves
    for i in [0, N*N-1):
        ret = nextMove(a, cell)
        if ret == null:
            return false
    // Check if the tour is closed
    if !neighbour(ret.x, ret.y, startx, starty):
        return false
    print the chessboard a
    return true

// Run the program until a closed Knight's Tour is found
while !findClosedTour():
    continue
```

Problem Java Code:

```
package Algoo;

import java.util.concurrent.ThreadLocalRandom;

class KnightTour
{
    public static final int N = 8;

    public static final int xcoordinate[] = {1, 1, 2, 2, -1, -1, -2, -2};

    public static final int ycoordinate[] = {2, -2, 1, -1, 2, -2, 1, -1};



    // the 8x8 chessboard
    boolean limits(int x, int y)
    {
        return ((x >= 0 && y >= 0) &&
                (x < N && y < N));
    }



    /* Checks whether a square is valid and empty or not */
    boolean isempty(int a[], int x, int y)
    {
        return (limits(x, y)) && (a[y * N + x] < 0);
    }




    /* Returns the number of empty squares adjacent to (x, y) */
    int getDegree(int a[], int x, int y)
    {
        int count = 0;
        for (int i = 0; i < N; ++i)
            if (isempty(a, (x + xcoordinate[i]),
                    (y + ycoordinate[i])))
                count++;

        return count;
    }
```


```
BoardCell nextMove(int a[], BoardCell cell)
  {
      int min_degree_idx = -1, c,
              min_degree = (N + 1), nextx, nexty;
      int start = ThreadLocalRandom.current().nextInt(1000) % N;
      for (int count = 0; count < N; ++count)
      {
          int i = (start + count) % N;
          nextx = cell.x + xcoordinate[i];
          nexty = cell.y + ycoordinate[i];
          if ((isempty(a, nextx, nexty)) &&
                  (c = getDegree(a, nextx, nexty)) < min_degree)
          {
              min_degree_idx = i;
              min_degree = c;
          }
      }

      if (min_degree_idx == -1)
          return null;
      // Store coordinates of next point
      nextx = cell.x + xcoordinate[min_degree_idx];
      nexty = cell.y + ycoordinate[min_degree_idx];

      a[nexty * N + nextx] = a[(cell.y) * N +
              (cell.x)] + 1;

      // Update next point
      cell.x = nextx;
      cell.y = nexty;

      return cell;
  }

  /* displays the chessboard with knight's moves */
  void print(int a[])
  {
      for (int i = 0; i < N; ++i)
      {
          for (int j = 0; j < N; ++j)
              System.out.printf("%d\t", a[j * N + i]);
          System.out.printf("\n");
      }
  }

  boolean neighbour(int x, int y, int xx, int yy)
  {
      for (int i = 0; i < N; ++i)
          if (((x + xcoordinate[i]) == xx) &&
                  ((y + ycoordinate[i]) == yy))
              return true;
```

```
        return false;}

    boolean findClosedTour()
    {

        int a[] = new int[N * N];
        for (int i = 0; i < N * N; ++i)
            a[i] = -1;
        int startx = 0;
        int starty = 0;

        // Current points are same as initial points
        BoardCell cell = new BoardCell(startx, starty);

        a[cell.y * N + cell.x] = 1; // Mark first move.

        BoardCell ret = null;
        for (int i = 0; i < N * N - 1; ++i)
        {
            ret = nextMove(a, cell);
            if (ret == null)
                return false;
        }

        if (!neighbour(ret.x, ret.y, startx, starty))
            return false;

        print(a);
        return true;
    }

    public static void main(String[] args)
    {
        // While we don't get a solution
        while (!new KnightTour().findClosedTour())
        {
            ;
        }
    }
}

class BoardCell
{
    int x;
    int y;

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

Problem Complexity Analysis:

The time complexity of this code is O(N^2 * log(N)) where N is the size of the board.

Problem Output screenshots:

At startx=0 and starty=0:

```
1   44  15  24  39  28  13  26
16  23  64  43  14  25  40  29
63  2   45  38  61  42  27  12
22  17  62  59  46  49  30  41
3   58  21  50  37  60  11  48
18  51  36  57  54  47  8   31
35  4   53  20  33  6   55  10
52  19  34  5   56  9   32  7
```

Figure 8 Task 2 Output 1

-At startx=1 and starty=6 :

```
4   27  6   21  2   49  16  19
7   22  3   48  17  20  1   50
28  5   26  23  64  51  18  15
25  8   45  40  47  38  57  52
44  29  24  61  56  63  14  37
9   32  41  46  39  60  53  58
30  43  34  11  62  55  36  13
33  10  31  42  35  12  59  54
```

Figure 9 Task 2 Output 2

-At startx=5 and starty=3 :

```
57  34  53  8   55  24  51  6
46  9   56  35  52  7   26  23
33  58  47  54  25  60  5   50
10  45  36  59  64  49  22  27
37  32  63  48  43  28  61  4
14  11  44  1   62  19  42  21
31  38  13  16  29  40  3   18
12  15  30  39  2   17  20  41
```

Figure 10 Task 2 Output 3

Comparison Between Algorithms:

The used algorithm is Warnsdorff's algorithm is a heuristic algorithm that uses a set of rules to choose the next move. It evaluates all the possible moves from the current position and selects the move that has the fewest number of accessible squares. By choosing the moves that reduce the number of available squares first.

Another algorithm which is backtracking algorithm is a brute-force search algorithm that systematically explores all possible paths until it finds a solution. It starts with an empty board and tries to place the knight at each possible position. If a position leads to a dead-end, it backtracks to the previous position and tries the next available position. The process continues until all squares are visited or there are no more available moves.[4]

Steps of this algorithm :

1. Initialize a fixed size chess board with all elements set to 0.
2. Set the element of the board at starting x and y coordinates to 1.
3. Call the solve method with the starting x and y coordinates, count set to 2, and starting x and y coordinates.
4. In the solve method, check if the count is greater than the total number of squares on the board.
5. If it is, check if the last move made results in a closed tour using the isClosedTour method.
6. If it is a closed tour, return true. Else, return false.
7. Get the list of possible next moves for the knight's position using getNextMoves method. And Sort the list of possible next moves based on the number of possible moves from each next move position using a lambda expression.
8. For each possible next move, check if the position is not already visited. If it is not visited, mark the position on the board with the count.
9. Call the solve method recursively with the new position and increment the count by 1, If the recursive call returns true, it means a closed tour has been found, return true. Else, mark the current position as not visited on the board, And If no closed tour has been found, return false.
10. Implement the getNextMoves method to get a list of all possible moves from the current position for the knight and Implement the isClosedTour method to check if the current position is a closed tour or not. And Implement the printBoard method to print the final knight's tour on the board.[4]

- The time complexity of this algorithm is O(8^(n^2)) where n is the size of the board . The reason for this is that there are a maximum of 8 possible moves from each square on the board and there are n^2 squares on the board. The algorithm tries all possible moves from each square until it either finds a closed tour or exhausts all possibilities.

Comparison between two algorithms:

- Warnsdorff's algorithm can find a solution faster than the backtracking algorithm.
- The backtracking algorithm is a reliable method that can always find a solution, but it can be slow for large board sizes. Warnsdorff's algorithm is a fast algorithm that works well for most board sizes, but it may fail to find a solution for certain starting positions.
- Time complexity: Warnsdorff's algorithm is generally faster than the backtracking algorithm since it can quickly eliminate moves that lead to a dead-end. However, it is not guaranteed to find a solution for all starting positions, and it may fail to find a solution for certain board sizes.
- Space complexity: Both algorithms have the same space complexity of O(N^2), where N is the size of the board.
- Completeness: Both algorithms are complete, meaning that they will find a solution if one exists.
- Optimality: Neither algorithm guarantees an optimal solution, meaning a solution that visits all squares on the board in the fewest number of moves. However, backtracking can be modified to find an optimal solution by exploring all possible paths and choosing the one with the shortest number of moves.
- Implementation: The implementation of Warnsdorff's algorithm is usually simpler than that of backtracking. This is because Warnsdorff's algorithm only requires a list of moves and a way to keep track of the squares that have already been visited, whereas backtracking requires a recursive function to explore all possible paths.

Overall, Warnsdorff's algorithm is good for solving the knight's tour problem. It's generally faster, simpler to implement, and has the same completeness and space complexity as backtracking. However, it doesn't guarantee an optimal solution, so backtracking might still be preferred in certain situations.[4]

Conclusion:

- In conclusion , The question of whether a chess knight can go exactly once through each cell on an 8x8 chessboard and end up at a cell one knight's move from the beginning cell has been investigated. On the basis of Warnsdorff's heuristic, we have also provided a solution using a greedy method.
- In order to avoid dead ends, the Warnsdorff heuristic directs us to choose the knight's next move to the square with the fewest available squares and we can locate a tour that visits every square on the chessboard.
- The answer is displaying the chessboard as a 2D grid and utilising the BoardCell class to maintain track of the knight's present location. We utilise the isempty function to determine whether a square is empty and check the bounds to make sure the knight stays inside the confines of the chessboard.
- We produce permissible movements using Warnsdorff's heuristic by repeatedly using the findClosedTour function until a closed tour is discovered. Upon completion of the closed tour, The resultant chessboard is printed.

Task 3

Problem

There is a row of n security switches protecting a military installation entrance. The switches can be manipulated as follows:
(i) The rightmost switch may be turned on or off at will.
(ii) Any other switch may be turned on or off only if the switch to its immediate right is on and all the other switches to its right, if any, are off.
(iii) Only one switch may be toggled at a time.
Design a Dynamic Programming algorithm to turn off all the switches, which are initially all on, in the minimum number of moves. (Toggling one switch is considered one move.) Also, find the minimum number of moves.

Problem description

The problem involves a row of n security switches that protect a military installation entrance. The switches can be manipulated in specific ways. The rightmost switch can be turned on or off at any time. However, for any other switch in the row, it can only be turned on or off if the switch to its immediate right is already on and all other switches to its right are turned off. Only one switch can be toggled at a time, and the goal of the problem is to turn off all the switches in the minimum number of moves. This problem can be solved using Dynamic Programming techniques.

Solution Description

1. Create a new integer array result of size switches.
2. Set result[0] to 1 and result[1] to 2.
3. For i from 2 to switches-1, do the following:
    - ➢ Set result[i] to result[i-2] + 1 + result[i-2] + result[i-1].
4. Return result[switches-1].

Problem Pseudocode

```
function securitySwitches_minimumMoves(switches):
 result[] = new integer array of size switches
 result[0] = 1
 result[1] = 2
 for i from 2 to switches-1:
 result[i] = result[i-2] + 1 + result[i-2] + result[i-1]
 return result[switches-1]
```

Figure 11 Task 3 Pseudocode

Problem Java Code

```
public class HelloApplication {
    public static int securitySwitches_minimumMoves(int switches) {
    int result[] = new int[switches];
    result[0] = 1; result[1] = 2;
    for (int i = 2;i < switches;i++)
    result[i] = result[i-2] + 1 + result[i-2] + result[i-1];
    return result[switches-1];
    }
    public static void main(String[] args) {
   int switches = 1;
   while (true) {
    System.out.print("Enter number of switches:");
    Scanner input = new Scanner(System.in);
    switches = input.nextInt();
    if(switches == 0) break;
    System.out.println(
   "Minimum number of moves for"
   + switches +
   "switches = "
   + securitySwitches_minimumMoves(switches));
   }
    }
   }
```

Problem Complexity Analysis

- Time complexity : O(n)
- Space complexity : O(n)

Where n is the number of SWITCHES

Problem Output Screenshot

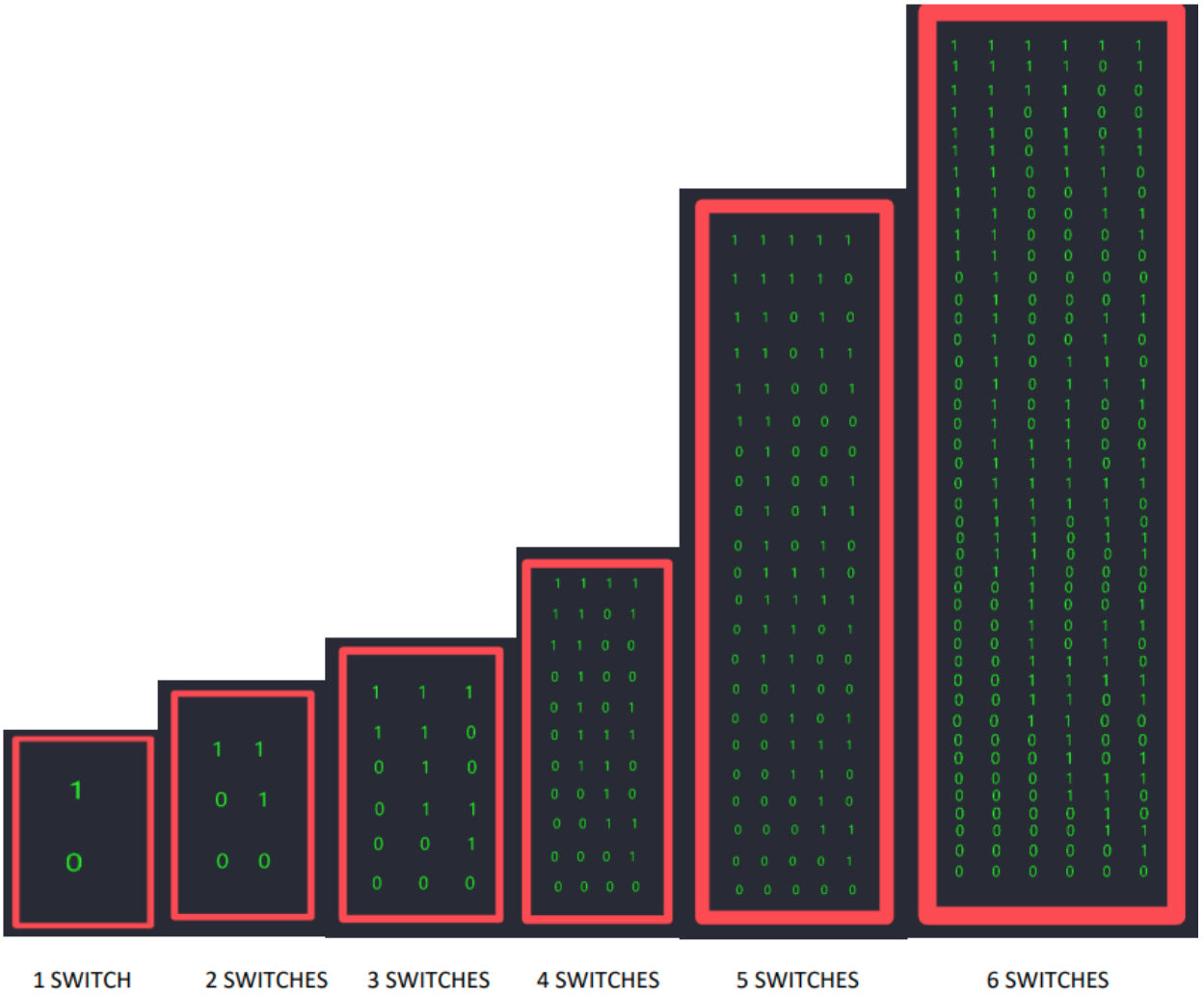


Figure 13 Task 3 | Output moves from 1 to 6

```
"C:\Program Files\Java\jdk-17.0.2\bin\java.exe"
Enter number of switches:1
Minimum number of moves for1switches = 1
Enter number of switches:2
Minimum number of moves for2switches = 2
Enter number of switches:3
Minimum number of moves for3switches = 5
Enter number of switches:4
Minimum number of moves for4switches = 10
Enter number of switches:5
Minimum number of moves for5switches = 21
Enter number of switches:6
Minimum number of moves for6switches = 42
Enter number of switches:0

Process finished with exit code 0
```

Comparison Between Algorithms

Another technique (Recursive Backtracking Algorithm):

This algorithm recursively explores all possible paths by toggling each switch one at a time and checking if the resulting state is valid. If a valid state is found, the algorithm moves on to the next switch and continues the search. If all switches are turned off, the algorithm returns 0. If the end of the switches is reached without finding a solution, the algorithm returns infinity. The algorithm uses memorization to store the states that have already been explored to avoid exploring duplicate states.

Pseudocode:

```
function minMovesRec(backtrackIndex, switchesOn):
 if all switches are off:
 return 0
 else if backtrackIndex >= len(switchesOn):
 return infinity
 minMoves = infinity
 for i from backtrackIndex to len(switchesOn)-1 do
 if canToggle(switchesOn, i):
 toggle(switchesOn, i)
 moves = minMovesRec(i+1, switchesOn)
 minMoves = min(minMoves, moves+1)
 toggle(switchesOn, i)
 return minMoves
```

Figure 14 Task 3 Another technique (Recursive Backtracking Algorithm) Pseudocode

Comparison between algorithms:

Dynamic programming Algorithm:

The dynamic programming method is generally the preferred approach for solving this problem as it provides an optimal solution efficiently by breaking down the problem into smaller subproblems and avoiding duplicate computations.

- Time Complexity: O(n)
- Space Complexity: O(n)

Recursive Backtracking Algorithm:

A recursive backtracking algorithm is a brute-force approach that explores all possible paths by toggling each switch one at a time and checking if the resulting state is valid. This algorithm can be slow for large inputs due to the large number of possible paths to explore. Additionally, the use of memorization to store explored states can require a significant amount of memory. However, this approach can be useful in cases where the input size is small or when the optimal solution is not required.

- Time Complexity: O(2^n)
- Space Complexity: O(n)

Conclusion:

In conclusion, the provided solution aims to solve the problem of turning off a row of security switches in the minimum number of moves using dynamic programming. It utilizes a bottom-up approach and maintains an array result to store the minimum number of moves required for each number of switches.

The solution follows a specific set of rules for manipulating the switches: the rightmost switch can be toggled at will, while any other switch can only be turned on or off if the switch to its immediate right is on and all other switches to its right are off. By applying these rules and calculating the minimum number of moves for each switch configuration, the solution determines the optimal strategy to turn off all the switches.

The time complexity of the solution is O(n), where n is the number of switches, as it iterates through the switches to calculate the minimum number of moves. The space complexity is also O(n) since it uses an array to store the results.

Overall, the solution provides an algorithmic approach to efficiently solve the problem and determine the minimum number of moves required to turn off all the switches in the given configuration.

Task 4

Problem

An evil king is informed that one of his 1000 wine barrels has been poisoned. The poison is so potent that a miniscule amount of it, no matter how diluted, kills a person in exactly 30 days. The king is prepared to sacrifice 10 of his slaves to determine the poisoned barrel.
(a) Can this be done before a feast scheduled in 5 weeks?
(b) Can the king achieve his goal with just eight slaves?
Design a Divide and Conquer algorithm to solve this problem.

Problem Description

There are 1000 wine barrels which one of them has been poisoned, the poison kills the person exactly after 30 days after drinking it. The King wants to serve the wine in a feast after 5 weeks which is 35 days and doesn't want anyone to die so he wants to know which barrel is poisoned using 10 of his slaves to sacrifice them.
The King wants to know if he can determine the poisoned barrel before the feast using only 10 slaves and do it in the most efficient way possible and complete the testing within 30 days before the feast.

Solution Description

a) Yes, it's possible to determine the poisoned barrel using only 10 slaves in less than 5 weeks or 35 days before the feast, this can be solved in exactly 30 days which is before the 5 weeks limit.
Here are the steps to solve this problem using divide and conquer:

1. divide the 1000 barrels into 10 groups with 100 barrels for each like A, B, C,...J
2. each slave will drink from a group (slave 1 drinks from A, slave 2 drinks from B, ..)
3. each slave will drink from the 100 barrels from its group
4. we divide each group of the 100 barrels into 10 groups each with 10 barrels, the groups will be numbered (1,2,3,4,...,10)
5. the first slave will drink from all groups with number 1 from all groups of A, B, C,.. and the second slave will drink from all groups with number 2 and so on
6. divide each group of 10 barrels into 10 groups with single barrels each also numbered from 1 to 10
7. the first slave will drink from all barrels with the number 1 in each group of them all, the second slave will drink form all barrels with number 2 from each group of them, and so on.
8. after 30 days, the first slave dies then we know which 100 barrel is poisoned (EX. if slave No. 6 died then we know that the barrels from 600 to 700 one of them are poisoned)
9. For the second slave to die we know that it's from which group of the 10 groups (Ex. if slave No.5 died then we know that barrels from 650 to 660 one of them are poisoned)
10. For the third slave then we know the exact barrel which is poisoned (Ex. if slave No.1 died then the barrel is No 651)

b) Yes, this problem can also be solved using only 8 slaves as you divide the 1000 barrels into 125 rather than 100 and continue by the same approach.

Problem Assumptions

1. The poison kills the person after exactly 30 days no matter the amount of the poison.
2. We only have 10 slaves to sacrifice them and three or less will die to know what the poisoned barrel is.
3. You have a time limit of 35 (5 weeks) – before the feast – to find the poisoned barrel.

Problem Pseudocode

```
function PoisonedBarrelProblem(slaves, arr)
    barrels = create a list of barrels from 0 to arr.length - 1
    DrinkFromBarrels(slaves, arr, 0, arr.length - 1, barrels)
    return result from sb as an integer

function DrinkFromBarrels(slaves, arr, first, last, barrels)
    if first < last then
        step = (last - first + 1) / slaves.size()
        for i = 0 to slaves.size() - 1 do
            begin = first + step * i
            end = begin + step - 1
            if i == slaves.size() - 1 then
                end = last
            DrinkProcess(slaves[i], arr, begin, end, barrels)
            DrinkFromBarrels(slaves, arr, begin, end, barrels)

function DrinkProcess(slaveNo, arr, start, end, barrels)
    for i = start to end do
        if arr[i] == 1 then
            append (slaveNo - 1) to sb
            return
    return

function main()
    slaves = create a list of integers from 1 to 10
    arr = create an integer array of length 1000 with all values set to 0
    poisonedIndex = generate a random integer between 0 and 999 inclusive
    set arr[poisonedIndex] = 1
    print "The poisoned barrel is at index (random generator) is " + poisonedIndex
    result = PoisonedBarrelProblem(slaves, arr)
    print "The poisoned barrel is at index (from the find poisoned barrel function) is " + result
```

Problem Java Code


```java
import java.util.*;

public class PoisonedBarrelProblem {
    static StringBuilder sb = new StringBuilder();

    public static void DrinkFromBarrels(List<Integer> slaves, int[] arr, int first,
int last, List<Integer> barrels) {
        if (first < last ) {
            int step = (last - first + 1) / slaves.size();
            for (int i = 0; i < slaves.size(); i++) {
                int begin = first + step * i;
                int end = begin + step - 1;
                if (i == slaves.size() - 1) {
                    end = last;
                }
                // make each slave drink the given mix from begin to end
                DrinkProcess(slaves.get(i), arr, begin, end, barrels);
                // recursively make the slaves drink from the remaining untested
barrels
                DrinkFromBarrels(slaves, arr, begin, end, barrels);
            }
        }
    }

    public static int findPoisonedBarrel(List<Integer> slaves, int[] arr) {
        List<Integer> barrels = new ArrayList<>();
        for (int i = 0; i < arr.length; i++) {
            barrels.add(i);
        }
        DrinkFromBarrels(slaves, arr, 0, arr.length - 1, barrels);
        return Integer.parseInt(sb.toString());
    }

    /* function to show who drinks from what and know who dies from the slaves and
who survives */
    public static void DrinkProcess(int slaveNo, int[] arr, int start, int end,
List<Integer> barrels) {
        for (int i = start; i <= end; i++) {
            if (arr[i] == 1) {
                //System.out.println("Slave " + slaveIndex + " drinks from barrel " +
i + " and dies");
                sb.append(slaveNo-1);
                return;
            }
        }
```

```
        //System.out.println("Slave " + slaveIndex + " drinks from barrels " + start
+ "-" + end + " and survives");
    }

    public static void main(String[] args) {
        // create a list of slaves ( 10 slaves )
        List<Integer> slaves = new ArrayList<>();
        for (int i = 1; i <= 10; i++) {
            slaves.add(i);
        }
        // create an array of wine barrels (all of them are 0's except the poisoned
one will be marked as 1)
        int[] arr = new int[1000];
        int poisonedIndex = (int) (Math.random() * 1000); //random generation of the
poisoned barrel
        arr[poisonedIndex] = 1;
        System.out.println("The poisoned barrel is at index ( random generator ) is "
+ poisonedIndex);

        int result = findPoisonedBarrel(slaves, arr);
        System.out.println("The poisoned barrel is at index ( from the find poisoned
barrel function ) is " + result);
    }
}
```

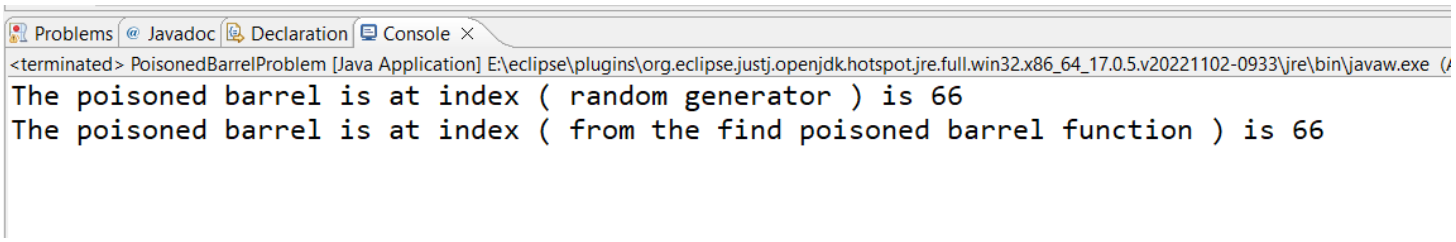
Problem Complexity Analysis

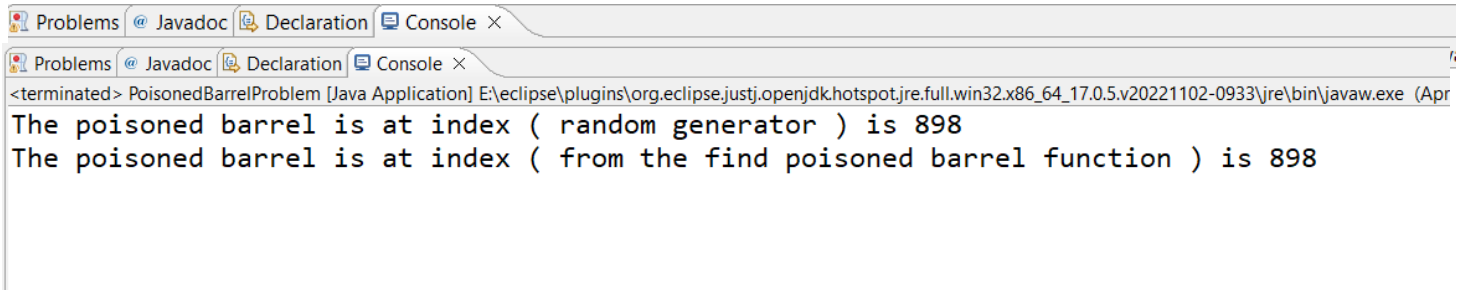
The time complexity of this algorithm depends on the number of slaves and the size of the range of indices to be tested. Let n be the length of the array arr. Since the range of indices to be tested is 0 to n - 1, the DrinkFromBarrels() method will be called O(log n) times, since each recursive call divides the range of indices to be tested in half. For each call to DrinkFromBarrels(), the method performs a loop over the number of slaves, which takes O(s) time, where s is the number of slaves. For each slave, the DrinkProcess() method is called, which takes O(n/s) time to check the part of the range that the slave drinks from.

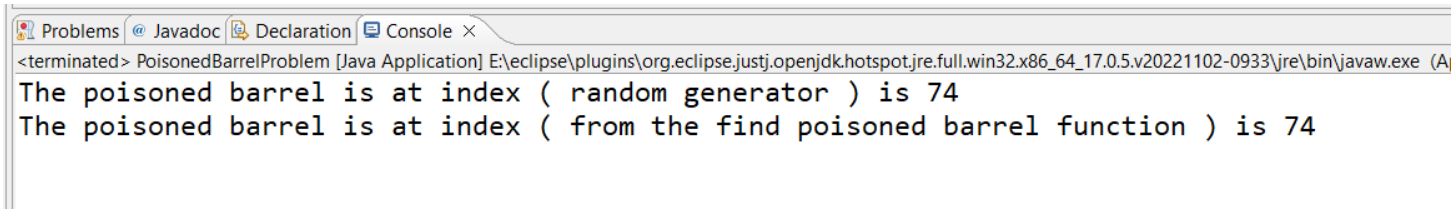
Therefore, the overall time complexity of the algorithm is O(s * log n * (n/s)), which simplifies to O(n log n). This is because the number of slaves s is a constant factor that does not depend on the size of the input array arr. As a result, the time complexity of the algorithm is dominated by the O(log n) factor from the recursive calls to DrinkFromBarrels() and the O(n) factor from the loop inside DrinkProcess().

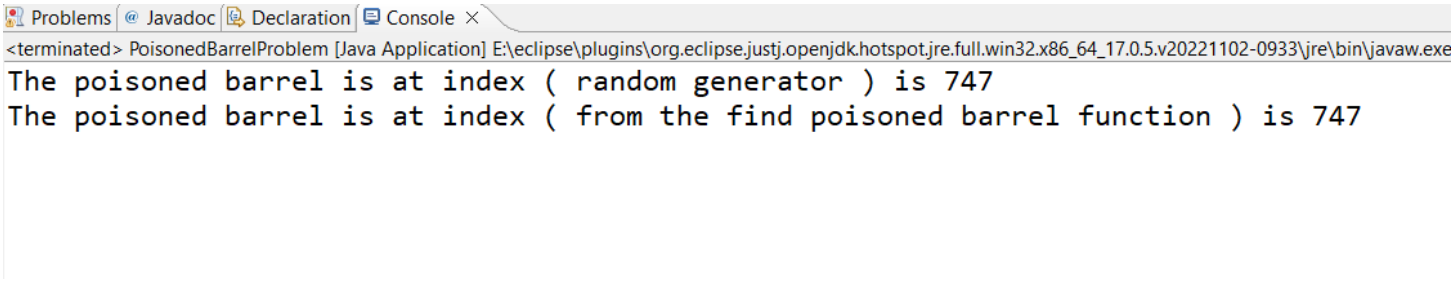
The Time complexity of this solution is O (n log n).

Problem Output screenshots









Other algorithm steps :

```
1.Divide the barrels into groups of 10. This will give us 100 groups of barrels.
2.Take one slave at a time and have them drink from each group of barrels.
3.If the slave dies after drinking from a group of barrels, we know that the poisoned barrel is in
that group.
4.Once we have identified the group with the poisoned barrel, divide the barrels in that group into
groups of 2.
5.Take one slave at a time and have them drink from each group of barrels.
6.If the slave dies after drinking from a group of barrels, we know that the poisoned barrel is in
that group.
7.Continue dividing the barrels in the group with the poisoned barrel into smaller groups of 2 until
we have identified the exact barrel.
```

Description and Complexity of this Algorithm:

This algorithm works by reducing the number of possible poisoned barrels at each step, starting from 1000 and dividing it by 10 at each step until we have identified the exact barrel. The time complexity of this algorithm is O(log n), where n is the number of barrels because we are dividing the number of barrels by 10 at each step (like a binary search).[3]

Other algorithm steps :

```
1.Assign each barrel a unique number from 1 to 1000.
2.Represent each barrel number in binary, using 10 digits (since there are 1000 barrels, which can be
represented using 10 digits in binary).
3.Label each barrel with its corresponding binary representation.
4.Divide the barrels into groups of 10 and have each slave drink from all 10 barrels in each group.
5.For each slave, record the binary representation of the barrel that caused the slave to die.
6.Concatenate the recorded binary representations of each slave to form a 10-digit binary string.
7.Divide the 1000 barrels into groups of 2, based on the position of the corresponding digit in the
binary string:
if the digit is 0, put the barrel in the first group
if the digit is 1, put the barrel in the second group
8.Have each slave drink from both barrels in one of the groups.
9.If a slave dies, the poisoned barrel is in that group. If not, it is in the other group.
10.Continue dividing the group with the poisoned barrel into smaller groups of 2 , based on the next
digit in the binary string , until the exact poisoned barrel is identified.
```

This algorithm also gives a O (n log n)

Conclusion:

- In conclusion, we have explored the problem of determining a single poisoned wine barrel among 1000 barrels using a limited number of slaves and within a specific time constraint. The goal is to identify the poisoned barrel before a feast scheduled in 5 weeks (35 days).
- By employing a Divide and Conquer algorithm, we can efficiently find the poisoned barrel within 30 days, allowing enough time for the necessary preparations before the feast. The algorithm divides the 1000 barrels into groups, starting with 10 groups of 100 barrels each. Each slave is assigned to drink from a specific group of barrels.
- Within each group, the barrels are further divided into smaller groups, initially into 10 groups of 10 barrels each. The slaves continue to drink from specific groups based on a numbering system.
- After 30 days, the first slave will die, indicating the group of 100 barrels where the poisoned barrel is located. The second slave's death will pinpoint the group of 10 barrels, and finally, the third slave's death will reveal the exact poisoned barrel.
- In response to the second part of the problem, it is also possible to achieve the same goal using only 8 slaves. In this case, the 1000 barrels are divided into 125 groups instead of 100, and the algorithm proceeds as before.
- Therefore, the king can determine the poisoned barrel with both 10 and 8 slaves before the feast within the given time limit. The provided Divide and Conquer algorithm ensures an efficient and effective solution to this problem, allowing the king to identify the poisoned barrel and prevent any harm during the feast.

Task 5

Problem:

There are n coins placed in a row. The goal is to form n/2 pairs of them by a sequence of moves. On the first move a single coin has to jump over one coin adjacent to it, on the second move a single coin has to jump over two adjacent coins, on the third move a single coin has to jump over three adjacent coins, and so on, until after n/2 moves n/2 coin pairs are formed. (On each move, a coin can jump right or left but it has to land on a single coin. Jumping over a coin pair counts as jumping over two coins. Any empty space between adjacent coins is ignored.) Determine all the values of n for which the problem has a solution and design an algorithm that solves it in the minimum number of moves for those n's.
Design a greedy algorithm to find the minimum number of moves.

Problem Description

The coin pairing problem involves a row of n coins that need to be paired in n/2 pairs using a sequence of moves. Each move involves a single coin jumping over one or more adjacent coins to land on an unpaired coin until n/2 pairs are formed. The goal is to determine all values of n for which the problem has a solution and to design an algorithm that solves the problem in the minimum number of moves for those values of n.

Solution Description:

The row of coins is handled from both directions
First for n/4 -1 times, the right most single coin is to be paired, so as it, the first pairing the row of coins are all present with no blank spaces or paired coins, so the right most coin is paired with the coin at only one coin separating both, so for a row of 8 coins the first move is as follows {1,1,1,1,1,0,1,2} changing the jumped coin to zero and the paired coin to 2, continue working the same way for the right most single coin with increasing the number of separating coins in the middle for n/4-1 times which in this example is one so right side is complete
Secondly for the remaining n/4 to n/2 times (to make pairs only half the coins need to move), the leftmost coin will move by n/4 checking if there is any white spaces to ignore along the way also adding the coin count along the way instead of index only so that if there is a pair it will be counted as jumping over 2 coins
so for the first leftmost coin in a row of 8 coins {0,1,1,2,1,0,1,2}
Then the second leftmost jumping over 3 coins {0,0,1,2,2,0,1,2}
Third leftmost most jumping over 4 coins {0,0,0,2,2,0,2,2}
The Leftmost Single coins are complete
All coins are paired successfully.

Problem Assumptions

1. The number of coins must be divisible by 4, it can't be odd as the final output must be pairs. But it has to be divisible by 4 because if it is just even there will not be the right amount of coins to do the jumps in the required sequence. Ex: if n = 6 {1, 1, 1, 1, 1, 1} If any coin jumps over one coin: {1, 1, 1, 0, 1, 2} Then the first coin will be the only one able to jump over 2 {0, 1, 1, 0, 2, 2} Leaving 2 coins that will never be able to jump over 3 coins to pair so the puzzle will fail no matter which coin moves first
2. 1 represents the present coin , 0 represents empty coin space and 2 represents a pair of coins

Problem Pseudocode

```
function coinPairingGreedy(n):
    coins = array of n integers initialized to 1
    nocount = 0

    // Check if n is divisible by 4
    if n % 4 != 0:
        print "Cannot solve the puzzle for n = " + n + ". n must be divisible by 4."
        return

    // Loop through the values of i from 1 to n/4 - 1:
    for i = 1 to n/4 - 1:
        // Find the rightmost unpaired coin that has i unpaired coins to its left.
        rightmostSingleCoin = findRightmostSingleCoinWithIcoinsToItsLeft(n-i)
        // If such a coin exists, jump it to the leftmost unpaired coin to its left.
        if rightmostSingleCoin != -1:
            jumpCoin(rightmostSingleCoin - i - 1 - countEmptySpacesToTheLeft(rightmostSingleCoin),
rightmostSingleCoin, coins, nocount)

    // Loop through the values of i from n/4 to n/2:
    for i = n/4 to n/2:
        // Find the leftmost unpaired coin that has n/2-i unpaired coins to its left.
        leftmostSingleCoin = i - n/4
        // If such a coin exists, jump it to the rightmost unpaired coin to its right.
        if leftmostSingleCoin != -1:
            coinscount = 0
            extraindex = 0
            for v = leftmostSingleCoin + 1 to n-1:
                if coinscount + coins[v] <= i:
                    coinscount = coinscount + coins[v]
                    extraindex = extraindex + 1
                else:
                    break
            jumpCoin(leftmostSingleCoin, leftmostSingleCoin + extraindex + 1, coins, nocount)

    // Print the final state of the coins array and the number of jumps made.
    print "Final state of the puzzle with " + n + " coins paired: " + coins
    print "With min number of moves equals: " + nocount

function findRightmostSingleCoinWithIcoinsToItsLeft(i):
    for j = i to 0:
        if coins[j] == 1:
            return j
    return -1 // Error: cannot find a rightmost single coin with i coins to its left

function countEmptySpacesToTheLeft(index):
    count = 0
    for i = index - 1 to 0:
        if coins[i] == 0:
            count = count + 1
    return count

function jumpCoin(fromIndex, toIndex, coins, nocount):
    coins[toIndex] = 2
    coins[fromIndex] = 0
    nocount = nocount + 1
```

Figure 22 Task 5 Pseudocode

Problem Java Code

```
import java.util.Arrays;

public class CoinPairingGreedy {
    public static void main(String[] args) {
    int n = 4;
    // Check if n is divisible by 4
    if (n % 4 != 0) {
        System.out.println("Cannot solve the puzzle for n = " + n + ". n must be divisible by
4.");
        return;
    }
     initializeCoins(n);

    for (int i = 1; i <= n/4 - 1; i++) {
        int rightmostSingleCoin = findRightmostSingleCoinWithIcoinsToItsLeft(n-i);
        if (rightmostSingleCoin!= -1){
        jumpCoin(rightmostSingleCoin- i -1 -
countEmptySpacesToTheLeft(rightmostSingleCoin),rightmostSingleCoin);
        }
    }

    for (int i = n/4; i <= n/2; i++) {

        int leftmostSingleCoin = i - n/4;
        if (leftmostSingleCoin != -1){
            int coinscount =0;
            int extraindex = 0;
            for (int v =leftmostSingleCoin+1; v<n;v++){
            if (coinscount + coins[v] <= i){
                coinscount = coinscount + coins[v];
                extraindex ++;
            }
            else{break;}
            }
        jumpCoin(leftmostSingleCoin, leftmostSingleCoin+extraindex+1);

        }
    }

    System.out.println("Final state of the puzzle with " + n + " coins paired: " +
Arrays.toString(coins));
    System.out.println("With min number of moves equals: " + nocount);
    }




private static int[] coins;
private static int nocount =0;

private static void initializeCoins(int n) {
    coins = new int[n];
    for (int i = 0; i < n; i++) {
        coins[i] = 1;
    }
```


```
}

private static int findRightmostSingleCoinWithIcoinsToItsLeft(int i) {
    for (int j = i; j >= 0; j--) {
        if (coins[j]==1 ) {
            return j;
        }
    }
    return -1; // Error: cannot find a rightmost single coin with I coins to its left
}

private static int countEmptySpacesToTheLeft(int index) {
    int count = 0;
    for (int i = index-1; i >= 0; i--) {
        if (coins[i] == 0) {
            count++;
        }
    }
    return count;
}

private static void jumpCoin(int fromIndex, int toIndex) {

    coins[toIndex] = 2;
    coins[fromIndex] = 0;
    nocount ++;
}

}
```

Problem Complexity Analysis

Basic operator is the comparison.
Complexity analysis for the helping functions first:
jumpCoin function has no comparisons ie : O(1)
countEmptySpacesToTheLeft : $\sum_{0}^{index-1} 1 = index - 1$ where index < n , so O(n)
findRightmostSingleCoinWithIcoinsToItsLeft:: $\sum_{0}^{i} 1 = i$ where i < n, so O(n)
initializeCoins has no Comparisons : O(1)
For the main function:
1st comparison O(1)
Then : O(1) + O(1) + $\sum_{1}^{n/4} n + \sum_{n/4}^{n/2} n = \sum_{1}^{n/2} n = \sum_{0}^{n/4} n - \sum_{0}^{1} n$ = n/4 * n – n = $n^2/4 - n$
ie : $O(n^2)$

Problem Output screenshots

for a row of 4 coins:

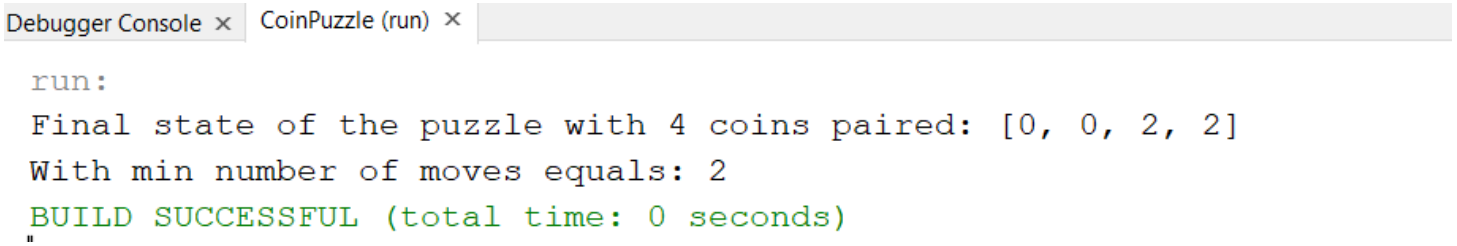


Figure 23 Task 5| Output 1

for a row of 8 coins:

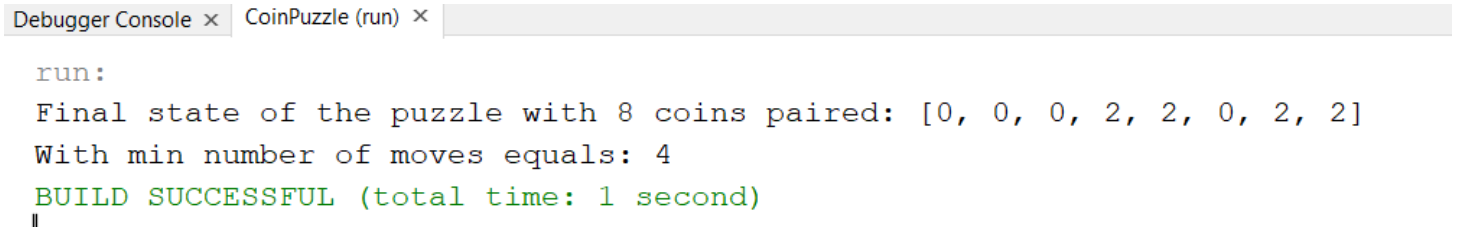


Figure 24 Task 5| Output 2

For a row of 16 coins:

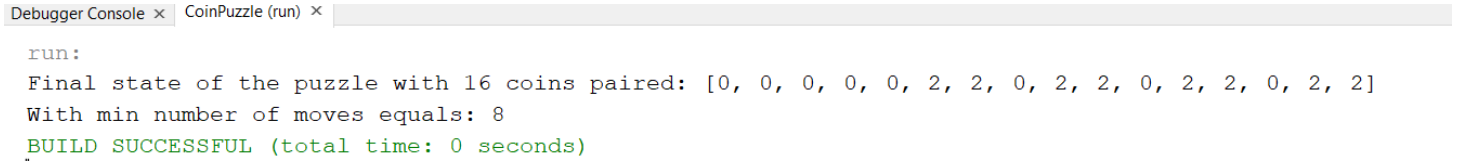


Figure 25 Task 5| Output 3

For a number not divisible by 4 like 6 coins:

Debugger Console × CoinPuzzle (run) ×

```
run:
Cannot solve the puzzle for n = 6. n must be divisible by 4.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 26 Task 5| Output 4

Comparison Between Algorithms

Another algorithm steps (Brute Force) :

```
1.Initialize an array called coins with size n and set each element to 1.
2.Initialize a variable called minMoves to infinity.
3.Generate all possible pairings of the coins using a nested loop that iterates through every
possible pair of coins. If both coins are unpaired, add the pair to a list called pairings.
4.Loop through each pairing in pairings. For each pairing:
   Create a temporary copy of the coins array.
   Initialize a variable called moves to 0.
   Loop through each pair of indices in the pairing and call a function called jumpCoin for each
   pair. The jumpCoin function should move a coin from one position to another and return the number
   of moves required.
   Add the number of moves returned by jumpCoin to moves.
   If moves is less than minMoves, set minMoves to moves and update the coins array with the
   temporary copy.
5.Print the final state of the coins array and the minimum number of moves required (minMoves).
```

Figure 27 Task 5| Brute Force Steps

Brute Force pseudocode:

```
function coinPairingBruteForce(n):
    // Initialize an array of size n called "coins" with each element set to 1.
    coins = array of n integers initialized to 1
    minMoves = infinity

    // Generate all possible pairings of the coins.
    pairings = generateAllPairings(n)

    // Loop through each pairing and calculate the number of moves required to pair all the coins.
    for pairing in pairings:
        tempCoins = coins.copy()
        moves = 0
        for pair in pairing:
            moves = moves + jumpCoin(pair[0], pair[1], tempCoins)
        if moves < minMoves:
            minMoves = moves
            coins = tempCoins

    // Print the final state of the coins array and the number of jumps made.
    print "Final state of the puzzle with " + n + " coins paired: " + coins
    print "With min number of moves equals: " + minMoves

function generateAllPairings(n):
    // Generate all possible pairings of the coins.
    pairings = []
    for i = 0 to n-1:
        for j = i+1 to n-1:
            if (coins[i] == 1 and coins[j] == 1):
                pairings.append([i,j])
    return pairings

function jumpCoin(fromIndex, toIndex, coins):
    if coins[fromIndex] == 0 or coins[toIndex] == 2:
        return 0
    coins[toIndex] = 2
    coins[fromIndex] = 0
    return 1
```

Figure 28 Task 5| Brute Force PseudoCode

Description and Complexity of Brute force algorithm:

The brute force approach generates all possible pairings of the coins and calculates the minimum number of moves required for each pairing. It then returns the minimum number of moves required among all the pairings. This approach is exhaustive and guarantees an optimal solution
Time complexity: $O(n * 2^{n/2})$, Space complexity: O(n)

Another algorithm steps (Dynamic Programming):

```
function coinPairingDynamic(n):
    // Initialize an array of size n called "coins" with each element set to 1.
    coins = array of n integers initialized to 1

    // Initialize a 2D array called "dp" with size n x n, filled with infinity.
    dp = 2D array of size n x n filled with infinity

    // Set dp[i][i] = 0 for all i.
    for i = 0 to n-1:
        dp[i][i] = 0

    // Loop through all possible subarrays of the coins array.
    for len = 2 to n:
        for i = 0 to n-len:
            j = i + len - 1
            // Case 1: Pair the ith coin with the jth coin directly.
            if coins[i] == 1 and coins[j] == 1:
                dp[i][j] = dp[i+1][j-1] + 1
            // Case 2: Pair the ith coin with another coin between i+1 and j-1.
            for k = i+1 to j-1:
                dp[i][j] = min(dp[i][j], dp[i][k-1] + dp[k+1][j-1] + (coins[i] + coins[j] == 2))

    // Print the final state of the coins array and the number of jumps made.
    print "Final state of the puzzle with " + n + " coins paired: " + coins
    print "With min number of moves equals: " + dp[0][n-1]

function min(a, b):
    if a < b:
        return a
    else:
        return b
```

Figure 29 Task 5| Dynamic Programming Pseudocode

Description and Complexity of Dynamic Programming Algorithm:

The dynamic programming approach breaks down the problem into smaller sub-problems and uses the solutions to those sub-problems to build up to the solution to the original problem. Specifically, the dynamic programming approach uses a 2D array to store the minimum number of moves required to pair all the coins in each subarray of the coins array. It fills in the 2D array using a nested loop that iterates through all possible subarrays of the coins array. For each subarray, the dynamic programming approach considers two cases: either the first and last coins in the subarray are paired directly, or they are paired indirectly through another coin. The dynamic programming approach uses the solutions to smaller subproblems (i.e., the minimum number of moves required to pair the coins between the first and last coins in the subarray) to solve the larger problem (i.e., the minimum number of moves required to pair all the coins in the subarray)

Time complexity: $O(n^3)$, Space complexity: $O(n^2)$.

Comparison:

1. Greedy solution:

The greedy solution uses a simple algorithm to pair the coins as efficiently as possible. The time complexity of the algorithm is $O(n^2)$ since we need to loop through all the coins and perform a constant-time operation for each coin. The space complexity of the greedy solution is O(n) since we only need to store the original coins array.

- Time complexity: $O(n^2)$
- Space complexity: O(n)
- May not produce the optimal solution (but it does in this scenario)
- Efficient for all sizes of n

2. Brute force solution:

The brute force solution generates all possible pairings of the coins and calculates the minimum number of moves required for each pairing. The time complexity of generating all possible pairings is $O(2^{n/2})$, since there are $2^{n/2}$ possible pairs of coins, and we need to generate all possible combinations of these pairs. The time complexity of calculating the minimum number of moves required for each pairing is O(n) since we need to loop through all the coins and perform a constant-time operation for each coin. The space complexity of the brute force solution is O(n) since we only need to store the original coins array and a temporary copy of the coins array for each pairing.

- Time complexity: $O(n * 2^{n/2})$
- Space complexity: O(n)
- Guarantees optimal solution
- Inefficient for large values of n

3. Dynamic programming solution:

The dynamic programming solution uses a 2D array of size n x n to store the minimum number of moves required to pair all the coins in each subarray of the coins array. The time complexity of filling the 2D array is $O(n^3)$ since we need to loop through all possible subarrays of the coins array and perform a constant-time operation for each subarray. The space complexity of the dynamic programming solution is $O(n^2)$ since we need to store the 2D array.

- Time complexity: $O(n^3)$
- Space complexity: $O(n^2)$
- Guarantees optimal solution
- Efficient for small to medium-sized values of n

Conclusion:

Greedy Algorithm is the better option amongst those algorithms as it has better time complexity with high accuracy , followed by dynamic programming approach that guarantees an optimal solution then brute force algorithm which is very easy to implement but very inefficient.

Task 6

Problem

There are 12 coins identical in appearance; either all are genuine or exactly one of them is fake. It is unknown whether the fake coin is lighter or heavier than the genuine one. You have a two-pan balance scale without weights. The problem is to find whether all the coins are genuine and, if not, to find the fake coin and establish whether it is lighter or heavier than the genuine ones.
Design a Dynamic Programing algorithm to solve the problem in the minimum number of weighings.

Problem Description

You have 12 coins, one of them may be fake and either lighter or heavier than the genuine coins. Without any prior knowledge about the fake coin's weight, you are tasked with identifying the fake coin and determining whether it is lighter or heavier than the genuine coins. To do this, you have access to a two-pan balance scale that can compare the weight of two groups of coins or the weight of one coin against another. Your objective is to determine the identity of the fake coin while minimizing the number of weighings required.
To solve this problem optimally, you can use dynamic programming to find the most efficient strategy for weighing the coins. It is important to assume that the coins are not damaged or altered during the weighing process, and that the scale is accurate and precise. Additionally, you can perform the weighings as many times as needed to find the solution. The ultimate goal is to determine the minimum number of weighings required to identify the fake coin and its weight relative to the genuine coins.

Assumptions

First, If the scale is accurate and precise, the weight difference between the pans and the weight difference between the coins being weighed should be exactly proportionate.
Second, It is reasonable to suppose that we can do weighings as often as necessary, but we like to reduce the number of weighings needed.
Third, Although we can theoretically split the coins into as many groups as necessary to complete the weighings, we want to utilise the most effective method possible.
Finally, By comparing a coin to a genuine coin of known weight, we may presumably tell if it is lighter or heavier and We may expect that the weighings will be done accurately and impartially, and that the coins won't be harmed or changed in any way.

Solution Description

1) Create the find_fake_coin function, which accepts a list of coin_weights as an argument. And then Establish the coin_weights list's length and assign it to the variable n.
2) Create a 2D list named dp that is n × n in size and initialise each entry to "=."
3) Use two stacked loops to iterate over all potential coin combinations:
The inner loop iterates across the range from 0 to n-1 and uses the variable j as the index for the second coin.
The outer loop iterates over the range from 0 to n-1 as well.

4) Examine three scenarios inside the loops to ascertain how the coin weights relate to one another:
First Case: Set dp[i][j] to '' to indicate that the false coin is lighter if i > 0 and coin_weights[i] coin_weights[j].
Second Case: Set dp[i][j] to '>' to indicate that the false coin is heavier if j > 0 and coin_weights[i] >

coin_weights[j].
Third Case: Set dp[i][j] to '=' to indicate that the weighing does not reveal any information about the false coin if i > 0 and j > 0.
5) Change the variable sign to the row of dp that indicates the second coin's weight in relation to the other coins, dp[1].then In order to identify the sign of the counterfeit coin (>,, or =), use the function check_sign with the sign list as argument.
6) To get the index of the false coin, call the check_index function with the sign list as input.
7) Based on the sign result, examine the following three scenarios to identify the type of false coin:
Return a tuple with the fake coin index and the text "lighter" if the sign is ">".
Return a tuple with the fake coin index and the text "heavier" if the sign is "".
Return the string "there is no fake coin" if the sign is "=".

8) Describe the check_sign function, which accepts an array as input. And Set the previous_sign variable's initial value to null.
9) Repeatedly go over the array's elements:
• Return the current sign if previous_sign is not None and the current sign differs from the prior sign.
• Change previous_sign to reflect the present sign.
10) Return None if a different symbol cannot be discovered.
11) Describe the check_index function, which accepts an array as input.
12) Set the previous_sign variable's initial value to null.
Iterate over the array's items and their indexes 16 times.
• Return the current index if previous_sign is not None and the current sign differs from the prior sign.
• Change previous_sign to reflect the present sign.
13) Return None if a different symbol cannot be identified. Then Define a list of coins with the same weights outside of the functions (for example, [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]).
14) Use coins as the input for the function find_fake_coin, and then assign the outcome to the variable result. And Print the result's value to show the outcome.[1,2]

Problem Pseudocode

```
function main()
        int[] coins = {2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2}
        result = findFakeCoin(coins)
        print(result)


function findFakeCoin(int[] coinWeights)
        int n = coinWeights.length
        // Initialize the dynamic programming table
        // dp[i][j] represents the result of weighing coins i and j
        char[][] dp = new char[n][n]
        // Iterate through all possible combinations of coins
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                // Case 1: Fake coin is lighter
                if (i > 0 && coinWeights[i] < coinWeights[j])
                    dp[i][j] = '<'

                // Case 2: Fake coin is heavier
                else if (j > 0 && coinWeights[i] > coinWeights[j])
                    dp[i][j] = '>'

                // Case 3: Fake coin is not in the current weighing
                else if (i > 0 && j > 0) {
                    dp[i][j] = '='

        char[] sign = dp[1]
        String result = checkSign(sign)
        int index = checkIndex(sign)
        if (result != null)
            if (result.equals(">"))
                return String.format("(%d, lighter)", index)
             else if (result.equals("<")) {
                return String.format("(%d, heavier)", index)

        else
            return "there is no fake coin"

        return null
```

```
function checkSign(char[] array)
    char previousSign = '\0'

    for (char sign : array)
        if (previousSign != '\0' && sign != previousSign)
            return Character.toString(sign);

        previousSign = sign

    return null

 function checkIndex(char[] array)
    char previousSign = '\0'

    for (int i <-- 0; i < array.length; i++)
        char sign = array[i];
        if (previousSign != '\0' && sign != previousSign)
            return i

        previousSign = sign

    return -1
```

Problem Java Code

```
package project;
import java.util.Arrays;
public class Project {
        public static void main(String[] args) {
            int[] coins = {2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2};
            String result = findFakeCoin(coins);
            System.out.println(result);
        }

        public static String findFakeCoin(int[] coinWeights) {
            int n = coinWeights.length;

            // Initialize the dynamic programming table
            // dp[i][j] represents the result of weighing coins i and j
            char[][] dp = new char[n][n];

            // Iterate through all possible combinations of coins
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    // Case 1: Fake coin is lighter
                    if (i > 0 && coinWeights[i] < coinWeights[j]) {
                        dp[i][j] = '<';
                    }
                    // Case 2: Fake coin is heavier
                    else if (j > 0 && coinWeights[i] > coinWeights[j]) {
                        dp[i][j] = '>';
                    }
                    // Case 3: Fake coin is not in the current weighing
                    else if (i > 0 && j > 0) {
                        dp[i][j] = '=';
                    }
                }
            }

            char[] sign = dp[1];
            String result = checkSign(sign);
            int index = checkIndex(sign);
            if (result != null) {
                if (result.equals(">")) {
                    return String.format("(%d, lighter)", index);
                } else if (result.equals("<")) {
                    return String.format("(%d, heavier)", index);
                }
            } else {
                return "there is no fake coin";
            }
            return null;
            }

        public static String checkSign(char[] array) {
            char previousSign = '\0';

            for (char sign : array) {
                if (previousSign != '\0' && sign != previousSign) {
                    return Character.toString(sign);
                }
```

```
                previousSign = sign;
            }
            return null;
        }

        public static int checkIndex(char[] array) {
            char previousSign = '\0';

            for (int i = 0; i < array.length; i++) {
                char sign = array[i];
                if (previousSign != '\0' && sign != previousSign) {
                    return i;
                }
                previousSign = sign;
            }
            return -1;
        }

}
```

Problem Complexity Analysis:

The time complexity of the given code is O(n^2), where n is the length of the input array.

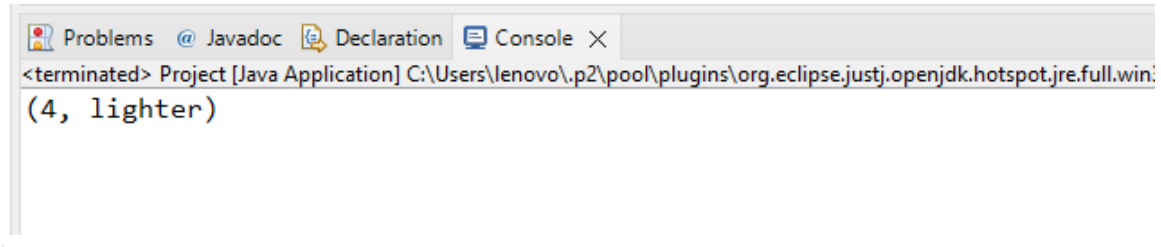
The code uses dynamic programming to solve the 12-coin weighing puzzle. It initializes a 2D array dp to store the results of all possible weighings between pairs of coins. The algorithm iterates through all possible pairs of coins and performs comparisons to determine the relationships between them. This requires nested loops that iterate through the input array twice, resulting in a time complexity of O(n^2).

The functions checkSign and checkIndex each iterate through the input array once, resulting in a time complexity of O(n) for each function. However, since they are both called only once in the main function, their overall contribution to the time complexity is negligible.

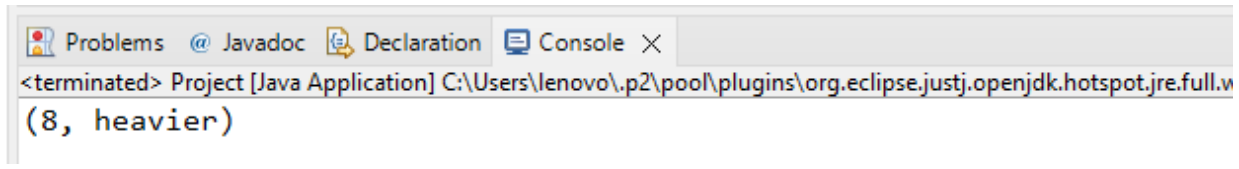
Therefore, the overall time complexity of the code is O(n^2).

Problem Output screenshots:

Input: {2,2,2,2,1,2,2,2,2,2,2,2}



Input: {2,2,2,2,2,2,2,2,3,2,2,2}



Input: {2,2,2,2,2,2,2,2,2,2,2,2}

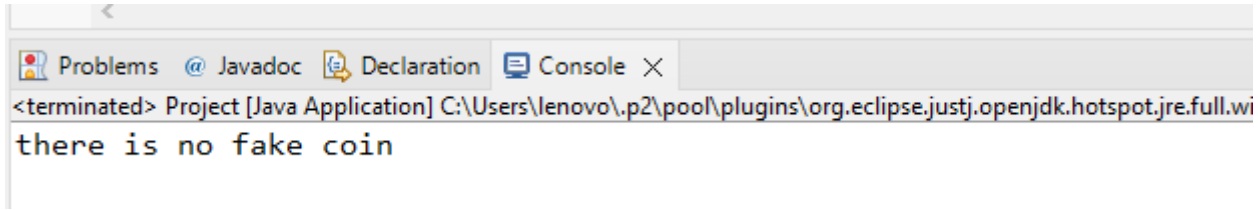


Figure 34 Task 6| Output 3

Comparison Between Algorithms

First by using Brute-Force:
Using brute force is another approach to solving this issue.
When using a brute force strategy to solve the fake coin problem, all potential coin combinations would be tested until the false coin was found. In the worst-case situation, if we have eight coins, for instance, a brute force strategy would entail weighing two coins at a time until we found the false coin. Since the brute force approach entails weighing every conceivable combination of coins until the false coin is found, it can call for additional weighings. However, it is a simple strategy that can be put into practise with little computational assistance.[6]

Second by using Dynamic Programming:
On the other hand, the technique used in dynamic programming entails segmenting the issue into smaller subproblems and addressing each one separately. Due to the fact that each subproblem is only solved once when using dynamic programming, less calculations may be necessary. However, storing the outcomes of each subproblem can demand additional computer power.

Problem Pseudocode (Brute-Force)

```
function main()
    int[] coins = {2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2}
    Object result = findFakeCoin(coins)

    if (result instanceof String)
        System.out.println(result)
     else if (result instanceof Object[])
        Object[] arrResult = (Object[]) result
        for (Object obj : arrResult)
            System.out.print(obj + " ")

function findFakeCoin(int[] coins)
    for (int i <-- 0; i < coins.length; i++) do
        if (i == 0)   // Check if the first coin is fake
            if (coins[1] == coins[2])
                if (coins[0] < coins[1])
                    return new Object[]{1, "lighter"}
                 else if (coins[0] > coins[1])
                    return new Object[]{1, "heavier"}
                 else
                    return "there is no fake coin"
       end for
        for (int j <-- i + 1; j < coins.length; j++) do
            char result = weigh(new int[]{coins[i]}, new int[]{coins[j]})
            if (result == '<')
                int fakeCoin = j + 1
                String fakeCoinWeight = "heavier"
                return new Object[]{fakeCoin, fakeCoinWeight};
             else if (result == '>') {
                int fakeCoin = j + 1
                String fakeCoinWeight = "lighter"
                return new Object[]{fakeCoin, fakeCoinWeight}
       end for
    return "there is no fake coin"
```

```
function weigh(int[] left, int[] right)
            int leftWeight = sum(left)
            int rightWeight = sum(right)
            if (leftWeight < rightWeight)
                return '<'
            else if (leftWeight > rightWeight)
                return '>'
            else
               return '='

    function sum(int[] array)
            int total = 0
            for (int i : array)
                total += i

            return total
```

Conclusion:

Dynamic programming and brute force are both approaches to solving problems, but they differ in their efficiency. Brute force involves trying every possible solution to a problem until the correct one is found. Dynamic programming, on the other hand, breaks down a problem into smaller subproblems. Overall, dynamic programming can be more efficient than brute force because it avoids redundant computations and takes advantage of overlapping subproblems to reduce the overall complexity of the problem.

Task 7

Problem

A computer game has a shooter and a moving target. The shooter can hit any of n > 1 hiding spots located along a straight line in which the target can hide. The shooter can never see the target; all he knows is that the target moves to an adjacent hiding spot between every two consecutive shots. Design a greedy algorithm that guarantees hitting the target.

Problem Description

There is a computer game with a shooter and a moving target. The shooter is positioned at a fixed location, and the target can hide in any of n > 1 hiding spots located along a straight line. The shooter cannot see the target's current location but knows that after two shots, the target moves to one of the adjacent hiding spots. The shooter can only shoot at one hiding spot at a time and needs to hit the target to win the game.
The task is to design a greedy algorithm that guarantees hitting the target. In other words, the algorithm should be able to hit the target no matter how the target moves after each shot, as long as the target does not leave the line of hiding spots. The algorithm should be as efficient as possible, minimizing the number of shots required to hit the target.[1]

Solution Description

a simple shooting game where the player tries to hit a randomly chosen target spot on a line of spots, the player always starts from the most left point to ensure hitting the target. Here's a more detailed description of how the code works:

The “main” function first prompts the user to enter the number of spots on the line. It then generates a random integer between 1 and the number of spots, which represents the target spot that the player must hit.

The “ShootTarget” function is called with two arguments: the number of spots on the line and the target spot.

Inside the “ShootTarget” function, a “Random” object is created to generate random numbers. A variable “step” and a variable “trial” are also initialized to 0.

The function then enters a loop that iterates over each spot on the line from 1 to “spots”. For each spot, it increments the “trial” counter and generates a random “step” value that is either -1 or +1.

If the current spot is equal to the target spot or the next spot is equal to the target spot, the function prints a message indicating that the player has hit the target, along with the number of trials it took to do so. It then returns from the function.

If the current spot is not equal to the target spot, the function prints a message indicating that the target is still hiding and calculates the next target spot based on the current target spot and the random “step” value. If the target is at the first or last spot on the line, it is moved one spot in the opposite direction. Otherwise, it is moved one spot in the direction of the “step”.

After calculating the next target spot, the function increments the “trial” counter and prints a message indicating that the target has moved to the new spot.

If the loop completes without hitting the target, the function prints a message indicating that the player has failed to hit the target.

Problem Assumptions

1. The hiding spots are evenly spaced along the line, and the shooter can shoot at any hiding spot with equal accuracy. If the hiding spots are not evenly spaced, or if the shooter has different levels of accuracy depending on the distance to the hiding spot, the algorithm may not be as effective.
2. The algorithm assumes that the target always moves to one of the adjacent hiding spots after two shots. If the target is able to move to any hiding spot on the line, the algorithm may not work.
3. The algorithm assumes that the shooter has an infinite supply of ammunition and can continue shooting until the target is hit. In a real-world scenario, the shooter would have a limited number of shots, and the algorithm would need to be modified to take this into account.

Problem Pseudocode

```
function ShootTarget(spots, target):
    create a new Random object
    set step to 0
    set trial to 0

    for i from 1 to spots:
        increment trial by 1
        set step to (randomly choose 0 or 1, then multiply by 2 and subtract 1)

        if i is equal to target, or i+1 is equal to target:
            if i is equal to target:
                print "Trial [trial] | Hits: [i] | shoot succeeded at -> [i]"
                print "you hit the target in spot [i] in [trial] trials"
            else:
                increment trial by 1
                print "Trial [trial] | Hits: [i] | target still to hide in: [target]"
                increment trial by 1
                print "Trial [trial] | Hits: [i+1] | shoot succeeded at -> [i+1]"
                print "you hit the target in spot [i+1] in [trial] trials"
            return

        print "Trial [trial] | Hits: [i] | target still to hide in: [target]"
        if target is equal to 1:
            increment target by 1
        else if target is equal to spots:
            decrement target by 1
        else:
            increment target by step
        increment trial by 1
        print "Trial [trial] | Hits: [i+1] | target just moved to hide in: [target]"

    print "Failed to shoot the target"


function main():
    print "Enter number of spots:"
    create a new Scanner object
    read in a value for spots from user input
    create a new Random object
    set target to a random integer between 1 and spots
    call ShootTarget(spots, target)
```

Problem Java Code

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

public class Main {
    public static void ShootTarget (int spots, int target) {
        Random random = new Random();
        int step;
        int trial=0;
        for (int i = 1; i <= spots; i++)
        {
            ++trial;
            step = random.nextInt(2) * 2 - 1;
            if(i==target || (i+1 == target))
            {
                if(i==target){
                    System.out.println("Trial "+trial +" | Hits :" + i + " | shoot succeeded
at -> " + i);
                    System.out.println("you hit the target in spot "+(i) +" in "+trial+"
trials");
                }
                else{
                    System.out.println("Trial "+trial + " | Hits :" + i + " | "+ "target
still to hide in :"+target);
                    trial++;
                    System.out.println("Trial "+trial +" | Hits :" + (i+1) +" | shoot
succeeded at -> " + (i+1));
                    System.out.println("you hit the target in spot "+(i+1) +" in "+trial+"
trials");
                }
                return;
            }
            System.out.println("Trial "+trial +" | Hits :" + i + " | "+ "target still to hide
in :"+target);
            if(target == 1)
            {
                target++;
            }
            else if(target == spots)
            {
                target--;
            }
            else
            {
                target+=step;
            }
            trial++;
            System.out.println("Trial "+trial +" | Hits :" + (i+1) +" | "+ "target just moved
to hide in :"+target);
        }
        System.out.println("Failed to shoot the target");;
    }
    public static void main(String[] args) {
        System.out.print("Enter number of spots:");
        Scanner input = new Scanner(System.in);
        int spots = input.nextInt();
        Random random = new Random();
```

```
        int target= random.nextInt(spots)+1 ;
        ShootTarget(spots,target);
    }
}
```

Problem Complexity Analysis

The time complexity of the ShootTarget function can be analyzed by considering the operations performed within its loop, which iterates over each spot on the line. Since this loop runs spots times, its time complexity is O(spots). Within the loop, the function performs a constant number of operations, including generating a random number, performing simple arithmetic, and printing messages. Therefore, the total time complexity of the loop is also O(spots).

If the target spot is hit, the function returns from the loop and the total time complexity of the ShootTarget function is O(1). Otherwise, if the loop completes without hitting the target, the function prints a message indicating that the player has failed to hit the target, which takes constant time, or O(1).

The main function prompts the user to enter the number of spots on the line and generates a random target spot. These operations take constant time, or O(1). The ShootTarget function is then called with a constant number of arguments, so the time complexity of the entire program is O(spots).

Therefore, **the time complexity of the entire program is O(n)**, and the **space complexity is O(1)** since the program uses only a constant amount of memory regardless of the size of the input.

Problem Output screenshots

NOTE: the shooter doesn't know where the target hides, but we display it just for tracking and checking the algorithm is working well

```
Enter number of spots:2
Trial 1 | Hits :1 | target still to hide in :2
Trial 2 | Hits :2 | shoot succeeded at -> 2
you hit the target in spot 2 in 2 trials
```

Figure 38 Task 7 | Output 1

```
Enter number of spots:12
Trial 1 | Hits :1 | target still to hide in :5
Trial 2 | Hits :2 | target just moved to hide in :6
Trial 3 | Hits :2 | target still to hide in :6
Trial 4 | Hits :3 | target just moved to hide in :7
Trial 5 | Hits :3 | target still to hide in :7
Trial 6 | Hits :4 | target just moved to hide in :8
Trial 7 | Hits :4 | target still to hide in :8
Trial 8 | Hits :5 | target just moved to hide in :7
Trial 9 | Hits :5 | target still to hide in :7
Trial 10 | Hits :6 | target just moved to hide in :8
Trial 11 | Hits :6 | target still to hide in :8
Trial 12 | Hits :7 | target just moved to hide in :9
Trial 13 | Hits :7 | target still to hide in :9
Trial 14 | Hits :8 | target just moved to hide in :8
Trial 15 | Hits :8 | shoot succeeded at -> 8
you hit the target in spot 8 in 15 trials
```

Figure 39 Task 7 | Output 2

```
Enter number of spots:8
Trial 1 | Hits :1 | target still to hide in :8
Trial 2 | Hits :2 | target just moved to hide in :7
Trial 3 | Hits :2 | target still to hide in :7
Trial 4 | Hits :3 | target just moved to hide in :6
Trial 5 | Hits :3 | target still to hide in :6
Trial 6 | Hits :4 | target just moved to hide in :5
Trial 7 | Hits :4 | target still to hide in :5
Trial 8 | Hits :5 | shoot succeeded at -> 5
you hit the target in spot 5 in 8 trials
```

Figure 40 Task 7 | Output 3

```
Enter number of spots:7
Trial 1 | Hits :1 | target still to hide in :2
Trial 2 | Hits :2 | shoot succeeded at -> 2
you hit the target in spot 2 in 2 trials
```

Figure 41 Task 7 | Output 4

Comparison Between Algorithms

Another Algorithm Pseudocode:

```
class ShooterGame:
    array[]
    target
    shot
    trial
    random = new Random()

    function __init__():
        print "How many hiding spots are there?"
        n = read integer from user
        array = new array of size n
        target = random integer between 0 and n-1
        shoot()

    function step():
        if target == array.length-1:
            target -= 1
        else if target == 0:
            target += 1
        else:
            target += (random integer between 0 and 1) * 2 - 1

    function shoot():
        trial = 0
        for i from 1 to array.length-2:
            trial += 1
            shot = i
            if i == target:
                showGame()
                print "You hit the target!"
                return
            showGame()
            step()

        for j from array.length-2 to 1:
            trial += 1
            shot = j
            if j == target:
                showGame()
                print "You hit the target!"
                return
            showGame()
            step()

        print "You missed the target."

    function showGame():
        for position from 0 to array.length-1:
            if position == shot:
                print "X"
            else if position == target:
                print "O"
            else:
                print "_"
        print "Trial: " + trial
        print "Array length: " + array.length
        print "Target index: " + target
        print "Shot index: " + shot
```

Figure 42 Task 7 Another Algorithm Pseudocode

Description:

The "ShooterGame" class simulates a game in which the player tries to shoot a target hidden behind one of several possible positions. The algorithm of the game can be described as follows:

1. The user is prompted to enter the number of hiding spots.
2. The game initializes an array with the number of hiding spots entered and randomly selects a target position.
3. The "shoot()" method is called to start the game.
4. The "shoot()" method loops through the array twice, shooting at each position until the target is hit.
5. For each position, the game keeps track of the number of trials and the positions of the target and the last shot.
6. If the player hits the target, the game prints a message indicating that the target has been hit, and the game ends.
7. If the player doesn't hit the target after all shots have been taken, the game ends and prints a message indicating that the target was not hit.

Compare time complexity:

The time complexity of the "ShooterGame" algorithm is O(n), where n is the number of hiding spots. Algorithm that we applied also is O(n).

Conclusion:

In conclusion, the provided solution presents a greedy algorithm for hitting a moving target in a computer game. The algorithm guarantees hitting the target by employing a systematic approach. It iterates through each spot on the line and adjusts the target's hiding spot based on random step values. By continuously tracking and adapting to the target's movement, the algorithm ensures that the target is eventually hit.

The algorithm's time complexity is O(n), where n represents the number of spots on the line. This complexity arises from iterating through each spot and performing constant-time operations within the loop. The space complexity of the algorithm is O(1), indicating that it utilizes only a constant amount of memory regardless of the input size.

Overall, the provided algorithm serves as a viable approach for hitting a moving target in a computer game, prioritizing efficiency by minimizing the number of shots required to achieve success.

References

[1] Levitin, A., & Levitin, M. (2011). Algorithmic Puzzles. Oxford University Press.
[2] https://www.geeksforgeeks.org/dynamic-programming/
[3]https://www.geeksforgeeks.org/introduction-to-divide-and-conquer-algorithm-data-structure-and-algorithm-tutorials/
[4] https://www.geeksforgeeks.org/greedy-algorithms/
[5] Thomas H. Cormen, Charles E. Leiserson: Introduction to Algorithms.
[6] https://www.freecodecamp.org/news/brute-force-algorithms-explained/