
**Ain Shams University**

**Faculty of Engineering**

**Design and Analysis of Algorithms (CSE332s)**

**Project Report**

**Team Members:**

| | |
|---|---|
| Andrew Ayman Samir | 2000003 |
| Mohamed Yasser Mohamed | 2002085 |
| Ahmed Abdelrahman Ahmed | 2002126 |
| Ereny Hany Hanna | 2000202 |
| Omar Salah Mansour | 2002081 |
| Mina Hany Hanna | 2001480 |
| Martin Ashraf Ibrahim | 2000966 |
| Lara Maurice Youssef | 2001024 |
| Kyrillos Ashraf Gamil | 2002015 |

## Table of Contents:

# Task (1):

**Description:**

Given a $2^n x 2^n$ ($n > 1$) board with one missing square, tile it with the right dominoes of only three colors so that no pair of dominoes that share an edge have the same color and we will use dynamic programming to solve this problem.

**Assumption:**

- The missing square is chosen randomly anywhere on the board
- The main assumption used is from Golomb's inductive proof of a tormino theorem [1][2], where he stated that a tormino could be covered with 4 twice as small copies of itself which makes it a rep-4 tile (a rep-tile variety). See Fig. 1.1.

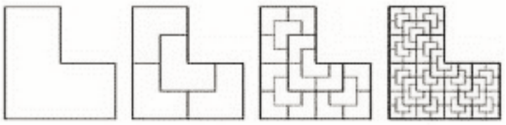


Fig. 1.1: A tormino as a rep-4 tile

- From the assumption above we could look at the $2^k x 2^k$board as containing one big piece of tormino and a smaller $2^{k-1} x 2^{k-1}$board with a 1x1 square removed. See Fig. 1.2.

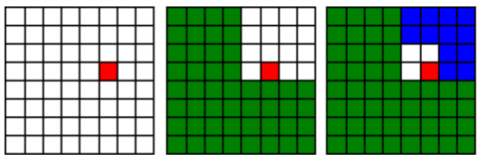
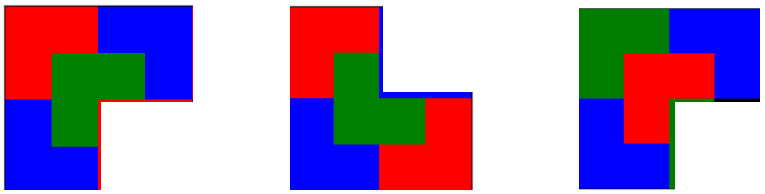


Fig. 1.2: A board can be presented at 1 large tormino and a square

- The dynamic programming memory will only store the results of torminoes of shape 4x4 or higher as there is no point in storing the configuration of a single tormino.
- when building a size 4 tormino (4x4) there is a predefined color assignment with a fixed rotation to help build the full board these assignments are:

- If the edge has the color red, the closest to it gets the color orange the other 2 at the sides get the color blue and the furthest gets red. See 1 in Fig. 1.3.
- If the edge has the color blue, the closest to it gets the color orange the other 2 at the sides get the color red and the furthest gets blue. See 2 in Fig. 1.3.
- If the edge has the color orange, the closest to it gets red, the other 2 at the sides get blue, and the furthest gets orange. See 3 in Fig. 1.3.



(1) (2) (3)

Fig. 1.3

- The first tormino that will be placed has a blue color.

**Detailed Solution:**

1) Tiling function:
    a) Once the function is called it goes through the board from a specific first coordinated( first call the first coordinated is 0,0) and tries to find the location of the missing square.
    b) Once the coordinates of the missing square are found, we can reduce the board into a large tormino and a large square with a missing 1x1 square, the square is the overlapping simpler subproblem that we will solve, then the tiling function will be called recursively till we reach a base case of size 1 in which we return.
    c) We will find the edge which is the closest color at the center to the tormino to be placed.
    d) Once we have the simplest case which is a 2x2 problem and the edge color is black(missing), we place a blue tormino in the remaining spots and return. See Fig. 1.4

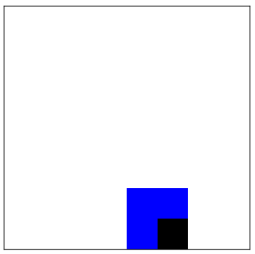


Fig. 1.4: First tormino placed

e) else we will call the placeTormino function.

2) placeTormino function:

a) We try to find the edge and it location.

b) We look if an answer is present in the dp memory with this size and edge color.

i) If an answer is present, we get the answer and get the orientation of the answer and compare it with the orientation of the tormino to be placed.

ii) If it has the same orientation place the tormino from memory into the board.

iii) If it has a different orientation, we call the rotate and place function which rotates the answer till it has the same orientation, and then place it on the board.

c) If no answer is present the size is 2x2 get the color to be placed either predefined for 4x4 or derived for bigger problems place the tormino store the answer and return.

d) If it is bigger than 2x2 which is 4x4 or bigger we define the colors and place 4 torminos according to the location and color of the edge like Fig. 1.3 by recursively calling the placeTormino function.

**Pseudo-code:**

```
FUNCTION findMissing(arr[0..2^n-1][0..2^n-1], size, firstX, firstY)
        //Finds the coordinates of the missing square
        //Input: The board, the size of search, first x and y coordinated
        //Output: Array missing[2] containing the coordinates of the missing square
        FOR i <- 0 TO size-1 DO
                FOR j <- 0 TO size-1 DO
```

```
                    IF arr[firstX+i][firstY+j] == -1
                        return missing <- The coordinates
                    ENDIF
                ENDFOR
            ENDFOR
            //If not found
            return missing <- -1, -1
ENDFUNCTION

FUNCTION findEdge(arr[0..2^n-1][0..2^n-1], size, firstX, firstY, wide=false)
            //Finds the colour and location of the edge, an edge is the adjacent block to
            //the center of the tormino
            //Inputs: The board, the size of search, first x and y locations, a wide boolean
            //to search in a larger scale than 2x2
            //Output: Array edge[3] where the first element represent the color and the
            //rest represent the location
            IF size>2 ANDnot wide DO
                //search in the middle 2x2
                return findEdge(arr, size/2, firstX+size/4, firstY+size/4)
            ELSE IF size > 4 DO
                //search the middle 4x4
                return findEdge(arr,size/2, firstX+size/4, firstY+size/4)
            ENDIF
            FOR the entire size DO
                IF element  is not 0 AND is not-1
                    edge <- [color of the element, x of the element, y of the
                                element]
                    return edge
                ENDIF
            ENDFOR
            return edge <-[-1, null, null]
ENDFUNCTION
```

```
FUNCTION findOrentaion(arr[0..2^n-1][0..2^n-1], size)
        //Given the answer look for the edge to determine the orientation
        //Input: answer as a 2d array, size of the answer
        //Output: Array edge[3] containing color, and position of the edge
        FOR the entire size of answer SEARCH
                IF element==100 //A special number for edge of answer
                        return edge <- [element color(100), x position, y position]
                ENDIF
        ENDFOR
        return edge <- [-1, null, null]
ENDFUNCTION
FUNCTION rotate90Clockwise(arr[0..2^n-1][0..2^n-1], n)
        //Rotates the answer 90 degrees clockwise
        //Input: Answer as a 2d array, size n of the array
        //Output: Rotated Answer as a 2d array
        FOR the entire answer size DO
                rotated <- rotated coordinates of the answer
        ENDFOR
        return rotated
ENDFUNCTION
FUNCTION rotateAndPlace(arr[0..2^n-1][0..2^n-1], size, firstX, firstY, edge[0..2],
                                savedDP[0..size], dpEdge[0..2])
        temp <- copy of savedDP
        WHILE (x-firstX), (y-firstY) of edge != x, y of dpEdge
                temp <- rotate90Clockwize(temp, size)
                dpEdge <- findOrentation(temp, size)
        ENDWHILE
        apply temp in the specified location on the board
ENDFUNCTION
FUNCTION placeTormino(arr[0..2^n-1][0..2^n-1], size, firstX, firstY, originEdge,
                                dp[0..N][3][0..a][0..a], lastTormino= false, color=-1)
        //Places the tormino after the first tormino is placed
```

```
//Inputs: The board, size of the subproblem, first x and y , orginEdge the edge
//of the previous subproblem, dp is the dynamic Progamming memory, last
//Tormino a bool that indicated that it is the edge tormino and last tormino to
//be placed, color is the color of the next tormino to be placed
edge <- findEdge(of the subproblem)
IF Answer is present ans size > 2
        savedDP <- get the answer
        dpEdge <- findOrentation of the answer
        IF the orientation is same as the subproblem
                place the answer
        ELSE
                rotateAndPlace(the answer)
        ENDIF
ELSE
        IF size == 2
                IF edge is black
                        findEdge(..., wide=true)
                newColor <- color
                IF lastTormino
                        newColor <- color of the origin edge
                ENDIF
                place tormino
                store the answer in dp
ENDIF
IF edge[0] = red
        color1= orange
        color2=blue
ELSE IF edge[0] = blue
        color1 = orange
        color2=red
ELSE IF edge[0] = orange
        color1=red
```

```
            color2=blue
        ENDIF
        //place the first tormino is the edge of the subploblem
        placeTormino(arr, size/2, firstX+size/4, firstY+size/4, edge, dp, false, color1)
        based on the quadrant of the edge place  place the rest three in the corrent
            quadrant
            placeTormino(..., color=color2)
            placeTormino(..., color=color20
            placecTormino(..., lastTormino=true)
        ENDIF
ENDFUNCTION
FUNTION tile(arr[0..2^n-1][0..2^n-1], size, firstX, firstY, dp)
        //The starting point of the tiling algorithm, gets the subproblem and places
        //the first tormino is the subproblem.
        //Input: The board, the size of the subproblem, firstX and Y of the
        //subproblem, The dynamic programming memory array.

        //Base Case
        IF size=1 return
        missing <- findMissing()
        IF there is missing square
            tile(The quadrant of in which the missing square)
        ENDIF
        edge <- findEdge()
        IF size=2
            place a tormino in all except missing square
            return
        ENDIF
        placeTormino()
ENDFUNCTION
```

**Complexity Analysis of the Algorithm:**

1. Finding Missing Squares:

As we search for the missing square in the entire board, the complexity will be $2^n.2^n$ or $O(2^{2n})$.

2. Finding edge color

   As we try to find the edge for a 2x2 or 4x4 and decrease the subproblem by half each time the time complexity will be $O(\log 2^n)$ or $O(n)$

3. Rotating and placing torminos

   The rotation operation is $O(2^{2n})$ since it iterates over the entire tormino grid. However, the overall time complexity of this function depends on the number of rotations required to achieve the correct orientation. In the worst case, it might rotate the tormino several times, which can also be approximated as $O(\log 2^n)$or $O(n)$.

4. Placing torminos

   Same as the finding the edge, due to its recursion of the smaller subproblems, its complexity can be approximately $O(n)$

5. Tiling the grid

   The operation to tile the grid depends on the number of iterations or level of recursion because it contains finding the missing square and placing torminos the time complexity will be $O(n.2^{2n})$

So the overall time of the algorithm is **$O(n.2^{2n})$**

**Sample Output:**

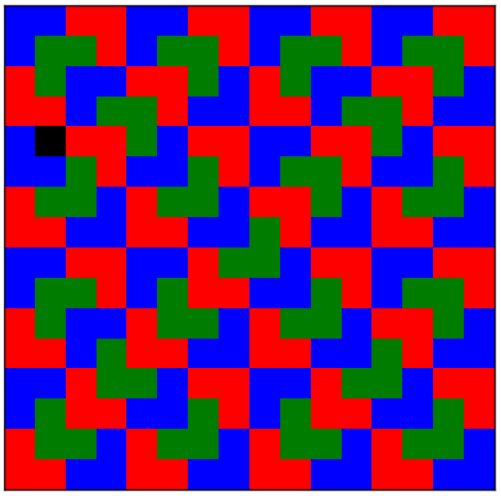


Fig 1.5: $2^4x2^4$

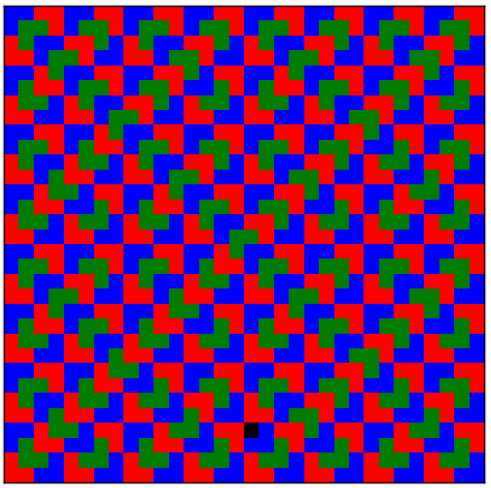


Fig 1.6: $2^5x2^5$

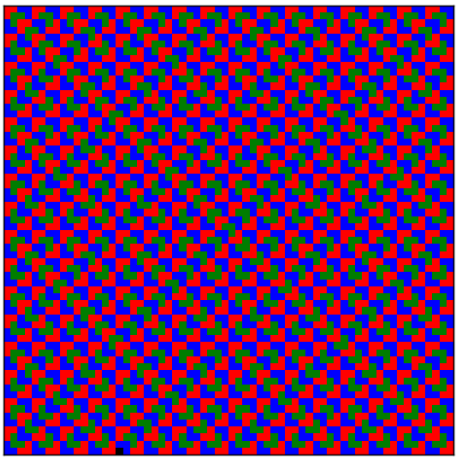


Fig 1.7: $2^6x2^6$

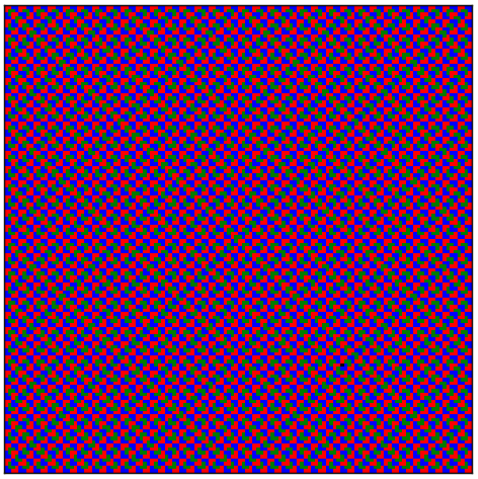


Fig 1.8: $2^7x2^7$

**Code:**

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>

#define N 7

using namespace std;

//Function to find the missing square
int* findMissing(int** arr, int size, int firstX, int firstY){
    int* missing = new int[2];
    for(int i = 0; i < size; i++){
```

```
        for(int j = 0; j < size; j++){
            if(arr[firstX+i][firstY+j] == -1){
                missing[0] = firstX+i;
                missing[1] = firstY+j;
                return missing;
            }
        }
    }
    missing[0] = -1;
    missing[1] = -1;
    return missing;
}
int* findEdge(int** arr, int size, int firstX, int firstY, bool
wide=false) {
    /*
        This function will return the edge color and location(to
know the missing quadrant)
        The wide parameter is a flag to look in a wider range
    */
   if (size > 2 && !wide)
        return findEdge(arr, size/2, firstX+size/4,
firstY+size/4);
    else if (size > 4){
        return findEdge(arr, size/2, firstX+size/4,
firstY+size/4, true);
    }
    int* edge = new int[3];
    edge[0] = -1;
    for(int i = 0; i < size; i++){
        for(int j = 0; j < size; j++){
            if(arr[firstX+i][firstY+j] != 0 &&
arr[firstX+i][firstY+j] != -1){
                int edgeColor = arr[firstX+i][firstY+j];
                edge[0] = edgeColor;
                edge[1] = firstX + i;
                edge[2] = firstY + j;
                return edge;
            }
        }
    }
    return edge;
}
int* findOrentation(int** arr, int size){
```

```
    int* edge = new int[3];
    edge[0] = -1;
    for(int i = size/2 - 1; i < size/2 + 1; i++){
        for(int j = size/2-1; j < size/2 + 1; j++){
            if(arr[i][j] == 100){
                int edgeColor = arr[i][j];
                edge[0] = edgeColor;
                edge[1] = i;
                edge[2] = j;
                return edge;
            }
        }
    }
    return edge;
}

int** rotate90Clockwise(int** arr, int n){
    int** rotated = new int*[n];
    for (int i = 0; i < n; i++) {
        rotated[i] = new int[n];
    }

    // Fill the new array with the rotated values
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            rotated[j][n - i - 1] = arr[i][j];
        }
    }

    // Return the new array
    return rotated;
}
void rotateAndPlace(int** arr, int size, int firstX, int firstY,
int* edge, int** savedDP, int* dpEdge){
    //We need to copy in the savedDP array
    int** temp = new int*[size];
    for(int i = 0; i < size; i++){
        temp[i] = new int[size];
    }
    for(int i = 0; i < size; i++){
        for(int j = 0; j < size; j++){
            temp[i][j] = savedDP[i][j];
        }
```

```
    }
    // This function will rotate the tormino and place it in the
correct orentation
    while(!(dpEdge[1] == edge[1]-firstX && dpEdge[2] ==
edge[2]-firstY)){
        temp = rotate90Clockwise(temp, size);
        dpEdge = findOrentation(temp, size);
    }
    // Place the tormino
    for(int i = 0; i < size; i++){
        for(int j = 0; j < size; j++){
            if(arr[firstX+i][firstY+j] == 0){
                arr[firstX+i][firstY+j] = temp[i][j];
            }
        }
    }
}
void placeTormino(int** arr, int size, int firstX, int firstY,
int* originEdge, int**** dp, bool lastTormino = false, int
color=-1){
    // lastTormino is a flag to indicate that this is the last
tormino to be placed and is used only in the 2x2 case
    //We need to find the edge color
    int* edge = findEdge(arr, size, firstX, firstY);
    // According to the edge color we will look for an answer in
the dp memory or we will find one
    if(dp[int(log2(size))-1][edge[0]-1] != nullptr && size>2){
        // Place the saved tormino
        //We need to check if it is in the right orentation
        int** savedDP = dp[int(log2(size))-1][edge[0]-1];
        // printFullGrid(arr);
        int* dpEdge = findOrentation(savedDP, size);
        if (dpEdge[1] == edge[1]-firstX && dpEdge[2] ==
edge[2]-firstY){
            // Correct orentation
            for(int i = 0; i < size; i++){
                for(int j = 0; j < size; j++){
                    if(arr[firstX+i][firstY+j] == 0){
                        arr[firstX+i][firstY+j] = savedDP[i][j];
                    }
                }
            }
        }
```

```
        else{
            // We need to rotate and place tormino into correct
orentaion
            rotateAndPlace(arr, size, firstX, firstY, edge,
savedDP, dpEdge);
        }
    }
    else {
        if(size == 2){
            //If edge is black try to find in a wider range
            if (edge[0] == -1)
                edge = findEdge(arr, size, firstX, firstY, true);
            int newColor = color;
            //If it is a last tormino place the same color as the
original edge
            if (lastTormino)
                newColor = originEdge[0];
            // If the size is 2 we will place the tormino
            for(int i = 0; i < size; i++){
                for(int j = 0; j < size; j++){
                    if(arr[firstX+i][firstY+j] == 0){
                        arr[firstX+i][firstY+j] = newColor;
                    }
                }
            }
            if(lastTormino){
                return;
            }
            //Store the answer
            int** dpNew = new int*[size];
            for(int i = 0; i < size; i++){
                dpNew[i] = new int[size];
            }
            for(int i = 0; i < size; i++){
                for(int j = 0; j < size; j++){
                    if(arr[firstX+i][firstY+j] == newColor){
                        dpNew[i][j] = arr[firstX+i][firstY+j];
                    } else{
                        dpNew[i][j] = 100;
                    }
                }
            }
            dp[int(log2(size))-1][edge[0]-1] = dpNew;
```

```
            return;
        }
        // Find tormino
        // A large tormino can be made up of 4 smaller torminos
        // We will start to put the tormino that is in the middle
        int colour1, colour2;

        if (edge[0]==1){
            colour1 = 3;
            colour2 = 2;
        }
        else if (edge[0]==2){
            colour1 = 3;
            colour2 = 1;
        }
        else if(edge[0]==3){
            colour1 = 1;
            colour2 = 2;
        }
        placeTormino(arr, size/2, firstX+size/4, firstY+size/4,
edge, dp, false,colour1);
        if(edge[1] >= size/2+firstX){
            if(edge[2] >= size/2+firstY){
                placeTormino(arr, size/2, firstX, firstY+size/2,
edge, dp, false,color=colour2);
                placeTormino(arr, size/2, firstX+size/2, firstY,
edge, dp, false,color=colour2);
                placeTormino(arr, size/2, firstX, firstY, edge,
dp, true);
            } else {
                placeTormino(arr, size/2, firstX, firstY, edge,
dp, false,color=colour2);
                placeTormino(arr, size/2, firstX+size/2,
firstY+size/2, edge, dp, false,color=colour2);
                placeTormino(arr, size/2, firstX, firstY+size/2,
edge, dp, true);
            }
        } else {
            if(edge[2] >= size/2+firstY){
                placeTormino(arr, size/2, firstX, firstY, edge,
dp, false,color=colour2);
                placeTormino(arr, size/2, firstX+size/2,
firstY+size/2, edge, dp, false,color=colour2);
```

```
                placeTormino(arr, size/2, firstX+size/2, firstY,
edge, dp, true);
            } else {
                placeTormino(arr, size/2, firstX+size/2, firstY,
edge, dp, false, color=colour2);
                placeTormino(arr, size/2, firstX, firstY+size/2,
edge, dp, false, color=colour2);
                placeTormino(arr, size/2, firstX+size/2,
firstY+size/2, edge, dp, true);
            }
        }

    }

}

void tile(int** arr, int size, int firstX, int firstY,
int****dp){

    if(size == 1){
        return;
    }
    // We need to find the missing square
    int* missing = findMissing(arr, size, firstX, firstY);

    // If there is missing square
    if(missing[0] > -1){
    //The grid can be broken down into a tormino and a square
with the missing square
        if(missing[0]-firstX < size/2){
            if(missing[1]-firstY < size/2){
                tile(arr, size/2, firstX, firstY, dp);
            } else {
                tile(arr, size/2, firstX, firstY+size/2, dp);
            }
        } else {
            if(missing[1]-firstY < size/2){
                tile(arr, size/2, firstX+size/2, firstY, dp);
            } else {
                tile(arr, size/2, firstX+size/2, firstY+size/2,
dp);
            }
        }
```

```
    }
    //tile the tormino
    // We need to find the edgeColor
    int* edge = findEdge(arr, size, firstX, firstY);
    int edgeColor = edge[0];
    //If the edge color is -1 which means this is the missing
square and in this case we will put an answer
    if(size == 2){
        for(int i = 0; i < size; i++){
            for(int j = 0; j < size; j++){
                if(arr[firstX+i][firstY+j] == 0){
                    //We will put the tormino next to the missing
square to be blue
                    arr[firstX+i][firstY+j] = 2;
                }
            }
        }
        return;
    }
   placeTormino(arr, size, firstX, firstY, edge, dp);
}

int main(){
    srand(time(nullptr));
    int x = pow(2, N);
    int y = pow(2, N);

    int randX = rand() % x;
    int randY = rand() % y;

    int** arr = new int*[x];
    for(int i = 0; i < x; i++){
        arr[i] = new int[x];
    }

    //Initalize the array with zeros
    for(int i = 0; i < x; i++){
        for(int j = 0; j < x; j++){
            arr[i][j] = 0;
        }
    }
    arr[randX][randY] = -1;
```

```
    // We will create a dynamic programming memory to store the
torminos solution
    int**** dp = new int***[N];
    for(int i = 0; i < N; i++){
        // In each size we will store 3 possible solutions for
each color
        dp[i] = new int**[3];
    }


    for(int i = 0; i < N; i++){
        for(int j = 0; j < 3; j++){
            // We will initialize the dp memory with -1 to
indicate that we have not found a solution yet
            dp[i][j] = nullptr;
        }
    }

    tile(arr, x, 0, 0, dp);
    for(int i = 0; i < x; i++){
        for(int j = 0; j < x; j++){
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

**Other Techniques:**

1. Divide And Conquer

   This is the widely used approach to solve this problem. The idea is to divide the board into 4 separate subproblems and to solve each problem separately to reach an overall solution. The subproblem is then divided into smaller subproblems till a solvable size

Implementation[3]:

- 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.
- If n > 2, divide the board into four $2^{n-1} \times 2^{n-1}$ boards and place one gray tromino to cover the three central squares that are not in the $2^{n-1} \times 2^{n-1}$ board with the missing square.
- Then tile each of the three $2^{n-1} \times 2^{n-1}$ boards recursively by the same algorithm.

2. Brute Force

   This is a naive solution to solve this problem as it tries all the possible tormino orientations and all the possible colors for the entire grid which will take a very long time to get an answer

**Table1.1: Comparison between 3 algorithms for Task 1**

| **Dynamic Programming** | **Divide And Conquer** | **Brute Force** |
|---|---|---|
| - Divides into overlapping subproblems<br>- Can build bigger problems for solutions of the smaller subproblems<br>- Has high space complexity<br>- But better time complexity than Divide and Conquer due to the avoidance of computing redundant computations | - Divide into separate subproblems<br>- Does not utilize the previous solutions<br>- Has a better space complexity than Dynamic Programming<br>- Has somewhat worse time complexity than Dynamic Programming as it computes redundant computations | - Tries all possible combinations<br>- Has a very high time complexity<br>- It can be a little better by using backtracking but it is still bad |

**Conclusion:**

Although dynamic programming to solve this problem seems somewhat complex but it has the best time complexity as it can store past configuration answers and can reuse it to solve the bigger subproblem at the cost of space complexity. Divide and

Conquer although has a better space complexity because it solves all subproblems independently, has worse time complexity than Dynamic Programming. Brute Force is the simplest of them all but has a huge time complexity and may take years to solve a reasonable-size problem

Note: In the code submitted there are 2 files one for the algorithm called Task_1.cpp, and the other is a Python file for visualizing the algorithm caller Task_1_gui.py but to run it it has to be in the same directory as the cpp file. And to change the size problem just change the N variable in the cpp file.

**References:**


[1] https://www.cut-the-knot.org/Curriculum/Geometry/Tromino.shtml

[2]https://www.researchgate.net/publication/274569582_Per-Tiling_Rep-Tiling_and_Penrose_Tiling_A_notion_to_edge_cordial_and_cordial_labeling

[3]Algorithmic Puzzles by Anany Levitin, Maria Levitin

[4]https://www.geeksforgeeks.org/tiling-problem-using-divide-and-conquer-algorithm/

# Task (2):

**Description:**

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.

**Assumption:**

- The Chessboard is a standard 8x8 grid.
- The knight can only move according to the standard knight move, which is L-Shaped move consisting of 2 steps: first step horizontal or vertical direction, followed by a step perpendicular to it.
- The knight starts moving from a random cell, and must visit all cells.
- The knight can visit each cell exactly once. The goal is to find a closed tour in all cells, which ends in a cell one knight's move away from a starting cell.

**Detailed Solution:**

**Greedy Algorithm:**

1) Initialize the chessboard as a 1D array of size N*N with -1 values.
2) Choose a starting position on the chessboard.
3) Mark the starting position as visited (1) in the solution array.
4) Implementation of greedy algorithm:
    a. For each possible move from the current position (x, y), calculate the accessibility degree, which is the number of unvisited squares that can be reached from the adjacent square.
    b. Choose the adjacent square with the minimum accessibility degree as the next move. If multiple squares have the same degree, choose one randomly.
    c. Update the current position (x, y) to the chosen square and mark it as visited

5) Repeat step 4 until all squares on the chessboard are visited or there are no valid moves.
6) Check if the knight's tour is closed by verifying if the last position is one knight's move away from the starting position. If not, return false.
7) Print the chessboard to display the knight's tour if it is closed. In the main function, repeatedly execute steps 3-7 until a closed knight's tour is found.

**Greedy Pseudo-code:**

```
Define N
x_moves[8] <--- {1,1,2,2,-1,-1,-2,-2}
y_moves[8] <--- {2,-2,1,-1,2,-2,1,-1}

Function is_Empty(a[], x, y):
return (x ≥ 0 AND y ≥ 0) AND (x < N AND y < N) AND (a[y * N + x] < 0)


Function getDegree(a[], x, y):
    count ← 0 for i ← 0 to N-1:
        if is_Empty(a, x + x_Moves[i], y + y_Moves[i]):
            count ← count + 1
    return count

Function nextMove(sol[], x, y):
    min_degree_index ← -1
    min_degree ← N + 1
    next_x ← 0
    next_y ← 0

start ← random() mod N
for count ← 0 to N-1:
```

```
    i ← (start + count) mod 8
    next_x ← x + x_Moves[i]
    next_y ← y + y_Moves[i]
    if is_Empty(sol, next_x, next_y) AND getDegree(sol, next_x, next_y) <
min_degree:
        min_degree_index ← i
        min_degree ← getDegree(sol, next_x, next_y)

if min_degree_index = -1:
    return false

next_x ← x + x_Moves[min_degree_index]
next_y ← y + y_Moves[min_degree_index]
sol[next_y * N + next_x] ← sol[y * N + x] + 1
x ← next_x
y ← next_y
return true

Function print(a[]):
    for i ← 0 to N-1:
        for j ← 0 to N-1:
            print a[j * N + i]
        print newline

Function neighbour(x, y, start_x, start_y):
    for i ← 0 to 7:
            if (x + x_Moves[i] = start_x) AND (y + y_Moves[i] = start_y):
                return true
    return false
```

```
Function findClosedKnightTour(start_x, start_y):
    sol[N * N] ⟵ -1
    for i ⟵ 0 to N * N - 1:
        sol[i] ⟵ -1
    x ⟵ start_x
    y ⟵ start_y
    sol[y * N + x] ⟵ 1

    for i ⟵ 0 to N * N - 2:
        if nextMove(sol, x, y) = false:
            return false

    if neighbour(x, y, start_x, start_y) = false:
        return false

    print(sol)
    return true

Function main():
    srand(current_time)
    start_x ⟵ 0
    start_y ⟵ 0
    print "Enter your start position x: "
    read start_x
    print "Enter your start position y: "
    read start_y

    while findClosedKnightTour(start_x, start_y) = false:
    // continue looping until a closed knight's tour is found
```

return 0

**Code:**

```
#include <iostream>
using namespace std;
#define N 8

// moves patterns of knight in chess

static int x_Moves[8] = { 1,1,2,2,-1,-1,-2,-2 };
static int y_Moves[8] = { 2,-2,1,-1,2,-2,1,-1 };




// checks whether a square is within the 8 x 8 chessboard  and
empty or not
bool is_Empty(int a[], int x, int y)
{
    return ((x >= 0 && y >= 0) && (x < N&& y < N)) && (a[y * N +
x] < 0);
}

// returns the number of empty squares adjacent to x and y

int getDegree(int a[], int x, int y)
{
    int count = 0;
    for (int i = 0; i < 8; ++i)
        if (is_Empty(a, (x + x_Moves[i]), (y + y_Moves[i])))
            count++;

    return count;
}

// picks next point using minimum degree approach
// returns false if it is not possible to pick

bool nextMove(int sol[], int* x, int* y)
{
```

```
    int min_degree_index = -1, c, min_degree = (8 + 1), next_x,
next_y;


    // from a random adjacent. Find the adjacent with minimum
degree.

    int start = rand() % N;
    for (int count = 0; count < N; ++count)
    {
        int i = (start + count) % 8;
        next_x = *x + x_Moves[i];
        next_y = *y + y_Moves[i];
        if ((is_Empty(sol, next_x, next_y)) &&
            (c = getDegree(sol, next_x, next_y)) < min_degree)
        {
            min_degree_index = i;
            min_degree = c;
        }
    }

    // if we didn't find cells
    if (min_degree_index == -1)
        return false;

    // store coordinates of next point
    next_x = *x + x_Moves[min_degree_index];
    next_y = *y + y_Moves[min_degree_index];

    // mark next move
    sol[next_y * N + next_x] = sol[(*y) * N + (*x)] + 1;

    // update next point
    *x = next_x;
    *y = next_y;

    return true;
}

//printing the solution board with visited cells and its visit
number
void print(int a[])
{
```

```
    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
            printf("%d\t", a[j * N + i]);
        printf("\n");
    }
}

// checks if the last cell visisted is 1 knight move away from
starting cell
bool neighbour(int x, int y, int start_x, int start_y)
{
    for (int i = 0; i < 8; ++i)
        if (((x + x_Moves[i]) == start_x) && ((y + y_Moves[i]) ==
start_y))
            return true;

    return false;
}



bool findClosedKnightTour(int start_x, int start_y)
{
    // initialzing the chessboard with -1 as it marks the non
visited cells
    int sol[N * N];
    for (int i = 0; i < N * N; ++i)
        sol[i] = -1;
    int x = start_x, y = start_y;
    sol[y * N + x] = 1; // Marking our starting point with 1 as
its visited

    // keep picking next points using our greedy approach , which
is picking next minimum degree cell to visit

    for (int i = 0; i < N * N - 1; ++i)
        if (nextMove(sol, &x, &y) == 0) {
            //print(sol);

            return false;
        }
    // checking if the knight tour is closed
```

```
    if (!neighbour(x, y, start_x, start_y))
        return false;

    print(sol);
    return true;
}


int main()
{
    int start_x;
    int start_y;
    cout << " Enter your start position x : " << '\n';
    cin >> start_x;
    cout << " Enter your start position y : " << '\n';
    cin >> start_y;

    while (!findClosedKnightTour(start_x, start_y))
    {
        // while no closed tour knight keep
        ;
    }

    return 0;
}
```

**Complexity:**

O(N^2) where N x N is the size of chessboard.

## Another Technique:

**Backtracking Pseudocode:**

Define N

x_moves[8] <--- {1,1,2,2,-1,-1,-2,-2}

y_moves[8] <--- {2,-2,1,-1,2,-2,1,-1}

```
Function isValid(x,y,board):
        return (x>=0 AND y>=0) AND (x<N AND y<N) AND (board[x][y]=-1)

Function printKnightTour(board):
        for i <-- 0 to N-1:
                for j <-- 0 to N-1:
                        print board[i][j]
                print newline

Function neighbour(x,y,start_x,start_y):
        For i <--0 to 7:
                If(x+x_Moves[i] = start_x) AND (y+y_Moves[i] = start_y):
                        Return true
        Return false

Function backtrackUntilClosedTour(board,x,y,moveCount,start_x,start_y):
        If moveCount==(N*N)+1:
                If neighbour(x,y,start_x,start_y):
                        Return true
                For i <-- 0 to 7:
                        next_x = x + x_Moves[i]
                        next_y = y + y_Moves[i]
                        If isValid(next_x,next_y,board):
                                board[next_x][next_y] = moveCount
```

```
                    backtrackUntilClosedTour(board,x,y,moveCount,start_x,
                    start_y):
                        Return true
                    board[next_x][next_y] = -1
Return false


Function findKnightTour(board,start_x,start_y):
    board[start_x][start_y]=1
    Return backtrackUntilClosedTour(board,start_x,start_y,2,start_x,start_y)


Function main():
    Start_x = 0
    Start_y = 0
    Print “Enter starting position X: ”
    Read start_x
    Print “Enter starting position Y: ”
    Read start_y

    Board <-- create 2D array of size N x N, initialized with -1
    If findKnightTour(board, start_x, start_y):
        Print “Knight’s tour is found”
        printKnightTour(board)
    Else:
        Print “No closed knight’s tour found”
```

**Comparison with other Technique:**

Used Backtracking technique as another solution.

**Table 2.1: Comparison Table for the Greedy Technique and Backtracking**

| | Greedy Technique | Backtracking Technique |
|---|---|---|
| **Usage** | - Works by selecting the next move based on the minimum degree heuristic, it chooses the cell with the fewest available moves.<br><br>- It efficiently calculates the degree of each potential move and selects the one with minimum degree.<br><br>- It provides a function to print the solution board with the visited cells and their visit number.<br><br>- The code keeps track of the visited cells using integer array , making it easy to check if the tour is finished. | - It tries to explore all possible moves until a closed tour is found or all possible ways are exhausted.<br><br>- It efficiently checks the validity of each move and backtracks when necessary.<br><br>-The algorithm uses 2D vector to represent the chessboard, making it easy to track the moves in an ordered way. |

In Conclusion, both algorithms aim to find a closed knight's tour on an 8x8 chessboard, but they use different approaches. The Greedy algorithm works with with a minimum degree heuristic, prioritizes moves with the fewest available options, while the Backtracking algorithm systematically explores all possibilities and backtracks when necessary.

**Output Sample with Description:**

Greedy Algorithm:

```
PS C:\Users\win10> cd "c:\Users\win10\Desktop\" ; if ($?) { g++ test.cpp -o test } ; if ($?) { .\test }
 Enter your start position x :
0
 Enter your start position y :
0
1       16      51      34      3       18      21      42
36      33      2       17      52      41      4       19
15      64      35      50      39      20      43      22
32      37      48      57      44      53      40      5
61      14      63      38      49      56      23      54
28      31      60      47      58      45      6       9
13      62      29      26      11      8       55      24
30      27      12      59      46      25      10      7
PS C:\Users\win10\Desktop>
```

```
PS C:\Users\win10\Desktop> cd "c:\Users\win10\Desktop\" ; if ($?) { g++ test.cpp -o test } ; if ($?) { .\test }
 Enter your start position x :
4
 Enter your start position y :
4
9       12      27      58      63      14      29      56
26      51      10      13      28      57      62      15
11      8       45      50      59      64      55      30
36      25      52      43      54      49      16      61
7       44      35      46      1       60      31      48
24      37      22      53      42      47      2       17
21      6       39      34      19      4       41      32
38      23      20      5       40      33      18      3
PS C:\Users\win10\Desktop>
```

```
PS C:\Users\win10\Desktop> cd "c:\Users\win10\Desktop\" ; if ($?) { g++ test.cpp -o test } ; if ($?) { .\test }
 Enter your start position x :
2
 Enter your start position y :
6
23      52      7       48      21      50      5       2
8       47      22      51      6       3       20      41
53      24      63      38      49      42      1       4
46      9       54      43      64      37      40      19
25      62      45      58      39      34      15      36
10      55      28      61      44      59      18      33
29      26      57      12      31      16      35      14
56      11      30      27      60      13      32      17
PS C:\Users\win10\Desktop>
```

## Input/Output Description:

The user enters the X and Y indices from where the Knight would start the tour, then the program simulates the tour with minimum number of moves, every cell is visited only once, and ending in a cell one knight's move away from the starting cell.

The numbers printed show the path, as step k is followed by step k+1.

**Conclusion:**

Both the greedy and backtracking algorithms offer different trade-offs. The greedy algorithm provides a quick solution with good efficiency but does not guarantee a closed tour in all cases. On the other hand, the backtracking algorithm ensures a complete search but may have higher time complexity. The choice of algorithm depends on the specific requirements, constraints, and desired trade-offs for solving the knight's tour problem.

**References:**

Greedy: https://www.geeksforgeeks.org/warnsdorffs-algorithm-knights-tour-problem/

Backtracking: https://www.geeksforgeeks.org/the-knights-tour-problem/

# Task (3)

**Description:**

There is a row of n security switches protecting a military installation entrance. The switches can be manipulated as follows:

1. The rightmost switch may be turned on or off at will.
2. 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.
3. Only one switch may be toggled at a time.

Design a Dynamic Programing 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.

**Assumption:**

We will number the switches left to right from 1 to n and denote the “on” and “off” states of a switch by a 1 and 0, respectively.

Consider now the instance of the puzzle initialized by representing by the bit string of n 0’s: 000. . .0. Before we can start turning on.

**Detailed Solution:**

**Dynamic Programming Algorithm:**

1) It resembles the solution of the fibonacci series using dynamic programming.

2) We discovered the pattern through which the numbers appear.

3) n = 1 > 1 , n = 2 > 2 , n=3 > 5 , n=4 > 10

4) pattern: $M(n) = M(n - 1) + 2M(n - 2) + 1$

5) We can even simplify it more and get the recurrence relation in just n which resulted in For even n’s: $= (2n+1 - 2)/3$; for odd n’s: $(2n+1 - 1)/3$.

| n=1 | n=2 | n=3 | n=4 |
|---|---|---|---|
| 1 | 1 1 | 1 1 1 | 1 1 1 1 |
| 0 | 0 1 | 1 1 0 | 1 1 0 1 |
| | 0 0 | 0 1 0 | 1 1 0 0 |

| | | | |
|---|---|---|---|
| | | 0 1 1 | 0 1 0 0 |
| | | 0 0 1 | 0 1 0 1 |
| | | 0 0 0 | 0 1 1 1 |
| | | | 0 1 1 0 |
| | | | 0 0 1 0 |
| | | | 0 0 1 1 |
| | | | 0 0 0 1 |
| | | | 0 0 0 0 |

Fig. 3.1: Results with different problem sized

**Dynamic Programming code:**

```
    // efficiently calculate number of steps
    public static long getMinMovesHelper(long[] dp, int n) {
      long res = -1;
      if (dp[n] != -1) return dp[n];
      res = getMinMovesHelper(dp, n - 1) + 2 *
getMinMovesHelper(dp, n - 2) + 1;
      dp[n] = res;
      return res;
    }
    // initialization to some important variables during
getMinMoves
    public static long getMinMoves(int n) {
      long[] dp = new long[n + 1];
      Arrays.fill(dp, -1);
      dp[0] = 0;
      if (n >= 1) {
        dp[1] = 1;
      }
      if (n >= 2) {
        dp[2] = 2;
      }
      return getMinMovesHelper(dp, n);
    }
```

**Complexity:**

Time: O(N) where N number of switches.

Space: O(N) where N number of switches (ignore function stack).

**Constant time code:**

```
public static long getMinMovesBase(int n) {
    if (n % 2 == 0) {
      return ((long) Math.pow(2, n + 1) - 2) / 3;
    }else{
      return ((long) Math.pow(2, n + 1) - 1) / 3;
    }
  }
```

**Complexity:**

Time: O(1) assuming power function is constant time else the function complexity will correlate to it.

Space: O(1).

**Another Technique:**

**Detailed Solution:**

we can also use the following observation: when we number the indexes in the array from *right* to left, starting with 0 for the rightmost entry, then the index at which a digit should toggle follows this sequence:

0 1 0 2 0 1 0 3 0 1 0 2 0…

| output numbering | in binary | index of rightmost 1 | Gray code |
|---|---|---|---|
| 1 | 00001 | 0 | 00001 |
| 2 | 00010 | 1 | 00011 |
| 3 | 00011 | 0 | 00010 |
| 4 | 00100 | 2 | 00110 |

| | | | |
|---|---|---|---|
| 5 | 00101 | 0 | 00111 |
| 6 | 00110 | 1 | 00101 |
| 7 | 00111 | 0 | 00100 |
| 8 | 01000 | 3 | 01100 |
| 9 | 01001 | 0 | 01101 |
| 10 | 01010 | 1 | 01111 |

Fig. 3.2 Converting from binary to Grey code

We can keep track of how many bulbs are on and stop the loop when all are on (without actually counting bulbs from zero in each iteration):

**code:**

```
// count leading zeros in number like 0 will result in 32
  public static int countLeadingZeros(int value) {
    int res = 0;
    while (value > 0) {
      value /= 2;
      res++;
    }
    return 32 - res;
  }

// to print the steps with min number of steps
  public static int calculateAndPrintSteps(int n) {
    int i = 1, onCount = 0;
    int[] switches = new int[n];
    printArr(switches);

    for (i = 1, onCount = 0; onCount < n; i++) {
      int index = n + countLeadingZeros(i & ~(i - 1)) - 32;
      switches[index] ^= 1; // Toggle
      onCount += switches[index] != 0 ? switches[index] : -1; //
Either increment or decrement
      printArr(switches);
    }
    return i - 1;
  }
```

**Complexity:**

Time: O(N * 2 ^(N) ) where N number of switches.

assuming countLeadingZeros: O(log(32)) ~ O(1)

assuming printArr: O(1) cause printing shouldnt be considered in complexity calculations.

Graph below (Fig. 3.3 )proves the time complexity by graphing with y log scale(Graph using python).

Space: O(N) where N number of switches.

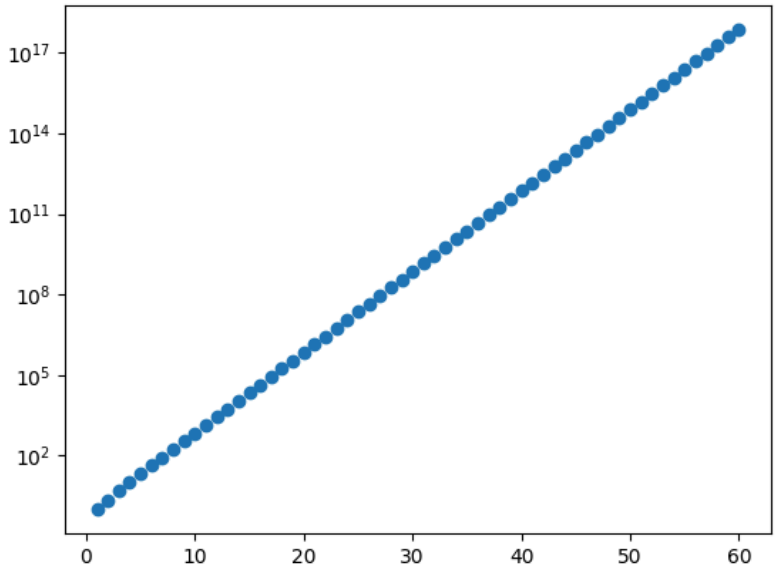



Fig. 3.3: Time complexity with size

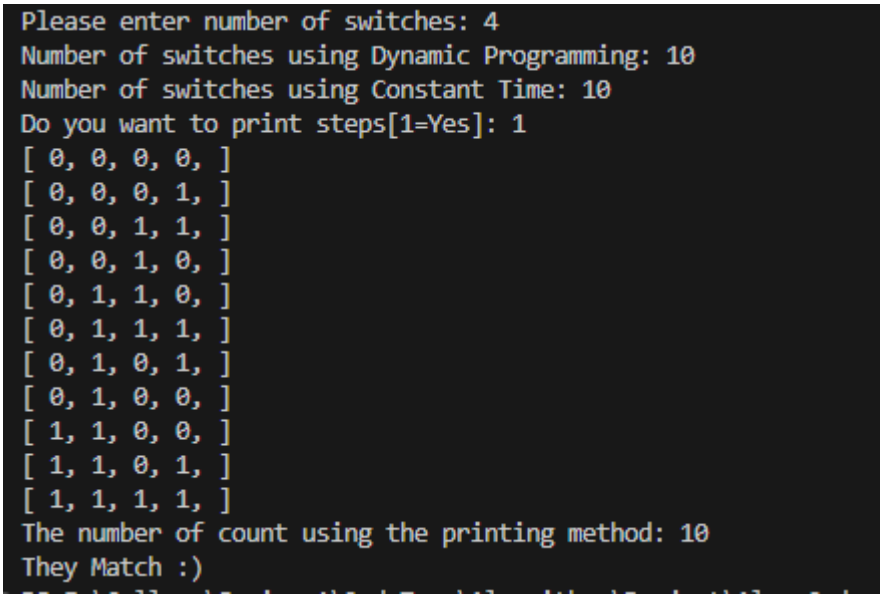
**Comparison with other Techniques:**

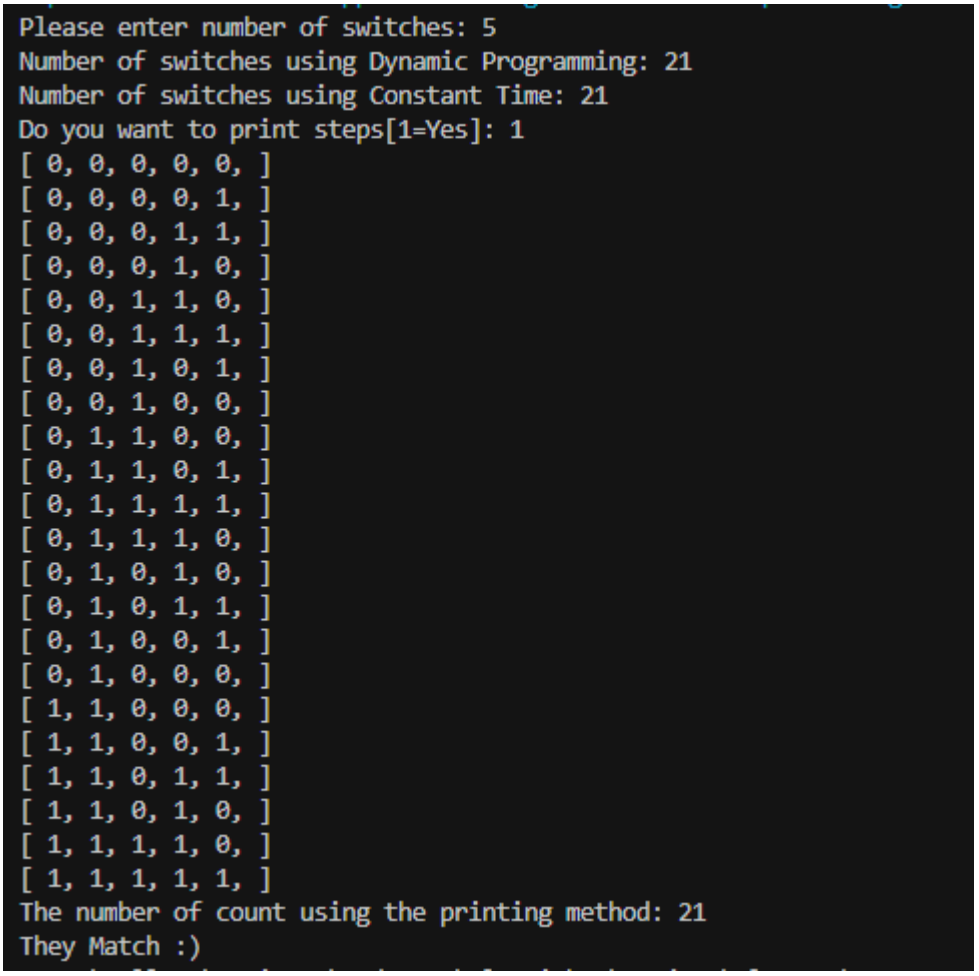
**Table 3.1: Comparison between 3 algorithms for Task 3**

| | Dynamic Programming | Math Technique | Gray Code Technique |
|---|---|---|---|
| **Usage** | - works be identifying the pattern then recursively calling and caching the values not to recalculate.<br><br>- It efficiently calculates the number of switches to turn on. | - Through manual Mathematical Solving we got the base equation which lowers on the complexity being more efficient.<br><br>- it calculates the number of steps in no | - works by using gray codes and the feature that between every two states at max 1 bit change so we used this to our advantage.<br>- It takes more |

| | | | |
|---|---|---|---|
| | | time | time but prints the steps correctly with efficiency in mind. |

**Output Sample with Description:**






**Conclusion:**

All Algorithms offer different trade-offs. but for Choosing the Right Technique:

- If you only need the minimum number of moves for a large number of switches, the Math Technique is the most efficient.
- If you need both the minimum number of moves and the sequence of switch flips, the Gray Code Technique is a good choice.
- For a general-purpose solution that works for any number of switches, Dynamic Programming offers a good balance of efficiency and functionality.

**References:**


- javascript - Lights Out Algorithm - FInd the minimum number of switches to turn on all switches - Stack Overflow
- Algorithmic Puzzles Book by Anany Levitin and Maria Levitin

# Task (4)

**Description:**

There are eight disks of different sizes and four pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on the top.
Use divide and conquer method to transfer all the disks to another peg by a sequence of moves. Only one disk can be moved at a time, and it is forbidden to place a larger disk on top of a smaller one.
Does the Dynamic Programming algorithm can solve the puzzle in 33 moves? If not then design an algorithm that solves the puzzle in 33 moves.

**Detailed Solution:**

**Divide and conquer algo:**

The algorithm begins with the selection of an integer k, such that 1 =< k =< n, where n is the total number of discs which in our problem equals 8. When there is only one disc, move it from peg 1 to peg 4 and stop. For n > 1, the algorithm proceeds recursively using the following three steps:

1. Move the n - k smallest discs from peg 1 to peg 2 using all four pegs. There are many ways to do this; the best may be a slight modification of the Tower of Hanoi solution. Regardless, moving these pieces is very quick because of the extra peg one can use beyond the Tower of Hanoi solution; it should allow you to move the discs in 2^(n-k-1) time, if not better.
2. Move the rest of the discs from peg 1 to peg 4, using pegs 1, 2, and 4 in the classic Tower of Hanoi solution. This takes 2^(k) time.
3. Then, move the n - k smallest discs to peg 4, using all four pegs. This again takes 2^(n-k-1) time.

It takes 33 moves.

## Code:

tart here | *try.cpp | *fr.cpp

```cpp
#include <iostream>
#include <cmath>
using namespace std;

    void hanoi(int n, int k, string from, string temp, string to) {
       if (n == 0) {
            cout<<"";
            return;
       }
       hanoi(n - 1, k, from, to, temp);
       cout<<"Move disc "<< (n + k) << " from " << from << " to " <<to<<"\n";
       hanoi(n - 1, k, temp, from, to);

   }
    void rev(int n, string from, string temp, string to, string temp2) {
        //k is random no, 1<=k<=n
       int k = (int) (n + 1 - round(sqrt(2 * n + 1)));
       if (k == 0){
           cout<<"Move disc " << 1  << " from " << from << " to " << to<<"\n";
           return;
       }
       rev(k, from , to, temp , temp2);
       hanoi(n-k,k, from, temp2, to);
       rev(k,temp,from,to,temp2);
   }

   int main(){
   int n = 8;
   rev(n, "A", "B", "D", "C");
    }
```

## Output:

```
Move disc 1 from A to B
Move disc 2 from A to C
Move disc 3 from A to D
Move disc 2 from C to D
Move disc 1 from B to D
Move disc 4 from A to C
Move disc 5 from A to B
Move disc 4 from C to B
Move disc 1 from D to A
Move disc 2 from D to C
Move disc 3 from D to B
Move disc 2 from C to B
Move disc 1 from A to B
Move disc 6 from A to D
Move disc 7 from A to C
Move disc 6 from D to C
Move disc 8 from A to D
Move disc 6 from C to A
Move disc 7 from C to D
Move disc 6 from A to D
Move disc 1 from B to D
Move disc 2 from B to C
Move disc 3 from B to A
Move disc 2 from C to A
Move disc 1 from D to A
Move disc 4 from B to C
Move disc 5 from B to D
Move disc 4 from C to D
Move disc 1 from A to B
Move disc 2 from A to C
Move disc 3 from A to D
Move disc 2 from C to D
Move disc 1 from B to D
33
Process returned 0 (0x0)   execution time : 0.034 s
Press any key to continue.
```

**Dynamic programming algorithm:**

To implement memoization in the Tower of Hanoi problem, we can store the results of subproblems in a vector.

In this modification, I've changed the return type of the Hanoi function to int to return the number of moves made, which is required for memoization. I've also added a vector<vector<int>> memo to store the results of subproblems. Before computing a subproblem, the code checks if the result is already stored in the memoization vector. If it is, it retrieves the result from the vector instead of recomputing it. After computing a subproblem, the result is stored in the memoization vector for future use.

Now it takes 15 moves instead of 33 moves.

**Assumptions:**

Dynamic programming is used to skip pattern of moves made before in the same problem of eight disks not other problems of different number of disks.

**Code:**

```cpp
#include <iostream>
#include <cmath>
#include <vector>

using namespace std;

vector<vector<int>> memo; // Memoization vector

int hanoi(int n, int k, string from, string temp, string to) {
    if (n == 0) {
        return 0;

    }

    if (memo[n][k] != -1) {
        return memo[n][k];

    }

    int moves = 0;
    moves += hanoi(n - 1, k, from, to, temp);
    cout << "Move disc " << (n + k) << " from " << from << " to " << to << "\n";
    moves++;
    moves += hanoi(n - 1, k, temp, from, to);
    memo[n][k] = moves; // Store result in memoization vector
    return moves;

}
```

```cpp
void rev(int n, string from, string temp, string to, string temp2) {
    int k = (int)(n + 1 - round(sqrt(2 * n + 1)));
    if (k == 0) {
        cout << "Move disc " << 1 << " from " << from << " to " << to << "\n";
        return;
    }
    rev(k, from, to, temp, temp2);
    hanoi(n - k, k, from, temp2, to);
    rev(k, temp, from, to, temp2);

}

int main() {
    int n = 8;
    // Initialize memoization vector with -1 values
    memo.assign(n + 1, vector<int>(n + 1, -1));
    rev(n, "A", "B", "D", "C");

}
```

**Output:**

```
Move disc 1 from A to B
Move disc 2 from A to C
Move disc 3 from A to D
Move disc 1 from B to D
Move disc 4 from A to C
Move disc 5 from A to B
Move disc 1 from D to A
Move disc 1 from A to B
Move disc 6 from A to D
Move disc 7 from A to C
Move disc 8 from A to D
Move disc 1 from B to D
Move disc 1 from D to A
Move disc 1 from A to B
Move disc 1 from B to D
15
Process returned 0 (0x0)   execution time : 0.027 s
Press any key to continue.
```

**Time Complexity:**

O(2^n) for divide and conquer and memoization.

Best case scenario in memoization would be O(n^2).

**Conclusion:**

Memoization helps in avoiding redundant calculations, but it doesn't change the inherent complexity of the problem.

Even though we avoid re-calculating previously computed results, we still need to compute O(2^n) unique subproblems in the worst case because the Tower of Hanoi problem has exponential complexity.

Memoization only improves efficiency by eliminating redundant work, but it doesn't change the fundamental exponential nature of the problem.

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

**Although, in the best-case scenario where memoization is fully utilized, the time complexity of the code would be O(n^2).**

**References:**


1. Reve's puzzle - Everything2.com
2. The Divide-and-Conquer Approach to the Generalized p-Peg Tower of Hanoi Problem A.A.K. Majumdar
3. Algorithmic Puzzles by Anany V. Levitin
4. java - Optimal solution for Reve's puzzle using recursion - Stack Overflow
5. rahulch95/Tower-Of-Hanoi-4-Stools-Solver: Solves the tower of hanoi problem recursively for four stools and visualizes it for you. It uses memoization to enhance the speed. (github.com)
6. https://www.sciencedirect.com/science/article/pii/S0166218X01002876?ref=pdf_download&fr=RR-2&rr=8812be2ab9cf794b
7. https://youtu.be/MbonokcLbNo?si=ioDpnYFswUk4gtJ1
8. https://medium.datadriveninvestor.com/tower-of-hanoi-solve-and-optimize-with-memoization-f215a1bd201f
9. https://oeis.org/A007664/a007664.pdf

# Task (5)

**Description:**

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.

**Detailed Solution:**

**Assumption:**

1. The number of coins must be divisible by 4.
2. The row of coins is sorted and there are no missing numbers.
3. Any empty space between adjacent coins is ignored.
4. On each move, a coin can jump right or left but it has to land on a single coin.
5. Jumping over a coin pair counts as jumping over two coins.

**Pseudocode:**

```
function Pairs(n):
  if n is not divisible by 4:
    print "There is no solution for this number of coins"
    return

  Initialize moves = 0, jumps = 1
  Generate a row of coins from 1 to n

  Start forming pairs:
  while there are still pairs to form:
    Form pairs from the rightmost coin:
```

```
    Iterate over the coins from right to left:
      if jumps > n / 4 - 1:
        break
      if the coin is not paired:
        Jump over the required number of coins and pair
        Increment jumps and moves

  Form pairs from the leftmost coin:
    Iterate over the coins from left to right:
      if the coin is not paired and can jump:
        Jump over the required number of coins and pair
        Increment jumps and moves

Display the result:
print "Minimum number of moves:" moves
print "Pairs formed:"
for each pair of coins formed:
  print the pair
```

**Steps:**

1. Check if the number of coins is divisible by 4. If not, there's no solution.
2. Initialize the number of moves and jumps to 0 and 1, respectively.
3. Generate a row of coins numbered from 1 to n.
4. Start forming pairs:
    - Iterate over the coins from right to left, forming pairs from the rightmost coin:
        -Jump over the required number of coins to form a pair.
        -Increment jumps and moves.
    - Iterate over the coins from left to right, forming pairs from the leftmost coin:
        -Jump over the required number of coins to form a pair.

-Increment jumps and moves.

5. Display the minimum number of moves required and the pairs of coins formed.

Following these steps, we can solve the problem efficiently by forming pairs of coins while minimizing the number of moves.

**Code:**

```
#include <iostream>
#include <vector>

using namespace std;

// Function to form pairs of coins starting from the rightmost
coin
void Right(vector<vector<int>> &row, int &jumps, int &moves) {
    int n = row.size();
    // Iterate over the coins from right to left
    for (int i = n - 1; i > 0; i--) {
        // Check if the required number of pairs has been formed
        if (jumps > (n / 4) - 1)
            break;

        // If the coin is already paired, skip it
        if (row[i][0] == -1)
            continue;

        int index = i - 1;
        int currentJumps = 0;

        // Keep jumping until the required number of coins to
jump over is reached
        while (currentJumps != jumps) {
            if (row[index].size() == 1 && row[index][0] != -1)
                currentJumps++;

            if (row[index].size() == 2)
                currentJumps += 2;

            index--;
        }
```

```
        // Find the next available coin to make a pair with after
the required number of jumps is reached
        while (row[index][0] == -1) {
            index--;
        }

        // Make the coin pair and update the coins row to
indicate the new empty position
        row[i].push_back(row[index][0]);
        row[index][0] = -1;

        jumps++;
        moves++;
    }
}

// Function to form pairs of coins starting from the leftmost
coin
void Left(vector<vector<int>> &row, int &jumps, int &moves) {
    int n = row.size();
    // Iterate over the coins from left to right
    for (int i = 0; i < n; i++) {
        // If it is a pair or empty space, skip it
        if (row[i][0] == -1 || row[i].size() == 2)
            continue;

        int index = i + 1;
        int currentJumps = 0;

        // Keep jumping until the required number of coins to
jump over is reached
        while (currentJumps != jumps) {
            if (row[index].size() == 1 && row[index][0] != -1)
                currentJumps++;

            if (row[index].size() == 2)
                currentJumps += 2;

            index++;
        }
```

```
        // Find the next available coin to make a pair with after
the required number of jumps is reached
        while (row[index][0] == -1) {
            index++;
        }

        // Make the coin pair and update the coins row to
indicate the new empty position
        row[index].push_back(row[i][0]);
        row[i][0] = -1;

        jumps++;
        moves++;
    }
}

// Function to generate a row of coins
vector<vector<int>> toRow(int n) {
    vector<vector<int>> row;

    // Make a row of coins numbered from 1 to n
    for (int i = 1; i < n + 1; i++) {
        row.push_back({i});
    }

    return row;
}

// Function to display the result
void displayResult(vector<vector<int>> &row, int &moves) {
    cout << "Minimum number of moves: " << moves << endl;

    for (int i = 0; i < row.size(); i++) {
        if (row[i][0] == -1)
            continue;

        cout << "( ";
        for (int j = 0; j < row[i].size(); j++) {
            cout << row[i][j] << " ";
        }
        cout << ") ";
    }
}
```

```
// Function to orchestrate the process of forming pairs of coins
void Pairs(int n) {
    // Check if the number of coins allows for a solution
    if (n % 4 != 0) {
        cout << "There is no solution for this number of coins "
<< endl;
        return;
    }

    int moves = 0, jumps = 1;
    // Generate the row of coins
    vector<vector<int>> row = toRow(n);

    // Start forming pairs from the rightmost coin until no more
pairs can be formed
    Right(row, jumps, moves);

    // Continue forming pairs from the leftmost coin until no
more pairs can be formed
    Left(row, jumps, moves);

    // Display the result
    displayResult(row, moves);
}

int main() {
    // Call the function to form pairs with 20 coins
    Pairs(14);
    return 0;
}
```

**Complexity Analysis:**

**Time Complexity:**

The time complexity of the code is O(n^2), where n is the number of coins. This is because the Right and Left functions each iterate over the coins row once, and within each iteration, they perform a while loop that can iterate up to n times. Therefore, the overall time complexity is O(n^2).

**Space Complexity:**

The space complexity of the code is O(n), where n is the number of coins. This is because the code uses a vector of vectors to store the coins row, which requires O(n) space. Additionally, the code uses a few integer variables to keep track of the number of moves and jumps, which also require O(1) space. Therefore, the overall space complexity is O(n).

**Other Techniques:**

We can use the Brute force technique to try all the possible pairings.

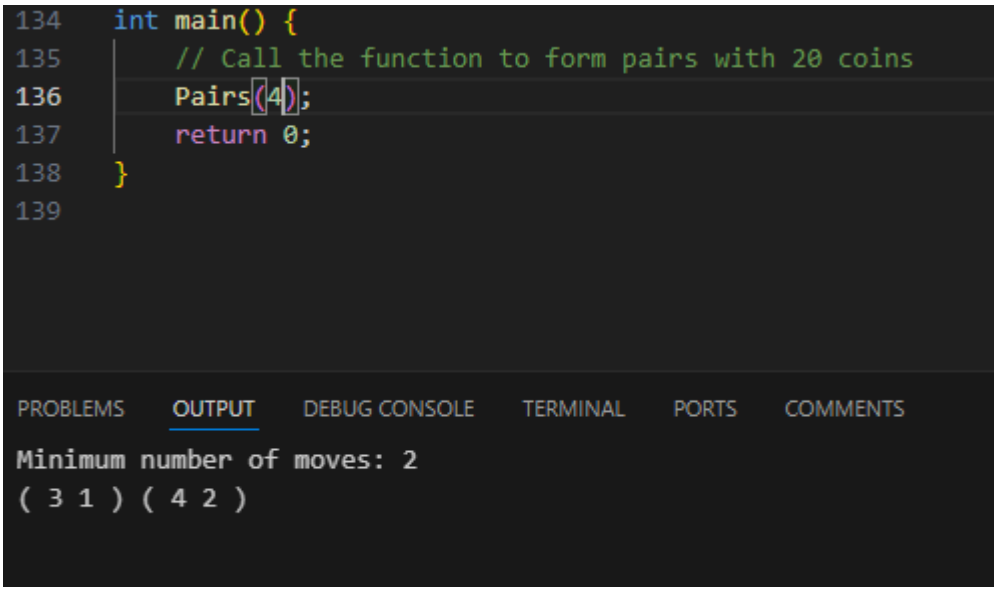
**Comparison:**

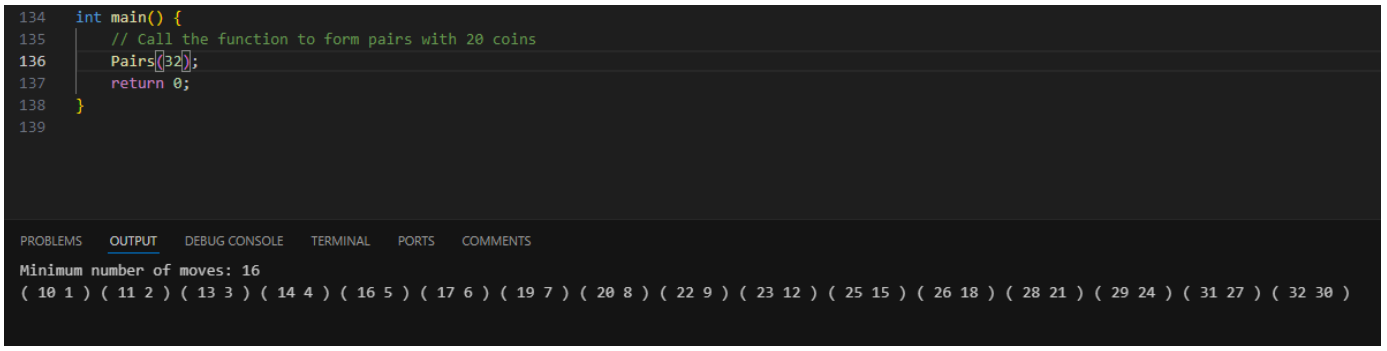
**Table 5.1: Comparison between 2 algorithms for Task 5**

| | **Greedy** | **Brute force** |
|---|---|---|
| **Usage** | The greedy algorithm iterates through the coins, starting from the rightmost and then the leftmost, attempting to pair them.<br>At each step, it tries to pair the current coin with the nearest available coin, ensuring that the pairing process follows a specific pattern.<br>It stops pairing when no more pairs can be formed according to the predefined pattern. | The brute-force technique would systematically try all possible pairings of coins.<br>It would generate all permutations of coin pairs and calculate the total number of moves for each permutation.<br>Finally, it would select the permutation with the minimum number of moves as the optimal solution. |
| **Time Complexity** | The time complexity of the code is O(n^2), where n is the number of coins. This is because the Right and Left functions each iterate over the coins row once, and within each iteration, they perform a while loop that can iterate up to n times. Therefore, the overall time complexity is O(n^2). | The brute-force technique, however, would have an exponential time complexity. If there are n coins, there would be (n-1)!! possible permutations to consider, where !! denotes the double factorial. This leads to a significantly higher time complexity compared to the greedy algorithm. |
| **Optimality** | The greedy algorithm provides a solution quickly but may not always | The brute-force technique guarantees finding the |

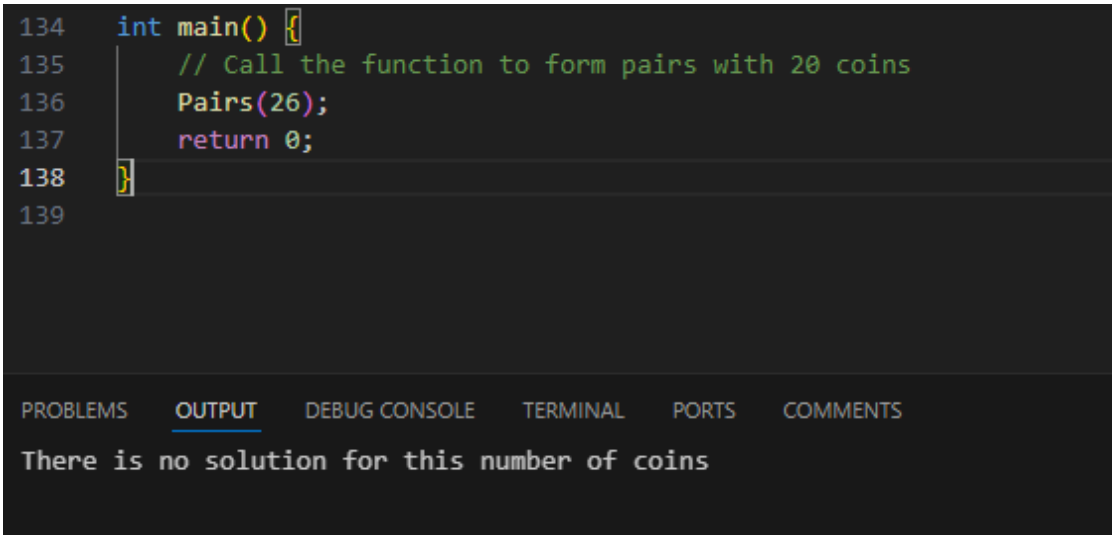
| | | |
|---|---|---|
| | yield the optimal solution. It makes locally optimal choices at each step without considering the global optimum. | optimal solution as it exhaustively searches through all possible combinations. However, this comes at the cost of increased time complexity, which becomes impractical for large values of n. |

**Sample output:**




These are 2 examples when the number of coins is divisible by 4 ,and the number of moves here is the minimum which is half the number of coins.




This is an example when the number of coins is even but not divisible by 4.

**Conclusion:**

In summary, the greedy algorithm provides a fast but potentially suboptimal solution, while the brute-force technique guarantees optimality but at the expense of significantly higher computational resources, especially for larger problem sizes.

**References:**


- https://www.geeksforgeeks.org/greedy-algorithms-general-structure-and-applications/
- Algorithmic Puzzles Book by Anany Levitin and Maria Levitin
- Introduction to The Design & Analysis of Algorithm by Anany Levitin

# Task (7)

**Description:**

A computer game has a shooter and a moving target. The shooter can hit any of n > 1 hiding spot 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 Dynamic Programing algorithm that guarantees hitting the target.

**Assumptions:**

- The N hiding spots are located along a straight line, and the shooter can hit any of the hiding spots accurately.
- The Target can start at any hiding spot from 1 to n.
- The target moves to an adjacent spot either behind or ahead randomly between every 2 consecutive shots.
- The shooter cannot see the target.

**Detailed Solution:**

**Approach:**

Dynamic Programming: We employ a dynamic programming algorithm to systematically explore all possible states and optimize shot sequences to hit the target efficiently.

**Pseudocode:**

```
function hit_target(n, shooter_pos, target_pos, dp, path):

    if shooter_pos is out of bounds:

        return infinite shots

    if shooter_pos == target_pos:

        return 1

    if n == 2:

        return 2 shots

    if state (shooter_pos, target_pos) is memoized:

        return memoized value
```

```
    simulate target's movement randomly

    simulate shooter's movement

    shots_required = min(1 + hit_target(next_shooter_pos,
next_target_pos), dp[shooter_pos][target_pos])

    update dp table and path table

    return shots_required
```

**Steps:**

Define two structures:

- State: Represents the state of the dynamic programming table, including whether the target is found and the number of shots taken.
- Shot: Represents a shot, including the shooter's position and the target's position.

Implement Recursive Function hit_target:

- This function recursively calculates the minimum number of shots required to hit the target from a given shooter position and target position.
- It also updates the dynamic programming table (dp) to memoize the results for already computed states

Base Cases:

- If the shooter reaches the target, the minimum shots required is 1.
- If n=2, it's guaranteed to hit the target with 2 shots at the same spot.

Memoization:

- Before recursing, the function checks the memoization table to see if the current state has already been computed.
- If the state is found, the function returns the previously computed result.

Simulating Target Movement:

- The target's movement is simulated randomly. It has a 50% chance of moving forward and a 50% chance of moving backward.

Simulating Shooter's Movement:

- The shooter's movement is simulated, ensuring it stays within the valid range of hiding spots.
- The shooter initial position is at n=1 (The second spot) and increments until it reaches the n-1  spot. this will guarantee hitting the target if the target initially started at an even spot.
- After reaching the n-1 spot the shooter starts again at the second spot now if the target was initially at an odd number spot it is guaranteed to be hit in this second trial.

Recursion:

- The function recursively calls itself for the next state to find the minimum shots required.

Storing Shot Path:

- The function stores the shot's path in the path table.

**Code Implementation:**

```
#include <iostream>

#include <vector>

#include <climits>

#include <cstdlib> // For rand() and srand()

#include <ctime>   // For seeding srand()

using namespace std;

struct State

{   bool found;

    int shots;

};

struct Shot

{   int shooter_pos;

    int target_pos;

};
```

```
int hit_target(int n, int shooter_pos, int target_pos,
vector<vector<State>> &dp, vector<vector<Shot>> &path)

{   // Check if the shooter's position is within the valid range

    if (shooter_pos < 0 || shooter_pos >= n)

    {  return INT_MAX; // Out of bounds, return worst-case
scenario (infinite shots)

    }

    // Base case: If shooter reaches target

    if (shooter_pos == target_pos)

    {

        dp[shooter_pos][target_pos].shots = 1;

        dp[shooter_pos][target_pos].found = true;

        return 1;

    }

    // Base case: If n=2, 2 shots at the same spot will guarantee
hitting the target

    if (n == 2)

    {

        dp[0][0].shots = 2;

        dp[0][0].found = true;

        return 2;

    }

    // Check memoization table before recursion

    if (dp[shooter_pos][target_pos].shots != INT_MAX)

    {

        return dp[shooter_pos][target_pos].shots;
```

```
    }

    // Simulate the target's movement randomly

    int new_target_pos = target_pos;

    if (rand() % 2 == 0)

    { // 50% chance of moving forward

        new_target_pos = min(n - 1, target_pos + 1);

    }

    else

    { // 50% chance of moving backward

        new_target_pos = max(0, target_pos - 1);

    }

    // Simulate the shooter's movement

    int new_shooter_pos;

    if (shooter_pos < n - 1 && shooter_pos >= 1)

    {

        new_shooter_pos = shooter_pos + 1; // Move one step
forward

    }

    else

    {

        new_shooter_pos = 1; // Reset shooter position if out of
bounds

    }

    // Recursive call to find minimum shots required

    dp[shooter_pos][target_pos].shots =
min(dp[shooter_pos][target_pos].shots, 1 + hit_target(n,
new_shooter_pos, new_target_pos, dp, path));
```

```
    // Store the shot's path

    if (dp[shooter_pos][target_pos].shots == 1 +
dp[new_shooter_pos][new_target_pos].shots)

    {

        path[shooter_pos][target_pos] = {new_shooter_pos,
new_target_pos};

    }

    return dp[shooter_pos][target_pos].shots;

}

int main()

{

    srand(time(0)); // Seed the random number generator

    int n; // Number of positions

    cout << "Enter the number of hiding spots: ";

    cin >> n;

    int target_pos; // Initial position of the target

    cout << "Enter the target position (0 to " << n - 1 << "): ";

    cin >> target_pos;

    // Validate the target position

    if (target_pos < 0 || target_pos >= n)

    {

        cout << "Invalid target position." << endl;

        return 1;

    }

    vector<vector<State>> dp(n, vector<State>(n, {false,
INT_MAX}));
```

```
    vector<vector<Shot>> path(n, vector<Shot>(n));

    int shooter_pos = 1; // Initial position of the shooter

    // Call the function to calculate minimum shots required

    int min_shots = hit_target(n, shooter_pos, target_pos, dp,
path);

    if (min_shots == INT_MAX)

    {

        cout << "Target cannot be hit." << endl;

    }

    else

    {

        cout << "Minimum shots required to hit the target: " <<
min_shots << endl;

        cout << "Shot Path:" << endl;

        int current_shooter = shooter_pos;

        int current_target = target_pos;

        while (current_shooter != current_target)

        {

            cout << "Shooter at position " << current_shooter <<
" shoots: The target position is " << current_target << endl;

            Shot next_shot =
path[current_shooter][current_target];

            current_shooter = next_shot.shooter_pos;

            current_target = next_shot.target_pos;

        }

        cout << "Target hit at position " << current_target <<
endl;
```

```
    }

    return 0;

}
```

**Complexity Analysis:**

**Time Complexity:**

The time complexity of the algorithm is O(n^2 ), where n is the number of hiding spots.

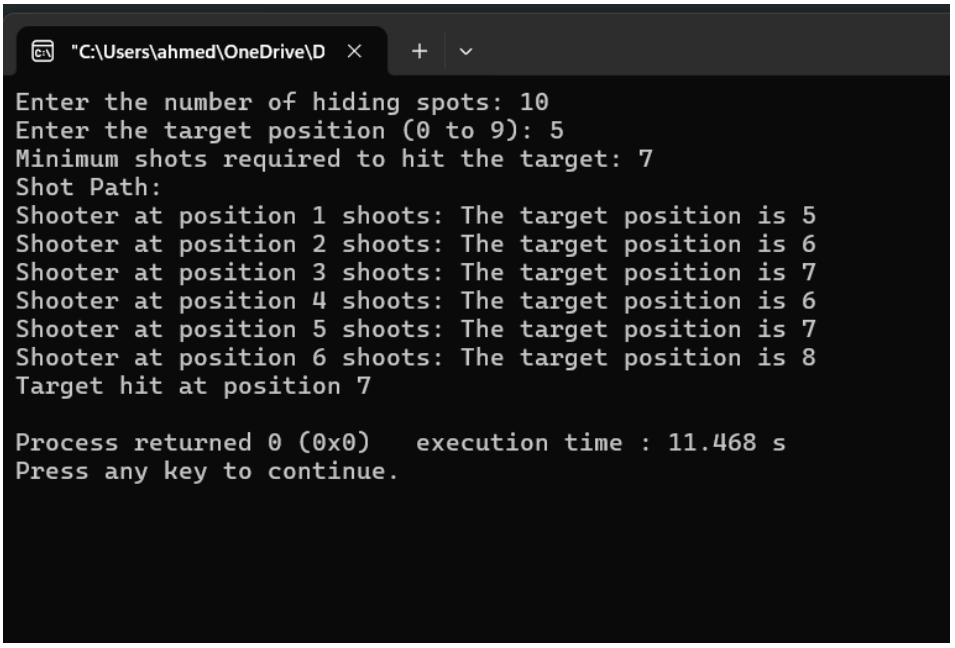
- This is because there are n possible shooter positions and n possible target positions for each shooter position.
- The memoization technique ensures that each state is computed only once, leading to efficient computation.
-

**Space Complexity:**

The space complexity of the algorithm is O(n 2 ), where n is the number of hiding spots.

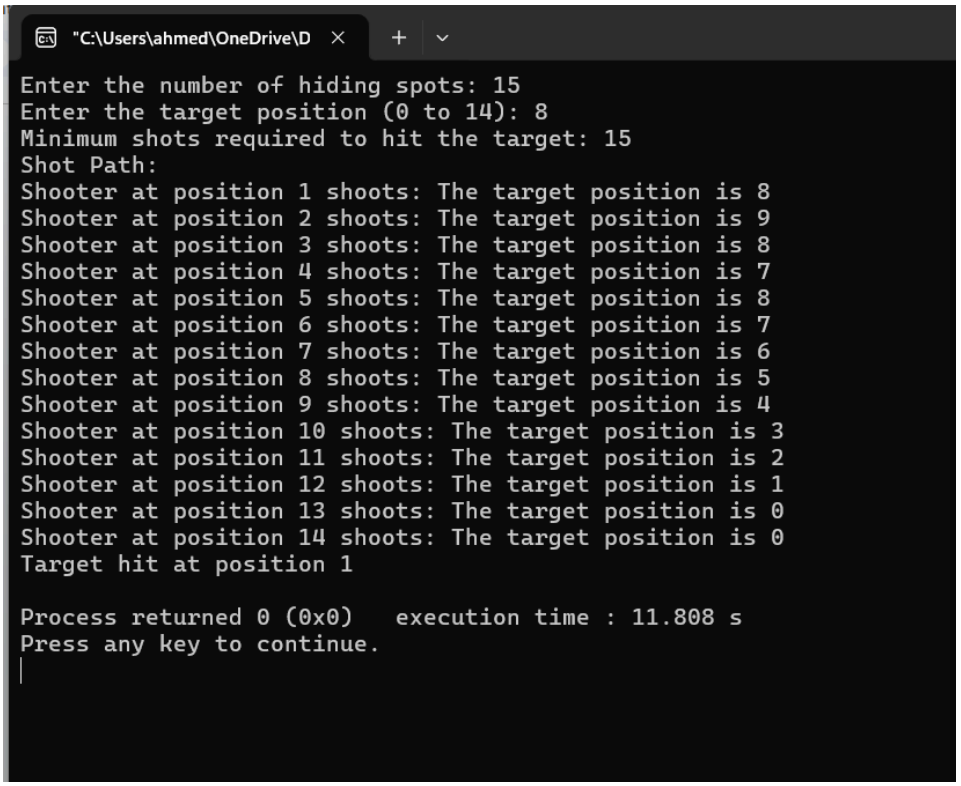
- This is due to the storage required for the dynamic programming tables (dp and path).

**Sample Output:**



Sample Output with number of hiding spots initialized to 10 and the target initial position was at spot 5.

It took 7 shots to hit the target and the target was hit at position 7, with the path of each shot shown




Sample Output with number of hiding spots initialized to 15 and the target initial position was at spot 8.

It took 15 shots to hit the target and the target was hit at position 1, with the path of each shot shown

**Another Implementation using a Greedy Algorithm:**

**Steps:**

Numbering the Hiding Spots:

- Start by numbering the hiding spots left to right from 1 to n.

Initial Shot Selection:

- The shooter makes the first shot at either spot 2 or spot n−1. These spots are chosen because they guarantee either hitting the target or ensuring that the target cannot move to a spot where it would be safe from subsequent shots.

Shooting Sequence:

Even-Numbered Target Spot:

- If the target is initially in an even-numbered spot:

- After the first shot, the target either gets hit or moves to an odd-numbered spot ≥3.
- The shooter then makes the second shot at spot 3, guaranteeing hitting the target or ensuring that the target will be at an even-numbered spot ≥4. The shooter continues shooting at consecutive spots 4,5,...,n−1 until the target is hit.

Odd-Numbered Target Spot:

- If the target is initially in an odd-numbered spot:
- The shooter's first n−2 shots will not hit the target because the locations of the target and shots will always have opposite parities. However, after the shot at spot n−1, the target will move to a spot with the same parity as n−1.
- The shooter repeats the symmetric sequence of shots, hitting consecutively spots n−1,n−2,...,2, guaranteeing hitting the target in this case as well.

**Pseudocode:**

```
function hitTarget(n, target_pos):

    if n <= 2:

        return n  // For n <= 2, two shots at the same spot solve
the problem



    // Start shooting sequence

    shooter_pos = 2

    shots = 0



    // Shoot at consecutive spots 2 to n-1

    for i = 2 to n-1:

        // Shoot at current position

        print("Shooter at position", shooter_pos, "shoots at
target position", i)

        shots++



        // Check if target is hit
```

```
        if i == target_pos:
            print("Target hit at position", target_pos)
            return shots

    // Shoot back to position 2
    for i = n - 1 down to 2:
        // Shoot at current position
        print("Shooter at position", i, "shoots at target
position", i)
        shots++

        // Check if target is hit
        if i == target_pos:
            print("Target hit at position", target_pos)
            return shots

    // If target is not hit within the sequence, return shots
fired
    return shots
```

**Comparison between Dynamic Programming and Greedy Algorithm:**

**Dynamic Programming Approach:**

Advantages:

- Guarantees finding the optimal solution by systematically exploring all possible shooter positions and target movements.

- Handles arbitrary values of n by recursively calculating the minimum shots required.
- Can track the path of shots taken to hit the target.

Disadvantages:

- Requires additional memory for memoization table and path tracking, leading to increased space complexity.
- Time complexity can be high for large values of n due to recursive function calls and memoization.

**Greedy Approach**

Advantages:

- Simple and straightforward approach based on a predefined sequence of shots.
- Guarantees hitting the target for any given value of n with a fixed number of shots.
- Requires minimal memory and computational resources.

Disadvantages:

- May not always find the optimal solution in all scenarios.
- Lack of flexibility for handling variations or edge cases of the problem.

**Conclusion:**

In summary, while the dynamic programming approach provides a more general and flexible solution that guarantees optimality, it comes with higher computational and memory overheads. On the other hand, the greedy approach offers simplicity and efficiency but may not always find the optimal solution and lacks adaptability to different problem instances.
Specifically in this problem I see that the greedy algorithm is a better choice as it has Time complexity of n and guarantees hitting the target.

**References:**

Dynamic Programming (DP) Tutorial with Problems - GeeksforGeeks

Solving Shooter's Moving Target Problem with Dynamic Programming Algorithm (devcodef1.com)

Algorithmic Puzzles Book by Anany Levitin and Maria Levitin

## Task (8)

**Detailed Assumptions:**

-49 boxes have real metal

-these 49 boxes have same weight

-one real metal weights 2 kg

-one box have fake metals

-one fake metal weights 1 kg

-this box weights less than the others

**Problem description:**

If you have 50 boxes that contains 50 pieces of metal all of the same known weight. one of these boxes contains fake metal pieces that weigh 1 kilogram less than the pieces in the rest of the boxes. You can use a digital scale only once to find this fake box. Design a brute force algorithm to solve this problem.

**Detailed solution:**

We have 50 boxes one of them contains fake metal and weights less than the rest.

We will number them from 1 to 50.

We will take one piece of metal from 1st box, two pieces of metal from 2nd box, three pieces of metal from 3rd box and so on…

From assumptions real metals weight 2 kg and we have 50 boxes so if there is no fake metal the total weight of pieces we took should equal to 2550 kg but it will be less than that because we have fake pieces.

To get the index of the box that contains fake metals we subtract the real total weight from the 2550kg

The subtraction solution equals the index of the box containing fake metals

## pseudo-code:

```
// Define the normal weight of a metal piece
NormalWeight = 2

// Create an array of boxes and initialize all with correct weights
```

Create array box with size 50

for i = 0 to 49 do

box[i] = 50 * NormalWeight

// Select a random box to contain fake pieces of metal

randomNumber = Random number between 0 and 49

// Place fake metal pieces in the randomly selected box

box[randomNumber] = 50 * (NormalWeight - 1)

// Initialize variables for sum calculation

sumWeWant = 0

sum = 0

// Iterate through the boxes, taking 1 piece from box 1, 2 pieces from box 2, etc.

// If there were no fake pieces, sum would be equal to sumWeWant

for i = 0 to 49 do

sum += (box[i] / 50) * (i + 1)

sumWeWant += (50 * NormalWeight / 50) * (i + 1)

// Use a digital scale to measure the sum and calculate the difference

rest = sumWeWant - sum - 1

// Output the difference and the index of the box containing fake pieces

Output rest

Output randomNumber

**Code:**

```
import java.util.Random;


public class Task8 {
    public static void main(String [] args) {
```

```
        //normal wieght of a metal piece so a fake one
will have weight of 1 (one kg less)
        int NormalWeight = 2;

        //creating array of boxes and initialising all
with right weights
        int[] box = new int[50];
        for (int i = 0; i < 50; i++) {
            box[i] = 50 * NormalWeight;
        }

        //selecting a random box to have fake pieces of
metal
        Random rand = new Random();
        int randomNumber = rand.nextInt(50);

        //now we have a random box that have fake metal
pieces
        box[randomNumber] = 50 * (NormalWeight - 1);
        System.out.println("The random fake box index = "
+ randomNumber);

        int sumWeWant = 0;
        int sum =0;

        //we take 1 piece from box 1 & 2 pieces from box 2
& 3 pieces from box 3 etc...
        //if there were no fake pieces sum would be equal
to sumWeWant
        for(int i=0;i<50;i++){
            sum += (box[i] /50)*(i+1) ;
            sumWeWant += (50*NormalWeight /50)*(i+1) ;
        }
```

```
        System.out.println("The calculated sum = "+ sum);
        System.out.println("If there is no fake metal the
sum would be = "+ sumWeWant);



        //we use digital scale to know the sum and remove
it from sum we want
        int rest = sumWeWant-sum -1;

        System.out.println("Difference between the two
gives us the index of the fake metal box --> " + rest);




    }
}
```

**Complexity analysis for the algorithm:**

O(n)

**A comparison between my algorithm and one other technique that can be used to solve the problem:**

We can also use divide and conquer algorithm to solve this problem.

We can divide the 50 boxes into 2 groups of 25 boxes and weigh one group, if it equals the actual weight we expect then the fake box exists in the other group if not then this group contains the fake box.

We keep repeating the previous step until we know the fake box.

**Table 8.1: Comparison Between 2 algorithms for Task 8**

| | Brute force | divide and conquer |
|---|---|---|
| | brute force usually exhaustively searches through all possible combinations of solutions but in this case and because we need to use the digital scale only once, there is only one possible solution which makes this algorithm perfect for this problem | divide and conquer can also solve this problem while using the digital scale only once but for each sub problem.<br>It is impossible to solve it using only one use of the scale through all the problem, the divide and conquer algorithm, we divide the problem into smaller subproblems so we can solve it. |

## PseudoCode for divide and conquer:

```
function find_fake_box(boxes){
/*This function uses divide and conquer to find the fake box among a list of boxes
with a single use of a digital scale.
Args:
        boxes: A list of integers representing the weights of the metal pieces in each
box.
Returns:
        The index of the box containing the fake metal piece.
 */
 n = len(boxes)
 //Base case: If there's only one box left, it must be the fake one
 if n == 1:
          return 0
//Divide boxes into two groups (can be adjusted for efficiency)
group_a = boxes[:n // 2]
group_b = boxes[n // 2:]
// Weigh the groups (simulated by summing their weights)
weight_a = sum(group_a)
weight_b = sum(group_b)
// Compare weights to identify the lighter group
```

```
if weight_a < weight_b:
        // Fake box is within group A
        return find_fake_box(group_a)
elif weight_a > weight_b:
        // Fake box is within group B
        return find_fake_box(group_b) + n / 2      // Adjust index for group B
else:
        // Weights are equal, fake box must be in the remaining single box
        return n
```

**Sample output of the solution for the different cases of the technique with proper description for the output:**

```
C:\Users\Lara\.jdks\openjdk-22.0.1\bin\java.exe "-javaagent:C:\Program Files\J
The random fake box index = 37
The calculated sum = 2512
If there is no fake metal the sum would be = 2550
Difference between the two gives us the index of the fake metal box --> 37

Process finished with exit code 0
```

```
C:\Users\Lara\.jdks\openjdk-22.0.1\bin\java.exe "-javaagent:C:\Program Files\J
The random fake box index = 20
The calculated sum = 2529
If there is no fake metal the sum would be = 2550
Difference between the two gives us the index of the fake metal box --> 20

Process finished with exit code 0
```

```
C:\Users\Lara\.jdks\openjdk-22.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrai
The random fake box index = 25
The calculated sum = 2524
If there is no fake metal the sum would be = 2550
Difference between the two gives us the index of the fake metal box --> 25

Process finished with exit code 0
```

From assumptions we have assumed that the weight of the real metal is 2kg (it can always be changed the code will still work well) and the fake ones are 1 kg less.
The code choses a random index between one and 50 (this is our box with fake metal) and then the code works as written in the detailed solution.

**Conclusion:**

When it comes to finding the fake box with just one use of the scale, Brute Force guarantees a solution but requires a lot of mathematical calculations. Divide and Conquer, on the other hand, shines with its efficiency. By cleverly dividing the boxes and comparing groups, it identifies the lighter group (potentially containing the fake box) in just one weighing. Even with additional weighings within the lighter group to pinpoint the fake one. Brute force is significantly faster making it the better algorithm for this problem.

**Reference:**

Brute force algo :
https://www.quora.com/You-have-10-boxes-with-1000-coins-each-either-all-real-or-all-fake-Real-coins-are-16-ounces-fake-ones-are-17-A-scale-that-tells-you-the-weight-of-those-coins-How-do-you-figure-out-which-boxes-are-fake-or-real-in-one

## Contribution List:

| | |
|---|---|
| Andrew Ayman Samir | Task 3 |
| Mohamed Yasser Mohamed | Task 1 |
| Ahmed Abdelrahman Ahmed | Task 7 |
| Ereny Hany Hanna | Task 4 |
| Omar Salah Mansour | Task 5 |
| Mina Hany Hanna | Task 2 |
| Martin Ashraf Ibrahim | Task 3 |
| Lara Maurice Youssef | Task 8 |
| Kyrillos Ashraf Gamil | Task 6 |