**Project Report**



**Faculty Of Engineering Ain Shams University**

**CSE332s: Design and Analysis of Algorithm**

**Team 12 Members**

|  |  |  |
| --- | --- | --- |
| **Name** | **ID** | **Task** |
| Fathy Abdlhady Fathy | 2001152 | 1 , 7 , 8 |
| Mohamed Ayman Mohamed Soliman | 2001048 | 1 , 7 |
| Yousef Shawky Mohamed | 2001500 | 7 , 8 |
| Ahmed Youssef Mansour Mahfouz | 2002238 | 6 |
| Youssef Wael Hamdy Ibrahim Ashmawy | 2001430 | 2 , 6 |
| Omar Nader Ahmed Gamal El Din | 2001714 | 3 |
| Mark Ramy Fathy | 2000923 | 4 |
| Heba Maher Abdelrahman | 2001400 | 4 |
| Mohamed Mostafa Mahmoud Mohamed | 2001299 | 6 |
| Omar Saleh Mohamed | 2001993 | 2 , 3 |
| Abdallah Mohamed | 2001803 | 5 |

Contents

[List Of Tables 1](#_Toc166179542)

[List Of Figures 1](#_Toc166179543)

[Task 1 : Tromino Tilings 2](#_Toc166179544)

[1.1 Problem Description 2](#_Toc166179545)

[1.2 Detailed Assumptions 2](#_Toc166179546)

[1.3 Detailed Solution 2](#_Toc166179547)

[1.3.1 Description of Solution 2](#_Toc166179548)

[1.3.2 Pseudo Code 3](#_Toc166179549)

[1.4 Complexity Analysis for the Algorithm 5](#_Toc166179550)

[1.5 Sample Output of the Solution 5](#_Toc166179551)

[1.6 Comparison with Another Algorithm 6](#_Toc166179552)

[1.7 Conclusion 6](#_Toc166179553)

[Task 2 : Knight Tour Closed 7](#_Toc166179554)

[2.1 Problem Description 7](#_Toc166179555)

[2.2 Detailed Assumptions 7](#_Toc166179556)

[2.3 Detailed Solution 8](#_Toc166179557)

[2.3.1 Description of Solution 8](#_Toc166179558)

[2.3.2 Pseudo Code 8](#_Toc166179559)

[2.4 Complexity Analysis for the Algorithm 9](#_Toc166179560)

[2.5 Sample Output of the Solution 9](#_Toc166179561)

[2.6 Comparison with Another Algorithm 10](#_Toc166179562)

[2.7 Conclusion 10](#_Toc166179563)

[Task 3 : Security Switches 11](#_Toc166179564)

[3.1 Problem Description 11](#_Toc166179565)

[3.2 Comment 11](#_Toc166179566)

[3.3 Detailed Solution 12](#_Toc166179567)

[3.3.1 Description of Solution 12](#_Toc166179568)

[3.3.2 Pseudo Code 13](#_Toc166179569)

[3.4 Complexity Analysis for the Algorithm 14](#_Toc166179570)

[3.5 Sample Output of the Solution 14](#_Toc166179571)

[3.6 Another solution 15](#_Toc166179572)

[3.7 Conclusion 16](#_Toc166179573)

[Task 4 : The Four- Peg Tower of Hanoi 17](#_Toc166179574)

[4.1 Problem Description 17](#_Toc166179575)

[4.2 Detailed Assumptions 17](#_Toc166179576)

[4.3 Detailed Solution 18](#_Toc166179577)

[4.3.1 Description of Solution 18](#_Toc166179578)

[4.3.2 Pseudo Code 18](#_Toc166179579)

[4.4 Complexity Analysis for the Algorithm 20](#_Toc166179580)

[4.5 Sample Output of the Solution 21](#_Toc166179581)

[4.6 Comparison with Another Algorithm 23](#_Toc166179582)

[4.7 Conclusion 24](#_Toc166179583)

[4.8 References 24](#_Toc166179584)

[Task 5 : Jumping Coins 25](#_Toc166179585)

[5.1 Problem Description 25](#_Toc166179586)

[5.2 Detailed Assumptions 25](#_Toc166179587)

[5.3 Detailed Solution 26](#_Toc166179588)

[5.3.1 Description of Solution 26](#_Toc166179589)

[5.3.2 Greedy Technique 26](#_Toc166179590)

[5.3.3 Brute-Force Technique 28](#_Toc166179591)

[5.4 Complexity Analysis for the Algorithm 32](#_Toc166179592)

[5.4.1 Greedy Technique 32](#_Toc166179593)

[5.4.1 Brute Force Technique 32](#_Toc166179594)

[5.5 Sample Output of the Solution 33](#_Toc166179595)

[5.6 Comparison Between the two techniques 35](#_Toc166179596)

[5.7 Conclusion 35](#_Toc166179597)

[Task 6 : Six Knights 36](#_Toc166179598)

[6.1 Problem Description 36](#_Toc166179599)

[6.2 Detailed Assumptions 36](#_Toc166179600)

[6.3 Detailed Solution 37](#_Toc166179601)

[6.3.1 Description of Solution 37](#_Toc166179602)

[6.3.2 Pseudo Code 37](#_Toc166179603)

[6.4 Complexity Analysis for the Algorithm 41](#_Toc166179604)

[6.5 Sample Output of the Solution 42](#_Toc166179605)

[6.6 Comparison with Another Algorithm 43](#_Toc166179606)

[6.7 Conclusion 43](#_Toc166179607)

[6.8 References 43](#_Toc166179608)

[Task 7 : Hitting a Moving Target 44](#_Toc166179609)

[7.1 Problem Description 44](#_Toc166179610)

[7.2 Detailed Assumptions 44](#_Toc166179611)

[7.3 Detailed Solution 45](#_Toc166179612)

[7.3.1 Description of Solution 45](#_Toc166179613)

[7.3.2 Pseudo Code 45](#_Toc166179614)

[7.4 Complexity Analysis for the Algorithm 47](#_Toc166179615)

[7.5 Sample Output of the Solution 47](#_Toc166179616)

[7.6 Comparison with Another Algorithm 47](#_Toc166179617)

[7.7 Conclusion 47](#_Toc166179618)

[Task 8 : Fake Box 48](#_Toc166179619)

[8.1 Problem Description 48](#_Toc166179620)

[8.2 Detailed Assumptions 48](#_Toc166179621)

[8.3 Detailed Solution 48](#_Toc166179622)

[8.3.1 Description of Solution 48](#_Toc166179623)

[8.3.2 Pseudo Code 49](#_Toc166179624)

[8.4 Complexity Analysis for the Algorithm 49](#_Toc166179625)

[8.5 Sample Output of the Solution 50](#_Toc166179626)

[8.6 Comparison with Another Algorithm 50](#_Toc166179627)

[8.7 Conclusion 50](#_Toc166179628)

# List Of Tables

[Table 1 6](#_Toc166179498)

[Table 2 10](#_Toc166179499)

[Table 3 16](#_Toc166179500)

[Table 4 23](#_Toc166179501)

[Table 5 43](#_Toc166179502)

[Table 6 47](#_Toc166179503)

[Table 7 50](#_Toc166179504)

# List Of Figures

[Figure 1 12](#_Toc166179523)

[Figure 2 15](#_Toc166179524)

[Figure 3 21](#_Toc166179525)

[Figure 4 21](#_Toc166179526)

[Figure 5 22](#_Toc166179527)

[Figure 6 36](#_Toc166179528)

# Task 1 : Tromino Tilings

## 1.1 Problem Description

Devise an algorithm for the following task: given a 2n × 2n (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.

## 1.2 Detailed Assumptions

1. The board is a square with dimensions 2n x 2n, where n > 1 (minimum size is 4x4).
2. There is exactly one missing square on the board.
3. The right tromino is an L-shaped tile formed by three squares.
4. There are three distinct colors available for the trominoes (Red, Green, Blue, for example).
5. The missing square can be located anywhere on the board.

## 1.3 Detailed Solution

### 1.3.1 Description of Solution

1. we always put tromino of value 1 in the center of board
2. we divide board into four quarters and if divided board is bigger than 2x2 board do first step then second step again else we do third step
3. we do this step if board == 2x2 board if this is the first time we access this quarter then we see all other trominoes around it so that we place the right tromino the we record it because all quarters of same type in other divided boards will have the same tromino

### 

### 1.3.2 Pseudo Code

function tromino(grid\_size, start\_x, start\_y, grid)

// Base Case: single cell

if grid\_size == 1 then

return

// Find the location of the missing square

missing\_x = null

missing\_y = null

for i = start\_x to start\_x + grid\_size - 1 do

for j = start\_y to start\_y + grid\_size - 1 do

if grid[i][j] != 0 and grid[i][j] != -1 then

missing\_x = i

missing\_y = j

break 2; // Exit both loops after finding the first missing square

endif

endfor

endfor

// Handle different cases based on missing square location

if grid\_size == 2 then

// Place tromino based on missing square quadrant

if missing\_x < start\_x + 1 and missing\_y < start\_y + 1 then

placeTromino(start\_x + 1, start\_y + 1, start\_x + 1, start\_y, start\_x, start\_y + 1, 1, grid)

else if missing\_x >= start\_x + 1 and missing\_y < start\_y + 1 then

placeTromino(start\_x + 1, start\_y + 1, start\_x + 1, start\_y, start\_x, start\_y - 1, 2, grid)

else if missing\_x < start\_x + 1 and missing\_y >= start\_y + 1 then

placeTromino(start\_x + 1, start\_y + 1, start\_x, start\_y + 1, start\_x - 1, start\_y + 1, 3, grid)

else

placeTromino(start\_x + 1, start\_y + 1, start\_x, start\_y, start\_x - 1, start\_y - 1, 4, grid)

endif

else

// Handle missing square in center quadrant

if missing\_x < start\_x + grid\_size / 2 and missing\_y < start\_y + grid\_size / 2 then

placeTromino(start\_x + grid\_size / 2, start\_y + grid\_size / 2 - 1, start\_x + grid\_size / 2, start\_y + grid\_size / 2, start\_x + grid\_size / 2 - 1, start\_y + grid\_size / 2, 0, grid)

else if ... (similar checks for other quadrants with missing center square)

endif

// Recursively call tromino for sub-grids

tromino(grid\_size / 2, start\_x, start\_y + grid\_size / 2, grid)

tromino(grid\_size / 2, start\_x, start\_y, grid)

tromino(grid\_size / 2, start\_x + grid\_size / 2, start\_y, grid)

tromino(grid\_size / 2, start\_x + grid\_size / 2, start\_y + grid\_size / 2, grid)

endif

end function

function placeTromino(x1, y1, x2, y2, x3, y3, quadrant, grid)

// Update neighboring cells based on tromino placement and quadrant

// ... (similar logic to original placeTrom function)

// Set the quadrant value for all three tromino squares in the grid

grid[x1][y1] = quadrant

grid[x2][y2] = quadrant

grid[x3][y3] = quadrant

end function

## 1.4 Complexity Analysis for the Algorithm

**Base Case:** the base case tromino(n = 1) takes constant time (O(1)).

**Recursive Calls:** The tromino function makes four recursive calls for sub-grids with size n/2. This seems similar to a logarithmic depth, but there's a crucial difference.

**placeTrom** **function:** While the number of iterations in placeTrom itself is constant, the key point is that placeTrom is called for every cell in the grid during the recursion. This happens because in each sub-grid, all the cells are checked for potential tromino placement based on the missing square's location.

**Combining Time:** The number of recursive calls grows with the grid size (n), leading to placeTrom being called for an increasing number of cells. This translates to the total work done growing proportionally to the square of the grid size (n^2).

**T(n) = 4T(n/2) + n**

**Overall Time Complexity:**

Using master theorem then **time complexity: O(n2)**

## 1.5 Sample Output of the Solution

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## 1.6 Comparison with Another Algorithm

Table 1

|  |  |  |
| --- | --- | --- |
|  | **Dynamic Programming** | **Divide & Conquer** |
| **Time Complexity** | O(n2) | O(n2) |
| **Space Complexity** | O(n) | O(n) |
| **Readability** | Complex and Difficult  And not easy to understand | Conceptually simple and straightforward |

## 1.7 Conclusion

By comparing the algorithms used to solve the problem we observe:

The dynamic programming approach is not any different compared to divide & conquer approach in time complexity and space complexity but when it comes to readability and implementation the divide & conquer is a lot easier compared to dynamic programming

# Task 2 : Knight Tour Closed

## 2.1 Problem 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.

## 2.2 Detailed Assumptions

1. Standard 8x8 chessboard is assumed.
2. Knight moves according to its L-shaped pattern.
3. Each square on the chessboard must be visited exactly once.
4. The tour must form a closed loop, returning to the starting square.
5. Backtracking is typically employed in the solution algorithm.
6. Some algorithms may introduce randomness for efficiency.
7. Heuristics are commonly used to guide the search process.
8. An optimal or any valid closed-circuit solution may be sought.

## 2.3 Detailed Solution

### 2.3.1 Description of Solution

Warnsdorff’s algorithm is a heuristic approach to solve the Knight's tour problem. It's based on the idea of always choosing the next move to be the one with the fewest onward moves (vertex with smallest degree). This heuristic helps to reduce the branching factor and improves the chances of finding a solution.

### 2.3.2 Pseudo Code

#### 2.3.2.1 High-Level

1. Initialize an empty chessboard.
2. Start from a given square.
3. Mark the starting square as visited.
4. Find the next valid move using Warnsdorff's heuristic:
   1. Consider all possible moves from the current square.
   2. Choose the move with the fewest accessible squares.
5. Update the current position and mark it as visited.
6. Repeat steps 4-5 until all squares are visited or no further moves are possible.
7. Check if the tour is closed by verifying if the knight can move back to the starting square.
8. Print the solution if a closed tour is found.

#### 2.3.2.1 In Details

A computer screen shot of text

Description automatically generated

## 2.4 Complexity Analysis for the Algorithm

Time complexity: O()

Space Complexity: O())

## 2.5 Sample Output of the Solution

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer screen

Description automatically generated

## 2.6 Comparison with Another Algorithm

Table 2

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Brute Force Approach** | **Warnsdorff's Heuristic** |
| **Advantages** | Guarantees to find a solution if it exists. | More efficient for larger chessboards. |
|  | Conceptually simple and straightforward | Often finds solutions faster. |
|  |  | Lower space complexity. |
| **Disadvantages** | Exponential time complexity: O(8^(N^2)). | Not guaranteed to find solution always. |
|  | Inefficient for large chessboards. | May fail for certain configurations. |
|  |  | Possibility of not finding a solution. |
| **Complexity** | Time: O(8^(N^2)), Space: O(N^2). | Time: O(N^2), Space: O(N^2). |

## 2.7 Conclusion

In conclusion, solving the Knight's Tour problem can be approached in two main ways: brute force and using Warnsdorff's heuristic.

The brute force method checks every possible move until it finds a closed tour or exhausts all options. It's easy to understand but can be very slow, especially for larger chessboards.

On the other hand, Warnsdorff's heuristic method uses a simple rule to guide the knight's movements, making it faster and more efficient. While it may not always find the best solution, it strikes a good balance between speed and accuracy.

Overall, for practical purposes, Warnsdorff's heuristic approach is often the better choice due to its speed and reliability, even though it might not always find the absolute best solution.

# Task 3 : Security Switches

## 3.1 Problem Description

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 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.

## 3.2 Comment

It is an optimization problem as we want the minimum number of moves using dynamic programming, we will go with a forward approach backward reasoning and as there are lots of possibilities it will be hard to draw the multistage graph so we will settle only with the recurrence relation.

## 3.3 Detailed Solution

### 3.3.1 Description of Solution

First analyze the problem and understand the recursion in it as it has overlapping problems, we can see that to turn off a switch we need its immediate right to be on and all the others to the right off

A diagram of a graph

Description automatically generated with medium confidence

Figure 1

We can see the pattern to solve a switch (i) we need to solve for the switches i+2 and the rest on the right to be off and then toggle switch(i) getting 0100…. To get it to be 011111… we need M(n-2) moves again leading to 2 M(n-2) till now then we have to solve M(N-1) of the original problem +1 for the first switch that is a base case leading to the recurrence relation below.

**Recurrence Relation:**

M(n) = M(n − 2) + 1 + M(n − 2) + M(n − 1)

Or

M(n) = M(n − 1) + 2M(n − 2) + 1 for n ≥ 3, M(1) = 1, M(2) = 2

* M(1) =1 and M(2) =2 are the base cases
* M(n − 1): Number of moves needed to solve for (n - 1) switches.
* M(n − 2): Number of moves needed to solve for (n - 2) switches, and we need to double it because we toggle the switches twice in that case.
* 1: Additional move needed to toggle the first switch

### 3.3.2 Pseudo Code

Function min\_moves(n):

If n is 1:

Return 1

If n is 2:

Return 2

Initialize an array dp of size n+1 with zeros

Set dp[1] to 1

Set dp[2] to 2

Print "starting from: 1 last steps: " + dp[1]

Print "starting from: 2 last steps: " + dp[2]

For i from 3 to n:

Set dp[i] to dp[i-1] + 2\*dp[i-2] + 1

Print "starting from: " + i + " last steps: " + dp[i]

Return dp[n]

End Function

## 3.4 Complexity Analysis for the Algorithm

For this code the time complexity will be O(n) using the recurrence relation

The puzzle can be solved in the minimum of switch toggles.

## 3.5 Sample Output of the Solution

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

## 3.6 Another solution

Another Solution This is another solution that solves the puzzle using decease-and-conquer algorithm. The solution was taken from the book: “Algorithmic Puzzles by Anany Levitin”. 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. To help ourselves with solving the general instance of the puzzle, let us start by solving its four smallest instances. Consider now the general instance of the puzzle represented by the bit string of n 1’s: 111. . . 1. Before we can turn off the first (leftmost) switch, the switches should be in the state 110. . . 0. Hence, to begin with, we need to turn off the last n−2 switches and do this in the minimum number of moves if we want to have an optimal algorithm. In other words, we should first solve the same problem as given for the last n−2 switches. This can be done recursively with the n = 1 and n = 2 instances solved directly. After that, we can toggle the first switch to get 010. . . 0. Now, before we can toggle the second switch we will need to pass through the state with all the switches following it “on,” which can be easily proved by mathematical induction. Getting all the switches from the third to the last one “on” can be achieved by reversing the optimal sequence of moves made previously to toggle the last n − 2 switches from the “on” position to the “off” position. This will give us 011. . . 1. Ignoring the first 0, we now have the n − 1 instance of the original puzzle, which can be solved recursively.

A group of numbers and equations

Description automatically generated

Figure 2

## 3.7 Conclusion

The approach described in the book "Algorithmic Puzzles by Anany Levitin" uses a decrease-and conquer strategy instead of dynamic programming.

Table 3

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Dynamic Programming** | **Decrease-and-Conquer** |
| **Base Cases** | The base case is when the subproblem size is 1, and the solution for that subproblem is already known | The base cases are the smallest instances of the problem, which are solved directly |
| **Solution Construction** | The solution is constructed by recursively solving subproblems and combining the results | The solution is constructed by solving the smallest instances of the problem first, then using those solutions to solve larger instances |

# Task 4 : The Four- Peg Tower of Hanoi

## 4.1 Problem 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. It’s required to use divide and conquer algorithm to transfer all the disks to another peg by a sequence of moves.

The constraints are that only one disk can be moved at a time, and it is forbidden to place a larger disk on top of a smaller one.

Also it’s required to use dynamic programming algorithm to transfer all the disks to another peg in 33 moves

## 4.2 Detailed Assumptions

1. The disks are labeled with consecutive integers starting from 1. Each disk is represented by an object of Disk class, which has a disk\_num attribute.
2. The moves variable is used to keep track of the total number of moves performed during solving the Tower of Hanoi Problem.
3. The Tower of Hanoi problem is being solved optimally, it aims to minimize the number of moves required to transfer the disks from the starting peg to the end peg.
4. The line that assigns k to the optimal value of k that is resulted in the function of getOptimalDiskNum is commented in the code if divide and conquer method is used an the value of k will be assigned to n/2 where n is the number of disks
5. The line that assigns k to n/2 is commented in case of using dynamic programming and the value of k will be assigned to the value resulted in OptimalK array in the getOptimalDiskNum function

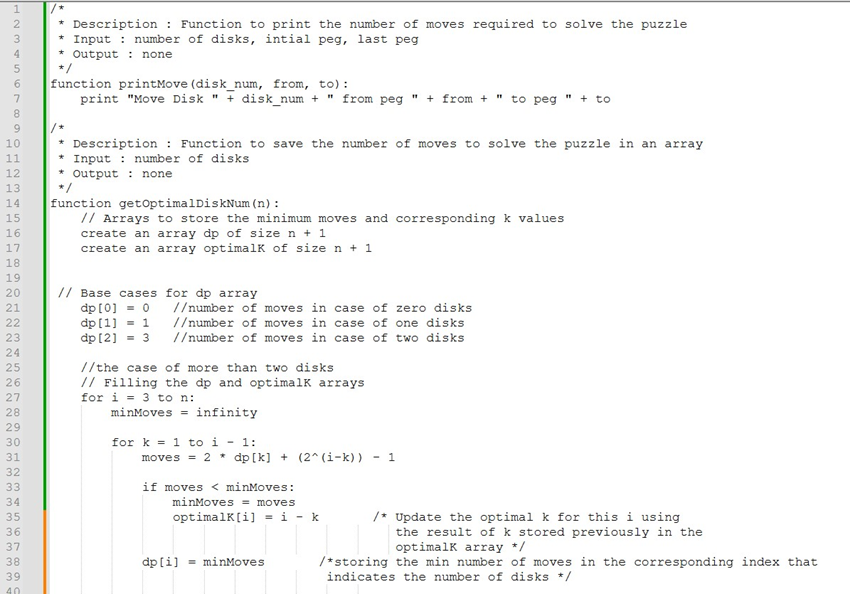
## 4.3 Detailed Solution

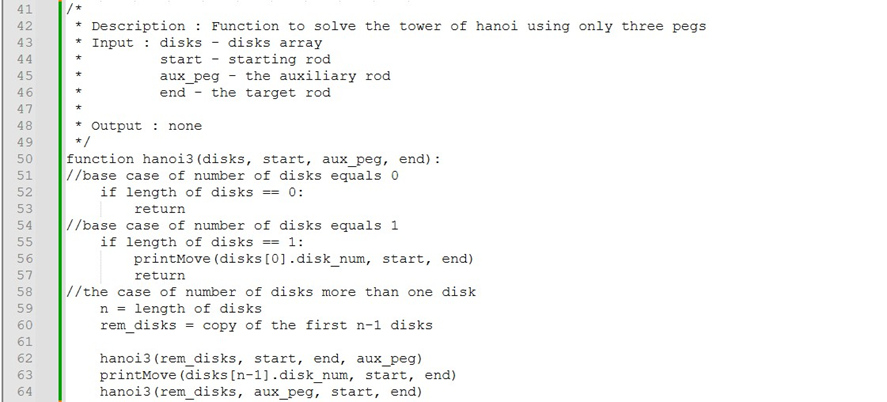
### 4.3.1 Description of Solution

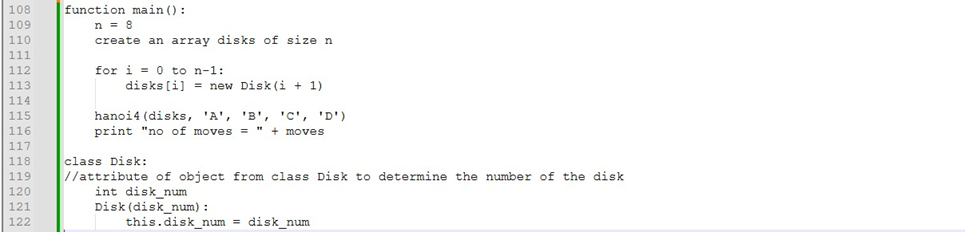
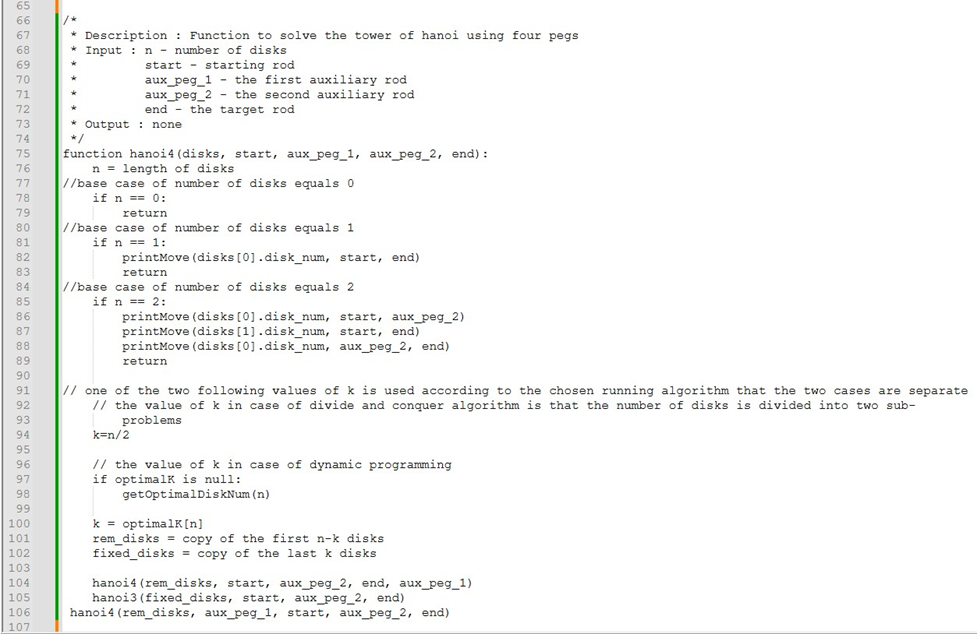
1. Initialize an array dp of size n+1 to store the number of moves required to transfer the disks from the starting peg to the end peg
2. Initialize an array optimalK of size n+1 to store to optimal value of k that will be used to divide the disks in dynamics programming technique while in divide and conquer the disks will be divided into halves
3. The code keep divide the disks into sub-problems until reaches the base case to transfer the disks to another peg

### 4.3.2 Pseudo Code

Both divide and conquer algorithm and dynamic programming algorithm are implemented in the following pseudocode







## 4.4 Complexity Analysis for the Algorithm

1. getOptimalDiskNum function:

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

1. hanoi3 function:

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

1. hanoi4 function:

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

1. main function:

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

The resulted Time Complexity: O(2^n)

The resulted Space Complexity: O(n)

## 4.5 Sample Output of the Solution

- **Case of five disks**

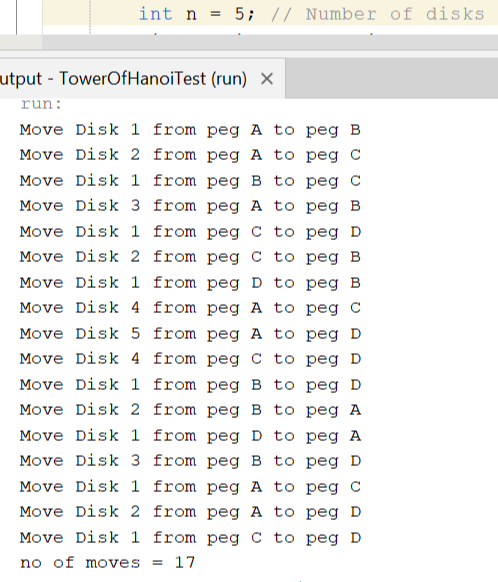
The five disks are divided into two sub-problems of size three and two disks based of the value of optimal k

Figure 3

The upper three disks are ordered using the four rods in the first call of function hanoi4

The lower two disks are ordered using the remaining three rods in the call of function hanoi3

The upper three disks move back again to be ordered above the bottom two disks using the four rods in the second call of function hanoi4

The number of moves is calculated based on dynamic programming that stores the result of the base case which are 0,1, and 2 disks in

0, 1, and 3 moves then storing the number of moves in

the dp array to be used in the next number of disks

- **Case of six disks**

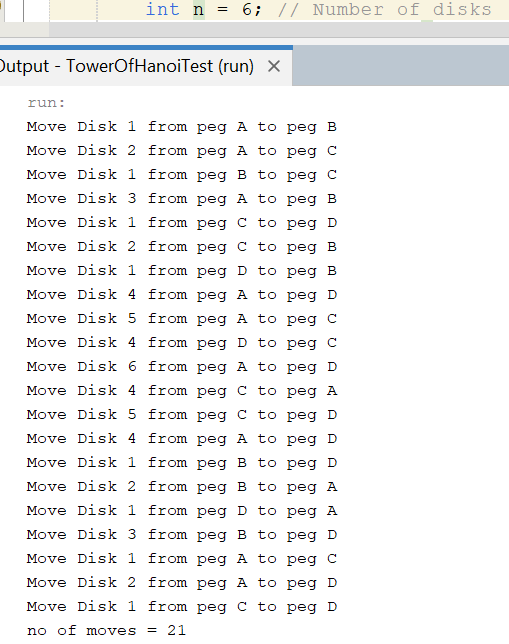
The six disks are divided into two sub-problems of equal size of three disks each based of the value of optimal k

Figure 4

The upper three disks are ordered using the four rods in the first call of function hanoi4

The lower three disks are ordered using the remaining three rods in the call of function hanoi3

The upper three disks move back again to be ordered above the bottom three disks using the four rods in the second call of function hanoi4

The number of moves is calculated based on

dynamic programming that stores the result of the base cases which are 0,1, and 2 disks in 0, 1, and 3 moves then storing the number of moves in the dp array to be used in the next number of disks

A screenshot of a computer

Description automatically generated- **Case of eight disks**

Figure 5

The six disks are divided into two sub-problems of equal size of four disks each based of the value of optimal k

In the first call of hanoi4

The upper four disks are ordered by recursively calling hanoi4 and divide the four disks into two sub-problems of the same size of two disks each based on the value of optimal k

The upper two disks (disk 1 and 2) ordered on peg C using the four rods in the call of function hanoi4 while the lower two disks (disk 3 and 4) are ordered on peg B using the three rods in the call of function hanoi3 then the upper two disks are ordered above the lower two disks on peg B using the four rods in the call of function hanoi4

The lower four disks are ordered by calling hanoi3 and recursively reduce the problem by one into sub-problem until it reaches the base case which is two disks (disk 5 an disk 6) then order disk 7 then order disk 8 on peg D

In the second call of hanoi4

The upper four disks are ordered by recursively calling hanoi4 and again divide the four disks into two sub-problems of the same size of two disks each based on the value of optimal k

The upper two disks (disk 1 and 2) ordered on peg A using the four rods in the call of function hanoi4 while the lower two disks (disk 3 and 4) are ordered on peg D using the three rods in the call of function hanoi3 then the upper two disks are ordered above the lower two disks on peg D using the four rods in the call of function hanoi4.

## 4.6 Comparison with Another Algorithm

The solution using decrease and conquer

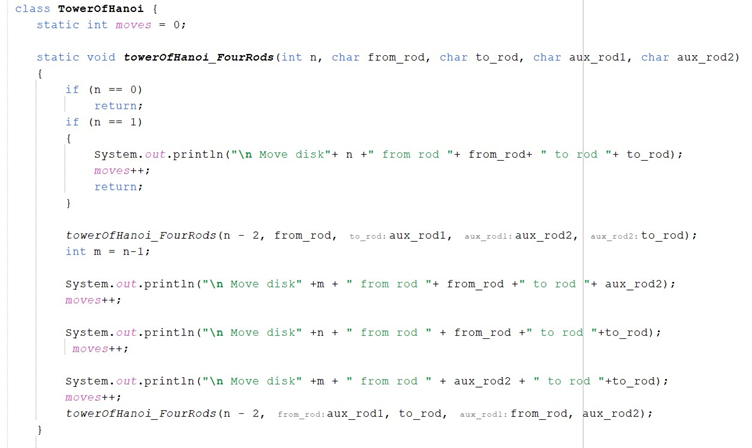


Table 4

|  |  |  |
| --- | --- | --- |
|  | **Divide and Conquer** | **Decrease and Conquer** |
| **Advantages** | Efficient for large problem size in case the number of disks increase | Simplicity as it reduces the problem by one until reaches the base case  Lower Memory requirements as it often operate on the problem in place |
| **Disadvantages** | Overhead of combining sub-problems  Increase memory usage | May not be efficient for large problems |
| **Number of moves** | Transfers 8 disks in 33 moves | Transfers 8 disks in 41 moves |

## 4.7 Conclusion

In Conclusion, using Divide and Conquer with the Dynamic Programming approach to divide the problem to two sub-problems and calculate the minimum number of moves required for different number of disks and storing them in the memory leads to optimizing the solution that offers a modified approach to solving the Tower of Hanoi Problem using four pegs, allowing more efficient solution compared to the traditional approach.

## 4.8 References

The Four- Peg Tower of Hanoi Puzzle by Richard Johnsonbaugh.

# Task 5 : Jumping Coins

## 5.1 Problem 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 which the problem has a solution and design an algorithm that solves it in the minimum number of moves for those n’s. Design greedy algorithm to find minimum number of moves.”

## 5.2 Detailed Assumptions

We assumed that in both solutions either the greedy or the brute force, the algorithm will start from the very first coin in the left most of the row and work its way to the right direction. As it’s arbitrary to start from any coin and work its way in any direction and there’s no constraints or cases that will go against this assumption, so there’s no problem to assume that we will start from that coin and in that direction.

## 5.3 Detailed Solution

### 5.3.1 Description of Solution

If we try by hand to solve this problem, we will find that some values of n work fine and other doesn’t, some of the values that works are 4, 12 and 28. We need to observe the pattern or the mathematical formula, and after that use it to deduce the next values, then finally test those values to see if the pattern or the mathematical formula was right or wrong. After looking for some formulas and pattern, there’s only one simple pattern or formula that satisfies the problem constraints, and it is as follows:

If we assume that the value of n is equal to the sum of two big piles of mathematically calculated number of fours, then you could put the 4, 12 and 28 as follows: 4 = (0) + (4), 12 = (4) + (4+4), 28 = (4+4+4) + (4+4+4+4) then you we can deduce a mathematical formula based on that, and it is as follows:

***n = ((2^(i)-1) + (2^ (i )) ) \* 4 = (2^(i+3)) - 4 where i is a non-negative integer.***

Then for i=0 -> n = 4, for i=1 -> n=12, for i=2 -> n=28, for i=3 -> n=60, etc.…

If you try n=60, it will work. So, the deduced formula is right, hence that was one of the requirements, what are the values of n that satisfies the problem constraints? **(2^(i+3)) – 4 where i is a non-negative integer.**

### 5.3.2 Greedy Technique

#### 5.3.2.1 Key Idea

Simply start from the left most of the row, set the number of moves to zero and start moving the first coin according to the constraints of the problem to form a pair in the left direction, then next ***choose the nearest coin to the empty space that it was left behind (greedy step)***, increment the number of moves then repeat the steps again until you form n/2 coin pairs in just ***n/2 moves which is logically the minimum amount of moves required to form n/2 coin pairs.***

#### 5.3.2.2 Pseudo Code

##### 5.3.2.2.1 High-Level

1. Set the current move to 0.
2. Start from the leftmost coin.
3. Iterate through the row until you reach the coin before last.

3.1 For each single coin you find while iterating, start to the column to the left of it.

* + 1. Increase the current move by one.
    2. Set the count of jumped over coins for this specific coin to 0.
    3. If the current jumped over coins count equals the current move, then put the single coin on the current column to form a pair, then break out of this loop to go to the next nearest single coin to the previous one that we selected.
    4. Else if the current jumped over coins count doesn’t equal the current move, then increase the jumped over coins by the number of coins in the current column and go to the next left column.

##### 5.3.2.2.2 In Details

numberOfMoves <- 0

For i = 0 to n-1 {

// skip empty spaces or coin pairs until you find the nearest coin (greedy step)

if(rowOfCoins[i] != 1)

continue

numberOfMoves <- numberOfMoves + 1

jumpedOverCoins <- 0

// making a coin move to form a pair of coins

For j=i+1 to n-1 {

if (jumpedOverCoins == numberOfMoves) {

rowOfCoins[i] <- 0;

rowOfCoins[j] <- rowOfCoins[j] + 1

break

}

// counting the number of jumped over coins

jumpedOverCoins <- jumpedOverCoins + rowOfCoins[j];

}

}

### 5.3.3 Brute-Force Technique

#### 5.3.3.1 Key Idea

Again simply start from the left most of the row, set the number of moves and start moving the first coin according to the constraints of the problem to form a pair in the right direction of the row, and if no place was found satisfying the constraints to form a coin pair then search in the left direction for a place, if also no place was found satisfying the constraints then we go back one step to the previous pair that was formed, unform it then try to find another different place for that previous singe coin to form a pair then go to the next single coin that we couldn’t find a place for it to form a pair in the first place, then we try to find a place for it, eventually we will repeat the steps and make our way brute forcing it with those steps until we make n/2 coin pairs in just n/2 moves which is logically the minimum amount of moves required to form n/2 coin pairs.

#### 5.3.3.2 Pseudo Code

##### 5.3.3.2.1 High-Level

1. Start by calling the function to run at the left most position and operate in the right direction of the row.
2. Check if the position passed is valid, if not then return false.
3. Set the number of moves to 1 and make a flag that determines if we are done or not and set it to false.
4. Set the count of jumped over coins to 0.
5. Start iterating from the column to the to the left or right of the passed position as it was specified in the function call, and do the following in each iteration:
   1. If the current jumped over coins count equals the current move, then put the single coin at the specified position on the current column to form a coin pair, increase the current move number by one, set the flag to true, then finally break out of this loop to go to the next nearest single coin to the previous one that we selected.
   2. Else if the current jumped over coins are greater than the current move then breaks out of the loop.
   3. Else if the current jumped over coins count are less than the current move, then increase the jumped over coins by the number of coins in the current column and go to the next left or right column as the direction was specified at the beginning of the loop in the function call.
6. Check if there’s still any single coins left and if so, set the flag to false.
7. If the flag is true, then return true.
8. Determine the next position of the next single coin that will be moved to form a coin pair.
9. Call the function again with the new position calculated in the right direction (go to step 2) and set the returned value out of the function to the flag.
10. If the flag is true, then return true.
11. Call the function again with the new position calculated in the left direction (go to step 2) and set the returned value out of the function to the flag.
12. Return the flag value.

##### 5.3.3.2.2 In Details

algorithmUsingBruteForce (position, currentMoveNumber, direction) {

if ((position < 0) || (position > n))

return false

numberOfMoves 🡨 currentMoveNumber

jumpedOverCoins 🡨 0

doneFlag 🡨 false

// check in the right direction the possibility to form a pair

if (direction == "right")) {

for i 🡨 position+1 to n-1 {

// move the coin to the column if the condition is satisfied

// or break if exceeded

If (jumpedOverCoins >= numberOfMoves) {

if (jumpedOverCoins == numberOfMoves) {

rowOfCoins[position] 🡨 0

rowOfCoins[i] 🡨 rowOfCoins[i] + 1

currentMoveNumber 🡨 currentMoveNumber + 1

doneFlag 🡨 true

}

break

}

// or jump over the column and count the jumped over coins

else if (jumpedOverCoins < numberOfMoves)

jumpedOverCoins 🡨 jumpedOverCoins + rowOfCoins[i]

}

}

// check in the left direction the possibility to form a pair

else if (direction == "left")) {

for i 🡨 position-1 to 0 {

// move the coin to the column if the condition is satisfied

// or break if exceeded

If (jumpedOverCoins >= numberOfMoves) {

if (jumpedOverCoins == numberOfMoves) {

rowOfCoins[position] 🡨 0

rowOfCoins[i] 🡨 rowOfCoins[i] + 1

currentMoveNumber 🡨 currentMoveNumber + 1

doneFlag 🡨 true

}

break

}

// or jump over the column and count the jumped over coins

else if (jumpedOverCoins < numberOfMoves)

jumpedOverCoins 🡨 jumpedOverCoins + rowOfCoins[i]

}

}

// check if there're still single coins

For i in rowOfCoins {

if (i == 1) {

doneFlag 🡨 false

break

}

}

// check if all are the coins are pairs

// if so, then we are done

if (doneFlag == true)

return true

// next single coin to be moved to form a pair

int nextPosition 🡨 position;

for i 🡨position+1 to n-1{

if (rowOfCoins[i] == 1){

nextPosition 🡨 i

break

}

}

doneFlag 🡨 algorithmUsingBruteForce (nextPosition, currentMoveNumber, "right")

// still, there're single coins?

If (doneFlag == true)

return true

doneFlag 🡨 algorithmUsingBruteForce (nextPosition, currentMoveNumber, "left")

return doneFlag

}

## 5.4 Complexity Analysis for the Algorithm

### 5.4.1 Greedy Technique

In general, you loop over n in the left direction skipping every empty space and every coin pair that was formed, until you find a single coin, then you start to iterate in the same left direction until you find the right spot to put that single coin so you form a coin pair, then repeat.

For the first coin, you will jump over one coin, for the second you will jump over two coins, etc.…

As an approximation we could say that the whole time is upper bounded by:

***[1 + 2 + 3 + 4 + … + n/2] + n/2 where each number represents a coin that we iterated that number of times to find the right spot for it, except the last (n/2) it represents the skipped empty spaces and coin pairs that we looped through, but we didn’t select any of them.***

***So, [1 + 2 + 3 + 4 + … + n/2] + n/2 = ((n/2) \* (n/2 + 1))/2 + (n/2) = ((n^2) / 8) + (3n/4) which is O(n^2) in all cases.***

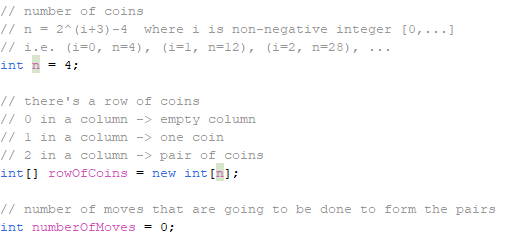
### 5.4.1 Brute Force Technique

It is expected that the brute force technique will have a very high running time, higher than the greedy technique by a lot as it goes through a lot of paths most of them are wrong until it reaches the right path that leads to the solution, but the greedy chooses the greediest path from the start which is likely to be the solution or an approximate to it.

But in our case, and due to the assumption that we start in both algorithms from the left most coin and work our way in the right direction, the brute force technique will reach the right path from the first try as if it was the greedy technique, and then it will stop.

***So, the brute force technique will be upper bounded by approximately the same running time as the greedy technique which is O(n^2) and that is in all cases.***

## 5.5 Sample Output of the Solution



A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A white sheet with numbers and lines

Description automatically generated

A white paper with many lines

Description automatically generated with medium confidence

A white background with blue and orange lines

Description automatically generated

## 5.6 Comparison Between the two techniques

As the two techniques has two different methods of finding the solution. But like discussed above, the two techniques will result in two different algorithms each will solve the problem and find the solution in approximately similar running time in all cases due to the problem nature and the assumption and implementations that were made.

## 5.7 Conclusion

The values of n that satisfies the problem constraints are equal to **(2^(i+3)) – 4, where i is a non-negative integer** and two algorithms were designed one using the greedy technique and the other one using the brute-force technique to solve the problem.

There might be in some cases, two different techniques that could produce two different algorithms each one of them has its own way of finding the solution to the problem but due to the problem nature, implementation, assumptions and so on, they could end up with the same time complexity, like in our case in the discussed problem in task 5.

In our case we saw the problem solved by two different techniques the first one is the main one that was required which is the greedy technique which produced a solution that was upper-bounded by ***O(n^2)*** in all cases, and the second technique was the brute-force technique which could have led to a much higher running time but as we said due to the assumptions, implementation and the problem nature that helped in our case, the algorithm based on the brute-force technique produced a solution that was upper-bounded by the same running time as the algorithm based on the greedy technique which is ***O(n^2).***

# Task 6 : Six Knights

## 6.1 Problem Description

There are six knights on a 3 × 4 chessboard: the three white knights are at the bottom row, and the three black knights are at the top row.

Design an iterative improvement algorithm to exchange the knights to get the position shown on the right of the figure in the minimum number of knights moves, not allowing more than one knight on a square at any time.



Figure 6

## 6.2 Detailed Assumptions

1. A chessboard of 4X3 is assumed.

2. Knights must follow an L-shaped pattern.

3. A knight can’t go outside the board boundaries.

4. Each Knight placement in a board is a State.

5. Best first search is implemented using heuristics.

6. The heuristic is implemented using Manhattan distance.

7. Once in a path you can’t visit the same state twice.

## 6.3 Detailed Solution

### 6.3.1 Description of Solution

1. An initial state is passed in the code.

2. The code starts by Generating moves for knights starting with the white knights.

3. Each state is stored inside a path and each path is stored inside priority queue that is sorted based on heuristic value.

4. The heuristic value of the path is the heuristic value of the last state of this path.

5. The top of the queue (the one with the lowest heuristic value) is hypothetically the closest board arrangement to our solution.

6. If our board arrangement is found its Heuristic will equal -1

### 6.3.2 Pseudo Code

FUNCTION calculateHeuristic(board)

// Inputs: Board is the 2D vector representing the game board

// Output: Returns the heuristic value calculated based on the board state

heuristic <= 0

// Define goal positions for white and black knights

whiteGoalPositions <= {{0, 0}, {0, 1}, {0, 2}}

blackGoalPositions <= {{3, 0}, {3, 1}, {3, 2}}

FOR i FROM 0 TO ROWS - 1 DO

FOR j FROM 0 TO COLS - 1 DO

IF board[i][j] == WHITE THEN

minDist <= INT\_MAX

FOR EACH goalPos IN whiteGoalPositions DO

dist <= ABS(i - goalPos.first) + ABS(j - goalPos.second)

minDist <= MIN(minDist, dist)

END FOR

heuristic += minDist

ELSE IF board[i][j] == BLACK THEN

minDist <= INT\_MAX

FOR EACH goalPos IN blackGoalPositions DO

dist <= ABS(i - goalPos.first) + ABS(j - goalPos.second)

minDist <= MIN(minDist, dist)

END FOR

heuristic += minDist

END IF

END FOR

END FOR

IF isGoalState(board) THEN

RETURN -1

END IF

RETURN heuristic / 3.0 + 3

END FUNCTION

FUNCTION isGoalState(board)

// Inputs: board is the 2D vector representing the game board

// Output: Returns boolean true if the board represents a goal state

RETURN (board[0][0] == WHITE AND board[0][1] == WHITE AND board[0][2] == WHITE) AND (board[3][0] == BLACK AND board[3][1] == BLACK AND board[3][2] == BLACK)

END FUNCTION

FUNCTION getKnightMoves(x, y, board)

// Inputs: x is the X coordinate of the knight, y is the Y coordinate of the knight, board is the 2D vector representing the game board

// Output: Returns a list of valid knight moves from the given position

moves <= []

directions <= {{1, 2}, {-2, 1}, {-2, -1}, {-1, -2}, {1, -2}, {2, -1}, {2, 1}, {-1, 2}}

FOR EACH dir IN directions DO

newX <= x + dir.first

newY <= y + dir.second

IF isEmptyAndWithinLimits(board, newX, newY) THEN

moves.push({newX, newY})

END IF

END FOR

RETURN moves

END FUNCTION

FUNCTION generateNextState(currentState, pq, currentPath)

// Inputs: currentState is the current state of the game, pq is the priority queue of paths, currentPath is the current path

// Output: Generates and pushes next path to the priority queue

FOR i FROM 0 TO ROWS - 1 DO

FOR j FROM 0 TO COLS - 1 DO

IF currentState.board[i][j] != 0 THEN

moves <= getKnightMoves(i, j, currentState.board)

FOR EACH move IN moves DO

nextBoard <= COPY currentState.board

SWAP nextBoard[i][j] AND nextBoard[move.first][move.second]

nextHeuristic <= calculateHeuristic(nextBoard)

nextState <= State(nextBoard, nextHeuristic)

IF NOT currentPath.hasDupeStates(nextState) THEN

currentPath.push(nextState)

pq.push(currentPath)

END IF

END FOR

END IF

END FOR

END FOR

END FUNCTION

// Performs Best-First Search to find the solution

FUNCTION bestFirstSearch(initialState)

// Inputs: initialState is the initial state of the game

// Output: Prints Path leading to solution if found

pq <= Priority Queue of Paths

initialHeuristic <= calculateHeuristic(initialState)

beginningState <= State(initialState, initialHeuristic)

initialPath <= Path(beginningState)

pq.push(initialPath)

WHILE NOT pq.empty() DO

currentPath <= pq.top()

currentState <= currentPath.path[currentPath.path.size() - 1]

pq.pop()

IF isGoalState(currentState.board) OR currentState.heuristic == -1 THEN

PRINT "\n\nSolution found:"

printBoard(currentState.board)

PRINT "Path: "

currentPath.printPath()

PRINT "\nTotal moves: " + currentPath.length() - 1

PRINT "\nTotal paths: " + pq.size()

RETURN

END IF

IF currentPath.length() <= MAX\_MOVES THEN

generateNextState(currentState, pq, currentPath)

END IF

END WHILE

PRINT "No solution found."

END FUNCTION

FUNCTION withinLimits(x, y)

// Inputs: x is the X coordinate of the knight, y is the Y coordinate of the knight

// Output: Returns boolean true if the coordinates are within the board limits

RETURN (x >= 0 AND y >= 0) AND (x < ROWS AND y < COLS)

END FUNCTION

FUNCTION isEmptyAndWithinLimits(board, x, y)

// Inputs: board is the 2D vector representing the game board, x is the X coordinate of the knight, y is the Y coordinate of the knight

// Output: Returns boolean true if the knight movement is within the board and the target square is empty

RETURN withinLimits(x, y) AND board[x][y] == 0

END FUNCTION

## 6.4 Complexity Analysis for the Algorithm

Time Complexity: O(N \* Log(N)).

Space complexity: O(N).

Where N is the number of paths in priority queue.

## 6.5 Sample Output of the Solution



Each state Generated in turn generates more states each chain of states is a path so this code starts with initial state generates a new state and adds it to this state and it’s a path so this path is put into Priority queue where it’s sorted based on Heuristic values of each state each new iteration it takes top path of queue and then it generates new states for it and so on till we reach goal.

## 6.6 Comparison with Another Algorithm

Table 5

|  |  |  |
| --- | --- | --- |
|  | **Brute force Approach** | **Best first search** |
| **Advantages** | Guarantees to find a solution if one exists | A faster approach then brute force and can work with any size of chessboard |
| **Disadvantages** | A huge time complexity | Not guaranteed to find a solution if heuristic is poorly chosen |
| **Time Complexity** | 8^N where N is the number of knights. | O(N\*Log(N)) where N is the number of paths in Priority queue |

## 6.7 Conclusion

Solving the Six knights problem requires one of 2 approaches Brute force, Searching algorithms (best first, Hill climbing, A\*).

The brute force method Generates all possible moves combinations till it reaches the goal, but can yield very high time complexity, and isn’t feasible for bigger chessboards.

Best first algorithm is better than brute force if heuristic calculations are good and can reach solution a lot faster than brute force approach, this approach is an approximation so it might not get the best path each time and it might encounter convergence problems.

## 6.8 References

- [Best first search](https://www.geeksforgeeks.org/best-first-search-informed-search/)

- [Wiki](https://proofwiki.org/wiki/Guarini%27s_Problem)

- [Puzzle stack exchange](https://puzzling.stackexchange.com/questions/25287/switch-the-knights/25301#25301)

- [Princeton university](https://assets.press.princeton.edu/chapters/s7714.pdf) pages 3,4 and 5.

- [Carnegie Mellon University.](https://www.cs.cmu.edu/~./awm/tutorials/hillclimb02.pdf)

# Task 7 : Hitting a Moving Target

## 7.1 Problem 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

## 7.2 Detailed Assumptions

**Target Movement**

* The target moves between adjacent spots in a straight line.
* There's no limit on the number of times the target can move.

**Shooter**

* The shooter can take one shot at a time, aiming at any of the spots.
* The shot is always successful if aimed at the spot where the target currently is.

**Other Assumptions**

* The number of spots (n) is greater than or equal to 1 (there must be at least one spot).

## 7.3 Detailed Solution

### 7.3.1 Description of Solution

We stimulate the target movements and we store the bullets in each hiding spot Then move forward in our hitting spot until we hit the spot n-1 then we hit the way back until we hit the spot 2 with this we guarantee hitting the target

### 7.3.2 Pseudo Code

Function getspot(i, j, n):

If i is less than 1:

Return j

Else if j is greater than n:

Return i

Else:

Generate a random integer between 0 and 999 (inclusive)

If the random integer is greater than or equal to 499:

Return i

Else:

Return j

End If

End Function

Function hittingAMovingTarget(n, currentHidingSpot):

Initialize an array dp of size n+1 with zeros

Initialize variables:

k = 1

count = n-1

shotspot = 2

Loop i from 0 to n+1:

If count is 1:

Increment dp[shotspot] by 1

Set k to -1

Reset count to n-1

Else:

Update dp[shotspot] as max(dp[shotspot-1]+1, dp[shotspot+1]+1)

Print "shot = " + shotspot + ", hiding spot = " + currentHidingSpot

If shotspot equals currentHidingSpot:

Break out of the loop

Decrement count

Update currentHidingSpot using getspot(currentHidingSpot-1, currentHidingSpot+1, n)

If (shotspot is not 2 and shotspot is not n-1) or count is not 1:

Increment shotspot by k

Return dp[currentHidingSpot]

End Function

## 7.4 Complexity Analysis for the Algorithm

Adding the time complexities of initialization and filling the DP table, we get **O(n) + O(n) = O(2n),** which can be simplified to **O(n).**

## 7.5 Sample Output of the Solution

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

## 7.6 Comparison with Another Algorithm

Table 6

|  |  |  |
| --- | --- | --- |
|  | **Dynamic Programming** | **Greedy** |
| **Time Complexity** | O(n) | O(n) |
| **Space Complexity** | O(n) | O(1) |
| **Optimum Solution** | Always minimum number of moves to hit target | A random number of moves not always minimum |

## 7.7 Conclusion

We use two methods to solve the problem

The first method is dynamic programming, we can say that it is an efficient method in that it always out the minimum number of moves to hit targe but it is has a high time complexity and space complexity.

The second method is greedy algorithm it’s advantage is it’s time complexity and space complexity

But it doesn't give a very efficient output

# Task 8 : Fake Box

## 8.1 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.

## 8.2 Detailed Assumptions

1. If we assume that each real box weigh [ a kilogram ] , then fake box weigh [ a-1 kilograms ]

2. Then each real metal piece in real boxes weight

3. Then each real metal piece in real boxes weight

## 8.3 Detailed Solution

### 8.3.1 Description of Solution

1. we give each box a unique number from 1 to 50

2. we take a number of metal pieces from each box depending on box's number

Example: 1 piece of box 1 and 2 pieces of box 2 and so on until we take 50 pieces of box 50

3. we weigh the gathered pieces and store the result

4. by computing 50 probabilities of expected results and storing them

5. comparing the result the digital scale with computed results, we find the fake box

### 8.3.2 Pseudo Code

findFakeBox(array boxes,float digitalScaleReading ,double realMetalPieceWeigt, double fakeMetalPieceWeigt)

// boxes: array of number of pieces taken from each box

float sum = 0;

arrayOfProbablities[50]

for i <- 1 to 50

sum = 0;

for j <- 0 to 49

if j+1 == i

sum += boxes[j] \* fakeMetalPieceWeigt;

else

sum += boxes[j] \* realMetalPieceWeigt;

endfor

arrayOfProbablities[i-1] = sum;

endfor

for i <- 0 to 49

if digitalScaleReading == arrayOfProbablities[i]

return i+1;

endfor

## 8.4 Complexity Analysis for the Algorithm

**T(n) = Σ*1*≤*i*≤*50 (*Σ*0*≤*j*≤*49 1*) *+* Σ*0*≤*i*≤*49 = O(1)***

## 8.5 Sample Output of the Solution

A black background with white text

Description automatically generated

A black screen with white text

Description automatically generated

## 8.6 Comparison with Another Algorithm

Table 7

|  |  |  |
| --- | --- | --- |
|  | **Brute Force** | **Divide & Conquer** |
| **Time Complexity** | O(n2) | O(n2) |
| **Space Complexity** | O(1) | O(1) |
| **Readability** | Easy | Easy |

## 8.7 Conclusion

In conclusion, we used two approaches to solve the problem

The brute force approach compare the scaled weight with each possibility of fake box number

On the other hand, the divide & conquer approach sort the possibilities and use binary search to find the fake box

But regardless of algorithms used all of them will lead to same efficiency of Brute force so brute force is better because of it is simplicity