

Project report

Orca Team



April 30, 2024

Ain Shams University

[Company address]

|  |  |
| --- | --- |
| **A logo of a university  Description automatically generated**Ain Shams University | CSE 332s: Algorithms |
| Faculty of Engineering | 3rd Year Electrical Engineering |
| Computer and Systems Eng. Dept. | Spring 2024 |

Table of Contents

[2. Team Member 2](#_Toc165401380)

[3. Tasks 2](#_Toc165401381)

[1. Task 01 2](#_Toc165401382)

[1. Problem Description: 2](#_Toc165401383)

[2. Algorithm: 3](#_Toc165401384)

[3. Pseudocode: 3](#_Toc165401385)

[4. Complexity Analysis: 3](#_Toc165401386)

[5. The used algorithm 3](#_Toc165401387)

[6. Screenshot from the output! 4](#_Toc165401388)

[7. Reference 5](#_Toc165401389)

[2. Task 02 5](#_Toc165401390)

[1. Problem Description: 5](#_Toc165401391)

[2. Algorithm: 5](#_Toc165401392)

[3. Pseudocode: 5](#_Toc165401393)

[4. Complexity Analysis: 6](#_Toc165401394)

[5. The Used Algorithm: 6](#_Toc165401395)

[6. Sample Output 6](#_Toc165401396)

[7. Conclusion 6](#_Toc165401397)

[8. Analytics 7](#_Toc165401398)

[9. Screenshot from the output! 8](#_Toc165401399)

[10. References 8](#_Toc165401400)

[3. Task 03 8](#_Toc165401401)

[1. Problem Description: 8](#_Toc165401402)

[2. Algorithm: 8](#_Toc165401403)

[3. Pseudocode: 8](#_Toc165401404)

[4. Complexity Analysis: 9](#_Toc165401405)

[5. The Used Algorithm: 9](#_Toc165401406)

[6. Sample Output 9](#_Toc165401407)

[7. Analytics 9](#_Toc165401408)

[4. Task 04 10](#_Toc165401409)

[1. Problem Description: 10](#_Toc165401410)

[2. Algorithm: 10](#_Toc165401411)

[3. Pseudocode: TODO 10](#_Toc165401412)

[4. Complexity Analysis: 10](#_Toc165401413)

[5. The Used Algorithm: 10](#_Toc165401414)

[6. Sample Output 10](#_Toc165401415)

[7. Screenshot from the output! 10](#_Toc165401416)

[5. Task 05 10](#_Toc165401417)

[1. Problem Description: 10](#_Toc165401418)

[2. Algorithm: 11](#_Toc165401419)

[3. Pseudocode: 11](#_Toc165401420)

[4. Complexity Analysis: 12](#_Toc165401421)

[5. The Used Algorithm: 12](#_Toc165401422)

[6. Sample Output 12](#_Toc165401423)

[7. Screenshot from the output! 12](#_Toc165401424)

[6. Task 06 // TODO 12](#_Toc165401425)

[1. Problem Description: 12](#_Toc165401426)

[2. Algorithm: 12](#_Toc165401427)

[3. Pseudocode: 12](#_Toc165401428)

[4. Complexity Analysis: 13](#_Toc165401429)

[5. The Used Algorithm: 13](#_Toc165401430)

[6. Sample Output 13](#_Toc165401431)

[7. Screenshot from the output! 13](#_Toc165401432)

[7. Task 08 13](#_Toc165401433)

[1. Problem Description: 13](#_Toc165401434)

[2. Algorithm: 13](#_Toc165401435)

[3. Pseudocode: 13](#_Toc165401436)

[4. Complexity Analysis: 13](#_Toc165401437)

[5. The Used Algorithm: 14](#_Toc165401438)

[6. Sample Output: 14](#_Toc165401439)

[7. Screenshot from the output! 14](#_Toc165401440)

# Team Member

|  |  |
| --- | --- |
| Student Name | Student ID |
| Mahmoud Abdelraouf Mahmoud | 2001436 |
| Eslam Mohamed Marzouk Abdelaziz | 2000252 |
| Ahmed Khaled Abdelmaksod Ebrahim | 2000218 |
| Abdelrahman Elsayed Ahmed Mohamed | 2002139 |
| Adham Khaled Abdelmaqsoud | 2000066 |
| Sara Ashraf Abdelhakam | 2000337 |
| Karim Ibrahim Saad Abd-Elrazek | 2001118 |
| Ahmed Mohammed Bakr Ahmed | 2000037 |
| Marwan Wael Mahmoud abbas | 2001244 |
| Somaya Ayman El-Sayed Ahmed | 2000423 |
| Mennatallah Amr Ali Abdel-Motaleb | 2002111 |
| Maya Ahmed Abdullah Ali | 2002124 |

# Tasks

## Task 01

### Problem Description:

We are given a 2n × 2n board with one missing square. Our task is to tile this board with right trominoes of three different colors, such that no pair of trominoes that share an edge have the same color. The right tromino is an L-shaped tile formed by three adjacent squares.

### Algorithm:

To solve this problem using dynamic programming, we can follow these steps:

1. **Define State:**

* Let's define our state as a tuple (i, mask), where:
* I represent the current column we are filling (from 0 to 2n - 1).
* mask is an integer that represents the colors used in the trominoes in the previous column. We can use binary representation where each bit represents whether a color is used or not.

1. **Base Case:**

* When i = 2n, we have filled the entire board, so we return 1 (indicating success).

1. **Recursive Step:**

* At each column i, we iterate through all possible ways to place a tromino.
* We consider three cases for each position in the column:

1. If the square is already filled, we move to the next square.
2. If we can place a tromino vertically, we recursively call the function for the next column with updated mask and position.
3. If we can place a tromino horizontally, we recursively call the function for the next column with updated mask and position.
4. **Memoization:**

* To avoid recomputation, we memoize the results for each state (i, mask) in a 2D array.

1. **Backtracking:**

* Once we have computed the count of valid tilings, we can backtrack to find the actual tiling by storing the colors used for each tromino in another 2D array.

### Pseudocode:

function countTilings(i, mask):

if i = 2n:

return 1

if dp[i][mask] is not null:

return dp[i][mask]

result = 0

for each position in column i:

if the square is already filled:

continue

for each possible tromino placement:

if placement is valid:

new\_mask = update mask based on colors used in current tromino

result += countTilings(i + 1, new\_mask)

dp[i][mask] = result

return result

### Complexity Analysis:

* Time Complexity: Since each state is computed only once, and there are O(2^(2n)) states, the time complexity is O(2^(2n)).
* Space Complexity: The memoization array requires O(n \* 2^n) space.

### The used algorithm

1. Divide and Conquer
2. Dynamic Programming
3. Brute Force

Let's compare the three algorithms: Divide and Conquer, Dynamic Programming, and Brute Force. We'll assess them based on their approach, efficiency, and ease of implementation.

1. Divide and Conquer:

* Approach: This algorithm divides the problem into smaller subproblems, tiles each subproblem separately, and then combines the solutions. It recursively divides the grid into quadrants until it reaches a base case.
* Efficiency: The algorithm has a time complexity of O(n^2), as each quadrant of the grid is processed independently.
* Ease of Implementation: The implementation is straightforward and intuitive. It involves recursive function calls to tile the grid.

1. Dynamic Programming:

* Approach: Dynamic programming stores the solutions to subproblems in a table and reuses these solutions to solve larger subproblems. It involves filling a memoization table with the count of valid tilings for each state.
* Efficiency: The time complexity is O(3^n), where n is the size of the grid. This is because each cell can have three different colors, and there are 3^(n^2) possible colorings. However, memoization reduces the actual computational time by avoiding redundant calculations.
* Ease of Implementation: Implementing dynamic programming can be more complex compared to Divide and Conquer. It requires careful handling of state transitions and memoization.

1. Brute Force:

* Approach: Brute force tries all possible colorings of the grid and checks if each coloring satisfies the constraints. It exhaustively searches through all possible combinations.
* Efficiency: The time complexity is very high, O(3^(n^2)), as it explores all possible colorings without any optimization.
* Ease of Implementation: Brute force is relatively easy to implement, as it involves iterating through all possible colorings and checking their validity.

1. Conclusion:

* Efficiency: Divide and Conquer is the most efficient among the three, with a time complexity of O(n^2). Dynamic programming follows, with a time complexity of O(3^n), but it benefits from memoization. Brute Force is the least efficient, with a time complexity of O(3^(n^2)).
* Ease of Implementation: Brute force is the easiest to implement, followed by Divide and Conquer. Dynamic programming requires more careful handling due to state transitions and memoization.
* In summary, if efficiency is the primary concern, Divide and Conquer is the best choice. However, if ease of implementation is more important or if the problem size is small enough to tolerate the exponential complexity, Brute Force might suffice. Dynamic Programming strikes a balance between efficiency and ease of implementation but may be overkill for smaller problem instances.

### Screenshot from the output!

A screenshot of a computer

Description automatically generated

Figure 1 – Simple output for the Brute Force Algorithm

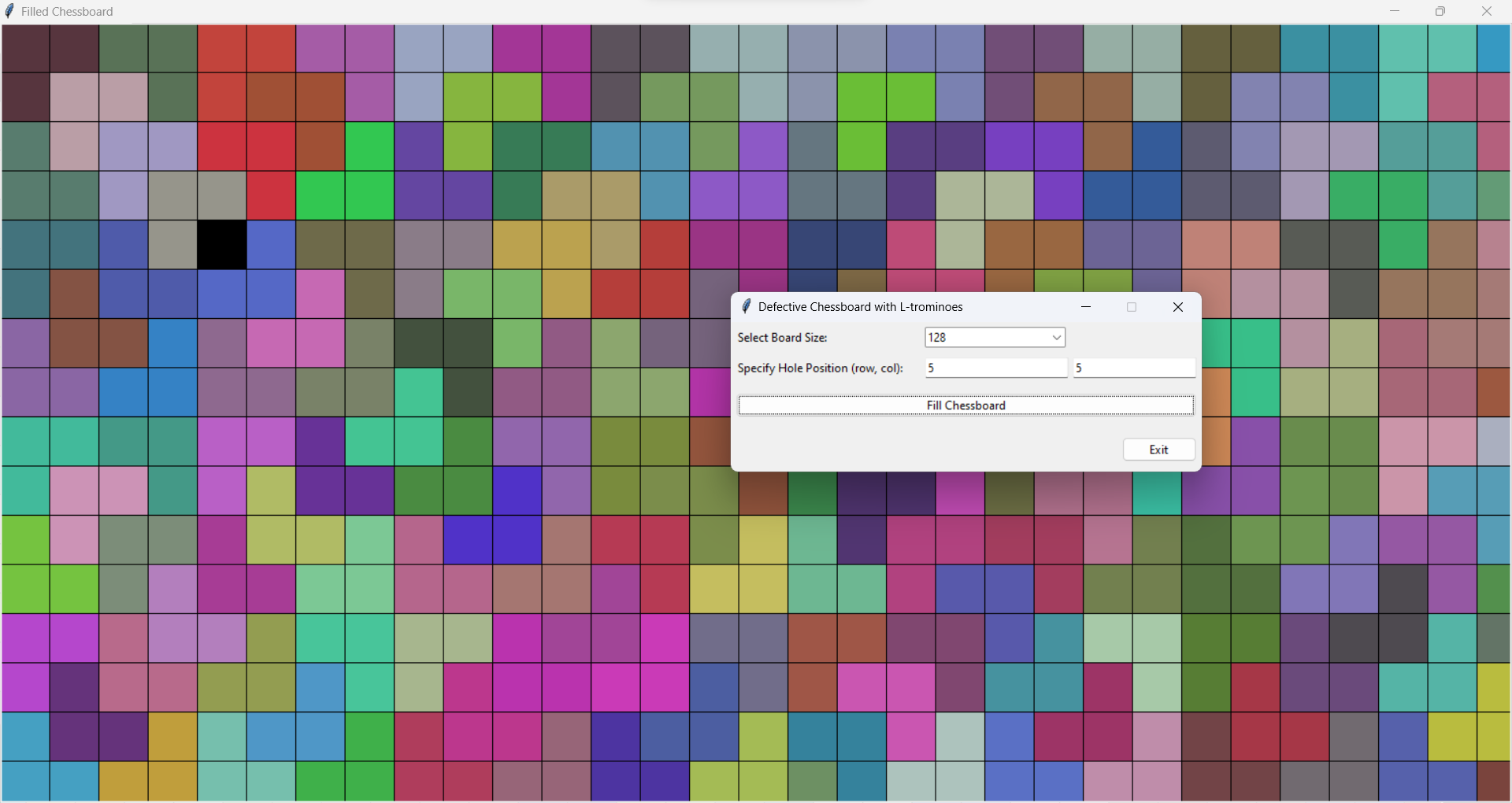


Figure 2 - Simple output for the Divide and Conquer Algorithm

### Reference

* <https://youtu.be/eSRNeIyX5dY?si=kjE_s2qZox5w9SW8>

## Task 02

### Problem Description:

Given an 8 × 8 chessboard, the task is to determine whether a chess knight can visit all the cells exactly once, ending at a cell one knight’s move away from the starting cell (i.e., a closed or re-entrant tour). A closed path tour means that the knight starts and ends at the same cell and visits each cell exactly once, moving like a knight in chess.

### Algorithm:

Greedy Algorithm for Closed Path Tour:

1. Initialize the chessboard and starting position.
2. Greedily pick the next move by selecting the move that leads to a cell with the fewest unvisited neighbors.
3. Continue picking moves until all cells are visited.
4. Check if the knight ends up one move away from the starting position.
5. If not, repeat the process until a closed path tour is found.

### Pseudocode:

while not finding closed path tour:

initializeTheBoard()

x = x\_init

y = y\_init

chessboard[x\_init][y\_init] = 0

for i from 0 to N \* N - 1:

if not pickNextMove(x, y):

continue

if not isNeighbourSquare(x, y, x\_init, y\_init):

continue

printChessBoard()

end

### Complexity Analysis:

* Time Complexity: O(N^3 \* m), where N is the width of the chessboard and m is the number of trials to find the solution. Each trial involves iterating through the entire chessboard to find the next move, which takes O(N^2) time, and performing this process for m trials.
* Space Complexity: O(N^2), representing the space required to store the chessboard, which is N × N in size.

### The Used Algorithm:

* The algorithm used is a greedy approach that iteratively selects the next move for the knight based on minimizing the number of unvisited neighbors at each step. It explores the chessboard until it finds a closed path tour, where the knight ends up one move away from the starting position. If a closed path tour is not found, the algorithm repeats the process until a solution is obtained.

### Sample Output

**at (0, 0)**

0 57 14 31 62 27 12 29

15 32 63 56 13 30 61 26

58 1 48 43 60 55 28 11

33 16 59 52 49 46 25 54

2 51 44 47 42 53 10 39

17 34 19 50 45 40 7 24

20 3 36 41 22 5 38 9

35 18 21 4 37 8 23 6

**at (2, 5)**

7 10 5 26 53 12 55 46

4 25 8 11 56 45 52 13

9 6 27 60 29 54 47 44

24 3 30 57 42 59 14 51

31 20 61 28 63 50 43 48

2 23 0 41 58 39 36 15

19 32 21 62 17 34 49 38

22 1 18 33 40 37 16 35

**at (3, 3)**

24 27 16 35 30 9 14 11

17 38 25 28 15 12 31 8

26 23 36 55 34 29 10 13

37 18 39 0 57 44 7 32

22 1 56 43 54 33 62 45

19 40 21 48 63 58 53 6

2 49 42 59 4 51 46 61

41 20 3 50 47 60 5 52

### Conclusion

* We can make a closed tour on a chess board using a knight without moving to a square twice.

### Analytics

A graph with numbers and lines

Description automatically generated

Figure 3 – No. of trails to find closed path.

A graph with numbers and lines

Description automatically generated

Figure 4 – No. of moves to find closed path.

### Screenshot from the output!

A screenshot of a computer

Description automatically generated

Figure 5 - Simple output for the Greedy Algorithm

### References

* <https://sites.science.oregonstate.edu/math_reu/proceedings/REU_Proceedings/Proceedings2004/2004Ganzfried.pdf>

## Task 03

### Problem Description:

* In this problem, there is a row of n security switches protecting a military installation entrance. The switches have specific rules for manipulation:
* The rightmost switch can be turned on or off at will.
* Any other switch can be turned on or off only if the switch to its immediate right is on, and all the other switches to its right are off.
* The objective is to design a dynamic programming algorithm to turn off all the switches, which are initially all on, in the minimum number of moves. Each toggle of a switch counts as one move. Additionally, the algorithm needs to determine the minimum number of moves required.

### Algorithm:

* Dynamic Programming Algorithm:

1. Initialize an array dp of size n to store the minimum number of moves required to turn off all switches from index 0 to index i.
2. Set dp[0] to 1 if the first switch is initially on (1) and 0 if it is off (0).
3. Iterate through the switches from index 1 to n-1:
   * If the current switch is on and the previous switch is off, set dp[i] = dp[i-1] + 1.
   * If the current switch is off, set dp[i] = dp[i-1].
   * If the current switch is on and the previous switch is on, set dp[i] = dp[i-1] + 1.
4. The minimum number of moves required to turn off all switches is dp[n-1].

### Pseudocode:

function min\_switch\_helper(n, ans):

if n <= 2:

return n

end if

if ans[n] is not -1:

return ans[n]

end if

x = min\_switch\_helper(n - 1, ans) + 2 \* min\_switch\_helper(n - 2, ans) + 1

ans[n] = x

return ans[n]

end function

function min\_switch(n):

ans = new array of size n + 1

for i from 0 to n:

ans[i] = -1

end for

return min\_switch\_helper(n, ans)

end function

### Complexity Analysis:

* Time Complexity: O(n), where n is the number of switches. The algorithm iterates through the switches once to compute the minimum number of moves.
* Space Complexity: O(n), as the algorithm uses an array of size n to store the minimum number of moves required.

### The Used Algorithm:

* The algorithm utilizes dynamic programming to efficiently compute the minimum number of moves required to turn off all switches. It considers the state of each switch and its relation to the previous switch to determine the minimum number of moves. This approach ensures an optimal solution to the problem.

### Sample Output

**@ n = 4**

10

**@ n = 5**

21

### Analytics

A graph with a line

Description automatically generated

Figure 6 – Min no. of moves to turn all switches off.

## Task 04

### Problem Description:

* In this problem, there are eight disks of different sizes and four pegs. The disks are initially stacked on the first peg in order of size, with the largest disk at the bottom and the smallest at the top. The objective is to transfer all the disks from the first peg to another peg using a sequence of moves, following the rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on top of a larger disk or an empty peg.

* The task is to determine whether a Dynamic Programming algorithm can solve the puzzle in 33 moves. If not, design an algorithm that accomplishes the task in 33 moves.

### Algorithm:

* Divide and Conquer Algorithm:

1. Recursively move the top n-1 disks from the source peg to an auxiliary peg, using the destination peg as a buffer.
2. Move the largest disk from the source peg to the destination peg directly.
3. Recursively move the n-1 disks from the auxiliary peg to the destination peg, using the source peg as a buffer.

### Pseudocode: TODO

Function moveDisks(n, source, destination, auxiliary):

if n == 1:

move disk from source to destination

else:

moveDisks(n-1, source, auxiliary, destination) // Move top n-1 disks to auxiliary peg

move disk from source to destination // Move largest disk to destination peg

moveDisks(n-1, auxiliary, destination, source) // Move n-1 disks from auxiliary to destination peg

Function solveTowerOfHanoi():

call moveDisks(8, sourcePeg, destinationPeg, auxiliaryPeg)

### Complexity Analysis:

* Time Complexity: The time complexity of the Tower of Hanoi problem solved using the Divide and Conquer algorithm is O(2^n), where n is the number of disks. Since n=8 in this case, the number of moves required is 2^8 - 1 = 255 moves. Therefore, the Dynamic Programming algorithm cannot solve the puzzle in 33 moves.
* Space Complexity: The space complexity of the algorithm is O(n), where n is the number of disks. This is due to the recursive calls made during the execution of the moveDisks function.

### The Used Algorithm:

* The Divide and Conquer algorithm recursively break down the problem of moving disks into smaller subproblems, moving smaller stacks of disks around until the entire stack is transferred to the destination peg. This approach ensures that the rules of the Tower of Hanoi puzzle are followed and that all disks are transferred successfully. However, it requires a number of moves that grows exponentially with the number of disks, making it unsuitable for solving the puzzle in 33 moves.

### Sample Output

### Screenshot from the output!

## Task 05

### Problem Description:

* In this problem, there are n coins initially placed in a row. The objective is to form n/2 pairs of coins through a sequence of moves. On each move, a single coin must jump over adjacent coins, with the number of adjacent coins to jump over increasing by one on each subsequent move. The goal is to determine all 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.

### Algorithm:

* Greedy Algorithm to Find Minimum Number of Moves:

1. Start from the leftmost coin.
2. On each move, jump the current coin over the next k adjacent coins, where k increases by one on each move.
3. Continue this process until n/2 pairs of coins are formed.

* Divide and Conquer Algorithm to Find Minimum Number of Moves:

1. Check if it's possible to form pairs of coins using the given value of n.
2. If possible, recursively calculate the minimum number of moves needed to form pairs using the divide and conquer approach.
3. Base case: If there are only 2 coins, return 1 move.
4. Increment the counter and call the function recursively with n - 2 until reaching the base case.
5. Return the minimum number of moves calculated.

This algorithmic approach combines the greedy strategy for efficiency with the divide and conquer technique for finding the minimum number of moves. The greedy approach handles the majority of cases efficiently, while the divide and conquer approach addresses scenarios where a direct solution is not feasible.

### Pseudocode:

// Function to check if it's possible to solve the problem for given 'n'

function check\_possible(n):

if n % 4 == 0:

return true

return false

// Function to check if the puzzle is solved

function check\_done(coins, size):

numberOfTwos = 0

numberOfZeroes = 0

for i from 0 to size - 1:

if coins[i] == 2:

numberOfTwos++

else if coins[i] == 0:

numberOfZeroes++

return numberOfTwos == size / 2 && numberOfZeroes == size / 2

// Function to perform a jump

function jump(coins, coin1\_position, coin2\_position):

coins[coin1\_position] = 0

coins[coin2\_position] = 2

// Function to check if a move is possible

function move\_possible(coins, pos1, pos2, moveNum):

if coins[pos1] && coins[pos2] != 1:

return false

sum = 0

for i from pos1 + 1 to pos2 - 1:

sum += coins[i]

if sum == moveNum:

return true

return false

// Function to find the minimum moves to solve the puzzle

function minimum\_moves(coins, size):

if !check\_possible(size):

return -1

move = 1

count = size - 1

for k from 0 to (size / 4) - 1:

while coins[count] != 1:

count--

for j from count - 1 down to 0:

if move\_possible(coins, j, count, move):

jump(coins, j, count)

move++

count--

break

for k from 0 to 3 \* size / 4:

if coins[k] != 1:

continue

for j from k + 1 to size - 1:

if move\_possible(coins, k, j, move):

jump(coins, k, j)

move++

break

return move - 1

### Complexity Analysis:

* Time Complexity: The time complexity of the greedy algorithm is O(n), as it iterates through the coins to form pairs and calculates the minimum number of moves.
* Space Complexity: The space complexity is also O(n), as the algorithm stores the sequence of moves needed to form pairs.

### The Used Algorithm:

* Greedy Algorithm:
* The greedy algorithm iterates through the coins and forms pairs by jumping each coin over adjacent coins in a sequence of moves.
* It starts with a single move, where a coin jumps over one adjacent coin, and increases the number of adjacent coins to jump over on each subsequent move until n/2 pairs of coins are formed.
* This approach ensures that the minimum number of moves is used to form pairs of coins while following the given rules.
* Divide and Conquer Algorithm:
* The divide and conquer algorithm recursively calculate the minimum number of moves needed to form pairs of coins.
* It checks if it's possible to form pairs of coins using the given value of n. If possible, it recursively calculates the minimum number of moves needed using the divide and conquer approach.
* The base case is when there are only 2 coins, where it returns 1 move.
* The algorithm then increments the counter and calls itself recursively with n - 2 until reaching the base case.
* Finally, it returns the minimum number of moves calculated.

### Sample Output

### Screenshot from the output!

## Task 06 // TODO

### Problem Description:

* In this problem, there are six knights positioned on a 3 × 4 chessboard: the three white knights are at the bottom row, and the three black knights are at the top row. The objective is to design an iterative improvement algorithm to exchange the positions of the knights to achieve a specific configuration, ensuring that no more than one knight occupies a square at any time. The goal is to achieve this in the minimum number of knights moves.

### Algorithm:

* Iterative Improvement Algorithm:

1. Start with the initial configuration of knights on the chessboard.
2. Repeat the following steps until the desired configuration is achieved or no further improvement can be made:
   * For each knight, consider all possible moves to adjacent squares on the board.
   * Evaluate the potential moves based on predefined criteria, such as minimizing the distance to the target configuration.
   * Choose the move that leads to the best improvement and apply it.
3. Continue iterating until the desired configuration is reached or until a predefined number of iterations is reached.

### Pseudocode:

Function iterativeImprovement():

Initialize the chessboard with the initial configuration of knights

Set iteration counter to 0

Repeat until desired configuration is achieved or maximum iterations reached:

For each knight on the board:

For each possible move to adjacent squares:

Evaluate the potential move based on predefined criteria

Choose the move that leads to the best improvement

Increment iteration counter

Return the final configuration of knights

### Complexity Analysis:

* Time Complexity: The time complexity of the iterative improvement algorithm depends on factors such as the size of the chessboard, the number of knights, and the criteria used to evaluate moves. In general, it can range from O(n^2) to O(n^3), where n is the number of iterations or the size of the chessboard.
* Space Complexity: The space complexity depends on the data structures used to represent the chessboard and the knights' positions. Typically, it is O(n), where n is the number of squares on the chessboard.

### The Used Algorithm:

* The iterative improvement algorithm systematically explores possible moves for each knight and selects the moves that lead to the best improvement toward the target configuration. By iteratively refining the positions of the knights based on predefined criteria, the algorithm aims to minimize the number of knights moves required to achieve the desired arrangement. This approach combines exploration and optimization to efficiently solve the problem.

### Sample Output

### Screenshot from the output!

## Task 08

### Problem Description:

* In this problem, you have 50 boxes containing 50 pieces of metal, all of the same known weight. However, one of these boxes contains fake metal pieces that weigh 1 kilogram less than the pieces in the rest of the boxes. The task is to design an algorithm to identify the fake box using a digital scale, with only one use allowed.

### Algorithm:

* The brute force algorithm iterates through each box, calculating the total weight of the metal pieces based on the box's index. It then places a new box containing known weight metal pieces on the digital scale. By subtracting the measured weight from the total weight calculated without the fake box, the algorithm identifies the fake box.

### Pseudocode:

Initialize total\_weight to 0

Initialize fake\_box\_index to 0

for index from 1 to 50:

balls\_count = index

total\_weight += balls\_count \* weight\_of\_non\_fake\_ball

new\_box\_weight = balls\_count \* weight\_of\_non\_fake\_ball

end

Place the new box on the digital scale

real\_weight = weight measured by the scale

fake\_box\_index = (total\_weight - real\_weight) / weight\_of\_non\_fake\_ball

Display "Weight Difference: ", (total\_weight - real\_weight)

Display "Fake Box index: ", fake\_box\_index

### Complexity Analysis:

* Time Complexity: O(n), where n is the number of boxes. The algorithm iterates through all 50 boxes once.
* Space Complexity: O(1), as the algorithm only uses a few variables for calculations.

### The Used Algorithm:

* The algorithm employs a brute force approach, calculating the total weight of metal pieces without the fake box and comparing it with the measured weight on the digital scale to identify the fake box.

### Sample Output:

Input:

2 21

Output:

Weight Difference: 42

Fake Box index: 21

Random Sample:

Entered Fake Box Index: 34

Entered Weight of Each Ball: 23

================================================================

Expected Weight: 29325

Sum of weights drawn from each Box weighed on digital balance: 29290

Calculated Fake Box Index: 34

Checking if it is a fake box by asking the box it self and it said: True, I am the the fake box and my id is 34

### Screenshot from the output!

A computer screen shot of a computer screen

Description automatically generated

Figure - Snapshot 01 from the GUI of Task 08

A screenshot of a computer

Description automatically generated

Figure - Snapshot 02 from the GUI of Task 08