

**Program: CESS**

***Course Code: CSE332***

***Course Name: Design and Analysis of Algorithms***

***Examination Committee***

**Dr. Gamal A. Ebrahim**

**By:**

**Seif Ahmed ElSayed Elewa 18P5662**

**Maryam Ahmed Galal Nouh 18P2824**

**Hana Yasser Amgad ElSokkary 18P5007**

**Abdel-Rahman Ibrahim Megahed 18P7423**

**Youssef Mohamed Mostafa Mansi 18P5848**

**Ain Shams University**

**Faculty of Engineering**

**International Credit Hours Engineering Programs (I-CHEP)**

**Spring Semester – 2021**

PROJECT SUBMISSION

Design and analysis of algorithms

Table Contents

[Table of figures 4](#_Toc73814292)

[Table of tables 4](#_Toc73814293)

[Task 1: 5](#_Toc73814294)

[Assumptions: 5](#_Toc73814295)

[Description of steps (greedy): 5](#_Toc73814296)

[Pseudocode: 5](#_Toc73814297)

[Complexity Analysis: 5](#_Toc73814298)

[Comparison: 5](#_Toc73814299)

[Sample output: 6](#_Toc73814300)

[Conclusion: 6](#_Toc73814301)

[Task 2 A 6](#_Toc73814302)

[Assumptions: 6](#_Toc73814303)

[Description of the steps: 6](#_Toc73814304)

[Pseudocode: 7](#_Toc73814305)

[Complexity Analysis: 8](#_Toc73814306)

[Comparison: 8](#_Toc73814307)

[Sample output: 8](#_Toc73814308)

[Conclusion: 9](#_Toc73814309)

[Task 2 B 9](#_Toc73814310)

[Assumptions: 9](#_Toc73814311)

[Description of steps: 10](#_Toc73814312)

[Ex: 10](#_Toc73814313)

[Pseudo code: 10](#_Toc73814314)

[Sample Output: 11](#_Toc73814315)

[Complexity Analysis: 12](#_Toc73814316)

[Comparison: 12](#_Toc73814317)

[Conclusion: 12](#_Toc73814318)

[Task 3 12](#_Toc73814319)

[Assumptions: 12](#_Toc73814320)

[Description of the steps: 13](#_Toc73814321)

[Pseudocode: 13](#_Toc73814322)

[Complexity Analysis: 13](#_Toc73814323)

[Comparison: 13](#_Toc73814324)

[Pseudocode: 14](#_Toc73814325)

[Sample output: 14](#_Toc73814326)

[Conclusion: 14](#_Toc73814327)

[Task 4 15](#_Toc73814328)

[Assumptions: 15](#_Toc73814329)

[Pseudocode: 15](#_Toc73814330)

[Description of the steps: 16](#_Toc73814331)

[Code: 17](#_Toc73814332)

[Complexity Analysis: 17](#_Toc73814333)

[Comparison: 17](#_Toc73814334)

[Sample output: 18](#_Toc73814335)

[Conclusion: 19](#_Toc73814336)

[Task 5 19](#_Toc73814337)

[Assumptions 19](#_Toc73814338)

[Pseudocode: 19](#_Toc73814339)

[Description of Steps: 19](#_Toc73814340)

[Complexity Analysis: 20](#_Toc73814341)

[Comparison: 20](#_Toc73814342)

[Sample Output: 20](#_Toc73814343)

[Conclusion: 20](#_Toc73814344)

[Task 6: 20](#_Toc73814345)

[Assumptions: 21](#_Toc73814346)

[Description of steps ( dynamic programming): 21](#_Toc73814347)

[Ex: 22](#_Toc73814348)

[Pseudo Code 23](#_Toc73814349)

[Complexity Analysis: 23](#_Toc73814350)

[Sample output: 23](#_Toc73814351)

[Description of steps (Brute Force Approach): 23](#_Toc73814352)

[Comparison 24](#_Toc73814353)

[Conclusion 24](#_Toc73814354)

[References 25](#_Toc73814355)

# Table of figures

[Figure 1: Output Greedy of task 1 6](#_Toc73814375)

[Figure 2: Output Divide and Conquer of Task 1 6](#_Toc73814376)

[Figure 3: Output 1 Task 2a 8](#_Toc73814377)

[Figure 4: Output 2 Task 2a 9](#_Toc73814378)

[Figure 5: Output 3 Task 2a 9](#_Toc73814379)

[Figure 6: Output 4 Task 2a 9](#_Toc73814380)

[Figure 7: Output of 4\*4 Square Task 2b 11](#_Toc73814381)

[Figure 8: Output of 15\*15 Square Task 2b 11](#_Toc73814382)

[Figure 9: Output of 20\*20 square Task 2b 12](#_Toc73814383)

[Figure 11: Output of Task 3 14](#_Toc73814384)

[Figure 12: Task 4 Output on the third iteration using the brute force approach 18](#_Toc73814385)

[Figure 13: Task 4 Output on the third iteration using the equation 18](#_Toc73814386)

[Figure 14: Task 5 output 20](#_Toc73814387)

[Figure 15:first Iteration Tromino Tiling 22](https://engasuedu.sharepoint.com/sites/OSProject/Shared%20Documents/Algo/Final%20Report.docx#_Toc73814388)

[Figure 16: third Iteration Tromino Tiling 22](https://engasuedu.sharepoint.com/sites/OSProject/Shared%20Documents/Algo/Final%20Report.docx#_Toc73814389)

[Figure 17: Second Iteration Tromino Tiling 22](https://engasuedu.sharepoint.com/sites/OSProject/Shared%20Documents/Algo/Final%20Report.docx#_Toc73814390)

[Figure 18: Fourth Iteration Tromino Tiling 22](https://engasuedu.sharepoint.com/sites/OSProject/Shared%20Documents/Algo/Final%20Report.docx#_Toc73814391)

[Figure 19: Impossible solutions for Tromino Tiling 22](https://engasuedu.sharepoint.com/sites/OSProject/Shared%20Documents/Algo/Final%20Report.docx#_Toc73814392)

[Figure 20: Task 6 output 23](#_Toc73814393)

# Table of tables

[Table 1: Case 1 Task 2a 7](#_Toc73814394)

[Table 2: Case 2 Task 2a 7](#_Toc73814395)

[Table 3: Task 4 Iterations to count triangles 16](#_Toc73814396)

# **Task** 1:

**Problem:** The stick of length N ‎[2]

**Objective:** Minimum number of cuts to get N pieces of the stick

## Assumptions:

* The stick is of length *n* units
* The stick is to be cut to *n* pieces
* Several pieces can be but at the same time

## Description of steps (greedy):

* We have a stick of length N that needs to be cut into N pieces of same length (1)
* We will cut the stick into two parts with maximum length. In the next step we will cut each part obtained into two maximum length parts. Finally, we will repeat this step to every part we obtain until we obtain n unit pieces.
* The solution will always be ceiling of log2(N) since the number of times needed is the number of times, we divide each piece into two parts hence N<=2x

## Pseudocode:

Algorithm stick(n)

//input: length of stick

//output: minimum number of cuts to get N pieces of the stick

while N > 1 do

N <- N - 1

i <- i + 1

end while

return i

## Complexity Analysis:

When calculating the mathematical equation returned, the function is of order of growth of O(n) since the main operation, here is the comparison, which is done N+1 times.

## Comparison:

The dynamic programming approach allows us to reach the optimal order of growth, O(1), since the main operation, here is the logarithmic function, is calculated once, when the function is called. Meanwhile the greedy algorithm is achieved with the order of growth of O(n). Here, we can see that the order of growth of a greedy algorithm is worse than the dynamic programming approach, specifically in this problem. ‎[1]

## Sample output:

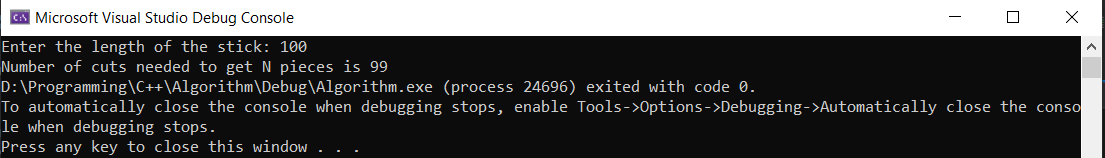


Figure 1: Output Greedy of task 1

Text

Description automatically generated

Figure 2: Output Divide and Conquer of Task 1

## Conclusion:

Dynamic programming approach is more suitable for this problem as it has better order of growth than the greedy approach.

# Task 2 A

**Problem:**Magic SquareProblem ‎[2]

**Objective:**Find the Magic Square of n x n matrix

## **Assumptions**:

* n is input by the user
* n is greater than or equal to three

## Description of the steps:

A magic square is a square of dimensions n x n, that contains the digits one through n², distributed across the matrix such that the sum of the diagonals equals the sum of all rows, which also equals the sum of all columns.

To solve this type of question, one must search first for the common value to be placed at the center of the matrix, which will be a permanent element in the sum of any row, column or diagonal. This element can be found by calculating the ceiling of n2/2. To find the value of the total sum for each row, column or diagonal, one must find the sum of all the elements of the matrix:

Note: In a 3 x 3 square, the sum is 15 while the common element is found to be 5.

Now, given an n x n table, it is required to find the values of n that would allow the matrix to be filled with digits one through nine, such that each 3 x 3 square belonging to the larger magic square must be a magic square itself.

For this to be possible, the central element of each 3 x 3 square in the general matrix must be the element 5. Now, the problem that occurs, is that the second row of the matrix will have n-2 elements of value 5, which violates the ground rule of a magic square concerning the sum of its diagonals, rows and columns. Hence, the only values of n that would satisfy these conditions would be n equals 3.

Considering the case where the question doesn’t specifically want that each square is a 3 x 3 magic square but that some of them are. In this case, we have two different tracks to manage. In the first track, n is a multiple of 3. Which means that simply we will have n magic squares of size 3 x 3, where every 3 rows or columns, a 3 x 3 magic square reappears at these coordinates. ‎[3]

On the other hand, another track to be explored is in the case of n being odd, which is the only other possible case for n, such that a n x n matrix is obtainable. In that case, a pattern is observable, where propagation of rows/columns one, two and three follows the pattern 2 – 1 – 2 – 3.

Case 1: (n = 6)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 6 | 1 | 8 | 6 | 1 | 8 |
| 7 | 5 | 3 | 7 | 5 | 3 |
| 2 | 9 | 4 | 2 | 9 | 4 |
| 6 | 1 | 8 | 6 | 1 | 8 |
| 7 | 5 | 3 | 7 | 5 | 3 |
| 2 | 9 | 4 | 2 | 9 | 4 |

Table 1: Case 1 Task 2a

Case 2: (n = 7)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 6 | 1 | 8 | 1 | 6 | 1 | 8 |
| 7 | 5 | 3 | 5 | 7 | 5 | 3 |
| 2 | 9 | 4 | 9 | 2 | 9 | 4 |
| 7 | 5 | 3 | 5 | 7 | 5 | 3 |
| 6 | 1 | 8 | 1 | 6 | 1 | 8 |
| 7 | 5 | 3 | 5 | 7 | 5 | 3 |
| 2 | 9 | 4 | 9 | 2 | 9 | 4 |

Table 2: Case 2 Task 2a

## Pseudocode:

Algorithm MagicSquare(array[0..n-1][0..n-1] , n)

//input: matrix of size n x n

//output: magic square of n x n matrix

if n < 3 then

// error message

for (int i <- 0; i < 3; i=i+1)

        for (int j <- 0; j < 3; j = j+1)

            array[i][j] <- 3 x 3 magic square

else if n%3 = 0

// copy paste the 3 x 3 magic square every 3 rows and every 3 columns

else if n%2 = 1

// fill rows and columns following the 2-1-2-3 rule

else

// error message

## Complexity Analysis:

The main operation here is assignment to migrate elements from one matrix to another, repeated n times in two nested for loops. The complexity here is of order of growth O(n2).

## Comparison:

Finding the magic square of an n x n matrix used to be done by trial and error, which would be tackled using the brute force approach. This guessing would lead to a complexity of O(n2!), since there are n x n elements, that could take values from 1 through 9, which is worse than the order of growth of this algorithm when computed using the dynamic programming approach ‎[1]

## Sample output:

Text

Description automatically generated

Figure 3: Output 1 Task 2a

Text

Description automatically generated

Figure 4: Output 2 Task 2a

Text

Description automatically generated

Figure 5: Output 3 Task 2a

Text

Description automatically generated

Figure 6: Output 4 Task 2a

## Conclusion:

As deducted from the comparison above, we can clearly conclude that dynamic programming is a better approach than brute force for solving this problem. This is due to the fact that dynamic programming uses past knowledge to simplify solving the problem.

# Task 2 B

**Problem:***Pseudo*-*Magic Square**Problem* ‎[2]

**Objective:**Create a pseudo-magic square where every  3\*3 square is a magic square.

## Assumptions:

* n is input by the user
* n is greater than or equal to three

## Description of steps:

Since any 3x3 magic square is also a pseudo-magic square , so we generated a 3x3 magic square that takes numbers from 1 to 9. then we generated nxn matrix where the magic square is in its upper left corner. To copy columns and rows in the right places, we should first check .

## Ex:

having 5x5 matrix:

* we should copy column 0 in column 3.
* 3=n-2.
* (n-2)%3==0: then column 0 will be copied in row 3, same for rows.

By copying rows and column in the right places , we guarantee achieving our goal by having n\*n table with integers 1 through 9,inclusive, so that every 3x3 square in it is a pseudo-magic square.

## Pseudo code:

**//**first we generate magic square

// second we generate the matrix that will contain the magic square in its upper left corner

    int mcg = 3

  int\*\* pseudo

   int y

    int z

      pseudo = new int\* [z]

For i=0 to z

        pseudo[i] = new int[y]

    for i=0 to 3

        for j = 0 to 3

            pseudo[i][j] = magicsq[i][j]

// copying rows and columns in right places

    for  i = 0 to z

        for j = 0 to z

             m = I // represents rows

            n = j     // represents columns

 // copying columns

            if n % 3 == 0

                n = 0        // column 0 will be copied in column n

            else if (n - 1) % 3 == 0

                n = 1 // column 1 will be copied in column (n-1)

            else if (n - 2) % 3 == 0

                n = 2 // column 2 will be copied in column(n-2)

            // copying rows

            if m % 3 == 0

                m = 0  // row 0 will be copied in row m

            else if (m - 1) % 3 == 0

                m = 1 // row1 will be copied in row(m-1)

            else if (m - 2) % 3 == 0

                m = 2 // // row 2 will be copied in row (n-2)

            pseudo[i][j] = magicsq[m][n]

    for  i = 0 to z

        for  j = 0 to y

            display pseudo[i][j]

## Sample Output:

Figure 7: Output of 4\*4 Square Task 2b

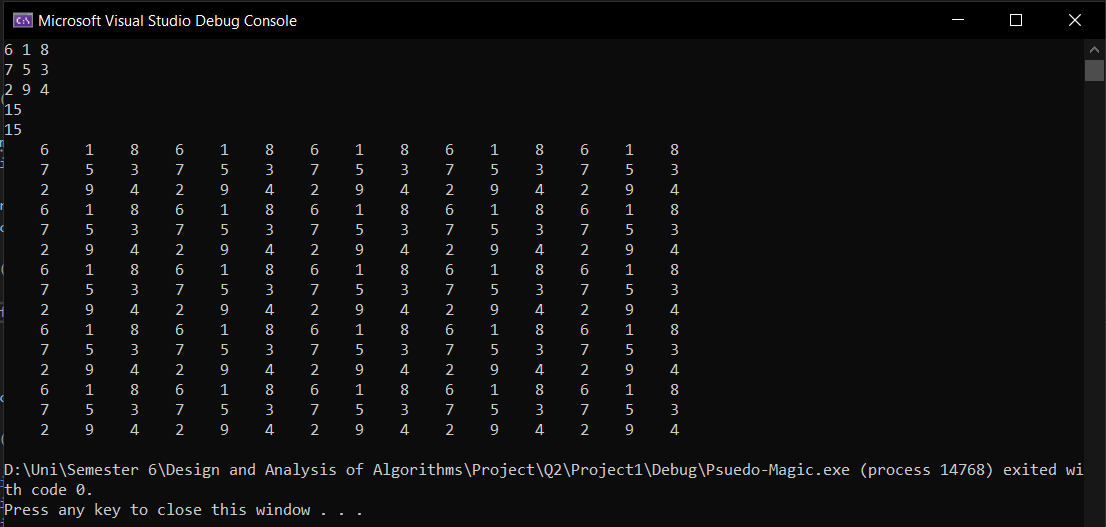


Figure 8: Output of 15\*15 Square Task 2b

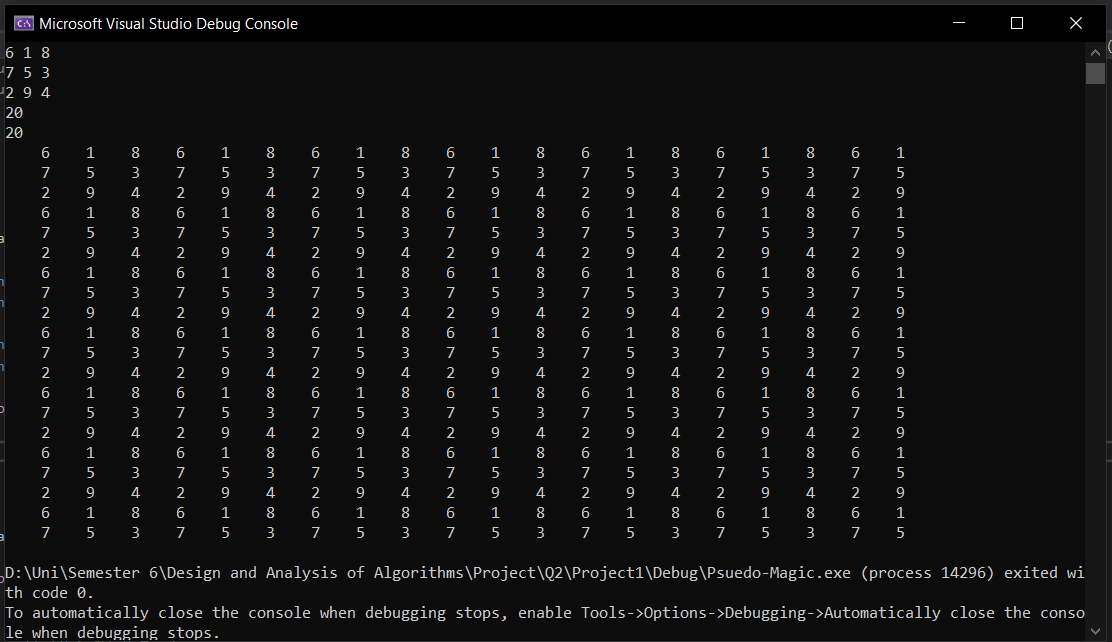


Figure 9: Output of 20\*20 square Task 2b

## Complexity Analysis:

O(n2), since we have an n\*n matrix , and we should check on rows and columns to ensure copying each of them in the right place, by getting the remainder of dividing row or column number (n) by 3. ‎[3]

## Comparison:

Dynamic programming is best approach to solve this problem, better than brute force that checks on every single possibility every time we try to fill the n\*n matrix  putting into consideration that every 3x3 should be a pseudo-magic square ‎[1].

## Conclusion:

Dynamic programming is the best approach to solve this problem , where it always guarantees to find the optimal solution by checking all possible ways to solve the problem and then picking the best one. It's an intelligent brute force.

# Task 3

**Problem:** *The Circle of N bulbs* ‎[2]

**Objective:** Switch on all the lightbulbs by toggling the minimum number of switches.

## Assumptions:

* Every light switch toggles its light bulb and the bulbs adjacent to it.
* Circle has n light bulbs where n > 2.
* All lightbulbs are originally closed.

## Description of the steps:

Since all the lights are off initially, it is obvious that the final state of the lights depends only on the parity (odd or even) of the number of times each switch is flipped and does not depend on the order in which the switches are manipulated. To turn on a light, we can either flip its switch and leave the adjacent switches off, or we flip all three switches, for a group of three lights. Assuming the lights are numbered from **1 to N** in clockwise order, we have two cases.

**N is divisible in 3:** In this case, we flip the switch for **1st** bulb, which lights the bulbs **1, 2 and N**. Then we flip the switch of **4th bulb**, which lights the bulb **3, 4, and 5**. In this manner, we flip the switch of every **3K + 1** bulb until we flip the switch of **N – 2** bulbs, which will light up the last 3 bulbs. This results in a total of **N/3** switch flips.

**N is not divisible by 3:** In this case, suppose we follow the same procedure as above. This will only turn on the bulbs in **multiple of 3**. So, at last, there will be **one or two bulbs**, which will be off. Flipping the switch of one of these bulbs, we turn off at least one bulb. This bulb, **say X**, which now has gone off due to the previous flip has been toggled even number of times, to turn it on, the switch of either **X** or one of its adjacent bulbs which has not been flipped needs to be flipped. In this way, we must flip the switch of each of the bulbs whose switch has not been flipped. This results in a total of **N** switch flips. ‎[4]

## Pseudocode:

Algorithm CircleOfLight(n)

//input: number of bulbs

//output: minimum number of flipped switches

If n % 3 == 0 then

Lights[0] = 1

Lights[3] = 1

Bulb = 2

For k < n/3

Lights[3k+1] = 1

bulb++

Else

Return n

## Complexity Analysis:

The main operation here is multiplication, repeated n/3 times in the for loop. The complexity here is of order of growth O(n).

## Comparison:

To get the minimum number of switches to be toggled directly from n, if n is divisible by 3 then the minimum number of switches will be n/3. If n is not divisible by 3 then n is the minimum number of switches to be toggled. The complexity of this algorithm is O(1) which is the highest efficiency an algorithm can run at. This is due to the main operation being the comparison ‎[1].

## Pseudocode:

Algorithm CircleOfLight(n)

//input: number of bulbs

//output: minimum number of flipped switches

If n%3 = 0

return n/3

else

return n

## Sample output:

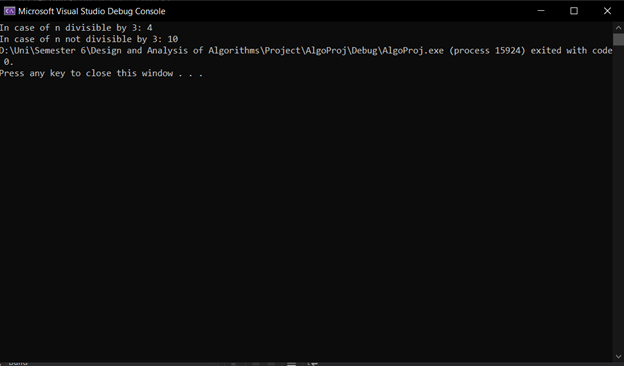


Figure 11: Output of Task 3

## Conclusion:

As shown above, the iterative improvement approach was not the most efficient way to solve this problem, which was proven by the complexity analysis of this algorithm, O(n). In fact, a more optimal algorithm was found using a different approach, which led to a complexity of O(1). This shows that here, the iterative improvement approach did not create an efficient solution to the problem.

# Task 4

**Problem:** *Counting triangles problem  ‎[2]*

**Objectives:** Count the equilateral triangles on each iteration using brute force approach for this algorithm.

## Assumptions:

* The first triangle is equilateral
* On each iteration, a new triangle is added for each uncovered edge
* The total number of triangles is what is to be found
* The iteration number is given as input by the user

## Pseudocode:

Algorithm countTriangles(array[0..n-1][0..n-1], n, sz)

// input: array of triangles and iteration number

// output: total number of triangles on this iteration

count <- 0

for i <- 0 to sz do

for j <- 0 to sz do

if array[i][j] != 0

count = count + 1

return count

Algorithm builtTriangles(array[0..14][0..14], n, sz)

// input: array of triangles and current iteration n and

iteration number input by the user

// output: triangles generated in array

// base case

if n = 2

array[sz/2][sz/2] <- 2

array [sz/2-1][sz/2] <- 2

array [sz/2][sz/2+1] <- 2

array [sz/2][sz/2-1] <- 2

return;

buildTriangles (array, n-1, sz)

for i <- 0 to sz-1 do

for j <- 0 to sz-1 do

// this is an existing triangle

if array[i][j] = n-1

// check which triangle around it is filled, to know which way to expand

if array[i+1][j] != 0

// right and left triangles exist too, expand above

if array[i][j+1]=n-1 && array[i][j-1]=n-1

// do not rebuild existing triangles

if array [i-1][j] = 0

array [i-1][j] <- n

if array [i][j+1] = 0

array [i][j+1] <- n

if array [i][j-1] = 0

array [i][j-1] <- n

if array[i][j+1] != 0

if array[i][j+1]!=n-1 || array[i][j-1]!=n-1

if array [i+1][j] = 0

array [i+1][j] <- n;

if array [i][j-1] = 0

array [i][j-1] <- n

if (array[i][j-1] != 0)

if array[i][j+1]!=n-1 || array[i][j-1]!=n-1

if array [i+1][j] = 0

array [i+1][j] <- n

if array [i][j+1] = 0

array [i][j+1] <- n

## Description of the steps:

1. Take from user the number of iterations
2. Build the triangles in a 2D array
3. Count the current, already existing, triangles
4. Return that number of triangles

|  |  |
| --- | --- |
| Iteration number | Triangles found |
| 1 | 1 |
| 2 | 1+3 = 4 |
| 3 | 4 + 6 = 1 + 3 + 6 = 10 |
| 4 | 10 + 9 = 1 + 3 + 6 + 9 = 19 |
| 5 | 19 + 12 = 1 + 3 + 6 + 9 + 12 = 31 |
| 6 | 31 + 16 = 1 + 3 + 6 + 9 + 12 + 16 = 46 |

Table 3: Task 4 Iterations to count triangles

From the table above, we can see that each iteration, a multiple of 3 is added to the original number of existing triangles. That multiple follows the series {0,1,2,3,4,…} which simply represents n-1. Therefore, the following recursive formula can be concluded: T(n) = T(n-1) + 3(n-1) [5]

Algorithm for counting the triangles – recursively

(input: n – iteration number; output: s – number of small triangles)

1. Calculate 3(n-1)
2. Recursively call T(n-1) = T(n-2) + 3(n-2)
3. Repeat steps 2 and 3 till n = 1, return 1
4. Finally, return T(n)

## Code:

int countTriangles(int n){

if (n <= 1)

return 1;

return countTriangles(n-1) + 3\*(n-1) ;

}

## Complexity Analysis:

O(n2) where the main operation is the multiplication

## Comparison:

Algorithm for counting the triangles – directly through n

(input: n – iteration number; output: s – number of small triangles)

1. Substitute with value of n in general formula s = 1 + 3 \* n(n-1)/2
2. Return s

Note: (using backward substitution)

T(n) = T(n-1) + 3(n-1)

= T(n-2) + 3(n-2) + 3(n-1)

= T(n-3) + 3(n-3) + 3(n-2) + 3(n-1)

…

= T(n-i) + 3(n-i) + … + 3(n-3) + 3(n-2) + 3(n-1)

For i = n – 1

= T(n-(n-1)) + 3(n-(n-1)) + … + 3(n-3) + 3(n-2) + 3(n-1)

= T(1) + 3(1) + … + 3(n-3) + 3(n-2) + 3(n-1)

= 1 + 3 + … + 3(n-3) + 3(n-2) + 3(n-1)

= 1 + 3 [ ∑ (1..n-1) i ]

= 1 + 3 \* (n-1-1+1)(n-1+1)/2

= 1 + 3 \* n(n-1)/2

This algorithm has a complexity of O(1), with the main operation, addition, which would be faster to compute and requires less memory space. However, this algorithm is not a brute force approach. ‎[1]

## Sample output:

(Third iteration)

A picture containing text, monitor, screenshot, electronics

Description automatically generated

Figure 12: Task 4 Output on the third iteration using the brute force approach

A picture containing text, monitor, screenshot, indoor

Description automatically generated

Figure 13: Task 4 Output on the third iteration using the equation

On the third iteration, as shown in the problem description given, the solution should be 10, since for n = 3, 1 + 3 \* n(n-1)/2 = 10.

## Conclusion:

As shown above, the brute force approach is not the most suitable, since for larger values of n, the program might crash. It is also not optimal in terms of time complexity nor space complexity, as O(n2) is not our best solution. In fact, the second algorithm found, with which the comparison was done, is better in terms of space and time complexity.

# Task 5

**Problem:***Counterfeit Coin problem* ‎[2]

**Objectives:** Find the index of the counterfeit coin using divide and conquer ‎[6]

Assumptions**:**

* Coins[]: array where the weight of each coin is stored
* The weight of the counterfeit coin is lighter than the weight of genuine coins.
* Since we do not have a real scale, the coin array helps us as if we are weighing the coins together

## Pseudocode:

Algorithm Get\_counterfeit( coins[0…n-1], start, end)

// input: array of coins and the start and the end of the array.

// output:  index of the counterfeit coin.

// split coins into 2 piles

If the array size is even

Then it will be divided into 2 equal piles and exclude 2 coins representing mid and mid1

Else

Then it will be divided into 2 equal piles and exclude only one coin that represents mid.

//we will compare the 2 equal piles and the lighter one is the one that contains the counterfeit coin

If (sum(coins, start, mid - 1) > sum(coins, mid1 + 1, end))

// function will be called and works on the lighter pile.

Get\_counterfeit(coins, mid1 + 1, end)

// steps will be repeated until 2 piles became equal

// we will compare the last 2 middle and return the lighter one.

if (coins[mid] < coins[mid1])

return mid

else if(coins[mid] > coins[mid1])

return mid1

// if coins[] doesn’t contain counterfeit coin

Return -1

## Description of Steps:

1. We check if the number of coins is even or not.
2. If even, then it will be divided into 2 equal piles and exclude 2 coins representing mid and mid1
3. If odd, Then it will be divided into 2 equal piles and exclude only one coin that represents mid.
4. we will compare the 2 equal piles and the lighter one is the one that contains the counterfeit coin .
5. function will be called and works on the lighter pile.
6. steps will be repeated until 2 piles became equal.
7. we will compare the last 2 middle and return the lighter one.
8. if coins[] doesn’t contain counterfeit coin, will return -1.

## Complexity Analysis:

O (log n) where the main operation is the comparison.

## Comparison:

This problem can be solved by using brute force technique:

Assuming having n coins {1, 2, 3 and 4} where one of them is a counterfeit coin that is lighter than all the other 3 genuine coins ‎[1].

1. Compare weights of coins 1 and 2.
2. If one weights less, return it.
3. Else, compare coins 3 and 4.

Complexity: O(n).

## Sample Output:

Graphical user interface, text

Description automatically generated

Figure 14: Task 5 output

## Conclusion:

In this problem, as we can see from the comparison done and the complexity analysis, we can conclude that the divide and conquer approach is more suitable than the brute force approach as it leads to more optimal order of growth.

# Task 6:

**Problem:** Tromino problem ‎[2]

**Objective:** How to fill board with Trominos except for a square of tiles

## **Assumptions**:

* The board is of size 2^n \* 2^n
* The Tromino is L shaped
* There is an empty square that will not be filled

## Description of steps ( dynamic programming):

We will start by picking an arbitrary square to indicate as a dead zone, then we will divide the remaining of our grid into 4 sub grids as quadrants and try to find the optimal solution to each one which will lead to the optimal solution of the main problem ‎[7].

1. Take input of the hole position in the grid
2. Divide the grid into 4 sub quadrants
3. Repeat the division on each quadrant until we get a grid of 2\*2
4. Check the holes condition in the quadrant
5. Eliminate the impossible cases ( 2 holes in the same quadrant)
6. Save the possible solutions ( 1 hole in the quadrant)
7. Return the solution of each individual quadrant to the bigger parent
8. Repeat the solution search until we get the solution of the whole grid

## Chart, box and whisker chart Description automatically generatedA picture containing text, crossword puzzle, colorful, mosaic Description automatically generatedA picture containing text, crossword puzzle, colorful, black Description automatically generatedA picture containing text, crossword puzzle, first-aid kit, colorful Description automatically generatedChart, box and whisker chart Description automatically generatedEx:

4\*4 Square

Figure 15:first Iteration Tromino Tiling

Figure 16: third Iteration Tromino Tiling

Figure 17: Second Iteration Tromino Tiling

Figure 18: Fourth Iteration Tromino Tiling

Figure 19: Impossible solutions for Tromino Tiling

## Pseudo Code

//Input : size of grid , hole position

//Output: Grid of trominos and a 2\*2 hole

Tile(size,x,y)

If ( size = 2) then

Check for number of holes

If (n\_holes >=2) then

Ignore solution

Else

Place Tromino around hole

Return solution

tile(n / 2, x, y + n / 2);

tile(n / 2, x, y);

tile(n / 2, x + n / 2, y);

tile(n / 2, x + n / 2, y + n / 2);

## Complexity Analysis:

Since we will divide each sub grid into 4 more sub grids our complexity for a problem of grid size 2n\*2n will be **n2** where the recurrence relation is T(n) = 4\*T(n/2).

## Sample output:

Text

Description automatically generated

Figure 20: Task 6 output

## Description of steps (Brute Force Approach):

In this method we will have to loop on every element to check if it’s a hole and then loop on the surrounding elements to check if there is an available position to place a Tromino and finally we will have to loop on the chosen position to place the Tromino so our complexity will be **n3**

## Comparison

The brute force algorithm works on all the grid elements which causes the time complexity to be bigger while the dynamic programming method works on certain solutions while discarding the impossible ones and saving the solutions done to greatly minimize the time complexity ‎[1].

## Conclusion

The dynamic programming method is better in time complexity and can be improved to reduce the time required but it consumes more memory to store the board orientation in each iteration.

# References

1. Levitin, A. (2012). *Introduction to the design and analysis of algorithms*. Pearson.
2. Levitin, A., & Levitin, M. (2020). *Algorithmic puzzles*. Oxford University Press.
3. *3x3 magic square algorithm problem - C++ Forum*. cplusplus.com. (n.d.). http://www.cplusplus.com/forum/beginner/231292/.
4. P. Winkler, [Mathematical Puzzles: A Connoisseur's Collection](https://www.amazon.com/exec/obidos/ISBN=1568812019/ctksoftwareincA/), A K Peters, 2004, pp. 81-82 (Light bulbs in a circle)
5. Published by bitnibblebyteblog View all posts by bitnibblebyteblog, bitnibblebyteblog, P. by, Bitnibblebyteblog, bitnibblebyteblog, V. all posts by, & (required), N. (2018, June 17). *Puzzle: Counting Triangles*. Bit Nibble Byte. https://bitnibblebyteblog.wordpress.com/2018/06/17/puzzle-counting-triangles/. Task 4
6. American Communication Association WWW archives. (1997). *Choice Reviews Online*, *34*(12). https://doi.org/10.5860/choice.34sup-021
7. OnirOnir 95.8k1717 gold badges113113 silver badges233233 bronze badges, MJDMJD 59.4k3535 gold badges247247 silver badges453453 bronze badges, Takahiro WakiTakahiro Waki 2, & kiro malakkiro malak 922 bronze badges. (1962, March 1). *Proof a $2^n$ by $2^n$ board can be filled using L shaped trominoes and 1 monomino*. Mathematics Stack Exchange. https://math.stackexchange.com/questions/461252/proof-a-2n-by-2n-board-can-be-filled-using-l-shaped-trominoes-and-1-monomi.