PROJECT SUBMISSION

Program: Computer Engineering and Software Systems Program

Course Code: CSE-332

Course Name: Design and Analysis of Algorithms

Semester: Spring 2021

Students Personal Information:

NAME	ID	GROUP	SECTION
Yomna Hussien Mohamed Abd El Hamid	18P5794	2	2
Sherif Ahmed Naiem Mohamed	18P6546	2	2
Omar Mohamed Lotfy El Said	18P5606	2	2
Ilaria Refaat Ghobrial	18P3050	2	2
Laila Mohamed Mohamed Abdelfatah	18P9654	2	2

Table of Contents

I. TASK 1 3
II. TASK 2 7
➢ Task 2-A 7
➢ Task 2-B 17
III. TASK 3 22
IV. TASK 4 27
V. TASK 5 32
VI. TASK 6 36
VII. REFERENCE 50

I. TASK 1

A. Problem Description

Find the minimum number of cuts required if you have a stick n unit long that needs to be cut into n unit pieces. It is allowed to cut several stick pieces at the same time.

Greedy approach: A global (overall) optimal solution can be reached by choosing the optimal choice at each step.

Greedy algorithms work on problems for which it is true that, at every step, there is a choice that is optimal for the problem up to that step, and after the last step, the algorithm produces the optimal solution of the complete problem.

In this problem at every step, we decide the cut that results in largest number of pieces and least difference in length between all cut pieces.
Every cut result in maximum 2* number of pieces.
In other words, we want to cut every piece in half or as close to this as possible.

For e.g.
The local optimal cut for a 100-inch piece is into 2 pieces 50 inch each
(Here we reached the maximum number of pieces (2) and a difference of 0 in length so that is the best decision)

And then the 2 50-inch pieces into 4 25-inch pieces
(Here we reached the maximum number of pieces (4) and a difference of 0 in length so that is the best decision)

This leads to the following conclusion that if the length of the longest stick is even it is cut into half—> this results in max number of pieces with 0 difference in length.

How about if the stick is odd?

For the 4 25-inch pieces the local optimal is to divide each stick into 12-inch pieces and 13-inch pieces this results in 4 12-inch sticks and 4 13-inch sticks.
(Here we reached the maximum number of pieces (8) and a difference of 1 in length so that is the best decision)
12 inches are divided into 6 and 13-inch are divided into 6,7
(Here we reached the maximum number of pieces (16) and a difference of 1 in length (12 are 6-inch and 4 are 7-inch) so that is the best decision)
6 are cut into 3 and 7 into 3,4
4 are cut into 2 and 3 into 1,2

This leads to the following conclusion that if the length of the longest stick is odd it is cut into length/2 and length/2 +1 —> this results in max number of pieces with 1 difference in length.

When do we stop?

When the longest piece is equal to 1 (when a piece is 1 unit long it is no longer cut)

B. The Algorithm

```
1- Input length of the stick
2-initialize a counter with 0
3-if the length of the longest stick is 1 break
4-else if the length of the longest stick is even
        length = length/2
        and increase counter by 1
5- else if the length of the longest stick is odd
        length=length/2 +1
        and increase counter by 1
```

C. Pseudo-Code

```
ALGORITHM cutStick (integer length)
//Input: length of the stick
//Output: minimum number of cuts required to cut a n unit long stick into n unit pieces
Counter<—0
while (1) do
        if length =1
                break
        else if length is even
                length=length/2
        else if length is odd
                length=length/2+1
Counter=Counter+1
End while
```

D. Code in Java

```java
package algo;
import java.util.Scanner;
public class ALGO {
    public static void main(String[] args) {
        System.out.print("input length of stick: ");
        Scanner input = new Scanner (System.in);
        int length = input.nextInt();
        int counter =0;
        while(true){
            if (length==1){
                break;
            }
            else if (length%2==0){
                length=length/2;
            }
            else if (length%2==1){
                length=(length/2)+1; }
            counter++; }
        System.out.print("minimum number of cuts is "+counter);
        System.out.print(System.lineSeparator());}}
```

E. Complexity Analysis for the Algorithm

The main operation here is the comparison it is done 1-3 times in each iteration
We keep on dividing the length by 2 with each iteration
So we can conclude that we iterate the while loop logN+1 times where N is the length of the original stick.
We ignore constants and multiplication of constants, so the complexity of this algorithm is **log N**

F. Comparison with Another Algorithm

Another algorithm that can be used to solve this task is the divide and conquer approach.
The divide and conquer approach involves dividing the problem into subproblems and solving smaller instances recursively.
The pseudocode for the divide and conquer will be as follows:

```
ALGORITHM cutStick (integer length)
//Input: length of the stick
//Output: minimum number of cuts required to cut a n unit long stick into n
unit pieces
Function divAndCon (integer length)
      if length =1
            return 0
      return 1+divAndCon((length+1)/2)
End while
```

Here we divide the problem each time into half until we reach the smallest instance then we collect the count.
Complexity of divide and conquer is :
Recurrence relation: $x(n) = x((n+1)/2) + 1 \quad x(1)=1$

$x(2^y)=x(2^(y-1)+0.5)+1$

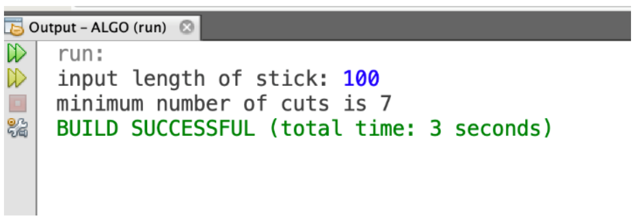
at k $\quad x(2^y)=x(2^(y-k))+0.5*k+1$

at k=y $\quad x(1)+y/2+1$

y=log n

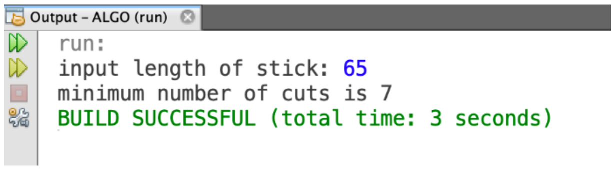
ignoring constants complexity of divide and conquer is same as greedy but they are both different approaches to solve the problem

G. Sample Output


Output – ALGO (run)

```
run:
input length of stick: 100
minimum number of cuts is 7
BUILD SUCCESSFUL (total time: 3 seconds)
```

Steps of solution:
Length = 100
1: 50 + 50
2: 25+ 25 + 25 + 25
3: 12 + 13 + 12 + 13 + 12 + 13+ 12 + 13
4: 6 + 6 + 6 + 7 + 6 + 6+ 6 + 7 + 6 + 6 + 6 + 7 + 6 + 6 + 6 + 7
5: 28 *3 units, 4 *4 units
6: 28 * 1 unit, 36 * 2 units
7: 100 *1 unit
Minimum number of cuts is 7


Output – ALGO (run)

```
run:
input length of stick: 65
minimum number of cuts is 7
BUILD SUCCESSFUL (total time: 3 seconds)
```

Steps of solution:
Length = 65
1: 32 + 33
2: 16+ 16 + 16 + 17
3: 8 + 8 + 8 + 8 + 8 + 8+ 8 + 9
4: 4 + 4 + 4 + 4 +4+ 4+ 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 5
5: 31 *2 units, 1 *3 units
6: 63 * 1 unit, 1 * 2 units
7: 65 *1 unit
Minimum number of cuts is 7

H. Conclusion

This task was performed using the greedy approach to get the minimum number of cuts needed to reach n 1-unit pieces. This was done through maximizing in length in every step.

II. TASK 2

➢ Task 2-A

A. Detailed Assumptions

Here we assume that the user inputs value n then the program creates a matrix n x n and fills it with numbers 1 to 9 inclusive then outputs the matrix and how many magic squares are there.

B. Problem Description

This task is divided into two parts in the first part of the question we need to find the **maximum** number of magic squares that could be found in a n x n matrix. n could be any integer greater than 3.

Dynamic programming approach: it is used for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

How are we going to apply this here?

In this problem we start by filling the first 3 columns in the first 3 rows by numbers from 1 to 9 such that this 3 x 3 square is a magic square. After storing (this makes it dynamic as we are going to use this matrix later) the 3 x3 matrix we use it to start filling the n x n matrix according to whether this value of n is odd or even. We fill the n x n matrix on phases later phases use earlier ones (this as well makes the program dynamic).

2	9	4
7	5	3
6	1	8

The 3x3 matrix

If this value of n is odd?

We copy the 2nd column of the 3x3 matrix in the 4th of n x n matrix and the 1st in the 5th then 2nd then 3rd then 2nd then 1st etc. After finishing columns, we copy 2nd row of n x n matrix into 4th then 1st into 5th then 2nd then 3rd then 2nd then 1st etc.
The matrix below illustrates this

Here we have maximum 4 magic squares (the number of magic squares = no. of 5)

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

Here we have max 9 magic squares

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
7	5	3	5	7	5	3
6	1	8	1	6	1	8

If this value of n is even?

n=4 is a special case there is no way it can have more than one magic square no matter the arrangement of numbers.

n=6 is treated as a multiple of 3, so here we discuss any even integer >= 8
we deal with the part of the matrix that is (n-3) x (n-3) the same way as the odd matrix we save it then in the last 3 columns we repeat the first 3 and same for the rows.
e.g., n=8, max number of magic squares = 9

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

e.g., n=10, max number of magic squares = 16

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

C. The Algorithm

```
1- input the value of n
2- if n is odd
        1. Generate magic square with size 3*3
        2. Set all flags to zero
        3. Fill the first 3*3 cells in the matrix n*n with magic square.
        4. m=3
        5. If m is odd
            • copy the values of the middle column in the magic square to the first 3 cells in the
              column of index m
            • increment number of magic square
        6. else if vertical flag =0
            • copy the values of the first column in the magic square to the first 3 cells in the
              column of index m
            • vertical flag =1
        7. else if vertical flag =1
            • copy the values of the third column in the magic square to the first 3 cells in the
              column of index m
            • vertical flag = 0
        8. Increment m
        9.if m =n
                move to steps 2.11
        10. Else repeat from step 2.5
```

11. k=3
12. If k is odd
    - copy second row in the matrix n*n to row of index k in the same matrix
    - increment number of magic square by n/2
13. Else if horizontal flag =0
    - copy first row in the matrix n*n to row of index k in the same matrix
    - horizontal flag=1
14. Else if horizontal flag =1
    - copy third row in the matrix n*n to row of index k in the same matrix
    - horizontal flag =0
15. Increment k
16.if k =n
    move to step 4
17. Else repeat from step 2.12

3- else n is even
1. Generate magic square
2. If n=4
    - repeat from step 2.1 with n = 3
    - copy first 3 cells of first column in matrix n*n in first 3 cells of column of index 3
    - copy first 3 cells of first row in matrix n*n in first 3 cells of row of index 3
    - move to step
3. Else
    - repeat from step 2.1 with n = n-3
    - fill the values of the last 3 columns with the values of the first 3 columns in matrix n*n
    - increment number of magic square by (n-3/2)
    - fill the values of the last 3 rows with the values of the first 3 rows in matrix n*n
    - increment number of magic square by (n/2)-1

4- print max number of magic square

D. Pseudo-Code

```
ALGORITHM MagicSquare (integer n)
//Input: integer n that gives the dimensions of the n x n matrix
//Output: The matrix and the maximum number of magic squares that could be found.
magicSquare[3][3]
maxNoMagicSquare<—1
Function printMatrix( integer **matrix, integer n ){
        for k <— 0 to k = n-1 do
          for m <— 0 to m=n-1 do
            Print ( matrix[k][m])
          End for
         End for
```

```
  End function
Function generateMagicSquare(){
        i <— 2
        j <—1
        for num <— 1 to num = 9 do
          if i = 3 And j = -1
            j <— 0
            i <— 1
          else
                if i =3
                      i <— 0
                if j < 0
                      j <—2
           if magicSquare[i][j]
             i -= 2
             j++
             Continue
           else
             magicSquare[i][j] <— num++
           i++
           j--
      End function
Function nOdd (integer **matrix, integer n){
        bool verticalFlag<—0
        bool horizontalFlag<—0
         generateMagicSquare()
         for i <—0 to 3 do
           for j<—0 to 3 do
             matrix[I][j]<—magicSquare[i][j]
           End for
         End for
         for m<—3 to n-1 do
           if  m%2=1
             matrix[0][m]|<—magicSquare[0][1]
             matrix[1][m]<—magicSquare[1][1]
             matrix[2][m]<—magicSquare[2][1]
             maxNoMagicSquare++

           else if verticalFlag=0
             matrix[0][m]<—magicSquare[0][0]
             matrix[1][m]<—magicSquare[1][0]
             matrix[2][m]<—magicSquare[2][0]
             verticalFlag<—1

           else if  verticalFlag=1
             matrix[0][m]<—magicSquare[0][2]
            matrix[1][m]<—magicSquare[1][2]
            matrix[2][m]<—magicSquare[2][2]
            verticalFlag<—0
        End for
        for k<—3 to n-1 do
            if  k%2=1
                for j<—0 to n-1 do
                    matrix[k][j]<—matrix[1][j]
                End for
                maxNoMagicSquare+=n/2

            else if  horizontalFlag=0
                for j<—0 to n-1 do
                    matrix[k][j]<—matrix[0][j]
                End for
                horizontalFlag<—1

            else if horizontalFlag=1
                for j<—0 to n-1 do
                    matrix[k][j]<—matrix[2][j]
                End for
                horizontalFlag<—0
        End for
    End function

Function nEven (integer **matrix,integer n)
        if n=4
            nOdd(matrix,3)
            for i<—0 to 3 do
                matrix[I][3]<—matrix[i][0]
            End for
            for j<—0 to 3 do
                matrix[3][j]<—matrix[0][j]
            End for
            Return
        nOdd(matrix,n-3)
        for j<—n-3 to n-1 do
            for I<—0 to n-4 do
                matrix[I][j]<—matrix[i][j-(n-3)]
            End for
        End for
        maxNoMagicSquare+=((n-3)/2)
        for i<—n-3 to n-1 do
            for j<—0 to n-1 do
                matrix[I][j]<—matrix[i-(n-3)][j]
            End for
        End for
        maxNoMagicSquare+=((n/2)-1)
    End function

    Integer ** matrix;
    matrix = new integer * [n];
    for i<—0 to n-1do
         matrix[i] <— new int[n]
    End for
    if n%2=1
         nOdd(matrix,n)
     else
          nEven(matrix,n)
     print(matrix,n)
     print("Maximum number of magic squares that could be found is: + maxNoMagicSquare)
```

E. Code in C++

```
#include <iostream>
#include<iomanip>
using namespace std;
int magicSquare[3][3];
int maxNoMagicSquare=1;
void print(int **matrix,int n){
    for (int k = 0;k < n;k++) {
        for (int m = 0;m < n;m++) {
            cout << "    " << matrix[k][m];
        }
        cout << endl;
    }
}
void generateMagicSquare(){
    int i = 2;
    int j = 1;
    for (int num = 1; num <= 3 * 3;) {
        if (i == 3 && j == -1)
        {   j = 0;
            i = 1;
        }
        else {
            if (i == 3)
                i = 0;
            if (j < 0)
                j =2;
        }
        if (magicSquare[i][j])
        {
            i -= 2;
            j++;
            continue;
        }
        else
            magicSquare[i][j] = num++;
        i++;
        j--;
    }
}
void nOdd (int**matrix,int n){
    bool verticalFlag=0;
    bool horizontalFlag=0;
    generateMagicSquare();
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            matrix[i][j]=magicSquare[i][j];
        }
    }
    for (int m=3;m<n;m++){
        if (m%2==1){
            matrix[0][m]=magicSquare[0][1];
            matrix[1][m]=magicSquare[1][1];
            matrix[2][m]=magicSquare[2][1];
            maxNoMagicSquare++;
        }
        else if (verticalFlag==0){
            matrix[0][m]=magicSquare[0][0];
            matrix[1][m]=magicSquare[1][0];
            matrix[2][m]=magicSquare[2][0];
            verticalFlag=1;
        }
        else if (verticalFlag==1){
            matrix[0][m]=magicSquare[0][2];
            matrix[1][m]=magicSquare[1][2];
            matrix[2][m]=magicSquare[2][2];
            verticalFlag=0;
        }
    }
    for (int k=3;k<n;k++){
        if (k%2==1){
            for(int j=0;j<n;j++){
                matrix[k][j]=matrix[1][j];
            }
            maxNoMagicSquare+=n/2;
        }
        else if (horizontalFlag==0){
            for(int j=0;j<n;j++){
                matrix[k][j]=matrix[0][j];
            }
            horizontalFlag=1;
        }
        else if (horizontalFlag==1){
            for(int j=0;j<n;j++){
                matrix[k][j]=matrix[2][j];
            }
            horizontalFlag=0;
        }
    }
}
```

```
void nEven (int**matrix,int n){
    if(n==4){
        nOdd(matrix,3);
        for (int i=0;i<4;i++){
            matrix[i][3]=matrix[i][0];
        }
        for (int j=0;j<4;j++){
            matrix[3][j]=matrix[0][j];
        }
        return;
    }
    nOdd(matrix,n-3);
    for (int j=n-3;j<n;j++){
        for(int i=0;i<n-3;i++){
            matrix[i][j]=matrix[i][j-(n-3)];
        }
    }
    maxNoMagicSquare+=((n-3)/2);
    for (int i=n-3;i<n;i++){
        for(int j=0;j<n;j++){
            matrix[i][j]=matrix[i-(n-3)][j];
        }
    }
    maxNoMagicSquare+=((n/2)-1);
}
int main(int argc, const char * argv[]) {
    int n;
    cout<<"please insert value of n to be tested"<<endl;
    cin>>n;
    int ** matrix;
    matrix = new int* [n];
    for (int i = 0;i < n;i++) {
        matrix[i] = new int[n];
    }
    if (n%2==1)
        nOdd(matrix,n);
    else
        nEven(matrix,n);
    print(matrix,n);
    cout<<"Maximum number of magic squares that could be found is:
"<<maxNoMagicSquare<<endl;
    return 0;
}
```

F. Complexity Analysis for the Algorithm

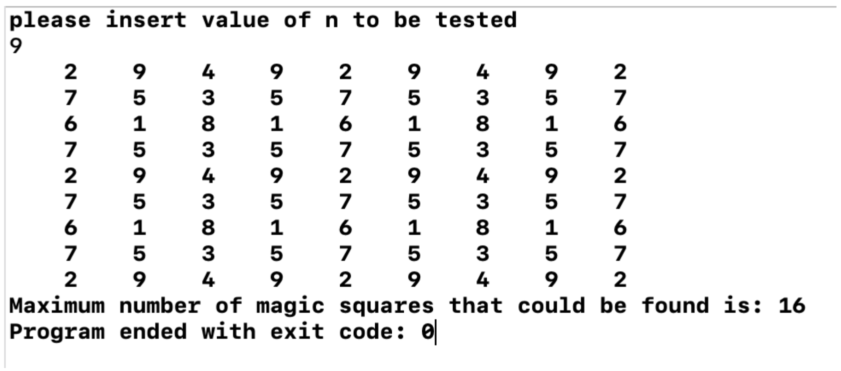
The main operation is the copying operation. After analyzing we find that the main function here is the nOdd function. This function’s complexity is (n-3)*3+n(n-3) as we have two for loops. After ignoring constants, we find that the complexity is O(n^2).

G. Comparison with Another Algorithm

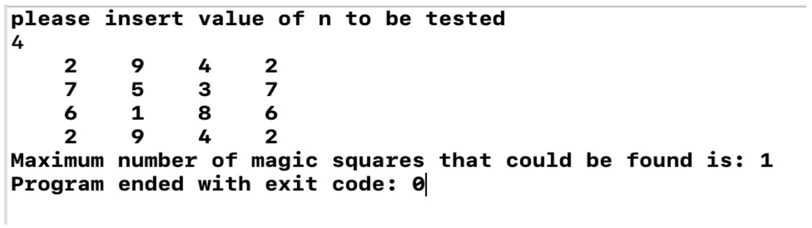
Another algorithm is the brute force algorithm. This approach is based on trying all possible combinations of arrangement of numbers then checking on how many magic squares there could be this would result in a complexity of around O(n^3)

H. Sample Output

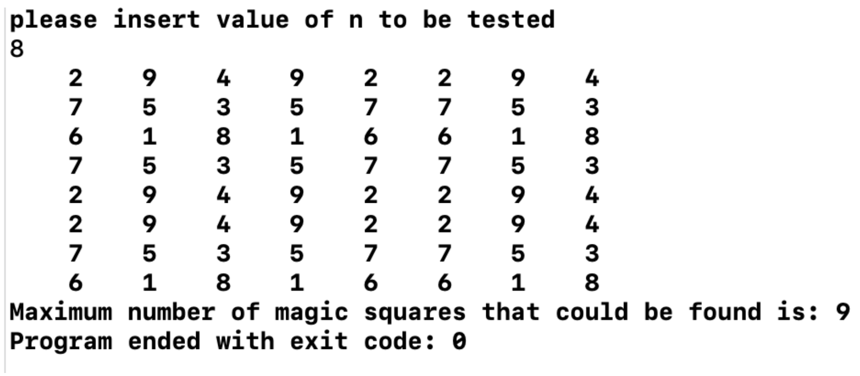
Here the value to be tested is 9 the matrix is filled with the sequence described above so we find that the output result is the same as the one in our study which proves that the program works successfully with odd numbers.
The output below proves that the code works with even values of n



```
please insert value of n to be tested
9
    2    9    4    9    2    9    4    9    2
    7    5    3    5    7    5    3    5    7
    6    1    8    1    6    1    8    1    6
    7    5    3    5    7    5    3    5    7
    2    9    4    9    2    9    4    9    2
    7    5    3    5    7    5    3    5    7
    6    1    8    1    6    1    8    1    6
    7    5    3    5    7    5    3    5    7
    2    9    4    9    2    9    4    9    2
Maximum number of magic squares that could be found is: 16
Program ended with exit code: 0
```



```
please insert value of n to be tested
4
    2    9    4    2
    7    5    3    7
    6    1    8    6
    2    9    4    2
Maximum number of magic squares that could be found is: 1
Program ended with exit code: 0
```



```
please insert value of n to be tested
8
    2    9    4    9    2    2    9    4
    7    5    3    5    7    7    5    3
    6    1    8    1    6    6    1    8
    7    5    3    5    7    7    5    3
    2    9    4    9    2    2    9    4
    2    9    4    9    2    2    9    4
    7    5    3    5    7    7    5    3
    6    1    8    1    6    6    1    8
Maximum number of magic squares that could be found is: 9
Program ended with exit code: 0
```

➢ Task 2-B

A. Detailed Assumptions

Show all values in a n*n table with numbers from 1 to 9 so that every 3*3 matrix is a pseudo magic square.

In this problem we take the size(n) of the table from the user then we generate a magic square of size 3*3 then we fill the n*n table with the values of the magic square in an alternating way so that the result is a matrix filled with pseudo magic squares.

B. The Algorithm

1. Input size of table from user(n).
2. Generate magic square in a matrix of size 3*3.
3. Create matrix with size n*n.
4. Create a for loop to traverse the matrix.
5. If the value of the index is divisible by 3 we assign value of row and column to 0.
6. If the (value of index -1) is divisible by 3 we assign the value of row and column to 1.
7. f the (value of index 2) is divisible by 3 we assign the value of row and column to 2.
8. Copy the element with the index reached from the magic square generated to the place in the matrix.
9. Print the matrix n*n.

C. Pseudo-Code

```
magicSquare[3][3]
Function generateMagicSquare()
   int i <— 2
   int j <— 1
   for num<—1 to 9 do
      if i = 3 And j = -1
        j <— 0
        i <— 1
      else
         if i = 3
            i <— 0

         if j < 0
            j <—2
      if magicSquare[i][j]
         i -= 2
```

```
            j++
            continue
        else
            magicSquare[i][j] = num++
        i++
        j--
End function

Function pseudoMagic(integer n)
    generateMagicSquare()
    Integer ** matrix
    matrix = new integer* [n]
    for i<—0 to n-1 do
        matrix[i] = new int[n]
    End for
    for k<—0 to n-1 do
            for m<—0 to n-1 do
                i <— k
                j <— m
                if  i % 3 = 0
                    i <— 0
                else if (i - 1) % 3 = 0
                     i <— 1
                else if ( i - 2) % 3 = 0
                     i <— 2
                if j % 3 = 0
                    j <— 0
                else if (j - 1) % 3 = 0
                    j <- 1
                else if (j - 2) % 3 = 0
                    j <— 2
                matrix[k][m] <— magicSquare[i][j]


        for k<—0 to n-1 do
            for m<—0 to n-1
                Print ( matrix[k][m])
            End for
         End for
End function
```

D. Code in C++

```
#include <iostream>
#include<iomanip>
using namespace std;
int magicSquare[3][3];
void generateMagicSquare(){
    int i = 2;
    int j = 1;
    for (int num = 1; num <= 9;) {
        if (i == 3 && j == -1)
        {   j = 0;
            i = 1;
        }
        else {
            if (i == 3)
                i = 0;

            if (j < 0)
                j =2;
        }
        if (magicSquare[i][j])
        {
            i -= 2;
            j++;
            continue;
        }
        else
            magicSquare[i][j] = num++;
        i++;
        j--;
    }
}
void pseudoMagic(int n){
    generateMagicSquare();
    int ** matrix;
    matrix = new int* [n];
    for (int i = 0;i < n;i++) {
        matrix[i] = new int[n];
    }
    for (int k = 0; k < n ;k++){
            for (int m = 0; m < n ; m++) {
                int i = k;
                int j = m;
                if ( i % 3 == 0) {
                    i = 0;
                }
                else if ( (i - 1) % 3 == 0) {
                     i = 1;
                }
                else if (( i - 2) % 3 == 0) {
                     i = 2;
                }
                if (j % 3 == 0) {
                    j = 0;
                }
                else if ((j - 1) % 3 == 0) {
                    j = 1;
                }
                else if ((j - 2) % 3 == 0) {
                    j = 2;
                }
                matrix[k][m] = magicSquare[i][j];
            }
        }
        //print
        for (int k = 0;k < n;k++) {
            for (int m = 0;m < n;m++) {
                cout << "     " << matrix[k][m];
            }
            cout << endl;
        }
}
int main(){
    int n;
    cout<<"please insert value of n to be tested"<<endl;
    cin >> n;

    pseudoMagic(n);
    return 0;
}
```

E. Complexity Analysis for the Algorithm

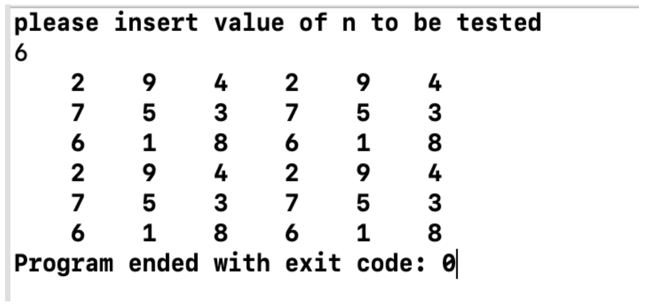
The main operation here is the comparison. This comparison is done 2-6 times per loop the loop loops n times in another for loop that loops n times so complexity here is O(n^2)

F. Comparison with Another Algorithm

Another algorithm that could be used is the brute force algorithm this is going to include distributing numbers in a random manner and we keep on rearranging them until every 3*3 square is a pseudo magic square this results in a very large complexity which proves that dynamic programming is a better approach to solve the problem.

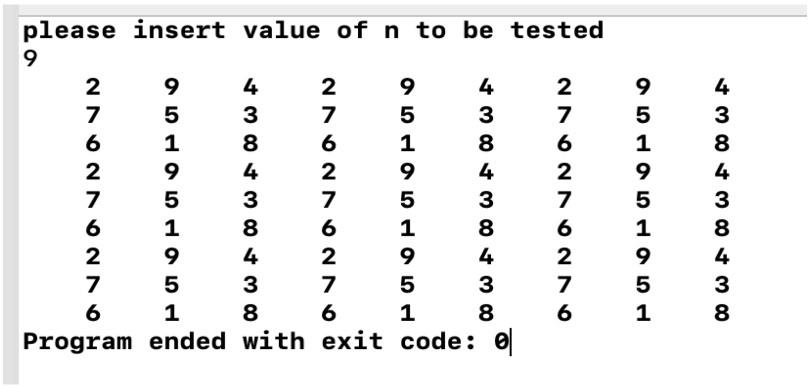
G. Sample Output

Here the value tested is n=6 and the result shows that every 3*3 square is pseudomagic as they have the same sum for rows and columns



```
please insert value of n to be tested
6
    2    9    4    2    9    4
    7    5    3    7    5    3
    6    1    8    6    1    8
    2    9    4    2    9    4
    7    5    3    7    5    3
    6    1    8    6    1    8
Program ended with exit code: 0
```

Here the value tested is n=9 and the result shows that every 3*3 square is pseudomagic as they have the same sum for rows and columns.



```
please insert value of n to be tested
9
    2    9    4    2    9    4    2    9    4
    7    5    3    7    5    3    7    5    3
    6    1    8    6    1    8    6    1    8
    2    9    4    2    9    4    2    9    4
    7    5    3    7    5    3    7    5    3
    6    1    8    6    1    8    6    1    8
    2    9    4    2    9    4    2    9    4
    7    5    3    7    5    3    7    5    3
    6    1    8    6    1    8    6    1    8
Program ended with exit code: 0
```

H. Conclusion

This task used the approach of dynamic programming as the matrix was filled every time using preserved values in the matrix itself or in another matrix such as the magic square matrix (3*3)

III. TASK 3

A. Problem Description

Assuming the lights are numbered from 1 to n. Because toggling the on/off states of three lights: its own and the two lights adjacent to it, we have two cases.

1- If N the number of lightbulbs is divisible by 3. In this case all we have to do is toggle of every 3k+1 light bulb where k starts from 0 to the switch of N-2 bulb. This results in a total of **N/3** switch flips.

2- If N is not divisible by 3. In this case we also do case one to flip all the multiple of 3. But there will be one or two bulbs that are turned off. While toggling them another one or two bulbs will be turned off. So in this case we have to go toggle each one of the light bulbs once. This results in a total of **N** switch flips.

Iterative improvement:

Iterative improvement approach is built on starting at a random configuration and repeatedly consider various moves; accept some and reject some and when you are stuck restart.

B. The Algorithm

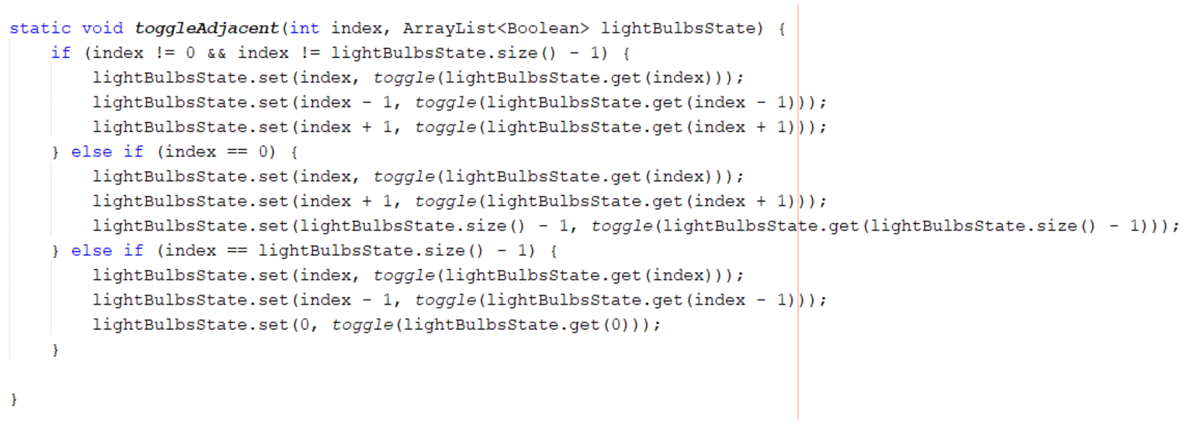
```
1.  Input number of number of bulbs=n
2.  Initialize an arraylist lightBulbsState of n items
3.  While (all values in lightbulbs State==false)
4.  If (n is a multiple of 3)
5.  For (from i=0 => 3 * a + 1 <= n – 2)
6.  Toggle(3*i+1)
7.  end for loop
8.  else
9.  for (from k=0 => n-1)
10. toggle(k)
11. end for loop
12. end if
13. end while loop
```

C. Pseudo-Code

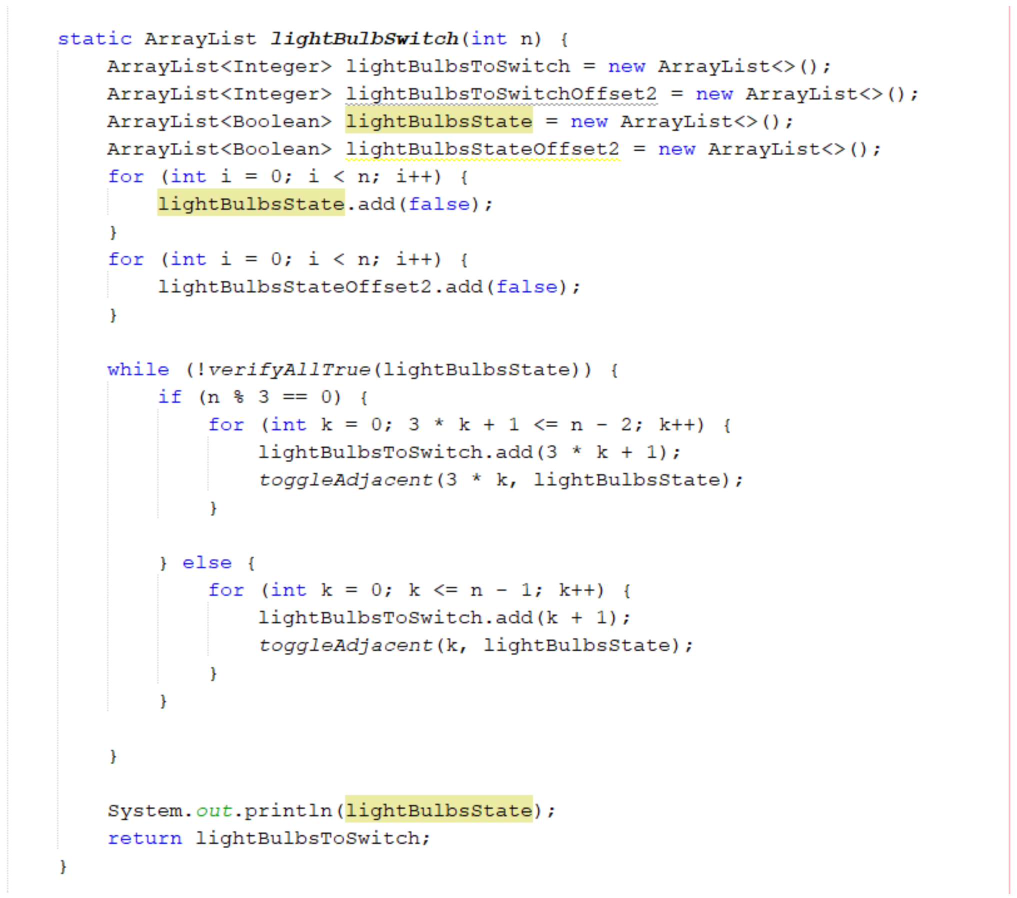
```
toggleAdjacent(int index, ArrayList<Boolean> lightBulbsState) {
    if index != 0 and index != (lightBulbsState.size() – 1) {
        lightBulbsState.set(index, toggle(lightBulbsState.get(index)))
        lightBulbsState.set(index - 1, toggle(lightBulbsState.get(index - 1)))
```

```
            lightBulbsState.set(index + 1, toggle(lightBulbsState.get(index + 1)))
        } else if index == 0 {
            lightBulbsState.set(index, toggle(lightBulbsState.get(index)));
            lightBulbsState.set(index + 1, toggle(lightBulbsState.get(index + 1)));
            lightBulbsState.set(lightBulbsState.size() - 1,
toggle(lightBulbsState.get(lightBulbsState.size() - 1)))
        } else if index == (lightBulbsState.size() – 1) {
            lightBulbsState.set(index, toggle(lightBulbsState.get(index)))
            lightBulbsState.set(index - 1, toggle(lightBulbsState.get(index - 1)))
            lightBulbsState.set(0, toggle(lightBulbsState.get(0)))
        }}
  lightBulbSwitch(int n) {
        ArrayList<Integer> lightBulbsToSwitch = new ArrayList<>();
        ArrayList<Boolean> lightBulbsState = new ArrayList<>();
        for (int i = 0; i < numberOfBulbs; i++) {
            lightBulbsState.add(false);
        }
        while (!verifyAllTrue(lightBulbsState)) {
            if numberOfBulbs % 3 == 0 {
                for (from k = 0 to 3 * k + 1 <= n - 2) {
                    lightBulbsToSwitch.add(3 * k + 1);
                    toggleAdjacent(3 * k, lightBulbsState);
                }
            } else {
                for ( from k = 0 to k <= numberOfBulbs - 1) {
                    lightBulbsToSwitch.add(k + 1);
                    toggleAdjacent(k, lightBulbsState);
                }
            }
        }
print(lightBulbsState);
        return lightBulbsToSwitch;
    }
static boolean toggle(boolean state) {
        if (state == false) {
            state = true;
        } else {
            state = false;
        }
        return state;
    }
     verifyAllTrue(ArrayList<Boolean> lightBulbsState) {

        for (from i = 0 to i < lightBulbsState.size()) {
            if (lightBulbsState.get(i) == false) {
                return false;
            }
        }
        return true;
    }
}
```

D. Code in Java



```java
static void toggleAdjacent(int index, ArrayList<Boolean> lightBulbsState) {
    if (index != 0 && index != lightBulbsState.size() - 1) {
        lightBulbsState.set(index, toggle(lightBulbsState.get(index)));
        lightBulbsState.set(index - 1, toggle(lightBulbsState.get(index - 1)));
        lightBulbsState.set(index + 1, toggle(lightBulbsState.get(index + 1)));
    } else if (index == 0) {
        lightBulbsState.set(index, toggle(lightBulbsState.get(index)));
        lightBulbsState.set(index + 1, toggle(lightBulbsState.get(index + 1)));
        lightBulbsState.set(lightBulbsState.size() - 1, toggle(lightBulbsState.get(lightBulbsState.size() - 1)));
    } else if (index == lightBulbsState.size() - 1) {
        lightBulbsState.set(index, toggle(lightBulbsState.get(index)));
        lightBulbsState.set(index - 1, toggle(lightBulbsState.get(index - 1)));
        lightBulbsState.set(0, toggle(lightBulbsState.get(0)));
    }

}
```



```java
static ArrayList lightBulbSwitch(int n) {
    ArrayList<Integer> lightBulbsToSwitch = new ArrayList<>();
    ArrayList<Integer> lightBulbsToSwitchOffset2 = new ArrayList<>();
    ArrayList<Boolean> lightBulbsState = new ArrayList<>();
    ArrayList<Boolean> lightBulbsStateOffset2 = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        lightBulbsState.add(false);
    }
    for (int i = 0; i < n; i++) {
        lightBulbsStateOffset2.add(false);
    }

    while (!verifyAllTrue(lightBulbsState)) {
        if (n % 3 == 0) {
            for (int k = 0; 3 * k + 1 <= n - 2; k++) {
                lightBulbsToSwitch.add(3 * k + 1);
                toggleAdjacent(3 * k, lightBulbsState);
            }

        } else {
            for (int k = 0; k <= n - 1; k++) {
                lightBulbsToSwitch.add(k + 1);
                toggleAdjacent(k, lightBulbsState);
            }
        }

    }

    System.out.println(lightBulbsState);
    return lightBulbsToSwitch;
}
```

E. Complexity Analysis for the Algorithm

The main operation is the light switch toggle which depends on the number of lightbulbs (N) if the number of lightbulbs is a multiple of 3 then the complexity will be Θ(N*N/3). If the number of light bulbs is not a multiple of 3 the complexity will be Θ(N*N). So, the average complexity is O(n^2).

F. Comparison with Another Algorithm

We will toggle all light bulbs. With a stopping condition that all the light bulbs are turned on. This technique will not give us the optimal solution because it will not give us the minimum switches to toggle. To have the optimal solution by a brute force a:

First, we need to check if did not skip any light bulbs the number of switches we toggled.

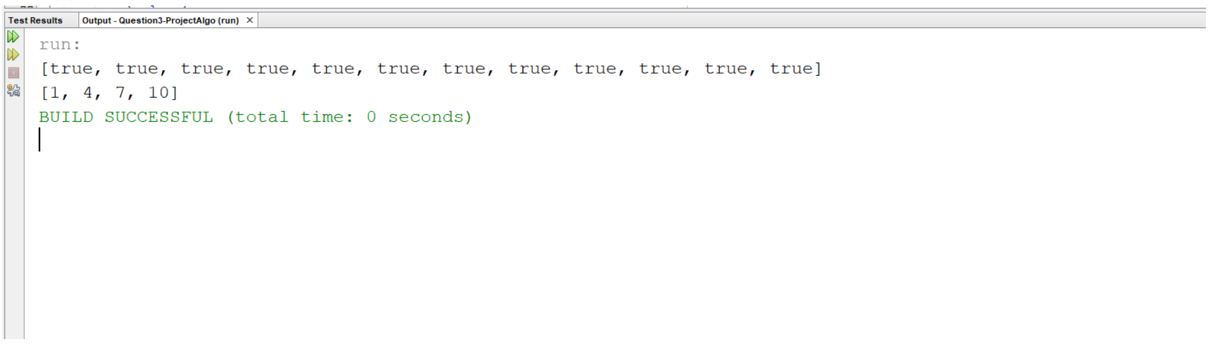
Second, we check the number of the switches we toggled if we skipped one light bulb.

Each time we iterate we add one to the offset till n-2. Finally, we compare between the number of light bulbs we switched with every offset and select the minimum number of switches.

The complexity of the brute force technique is O(n^3).

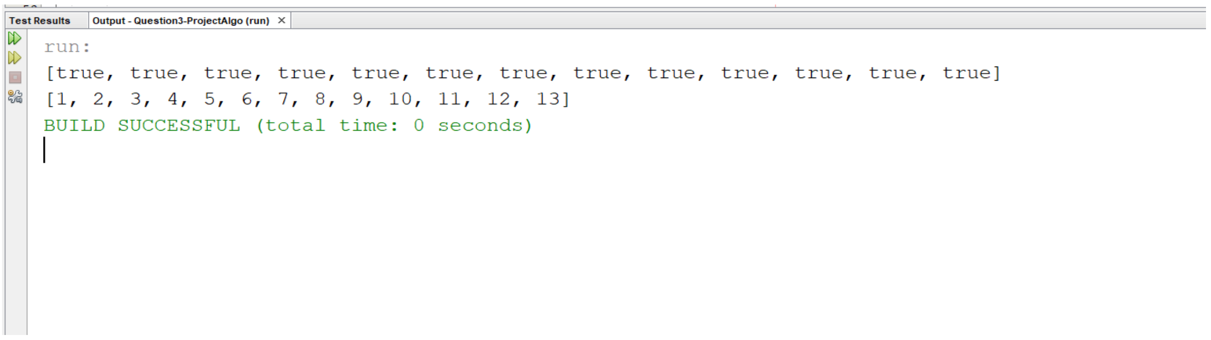
G. Sample Output

If n =12



```
run:
[true, true, true, true, true, true, true, true, true, true, true, true]
[1, 4, 7, 10]
BUILD SUCCESSFUL (total time: 0 seconds)
```

If n =13



```
run:
[true, true, true, true, true, true, true, true, true, true, true, true, true]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
BUILD SUCCESSFUL (total time: 0 seconds)
```

H. Conclusion

We can conclude that this task was done better with higher efficiency with the iterative improvement technique than with the brute force technique.

IV. TASK 4

A. Problem Description

Given an equilateral triangle, the task is to find how many small triangles are there after N iteration, on each iteration new equilateral triangles are added all around the outside.

For example:

- In the first iteration, we start by 1 equilateral triangle as shown in figure.1
- In the second iteration, we add another three equilateral triangles on the outside of the first triangle we started with, so we now have 4 small triangles as shown in figure.2
- In the third iteration, we add another six equilateral triangles on the outside of the three triangles we added in iteration 2, so we now have 10 small triangles as shown in figure.3

In this task, we will make an algorithm that will take number of iterations we want to apply as an input and give us the total number of triangles at the end of the iterations as a result

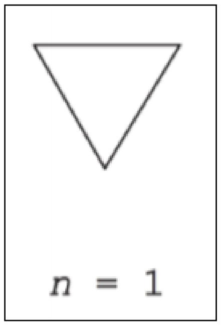


Figure 1: First Iteration

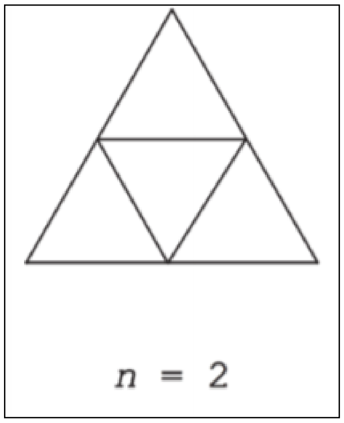


Figure 2: Second Iteration

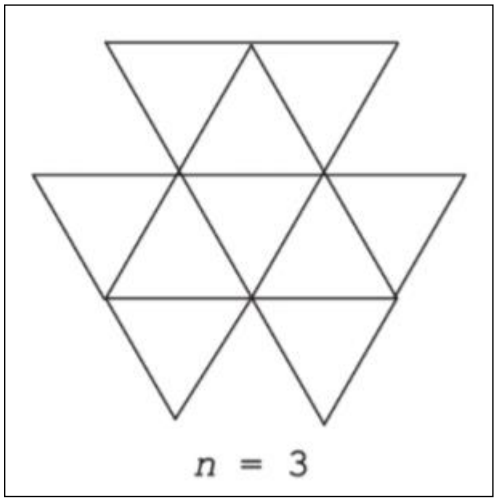


Figure 3: Third Iteration

B. Detailed Assumptions

- In the first iteration start with one equilateral triangle.
- All triangles added must be equilateral and of the same edge length.
- Triangles are added on the outside only.
- Let **n** is the number of iterations
- Let **T(n)** = total number of small triangles at iteration **n**

C. The Algorithm

To solve this problem, we will observe the first four iterations and their solutions.

In this table we will notice that in each iteration, the total number of small triangles equals to the total number of triangles of the last iteration plus (3 multiplied by (iteration number minus 1))

n	T(n)
1	1
2	1+3=4
3	4+6=10
4	10+9=19

So, the equation that satisfies this is **T(n) = T(n-1) + 3*(n-1)**

D. Pseudo-code

```
Algorithm T(n)
//Computes number of small triangles in iteration n recursively
//Input: Non-negative number
//Output: Number of small triangles
If n=1 return 1
Else return T(n) = T(n-1) + 3*(n-1)
```

E. Code in Java

Class Triangle

```java
public class Triangle {

    public static int CountTriangles(int n){

        if(n==1)
            return 1;
        return CountTriangles(n-1)+(3*(n-1));

    }
}
```

Main

```java
import java.util.*;
public class AlgoProject41 {
    public static void main(String[] args) {
        int x;
        int y;
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter number of iterations you
want");
        x=sc.nextInt();
        y=Triangle.CountTriangles(x);
        System.out.println("The number of triangles in iteration
"+x+" = "+y);
    }
}
```

F. Complexity Analysis for the Algorithm

Size: n

Basic operation: multiplication

Recurrence relation: M(n) = M(n-1) + 1
M(1) = 0

M(n) = M(n-1) + 1, M(1) = 0

= M(n-1) + 1

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

= M(n-2) + 2

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

= M(n-3) + 3

.

.

= M(n-i) + i

At i = n-1 then, M(n-(n-1)) + (n-1) = M(1) + (n-1) = 0 + n-1 = n-1

Then complexity ∈ O(n)

G. Comparison with Another Algorithm

Another algorithm to solve this puzzle is by applying this equation $\frac{3}{2}(n-1)n+1$ which can be proved by mathematical induction.

Proof:
The number of the small new triangles added on the nth iteration is equal to $3(n-1)$ for $n > 1$.
This means that the total number of small triangles after the nth iteration can be computed as

$1 + 3 \cdot 1 + 3 \cdot 2 + \cdots + 3(n-1) = 1 + 3(1 + 2 + \cdots + (n-1))$

Which equals $\sum_{n=1}^{n-1} \frac{1}{2}(n-1)n$

$=1+3\left(\frac{1}{2}n(n-1)\right) = \frac{3}{2}(n-1)n+1$

Pseudo-code for this algorithm:

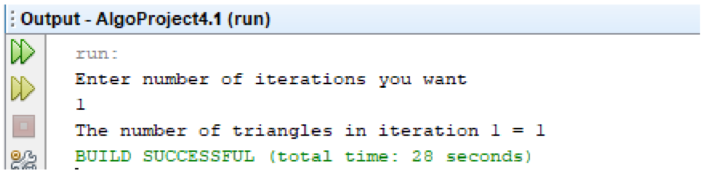
```
Function Count Triangles (integer n)
//Computes number of small triangles in iteration n directly using equation
//Input: Non-negative number
//Output: Number of small triangles
        return (((3/2)*(n-1)*n)+1);
```

since this algorithm just applies one direct equation to get the solution then, complexity ∈ O(*1*)

which will be faster than the first algorithm that uses recursive function to get the solution

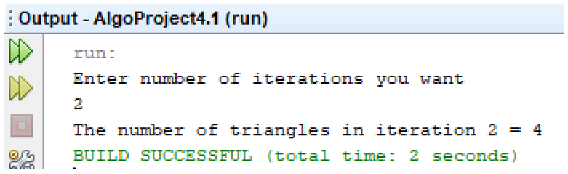
H. Sample output

If n = 1
1 + (3*(1-1))
Ans: 1



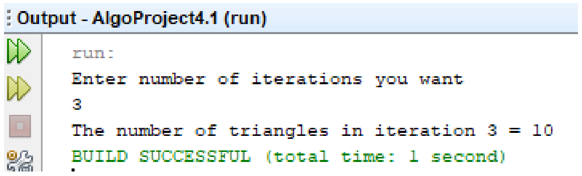
```
Output - AlgoProject4.1 (run)
run:
Enter number of iterations you want
1
The number of triangles in iteration 1 = 1
BUILD SUCCESSFUL (total time: 28 seconds)
```

If n = 2
1 + (3*(2-1))
Ans: 4



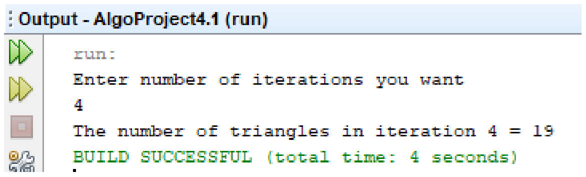
```
Output - AlgoProject4.1 (run)
run:
Enter number of iterations you want
2
The number of triangles in iteration 2 = 4
BUILD SUCCESSFUL (total time: 2 seconds)
```

If n = 3
4 + (3*(3-1))
Ans: 10



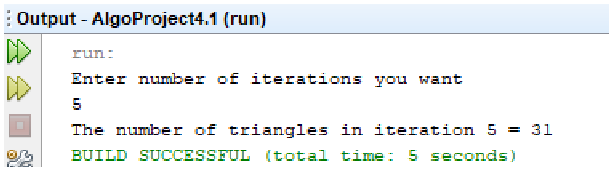
```
Output - AlgoProject4.1 (run)
run:
Enter number of iterations you want
3
The number of triangles in iteration 3 = 10
BUILD SUCCESSFUL (total time: 1 second)
```

If n = 4
10 + (3*(4-1))
Ans: 19


Output - AlgoProject4.1 (run)

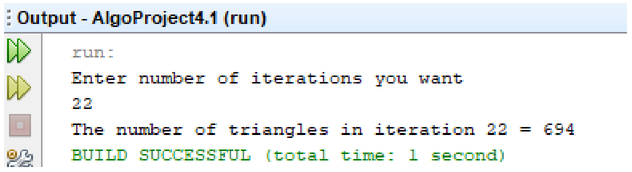
```
run:
Enter number of iterations you want
4
The number of triangles in iteration 4 = 19
BUILD SUCCESSFUL (total time: 4 seconds)
```

If n = 5
19 + (3*(5-1))
Ans: 31


Output - AlgoProject4.1 (run)

```
run:
Enter number of iterations you want
5
The number of triangles in iteration 5 = 31
BUILD SUCCESSFUL (total time: 5 seconds)
```

If n = 22
631 + (3*(22-1))
Ans: 694


Output - AlgoProject4.1 (run)

```
run:
Enter number of iterations you want
22
The number of triangles in iteration 22 = 694
BUILD SUCCESSFUL (total time: 1 second)
```

I. Conclusion

Brute force is a straight-forward technique to solve problem, it can be used in very wide variety of problem. Brute force algorithms useful for solving small size instance of problem, also it can give an important theoretical or educational purpose. Brute force algorithms are often easier to implement than a more complicated one that’s because of its simplicity and also sometimes it can be more efficient.

V. TASK 5

A. Detailed Assumptions

ASSUME: spring scale indicates the exact weight of coins being weighed.
ASSUME: A fake coin must be included in the n coins given. (If n=1 then the coin is fake)

B. Problem Description

Find the fake coin if you have n metallic coins that all look identical but one of these coins is made of lighter metal, you are given a spring scale that enable you to compare two sets of coins. You should find the fake coin in minimum number of weighing.

Divide and conquer approach: we reach the solution by dividing the problem into subproblems and solving each subproblem recursively, then combine these subproblems to reach to a solution for this problem.

In this problem we divide the coins into 3 piles, weigh two of them if they are equal we repeat the previous step with the remaining pile, else we take the lighter pile on the scale and repeat the first step.

C. The Algorithm

1) Input number of coins
2) If n=1 then it is fake
3) Else divide n into 3 parts
   a = ceiling (n/3)
   b = ceiling (n/3)
   c = n-(2*ceiling(n/3))
4) Weigh a and b
5) If scale is balanced repeat from step 1 with c as number of coins
6) Else if a is less than b repeat from step 1 with a as number of coins
7) Else repeat from step 1 with b as number of coins

D. Pseudo-code

```
Algorithm Find_Fake_Min(integer n)
//input: number of coins
//output: Fake coin found printed and the number of iterations took to find it

If n=1
        Print Coin is fake
        return

Else a=ceiling(n/3),b=ceiling(n/3), c= n-(2*ceiling(n/3))
Weigh a and b
If scale balances
        Return 1+Find_Fake_Min(c)
Else if a lighter than b
         Return 1+Find_Fake_Min(a)
Else
         Return 1+Find_Fake_Min(b)
```

E. Code in Python

```
def getmin(data, start, end, total_items):
    if total_items == 1:
        print("coin is fake")
        return 0

    elif total_items == 2:
       print("coin is fake")
      return 1
    else:
        partition = int(total_items/3)
        a_weight = sum(data[start:start+partition])
        b_weight = sum(data[start+partition:start+2*partition])
        c_weight = sum(data[start+2*partition:end])
        if a_weight == b_weight:
            return 1 + getmin(data, start+2*partition, end, end-
(start+2*partition))
        else:
            if a_weight > b_weight:
                return 1 + getmin(data, start+partition, start+2*partition,
partition)

            else:
                return 1 + getmin(data, start, start+partition, partition)

n = int(input("Input number of coins: "))
data = [1]*n
data[random.randint(0, n-1)] = 0
total_weighing = getmin(data, 0, len(data), len(data))
print("number of weightings to find fake coin is ",total_weighing
```

F. Complexity Analysis for the Algorithm

The main operation is the comparison it is done 1-3 times in each function call

We keep dividing the number of coins by 3 each function call

$$T(n) = 1 + T(\frac{n}{3})$$

a= 1, b=3, d=0
since $a = b^d$
$\Theta(n^d log\ n)$
So, the complexity of this algorithm is $\mathbf{log}\ \boldsymbol{n}$

G. Comparison with Another Algorithm

Another algorithm that can be used to solve this problem is Russian peasant multiplication

The Russian peasant multiplication is based on multiplying numbers that uses a process of halving and doubling without using multiplication operator. We double the first number and halve the second number till the number becomes 1

```
multiply(n, m)
      if n = 1 then
         return m
      else if n is odd then
         return m + multiply(n/2, m*2)
      else
         return multiply(n/2, m*2)
```

$$T(n) = m + T(\frac{n}{2})$$

a= 1, b=3, d=0
since $a = b^d$
$\Theta(n^d log\ n)$
complexity of Russian peasant multiplication is **log**

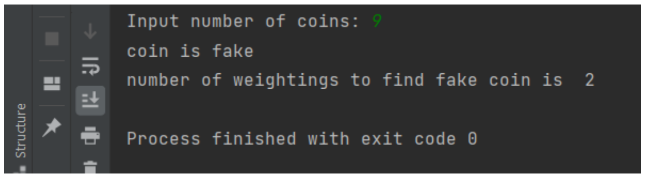
H. Sample Output

steps of Solution:

if n=9 coins, fake coin is the second coin

1. a=3, b=3 c=3
   a<b
   n=a
2. a=1,b=1,c=1
   a>b
   n=b
   =1
   fake coin

number of weightings to find fake coin is 2



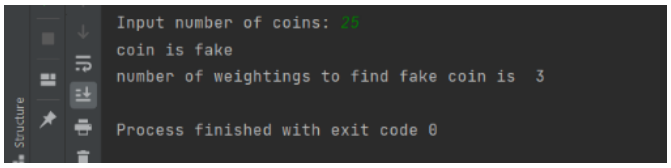
```
Input number of coins: 9
coin is fake
number of weightings to find fake coin is  2

Process finished with exit code 0
```

steps of Solution:

if n=25 coins, fake coin is the first coin

1. a=9, b=9 c=7
   a<b
   n=a
2. a=3,b=3,c=3
   a<b
   n=a
3. a=1,b=1,c=1
   a<b
   n=a
   a=1
   fake coin

number of weightings to find fake coin is 3



```
Input number of coins: 25
coin is fake
number of weightings to find fake coin is  3

Process finished with exit code 0
```

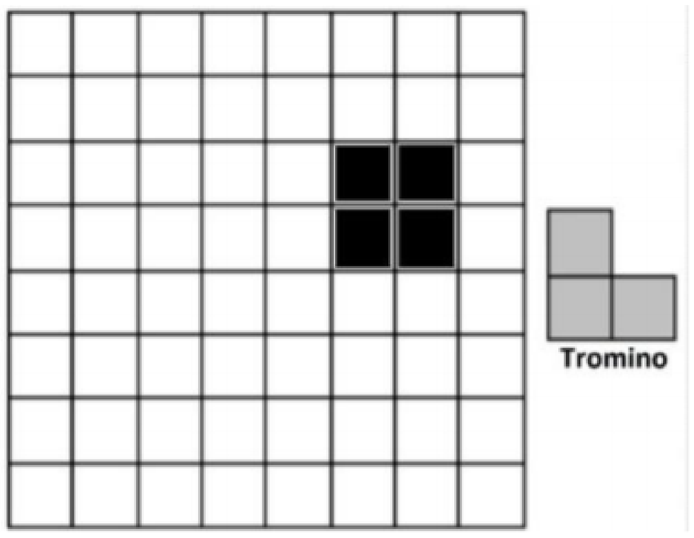
I. Conclusion

we can conclude that this task is done with higher efficiency when using the divide and conquer algorithm that leads to reaching the fake coin in minimum number of weighs.

VI. TASK 6

A. Problem Description

Given a n-by-n board where n is of form 2 ^k where k >= 1 (Basically n is a power of 2 with minimum value as 2). The board has four missing cells (of size 2 x 2). Design a dynamic programming algorithm to fill the board with nonoverlapping torminas (L shaped tiles that are 2 x 2 square with one cell of size 1×1 missing.) except for the four adjacent tiles that can be located anywhere on the board.



B. Detailed Assumptions

- The user chooses the size of the grid and the location of the four adjacent missing tiles.
- The problem is impossible to be solved if the four adjacent missing tiles are covering the half of any 2x2 squares.
- The user will be notified if he/she entered a location for four adjacent missing tiles that has no solution for the problem and will be asked to reenter another valid location.
- The filling process is done with this reparative sequence:
  1. The middle
  2. The first quadrant
  3. The second quadrant
  4. The third quadrant
  5. The fourth quadrat

C. The Algorithm

Dynamic Programming

Dynamic Programming is a technique for solving problems with overlapping subproblems. In this, we store the result of the sub-problem that is solved once for future re-use. The technique of storing sub-problem solutions is called memorization.

There are two key attributes that problems have in order for it to be solved using Dynamic Programming.

Optimal substructure - The optimal solution can be constructed from optimal solutions to its subproblems.

Overlapping sub-problems - The problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems.

If these two attributes are there, then we can use two techniques (memorization and tabulation) that both have the purpose of storing and re-using sub-problems solutions that may drastically improve performance.

D. Pseudo-code

Global variables: gridArray[128][128], statesArray[4] and trominoCounter initially 0

```
Function TILE (integer N, integer X, integer Y)
//Input: N: the size of the grid.
          X and Y: variables initially equal to zero that helps in filling the squares by order.
//Output: Filling the grid with torminas.


If the size of grid = = 2 then
   If the 2x2 square is all holes then return.
   Else trominoCounter ++
         fill the 2x2 square with a suitable tromino using trominoCounter value.

For I ← X to I < X + N
  For J ← Y to J < Y + N
    Find the location of the first quadrant square of four adjacent missing tiles.
```

```
If missing tile is in 1st quadrant
    If this state is saved in the states array then call function PLACE () to place a tromino in the
    middle of the quadrants according to the hole's location.
    Else call function MEMORIZE () to save the state in the states array.

If missing tile is in 2nd quadrant
    If this state is saved in the states array then call function PLACE () to place a tromino in the
    middle of the quadrants according to the hole's location.
    Else call function MEMORIZE () to save the state in the states array.

If missing tile is in 3rd quadrant
    If this state is saved in the states array then call function PLACE () to place a tromino in the
    middle of the quadrants according to the hole's location.
    Else call function MEMORIZE () to save the state in the states array.

If missing tile is in 4th quadrant
    If this state is saved in the states array then call function PLACE () to place a tromino in the
    middle of the quadrants according to the hole's location.
    Else call function MEMORIZE () to save the state in the states array.
 // diving the tile again into 4 quadrants
          Tile (n / 2, x, y); // 1
          Tile (n / 2, x, y + n / 2); //2
          Tile (n / 2, x + n / 2, y); // 3
          Tile (n / 2, x + n / 2, y + n / 2); //4
```

Function place (integer X1, integer Y1, integer X2, integer Y2, integer X3, integer Y3, integer R, integer X, integer N, integer Y, integer C)

//Input: N: the size of the grid.
X and Y: variables initially equal to zero that helps in filling the squares by order.
R and C: variables show the row and the column location of the first quadrant square of four adjacent missing tiles.
X1, Y1, X2, Y2, X3 and Y3: variables show the row and the column location of the three squares the construct the tromino.

//Output: place a tromino in the middle of the quadrants according to the hole's location.

```
If the 2x2 middle square is filled completely with holes then return
Else    trominoCounter++;
        gridArray[x1][y1] = trominoCounter;
        gridArray[x2][y2] = trominoCounter;
        gridArray[x3][y3] = trominoCounter;
```

Function memorize (integer states[] , integer R , integer C , integer X , integer Y , integer N)
//Input: N: the size of the grid.
X and Y: variables initially equal to zero that helps in filling the squares by order.
R and C: variables show the row and the column location of the first quadrant square of four adjacent missing tiles
States []: array that stores the different hole locations in 2x2 square quadrants
//Output: stores the different hole locations in 2x2 square quadrants and place a tromino in the middle of the quadrants according to the hole's location

```
If missing tile is in 1st quadrant
  save the state in the states array.
  call function PLACE () to place a tromino in the middle of the quadrants according to the
  hole's location.

If missing tile is in 2nd quadrant
  save the state in the states array.
  call function PLACE () to place a tromino in the middle of the quadrants according to the
  hole's location.

If missing tile is in 3rd quadrant
  save the state in the states array.
  call function PLACE () to place a tromino in the middle of the quadrants according to the
  hole's location.

If missing tile is in 4th quadrant
  save the state in the states array.
  call function PLACE () to place a tromino in the middle of the quadrants according to the
  hole's location.
```

E. Code in Java

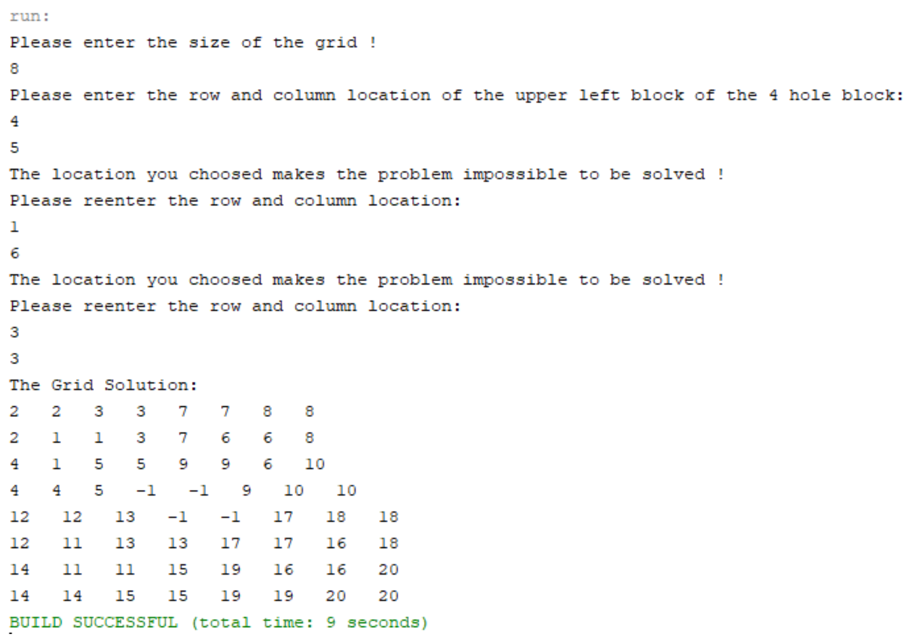
```
package task6algo;

// Java program to place tiles
public class DynamicTrominos
{
static int size_of_grid,trominoCounter = 0;
static int[][] gridArray = new int[128][128];
static int[]statesArray = new int [4];

// Placing tile at the given coordinates
static void place(int x1, int y1, int x2,int y2, int x3, int y3, int r , int
x , int n, int y, int c)
{
    if((r < x + n / 2 && c < y + n / 2)&&(r+1 >= x + n / 2 && c+1 >= y + n /
2)){
        return;

    }
    else{
        trominoCounter++;
      gridArray[x1][y1] = trominoCounter;
      gridArray[x2][y2] = trominoCounter;
      gridArray[x3][y3] = trominoCounter;
    }
}
// Quadrant names
// 1 2
// 3 4

static void memorize(int states[] , int r , int c , int x , int y , int n)
{
    if (r < x + n / 2 && c < y + n / 2)
    {
        states [0] = 1;
        place(x + n / 2, y + (n / 2) - 1, x + n / 2,y + n / 2, x + n / 2 -
1, y + n / 2,r,x,n,y,c);
    }
    else if (r < x + n / 2 && c >= y + n / 2)
    {
        states [1] = 2;
        place(x + n / 2, y + (n / 2) - 1, x + n / 2,y + n / 2, x + n / 2 -
1, y + n / 2 - 1,r,x,n,y,c);
    }
    else if (r >= x + n / 2 && c < y + n / 2)
    {
        states [2] = 3;
         place(x + (n / 2) - 1, y + (n / 2), x + (n / 2),y + n / 2, x + (n /
2) - 1, y + (n / 2) - 1,r,x,n,y,c);

    }
    else if (r >= x + n / 2 && c >= y + n / 2)
    {
        states [3] = 4;
        place(x + (n / 2) - 1, y + (n / 2), x + (n / 2),y + (n / 2) - 1, x +
(n / 2) - 1,y + (n / 2) - 1,r,x,n,y,c);
//
    }

}

// Function based on the dynamic programming
static int tile(int n, int x, int y)
{
        int r = 0, c = 0;
        if (n == 2)
        {
           if((gridArray[x][y] == -1 )&& (gridArray[x][y+1] == -1) &&
(gridArray[x+1][y] == -1) && (gridArray[x+1][y+1] == -1))
           {
              return 0;
           }
           else{
                trominoCounter++;

                for (int i = 0; i < n; i++)
                 {
                        for (int j = 0; j < n; j++)
                        {
                            if (gridArray[x + i][y + j] == 0)
                             {
                                gridArray[x + i][y + j] = trominoCounter;
                             }
                        }
                 }

                return 0;
           }
        }

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

                   }
            }
        }
         // If missing tile is in 1st quadrant
         if(r < x + n / 2 && c < y + n / 2)
         {
             if(statesArray[0] == 1)
             {
                place(x + n / 2, y + (n / 2) - 1, x + n / 2,y + n / 2, x + n
/ 2 - 1, y + n / 2,r,x,n,y,c);
             }
             else
             {
             memorize(statesArray , r , c , x , y ,  n);
             }
         }
         // If missing tile is in 2nd quadrant
         if(r < x + n / 2 && c >= y + n / 2)
         {
             if(statesArray[1] == 2)
             {
                place(x + n / 2, y + (n / 2) - 1, x + n / 2,y + n / 2, x + n
/ 2 - 1, y + n / 2 - 1,r,x,n,y,c);
             }
             else
             {
             memorize(statesArray , r , c , x , y ,  n);
             }
         }

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

         // If missing tile is in 4th quadrant
         if(r >= x + n / 2 && c >= y + n / 2)
         {
             if(statesArray[3] == 4)
             {
                place(x + (n / 2) - 1, y + (n / 2), x + (n / 2),y + (n / 2)
- 1, x + (n / 2) - 1,y + (n / 2) - 1,r,x,n,y,c);

             }
             else
             {
               memorize(statesArray , r , c , x , y ,  n);
             }
         }




      // diving the tile again into 4 quadrants
        tile(n / 2, x, y);          // 1
      tile(n / 2, x, y + n / 2); //2
      tile(n / 2, x + n / 2, y); // 3
      tile(n / 2, x + n / 2, y + n / 2); //4


      return 0;
}

}




package task6algo;

import static task6algo.DynamicTrominos.tile;
import java.util.Scanner;
import static task6algo.DynamicTrominos.gridArray;


public class Task6AlgoDynamic {

public static void main(String[] args)
{
    int row;
    int column;
    System.out.println("Please enter the size of the grid !");
    Scanner sc =new Scanner(System.in);
    int size_of_grid = sc.nextInt();


        System.out.println("Please enter the row and column location of the
upper left block of the 4 hole block:");
        row = sc.nextInt();
        column = sc.nextInt();

        while((row + column) % 2 != 0){
            System.out.println("The location you choosed makes the problem
impossible to be solved !");
            System.out.println("Please reenter the row and column
location:");
            row = sc.nextInt();
            column = sc.nextInt();
        }
      // Here tile can not be placed
      gridArray[row][column] = -1;
        gridArray[row][column + 1] = -1;
        gridArray[row + 1][column] = -1;
        gridArray[row + 1][column + 1] = -1;
      tile(size_of_grid, 0, 0);

        System.out.println("The Grid Solution:");
      // The grid is
      for (int i = 0; i < size_of_grid; i++)
      {
      for (int j = 0; j < size_of_grid; j++)
            System.out.print(gridArray[i][j] + "   ");
      System.out.println();
        }}}
```

Detailed Steps of The Solution Using Dynamic Programming Algorithm:

- The user is asked to enter the size of the grid.
- The user entered 8, to have 8x8 grid size.
- The user is asked to enter the row and column location of the upper left block of the 4-hole block.
- The user entered row = 4 and column = 5.
- The user was warned that the location he/she chooses makes the problem impossible to be solved!
- The user is asked to reenter the row and column location.
- The user entered row = 1 and column = 6.
- The user was warned that the location he/she chooses makes the problem impossible to be solved!
- The user is asked to reenter the row and column location.
- The user entered row = 3 and column = 3.
- The Grid Solution is printed.

F. Sample Output of Dynamic Programming Code

Description for the output: The output is 8x8 grid filed with exactly 20 torminas and contains four adjacent missing tiles in the center of the 8x8 grid that are denoted by -1 value.



```
run:
Please enter the size of the grid !
8
Please enter the row and column location of the upper left block of the 4 hole block:
4
5
The location you choosed makes the problem impossible to be solved !
Please reenter the row and column location:
1
6
The location you choosed makes the problem impossible to be solved !
Please reenter the row and column location:
3
3
The Grid Solution:
2   2   3   3   7   7   8   8
2   1   1   3   7   6   6   8
4   1   5   5   9   9   6   10
4   4   5   -1   -1   9   10   10
12   12   13   -1   -1   17   18   18
12   11   13   13   17   17   16   18
14   11   11   15   19   16   16   20
14   14   15   15   19   19   20   20
BUILD SUCCESSFUL (total time: 9 seconds)
```

G. Complexity analysis for the Dynamic Programming Algorithm

- Recurrence relation for above recursive algorithm is **$T(n) = 4T(n/2) + C$** and C is a constant.
- The time complexity is **$O(n^2)$**

H. Comparison with Another Algorithm

Divide-and-Conquer

Divide-and-conquer is a technique used for designing algorithms that consist of dividing the problem into smaller subproblems hoping that the solutions of the subproblems are easier to find and then composing the partial solutions into the solution of the original problem.

Divide-and-Conquer Pseudocode:

```
Global variables: gridArray[128][128], statesArray[4] and trominoCounter initially 0

Function TILE (integer N, integer X, integer Y)

//Input: N: the size of the grid.
           X and Y: variables initially equal to zero that helps in filling the squares by order.

//Output: Filling the grid with trominos.

If the size of grid = = 2 then
   If the 2x2 square is all holes then return.
   Else trominoCounter ++
        fill the 2x2 square with a suitable tromino using trominoCounter value.

For I ← X to I < X + N
  For J ← Y to J < Y + N
    Find the location of the first quadrant square of four adjacent missing tiles.



If missing tile is in 1st quadrant
   call function PLACE () to place a tromino in the middle of the quadrants according to the
   hole’s location.


If missing tile is in 2nd quadrant
   call function PLACE () to place a tromino in the middle of the quadrants according to the
   hole’s location.

If missing tile is in 3rd quadrant
   call function PLACE () to place a tromino in the middle of the quadrants according to the
   hole’s location.
```

```
If missing tile is in 4th quadrant
    call function PLACE () to place a tromino in the middle of the quadrants according to the
    hole's location.


 // diving the tile again into 4 quadrants
            Tile (n / 2, x, y); // 1
             Tile (n / 2, x, y + n / 2); //2
             Tile (n / 2, x + n / 2, y); // 3
             Tile (n / 2, x + n / 2, y + n / 2); //4
```

Function place (integer X1, integer Y1, integer X2, integer Y2, integer X3, integer Y3, integer R, integer X, integer N, integer Y, integer C)

//Input: N: the size of the grid.
X and Y: variables initially equal to zero that helps in filling the squares by order.
R and C: variables show the row and the column location of the first quadrant square of four adjacent missing tiles.
X1, Y1, X2, Y2, X3 and Y3: variables show the row and the column location of the three squares the construct the tromino.

//Output: place a tromino in the middle of the quadrants according to the hole's location.

```
If the 2x2 middle square is filled completely with holes then return
Else    trominoCounter++;
             gridArray[x1][y1] = trominoCounter;
             gridArray[x2][y2] = trominoCounter;
             gridArray[x3][y3] = trominoCounter;
```

Divide-and-Conquer Code in JAVA:

```
package task6algov2;

// Java program to place tiles
public class DivideAndConcorTrominos
{
static int size_of_grid,trominoCounter = 0;
static int[][] gridArray = new int[128][128];
static int[][] statesArray = new int [10][2];

// Placing tile at the given coordinates
static void place(int x1, int y1, int x2,int y2, int x3, int y3, int r , int
x , int n, int y, int c)
{
    if((r < x + n / 2 && c < y + n / 2)&&(r+1 >= x + n / 2 && c+1 >= y + n /
2)){
        return;
    }
    else{
        trominoCounter++;
      gridArray[x1][y1] = trominoCounter;
      gridArray[x2][y2] = trominoCounter;
      gridArray[x3][y3] = trominoCounter;
    }
}
// Quadrant names
// 1 2
// 3 4

// Function based on divide and conquer
static int tile(int n, int x, int y)
{
      int r = 0, c = 0;
      if (n == 2)
      {
           if((gridArray[x][y] == -1 )&& (gridArray[x][y+1] == -1) &&
(gridArray[x+1][y] == -1) && (gridArray[x+1][y+1] == -1))
           {
              return 0;
           }
           else{
                trominoCounter++;

                for (int i = 0; i < n; i++)
                 {
                        for (int j = 0; j < n; j++)
                        {
                        if (gridArray[x + i][y + j] == 0)
                        {
                                gridArray[x + i][y + j] = trominoCounter;
                        }
                        }
                 }

                return 0;
           }
        }

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

            }
```

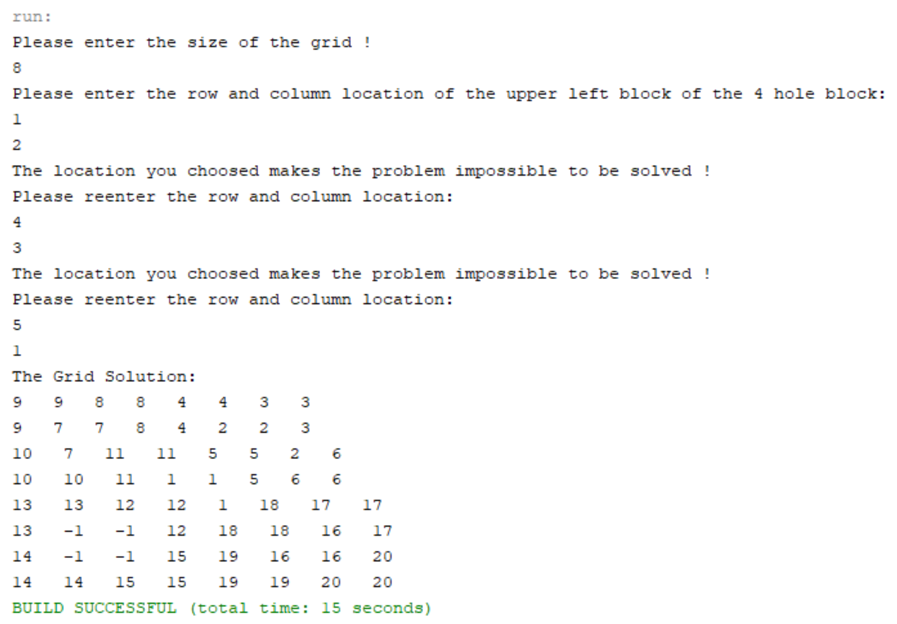
```
       }
       }


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

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


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



       // If missing Tile is in 4th quadrant
       else if (r >= x + n / 2 && c >= y + n / 2)
       place(x + (n / 2) - 1, y + (n / 2), x + (n / 2),
                   y + (n / 2) - 1, x + (n / 2) - 1,
                   y + (n / 2) - 1,r,x,n,y,c);


       // diving it again in 4 quadrants
            tile(n / 2, x, y);          // 1
       tile(n / 2, x, y + n / 2); //2
       tile(n / 2, x + n / 2, y); // 3
       tile(n / 2, x + n / 2, y + n / 2); //4/


       return 0;
}
}



package task6algov2;

import static task6algov2.DivideAndConcorTrominos.tile;
import java.util.Scanner;
import static task6algov2.DivideAndConcorTrominos.gridArray;

public class Task6AlgoDivideAndConcor {

public static void main(String[] args)
{
    int row;
    int column;
    System.out.println("Please enter the size of the grid !");
    Scanner sc =new Scanner(System.in);
    int size_of_grid = sc.nextInt();


        System.out.println("Please enter the row and column location of the
upper left block of the 4 hole block:");
        row = sc.nextInt();
        column = sc.nextInt();

        while((row + column) % 2 != 0){
            System.out.println("The location you choosed makes the problem
impossible to be solved !");
            System.out.println("Please reenter the row and column
location:");
            row = sc.nextInt();
            column = sc.nextInt();
        }
      // Here tile can not be placed
      gridArray[row][column] = -1;
        gridArray[row][column + 1] = -1;
        gridArray[row + 1][column] = -1;
        gridArray[row + 1][column + 1] = -1;
      tile(size_of_grid, 0, 0);

        System.out.println("The Grid Solution:");

      // The grid is
      for (int i = 0; i < size_of_grid; i++)
      {
      for (int j = 0; j < size_of_grid; j++)
            System.out.print(gridArray[i][j] + "   ");
      System.out.println();
      }
}
}
```

Detailed Steps of The Solution Using Divide-and-Conquer Algorithm:

1. The user is asked to enter the size of the grid.
2. The user entered 8, in order to have 8x8 grid size.
3. The user is asked to enter the row and column location of the upper left block of the 4-hole block
4. The user entered row = 1 and column = 2.
5. The user was warned that the location he/she chooses makes the problem impossible to be solved!
6. The user is asked to reenter the row and column location.
7. The user entered row = 4 and column = 3.
8. The user was warned that the location he/she chooses makes the problem impossible to be solved!
9. The user is asked to reenter the row and column location.
10. The user entered row = 5 and column = 1.
11. The Grid Solution is printed.

Divide-and-Conquer Code Output:

Description for the output: The output is 8x8 grid filed with exactly 20 trominos and contains four adjacent missing tiles in the center of the third quadrant of the grid that are denoted by -1 value.



```
run:
Please enter the size of the grid !
8
Please enter the row and column location of the upper left block of the 4 hole block:
1
2
The location you choosed makes the problem impossible to be solved !
Please reenter the row and column location:
4
3
The location you choosed makes the problem impossible to be solved !
Please reenter the row and column location:
5
1
The Grid Solution:
9    9    8    8    4    4    3    3
9    7    7    8    4    2    2    3
10   7    11   11   5    5    2    6
10   10   11   1    1    5    6    6
13   13   12   12   1    18   17   17
13   -1   -1   12   18   18   16   17
14   -1   -1   15   19   16   16   20
14   14   15   15   19   19   20   20
BUILD SUCCESSFUL (total time: 15 seconds)
```

Complexity analysis for the Divide-and-Conquer Algorithm:

- Recurrence relation for above recursive algorithm is **T(n) = 4T(n/2) + C** and C is a constant.
- The time complexity is **O(n^2)**

I. Conclusion

- The dynamic programming approach is an extension of the divide-and-conquer problem. It extends Divide-and-Conquer problems with two techniques (memorization and tabulation) that stores the solutions of sub-problems and re-use whenever necessary.

- The Dynamic Programming Algorithm didn't drastically improve performance compared to the Divide-and-Conquer Algorithm in solving this problem as both of the algorithms has time complexity of **O(n^2).**

VII. REFERENCE

➢ Levitin, Anany, and Maria Levitin. Algorithmic puzzles. OUP USA, 2011.