CSE332:

Design and Analysis of Algorithms

Project

Table of Contents

[1. PROJECT REQUIREMENTS 2](#_Toc102845234)

[1.1 Task 1 2](#_Toc102845235)

[1.1.1 Problem description 2](#_Toc102845236)

[1.1.2 Solution 2](#_Toc102845237)

[1.1.2.1 Pseudocode 2](#_Toc102845238)

[1.1.2.2 Code 7](#_Toc102845239)

[1.1.2.3 Solution description 9](#_Toc102845240)

[1.1.2.4 Complexity analysis 11](#_Toc102845242)

[1.1.2.5 Comparison between another algorithm 11](#_Toc102845243)

[1.1.2.6 Sample of the output 16](#_Toc102845244)

[1.2 Task 2 26](#_Toc102845245)

[1.2.1 Problem description 26](#_Toc102845246)

[1.2.2 Solution 26](#_Toc102845247)

[1.2.2.1 Pseudocode 26](#_Toc102845248)

[1.2.2.2 Code 29](#_Toc102845249)

[1.2.2.3 Solution description 32](#_Toc102845250)

[1.2.2.4 Complexity analysis 33](#_Toc102845251)

[1.2.2.5 Comparison between another algorithm 33](#_Toc102845252)

[1.2.2.6 Sample of the output 35](#_Toc102845253)

[1.3 Task 3 40](#_Toc102845254)

[1.3.1 Problem description 40](#_Toc102845255)

[1.3.2 Solution 40](#_Toc102845256)

[1.3.2.1 Pseudocode 40](#_Toc102845257)

[1.3.2.2 Code 44](#_Toc102845258)

[1.3.2.3 Solution description 46](#_Toc102845259)

[1.3.2.4 Complexity analysis 48](#_Toc102845260)

[1.3.2.5 Comparison between another algorithm 49](#_Toc102845261)

[1.3.2.6 Sample of the output 53](#_Toc102845262)

[1.4 Task 4 55](#_Toc102845263)

[1.4.1 Problem description 55](#_Toc102845264)

[1.4.2 Solution 55](#_Toc102845265)

[1.4.2.1 Pseudocode 55](#_Toc102845266)

[1.4.2.2 Code 56](#_Toc102845267)

[1.4.2.3 Solution description 57](#_Toc102845268)

[1.4.2.4 Complexity analysis 59](#_Toc102845269)

[1.4.2.5 Comparison between another algorithm 59](#_Toc102845270)

[1.4.2.6 Sample of the output 63](#_Toc102845271)

[1.5 Task 5 69](#_Toc102845276)

[1.5.1 Problem description 69](#_Toc102845277)

[1.5.2 Solution 69](#_Toc102845278)

[1.5.2.1 Pseudocode 69](#_Toc102845279)

[1.5.2.2 Code 70](#_Toc102845280)

[1.5.2.3 Solution description 71](#_Toc102845281)

[1.5.2.4 Complexity analysis 72](#_Toc102845282)

[1.5.2.5 Comparison between another algorithm 74](#_Toc102845283)

[1.5.2.6 Sample of the output 75](#_Toc102845284)

[1.6 Task 6 78](#_Toc102845285)

[1.6.1 Problem description 78](#_Toc102845286)

[1.6.2 Solution 78](#_Toc102845287)

[1.6.2.1 Pseudocode 78](#_Toc102845288)

[1.6.2.2 Code 81](#_Toc102845289)

[1.6.2.3 Solution description 83](#_Toc102845290)

[1.6.2.4 Complexity analysis 83](#_Toc102845291)

[1.6.2.5 Comparison between another algorithm 85](#_Toc102845292)

[1.6.2.6 Sample of the output 85](#_Toc102845293)

[2. REFERENCES iv](#_Toc102845234)

#### List of Figures:

Figure 1: output for n = 5 14

[Figure 2: output for n =6 12](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845756)

[Figure 3: Output for number of rows = 4 20](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845758)

[Figure 4: Output for number of rows = 5 21](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845759)

[Figure 5: Output for number of rows = 8 23](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845760)

[Figure 6: Output for number of rows = 9 25](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845761)

[Figure 7: Output for number of cells = 1 36](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845762)

[Figure 8: Output for number of cells = 7 36](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845763)

[Figure 9: Output for number of cells = 8 39](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845764)

[Figure 10: Output for the Code 54](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845765)

[Figure 11: Output for number (3) 63](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845766)

[Figure 12: Output for number (3) 63](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845767)

[Figure 13: Output for number (16) using optimal code 64](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845768)

[Figure 14: Output for number (16) using greedy algorithm 64](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845769)

[Figure 15: Output for number (18) 65](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845770)

[Figure 16: Output for number (512) 66](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845771)

[Figure 17: Output for number (1024) 66](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845772)

[Figure 18: Output for number (64) 67](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845773)

[Figure 19: Output for number (48) 67](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845774)

[Figure 20: Output for number (2) 68](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845775)

[Figure 21: Output for number (4) 75](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845776)

[Figure 22: Output for number (3) 76](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845777)

[Figure 23: Output for number (6) 77](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845778)

[Figure 24: main to run the Output 85](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845779)

[Figure 25: Output for number (8) using dynamic programming 86](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845780)

[Figure 26: Output for number (8) using divide and conquer 86](file:///C:\Users\lenovo\Desktop\Junior\Semester%206\Design%20and%20Analysis%20of%20Algorithms\CSE332%20Project%20Final.docx#_Toc102845781)

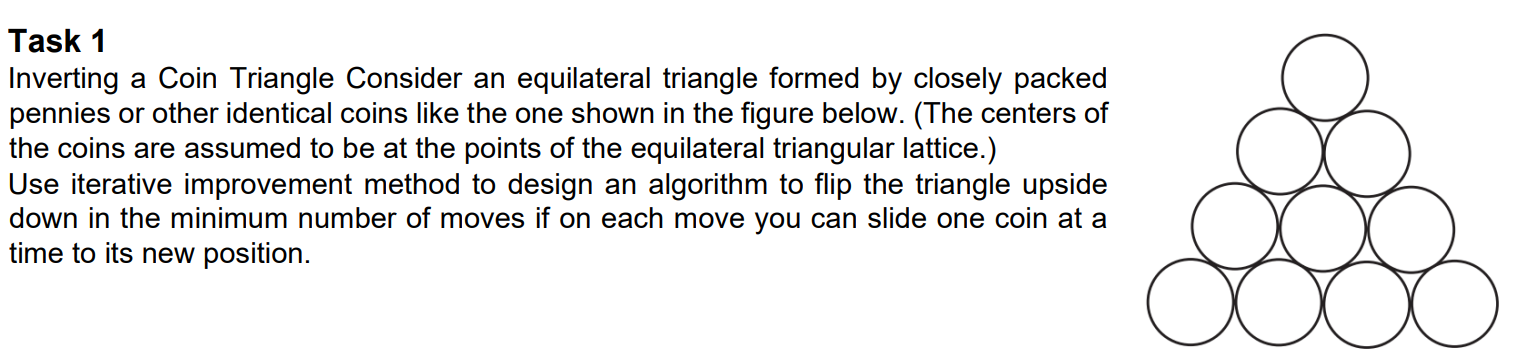
**Introduction**

In this report we have solved 6 puzzles using the required algorithms and we write the implementation for these algorithms then we calculate the complexity for each algorithm and finally we compared the required algorithm with the optimal algorithm and also we wrote its implementation and we calculated its complexity.

# PROJECT REQUIREMENTS

## Task 1

### 1.1.1 Problem description

****

### 1.1.2 Solution

#### 1.1.2.1 Pseudocode

Algorithm Invert a Triangle of Coins (int no\_of\_rows)

//input no\_of\_rows “Integer specified by the user”

//output Inverted Triangle of coins

Class rows (no\_of\_coins,totalrows)

{

addcoins(addcoins)

{

no\_of\_coins ← 0

no\_of\_coins ← no\_of\_coins + addcoins

updatespaces()

}

removecoins(removeCoins)

{

no\_of\_coins ← 0

no\_of\_coins ← no\_of\_coins - removeCoins

updatespaces()

}

updatespaces()

{

Spaces ← 0

Spaces ← totalrows - no\_of\_coins

}

}

Class pyramid (total\_rows)

{

Create a list called “rowlist”

Create a list called “rowlist2”

Create a list called “manipulatedrows”

for i ← 1 to (total\_rows+1) do

rowlist.append(rows(i,total\_rows))

no\_of\_iterations()

{

coins\_number ← 0

for i ← 1 to (total\_rows+1) do

coins\_number ← coins\_number + i

iterations ← (floor(coins\_number/3))

print("number of iterartions are: ")

print(iterations)

}

showPyramid(self)

{

Create a list called “temparr”

for row ← 0 to rowlist do

for i ← 0 to row.spaces do

temparr.append("")

for i ← 0 to row.no\_of\_coins do

temparr.append("1")

for i ← 0 to row.spaces do

temparr.append("")

for i ← 0 to len(temparr) do

print(i, end =" ")

print("\n")

temparr.clear()

}

getmanipulated()

{

manipulatedrows.append(rowlist[0])

int max ← 0

if (len(rowlist)%2 = 0)

{

max ← ((len(rowlist)/2)+1)

for i ← max to len(rowlist) do

manipulatedrows.append(rowlist[i])

}

else

{

manipulatedrows.append(rowlist[0])

max ← floor(int((len(rowlist)/2)+1))

for i ← max to len(rowlist)) do

manipulatedrows.append(rowlist[i])

}

updaterowlist()

{

index ← self.total\_rows - 2

s ← 1

for k ← 0 to (len(rowlist2)) do

for i ← 0 to (len(rowlist)-s) do

rowlist[i].removecoins(1)

firstrow ← self.rowlist2[index]

rowlist.append(firstrow)

showPyramid()

index ← index - 1

s ← s + 1

}

}

#### 1.1.2.2 Code

from math import ceil, floor

from mimetypes import init

from tempfile import tempdir

class pyramid:

    rowlist=[]

    rowlist2 =[]

    manipulatedrows=[]

    def \_\_init\_\_(self,total\_rows):

        self.total\_rows = total\_rows

        for i in range(1,total\_rows+1):

            self.rowlist.append(rows(i,total\_rows))

        for i in range(1,total\_rows):

            self.rowlist2.append(rows(i,total\_rows))

    def no\_of\_iterations(self):

        coins\_number = 0

        for i in range(1,(self.total\_rows+1)):

            coins\_number = coins\_number+i

        iterations = int (floor(coins\_number/3))

        return iterations

        """print("number of iterartions are: ")

        print(iterations)"""

    def showPyramid(self):

        temparr=[]

        for row in self.rowlist:

            for i in range(row.spaces):

                temparr.append("")

            for i in range(row.no\_of\_coins) :

                temparr.append("1")

            for i in range(row.spaces):

                temparr.append("")

            for i in temparr:

                 print(i, end =" ")

            print("\n")

            temparr.clear()

    def getmanipulated(self):

        self.manipulatedrows.append(self.rowlist[0])

        if (len(self.rowlist)%2 == 0):

            max=int((len(self.rowlist)/2)+1)

            for i in range(max,len(self.rowlist)):

                self.manipulatedrows.append(self.rowlist[i])

        else:

            self.manipulatedrows.append(self.rowlist[0])

            max= floor(int((len(self.rowlist)/2)+1))

            for i in range(max,len(self.rowlist)):

                self.manipulatedrows.append(self.rowlist[i])

    def updaterowlist(self):

        index = self.total\_rows - 2

        s = 1

        for k in range (len(self.rowlist2)):

            for i in range (len(self.rowlist)-s):

                self.rowlist[i].removecoins(1)

            #self.showPyramid()

            firstrow=self.rowlist2[index]

            self.rowlist.append(firstrow)

            self.showPyramid()

            index = index - 1

            s = s + 1

class rows:

     # init method or constructor

    def \_\_init\_\_(self,no\_of\_coins,totalrows):

        self.no\_of\_coins = no\_of\_coins

        self.spaces = totalrows - no\_of\_coins

        self.totalrows=totalrows

    def addcoins(self,addcoins):

        self.no\_of\_coins=self.no\_of\_coins+addcoins

        self.updatespaces()

    def removecoins(self,removeCoins):

        self.no\_of\_coins=self.no\_of\_coins-removeCoins

        self.updatespaces()

    def updatespaces(self):

        self.spaces=self.totalrows-self.no\_of\_coins

if \_\_name\_\_=="\_\_main\_\_":

    no\_of\_rows = int (input("Enter number of rows: "))

    mypyramid = pyramid(no\_of\_rows)

    mypyramid.showPyramid()

    mypyramid.getmanipulated()

    mypyramid.updaterowlist()

    print ('\n')

    print ("Final Result")

    mypyramid.showPyramid()

#### 1.1.2.3 Solution description

1. The user enter number of rows
2. We used this number to draw a triangle of coins
3. First we draw triangle of coins Note: every coin is represented by “1” in the code
4. Our goal is to invert this triangle of coins:

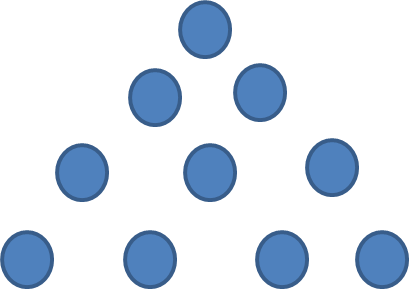
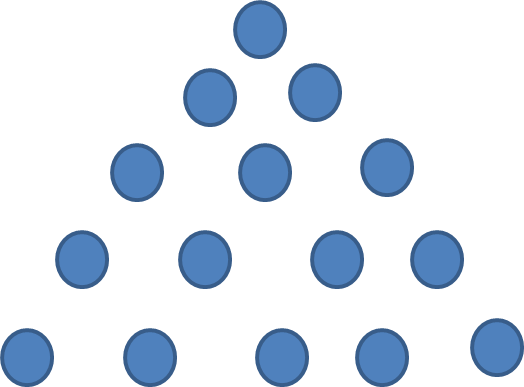
* In the beginning we need to know that we have symmetric shape in the middle and we will slide only the edge coins
* Firstly we will slide number of coins equal to “Specific Number”

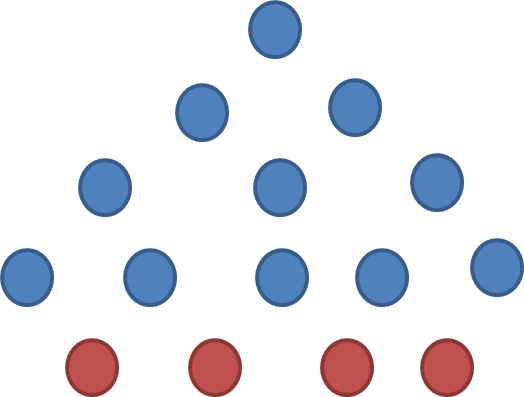
Specific Number = ((the number of coins in the last row)-1)

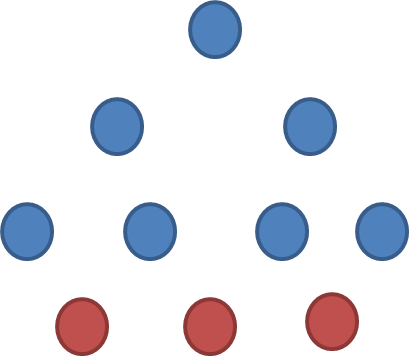
Note: We slide only one coin at a time to its new position

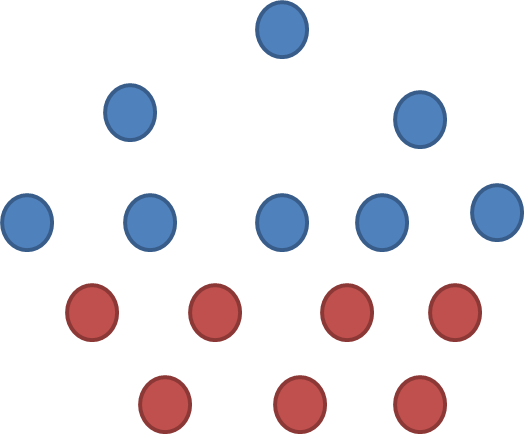
* Second we will slide number of coins equal to “Specific Number - 1”, our notes is still applicable
* Third we will repeat step number 2 until we reach that the “Specific Number = 0”
* Fourth we will find that we have got the inverted triangle
* Finally I will found that these steps helps me to achieve the minimum number of moves

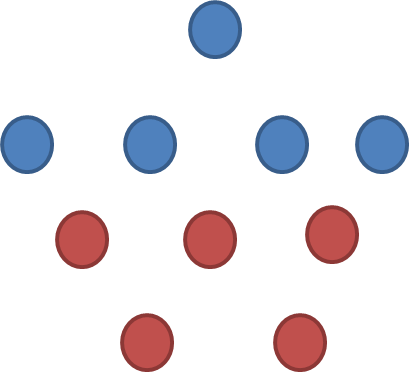
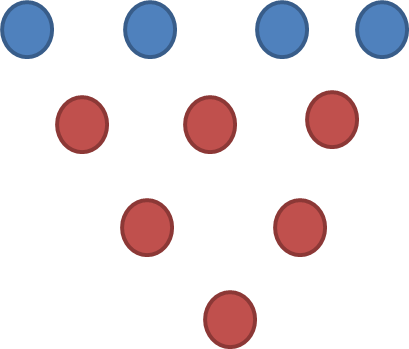
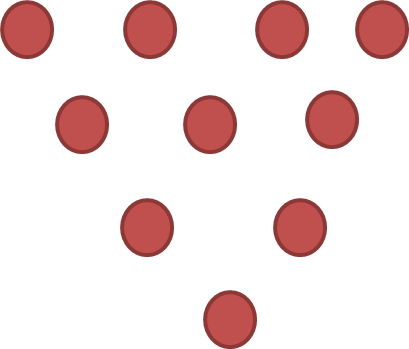
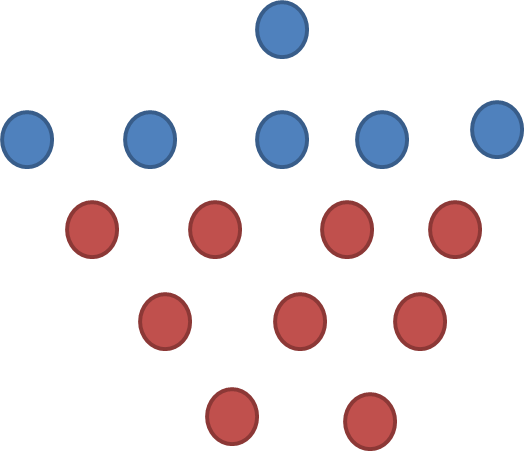
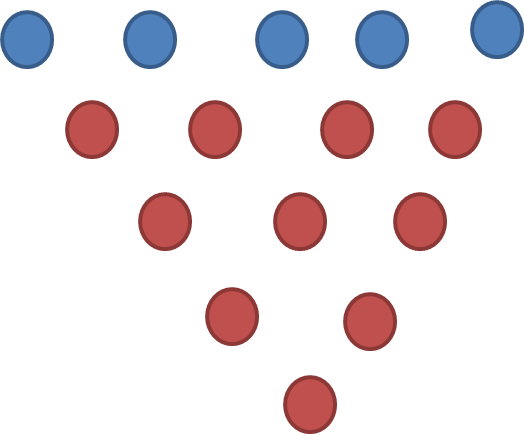
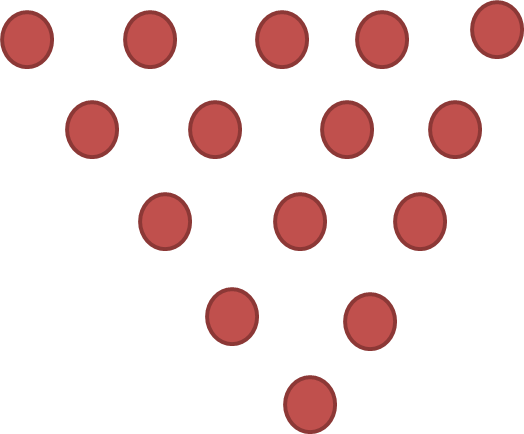
1. To Solve this puzzle I will use iterative improvement algorithm: Iterative Improvement Meaning: It means that you repeat your logic for number of iterations; in each iteration your problem is getting better and get nearer to the final answer.
2. These figures can help in understanding my solution:











**For number of rows = 5**

**For number of rows = 4**

#### 1.1.2.4 Complexity analysis

The total number of moves made by the algorithm, M(k), is obviously the minimum needed to make the kth row the base of the inverted triangle, because each coin move increases the number of coins in a row that must be lengthened and simultaneously decreases the number of coins in a row that must be shortened.

M(k) can be computed as follows:

M(k) = = - +

= (n-k) (⎣⎦ + 1) - ⎣⎦((⎣⎦ + 1) +

= (⎣⎦ + 1) ⎡⎤ +

If n-k is odd, the formula can be simplified to:

M(k) = (+1) + = = O()

If n-k is even, the formula can be simplified to:

M(k) = (+1) + = = O()

Where

k = (n + 2)/3.

Tn = n(n + 1)/2 is the total number of coins in the triangle.

#### 1.1.2.5 Comparison between another algorithm

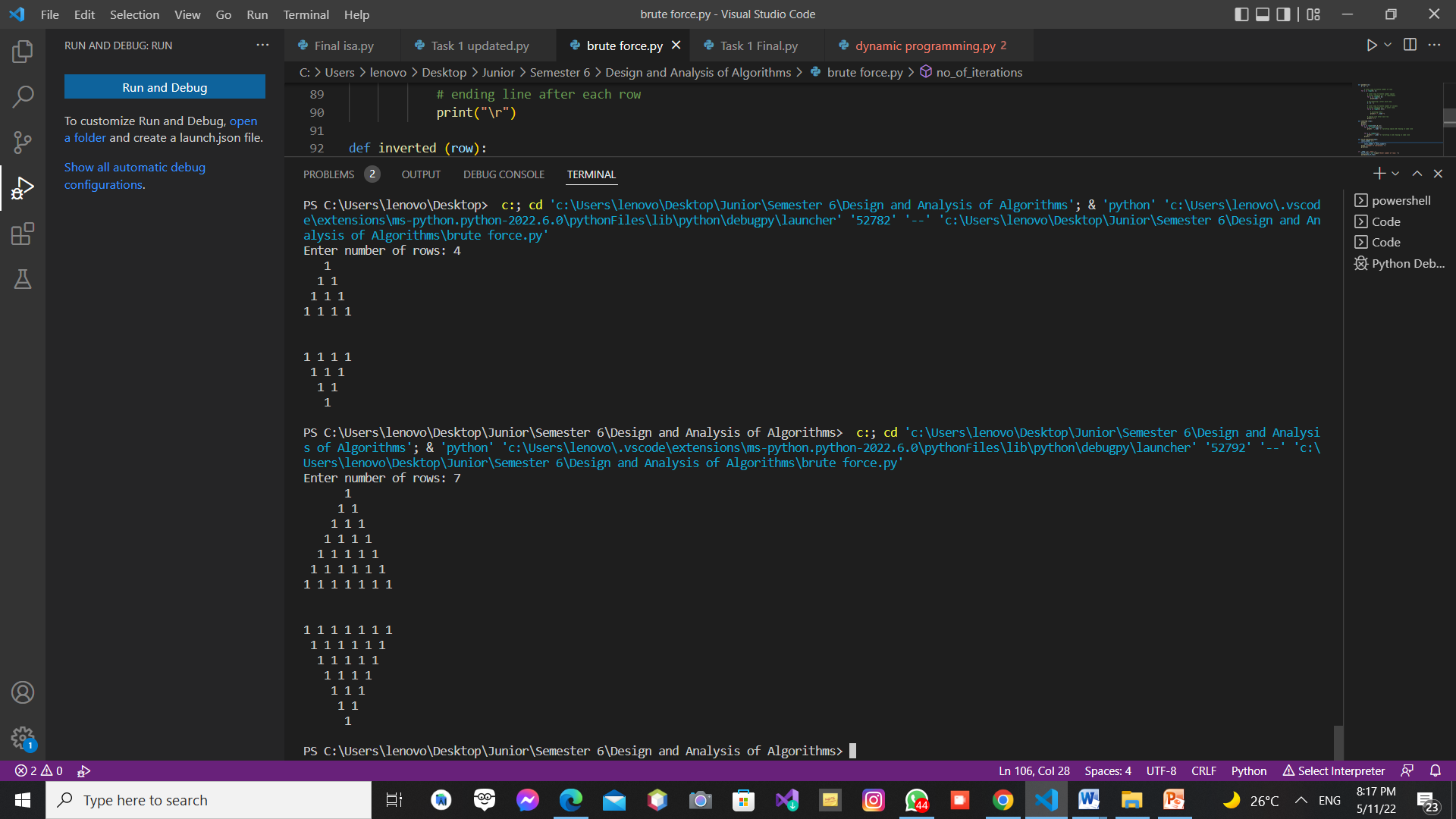
Iterative improvement achieved the minimum number of iterations which is achieved using this formula = n(n + 1)/2 coins is ⎣/3⎦.

1. We can also solve this puzzle using another algorithm “brute force” but is won’t be optimized because the number if iterations will equal the number of coins

* Brute Force Algorithm meaning: straightforward method of solving a problem by trying every possibility rather than advanced techniques to improve efficiency.

1. We can solve this puzzle using “Dynamic Programming” algorithm

* Dynamic Programming Algorithm meaning: is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property.
* Solution Description for applying dynamic programming on this puzzle:
* First I will get the number of rows from the user
* Second I will draw triangle of coins. Note: every coin is represented by “1” in the code
* Third I will divide my triangle to subproblems “Smaller triangles”
* Fourth I will memorize the result of the subproblems to use them in solving the other subproblems. Note: In each time we call the solving function we first check the memorized results to use them directly and after that we complete our answer.
* Finally we will get our inverted triangle
* Note: The Code take into consideration sliding the coins and sliding only one coin at a time to its new position
* Sample of the output: ”Theses screenshots shows the first step and the final result it doesn’t show the intermediate steps ”



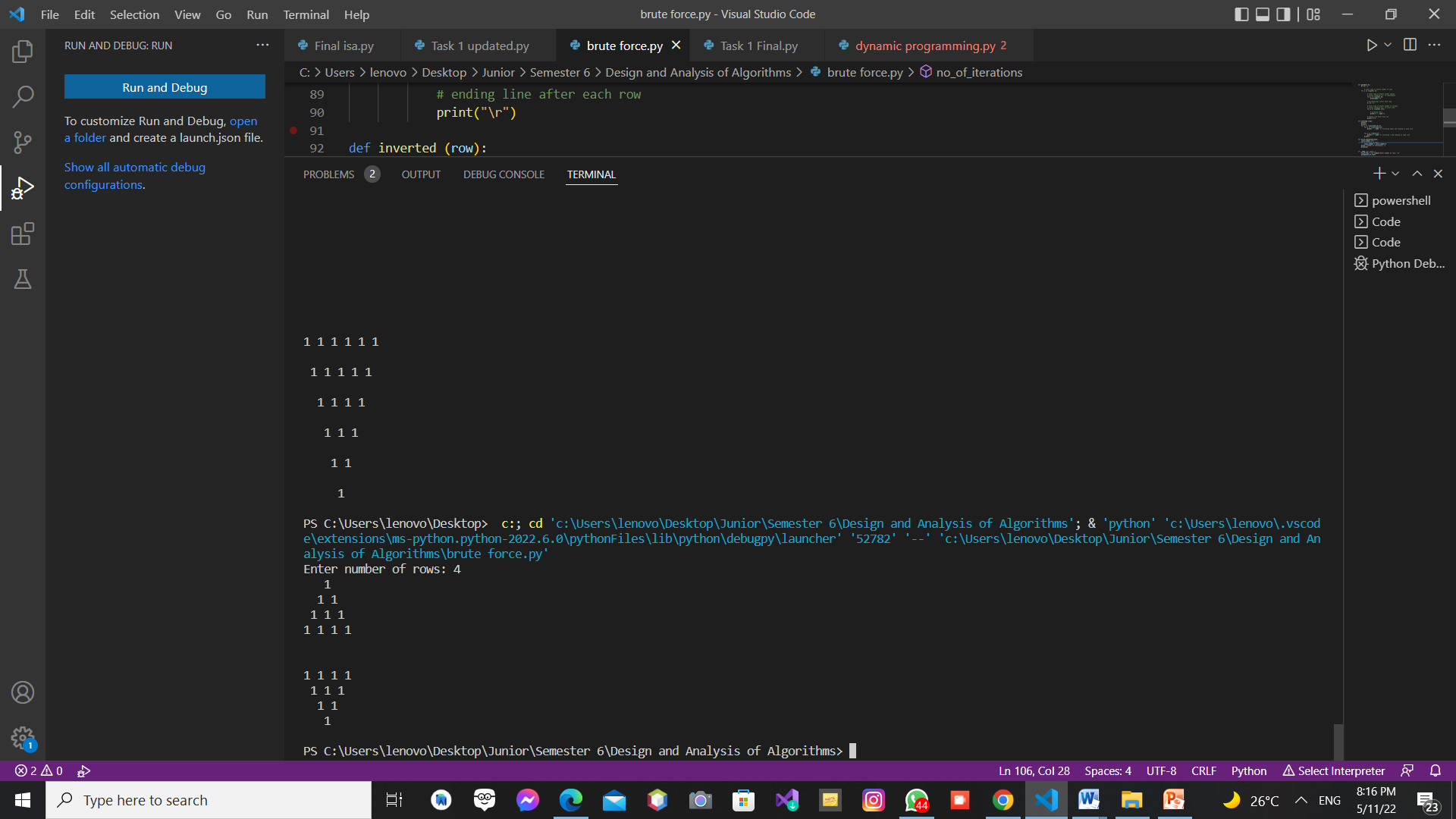


Figure 1: output for n = 4

Figure 2: output for n =7

**Pseudocode for brute force algorithm**

Algorithm Invert a Triangle of Coins (int no\_of\_rows)

//input no\_of\_rows “Integer specified by the user”

//output Inverted Triangle of coins

Class rows (no\_of\_coins,totalrows)

{

addcoins(addcoins)

{

no\_of\_coins ← 0

no\_of\_coins ← no\_of\_coins + addcoins

updatespaces()

}

updatespaces()

{

Spaces ← 0

Spaces = totalrows - no\_of\_coins

}

}

Class pyramid (total\_rows)

{

Create a list called “rowlist”

Create a list called “manipulatedrows”

for i ← 1 to (total\_rows+1) do

rowlist.append(rows(i,total\_rows))

no\_of\_iterations()

{

coins\_number ← 0

for i ← 1 to (total\_rows+1) do

coins\_number ← coins\_number + i

print("number of iterartions are: ")

print(coins\_number)

}

showPyramid(self)

{

Create a list called “temparr”

for row ← 0 to rowlist do

for i ← 0 to row.spaces do

temparr.append("")

for i ← 0 to row.no\_of\_coins do

temparr.append("1")

for i ← 0 to row.spaces do

temparr.append("")

for i ← 0 to len(temparr) do

print(i, end =" ")

print("\n")

temparr.clear()

}

getmanipulated()

{

for i ← 0 to (len(rowlist)) do

manipulatedrows.append(rowlist[i])

}

updaterowlist()

{

j ← 0

if (j <= (len(manipulatedrows)))

{

firstrow ← manipulatedrows[j]

rowlist.remove(firstrow)

rowlist.append(firstrow)

manipulatedrows.remove(firstrow)

showPyramid()

j ← j +1

}

}

**Pseudocode for dynamic programming algorithm**

Algorithm Invert a Triangle of Coins (int no\_of\_rows)

//input no\_of\_rows “Integer specified by the user”

//output Inverted Triangle of coins

Class rows (no\_of\_coins,totalrows)

{

addcoins(addcoins)

{

no\_of\_coins ← 0

no\_of\_coins ← no\_of\_coins + addcoins

updatespaces()

}

removecoins(removeCoins)

{

no\_of\_coins ← 0

no\_of\_coins ← no\_of\_coins - removeCoins

updatespaces()

}

updatespaces()

{

Spaces ← 0

Spaces ← totalrows - no\_of\_coins

}

}

Class pyramid (total\_rows)

{

Create a list called “rowlist”

Create a list called “rowlist2”

Create a list called “savingList”

Create a list called “manipulatedrows”

for i ←1 to (total\_rows+1) do

rowlist.append(rows(i,total\_rows))

showPyramid(self)

{

Create a list called “temparr”

for row ← 0 to rowlist do

for i ← 0 to row.spaces do

temparr.append("")

for i ← 0 to row.no\_of\_coins do

temparr.append("1")

for i ← 0 to row.spaces do

temparr.append("")

for i ← 0 to len(temparr) do

print(i, end =" ")

print("\n")

temparr.clear()

}

getmanipulated()

{

manipulatedrows.append(rowlist[0])

int max ← 0

if (len(rowlist)%2 = 0)

{

max ← int((len(rowlist)/2)+1)

for i ← max to len(rowlist) do

manipulatedrows.append(rowlist[i])

}

else

{

manipulatedrows.append(rowlist[0])

max ← floor(int((len(rowlist)/2)+1))

for i ← max to len(rowlist)) do

manipulatedrows.append(rowlist[i])

}

manipulatingFunction()

{

for i ← 1 to (total\_rows+1) do

savingList = mypyramid.updaterowlist(i)

fsMemoizer(i, savingList)

}

fsMemoizer (i, savingList[])

{

dict = {i: savingList}

}

updaterowlist(input\_i)

{

Create a list called “list”

for s ← 0 to (len(fsMemoizer.dict)) do

i ← fsMemoizer.dict(i)

list ← fsMemoizer.dict.(savingList)

rowlist.append(list)

for i ← 0 to (input\_i - i) do

index ← total\_rows - 2

s ← 1

for k ← 0 to (len(rowlist2)) do

for i ← 0 to (len(self.rowlist)-s) do

rowlist[i].removecoins(1)

firstrow ← rowlist2[index]

rowlist.append(firstrow)

showPyramid()

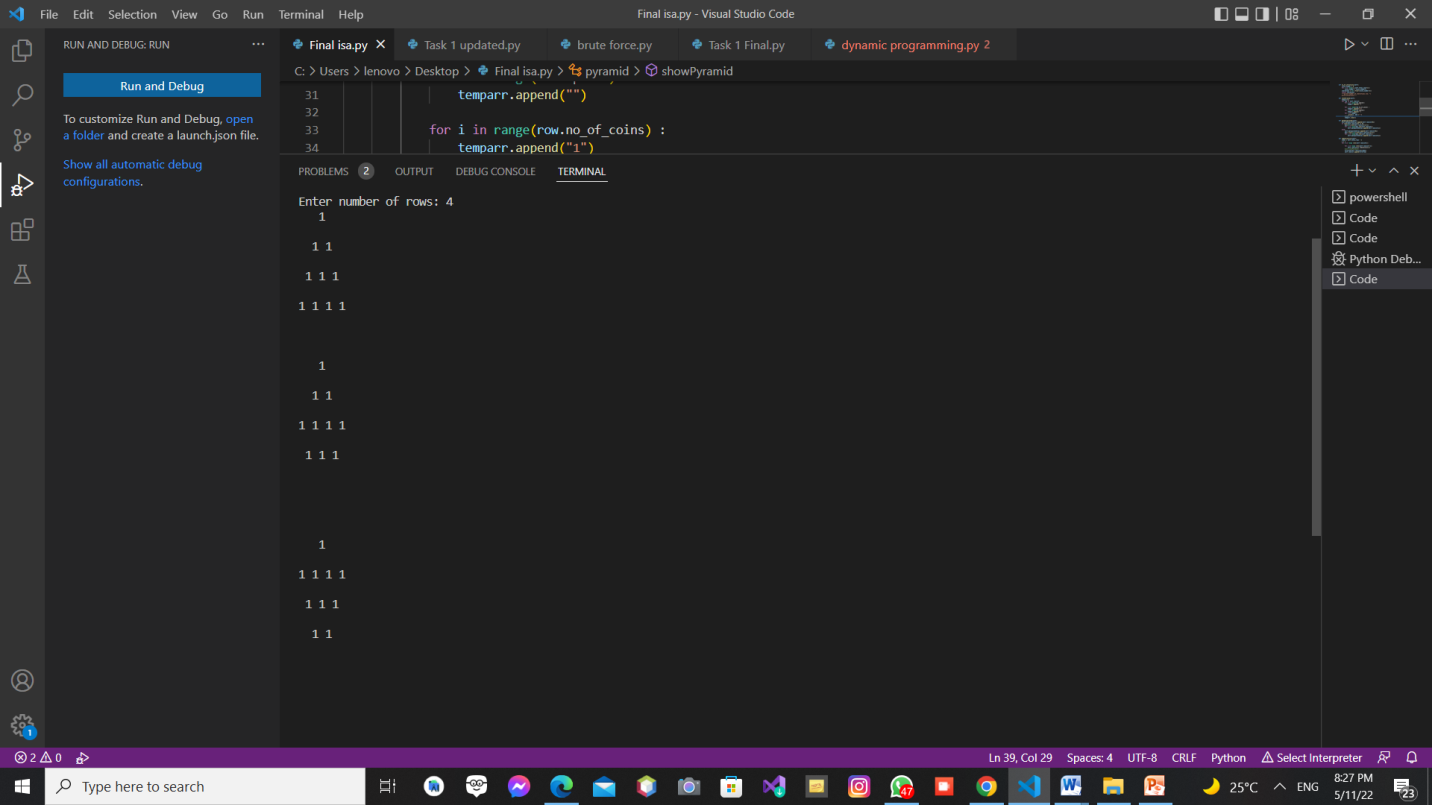
index ← index - 1

s ← s + 1

}}

#### 1.1.2.6 Sample of the output

Screenshots for the Iterative improvement algorithm implementation: “This Code output shows the triangle every time it has clear shape”



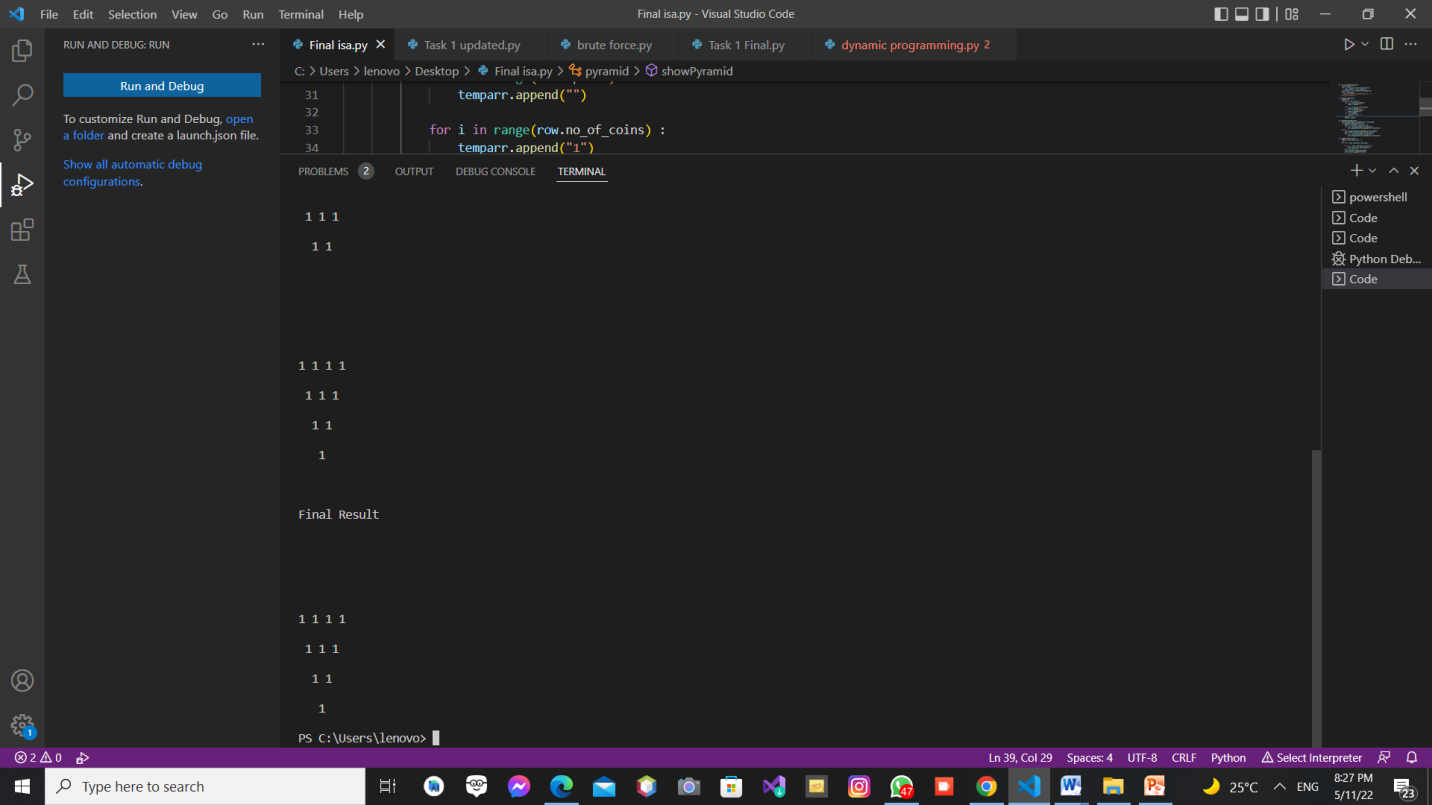
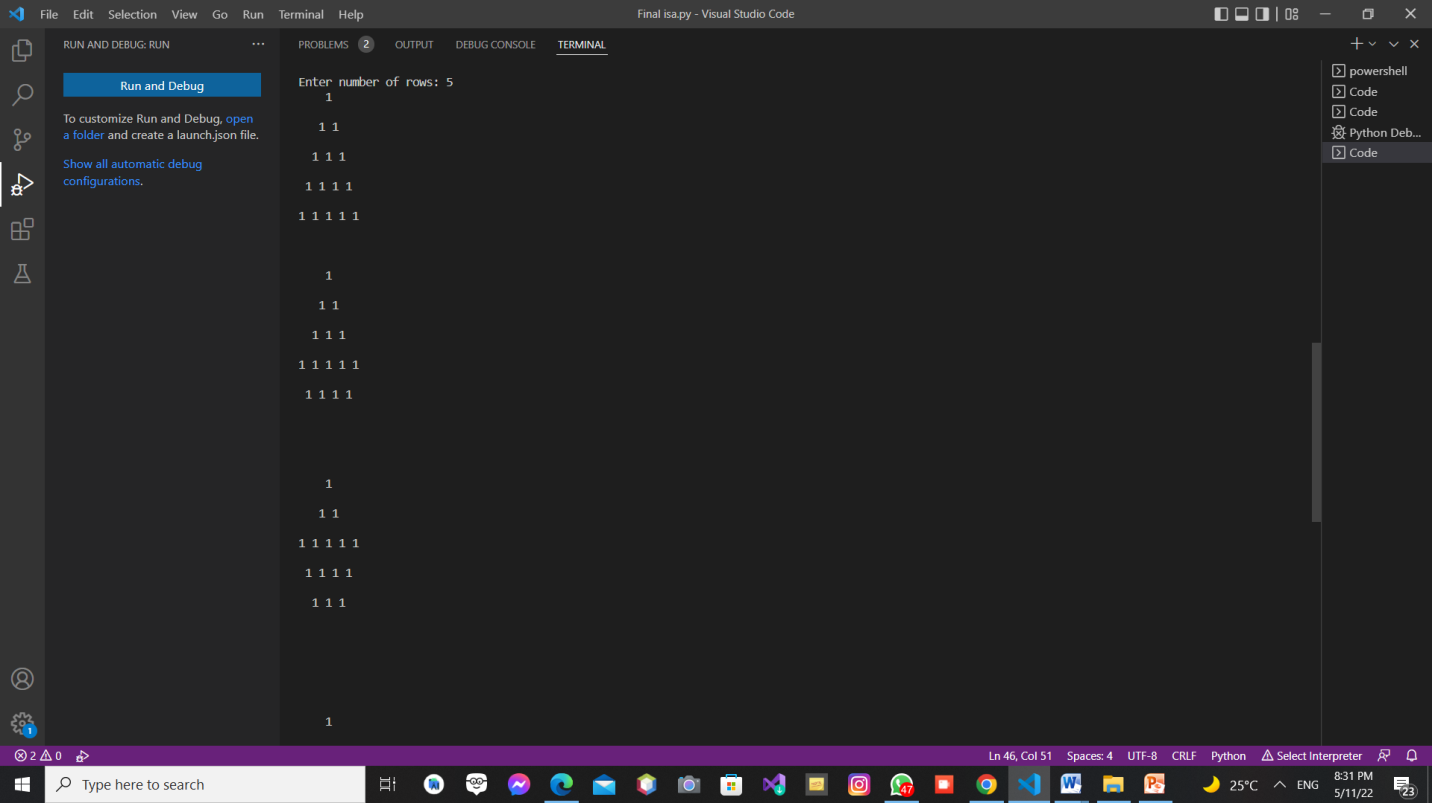


Figure 3: Output for number of rows = 4

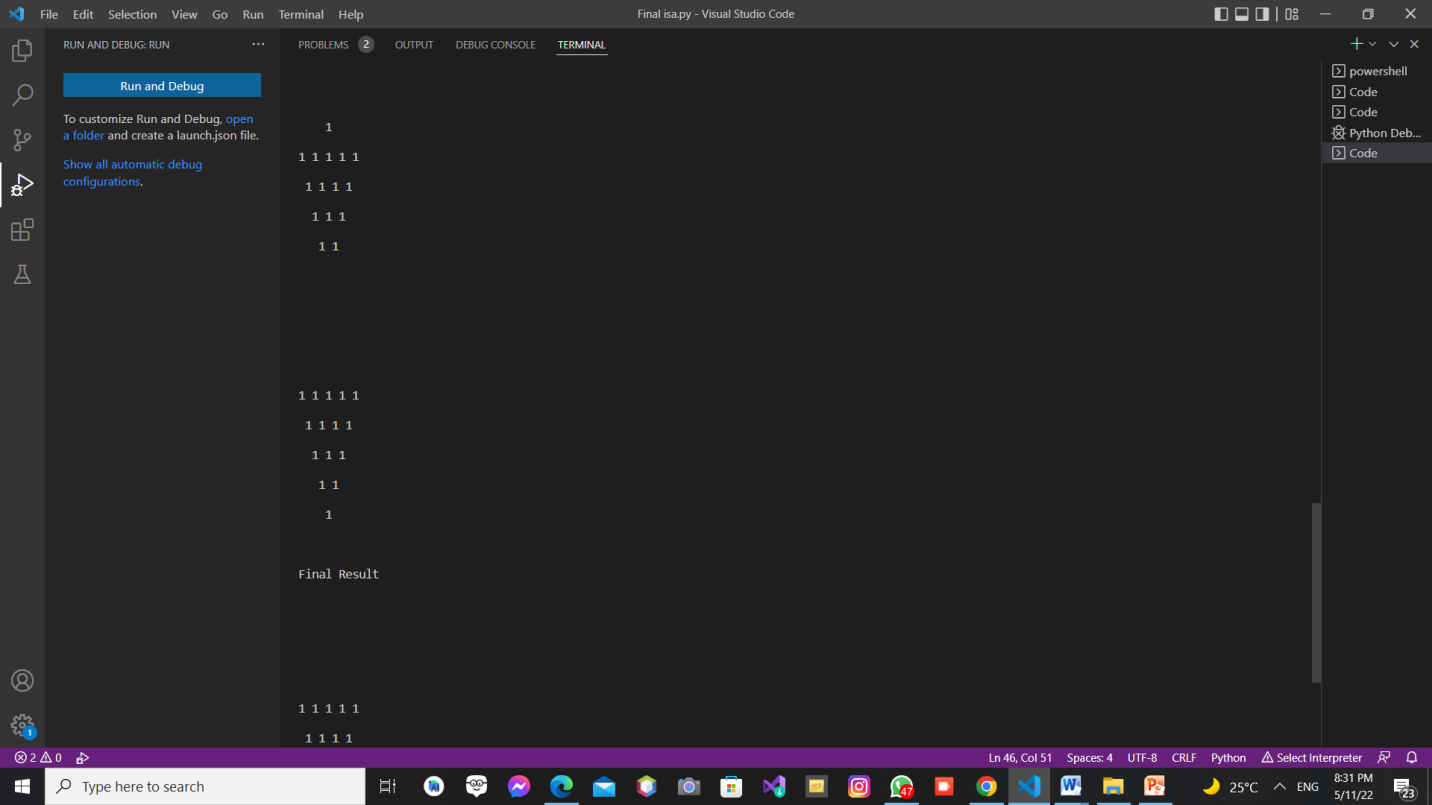
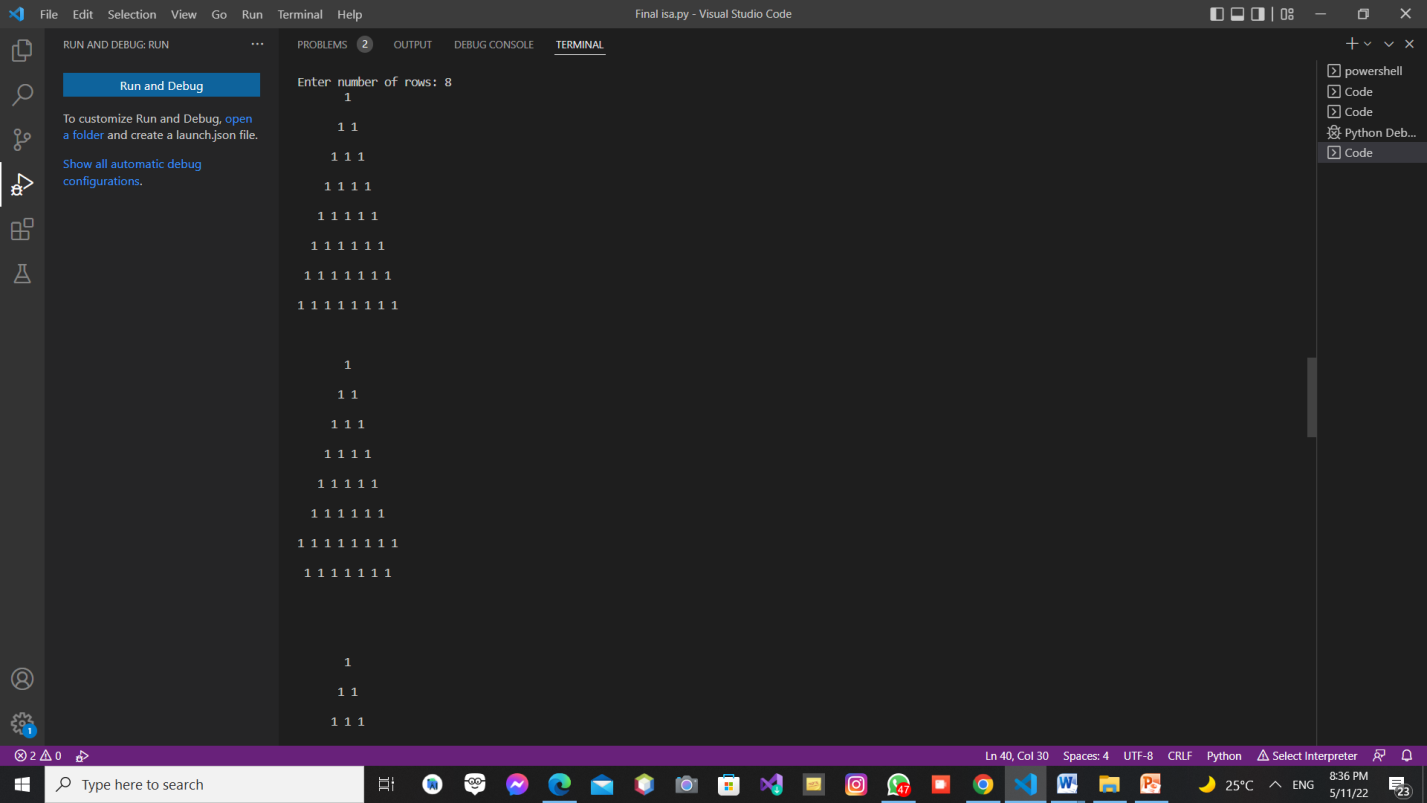
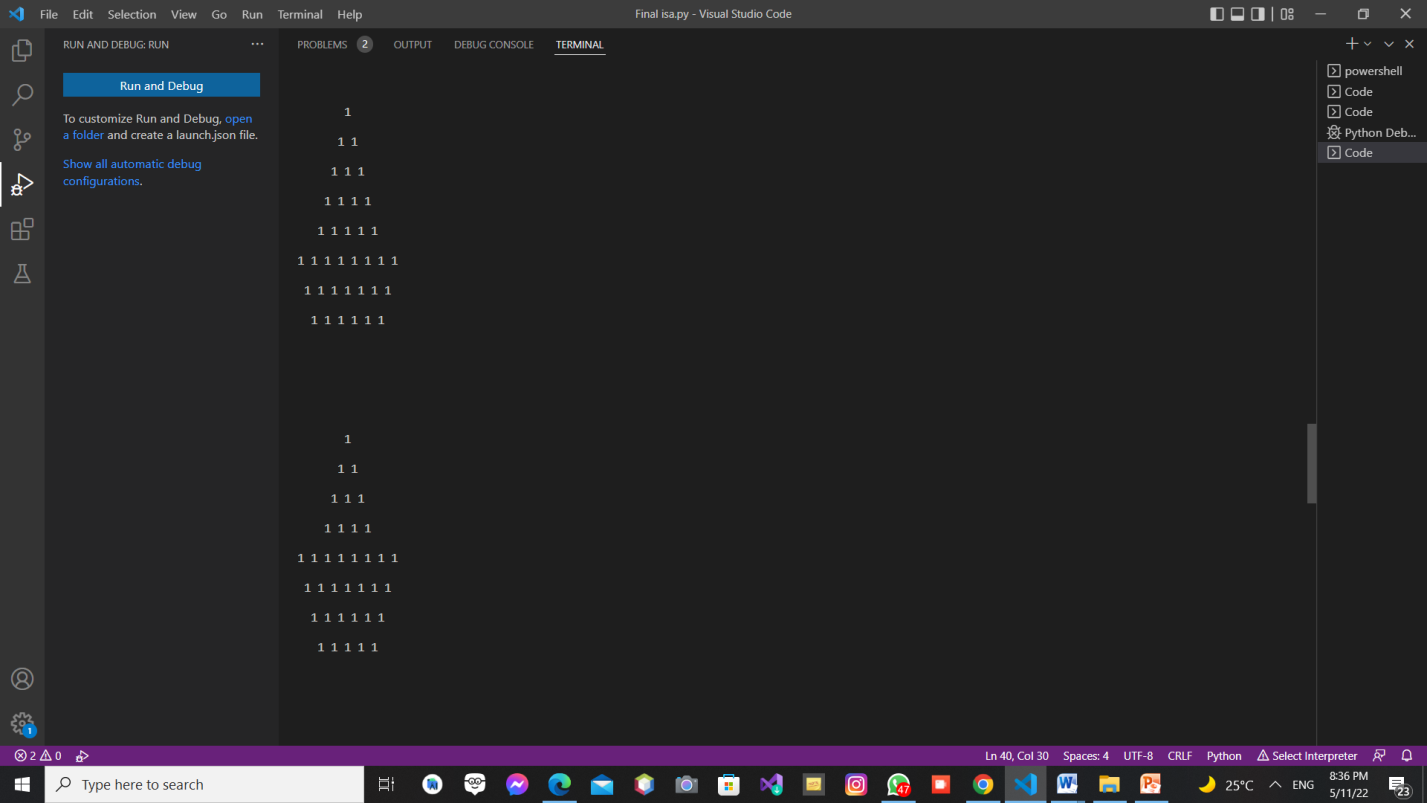
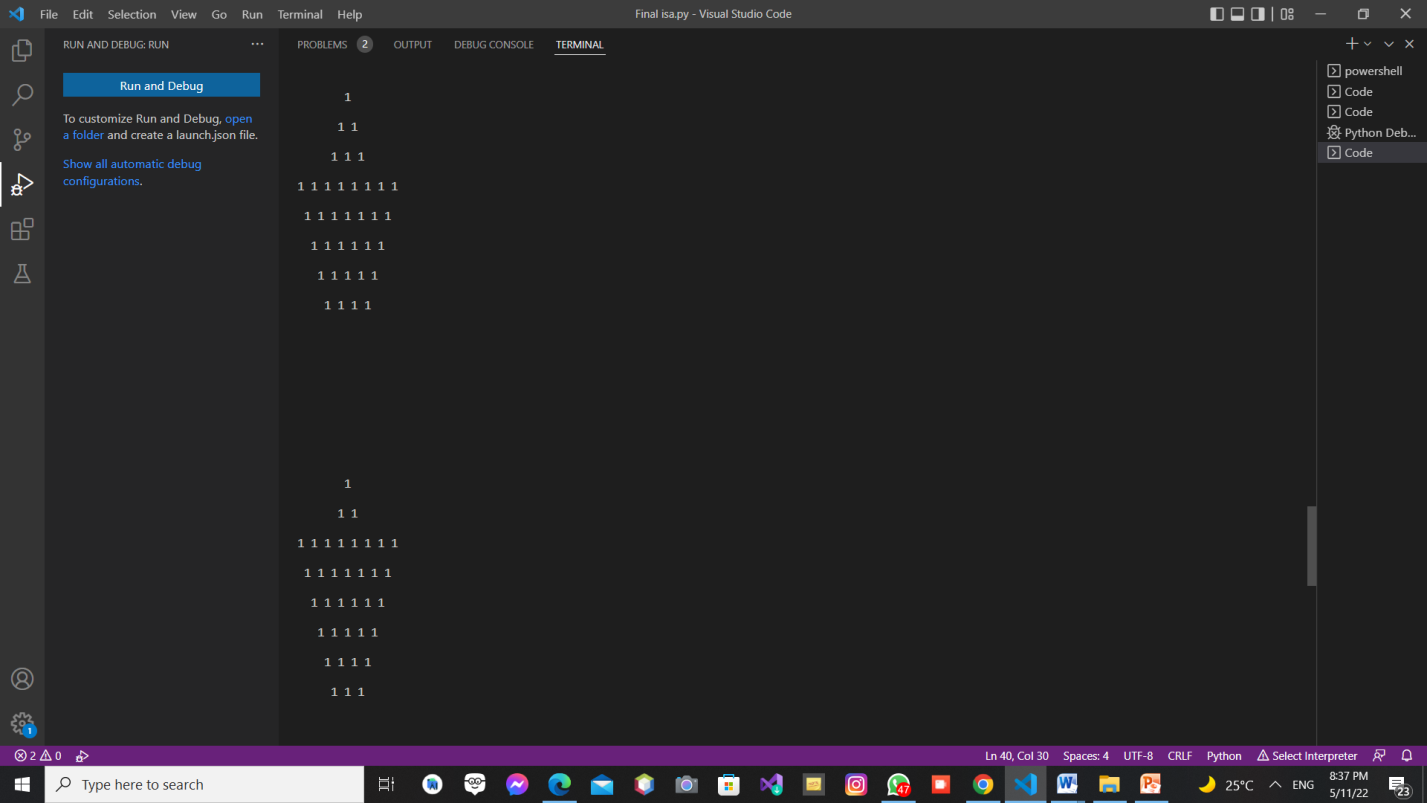


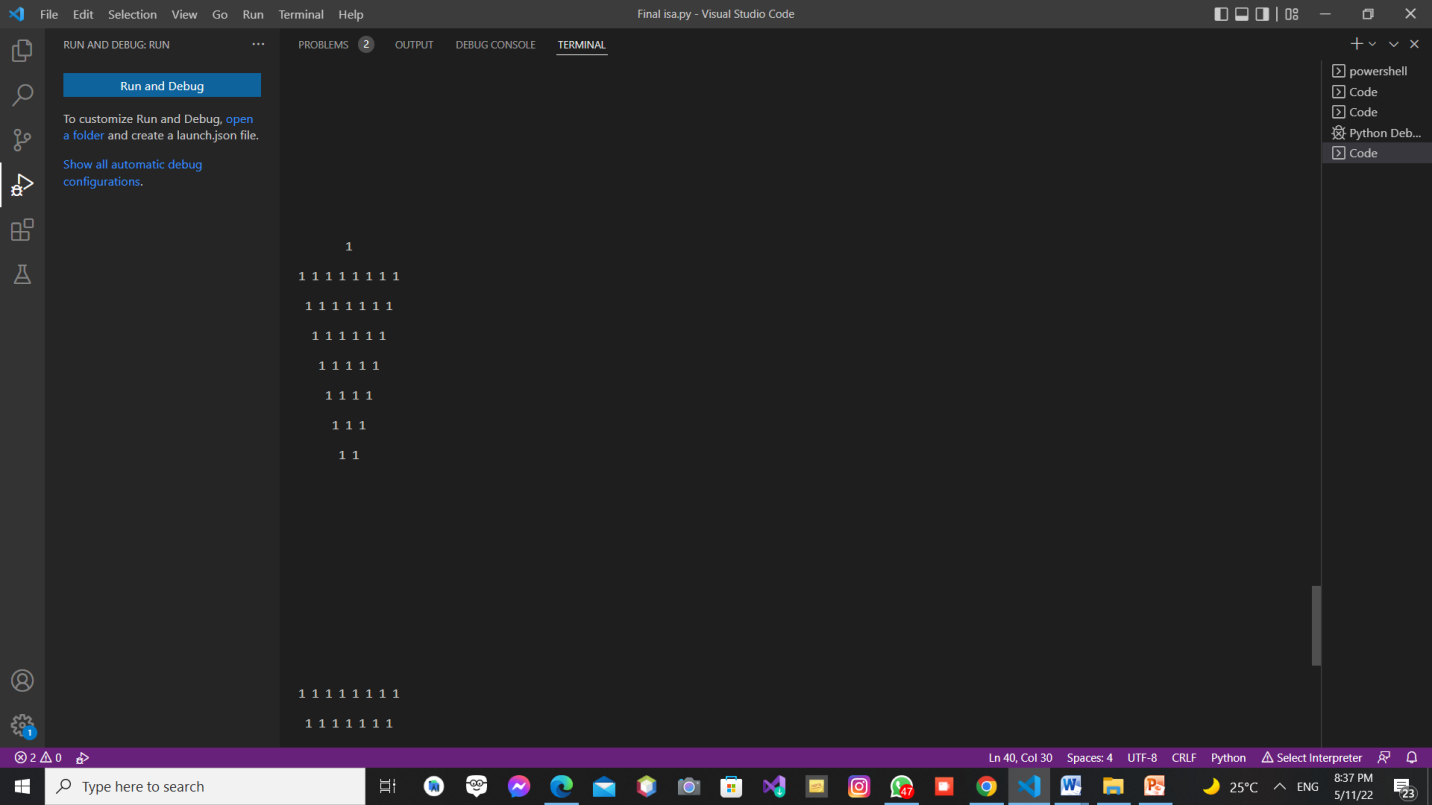


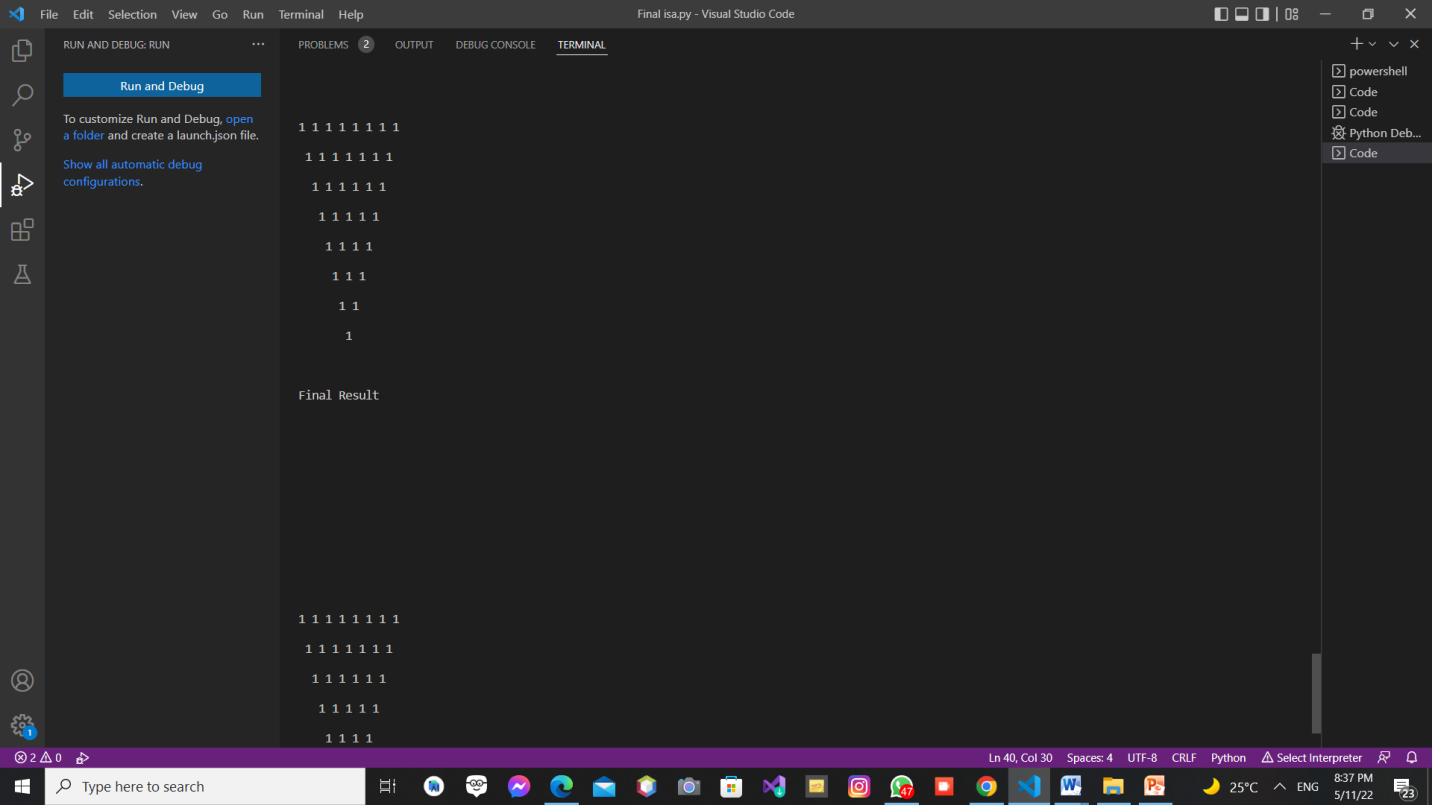
Figure 4: Output for number of rows = 5











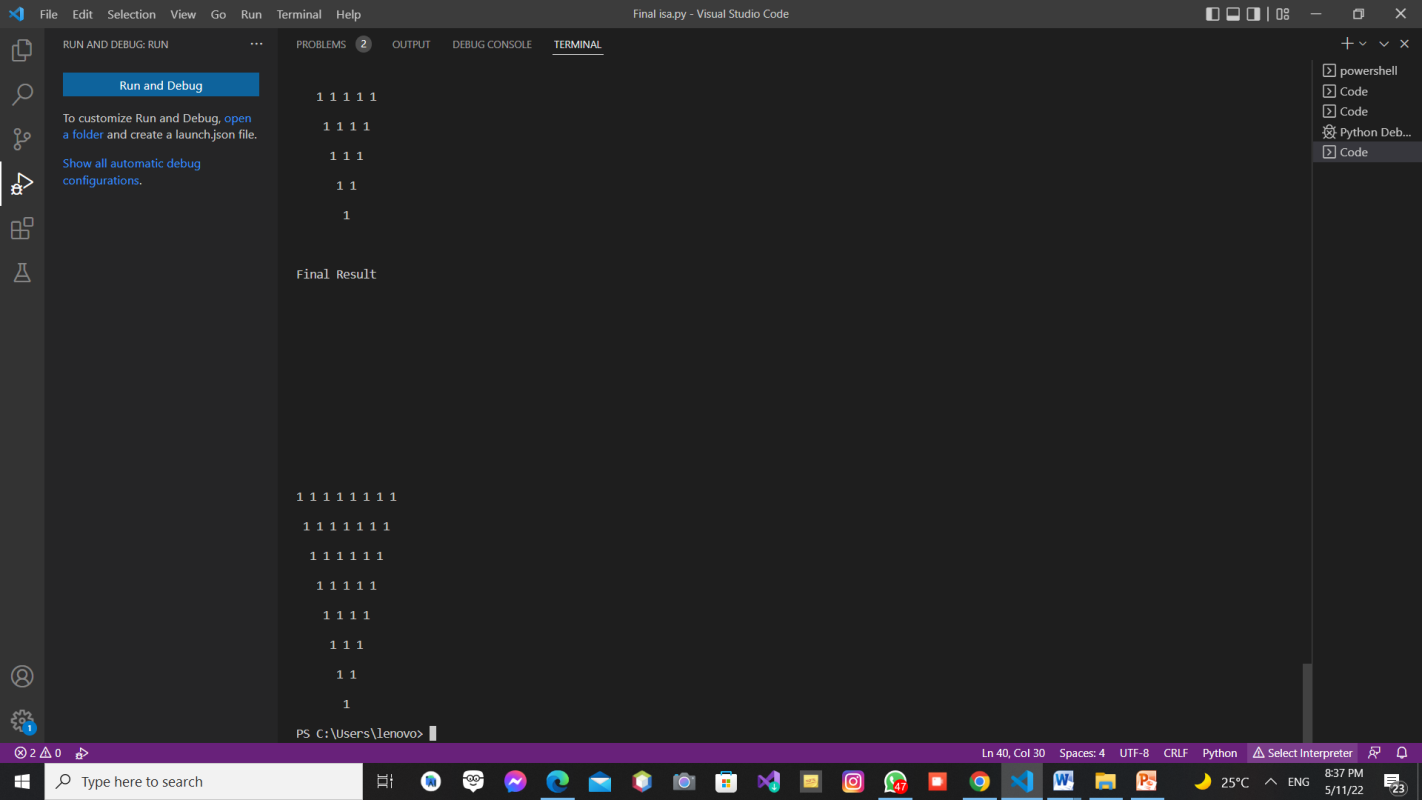
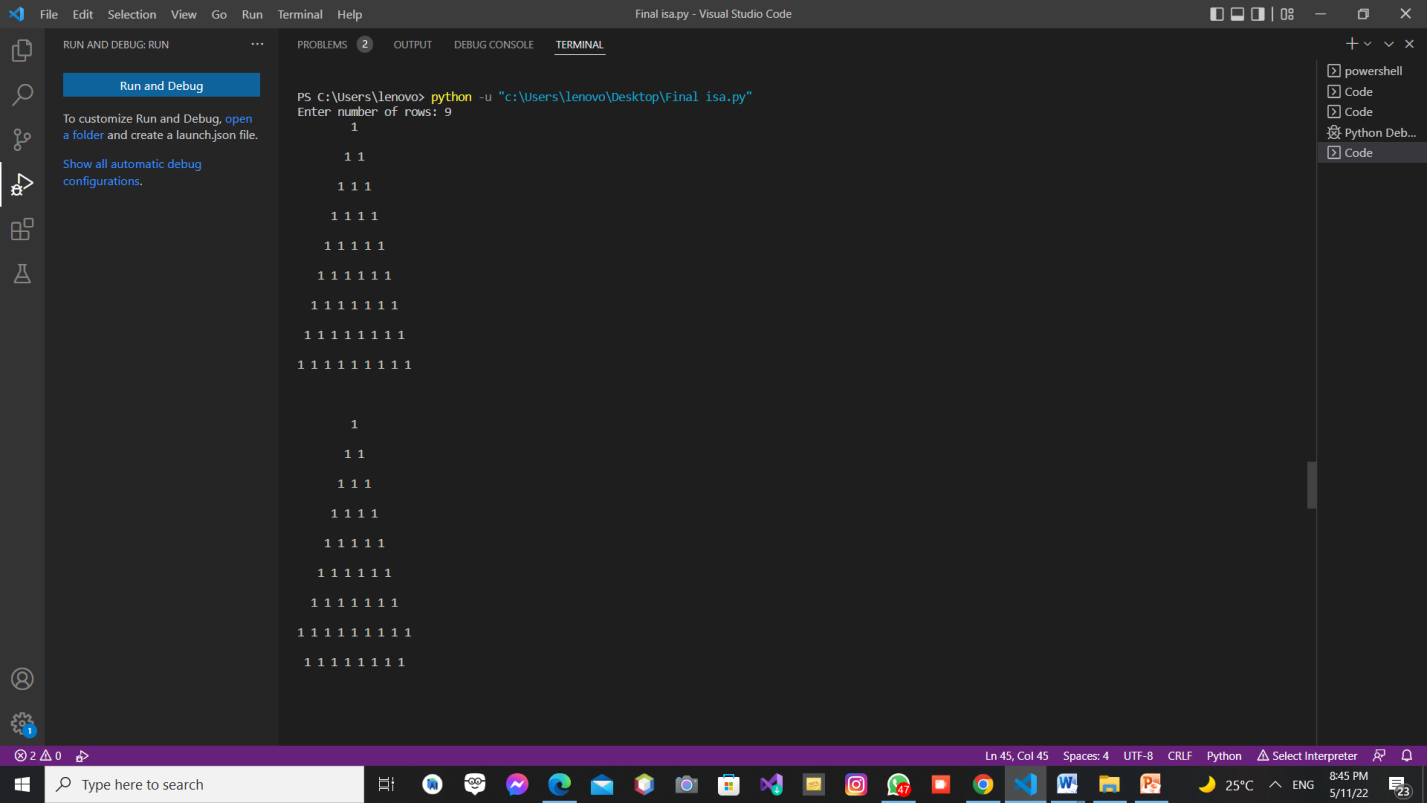
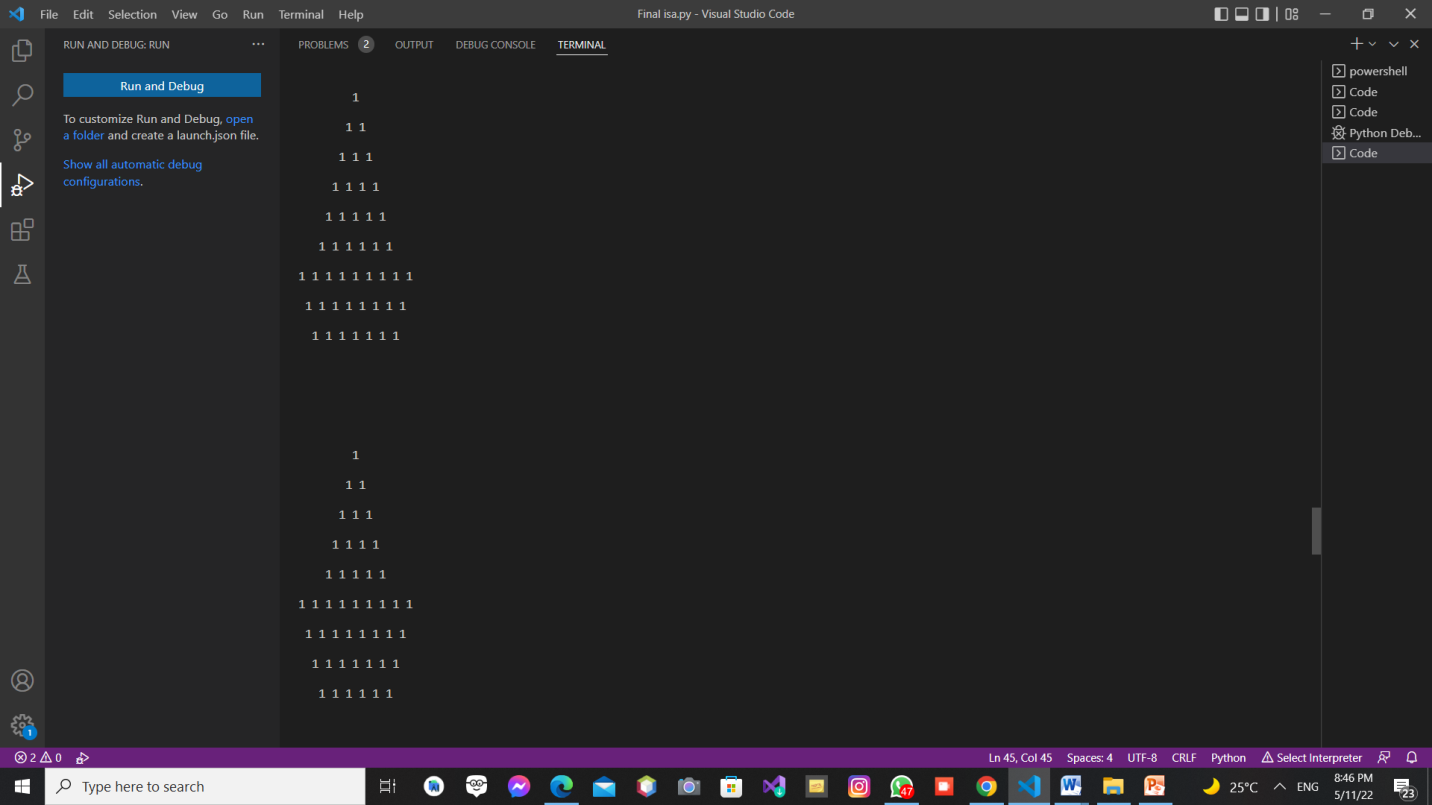
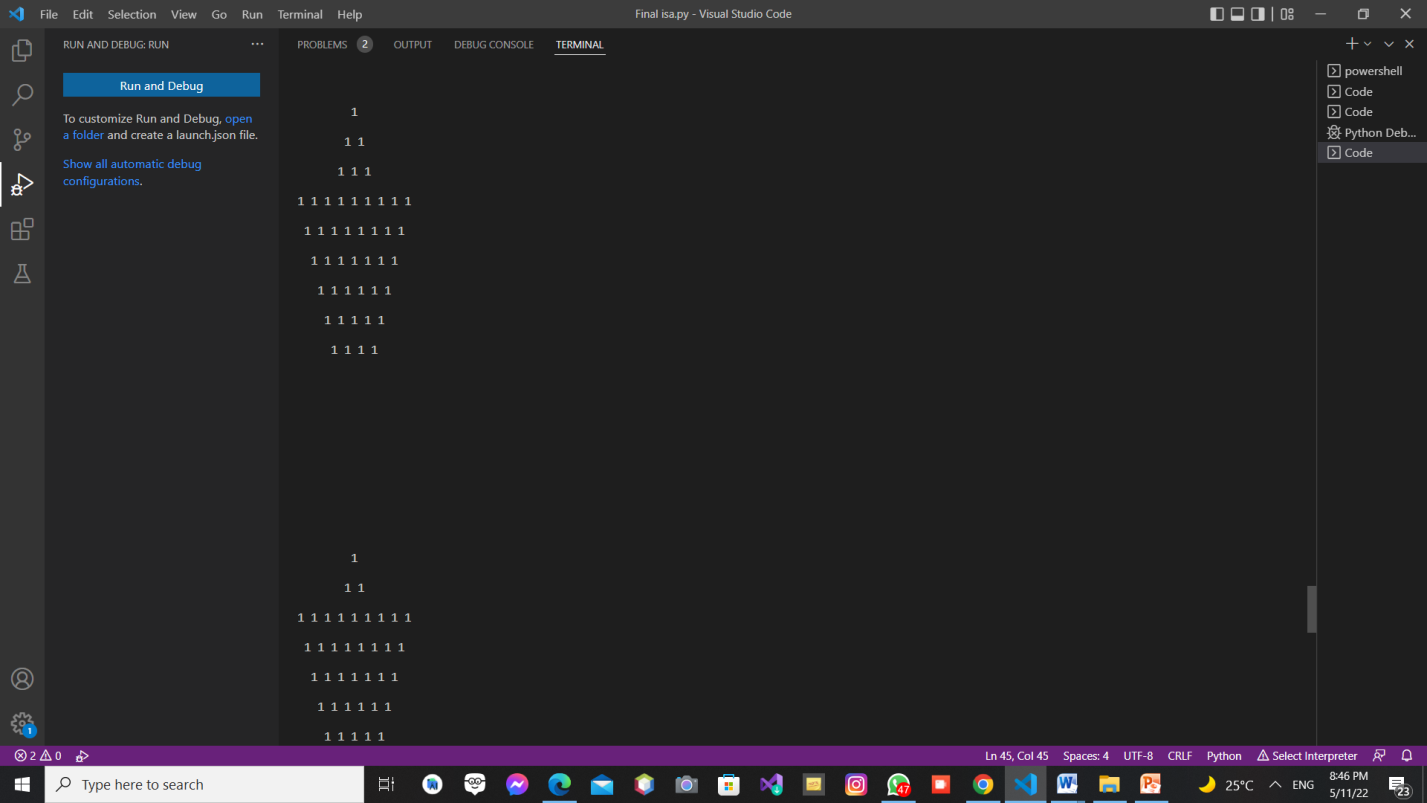
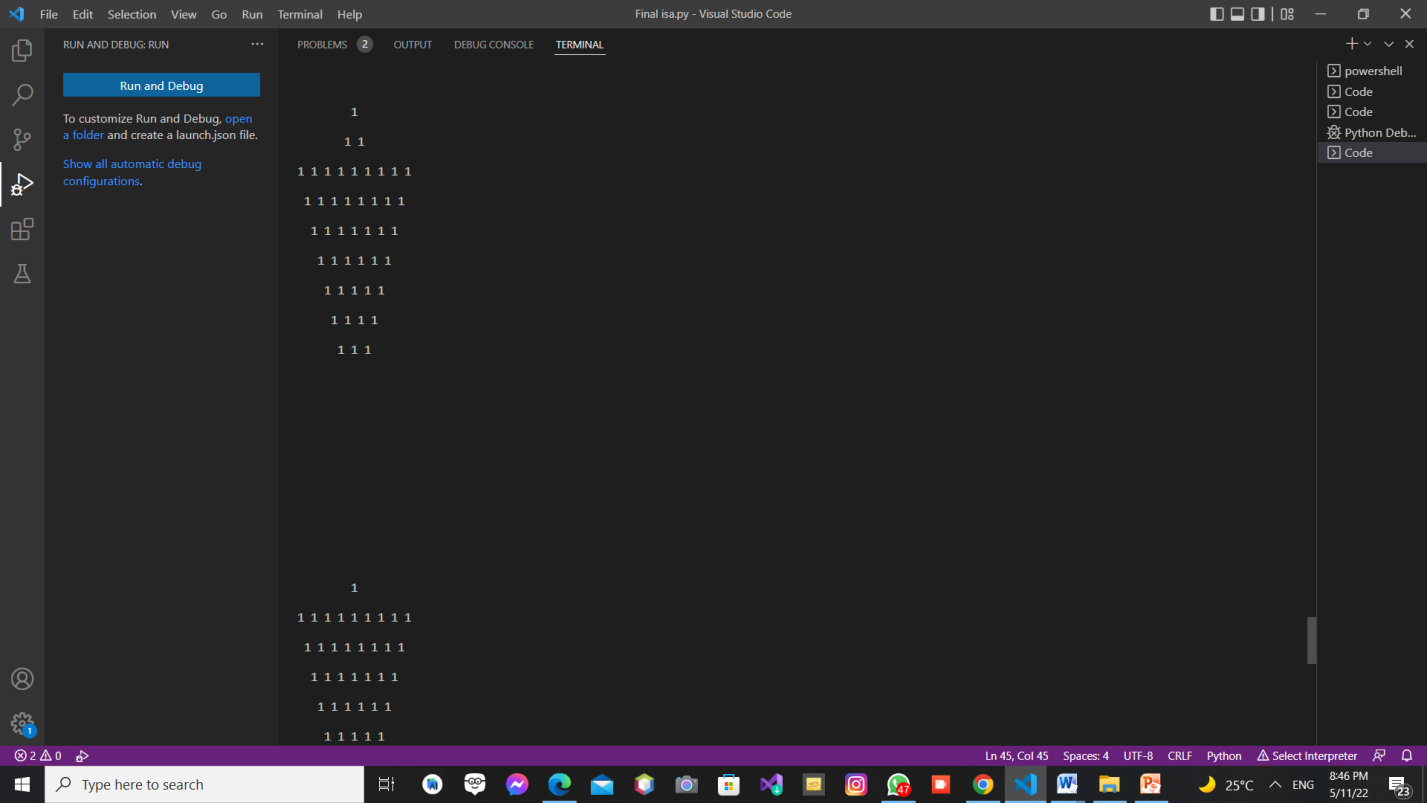


Figure 5: Output for number of rows = 8









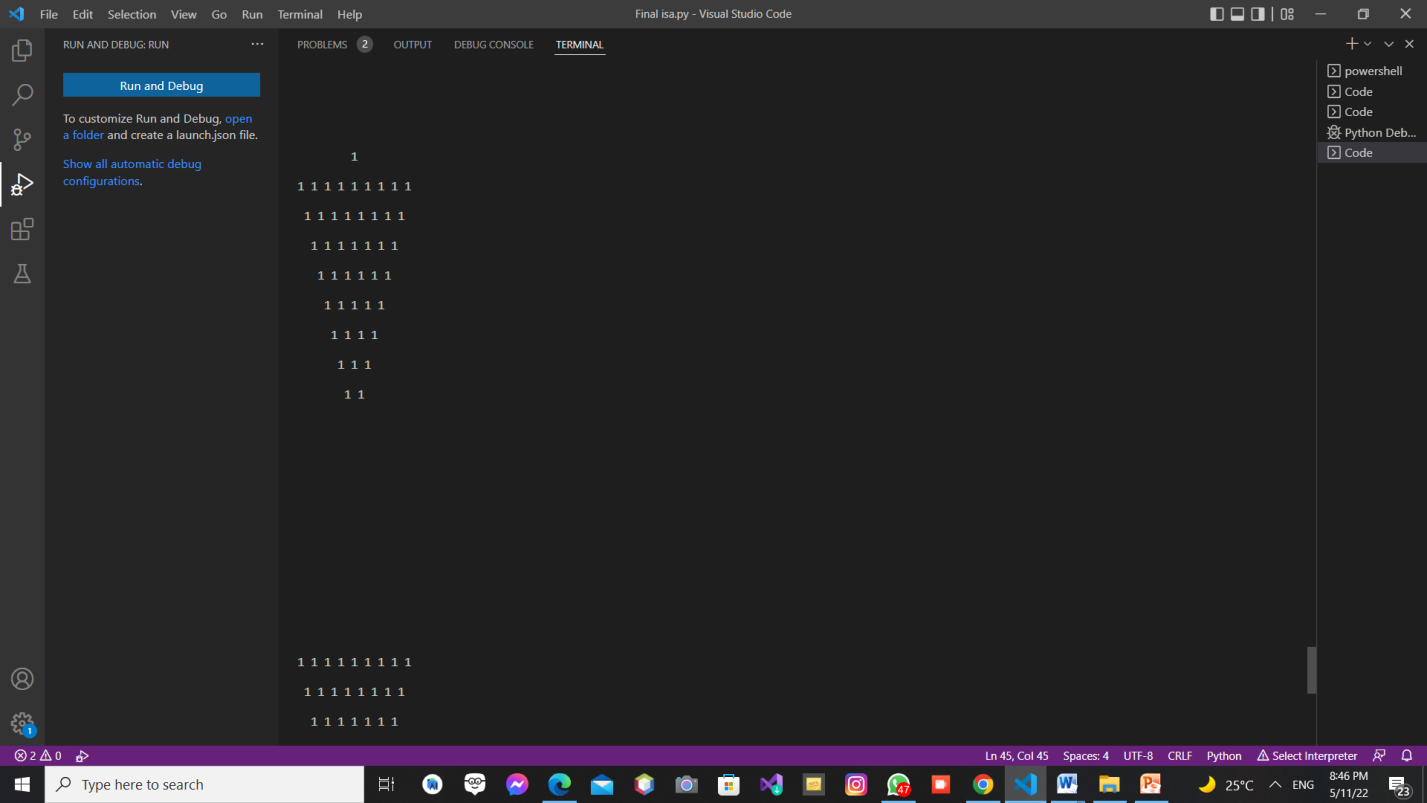
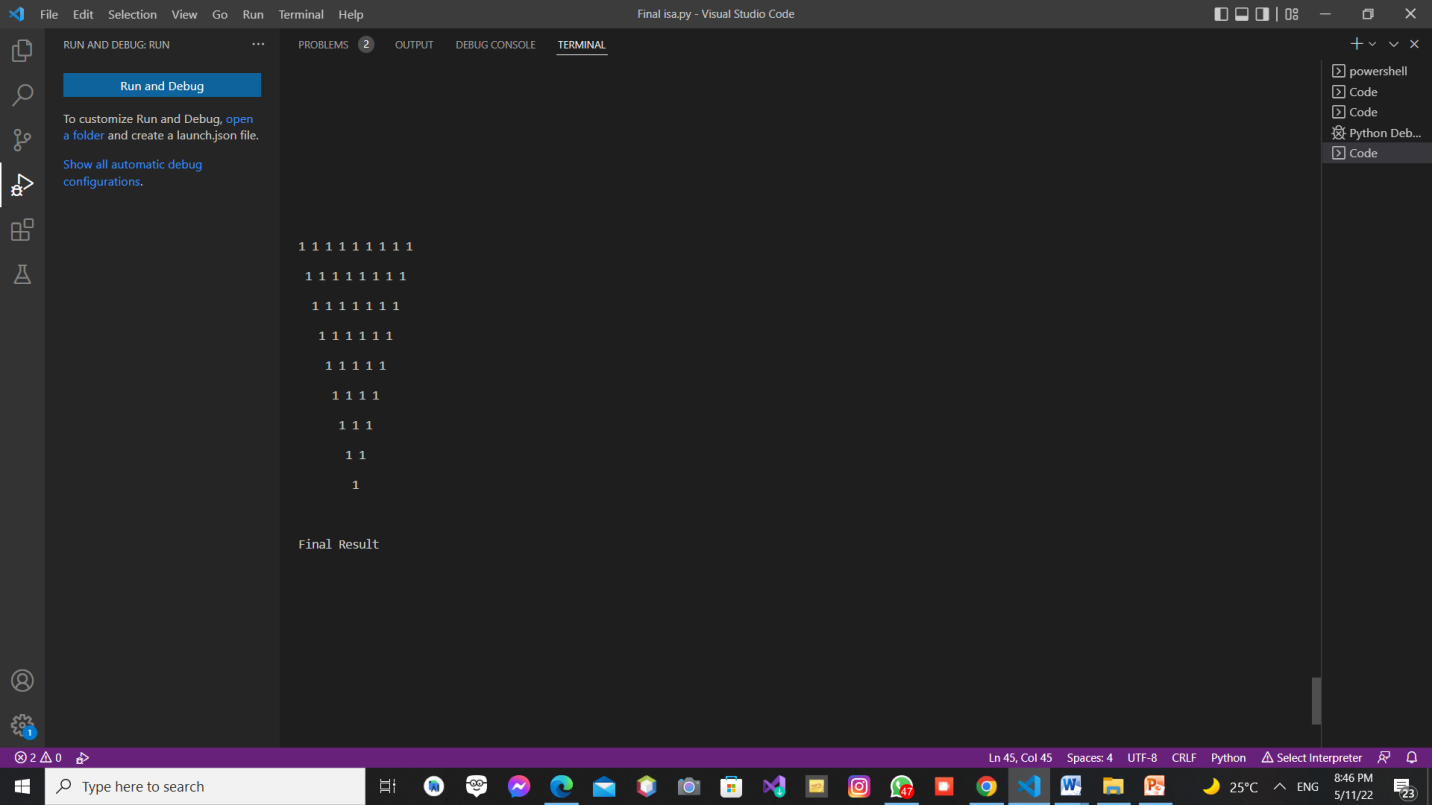
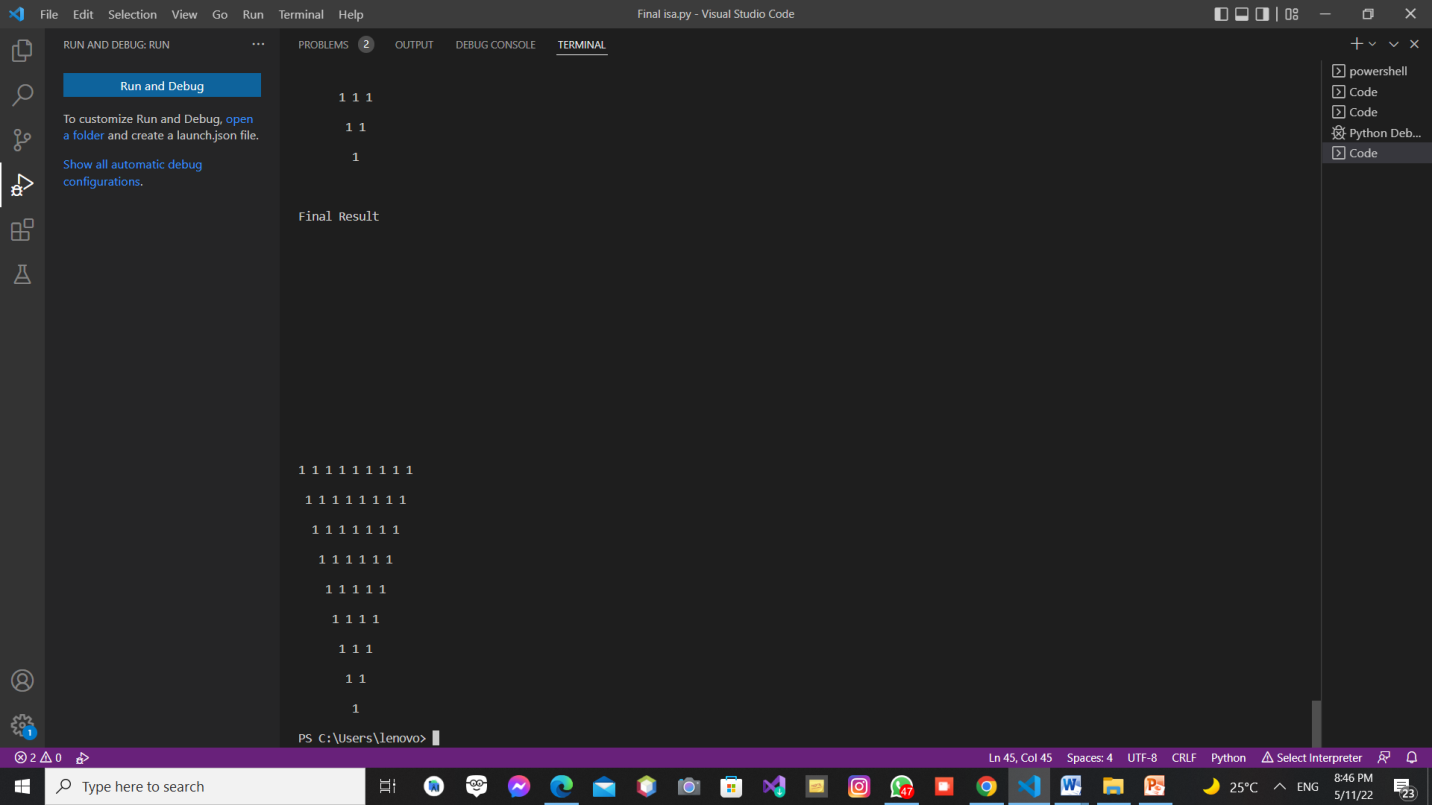
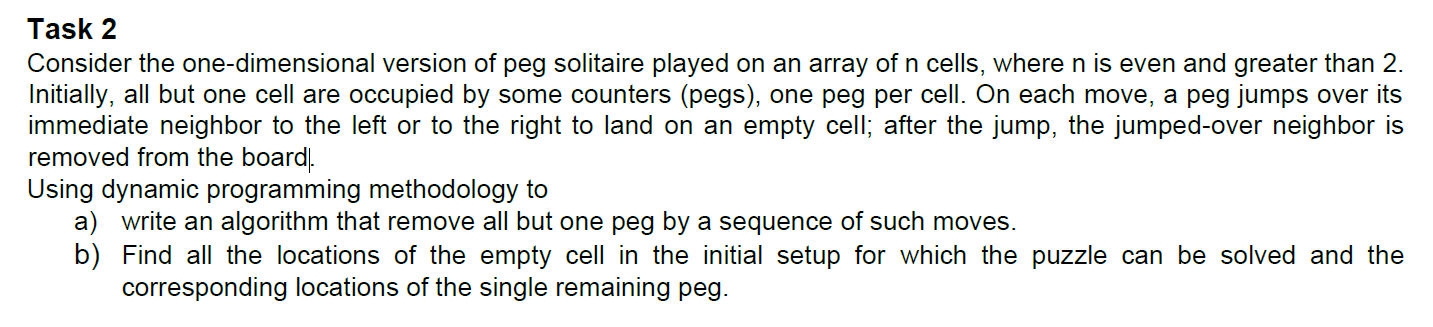


Figure 6: Output for number of rows = 9

## Task 2

### Problem description



### Solution

#### Pseudocode

action when left has 2 1's and right has even number of 1's which is greater than 2  
def l2rlargerequal2even(board , firstzeroindex,start):  
 board[start]

board[start+2] ←board[start+2]

board[start]  
 board[start+1] ←0  
 print(board)  
 board[firstzeroindex+1]

board[firstzeroindex+3] ←board[firstzeroindex+3]

board[firstzeroindex+1]  
 board[firstzeroindex+2] ←0  
 print(board)  
  
#action when left has no 1's and right has 2 1's  
def l0r2(board , start , end):  
 board[end]

board[start+1] ←board[start+1]

board[end]  
 board[start+2] ←0  
 print(board)  
  
#action when right has no 1's and left has 2 1's  
def r0l2(board , start ):  
 board[start]

board[start+2] ←board[start+2]

board[start]  
 board[start+1] ←0  
 print(board)

# actions to do when empty cell position wont lead to a board with only one peg  
def restofstates():  
 print("The choice of empty cell doesn't allow board to be reduced to only 1 peg","\n", "For a board to be reduced choices of empty cell must be 2 , 5 , n-1 and n-4 only" )  
  
# action to do when there is 2 1's on the right and left side of 2 zeros has even number of 1's and greater than 2  
def r2llargerequal2even(board , firstzeroindex,end):  
 board[end]

board[end-2]=board[end-2]

board[end]  
 board[end-1]=0  
 print(board)  
 board[firstzeroindex]

board[firstzeroindex-2] ←board[firstzeroindex-2]

board[firstzeroindex]  
 board[firstzeroindex-1]=0  
 print(board)  
  
  
# dictionary that stores the states the peg board will be in and also stores  
# the appropriate action for each state  
# l = left , r = right , and number donates how many 1's  
solDict={  
 'l0r2': l0r2,  
 'l2r>=2even':l2rlargerequal2even ,  
 'r0l2': r0l2,  
 'r2l>=2even': r2llargerequal2even,  
 'rest':restofstates  
 # 'l0r>2': ,  
 # 'l1r>=1': ,  
 # 'l>2r>=1': ,  
 # 'r0l>2': ,  
 # 'r1l>=1': ,  
 # 'r>2l>=1':  
}  
  
  
# recursive function that solves according to the above constraints  
def solve(board , start , end):  
 left = 0  
 right = 0  
 index = start  
 zeroindex = 0  
  
  
 #stop when reached the end of the board  
 if (start >= end ):  
 return  
  
 # count how many 1's on the left of the 2 zeros  
  
 for i to range(start,end+1) do  
 if board[i] == 1:  
 left ← left + 1  
 elif board[i] == 0:  
 zeroindex = index  
 break  
 index ← index + 1  
  
 # count how many 1's on the right of the 2 zeros  
  
 for i to range(zeroindex, end+1) do   
 if board[i] == 1:  
 right ← right + 1  
  
  
 # choose from Dictionary the appropriate action to make  
  
 if(left==0 and right==2):  
 findInDict = 'l0r2'  
 solDict[findInDict](board,start,end)  
 else (left==2 and right >= 2 and right % 2 == 0):  
 findInDict = 'l2r>=2even'  
 solDict[findInDict](board,zeroindex,start)  
 # recursive call to the solve function sending it a subproblem  
 solve(board,start+2,end)  
 else (right == 0 and left == 2):  
 findInDict = 'r0l2'  
 solDict[findInDict](board, start)  
 else (right==2 and left >= 2 and left % 2 == 0):  
 findInDict = 'r2l>=2even'  
 solDict[findInDict](board, zeroindex, end)  
 # recursive call to the solve function sending it a subproblem  
 solve(board, start, end-2)  
 else:  
 findInDict = 'rest'  
 solDict[findInDict]()  
  
  
# intial move to place 2 zeros next to eachother to begin solution then call solve function  
def initialmove(board , empty , n):  
  
 if (empty == n-4):  
 board[empty + 2]

board[empty] ← board[empty]

board[empty + 2]  
 board[empty + 1] = 0  
  
 else empty == 1 or empty == 0 :  
 board[empty + 2]

board[empty] ← board[empty]

board[empty + 2]  
 board[empty + 1] = 0  
 else:  
 board[empty-2],board[empty]=board[empty],board[empty-2]  
 board[empty-1]=0  
  
 print(board)  
 solve(board, 0, len(board) - 1)  
  
  
  
# check if user entered odd number or 2 or 0  
def checknumberofcells(n):  
 if n % 2 != 0 or n == 0 or n == 2 :  
 return True

#### Code

1. from time import perf\_counter  
   # action when left has 2 1's and right has even number of 1's which is greater than 2  
   def l2rlargerequal2even(board , firstzeroindex,start):  
    board[start],board[start+2]=board[start+2],board[start]  
    board[start+1]=0  
    print(board)  
    board[firstzeroindex+1],board[firstzeroindex+3]=board[firstzeroindex+3],board[firstzeroindex+1]  
    board[firstzeroindex+2]=0  
    print(board)  
     
   #action when left has no 1's and right has 2 1's  
   def l0r2(board , start , end):  
    board[end],board[start+1]=board[start+1],board[end]  
    board[start+2]=0  
    print(board)  
     
   #action when right has no 1's and left has 2 1's  
   def r0l2(board , start ):  
    board[start],board[start+2]=board[start+2],board[start]  
    board[start+1]=0  
    print(board)  
     
   # actions to do when empty cell position wont lead to a board with only one peg  
   def restofstates():  
    print("The choice of empty cell doesn't allow board to be reduced to only 1 peg","\n", "For a board to be reduced choices of empty cell must be 2 , 5 , n-1 and n-4 only" )  
     
   # action to do when there is 2 1's on the right and left side of 2 zeros has even number of 1's and greater than 2  
   def r2llargerequal2even(board , firstzeroindex,end):  
    board[end],board[end-2]=board[end-2],board[end]  
    board[end-1]=0  
    print(board)  
    board[firstzeroindex],board[firstzeroindex-2]=board[firstzeroindex-2],board[firstzeroindex]  
    board[firstzeroindex-1]=0  
    print(board)  
     
     
   # dictionary that stores the states the peg board will be in and also stores  
   # the appropriate action for each state  
   # l = left , r = right , and number donates how many 1's  
   solDict={  
    'l0r2': l0r2,  
    'l2r>=2even':l2rlargerequal2even ,  
    'r0l2': r0l2,  
    'r2l>=2even': r2llargerequal2even,  
    'rest':restofstates  
    # 'l0r>2': ,  
    # 'l1r>=1': ,  
    # 'l>2r>=1': ,  
    # 'r0l>2': ,  
    # 'r1l>=1': ,  
    # 'r>2l>=1':  
   }  
     
     
   # recursive function that solves according to the above constraints  
   def solve(board , start , end):  
    left = 0  
    right = 0  
    index = start  
    zeroindex = 0  
     
     
    #stop when reached the end of the board  
    if (start >= end ):  
    return  
     
    # count how many 1's on the left of the 2 zeros  
     
    for i in range(start,end+1):  
    if board[i] == 1:  
    left = left + 1  
    elif board[i] == 0:  
    zeroindex = index  
    break  
    index = index + 1  
     
    # count how many 1's on the right of the 2 zeros  
     
    for i in range(zeroindex, end+1):  
    if board[i] == 1:  
    right = right + 1  
     
     
    # choose from Dictionary the appropriate action to make  
     
    if(left==0 and right==2):  
    findInDict = 'l0r2'  
    solDict[findInDict](board,start,end)  
    else (left==2 and right >= 2 and right % 2 == 0):  
    findInDict = 'l2r>=2even'  
    solDict[findInDict](board,zeroindex,start)  
    # recursive call to the solve function sending it a subproblem  
    solve(board,start+2,end)  
    else (right == 0 and left == 2):  
    findInDict = 'r0l2'  
    solDict[findInDict](board, start)  
    else (right==2 and left >= 2 and left % 2 == 0):  
    findInDict = 'r2l>=2even'  
    solDict[findInDict](board, zeroindex, end)  
    # recursive call to the solve function sending it a subproblem  
    solve(board, start, end-2)  
    else:  
    findInDict = 'rest'  
    solDict[findInDict]()  
     
     
   # intial move to place 2 zeros next to eachother to begin solution then call solve function  
   def initialmove(board , empty , n):  
     
    if (empty == n-4):  
    board[empty + 2], board[empty] = board[empty], board[empty + 2]  
    board[empty + 1] = 0  
     
    else empty == 1 or empty == 0 :  
    board[empty + 2]

board[empty] ← board[empty]

board[empty + 2]  
 board[empty + 1] = 0  
 else:  
 board[empty-2],board[empty]=board[empty],board[empty-2]  
 board[empty-1]=0  
  
 print(board)  
 solve(board, 0, len(board) - 1)  
  
  
  
# check if user entered odd number or 2 or 0  
def checknumberofcells(n):  
 if n % 2 != 0 or n == 0 or n == 2 :  
 return True  
  
  
def main():  
 n = int(input("Enter Number Of Cells : "))  
 if checknumberofcells(n):  
 print("N should be Even and Greater than 2")  
 else:  
 # initialize board and put empty cell in its place  
 board = [1]\*n  
 print(board)  
 empty = int(input("Choose Position Of Empty Cell : "))  
 board[empty-1]=0  
 print(board)  
 initialmove(board,empty-1 ,n-1)

#### Solution description

The board contains even number of cells from 1 to n where the empty cell is located between 2 and 5 where its symmetrically as it could be n-1 or n-4. The solution of the problem occurs in 4 cases to apply dynamic programming and as the array consist of 0 and 1.

First case: if the left side is 0 and the right is 2 as [0011] or vise versa and will only one single peg will be on the board so it’s a success case

Second case: if l =1 and r ≥ 1 as [101---1] or vise versa as r=1 and l ≥ 1 by solving the puzzle the remaining will be more than one peg so it will be a dead-end case

Third case: if l = 2 and r ≥ 2 and r is an even number or vise versa as [11001-1] by solving the puzzle one peg will remain so it’s another success case

Fourth case: if l > 2 and r ≥ 1 as [ 1----110011-1] where the array is divided into two arrays

A: [1-----110] and B: [011-----1] and with each implementation reduce the two 0’s in the furthermost in the arrays or vise versa but this will occur when both A and B is joined it will contain two ones so it’s a dead-end case.

#### Complexity analysis

#### Comparison between another algorithm

def generateBoard(n):

return [1]\*n

def solve(board):

if checkBoard(board):

return True

elif checkUnsolvable(board):

return False

moves = []

for i in range(len(board)):

if i < len(board)-2:

if board[i] and board[i+1] and not board[i+2]:

moves.append((i, 'right'))

if i > 1:

if board[i] and board[i-1] and not board[i-2]:

moves.append((i, 'left'))

for move in moves:

newBoard = makeMove(board, move)

if solve(newBoard):

return True

continue

return False

def makeMove(board, move):

index, direction = move

b = [element for element in board]

if direction == 'right':

b[index] = 0

b[index+1] = 0

b[index+2] = 1

elif direction == 'left':

b[index] = 0

b[index-1] = 0

b[index-2] = 1

return b

def checkBoard(board):

if sum(board) == 1:

return True

return False

def checkUnsolvable(board):

expression1 = '1000+1' #RE for a proven to be unsolvable board

expression2 = '00100' #RE for a proven to be unsolvable board

string = ''.join([str(element) for element in board])

if re.search(expression1, string) or re.search(expression2, string):

return True

return False

def countSolutions(board):

indices = []

for i in range(len(board)):

b = [element for element in board]

b[i] = 0

if solve(b):

indices.append(i+1)

return indices

n = int(input())

print(countSolutions(generateBoard(n)))

#### 1.2.2.6 Sample of the output

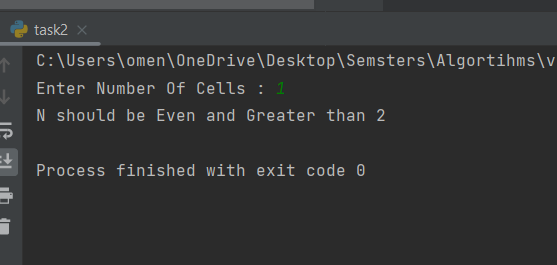


Figure 7: Output for number of cells = 1

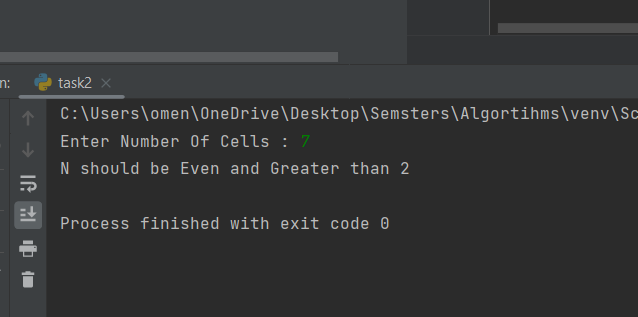
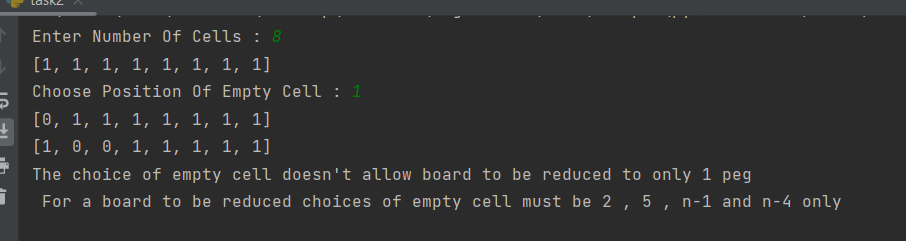
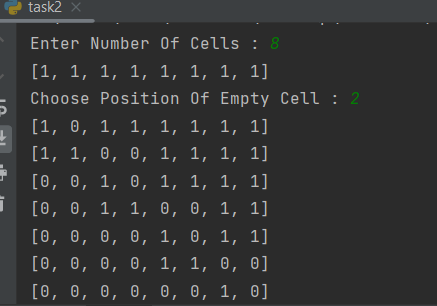
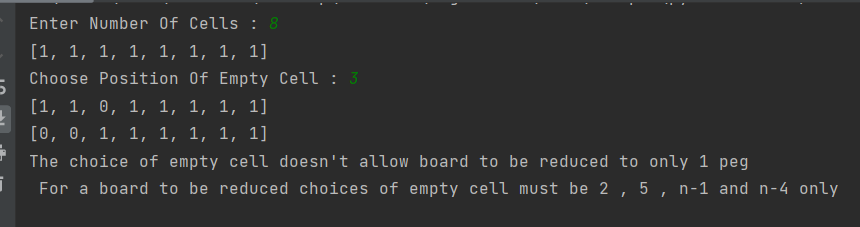
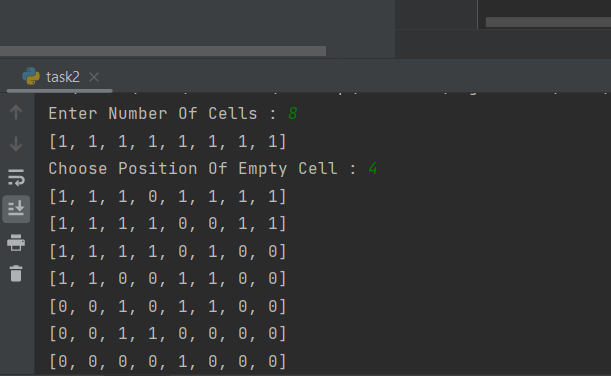


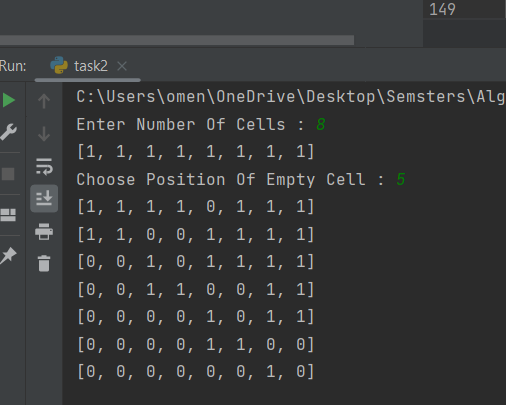
Figure 8: Output for number of cells = 7

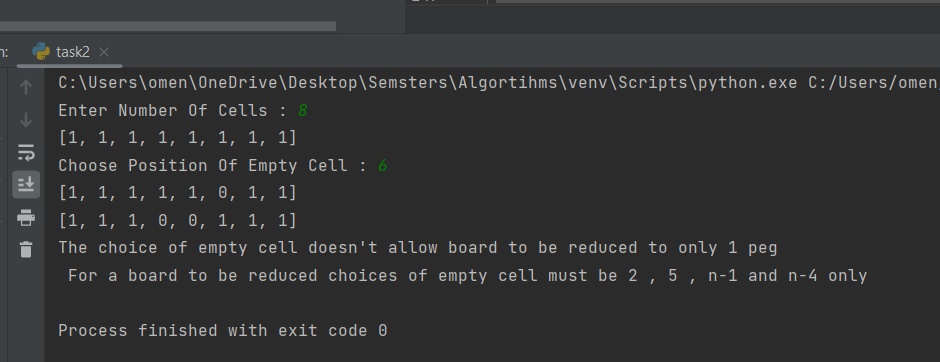


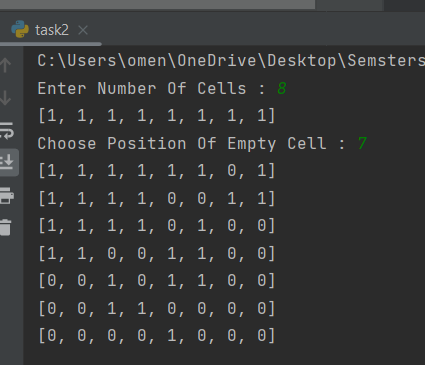












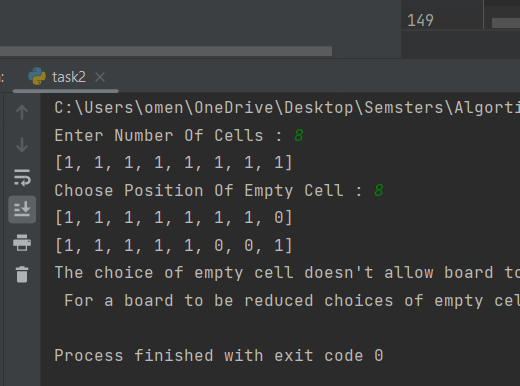


Figure 9: Output for number of cells = 8

## Task 3

### Problem description

Text

Description automatically generated

### Solution

#### Pseudocode

//initialize the board with knights in their places

board <-- [(1, 'b1'), (2, 'b2'), (3, 'b3'), (4, 'null'), (5, 'null'), (6, 'null'), (7, 'null'), (8, 'null'),

(9, 'null'), (10, 'w1'), (11, 'w2'), (12, 'w3')

create an object of class task3 with its board = board

object.divide1()

object.divide2()

class task 3:

task3(board){

this.board=board

}//initialization constructor

//this function is responsible to move the knights correctly according to the adjacency grapgh //of the knights' movements and ensures that no knights overlap, returns true if move was //sucessful, false otherwise

function move(knight,destination){

pos = self.position(knight) // a function that returns the position of a given knight

// 12 if conditions to ensure that the movements are according to the adjacency grapgh

if pos == 1

if destination not in (6, 8)

return False

if pos == 2

if destination not in (9, 7)

return False

.

.

.

if pos == 12

if destination not in (5, 7)

return False

if board[destination][2] == 'null'{ //if destination is empty

board[pos][2] ='null' // set current position to be empty

board[destination][2] = k // move knight to destination

return True

}

else

return False

}

// function to return a knight's position

position(k){

for i in board:

if i[1] == k

return i[0]

return 0

}

//first subproblem, 2knights and 6positions

divide1(){

anticlk = [9, 4, 11, 6, 7, 2] // the 6 positions which the function is going to maneuver

move('w2', 6) // initial move to easilymove the rest

for i in anticlk: // move the black knight until it is in the right position

if position('b2') == 11

break

move('b2', i)

for i in anticlk: // move the white knight until it is in the right position

if position('w2') == 2

break

else:

if move('w2', i)

continue

}

// second sub problem, 4knights and 10 positions

divide2(){

anticlk = [1, 6, 7, 12, 5, 10, 9, 4, 3, 8, 1, 6] // the 10 positions with the first 2 repeated to avoid double loops

for i in anticlk: // move the first black knight to allow the wight knights to take its place

if position('b1') == 7

break

move('b1', i)

for i in anticlk: // move the second black knight to allow the wight knights to take its place

if position('b3') == 6

break

move('b3', i)

for i in anticlk: // move the white knights into place

if position('w1') == 1

break

move('w1', i)

for i in anticlk: // move the white knights into place

if position('w3') == 3

break

move('w3', i)

for i in anticlk: // move the black knights into place

if position('b1') == 10

break

move('b1', i)

for i in anticlk: // move the black knights into place

if position('b3') == 12

break

move('b3', i)

}

#### Code

1. class task3:  
     
    def \_\_init\_\_(self, board):  
    self.board = board  
    self.numberofsteps = 0  
     
    def position(self, k):  
    board = self.board  
    for i in board:  
    if i[1] == k:  
    return i[0]  
     
    return 0  
     
    def move(self, k, destination):  
    pos = self.position(k)  
    if pos == 1:  
    if destination not in (6, 8):  
    print('move failed : ' + k + ' to ' + str(destination))  
    return False  
    if pos == 2:  
    if destination not in (9, 7):  
    print('move failed : ' + k + ' to ' + str(destination))  
    return False  
    if pos == 3:  
    if destination not in (8, 4):  
    print('move failed : ' + k + ' to ' + str(destination))  
    return False  
    if pos == 4:  
    if destination not in (3, 9, 11):  
    print('move failed : ' + k + ' to ' + str(destination))  
    return False  
    if pos == 5:  
    if destination not in (10, 12):  
    print('move failed : ' + k + ' to ' + str(destination))  
    return False  
    if pos == 6:  
    if destination not in (1, 7, 11):  
    print('move failed : ' + k + ' to ' + str(destination))  
    return False  
    if pos == 7:  
    if destination not in (2, 6, 12):  
    print('move failed : ' + k + ' to ' + str(destination))  
    return False  
    if pos == 8:  
    if destination not in (1, 3):  
    print('move failed : ' + k + ' to ' + str(destination))  
    return False  
    if pos == 9:  
    if destination not in (2, 4, 10):  
    print('move failed : ' + k + ' to ' + str(destination))  
    return False  
    if pos == 10:  
    if destination not in (5, 9):  
    print('move failed : ' + k + ' to ' + str(destination))  
    return False  
    if pos == 11:  
    if destination not in (4, 6):  
    print('move failed : ' + k + ' to ' + str(destination))  
    return False  
    if pos == 12:  
    if destination not in (5, 7):  
    print('move failed : ' + k + ' to ' + str(destination))  
    return False  
     
    if self.board[destination][1] == 'null':  
    self.board[pos] = (pos, 'null')  
    self.board[destination] = (destination, k)  
    print('knight ' + k + ' was moved to ' + str(destination))  
    self.numberofsteps += 1  
    return True  
    else:  
    print('move failed (not empty) : ' + k + ' to ' + str(destination))  
    return False  
     
    def divide1(self):  
    self.move('w2', 6)  
    anticlk = [9, 4, 11, 6, 7, 2]  
    for i in anticlk:  
    if self.position('b2') == 11:  
    break  
    self.move('b2', i)  
    for i in anticlk:  
    if self.position('w2') == 2:  
    break  
    else:  
    self.move('w2', i)  
      
    print(self.board)  
     
    def divide2(self):  
    anticlk = [1, 6, 7, 12, 5, 10, 9, 4, 3, 8, 1, 6]  
    for i in anticlk:  
    if self.position('b1') == 7:  
    break  
    self.move('b1', i)  
    for i in anticlk:  
    if self.position('b3') == 6:  
    break  
    self.move('b3', i)  
    for i in anticlk:  
    if self.position('w1') == 1:  
    break  
    self.move('w1', i)  
    for i in anticlk:  
    if self.position('w3') == 3:  
    break  
    self.move('w3', i)  
    for i in anticlk:  
    if self.position('b1') == 10:  
    break  
    self.move('b1', i)  
    for i in anticlk:  
    if self.position('b3') == 12:  
    break  
    self.move('b3', i)  
    print(self.board)  
     
      
     
   brd = [(0, 'null'), (1, 'b1'), (2, 'b2'), (3, 'b3'), (4, 'null'), (5, 'null'), (6, 'null'), (7, 'null'), (8, 'null'),  
    (9, 'null'), (10, 'w1'), (11, 'w2'), (12, 'w3')]  
   chess = task3(brd)  
   print(chess.board)  
   chess.divide1()  
   chess.divide2()  
   print(chess.numberofsteps)

#### Solution description

Firstly, we index our 4x3 chess board numerically for better representation :

Table, calendar

Description automatically generated with 3 black knights at postitons 1,2,3 and 3 white knights at positions 10,11,12

Then we draw an adjacency graph, which is the key for the solution, this undirected graph represents all the possible movements from each positions according to the correct movement of a knight in chess (ie. 2 squares in 1 directions,1 square in another , or L-shaped movements)

Diagram

Description automatically generatedDiagram

Description automatically generatedthis leaves us with the following adjacency graph , which concludes that no there is only one way to divide our problem into subproblems, by taking the 2 middle knights in the inner graph loop and swap them , then do the same with the outer graph loop and the 4 corner knights

Diagram, schematic

Description automatically generated

#### Complexity analysis

-The time complexity of the position function is

P(n) = n while n is the number of squares in our board (12)

-The time complexity of the move function is

N the time complexity of the position function

1\*11 +2 the time complexity of the 12 of the guard conditions that ensure movement follows adjacency graph

5 the time complexity for successfully moving the knight

M(n) = n+ 18

-The time complexity of the divide1 function is:

The first call of the function move: n+18

k Is the number of elements in the anticlk list (6)

This has total complexity of 2n+43

-The time complexity of the divide2 function is:

The loops have time complexity:

k Is the number of elements in the anticlk list (12)

Total time complexity is: 6\*(31+n) = 6n + 186

The total time complexity of the algorithm is 6n + 186 + 2n+43 = 8n + 229

#### Comparison between another algorithm

Another algorithm is the backtracking algorithm which searches for the optimal solution (16 steps) which is 14 steps faster than the divide and conquer code, it pseudo code:

Initialize board = { (1='B1'), (2= 'B2'), (3= 'B3'), (4 = ' '), (5 = ' '), (6 = ' '), (7 = ' '), (8 = ' '), (9 = ' '), (10='W1'), (11='W2'), (12='W3')}

Initialize solution board = { (1='W1'), (2= 'W2'), (3= 'W3'), (4 = ' '), (5 = ' '), (6 = ' '), (7 = ' '), (8 = ' '), (9 = ' '), (10='B1'), (11='B2'), (12='B3')}

initialize solution vector

If solution positions are reached print the chessboard Else

check if the next movement is valid (according to adjacency graph and destination is empty)

Add the next move to the solution vector and recursively check if this move would lead to a proper correct solution.

if the move chosen doesn't lead to a solution, then remove this move from the solution vector and try one of the other connected nodes,

If the alternative moves don't work, return false. This will remove the previously added node in recursion. If false is returned by the very first call of recursion,

then "no solution exists".

The python code of backtracking:

(note that the code found for backtracking solves for n\*n chessboard meaning only 3\*3 or 4\*4)

# Python3 program to solve Knight Tour problem using Backtracking

# Chessboard Size

n = 3

def isSafe(x, y, board):

'''

A utility function to check if i,j are valid indexes

for N\*N chessboard

'''

if(x >= 0 and y >= 0 and x < n and y < n and board[x][y] == -1):

return True

return False

def printSolution(n, board):

'''

A utility function to print Chessboard matrix

'''

for i in range(n):

for j in range(n):

print(board[i][j], end=' ')

print()

def solveKT(n):

'''

This function solves the Knight Tour problem using

Backtracking. This function mainly uses solveKTUtil()

to solve the problem. It returns false if no complete

tour is possible, otherwise return true and prints the

tour.

Please note that there may be more than one solutions,

this function prints one of the feasible solutions.

'''

# Initialization of Board matrix

board = [[-1 for i in range(n)]for i in range(n)]

# move\_x and move\_y define next move of Knight.

# move\_x is for next value of x coordinate

# move\_y is for next value of y coordinate

move\_x = [2, 1, -1, -2, -2, -1, 1, 2]

move\_y = [1, 2, 2, 1, -1, -2, -2, -1]

# Since the Knight is initially at the first block

board[0][0] = 0

# Step counter for knight's position

pos = 1

# Checking if solution exists or not

if(not solveKTUtil(n, board, 0, 0, move\_x, move\_y, pos)):

print("Solution does not exist")

else:

printSolution(n, board)

def solveKTUtil(n, board, curr\_x, curr\_y, move\_x, move\_y, pos):

'''

A recursive utility function to solve Knight Tour

problem

'''

if(pos == n\*\*2):

return True

# Try all next moves from the current coordinate x, y

for i in range(3):

new\_x = curr\_x + move\_x[i]

new\_y = curr\_y + move\_y[i]

if(isSafe(new\_x, new\_y, board)):

board[new\_x][new\_y] = pos

if(solveKTUtil(n, board, new\_x, new\_y, move\_x, move\_y, pos+1)):

return True

# Backtracking

board[new\_x][new\_y] = -1

return False

#### 1.3.2.6 Sample of the output

A computer screen capture

Description automatically generated with medium confidence

A computer screen capture

Description automatically generated with medium confidence

A computer screen capture

Description automatically generated with medium confidence

Figure 10: Output for the Code

## Task 4

### Problem description

Graphical user interface, text

Description automatically generated

### Solution

#### Pseudocode

Donecheck ( Boxes list)

Doneflag = false

For i=0 to list’s size do

If ( Boxes[i] > 1 )

Doneflag = true

Break

return Doneflag

Maintask(Boxes list)

Index 🡨 0

Loop 🡨 true

While loop== true do

If( Boxes[index] > 1)

If ( Boxes[index] %2 ==0 )

Increase boxes list size by 1

Boxes[index+1] = Boxes[index] / 2

Boxes[index] = 0

Increment index by 1

Loop = Donecheck(Boxes list)

Continue

If ( Boxes[index] %2 != 0 )

Increase boxes list size by 1

Boxes[index+1] = Boxes[index] / 2

Boxes[index] = 1

Increment index by 1

Loop= Donecheck(Boxes list)

Continue

#### Code

def donecheck(boxes):  
 doneFlag= False  
 for x in boxes:  
 if x > 1 :  
 doneFlag=True  
 break  
 return doneFlag  
  
  
def maintask(boxes:list):  
 index=0  
 loop=True  
 while (loop == True):  
 if (boxes[index]>1):  
 if (boxes[index]%2==0 ):  
 boxes.append(0)  
 boxes[index+1]=boxes[index]//2  
 boxes[index]=0  
 index=index+1  
 loop = donecheck(boxes)  
 print(boxes)  
 continue  
 elif (boxes[index]%2 != 0 ):  
 boxes.append(0)  
 boxes[index+1]=boxes[index]//2  
 boxes[index]=1  
 index=index+1  
 loop = donecheck(boxes)  
 print(boxes)  
 continue  
  
  
boxes = []  
print("Enter Number of Pennies:")  
n = input()  
boxes.append(int(n))  
print(boxes)  
maintask(boxes)

#### Solution description

The problem solution is to combine 2 pennies into 1 penny and transfer it to the box on the right

1 transfer at a time, this will cost too many steps. So the greedy solution to this problem is to reduce the number of steps by doing the following steps:

1- Start with one box with all n pennies in it and the while condition is true ( loop == True)

2- Check the number of pennies n in this box[i] if it even or odd

3- If even divide the number of pennies by 2 , add another box[i+1] to the right, add the n/2 pennies to the new box[i+1] and add the value 0 to that box[i] ( even numbers contain n/2 couple of pennies)

4- If odd divide the number of pennies by 2 , add another box[i+1] to the right, add the n/2 pennies to the new box[i+1] and add the value 1 to that box[i] ( odd numbers contain n/2 couple of pennies and 1 extra penny uncoupled)

5- Increment the indexes of boxes to start the while loop from that box ( index = index+1]

6- Before starting the next iteration , check whether or not all boxes have values greater than 1 ( by calling the function "donecheck(boxes)",if any of the boxes have values equal 0 or 1, doneFlag is set to be to be false making the while condition false ( loop=donecheck(boxes) ) to stop iterating and by this way we reached to the end of the solution

7- If values in any of the boxes have a value greater than 1 iterate again with the last recent value of index until donecheck returns false

#### Complexity analysis

O(log n)

#### Comparison between another algorithm

*Optimal Algorithm 1: ”output is readable from right to left”*

*Pseudocode:*

*i = 5 bits*

*While( i > 0)*

*If ( ( n % i ) not equal 0 )*

*Print ( 1)*

*Else*

*Print(0)*

*i = i/2*

*Code:*

i = 1 << 5  
 while (i > 0):  
  
 if ((n & i) != 0):  
  
 print("1", end=" ")  
  
 else:  
 print("0", end=" ")  
  
 i = i // 2  
  
  
bin(3)  
print()

*Optimal Code*

*A screenshot of a computer

Description automatically generated*

*A screenshot of a computer

Description automatically generatedOur code:*

*A screenshot of a computer

Description automatically generatedOptimal Code:*

*A screenshot of a computer

Description automatically generatedOur Code:*

*Optimal Algorithm 2:*

*Pseudo code:*

*Bin(n)*

*If n>1*

*Bin( n / 2 )*

*Print ( n % 2 )*

*Code:*

def bin(n):

    if n > 1:

        bin(n // 2)

    print(n % 2, end=" ")

# Driver Code

if \_name\_ == "\_main\_":

    bin(8) #trying any number

*Optimal Code**”output is readable from right to left”*

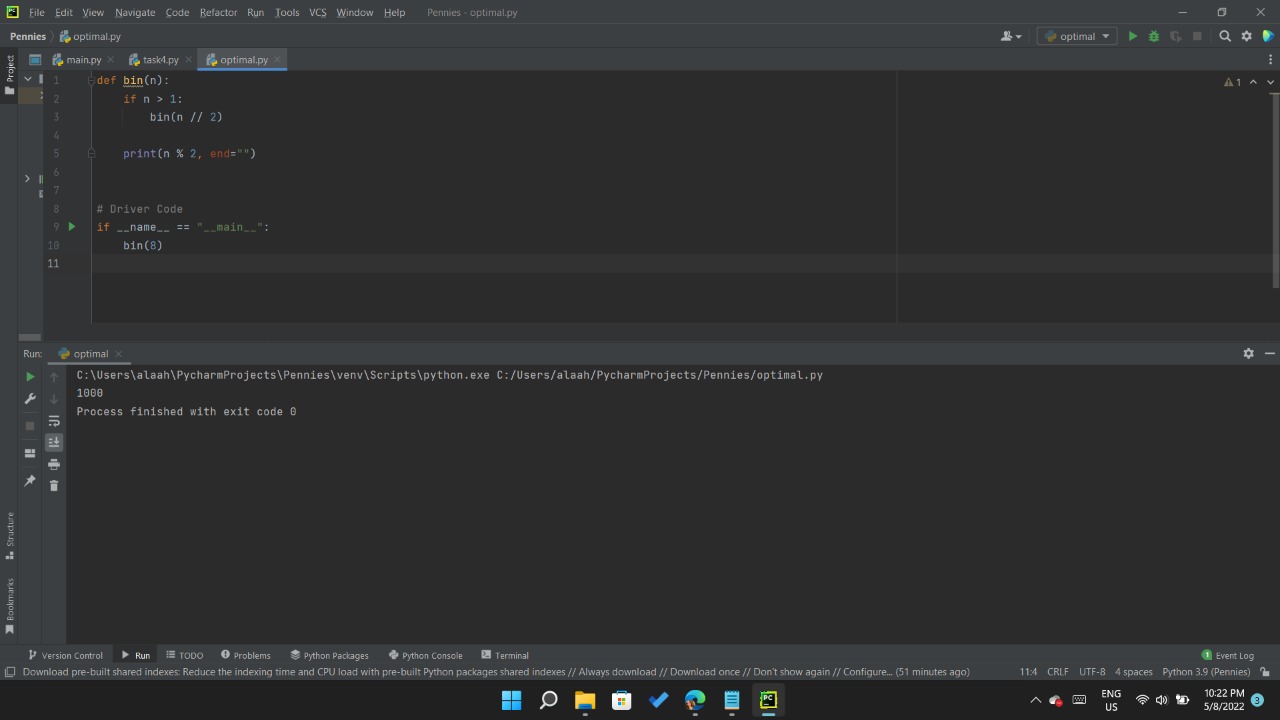
**

Figure 11: Output for number (8)

#### 1.4.2.6 Sample of the output

*Our code: “output is readable from left to right”*

**

Figure 12: Output for number (8)

*Optimal Code:*

*Our Code:*

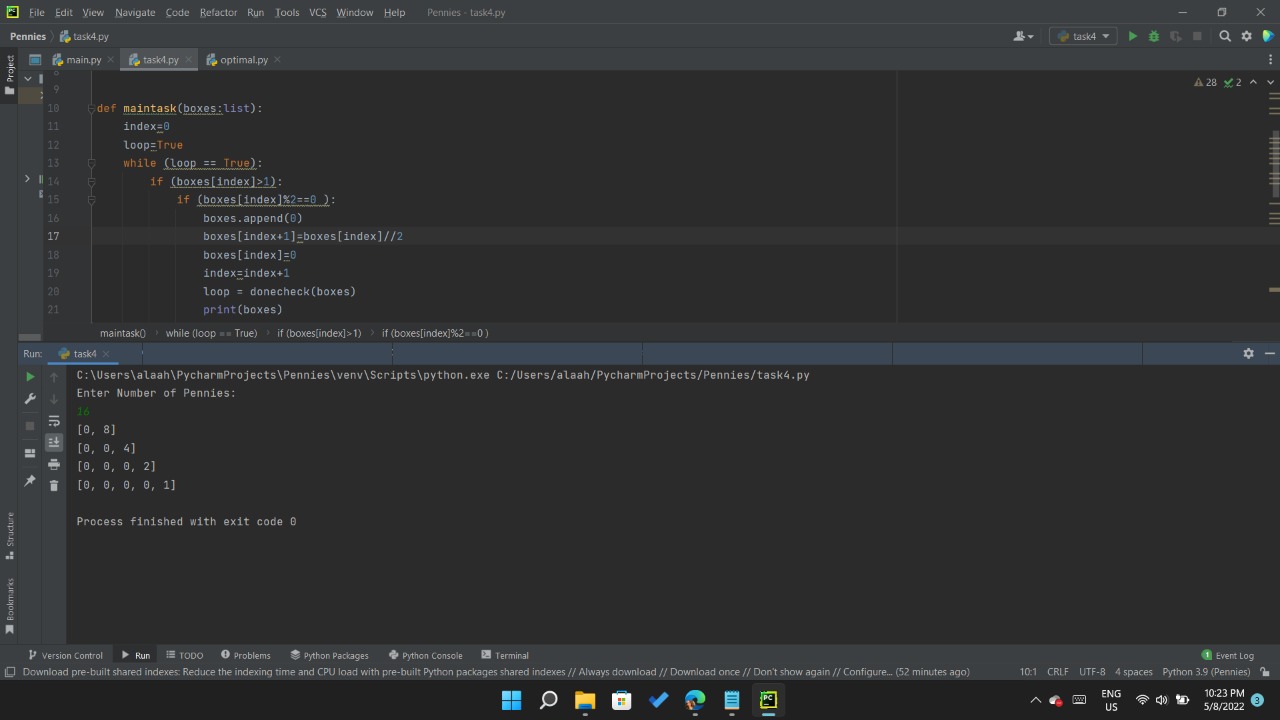
**

Figure 13: Output for number (16) using optimal code

*A screenshot of a computer

Description automatically generated*

Figure 14: Output for number (16) using greedy algorithm

Sample output of the solution for the different cases of the algorithms:

A screenshot of a computer

Description automatically generated

Figure 15: Output for number (18)

A screenshot of a computer

Description automatically generated

Figure 16: Output for number (512)

A screenshot of a computer

Description automatically generated

Figure 17: Output for number (1024)

A screenshot of a computer

Description automatically generated

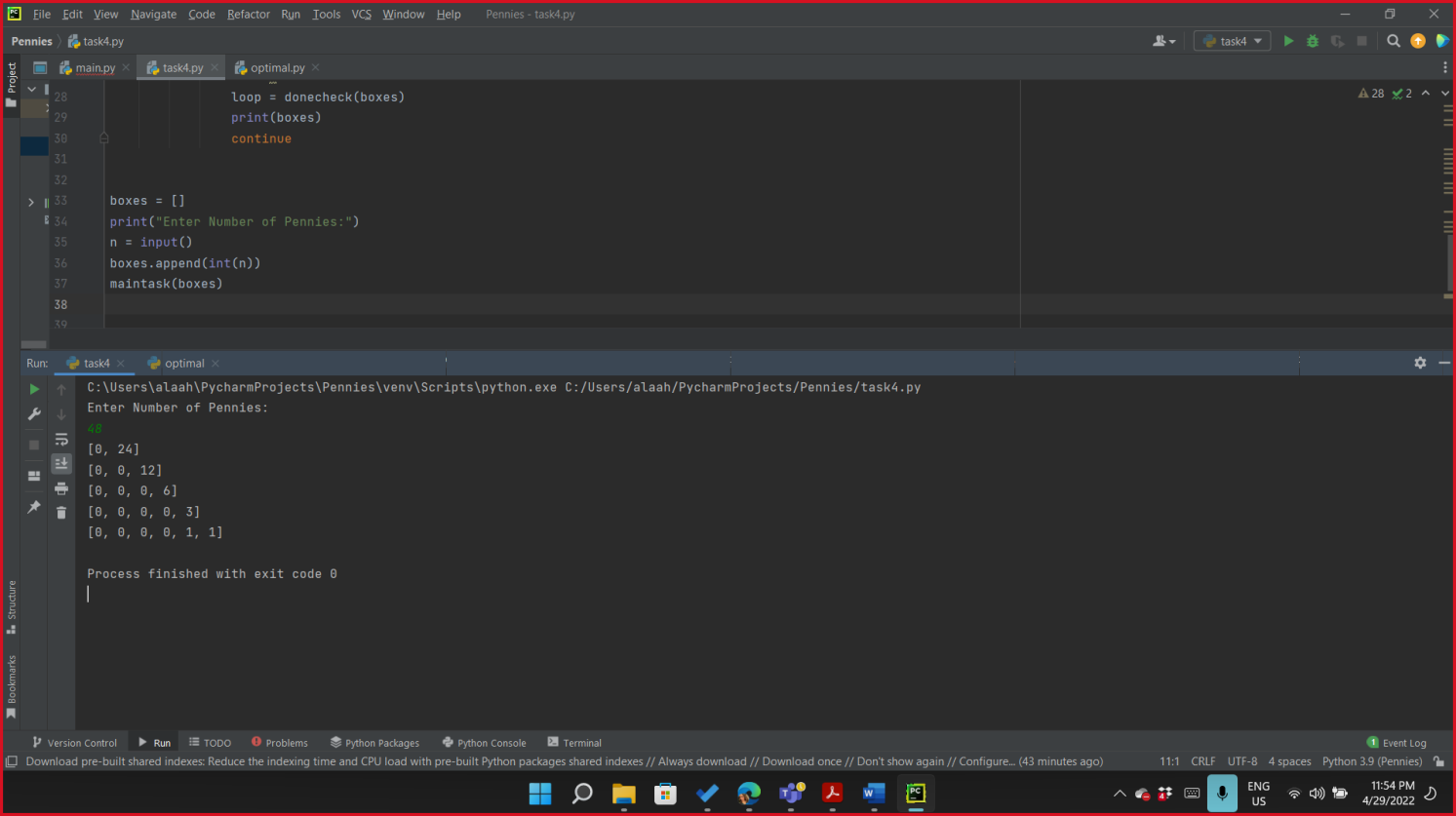


Figure 18: Output for number (64)

Figure 19: Output for number (48)

Task 4 part A:

Does the final distribution of pennies depend on the order in which the machine processes the coin pairs?

-The algorithm will run the same way but the binary representation of the number will be read from right to left instead from left to right.

Task 4 part B:

What is the minimum number of boxes needed to distribute n pennies?

A screenshot of a computer

Description automatically generated-for n>=2 minimum number of boxes needed is 2 because in the first iteration you will take 2 pennies

Figure 20: Output for number (2)

Task 4 part C:

How many iterations does the machine make before stopping?

if we start with 17 pennies in the leftmost box the final distribution of pennies will be

1st iteration 15 , 2nd iteration 13, 3rd iteration 11, and so on

-Log n iterations

## Task 5

### Problem description

Graphical user interface, text

Description automatically generated

### Solution

#### Pseudocode

Create new Array Switches

OFF ( start , end)

N = end – start +1

If ( N ==1)

Toggle(end)

Else if ( N== 2)

Toggle ( start)

Toggle ( end )

Else

OFF( start +2 , end )

Toggle ( start )

ON ( start + 2, end)

OFF ( start + 1 , end)

ON ( start , end)

N = end – start +1

IF( N == 1)

Toggle(end)

Else if ( N == 2)

Toggle(end)

Toggle(start)

Else

ON ( start + 1, end)

OFF ( start + 2, end)

Toggle(start)

ON (start +2, end)

Toggle (i)

If ( switches[ i ] == 1)

switches [ i ] = 0

Else

Switches[ i ] = 1

Print Switches array

#### Code

switches = []

def Off(start, end):

    n = end - start + 1

    if n == 1:

        toggle(end)

    elif n == 2:

        toggle(start)

        toggle(end)

    else:

        Off(start + 2, end)

        toggle(start)

On(start + 2, end)

Off(start + 1 , end)

def On(start, end):

    n = end - start + 1

    if n == 1:

        toggle(end)

    elif n == 2:

        toggle(end)

        toggle(start)

    else:

        On(start + 1, end)

        Off(start + 2, end)

        toggle(start)

        On(start +2, end)

def toggle(i):

    if switches[i] == 1:

        switches[i] = 0

    else:

        switches[i] = 1

    print(switches)

def main():

    global switches

    n = int(input("Enter number of switches: "))

    switches = [1] \* n

    print(switches)

    Off(0, n - 1)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

#### Solution description

This decrease and conquer approach manages to make this problem easier to solve, where the user will input n which is the number of switches to turn off and output is every step taken towards the solution. Once n is entered function Off() is called to start executing;

* If n is either 1 or 2, it will be handled as base case in which toggling is known and standard in all cases
* If n > 2 then it will enter a state where a few steps are repeated until solved.
* Off function executes as follows
  + Turn off starting from index (n+2)
  + Then, toggle index n
  + Afterwards turn on switches starting from index (n+2)
  + Finally, turn off the switches from index (n+1)
* On function executes as follows
  + Turn On starting from index (n+1)
  + Afterwards turn off switches starting from index (n+1)
  + Then, toggle index n
  + Finally, turn on the switches from index (n+2)

#### Complexity analysis

First algorithm time complexity

1. Recurrence relationship

After simplification

After simplification

After simplification

Solving the recurrence relationship

for n>=1 O(2n)

#### Comparison between another algorithm

Second Algorithm Pseudocode

SwitchOff(int n)

{

PatternCount = [-1] \* n

If n -> even

PatternCount[n-1] = 0

PatternCount[n-2] = 3

Else

PatternCount[n-1] = 1

PatternCount[n-2] = 0

While (!Solved(switches))

For i = 0 -> n

If PatternCount[i] = -1

If switches[i+1] = 1 and switches[i+2….n-1] =0

Toggle switches[i]

Print(switches)

Else

}

Boolean Solved(int [] array)

{

Bool solved = true

For i = 0 in switches

If switches[i] = 1

Solved = false

Return solved

}

Second algorithm complexity

O(n2)

#### 1.5.2.6 Sample of the output

A screenshot of a computer

Description automatically generated with medium confidence

Figure 21: Output for number (4)

A computer screen capture

Description automatically generated with medium confidence

Figure 22: Output for number (3)

A computer screen capture

Description automatically generated with medium confidence

A picture containing text, electronics, screenshot, computer

Description automatically generated

Figure 23: Output for number (6)

## 1.6 Task 6

### 1.6.1 Problem description

Text

Description automatically generated

### 1.6.2 Solution

#### 1.6.2.1 Pseudocode

//call the FrameStewartSolution function giving it atleast number of disks on initial peg

@fsmemoizer //decorator for memoizer

FrameStewartSolution (ndisks , , start=1, end=4, pegs=set([1, 2, 3, 4]))

{

If ndisks == zero then return nothing

If ndisks == 1 and number of pegs >1 {move that disk to the end peg}

If number of pegs == 3

{

call the towers3 function which solves the normal towers of Hanoi of 3 pegs

}

If number of pegs >= 3 and ndisks > 0 :

{

For each K:

Find helper peg

//3 Recursive calls to FrameStwewartSolution

LHSmoves = FrameStewartSolution(kdisks, start, helper\_peg, pegs)

Mymoves = FrameStewartSolution(ndisks - kdisks, start, end, pegs\_for\_my\_moves)

RHSmoves = FrameStewartSolution(kdisks, helper\_peg, end, pegs)

Check if any of the 3 moves returned NONE that means it was is bad path so this K is ignored

movelist = LHSMoves + MyMoves + RHSMoves

if moveslist is shorter than Bestscore

{

Bestsolution = movelist

Bestscore = Length of movelist

Return BestSolution

}

If there was no solution{ return NONE}

} // end of FrameStewartSolution

//fsmemoizer function

fsmemoizer(f)

{

//initialize dictionary to as a memory

Cx{}

f2(\*args)

{

//compute the key of dictionary entry by taking the values of the argument sent to the function as keyvalue

key= json.dumps(args)

if arguments found in dictionary{ return value}

else {store it in the Cx for later use}

}

} // end of memoizer

Towers3 (ndisks ,start=1 ,target=3 ,peg\_set=set([1 ,2 ,3]))

{

If ndisks == 0 or start == end {return no moves}

Mymove = move first disk form initial to final peg

If ndisks == 1 {return mymove only}

Determine helperpeg

// 2 recursive calls to Towers3

Movestomymoves = towers3(ndisks -1 ,start ,helper\_peg,peg\_set)

Movesaftermymoves = towers3(ndisks -1 ,helper\_peg ,target,peg\_set)

Return sum of 3 moves made

} //end of Towers3 function

#### 1.6.2.2 Code

import json  
  
  
# tower3 solves normal towers of hanoi of 3 pegs  
def towers3(ndisks ,start=1 ,target=3 ,peg\_set=set([1 ,2 ,3])):  
 if ndisks == 0 or start == target: # if there are no disks, or no move to make  
 return [] # no moves  
 my\_move = "move(%s,%s) " %(start ,target)  
 if ndisks == 1: # trivial case if there is only one disk, just move it  
 return [my\_move]  
 helper\_peg = peg\_set.difference([start ,target]).pop()  
 moves\_to\_my\_move = towers3(ndisks -1 ,start ,helper\_peg,peg\_set)  
 moves\_after\_my\_move = towers3(ndisks -1 ,helper\_peg ,target,peg\_set)  
 return moves\_to\_my\_move + [my\_move] + moves\_after\_my\_move  
  
# memoizer used to store solutions to be used later , this is the dynamic programming enhancement  
# if you wish to see its importance comment out the memoizer function and the decorator then call the  
# framestewartsolution with 16 disks ( it will take almost 60 seconds) but with memoizer almost instantly under 1 second  
def fsMemoizer(f): # just a junky quick memoizer  
 cx = {}  
 def f2(\*args):  
 try:  
 key= json.dumps(args)  
 except:  
 key =json.dumps(args[:-1] + (sorted(list(args[-1])),))  
 if key not in cx:  
 cx[key] = f(\*args)  
 return cx.get(key)  
  
 return f2  
  
  
@fsMemoizer  
def FrameStewartSolution(ndisks, start=1, end=4, pegs=set([1, 2, 3, 4])):  
 if ndisks == 0 or start == end: # zero disks require zero moves  
 return []  
 if ndisks == 1 and len(pegs) > 1: # if there is only 1 disk it will only take one move  
 return ["move(%s,%s)" % (start, end)]  
 if len(pegs) == 3: # 3 pegs is well defined optimal solution of 2^n-1  
 return towers3(ndisks, start, end, pegs)  
 if len(pegs) >= 3 and ndisks > 0:  
 best\_solution = float("inf")  
 best\_score = float("inf")  
 for kdisks in range(1, ndisks):  
 helper\_pegs = list(pegs.difference([start, end]))  
 LHSMoves = FrameStewartSolution(kdisks, start, helper\_pegs[0], pegs)  
 pegs\_for\_my\_moves = pegs.difference([helper\_pegs[0]])# cant use the peg our LHS stack is sitting on  
 MyMoves = FrameStewartSolution(ndisks - kdisks, start, end, pegs\_for\_my\_moves) # misleading variable name but meh  
 RHSMoves = FrameStewartSolution(kdisks, helper\_pegs[0], end, pegs) # move the intermediat stack to  
 if any(move is None for move in [LHSMoves, MyMoves, RHSMoves]): continue # bad path :(  
 move\_list = LHSMoves + MyMoves + RHSMoves  
 if (len(move\_list) < best\_score):  
 best\_solution = move\_list  
 best\_score = len(move\_list)  
 if best\_score < float("inf"):  
 return best\_solution  
 # all other cases where there is no solution (namely one peg, or 2 pegs and more than 1 disk)  
 return None  
  
  
  
  
  
if \_\_name\_\_ == '\_\_main\_\_':

# this will show a list of 33 moves performed  
 print(FrameStewartSolution(8))

#### 1.6.2.3 Solution description

* Since the problem is an obvious extension of the Tower of Hanoi puzzle , it is natural to use a similar recursive approach.
* Namely, if ndisks > 2, transfer k smallest disks to an intermediate peg recursively using help of all four pegs (LHSmove), then
* move the remaining n − k disks to the destination peg by the classic recursive algorithm for the three-peg Tower of Hanoi puzzle (MYmove)
* finally , transfer the k smallest disks to the destination peg recursively using all four pegs.(RHSmove)
* If n = 1 or 2, solve the trivial instances of the problem in one and three moves , respectively, as it is done in the three-peg Tower of Hanoi solution.
* The value of parameter k must be selected to minimize the total number of disk moves made by the algorithm. That why we try all possible values of K then choose the K which results in the minimum number of total moves
* For dynamic programming enhancement we use a simple memoizer which stores the the parameters and their return values of a call to the main function so when this expensive function is called again with same parameter instead of calculating all over again the memoizer return the stored value of the return value of such prameters.
* The effect of dynamic programming time enhancements can be noticed mostly if the same function was called for 16 initial disks and comment out the memoizer and then using a memorize , the effect is huge . with a memoizer it takes less than a second while without it , it takes almost 60 seconds!!

#### 1.6.2.4 Complexity analysis

Recurrence relation for the number of

moves, *R*(*n*)*,* made by this algorithm to move *n* disks:

*R*(*n*) = [2*R*(*k*) + − 1] for *n >* 2*, R*(1) = 1*, R*(2) = 3*.*

* − 1 is the recurrence relation for Towers of Hanoi with 3 pegs
* 2 R(k) is the steps taken to move K disks initially to a certain helper peg (LHSmove) then in then end to move the same K disks the target peg (RHSmove)
* since we try all K values then choose the K which produces the least number of moves

Towers of Hanoi recurrence analysis:

Text, letter

Description automatically generated

Asymptotic notation for Towers of Hanoi is − 1 so Ѳ()

IN OUR CASE :

The initial number of disks is 8 so

The code tries all K values and each try is Ѳ() so this search’s asymptotic notation is Ѳ() as well

We can use this algorithm to deduce K for our case :



K is calculated to be 4 so :

R(n) = 2 *R*(*4*) + − 1 for *n >* 2*, R*(1) = 1*, R*(2) = 3*.*

R(4) = 2 R(1) + – 1 = 9

R(n) = 18 + − 1

R(n) = + 17

R(n) = (+ 273)

So the asymptotic notation of the code is Ѳ()

**Of course this time is drastically reduced with the help of the dynamic programming enhancements**

#### 1.6.2.5 Comparison between another algorithm

Another algorithm we can use to solve this problem uses a decrease and conquer algorithm

Code:

# slower solution using divide and qonquer  
def slowersolution(disk, source, temppeg1, temppeg2, destination):  
 if disk == 1:  
 print((source, destination))  
 elif disk == 2:  
 print((source, temppeg1))  
 print((source, destination))  
 print((temppeg1, destination))  
 else:  
 slowersolution(disk - 2, source, temppeg2, destination, temppeg1)  
 print((source, temppeg2))  
 print((source, destination))  
 print((temppeg2, destination))  
 slowersolution(disk - 2, temppeg1, source, temppeg2, destination)

This algorithm takes 45 moves unlike the dynamic programming algorithm we made which takes 33 moves only to solve so it makes it the next best solution (suboptimal).

#### 1.6.2.6 Sample of the output

Main:

Graphical user interface, application

Description automatically generated

Figure 24: main to run the Output

OUR CODE (optimal):

Graphical user interface, text, application

Description automatically generated

Figure 25: Output for number (8) using dynamic programming

Text

Description automatically generatedSlower code (suboptimal):

Figure 26: Output for number (8) using decrease and conquer

# REFRENCES

[1] Levitin, A. and Levitin, M., 2011. ALGORITHMIC PUZZLES. Oxford University Press; Illustrated edition, p.280.

[2] Levitin, A., 2006. Introduction to the Design and Analysis of Algorithms. 2nd ed. Addison Wesley, p.592.