```
#week1 --BFS
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph=defaultdict(list)
    def addEdge(self,u,v):
        self.graph[u].append(v)
    def BFS(self,s):
        visited=[False] * (len(self.graph))
        queue=[]
        queue.append(s)
        visited[s]=True
        while queue:
            s= queue.pop(0)
            print(s,end=" ")
            for i in self. graph[s]:
                if visited[i]==False:
                    queue.append(i)
                    visited[i]=True
g=Graph()
g.addEdge(0,1)
g.addEdge(0,2)
g.addEdge(1,2)
g.addEdge(2,0)
g.addEdge(2,3)
g.addEdge(3,3)
print("Following BFS traversal" "(starting from vertex 2)")
g.BFS(2)

--------------------------------------------------------------------------
-------------------------------------------------------------------------

#WEEK2 --DFS
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph=defaultdict(list)
    def addEdge(self,u,v):
        self.graph[u].append(v)
    def DFSUtil(self,v,visited):
        visited.add(v)
        print(v,end=' ')
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour,visited)
    def DFS(self,v):
        visited=set()
        self.DFSUtil(v,visited)
g=Graph()
g.addEdge(0,1)
g.addEdge(0,2)
g.addEdge(1,2)
g.addEdge(2,0)
g.addEdge(2,3)
```

```
g.addEdge(3,3)
print("Following is DFS from (starting from vertex 2)")
g.DFS(2)
```

-------------------------------------------------------------------------
-------------------------------------------------------------------------

```
#WEEK 3 --A* SEARCH


def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}               #store distance from starting node
    parents = {}         # parents contains an adjacency map of all nodes
    #distance of starting node from itself is zero
    g[start_node] = 0
    #start_node is root node i.e it has no parent nodes
    #so start_node is set to its own parent node
    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None
        #node with lowest f() is found
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                #nodes 'm' not in first and last set are added to first
                #n is set its parent
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                #for each node m,compare its distance from start i.e g(m)
to the
                #from start through n node
                else:
                    if g[m] > g[n] + weight:
                        #update g(m)
                        g[m] = g[n] + weight
                        #change parent of m to n
                        parents[m] = n
                        #if m in closed set,remove and add to open
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None

        # if the current node is the stop_node
```

```
        # then we begin reconstructin the path from it to the start_node
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
            return path
        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_set.remove(n)
        closed_set.add(n)
    print('Path does not exist!')
    return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

#for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J': 0
    }
    return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
}
```

```
aStarAlgo('A', 'J')
```

--------------------------------------------------------------------------
--------------------------------------------------------------------------
-

```
#WEEK 4 --TRAVELLING SALESPERSON PROBLEM

from sys import maxsize
from itertools import permutations
v=4
def travellingsalesmanproblem(graph,s):
    vertex=[]
    for i in range(v):
        if i!=s:
            vertex.append(i)
    min_path=maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:
        current_pathweight=0
        k=s
        for j in i:
            current_pathweight+=graph[k][j]
            k=j
        current_pathweight+=graph[k][s]
        min_path=min(min_path,current_pathweight)
    return min_path
if __name__ == "__main__":
    graph=[[0,10,15,20],[10,0,35,25],[15,35,0,30],[20,25,30,0]]
    s=0
    print(travellingsalesmanproblem(graph,s))
```

--------------------------------------------------------------------------
--------------------------------------------------------------------------
--

```
#WEEK 5 --GRAPH COLOURING


colors=['Red','Blue','Green','Yellow','Black']
states=['Andhra','Karnataka','Tamilnadu','Kerala']
neighbors={}
neighbors['Andhra']=['Karnataka','Tamilnadu']
neighbors['Karnataka']=['Andhra','Tamilnadu','Kerala']
neighbors['Tamilnadu']=['Andhra','Karnataka','Kerala']
neighbors['Kerala']=['Karnataka','Tamilnadu',]
colors_of_states={}
def promising(state,color):
    for neighbor in neighbors.get(state):
     color_of_neighbor=colors_of_states.get(neighbor)
    if color_of_neighbor==color:
        return False
```

```
    return True
def get_color_for_state(state):
  for color in colors:
   if promising(state,color):
       return color
def main():
 for state in states:
  colors_of_states[state]=get_color_for_state(state)
  print(colors_of_states)
main()
```

---------------------------------------------------------------------------
--------------------------------------------------------------------------

```
#WEEK 7 --WATER JUG PROBLEM


from collections import defaultdict

# jug1 and jug2 contain the value
# for max capacity in respective jugs
# and aim is the amount of water to be measured.
jug1, jug2, aim = 4, 3, 2

# Initialize dictionary with
# default value as false.
visited = defaultdict(lambda: False)

# Recursive function which prints the
# intermediate steps to reach the final
# solution and return boolean value
# (True if solution is possible, otherwise False).
# amt1 and amt2 are the amount of water present
# in both jugs at a certain point of time.
def waterJugSolver(amt1, amt2):

      # Checks for our goal and
      # returns true if achieved.
      if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
            print(amt1, amt2)
            return True

      # Checks if we have already visited the
      # combination or not. If not, then it proceeds further.
      if visited[(amt1, amt2)] == False:
            print(amt1, amt2)

            # Changes the boolean value of
            # the combination as it is visited.
            visited[(amt1, amt2)] = True

            # Check for all the 6 possibilities and
            # see if a solution is found in any one of them.
            return (waterJugSolver(0, amt2) or
```

```
                        waterJugSolver(amt1, 0) or
                        waterJugSolver(jug1, amt2) or
                        waterJugSolver(amt1, jug2) or
                        waterJugSolver(amt1 + min(amt2, (jug1-amt1)),
                        amt2 - min(amt2, (jug1-amt1))) or
                        waterJugSolver(amt1 - min(amt1, (jug2-amt2)),
                        amt2 + min(amt1, (jug2-amt2))))

      # Return False if the combination is
      # already visited to avoid repetition otherwise
      # recursion will enter an infinite loop.
      else:
            return False

print("Steps: ")

# Call the function and pass the
# initial amount of water present in both jugs.
waterJugSolver(0, 0)
---------------------------------------------------------------------------
------------------------------------------------------------------------
#WEEK 6 -- MISSIOINARIES AND CANNINBALS PROBLEM


#Python program to illustrate Missionaries & cannibals Problem
#This code is contributed by Sunit Mal
print("\n")
print("\tGame Start\nNow the task is to move all of them to right side of
the river")
print("rules:\n1. The boat can carry at most two people\n2. If cannibals
num greater then missionaries then the cannibals would eat the
missionaries\n3. The boat cannot cross the river by itself with no people
on board")
lM = 3            #lM = Left side Missionaries number
lC = 3            #lC = Laft side Cannibals number
rM=0             #rM = Right side Missionaries number
rC=0             #rC = Right side cannibals number
userM = 0        #userM = User input for number of missionaries for right
to left side travel
userC = 0        #userC = User input for number of cannibals for right to
left travel
k = 0
print("\nM M M C C C |     --- | \n")
try:
    while(True):
        while(True):
            print("Left side -> right side river travel")
            #uM = user input for number of missionaries for left to right
travel
            #uC = user input for  number of cannibals for left to right
travel
            uM = int(input("Enter number of Missionaries travel => "))
            uC = int(input("Enter number of Cannibals travel => "))
```

```
            if((uM==0)and(uC==0)):
                print("Empty travel not possible")
                print("Re-enter : ")
            elif(((uM+uC) <= 2)and((lM-uM)>=0)and((lC-uC)>=0)):
                break
            else:
                print("Wrong input re-enter : ")
        lM = (lM-uM)
        lC = (lC-uC)
        rM += uM
        rC += uC

        print("\n")
        for i in range(0,lM):
            print("M ",end="")
        for i in range(0,lC):
            print("C ",end="")
        print("| --> | ",end="")
        for i in range(0,rM):
            print("M ",end="")
        for i in range(0,rC):
            print("C ",end="")
        print("\n")

        k +=1

        if(((lC==3)and (lM ==
1))or((lC==3)and(lM==2))or((lC==2)and(lM==1))or((rC==3)and (rM ==
1))or((rC==3)and(rM==2))or((rC==2)and(rM==1))):
            print("Cannibals eat missionaries:\nYou lost the game")

            break

        if((rM+rC) == 6):
            print("You won the game : \n\tCongrats")
            print("Total attempt")
            print(k)
            break
        while(True):
            print("Right side -> Left side river travel")
            userM = int(input("Enter number of Missionaries travel => "))
            userC = int(input("Enter number of Cannibals travel => "))

            if((userM==0)and(userC==0)):
                    print("Empty travel not possible")
                    print("Re-enter : ")
            elif(((userM+userC) <= 2)and((rM-userM)>=0)and((rC-
userC)>=0)):
                break
            else:
                print("Wrong input re-enter : ")
        lM += userM
        lC += userC
        rM -= userM
```

```
        rC -= userC

        k +=1
        print("\n")
        for i in range(0,lM):
            print("M ",end="")
        for i in range(0,lC):
            print("C ",end="")
        print("| <-- | ",end="")
        for i in range(0,rM):
            print("M ",end="")
        for i in range(0,rC):
            print("C ",end="")
        print("\n")



        if(((lC==3)and (lM ==
1))or((lC==3)and(lM==2))or((lC==2)and(lM==1))or((rC==3)and (rM ==
1))or((rC==3)and(rM==2))or((rC==2)and(rM==1))):
            print("Cannibals eat missionaries:\nYou lost the game")
            break
except EOFError as e:
    print("\nInvalid input please retry !!")
```

---------------------------------------------------------------------------
-------------------------------------------------------------------------

```
#WEEK 9 --TIC-TAC-TOE


import os
import time

board = [' ',' ',' ',' ',' ',' ',' ',' ',' ',' ']
player = 1

########win Flags##########
Win = 1
Draw = -1
Running = 0
Stop = 1
###########################
Game = Running
Mark = 'X'

#This Function Draws Game Board
def DrawBoard():
    print(" %c | %c | %c " % (board[1],board[2],board[3]))
    print("___|___|___")
    print(" %c | %c | %c " % (board[4],board[5],board[6]))
    print("___|___|___")
```

```
    print(" %c | %c | %c " % (board[7],board[8],board[9]))
    print("   |   |   ")

#This Function Checks position is empty or not
def CheckPosition(x):
    if(board[x] == ' '):
        return True
    else:
        return False

#This Function Checks player has won or not
def CheckWin():
    global Game
    #Horizontal winning condition
    if(board[1] == board[2] and board[2] == board[3] and board[1] != '
'):
        Game = Win
    elif(board[4] == board[5] and board[5] == board[6] and board[4] != '
'):
        Game = Win
    elif(board[7] == board[8] and board[8] == board[9] and board[7] != '
'):
        Game = Win
    #Vertical Winning Condition
    elif(board[1] == board[4] and board[4] == board[7] and board[1] != '
'):
        Game = Win
    elif(board[2] == board[5] and board[5] == board[8] and board[2] != '
'):
        Game = Win
    elif(board[3] == board[6] and board[6] == board[9] and board[3] != '
'):
        Game=Win
    #Diagonal Winning Condition
    elif(board[1] == board[5] and board[5] == board[9] and board[5] != '
'):
        Game = Win
    elif(board[3] == board[5] and board[5] == board[7] and board[5] != '
'):
        Game=Win
    #Match Tie or Draw Condition
    elif(board[1]!=' ' and board[2]!=' ' and board[3]!=' ' and
board[4]!=' ' and board[5]!=' ' and board[6]!=' ' and board[7]!=' ' and
board[8]!=' ' and board[9]!=' '):
        Game=Draw
    else:
        Game=Running

print("Tic-Tac-Toe Game Designed By Sourabh Somani")
print("Player 1 [X] --- Player 2 [O]\n")
print()
print()
print("Please Wait...")
time.sleep(3)
```

```
while(Game == Running):
    os.system('cls')
    DrawBoard()
    if(player % 2 != 0):
        print("Player 1's chance")
        Mark = 'X'
    else:
        print("Player 2's chance")
        Mark = 'O'
    choice = int(input("Enter the position between [1-9] where you want
to mark : "))
    if(CheckPosition(choice)):
        board[choice] = Mark
        player+=1
        CheckWin()

os.system('cls')
DrawBoard()
if(Game==Draw):
    print("Game Draw")
elif(Game==Win):
    player-=1
    if(player%2!=0):
        print("Player 1 Won")
    else:
        print("Player 2 Won")


----------------------------------------------------------------------------
----------------------------------------------------------------------------
-

#WEEK 8 --HANGMAN PROBLEM


import random
import time
# Initial Steps to invite in the game:
print("\nWelcome to Hangman game by IT SOURCECODE\n")
name = input("Enter your name: ")
print("Hello " + name + "! Best of Luck!")
time.sleep(2)
print("The game is about to start!\n Let's play Hangman!")
time.sleep(3)
# The parameters we require to execute the game:
def main():
    global count
    global display
    global word
    global already_guessed
    global length
    global play_game
```

```
    words_to_guess =
["january","border","image","film","promise","kids","lungs","doll","rhyme
","damage"
                   ,"plants"]
    word = random.choice(words_to_guess)
    length = len(word)
    count = 0
    display = '_' * length
    already_guessed = []
    play_game = ""

# A loop to re-execute the game when the first round ends:

def play_loop():
    global play_game
    play_game = input("Do You want to play again? y = yes, n = no \n")
    while play_game not in ["y", "n","Y","N"]:
        play_game = input("Do You want to play again? y = yes, n = no
\n")
    if play_game == "y":
        main()
    elif play_game == "n":
        print("Thanks For Playing! We expect you back again!")
        exit()

# Initializing all the conditions required for the game:
def hangman():
    global count
    global display
    global word
    global already_guessed
    global play_game
    limit = 5
    guess = input("This is the Hangman Word: " + display + " Enter your
guess: \n")
    guess = guess.strip()
    if len(guess.strip()) == 0 or len(guess.strip()) >= 2 or guess <=
"9":
        print("Invalid Input, Try a letter\n")
        hangman()


    elif guess in word:
        already_guessed.extend([guess])
        index = word.find(guess)
        word = word[:index] + "_" + word[index + 1:]
        display = display[:index] + guess + display[index + 1:]
        print(display + "\n")

    elif guess in already_guessed:
        print("Try another letter.\n")

    else:
        count += 1
```

```
        if count == 1:
            time.sleep(1)
            print("   _____ \n"
                  "  |      \n"
                  "  |      \n"
                  "  |      \n"
                  "  |      \n"
                  "  |      \n"
                  "  |      \n"
                  "__|__\n")
            print("Wrong guess. " + str(limit - count) + " guesses
remaining\n")

        elif count == 2:
            time.sleep(1)
            print("   _____ \n"
                  "  |     | \n"
                  "  |     |\n"
                  "  |      \n"
                  "  |      \n"
                  "  |      \n"
                  "  |      \n"
                  "__|__\n")
            print("Wrong guess. " + str(limit - count) + " guesses
remaining\n")

        elif count == 3:
           time.sleep(1)
           print("   _____ \n"
                 "  |     | \n"
                 "  |     |\n"
                 "  |     | \n"
                 "  |      \n"
                 "  |      \n"
                 "  |      \n"
                 "__|__\n")
           print("Wrong guess. " + str(limit - count) + " guesses
remaining\n")

        elif count == 4:
            time.sleep(1)
            print("   _____ \n"
                  "  |     | \n"
                  "  |     |\n"
                  "  |     | \n"
                  "  |     O \n"
                  "  |      \n"
                  "  |      \n"
                  "__|__\n")
            print("Wrong guess. " + str(limit - count) + " last guess
remaining\n")

        elif count == 5:
```

```
            time.sleep(1)
            print("   _____ \n"
                  "  |     | \n"
                  "  |     |\n"
                  "  |     | \n"
                  "  |     O \n"
                  "  |    /|\ \n"
                  "  |    / \ \n"
                  "__|__\n")
            print("Wrong guess. You are hanged!!!\n")
            print("The word was:",already_guessed,word)
            play_loop()

    if word == '_' * length:
        print("Congrats! You have guessed the word correctly!")
        play_loop()

    elif count != limit:
        hangman()


main()


hangman()
```

-------------------------------------------------------------------------
-------------------------------------------------------------------------
-

```
#WEEK 10 --N-QUEENS
#Number of queens
print ("Enter the number of queens")
N = int(input())

#chessboard
#NxN matrix with all elements 0
board = [[0]*N for _ in range(N)]

def is_attack(i, j):
    #checking if there is a queen in row or column
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonals
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False

def N_queen(n):
    #if n is 0, solution found
```

```
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            '''checking if we can place a queen here or not
            queen will not be placed if the place is being attacked
            or already occupied'''
            if (not(is_attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                #recursion
                #wether we can put the next queen with this arrangment or
not
                if N_queen(n-1)==True:
                    return True
                board[i][j] = 0

    return False

N_queen(N)
for i in board:
    print (i)
--------------------------------------------------------------------------
------------------------------------------------------------------------
#week -11 monty hall problem

import numpy
from pomegranate import *
guest = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})
prize = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})
monty = ConditionalProbabilityTable(
 [[ 'A', 'A', 'A', 0.0 ],
 [ 'A', 'A', 'B', 0.5 ],
 [ 'A', 'A', 'C', 0.5 ],
 [ 'A', 'B', 'A', 0.0 ],
 [ 'A', 'B', 'B', 0.0 ],
 [ 'A', 'B', 'C', 1.0 ],
 [ 'A', 'C', 'A', 0.0 ],
 [ 'A', 'C', 'B', 1.0 ],
 [ 'A', 'C', 'C', 0.0 ],
 [ 'B', 'A', 'A', 0.0 ],
 [ 'B', 'A', 'B', 0.0 ],
 [ 'B', 'A', 'C', 1.0 ],
 [ 'B', 'B', 'A', 0.5 ],
 [ 'B', 'B', 'B', 0.0 ],
 [ 'B', 'B', 'C', 0.5 ],
 [ 'B', 'C', 'A', 1.0 ],
 [ 'B', 'C', 'B', 0.0 ],
 [ 'B', 'C', 'C', 0.0 ],
 [ 'C', 'A', 'A', 0.0 ],
 [ 'C', 'A', 'B', 1.0 ],
 [ 'C', 'A', 'C', 0.0 ],
 [ 'C', 'B', 'A', 1.0 ],
 [ 'C', 'B', 'B', 0.0 ],
 [ 'C', 'B', 'C', 0.0 ],
```

```
 [ 'C', 'C', 'A', 0.5 ],
 [ 'C', 'C', 'B', 0.5 ],
 [ 'C', 'C', 'C', 0.0 ]], [guest, prize])

s1 = State(guest, name="guest")
s2 = State(prize, name="prize")
s3 = State(monty, name="monty")

# Create the Bayesian network object with a useful name
model = BayesianNetwork("Monty Hall Problem")
# Add the three states to the network
model.add_states(s1, s2, s3)
#Add edges which represent conditional dependencies, where the second
node is
model.add_edge(s1, s3)
model.add_edge(s2, s3)
#Model baked to finalize the internals
model.bake()
print(model.probability([['A', 'A', 'A'],
                             ['A', 'A', 'B'],
                             ['C', 'A', 'B'],['A','B','B']]))
print(model.predict([['A', 'B', None],
                         ['A', 'C', None],
                         ['C', 'C', None],
                     [None,'B','B'],
          ['A',None,'B']]))
print(model.predict([['A', 'B', None],
                 ['A', None, 'C'],
                 [None, 'B', 'A']]))
--------------------------------------------------------------------------
-------------------------------------------
#WEEK -12 HIDDEN MARKOV MODEL

import numpy as np
import itertools
import pandas as pd
# create state space and initial state probabilities
states = ['sleeping', 'eating', 'pooping']

hidden_states = ['healthy', 'sick']
pi = [0.5, 0.5]
state_space = pd.Series(pi, index=hidden_states, name='states')
print(state_space)



a_df = pd.DataFrame(columns=hidden_states, index=hidden_states)
a_df.loc[hidden_states[0]] = [0.7, 0.3]
a_df.loc[hidden_states[1]] = [0.4, 0.6]

print(a_df)

observable_states = states
```

```
b_df = pd.DataFrame(columns=observable_states, index=hidden_states)
b_df.loc[hidden_states[0]] = [0.2, 0.6, 0.2]
b_df.loc[hidden_states[1]] = [0.4, 0.1, 0.5]

print(b_df)

def HMM(obsq,b_df,a_df,pi,states,hidden_states):

      hidst=list(itertools.combinations_with_replacement(hidden_states,le
n(obsq)))
            sum=0
            for k in hidst:
                        prod=1
                        for j in range(len(k)):
                              for i in obsq:
                                    c=0
                                    if c==0:

      prod*=b_df[i][k[j]]*pi[hidden_states.index(k[j])]
                                          c=1
                                    else:
                                          prod*=a_df[k[j]][k[j-
1]]*b_df[i][k[j]]
                        sum+=prod
                        c=0
            return sum

def vertibi(obsq,b_df,a_df,pi,states,hidden_states):
            sum=0

      hidst=list(itertools.combinations_with_replacement(hidden_states,le
n(obsq)))
            for k in hidst:
                        sum1=0
                        prod=1
                        for j in range(len(k)):
                              for i in obsq:
                                    c=0
                                    if c==0:

      prod*=b_df[i][k[j]]*pi[hidden_states.index(k[j])]
                                          c=1
                                    else:
                                          prod*=a_df[k[j]][k[j-
1]]*b_df[i][k[j]]
                        c=0
                        sum1+=prod
                        if(sum1>sum):
                              sum=sum1
                              hs=k
            return sum,hs
```

```
obsq=['pooping','sleeping']
print(HMM(obsq,b_df,a_df,pi,states,hidden_states))
print(vertibi(obsq,b_df,a_df,pi,states,hidden_states))
```