#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 travel

#uC = user input for number of cannibals for left to right

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

not

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

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

1]]\*b\_df[i][k[j]]

return sum

sum+=prod c=0

else:

prod\*=a\_df[k[j]][k[j-

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

1]]\*b\_df[i][k[j]]

c=0 sum1+=prod

else:

prod\*=a\_df[k[j]][k[j-

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))