Q.1 Program to print maltipication table for given no.

```
In [1]: num = int(input("Enter the number: "))

        print("Multiplication Table of", num)

        for i in range(1, 11):

         print(num,"X",i,"=",num * i)
```

```
Enter the number: 6
Multiplication Table of 6
6 X 1 = 6
6 X 2 = 12
6 X 3 = 18
6 X 4 = 24
6 X 5 = 30
6 X 6 = 36
6 X 7 = 42
6 X 8 = 48
6 X 9 = 54
6 X 10 = 60
```

Q.2 Program to check whether given no is prime or not.

```
: num = int(input("Enter the number: "))
  # If given number is greater than 1
  if num > 1:
      # Iterate from 2 to n / 2
      for i in range(2, int(num/2)+1):
          # If num is divisible by any number between
          # 2 and n / 2, it is not prime
          if (num % i) == 0:
              print(num, "is not a prime number")
              break
      else:
          print(num, "is a prime number")
  else:
      print(num, "is not a prime number")
```

```
Enter the number: 8
8 is not a prime number
```

Q.3 Write a program To implement Simple Chatbot.

In [28]:
```
qna={
    "hi":"hey",
    "how are you":"I am fine",
    "what is your name":"my name is ram",
    "how old you are":"I am 10 year old",

}
while True:


        qse = input()

        if(qse =="quit"):
            break

        else:

           print(qna[qse])
```

```
hi
hey
what is your name
my name is ram
quit
```

In [*]:
```
import time
now = time.ctime()
qna={
    "hi":"hey",
    "how are you":"I am fine",
    "what is your name":"my name is ram",
    "how old you are":"I am 10 year old",
    "what is the time now":now,
}
while True:


        qse = input()

        if(qse =="quit"):
            break

        else:

           print(qna[qse])
```

```
hi
hey
what is the time now
Sat Sep 16 12:20:46 2023
```

Q.4 Write a Program to implement code in BFS:

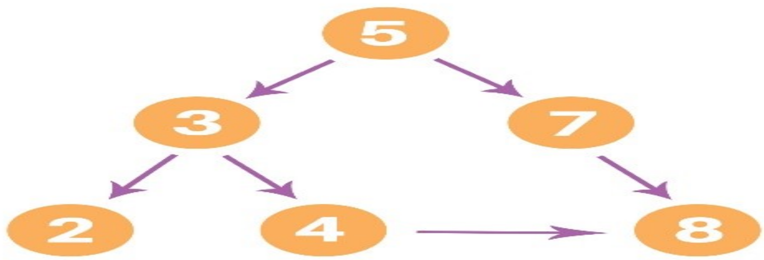
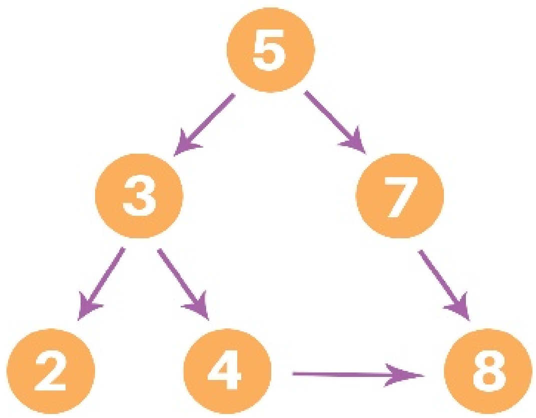


FIGURE 0

```python
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

```
Following is the Breadth-First Search
5 3 7 2 4 8
```

Q.5 Write a Program to implement code in DFS:




```python
# Using a Python dictionary to act as an adjacency list
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

```
Following is the Depth-First Search
5
3
2
4
8
7
```

Q.6 Write a Program to implement code in Water Jug Problem.

```
from collections import deque
def Solution(a, b, target):
	m = {}
	isSolvable = False
	path = []

	q = deque()

	#Initializing with jugs being empty
	q.append((0, 0))

	while (len(q) > 0):

		# Current state
		u = q.popleft()
		if ((u[0], u[1]) in m):
			continue
		if ((u[0] > a or u[1] > b or
			u[0] < 0 or u[1] < 0)):
			continue
		path.append([u[0], u[1]])

		m[(u[0], u[1])] = 1

		if (u[0] == target or u[1] == target):
			isSolvable = True

			if (u[0] == target):
				if (u[1] != 0):
					path.append([u[0], 0])
			else:
				if (u[0] != 0):

					path.append([0, u[1]])
```

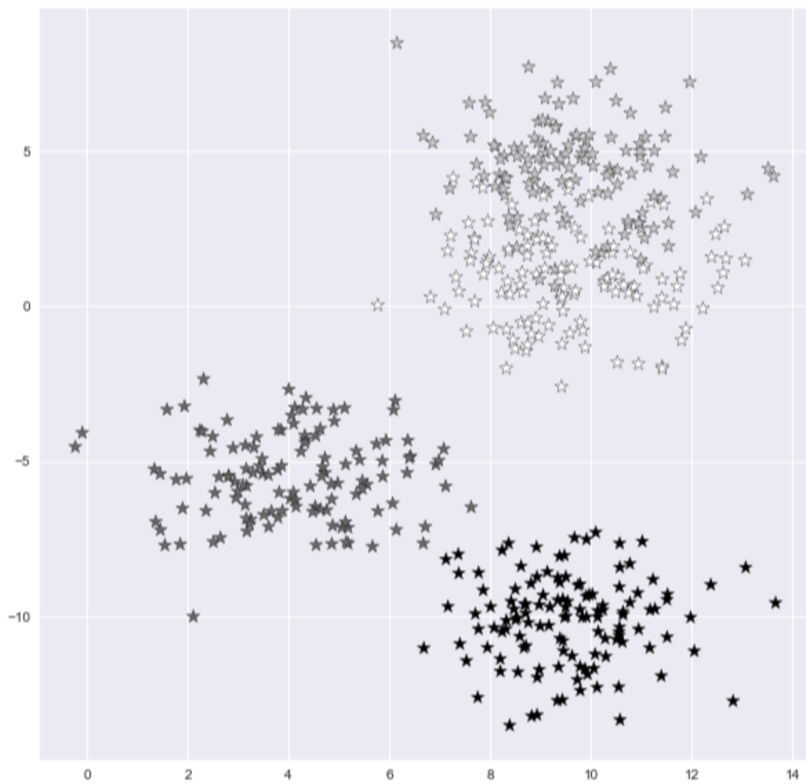
```
            sz = len(path)
            for i in range(sz):
                print("(", path[i][0], ",",
                    path[i][1], ")")
            break

        q.append([u[0], b]) # Fill Jug2
        q.append([a, u[1]]) # Fill Jug1

        for ap in range(max(a, b) + 1):
            c = u[0] + ap
            d = u[1] - ap

            if (c == a or (d == 0 and d >= 0)):
                q.append([c, d])

            c = u[0] - ap
            d = u[1] + ap

            if ((c == 0 and c >= 0) or d == b):
                q.append([c, d])

        q.append([a, 0])

        q.append([0, b])

    if (not isSolvable):
        print("Solution not possible")

if __name__ == '__main__':

    Jug1, Jug2, target = 4, 3, 2
    print("Path from initial state "
        "to solution state ::")

    Solution(Jug1, Jug2, target)
```

```
Path from initial state to solution state ::
( 0 , 0 )
( 0 , 3 )
( 4 , 0 )
( 4 , 3 )
( 3 , 0 )
( 1 , 3 )
( 3 , 3 )
( 4 , 2 )
( 0 , 2 )
```

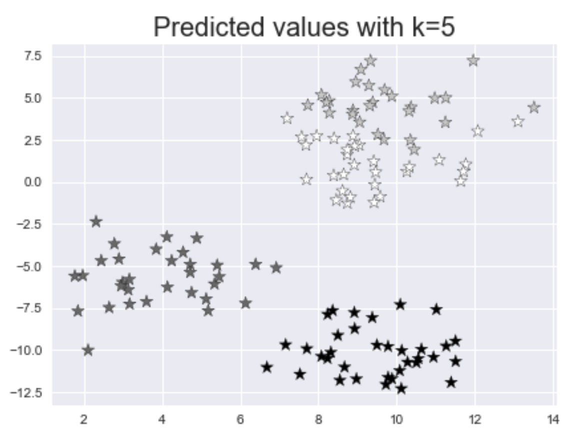
Q.7 Write a Program to implement k-nearest neighbor algorithm.

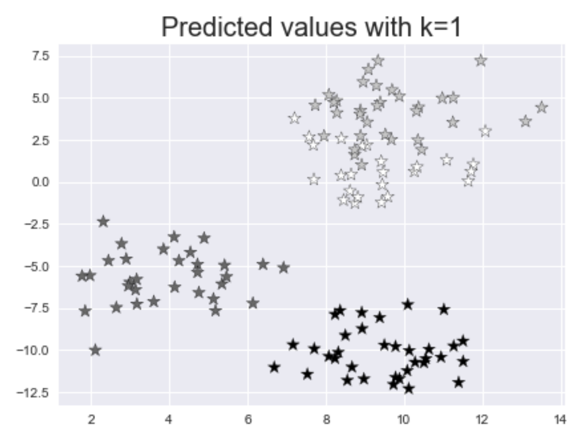
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
X, y = make_blobs(n_samples = 500, n_features = 2, centers = 4,cluster_std = 1.5, random_state = 4)
plt.style.use('seaborn')
plt.figure(figsize = (10,10))
plt.scatter(X[:,0], X[:,1], c=y, marker= '*',s=100,edgecolors='black')
plt.show()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 0)
knn5 = KNeighborsClassifier(n_neighbors = 5)
knn1 = KNeighborsClassifier(n_neighbors=1)
knn5.fit(X_train, y_train)
knn1.fit(X_train, y_train)

y_pred_5 = knn5.predict(X_test)
y_pred_1 = knn1.predict(X_test)
from sklearn.metrics import accuracy_score
print("Accuracy with k=5", accuracy_score(y_test, y_pred_5)*100)
print("Accuracy with k=1", accuracy_score(y_test, y_pred_1)*100)

plt.figure(figsize = (15,5))
plt.subplot(1,2,1)
plt.scatter(X_test[:,0], X_test[:,1], c=y_pred_5, marker= '*', s=100,edgecolors='black')
plt.title("Predicted values with k=5", fontsize=20)

plt.subplot(1,2,2)
plt.scatter(X_test[:,0], X_test[:,1], c=y_pred_1, marker= '*', s=100,edgecolors='black')
plt.title("Predicted values with k=1", fontsize=20)
plt.show()
```

Out put:



```
Accuracy with k=5 93.60000000000001
Accuracy with k=1 90.4
```



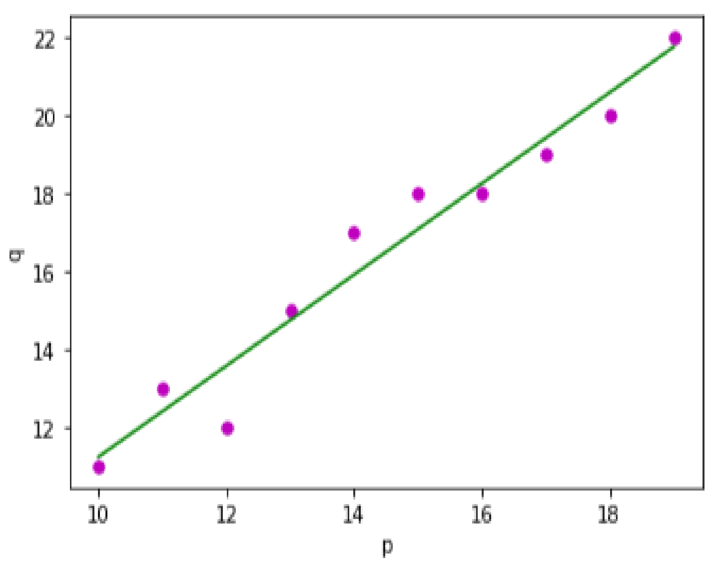


Q.8 Write a Program to implement Regression algorithm.

```
import numpy as nmp
import matplotlib.pyplot as mtplt

def estimate_coeff(p, q):
# Here, we will estimate the total number of points or observation
    n1 = nmp.size(p)
# Now, we will calculate the mean of a and b vector
    m_p = nmp.mean(p)
    m_q = nmp.mean(q)

# here, we will calculate the cross deviation and deviation about a
    SS_pq = nmp.sum(q * p) - n1 * m_q * m_p
    SS_pp = nmp.sum(p * p) - n1 * m_p * m_p

# here, we will calculate the regression coefficients
    b_1 = SS_pq / SS_pp
    b_0 = m_q - b_1 * m_p

    return (b_0, b_1)

def plot_regression_line(p, q, b):
# Now, we will plot the actual points or observation as scatter plot
    mtplt.scatter(p, q, color = "m",
            marker = "o", s = 30)

# here, we will calculate the predicted response vector
    q_pred = b[0] + b[1] * p

# here, we will plot the regression line
    mtplt.plot(p, q_pred, color = "g")

# here, we will put the labels
    mtplt.xlabel('p')
    mtplt.ylabel('q')
```

```python
# here, we will define the function to show plot
    mtplt.show()

def main():
# entering the observation points or data
    p = nmp.array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
    q = nmp.array([11, 13, 12, 15, 17, 18, 18, 19, 20, 22])

# now, we will estimate the coefficients
    b = estimate_coeff(p, q)
    print("Estimated coefficients are :\nb_0 = {} \  \nb_1 = {}".format(b[0], b[1]))

# Now, we will plot the regression line
    plot_regression_line(p, q, b)

if __name__ == "__main__":
    main()
```

```
Estimated coefficients are :
b_0 = -0.460606060606060609 \
b_1 = 1.1696969696969697
```



Q.9 Implementing the Hill Climbing Algorithm in Python

```
import random

import numpy as np

import networkx as nx


#coordinate of the points/cities

coordinate = np.array([[1,2], [30,21], [56,23], [8,18], [20,50], [3,4], [11,6], [6,7],
[15,20], [10,9], [12,12]])


#adjacency matrix for a weighted graph based on the given coordinates

def generate_matrix(coordinate):

    matrix = []

    for i in range(len(coordinate)):

        for j in range(len(coordinate)) :

            p = np.linalg.norm(coordinate[i] - coordinate[j])

            matrix.append(p)

    matrix = np.reshape(matrix, (len(coordinate),len(coordinate)))

    #print(matrix)

    return matrix


#finds a random solution

def solution(matrix):
```

```
    points = list(range(0, len(matrix)))
    solution = []
    for i in range(0, len(matrix)):
        random_point = points[random.randint(0, len(points) - 1)]
        solution.append(random_point)
        points.remove(random_point)
    return solution



#calculate the path based on the random solution
def path_length(matrix, solution):
    cycle_length = 0
    for i in range(0, len(solution)):
        cycle_length += matrix[solution[i]][solution[i - 1]]
    return cycle_length


#generate neighbors of the random solution by swapping cities and returns the best neighbor
def neighbors(matrix, solution):
    neighbors = []
    for i in range(len(solution)):
        for j in range(i + 1, len(solution)):
```

```
            neighbor = solution.copy()
            neighbor[i] = solution[j]
            neighbor[j] = solution[i]
            neighbors.append(neighbor)

    #assume that the first neighbor in the list is the best neighbor
    best_neighbor = neighbors[0]
    best_path = path_length(matrix, best_neighbor)

    #check if there is a better neighbor
    for neighbor in neighbors:
        current_path = path_length(matrix, neighbor)
        if current_path < best_path:
            best_path = current_path
            best_neighbor = neighbor
    return best_neighbor, best_path


def hill_climbing(coordinate):
    matrix = generate_matrix(coordinate)

    current_solution = solution(matrix)
```

```
    current_path = path_length(matrix, current_solution)

    neighbor = neighbors(matrix,current_solution)[0]

    best_neighbor, best_neighbor_path = neighbors(matrix, neighbor)


    while best_neighbor_path < current_path:

        current_solution = best_neighbor

        current_path = best_neighbor_path

        neighbor = neighbors(matrix, current_solution)[0]

        best_neighbor, best_neighbor_path = neighbors(matrix, neighbor)


    return current_path, current_solution

final_solution = hill_climbing(coordinate)

print("The solution is \n", final_solution[1])
```

Output—

```
The solution is
 [3, 8, 4, 2, 1, 6, 5, 0, 7, 9, 10]
```

Q.10 Write a python program to generate Calendar for the given month and year?.

```python
import calendar

year = int(input ("Please enter the Year: ")) # Here, it will take the year
month = int(input ("Please enter the month: "))    # Here, it will take the month

# Now, we will display the calendar
print("The Calendar of: ", calendar.month(year, month))
```

```
Please enter the Year: 2000
Please enter the month: 12
The Calendar of:     December 2000
Mo Tu We Th Fr Sa Su
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

Q.11 Python Program to Remove Punctuation from a String

In [9]:
```python
# define punctuation
punctuation = '''''!()-[]{};:'"\,<>./?@#$%^&*_~'''
# take input from the user   keywords
my_str = input("Enter a string: ")
# remove punctuation from the string
no_punct = ""
for char in my_str:
   if char not in punctuation:
       no_punct = no_punct + char
# display the unpunctuated string
print(no_punct)
```

```
Enter a string: Hello ...... this is encripted (@#$%^^&*)keywords
Hello  this is encripted keywords
```

Q.12 **Write a program to implement Hangman game using python.**

Description:

Hangman is a classic word-guessing game. The user should guess the word correctly by entering alphabets of the user choice. The Program will get input as single alphabet from the user and it will matchmaking with the alphabets in the original

```python
import time
import random
name = input("What is your name? ")
```

```
print ("Hello, " + name, "Time to play hangman!")
time.sleep(1)
print ("Start guessing...\n")
time.sleep(0.5)
## A List Of Secret Words
words = ['python','programming','treasure','creative','medium','horror']
word = random.choice(words)
guesses = ''
turns = 5
while turns > 0:
    failed = 0
    for char in word:
        if char in guesses:
            print (char,end="")
        else:
            print ("_",end=""),
            failed += 1
    if failed == 0:
        print ("\nYou won")
        break
    guess = input("\nguess a character:")
    guesses += guess
    if guess not in word:
        turns -= 1
        print("\nWrong")
        print("\nYou have", + turns, 'more guesses')
        if turns == 0:
            print ("\nYou Lose")
```

Output–

```
What is your name? sapna
Hello, sapna Time to play hangman!
Start guessing...

______
guess a character:p

Wrong

You have 4 more guesses
______
guess a character:m
m____m
guess a character:e
me___m
guess a character:d
med__m
guess a character:i
medi_m
guess a character:a
```

```
Wrong

You have 3 more guesses
medi_m
guess a character:u
medium
You won
```

Q.13 Write a python program to implement Lemmatization using NLTK

```
import nltknltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import SnowballStemmer
text = 'Jim has an engineering background and he works as project
manager! Before he was working as a developer in a software
company'
snow = SnowballStemmer('english')stemmed_sentence = []
# Word Tokenizer
words = word_tokenize(text)
for w in words:
    # Apply Stemming
    stemmed_sentence.append(snow.stem(w))
stemmed_text = " ".join(stemmed_sentence)

stemmed_text
```

Output—

```
'jim has an engin background and he work as project manag ! befor he was
work as a develop in a softwar compani'
```

Q.14 Write a python program to remove stop words for a given passage from a text file using NLTK?.

```
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
# Add text
text = "How to remove stop words with NLTK library in Python"
print("Text:", text)
# Convert text to lowercase and split to a list of words
tokens = word_tokenize(text.lower())
print("Tokens:", tokens)
# Remove stop words
english_stopwords = stopwords.words('english')
tokens_wo_stopwords = [t for t in tokens if t not in english_stopwords]
print("Text without stop words:", " ".join(tokens_wo_stopwords))
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\Admin\AppData\Roaming\nltk_data...
```

```
Text: How to remove stop words with NLTK library in Python
Tokens: ['how', 'to', 'remove', 'stop', 'words', 'with', 'nltk', 'library', 'in', 'python']
Text without stop words: remove stop words nltk library python
```

```
[nltk_data]   Unzipping corpora\stopwords.zip.
```

Q.15 Write a python program implement tic-tac-toe using alpha beta pruning

```
import math

# Constants for representing the players and empty cells
EMPTY = "-"
PLAYER_X = "X"
PLAYER_O = "O"

# The game board
board = [EMPTY, EMPTY, EMPTY,
         EMPTY, EMPTY, EMPTY,
         EMPTY, EMPTY, EMPTY]

# Function to print the game board
def print_board(board):
```

```
    print("---------")
    for i in range(3):
        print("|", board[i*3], "|", board[i*3 + 1], "|", board[i*3 + 2],
"|")
        print("---------")

# Function to check if a player has won
def check_winner(board):
    winning_combinations = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # columns
        [0, 4, 8], [2, 4, 6]  # diagonals
    ]

    for combination in winning_combinations:
        if board[combination[0]] == board[combination[1]] ==
board[combination[2]] != EMPTY:
            return board[combination[0]]

    if EMPTY not in board:
        return "tie"

    return None

# Function to evaluate the game board
def evaluate(board):
    winner = check_winner(board)

    if winner == PLAYER_X:
        return 1
    elif winner == PLAYER_O:
        return -1
    else:
        return 0

# Minimax function with alpha-beta pruning
def minimax(board, depth, alpha, beta, maximizing_player):
    if check_winner(board) is not None or depth == 0:
        return evaluate(board)

    if maximizing_player:
        max_eval = -math.inf
        for i in range(9):
            if board[i] == EMPTY:
                board[i] = PLAYER_X
```

```python
                eval_score = minimax(board, depth - 1, alpha, beta, False)
                board[i] = EMPTY
                max_eval = max(max_eval, eval_score)
                alpha = max(alpha, eval_score)
                if beta <= alpha:
                    break
        return max_eval
    else:
        min_eval = math.inf
        for i in range(9):
            if board[i] == EMPTY:
                board[i] = PLAYER_O
                eval_score = minimax(board, depth - 1, alpha, beta, True)
                board[i] = EMPTY
                min_eval = min(min_eval, eval_score)
                beta = min(beta, eval_score)
                if beta <= alpha:
                    break
        return min_eval

# Function to find the best move using minimax with alpha-beta pruning
def find_best_move(board):
    best_score = -math.inf
    best_move = None

    for i in range(9):
        if board[i] == EMPTY:
            board[i] = PLAYER_X
            move_score = minimax(board, 9, -math.inf, math.inf, False)
            board[i] = EMPTY

            if move_score > best_score:
                best_score = move_score
                best_move = i

    return best_move

# Main game loop
while True:
    print_board(board)
    winner = check_winner(board)

    if winner is not None:
        if winner == "tie":
            print("It's a tie!")
```

```
        else:
            print("Player", winner, "wins!")
        break

    if len([cell for cell in board if cell != EMPTY]) % 2 == 0:
        # Player O's turn
        while True:
            move = int(input("Enter O's move (0-8): "))
            if board[move] == EMPTY:
                board[move] = PLAYER_O
                break
            else:
                print("Invalid move! Try again.")
    else:
        # Player X's turn
        move = find_best_move(board)
        board[move] = PLAYER_X
```

Output—

```
---------
| - | - | - |
---------
| - | - | - |
---------
| - | - | - |
---------
Enter O's move (0-8): 0
---------
| O | - | - |
---------
| - | - | - |
---------
| - | - | - |
---------
---------
| O | - | - |
---------
| - | X | - |
---------
| - | - | - |
---------
Enter O's move (0-8): 8
---------
| O | - | - |
---------
| - | X | - |
```

```
---------
| - | - | O |
---------
---------
| O | X | - |
---------
| - | X | - |
---------
| - | - | O |
---------
Enter O's move (0-8): 7
---------
| O | X | - |
---------
| - | X | - |
---------
| - | O | O |
---------
---------
| O | X | - |
---------
| - | X | - |
---------
| X | O | O |
---------
Enter O's move (0-8): 2
---------
| O | X | O |
---------
| - | X | - |
---------
| X | O | O |
---------
---------
| O | X | O |
---------
| - | X | X |
---------
| X | O | O |
---------
Enter O's move (0-8): 3
---------
| O | X | O |
---------
| O | X | X |
---------
| X | O | O |
---------
It's a tie!
```

Q.16 Write a Python program to accept a string. Find and print the number of upper case alphabets and lower case alphabets.

```
In [23]: x=input("Enter the string:- ")
         def char(x):
           u=0
           l=0
           for i in x:
               if i>='a' and i<='z':
                l+=1

               if i >='A' and i<='Z':
                u+=1

           print("LowerCase letter in the String",l)
           print("UpperCase letter in the String",u)
         char(x)
```

```
Enter the string:- PythOn pRograM
LowerCase letter in the String 9
UpperCase letter in the String 4
```

Q.17 Write a Python program to solve tic-tac-toe problem.

```
# Tic-Tac-Toe Program using
# random number in Python

# importing all necessary libraries
import numpy as np
import random
from time import sleep

# Creates an empty board
 def create_board():

    return(np.array([[0, 0, 0],

                    [0, 0, 0],
```

```
                 [0, 0, 0]]))

# Check for empty places on board



def possibilities(board):
    l = []


    for i in range(len(board)):
        for j in range(len(board)):


            if board[i][j] == 0:
                l.append((i, j))
    return(l)

# Select a random place for the player



def random_place(board, player):
    selection = possibilities(board)
    current_loc = random.choice(selection)
    board[current_loc] = player
    return(board)
```

```
# Checks whether the player has three
# of their marks in a horizontal row


def row_win(board, player):
    for x in range(len(board)):
        win = True

        for y in range(len(board)):
            if board[x, y] != player:
                win = False
                continue

        if win == True:
            return(win)
    return(win)

# Checks whether the player has three
# of their marks in a vertical row


def col_win(board, player):
    for x in range(len(board)):
        win = True
```

```
        for y in range(len(board)):
            if board[y][x] != player:
                win = False
                continue

        if win == True:
            return(win)
    return(win)

# Checks whether the player has three
# of their marks in a diagonal row


def diag_win(board, player):
    win = True
    y = 0
    for x in range(len(board)):
        if board[x, x] != player:
            win = False
    if win:
        return win
    win = True
    if win:
```

```
        for x in range(len(board)):
            y = len(board) - 1 - x
            if board[x, y] != player:
                win = False
    return win
 # Evaluates whether there is
# a winner or a tie
def evaluate(board):
    winner = 0
  for player in [1, 2]:
        if (row_win(board, player) or
                col_win(board, player) or
                diag_win(board, player)):

            winner = player

    if np.all(board != 0) and winner == 0:
        winner = -1
    return winner

# Main function to start the game



def play_game():
```

```
    board, winner, counter = create_board(), 0, 1
    print(board)
    sleep(2)

    while winner == 0:
        for player in [1, 2]:
            board = random_place(board, player)
            print("Board after " + str(counter) + " move")
            print(board)
            sleep(2)
            counter += 1
            winner = evaluate(board)
            if winner != 0:
                break
    return(winner)


# Driver Code
print("Winner is: " + str(play_game()))
```

Output—

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
Board after 1 move
[[0 0 0]
 [0 0 0]
```

```
 [0 0 1]]
Board after 2 move
[[0 2 0]
 [0 0 0]
 [0 0 1]]
Board after 3 move
[[0 2 0]
 [1 0 0]
 [0 0 1]]
Board after 4 move
[[0 2 2]
 [1 0 0]
 [0 0 1]]
Board after 5 move
[[0 2 2]
 [1 0 0]
 [0 1 1]]
Board after 6 move
[[2 2 2]
 [1 0 0]
 [0 1 1]]
Winner is: 2
```

Q.18 Write a Program to Implement Tower of Hanoi using Python.

]:

```python
# Creating a recursive function
def tower_of_hanoi(disks, source, auxiliary, target):
    if(disks == 1):
        print('Move disk 1 from rod {} to rod {}.'.format(source, target))
        return
    # function call itself
    tower_of_hanoi(disks - 1, source, target, auxiliary)
    print('Move disk {} from rod {} to rod {}.'.format(disks, source, target))
    tower_of_hanoi(disks - 1, auxiliary, source, target)


disks = int(input('Enter the number of disks: '))
# We are referring source as A, auxiliary as B, and target as C
tower_of_hanoi(disks, 'A', 'B', 'C')  # Calling the function
```

```
Enter the number of disks: 3
Move disk 1 from rod A to rod C.
Move disk 2 from rod A to rod B.
Move disk 1 from rod C to rod B.
Move disk 3 from rod A to rod C.
Move disk 1 from rod B to rod A.
Move disk 2 from rod B to rod C.
Move disk 1 from rod A to rod C.
```

Q.20 Write a python program to sort the sentence in alphabetical order?

```python
my_str = input("Enter a string: ")
# breakdown the string into a list of words
words = my_str.split()
# sort the list
words.sort()
# display the sorted words
for word in words:
    print(word)


```

```
Enter a string: All the student are good
All
are
good
student
the
```

Q.21 Write a Python program to solve 8-puzzle problem.

```python
# Python code to display the way from the root
# node to the final destination node for N*N-1 puzzle
# algorithm by the help of Branch and Bound technique
# The answer assumes that the instance of the
# puzzle can be solved

# Importing the 'copy' for deepcopy method
import copy

# Importing the heap methods from the python
# library for the Priority Queue
from heapq import heappush, heappop

# This particular var can be changed to transform
# the program from 8 puzzle(n=3) into 15
```

```python
# puzzle(n=4) and so on ...
n = 3

# bottom, left, top, right
rows = [ 1, 0, -1, 0 ]
cols = [ 0, -1, 0, 1 ]

# creating a class for the Priority Queue
class priorityQueue:

    # Constructor for initializing a
    # Priority Queue
    def __init__(self):
        self.heap = []

    # Inserting a new key 'key'
    def push(self, key):
        heappush(self.heap, key)

    # funct to remove the element that is minimum,
    # from the Priority Queue
    def pop(self):
        return heappop(self.heap)

    # funct to check if the Queue is empty or not
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

# structure of the node
class nodes:

```

```python
    def __init__(self, parent, mats, empty_tile_posi,
                 costs, levels):

        # This will store the parent node to the
        # current node And helps in tracing the
        # path when the solution is visible
        self.parent = parent

        # Useful for Storing the matrix
        self.mats = mats

        # useful for Storing the position where the
        # empty space tile is already existing in the matrix
        self.empty_tile_posi = empty_tile_posi

        # Store no. of misplaced tiles
        self.costs = costs

        # Store no. of moves so far
        self.levels = levels

    # This func is used in order to form the
    # priority queue based on
    # the costs var of objects
    def __lt__(self, nxt):
        return self.costs < nxt.costs

# method to calc. the no. of
# misplaced tiles, that is the no. of non-blank
# tiles not in their final posi
def calculateCosts(mats, final) -> int:

    count = 0
    for i in range(n):
```

```
        for j in range(n):
            if ((mats[i][j]) and
                (mats[i][j] != final[i][j])):
                count += 1

    return count

def newNodes(mats, empty_tile_posi, new_empty_tile_posi,
             levels, parent, final) -> nodes:

    # Copying data from the parent matrixes to the present matrixes
    new_mats = copy.deepcopy(mats)

    # Moving the tile by 1 position
    x1 = empty_tile_posi[0]
    y1 = empty_tile_posi[1]
    x2 = new_empty_tile_posi[0]
    y2 = new_empty_tile_posi[1]
    new_mats[x1][y1], new_mats[x2][y2] = new_mats[x2][y2], new_mats[x1][y1]

    # Setting the no. of misplaced tiles
    costs = calculateCosts(new_mats, final)

    new_nodes = nodes(parent, new_mats, new_empty_tile_posi,
                      costs, levels)
    return new_nodes

# func to print the N by N matrix
def printMatsrix(mats):

    for i in range(n):
        for j in range(n):
            print("%d " % (mats[i][j]), end = " ")

```

```
        print()

# func to know if (x, y) is a valid or invalid
# matrix coordinates
def isSafe(x, y):

    return x >= 0 and x < n and y >= 0 and y < n

# Printing the path from the root node to the final node
def printPath(root):

    if root == None:
        return

    printPath(root.parent)
    printMatsrix(root.mats)
    print()

# method for solving N*N - 1 puzzle algo
# by utilizing the Branch and Bound technique. empty_tile_posi is
# the blank tile position initially.
def solve(initial, empty_tile_posi, final):

    # Creating a priority queue for storing the live
    # nodes of the search tree
    pq = priorityQueue()

    # Creating the root node
    costs = calculateCosts(initial, final)
    root = nodes(None, initial,
                 empty_tile_posi, costs, 0)

    # Adding root to the list of live nodes
    pq.push(root)
```

```
        # Discovering a live node with min. costs,
        # and adding its children to the list of live
        # nodes and finally deleting it from
        # the list.
        while not pq.empty():

            # Finding a live node with min. estimatsed
            # costs and deleting it form the list of the
            # live nodes
            minimum = pq.pop()

            # If the min. is ans node
            if minimum.costs == 0:

                # Printing the path from the root to
                # destination;
                printPath(minimum)
                return

            # Generating all feasible children
            for i in range(n):
                new_tile_posi = [
                    minimum.empty_tile_posi[0] + rows[i],
                    minimum.empty_tile_posi[1] + cols[i], ]

                if isSafe(new_tile_posi[0], new_tile_posi[1]):

                    # Creating a child node
                    child = newNodes(minimum.mats,
                                minimum.empty_tile_posi,
                                new_tile_posi,
                                minimum.levels + 1,
                                minimum, final,)
```

```
                    # Adding the child to the list of live nodes
                    pq.push(child)

# Main Code

# Initial configuration
# Value 0 is taken here as an empty space
initial = [ [ 1, 2, 3 ],
            [ 5, 6, 0 ],
            [ 7, 8, 4 ] ]

# Final configuration that can be solved
# Value 0 is taken as an empty space
final = [ [ 1, 2, 3 ],
          [ 5, 8, 6 ],
          [ 0, 7, 4 ] ]

# Blank tile coordinates in the
# initial configuration
empty_tile_posi = [ 1, 2 ]

# Method call for solving the puzzle
solve(initial, empty_tile_posi, final)
```

Output----

```
1   2   3
5   6   0
7   8   4

1   2   3
5   0   6
7   8   4
```

```
1   2   3
5   8   6
7   0   4

1   2   3
5   8   6
0   7   4
```

Q.22 Write a Python program for the following Cryptorithmetic problems

Input: *arr[][] = {“SIX”, “SEVEN”, “SEVEN”}, S = “TWENTY”*
Output: *Yes*
Explanation:
One of the possible ways is:
1. *Map the characters as the following, ‘S’? 6, ‘I’?5, ‘X’?0, ‘E’?8, ‘V’?7, ‘N’?2, ‘T’?1, ‘W’?’3', ‘Y’?4.*
2. *Now, after encoding the strings “SIX”, “SEVEN”, and “TWENTY”, modifies to 650, 68782 and 138214 respectively.*
3. *Thus, the sum of the values of “SIX”, “SEVEN”, and “SEVEN” is equal to (650+ 68782+ 68782 = 138214), which is equal to the value of the string “TWENTY”.*

Therefore, print “Yes”.

```
# Python program for the above approach


# Function to check if the
# assignment of digits to
# characters is possible
def isSolvable(words, result):
    # Stores the value
    # assigned to alphabets
    mp = [-1]*(26)

    # Stores if a number
    # is assigned to any
```

```
# character or not
used = [0]*(10)

# Stores the sum of position
# value of a character
# in every string
Hash = [0]*(26)

# Stores if a character
# is at index 0 of any
# string
CharAtfront = [0]*(26)

# Stores the string formed
# by concatenating every
# occurred character only
# once
uniq = ""

# Iterator over the array,
# words
for word in range(len(words)):
    # Iterate over the string,
    # word
    for i in range(len(words[word])):
        # Stores the character
        # at ith position
        ch = words[word][i]
```

```
        # Update Hash[ch-'A]
        Hash[ord(ch) - ord('A')] += pow(10, len(words[word]) - i - 1)

        # If mp[ch-'A'] is -1
        if mp[ord(ch) - ord('A')] == -1:
            mp[ord(ch) - ord('A')] = 0
            uniq += str(ch)

        # If i is 0 and word
        # length is greater
        # than 1
        if i == 0 and len(words[word]) > 1:
            CharAtfront[ord(ch) - ord('A')] = 1

# Iterate over the string result
for i in range(len(result)):
    ch = result[i]

    Hash[ord(ch) - ord('A')] -= pow(10, len(result) - i - 1)

    # If mp[ch-'A] is -1
    if mp[ord(ch) - ord('A')] == -1:
        mp[ord(ch) - ord('A')] = 0
        uniq += str(ch)

    # If i is 0 and length of
    # result is greater than 1
    if i == 0 and len(result) > 1:
        CharAtfront[ord(ch) - ord('A')] = 1
```

```
    mp = [-1]*(26)

    # Recursive call of the function
    return True

# Auxiliary Recursive function
# to perform backtracking
def solve(words, i, S, mp, used, Hash, CharAtfront):
    # If i is word.length
    if i == len(words):
        # Return true if S is 0
        return S == 0

    # Stores the character at
    # index i
    ch = words[i]

    # Stores the mapped value
    # of ch
    val = mp[ord(words[i]) - ord('A')]

    # If val is not -1
    if val != -1:
        # Recursion
        return solve(words, i + 1, S + val * Hash[ord(ch) - ord('A')], mp, used,
Hash, CharAtfront)

    # Stores if there is any
```

```
    # possible solution
    x = False

    # Iterate over the range
    for l in range(10):
        # If CharAtfront[ch-'A']
        # is true and l is 0
        if CharAtfront[ord(ch) - ord('A')] == 1 and l == 0:
            continue

        # If used[l] is true
        if used[l] == 1:
            continue

        # Assign l to ch
        mp[ord(ch) - ord('A')] = l

        # Marked l as used
        used[l] = 1

        # Recursive function call
        x |= solve(words, i + 1, S + l * Hash[ord(ch) - ord('A')], mp, used, Hash,
CharAtfront)

        # Backtrack
        mp[ord(ch) - ord('A')] = -1

        # Unset used[l]
        used[l] = 0
```

```
    # Return the value of x;
    return x

arr = [ "SIX", "SEVEN", "SEVEN" ]
S = "TWENTY"

# Function Call
if isSolvable(arr, S):
    print("Yes")
else:
    print("No")
```

output—

Yes

Q.23 Write a Python program to implement Mini-Max Algorithm.

```python
import math

def minimax (curDepth, nodeIndex,
maxTurn, scores,
targetDepth):

  # base case : targetDepth reached
  if (curDepth == targetDepth):
      return scores[nodeIndex]

  if (maxTurn):
      return max(minimax(curDepth + 1, nodeIndex * 2,
      False, scores, targetDepth),
      minimax(curDepth + 1, nodeIndex * 2 + 1,
      False, scores, targetDepth))

  else:
      return min(minimax(curDepth + 1, nodeIndex * 2,
      True, scores, targetDepth),
      minimax(curDepth + 1, nodeIndex * 2 + 1,
      True, scores, targetDepth))

  # Driver code
scores = [3, 5, 2, 9, 12, 5, 23, 23]
treeDepth = math.log(len(scores), 2)
print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))
```

Output—

```
The optimal value is : 12
```

Q.24 Write a Python program to implement A* algorithm.

```python
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

        #ditance 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:
```

```python
                    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': 99,
            'D': 1,
            'E': 7,
            'G': 0,

        }

        return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],

}
aStarAlgo('A', 'G')
```

output:-

```
Path found: ['A', 'E', 'D', 'G']
```

Q.25 Write a Program to Implement Monkey Banana Problem using Python

```
import random

class Monkey:
        def __init__(self, bananas):
                self.bananas = bananas

        def __repr__(self):
                return "Monkey with %d bananas." % self.bananas

monkeys = [Monkey(random.randint(0, 50)) for i in range(5)]


print "Random monkeys:"
print monkeys

print

def number_of_bananas(monkey):
    """Returns number of bananas that monkey has."""
    return monkey.bananas

print "number_of_bananas( FIRST MONKEY ): ", number_of_bananas(monkeys[0])

print

max_monkey = max(monkeys, key=number_of_bananas)
print "Max monkey: ", max_monkey
```

output—

```
Random monkeys:
[Monkey with 5 bananas., Monkey with 29 bananas., Monkey with 43 bananas.,
Monkey with 1 bananas., Monkey with 32 bananas.]
number_of_bananas( FIRST MONKEY ):  5
Max monkey:  Monkey with 43 bananas.
```