1. **Write a python program to print the multiplication table for the given number.**

num = int(input("Enter a number: "))

for i in range(1, 11):

print(num, "x", i, "=", num \* i)

1. **Write a python program to check whether the given number is prime or not.**

def is\_prime(n):

if n <= 1:

return False

for i in range(2, n):

if n % i == 0:

return False

return True

number = int(input("Enter a number: "))

if is\_prime(number):

print(number, "is a prime number.")

else:

print(number, "is not a prime number.")

**3. Write a python program to find factorial of a given number.**

def factorial(n):

if n == 0:

return 1

return n \* factorial(n-1)

number = int(input("Enter a number: "))

result = factorial(number)

print("The factorial of", number, "is", result)

**4.Write a Program to Implement Tower of Hanoi using Python.**

def TowerOfHanoi(n , from\_rod, to\_rod, aux\_rod):

if n == 1:

print("Move disk 1 from rod",from\_rod,"to rod",to\_rod)

return

TowerOfHanoi(n-1, from\_rod, aux\_rod, to\_rod)

print("Move disk",n,"from rod",from\_rod,"to rod",to\_rod)

TowerOfHanoi(n-1, aux\_rod, to\_rod, from\_rod)

n = 4

TowerOfHanoi(n, 'A', 'C', 'B')

**5. Write a python program to implement simple Chatbot.**

print("Hi, I'm a chatbot. How can I help you today?")

while True:

user\_input = input("You: ")

if user\_input.lower() in ["hi", "hello", "hey"]:

print("Chatbot: Hello! How can I help you today?")

elif user\_input.lower() == "what's your name?":

print("Chatbot: My name is Chatbot.")

elif user\_input.lower() == "what do you do?":

print("Chatbot: I help people with information and answer questions. What can I help you with today?")

elif user\_input.lower() in ["bye", "exit", "quit"]:

print("Chatbot: Have a great day! Bye.")

break

else:

print("Chatbot: I'm sorry, I don't understand what you're saying. Can you rephrase that?")

**6.Write a python program to sort the sentence in alphabetical order.**

sentence = input("Enter a sentence: ")

words = sentence.split()

words.sort()

print("The sorted words are:")

for word in words:

print(word)

**7. Write a python program to generate Calendar for a given month and year.**

import calendar

y = int(input("Input the year : "))

m = int(input("Input the month : "))

print(calendar.month(y, m))

**8. Write a python program to implement Simple Calculator.**

def calculate(operation, num1, num2):

if operation == '+':

return num1 + num2

elif operation == '-':

return num1 - num2

elif operation == '\*':

return num1 \* num2

elif operation == '/':

return num1 / num2

else:

return "Invalid operator"

print("Simple Calculator")

num1 = float(input("Enter first number: "))

num2 = float(input("Enter second number: "))

operation = input("Enter an operator (+, -, \*, /): ")

result = calculate(operation, num1, num2)

print("Result: ", result)

**9 ) Write a python program to implement Breadth First Search Traversal.**

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] \* (max(self.graph) + 1)

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 is Breadth First Traversal"

" (starting from vertex 2)")

g.BFS(2)

**10. Write a python program to implement Depth First Search Traversal.**

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)

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

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)

**11. Write a python program to implement Water Jug Problem.**

from collections import defaultdict

jug1, jug2, aim = 3, 5, 4

visited = defaultdict(lambda: False)

def waterJugSolver(amt1, amt2):

if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):

print(amt1, amt2)

return True

if visited[(amt1, amt2)] == False:

print(amt1, amt2)

visited[(amt1, amt2)] = True

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))))

else:

return False

print("Steps: ")

waterJugSolver(0, 0)

**12. Write a Python Program to Implement Tic-Tac-Toe game.**

board = [' ' for x in range(9)]

def print\_board():

row1 = "| {} | {} | {} |".format(board[0], board[1], board[2])

row2 = "| {} | {} | {} |".format(board[3], board[4], board[5])

row3 = "| {} | {} | {} |".format(board[6], board[7], board[8])

print()

print(row1)

print(row2)

print(row3)

print()

def player\_move(icon):

if icon == 'X':

number = 1

elif icon == 'O':

number = 2

print("Your turn player {}".format(number))

choice = int(input("Enter your move (1-9): ").strip())

if board[choice - 1] == ' ':

board[choice - 1] = icon

else:

print()

print("That space is already taken!")

def is\_victory(icon):

if (board[0] == icon and board[1] == icon and board[2] == icon) or \

(board[3] == icon and board[4] == icon and board[5] == icon) or \

(board[6] == icon and board[7] == icon and board[8] == icon) or \

(board[0] == icon and board[3] == icon and board[6] == icon) or \

(board[1] == icon and board[4] == icon and board[7] == icon) or \

(board[2] == icon and board[5] == icon and board[8] == icon) or \

(board[0] == icon and board[4] == icon and board[8] == icon) or \

(board[2] == icon and board[4] == icon and board[6] == icon):

return True

else:

return False

def is\_draw():

if ' ' not in board:

return True

else:

return False

while True:

print\_board()

player\_move('X')

print\_board()

if is\_victory('X'):

print("X wins! Congratulations!")

break

elif is\_draw():

print("It's a draw!")

break

player\_move('O')

if is\_victory('O'):

print\_board()

print("O wins! Congratulations!")

break

elif is\_draw():

print("It's a draw!")

break

**13. Write a Python program to implement Best-First search algorithm.**

graph = {

'A': [('B', 4), ('C', 3)],

'B': [('D', 5), ('E', 6)],

'C': [('F', 8)],

'D': [],

'E': [('G', 3)],

'F': [('H', 2)],

'G': [('I', 4)],

'H': [('J', 6)],

'I': [],

'J': []

}

def best\_first\_search(graph, start, goal):

visited = set()

queue = [(0, start)]

while queue:

cost, node = queue.pop(0)

if node == goal:

return True

visited.add(node)

neighbors = graph[node]

for neighbor, neighbor\_cost in neighbors:

if neighbor not in visited:

queue.append((neighbor\_cost, neighbor))

queue.sort()

return False

start = 'A'

goal = 'J'

found = best\_first\_search(graph, start, goal)

if found:

print("Goal reached!")

else:

print("Goal not found.")

**14. Write a Program to Implement Travelling Salesman Problem using Python.**

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))

**OUTPUT : - 80 .**

**15. Write a Python program for Hill climbing algorithm.**

import random

def objective\_function(x):

return x\*\*2

def hill\_climbing(objective\_function, x\_min, x\_max, max\_iter):

x = random.uniform(x\_min, x\_max)

for i in range(max\_iter):

current\_obj = objective\_function(x)

delta = random.uniform(-0.1, 0.1)

neighbor = x + delta

neighbor\_obj = objective\_function(neighbor)

if neighbor\_obj < current\_obj:

x = neighbor

else:

return x

return x

solution = hill\_climbing(objective\_function, -100, 100, 100)

print("Solution: x = %.2f, f(x) = %.2f" % (solution, objective\_function(solution)))

**16. Write a Python program to implement Simulated Annealing algorithm.**

import math

import random

def objective\_function(x, y):

return math.sin(x) \* math.cos(y)

def acceptance\_probability(delta, temperature):

if delta < 0:

return 1.0

else:

return math.exp(-delta / temperature)

def simulated\_annealing(objective\_function, x\_min, x\_max, y\_min, y\_max, max\_iter):

x = random.uniform(x\_min, x\_max)

y = random.uniform(y\_min, y\_max)

current\_obj = objective\_function(x, y)

temperature = 1000.0

cooling\_rate = 0.03

for i in range(max\_iter):

delta\_x = random.uniform(-0.1, 0.1)

delta\_y = random.uniform(-0.1, 0.1)

neighbor\_x = x + delta\_x

neighbor\_y = y + delta\_y

neighbor\_obj = objective\_function(neighbor\_x, neighbor\_y)

delta\_obj = neighbor\_obj - current\_obj

if acceptance\_probability(delta\_obj, temperature) > random.random():

x = neighbor\_x

y = neighbor\_y

current\_obj = neighbor\_obj

temperature \*= 1 - cooling\_rate

return (x, y)

solution = simulated\_annealing(objective\_function, -10, 10, -10, 10, 1000)

print("Solution: x = %.2f, y = %.2f, f(x,y) = %.2f" % (solution[0], solution[1], objective\_function(solution[0], solution[1])))

**17. Write a Python program to implement Genetic Algorithm (GA).**

import random

def fitness\_function(chromosome):

return sum(chromosome)

def genetic\_algorithm(population\_size, chromosome\_length, fitness\_function, mutation\_probability=0.1):

population = [[random.randint(0, 1) for j in range(chromosome\_length)] for i in range(population\_size)]

while True:

fitness\_values = [fitness\_function(chromosome) for chromosome in population]

parents = [population[i] for i in range(population\_size) if fitness\_values[i] == max(fitness\_values)]

next\_generation = []

while len(next\_generation) < population\_size:

parent1 = random.choice(parents)

parent2 = random.choice(parents)

crossover\_point = random.randint(1, chromosome\_length - 1)

child = parent1[:crossover\_point] + parent2[crossover\_point:]

for i in range(chromosome\_length):

if random.random() < mutation\_probability:

child[i] = 1 - child[i]

next\_generation.append(child)

population = next\_generation

if max(fitness\_values) == chromosome\_length:

return parents[0]

chromosome\_length = 10

population\_size = 50

solution = genetic\_algorithm(population\_size, chromosome\_length, fitness\_function)

print("Solution found:", solution)

**18. Write a Python program to implement Missionaries and Cannibals problem.**

from queue import Queue

INITIAL\_STATE = (3, 3, 1)

GOAL\_STATE = (0, 0, 0)

ACTIONS = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]

def is\_valid(state):

if state[0] < 0 or state[1] < 0 or state[0] > 3 or state[1] > 3:

return False

if state[0] > 0 and state[0] < state[1]:

return False

if state[0] < 3 and state[0] > state[1]:

return False

return True

def successors(state):

boat = state[2]

states = []

for action in ACTIONS:

if boat == 1:

new\_state = (state[0] - action[0], state[1] - action[1], 0)

else:

new\_state = (state[0] + action[0], state[1] + action[1], 1)

if is\_valid(new\_state):

states.append(new\_state)

return states

def bfs(initial\_state, goal\_state):

frontier = Queue()

frontier.put(initial\_state)

visited = set()

visited.add(initial\_state)

parent = {}

while not frontier.empty():

state = frontier.get()

if state == goal\_state:

path = []

while state != initial\_state:

path.append(state)

state = parent[state]

path.append(initial\_state)

path.reverse()

return path

for next\_state in successors(state):

if next\_state not in visited:

frontier.put(next\_state)

visited.add(next\_state)

parent[next\_state] = state

return None

path = bfs(INITIAL\_STATE, GOAL\_STATE)

if path is None:

print("No solution found")

else:

print("Optimal path:")

for state in path:

print(state)

**Hangman Game Problem : -**

import time

name = input("What is your name? ")

print ("Hello, " + name, "Time to play hangman!")

time.sleep(1)

print ("Start guessing...")

time.sleep(0.5)

word = ("secret")

guesses = ''

turns = 10

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 ("You won")

break

guess = input("guess a character:")

guesses += guess

if guess not in word:

turns -= 1

print ("Wrong")

print ("You have", + turns, 'more guesses' )

if turns == 0:

print ("You Lose" )

**N Queen Problem : -**

global N

N = 4

def printSolution(board):

for i in range(N):

for j in range(N):

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

print()

def isSafe(board, row, col):

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, N, 1), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solveNQUtil(board, col):

if col >= N:

return True

for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

if solveNQUtil(board, col + 1) == True:

return True

board[i][col] = 0

return False

def solveNQ():

board = [ [0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0]

]

if solveNQUtil(board, 0) == False:

print ("Solution does not exist")

return False

printSolution(board)

return True

solveNQ()

**A\* Algorithm :-**

from copy import deepcopy

import numpy as np

import time

def bestsolution(state):

bestsol = np.array([], int).reshape(-1, 9)

count = len(state) - 1

while count != -1:

bestsol = np.insert(bestsol, 0, state[count]['puzzle'], 0)

count = (state[count]['parent'])

return bestsol.reshape(-1, 3, 3)

def all(checkarray):

set=[]

for it in set:

for checkarray in it:

return 1

else:

return 0

def misplaced\_tiles(puzzle,goal):

mscost = np.sum(puzzle != goal) - 1

return mscost if mscost > 0 else 0

def coordinates(puzzle):

pos = np.array(range(9))

for p, q in enumerate(puzzle):

pos[q] = p

return pos

def evaluvate\_misplaced(puzzle, goal):

steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8], 3),('left', [0, 3, 6], -1),('right', [2, 5, 8], 1)],

dtype = [('move', str, 1),('position', list),('head', int)])

dtstate = [('puzzle', list),('parent', int),('gn', int),('hn', int)]

costg = coordinates(goal)

parent = -1

gn = 0

hn = misplaced\_tiles(coordinates(puzzle), costg)

state = np.array([(puzzle, parent, gn, hn)], dtstate)

dtpriority = [('position', int),('fn', int)]

priority = np.array([(0, hn)], dtpriority)

while 1:

priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])

position, fn = priority[0]

priority = np.delete(priority, 0, 0)

puzzle, parent, gn, hn = state[position]

puzzle = np.array(puzzle)

blank = int(np.where(puzzle == 0)[0])

gn = gn + 1

c = 1

start\_time = time.time()

for s in steps:

c = c + 1

if blank not in s['position']:

openstates = deepcopy(puzzle)

openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']], openstates[blank]

if ~(np.all(list(state['puzzle']) == openstates, 1)).any():

end\_time = time.time()

if (( end\_time - start\_time ) > 2):

print(" The 8 puzzle is unsolvable \n")

break

hn = misplaced\_tiles(coordinates(openstates), costg)

# generate and add new state in the list

q = np.array([(openstates, position, gn, hn)], dtstate)

state = np.append(state, q, 0)

# f(n) is the sum of cost to reach node

fn = gn + hn

q = np.array([(len(state) - 1, fn)], dtpriority)

priority = np.append(priority, q, 0)

if np.array\_equal(openstates, goal):

print(' The 8 puzzle is solvable \n')

return state, len(priority)

return state, len(priority)

puzzle = []

puzzle.append(2)

puzzle.append(8)

puzzle.append(3)

puzzle.append(1)

puzzle.append(6)

puzzle.append(4)

puzzle.append(7)

puzzle.append(0)

puzzle.append(5)

goal = []

goal.append(1)

goal.append(2)

goal.append(3)

goal.append(8)

goal.append(0)

goal.append(4)

goal.append(7)

goal.append(6)

goal.append(5)

state, visited = evaluvate\_misplaced(puzzle, goal)

bestpath = bestsolution(state)

print(str(bestpath).replace('[', ' ').replace(']', ''))

totalmoves = len(bestpath) - 1

print('\nSteps to reach goal:',totalmoves)

visit = len(state) - visited

print('Total nodes visited: ',visit, "\n")