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

Program:

```
# Program to print the multiplication table of a given number

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

# Loop to print the table
for i in range(1, 11):
   print(num, "x", i, "=", num * i)
```

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

Program:

```
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.

Program:

```
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.

The Tower of Hanoi is a mathematical problem which consists of three rods and multiple (say, *n*) disks. Initially, all the disks are placed on one rod, one over the other in ascending order of size similar to a cone-shaped tower.

The objective of this problem is to move the stack of disks from the initial rod to another rod, following these rules:

- A disk cannot be placed on top of a smaller disk
- No disk can be placed on top of the smaller disk.

The goal is to move all the disks from the leftmost rod to the rightmost rod. To move *n* disks from one rod to another, $2^n - 1$ steps are required. So, to move 3 disks from starting the rod to the ending rod, a total of 7 steps are required.

The idea is to use the auxiliary rod to reach the destination using recursion. The steps to follow are −

Step 1 − Move *n*-1 disks from source rod to auxiliary rod
Step 2 − Move n^{th} disk from source rod to destination rod
Step 3 − Move *n*-1 disks from auxiliary rod to destination rod

Program:

```
# Recursive Python function to solve tower of hanoi

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)

# Driver code

n = 4

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

# A, C, B are the name of rods
```

5. Write a python program to implement simple Chatbot.

Program:

```
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.

Program:

```
sentence = input("Enter a sentence: ")

words = sentence.split()
words.sort()

print("The sorted words are:")
```

```
for word in words:
    print(word)
```

Explanation:

In this program, the input() function is used to prompt the user to enter a sentence. The split() method is then used to split the sentence into individual words and store them in a list called words. The sort() function is used to sort the words in alphabetical order.

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

Program:

```
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.

Program:

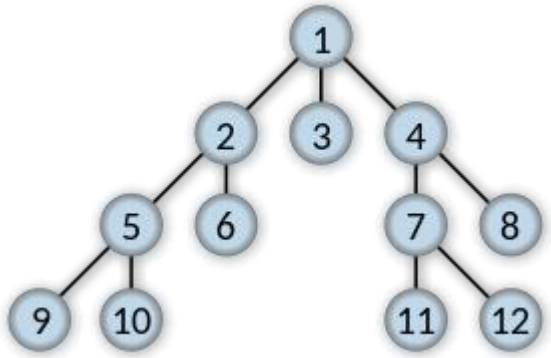
```
def calculate(operation, num1, num2):
    if operation == '+':
        return num1 + num2
    elif operation == '-':
        return num1 - num2
    elif operation == '*':
        return num1 * num2
    elif operation == '/':
        return num1 / num2
    else:
        return "Invalid operator"

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.

Breadth-first search (**BFS**) is an algorithm for searching a tree or graph data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.

Example:



Program:

```
# Python Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s.
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation


class Graph:

        # Constructor
        def __init__(self):
```

```
        # default dictionary to store graph
        self.graph = defaultdict(list)

# function to add an edge to graph
def addEdge(self, u, v):
        self.graph[u].append(v)

# Function to print a BFS of graph
def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (max(self.graph) + 1)

        # Create a queue for BFS
        queue = []

        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

                # Dequeue a vertex from
                # queue and print it
                s = queue.pop(0)
                print(s, end=" ")

                # Get all adjacent vertices of the
                # dequeued vertex s. If an adjacent
                # has not been visited, then mark it
                # visited and enqueue it
                for i in self.graph[s]:
```

```
                    if visited[i] == False:
                        queue.append(i)
                        visited[i] = True

# Driver code

# Create a graph
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)
```

Explanation: **Defaultdict** is a container like dictionaries present in the module **collections**. Defaultdict is a sub-class of the dictionary class that returns a dictionary-like object. The functionality of both dictionaries and defaultdict are almost same except for the fact that defaultdict never raises a KeyError. It provides a default value for the key that does not exist.

The task of constructor __init__(self) is to initialize (assign values) to the data members of the class when an object of the class is created. The keyword **self** represents the instance of a class and binds the attributes with the given arguments.

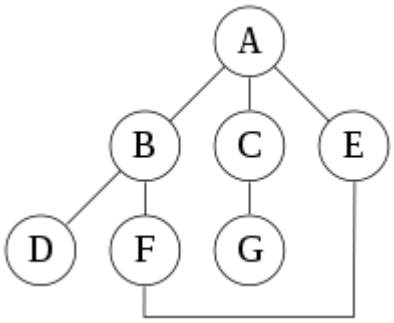
10. Write a python program to implement Depth First Search Traversal.

Depth-first search (**DFS**) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before

backtracking. Extra memory, usually a stack, is needed to keep track of the nodes discovered so far along a specified branch which helps in backtracking of the graph.

Example:

For the following graph:



A depth-first search starting at the node A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G.

Program:

```
# Python program to print DFS traversal
# from a given graph
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation


class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)
```

```
        # function to add an edge to graph
        def addEdge(self, u, v):
                self.graph[u].append(v)

        # A function used by DFS
        def DFSUtil(self, v, visited):

                # Mark the current node as visited
                # and print it
                visited.add(v)
                print(v, end=' ')

                # Recur for all the vertices
                # adjacent to this vertex
                for neighbour in self.graph[v]:
                        if neighbour not in visited:
                                self.DFSUtil(neighbour, visited)

        # The function to do DFS traversal. It uses
        # recursive DFSUtil()
        def DFS(self, v):

                # Create a set to store visited vertices
                visited = set()

                # Call the recursive helper function
                # to print DFS traversal
                self.DFSUtil(v, visited)

# Driver's code

# Create a graph
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)")

# Function call

g.DFS(2)
```

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

The problem statement for Water Jug Problem:

You are given 2 jugs with the capacity *'m'* and *'n'* respectively. Neither has any measuring marker on it. Initially, they are given empty. There is an unlimited supply of water. You can either fill the whole jug or a quantity that is less than the given capacity of jugs. Now, you are also given a third positive integer *'d'*. Using the 2 given jugs, you need to come up with a solution to have *'d'* amount of water in them and return the number of steps you took to reach that capacity.

Example:

Input: 3, 5, 4

Output: 7 (No. of steps)

Explanation: The following steps are taken:

1. Fill the 5-liter jug completely.
2. Transfer 3 liters from a 5-liter jug to a 3-liter jug.
3. Empty the 3-liter capacity jug.
4. Transfer the remaining 2 liters from a 5-liter jug to a 3-liter jug.
5. Now, fill the 5-liter jug fully.
6. Pour 1 liter from a 5-liter jug into a 3-liter jug.
7. Then we have 4 liters in the first jug, now empty the 2nd jug.

Program:

```
# This function is used to initialize the
# dictionary elements with a default value.

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 = 3, 5, 4
# 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)
```

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

Program:

```
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()
```

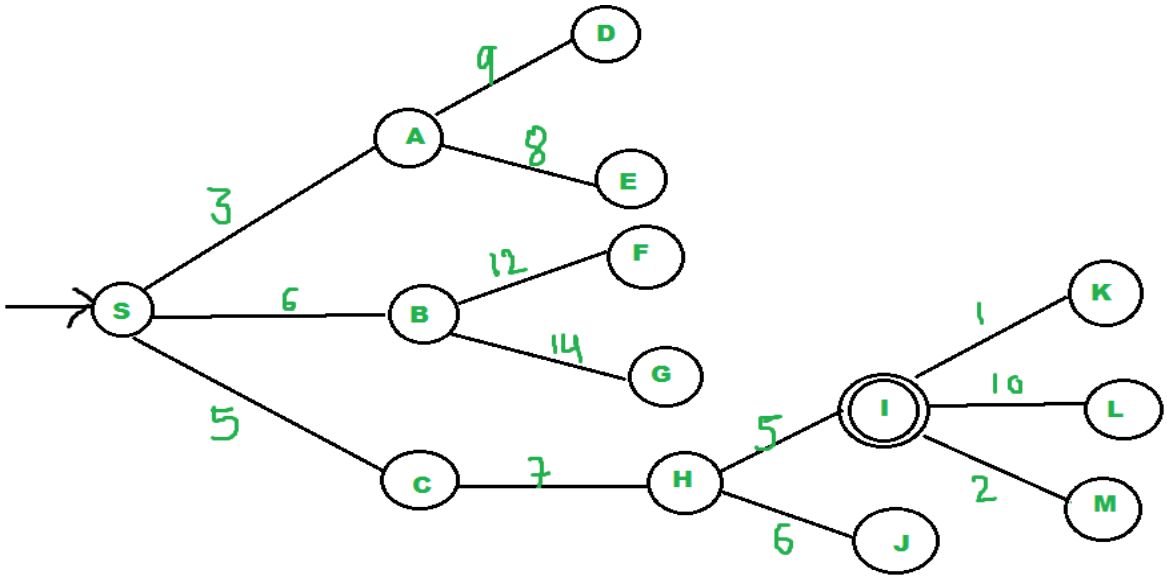
```
        print("That space is already taken!")

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 (Greedy Search).

Best-First Search is a search algorithm that explores a graph by selecting the most promising node at each step, based on an estimate of the cost to reach the goal. The algorithm maintains a *priority queue* of the nodes to be explored, sorted by their estimated cost to the goal. The node with the lowest estimated cost is selected first, and its neighbors are added to the priority queue, with their estimated costs. The algorithm continues this process until either the goal is found or the priority queue is empty.

Let us consider the following graph:



Steps are as follows:

We start from source “S” and search for goal “I” using given costs and Best First search.

- Priority Queue (PQ) initially contains S.
- We remove S from PQ and process unvisited neighbors of S to PQ.
- PQ now contains {A, C, B} (C is put before B because C has lesser cost)

- We remove A from PQ and process unvisited neighbors of A to PQ.
- PQ now contains {C, B, E, D}.
- We remove C from PQ and process unvisited neighbors of C to PQ.
- PQ now contains {B, H, E, D}.
- We remove B from PQ and process unvisited neighbors of B to PQ.
- PQ now contains {H, E, D, F, G}
- We remove H from PQ.
- Since our goal “I” is a neighbor of H, we return.

Program:

```
# Define a graph using a dictionary
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': []
}

# Define the Best-First Search algorithm
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

# Example usage
start = 'A'
goal = 'J'
found = best_first_search(graph, start, goal)
if found:
    print("Goal reached!")
else:
    print("Goal not found.")
```

Explanation:

In this program, the Best-First Search algorithm is implemented using a priority queue to store the nodes to be explored, with the priority given by the estimated cost to the goal node. The algorithm starts from the *start* node and explores the neighbors of each node in order of increasing cost to the goal node until the *goal* node is found or there are no more nodes to explore. If the *goal* node is found, the algorithm returns *True*, otherwise it returns *False*.

The graph is represented as a dictionary, where each key represents a node and the corresponding value is a list of pairs representing the neighbors of the node and the cost to reach each neighbor. The algorithm uses a set to keep track of the visited nodes and a list to implement the priority queue.

14. Write a Python program to implement A* search algorithm.

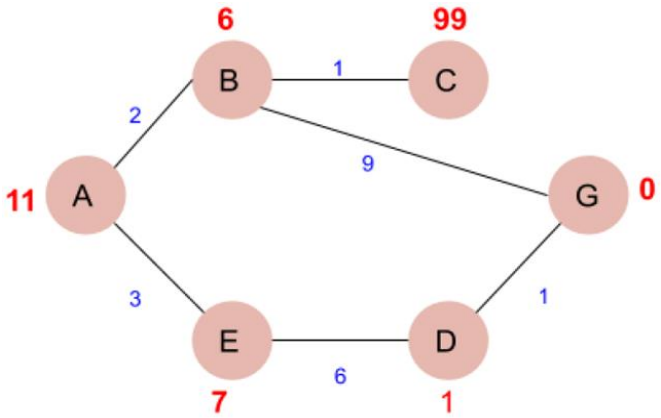
The A* search algorithm is the most commonly known form of best-first search. A lot of games and web-based maps use this algorithm for finding the shortest path efficiently. A* algorithm extends the path that minimizes the following function-

$$\mathbf{f(n) = g(n) + h(n)}$$

where,
'n' is the last node on the path,
g(n) is the cost of the path from start node to node 'n', and
h(n) is a heuristic function that estimates cost of the cheapest path from node 'n' to the goal node.

The A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster.

Let us take the following graph as an example:



Numbers written on edges represent the distance between nodes. Numbers written on nodes represent the heuristic value.
Given the graph, find the cost-effective path from A to G. That is A is the source node and G is the goal node.
Now from A, we can go to point B or E, so we compute f(x) for each of them,
A → B = g(B) + h(B) = 2 + 6 = 8
A → E = g(E) + h(E) = 3 + 7 = 10
Since the cost for A → B is less, we move forward with this path and compute the f(x) for the children nodes of B.
Now from B, we can go to point C or G, so we compute f(x) for each of them,
A → B → C = (2 + 1) + 99= 102
A → B → G = (2 + 9) + 0 = 11
Here the path A → B → G has the least cost but it is still more than the cost of A → E, thus we explore this path further.
Now from E, we can go to point D, so we compute f(x),
A → E → D = (3 + 6) + 1 = 10
Comparing the cost of A → E → D with all the paths we got so far and as this cost is least of all we move forward with this path.
Now compute the f(x) for the children of D
A → E → D → G = (3 + 6 + 1) +0 = 10

Now comparing all the paths that lead us to the goal, we conclude that **A → E → D → G** is the most cost-effective path to get from A to G.

Program:

```
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 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')
```

15. Write a Program to Implement Travelling Salesman Problem using Python.

The travelling salesman problem abides by a salesman and a set of cities. The salesman has to visit every one of the cities starting from a certain one (e.g., the hometown) and to return to the same city. The challenge of the problem is that the travelling salesman needs to minimize the total length of the trip.
Suppose the cities are x_1 x_2..... x_n where cost c_{ij} denotes the cost of travelling from city x_i to x_j. The travelling salesperson problem is to find a route starting and ending at x_1 that will take in all cities with the minimum cost.
The problem is a famous NP-hard problem. There is no polynomial-time based solution for this problem. Different solutions for the travelling salesman problem include are naive solution, dynamic programming, branch and bound, or heuristics such as nearest-neighbor or genetic algorithms.

Naive Solution:

1. Consider city 1 as the starting and ending point.
2. Generate all (n-1)! permutations of cities.
3. Calculate the cost of every permutation and keep track of the minimum cost permutation.
4. Return the permutation with minimum cost.

Time Complexity: Θ(n!)

Program:

```
# TSP using naive approach
from sys import maxsize
from itertools import permutations
V = 4

def travellingSalesmanProblem(graph, s):
        # store all vertex apart from source vertex
      vertex = []
      for i in range(V):
           if i != s:
                vertex.append(i)

        # store minimum weighted Cycle
      min_path = maxsize
      next_permutation=permutations(vertex)
      for i in next_permutation:
          # store current Path weight(cost)
          current_pathweight = 0
```

```
            # compute current path weight
            k = s
            for j in i:
                  current_pathweight += graph[k][j]
                  k = j
            current_pathweight += graph[k][s]
            # update minimum
            min_path = min(min_path, current_pathweight)

      return min_path


# Driver Code
if __name__ == "__main__":
         # matrix representation of graph
      graph = [[0, 10, 15, 20], [10, 0, 35, 25],
                  [15, 35, 0, 30], [20, 25, 30, 0]]
      s = 0
      print(travellingSalesmanProblem(graph, s))
```

16. Write a Python program for Hill climbing algorithm.

Hill climbing is a simple optimization algorithm in the field of artificial intelligence that is used to find the best solution (i.e., maximum or minimum) to a given problem by iteratively improving a candidate solution. It is a local search algorithm that starts from an initial solution and iteratively improves the solution by moving to the neighboring solutions that have a better objective value until a local optimum is reached. The algorithm is called "hill climbing" because it is analogous to climbing a hill, where the goal is to reach the highest peak (i.e., global optimum) by taking small steps in the direction of the steepest slope.

The hill climbing algorithm is widely used in various AI applications such as optimization problems, image processing, machine learning, and robotics. However, one of the main limitations of hill climbing is that it may get stuck in a local optimum, which means that it finds the best solution in a limited area of the search space, but not the global optimum. To overcome this limitation, various extensions of the basic hill climbing algorithm, such as simulated annealing, genetic algorithms, and tabu search, have been proposed.

Program:

import random

Define the objective function

```
def objective_function(x):
    return x**2

# Define the hill climbing algorithm
def hill_climbing(objective_function, x_min, x_max, max_iter):
    # Generate a random initial solution
    x = random.uniform(x_min, x_max)

    # Iterate until max_iter is reached or there is no improvement
    for i in range(max_iter):
        # Calculate the objective value for the current solution
        current_obj = objective_function(x)

        # Generate a random neighbor solution
        delta = random.uniform(-0.1, 0.1)
        neighbor = x + delta

        # Calculate the objective value for the neighbor solution
        neighbor_obj = objective_function(neighbor)

        # Check if the neighbor solution is better than the current solution
        if neighbor_obj < current_obj:
            # If so, update the current solution
            x = neighbor
        else:
            # If not, terminate the algorithm and return the current solution
            return x

    # Return the current solution if max_iter is reached
    return x

# Test the algorithm
solution = hill_climbing(objective_function, -100, 100, 100)
print("Solution: x = %.2f, f(x) = %.2f" % (solution, objective_function(solution)))
```

Explanation:

In this example, the objective function is simply *x^2*, and the algorithm tries to find the minimum of this function within the range [-100, 100]. The *hill_climbing* function takes as input the objective function, the minimum and maximum values for *x*, and the maximum number of iterations to perform. It then iterates until either the maximum number of iterations is reached or there is no improvement in the objective value. In each iteration, the function generates a random neighbor solution by adding a random delta value to the current solution. If the neighbor solution has a lower objective value than the current solution, it is accepted as the new current solution. Otherwise, the algorithm terminates and returns the current solution.

Finally, the function returns the solution with the minimum objective value found during the iterations.

17. Write a Python program to implement Simulated Annealing algorithm.

Simulated annealing is a meta-heuristic algorithm in the field of artificial intelligence that is used to find a near-optimal solution to a given problem. It is inspired by the physical process of annealing in which a metal is heated and slowly cooled to reduce its defects and increase its strength. Similarly, simulated annealing starts with a high temperature (i.e., high probability of accepting a worse solution) and gradually reduces the temperature (i.e., low probability of accepting a worse solution) over time to converge to a near-optimal solution.
Simulated annealing is a stochastic optimization algorithm that iteratively generates random solutions by making small modifications to the current solution. If the new solution is better than the current solution, it is accepted as the new current solution. However, if the new solution is worse than the current solution, it may still be accepted with a certain probability that depends on the temperature and the difference in objective function value between the new and current solutions. This probability is defined by the acceptance probability function, which is designed to gradually decrease the acceptance probability as the temperature decreases.
The simulated annealing algorithm is often used to solve combinatorial optimization problems, such as the traveling salesman problem, the quadratic assignment problem, and the job shop scheduling problem. It has been shown to be effective in finding near-optimal solutions for many real-world problems, and can often outperform other optimization algorithms, such as genetic algorithms and local search algorithms.

Program:

```
import math
import random

# Define the objective function
def objective_function(x, y):
    return math.sin(x) * math.cos(y)
```

```
# Define the acceptance probability function
def acceptance_probability(delta, temperature):
    if delta < 0:
        return 1.0
    else:
        return math.exp(-delta / temperature)

# Define the simulated annealing algorithm
def simulated_annealing(objective_function, x_min, x_max, y_min, y_max, max_iter):
    # Generate a random initial solution
    x = random.uniform(x_min, x_max)
    y = random.uniform(y_min, y_max)
    current_obj = objective_function(x, y)

    # Define the temperature schedule
    temperature = 1000.0
    cooling_rate = 0.03

    # Iterate until max_iter is reached or the temperature is too low
    for i in range(max_iter):
        # Generate a random neighbor solution
        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

        # Calculate the objective value for the neighbor solution
        neighbor_obj = objective_function(neighbor_x, neighbor_y)

        # Calculate the difference in objective value between the current and neighbor solutions
        delta_obj = neighbor_obj - current_obj

        # Determine whether to accept the neighbor solution
        if acceptance_probability(delta_obj, temperature) > random.random():
```

```
            x = neighbor_x
            y = neighbor_y
            current_obj = neighbor_obj

        # Decrease the temperature
        temperature *= 1 - cooling_rate

    # Return the final solution
    return (x, y)

# Test the algorithm
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])))
```

Explanation:

In this example, the objective function is *sin(x) * cos(y)*, and the algorithm tries to find the maximum of this function within the range [-10, 10] for both *x* and *y*. The *simulated_annealing* function takes as input the objective function, the minimum and maximum values for *x* and *y*, and the maximum number of iterations to perform. It then iterates until either the maximum number of iterations is reached or the temperature is too low. In each iteration, the function generates a random neighbor solution by adding a random delta value to the current solution. If the neighbor solution has a higher objective value than the current solution, it is accepted as the new current solution. Otherwise, the neighbor solution is accepted with a probability that depends on the difference in objective value between the current and neighbor solutions, and the current temperature. The temperature decreases according to a predefined cooling rate at each iteration. Finally, the function returns the solution with the maximum objective value found during the iterations.

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

Genetic algorithm is a meta-heuristic optimization algorithm that is based on the process of natural selection and genetics. It is inspired by the biological process of evolution and uses the principles of genetics to evolve a population of candidate solutions to a problem towards

an optimal or near-optimal solution. The genetic algorithm mimics the natural selection process by evaluating the fitness of each candidate solution in the population, and selecting the fittest individuals to generate a new population through crossover and mutation.

In a genetic algorithm, each candidate solution is represented as a chromosome, which is composed of genes that encode the solution variables. The fitness function evaluates the quality of each chromosome, which is used to determine the selection probability of each chromosome in the population. The fittest chromosomes are selected to generate a new population through crossover, which involves swapping genes between two parent chromosomes, and mutation, which involves randomly changing genes in a chromosome.

The genetic algorithm iteratively generates new populations by selecting the fittest chromosomes from the previous population and applying crossover and mutation to generate new offspring. This process continues until a satisfactory solution is found or a predetermined stopping criterion is met.

Genetic algorithms are commonly used to solve optimization problems, such as scheduling, resource allocation, and machine learning. They are particularly useful for problems where the search space is large and the objective function is nonlinear or multimodal. However, genetic algorithms do not guarantee finding the optimal solution, and the quality of the solution depends on the quality of the fitness function, the size of the population, and the genetic operators used.

Program:

```
import random

# Define the fitness function
def fitness_function(chromosome):
    return sum(chromosome)

# Define the genetic algorithm
def genetic_algorithm(population_size, chromosome_length, fitness_function, mutation_probability=0.1):
    # Initialize the population with random chromosomes
    population = [[random.randint(0, 1) for j in range(chromosome_length)] for i in range(population_size)]
```

```
    # Loop until a satisfactory solution is found
    while True:
        # Evaluate the fitness of each chromosome in the population
        fitness_values = [fitness_function(chromosome) for chromosome in population]

        # Select the fittest chromosomes to be the parents of the next generation
        parents = [population[i] for i in range(population_size) if fitness_values[i] == max(fitness_values)]

        # Generate the next generation by applying crossover and mutation to the parents
        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)

        # Update the population with the next generation
        population = next_generation

        # Check if a satisfactory solution has been found
        if max(fitness_values) == chromosome_length:
            return parents[0]

# Example usage
chromosome_length = 10
population_size = 50
solution = genetic_algorithm(population_size, chromosome_length, fitness_function)
print("Solution found:", solution)
```

Explanation:

In this example, the genetic algorithm uses binary chromosomes of length *chromosome_length* to represent a potential solution to the problem. The fitness function calculates the fitness of each chromosome as the sum of its elements. The genetic algorithm iteratively generates a new population by selecting the fittest chromosomes from the previous generation as parents, and applying crossover and mutation to generate new offspring. The algorithm terminates when a satisfactory solution is found, i.e., a chromosome with maximum fitness value equal to the chromosome length.

19. Write a Python program to implement Missionaries and Cannibals problem.

The Missionaries and Cannibals problem is a classic problem in Artificial Intelligence that involves three missionaries and three cannibals who need to cross a river using a boat that can carry at most two people. The problem is to find a way to get all six people across the river without ever having a situation where the cannibals outnumber the missionaries on either side of the river, as the cannibals would then eat the missionaries.

The problem can be formulated as a search problem, where each state is represented by the number of missionaries and cannibals on either side of the river and the position of the boat. The initial state is (3, 3, 1), meaning that there are three missionaries and three cannibals on the left bank of the river, the boat is on the left bank, and the right bank is empty. The goal state is (0, 0, 0), meaning that all six people have successfully crossed the river.

Program (Using Breadth First Search):

```
from queue import Queue

# Define the initial and goal states
INITIAL_STATE = (3, 3, 1)
GOAL_STATE = (0, 0, 0)

# Define the actions that can be taken in each state
ACTIONS = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]

# Define a function to check if a state is valid
```

```
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

# Define a function to get the possible successor states of a given state
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

# Define the Breadth-First Search algorithm
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

# Find the optimal path from the initial state to the goal state
path = bfs(INITIAL_STATE, GOAL_STATE)

# Print the optimal path
if path is None:
  print("No solution found")
else:
  print("Optimal path:")
  for state in path:
    print(state)
```

20. Write a Python program to implement Hangman game.

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 match with the alphabets in the original word. If the user given alphabet matches with the alphabet from the original word then user must guess the remaining alphabets. Once the user successfully found the word he will be declared as winner otherwise user loses the game.

Program:

```
import random

# List of words to choose from
word_list = ["python", "java", "ruby", "javascript", "php", "csharp", "html", "css"]
```

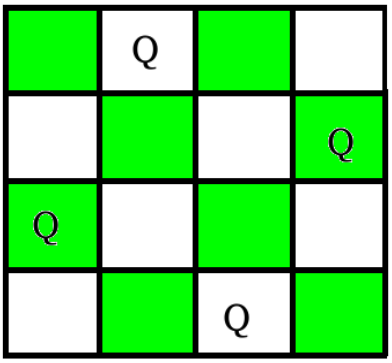
```
# Select a random word from the list
word = random.choice(word_list)

# Convert the word to a list of characters
word_letters = list(word)

# Create a list to store guessed letters
guessed_letters = []

# Create a variable to keep track of the number of incorrect guesses
num_guesses = 0

# Create a string to represent the current state of the word
display_word = "_" * len(word)

# Loop until the player has guessed the word or run out of guesses
while num_guesses < 6 and "_" in display_word:
    # Print the current state of the game
    print(f"Word: {display_word}")
    print(f"Guessed letters: {guessed_letters}")
    print(f"Number of incorrect guesses: {num_guesses}")

    # Ask the player to guess a letter
    guess = input("Guess a letter: ").lower()

    # Check if the letter has already been guessed
    if guess in guessed_letters:
        print("You already guessed that letter. Try again.")
        continue

    # Add the letter to the list of guessed letters
    guessed_letters.append(guess)

    # Check if the letter is in the word
    if guess in word_letters:
        # Update the display word with the guessed letter
        for i in range(len(word_letters)):
            if word_letters[i] == guess:
                display_word = display_word[:i] + guess + display_word[i+1:]
    else:
        # Increment the number of incorrect guesses
        num_guesses += 1
        print("Incorrect guess.")

# Check if the player has won or lost
if "_" not in display_word:
    print(f"Congratulations, you guessed the word! The word was {word}.")
else:
    print(f"Sorry, you ran out of guesses. The word was {word}.")
```

Explanation:

This program uses a list of words to choose from, selects a random word, and prompts the player to guess letters until s/he has guessed the word or is run out of guesses. The current state of the game is displayed after each guess. The player has a maximum of six incorrect guesses before s/he loses the game. If the player guesses all the letters correctly, s/he wins the game.

21. Write a Python program to implement N-Queen problem.

The N-queen problem is a problem in which we figure out a way to put N queens on an N×N chessboard in such a way that no queen should attack the other. For basic info about the queen in a chess game, you should know that a queen can move in any direction (vertically, horizontally, and diagonally) and to any number of places. In the figure below you can see how to place 4 queens on a 4×4 chessboard.



Program:

```
# Taking number of queens as input from user
print ("Enter the number of queens")
N = int(input())

# here we create a chessboard
# NxN matrix with all elements set to 0
board = [[0]*N for _ in range(N)]

def attack(i, j):
    #checking vertically and horizontally
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:  # 1 means a queen is placed in the position
            return True
    #checking diagonally
    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_queens(n):
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            if (not(attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                if N_queens(n-1)==True:
                    return True
                board[i][j] = 0

    return False

N_queens(N)
for i in board:
    print (i)
```