1.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
```

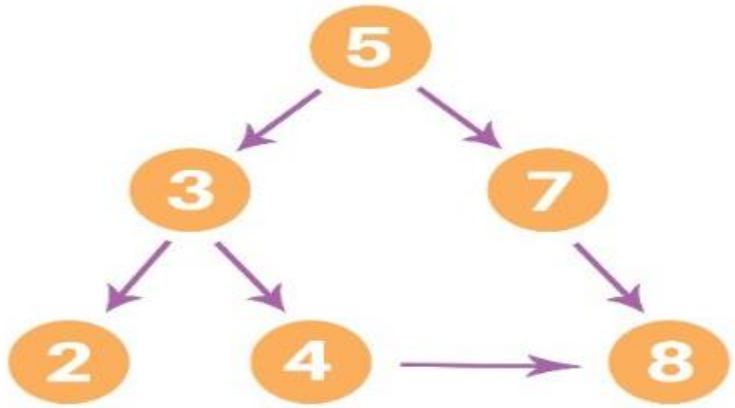
```
if (u[0] == target or u[1] == target):
        isSolvable = True

        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)
```

2.BFS

```
from collections import deque

# Define a graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

def bfs(graph, start):
    visited = set()
    queue = deque()
    queue.append(start)

    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            print(vertex, end=' ')
            visited.add(vertex)
            queue.extend(neighbor for neighbor in graph[vertex] if neighbor not in visited)

# Example usage:
start_vertex = 'A'
print("Breadth-First Search starting from vertex A:")
bfs(graph, start_vertex)
```

or




```
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
```

BFS using a Recursive Method

```
def dfs_recursive(graph, source,path = []):

    if source not in path:
        path.append(source)

        if source not in graph:
            # leaf node, backtrack
            return path

        for neighbour in graph[source]:

            path = dfs_recursive(graph, neighbour, path)


    return path

graph = {"A":["B","C","D"],
   "B":["E"],
   "C":["G","F"],
   "D":["H"],
   "E":["I"],
   "F":["J"],
   "G":["K"]}
dfs_element = dfs_recursive(graph, "A")
print(dfs_element)
```

3.DFS

```
# 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')
```

DFS using a Recursive Method

```
def dfs_recursive(graph, source,path = []):

      if source not in path:
         path.append(source)

         if source not in graph:
            # leaf node, backtrack
            return path

         for neighbour in graph[source]:

            path = dfs_recursive(graph, neighbour, path)


      return path

graph = {"A":["B","C","D"],
   "B":["E"],
   "C":["G","F"],
   "D":["H"],
   "E":["I"],
   "F":["J"],
   "G":["K"]}
```

```
dfs_element = dfs_recursive(graph, "A")
print(dfs_element)
```

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
```

Write a program for Multiplication table.

```
#Itration program in python

multiplicand = int(input('Enter the multiplicant       : '))
multiplier = int(input('Enter the maximum multiplier : '))

i=0
while i<=multiplier :
 print(f"{multiplicand}*{i}= {multiplicand * i}")
 i += 1
```

```
Enter the multiplicant       : 5
Enter the maximum multiplier : 10
5*0= 0
5*1= 5
5*2= 10
5*3= 15
5*4= 20
5*5= 25
5*6= 30
5*7= 35
5*8= 40
5*9= 45
5*10= 50
```

What is a Membership Operator?

Membership operators in Python are operators used to test whether a value exists in a sequence, such as a list, tuple, or string. The membership operators available in Python are,

- **in:** The in operator returns True if the value is found in the sequence.
- **not in:** The not in operator returns True if the value is not found in the sequence

```
# membership operator in Python
# using "in" operator

fruits = ["apple", "banana", "cherry"]
if "banana" in fruits:
    print("Yes, banana is a fruit!")

# using "not in" operator
if "orange" not in fruits:
    print("Yes, orange is not in the list of fruits")
```

```
Yes, banana is a fruit!
Yes, orange is not in the list of fruits
```

Write a python Program for Travelling Salesman Problem in AI.

```
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))
```

Write a python Program for factorial of number

```
import math
def fact(n):
    return(math.factorial(n))

num = int(input("Enter the number:"))
f = fact(num)
print("Factorial of", num, "is", f)
```

or

```
n=int(input("Enter number:"))
fact=1
for i in range(1,n+1,1):
    fact=fact*i
print(n,"!=",fact)
```

Write a python Program for Prime No.

```
Taking the input from User
number = int(input("Enter The Number"))
if number > 1:
    for i in range(2,int(number/2)+1):
        if (number % i == 0):
            print(number, "is not a Prime Number")
            break
    else:
        print(number,"is a Prime number")
# If the number is less than 1 it can't be Prime
else:
    print(number,"is not a Prime number")
```

Program to display the Fibonacci sequence up to n-th term

```
nterms = int(input("How many terms? "))
```

```
# first two terms
n1, n2 = 0, 1
count = 0

# check if the number of terms is valid
if nterms <= 0:
   print("Please enter a positive integer")
# if there is only one term, return n1
elif nterms == 1:
   print("Fibonacci sequence upto",nterms,":")
   print(n1)
# generate fibonacci sequence
else:
   print("Fibonacci sequence:")
   while count < nterms:
      print(n1)
      nth = n1 + n2
      # update values
      n1 = n2
      n2 = nth
      count += 1
```

The Multiplication Table

```
number = int(input ("Enter the number of which the user wants to print the multiplication
table: "))
# We are using "for loop" to iterate the multiplication 10 times
print ("The Multiplication Table of: ", number)
for count in range(1, 11):
   print (number, 'x', count, '=', number * count)
```

Program for a* algorithm

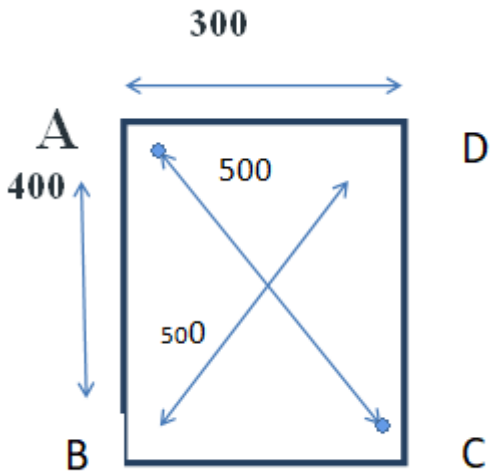
```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}              #store distance from starting node
    parents = {}        # parents contains an adjacency map of all nodes
    #distance of starting node from itself is zero
    g[start_node] = 0
    #start_node is root node i.e it has no parent nodes
    #so start_node is set to its own parent node
```

```
parents[start_node] = start_node
while len(open_set) > 0:
    n = None
    #node with lowest f() is found
    for v in open_set:
        if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
            n = v
    if n == stop_node or Graph_nodes[n] == None:
        pass
    else:
        for (m, weight) in get_neighbors(n):
            #nodes 'm' not in first and last set are added to first
            #n is set its parent
            if m not in open_set and m not in closed_set:
                open_set.add(m)
                parents[m] = n
                g[m] = g[n] + weight
            #for each node m,compare its distance from start i.e g(m) to the
            #from start through n node
            else:
                if g[m] > g[n] + weight:
                    #update g(m)
                    g[m] = g[n] + weight
                    #change parent of m to n
                    parents[m] = n
                    #if m in closed set,remove and add to open
                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)
    if n == None:
        print('Path does not exist!')
        return None

    # if the current node is the stop_node
    # then we begin reconstructin the path from it to the start_node
    if n == stop_node:
        path = []
        while parents[n] != n:
            path.append(n)
            n = parents[n]
```

```
            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
            return path
        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_set.remove(n)
        closed_set.add(n)
    print('Path does not exist!')
    return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
    #for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J': 0
    }
    return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
```

```
        'D': [('B', 2), ('C', 1), ('E', 8)],
        'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
        'F': [('A', 3), ('G', 1), ('H', 7)],
        'G': [('F', 1), ('I', 3)],
        'H': [('F', 7), ('I', 2)],
        'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
    }

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

Program for hill climbing algorithm in AI.




```
import random

def randomSolution(tsp):
    cities = list(range(len(tsp)))
    solution = []

    for i in range(len(tsp)):
        randomCity = cities[random.randint(0, len(cities) - 1)]
        solution.append(randomCity)
        cities.remove(randomCity)

    return solution

def routeLength(tsp, solution):
    routeLength = 0
    for i in range(len(solution)):
        routeLength += tsp[solution[i - 1]][solution[i]]
```

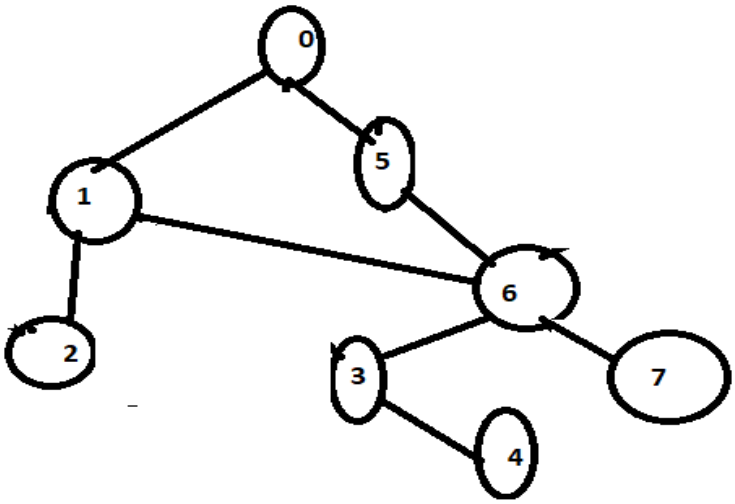
```
    return routeLength

def getNeighbours(solution):
    neighbours = []
    for i in range(len(solution)):
        for j in range(i + 1, len(solution)):
            neighbour = solution.copy()
            neighbour[i] = solution[j]
            neighbour[j] = solution[i]
            neighbours.append(neighbour)
    return neighbours

def getBestNeighbour(tsp, neighbours):
    bestRouteLength = routeLength(tsp, neighbours[0])
    bestNeighbour = neighbours[0]
    for neighbour in neighbours:
        currentRouteLength = routeLength(tsp, neighbour)
        if currentRouteLength < bestRouteLength:
            bestRouteLength = currentRouteLength
            bestNeighbour = neighbour
    return bestNeighbour, bestRouteLength

def hillClimbing(tsp):
    currentSolution = randomSolution(tsp)
    currentRouteLength = routeLength(tsp, currentSolution)
    neighbours = getNeighbours(currentSolution)
    bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

    while bestNeighbourRouteLength < currentRouteLength:
        currentSolution = bestNeighbour
        currentRouteLength = bestNeighbourRouteLength
        neighbours = getNeighbours(currentSolution)
        bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

    return currentSolution, currentRouteLength

def main():
    tsp = [
        [0, 400, 500, 300],
        [400, 0, 300, 500],
        [500, 300, 0, 400],
        [300, 500, 400, 0]
    ]

    print(hillClimbing(tsp))
```

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

output:

We have four cities, each located in the corner of a rectangular shape. The long side of the rectangle is 400 kilometers (or whatever unit you like) long, while the short side is 300. That makes the diagonal 500 kilometers long. It seems obvious that the shortest routes actually travel the sides of this rectangle, which would make the length of the shortest route 2 x 400 + 2 x 300 = 1400 kilometers.

Program for Bidirectional search**.**




```
class Node:
    def __init__(self, val, neighbors=[]):
        self.val = val
        self.neighbors = neighbors
        self.visited_right = False  # whether the node was reached by the BFS that started from source
        self.visited_left = False  # whether the node was reached by the BFS that started from destination
        self.parent_right = None  # used for retrieving the final path from source to the meeting point
        self.parent_left = None  # used for retrieving the final path from the meeting point to destination


# class Queue:
#     pass  # implement it yourself
from collections import deque
```

```
def bidirectional_search(s, t):
    def extract_path(node):
        """return the path when both BFS's have met"""
        node_copy = node
        path = []

        while node:
            path.append(node.val)
            node = node.parent_right

        path.reverse()
        del path[-1]  # because the meeting node appears twice

        while node_copy:
            path.append(node_copy.val)
            node_copy = node_copy.parent_left
        return path

    q = deque([])
    q.append(s)
    q.append(t)
    s.visited_right = True
    t.visited_left = True

    while len(q) > 0:
        n = q.pop()

        if n.visited_left and n.visited_right:  # if the node visited by both BFS's
            return extract_path(n)

        for node in n.neighbors:
            if n.visited_left == True and not node.visited_left:
                node.parent_left = n
                node.visited_left = True
                q.append(node)
            if n.visited_right == True and not node.visited_right:
                node.parent_right = n
                node.visited_right = True
                q.append(node)

    # not found
    return False


n0 = Node(0)
n1 = Node(1)
```

```
n2 = Node(2)
n3 = Node(3)
n4 = Node(4)
n5 = Node(5)
n6 = Node(6)
n7 = Node(7)
n0.neighbors = [n1, n5]
n1.neighbors = [n0, n2, n6]
n2.neighbors = [n1]
n3.neighbors = [n4, n6]
n4.neighbors = [n3]
n5.neighbors = [n0, n6]
n6.neighbors = [n1, n3, n5, n7]
n7.neighbors = [n6]
print(bidirectional_search(n0, n4))
```

Write a Python program to solve tower of Hanoi problem.

```
# 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
```

8 Puzzle problem

```
# 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
```

```
8.  import copy
9.
10. # Importing the heap methods from the python
11. # library for the Priority Queue
12. from heapq import heappush, heappop
13.
14. # This particular var can be changed to transform
15. # the program from 8 puzzle(n=3) into 15
16. # puzzle(n=4) and so on ...
17. n = 3
18.
19. # bottom, left, top, right
20. rows = [ 1, 0, -1, 0 ]
21. cols = [ 0, -1, 0, 1 ]
22.
23. # creating a class for the Priority Queue
24. class priorityQueue:
25.
26.     # Constructor for initializing a
27.     # Priority Queue
28.     def __init__(self):
29.         self.heap = []
30.
31.     # Inserting a new key 'key'
32.     def push(self, key):
33.         heappush(self.heap, key)
34.
35.     # funct to remove the element that is minimum,
36.     # from the Priority Queue
37.     def pop(self):
38.         return heappop(self.heap)
39.
40.     # funct to check if the Queue is empty or not
41.     def empty(self):
42.         if not self.heap:
43.             return True
44.         else:
45.             return False
46.
47. # structure of the node
48. class nodes:
49.
50.     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)
```

Swap two no's

```
P = int( input("Please enter value for P: "))

Q = int( input("Please enter value for Q: "))


# To Swap the values of two variables using Addition and subtraction operator

P = P + Q

Q = P - Q
```

```
P = P - Q


print ("The Value of P after swapping: ", P)
print ("The Value of Q after swapping: ", Q)
```

Python Program to Check Leap Year

```
# Default function to implement conditions to check leap year
def CheckLeap(Year):
  # Checking if the given year is leap year
  if((Year % 400 == 0) or
     (Year % 100 != 0) and
     (Year % 4 == 0)):
    print("Given Year is a leap Year");
  # Else it is not a leap year
  else:
    print ("Given Year is not a leap Year")
# Taking an input year from user
Year = int(input("Enter the number: "))
# Printing result
CheckLeap(Year)
```

Python Program to check Armstrong Number

```
# Python program to check if the number is an Armstrong number or not
```

```
# take input from the user
num = int(input("Enter a number: "))

# initialize sum
sum = 0

# find the sum of the cube of each digit
temp = num
while temp > 0:
   digit = temp % 10
   sum += digit ** 3
   temp //= 10

# display the result
if num == sum:
   print(num,"is an Armstrong number")
else:
   print(num,"is not an Armstrong number")
```

Program to display calendar of the given month and year

```
# importing calendar module
import calendar

yy = 2023  # year
```

```
mm = 10    # month

# To take month and year input from the user
# yy = int(input("Enter year: "))
# mm = int(input("Enter month: "))

# display the calendar
print(calendar.month(yy, mm))
```

Python Program to Make a Simple Calculator

```
# This function adds two numbers
def add(x, y):
    return x + y

# This function subtracts two numbers
def subtract(x, y):
    return x - y

# This function multiplies two numbers
def multiply(x, y):
    return x * y

# This function divides two numbers
def divide(x, y):
    return x / y
```

```
print("Select operation.")
print("1.Add")
print("2.Subtract")
print("3.Multiply")
print("4.Divide")

while True:
    # take input from the user
    choice = input("Enter choice(1/2/3/4): ")

    # check if choice is one of the four options
    if choice in ('1', '2', '3', '4'):
        try:
            num1 = float(input("Enter first number: "))
            num2 = float(input("Enter second number: "))
        except ValueError:
            print("Invalid input. Please enter a number.")
            continue

        if choice == '1':
            print(num1, "+", num2, "=", add(num1, num2))

        elif choice == '2':
```

```
        print(num1, "-", num2, "=", subtract(num1, num2))

    elif choice == '3':
        print(num1, "*", num2, "=", multiply(num1, num2))

    elif choice == '4':
        print(num1, "/", num2, "=", divide(num1, num2))

    # check if user wants another calculation
    # break the while loop if answer is no
    next_calculation = input("Let's do next calculation? (yes/no): ")
    if next_calculation == "no":
      break
else:
    print("Invalid Input")
```

Reverse a Number using a while loop

```
num = 1234
reversed_num = 0

while num != 0:
    digit = num % 10
    reversed_num = reversed_num * 10 + digit
    num //= 10

print("Reversed Number: " + str(reversed_num))
```

Countdown time in Python

```
import time

def countdown(time_sec):
    while time_sec:
        mins, secs = divmod(time_sec, 60)
        timeformat = '{:02d}:{:02d}'.format(mins, secs)
        print(timeformat, end='\r')
        time.sleep(1)
```

```
        time_sec -= 1

    print("stop")

countdown(5)
```