Water-Jug

In [2]:

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

    q = deque()

    q.append((0, 0))

    while (len(q) > 0):


        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])
        q.append([a, u[1]])

        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 )
```

BFS

In [3]:

```
from collections import deque

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 v


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

```
Breadth-First Search starting from vertex A:
A B C D E F
```

BFS using a Recursive Method

In [4]:

```
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)
```

```
['A', 'B', 'E', 'I', 'C', 'G', 'K', 'F', 'J', 'D', 'H']
```

DFS

In [5]:

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

visited = set()

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

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

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

DFS using a Recursive Method

In [6]:

```
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)
```

```
['A', 'B', 'E', 'I', 'C', 'G', 'K', 'F', 'J', 'D', 'H']
```

Simple Chatbot

In [9]:

```
qna={
    "hi":"hey",
    "how are you":"I am fine",
    "what is your name":"my name is ram",
    "how old are you":"I am 10 years old"
}
while True:
    qse=input()
    if(qse=="quit"):
        break
    else:
        print(qna[qse])
```

```
hi
hey
quit
```

Multiplication Table

In [8]:

```
multiplicand=int(input("Enter the multiplicand: "))
multiplier=int(input("Enter the maximum multiplier: "))

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

```
Enter the multiplicand: 5
Enter the maximum multiplier: 6
5*0=0
5*1=5
5*2=10
5*3=15
5*4=20
5*5=25
5*6=30
```

Travelling Salesman Problem

In [11]:

```
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))
```

```
80
```

Factorial

In [12]:

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

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

```
Enter the number:5
Factorial of 5 is 120
```

Prime

In [14]:

```
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")
else:
    print(number,"is not a Prime number")
```

```
Enter The Number3
3 is a Prime number
```

Fibonacci

In [15]:

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

n1, n2 = 0, 1
count = 0
```

```
if nterms <= 0:
   print("Please enter a positive integer")

elif nterms == 1:
   print("Fibonacci sequence upto",nterms,":")
   print(n1)

else:
   print("Fibonacci sequence:")
   while count < nterms:
       print(n1)
       nth = n1 + n2
       n1 = n2
       n2 = nth
       count += 1
```

```
How many terms? 5
Fibonacci sequence:
0
1
1
2
3
```

Multiplication Table

In [16]:

```
number = int(input ("Enter the number of which the user wants to print the multiplic
print ("The Multiplication Table of: ", number)
for count in range(1, 11):
   print (number, 'x', count, '=', number * count)
```

```
Enter the number of which the user wants to print the multiplication table: 5
The Multiplication Table of:  5
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

A* Algorithm

In [18]:

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}
    parents = {}

    g[start_node] = 0

    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None

        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):

                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                else:
                    if g[m] > g[n] + weight:

                        g[m] = g[n] + weight

                        parents[m] = n

                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None


        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

        open_set.remove(n)
        closed_set.add(n)
    print('Path does not exist!')
    return None


def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

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]


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')
```

```
Path found: ['A', 'F', 'G', 'I', 'J']
```

Out[18]:
```
['A', 'F', 'G', 'I', 'J']
```

Hill Climbing Algorithm

In [19]:
```
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()
```

```
([1, 0, 3, 2], 1400)
```

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.

Bidirectional Search

In [22]:

```
class Node:
    def __init__(self, val, neighbors=[]):
        self.val = val
        self.neighbors = neighbors
        self.visited_right = False
        self.visited_left = False
        self.parent_right = None
        self.parent_left = None



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]

        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:
            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)


    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))
```

```
[0, 5, 6, 3, 4]
```

Tower of Hanoi

In [23]:

```
def tower_of_hanoi(disks, source, auxiliary, target):
    if(disks == 1):
        print('Move disk 1 from rod {} to rod {}.'.format(source, target))
        return

    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: '))

tower_of_hanoi(disks, 'A', 'B', 'C')
```

```
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.
```

8 Puzzle

In [25]:
```python
import copy


from heapq import heappush, heappop

n = 3

rows = [ 1, 0, -1, 0 ]
cols = [ 0, -1, 0, 1 ]

class priorityQueue:


    def __init__(self):
        self.heap = []


    def push(self, key):
        heappush(self.heap, key)


    def pop(self):
        return heappop(self.heap)


    def empty(self):
        if not self.heap:
            return True
        else:
            return False


class nodes:

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


        self.parent = parent


        self.mats = mats


        self.empty_tile_posi = empty_tile_posi


        self.costs = costs


        self.levels = levels


    def __lt__(self, nxt):
        return self.costs < nxt.costs

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:


    new_mats = copy.deepcopy(mats)


    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]


    costs = calculateCosts(new_mats, final)

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


def printMatsrix(mats):

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

        print()


def isSafe(x, y):

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


def printPath(root):

    if root == None:
        return

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


def solve(initial, empty_tile_posi, final):


    pq = priorityQueue()


    costs = calculateCosts(initial, final)
    root = nodes(None, initial,
                empty_tile_posi, costs, 0)
```

```python
    pq.push(root)

    while not pq.empty():

        minimum = pq.pop()

        if minimum.costs == 0:

            printPath(minimum)
            return

        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]):

                child = newNodes(minimum.mats,
                                 minimum.empty_tile_posi,
                                 new_tile_posi,
                                 minimum.levels + 1,
                                 minimum, final,)

                pq.push(child)

initial = [ [ 1, 2, 3 ],
            [ 5, 6, 0 ],
            [ 7, 8, 4 ] ]

final = [ [ 1, 2, 3 ],
          [ 5, 8, 6 ],
          [ 0, 7, 4 ] ]

empty_tile_posi = [ 1, 2 ]

solve(initial, empty_tile_posi, final)
```

```
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
```

Swap 2 Numbers

In [26]:
```
P = int( input("Please enter value for P: "))
Q = int( input("Please enter value for Q: "))

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)
```

```
Please enter value for P: 4
Please enter value for Q: 5
The Value of P after swapping:  5
The Value of Q after swapping:  4
```

Leap Year

In [28]:
```
def CheckLeap(Year):

  if((Year % 400 == 0) or
     (Year % 100 != 0) and
     (Year % 4 == 0)):
    print("Given Year is a leap Year");

  else:
    print ("Given Year is not a leap Year")

Year = int(input("Enter the number: "))

CheckLeap(Year)
```

```
Enter the number: 2018
Given Year is not a leap Year
```

Armstrong Number

In [31]:
```
num = int(input("Enter a number: "))


sum = 0


temp = num
while temp > 0:
   digit = temp % 10
   sum += digit ** 3
   temp //= 10


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

```
Enter a number: 153
153 is an Armstrong number
```

Calender

In [32]:

```
import calendar

yy = 2023  # year
mm = 10    # month

print(calendar.month(yy, mm))
```

```
    October 2023
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
```

Calculator

In [34]:

```
def add(x, y):
    return x + y


def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y


def divide(x, y):
    return x / y


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

while True:

    choice = input("Enter choice(1/2/3/4): ")


    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))


        next_calculation = input("Let's do next calculation? (yes/no): ")
        if next_calculation == "no":
          break
    else:
        print("Invalid Input")
```

```
Select operation.
1.Add
2.Subtract
3.Multiply
4.Divide
Enter choice(1/2/3/4): 3
Enter first number: 66
Enter second number: 2
66.0 * 2.0 = 132.0
Let's do next calculation? (yes/no): no
```

Reverse a number using a while loop

In [35]:

```
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))
```

```
Reversed Number: 4321
```

Countdown Time

In [36]:

```
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)
```

```
stop1
```

In [ ]: