1. Write a program to find whether a given string is present in the array of strings using

a. Linear Search Algorithm

Code

```
str1 = input("Enter the string : ")
str2 = input("Enter the reference string : ")
cnt = 0
flag = 0

if len(str1)==len(str2):
    for i in range(len(str1)):
        if str1[i]==str2[i]:
            cnt +=1
    if cnt==len(str1):
        print("Your string is matched")
    else:   print("Your string is not matching")
else:
    print("Your string is not matching")
```

Output Case-1

```
Enter the string : PrashanthS
Enter the reference string : PrashanthS
Your string is matched

***Repl Closed***
```

Output Case-2

```
Enter the string : Charles
Enter the reference string : Charels
Your string is not matching

***Repl Closed***
```

b. Binary Search Algorithm

Code

```python
def Binary_Search(list1,str2):
    low = 0
    high = len(list1)
    middle = 0

    while(low<=high):
        middle = int((low+high)/2)
        if list1[middle][0]==str2:
            return list1[middle][1]
        if str2<list1[middle][0]:
            high = middle - 1
        else:
            low = middle + 1

if __name__ == '__main__':
    str1 = "if you're not paying for the product, you are the product"
    str2 = "for"
    list1 = str1.split(" ")
    list1 = [(list1[i],i) for i in range(len(list1))]
    list1 = sorted(list1)
    print("{} is present at the index : {}".format(str2,Binary_Search(list1,str2)))
```

Output:

```
for is present at the index : 4

***Repl Closed***
```

2. Write a program to find the shortest path in a graph using Floyd-Warshall algorithm.

Code:

```python
def Replace(list1):
    for i in range(len(list1)):
        for j in range(len(list1)):
            if (list1[i][j] == 'inf'):
                list1[i][j] = 9999
            elif (list1[i][j] == 9999):
                list1[i][j] = 'inf'
            else:
                list1[i][j] = int(list1[i][j])
    return(list1)

def FloydWarshall(list1):
    for k in range(len(list1)):
        for i in range(len(list1)):
            for j in range(len(list1)):
                list1[i][j] = min(list1[i][j] , (list1[i][k] + list1[k][j]))
    return list1

# list1 = [
#       [0,3,'inf',7] ,
#       [8,0,2,'inf'] ,
#       [5,'inf',0,1] ,
#       [2,'inf','inf',0]]

# list1 = [['0', '3', '5', '6'],
#         ['5', '0', '2', '3'],
#         ['3', '6', '0', '1'],
#         ['2', '5', '7', '0']]

n = int(input("Enter the number of vertices : "))
print("Please Enter the respected details : ")
print("""Caution:
            1)For self-loops enter '0'
            2)If there is no path is between the two vertices enter 'inf'
            3)Enter the distance from a vertice to all other vertices seperated by co
            3)Enter the distance from a vertice to all other vertices seperated by c

list1 = [list(input(f"Enter the distance from vertex {i} to all : ").split(',')) for

list1 = Replace(list1)
list1 = FloydWarshall(list1)
list1 = Replace(list1)
print(list1)
```

Output:

```
Enter the number of vertices : 4
Please Enter the respected details :
Caution:
            1)For self-loops enter '0'
            2)If there is no path is between the two vertices enter 'inf'
            3)Enter the distance from a vertice to all other vertices seperated by comma
[[0, 3, 5, 6], [5, 0, 2, 3], [3, 6, 0, 1], [2, 5, 7, 0]]

***Repl Closed***
```

3. Write a program to implement the Ford-Fulkerson Method.

Code

```
import sys

def BreadthFirstSearch(residual_graph,source,sink,parentTracker):
    queue = []
    visited = []

    for x in range(0,N):  ## Initially the visited[] must be 0
        visited.append(0)

    visited[source] = True
    queue.append(source)
    parentTracker[source] = -1

    while not len(queue) == 0:
        u = queue.pop(0)     ## popping out the element from the queue
        for v in range(0,N): ## comparing the popped element with its neighbours [Ex
            if (residual_graph[u][v]>0 and visited[v]==False):  # if there must be a
                visited[v] = True
                queue.append(v)
                parentTracker[v] = u

    if visited[sink]:   ## sink will hold the last node, if the last node is visited
        return True
    else:
        return False

def FordFulkersonAlgorithm(graph,source,sink):
    u,v  = 0,0  # (pointers pointing the inidividual nodes in the graph) [ u --> row
    residual_graph = graph
    maxflow = 0  ## will keep track of the count of maximum flow
    while BreadthFirstSearch(residual_graph,source,sink,parent_tracker): # will retu
        ## Upto this we took a path from source to sink

        ## Checking the smallest weight from that path
        pathflow = INFINITE
        v = sink
        while not v == source:
            u = parent_tracker[v]
```

```python
                pathflow = min(pathflow,residual_graph[u][v])  ## smallest weight from t
                v = parent_tracker[v]

            ## Subtracting the weights of the chosen path from the smallest weight
            v = sink
            while not v == source:
                u = parent_tracker[v]
                residual_graph[u][v] = residual_graph[u][v] - pathflow
                residual_graph[v][u] = residual_graph[v][u] - pathflow
                v = parent_tracker[v]

            maxflow = maxflow + pathflow
        return maxflow

if __name__ == "__main__":

    N = 6  ## number of nodes
    parent_tracker = []
    INFINITE = sys.maxsize

    graph = \
    [
        [0,16,13,0,0,0] ,
        [0,0,10,12,0,0],
        [0,4,0,0,14,0],
        [0,0,9,0,0,20],
        [0,0,0,7,0,4],
        [0,0,0,0,0,0]
    ]
    source = 0  ## initial node
    sink = 5  ## final node

    for x in range(0,N):  ## initially all parent trackers are 0
        parent_tracker.append(0)

    print("The maximum possible flow : {}".format(FordFulkersonAlgorithm(graph,source
```

Output:

```
The maximum possible flow : 23


***Repl Closed***
```

4. Write programs to implement the following algorithms:

a. Fractional Knapsack Problem

Code

```python
def Maximum(profit_weight):
    maxi = (0,0)
    for i,j in profit_weight:
        if maxi[1]<j:
            maxi = (i,j)
    return maxi

def Knapsack(input_profit,input_weight,input_capacity):
    capacity = input_capacity
    profit_weight = [ int(j/i) for i,j in zip(input_weight,input_profit)]
    weight = [(i,input_weight[i]) for i in range(len(input_weight))]
    profit = [(i,input_profit[i]) for i in range(len(input_profit))]
    profit_weight = [(i,profit_weight[i]) for i in range(len(profit_weight))]

    x = []
    for i in range(len(input_profit)):
        x.append(0)

    for i in range(len(weight)):
        if capacity>0:
            maximum = Maximum(profit_weight)
            index = maximum[0]
            if ((weight[maximum[0]][1])<=capacity):
                capacity = capacity - weight[index][1]
                x[index] = 1
                profit_weight[index] = (0,0)
            elif ((capacity - weight[index][1])  < 0):
                x[index] = capacity/(weight[index][1])
                profit_weight[index] = (0,0)
                capacity = 0
            else:
                x[index] = 0
                profit_weight[index] = (0,0)

            # Calculating summation of x_p
            x_p = 0
            for i,j in profit:
                x_p = x_p + j*x[i]

            # Calculation summation of x_w
            x_w = 0
            for i,j in weight:
```

```python
                x_w = x_w + j*x[i]

            if int(x_w) == input_capacity:
                print("Maximum value in Knapsack : ",x_p)

if __name__ == '__main__':
    print("Shop-1")
    input_profit = [10,5,15,7,6,18,3]
    input_weight = [2,3,5,7,1,4,1]
    input_capacity = 15
    Knapsack(input_profit,input_weight,input_capacity)

    print("Shop-2")
    input_profit = [60,40,100,120]
    input_weight = [10,40,20,30]
    input_capacity = 50
    Knapsack(input_profit,input_weight,input_capacity)
```

Result

```
Shop-1
Maximum value in Knapsack :  55.333333333333
Shop-2
Maximum value in Knapsack :  240.0
```

b. 0/1 Knapsack Problem

```python
## creating an empty i*w matrix
def Empty_list(len_profit_list,len_weight_list):
    list1 = []
    temp_list1 = []
    for i in range(len_profit_list + 1):
        for w in range(len_weight_list + 1):
            temp_list1.append(0)
        list1.append(temp_list1)
        temp_list1 = []
    return list1

def KnapSack(profit_list,weight_list):
    len_profit_list    = len(profit_list)
    len_weight_list = len(profit_list) * 2
    list1 = Empty_list(len_profit_list,len_weight_list)

    for i in range(len_profit_list + 1):
        for w in range(len_weight_list + 1):
            if (i==0 or w==0):  # By-default, 1st row and 1st column --> 0
                list1[i][w] = 0
```

```python

            elif (w>=weight_list[i-1]):  # (list1[i-1][w-w[i]]>=0)
                list1[i][w] = max(   (list1[i-1][w])  ,  (list1[i-1][w-weight_list[i-

            elif (w<weight_list[i-1]):  # (list1[i-1][w-w[i]]<0) (i.e list1[i-1]negat
                list1[i][w] = list1[i-1][w]

    return (list1[len_profit_list][len_weight_list])

if __name__ == '__main__':
    profit_list   = list(map(int,input("Enter the profit list   : ").split()))
    weight_list = list(map(int,input("Enter the weight list : ").split()))
    print("Total profit : ",KnapSack(profit_list,weight_list))
```

Result

```
Enter the profit list   : 1 2 5 6
Enter the weight list : 2 3 4 5
Total profit :  8

***Repl Closed***
```