```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 18:53:03 2024

@author: USER
"""

def recursive_fibonacci(n):
   if n <= 1:
       return n
   else:
       return(recursive_fibonacci(n-1) + recursive_fibonacci(n-2))
n_terms = 10
# check if the number of terms is valid
if n_terms <= 0:
   print("Invalid input ! Please input a positive value")
else:
   print("Fibonacci series:")
   for i in range(n_terms):
       print(recursive_fibonacci(i))
```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 18:57:10 2024

@author: USER
"""

def binarySearch(arr, l, r, x):
    if r >= l:
        mid = l + (r - l) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binarySearch(arr, l, mid-1, x)
        else:
            return binarySearch(arr, mid + 1, r, x)
    else:
        return -1
arr = [2, 3, 4, 10, 40]
x = 10
result = binarySearch(arr, 0, len(arr)-1, x)
if result != -1:
    print("Element is present at index % d" % result)
else:
    print("Element is not present in array")
```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 19:53:27 2024

@author: USER
"""

def binarySearch(v, To_Find):
    lo = 0
    hi = len(v) - 1
    while hi - lo > 1:
        mid = (hi + lo) // 2
        if v[mid] < To_Find:
            lo = mid + 1
        else:
            hi = mid
    if v[lo] == To_Find:
        print("Found At Index", lo)
    elif v[hi] == To_Find:
        print("Found At Index", hi)
    else:
        print("Not Found")
if __name__ == '__main__':
    v = [1, 3, 4, 5, 6]
    To_Find = 3
    binarySearch(v, To_Find)
    To_Find = 10
    binarySearch(v, To_Find)
```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 22:06:07 2024

@author: USER
"""


def divideAndConquer_Max(arr, ind, len):
    maximum = -1;
    if (ind >= len - 2):
        if (arr[ind] > arr[ind + 1]):
            return arr[ind];
        else:
            return arr[ind + 1];
    maximum = divideAndConquer_Max(arr, ind + 1, len);
    if(arr[ind] > maximum):
        return arr[ind];
    else:
        return maximum;
def divideAndConquer_Min(arr, ind, len):
    minimum = 0;
    if (ind >= len - 2):
        if (arr[ind] < arr[ind + 1]):
            return arr[ind];
        else:
            return arr[ind + 1];
    minimum = divideAndConquer_Min(arr, ind + 1, len);
    if(arr[ind] < minimum):
        return arr[ind];
    else:
        return minimum;
```

```
if __name__ == '__main__':
    minimum, maximum = 0, -1;
# array initialization
    arr = [6, 4, 8, 90, 12, 56, 7, 1, 63];
    maximum = divideAndConquer_Max(arr, 0, 9);
    minimum = divideAndConquer_Min(arr, 0, 9);
    print("The minimum number in the array is: ", minimum)
    print("The maximum number in the array is: ", maximum)
```

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 22:09:52 2024

@author: USER
"""


import numpy as np
def strassen(A, B):
    if A.shape == (1, 1): # base case
        return A * B
    else: # divide matrices into quadrants
        n = A.shape[0]
        m = n // 2
        a = A[:m, :m]
        b = A[:m, m:]
        c = A[m:, :m]
        d = A[m:, m:]
        e = B[:m, :m]
        f = B[:m, m:]
        g = B[m:, :m]
        h = B[m:, m:]
# compute 7 matrix multiplications
        p1 = strassen(a, f - h)
        p2 = strassen(a + b, h)
        p3 = strassen(c + d, e)
        p4 = strassen(d, g - e)
        p5 = strassen(a + d, e + h)
        p6 = strassen(b - d, g + h)
        p7 = strassen(a - c, e + f)
# compute the final product
```

```
    c1 = p5 + p4 - p2 + p6
    c2 = p1 + p2
    c3 = p3 + p4
    c4 = p1 + p5 - p3 - p7
# concatenate the quadrants
    C = np.concatenate((np.concatenate((c1, c2), axis=1), np.concatenate((c3, c4), axis=1)), axis=0)
    return C
# test the function
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
print(strassen(A, B))
```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 22:16:52 2024

@author: USER
"""

from collections import defaultdict
class Graph:
    def __init__(self,vertices):
        self.graph = defaultdict(list) #dictionary containing adjacency List
        self.V = vertices #No. of vertices
    def addEdge(self,u,v):
        self.graph[u].append(v)
    def topologicalSortUtil(self,v,visited,stack):
        visited[v] = True
        for i in self.graph[v]:
            if visited[i] == False:
                self.topologicalSortUtil(i,visited,stack)
        stack.insert(0,v)
    def topologicalSort(self):
        visited = [False]*self.V
        stack =[]
        for i in range(self.V):
            if visited[i] == False:
                self.topologicalSortUtil(i,visited,stack)
        print (stack)
g= Graph(6)
g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
```

```
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);
print("Following is a Topological Sort of the given graph")
g.topologicalSort()
```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 22:25:49 2024

@author: USER
"""


def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[i] < arr[l]:
        largest = l
    if r < n and arr[largest] < arr[r]:
        largest = r
    if largest != i:
        (arr[i], arr[largest]) = (arr[largest], arr[i])
        heapify(arr, n, largest)
def heapSort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        (arr[i], arr[0]) = (arr[0], arr[i]) # swap
        heapify(arr, i, 0)
arr = [12, 11, 13, 5, 6, 7, ]
heapSort(arr)
n = len(arr)
print('Sorted array is')
for i in range(n):
    print(arr[i])
```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 22:29:37 2024

@author: USER
"""

INF = 100000
def min(x, y):
    if x < y:
        return x
    return y
def coin_change(d, n, k):
    M = [0]*(n+1)
    for j in range(1, n+1):
        minimum = INF
        for i in range(1, k+1):
            if(j >= d[i]):
                minimum = min(minimum, 1+M[j-d[i]])
                M[j] = minimum
    return M[n]
if __name__ == '__main__':
    d = [0, 1, 2, 3]
    print(coin_change(d, 14, 3))
```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 22:38:29 2024

@author: USER
"""


nV = 4
INF = 999
def floyd_warshall(G):
    distance = list(map(lambda i: list(map(lambda j: j, i)), G))
    for k in range(nV):
        for i in range(nV):
            for j in range(nV):
                distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])
    print_solution(distance)
def print_solution(distance):
    for i in range(nV):
        for j in range(nV):
            if(distance[i][j] == INF):
                print("INF", end=" ")
            else:
                print(distance[i][j], end=" ")
        print(" ")
G = [[0, 3, INF, 5],
     [2, 0, INF, 4],
     [INF, 1, 0, INF],
     [INF, INF, 2, 0]]
floyd_warshall(G)
```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 22:45:13 2024

@author: USER
"""


def knapSack(W, wt, val, n):
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])
            else:
                K[i][w] = K[i-1][w]
    return K[n][W]

val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))
```

```
class Graph():
  def __init__(self, vertices):
    self.V = vertices
    self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

  def printSolution(self, dist):
    print("Vertex \t Distance from Source")
    for node in range(self.V):
      print(node, "\t\t", dist[node])

  def minDistance(self, dist, sptSet):
    min = 1e7
    for v in range(self.V):
      if dist[v] < min and sptSet[v] == False:
        min = dist[v]
        min_index = v
    return min_index

  def dijkstra(self, src):
    dist = [1e7] * self.V
    dist[src] = 0
    sptSet = [False] * self.V
    for cout in range(self.V):
      u = self.minDistance(dist, sptSet)
      sptSet[u] = True
      for v in range(self.V):
        if (
          self.graph[u][v] > 0
          and sptSet[v] == False
          and dist[v] > dist[u] + self.graph[u][v]
        ):
```

```
                    dist[v] = dist[u] + self.graph[u][v]
        self.printSolution(dist)


g = Graph(9)
g.graph = [
    [0, 4, 0, 0, 0, 0, 0, 8, 0],
    [4, 0, 8, 0, 0, 0, 0, 11, 0],
    [0, 8, 0, 7, 0, 4, 0, 0, 2],
    [0, 0, 7, 0, 9, 14, 0, 0, 0],
    [0, 0, 0, 9, 0, 10, 0, 0, 0],
    [0, 0, 4, 14, 10, 0, 2, 0, 0],
    [0, 0, 0, 0, 0, 2, 0, 1, 6],
    [8, 11, 0, 0, 0, 0, 1, 0, 7],
    [0, 0, 2, 0, 0, 0, 6, 7, 0],
]
g.dijkstra(0)
```

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 22:56:22 2024

@author: USER
"""
string = 'BCAADDDCCACACAC'
class NodeTree(object):
    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right

    def children(self):
        return (self.left, self.right)

    def nodes(self):
        return (self.left, self.right)

    def __str__(self):
        return '%s_%s' % (self.left, self.right)



def huffman_code_tree(node, left=True, binString=''):
    if type(node) is str:
        return {node: binString}
    (l, r) = node.children()
    d = dict()
    d.update(huffman_code_tree(l, True, binString + '0'))
    d.update(huffman_code_tree(r, False, binString + '1'))
    return d
```

```
freq = {}
for c in string:
    if c in freq:
        freq[c] += 1
    else:
        freq[c] = 1

freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)
nodes = [(key, freq) for key, freq in freq]

while len(nodes) > 1:
    (key1, c1) = nodes[-1]
    (key2, c2) = nodes[-2]
    nodes = nodes[:-2]
    node = NodeTree(key1, key2)
    nodes.append((node, c1 + c2))
    nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

huffmanCode = huffman_code_tree(nodes[0][0])
print(' Char | Huffman code ')
print('---------------------- ')
for (char, frequency) in freq:
    print(' %-4r |%12s' % (char, huffmanCode[char]))
```

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 23:02:15 2024

@author: USER
"""


#import numpy as np
#import scipy as sp
# Get matrices
c = [-8, -12, -22]
A = [[17, 27, 34], [12, 21, 15]]
b = [91800, 42000]
# define the upper bound and the lower bound
R = (0, None)
T = (0, None)
M = (0, None)
# Implementing the Simplex Algorithm
from scipy.optimize import linprog
# Solve the problem by Simplex method in Optimization
res = linprog(c, A_ub=A, b_ub=b, bounds=(R, T, M), method='simplex', options={"disp": True}) #
linearprogramming problem
print(res)
```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 23:09:21 2024

@author: USER
"""

N = int(input("Enter the number of queens: "))
board = [[0] * N for _ in range(N)]

def attack(i, j):
    for k in range(0, N):
        if board[i][k] == 1 or board[k][j] == 1:
            return True
    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):
                    return True
                board[i][j] = 0
```

```
    return False


if N_queens(N):
    print("Solution exists. Placement of queens:")
    for i in board:
        print(i)
else:
    print("Solution does not exist for N queens.")
```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 23:12:56 2024

@author: USER
"""


def isSubsetSum(set, n, sum):
    # If the target sum is 0, an empty subset is always a solution
    if sum == 0:
        return True
    # If there are no elements left in the set and the target sum is not 0, no solution
    if n == 0 and sum != 0:
        return False
    # If the last element of the set is greater than the target sum, exclude it
    if set[n - 1] > sum:
        return isSubsetSum(set, n - 1, sum)
    # Check if there is a solution by either excluding or including the last element
    return isSubsetSum(set, n - 1, sum) or isSubsetSum(set, n - 1, sum - set[n - 1])

# Example usage
set = [3, 34, 4, 12, 5, 2]
sum = 9
n = len(set)

if isSubsetSum(set, n, sum):
    print("Found a subset with the given sum")
else:
    print("No subset with the given sum")
```

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 23:19:12 2024

@author: USER
"""

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
```

```python
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))
```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan  9 23:20:52 2024

@author: USER
"""


# Python3 program to solve
# Traveling Salesman Problem using
# Branch and Bound.
import math
maxsize = float('inf')

# Function to copy temporary solution
# to the final solution
def copyToFinal(curr_path):
	final_path[:N + 1] = curr_path[:]
	final_path[N] = curr_path[0]

# Function to find the minimum edge cost
# having an end at the vertex i
def firstMin(adj, i):
	min = maxsize
	for k in range(N):
		if adj[i][k] < min and i != k:
			min = adj[i][k]

	return min

# function to find the second minimum edge
# cost having an end at the vertex i
```

```
def secondMin(adj, i):
	first, second = maxsize, maxsize
	for j in range(N):
		if i == j:
			continue
		if adj[i][j] <= first:
			second = first
			first = adj[i][j]

		elif(adj[i][j] <= second and
			adj[i][j] != first):
			second = adj[i][j]

	return second

# function that takes as arguments:
# curr_bound -> lower bound of the root node
# curr_weight-> stores the weight of the path so far
# level-> current level while moving
# in the search space tree
# curr_path[] -> where the solution is being stored
# which would later be copied to final_path[]
def TSPRec(adj, curr_bound, curr_weight,
			level, curr_path, visited):
	global final_res

	# base case is when we have reached level N
	# which means we have covered all the nodes once
	if level == N:

		# check if there is an edge from
```

```
        # last vertex in path back to the first vertex
        if adj[curr_path[level - 1]][curr_path[0]] != 0:

            # curr_res has the total weight
            # of the solution we got
            curr_res = curr_weight + adj[curr_path[level - 1]]\
                                            [curr_path[0]]
            if curr_res < final_res:
                copyToFinal(curr_path)
                final_res = curr_res
        return

    # for any other level iterate for all vertices
    # to build the search space tree recursively
    for i in range(N):

        # Consider next vertex if it is not same
        # (diagonal entry in adjacency matrix and
        # not visited already)
        if (adj[curr_path[level-1]][i] != 0 and
                            visited[i] == False):
            temp = curr_bound
            curr_weight += adj[curr_path[level - 1]][i]

            # different computation of curr_bound
            # for level 2 from the other levels
            if level == 1:
                curr_bound -= ((firstMin(adj, curr_path[level - 1]) +
                                firstMin(adj, i)) / 2)
            else:
                curr_bound -= ((secondMin(adj, curr_path[level - 1]) +
```

```
                                                                firstMin(adj, i)) / 2)

                                # curr_bound + curr_weight is the actual lower bound
                                # for the node that we have arrived on.
                                # If current lower bound < final_res,
                                # we need to explore the node further
                                if curr_bound + curr_weight < final_res:
                                        curr_path[level] = i
                                        visited[i] = True

                                        # call TSPRec for the next level
                                        TSPRec(adj, curr_bound, curr_weight,
                                                level + 1, curr_path, visited)

                                # Else we have to prune the node by resetting
                                # all changes to curr_weight and curr_bound
                                curr_weight -= adj[curr_path[level - 1]][i]
                                curr_bound = temp

                                # Also reset the visited array
                                visited = [False] * len(visited)
                                for j in range(level):
                                        if curr_path[j] != -1:
                                                visited[curr_path[j]] = True

# This function sets up final_path
def TSP(adj):

        # Calculate initial lower bound for the root node
        # using the formula 1/2 * (sum of first min +
        # second min) for all edges. Also initialize the
```

```
	# curr_path and visited array
	curr_bound = 0
	curr_path = [-1] * (N + 1)
	visited = [False] * N

	# Compute initial bound
	for i in range(N):
		curr_bound += (firstMin(adj, i) +
					secondMin(adj, i))

	# Rounding off the lower bound to an integer
	curr_bound = math.ceil(curr_bound / 2)

	# We start at vertex 1 so the first vertex
	# in curr_path[] is 0
	visited[0] = True
	curr_path[0] = 0

	# Call to TSPRec for curr_weight
	# equal to 0 and level 1
	TSPRec(adj, curr_bound, 0, 1, curr_path, visited)

# Driver code

# Adjacency matrix for the given graph
adj = [[0, 10, 15, 20],
	[10, 0, 35, 25],
	[15, 35, 0, 30],
	[20, 25, 30, 0]]
N = 4
```

```python
# final_path[] stores the final solution
# i.e. the // path of the salesman.
final_path = [None] * (N + 1)

# visited[] keeps track of the already
# visited nodes in a particular path
visited = [False] * N

# Stores the final minimum weight
# of shortest tour.
final_res = maxsize

TSP(adj)

print("Minimum cost :", final_res)
print("Path Taken : ", end = ' ')
for i in range(N + 1):
	print(final_path[i], end = ' ')

# This code is contributed by ng24_7
```

```
import numpy as np
from scipy.optimize import linear_sum_assignment


def solve_assignment_problem(cost_matrix):
    row_ind, col_ind = linear_sum_assignment(cost_matrix)
    total_cost = cost_matrix[row_ind, col_ind].sum()
    assignments = list(zip(row_ind, col_ind))
    return assignments, total_cost


# Example cost matrix
cost_matrix = np.array([
    [4, 7, 6],
    [5, 8, 7],
    [6, 9, 8]
])


assignments, total_cost = solve_assignment_problem(cost_matrix)


print("Assignments:")
for assignment in assignments:
    print(f"Worker {assignment[0]} -> Job {assignment[1]}")


print(f"Total Cost: {total_cost}")
```