```
def dijkstras(graph, source):
    visited = set()
    distances = {node: float('inf') for node in graph.keys()}
    distances[source] = 0

    while len(visited) < len(graph):
        min_node = None
        min_dist = float('inf')

        for node in graph:
            if node not in visited and distances[node] < min_dist:
                min_dist = distances[node]
                min_node = node

        visited.add(min_node)
        for neighbor, wt in graph[min_node].items():
            if distances[neighbor] > distances[min_node] + wt:
                distances[neighbor] = distances[min_node] + wt
    return distances

graph = {
    'A': {'B': 5, 'C': 2},
    'B': {'D': 4, 'E': 2},
    'C': {'B': 8, 'E': 7},
    'D': {},
    'E': {'D': 1},
}
final = dijkstras(graph, 'A')
print("The shortest path from A to:")
for node, distance in final.items():
    print(node, "is", distance)




def prims(graph):
    vt =set([1])
    et=[]

    while len(vt)<len(graph):
        min_edge = None
        min_wt = float('inf')

        for v in vt:
            for u,wt in graph[v]:
                if u not in vt and wt<min_wt:
                    min_wt = wt
                    min_edge = (v,u)

        vt.add(min_edge[1])
        et.append(min_edge)

    return et
graph={
    1: [(2,4),(4,8)],
    2: [(1, 4), (3, 3), (4, 1)],
    3: [(2, 3), (4, 7), (6, 8)],
    4: [(1, 8), (6, 3), (3, 7), (2, 1)],
    6: [(4, 3), (3, 8)]
}

min_span_tree = prims(graph)
print("the corresponding edges in minimum spanning tree are :")
for item in min_span_tree :
    print(item[0]," - ",item[1])



def partition(arr,low,high):
```

```python
    piv = arr[low]
    i = low + 1
    j = high
    while True:
        while i<=j and arr[i]<piv:
            i += 1
        while i<=j and arr[j]>piv:
            j -= 1
        if i<=j:
            arr[i],arr[j] = arr[j],arr[i]
        else :
            break

    arr[low],arr[j] = arr[j],arr[low]

    return j
def quicksort(arr,low,high):
    if low<high:
        piv_ind = partition(arr,low,high)
        quicksort(arr,low,piv_ind-1)
        quicksort(arr,piv_ind+1,high)

arr = [5,7,2,9,6,1,3,4,8]
quicksort(arr,0,len(arr)-1)
print("the sorted array is ",arr)
```

```python
def stablemarriage(men_pref, women_pref):
    free_men = list(men_pref.keys())
    engaged = {}

    while free_men:
        m = free_men.pop(0)
        w = men_pref[m][0]

        if w not in engaged:
            engaged[w] = m
        else:
            m1 = engaged[w]
            if women_pref[w].index(m) < women_pref[w].index(m1):
                engaged[w] = m
                free_men.append(m1)
            else:
                men_pref[m].remove(w)

    return engaged

men_pref = {
    'A': ['V', 'W', 'X'],
    'B': ['W', 'V', 'X'],
    'C': ['V', 'W', 'X']
}

women_pref = {
    'V': ['A', 'B', 'C'],
    'W': ['B', 'C', 'A'],
    'X': ['C', 'A', 'B']
}

final_matching = stablemarriage(men_pref, women_pref)

print("These are engaged pairs:")
for pko, pku in final_matching.items():
    print(pko, "-", pku)
```

```python
def maxprofit(req):
    req.sort(key=lambda x:x[1])
    n = len(req)
    dp = [0]*(n+1)
    for i in range(1,n+1):
        j = i-1
        while j>0 and req[j][1]>req[i-1][0]:
            j -= 1
        dp[i] = max(req[i-1][2]+dp[j+1],dp[i-1])
    return dp[n]
req = [(1,2,100),(2,5,200),(3,6,300),(4,8,400),(5,9,500)]
k = maxprofit(req)
print("the maximum profit is ",k)
```

```python
def insertionsort(arr):
    for i in range(1,len(arr)):
        j=i
        while j>0 and arr[j-1]>arr[j]:
            arr[j-1],arr[j]= arr[j],arr[j-1]
            j = j-1
arr = [2,8,3,9,4,1,5,7,6]
insertionsort(arr)
print("the sorted array is : ",arr)
```

```python
def quicksort(arr):
    if len(arr)<=1:
        return arr
    left =[]
    right = []
    piv = arr[0]
    for elem in arr[1:] :
        if elem <= piv:
            left.append(elem)
        else:
            right.append(elem)

    return quicksort(left) + [piv] + quicksort(right)

inplist = [6,2,7,1,9,3,4,0,6,5,8]
print("Sorted array is ",quicksort(inplist))
```

```python
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]

# Example items with weights and values
weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
capacity = 5
```

```
max_value = knapsack(weights, values, capacity)
print("Maximum value:", max_value)
```

```
def mergesort(arr):
    if len(arr)<=1:
        return arr
    mid = len(arr)//2
    left_half = arr[:mid]
    right_half = arr[mid:]
    left_half = mergesort(left_half)
    right_half = mergesort(right_half)
    return merge(left_half,right_half)
def merge(lh,rh):
    res =[]
    li,ri = 0,0
    while li<len(lh) and ri<len(rh) :
        if lh[li]<rh[ri]:
            res.append(lh[li])
            li += 1
        else:
            res.append(rh[ri])
            ri+=1
    res.extend(lh[li:])
    res.extend(rh[ri:])
    return res

arr = [4,1,2,9,5,8,3,7,0,6]
sorted_arr = mergesort(arr)
print(sorted_arr)
```

```
def issafe(arr,x,y,n):
    for i in range(x):
        if arr[i][y]==1:
            return False
    c = y
    r = x
    while c >= 0 and r >=0:
        if arr[r][c] ==1 :
            return False
        c-= 1
        r-=1
    c = y
    r = x
    while r>=0 and c<n:
        if arr[r][c] ==1 :
            return False
        c+=1
        r-=1
    return True

def nqueens(arr,x,n):
    if x>=n:
        return True
    for col in range(n):
        if issafe(arr,x,col,n):
            arr[x][col]=1
            if nqueens(arr,x+1,n):
                return True
            arr[x][col]=0
    return False
```

```
n = int(input("enter the size "))
arr = [[0 for _ in range(n)] for _ in range(n)]
if nqueens(arr,0,n):
    for i in range(n):
        for j in range(n):
            if arr[i][j] == 1:
                print("Q ",end = ' ')
            else :
                print(". ",end = ' ')
        print()
else:
    print("not posssible")
```

```
def bellmanFord(edgeList,V):
    dist = [float('inf') for _ in range(V)]
    dist[0]=0
    for i in range(V-1): # if the graph is of 6 vertices should be processed 5 times so
        for edge in edgeList:
            u,v,w = edge
            u = ord(u)-97
            v = ord(v)-97
            dist[v] = min(dist[v],dist[u]+w)
    return dist


if __name__ == "__main__":
    # An edge list containing all the edges in the form: (from, to, weight)
    edgeList = [('a','b',-4),
                ('a','f',-3),
                ('b','d',-1),
                ('b','e',-2),
                ('c','b',8),
                ('c','f',3),
                ('d','a',6),
                ('d','f',4),
                ('e','c',-3),
                ('e','f',2)]

    # No. of vertices in the graph
    V = 6

    # Function call
    dist = bellmanFord(edgeList, V)

    # To store the characters according to index
    temp = "abcdef"

    print("Distances from a: ")
    for i in range(V):
        print(f"{temp[i]} : {dist[i]}")
```

```
def subsetsum(ins,n,ts):
    if(ts==0):
        return []
    if(n==0):
        return None
    if(ins[n-1]>ts):
        return subsetsum(ins,n-1,ts)
    included=subsetsum(ins,n-1,ts-ins[n-1])
```

```
    if included is not None:
        return included+[ins[n-1]]      ######BRACKET IS DOUBLE TIME
    else:
        return subsetsum(ins,n-1,ts)

ins=[3,34,56,4,2]
ts=9
n=len(ins)
included=subsetsum(ins,n,ts)
if included is not None:
    print("the given subset is possible ",included)
else:
    print("no subset is possible")
```