**BELLMAN FORD**

**def** bellman(graph,source):  
 dist = [float(**"inf"**)]\*(len(graph)+1)  
 dist[source] = 0  
 parent = [**None**]\*(len(graph)+1)  
 edge = []  
 **for** i **in** range(len(graph)):  
 **for** u,v,w **in** graph:  
 **if** dist[u] != float(**"inf"**) **and** dist[u]+w <dist[v]:  
 dist[v] = dist[u]+w  
 parent[v] = u  
 **for** i **in** range(len(parent)):  
 **if** parent[i] == **None**:  
 **continue** edge.append([parent[i],i])  
 **return** edge,dist  
  
graph = []  
n = int(input(**"Enter the number of nodes: "**))  
**for** i **in** range(n):  
 no = int(input(**"Enter the number of nodes connected to %d: "** % (i+1)))  
 **for** j **in** range(no):  
 node,weight = map(int,input(**"enter connected node and weight:\n"**).split())  
 graph.append([i+1,node,weight])  
source = int(input(**"enter the source node: "**))  
edge,dist = bellman(graph,source)  
print(**"Graph input: "**)  
**for** i **in** graph:  
 print (i)  
print(**"Edges selcted: "**, edge)  
print(**"Distance: "**, dist)

COUNTING INVERSIONS

**def** merge\_count(A,B):  
 C,r = [],0  
 **while** A **and** B:  
 **if** A[0] < B[0]:  
 C.append(A.pop(0))  
 **else**:  
 C.append(B.pop(0))  
 r += len(A)  
 **if** A:  
 C += A  
 **if** B:  
 C += B  
 **return** C,r  
**def** sort\_count(arr):  
 **if** len(arr) == 1:  
 **return** arr,0  
 mid = (len(arr)-1)//2  
 A,ra = sort\_count(arr[:mid+1])  
 B,rb = sort\_count(arr[mid+1:])  
 C,r = merge\_count(A,B)  
 **return** C,(ra+rb+r)  
  
a = list(map(int,input(**"Enter array of numbers"**).split()))  
b,count = sort\_count(a)  
print (**"Array: "**+str(a))  
print (**"Number of inversions: "**+str(count))

DIASTRAS

**def** dijkstras(s):  
 d,visited = [999]\*len(graph),[0]\*(len(graph))  
 d[s-1] = 0  
 visited[s-1] = 1  
 vertices = [s]  
 **while** len(vertices) != len(graph):  
 edge={}  
 **for** u **in** vertices:  
 **for** v **in** graph[u]:  
 **if** visited[v-1] == 0:  
 t = d[u-1]+graph[u][v]  
 edge.update({t:[u,v]})  
 min\_d = min(edge.keys())  
 node = edge[min\_d][1]  
 d[node-1] = min\_d  
 visited[node-1] = 1  
 vertices.append(node)  
 **return** d  
  
graph = {}  
n = int(input(**"Enter the number of nodes in graph: "**))  
**for** i **in** range(n):  
 graph[i+1] = {}  
 no = int(input(**"Enter the number of nodes connected to %d"** % (i+1)))  
 **for** j **in** range(no):  
 node,weight = map(int,input(**"Enter connected node and weight of edge: "**))  
 graph[i+1].update({node:weight})  
**for** i **in** graph:  
 print(i,**":"**,graph[i])  
print (**"Distance array: "**)  
print(dijkstras(1))

PRIMS

**def** Prims(graph):  
 U,V = set([1]),set(graph.keys())  
 result = []  
 **while** len(U) != len(V):  
 minw = float(**'inf'**)  
 **for** u **in** U:  
 **for** v **in** graph[u]:  
 **if** v **in** V-U:  
 **if** minw > graph[u][v]:  
 minw = graph[u][v]  
 mine = [u,v]  
 mine.append(minw)  
 result.append(mine)  
 U.add(mine[1])  
 **return** result  
  
graph = {}  
n = int(input(**"Enter the number of nodes: "**))  
**for** i **in** range(n):  
 graph[i+1]={}  
 no = int(input(**"Enter the number of nodes connected to %d"** % (i+1)))  
 **for** j **in** range(no):  
 node,weight = map(int,input(**"Enter connected node and weight of edge: "**))  
 graph[i+1].update({node:weight})  
**for** i **in** graph:  
 print (i,**":"**,graph[i])  
print (**"Tree: "**)  
print (Prims(graph))

KRUSKAL

**def** find(i):  
 **if** parent[i] == -1:  
 **return** i  
 **else**:  
 **return** find(parent[i])  
**def** union(i,j):  
 i\_s = find(i)  
 j\_s = find(j)  
 parent[i\_s] = j\_s  
**def** Kruskals():  
 result = []  
 **for** k **in** range(4):  
 u = graph[k][0]  
 v = graph[k][1]  
 u\_s = find(u)  
 v\_s = find(v)  
 **if** v\_s != u\_s:  
 result.append([u,v,graph[k][2]])  
 union(u,v)  
 **return** result  
  
graph = []  
parent = [-1]\*(len(graph)+1)  
n = int(input(**"enter the number of nodes: "**))  
**for** i **in** range(n):  
 no = int(input(**"Enter the number of nodes connected to %d: "** % (i+1)))  
 **for** j **in** range(no):  
 node,weight = map(int,input(**"Enter the node and weight: "**))  
 graph.append([(i+1),node,weight])  
graph.sort(key=**lambda** x:x[2])  
**for** i **in** graph:  
 print (i)  
print (**"Tree: "**)  
print (Kruskals())

KNAPSACK

**def** matrix(M,W):  
 **for** i **in** range(1,len(items)+1):  
 row = [0]   
 wi = items[i][0]  
 vi = items[i][1]  
 **for** w **in** range(1,W+1):  
 **if** w < wi:  
 row.append(M[i-1][w])  
 **else**:  
 row.append(max(M[i-1][w],vi+M[i-1][w-wi]))  
 M.append(row)  
 **return** M,M[len(items)][W]  
**def** knapsack(M,W):  
 result = []  
 i = len(items)  
 k = W  
 **while** i > 0 **and** k > 0:  
 wi = items[i][0]  
 **if** M[i][k]!=M[i-1][k]:  
 result.append(i)  
 k = k-wi  
 i = i-1  
 **return** result  
  
items={}  
n = int(input(**"enter the number of items: "**))  
**for** i **in** range(n):  
 a = list(map(int,input(**"Enter the weight and value of %d: "** % (i+1)).split()))  
 items.update({(i+1):a})  
W = int(input(**"Enter the max weight of knapsack: "**))  
M = [[0]\*(W+1)]  
M,mat = matrix(items,M,W)  
result = knapsack(items,M,W)  
**for** i **in** M:  
 print (i)  
print (**"Max value for knapsack: "**+str(mat))  
print (**"Items selected for knapsack: "**+str(result))

WEIGHTED INTERVAL SCHEDULING

intervals = []  
n = int(input(**"Enter the number of intervals: "**))  
**for** i **in** range(n):  
 u = int(input(**"Enter start time: "**))  
 v = int(input(**"Enter end time: "**))  
 w = int(input(**"Enter interval weight: "**))  
 intervals.append([u,v,w])  
intervals.sort(key=**lambda** x:x[1])  
p = [0]\*(len(intervals)+1)  
M = [-1]\*(len(intervals)+1)  
M[0] = 0  
**def** predecessor():  
 **for** i **in** range(len(intervals)-1,-1,-1):  
 start\_time = intervals[i][0]  
 **for** j **in** range(i-1,-1,-1):  
 end\_time = intervals[j][1]  
 **if** end\_time <= start\_time:  
 p[i+1] = j+1  
 **break  
def** OPT():  
 **for** i **in** range(1,len(intervals)+1):  
 vi = intervals[i-1][2]  
 M[i] = max(vi+M[p[i]],M[i-1])  
**def** solution():  
 final\_set = []  
 profit = 0  
 j = len(intervals)  
 **while** j != 0:  
 **if** intervals[j-1][2]+M[p[j]] > M[j-1]:  
 final\_set.append(j)  
 profit += intervals[j-1][2]  
 j = p[j]  
 **else**:  
 j -= 1  
 **return** profit,final\_set  
predecessor()  
OPT()  
profit,final\_set=solution()  
print (profit)  
print (final\_set)