In [ ]:
```
dice problem
```

In [2]:
```
def printcombo(c,n):
    for i in range(n):
        print(c[i],end="")
    print("")
def generate(d,n,curr,tar):
    if curr==n:
        sum=0
        for i in range(n):
            sum+=d[i]
        if sum==tar:
            printcombo(d,n)
        return
    for i in range(1,6+1):
        d[curr]=i
        generate(d,n,curr+1,tar)
n=2
tar=10
dice={}
generate(dice,n,0,tar)
```

```
46
55
64
```

In [ ]:
```
TSP
```

In [1]:
```
def tsp(graph):
    n=len(graph)
    visiteds=(1<<n)-1
    memo=[[None]*(1<<n) for _ in range(n)]
    def visit(city,visited):
        if visited==visiteds:
            return graph[city][0]
        if memo[city][visited] is not None:
            return memo[city][visited]
        minc = float('inf')
        for nextc in range(n):
            if not visited&(1 << nextc):
                cost=graph[city][nextc]+visit(nextc,visited | (1<<nextc))
                if cost<minc:
                    minc=cost
        memo[city][visited]=minc
        return minc
    return visit(0,1)
graph=[[0,3,2,3],[3,0,2,4],[2,2,0,2],[3,4,2,0]]
shortest=tsp(graph)
print(shortest)
```

```
10
```

In [ ]:
```
obst
```

In [2]:
```
def optcost(freq, i, j):
    if j < i:
        return 0
    if j == i:
        return freq[i]
    fsum = Sum(freq, i, j)
    Min = 10000000
    for r in range(i, j + 1):
        cost = (optcost(freq, i, r - 1) +
                optcost(freq, r + 1, j))
        if cost < Min:
            Min = cost
    return Min + fsum
def optimalSearchTree(keys, freq, n):
    return optcost(freq, 0, n - 1)
def Sum(freq, i, j):
    s = 0
    for k in range(i, j + 1):
        s += freq[k]
    return s
keys = [10,20,30,40]
freq = [2,3,2,4]
n = len(keys)
print("Cost of Optimal BST is", optimalSearchTree(keys, freq, n))
```

```
Cost of Optimal BST is 21
```

In [ ]:
```
assembly sh
```

In [4]:
```
def fun(a, t, cl, cs, x1, x2, n):
    if cs == n - 1:
    if cl== 0:
        return x1
        else:
        return x2
    same = fun(a, t, cl, cs + 1, x1, x2, n) + a[cl][cs + 1]
    diff = fun(a, t, not cl, cs + 1, x1, x2, n) + a[not cl][cs + 1] + t[cl]
    return min(same, diff)
n=4 # number of stations
a=[[4, 5, 3, 2], [2, 10, 1, 4]] # time taken at each station
t=[[0, 7, 4, 5], [0, 9, 2, 8]] # time taken to switch lines
e1=10 # time taken to enter first line
e2=12 # time taken to enter second line
x1=18 # time taken to exit first line
x2=7 # time taken to exit second line
x=fun(a, t, 0, 0, x1, x2, n) + e1 + a[0][0]
y=fun(a, t, 1, 0, x1, x2, n) + e2 + a[1][0]
print(min(x, y))
```

```
35
```

In [1]:

```
def assembly_line_scheduling(a, t, e, x, n):
    T1 = [0] * (n + 1)
    T2 = [0] * (n + 1)
    T1[1] = e[0] + a[0][0]
    T2[1] = e[1] + a[1][0]
    for j in range(2, n + 1):
        T1[j] = min(T1[j - 1] + a[0][j - 1], T2[j - 1] + t[1][j - 1] + a[0]
        T2[j] = min(T2[j - 1] + a[1][j - 1], T1[j - 1] + t[0][j - 1] + a[1]

    final_time = min(T1[n] + x[0], T2[n] + x[1])
    return final_time
#a = [[4, 5, 3], [2, 10, 1]]
#t = [[0, 7, 4], [0, 9, 2]]
#e = [10, 12]
#x = [18, 7]
#n = 3
n=int(input("ENTER NO OF LINES"))
a=[[0]*n]*2
t=[[0]*n]*2
e,x=[],[]
for i in range(0,2):
        for j in range(0,n):
                a[i][j]=int(input("ENTER COST: "))
for i in range(0,2):
        for j in range(0,n):
                t[i][j]=int(input("ENTER TIME: "))
for i in range(0,2):
        ele=int(input("ENTER ENTRY TIME: "))
        e.append(ele)
for i in range(0,2):
        ele=int(input("ENTER EXIT TIME: "))
        x.append(ele)
min_time = assembly_line_scheduling(a, t, e, x, n)
print(f"The minimum time to process through the assembly lines is {min_time
```

```
ENTER NO OF LINES2
ENTER COST: 2
ENTER COST: 9
ENTER COST: 8
ENTER COST: 6
ENTER TIME: 4
ENTER TIME: 5
ENTER TIME: 7
ENTER TIME: 9
ENTER ENTRY TIME: 10
ENTER ENTRY TIME: 90
ENTER EXIT TIME: 12
ENTER EXIT TIME: 23
The minimum time to process through the assembly lines is 36
```

In [ ]:

```
KNAPSACH PROBLEM
```

In [2]:
```
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w - w
            else:
                dp[i][w] = dp[i - 1][w]
    return dp[n][capacity]
weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
capacity = 8
print(knapsack(weights, values, capacity))
```

```
10
```

In [ ]:
```
warshalls algorithm
```

In [3]:
```
nV = 5
INF = 999
def floyd(G):
    dis = 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):
                dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j])
    printt(dis)
def printt(dis):
    for i in range(nV):
        for j in range(nV):
            if(dis[i][j] == INF):
                print("INF", end=" ")
            else:
                print(dis[i][j], end="  ")
        print(" ")
G = [[0, 3, INF, INF ,2],
     [INF,0, 2, INF, INF],
     [INF,INF, 0,2, INF],
     [INF,INF,INF,0,2],
     [INF,3, 3,INF, 0]]
floyd(G)
```

```
0  3  5  7  2
INF 0  2  4  6
INF 7  0  2  4
INF 5  5  0  2
INF 3  3  5  0
```

In [ ]:
```
def BellmanFord(graph, source):
    distance = [float("Inf")] * (len(graph)-1)
    distance[source] = 0
    for _ in range(len(graph) - 1):
        for u, v, w in graph:
            if distance[u] != float("Inf") and distance[u] + w < distance[v]
                distance[v] = distance[u] + w
    for u, v, w in graph:
        if distance[u] != float("Inf") and distance[u] + w < distance[v]:
            print("Graph contains a negative weight cycle")
            return
    return distance
n=int(input("enter no. of edges:"))
print("edges & costs:")
graph=[]
for i in range(n):
    a=[]
    for j in range(3):
        s=int(input())
        a.append(s)
    graph.append(a)
distances = BellmanFord(graph, 0)
print(distances)
```

In [ ]: