In [14]:
```python
import copy
inf = float('inf')
```

In [23]:
```python
class TSP :

    def __init__ (self,city_matrix=None,source=0):
        self.city_matrix = [[0]*4]*4 if city_matrix is None else city_matrix
        self.n = len(self.city_matrix)
        self.source = source


    def solve(self):
        # intially mincost is infinity
        minCost = inf
        for i in range (self.n):
            print("Path ", end="")
            cost = self._solve(copy.deepcopy(city_matrix),i,i)
            print(f" -> {i+1}    :    Cost = {cost}")
            if cost and cost < minCost:
                minCost = cost
        return minCost


    def _solve(self,city_matrix,currCity=0, source=0):

        if self.n < 2:
            return 0

        print(f" -> {currCity+1} ",end="")

        # set all currcity as inf as they shouldnt be visited
        for i in range(self.n):
            city_matrix[i][currCity] = inf

        # get nearest city to current city
        currMin, currMinPos = inf,0
        for j in range(self.n):
            if currMin > city_matrix[currCity][j]:
                currMin,currMinPos = city_matrix[currCity][j], j

        # If currMin is infinity(i.e. all cities have been visited,
        # return cost of moving from this last city to start city to complete the path-loop)
        if currMin == inf:
            return self.city_matrix[currCity][source]

        # set dist from currCity to next city as inf and vice versa as inf
        city_matrix[currCity][currMinPos] = city_matrix[currMinPos][currCity] = inf

        # calling recursion for next neighbouring cities
        return currMin + self._solve(city_matrix, currMinPos, source)
```

In [25]:
```python
if __name__ == '__main__':
    city_matrix = [
        [inf, 10,  15,  20],
        [10,  inf, 35,  25],
        [15,  35,  inf, 30],
        [20,  25,  30,  inf]
    ]

    source_city = 0
    tsp = TSP(city_matrix, source_city)
    print(f"Optimal Cost : {tsp.solve()}")
```

```
Path  -> 1  -> 2  -> 4  -> 3  -> 1    :    Cost = 80
Path  -> 2  -> 1  -> 3  -> 4  -> 2    :    Cost = 80
Path  -> 3  -> 1  -> 2  -> 4  -> 3    :    Cost = 80
Path  -> 4  -> 1  -> 2  -> 3  -> 4    :    Cost = 95
Optimal Cost : 80
```

In [ ]:

In [28]:
```python
import random
from array import array

board = [[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0
neighbour = [[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0
queens = [0,0,0,0,0,0,0,0]
```

In [29]:
```python
def collision_count(row, column):
    coll = 0
    cr = row
    cc = column

    for j in range(8): # lef - rig
        if j == column: # queen under consideration
            continue
        if board[row][j] == 1 :
            coll += 1

    for j in range(8): # top - bot
        if j == row: # queen under consideration
            continue
        if board[j][column] == 1 :
            coll += 1

    while(cr < 7 and  cc < 7):
        cc += 1
        cr +=1
        if board[cr][cc] == 1:
            coll += 1

    cr, cc = row, column
    while(cc > 0 and cr > 0):
        cr -= 1
        cc -=1
        if board[cr][cc] == 1:
            coll += 1

    cr, cc = row, column
    while(cc > 0 and cr < 7):
        cr += 1
        cc -=1
        if board[cr][cc] == 1:
            coll += 1

    cr, cc = row, column
    while(cc < 7 and cr > 0):
        cr -= 1
        cc +=1
        if board[cr][cc] == 1:
            coll += 1

    return coll
```

In [30]:
```python
def totalcoll():
    totcoll = 0
    for i in range(8):
        totcoll += collision_count(i,queens[i]) # for each row
    return totcoll
```

In [32]:
```python
while True:
#     print("f")
    board = [[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0
    for i in range(8):
        queens[i] = random.randrange(0,8)
        board[i][queens[i]] = 1

    totalcollision =  totalcoll()
#     print("colli ", totalcollision)
    while True:
        for i in range(8):
            oldqueen = queens[i]
            for j in range(8):
                queens[i] = j
                board[i][oldqueen] = 0
                board[i][queens[i]] = 1
                neighbour[i][j] = totalcoll()#generate n
                board[i][queens[i]] = 0
                board[i][oldqueen] = 1
            queens[i] = oldqueen
        min = neighbour[0][0]
        minqueencol = 0
        minqueenrow = 0
        for i in range(8):
            for j in range(8):
                if(neighbour[i][j]<min):
                    min = neighbour[i][j] # should store/generate this state(?)
                    minqueenrow = i
                    minqueencol = j

        if min<totalcollision:
            totalcollision = min

            board[minqueenrow][queens[minqueenrow]] = 0 #
            queens[minqueenrow] = minqueencol
            board[minqueenrow][queens[minqueenrow]] = 1
        else:
            break
    print("colli ", totalcollision)
    if totalcollision == 0:
        break
#     print("colli ", totalcollision)

for i in range(8):
    for j in range(8):
        print(board[i][j], end=" ")
    print()
```

```
colli  4
colli  0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
```

In [ ]:

In [ ]:

In [26]:
```python
def Hanoi(n, A, B, C):
    count = 1
    if n==1 :
        print("Move disk ", n, " from " , A, " to ", C)
        return count
    else :
        count += Hanoi(n-1, A, C, B)
        print("Move disk ", n, " from " , A, " to ", C)
        count += Hanoi(n-1, B, A, C)
        return count
```

In [27]:
```python
limit1=int(input("Enter limit of first jug : "))
limit2=int(input("Enter limit of second jug : "))
aim=int(input("Enter aim:- "))
print("Intial State : "+"0 0")
print("Goal State : "+"0 "+str(aim))

if(limit1>limit2):
    limit1,limit2=limit2,limit1

def waterJug(x,y):

    print(str(x)+" "+str(y))

    if(y==aim):
        return

    elif(y==limit2):
        waterJug(0,x)

    elif(y==0 and x!=0):
        waterJug(0,x)

    elif(x==aim):
        waterJug(x,0)

    elif(x<limit1):
        waterJug(limit1,y)

    elif(x+y<=limit2):
        waterJug(0,x+y)

    elif(x+y>limit2):
        waterJug(x-(limit2-y),limit2)

waterJug(0,0)
```

```
Enter limit of first jug : 4
Enter limit of second jug : 3
Enter aim:- 2
Intial State : 0 0
Goal State : 0 2
0 0
3 0
0 3
3 3
2 4
0 2
```

In [ ]:
```python
x = 0
y = 0
m = 4
n = 3
# 2 8 2 6 3 8

print("Initial state = (0,0)")
print("Capacities = (4,3)")
print("Goal state = (2,y)")

while x != 2:
    r = input("RULES\n\n1. Fill X\n2. Fill Y\n3. Empty X\n4. Empty Y\n5. Fill Y from X\n6. Fill X from Y\n7. Tr
    r = int(r)

    if(r == 1):
        x = m
    elif(r == 2):
        y = n
    elif(r == 3):
        x = 0
    elif(r == 4):
        y = 0
    elif(r == 5):
        t = n - y
        y = n
        x -= t
    elif(r == 6):
        t = m - x
        x = m
        y -= t
    elif(r == 7):
        y += x
        x = 0
    elif(r == 8):
        x += y
        y = 0
    print("Current Status : ", x, y)
    print ()
```