Important Note: We know that TSP doesn't have any optimal solutions, so it

can produce different solutions on different methods based on efficiency.

Bank Locations

```
ID  Latitude           Longitude           Address

1   23.8728568         90.3984184          Uttara Branch

2   23.8513998         90.3944536          City Bank Airport

3   23.8330429         90.4092871          City Bank Nikunja

4   23.8679743         90.3840879          City Bank Beside Uttara
Diagnostic

5   23.8248293         90.3551134          City Bank Mirpur 12

6   23.827149          90.4106238          City Bank Le Meridien

7   23.8629078         90.3816318          City Bank Shaheed Sarani

8   23.8673789         90.429412           City Bank Narayanganj

9   23.8248938         90.3549467          City Bank Pallabi

10  23.813316          90.4147498          City Bank JFP
```

Traveling Sales Man problem

The problem is to visit all the cities or locations exactly once with minimum traveling distance or cost as much as possible.

Calculating the distance between these bank addresses by their Latitute , Longitude using Haversine formula,

and storing them in a cost matrix "g" for further evaluation

In [1]:

```python
from math import radians, cos, sin, asin, sqrt

def distance(l1, l2, l3, l4):

    # radians which converts from degrees to radians.
    # Haversine formula
```

```
    d_lon = radians(l4) - radians(l3)
    d_lat = radians(l2) - radians(l1)
    a = sin(d_lat / 2) ** 2 + cos(l1) * cos(l2) * sin(d_lon / 2) ** 2

    c = 2 * asin(sqrt(a))

    # Radius of earth in kilometers. We can use 3956 for miles
    r = 6371

    # calculating  the result
    return c * r

lat_longs = [
    [23.8728568, 90.3984184], [23.8513998, 90.3944536], [23.8330429, 90.4092871],
    [23.8679743, 90.3840879], [23.8248293, 90.3551134], [23.827149, 90.4106238],
    [23.8629078, 90.3816318], [23.8673789, 90.429412], [23.8248938, 90.3549467],
    [23.813316, 90.4147498]
]

g = [[0] * 10 for _ in range(10)] # The distance between two points in form of adjace


# Loops for calculating distance between two location using their latitude and longit
for i in  range(10): # 10 -> count of cities
    for j in range(10): # 10 -> count of cities
        if i != j:
            lat1 = lat_longs[i][0]
            lat2 = lat_longs[j][0]
            lng1 = lat_longs[i][1]
            lng2 = lat_longs[j][1]
            g[i][j] = round(distance(lat1, lat2, lng1, lng2), 2)
```

In [2]:

```
g
```

Out[2]:

```
[[0, 2.39, 4.44, 0.73, 5.51, 5.1, 1.24, 1.21, 5.5, 6.64],
 [2.39, 0, 2.09, 1.87, 3.19, 2.74, 1.35, 2.11, 3.18, 4.28],
 [4.44, 2.09, 0, 3.96, 1.83, 0.66, 3.43, 3.87, 1.83, 2.2],
 [0.73, 1.87, 3.96, 0, 4.88, 4.61, 0.57, 1.52, 4.88, 6.15],
 [5.51, 3.19, 1.83, 4.88, 0, 1.63, 4.31, 5.26, 0.01, 2.12],
 [5.1, 2.74, 0.66, 4.61, 1.63, 0, 4.08, 4.51, 1.64, 1.54],
 [1.24, 1.35, 3.43, 0.57, 4.31, 4.08, 0, 1.66, 4.31, 5.6],
 [1.21, 2.11, 3.87, 1.52, 5.26, 4.51, 1.66, 0, 5.26, 6.03],
 [5.5, 3.18, 1.83, 4.88, 0.01, 1.64, 4.31, 5.26, 0, 2.13],
 [6.64, 4.28, 2.2, 6.15, 2.12, 1.54, 5.6, 6.03, 2.13, 0]]
```

Now we got the distances between cities in form of km in matrix "g"

For Reference `g[0][0] = "Uttara Branch"` & `g[0][1] = "Distance from Uttara Branch to City Bank Airport"`

` And g[1][0] = "Distance from Uttara Branch to City Bank Airport" ` and so on.**

Now we will convert this matrix to an adjacency Matrix for "Nearest Neighbour" Method.

In [3]:

```
# Branch Names

branch_names = ['Uttara Branch', 'City Bank Airport', 'City Bank Nikunja', 'City Bank
            'City Bank Mirpur 12', 'City Bank Le Meridien', 'City Bank Shaheed Sarani',
            'City Bank Pallabi', 'City Bank JFP']

# Adjacency matrix for storing distance between cities -> Only for Nearest Neighbour .
dis = {}

# Inserting distances to the matrix
```

```
for idx in range(10): # We know that the count of branches are 10
    dis[branch_names[idx]] = {}
    for cost_distance in range(10):
        if idx != cost_distance:
            dis[branch_names[idx]][branch_names[cost_distance]] = g[idx][cost_distanc
```

Let's check the Adjacency Matrix

In [4]:
```
for k,v in dis.items():
    print(k, '>>>', v)
```

```
Uttara Branch >>> {'City Bank Airport': 2.39, 'City Bank Nikunja': 4.44, 'City Bank B
eside Uttara Diagnostic': 0.73, 'City Bank Mirpur 12': 5.51, 'City Bank Le Meridien':
5.1, 'City Bank Shaheed Sarani': 1.24, 'City Bank Narayanganj': 1.21, 'City Bank Pall
abi': 5.5, 'City Bank JFP': 6.64}
City Bank Airport >>> {'Uttara Branch': 2.39, 'City Bank Nikunja': 2.09, 'City Bank B
eside Uttara Diagnostic': 1.87, 'City Bank Mirpur 12': 3.19, 'City Bank Le Meridien':
2.74, 'City Bank Shaheed Sarani': 1.35, 'City Bank Narayanganj': 2.11, 'City Bank Pal
labi': 3.18, 'City Bank JFP': 4.28}
City Bank Nikunja >>> {'Uttara Branch': 4.44, 'City Bank Airport': 2.09, 'City Bank B
eside Uttara Diagnostic': 3.96, 'City Bank Mirpur 12': 1.83, 'City Bank Le Meridien':
0.66, 'City Bank Shaheed Sarani': 3.43, 'City Bank Narayanganj': 3.87, 'City Bank Pal
labi': 1.83, 'City Bank JFP': 2.2}
City Bank Beside Uttara Diagnostic >>> {'Uttara Branch': 0.73, 'City Bank Airport':
1.87, 'City Bank Nikunja': 3.96, 'City Bank Mirpur 12': 4.88, 'City Bank Le Meridie
n': 4.61, 'City Bank Shaheed Sarani': 0.57, 'City Bank Narayanganj': 1.52, 'City Bank
Pallabi': 4.88, 'City Bank JFP': 6.15}
City Bank Mirpur 12 >>> {'Uttara Branch': 5.51, 'City Bank Airport': 3.19, 'City Bank
Nikunja': 1.83, 'City Bank Beside Uttara Diagnostic': 4.88, 'City Bank Le Meridien':
1.63, 'City Bank Shaheed Sarani': 4.31, 'City Bank Narayanganj': 5.26, 'City Bank Pal
labi': 0.01, 'City Bank JFP': 2.12}
City Bank Le Meridien >>> {'Uttara Branch': 5.1, 'City Bank Airport': 2.74, 'City Ban
k Nikunja': 0.66, 'City Bank Beside Uttara Diagnostic': 4.61, 'City Bank Mirpur 12':
1.63, 'City Bank Shaheed Sarani': 4.08, 'City Bank Narayanganj': 4.51, 'City Bank Pal
labi': 1.64, 'City Bank JFP': 1.54}
City Bank Shaheed Sarani >>> {'Uttara Branch': 1.24, 'City Bank Airport': 1.35, 'City
Bank Nikunja': 3.43, 'City Bank Beside Uttara Diagnostic': 0.57, 'City Bank Mirpur 1
2': 4.31, 'City Bank Le Meridien': 4.08, 'City Bank Narayanganj': 1.66, 'City Bank Pa
llabi': 4.31, 'City Bank JFP': 5.6}
City Bank Narayanganj >>> {'Uttara Branch': 1.21, 'City Bank Airport': 2.11, 'City Ba
nk Nikunja': 3.87, 'City Bank Beside Uttara Diagnostic': 1.52, 'City Bank Mirpur 12':
5.26, 'City Bank Le Meridien': 4.51, 'City Bank Shaheed Sarani': 1.66, 'City Bank Pal
labi': 5.26, 'City Bank JFP': 6.03}
City Bank Pallabi >>> {'Uttara Branch': 5.5, 'City Bank Airport': 3.18, 'City Bank Ni
kunja': 1.83, 'City Bank Beside Uttara Diagnostic': 4.88, 'City Bank Mirpur 12': 0.0
1, 'City Bank Le Meridien': 1.64, 'City Bank Shaheed Sarani': 4.31, 'City Bank Naraya
nganj': 5.26, 'City Bank JFP': 2.13}
City Bank JFP >>> {'Uttara Branch': 6.64, 'City Bank Airport': 4.28, 'City Bank Nikun
ja': 2.2, 'City Bank Beside Uttara Diagnostic': 6.15, 'City Bank Mirpur 12': 2.12, 'C
ity Bank Le Meridien': 1.54, 'City Bank Shaheed Sarani': 5.6, 'City Bank Narayangan
j': 6.03, 'City Bank Pallabi': 2.13}
```

Brute Force Method

This method is optimal but takes too much time, because the time complexity is `O(n!)` where n is the number of cities .

The basic idea of this method is to generate all permutations of the locations and calculate their distance sums than compare with each other to find out the minimum distance.

In [5]:
```
from itertools import permutations
from sys import maxsize
```

```python
def tsp_brute_force(graph, s):

    # store all vertex apart from source vertex
    vertex = []
    for i in range(1, 10):
        vertex.append(i)

    # store minimum distance

    min_path = maxsize
    next_permutation = permutations(vertex)
    route = ['Uttara Branch']
    path_indexes = None
    for i in next_permutation:

        # store current Path weight(cost)
        current_pathweight = 0

        # compute current path weight
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]

        # update minimum
        if current_pathweight < min_path:
            min_path = current_pathweight
            path_indexes = list(i)

    for p in path_indexes:
        route.append(branch_names[p])

    return min_path, route

s = 0
result = tsp_brute_force(g, s)
print("Distance is : ", result[0])
print("Route is : ", ' >>> '.join(result[1]))
```

```
Distance is :  15.24
Route is :  Uttara Branch >>> City Bank Narayanganj >>> City Bank Nikunja >>> City Ba
nk Le Meridien >>> City Bank JFP >>> City Bank Mirpur 12 >>> City Bank Pallabi >>> Ci
ty Bank Airport >>> City Bank Shaheed Sarani >>> City Bank Beside Uttara Diagnostic
```

Nearest Neighbour Method

The idea behind the nearest neighbor algorithm is to start at a given/random location and visit the closest location that has not yet been visited with minimum distance. This process is repeated until all locations have been visited. The nearest neighbor algorithm is simple to implement and has a time complexity of `O(n^2)`, where n is the number of locations.

In [6]:
```python
# Nearest Neighbour Method

def tsp_nearest_neighbor(cities, distances):

    route = [cities[0]]
    remaining_cities = cities[1:]

    while remaining_cities:
        closest_city = min(remaining_cities, key=lambda x: distances[route[-1]][x])
        route.append(closest_city)
        remaining_cities.remove(closest_city)

    route.append(route[0])
```

```
    dist = 0
    for d in range(len(route) - 1):
        dist += distances[route[d]][route[d+1]]

    return route, dist

route, distance = tsp_nearest_neighbor(branch_names, dis)


print("Distance is: ", distance)
print()
print("Route is :", ' >>> '.join(route[:-1]))
```

```
Distance is:  15.54

Route is : Uttara Branch >>> City Bank Beside Uttara Diagnostic >>> City Bank Shaheed Sarani >>> City Bank Airport >>> City Bank Nikunja >>> City Bank Le Meridien >>> City Bank JFP >>> City Bank Mirpur 12 >>> City Bank Pallabi >>> City Bank Narayanganj
```

Dynmic Programming Approach

This method do have advantages and disadvantages, It calculates most optimal solution but takes longer to do that. For example if we pass 15 locations it will take almost 13 - 16 seconds. And if we optimize it using memoization memory will be a issue rather than run time

The idea here to recursively visit all distinct permutations of the locations and find out the minimum distance

Beside I was unable to find a way to keep track of the path in this algorithm, maybe in future I will.

In [7]:

```
# We will use bitmask to track visited cities , It's faster than real values.
# However I coudn't find a way to track the path for this algorithm.

n = 10 # Count of cities
visited = (1<<n) - 1
dp = [[-1]*n]*(visited+1) # We know that number of distinct possible combinations is

def tsp_dp(mask, pos):
    if mask == visited:
        return g[pos][0]

    # Lookup case
    if dp[mask][pos] != -1:
        return dp[mask][pos]

    # Loop through unvisited cities
    ans = 10**9
    for city in range(n):

        if mask & (1 << city) == 0:
            new_ans = g[pos][city] + tsp_dp(mask | (1 << city), city)
            if new_ans < ans:
                ans = new_ans


    dp[mask][pos] = ans
    return ans

val = tsp_dp(1, 0)
print("Distance is : ", val)
```

```
Distance is :  14.27
```

Did So Much Research!

Thank

You !

In [8]:

```
Note: you may need to restart the kernel to use updated packages.
```

In [ ]: