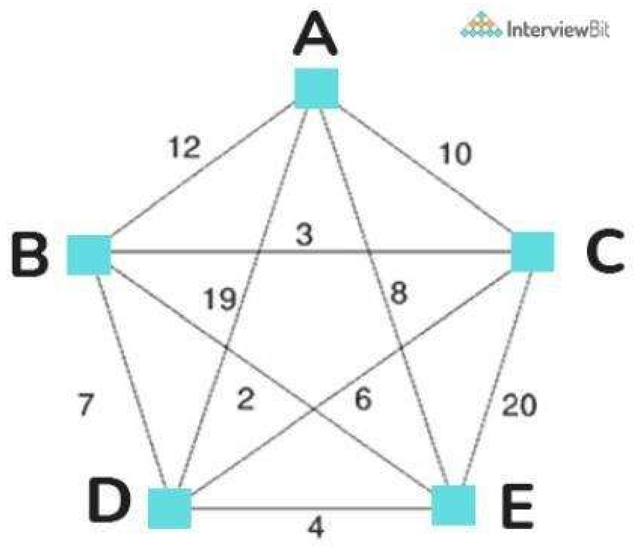
DAA
MINI PROJECT

BHAVYA MALHOTRA (RA2111003010951)
ABHIGYAN KASHYAP (RA2111003010936)

Problem Statement: Given a set of cities and the distance between every pair of cities as an adjacency matrix, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.



The above Problem can be solved using **Travelling Salesman Approach** using **Dynamic Programming**, **Greedy Algorithm** and **Simple Approach** ● **Travelling Salesman Problem Using Dynamic Programming**

In the travelling salesman problem algorithm, we take a subset N of the required cities that need to be visited, the distance among the cities dist, and starting city s as inputs. Each city is identified by a unique city id which we say like 1,2,3,4,5………n

Here we use a dynamic approach to calculate the cost function Cost(). Using recursive calls, we calculate the cost function for each subset of the original problem.

To calculate the cost(i) using **Dynamic Programming**, we need to have some recursive relation in terms of sub-problems.

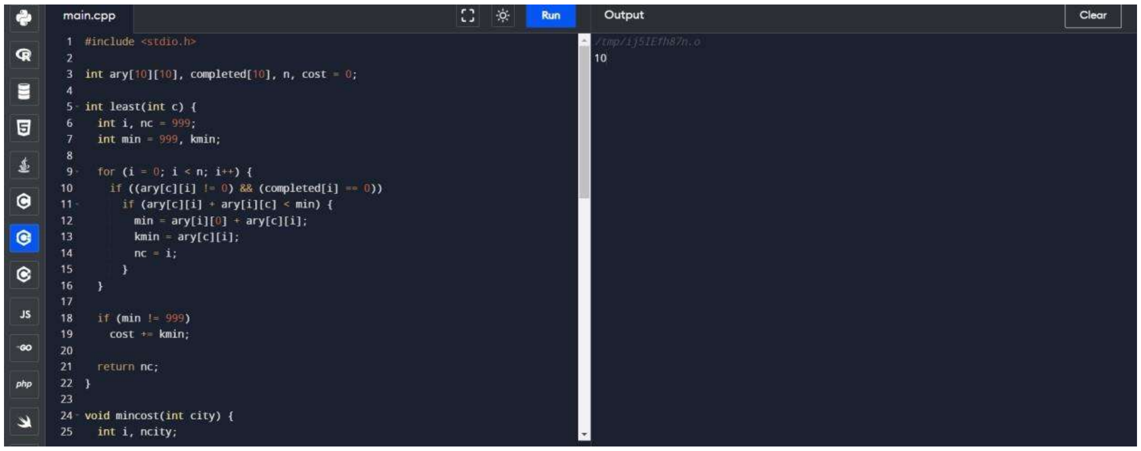
We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on.

CODE IMPLEMENTATION USING C++:

```
#include <bits/stdc++.h> using

namespace std;

#define V 4 int travellingSalesmanProblem(int

graph[][V], int s)

{

   vector<int> vertex;     for

(int i = 0; i < V; i++)         if (i

!= s)

vertex.push_back(i);


   int min_path = INT_MAX;

   do {        int

current_pathweight = 0;
```

```
        int k = s;          for (int i = 0; i < vertex.size();
i++) {             current_pathweight +=
graph[k][vertex[i]];             k = vertex[i];
        }
        current_pathweight += graph[k][s];

        // update minimum          min_path =
min(min_path, current_pathweight);

    } while (          next_permutation(vertex.begin(),
vertex.end()));

    return min_path;
}
int main()
{     int graph[][V] = { { 0, 10, 15, 20
},
                        { 10, 0, 35, 25 },
                        { 15, 35, 0, 30 },
{ 20, 25, 30, 0 } };     int s = 0;

    cout << travellingSalesmanProblem(graph, s) << endl;
    return 0;

}
```

OUTPUT:



```
#include <stdio.h>

int ary[10][10], completed[10], n, cost = 0;

int least(int c) {
  int i, nc = 999;
  int min = 999, kmin;

  for (i = 0; i < n; i++) {
    if ((ary[c][i] != 0) && (completed[i] == 0))
      if (ary[c][i] + ary[i][c] < min) {
        min = ary[i][0] + ary[c][i];
        kmin = ary[c][i];
        nc = i;
      }
  }

  if (min != 999)
    cost += kmin;

  return nc;
}

void mincost(int city) {
  int i, ncity;
```

```
/tmp/1j5IEfh87n.o
10
```

TIME COMPLEXITY:

There are at most O(n2^n) subproblems, and each one takes linear time to solve. The total running time is, therefore, O(n^22^n). The time complexity is much less than O(n!) but still exponential.

- **Travelling Salesman Problem Using Greedy Approach**

This problem can be related to the Hamiltonian Cycle Problem, in a way that here we know a Hamiltonian cycle exists in the graph, but our job is to find the cycle with minimum cost. Also, in a particular TSP graph, there can be many hamiltonian cycles but we need to output only one that satisfies our required aim of the problem.

Approach: This problem can be solved using Greedy Approach. The steps are given below:

1. Create two primary data holders:
    - A list that holds the indices of the cities in terms of the input matrix of distances between cities.
    - Result array which will have all cities that can be displayed out to the console in any manner.
2. Perform traversal on the given adjacency matrix **tsp[][]** for all the city and if the cost of reaching any city from the current city is less than current cost the update the cost.
3. Generate the minimum path cycle using the above step and return their minimum cost.

CODE IMPLEMENTATION USING C++:

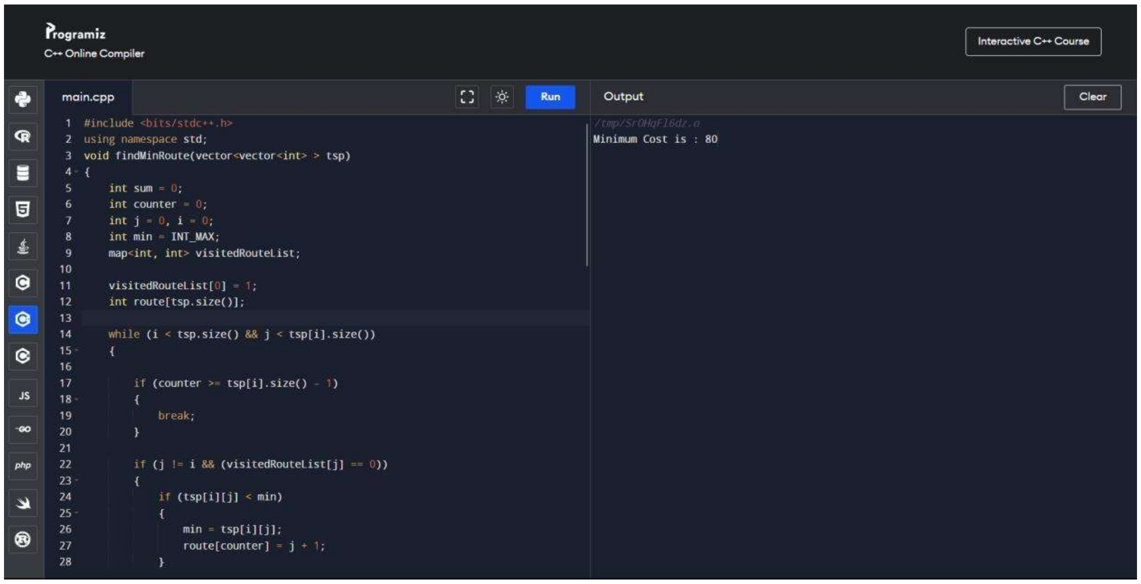
```
#include <bits/stdc++.h> using

namespace std;


// Function to find the minimum // cost path

for all the paths void

findMinRoute(vector<vector<int> > tsp)

{

   int sum = 0;

int counter = 0;

int j = 0, i = 0;

   int min = INT_MAX;

map<int, int> visitedRouteList;
```

```
   // Starting from the 0th indexed city i.e., the first city


   visitedRouteList[0] = 1;
int route[tsp.size()];


   // Traverse the adjacency matrix tsp[][]
while (i < tsp.size() && j < tsp[i].size())
   {


         // Corner of the Matrix
 if (counter >= tsp[i].size() - 1)
         {
                 break;
         }


    // If this path is unvisited then and if the cost is less then update the cost      if
(j != i && (visitedRouteList[j] == 0))
         {               if (tsp[i][j]
< min)
                 {
                         min = tsp[i][j];
                         route[counter] = j + 1;
                 }
```

```
          }
 j++;


          // Check all paths from the ith indexed city
 if (j == tsp[i].size())
          {
                    sum += min;
          min = INT_MAX;
                    visitedRouteList[route[counter] - 1] = 1;
                    j = 0;
                    i = route[counter] - 1;
                    counter++;
          }
   }


   // Update the ending city in array from city which was last visited
i = route[counter - 1] - 1;


   for (j = 0; j < tsp.size(); j++)
   {


          if ((i != j) && tsp[i][j] < min)
          {
```

```
                    min = tsp[i][j];
route[counter] = j + 1;
            }
    }
    sum += min;

    // Started from the node where we finished as well.
cout << ("Minimum Cost is : ");     cout << (sum);
}

// Driver Code int
main()
{

    // Input Matrix     vector<vector<int> > tsp = {
{ -1, 10, 15, 20 },
                                                            { 10, -1, 35, 25 },
                                                            { 15, 35, -1, 30 },
                                                            { 20, 25, 30, -1 } };

    findMinRoute(tsp);
}
```

OUTPUT:



```cpp
#include <bits/stdc++.h>
using namespace std;
void findMinRoute(vector<vector<int> > tsp)
{
    int sum = 0;
    int counter = 0;
    int j = 0, i = 0;
    int min = INT_MAX;
    map<int, int> visitedRouteList;

    visitedRouteList[0] = 1;
    int route[tsp.size()];

    while (i < tsp.size() && j < tsp[i].size())
    {

        if (counter >= tsp[i].size() - 1)
        {
            break;
        }

        if (j != i && (visitedRouteList[j] == 0))
        {
            if (tsp[i][j] < min)
            {
                min = tsp[i][j];
                route[counter] = j + 1;
            }
```

```
/tmp/SrOHqF16dz.o
Minimum Cost is : 80
```

TIME COMPLEXITY:

The time complexity of the Travelling Salesman Approach is dependent on the specific algorithm and data structures used. In this code, the time complexity is **O(n^2 * log n).** It can be improved using data structures like priority queue and hash tables.

- **Travelling Salesman Problem Using Simple Approach**

<u>Problem Statement</u>

- Given a set of cities and the distance between every pair of cities as an adjacency matrix, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

-> Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.

-> Now, we will generate all possible permutations of cities which are (n-1)!.

-> Find the cost of each permutation and keep track of the minimum cost permutation.

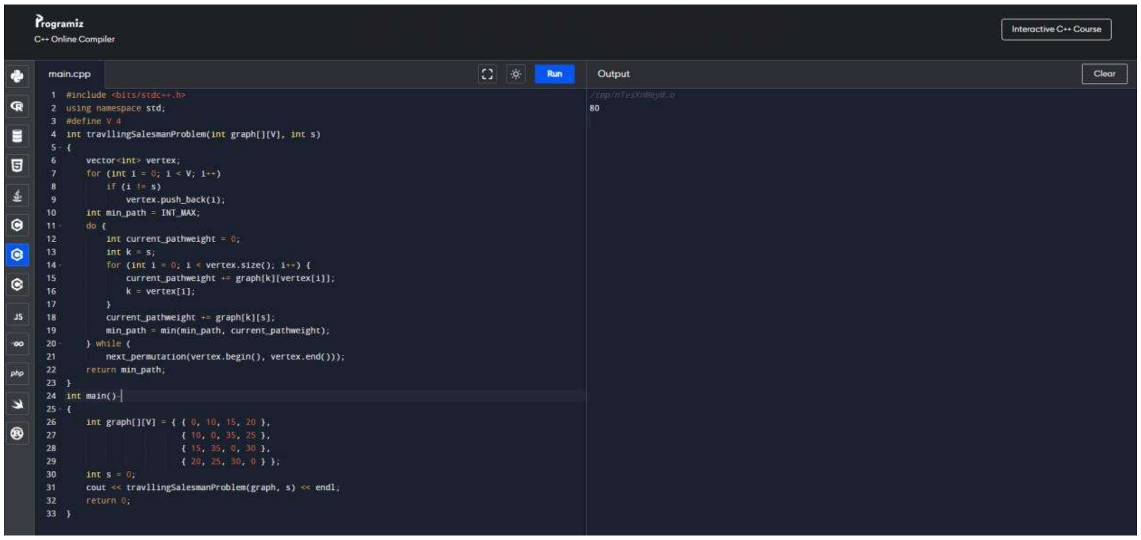
-> Return the permutation with minimum cost.

CODE IMPLEMENTATION USING C++

```
#include <bits/stdc++.h> using

namespace std;

#define V 4 int travllingSalesmanProblem(int

graph[][V], int s)

{

   vector<int> vertex;

for (int i = 0; i < V; i++)

if (i != s)

         vertex.push_back(i);


   int min_path = INT_MAX;

   do {       int

current_pathweight = 0;

      int k = s;       for (int i = 0; i <

vertex.size(); i++) {
```

```
current_pathweight +=
graph[k][vertex[i]];

            k = vertex[i];
        }
        current_pathweight += graph[k][s];


        // update minimum         min_path =
min(min_path, current_pathweight);


    } while (         next_permutation(vertex.begin(),
vertex.end()));


    return min_path;
}
int main()
{     int graph[][V] = { { 0, 10, 15, 20
},
                       { 10, 0, 35, 25 },
                       { 15, 35, 0, 30 },
{ 20, 25, 30, 0 } };     int s = 0;
    cout << travllingSalesmanProblem(graph, s) << endl;
    return 0;
}
```

OUTPUT



```cpp
#include <bits/stdc++.h>
using namespace std;
#define V 4
int travllingSalesmanProblem(int graph[][V], int s)
{
    vector<int> vertex;
    for (int i = 0; i < V; i++)
        if (i != s)
            vertex.push_back(i);
    int min_path = INT_MAX;
    do {
        int current_pathweight = 0;
        int k = s;
        for (int i = 0; i < vertex.size(); i++) {
            current_pathweight += graph[k][vertex[i]];
            k = vertex[i];
        }
        current_pathweight += graph[k][s];
        min_path = min(min_path, current_pathweight);
    } while (
        next_permutation(vertex.begin(), vertex.end()));
    return min_path;
}
int main()
{
    int graph[][V] = { { 0, 10, 15, 20 },
                       { 10, 0, 35, 25 },
                       { 15, 35, 0, 30 },
                       { 20, 25, 30, 0 } };
    int s = 0;
    cout << travllingSalesmanProblem(graph, s) << endl;
    return 0;
}
```

```
/tmp/nTvsXnMeyM.o
80
```

Time complexity: O(N!), Where N is the number of cities. **Space complexity:** O(1).

JUSTIFICATION

BEST APPROACH AMONG THREE:-

If the Travelling Salesman Problem instance is small or moderate in size, and finding the optimal solution is a priority, the dynamic approach may be the best choice. However, if computational efficiency is a higher priority and finding an approximate solution quickly is acceptable**, "the greedy approach may be suitable".** The simple approach is usually not practical for large-scale TSP instances due to its exponential computational complexity

It's also worth mentioning that there are many other advanced techniques and algorithms specifically designed for solving TSP, such as genetic algorithms, ant colony optimization, simulated annealing, and others, which can often provide good solutions with a good trade-off between solution quality and computational time. The best approach for solving TSP ultimately depends on the specific problem requirements and constraints, and it may require experimentation and comparison to determine the most suitable approach for a particular use case.

*******THE END*******