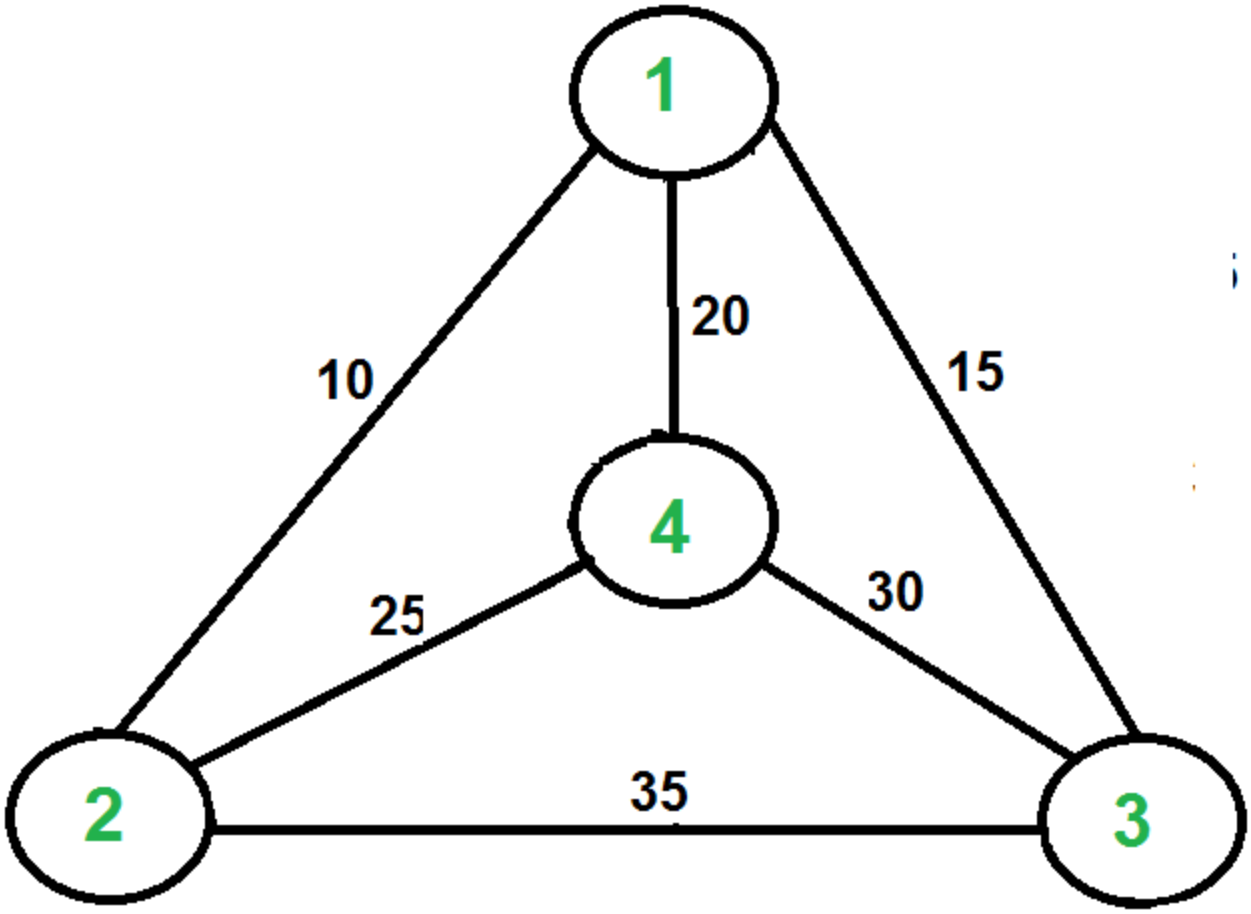
Travelling Salesman Problem using Dynamic Programming

Travelling Salesman Problem (TSP):
Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80. The problem is a famous NP-hard problem. There is no polynomial-time know

solution for this problem. The following are different solutions for the traveling salesman problem.

Naive Solution:

1) Consider city 1 as the starting and ending point.

2) Generate all (n-1)! Permutations of cities.

3) Calculate the cost of every permutation and keep track of the minimum cost permutation.

4) Return the permutation with minimum cost.

Time Complexity: Θ(n!)

Dynamic Programming:

Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output. For every other vertex I (other than 1), we find the minimum cost path with 1 as the starting point, I as the ending point, and all vertices appearing exactly once. Let the cost of this path cost (i), and the cost of the corresponding Cycle would cost (i) + dist(i, 1) where dist(i, 1) is the distance from I to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far.

Now the question is how to get cost(i)? To calculate the cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems.

Let us define a term *C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i*. 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. Note that 1 must be present in every subset.

```
If size of S is 2, then S must be {1, i},

 C(S, i) = dist(1, i)

Else if size of S is greater than 2.

 C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i
and j != 1.
```

Below is the dynamic programming solution for the problem using top down recursive+memoized approach:-

For maintaining the subsets we can use the bitmasks to represent the remaining nodes in our subset. Since bits are faster to operate and there are only few nodes in graph, bitmasks is better to use.

For example: –

10100 represents node 2 and node 4 are left in set to be processed

010010 represents node 1 and 4 are left in subset.

NOTE:- ignore the 0th bit since our graph is 1-based

```
#include <iostream>


using namespace std;


// there are four nodes in example graph (graph is 1-based)

const int n = 4;

// give appropriate maximum to avoid overflow

const int MAX = 1000000;


// dist[i][j] represents shortest distance to go from i to j

// this matrix can be calculated for any given graph using

// all-pair shortest path algorithms

int dist[n + 1][n + 1] = {
```

```
    { 0, 0, 0, 0, 0 },    { 0, 0, 10, 15, 20 },

    { 0, 10, 0, 25, 25 }, { 0, 15, 25, 0, 30 },

    { 0, 20, 25, 30, 0 },

};



// memoization for top down recursion

int memo[n + 1][1 << (n + 1)];



int fun(int i, int mask)

{

    // base case

    // if only ith bit and 1st bit is set in our mask,

    // it implies we have visited all other nodes already

    if (mask == ((1 << i) | 3))

        return dist[1][i];

    // memoization

    if (memo[i][mask] != 0)

        return memo[i][mask];
```

```
    int res = MAX; // result of this sub-problem



    // we have to travel all nodes j in mask and end the

    // path at ith node so for every node j in mask,

    // recursively calculate cost of travelling all nodes in

    // mask except i and then travel back from node j to

    // node i taking the shortest path take the minimum of

    // all possible j nodes



    for (int j = 1; j <= n; j++)

        if ((mask & (1 << j)) && j != i && j != 1)

            res = std::min(res, fun(j, mask & (~(1 << i)))

                                    + dist[j][i]);

    return memo[i][mask] = res;

}

// Driver program to test above logic

int main()

{

    int ans = MAX;
```

```
    for (int i = 1; i <= n; i++)

        // try to go from node 1 visiting all nodes in

        // between to i then return from i taking the

        // shortest route to 1

        ans = std::min(ans, fun(i, (1 << (n + 1)) - 1)

                                + dist[i][1]);



    printf("The cost of most efficient tour = %d", ans);



    return 0;

}



// This code is contributed by Serjeel Ranjan
```

Output

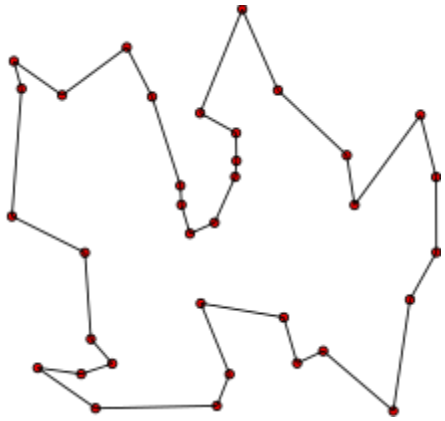
```
The cost of most efficient tour = 80
```

Time Complexity : $O(n^2*2^n)$ where $O(n* 2^n)$ are maximum number of unique subproblems/states and O(n) for transition (through for loop as in code) in every states.

Auxiliary Space: $O(n*2^n)$, where n is number of Nodes/Cities here.

For a set of size n, we consider n-2 subsets each of size n-1 such that all subsets don't have nth in them. Using the above recurrence relation, we can write a dynamic programming-based solution. There are at most $O(n*2^n)$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2*2^n)$. The time complexity is much less than O(n!) but still exponential. The

space required is also exponential. So this approach is also infeasible even for a slightly higher number of vertices. We will soon be discussing approximate algorithms for the traveling salesman problem.



[Solution of a travelling salesman problem: the black line shows the shortest possible loop that connects every red dot.]

The **travelling salesman problem** (also called the **travelling salesperson problem** or **TSP**) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research.

The travelling purchaser problem and the vehicle routing problem are both generalizations of TSP.

In the theory of computational complexity, the decision version of the TSP (where given a length *L*, the task is to decide whether the graph has a tour of at most *L*) belongs to the class of NP-complete problems. Thus, it is possible that the worst-case running time for any algorithm for the TSP increases superpolynomially (but no more than exponentially) with the number of cities.

The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, many heuristics and exact algorithms are known, so that some instances with tens of thousands of cities can be solved completely and even problems with millions of cities can be approximated within a small fraction of 1%.[1]

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept *city* represents, for example, customers, soldering points, or DNA fragments, and the concept *distance* represents travelling times or cost, or a similarity measure between DNA

fragments. The TSP also appears in astronomy, as astronomers observing many sources will want to minimize the time spent moving the telescope between the sources; in such problems, the TSP can be embedded inside an optimal control problem. In many applications, additional constraints such as limited resources or time windows may be imposed.