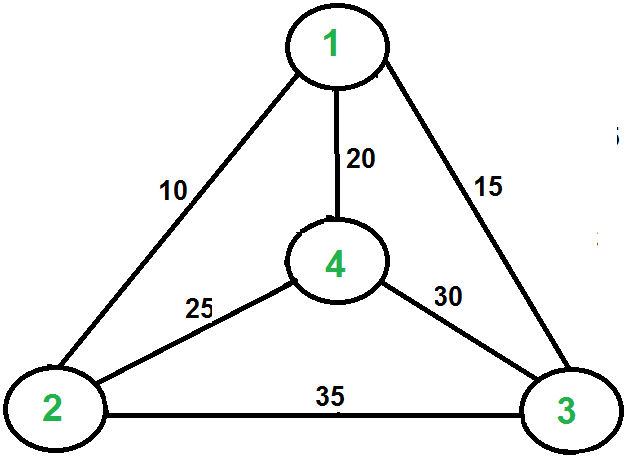
Travelling Salesman Problem

**Travelling Salesman Problem (TSP):** Given a set of cities and 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](https://www.geeksforgeeks.org/backtracking-set-7-hamiltonian-cycle/) and TSP. The Hamiltoninan cycle problem is to find if there exist 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.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/Euler12.png)

For example, consider the graph shown in figure on 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](https://www.geeksforgeeks.org/np-completeness-set-1/)problem. There is no polynomial time know solution for this problem.

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](https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/)of cities.

3) Calculate cost of every permutation and keep track of 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 be cost(i), the cost of corresponding Cycle would be 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 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.

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 dynamic programming based solution. There are at most O(n\*2n) subproblems, and each one takes linear time to solve. The total running time is therefore O(n2\*2n). The time complexity is much less than O(n!), but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.

# Differentiate between Divide & Conquer Method vs Dynamic Programming.

|  |  |
| --- | --- |
| **Divide & Conquer Method** | **Dynamic Programming** |
| **1.**It deals (involves) three steps at each level of recursion: **Divide** the problem into a number of subproblems. **Conquer** the subproblems by solving them recursively. **Combine** the solution to the subproblems into the solution for original subproblems. | **1.**It involves the sequence of four steps:   * Characterize the structure of optimal solutions. * Recursively defines the values of optimal solutions. * Compute the value of optimal solutions in a Bottom-up minimum. * Construct an Optimal Solution from computed information. |
| **2.** It is Recursive. | **2.** It is non Recursive. |
| **3.** It does more work on subproblems and hence has more time consumption. | **3.** It solves subproblems only once and then stores in the table. |
| **4.** It is a top-down approach. | **4.** It is a Bottom-up approach. |
| **5.** In this subproblems are independent of each other. | **5.** In this subproblems are interdependent. |
| **6. For example:** Merge Sort & Binary Search etc. | **6. For example:** Matrix Multiplication. |

**Differentiate between Dynamic Programming and Greedy Method**

|  |  |
| --- | --- |
| **Dynamic Programming** | **Greedy Method** |
| 1. Dynamic Programming is used to obtain the optimal solution. | 1. Greedy Method is also used to get the optimal solution. |
| 2. In Dynamic Programming, we choose at each step, but the choice may depend on the solution to sub-problems. | 2. In a greedy Algorithm, we make whatever choice seems best at the moment and then solve the sub-problems arising after the choice is made. |
| 3. Less efficient as compared to a greedy approach | 3. More efficient as compared to a greedy approach |
| 4. Example: 0/1 Knapsack | 4. Example: Fractional Knapsack |
| 5. It is guaranteed that Dynamic Programming will generate an optimal solution using Principle of Optimality. | 5. In Greedy Method, there is no such guarantee of getting Optimal Solution. |

**RELEVANT READING MATERIAL AND REFERENCES:**

**Source Notes:**

1. <https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_travelling_salesman_problem.htm>
2. <https://www.javatpoint.com/divide-and-conquer-method-vs-dynamic-programming>
3. <https://www.javatpoint.com/dynamic-programming-vs-greedy-method>

**Lecture Video:**

1. <https://www.youtube.com/watch?v=hvDx7q6vcWM&feature=youtu.be>
2. <https://youtu.be/XaXsJJh-Q5Y>

**Online Notes:**

1. <http://vssut.ac.in/lecture_notes/lecture1428551222.pdf>

**Text Book Reading:**

1. Cormen, Leiserson, Rivest, Stein, “*Introduction to Algorithms*”, Prentice Hall of India, 3rd edition 2012. problem, Graph coloring.

**In addition: PPT can be also be given.**