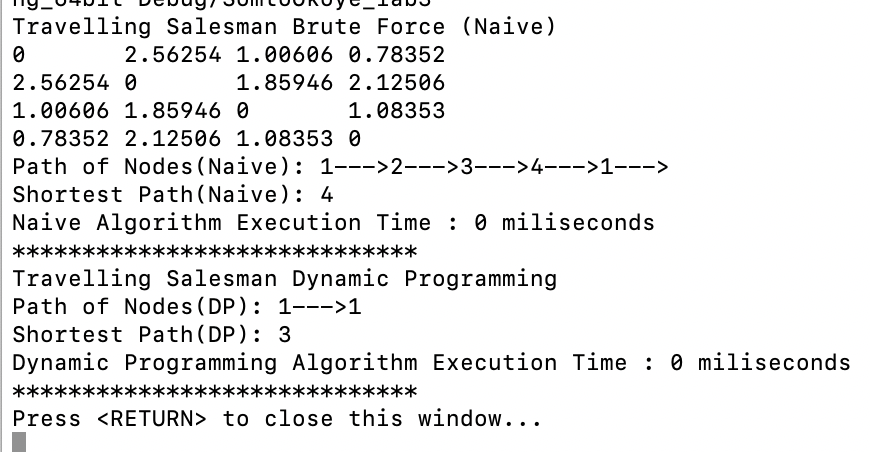
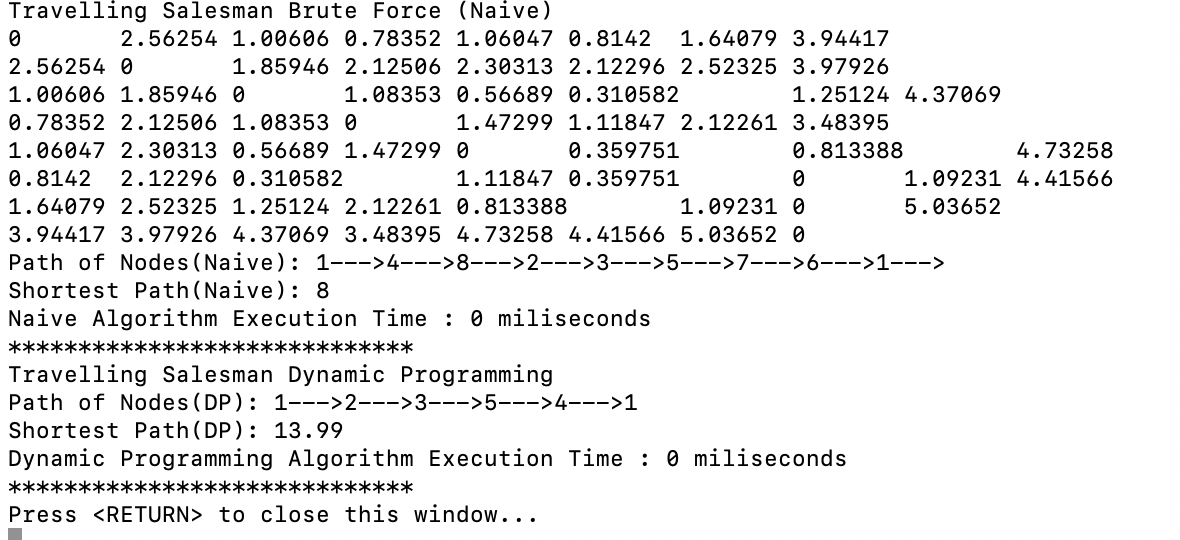
For the results, I noticed the path obtained were different, although not significant, it remained constant for nodes 4,5 and 6 for the dynamic programming. My program sometimes crashes sometimes after 14 nodes. I am not sure why and given enough time, I would analyze the code to decipher the problem. Output it shown for some of the processes run, I believe my program can run up to 12 nodes.

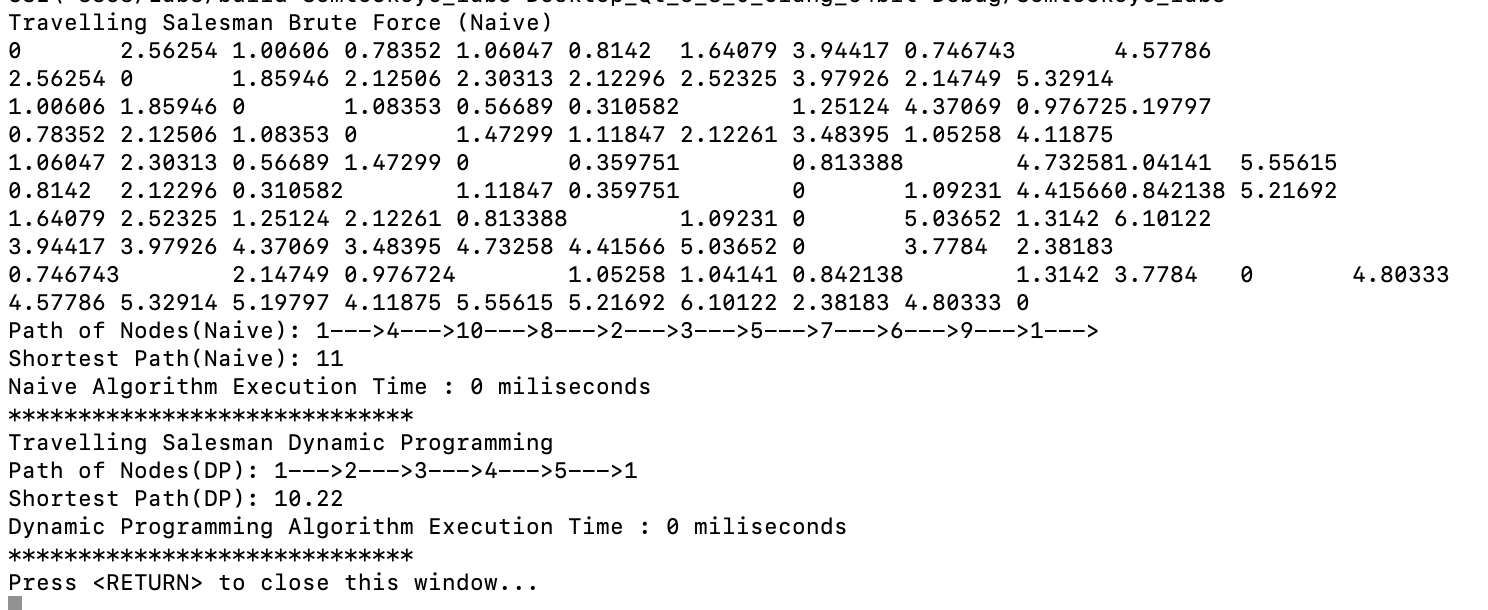
**4 Nodes Output**



**8 Nodes Output**



**10 Nodes Output**

****

For Lab 3, I implemented two algorithms for the travelling salesman problem. The Naïve/Brute Force algorithm and the dynamic programming algorithm. Brute Force Approach takes O (nn) time, because we have to check (n-1)! paths (i.e all permutations) and have to find minimum among them. For the dynamic programming can only be applied after subproblems are obtained. I divided into subproblems which is shown below,

**Recursive Equation**

**For a subset of cities S ⊆ {1, 2,...,n} that includes 1, and j ∈ S, let C(S,j) be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.**

Let us consider a graph ***G = (V, E)***, where ***V*** is a set of cities and ***E*** is a set of weighted edges. An edge ***e (u, v)*** represents that vertices ***u*** and ***v*** are connected. Distance between vertex ***u*** and ***v*** is ***d (u, v)***, which should be non-negative. Suppose we have started at city ***1*** and after visiting some cities now we are in city ***j***. Hence, this is a partial tour. We certainly need to know ***j***, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem. For a subset of cities ***S Є {1, 2, 3, ..., n}*** that includes ***1***, and ***j Є S***, let ***C(S, j)*** be the length of the shortest path visiting each node in **S** exactly once, starting at ***1*** and ending at ***j***. When |***S***| > 1, we define ***C (S, 1)*** = ∝ since the path cannot start and end at **1**.

Now, let express **C (S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that

*C* (*S*,*j*) = *minC* (*S*− { *j* },*I* ) + *d* (*I* ,*j* ) *where I* ∈*S and I* ≠ *j*

**Time Complexity**

We already discussed dynamic programming contains sub problems, here after reaching ith node finding remaining minimum distance to that ith node is a sub-problem. If we solve recursive equation we will get total **(n-1) 2(n-2) sub**-problems, which is**O (n2n)**. Each sub-problem will take **O (n)** time (finding path to remaining **(n-1)** nodes). Therefore, total time complexity is **O (n2n) \* O (n) = O (n22n)** Space complexity is also number of sub-problems which is **O (n2n)** and mentioned earlier, Brute Force Approach takes O (nn) time

For the Project, I implemented the Naïve and Dynamic Programing algorithm for the travelling sales problem. For this project, I used the factory design pattern. For this method, I have static method of a class that returns an object of that class' type. I chose a factory pattern, because not only was it easier for me to implement it given my algorithms for the Naive and Dynamic programming solution for the travelling salesman problem, In my implementation, based on the design, I am able to use the factory method to return existing instancing of specific objects.

For the design, I have an algorithm class which I use as my factory method to hold my static functions as well as some virtual functions that are used in the program. I have two classes for the traveling salesman problem, one class for the Naïve algorithm and another for the dynamic programming algorithm. These two classes implement the algorithm interface class. I read in the file’s separate manner for the naïve and dynamic programming. I noticed the path obtained is different for the dynamic programming and the Naïve algorithm (given enough time, I will go back and analyze code to correct it)

My main class contains a few functions that include the algo\_type function that specifies what kind of algorithm is implemented. The idea behind this, is given more algorithms, it would be somewhat easier to modify and configure, as I just add the necessary classes, implement them with the interface class and the factory method. This provides an easier form for extensibility. Another reason why the factory method was my choice of design pattern for this. I was going to combine the strategy pattern as well, but I had troubles figuring out the concrete class and how to integrate the interface class. A UML of my program is shown below:

