Problem 1 (Traveling Problem) **YASIN AYDIN**

(a) The problem here will be finding the shortest path from the node Istanbul to every other city using minimum time, if such a path exists.
The sub problem can be defined as:
A(i,d) = length of the shortest path from i to d with the rules given in the assignment's guide.

Optimal substructure property:
If a node x lies in the shortest path from a source node s to destination node d then the shortest path from s to d is combination of shortest path from s to x and shortest path from x to d.
Proof by cut and paste.
$mincost(s, d) = minCost(s, i) + minCost(i, d)$ if i exists.

Overlapping computations:
As explained above, if a node x lies in the shortest path from a source node s to destination node d then the shortest path from s to d is combination of shortest path from s to x and shortest path from x to d. So, in the recursive solution the recursive tree will have re-computation of the path from s to x (once while computing **s to d** once when computing **s to x**) at least 2 time according to this example. Hence, we will have repetition which will decrease the run-time.

Defining the problem recursively respecting a topological ordering:

The problems asks us to find shortest path from Istanbul bus station to another station in other city so we can take cities as vertices to consider a DAG but instead I took each city station as vertex. So, starting from the 0th vertex (Istanbul-Harem bus station) to the desired city station (1st vertex is the train station in Istanbul). We have the current location as 0 and if we proceed by increasing the vertex index at the end we will reach the destination. The recursion rolling back from the destination station to start since there is a topological ordering:

$$F_n, F_{n-1}, F_{n-2}, ..., F_1, F_0$$

In the code, we will see a for loop that is ensuring this topological ordering:
for $i = s + 1$ to $d - 1$ **do**
c = minCost(costMatrix, s, i) +minCost(costMatrix, i, d)

The topological ordering ensures that we use the paths only once as it is a rule to use the switch option between the cities only once.

(b) The pseudocode of a naive recursive algorithm: **(Algoritm 1)**

Algorithm 1 Naive Recursive Algorithm

```
1: function MINCOST(costMatrix, s, d)                    ▷ s=start, d=destination
2:     if size of costMatrix <= 2 then
3:         return 0                                      ▷ for the cases of no city and 1 city
4:     end if
5:     if s == d or s + 1 == d then
6:         return costMatrix[s][d]
7:     end if
8:     min ← costMatrix[s][d]
9:     for i = s + 1 to d − 1 do
10:        c ← minCost(costMatrix, s, i) + minCost(costMatrix, i, d)
11:        if c is less than min then
12:            min ← c
13:        end if
14:    end for
15:    return min
16: end function
```

The complexity analysis:
Time complexity is $O(2^V)$, where V is the number of city stations ($citystation = 2 * (cities)$).
The reason is that at each call the function calls itself twice which makes it exponential.
Space complexity is $O(V)$ since the space complexity is the deepest level of recursion an it is the number of city stations.

(c) 1. Pseudocode of the dynamic programming algorithm with Memoization: **(Algoritm 2)**

Algorithm 2 DP-Algorithm with Memoization

```
1: function SHORTESTPATHFINDER(graph, path, sNum, start, destination)
2:     for k = 0 to |V| − 1 do
3:         for i = 0 to |V| − 1 do
4:             for j = 0 to |V| − 1 do
5:                 if graph[i][k] + graph[i][k] is less than graph[i][j] then
6:                     graph[i][j] ← graph[i][k] + graph[i][k]
7:                     path[i][j] ← path[k][i]
8:                 end if
9:             end for
10:        end for
11:    end for
12:    Print the path
13:    return graph[start][destination]    ▷ The shortest path cost from start to destination
14: end function
```

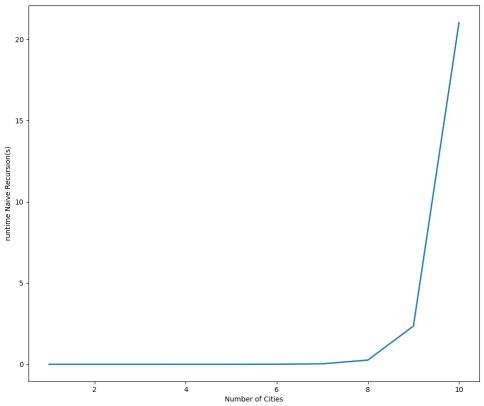
2. The complexity analysis: (The V is the number of cities.)

   i. The asymptotic best/average/worst time complexity is $\Theta(V^3)$ since there is 3 nested for loops with each iterating V times.

   ii. The space complexity is $\Theta(V^2)$ because we only have to hold 2 matrix with city stations. One for the computation of the shortest paths and one for storing the paths. So, we need 2 VxV matrix. $\Theta(2(V)^2) \rightarrow \Theta(V^2)$

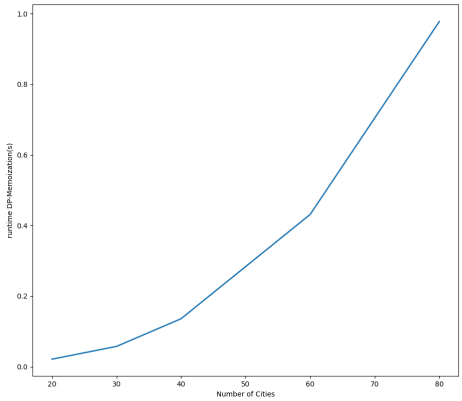
3. Experimental evaluations of these two algorithms:

   i. The time graph of **Algorithm 1** with number of cities $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$:



   As expected, the graph gave us similar curve to $\Theta(2^n)$. The reason is that the algorithm works by calling itself twice each time. I tried up to 12 cities but couldn't get results since it took too much time but the current resulting graph shows the curve clearly.

   ii. The time graph of **Algorithm 2** with number of cities $[20, 30, 40, 60, 80]$:



   As expected, the graph gave us similar curve to $\Theta(n^3)$. The reason is that the algorithm only needs to fill the matrix for each city station which is $\Theta(V^3)$. (I could get 80 cities by changing the default recursion limit to 1500.)

SPECS:
Intel(R) Core(TM) i5-9500H CPU @ Base speed: 3.70GHz, 16.00GB RAM, Windows OS

Problem 2 (Python Code and Benchmarks) **YASIN AYDIN**

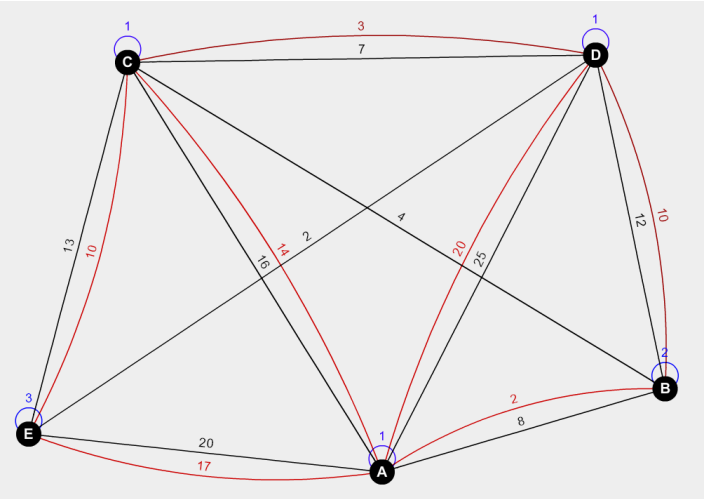
(a) Implementations are in the zip file.

(b) Benchmark suite of at least 5 instances to test the correctness:

Black-box testing:

- All the tested random sets gave correct answers.
- Tested extremes:
  i. 0 city,
  ii. 1 item only,
  iii. 5 city with random costs,
  iv. all costs are 0,
  v. all costs are infinity.

The tested codes are in the related .py files and they are commented out. **Extra:** The result of 5 hand-made city travel map is the following and the result of the path and the cost is correct.




```
Shortest itinerary from Istanbul bus station to nth city costs:
Matrix for the shortest distances between pair of stations:
    0       1       5       3       9       10      14      13      16      18
    1       0       4       2       8       9       13      12      15      17
    5       4       0       2       4       5       9       8       11      14
    3       2       2       0       6       7       11      10      13      16
    9       8       4       6       0       1       5       4       7       10
   10       9       5       7       1       0       4       3       6       9
   14      13       9      11       5       4       0       1       2       5
   13      12       8      10       4       3       1       0       3       6
   16      15      11      13       7       6       2       3       0       3
   18      17      14      16      10       9       5       6       3       0
Path to 4 from Istanbul:
0 ->
1 ->
3 ->
=> 8
(DP-Memoization)Total minimum cost: 16

Process finished with exit code 0
```

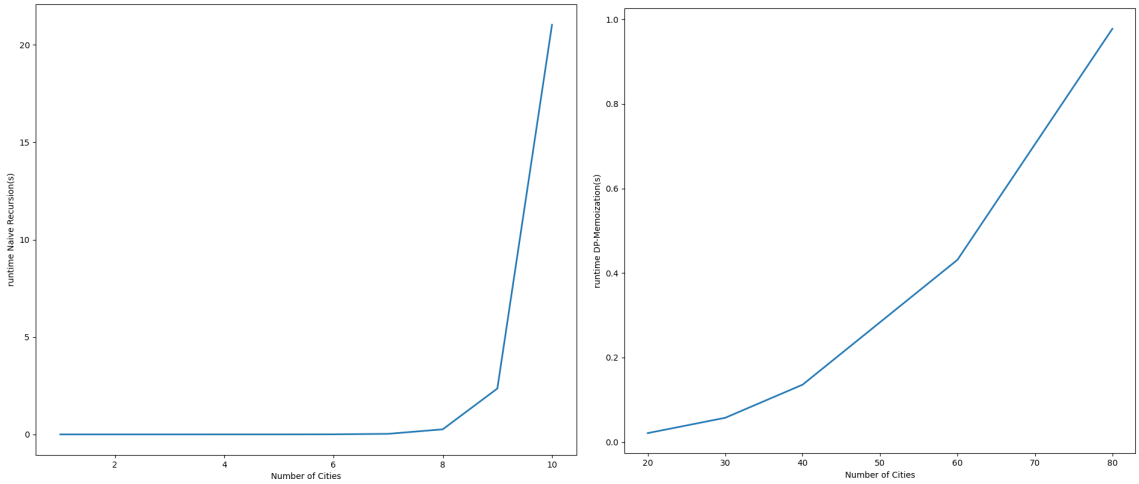
White-box testing:

- When we compare the codes with the algorithm that was demonstrated above they are supposed to work same way and return the desired outputs.
- Also, when the codes were debugged it was seen that codes work as it is supposed to.

According to these tests, we can conclude that both algorithms are correct.

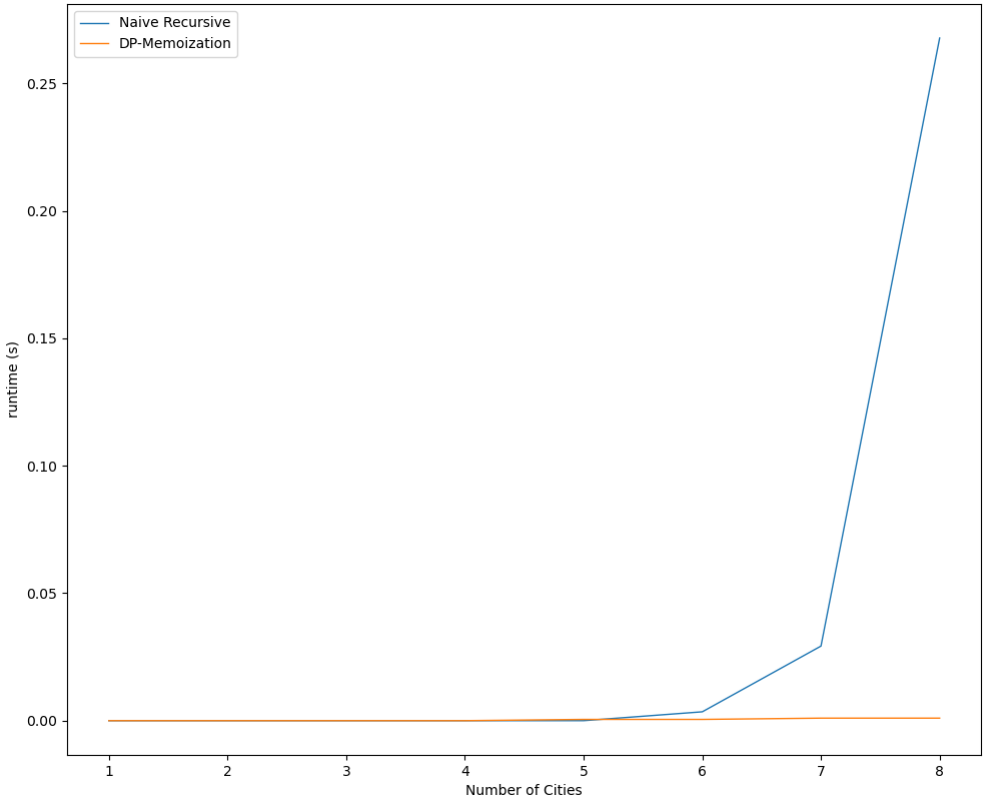
(c) Benchmark suite to test the performance according to run-time:

The time graph of **Algorithm 1** with number of cities $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ and the time graph of **Algorithm 2** with number of cities $[20, 30, 40, 60, 80]$:



These graphs above are giving us their curves with different city numbers but in order to compare the performance we will have to test them with same number of cities and when we do that with **number of city from 1 to 10** we obtain the graph below which shows the enormous difference between them.

The time graph of **Algorithm 1** and **Algorithm 2** with number of cities $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$:



According to these graphs and complexity analyses, we can conclude that using dynamic programming is much more logical for this problem. The performance in time and space is much better than the naive recursive approach.