CST 370 – Spring (B) 2022
Homework 3
Due: 3/22/2022 (Tuesday) (11:55 PM)

How to turn in: Write **three programs** in **either C++ or Java** and submit them on Canvas before the due.

- You **can submit** your programs **multiple times** before the due. However, the **last submission will be used for grading**.
- You have to **submit three programs together**, especially at your last submission. **If you submit, for example, only one program** at the last submission, **we are able to see only that program when we grade** your homework.
- Due time is 11:55(PM). Since there could be a long delay between your computer and Canvas, you should submit it early.
- When you submit your homework program, don't forget to include "Title", "Abstract", "ID", "Name", and "Date".

1. Write a C++ program (or Java program) called **hw3_1.cpp (or hw3_1.java)** that reads the number of input values from a user. Then, read the input values from the user. After that, your program should display them in the ascending order. When you display the numbers, you have to use a short representation if there are consecutive numbers. For example, if there are three numbers such as 51, 52, and 53 in the input values, you have to display them 51–53 to save space. For the assignment, you can assume that the input values are positive integers and unique (= No duplications in the input numbers).

Input format: This is a sample input from a user.

```
6
51
27
53
77
52
75
```

The first number (= 6 in the example) indicates that there will be 6 values in the input. Then, the following lines are the actual values.

Sample Run 0: Assume that a user typed the following lines

```
6
51
27
53
77
52
75
```

This is the correct output.

```
27 51-53 75 77
```

Sample Run 1: Assume that a user typed the following lines

```
7
26
51
27
53
77
52
75
```

This is the correct output.

```
26-27 51-53 75 77
```

Sample Run 2: Assume that a user typed the following lines

```
10
4
5
6
1
2
3
7
9
10
11
```

This is the correct output.

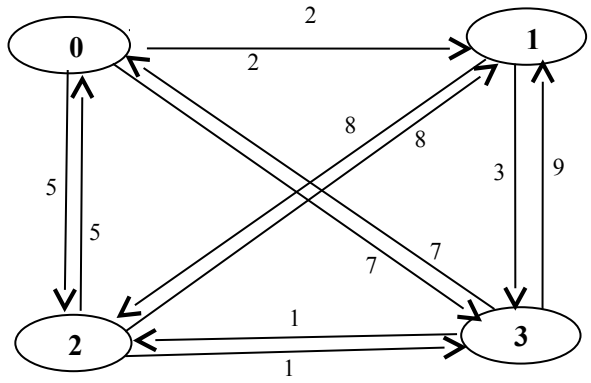
```
1-7 9-11
```

2. Write a C++ (or Java) program called **hw3_2.cpp or (hw3_2.java)** that reads an input graph data from a user. Then, it should present a path for the travelling salesman problem (TSP). In the assignment, you can assume that the **maximum number of vertices** in the input graph is **less than or equal to 15**.

Input format: This is a sample input from a user.

```
4
12
0 1 2
0 3 7
0 2 5
1 0 2
1 2 8
1 3 3
2 0 5
2 1 8
2 3 1
3 0 7
3 1 9
3 2 1
0
```

The first line (= 4 in the example) indicates that there are four vertices in the graph. The next line (= 12 in the example) indicates the number of edges in the graph. The remaining 12 lines are the edge information with the "source vertex", "destination vertex", and "cost". The last line (= 0 in the example) indicates the starting vertex of the travelling salesman problem. This is the graph with the input information provided.



Sample Run 0: Assume that the user typed the following lines

```
4
12
0 1 2
0 3 7
0 2 5
1 0 2
1 2 8
1 3 3
2 0 5
2 1 8
2 3 1
```

```
3 0 7
3 1 9
3 2 1
0
```

This is the correct output. Your program should present the path and total cost in separate lines.

```
Path:0->1->3->2->0
Cost:11
```

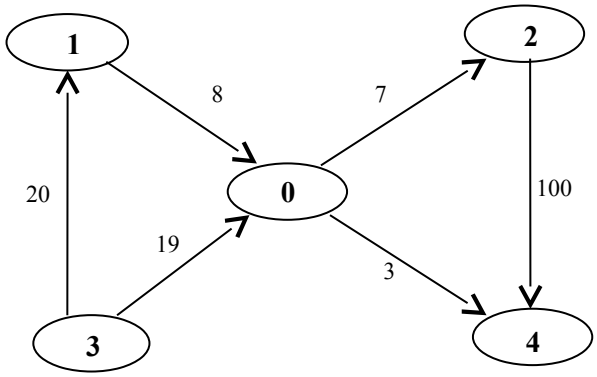
Sample Run 1: Assume that the user typed the following lines

```
5
6
0 2 7
3 1 20
0 4 3
1 0 8
2 4 100
3 0 19
3
```

This is the correct output.

```
Path:
Cost:-1
```

Note that if there is no path for the TSP, your program should **present empty path and -1 cost**.



Sample Run 2: Assume that the user typed the following lines

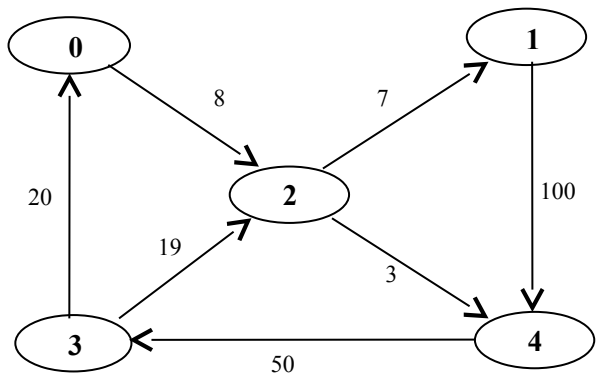
```
5
7
0 2 8
2 1 7
2 4 3
1 4 100
```

```
3 0 20
3 2 19
4 3 50
3
```

This is the correct output of your program.

```
Path:3->0->2->1->4->3
Cost:185
```

This is the directed graph of the input data:



[Hint]: To solve this problem, you can use all permutations of the vertices, except the starting vertex. For example, there are three vertices 1, 2, and 3, in the first sample run, except the starting vertex 0. This is all permutations with the three vertices

1, 2, 3
1, 3, 2
2, 1, 3,
2, 3, 1
3, 1, 2
3, 2, 1

For each permutation, you can calculate the cost. For example, in the case of the permutation "1, 2, 3", add the starting vertex city at the very beginning and end to the permutation which generates "0, 1, 2, 3, 0". This list corresponds to the path "0 ☾ 1 ☾ 2 ☾ 3 ☾ 0". Therefore, the cost of this path is "18". Similarly, the next permutation (= 1, 3, 2) corresponds to "0 ☾ 1 ☾ 3 ☾ 2 ☾ 0". The cost is 11. This way, you can check all possible paths for the TSP.

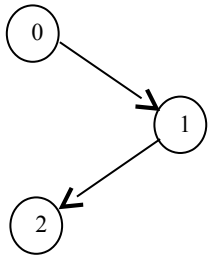
This is a sample C++ program to display permutations: https://repl.it/@YBYUN/permutationsusingSTL
And, this is a sample Java program: https://replit.com/@YBYUN/JavaPermutations

3. Write a C++ (or Java) program called **hw3_3.cpp (**or **hw3_3.java)** that implements the ***Depth-First Search (DFS) algorithm*** using a stack and a mark array as you learned in the class.

Input format: This is a sample input from a user.

```
3
2
0 1
1 2
```

The first line (= 3 in the example) indicates that there are three vertices in the graph. For the homework, you can assume that the first vertex starts from the number 0. The second line (= 2 in the example) represents the number of edges, and following two lines are the edge information. This is the graph with the input information.



Sample Run 0: Assume that the user typed the following lines

```
3
2
0 1
1 2
```

This is the correct output. Your program should display the mark array of DFS. For the problem, you can **assume that the starting vertex is always 0**. And also, you can **assume that the graph is connected**.

```
Mark[0]:1
Mark[1]:2
Mark[2]:3
```

Sample Run 1: Assume that the user typed the following lines

```
5
6
0 1
0 2
0 3
1 3
2 3
3 4
```

This is the correct output.

```
Mark[0]:1
Mark[1]:2
Mark[2]:5
Mark[3]:3
Mark[4]:4
```

Sample Run 2: Assume that the user typed the following lines

```
5
6
0 1
0 2
0 3
1 4
2 3
3 4
```

This is the correct output.

```
Mark[0]:1
Mark[1]:2
Mark[2]:4
Mark[3]:5
Mark[4]:3
```

[Hint] In the lecture, we used a stack for the operation of DFS algorithm. So, you can use an explicit stack to implement it. However, you can use a recursive function which uses an implicit stack. The following is the pseudocode with a recursive function and it may make your implementation easier.

// This is the main function named DFS().

ALGORITHM *DFS*(G)

```
//Implements a depth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//          in the order they are first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)    // call dfs() function
```

// This is the recursive function named dfs().
// Don't be confused with the main function DFS().

```
dfs(v)
//visits recursively all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are encountered
//via global variable count
count ← count + 1;   mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)   // call dfs() function
```