Preliminary: Priority queue

(*This part should not take more than 45 minutes, preferably 15 minutes*)

1. Download the simplePriorityQueue.py code from Class Materials

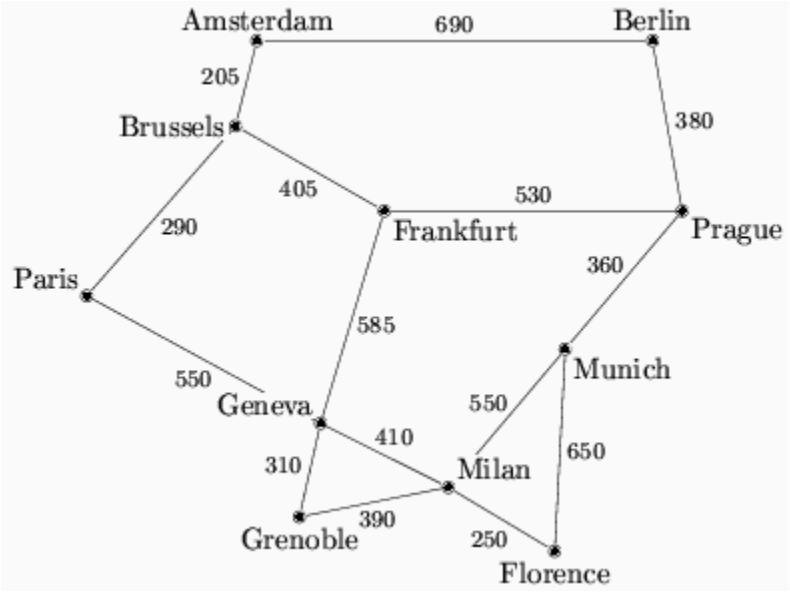
2. Learn how to use the Simple_Priority_Queue class. Examples are already given in the code (commented). To import Simple_Priority_Queue to your program, place simplePrioirtyQueue.py in the same folder as your program, the add the following line.

   from simplePriorityQueue import Simple_Priority_Queue

NOTE:

This priority queue is simply the *heap* data structure in which you are supposed to have studied from Data Structure and Algorithms course. Therefore, you are assumed to have adequate familiarity with it.

PROBLEM A: SHORTEST ROUTE ON MAP



Find the shortest path from a city to a destination e.g. from Amsterdam to Florence.

INPUT:

- The first line is the number of cities, n.
- Given that cities are represented by numbers, starting from 0, each following line of the input file gives information of an edge. For example, a line containing 5 3 305 means that there is an edge from city 5 to city 3 (and vice versa) with the distance 305 km.

OUTPUT: One number, the shortest total route from city 0 to city n-1.

Test cases: Example_maps.zip (already in the Class Materials)

1) Use the following code to read the input map file, which will store the map as a collection of adjacency lists.

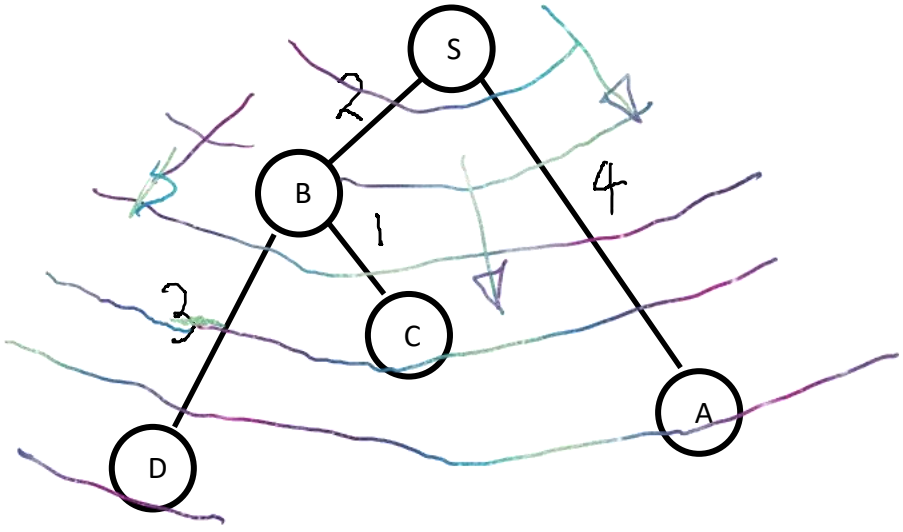
```
from sys import stdin

INF = 1000000000
V = int(input())
adj = [[] for i in range(V)]

for line in stdin:
    x = line.split()
    u = int(x[0])
    v = int(x[1])
    w = int(x[2])
    adj[u].append((v,w))
    adj[v].append((u,w))
```

KEEP IN MIND that a map is a *weighted* graph. Therefore, the information stored in the adjacency lists is not only adjacencies, but also the weight of each adjacency (in this case, the distance).

2) The technique to solve map problem is a generalization of Breadth-First Search (BFS), called ***Uniform Cost Search (UCS)***.
BFS is applicable, for finding solution in minimal number of steps, only when every search to reach successor state is of <u>equal cost</u>. For example, in maze, every search to the adjacent position takes 1 step, equally.
When each search does not necessary cost equally, UCS is an optimized form of continuous BFS. UCS expands the search outward from the state with smaller *total* cost to the state with larger *total* cost.



In this example, if the search expands uniformly from S, state B will be reached first (cost = 2), then C (cost = 3, from S), A (cost = 4), D (cost = 5, from S), respectively.

There is no need to sweepingly scan the search outward, because the cost of reaching states A, B, C, and D, can establish the order of states to be searched.

Essentially, UCS is simply BFS that operates on priority-queue instead of First-In-First-Out queue. The key of each state is the total cost of reaching that state, from the initial state. Consequently, the priority-queue will always dequeue the state whose total cost is the smallest.

Write a program that solves the "shortest route on map" problem using UCS technique.

Hint The successor states of state u are those indicated in adj[u].

Alternative explanation of UCS (a lecturing video):
https://www.youtube.com/watch?v=dRMvK76xQJI

3) [OPTIONAL]
This step is not required for the course. It is, however, suggested for students who complete the solution early in the class.

The more advanced solution will utilize a more advanced version of priority queue that allows the priority of an element to be elevated. The advanced priority queue module, priorityqueue.py, is provided in the Class Materials as well.

With this advanced priority queue, only one copy of a state is required to be stored in the priority queue, and when shorter path to such a state is discovered, the key of that state can be updated while the position of that state in the priority queue is also updated accordingly.