/ 20 P

Programming Task P2.

Enrollment Key: `ExamTime2019`

Submission: see Section 3 of the Technical Guide

Structure

In this exercise, you should implement an algorithm to find the minimum spanning tree (MST) for a given graph $G = (V, E)$, where G is a *connected*, *weighted*, *undirected* and *simple* graph. One way of finding the MST of a given graph is by using Kruskal's algorithm. Your task is to complete the implementation of the following method:

- **`kruskal(`**G**`)`**: finds a minimum spanning tree for a given graph $G = (V, E)$, and outputs its cost, i.e., the sum of the weights of all edges in the computed MST.

We provide a template for this exercise (along with code to read the input and write the cost of the MST to the output) where you can provide the implementation of the algorithm.

A successful implementation of Kruskal's algorithm depends on a correct and efficient implementation of a **Union-Find** structure. A Union-Find data structure maintains a *family* of N disjoint nonempty sets (one set per connected component). Each set in the family has one designated element called its *label* and is identified by that label. The data structure supports the following three operations:

1. **`create(`**N**`)`**: creates a new set $\{x\}$ for all $x \in \{0, \ldots, N-1\}$ and adds it to the family.
2. **`union(`**x**`, `** y**`)`**: changes the family by replacing two sets, the one containing x and the one containing y, by a single set that is the union of these two sets. The label of the union is either `find(`x`)` or `find(`y`)`.
3. **`find(`**x**`)`**: returns the label of the set containing x.

In the code template we also provide a correct, but **inefficient** implementation of the Union-Find structure. To obtain full points for this exercise, your task is to modify the implementation of this structure such that:

- **`find(`**x**`)`** and **`union(`**x**`, `** y**`)`** operate in time $O(\log(N))$ (amortized over one run of the algorithm), where N is the number of elements stored in the family.
- **`create(`**N**`)`** initializes the structure and any additional fields that might be used in the `find(`x`)` and `union(`x`, `y`)` routines. The modified routine should operate in time $O(N)$, where N is number of elements stored in the family.

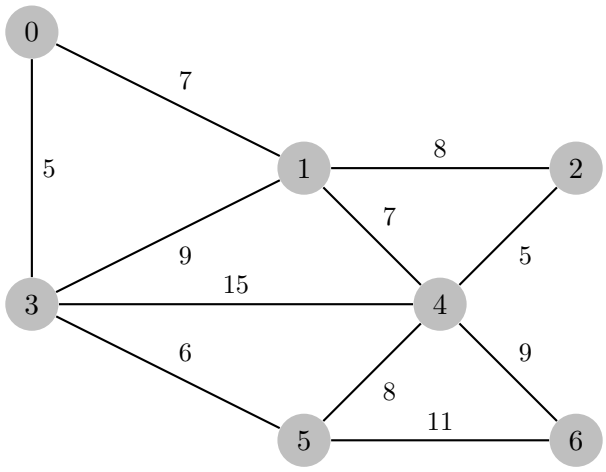
The provided template is tailored towards the implementation of the Kruskal's algorithm. However, any other algorithm that will compute the MST in $O(|E| \cdot \log(|V|))$ will also be accepted, assuming it passes all automatic tests.

To validate the correctness of your implementation, the cost of the computed MST is written to the output. Your implementation can assume $|V|$ vertices, such that $2 \leq |V| \leq 10^6$, and $|E|$ edges such

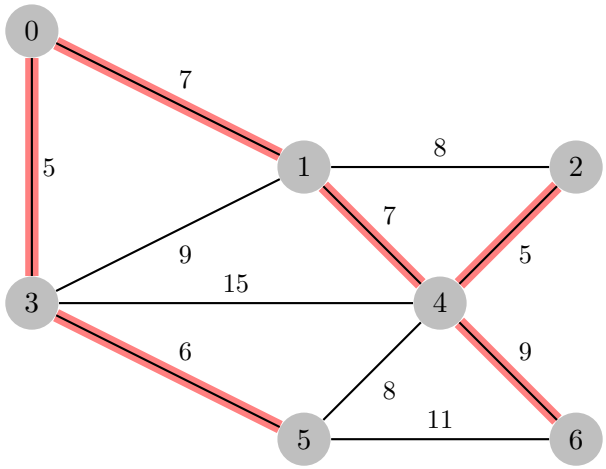
that $|V| - 1 \leq |E| \leq 10^6$. The weight of each edge is specified with an integer number w such that $1 \leq w \leq 10^6$.

Example

Consider the graph below with 7 vertices and 11 edges (weights are specified next to each edge in the graph):



A minimum spanning tree is shown with the highlighted edges:



The cost of the tree is the sum of edges $\{0,3\}, \{3,5\}, \{0,1\}, \{1,4\}, \{2,4\}$ and $\{4,6\}$, which is $5 + 6 + 7 + 7 + 5 + 9 = 39$.

Grading

Overall, you can obtain a maximum of 20 judge points for this programming task. To get full points your program should require $O(|E| \cdot \log(|V|))$ time to compute the cost of the MST of the graph (with reasonable hidden constants).

Slower solutions can obtain partial points, namely an $O(|E| \cdot |V|)$ solution can obtain up to 10 points (with reasonable hidden constants). This solution can be achieved by implementing Kruskal's algorithm with the provided Union-Find data-structure.

Instructions and assistance for programming task P2

For this exercise, we provide a program template as an Eclipse project in your workspace that helps you reading the input and writing the output. Importing any additional Java class is **not allowed** (with the exception of the already imported ones `java.io.{InputStream, OutputStream}` and `java.util.{Arrays, Comparator, Scanner}` class).

In this exercise we permit the use of `java.util.Arrays` library, that provides various methods for manipulating arrays (such as sorting and searching). One such function is the `Arrays.sort` method that provides efficient sorting based on merge-sort in $O(N \cdot \log(N))$ time:

```
Arrays.sort(T[] a, Comparator<? super T> c)
```

The methods works such that it sorts the specified array of objects according to the order induced by the specified comparator. To illustrate usage, consider the class *Employee* that represents employees with their name, age and salary:

```
public class Employee {
  public String name;
  public int age;
  public int salary;
  public Employee(String name, int age, int salary) {
    this.name   = name;
    this.age    = age;
    this.salary = salary;
  }
}
```

To sort the array of employees `listOfEmployees` by their salaries, we can invoke the method providing a `Comparator` instance, that compares the salaries:

```
public void sortEmployeesBySalary (Employee[] listOfEmployees) {
  //
  // Create an instance of Comparator for a given Employee class
  //
  Comparator<Employee> employeeComparator = new Comparator<Employee>() {
    //
    // Implement the comparison routine of two Employee instances
    //
    public int compare(Employee e1, Employee e2) {
      return e1.salary - e2.salary;
    }
  }
  //
  // Use the Comparator instance to perform the sorting.
  //
  Arrays.sort(listOfEmployees, employeeComparator);
}
```

Use the `Array.sort` or any other available method in `java.util.Arrays` on your convenience.

The project also contains data for your local testing and a `JUnit` program that runs your `Main.java` on all the local tests – just open and run `StructureTest.launch` in the project. The local test

data are different and generally smaller than the data that are used in the online judge.

Submit only your `Main.java`.

The input and output are handled by the template – you should not need the rest of this text.

Input The input of this problem consists of a number of test-cases. The first line contains T, the number of test-cases. Each of the T cases is independent of the others, and contains several lines:

1. The first line of each test case contains the number of vertices in the graph $|V| \in \{2, \ldots, 10^6\}$ and the number of edges $|E| \in \{|V| - 1, \ldots, 10^6\}$.
2. Then it is followed by E lines, such that each contains 3 integer numbers: u, v and w. The numbers u, $v \in \{0, \ldots, |V| - 1\}$ (such that $u \neq v$) represent the end points of an edge and $w \in \{1, \ldots, 10^6\}$ represents its weight. As we have undirected and simple graph each (u, v) pair is commutative (i.e. (u, v) and (v, u) represent the same edge $\{u, v\}$), and each pair is distinct (the input does not contain the same edge more than once).

Output For every case, the output is the cost of the minimum spanning tree of the input graph.

The output contains one line for each test-case. More precisely, the i-th line of the output contains an integer number that represents the cost of the minimum spanning tree of the input graph. The output is terminated with an end-line character.

Example input:

```
2
7 11
1 0 7
2 1 8
3 0 5
3 1 9
4 1 7
4 2 5
4 3 15
5 3 6
5 4 8
6 4 9
6 5 11
4 6
0 1 3
0 2 2
0 3 5
1 2 4
1 3 6
2 3 1
```

Example output:

```
39
6
```