PROBLEM

Possible Paths in a Tree

Hard Accuracy: 74.46% Submissions: 12K+ Points: 8

Given a **weighted** tree with **n** nodes and (**n-1**) edges. You are given **q queries**. Each query contains a number **x**. For each query, find the number of paths in which the maximum edge weight is less than or equal to **x**.

Note: Path from A to B and B to A are considered to be the same.

Example 1:

Input:
n = 3
edges {start, end, weight} = {{1, 2, 1}, {2, 3, 4}}
q = 1
queries[] = {3}
Output:
1
Explanation:
Query 1: Path from 1 to 2

Example 2:

Input:
n = 7
edges {start, end, weight} = {{1, 2, 3}, {2, 3, 1}, {2, 4, 9}, {3, 6, 7}, {3, 5, 8}, {5, 7, 4}}
q = 3
queries[] = {1, 3, 5}
Output:
1 3 4
Explanation:
Query 1: Path from 2 to 3
Query 2: Path from 1 to 2, 1 to 3, and 2 to 3
Query 3: Path from 1 to 2, 1 to 3, 2 to 3, and 5 to 7

Your Task:
You don't need to read input or print anything. Complete the function **maximumWeight()**which takes integers **n,** list of **edges** where each edge is given by {start,end,weight}, an integer **q** and a list of q **queries** as input parameters and returns a list of integers denoting the number of possible paths for each query.

Expected Time Complexity: O(nlogn + qlogn)
Expected Auxiliary Space: O(n)

Constraints:
$2 \le n \le 10^4$
$1 \le q \le 10^4$
$1 \le$ edges[i][0], edges[i][1] $\le n$
edges[i][0] != edges[i][1]
$0 \le$ edges[i][2] $\le 10^5$
$0 \le$ queries[i] $\le 10^5$

CODE

```
class Solution {
   int[] par;
   int[] rank;
   ArrayList<Integer> maximumWeight(int n, int edges[][], int q, int queries[]) {
      ArrayList<Integer> res = new ArrayList<>();

      for(int i=0;i<q;i++){
         res.add(0);
      }



      par = new int[n+1];
      rank = new int[n+1];
      Arrays.fill(par,-1);
      Arrays.fill(rank,1);

      ArrayList<Edge> edgesLs = new ArrayList<>();



      for(int i=0;i<edges.length;i++){
         edgesLs.add(new Edge(edges[i][0],edges[i][1],edges[i][2]));
      }

      Collections.sort(edgesLs,(e1,e2)->e1.w-e2.w);

      ArrayList<Query> queryLs = new ArrayList<>();

      for(int i=0;i<queries.length;i++){
         queryLs.add(new Query(i,queries[i]));
      }
```

```
    Collections.sort(queryLs, (q1,q2)-> q1.q - q2.q);
    int index = 0;
    int inter = 0;
    for(Query que: queryLs){

        while(index < edgesLs.size()){
            Edge e = edgesLs.get(index);

            if(e.w <= que.q){
                inter += merge(e.s,e.e);
            }
            else{
                break;
            }
            index++;
        }

        res.set(que.p,inter);
    }

    return res;
}

int findParent(int n){
    if(par[n] == -1){
        return  n;
    }

    return findParent(par[n]);
}

int merge(int n1, int n2){
    int p1 = findParent(n1);
    int p2 = findParent(n2);

    int res = rank[p1]*rank[p2];
    par[p2] = p1;
    rank[p1] += rank[p2];
    return res;
}
```

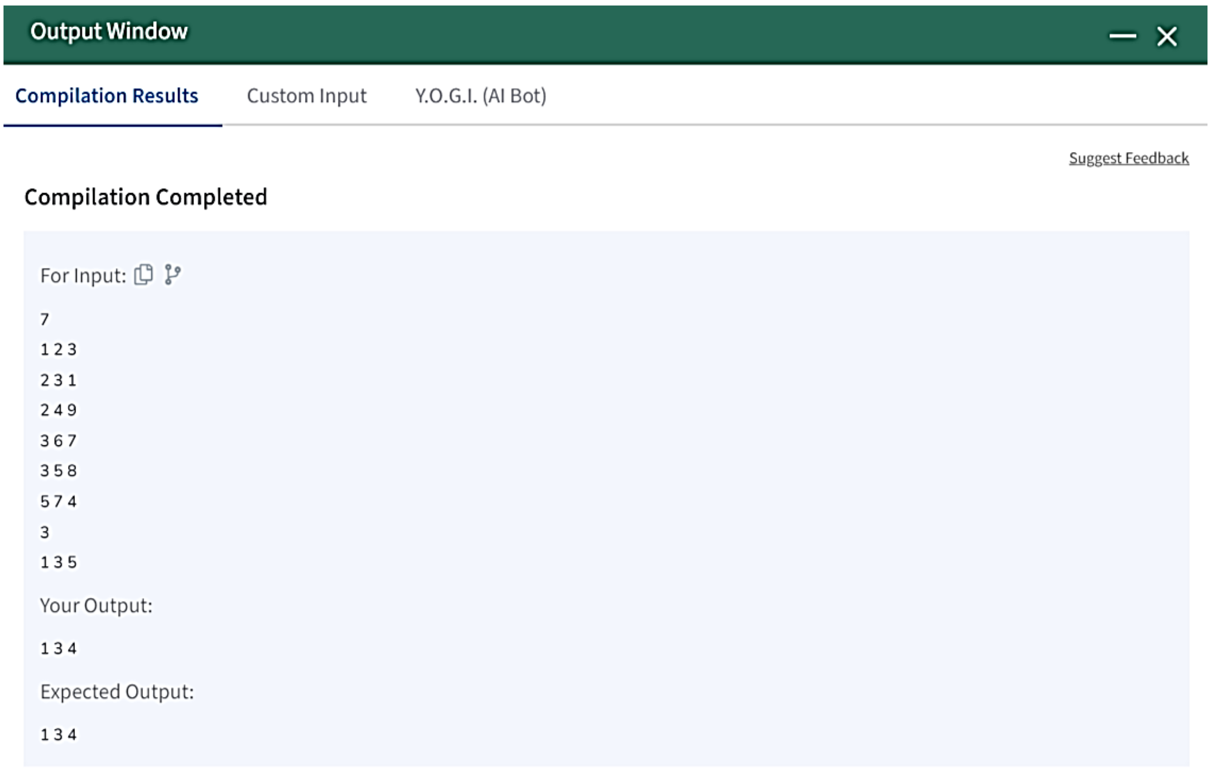
```
    class Query {
        int q;
        int p;

        public Query(int p, int q){
            this.p = p;
            this.q = q;
        }
    }

    class Edge {
        int s;
        int e;
        int w;

        public Edge(int s, int e, int w){
            this.s = s;
            this.e = e;
            this.w = w;
        }
    }
}
```

OUTPUT



Compilation Completed

For Input:

```
7
1 2 3
2 3 1
2 4 9
3 6 7
3 5 8
5 7 4
3
1 3 5
```

Your Output:

```
1 3 4
```

Expected Output:

```
1 3 4
```

EXPLANATION

Initialization:

Two arrays par and rank are created to keep track of parent nodes and set ranks respectively.
An ArrayList res is created to store the results of each query.

```
int[] par; // Array to store parent nodes
int[] rank; // Array to store set ranks
ArrayList<Integer> res = new ArrayList<>(); // List to store query results
```

Disjoint Set Initialization:
par and rank arrays are initialized with appropriate values.
-1 indicates that each node is initially its own parent (representative of its set), and 1 indicates the initial rank of each set.

```
par = new int[n + 1];
rank = new int[n + 1];
Arrays.fill(par, -1); // Initialize parent array with -1
Arrays.fill(rank, 1); // Initialize rank array with 1
```

Edge and Query Processing:

Edges are sorted by weight, and queries are sorted by value.
For each query, edges are processed until their weight exceeds the query value.
The merge method merges sets of nodes connected by the current edge, updating the parent and rank arrays, and accumulating the count of edges for each query.

```
ArrayList<Edge> edgesLs = new ArrayList<>();
ArrayList<Query> queryLs = new ArrayList<>();

// Sorting edges by weight
Collections.sort(edgesLs, (e1, e2) -> e1.w - e2.w);

// Sorting queries by value
Collections.sort(queryLs, (q1, q2) -> q1.q - q2.q);

// Processing queries
for (Query que : queryLs) {
   while (index < edgesLs.size()) {
      Edge e = edgesLs.get(index);
      if (e.w <= que.q) {
         inter += merge(e.s, e.e); // Merging sets of nodes connected by the edge
      } else {
         break;
      }
      index++;
```

```
    }
    res.set(que.p, inter); // Storing the result for the current query
}
```

Disjoint Set Operations:

The findParent method finds the representative (parent) of a node in its set recursively.
The merge method merges two sets represented by nodes n1 and n2, updating the parent and rank arrays accordingly.

```
int findParent(int n) {
    if (par[n] == -1) {
        return n; // Node n is its own parent (representative)
    }
    return findParent(par[n]); // Recursively find parent
}

int merge(int n1, int n2) {
    int p1 = findParent(n1); // Find parent of n1
    int p2 = findParent(n2); // Find parent of n2
    // Merging sets represented by p1 and p2
    par[p2] = p1; // Update parent of p2 to p1
    rank[p1] += rank[p2]; // Update rank of p1 with rank of p2
    return rank[p1] * rank[p2]; // Return the size of the merged set
}
```

Inner Classes:

Query class represents a single query with two attributes: p for the index of the query result and q for the query value.
Edge class represents a single edge with three attributes: s and e for the nodes it connects, and w for its weight.

```
class Query {
    int q; // Query value
    int p; // Index of the query result
    public Query(int p, int q) {
        this.p = p;
        this.q = q;
    }
}

class Edge {
    int s; // Start node of the edge
    int e; // End node of the edge
    int w; // Weight of the edge
    public Edge(int s, int e, int w) {
        this.s = s;
        this.e = e;
        this.w = w;
    }
}
```

Result Storage:
The result for each query is stored in the res ArrayList.
res.set(que.p, inter); // Store the result for the current query
In summary, the code uses a disjoint set union (DSU) data structure to efficiently process queries on a weighted tree. It iterates through the sorted queries and processes edges whose weight is less than or equal to the query value, merging sets of connected nodes and accumulating the count of such edges for each query.