1168. Optimize Water Distribution in a Village

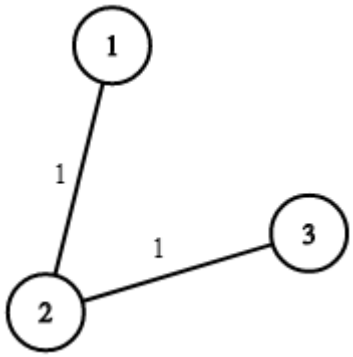
Description

There are `n` houses in a village. We want to supply water for all the houses by building wells and laying pipes.

For each house `i`, we can either build a well inside it directly with cost `wells[i]`, or pipe in water from another well to it. The costs to lay pipes between houses are given by the array `pipes`, where each `pipes[i] = [house1, house2, cost]` represents the cost to connect `house1` and `house2` together using a pipe. Connections are bidirectional.

Find the minimum total cost to supply water to all houses.

Example 1:



Input: n = 3, wells = [1,2,2], pipes = [[1,2,1],[2,3,1]]

Output: 3

Explanation:

The image shows the costs of connecting houses using pipes.

The best strategy is to build a well in the first house with cost 1 and connect the other houses to it with cost 2 so the total cost is 3.

Constraints:

- `1 <= n <= 10000`
- `wells.length == n`
- `0 <= wells[i] <= 10^5`

- `1 <= pipes.length <= 10000`
- `1 <= pipes[i][0], pipes[i][1] <= n`
- `0 <= pipes[i][2] <= 10^5`
- `pipes[i][0] != pipes[i][1]`

Solution

Create a virtual node with number 0, which connects to each of the `n` nodes with cost `wells[i]` if the house number is `i + 1`. Then the problem becomes a minimum spanning tree problem.

Use Kruskal's algorithm to obtain a minimum spanning tree. First use a new array to store all the edges, including the newly added edges, and sort the edges according to weights in ascending order. Then loop over the edges in sorted order. For each edge, if the two nodes are not in the same component, then connect them and add the cost to the total cost, with the nodes' components updated. Finally, return the total cost.

```
class Solution {
    public int minCostToSupplyWater(int n, int[] wells, int[][] pipes) {
        int pipesCount = pipes.length;
        int newPipesCount = pipesCount + n;
        int[][] newPipes = new int[newPipesCount][3];
        for (int i = 0; i < pipesCount; i++) {
            int[] pipe = pipes[i];
            int start = pipe[0], end = pipe[1], cost = pipe[2];
            if (start > end) {
                int temp = start;
                start = end;
                end = temp;
            }
            newPipes[i][0] = start;
            newPipes[i][1] = end;
            newPipes[i][2] = cost;
        }
        for (int i = 0; i < n; i++) {
            newPipes[pipesCount + i][0] = 0;
            newPipes[pipesCount + i][1] = i + 1;
            newPipes[pipesCount + i][2] = wells[i];
        }
        Arrays.sort(newPipes, new Comparator<int[]>() {
            public int compare(int[] pipe1, int[] pipe2) {
                return pipe1[2] - pipe2[2];
            }
        });
```

```
        Map<Integer, Integer> nodeGroupMap = new HashMap<Integer, Integer>
();
        Map<Integer, Set<Integer>> groupNodesMap = new HashMap<Integer,
Set<Integer>>();
        for (int i = 0; i <= n; i++) {
            nodeGroupMap.put(i, i);
            Set<Integer> set = new HashSet<Integer>();
            set.add(i);
            groupNodesMap.put(i, set);
        }
        int totalCost = 0;
        for (int i = 0; i < newPipesCount; i++) {
            int start = newPipes[i][0], end = newPipes[i][1], cost =
newPipes[i][2];
            int group1 = nodeGroupMap.get(start), group2 =
nodeGroupMap.get(end);
            if (group1 != group2) {
                totalCost += cost;
                Set<Integer> set1 = groupNodesMap.getOrDefault(group1, new
HashSet<Integer>());
                Set<Integer> set2 = groupNodesMap.getOrDefault(group2, new
HashSet<Integer>());
                if (group1 < group2) {
                    for (int node : set2)
                        nodeGroupMap.put(node, group1);
                    set1.addAll(set2);
                    groupNodesMap.put(group1, set1);
                    groupNodesMap.remove(group2);
                } else {
                    for (int node : set1)
                        nodeGroupMap.put(node, group2);
                    set2.addAll(set1);
                    groupNodesMap.put(group2, set2);
                    groupNodesMap.remove(group1);
                }
            }
        }
        return totalCost;
    }
}
```