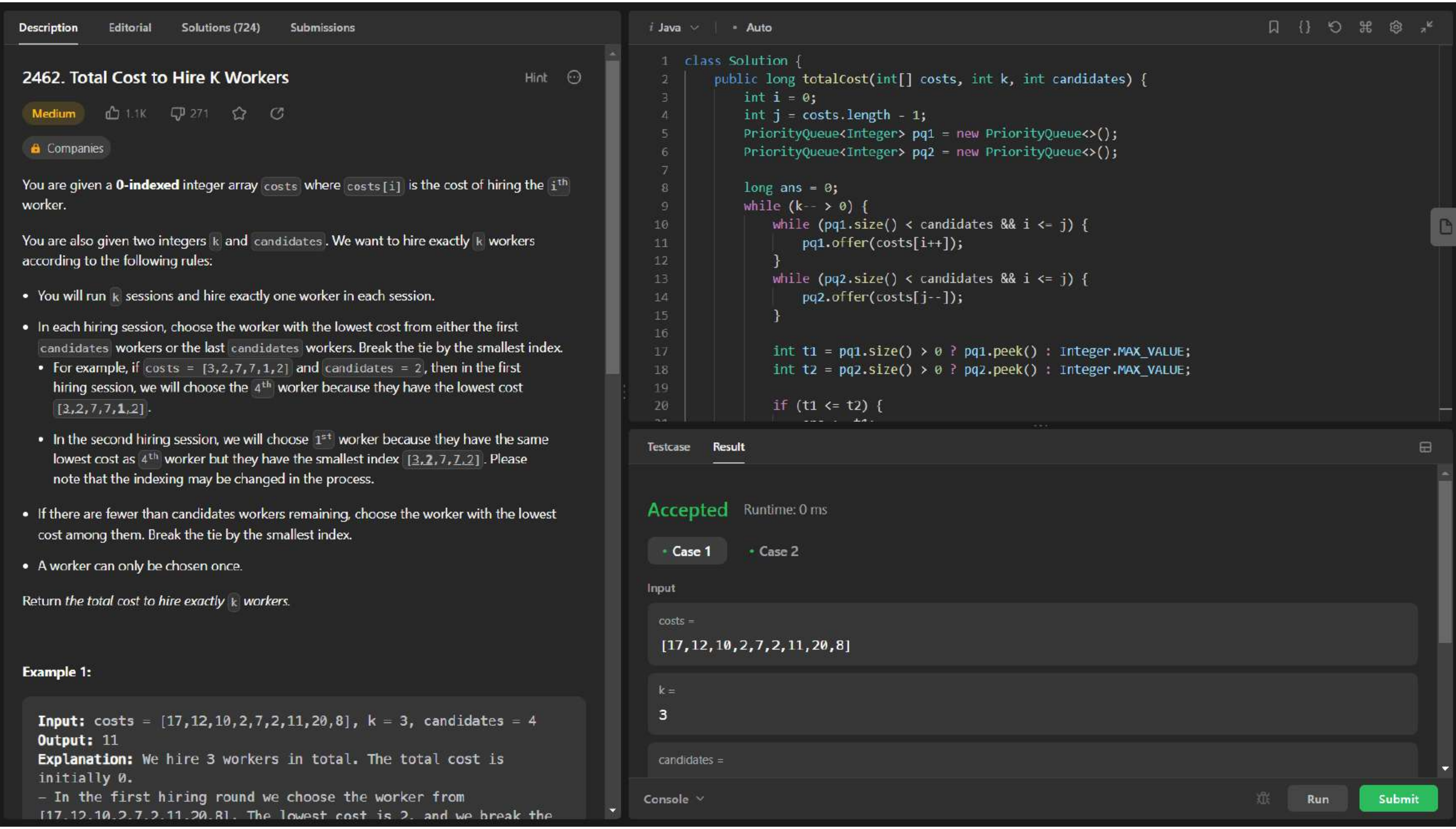


2462. Total Cost to Hire K Workers

You are given a **0-indexed** integer array `costs` where `costs[i]` is the cost of hiring the i^{th} worker.

You are also given two integers `k` and `candidates`. We want to hire exactly `k` workers according to the following rules:

- You will run `k` sessions and hire exactly one worker in each session.
- In each hiring session, choose the worker with the lowest cost from either the first `candidates` workers or the last `candidates` workers. Break the tie by the smallest index.
  - For example, if `costs = [3,2,7,7,1,2]` and `candidates = 2`, then in the first hiring session, we will choose the 4^{th} worker because they have the lowest cost `[3,2,7,7,1,2]`.
  - In the second hiring session, we will choose 1^{st} worker because they have the same lowest cost as 4^{th} worker but they have the smallest index `[3,2,7,7,2]`. Please note that the indexing may be changed in the process.
- If there are fewer than candidates workers remaining, choose the worker with the lowest cost among them. Break the tie by the smallest index.
- A worker can only be chosen once.

Return *the total cost to hire exactly* `k` *workers.*

Example 1:

```
Input: costs = [17,12,10,2,7,2,11,20,8], k = 3, candidates = 4
Output: 11
Explanation: We hire 3 workers in total. The total cost is
initially 0.
- In the first hiring round we choose the worker from
[17,12,10,2,7,2,11,20,8]. The lowest cost is 2, and we break the
```

```java
class Solution {
    public long totalCost(int[] costs, int k, int candidates) {
        int i = 0;
        int j = costs.length - 1;
        PriorityQueue<Integer> pq1 = new PriorityQueue<>();
        PriorityQueue<Integer> pq2 = new PriorityQueue<>();

        long ans = 0;
        while (k-- > 0) {
            while (pq1.size() < candidates && i <= j) {
                pq1.offer(costs[i++]);
            }
            while (pq2.size() < candidates && i <= j) {
                pq2.offer(costs[j--]);
            }

            int t1 = pq1.size() > 0 ? pq1.peek() : Integer.MAX_VALUE;
            int t2 = pq2.size() > 0 ? pq2.peek() : Integer.MAX_VALUE;

            if (t1 <= t2) {
```

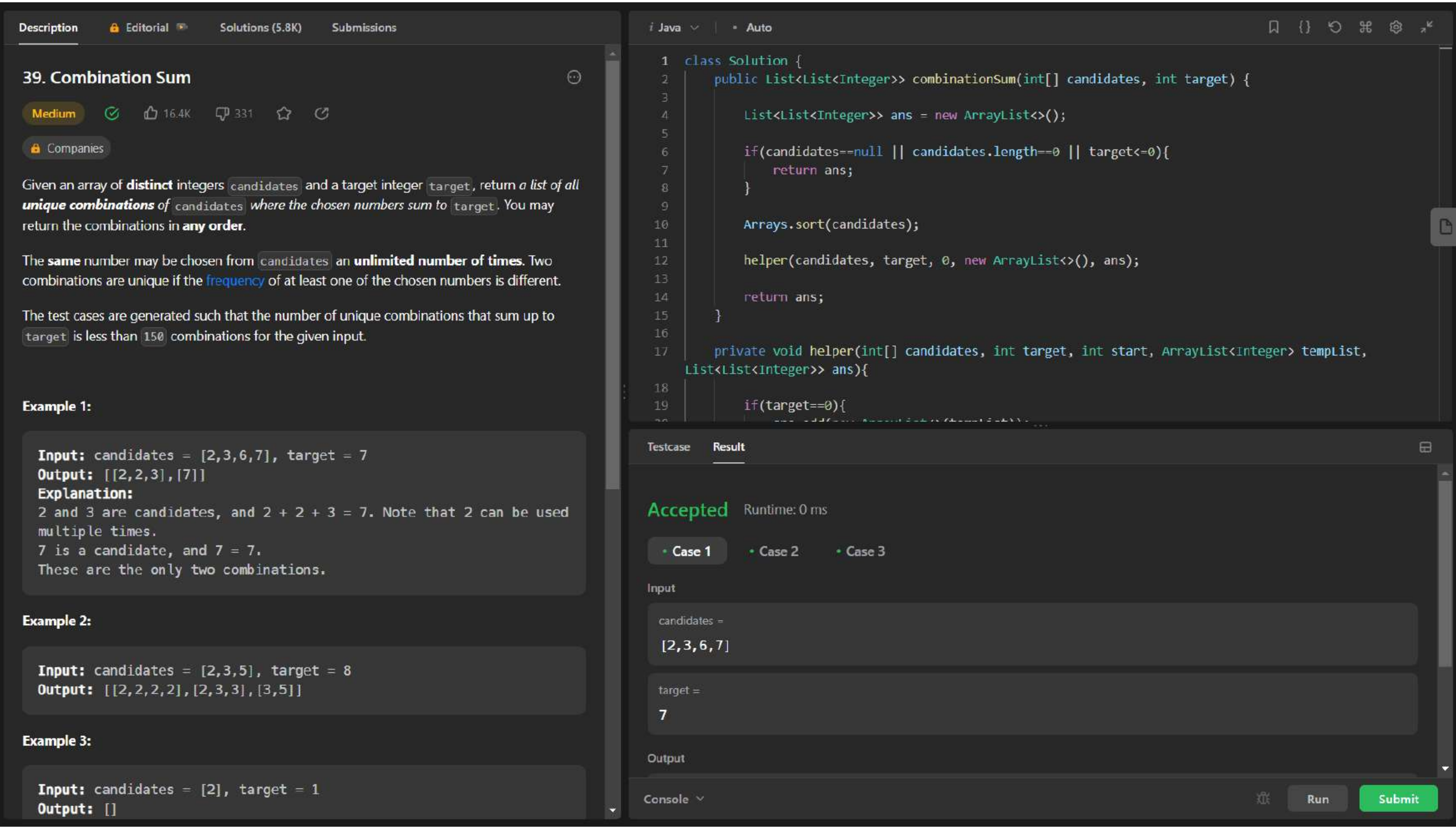
Accepted Runtime: 0 ms

Input

costs =
[17,12,10,2,7,2,11,20,8]

k =
3

candidates =



39. Combination Sum

Given an array of **distinct** integers `candidates` and a target integer `target`, return *a list of all **unique combinations** of* `candidates` *where the chosen numbers sum to* `target`. You may return the combinations in **any order**.

The **same** number may be chosen from `candidates` an **unlimited number of times**. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

The test cases are generated such that the number of unique combinations that sum up to `target` is less than `150` combinations for the given input.

Example 1:

```
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
Explanation:
2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used
multiple times.
7 is a candidate, and 7 = 7.
These are the only two combinations.
```

Example 2:

```
Input: candidates = [2,3,5], target = 8
Output: [[2,2,2,2],[2,3,3],[3,5]]
```

Example 3:

```
Input: candidates = [2], target = 1
Output: []
```

```java
class Solution {
    public List<List<Integer>> combinationSum(int[] candidates, int target) {

        List<List<Integer>> ans = new ArrayList<>();

        if(candidates==null || candidates.length==0 || target<=0){
            return ans;
        }

        Arrays.sort(candidates);

        helper(candidates, target, 0, new ArrayList<>(), ans);

        return ans;
    }

    private void helper(int[] candidates, int target, int start, ArrayList<Integer> tempList,
List<List<Integer>> ans){

        if(target==0){
```

Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

candidates =

[2,3,6,7]

target =

7

Output