857. Minimum Cost to Hire K Workers

Hard | Greedy | Array | Sorting | Heap (Priority Queue) | Leetcode Link

Problem Description

In this problem, you are responsible for hiring a group of `n` workers. Each worker has two associated parameters: `quality` and `wage`. The `quality` of a worker represents a numerical value for the worker's skill level, while the `wage` indicates the minimum salary that the worker expects.

To form a paid group, exactly `k` workers need to be hired under two constraints:

1. The payment of each worker must be proportional to their quality relative to the other workers in the group. This means if one worker has twice the quality compared with another worker in the group, they should also be paid twice as much.
2. No worker can be paid less than their minimum wage expectation.

The goal is to calculate the minimum amount of money required to hire exactly `k` workers such that both of the aforementioned conditions are satisfied.

Intuition

The solution involves the concept of 'efficiency', which in this context is defined as the ratio of wage to quality for each worker. The intuition is to first sort the workers by their efficiency. This allows us to consider the workers in order of how much wage they require per unit quality.

Once sorted, the algorithm uses a min-heap to keep track of the `k` workers with the largest qualities, since any payment must be at least as large as the `k`-th worker's quality times the efficiency. By iterating through the sorted workers and maintaining the sum of the largest `k` qualities, when the `k`-th worker is added to the heap, the smallest ratio that satisfies all the workers' minimum wage expectations can be calculated. This is done by multiplying the total sum of the chosen qualities by the current worker's efficiency.

During iteration, the algorithm keeps updating the answer to record the least amount of money needed to form a paid group. This is achieved by maintaining a total quality sum and updating it as the group changes. When a new worker is added and the group exceeds `k` workers, the worker with the highest quality (and therefore the one contributing most to the total wage) is removed from the heap and the total quality sum. This step ensures that the maintained group always has `k` workers. The answer is updated whenever a potentially lower total wage for the group is found.

The use of a heap allows efficient removal of the worker with the highest quality in `O(log k)` time and helps to maintain the total quality sum and minimum possible wage for the group at each step.

Solution Approach

The implementation of the solution provided in Python involves a sorting step and the use of a min-heap. The overall approach follows an efficient algorithm to minimize the total wage based on the ratio of wage to quality defined as efficiency. Here's a step-by-step explanation of how the solution works:

1. **Sorting**: We start by computing the efficiency for each worker, which is the ratio of `wage[i]` to `quality[i]`. Then, we sort the workers based on this efficiency in ascending order. This ensures that we consider workers in increasing order of how much they expect to be paid relative to their quality.
2. **Initializing the Heap**: We use a min-heap (in Python, a min-heap can be utilized by pushing negative values into it) to keep track of the `k` largest qualities encountered thus far in the sorted array. A heap is advantageous here because it allows us to efficiently retrieve and remove the worker with the highest quality, which in turn affects the total payment calculation.
3. **Iterative Calculation**: Next, we iterate through the sorted workers and do the following:
   - Add the current worker's quality to the total quality sum (`tot`).
   - Push the negative of the current worker's quality to the heap to maintain the `k` largest qualities.
   - If the heap size exceeds `k`, it means we have more than `k` workers in the current group, so we pop from the heap (which removes the worker with the highest quality from our consideration) and reduce our total quality sum accordingly.
   - If the heap size is exactly `k`, we calculate the current cost to hire `k` workers based on the worker with the current efficiency and update the answer with the minimum cost observed so far. This is done by `ans = min(ans, w / q * tot)`, where `w / q` represents the efficiency of the last worker added to meet the group size of `k`.
4. **Returning the Result**: After going through all workers, the variable `ans` stores the minimum amount of money needed to hire `k` workers while satisfying the constraints. We return `ans` as the solution to the problem.

The key patterns used in this solution include sorting based on a custom comparator (efficiency), and utilizing a heap for dynamic subset selection - in this case, maintaining a subset of workers of size `k` with the greatest quality.

Algorithm Complexity: The time complexity of this algorithm is mainly governed by the sorting step, which is `O(n log n)`, where `n` is the number of workers. Operating on the heap takes `O(log k)` time for each insertion and deletion, and since we go through the workers once, the overall complexity for heap operations is `O(n log k)`. Hence, the total time complexity is `O(n log n + n log k)`. The space complexity is `O(n)` to store the sorted efficiencies and the heap.

Example Walkthrough

Let's walk through the solution with a small example.

Suppose we have 4 workers, each with the following `quality` and `wage` expectations:

```
Workers:   1   2   3   4
Quality:   10  20  15  30
Wage:      60  120 90  180
```

We want to hire `k = 2` workers while meeting the conditions specified in the problem.

Step 1: Sorting First, we calculate the efficiency `wage[i] / quality[i]` for each worker:

```
Efficiency: 6   6   6   6
```

Notice that in this case, all workers have the same efficiency, meaning any order keeps them sorted by efficiency. However, this won't always be the case.

Step 2: Initializing the Heap Now we initialize a min-heap to keep track of the largest qualities as we iterate through the workers. Initially, the heap is empty.

Step 3: Iterative Calculation We iterate through the sorted workers, keeping track of the total quality (`tot`) and potential total wage (`ans`).

- For worker 1, `quality = 10`. We add it to `tot` and push `-10` to the heap.
- For worker 2, `quality = 20`. We add it to `tot`, giving us `tot = 30`, and push `-20` to the heap.
- The heap size is now equal to `k`, so we calculate the total payment for these `k` workers using the current efficiency, which is `6 * tot = 180`. We set `ans = 180`.

To consider other possible combinations since the heap size now exceeds `k`, we must pop from the heap:

- Since the heap is a min-heap with negatives, when we pop, we're removing the worker with the highest quality, in this case `-10` (worker 1), leaving the heap with `-20` and `tot = 20`.
- We then move to worker 3, with `quality = 15`. Adding to `tot` gives us 35, and we push `-15` to the heap.
- The heap size again exceeds `k`, so we pop the largest quality, which is now `-20` (worker 2), and `tot` becomes `15` (as we keep worker 3 and the new worker 4 in our consideration).
- We must update `ans` if it gets lower. Worker 4 gets added with an efficiency of `6` and `quality = 30`. The new `tot` is `45`, and `ans` becomes `min(180, 6 * 45) = 270`.

We proceed like this until all workers have been considered.

Step 4: Returning the Result After evaluating all possible groupings of `k` workers, we find that the minimum total payment we can get away with to hire `k = 2` workers while meeting the constraints is `ans = 180`.

This is a very simplified example for clarity, where efficiencies were all equal. In practice, efficiencies would vary, and the sorting step would play a critical role in determining which workers could potentially be hired together within budget constraints.

Python Solution

```python
import heapq
from typing import List

class Solution:
    def mincostToHireWorkers(
        self, quality: List[int], wage: List[int], k: int
    ) -> float:
        # Pair each worker's quality with their minimum wage
        # and sort the workers based on their wage-to-quality ratio.
        workers = sorted(zip(quality, wage), key=lambda x: x[1] / x[0])

        # Initialize the answer as infinity to be later minimized
        # Initialize the total quality of workers hired so far as zero
        min_cost = float('inf')
        total_quality = 0

        # Max heap to store the negative of qualities,
        # since heapq in Python is a min heap by default
        max_heap = []

        # Iterate over each worker
        for q, w in workers:
            # Add the current worker's quality to the total quality
            total_quality += q
            # Push the negative quality to the heap
            heapq.heappush(max_heap, -q)

            # If we have more than k workers, remove the one with the highest quality
            if len(max_heap) > k:
                # Since we stored negative qualities, popping from heap retrieves
                # and removes the biggest quality from the total
                total_quality += heapq.heappop(max_heap)

            # If we've collected a group of k workers
            if len(max_heap) == k:
                # Calculate the current cost for this group of workers, which is
                # the wage-to-quality ratio of the current worker times total quality
                # and update the minimum cost if it's less than the previous minimum
                min_cost = min(min_cost, w / q * total_quality)

        # Return the minimum cost found to hire k workers
        return min_cost

# Example usage:
# sol = Solution()
# result = sol.mincostToHireWorkers([10, 20, 5], [70, 50, 30], 2)
# print(result)  # Should print the minimum cost to hire 2 workers
```

Java Solution

```java
import java.util.Arrays;
import java.util.PriorityQueue;

class Solution {

    // This class represents workers with their quality and wage ratio
    class Worker {
        double wageQualityRatio;
        int quality;

        Worker(int quality, int wage) {
            this.quality = quality;
            this.wageQualityRatio = (double) wage / quality;
        }
    }

    public double mincostToHireWorkers(int[] quality, int[] wage, int k) {
        int numWorkers = quality.length;
        Worker[] workers = new Worker[numWorkers];

        // Create Worker instances combining both quality and wage.
        for (int i = 0; i < numWorkers; ++i) {
            workers[i] = new Worker(quality[i], wage[i]);
        }

        // Sort the workers based on their wage-to-quality ratio.
        Arrays.sort(workers, (a, b) -> Double.compare(a.wageQualityRatio, b.wageQualityRatio));

        // Use a max heap to keep track of the highest quality workers.
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);
        double minCost = Double.MAX_VALUE; // Initialize minimum cost to a large number.
        int totalQuality = 0; // Tracks the total quality of hired workers.

        // Iterate through the sorted workers by increasing wage-to-quality ratio.
        for (Worker worker : workers) {
            // Add the current worker's quality to the total.
            totalQuality += worker.quality;
            // Add the current worker's quality to the max heap.
            maxHeap.offer(worker.quality);

            // If we have enough workers (exactly k), we can try to form an answer.
            if (maxHeap.size() == k) {
                // Calculate the cost based on current worker's wage-to-quality ratio.
                minCost = Math.min(minCost, totalQuality * worker.wageQualityRatio);
                // Remove the worker with the highest quality (to maintain only k workers).
                totalQuality -= maxHeap.poll();
            }
        }

        return minCost; // Return the minimum cost found.
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

class Solution {
public:
    // Method that returns the minimum cost to hire k workers based on their quality and the minimum wage they will work for.
    double mincostToHireWorkers(vector<int>& quality, vector<int>& wage, int k) {
        int numWorkers = quality.size();
        vector<pair<double, int>> workerRatios(numWorkers); // This will contain wage/quality ratio and quality.

        // Generate wage to quality ratios for all workers and store them with the quality.
        for (int i = 0; i < numWorkers; ++i) {
            workerRatios[i] = {(double) wage[i] / quality[i], quality[i]};
        }

        // Sort the workers based on their wage/quality ratio. Lower ratio means more cost-effective.
        sort(workerRatios.begin(), workerRatios.end());

        priority_queue<int> qualityMaxHeap; // Max heap to maintain the top k largest qualities.
        double minCost = 1e9; // Initialize minimum cost with an high value.
        int totalQuality = 0; // Total quality of the hired workers.

        // Loop through the sorted worker ratios
        for (auto& [ratio, workerQuality] : workerRatios) {
            totalQuality += workerQuality; // Add current worker's quality to the total quality.
            qualityMaxHeap.push(workerQuality); // Add current worker's quality to max heap.

            // Once we have a group of k workers.
            if (qualityMaxHeap.size() == k) {
                // Calculate the cost of hiring the current group and update minimum cost if necessary.
                minCost = min(minCost, totalQuality * ratio);
                // Remove the worker with the highest quality as we want to try for a more cost-effective group next.
                totalQuality -= qualityMaxHeap.top();
                qualityMaxHeap.pop();
            }
        }
        // Return the minimum cost found for hiring k workers.
        return minCost;
    }
};
```

Typescript Solution

```typescript
function minCostToHireWorkers(quality: Array<number>, wage: Array<number>, k: number): number {
    let numWorkers = quality.length;
    let workerRatios: Array<{ ratio: number; workerQuality: number }> = [];

    // Generate wage to quality ratio for all workers and store them along with the quality.
    for (let i = 0; i < numWorkers; ++i) {
        workerRatios.push({ ratio: wage[i] / quality[i], workerQuality: quality[i] });
    }

    // Sort the workers based on their wage to quality ratio. Lower ratio means more cost-effective.
    workerRatios.sort((a, b) => a.ratio - b.ratio);

    let qualityMaxHeap: Array<number> = []; // Using array as max heap.
    let minCost: number = Number.MAX_VALUE; // Initialize minimum cost to the highest possible value.
    let totalQuality: number = 0; // Total quality of the hired workers.

    workerRatios.forEach(worker => {
        totalQuality += worker.workerQuality; // Add current worker's quality to total
        qualityMaxHeap.push(worker.workerQuality);

        // Maintain the max heap property by sorting only the k highest qualities.
        if (qualityMaxHeap.length > k) {
            qualityMaxHeap.sort((a, b) => b - a); // Sort to get the highest quality on top
            totalQuality -= qualityMaxHeap[0]; // Remove the highest quality
            qualityMaxHeap.shift(); // Remove the first element from array
        }

        // Once we have exactly k workers, calculate the cost of hiring the group
        if (qualityMaxHeap.length === k) {
            minCost = Math.min(minCost, totalQuality * worker.ratio);
        }
    });

    // Return the minimum cost found for hiring k workers.
    return minCost;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code consists of several components:

1. **Sorting**: The list `t` is sorted based on the ratio of `wage` to `quality` using Python's built-in sort function, which has a time complexity of `O(n log n)` where `n` is the number of workers.
2. **Heap Operations**: For each worker in the sorted list, the algorithm performs heap push (`heappush`) and possibly heap pop (`heappop`) operations when the heap size is equal to `k`. Pushing onto the heap has a time complexity of `O(log k)` and popping from the heap also has a complexity of `O(log k)`.

Given there are `n` workers, and therefore `n heappush` operations, and at most `n heappop` operations when the heap size reaches `k`, the heap operations give us a total time complexity of `O(n log k)`.

So, the overall time complexity is the sum of the sorting complexity and the heap operations complexity: `O(n log n + n log k)`.

Space Complexity

The space complexity is determined by the extra space used by the heap and the sorted list:

1. **Sorted List**: The sorted list `t` takes `O(n)` space since it contains a tuple for each worker.
2. **Heap**: The heap `h` can store up to `k` elements, so it has a space complexity of `O(k)`.

Therefore, the total space complexity is the sum of the space complexities for the sorted list and the heap, which is `O(n + k)`.