1383. Maximum Performance of a Team

Hard Weekly Contest Heap (Priority Queue)

Problem Description

In this problem, we are given two integer values — the number of engineers and k (the maximum number of engineers we can select), along with two integer arrays `speed` and `efficiency`, each of length n. The `speed` array represents the speed of each engineer, while the `efficiency` array represents the efficiency of each engineer.

Our task is to form a team with at most k engineers to achieve the maximum performance. The performance of a team is defined as the sum of their speeds multiplied by the minimum efficiency among them.

We need to calculate the maximum possible performance of a team formed under these constraints and return this maximum value modulo $10^9 + 7$ to handle very large numbers.

Intuition

To achieve the maximum performance, we need to consider both speed and efficiency in our selection strategy. A greedy approach can help us select the most suitable candidates.

One key insight is that if we pick an engineer with a certain efficiency, then the total efficiency of the team cannot exceed this value. Therefore, to maximize the team's performance at each step, we aim to add engineers with the highest speeds possible without dropping the minimum efficiency of the team below the current candidate's efficiency.

Here's how we arrive at the solution approach:

1. We pair the speed and efficiency for each engineer and then sort these pairs in descending order of efficiency. This ensures that as we iterate through the engineers, we maintain the invariant that we only consider teams where the minimum efficiency is at least as much as the current engineer's efficiency.
2. We use a min-heap (which in Python is conveniently implemented through the `heapq` module) to keep track of the smallest speeds in our current team selection. This is useful because, if the team has more than k engineers, we will need to remove the one with the smallest speed to add a new one.
3. As we iterate through the sorted engineer list, we do the following:
   - Add the current engineer's speed to the total speed (`tot`).
   - Calculate the current performance by multiplying the total speed with the engineer's efficiency.
   - Check if this current performance is greater than the maximum performance found so far and update `ans` if it is.
   - Add the current engineer's speed to the heap.
   - If the heap's size exceeds k, remove the smallest speed from it and adjust the total speed (`tot`) accordingly.
4. The largest value found during this process is our maximum performance. Because large numbers are involved, we return the result modulo $10^9 + 7$.

This approach ensures that at any point, the team is formed by choosing engineers that contribute maximally to the speed while being limited by the engineer with the lowest efficiency in the team.

Solution Approach

The solution makes good use of both sorting and a min-heap to achieve the desired outcome. Here's a step-by-step explanation of the implementation:

1. **Sorting by Efficiency**: We start by creating a list of tuples `t` that pairs each engineer's `speed` with their corresponding `efficiency`. We then sort this list in descending order based on `efficiency`. This allows us to process the engineers in order of decreasing efficiency, so at each step, we consider the highest possible efficiency for the team and try to maximize speed.

```
t = sorted(zip(speed, efficiency), key=lambda x: -x[1])
```

2. **Initial Declarations**: We initialize three variables:
   - `ans` to track the maximum performance encountered so far.
   - `tot` to keep the running sum of the speeds of the engineers in the current team.
   - `mod` to store the modulus value of $10^9 + 7$ for use at the end of the calculation.

```
ans = tot = 0
mod = 10**9 + 7
```

3. **Min-Heap Usage**: We declare a list `h` which will serve as our min-heap using the `heapq` library. This is where we will store the speeds of the engineers we choose for our team. A min-heap allows us to efficiently retrieve and remove the engineer with the smallest speed when needed.

```
h = []
```

4. **Iterating over Engineers**: We now iterate over each engineer in the sorted list `t`. For each engineer, we perform the following steps:
   - Add the engineer's speed to `tot`.
   - Calculate a potential maximum performance `tot * e` and compare this with `ans`, updating `ans` if it's larger. This step considers that the current engineer's efficiency sets the new minimum efficiency for the team.
   - Add the engineer's speed to the min-heap.
   - If the heap size reaches k, we remove the smallest speed using `heappop`, which would be the one at the root of the min-heap, and subtract that value from `tot`. This step ensures that the team size does not exceed k.

```
for s, e in t:
    tot += s
    ans = max(ans, tot * e)
    heappush(h, s)
    if len(h) == k:
        tot -= heappop(h)
```

5. **Modulo Operation**: After the loop is done, `ans` holds the maximum performance value. We return `ans % mod` to ensure the result adheres to the modulo constraint.

```
return ans % mod
```

The use of sorting ensures that we're always considering engineers in the order of their efficiency, from highest to lowest, which is crucial for maximizing performance. The min-heap is essential for managing the team size and quickly ejecting the least-contributing member (in terms of speed) when the team exceeds the size k. With this strategy, the complexity of the solution is dictated by the sorting step, which is $O(n \log n)$, and the heap operations, which have $O(\log k)$ complexity for each of the n engineers, making the overall complexity $O(n \log n + n \log k)$.

Example Walkthrough

Let's illustrate the solution approach with a small example:

- Suppose we have $n = 4$ engineers and $k = 2$ as the maximum number of engineers we can select.
- Let's assume the `speed` array is $[5, 2, 3, 9]$ and the `efficiency` array is $[10, 1, 4, 7]$.

Following the steps described in the solution approach:

1. **Sorting by Efficiency**: We first pair each engineer's speed with their corresponding efficiency and sort the data by efficiency in descending order.

   Pairs formed: $[(5, 10), (2, 1), (3, 4), (9, 7)]$

   After sorting by efficiency: $[(5, 10), (9, 7), (3, 4), (2, 1)]$

2. **Initial Declarations**:

   Variable initialization:

   - `ans` = 0
   - `tot` = 0
   - `mod` = $10^9 + 7$

3. **Min-Heap Usage**:

   Initialize an empty min-heap `h`.

4. **Iterating over Engineers**: Now we consider each engineer in `t`.

   - For engineer (5, 10):
     - `tot` becomes 5.
     - Potential performance: $5 \times 10 = 50$, `ans` is updated to 50.
     - Add speed 5 to the heap. Heap: `h` = [5].
   - For engineer (9, 7):
     - `tot` becomes 14 (5 + 9).
     - Potential performance: $14 \times 7 = 98$, `ans` is updated to 98.
     - Add speed 9 to the heap. Heap: `h` = [5, 9].
   - For engineer (3, 4):
     - `tot` becomes 17 (14 + 3).
     - Potential performance: $17 \times 4 = 68$, which is less than `ans` (98), so no update is needed.
     - Add speed 3 to the heap. Heap: `h` = [3, 5, 9].
     - Since the heap now exceeds k (size is 3), we remove the smallest speed 3 (heap pop). `tot` becomes 14 (17 - 3).
   - For engineer (2, 1):
     - `tot` becomes 16 (14 + 2).
     - Potential performance: $16 \times 1 = 16$, which is less than `ans` (98), so no update is needed.
     - Add speed 2 to the heap. Heap: `h` = [2, 5, 9].
     - Again, since the heap size exceeds k, we remove the smallest speed 2 (heap pop). `tot` becomes 14 (16 - 2).

   The loop ends and the largest value found during this process is `ans` = 98.

5. **Module Operation**:

   Finally, we return `ans % mod`, which is 98 % (10^9 + 7) = 98 because 98 is already less than $10^9 + 7$.

By the end of this process, we've determined that the maximum performance that can be achieved with a team of at most k engineers is 98.

Python Solution

```
from typing import List
from heapq import heappush, heappop

class Solution:
    def maxPerformance(self, n: int, speed: List[int], efficiency: List[int], k: int) -> int:
        # Combine the speed and efficiency into a single list of tuples
        # Sort the engineers by efficiency in descending order
        engineers = sorted(zip(speed, efficiency), key=lambda x: -x[1])

        max_performance = total_speed = 0
        MOD = 10**9 + 7
        min_speed_heap = []  # A min-heap to track the smallest speeds.

        # Iterate through the engineers sorted by efficiency.
        for current_speed, current_efficiency in engineers:
            # Add the current engineer's speed to the total speed.
            total_speed += current_speed

            # Calculate the current performance with the current engineer.
            max_performance = max(max_performance, total_speed * current_efficiency)

            # Add the current speed to the min heap.
            heappush(min_speed_heap, current_speed)

            # If the size of the team exceeds k, remove the engineer with the smallest speed.
            if len(min_speed_heap) > k:
                # Remove the smallest speed from the total speed as we pop from the heap.
                total_speed -= heappop(min_speed_heap)

        # The result could be large so return the result modulo 10^9 + 7.
        return max_performance % MOD
```

Java Solution

```
class Solution {
    private static final int MODULUS = (int) 1e9 + 7; // Define a constant for the modulus value

    public int maxPerformance(int n, int[] speed, int[] efficiency, int k) {
        // Create an array to store speed and efficiency pairs
        int[][] engineers = new int[n][2];
        for (int i = 0; i < n; ++i) {
            engineers[i] = new int[] {speed[i], efficiency[i]};
        }

        // Sort the engineers array by efficiency in descending order
        Arrays.sort(engineers, (a, b) -> b[1] - a[1]);

        // Use a min heap to keep track of the k highest speeds
        PriorityQueue<Integer> speedQueue = new PriorityQueue<>();

        long totalSpeed = 0;  // Total speed of the selected engineers
        long maxPerformance = 0;  // Max performance found so far

        // Iterate through the sorted engineers
        for (int[] engineer : engineers) {
            int currentSpeed = engineer[0];
            int currentEfficiency = engineer[1];

            // Add the current speed to the total and update the max performance
            totalSpeed += currentSpeed;
            maxPerformance = Math.max(maxPerformance, totalSpeed * currentEfficiency);

            // Add the current speed to the speed queue
            speedQueue.offer(currentSpeed);

            // If the size of the speedQueue exceeds k, remove the slowest engineer
            if (speedQueue.size() > k) {
                // Polling removes the smallest element (min heap property)
                totalSpeed -= speedQueue.poll();
            }
        }

        // Return the max performance modulo the defined modulus to prevent overflow
        return (int) (maxPerformance % MODULUS);
    }
}
```

C++ Solution

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

using namespace std;

class Solution {
public:
    int maxPerformance(int numEngineers, vector<int>& speeds, vector<int>& efficiencies, int maxTeamSize) {
        // Pair speed with efficiency, and sort engineers by increasing efficiency
        vector<pair<int, int>> engineerPairs(numEngineers);
        for (int i = 0; i < numEngineers; ++i) {
            engineerPairs[i] = {efficiencies[i], speeds[i]};
        }
        // Sort the engineers by efficiency in descending order
        sort(engineerPairs.rbegin(), engineerPairs.rend());

        // Min heap to keep track of the least speed in the current team for easy removal
        priority_queue<int, vector<int>, greater<int>> speedHeap;
        long long maxPerformance = 0, totalSpeed = 0;
        int modulus = 1e9 + 7; // For taking modulo after calculations to prevent overflow

        for (auto& engineer : engineerPairs) {
            int currentSpeed = engineer.second;
            int currentEfficiency = engineer.first;

            // Add the current engineer's speed to the total speed
            totalSpeed += currentSpeed;

            // Update max performance with the new team configuration
            // Here, the performance is calculated as the product of total speed and
            // the efficiency of the least efficient engineer in the team
            maxPerformance = max(maxPerformance, totalSpeed * currentEfficiency);

            // Add the current speed to the min heap
            speedHeap.push(currentSpeed);

            // If the team size exceeds the max allowed size, remove the engineer
            // with the lowest speed from the team
            if (speedHeap.size() > maxTeamSize) {
                totalSpeed -= speedHeap.top(); // Subtract the lowest speed
                speedHeap.pop(); // Remove the lowest speed
            }
        }

        // Return the maximum performance modulo 1e9+7
        return static_cast<int>(maxPerformance % modulus);
    }
};
```

Typescript Solution

```
type Engineer = {
    speed: number;
    efficiency: number;
};

function maxPerformance(numEngineers: number, speeds: number[], efficiencies: number[], maxTeamSize: number): number {
    // Create an array of objects containing speed and efficiency
    const engineers: Engineer[] = speeds.map((speed, index) => ({
        speed: speed,
        efficiency: efficiencies[index],
    }));

    // Sort engineers primarily by efficiency in descending order
    engineers.sort((a, b) => b.efficiency - a.efficiency);

    // Min heap to keep track of the least speed in the current team for easy removal
    let speedHeap: number[] = [];
    let maxPerformance: number = 0,
    let totalSpeed: number = 0;
    const modulus: number = 1e9 + 7; // For taking modulo after calculations to prevent overflow

    engineers.forEach(engineer => {
        // Add the current engineer's speed to the total speed
        totalSpeed += engineer.speed;
        // Update max performance with the new team configuration
        maxPerformance = Math.max(maxPerformance, totalSpeed * engineer.efficiency);

        // Add current speed to the min heap
        speedHeap.push(engineer.speed);
        speedHeap.sort((a, b) => a - b); // Convert array to a min heap by sorting in ascending order

        // If the team size exceeds the max allowed size, remove the engineer with the lowest speed
        if (speedHeap.length > maxTeamSize) {
            totalSpeed -= speedHeap.shift()!; // Shift operation removes the first element, which is the smallest due to sorting
        }
    });

    return maxPerformance % modulus;
}
```

Time and Space Complexity

Time Complexity

The given Python code sorts an array of tuples based on the efficiency, uses a min-heap for maintaining the k highest speeds, and iterates through the sorted list to calculate the answer.

1. Sorting the array of tuples based on efficiency has a time complexity of $O(n \log n)$ where n is the number of engineers.
2. Iterating through the sorted array to calculate the maximum performance occurs in $O(n)$, as each element is visited exactly once.
3. For every engineer processed inside the loop, it may add an element to the min-heap, which is $\log k$ where k is the maximum team size. However, this operation will only happen up to k times because once the heap size reaches k, engineers are popped out for every new engineer added.
4. The `heappop` operation, which occurs when the heap reaches the size k, is also $O(\log k)$.

Therefore, the total time complexity combines the sorting and the heap operations: $O(n \log n) + O(n \log k)$. Since n can be at most n, in the worst case, it simplifies to $O(n \log n)$.

Space Complexity

The space complexity should account for the sorted array of tuples, and the min-heap used to store up to k elements:

1. The sorted array of tuples has a space complexity of $O(n)$, as it stores n pairs of `(speed, efficiency)`.
2. The min-heap also has a space complexity of $O(k)$ as it may store up to k elements at a time.

The total space complexity is the higher of the two (n or k) when k is greater than n, otherwise $O(k)$. In asymptotic notation, we express this as $O(\max(n, k))$, but since $k <= n$, it simplifies to $O(n)$.