1383. Maximum Performance of a Team Sorting Heap (Priority Queue) Greedy Array Leetcode Link Hard

select), along with two integer arrays speed and efficiency, each of length n. The speed array represents the speed of each

In this problem, we are given two integer values n (the number of engineers) and k (the maximum number of engineers we can

Problem Description

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 the speeds of the selected engineers 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.

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.

Intuition

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

one with the smallest speed to add a new one.

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

as we iterate through the engineers, we maintain the invariant that we only consider teams where the minimum efficiency is at

- 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.

being limited by the engineer with the lowest efficiency in the team.

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

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.

- This approach ensures that at any point, the team is formed by choosing engineers that contribute maximally to the speed while
- 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.

If the heap's size exceeds k, remove the smallest speed from it and adjust the total speed (tot) accordingly.

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.

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

2. Initial Declarations: We initialize three variables: ans to track the maximum performance encountered so far.

1 for s, e in t:

1 return ans % mod

Example Walkthrough

descending order.

2. Initial Declarations:

Variable initialization:

 \circ mod = 10^9 + 7

tot += s

heappush(h, s)

if len(h) == k:

ans = max(ans, tot * e)

tot -= heappop(h)

adheres to the modulo constraint.

overall complexity 0(n log n + n log k).

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

Following the steps described in the solution approach:

ans = tot = 0

 $2 \mod = 10**9 + 7$

Solution Approach

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. 1 h = []

4. Iterating over Engineers: We now iterate over each engineer in the sorted list t. For each engineer, we perform the following

If the heap size reaches k, we remove the smallest speed using heappop, which would be the one at the root of the min-

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

considers that the current engineer's efficiency sets the new minimum efficiency for the team.

heap, and subtract that value from tot. This step ensures that the team size does not exceed k.

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

Add the engineer's speed to the min-heap.

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 0(n log n), and the heap operations, which have 0(log k) complexity for each of the n engineers, making the

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

The use of sorting ensures that we're always considering engineers in the order of their efficiency, from highest to lowest, which is

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

Let's assume the speed array is [5, 2, 3, 9] and the efficiency array is [10, 1, 4, 7].

• Suppose we have n = 4 engineers and k = 2 as the maximum number of engineers we can select.

Initialize an empty min-heap h. 4. Iterating over Engineers: Now we consider each engineer in t.

o ans = 0

 \circ tot = 0

3. Min-Heap Usage:

For engineer (5, 10):

• Potential performance: 17 * 4 = 68, which is less than ans (98), so no update is needed.

Potential performance: 16 * 1 = 16, which is less than ans (98), so no update is needed.

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

- Potential performance: 5 * 10 = 50. ans is updated to 50. Add speed 5 to the heap. Heap h: [5].
- tot becomes 14 (5 + 9). Potential performance: 14 * 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).

Add speed 3 to the heap. Heap h: [3, 9, 5].

Add speed 2 to the heap. Heap h: [2, 9, 5].

■ 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).

tot becomes 5.

For engineer (9, 7):

- 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. Modulo Operation:
- engineers is 98. Python Solution

from heapq import heappush, heappop

mod = 10**9 + 7

1 from typing import List

class Solution:

10

11

12

13

14

15

16

17

18

25

26

27

28

29

30

31

32

10

11

12

13

20

22

24

Calculate the current performance with the current engineer. max_performance = max(max_performance, total_speed * curr_efficiency) 20 21 22 # Add the current speed to the min heap. 23 heappush(min_speed_heap, curr_speed) 24

If the size of the team exceeds k, remove the engineer with the smallest speed.

Remove the smallest speed from the total speed as we pop from the heap.

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

// Total speed of the selected engineers

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

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

min_speed_heap = [] # A min-heap to track the smallest speeds.

Add the current engineer's speed to the total speed.

and sort it in descending order of efficiency.

Iterate through the engineers sorted by efficiency.

total_speed -= heappop(min_speed_heap)

The result could be large; we return the result modulo 10^9 + 7.

public int maxPerformance(int n, int[] speed, int[] efficiency, int k) {

// Create an array to store speed and efficiency pairs

Arrays.sort(engineers, (a, b) -> b[1] - a[1]);

// Iterate through the sorted engineers

int currentSpeed = engineer[0];

int currentEfficiency = engineer[1];

for (var engineer : engineers) {

engineers[i] = new int[] {speed[i], efficiency[i]};

// Sort the engineers array by efficiency in descending order

for curr_speed, curr_efficiency in combined:

max_performance = total_speed = 0

total_speed += curr_speed

if len(min_speed_heap) > k:

int[][] engineers = new int[n][2];

for (int i = 0; i < n; ++i) {

return max performance % mod

14 // Use a min heap to keep track of the k highest speeds PriorityQueue<Integer> speedQueue = new PriorityQueue<>(); 15 16 17 long totalSpeed = 0; 18 long maxPerformance = 0; // Max performance found so far 19

C++ Solution

#include <vector>

class Solution {

public:

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

50

49 };

};

9

11

12

13

14

15

16

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

});

#include <algorithm>

using namespace std;

2 #include <queue>

Java Solution

1 class Solution {

25 // Add the current speed to the total and update the max performance totalSpeed += currentSpeed; 26 maxPerformance = Math.max(maxPerformance, totalSpeed * currentEfficiency); 27 28 // Add the current speed to the speed queue 29 30 speedQueue.offer(currentSpeed); 31 32 // If the size of the speedQueue exceeds k, remove the slowest engineer if (speedQueue.size() > k) { 33 34 // Polling removes the smallest element (min heap property) 35 totalSpeed -= speedQueue.poll(); 36 37 38 // Return the max performance modulo the defined modulus to prevent overflow 39 return (int) (maxPerformance % MODULUS); 41 42 } 43

int maxPerformance(int numEngineers, vector<int>& speeds, vector<int>& efficiencies, int maxTeamSize) {

// Pair speed with efficiency, and sort engineers by increasing efficiency

// Min heap to keep track of the lowest speed in the current team for easy removal

int modulus = 1e9 + 7; // For taking modulo after calculations to prevent overflow

// Here, the performance is calculated as the product of total speed and

maxPerformance = max(maxPerformance, totalSpeed * currentEfficiency);

// If the team size exceeds the max allowed size, remove the engineer

totalSpeed -= speedHeap.top(); // Subtract the lowest speed

function maxPerformance(numEngineers: number, speeds: number[], efficiencies: number[], maxTeamSize: number): number {

vector<pair<int, int>> engineerPair(numEngineers);

sort(engineerPair.rbegin(), engineerPair.rend());

long long maxPerformance = 0, totalSpeed = 0;

int currentSpeed = engineer.second;

int currentEfficiency = engineer.first;

// Add the current speed to the min heap

// with the lowest speed from the team

if (speedHeap.size() > maxTeamSize) {

// Return the maximum performance modulo 1e9+7

return static_cast<int>(maxPerformance % modulus);

// Create an array of objects containing speed and efficiency

const engineers: Engineer[] = speeds.map((speed, index) => ({

// Sort engineers primarily by efficiency in descending order

// Min heap to keep track of the lowest speed in the current team for easy removal

// Calculate the maximum performance with the new team configuration

maxPerformance = Math.max(maxPerformance, totalSpeed * engineer.efficiency);

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

engineers.sort((a, b) => b.efficiency - a.efficiency);

for (auto& engineer : engineerPair) {

totalSpeed += currentSpeed;

speedHeap.push(currentSpeed);

engineerPair[i] = {efficiencies[i], speeds[i]};

priority_queue<int, vector<int>, greater<int>> speedHeap;

// Add the current engineer's speed to the total speed

speedHeap.pop(); // Remove the lowest speed

// Update max performance with the new team configuration

// the efficiency of the least efficient engineer in the team

// Sort the engineers primarily by efficiency in descending order

for (int i = 0; i < numEngineers; ++i) {</pre>

const speedHeap: number[] = []; 17 let maxPerformance: number = 0; 18 let totalSpeed: number = 0; 19 const modulus: number = 1e9 + 7; // For taking modulo after calculations to prevent overflow 20 21 22 engineers.forEach(engineer => {

}));

Typescript Solution

speed: number;

efficiency: number;

speed: speed,

efficiency: efficiencies[index],

totalSpeed += engineer.speed;

// Add current speed to the heap

if (speedHeap.length > maxTeamSize) {

speedHeap.push(engineer.speed);

return maxPerformance % modulus;

Time and Space Complexity

out for every new engineer added.

type Engineer = {

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 0(log k) where k is the maximum

4. The 'heappop' operation, which occurs when the heap reaches the size k, is also $0(\log k)$.

Therefore, the total time complexity combines the sorting and the heap operations: 0(n log n) + 0(n log k). Since k can be at most n, in the worst case, it simplifies to $O(n \log n)$. Space Complexity

team size. However, this operation will only happen up to k times because once the heap size reaches k, engineers are popped

totalSpeed -= speedHeap.shift() as number; // Shift operation removes the first element, which is the smallest due to mir

1. The sorted array of tuples has a space complexity of O(n), as it stores n pairs of (speed, efficiency).

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

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: O(n) when n is greater than k, otherwise O(k). In asymptotic notation, we express this as $0(\max(n, k))$, but since $k \ll n$, it simplifies to 0(n).