<https://leetcode.com/problems/maximum-performance-of-a-team/description/>

You are given two integers n and k and two integer arrays speed and efficiency both of length n. There are n engineers numbered from 1 to n. speed[i] and efficiency[i] represent the speed and efficiency of the ith engineer respectively.

Choose at most k different engineers out of the n engineers to form a team with the maximum performance.

The performance of a team is the sum of its engineers' speeds multiplied by the minimum efficiency among its engineers.

Return the maximum performance of this team. Since the answer can be a huge number, return it modulo 10^9 + 7.

**Example 1:**

**Input:** n = 6, speed = [2,10,3,1,5,8], efficiency = [5,4,3,9,7,2], k = 2

**Output:** 60

**Explanation:** We have the maximum performance of the team by selecting engineer 2 (with speed=10 and efficiency=4) and engineer 5 (with speed=5 and efficiency=7). That is, performance = (10 + 5) \* min(4, 7) = 60.

**Example 2:**

**Input:** n = 6, speed = [2,10,3,1,5,8], efficiency = [5,4,3,9,7,2], k = 3

**Output:** 68

**Explanation:**This is the same example as the first but k = 3. We can select engineer 1, engineer 2 and engineer 5 to get the maximum performance of the team. That is, performance = (2 + 10 + 5) \* min(5, 4, 7) = 68.

**Example 3:**

**Input:** n = 6, speed = [2,10,3,1,5,8], efficiency = [5,4,3,9,7,2], k = 4

**Output:** 72

**Attempt 1: 2023-03-16**

**Solution 1: Priority Queue (10 min, exactly similar to L2542.** [Maximum Subsequence Score](https://leetcode.com/problems/maximum-subsequence-score/)**)**

**Style 1: Have smaller intermediate result**

class Solution {

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

        long MOD = (long)1e9 + 7;

        // PriorityQueue to hold the smallest 'k' elements from speed

        PriorityQueue<Integer> minPQ = new PriorityQueue<>();

        // Initialize an array of arrays to hold pairs from speed and efficiency

        int[][] pair = new int[n][2];

        for(int i = 0; i < n; i++) {

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

        }

        // Sort by efficiency[i] in a pair {speed[i], efficiency[i]}

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

        long speedSum = 0;

        long result = 0;

        for(int i = 0; i < n; i++) {

            speedSum += pair[i][0];

            minPQ.offer(pair[i][0]);

            // The difference than L2542 is in L1383 "Choose at most k

            // different engineers out of the n engineers", but in

            // L2542 is " You must choose a subsequence of indices

            // from nums1 of length k."

            //if(minPQ.size() == k) {

            if(minPQ.size() > k) {

                //result = Math.max(result, speedSum \* pair[i][1]);

                speedSum -= minPQ.poll();

            }

            result = Math.max(result, speedSum \* pair[i][1]);

        }

        return (int) (result % MOD);

    }

}

Time Complexity: O(m log m + 2m log k)

Space Complexity: O(m + k)

**Style 2: No smaller intermediate result**

class Solution {

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

        long MOD = (long)1e9 + 7;

        // PriorityQueue to hold the smallest 'k' elements from speed

        PriorityQueue<Integer> minPQ = new PriorityQueue<>();

        // Initialize an array of arrays to hold pairs from speed and efficiency

        int[][] pair = new int[n][2];

        for(int i = 0; i < n; i++) {

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

        }

        // Sort by efficiency[i] in a pair {speed[i], efficiency[i]}

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

        long speedSum = 0;

        long result = 0;

        for(int i = 0; i < n; i++) {

            speedSum += pair[i][0];

            minPQ.offer(pair[i][0]);

            result = Math.max(result, speedSum \* pair[i][1]);

            // The difference than L2542 is in L1383 "Choose at most k

            // different engineers out of the n engineers", but in

            // L2542 is " You must choose a subsequence of indices

            // from nums1 of length k."

            if(minPQ.size() == k) {

            //if(minPQ.size() > k) {

                //result = Math.max(result, speedSum \* pair[i][1]);

                speedSum -= minPQ.poll();

            }

            //result = Math.max(result, speedSum \* pair[i][1]);

        }

        return (int) (result % MOD);

    }

}

Time Complexity: O(m log m + 2m log k)

Space Complexity: O(m + k)

**Refer to**

<https://leetcode.com/problems/maximum-performance-of-a-team/solutions/539687/java-c-python-priority-queue/>

**Solution 2: Priority Queue**

We keep the queue with maximum size of k.

Each time when we introduce a new engineer,

we need only O(logK) to find the smallest speed in the team now.

**Complexity**

Time O(NlogN) for sorting

Time O(NlogK) for priority queue

Space O(N)

**Java**

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

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

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

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

Arrays.sort(ess, (a, b) -> b[0] - a[0]);

PriorityQueue<Integer> pq = new PriorityQueue<>(k, (a, b) -> a - b);

long res = 0, sumS = 0;

for (int[] es : ess) {

pq.add(es[1]);

sumS = (sumS + es[1]);

if (pq.size() > k) sumS -= pq.poll();

res = Math.max(res, (sumS \* es[0]));

}

return (int) (res % (long)(1e9 + 7));

}

For above PriorityQueue solution, a trick is that each time when you add a new engineer and then lay off the engineer with the least speed, it could be the newly added one. So the new res of the second parameter of Math.max is actually incorrect!

res = Math.max(res, (sumS \* es[0]));

However, in this case the new res value will be definitely lower than the existing res, so it does not matter.

So in this solution you may get an incorrect intermediate result but it doesn't hurt..... it really takes me some while to try to understand it...

I have a solution that avoids the ugly intermediate result.

class Solution {

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

int[][] engineers = new int[speed.length][2];

for (int i = 0; i < speed.length; ++i) {

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

}

Arrays.sort(engineers, (a,b)->(b[0]-a[0]));

PriorityQueue<int[]> pq = new PriorityQueue<int[]>((a,b)-> (a[1]-b[1]));

long sumS = 0;

int minEfficiency = Integer.MAX\_VALUE;

long result = 0;

for (int[] e : engineers) {

minEfficiency = Math.min(e[0] , minEfficiency);

sumS += e[1];

result = Math.max(result, sumS \*minEfficiency);

pq.offer(e);

if (pq.size() == k) {

int[] removedEng = pq.poll();

sumS -= removedEng[1];

}

}

return (int) (result % ((long) Math.pow(10,9) + 7));

}

}

**Refer to**

<https://algo.monster/liteproblems/1383>

**Problem Description**

In this problem, we are given two integer values n (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 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.

**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:

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.

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.

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 equals k, remove the smallest speed from it and adjust the total speed (tot) accordingly, preparing to accept new speed from next iteration if any.

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:

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])

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

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 = []

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)

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:

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)]

Initial Declarations:

Variable initialization:

ans = 0

tot = 0

mod = 10^9 + 7

Min-Heap Usage:

Initialize an empty min-heap h.

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

For engineer (5, 10):

tot becomes 5.

Potential performance: 5 \* 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 \* 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 \* 4 = 68, which is less than ans (98), so no update is needed.

Add speed 3 to the heap. Heap h: [3, 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).

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

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

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.

Modulo 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.

**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 (var 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);

    }

}

[Time and Space Complexity](https://algo.monster/problems/runtime_summary)

**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.

Sorting the array of tuples based on efficiency has a time complexity of O(n log n) where n is the number of engineers.

Iterating through the sorted array to calculate the maximum performance occurs in O(n), as each element is visited exactly once.

For every engineer processed inside the loop, it may add an element to the min-heap, which is O(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.

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 k 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:

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

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 O(max(n, k)), but since k <= n, it simplifies to O(n).

**Refer to**

[L2542.Maximum Subsequence Score (Ref.L1383)](note://WEBcc07e9686d4ff6360b7baba2fd021136)