<https://leetcode.com/problems/maximum-subsequence-score/description/>

You are given two **0-indexed** integer arrays nums1 and nums2 of equal length n and a positive integer k. You must choose a **subsequence** of indices from nums1 of length k.

For chosen indices i0, i1, ..., ik - 1, your **score** is defined as:

The sum of the selected elements from nums1 multiplied with the **minimum** of the selected elements from nums2.

It can defined simply as: (nums1[i0] + nums1[i1] +...+ nums1[ik - 1]) \* min(nums2[i0] , nums2[i1], ... ,nums2[ik - 1]).

Return the **maximum** possible score.

A **subsequence** of indices of an array is a set that can be derived from the set {0, 1, ..., n-1} by deleting some or no elements.

**Example 1:**

**Input:** nums1 = [1,3,3,2], nums2 = [2,1,3,4], k = 3

**Output:** 12

**Explanation:**

The four possible subsequence scores are:

- We choose the indices 0, 1, and 2 with score = (1+3+3) \* min(2,1,3) = 7.

- We choose the indices 0, 1, and 3 with score = (1+3+2) \* min(2,1,4) = 6.

- We choose the indices 0, 2, and 3 with score = (1+3+2) \* min(2,3,4) = 12.

- We choose the indices 1, 2, and 3 with score = (3+3+2) \* min(1,3,4) = 8.

Therefore, we return the max score, which is 12.

**Example 2:**

**Input:** nums1 = [4,2,3,1,1], nums2 = [7,5,10,9,6], k = 1

**Output:** 30

**Explanation:**

Choosing index 2 is optimal: nums1[2] \* nums2[2] = 3 \* 10 = 30 is the maximum possible score.

**Constraints:**

n == nums1.length == nums2.length

1 <= n <= 10^5

0 <= nums1[i], nums2[j] <= 10^5

1 <= k <= n

**Attempt 1: 2023-03-15**

**Solution 1: Priority Queue (180 min)**

class Solution {

    public long maxScore(int[] nums1, int[] nums2, int k) {

        // PriorityQueue to hold the smallest 'k' elements from nums1

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

        int n = nums1.length;

        // Initialize an array of arrays to hold pairs from nums1 and nums2

        int[][] pair = new int[n][2];

        for(int i = 0; i < n; i++) {

            pair[i] = new int[]{nums1[i], nums2[i]};

        }

        // Sort by nums2[i] in a pair {nums1[i], nums2[i]}

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

        long nums1Sum = 0;

        long result = 0;

        for(int i = 0; i < n; i++) {

            nums1Sum += pair[i][0];

            minPQ.offer(pair[i][0]);

            if(minPQ.size() == k) {

                result = Math.max(result, nums1Sum \* pair[i][1]);

                nums1Sum -= minPQ.poll();

            }

        }

        return result;

    }

}

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

Space Complexity: O(m + k)

**Refer to**

<https://leetcode.com/problems/maximum-subsequence-score/solutions/3082106/java-c-python-priority-queue/>

**Intuition**

Almost exactly same as

[1383. Maximum Performance of a Team](https://leetcode.com/problems/maximum-performance-of-a-team/discuss/539687/JavaC%2B%2BPython-Priority-Queue).

**Explanation**

We iterate all pairs (A[i], B[i]) with B[i] from big to small,

We keep the priority queue with maximum size of k.

Each time when we introduce a new pair of (A[i], B[i]),

the current minimum value on B is B[i]

the current sum value on A is sum(priority queue)

If the size of queue > k,

we pop the minimum A[i].

also update total sum -= A[i]

If the size of queue == k,

we update res = res = max(res, sum \* B[i])

**Complexity**

Time O(nlogn)

Space O(n)

**Java**

public long maxScore(int[] speed, int[] efficiency, int k) {

int n = speed.length;

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();

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

}

return res;

}

**Refer to**

<https://algo.monster/liteproblems/2542>

**Problem Description**

In this problem, we are presented with two 0-indexed integer arrays nums1 and nums2 of the same length n and a positive integer k. Our objective is to find the maximum possible score by picking a subsequence of indices from nums1 with a length of k. The score is calculated by summing up the values at the selected indices from nums1 and then multiplying the sum by the minimum value found at the corresponding indices in nums2. In other words, if we select indices i0, i1, ... ik-1, the score would be (nums1[i0] + nums1[i1] + ... + nums1[ik-1]) \* min(nums2[i0], nums2[i1], ..., nums2[ik-1]). A subsequence of indices we choose should not be confused with a subsequence of elements; we are choosing the indices themselves, which can be non-consecutive.

**Intuition**

To maximize the score, we intuitively want to select the k indices that would result in the largest possible sum from nums1 while also ensuring that the minimum value selected from nums2 is as high as possible, because it multiplies the entire sum.

If we were just trying to maximize the sum of selected values from nums1, we would simply choose the k highest values. However, the challenge here is that we also need to consider nums2, as its minimum value among the chosen indices will act as a multiplier for our sum from nums1.

**This leads to the strategy of pairing the elements from nums1 and nums2 and sorting these pairs in descending order based on the values from nums2, because we are interested in larger values of nums2 due to its role as a multiplier. Now, since our final score involves a sum from nums1 and a minimum from nums2, we wish to select the top k indices with respect to the product of sums and minimums.**

**To implement this, we keep a running sum of the nums1 values and use a min heap to keep track of the k smallest nums1 values that we have encountered. This is because, as we iterate through our sorted pairs, we want to have the ability to quickly identify and remove the smallest value from our current selection in order to replace it with a potentially better option. At each iteration, if our heap is full (i.e. has k elements), we calculate a possible score using our running sum and the current value from nums2. We update our maximum score if the newly calculated score exceeds our current maximum.**

By following this strategy, we ensure that the eventual k-sized subsequence of indices from nums1 (with corresponding values from nums2) will provide the maximum score.

**Learn more about Greedy, Sorting and Heap (Priority Queue) patterns.**

**Solution Approach**

The implementation of the solution involves a sort operation followed by the use of a min heap. Here is how this approach unfolds:

**Pairing and Sorting:** We start by creating pairs (a, b) from nums2 and nums1, respectively. This allows us to process elements from both arrays simultaneously. The pairing is done using Python's zip function, and then we sort these pairs in descending order based on the first element of each pair, which comes from nums2.

**Min Heap Initialization:** A min heap (q) is a data structure that allows us to efficiently keep track of the k smallest elements of nums1 we have encountered so far. In Python, a min heap can be easily implemented using a list and the heapq module's functions: heappush and heappop.

**Iterating Over Pairs:** With our pairs sorted, we iterate over each (a, b). Variable a is from nums2 and b is from nums1. The variable s maintains a running sum of the values of nums1 we have added to our heap so far.

**Maintaining the Heap and Calculating Scores:** In each iteration, we add b to the heap and to the running sum s. If the heap contains more than k elements, we remove the smallest element (which is at the root of the min heap). This ensures that our heap always contains the k largest elements from our current range of nums1 considered so far. We calculate the potential score by multiplying our running sum s with the minimum value from nums2 (which is a, because our pairs are sorted by the values from nums2 in descending order). We update the maximum score ans with this potential score if it's higher than the current ans.

**Maximizing the Score:** The heap's invariant, which always maintains the largest k elements from nums1 seen so far, guarantees that the running sum s is as large as it can be without considering the current pair's nums1 value, b. Since a is the minimum of nums2 for the current subsequence being considered, we get the maximum potential score for this subsequence in each iteration. By maintaining the max score throughout the iterations, we ensure we get the maximum score possible across all valid subsequences.

The implementation can be summarized by the following pseudocode:

nums = sort pairs (nums2, nums1) in descending order of nums2 values

q = initialize a min heap

s = 0  # Running sum of nums1 values in the heap

ans = 0  # Variable to store the maximum score

for each (a, b) in nums:

    add b to the running sum s

    push b onto the min heap q

    if the size of heap q exceeds k:

        remove the smallest element from q

        subtract its value from the running sum s

    update ans if (s \* a) is greater than current ans

return ans

By traversing the sorted pairs and using a min heap to effectively manage the heap size and running sum, we ensure the efficient computation of the maximum score. This solution has a time complexity of O(n log n) due to the [sorting](https://algo.monster/problems/sorting_summary) operation and O(n log k) due to heap operations across n elements, with each such operation having a time complexity of O(log k).

**Example Walkthrough**

Let's consider the following example to illustrate the solution approach.

Suppose we have the following inputs: nums1 = [3, 5, 2, 7] nums2 = [8, 1, 4, 3] k = 2

We want to find the maximum score by selecting k=2 indices from nums1 and using the corresponding nums2 elements as the multiplier.

Pairing and Sorting: First, we pair the elements from nums1 and nums2 and sort them based on nums2 values in descending order. Pairs: [(8, 3), (4, 2), (3, 7), (1, 5)] After sorting: [(8, 3), (4, 2), (3, 7), (1, 5)] (already sorted in this case)

Min Heap Initialization: We initialize an empty min heap q.

Iterating Over Pairs: We iterate over the sorted pairs and maintain a running sum s of the elements from nums1 that are currently in our heap.

Maintaining the Heap and Calculating Scores:

First iteration (pair (8, 3)): Push 3 to min heap q. Now q = [3] and running sum s = 3. Score if this subsequence is finalized: s \* a = 3 \* 8 = 24.

Second iteration (pair (4, 2)): Push 2 to min heap q. Now q = [2, 3] and running sum s = 3 + 2 = 5. Score if this subsequence is finalized: s \* a = 5 \* 4 = 20.

We don't need to remove any elements from the heap as it contains exactly k=2 elements.

Third iteration (pair (3, 7)): Although 7 from nums1 is larger, the corresponding 3 from nums2 would decrease the multiplier, so we continue without adding this pair to our heap/subsequence.

Fourth iteration (pair (1, 5)): We also skip this pair as adding 5 would lead to lowering the minimum value of nums2 to 1, which isn't beneficial.

Maximizing the Score: Throughout the process, we keep track of the running maximum score. After going through all pairs, the maximum score is from the second iteration with a score of 20.

Thus, the maximum score possible for the given example is 20.

Applying this solution approach allows us to efficiently find the maximum score without examining every possible subsequence, which could be intractably large for larger arrays. By prioritizing pairs based on nums2 values (the multipliers) and keeping a running sum of the largest nums1 values, we end up with the optimal subsequence.

**Java Solution**

import java.util.Arrays;

import java.util.PriorityQueue;

class Solution {

    public long maxScore(int[] nums1, int[] nums2, int k) {

        // Get the length of the given arrays

        int n = nums1.length;

        // Initialize an array of arrays to hold pairs from nums1 and nums2

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

        for (int i = 0; i < n; ++i) {

            numsPairs[i] = new int[] {nums1[i], nums2[i]};

        }

        // Sort the pairs based on the second element in decreasing order

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

        long maxScore = 0; // This will hold the maximum score

        long sum = 0; // This will hold the sum of the smallest 'k' elements from nums1

        // PriorityQueue to hold the smallest 'k' elements from nums1

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

        for (int i = 0; i < n; ++i) {

            sum += numsPairs[i][0]; // Add the value from nums1

            minHeap.offer(numsPairs[i][0]); // Add value to the min heap

            if (minHeap.size() == k) { // If we have 'k' elements in the min heap

                // Calculate potential score for the current combination and update maxScore if it's higher

                maxScore = Math.max(maxScore, sum \* numsPairs[i][1]);

                // Remove the smallest value to make room for the next iteration

                sum -= minHeap.poll();

            }

        }

        // Return the calculated max score

        return maxScore;

    }

}

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

**Time Complexity**

The given code performs several operations with distinct time complexities:

Sorting the combined list nums: This operation has a time complexity of O(n log n), where n is the length of the combined list. Since this length is determined by nums2, the time complexity is O(m log m) where m is the length of nums2.

Iterating over the sorted list nums: The iteration itself has a linear time complexity O(m).

Pushing elements onto a heap of size k: Each push operation has a time complexity of O(log k). Since we perform this operation m times, the total time complexity for all push operations is O(m log k).

Popping elements from the heap of size k: Each pop operation has a time complexity of O(log k), and since a pop operation is performed each time the heap size reaches k, this happens up to m times. The total time complexity for all pop operations is O(m log k).

Combining all of these operations, the overall time complexity of the code is O(m log m + m + m log k + m log k). Simplifying this expression, we get the final time complexity of O(m log m + 2m log k), which can be approximated to O(m log(mk)) since log m and log k are the dominating terms.

**Space Complexity**

The space complexity of the code is determined by:

The space required for the sorted list nums: This is O(m).

The space required for the heap q: In the worst case, the heap will have up to k elements, leading to a space complexity of O(k).

Therefore, the combined space complexity is O(m + k). Since one does not dominate the other, we represent them both in the final space complexity expression.

**Learn more about how to find time and space complexity quickly.**

**Refer to**

[L1383.Maximum Performance of a Team](note://WEBf5367f64c9712bbc2f91577b02b69305)

[L857.Minimum Cost to Hire K Workers](note://WEBd94d3e37d3953799acf25dbce88d5bf2)