2542. Maximum Subsequence Score

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

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.

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:

1. **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`.
2. **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`.
3. **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.
4. **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`.
5. **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 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.

1. **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)
2. **Min Heap Initialization**: We initialize an empty min heap `q`.
3. **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.
4. **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.

5. **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.

Python Solution

```python
from heapq import heappush, heappop

class Solution:
    def maxScore(self, cards1: List[int], cards2: List[int], k: int) -> int:
        # Combine the two lists into one by creating tuples of cards from cards2 and cards1
        # and sort the combined list in descending order based on cards2 values.
        combined_cards = sorted(zip(cards2, cards1), reverse=True)

        # Initialize a min-heap to keep track of the smallest elements.
        min_heap = []
        # Initialize sum and answer
        current_sum = 0
        max_score = 0

        # Iterate over the sorted list of combined cards.
        for card2_value, card1_value in combined_cards:
            # Add the value from cards1 to the current sum.
            current_sum += card1_value
            # Push the value from cards1 to the min-heap.
            heappush(min_heap, card1_value)

            # If the heap size reaches k, update the maximum score.
            # This corresponds to choosing k cards from cards1.
            if len(min_heap) == k:
                max_score = max(max_score, current_sum * card2_value)
                # Remove the smallest element from the current sum as we want the largest k elements.
                current_sum -= heappop(min_heap)

        # Return the maximum possible score.
        return max_score

```

Java Solution

```java
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;
    }
}

```

C++ Solution

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

class Solution {
public:
    long long maxScore(std::vector<int>& nums1, std::vector<int>& nums2, int k) {
        int n = nums1.size(); // Get the size of the input vectors
        std::vector<std::pair<int, int>> nums(n);

        // Combine the elements from nums2 and nums1 into pairs with a negative value from nums2
        for (int i = 0; i < n; ++i) {
            nums[i] = {-nums2[i], nums1[i]};
        }

        // Sort the vector of pairs based on the first element in non-decreasing order
        std::sort(nums.begin(), nums.end());

        // Use a min heap to keep track of the k largest elements from nums1
        std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;
        long long ans = 0, sum = 0;

        // Iterate over the sorted pairs
        for (auto& [negNum2, num1] : nums) {
            sum += num1; // Add the value from nums1 to the sum
            minHeap.push(num1); // Push the value from nums1 into the min heap

            // Once the heap size reaches k, we calculate the potential maximum score
            if (minHeap.size() == k) {
                // Current score is the sum times the negated value from nums2 (to make it positive again)
                ans = std::max(ans, sum * -negNum2);

                // Remove the smallest element from sum to maintain the top k largest elements
                sum -= minHeap.top();
                minHeap.pop(); // Remove the element from the heap
            }
        }

        // Return the maximum score found
        return ans;
    }
};

```

Typescript Solution

```typescript
// Importing the needed modules for priority queue functionality
import { PriorityQueue } from 'typescript-collections';

// Function to calculate the maximum score
function maxScore(nums1: number[], nums2: number[], k: number): number {
    const n: number = nums1.length; // Get the size of the input arrays

    // Initialize an array of pairs
    const nums: [number, number][] = new Array(n);

    // Combine the elements from nums2 and nums1 into pairs with a negative value from nums2
    for (let i = 0; i < n; ++i) {
        nums[i] = [-nums2[i], nums1[i]];
    }

    // Sort the array of pairs based on the first element in non-decreasing order
    nums.sort((a, b) => a[0] - b[0]);

    // Initialize a min heap to keep track of the k largest elements from nums1
    const minHeap = new PriorityQueue<number>((a, b) => b - a);
    let ans: number = 0;
    let sum: number = 0;

    // Iterate over the sorted pairs
    for (const [negNum2, num1] of nums) {
        sum += num1; // Add the value from nums1 to the sum
        minHeap.enqueue(num1); // Enqueue the value from nums1 onto the min heap

        // Once the heap size reaches k, calculate the potential maximum score
        if (minHeap.size() === k) {
            // Current score is the sum times the negated value from nums2 (to make it positive again)
            ans = Math.max(ans, sum * -negNum2);

            // Remove the smallest element from sum to maintain the top k largest elements
            sum -= minHeap.dequeue(); // Dequeue the smallest element from the min heap
        }
    }

    // Return the maximum score found
    return ans;
}

```

Time and Space Complexity

Time Complexity

The given code performs several operations with distinct time complexities:

1. 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`.
2. Iterating over the sorted list `nums`: The iteration itself has a linear time complexity `O(m)`.
3. 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)`.
4. 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:

1. The space required for the sorted list `nums`: This is `O(m)`.
2. 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.