2542. Maximum Subsequence Score

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 this 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**: For 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 (selected so far). We calculate the possible 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 one.

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:

```
1  nums = sort pairs (nums2, nums1) in descending order of nums2 values
2  q = initialize a min heap
3  s = 0 # Running sum of nums1 values in the heap
4  ans = 0 # Variable to store the maximum score
5  for each (a, b) in nums:
6      add b to the running sum s
7      push b onto the min heap q
8      if the size of heap q exceeds k:
9          remove the smallest element from q
10         subtract its value from the running sum s
11     update ans if (s * a) is greater than current ans
12 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 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 28.

Thus, the maximum score possible for the given example is 28.

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
1   from heapq import heappush, heappop
2
3   class Solution:
4       def maxScore(self, cards1: List[int], cards2: List[int], k: int) -> int:
5           # Combine the two lists into one by creating tuples of cards from cards2 and cards1
6           # and sort the combined list in descending order based on cards2 values.
7           combined_cards = sorted(zip(cards2, cards1), reverse=True)
8
9           # Initialize a min-heap to keep track of the smallest elements.
10          min_heap = []
11          # Initialize sum and answer
12          current_sum = 0
13          max_score = 0
14
15          # Iterate over the sorted list of combined cards.
16          for card2_value, card1_value in combined_cards:
17              # Add the value from card1 to the current sum.
18              current_sum += card1_value
19              # Push the value from card1 to the min-heap.
20              heappush(min_heap, card1_value)
21
22              # If the heap size reaches k, update the maximum score.
23              # This corresponds to choosing k cards from cards1.
24              if len(min_heap) == k:
25                  max_score = max(max_score, current_sum * card2_value)
26                  # Remove the smallest element from the current sum as we want the largest k elements.
27                  current_sum -= heappop(min_heap)
28
29          # Return the maximum possible score.
30          return max_score
```

Java Solution

```java
1   import java.util.Arrays;
2   import java.util.PriorityQueue;
3
4   class Solution {
5       public long maxScore(int[] nums1, int[] nums2, int k) {
6           // Get the length of the given arrays
7           int n = nums1.length;
8           // Initialize an array of arrays to hold pairs from nums1 and nums2
9           int[][] numsPairs = new int[n][2];
10          for (int i = 0; i < n; ++i) {
11              numsPairs[i] = new int[] {nums1[i], nums2[i]};
12          }
13
14          // Sort the pairs based on the second element in decreasing order
15          Arrays.sort(numsPairs, (a, b) -> b[1] - a[1]);
16
17          long maxScore = 0; // This will hold the maximum score
18          long sum = 0; // This will hold the sum of the smallest 'k' elements from nums1
19          PriorityQueue<Integer> minHeap = new PriorityQueue<>();
20          for (int i = 0; i < n; ++i) {
21              sum += numsPairs[i][0]; // Add the value from nums1
22              minHeap.offer(numsPairs[i][0]); // Add value to the min heap
23
24              if (minHeap.size() == k) { // If we have 'k' elements in the min heap
25                  // Calculate potential score for the current combination and update maxScore if it's higher
26                  maxScore = Math.max(maxScore, sum * numsPairs[i][1]);
27                  // Remove the smallest value to make room for the next iteration
28                  sum -= minHeap.poll();
29              }
30          }
31          // Return the calculated max score
32          return maxScore;
33      }
34  }
```

C++ Solution

```cpp
1   #include <vector>
2   #include <queue>
3   #include <algorithm>
4
5   class Solution {
6   public:
7       long long maxScore(std::vector<int>& nums1, std::vector<int>& nums2, int k) {
8           int n = nums1.size(); // Get the size of the input vectors
9           std::vector<std::pair<int, int>> nums(n);
10
11          // Combine the elements from nums2 and nums1 into pairs with a negative value from nums2
12          for (int i = 0; i < n; ++i) {
13              nums[i] = {-nums2[i], nums1[i]};
14          }
15
16          // Sort the vector of pairs based on the first element in non-decreasing order
17          std::sort(nums.begin(), nums.end());
18
19          // Use a min heap to keep track of the k largest elements from nums1
20          std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;
21          long long ans = 0, sum = 0;
22
23          // Iterate over the sorted pairs
24          for (auto& [negNum2, num1] : nums) {
25              num1 += num1; // Add the value from nums1 to the sum
26              minHeap.push(num1); // Push the value from num1 into the min heap
27
28              // Once the heap size reaches k, we calculate the potential maximum score
29              if (minHeap.size() == k) {
30                  // Current score is the sum times the negated value from num2 (to make it positive again)
31                  ans = std::max(ans, sum * -negNum2);
32
33                  // Remove the smallest element from sum to maintain the top k largest elements
34                  sum -= minHeap.top();
35                  minHeap.pop(); // Remove the element from the heap
36              }
37          }
38
39          // Return the maximum score found
40          return ans;
41      }
42  };
```

Typescript Solution

```typescript
1   // Importing the needed modules for priority-queue functionality
2   import { PriorityQueue } from 'typescript-collections';
3
4   // Function to calculate the maximum score
5   function maxScore(nums1: number[], nums2: number[], k: number): number {
6       const n: number = nums1.length; // Get the size of the input arrays
7
8       // Initialize an array of pairs
9       const nums: [number, number][] = new Array(n);
10
11      // Combine the elements from nums2 and nums1 into pairs with a negative value from nums2
12      for (let i = 0; i < n; ++i) {
13          nums[i] = [-nums2[i], nums1[i]];
14      }
15
16      // Sort the array of pairs based on the first element in non-decreasing order
17      nums.sort((a, b) => a[0] - b[0]);
18
19      // Initialize a min heap to keep track of the k largest elements from nums1
20      const minHeap = new PriorityQueue<number>((a, b) => b - a);
21      let ans: number = 0;
22      let sum: number = 0;
23
24      // Iterate over the sorted pairs
25      for (const [negNum2, num1] of nums) {
26          sum += num1; // Add the value from num1 to the sum
27          minHeap.enqueue(num1); // Enqueue the value from num1 onto the min heap
28
29          // Once the heap size reaches k, calculate the potential maximum score
30          if (minHeap.size() === k) {
31              // Current score is the sum times the negated value from num2 (to make it positive again)
32              ans = Math.max(ans, sum * -negNum2);
33
34              // Remove the smallest element from sum to maintain the top k largest elements
35              sum -= minHeap.dequeue(); // Dequeue the smallest element from the min heap
36          }
37      }
38
39      // Return the maximum score found
40      return ans;
41  }
```

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(n log n) where n is the length of nums2.

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

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 n times, the total time complexity for all push operations is O(n 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 n times. The total time complexity for all pop operations is O(n log k).

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

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(n + k). Since one does not dominate the other, we represent them both in the final space complexity expression.