2099. Find Subsequence of Length K With the Largest Sum

Easy Array Hash Table Sorting Heap (Priority Queue) Leetcode Link

Problem Description

In this problem, we are given an array of integers named `nums` and another integer `k`. Our goal is to find a subsequence of the array `nums` that consists of `k` elements and has the largest possible sum. A subsequence is defined as a sequence that can be obtained from the original array by removing some or no elements, without changing the order of the remaining elements. The problem statement requires us to return any subsequence that satisfies the condition of having `k` elements and the largest sum. If there are multiple subsequences with the same largest sum, we can return any one of them.

Intuition

To approach this solution, we need to focus on finding the elements that would contribute to the largest sum. Naturally, larger numbers will contribute more to the sum than smaller numbers. Therefore, the subsequence with the largest sum within a given size `k` will always include the `k` largest elements from the original `nums` array.

However, since we are interested in a subsequence, it's important to maintain the original order of elements. The provided solution achieves this by first finding the indices of the `k` largest elements, and then reconstructing the subsequence by accessing the elements using these indices in their sorted order.

1. The solution starts by creating a list of indices `idx` for all elements in `nums`.
2. It sorts this list of indices `idx` based on the values in `nums` that they correspond to, using a lambda function as the key to the `sort` method.
3. By sorting `idx`, the last `k` indices now represent the largest `k` numbers in `nums`.
4. The subsequence is then created by selecting the elements of `nums` using the sorted list of the last `k` indices.
5. Since we need to return the subsequence in its original order, we sort the list of the last `k` indices, ensuring that the corresponding `k` elements will be in the same order as they appeared in `nums`.

By following this method, we can efficiently retrieve any subsequence of length `k` that yields the largest sum while maintaining its original order from `nums`.

Solution Approach

In the provided reference solution, several key programming concepts and Python-specific tools are employed to extract the `k` largest sum subsequence:

1. List Comprehension: Python's list comprehension is used to concisely generate lists without writing out complex for-loops. In this solution, list comprehensions are used twice, once to generate the sorted indices and then to generate the actual subsequence.

2. Sorting with Custom Key Function: `list.sort()` method is used with a custom key function. The key function is provided by a lambda expression `lambda i: nums[i]` which sorts the list of indices `idx` based on the value of elements in `nums` at each index.

3. Slicing: Python's slicing operation `[-k:]` is used to obtain the last `k` elements from the sorted list of indices, which represents the indices of the `k` largest numbers.

The steps to implement this approach are as follows:

- Start by creating a list of indices `idx` with `range(len(nums))` which basically gives us a list `[0, 1, 2, ..., len(nums) - 1]`. Each index here is a direct reference to the corresponding element in `nums`.

- Next, sort the list `idx` using the `sort` method with a key that references the original list's values `nums[i]`. After sorting, for an array `nums = [1, 3, 5, 7, 9]`, and say `k = 3`, the `idx` array will look like `[4, 3, 2, 1, 0]` because we are sorting by the values of `nums` in descending order.

- Slice the last `k` elements from the sorted `idx` to get the indices of the `k` largest elements. For our example, we will get `[4, 3, 2]`.

- Before creating the final subsequence, we need to ensure that its order is the same as the original array's order. We achieve this by sorting the slice of indices.

- Finally, the solution applies the sorted indices to `nums` to produce the subsequence with the largest sum. We use a list comprehension to achieve this: `[nums[i] for i in sorted(idx[-k:])]`.

This algorithm effectively combines Python's powerful list manipulation features to provide a simple yet efficient solution. The complexity of the solution is dominated by the sorting step, which is typically O(n log n) where `n` is the number of elements in `nums`.

Example Walkthrough

Let's take a small sample array `nums = [7, 1, 5, 3, 6, 4]` and assume we want a subsequence of length `k = 3` with the largest sum.

Initial Steps

1. First, create an array of indices, `idx`, which will initially be `[0, 1, 2, 3, 4, 5]`.

Sorting by Reference to `nums` Values

2. Next, we sort the `idx` array while referencing the elements it points to in `nums`. The sorting will be done in descending order based on the values in `nums`, which means the highest numbers come first:
   - After sorting, `idx` becomes `[0, 4, 2, 5, 3, 1]` since the corresponding `nums` values are `[7, 6, 5, 4, 3, 1]`.

Slicing to Get Largest Elements

3. Then we take the last `k` elements of the sorted `idx`. In this case `k = 3`, so we slice the last three indices, getting `[2, 5, 3]` which corresponds to `nums` values `[5, 4, 3]`.

Sorting Indices to Maintain Original Order

4. Before creating the final subsequence, sort the slice `[2, 5, 3]` to maintain the original order of elements from `nums`. When we sort this slice, we get `[2, 3, 5]`.

Creating the Final Subsequence

5. Using these sorted indices, we construct our subsequence by taking the values from `nums` at these positions, hence `[nums[i] for i in sorted(idx[-k:])]` becomes `[nums[2], nums[3], nums[5]]` which evaluates to `[5, 3, 4]`.

Result

The subsequence `[5, 3, 4]` has a sum of `12`, which is the largest sum possible for any 3 element subsequence in the original `nums`. Thus, the final answer for this example is `[5, 3, 4]`.

In summary, by identifying and extracting the indices of the `k` largest numbers, sorting those indices to maintain the initial array's order, and then building a subsequence from these indices, we maximally leverage Python's list manipulation abilities to efficiently solve the problem.

Python Solution

```python
class Solution:
    def maxSubsequence(self, nums: List[int], k: int) -> List[int]:
        # Create a list of indices that correspond to the elements in 'nums'.
        indices = list(range(len(nums)))

        # Sort the list of indices based on the values in 'nums' they point to.
        indices.sort(key=lambda i: nums[i])

        # Select the last 'k' elements from the sorted indices since they point to
        # the elements with the 'k' largest values in 'nums'.
        largest_indices = indices[-k:]

        # Sort the selected indices to maintain the original order of 'nums'.
        sorted_largest_indices = sorted(largest_indices)

        # Return the subsequence of 'nums' pointed by the sorted largest indices,
        # which constitutes the k-largest elements in their original order.
        max_subsequence = [nums[i] for i in sorted_largest_indices]

        return max_subsequence

```

Java Solution

```java
class Solution {
    public int[] maxSubsequence(int[] nums, int k) {
        // Initialize an array 'ans' to store the result subsequence of length k
        int[] ans = new int[k];

        // Create a list 'indices' to keep track of the original indices of the array elements
        List<Integer> indices = new ArrayList<>();

        // Loop to fill 'indices' with the array indices
        for (int i = 0; i < nums.length; ++i) {
            indices.add(i);
        }

        // Sort 'indices' based on the values in 'nums' from highest to lowest
        indices.sort((i1, i2) -> Integer.compare(nums[i2], nums[i1]));

        // Initialize a temporary array 'topIndices' to store the first k sorted indices
        int[] topIndices = new int[k];
        for (int i = 0; i < k; ++i) {
            topIndices[i] = indices.get(i);
        }

        // Sort 'topIndices' to maintain the original order of selected k elements
        Arrays.sort(topIndices);

        // Fill the 'ans' array with the elements corresponding to the sorted indices
        for (int i = 0; i < k; ++i) {
            ans[i] = nums[topIndices[i]];
        }

        // Return the result array containing the max subsequence of length k
        return ans;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Method to find the subsequence of 'k' numbers with the largest sum
    vector<int> maxSubsequence(vector<int>& nums, int k) {
        // Size of the input array
        int numSize = nums.size();

        // Pair of index and value from the input array
        vector<pair<int, int>> indexedNums;

        // Populate the indexedNums with pairs of indices and their respective values
        for (int i = 0; i < numSize; ++i) {
            indexedNums.push_back({i, nums[i]});
        }

        // Sort the indexedNums by their values in descending order
        sort(indexedNums.begin(), indexedNums.end(), [](const auto& x1, const auto& x2) {
            return x1.second > x2.second;
        });

        // Sort only the first 'k' elements of indexedNums by their original indices to maintain the original order
        sort(indexedNums.begin(), indexedNums.begin() + k,
             [](const auto& x1, const auto& x2) {
                 return x1.first < x2.first;
             });

        // Prepare a vector to store the answer subsequence
        vector<int> ans;
        ans.reserve(k); // Reserve space for 'k' elements to avoid reallocations

        // Populate the answer vector with the 'k' largest elements in their original order
        for (int i = 0; i < k; ++i) {
            ans.push_back(indexedNums[i].second);
        }

        // Return the final answer subsequence
        return ans;
    }
};

```

Typescript Solution

```typescript
function maxSubsequence(nums: number[], k: number): number[] {
    // Size of the input array
    let numSize: number = nums.length;

    // Pair of index and value from the input array
    let indexedNums: { index: number, value: number }[] = [];

    // Populate the indexedNums with objects containing indices and their respective values
    for (let i = 0; i < numSize; ++i) {
        indexedNums.push({ index: i, value: nums[i] });
    }

    // Sort the indexedNums by their values in descending order
    indexedNums.sort((a, b) => b.value - a.value);

    // Sort only the first 'k' elements of indexedNums by their original indices to maintain the original order
    let firstKElements: { index: number, value: number }[] = indexedNums.slice(0, k);
    firstKElements.sort((a, b) => a.index - b.index);

    // Prepare an array to store the answer subsequence
    let answer: number[] = firstKElements.map(element => element.value);

    // Return the final answer subsequence
    return answer;
}

```

Time and Space Complexity

The time complexity of the code is as follows:

1. Creating the `idx` list with list comprehension has a time complexity of `O(n)` where `n` is the number of elements in `nums`.

2. Sorting the `idx` list using the key, which is based on the values in `nums`, with the `sort()` function is `O(n log n)`.

3. Slicing the last `k` elements from the sorted `idx` list is `O(k)` because it requires iterating over the `k` elements to create a new list.

4. Sorting the sliced list of `k` indices is `O(k log k)`.

5. The list comprehension in the return statement to create the final list of numbers from their indices takes `O(k)`.

The overall time complexity is therefore dominated by the `O(n log n)` step, which is the sorting of the `idx` list.

The space complexity of the code is:

1. The additional list `idx` that stores the indices takes `O(n)` space.

2. No additional space other than variables for sorting and slicing are used, which does not depend on the size of the input and hence is `O(1)`.

3. The output list that is returned has `k` elements, so it takes `O(k)` space.

Therefore, the total space complexity is `O(n + k)`, since you need to store the indices and the final output list.