2386. Find the K-Sum of an Array

Hard | Array | Sorting | Heap (Priority Queue) | Leetcode Link

Problem Description

In this problem, you receive an array of integers called `nums` and a positive integer `k`. The goal is to figure out the `k`th largest sum of a subsequence within this array. A subsequence is a sequence that can be derived from the array by removing some or none of the items, while maintaining the order of the remaining items. Additionally, it's important to note that the empty subsequence (a sequence with no elements) has a sum of 0, and this should be considered when determining the K-Sum. The challenge then lies in efficiently finding the K-Sum because as the array grows with more elements, the number of subsequences that can be generated increases exponentially.

Intuition

To arrive at the solution, consider the nature of subsequences and the sums they generate. If the array contains positive numbers only, the largest subsequence sum is the sum of the entire array, and the smallest would be 0 (the empty subsequence). However, negative numbers complicate this because including a negative number in a subsequence can reduce the sum, so it may not always be beneficial to include all positive numbers in a subsequence.

The intuition behind the solution involves two key insights:

1. **Max Sum with Positive Numbers**: First, we calculate the maximum possible sum of the array as if all numbers were positive. This is because the highest K-Sum, specifically the 1st K-Sum, will not include any of the negative numbers (assuming we're only dealing with positive or non-positive numbers now).

2. **A Priority Queue to Find the K-th Sum**: After transforming all negative numbers into positive, we can sort the array. We then use a min-heap (priority queue) to systematically generate the sums of subsequences. By doing this, we ensure that we keep track of the smallest sums we can pop from the heap until we reach the `k`th smallest. We consider two operations with the heap: either we add the next number in the sorted array to the current sum (expanding the subsequence by one), or we replace the last added number with the next number (changing the last element of the subsequence), but only if it's not the first element in the subsequence (to avoid duplicates).

These steps allow us to cleverly sidestep the brute-force approach of generating every possible subsequence and its sum, which would be impractical for larger arrays.

Solution Approach

The solution approach utilizes several algorithms and data structures to efficiently find the K-Sum of the array.

1. **Dealing with Max Sum**: We initialize a variable `mx` with 0 to store the maximum sum. We iterate over `nums` and check each value. If the value is positive, we add it to `mx`. If it is negative (or non-positive, to be more general), it means including this value in our subsequence would decrease the sum. So, we take the absolute value of that number and then include it in the array `nums`, effectively transforming the array to contain only non-negative numbers. Later, we sort this transformed array, which is crucial for the priority queue operations.

2. **Priority Queue (Min-Heap)**: A priority queue is implemented using a min-heap to keep track of the smallest sums that could lead to the K-Sum when combined with the subsequent elements in the array. We use this to efficiently get the `k`th smallest sum. The heap is initialized with a tuple containing 0 (the sum of the empty subsequence) and 0 (the index indicating we haven't started to use any number from `nums`).

3. **Heap Operation to Find K-th Largest Sum**: For each `k-1` iterations (since we're trying to find the `k`th largest, we do one less), we pop the smallest sum from the heap. For each popped sum, which comes as a tuple `(sum, index)`, we check if `index` is less than the length of `nums`. If it is, we push a new tuple onto the heap: the current sum `s` plus the number at the current index `nums[i]`, and increment the index by 1. This operation represents expanding the subsequence to include the `nums[i]`.

4. **Avoiding Duplication**: If the index `i` is not zero, we push yet another tuple onto the heap which accounts for swapping the last number of the subsequence with the current number at index `i` of `nums`. The calculation is `s + nums[i] - nums[i-1]`, and the index is incremented by 1. This operation is crucial because it effectively generates sums that include the current number, but not the previous, preventing overcounting and ensuring that all subsequences considered are unique.

5. **Retrieve the K-th Largest Sum**: After performing the for-loop of `k-1` iterations, the heap's smallest sum is the `k`th largest subsequence sum because all smaller sums have been popped off the heap already.

6. **Calculating the Final Answer**: Since `mx` represents the largest sum we could possibly make without negative numbers, and `h[0][0]` contains the smallest sum at the `k`th position, subtracting `h[0][0]` from `mx` gives us the actual value of the `k`th largest subsequence sum, even when considering the original array with its negative numbers.

In summary, the implementation cleverly leverages a heap to simulate the process of generating subsequence sums and smartly navigates the subsequence space to find the K-Sum efficiently without generating all possible subsequences.

Example Walkthrough

Let's use a small example to illustrate the solution approach.

Consider the array `nums = [3, -1, 2]` and `k = 2`. The task is to find the 2nd largest sum of a subsequence within this array.

1. **Dealing with Max Sum**: First, we initialize `mx` with 0. After iterating over `nums`, we will transform it into `[3, 1, 2]` as we convert the negative number to a positive by taking the absolute value. The `mx` will now be the sum of this array, which is `6`.

2. **Priority Queue (Min-Heap)**: Initialize a min-heap that starts with the tuple `(0, 0)`, indicating the sum of the empty subsequence and the starting index.

3. **Heap Operation to Find 2nd Largest Sum**: Now, we will perform `k - 1` iterations over the heap to find the 2nd largest sum:

   - Pop the top of the heap which is `(0, 0)` and since `index` is less than the length of `nums`, push `(0 + nums[0], 0 + 1)` which is `(3, 1)` and `(0 + nums[1], 1 + 1)` which is `(1, 2)` into the heap.
   - Now the heap contains `(1, 2)` and `(3, 1)`. We pop the smallest sum again, which is `(1, 2)`, and since `index` is less than the length of `nums`, we push `(1 + nums[2], 2 + 1)` which is `(3, 3)` into the heap.

4. **Avoiding Duplication**: There isn't any duplication avoidance necessary in this example since there are no numbers in `nums` to follow `nums[2]`.

5. **Retrieve the 2nd Largest Sum**: After `k - 1` iterations, our heap contains `(3, 1)` and `(3, 3)`. The smallest sum at the top of the heap is `(3, 1)`, which represents the 2nd largest subsequence sum.

6. **Calculating the Final Answer**: `mx` is `6`, and the `k`th smallest sum from our heap is `3`. So, the 2nd largest subsequence sum is `mx - heap's top sum`, which equals `6 - 3 = 3`.

The 2nd largest sum of a subsequence in the array `[3, -1, 2]` is `3`. This sum can be obtained from the subsequence `[3]` or `[2, -1, 3]` in the original array before we transformed the negative numbers into positive.

Python Solution

```python
from heapq import heappush, heappop

class Solution:
    def kthLargestSum(self, nums: List[int], k: int) -> int:
        # Calculate the sum of positive elements, and make all elements positive
        max_sum = 0
        for index, value in enumerate(nums):
            if value > 0:
                max_sum += value
            else:
                nums[index] = -value
        nums.sort()  # Sort the array

        # Create a min-heap to store the sums and their corresponding indexes
        min_heap = [(0, 0)]

        # Pop and push the next two sums to the heap (k - 1) times
        for _ in range(k - 1):
            current_sum, index = heappop(min_heap)
            if index < len(nums):
                # Push the next sum which includes the nums[index]
                heappush(min_heap, (current_sum + nums[index], index + 1))

                # Avoid duplicating the first element of the sorted nums array
                if index:
                    # Calculate the sum by swapping out the previous element (thus maintaining the count of elements in current sum)
                    heappush(min_heap, (current_sum + nums[index] - nums[index - 1], index + 1))

        # The k-th largest sum is the max_sum minus the smallest sum in the heap
        return max_sum - min_heap[0][0]

# Note: The code assumes the existence of `List` imported from `typing` module.
# If not present, add: from typing import List
```

Java Solution

```java
import java.util.PriorityQueue;
import java.util.Comparator;
import java.util.Arrays;
import javafx.util.Pair; // Make sure to import the correct package for the Pair class based on the development environment

class Solution {
    public long kSum(int[] nums, int k) {
        // Initialize variable to store the sum of positive numbers.
        long maxSumPositives = 0;
        int length = nums.length;

        // Convert negative numbers to positive and sum all positive numbers.
        for (int i = 0; i < length; ++i) {
            if (nums[i] > 0) {
                maxSumPositives += nums[i];
            } else {
                nums[i] = -nums[i];
            }
        }

        // Sort the modified array in non-decreasing order.
        Arrays.sort(nums);

        // Use a priority queue to keep track of the sum-pairs efficiently.
        PriorityQueue<Pair<Long, Integer>> minHeap = new PriorityQueue<>(Comparator.comparing(Pair::getKey));
        minHeap.offer(new Pair<>(0L, 0)); // Offer initial pair of (sum=0, index=0).

        // Loop until we find the kth smallest sum.
        while (--k > 0) {
            Pair<Long, Integer> currentPair = minHeap.poll(); // Poll the smallest sum pair.
            long currentSum = currentPair.getKey();
            int currentIndex = currentPair.getValue();

            // If there is a next index, offer new pairs into the priority queue.
            if (currentIndex < length) {
                minHeap.offer(new Pair<>(currentSum + nums[currentIndex], currentIndex + 1));
                // Avoid duplicate sums by checking if currentIndex is greater than 0.
                if (currentIndex > 0) {
                    minHeap.offer(new Pair<>(currentSum + nums[currentIndex] - nums[currentIndex - 1], currentIndex + 1));
                }
            }
        }

        // Return the maximum sum of positives minus the k-th smallest sum (the top element in the queue).
        return maxSumPositives - minHeap.peek().getKey();
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;

// Define pair type with long long and int for use in priority queue
using PairLLInt = pair<long long, int>;

class Solution {
public:
    long long kSum(vector<int>& nums, int k) {
        long long maxSumNonNegatives = 0; // To store the sum of non-negative numbers
        int size = nums.size();
        // Convert negative numbers to positive and calculate maxSumNonNegatives
        for (int i = 0; i < size; ++i) {
            if (nums[i] > 0) {
                maxSumNonNegatives += nums[i];
            } else {
                nums[i] = -nums[i];
            }
        }
        // Sort nums to utilize the non-decreasing property in generating sums
        sort(nums.begin(), nums.end());

        // Min-heap to store the smallest sums and their indices
        priority_queue<PairLLInt, vector<PairLLInt>, greater<PairLLInt>> minHeap;
        minHeap.push({0, 0}); // Initial pair

        // Generate the first k-1 sums
        while (--k) {
            auto [sum, idx] = minHeap.top();
            minHeap.pop();

            if (idx < size) {
                minHeap.push({sum + nums[idx], idx + 1});
                // Avoid duplicates by checking if the current sum results from
                // a combination of previous numbers
                if (idx > 0) {
                    minHeap.push({sum + nums[idx] - nums[idx - 1], idx + 1});
                }
            }
        }

        // k-th sum is the smallest sum which is the difference
        // between maxSumNonNegatives and the top element of min-heap
        return maxSumNonNegatives - minHeap.top().first;
    }
};
```

Typescript Solution

```typescript
function kSum(nums: number[], k: number): number {
    let maxSumNonNegatives: number = 0; // Store the sum of non-negative numbers
    let size: number = nums.length;

    // Convert negative numbers to positive and calculate maxSumNonNegatives
    for (let i = 0; i < size; ++i) {
        if (nums[i] > 0) {
            maxSumNonNegatives += nums[i];
        } else {
            nums[i] = -nums[i];
        }
    }

    // Sort nums to utilize its non-decreasing property in generating sums
    nums.sort((a, b) => a - b);

    // Define the priority queue to store the smallest sums and their indices
    let minHeap: Array<[number, number]> = [];

    // Function to add elements to the min-heap
    const addToMinHeap = (sum: number, index: number): void => {
        minHeap.push([sum, index]);
        // Percolate up to maintain heap invariant
        let current = minHeap.length - 1;
        while (current > 0) {
            let parent = Math.floor((current - 1) / 2);
            if (minHeap[current][0] < minHeap[parent][0]) {
                [minHeap[current], minHeap[parent]] = [minHeap[parent], minHeap[current]];
                current = parent;
            } else {
                break;
            }
        }
    };

    // Function to extract the smallest element from the min-heap
    const extractFromMinHeap = (): [number, number] => {
        const smallest = minHeap[0];
        const lastElement = minHeap.pop()!;
        if (minHeap.length > 0) {
            minHeap[0] = lastElement;
            // Percolate down to maintain heap invariant
            let current = 0;
            while (true) {
                let leftChild = current * 2 + 1;
                let rightChild = current * 2 + 2;
                let smallestChild = current;

                if (leftChild < minHeap.length && minHeap[leftChild][0] < minHeap[smallestChild][0]) {
                    smallestChild = leftChild;
                }
                if (rightChild < minHeap.length && minHeap[rightChild][0] < minHeap[smallestChild][0]) {
                    smallestChild = rightChild;
                }
                if (smallestChild !== current) {
                    [minHeap[current], minHeap[smallestChild]] = [minHeap[smallestChild], minHeap[current]];
                    current = smallestChild;
                } else {
                    break;
                }
            }
        }
        return smallest;
    };

    addToMinHeap(0, 0); // Initial element

    // Generate the first k-1 sums
    while (--k) {
        const [sum, idx] = extractFromMinHeap();

        if (idx < size) {
            addToMinHeap(sum + nums[idx], idx + 1);
            // Avoid duplicates by checking if the current sum is from
            // a combination of previous numbers
            if (idx > 0) {
                addToMinHeap(sum + nums[idx] - nums[idx - 1], idx + 1);
            }
        }
    }

    // k-th sum is the smallest sum which is the difference
    // between maxSumNonNegatives and the top element of min-heap
    const [smallestSum, ] = extractFromMinHeap();
    return maxSumNonNegatives - smallestSum;
}

// Usage example:
// const nums: number[] = [1, 2, 3];
// const k: number = 2;
// console.log(kSum(nums, k)); // Call the kSum function with the example parameters
```

Time and Space Complexity

Time Complexity

The time complexity of the `kSum` function involves several components:

1. **Initialization and Sorting**: Initializing `mx` and converting negative numbers in the `nums` list to positive ones takes `O(n)` time where `n` is the length of `nums`. Sorting the modified `nums` list takes `O(n log n)` time.

2. **Heap Operations**: The function performs a series of heap operations, namely `heappop` and `heappush`, within a loop that runs `k - 1` times. Each heap operation can be done in `O(log k)` time. However, in the worst-case scenario, there can be up to `2*(k-1)` elements in the heap, as for each pop operation, potentially two elements are pushed back. This leads to a time complexity of `O((k - 1) * log(k))` for all heap operations.

Combining these operations, the overall time complexity is determined by `O(n log n + (k - 1) * log(k))`. Since the heap operations depend on `k`, the larger of `n log n` and `(k - 1) * log(k)` will dominate the time complexity.

Thus, the total time complexity is `O(max(n log n, (k - 1) * log(k)))`.

Space Complexity

The space complexity of the `kSum` function consists of:

1. **Heap Space**: The heap can contain up to `2*(k-1)` elements in the worst-case scenario, which contributes `O(k)` space complexity.

2. **Other Variables**: The space used by variables `mx`, `s`, and `i`, is `O(1)`.

Therefore, the total space complexity of the function is `O(k)` since this is the term that grows with the input size and dominates the space complexity.