373. Find K Pairs with Smallest Sums

Medium Array Heap (Priority Queue)

Problem Description

In this problem, we are working with two key concepts: pairs and sum minimization. We have two arrays, `nums1` and `nums2`, both sorted in non-decreasing order. Our task is to find `k` pairs, each composed of one element from each array. The objective is to identify the pairs with the smallest possible sums. Specifically, we want to find the `k` pairs whose sums are the smallest among all possible combinations from `nums1` and `nums2`.

If we were to list all possible pairs and their sums, the list would be quite extensive. However, since the arrays are already sorted, we can infer that the smallest sums will involve the smallest elements from both arrays.

Intuition

The solution employs a min-heap, a binary tree where the parent node is always less than or equal to its children. This is an ideal data structure for efficiently finding the smallest elements.

The algorithm initializes by creating a min-heap and pre-filling it with pairs combining each of the first `k` elements of `nums1` with the first element of `nums2`.

Here's the step-by-step process to understand the solution:

1. Initialize the min-heap `q`. We start by forming pairs by taking each element from `nums1` (up to the first `k` elements to avoid unnecessary work) and pairing it with the first element in `nums2`. The reason for pairing with the first element from `nums2` is that we are trying to create pairs with the smallest possible sum to begin with.
2. Heapify `q` to ensure the smallest sum pair is at the top of the heap.
3. Prepare a list `ans` to store the resulting k smallest sum pairs.
4. Use a loop to process the min-heap. While the min-heap is not empty and `k` is greater than 0, we do the following:
   - Pop the smallest sum pair from the heap. This pair is guaranteed to be one of the smallest sum pairs because of the min-heap's properties.
   - Append this pair to the `ans` list.
   - Decrease `k` by 1, as we have successfully found one of the k smallest sum pairs.
   - Check if there's a next element in `nums2` that can be paired with the same `nums1` element. If so, add this new pair to the heap so that it can be considered in the next iteration.
5. Return the `ans` list containing up to `k` smallest sum pairs.

By utilizing a min-heap, we ensure that at each step, we pop the minimum sum pair without having to check all combinations. Moreover, since `nums1` and `nums2` are sorted, we can confidently move to the next element in `nums2` for potential pairing without worrying about missing smaller sums.

Solution Approach

The solution we're discussing uses a min-heap to efficiently find the `k` pairs with the smallest sums. Let's walk through the implementation step by step, taking advantage of algorithms, data structures, and patterns.

1. **Min-Heap Initialization and Heapify**: The code snippet `q = [[u + nums2[0], i, 0] for i, u in enumerate(nums1[:k])]` initializes our min-heap `q`. For each element in `nums1` (limited to the first `k` elements), we create a list with the sum (`u + nums2[0]`), the index of the `nums1` element (`i`), and the index `0` indicating we've used the first element of `nums2`. The `heapify(q)` call then converts the list `q` into a heap data structure.
2. **Processing the Min-Heap**: The `while` loop continues as long as there are elements in the heap (i.e., potential pairs to consider) and `k` is greater than 0. In each iteration, we use `heappop(q)` to remove and return the smallest element from the heap, which is the pair with the current smallest sum.
3. **Recording the Result**: The current smallest sum pair's indices in `nums1` and `nums2` are stored in the `_`, `i`, `j` variables. We then append this pair `[nums1[i], nums2[j]]` to our answer list `ans`.
4. **Generating New Pairs**: After finding a pair with a small sum, we check if there is a next element in `nums2` to pair with the same `nums1` element. The `if j + 1 < len(nums2)` check is used for this purpose, and if true, we form a new pair with the same element from `nums1` and the next element in `nums2`, then push it into the heap using `heappush(q, [nums1[i] + nums2[j + 1], i, j + 1])` to make sure it can be considered in subsequent iterations.
5. **Finding the k Smallest Sums**: The heap keeps track of the current smallest pairs, and the loop ensures that each time we pop from the heap and push a new pair into it, we're considering the next smallest possible pair. This continues until we've found `k` pairs or there are no more new pairs to consider.

The key algorithms and data structures used here are the heap data structure (a min-heap in particular) for keeping track of the smallest pairs and a list for our result.

The pattern is utilizing a greedy-like approach: in each step, we're always taking the smallest available option (the pair with the smallest sum) and then exploring slightly larger options by considering the next possible element from `nums2`.

This approach results in a time-efficient solution because we avoid generating all possible pairs, which would have resulted in a much higher time complexity.

Example Walkthrough

Let's walk through a small example using the provided solution approach to understand how it works.

Consider the following input:

- `nums1 = [1, 7, 11]`
- `nums2 = [2, 4, 6]`
- `k = 3`

Following the steps of the solution approach:

1. **Min-Heap Initialization and Heapify**: We initialize the min-heap with elements paired as follows: `(nums1[i] + nums2[0], i, 0)`. Since we're only interested in the first `k` pairs, and `k` is `3`, we consider only the first three elements from `nums1`. Therefore, our min-heap `q` starts with the following elements (assuming we include all three elements of `nums1` and `k` limits other factors):

   ```
   q = [
     (1 + 2, 0, 0),   # sum = 3, index in nums1 = 0, index in nums2 = 0
     (7 + 2, 1, 0),   # sum = 9, index in nums1 = 1, index in nums2 = 0
     (11 + 2, 2, 0),  # sum = 13, index in nums1 = 2, index in nums2 = 0
   ]
   ```

   After heapification, `q` ensures that the pair with the smallest sum is at the top of the heap.
2. **Processing the Min-Heap**: We start processing the heap by popping elements while `k > 0`. The smallest sum pair is at the top, so we pop `(3, 0, 0)` first.
3. **Recording the Result**: We extract the indices `0` from `nums1` and `0` from `nums2` and append the pair `[1, 2]` to our result list `ans`. Now, `ans = [[1, 2]]`.
4. **Generating New Pairs**: Since `nums1[0]` was paired with `nums2[0]`, we now pair `nums1[0]` with the next element in `nums2`, which is `nums2[1]` (value `4`), and add this new pair to the heap. Now, the heap `q` looks like:

   ```
   q = [
     (1 + 4, 0, 1),   # sum = 5, new pair
     (7 + 2, 1, 0),
     (11 + 2, 2, 0)
   ]
   ```

   Heapification ensures the smallest pair stays at the top, which is now `(5, 0, 1)`.
5. **Finding the k Smallest Sums**: We continue the process, now popping `(5, 0, 1)` from the heap and appending `[1, 4]` to `ans`. Subsequently, we form a new pair with `nums1[0]` and `nums2[2]`, and the heap `q` gets updated after heapification. We repeat this process until `k` becomes `0`.

After these steps, our result list `ans` is `[[1, 2], [1, 4], [1, 6]]`, which are the `k` smallest sum pairs from `nums1` and `nums2`.

By following this strategy, we avoid generating all possible pairs and efficiently find the k smallest sums using a min-heap to guide our pair selections.

Python Solution

```python
from typing import List
from heapq import heapify, heappop, heappush

class Solution:
    def k_smallest_pairs(self, nums1: List[int], nums2: List[int], k: int) -> List[List[int]]:
        # Create a priority queue to hold the sum of pairs along with the indices in nums1 and nums2
        # Only consider the first k numbers in nums1 for initial pairing, as we are looking for the k smallest pairs
        priority_queue = [[u + nums2[0], i, 0] for i, u in enumerate(nums1[:k])]
        heapify(priority_queue)  # Convert list into a heap

        result = []  # Initialize a list to hold the result pairs

        # Iterate until the priority queue is empty or we have found k pairs
        while priority_queue and k > 0:
            # Pop the smallest sum pair from the heap
            sum_pair, index1, index2 = heappop(priority_queue)

            # Append the corresponding values from nums1 and nums2 to the result
            result.append([nums1[index1], nums2[index2]])

            k -= 1  # Decrement k as we have found one of the k smallest pairs

            # If there are more elements in nums2 to pair with the current nums1 element, push the next pair onto the heap
            if index2 + 1 < len(nums2):
                heappush(priority_queue, [nums1[index1] + nums2[index2 + 1], index1, index2 + 1])

        return result  # Return the list of k smallest pairs

```

Java Solution

```java
class Solution {
    public List<List<Integer>> kSmallestPairs(int[] nums1, int[] nums2, int k) {
        // Create a priority queue to hold the arrays with a comparator to prioritize by the sum of pairs.
        PriorityQueue<int[]> queue = new PriorityQueue<>((a, b) -> a[0] - b[0]);

        // Initialize the priority queue with the first k pairs from nums1 and the first element from nums2.
        for (int i = 0; i < Math.min(nums1.length, k); ++i) {
            queue.offer(new int[] {nums1[i] + nums2[0], i, 0});
        }

        // Prepare a list to store the k smallest pairs.
        List<List<Integer>> result = new ArrayList<>();

        // Keep polling from the priority queue to find the next smallest pair
        while (!queue.isEmpty() && k > 0) {
            // Poll the smallest sum pair from the priority queue.
            int[] currentPair = queue.poll();

            // Add the new pair [nums1[index1], nums2[index2]] to the result list.
            result.add(Arrays.asList(nums1[currentPair[1]], nums2[currentPair[2]]));

            // Decrease the remaining pairs count.
            --k;

            // If there's a next element in nums2, offer the next pair from nums1 and nums2 into the priority queue.
            if (currentPair[2] + 1 < nums2.length) {
                queue.offer(new int[] {nums1[currentPair[1]] + nums2[currentPair[2] + 1], currentPair[1], currentPair[2] + 1});
            }
        }

        // Return the list of k smallest pairs.
        return result;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to find k smallest pairs with minimal sum
    vector<vector<int>> kSmallestPairs(vector<int>& nums1, vector<int>& nums2, int k) {
        // Lambda function to compare pairs based on the sum of elements they point to in nums1 and nums2
        auto comparePairs = [&nums1, &nums2](const pair<int, int>& a, const pair<int, int>& b) {
            return nums1[a.first] + nums2[a.second] > nums1[b.first] + nums2[b.second];
        };

        // Get the sizes of the input arrays
        int size1 = nums1.size();
        int size2 = nums2.size();

        // Variable to store the final pairs
        vector<vector<int>> result;

        // Priority queue to keep pairs in ascending order based on their sum
        priority_queue<pair<int, int>, vector<pair<int, int>>, decltype(comparePairs)> minHeap(comparePairs);

        // Initialize the priority queue with the first pair from each element in nums1
        for (int i = 0; i < min(k, size1); ++i) {
            minHeap.emplace(i, 0);
        }

        // Extract the k smallest pairs
        while (k-- > 0 && !minHeap.empty()) {
            auto [index1, index2] = minHeap.top();
            minHeap.pop();

            // Add the current smallest pair to the result
            result.push_back({nums1[index1], nums2[index2]});

            // If there's a next element in nums2, add the new pair to the priority queue
            if (index2 + 1 < size2) {
                minHeap.emplace(index1, index2 + 1);
            }
        }

        // Return all k smallest pairs found
        return result;
    }
};

```

Typescript Solution

```typescript
// Importing required modules
import { PriorityQueue } from 'typescript-collections'; // Placeholder import for PQ functionality

// Defining the type for a pair of indices
type IndexPair = [number, number];

// Function to compare pairs based on the sum of elements they point to in nums1 and nums2
function comparePairs(nums1: number[], nums2: number[], a: IndexPair, b: IndexPair): boolean {
  return nums1[a[0]] + nums2[a[1]] > nums1[b[0]] + nums2[b[1]];
}

// Function to find k smallest pairs with minimal sum
function kSmallestPairs(nums1: number[], nums2: number[], k: number): number[][] {
  // Get the sizes of the input arrays
  const size1 = nums1.length;
  const size2 = nums2.length;

  // Variable to store the final pairs
  const result: number[][] = [];

  // Priority queue to keep pairs in ascending order based on their sum
  const minHeap = new PriorityQueue<IndexPair>((a, b) => comparePairs(nums1, nums2, a, b));

  // Initialize the priority queue with the first pair from each element in nums1
  for (let i = 0; i < Math.min(k, size1); i++) {
    minHeap.enqueue([i, 0]);
  }

  // Extract the k smallest pairs
  while (k > 0 && !minHeap.isEmpty()) {
    const [index1, index2] = minHeap.dequeue()!;

    // Add the current smallest pair to the result
    result.push([nums1[index1], nums2[index2]]);

    // If there's a next element in nums2, add the new pair to the priority queue
    if (index2 + 1 < size2) {
      minHeap.enqueue([index1, index2 + 1]);
    }

    k--;
  }

  // Return all k smallest pairs found
  return result;
}

```

Time and Space Complexity

The given Python code implements a heap to find the `k` smallest pairs of sums from two integer arrays `nums1` and `nums2`. Here we will discuss its time complexity and space complexity.

Time Complexity

The time complexity of the algorithm is as follows:

1. **Heap Initialization**: The code creates a min-heap and initializes it with the sums of elements from `nums1` and the first element of `nums2`. Since the heap is initialized with at most `k` elements (and not more than the length of `nums1`), the complexity of this step is `O(k)` since each heap insertion is `O(log k)` and we do it at most `k` times.
2. **Heap Operations**: Then, in each iteration, the algorithm pops an element from the heap and potentially pushes a new element into the heap. Since we perform `k` iterations (bounded by `k` and the length of the output), and both `heappop` and `heappush` operations have a time complexity of `O(log k)`, the complexity for this part is `O(k log k)`.

Overall, the time complexity is `O(k) + O(k log k)`, which simplifies to `O(k log k)` because as `k` grows, the `k log k` term dominates.

Space Complexity

The space complexity of the algorithm is as follows:

1. **Heap Space**: The heap size is at most `k`, as it stores pairs of indices and their corresponding sum, giving us `O(k)`.
2. **Output List**: The list `ans` to store the answer pairs. In the worst case, it will contain `k` pairs, leading to `O(k)` space complexity.

The overall space complexity combines the heap space and the output list, but since both are `O(k)`, the total space complexity remains `O(k)`.