Documentation

The problem of finding k pairs with the smallest sums from two sorted arrays is a classic example of leveraging sorted properties for efficient computation. The challenge lies in identifying the smallest sums among all possible pairs without exhaustively generating and comparing them. Since both arrays are sorted in non-decreasing order, the smallest sums are guaranteed to include the smallest elements from the arrays, providing a foundation for an optimized approach. Instead of a brute-force method that computes sums for all n×m pairs, where n and mm are the lengths of the arrays, the solution uses a prioritized selection strategy based on a min-heap.

The essence of the solution lies in a systematic exploration of pairs using a min-heap data structure. Initially, pairs combining the smallest element from the first array with each element of the second array are pushed into the heap. The heap is structured to keep the pair with the smallest sum at the top, ensuring that we always process the smallest available pair first. This allows for efficient extraction of pairs with minimal sums while maintaining a manageable computation load.

At each step, the smallest pair is extracted from the heap, and it is added to the result list. Following this, a new pair is formed by advancing the index in the second array (nums2) for the same element in the first array (nums1). This ensures that we explore the next smallest potential pair involving the same element from nums1. The process continues until k pairs have been extracted or there are no more pairs to process in the heap.

The algorithm's efficiency is rooted in its use of the heap. Each insertion or extraction operation in the heap takes O(log k) time. Since at most k pairs are processed, the overall time complexity of the solution is O(klogk). This is a significant improvement over the naive approach, which would require O(n·m) time to compute all pairs. The use of the heap ensures that we only deal with the most relevant pairs at any given time, minimizing unecessary computations.

In terms of space, the heap stores at most k pairs, and the result list also requires O(k) space. Therefore, the overall space complexity is O(k). This makes the approach highly space-efficient, especially given the constraints where k can be significantly smaller than the total number of possible pairs. The algorithm's design ensures that it scales well with larger input sizes.

This solution effectively handles edge cases. If k exceeds the total number of pairs, the algorithm naturally terminates once all pairs have been processed, ensuring correctness. Duplicate elements in either array do not pose a problem, as all possible combinations are systematically evaluated. These considerations make the solution robust, gracefully handling both typical and extreme scenarios.

The solution ensures an optimal balance between time and space complexity by focusing on the smallest sums and efficiently managing pair exploration with a heap. This approach exemplifies the power of efficiently leveraging data structures like heaps to solve computational problems, especially when dealing with sorted inputs and prioritized outputs.