373. Find K Pairs with Smallest Sums

Overview

The solution for finding the `k` smallest pairs from two sorted arrays. Each pair consists of one element from `nums1` and one element from `nums2`, and the goal is to find `k` pairs with the smallest sums.

Problem Statement

Given two sorted integer arrays `nums1` and `nums2`, and an integer `k`, return the `k` pairs with the smallest sums. Each pair should be one element from `nums1` and one from `nums2`.

Example

```
Input: nums1 = [1, 7, 11], nums2 = [2, 4, 6], k = 3
Output: [[1, 2], [1, 4], [1, 6]]
```

Approach Summary

We use a **min-heap (priority queue)** to efficiently track and extract the smallest sum pairs. Initially, we form pairs using each element of `nums1` with the first element of `nums2`, and add them to the heap. The heap is then used to extract the current smallest pair, and each time we extract a pair, we push a new pair formed by advancing the index of `nums2`.

Key Concepts

- **Priority Queue (Min-Heap)**: Stores the pairs in increasing order of their sums, ensuring that we always get the smallest sum first.
- **Pair Tracking**: Each pair includes three values: one from `nums1`, one from `nums2`, and the index of the element in `nums2` currently paired with the element from `nums1`.

Java Implementation

```java
class Solution {

    public List<List<Integer>> kSmallestPairs(int[] nums1, int[] nums2, int k) {
```

```
        List<List<Integer>> smallestPairs = new ArrayList<>();
        PriorityQueue<int[]> pairPriorityQueue = new PriorityQueue<>(
                (pair1, pair2) -> Integer.compare(pair1[0] + pair1[1], pair2[0] + pair2[1])
        );
        for (int i = 0; i < nums1.length && i < k; i++) {
            pairPriorityQueue.offer(new int[] { nums1[i], nums2[0], 0 });
        }
        while (k-- > 0) {
            int[] currentSmallestPair = pairPriorityQueue.poll();
            smallestPairs.add(Arrays.asList(currentSmallestPair[0], currentSmallestPair[1]));
            int indexInArray2 = currentSmallestPair[2];
            if (indexInArray2 + 1 < nums2.length) {
                pairPriorityQueue.offer(new int[] { currentSmallestPair[0], nums2[indexInArray2 + 1], indexInArray2 + 1 });
            }
        }
        return smallestPairs;
    }
}
```

Code Breakdown

1. **Result List Initialization**:

```
List<List<Integer>> result = new ArrayList<>();
```

- This list will store the final `k` smallest pairs.

2. **Priority Queue (Min-Heap) Initialization**:

```
PriorityQueue<int[]> pairPriorityQueue = new PriorityQueue<>(
    (pair1, pair2) -> Integer.compare(pair1[0] + pair1[1], pair2[0] + pair2[1])
);
```

   - We use a min-heap to keep pairs sorted by the sum of the two numbers in each pair. This allows us to always extract the smallest sum pair efficiently.
   - Example: For input `nums1 = [1, 7, 11]` and `nums2 = [2, 4, 6]`, the heap starts by holding pairs like `[1, 2]`, `[7, 2]`, and `[11, 2]`.

3. **Heap Initialization with First Element of nums2**:

```
for (int i = 0; i < nums1.length && i < k; i++) {
    pairPriorityQueue.offer(new int[] { nums1[i], nums2[0], 0 });
}
```

   - For each element in `nums1` (up to `k`), we initially pair it with the first element of `nums2`.
   - Example: Heap contains `[1, 2, 0]`, `[7, 2, 0]`, `[11, 2, 0]` (third element tracks the index in `nums2`).

> The `0` at the end of each array represents that the pair consists of `nums1[i]` paired with `nums2[0]`. As you process the pairs in the priority queue, the `0` will increment (e.g., to `1`, `2`, etc.) to track the index in `nums2` so that the next pair can use the next element in `nums2`.

> This setup allows you to track which element from `nums2` is paired with `nums1[i]` in future iterations. For example, later you might replace the third element (the index `0`) with `1`, `2`, and so on, to represent pairs like `[nums1[i], nums2[1]]`, `[nums1[i], nums2[2]]`, etc.
>
> The third element is a key component for iterating over `nums2` without having to reprocess the entire `nums2` array.

4. **Processing the Heap**:

```
while (k-- > 0) {
    int[] currentSmallestPair = pairPriorityQueue.poll();
    result.add(Arrays.asList(currentSmallestPair[0], currentSmallestPair[1]));
```

   - Extract the smallest pair from the heap and add it to the result list.
   - Example: The pair `[1, 2]` is extracted first because `1 + 2 = 3` is the smallest sum.

> **poll()**: Extracts the smallest pair (based on sum) to add it to the result list.

5. **Advancing in nums2**:

```
int indexInArray2 = currentSmallestPair[2];
if (indexInArray2 + 1 < nums2.length) {
    pairPriorityQueue.offer(new int[] { currentSmallestPair[0], nums2[indexInArray2
+ 1], indexInArray2 + 1 });
}
```

- After extracting a pair, move to the next element in `nums2` for the current element in `nums1` and push the new pair into the heap.
- Example: After extracting `[1, 2]`, we now push `[1, 4]` (i.e., `nums2[1]`) into the heap.

> **offer()**: Adds a new pair to the priority queue. The priority queue ensures the pairs are **rearranged** based on the priority (in this case, the sum of the two numbers in the pair). The pair with the **smallest sum** always moves to the **head** (front of the queue). The queue doesn't maintain a strict order like an array, but it ensures the smallest element (in terms of sum) is always processed next.

Step-by-Step Example

Input

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

1. **Heap Initialization**:
   - Initial heap: `[1, 2]`, `[7, 2]`, `[11, 2]`.
2. **First Iteration**:
   - Extract the smallest pair `[1, 2]` (sum = 3).
   - Result: `[[1, 2]]`.
   - Push `[1, 4]` to the heap.
   - Heap now: `[1, 4]`, `[7, 2]`, `[11, 2]`.
3. **Second Iteration**:
   - Extract the next smallest pair `[1, 4]` (sum = 5).
   - Result: `[[1, 2], [1, 4]]`.
   - Push `[1, 6]` to the heap.
   - Heap now: `[1, 6]`, `[7, 2]`, `[11, 2]`.
4. **Third Iteration**:
   - Extract the next smallest pair `[1, 6]` (sum = 7).
   - Result: `[[1, 2], [1, 4], [1, 6]]`.
   - Heap now: `[7, 2]`, `[11, 2]`.

Output

```
[[1, 2], [1, 4], [1, 6]]
```