Find K Pairs with Smallest Sums

You are given two integer arrays **nums1** and **nums2** sorted in ascending order and an integer **k**.

Define a pair **(u,v)** which consists of one element from the first array and one element from the second array.

Find the k pairs **$(u_1,v_1),(u_2,v_2) \ldots (u_k,v_k)$** with the smallest sums.

Example 1:

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

Return: [1,2],[1,4],[1,6]

The first 3 pairs are returned from the sequence:
[1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]
```

Example 2:

```
Given nums1 = [1,1,2], nums2 = [1,2,3],  k = 2

Return: [1,1],[1,1]

The first 2 pairs are returned from the sequence:
[1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]
```

Example 3:

```
Given nums1 = [1,2], nums2 = [3],  k = 3

Return: [1,3],[2,3]

All possible pairs are returned from the sequence:
[1,3],[2,3]
```

Credits:
Special thanks to @elmirap and @StefanPochmann for adding this problem and creating all test cases.

Solution 1

Several solutions from naive to more elaborate. I found it helpful to visualize the input as an **m×n matrix** of sums, for example for nums1=[1,7,11], and nums2=[2,4,6]:

```
      2   4   6
   +------------
 1 |  3   5   7
 7 |  9  11  13
11 | 13  15  17
```

Of course the smallest pair overall is in the top left corner, the one with sum 3. We don't even need to look anywhere else. After including that pair in the output, the next-smaller pair must be the next on the right (sum=5) or the next below (sum=9). We can keep a "horizon" of possible candidates, implemented as a heap / priority-queue, and roughly speaking we'll grow from the top left corner towards the right/bottom. That's what my solution 5 does. Solution 4 is similar, not quite as efficient but a lot shorter and my favorite.

Solution 1: Brute Force (accepted in 560 ms)

Just produce all pairs, sort them by sum, and return the first k.

```
def kSmallestPairs(self, nums1, nums2, k):
    return sorted(itertools.product(nums1, nums2), key=sum)[:k]
```

Solution 2: Clean Brute Force (accepted in 532 ms)

The above produces tuples and while the judge doesn't care, it's cleaner to make them lists as requested:

```
def kSmallestPairs(self, nums1, nums2, k):
    return map(list, sorted(itertools.product(nums1, nums2), key=sum)[:k])
```

Solution 3: Less Brute Force (accepted in 296 ms)

Still going through all pairs, but only with a generator and `heapq.nsmallest`, which uses a heap of size k. So this only takes O(k) extra memory and O(mn log k) time.

```python
def kSmallestPairs(self, nums1, nums2, k):
    return map(list, heapq.nsmallest(k, itertools.product(nums1, nums2), key=sum)
)
```

Or (accepted in 368 ms):

```python
def kSmallestPairs(self, nums1, nums2, k):
    return heapq.nsmallest(k, ([u, v] for u in nums1 for v in nums2), key=sum)
```

Solution 4: Efficient (accepted in 112 ms)

The brute force solutions computed the whole matrix (see visualization above). This solution doesn't. It turns each row into a generator of triples [u+v, u, v], only computing the next when asked for one. And then merges these generators with a heap. Takes O(m + k*log(m)) time and O(m) extra space.

```python
def kSmallestPairs(self, nums1, nums2, k):
    streams = map(lambda u: ([u+v, u, v] for v in nums2), nums1)
    stream = heapq.merge(*streams)
    return [suv[1:] for suv in itertools.islice(stream, k)]
```

Solution 5: More efficient (accepted in 104 ms)

The previous solution right away considered (the first pair of) all matrix rows (see visualization above). This one doesn't. It starts off only with the very first pair at the top-left corner of the matrix, and expands from there as needed. Whenever a pair is chosen into the output result, the next pair in the row gets added to the priority queue of current options. Also, if the chosen pair is the first one in its row, then the first pair in the next row is added to the queue.

```python
def kSmallestPairs(self, nums1, nums2, k):
    queue = []
    def push(i, j):
        if i < len(nums1) and j < len(nums2):
            heapq.heappush(queue, [nums1[i] + nums2[j], i, j])
    push(0, 0)
    pairs = []
    while queue and len(pairs) < k:
        _, i, j = heapq.heappop(queue)
        pairs.append([nums1[i], nums2[j]])
        push(i, j + 1)
        if j == 0:
            push(i + 1, 0)
    return pairs
```

written by StefanPochmann original link here

Solution 2

Because both array are sorted, so we can keep track of the paired index. Therefore, we do not need to go through all combinations when k < nums1.length + num2.length. Time complexity is O(k*m) where m is the length of the shorter array.

```
public List<int[]> kSmallestPairs(int[] nums1, int[] nums2, int k) {
        List<int[]> ret = new ArrayList<int[]>();
        if (nums1.length == 0 || nums2.length == 0 || k == 0) {
            return ret;
        }

        int[] index = new int[nums1.length];
        while (k-- > 0) {
            int min_val = Integer.MAX_VALUE;
            int in = -1;
            for (int i = 0; i < nums1.length; i++) {
                if (index[i] >= nums2.length) {
                    continue;
                }
                if (nums1[i] + nums2[index[i]] < min_val) {
                    min_val = nums1[i] + nums2[index[i]];
                    in = i;
                }
            }
            if (in == -1) {
                break;
            }
            int[] temp = {nums1[in], nums2[index[in]]};
            ret.add(temp);
            index[in]++;
        }
        return ret;
    }
```

written by mylzsd original link here

Solution 3

Frist, we take the first k elements of nums1 and paired with nums2[0] as the starting pairs so that we have (0,0), (1,0), (2,0),.....(k-1,0) in the heap.
Each time after we pick the pair with min sum, we put the new pair with the second index +1. ie, pick (0,0), we put back (0,1). Therefore, the heap alway maintains at most min(k, len(nums1)) elements.

```
public class Solution {
    class Pair{
        int[] pair;
        int idx; // current index to nums2
        long sum;
        Pair(int idx, int n1, int n2){
            this.idx = idx;
            pair = new int[]{n1, n2};
            sum = (long) n1 + (long) n2;
        }
    }
    class CompPair implements Comparator<Pair> {
        public int compare(Pair p1, Pair p2){
            return Long.compare(p1.sum, p2.sum);
        }
    }
    public List<int[]> kSmallestPairs(int[] nums1, int[] nums2, int k) {
        List<int[]> ret = new ArrayList<>();
        if (nums1==null || nums2==null || nums1.length ==0 || nums2.length ==0) r
eturn ret;
        int len1 = nums1.length, len2=nums2.length;

        PriorityQueue<Pair> q = new PriorityQueue(k, new CompPair());
        for (int i=0; i<nums1.length && i<k ; i++) { // only need first k number
in nums1 to start
            q.offer( new Pair(0, nums1[i],nums2[0]) );
        }
        for (int i=1; i<=k && !q.isEmpty(); i++) { // get the first k sums
            Pair p = q.poll();
            ret.add( p.pair );
            if (p.idx < len2 -1 ) { // get to next value in nums2
                int next = p.idx+1;
                q.offer( new Pair(next, p.pair[0], nums2[next]) );
            }
        }
        return ret;
    }
}
```

written by yubad2000 original link here