# Solution: Find K Pairs with Smallest Sums

Let's solve the Find K Pairs with Smallest Sums problem using the K-Way Merge pattern.

We'll cover the following



## Statement

Given two lists and an integer $k$, find $k$ pairs of numbers with the smallest sum so that in each pair, each list contributes one number to the pair.

**Constraints:**

- Input lists should be sorted in ascending order.
- If the value of $k$ exceeds the total number of valid pairs that may be formed, return all the pairs.

## Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

### Naive approach

One way to solve this problem is by creating all possible pairs from the given lists. This creation of pairs costs $O(n_1 * n_2)$, where $n_1$ and $n_2$ are the lengths of two lists.

> **Note:** Since we're not certain whether or not the lengths of the two lists are the same, we'll use $n_1$ and $n_2$ for the first and second list, respectively.

Once we've created all pairs, sort them according to their sums, and remove the first $k$ pairs from the list. Sorting and removal of pairs costs us $O(n_1 * n_2 * \log(n_1 * n_2))$.

- Step 1: Let's take two lists as an example, $[4, 7, 9]$ and $[4, 7, 9]$. We can make $9$ pairs from these two lists, and those are the following:

$$[[4, 4], [4, 7], [4, 9], [7, 4], [7, 7], [7, 9], [9, 4], [9, 7], [9, 9]]$$

- Step 2: Now, let's sort the pairs with the order of their sums:

$$[[4, 4], [4, 7], [7, 4], [4, 9], [9, 4], [7, 7], [7, 9], [9, 7], [9, 9]]$$

- Step 3: Let's remove the first $k$ pairs from the sorted list of pairs above and return our output.

  Suppose $k$ is $5$, so the first $5$ pairs with the smallest sums will be the following:

$$[[4, 4], [4, 7], [7, 4], [4, 9], [9, 4]$$

Overall, this approach of solving the pattern costs us $O(n_1 * n_2 * \log(n_1 * n_2))$ time complexity. The space complexity is $O(n_1 * n_2)$.

### Optimized approach using k-way merge

We can use the k-way merge pattern here because it helps solve problems where we need to compute a result by comparing the elements of multiple sorted lists.

While traversing through the elements of both lists and making pairs, we use a min-heap to track the pair with the smallest sum encountered so far.

As proposed above, we use a min-heap to store three things:

- The smallest sum (at the root of the min-heap).
- The sum of each pair.
- The list indexes of each element in the pair.

Initially, we can start making pairs by adding only the first element of the second list to each element of the first list.

Let's take two lists as an example, $[2, 8, 9]$ and $[1, 3, 6]$. We can make 3 pairs in the first iteration and store them in the min-heap:

$$[(2 + 1) = 3, (8 + 1) = 9, (9 + 1) = 10]$$

Now, we start a second loop that iterates while elements are in the min-heap and while we have yet to find all $k$ smallest pairs. In each iteration, we perform three steps:

1. Pop the smallest pair from the min-heap, noting the sum of the pair and the list indexes of each element, calling the `i` and `j` indexes, respectively.
2. Add this pair to the result list.
3. Increment `j` to move forward in the second list, and compose a new pair with the same element from the first list and the next element in the second list. This step is skipped if a new pair can't be formed when the popped pair's `j` was already pointing to the end of the second list.

Supposing $k = 9$ in our example, the sequence of pairs pushed and popped in the second step is as follows:

```
1.  Pop:  (2 + 1) = 3  // 1st pair // Min-heap: [(8 + 1) = 9, (9 + 1) = 10]
2.  Push: (2 + 3) = 5              // Min-heap: [(2 + 3) = 5, (8 + 1) = 9, (9 + 1) = 10]
3.  Pop:  (2 + 3) = 5  // 2nd pair // Min-heap: [(8 + 1) = 9, (9 + 1) = 10]
4.  Push: (2 + 6) = 8              // Min-heap: [(2 + 6) = 8, (8 + 1) = 9, (9 + 1) = 10]
5.  Pop:  (2 + 6) = 8  // 3rd pair // Min-heap: [(8 + 1) = 9, (9 + 1) = 10]

<-- No pair to push as at the end of list 2 -->

6.  Pop:  (8 + 1) = 9  // 4th pair // Min-heap: [(9 + 1) = 10]
7.  Push: (8 + 3) = 11             // Min-heap: [(9 + 1) = 10, (8 + 3) = 11]
8.  Pop:  (9 + 1) = 10 // 5th pair // Min-heap: [(8 + 3) = 11]
9.  Push: (9 + 3) = 12             // Min-heap: [(8 + 3) = 11, (9 + 3) = 12]
10. Pop:  (8 + 3) = 11 // 6th pair // Min-heap: [(9 + 3) = 12]
11. Push: (8 + 6) = 14             // Min-heap: [(9 + 3) = 12, (8 + 6) = 14]
```

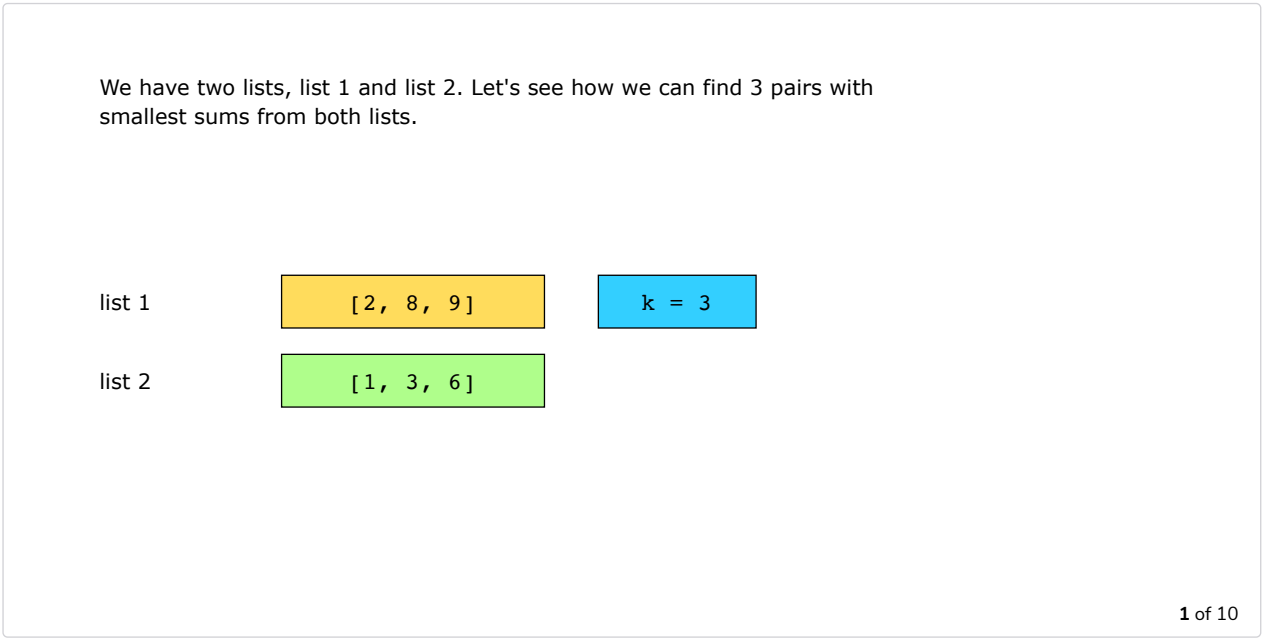
```
12. Pop:  (9 + 3) = 12 // 7th pair // Min-heap: [(8 + 6) = 14]
13. Push: (9 + 6) = 15            // Min-heap: [(8 + 6) = 14, (9 + 6) = 15]
14. Pop:  (8 + 6) = 14 // 8th pair // Min-heap: [(9 + 6) = 15]

<-- No pair to push as at the end of list 2 -->

15. Pop:  (9 + 6) = 15 // 9th
```

At all times, the smallest sum is at the root of the min-heap. Overall, we remove $k$ pairs from the min-heap.

The slides below illustrate how we would like the algorithm to run:




Let's look at the code for this solution below:

Java

```java
class Pair {
    int sum;
    int i;
    int j;

    public Pair(int sum, int i, int j) {
        this.sum = sum;
        this.i = i;
        this.j = j;
    }
}
class FindKPairs {

    public static List<List<Integer>> kSmallestPairs(int[] list1, int[] list2, int k) {
        List<List<Integer>> pairs = new ArrayList<>();
        // storing the length of lists to use it in a loop later
        int listLength = list1.length;
        // declaring a min-heap to keep track of the smallest sums
        PriorityQueue<Pair> minHeap = new PriorityQueue<>((a, b) -> a.sum - b.sum);
        // iterate over the length of list 1
        for (int i = 0; i < Math.min(k, listLength); i++) {
            // computing sum of pairs all elements of list1 with first index
            // of list2 and placing it in the min-heap
            minHeap.add(new Pair(list1[i] + list2[0], i, 0));
        }

        int counter = 1;
        // iterate over elements of min-heap and only go upto k
```

### Solution summary

Let's review the solution we've used to solve this problem:

1. We start by pairing only the first element of the second list with each element of the first list. The sum of each pair and their respective indexes from the lists, `i` and `j`, are stored on a min-heap.
2. Next, we use a second loop in which at each step, we do the following:

    - the second element is the next element in the second list.
    - We push this pair along with its sum onto the min-heap.
    - We keep a count of the steps and stop when the min-heap becomes empty or when we have found $k$ pairs.

### Time complexity

The first round of pushing pairs onto the heap, with the first element of the second list fixed, takes $O(m \log m)$ time, where $m = min(k, n_1)$ and $n_1$ is the length of the first list.

The second loop will run at most $k$ times. Assuming that each iteration involves pushing onto the min-heap, its size will stay constant, that is, $m$. Each push and pop operation therefore costs $O(\log m)$. So, the cost of the second loop is $O(k \log m)$.

Therefore, the total time complexity of this method is $O((m + k) \log m)$.

### Space complexity

The space complexity of this algorithm is $O(m)$, where $m = min(k, n_1)$.

Back
Find K Pairs with Smal...

Next
Merge K Sorted Lists

Mark as Completed