Problem 1

Given \( k \) lists of integers, each of them sorted and having length \( n \), design an efficient algorithm to merge all the \( k \) lists into a single sorted list. What is the running time of your algorithm?

**Solution:**

**Solution using heaps:**
The idea is to have a pointer that points initially to the first (smallest) element of each array, and in any iteration to store the values the pointers point to in a min-heap.

**Algorithm:**
1. Initialise \( k \) pointers \((p_1, \ldots, p_k)\) to the start of the \( k \) sorted lists and make a min-heap of size \( k \) with the corresponding values. Initialise a new list \( L \) to empty list.
2. Repeat till all pointers point to null.
   (a) Pop the min element from the heap and add it to \( L \), and if that element was pointed to by \( p_i \), move \( p_i \) to the next element in list \( i \) if possible.
   (b) Add the new element pointed to by \( p_i \) (if it exists) to the min heap and run heapify.

The running time of this algorithm, is \( O(nk \log k) \). (In each step, at least one pointer is advanced, so there are at most \( nk \) steps. Also, in each step there is a heapify operation on a heap of size \( k \), which takes \( O(\log k) \))

To see correctness, the root of the min heap is always the smallest element, and the heap has the smallest of the unseen elements in each list, and therefore the root has to be the smallest among all the unseen elements, which we add to \( L \). Therefore, each element added to \( L \) is larger than all the elements in \( L \) and smaller than the elements that are unseen.

**Solution using Divide and Conquer:**
The idea is to split the \( k \) lists to 2 sets of \( k/2 \) lists, and recursively call the function on the 2 sets and merge the results. If the function is called with 2 lists, merge the 2 lists in linear time and return the merged list.

**Function(set of lists):**
1. If the set has just 2 lists, merge them(using 2 pointers) and return the merged list.
2. If not, split the input set of lists into 2 sets of equal size, and call this function recursively on each part.
3. Merge the 2 lists returned by the 2 recursive calls(using 2 pointers), and return the merged list.
To analyse the running time, we write the following recursion.

\[ T(2) = O(n) \]
\[ T(\ell) = 2 * T(\ell/2) + O(n\ell) = 2^i * T(\ell/2^i) + i * O(n\ell) \]

Setting \( \ell \) to \( k \) and setting \( i \) to \( \log k \),

\[ T(k) = k * O(n) + nk * O(\log k) = O(nk \log k) \]

**Problem 2**

Design a simple algorithm, using a max-heap, that finds the \( k^{th} \) smallest integer from a given list of \( n \) integers. The algorithm should run in time \( O(n \log k) \).

**Solution:**

The idea is to maintain a heap of size \( k \) which contains the smallest \( k \) elements seen so far.

**Algorithm:**

1. Take the first \( k \) elements of the list and create a heap.

2. For each subsequent element in the list, compare it to the root of the heap. If the element is smaller than the root, replace the root with the element and heapify.

3. Once all elements of the list have been considered, return the root of the max heap

**Correctness:** At any point, the heap contains the \( k \) smallest elements. Proof: We prove this by induction. This is true initially when we have seen only the first \( k \) elements. The root of the heap contains the largest of these elements. Assume, after some iteration \( t \), we have the \( k \) smallest numbers seen so far in the heap. Consider iteration \( t + 1 \). If the list element is larger than the root, it is larger than every element in the heap, and therefore we ignore it as it is not the \( k^{th} \) smallest so far. If the list element is smaller than the root, then we replace the root with the list element as the root is not the \( k^{th} \) smallest element seen so far as there are \( k - 1 \) elements in the heap and the new element seen that are smaller than it. Therefore, at any point, the element we choose to ignore cannot be the answer we require.

At the end of the loop, we have a max heap that contains exactly \( k \) elements, which form the smallest \( k \) numbers in the list, and the largest of which is the root. Therefore, we return the root.

**Run Time:** Initially we take \( O(k) \) time to create the heap, and in each subsequent iteration, we take at most \( O(\log k) \) for the heapify operation. Therefore, the total time is \( O(k + (n - k) \log k) \), or \( O(n \log k) \)
Problem 3

1. Show that the height of any tree $T$ is at most $\log |T|$. 

Solution: We show this by induction. Consider the base case, when the tree has just 1 element. The height of the tree is 0. Now assume that for $|T| \leq k$, the claim is true. Consider the step where we merge 2 trees, $T_1$ and $T_2$, of size at most $k$. By inductive hypothesis, the heights of the trees are bounded by $\log |T_1|$ and $\log |T_2|$. Assume without loss of generality that $|T_1| \geq |T_2|$ The height of the merged tree is $\max(h(T_1), h(T_2) + 1)$. Clearly, $h(T_1) \leq \log |T_1| \leq \log(|T_1| + |T_2|)$, therefore the claim is true for the merged tree if the former term is larger. If the latter term is larger, height of the merged tree $\leq 1 + \log |T_2| = \log(2|T_2|) \leq \log(|T_1| + |T_2|)$, proving the claim for the merged tree.

2. Give an upper bound on the time required to implement either of the two operations above. If $G(V,E)$ has $n$ vertices and $m$ edges, what is the total running time to find a minimum spanning tree?

Solution: If each node has its parent information, checking if 2 nodes belong to the same component is simply checking if they have the same root. We use the result from the previous part to show that this is at most $\log n$. Merging two trees is a constant time operation. The initial sorting of the edges takes $m \log n$. Therefore the total time taken is at most $m \log n$, as for each edge we do 2 checks and at most 1 merge.

Although it doesn’t really help with this case(because of the sorting step), the second part can be made more efficient by making the parent of every node point to be updated with the root node of the tree every time it is accessed. This is called flattening or path compression (read up on disjoint set data structure).

}\n\[</text>\]