Please implement following two tasks in Parallel using Java 8.

Task1: Recursive Serial Summation

Consider an array of integers for which you need to find the sum of all of the integers in the array. This can be done recursively by computing the middle index of the array and making recursive calls on the left and right subarrays. The left subarray works on array indices from first to mid, and the right subarray works on array indices from mid+1 to last. The recursive calls return the sum for their respective subarrays. Simply add the left and right subarray summations together and return the resulting sum. The base case is when first and last are the same index, in which case the integer at that index is returned as the sum of the subarray.

Recursive Parallel Summation

With Java 8, it is straightforward to convert the above algorithm into one that automatically uses multiple processors if available. However, we must identify the part of the algorithm that can safely be run in parallel. For the parallel sum, the order that the subarray summation results are completed does not matter. That is, it does not matter if left completes before right or vice versa, as long as they both complete before the summation of left and right is returned from the recursive procedure. When we allow left and right to work on their subtasks simultaneously (if processors are available), we refer to this as a **fork**. When we wait for the subtasks to complete before continuing on, we refer to this as **join**. We now parallelize the above summation code to include fork and join at the appropriate spots in the algorithm.

Task2: Radix Sort in Parallel

Sorting

You are familiar with sorting algorithms that compare elements such as selection sort and merge sort. The order notation for an efficient sorting algorithm that compares elements is $O(nlogn)$. There are sorting algorithms that do not directly compare elements to each other. One of these is radix sort with an order notation of $O(n)$, although it can have a large coefficient, as we will see.

Radix Sort 1

Suppose you have numerous items stored in an array that you wish to sort using radix sort. The usual way that radix sort is implemented is to start by examining the last character of the sort key. For simplicity, let's assume that the sort keys contain only lower case letters and that we want to sort in ascending order. Based on the ASCII value of the last character of the sort key, the item with that sort key is placed on a particular queue. Thus, all of the items with the same last character in their sort key will be placed on the same queue. The items are then removed from the queues starting with the queue for 'a', then 'b', and so forth. Each queue is completely emptied before moving on to the next queue. The items are placed back on the array in the order that they were removed from the queues. Of course, the items are not at all sorted at this point.

Next, the entire process outlined above is repeated for the second to last character, then the third to last, and so forth. The items will be sorted after the first character in the sort key has been processed. Thus, even though the algorithm is $O(n)$, there is a coefficient equal to the number of characters in the sort key, and **all** of the characters

will be examined, even if the first character of the sort keys are all different. Further, sort keys are rarely all the same length. If an attempt is made to access a sort key character that is out of range, one can return a special value that will place that sort key in the very first queue (I typically reserve the very first queue for ASCII values that do not correspond to letters or digits). If two sort keys are identical except for differing lengths, the shorter sort key will be placed earlier in the sorted array.

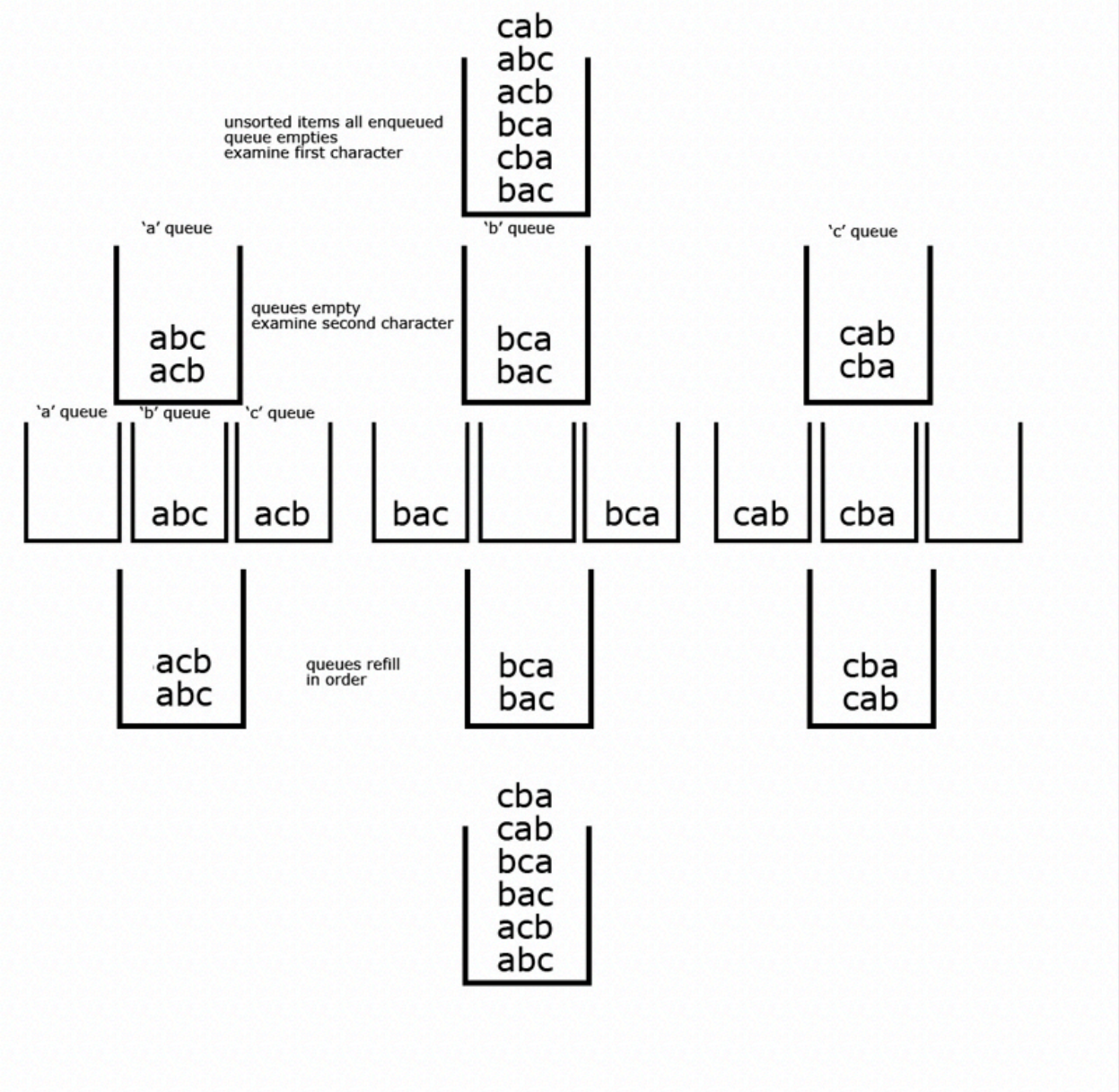
Radix Sort 2

It is possible to implement radix sort starting with the first character if recursion is employed. Create a single queue and place all of the items to be sorted on it. Call a procedure that accepts a queue and the index of the current character to be examined as parameters. The initial call to this procedure will set the current character to 1 for the first character in the sort key.

The procedure removes items from the incoming queue and places them on to a new set of initially empty queues by examining the sort key character specified by the current character parameter. For each of these new queues that contains more than one item (or if a user specified maximum character is reached), a recursive call is made that increases the current character by one. In this way, after recursion has completed on one of the new queues, the items in that queue will be sorted. After any necessary recursion has completed, the items are collected from the queues, in order, and placed back on the queue that was passed to the procedure.

This version may appear to be inefficient due to the large number of queues created for each recursion call. However, because the branching factor is so large, very few recursive levels are actually

required as most of the queues at the second or third recursive level will be empty or only contain one item (unless there are a lot of nearly duplicate keys). This version is appealing as it starts with the first character, and, in most cases, does not process that many characters in the sort key.



Parallelize

With Java 8, it is straightforward to convert the above algorithm into one that automatically uses multiple processors if available. However, we must identify the part of the algorithm that can safely be run in parallel. For parallel radix sort, the order in which the set of queues are sorted does not matter. That is, the queue for 'z' can be sorted before the queue for 'a'. However, emptying the set of queues and placing their contents back on the incoming queue must be done in order. When we allow each queue to work on their subtask simultaneously (if processors are available), we refer to this as a **fork**. When we wait for the subtasks to complete before continuing on, we refer to this as **join**. Thus, only after all of the queues have been forked should you start to join them. We can fork when making a recursive call on a queue, and we can join when ready to empty a queue.