EXERCISE 1: MSD String Sort

Modify the MSD radix sort implementation to use an array of queues for key-indexed counting instead of the count and aux arrays.

Assume that we would like to sort the strings in the subarray \(a[lo ... hi]\) based on the \(d^{th}\) character, where \(a[]\) is an array of strings of the same length \(w\). Proceed as follows:

- Create an array of queues called \(bins[]\) of size \(R=256\).
- Add every string from \(a[lo]\) to \(a[hi]\) to a queue in \(bins[]\), such that the queue at \(bins[r]\) holds all the strings whose \(d^{th}\) character is \(r\).
- Copy the strings from \(bins[]\) back to \(a[lo ... hi]\) according to their order in \(bins[]\) (e.g. strings in the queue at \(bins[10]\) are copied to \(a[]\) before strings in the queue at \(bins[11]\)).

Implement your solution by modifying the given code template. A simplified version of the MSD radix sort algorithm seen in lecture is provided below for your reference.

```java
public static void sort(String[] a) {
    if (a.length == 0) return;

    int w = a[0].length();
    String[] aux = new String[a.length];
    sort(a, aux, w, 0, a.length - 1, 0);
}

private static void sort(String[] a, String[] aux, int w, int lo, int hi, int d) {
    if (hi <= lo || d == w) return;

    int[] count = new int[R+1];
    for (int i = lo; i <= hi; i++)
        count[a[i].charAt(d) + 1]++;

    for (int r = 0; r < R; r++)
        count[r+1] += count[r];

    for (int i = lo; i <= hi; i++)
        aux[count[a[i].charAt(d)]++] = a[i];

    for (int i = lo; i <= hi; i++)
        a[i] = aux[i - lo];

    for (int r = 0; r < R; r++) {
        int start = lo + count[r];
        int end = lo + count[r+1] - 1;
        sort(a, aux, w, start, end, d+1);
    }
}
```
class QueuesMSD {
    private static final int R = 256;

    public static void sort(String[] a) {
        int size = a.length;
        if (size == 0) return;

        // all strings are assumed to be of the same length
        int w = a[0].length();
        sort(a, 0, size-1, w, 0);
    }

    // Sort from a[lo] to a[hi], starting at the dth character.
    private static void sort(String[] a, int lo, int hi, int w, int d) {
        if (hi <= lo || d >= w) return;

        // The queue at bins[r] holds all the strings whose dth character is r.
        Queue<String>[] bins = (Queue<String>[]) new Queue[R];
        for (int r = 0; r < R; r++)
            bins[r] = new Queue<String>();

        // TODO: Add each string in the range a[lo ... hi]
        // to its correct bin based on the dth character.

        // TODO: Use the bins array to distribute the strings
        // back to a[lo ... hi] sorted based on the dth character.

        // TODO: Recursively apply MSD to sort each bin.
        int from = lo;
        for (int r = 0; r < R; r++) {
            int to = ________________________________;
            sort(a, from, to, w, d+1);
            from += _________________________________;
        }
    }
}
EXERCISE 2: DNA Fragment Collection

Design a data type to store a collection of gene fragments over the DNA alphabet {A, C, T, G}, according to the following API:

```java
public class FragmentCollection {
    public FragmentCollection() create an empty collection of DNA fragments
    public void add(String fragment) add the DNA fragment to the collection
    public int prefixCount(String p) number of DNA fragments that start with prefix p
}
```

Here is an example:

```java
FragmentCollection fc = new FragmentCollection();
fc.add("AC");
fc.add("TACG");
fc.add("TCGAA");
fc.add("CGA");
fc.add("AGCT");
fc.add("TCGG");
fc.add("TCGG"); // added twice, will be counted twice
fc.prefixCount("""); // returns 7 (number of adds)
fc.prefixCount("T"); // returns 4 (TACG, TCGAA, TCGG, TCGG)
fc.prefixCount("TC"); // returns 3 (TCGAA, TCGG, TCGG)
fc.prefixCount("G"); // returns 0
```

Performance requirements:

- Given \( N \) fragments to be added, \( \text{add(fragment)} \) must run in time that is proportional to the length \( W \) of the fragment and independent of \( N \).
- If there are \( M \) fragments that match a given prefix \( p \), \( \text{prefixCount(p)} \) must run in time that is proportional to the length \( W \) of the prefix \( p \) and independent of \( N \) and \( M \).

EXERCISE 3: Suffix Arrays

The following code creates an array of suffixes for a string \( s \) of length \( n \), where \( \text{suffixes[i]} \) is the substring of \( s \) starting at index \( i \) and finishing at index \( n-1 \). How much memory does this code use as a function of the number of \( n \)?

```java
String[] suffixes = new String[n];
for (int i = 0; i < n; i++)
                suffixes[i] = s.substring(i, n);
```

Describe a more efficient way to store the suffixes.