Interview Question: Anagram Detection

This problem frequently comes up as a job interview question because there are many different ways to solve it and the "best" options each involve making appropriate use of data structures and standard algorithms.

Problem Statement:

Two strings are said to be ***anagrams*** of one another if you can turn the first string into the second by rearranging its letters. For example, "table" and "bleat" are anagrams, as are "tear" and "rate." Your job is to write a function that takes in two strings as input and determines whether they're anagrams of one another.

Possible Follow-Up Questions:

- Do we need to worry about case-sensitivity? Or, are the inputs guaranteed to be in lower-case? **Your algorithm can be case-sensitive?**
- Do the strings just consist of letters, or can they have other characters in them? **Assume they're just letters**.
- Are the strings necessarily the same length? **No, they can be different lengths**.
- Is a string an anagram of itself? **Yes.**
- Are all the letters from the English alphabet? **Yes**.

What to Watch For

- Make sure that their solution works on strings of different lengths.
- Make sure that their solution works on a string and itself.
- Make sure that their solution works on the empty string.
- Make sure that their solution handles repetition correctly. For example, "are" and "area" are not anagrams, and "ban" and "banana" aren't anagrams.
- Some programming languages (C, Java, etc.) require you to use a special function like strcmp or String.equals to check if two strings are equals. Other languages (C++, JavaScript, etc.) allow you to test equality using ==. Make sure the solution you see adheres to the proper conventions.

Possible Solution Routes

Option 1: Brute force

One option is to list off all permutations of the first string and see if any of them are equal to the second string. You can do this recursively or by using a library function that lists permutations. This is considered a "poor" solution because it misses significantly easier approaches and takes at least $n!$ time in the worst case.

Sample solution:

```
private boolean areAnagrams(String first, String second) {
      return areAnagramsRec("", first, second);
}
private boolean areAnagramsRec(String soFar, String remaining,
                               String target) {
      if (remaining.length() == 0) {
            return soFar.equals(target);
      }
      for (int i = 0; i < remaining.length(); i++) {
            String whatsLeft = remaining.substring(0, i) +
                               remaining.substring(i+1);
            if (areAnagramsRec(soFar + remaining.charAt(i),
                               whatsLeft, target)) return true;
      }
      return false;
}
```

Option 2: Sorting

Two strings are anagrams of one another if they're equal when their letters are sorted. This means that you can test for whether two strings are anagrams of one another by sorting the characters in each string and testing whether the sorted strings are equal. There are lots of different ways to sort an array of strings, some of which end up looking more like the approaches outlined later on.

This type of solution is probably a "good" solution. With a standard sorting algorithm like quicksort or heapsort, it runs in time $O(n \log n)$. Using counting sort, this will run in time $O(n)$ (though note that the counting sort solution ends up looking a lot more like the histogram approach we'll talk about later on).

Sample solution:

```
private boolean areAnagrams(String first, String second) {
      char[] one = Arrays.sort(first.toCharArray());
      char[] two = Arrays.sort(second.toCharArray());
      return Arrays.equals(one, two);
}
```

Option 3: Counting Characters

Another approach would be to count up how many times each character appears in each string and confirm that each string has the same number of each character as the other. This approach will work, but is slower in practice than the histogram-based approach outlined later on. If you see this, watch for an easy edge case: if you don't check that the string lengths are the same, it's easy to accidentally return true because every character *present in the first string* appears with the same frequency in the second string, but not the other way around.

The time complexity of this approach depends on the particular implementation. Usually, they all use O(1) space.

Sample solution 1, runtime is $O(n^2)$ in the case where all characters are the same:

```
private boolean areAnagrams(String first, String second) {
      if (first.length() != second.length()) return false;

      for (int i = 0; i < first.length(); i++) {
            char currCh = first.charAt(i);
            if (numCopiesOf(currCh, first) != numCopiesOf(currCh, second)) {
                  return false;
            }
      }

      return true;
}

private int numCopiesOf(char ch, String str) {
      int result = 0;
      for (int i = 0; i < str.length(); i++) {
            if (str.charAt(i) == ch) result++;
      }
      return result;
}
```

Sample solution 2, runtime is $O(n)$ but with a high constant factor:

```
private boolean areAnagrams(String first, String second) {
      for (char ch = 'a'; ch <= 'z'; ch++) {
            if (numCopiesOf(ch, first) != numCopiesOf(ch, second)) {
                  return false;
            }
      }
      return true;
}

private int numCopiesOf(char ch, String str) {
      int result = 0;
      for (int i = 0; i < str.length(); i++) {
            if (str.charAt(i) == ch) result++;
      }
      return result;
}
```

Option 4: Histogramming

This final approach works by building a frequency histogram of the characters in each string and checking whether those histograms are the same. There are lots of variations on this theme: you can build the histogram as an array or as a hash table, you can build histograms for each string and compare them, or build a histogram for one and then destructively modify it for the second, etc. You can even think of counting sort as belonging to this family.

These approaches typically use O(n) time and O(1) space, making them among the fastest approaches to solving this problem.

Sample solution:

```
private boolean areAnagrams(String first, String second) {
      if (first.length() != second.length()) return false;

      int[] frequencies = new int[26];
      for (int i = 0; i < first.length(); i++) {
            frequencies[first.charAt(i) - 'a']++;
      }
      for (int i = 0; i < second.length(); i++) {
            if (frequencies[second.charAt(i)] == 0) return false;
            frequencies[second.charAt(i)]--;
      }
      return true;
}
```