Question 1.4 from CTCI: Anagrams

July 25, 2017

Question

Write a method to decide if two strings are anagrams or not. An anagram is a string that is formed by rearranging the letters of another string. For example, the string *iceman* is an anagram of *cinema*. The string *praise*, however, is not an anagram of the string *pairs*.

Explanation and Algorithm

There are a few different ways to solve this problem:

A simple way to solve the problem would be to use a sort method to sort the input strings and check if the sorted strings are equal. If the sorted strings are equal, then return true. Otherwise, return false.

The second way is to use two integer arrays to keep track of the number of occurrences for each character in the input strings. First, the length of both strings must be checked since a string cannot be an anagram of another string if both are not equal in length. Then, noting this length, two integer arrays must be created to keep track of the number of occurrences of each character in the input strings. One integer array will track this information for the first input string, the other integer array for the second input string. After this, both integer arrays should be looped through and the counts for each character checked. If the counts for a character are not equal, then the strings cannot be anagrams. Otherwise, the strings are anagrams.

The third way only uses a single integer array to keep track of the number of occurrences for each character in the input strings. The length of both strings must be checked. If the lengths of the input strings are not equal, the strings cannot be anagrams. A single integer array should be created to keep track of the character counts. First, the number of characters in the first input string should be counted and tracked in the array. Additionally, for every new character encountered, a variable for the number of unique characters in the string should be updated. Then, for every character in the second input string, if the count for that character in the array is zero, then the strings are not anagrams. Otherwise,

the count for that character in the array must be decremented. If the new count is zero, then that character has been fully processed and we increment a variable that tracks the number of completed characters in the second string. If the number of completed characters in the second string is equal to the number of unique characters in the first string, then it must be checked that the second string was processed completely. If these conditions are met, then the strings are anagrams. Otherwise, the strings are not anagrams.

Hints

1. Consider the definition of an anagram. What is the unique property of anagrams?
2. How might this property (equal counts of characters in both strings) be tracked? What data structure could you use?
3. Try using an array to keep track of character counts in each string. How would you check that the character counts are equal?

Code

```
/*Solution 1 (CTCI) */

public static boolean anagram(String s, String t){

   return sort(s) == sort(t);
}
```

```
/* Solution 2 */

public static boolean checkAnagram(String s1, String s2){

   if(s1.length() != s2.length()){

      return false;

   }

   int length = s1.length();
   int val1, val2;
   int i;

   int[] s1chars = new int[256];
   int[] s2chars = new int[256];
```

```
   for(i = 0; i < length; i++){

      val1 = s1.charAt(i);
      val2 = s2.charAt(i);

      s1chars[val1]++;
      s2chars[val2]++;

   }

   for(i = 0; i < 256; i++){

      if(s1chars[i] != s2chars[i]){
         return false;
      }

   }

   return true;

}
```

```
/* Solution 3 (CTCI)*/

public static boolean anagram(String s, String t){

   if(s.length() != t.length()) return false;
   int[] letters = new int[256];
   int num_unique_chars = 0;
   int num_completed_t = 0;
   char[] s_array = s.toCharArray();

   for(char c: s_array) { // count number of each char in s.
      if(letters[c] == 0) ++num_unique_chars;
      ++letters[c];
   }
   for(int i = 0; i < t.length; ++i){
      int c = (int) t.charAt(i);
      if(letters[c] == 0){ // Found more of char c in t than in s.
         return false;
      }
      --letters[c];
      if(letters[c] == 0){
         ++num_completed_t;
         if(num_completed_t == num_unique_chars){
            // It's a match if t has been processed completely.
            return i == t.length() - 1;
```

```
        }
      }

    }

    return false;

}
```

Big O analysis

1. Solution 1: The running time is dependent on the $sort()$ method used.
2. Solution 2: The running time is $O(n)$.
3. Solution 3: The running time is $O(n)$.

Interviewer Considerations

Here are a few things to ask yourself as interviewer when judging the performance of your interviewee:

- Did the interviewee ask questions to clarify ambiguous portions of the problem statement? Things like input, output, data types, return type etc. Did they run into issues later because they did not have the foresight to ask?
- Were they able to classify and generalize the type of problem they were looking at? Were they able to do the same with potential solution algorithms? Did they compare the pros and cons of potential solutions *before* coding?
- Were they able to code a fully working solution with no logic errors and minimal syntax errors? If applicable, did they code the problem in a way that showed mastery of their chosen coding language including sensible library calls, programming constructs and coding conventions. Is the code easily readable?
- Did they run through their solutions and code with proper examples? Did they cover all corner cases?
- Did they do Big O analysis? Was it accurate? Did they consider both time and space? Did they distinguish between the Big O of multiple solutions if applicable?
- Through out the interview, did they clearly communicate their thought process through out all of the above? Did they observe, at least, basic social norms?

Interviewee rating

The basis of effective practicing is honest, constructive feedback. Please rate your interviewee honestly based on the following criteria. The ratings are meant to give an objective guideline to distinguish between the performance of interviewee's for a particular question, a list of things to look out for as an interviewer and a springboard for constructive criticism. It's highly encouraged to briefly talk about the mock interview afterward to highlight strengths and weaknesses so everyone what they can rely on and what ought to be improved. Furthermore, many companies use a similar system across interviews to decide who gets to advance to the next stages of interviews and to decide who gets the job. Being aware and becoming comfortable with this sort of performance analysis will also help you analyze your own performance in real interviews and improve.

1 star- Poor performance.

- Did not ask many or any clarifying questions and committed errors that could have been avoided by asking.
- Was not able to correctly classify the type of problem or the category of algorithms that might solve it.
- Was not able to get the solution despite getting many or all hints.
- Wrote no code or conceptually and syntactically flawed code.
- Did not work through any examples.
- Did not communicate in any way their thought process in any meaningful way or didn't talk at all.
- No or completely wrong Big O analysis.
- No insight into thought process through out.

2 stars - Below average performance.

- Asked no or non pertinent clarifying questions about the problem.
- Was not able to classify the question or algorithm or show any insights into how to generalize the problem or was able to do so in only the most superficial sense.
- Was able to get an almost working solution or just brute force after being given many hints.
- Wrote some code that worked only in minimal cases and/or was riddled with flaws.
- Worked through no examples or only the most straight forward type not considering corner cases.

- Attempted Big O analysis, but over simplified it or was wrong in more complex cases.
- Minimal insight into thought process.

3 stars - Average performance.

- Asked a proper amount of clarifying questions.
- Was able to recognize the class of problem the question came from and possible solutions.
- Was able to get at least one working solution with a few hints.
- Code got to working or near working state after a few revisions.
- Ran through code with an example at least once.
- Proper big O analysis for simple, generally known cases.
- Adequate thought process displayed for the question.

4 stars - Good performance.

- Asked important pertinent questions for the problem.
- Was able to code or conceptualize multiple working solutions with minimal hints and indicated thought process.
- Worked competently through examples including all or most corner cases.
- Ran through code with all important examples or proofs of working.
- Proper big O analysis for all, but the most complex algorithms.
- Displayed thought in a way that the average computer scientist would understand.

5 stars - Stellar performance.

- Asked questions to clarify key aspects of the questions if applicable.
- Was able to quickly and correctly identify the sort of problem, the approaches that may work and communicated them and analyze the quality of potential approaches before coding.
- Got the full solution(s) to the problem with no or minimal hints.
- Wrote fully working code with minimal syntax errors, no major conceptual errors and used proper coding conventions such as modularity and readability.
- Worked through pertinent examples which included both general cases and important corner cases.

- Communicated their thought process through out the entire process with enough clarity a layman could understand the logic (with exception of special knowledge, of course).
- In depth Big O analysis for solutions and proper comparisons between solutions.