Question 10.2 from CTCI: Grouped Anagrams

Benjamin De Brasi

September 25, 2017

Question

Group Anagrams: Write a method to sort an array of strings so that all the anagrams are next to each other.

Explanation and Algorithm

Hints

1. How do you check if two words are anagrams of each other? Think about what the definition of "anagram" is. Explain it in your own words.
2. Two words are anagrams if they contain the same characters but in different orders. How can you put characters in order?
3. Can you leverage a standard sorting algorithm?
4. Do you even need to truly "sort"? Or is just reorganizing the list sufficient?

Code

```
/*Answer 1 */

boolean containsTree(TreeNode t1, TreeNode t2) {

   StringBuilder string1 new StringBuilder();
   StringBuilder string2 = new StringBuilder();

   getOrderString(t1, string1);
   getOrderString(t2, string2);
```

```
   return string1.indexOf(string2.toString()) != -1;
}

void getOrderString(TreeNode node, StringBuilder sb) {

   if (node == null) {
      sb.append("X"); II Add null indicator
      return;
   }

   sb.append(node.data + " "); // Add root
   getOrderString(node.left, sb); // Add left
   getOrderString(node.right, sb); // Add right
}
```

```
/* Answer 2 */

boolean containsTree(TreeNode t1, TreeNode t2) {
   if (t 2 == null) return true; // The empty tree is always a subtree
      return subTree(t1, t2);
}

   boolean subTree(TreeNode r1, TreeNode r2) {
   if (r1 == null) {
      return false; // big tree empty & subtree still not found.
   } else if (r1.data == r2.data && matchTree(r1, r2)) {
      return true;
   }
   return subTree(r1.left, r2) I I subTree(r1.right, r2);
}

boolean matchTree(TreeNode r1, TreeNode r2) {
   if (r1 == null && r2 == nUll) {
      return true; II nothing left in the subtree
   } else if (r1 == null I I r2 == nUll) {
      return false; II exactly tree is empty, therefore trees don't match
   } else if (r1.data != r2.data) {
      return false; II data doesn ' t match
   } else {
      return matchTree(r1.left, r2.left) && matchTree(r1.right,
          r2.right);
      }
   }
}
```

Big O analysis

When might the simple solution be better, and when might the alternative approach be better? This is a great conversation to have with your interviewer. Here are a few thoughts on that matter:

1. The simple solution takes $O(n + m)$ memory. The alternative solution takes $O(log(n) + log(m))$ memory. Remember: memory usage can be a very big deal when it comes to scalability.

2. The simple solution is $O(n + m)$ time and the alternative solution has a worst case time of $O(nm)$. However, the worst case time can be deceiving; we need to look deeper than that.

3. A slightly tighter bound on the runtime, as explained earlier, is $O(n+km)$, where k is the number of occurrences of T2's root in T1. Let's suppose the node data for T1 and T2 were random numbers picked between a and p. The value of k would be approximately $\frac{n}{p}$. Why? Because each of n nodes in T1 has a $\frac{1}{p}$ chance of equaling the root, so approximately $\frac{n}{p}$ nodes in n should equal T2 . root. So, let's say $p = 1000$, $n = 1000000$ and $m = 100$. We would do somewhere around 1,100,000 node checks $(1,100,000 = 1000000 + \frac{100*1000000}{1000})$.

4. More complex mathematics and assumptions could get us an even tighter bound. We assumed in 3 above tha t if we call matchTree, we would end up traversing all m nodes of T2 . It 's far more likely, though, that we will find a difference very early on in the tree and will then exit early.

In summary, the alternative approach is certainly more optimal in terms of space and is likely more optimal in terms of time as well. It all depends on what assumptions you make and whether you prioritize reducing the average case run-time at the expense of the worst case run-time This is an excellent point to make to your interviewer.

Sources

Question, answer and other material taken from Cracking the Coding Interview 6th edition by Gayle Laakmann McDowell.