1

Solutions to Arrays and Strings

1.1 Is Unique: Implement an algorithm to determine if a string has all unique characters. What if you cannot use additional data structures?

pg 90

SOLUTION

You should first ask your interviewer if the string is an ASCII string or a Unicode string. Asking this question will show an eye for detail and a solid foundation in computer science. We'll assume for simplicity the character set is ASCII. If this assumption is not valid, we would need to increase the storage size.

One solution is to create an array of boolean values, where the flag at index `i` indicates whether character `i` in the alphabet is contained in the string. The second time you see this character you can immediately return `false`.

We can also immediately return `false` if the string length exceeds the number of unique characters in the alphabet. After all, you can't form a string of 280 unique characters out of a 128-character alphabet.

> It's also okay to assume 256 characters. This would be the case in extended ASCII. You should clarify your assumptions with your interviewer.

The code below implements this algorithm.

```
1  boolean isUniqueChars(String str) {
2    if (str.length() > 128) return false;
3
4    boolean[] char_set = new boolean[128];
5    for (int i = 0; i < str.length(); i++) {
6      int val = str.charAt(i);
7      if (char_set[val]) { // Already found this char in string
8        return false;
9      }
10     char_set[val] = true;
11   }
12   return true;
13 }
```

The time complexity for this code is `O(n)`, where n is the length of the string. The space complexity is `O(1)`. (You could also argue the time complexity is `O(1)`, since the for loop will never iterate through more than 128 characters.) If you didn't want to assume the character set is fixed, you could express the complexity as `O(c)` space and `O(min(c, n))` or `O(c)` time, where `c` is the size of the character set.

We can reduce our space usage by a factor of eight by using a bit vector. We will assume, in the below code, that the string only uses the lowercase letters a through z. This will allow us to use just a single `int`.

```
1  boolean isUniqueChars(String str) {
2    int checker = 0;
3    for (int i = 0; i < str.length(); i++) {
4      int val = str.charAt(i) - 'a';
5      if ((checker & (1 << val)) > 0) {
6        return false;
7      }
8      checker |= (1 << val);
9    }
10   return true;
11 }
```

If we can't use additional data structures, we can do the following:

1. Compare every character of the string to every other character of the string. This will take $O(n^2)$ time and $O(1)$ space.
2. If we are allowed to modify the input string, we could sort the string in $O(n \log(n))$ time and then linearly check the string for neighboring characters that are identical. Careful, though: many sorting algorithms take up extra space.

These solutions are not as optimal in some respects, but might be better depending on the constraints of the problem.

1.2 Check Permutation: Given two strings, write a method to decide if one is a permutation of the other.

pg 90

SOLUTION

Like in many questions, we should confirm some details with our interviewer. We should understand if the permutation comparison is case sensitive. That is: is God a permutation of dog? Additionally, we should ask if whitespace is significant. We will assume for this problem that the comparison is case sensitive and whitespace is significant. So, "god " is different from "dog".

Observe first that strings of different lengths cannot be permutations of each other. There are two easy ways to solve this problem, both of which use this optimization.

Solution #1: Sort the strings.

If two strings are permutations, then we know they have the same characters, but in different orders. Therefore, sorting the strings will put the characters from two permutations in the same order. We just need to compare the sorted versions of the strings.

```
1  String sort(String s) {
2    char[] content = s.toCharArray();
3    java.util.Arrays.sort(content);
4    return new String(content);
5  }
6
7  boolean permutation(String s, String t) {
8    if (s.length() != t.length()) {
9      return false;
10   }
```