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

Hints: #44, #117, #132

pg 192

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.

Hints: #1, #84, #122, #131

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    )
```

```
11    return sort(s).equals(sort(t));
12 )
```

Though this algorithm is not as optimal in some senses, it may be preferable in one sense: It's clean, simple and easy to understand. In a practical sense, this may very well be a superior way to implement the problem.

However, if efficiency is very important, we can implement it a different way.

Solution #2: Check if the two strings have identical character counts.

We can also use the definition of a permutation—two words with the same character counts—to implement this algorithm. We simply iterate through this code, counting how many times each character appears. Then, afterwards, we compare the two arrays.

```
1  boolean permutation(String s, String t) {
2     if (s.length() != t.length()) {
3        return false;
4     )
5
6     int[] letters = new int[128]; // Assumption
7
8     char[] s_array = s.toCharArray();
9     for (char c : s_array) { // count number of each char in s.
10       letters[c]++;
11    )
12
13    for (int i = 0; i < t.length(); i++) {
14       int c = (int) t.charAt(i);
15       letters[c]--;
16       if (letters[c] < 0) {
17          return false;
18       )
19    )
20
21    return true;
22 )
```

Note the assumption on line 6. In your interview, you should always check with your interviewer about the size of the character set. We assumed that the character set was ASCII.

1.3 URLify: Write a method to replace all spaces in a string with '%20'. You may assume that [illegible]ing

1.3 **URLify:** Write a method to replace all spaces in a string with '%20'. You may assume that the string has sufficient space atthe end to hold the additional characters, and that you are given the "true" length of the string. (Note: If implementing inJava, please use a character array so that you can perform this operation in place.)

EXAMPLE

Input: "Mr John Smith ", 13

Output: "Mr%20John%20Smith"

Hints: #53, #118

size of the character set. We assumed

1.3 **URLify:** Write a method to replace all spaces in a string with '%20'. You may assume that the string has sufficient space at the end to hold the additional characters, and that you are given the "true" length of the string. (Note: if implementing in Java, please use a character array so that you can perform this operation in place.)

EXAMPLE

Input: "Mr John Smith ", 13

Output: "Mr%20John%20Smith"

pg 90

SOLUTION

A common approach in string manipulation problems is to edit the string starting from the end and working backwards. This is useful because we have an extra buffer at the end, which allows us to change characters without worrying about what we're overwriting.

We will use this approach in this problem. The algorithm employs a two-scan approach. In the first scan, we count the number of spaces. By tripling this number, we can compute how many extra characters we will have in the final string. In the second pass, which is done in reverse order, we actually edit the string. When we see a space, we replace it with %20. If there is no space, then we copy the original character.

The code below implements this algorithm.

```
1  void replaceSpaces(char[] str, int trueLength) {
2    int spaceCount = 0, index, i = 0;
3    for (i = 0; i < trueLength; i++) {
4      if (str[i] == ' ') {
5        spaceCount++;
6      }
7    }
8    index = trueLength + spaceCount * 2;
9    if (trueLength < str.length) str[trueLength]= '\0'; // End array
10   for (i = trueLength - 1; i >= 0; i--) {
11     if (str[i] == ' ') {
12       str[index - 1] = '0';
13       str[index - 2] = '2';
14       str[index-  3] = '%';
15       index = index - 3;
16     } else {
17       str[index - 1] = str[i];
18       index--;
19     }
20   }
21 }
```

We have implemented this problem using character arrays, because Java strings are immutable. If we used strings directly, the function would have to return a new copy of the string, but it would allow us to implement this in just one pass.

1.4 **Palindrome Permutation:** Given a string, write a function to check if it is a permutation of a palindrome. A palindrome isa word or phrase that is the same forwards and backwards. A permutation is a rearrangement of letters. The palindrome does not need to be limited to just dictionary words.

EXAMPLE

Input: `Tact Coa`

Output: `True (permutations: "taco cat", "atco cta", etc.)`

Hints: #106, #121, #134, #136

1.4 **Palindrome Permutation:** Given a string, write a function to check if it is a permutation of a palindrome. A palindrome is a word or phrase that is the same forwards and backwards. A permutation is a rearrangement of letters. The palindrome does not need to be limited to just dictionary words.

EXAMPLE

Input: `Tact Coa`

Output: `True` (permutations: "`taco cat`", "`atco cta`", etc.)

pg 91

SOLUTION

This is a question where it helps to figure out what it means for a string to be a permutation of a palindrome. This is like asking what the "defining features" of such a string would be.

A palindrome is a string that is the same forwards and backwards. Therefore, to decide if a string is a permutation of a palindrome, we need to know if it can be written such that it's the same forwards and backwards.

What does it take to be able to write a set of characters the same way forwards and backwards? We need to have an even number of almost all characters, so that half can be on one side and half can be on the other side. At most one character (the middle character) can have an odd count.

For example, we know `tactcoapapa` is a permutation of a palindrome because it has two Ts, four As, two

Cs, two Ps, and one 0. That 0 would be the center of all possible palindromes.

To be more precise, strings with even length (after removing all non-letter characters) must have all even counts of characters. Strings of an odd length must have exactly one character with an odd count. Of course, an "even" string can't have an odd number of exactly one character, otherwise it wouldn't be an even-length string (an odd number + many even numbers = an odd number). Likewise, a string with odd length can't have all characters with even counts (sum of evens is even). It's therefore sufficient to say that, to be a permutation of a palindrome, a string can have no more than one character that is odd. This will cover both the odd and the even cases.

This leads us to our first algorithm.

Solution #1

Implementing this algorithm is fairly straightforward. We use a hash table to count how many times each character appears. Then, we iterate through the hash table and ensure that no more than one character has an odd count.

```
1  boolean isPermutationOfPalindrome(String phrase) {
2    int[] table = buildCharFrequencyTable(phrase);
3    return checkMaxOneOdd(table);
4  )
5
6  /* Check that no more than one character has an odd count.*/
7  boolean checkMaxOneOdd(int[] table){
8    boolean foundOdd = false;
9    for (int count : table) {
10     if (count % 2 == 1) {
11       if (foundOdd) {
12       return false;
13       )
14     foundOdd = true;
15     )
16   )
17   return true;
18 )
19
20 /* Map each character to a number. a -> 0, b -> 1, c-> 2,  etc.
21  * This is case insensitive. Non-letter characters map to -1. */
22 int getCharNumber(Character c) {
23   int a = Character.getNumericValue('a');
24   int z = Character.getNumericValue('z');
25   int val = Character.getNumericValue(c);
26   if (a <= val && val <= z){
27     return val - a;
28   )
29   return -1,
30 )
31
32 /* Count how many times each character appears. */
33 int[] buildCharFrequencyTable(String phrase) {
34   int[] table = new int[Character.getNumericValue('z') -
35                         Character.getNumericValue('a') +1];
36   for (char c : phrase.toCharArray()) {
37     int x = getCharNumber(c);
```

```
        if (x != -1) {
38          table[x]++;
39      )
40
41  )
42  return table;
43 )
```

This algorithm takes O(N) time, where N is the length of the string.

Solution #2

We can't optimize the big O time here since any algorithm will always have to look through the entire string. However, we can make some smaller incremental improvements. Because this is a relatively simple problem, it can be worthwhile to discuss some small optimizations or at least some tweaks.

Instead of checking the number of odd counts at the end, we can check as we go along. Then, as soon as we get to the end, we have our answer.

```
1  boolean isPermutationOfPalindrome(String phrase) {
2      int countOdd = 0;
3      int[] table = new int[Character.getNumericValue('z')
4                            Character.getNumericValue('a') + 1];
5      for (char c : phrase.toCharArray()) {
6          int x = getCharNumber(c);
7          if (x != -1) {
8              table[x]++;
9              if (table[x] % 2 == 1) {
10                 countOdd++;
11             ) else {
12                 countOdd--;
13             )
14         )
15     )
16     return countOdd <= 1;
16 )
```

It's important to be very clear here that this is not necessarily more optimal. It has the same big O time and might even be slightly slower. We have eliminated a final iteration through the hash table, but now we have to run a few extra lines of code for each character in the string.

You should discuss this with your interviewer as an alternate, but not necessarily more optimal, solution.

Solution #3

If you think more deeply about this problem, you might notice that we don't actually need to know the counts. We just need to know if the count is even or odd. Think about flipping a light on/off (that is initially off). If the light winds up in the off state, we don't know how many times we flipped it, but we do know it was an even count.

Given this, we can use a single integer (as a bit vector). When we see a letter, we map it to an integer between 0 and 26 (assuming an English alphabet). Then we toggle the bit at that value. At the end of the iteration, we check that at most one bit in the integer is set to 1.

We can easily check that no bits in the integer are 1: just compare the integer to 0. There is actually a very elegant way to check that an integer has exactly one bit set to 1.

Picture an integer like `00010000`. We could of course shift the integer repeatedly to check that there's only a single 1. Alternatively, if we subtract 1 from the number, we'll get `00001111`. What's notable about this

is that there is no overlap between the numbers (as opposed to say `00101000`, which, when we subtract 1 from, we get `00100111`.) So, we can check to see that a number has exactly one 1 because if we subtract 1 from it and then AND it with the new number, we should get `0`.

```
00010000 - 1 = 00001111
00010000 & 00001111 = 0
```

This leads us to our final implementation.

```
1  boolean isPermutationOfPalindrome(String phrase)  {
2    int bitVector = createBitVector(phrase);
3    return bitVector == 0 || checkExactlyOneBitSet(bitVector);
4  }
5
6  /* Create a bit vector for the string. For each letter with value i, toggle the
7   * ith bit. */
8  int createBitVector(String phrase) {
9    int bitVector = 0;
10   for (char c : phrase.toCharArray()) {
11     int x = getCharNumber(c);
12     bitVector = toggle(bitVector, x);
13   }
14   return bitVector;
15 }
16
17 /* Toggle the ith bit in the integer. */
18 int toggle(int bitVector, int index) {
19   if (index < 0) return bitVector;
20
21   int mask = 1 << index;
22   if ((bitVector & mask) == 0) {
23     bitVector |= mask;
24   } else {
25     bitVector &= ~mask;
26   }
27   return bitVector;
28 }
29
30 /* Check that exactly one bit is set by subtracting one from the integer and
31  * ANDing it with the original integer. */
32 boolean checkExactlyOneBitSet(int bitVector) {
33   return (bitVector & (bitVector - 1)) == 0;
34 }
```

Like the other solutions, this is `O(N)`.

It's interesting to note a solution that we did not explore. We avoided solutions along the lines of "create all possible permutations and check if they are palindromes." While such a solution would work, it's entirely infeasible in the real world. Generating all permutations requires factorial time (which is actually worse than exponential time), and it is essentially infeasible to perform on strings longer than about 10-15 characters.

I mention this (impractical) solution because a lot of candidates hear a problem like this and say, "In order to check if A is in group B, I must know everything that is in B and then check if one of the items equals A." That's not always the case, and this problem is a simple demonstration of it. You don't need to generate all permutations in order to check if one is a palindrome.

1.5 **One Away:** There are three types of edits that can be performed on strings: insert a character, remove a character, or replace a character. Given two strings, write a function to check if they are one edit (or zero edits) away.

EXAMPLE

```
pale,  ple   -> true
pales, pale  -> true
pale,  bale  -> true
pale,  bake  -> false
```

1 edit
1 edit
1 edit
2 edits

Hints: #23, #97, #130

1.5 **One Away:** There are three types of edits that can be performed on strings: insert a character, remove a character, or replace a character. Given two strings, write a function to check if they are one edit (or zero edits) away.

EXAMPLE

```
pale,  ple  -> true
pales, pale -> true
pale,  bale -> true
pale,  bae -> false
```

pg 91

SOLUTION

There is a "brute force" algorithm to do this. We could check all possible strings that are one edit away by testing the removal of each character (and comparing), testing the replacement of each character (and comparing), and then testing the insertion of each possible character (and comparing).

That would be too slow, so let's not bother with implementing it.

This is one of those problems where it's helpful to think about the "meaning" of each of these operations. What does it mean for two strings to be one insertion, replacement, or removal away from each other?

- **Replacement:** Consider two strings, such as `bale` and `pale`, that are one replacement away. Yes, that does mean that you could replace a character in `bale` to make `pale`. But more precisely, it means that they are different only in one place.
- **Insertion:** The strings `apple` and `aple` are one insertion away. This means that if you compared the strings, they would be identical—except for a shift at some point in the strings.
- **Removal:** The strings `apple` and `aple` are also one removal away, since removal is just the inverse of insertion.

We can go ahead and implement this algorithm now. We'll merge the insertion and removal check into one step, and check the replacement step separately.

Observe that you don't need to check the strings for insertion, removal, and replacement edits. The lengths of the strings will indicate which of these you need to check.

```
1  boolean oneEditAway(String first, String second) {
2    if (first.length() == second.length()) {
3       return oneEditReplace(first, second);
4    ) else if (first.length() + 1 == second.length()) {
5       return oneEditInsert(first, second);
6    ) else if (first.length() - 1 == second.length()) {
7       return oneEditInsert(second, first);
8    )
9    return false;
10 )
11
12 boolean oneEditReplace(String s1, String s2) {
13   boolean foundDifference = false;
14   for (int i = 0; i < s1.length(); i++) {
15     if (s1.charAt(i) != s2.charAt(i)) {
16       if (foundDifference) {
17         return false;
18       )
19
```

```
            foundDifference = true;
        }
    }
    return true;
}

/* Check if you can insert a character into s1 to make s2. */
boolean oneEditInsert(String s1, String s2) {
    int index1 = 0;
    int index2 = 0;
    while (index2 < s2.length() && index1 < s1.length()) {
        if (s1.charAt(index1) != s2.charAt(index2)) {
            if (index1 != index2) {
                return false;
            }
            index2++;
        } else {
            index1++;
            index2++;
        }
    }
    return true;
}
```

This algorithm (and almost any reasonable algorithm) takes `O(n)` time, where n is the length of the shorter string.

> Why is the runtime dictated by the shorter string instead of the longer string? If the strings are the same length (plus or minus one character), then it doesn't matter whether we use the longer string or the shorter string to define the runtime. If the strings are very different lengths, then the algorithm will terminate in `O(1)` time. One really, really long string therefore won't significantly extend the runtime. It increases the runtime only if both strings are long.

We might notice that the code for `oneEditReplace` is very similar to that for `oneEditInsert`. We can merge them into one method.

To do this, observe that both methods follow similar logic: compare each character and ensure that the strings are only different by one. The methods vary in how they handle that difference. The method `oneEditReplace` does nothing other than flag the difference, whereas `oneEditInsert` increments the pointer to the longer string. We can handle both of these in the same method.

```
boolean oneEditAway(String first, String second) {
    /* Length checks. */
    if (Math.abs(first.length() - second.length()) > 1) {
        return false;
    }

    /* Get shorter and longer string.*/
    String s1 = first.length() < second.length() ? first : second;
    String s2 = first.length() < second.length() ? second : first;

    int index1 = 0;
    int index2 = 0;
    boolean foundDifference = false;
    while (index2 < s2.length() && index1 < s1.length()) {
        if (s1.charAt(index1) != s2.charAt(index2)) {
```

```
            /* Ensure that this is the first difference found.*/
16          if (foundDifference) return false;
17          foundDifference = true;
18
19          if (s1.length() == s2.1ength()) { // On replace, move shorter pointer
20            index1++;
21          )
22        ) else {
23          index1++; // If matching, move shorter pointer
24        )
25        index2++; // Always move pointer for longer string
26      )
27      return true;
28    )
29
```

Some people might argue the first approach is better, as it is clearer and easier to follow. Others, however, will argue that the second approach is better, since it's more compact and doesn't duplicate code (which can facilitate maintainability).

You don't necessarily need to "pick a side." You can discuss the tradeoffs with your interviewer.

Hints: #23, #44

1.6 **String Compression:** Implement a method to perform basic string compression using the counts of repeated characters. For example, the string aabcccccaaa would become a2b1c5a3. If the "compressed" string would not become smaller than the original string, your method should return the original string. You can assume the string has only uppercase and lowercase letters (a - z).

pg 199

Hints: #92, #110

---

pg 201

1.6 **String Compression:** Implement a method to perform basic string compression using the counts of repeated characters For example, the string `aabcccccaaa` would become `a2b1c5a3`. If the "compressed" string would not become smaller than the original string, your method should return the original string. You can assume the string has only uppercase and lowercase letters (a - z).

pg 91

SOLUTION

At first glance, implementing this method seems fairly straightforward, but perhaps a bit tedious. We iterate through the string, copying characters to a new string and counting the repeats. At each iteration, check if the current character is the same as the next character. If not, add its compressed version to the result.

How hard could it be?

```
1  String compressBad(String str) {
2    String compressedString = "";
3    int countConsecutive = 0;
4    for (int i = 0; i < str.length(); i++)
5      countConsecutive++;
6
7      /* If next character is different than current, append this char to result.*/
8      if (i + 1 >= str.length() ||  str.charAt(i) != str.charAt(i + 1)) {
9        compressedString += "" + str.charAt(i) + countConsecutive;
10       countConsecutive = 0;
11     )
12   )
13   return compressedString.length() < str.length() ? compressedString : str;
14 )
```

This works. Is it efficient, though? Take a look at the runtime of this code.

The runtime is $O(p + k^2)$, where p is the size of the original string and k is the number of character sequences. For example, if the string is `aabccdeeaa`, then there are six character sequences. It's slow because string concatenation operates in $O(n^2)$ time (see `StringBuilder` on pg 89).

We can fix this by using a `StringBuilder`.

```
String compress(String str) {
   StringBuilder compressed = new StringBuilder();
   int countConsecutive = 0;
   for (int i = 0; i < str.length(); i++) {
      countConsecutive++;

      /* If next character is different than current, append this char to result.*/
      if (i + 1 >= str.length() || str.charAt(i) !. str.charAt(i + 1)) {
         compressed.append(str.charAt(i));
         compressed.append(countConsecutive);
         countConsecutive = 0;
      }
   }
   return compressed.length() < str.length() ? compressed.toString() : str;
}
```

Both of these solutions create the compressed string first and then return the shorter of the input string and the compressed string.

Instead, we can check in advance. This will be more optimal in cases where we don't have a large number of repeating characters. It will avoid us having to create a string that we never use. The downside of this is that it causes a second loop through the characters and also adds nearly duplicated code.

```
String compress(String str) {
   /* Check final length and return input string if it would be longer. */
   int finalLength = countCompression(str);
   if (finalLength >= str.length()) return str;

   StringBuilder compressed = new StringBuilder(finalLength); // initial capacity
   int countConsecutive = 0;
   for (int i = 0; i < str.length(); i++) {
      countConsecutive++;

      /* If next character is different than current, append this char to result.*/
      if (i + 1 >. str.length() || str.charAt(i) != str.charAt(i + 1)) {
         compressed.append(str.charAt(i));
         compressed.append(countConsecutive);
         countConsecutive = 0;
      }
   }
   return compressed.toString();
}

int countCompression(String str) {
   int compressedLength = 0;
   int countConsecutive = 0;
   for (int i = 0; i < str.length(); i++) {
      countConsecutive++;

      /* If next character is different than current, increase the length.*/
      if (i + 1 >= str.length() || str.charAt(i) != str.charAt(i + 1)) {
         compressedLength += 1 + String.valueOf(countConsecutive).1ength();
         countConsecutive = 0;
      }
   }
   return compressedLength;
}
```

One other benefit of this approach is that we can initialize `StringBuilder` to its necessary capacity up-front. Without this, `StringBuilder` will (behind the scenes) need to double its capacity every time it hits capacity. The capacity could be double what we ultimately need.

1.7 **Rotate Matrix:** Given an image represent

pg 204

1.9 **String Rotation:** Assume you have a method `isSubstring`which checks if one word is a substring of another. Given two strings, `s1` and `s2`, write code to check if `s2` is a rotation of `s1` using only one call to `isSubstring` (e.g., "`waterbottle`" is a rotation of "`erbottlewat`").

Hints: #34, #88, #104

1.9 **String Rotation:** Assume you have a method `isSubstring` which checks if one word is a substring of another. Given two strings, `s1` and `s2`, write code to check if `s2` is a rotation of `s1` using only one call to `isSubstring` (e.g., "`waterbottle`" is a rotation of "`erbottlewat`").

pg 91

SOLUTION

If we imagine that `s2` is a rotation of `s1`, then we can ask what the rotation point is. For example, if you rotate `waterbottle` after `wat`, you get `erbottlewat`. In a rotation, we cut `s1` into two parts, `x` and `y`, and rearrange them to get `s2`.

```
s1 = xy = waterbottle
x = wat
```

```
y = erbottle
s2 = yx = erbottlewat
```

So, we need to check if there's a way to split s1 into x and y such that xy = s1 and yx = s2. Regardless of where the division between x and y is, we can see that yx will always be a substring of xyxy. That is, s2 will always be a substring of s1s1.

And this is precisely how we solve the problem: simply do `isSubstring(s1s1, s2)`.

The code below implements this algorithm.

```
1  boolean isRotation(String s1, String s2) {
2    int len = s1.length();
3    /* Check that s1 and s2 are equal length and not empty */
4    if (len == s2.length() && len > 0) {
5      /* Concatenate s1 and s1 within new buffer */
6      String s1s1 = s1 + s1;
7      return isSubstring(s1s1, s2);
8    )
9    return false;
10 )
```

The runtime of this varies based on the runtime of `isSubstring`. But if you assume that `isSubstring` runs in O(A+B) time (on strings of length A and B), then the runtime of `isRotation` is O(N).