Extraordinary Substrings Hackerrank Solution

M Mosston

Cracking the Code: An In-Depth Guide to the Extraordinary Substrings HackerRank Solution

So, you've stumbled upon the "Extraordinary Substrings" problem on HackerRank. It's a fun challenge that tests your understanding of string manipulation and potentially, dynamic programming. Don't worry, we're going to break it down step-by-step, from understanding the problem to crafting a robust and efficient solution. This isn't just a solution; it's a journey into the world of algorithmic thinking.

Understanding the Problem:

The core of the "Extraordinary Substrings" problem (the specific phrasing might vary slightly depending on the platform's version) revolves around identifying substrings within a given string that meet a specific criterion: each character in the substring must appear an odd number of times. Let's illustrate with an example:

Consider the string "aabbcc". Substrings like "abc" or "aabbc" wouldn't qualify because some characters appear an even number of times. However, "a", "b", "c", "aa", "bb", "cc", and "aabbcc" would be considered extraordinary substrings. Our goal is to count all such extraordinary substrings.

Visualizing the Problem:

Imagine the string "abca". We can visualize all possible substrings and then check the character counts for oddness:

```
a - Extraordinary (a: 1)
```

```
ab - Not Extraordinary (a:1, b:1)
abc - Not Extraordinary (a:1, b:1, c:1)
abca- Extraordinary (a:2, b:1, c:1)
b - Extraordinary (b: 1)
bc - Not Extraordinary (b:1, c:1)
bca - Extraordinary (b:1, c:2, a:1)
c - Extraordinary (c: 1)
ca - Not Extraordinary (c:1, a:1)
a - Extraordinary (a: 1)
```

As you can see, manually counting these gets tedious quickly. That's where algorithmic thinking shines!

How-To: A Step-by-Step Approach to the Solution

Several approaches can solve this problem, but a common and efficient one involves using bit manipulation and dynamic programming. Let's break down this method:

1. Bit Manipulation Representation: We'll use a bitmask to represent the parity (even or odd) of the character counts. Each bit in the mask corresponds to a character in the alphabet (assuming lowercase). If a bit is set (1), the corresponding character appears an odd number of times; otherwise (0), it appears an even number of times.

2. Dynamic Programming Approach: We'll create a table (or dictionary) to store the count of extraordinary substrings ending at each index of the input string. The key is to leverage the counts calculated for previous substrings to efficiently determine the counts for the current substring.

3. Iterative Calculation: We iterate through the input string. For each character, we update the bitmask based on its presence. If the updated bitmask has all bits set (meaning all characters in the substring have an odd count), we increment our extraordinary substring count.

Python Code Example:

```python
def extraordinarySubstrings(s):
count = 0
mask = 0
dp = {0: 1} # Initialize dp with an empty substring having count 1

for char in s:
```

```
index = ord(char) - ord('a') #Get character index (0-25)
mask ^= (1 << index) #XOR operation to toggle the bit

if mask in dp:
count += dp[mask]
dp[mask] += 1
else:
dp[mask] = 1

return count
```

Example usage

```
string = "aabbcc"
result = extraordinarySubstrings(string)
print(f"Number of extraordinary substrings in '{string}': {result}")
```

Explanation of the Code:

`ord(char) - ord('a')`: This line converts each character into its numerical index (a=0, b=1, etc.).
`mask ^= (1 << index)`: This performs a bitwise XOR operation to toggle the bit corresponding to the current character in the `mask`.
`dp`: This dictionary stores the counts of substrings ending at each index for a particular bitmask.

Optimizations and Considerations:

Space Complexity: The space complexity of the above solution is O(2^26) in the worst case (all possible bitmasks for a 26-letter alphabet), which can be quite large. For smaller alphabets or constraints on input string length, this is generally acceptable. You can optimize further if space becomes a critical factor.
Alphabet Size: The code assumes a lowercase English alphabet. You might need to adjust the `ord()` calculations if you're working with a different character set.

Key Points Summary:

The "Extraordinary Substrings" problem focuses on counting substrings where each character appears an odd number of times.
Bit manipulation is an efficient way to represent and update character counts.

Dynamic programming significantly reduces redundant calculations, leading to an optimized solution. Consider space complexity when dealing with large alphabets.

Frequently Asked Questions (FAQs):

1. Why use bit manipulation? Bit manipulation allows for compact and fast representation of character counts. Checking for odd counts becomes a simple bitwise check.

2. What if the input string is very long? For extremely long strings, consider more space-efficient data structures or techniques to handle the dynamic programming table. You might explore optimized hashmaps or even approximation algorithms.

3. Can I solve this without dynamic programming? Yes, but it would be significantly less efficient. A brute-force approach would require checking every possible substring, leading to an O(n^3) time complexity (where n is the string length).

4. What if the input string contains uppercase letters? You would need to modify the code to handle both uppercase and lowercase letters, either by converting everything to lowercase or expanding the bitmask to include both cases.

5. How can I further optimize my solution? Analyze the code's performance with profiling tools to identify bottlenecks. Consider using faster data structures or exploring different algorithms if performance is critical.

This comprehensive guide provides you with a solid understanding of the "Extraordinary Substrings" problem and a practical, efficient solution using Python. Remember to practice and experiment with different approaches to further refine your problem-solving skills! Happy coding!

Link Note Extraordinary Substrings Hackerrank Solution

stop yelling and love me more please mom
positive parenting is easier than you think
jennifer n smith
sonic the hedgehog 30th anniversary celebration
ian flynn
stones of jerusalem bodie thoene

Finding sum of all integer substring using dynamic programming Aug 13, 2022 · I was solving Sam and substrings problem from hackerrank. It is basically finding sum of all substrings of a string having all integers. Samantha and Sam are playing a numbers …

Number of Wonderful Substrings - LeetCode Given a string word that consists of the first ten lowercase English letters ('a' through 'j'), return the number of wonderful non-empty substrings in word. If the same substring appears multiple …

perrinod/hacker-rank-solutions: My HackerRank Solutions - GitHub My HackerRank Solutions. Contribute to perrinod/hacker-rank-solutions development by creating an account on GitHub.

HackerRank: Sam and substrings - Code Review Stack Exchange Jan 19, 2022 · Given a number as a string, no leading zeros, determine the sum of all integer values of substrings of the string. Given an integer as a string, sum all of its substrings cast as …

HackerRank Challenge - Special Substrings · GitHub SPECIAL [str] ||= all_substrs (str).select { |i| palindrome? (i) } HackerRank Challenge - Special Substrings. GitHub Gist: instantly share code, notes, and snippets.

Sam and substrings - HackerRank Samantha and Sam are playing a numbers game. Given a number as a string, no leading zeros, determine the sum of all integer values of substrings of the string. Given an integer as a string, …

Hackerrank-Solutions/Sam and substrings at main - GitHub Link- https://www.hackerrank.com/challenges/sam-and-substrings/problem #include const int mod = 1000000007; using namespace std; // Complete the …

Sam and substrings Discussions | Algorithms - HackerRank Each digit contributes to multiple substrings. You can calculate the total contribution of each digit by considering its position in the number. Efficient Calculation: Use a mathematical formula to …

Hackerrank-solutions/Sam and sub-strings at master - GitHub You signed in with another tab or window. Reload to refresh your session. You signed out in another tab or window. Reload to refresh your session. You switched accounts on another tab …
Substring calculation with Java HackerRank Solutions Calculate the result using substring () method and print it. This Java solution optimally calculates substrings to tackle HackerRank challenges. Leveraging dynamic programming and efficient …

HackerRank_solutions/Java/Strings/Java Substring/Solution.java … 317 efficient solutions to HackerRank problems. Contribute to RodneyShag/HackerRank_solutions development by creating an account on GitHub.

Extraordinary Substrings Hackerrank Solution [PDF] Extraordinary substrings HackerRank solution: This comprehensive guide delves into the intricacies of finding extraordinary substrings within a given string, providing a detailed …

hackerrank/contests/w3/sam-and-substrings.java at master N [n-1] (1)*10^ (n-1) = RSum (RSum (N [j] (n-j) , j from i to n - 1) * 10^i , i from 0 to n - 1) */ public class Solution { private final static int MOD = 1000000007; public static void main (String [] …

Hackerrank Java Substring Comparisons · GitHub public static String getSmallestAndLargest(String s, int k) { String smallest = ""; String largest = ""; int count=0; String[] str=new String[s.length()-k+1]; for (int i =0;i