1461. Check If a String Contains All Binary Codes of Size K

31 May 2022 05:18 PM

Given a binary string `s` and an integer `k`, return `true` *if every binary code of length* `k` *is a substring of* `s`. Otherwise, return `false`.

Example 1:

```
Input: s = "00110110", k = 2
Output: true
Explanation: The binary codes of length 2 are "00",
"01", "10" and "11". They can be all found as
substrings at indices 0, 1, 3 and 2 respectively.
```

Example 2:

```
Input: s = "0110", k = 1
Output: true
Explanation: The binary codes of length 1 are "0" and
"1", it is clear that both exist as a substring.
```

Example 3:

```
Input: s = "0110", k = 2
Output: false
Explanation: The binary code "00" is of length 2 and
does not exist in the array.
```

1) Brute force :

$\rightarrow O(2^k)$

- Generate all binary code of length k
- Check if the generated code is present in S $\rightarrow O(n)$

→ if (S. contains (All the generated_code)) ⇒ True

→ Else return false

T.C ⇒ $O(n 2^k)$

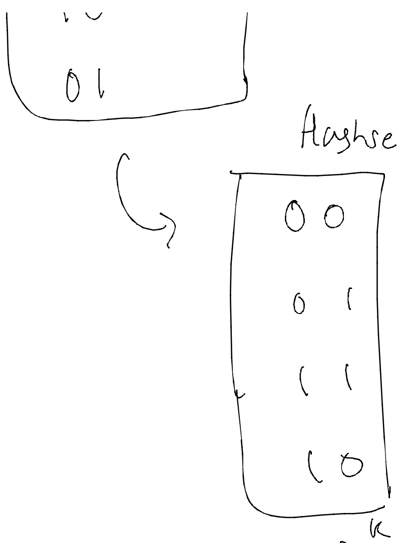
2) Optimised

S = "00110110", K = 2

* Creating all the substring of size k from the input s

00 11
01 10
11
10
01

⇒ Complete list of substring of size 2 and to remove duplicate use HashSet



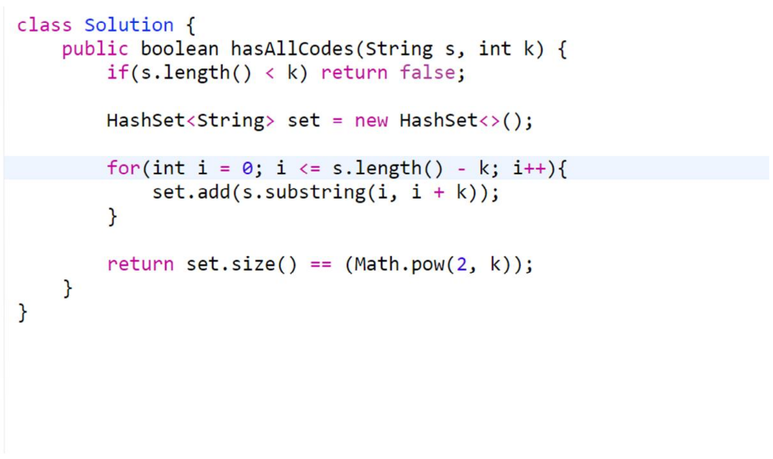
$$2^k = 2^2 = 4 == 4 \text{ (Hashset)}$$

Binary code of $k=2 \Rightarrow 2^k$ and compare with Hashset if it exact match return true else return false

The index of the array start from 0, we need to create substrings from [0 index to n−k+1) or [0, n−k], In this case k=2 so it becomes n−2+1, which is n−1.

$$T.C = O(n-k) \cancel{\times} O(k) \qquad S.C = O(2^k)$$

↳ for traversing s



```
class Solution {
    public boolean hasAllCodes(String s, int k) {
        if(s.length() < k) return false;

        HashSet<String> set = new HashSet<>();

        for(int i = 0; i <= s.length() - k; i++){
            set.add(s.substring(i, i + k));
        }

        return set.size() == (Math.pow(2, k));
    }
}
```

LeetCode Solution :

Approach 2: Hash

Maybe you think the approach above is not fast enough. Let's write the hash function ourselves to improve the speed.

Note that we will have at most 2^k string, can we map each string to a number in $[0, 2^k - 1]$?

We can. Recall the binary number, we can treat the string as a binary number, and take its decimal form as the hash value. In this case, each binary number has a unique hash value. Moreover, the minimum is all `0`, which is zero, while the maximum is all `1`, which is exactly $2^k - 1$.

Because we can directly apply bitwise operations to decimal numbers, it is not even necessary to convert the binary number to a decimal number explicitly.

What's more, we can get the current hash from the last one. This method is called Rolling Hash. All we need to do is to remove the most significant

> For example, say `s="11010110"`, and `k=3`, and we just finish calculating the hash of the first substring: `"110"` (`hash` is 4+2=6, or `110`). Now we want to know the next hash, which is the hash of `"101"`.
>
> We can start from the binary form of our hash, which is `110`. First, we shift left, resulting `1100`. We do not need the first digit, so it is a good idea to do `1100 & 111 = 100`. The all-one `111` helps us to align the digits. Now we need to apply the lowest digit of `"101"`, which is `1`, to our hash, and by using `|`, we get `100 | last_digit = 100 | 1 = 101`.

Write them together, we have: `new_hash = ((old_hash << 1) & all_one) | last_digit_of_new_hash`.

```
    public static boolean hasAllCodes(String s, int k) {
        int need = 1 << k;
        boolean[] got = new boolean[need];
        int allOne = need - 1;
        int hashVal = 0;

        for (int i = 0; i < s.length(); i++) {
            // calculate hash for s.substr(i-k+1,i+1)
            hashVal = ((hashVal << 1) & allOne) |
(s.charAt(i) - '0');
            // hash only available when i-k+1 > 0
            if (i >= k - 1 && !got[hashVal]) {
                got[hashVal] = true;
                need--;
                if (need == 0) {
                    return true;
                }
```

Let N be the length of `s`.

- Time complexity: $\mathcal{O}(N)$. We need to iterate the string, and use $\mathcal{O}(1)$ to calculate the hash of each substring.
- Space complexity: $\mathcal{O}(2^k)$. There are 2^k elements in the list.

```
    public static boolean hasAllCodes(String s, int k) {
        int need = 1 << k;
        boolean[] got = new boolean[need];
        int allOne = need - 1;
        int hashVal = 0;

        for (int i = 0; i < s.length(); i++) {
            // calculate hash for s.substr(i-k+1,i+1)
            hashVal = ((hashVal << 1) & allOne) |
(s.charAt(i) - '0');
            // hash only available when i-k+1 > 0
            if (i >= k - 1 && !got[hashVal]) {
                got[hashVal] = true;
                need--;
                if (need == 0) {
                    return true;
                }
            }
        }
        return false;
```

- Space complexity: $O(2^k)$. There are 2^k elements in the list.