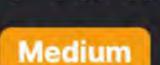
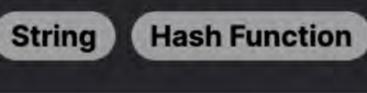
# 1461. Check If a String Contains All Binary Codes of Size K



**Bit Manipulation** 

Hash Table



Rolling Hash

Leetcode Link

## **Problem Description**

determine whether or not every possible binary number of length k could be found as a substring within the binary string s. A substring is any consecutive sequence of characters within a string. If all possible binary numbers of length k are present, we should return true. If at least one such binary number is missing, we should return false.

In this problem, we are given a binary string s which is comprised of only '0's and '1's. We are also given an integer k. The task is to

present but '00' and '11' are not, so the function should return false. The problem challenges us to examine segments within the string and to account for every possible combination without missing

For example, if s = "0110" and k = 2, the binary codes of length 2 are '00', '01', '10', and '11'. In the given string '01' and '10' are

any, ensuring the string s could represent all possible binary numbers of the given length k.

#### To solve this problem, the intuition is to generate all possible substrings of length k from the given string s and store them in a set to

Intuition

The next step is to compare the number of unique substrings obtained with the total number of possible combinations for a binary number of length k. Since there are two possible values (0 or 1) for each position, the total number of unique k-length binary

avoid duplicates. A set is chosen because it automatically handles duplicates and allows us to count unique substrings efficiently.

numbers is 2<sup>k</sup>. The Python code accomplishes this by:

2. Checking if the number of unique substrings (len(ss)) equals 2<sup>k</sup> (expressed in Python as 1 << k, which is a bit shift operation equivalent to raising 2 to the power of k).

If these two numbers match, all binary codes of length k are present in the string s, and the function returns true. Otherwise, at least

Generating all possible k-length substrings from susing a set comprehension.

one code is missing, so it returns false.

Solution Approach The solution to this problem uses set comprehension and bitwise operations as its main algorithmic tools. Let's walk through the

### 1. Set comprehension: The solution begins by creating a set named ss. Set comprehension is used to populate ss with every

implementation step by step:

1 return len(ss) == 1 << k

unique substring of length k present in the string s. This is done by iterating over the string with a moving window of size k. For each position i from 0 to (len(s) - k), a substring of length k is extracted and added to the set.

The range used for iteration goes up to len(s) - k + 1 because when extracting a substring of length k starting at position i, the last valid position of i is when i + k equals len(s), meaning the substring ends at the last character of s. 2. Bitwise Shift: To determine the total number of unique binary numbers of length k, the solution uses the bitwise left shift

by 2 exactly k times. The result is 2<sup>k</sup>, which represents the total number of unique binary strings of length k.

```
operation 1 << k. This operation shifts the number 1 to the left k times in binary, which is equivalent to multiplying the number 1
```

by leveraging the properties of binary numbers and bitwise operations.

Move to the next character: The next substring is "011".

binary numbers (000, 001, 010, 011, 100, 101, 110, 111).

By performing the bitwise shift operation 1 << k:

1  $ss = \{s[i:i+k] \text{ for } i \text{ in range}(len(s) - k + 1)\}$ 

calculation of total combinations without having to invoke heavier mathematical operations. This approach is both space-efficient, since only unique substrings are kept, and time-efficient, as it avoids unnecessary calculations

In this step, the solution compares the size of the set ss (the number of unique k-length substrings in s) with 2^k. If the sizes

match, then s contains every possible binary code of length k, and the method returns true. Otherwise, it returns false.

The data structure used (set) is essential for efficiently managing unique values and the operation (bitwise shift) simplifies the

Example Walkthrough

1. Set comprehension: We create a set ss to store unique substrings of length k from s. Here's how we do it:

### Continue this process until the end of the string: We then get "110".

1 ss = {"001", "011", "110"}

The set comprehension in the code is executed like this:

To illustrate the solution approach, let's consider a small example with the string s = "00110" and k = 3.

Start at the beginning of the string: The first k-length substring is "001".

- Now, ss contains all k-length substrings that appear in s. In this case, we have exactly 3 different substrings.
- 1 return len(ss) == 1 << 3 # Evaluates to 3 == 8

In this case, the condition is false because len(ss) is 3 but we need it to be 8 to cover all possible binary numbers of length 3.

Using this approach, the algorithm efficiently determines whether the binary string s contains every possible binary number as a

substring of length k. In this example, some binary numbers like '000', '010', '100', '101', and '111' are missing from s, so not all binary

2. Bitwise Shift: Now we need to check if we have all possible binary numbers of length k. Since k is 3, there are 2^3 or 8 possible

```
numbers of length k are represented in it.
```

def hasAllCodes(self, s: str, k: int) -> bool:

# binary strings of length k)

# Create a set to store all unique substrings of length k

# Check if the number of unique substrings of length k

# is equal to 2^k (which is the total number of possible

// If there are not enough substrings to cover all possible

// Create an array to keep track of which binary numbers

int currentValue = Integer.parseInt(s.substring(0, k), 2);

// Iterate over the string to check all possible substrings

// Calculate the bit to remove from the front of the

// Check if there is any k-bit code that has not been visited

return true; // All k-bit codes are found, return true

if (!isVisited) return false; // Return false if any code is not found

for (bool isVisited : visited) {

// Convert the first 'k' bits of the string to a numerical value.

// binary numbers of length k, return false.

if  $(stringLength - k + 1 < (1 << k)) {$ 

boolean[] visited = new boolean[1 << k];</pre>

for (int i = k; i < stringLength; ++i) {</pre>

return false;

// of length k.

// of length k we have seen.

visited[currentValue] = true;

unique\_substrings = {s[i: i + k] for i in range(len(s) - k + 1)}

Therefore, the output would be false.

**Python Solution** 

return len(unique\_substrings) == (1 << k) 10

```
public boolean hasAllCodes(String s, int k) {
   // Calculate the length of the string.
    int stringLength = s.length();
```

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

Java Solution

class Solution {

1 class Solution:

```
// current sliding window of size k.
                int frontBit = (s.charAt(i - k) - '0') \ll (k - 1);
27
               // Calculate the bit to add to the back of the current
28
29
               // sliding window of size k.
                int backBit = s.charAt(i) - '0';
30
31
               // Update the current sliding window value.
33
                currentValue = (currentValue - frontBit) << 1 | backBit;</pre>
34
35
               // Mark this new value as seen.
               visited[currentValue] = true;
36
37
38
           // If any binary number of length k hasn't been visited,
39
           // return false, otherwise return true.
40
            for (boolean hasVisited : visited) {
41
                if (!hasVisited) {
42
43
                    return false;
44
45
46
           return true;
47
48 }
49
C++ Solution
 1 class Solution {
2 public:
       bool hasAllCodes(string str, int k) {
            int strSize = str.size(); // Size of input string
           // Check if there is enough length in the string to have all k-bit codes
           if (strSize - k + 1 < (1 << k)) return false;
           // Initialize a boolean vector to track which k-bit patterns have been visited
           vector<bool> visited(1 << k, false);</pre>
12
           // Convert the first k bits of the string to a number and mark as visited
13
            int currentNumber = stoi(str.substr(0, k), nullptr, 2);
14
            visited[currentNumber] = true;
15
           // Traverse the rest of the string to find all distinct k-bit codes
16
17
            for (int i = k; i < strSize; ++i) {</pre>
               // Remove the leading bit and leave space at the end
19
                int leadingBit = (str[i - k] - '0') \ll (k - 1);
20
               // Extract the last bit of the current sliding window
21
               int lastBit = str[i] - '0';
22
               // Update currentNumber by removing the leading bit and adding the new trailing bit
23
                currentNumber = (currentNumber - leadingBit) << 1 | lastBit;</pre>
24
               // Mark the new k-bit number as visited
25
                visited[currentNumber] = true;
26
27
```

#### function hasAllCodes(str: string, k: number): boolean { let strSize: number = str.length; // Size of input string

Typescript Solution

28

29

30

31

32

33

35

34 };

```
// Check if there is enough length in the string to have all k-bit codes
        if (strSize - k + 1 < (1 << k)) return false;
        // Initialize an array to track which k-bit patterns have been visited
         let visited: boolean[] = new Array(1 << k).fill(false);</pre>
 10
         // Helper function to convert a binary string to a number
 11
         const binToNum = (bin: string): number => parseInt(bin, 2);
 12
 13
         // Convert the first k bits of the string to a number and mark as visited
 14
         let currentNumber: number = binToNum(str.substring(0, k));
         visited[currentNumber] = true;
 15
 16
 17
         // Traverse the rest of the string to find all distinct k-bit codes
         for (let i: number = k; i < strSize; ++i) {</pre>
 18
 19
             // Remove the leading bit and leave space at the end
             let leadingBit: number = (parseInt(str[i - k]) << (k - 1));</pre>
 20
 21
             // Extract the last bit of the current sliding window
             let lastBit: number = parseInt(str[i]);
 22
 23
             // Update currentNumber by removing the leading bit and adding the new trailing bit
             currentNumber = ((currentNumber - leadingBit) << 1) | lastBit;</pre>
 24
 25
             // Mark the new k-bit number as visited
 26
             visited[currentNumber] = true;
 27
 28
 29
        // Check if there is any k-bit code that has not been visited
 30
         for (let isVisited of visited) {
             if (!isVisited) return false; // Return false if any code is not found
 31
 32
 33
         return true; // All k-bit codes are found, return true
 34 }
 35
Time and Space Complexity
```

Therefore, the total time complexity of the function has AllCodes is 0(k \* (len(s) - k + 1)).

#### Space Complexity The space complexity is driven by the space required to store all unique substrings of length k from the string s. The maximum

**Time Complexity** The time complexity of the given code primarily comes from the set comprehension used to create a set ss of all substrings of length k from the string s. The comprehension iterates over each index of the string from 0 to len(s) - k inclusively, which results in len(s) - k + 1 iterations. For each iteration, a substring of length k is created, which has its own time complexity of 0(k). Thus, the overall time complexity of this part is 0(k \* (len(s) - k + 1)).

Additionally, the operation 1 << k performs a bitwise left shift which has a time complexity of 0(1) since it is an operation over a

fixed-size integer (which is independent of the input size).

number of unique substrings that can be stored in set ss is 20k, since each k-length substring is a possible combination of k bits, and there are 2<sup>k</sup> possible combinations of k bits.

which is len(s) - k + 1. Therefore, the space complexity of the set ss is  $0(min(2^k, len(s) - k + 1))$ .

Considering both possible limits, the overall space complexity of the function has AllCodes is  $0(\min(2^k, len(s) - k + 1))$ .

However, the number of substrings that can actually be stored is limited by the number of substrings that we can generate from s,