<https://leetcode.com/problems/check-if-a-string-contains-all-binary-codes-of-size-k/description/>

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.

**Constraints:**

1 <= s.length <= 5 \* 10^5

s[i] is either '0' or '1'.

1 <= k <= 20

**Attempt 1: 2025-03-16**

**Solution 1: Brute Force (30 min, TLE 198/201)**

class Solution {

public boolean hasAllCodes(String s, int k) {

int n = s.length();

// Total number of unique binary codes of length k

int totalCodes = 1 << k; // Equivalent to 2^k

// Iterate through all possible binary codes of length k

for (int num = 0; num < totalCodes; num++) {

// Generate the binary code for the current number

String binaryCode = Integer.toBinaryString(num);

// Pad with leading zeros to ensure the length is k

while (binaryCode.length() < k) {

binaryCode = "0" + binaryCode;

}

// Check if the binary code exists in the string s

if (!s.contains(binaryCode)) {

return false;

}

}

return true;

}

}

Time Complexity : O(N\*2^k), there are 2^k binary codes possible and for each code,

we need N iteration in worst case to find the binary code in the string

Space Complexity: O(1)

**Solution 2: Hash Table (10 min)**

class Solution {

    public boolean hasAllCodes(String s, int k) {

        int n = s.length();

        // Total number of unique binary codes of length k

        int totalCodes = 1 << k; // Equivalent to 2^k

        // If the string is too short to contain all codes, return false

        if (n < k) {

            return false;

        }

        // Use a set to store all unique substrings of length k

        Set<String> seen = new HashSet<>();

        // Slide a window of size k across the string

        for (int i = 0; i <= n - k; i++) {

            String substring = s.substring(i, i + k);

            seen.add(substring);

            // Early exit if all codes are found

            if (seen.size() == totalCodes) {

                return true;

            }

        }

        // Check if all codes were found

        return seen.size() == totalCodes;

    }

}

Time Complexity : O(N\*k), where N is the size of string and k is length of binary code.

Space Complexity : O(N\*k)

**Solution 3: Rolling Hash (30 min)**

class Solution {

    public boolean hasAllCodes(String s, int k) {

        int n = s.length();

        if (n < k) {

            return false;

        }

        int mask = (1 << k) - 1;

        Set<Integer> seen = new HashSet<>();

        int num = 0;

        // Initialize the first window

        for (int i = 0; i < k; i++) {

            num = (num << 1) | (s.charAt(i) - '0');

        }

        seen.add(num);

        // Slide the window across the string

        for (int i = k; i < n; i++) {

            num = ((num << 1) | (s.charAt(i) - '0')) & mask;

            seen.add(num);

            // Early exit if all possible codes are found

            if (seen.size() == (1 << k)) {

                return true;

            }

        }

        return seen.size() == (1 << k);

    }

}

Time Complexity : O(N), where N is the size of string and k is length of binary code.

Space Complexity : O(2^k)

**Refer to**

<https://leetcode.com/problems/check-if-a-string-contains-all-binary-codes-of-size-k/solutions/1105885/check-if-a-string-contains-all-binary-codes-of-size-k-short-easy-w-explanation/>

***1. Brute Force Solution (TLE)***

By looking at the problem description, one way we can think of solving the problem is by finding all *binaryCodes* of length k and checking if it exists in s. But this solutions isn't efficient and times out.

bool hasAllCodes(string s, int k) {

int n = s.size();

for(int num = 0; num <= (1 << k); num++){

string binaryCode = bitset<20>(num).to\_string().substr(20 - k, k);

if(s.find(binaryCode) == string::npos) return false;

}

return true;

}

**Time Complexity :** **O(N\*2^k)**, there are 2^k binary codes possible and for each code, we need N iteration in worst case to find the binary code in the string

***2. HashSet Solution***

We can observe that we only need to verify if all possible codes of length k exists in the string s. Instead of generating all the codes and then checking if each one exists in s, we can just **check all the substrings of length k in the string s**. If number of all the **distinct substrings of length k is 2^k**, we are sure that all binary codes exists in the given string. We can use a hashset to keep track of all distinct substrings found.

bool hasAllCodes(string s, int k) {

int n = s.size();

unordered\_set<string> set;

for(int i = 0; i <= n - k; i++) set.insert(s.substr(i, k));

// if number of distinct substrings in s equals 2^k (1 << k), return true

return set.size() == (1 << k);

}

**Time Complexity :** **O(N\*k)**, where N is the size of string and k is length of binary code.

**Space Complexity :** **O(N\*k)**

***3. Custom Hashset Solution***

There is a high constant time factor for calculating hash for a substring every time in the previous approach. Moreover, we are storing all strings and hence the space required is also higher. We have a much better way - calculating hash for each code by converting to its decimal equivalent and using it as hash index-

bool hasAllCodes(string s, int k) {

int n = s.size(), sizeNeeded = 1 << k, count = 0;

bool set[sizeNeeded]; memset(set, false, sizeof(set)); // initialising set of 'sizeNeeded' elements with false

for(int i = 0; i <= n - k; i++){

int hash = 0, shift = k - 1;

for(int j = i; j < i + k; j++)

hash += (s[j] - '0') << shift--; // hash will be equal to decimal value of substring

if(!set[hash])

set[hash] = true, count++;

}

return count == sizeNeeded;

}

**Time Complexity :** **O(N\*k)**, where N is the size of string and k is length of binary code.

**Space Complexity :** **O(2^k)**

***4. Optimized Custom Hashset Solution (Using rolling hash)***

We don't even need to calculate the full has for each of the substring of length k as we were doing so in the last approach. We can just calculate hash for first substring of length k and thereafter, we can apply ***rolling hash technique***.

This approach is nothing but removing the last character of previous substring and adding the next character from s and calculating the hash from this instead of recalculating the whole hash.

hash = hash << 1 : one shift to left

hash &= allOnes : removes kth bit, ensures our hash always less than 2^k

hash |= s[i] - '0' : calculates hash with new character after removing first character of previous substring

Eg. Suppose string is 110101 and k = 3. We start with hash = 0. Here size = 2^k = 8 and allOnes = 7 (111).

1. hash = 000 << 1 = 000 || hash &= (111) = 000(0) || hash |= 1 = 001 👉 hash = 001 (1)

2. hash = 001 << 1 = 010 || hash &= (111) = 010(2) || hash |= 1 = 011 👉 hash = 011 (3)

3. hash = 011 << 1 = 110 || hash &= (111) = 110(6) || hash |= 0 = 110 👉 hash = 110 (6)

4. hash = 110 << 1 = 1100 || hash &= (111) = 100(4) || hash |= 1 = 101 👉 hash = 101 (5)

^exceeds (2^k) ^2nd step removes kth bit^

5. hash = 101 << 1 = 1010 || hash &= (111) = 010(2) || hash |= 0 = 010 👉 hash = 010 (2)

6. hash = 010 << 1 = 100 || hash &= (111) = 100(4) || hash |= 1 = 101 👉 hash = 101 (5)

We can observe that using rolling hash, after first k-2 steps, our hash value is always equal

to the decimal equivalent of substring of length k. This can be used as for indexing the set which

will store the distinct substrings of length k found. If number of distinct substrings is equal to size,

then the answer will be true.

**Code**

bool hasAllCodes(string s, int k) {

int n = s.size(), size = 1 << k, count = 0, hash = 0, allOnes = ~size;

bool set[size]; memset(set, false, sizeof(set));

for(int i = 0; i < n; i++){

// below line combines all above-metioned steps for rolling hash calculation

hash = ((hash << 1) & allOnes) | (s[i] - '0');

if(i >= k - 1 && !set[hash]) // first k - 1 iteration are to calculate hash of first substring of length k

set[hash] = true, count++;

}

return count == size;

}

**Time Complexity :** **O(N)**, where N is the size of string and k is length of binary code.

**Space Complexity :** **O(2^k)**

**Refer to Deepseek**

**1. Brute Force Solution**

The provided C++ code checks if a string s contains all binary codes of size k. It does this by generating all possible binary codes of length k and checking if each one exists in the string s. To convert this code to Java, we need to handle the following:

Binary Code Generation: In C++, bitset<20>(num).to\_string().substr(20 - k, k) is used to generate a binary string of length k. In Java, we can use Integer.toBinaryString(num) and pad it with leading zeros to ensure it has length k.

String Search: In C++, s.find(binaryCode) checks if a substring exists. In Java, we can use s.contains(binaryCode).

Here’s the equivalent Java code:

class Solution {

public boolean hasAllCodes(String s, int k) {

int n = s.length();

// Total number of unique binary codes of length k

int totalCodes = 1 << k; // Equivalent to 2^k

// Iterate through all possible binary codes of length k

for (int num = 0; num < totalCodes; num++) {

// Generate the binary code for the current number

String binaryCode = Integer.toBinaryString(num);

// Pad with leading zeros to ensure the length is k

while (binaryCode.length() < k) {

binaryCode = "0" + binaryCode;

}

// Check if the binary code exists in the string s

if (!s.contains(binaryCode)) {

return false;

}

}

return true;

}

}

**Explanation**

Total Codes: The total number of unique binary codes of length k is 2^k, calculated using 1 << k.

Binary Code Generation: For each number num from 0 to 2^k - 1, we generate its binary representation using Integer.toBinaryString(num). If the binary string is shorter than k, we pad it with leading zeros.

Substring Check: We use s.contains(binaryCode) to check if the generated binary code exists in the string s. If any binary code is missing, we return false.

Final Result: If all binary codes are found in the string, we return true.

**2. Optimized HashSet Approach**

The above solution works but is inefficient for larger values of k (e.g., k = 20) because it generates and checks all 2^k binary codes. A more efficient approach is to use a sliding window and a hash set to track all unique substrings of length k in the string s. If the number of unique substrings equals 2^k, then the string contains all binary codes of length k.

Here’s the optimized Java solution:

class Solution {

public boolean hasAllCodes(String s, int k) {

int n = s.length();

// Total number of unique binary codes of length k

int totalCodes = 1 << k; // Equivalent to 2^k

// If the string is too short to contain all codes, return false

if (n < k) {

return false;

}

// Use a set to store all unique substrings of length k

Set<String> seen = new HashSet<>();

// Slide a window of size k across the string

for (int i = 0; i <= n - k; i++) {

String substring = s.substring(i, i + k);

seen.add(substring);

// Early exit if all codes are found

if (seen.size() == totalCodes) {

return true;

}

}

// Check if all codes were found

return seen.size() == totalCodes;

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Explanation of Optimized HashSet Approach**

Early Exit: If the string length n is less than k, return false immediately.

Sliding Window: Use a sliding window of size k to extract all substrings of length k from the string.

Hash Set: Store each substring in a hash set to track unique substrings.

Early Exit: If the size of the set reaches 2^k, return true immediately.

Final Check: After processing all substrings, check if the set size equals 2^k.

This optimized approach has a time complexity of O(n), where n is the length of the string, making it much more efficient for larger inputs.

**3. Rolling Hash Solution**

Okay, let's see. The problem is to check if a given binary string contains all possible binary codes of length k. Hmm, so for example, if k is 2, then all possible codes are 00, 01, 10, 11. We need to make sure that each of these appears as a substring in the given string.

First, I need to think about how to approach this efficiently. The brute force way would be to generate all possible binary codes of size k and check each one in the string. But that's probably not efficient for larger k. Because generating all possible codes would be 2^k possibilities, which is manageable for small k, but if k is up to 20, then 2^20 is over a million, which is about a million checks. But even that might be okay, but checking each one in the string could be time-consuming, especially if the string is long. So maybe there's a smarter way.

Wait, maybe we can collect all the substrings of length k in the given string and check if the total number is 2^k. But that's not exactly right. Because even if there are 2^k unique substrings, they might not cover all possibilities. Wait, no. If the number of unique substrings of length k is exactly 2^k, then that would mean all possible codes are present, since there are 2^k possible binary codes. So, the problem reduces to checking whether the number of unique substrings of length k in the given string is exactly 2^k.

So the approach could be:

1. Generate all possible substrings of length k in the string.

2. Store them in a hash set.

3. After collecting all of them, check if the size of the set is equal to 2^k.

But wait, how efficient is this? For a string of length n, there are (n - k + 1) substrings of length k. So if the string is of length 1e5, and k is 20, then n -k +1 is like 99981. So adding each substring to a hash set would be O(n) time, which is acceptable. But each substring is of length k, so creating each substring is O(k) time. So the total time is O(nk), which for n up to say 5e5 and k up to 20, that's 1e7 operations, which should be manageable.

But maybe there's a way to optimize this. For example, using a sliding window approach and hashing each substring in O(1) time per step. Like, using a rolling hash. But in Java, strings are handled as objects, so each substring would create a new object, which could be memory-intensive. However, for k up to 20, each substring is a short string, so maybe it's manageable.

Alternatively, we can convert each substring into an integer and store the integers in a set. Since each substring is a binary string, converting it to an integer would be possible as long as k is up to 30 or 31 (since in Java, integers are 32-bit). For example, if k is 20, the maximum value is 2^20-1 which is around a million, which is within the integer limit. So for k up to 30, this should work. However, the problem states that k can be up to 20, which is within the integer's capacity. Wait, 2^20 is 1,048,576, which is less than 2^31-1. So yes, using integers would be feasible.

So the steps would be:

Loop through the string, for each position i from 0 to (length - k), extract the substring s[i..i+k-1], convert it to an integer, add to a set. Then check if the size of the set is 2^k.

But how to convert the binary string to an integer efficiently?

Well, for each character in the substring, we can build the integer by shifting left and adding the current bit (0 or 1). For example, for each character in the substring:

num = num << 1 | (current\_char - '0')

But this would take O(k) time per substring, which is the same as before. But perhaps for each position, we can compute the number incrementally using a sliding window approach.

Wait, here's an idea: as we slide the window by one character to the right, the new number is (old\_num << 1) | (new\_char) & mask, where mask is (1 << k) -1. This way, each step is O(1) time, and we don't have to recompute the entire number each time. But this requires building the first number, then updating for each next character.

Let's see. For example, for the initial window of length k, we compute its integer value. Then, for each next window, subtract the leftmost bit's contribution (which would be the bit that is now out of the window), shift left, add the new bit. Wait, but in binary, the leftmost bit is the highest bit. For example, if k is 3 and the window is "101", then the number is 5 (binary 101). When we slide to the right, the new window is "010" (assuming next character is '0'). The leftmost '1' (value 4) is removed, then shift left by 1 (so 5 - 4 = 1, shift gives 2), add new bit 0: 2 +0 =2. But the new window is "010", which is 2. So that works.

Wait, but perhaps the mask is needed to keep the number within k bits. Because when shifting left, the bits may overflow. For example, if the window is "111" (7) and we shift left, add 1, the new number is 15. But with k=3, the new window is "111" again. Wait, maybe not. Let me think. The mask is (1 << k) -1. So when we compute the next number, we do (current << 1 | new\_bit) & mask. So for example, for k=3, mask is 0b111. So even if shifting left and adding a new bit, we mask out the higher bits beyond k. So this approach would correctly compute the integer for each window.

So here's the algorithm:

Initialize a set to store the integers.

Compute the initial window's integer value:

num = 0;

for the first k characters:

num = (num << 1) | (s.charAt(i) - '0');

Add this num to the set.

Then, for each subsequent window (from i = k to s.length() -1):

remove the leftmost bit (which is (s.charAt(i -k) - '0') ) \* (1 << (k-1)) ), shift left, add the new bit.

Wait, no. Let me think again. Let's see:

The current window is s[i-k+1 ... i]. When we move to the next character, the window is s[i-k+2 ... i+1].

The previous number is num. The leftmost bit is (num >> (k-1)) & 1. But when we shift left, that leftmost bit is multiplied by 2^(k-1), which when shifted left, becomes 2^k. But since the window is moving, we need to remove that leftmost bit, shift the remaining bits, then add the new bit.

Wait, perhaps:

**When the window moves right by one character, the new number is (num << 1) | (new\_char) & mask. Because the mask is (1 << k) -1, which ensures that we only keep the last k bits.**

Wait, let's test this.

For example:

Original window: 101 (k=3). num is 5. mask is 7 (0b111).

Next character is 0.

new\_num = (5 << 1) | 0 → 10. 10 & 7 is 2 (0b010), which is correct.

Another example:

Original window: 111 (7). Next character is 1.

new\_num = (7 <<1 |1) → 15. 15 &7 =7. So the new window is 111, but the actual next window is the last three 1's again.

Wait, but in this case, the original window is 111 (positions 0-2). Next window is positions 1-3. If the next character (position3) is 1, then the new window is 111 again. So the new\_num is 7, which is correct.

So this approach works. So the formula is:

new\_num = ((num << 1) | (next\_char)) & mask.

So for each new character after the initial window, we compute the new\_num using this formula.

So the steps are:

- Check if the length of s is less than k. If so, return false.

- Compute mask = (1 << k) -1.

- Initialize num to 0.

- Precompute the first window:

for i from 0 to k-1:

num = (num <<1 ) | (s.charAt(i) - '0')

add to set.

Then for i from k to s.length() -1:

num = ((num <<1) | (s.charAt(i) - '0')) & mask

add to set.

At the end, check if the size of the set is 2^k.

Yes, that's O(n) time, since each step is O(1), and no O(k) per substring. So this is better than the initial approach.

So in Java code:

public boolean hasAllCodes(String s, int k) {

if (s.length() < k) return false;

int mask = (1 << k) -1;

Set<Integer> seen = new HashSet<>();

int num =0;

for (int i=0; i < k; i++) {

num = (num <<1) | (s.charAt(i) - '0');

}

seen.add(num);

for (int i=k; i < s.length(); i++) {

num = ((num <<1) | (s.charAt(i) - '0')) & mask;

seen.add(num);

// early exit if possible

if (seen.size() == (1 <<k)) return true;

}

return seen.size() == (1 <<k);

}

Wait, but in the loop after adding the initial num, we process each next character, compute the new num, add to the set. Then, after each addition, check if the size is already 2^k. If so, return early.

This would optimize in cases where the set reaches the required size before processing all possible substrings.

But in the worst case, it's O(n) time, which is acceptable.

But what about when k is 0? Well, the problem says 1 <= k <= 20, according to the problem statement? Let me check the problem statement.

Problem says 1 <= k <= 20. So no need to handle k=0.

So this code should handle all cases where s.length() >=k.

But what if the string is exactly length k? Then, the first loop adds the initial num, and then the second loop doesn't run. Then the size is 1. So the code checks if 1 == 2^k. Which is only possible if k=0, but k is >=1. So in that case, returns false, which is correct.

Another test case: s = "00110110", k=2.

All possible 2-bit codes are 00,01,10,11, which are present in "00", "01", "11", "10", "01", "10", "00", "11", "10" (substrings of length 2). Wait, let's see:

The substrings are:

0: 00 → 0

1: 01 →1

2: 11 →3

3: 10 →2

4: 01 →1

5: 10 →2

6: 00 →0

7: 01 →1

Wait, but the example from the problem says that this s should return true for k=2. But according to the code, the set would contain 0,1,3,2. Which are 00,01,11,10. So the size is 4, which is 2^2=4. So returns true. So code works.

Another example: s is "0000", k=2. The substrings are 00,00,00. The set size is 1. So returns false. Correct.

So this approach should work.

Now, the code uses integers. But for k=20, the maximum possible number is 2^20-1, which is 1,048,575. Which is within the range of integers (Java's int is 32 bits). So storing these in a set as integers is feasible.

But wait, what about for k=20? (1 <<20) is 1,048,576. So mask is (1 <<20) -1, which is 0xFFFFF (1,048,575). So the code will generate numbers from 0 to 1,048,575, which can be stored in an integer. So the code is correct.

Thus, this approach should work.

To solve this problem, we need to determine if a given binary string contains all possible binary codes of length k. The solution involves efficiently checking all substrings of length k in the given string and verifying if they cover all possible combinations.

**Approach**

Check String Length: If the length of the string is less than k, it's impossible to have all binary codes of length k, so return false.

Use a Rolling Hash Technique: Convert each substring of length k into an integer using bit manipulation. This allows us to efficiently slide the window across the string and compute the integer value of each new substring in constant time.

Track Unique Substrings: Use a hash set to store the integer values of all unique substrings of length k.

Check Set Size: If the size of the hash set equals 2^k, then all possible binary codes of length k are present in the string.

**Solution Code**

import java.util.HashSet;

import java.util.Set;

public class Solution {

public boolean hasAllCodes(String s, int k) {

int n = s.length();

if (n < k) {

return false;

}

int mask = (1 << k) - 1;

Set<Integer> seen = new HashSet<>();

int num = 0;

// Initialize the first window

for (int i = 0; i < k; i++) {

num = (num << 1) | (s.charAt(i) - '0');

}

seen.add(num);

// Slide the window across the string

for (int i = k; i < n; i++) {

num = ((num << 1) | (s.charAt(i) - '0')) & mask;

seen.add(num);

// Early exit if all possible codes are found

if (seen.size() == (1 << k)) {

return true;

}

}

return seen.size() == (1 << k);

}

}

**Explanation**

Initial Check: If the string length is less than k, immediately return false since it's impossible to have all binary codes of length k.

Bit Masking: Use a mask (1 << k) - 1 to ensure we only consider the last k bits of the computed integer, which helps in efficiently sliding the window.

Rolling Hash: Compute the integer value of the first window of length k. For subsequent windows, update the integer by shifting left, adding the new bit, and applying the mask to keep only the relevant bits.

Tracking Uniques: Use a set to track all unique integers derived from the substrings. If the set size reaches 2^k, all possible binary codes are present, and we return true immediately for efficiency.

Final Check: After processing all windows, check if the set size matches 2^k to confirm the presence of all binary codes.

This approach ensures we efficiently check each possible substring in linear time, making the solution optimal even for larger values of k up to 20.