<https://leetcode.ca/all/1698.html>

Given a string s, return the number of **distinct** **substrings** of s.

A **substring** of a string is obtained by deleting any number of characters (possibly zero) from the front of the string and any number (possibly zero) from the back of the string.

**Example 1:**

**Input**: s = "aabbaba"

**Output**: 21

**Explanation**:

The set of distinct strings is ["a","b","aa","bb","ab","ba","aab","abb","bab","bba","aba","aabb","abba","bbab","baba","aabba","abbab","bbaba","aabbab","abbaba","aabbaba"]

**Example 2:**

**Input**: s = "abcdefg"

**Output**: 28

**Constraints:**

1 <= s.length <= 500

s consists of lowercase English letters.

**Follow up:**

Can you solve this problem in O(n) time complexity?

**Attempt 1: 2024-06-23**

**Solution 1: DFS (10 min, TLE)**

**Style 1:**

class Solution {

    long count = 0;

    public long countDistinct(String s) {

        helper(s, 0, "");

        return count;

    }

    private void helper(String str, int start, String current) {

        if (start == str.length()) {

            return;

        }

        for (int end = start + 1; end <= str.length(); end++) {

            current = str.substring(start, end);

            count += (long)countUniqueChars(current);

        }

        helper(str, start + 1, current);

    }

    public int countUniqueChars(String str) {

        Map<Character, Integer> charCountMap = new HashMap<>();

        for (char c : str.toCharArray()) {

            charCountMap.put(c, charCountMap.getOrDefault(c, 0) + 1);

        }

        return charCountMap.size();

    }

}

**Style 2:**

class Solution {

    public int countDistinct(String s) {

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

        for (int i = 0; i < s.length(); i++) {

            helper(s, i, substrings);

        }

        return substrings.size();

    }

    private void helper(String s, int start, Set<String> substrings) {

        if (start == s.length()) {

            return;

        }

        StringBuilder currentSubstring = new StringBuilder();

        for (int end = start; end < s.length(); end++) {

            currentSubstring.append(s.charAt(end));

            substrings.add(currentSubstring.toString());

        }

        helper(s, start + 1, substrings);

    }

}

**Solution 2: Rolling Harsh (360 min)**

**Below style is exactly same as** [L1044.Longest Duplicate Substring (Ref.L214)](note://72463BD488CE4A2294475F16F2F46FF3)

import java.util.HashSet;

import java.util.Set;

public class Solution {

    private static final int base = 256;

    private static final int mod = (int)1e9 + 7;

    public int countDistinctSubstrings(String s) {

        Set<Long> hashes = new HashSet<>();

        int n = s.length();

        for (int pattern\_len = 1; pattern\_len <= n; pattern\_len++) {

            // 'pattern\_hash' is the initial hash for substring as s[0, pattern\_len - 1]

            long pattern\_hash = 0;

            // 'multiplier' used for rolling hash

            long multiplier = 1;

            // Calculate 'multiplier' as base^(pattern\_len - 1)

            for(int i = 0; i < pattern\_len - 1; i++) {

                multiplier = (multiplier \* base) % mod;

            }

            // Calculate 'pattern\_hash'

// Hash(t[0, m - 1]) = t[0] \* b^(m - 1) + t[1] \* b^(m - 2) + ... + t[m - 1] \* b^0, t[0]代表该字符的ASCII码

            for (int i = 0; i < pattern\_len; i++) {

                pattern\_hash = (pattern\_hash \* base + s.charAt(i)) % mod;

            }

            hashes.add(pattern\_hash);

            // Try all other substrings with rolling hash

            int from = 0;

            int to = pattern\_len;

            while(to < n) {

                // Update pattern\_hash with rolling harsh

// Hash(t[1, m]) = (Hash(t[0, m - 1]) - t[0] \* b^(m - 1)) \* b + t[m] \* b^0

                pattern\_hash = (base \* (pattern\_hash - s.charAt(from++) \* multiplier) + s.charAt(to++)) % mod;

                if(pattern\_hash < 0) {

                    pattern\_hash += mod;

                }

                hashes.add(pattern\_hash);

            }

        }

        return hashes.size();

    }

    public static void main(String[] args) {

        Solution solution = new Solution();

        String s1 = "aabbaba";

        String s2 = "abcdefg";

        System.out.println("Distinct substrings in \"" + s1 + "\": " + solution.countDistinctSubstrings(s1)); // Output: 21

        System.out.println("Distinct substrings in \"" + s2 + "\": " + solution.countDistinctSubstrings(s2)); // Output: 28

    }

}

**Refer to ChatGPT**

**Explanation**

**Constants:**

**BASE** is the base value for the rolling hash.

**MODULUS** is the modulus value to prevent overflow and reduce hash collisions.

**countDistinctSubstrings Method:**

**Set to Track Hashes:** A Set<Long> is used to store unique hash values of substrings.

**Nested Loops for Substrings:**

The outer loop iterates over the possible lengths of substrings.

The inner loop computes the rolling hash for substrings of the current length.

**Initial Hash Calculation:**

For the first substring of each length, calculate the hash value directly.

Track the power of BASE for efficient removal of the leading character in the rolling hash.

**Rolling Hash Calculation:**

Update the hash value by removing the hash contribution of the leading character and adding the hash contribution of the trailing character.

Ensure that the hash value remains positive by adding MODULUS before taking the modulo.

**Store Hashes:** Add each computed hash to the set.

**Main Method:**

Demonstrates the usage of countDistinctSubstrings with sample strings and prints the results.

**Key Points**

**Efficiency:** The rolling hash allows efficient computation of hash values for all substrings of each length, reducing the time complexity compared to a naive approach.

**Handling Collisions:** Using a set ensures that only unique substrings are counted.

**Modulus Operation:** The modulus operation helps in avoiding overflow and managing hash collisions effectively.

This solution leverages the rolling hash technique to efficiently solve the problem of counting distinct substrings in a given string.

import java.util.HashSet;

import java.util.Set;

public class Solution {

private static final int BASE = 256;

private static final int MODULUS = 1000000007;

public int countDistinctSubstrings(String s) {

Set<Long> hashes = new HashSet<>();

int n = s.length();

for (int length = 1; length <= n; length++) {

long hash = 0;

long power = 1;

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

hash = (hash \* BASE + s.charAt(i)) % MODULUS;

if (i > 0) {

power = (power \* BASE) % MODULUS;

}

}

hashes.add(hash);

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

hash = (hash \* BASE + s.charAt(i)) % MODULUS;

hash = (hash - s.charAt(i - length) \* power % MODULUS + MODULUS) % MODULUS;

hashes.add(hash);

}

}

return hashes.size();

}

public static void main(String[] args) {

Solution solution = new Solution();

String s1 = "aabbaba";

String s2 = "abcdefg";

System.out.println("Distinct substrings in \"" + s1 + "\": " + solution.countDistinctSubstrings(s1)); // Output: 21

System.out.println("Distinct substrings in \"" + s2 + "\": " + solution.countDistinctSubstrings(s2)); // Output: 28

}

}

**Refer to**

<https://github.com/doocs/leetcode/blob/main/solution/1600-1699/1698.Number%20of%20Distinct%20Substrings%20in%20a%20String/README_EN.md>

String hashing is a method to map a string of any length to a non-negative integer, and the probability of collision is almost zero. String hashing is used to calculate the hash value of a string, which can quickly determine whether two strings are equal.

We take a fixed value BASE, treat the string as a number in BASE radix, and assign a value greater than 0 to represent each character. Generally, the values we assign are much smaller than BASE. For example, for a string composed of lowercase letters, we can set a=1, b=2, ..., z=26. We take a fixed value MOD, calculate the remainder of the BASE radix number to MOD, and use it as the hash value of the string.

Generally, we take BASE=131 or BASE=13331, at which point the probability of collision of the hash value is extremely low. As long as the hash values of two strings are the same, we consider the two strings to be equal. Usually, MOD is taken as 2^64. In C++, we can directly use the unsigned long long type to store this hash value. When calculating, we do not handle arithmetic overflow. When overflow occurs, it is equivalent to automatically taking the modulus of 2^64, which can avoid inefficient modulus operations.

Except for extremely specially constructed data, the above hash algorithm is unlikely to cause collisions. In general, the above hash algorithm can appear in the standard answer of the problem. We can also take some appropriate BASE and MOD values (such as large prime numbers), perform several groups of hash operations, and only consider the original strings equal when the results are all the same, making it even more difficult to construct data that causes this hash to produce errors.

The time complexity is O(N^2), and the space complexity is O(N^2). Here, n is the length of the string.

class Solution {

public int countDistinct(String s) {

int base = 131;

int n = s.length();

long[] p = new long[n + 10];

long[] h = new long[n + 10];

p[0] = 1;

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

p[i + 1] = p[i] \* base;

h[i + 1] = h[i] \* base + s.charAt(i);

}

Set<Long> ss = new HashSet<>();

for (int i = 1; i <= n; ++i) {

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

long t = h[j] - h[i - 1] \* p[j - i + 1];

ss.add(t);

}

}

return ss.size();

}

}

**Refer to**

[L2262.Total Appeal of A String (Ref.L828)](note://WEBb28ca12f4f1b056a97a22c14bef90a52)

[L828.Count Unique Characters of All Substrings of a Given String (Ref.L2262](note://WEBf394faa89bf4c60aa6f2c29c102ef307)

[L214.Shortest Palindrome (Ref.L1044,L1698)](note://C98FF24B295744F888F2B8C18AD71093)

[L1044.Longest Duplicate Substring (Ref.L214)](note://72463BD488CE4A2294475F16F2F46FF3)

[Rabin-Karp Algorithm](note://1DFFF586710745318EE15736B2372884)

[Rabin-Karp 指纹字符串查找算法](note://49C117E848D24EB4A587297908FCDF55)