<https://leetcode.com/problems/count-unique-characters-of-all-substrings-of-a-given-string/description/>

Let's define a function countUniqueChars(s) that returns the number of unique characters in s.

For example, calling countUniqueChars(s) if s = "LEETCODE" then "L", "T", "C", "O", "D" are the unique characters since they appear only once in s, therefore countUniqueChars(s) = 5.

Given a string s, return the sum of countUniqueChars(t) where t is a substring of s. The test cases are generated such that the answer fits in a 32-bit integer.

Notice that some substrings can be repeated so in this case you have to count the repeated ones too.

**Example 1:**

**Input:** s = "ABC"

**Output:** 10

**Explanation:** All possible substrings are: "A","B","C","AB","BC" and "ABC".Every substring is composed with only unique letters.Sum of lengths of all substring is 1 + 1 + 1 + 2 + 2 + 3 = 10

**Example 2:**

**Input:** s = "ABA"

**Output:** 8

**Explanation:** The same as example 1, except countUniqueChars("ABA") = 1.

**Example 3:**

**Input:** s = "LEETCODE"

**Output:** 92

**Constraints:**

1 <= s.length <= 10^5

s consists of uppercase English letters only.

**Attempt 1: 2024-06-17**

**Solution 1: DFS (180 min, TLE 32/76)**

**Based on find all substrings of a given string (no need to consider the duplicates, just need to find all substrings even they are same substrings, e.g "LEETCODE" as given string, the "E" on index = 1 or 2 are identical substrings, but they are valid substrings, no need duplicate check)**

class Solution {

    int count = 0;

    public int uniqueLetterString(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 += 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);

        }

        int uniqueCount = 0;

        for (int count : charCountMap.values()) {

            if (count == 1) {

                uniqueCount++;

            }

        }

        return uniqueCount;

    }

}

Time Complexity: O(n^3)

Space Complexity: O(n)

**Time Complexity**

Recursive Calls:

The helper function iterates over all possible substrings of the string. For each starting index start, it considers all substrings starting from start.

The total number of substrings in a string of length nnn is O(n^2), as there are n+(n−1)+⋯+1=n(n+1) substrings.

String Substring Creation:

For each substring current, the substring() operation in Java takes O(k) time, where k is the length of the substring.

Counting Unique Characters:

The countUniqueChars method iterates over each substring and uses a hashmap to count character frequencies. This operation takes O(k), where k is the length of the substring.

Combining these factors:

There are O(n^2) substrings in total.

The average substring length is O(n/2), leading to O(n^2⋅n)=O(n^3) complexity for substring creation and unique character counting.

Overall **time complexity**: **O(n^3)**.

**Space Complexity**

Substring Storage:

The current string is passed recursively and reconstructed in each iteration. While strings are immutable, the temporary storage contributes to the memory usage. On average, this takes O(n) for the longest substring.

HashMap:

The countUniqueChars function uses a HashMap to store character frequencies. In the worst case, the hashmap size is proportional to the number of distinct characters in the substring, i.e., O(k), where k≤26 (for lowercase alphabets).

Recursive Stack:

The recursion depth is O(n) because of the helper function.

Overall **space complexity**: **O(n+k)) ≈ O(n)**.

**Solution 2: DP (720 min)**

**Style 1:**

class Solution {

    public int uniqueLetterString(String s) {

        int MOD = (int)1e9 + 7;

        int[][] charRecentTwoIndexes = new int[26][2];

        for(int i = 0; i < 26; i++) {

            Arrays.fill(charRecentTwoIndexes[i], -1);

        }

        int result = 0;

        int n = s.length();

        // 每个字母出现的位置就很重要了，由于上面的分析说了，只要知道三个位置，就可以

        // 求出中间的字母的贡献值，为了节省空间，只保留每个字母最近两次的出现位置，

        // 这样加上当前位置i，就可以知道前一个字母的贡献值了。这里使用一个长度为 26x2

        // 的二维数组 idx，因为题目中限定了只有26个大写字母。这里只保留每个字母的前两个

        // 出现位置，均初始化为 -1。然后遍历S中每个字母，对于每个字符减去A，就是其对应

        // 位置，此时将该字母上一次出现的位置(前一个位置)的贡献值累加到结果 res 中，

// 假如当前字母是首次出现，也不用担心，因为在设定中该字母出现的前两个位置都是 -1，

// 相减后为0，所以累加值还是0。然后再更新 idx 数组的值

        for(int i = 0; i < n; i++) {

            int index = s.charAt(i) - 'A';

            result = (result + (i - charRecentTwoIndexes[index][1]) \* (charRecentTwoIndexes[index][1] - charRecentTwoIndexes[index][0]) % MOD) % MOD;

            charRecentTwoIndexes[index][0] = charRecentTwoIndexes[index][1];

            charRecentTwoIndexes[index][1] = i;

        }

        // 由于每次都是计算该字母之前一个位置的贡献值[e.g 假设字母A出现在位置{3,6,9}，

// 第一次实际计算的是3之前的虚拟位置-1的贡献值，包括3在内算上之前最近的2个位置

// 的组合是{-1,-1,3}，我们计算的就是中间的-1的贡献值，也就是A在当前位置3之前

// 出现过的位置-1的贡献值，以此类推后续的组合依次为{-1,3,6}实际计算的是3的贡献

// 值，{3,6,9}实际计算的是6的贡献值，此时此刻 for loop 已经循环了3次了，然而

// 针对字母A我们并没有计算其在坐标9的贡献值，需要加一轮{6,9,N}由此得出，而且加

// 一轮必须是针对所有字符(字符串中出现过的所有字符)而不仅仅是这里举例的A]，所以

// 最后还需要一个 for 循环去计算每个字母最后出现位置的贡献值，此时由于身后没有

// 该字母了，就用位置N来代替即可

        for(int i = 0; i < 26; i++) {

            result = (result + (n - charRecentTwoIndexes[i][1]) \* (charRecentTwoIndexes[i][1] - charRecentTwoIndexes[i][0]) % MOD) % MOD;

        }

        return result;

    }

}

Time complexity: O(N)

Space complexity: O(1)

**Refer to**

<https://leetcode.com/problems/count-unique-characters-of-all-substrings-of-a-given-string/solutions/128952/java-c-python-one-pass-o-n/>

**Foreword**

Quite similar to the question in 2262. Total Appeal of A String,

which I think slightly simpler than this question.

I also suggest you practicing that first.

I have the similar intuition and approach for both questions.

**Intuition**

Let's think about how a character can be found as a unique character.

Think about string "XAXAXXAX" and focus on making the second "A" a unique character.

We can take "XA(XAXX)AX" and between "()" is our substring.

We can see here, to make the second "A" counted as a uniq character, we need to:

insert "(" somewhere between the first and second A

insert ")" somewhere between the second and third A

For step 1 we have "A(XA" and "AX(A", 2 possibilities. (index = 3 - 1 = 2)

For step 2 we have "A)XXA", "AX)XA" and "AXX)A", 3 possibilities. (index = 6 - 3 = 3)

So there are in total 2 \* 3 = 6 ways to make the second A a unique character in a substring.

In other words, there are only 6 substring, in which this A contribute 1 point as unique string.

Instead of counting all unique characters and struggling with all possible substrings, we can count for every char in S, how many ways to be found as a unique char.

We count and sum, and it will be out answer.

**Explanation**

index[26][2] record last two occurrence index for every upper characters.

Initialise all values in index to -1.

Loop on string S, for every character c, update its last two occurrence index to index[c].

Count when loop. For example, if "A" appears twice at index 3, 6, 9 seperately (e.g S = "XXXAXXAXXAX"), we need to count:

Start with: (3-(-1)) \* (-1 - (-1)) = 4 \* 0 = 0

For the first "A" possibilities: (6-3) \* (3-(-1)) = 3 \* 4 = 12

For the second "A": (9-6) \* (6-3) = 3 \* 3 = 9

For the third "A": (N-9) \* (9-6) (N = S.length() = 10) = 1 \* 3 = 3

**Complexity**

One pass, time complexity O(N).

Space complexity O(1).

public int uniqueLetterString(String S) {

int[][] index = new int[26][2];

for (int i = 0; i < 26; ++i) Arrays.fill(index[i], -1);

int res = 0, N = S.length(), mod = (int)Math.pow(10, 9) + 7;

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

int c = S.charAt(i) - 'A';

res = (res + (i - index[c][1]) \* (index[c][1] - index[c][0]) % mod) % mod;

index[c] = new int[] {index[c][1], i};

}

for (int c = 0; c < 26; ++c)

res = (res + (N - index[c][1]) \* (index[c][1] - index[c][0]) % mod) % mod;

return res;

}

**Refer to**

<https://www.cnblogs.com/grandyang/p/11616485.html>

这道题给了我们一个字符串S，要统计其所有的子串中不同字符的个数之和，这里的子串是允许重复的，而且说结果需要对一个超大数取余，这暗示了返回值可能会很大，这样的话对于纯暴力的解法，比如遍历所有可能的子串并统计不同字符的个数的这种解法肯定是不行的。这道题还真是一点没有辱没其 Hard 标签，确实是一道很有难度的题，不太容易想出正确解法。还好有 [李哥 lee215 的帖子](<https://leetcode.com/problems/unique-letter-string/discuss/128952/One-pass-O(N)-Straight-Forward)>，一个帖子的点赞数超过了整个第一页所有其他帖子的点赞数之和，简直是刷题界的 Faker，你李哥永远是你李哥。这里就按照李哥的帖子来讲解吧，**首先来看一个字符串 CACACCAC，若想让第二个A成为子串中的唯一，那么必须要知道其前后两个相邻的A的位置，比如 CA(CACC)AC，括号中的子串 CACC 中A就是唯一的存在，同样，对于 CAC(AC)CAC，括号中的子串 AC 中A也是唯一的存在。这样就可以观察出来，只要左括号的位置在第一个A和第二个A之间（共有2个位置），右括号在第二个A和第三个A之间（共有3个位置），这样第二个A在6个子串中成为那个唯一的存在。换个角度来说，只有6个子串可以让第二个A作为单独的存在从而在结果中贡献。这是个很关键的转换思路，与其关注每个子串中的单独字符个数，不如换个角度，对于每个字符，统计其可以在多少个子串中成为单独的存在，同样可以得到正确的结果**。这样的话，每个字母出现的位置就很重要了，由于上面的分析说了，只要知道三个位置，就可以求出中间的字母的贡献值，**为了节省空间，只保留每个字母最近两次的出现位置**，这样加上当前位置i，就可以知道前一个字母的贡献值了。这里使用一个长度为 26x2 的二维数组 idx，因为题目中限定了只有26个大写字母。这里只保留每个字母的前两个出现位置，均初始化为 -1。然后遍历S中每个字母，对于每个字符减去A，就是其对应位置，此时将前一个字母的贡献值累加到结果 res 中，假如当前字母是首次出现，也不用担心，前两个字母的出现位置都是 -1，相减后为0，所以累加值还是0。然后再更新 idx 数组的值。由于每次都是计算该字母前一个位置的贡献值，所以最后还需要一个 for 循环去计算每个字母最后出现位置的贡献值，此时由于身后没有该字母了，就用位置N来代替即可，参见代码如下：

class Solution {

public:

int uniqueLetterString(string S) {

int res = 0, n = S.size(), M = 1e9 + 7;

vector<vector<int>> idx(26, vector<int>(2, -1));

// 每个字母出现的位置就很重要了，由于上面的分析说了，只要知道三个位置，就可以

// 求出中间的字母的贡献值，为了节省空间，只保留每个字母最近两次的出现位置，

// 这样加上当前位置i，就可以知道前一个字母的贡献值了。这里使用一个长度为 26x2

// 的二维数组 idx，因为题目中限定了只有26个大写字母。这里只保留每个字母的前两个

// 出现位置，均初始化为 -1。然后遍历S中每个字母，对于每个字符减去A，就是其对应

// 位置，此时将前一个字母的贡献值累加到结果 res 中，假如当前字母是首次出现，

// 也不用担心，前两个字母的出现位置都是 -1，相减后为0，所以累加值还是0。然后

// 再更新 idx 数组的值

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

int c = S[i] - 'A';

res = (res + (i - idx[c][1]) \* (idx[c][1] - idx[c][0]) % M) % M;

idx[c][0] = idx[c][1];

idx[c][1] = i;

}

// 由于每次都是计算该字母前一个位置的贡献值，所以最后还需要一个 for 循环去计算

// 每个字母最后出现位置的贡献值，此时由于身后没有该字母了，就用位置N来代替即可

for (int c = 0; c < 26; ++c) {

res = (res + (n - idx[c][1]) \* (idx[c][1] - idx[c][0]) % M) % M;

}

return res;

}

};

**Style 2:**

class Solution {

    public int uniqueLetterString(String S) {

        int res = 0, n = S.length(), cur = 0;

        int M = (int) 1e9 + 7;

        int[] lastSeenCharIndexes = new int[26];

        int[] secondLastSeenCharIndexes = new int[26];

        for (int i = 0; i < n; ++i) {

            int c = S.charAt(i) - 'A';

            cur = cur + 1 + i - 2 \* lastSeenCharIndexes[c] + secondLastSeenCharIndexes[c];

            res += cur;

            secondLastSeenCharIndexes[c] = lastSeenCharIndexes[c];

            lastSeenCharIndexes[c] = i + 1;

        }

        return res;

    }

}

Time Complexity: O(N)

Space Complexity: O(1)

**Refer to**

<https://www.cnblogs.com/grandyang/p/11616485.html>

**以下这种方法是在** [L2262.Total Appeal of A String (Ref.L828)](note://WEBb28ca12f4f1b056a97a22c14bef90a52) **基础上的改进，因为题目定义的计算子字符串中 unique char 的个数的定义改变了，附加条件是该 unique char 在子字符串中的频率必须为1**

我们也可以换一种解法，使得其更加简洁一些，思路稍微有些不同，这里参考了 [大神 meng789987 的帖子] (<https://leetcode.com/problems/count-unique-characters-of-all-substrings-of-a-given-string/solutions/158378/concise-dp-o-n-solution/>)。使用的是动态规划 Dynmaic Programming 的思想，用一个一维数组 dp，**其中 dp[i] 表示以 S[i] 为结尾的所有子串中的单独字母个数之和，这样只要把 [0, n-1] 范围内所有的 dp[i] 累加起来就是最终的结果了**。更新 dp[i] 的方法关键也是要看重复的位置，比如当前是 AB 的话，此时 dp[1]=3，因为以B结尾的子串是 B 和 AB，共有3个单独字母。若此时再后面加上个C的话，由于没有重复出现，则以C结尾的子串 C，BC，ABC 共有6个单独字母，即 dp[2]=6，怎么由 dp[1] 得到呢？首先新加的字母本身就是子串，所以一定是可以贡献1的，然后由于之前都没有C出现，则之前的每个子串中C都可以贡献1，而原本的A和B的贡献值也将保留，所以总共就是 dp[2] = 1+dp[1]+2 = 6。**但若新加的字母是A的话，就比较 tricky 了，首先A本身也是子串，有稳定的贡献1，由于之前已经有A的出现了，所以只要知道了之前A的位置，那么中间部分是没有A的，即子串 B 中没有A，A可以贡献1，但是对于之前的有A的子串，比如 AB，此时新加的A不但不能贡献，反而还会伤害之前A的贡献值，即变成 ABA 了后，不但第二个A不能贡献，连第一个A之前的贡献值也要减去，此时 dp[2] = 1+dp[1]+(2-1)-(1-0) = 4。其中2是当前A的位置，1是前一个A的位置(= 0)加1(0 + 1 = 1)，0是再前一个A的位置(= -1)加1(-1 + 1 = 0)。**讲到这里应该就比较清楚了吧，这里还是要知道每个字符的前两次出现的位置，这里用两个数组 first 和 second，**不过需要注意的是，这里保存的是位置加1**。又因为每个 dp 值只跟其前一个 dp 值有关，所以为了节省空间，并不需要一个 dp 数组，而是只用一个变量 cur 进行累加即可，记得每次循环都要把 cur 存入结果 res 中。那么每次 cur 的更新方法就是前一个 cur 值加上1，再加上当前字母产生的贡献值，减去当前字母抵消的贡献值，参见代码如下：

class Solution {

public:

int uniqueLetterString(string S) {

int res = 0, n = S.size(), cur = 0, M = 1e9 + 7;

vector<int> first(26), second(26);

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

int c = S[i] - 'A';

// cur = 1 + cur + (i - first[c]) - (first[c] - second[c]);

cur = cur + 1 + i - first[c] \* 2 + second[c];

res = (res + cur) % M;

second[c] = first[c];

first[c] = i + 1;

}

return res;

}

};

**Refer to**

**[Java] Clean O(n) Solution || detailed explanation of why it works**

<https://leetcode.com/problems/count-unique-characters-of-all-substrings-of-a-given-string/>

The Key of solving this problem in linear time is find the relationship between **sum of countUniqueChars() for substrings ending index i** and **sum of countUniqueChars() for substrings ending index i-1**. This article aims to leave no room for ambiguity​ of how this algorithm works.

For each index i, consider all substrings ending at index i:

s[0 : i] , ... , s[i-1 : i] , s[i : i];

Comparing with all substrings ending at index i-1:

s[0: i-1], ... , s[i-1, i-1]

We know the result for s[i : i] which only contains 1 character:

countUniqueChars(s[i: i]) = 1

Then we want to find the relationship between:

countUniqueChars(s[0: i]) v.s. countUniqueChars(s[0: i-1])

countUniqueChars(s[1: i]) v.s. countUniqueChars(s[1: i-1])

... ...

countUniqueChars(s[i-1: i]) v.s. countUniqueChars(s[i-1: i-1])

Denote c = s.charAt(i), then for all substrings ending at index i-1, there could be 3 cases:

// denote char c = s.charAt(i);

Case 1. s[k : i-1] contains 0 c ==> countUniqueChars(s[k : i]) = countUniqueChars(s[k : i-1]) + 1

Case 2. s[k : i-1] contains 1 c ==> countUniqueChars(s[k : i]) = countUniqueChars(s[k : i-1]) - 1

Case 3. s[k : i-1] contains at least 2 c ==> countUniqueChars(s[k : i]) = countUniqueChars(s[k : i-1])

The reasoning is pretty starightforward:

Case 1. s[k : i] has one more unique char (c itself) comparing to s[k : i-1]

Case 2. s[k : i] has one less unique char (c itself) comparing to s[k : i-1]

Case 3. any unique char in s[k : i-1] is still unique in s[k : i];

c itself is not an unique char for either s[k : i-1] or s[k : i]

Thus we find the relation between the **sum of countUniqueChars() for substrings ending index i** and **sum of countUniqueChars() for substrings ending index i-1**.

sum of countUniqueChars() for index i

= sum of countUniqueChars() for index i-1 + #(case 1) - #(case 2) + 1

where the last 1 stands for the substring s[i : i]. The remaining things are just calculating the number of substrings in Case 1 & Case 2. We denote:

// p : index of last appearance of c; or -1 if not exist (c never appears)

// q : index of second last appearance of c; or -1 if not exist (c never appears twice)

second last last s.charAt(i)

char C C C

|------------|------------|----------|-|-----

index 0 q p i

^

i-1

Then for s[k : i-1] whose starting point is k:

For k:

1. k in [p+1, i-1] <==> c appears 0 time in s[k : i-1]

也就是说只有当k在[p+1,i-1]的范围的时候才会满足 Case 1. s[k : i-1] contains 0 c 的情况，

一共有(i - 1) - (p + 1) + 1 = i - p - 1种情况

2. k in [q+1, p] <==> c appears 1 time in s[k : i-1]

也就是说只有当k在[q+1,p]的范围的时候才会满足 Case 2. s[k : i-1] contains 1 c 的情况，

一共有p - (q + 1) + 1 = p - 1种情况

3. k in [0, q] <==> c appears at least 2 times in s[k : i-1]

也就是说只有当k在[0,q]的范围的时候才会满足 Case 3. s[k : i-1] contains 2 c 的情况，

但是因为直接不是 unique character的情况，所以我们可以直接跳过这种情况

Thus we have:

#(Case 1) = (i-1) - (p+1) + 1

= i - p - 1

#(Case 2) = p - (q+1) + 1

= p - q

which leads to

sum of countUniqueChars() for index i

= sum of countUniqueChars() for index i-1 + (i - p - 1) - (p - q) + 1

= sum of countUniqueChars() for index i-1 + (i - p - p + q)

Notice we set the default value of p & q to -1 to handles the cases with invalid intervals. Now we finish the reasoning. Getting the solution is very straightforward.

class Solution {

public int uniqueLetterString(String s) {

int[] lastSeen = new int[26];

Arrays.fill(lastSeen, -1);

int[] secLastSeen = new int[26];

Arrays.fill(secLastSeen, -1);

int count = 0, res = 0;

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

int idx = s.charAt(i) - 'A';

int p = lastSeen[idx];

int q = secLastSeen[idx];

count += i - p - p + q;

res += count;

secLastSeen[idx] = lastSeen[idx];

lastSeen[idx] = i;

}

return res;

}

}

**Generate all the subsequences of a string using recursion**

**To generate all subsequences of a string using recursion, you need a recursive function that processes the string character by character, building up subsequences and adding them to a result set. Here’s a Java solution for generating all subsequences of a given string using recursion:**

import java.util.ArrayList;

import java.util.List;

public class SubstringGenerator {

// Helper method to start the recursive process

public static List<String> generateAllSubsequences(String str) {

List<String> result = new ArrayList<>();

generateSubsequencesRecursive(str, 0, "", result);

return result;

}

// Recursive method to generate subsequences

private static void generateSubsequencesRecursive(String str, int start, String current, List<String> result) {

if (start == str.length()) {

return;

}

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

current += str.charAt(i);

result.add(current);

generateSubsequencesRecursive(str, i + 1, current, result);

current = current.substring(0, current.length() - 1); // Backtrack

}

}

public static void main(String[] args) {

String str = "abc";

List<String> substrings = generateAllSubsequences(str);

for (String substring : substrings) {

System.out.println(substring);

}

}

}

**Explanation**

generateAllSubsequences Method:

This method initializes the recursive process. It creates a list result to store the substrings and calls generateSubsequencesRecursive to start the recursion.

generateSubsequencesRecursive Method:

This method is the core of the recursive solution. It takes the original string str, the current starting index start, the current subsequence being built current, and the list to store results result.

The base case checks if start is equal to the length of the string, in which case it returns since no more subsequences can be generated starting from this index.

The loop iterates over the remaining characters starting from start. For each character, it appends it to the current subsequence and adds this subsequence to the result list.

It then recursively calls itself with the next starting index i + 1.

After the recursive call, it backtracks by removing the last character from current to explore other potential subsequence.

Main Method:

The main method demonstrates how to use the generateAllSubsequences method. It generates all subsequences for the string "abc" and prints them.

**Benefits**

Comprehensive: This approach ensures that all possible subsequences are generated.

Recursive: Uses the power of recursion to build subsequences incrementally.

Backtracking: The backtracking step ensures that each character is considered in all possible positions.

**We can rewrite the generateSubsequencesRecursive method to use a style where each recursive call decides whether to include or exclude the current character, similar to the example you provided. Here is the updated solution:**

import java.util.ArrayList;

import java.util.List;

public class SubstringGenerator {

// Helper method to start the recursive process

public static List<String> generateAllSubsequences(String str) {

List<String> result = new ArrayList<>();

generateSubsequencesRecursive(str, 0, "", result);

return result;

}

// Recursive method to generate subsequences

private static void generateSubsequencesRecursive(String str, int start, String current, List<String> result) {

if (start == str.length()) {

return;

}

// Skip current character

generateSubsequencesRecursive(str, start + 1, current, result);

// Include current character

current += str.charAt(start);

result.add(current); // Add to result before making the recursive call

generateSubsequencesRecursive(str, start + 1, current, result);

}

public static void main(String[] args) {

String str = "abc";

List<String> substrings = generateAllSubsequences(str);

for (String substring : substrings) {

System.out.println(substring);

}

}

}

**Explanation**

generateAllSubsequences Method:

This method initializes the process by creating a list result to store the substrings and calls generateSubsequencesRecursive with the initial parameters.

generateSubsequencesRecursive Method:

This method generates subsequences recursively.

The base case checks if start has reached the length of the string. If so, it returns because no further subsequences can be generated.

The method makes two recursive calls:

Skip current character: This call continues with the next index without adding the current character to the current subsequence.

Include current character: This call adds the current character to the current subsequence, adds the new subsequence to the result list, and then continues with the next index.

By making these two recursive calls, every possible combination of including or excluding each character is explored, generating all subsequences.

Main Method:

The main method demonstrates how to use the generateAllSubsequences method. It generates all substrings for the string "abc" and prints them.

**Benefits**

Clarity: This approach clearly separates the decision to include or exclude each character, making the recursion logic easier to understand.

Comprehensiveness: Ensures that all possible subsequences are generated by considering every combination of including and excluding characters.

This solution efficiently generates all subsequences of the given string by leveraging recursive calls to explore all possible combinations of characters.

**They are both backtracking solution, just different style, one is for loop in recursion method, one is two different branches as skip or include current character, what are the difference between ?**

You are correct that both solutions are backtracking solutions, and they generate all subsequences of a string using recursion. However, they differ in style and the way they explore the solution space. Here are the differences between the two approaches:

**Approach 1: For Loop in Recursion Method**

private static void generateSubsequencesRecursive(String str, int start, String current, List<String> result) {

if (start == str.length()) {

return;

}

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

current += str.charAt(i);

result.add(current);

generateSubsequencesRecursive(str, i + 1, current, result);

current = current.substring(0, current.length() - 1); // Backtrack

}

}

**Characteristics:**

Looping Through Options: This method uses a for loop to iterate through each possible starting index for the next character to include in the subsequence. This means that for each recursive call, the method explores all possible continuations of the subsequence starting from the current index.

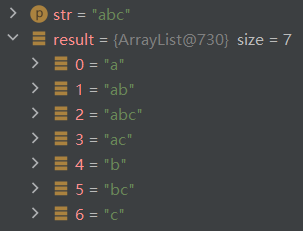
Explicit Backtracking: After each recursive call, the method explicitly backtracks by removing the last character from the current subsequence.

Subset Generation: This approach generates all subsequences that start from each character in the original string.

**The final generated all subsequences adding into result order for Approach 1 as below:**

**加入的顺序不同体现了两种思路的不同，都很重要，需要记忆两种思路的写法**

abc -> 1st add "a" -> 2nd add "ab" -> 3rd add "abc" -> 4th add "ac" -> 5th add "b" -> 6th add "bc" -> 7th add "c"



**Approach 2: Two Branches (Include or Skip)**

private static void generateSubsequencesRecursive(String str, int start, String current, List<String> result) {

if (start == str.length()) {

return;

}

// Skip current character

generateSubsequencesRecursive(str, start + 1, current, result);

// Include current character

current += str.charAt(start);

result.add(current); // Add to result before making the recursive call

generateSubsequencesRecursive(str, start + 1, current, result);

}

**Characteristics:**

Binary Decisions: This method uses a binary decision process to either include or skip the current character. For each character, it makes two recursive calls: one excluding the character and one including it.

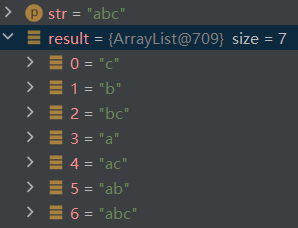
Implicit Backtracking: The method naturally backtracks by making two separate recursive calls without the need to explicitly modify the current subsequence after the recursive call.

Power Set Generation: This approach can be thought of as generating the power set of the string's characters, as it considers every combination of included and excluded characters.

**The final generated all subsequences adding into result order for Approach 2 as below:**

**加入的顺序不同体现了两种思路的不同，都很重要，需要记忆两种思路的写法**

abc -> 1st add "c" -> 2nd add "b" -> 3rd add "bc" -> 4th add "a" -> 5th add "ac" -> 6th add "ab" -> 7th add "abc"



**Approach 1 and 2 Differences**

Control Flow:

The for loop approach explicitly iterates through each possible continuation of the subsequence, making it more iterative in nature within each recursive call.

The binary decision approach recursively branches into two paths at each step, making it more declarative and simpler to understand the inclusion/exclusion logic.

Backtracking Mechanism:

The for loop approach requires explicit backtracking by modifying the current subsequence after each recursive call.

The binary decision approach inherently backtracks by making separate recursive calls for inclusion and exclusion, thus naturally restoring the state.

Output Order:

The for loop approach tends to generate subsequences in a lexicographical order starting from each character.

The binary decision approach generates subsequences in an order that reflects the power set generation process, potentially resulting in a different order of subsequences.

**Performance and Complexity**

Both approaches have similar time complexity in the worst case because they both explore all possible subsequences. The total number of subsequences for a string of length n is n(n + 1) / 2, and both methods will generate all of these subsequences.

**For Loop Approach: This method generates subsequences in an order determined by the starting index, potentially leading to better grouping of subsequences starting from each character.**

**Binary Decision Approach: This method can be more intuitive and easier to understand as it directly reflects the decision-making process of including or excluding each character.**

Choosing between these approaches depends on the specific requirements of the problem and personal preference for readability and clarity. Both achieve the goal of generating all subsequences recursively.

**Now we come to the actual topic, in this problem we don't need generate all subsequences, instead, we need generate all substrings**

**Generate all the substrings of a string using recursion**

To generate all substrings of a string using recursion, you need to ensure that you only consider contiguous sequences of characters.

Here is the correct recursive approach:

import java.util.ArrayList;

import java.util.List;

public class SubstringGenerator {

public static List<String> generateAllSubstrings(String str) {

List<String> result = new ArrayList<>();

generateSubstringsRecursive(str, 0, result);

return result;

}

private static void generateSubstringsRecursive(String str, int start, List<String> result) {

if (start == str.length()) {

return;

}

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

result.add(str.substring(start, end));

}

generateSubstringsRecursive(str, start + 1, result);

}

public static void main(String[] args) {

String str = "abc";

List<String> substrings = generateAllSubstrings(str);

for (String substring : substrings) {

System.out.println(substring);

}

}

}

**Explanation**

generateAllSubstrings Method:

This method initializes the recursive process by creating a list result to store the substrings and calling generateSubstringsRecursive with the initial parameters.

generateSubstringsRecursive Method:

This method generates substrings recursively.

The base case checks if start has reached the length of the string. If so, it returns because no further substrings can be generated starting from this index.

The method uses a for loop to generate all substrings that start from the current index start and end at every possible index from start + 1 to the end of the string.

It then recursively calls itself with the next starting index start + 1.

Main Method:

The main method demonstrates how to use the generateAllSubstrings method. It generates all substrings for the string "abc" and prints them.

**Benefits and Use Cases**

Contiguous Substrings: This approach ensures that only contiguous substrings are generated, which is the correct definition of substrings.

Recursive Structure: By advancing the start index in each recursive call, this method ensures all starting points are covered.

This corrected approach efficiently generates all substrings of a given string by leveraging recursion while ensuring only contiguous sequences are considered.

**What's the difference between the recursion method of generate all substrings of a string and generate all subsequences of a string**

The main difference between generating all substrings and generating all subsequences of a string lies in the definition and handling of contiguity:

Substrings: Substrings are contiguous sequences of characters within the string.

Subsequences: Subsequences are sequences that can be derived from the string by deleting some or no characters without changing the order of the remaining characters. They do not need to be contiguous.

**Recursive Method for Generating All Substrings**

To generate all substrings using recursion, we ensure that we only consider contiguous sequences:

import java.util.ArrayList;

import java.util.List;

public class SubstringGenerator {

public static List<String> generateAllSubstrings(String str) {

List<String> result = new ArrayList<>();

generateSubstringsRecursive(str, 0, result);

return result;

}

private static void generateSubstringsRecursive(String str, int start, List<String> result) {

if (start == str.length()) {

return;

}

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

result.add(str.substring(start, end));

}

generateSubstringsRecursive(str, start + 1, result);

}

public static void main(String[] args) {

String str = "abc";

List<String> substrings = generateAllSubstrings(str);

for (String substring : substrings) {

System.out.println(substring);

}

}

}

**Recursive Method for Generating All Subsequences**

To generate all subsequences using recursion, we consider both the inclusion and exclusion of each character:

import java.util.ArrayList;

import java.util.List;

public class SubsequenceGenerator {

public static List<String> generateAllSubsequences(String str) {

List<String> result = new ArrayList<>();

generateSubsequencesRecursive(str, 0, "", result);

return result;

}

private static void generateSubsequencesRecursive(String str, int index, String current, List<String> result) {

if (index == str.length()) {

result.add(current);

return;

}

// Exclude the current character and move to the next

generateSubsequencesRecursive(str, index + 1, current, result);

// Include the current character and move to the next

generateSubsequencesRecursive(str, index + 1, current + str.charAt(index), result);

}

public static void main(String[] args) {

String str = "abc";

List<String> subsequences = generateAllSubsequences(str);

for (String subsequence : subsequences) {

System.out.println(subsequence);

}

}

}

**Key Differences**

Contiguity:

Substrings: The substrings method only considers contiguous segments of the string, using a nested loop (or recursion with a loop) to handle all possible start and end indices.

Subsequences: The subsequences method considers each character independently, using two recursive branches to either include or exclude each character, hence handling non-contiguous segments.

Approach:

Substrings: Uses a loop within the recursive function to generate all possible substrings starting from each character.

Subsequences: Uses a binary decision (include/exclude) at each character position, leading to a tree of possible sequences.

Result Size:

Substrings: For a string of length n, there are n(n+1)/2 substrings.

Subsequences: For a string of length n, there are 2^n subsequences, including the empty string.

Example:

For the string "abc":

Substrings: "a", "ab", "abc", "b", "bc", "c"

Subsequences: "", "a", "b", "c", "ab", "ac", "bc", "abc"

These distinctions highlight the different problem spaces and the appropriate recursive strategies for each.

**Refer to**

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