<https://leetcode.com/problems/distinct-subsequences-ii/>

Given a string s, return *the number of* ***distinct non-empty subsequences*** *of* s. Since the answer may be very large, return it **modulo** 10^9 + 7.

A **subsequence** of a string is a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e.,"ace"is a subsequence of"abcde"while"aec"is not.

**Example 1:**

Input: s = "abc"

Output: 7

Explanation: The 7 distinct subsequences are "a", "b", "c", "ab", "ac", "bc", and "abc".

**Example 2:**

Input: s = "aba"

Output: 6

Explanation: The 6 distinct subsequences are "a", "b", "ab", "aa", "ba", and "aba".

**Example 3:**

Input: s = "aaa"

Output: 3

Explanation: The 3 distinct subsequences are "a", "aa" and "aaa".

**Constraints:**

* 1 <= s.length <= 2000
* s consists of lowercase English letters.

**Attempt 1: 2023-07-26**

**首先要解决的问题不涉及跑过，不涉及数字大而用到的modulo** 10^9 + 7**，而是先要解决在小规模输入下如何打印一个字符串所有的distinct subsequence，由于字符前后关系不能改变，所以一定不是permutation，而是更接近于combination，可以参考L77递归模板**

**本题的参考答案中和L77递归模板中最大的不同就是在每一层递归中加入了一个set**

**关键问题：每一层递归中的set过滤重复字符是什么原理？为何需要每一层重新生成一个新的set来过滤当前层的重复字符？有什么背后的逻辑支撑吗？**

**Refer to**

<https://grandyang.com/leetcode/940/>

这道题是之前那道 [Distinct Subsequences](http://www.cnblogs.com/grandyang/p/4294105.html) 的类似题目，这里只有一个字符串，让找出所有不同的子序列，如果字符串中没有重复字符，可以直接得到子序列的个数，但是这里由于重复字符的存在，就大大增加了难度。由于题目中提示了结果可能非常大，要对一个超大数取余，就相当于明确说了要用动态规划 Dynamic Programming 来做，下面就要来考虑 dp 数组的定义和状态转移方程的推导了。刚开始博主也是考虑用一个一维数组 dp，其中 dp[i] 表示以 S[i] 结尾的不同子序列的个数，就像 [这个帖子](https://leetcode.com/problems/distinct-subsequences-ii/discuss/192030/Java-DP-O/(N2/)-time-greater-O/(N/)-time-greater-O/(1/)-space) 中定义的一样，但是状态转移方程不好推导，那个帖子虽然代码可以跑通，但是解释的却不通，博主也纳闷这算是歪打正着么，希望哪位大神来解释一下。这里还是根据 [lee215 大神的帖子](https://leetcode.com/problems/distinct-subsequences-ii/discuss/192017/C++JavaPython-4-lines-O/(N/)-Time-O/(1/)-Space) 来讲解吧。这里使用一个大小为 26 的一维数组 dp，其中 dp[i] 表示以字符 i+’a’ 结尾的不同子序列的个数，因为题目中限定了只有小写字母，所以只有 26 个。以 aba 这个例子来分析一下，当遇到开头的a时，那么以a结尾的子序列只有一个，就是a，当遇到中间的b时，此时知道以b结尾的子序列有2个，分别是 b 和 ab，是怎么得来的呢，其实是空串和a后面分别加个b得来的，此时貌似得到的值和通过 sum(dp)+1 计算的结果相等，再来验证一下这个成不成立。当遇到末尾的a的时候，那么此时以a结尾的子序列就有4个，分别是 a，aa，ba，aba，是怎么得来的？在这个a加入之前，当前所有的子序列有，a，b，ab，如果再算上一个空串，[]，a，b，ab，则在其后面各加上一个b，就可以得到结果了，貌似也符合 sum(dp)+1 的规律，这其实也并不难理解，**因为在当前不同序列的基础上，加上任何一个字符都会得到另一个不同的子序列**，后面的加1是为了加上空串的情况，这个就是状态转移方程了，最终的结果是把 dp 数组累加起来取余后返回即可

<https://zhuanlan.zhihu.com/p/573517583>

**Solution 1:  Brute Force (600 min)**

**Style 1: 按照L77中的for loop写法如下**

**首先L77中的for循环递归的写法如下**

class Solution {

public List<List<Integer>> combine(int n, int k) {

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

int[] candidates = new int[n + 1];

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

candidates[i] = i;

}

// Since range [1,n], start index not 0 but 1

helper(candidates, result, new ArrayList<Integer>(), k, 1);

return result;

}

private void helper(int[] candidates, List<List<Integer>> result, List<Integer> tmp, int k, int index) {

if(tmp.size() == k) {

result.add(new ArrayList<Integer>(tmp));

return;

}

for(int i = index; i < candidates.length; i++) {

tmp.add(candidates[i]);

helper(candidates, result, tmp, k, i + 1);

tmp.remove(tmp.size() - 1);

}

}

}

**L940基于L77 for loop但是在每一层循环都添加了set来除掉当前层重复部分的写法:**

class Solution {

public int distinctSubseqII(String s) {

char[] arr = s.toCharArray();

return helper(arr, 0);

}

public int helper(char[] arr, int index) {

if(index >= arr.length) {

return 0;

}

Set<Character> set = new HashSet<>();

int ans = 0;

for(int i = index; i < arr.length; i++) {

if(set.contains(arr[i])) {

continue;

}

set.add(arr[i]);

// The '+1' is because the arr[i](current char)

// itself is also a unique subsequence

ans += 1 + helper(arr, i + 1);

//set.remove(arr[i]);

// We cannot remove arr[i] like L77 backtracking

// style, because we need to keep arr[i] to filter

// out duplicate newly added element

}

return ans;

}

}

Time Complexity: O(N^2)

Space Complexity: O(N^2)

**人工跟踪每一步:**

e.g "aba"

[level 1 recursion]

helper("aba",0)

|

lvl1 set={},index=0

lvl1 ans=0

|

lvl1 for(i=index(0))

|

lvl1 set.add('a')={a}

|

lvl1 ans(=0)+=1+lvl2 ans(=?)

|

[level 2 recursion]

helper("aba",1)

|

lvl2 set={},index=1

lvl2 ans=0

|

lvl2 for(i=index(1))

|

lvl2 set.add('b')={b}

|

lvl2 ans(=0)+=1+lvl3 ans(=?)

|

[level 3 recursion]

helper("aba",2)

|

lvl3 set={},index=2

lvl3 ans=0

|

lvl3 for(i=index(2))

|

lvl3 set.add('a')={a}

|

lvl3 ans(=0)+=1+lvl4 ans(=?)

|

[level 4 recursion]

helper("aba",3)

|

if(index >= arr.length)

return 0

return lvl4 ans=0

|

[back to level 3 recursion]

lvl3 ans(=0)+=1+lvl4 ans(=0)=>1+0=1

|

lvl3 for(i=index(2),i++ ==> 2->3) skip

return lvl3 ans=1

|

[back to level 2 recursion]

lvl2 ans(=0)+=1+lvl3 ans(=1)=>1+1=2

|

lvl2 for(i=index(1),i++ ==> 1->2)

|

lvl2 set.add('a')={b,a}

|

[level 3 recursion]

helper("aba",3)

|

if(index >= arr.length)

return 0

return lvl3 ans=0

|

[back to level 2 recursion]

lvl2 ans(=2)+=1+lvl3 ans(=0)=>2+1=3

|

lvl2 for(i++ 2->3) skip

return lvl2 ans=3

|

[back to level 1 recursion]

lvl1 ans(=0)+=1+lvl2 ans(=3)=>1+3=4

|

lvl1 for(i++ 0->1)

|

lvl1 set.add('b')={a,b}

|

[level 2 recursion]

helper("aba",2)

|

lvl2 ans=0

|

lvl2 for(i=index(2))

|

lvl2 set.add('a')={a}

|

[level 3 recursion]

helper("aba",3)

|

if(index >= arr.length)

return 0

return lvl3 ans=0

|

[back to level 2 recursion]

lvl2 ans(=0)+=1+lvl3 ans(=0)=>1+0=1

|

lvl2 for(i++ 2->3) skip

return lvl2 ans=1

|

[back to level 1 recursion]

lvl1 ans(=4)+=1+lvl2 ans(=1)=>4+2=6

|

lvl1 for(i++ 1->2)

|

lvl1 set.add('a') => since lvl1 set={a,b} already contains 'a', skip

|

lvl1 for(i++ 2->3) skip

|

ans=6

|

end

**加入对set的更新状态的log后的基本写法**

class Solution {

public int distinctSubseqII(String s) {

char[] arr = s.toCharArray();

return helper(arr, 0, 1);

}

public int helper(char[] arr, int index, int lvl) {

System.out.println("level " + lvl + " recursion start");

if(index >= arr.length) {

return 0;

}

Set<Character> set = new HashSet<>();

System.out.println("New level " + lvl + " set initialize as empty {}");

int ans = 0;

for(int i = index; i < arr.length; i++) {

if(set.contains(arr[i])) {

System.out.println("level " + lvl + " set already contains '" + arr[i] + "', ignore '" + arr[i] + "'");

continue;

}

set.add(arr[i]);

System.out.println("level " + lvl + " set add '" + arr[i] + "'");

ans += 1 + helper(arr, i + 1, lvl + 1);

}

StringBuffer sb = new StringBuffer();

for(Character c : set) {

sb.append(c).append(",");

}

System.out.println("level " + lvl + " set now contains {" + sb.deleteCharAt(sb.length() - 1).toString() + "}");

return ans;

}

}

**打印状态表**

level 1 recursion start

New level 1 set initialize as empty {}

level 1 set add 'a'

level 2 recursion start

New level 2 set initialize as empty {}

level 2 set add 'b'

level 3 recursion start

New level 3 set initialize as empty {}

level 3 set add 'a'

level 4 recursion start

level 3 set now contains {a}

level 2 set add 'a'

level 3 recursion start

level 2 set now contains {a,b}

level 1 set add 'b'

level 2 recursion start

New level 2 set initialize as empty {}

level 2 set add 'a'

level 3 recursion start

level 2 set now contains {a}

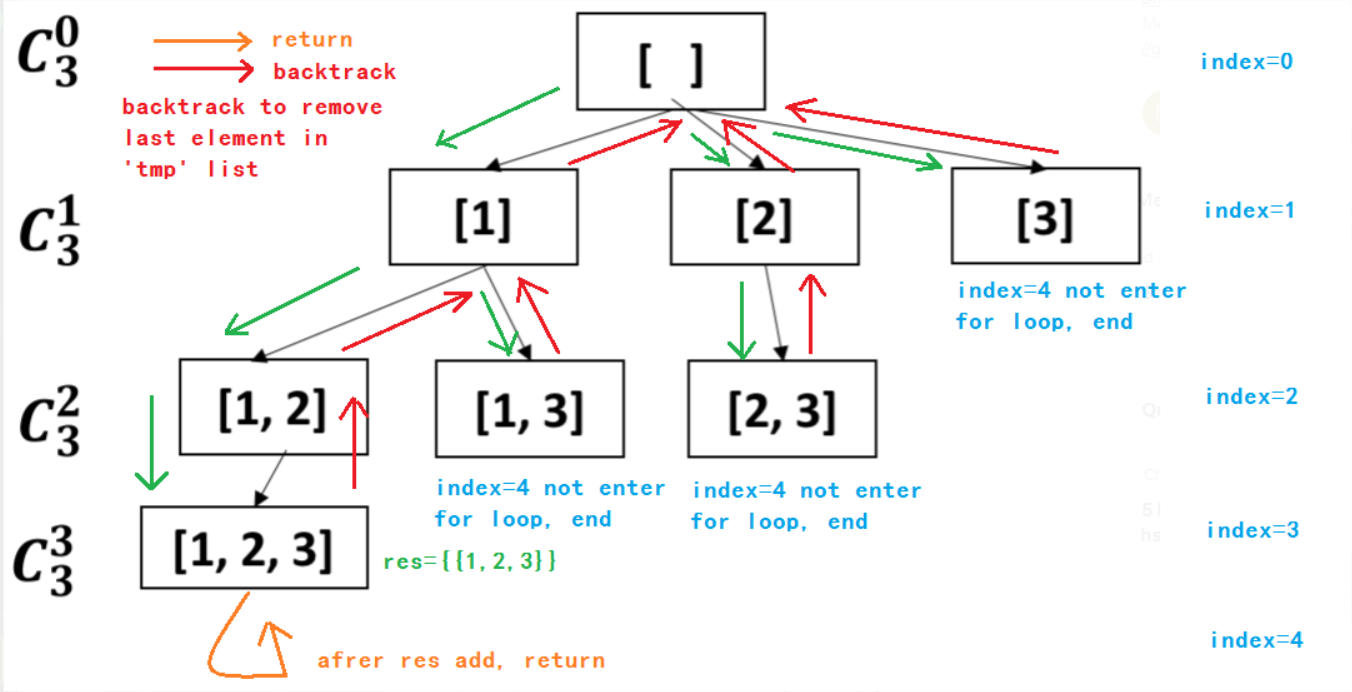
level 1 set already contains 'a', ignore 'a'

level 1 set now contains {a,b}

6

**如果想打印所有distinct subequence怎么办？(参见L90/P11.2.Subsets II的处理方法)**

**想通过for循环风格而非pick，not pick风格（参见L77. Combination中的两种递归方法）就打印出所有sequence有一定难度，然而受到L77的backtracking for循环模板的影响，并按照如下L77过程图的提示: 本质上要打印所有index下的状态[], [1], [2], [3], [1,2], [1,3], [2,3], [1,2,3], 而非只打印最终index=3状态下的[1,2,3]**



**发现只要打破只在index >= arr.length的时候才将记录递归结果的tmp加载到结果result中的制约就可以记录任意长度(从0到arr.length)的组合，那么解决方案就自然而然的出现了，即引入在index < arr.length状态下也把tmp加载到结果result中但是不返回而是继续进行当前层计算的逻辑**

class Solution {

public int distinctSubseqII(String s) {

char[] arr = s.toCharArray();

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

helper(arr, 0, result, new ArrayList<>());

return result.size() - 1;

}

public void helper(char[] arr, int index, List<List<Character>> result, List<Character> tmp) {

if(index >= arr.length) {

result.add(new ArrayList<>(tmp));

return;

}

// 要记录所有index下的tmp状态，必须在index < arr.length状态下也把tmp加载

// 到结果result中但是不返回而是继续进行当前层计算的逻辑

if(index < arr.length) {

result.add(new ArrayList<>(tmp));

}

Set<Character> set = new HashSet<>();

for(int i = index; i < arr.length; i++) {

if(set.contains(arr[i])) {

continue;

}

set.add(arr[i]);

tmp.add(arr[i]);

helper(arr, i + 1, result, tmp);

tmp.remove(tmp.size() - 1);

}

}

}

=======================================================================================

优化一下条件如下:

class Solution {

public int distinctSubseqII(String s) {

char[] arr = s.toCharArray();

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

helper(arr, 0, result, new ArrayList<>());

return result.size() - 1;

}

public void helper(char[] arr, int index, List<List<Character>> result, List<Character> tmp) {

if(index > arr.length) {

return;

}

// 要记录所有index下的tmp状态，必须把所有index <= arr.length状态下的tmp

// 都记录到result中且不返回而是继续进行当前层计算的逻辑

result.add(new ArrayList<>(tmp));

Set<Character> set = new HashSet<>();

for(int i = index; i < arr.length; i++) {

if(set.contains(arr[i])) {

continue;

}

set.add(arr[i]);

tmp.add(arr[i]);

helper(arr, i + 1, result, tmp);

tmp.remove(tmp.size() - 1);

}

}

}

**Style 2: 按照L77中的pick and not pick写法如下**

**基于L77和以下链接的基本改造**

<https://leetcode.com/problems/distinct-subsequences-ii/solutions/2395259/can-anyone-optimize-this-recursive-code-its-giving-tle-for-pcrdhwdxmqdznbenhwjsenjhvulyve/>

class Solution {

public int distinctSubseqII(String s) {

List list = new LinkedList<>();

Set<List> set = new HashSet<>();

f(s, s.length() - 1, list, set);

return set.size() - 1;

}

void f(String s, int i, List list, Set<List> set){

if(i < 0) {

set.add(new LinkedList(list));

return;

}

list.add(s.charAt(i));

f(s, i-1, list, set);

list.remove(list.size() - 1);

f(s, i-1, list, set);

return;

}

}

**依然沿用L77的list(拥有L77 pick and not pick style的方式，能够通过list result打印出所有独特组合，但是不具备除重功能)**

class Solution {

public List<List<Character>> combine(String s) {

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

char[] chars = s.toCharArray();

helper(chars, result, new ArrayList<Character>(),0);

return result;

}

private void helper(char[] chars, List<List<Character>> result, List<Character> tmp, int index) {

if(index == chars.length) {

result.add(new ArrayList<Character>(tmp));

return;

}

// Based on tree analysis, add return condition when index > n

if(index > chars.length) {

return;

}

// Not pick

helper(chars, result, tmp, index + 1);

// Pick

tmp.add(chars[index]);

helper(chars, result, tmp, index + 1);

tmp.remove(tmp.size() - 1);

}

}

**改用set满足L940的要求(拥有L77 pick and not pick style的方式，能够通过set result打印出所有独特组合，然后通过result.size() - 1得到个数，TLE 49/110)**

class Solution {

public int distinctSubseqII(String s) {

Set<List<Character>> result = new HashSet<>();

char[] chars = s.toCharArray();

helper(chars, result, new ArrayList<Character>(),0);

// The '-1' means remove the empty 'List<Character>' list which always in set

return result.size() - 1;

}

private void helper(char[] chars, Set<List<Character>> result, List<Character> tmp, int index) {

if(index == chars.length) {

result.add(new ArrayList<Character>(tmp));

return;

}

// Based on tree analysis, add return condition when index > n

if(index > chars.length) {

return;

}

// Not pick

helper(chars, result, tmp, index + 1);

// Pick

tmp.add(chars[index]);

helper(chars, result, tmp, index + 1);

tmp.remove(tmp.size() - 1);

}

}

Time Complexity: O(N^2)

Space Complexity: O(N^2)

**Solution 2: Recursion + Memoization**

**首先解决modulo 10^9+7的问题, 然后加入memoization**

class Solution {

private static final int MOD = 1000000007;

public int distinctSubseqII(String s) {

Integer[] memo = new Integer[s.length() + 1];

char[] arr = s.toCharArray();

return helper(arr, 0, memo);

}

public int helper(char[] arr, int index, Integer[] memo) {

if(memo[index] != null) {

return memo[index];

}

if(index >= arr.length) {

return 0;

}

Set<Character> set = new HashSet<>();

int ans = 0;

for(int i = index; i < arr.length; i++) {

if(set.contains(arr[i])) {

continue;

}

set.add(arr[i]);

// The '+1' is because the arr[i](current char)

// itself is also a unique subsequence

// Be careful, the 'MOD' has to use 3 times, especially the first

// ans % MOD is required

ans = (ans % MOD + 1 + helper(arr, i + 1, memo) % MOD) % MOD;

//set.remove(arr[i]);

// We cannot remove arr[i] like L77 backtracking

// style, because we need to keep arr[i] to filter

// out duplicate newly added element

}

return memo[index] = ans;

}

}

Time Complexity: O(N^2)

Space Complexity: O(N^2)

**Refer to**

<https://leetcode.com/problems/distinct-subsequences-ii/solutions/2888119/java-simple-solution-using-recursion-memoization/>

class Solution {

private static final int MOD = 1\_000\_000\_007;

public long helper(char[] arr, int i, Long[] dp) {

if(i>=arr.length) return 0;

if(dp[i] != null) return dp[i];

Set<Character> set = new HashSet<>();

long ans = 0;

for(int j=i;j<arr.length;j++) {

if(set.contains(arr[j])) continue;

set.add(arr[j]);

ans = (ans%MOD + 1 + helper(arr, j+1, dp)%MOD)%MOD;

}

return dp[i] = ans;

}

public int distinctSubseqII(String s) {

char[] arr = s.toCharArray();

int n = arr.length;

Long[] dp = new Long[n];

return (int)(helper(arr, 0, dp)%MOD);

}

}

**Solution 3: DP (360 min)**

**这题由于前面递归解法过程中不规则，虽然依然贯彻了"顶"和"底"的思路，但是中间的推导过程由于涉及到采用何种方法可以等效替换递归中的set过滤当前层重复的问题，这样一来就需要引入新的参数，而且初始化并非完全按照"底"的状态来，在前述递归解法中"底"是最终返回0，然而在这里初始化为1，来满足空串的处理，而在最后结果中再-1去掉空串**

**不考虑取module = 10^9 + 7的时候的基本模板("顶"和"底"的思路依然符合递归中状态)**

class Solution {

public int distinctSubseqII(String s) {

// end[c] to denote the number of distinct subsequences ending with char c so far

// note "so far" -> not include current round ones, used to replicate the usage

// as 'set' in recursion solution

int[] end = new int[26];

int len = s.length();

// dp[i] means how many distinct subsequence

// based on substring of s as s[i,len)

int[] dp = new int[len + 1];

dp[len] = 1;

/\*\*

e.g "aba", len = 3

dp[len] = 1 (empty subsequence, count in as 1)

end['a' - 'a'] = end[0] = 0 so far -> {}

dp[len - 1] = 1 ('a' has 1 distinct non-empty subsequece as itself {'a'})

end['a' - 'a'] = end[0] = 1 updated -> {'a'}

end['b' - 'a'] = end[1] = 0 so far -> {}

dp[len - 2] = 3 ('ba' has 3 distinct non-empty subsequeces, {'b','a','ba'})

end['b' - 'a'] = end[1] = 2 updated -> {'b','ab'}

end['a' - 'a'] = end[0] = 1 so far -> {'a'}

dp[len - 3] = 6 ('aba' has 6 distinct non-empty subsequences, {'b','a','ba','ab','aa','aba'})

end['a' - 'a'] = end[0] = 4 updated -> {'a','ba','aa','aba'}

\*/

for(int i = len - 1; i >= 0; i--) {

int prev = dp[i + 1];

dp[i] = dp[i + 1] \* 2 - end[s.charAt(i) - 'a'];

end[s.charAt(i) - 'a'] = prev;

}

return dp[0] - 1;

}

}

**考虑取module = 10^9 + 7的时候的完整解("顶"和"底"的思路依然符合递归中状态, PASS 109/110)**

class Solution {

public int distinctSubseqII(String s) {

int MOD = 1000000007;

// end[c] to denote the number of distinct subsequences ending with char c so far

// note "so far" -> not include current round ones, used to replicate the usage

// as 'set' in recursion solution

long[] end = new long[26];

int len = s.length();

// dp[i] means how many distinct subsequence

// based on substring of s as s[i,len)

long[] dp = new long[len + 1];

// 与recursion中的初始化"底"dp[len]应该等于0，但并不适用于dp中，根据对于

// 空串的处理方便化改造为dp[len] = 1

dp[len] = 1;

/\*\*

e.g "aba", len = 3

dp[len] = 1 (empty subsequence, count in as 1)

end['a' - 'a'] = end[0] = 0 so far -> {}

dp[len - 1] = 1 ('a' has 1 distinct non-empty subsequece as itself {'a'})

end['a' - 'a'] = end[0] = 1 updated -> {'a'}

end['b' - 'a'] = end[1] = 0 so far -> {}

dp[len - 2] = 3 ('ba' has 3 distinct non-empty subsequeces, {'b','a','ba'})

end['b' - 'a'] = end[1] = 2 updated -> {'b','ab'}

end['a' - 'a'] = end[0] = 1 so far -> {'a'}

dp[len - 3] = 6 ('aba' has 6 distinct non-empty subsequences, {'b','a','ba','ab','aa','aba'})

end['a' - 'a'] = end[0] = 4 updated -> {'a','ba','aa','aba'}

\*/

for(int i = len - 1; i >= 0; i--) {

long prev = dp[i + 1];

dp[i] = (dp[i + 1] \* 2 % MOD - end[s.charAt(i) - 'a'] + MOD) % MOD;

end[s.charAt(i) - 'a'] = prev;

}

return (int)(dp[0] - 1);

}

}

**Refer to**

<https://leetcode.com/problems/distinct-subsequences-ii/solutions/242780/concise-solution-with-well-explanation/>

dp[i] is the number of distinct subsequences for substring s[0..i]. **It includes the empty string "" to make things easy. We'll exclude this one in the end by simply minus 1**, so the answer isdp[n-1] - 1.

If all characters are distinct, then dp[i] = dp[i-1]\*2, that is all previous sub-sequences without s[i], plus all previous sub-sequences appended with s[i].

If there are duplicate characters, We use end[c] to denote the number of distinct subsequences ending with char c so far. So number of all previous subsequences with s[i] should be subtracted by previous end[s[i]]. That is:dp[i] = dp[i-1] \* 2 - end[s[i]]

It is easy to compact dp from O(n) to O(1) as d[i] only depends on dp[i-1]. The code is quite simple actually:

public int DistinctSubseqII(string s) {

int res = 1, MOD = (int)1e9 + 7;

var end = new int[26];

foreach (var c in s) {

var pre = res;

res = (res \* 2 % MOD - end[c - 'a'] + MOD) % MOD;

end[c - 'a'] = pre;

}

return res - 1; // exclude ""

}

**Why is your end[c - 'a' ] = pre and not res?**

<https://leetcode.com/problems/distinct-subsequences-ii/solutions/242780/concise-solution-with-well-explanation/comments/345603>

Let's say we're handling i-th char s[i]=c, its previous duplicate is at j-th, all char after j and before i are not c, like this

j i

"....xc.....c".

What sequences are counted twice because of duplicate c? All sequences for substring s[0..j-1], appended j-th char will duplicate those appended i-th char.

So end[c] is res[j-1] that is previous result of res[j], not res[j]. And end[c] is updated to res[i-1] as well.

**How adding mod is helping us to avoid overflow ?**

<https://leetcode.com/problems/distinct-subsequences-ii/solutions/242780/concise-solution-with-well-explanation/comments/1332646>

It helping us to avoid negative numbers.

res \* 2 - big positive number, but res \* 2 % MOD may be small number.

end[c - 'a'] may be little or big too, because it is value from some previous steps.

When we do subtraction res \* 2 % MOD - end[c - 'a'] we may get negative result. But both this numbers are less than MOD, so

-MOD < res \* 2 % MOD - end[c - 'a'] < MOD

0 < res \* 2 % MOD - end[c - 'a'] + MOD < 2\*MOD

**Refer to**

<https://leetcode.com/problems/distinct-subsequences-ii/solutions/199165/java-beats-100-o-n-space-o-1-space-10-lines-with-simple-explanation/>

Lets break down to two cases for every character iterated in string:

New unique character

Transforms existing set of total combinations by creating new combos via **appending** each string.

simplifies to setSize = setSize \* 2

Character that appeared before

Still appends the existing set, but will generate clashing combinations.

Will generate the same combos as the **last time it appeared**, so if we cache previous combinations and remove it with new combinations, we will have a unique set again.

public int distinctSubseqII(String S) {

int[] dict = new int[26]; // Save 'total' count when a character appears.

int total = 1; //Empty string, starting at count 1

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

int combo = total \* 2 - dict[c - 'a']; // New - Duplicates

dict[c - 'a'] = total; // if 'c' ever appears again, it will clash with the current combos.

total = combo < 0 ? combo + 1000000007 : combo % 1000000007; // mod and fix negative mods

}

return total - 1; // Subtract the empty string

}