<https://leetcode.com/problems/scramble-string/description/>

We can scramble a string s to get a string t using the following algorithm:

If the length of the string is 1, stop.

If the length of the string is > 1, do the following:

Split the string into two non-empty substrings at a random index, i.e., if the string is s, divide it to x and y where s = x + y.

**Randomly** decide to swap the two substrings or to keep them in the same order. i.e., after this step, s may become s = x + y or s = y + x.

Apply step 1 recursively on each of the two substrings x and y.

Given two strings s1 and s2 of **the same length**, return true if s2 is a scrambled string of s1, otherwise, return false.

**Example 1:**

Input: s1 = "great", s2 = "rgeat"

Output: true

Explanation: One possible scenario applied on s1 is:

"great" --> "gr/eat" // divide at random index.

"gr/eat" --> "gr/eat" // random decision is not to swap the two substrings and keep them in order.

"gr/eat" --> "g/r / e/at" // apply the same algorithm recursively on both substrings. divide at random index each of them.

"g/r / e/at" --> "r/g / e/at" // random decision was to swap the first substring and to keep the second substring in the same order.

"r/g / e/at" --> "r/g / e/ a/t" // again apply the algorithm recursively, divide "at" to "a/t".

"r/g / e/ a/t" --> "r/g / e/ a/t" // random decision is to keep both substrings in the same order.

The algorithm stops now, and the result string is "rgeat" which is s2.

As one possible scenario led s1 to be scrambled to s2, we return true.

**Example 2:**

Input: s1 = "abcde", s2 = "caebd"

Output: false

**Example 3:**

Input: s1 = "a", s2 = "a"

Output: true

**Constraints:**

* s1.length == s2.length
* 1 <= s1.length <= 30
* s1 and s2 consist of lowercase English letters

**其实本题的原版描述如下，是更直观的树状结构图:**

Given a string  *s1*, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of  *s1* = "great":

great

/ \

gr eat

/ \ / \

g r e at

/ \

a t

To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node "gr" and swap its two children, it produces a scrambled string "rgeat".

rgeat

/ \

rg eat

/ \ / \

r g e at

/ \

a t

We say that "rgeat" is a scrambled string of "great".

Similarly, if we continue to swap the children of nodes "eat" and "at", it produces a scrambled string "rgtae".

rgtae

/ \

rg tae

/ \ / \

r g ta e

/ \

t a

We say that "rgtae" is a scrambled string of "great".

Given two strings  *s1* and  *s2* of the same length, determine if  *s2* is a scrambled string of  *s1*.

Example 1:

Input: s1 = "great", s2 = "rgeat"

Output: true

Example 2:

Input: s1 = "abcde", s2 = "caebd"

Output: false

**Attempt 1: 2023-11-23**

**Solution 1: Native DFS (180 min, TLE 286/290)**

**Style 1: 每层循环都是传递的实际获取的字符串s1和s2**

class Solution {

public boolean isScramble(String s1, String s2) {

if(s1.length() != s2.length()) {

return false;

}

if(s1.equals(s2)) {

return true;

}

int[] freq = new int[26];

for(int i = 0; i < s1.length(); i++) {

freq[s1.charAt(i) - 'a']++;

freq[s2.charAt(i) - 'a']--;

}

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

if(freq[i] != 0) {

return false;

}

}

for(int i = 1; i < s1.length(); i++) {

// There exist a cut point to split s1 into s11 and s12, also split s2 into s21 and s22, where s11 and s21 are scramble, and s12 and s22 are also scramble

// e.g s1 = "great", s2 = "rgtae"

// s11 = "gr", s12 = "eat", s21 = "rg", s22 = "tae"

String s11 = s1.substring(0, i);

String s12 = s1.substring(i);

String s21 = s2.substring(0, i);

String s22 = s2.substring(i);

if(isScramble(s11, s21) && isScramble(s12, s22)) {

return true;

}

// OR s11 and s22 are scramble, and s12 and s21 are also scramble

// e.g s1 = "great", s2 = "taerg"

// s11 = "gr", s12 = "eat", s21 = "tae", s22 = "rg" (before swap s21, s22)

// but be careful, after split s2 into s21 and s22 by cut point

// we have to swap two substrings, the s2\_new = s22 + s21, and

// also the cut point shift, the length of s22 equals s11 and

// the length of s21 equals s12

s22 = s2.substring(s2.length() - i);

s21 = s2.substring(0, s2.length() - i);

if(isScramble(s11, s22) && isScramble(s12, s21)) {

return true;

}

}

return false;

}

}

Time complexity: O(5^N)

Space complexity: O(N^2)

**Time Complexity analysis:**

**Refer to**

<https://leetcode.com/problems/scramble-string/solutions/29392/share-my-4ms-c-recursive-solution/comments/293854>

To the people who are unable to get why this code is exponential in run time have a look at this. I have written in layman terms

For string of size n take time to be T(n) for it

We are starting from i=1 and going till n-1

This line of code isScramble(s1.substr(0,i), s2.substr(0,i)) && isScramble(s1.substr(i), s2.substr(i)))

is T(i) + T(n-i)

and this line of code just below isScramble(s1.substr(0,i), s2.substr(len-i)) && isScramble(s1.substr(i), s2.substr(0,len-i))

is again T(i) + T(n-i)

So basically for T(n) we have:

T(n) = for i in 1 to n-1 : +2\*(T(i) + T(n-i))

i.e.

T(n) = 2\*(T(1) + T(n-1) + T(2) + T(n-2) + ... + T(n-1) + T(1)) ,

now we have two equal sequences T(1) to T(n-1) from left to right and T(n-1) to T(1) from right to left. So above equation can be written as

T(n) = 2 \* 2(T(1) + T(2) + ...+ T(n-2) + T(n-1))

T(n) = 4 \* (T(1) + T(2) + ...+ T(n-2) ) + 4 \* T(n-1)

if we substitute n-1 in place of n above we get T(n-1) = 4 \* (T(1) + T(2) + ...+ T(n-2) ) , using this in the above equation we get

T(n) = T(n-1) + 4 \* T(n-1)

T(n) = 5 \* T(n-1)

If we open this we get

T(n) = 5 \* 5 \* T(n-2) = 5 \* 5 \* 5 \* T(n-3) = 5 \* 5 \* ... \* 5 (n times)

so T(n) = O(5^n)

**Refer to**

<https://grandyang.com/leetcode/87/>

这道题定义了一种搅乱字符串，就是说假如把一个字符串当做一个二叉树的根，然后它的非空子字符串是它的子节点，然后交换某个子字符串的两个子节点，重新爬行回去形成一个新的字符串，这个新字符串和原来的字符串互为搅乱字符串。这道题可以用递归 Recursion 或是动态规划 Dynamic Programming 来做，我们先来看递归的解法，**简单的说，就是 s1 和 s2 是 scramble 的话，那么必然存在一个在 s1 上的长度 l1，将 s1 分成 s11 和 s12 两段，同样有 s21 和 s22，那么要么 s11 和 s21 是 scramble 的并且 s12 和 s22 是 scramble 的；要么 s11 和 s22 是 scramble 的并且 s12 和 s21 是 scramble 的。** 就拿题目中的例子 rgeat 和 great 来说，rgeat 可分成 rg 和 eat 两段， great 可分成 gr 和 eat 两段，rg 和 gr 是 scrambled 的， eat 和 eat 当然是 scrambled。根据这点，我们可以写出代码如下：

// Recursion

class Solution {

public:

bool isScramble(string s1, string s2) {

if (s1.size() != s2.size()) return false;

if (s1 == s2) return true;

string str1 = s1, str2 = s2;

sort(str1.begin(), str1.end());

sort(str2.begin(), str2.end());

if (str1 != str2) return false;

for (int i = 1; i < s1.size(); ++i) {

string s11 = s1.substr(0, i);

string s12 = s1.substr(i);

string s21 = s2.substr(0, i);

string s22 = s2.substr(i);

if (isScramble(s11, s21) && isScramble(s12, s22)) return true;

s21 = s2.substr(s1.size() - i);

s22 = s2.substr(0, s1.size() - i);

if (isScramble(s11, s21) && isScramble(s12, s22)) return true;

}

return false;

}

};

**Refer to**

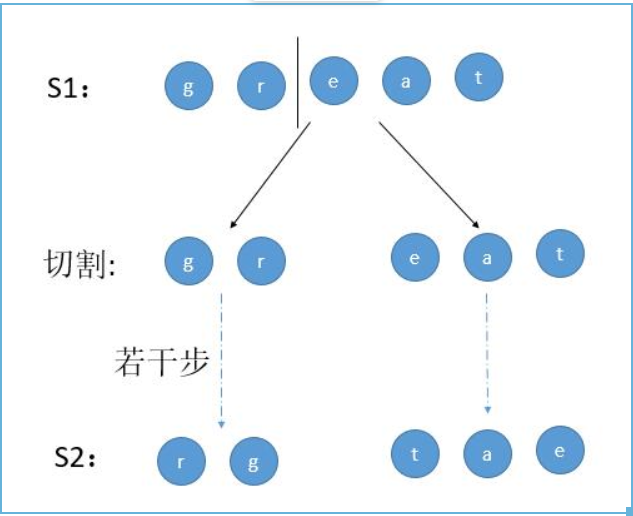
<https://leetcode.wang/leetCode-87-Scramble-String.html>

# **解法一 递归**

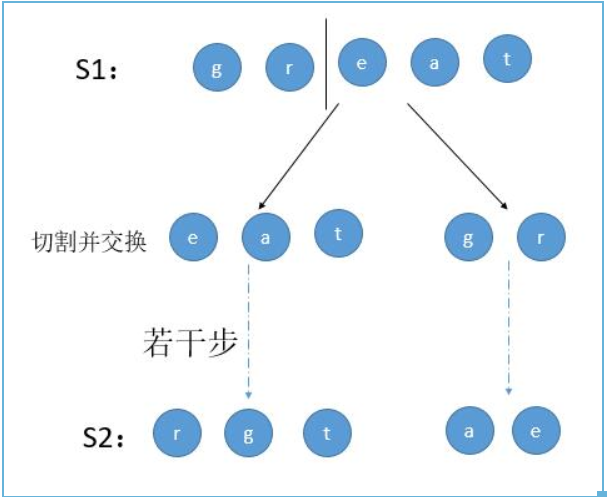
开始的时候，由于给出的图示很巧都是平均分的，我以为只能平均分字符串，看了[这里](https://leetcode.com/problems/scramble-string/discuss/29387/Accepted-Java-solution)，明白其实可以任意位置把字符串分成两部分，这里需要注意一下。

这道题很容易想到用递归的思想去解，假如两个字符串 great 和 rgeat。考虑其中的一种切割方式。

第 1 种情况：S1 切割为两部分，然后进行若干步切割交换，最后判断两个子树分别是否能变成 S2 的两部分。



第 2 种情况：S1 切割并且交换为两部分，然后进行若干步切割交换，最后判断两个子树是否能变成 S2 的两部分。



上边是一种切割方式，我们只需要遍历所有的切割点即可。

public boolean isScramble(String s1, String s2) {

if (s1.length() != s2.length()) {

return false;

}

if (s1.equals(s2)) {

return true;

}

//判断两个字符串每个字母出现的次数是否一致

int[] letters = new int[26];

for (int i = 0; i < s1.length(); i++) {

letters[s1.charAt(i) - 'a']++;

letters[s2.charAt(i) - 'a']--;

}

//如果两个字符串的字母出现不一致直接返回 false

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

if (letters[i] != 0) {

return false;

}

}

//遍历每个切割位置

for (int i = 1; i < s1.length(); i++) {

//对应情况 1 ，判断 S1 的子树能否变为 S2 相应部分

if (isScramble(s1.substring(0, i), s2.substring(0, i)) && isScramble(s1.substring(i), s2.substring(i))) {

return true;

}

//对应情况 2 ，S1 两个子树先进行了交换，然后判断 S1 的子树能否变为 S2 相应部分

if (isScramble(s1.substring(i), s2.substring(0, s2.length() - i)) &&

isScramble(s1.substring(0, i), s2.substring(s2.length() - i)) ) {

return true;

}

}

return false;

}

**Style 2: 每层循环并非传递的实际获取的字符串s1和s2，而是子字符串在原始s1和s2上的起始位置和长度(每次子字符串s1和s2的长度必须相同)，所以实际上需要三个参数，子字符串s1开始的位置i，子字符串s2开始的位置j，子字符串的长度**

class Solution {

public boolean isScramble(String s1, String s2) {

if(s1.length() != s2.length()) {

return false;

}

int n = s1.length();

return helper(s1, s2, 0, 0, n);

}

private boolean helper(String s1, String s2, int i, int j, int len) {

// No substring length check required anymore since

// on each recursion level pass in same length 'len'

// which will cut out same length substring on s1, s2

// Base condition when exceed length

if(i + len > s1.length() || j + len > s1.length()) {

return false;

}

// Base condition

if(s1.substring(i, i + len).equals(s2.substring(j, j + len))) {

return true;

}

// Check if s1[i:i+len-1] and s2[j:j+len-1] have same characters

int[] freq = new int[26];

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

freq[s1.charAt(i + k) - 'a']++;

freq[s2.charAt(j + k) - 'a']--;

}

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

if(freq[k] != 0) {

return false;

}

}

for(int k = 1; k < len; k++) {

// There exist a cut point to split current s1 into s11 and s12,

// also split current s2 into s21 and s22, where s11 and s21 are

// scramble, and s12 and s22 are also scramble

// e.g s1 = "great", s2 = "rgtae"

// s11 = "gr", s12 = "eat", s21 = "rg", s22 = "tae"

// In general, we will check these two pairs if both scramble in next

// recursion with base condition:

// s1.substring(i, i + len).equals(s2.substring(j, j + len))

// and if not equal will continue to check if same characters with freq map

// Pair1(s11 vs s21):

// s11 = s1[i:i+k), s21 = s2[j:j+k) -> helper(s1, s2, i, j, k)

// Pair2(s12 vs s22):

// s12 = s1[i+k:i+len), s22 = s2[j+k:j+len) -> helper(s1, s2, i + k, j + k, len - k)

if(helper(s1, s2, i, j, k) && helper(s1, s2, i + k, j + k, len - k)) {

return true;

}

// OR s11 and s22 are scramble, and s12 and s21 are also scramble

// e.g s1 = "great", s2 = "taerg"

// s11 = "gr", s12 = "eat", s21 = "tae", s22 = "rg" (before swap s21, s22)

// but be careful, after split s2 into s21 and s22 by cut point

// we have to swap two substrings, the s2\_new = s22 + s21, and

// also the cut point shift, the length of s22 equals s11 and

// the length of s21 equals s12

// In general, we will check these two pairs if both scramble in next

// recursion with base condition:

// s1.substring(i, i + len).equals(s2.substring(j, j + len))

// and if not equal will continue to check if same characters with freq map

// Pair1(s11 vs s22):

// s11 = s1[i:i+k), s22 = s2[j+len-k:j+len) -> helper(s1, s2, i, j + len - k, k)

// Pair2(s12 vs s21):

// s12 = s1[i+k:i+len), s21 = s2[j:j+len-k) -> helper(s1, s2, i + k, j, len - k)

if(helper(s1, s2, i, j + len - k, k) && helper(s1, s2, i + k, j, len - k)) {

return true;

}

}

return false;

}

}

**Refer to**

<https://leetcode.com/problems/scramble-string/solutions/3358560/dynamic-programming-solution-to-check-if-strings-are-scrambled-with-easy-and-full-explanation/>

**Solution 2: DFS + Memoization (30min)**

**Style 1: 基于Solution 1的Style 1每层循环都是传递的实际获取的字符串s1和s2的情况，在memo中也存储字符串本身**

**重点：两个字符串并没有办法只用1个记录切割点的变量i来记录，记录的时候必须同时照顾s1和s2两者的状态，本方法中采用了s1 + "\_" + s2的组合配合Hash Table的模式同时记录两者状态**

class Solution {

public boolean isScramble(String s1, String s2) {

Map<String, Boolean> memo = new HashMap<>();

return helper(s1, s2, memo);

}

private boolean helper(String s1, String s2, Map<String, Boolean> memo) {

String key = s1 + "\_" + s2;

if(memo.containsKey(key)) {

return memo.get(key);

}

if(s1.length() != s2.length()) {

memo.put(key, false);

return false;

}

if(s1.equals(s2)) {

memo.put(key, true);

return true;

}

int[] freq = new int[26];

for(int i = 0; i < s1.length(); i++) {

freq[s1.charAt(i) - 'a']++;

freq[s2.charAt(i) - 'a']--;

}

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

if(freq[i] != 0) {

memo.put(key, false);

return false;

}

}

for(int i = 1; i < s1.length(); i++) {

// There exist a cut point to split s1 into s11 and s12, also split s2 into s21 and s22, where s11 and s21 are scramble, and s12 and s22 are also scramble

// e.g s1 = "great", s2 = "rgtae"

// s11 = "gr", s12 = "eat", s21 = "rg", s22 = "tae"

String s11 = s1.substring(0, i);

String s12 = s1.substring(i);

String s21 = s2.substring(0, i);

String s22 = s2.substring(i);

if(helper(s11, s21, memo) && helper(s12, s22, memo)) {

memo.put(key, true);

return true;

}

// OR s11 and s22 are scramble, and s12 and s21 are also scramble

// e.g s1 = "great", s2 = "taerg"

// s11 = "gr", s12 = "eat", s21 = "tae", s22 = "rg" (before swap s21, s22)

// but be careful, after split s2 into s21 and s22 by cut point

// we have to swap two substrings, the s2\_new = s22 + s21, and

// also the cut point shift, the length of s22 equals s11 and

// the length of s21 equals s12

s22 = s2.substring(s2.length() - i);

s21 = s2.substring(0, s2.length() - i);

if(helper(s11, s22, memo) && helper(s12, s21, memo)) {

memo.put(key, true);

return true;

}

}

memo.put(key, false);

return false;

}

}

Time complexity: O(N^4)

The time complexity of the algorithm is O(n^4), n is the length of the strings. This is because the algorithm checks all possible splits of the two strings, which takes O(n^2) time, and for each split, it recursively checks if the substrings are scrambled versions of each other, which takes O(n^2) time in the worst case. Therefore, the overall time complexity is O(n^2 \* n^2) = O(n^4).

Space complexity: O(N^4)

The space complexity of the algorithm is also O(n^4), due to the use of the unordered map to store previously computed substrings. In the worst case, the map can store all possible substrings of the two strings, which takes O(n^4) space. Additionally, the algorithm uses three arrays to keep track of the frequency of characters in the two strings and the current substring being checked, which also takes O(n^3) space in the worst case. Therefore, the overall space complexity is O(n^4).

However, the use of the unordered map to store previously computed substrings allows the algorithm to avoid recomputing the same substrings multiple times, which can significantly improve the performance of the algorithm for large inputs.

**Time Complexity analysis:**

**Refer to**

<https://leetcode.com/problems/scramble-string/solutions/3357574/day-364-100-java-c-python-explained-intution-dry-run-proof/>

##### • The time complexity of the algorithm is O(n^4), n is the length of the strings. This is because the algorithm checks all possible splits of the two strings, which takes O(n^2) time, and for each split, it recursively checks if the substrings are scrambled versions of each other, which takes O(n^2) time in the worst case. Therefore, the overall time complexity is O(n^2 \* n^2) = O(n^4).

##### • The space complexity of the algorithm is also O(n^4), due to the use of the unordered map to store previously computed substrings. In the worst case, the map can store all possible substrings of the two strings, which takes O(n^4) space. Additionally, the algorithm uses three arrays to keep track of the frequency of characters in the two strings and the current substring being checked, which also takes O(n^3) space in the worst case. Therefore, the overall space complexity is O(n^4).

##### • However, the use of the unordered map to store previously computed substrings allows the algorithm to avoid recomputing the same substrings multiple times, which can significantly improve the performance of the algorithm for large inputs.

**Refer to**

<https://leetcode.wang/leetCode-87-Scramble-String.html>

当然，我们可以用 memoization 技术，把递归过程中的结果存储起来，如果第二次递归过来，直接返回结果即可，无需重复递归。

public boolean isScramble(String s1, String s2) {

HashMap<String, Integer> memoization = new HashMap<>();

return isScrambleRecursion(s1, s2, memoization);

}

public boolean isScrambleRecursion(String s1, String s2, HashMap<String, Integer> memoization) {

//判断之前是否已经有了结果

int ret = memoization.getOrDefault(s1 + "#" + s2, -1);

if (ret == 1) {

return true;

} else if (ret == 0) {

return false;

}

if (s1.length() != s2.length()) {

memoization.put(s1 + "#" + s2, 0);

return false;

}

if (s1.equals(s2)) {

memoization.put(s1 + "#" + s2, 1);

return true;

}

int[] letters = new int[26];

for (int i = 0; i < s1.length(); i++) {

letters[s1.charAt(i) - 'a']++;

letters[s2.charAt(i) - 'a']--;

}

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

if (letters[i] != 0) {

memoization.put(s1 + "#" + s2, 0);

return false;

}

for (int i = 1; i < s1.length(); i++) {

if (isScramble(s1.substring(0, i), s2.substring(0, i)) && isScramble(s1.substring(i), s2.substring(i))) {

memoization.put(s1 + "#" + s2, 1);

return true;

}

if (isScramble(s1.substring(0, i), s2.substring(s2.length() - i))

&& isScramble(s1.substring(i), s2.substring(0, s2.length() - i))) {

memoization.put(s1 + "#" + s2, 1);

return true;

}

}

memoization.put(s1 + "#" + s2, 0);

return false;

}

**Style 2: 基于Solution 1的Style 2每层循环都是传递的实际获取的字符串s1和s2的情况，在memo中也存储字符串本身**

class Solution {

public boolean isScramble(String s1, String s2) {

if(s1.length() != s2.length()) {

return false;

}

int n = s1.length();

Boolean[][][] memo = new Boolean[n + 1][n][n];

return helper(s1, s2, 0, 0, n, memo);

}

private boolean helper(String s1, String s2, int i, int j, int len, Boolean[][][] memo) {

// No substring length check required anymore since

// on each recursion level pass in same length 'len'

// which will cut out same length substring on s1, s2

// Base condition when exceed length

if(i + len > s1.length() || j + len > s1.length()) {

return memo[len][i][j] = false;

}

// Base condition

if(s1.substring(i, i + len).equals(s2.substring(j, j + len))) {

return memo[len][i][j] = true;

}

// Exist in memo means calculated before

if(memo[len][i][j] != null) {

return memo[len][i][j];

}

// Check if s1[i:i+len-1] and s2[j:j+len-1] have same characters

int[] freq = new int[26];

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

freq[s1.charAt(i + k) - 'a']++;

freq[s2.charAt(j + k) - 'a']--;

}

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

if(freq[k] != 0) {

return memo[len][i][j] = false;

}

}

for(int k = 1; k < len; k++) {

// There exist a cut point to split current s1 into s11 and s12,

// also split current s2 into s21 and s22, where s11 and s21 are

// scramble, and s12 and s22 are also scramble

// e.g s1 = "great", s2 = "rgtae"

// s11 = "gr", s12 = "eat", s21 = "rg", s22 = "tae"

// In general, we will check these two pairs if both scramble in next

// recursion with base condition:

// s1.substring(i, i + len).equals(s2.substring(j, j + len))

// and if not equal will continue to check if same characters with freq map

// Pair1(s11 vs s21):

// s11 = s1[i:i+k), s21 = s2[j:j+k) -> helper(s1, s2, i, j, k)

// Pair2(s12 vs s22):

// s12 = s1[i+k:i+len), s22 = s2[j+k:j+len) -> helper(s1, s2, i + k, j + k, len - k)

if(helper(s1, s2, i, j, k, memo) && helper(s1, s2, i + k, j + k, len - k, memo)) {

return memo[len][i][j] = true;

}

// OR s11 and s22 are scramble, and s12 and s21 are also scramble

// e.g s1 = "great", s2 = "taerg"

// s11 = "gr", s12 = "eat", s21 = "tae", s22 = "rg" (before swap s21, s22)

// but be careful, after split s2 into s21 and s22 by cut point

// we have to swap two substrings, the s2\_new = s22 + s21, and

// also the cut point shift, the length of s22 equals s11 and

// the length of s21 equals s12

// In general, we will check these two pairs if both scramble in next

// recursion with base condition:

// s1.substring(i, i + len).equals(s2.substring(j, j + len))

// and if not equal will continue to check if same characters with freq map

// Pair1(s11 vs s22):

// s11 = s1[i:i+k), s22 = s2[j+len-k:j+len) -> helper(s1, s2, i, j + len - k, k)

// Pair2(s12 vs s21):

// s12 = s1[i+k:i+len), s21 = s2[j:j+len-k) -> helper(s1, s2, i + k, j, len - k)

if(helper(s1, s2, i, j + len - k, k, memo) && helper(s1, s2, i + k, j, len - k, memo)) {

return memo[len][i][j] = true;

}

}

return memo[len][i][j] = false;

}

}

Time complexity: O(N^4), where N is the length of the input strings. This is because we have three nested loops, and the length of the substrings can be up to N.

Space complexity: O(N^3)

**Refer to**

<https://leetcode.com/problems/scramble-string/solutions/3358560/dynamic-programming-solution-to-check-if-strings-are-scrambled-with-easy-and-full-explanation/>

# **Intuition**

The problem can be solved using a dynamic programming approach where we check if the substrings of s1 and s2 are scrambled strings or not. We start by checking the base cases, which is when the length of the substring is 1. We then recursively check all possible split points and if the substrings are swapped or not.

# **Approach**

The solution uses a dynamic programming approach to solve the problem. Let dp[i][j][len] be true if the substring s1[i:i+len] is a scrambled string of the substring s2[j:j+len], and false otherwise.

The base case is when len = 1, in which case we just need to check if the characters in the substrings are equal. If they are, then we set dp[i][j][1] to true, otherwise to false.

For the recursive case, we try all possible split points of the substrings s1[i:i+len] and s2[j:j+len]. If the substrings are split at position k, we have two possibilities: either the substrings are not swapped, or they are swapped. If they are not swapped, then we need to check if the left halves and right halves are scrambled strings, i.e., dp[i][j][k] && dp[i+k][j+k][len-k]. If they are swapped, then we need to check if the left half of s1 and the right half of s2 are scrambled strings, and if the right half of s1 and the left half of s2 are scrambled strings, i.e., dp[i][j+len-k][k] && dp[i+k][j][len-k].

The overall solution is the value of dp[0][0][n], where n is the length of the input strings.

# **Complexity**

* Time complexity: The time complexity of the solution is O(n^4), where n is the length of the input strings. This is because we have three nested loops, and the length of the substrings can be up to n.
* Space complexity: The space complexity of the solution is O(n^3), where n is the length of the input strings. This is because we need to store the results of all possible substring combinations in the 3D array dp.

# **Code**

class Solution {

public:

bool isScramble(string s1, string s2) {

if (s1 == s2) {

return true;

}

int n = s1.length();

if (n != s2.length()) {

return false;

}

vector<int> count(26, 0);

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

count[s1[i] - 'a']++;

count[s2[i] - 'a']--;

}

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

if (count[i] != 0) {

return false;

}

}

vector<vector<vector<int>>> dp(n, vector<vector<int>>(n, vector<int>(n + 1, -1)));

return isScrambleHelper(s1, s2, 0, 0, n, dp);

}

bool isScrambleHelper(string& s1, string& s2, int i, int j, int len, vector<vector<vector<int>>>& dp) {

if (dp[i][j][len] != -1) {

return dp[i][j][len];

}

if (s1.substr(i, len) == s2.substr(j, len)) {

return dp[i][j][len] = true;

}

vector<int> count(26, 0);

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

count[s1[i + k] - 'a']++;

count[s2[j + k] - 'a']--;

}

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

if (count[k] != 0) {

return dp[i][j][len] = false;

}

}

for (int k = 1; k < len; k++) {

if ((isScrambleHelper(s1, s2, i, j, k, dp) && isScrambleHelper(s1, s2, i + k, j + k, len - k, dp)) ||

(isScrambleHelper(s1, s2, i, j + len - k, k, dp) && isScrambleHelper(s1, s2, i + k, j, len - k, dp))) {

return dp[i][j][len] = true;

}

}

return dp[i][j][len] = false;

}

};

**Solution 3: DP (120min)**

class Solution {

public boolean isScramble(String s1, String s2) {

if(s1.length() != s2.length()) {

return false;

}

if(s1.equals(s2)) {

return true;

}

int n = s1.length();

int[] freq = new int[26];

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

freq[s1.charAt(i) - 'a']++;

freq[s2.charAt(i) - 'a']--;

}

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

if(freq[i] != 0) {

return false;

}

}

// dp[s1/s2 length()][s1 start index][s2 start index]

// In Native DFS we have

// top: {n, 0, 0} -> bottom: {1, n - len, n - 1en}

// For loop start with dp[1][n - len][n - len], finally return dp[n][0][0]

// dp[len][i][j]表示s1[i, i + len)和s2[j, j + len)两个字符串是否满足条件

boolean[][][] dp = new boolean[n + 1][n][n];

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

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

for(int j = n - len; j >= 0; j--) {

// 长度是1无需切割(Native DFS中并无专门对应语句)

if(len == 1) {

dp[len][i][j] = s1.charAt(i) == s2.charAt(j);

} else {

for(int k = 1; k < len; k++) {

// helper(s1, s2, i, j, k) && helper(s1, s2, i + k, j + k, len - k)

// helper(s1, s2, i, j + len - k, k) && helper(s1, s2, i + k, j, len - k)

if(dp[k][i][j] && dp[len - k][i + k][j + k] || dp[k][i][j + len - k] && dp[len - k][i + k][j]) {

dp[len][i][j] = true;

}

}

}

}

}

}

return dp[n][0][0];

}

}

Time complexity: O(N^4)

Space complexity: O(N^3)

**Refer to**

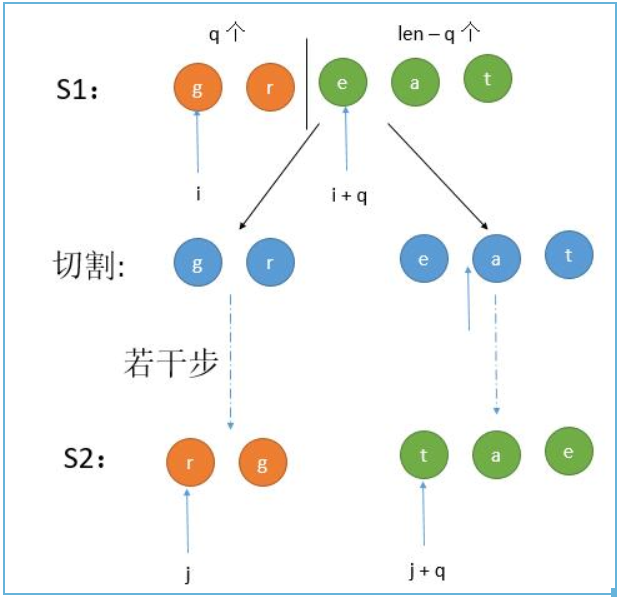
<https://leetcode.wang/leetCode-87-Scramble-String.html>

# **解法二 动态规划**

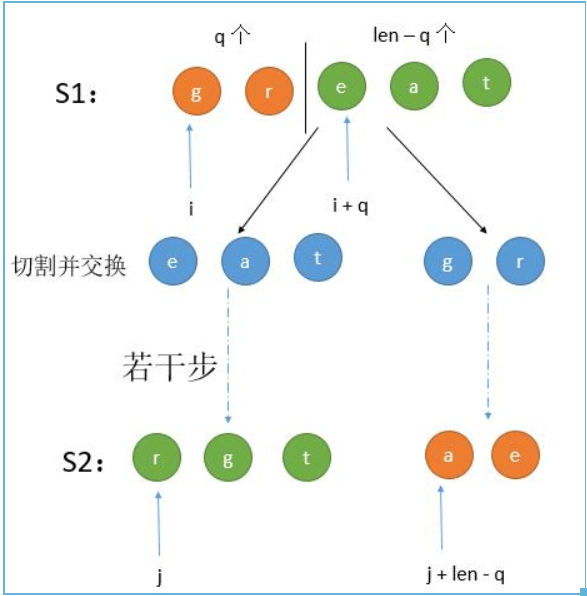
既然是递归，压栈压栈压栈，出栈出栈出栈，我们可以利用动态规划的思想，省略压栈的过程，直接从底部往上走。

我们用 dp [ len ][ i ] [ j ] 来表示 s1[ i, i + len ) 和 s2 [ j, j + len ) 两个字符串是否满足条件。换句话说，就是 s1 从 i 开始的 len 个字符是否能转换为 S2 从 j 开始的 len 个字符。那么解法一的两种情况，递归式可以写作。

第 1 种情况，参考下图： 假设左半部分长度是 q，dp [ len ][ i ] [ j ] = dp [ q ][ i ] [ j ] && dp [ len - q ][ i + q ] [ j + q ] 。也就是 S1 的左半部分和 S2 的左半部分以及 S1 的右半部分和 S2 的右半部分。



第 2 种情况，参考下图： 假设左半部分长度是 q，那么 dp [ len ][ i ] [ j ] = dp [ q ][ i ] [ j + len - q ] && dp [ len - q ][ i + q ] [ j ] 。也就是 S1 的右半部分和 S2 的左半部分以及 S1 的左半部分和 S2 的右半部分。



public boolean isScramble4(String s1, String s2) {

if (s1.length() != s2.length()) {

return false;

}

if (s1.equals(s2)) {

return true;

}

int[] letters = new int[26];

for (int i = 0; i < s1.length(); i++) {

letters[s1.charAt(i) - 'a']++;

letters[s2.charAt(i) - 'a']--;

}

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

if (letters[i] != 0) {

return false;

}

}

int length = s1.length();

boolean[][][] dp = new boolean[length + 1][length][length];

//遍历所有的字符串长度

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

//S1 开始的地方

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

//S2 开始的地方

for (int j = 0; j + len <= length; j++) {

//长度是 1 无需切割

if (len == 1) {

dp[len][i][j] = s1.charAt(i) == s2.charAt(j);

} else {

//遍历切割后的左半部分长度

for (int q = 1; q < len; q++) {

dp[len][i][j] = dp[q][i][j] && dp[len - q][i + q][j + q]

|| dp[q][i][j + len - q] && dp[len - q][i + q][j];

//如果当前是 true 就 break，防止被覆盖为 false

if (dp[len][i][j]) {

break;

}

}

}

}

}

}

return dp[length][0][0];

}

时间复杂度: O(N^4)

空间复杂度: O(N^3)

**Refer to**

<https://grandyang.com/leetcode/87/>

当然，这道题也可以用动态规划 Dynamic Programming，根据以往的经验来说，根字符串有关的题十有八九可以用 DP 来做，那么难点就在于如何找出状态转移方程。参见网友 [Code Ganker 的博客](http://blog.csdn.net/linhuanmars/article/details/24506703)，这其实是一道三维动态规划的题目，使用一个三维数组 dp[i][j][n]，其中i是 s1 的起始字符，j是 s2 的起始字符，而n是当前的字符串长度，dp[i][j][len] 表示的是以i和j分别为 s1 和 s2 起点的长度为 len 的字符串是不是互为 scramble。有了 dp 数组接下来看看状态转移方程，也就是怎么根据历史信息来得到 dp[i][j][len]。判断这个是不是满足，首先是把当前 s1[i…i+len-1] 字符串劈一刀分成两部分，然后分两种情况：第一种是左边和 s2[j…j+len-1] 左边部分是不是 scramble，以及右边和 s2[j…j+len-1] 右边部分是不是 scramble；第二种情况是左边和 s2[j…j+len-1] 右边部分是不是 scramble，以及右边和 s2[j…j+len-1] 左边部分是不是 scramble。如果以上两种情况有一种成立，说明 s1[i…i+len-1] 和 s2[j…j+len-1] 是 scramble 的。而对于判断这些左右部分是不是 scramble 是有历史信息的，因为长度小于n的所有情况都在前面求解过了（也就是长度是最外层循环）。上面说的是劈一刀的情况，对于 s1[i…i+len-1] 有 len-1 种劈法，在这些劈法中只要有一种成立，那么两个串就是 scramble 的。总结起来状态转移方程是：

dp[i][j][len] = || (dp[i][j][k] && dp[i+k][j+k][len-k] || dp[i][j+len-k][k] && dp[i+k][j][len-k])

对于所有 1<=k<len，也就是对于所有 len-1 种劈法的结果求或运算。因为信息都是计算过的，对于每种劈法只需要常量操作即可完成，因此求解递推式是需要 O(len)（因为 len-1 种劈法）。如此总时间复杂度因为是三维动态规划，需要三层循环，加上每一步需要线行时间求解递推式，所以是 O(n^4)。虽然已经比较高了，但是至少不是指数量级的，动态规划还是有很大优势的，空间复杂度是 O(n^3)。代码如下：

class Solution {

public:

bool isScramble(string s1, string s2) {

if (s1.size() != s2.size()) return false;

if (s1 == s2) return true;

int n = s1.size();

vector<vector<vector<bool>>> dp (n, vector<vector<bool>>(n, vector<bool>(n + 1)));

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

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

for (int j = 0; j <= n - len; ++j) {

if (len == 1) {

dp[i][j][1] = s1[i] == s2[j];

} else {

for (int k = 1; k < len; ++k) {

if ((dp[i][j][k] && dp[i + k][j + k][len - k]) || (dp[i + k][j][len - k] && dp[i][j + len - k][k])) {

dp[i][j][len] = true;

}

}

}

}

}

}

return dp[0][0][n];

}

};