<https://leetcode.com/problems/palindrome-partitioning/>

Given a string s, partition s such that every substring of the partition is a **palindrome**. Return *all possible palindrome partitioning of*s.

**Example 1:**

Input: s = "aab"

Output: [["a","a","b"],["aa","b"]]

**Example 2:**

Input: s = "a"

Output: [["a"]]

**Constraints:**

* 1 <= s.length <= 16
* s contains only lowercase English letters.

**Attempt 1: 2023-01-10**

**Solution 1:  DFS + Backtracking (30 min)**

**Style 1: No 'startIndex' in recursion call and truncate on original input String (more intuitive and readable)**

class Solution {

public List<List<String>> partition(String s) {

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

helper(s, new ArrayList<String>(), result);

return result;

}

private void helper(String s, List<String> list, List<List<String>> result) {

if(s.length() == 0) {

result.add(new ArrayList<String>(list));

return;

}

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

String cur = s.substring(0, i);

String next = s.substring(i);

// If current substring (section index from 0 to i - 1) is a palindrome

// then move on to check the remain substring (section index from i to end)

if(isPalindrome(s, 0, i - 1)) {

list.add(cur);

helper(next, list, result);

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

}

}

}

private boolean isPalindrome(String s, int start, int end) {

while(start < end) {

if(s.charAt(start) != s.charAt(end)) {

return false;

}

start++;

end--;

}

return true;

}

}

**Refer to**

<https://leetcode.com/problems/palindrome-partitioning/solutions/182307/java-backtracking-template-general-approach/>

All backtracking problems are composed by these three steps: choose, explore, unchoose. So for each problem, you need to know:

1. choose what? For this problem, we choose each substring.
2. how to explore? For this problem, we do the same thing to the remained string.
3. unchoose Do the opposite operation of choose.

Let's take this problem as an example:1.Define helper(): Usually we need a helper function in backtracking problem, to accept more parameters.2.Parameters: Usually we need the following parameters

1. The object you are working on: For this problem is String s.

2. A start index or an end index which indicate which part you are working on: For this problem, we use substring to indicate the start index.

3. A step result, to remember current choose and then do unchoose : For this problem, we use List<String> step.

4. A final result, to remember the final result. Usually when we add, we use 'result.add(new ArrayList<>(step))' instead of 'result.add(step)', since step is reference passed. We will modify step later, so we need to copy it and add the copy to the result;

3.Base case: The base case defines when to add step into result, and when to return.

4.Use for-loop : Usually we need a for loop to iterate though the input String s, so that we can choose all the options.

5.Choose : In this problem, if the substring of s is palindrome, we add it into the step, which means we choose this substring.

6.Explore : In this problem, we want to do the same thing to the remaining substring. So we recursively call our function.

7.Un-Choose : We draw back, remove the chosen substring, in order to try other options.

The above is mainly the template, the code is shown below:

public List<List<String>> partition(String s) {

// Backtracking

// Edge case

if(s == null || s.length() == 0) return new ArrayList<>();

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

helper(s, new ArrayList<>(), result);

return result;

}

public void helper(String s, List<String> step, List<List<String>> result) {

// Base case

if(s == null || s.length() == 0) {

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

return;

}

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

String temp = s.substring(0, i);

if(!isPalindrome(temp)) continue; // only do backtracking when current string is palindrome

step.add(temp); // choose

helper(s.substring(i, s.length()), step, result); // explore

step.remove(step.size() - 1); // unchoose

}

return;

}

public boolean isPalindrome(String s) {

int left = 0, right = s.length() - 1;

while(left <= right) {

if(s.charAt(left) != s.charAt(right))

return false;

left ++;

right --;

}

return true;

}

**Style 2: Have 'startIndex' and no truncate on original input String in recursion call**

class Solution {

public List<List<String>> partition(String s) {

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

helper(s, 0, new ArrayList<String>(), result);

return result;

}

private void helper(String s, int startIndex, List<String> list, List<List<String>> result) {

if(startIndex == s.length()) {

result.add(new ArrayList<String>(list));

return;

}

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

if(isPalindrome(s, startIndex, i)) {

list.add(s.substring(startIndex, i + 1));

helper(s, i + 1, list, result);

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

}

}

}

private boolean isPalindrome(String s, int start, int end) {

while(start < end) {

if(s.charAt(start) != s.charAt(end)) {

return false;

}

start++;

end--;

}

return true;

}

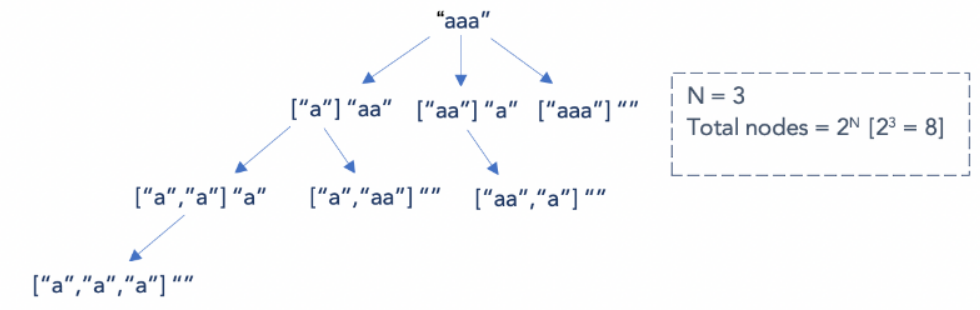
}

**For Style 2 we can draw a picture like below**

**Complexity Analysis**

* Time Complexity : O(N⋅2^N), where N is the length of string s. This is the worst-case time complexity when all the possible substrings are palindrome.

Example, if s is a, the recursive tree can be illustrated as follows:



Hence, there could be 2^N possible substrings in the worst case. For each substring, it takes O(N) time to generate the substring and determine if it is a palindrome or not. This gives us a time complexity of O(N⋅2^N)

* Space Complexity: O(N), where N is the length of the string s. This space will be used to store the recursion stack. For s = a, the maximum depth of the recursive call stack is 3 which is equivalent to N.