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*.

For example, given *s* = `"aab"`,
Return

```
[
  ["aa","b"],
  ["a","a","b"]
]
```

Solution 1

```java
public class Solution {
    public static List<List<String>> partition(String s) {
        int len = s.length();
        List<List<String>>[] result = new List[len + 1];
        result[0] = new ArrayList<List<String>>();
        result[0].add(new ArrayList<String>());

        boolean[][] pair = new boolean[len][len];
        for (int i = 0; i < s.length(); i++) {
            result[i + 1] = new ArrayList<List<String>>();
            for (int left = 0; left <= i; left++) {
                if (s.charAt(left) == s.charAt(i) && (i-left <= 1 || pair[left +
1][i - 1])) {
                    pair[left][i] = true;
                    String str = s.substring(left, i + 1);
                    for (List<String> r : result[left]) {
                        List<String> ri = new ArrayList<String>(r);
                        ri.add(str);
                        result[i + 1].add(ri);
                    }
                }
            }
        }
        return result[len];
    }
}
```

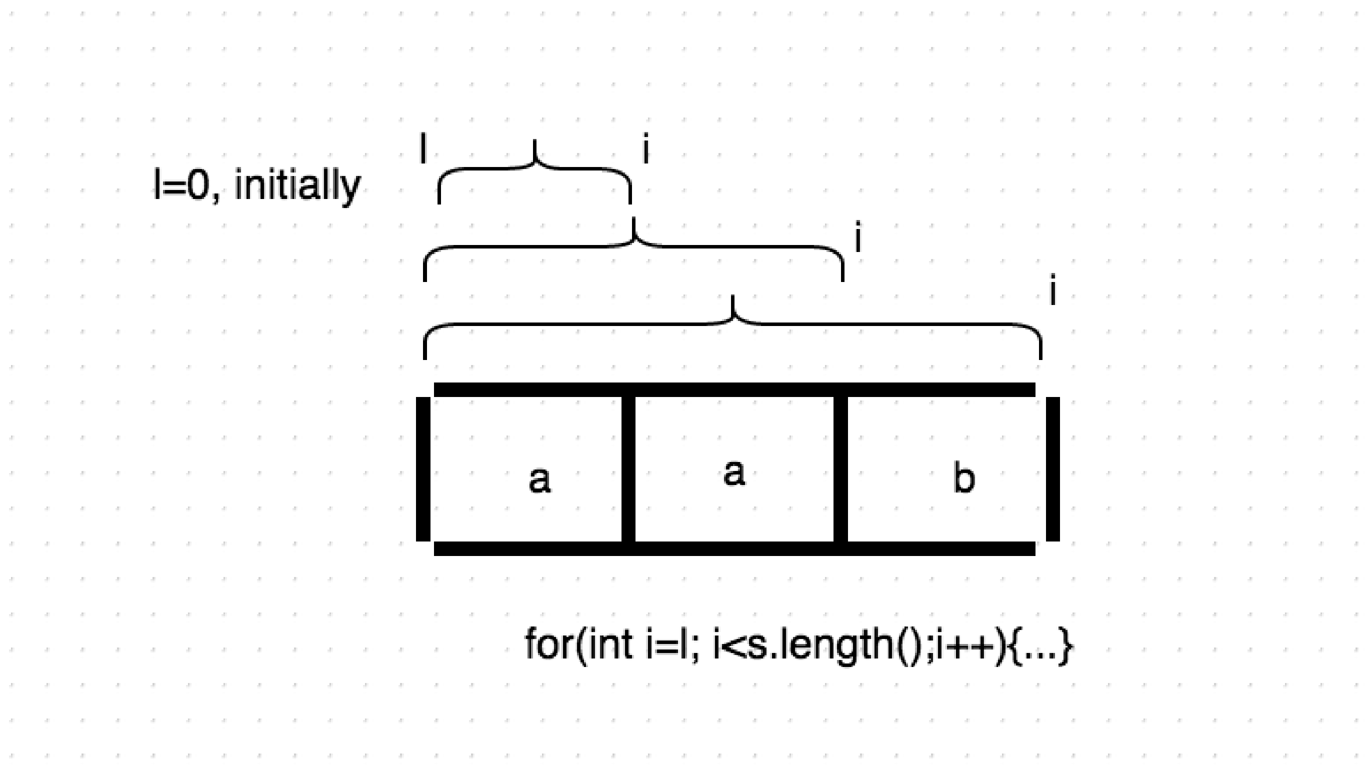
Here the **pair** is to mark a range for the substring is a Pal. if pair[i][j] is true, that means sub string from i to j is pal.

The **result[i]**, is to store from beginng until current index i (Non inclusive), all possible partitions. From the past result we can determine current result.

written by jianwu original link here

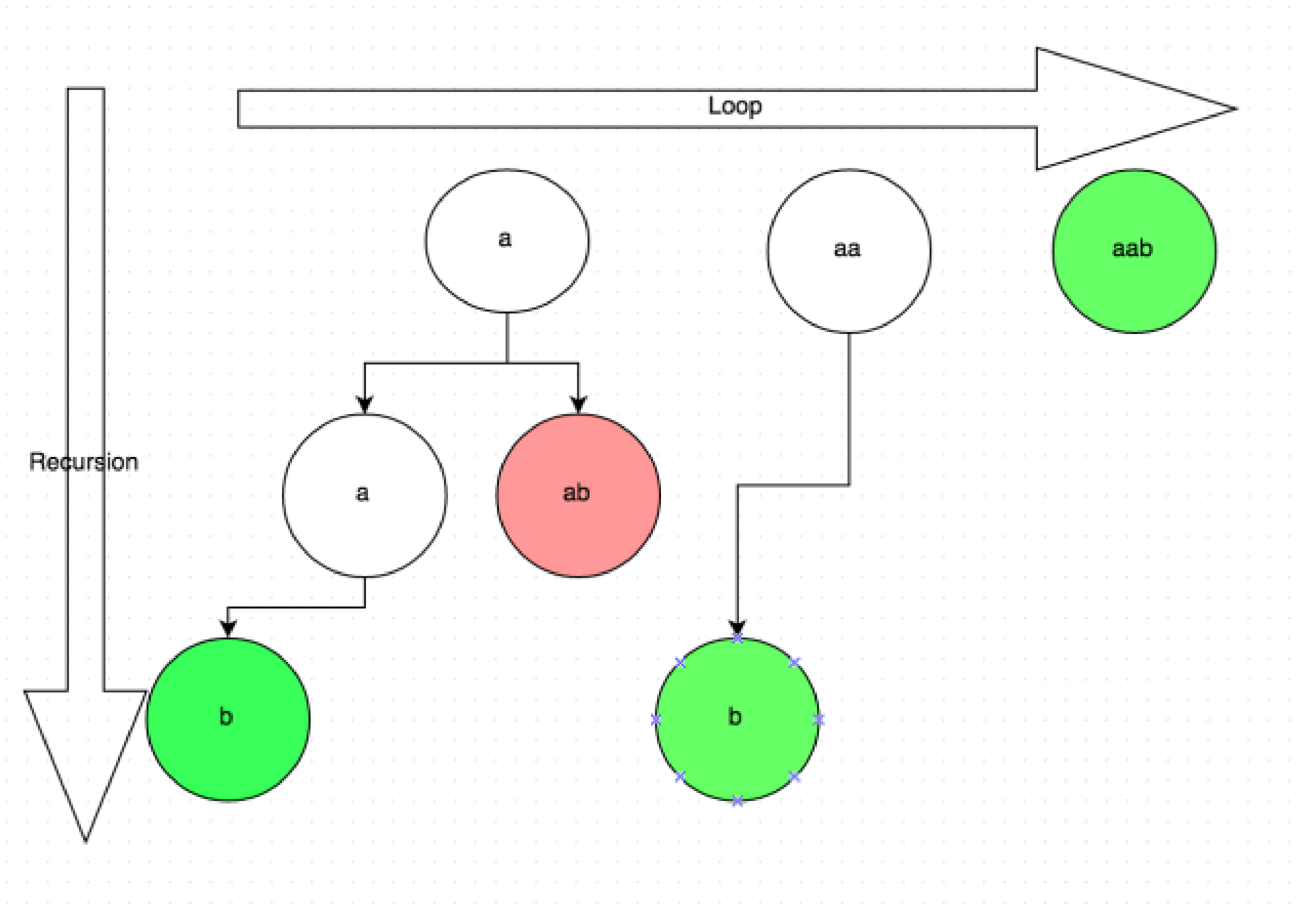
Solution 2

if the input is "aab", check if [0,0] "a" is palindrome. then check [0,1] "aa", then [0,2] "aab". While checking [0,0], the rest of string is "ab", use ab as input to make a recursive call.



in this example, in the loop of i=l+1, a recursive call will be made with input = "ab". Every time a recursive call is made, the position of l move right.

How to define a correct answer? Think about DFS, if the current string to be checked (Palindrome) contains the last position, in this case "c", this path is a correct answer, otherwise, it's a false answer.



line 13: is the boundary to check if the current string contains the last element. l>=s.length()

```
public class Solution {
        List<List<String>> resultLst;
        ArrayList<String> currLst;
        public List<List<String>> partition(String s) {
            resultLst = new ArrayList<List<String>>();
            currLst = new ArrayList<String>();
            backTrack(s,0);
            return resultLst;
        }
        public void backTrack(String s, int l){
            if(currLst.size()>0 //the initial str could be palindrome
                && l>=s.length()){
                    List<String> r = (ArrayList<String>) currLst.clone();
                    resultLst.add(r);
            }
            for(int i=l;i<s.length();i++){
                if(isPalindrome(s,l,i)){
                    if(l==i)
                        currLst.add(Character.toString(s.charAt(i)));
                    else
                        currLst.add(s.substring(l,i+1));
                    backTrack(s,i+1);
                    currLst.remove(currLst.size()-1);
                }
            }
        }
        public boolean isPalindrome(String str, int l, int r){
            if(l==r) return true;
            while(l<r){
                if(str.charAt(l)!=str.charAt(r)) return false;
                l++;r--;
            }
            return true;
        }
}
```

written by charlie+yupeng original link here

Solution 3

The Idea is simple: loop through the string, check if substr(0, i) is palindrome. If it is, recursively call dfs() on the rest of sub string: substr(i+1, length). keep the current palindrome partition so far in the 'path' argument of dfs(). When reaching the end of string, add current partition in the result.

```
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string> > ret;
        if(s.empty()) return ret;

        vector<string> path;
        dfs(0, s, path, ret);

        return ret;
    }

    void dfs(int index, string& s, vector<string>& path, vector<vector<string> >&
ret) {
        if(index == s.size()) {
            ret.push_back(path);
            return;
        }
        for(int i = index; i < s.size(); ++i) {
            if(isPalindrome(s, index, i)) {
                path.push_back(s.substr(index, i - index + 1));
                dfs(i+1, s, path, ret);
                path.pop_back();
            }
        }
    }

    bool isPalindrome(const string& s, int start, int end) {
        while(start <= end) {
            if(s[start++] != s[end--])
                return false;
        }
        return true;
    }
};
```

written by zhangyu917 original link here