267. Palindrome Permutation II

Previous Next

June 11, 2017 | 25.9K views
Average Rating: 4.21 (24 votes)

Given a string `s`, return all the palindromic permutations (without duplicates) of it. Return an empty list if no palindromic permutation could be form.

Example 1:

```
Input: "aabb"
Output: ["abba", "baab"]
```

Example 2:

```
Input: "abc"
Output: []
```

Solution

Approach #1 Brute Force [Time Limit Exceeded]

The simplest solution is generate every possible permutation of the given string s and check if the generated permutation is a palindrome. After this, the palindromic permutations can be added to a set in order to eliminate the duplicates. At the end, we can return an array comprised of the elements of this set as the resultant array.

Let's look at the way these permutations are generated. We make use of a recursive function `permute` which takes the index of the current element $current_index$ as one of the arguments. Then, it swaps the current element with every other element in the array, *lying towards its right*, so as to generate a new ordering of the array elements. After the swapping has been done, it makes another call to `permute` but this time with the index of the next element in the array. While returning back, we reverse the swapping done in the current function call. Thus, when we reach the end of the array, a new ordering of the array's elements is generated.

The animation below depicts the ways the permutations are generated.

Java

Copy

```
public class Solution {
    Set < String > set = new HashSet < > ();
    public List < String > generatePalindromes(String s) {
        permute(s.toCharArray(), 0);
        return new ArrayList < String > (set);
    }
    public boolean isPalindrome(char[] s) {
        for (int i = 0; i < s.length; i++) {
            if (s[i] != s[s.length - 1 - i])
                return false;
        }
        return true;
    }
    public void swap(char[] s, int i, int j) {
        char temp = s[i];
        s[i] = s[j];
        s[j] = temp;
    }
    void permute(char[] s, int i) {
        if (i == s.length) {
            if (isPalindrome(s))
                set.add(new String(s));
        } else {
            for (int k = i; k < s.length; k++) {
                swap(s, i, k);
                permute(s, i + 1);
                swap(s, i, k);
            }
```

Complexity Analysis

- Time complexity : $O((n+1)!)$. A total of $n!$ permutations are possible. For every permutation generated, we need to check if it is a palindrome, each of which requires $O(n)$ time.
- Space complexity : $O(n)$. The depth of the recursion tree can go upto n.

Approach #2 Backtracking [Accepted]

Algorithm

It might be possible that no palindromic permutation could be possible for the given string s. Thus, it is useless to generate the permutations in such a case. Taking this idea, firstly we check if a palindromic permutation is possible for s. If yes, then only we proceed further with generating the permutations. To check this, we make use of a hashmap map which stores the number of occurences of each character(out of 128 ASCII characters possible). If the number of characters with odd number of occurences exceeds 1, it indicates that no palindromic permutation is possible for s. To look at this checking process in detail, look at Approach 4 of the article for Palindrome Permutation.

Once we are sure that a palindromic permutation is possible for s, we go for the generation of the required permutations. But, instead of wildly generating all the permutations, we can include some smartness in the generation of permutations i.e. we can generate only those permutations which are already palindromes.

One idea to to do so is to generate only the first half of the palindromic string and to append its reverse string to itself to generate the full length palindromic string.

Based on this idea, by making use of the number of occurences of the characters in s stored in map, we create a string st which contains all the characters of s but with the number of occurences of these characters in st reduced to half their original number of occurences in s.

Thus, now we can generate all the permutations of this string st and append the reverse of this permuted string to itself to create the palindromic permutations of s.

In case of a string s with odd length, whose palindromic permutations are possible, one of the characters in s must be occuring an odd number of times. We keep a track of this character, ch, and it is kept separte from the string st. We again generate the permutations for st similarly and append the reverse of the generated permutation to itself, but we also place the character ch at the middle of the generated string.

In this way, only the required palindromic permutations will be generated. Even if we go with the above idea, a lot of duplicate strings will be generated.

In order to avoid generating duplicate palindromic permutations in the first place itself, as much as possible, we can make use of this idea. As discussed in the last approach, we swap the current element with all the elements lying towards its right to generate the permutations. Before swapping, we can check if the elements being swapped are equal. If so, the permutations generated even after swapping the two will be duplicates(redundant). Thus, we need not proceed further in such a case.

See this animation for a clearer understanding.

Java

Copy

```
public class Solution {
    Set < String > set = new HashSet < > ();
    public List < String > generatePalindromes(String s) {
        int[] map = new int[128];
        char[] st = new char[s.length() / 2];
        if (!canPermutePalindrome(s, map))
            return new ArrayList < > ();
        char ch = 0;
        int k = 0;
        for (int i = 0; i < map.length; i++) {
            if (map[i] % 2 == 1)
                ch = (char) i;
            for (int j = 0; j < map[i] / 2; j++) {
                st[k++] = (char) i;
            }
        }
        permute(st, 0, ch);
        return new ArrayList < String > (set);
    }
    public boolean canPermutePalindrome(String s, int[] map) {
        int count = 0;
        for (int i = 0; i < s.length(); i++) {
            map[s.charAt(i)]++;
            if (map[s.charAt(i)] % 2 == 0)
                count--;
            else
                count++;
        }
```

Complexity Analysis

- Time complexity : $O((\frac{n}{2}+1)!)$. Almost $\frac{n}{2}!$ permutations need to be generated in the worst case. Further, for each permutation generated, `string.reverse()` function will take $n/4$ time.
- Space complexity : $O(n)$. The depth of recursion tree can go upto $n/2$ in the worst case.

Rate this article:

Previous Next

Comments: 8 Sort By

Type comment here... (Markdown is supported)

Preview Post

sankalj ★ 49 February 12, 2020 2:06 PM

Anyone else think this should be a hard problem rather than medium?

36 Share Reply

jichen11227 ★ 20 March 2, 2019 10:45 AM

Thanks for the solution! I think one more thing that can make your code easier to read is to have a good var name. s, i, ch is very hard to remember

17 Share Reply

SHOW 1 REPLY

ashish53v ★ 313 January 17, 2018 10:16 AM

My easy Java solution

```
class Solution {
    public List<String> generatePalindromes(String s) {
```

Read More

11 Share Reply

jeqiye2003cn ★ 2 April 29, 2020 5:45 PM

I feel the problems in backtracking category are in general more difficult. Their easy seems like a medium problem, medium are more of hard problems, and hard are not more difficult, but takes more whiteboard space than medium ones

1 Share Reply

jimhorng ★ 5 January 23, 2018 6:48 AM

Since the `Set < String > set = new HashSet < > ();` need to store all the permutation before return, so Space Complexity is at most O(n/2+1)), which is the same as time complexity, right?

1 Share Reply

SHOW 1 REPLY

RogerFederer ★ 855 December 17, 2017 5:27 AM

```
class Solution(object):
    def permutePalindromesDetail(self, compressed, odd_char, result, tmp):
        if not compressed:
```

Read More

1 Share Reply

SHOW 1 REPLY

s961206 ★ 751 July 24, 2019 12:30 AM

My solution that beats 99.83% (Skip duplicate branch)

```
class Solution {
List res;
String single = "";
```

Read More

0 Share Reply

Ewaole ★ 10 April 8, 2019 9:24 AM

Thanks for the clear explanation. Just one query, in function "permute", what's the necessity of using set, if it is already handling the duplicate permutations. I might be missing something, a small clarification will be helpful. Thanks.

0 Share Reply

SHOW 1 REPLY