Previous Next

266. Palindrome Permutation

June 6, 2017 | 54.8K views

Average Rating: 4.53 (45 votes)

Given a string, determine if a permutation of the string could form a palindrome.

Example 1:

```
Input: "code"
Output: false
```

Example 2:

```
Input: "aab"
Output: true
```

Example 3:

```
Input: "carerac"
Output: true
```

Solution

Approach #1 Brute Force [Accepted]

If a string with an even length is a palindrome, every character in the string must always occur an even number of times. If the string with an odd length is a palindrome, every character except one of the characters must always occur an even number of times. Thus, in case of a palindrome, the number of characters with odd number of occurrences can't exceed 1(1 in case of odd length and 0 in case of even length).

Based on the above observation, we can find the solution for the given problem. The given string could contain almost all the ASCII characters from 0 to 127. Thus, we iterate over all the characters from 0 to 127. For every character chosen, we again iterate over the given string s and find the number of occurrences, ch, of the current character in s. We also keep a track of the number of characters in the given string s with odd number of occurrences in a variable count.

If, for any character currently considered, its corresponding count, ch, happens to be odd, we increment the value of count, to reflect the same. In case of even value of ch for any character, the count remains unchanged.

If, for any character, the $count$ becomes greater than 1, it indicates that the given string s can't lead to the formation of a palindromic permutation based on the reasoning discussed above. But, if the value of count remains lesser than 2 even when all the possible characters have been considered, it indicates that a palindromic permutation can be formed from the given string s.

Java Copy

```java
public class Solution {
    public boolean canPermutePalindrome(String s) {
        int count = 0;
        for (char i = 0; i < 128 && count <= 1; i++) {
            int ct = 0;
            for (int j = 0; j < s.length(); j++) {
                if (s.charAt(j) == i)
                    ct++;
            }
            count += ct % 2;
        }
        return count <= 1;
    }
}
```

Complexity Analysis

- Time complexity : $O(128 * n)$. We iterate constant number of times(128) over the string s of length n giving a time complexity of $128n$.
- Space complexity : $O(1)$. Constant extra space is used.

Approach #2 Using HashMap [Accepted]

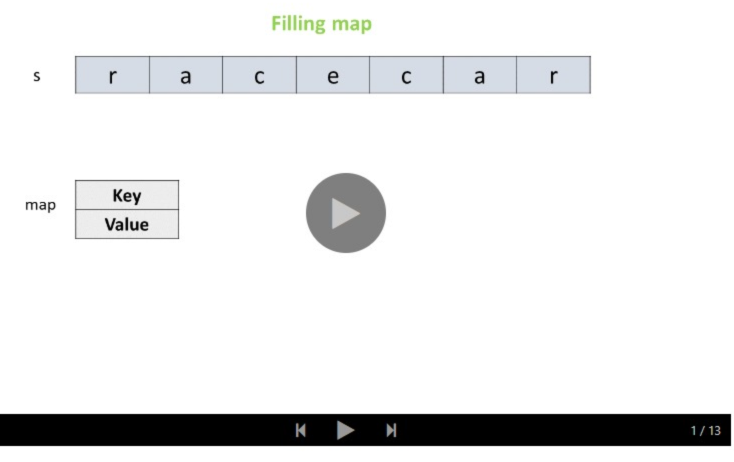
Algorithm

From the discussion above, we know that to solve the given problem, we need to count the number of characters with odd number of occurrences in the given string s. To do so, we can also make use of a hashmap, map. This map takes the form $(\text{character}, \text{number of occurrences of character})$.

We traverse over the given string s. For every new character found in s, we create a new entry in the map for this character with the number of occurrences as 1. Whenever we find the same character again, we update the number of occurrences appropriately.

At the end, we traverse over the map created and find the number of characters with odd number of occurrences. If this count happens to exceed 1 at any step, we conclude that a palindromic permutation isn't possible for the string s. But, if we can reach the end of the string with count lesser than 2, we conclude that a palindromic permutation is possible for s.

The following animation illustrates the process.



Java Copy

```java
public class Solution {
 public boolean canPermutePalindrome(String s) {
     HashMap < Character, Integer > map = new HashMap < > ();
     for (int i = 0; i < s.length(); i++) {
         map.put(s.charAt(i), map.getOrDefault(s.charAt(i), 0) + 1);
     }
     int count = 0;
     for (char key: map.keySet()) {
         count += map.get(key) % 2;
     }
     return count <= 1;
 }
}
```

Complexity Analysis

- Time complexity : $O(n)$. We traverse over the given string s with n characters once. We also traverse over the map which can grow upto a size of n in case all characters in s are distinct.
- Space complexity : $O(1)$. The map can grow up to a maximum number of all distinct elements. However, the number of distinct characters are bounded, so as the space complexity.

Approach #3 Using Array [Accepted]

Algorithm

Instead of making use of the inbuilt Hashmap, we can make use of an array as a hashmap. For this, we make use of an array map with length 128. Each index of this map corresponds to one of the 128 ASCII characters possible.

We traverse over the string s and put in the number of occurrences of each character in this map appropriately as done in the last case. Later on, we find the number of characters with odd number of occurrences to determine if a palindromic permutation is possible for the string s or not as done in previous approaches.

Java Copy

```java
public class Solution {
    public boolean canPermutePalindrome(String s) {
        int[] map = new int[128];
        for (int i = 0; i < s.length(); i++) {
            map[s.charAt(i)]++;
        }
        int count = 0;
        for (int key = 0; key < map.length && count <= 1; key++) {
            count += map[key] % 2;
        }
        return count <= 1;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. We traverse once over the string s of length n. Then, we traverse over the map of length 128(constant).
- Space complexity : $O(1)$. Constant extra space is used for map of size 128.

Approach #4 Single Pass [Accepted]:

Algorithm

Instead of first traversing over the string s for finding the number of occurrences of each element and then determining the count of characters with odd number of occurrences in s, we can determine the value of count on the fly while traversing over s.

For this, we traverse over s and update the number of occurrences of the character just encountered in the map. But, whevenever we update any entry in map, we also check if its value becomes even or odd. We start of with a count value of 0. If the value of the entry just updated in map happens to be odd, we increment the value of count to indicate that one more character with odd number of occurrences has been found. But, if this entry happens to be even, we decrement the value of count to indicate that the number of characters with odd number of occurrences has reduced by one.

But, in this case, we need to traverse till the end of the string to determine the final result, unlike the last approaches, where we could stop the traversal over map as soon as the count exceeded 1. This is because, even if the number of elements with odd number of occurrences may seem very large at the current moment, but their occurrences could turn out to be even when we traverse further in the string s.

At the end, we again check if the value of count is lesser than 2 to conclude that a palindromic permutation is possible for the string s.

Java Copy

```java
public class Solution {
    public boolean canPermutePalindrome(String s) {
        int[] map = new int[128];
        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++;
        }
        return count <= 1;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. We traverse over the string s of length n once only.
- Space complexity : $O(128)$. A map of constant size(128) is used.

Approach #5 Using Set [Accepted]:

Algorithm

Another modification of the last approach could be by making use of a set for keeping track of the number of elements with odd number of occurrences in s. For doing this, we traverse over the characters of the string s. Whenever the number of occurrences of a character becomes odd, we put its entry in the set. Later on, if we find the same element again, lead to its number of occurrences as even, we remove its entry from the set. Thus, if the element occurs again(indicating an odd number of occurrences), its entry won't exist in the set.

Based on this idea, when we find a character in the string s that isn't present in the set(indicating an odd number of occurrences currently for this character), we put its corresponding entry in the set. If we find a character that is already present in the set(indicating an even number of occurrences currently for this character), we remove its corresponding entry from the set.

At the end, the size of set indicates the number of elements with odd number of occurrences in s. If it is lesser than 2, a palindromic permutation of the string s is possible, otherwise not.

Below code is inspired by @StefanPochmann

Java Copy

```java
public class Solution {
    public boolean canPermutePalindrome(String s) {
        Set < Character > set = new HashSet < > ();
        for (int i = 0; i < s.length(); i++) {
            if (!set.add(s.charAt(i)))
                set.remove(s.charAt(i));
        }
        return set.size() <= 1;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. We traverse over the string s of length n once only.
- Space complexity : $O(1)$. The set can grow up to a maximum number of all distinct elements. However, the number of distinct characters are bounded, so as the space complexity.

Rate this article: ★★★★★

Previous Next

Comments: 21 Sort By

Type comment here... (Markdown is supported)

Preview Post

maggie000 ★ 32 February 2, 2018 8:48 PM

The space complexity of Approach #5 should not be O(n). It should be O(128). If you make assumption that "String only contains ASCII characters from 0 to 127", this should be true in this approach too.According to the pigeonhole principle, the set size could only be 128 or less.

31 Share Reply

SHOW 2 REPLIES

yuhui4 ★ 41 October 4, 2018 9:01 PM

Any methods that use map or set should have space complexity O(1) as the char number should be less than 256 as the assumption in the O(128) or O(256)

16 Share Reply

androso ★ 6 August 12, 2018 12:22 AM

Use a bitset, then flip the bit for each character.
If a bit is still set, then the character occurred an odd number of times.

```
public static boolean hasPalindrome(String s) {
```

Read More

6 Share Reply

SHOW 1 REPLY

leetcodefan ★ 1942 January 3, 2019 3:39 AM

Comprehensive and inspiring. Thank you.

2 Share Reply

kevin109104 ★ 9 October 24, 2018 9:46 AM

32 ms Python3. We know palindromes have 0 or 1 unpaired characters. Similar to approach 5 above

```
class Solution:
    def canPermutePalindrome(self, s):
```

Read More

2 Share Reply

premagopu ★ 1 September 20, 2018 8:38 PM

I tried solution 5 with input "tact coa". The hashset includes space as a value so the count is not less than equal to 1.

1 Share Reply

mrkingdom75 ★ 1 December 13, 2017 4:41 PM

@Nu1L "aab" is not Palindrome but it is Palindrome Permutation

1 Share Reply

SHOW 3 REPLIES

Nu1L ★ 34 October 29, 2017 8:17 AM

"aab" was Palindrome Permutation? The description maybe wrong, am I right?

1 Share Reply

SHOW 2 REPLIES

Jenniferfight ★ 21 June 18, 2019 6:01 AM

I am not sure the solution3, map[s.charAt(i)] means map[letter ASCII]? Why it can just use key from 0 to length to search?

0 Share Reply

vjsfbay ★ 54 March 28, 2018 3:00 AM

Such an elegant solution (#5)

0 Share Reply

‹ 1 2 3 ›