Solution: Palindrome Permutation

Let's solve the Palindrome Permutation problem using the Knowing What to Track pattern.

We'll cover the following

Statement

For a given string, find whether or not a permutation of this string is a palindrome. You should return TRUE if such a permutation is possible and FALSE if it isn't possible.

Constraints:

- $1 \leq \text{string length} \leq 1000$
- The string will contain lowercase English letters.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive solution is to first compute all possible permutations of a given string and then iterate over each permutation to see if it's a palindrome. We can use two pointers to traverse the computed permutation from the beginning and the end simultaneously, comparing each character at every step. If the two pointers point to the same character until they cross each other, the string is a palindrome.

The time complexity of this approach is $O(n!)$ + $O(n)$, where n is the number of characters in the input string. The space complexity is $O(n!)$.

Optimized approach using frequency counting

This is the stereotypical example of a problem whose naive solution is very inefficient, yet by identifying a few key properties of the problem, we are able to devise a solution with superior time and space complexity measures. In this case, there are three key observations:

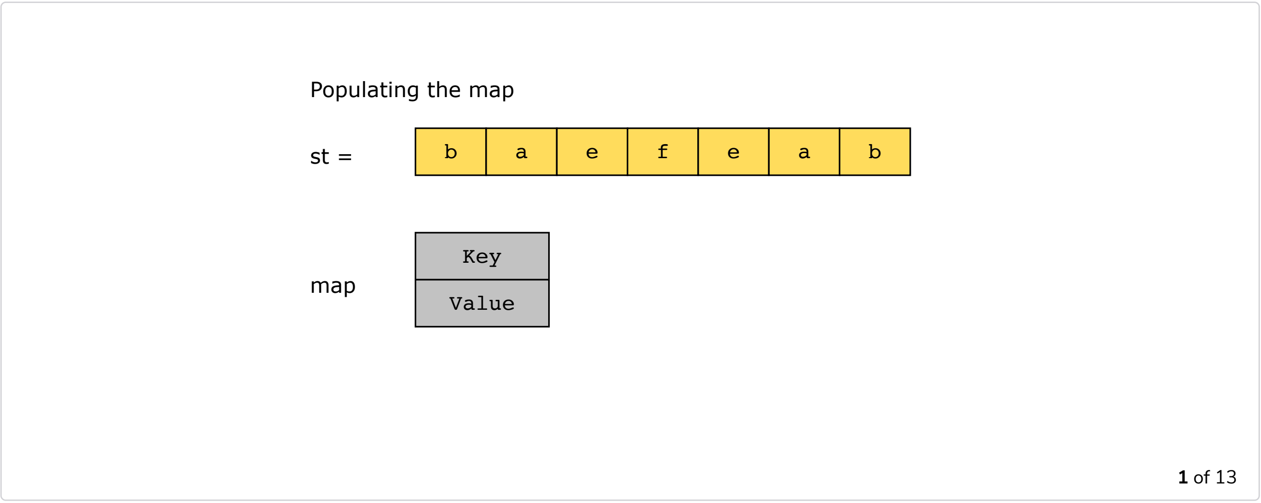
- In a palindrome of even length, all the characters occur an even number of times. For example, in the palindrome "aaaabbaaaa", "a" occurs four times and "b" occurs twice.

- In a palindrome of odd length, the character in the middle of the string appears once, and all the others occur an even number of times. For example, in the palindrome “aaabaaa”, “a” occurs six times and “b” occurs once.
- The observations above are true for all the permutations of a palindrome. For example, “aaaaaaaabb” is a permutation of “aaaabbaaaa” in which “a” still occurs four times and “b” occurs twice.

So, to decide whether or not a given string has a permutation that is a palindrome, all we need to do is count the number of times each character appears in it and then check how many characters appear an odd number of times.

- If no characters appear an odd number of times, then the string is of even length and has a permutation that is a palindrome.
- If only one character appears an odd number of times, then the string is of odd length and has a permutation that is a palindrome.
- If more than one character appears an odd number of times, then the string does not have a permutation that is a palindrome.

The slides below illustrate the working of our proposed solution:



> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

Building on the above idea, the first step in our algorithm is to traverse the input string and populate a hash map with characters and their frequencies. If a character is already present, we’ll increment its frequency by 1. Otherwise, we set its frequency $= 1$.

Java

```java
class PalindromePermutation {

  public static String printArrayWithMarkers(String arr, int pvalue) {
    String out = "";
    for (int i = 0; i < pvalue; i++) {
      out += arr.charAt(i);
    }
    out += "«" + arr.charAt(pvalue) + "»";
    for (int i = pvalue + 1; i < arr.length(); i++) {
      out += arr.charAt(i);
    }
    }
    return out;
  }

  public static boolean permutePalindrome(String st) {
    // Create a hashmap to keep track of the characters and their occurrences
    System.out.println("\n\tPopulating the hashmap");
    HashMap < Character,Integer > frequencies = new HashMap < Character,Integer > ();
    int index = 0;
    for (int i = 0; i < st.length(); i++) {
      System.out.println("\t\t" + printArrayWithMarkers(st, index));
      index += 1;
      System.out.println("\t\tCharacter: " + st.charAt(i));
      System.out.println("\t\tHashmap: " + frequencies);
      if (frequencies.containsKey(st.charAt(i))) {
        System.out.println("\t\t\t" + st.charAt(i) + " is already present in the hashmap");
        System.out.print("\t\t\tUpdating its frequency  ⟶ ");
        frequencies.put(st.charAt(i), frequencies.get(st.charAt(i)) + 1);
```

Populating the hash map

Next, we traverse the hash map and count the characters with an odd number of occurrences. We keep track of this number using a variable `count` that is incremented every time a character with an odd frequency is encountered.

Java

```java
class PalindromePermutation {

  public static String printArrayWithMarkers(String arr, int pvalue) {
    String out = "";
    for (int i = 0; i < pvalue; i++) {
      out += arr.charAt(i);
    }
    out += "«" + arr.charAt(pvalue) + "»";
    for (int i = pvalue + 1; i < arr.length(); i++) {
      out += arr.charAt(i);
    }
    return out;
  }

  public static boolean permutePalindrome(String st) {
    // Create a hashmap to keep track of the characters and their occurrences
    System.out.println("\n\tPopulating the hashmap");
    HashMap < Character,Integer > frequencies = new HashMap < Character,Integer > ();
    int index = 0;
    for (int i = 0; i < st.length(); i++) {
      System.out.println("\t\t" + printArrayWithMarkers(st, index));
      index += 1;
      System.out.println("\t\tCharacter: " + st.charAt(i));
      System.out.println("\t\tHashmap: " + frequencies);
      if (frequencies.containsKey(st.charAt(i))) {
        System.out.println("\t\t\t" + st.charAt(i) + " is already present in the hashmap");
        System.out.print("\t\t\tUpdating its frequency  ⟶ ");
        frequencies.put(st.charAt(i), frequencies.get(st.charAt(i)) + 1);
```

Counting the numbers

Lastly, we check if `count` is greater than 1. If yes, no permutation is a palindrome. Otherwise, at least one of the permutations of the given string is a palindrome.

Java

```java
class PalindromePermutation {

  public static String printArrayWithMarkers(String arr, int pvalue) {
    String out = "";
    for (int i = 0; i < pvalue; i++) {
      out += arr.charAt(i);
    }
    out += "«" + arr.charAt(pvalue) + "»";
    for (int i = pvalue + 1; i < arr.length(); i++) {
      out += arr.charAt(i);
    }
    return out;
  }
  public static boolean permutePalindrome(String st) {
    // Create a hashmap to keep track of the characters and their occurrences
    System.out.println("\n\tPopulating the hashmap");
    HashMap < Character,Integer > frequencies = new HashMap < Character,Integer > ();
    int index = 0;
    for (int i = 0; i < st.length(); i++) {
      System.out.println("\t\t" + printArrayWithMarkers(st, index));
      index += 1;
      System.out.println("\t\tCharacter: " + st.charAt(i));
      System.out.println("\t\tHashmap: " + frequencies);
      if (frequencies.containsKey(st.charAt(i))) {
        System.out.println("\t\t\t" + st.charAt(i) + " is already present in the hashmap");
        System.out.print("\t\t\tUpdating its frequency  ⟶ ");
        frequencies.put(st.charAt(i), frequencies.get(st.charAt(i)) + 1);
        System.out.println(frequencies + "\n");
```

Palindrome Permutation

Just the code

Here’s the complete solution to this problem:

Java

```java
class PalindromePermutation {

  public static boolean permutePalindrome(String st) {

    HashMap < Character,Integer > frequencies = new HashMap < Character,Integer > ();
    int index = 0;
    for (int i = 0; i < st.length(); i++) {
      index += 1;
      if (frequencies.containsKey(st.charAt(i))) {
        frequencies.put(st.charAt(i), frequencies.get(st.charAt(i)) + 1);
      }
      else {
        frequencies.put(st.charAt(i), 1);
      }
    }

    int count = 0;

    for (Character ch: frequencies.keySet()) {
      if (frequencies.get(ch) % 2 != 0) {
        count += 1;
      }
    }

    if (count <= 1) {
      return true;
    }
    else {
```

Palindrome Permutation

Solution summary

To recap, the solution to this problem can be divided into the following parts:

1. Populate a hash map with string characters and their frequencies.
2. Count the characters with frequency ≥ 1.
3. If `count` > 1, return FALSE. Otherwise, return TRUE.

There are two parts in this solution. In the first part, we iterate over a string of n characters, checking their presence in a hash map and adding/updating the hash map as we go. The average time complexity for looking up values in a hash map is $O(1)$. The worst case time complexity for looking up values in a hash map is $O(k)$, where k is the number of elements in the hash map. Since the string only contains lowercase English letters, there are 26 distinct characters in this case.

Thus, the average time complexity of the first part of the solution is $O(n)$ while the worst case time complexity is $O(n * k)$. In this scenario, the worst case time complexity is $O(26 * n)$, which simplifies to $O(n)$.

In the second part, our loop iterates at most 26 times. Hence the time complexity of the second part is $O(1)$.

In summary, our solution's average and worst case time complexity is $O(n)$.

Space complexity

The hash map can grow up to n if all the characters in the string are distinct. However, the number of distinct characters is bounded and so is the space complexity. So, the space complexity is $O(1)$.

← Back
Palindrome Permutati...

Next →
Design Tic-Tac-Toe

✓ Mark as Completed