Topic Introduction

Today, let’s dive into the fascinating world of **palindromic strings** and how they appear in interview problems. A **palindrome** is a string that reads the same forwards and backwards. Think "racecar" or "level." Palindromes are a classic theme in coding interviews because they require you to manipulate strings, count characters, and sometimes leverage dynamic programming.

But what connects the problems we’ll focus on today isn’t just palindromes, but *how* we check for them, build them, or find them in tricky ways:

- Sometimes you want to know: *Can this string become a palindrome if you rearrange it?*
- Or: *How many palindromic permutations can you actually generate?*
- Or: *What’s the longest palindromic subsequence hiding in a string?*

Why are These Problems Grouped Together?

All three problems below explore palindromic strings but from different angles: checking the *possibility* of a palindrome, *generating* all palindromic permutations, and *finding* the longest palindromic subsequence. The first two are closely related (both about rearrangements), while the last one adds the twist of subsequences and dynamic programming.

When do you see palindromes in interviews?

- When an interviewer wants to test your ability to count, use hash maps/sets, or identify subtle constraints.
- When they want to see if you can recognize DP substructure (especially for substrings or subsequences).
- When they want to see you reason about permutations and string construction.

Let’s warm up with a simple example (not one of our chosen problems):

Suppose you’re asked: *Is "aabb" a palindrome?*
Easy! "aabb" is not, but "abba" is.
If you can rearrange "aabb" to "abba", then it can become a palindrome!

Notice how we’re already thinking about character counts and symmetry.

Problem 1: Palindrome Permutation

LeetCode Link

Problem Statement (In My Words)

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

Example Input/Output

Input: s = "code"
Output: False
Why? No arrangement of "code" is a palindrome.

Input: s = "aab"

Output: True

Why? "aba" is a palindrome, and it's a permutation of "aab".

Thought Process

- First, recall: A palindrome has mirrored halves. For even-length palindromes, every character appears an even number of times. For odd-length palindromes, only one character can appear an odd number of times (it sits in the middle), all others must appear an even number of times.
- So, this problem reduces to:

Count the number of characters with odd counts.

If more than one, it’s impossible to permute into a palindrome.

Pen-and-paper suggestion:

Write out a few strings and count character frequencies. Try “civic”, “aabb”, “aabbc”, “abc”.

Additional Test Case

Try: s = "carerac"

Expected Output: True

Try to work this out before looking at further hints!

Brute-force Approach

Generate all permutations of the string and check if any is a palindrome.

- **Time Complexity:** O(n!) to generate permutations; O(n) per check. Not feasible for n > 8.

Optimal Approach

Core Pattern:

Use a hash map (dictionary) to count character frequencies. Count how many characters appear an odd number of times.

Step-by-step Logic:

- Count how many times each character appears.
- For each character, check if its count is odd.
- If more than one character has an odd count, return False.
- Otherwise, return True.

Python Solution

```python
def canPermutePalindrome(s):
    # Count character occurrences
    char_count = {}
    for c in s:
        if c in char_count:
            char_count[c] += 1
        else:
            char_count[c] = 1

    # Count how many characters have an odd count
    odd_count = 0
    for count in char_count.values():
        if count % 2 == 1:
            odd_count += 1

    # At most one character can have an odd count
    return odd_count <= 1

# Example usage
print(canPermutePalindrome("aab"))    # True
print(canPermutePalindrome("code"))   # False
```

Time Complexity: O(n), where n is the length of the string.

Space Complexity: O(k), where k is the number of distinct characters.

Code Walkthrough

- We use a dictionary `char_count` to store how many times each character appears.
- We loop through the string and update counts.
- We then count how many characters have *odd* counts.
- If 0 or 1 characters have odd counts, a palindrome permutation is possible.

Example Trace

Input: `"aab"`

- Counts: {'a': 2, 'b': 1}
- Odd counts: 'b' (1 odd)
- `odd_count` = 1 → return True.

Try This Test Case!

s = "aabbc"

What will the code return?

(*Hint: Count the characters and see how many odds you get.*)

Take a moment to solve this on your own before jumping into the solution.

Problem 2: Palindrome Permutation II

LeetCode Link

Problem Statement (In My Words)

Given a string, return *all* unique palindromic permutations of the string.

How is this Different?

- The previous problem just checked *possibility*.
- Now, you need to *generate* all possible palindromic permutations.

Example Input/Output

Input: s = "aabb"
Output: ["abba", "baab"]

Input: s = "abc"
Output: []
Why? No palindromic permutations possible.

Brute-force Approach

Generate all permutations, check each for palindrome, add to results if not already included.

- **Time Complexity:** O(n! x n)
- Too slow for a long string.

Optimal Approach

Core Pattern:
Use character counts to check if a palindromic permutation is possible (like in Problem 1).
If possible, build half the palindrome and permute it, then mirror it to get the full palindrome.

Step-by-step Logic:

- Count character frequencies.
- Check if a palindrome permutation is possible (at most one odd-count char).
- Build "half" of the palindrome using half the count of each character.
- Generate all unique permutations for this half.
- For each half-permutation, mirror it and insert the odd-count character (if any) in the middle.

Pseudocode

```
function generatePalindromes(s):
    count = hashmap of char frequencies in s
    odd_count = number of chars with odd frequencies
    if odd_count > 1:
        return []

    half = []  // characters for half permutation
    mid = ""   // odd character if any
    for each char, freq in count:
        if freq % 2 == 1:
            mid = char
        add char (freq // 2) times to half

    results = []
    permute unique arrangements of half:
        palindrome = perm + mid + reverse(perm)
        add palindrome to results
    return results
```

Example

Input: "aabb"

- Counts: {'a':2, 'b':2}
- Half: ['a', 'b']
- Permutations of "ab": "ab", "ba"
- Palindromes: "abba", "baab"

Another Test Case to Try

s = "aabbc"

Expected Output: ["abcba", "bacab", "acbca", "cabac", "bcaab", "cbaac"]

(Try to list them yourself!)

Code Walkthrough of the Process

- **Count characters:**
  - If more than one odd count, return []
- **Build half:**
  - Add each character freq//2 times
- **Unique permutations:**
  - Use backtracking or itertools.permutations with a set for uniqueness.
- **Build palindromes:**
  - For each permutation, assemble: perm + (odd char if any) + perm reversed

Example Trace

Input: "aabb"

- Half: ['a', 'b']
- Unique perms: ["ab", "ba"]
- Output: ["abba", "baab"]

Time & Space Complexity

- **Time:** O((n/2)!) for half-length permutation generation, times O(n) to assemble each palindrome.
- **Space:** O((n/2)!) for storing permutations and results.

Problem 3: Longest Palindromic Subsequence

LeetCode Link

What's the Challenge?

This problem is different:
Instead of rearranging, you want the *longest subsequence* (not substring!) of the input that is a palindrome.
A subsequence is any sequence derived by deleting some (or no) letters, without reordering the rest.

Example Input/Output

Input: s = "bbbab"
Output: 4
Why? "bbbb" is a palindromic subsequence.

Input: s = "cbbd"
Output: 2
Why? "bb" is the longest palindromic subsequence.

Brute-force Approach

Try all subsequences and check if they're palindromic.

- **Time Complexity:** O(2^n x n)
- Not practical.

Optimal Approach

Core Pattern:
Dynamic Programming (DP) with memoization.

Step-by-step Logic:

- Let dp[i][j] be the length of the longest palindromic subsequence in s[i:j+1].
- If s[i] == s[j], then dp[i][j] = 2 + dp[i+1][j-1]
- Else, dp[i][j] = max(dp[i+1][j], dp[i][j-1])
- Base case: dp[i][i] = 1 (a single char is a palindrome)

Pseudocode

```
function longestPalindromeSubseq(s):
    n = length of s
    dp = 2D array of size n x n, all 0

    for i from n-1 downto 0:
        dp[i][i] = 1
        for j from i+1 to n-1:
            if s[i] == s[j]:
                dp[i][j] = 2 + dp[i+1][j-1]
            else:
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])

    return dp[0][n-1]
```

Example Trace

Input: "bbbab"

- dp[0][4] will eventually store the answer.
- As you fill out the table, you’ll see that "bbbb" (positions 0,1,2,3) is the answer.

Another Test Case

s = "agbdba"
Expected Output: 5
("abdba" or "abdba" is the longest palindromic subsequence)

Try implementing this logic and running through the test case!

Time & Space Complexity

- **Time:** O(n^2) for DP table of size n x n.
- **Space:** O(n^2) for the table (can optimize to O(n) if only previous row/column is needed).

Summary and Next Steps

Why were these problems grouped?

They all revolve around palindromic strings but test different skills: frequency counting, permutation building, and dynamic programming. Mastering these helps you handle a wide variety of string and DP problems in interviews.

Key patterns and insights:

- Use character counts to check for palindromic permutations.
- Build palindromic permutations by permuting half and mirroring.
- Dynamic programming is your friend for subsequence problems.

Common pitfalls:

- Forgetting to handle odd-length palindromes correctly.
- Not accounting for duplicate permutations in Palindrome Permutation II.
- Confusing substring with subsequence in the DP problem.

Action List

- Solve all 3 problems on your own — even the one with code provided!
- Try solving Problem 2 and 3 using a different technique (e.g., recursive backtracking for subsequence).
- Explore other problems using palindromic logic, like Longest Palindromic Substring.
- Compare your solution with others — look for efficiency and clarity.
- If you get stuck, don’t worry! Review, experiment, and keep practicing.

Stay curious and keep coding — palindromes await!