Topic Introduction

Today, we're diving deep into the world of **palindromes** — strings that read the same forwards and backwards. You've probably heard the word "palindrome" before (think "racecar" or "level"). But, did you know how often palindrome patterns pop up in coding interviews? If not, you're about to find out!

What is a Palindrome?

A **palindrome** is a sequence (usually a string) that is identical when read both forwards and backwards. For example:

- "madam"
- "noon"
- "A man, a plan, a canal: Panama" (if you ignore spaces and punctuation)

How Does Palindrome Checking Work?

The typical way to check for a palindrome is to compare the first character with the last, the second with the second-to-last, and so on, moving towards the center. If all pairs match, it's a palindrome!

Why Is This Pattern Useful in Interviews?

- **String manipulation** is a common interview topic.
- Palindrome checks test your understanding of **two-pointer techniques**.
- These problems can be extended to trickier scenarios, like allowing deletions or searching for palindromic substrings.

Quick Example (not one of today's problems):

Check if "deified" is a palindrome:

- Compare first and last: 'd' == 'd'
- Next: 'e' == 'e'
- Next: 'i' == 'i'
- Reached the middle. All matched!
- So, it's a palindrome.

Today's set of problems all revolve around palindromes but each with their own twist:

- **Valid Palindrome** (LeetCode 125): Check if a string is a palindrome, ignoring non-alphanumeric characters.
- **Valid Palindrome II** (LeetCode 680): Same as above, but you can remove *at most one* character to make it a palindrome.
- **Longest Palindromic Substring** (LeetCode 5): Find the longest palindromic substring in a given string.

Why group these together?

All three are about **palindrome validation and searching**. The first checks for palindromes with cleaning, the second introduces a single allowed deletion, and the third finds the longest palindromic substring. Together, they give you a full workout on palindrome techniques!

Let's start with the basics and build up!

Problem 1: Valid Palindrome

LeetCode 125. Valid Palindrome

Problem Statement (Rephrased)

Given a string, determine if it is a palindrome, **ignoring non-alphanumeric characters** and **case differences**.

Example

Input: `"A man, a plan, a canal: Panama"`
Output: `True`
Explanation: After removing non-alphanumerics and lowercasing, the string becomes "amanaplanacanalpanama", which is a palindrome.

Another Test Case

Input: `"race a car"`
Output: `False`

Thought Process

- We need to compare the string from both ends, skipping over any characters that aren't letters or numbers.
- Also, comparisons should be case-insensitive.

Try this by hand:

Input: `"No lemon, no melon"`
- Remove non-alphanumerics: "nolemonnomelon"
- Lowercase: "nolemonnomelon"
- Compare from both ends? Yes, all match. So, it's a palindrome.

Brute-Force Approach

- Clean the string (remove non-alphanumeric and lowercase).

- Reverse the cleaned string and compare to the original cleaned string.
- Time Complexity: O(n) for cleaning, O(n) for reversing, O(n) for comparison -> O(n)
- Space Complexity: O(n) for the cleaned string.

Optimal Approach

Let's use **two pointers**: one at the start, one at the end.
Move both pointers towards the center, skipping non-alphanumeric characters, and comparing the lowercase characters.

Step-by-Step Logic

- Set `left` at 0 and `right` at len(s) - 1.
- While `left < right`:
    - Move `left` forward if not alphanumeric.
    - Move `right` backward if not alphanumeric.
    - Compare `s[left]` and `s[right]` (lowercased).
    - If not equal, return False.
    - Move both pointers inward.
- If we finish the loop, return True.

Python Solution

```python
def isPalindrome(s):
    """
    :type s: str
    :rtype: bool
    """
    left, right = 0, len(s) - 1

    while left < right:
        # Skip non-alphanumeric characters for left pointer
        while left < right and not s[left].isalnum():
            left += 1
        # Skip non-alphanumeric characters for right pointer
        while left < right and not s[right].isalnum():
            right -= 1
        # Compare characters in lowercase
        if s[left].lower() != s[right].lower():
            return False
        left += 1
        right -= 1
    return True
```

Time and Space Complexity

- Time: O(n) — Each character is checked at most once.
- Space: O(1) — No extra space used, just pointers.

Explanation and Trace

- `left` and `right` scan towards the middle.
- `isalnum()` checks if a character is a letter or number (Python built-in).
- `.lower()` converts to lowercase.

Trace Example

Input: `"A man, a plan, a canal: Panama"`

- left='A', right='a' -> both are 'a' (after lowercasing)
- left=' ', skip (not alnum)
- right='m', keep going...
- Continue moving pointers and comparing.
- All pairs match, so return True.

Try it Yourself

Input: `".,!"`
What does the function return? (Hint: Empty string is a palindrome.)

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

Problem 2: Valid Palindrome II

LeetCode 680. Valid Palindrome II

Problem Statement (Rephrased)

Given a string, determine if it can become a palindrome by **removing at most one character**.

Example

Input: `"abca"`
Output: `True`
Explanation: Remove 'b' or 'c' to get "aca" or "aba", both palindromes.

Another Test Case

Input: `"abc"`
Output: `False`

How is this different from Problem 1?

- Now, you can skip ONE mismatched character to try to force a palindrome.

Brute-Force Idea

- For each character, remove it and check if the new string is a palindrome (using Problem 1's technique).
- Time Complexity: O(n^2) — for each character, O(n) to check.

Optimal Approach

We can still use the **two-pointer technique**, but now, when we find a mismatch, we have two options:

- Skip the left character and check if the rest is palindrome.
- Skip the right character and check if the rest is palindrome.

If either is a palindrome, return True.

Step-by-Step Logic

- Start with left and right pointers.
- While left < right:
    - If characters match, move both pointers.
    - If they don't:
        - Try skipping left: check if s[left+1:right+1] is palindrome.
        - Try skipping right: check if s[left:right] is palindrome.
        - If either is True, return True.
        - Else, return False.
- If loop completes, return True.

Pseudocode

```
function validPalindrome(s):
    left = 0, right = len(s) - 1
    while left < right:
        if s[left] == s[right]:
            left += 1
            right -= 1
        else:
            # Try removing left or right character
            return (isPalindromeRange(s, left+1, right) or
                    isPalindromeRange(s, left, right-1))
    return True

function isPalindromeRange(s, i, j):
    while i < j:
        if s[i] != s[j]:
            return False
        i += 1
        j -= 1
    return True
```

Example Input/Output

Input: `"deeee"`

Output: `True`

Explanation: Removing 'd' gives "eeee", which is a palindrome.

Another Test Case

Input: `"abcda"`

Output: `False`

Try removing any single character — you can't get a palindrome.

Trace With Example

Input: `"abca"`

- a == a: move in
- b != c:
    - Remove b: check `"aca"` -> palindrome
    - Remove c: check `"aba"` -> palindrome
    - Either way, it works. Return True.

Time and Space Complexity

- Time: O(n) — Only one mismatch, so at most two O(n) palindrome checks.
- Space: O(1)

Problem 3: Longest Palindromic Substring

LeetCode 5. Longest Palindromic Substring

Problem Statement (Rephrased)

Given a string, find the **longest contiguous substring** which is a palindrome.

Example

Input: `"babad"`
Output: `"bab"` (or `"aba"` — either is valid)

Another Test Case

Input: `"cbbd"`
Output: `"bb"`

What's Different Here?

- Now, instead of checking the whole string, we're searching for the **longest substring** that is a palindrome.
- This requires checking all possible substrings or using a smarter approach.

Brute-Force Idea

- Check all substrings and see if they are palindromes. Keep track of the longest one.
- Time Complexity: O(n^3) (O(n^2) substrings, O(n) check for each)
- Not efficient!

Optimal Approach: Expand Around Center

- A palindrome mirrors around its center.
- There are 2n-1 centers (each character, plus gaps between characters).
- For each center, expand outwards as long as characters match.

Step-by-Step Logic

- For every index in the string:
    - Expand around that character (odd length palindrome).
    - Expand around the gap between that character and the next (even length).

- For each expansion, keep track of the start and end index of the longest palindrome found.
- At the end, return the substring using those indices.

Pseudocode

```
function longestPalindrome(s):
    start, end = 0, 0
    for i in 0 to len(s)-1:
        len1 = expandAroundCenter(s, i, i)      # Odd length
        len2 = expandAroundCenter(s, i, i+1)    # Even length
        maxlen = max(len1, len2)
        if maxlen > end - start:
            start = i - (maxlen - 1) // 2
            end = i + maxlen // 2
    return s[start:end+1]

function expandAroundCenter(s, left, right):
    while left >= 0 and right < len(s) and s[left] == s[right]:
        left -= 1
        right += 1
    return right - left - 1  # Length of palindrome
```

Example Input/Output

Input: `"babad"`
Output: `"bab"` or `"aba"`

Another Test Case

Input: `"forgeeksskeegfor"`
Output: `"geeksskeeg"`

Trace With Example

Input: `"cbbd"`

- Center at `b`, expand: "bb" is the longest palindrome.

Time and Space Complexity

- Time: O(n^2) — For each center, expand up to O(n)
- Space: O(1)

Summary and Next Steps

Today we tackled three classic palindrome-related problems:

- **Valid Palindrome:** Two-pointer scan, skip non-alphanumerics, and compare.
- **Valid Palindrome II:** Same, but allow for a single deletion at one mismatch.
- **Longest Palindromic Substring:** Expand around every center to find the max palindrome.

Key Patterns to Remember:

- The **two-pointer technique** is your friend for checking palindromes.
- When allowed deletions, try both options (skip left or right).
- For substring searches, expanding around centers is a powerful approach.

Common Mistakes:

- Forgetting to skip non-alphanumeric characters or to lowercase when needed.
- Not handling even and odd length palindromes for substring problems.
- Not considering all possible centers for expansion.

Action List:

- Solve all 3 problems on your own — even the one with code provided.
- For Problem 2, try implementing a recursive version for practice.
- For Problem 3, look up Manacher’s Algorithm for a linear time solution if you’re feeling bold!
- Try writing test cases that include spaces, punctuation, and both even/odd length palindromes.
- Compare your code with others to see different styles and edge case handling.
- Practice explaining your approach out loud — it helps deepen understanding!

And remember: It’s totally fine if you get stuck — keep practicing, and these patterns will soon feel like second nature!