

Overview

Approach 1: Check All Substrings

Intuition

We can start with a brute-force approach. We will simply check if each substring is a palindrome, and take the longest one that is.

First, let's talk about how we can check if a given string is a palindrome. This is a classic problem and we can do it using two pointers. If a string is a palindrome, the first character is equal to the last character. The second character is equal to the second last character, and so on.



We initialize two pointers: one at the start of the string and another at the end of it. We check if the characters at the pointers are equal - if they aren't, we know the string cannot be a palindrome. If they are equal, we move to the next pair of characters by moving the pointers toward each other. We continue until we either find a mismatch or the pointers meet. If the pointers meet, then we have checked all pairs and we know the string is a palindrome.

One bonus to using this algorithm is that we frequently exit early on strings that are not palindromes. If you had a string of length `1000` and the third and third last characters did not match, we would exit the algorithm after only 3 iterations.

There's another optimization that we can do. Because the problem wants the longest palindrome, we can start by checking the longest-length substrings and iterate toward the shorter-length substrings. This way, the first time we find a substring that is a palindrome, we can immediately return it as the answer.

Algorithm

1. Create a helper method `check(i, j)` to determine if a substring is a palindrome.
 - To save space, we will not pass the substring itself. Instead, we will pass two indices that represent the substring in question. The first character will be `s[i]` and the last character will be `s[j - 1]`.
 - In this function, declare two pointers `left = i` and `right = j - 1`.
 - While `left < right`, do the following steps:
 - If `s[left] != s[right]`, return `false`.
 - Otherwise, increment `left` and decrement `right`.
 - If we get through the while loop, return `true`.
2. Use a for loop to iterate a variable `length` starting from `s.length` until `1`. This variable represents the length of the substrings we are currently considering.
3. Use a for loop to iterate a variable `start` starting from `0` until and including `s.length - length`. This variable represents the starting point of the substring we are currently considering.
4. In each inner loop iteration, we are considering the substring starting at `start` until `start + length`. Pass these values into `check` to see if this substring is a palindrome. If it is, return the substring.

Implementation

 Copy

Java

Python3

```
1 class Solution {
2     public String longestPalindrome(String s) {
3         for (int length = s.length(); length > 0; length--) {
4             for (int start = 0; start <= s.length() - length; start++) {
5                 if (check(start, start + length, s)) {
6                     return s.substring(start, start + length);
7                 }
8             }
9         }
10
11         return "";
12     }
13
14     private boolean check(int i, int j, String s) {
15         int left = i;
16         int right = j - 1;
17
18         while (left < right) {
19             if (s.charAt(left) != s.charAt(right)) {
20                 return false;
21             }
22
23             left++;
24             right--;
25         }
26
27         return true;
```

Complexity Analysis

Given n as the length of `s`,

- Time complexity: $O(n^3)$

The two nested for loops iterate $O(n^2)$ times. We check one substring of length `n`, two substrings of length `n - 1`, three substrings of length `n - 2`, and so on.

There are n substrings of length 1, but we don't check them all since any substring of length 1 is a palindrome, and we will return immediately.

Therefore, the number of substrings that we check in the worst case is $1 + 2 + 3 + \dots + n - 1$. This is the partial sum of [this series](#) for $n - 1$, which is equal to $\frac{n \cdot (n-1)}{2} = O(n^2)$.

In each iteration of the while loop, we perform a palindrome check. The cost of this check is linear with n as well, giving us a time complexity of $O(n^3)$.

Note that this time complexity is in the worst case and has a significant constant divisor that is dropped by big O. Due to the optimizations of checking the longer length substrings first and exiting the palindrome check early if we determine that a substring cannot be a palindrome, the practical runtime of this algorithm is not too bad.

- Space complexity: $O(1)$

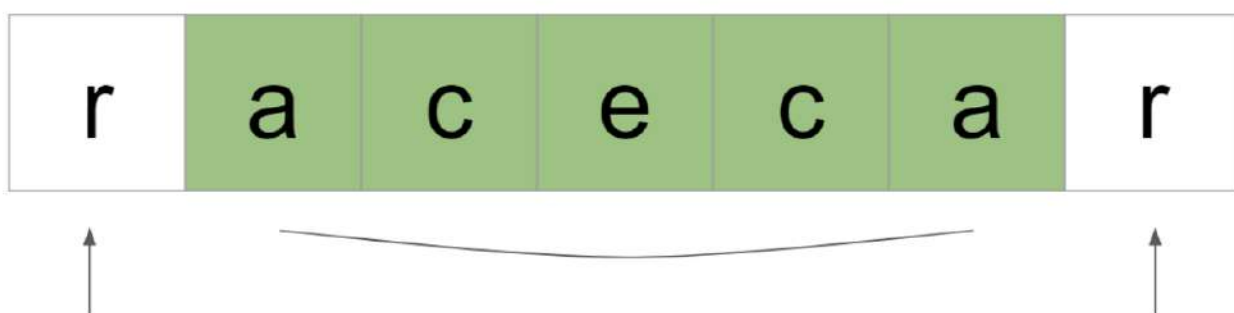
We don't count the answer as part of the space complexity. Thus, all we use are a few integer variables.

Approach 2: Dynamic Programming

Intuition

Let's say that we knew the substring with inclusive bounds i, j was a palindrome. If $s[i - 1] == s[j + 1]$, then we know the substring with inclusive bounds $i - 1, j + 1$ must also be a palindrome, and this check can be done in constant time.

We can flip the direction of this logic as well - if $s[i] == s[j]$ and the substring $i + 1, j - 1$ is a palindrome, then the substring i, j must also be a palindrome.



Characters at pointers are both "r"
Substring one position closer is also a palindrome

We know that all substrings of length 1 are palindromes. From this, we can check if each substring of length 3 is a palindrome using the above fact. We just need to check every i, j pair where $j - i = 2$. Once we know all palindromes of length 3, we can use that information to find all palindromes of length 5, and then 7, and so on.

What about even-length palindromes? A substring of length 2 is a palindrome if both characters are equal. That is, $i, i + 1$ is a palindrome if $s[i] == s[i + 1]$. From this, we can use the earlier logic to find all palindromes of length 4, then 6, and so on.

Let's use a table `dp` with dimensions of $n * n$. `dp[i][j]` is a boolean representing if the substring with inclusive bounds i, j is a palindrome. We initialize `dp[i][i] = true` for the substrings of length 1, and then `dp[i][i + 1] = (s[i] == s[i + 1])` for the substrings of length 2.

Now, we need to populate the table. We iterate over all i, j pairs, starting with pairs that have a difference of 2 (representing substrings of length 3), then pairs with a difference of 3, then 4, and so on. For each i, j pair, we check the condition from earlier:

```
s[i] == s[j] && dp[i + 1][j - 1]
```

If this condition is true, then the substring with inclusive bounds i, j must be a palindrome. We set `dp[i][j] = true`.


Because we are starting with the shortest substrings and iterating toward the longest substrings, every time we find a new palindrome, it must be the longest one we have seen so far. We can use this fact to keep track of the answer on the fly.

Algorithm

1. Initialize `n = s.length` and a boolean table `dp` with size `n * n`, and all values to `false`.
2. Initialize `ans = [0, 0]`. This will hold the inclusive bounds of the answer.
3. Set all `dp[i][i] = true`.
4. Iterate over all pairs $i, i + 1$. For each one, if `s[i] == s[i + 1]`, then set `dp[i][i + 1] = true` and update `ans = [i, i + 1]`.
5. Now, we populate the `dp` table. Iterate over `diff` from 2 until `n`. This variable represents the difference $j - i$.
6. In a nested for loop, iterate over i from 0 until `n - diff`.
 - Set `j = i + diff`.
 - Check the condition: if `s[i] == s[j] && dp[i + 1][j - 1]`, we found a palindrome.
 - In that case, set `dp[i][j] = true` and `ans = [i, j]`

7. Retrieve the answer bounds from `ans` as `i, j` . Return the substring of `s` starting at index `i` and ending with index `j` .

Implementation

 Copy

Java

Python3

```
1 class Solution {
2     public String longestPalindrome(String s) {
3         int n = s.length();
4         boolean[][] dp = new boolean[n][n];
5         int[] ans = new int[]{0, 0};
6
7         for (int i = 0; i < n; i++) {
8             dp[i][i] = true;
9         }
10
11        for (int i = 0; i < n - 1; i++) {
12            if (s.charAt(i) == s.charAt(i + 1)) {
13                dp[i][i + 1] = true;
14                ans[0] = i;
15                ans[1] = i + 1;
16            }
17        }
18
19        for (int diff = 2; diff < n; diff++) {
20            for (int i = 0; i < n - diff; i++) {
21                int j = i + diff;
22                if (s.charAt(i) == s.charAt(j) && dp[i + 1][j - 1]) {
23                    dp[i][j] = true;
24                    ans[0] = i;
25                    ans[1] = j;
26                }
27            }
28        }
29    }
30 }
```

Complexity Analysis

Given n as the length of `s` ,

- Time complexity: $O(n^2)$

We declare an $n * n$ table `dp` , which takes $O(n^2)$ time. We then populate $O(n^2)$ states `i, j` - each state takes $O(1)$ time to compute.

- Space complexity: $O(n^2)$

The table `dp` takes $O(n^2)$ space.

Approach 3: Expand From Centers

Intuition

In the first approach, the palindrome check cost $O(n)$. In the second approach, the palindrome check cost $O(1)$. This allowed us to improve the time complexity from $O(n^3)$ to $O(n^2)$.

The problem with the second approach is that we **always** iterated over $O(n^2)$ states of `i`, `j`. Can we optimize further to minimize the number of iterations required?

In the first approach, we implemented a palindrome check using two pointers. We started by checking the first and last characters, then the second and second last characters, and so on.

Instead of starting the pointers at the edges and moving inwards, the same logic can be applied when starting the pointers at the center and moving outwards. A palindrome mirrors around its center. Let's say you had `s = "racecar"`. If we start both pointers at the middle (`"e"`) and move them away from each other, we can see that at every iteration, the characters match: `e -> c -> a -> r`.

The previous two approaches focused on the bounds of a substring - `i`, `j`. There are $O(n^2)$ bounds, but only $O(n)$ centers. For each index `i`, we can consider odd-length palindromes by starting the pointers at `i`, `i`. To consider the even length palindromes, we can start the pointers at `i`, `i + 1`. There are n starting points for the odd-length palindromes and $n - 1$ starting points for the even-length palindromes - that's $2n - 1 = O(n)$ starting points in total.

This is very promising - we can lower the minimum iterations required if we focus on the centers instead of on the bounds. Let's use a helper method `expand(i, j)` that starts two pointers `left = i` and `right = j`. In this method, we will consider `i, j` as a center. When `i == j`, we are considering odd-length palindromes. When `i != j`, we are considering even-length palindromes. We will expand from the center as far as we can to find the longest palindrome, and then return the length of this palindrome.

Let's say that we have a center `i, i`. We call `expand` and find a length of `length`. What are the bounds of the palindrome? Because we are centered at `i, i`, it means `length` must be odd. If we perform floor division of `length` by 2, we will get the number of characters `dist` on each side of the palindrome. For example, given `s = "racecar"`, we have `length = 7` and `dist = 7 / 2 = 3`. There are 3 characters on each side - "rac" on the left and "car" on the right. Therefore, we can determine that the bounds of the palindrome are `i - dist, i + dist`.

What about a center at `i, i + 1`? `length` must be even now. If we have a palindrome with length 2, then `length / 2 = 1`, but there are zero characters on each side of the center. We can see that `dist` is too large by 1. Therefore, we will calculate `dist` as `(length / 2) - 1` instead. Now, `dist` correctly represents the number of characters on each side. The bounds of the palindrome are `i - dist, i + 1 + dist`.

Algorithm

1. Create a helper method `expand(i, j)` to find the length of the longest palindrome centered at `i, j`.
 - Set `left = i` and `right = j`.
 - While `left` and `right` are both in bounds and `s[left] == s[right]`, move the pointers away from each other.
 - The formula for the length of a substring starting at `left` and ending at `right` is `right - left + 1`.

- However, when the while loop ends, it implies `s[left] != s[right]` . Therefore, we need to subtract `2` . Return `right - left - 1` .

2. Initialize `ans = [0, 0]` . This will hold the inclusive bounds of the answer.

3. Iterate `i` over all indices of `s` .

- Find the length of the longest odd-length palindrome centered at `i` : `oddLength = expand(i, i)` .
- If `oddLength` is the greatest length we have seen so far, i.e. `oddLength > ans[1] - ans[0] + 1` , update `ans` .
- Find the length of the longest even-length palindrome centered at `i` : `evenLength = expand(i, i + 1)` .
- If `evenLength` is the greatest length we have seen so far, update `ans` .

4. Retrieve the answer bounds from `ans` as `i, j` . Return the substring of `s` starting at index `i` and ending with index `j` .

Complexity Analysis

Given n as the length of `s`,

- Time complexity: $O(n^2)$

There are $2n - 1 = O(n)$ centers. For each center, we call `expand`, which costs up to $O(n)$.

Although the time complexity is the same as in the DP approach, the average/practical runtime of the algorithm is much faster. This is because most centers will not produce long palindromes, so most of the $O(n)$ calls to `expand` will cost far less than n iterations.

The worst case scenario is when every character in the string is the same.

- Space complexity: $O(1)$

We don't use any extra space other than a few integers. This is a big improvement on the DP approach.

Approach 4: Manacher's Algorithm

Believe it or not, this problem can be solved in linear time.

[Manacher's algorithm](#) finds the longest palindromic substring in $O(n)$ time and space.

Note: this algorithm is completely out of scope for coding interviews. Because of this, we will not be talking about the algorithm in detail. This approach has been included for the sake of completeness and for those who are curious about algorithms beyond the scope of interviews.

If you wish to learn more about Manacher's algorithm, please reference the above link.

Approach 1: Check All Substrings

class Solution:

```
def longestPalindrome(self, s: str) -> str:
    def check(i, j):
        left = i
        right = j - 1
        while left < right:
            if s[left] != s[right]:
                return False

            left += 1
            right -= 1
        return True
    for length in range(len(s), 0, -1):
        for start in range(len(s) - length + 1):
            if check(start, start + length):
                return s[start:start + length]
    return ""
```

Approach 2: Dynamic Programming

class Solution:

```
def longestPalindrome(self, s: str) -> str:
    n = len(s)
    dp = [[False] * n for _ in range(n)]
    ans = [0, 0]
    for i in range(n):
        dp[i][i] = True
    for i in range(n - 1):
        if s[i] == s[i + 1]:
            dp[i][i + 1] = True
            ans = [i, i + 1]
    for diff in range(2, n):
        for i in range(n - diff):
            j = i + diff
            if s[i] == s[j] and dp[i + 1][j - 1]:
                dp[i][j] = True
                ans = [i, j]
    i, j = ans
    return s[i:j + 1]
```

Approach 3: Expand From Centers

class Solution:

def longestPalindrome(**self**, s: str) -> str:

def expand(i, j):

 left = i

 right = j

while left >= 0 and right < len(s) and s[left] == s[right]:

 left -= 1

 right += 1

return right - left - 1

 ans = [0, 0]

for i **in** range(len(s)):

 odd_length = expand(i, i)

if odd_length > ans[1] - ans[0] + 1:

 dist = odd_length // 2

 ans = [i - dist, i + dist]

 even_length = expand(i, i + 1)

if even_length > ans[1] - ans[0] + 1:

 dist = (even_length // 2) - 1

 ans = [i - dist, i + 1 + dist]

 i, j = ans

return s[i:j + 1]

Approach 4: Manacher's Algorithm

class Solution:

def longestPalindrome(self, s: str) -> str:

s_prime = '#' + '#'.join(s) + '#'

n = len(s_prime)

palindrome_radii = [0] * n

center = radius = 0

for i in range(n):

mirror = 2 * center - i

if i < radius:

palindrome_radii[i] = min(radius - i, palindrome_radii[mirror])

while (i + 1 + palindrome_radii[i] < n and

i - 1 - palindrome_radii[i] >= 0 and

s_prime[i + 1 + palindrome_radii[i]] == s_prime[i - 1 - palindrome_radii[i]]):

palindrome_radii[i] += 1

if i + palindrome_radii[i] > radius:

center = i

radius = i + palindrome_radii[i]

max_length = max(palindrome_radii)

center_index = palindrome_radii.index(max_length)

start_index = (center_index - max_length) // 2

longest_palindrome = s[start_index: start_index + max_length]

return longest_palindrome