17. Interview Question

By: Steve Krenzel

Previous Index Next

Longest Palindrome

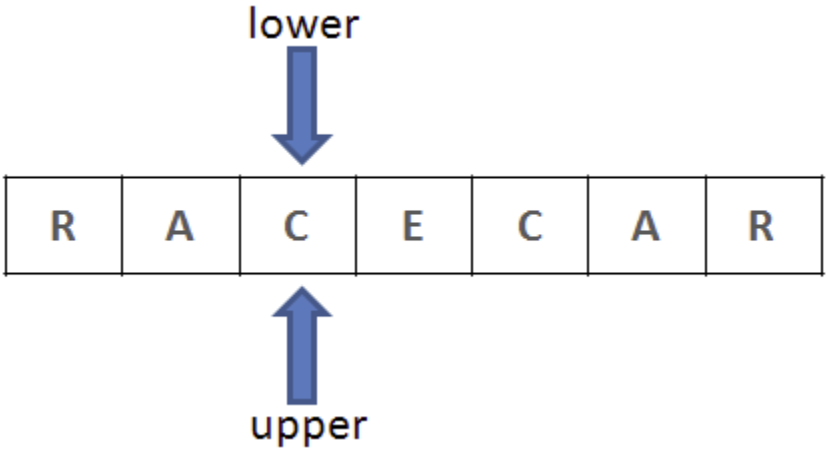
I was asked about finding the longest palindrome in a string today. Here is how I said I'd approach the problem.

Algorithm:

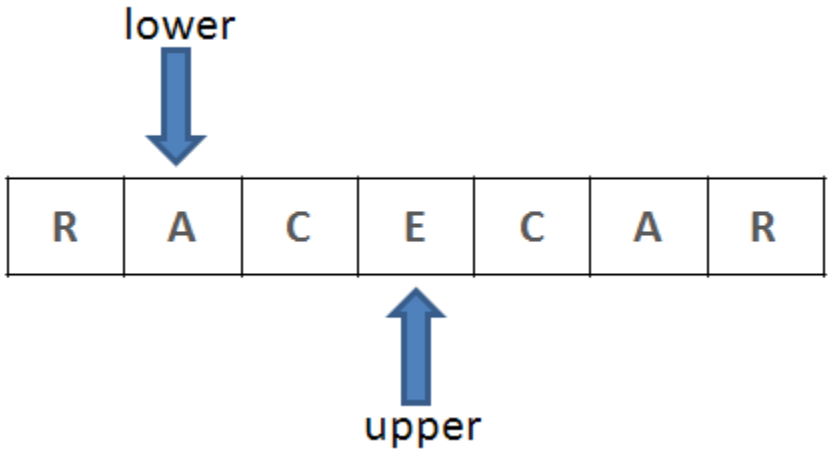
Assume for simplicity that the palindrome is a length that is an odd number. Also, we treat a single character as a palindrome of length 1.

For each position in the string we store an upper and lower pointer that tracks the upper and lower bounds of the largest palindrome centered at that position.

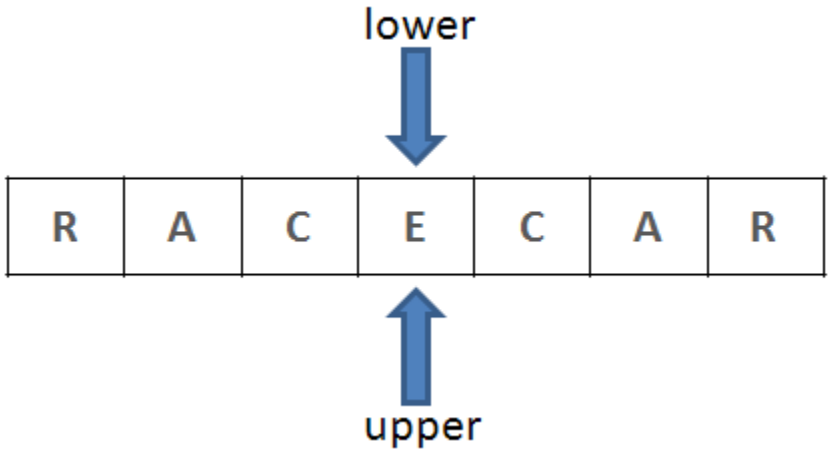
For instance, position 2 in the string "racecar" would start as:



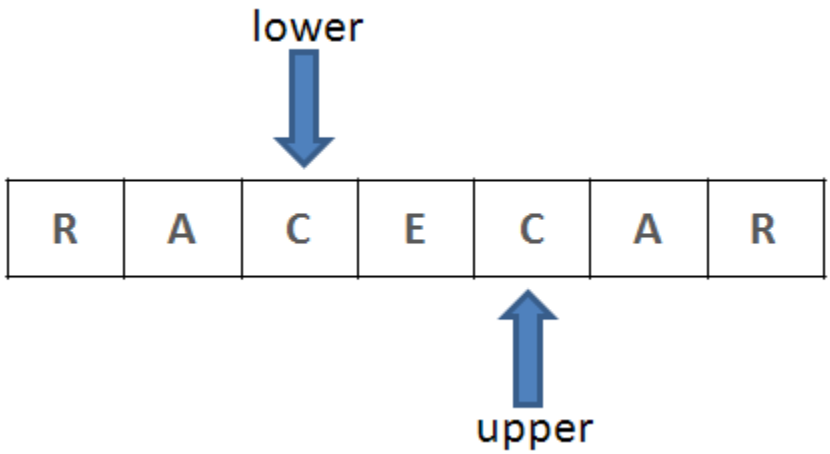
At this point, the longest known palindrome at position 2 is the single character 'c'. We then increment the upper pointer and decrement the lower pointer and compare their respective values.



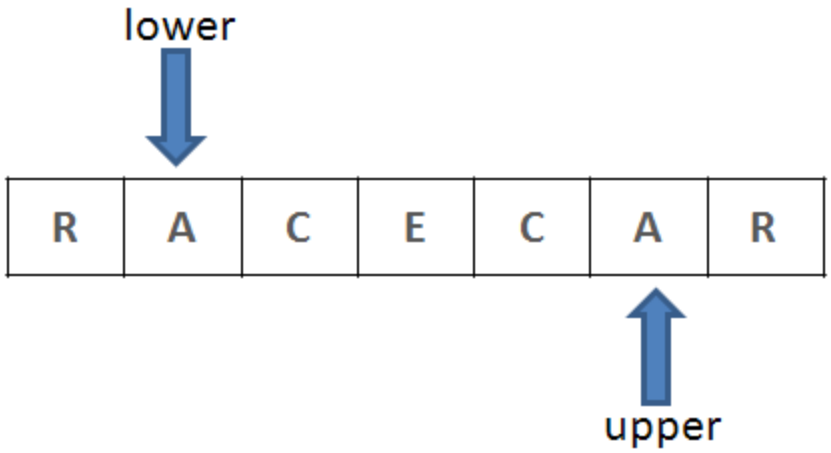
Since the upper and lower values are not the same, we stop and now know that the longest palindrome centered at position 2 is 'c'. We then start the process again at the next string position, 3.



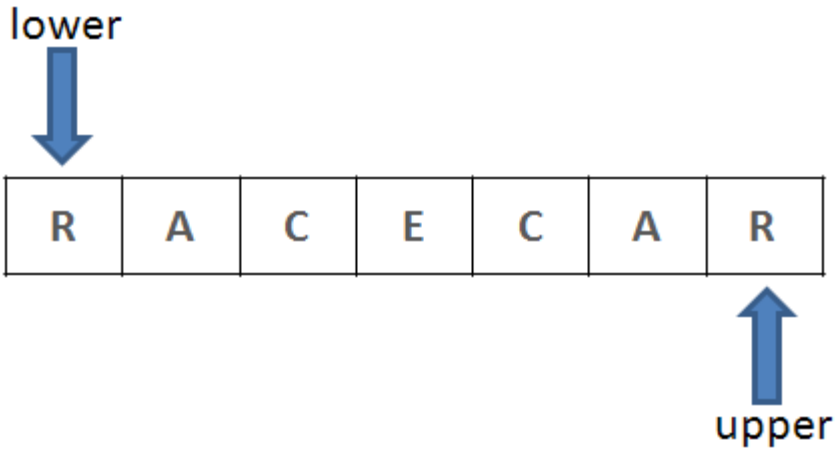
We increment and decrement the upper and lower pointers and see that they are equal. Our current longest palindrome centered at position 3 is now 'cec'.



Since they were equal, we increment and decrement the pointers again and see that they are still equal. Our new longest palindrome centered at position 3 is now 'aceca'.



We continue incrementing and decrementing the upper and lower pointers until they don't match or they hit one of the ends of the string.

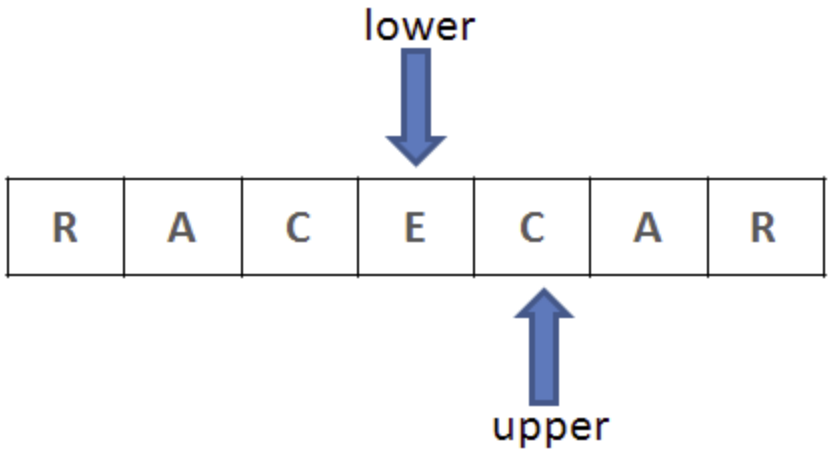


Since our pointers are equal, we increment and decrement them again. We hit an end of the string and the pointers are equal so we know that our longest palindrome centered at position 3 is 'racecar'.

We're done with position 3 and continue this same process again at position 4. We'll do this until we've covered every position in the string.

NOTE: We initally assumed that the palindrome had an odd length. This

algorithm works for even palindromes as well. The only change we make is that when we initialize the pointers, instead of having the lower and upper pointer point to the same position we have the upper pointer point at the next position. Everything else with the algorithm remains the same. i.e. The initial pointers for an even palindrome 'centered' at position 3 would look like:



Runtime:

Worst-Case: O(N^2)
Best-Case: O(N)

The worst case is achieved when the string is a single repeated character because every substring is a palindrome.

The best case is achieved when the string contains no palindromes.

Typically if a string only contains a single palindrome (the fewer the better), the closer to O(N) it will run. This is because every time it checks a position in the string, it checks the character before and after that position, and if they don't match then it stops looking for the palindrome. Positions in the string can be discarded after only one lookup if that position doesn't have a palindrome, so if there are no palindromes you only do N comparisons.

Here an implementation of the algorithm:

```
def findLongestPalindrome(string):
    return max([getPalindromeAt(i, string) for i in xrange(len(string))],
               key = lambda a: len(a)) if len(string) > 0 else ''

def getPalindromeAt(position, string):
    longest = (position, position)
    for lower, upper in [(position - 1, position + 1),
                         (position, position + 1)]:
        while lower >= 0 and
              upper < len(string) and
              string[lower] == string[upper]:
            upper += 1
            lower -= 1
        longest = max(longest, (lower + 1, upper - 1),
                      key = lambda a: a[1] - a[0])
    return string[longest[0] : longest[1] + 1]
```

It's up on github too with doctests and more comments.

About the author
I'm Steve Krenzel, a software engineer and co-founder of Thinkfuse.
Feel free to send me an email at sgk284@gmail.com

Previous Next