Google

Interview Question

Software Engineer Interview

"You're given a string, and you want to split it into as few strings as possible such that each string is a palindrome."

Answer Add Tags

Interview Answer

13 Answers

▲ 0 ▼

Trivial - the number of palindromes equals to the number of characters appearing the odd number of times. You cannot do better than this. Each of those characters should be taken as the "seed" of a palindrome. Other letters should be taken in pairs and used to buld the palindromes by taking any "seed" and appending one letter at the beginning, and the same letter at the end.
The whole thing runs in O(n) time.

vsp on Oct 7, 2010

▲ 0 ▼

Can you clarify? "abababab" -- each letter appears an even number of times (4), hence you claim there are 0 palindromes and cannot do better. One ideal way to break this up would be to split it into "ababa" and "bab" -- 2 palindromes. What do you mean by seed, which ones are the seeds...? I think you need to flesh it out a little more (possibly with pseudocode) before you can leap to the claim that it's O(n). If not for me, then for the interviewers.

Anonymous on Oct 7, 2010

▲ 0 ▼

@VSP
yes that would be great if you can expand a little?

anonnymoous on Oct 11, 2010

▲ 0 ▼

The minimum number of palindromes are the number of letters that appear odd amount of times, if all the letters appear even amount of times the answer will be 1

jerom on Oct 13, 2010

▲

It can't be based just on the frequency of the letters. Consider ways we could form a word with two A's and two B's:

1

▼

If the word is ABBA, then it is already a palindrome, so we don't need to do anything. Hence the answer is 0 splits (or, equivalently, 1 final string)
If the word is ABAB, then we have to split it at least once (since it's not a palindrome), and we can do it by only splitting it once by either splitting it into ABA + B or A + BAB. Hence the answer is 1 split (or, equivalently, 2 final strings).

By the way, on the interview, what's more important that just getting it right is the ability to understand the question, show your ability to think through the problem, and think iteratively. First come up with a naive way and then improve upon that.

OP on Oct 13, 2010

▲

0

▼

One simple and inefficient solution is to go through all possible sub-strings (sub-arrays) starting with the biggest one, and see if we find a palindrome. [There are n * (n + 1) / 2 such sub arrays in the worst case.]

If we find a palindrome, we remove that substring from the string, and start over again.

Finally, we will be left with a string that has no palindromes of size 2 or greater. Stop and count all the chars because they are palindromes of size 1.

Complexity = O(n^3) because there are O(n^2) substrings and it takes n time to check for palindrome.

There must be a better solution.

Vijay on Oct 15, 2010

▲

3

▼

You can find all maximal substring palindromes in O(n) using a generalized suffix tree. After that, you can use a dynamic programming algorithm to find an optimal way to split the string into the fewest palindromes:
OPT(i) = min(1 + OPT(j-1) | substr(j,i) is a palindrome)

This has a worst case O(n^2) running time, although with some optimizations (based on deciding which palindromes are worth trying) we can make it run in O(n) on average. Of course, the constant in that running time is probably rather bulky :P

@vsp,jerom: I think you'd be onto something if we were allowed to rearrange characters in the string, but I don't think that's what the question is asking o_O

Ian G on Oct 21, 2010

▲

0

▼

The key intuition, in my opinion, is to start from the first character and look for all candidates that may end a palindrome. For that u need to scan the the string from the end and find all occurrences of the first character and create the corresponding substrings. Thereafter, you need to check if each substring is in fact a palindrome, if such, start working on reminder. If no palindrome was found you need to split the first character (as palindrome of single character) and move to second character.

Anonymous on Nov 18, 2010

▲

2

▼

This is a really interesting problem. I think Ian G is on the right track with generalized suffix trees and dynamic programming but I wouldn't expect anyone to write up complete code for that during an interview.

Note that the greedy algorithm where you find the largest palindromes first produces incorrect results. Consider this example: ABAABABA
Greedy algorithm: ABAABA + B + A = 3 palindromes
Correct answer: ABA + ABABA = 2 palidromes

Here's a straightforward dynamic programming solution. Let M[i, j] = the minimum number of palindromes in string s from index i to j, inclusive. The basis is M[i, i] = 1 because each single character is a palindrome. Then M[i, j] = min(M(i, k) + M(k+1, j) over all values k from i to j-1.

```
bool ispalindrome(const char *s, int i, int j)
{
  while (i < j) // Ignore middle char in odd length strings
    if (s[i++] != s[j--])
      return false;
  return true;
}

int minpalindromes(const char *s)
{
  int i, j, k, len;
  int n = strlen(s);
  int **M; // Allocate n x n array (not shown)

  for (len = 1; len <= n; len++) // Bottom-up: Start with short substrings.
    for (i = 0; i <= n - len; i++) {
      j = i + len - 1;
      if (ispalindrome(s, i, j))
        M[i][j] = 1;
      else {
        M[i][j] = len; // Assume worst-case. Improve on it below.
        for (k = i; k < j; k++) { // Consider splitting string at each position.
          int sum = M[i][k] + M[k+1][j];
          if (sum < M[i][j])
            M[i][j] = sum; // Save minimum value
        }
      }
    }
  return M[0][n-1]; // Return result for entire string
}
```

O(n^3) time and O(n^2) space. I suspect a suffix tree would improve performance but haven't coded it up yet.

lamont on Nov 22, 2010

▲
1
▼

Take the string and copy it to another string.
Now reverse the string. So now you have two string. The original and its reverse.
Use the standard "Longest Common Substring(LCS)" algorithm on these two strings.
Once the common substring is found, remove this substring from both strings and find the LCS of the resultant strings. Continue till either no more substrings are found with length greater than 1 or one/both of the strings become empty.

Complexity: It is O(n^2) which is the standard complexity of LCS. Also, removing substring from string can be done in O(1) by using memcpy if C/C++ is being used.

Anonymous on Dec 5, 2010

▲
3
▼

Reversing the string and looking for common substrings will give false positives. Consider this string: ABCDEXYEDCBA
It has no palindromes greater than length 1. Reversing it we have: ABCDEYXEDCBA
The longest common substring is ABCDE. It's falsely identified as a palindrome. What we should be looking for is a common substring which emanates in both directions *from the same position in the string*. That's where a suffix tree comes in handy.

Also, the greedy algorithm of finding the longest palindrome first can produce incorrect results. See my comment above (Nov 22) for an example.

lamont on Dec 14, 2010

▲
0
▼

The following solution works in O(n ^ 3), and is build upon dynamic programming (O(n^2)). If you find other way of checking if a substring is palindrome (not in O(n)), then you may decrease the time complexity.

```
public class SplitingPalindromes {
  public static String reverseIt(String source)
  {
    int i, len = source.length();
    StringBuffer dest = new StringBuffer(len);

    for (i = (len - 1); i >= 0; i--)
      dest.append(source.charAt(i));
    return dest.toString();
  }

  private static boolean isPalindrome(String a)
  {
    return a.endsWith(reverseIt(a));
  }

  public static void solve(String initial)
  {
    int[] cost = new int[initial.length()];
    int[] position = new int[initial.length()];

    cost[0] = 1;
    position[0] = -1;
    for(int i = 1; i cost[j-1] + 1)
          {
            cost[i] = cost[j-1] + 1;
            position[i] = j-1;
          }
        }
      }
    }
    Stack stack = new Stack();
    int pos1 = position[initial.length() - 1];
    int pos2 = initial.length() - 1;
    while(pos1 >= 0)
    {
      stack.add(initial.substring(pos1+1,pos2+1));
      pos2 = pos1;
      pos1 = position[pos2];
    }
    if(pos1 == -1)
    {
      stack.add(initial.substring(0,pos2+1));
    }

    while(!stack.isEmpty())
    {
      System.out.println(stack.pop());
    }
  }

  public static void main(String[] args) {
    solve("ABAABABA");
    solve("AAAAAAAAAAB");
    solve("ABC");
    solve("ABAC");
    solve("CABA");
  }

}
```

Victor on Jan 17, 2011

▲ Here is a simple way to build the set of all palindromes in O(n^2) time.

2 What you do is you take every character (or two adjacent characters) and find all palindromes such that your character is in the middle. It's pretty easy to see that you can do

▼ this in O(n) time to find all palindromes with a given character in the middle, because if you have a palindrome then knocking a character off each end leaves you with another palindrome. Therefore it takes O(n^2) to find them all.

Then, as others have mentioned, you can use dynamic programming to find the best covering.

Anonymous on Feb 25, 2011

Interviews > Software Engineer > Google

Add Answers or Comments

To comment on this, Sign In or Sign Up.

Sponsored Jobs

Ericsson RAN Product SW Developer
Ericsson-Worldwide – Nanjing

Account Exec 3, Direct Sales
Dell – Nanjing