LeetCode
Online Portal for IT Interview

- About Us
- Activity
- Forums
- Groups
- Interview Questions Online Judge
- Members

by 1337c0d3r

Longest Palindromic Substring Part II

Nov 20, 2011 at 11:07 pm in string by 1337c0d3r

> Given a string S, find the longest palindromic substring in S.

Note:
This is Part II of the article: Longest Palindromic Substring. Here, we describe an algorithm (Manacher's algorithm) which finds the longest palindromic substring in linear time. Please read Part I for more background information.

In my previous post we discussed a total of four different methods, among them there's a pretty simple algorithm with $O(N^2)$ run time and constant space complexity. Here, we discuss an algorithm that runs in O(N) time and O(N) space, also known as Manacher's algorithm.

Hint:
Think how you would improve over the simpler $O(N^2)$ approach. Consider the worst case scenarios. The worst case scenarios are the inputs with multiple palindromes overlapping each other. For example, the inputs: "aaaaaaaaa" and "cabcbabcbabcba". In fact, we could take advantage of the palindrome's symmetric property and avoid some of the unnecessary computations.

An O(N) Solution (Manacher's Algorithm):
First, we transform the input string, S, to another string T by inserting a special character '#' in between letters. The reason for doing so will be immediately clear to you soon.

For example: S = "abaaba", T = "#a#b#a#a#b#a#".

To find the longest palindromic substring, we need to expand around each T_i such that $T_{i-d} \ldots T_{i+d}$ forms a palindrome. You should immediately see that *d* is the length of the palindrome itself centered at T_i.

We store intermediate result in an array P, where P[i] equals to the length of the palindrome centers at T_i. The longest palindromic substring would then be the maximum element in P.

Using the above example, we populate P as below (from left to right):

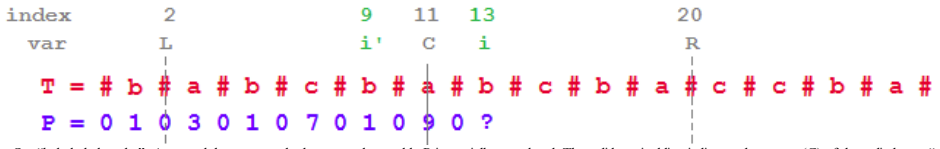
```
T = # a # b # a # a # b # a #
P = 0 1 0 3 0 1 6 1 0 3 0 1 0
```

Looking at P, we immediately see that the longest palindrome is "abaaba", as indicated by $P_6 = 6$.

Did you notice by inserting special characters (#) in between letters, both palindromes of odd and even lengths are handled graciously? (Please note: This is to demonstrate the idea more easily and is not necessarily needed to code the algorithm.)

Now, imagine that you draw an imaginary vertical line at the center of the palindrome "abaaba". Did you notice the numbers in P are symmetric around this center? That's not only it, try another palindrome "aba", the numbers also reflect similar symmetric property. Is this a coincidence? The answer is yes and no. This is only true subjected to a condition, but anyway, we have great progress, since we can eliminate recomputing part of P[i]'s.

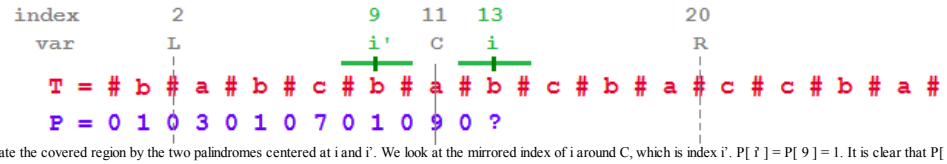
Let us move on to a slightly more sophisticated example with more some overlapping palindromes, where S = "babcbabcbaccba".



Above image shows T transformed from S = "babcbabcbaccba". Assumed that you reached a state where table P is partially completed. The solid vertical line indicates the center (C) of the palindrome "abcbabcba". The two dotted vertical line indicate its left (L) and right (R) edges respectively. You are at index i and its mirrored index around C is i'. How would you calculate P[i] efficiently?

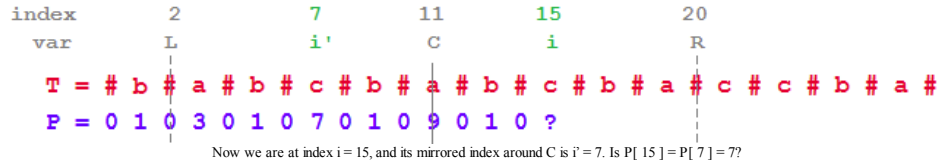
Assume that we have arrived at index i = 13, and we need to calculate P[13] (indicated by the question mark ?). We first look at its mirrored index i'

around the palindrome's center C, which is index i' = 9.



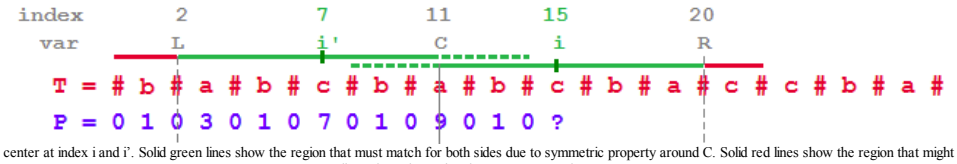
The two green solid lines above indicate the covered region by the two palindromes centered at i and i'. We look at the mirrored index of i around C, which is index i'. P[i'] = P[9] = 1. It is clear that P[i] must also be 1, due to the symmetric property of a palindrome around its center.

As you can see above, it is very obvious that P[i] = P[i'] = 1, which must be true due to the symmetric property around a palindrome's center. In fact, all three elements after C follow the symmetric property (that is, P[12] = P[10] = 0, P[13] = P[9] = 1, P[14] = P[8] = 0).



Now we are at index i = 15, and its mirrored index around C is i' = 7. Is P[15] = P[7] = 7?

Now we are at index i = 15. What's the value of P[i]? If we follow the symmetric property, the value of P[i] should be the same as P[i'] = 7. But this is wrong. If we expand around the center at T_{15}, it forms the palindrome "a#b#c#b#a", which is actually shorter than what is indicated by its symmetric counterpart. Why?



Colored lines are overlaid around the center at index i and i'. Solid green lines show the region that must match for both sides due to symmetric property around C. Solid red lines show the region that might not match for both sides. Dotted green lines show the region that crosses over the center.

It is clear that the two substrings in the region indicated by the two solid green lines must match exactly. Areas across the center (indicated by dotted green lines) must also be symmetric. Notice carefully that P[i '] is 7 and it expands all the way across the left edge (L) of the palindrome (indicated by the solid red lines), which does not fall under the symmetric property of the palindrome anymore. All we know is P[i] ≥ 5, and to find the real value of P[i] we have to do character matching by expanding past the right edge (R). In this case, since P[21] ≠ P[1], we conclude that P[i] = 5.

Let's summarize the key part of this algorithm as below:

if P[i'] ≤ R – i,
then P[i] ← P[i']
else P[i] ≥ P[i']. (Which we have to expand past the right edge (R) to find P[i].

See how elegant it is? If you are able to grasp the above summary fully, you already obtained the essence of this algorithm, which is also the hardest part.

The final part is to determine when should we move the position of C together with R to the right, which is easy:

If the palindrome centered at i does expand past R, we update C to i, (the center of this new palindrome), and extend R to the new palindrome's right edge.

In each step, there are two possibilities. If P[i] ≤ R – i, we set P[i] to P[i'] which takes exactly one step. Otherwise we attempt to change the palindrome's center to i by expanding it starting at the right edge, R. Extending R (the inner while loop) takes at most a total of N steps, and positioning and testing each centers take a total of N steps too. Therefore, this algorithm guarantees to finish in at most 2*N steps, giving a linear time solution.

```
// Transform S into T.
// For example, S = "abba", T = "^#a#b#b#a#$".
// ^ and $ signs are sentinels appended to each end to avoid bounds checking
string preProcess(string s) {
  int n = s.length();
  if (n == 0) return "^$";
  string ret = "^";
  for (int i = 0; i < n; i++)
    ret += "#" + s.substr(i, 1);

  ret += "#$";
  return ret;
}

string longestPalindrome(string s) {
  string T = preProcess(s);
```

```
  int n = T.length();
  int *P = new int[n];
  int C = 0, R = 0;
  for (int i = 1; i < n-1; i++) {
    int i_mirror = 2*C-i; // equals to i' = C - (i-C)

    P[i] = (R > i) ? min(R-i, P[i_mirror]) : 0;

    // Attempt to expand palindrome centered at i
    while (T[i + 1 + P[i]] == T[i - 1 - P[i]])
      P[i]++;

    // If palindrome centered at i expand past R,
    // adjust center based on expanded palindrome.
    if (i + P[i] > R) {
      C = i;
      R = i + P[i];
    }
  }

  // Find the maximum element in P.
  int maxLen = 0;
  int centerIndex = 0;
  for (int i = 1; i < n-1; i++) {
    if (P[i] > maxLen) {
      maxLen = P[i];
      centerIndex = i;
    }
  }
  delete[] P;

  return s.substr((centerIndex - 1 - maxLen)/2, maxLen);
}
```

Note:
This algorithm is definitely non-trivial and you won't be expected to come up with such algorithm during an interview setting. However, I do hope that you enjoy reading this article and hopefully it helps you in understanding this interesting algorithm. You deserve a pat if you have gone this far!

Further Thoughts:

- In fact, there exists a sixth solution to this problem — Using suffix trees. However, it is not as efficient as this one (run time O(N log N) and more overhead for building suffix trees) and is more complicated to implement. If you are interested, read Wikipedia's article about Longest Palindromic Substring.
- What if you are required to find the longest palindromic subsequence? (Do you know the difference between substring and subsequence?)

Useful Links:
» Manacher's Algorithm O(N) 时间求字符串的最长回文子串 (Best explanation if you can read Chinese)
» A simple linear time algorithm for finding longest palindrome sub-string
» Finding Palindromes
» Finding the Longest Palindromic Substring in Linear Time
» Wikipedia: Longest Palindromic Substring

← Longest Palindromic Substring Part I
Palindrome Number →

23 responses to *Longest Palindromic Substring Part II*

1. wayne said on November 21, 2011 *Reply*

   i think my solution is simpler than this one, the key point of my solution is:
   The center point of palindromic substring is always follow this pattern, either is "…..XyX….." or "….XX….".

   so you can scan once and then find those center point of palindromic substring and then expand it on each center points to find the one with maxium length.

   i ve already posted my java solution in the comments of Longest Palindromic Substring Part I

   - 
     1337c0d3r said on November 22, 2011 *Reply*

     Yes your solution is simpler but runs in O(N^2) worst case. It is already discussed in my previous post.

- wayne said on November 22, 2011 *Reply*

  i agreed, it is O(N^2) worst case, thanks.

  - 

    1337c0d3r said on November 22, 2011 *Reply*

    No problem.
    Basically this algorithm is an improvement over your method. It is using the symmetric property of a palindrome to eliminate some of the recomputations of palindrome's length, and amazingly improve it to a linear time solution.

2. Sreekar said on November 21, 2011 *Reply*

   Looks like this is O(N^2) algorithm as there is a while loop in for loop. Could you please clarify?

   - 

     1337c0d3r said on November 22, 2011 *Reply*

     Even with the extra while loop inside, it is guaranteed in the worst case the algorithm completes in 2*n steps.

     Think of how the i and right edge (R) relates. In the loop each time, you look if this index is a candidate to re-position the palindrome's center. If it is, you increment the existing R one at a time. See? R could only be incremented at most N steps. Once you incremented a total of N steps, it couldn't be incremented any more. It's not like you will increment R all the time in the while loop. This is called amortized O(1).

3. Sreekar said on November 22, 2011 *Reply*

   Got it. Thanks for the clarification. Great solution

   - 

     1337c0d3r said on November 22, 2011 *Reply*

     Thanks! Hope you understands it. Let me know if you have any more questions!

4. 

   flo said on November 22, 2011 *Reply*

   Great write-up. Thanks for the article

   - 

     1337c0d3r said on November 22, 2011 *Reply*

     Thanks!
     My goal of writing this article is to provide an intuitive way to understand the algorithm. I hope you really appreciate the beauty of this algorithm.

5. vaibhav said on November 24, 2011 *Reply*

   while explaining how to fill P[i] you mentioned
   "
   if P[i'] ≤ R – i,
   then P[i] ← P[i']
   else P[i] ≥ P[i']. (Which we have to expand past the right edge (R) to find P[i].
   "
   is the else statement right?? shouldnt be "else P[i'] ≥ P[i]"

6. bleu said on November 24, 2011 *Reply*

   It should rather be:
   else P[i] ≥ (R-i) (Which we have to expand past the right edge (R) to find P[i])
   .. Also note how coherent the reasoning in the bracket sounds now.

   Explanation :
   When P[i'] > R-i then all we know, by symmetry about C, is :
   P[i'] > R-i .. by obviousness

and
P[i] ≥ R-i .. by the meaning of R
From this we clearly cannot conclude upon max(P[i'], P[i])

7. Fei said on November 25, 2011 *Reply*

   Using suffix tree can do this in O(n). And building suffix tree can be also done in O(n): http://blog.csdn.net/g9yuayon/article/details/2574781

   But this algorithm is pretty cool too!

   - 
     1337c0d3r said on November 29, 2011 *Reply*

     Thanks Fei! I will look into that.

8. LB said on November 29, 2011 *Reply*

   Clear explanation. It could hardly be any better

   Thumbs up for elucidating this magic O(n) solution in such intuitive manner. You got talent to clearly expressing an algorithm, which I even missed in books like Cormen's Algorithm!

   - 
     1337c0d3r said on November 29, 2011 *Reply*

     Thanks! Good to know I've done my job — to introduce tricky but interesting algorithms in an intuitive manner.

9.
   Bahlul Haider said on December 1, 2011 *Reply*

   Really beautiful algorithm.

10. Googmeister said on December 6, 2011 *Reply*

    Wonderful writeup with great illustrations! I think there is one minor bug in your code: if s itself is a palindrome, then the following line accesses the array out of bounds.

    ```
    while (T[i + 1 + P[i]] == T[i - 1 - P[i]])
    ```

    - 
      1337c0d3r said on December 6, 2011 *Reply*

      ahh… you are right! Thanks for your sharp observation.

      I thought that I feel something is not right when I decided to add '$' both to the begin and the end of the input string. (It should be adding two different sentinels '^' and '$' to the begin and the end of the string. This should avoid bounds checking and the out of bounds problem)

11. Karthick said on December 12, 2011 *Reply*

    Thanks for such a lucid explanation. I had already visited all the references that you had suggested at the end. I was not able to understand the essence of it until I read yours.

    Well, I have one question.
    Is it possible to run the algorithm without using the '#','^','$' symbols?

    -
      Karthick said on December 12, 2011 *Reply*

      As an after-thought, I have one doubt.
      Why are we using the line
      P[i] = (R > i) ? min(R-i, P[i_mirror]) : 0;

      Can you please clarify this?

12. 
MSJ said on January 22, 2012 *Reply*

there are another two solutions
1) suffix array version preprocess requires N*logN query is O(N) http://www.mashengjun.info/?p=901
2) another O(N) solution http://www.mashengjun.info/?p=464 using up&down pointer

- 
1337c0d3r said on January 22, 2012 *Reply*

Did you try running your code through Online Judge? http://www.leetcode.com/onlinejudge
It did not pass all test cases.

Leave a reply

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code> <del datetime=""> <em> <i> <q cite=""> <strike> <strong>`

Post Comment

2 trackbacks

- **Longest Palindromic Substring Part I | LeetCode**

  on November 20, 2011

- **Palindrome Number | LeetCode**

  on January 4, 2012

Search

- About Us
- Activity
- Forums
- Groups
- Interview Questions Online Judge
- Members

LeetCode

- Log In
- Sign Up

↑ show LeetCode coding panel ↑