LeetCode
Online Portal for IT Interview

About | Activity | Discuss | Members | Online Judge

Search

# Longest Palindromic Substring Part II

by
1337c0d3r

November 20, 2011 in string

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 $T_{i-d}$ … $T_{i+d}$ 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):

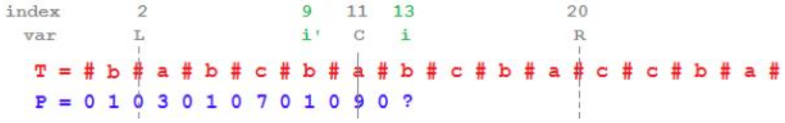
```
T = # a # b # a # a # b # a #
P = 0 1 0 3 0 1 6 1 0 3 0 1 0
```

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.

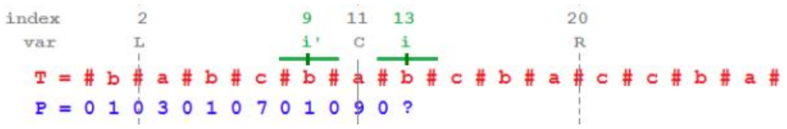
Let us move on to a slightly more sophisticated example with more some overlapping palindromes, where S = "babcbabcbaccba".



```
index       2           9   11  13          20
 var        L           i'  C   i           R
T = # b # a # b # c # b # a # b # c # b # a # c # c # b # a #
P = 0 1 0 3 0 1 0 7 0 1 0 9 0 ?
```

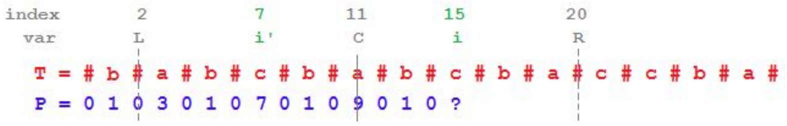
Above image shows T transformed from S = "babcbabcbaccba". Assumed that you reached a state where table P is partially completed. The solid vertical line indicates the center (C) of the palindrome "abcbabcba". The two dotted vertical line indicate its left (L) and right (R) edges respectively. You are 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.


index 2 9 11 13 20
var L i' C i R
T = # b # a # b # c # b # a # b # c # b # a # c # c # b # a #
P = 0 1 0 3 0 1 0 7 0 1 0 9 0 ?

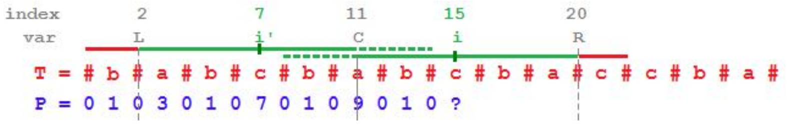
The two green solid lines above indicate the covered region by the two palindromes centered at i and i'. We look at the mirrored index of i around C, which is index i'. P[ i' ] = P[ 9 ] = 1. It is clear that P[ i ] must also be 1, due to the symmetric property of a palindrome around its center.

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).


index 2 7 11 15 20
var L i' C i R
T = # b # a # b # c # b # a # b # c # b # a # c # c # b # a #
P = 0 1 0 3 0 1 0 7 0 1 0 9 0 1 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?


index 2 7 11 15 20
var L i' C i R
T = # b # a # b # c # b # a # b # c # b # a # c # c # b # a #
P = 0 1 0 3 0 1 0 7 0 1 0 9 0 1 0 ?

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

Rating: 4.8/**5** (114 votes cast)

Longest Palindromic Substring Part II, 4.8 out of 5 based on 114 ratings

← Longest Palindromic Substring Part I    Palindrome Number →

## 100 responses to *Longest Palindromic Substring Part II*

**wayne** said on **November 21, 2011**

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

Reply

**-13**

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

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

Reply

**0**

**wayne** said on **November 22, 2011**

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

Reply

-6

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

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.

Reply

0

**Andres** said on **September 24, 2012**

Excellent post, I learnt a lot

Thanks!

Reply

-1

**GuojiaAgain** said on **January 20, 2014**

Your implementation is simpler but cost more time.

Reply

-1

**Sreekar** said on **November 21, 2011**

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

Reply

-4

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

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).

Reply

+5

**Vikas** said on **January 5, 2013**

How is it amortised O(1) ? I am confused from the definition from amortise.

Amortised to me means , for a worst case of runs of some operations its amortised cost over all of them

Reply

-2

**yy** said on **March 20, 2014**

Great, I see why it's amortized O(1) now. Thanks!

Reply

-2

**Jason** said on **September 16, 2014**

It seems a O(N^2) alg. For example, "abcdcba", go through the string needs N step, and the while loop needs N/2 step when meets character 'd', so O（N*N/2）=O(N^2)?

Reply

+1

**Sreekar** said on **November 22, 2011**

Got it. Thanks for the clarification. Great solution

Reply

+2

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

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

Reply

**-1**

**coderrush** said on **February 29, 2012**

I am wondering if the algorithm is O(NlgN) or O(N)?

Reply

**+1**

**flo** said on **November 22, 2011**

Great write-up. Thanks for the article

Reply

**+1**

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

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.

Reply

**+3**

**J** said on **May 5, 2012**

I would if I could understand it.

Reply

**0**

**vaibhav** said on **November 24, 2011**

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 ]“

Reply

**+8**

**ironmanz** said on **May 17, 2013**

I agree with you.

Reply

**0**

**bleu** said on **November 24, 2011**

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])

Reply

**+7**

**J** said on **May 5, 2012**

Don't see the coherency.

Reply

**0**

**ClaireQu** said on **September 17, 2014**

I agree with you

Reply

0

**Fei** said on **November 25, 2011**

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!

Reply

+2

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

Thanks Fei! I will look into that.

Reply

0

**LB** said on **November 29, 2011**

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!

Reply

+1

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

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

Reply

+1

**Bahlul Haider** said on **December 1, 2011**

Really beautiful algorithm.

Reply

0

**Googmeister** said on **December 6, 2011**

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]])
```

Reply

0

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

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)

Reply

0

**online chesstle** said on **February 15, 2014**

s.substr((centerIndex – 1 – maxLen)/2, maxLen); is there a bug here as the rest of the code is treating P[i] as the length of the palindrome on either side of the center and not as the total length of the palindrome.

Reply

0

**online chesstle** said on **February 15, 2014**

a fault in the code here? s.substr((centerIndex – 1 – maxLen)/2, maxLen);

the core algortihm is handling P[i] as the length of the palindrome on either side of the center and not as the total length of the palindrome.

Reply

0

**online chesstle** said on **February 15, 2014**

nevermind, accounting for the hashes.

Reply

0

**Karthick** said on **December 12, 2011**

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?

Reply

0

**Karthick** said on **December 12, 2011**

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?

Reply

0

**MSJ** said on **January 22, 2012**

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

Reply

0

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

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

Reply

0

**caopeng** said on **March 2, 2012**

The seconde is not O(N) it's O(N^2) i think…

Reply

0

**Caopeng** said on **March 1, 2012**

The first i_mirror is -1 which is less than 0 so there may be run time error?

Reply

+1

**Anonymous** said on **March 19, 2012**

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

It's really O(N)?

Reply

0

**Vyas** said on **April 9, 2012**

A Very nice explanation!!!!:)
One thing I could not understand… "In this case, since P[ 21 ] ≠ P[ 1 ], we conclude that P[ i ] = 5."..
In this statement, from what I have understood, I think it should be P[21] ≠ P[8]. Please correct me if I'm wrong…..

Reply 0

**Kareem** said on **May 1, 2012**

The conclusion of the algorithm above states that
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 ].

The first check should be P[ i' ] < R – i, as when they are equal the proper value of p[i] can not be fully determined with P[ i' ] only but needs to expand.
for example: string #b#b#a#b#a#b#a# with i = 9, c = 7, R = 12

Reply +1

**Li** said on **August 2, 2012**

Agree with you.

Reply 0

**J** said on **May 5, 2012**

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.

i is in green area (index 15). So I should have the same value as i[7]. but it doesn't.

So what is going on?

Reply 0

**J** said on **May 5, 2012**

The first two lines are quotes from the written explanation.

Reply 0

**Chev** said on **May 27, 2012**

Is there a problem in the following code? i_mirror will be -1 and P[i_mirror] will be out of boundary when C=0 and i =1, or do I miss something? Thanks.
…
int C = 0, R = 0;
for (int i = 1; i i) ? min(R-i, P[i_mirror]) : 0;
…
}

Reply 0

**Chev** said on **May 27, 2012**

I see! It is my mistake. Thanks!

Reply 0

**Chev** said on **May 27, 2012**

Here is the piece of code where I am confused:
int C = 0, R = 0;
for (int i = 1; i i) ? min(R-i, P[i_mirror]) : 0;

Reply 0

**Chev** said on **May 27, 2012**

Got it! I am clear now. Thanks!

Reply 0

**Chuanyou Li** said on **July 6, 2012**

Should P[i] R – i ??

Reply 0

**Noone** said on **September 5, 2012**

Have to disagree with the others. You have actually managed to complicate a simple algorithm!

The key idea is quite simple actually.

Reply

**0**

**Subramanian Ganapathy** said on **October 14, 2012**

Recurrence:

L[i] = max{L[i-1] + 1, #Arr[i-1 - L[i-1]…i] is univalue.
L[i-1]+2 #Arr[i] == Arr[i-1-L[i-1]]}
L[i]=1 otherwise.

Am i missing something here? this recurrrence solves it and it is a lot simpler.

Reply

**0**

**Subramanian Ganapathy** said on **October 14, 2012**

My bad 😐 my recurrence messes up the overlapping palindromes case, awesome solution and nice explanation. you deserve a pat on your back 🙂 thank you

Reply

**0**

**Karthick** said on **November 1, 2012**

The following is the implementation of the Manacher's algorithm without pre-processing the input string. It is a bit clumsy – sorry for that. I tested it using the online judge here and it seems to be working fine.

Language : java

```java
    public String longestPalindrome(String s) {
        if(s==null)
            return "";

        int len=s.length();
        int[] p=new int[2*len-1];
        p[0]=1;
        int R=0,C=0;
        int curLen,l,r,start;

        for(int i=1;ii) ? Math.min(curLen,p[iMirror]) : ( i%2==0 ? 1 : 0)
            if(i%2==0){
                l=(i/2-p[i]/2-1);
                r=(i/2+p[i]/2+1);
            }
            else{
                l=(i/2-p[i]/2);
                r=(i/2+p[i]/2+1);
            }

            while(l>=0 && rR){
                C=i;
                R=r-1;
            }
        }

        int maxIndex=getMaxIndex(p);
        if(maxIndex%2==0)
            start=(maxIndex/2-p[maxIndex]/2);
        else
            start=(maxIndex/2-p[maxIndex]/2 +1);

        return s.substring(start,start+p[maxIndex]);

    }

    int getMaxIndex(int p[]){
        int len=p.length;
        int maxIndex=0;
        for(int i=1;ip[maxIndex])
                maxIndex=i;
        }
        return maxIndex;
    }
```

Reply

**-2**

**Karthick** said on **November 1, 2012**

Sorry,there seems to be some problem – some part of the code seems to be omitted when I post my code using the

tag. So, I am posting it without the code tag.

```
public String longestPalindrome(String s) {

if(s==null)
return "";

int len=s.length();
int[] p=new int[2*len-1];
p[0]=1;
int R=0,C=0;
int curLen,l,r,start;
for(int i=1;ii) ? Math.min(curLen,p[iMirror]) : ( i%2==0 ? 1 : 0) );
if(i%2==0){
l=(i/2-p[i]/2-1);
r=(i/2+p[i]/2+1);
}
else{
l=(i/2-p[i]/2);
r=(i/2+p[i]/2+1);
}

while(l>=0 && rR){
C=i;
R=r-1;
}

}

int maxIndex=getMaxIndex(p);
if(maxIndex%2==0)
start=(maxIndex/2-p[maxIndex]/2);
else
start=(maxIndex/2-p[maxIndex]/2 +1);

return s.substring(start,start+p[maxIndex]);

}

int getMaxIndex(int p[]){
int len=p.length;
int maxIndex=0;
for(int i=1;ip[maxIndex])
maxIndex=i;
}
return maxIndex;
}
```

Reply

0

**abhay** said on **January 5, 2013**

nice explaination thanks for article..
😃

Reply

0

**Naveen Kumar** said on **January 16, 2013**

I am stuck at summarized part of this algo.
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 ].)
how could we say which one be large for else part.?
even in example for i=15,p[i]=5,i'=7 p[i']=7;
i m confused here. plz help me.
Thanks..

Reply

0

**Mony Sim** said on **January 20, 2013**

try simple solution.

```
bool is palindrome(string s)
{ int len=s.length(), a=0, b=len-1;
while(a<b)
{
```

```
if(s[a]!=s[b])
return false;
}
return true;
}
```

Reply

-2

**Mony Sim** said on **January 20, 2013**

try simple solution.

```
bool is palindrome(string s)
{ int len=s.length(), a=0, b=len-1;
while(a<b)
{
if(s[++a]!=s[–b])
return false;
}
return true;
}
```

Reply

0

**rajat rastogi** said on **February 2, 2013**

Dude your solution is O(n^2) take a case of a string aaaaaa, palindrome length is 6 and solution for this string confirms O(n^2)

Reply

0

**William Gozali** said on **April 14, 2013**

I don't think so.

Take a look at the explanation, it is guaranteed that the operation needed will not exceed O(N). Maybe you should try to simulate the algorithm with that input

Reply

0

**Žilvinas** said on **February 4, 2013**

Thanks so much, exelent tutorial!

Reply

0

**Nipun Poddar** said on **February 6, 2013**

really nice one..helpedme to learn a lot.. 🙂

Reply

0

**Kuldeep Yadav** said on **March 15, 2013**

GOOD WORK 🙂

Reply

0

**Ayaskant Swain** said on **April 25, 2013**

Wonderful post. Thanks for posting these kind of problems and their solutions which will help in interviews. I have a question in this part-II solution.

I think the complexity will be still O(n * n), since you are traversing the string twice actually. One for modifying the original input string to insert characters ^,#,$ and then again you will do another traversal from the beginning to end of the string to search for actual palindromes in the string.

Reply

0

**lagrange** said on **June 2, 2013**

我觉得那个中文的blog, p[i]表示向左/右延展的长度, 比p[i]表示整个substr的长度要更容易理解一些

Reply

+1

**zhuyao** said on **September 2, 2014**

aglee!

Reply

0

**shivali** said on **June 17, 2013**

can you please xplain how you calculated the complexity of the above algorithm

Reply

0

**shivali** said on **June 18, 2013**

```
//longest pallindrome in a string(c++)
#include
#include
#define lsfor(i,a,b) for(i=a;i<b;i++)

using namespace std;
char str[100];
int n,curr_len=0,max_len=1,l,r,t;

int main()
{
int i;
cout<<"enter no.of test cases:"<>t;
while(t–)
{
cout<<"please enter your string"<>str;
n=strlen(str);
if(n==1)
{cout<<"1"<<endl;
return(0);}

if(n==2)

{cout<<"2"<=0&&r<=n-1)
{
if(str[l]==str[r])
{
curr_len+=2;
if(max_len<curr_len)
max_len=curr_len;
l–,r++;
}

else
break;
}
}
if(curr_len==1)
cout<<"sorry no palindrome"<<endl;
else
cout<<max_len<<endl;
}
return(0);

}
```

Reply

0

**shivali** said on **June 18, 2013**

```
//edited://longest pa
#include
#include
#define lsfor(i,a,b) for(i=a;i<b;i++)

using namespace std;
char str[100];
int n,curr_len=0,max_len=1,l,r,t;

int main()
{
int i;
cout<<"enter no.of test cases:"<>t;
while(t–)
{
cout<<"please enter your string"<>str;
n=strlen(str);
if(n==1)
{cout<<"1"<<endl;
return(0);}
if(n==2)

{cout<<"2"<=0&&r<=n-1)
{
if(str[l]==str[r])
{
curr_len+=2;
if(max_len<curr_len)
max_len=curr_len;
l–,r++;
}

else
break;
}
}
if(max_len==1)
cout<<"sorry no palindrome"<<endl;
else
cout<<max_len<<endl;
}
return(0);

}
```

Reply

0

**jackyhou** said on **July 3, 2013**

```
int find_long_palindrome_line(char * str,char *substr)
{
if(str==NULL)
return -1;

int len=strlen(str);
if(len==0)
return 1;

int i=0;
int j=len-1;
int end = len-1;
int curindex=0;

int tmp_len=len*2+1;
char * tmp_allstr=(char *)malloc(sizeof(char)*(tmp_len));
int *i_arr=(int *)malloc(sizeof(int)*(tmp_len));

int index4_tmp_allstr=0;
for(int i=0;i<len;i++)
{
tmp_allstr[index4_tmp_allstr++]='#';
tmp_allstr[index4_tmp_allstr++]=str[i];
}
tmp_allstr[index4_tmp_allstr]='#';

memset(i_arr,0,sizeof(int)*(tmp_len));

for(int i=2;i0 && right <(tmp_len))
{
if(tmp_allstr[left]==tmp_allstr[right])
{
i_arr[i]++;
}
else
{
break;
}
}
else
{
break;
}
}
}

int findI=0;
int findMaxLen=0;
for(int i=2;ifindMaxLen)
{
findMaxLen=i_arr[i];
findI=i;
}

}
int realSubLen=0;
if(findMaxLen>0)
```

```
{
for(int i=findI-(findMaxLen)+1;i<=findI+(findMaxLen)-1;i=i+2)
{
substr[realSubLen++]=tmp_allstr[i];
}
substr[realSubLen]=";
}

free (tmp_allstr);
tmp_allstr=NULL;
free (i_arr);//=(int *)malloc(sizeof(int)*(tmp_len));
i_arr=NULL;
return 1;
}
```

Reply

0

**JS** said on **July 10, 2013**

Won't the preprocess() take quadratic time? substr() is linear is time.

Reply

+1

**Saber** said on **July 23, 2013**

The algorithm you write is wrong. I believe it is because u type it faster than what u think:)
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 ].

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

Reply

0

**shuaiyangyang** said on **August 20, 2013**

if p[i] > R -i then p[i] should equal to R-i.

only if p[i] = R-i then you have to expand

Reply

-1

**shuaiyangyang** said on **August 20, 2013**

Hey 1337, The relation should be:

if P[ i' ] < R – i,

then P[ i ] ← P[ i' ]

else if P[ i ] = R – i. (Which we have to expand past the right edge (R) to find P[ i ].

else p[i] = R – i

Reply

-1

**Nam** said on **January 5, 2015**

I agree with you.

However, for worse case scenario of strings such as "aaaaaaa", the run time is O(n^2).
I think this algorithm is O(n) on average.

Reply

0

**Fentoyal** said on **September 1, 2013**

This problem could be solved in O(n) time and O(1) space.
The test case attached here is what leetcode claims "wrong answer", but I run it on my computer and it is exactly the same as the "expected answer". I do not know why tho.

```
#include
#include
using namespace std;
class Solution {
int longestPalindromSubstrHelper(const string & str, bool is_even, int &cur_max_pivot, int &
cur_max_radius)
{
int cur_radius = 0, cur_pivot = 0;
for (size_t i = 0; i < str.size(); ++i)
{

cur_radius = i – cur_pivot;
// cout<<cur_radius<<" "<<cur_pivot<<" "<<i<= 0 && str[cur_pivot - cur_radius + is_even] == str[i]
&&cur_radius >= cur_max_radius)
{
cur_max_radius = cur_radius;
cur_max_pivot = cur_pivot;
}
while ((cur_pivot – cur_radius + is_even < 0 ||str[cur_pivot - cur_radius +is_even] != str[i] ) &&
cur_pivot < i)
{
cur_pivot++;
cur_radius–;
//cout<<cur_radius<<" "<<cur_pivot<<" "<<i<<endl;
}
}
return 2 * cur_max_radius + !is_even;

}

public:
string longestPalindrome(string str) {
// Start typing your C/C++ solution below
// DO NOT write int main() function

int even_radius = 0, even_pivot = 0, odd_radius = 0, odd_pivot = 0;
int even_len = longestPalindromSubstrHelper(str, 1, even_pivot, even_radius );

int odd_len = longestPalindromSubstrHelper(str, 0, odd_pivot, odd_radius);
//cout<<even_len<<odd_len<= odd_len)
{
return str.substr(even_pivot – even_radius + 1, 2 * even_radius);
}

return str.substr(odd_pivot – odd_radius, 2 * odd_radius+1);

}

};

int main()
{
Solution s;
//cout<<longestPalindromSubstrHelper("abababab",1)<<endl;
cout<<s.longestPalindrome
("32101232100123210012321001232100123210012321001232100123210012321001232100123210012321001232100123210012321001232100123210
<<endl;
// cout<<longestPalindromSubstr("ababababa")<<endl;
}
```

Reply

+1

**STANLEY** said on **October 28, 2013**

if the string is atabcccbatabccccbata the cause the longest p. string's middle is 't' in central. But as your algorithm, the when the i =t R is larger than i; so the t is gonna to equals to the t at the second place. is 3. how could this figure out?

Reply

0

**stanley** said on **October 28, 2013**

I'm not sure that cause you mean the R is the position which generated by the pivot's left bound of previous D. And when you met i < R it will equals to the min(), so If the string is atabcccbatabcccbata the when D at the position of the second c, the R is gonna to be the a behind the second t. But so when the i turn to the t, t should equals to the min(), but t is the pivot of the longest substring, so how this work in the algorithm? I've got a little confuse.

Reply

0

**Atul Kumar** said on **November 15, 2013**

Great !!!

Reply

0

**Mark** said on **November 30, 2013**

A misleading post.

Reply

+2

**Albert Chen** said on **February 2, 2014**

Building suffix tree is O(n) and preprocessing a general tree for O(1) LCA queries is O(n).

We can build an extended suffix tree by inserting suffixes of reversed text into the same tree.

After these constructions, we enumerate the mid point in text (and we know the corresponding point in the reversed text). By looking at the LCA of the corresponding points in text and reversed text. We can decide in constant time that the longest palindrome from this mid point.

So we will be able to solve the problem in linear time (for odd length text.)

For more details, look into the book Algorithms on Strings, Trees and Sequences.

Reply

0

**Leet** said on **February 3, 2014**

```
for (int i = 1; i < t.length-1; i++) {

                while (t[i + (1 + p[i])] == t[i - (1 + p[i])])
                      p[i]++;
             }
```

this itself wil give the solution..
Whats the need of center and right part.. plz expalin me

Reply

0

**online chesstle** said on **February 15, 2014**

s.substr((centerIndex – 1 – maxLen)/2, maxLen); is there a bug here as the rest of the code is treating P[i] as the length of the palindrome on either side of the center and not as the total length of the palindrome. maxlen is derived from P…

Reply

0

**Acmerblog** said on **March 22, 2014**

great job.I'm not sure that cause you mean the R is the position which generated by the pivot's left bound of previous D. And when you met i < R it will equals to the min(), so If the string is atabcccbatabcccbata the when D at the position of the second c, the R is gonna to be the a behind the second t. But so when the i turn to the t, t should equals to the min(), but t is the pivot of the longest substring, so how this work in the algorithm? I've got a little confuse.

Reply

+1

**Donne** said on **April 9, 2014**

Why is the algoritmn O(n). The while loop where we attempt to expand the palindrome would need O(n) in worst case. eg. in case of b of the center. We are indirectly traversing all the nodes in that loop. So wont the order be O(n^2) ?

Reply

0

**Alex** said on **April 11, 2014**

Excellent post…. i don't find any other posts which is as good and elaborate as this !! Good job.

Reply

0

**programmingmonkey** said on **April 29, 2014**

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

perhaps this part needs a little more index checking to avoid segment fault?

Reply

0

**Bob** said on **June 3, 2014**

Thank you for your clear explanation！！

Reply

0

**ash** said on **August 4, 2014**

for (int i = 1; i i) ? min(R-i, P[i_mirror]) : 0;

//There is a problem here for i=1 and first run(C=0) through the loop i_mirror = -1;
P[-1] … you are accessing wrong address..

Reply

0

**killa** said on **August 13, 2014**

You've no idea how difficult it is for a chinese to understand your method. But you are brilliant. Thanks.

Reply

0

**ccen** said on **August 20, 2014**

learn a lot, thank you

Reply

0

**Dinesh** said on **August 30, 2014**

Excellent O(N) time complexity solution. Thanks for the post 😃

Reply

0

**B_Khan** said on **November 2, 2014**

Great Explanation. Thanks !

Reply

0

**adiggo** said on **November 6, 2014**

it is so brilliant, but so hard to come up with.

Reply

0

**sagiv** said on **November 20, 2014**

great post! thanks

Reply

0

**Leo** said on **November 28, 2014**

Using symmetry to avoid recalculation. Simple and elegant.

Reply

0

**Nick Nussbaum** said on **January 5, 2015**

Rather than create S I made lambda to provide the equivalent of accessing S using the input string.
A few more operations in the inner loop, as a tradeoff for space
I also added a little bounds checking

```
string LongestPalindrome(const string input)
{
    int c = 0;
    int max = 0;
    // create an indexed accessor for a virtual string S which has $a$b$c
    auto S_at = [&input](int index)->char { return ((index & 1) ? input[i
    int sizeP = (input.length() * 2) + 1;
    int* P = new int[sizeP];
    P[0] = 0;
    // find longest Palindromes forcentered on each index in S
    for (int i = 1; i  i) ? min(P[c - (i - c)], max - i) : 0;
        // Try to expand Palindrome but not past string boundaries
        int bounds = min(sizeP - i - 1, i - 1);
        while (bounds-- >= 0 && S_at(i + P[i] + 1) == S_at(i - P[i] - 1))
        {
            P[i]++;
        }
        // If palindrome was extend past max then update Center to i and
        if (i + P[i] > max)
        {
            c = i;
            max = i + P[i];
        }
    }
    auto maxP = std::max_element(P, P + sizeP);
    int start = (maxP - P - *maxP)/2;
    return input.substr(start, *maxP);
}
```

Reply

0

**Nick Nussbaum** said on **January 5, 2015**

The seems to have mangled the code badly

```
string LongestPalindrome(const string input)
{
int c = 0;
int max = 0;
// create an indexed accessor for a virtual string S which has $a$b$c$ for input
string abc
auto S_at = [&input](int index)->char { return ((index & 1) ? input[index / 2] :
'$'); };
int sizeP = (input.length() * 2) + 1;
int* P = new int[sizeP];
P[0] = 0;
// find longest Palindromes for centered on each index in S
for (int i = 1; i i) ? min(P[c - (i - c)], max - i) : 0;
// Try to expand Palindrome but not past string boundaries
int bounds = min(sizeP - i - 1, i - 1);
while (bounds-- >= 0 && S_at(i + P[i] + 1) == S_at(i - P[i] - 1))
{
P[i]++;
}
// If palindrome was extend past max then update Center to i and update the
right edge
if (i + P[i] > max)
{
c = i;
max = i + P[i];
}
}
auto maxP = std::max_element(P, P + sizeP);
int start = (maxP - P - *maxP)/2;
return input.substr(start, *maxP);
}
```

Reply

0

**Nick Nussbaum** said on **January 5, 2015**

Still mangling things.
Here's the mangled lines of the first post

```
    // find longest Palindromes centered on each index in S

    for (int i = 1; i  i) ? min(P[c - (i - c)], max - i) : 0;
```

Reply

0

**yuhang** said on **January 15, 2015**

I'm scared a while pays a whole lot more. "Zhejiang Institute with regards to Sports And Physical Eduction timberland Homme . Professor Cong Wu Titled Ping said this type of show."It ought to be considered that, tactic target, a greater number of a depiction great importance. Regarding feed this sort international reach and international methods, Meiyouzhengfu provider or perhaps you grueling." Includes Xueyuanjiaoshou Ph.D., Site Wuhan, Deputy Admin along with Pastime Ju have Tongyangguandian Difficulties of a good.You have totally different research. Shanxi To Reiter for decades which corporate entity's fundamental professional Qin, can't do primary example, ones national sports field a new new, sunrise community symptoms together with the in order to do it aspiration.

Reply

0

## Leave a reply

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

Name *

Email *

Website

*To embed your code, please use <code>your code here</code>.*

You may use the `<code>` tag to embed your code.

Post Comment

## 21 trackbacks

**Longest Palindromic Substring Part I | LeetCode**

*on November 20, 2011*

**Palindrome Number | LeetCode**

*on January 4, 2012*

**Longest Palindromic Substring « Interview Algorithms**

*on August 16, 2012*

**Longest Palindromic Substring » KEVIN'S BLOG**

*on October 27, 2012*

**Longest Palindromic Substring » Kevin's Tech Blog**

*on February 25, 2013*

**leetcode: Longest Palindromic Substring solution | 烟客旅人 sigmainfy**

*on June 5, 2013*

**[Leetcode]Longest Palindromic Substring | Wei's blog**

*on October 28, 2013*

**Manacher's algorithm (algorithm to find longest palindrome substring in linear time) | Ask Programming & Technology**

*on November 8, 2013*

**[LeetCode]Longest Palindromic Substring | MistySoul**

*on November 9, 2013*

**Longest Palindromic Substring | Jiting**

*on January 12, 2014*

**Solution to gamma2011 (Count-Palindromic-Slices) by codility | Code Says**

*on February 5, 2014*

**Longest Palindromic Substring | Elvis_Yu**

*on February 7, 2014*

**(Leetcode) Longest Palindromic Substring | Changhaz's Codeplay**

*on April 21, 2014*

**CodeNirvana: Palidromic Number | {.SMan.Soft.}**

*on May 9, 2014*

**CodeNirvana: Palidromic Numbers | {.SMan.Soft.}**

*on May 9, 2014*

**学习manacher（最长公共回文串算法）_58,AM,BBA,c#,CA,CTE,DC,EF,F1,FB,GE,HTML,Http,ie,IT,LeetCode,Logs,ROM,Set,Span,String,WP,XP,代码,位置,关系,博客,处理方法,字符串,学习,定义,怎么办,方法,时间,模式,比较,现在,矛盾**

*on September 5, 2014*

**LeetCode 最长回文子字符串 « 狐说的小站**

*on September 18, 2014*

**LeetCode in Swift: Longest Palindromic Substring – Guan Gui**

*on September 25, 2014*

**Data Structures and Algorithms Tutorials | TheShayna.Com**

*on November 10, 2014*

**[LeetCode] Longest Palindromic Substring 解题报告 | 水中的鱼(镜像)**

*on December 31, 2014*

**Python：最长回文子串**

*on January 8, 2015*

About Activity Discuss Members Online Judge