

Finding all sub-palindromes in $O(N)$

Table of Contents

- [Statement](#)
- [More precise statement](#)
- [Solution](#)
- [Trivial algorithm](#)
- [Manacher's algorithm](#)
- [Complexity of Manacher's algorithm](#)
- [Implementation of Manacher's algorithm](#)
- [Problems](#)

Statement

Given string s with length n . Find all the pairs (i, j) such that substring $s[i \dots j]$ is a palindrome. String t is a palindrome when $t = t_{rev}$ (t_{rev} is a reversed string for t).

More precise statement

It's clear that in the worst case we can have $O(n^2)$ palindrome strings, and at the first glance it seems that there is no linear algorithm for this problem.

But the information about the palindromes can be kept in **a more compact way**: for each position $i = 0 \dots n - 1$ we'll find the values $d_1[i]$ and $d_2[i]$, denoting the number of palindromes accordingly with odd and even lengths with centers in the position i .

For instance, string $s = abababc$ has three palindromes with odd length with centers in the position $s[3] = b$, i. e. $d_1[3] = 3$:

$$\begin{array}{ccccccc} & & \overbrace{a & b & a}^{d_1[3]=3} & & \\ & & & b & & & \\ & & & \underbrace{\hspace{1.5em}}_{s_3} & & & \\ a & b & a & & b & a & bc \end{array}$$

And string $s = cbaabd$ has two palindromes with even length with centers in the position $s[3] = a$, i. e. $d_2[3] = 2$:

$$\begin{array}{ccccccc} & & \overbrace{c & b & a}^{d_2[3]=2} & & \\ & & & a & & & \\ & & & \underbrace{\hspace{1.5em}}_{s_3} & & & \\ c & b & a & & a & b & d \end{array}$$

So the idea is that if we have a sub-palindrome with length l with center in some position i , we also have sub-palindromes with lengths $l - 2, l - 4$ etc. with centers in i . So these two arrays $d_1[i]$ and $d_2[i]$ are enough to keep the information about all the sub-palindromes in the string.

It's a surprising fact that there is an algorithm, which is simple enough, that calculates these "palindromity arrays" $d_1[]$ and $d_2[]$ in linear time. The algorithm is described in this article.

Solution

In general, this problem has many solutions: with [String Hashing](#) it can be solved in $O(n \cdot \log n)$, and with [Suffix Trees](#) and fast LCA this problem can be solved in $O(n)$.

But the method described here is **sufficiently** simpler and has less hidden constant in time and memory complexity. This algorithm was discovered by **Glenn K. Manacher** in 1975.

Trivial algorithm

To avoid ambiguities in the further description we denote what "trivial algorithm" is.

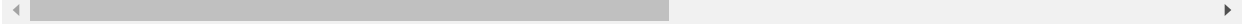
It's the algorithm that does the following. For each center position i it tries to increase the answer by one until it's possible, comparing a pair of corresponding characters each time.

Such algorithm is slow, it can calculate the answer only in $O(n^2)$.

The implementation of the trivial algorithm is:

```
vector<int> d1(n), d2(n);
for (int i = 0; i < n; i++) {
    d1[i] = 1;
    while (0 <= i - d1[i] && i + d1[i] < n &&
           d1[i]++);
}

d2[i] = 0;
while (0 <= i - d2[i] - 1 && i + d2[i] < n
       d2[i]++);
}
```



Manacher's algorithm

We describe the algorithm to find all the sub-palindromes with odd length, i. e. to calculate $d_1[]$; the solution for all the sub-palindromes with even length (i. e. calculating the array $d_2[]$) will be a minor modification for this one.

For fast calculation we'll keep the **borders** (l, r) of the rightmost found sub-palindrome (i. e. the palindrome with maximal r). Initially we assume $l = 0, r = -1$.

So, we want to calculate $d_1[i]$ for the next i , and all the previous values in $d_1[]$ have been already calculated.

We do the following:

- If i is outside the current sub-palindrome, i. e. $i > r$, we'll just launch the trivial algorithm.

So we'll increase $d_1[i]$ consecutively and check each time if the current substring $[i - d_1[i] \dots i + d_1[i]]$ is a palindrome. When we find first divergence or meet the boundaries of s , we'll stop. In this case we've finally calculated $d_1[i]$. After this, we must not forget to update (l, r) .

- Now consider the case when $i \leq r$. We'll try to extract some information from the already calculated values in $d_1[]$. So, let's flip the position i inside the sub-palindrome (l, r) , i. e. we'll get the position $j = l + (r - i)$, and we'll look on the value $d_1[j]$. Because j is the position symmetrical to i , we'll **almost always** can assign $d_1[i] = d_1[j]$. Illustration of this (palindrome around j is actually "copied" into the palindrome around i):

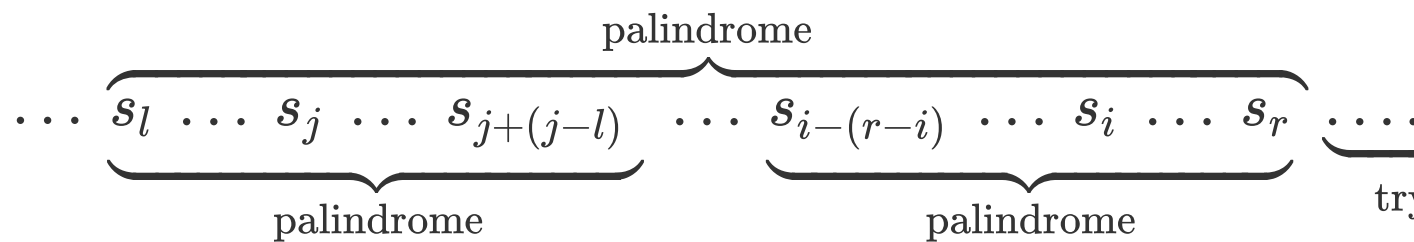
$$\dots s_l \dots \underbrace{s_{j-d_1[j]+1} \dots s_j \dots s_{j+d_1[j]-1}}_{\text{palindrome}} \dots \overbrace{s_{i-d_1[i]+1} \dots s_{i+d_1[i]-1}}^{\text{palindrome}} \dots$$

But there is a **tricky case** to be handled correctly: when the "inner" palindrome reaches the borders of the "outer" one, i. e. $j - d_1[j] + 1 \leq l$ (or, which is the same, $i + d_1[i] - 1 \geq r$). Because the symmetry outside the "outer" palindrome is not guaranteed, just assigning $d_1[i] = d_1[j]$ will be incorrect: we have not enough data to state that the palindrome in the position i has the same length.

Actually, we should "cut" the length of our palindrome, i. e. assign $d_1[i] = r - i$, to handle such situations

correctly. After this we'll run the trivial algorithm which will try to increase $d_1[i]$ while it's possible.

Illustration of this case (the palindrome with center j is already "cut" to fit the "outer" palindrome):



It is shown on the illustration that, though the palindrome with center j could be larger and go outside the "outer" palindrome, in the position i we can use only the part that entirely fits into the "outer" palindrome. But the answer for the position i can be much longer than this part, so next we'll run our trivial algorithm that will try to grow it outside our "outer" palindrome, i. e. to the region "try moving here".

At the end, it's necessary to remind that we should not forget to update the values (l, r) after calculating each $d_1[i]$.

Also we'll repeat that the algorithm was described to calculate the array for odd palindromes $d_1[]$, the algorithm is similar for the array of even palindromes $d_2[]$.

Complexity of Manacher's algorithm

At the first glance it's not obvious that this algorithm has linear time complexity, because we often run the naive algorithm while searching the answer for a particular position.

But more careful analysis shows that the algorithm is linear however. We need to mention [Z-function building](#)

algorithm which looks similar to this algorithm and also works in linear time.

Actually, we can notice that every iteration of trivial algorithm makes r increase by one. Also r cannot be decreased during the algorithm. So, trivial algorithm will make $O(n)$ iterations in total.

Also, other parts of Manacher's algorithm work obviously in linear time, we get $O(n)$ time complexity.

Implementation of Manacher's algorithm

For calculating $d_1[]$, we get the following code:

```
vector<int> d1(n);
for (int i = 0, l = 0, r = -1; i < n; i++) {
    int k = (i > r) ? 1 : min(d1[l + r - i], r
    while (0 <= i - k && i + k < n && s[i - k]
        k++;
    }
    d1[i] = k--;
    if (i + k > r) {
        l = i - k;
        r = i + k;
    }
}
```

For calculating $d_2[]$, the code looks similar, but with minor changes in arithmetical expressions:

```
vector<int> d2(n);
for (int i = 0, l = 0, r = -1; i < n; i++) {
    int k = (i > r) ? 0 : min(d2[l + r - i + 1
    while (0 <= i - k - 1 && i + k < n && s[i
        k++;
    }
```

```
    }  
    d2[i] = k--;  
    if (i + k > r) {  
        l = i - k - 1;  
        r = i + k ;  
    }  
}
```

Problems

UVA #11475 "Extend to Palindrome"

(c) 2014-2019 translation by <http://github.com/e-maxx-eng>

07:80/112