Last update: March 24, 2025 Translated From: e-maxx.ru

Manacher's Algorithm - Finding all sub-palindromes in $O(N)$

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

In the worst case string might have up to $O(n^2)$ palindromic substrings, 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 compact way**: for each position i we will find the number of non-empty palindromes centered at this position.

Palindromes with a common center form a contiguous chain, that is if we have a palindrome of length l centered in i, we also have palindromes of lengths $l - 2$, $l - 4$ and so on also centered in i. Therefore, we will collect the information about all palindromic substrings in this way.

Palindromes of odd and even lengths are accounted for separately as $d_{odd}[i]$ and $d_{even}[i]$. For the palindromes of even length we assume that they're centered in the position i if their two central characters are $s[i]$ and $s[i-1]$.

For instance, string $s = ababab c$ has three palindromes with odd length with centers in the position $s[3] = b$, i. e. $d_{odd}[3] = 3$:

$$a \overbrace{b\, a \underbrace{\, b \,}_{s_3} a\, b}^{d_{odd}[3]=3} c$$

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

$$c \overbrace{b\, a \underbrace{\, a \,}_{s_3} b}^{d_{even}[3]=2} d$$

It's a surprising fact that there is an algorithm, which is simple enough, that calculates these "palindromity arrays" $d_{odd}[]$ and $d_{even}[]$ 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.

Another modern way to solve this problem and to deal with palindromes in general is through the so-called palindromic tree, or eertree.

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 as long as it's possible, comparing a pair of corresponding characters each time.

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

The implementation of the trivial algorithm is:

```
vector<int> manacher_odd_trivial(string s) {
    int n = s.size();
    s = "$" + s + "^";
    vector<int> p(n + 2);
    for(int i = 1; i <= n; i++) {
        while(s[i - p[i]] == s[i + p[i]]) {
            p[i]++;
        }
    }
    return vector<int>(begin(p) + 1, end(p) - 1);
}
```

Terminal characters `$` and `^` were used to avoid dealing with ends of the string separately.

Manacher's algorithm

We describe the algorithm to find all the sub-palindromes with odd length, i. e. to calculate $d_{odd}[]$.

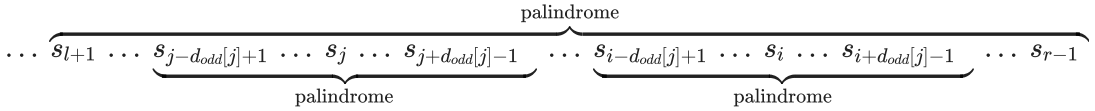
For fast calculation we'll maintain the **exclusive borders** (l, r) of the rightmost found (sub-)palindrome (i. e. the current rightmost (sub-)palindrome is $s[l+1]s[l+2]\dots s[r-1]$). Initially we set $l = 0, r = 1$, which corresponds to the empty string.

So, we want to calculate $d_{odd}[i]$ for the next i, and all the previous values in $d_{odd}[]$ have been already calculated. We do the following:

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

  So we'll increase $d_{odd}[i]$ consecutively and check each time if the current rightmost substring $[i - d_{odd}[i] \dots i + d_{odd}[i]]$ is a palindrome. When we find the first mismatch or meet the boundaries of s, we'll stop. In this case we've finally calculated $d_{odd}[i]$. After this, we must not forget to update (l, r). r should be updated in such a way that it represents the last index of the current rightmost sub-palindrome.

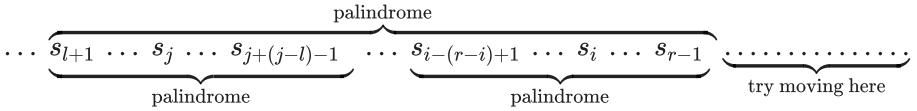
- Now consider the case when $i \le r$. We'll try to extract some information from the already calculated values in $d_{odd}[]$. So, let's find the "mirror" position of i in the sub-palindrome (l, r), i.e. we'll get the position $j = l + (r - i)$, and we check the value of $d_{odd}[j]$. Because j is the position symmetrical to i with respect to $(l + r)/2$, we can **almost always** assign $d_{odd}[i] = d_{odd}[j]$. Illustration of this (palindrome around j is actually "copied" into the palindrome around i):



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

  Actually, we should restrict the length of our palindrome for now, i. e. assign $d_{odd}[i] = r - i$, to handle such situations correctly. After this we'll run the trivial algorithm which will try to increase $d_{odd}[i]$ while it's possible.

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



  It is shown in the illustration that though the palindrome with center j could be larger and go outside the "outer" palindrome, but with i as the center we can use only the part that entirely fits into the "outer" palindrome. But the answer for the position i ($d_{odd}[i]$) can be much bigger 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".

Again, we should not forget to update the values (l, r) after calculating each $d_{odd}[i]$.

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.

However, a more careful analysis shows that the algorithm is linear. In fact, Z-function building algorithm, which looks similar to this algorithm, also works in linear time.

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

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

Implementation of Manacher's algorithm

For calculating $d_{odd}[]$, we get the following code. Things to note:

- i is the index of the center letter of the current palindrome.
- If i exceeds r, $d_{odd}[i]$ is initialized to 0.
- If i does not exceed r, $d_{odd}[i]$ is either initialized to the $d_{odd}[j]$, where j is the mirror position of i in (l, r), or $d_{odd}[i]$ is restricted to the size of the "outer" palindrome.
- The while loop denotes the trivial algorithm. We launch it irrespective of the value of k.
- If the size of palindrome centered at i is x, then $d_{odd}[i]$ stores $\frac{x+1}{2}$.

```
vector<int> manacher_odd(string s) {
    int n = s.size();
    s = "$" + s + "^";
    vector<int> p(n + 2);
    int l = 0, r = 1;
    for(int i = 1; i <= n; i++) {
        p[i] = max(0, min(r - i, p[l + (r - i)]));
        while(s[i - p[i]] == s[i + p[i]]) {
            p[i]++;
        }
        if(i + p[i] > r) {
            l = i - p[i], r = i + p[i];
        }
    }
    return vector<int>(begin(p) + 1, end(p) - 1);
}
```

Working with parities

Although it is possible to implement Manacher's algorithm for odd and even lengths separately, the implementation of the version for even lengths is often deemed more difficult, as it is less natural and easily leads to off-by-one errors.

To mitigate this, it is possible to reduce the whole problem to the case when we only deal with the palindromes of odd length. To do this, we can put an additional `#` character between each letter in the string and also in the beginning and the end of the string:

$$abcbcba \to \#a\#b\#c\#b\#c\#b\#a\#,$$

$$d = [1, 2, 1, 2, 1, 4, 1, 8, 1, 4, 1, 2, 1, 2, 1].$$

As you can see, $d[2i] = 2d_{even}[i] + 1$ and $d[2i + 1] = 2d_{odd}[i]$ where d denotes the Manacher array for odd-length palindromes in `#`-joined string, while d_{odd} and d_{even} correspond to the arrays defined above in the initial string.

Indeed, `#` characters do not affect the odd-length palindromes, which are still centered in the initial string's characters, but now even-length palindromes of the initial string are odd-length palindromes of the new string centered in `#` characters.

Note that $d[2i]$ and $d[2i + 1]$ are essentially the increased by 1 lengths of the largest odd- and even-length palindromes centered in i correspondingly.

The reduction is implemented in the following way:

```
vector<int> manacher(string s) {
    string t;
    for(auto c: s) {
        t += string("#") + c;
    }
    auto res = manacher_odd(t + "#");
    return vector<int>(begin(res) + 1, end(res) - 1);
}
```

For simplicity, splitting the array into d_{odd} and d_{even} as well as their explicit calculation is omitted.

Problems

- Library Checker - Enumerate Palindromes
- Longest Palindrome
- UVA 11475 - Extend to Palindrome
- GYM - (Q) QueryreuQ
- CF - Prefix-Suffix Palindrome
- SPOJ - Number of Palindromes
- Kattis - Palindromes