Palindromic Substrings in O(n) with Manacher's Algorithm

Last modified: October 21, 2020

by Said Sryheni
(https://www.baeldung.com/cs/author/saeedsryhini)

**Searching
(https://www.baeldung.com/cs/category/algorithms/searching)**

1. Overview

When dealing with string problems, we'll come across a term called palindromes (/java-palindrome).

In this tutorial, we'll show what a palindrome is. Also, we'll explain Mahacher's (/java-palindrome-substrings) algorithm, which handles palindrome substrings inside a string s.

Finally, we'll give some applications that use Manacher's algorithm.

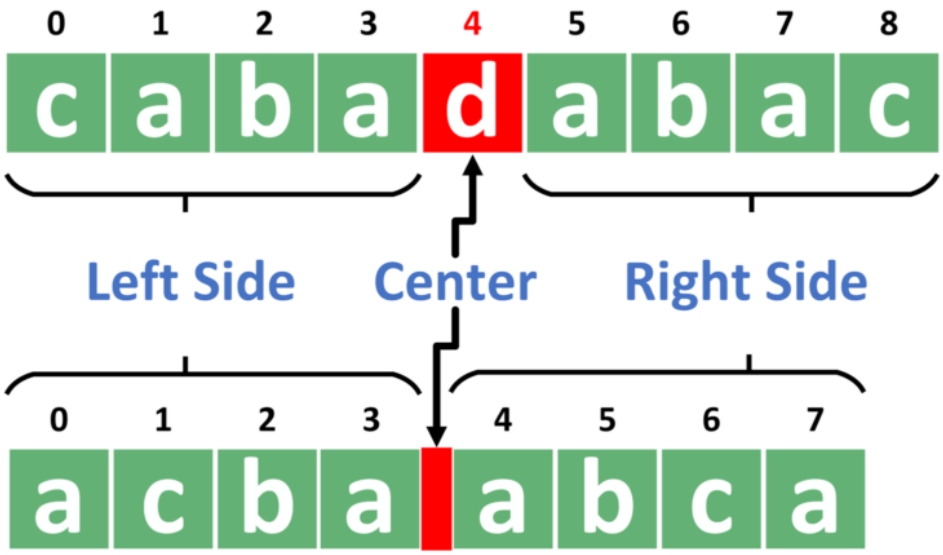
2. Definitions

2.1. Definition of Palindromes

First of all, we need to define what a palindrome is. **A palindrome string is the same when reading forward and backward**. For example, strings $\{aba, aa, a, abcacba\}$ are palindromes, while $\{ab, cbabd\}$ are not.

From this definition, we can define the center of a palindrome. The center of a palindrome is the middle of the string that divides it into halves. **Each of these halves is a mirror image of the other**.

Let's take the following example:



As we can see, **in the case of odd-length palindromes, the center is an element from the palindrome itself**. The part before the center is called the left side of the palindrome. Similarly, the part after the center is called the right side.

However, **in the case of even-length palindromes, we don't have a center element**. Instead, we just have a left and right side of the palindrome.

In both cases, the right side of the palindrome is a mirror image of the left side. In other words, the left and right sides are the reverses of each other.

One more note to consider is that if a string s is a palindrome, then if we remove the first and last characters from s, we'll get a palindrome string as well.

Similarly, if a string s is a palindrome, then adding the same character to the beginning and the end of s will result in a palindrome string as well.

These properties will prove useful once we have a look at the algorithms later.

2.2. Defining the Problem

Now that we have a better understanding of palindromes, we can explain the problem that Manacher's algorithm solves.

In the beginning, the problem gives us a string s. After that, **the problem asks us to calculate two arrays $oddP$ and $evenP$. Both arrays should store an indication of the longest palindrome substrings (LPS) that are centered at each index i.**

As a result, $s[i - oddP[i], i + oddP[i]]$ will be the longest odd-length palindrome whose center is at index i. Likewise, $s[i - evenP[i] + 1, i + evenP[i]]$ will be the longest even-length palindrome, such that the last character on its left side is at index i.

To begin with, we'll explain the naive approach to solve this problem. After that, we'll explain the optimization that Manacher's algorithm makes over the naive approach.

3. Naive Approach

The naive approach is straightforward. From each index inside the string, it tries to expand both sides as long as possible.

Let's take a look at the naive approach implementation:

Algorithm 1: Naive Approach

Data: s: The string to process
n: The length of string s
Result: Returns the oddP and evenP arrays

```
for i ← 0 to n − 1 do
    oddP[i] ← 0;
    L ← i − oddP[i] − 1;
    R ← i + oddP[i] + 1;
    while L ≥ 0 AND R < n AND s[L] = s[R] do
        oddP[i] ← oddP[i] + 1;
        L ← L − 1;
        R ← R + 1;
    end
    evenP[i] ← 0;
    L ← i − evenP[i];
    R ← i + evenP[i] + 1;
    while L ≥ 0 AND R < n AND s[L] = s[R] do
        evenP[i] ← evenP[i] + 1;
        L ← L − 1;
        R ← R + 1;
    end
end
return oddP, evenP;
```

We iterate over all possible indices i inside the string s. For each index, we set $oddP[i]$ to zero. Also, we set L and R to the indexes of the next range to check.

Next, we check whether the current range forms a palindrome or not. The idea is that if the range $[L, R]$ forms a palindrome, and we moved to the next range, which is $[L-1, R+1]$; we only need to check the characters at index $L-1$ and $R+1$.

If the current range forms a palindrome, then we increase $oddP[i]$ by one. Also, we extend our range by one to the left and right sides.

After that, we perform a similar operation to calculate $evenP$ array. The difference is that we pay attention to the initial values of L and R. The reason is that in even-length palindromes, i is the index of the last character on the left side of the palindrome substring. The rest is similar to calculating the $oddP$ array.

In the end, we return the calculated $oddP$ and $evenP$ arrays.

The complexity of the naive approach is $O(n^2)$, where n is the length of the string s.

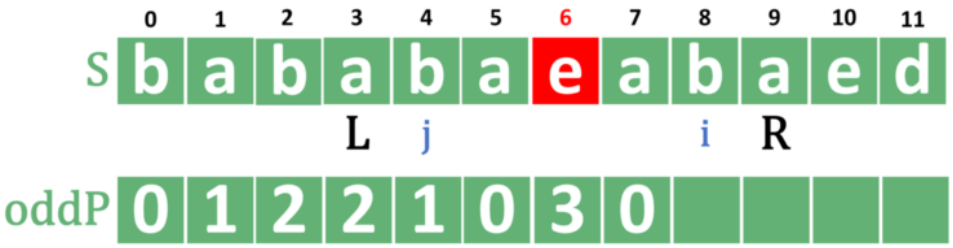
4. Manacher's Algorithm

Manacher's algorithm calculates the $oddP$ and $evenP$ arrays in a similar way to the naive approach. However, if possible, it tries to use the already calculated values, rather than checking all the possible ranges from scratch.

Let's explain the general idea first. After that, we can discuss implementation and complexity.

4.1. General Idea

Let's take the following example to explain the idea of odd-length palindromes better:



Suppose we've calculated all values of $oddP$ from index 0 to 7, and we need to calculate $oddP[8]$.

Before calculating $oddP[8]$, we'll keep the palindrome substring that ends as far to the right as possible so far. The reason for choosing the right-most range will be explained in section 4.4. In our example, that is the substring $s[3, 9]$. Hence, $L = 3$ and $R = 9$.

Now, to calculate $oddP[8]$, we can follow the naive approach. However, **we can use the already calculated values to reduce the number of comparisons**.

Since we know that the substring $s[L, R]$ is a palindrome, we can calculate the mirror image of index i based on the center of the range $[L, R]$. In this example, the center is index 6. Therefore, the mirror of $i = 8$ is $j = 4$.

Since the left side is a mirror image of the right side, we might say that $oddP[i] = oddP[j]$.

However, there are two cases to consider. The first case is similar to the given example. **We can extend $oddP[i]$ over the value of $oddP[j] = 1$.** Therefore, we can start extending $oddP[i]$ from 2 and moving forward, without checking the value of $oddP[i] = 1$. In the example, we'll get $oddP[8] = 2$.

The second case is given in the second example:

In this case, **we can't say that $oddP[i] = oddP[j] = 4$** because $oddP[j]$ corresponds to the range $[0, 8]$, which is outside the range with $L = 3$ and $R = 9$.

Therefore, **we can only be sure of the value of $oddP[i] = R - i = 1$ because it's the maximum range contained in the range** $[3, 9]$. After that, we can try to extend the range following the naive approach. In this example, the range can't be extended anymore.

4.2. Algorithm for Odd-Length Palindromes

Now that we understand the general idea, we can check the implementation. Take a look at the implementation of Manacher's algorithm for odd-length palindromes:

Algorithm 2: Manacher's Algorithm for Odd-Length Palindromes

```
Data: s: The string to process
      n: The length of string s
Result: Returns the oddP array
L ← 0;
R ← −1;
for i ← 0 to n − 1 do
    oddP[i] ← 0;
    if i ≤ R then
        oddP[i] ← min(oddP[L + R − i], R − i);
    end
    left ← i − oddP[i] − 1;
    right ← i + oddP[i] + 1;
    while left ≥ 0 AND right < n AND s[left] = s[right] do
        oddP[i] ← oddP[i] + 1;
        left ← left − 1;
        right ← right + 1;
    end
    if i + oddP[i] > R then
        L ← i − oddP[i];
        R ← i + oddP[i];
    end
end
return oddP;
```

In the beginning, we initialize L with zero and R with -1, indicating an empty range. Then, we iterate over the string s from left to right.

For each index i, we initialize $oddP[i]$ to the default value of the naive approach, which is zero. Now, we try to use already calculated values.

If i is inside the range $[L, R]$, we update the value of $oddP[i]$ to become the value of its mirror index. However, we pay attention to the case when the mirror index result becomes outside the range $[L, R]$.

Therefore, we take the minimum between $oddP$ for the mirror index, and the maximum length we can take from the current palindrome range $[L, R]$.

Since $L + R - i$ is the mirror index of i, then it's enough to check if $R - i$ is smaller than $oddP[L + R - i]$. The reason is that $R - i$ equals $j - L$, where j is the mirror index of i. Therefore, it's enough to check whether any of them is smaller than $oddP$ for the mirror index.

Once we assigned the initial value to $oddP[i]$, we simply follow the naive approach and try to expand the range centered at index i as far as possible. But, **in this case, we**

start from $oddP[i] + 1$.

After that, we check whether we need to update the values of L and R. Since we need to store the palindrome range that goes as far to the right as possible, we check whether the current range is further to the right than the stored range. If so, we update the value of L and R.

Finally, we return the calculated array of $oddP$.

4.3. Algorithm for Even-Length Palindromes

The algorithm for even-length palindromes has small modifications over the one for odd-length palindromes. Take a look at its implementation:

Algorithm 3: Manacher's Algorithm for Even-Length Palindromes

```
Data: s: The string to process
      n: The length of string s
Result: Returns the evenP array
L ← 0;
R ← −1;
for i ← 0 to n − 1 do
    evenP[i] ← 0;
    if i ≤ R then
        evenP[i] ← min(evenP[L + R − i], R − i);
    end
    left ← i − evenP[i];
    right ← i + evenP[i] + 1;
    while left ≥ 0 AND right < n AND s[left] = s[right] do
        evenP[i] ← evenP[i] + 1;
        left ← left − 1;
        right ← right + 1;
    end
    if i + evenP[i] > R then
        L ← i − evenP[i] + 1;
        R ← i + evenP[i];
    end
end
return evenP;
```

This algorithm is similar to algorithm 2. However, we change the initial values of $left$ and $right$ to fit with the next range to check. The reason is that even-length palindromes have two center indexes. In our implementation, i is the left index between the two center indexes.

After that, we try to extend the palindrome range as far as possible. **We pay attention to start the comparison from the value of $evenP[i]$, rather than starting it always from zero**.

Once we're finished, we update the currently stored right-most range.

Finally, we return the calculated array of $evenP$.

4.4. Proof of Concept

First of all, let's prove the optimality of always keeping in hand the right-most palindrome substring. Suppose we were to keep a range whose right side is smaller. However, it's a larger range, and its beginning is further to the left than the stored range.

In this case, we'll minimize the value of $R - i$ that we consider when calculating either $oddP[i]$ or $evenP[i]$. Hence, we might end up with a smaller initial value for $oddP[i]$ or $evenP[i]$. As a result, we'll end up performing more comparisons than we're supposed to.

From that, we can conclude that keeping the right-most range is always optimal.

Now, let's discuss the complexity.

4.5. Complexity

At first glance, we might think that complexity is similar to the naive approach. However, a closer inspection will show us that, every time, we start $oddP[i]$ either from $oddP[L + R - i]$ or from $R - i$.

If $oddP[L + R - i]$ is smaller than $R - i$, then the range won't be extended any more. The reason is that if the range were to extend, the value $oddP[L + R - i]$ would've been larger because both sides of the range $[L, R]$ are mirrors of each other.

Therefore, in this case, we won't make any successful comparisons.

Otherwise, if $R - i$ is smaller than $oddP[L + R - i]$, then we might make successful comparisons. However, in this case, we'll be exploring a palindrome range that is farther to the right than the current range. Hence, every successful comparison will result in a later forward movement of R to the right.

As a result, each successful comparison operation also results in a one-step movement for R forward. Also, we can notice that R never gets reduced.

Therefore, the inner while loop gets executed at most n times. Hence, **the complexity of Manacher's algorithm is** $O(n)$, where n is the number of characters inside the string s.

5. Applications

Let's examine a few applications where Manacher's algorithm can be used.

5.1. Longest Palindrome Substring

Suppose the problem gives us a string s and asks us to calculate the longest palindrome substring inside the string s.

This is a straightforward application for Manacher's algorithm. We simply calculate both arrays $oddP$ and $evenP$ as shown in algorithms 2 and 3.

Since these arrays store the longest palindrome substring for each index, **we just need to check the values** $oddP[i] \times 2 + 1$ **and** $evenP[i] \times 2$. From all these values, the answer is the maximum possible among them.

The time complexity of the described approach is $O(n)$, where n is the length of string s.

5.2. Number of Palindrome Substrings

Another application is to calculate the number of palindrome substrings inside a given string s.

Similarly, we calculate both arrays $oddP$ and $evenP$ as shown in algorithms 2 and 3. After that, we iterate over all possible values of these two arrays.

The array $oddP$ stores the length of each side of the longest palindrome substring. By removing the first and last character of each of these substrings, we get a new palindrome.

Hence, we should calculate:

$$sumOdd = \sum_{i=0}^{n-1} oddP[i] + 1$$

Note that, we added one for each index i indicating that every single character is a palindrome of its own.

In addition, we need to calculate:

$$sumEven = \sum_{i=0}^{n-1} evenP[i]$$

Finally, we should return the value of $sumOdd + sumEven$.

The described solution can be implemented in $O(n)$ **time complexity**, where n is the length of string s.

5.3. Number of Different Palindrome Substrings

This problem is similar to the previous one. However, instead of calculating the number of palindrome substrings in general, we're asked to calculate the number of different palindrome substrings. Therefore, if two or more equal substrings are palindromes, the answer needs to be incremented only by one.

This problem is a little tricky and needs a good understanding of Manacher's algorithm.

First of all, **when we take the LPS value of the mirror of index** i**, this value corresponds to the exact same substring as the one at index** i. We don't need to calculate

this value because it's already been calculated when we were at the mirror index.

However, **each extending operation might result in a new different substring**. Note that, the expansion operation is done at most n times.

We can use a hash function, similar to the one used in the Rabin-Karp (/cs/rabin-karp-algorithm) algorithm. The hashing function can give us the hash value of a certain substring in constant time complexity. Therefore, we can calculate the hash value of all the substrings resulting from expansion operations in linear time.

Finally, the answer is the number of the resulting hash values without repetition. To do this, we can insert all the hash values into a hash-set (/java-hashset). Then, the answer will be the size of the hash-set because it adds the same value only once.

This approach can be efficiently implemented in $O(n)$ time complexity, where n is the length of the string s.

6. Conclusion

In this tutorial, we explained the term palindrome and discussed palindrome substrings inside a given string.

First of all, we explained the naive approach.

After that, we explained Manacher's algorithm with both its versions that get the odd-length and even-length palindromes.

Finally, we presented some problems that can be solved using Manacher's algorithm in linear time complexity.

Be the First to Comment!

0 COMMENTS

CATEGORIES

CORE CONCEPTS (/CS/CATEGORY/CORE-CONCEPTS)
ALGORITHMS (/CS/CATEGORY/ALGORITHMS)
ARTIFICIAL INTELLIGENCE (/CS/CATEGORY/AI)
GRAPH THEORY (/CS/CATEGORY/GRAPH-THEORY)
SECURITY (/CS/CATEGORY/SECURITY)
LATEX (/CS/CATEGORY/LATEX)

SERIES

ABOUT

ABOUT BAELDUNG (HTTPS://WWW.BAELDUNG.COM/ABOUT)
THE FULL ARCHIVE (/CS/FULL_ARCHIVE)
EDITORS (HTTPS://WWW.BAELDUNG.COM/EDITORS)

TERMS OF SERVICE (HTTPS://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
PRIVACY POLICY (HTTPS://WWW.BAELDUNG.COM/PRIVACY-POLICY)
COMPANY INFO (HTTPS://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)