Shortest Palindrome

Given a string S, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome you can find by performing this transformation.

For example:

Given `"aacecaaa"`, return `"aaacecaaa"`.

Given `"abcd"`, return `"dcbabcd"`.

Credits:
Special thanks to @ifanchu for adding this problem and creating all test cases.
Thanks to @Freezen for additional test cases.

Solution 1

We can construct the following string and run KMP algorithm on it: (s) + (some symbol not present in s) + (reversed string)

After running KMP on that string as result we get a vector**p** with values of a prefix function for each character (for definition of a prefix function see KMP algorithm description). We are only interested in the last value because it shows us the largest suffix of the reversed string that matches the prefix of the original string. So basically all we left to do is to add the first k characters of the reversed string to the original string, where k is a difference between original string size and the prefix function for the last character of a constructed string.

```
class Solution {
public:
    string shortestPalindrome(string s) {
        string rev_s = s;
        reverse(rev_s.begin(), rev_s.end());
        string l = s + "#" + rev_s;

        vector<int> p(l.size(), 0);
        for (int i = 1; i < l.size(); i++) {
            int j = p[i - 1];
            while (j > 0 && l[i] != l[j])
                j = p[j - 1];
            p[i] = (j += l[i] == l[j]);
        }

        return rev_s.substr(0, s.size() - p[l.size() - 1]) + s;
    }
};
```

written by Sammax original link here

Solution 2

The idea is to use two anchors `j` and `i` to compare the String from beginning and end. If `j` can reach the end, the String itself is Palindrome. Otherwise, we divide the String by `j`, and get `mid = s.substring(0, j)` and `suffix`.

We reverse `suffix` as beginning of result and recursively call `shortestPalindrome` to get result of `mid` then appedn `suffix` to get result.

```
    int j = 0;
    for (int i = s.length() - 1; i >= 0; i--) {
        if (s.charAt(i) == s.charAt(j)) { j += 1; }
    }
    if (j == s.length()) { return s; }
    String suffix = s.substring(j);
    return new StringBuffer(suffix).reverse().toString() + shortestPalindrome(s.s
ubstring(0, j)) + suffix;
```

written by xcv58 original link here

Solution 3

Firstly, let me share my understanding of KMP algorithm. The key of KMP is to build a look up table that records the match result of prefix and postfix. Value in the table means the max len of matching substring that exists in both prefix and postfix. In the prefix this substring should starts from 0, while in the postfix this substring should ends at current index.

For example, now we have a string "ababc" The KMP table will look like this:

> a b a b c
>
> 0 0 1 2 0

(Note: we will not match substring with itself, so we will skip index 0)

So how does this table help us search string match faster?

Well, the answer is if we are trying to match a char after postfix with target string and failed, then we can smartly shift the string, so that the matching string in prefix will replace postfix and now we can try to match the char after prefix with this char in target.

Take above string as an example.

Now we try to match string "ababc" with "abababc".

We will initially have match as below

> a b a b a b c (string x)
>
> a b a b c (string y)
>
> 0 1 2 3 4 5 6

We found char at index 4 does not match, then we can use lookup table and shift the string y wisely. We found table[3] = 2, which means we can shift the string y rightward by 2, and still have same but shorter prefix before index 4, like this:

> a b a b a b c (string x)
>
> ___a b a b c (string y)
>
> 0 1 2 3 4 5 6

If there is a long gap between prefix and postfix, this shift can help us save a lot of time. In the brute force way, we cannot do that because we have no information of the string. We have to compare each possible pair of chars. While in kmp, we know the information of string y so we can move smartly. We can directly jump to the next possible matching pair while discard useless pair of chars.

We are almost done with KMP, but we still have one special case that needs to be

taken care of.

Say now we have a input like this:

> a a b a a a (input String)
>
> 0 1 2 3 4 5 (index)
>
> 0 1 0 1 2 ? (KMP table)

How should we build the KMP table for this string?

Say the pointer in prefix is "x", which is at index 2 now and the pointer in postfix is "y" which is at index 5 now. we need to match "b" pointed by x with "a" pointed by y. It is an unmatched pair, how should we update the cell?

Well, we really don't need to reset it to 0, that will make us skip a valid shorter matching substring "aa". What we do now is just to shorten the length of substring by 1 unit and try to match a shorter substring "aa". This can be done by moving pointer x to the index recorded in [indexOf(x)-1] while keep pointer y stay still. This is because by following the value in KMP table we can always make sure previous part of prefix and postfix is matched even we have shorten their length, so we only need to care about the char after matched part in prefix and postfix.

Use above example:

Firstly we try to compare prefix "aab" with postfix "aaa", pointer in prefix now points to "b" while pointer in postfix now points to "a". So this means current len of postfix/prefix will not give a match, we need to shorten it.

So in the second step, we will fix pointer in postfix, and move pointer in prefix so that we can compare shorter prefix and postfix. The movement of pointer in prefix (say at index x) is done by using KMP table. We will set pointer in prefix to be table [indexOf(x)-1]. In this case, we will move prefix pointer to index 1. So now we try to compare prefix "aa" with postfix "aa".

Finally, we found the matching prefix and postfix, we just update the cell accordingly.

Above is my understanding of KMP algorithm, so how could we apply KMP to this problem

========== I am just a splitter ==========

This problem asks us to add string before the input so the result string will be a palindrome. We can convert it to an alternative problem"find the longest palindrome substring starts from index 0". If we can get the length of such substring, then we can easily build a palindrome string by inserting the reverse part of substring after such substring before the original string.

Example:

input string:

> abacd

longest palindrome substring starts from 0:

> aba

Insert the reverse part of substring after palindrome substring before the head:

> dcabacd

Now the problem becomes how to find the longest palindrome substring starts from 0. We can solve it by using a trick + KMP.

The trick is to build a temp string like this:

> s + "#" + reverse(s)

Then we run KMP on it, the value in last cell will be our solution. In this problem, we don't need to use KMP to match strings but instead we use the lookup table in KMP to find the palindrome.

We add "#" here to force the match in reverse(s) starts from its first index What we do in KMP here is trying to find a match between prefix in s and a postfix in reverse(s). The match part will be palindrome substring.

Example: input:

> catacb

Temp String:

> catacb # bcatac

KMP table:

> c a t a c b # b c a t a c
>
> 0 0 0 0 1 0 0 0 1 2 3 4 5

In the last cell, we got a value 5. It means in s we have a substring of length 5 that is palindrome.

So, above is my understanding of KMP any solution towards this problem. Below is my code

```
public String shortestPalindrome(String s) {
    String temp = s + "#" + new StringBuilder(s).reverse().toString();
    int[] table = getTable(temp);

    //get the maximum palin part in s starts from 0
    return new StringBuilder(s.substring(table[table.length - 1])).reverse().toSt
ring() + s;
}

public int[] getTable(String s){
    //get lookup table
    int[] table = new int[s.length()];

    //pointer that points to matched char in prefix part

    int index = 0;
    //skip index 0, we will not match a string with itself
    for(int i = 1; i < s.length(); i++){
        if(s.charAt(index) == s.charAt(i)){
            //we can extend match in prefix and postfix
            table[i] = table[i-1] + 1;
            index ++;
        }else{
            //match failed, we try to match a shorter substring

            //by assigning index to table[i-1], we will shorten the match string
length, and jump to the
            //prefix part that we used to match postfix ended at i - 1
            index = table[i-1];

            while(index > 0 && s.charAt(index) != s.charAt(i)){
                //we will try to shorten the match string length until we revert
to the beginning of match (index 1)
                index = table[index-1];
            }

            //when we are here may either found a match char or we reach the boun
dary and still no luck
            //so we need check char match
            if(s.charAt(index) == s.charAt(i)){
                //if match, then extend one char
                index ++ ;
            }

            table[i] = index;
        }

    }

    return table;
}
```

If I messed up or misunderstood something, please leave comment below. Thanks ~ written by hpplayer original link here

From Leetcoder.