Previous Next

214. Shortest Palindrome

June 19, 2017 | 64.2K views

Average Rating: 3.80 (54 votes)

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.

Example 1:

```
Input: "aacecaaa"
Output: "aaacecaaa"
```

Example 2:

```
Input: "abcd"
Output: "dcbabcd"
```

Solution

Approach #1 Brute force [Accepted]

Intuition

According to the question, we are allowed to insert the characters only at the beginning of the string. Hence, we can find the largest segment from the beginning that is a palindrome, and we can then easily reverse the remaining segment and append to the beginning. This must be the required answer as no shorter palindrome could be found than this by just appending at the beginning.

For example: Take the string $"\text{abcbabcab}"$. Here, the largest palindrome segment from beginning is $"\text{abcba}"$, and the remaining segment is $"\text{bcab}"$. Hence the required string is reverse of $"\text{bcab}"$(= $"\text{bacb}"$) + original string(= $"\text{abcbabcab}"$) = $"\text{bacbabcbabcab}"$.

Algorithm

- Create the reverse of the original string s, say rev. This is used for comparison to find the largest palindrome segment from the front.
- Iterate over the variable i from 0 to the $\text{size(s)} - 1$:
    - If $s[0 : n-i] == rev[i :]$ (i.e. substring of s from 0 to $n-i$ is equal to the substring of rev from i to the end of string). This essentially means that that substring from 0 to $n-i$ is a palindrome, as rev is the reverse of s.
    - Since, we find the larger palindromes first, we can return reverse of largest palindrome + s as soon as we get it.

C++ Copy

```cpp
string shortestPalindrome(string s)
{
    int n = s.size();
    string rev(s);
    reverse(rev.begin(), rev.end());
    int j = 0;
    for (int i = 0; i < n; i++) {
        if (s.substr(0, n - i) == rev.substr(i))
            return rev.substr(0, i) + s;
    }
    return "";
}
```

Complexity Analysis

- Time complexity: $O(n^2)$.
    - We iterate over the entire length of string s.
    - In each iteration, we compare the substrings which is linear in size of substrings to be compared.
    - Hence, the total time complexity is $O(n*n) = O(n^2)$.
- Space complexity: $O(n)$ extra space for the reverse string rev.

Approach #2 Two pointers and recursion [Accepted]

Intuition

In Approach #1, we found the largest palindrome substring from the string using substring matching which is $O(n)$ in length of substring. We could make the process more efficient if we could reduce the size of string to search for the substring without checking the complete substring each time.

Lets take a string $"\text{abcbabcaba}"$. Let us consider 2 pointers i and j. Initialize $i = 0$. Iterate over j from $n-1$ to 0, incrementing i each time s[i]==s[j]. Now, we just need to search in range $[0, i)$. This way, we have reduced the size of string to search for the largest palindrome substring from the beginning. The range $[0,\text{i})$ must always contain the largest palindrome substring. The proof of correction is that: Say the string was a perfect palindrome, i would be incremented n times. Had there been other characters at the end, i would still be incremented by the size of the palindrome. Hence, even though there is a chance that the range $[0,\text{i})$ is not always tight, it is ensured that it will always contain the longest palindrome from the beginning.

The best case for the algorithm is when the entire string is palindrome and the worst case is string like $"\text{aababababababa}"$, wherein i first becomes 12(check by doing on paper), and we need to recheck in [0,12) corresponding to string $"\text{aabababababa}"$. Again continuing in the same way, we get $i = 10$. In such a case, the string is reduced only by as few as 2 elements at each step. Hence, the number of steps in such cases is linear($n/2$).

This reduction of length could be easily done with the help of a recursive routine, as shown in the algorithm section.

Algorithm

The routine $\text{shortestPalindrome}$ is recursive and takes string s as parameter:

- Initialize $i = 0$
- Iterate over j from $n-1$ to 0:
    - If s[i]==s[j], increase i by 1
- If i equals the size of s, the entire string is palindrome, and hence return the entire string s.
- Else:
    - Return reverse of remaining substring after i to the end of string + $\text{shortestPalindrome}$ routine on substring from start to index $i-1$ + remaining substring after i to the end of string.

C++ Copy

```cpp
string shortestPalindrome(string s)
{
    int n = s.size();
    int i = 0;
    for (int j = n - 1; j >= 0; j--) {
        if (s[i] == s[j])
            i++;
    }
    if (i == n)
        return s;
    string remain_rev = s.substr(i, n);
    reverse(remain_rev.begin(), remain_rev.end());
    return remain_rev + shortestPalindrome(s.substr(0, i)) + s.substr(i);
}
```

Complexity analysis

- Time complexity: $O(n^2)$.
    - Each iteration of $\text{shortestPalindrome}$ is linear in size of substring and the maximum number of recursive calls can be $n/2$ times as shown in the Intuition section.
    - Let the time complexity of the algorithm be T(n). Since, at the each step for the worst case, the string can be divide into 2 parts and we require only one part for further computation. Hence, the time complexity for the worst case can be represented as : $T(n) = T(n-2) + O(n)$. So, $T(n) = O(n) + O(n-2) + O(n-4) + ... + O(1)$ which is $O(n^2)$.

Thanks @CONOVER for the time complexity analysis.

- Space complexity: $O(n)$ extra space for remain_rev string.

Approach #3 KMP [Accepted]

Intuition

We have seen that the question boils down to finding the largest palindrome substring from the beginning.

The people familiar with KMP(Knuth–Morris–Pratt) algorithm may wonder that the task at hand can be easily be compared with the concept of the lookup table in KMP.

KMP Overview:

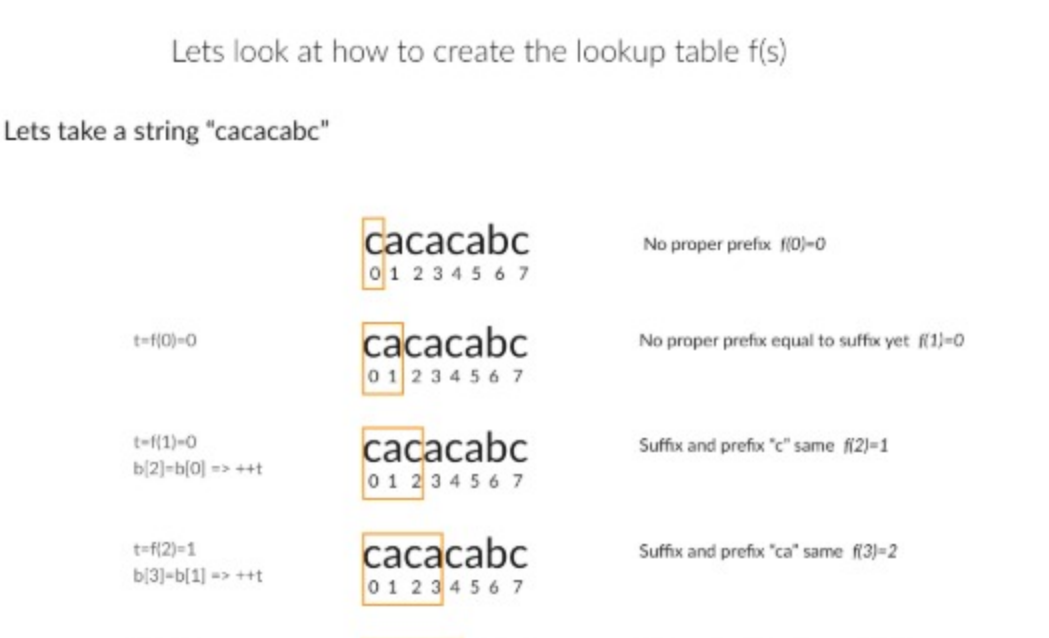
KMP is a string matching algorithm that runs in $O(n+m)$ times, where n and m are sizes of the text and string to be searched respectively. The key component of KMP is the failure function lookup table,say $f(s)$. The purpose of the lookup table is to store the length of the proper prefix of the string $b_1b_2...b_s$ that is also a suffix of $b_1b_2...b_s$. This table is important because if we are trying to match a text string for $b_1b_2...b_n$, and we have matched the first s positions, but when we fail, then the value of lookup table for s is the longest prefix of $b_1b_2...b_n$ that could possibly match the text string upto the point we are at. Thus, we don't need to start all over again, and can resume searching from the matching prefix.

The algorithm to generate the lookup table is easy and inutitive, as given below:

```
f(0) = 0
for(i = 1; i < n; i++)
{
    t = f(i-1)
    while(t > 0 && b[i] != b[t])
        t = f(t-1)
    if(b[i] == b[t]){
        ++t
    f(i) = t
}
```

- Here, we first set f(0)=0 since, no proper prefix is available.
- Next, iterate over i from 1 to $n-1$:
    - Set $t = f(i-1)$
    - While t>0 and char at i doesn't match the char at t position, set $t = f(t)$, which essentially means that we have problem matching and must consider a shorter prefix, which will be $b_{f(t-1)}$, until we find a match or t becomes 0.
    - If $b_i == b_t$, add 1 to t
    - Set $f(i) = t$

The lookup table generation is as illustrated below:



So, the final table:

i	0	1	2	3	4	5	6	7
f(i)	0	0	1	2	3	4	0	1

Wait! I get it!!

In Approach #1, we reserved the original string s and stored it as rev. We iterate over i from 0 to $n-1$ and check for $s[0 : n-i] == rev[i :]$. Pondering over this statement, had the rev been concatenated to s, this statement is just finding the longest prefix that is equal to the suffix. Voila!

Algorithm

- We use the KMP lookup table generation
- Create new_s as $s + "\#" + \text{reverse(s)}$ and use the string in the lookup-generation algorithm
    - The "#" in the middle is required, since without the #, the 2 strings could mix with each ther, producing wrong answer. For example, take the string $"\text{aaaa}"$. Had we not inserted "#" in the middle, the new string would be $"\text{aaaaaaaa}"$ and the largest prefix size would be 7 corresponding to "aaaaaaa" which would be obviously wrong. Hence, a delimiter is required at the middle.
- Return reversed string after the largest palindrome from beginning length(given by $n - \text{f[n\_new-1]}$) + original string s

C++ Copy

```cpp
string shortestPalindrome(string s)
{
    int n = s.size();
    string rev(s);
    reverse(rev.begin(), rev.end());
    string s_new = s + "#" + rev;
    int n_new = s_new.size();
    vector<int> f(n_new, 0);
    for (int i = 1; i < n_new; i++) {
        int t = f[i - 1];
        while (t > 0 && s_new[i] != s_new[t])
            t = f[t - 1];
        if (s_new[i] == s_new[t])
            ++t;
        f[i] = t;
    }
    return rev.substr(0, n - f[n_new - 1]) + s;
}
```

Complexity analysis

- Time complexity: $O(n)$.
    - In every iteration of the inner while loop, t decreases until it reaches 0 or until it matches. After that, it is incremented by one. Therefore, in the worst case, t can only be decreased up to n times and increased up to n times.
    - Hence, the algorithm is linear with maximum $(2*n)*2$ iterations.
- Space complexity: $O(n)$. Additional space for the reverse string and the concatenated string.

Rate this article:

Previous Next

‹ 1 2 3 4 ›