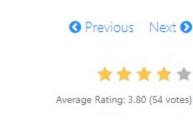
214. Shortest Palindrome

June 19, 2017 | 64.2K views



6 0 0

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"
```

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 "abcbabcab". Here, the largest palindrome segment from beginning is "abcba", and the remaining segment is "bcab". Hence the required string is reverse of "bcab" (= "bacb") + original string(= "abcbabcab") = "bacbabcbabcab". Algorithm

 \bullet 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 size(s) -1:

- 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. \circ Since, we find the larger palindromes first, we can return reverse of largest palindrome + s as
- Copy C++ 1 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++) {

```
9
               return rev.substr(0, i) + s;
 10
 11
        return "";
 12 }
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.

        \circ Hence, the total time complexity is O(n*n) = O(n^2).
  • Space complexity: O(n) extra space for the reverse string rev.
```

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

Approach #2 Two pointers and recursion [Accepted]

Lets take a string "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

Intuition

still be incremented by the size of the palindrome. Hence, even though there is a chance that the range $|0,i\rangle$ 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 "aabababababa", wherein i first becomes 12(check by doing on paper), and we need to recheck in

have reduced the size of string to search for the largest palindrome substring from the beginning. The range [0,i) must always contain the largest palindrome substring. The proof of correction is that: Say the string was

[0,12] corresponding to string "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.

 \circ Return reverse of remaining substring after i to the end of string + shortestPalindrome routine on substring from start to index i-1 + remaining substring after i to the end of string.

Copy Copy

```
9
       if (i == n)
 10
            return s;
 11
       string remain_rev = s.substr(i, n);
 12
       reverse(remain_rev.begin(), remain_rev.end());
 13
        return remain_rev + shortestPalindrome(s.substr(0, i)) + s.substr(i);
 14 }
Complexity analysis

    Time complexity: O(n<sup>2</sup>).

    Each iteration of 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.
        o 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).
```

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).

be compared with the concept of the lookup table in KMP.

KMP Overview:

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

t = f(i-1)while(t > 0 && b[i] != b[t]) t = f(t-1)

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

```
\circ Set f(i) = t
The lookup table generation is as illustrated below:
                                                       KNUTH-MORRIS-PRATT
                                       Lets look at how to create the lookup table f(s)
                         Lets take a string "cacacabc"
                                                        cacacabc
                                                                                No proper prefix f(0)=0
                                                        0 1 2 3 4 5 6 7
                                    t=f(0)=0
                                                        cacacabc
                                                                               No proper prefix equal to suffix yet f(1)=0
```

 \circ 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 $b_{f(t-1)}$,

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! We use the KMP lookup table generation • Create $\underline{\text{new}}_s$ as $s + \#" + \underline{\text{reverse}}(s)$ 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 "aaaa". Had we not inserted "#" in the middle, the new string would be "aaaaaaaaa" and the largest prefix size would be 7 corresponding to "aaaaaaa" which would be obviously wrong. Hence, a delimiter is required at • Return reversed string after the largest palindrome from beginning length(given by $n - f[n_new-1]$) + 1 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];

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.

}

Complexity analysis

Comments: 40 Type comment here... (Markdown is supported)

HarveyW ★ 224 ② October 10, 2018 8:38 PM

jpmorganchase 🛊 26 🗿 September 9, 2018 6:54 AM

38 A V C Share Reply

14 A V C Share Share

AnitAgg * 17 ② June 22, 2019 1:30 AM

having difficulty understand this guy's English. "but...then..."

Preview

SHOW 1 REPLY firejox # 29 @ January 27, 2019 11:32 PM We could use Fast Fourier Transform to solve the problem. 16 A V C Share A Reply SHOW 8 REPLIES

• 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.

- SHOW 1 REPLY town9628 🛊 10 🗿 July 8, 2018 2:07 AM python is really slow, and the solution o(n^2) in python might not be accepted in cases.
- dongseong ★ 78 ② October 24, 2018 11:02 AM O(n) Rabin-Karp based on solution 1. I failed to understand solution 2 & 3. I'm not sure I can do KMP in interview. However I'm sure I can do Rabin-Karp. Folloing code recude substring comparison time from
- 4 A V C Share Share SHOW 2 REPLIES

3 A V C Share Share

MitchellHe # 240 July 15, 2017 12:13 AM

string is reduced by half each recursion?

SHOW 1 REPLY

(1234)

with one difference that when pointer in reserve string reaches end, the pointer in s is the position we Read More 4 A V Share Share Reply amarsuchak *3 October 2, 2019 5:05 AM If you simply use the SST with the KPMG with the AMEX recursive DP algorithm, you can achieve O(n), O(1) easily. It is very intuitive classical problem.

Read More

smart approach 3 using KMP. One possible improvement I feel is that we might not need to build s+"#"+reverse(s). To find longest prefix in s that matches the suffix in its reverse, like you mentioned, we keep moving s to the right and compare it to its reserve. This is very similar to general string match

Solution

Approach #1 Brute force [Accepted] Intuition

 \circ If s[0:n-i]==rev[i:] (i.e. substring of s from 0 to n-i is equal to the substring of rev

soon as we get it.

if (s.substr(0, n - i) == rev.substr(i)) 8

a perfect palindrome, i would be incremented n times. Had there been other characters at the end, i would

• Initialize i=0

1 string shortestPalindrome(string s)

if(s[i] == s[j])

for (int j = n - 1; j >= 0; j--) {

int n = s.size();

int i = 0;

C++

8

Algorithm The routine shortest Palindrome is recursive and takes string s as parameter:

string to search for the substring without checking the complete substring each time.

• Iterate over j from n-1 to 0: o If s[i] == s[j], increase i by 1 ullet If i equals the size of s, the entire string is palindrome, and hence return the entire string s. · Else:

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

f(0) = 0for(i = 1; i < n; i++)

Here, we first set f(0)=0 since, no proper prefix is available.

until we find a match or t becomes 0.

t=f(3)=2

t=f(4)=3

t=f(5)=4

So, the final table:

f(i)

Wait! I get it!!

Algorithm

C++

2

5

8

9 10

11 12

13

14

15

16 17

18 }

the middle.

++t;

f[i] = t;

Time complexity: O(n).

original string 8

b[4]=b[2] => ++t

b[5]=b[3] => ++t

b[6]!=b[4] => t=f(3)=2

b[6]!=b[2] => t = f(1)=0

0

0

b[6]!=b[0] => ++t

 $if(b[i] == b[t]){$

• Next, iterate over i from 1 to n-1:

 \circ If $b_i == b_t$, add 1 to t

 \circ Set t = f(i-1)

++t f(i) = t

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:

```
01234567
t=f(1)=0
                                             Suffix and prefix "c" same f(2)=1
                     cacacabc
b[2]=b[0] => ++t
                     01234567
t=f(2)=1
                                             Suffix and prefix "ca" same f(3)=2
                     cacacabc
b[3]=b[1] => ++t
                     01234567
```

cacacabc

01234567

cacacabc

0 1 2 3 4 5 6 7

cacacabc

01234567

cacacabc

01234567

3

2

2

1

1

0

Suffix and prefix "cac" same f(4)=3

Suffix and prefix "caca" same f(5)=4

No proper prefix equal to suffix f(6)=0

7

1

Copy Copy

Next **①**

Sort By ▼

Post

Suffix and prefix "c" same f(7)=1

6

0

5

4

4

3

In Approach #1, we reserved the original string s and stored it as rev. We iterate over i from 0 to n-1 and if (s_new[i] == s_new[t]) return rev.substr(0, n - f[n_new - 1]) + s;

Rate this article: * * * * *

O Previous

It is mentioned that brute force solution is accepted but it is howing memory limit exceeded when I run 7 A V C Share Share

We could also use Manacher's Algo for palindrome to solve the problem.

4 A V C Share Share O(n) to O(1) in solution 1.

cuiyan1029 * 6 @ July 14, 2019 9:30 AM

Approach 1 has TLE now with the last test case.

5 A V C Share Share

SHOW 1 REPLY

+O(N). By replacement, we have T(N) = O(N) + O(N/2) + O(N/4) + ... + O(1) <= O(2N) = O(N). So I think T(N) = O(N) for Solution #2. Read More 2 A V C Share Share

For Solution #2, why the time cost is O(NlogN) rather than o(N) when the article claims that the size of

Let T(N) indicates the time complexity of the algorithm in regard of a string of length N. T(N) = T(N/2)