FIND THE LONGEST PALINDROMIC SUBSTRING: MANACHER ALGORITHM

LARRY, ZHIRONG LI

1. Transform Step

It is sort of troublesome when writing code to differentiate an odd-length palindrome substring from an even-length one. By inserting some special character, e.g., '#' which does not appear in the original string in between every two characters, the problem can be equivalently converted over to finding an odd-length palindrome substring.

For instance, the original string is "abaaba", we can change the original string to "a#b#a#a#b#a". We know that the longest palindrome substring is the string itself "abaaba" of even length. In the transformed domain, the longest palindrome substring is centered on the 3rd '#' with (a#b#a)#(a#b#a) of odd-length.

Another example, the original string is "abcba", the longest substring in the transformed domain is (a#b#)c(#b#a) of odd-length again.

In conclusion, in the transformed domain, odd-length palindrome's center will land at '#' and even-length palindrome will center around a character in the original string. Moreover, since we doubled the string length, the original length of the palindrome substring is just a single sided substring starting from the center to one end, e.g, (a#b#a)#(a#b#a), the length is calculated from center '#' to 'a' on the right end, which is 6.

2. Auxiliary Arrays

We need to operate on char* ori_str = "abaaba" and we do need to convert this string to its transformed representation form. In order to avoid index from crossing the boundary when doing character comparisons, we need to add another special character '$' at the beginning of the transformed string and the other end is taken care of by '\0'. So the actual string starts at index 1.

```
const int ori_strlen = strlen(ori_str);
const int trans_strlen = 2*ori_strlen;
char* str = (char*)malloc(trans_strlen+1);
if (str==NULL) exit (1);
str[0]='$'; str[1]= ori_str[0];
for (int n=1;n<ori_strlen;n++){ str[2*n]='#'; str[2*n+1] = ori_str[n]; }
str[trans_strlen]='\0';
printf ("Transformed string: %s\n",str);
//do something
free(str);
```

In addition to that, we need to create an auxiliary int array P, in which P[n] indicated if centered around position n, the distance to the right end of the longest palindrome string.

```
int* P = (int*) malloc(trans_strlen*sizeof(int));
if (P==NULL) exit (1);
//do something
free(P);
```

Date: April 21, 2012.

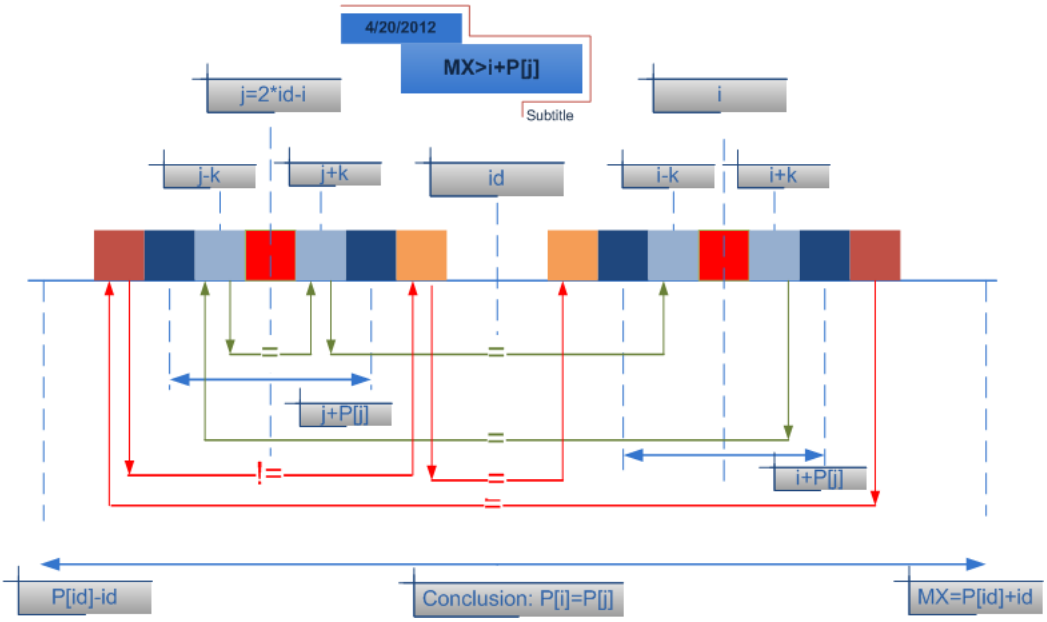

FIGURE 3.1. $mx - i > P[j]$

3. Lower Bound of P[i] when MX>i

Keep in mind that P[id] stores the distance from the center (positioned at id) to the right end of the palindrome, the original palindrome length is just equal to P[id]+1; For example,

```
Index = 0 1 2 3 4 5 6 7 8 9 10 11 12
str[] = $ a # b # a # a # b # a '\0';
P[]   = 0 0 0 2 0 1 5 1 0 2 0 0;
```

This P[] array is calculated from the left end, position 1, to the right end. P[0] is always initialized to be 0.

Like the K-M-P string search algorithm, we need to avoid any unnecessary comparisons based on the information from the previous comparisons.

We need to keep record of id, which is used to indicate for current i, for all j<i, id corresponds to the largest P[j]+j. In another word, id corresponds to the index of the center for the current rightmost boundary of a palindrome substring. I.e., $id \triangleq argmax\{P[j] + j : j < i\}$ and $mx \triangleq P[id] + id$;

There is an elegant observation establishing the lower bound of P[i] when P[id]+id=mx>i, which can be mathematically formulated as

$$P[i] \geq \min(P[2 \times id - i], mx - i)$$

This is based on the following observations.

- $j = 2 \times id - i$ corresponds to the mirror of i relative to id.
- If $mx > i$, this means we could skip some characters because these characters are for sure to be palindromic.

We have the following two cases when $mx > i$

Case I $mx - i > P[j]$: As we can see from the diagram 3.1, indeed, $P[i] = P[j]$.

Case II $mx - i \leq P[j]$: As we can see from the Figure 3.2, we need to check str[mx+1] == str[2i-mx-1] and so on.

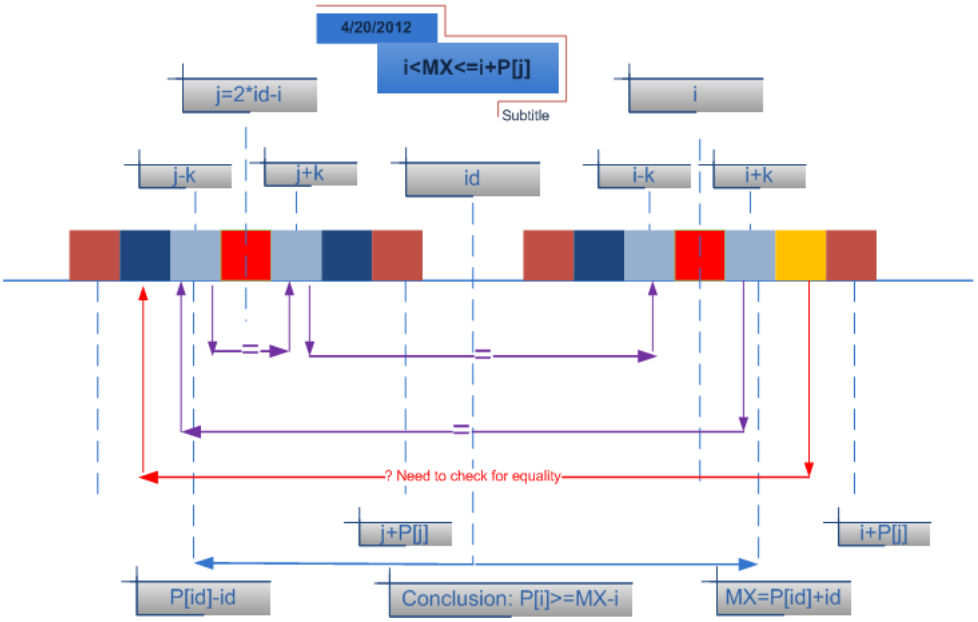

FIGURE 3.2. $mx \leq i + P[j]$

Notice that we can directly set the initial value of P[i] to this lower bound and start checking from str[i+P[i]+1] == str[i-P[i]-1]. Indeed, $P[i] = mx - i$ since the line under check (in red) must take inequality sign. Otherwise, P[id] should really be the distance at least to mx+1 so it is not the right length of the palindrom. By the palindromic property centered at j and id, str[mx+1] != str[2i-mx-1] .

Therefore, this P[] creation process can be finished in one scan.

4. Update on P[]

Update the P[]

```
int id = 0;
int mx = 0;
for(int i=1;i<trans_strlen;i++){
    int j = 2*id-i;
    if (mx>i) P[i] = min(mx-i,P[j]); else P[i]=0;
    while(str[i+P[i]+1] == str[i-P[i]-1]) ++P[i];
    //update id and mx
    if (i+P[i]>mx){ id = i; mx = i+P[i]; }
}
```

5. Find the Longest Palindromic Substring

Find the longest palindromic substring

```
int max_len = 0;
int max_index = 0;
for(int i=1;i<trans_strlen;++i){
```

```
    if (P[i]>max_len){ max_len = P[i]; max_index = i; }
}
char* longest_palindrome = (char*)malloc(max_len+1);
if (longest_palindrome == NULL) exit(1);
//if land at #, the max-len should be reduced by one
if (str[max_index-max_len] == '#') max_len--;
int longest_palindrome_pos = (max_index-max_len)/2;
for(int i=0;i<=max_len;i++) longest_palindrome[i]= ori_str[longest_palindrome_pos++];
longest_palindrome[max_len]='\0';
printf("The longest palindromic substring: %s\n",longest_palindrome);
```

6. Complexity Analysis

We do need one scan to create the P[]. For each i, we do at most N character-by-character comparisons where N is the size of the original string.

If $mx > i$, then if $mx - i > P[j]$, indeed the next comparison must fail (1 comparison performed). Otherwise we do comparisons starting at $mx + 1$ relative to i. If this process is successful, then we do increase the mx value. The number of comparisons corresponds to how much we increase the value of mx. Since this amount of increase can not be larger than N and mx is non-decreasing so the overall number of comparisons for updating mx is upper bounded by N.

If $mx \leq i$ and if $P[i] = 0$, we only do one comparison and increase i by one. If there is a match, which in turn means $P[i] \geq 1$, thus we do update mx again. Therefore, no matter in which case, the total number of comparisons will be bounded from above by $3N$, i.e., the sum of the total number of comparisons assocated with mx update process ($\leq N$) and the total number of one-time comparison for each i where $mx \leq i$ and $P[i] = 0$ ($\leq 2N$).