Program Name-

Answer several queries on the substrings of the given input string that whether the substring is a palindrome or not.

Project Category- Strings, Hashing, Dynamic Programming

Programming Paradigm Used- Cumulative Hash Function, Dynamic Programming, Modulo Operations

Examples-

Suppose our input string is "abaaabaaaba" and the queries- [0, 10], [5, 8], [2, 5], [5, 9]

We have to tell that the substring having the starting and ending indices as above is a palindrome or not.

[0, 10] → Substring is "abaaabaaaba" which is a palindrome.
[5, 8] → Substring is "baaa" which is not a palindrome.
[2, 5] → Substring is "aaab" which is not a palindrome.
[5, 9] → Substring is "baaab" which is a palindrome.

Let us assume that there are Q such queries to be answered and N be the length of our input string. There are the following two ways to answer these queries

O (N.Q) Time and O(1) Auxiliary Space Method-

One by one we go through all the substrings of the queries and check whether the substring under consideration is a palindrome or not.

Since there are Q queries and each query can take O(N) worse case time to answer, this method takes O(Q.N) time in the worst case. Although this is an in-place/space-efficient algorithm, but still there are more efficient method to do this.

O(N+Q) Time and O(N) Auxiliary Space Method-

We use string hashing. What we do is that we calculate cumulative hash values of the string in the original string as well as the reversed string in two arrays- prefix[] and suffix[].

How to calculate the cumulative hash values ?

Suppose our string is str[] , then the cumulative hash function to fill our prefix[] array used is-

prefix[0] = 0
prefix[i] = str[0] + str[1] * 101 + str[2] * 101^2 + ... + str[i-1] * 101^{i-1}

For example , take the string- "abaaabxyaba"

prefix[0] = 0
prefix[1] = 97 (ASCII Value of 'a' is 97)
prefix[2] = 97 + 98 * 101
prefix[3] = 97 + 98 * 101 + 97 * 101^2
.....
.....
prefix[11] = 97 + 98 * 101 + 97 * 101^2 +.....+ 97 * 101^{10}

Now the reason to store in that way is that we can easily find the hash value of any substring in O(1) time using-

hash (L, R) = prefix[R+1] – prefix[L]

For example

hash (1, 5) = hash ("baaab") = prefix[6] – prefix[1] = *98 * 101 + 97 * 101^2 + 97 * 101^3 + 97 * 101^4 + 98 * 101^5 = 1040184646587* [We will use this weird value later to explain what's happening].

Similar to this we will fill our suffix[] array as-

suffix[0] = 0
suffix[i] = str[n-1] + str[n-2] * 101 + str[n-3] * 101^2 + ... + str[n-i] * 101^{i-1}

For example , take the string- "abaaabxyaba"

suffix[0] = 0
suffix[1] = 97 (ASCII Value of 'a' is 97)
suffix[2] = 97 + 98 * 101
suffix[3] = 97 + 98 * 101 + 97 * 101^2

.....
.....
suffix[11] = 97 + 98 * 101 + 97 * 101^2 +.....+ 97 * 101^{10}

Now the reason to store in that way is that we can easily find the reverse hash value of any substring in O(1) time using-

reverse_hash (L, R) = hash (R, L) = suffix[n-L] – suffix[n-R-1]

where n = length of string.

Here n = 11,

So,

reverse_hash(1,5) = reverse_hash ("baaab") = hash ("baaab") [Reversing "baaab" gives "baaab"]

hash ("baaab") = suffix[11-1] – suffix[11-5-1] = suffix[10] – suffix[5] =

*98 * 101^5 + 97 * 101^6 + 97 * 101^7 + 97 * 101^8 + 98 * 101^9 = 108242031437886501387*

Now there doesn't seem to be any relation between these two weird integers – *1040184646587* and *108242031437886501387*

Think again.
Is there any relation between these two massive integers ?

Yes, there is and this observation is the core of this program/article.

*1040184646587 * 101^4 = 108242031437886501387*

Try thinking about this and you will find that any substring starting at index- L and ending at index- R (both inclusive) will be a palindrome if and only if-

(prefix [R + 1] – prefix [L]) / (101^L) = (suffix [n - L] – suffix [n – R- 1]) / (101^{n-R-1})

The rest part is just implementation.

Implementation Details-

The function computerPowers() in the program computes the powers of 101 using dynamic programming.

Overflow Issues-

As, we can see that the hash values and the reverse hash values can become huge for even the small strings of length – 8. Since C and C++ doesn't provide support for such large numbers, so it will cause overflows. To avoid this we will take modulo of a prime (a prime number is chosen for some specific mathematical reasons). We choose the biggest possible prime which fits in an integer value. The best such value is 1000000007. Hence all the operations are done modulo 1000000007.

However Java and Python has no such issues and can be implemented without the modulo operator.

The fundamental modulo operations which are used extensively in the program are listed below.

1) *Addition-*

(a + b) %M = (a %M + b % M) % M
(a + b + c) % M = (a % M + b % M + c % M) % M
(a + b + c + d) % M = (a % M + b % M + c % M+ d% M) % M
…. ….. ….. ……
…. ….. ….. ……

2) Multiplication-

(a * b) % M = (a * b) % M
(a * b * c) % M = ((a * b) % M * c % M) % M
(a * b * c * d) % M = ((((a * b) % M * c) % M) * d) % M
…. ….. ….. ……
…. ….. ….. ……

This property is used by modPow() function which computes power of a number modulo M

3) *Mixture of addition and multiplication-*

(a * x + b * y + c) % M = ((a * x) % M +(b * y) % M+ c % M) % M

4) ***Subtraction-***

(a - b) % M = (a % M – b % M + M) % M [Correct]
(a - b) % M = (a % M – b % M) % M [Wrong]

5) ***Division-***

(a / b) % M = (a * MMI(b)) % M

Where MMI() is a function to calculate *Modulo Multiplicative Inverse* [See web for more information].
In our program this is implemented by the function- findMMI().

Note-

There are two codes in this project-

O (N.Q) Time and O(1) Auxiliary Space Method

O(N+Q) Time and O(N) Auxiliary Space Method