Palindrome Substring Queries

Given a string and several queries on the substrings of the given input string to check whether the substring is a palindrome or not.

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

Method 1 (Naive)

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.

Method 2 (Cumulative Hash)

The idea is similar to Rabin Karp string matching. 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] * 1012 +
                              ...... + str[i-1] * 101i-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 * 1012
...........................
...........................
prefix[11] = 97 + 98 * 101 + 97 * 1012 +
                         ........+ 97 * 10110
```

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 * 1012 + 97 * 1013 + 97 * 1014 + 98 * 1015 = 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] * 10^1 + 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.

For “abaaabxyaba”, n = 11
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 * 1015 + 97 * 1016 + 97 * 1017 + 97 * 1018 + 98 * 1019 = 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

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

The rest part is just implementation.

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. In our program this is implemented by the function- findMMI().

```
/* A C++ program to answer queries to check whether
   the substrings are palindrome or not efficiently */
#include<bits/stdc++.h>
using namespace std;

#define p 101
#define MOD 1000000007

// Structure to represent a query. A query consists
// of (L,R) and we have to answer whether the substring
// from index-L to R is a palindrome or not
struct Query
{
    int L, R;
};

// A function to check if a string str is palindrome
// in the ranfe L to R
bool isPalindrome(string str, int L, int R)
{
    // Keep comparing characters while they are same
    while (R > L)
        if (str[L++] != str[R--])
            return(false);
    return(true);
}

// A Function to find pow (base, exponent) % MOD
// in log (exponent) time
unsigned long long int modPow(unsigned long long int base,
                              unsigned long long int exponent)
{
    if (exponent == 0)
        return 1;
    if (exponent == 1)
        return base;

    unsigned long long int temp = modPow(base, exponent/2);

    if (exponent %2 ==0)
        return (temp%MOD * temp%MOD) % MOD;
    else
        return ((( temp%MOD * temp%MOD)%MOD) * base%MOD) % MOD;
}

// A Function to calculate Modulo Multiplicative Inverse of 'n'
unsigned long long int findMMI(unsigned long long int n)
{
    return modPow(n, MOD-2);
}

// A Function to calculate the prefix hash
void computePrefixHash(string str, int n, unsigned long long
                       int prefix[], unsigned long long int power[])
{
    prefix[0] = 0;
    prefix[1] = str[0];

    for (int i=2; i<=n; i++)
        prefix[i] = (prefix[i-1]%MOD +
                    (str[i-1]%MOD * power[i-1]%MOD)%MOD)%MOD;
    return;
}


// A Function to calculate the suffix hash
// Suffix hash is nothing but the prefix hash of
// the reversed string
void computeSuffixHash(string str, int n,
                       unsigned long long int suffix[],
                       unsigned long long int power[])
{
    suffix[0] = 0;
    suffix[1] = str[n-1];

    for (int i=n-2, j=2; i>=0 && j<=n; i--,j++)
        suffix[j] = (suffix[j-1]%MOD +
                     (str[i]%MOD * power[j-1]%MOD)%MOD)%MOD;
    return;
}

// A Function to answer the Queries
void queryResults(string str, Query q[], int m, int n,
                  unsigned long long int prefix[],
                  unsigned long long int suffix[],
                  unsigned long long int power[])
{
    for (int i=0; i<=m-1; i++)
    {
        int L = q[i].L;
        int R = q[i].R;

        // Hash Value of Substring [L,R]
        unsigned long long hash_LR =
            ((prefix[R+1]-prefix[L]+MOD)%MOD *
             findMMI(power[L])%MOD)%MOD;

        // Reverse Hash Value of Substring [L,R]
        unsigned long long reverse_hash_LR =
            ((suffix[n-L]-suffix[n-R-1]+MOD)%MOD *
             findMMI(power[n-R-1])%MOD)%MOD;

        // If both are equal then the substring is a palindrome
        if (hash_LR == reverse_hash_LR )
        {
            if (isPalindrome(str, L, R) == true)
                printf("The Substring [%d %d] is a "
                       "palindrome\n", L, R);
            else
                printf("The Substring [%d %d] is not a "
                       "palindrome\n", L, R);
        }

        else
            printf("The Substring [%d %d] is not a "
                   "palindrome\n", L, R);
    }

    return;
}

// A Dynamic Programming Based Approach to compute the
// powers of 101
void computePowers(unsigned long long int power[], int n)
{
    // 101^0 = 1
    power[0] = 1;

    for(int i=1; i<=n; i++)
        power[i] = (power[i-1]%MOD * p%MOD)%MOD;

    return;
}
```

```

/* Driver program to test above function */
int main()
{
    string str = "abaaabaaaba";
    int n = str.length();

    // A Table to store the powers of 101
    unsigned long long int power[n+1];

    computePowers(power, n);

    // Arrays to hold prefix and suffix hash values
    unsigned long long int prefix[n+1], suffix[n+1];

    // Compute Prefix Hash and Suffix Hash Arrays
    computePrefixHash(str, n, prefix, power);
    computeSuffixHash(str, n, suffix, power);

    Query q[] = {{0, 10}, {5, 8}, {2, 5}, {5, 9}};
    int m = sizeof(q)/sizeof(q[0]);

    queryResults(str, q, m, n, prefix, suffix, power);
    return (0);
}
```

Output :

```
The Substring [0 10] is a palindrome
The Substring [5 8] is not a palindrome
The Substring [2 5] is not a palindrome
The Substring [5 9] is a palindrome
```