Periodic words (periodicwords)

Viene fornita una stringa S, seguita da Q query che forniscono due interi, l e r. Una stringa F si dice periodica se $F = t + t + \cdots + t$ per qualche stringa $t \neq F$. Il problema chiede per ogni query, se la sottostringa $S[l \dots r]$ è periodica.

Soluzione in $\mathcal{O}(N \log N + Q\sqrt[3]{N})$

Questo problema è risolvibile tramite un hashing polinomiale modulo P, dove P è un numero primo arbitrariamente grande, come $10^9 + 7$.

Definiamo con $H(S)$ l'hash della stringa $S = [a_0, a_1, \dots, a_n]$, che sarà generato nel seguente modo:
$H(S) = a_0 + a_1 p^1 + a_2 p^2 + \cdots + a_{n-1} p^{n-1}$
Dove p è un numero intero positivo scelto arbitrariamente.

Quindi $H(S[l \dots r]) \equiv H(S[0 \dots r]) - H(S[0 \dots l-1]) \pmod P$ (se $l \neq 0$)
Due stringhe $S[a \dots b], S[c \dots d]$ (con $c > b$) sono uguali con probabilità $\frac{P-1}{P}$ se $H(S[a \dots b])p^{c-a} \equiv H(S[c \dots d]) \pmod P$

Per trovare $H(S[l \dots r])$ in $\mathcal{O}(1)$ si può quindi ricorrere all'uso di un prefix array, precomputabile in $\mathcal{O}(N)$.

Si può dimostrare facilmente che una stringa è periodica se e solo se l'intervallo di periodicità è lungo quanto un divisore della lunghezza della stringa.
Di conseguenza, per ogni query è sufficiente controllare se la stringa è periodica per una sottostringa di lunghezza uguale ad un divisore.
Per eseguire questo controllo in $\mathcal{O}(1)$ è necessaria un osservazione:

Claim 0.0.1 Stringa periodica

Se una stringa S è periodica per una sottostringa di lunghezza d, allora:
$H(S[l, r-d])p^d \equiv H(S[l+d, r]) \pmod P$

Possiamo trovare i divisori di tutti i numeri fino ad N in $\mathcal{O}(N \log N)$ tramite un crivello di Eratostene. N avrà al più $\sqrt[3]{N}$ divisori, di conseguenza la complessità finale sarà $\mathcal{O}(N \log N + Q\sqrt[3]{N})$

```cpp
#include <bits/stdc++.h>
#pragma GCC optimize("Ofast")
using namespace std;

int N;
string S;
int Q;

vector<long long> prefix;
const long long coeff = 0x3f4f5f6f;
const long long mod = 1000000007;
vector<vector<int>> divs;
vector<long long> basePower;


void precompute()
{
    prefix.resize(N);
    divs.resize(N+1);
    basePower.resize(N);

    long long tmp_coeff = coeff;
    prefix[0] = (long long)(S[0]-'a'+1);
    basePower[0] = 1;

    for(int i = 1; i < N; i++)
```

```
    {
        basePower[i] = tmp_coeff;
        prefix[i] = (prefix[i-1] + (long long)(S[i]-'a'+1) * tmp_coeff)%mod;
        tmp_coeff = (tmp_coeff * coeff)%mod;
    }

    //Sieve
    for (int i = 1; i <= N/2; ++i) {
        for (int j = 2 * i; j <= N; j += i) {
        divs[j].emplace_back(i);
    }
}
}

inline long long stringHash(int l, int r)
{
    if(l == 0) return prefix[r];

    return (prefix[r]-prefix[l-1]+mod)%mod;
}

inline bool compareHash(int l1,int r1, int l2, int r2)
{
    int div = r1-r2;

    long long fHash = stringHash(l1, r1);
    long long sHash = stringHash(l2, r2);

    if((sHash*basePower[div])%mod == fHash) return true;

    return false;
}

inline string solve(int l, int r)
{
    int lenght = r-l+1;
    for(auto div : divs[lenght])
    {
        if(compareHash(l+div, r, l, r-div))
        {
            return "YES";
        }
    }
    return "NO";
}

int main()
{
    cin >> N >> S >> Q;
    precompute();

    int l,r;
    for(int i = 0; i < Q; i++)
    {
        cin >> l >> r;
        cout <<  solve(l, r) << "\n";

    }
}
```