SPOJ MAXMATCH

Maximum Self-Matching

Prof. Edson Alves – UnB/FGA

Problema

You're given a string s consisting of letters 'a', 'b' and 'c'.

The matching function $m_s(i)$ is defined as the number of matching characters of s and its i-shift. In other words, $m_s(i)$ is the number of characters that are matched when you align the 0-th character of s with the i-th character of its copy.

You are asked to compute the maximum of $m_s(i)$ for all i $(1 \leq i \leq |s|)$. To make it a bit harder, you should also output all the optimal i's in increasing order.

Entrada e saída

Input

The first and only line of input contains the string s $(2 \leq |s| \leq 10^5)$.

Output

The first line of output contains the maximal $m_s(i)$ over all i.

The second line of output contains all the i's for which $m_s(i)$ reaches maximum.

Exemplo de entradas e saídas

Sample Input

caccacaa

Sample Output

4

3

Solução $O(N^2)$

- A função $m_s(i)$ corresponde à distância de Hamming entre a string s e a substring $b_i = s[i..(N-1)]$, com $i = 1, 2, \ldots, N$
- Esta distância de Hamming entre as strings s e t é dada por

$$D(s,t) = \sum_{i=0}^{m}(1 - \delta_{s[i]}(t[i])),$$

  onde $m = \min(|s|, |t|)$ e $\delta_j(j) = 1$ e $\delta_j(k) = 0$, se $j \neq k$
- Assim, D tem complexidade $O(N)$
- Uma solução que computa $D(s, b_i)$ para todos os valores de i tem complexidade $O(N^2)$, e leva ao veredito TLE

Solução $O(N \log N)$

- Considere as strings t_k tais que $t_k[i] = \delta_k(s[i])$
- Na string dada no exemplo, $t_a =$ "`01001011`", $t_b =$ "`00000000`" e $t_c =$ "`10110100`"
- A ideia é computar $m_s(i)$ como a soma das funções $m_i^k(s)$, com $k =$ 'a', 'b' e 'c', onde $m_i^k(s)$ é calculada a partir da string t_k
- Usando esta representação binária das strings, o cálculo da distância de Hamming corresponde ao produto escalar entre a string t_k e a substring b_i
- Estes produtos escalares surgem na multiplicação dos polinômios correspondentes a strings t_k e a reversa da string b_i

Solução $O(N \log N)$

- Assim, os valores de $m_i^k(s)$ para cada i podem ser computados todos de uma só vez, por meio da multiplicação de polinômios, em $O(N \log N)$
- Atente que, devido à multiplicação polinomial, o valor de $m_i^k(s)$ será o coeficiente do monômio de grau $i + N$, onde N é o tamando da string t_k
- Embora $m_i^k(N) = 0$, é preciso considerá-lo na composição final da resposta, uma vez que 0 pode ser o valor máximo obtido, e neste caso o índice N também deve ser listado
- Repetido o processo para cada valor de k, o problema pode ser resolvido em $O(N \log N)$

Solução $O(N \log N)$

```cpp
#include <bits/stdc++.h>

using namespace std;

const double PI { acos(-1.0) };

int reversed(int x, int bits)
{
    int res = 0;

    for (int i = 0; i < bits; ++i)
    {
        res <<= 1;
        res |= (x & 1);
        x >>= 1;
    }

    return res;
}
```

Solução $O(N \log N)$

```
void fft(vector<complex<double>>& xs, bool invert = false)
{
    int N = (int) xs.size();

    if (N == 1)
        return;

    int bits = 1;

    while ((1 << bits) != N)
        ++bits;

    for (int i = 0; i < N; ++i)
    {
        auto j = reversed(i, bits);

        if (i < j)
            swap(xs[i], xs[j]);
    }
```

Solução $O(N \log N)$

```
    for (int size = 2; size <= N; size *= 2)
    {
        auto signal = (invert ? 1 : -1);
        auto theta = 2 * signal * PI / size;
        complex<double> S1 { cos(theta), sin(theta) };

        for (int i = 0; i < N; i += size)
        {
            complex<double> S { 1 }, k { invert ? 2.0 : 1.0 };

            for (int j = 0; j < size / 2; ++j)
            {
                auto a { xs[i + j] }, b { xs[i + j + size/2] * S };
                xs[i + j] = (a + b) / k;
                xs[i + j + size/2] = (a - b) / k;
                S *= S1;
            }
        }
    }
}
```

Solução $O(N \log N)$

```
pair<int, vector<int>> solve(const string& s)
{
    const string cs { "abc" };

    int m = 0, N = (int) s.size();
    vector<int> is, ms(N + 1, 0);

    int size = 1;

    while (size < N + 1)
        size *= 2;

    size *= 2;

    for (auto c : cs)
    {
        vector<complex<double>> xs(size), ys(size);

        for (int i = 0; i < N; ++i)
            xs[i] = (s[i] == c ? 1 : 0);
```

Solução $O(N \log N)$

```
        for (int i = 0; i < N; ++i)
            ys[i] = xs[N - 1 - i];

        fft(xs);
        fft(ys);

        for (int i = 0; i < size; ++i)
            xs[i] *= ys[i];

        fft(xs, true);

        for (int i = 1; i < N; ++i)
            ms[i] += (int) round(xs[N - 1 + i].real());
    }
```

Solução $O(N \log N)$

```
    for (int i = 1; i <= N; ++i) {
        if (ms[i] > m) {
            m = ms[i];
            is = vector<int> { i };
        } else if (ms[i] == m)
            is.push_back(i);
    }

    return make_pair(m, is);
}

int main()
{
    string s;
    cin >> s;

    auto ans = solve(s);

    cout << ans.first << '\n';
```

Solução $O(N \log N)$

```
    for (size_t i = 0; i < ans.second.size(); ++i)
        cout << ans.second[i] << (i + 1 == ans.second.size() ? '\n' : ' ');

    return 0;
}
```