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	Sample Output
caccacaa	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 $O(N \log N)$

```cpp
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 $O(N \log N)$

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

    return 0;
}
```

Bônus: redução da constante por meio de bijeção

- É possível reduzir a constante da solução por meio do uso de uma bijeção
- Seja T o texto principal e P o padrão a ser identificado em substrings de tamanho P de T
- Define, para cada caractere α do alfabeto Σ, o vetor binário v_{T_α}, onde $v_{T_\alpha}(i) = 1$ se $T[i] = \alpha$, ou zero, caso contrário
- Defina, de forma semelhante, o vetor v_{P_α}
- A solução anterior realizada a operação

$$\mathrm{IFFT}(\mathrm{FFT}(v_{T_\alpha}) \otimes \mathrm{FFT}(v_{P_\alpha}))$$

$|\Sigma|$ vezes, onde $\otimes$ é a multiplicação termo-a-termo

Bônus: redução da constante por meio de bijeção

- É possível reduzir o número de operações para $|\Sigma|/2$ por meio de uma bijeção entre os caracteres de Σ e números complexos
- Neste mapeamento, cada letra deve estar associada a um ângulo do círculo trigonométrico
- Para que as operações preservem as propriedades desejadas, os ângulo escolhidos devem ser as raízes $|\Sigma|$-ésimas da unidade
- Por exemplo, para $N = 3$, os caracteres devem ser mapeados aos ângulos $0, \frac{2\pi}{3}$ e $\frac{4\pi}{3}$
- As raízes $|\Sigma|$-ésimas da unidade são dadas por

$$w_k = \cos\left(\frac{2\pi k}{|\Sigma|}\right) + i \sin\left(\frac{2\pi k}{|\Sigma|}\right), \qquad k = 0, 1, \ldots, |\Sigma| - 1$$

Bônus: redução da constante por meio de bijeção

- Além da relação com as raízes da unidade, o mapeamento deve atender o seguinte critério: no produto termo-a-termo das transformadas de v_{T_α} e v_{P_α}, o produto entre caracteres iguais deve resultar em 1; entre caracteres diferentes, deve resultar em um número complexo com parte real igual a $\cos(\frac{2\pi}{|\Sigma|})$
- Isto pode ser alcançado mantendo-se os índices de 0 a $|\Sigma| - 1$ para o alfabeto do texto T, e usar os complementares $j = |\Sigma| - i$ para os índices do alfabeto para o padrão P
- Por exemplo, para $\Sigma = \{ a, b, c \}$, os índices para T seriam $a = 0, b = 1, c = 2$, e para P teríamos $a = 0, b = 2, c = 1$
- Observe que

$$a \cdot a = w_0 \cdot w_0 = 1$$

$$b \cdot b = w_1 \cdot w_2 = 1$$

$$c \cdot c = w_2 \cdot w_1 = 1$$

Bônus: redução da constante por meio de bijeção

- Ainda no caso $N = 3$, dado o número de comparações t de P sobre T e a parte real da convolução para cada índice p, vale que

$$D(T, P, t) = \mathrm{round}\left(\frac{t + 2\mathrm{Re}(v(p))}{|\Sigma|}\right),$$

onde $D(T, P, t)$ é a distância de Hamming entre T e P numa substring de tamanho t

- Usando o caso de teste do problema, no cenário abaixo

```
caccacaa
          caccacca
```

temos $T = caccacaa, P = caccacaa, |\Sigma| = 3$.

- São 4 posições corretas e 1 incorreta, de modo que a soma dos $\mathrm{Re}(v(p))$ seria igual a $4 \cdot 1 - 1 \cdot \frac{1}{2} = 3.5$, e como $t = 5$, vale que $\mathrm{round}((5 + 2 \cdot 3.5)/3) = 4$

Solução $O(N \log N)$

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

using namespace std;

typedef long long ll;

#define all(x) x.begin(), x.end()
#define MSOne(S) (1ull << (63 - __builtin_clzll(S)))
#define fastio ios_base::sync_with_stdio(0); \
               cin.tie(0); \
               cout.tie(0)

// FFT - CP Algorithm
using cd = complex<double>;
const double PI = acos(-1);

void fft(vector<cd>& a, bool invert) {
    int n = a.size();
```

Solução $O(N \log N)$

```
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)
            j ^= bit;
        j ^= bit;

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

    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
```

Solução $O(N \log N)$

```
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i + j], v = a[i + j + len / 2] * w;
                a[i + j] = u + v;
                a[i + j + len / 2] = u - v;
                w *= wlen;
            }
        }
    }

    if (invert) {
        for (cd& x : a)
            x /= n;
    }
}

string abc = "abc", acb = "acb";
int main() {
    fastio;
```

Solução $O(N \log N)$

```
    string s;
    cin >> s;
    string r(s);

    int N = s.size();
    ll size = MSOne((N + N) << 1);

    vector<cd> fs(size, 0), fr(size, 0);
    for (int j = 0; j < N; j++) {
        int sh = abc.find(s[j]);
        double sp = (2 * PI * sh) / 3;
        fs[j] = cd(cos(sp), sin(sp));

        int rh = acb.find(s[N - j - 1]);
        double rp = (2 * PI * rh) / 3;
        fr[j] = cd(cos(rp), sin(rp));
    }
    fft(fs, false);
    fft(fr, false);
```

Solução $O(N \log N)$

```
    for (int i = 0; i < size; i++) fs[i] *= fr[i];
    fft(fs, true);

    int mx = 0;
    vector<int> ms;
    for (int i = N; i < N + N; i++) {
        double t = fs[i].real();
        int tot = 2 * N - i - 1;
        double p = (tot + 2 * t) / 3.00;
        int k = (int)round(p);
        mx = max(mx, k);
        ms.push_back(k);
    }

    cout << mx << "\n";
    for (int i = 0; i < ms.size(); i++) if (ms[i] == mx) cout << i + 1 << " ";
    cout << "\n";

    return 0;
}
```

Referência

SCHOENMEYR, Tor, **ZHANG, David Yu**. FFT-based algorithms for the string matching with mismatches problem. Journal of Algorithms, Volume 57, Issue 2, November 2004, pg. 130-139.