

# 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.

### 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  $i = 1, 2, \dots, N$
- 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  $k = 'a', 'b' \text{ e } 'c'$ , 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 tamanho 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 $O(N \log N)$

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 const double PI { acos(-1.0) };
6
7 int reversed(int x, int bits)
8 {
9     int res = 0;
10
11     for (int i = 0; i < bits; ++i)
12     {
13         res <<= 1;
14         res |= (x & 1);
15         x >>= 1;
16     }
17
18     return res;
19 }
```



## Solução $O(N \log N)$

```
21 void fft(vector<complex<double>>& xs, bool invert = false)
22 {
23     int N = (int) xs.size();
24
25     if (N == 1)
26         return;
27
28     int bits = 1;
29
30     while ((1 << bits) != N)
31         ++bits;
32
33     for (int i = 0; i < N; ++i)
34     {
35         auto j = reversed(i, bits);
36
37         if (i < j)
38             swap(xs[i], xs[j]);
39     }
```

## Solução $O(N \log N)$

```
41  for (int size = 2; size <= N; size *= 2)
42  {
43      auto signal = (invert ? 1 : -1);
44      auto theta = 2 * signal * PI / size;
45      complex<double> S1 { cos(theta), sin(theta) };
46
47      for (int i = 0; i < N; i += size)
48      {
49          complex<double> S { 1 }, k { invert ? 2.0 : 1.0 };
50
51          for (int j = 0; j < size / 2; ++j)
52          {
53              auto a { xs[i + j] }, b { xs[i + j + size/2] * S };
54              xs[i + j] = (a + b) / k;
55              xs[i + j + size/2] = (a - b) / k;
56              S *= S1;
57          }
58      }
59  }
60 }
```

## Solução $O(N \log N)$

```
62 pair<int, vector<int>> solve(const string& s)
63 {
64     const string cs { "abc" };
65
66     int m = 0, N = (int) s.size();
67     vector<int> is, ms(N + 1, 0);
68
69     int size = 1;
70
71     while (size < N + 1)
72         size *= 2;
73
74     size *= 2;
75
76     for (auto c : cs)
77     {
78         vector<complex<double>> xs(size), ys(size);
79
80         for (int i = 0; i < N; ++i)
81             xs[i] = (s[i] == c ? 1 : 0);
```

## Solução $O(N \log N)$

```
83     for (int i = 0; i < N; ++i)
84         ys[i] = xs[N - 1 - i];
85
86     fft(xs);
87     fft(ys);
88
89     for (int i = 0; i < size; ++i)
90         xs[i] *= ys[i];
91
92     fft(xs, true);
93
94     for (int i = 1; i < N; ++i)
95         ms[i] += (int) round(xs[N - 1 + i].real());
96 }
```

## Solução $O(N \log N)$

```
98     for (int i = 1; i <= N; ++i) {
99         if (ms[i] > m) {
100             m = ms[i];
101             is = vector<int> { i };
102         } else if (ms[i] == m)
103             is.push_back(i);
104     }
105
106     return make_pair(m, is);
107 }
108
109 int main()
110 {
111     string s;
112     cin >> s;
113
114     auto ans = solve(s);
115
116     cout << ans.first << '\n';
```

## Solução $O(N \log N)$

```
118     for (size_t i = 0; i < ans.second.size(); ++i)
119         cout << ans.second[i] << (i + 1 == ans.second.size() ? '\n' : ' ');
120
121     return 0;
122 }
```

## 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  $\alpha$  do alfabeto  $\Sigma$ , o vetor binário  $v_{T_\alpha}$ , onde  $v_{T_\alpha}(i) = 1$  se  $T[i] = \alpha$ , ou zero, caso contrário
- Defina, de forma semelhante, o vetor  $v_{P_\alpha}$
- A solução anterior realizada a operação

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

$|\Sigma|$  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  $\Sigma$  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 ângulos 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

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



## 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  $v_{T_\alpha}$  e  $v_{P_\alpha}$ , 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

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

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

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

## Solução $O(N \log N)$

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 typedef long long ll;
6
7 #define all(x) x.begin(), x.end()
8 #define MSOne(S) (1ull << (63 - __builtin_clzll(S)))
9 #define fastio ios_base::sync_with_stdio(0); \
10             cin.tie(0); \
11             cout.tie(0)
12
13 // FFT - CP Algorithm
14 using cd = complex<double>;
15 const double PI = acos(-1);
16
17 void fft(vector<cd>& a, bool invert) {
18     int n = a.size();
```

## Solução $O(N \log N)$

```
20  for (int i = 1, j = 0; i < n; i++) {
21      int bit = n >> 1;
22      for (; j & bit; bit >>= 1)
23          j ^= bit;
24      j ^= bit;
25
26      if (i < j)
27          swap(a[i], a[j]);
28  }
29
30  for (int len = 2; len <= n; len <<= 1) {
31      double ang = 2 * PI / len * (invert ? -1 : 1);
32      cd wlen(cos(ang), sin(ang));
33      for (int i = 0; i < n; i += len) {
34          cd w(1);
```

## Solução $O(N \log N)$

```
34         cd w(1);
35         for (int j = 0; j < len / 2; j++) {
36             cd u = a[i + j], v = a[i + j + len / 2] * w;
37             a[i + j] = u + v;
38             a[i + j + len / 2] = u - v;
39             w *= wlen;
40         }
41     }
42 }
43
44 if (invert) {
45     for (cd& x : a)
46         x /= n;
47 }
48 }
49
50 string abc = "abc", acb = "acb";
51 int main() {
52     fastio;
```

## Solução $O(N \log N)$

```
54  string s;  
55  cin >> s;  
56  string r(s);  
57  
58  int N = s.size();  
59  ll size = MSOne((N + N) << 1);  
60  
61  vector<cd> fs(size, 0), fr(size, 0);  
62  for (int j = 0; j < N; j++) {  
63      int sh = abc.find(s[j]);  
64      double sp = (2 * PI * sh) / 3;  
65      fs[j] = cd(cos(sp), sin(sp));  
66  
67      int rh = acb.find(s[N - j - 1]);  
68      double rp = (2 * PI * rh) / 3;  
69      fr[j] = cd(cos(rp), sin(rp));  
70  }  
71  fft(fs, false);  
72  fft(fr, false);
```

## Solução $O(N \log N)$

```
74  for (int i = 0; i < size; i++) fs[i] *= fr[i];
75  fft(fs, true);
76
77  int mx = 0;
78  vector<int> ms;
79  for (int i = N; i < N + N; i++) {
80      double t = fs[i].real();
81      int tot = 2 * N - i - 1;
82      double p = (tot + 2 * t) / 3.00;
83      int k = (int)round(p);
84      mx = max(mx, k);
85      ms.push_back(k);
86  }
87
88  cout << mx << "\n";
89  for (int i = 0; i < ms.size(); i++) if (ms[i] == mx) cout << i + 1 << " ";
90  cout << "\n";
91
92  return 0;
93 }
```