Paradigmas de Resolução de Problemas

Divisão e Conquista – Transformada Rápida de Fourier: Exercícios Resolvidos

---

Prof. Edson Alves - UnB/FGA

2020

Sumário

OJ 12879 – Golf Bot

Problema

Do you like golf? I hate it. I hate golf so much that I decided to build the ultimate golf robot, a robot that will never miss a shot. I simply place it over the ball, choose the right direction and distance and, flawlessly, it will strike the ball across the air and into the hole. Golf will never be played again.

Unfortunately, it doesn't work as planned. So, here I am, standing in the green and preparing my first strike when I realize that the distance-selector knob built-in doesn't have all the distance options! Not everything is lost, as I have 2 shots.

Given my current robot, how many holes will I be able to complete in 2 strokes or less? The ball must be always on the right line between the tee and the hole. It isn't allowed to overstep it and come back.

Entrada e saída

Input

The input file contains several test cases, each of them as described below.

The first line has one integer: N, the number of different distances the Golf Bot can shoot. Each of the following N lines has one integer, k_i, the distance marked in position i of the knob.

Next line has one integer: M, the number of holes in this course. Each of the following M lines has one integer, d_j, the distance from Golf Bot to hole j.

Constraints:

- $1 \leq N, M \leq 200000$
- $1 \leq k_i, d_j \leq 200000$

Entrada e saída

Output

For each test case, you should output a single integer, the number of holes Golf Bot will be able to complete. Golf Bot cannot shoot over a hole on purpose and then shoot backwards.

Sample Output Explanation

Golf Bot can shoot 3 different distances (1, 3 and 5) and there are 6 holes in this course at distances 2, 4, 5, 7, 8 and 9. Golf Bot will be able to put the ball in 4 of these:

- The 1st hole, at distance 2, can be reached by striking two times a distance of 1.
- The 2nd hole, at distance 4, can be reached by striking with strength 3 and then strength 1 (or vice-versa).
- The 3rd hole can be reached with just one stroke of strength 5.
- The 5th hole can be reached with two strikes of strengths 3 and 5.

Holes 4 and 6 can never be reached.

Exemplo de entradas e saídas

Sample Input

```
3
1
3
5
6
2
4
5
7
8
9
```

Sample Output

```
4
```

Solução $O(K \log K + M)$

- Para cada distância d_j, é preciso verificar se é possível atingí-la com uma ou duas tacadas
- Com uma tacada basta verificar se d_j é igual a algum dos k_i, o que pode ser feito em $O(1)$ com uma tabela *hash*
- Já para duas tacadas é a verificação pode ser feita em $O(N)$, também usando tabelas *hash*
- Contudo, este algoritmo teria complexidade $O(NM)$, o que resultaria em um veredito TLE, uma vez que $N, M \leq 2 \times 10^5$
- A transformada de Fourier permite resolver este problema com complexidade $O(K \log K + M)$, onde $K = \max\{k_1, k_2, \ldots, k_N\}$

Solução $O(K \log K + M)$

- Seja $p(x) = a_0 + a_1 x + a_2 x^2 + \ldots$ um polinômio tal que $a_0 = 1$ e $a_r = 1$, se $r > 0$ e existe ao menos um $k_i = r$, ou $a_i = 0$, caso contrário
- Para o exemplo de entrada, $p(x) = 1 + x + x^3 + x^5$
- Uma vez que $1 = x^0$, cada monômio de $p(x)$ representa uma tacada possível (ou mesmo não dar tacada, no caso de x^0) onde o grau é a distância atingida pelo golpe
- As distâncias possíveis de serem atingidas com duas tacadas são dadas pelos monômios não nulos do polinômio $q(x) = p^2(x)$, pois

$$q(x) = p^2(x) = (a_0 + a_1 x + a_2 x^2 + \ldots)(a_0 + a_1 x + a_2 x^2 + \ldots)$$

  e a distributividade vai associar cada par de monômios possível
- Logo, para cada d_j basta verificar se a_j é igual a zero ou não
- O produto $p^2(x)$ pode ser obtido com a FFT em $O(K \log K)$, permitindo checar cada d_j em $O(1)$

Solução $O(K \log K + M)$

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

using namespace std;

const double PI { acos(-1.0) }, EPS { 1e-6 };

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

    if (N == 1)
        return;

    vector<complex<double>> es(N/2), os(N/2);

    for (int i = 0; i < N/2; ++i)
        es[i] = xs[2*i];

    for (int i = 0; i < N/2; ++i)
        os[i] = xs[2*i + 1];

```

Solução $O(K \log K + M)$

```
22      fft(es, invert);
23      fft(os, invert);
24
25      auto signal = (invert ? 1 : -1);
26      auto theta = 2 * signal * PI / N;
27      complex<double> S { 1 }, S1 { cos(theta), sin(theta) };
28
29      for (int i = 0; i < N/2; ++i)
30      {
31          xs[i] = (es[i] + S * os[i]);
32          xs[i] /= (invert ? 2 : 1);
33
34          xs[i + N/2] = (es[i] - S * os[i]);
35          xs[i + N/2] /= (invert ? 2 : 1);
36
37          S *= S1;
38      }
39  }
40
41  int solve(const vector<int>& ks, const vector<int>& ds)
42  {
```

Solução $O(K \log K + M)$

```
43      auto K = *max_element(ks.begin(), ks.end());
44
45      int size = 1;
46
47      while (size < K + 1)
48          size *= 2;
49
50      size *= 2;
51
52      vector<complex<double>> xs(size, 0);
53
54      for (auto k : ks)
55          xs[k] = 1;
56
57      xs[0] = 1;
58
59      fft(xs);
60
61      for (int i = 0; i < size; ++i)
62          xs[i] *= xs[i];
63
```

Solução $O(K \log K + M)$

```
64      fft(xs, true);
65
66      int ans = 0;
67
68      for (auto d : ds)
69          ans += (d < size and fabs(xs[d].real()) > EPS ? 1 : 0);
70
71      return ans;
72  }
73
74  int main()
75  {
76      ios::sync_with_stdio(false);
77
78      int N;
79
80      while (cin >> N)
81      {
82          vector<int> ks(N);
83
```

Solução $O(K \log K + M)$

```
84          for (int i = 0; i < N; ++i)
85              cin >> ks[i];
86
87          int M;
88          cin >> M;
89
90          vector<int> ds(M);
91
92          for (int i = 0; i < M; ++i)
93              cin >> ds[i];
94
95          auto ans = solve(ks, ds);
96
97          cout << ans << '\n';
98      }
99
100     return 0;
101 }
```

SPOJ MAXMATCH – Maximum Self-Matching

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 $O(N \log N)$

```
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     }
40 
```

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
55                  xs[i + j] = (a + b) / k;
56                  xs[i + j + size/2] = (a - b) / k;
57                  S *= S1;
58              }
59          }
60      }
61  }
```

Solução $O(N \log N)$

```
63 pair<int, vector<int>> solve(const string& s)
64 {
65     const string cs { "abc" };
66
67     int m = 0, N = (int) s.size();
68     vector<int> is, ms(N + 1, 0);
69
70     int size = 1;
71
72     while (size < N + 1)
73         size *= 2;
74
75     size *= 2;
76
77     for (auto c : cs)
78     {
79         vector<complex<double>> xs(size), ys(size);
80
81         for (int i = 0; i < N; ++i)
82             xs[i] = (s[i] == c ? 1 : 0);
83
```

Solução $O(N \log N)$

```
84          for (int i = 0; i < N; ++i)
85              ys[i] = xs[N - 1 - i];
86
87          fft(xs);
88          fft(ys);
89
90          for (int i = 0; i < size; ++i)
91              xs[i] *= ys[i];
92
93          fft(xs, true);
94
95          for (int i = 1; i < N; ++i)
96              ms[i] += (int) round(xs[N - 1 + i].real());
97      }
98
99      for (int i = 1; i <= N; ++i)
100     {
101         if (ms[i] > m)
102         {
103             m = ms[i];
104             is = vector<int> { i };
```

Solução $O(N \log N)$

```
105         } else if (ms[i] == m)
106             is.push_back(i);
107     }
108 
109     return make_pair(m, is);
110 }
111 
112 int main()
113 {
114     string s;
115     cin >> s;
116 
117     auto ans = solve(s);
118 
119     cout << ans.first << '\n';
120 
121     for (size_t i = 0; i < ans.second.size(); ++i)
122         cout << ans.second[i] << (i + 1 == ans.second.size() ? '\n' : ' ');
123 
124     return 0;
125 }
```

Referências

1. OJ 12879 – Golf Bot
2. SPOJ MAXMATCH - Maximum Self-Matching