Algoritmos de Busca

A busca é o algoritmo fundamental dentre os algoritmos de strings e, conforme dito anteriormente, se equivalente, em importância, aos algoritmos de ordenação no estudo de algoritmos.

A busca em strings consiste em determinar se uma string `pat`, de tamanho `m`, ocorre ou não em uma string `text`, de tamanho `n` (ou o número de tais ocorrências).

Os principais algoritmos de busca em strings são: a busca completa, o algoritmo de Knuth-Morris-Pratt (KMP) e o algoritmo de Boyer-Moore (BM). O primeiro deles é de fácil entendimento e codificação; os dois últimos são conceitualmente divididos em duas etapas:

1. **pré-processamento do padrão**, que envolve a construção de certas tabelas;
2. **busca**, onde se determina a primeira (ou todas) ocorrência(s) de `pat` em `text`.

Um outro algoritmo de busca, baseado no uso de funções *hash*, é o algoritmo de Rabin Karp.

Busca Completa

A busca completa compara cada uma das `n - m + 1` substrings de tamanho `m` de `text` com `pat`, reportando cada igualdade. Como a comparação tem complexidade `O(m)` e o número de substrings é `O(n)`, temos um algoritmo com complexidade `O(mn)`.

Uma possível implementação em C++ é ilustrada abaixo.

```
int occurrences(const string& text, const string& pat)
{
    int n = text.size();
    int m = pat.size();

    int occ = 0;    // Número de ocorrências de pat em text

    for (int i = 0; i <= n - m; ++i)
        occ += (pat == text. substr(i, m) ? 1 : 0);

    return occ;
}
```

O único cuidado a ser tomado, na implementação, é se certificar que todas as substrings foram verificadas (atentar para o `<=` na condição do laço).

Outro ponto importante a se notar é que a comparação entre `pat` e a subtring pode ser feita tanto da esquerda para direita quanto em sentido oposto, e estas duas alternativas constituem as ideias fundamentais para os outros dois algoritmos: KMP e BM.

Algoritmo de Morris-Pratt

Na função `occurrences()` descrita acima, as comparações feitas entre a substring em questão e o padrão são independentes, o que resulta em várias comparações sendo feitas mais de uma vez e desnecessáriamente.

Por exemplo, considere `text = "xyzabcdfgh"` e `pat = "abcde"`. A comparação entre a substring com início no índice 3 (a saber, `abcdf`) e o padrão falha no último caractere (`'f' != 'e'`), localizado no índice 7. Como todos os caracteres do padrão são distintos, é possível identificar que o padrão não pode ocorrer a partir dos índices de 4 a 6, mas `occurrences()` ainda assim realiza tais comparações.

O algoritmo de Morris-Pratt explora justamente as comparações entre caracteres já feitas, movendo o índice de ínicio das comparações entre as substrings e o padrão para a posição mais distante possível. Para melhor explicar o funcionamento de tal algoritmo, precisamos de algumas definições preliminares.

Chamamos **salto seguro** `s` ao inteiro positivo tal que há garantias de que o padrão não pode acontecer entre as posições `i` e `i + s` do texto, mas que pode-se iniciar o padrão em `i+s`.

Quando o padrão contém caracteres distintos, é seguro saltar para a posição onde aconteceu a falha. Contudo, devemos ter cuidado quando há repetições de letras no padrão. Mais precisamente, para que o salto seja seguro, devemos identificar a maior borda possível para `pat[1..j]`: devemos saltar para a posição onde esta borda se inicia. Assim, o **salto seguro de Morris-Pratt** para o padrão `pat[1..j], j = 1, 2, ..., m` é dado por

```
MP_shift[j] = j - |border(pat[1..j])|
```

Lembre-se de que `border(s)` é a maior substring `x` de `s` que é, ao mesmo tempo, sufixo e prefixo de `s`. No caso especial de uma string vazia, o salto deve assumir o valor mínimo de 1, de modo que devemos fazer `MP_shift[0] = 1`.

De posse do vetor `MP_shift`, temos uma possível implementação do algoritmo de Morris-Pratt.

```
int MP(const string& text, const string& pat)
{
    int n = text.size();
    int m = pat.size();
    int i = 0, j = 0, occ = 0;

    vector<int> bords = borders(pat);

    while (i <= n - m)
    {
        while (j < m and pat[j] == text[i + j])
            ++j;

        if (j == m)
            ++occ;

        int MP_shift = j - bords[j];

        i += MP_shift;
        j = max(0, j - MP_shift);
    }

    return occ;
}
```

O algoritmo de Morris-Pratt realiza, no máximo, um número de comparações linear em termos dos tamanhos de `text` e `pat`, a saber, `2n - m` comparações. Isto pode ser observado da seguinte maneira: a comparação `pat[j] == text[i + j]` pode falhar, no máximo, `n - m + 1` vezes (o primeiro laço é executado `n - m + 1` vezes) e pode ter sucesso, no máximo, `n` vezes (quando o texto é período e o padrão é o período). Caso a primeira comparação seja bem sucedida, ela não pode falhar no índice 0. O pior caso, em termos de número de comparações, acontece quando o padrão tem apenas duas letras distinas e o texto é uma repetição de `n - 1` vezes a primeira letra do padrão e a última letra é igual a segunda letra do padrão (por exemplo, `pat = ab` e `text = aaaaaaaaaaaab`. Neste cenário, a primeira comparação será bem sucedida, `n - 1` vezes, haverão `n - 2` falhas (em relação ao último caractere do padrão), e uma última comparação bem sucedida no último caractere. Daí o máximo de comparações será igual a `(n - 1) + (n - 2) + 2 = 2n - 2 = 2n - m`.

Assim, o algoritmo MP é linear em relação ao tamanho do texto. Porém, para determinar sua complexidade, falta determinar a complexidade da construção do vetor `MP_shift`. Este vetor pode ser construído a partir de uma variação do próprio MP, de modo que teremos uma construção também linear em relação ao tamanho do padrão. Daí, a complexidade do algoritmo MP é `O(n)`.

Para construir `MP_shift`, determinaremos os valores de `border(pat[1..j])` a partir de uma variante do algoritmo MP, onde `text == pat`, apresentada a seguir.

```
// Computa o tamanho das bordas de pat: bord[j] é o tamanho da borda da
// substring pat[0..(j-1)]
vector<int>
borders1(const string& pat)
{
    int m = pat.size();
    int i = 1, j = 0;

    vector<int> bord(m + 1, 0);    // Inicialmente, bord[j] = 0 para todo j
    bord[0] = -1;

    while (i < m + 1)
    {
        while (i + j < m and pat[j] == pat[i + j])
        {
            ++j;
            bord[i + j] =  max(bord[i + j], j);
        }

        i += j - bord[j];
        j = max(0, bord[j]);
    }

    return bord;
}
```

Vale notar que, no momento das atualizações de `i` e `j`, ao final do primeiro laço, os valores de `bord[j]` já estão devidamente computados, e que `i` inicia em 1, não em zero (`bord[0] = -1`: uma string de tamanho zero não tem bordas não triviais e o valor -1 faz com que a atualização de `i` seja igual a, no mínimo, um).

Contudo, o algoritmo mais comum para o cálculo das bordas é apresentado abaixo. Assim como a variação do MP, também é linear em relação ao tamanho de `pat`.

```
// Computa o tamanho das bordas de pat: bord[j] é o tamanho da borda da
// substring pat[0..(j-1)]
vector<int> borders2(const string& pat)
{
    int m = pat.size();
    int t = -1;

    vector<int> bord(m + 1);
    bord[0] = -1;

    for (int j = 1; j <= m; ++j)
    {
        while (t >= 0 and pat[t] != pat[j -  1])
            t = bord[t];

        ++t;
        bord[j] = t;
    }

    return bord;
}
```

Algoritmo de Knuth-Morris-Pratt

O algoritmo de Morris-Pratt tem como invariante

```
inv(i, j) = (pat[1..j] = text[i + 1, i + j + 1]),
```

que permite os saltos seguros e que leva a complexidade `O(n + m)`. Contudo, este invariante pode ser melhorada, ao se incorporar a propriedade da diferença, isto é,

```
inv2(i, j) = (pat[1..j] = text[i + 1, j + 1]) and (pat[j + 1] != text[i + j + 1]
```

Para entender o porque da melhora, considere o seguinte exemplo: seja `pat = "abcabc"` e `text = "abcabdabc"`. Os 5 primeiros caracteres de ambos coincidem, e a diferença ocorre no sexto caractere: `pat[5] = 'c' != text[5] = 'd'`. Mas `border[5] = 2 ("ab")`, o que deslocaria a próxima comparação para `pat[2]` e `text[5]`. Contudo, esta comparação é idêntica a anterior, pois a borda "ab" não é própria, isto é, o próximo caractere ('c') gera uma nova borda "abc". A contribuição de Knuth para o algoritmo de Morris-Pratt é essa: incorporar a propriedade da diferença e definir as **bordas estritas**, onde o próximo caractere não gera uma nova borda.

Ao definirmos `Strong_Bord[j]` como o menor inteiro `k` que satisfaz a condição

```
`cond(j, k) = (pat[1..k] é sufixo próprio de pat[1..j]) and
    (pat[k + 1] != pat[j + 1])
```

(ou -1, caso não exista tal inteiro) e

```
KMP_Shift[j] = j - Strong_Bord[j],
```

a implementação do KMP é praticamente idêntica ao do MP:

```
int KMP(const string& text, const string& pat)
{
    int n = text.size();
    int m = pat.size();
    int i = 0, j = 0, occ = 0;

    vector<int> bords = strong_borders(pat);

    while (i <= n - m)
    {
        while (j < m and pat[j] == text[i + j])
            ++j;

        if (j == m)
            ++occ;

        int KMP_shift = j - strong_bords[j];

        i += KMP_shift;
        j = max(0, j - KMP_shift);
    }

    return occ;
}
```

Para o cálculo das bordas estritas basta observar que uma borda estrita é uma borda cujo próximo caractere do prefixo e do sufixo diferente. Caso estes caracteres sejam iguais, a borda estrita então deve ser avaliada novamente no prefixo encontrado.

Abaixo segue uma implementação em C++ para o cálculo das bordas estritas, que é uma variação do segundo algoritmo de bordas apresentado para o algoritmo de Morris-Pratt. Atente ao fato de que a primeira condição do `if` (isto é, `j == m`) subentende um caractere sentinela concatenado ao final do padrão.

```
vector<int>
strong_borders(const string& pat)
{
    int m = pat.size();
    int t = -1;

    vector<int> sbord(m + 1);
    sbord[0] = -1;

    for (int j = 1; j <= m; ++j)    // t é igual a bord[j - 1]
    {
        while (t >= 0 and pat[t] != pat[j - 1])
            t = sbord[t];

        ++t;

        if (j == m or pat[t] != pat[j])
            sbord[j] = t;
```

```
            else
                sbord[j] = sbord[t];
        }

        return sbord;
    }
```

Algoritmo de Karp-Rabin

O algoritmo de Karp-Rabin é baseado no uso de funções *hash* para a comparação entre strings. Uma função *hash* é uma função `f: A -> B` entre dois conjuntos `A` e `B` com as seguintes propriedades:

1. `f(a)` pode ser computada "eficientemente" para todos elementos `a` de `A`;
2. é altamente provável que `f(x) != f(y)` se `x != y`.

Vale observar mais atentamente a segunda propriedade: sendo função, temos que `f(x) != f(y)` implica em `x != y`. Porém, a definição de funções não veda a possibilidade de termos `f(x) = f(y)` com `x != y`. Esta situação é denominada **colisão**, e uma boa função *hash* tem uma baixa probabilidade de ocorrência de colisões. Se `f` for construída de tal forma que não exista a possibilidade de colisões, a função `f` é dita uma função de *hash* perfeita.

No contexto de strings, o conjunto `A` é o conjunto `S` de todas as strings possíveis de um alfabeto fixo, e `B` é, em geral, o conjunto dos números inteiros.

A ideia central do algoritmo de Karp-Rabin é computar o hash `f(pattern)` do padrão e compará-lo com os *hashes* `f(text[i+1..i+m]` de todas as `n - m + 1` substrings de tamanho `m` de `text` (`n = |text|, m = |pattern|`). Contudo, o algoritmo necessita de uma propriedade adicional para `f`: `f(i+2..i+m+1)` deve ser "facilmente computável" a partir de `f(i+1..i+m)`. Por conta desta propriedade o algoritmo de Karp-Rabin também é conhecido como *rolling hash*.

Um exemplo de *hash* com as três propriedades necessárias é o seguinte: considere o alfabeto `A` e seja `id(a)` um inteiro entre 1 e `|A|` que representa o índice (posição) do caractere `a` em `A`. Seja

```
f: S -> Z(p)
    s -> f(s[1..m]) = (id(s[m]) + id(s[m-1])*(|A| + 1) + ... +  id(s[1])*(|A| + 1)^{m-1}) % p
```

onde `p` é um número primo. Se `t = s[2..m+1]` então

```
f(t) = f(s[2..m+1])
= (id(s[m+1]) + id(s[m])*(|A| + 1) + ... +  id(s[2])*(|A| + 1)^{m-1}) % p
= ((f(s) - id(s[1])*(|A| + 1)^{m-1})*(|A| + 1) + id(s[m + 1]) % p
```

Assim, `f(s[2..m+1])` pode ser obtido a partir de `f(s[1..m])` em `O(1)` se o valor de `(|A|+1)^{m-1}` for precomputado.

Uma implementação possível desta função em C/C++ é dada a seguir, com um exemplo de seu uso.

```
#include <iostream>

using namespace std;
using ll = long long;

// res = a^n (mod p)
ll fast_mod_exp(ll a, ll n, ll p)
{
    ll res = 1, base = a;

    while (n)
    {
        if (n & 1)
        {
            res *= base;
            res %= p;
        }

        base *= base;
        base %= p;

        n >>= 1;
    }
```

```
    return res;
}

const string A { "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789  " };
const ll size = A.size();
const ll P = 1000000007;    // P = 10^9 + 7

ll idA(char c)
{
    return A.find(c);
}

// res = f(s[pos..pos + m - 1]), size = |A|, prev = f(s[pos-1..pos+m])
ll f(const string& s, int pos, ll prev, int m)
{
    ll res = 0;

    if (pos == 0)
    {
        for (int i = 0; i < m; ++i)
            res = (size*res + idA(s[pos + i])) % P;
    } else
    {
        ll SM = fast_mod_exp(size, m - 1, P);

        res = (prev - idA(s[pos - 1])*SM)*size + idA(s[pos+m-1]);
        res %= P;
    }

    return res;
}

int main()
{
    string s = "abcdef", pattern = "cde";

    int m = pattern.size();
    ll h = f(pattern, 0, 0, m);

    printf("pattern (%s) hash = %lld \n", pattern.c_str(), h);

    h = f(s, 0, 0, m);
    printf("s[0..2] (%s) hash = %lld \n", s.substr(0, m).c_str(), h);

    h = f(s, 1, h, m);
    printf("s[1..3] (%s) hash = %lld \n", s.substr(1, m).c_str(), h);

    h = f(s, 2, h, m);
    printf("s[2..4] (%s) hash = %lld \n", s.substr(2, m).c_str(), h);

    h = f(s, 3, h, m);
    printf("s[3..5] (%s) hash = %lld \n", s.substr(3, m).c_str(), h);

    return 0;
}
```

As funções apresentadas acima permitem uma implementação do algoritmo de Karp-Rabin em C/C++:

```
int Karp_Rabin(const string& text, const string& pattern)
{
    int m = pattern.size(), n = text.size(), count = 0;
    ll r = f(pattern, 0, 0, m), h = 0;

    for (int i = 0; i < n - m + 1; ++i)
    {
        h = f(text, i, h, m);

        if (h == r and text.substr(i, m) == pattern)
            ++count;
    }

    return count;
}
```

Se o padrão for pequeno (entre 8 e 12 caracteres), é possível definir uma função de hash `g` perfeita, que evita a necessidade de verificar se a igualdade entre os *hashes* são ou não uma colisão. Considere que o número de caracteres `|A|` do alfabeto pode ser representado por `n` bits, e que uma variável do tipo `long long` tenha 64 *bits* de tamanho. Se o tamanho `m` do padrão for menor ou igual a `64/n`, então a função `g` abaixo é um *hash* perfeito.

```
ll g(const string& s, int pos, ll prev, int m, int n)
{
    ll res = 0;

    if (pos == 0)
    {
        for (int i = 0; i < m; ++i)
        {
            res <<= n;
            res |= idA(s[i]);
        }
    } else
    {
        ll mask = ( 1 << n*m) - 1;

        res = ((prev << n) | idA(s[pos + m - 1])) & mask;
    }

    return res;
}
```

No pior caso (por exemplo, onde o padrão e o texto consistem na repetição de um mesmo caractere), o algoritmo de Karp-Rabin tem complexidade `O(nm)`. Se um *hash* perfeito for utilizado, o algoritmo tem complexidade `O(n + m)`. Na prática, para padrões grandes, é melhor utilizar o KMP: o algoritmo de Rabin-Karp é uma alternativa mais adequada a padrões pequenos.

Z Function

A Z Function é uma função que recebe como parâmetro uma string `S` de tamanho `n` (indexada em `0`) e retorna o vetor `z`, onde cada posição `i` do vetor `z` tem o seguinte significado:

```
z[i] - maior prefixo comum da string S e da substring S[i..n-1]
```

Isto é, `z[i]` guarda o tamanho do maior prefixo comum de `S` e do sufixo de `S` que começa na posição `i`. Por definição, `z[0]` tem valor `0`, pois o sufixo de `S` que começa na posição `0` é a própria string `S` (mas isso pode ser ajustado a depender da necessidade do problema). Por exemplo, dada a string `abacaba`, o vetor `z` teria os valores seguintes armazenados:

```
i    - 0 1 2 3 4 5 6
S    - a b a c a b a
z[i] - 0 0 1 0 3 0 1
```

O sufixo que começa na posição `1` tem como primeira letra o `b`, enquanto a primeira letra de `abacaba` é `a`; logo o maior prefixo comum dessas strings é `0` e, consequentemente, `z[1] = 0`. O sufixo que começa na posição `4` tem como primeira letra o `a`, como segunda o `b` e como terceira o `a`; essas 3 letras são as 3 primeiras letras da string `abacaba`, logo `z[4] = 3`.

Versão Ingênua O(n²)

A complexidade de computação do vetor `z`, isto é, a complexidade da própria Z Function é `O(n)`, sendo `n` o tamanho de `S`. Antes de mostrar o código da Z Function, vamos analisar uma versão mais ingênua do algoritmo, que tem complexidade `O(n²)`:

```
vector<int> naive_z_function(const string &s) {
    int n = s.size();
    vector<int> z(n, 0);
    for(int i = 1; i < n; i++) {
        while(i + z[i] < n && s[z[i]] == s[i + z[i]]) {
            z[i]++;
        }
    }
```

```
    return z;
}
```

A ideia do código acima é simples: inicializamos o vetor `z` de tamanho `n` com todas suas posições zeradas. Logo após, para cada posição `i` de `1` a `n-1`, aumentamos o tamanho do maior prefixo comum de `S` e `S[i..n-1]` enquanto o carácter atual não está fora da string e enquanto os caracteres equivalentes forem iguais. Por exemplo: para uma posição `i` qualquer, inicialmente, `z[i] = 0`; logo, verificamos se `s[0]` e `s[i]` são iguais; se forem, incrementamos `z[i]` em `1` e agora verificamos se `s[1]` e `s[i + 1]` são iguais; se forem iguais também, incrementamos `z[i]` novamente; fazemos isso até encontrar dois carácteres diferentes ou até chegarmos ao final da string.

Para cada posição `i`, no máximo `O(n)` operações são feitas (isto acontece quando o sufixo inteiro que começa em `i` é um prefixo de `S`). Como a string `S` tem `n` posições, temos a complexidade final do algoritmo ingênuo de $O(n^2)$.

Versão Original O(n)

Tendo a função mostrada acima como inspiração, é possível modificá-la para atingirmos a Z Function com complexidade `O(n)`. Abaixo segue o código em C++ da Z Function:

```
vector<int> z_function(const string &s) {
    int n = s.size();
    vector<int> z(n, 0);
    int L = 0, R = 0;
    for(int i = 1; i < n; i++) {
        if(i <= R) {
            z[i] = min(z[i - L], R - i + 1);
        }
        while(z[i] + i < n && s[z[i] + i] == s[z[i]]) {
            z[i]++;
        }
        if(R < i + z[i] - 1) {
            L = i, R = i + z[i] - 1;
        }
    }
    return z;
}
```

Perceba que as únicas diferenças da versão original para a ingênua são os dois `ifs` dentro do for e as variáveis `L` e `R`. O `while` é exatamente igual ao da versão ingênua: isso é interessante porque a essência de ambas as abordagens são parecidas, com a sutil diferença de que na original utilizaremos prefixos comuns já calculados para diminuir o número de vezes que o `while` roda.

A bela ideia desse algoritmo é o uso dos dois "ponteiros" `L` e `R`. Quando o `for` está numa posição `i` qualquer, esses ponteiros irão representar o começo e o fim de um maior prefixo comum não nulo (isto é, maior que 0) já encontrado entre `S` e algum sufixo `S[k..n-1]`, para `k < i`. Em específico, os ponteiros representam o começo e o fim de algum sufixo `S[k..n-1]` de tal forma que `R` é o máximo possível.

Por exemplo, seja `S = abacababac`. O vetor `z` de `S` teria a seguinte forma:

```
i    - 0 1 2 3 4 5 6 7 8 9
S    - a b a c a b a b a c
z[i] - 0 0 1 0 3 0 4 0 1 0
```

Suponha que o `for` está na nona iteração (`i = 8`). Neste ponto, 3 maiores prefixos comuns não nulos já foram encontrados para trás: entre `S` e `S[2..9]` com tamanho 1, entre `S` e `S[4..9]` com tamanho 3 e entre `S` e `S[6..9]` com tamanho 4. Ou seja, respectivamente, a substring `S[2..2]` é igual ao prefixo `S[0..0]`, a substring `S[4..6]` é igual ao prefixo `S[0..2]` e a substring `S[6..9]` é igual ao prefixo `S[0..3]`.

O intervalo de cada uma dessas substrings (`(2, 2)`, `(4, 6)` e `(6, 9)`) seriam candidatos a serem os valores guardados nos ponteiros `L` e `R`; como nesse caso o intervalo que tem maior `R` é o intervalo `(6, 9)` (`9` no caso), então, no começo da nona iteração (`i = 8`), as variáveis terão os valores `L = 6` e `R = 9`.

Agora é onde entra em ação o primeiro `if`:

```
if (i <= R) {
    z[i] = min(z[i - L], R - i + 1);
}
```

Se esta condição for verdadeira, isto é, se `i` está dentro do intervalo `[L, R]`, quer dizer que a substring `S[i..R]` é exatamente igual à substring `S[i-L..R-L]`, pois como `L` e `R` representam as pontas de um prefixo comum já calculado, tem-se que `S[0..R-L] = S[L..R]`. A operação `i-L` encontra a posição de `S[0..R-L]` equivalente à posição `i` de `S[L..R]`. E como `z[i-L]` já foi calculado, sabemos exatamente qual é o maior prefixo comum entre `S` e `S[i..R]`. Igualamos, então, `z[i]` ao mínimo entre `z[i-L]` e `R-i+1`, pois a única informação que temos é sobre a substring `S[i..R]` e, caso `z[i-L]` seja maior que o tamanho de tal substring, não sabemos se os carácteres depois de `S[R]` serão iguais aos carácteres depois de `S[R-L]`. Então, a ideia básica desse `if` é "adiantar" alguns carácteres para que o `while` rode normalmente mas menos vezes.

O segundo if tem uma função muito simples: atualizar os "ponteiros" `L` e `R` caso a ponta direita da substring `S[i..i+z[i]-1]` vá "mais longe" que a ponta direita da substring armazenada atualmente; ou seja, se `i+z[i]-1` for maior que `R`, então os "ponteiros" passam a "apontar" para a substring `S[i..i+z[i]-1]`.

```
if (R < i + z[i] - 1) {
    L = i, R = i + z[i] - 1;
}
```

Para analisar a complexidade desse algoritmo, temos que analisar quantas iterações o `while` faz. Para cada iteração que o `while` faz, o `R` cresce, indiretamente, em `1`. Isto acontece porque o `while` só verifica posições depois de `R`, já que o primeiro `if` faz o matching das posições anteriores em `O(1)`. Assim, como o `R` cresce no máximo `n` vezes, o `while` roda no máximo `n` vezes. Então, a complexidade amortizada do algoritmo é `O(n)`.

Aplicação #1 - Matching de Strings

Uma das aplicações possíveis para a Z Function é, também, o matching de strings. Isto é, dados um texto `T` de tamanho `n` e um padrão de busca `P` de tamanho `m` (ambos indexados em 0), é possível saber todas as ocorrências de `P` em `T`. A ideia da aplicação é a seguinte: definamos a string `S` como:

```
S = P + '#' + T
```

Isto é, a string `S` é formada pelo padrão `P` concatenado a algum carácter separador e concatenado ao texto `T`. Observação: o carácter separador não pode aparecer nem em `P` e nem em `T`.

Tendo `S` em mãos, calculamos o seu respectivo vetor `z` (perceba que, nesse caso, o vetor `z` tem tamanho `n+m+1`, pois `S` é a concatenação de `P`, do carácter separador e de `T`). Um matching de `P` acontece em uma posição `i` de `T` se, e somente se, `z[i + m + 1] = m`.

Para exemplificar a afirmação acima, peguemos um exemplo: seja `P = "ana"` (`m = 3`) e `T = "banana"` (`n = 6`); concatenamos as strings para obter a string `S = ana#banana`. O vetor `z` de `S` vai ter os seguintes valores:

```
i    -  0 1 2 3 4 5 6 7 8 9
S    -  a n a # b a n a n a
z[i] -  0 0 1 0 0 3 0 3 0 1
```

As posições que importam nesse caso são as que estão após o carácter separador; portanto, analisemos as posições no intervalo `[4..9]`. Como a Z Function calcula o maior prefixo comum entre a própria string e cada um de seus sufixos, se colocarmos o padrão de busca como prefixo de `S`, e, para algum `i`, `z[i] = m`, então temos certeza que a substring que começa na posição `i` e tem tamanho `m` é exatamente igual ao padrão de busca! No exemplo dado acima, é fácil ver que as substrings que começam nas posições `5` e `7` são matches de `P` e, de fato, `z[5] = z[7] = m = 3`.

Em `S`, temos o padrão `P` e o carácter separador concatenados antes do texto `T`; por esse motivo, os índices de `z` em que um matching ocorre não são os índices verdadeiros de `T`. Para saber as posições exatas dos matchings em `T`, basta subtrair `m+1` dos índices que ocorrem os matches. No exemplo acima, os matchings ocorrem em `5` e `7`, mas os indíces verdadeiros de `T` são `5 - (m+1) = 5 - 4 = 1` e `7 - (m+1) = 7 - 4 = 3`, respectivamente.

Como `S` tem tamanho igual a `n+m+1` e a complexidade da Z Function é linear em relação ao tamanho da string de input, a complexidade da estratégia de string matching descrita acima é `O(n + m)`.

Aplicação #2 - Matching de Strings com apenas um caractere de diferença

Outro problema bem interessante que pode ser resolvido com a ajuda da Z Function é um problema similar ao anterior: matching de strings com, no máximo, 1 carácter de diferença. Por exemplo, se `P = caco` (`m = 4`) e `T = cabococacoto` (`n = 12`), consideraríamos as substrings em negrito como ocorrências de `P` em `T`: **cabo**cacoto e caboco**caco**to. Na

primeira ocorrência, o terceiro carácter do padrão `P` é diferente do terceiro carácter da aparição, mas, como dito anteriormente, é permitido no máximo 1 carácter de diferença. No segundo caso, a aparição é um matching perfeito. Esse tipo de matching é bem útil em problemas do mundo real, quando a busca feita por alguém não precisa ser totalmente perfeita.

Para resolver este problema, precisaremos criar duas strings: `S = P + '#' + T` (igual à `S` do problema anterior) e `S' = rev(P) + '#' + rev(T)`, onde o significado de `rev(A)` é: a string `A` ao contrário. Para `P` e `T` do exemplo acima, teríamos:

```
P      = caco
rev(P) = ocac
T      = cabococacoto
rev(T) = otocacocobac
S      = caco#cabococacoto
S'     = ocac#otocacocobac
```

Com `S` e `S'` em mãos, calculamos os seus respectivos vetores z: `z` e `z'`. Os valores para `z` e `z'` serão:

```
i     - 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
S     - c  a  c  o  #  c  a  b  o  c  o  c  a  c  o  t  o
z[i]  - 0  0  1  0  0  2  0  0  0  1  0  4  0  1  0  0  0

i     - 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
S'    - o  c  a  c  #  o  t  o  c  a  c  o  c  o  b  a  c
z'[i] - 0  0  0  0  0  1  0  4  0  0  0  2  0  1  0  0  0
```

O significado de `z[i]` (nesse caso) já é conhecido por nós: maior prefixo comum entre o padrão de busca `P` e a substring `T[i-(m+1)..n-1]` (isto é, sufixo de `T` começando na posição `i-(m+1)`) do texto `T`; mas e o significado de `z'[i]` em relação à string S, qual é? Em termos abstratos, `z'` armazena os tamanhos dos maiores sufixos comuns do padrão de busca `P` e dos prefixos de `T`.

Por exemplo, para clarificar, peguemos o primeiro matching do exemplo acima e o padrão `P`: `cabo` e `P = caco`; é fácil ver que o maior prefixo comum entre os dois tem tamanho `2` (`ca`) e o maior sufixo comum entre os dois tem tamanho 1 (`o`). Ou seja, indo do "começo" para o "fim" de ambas as strings, um "matching" de tamanho `2` é encontrado, enquanto partindo do "fim" das strings para o "começo", um matching de tamanho `1` é encontrado. Em outras palavras, `3` carácteres iguais entre `cabo` e `P = caco` foram encontrados. Os únicos (único, nesse caso) carácteres que não deram matching são os carácteres entre o maior prefixo comum e o maior sufixo comum de ambas as strings (isto é, os carácteres no "meio" das strings). Como `3` carácteres certeiros foram encontrados e o tamanho do padrão `P` é `m = 4`, sabemos que apenas um carácter entre o padrão `P = caco` e a substring `cabo` de `T` são diferentes, o que está dentro das condições do problema. Portanto, consideramos `cabo` como um matching de `caco` (não é um matching perfeito, mas é um matching). Em termos gerais, o números de carácteres "certeiros" entre uma substring `A` de `T` e o padrão `P` é `min(m, tamanho_maior_prefixo_comum(A, P) + tamanho_maior_sufixo_comum(A, P))`; se esse valor for `m` ou `m - 1`, a substring `A` é considerada um matching. Para achar tais tamanho para uma substring de `T` começando em `i` é só usar `z[i]` e `z'[k]`, sendo `k` o índice ajustado (pois `S'` é a concatenação das strings reversas, daí `k = (n + m) - m - i = n - i`).

Por fim, a Z Function é uma função bem poderosa que pode ser usada para resolver vários tipos de problemas de string (talvez até outro temas?), bastando apenas ter boa criatividade derivada de prática para usar suas propriedades de maneiras inteligentes.

Exercícios

1. Codeforces
    i. 625B - War of the Corporations
2. URI
    i. 2049 - Números de Ahmoc
3. UVA
    i. 455 - Periodic Strings
    ii. 10298 - Power Strings
    iii. 11362 - Phone List

iv. 11475 - Extend to Palindrome

v. 11576 - Scrolling Sign

vi. 11888 - Abnormal 89's

Referências

HALIM, Steve; HALIM, Felix. Competitive Programming 3, Lulu, 2013.

CROCHEMORE, Maxime; RYTTER, Wojciech. Jewels of Stringology: Text Algorithms, WSPC, 2002.