Programação Dinâmica

Vários problemas relacionados às strings podem ser resolvidos com o uso de Programação Dinâmica.

Edit Distance

Sejam `s` e `t` duas strings, com `s != t`. Quando comparadas por meio de um algoritmo de *matching*, dentre os motivos que levam o algoritmo a retornar falso estão:

1. ambas strings tem mesmo tamanho, mas diferem em um ou mais símbolos. Por exemplo,

```
s = "banana"
t = "bacana"
```

onde `s[2] != t[2]`;

1. a primeira string é mais longa que a segunda, e poderíam se tornar iguais se removidos os caracteres excedentes. Por exemplo, se

```
s = 'aspectos'
r = 'seco'
```

então `s == r` se removidos os caracteres das posições 0, 2, 4 e 6 de `s`;

1. a primeira string é mais curta do que a segunda, e poderia se igualar a primeira se adicionados os caracateres ausentes. Por exemplo,

```
s = 'fga'
r = 'formigas'
```

se tornariam iguais com a adição dos caracteres `ormis` em `s`, nas devidas posições.

Na prática, é possível obter qualquer string `r` a partir de uma string `s` dada, usando uma sequência finita das operações descritas acima (alterar um caractere, adicionar um caractere ou remover um caractere). O problema denominado *edit distance* consiste em determinar o número mínimo de operações a serem feitas (ou, em termos mais gerais, o custo mínimo desta transformação, se a cada operação for associado um determinado custo).

Se denotarmos `edit(s, t)` como o menor número de operações que transforma `s` em `t` teremos as seguintes propriedades:

1. `edit(s, t) >= 0`
2. `edit(s, t) = 0` se, e somente se, `s = t`
3. `edit(s, t) = edit(t, s)` (simetria)
4. `edit(s, t) <= edit(s, r) + edit(r, t)` (desigualdade triangular)

Considere que `|s| = m, |t| = n` . Para determinar `edit(s, t)` , devemos construir uma tabela auxiliar de estados `st` , onde `st(i, j) = edit(s[1..i], t[1..j]` , com `0 < i <= m, 0 < j <= n` . O casos bases acontecem quando uma das duas strings é vazia: nestes casos, o mínimo de operações a serem feitas é igual um número de inserções correspondente ao tamanho da string não vazia. Em notação simbólica,

```
st(0, j) = j, st(i, 0) = i
```

Se os custos de inserção, de remoção e de alteração forem `c_i, c_r, c_s` , respectivamente, então os casos bases devem ser

```
st(0, j) = j*c_i    // j inserções
st(i, 0) = i*c_r    // i remoções
```

A transição entre os estados será, dentre as três operações, a de menor custo. uma transição por inserção seria dada por

```
st(i, j) = st(i, j - 1) + c_i,      // Adicionar t[j]
```

por remoção seria

```
st(i, j) = st(i - 1, j) + c_r,      // Remover s[i]
```

e por alteração seria

```
// Mantém s[i] ou substitui s[i] por t[j]
st(i, j) = st(i - 1, j - 1) + c_s * (s[i] == t[j] ? 0 : 1)
```

Assim,

```
st(i, j) = min(st(i, j - 1) + c_i,
               st(i - 1, j) + c_r,
               st(i - 1, j - 1) + c_s * (s[i] == t[j] ? 0 : 1)
           )
```

Como a tabela tem `mn` estados, e cada transição é feita em `O(1)` , o algoritmo tem complexidade `O(mn)` .

Abaixo uma implementação *bottom-up* em C++. Recorde que, em C++, as strings são indexadas a partir de zero, e não um, como na notação anterior. Desta forma, no código abaixo `st[i][j]` significa o custo mínimo para transformar `s.substr(i)` em `t.substr(j)` .

```cpp
#define MAX_M   1000
#define MAX_N   1000

int st[MAX_M][MAX_N];
int c_i = 1, c_r = 1, c_s = 1;      // Custos iguais a um

int edit(const string& s, const string& t)
{
    int m = s.size();
    int n = t.size();

    for (int i = 0; i <= m; ++i)
        st[i][0] = i*c_r;

    for (int j = 1; j <= n; ++j)
        st[0][j] = j*c_i;

    for (int i = 1; i <= m; ++i)
        for (int j = 1; j <= n; ++j)
        {
            int insertion = st[i][j - 1] + c_i;
            int deletion = st[i - 1][j] + c_r;
            int change = st[i - 1][j - 1] + c_s * (s[i - 1] == t[j - 1] ? 0 : 1);
            st[i][j] = min(insertion, deletion);
            st[i][j] = min(st[i][j], change);
        }
```

```
    return st[m][n];
}
```

Observe que a implementação acima tem custo de memória `O(mn)` . É possível implementar o mesmo algoritmo usando apenas `O(n)` de memória, uma vez que é necesário apenas a linha anterior para computar os valores da próxima linha.

```
#define MAX_N   1000

int a[MAX_N], b[MAX_N];
int c_i = 1, c_r = 1, c_s = 1;      // Custos iguais a um

// Implementação _bottom-up_, O(mn), memória O(n)
int edit2(const string& s, const string& t)
{
    int m = s.size();
    int n = t.size();

    int *prev = a, *line = b;

    for (int j = 0; j <= n; ++j)
        prev[j] = j*c_i;

    for (int i = 1; i <= m; ++i)
    {
        line[0] = i*c_r;

        for (int j = 1; j <= n; ++j)
        {
            int insertion = line[j - 1] + c_i;
            int deletion = prev[j] + c_r;
            int change = prev[j - 1] + c_s * (s[i - 1] == t[j - 1] ? 0 : 1);
            line[j] = min(insertion, deletion);
            line[j] = min(line[j], change);
        }

        swap(line, prev);
    }

    return prev[n];
}
```

Esta segunda implementação pode ser necessária em competições com limites de memória restritos. Porém, esta implementação torna deveras mais complicado determinar as operações necessárias para se obter `t` a partir de `s` : na primeira implementação, basta manter um registro da operação responsável pela atualização de cada elemento da tabela `st` , conforme apresentado no código abaixo.

```
// x        Deletion
// c        Insertion of char c
// -        Keep
// [c->d]   Change (c to d)
string edit_operations(const string& s, const string& t)
{
    int m = s.size();
    int n = t.size();

    for (int i = 0; i <= m; ++i)
    {
        st[i][0] = i*c_r;
        ps[i][0] = 'r';
    }

    for (int j = 1; j <= n; ++j)
    {
        st[0][j] = j*c_i;
        ps[0][j] = 'i';
    }

    for (int i = 1; i <= m; ++i)
        for (int j = 1; j <= n; ++j)
        {
            int insertion = st[i][j - 1] + c_i;
            int deletion = st[i - 1][j] + c_r;
            int change = st[i - 1][j - 1] + c_s * (s[i - 1] == t[j - 1] ? 0 : 1);
```

```
                st[i][j] = min(insertion, deletion);
                st[i][j] = min(st[i][j], change);

                if (insertion <= deletion  and insertion <= change)
                    ps[i][j] = 'i';
                else if (deletion <= change)
                    ps[i][j] = 'r';
                else
                    ps[i][j] = 's';
            }

        int i = m, j = n;
        stack<string> S;
        char buffer[128];
        string subs = "[x->y]";

        while (i or j)
        {
            switch (ps[i][j]) {
            case 'i':
                sprintf(buffer, "%c", t[j - 1]);
                --j;
                break;

            case 'r':
                sprintf(buffer, "x");
                --i;
                break;

            case 's':
                if (s[i-1] == t[j-1])
                    sprintf(buffer, "-");
                else
                    sprintf(buffer, "[%c->%c]", s[i - 1], t[j - 1]);

                --i;
                --j;
            }

            S.push(buffer);
        }

        ostringstream os;

        while (not S.empty())
        {
            os << S.top();
            S.pop();
        }

        return os.str();
    }
```

Maior Subsequência Comum

Dadas duas strings `s` e `t`, o problema de se determinar a maior subsequência comum a elas (*Longest Common Subsequence - LCS*) pode ser interpretado como uma variante do *edit distance*.

Basta notar que tal sequência é formada por caracteres comuns às duas strings. Se atribuídos pesos iguais a zero às operações de inserção e remoção, peso infinitamente negativo à substituição e peso um à opção de manter os caracteres iguais, a LCS surge como o caminho com maior custo na tabela da *edit distance*.

```
#define MAX_M   1001
#define MAX_N   1001

#define INF 1000000010

int st[MAX_M][MAX_N], a[MAX_N], b[MAX_N];
int c_i = 0, c_r = 0, c_s = 1;      // Custos adaptados
char ps[MAX_M][MAX_N];

// Implementação _bottom-up_, O(mn), memória O(mn)
int lcs(const string& s, const string& t)
```

```
    {
        int m = s.size();
        int n = t.size();

        for (int i = 0; i <= m; ++i)
            st[i][0] = i*c_r;

        for (int j = 1; j <= n; ++j)
            st[0][j] = j*c_i;

        for (int i = 1; i <= m; ++i)
            for (int j = 1; j <= n; ++j)
            {
                int insertion = st[i][j - 1] + c_i;
                int deletion = st[i - 1][j] + c_r;
                int change = st[i - 1][j - 1] + c_s * (s[i - 1] == t[j - 1] ? 1 : -INF);
                st[i][j] = max(insertion, deletion);
                st[i][j] = max(st[i][j], change);
            }

        return st[m][n];
    }
```

A LCS pode ser recuperada de forma idêntica a utiliza para a recuperação das operações da *edit distance*.

```
    string lcs_str(const string& s, const string& t)
    {
        int m = s.size();
        int n = t.size();

        for (int i = 0; i <= m; ++i)
        {
            st[i][0] = i*c_r;
            ps[i][0] = 'r';
        }

        for (int j = 1; j <= n; ++j)
        {
            st[0][j] = j*c_i;
            ps[0][j] = 'i';
        }

        for (int i = 1; i <= m; ++i)
            for (int j = 1; j <= n; ++j)
            {
                int insertion = st[i][j - 1] + c_i;
                int deletion = st[i - 1][j] + c_r;
                int change = st[i - 1][j - 1] + c_s * (s[i - 1] == t[j - 1] ? 1 : -INF);

                st[i][j] = max(insertion, deletion);
                st[i][j] = max(st[i][j], change);

                if (insertion >= deletion and insertion >= change)
                    ps[i][j] = 'i';
                else if (deletion >= change)
                    ps[i][j] = 'r';
                else
                    ps[i][j] = 's';
            }

        int i = m, j = n;
        stack<char> S;

        while (i or j)
        {
            switch (ps[i][j]) {
            case 'i':
                --j;
                break;

            case 'r':
                --i;
                break;

            case 's':
                if (s[i-1] == t[j-1])
```

```
                S.push(s[i-1]);

            --i;
            --j;
        }
    }

    ostringstream os;

    while (not S.empty())
    {
        os << S.top();
        S.pop();
    }

    return os.str();
}
```

Quando todos os elementos de `s` e de `t` são distintos (isto é, `s[i] != s[j]` se `i != j`, o mesmo para `t`), o problema de se determinar a LCS pode ser reduzido ao problema de se determinar a maior sequência crescente (*Longest Increasing Subsequence* - *LIS*). Para tal, basta criar uma sequência de índices crescente `{ a_i }` tal que `s[a_i] = t[j]` para algum `j`. Em outras palavras, `{ a_i }` é sequência crescente de índices de `s` dos caracteres de `s` que coincidem com algum dos caracteres de `t`. Determinada a sequência `{ a_i }`, a LIS desta sequência corresponderá a LCS dentre as duas strings.

A vantagem desta abordagem é que, enquanto a LCS tem implementação `O(n^2)`, a LIS pode ser implementada em `O(n log n)`. Abaixo segue uma implementação possível da LIS com a complexidade `O(n log n)`.

```
#define MAX 1000010

int dp[MAX];

// Calcula o tamanho da LIS. Complexidade: O(nlog n)
int lis(const vector<int>& a)
{
    dp[0] = -1;
    int n = 0;

    for (auto x : a)
    {
        if (x > dp[n])      // Aumenta a LIS, quando possível
        {
            dp[++n] = x;
        } else              // Melhora os índices para possíveis aumentos futuros
        {
            auto it = lower_bound(dp + 1, dp + n, x);
            *it = min(*it, x);
        }
    }

    return n;
}
```

Maior Subsequência Palíndroma

Outro problema de string que pode ser resolvido por meio da programação dinâmica é o de se encontrar a maior subsequência de uma string `s` que forma um palíndromo (*Longest Palindrome Subsequence* - *LPS*). Outra maneira de se enunciar este mesmo problema é a seguinte: qual é o maior palíndromo que pode ser formado removendo `m` (`0 <= m <= n`) caracteres, de quaisquer posições, de uma string de tamanho `n` ?

Este problema sempre tem solução, pois uma string com apenas um caractere é um palíndromo (assim como strings vazias). Esta observação nos dá os casos bases do problema: se `LPS[i,j]` é o tamanho da maior subsequência palíndroma da substring `s[i..j]`, então

```
LPS[i,i] = 1
LPS[i,j] = 0    se i > j
```

Temos três transições possíveis:

1. Remover o caractere mais à esquerda da string (neste caso, `LPS[i,j] = LPS[i+1,j]`);
2. Remover o caractere mais à direita da string (neste caso, `LPS[i,j] = LPS[i,j-1]`);
3. Casos os caracteres que estão nos extremos da strings sejam iguais, removê-los e aumentar em duas unidades o tamanho da LPS (isto é, `LPS[i,j] = LPS[i+1,j-1] + 2` , se `s[i] == s[j]`).

Uma possível implementação para este problema se encontra a seguir:

```
#define MAX_N   1001

int st[MAX_N][MAX_N];

int lp_dp(const string& s, int i, int j)
{
    if (i > j)
        return 0;

    if (i == j)
        return 1;

    if (st[i][j] != - 1)
        return st[i][j];

    int res = max(lp_dp(s, i + 1, j), lp_dp(s, i, j - 1));

    if (s[i] == s[j])
        res = max(res, lp_dp(s, i + 1, j - 1) + 2);

    st[i][j] = res;

    return res;
}

// Implementação _top-down_, O(n^2), memória O(n^2)
int longest_palindrome(const string& s)
{
    int n = s.size();

    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
        st[i][j] = - 1;

    return lp_dp(s, 0, n - 1);
}
```

Assim como no caso da LCS, é possível recuperar a string correspondente a LPS armazendo-se os estados utilizados na atualização de cada estado e usando-se recursão ou uma pilha.

Exercícios

1. URI
    i. 1833 - Decoração Natalina
2. UVA
    i. 526 - String Distance and Transform Process
    ii. 10192 - Vacation
    iii. 10405 - Longest Common Subsequence
    iv. 10635 - Prince and Princess
    v. 10739 - String to Palindrome
    vi. 11151 - Longest Palindrome

Referências

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

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