

# Strings

String e Programação Dinâmica – Maior Subsequência Comum

---

Prof. Edson Alves - UnB/FGA

2019

1. Maior Subsequência Comum
2. Variantes da LCS

# Maior Subsequência Comum

---

## Definição

- Uma subsequência comum  $b = sc(S, T)$  entre duas strings  $S$  e  $T$  é uma sequência de pares de índices  $(i_k, j_k)$  tais que

$$1 \leq i_k \leq |S|, 1 \leq j_k \leq |T| \quad \text{e} \quad S[i_k] = T[j_k],$$

para todo  $k = 1, 2, \dots, |b|$

- Por exemplo, se  $S = \text{"casa"}$  e  $T = \text{"nevasca"}$ , então

$$b_1 = \{(3, 5)\}, b_2 = \{(1, 6), (2, 7)\} \quad \text{e} \quad b_3 = \{(2, 4), (3, 5), (4, 7)\}$$

são subsequências comuns de  $S$  e  $T$

- O problema de se determinar a maior subsequência comum entre  $S$  e  $T$  (*Longest Common Subsequence – LCS*) consiste em determinar o maior elemento do conjunto dos tamanhos substrings comuns, isto é,

$$LCS(S, T) = \max\{|b| \mid b = sc(S, T)\}$$

- Observe que pode existir duas (ou mais) subsequências comuns  $b_1$  e  $b_2$  de  $S$  e  $T$  tais que  $|b_1| = |b_2| = LCS(S, T)$

- O LCS pode ser interpretado como uma variante do *edit distance*
- Basta notar que uma sequência  $b$  de índices tal que  $|b| = LCS(S, T)$  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 1 à opção de manter os caracteres iguais, a LCS surge como o caminho com maior custo na tabela da  $edit(S, T)$
- Isto porque, com a atribuição de pesos descrita, serão mantido o maior número de caracteres comuns entre ambas possível, e as inserções e remoções, de custo zero, completarão a transformação
- Esta abordagem tem complexidade  $O(nm)$

# Visualização de $LCS(S, T)$

$LCS(i, j)$		'n'	'e'	'v'	'a'	's'	'c'	'a'
'c'	0	0	0	0	0	0	0	0
'a'	0	0	0	0	1	1	1	2
's'	0	0	0	0	1	2	2	2
'a'	0	0	0	0	1	2	2	3

# Implementação da LCS em C++

```
9 int LCS(const string& s, const string& t)
10 {
11     const int c_i = 0, c_r = 0, c_s = 1;           // Custos modificados
12     int m = s.size(), n = t.size();
13
14     for (int i = 0; i <= m; ++i)
15         st[i][0] = i*c_r;
16
17     for (int j = 1; j <= n; ++j)
18         st[0][j] = j*c_i;
19
20     for (int i = 1; i <= m; ++i)
21         for (int j = 1; j <= n; ++j) {
22             int insertion = st[i][j - 1] + c_i;
23             int deletion = st[i-1][j] + c_r;
24             int change = st[i-1][j-1] + c_s*(s[i-1] == t[j-1] ? 1 : -oo);
25             st[i][j] = max({ insertion, deletion, change });
26         }
27
28     return st[m][n];
29 }
```

## **Variantes da LCS**

---



# Identificação da LCS

- Assim como o problema de *edit distance*, uma variante comum do LCS é determinar a sequência de operações que leva à maior subsequência comum
- A implementação é idêntica à proposta para  $edit(S, T)$ , uma vez aplicada a modificação dos pesos e a alteração da operação  $\min()$  por  $\max()$ , de modo que a complexidade permanece sendo  $O(nm)$
- A maior subsequência comum corresponde aos caracteres onde os caracteres foram mantidos
- Assim esta rotina pode ser modificada para exibir a sequência, e não as operações que levaram a ela

# Identificação da LCS em C++

```
74 // -      Deletion
75 // c      Insertion of char c
76 // =      Keep
77 // [c->d]  Change (c to d)
78 string LCS(const string& s, const string& t)
79 {
80     const int c_i = 0, c_r = 0, c_s = 1;      // Custos modificados
81     int m = s.size(), n = t.size();
82
83     for (int i = 0; i <= m; ++i)
84     {
85         st[i][0] = i*c_r;
86         ps[i][0] = 'r';
87     }
88
89     for (int j = 1; j <= n; ++j)
90     {
91         st[0][j] = j*c_i;
92         ps[0][j] = 'i';
93     }
94 }
```

# Identificação da LCS em C++

```
95     for (int i = 1; i <= m; ++i)
96         for (int j = 1; j <= n; ++j) {
97             int insertion = st[i][j - 1] + c_i;
98             int deletion = st[i-1][j] + c_r;
99             int change = st[i-1][j-1] + c_s*(s[i-1] == t[j-1] ? 1 : -oo);
100             st[i][j] = max({ insertion, deletion, change });
101
102             ps[i][j] = (insertion >= deletion and insertion >= change) ?
103                 'i' : (deletion >= change ? 'r' : 's');
104         }
105
106     int i = m, j = n;
107     string b;
108
109     while (i or j)
110     {
111         switch (ps[i][j]) {
112             case 'i':
113                 --j;
114                 break;
115
```

# Identificação da LCS em C++

```
116     case 'r':  
117         --i;  
118         break;  
119  
120     case 's':  
121         if (s[i-1] == t[j-1])  
122             b.push_back(s[i-1]);  
123  
124         --i;  
125         --j;  
126     }  
127 }  
128  
129 reverse(b.begin(), b.end());  
130  
131 return b;  
132 }
```

- Quando todos os caracteres de  $S$  e de  $T$  são distintos (isto é,  $S[i] \neq S[j]$  se  $i \neq j$ , 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, seja  $\{a_i\}$  a sequência crescente de índices de  $S$  tais que  $S[a_i] = t[j]$  para algum  $j \in [1, m]$
- Em outras palavras,  $\{a_i\}$  é sequência crescente de índices de caracteres de  $S$  que coincidem com algum dos caracteres de  $T$
- Seja  $\{b_k\}$  a sequência onde  $b_k = j_k$ , onde  $S[a_k] = T[j_k]$
- A LIS de  $\{b_k\}$  corresponderá a LCS entre as duas strings
- A vantagem desta abordagem é que, enquanto a LCS tem implementação  $O(nm)$ , a LIS pode ser implementada em  $O(n \log n)$

## Visualização de $LCS(S, T)$ como $LIS$

$i$	1	2	3	4	5	6	7	8	9	10
$S$	't'	'r'	'a'	'p'	'e'	'z'	'i'	'o'		
$T$	'r'	'e'	't'	'i'	'c'	'u'	'l'	'a'	'd'	'o'
$a_i$										
$b_i$										

## Visualização de $LCS(S, T)$ como $LIS$

$i$	1	2	3	4	5	6	7	8	9	10
$S$	<u>'t'</u>	'r'	'a'	'p'	'e'	'z'	'i'	'o'		
$T$	'r'	'e'	<u>'t'</u>	'i'	'c'	'u'	'l'	'a'	'd'	'o'
$a_i$	1									
$b_i$	3									

## Visualização de $LCS(S, T)$ como $LIS$

$i$	1	2	3	4	5	6	7	8	9	10
$S$	't'	<u>'r'</u>	'a'	'p'	'e'	'z'	'i'	'o'		
$T$	<u>'r'</u>	'e'	't'	'i'	'c'	'u'	'l'	'a'	'd'	'o'
$a_i$	1	2								
$b_i$	3	1								



## Visualização de $LCS(S, T)$ como $LIS$

$i$	1	2	3	4	5	6	7	8	9	10
$S$	't'	'r'	<u>'a'</u>	'p'	'e'	'z'	'i'	'o'		
$T$	'r'	'e'	't'	'i'	'c'	'u'	'l'	<u>'a'</u>	'd'	'o'
$a_i$	1	2	3							
$b_i$	3	1	8							

## Visualização de $LCS(S, T)$ como $LIS$

$i$	1	2	3	4	5	6	7	8	9	10
$S$	't'	'r'	'a'	'p'	<u>'e'</u>	'z'	'i'	'o'		
$T$	'r'	<u>'e'</u>	't'	'i'	'c'	'u'	'l'	'a'	'd'	'o'
$a_i$	1	2	3	5						
$b_i$	3	1	8	2						

## Visualização de $LCS(S, T)$ como $LIS$

$i$	1	2	3	4	5	6	7	8	9	10
$S$	't'	'r'	'a'	'p'	'e'	'z'	<u>'i'</u>	'o'		
$T$	'r'	'e'	't'	<u>'i'</u>	'c'	'u'	'l'	'a'	'd'	'o'
$a_i$	1	2	3	5	7					
$b_i$	3	1	8	2	4					

## Visualização de $LCS(S, T)$ como $LIS$

$i$	1	2	3	4	5	6	7	8	9	10
$S$	't'	'r'	'a'	'p'	'e'	'z'	'i'	<u>'o'</u>		
$T$	'r'	'e'	't'	'i'	'c'	'u'	'l'	'a'	'd'	<u>'o'</u>
$a_i$	1	2	3	5	7	8				
$b_i$	3	1	8	2	4	10				

## Visualização de $LCS(S, T)$ como $LIS$

$i$	1	2	3	4	5	6	7	8	9	10
$S$	't'	'r'	'a'	'p'	'e'	'z'	'i'	'o'		
$T$	<u>'r'</u>	<u>'e'</u>	't'	<u>'i'</u>	'c'	'u'	'l'	'a'	'd'	<u>'o'</u>
$a_i$	1	2	3	5	7	8				
$b_i$	3	1	8	2	4	10				

$$LIS(b_k) = \{1, 2, 4, 10\}$$

$$LCS(S, T) = \text{"reio"}$$

# Implementação da LCS como LIS em C++

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 int lis(const vector<int>& as)
6 {
7     vector<int> st(as.size(), -1);
8     int n = 0;
9
10    for (auto x : as)
11    {
12        if (x > st[n]) // Aumenta a LIS, quando possível
13            st[++n] = x;
14        else {        // Melhora os elementos já encontrados
15            auto it = lower_bound(st.begin() + 1, st.begin() + n, x);
16            *it = min(*it, x);
17        }
18    }
19
20    return n;
21 }
```

# Implementação da LCS como LIS em C++

```
23 int lcs(const string& S, const string& T)
24 {
25     map<char, int> idx;
26
27     for (size_t i = 0; i < T.size(); ++i)
28         idx[T[i]] = i;
29
30     vector<int> bs;
31
32     for (const auto& c : S)
33     {
34         auto it = idx.find(c);
35
36         if (it != idx.end())
37             bs.push_back(it->second);
38     }
39
40     return lis(bs);
41 }
```

1. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.
2. **CROCHEMORE**, Maxime; **RYTTER**, Wojciech. *Jewels of Stringology: Text Algorithms*, WSPC, 2002.