Programação Dinâmica

Programação Dinâmica

Programação dinâmica, assim como o método de divisão e conquista, soluciona problemas pela combinação de soluções de subproblemas.

- **Algoritmos de divisão e conquista** dividem o problema em **subproblemas disjuntos**, soluciona-os recursivamente e então combina suas soluções para encontrar a solução do problema original.

- **Programação dinâmica** é aplicada quando os **subproblemas que se sobrepõem**, isto é, quando subproblemas compartilham subproblemas.

Neste contexto, divisão e conquista faz muito mais trabalho do que o necessário, solucionando repetidamente subproblemas, enquanto que a programação dinâmica soluciona o subproblema e armazena sua solução, em uma tabela, sendo reaproveitada.

Programação Dinâmica

Geralmente, a programação dinâmica é aplicada em **problemas de otimização**, com um valor ótimo (máximo ou mínimo) mas com diversas soluções distintas que alcançam tal valor. Ao construir um algoritmo de programação dinâmica observa-se uma sequência de quatro passos:

1 – Caracterizar uma estrutura de solução ótima.

2 – Definir, recursivamente, o valor ótimo.

3 – Computar o valor de uma solução ótima de uma maneira ascendente.

4 – Construir uma solução ótima a partir de informações computadas.

Os passos de 1-3 são básicos, em programação dinâmica, para encontrar o valor ótimo da solução do problema. O passo 4 deve ser executado caso seja necessária uma solução explícita. Para fazer o passo 4 é preciso manter informações adicionais durante o passo 3.

Problema do Corte de Hastes de Ferro

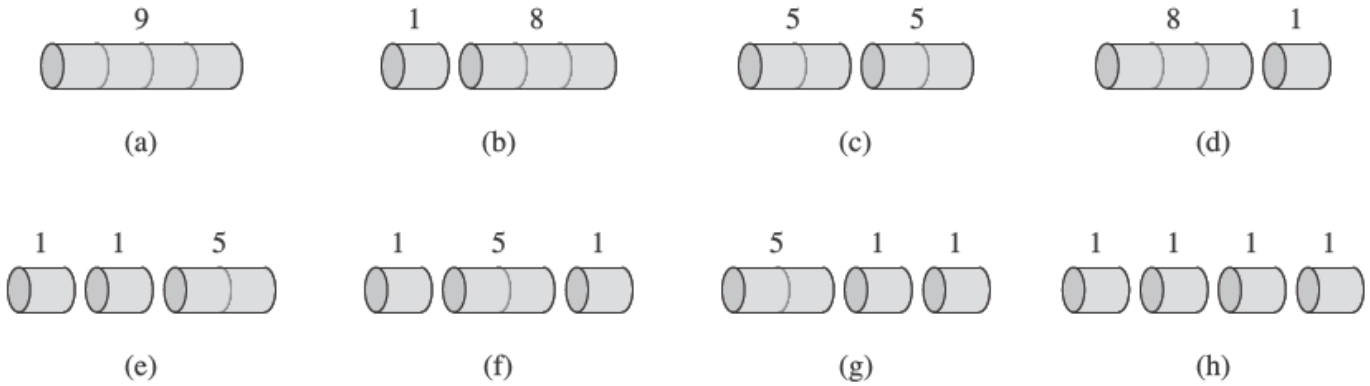
Dada uma haste de tamanho n metros e uma tabela de preços p_i para i=1,2,…,n, determine o máximo preço de revenda r_n possível, cortando a haste e vendendo em pedaços. Note que se o preço p_n para uma haste de tamanho n for grande o suficiente, uma solução ótima pode ser não cortar a haste.

Considere a seguinte tabela de preços p_i:

Tamanho i	1	2	3	4	5	6	7	8	9	10
Preço pi	1	5	8	9	10	17	17	20	24	30

Caracterizar uma estrutura de solução ótima

Consiste em determinar uma forma de dividir o problema em subproblemas que permita alcançar uma solução ótima. Uma ideia é cortar a haste de tamanho n de todas as maneiras possíveis e encontrar o máximo preço de revenda (r) somando os preços dos pedaços. Porém uma haste de tamanho n pode ser cortada, ou decomposta, de 2^{n-1} formas diferentes. A Figura 1 mostra os cortes possíveis e o preço de cada pedaço para uma haste de tamanho n=4.



Caracterizar uma estrutura de solução ótima

Temos a opção de cortar, ou não cortar, a haste em uma distância de i metros a partir da extremidade esquerda para i=1, 2, … , n-1. Denotamos uma decomposição da haste em uma forma aditiva ordinária, como 7=3+2+2 indicando que uma haste de tamanho 7 foi decomposta em 3 pedaços com os respectivos tamanhos de 3, 2 e 2.

Desta forma se uma solução ótima corta a haste em k pedaços, para algum $1 \leq k \leq n$, então uma decomposição ótima

$$n = i_1 + i_2 + \ldots + i_k$$

da haste em pedaços de tamanhos $i_1, i_2, \ldots, i_k$ determinam o correspondente ao máximo valor de revenda

$$r_n = p_1 + p_2 + \ldots + p_k.$$

Caracterizar uma estrutura de solução ótima

Para nosso exemplo na Figura 1 acima, o máximo valor de revenda r = 10 determinado pela decomposição 4=2+2. Da mesma forma é possível determinar o preço de revenda ótimo (ri), para as hastes com os tamanhos i = 1, 2, … , 10, observando todos as formas de decompô-las e somando seus os preços de cada pedaço dado pela Tabela 1. Assim chegamos às seguintes soluções ótimas:

r_1 = 1 a partir da solução 1 = 1 (sem cortes),
r_2 = 5 a partir da solução 2 = 2 (sem cortes),
r_3 = 8 a partir da solução 3 = 3 (sem cortes),
r_4 = 10 a partir da solução 4 = 2 + 2,
r_5 = 13 a partir da solução 5 = 2+3,
r_6 = 17 a partir da solução 6 = 6 (sem cortes),
r_7 = 18 a partir da solução 7 = 1 + 6 ou 7 = 2 + 2 + 3,
r_8 = 22 a partir da solução 8 = 2 + 6,
r_9 = 25 a partir da solução 9 = 3 + 6,
r_{10} = 30 a partir da solução 10 = 10 (sem cortes).

Caracterizar uma estrutura de solução ótima

De forma mais genérica, podemos enquadrar os valores rn, para n > 1, em termos de valores de revendas ótimos de hastes mais curtas através da fórmula

$$r_n = \text{máx}(p_n , r_1+r_{n-1} , r_2+r_{n-2} , \ldots , r_{n-1}+r_1) \qquad (1)$$

onde, o primeiro argumento de máx, p_n , é o preço da haste de tamanho n, sem corte. Os outros n-1 argumentos correspondem ao máximo preço de revenda obtido fazendo um corte inicial das hastes em dois pedaços de tamanhos i e n-i, para cada i = 1, 2, … , n-1, e, em seguida, cortar otimamente esses pedaços a fim de obter valores de revenda r_i e r_{n-i} desses dois pedaços. Como não sabemos antecipadamente qual valor de i otimiza o preço de revenda, temos que considerar todos os valores possíveis de i e escolher aquele que maximiza.

Caracterizar uma estrutura de solução ótima

Por exemplo, vamos fazer para n=4 e assim colocar na fórmula

$r_n = máx (p_4, r_1+r_3 , r_2+r_2 , r_3+r_1)$

mas quanto vale r_1, r_2 e r_3 ? A fórmula responde

$r_1 = máx (p_1) = máx (1) = 1$

$r_2 = máx (p_2 , r_1+r_1) = máx (5, 1+1) = máx(5, 2) = 5$

$r_3 = máx (p_3, r_1+r_2 , r_2+r_1) = máx (8, 1+5 , 5+1) = máx (8, 6 , 6) = 8$

assim temos que

$r_n = máx (9, 1+8 , 5+5 , 8+1) = máx(9, 9, 10, 9) = 10$

Caracterizar uma estrutura de solução ótima

Observe que para solucionar o problema original, r_4 , dividimos em subproblemas do mesmo tipo mas menores, para os quais encontramos soluções ótimas que são incorporadas na solução ótima geral. Com o primeiro corte podemos considerar os dois pedaços resultantes como instâncias independentes do problema de Corte de Hastes, e por esses motivos dizemos que o problema tem uma subestrutura ótima.

Implementação recursiva de cima para baixo

Uma vez que determinamos que o problema tem uma subestrutura ótima, e **determinamos um valor ótimo de forma recursiva** a partir da fórmula (1) acima, podemos reescrevê-la pensando em uma implementação recursiva top-down.

Podemos formular a solução sem nenhum corte, dizendo que o primeiro pedaço tem tamanho i = n e receita p_n e que o restante tem tamanho 0 com receita correspondente $r_0 = 0$. Assim obtemos uma versão mais simples da equação (1).

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}).$$

Nesta formulação, uma solução ótima incorpora a solução para apenas um subproblema relacionado - o restante - em vez de dois. O procedimento a seguir implementa a solução proposta pela reescrita da fórmula.

Implementação recursiva de cima para baixo

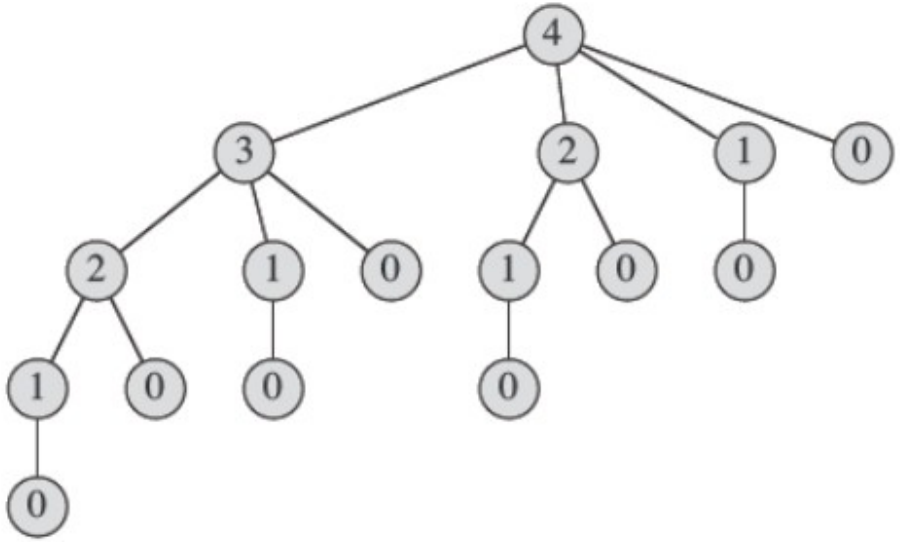
```
algoritmo corte_haste ( p, n )
    se n== 0 então
        retorne ( 0 )
    fim_se
    q = - ∞
    para i = 1 até n
        q = máx( q, p[i] + corte_haste( p, n-1)
    fim_para
    retorne( q )
fim_algoritmo
```

Execução para n=4

```
corte_haste(p, 4)
   q = - ∞
   q = máx(- ∞, 1, corte_haste(p, 3)

   corte_haste(p,3)
      q = - ∞
       i=1 q = máx(- ∞, 1, corte_haste(p, 2)

       corte_haste(p,2)
          q = - ∞
           i=1  q = máx(- ∞, 1, corte_haste(p, 1))

           corte_haste(p,1)
              q = - ∞
              i=1 q = máx(- ∞, 1,corte_haste(p, 0)) =
                      = máx(- ∞, 1, 0) =1
             q = máx(- ∞, 1, 1) = 1

           i=2 q = máx( 1, 5, corte_haste(p,0) ) =
                  = máx( 1, 5, 0) = 5
               q = máx(- ∞, 1, 5) = 5
```

Implementação recursiva de top-down

Por que corte-haste é tão ineficiente? O problema é que corte-haste chama a si mesmo recursivamente várias vezes com os mesmos valores de parâmetros; ele resolve os mesmos subproblemas repetidamente. A Figura a seguir ilustra o que acontece para n = 4.



corte-haste(p, n) chama corte-haste(p, n-i) para i = 1,2,...,n. Equivalentemente, corte-haste(p, n) chama corte-haste(p, j) para cada j=0,1,...,n-1. Quando esse processo se desenrola recursivamente, a quantidade de trabalho feito, como uma função de n, cresce explosivamente.

Para analisar o tempo de execução do CUT-ROD, deixe T(n) denotar o número total de chamadas feitas para CUT-ROD quando chamado com seu segundo parâmetro igual a n. Esta expressão é igual ao número de nós em uma subárvore cuja raiz é rotulada n na árvore de recursão. A contagem inclui a chamada inicial em sua raiz. Assim,

$T(0) = 1$ e

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

Usando Programação Dinâmica

Programação Dinâmica funciona da seguinte maneira. Tendo observado que uma solução recursiva ingênua é ineficiente porque ele soluciona o mesmo subproblema repetidamente, assim, arranjamos para cada subproblema ser solucionado somente uma vez, salvando sua solução. Se for preciso referenciar esta solução do subproblema novamente depois, podemos apenas consultá-la, custando menos que recalculá-la.

A programação dinâmica, portanto, usa memória adicional para economizar tempo de computação; ela serve como um exemplo do *trade-off* entre tempo e memória.

O algoritmo corte-haste-memorizado a seguir apresenta esta solução.

Corte de Haste Memorizado

```
algoritmo corte-haste-memorizado(p,n)
    seja r[0..n] um novo vetor
    para i = 0 até n
        r[i]=-∞
    fim_para
    retorne(corte_haste_memorizado_aux(p, n, r))
fim_algoritmo
```

```
algoritmo corte_haste_memorizado_aux(p, n, r)
    se r[n]>0 então
      retorne(r[n])
    fim_se
    se n==0 então
      q=0
    senão
      q=-∞
      para i = 1 até n
        q=máx(q,p[i]+corte_haste_memorizado_aux(p,n-i,r))
      fim_para
    fim_se
    r[n]=q
    retorne(q)
fim_algoritmo
```

Corte de Haste - Ascendente

Para a abordagem de programação dinâmica ascendente,corte-haste-ascendente usa a ordenação natural dos subproblemas: um problema de tamanho i é “menor” do que um subproblema de tamanho j se i < j . Assim, o procedimento resolve subproblemas de tamanhos j = 0, 1, ... ,n, nessa ordem.

Tanto corte-haste-memorizado quanto corte-haste-ascendente tem complexidade de $O(n^2)$.

Corte de Haste - Ascendente

```
algoritmo corte_hastes_ascendente(p, n)
      seja r [0..n] um novo vetor
      r[0] = 0
      para j = 1 até  n
         q= -∞
         para i = 1 até j
            q = máx( q, p[i] + r[j - ii]
         fim_para
         r[j] = q
      fim_para
      retorne(r[n])
fim_algoritmo
```

Reconstruindo uma solução

Nossas soluções de programação dinâmica para o problema de corte de haste retornam o valor de uma solução ótima, mas não retornam uma solução real: uma lista de tamanhos de peças.

Podemos estender a abordagem de programação dinâmica para registrar não apenas o valor ótimo computado para cada subproblema, mas também uma escolha que levou ao valor ótimo. Com essas informações, podemos imprimir prontamente uma solução ótima, usando o algoritmo corte-haste-solução.

Reconstruindo uma solução

```
algoritmo corte_hastes_ascendente_estendido(p, n)
        seja r [0..n] e s[1..n] novos vetores
        r[0] = 0
        para j = 1 até  n
           q= -∞
           para i = 1 até j
               se q < p[i]+r[ j-i ] então
                  q = p[i] + r[ j-i ]
                  s[j] = i
               fim_se
           fim_para
           r[j] = q
        fim_para
        retorne(r e s)
fim_algoritmo
```

Reconstruindo uma solução

```
algoritmo corte_hastes_solução(p, n)
        (r,s) = corte_haste_ascendente_estendido(p,n)
        enquanto n>0 faça
           escreva s[n]
           n = n - s[n]
        fim_enquanto
fim_algoritmo
```

A Mais Longa Subsequência Comum.

Aplicações biológicas geralmente necessitam da comparação de DNA de dois, ou mais, diferentes organismos. Uma fita de DNA consiste de uma string de moléculas chamadas bases, onde as bases possíveis são Adenina, Guanina, Citosina e Timina. Representando cada uma das bases por sua letra inicial, podemos expressas uma fita de DNA como uma string sobre o conjunto finito {A, C, G, T}. Por exemplo o DNA de um organismo pode ser S1=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA e o DNA de outro S2=GTCGTTCGGAATGCCGTTGCTCTGTAAA. Uma razão para comparar as duas fitas de DNA é determinar o quão similar as duas fitas são, assim como determinar o quanto proximamente relacionados os dois organismos são. Podemos definir similaridade de muitas maneiras diferentes.

A Mais Longa Subsequência Comum.

Por exemplo, podemos dizer que duas fitas são similares quando:
1 – uma string é substring da outra; ou
2 – se o número de mudanças necessárias para transformar uma na outra for pequena; ou ainda;
3 – encontrando uma terceira fita S3, na qual as bases que aparecem em S3 aparecem em S1 e S2 e essas bases devem aparecer na mesma ordem mas não necessariamente consecutivas. Quanto maior a fita S3 maior a similaridade entre S1 e S2. Em nosso exemplo a mais longa fita S3 é GTCGTCGGAAGCCGGCCGAA.

S1 = ACCG**GTCG**AG**T**G**CG**CG**GAAGCCG**G**CCGAA**
S2 =**GTCGT**T**CGGAA**T**GCCG**TT**GC**T**C**T**G**T**AA**A
S3 =**GTCGTCGGAAGCCGGCCGAA**

A Mais Longa Subsequência Comum.

Problema da Mais Longa Subsequencia Comum: Dada uma sequência X = {x_1, x_2, …, x_m}, e outra Z={z_1, z_2, …, z_k} dizemos que Z é subsequência de X se existe uma sequência estritamente crescente (i_1, i_2, …, i_k) de índices de X tal que, para todo j=1, 2, …, k , temos $x_{ij} = z_j$. Por exemplo: Z = { BCDB } é uma subsequencia de X = { ABCBDAB } com correspondência a sequência de índices {2, 3, 5, 7}.

Dadas duas sequências X e Y, dizemos que Z é uma **subsequência comum de** X e Y se Z é uma subsequência de ambas X e Y. Por exemplo, X = { A, B, C, B, D, A, B } e Y = { B, D, C, A, B, A }, a sequência {B, C, A } é uma subsequencia comum de X e Y. Porém {B, C, A} não é a **mais longa subsequência comum** (LCS) de X e Y, pois tem tamanho 3, enquanto a sequência { B, C, B, A } tem tamanho 4. Então {B, C, B, A} é uma LCS de X e Y, assim como a sequência {B, D, A, B}. O problema então consiste em dada duas sequências X={x_1,x_2,…,x_m} e Y = {y_1,y_2,…,y_n} desejamos encontrar a subsequência comum de máximo tamanho de X e Y.

Caracterizando uma subsequencia mais longa

Em uma abordagem de força bruta para resolver o problema LCS, enumeraríamos todas as subsequências de X e verificaríamos cada subsequência para ver se ela também é uma subsequência de Y , mantendo o controle da subsequência mais longa que encontramos. Cada subsequência de X corresponde a um subconjunto dos índices {1, 2,..., m} de X . Como X tem 2^m subsequências, essa abordagem requer tempo exponencial, tornando-a impraticável para sequências longas.

O problema LCS tem uma propriedade de subestrutura ótima como mostra o teorema a seguir. Como veremos, as classes naturais de subproblemas correspondem a pares de “prefixos” das duas sequências de entrada. Para ser preciso, dada uma sequência $X=\{x_1, x_2, \ldots , x_m\}$, definimos o i-ésimo prefixo de X, para i=0, 1,..., m, como $X_i = \{x_1, x_2 , \ldots , x_i\}$. Por exemplo, se X = {A, B, C, B, D, A, B}, então X_4 = {A, B, C, B} e X_0 é a sequência vazia.

Caracterizando uma subsequencia mais longa

Teorema (Subestrutura ótima de uma LCS)

Sejam $X = \{x_1, x_2, \ldots, x_m\}$ e $Y=\{y_1,y_2,\ldots,y_n\}$ sequencias, e seja $Z=\{z_1,z_2,\ldots,z_k\}$ qualquer LCS de X e Y .

1. se $x_m = y_n$, então $z_k = x_m = y_n$ e Z_{k-1} é uma LCS of X_{m-1} e Y_{n-1}.
2. se $x_m \neq y_n$, então $z_k \neq x_m$ implica que Z é uma LCS de X_{m-1} e Y.
3. se $x_m \neq y_n$, então $z_k \neq y_n$ implica que Z é uma LCS de X e Y_{n-1}.

Caracterizando uma subsequencia mais longa

Prova:

(1) Se $Z_k \neq x_m$, então poderíamos anexar $x_m = y_n$ a Z para obter uma subsequência comum de X e Y de comprimento k+1, contradizendo a suposição de que Z é a maior subsequência comum de X e Y . Portanto, devemos ter $z_k = x_m = y_n$. Agora, o prefixo Z_{k-1} é uma subsequência comum de comprimento (k-1) de X_{m-1} e Y_{n-1}. Desejamos mostrar que é um LCS. Suponha, para fins de contradição, que existe uma subsequência comum W de X_{m-1} e Y_{n-1} com comprimento maior que k-1. Então, anexar $x_m = y_n$ a W produz uma subsequência comum de X e Y cujo comprimento é maior que k, o que é uma contradição.

(2) Se $z_k \neq x_m$, então Z é uma subsequência comum de X_{m-1} e Y. Se houvesse uma subsequência comum W de X_{m-1} e Y com comprimento maior que k, então W também seria uma subsequência comum de X_m e Y , contradizendo a suposição de que Z é um LCS de X e Y .

(3) A prova é simétrica a (2).

Uma solução recursiva

O Teorema implica que devemos examinar um ou dois subproblemas ao encontrar uma LCS de $X=\{x_1, x_2, \ldots, x_m\}$ e $Y=\{y_1, y_2, \ldots, y_n\}$.

Se $x_m = y_n$, devemos encontrar uma LCS de X_{m-1} e Y_{n-1}. Acrescentar $x_m = y_n$ a esta LCS produz uma LCS de X e Y.

Se $x_m \neq y_n$, então devemos resolver dois subproblemas:encontrar uma LCS de X_{m-1} e Y e encontrar uma LCS de X e Y_{n-1}.

- Qualquer uma dessas duas LCSs que for maior é uma LCS de X e Y. Como esses casos esgotam todas as possibilidades, sabemos que uma das soluções ótimas de subproblemas deve aparecer dentro de uma LCS de X e Y.

Uma solução recursiva

Assim como no problema de multiplicação de cadeia de matrizes, nossa solução recursiva para o problema LCS envolve estabelecer uma recorrência para o valor de uma solução ótima.
Vamos definir c[i,j] como o comprimento de um LCS das sequências Xi e Yj . Se i = 0 ou j = 0, uma das sequências tem comprimento 0 e, portanto, a LCS tem comprimento 0. A subestrutura ótima do problema LCS fornece a fórmula recursiva

$$c[i,j] = \begin{cases} 0 & \text{se } i=0 \text{ ou } j=0 \\ c[i-1, j-1]+1 & \text{se } i,j > 0 \text{ e } x_i = y_j \\ \text{máx}(c[i,j-1] , c[i-1, j]) & \text{se } i,j > 0 \text{ e } x_i \neq y_j \end{cases}$$

Uma solução recursiva

algoritmo LCS-recursivo(...)

Computando o tamanho da LCS

Com base na recorrência, poderíamos facilmente escrever um algoritmo recursivo de tempo exponencial para calcular o comprimento de um LCS de duas sequências. Como o problema LCS tem apenas Θ(mn) subproblemas distintos, podemos usar programação dinâmica para calcular as soluções de baixo para cima.

O procedimento LCS-LENGTH pega duas sequências $X=\{x_1,x_2,\ldots,x_m\}$ e $Y=\{y_1,y_2,\ldots,y_n\}$ como entradas. Ele armazena os valores c[i.j] em uma tabela c[0..m,0..n], e calcula as entradas em ordem de linha principal. (Ou seja, o procedimento preenche a primeira linha de c da esquerda para a direita, depois a segunda linha e assim por diante.) O procedimento também mantém a tabela b[1..m,1..n] para nos ajudar a construir uma solução ótima. Intuitivamente, b[i,j] aponta para a entrada da tabela correspondente à solução ótima do subproblema escolhida ao calcular c[i,j]. O procedimento retorna as tabelas b e c; c[m,n] contém o comprimento de uma LCS de X e Y.

Computando o tamanho da LCS

```
algoritmo LCS-LENGTH(X,Y)
    m = X.length
    n = Y.length
    sejam b[1..m, 1..n] e c[0..m,0..n] novas tabelas
    para i = 1 até m
        c[i.0] = 0
    para j = 0 até n
        C[0,j] = 0
    para i = 1 até m
        para j = 1 até n
            se xi == yj então
                c[i,j] = c[i-1,j-1] + 1
                b[i.j] = “diagonal”
            senão
                se c[i-1,j] ≥ c[i,j-1] então
                    c[i,j] = c[i-1, j]
                    b[i,j] = “cima”
                senão
                    c[i,j] = c[i,j-1]
                    b[i,j] = “esquerda”
retorne c e b
```

Construindo uma LCS

A tabela b retornada por LCS-LENGTH nos permite construir rapidamente uma LCS de X = {x_1,x_2,…, x_m} e Y = {y_1,y_2 ,…,y_n}. Simplesmente começamos em b[m,n] e traçamos através da tabela seguindo as setas. Sempre que encontramos um “diagonal” na entrada b[i,j], isso implica que xi = yj é um elemento do LCS que LCS-LENGTH encontrou. Com este método encontramos os elementos da LCS em ordem contrária.

Construindo uma LCS

- algoritmo Print-LCS(...)