Programação Dinâmica

Técnicas de Projeto de Algoritmos

Aula 13
Alessandro L. Koerich

Pontifícia Universidade Católica do Paraná (PUCPR)

Ciência da Computação – 7º Período
Engenharia de Computação – 5º Período

Programa do PA

Introdução
- 1. Resolução de Problemas e Tipos de Problemas

Fundamentos da Análise da Eficiência de Algoritmos
- 2. Fundamentos
- 3. Notação Assintótica e Classe de Eficiência
- 4. Análise Matemática de Algoritmos
- 5. Análise Empírica de Algoritmos

Técnicas de Projeto de Algoritmos
- 6. Força Bruta
- 7. Dividir & Conquistar
- 8. Decrementar & Conquistar
- 9. Transformar & Conquistar
- 10. Compromisso Tempo-Espaço
- 11. Programação Dinâmica
- 12. Estratégia Gulosa
- 13. *Backtracking* & *Branch and Bound*
- 14. Algoritmos Aproximados

Limitações
- 15. Teorema do Limite Inferior
- 16. Árvores de Decisão
- 17. Problemas P, NP e NPC

Aulas Anteriores

- Estratégia Força Bruta
- Estratégia Dividir & Conquistar
- Estratégia Reduzir & Conquistar
- Estratégia Transformar & Conquistar
- Estratégia Compromisso Tempo–Espaço
- Estratégia Gulosa

Plano de Aula

- Introdução
- Programação de Linha de Montagem
- Resumo

Introdução

- A **programação dinâmica** se aplica tipicamente a **problemas de otimização** onde uma série de escolhas deve ser feita, a fim de se alcançar um **solução ótima**

Introdução

- Resolve problemas combinando as soluções de subproblemas
- Aplicado quando os **subproblemas** não são independentes, isto é, quando os **subproblemas** compartilham **subsubproblemas**.
- Resolve cada **subsubproblema** somente uma vez e grava a resposta em uma tabela, evitando assim o trabalho de recalcular a resposta toda a vez que o **subsubproblema** é encontrado

Introdução

- Em geral, a programação dinâmica é aplicada em problemas de otimização.
- Problemas de otimização
  - Muitas soluções possíveis;
  - Cada solução tem um valor;
  - Desejamos encontrar uma solução com um valor ótimo (min ou máx).

Introdução

- O desenvolvimento de um algoritmo de programação dinâmica pode ser desmembrado em:
  - Caracterizar a estrutura de uma solução ótima
  - Definir recursivamente o valor de uma solução ótima
  - Calcular o valor de uma solução ótima em um processo *bottom–up*
  - Construir uma solução ótima a partir de informações calculadas

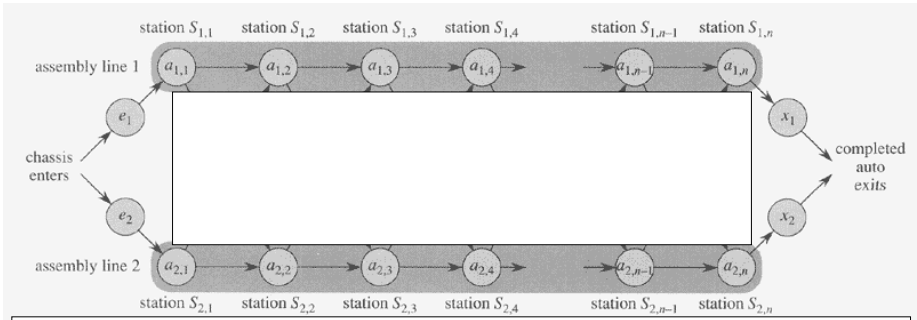
Introdução

- Usaremos a programação dinâmica para resolver alguns problemas de otimização.
- **Primeiro exemplo**: programação de duas linhas de montagem

Programação de Linha de Montagem

- Uma fábrica de automóveis com duas linhas de montagem
  - Um chassis entra em cada linha de montagem
  - Peças são adicionadas a ele em uma série de estações
  - O automóvel sai pronto no final da linha

Programação de Linha de Montagem



- Cada linha tem n estações numeradas com j=1,2,3,...,n
- Indicamos a j–ésima estação na linha i por $S_{i,j}$.
- A j–ésima estação da linha 1 ($S_{1,j}$) executa a mesma função que a j–ésima estação da linha 2 ($S_{1,j}$)

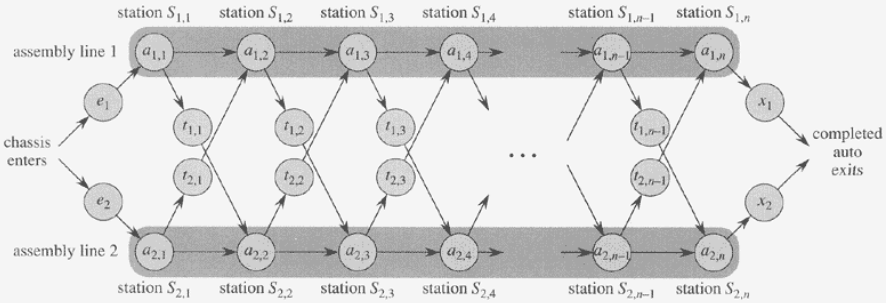
Programação de Linha de Montagem

- Porém, as estações foram construídas em épocas diferentes e com tecnologias diferentes, assim, o tempo exigido em cada estação varia.
- Indicamos o tempo de montagem exigido na estação $S_{i,j}$ por $a_{i,j}$.
- Temos também e_i e x_i como os tempos para um chassis entrar na linha de montagem i e sair concluído da linha de montagem i respectivamente.

Programação de Linha de Montagem

- Normalmente, um chassis entra e sai de uma mesma linha de montagem.
- Porém, no caso de um pedido urgente, um automóvel parcialmente concluído pode ser passado de uma linha de montagem a outra.

Programação de Linha de Montagem



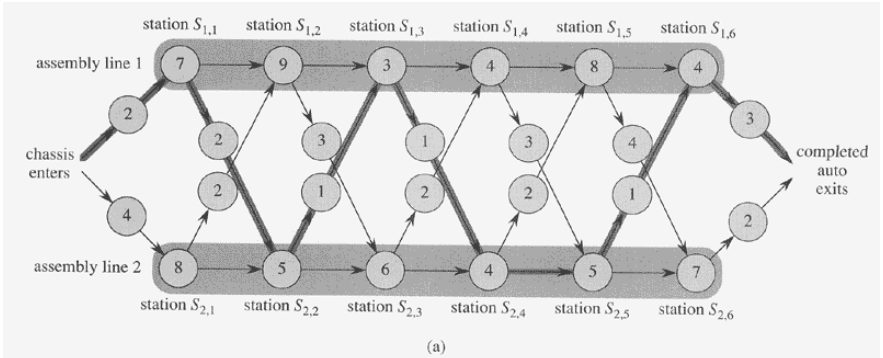
- O tempo para transferir um chassi da linha de montagem i depois da passagem pela estação $S_{i,j}$ é $t_{i,j}$, onde i =1,2 e j=1,2,...,n−1

Programação de Linha de Montagem

- Problema

  Determinar que estações escolher na linha 1 e quais escolher na linha 2 de modo a minimizar o tempo total de passagem de um único automóvel pela fábrica.

Programação de Linha de Montagem



- O tempo total mais rápido resulta da escolha das estações 1, 3 e 6 da linha 1 e das estações 2, 4 e 5 da linha 2.

Programação de Linha de Montagem

Como resolver o problema?

- Força Bruta: enumerar todos os modos possíveis e calcular quanto tempo cada um deles demora.
- Existem 2^n maneiras possíveis de escolher estações → $\Omega(2^n)$ → impraticável para n grande
- Solução possível → programação dinâmica

Programação de Linha de Montagem

- **Etapa 1**: A estrutura do caminho mais rápido pela fábrica
- Caracterizar a estrutura de uma solução ótima
- Considerar o modo mais rápido possível para um chassis seguir desde o ponto de partida passando pela estação $S_{1,j}$.
  - Se $j = 1$, fácil: determinar somente quanto tempo demora para passar pela estação $S_{1,j}$
  - Se $j \geq 2$, há duas opções para obter $S_{1,j}$:
    - Através de $S_{1,j-1}$, e depois diretamente para $S_{1,j}$
    - Através de $S_{2,j-1}$, e depois transferido para $S_{1,j}$

Programação de Linha de Montagem

Supondo que o caminho mais rápido é através de $S_{1,j-1}$

- Observação chave: devemos ter pego um caminho mais rápido a partir da entrada através de $S_{1,j-1}$ nesta solução.
- Se houvesse um caminho mais rápido através de $S_{1,j-1}$, nós o usaríamos para obter um caminho mais rápido através de $S_{1,j}$

Programação de Linha de Montagem

Supondo agora que o caminho mais rápido é através de $S_{2,j-1}$

- Observação chave: Novamente, devemos ter pego um caminho mais rápido a partir da entrada através de $S_{2,j-1}$ nesta solução.
- Se houvesse um caminho mais rápido através de $S_{2,j-1}$, nós o usaríamos para obter um caminho mais rápido através de $S_{1,j}$

Programação de Linha de Montagem

Geralmente: Uma solução ótima para um problema (o caminho mais rápido através $S_{1,j}$) contém dentro dele um solução ótima para subproblemas (o caminho mais rápido através $S_{1,j-1}$ ou $S_{2,j-1}$)

Isto é → uma subestrutura ótima.

Programação de Linha de Montagem

Usar subestruturas ótimas para construir soluções ótimas para o problema a partir de soluções ótimas para subproblemas

O caminho mais rápido através de $S_{1,j}$ é tanto:

- Caminho mais rápido através de $S_{1,j-1}$, e depois diretamente através de $S_{1,j}$ ou
- Caminho mais rápido através de $S_{2,j-1}$, transferência da linha 2 para linha 1, e depois através $S_{1,j}$

Programação de Linha de Montagem

Simetricamente. . .

O caminho mais rápido através de $S_{2,j}$ é tanto:

- Caminho mais rápido através de $S_{2,j-1}$, e depois diretamente através de $S_{2,j}$ ou
- Caminho mais rápido através de $S_{1,j-1}$, transferência da linha 1 para linha 2, e depois através $S_{2,j}$

Programação de Linha de Montagem

Portanto, para resolver problemas de encontrar um caminho mais rápido através de $S_{1,j}$ e $S_{2,j}$, resolver os subproblemas de encontrar um caminho mais rápido através de $S_{1,j-1}$ e $S_{2,j-1}$.

Programação de Linha de Montagem

- **Etapa 2**: Solução Recursiva
- Definir recursivamente o valor de uma solução ótima em termos das soluções ótimas dos subproblemas
- Subproblemas: encontrar o caminho mais rápido pela estação j em ambas as linhas, para $j = 1, 2, \ldots, n$.

Programação de Linha de Montagem

- Seja $f_i[j]$ = o tempo mais rápido possível para levar um chassi desde o ponto de partida até a estação $S_{i,j}$, onde $i = 1, 2$ e $j = 1, 2, \ldots, n$.
- Meta: f^* = tempo mais rápido para levar um chassi por todo o percurso na fábrica.

$$f^* = min\ (\ f_1[n]+x_1, f_2[n]+x_2\ )$$

onde $f_1[1]=e_1 + a_{1,1}$ e $f_2[1]=e_2 + a_{2,1}$

Programação de Linha de Montagem

- Para $j = 2, \ldots, n$:

$$f_1[1] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j})$$

$$f_2[1] = \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j})$$

Programação de Linha de Montagem

- Combinando as equações anteriores, obtemos as equações recursivas:

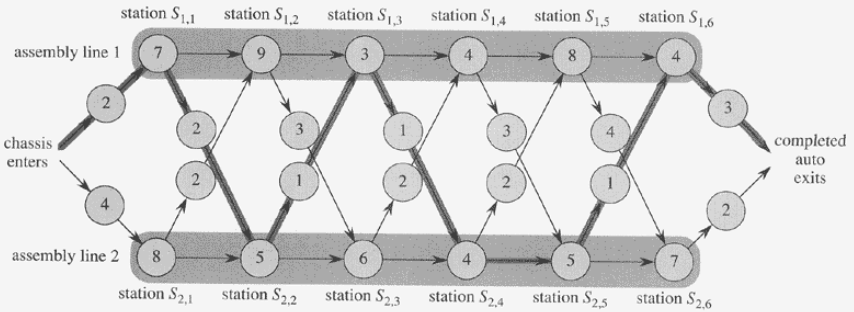
$$f_1[j] = \begin{cases} e_1 + a_{1,1} & se\ j = 1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & se\ j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & se\ j = 1 \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & se\ j \geq 2 \end{cases}$$

Programação de Linha de Montagem

- $f_i[j]$ fornece o valor de uma solução ótima. E se quisermos construir uma solução ótima?
- Definimos $l_i[j]$ = # linha (1 ou 2) cuja estação $j-1$ é usada em um caminho mais rápido pela estação $S_{i,j}$. Onde $i = 1, 2$ e $j = 2, 3, \ldots, n$
- l^* = # linha cuja estação n é usada em um caminho mais rápido pela fábrica inteira.

Programação de Linha de Montagem



j	1	2	3	4	5	6	
$f_1[j]$	9	18	20	24	32	35	$f^* = 38$
$f_2[j]$	12	16	22	25	30	37	

j	2	3	4	5	6	
$l_1[j]$	1	2	1	1	2	$l^* = 1$
$l_2[j]$	1	2	1	2	2	

(b)

Figure 15.2 **(a)** An instance of the assembly-line problem with costs e_i, $a_{i,j}$, $t_{i,j}$, and x_i indicated. The heavily shaded path indicates the fastest way through the factory. **(b)** The values of $f_i[j]$, f^*, $l_i[j]$, and l^* for the instance in part (a).

Programação de Linha de Montagem

- Vamos através do caminho ótimo dado pelos valores de l (linhas sombreadas na figura anterior).

Programação de Linha de Montagem

- **Etapa 3**: Cálculo dos Tempos mais Rápidos (Computar uma solução ótima)
- Poderíamos somente escrever um algoritmo recursivo baseado nas recorrências anteriores.
- Porém, seu tempo de execução é exponencial em n.

Programação de Linha de Montagem

- Seja $r_i(j)$ o número de referências feitas a $f_i[j]$ em um algoritmo recursivo
- A partir da primeira equação temos:

$$r_1(n) = r_2(n) = 1$$

- Pelas recorrências temos:

$$r_1(j) = r_2(j) = r_1(j+1)+r_2(j+1)$$

para $j = 1, 2, \ldots, n-1$

Programação de Linha de Montagem

- Assim, $r_i(j) = 2^{n-j}$
- Prova: Indução sobre j, decrescente a partir de n
- Base: $j = n$, $2^{n-j} = 2^0 = 1 = r_i(n)$
- Passo de indução: Assumir $r_i(j+1) = 2^{n-(j+1)}$

  Então, $r_i(j) = r_i(j+1) + r_2(j+1)$

$= 2^{n-(j+1)} + 2^{n-(j+1)}$

$= 2^{n-(j+1)+1}$

$= 2^{n-j} \quad \rightarrow \Theta(2^n)$

Programação de Linha de Montagem

- Portanto, $f_1[1]$ sozinho é referenciado 2^{n-1} vezes
- Portanto, *top–down* não é uma boa maneira de computar $f_i[j]$.
- Observação: Podemos fazer melhor. $f_i[j]$ depende somente de $f_1[j-1]$ e $f_2[j-1]$ para $j \geq 2$.
- Portanto, a computação deve ser feita em ordem crescente de $j \rightarrow \Theta(n)$

Programação de Linha de Montagem

- O procedimento *Fast–Way* toma como entrada os valores ($a_{i,j}$, $t_{i,j}$, e_i e x_i), bem como n, o número de estações em cada linha de montagem.

Programação de Linha de Montagem

```
FASTEST-WAY(a, t, e, x, n)
 1  f1[1] ← e1 + a1,1
 2  f2[1] ← e2 + a2,1
 3  for j ← 2 to n
 4      do if f1[j − 1] + a1,j ≤ f2[j − 1] + t2,j−1 + a1,j
 5            then f1[j] ← f1[j − 1] + a1,j
 6                 l1[j] ← 1
 7            else f1[j] ← f2[j − 1] + t2,j−1 + a1,j
 8                 l1[j] ← 2
 9         if f2[j − 1] + a2,j ≤ f1[j − 1] + t1,j−1 + a2,j
10            then f2[j] ← f2[j − 1] + a2,j
11                 l2[j] ← 2
12            else f2[j] ← f1[j − 1] + t1,j−1 + a2,j
13                 l2[j] ← 1
14  if f1[n] + x1 ≤ f2[n] + x2
15     then f* = f1[n] + x1
16          l* = 1
17     else f* = f2[n] + x2
18          l* = 2
```


Programação de Linha de Montagem

- **<u>Etapa 4</u>**: Construção do Caminho mais Rápido pela Fábrica (Construindo uma solução ótima)
- Após calculados $f_i[j]$, f^*, $l_i[j]$ e l^*, podemos construir a seqüência de estações usadas no caminho mais rápido pela fábrica.
- Procedimento *Print–Stations*

Programação de Linha de Montagem

```
PRINT-STATIONS(l, n)
1  i ← l*
2  print "line " i ", station " n
3  for j ← n downto 2
4      do i ← li[j]
5         print "line " i ", station " j − 1
```


Programação de Linha de Montagem

- Procedimento *Print–Stations*
- No exemplo da figura, teríamos:
  - Linha 1: estação 6
  - Linha 2: estação 5
  - Linha 2: estação 4
  - Linha 1: estação 3
  - Linha 2: estação 2
  - Linha 1: estação 1

Resumo

- Características da Programação Dinâmica:
  - O problema precisa ter a propriedade da subestrutura ótima
  - Então começamos com uma solução recursiva, mas ela será inviável
  - Com isso, a transformamos em uma solução iterativa, que irá ter tempo polinomial, com a característica de calcular primeiro o valor de uma solução ótima e só depois construir a solução.

Elementos da Programação Dinâmica:
Subproblemas Sobrepostos

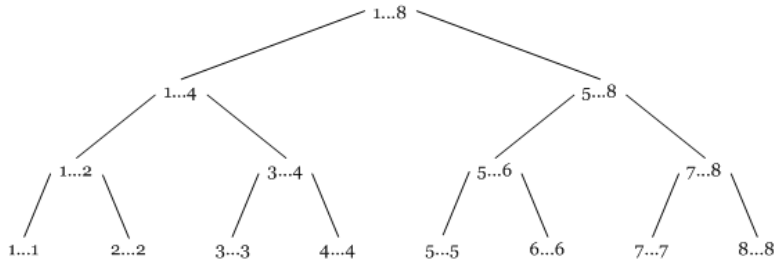
Subproblemas sobrepostos

- O segundo ingrediente que um problema de otimização deve ter para a programação dinâmica ser aplicável.
- O espaço de subproblemas deve ser pequeno .

Elementos da Programação Dinâmica:
Subproblemas Sobrepostos

Subproblemas sobrepostos

- ocorrem quando um algoritmo recursivo revisita o mesmo problema repetidamente
- Bons algoritmos "*dividir e conquistar*" geralmente geram um novo problema em cada estágio da recursão.
- Exemplo: Merge–Sort



Elementos da Programação Dinâmica:
Subproblemas Sobrepostos

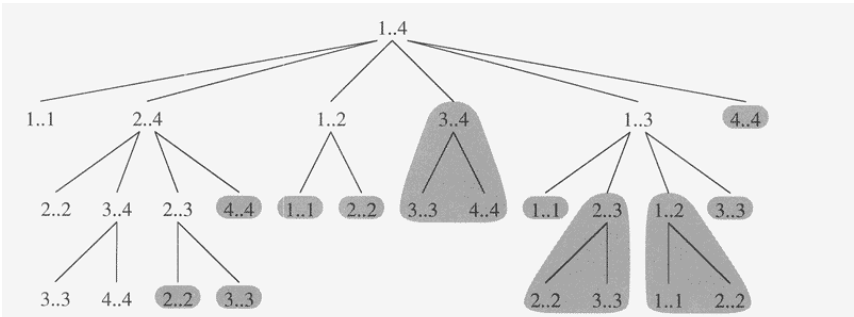


Figure 15.5 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN(p, 1, 4). Each node contains the parameters i and j. The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN(p, 1, 4).

- Recursivo: $\Omega(2^n)$
- Programação Dinâmica: $O(n^2)$

Elementos da Programação Dinâmica: Memoização

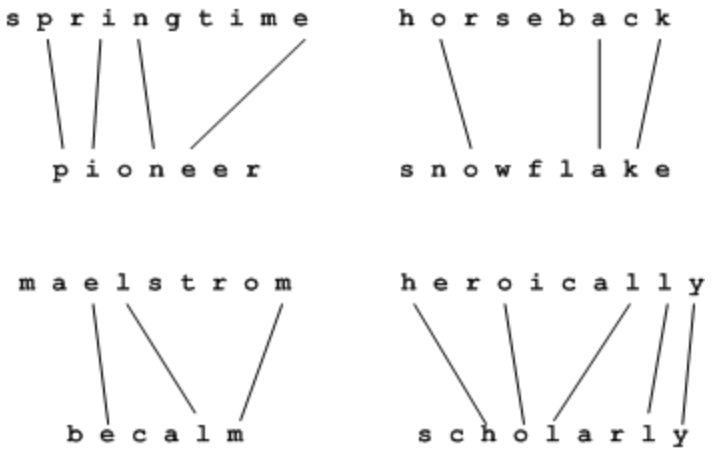
Método Alternativo: *Memoization*

- "Armazene, não recompute"
- Faça uma tabela indexada por subproblemas
- Quando resolver um subproblema:
  - Buscar na tabela
  - Se a resposta for sim, use–a
  - Senão, compute a resposta e armazene–a
- Em programação dinâmica, vamos um passo adiante. Determinamos em que ordem queremos acessar a tabela e preenchemos desta maneira.

Subsequência Comum Mais Longa

- Problema: Dadas duas sequências, $X=<x_1, x_2, ..., x_m>$ e $Y=<y_1,y_2,...y_n>$, encontrar a subsequência comum a ambas cujo comprimento seja o mais longo.
- Uma subsequência não precisa ser consecutiva (contínua), mas ela deve estar em ordem.
- O problema da LCS pode se resolvido por força bruta ou programação dinâmica.

Subsequência Comum Mais Longa



Subsequência Comum Mais Longa

Algoritmo "Força–Bruta"

- Para cada subseqüência de X, verificar se é uma subseqüência de Y.
- Tempo: Θ ($n\ 2^m$)
  - 2^m subseqüências de X para verificar
  - Cada subseqüência leva Θ (n) para verificar. Varrer Y para a primeira letra, a partir dela, varrer pela segunda letra, e assim por diante.

Subsequência Comum Mais Longa

Etapa 1: Caracterização de uma Subseqüência Comum mais Longa

- Algoritmo "Força–Bruta": Para cada subseqüência de X, verificar se é uma subseqüência de Y.
- Cada subseqüência de X corresponde a um subconjunto dos índices $\{1,2,\ldots, m\}$ de X. Existem 2^m subseqüências de X.
- Tempo: $\Theta(n\,2^m)$
  - 2^m subseqüências de X para verificar
  - Cada subseqüência leva $\Theta(n)$ para verificar. Varrer Y para a primeira letra, a partir dela, varrer pela segunda letra, e assim por diante.

Subsequência Comum Mais Longa

Etapa 1 (cont.)

- Porém o problema da LCS tem uma propriedade de subestrutura ótima.
- Dada uma seqüência $X = < x_1,x_2,\ldots,x_m >$, definimos o *i–ésimo* prefixo de X, para $i=0,1,\ldots,m$ como $X_i=< x_1,x_2,\ldots,x_i >$.
- Ex: Se $X = < A, B, C, B, D, A, B>$, então $X_4 = < A, B, C, B >$ e X_0 é a seqüência vazia.

Subsequência Comum Mais Longa

Etapa 1 (cont.)

Sejam as seqüências $X = < x_1,x_2,\ldots,x_m >$ e $Y = < y_1,y_2,\ldots,y_m >$ e seja $Z = < z_1,z_2,\ldots,z_m >$ qualquer LCS de X e Y.

Notação:

X_i = prefixo $<x_1,\ldots,x_i>$
Y_i = prefixo $<y_1,\ldots,y_i>$

Teorema:

Seja $Z = <z_1,\ldots,z_k>$ qualquer LCS de X e Y.

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

Subsequência Comum Mais Longa

Etapa 1 (cont.)

Prova:

1. Primeira mostrar que $z_k=x_m=y_n$. Suponha que não. Então, faça uma subseqüência $Z'= <z_1,\ldots,z_k, x_m>$. É uma subseqüência comum de X e Y e tem comprimento $k + 1 \Rightarrow Z'$ é uma subseqüência comum mais longa que $Z \Rightarrow$ contradiz Z sendo uma LCS.

   Agora mostrar que Z_{k+1} é uma LCS de X_{m-1} e Y_{n-1}. Claramente, é uma subseqüência comum. Agora suponha que existe uma subseqüência comum W de X_{m-1} e Y_{n-1} que é mais longa que $Z_{k+1} \Rightarrow$ comprimento de $W \geq k$. Faça a subseqüência W' anexando x_m a W. W' é uma subseqüência comum de X e Y, tem comprimento $\geq k+1$ $\Rightarrow$ contradiz Z sendo uma LCS.

Subsequência Comum Mais Longa

Etapa 1 (cont.)

Prova (cont.):

2. Se $z_k \neq x_m$, então Z é uma subseqüência comum de X_{m-1} e Y. Suponha que exista uma subseqüência W de X_{m-1} e Y com comprimento > k. Então W é uma subseqüência comum de X e $Y \Rightarrow$ contradiz Z sendo uma LCS.

3. Simétrica a 2.

Portanto, uma LCS de duas seqüências contém como um prefixo uma LCS de prefixos das seqüências.

Subsequência Comum Mais Longa

Etapa 2 : Formulação Recursiva

- Do teorema anterior, temos que existem 1 ou 2 subproblemas a examinar quando se encontra uma LCS de $X = < x_1,x_2,...,x_m >$ e $Y = < y_1,y_2,...,y_m >$
  - Se $x_m=y_n$, devemos encontrar uma LCS de X_{m-1} e Y_{n-1}.
  - Se $x_m \neq y_n$, devemos resolver 2 subproblemas:
    - Encontrar uma LCS de X_{m-1} e Y.
    - Encontrar uma LCS de X e Y_{n-1}.

Subsequência Comum Mais Longa

Etapa 2 (cont.): Formulação Recursiva

- A solução recursiva para o problema da LCS envolve estabelecer uma recorrência para o valor de uma solução ótima.
- Definindo $c[\,i,j\,]$ = comprimento da LCS de X_i e Y_j. Se i=0 ou j=0, uma das seqüencias tem comprimento 0, logo LCS = 0.
- A subestrutura ótima do problema da LCS fornece a fórmula recursiva:

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

.

Subsequência Comum Mais Longa

Etapa 3: Calculando o Comprimento da LCS

- O procedimento *LCS–LENGTH* toma duas seqüencias X e Y como entradas.
- Armazena os valores de $c[i,j]$ em uma tabela $c[0...m,0...n]$.
- Mantém uma tabela $b[1...m,1...n]$ para construir a solução ótima. $b[i,j]$ aponta para a entrada da tabela correspondente à solução ótima do subproblema escolhida ao se calcular $c[i,j]$.

Subsequência Comum Mais Longa

Etapa 3(cont.): Cálculo do comprimento da solução ótima

```
LCS-LENGTH(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m
 4       do c[i, 0] ← 0
 5  for j ← 0 to n
 6       do c[0, j] ← 0
 7  for i ← 1 to m
 8       do for j ← 1 to n
 9            do if x_i = y_j
10                 then c[i, j] ← c[i − 1, j − 1] + 1
11                      b[i, j] ← "↖"
12                 else if c[i − 1, j] ≥ c[i, j − 1]
13                      then c[i, j] ← c[i − 1, j]
14                           b[i, j] ← "↑"
15                      else c[i, j] ← c[i, j − 1]
16                           b[i, j] ← "←"
17  return c and b
```


Subsequência Comum Mais Longa

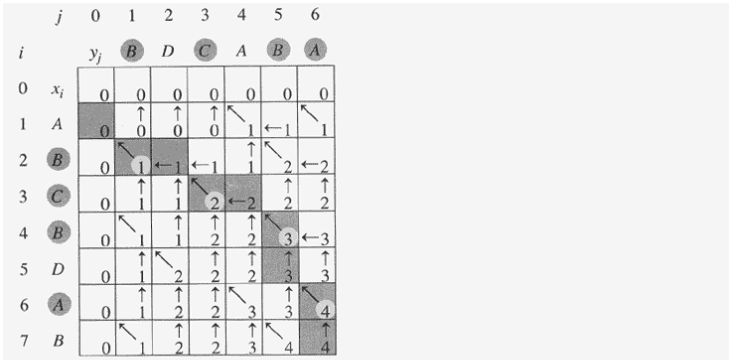


Figure 15.6 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i - 1, j]$, $c[i, j - 1]$, and $c[i - 1, j - 1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the path is shaded. Each "↖" on the path corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

Subsequência Comum Mais Longa

Etapa 4: Cálculo do comprimento da solução ótima

- A chamada inicial é PRINT–LCS (*b*,*X*,*m*,*n*)
- $b[i,j]$ aponta para a entrada da tabela cujo subproblema usamos para resolver LCS de X_i e Y_j.
- Quando $b[i,j]$ = ↖, estendemos LCS em um caractere. Então a subseqüência comum mais longa = entradas contendo ↖.

Subsequência Comum Mais Longa

Construção de uma LCS

```
PRINT-LCS(b, X, i, j)
1  if i = 0 or j = 0
2     then return
3  if b[i, j] = "↖"
4     then PRINT-LCS(b, X, i − 1, j − 1)
5          print x_i
6  elseif b[i, j] = "↑"
7     then PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

Resumo

Características da Programação Dinâmica:

- O problema precisa ter a propriedade da subestrutura ótima
- Então começamos com uma solução recursiva, mas ela será inviável
- Com isso, a transformamos em uma solução interativa, que irá ter tempo polinomial, com a característica de calcular primeiro o valor de uma solução ótima e só depois construir a solução.