Análise e Técnicas de Algoritmos

Jorge Figueiredo

Programação Dinâmica

Motivação

Números de Fibonacci
Entrada: Um número inteiro n.
Saída: O número de Fibonacci F_n, definido da seguinte forma:
$F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2}$ para $n \geq 2$.

Solução clássica utiliza recursão

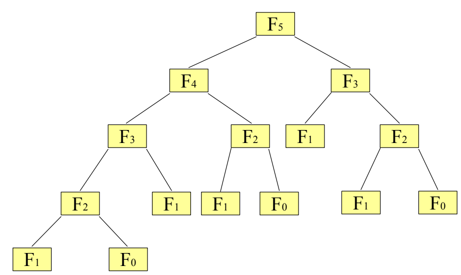
```
Fib(n)
if n ≤ 1 then return n
return Fib(n – 1) + Fib(n - 2)
```


Sobre a Solução Apresentada

- Sabemos provar a corretude do algoritmo.
- Análise através da resolução de uma relação de recorrência:
  - $T(n) = T(n – 1) + T(n – 2) + c$
  - $O(2^n)$
- Solução ineficiente.

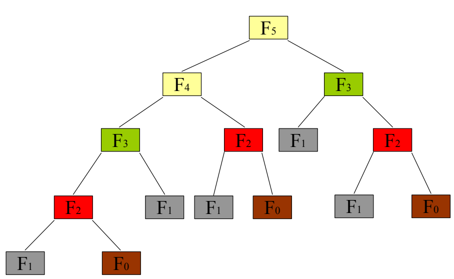
Sobre a Solução Apresentada

- Qual o motivo da ineficiência?



Sobre a Solução Apresentada

- Cálculo repetido, desnecessário!!!

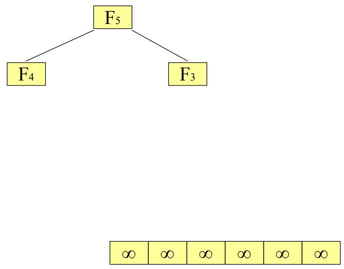


Ainda Sobre Fibonacci

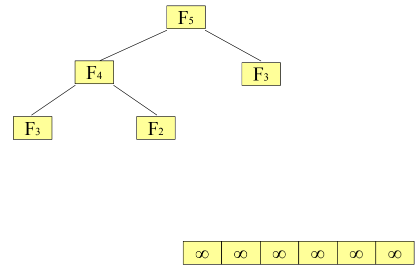
- Solução alternativa:
  - Utilizar um array $f[0, ..., n]$ para guardar os valores calculados.
  - Inicialmente, f contém apenas símbolos especiais ∞.

```
Fib1(n)
if f[n] ≠ ∞ then return f[n]
if n ≤ 1 then return f[n] ← n
return f[n] ← Fib1(n – 1) + Fib1(n - 2)
```

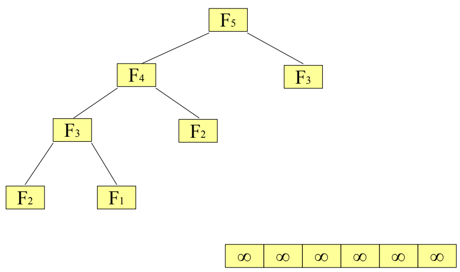
Ainda Sobre Fibonacci



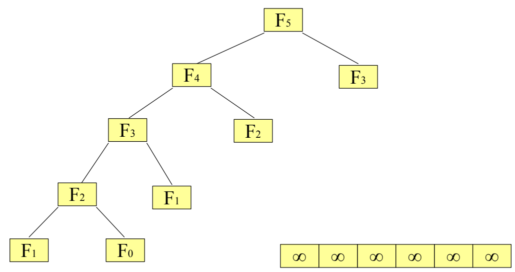
Ainda Sobre Fibonacci



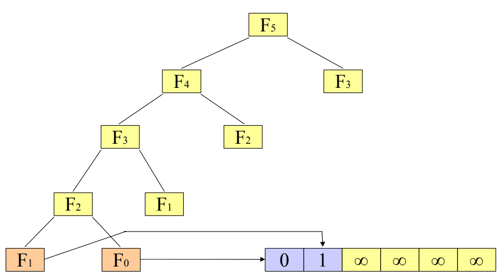
Ainda Sobre Fibonacci



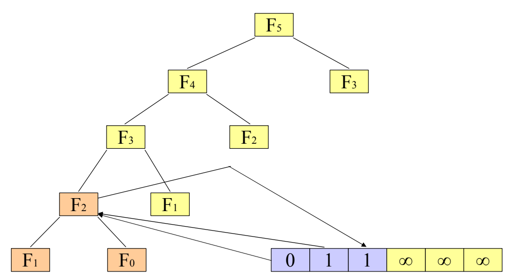
Ainda Sobre Fibonacci



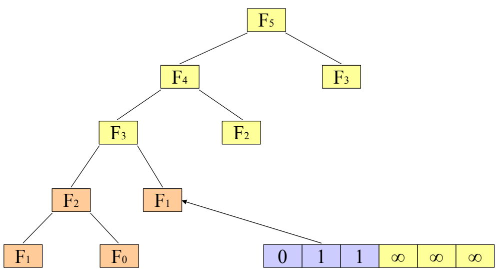
Ainda Sobre Fibonacci



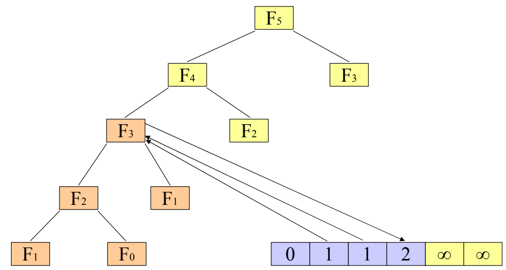
Ainda Sobre Fibonacci



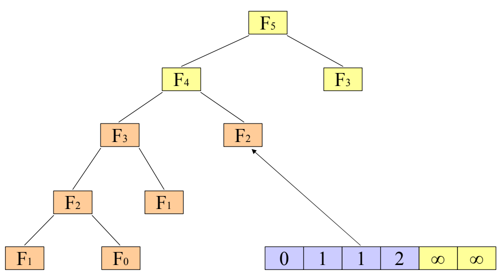
Ainda Sobre Fibonacci



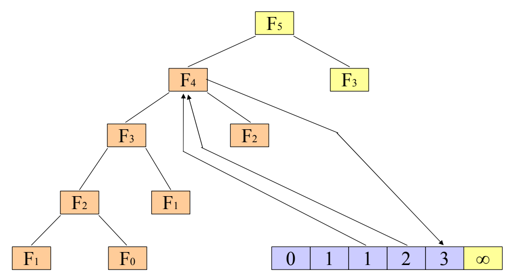
Ainda Sobre Fibonacci



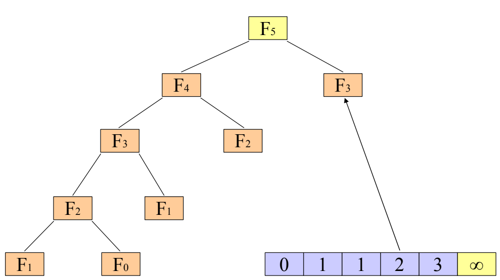
Ainda Sobre Fibonacci



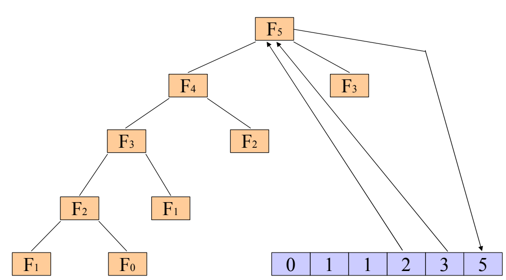
Ainda Sobre Fibonacci



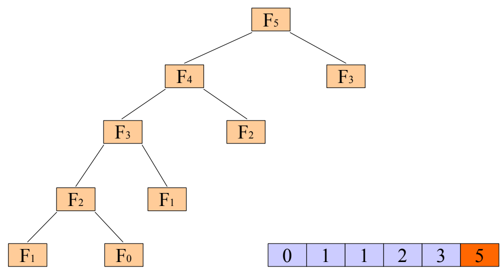
Ainda Sobre Fibonacci



Ainda Sobre Fibonacci



Ainda Sobre Fibonacci



Ainda Sobre Fibonacci

- Uma outra solução alternativa:
  – Eliminar as chamadas recursivas.
  – Utilizar o array para armazenar dados calculados.
  – Estratégia bottom-up.

```
Fib2(n)
f[0] ← 0
f[1] ← 1
for i ← 2 to n do
      f[i] ← f[i – 1] + f[i - 2]
return f[n]
```

Análise das Soluções Alternativas

- É fácil identificar que **Fib2** é O(n).
- **Fib1** é também O(n). Tratar pilha de recursão.
- Abordagem utilizada:
  – *Encontrar função recursiva apropriada.*
  – *Adicionar memorização para armazenar resultados de subproblemas.*
  – *Determinar uma versão bottom-up, iterativa.*

Fib2 é PROGRAMAÇÃO DINÂMICA!!!!!

Programação Dinâmica

- Aplicado quando recursão produz repetição dos mesmos subproblemas.
- Proposta: reusar computação.
- PD = DC + tabela.
- Versão bottom-up é mais compacta e fácil de efetuar análise.
- Estratégia utilizada em problemas de otimização.

Exemplo: Número de Combinações

Número de Combinações
Entrada: Dois números inteiros *n* e *r*, em que *n* indica o número de elementos dos quais tenho que escolher *r*.
Saída: O número possível de combinações de *r* itens.

Algoritmo Baseado em Divisão e Conquista

Para escolher *r* itens de *n*, podemos proceder de duas formas:

- Escolher o primeiro item. Depois escolher *r-1* itens dos *n-1* itens restantes.
- Não escolher o primeiro item. Escolher, então, *r* itens dos *n-1* itens restantes.

```
Escolha(r, n)
if r = 0 ou n = r then
      return 1
else
      return Escolha(r-1, n-1) + Escolha(r, n-1)
```

Análise do Algoritmo DeC

- A análise do algoritmo requer a resolução da seguinte relação de recorrência:
  - $T(n) = 2.T(n-1) + c$.
- $T(n) = O(2^n)$.
- Da mesma forma que no exemplo de Fibonacci, essa solução faz cálculos repetidos.
- Solução: Programação Dinâmica.

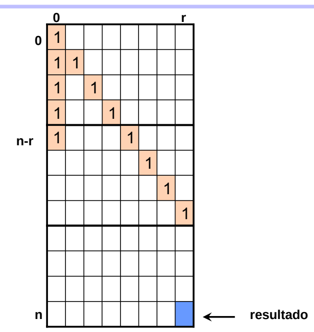
Algoritmo Utilizando Programação Dinâmica

```
Escolha(r, n)
for i ← 0 to n-r do
    T[i, 0] ← 1
for i ← 0 to r do
    T[i, i] ← 1
for j ← 1 to r do
    for i ← j+1 to n-r+j do
        T[i, j] ← T[i-1, j-1] + T[i-1, j]
return T[n, r]
```

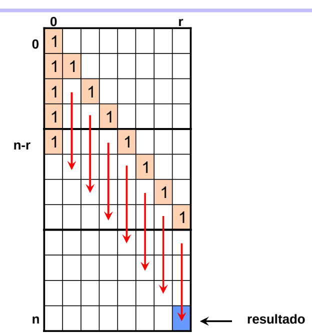

Considerações sobre o Algoritmo PD

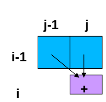
- O(n.r)
- Duas partes:
  - Primeira parte relacionada com o caso base: inicialização da tabela.
  - Segunda parte define como o restante da tabela deve ser preenchida.
- Tabela:
  - T[n, r].
  - O valor armazenado na célula T[i, j] indica o número possível de combinações de escolher j itens dentre i itens.

Tabela Após o Preenchimento Inicial



Caminhamento e Padrão de Preenchimento





Caracterização de PD

- Quando a estratégia de DeC gera um número grande de problemas idênticos, recursão se torna muito caro.
- Melhor armazenar as soluções parciais em uma tabela.
- Como transformar DeC em PD:
  - A parte do algoritmo que corresponde a conquista (recursão) deve ser substituída por olhada na tabela.
  - Em vez de retornar um valor, armazená-lo na tabela.
  - Caso base para iniciar a tabela.
  - Determinar padrão de preenchimento do restante da tabela.

Quando Aplicar Programação Dinâmica

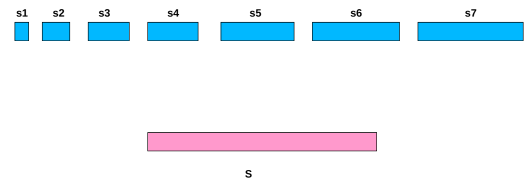
- Aplicar em problemas que, em princípio, parece requerer muito tempo para ser resolvido (em geral é de ordem exponencial).
- Principais características:

  Princípio da Otimalidade (subproblemas ótimos): o valor ótimo global pode ser definido em termos dos valores ótimos dos subproblemas.

  Overlap de Subproblemas: os subproblemas não são independentes. Existe um overlap entre eles (logo, devem ser construídos bottom-up).

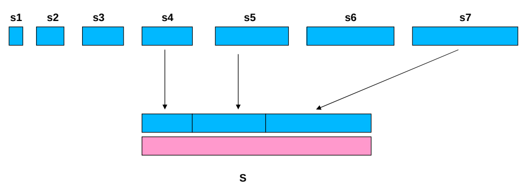
Exemplo: Mochila Binária

Considere ***n*** itens de tamanhos ***s_1, s_2, ..., s_n***. Existe um subconjunto destes itens cuja soma total é exatamente ***S***?



Exemplo: Mochila Binária

Considere ***n*** itens de tamanhos ***s_1, s_2, ..., s_n***. Existe um subconjunto destes itens cuja soma total é exatamente ***S***?



Exemplo: Mochila Binária

- Podemos generalizar para situações em que temos i itens e o tamanho da mochila é j.
- Para saber se retornamos verdadeiro, temos que investigar duas possibilidades:
  – O i-ésimo item é usado para completar o tamanho j.
  – O i-ésimo item não é utilizado para completar o tamanho j. j é alcançado com i-1 itens.
- Solução:
  – Utilizar uma tabela T[n, S] para armazenar TRUE se é possível completar exatamente S com n primeiros elementos.
  – $T[i, j] = T[i-1, j - s_i]$ ou $T[i-1, j]$

Algoritmo DeC: Mochila Binária

```
Mochila(i, j)
if i = 0  then
      return (j=0)
else
      if Mochila(i-1, j) then
            return true
      else
            if s_i ≤ j then
                  return  Mochila(i-1, j – s_i)
```

Algoritmo PD: Mochila Binária

```
Mochila(n, S)
T[0, 0] = true
for j ← 1  to S do
      T[0, j] ← false
for i ← 1 to n do
      for j ← 0 to S do
            T[i, j] ← T[i-1, j]
            if j – s_i ≥ 0 then
                  T[i, j] ← T[i, j] ∨ T[i-1, j – s_i]
return  T[n, S]
```

Programação Dinâmica

- Mais eficiente do que o método da força bruta, quando existe *overlap* de subproblemas.
- Divisão-e-Conquista + memória.
- Características:
  - Subestrutura ótima.
  - Tabela.
  - Bottom-up.

Maior Subseqüência Comum (LCS)

Maior Subseqüência Comum

X= {A B C B D A B }, Y= {B D C A B A}

Maior Subseqüência comum é:

X = A B C B D A B

Y = B D C A B A

Solução para LCS

- Solução força bruta: comparar cada subseqüência de X com os símbolos de Y.
- Se |X| = m, |Y| = n: 2^m subseqüências de X
- Solução força bruta é $O(n\,2^m)$
- LCS exibe *subestrutura ótima*: soluções de subproblemas fazem parte da solução final.
- Existe melhor idéia?

A solução usando PD

- Achar LCS para prefixos de X e Y
  - Sejam X_i, Y_j prefixos de X e Y de tamanhos *i* e *j* respectivamente
- *c[i,j]* é o tamanho da LCS de X_i e Y_j
- Logo, LCS de X e Y vai ser guardado em *c[m,n]*
- Como definir uma solução recursiva para *c[i,j]*?

Solução Recursiva

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- Caso base: *i = j = 0* (substrings vazios de X e Y)
- Se X_0 e/ou Y_0 são strings vazios: para todo *i* e *j*: *c[0, j] = c[i,0] = 0*

Solução Recursiva

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

Primeiro Caso:

- *x[i]=y[j]*: mais um símbolo em X e Y confere.
- Logo, LCS para X_i e Y_j é igual ao LCS de X_{i-1} e Y_{i-1}, mais 1.

Solução Recursiva

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(\, c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

Segundo caso:

- *x[i] ≠ y[j]*
- Se os símbolos não casam, não podemos melhorar a nossa resposta: (i.e. max(LCS(X_i, Y_{j-1}) e LCS(X_{i-1},Y_j)).

Por que não repetir LCS(X_{i-1}, Y_{j-1}) ?

O Algoritmo

```
LCS(X, Y)
m ← length(X)
n ← length(Y)
for i ← 1  to m do
      c[i, 0] ← 0
for j ← 1 to n do
      c[0, j] ← 0
for i ← 1 to m do
      for j ← 1 to n do
            if (X_i == Y_j) then
                  c[i, j] ← c[i-1, j-1] + 1
            else
                  c[i, j]← max(c[i-1, j], c[i, j-1])
return  c[m, n]
```

Exemplo

- Considere:
  - X = ABCB
  - Y = BDCAB

Determine o tamanho da LCS de X e Y.

Exemplo

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5

ABCB
BDCAB

Exemplo: caso base

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 1 to m   c[i,0] = 0
for j = 1 to n   c[0,j] = 0

ABCB
BDCA

Exemplo: primeiro valor

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**				
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**				
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1			
4	B	0					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2		
4	B	0					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1				

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

Análise do Algoritmo LCS

- Qual o tempo de execução?

O(m*n)

cada c[i,j] é calculado em tempo constante, e existem m*n células

Como encontrar a LCS

- Mesmo esquema usado nos problemas da mochila e distância de edição mínima
- Cada *c[i,j]* depende de *c[i-1,j]*, *c[i,j-1]* e *c[i-1, j-1]*.
- Podemos identificar como cada c[i,j] foi obtido:

2	2
2	3

Por exemplo,
c[i,j] = c[i-1,j-1] +1 = 2+1=3

Sabemos que:

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- Começar de *c[m,n]* e retornar
- Sempre que *c[i,j] = c[i-1, j-1]+1*, guardar *x[i]* (*x[i]* faz parte da LCS)
- Se i=0 or j=0 (retorno chega ao fim)
- A saída é o conjunto de símbolos guardados em ordem inversa

Voltando ao nosso exemplo

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

---

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

LCS (ordem inversa): **B C B**

Problema: Multiplicação de Cadeia de Matrizes

- Seja a seqüência (cadeia) <A_1, A_2,…, A_n> de n matrizes.
- Computar o produto $A_1A_2…A_n$ de forma a minimizar o número de multiplicações
- Duas matrizes A e B podem ser multiplicadas se forem compatíveis
  - Número de colunas de A = Número de linhas de B
  - A(p*q) * B(q*r) → C(p*r)
  - O número de multiplicações é p*q*r

Exemplo:

- $<A_1, A_2, A_3>$ (10, 100, 5, 50)
  - $((A_1A_2)A_3) \rightarrow$ 10*100*5 + 10*5*50 = 5000 + 2500 = 7500
  - $(A_1(A_2A_3)) \rightarrow$ 100*5*50 + 10*100*50 = 25000 + 50000 = 75000

- Seja A_i de dimensão $p_{i-1}*p_i$
- Encontrar forma de definir parênteses para minimizar o número de multiplicações.
- Quantas formas diferentes?
  - **$\Omega(2^n)$.**

Impraticável verificar todas as possibilidades

Análise e Técnicas de Algoritmos – 2005.1 © Jorge Figueiredo, DSC/UFCG

Multiplicação de Duas Matrizes

```
Multiplica-Matrizes(A, B)
if colunas[A] ≠ Linhas [B] then
      ERRO
else
      for i ← 1 to Linhas[A] do
            for j ← 1 to Colunas[B] do
                  C[i, j] ← 0
                  for k ← 1 to Colunas[A] do
                        c[i, j] ← c[i, j] + A[i,k].B[k,j]
return  C
```

Análise e Técnicas de Algoritmos – 2005.1 © Jorge Figueiredo, DSC/UFCG

Qual a Idéia?

- Notação: $A_{i..j}$ = resultado da avaliação de $A_iA_{i+1}\ldots A_j$ $(i \le j)$
- Qualquer forma de colocar parênteses em $A_iA_{i+1}\ldots A_j$ deve dividir a cadeia entre A_k e A_{k+1}, para algum inteiro k, $i \le k < j$
  - Custo = custo de computar $A_{i..k}$ + custo de computar $A_{k+1..j}$ + custo de multiplicar $A_{i..k}$ e $A_{k+1..j}$
  - A sub-cadeia $A_iA_{i+1}\ldots A_k$ deve ter *parentização* ótima
  - A sub-cadeia $A_{k+1}A_{i+1}\ldots A_j$ deve ter *parentização* ótima

Análise e Técnicas de Algoritmos – 2005.1 © Jorge Figueiredo, DSC/UFCG

Subestrutura Ótima

Mínimo
Custo_$A_{1..6}$ + Custo_$A_{7..9}$+$p_0p_6p_9$

$A_1A_2A_3A_4A_5A_6A_7A_8A_9$

Suponha $((A_1A_2)(A_3((A_4A_5)A_6)))$ $((A_7A_8)A_9)$ É ótima

então (A_1A_2) $(A_3((A_4A_5)A_6))$ Deve ser ótima para $A_1A_2A_3A_4A_5A_6$

senão, se $(A_1(A_2A_3))$ $((A_4A_5)A_6)$ É ótima para $A_1A_2A_3A_4A_5A_6$

então $((A_1(A_2A_3))((A_4A_5)A_6))$ $((A_7A_8)A_9)$ Seria melhor do que

$((A_1A_2)(A_3((A_4A_5)A_6)))$ $((A_7A_8)A_9)$

Análise e Técnicas de Algoritmos – 2005.1 © Jorge Figueiredo, DSC/UFCG

A Solução

- Sub-problema: determinar o custo mínimo de $A_iA_{i+1}\ldots A_j$ $(1 \le i \le j \le n)$
- m[i..j] = número mínimo de multiplicações para calcular a matriz $A_{i..j}$
- s[i, j] = k, em que m[i, j] = m[i, k] + m[k+1, j] + $p_{i-1}p_kp_j$

$$m[i,j] = \begin{cases} 0, & se\ i = j \\ \min\limits_{1 \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j & se\ i < j \end{cases}$$

Análise e Técnicas de Algoritmos – 2005.1 © Jorge Figueiredo, DSC/UFCG

- A resposta está em m[1, n].
- Necessidade de Programação Dinâmica: overlap de problemas.
- Caso base: m[i, i] = 0.
- Calcular primeiro m[i, i+1], depois m[i, i+2], …
- Caminhamento por diagonal.


```
MATRIX-CHAIN-ORDER(p)
1  n ← length[p] − 1
2  for i ← 1 to n
3      do m[i, i] ← 0
4  for l ← 2 to n                    ▷ l is the chain length.
5      do for i ← 1 to n − l + 1
6             do j ← i + l − 1
7                m[i, j] ← ∞
8                for k ← i to j − 1
9                    do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                      if q < m[i, j]
11                         then m[i, j] ← q
12                              s[i, j] ← k
13 return m and s
```

$O(n^3)$, $\Omega(n^3)$ ➔ $\Theta(n^3)$ running time

$\Theta(n^2)$ space

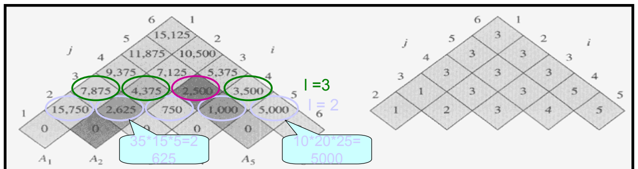


Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

m[3,5] = min { m[3,4]+m[5,5] + 15*10*20 =750 + 0 + 3000 = 3750 ; m[3,3]+m[4,5] + 15*5*20 =0 + 1000 + 1500 = 2500 }

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the m table, and only the upper triangle is used in the s table. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$
$$= 7125.$$

Colocando os Parêntesis

- s[i, j] armazena o valor de k ótimo para $A_i A_{i+1} \ldots A_j$, dividindo a matriz em A_k e A_{k+1}
  - $A_{1..n}$ ➔ $A_{1..s[1..n]} A_{s[1..n]+1..n}$
  - $A_{1..s[1..n]}$ ➔ $A_{1..s[1,\ s[1..n]]} A_{s[1,\ s[1..n]]+1..s[1..n]}$

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i = j
   then print "A"_i
   else print "("
        PRINT-OPTIMAL-PARENS(s, i, s[i, j])
        PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
        print ")"
```