PROGRAMAÇÃO DINÂMICA

- Programação dinâmica é tipicamente aplicada para problemas de otimização.
- O desenvolvimento de um algoritmo de programação dinâmica pode ser divido em 4 etapas

1. Caracterizar uma solução ótima;
2. Define recursivamente a solução ótima;
3. Calcular o valor da solução ótima de forma bottom-up;
4. Construir uma solução ótima das informações computadas.

Numero de Fibonacci

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Algoritmo recursivo:

```
int fib( int n )
{
   if ( n < 2 ) return n;
   else return fib(n-1) + fib(n-2);
}
```

t_n: tempo requerido para calcular f_n (n-ésimo ordem de numero Fibonacci.

$$t_n = t_{n-1} + t_{n-2}$$

$$t_1 = t_2 = c, \text{ onde c é um constante}$$

Então

$$t_n = cf_n$$

Sabemos que

$$\lim_{n\to\infty} \frac{f_{n+1}}{f_n} = \frac{1+\sqrt{5}}{2} = 1.61803...$$

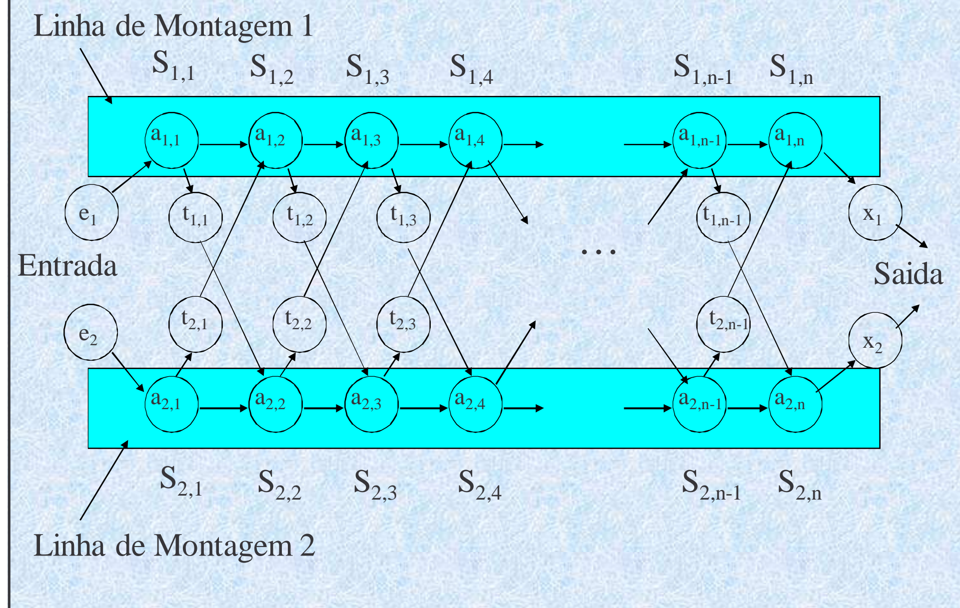
Finalmente, $t_n = O(f_n) = O(1.618..^n)$

Algoritmo Iterativo

```
int fib( int n )
{
    int k, f1, f2;
    if ( n < 2 ) return n;
    else
    {
        f1 = f2 = 1;
        for(k=2;k<n;k++)
        {
            f = f1 + f2;
            f2 = f1;
            f1 = f;
        } return f;
    }
}
```

$T_n = O(n)$

Exemplo 1: Linha de Montagem



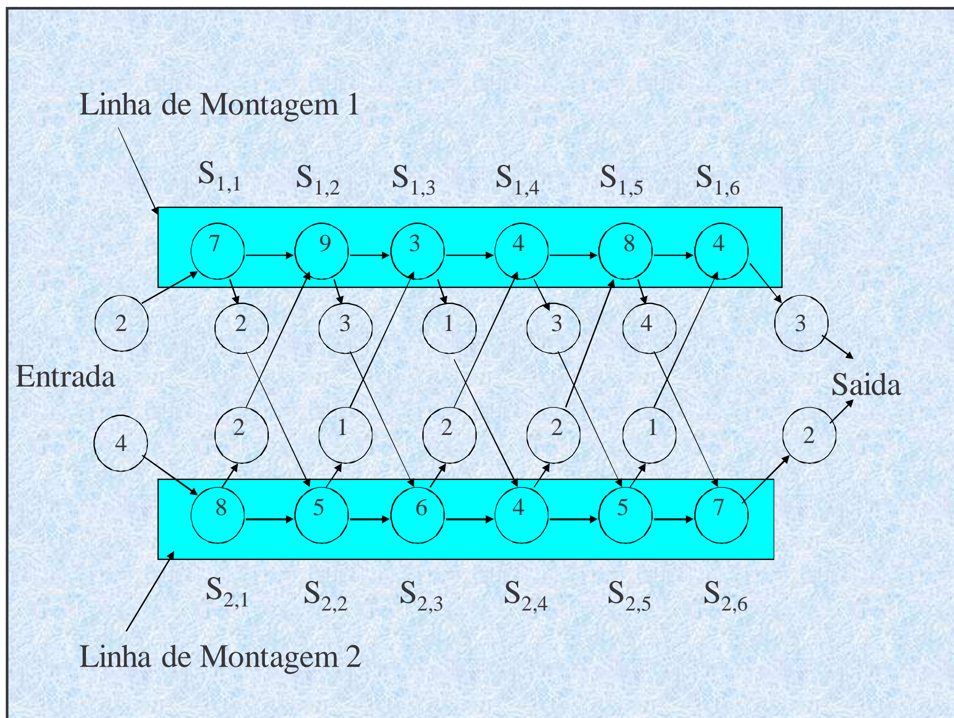
Descrição:

1. Duas linhas de montagem, cada uma tem *n* estações, $S_{i,j}$, i = 1, 2; j = 1, 2, ..., n.
2. A j-ésima estação na linha 1 tem a mesma função com a j-ésima estação na linha 2;
3. O tempo requerido na estação $S_{i,j}$ é denotado por $a_{i,j}$, varia de uma a outra;
4. O tempo que leva de uma estação a próxima da mesma linha pode ser ignorado;
5. O tempo que leva para transferir as peças da linha i para j depois de passar a estação $S_{i,j}$ é $t_{i,j}$.

Objetivo:

Determinar uma combinação de estações das duas linhas de montagem para minimizar tempo de montagem.

Usando o método de força brutal, a ordem de complexidade é de $O(2^n)$.



Passo 1: Estrutura de uma solução ótima

Considera as duas estações $S_{1,j}$ e $S_{2,j}$. O caminho mais rápido que passa pela estação $S_{1,j}$ é

- O caminho mais rápido que passa pela estação $S_{1,j-1}$ e diretamente avança para a estação $S_{1,j}$, ou
- O caminho mais rápido que passa pela estação $S_{2,j-1}$ e transfere de linha 2 a linha 1, então passa pela estação $S_{1,j}$.

Usando um raciocínio simétrico, O caminho mais rápido passando a estação $S_{2,j}$ é

- O caminho mais rápido que passa pela estação $S_{2,j-1}$ e diretamente avança para a estação $S_{2,j}$, ou
- O caminho mais rápido que passa pela estação $S_{1,j-1}$ e transfere de linha 1 a linha 2, então passa pela estação $S_{2,j}$.

Passo 2: Uma solução recursiva

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

$$f_1[1] = e_1 + a_{1,1}$$

$$f_2[1] = e_2 + a_{2,1}$$

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

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

f^*: O tempo mais rápido que passa pela a fabrica;

$f_i[j]$: O tempo mais rápido que passa pela estação $S_{i,j}$ do ponto de começo;

Passo 3:

Calcula o tempo mais rápido

Fastest-Way(a, t, e, x, n)

- $f_1[1] \leftarrow e_1 + a_{1,1}$
- $f_2[1] \leftarrow e_2 + a_{2,1}$
- **for** $j \leftarrow 2$ to n
- **do if** $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$
- **then** $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$
- $l[j] = 1$
- **else** $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$
- $l[j] = 2$
- **if** $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$
- **then** $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$
- $l[j] = 2$
- **else** $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$
- $l[j] = 1$
- **if** $f_1[n] + x_1 \leq f_2[n] + x_2$
- **then** $f^* \leftarrow f_1[n] + x_1$
- $l^* = 1$
- **else** $f^* \leftarrow f_2[n] + x_2$
- $l^* = 2$

Passo 4:

Construindo o caminho mais rápido

Print-Stations(l, n)

- i ← l*
- print “linha” i, “estacao” n
- **for** j ← n downto 2
- **do** i ← l[j]
- print “linha” i, “estacao” j-1

Exemplo 2: Multiplicação de cadeia de matrizes:

- Multiplicação de matrizes é uma operação associativa, i.e., $A_1(A_2A_3) = (A_1 A_2)A_3$
- O custo para multiplicar uma matriz de $\boldsymbol{n \times m}$ por uma matriz de $\boldsymbol{m \times p}$ é $\boldsymbol{O(nmp)}$ (ou $\boldsymbol{O(n^3)}$ para duas matrizes de $\boldsymbol{n \times n}$).
- $\mathbf{A_2A_3}$ 100x5x50 = 25000 => $\mathbf{A_2A_3}$ (100x50)

  $\mathbf{A_1(A_2A_3)}$ 10x100x50 = 50000 => $\mathbf{A_1A_2A_3}$ (10x50)

  Total 75000

- Um escolhe de parentização ruim pode ser caro. Por exemplo,

Matriz	Linhas	Colunas
$\mathbf{A_1}$	10	100
$\mathbf{A_2}$	100	5
$\mathbf{A_3}$	5	50

O custo para $\mathbf{(A_1 A_2) A_3}$ é

$\mathbf{A_1A_2}$ 10x100x5 = 5000 => $\mathbf{A_1 A_2}$ (10x5)

$\mathbf{(A_1A_2) A_3}$ 10x5x50 = 2500 => $\mathbf{A_1A_2A_3}$ (10x50)

Total 7500

mas, para $\mathbf{A_1 (A_2 A_3)}$

$\mathbf{A_2A_3}$ 100x5x50 = 25000 => $\mathbf{A_2A_3}$ (100x50)

$\mathbf{A_1(A_2A_3)}$ 10x100x50 = 50000 => $\mathbf{A_1A_2A_3}$ (10x50)

Total 75000

Objetivo

Dada uma seqüência de matrizes <A_1, A_2, ..., A_n>, matriz A_i tem a dimensionalidade $p_{i-1} \times p_i$, encontra uma parentização para o produto $A_1 A_2 ... A_n$ minimizando numero escalar de multiplicações.

Exemplo, o produto $A_1A_2A_3A_4$ pode ser calculado pelas seguintes 5 formas:

$(A_1(A_2(A_3A_4)))$
$(A_1((A_2A_3)A_4))$
$((A_1A_2)(A_3A_4))$
$((A_1(A_2A_3))A_4)$
$(((A_1A_2)A_3)A_4)$

Algoritmo para multiplicação de duas matrizes:

```
Matriz-Multiply(A, B)
•   if columns[A] ≠ rows[B]
•      then error "dimensoes imcompativeis"
•      else for i ← 1 to rows[A]
•               do for j ← 1 to columns[B]
•                      do  C[i, j] ← 0
•                          for k ← 1 to columns[A]
•                              do C[i, j] ← C[i, j] + A[i, k]B[k, j]
•   return C
```

Passo 1:

- $A_{i..j}$: O produto $A_iA_{i+1}A_{i+2}...A_j$, $i \le j$;
- Existe um numero inteiro k ($i \le k < j$) que divide a seqüência em duas sub-cadeias: $A_{i..k}A_{k+1..j}$;
- Custo($A_{i..j}$) = Custo($A_{i..k}$)+ Custo($A_{k+1..j}$)
  +Custo(multiplicação das duas)
- A sub-estrutura ótima é a seguinte:
  Suponha que a ótima parentização de $A_iA_{i+1}A_{i+2}...A_j$ divide o produto entre A_k e A_{k+1}. Então, esta parentização para a sub-cadeia $A_iA_{i+1}A_{i+2}...A_k$ é ótima também.
  A afirmação similar para $A_{k+1}A_{k+2}A_{k+k}...A_j$ também é verdade.

Passo 2: Uma solução recursiva

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

- m[i, j]: numero mínimo de multiplicações escalares necessárias
  para calcular $A_{i..j}$.
- Cada matriz A_i tem a dimensionalidade $p_{i-1} \times p_i$,

Passo 3:

Calcula o menor custo

$p = < p_0 p_1 p_2 \ldots p_n >$

```
Matrix-Chain-Order(p)
  n ← length(p)-1
  for i = 1 to n
      do m[i, i] = 0
  for l ← 2 to n
      do for i ← 1 to n-l+1
          do  j ← i+l-1
              m[i, j] ← ∞
              for k ← i to j-1
                  do q  ← m[i,k]+m[k+1,j]+p_{i-1}p_k p_j
                      if q < m[i,j]
                          then m[i,j] ← q
                               s[i,j] ← k
return m e s
```

Exemplo

Matriz	**Linhas**	**Colunas**
$\mathbf{A_1}$	30	35
$\mathbf{A_2}$	35	15
$\mathbf{A_3}$	15	5
$\mathbf{A_4}$	5	10
$\mathbf{A_5}$	10	20
$\mathbf{A_6}$	20	25

Exemplo

i						
6	0					
5	5000	0				
4	3500	1000	0			
3	5375	2500	750	0		
2	10500	7125	4375	2625	0	
1	15125	11875	9375	7875	15750	0
j	6	5	4	3	2	1

Exemplo

i					
5	5				
4	5	4			
3	3	3	3		
2	3	3	3	2	
1	3	3	3	1	1
j	6	5	4	3	2

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \times 15 \times 20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \times 5 \times 20 = 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4735 + 0 + 35 \times 10 \times 20 = 11375 \end{cases}$$

$$= 7125$$

Passo 4:

Construindo uma solução ótima

```
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 ")"
```

Subseqüência Comum Mais Longa

Dada uma seqüência $X = <x_1, x_2, \ldots, x_m>$, outra seqüência $z = <z_1, z_2, \ldots, z_k>$ é uma **subseqüência** de X se existe uma seqüência estritamente crescente $<i_1, i_2, \ldots, i_k>$ de índices de X tais que, para todo $j = 1, 2, \ldots, k$, temos

$$x_{i_j} = z_j$$

Exemplo,

X = <A, B, C, B, D, A, B> e Z = <B, C, D, B>
Z é uma subseqüência de X com seqüência de índice
Correspondente <2, 3, 5, 7>.

Subseqüência Comum Mais Longa

Dada duas seqüências X e Y, dizemos que uma seqüência z é uma **subseqüência comum** de X e Y se Z é uma subseqüência de X e de Y ao mesmo tempo.

Exemplo,

X = <A, B, C, B, D, A, B> e Y = <B, D, C, A, B, A>
A seqüência <B, C, A> é uma subseqüência comum.
A seqüência <B, C, B, A> é uma **subseqüência comum mais longa** (**LCS – longest common subsequence**)

Subseqüência Comum Mais Longa

Objetivo

Dada duas seqüências $X = <x_1, x_2, ..., x_m>$ e $Y = <y_1, y_2, ..., y_n>$,

Desejamos encontrar uma subseqüência **comum** de comprimento

máximo de X e Y.

Subseqüência Comum Mais Longa

Exemplo

Em aplicações biológicas, freqüentemente queremos comparar o DNA

De dois ou mais organismos diferentes. Uma cadeia de DNA consiste

Em uma cadeia de moléculas chamadas bases, na qual as bases

possíveis são adenina, guanina, citosina e timina. Então, uma cadeia

De DNA pode ser expressa como uma cadeia sobre o conjunto finito

{A, C, G, T}.

Subseqüência Comum Mais Longa

Exemplo

S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA

Quanto mais longa a cadeia S_3 que podemos encontrar,

Maior será a semelhança entre S_1 e S_2.

Em nosso exemplo, S_3 = GTCGTCGGAAGCCGGCCGAA

Subseqüência Comum Mais Longa

1) Caracterização da Solução

Dada uma seqüência $X = <x_1, x_2, ..., x_m>$, definimos o *i*-ésimo prefixo de X, para $i = 0, 1, ..., m$, como $X_i = < x_1, x_2, ..., x_i>$

Por exemplo, X = <A, B, C, B, D, A, B>, então X_4 = <A, B, C, B> é 4-ésimo préfixo. X_0 é uma seqüência vazia.

Subseqüência Comum Mais Longa

1) Caracterização da Solução

Teorema (Subestrutura ótima de uma LCS)

Seja as seqüências X = <x_1, x_2, ..., x_m> e Y = <y_1, y_2, ..., y_n>,

e seja Z = <z_1, z_2, ..., z_l> qualquer LCS de X e Y.

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

Subseqüência Comum Mais Longa

2) Uma Solução Recursiva

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

3) Calculando o comprimento de uma LCS

```
LCS-LENGTH(X, Y)
1 m ← comprimento [X]
2 n ← comprimento [Y]
3 for i ← 0 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 ← 0 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 e b
```

4) A Construção de uma LCS

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

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4