Programação Dinâmica

Prof. Marcio Delamaro ICMC/USP

Características

- Resolve problemas combinando soluções para subproblemas

Características

- Resolve problemas combinando soluções para subproblemas
- Opa!!! Divisão e conquista????

Características

- Resolve problemas combinando soluções para subproblemas
- Opa!!! Divisão e conquista????
- Na programação dinâmica, subproblemas não são independentes
- Subproblemas compartilham subproblemas

O que é programação dinâmica

- Dynamic programming is a fancy name for recursion with a table. Instead of solving subproblems recursively, solve them sequentially and store their solutions in a table. The trick is to solve them in the right order so that whenever the solution to a subproblem is needed, it is already available in the table. Dynamic programming is particularly useful on problems for which divide-and-conquer appears to yield an exponential number of subproblems, but there are really only a small number of subproblems repeated exponentially often. In this case, it makes sense to compute each solution the first time and store it away in a table for later use, instead of recomputing it recursively every time it is needed.
  — Ian Parberry

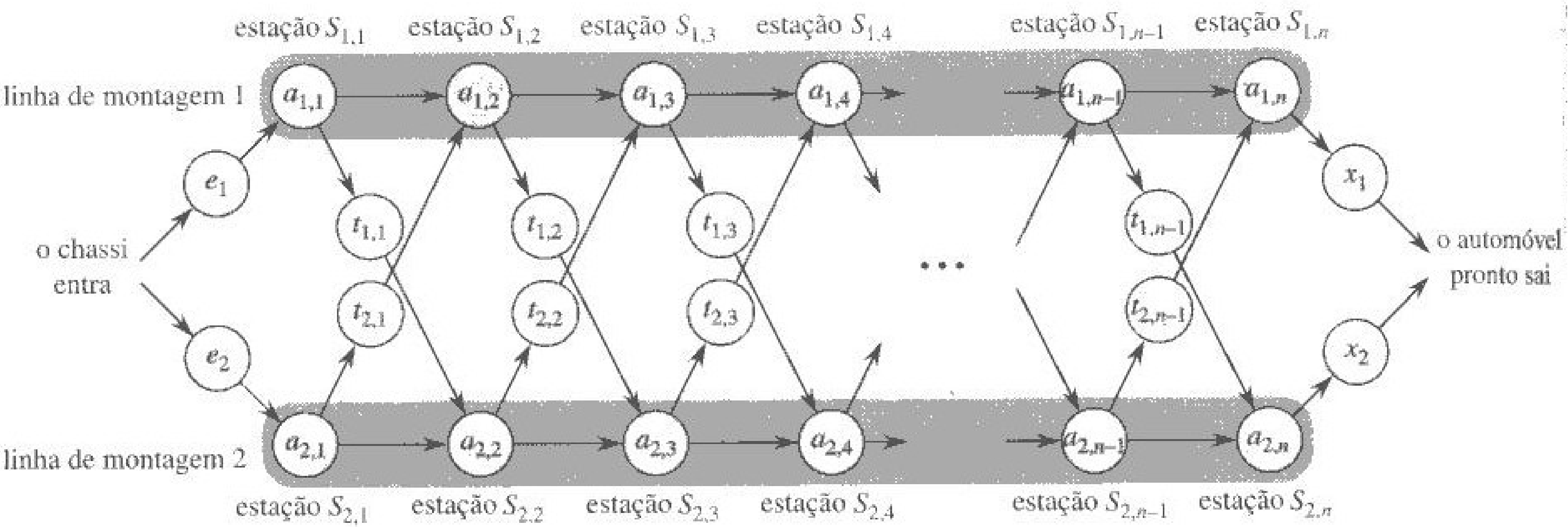
Visão Geral

1) Caracterizar a estrutura de uma solução ótima
2) Definir recursivamente o valor de uma solução ótima
3) Calcular o valor de uma solução ótima bottom-up
4) Construir uma solução ótima a partir das informações calculadas

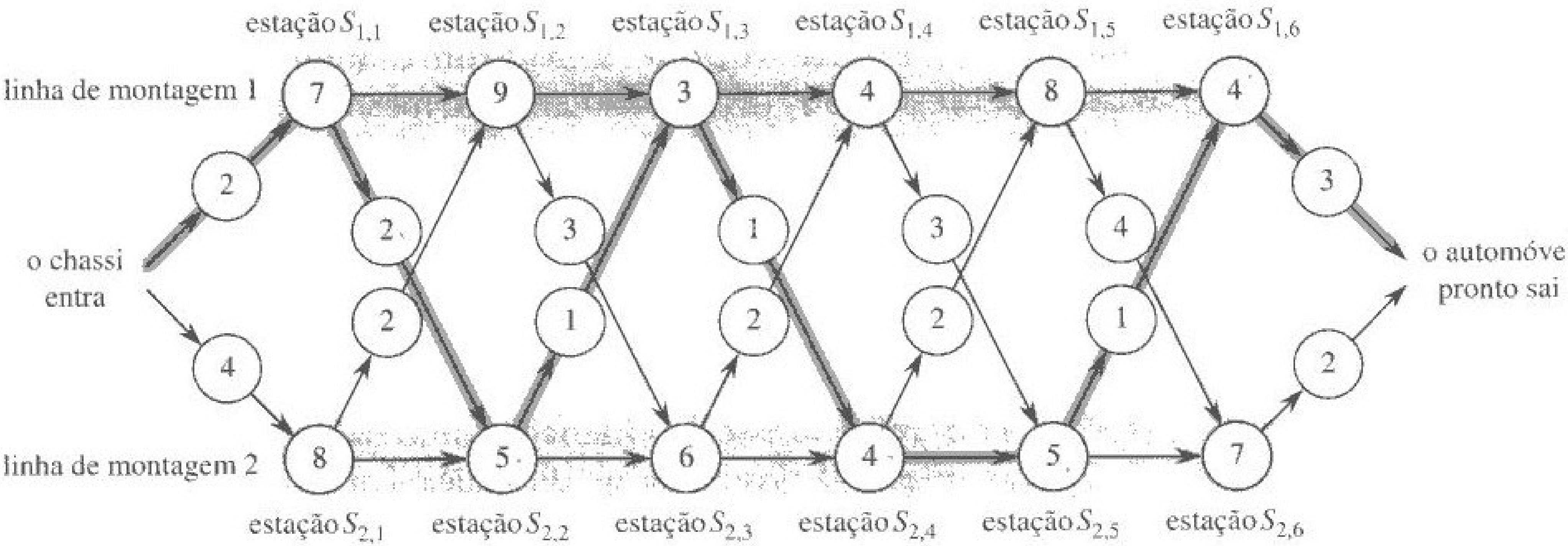
Exemplo – linha de montagem

- Fábrica tem duas linhas de montagem cada uma com n estações
- Cada estação tem tempo de montagem $a_{i,j}$
- Tempo para entrar (e_i) e sair (x_i) de cada linha
- Custo para transferir de uma uma linha para outra $t_{i,j}$
- Determinar quais estações escolher para minimizar o tempo total de passagem do automóvel.

Linha de montagem



Linha de montagem – exemplo



(a)

Etapa 1 – Estrutura ótima

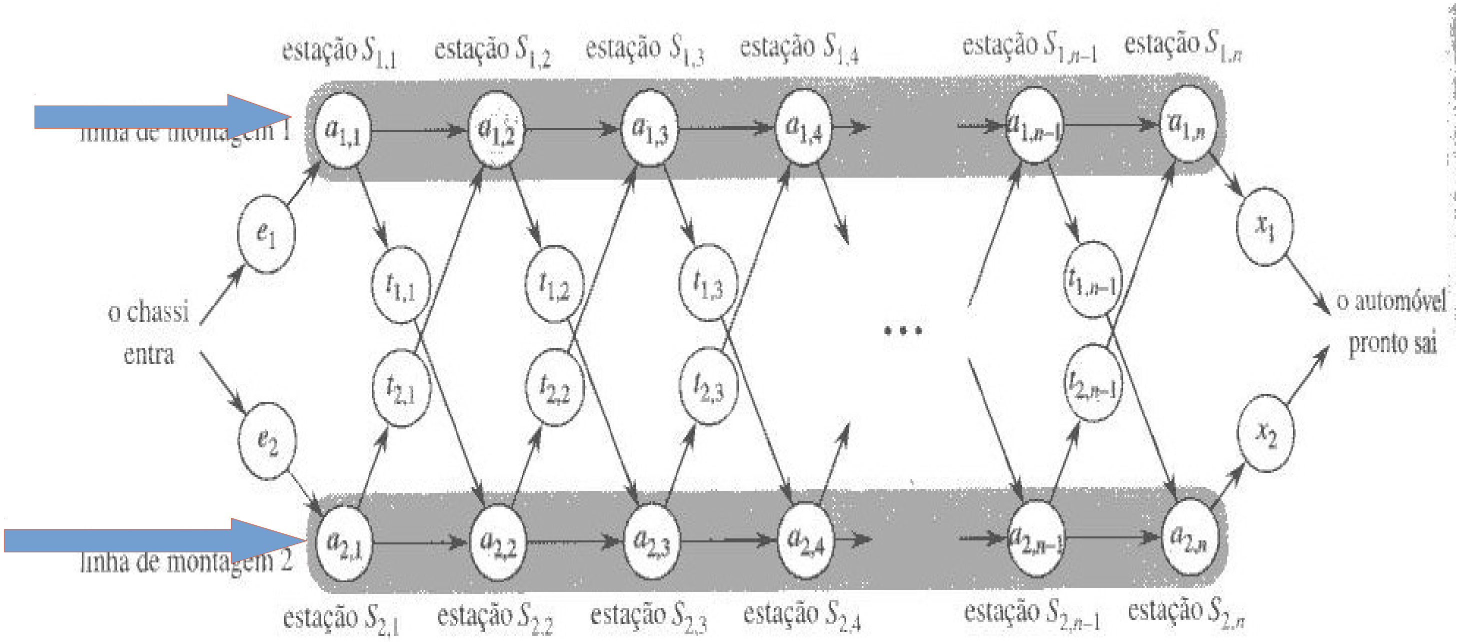
- Como é a estrutura de uma solução ótima?

Etapa 1 – Estrutura ótima

- Como é a estrutura de uma solução ótima?
- Ou, como executar a estação $S_{i,j}$ com o menor custo possível?

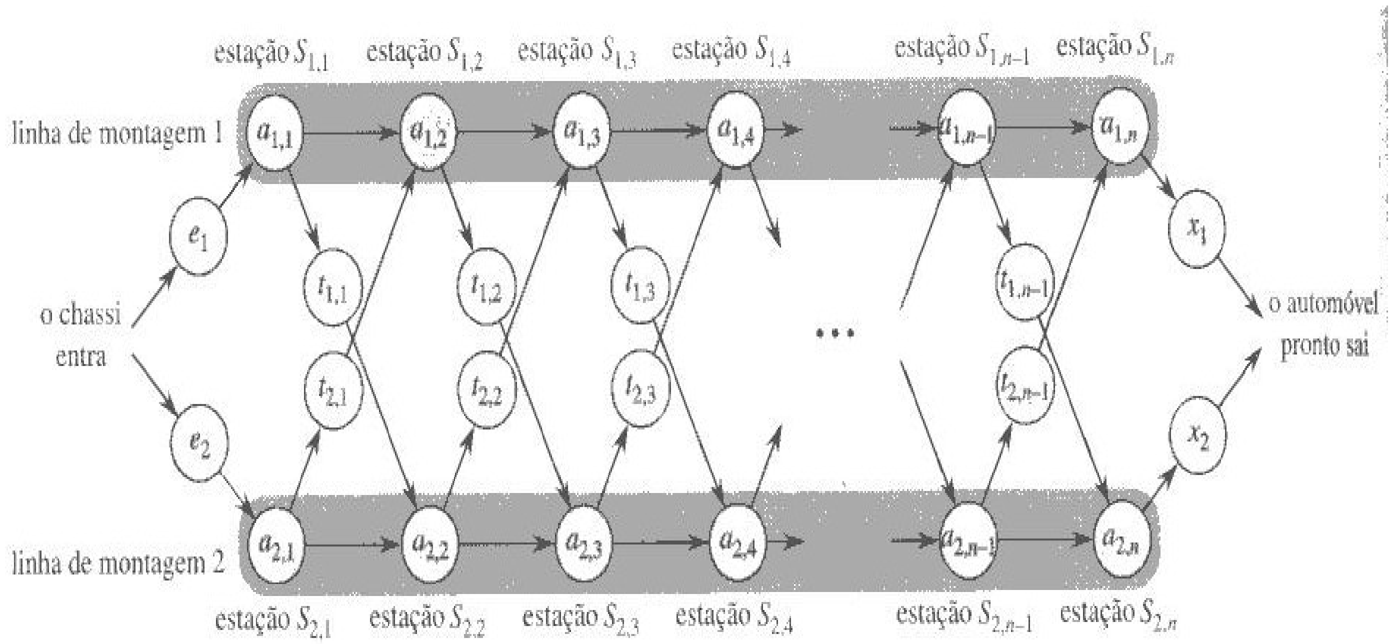
Etapa 1 – Estrutura ótima

- Como é a estrutura de uma solução ótima?
- Ou, como executar a estação Si,j com o menor custo possível?
- Se j = 1, só existe uma opção possível
-



Etapa 1 – Estrutura ótima

- Como é a estrutura de uma solução ótima?
- Ou, como executar a estação Si,j com o menor custo possível?
- Se j = 1, só existe uma opção possível
- Se j = 2, 3, ..



Estrutura ótima

- Então, uma solução ótima tem sempre essa cara:
  - Caminho mais rápido passando pela estação j-1 da mesma linha
  - Caminho mais rápido passando pela estação j-1 da outra linha, somando-se o custo de transferência

Etapa 2 – solução recursiva

- Vamos definir duas funções, f1 e f2
- f1(j) computa o menor custo para executar a estação j da linha 1
- f2(j) computa o menor custo para executar a estação j da linha 2

Etapa 2 – solução recursiva

- Vamos definir duas funções, f1 e f2
- f1(j) computa o menor custo para executar a estação j da linha 1
- f2(j) computa o menor custo para executar a estação j da linha 2
- Vamos implementar essas funções!!!

Solução recursiva

```
f1(j)
   if ( j == 1 )
     return e1 + a1[1]

   c1 = f1(j-1) + a1[j]
   c2 = f2(j-1) + t2[j-1] + a1[j]

   if c1 < c2 return c1
   return c2
```

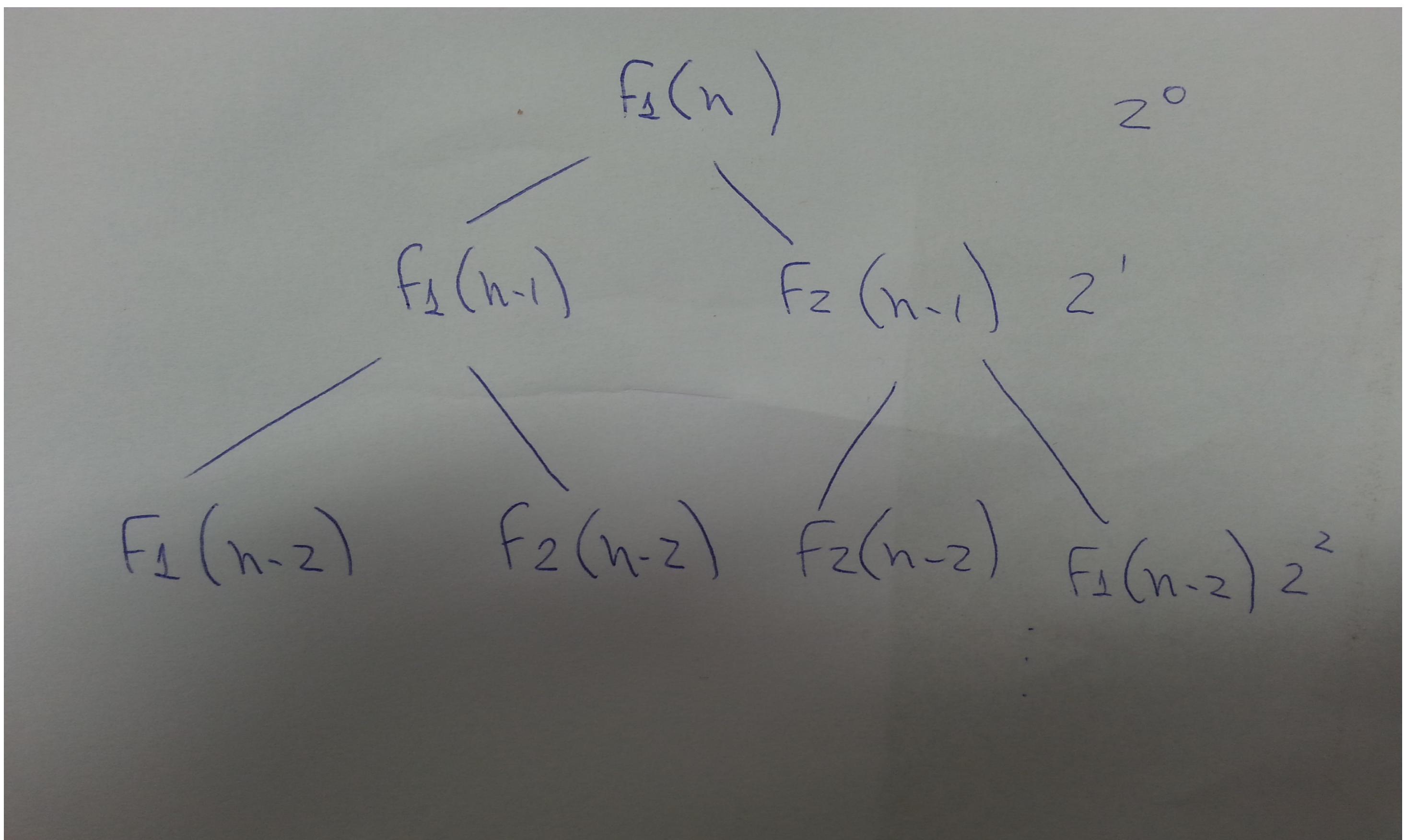
```
f2(j)
   if ( j == 1 )
     return e2 + a2[1]

   c1 = f2(j-1) + a2[j]
   c2 = f1(j-1) + t1[j-1] + a2[j]

   if c1 < c2 return c1
   return c2
```

Solução recursiva

- Quantas chamadas a f1 e f2 são necessárias se tivermos n estações?

Solução recursiva

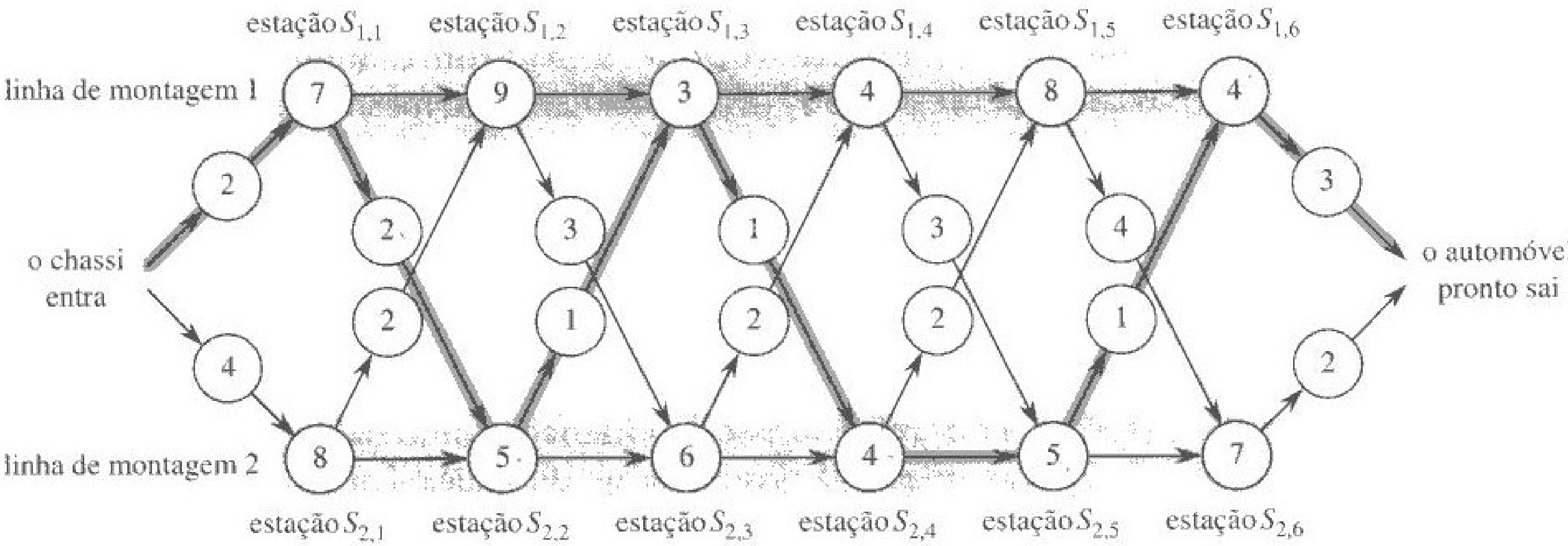
- Quantas chamadas a f1 e f2 são necessárias se tivermos n estações?



Etapa 3 – cálculo dos tempos

- Solução iterativa, realizada botton-up
- Vamos armazenar custos em duas tabelas f1 e f2
- A partir de f1[1] e f2[1] conseguimos calcular f1[2] e f2[2]
- E assim por diante

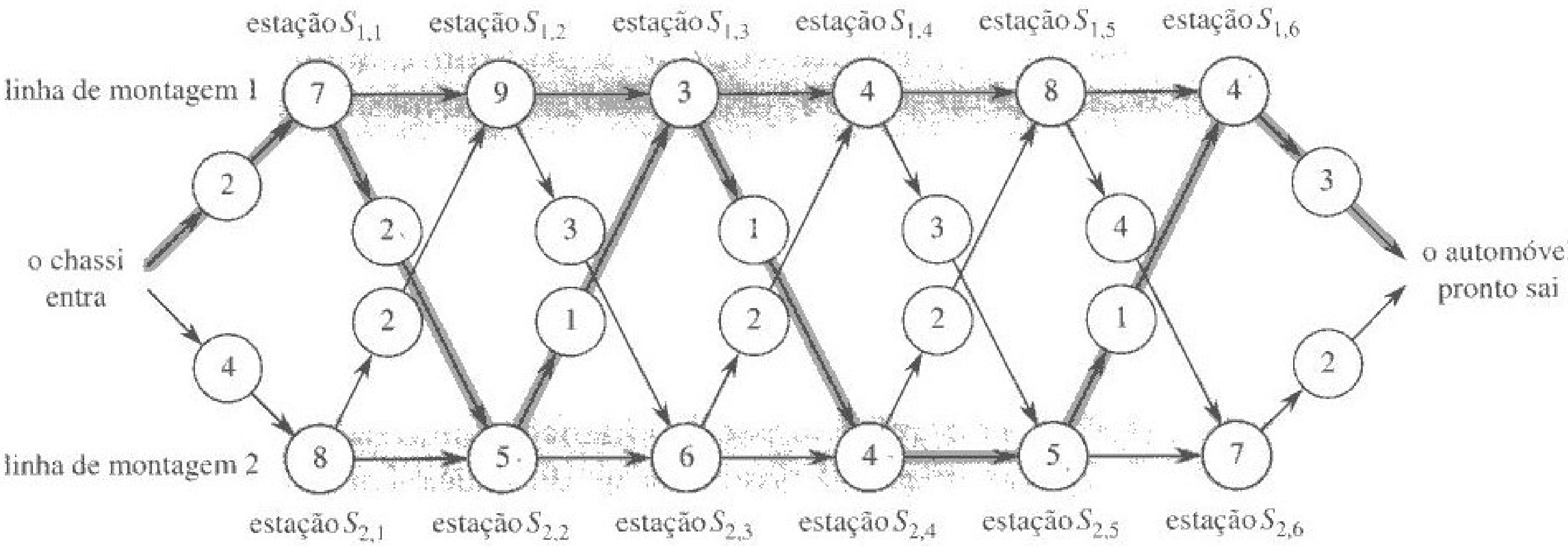
Cálculo dos tempos



(a)

	1	2	3	4	5	6
f1						
f2						

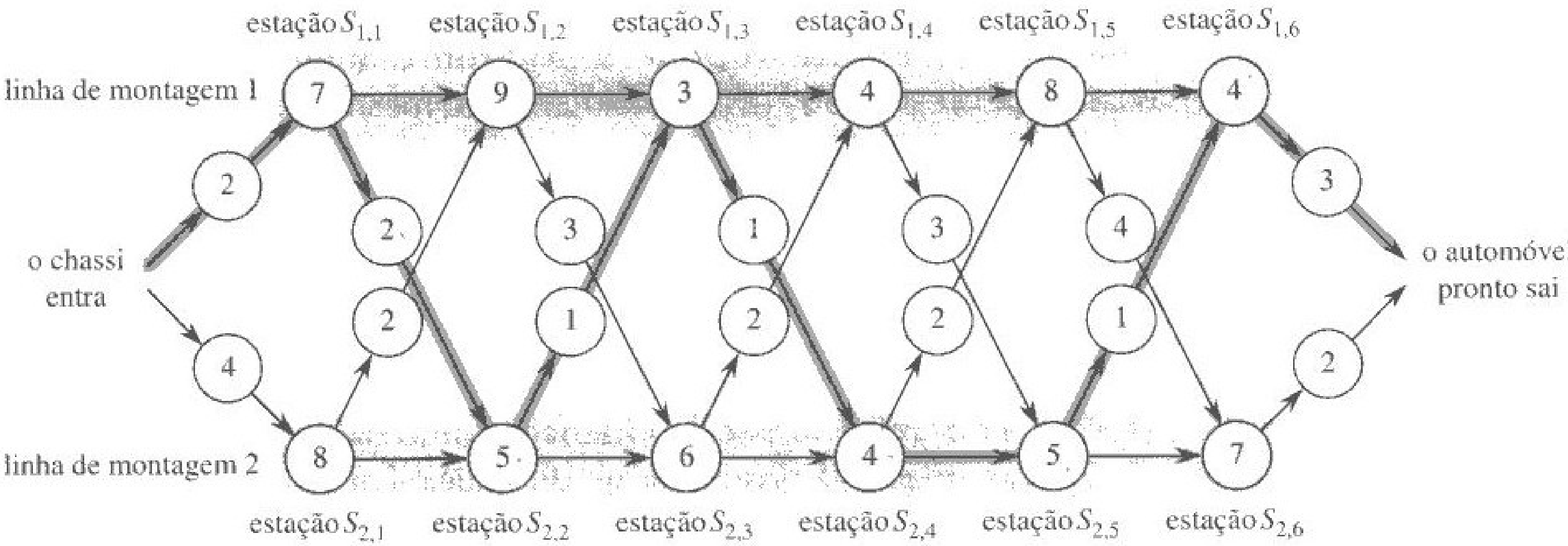
Cálculo dos tempos



(a)

	1	2	3	4	5	6
f1	9					
f2	12					

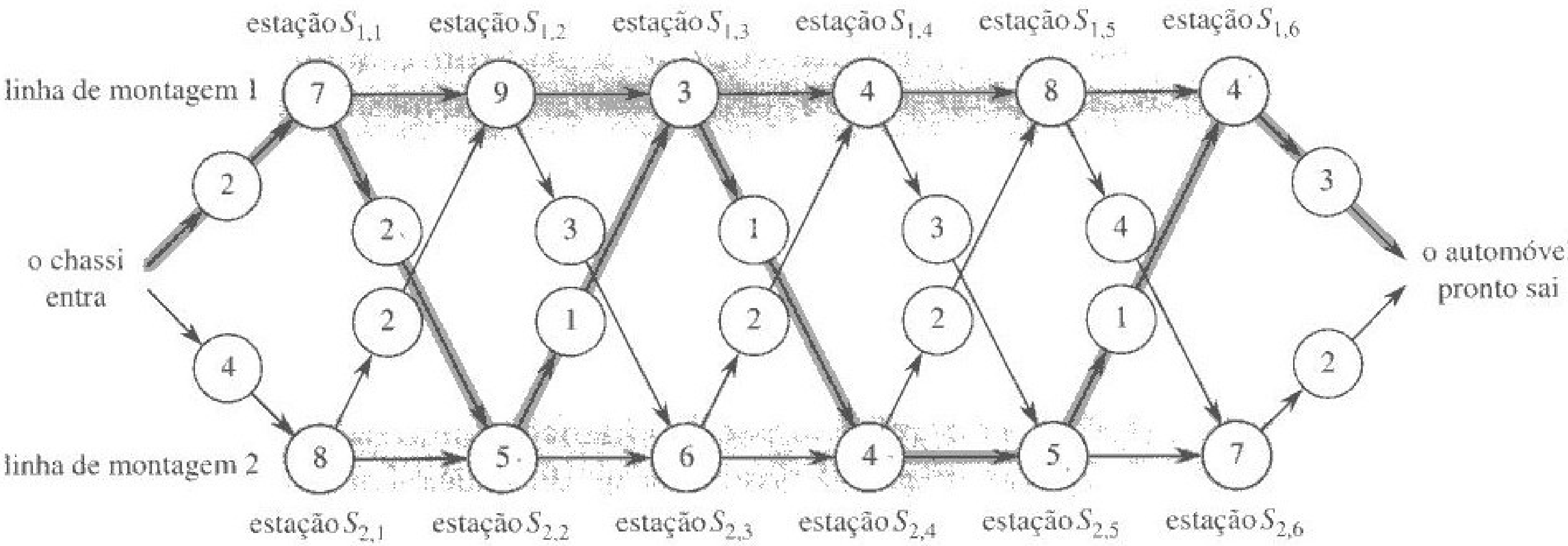
Cálculo dos tempos



(a)

	1	2	3	4	5	6
f1	9	18				
f2	12	16				

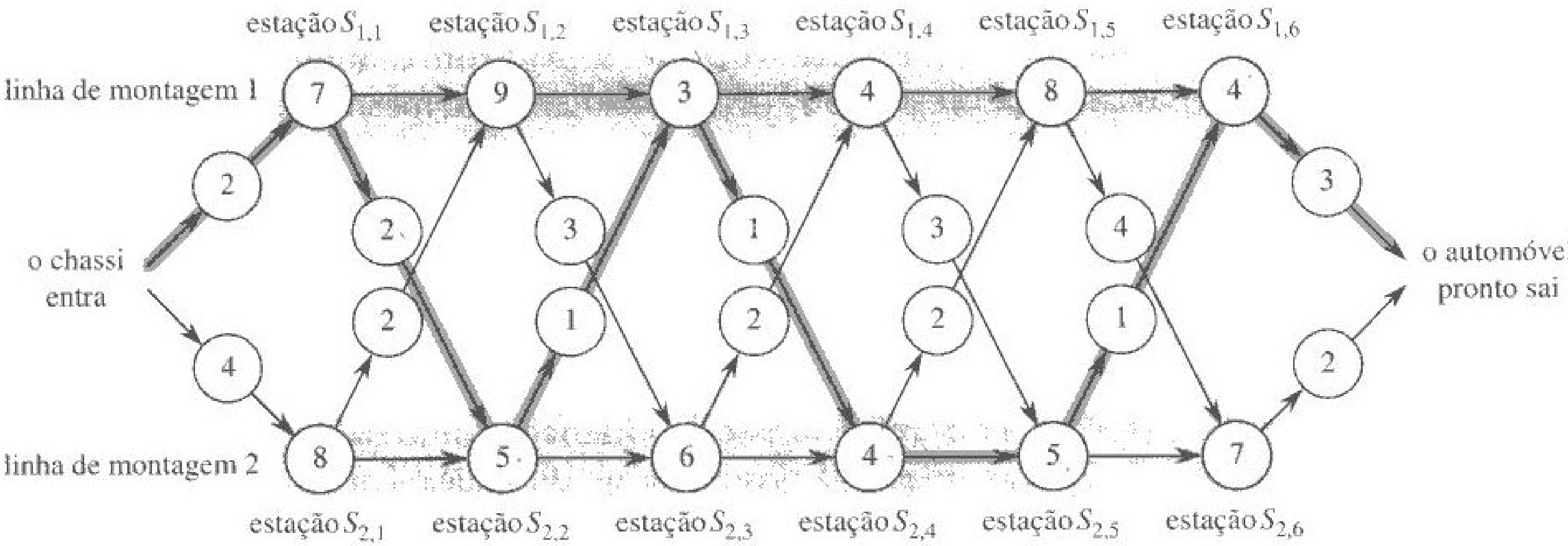
Cálculo dos tempos



(a)

	1	2	3	4	5	6
f1	9	18	20			
f2	12	16	22			

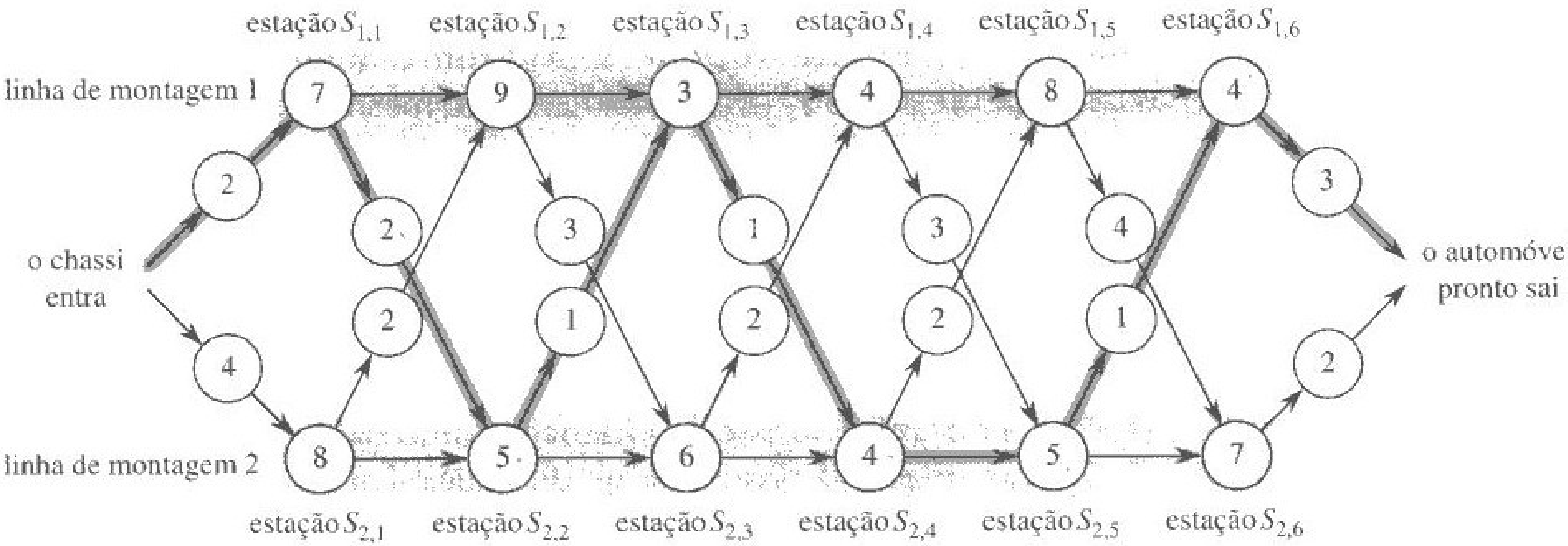
Cálculo dos tempos



(a)

	1	2	3	4	5	6
f1	9	18	20	24		
f2	12	16	22	25		

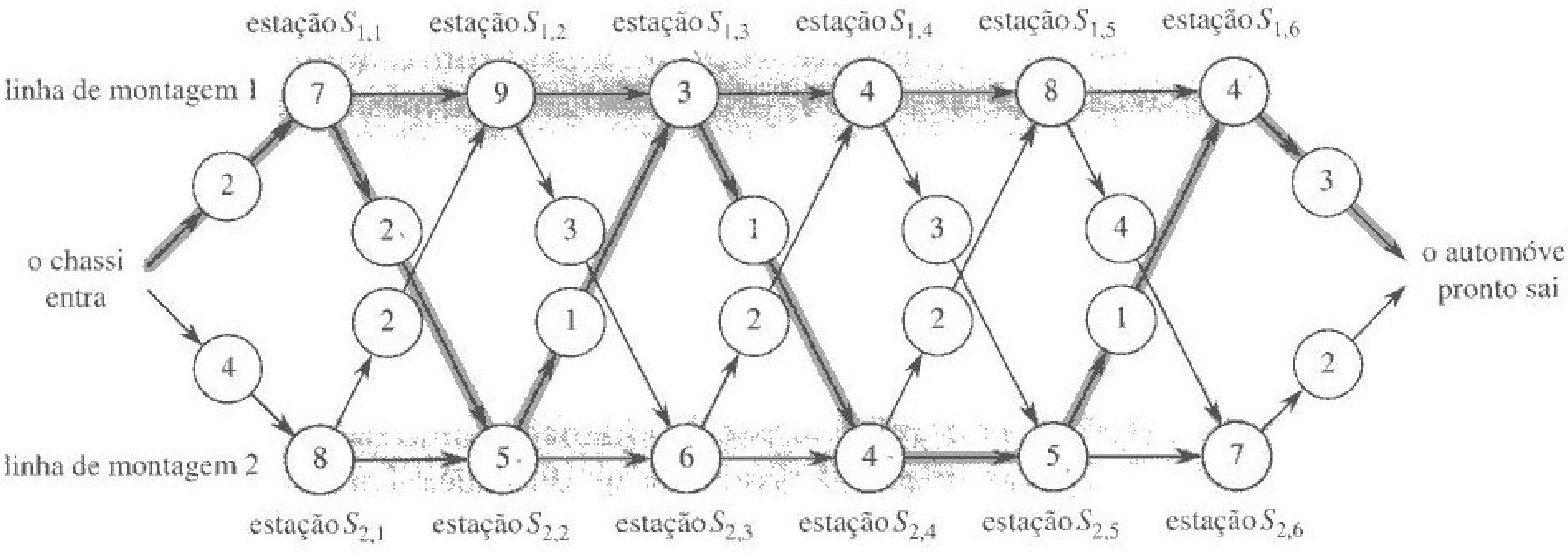
Cálculo dos tempos



(a)

	1	2	3	4	5	6
f1	9	18	20	24	32	
f2	12	16	22	25	30	

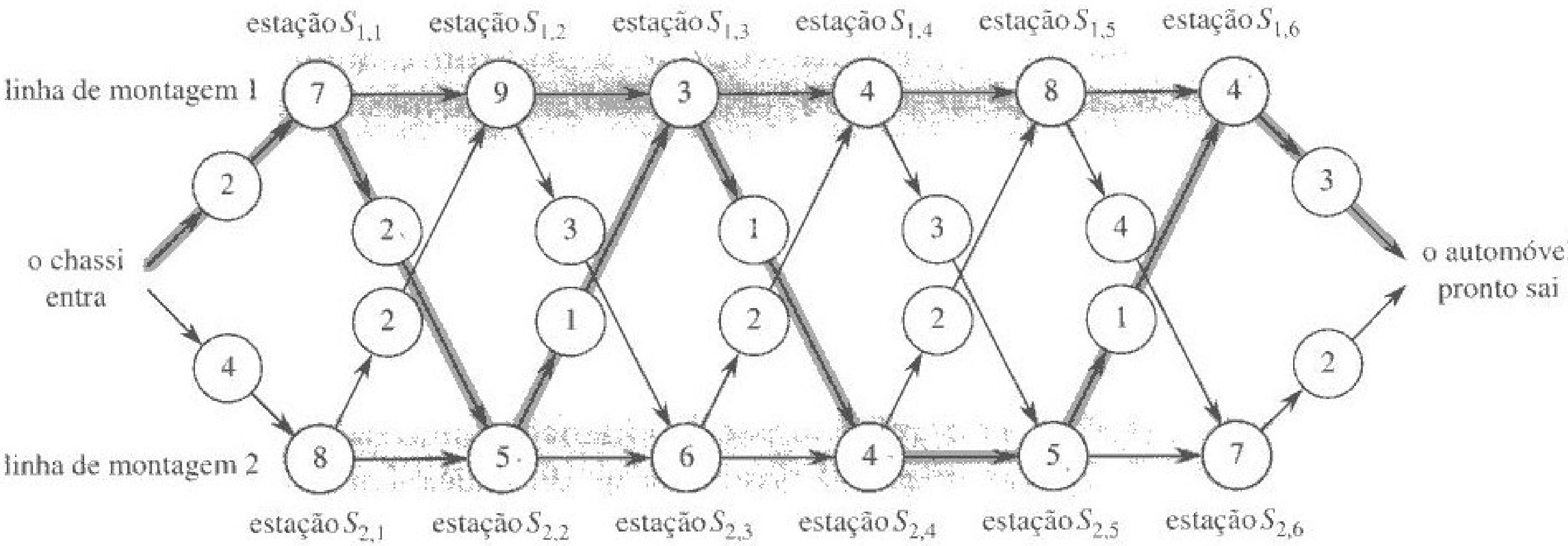
Cálculo dos tempos



(a)

	1	2	3	4	5	6
f1	9	18	20	24	32	35
f2	12	16	22	25	30	37

Cálculo dos tempos



(a)

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

$f^* = 38$

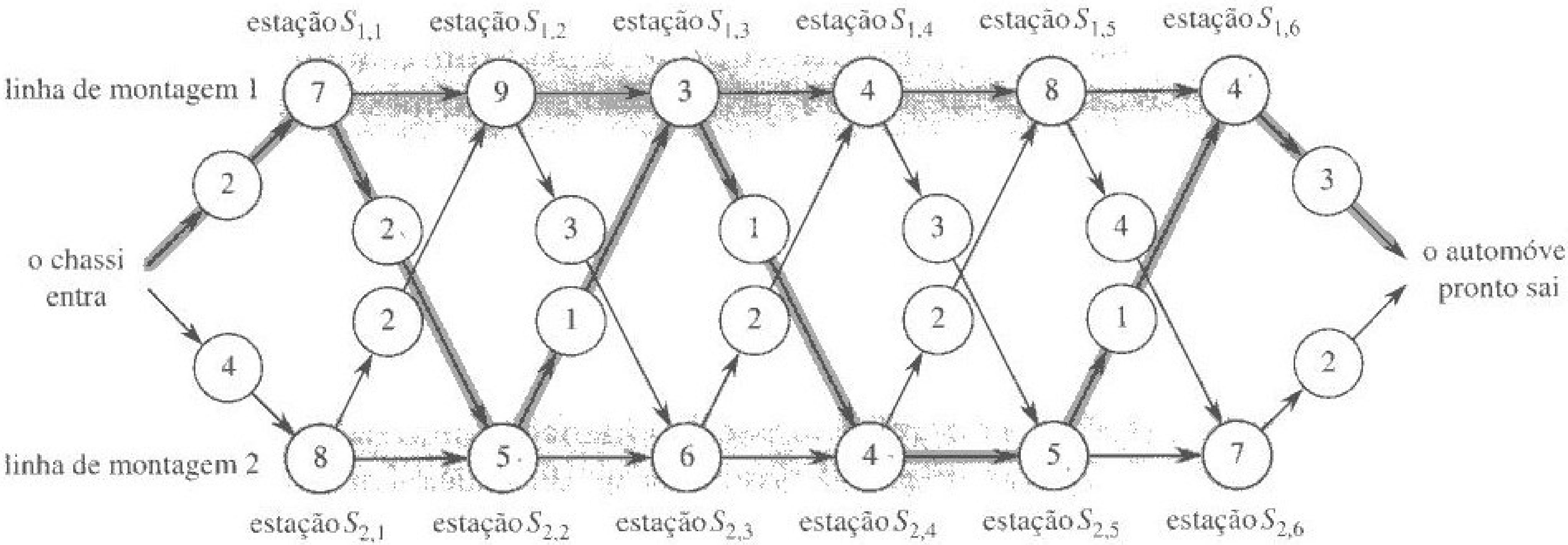
Cálculo dos tempos

```
fastest_way(n)

f1[1] = e1 + a1[1]
f2[1] = e2 + a2[1]

for j = 2 to n
     c1 = f1[j-1] + a1[j] // custo de continuar na linha 1
     c2 = f2[j-1] + t2[j-1] + a1[j] // custo de mudar da linha 2 p/ a 1
     if c1 <= c2
          f1[j] = c1
     else
          f1[j] = c2

     c1 = f2[j-1] + a2[j] // custo de continuar na linha 2
     c2 = f1[j-1] + t1[j-1] + a2[j] // custo de mudar da linha 1 p/ a 2
     if c1 <= c2
          f2[j] = c1
     else
          f2[j] = c2

if f1[n] + x1 <= f2[n] + x2
     f* = f1[n] + x1
else
     f* = f2[n] + x2
```

Etapa 4 – construção do caminho

- Já sabemos qual é o menor custo possível
- Só falta registrar qual o caminho seguido para se chegar até aquele custo
- Em cada passo do algoritmo vamos registrar a decisão de continuar ou de mudar linha

Construção do caminho



(a)

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

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

(b)

Construção do caminho

```
fastest_way(n)

f1[1] = e1 + a1[1]
f2[1] = e2 + a2[1]

for j = 2 to n
     c1 = f1[j-1] + a1[j] // custo de continuar na linha 1
     c2 = f2[j-1] + t2[j-1] + a1[j] // custo de mudar da linha 2 p/ a 1
     if c1 <= c2
          f1[j] = c1; l1[j] = 1
     else
          f1[j] = c2; l1[j] = 2

     c1 = f2[j-1] + a2[j] // custo de continuar na linha 2
     c2 = f1[j-1] + t1[j-1] + a2[j] // custo de mudar da linha 1 p/ a 2
     if c1 <= c2
          f2[j] = c1; l2[j] = 2
     else
          f2[j] = c2; l2[j] = 1

if f1[n] + x1 <= f2[n] + x2
     f* = f1[n] + x1; l* = 1
else
     f* = f2[n] + x2; l* = 2
```

Solução final

```
print-stations(l,n)
   i = l*
   print linha i, estação n
   for j = n downto 2
       i = l_i[j]
       print linha i estação j-1
```

1	2	1	1	2
1	2	1	2	2

l* = 1

Solução final

```
print-stations(l,n)
   i = l*
   print linha i, estação n
   for j = n downto 2
       i = l_i[j]
       print linha i estação j-1
```

2				6
1	2	1	1	2
1	2	1	2	2

l* = 1

i = 1

linha 1 estação 6

Solução final

```
print-stations(l,n)
   i = l*
   print linha i, estação n
   for j = n downto 2
       i = l_i[j]
       print linha i estação j-1
```

2				6	
1	2	1	1	2	l* = 1
1	2	1	2	2	

i = 1 linha 1 estação 6
i = 2 linha 2 estação 5

Solução final

```
print-stations(l,n)
   i = l*
   print linha i, estação n
   for j = n downto 2
       i = l_i[j]
       print linha i estação j-1
```

2				6
1	2	1	1	2
1	2	1	2	2

l* = 1

i = 1 linha 1 estação 6
i = 2 linha 2 estação 5
i = 2 linha 2 estação 4

Solução final

```
print-stations(l,n)
   i = l*
   print linha i, estação n
   for j = n downto 2
       i = l_i[j]
       print linha i estação j-1
```

2				6
1	2	1	1	2
1	2	1	2	2

l* = 1

i = 1	linha 1 estação 6
i = 2	linha 2 estação 5
i = 2	linha 2 estação 4
i = 1	linha 1 estação 3

Solução final

```
print-stations(l,n)
   i = l*
   print linha i, estação n
   for j = n downto 2
       i = l_i[j]
       print linha i estação j-1
```

2				6
1	2	1	1	2
1	2	1	2	2

l* = 1

i = 1	linha 1 estação 6
i = 2	linha 2 estação 5
i = 2	linha 2 estação 4
i = 1	linha 1 estação 3
i = 2	linha 2 estação 2

Solução final

```
print-stations(l,n)
   i = l*
   print linha i, estação n
   for j = n downto 2
       i = l_i[j]
       print linha i estação j-1
```

2				6
1	2	1	1	2
1	2	1	2	2

l* = 1

i = 1	linha 1 estação 6
i = 2	linha 2 estação 5
i = 2	linha 2 estação 4
i = 1	linha 1 estação 3
i = 2	linha 2 estação 2
i = 1	linha 1, estação 1

Exemplo 2 – multiplicação de cadeias de matrizes

- $A_{p \times q} \,.\, B_{q \times r} = ???$

Exemplo 2 – multiplicação de cadeias de matrizes

- $A_{p \times q} \cdot B_{q \times r} = C_{p \times r}$
- A e B são compatíveis, isso é, A tem o mesmo número de colunas que as linhas de B
- O número de multiplicações necessárias para multiplicar A e B é *???*

Exemplo 2 – multiplicação de cadeias de matrizes

- $A_{p \times q} \cdot B_{q \times r} = C_{p \times r}$
- A e B são compatíveis, isso é, A tem o mesmo número de colunas que as linhas de B
- O número de multiplicações necessárias para multiplicar A e B é pqr

Exemplo 2 – multiplicação de cadeias de matrizes

- $A_{p \times q} \cdot B_{q \times r} = C_{p \times r}$
- A e B são compatíveis, isso é, A tem o mesmo número de colunas que as linhas de B
- O número de multiplicações necessárias para multiplicar A e B é pqr
- Sejam A_1 10 X 100; A_2 100 X 5; e A_3 5 X 50
- Número de multiplicações para obter $A_1 A_2 A_3$????

Exemplo 2 – multiplicação de cadeias de matrizes

- $A_{p \, X \, q} \, . \, B_{q \, X \, r} = C_{p \, X \, r}$
- A e B são compatíveis, isso é, A tem o mesmo número de colunas que as linhas de B
- O número de multiplicações necessárias para multiplicar A e B é pqr
- Sejam A_1 10 X 100; A_2 100 X 5; e A_3 5 X 50
- Número de multiplicações
  - $((A_1 A_2) A_3)$:
  - $(A_1 (A_2 A_3))$:

Exemplo 2 – multiplicação de cadeias de matrizes

- $A_{p \times q} \,.\, B_{q \times r} = C_{p \times r}$
- A e B são compatíveis, isso é, A tem o mesmo número de colunas que as linhas de B
- O número de multiplicações necessárias para multiplicar A e B é pqr
- Sejam A_1 10 X 100; A_2 100 X 5; e A_3 5 X 50
- Número de multiplicações
  - $((A_1\, A_2)\, A_3)$: 7.500
  - $(A_1\, (A_2\, A_3))$: 75.000

Definição

- Dada a cadeia $<A_1, A_2, \ldots, A_n>$ de n matrizes na qual a matriz A_i tem dimensões $p_{i-1} \times p_i$, coloque completamente entre parênteses o produto $A_1 A_2 \ldots A_n$ de um modo que se minimize o número de multiplicações

- O número de alternativas para colocação de parênteses é $\Omega(2^{n-1})$

Definição

- Dada a cadeia $<A_1, A_2, \ldots, A_n>$ de n matrizes na qual a matriz A_i tem dimensões p_{i-1} X p_i , coloque completamente entre parênteses o produto $A_1 A_2 \ldots A_n$ de um modo que se minimize o número de multiplicações

A_1	A_2	A_3	A_4	A_5
p_0 x p_1	p_1 x p_2	p_2 x p_3	p_3 x p_4	p_4 x p_5

Etapa 1 – colocação ótima de parênteses

- $A_{i..j}$ é a matriz resultante de se multiplicarem as matrizes de i até j
- Se $i < j$ deve-se escolher um k, tal que $i \leq k < j$ para dividir o intervalo
- O custo vai ser o de multiplicar $A_{i..k}$ mais o de multiplicar $A_{k+1...j}$ mais o de multiplicar as duas matrizes resultantes

$$A_i A_{i+1} A_{i+2} \quad \ldots \quad A_{j-2} A_{j-1} A_j$$

$$(A_i A_{i+1} A_{i+2} \ldots A_k) \, (A_{k+1} \ldots A_{j-2} A_{j-1} A_j)$$

Etapa 1 – colocação ótima de parênteses

- $A_{i..j}$ é a matriz resultante de se multiplicarem as matrizes de i até j
- Se i < j deve-se escolher um k, tal que $i \leq k < j$ para dividir o intervalo
- O custo vai ser o de multiplicar $A_{i..k}$ mais o de multiplicar $A_{k+1...j}$ mais o de multiplicar as duas matrizes resultantes
- A solução ótima para $A_{i..j}$ requer que tenhamos uma solução ótima para $A_{i..k}$ e para $A_{k+1...j}$

Etapa 2 – Solução recursiva

- Vamos considerar que m[i,j] seja o custo (número de multiplicações) para se achar $A_{i..j}$
- Dividindo esse intervalo em 2, temos m[i,j] = m[i,k] + m[k+1,j] + custo de multiplicar as duas matrizes

Etapa 2 – Solução recursiva

- Vamos considerar que m[i,j] seja o custo (número de multiplicações) para se achar $A_{i..j}$
- Dividindo esse intervalo em 2, temos m[i,j] = m[i,k] + m[k+1,j] + custo de multiplicar as duas matrizes
- Qual é esse custo?

Etapa 2 – Solução recursiva

- Vamos considerar que m[i,j] seja o custo (número de multiplicações) para se achar $A_{i..j}$
- Dividindo esse intervalo em 2, temos m[i,j] = m[i,k] + m[k+1,j] + custo de multiplicar as duas matrizes

A_1	A_2	A_3	A_4	A_5
$p_0 \times p_1$	$p_1 \times p_2$	$p_2 \times p_3$	$p_3 \times p_4$	$p_4 \times p_5$

k = 3 $(A_1\ A_2\ A_3)\ (A_4\ A_5) \Longrightarrow p_0 \times p_3\ .\ p_3 \times p_5$ custo é $p_0\ p_3\ p_5$

Etapa 2 – Solução recursiva

- Vamos considerar que m[i,j] seja o custo (número de multiplicações) para se achar $A_{i..j}$
- Dividindo esse intervalo em 2, temos m[i,j] = m[i,k] + m[k+1,j] + custo de multiplicar as duas matrizes

A_1	A_2	A_3	A_4	A_5
$p_0 \times p_1$	$p_1 \times p_2$	$p_2 \times p_3$	$p_3 \times p_4$	$p_4 \times p_5$

k = 3 $(A_1 A_2 A_3)(A_4 A_5)$ ==> $p_0 \times p_3 \,.\, p_3 \times p_5$ custo é $p_0\ p_3\ p_5$

k = 2 $(A_1 A_2)(A_3 A_4 A_5)$ ==> $p_0 \times p_2 \,.\, p_2 \times p_5$ custo é $p_0\ p_2\ p_5$

Etapa 2 – Solução recursiva

- Vamos considerar que m[i,j] seja o custo (número de multiplicações) para se achar $A_{i..j}$
- Dividindo esse intervalo em 2, temos $m[i,j] = m[i,k] + m[k+1,j] +$ custo de multiplicar as duas matrizes

A_1	A_2	A_3	A_4	A_5
$p_0 \times p_1$				$p_4 \times p_5$

Custo é $p_{i-1}\, p_k\, p_j$

k = 3 … $\times p_3 \,.\, p_3 \times p_5$ custo é $p_0\, p_3\, p_5$

k = 2 … $p_0 \times p_2 \,.\, p_2 \times p_5$ custo é $p_0\, p_2\, p_5$

Etapa 2 – Solução recursiva

- Vamos considerar que m[i,j] seja o custo (número de multiplicações) para se achar $A_{i..j}$
- Dividindo esse intervalo em 2, temos m[i,j] = m[i,k] + m[k+1,j] + custo de multiplicar as duas matrizes

A_1	A_2	A_3	A_4	A_5
$p_0 \times p_1$	[illegible]	[illegible]	[illegible]	$p_4 \times p_5$

Custo é $p_{i-1}\ p_k\ p_j$

k = 3 … $\times p_3 \,.\, p_3 \times p_5$ custo é $p_0\ p_3\ p_5$

k = 2 … $p_0 \times p_2 \,.\, p_2 \times p_5$ custo é $p_0\ p_2\ p_5$

Etapa 2 – Solução recursiva

- Vamos considerar que m[i,j] seja o custo (número de multiplicações) para se achar $A_{i..j}$
- Dividindo esse intervalo em 2, temos
  m[i,j] = m[i,k] + m[k+1,j] + custo de multiplicar as duas matrizes
- m[i,j] = m[i,k] + m[k+1,j] + $p_{i-1}\, p_k\, p_j$
- Se i = j então m[i,j] = 0
- Se i < j temos que achar o melhor valor de k (existem j-i possibilidades) que faça com que m[i,j] seja o menor possível
- Esse valor de k fica guardado em s[i,j]

Etapa 3 – custos ótimos

- Vamos usar um array p[0..n], que contém as dimensões de cada matriz
- m[1..n, 1..n] armazena os custos de m[i,j]
- s[1..n, 1..n] registra qual índice k ótimo foi usado no cálculo de m[i,j]

Tabelas

m armazena os custos de m[i,j]

custo para multiplicar $A_2 \ . \ . \ . \ A_5$

m

	1	2	3	4	5	6
1						
2					7125	
3						
4						
5						
6						

s registra qual índice k ótimo foi usado no cálculo de m[i,j]

Esse custo foi obtido fazendo-se
$(A_1A_2A_3)\ (A_4A_5)$

s

	1	2	3	4	5	6
1						
2					3	
3						
4						
5						
6						

p

30	35	15	5	10	20	25
0						6

Matrizes 30x35; 35x15; 15x5; 5x10; 10x20; 20x25

Computar os custos

```
matrix-chain-order(p)
   n = comprimento[p] - 1
   for i = 1 to n
      m[i,i] = 0
   for l = 2 to n
      for i = 1 to n - l + 1
          j = i + l - 1
          m[i,j] = infinito
          for k = i to j -1
             q = m[i,k] + m[k+1,j]
              + p[i-1] . p[k] . p[j]
             if q < m[i,j]
                 m[i,j] = q
                 s[i,j] = k

   return m e s
```

Computar os custos

```
matrix-chain-order(p)
   n = comprimento[p] - 1
   for i = 1 to n
      m[i,i] = 0
   for l = 2 to n
      for i = 1 to n - l + 1
          j = i + l - 1
          m[i,j] = infinito
          for k = i to j -1
             q = m[i,k] + m[k+1,j]
              + p[i-1] . p[k] . p[j]
             if q < m[i,j]
                 m[i,j] = q
                 s[i,j] = k

   return m e s
```

m

1				
2				
3				
4				
5				
6				

1	2	3	4	5	6

s

1				
2				
3				
4				
5				
6				

30	35	15	5	10	20	25
0						6

Computar os custos

```
matrix-chain-order(p)
   n = comprimento[p] - 1
   for i = 1 to n
      m[i,i] = 0
   for l = 2 to n
      for i = 1 to n - l + 1
          j = i + l - 1
          m[i,j] = infinito
          for k = i to j -1
             q = m[i,k] + m[k+1,j]
              + p[i-1] . p[k] . p[j]
             if q < m[i,j]
                 m[i,j] = q
                 s[i,j] = k

   return m e s
```

m

1	0					
2		0				
3			0			
4				0		
5					0	
6						0
	1	2	3	4	5	6

s

1						
2						
3						
4						
5						
6						

30	35	15	5	10	20	25
0						6

Computar os custos

```
matrix-chain-order(p)
   n = comprimento[p] - 1
   for i = 1 to n
      m[i,i] = 0
   for l = 2 to n
      for i = 1 to n - l + 1
          j = i + l - 1
          m[i,j] = infinito
          for k = i to j -1
             q = m[i,k] + m[k+1,j]
              + p[i-1] . p[k] . p[j]
             if q < m[i,j]
                 m[i,j] = q
                 s[i,j] = k

   return m e s
```

m

1	0	???				
2		0				
3			0			
4				0		
5					0	
6						0

1	2	3	4	5	6

s

1						
2						
3						
4						
5						
6						

30	35	15	5	10	20	25

0 6

m[1,2] = m[1,1] + m[2,2] + p[0] . p[1] . p[2] =

Computar os custos

```
matrix-chain-order(p)
   n = comprimento[p] - 1
   for i = 1 to n
      m[i,i] = 0
   for l = 2 to n
      for i = 1 to n - l + 1
          j = i + l - 1
          m[i,j] = infinito
          for k = i to j -1
             q = m[i,k] + m[k+1,j]
              + p[i-1] . p[k] . p[j]
             if q < m[i,j]
                 m[i,j] = q
                 s[i,j] = k

   return m e s
```

m

	1	2	3	4	5	6
1	0	15750				
2		0				
3			0			
4				0		
5					0	
6						0

s

	1	2	3	4	5	6
1		2				
2						
3						
4						
5						
6						

0						6
30	35	15	5	10	20	25

m[1,2] = m[1,1] + m[2,2] + p[0] . p[1] . p[2] = 15750

Computar os custos

```
matrix-chain-order(p)
   n = comprimento[p] - 1
   for i = 1 to n
      m[i,i] = 0
   for l = 2 to n
      for i = 1 to n - l + 1
          j = i + l - 1
          m[i,j] = infinito
          for k = i to j -1
             q = m[i,k] + m[k+1,j]
              + p[i-1] . p[k] . p[j]
             if q < m[i,j]
                 m[i,j] = q
                 s[i,j] = k

   return m e s
```

m

	1	2	3	4	5	6
1	0	15750				
2		0	???			
3			0			
4				0		
5					0	
6						0

s

	1	2	3	4	5	6
1		2				
2						
3						
4						
5						
6						

30	35	15	5	10	20	25

0 6

m[2,3] = m[2,2] + m[3,3] + p[1] . p[2] . p[3] =

Computar os custos

```
matrix-chain-order(p)
   n = comprimento[p] - 1
   for i = 1 to n
      m[i,i] = 0
   for l = 2 to n
      for i = 1 to n - l + 1
          j = i + l - 1
          m[i,j] = infinito
          for k = i to j -1
             q = m[i,k] + m[k+1,j]
              + p[i-1] . p[k] . p[j]
             if q < m[i,j]
                 m[i,j] = q
                 s[i,j] = k

   return m e s
```

m

	1	2	3	4	5	6
1	0	15750				
2		0	2625			
3			0			
4				0		
5					0	
6						0

s

	1	2	3	4	5	6
1		1				
2			2			
3						
4						
5						
6						

30	35	15	5	10	20	25

0 6

m[2,3] = m[2,2] + m[3,3] + p[1] . p[2] . p[3] = 2625

Computar os custos

```
matrix-chain-order(p)
   n = comprimento[p] - 1
   for i = 1 to n
      m[i,i] = 0
   for l = 2 to n
      for i = 1 to n - l + 1
          j = i + l - 1
          m[i,j] = infinito
          for k = i to j -1
             q = m[i,k] + m[k+1,j]
              + p[i-1] . p[k] . p[j]
             if q < m[i,j]
                 m[i,j] = q
                 s[i,j] = k

   return m e s
```

m

	1	2	3	4	5	6
1	0	15750				
2		0	2625			
3			0	750		
4				0	1000	
5					0	5000
6						0

s

	1	2	3	4	5	6
1		1				
2			2			
3				3		
4					4	
5						5
6						

0						6
30	35	15	5	10	20	25

Computar os custos

```
matrix-chain-order(p)
   n = comprimento[p] - 1
   for i = 1 to n
      m[i,i] = 0
   for l = 2 to n
      for i = 1 to n - l + 1
          j = i + l - 1
          m[i,j] = infinito
          for k = i to j -1
             q = m[i,k] + m[k+1,j]
              + p[i-1] . p[k] . p[j]
             if q < m[i,j]
                 m[i,j] = q
                 s[i,j] = k

   return m e s
```

m

	1	2	3	4	5	6
1	0	15750	???			
2		0	2625			
3			0	750		
4				0	1000	
5					0	5000
6						0

s

	1	2	3	4	5	6
1		1				
2			2			
3				3		
4					4	
5						5
6						

30	35	15	5	10	20	25
0						6

Computar os custos

m[1,3] = mínimo entre:

$(A_1 (A_2 A_3))$ ==> k = 1
m[1,1] + m[2,3] + p[0] . p[1] . p[3]

$(A_1 A_2) A_3)$ ==> k = 2
m[1,2] + m[3,3] + p[0] . p[2] . p[3]

m

	1	2	3	4	5	6
1	0	15750	???			
2		0	2625			
3			0	750		
4				0	1000	
5					0	5000
6						0

s

	1	2	3	4	5	6
1		1				
2			2			
3				3		
4					4	
5						5
6						

30	35	15	5	10	20	25

0   6

Computar os custos

m[1,3] = mínimo entre:

m[1,1] + m[2,3] + p[0] . p[1] . p[3] = 7875

m[1,2] + m[3,3] + p[0] . p[2] . p[3] = 16205

m

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625			
3			0	750		
4				0	1000	
5					0	5000
6						0

s

	1	2	3	4	5	6
1		1	1			
2			2			
3				3		
4					4	
5						5
6						

30	35	15	5	10	20	25
0						6

Computar os custos

```
matrix-chain-order(p)
   n = comprimento[p] - 1
   for i = 1 to n
      m[i,i] = 0
   for l = 2 to n
      for i = 1 to n - l + 1
          j = i + l - 1
          m[i,j] = infinito
          for k = i to j -1
             q = m[i,k] + m[k+1,j]
              + p[i-1] . p[k] . p[j]
             if q < m[i,j]
                 m[i,j] = q
                 s[i,j] = k

   return m e s
```

m

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

s

	1	2	3	4	5	6
1		1	1	3	3	3
2			2	3	3	3
3				3	3	3
4					4	5
5						5
6						

30	35	15	5	10	20	25
0						6

Etapa 4 – solução ótima

- Falta mostrar como as matrizes devem ser acomodadas entre parênteses
- Para isso, usamos a matriz s
- s[i,j] = k , indica que para multiplicar $A_{i,j}$ devemos quebrar em $A_{i,k}$ e $A_{k+1,J}$
- No exemplo, s[1,6] = 3 então temos
  - $((A_{1,3}) (A_{4,6}))$

Solução ótima

- Mas como fazemos para multiplicar otimamente $A_{1,3}$ e $A_{4,6}$?
- Basta olhar em s[1,3] = 1 e s[4,6] = 5
  - $(A_{1,3}) \rightarrow (A_1 (A_{2,3}))$
  - $(A_{4,6}) \rightarrow ((A_{4,5}) A_6)$

Solução ótima

- Mas como fazemos para multiplicar otimamente $A_{1,3}$ e $A_{4,6}$?
- Basta olhar em s[1,3] = 1 e s[4,6] = 5
  - $(A_{1,3}) \rightarrow (A_1 (A_{2,3}))$
  - $(A_{4,6}) \rightarrow ((A_{4,5}) A_6)$
- E depois só falta $A_{2,3}$ e $A_{4,5}$

Solução ótima

- Para isso basta um simples algoritmo recursivo, que recebe s, i e j como parâmetros

```
print-parens(s, i, j)
```

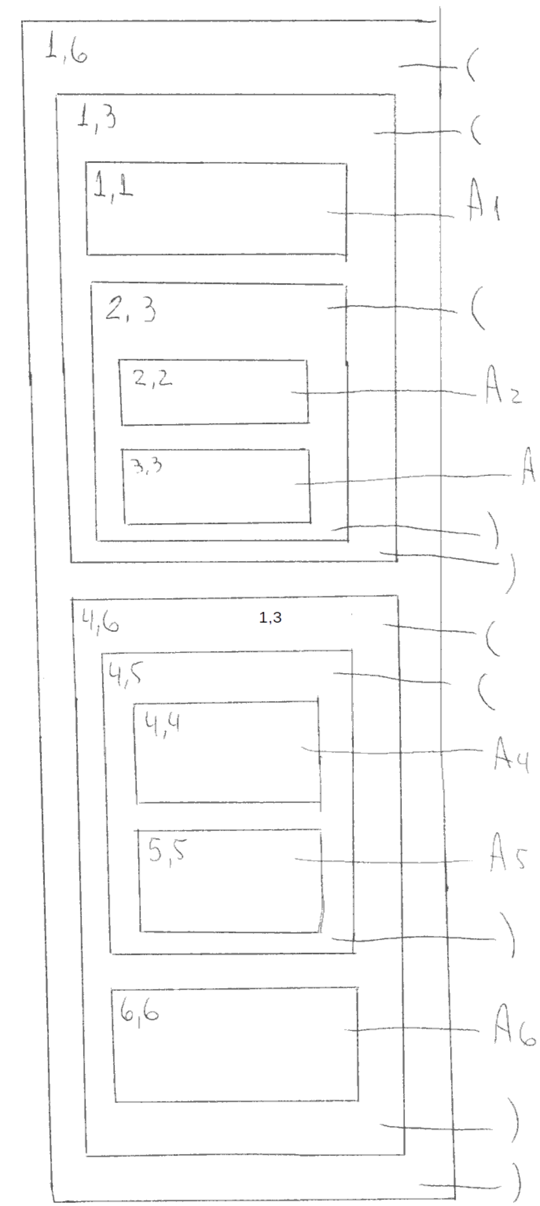
Solução ótima

- Para isso basta um simples algoritmo recursivo, que recebe s, i e j como parâmetros

```
print-parens(s, i, j)

if i = j
   print A_i
else
   print “(“
   print-parens(s, i, s[i,j])
   print-parens(s, s[i,j]+1, j)
   print “)”
```

Solução ótima

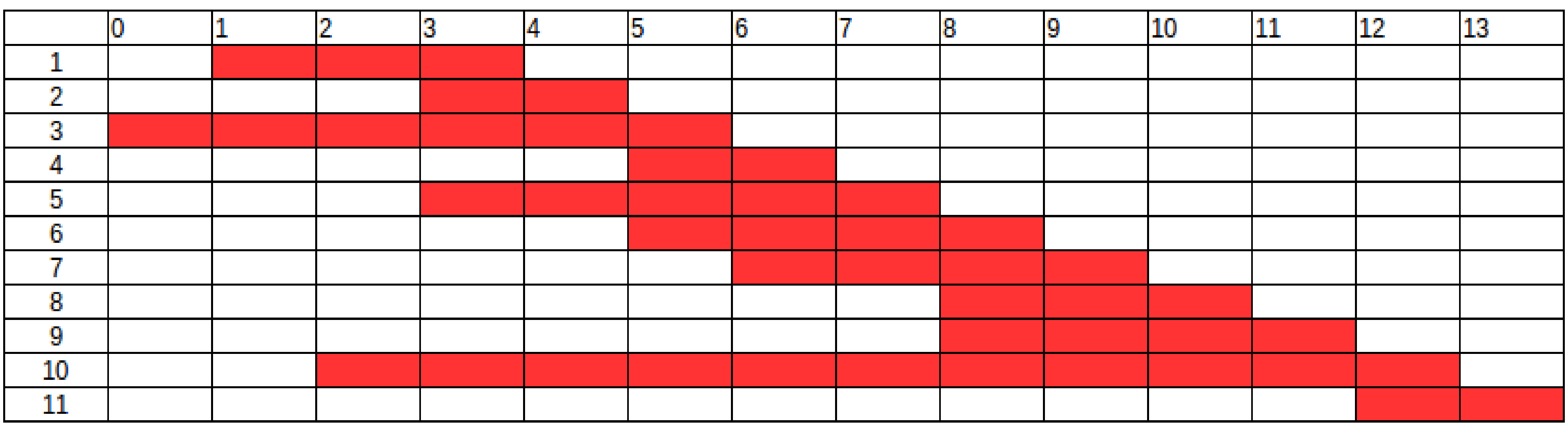
```
print-parens(s, i, j)

if i = j
   print A_i
else
   print "("
   print-parens(s, i, s[i,j])
   print-parens(s, s[i,j]+1, j)
   print ")"
```



	1	1	3	3	3
		2	3	3	3
			3	3	3
				4	5
					5

Alocação de recursos

- 11 atividades a serem distribuídas em 14 unidades de tempo;
- queremos selecionar um conjunto máximo de atividades que não têm sobreposição de tempo.



O problema

- $S = \{a_1, a_2, \ldots, a_n\}$
- Cada atividade i tem um tempo de início s_i e um tempo de fim f_i
- A atividade a_i ocorre no intervalo $[s_i, f_i)$
- a_i e a_j são compatíveis se seus intervalos não se sobrepõem
-

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

O problema

- $S = \{a_1, a_2, \ldots, a_n\}$
- Cada atividade i tem
  um tempo de fim f_i
- A atividade a_i ocorre no lo $[s_i,f_i)$
- a_i e a_j são compatíveis se us intervalos não se sobrepõem
-

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Selecionar um subconjunto de tamanho máximo de atividades mutuamente compatíveis

Subestrutura ótima

- $S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$
  S_{ij} é o conjunto de atividades em S que podem começar após a atividade a_i terminar e antes da atividade a_j começar
- Adicionamos
  - $a_0 \rightarrow f_0 = 0$
  - $a_{n+1} \rightarrow s_{n+1} = \infty$
- Portanto $S_{o,\, n+1}$ é a solução para o problema que queremos

Subestrutura ótima

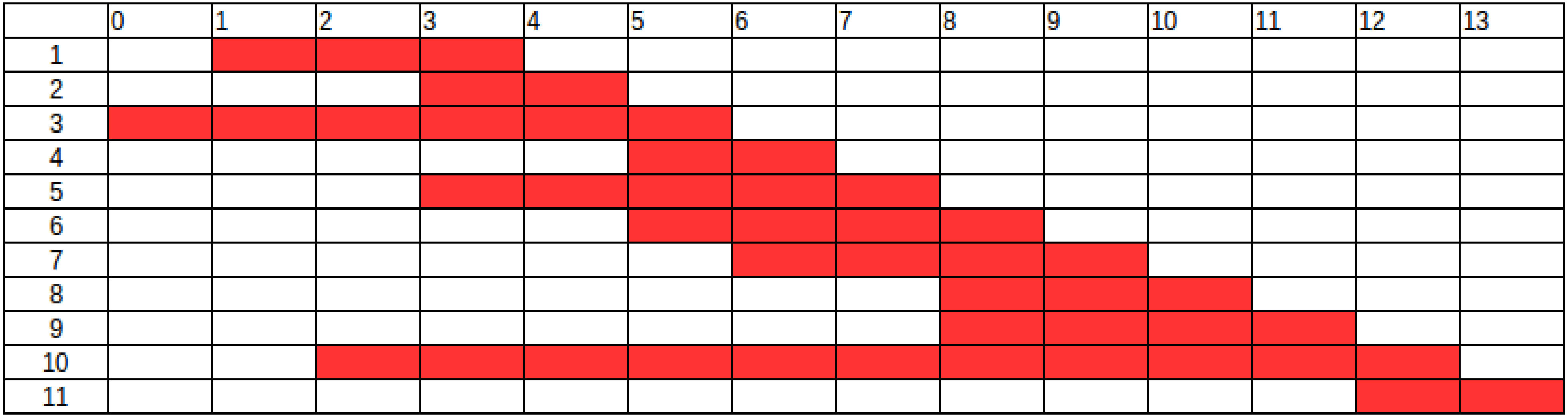
- Para resolver o problema para S_{ij} vamos adicionar uma atividade a_k
- Com isso dividimos o nosso problema em dois pois teremos agora dois subconjuntos:
- $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

Solução recursiva

- Seja c[i,j] o número (máximo) de atividades em S_{ij}
- Temos que c[i,j] = 0 sempre que $S_{ij} = \theta$ em particular, quando $i \geq j$
- Caso contrário c[i,j] = c[i,k]+c[k,j]+1 para o melhor valor possível de k

Vamos resolver?

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



Vamos resolver?

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

```
para d = 2 ate N+1
   i = 0
   enquanto i + d <= N+1
      j = i + d
      k = melhor valor para dividir S_ij
      se k não existe
           c[i][j] = 0
      senão
            c[i][j] = c[i][k] + c[k][j] + 1
      i++
```

Tabela de custos

	0	1	2	3	4	5	6	7	8	9	10	11	12
0		0	0	0	1	0	1	1	2	2	0	3	4
1			0	0	0	0	0	0	1	1	0	2	3
2				0	0	0	0	0	1	1	0	2	3
3					0	0	0	0	0	0	0	1	2
4						0	0	0	0	0	0	1	2
5							0	0	0	0	0	1	2
6								0	0	0	0	0	1
7									0	0	0	0	1
8										0	0	0	1
9											0	0	1
10												0	0
11													0
12													

Atividades escolhidas

- Falta definir quais atividades foram as escolhidas

```
para d = 2 ate N+1
   i = 0
   enquanto i + d <= N+1
      j = i + d
      k = melhor valor para dividir S_ij
      se k não existe
           c[i][j] = 0
      senão
            c[i][j] = c[i][k] + c[k][j] + 1
            l[i][j] = k
      i++
```

Atividades escolhidas

	0	1	2	3	4	5	6	7	8	9	10	11	12
0		0	0	0	1	0	1	1	1	1	0	1	1
1			0	0	0	0	0	0	4	4	0	4	4
2				0	0	0	0	0	4	4	0	4	4
3					0	0	0	0	0	0	0	7	7
4						0	0	0	0	0	0	8	8
5							0	0	0	0	0	8	8
6								0	0	0	0	0	11
7									0	0	0	0	11
8										0	0	0	11
9											0	0	11
10												0	0
11													0
12													

Atividades escolhidas

```
print_sol( i, j )
    k = l[i][j]
    if k == 0 retorna
    print k
    print_sol(i,k)
    print_sol(k,j)
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
0		0	0	0	1	0	1	1	1	1	0	1	1
			0	0	0	0	0	0	4	4	0	4	4
				0	0	0	0	0	4	4	0	4	4
					0	0	0	0	0	0	0	7	7
						0	0	0	0	0	0	8	8
							0	0	0	0	0	8	8
6								0	0	0	0	0	11
7									0	0	0	0	11
8										0	0	0	11
9											0	0	11
10												0	0
11													0
12													

Atividades escolhidas

```
print_sol( i, j )
    k = l[i][j]
    if k == 0 retorna
    print k
    print_sol(i,k)
    print_sol(k,j)
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
0		0	0	0	1	0	1	1	1	1	0	1	1
			0	0	0	0	0	0	4	4	0	4	4
				0	0	0	0	0	4	4	0	4	4
					0	0	0	0	0	0	0	7	7
						0	0	0	0	0	0	8	8
							0	0	0	0	0	8	8
6								0	0	0	0	0	11
7									0	0	0	0	11
8										0	0	0	11
9											0	0	11
10												0	0
11													0
12													

print_sol(0,12)

1 4 8 11

Atividades escolhidas