

Programação Dinâmica

Prof. Anderson Almeida Ferreira

Adaptado do material elaborado por Andrea
Iabrudi Tavares

Programação Dinâmica

- 1950, Bellman
- Evitar recálculos dos subproblemas em comum
 - Menor para maior (bottom-up)
 - Tabelas ou memorização
- É uma técnica de programação
- Foi desenvolvida na época em que “programação” significava “método tabular”.
- Usada para problemas de otimização
 - Encontre a solução a com o valor ótimo.
 - Minimizar ou maximizar

Programação Dinâmica

- **Quatro passos do método**
 - Caracterize a estrutura de uma solução ótima.
 - Recursivamente defina o valor de uma solução ótima.
 - Compute o valor de uma solução ótima de maneira bottom-up.
 - Construa a solução ótima por meio da informação computada.

Fibonacci: definindo recorrência

```
RECFIBO( $n$ ):  
  if ( $n < 2$ )  
    return  $n$   
  else  
    return RECFIBO( $n - 1$ ) + RECFIBO( $n - 2$ )
```

- **Grafo de recorrência**
 - Subproblemas – nós
 - Dependência – arestas

Fibonacci: definindo recursão

RECFIBO(n):

```
if ( $n < 2$ )
    return  $n$ 
else
    return RECFIBO( $n - 1$ ) + RECFIBO( $n - 2$ )
```

- Grafo de recorrência
 - Subproblemas – nós
 - Dependência – arestas

MEMFIBO(n):

```
if ( $n < 2$ )
    return  $n$ 
else
    if  $F[n]$  is undefined
         $F[n] \leftarrow$  MEMFIBO( $n - 1$ ) + MEMFIBO( $n - 2$ )
    return  $F[n]$ 
```

- **Memorização**

Fibonacci: definindo recursão

RECFIBO(n):

```
if ( $n < 2$ )
    return  $n$ 
else
    return RECFIBO( $n - 1$ ) + RECFIBO( $n - 2$ )
```

- Grafo de recorrência
 - Subproblemas – nós
 - Dependência – arestas

MEMFIBO(n):

```
if ( $n < 2$ )
    return  $n$ 
else
    if  $F[n]$  is undefined
         $F[n] \leftarrow$  MEMFIBO( $n - 1$ ) + MEMFIBO( $n - 2$ )
    return  $F[n]$ 
```

- **Memorização**

Fibonacci: usando tabela

RECFIBO(n):

```
if ( $n < 2$ )
    return  $n$ 
else
    return RECFIBO( $n - 1$ ) + RECFIBO( $n - 2$ )
```

MEMFIBO(n):

```
if ( $n < 2$ )
    return  $n$ 
else
    if  $F[n]$  is undefined
         $F[n] \leftarrow$  MEMFIBO( $n - 1$ ) + MEMFIBO( $n - 2$ )
    return  $F[n]$ 
```

ITERFIBO(n):

```
 $F[0] \leftarrow 0$ 
 $F[1] \leftarrow 1$ 
for  $i \leftarrow 2$  to  $n$ 
     $F[i] \leftarrow F[i - 1] + F[i - 2]$ 
return  $F[n]$ 
```

- Grafo de recorrência
 - Subproblemas – nós
 - Dependência – arestas
- Memorização

Fibonacci: usando tabela

RECFIBO(n):

```
if ( $n < 2$ )
    return  $n$ 
else
    return RECFIBO( $n - 1$ ) + RECFIBO( $n - 2$ )
```

MEMFIBO(n):

```
if ( $n < 2$ )
    return  $n$ 
else
    if  $F[n]$  is undefined
         $F[n] \leftarrow$  MEMFIBO( $n - 1$ ) + MEMFIBO( $n - 2$ )
    return  $F[n]$ 
```

ITERFIBO(n):

```
 $F[0] \leftarrow 0$ 
 $F[1] \leftarrow 1$ 
for  $i \leftarrow 2$  to  $n$ 
     $F[i] \leftarrow F[i - 1] + F[i - 2]$ 
return  $F[n]$ 
```

- Grafo de recorrência
 - Subproblemas – nós
 - Dependência – arestas
- Memorização
- **Tabela**
 - Ordenação parcial

Fibonacci: economizando espaço

RECFIBO(n):

```
if ( $n < 2$ )
    return  $n$ 
else
    return RECFIBO( $n - 1$ ) + RECFIBO( $n - 2$ )
```

MEMFIBO(n):

```
if ( $n < 2$ )
    return  $n$ 
else
    if  $F[n]$  is undefined
         $F[n] \leftarrow$  MEMFIBO( $n - 1$ ) + MEMFIBO( $n - 2$ )
    return  $F[n]$ 
```

ITERFIBO(n):

```
 $F[0] \leftarrow 0$ 
 $F[1] \leftarrow 1$ 
for  $i \leftarrow 2$  to  $n$ 
     $F[i] \leftarrow F[i - 1] + F[i - 2]$ 
return  $F[n]$ 
```

ITERFIBO2(n):

```
prev  $\leftarrow 1$ 
curr  $\leftarrow 0$ 
for  $i \leftarrow 1$  to  $n$ 
    next  $\leftarrow$  curr + prev
    prev  $\leftarrow$  curr
    curr  $\leftarrow$  next
return curr
```

- Grafo de recorrência
 - Subproblemas – nós
 - Dependência – arestas

- Memorização

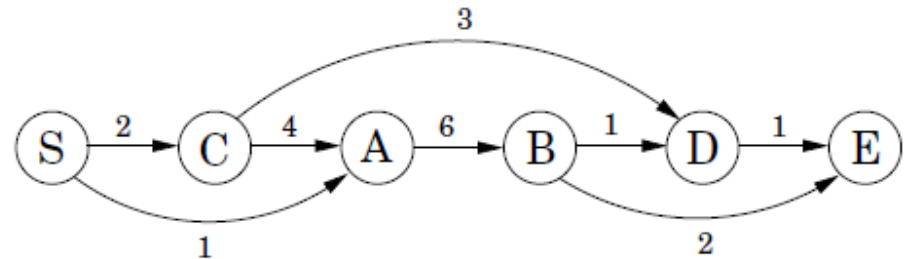
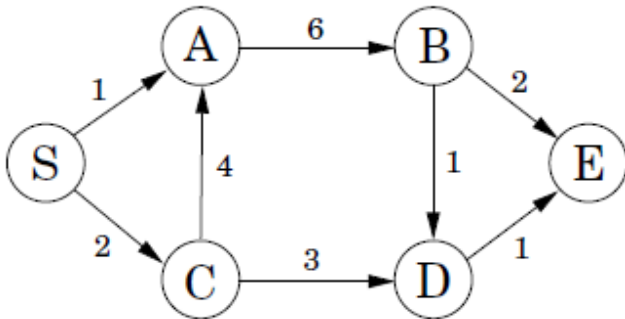
Tabela

- Ordenação parcial
- Economizando memória

Problemas alvo para Programação Dinâmica (PD)

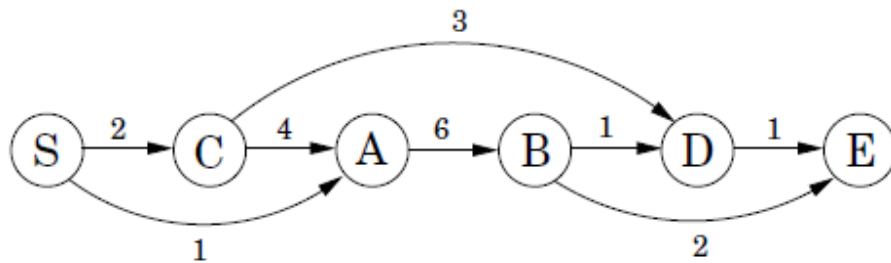
- Problema pode ser dividido em subproblemas menores.
- Sub-estrutura ótima (princípio da otimalidade)
 - Solução **ótima** do problema inclui soluções **ótimas** dos subproblemas.
- Subproblemas **são** sobrepostos.
 - Número “pequeno” de subproblemas distintos.

Linearização de Grafos Direcionados Acíclicos (DAGs)



Algoritmo de Linearização:
Percorrer vértices na ordem de grau de entrada.
Diminuir das arestas a cada passo.

Recorrência a partir de DAGs



Algoritmo de menor caminho em DAGs

```
initialize all  $\text{dist}(\cdot)$  values to  $\infty$   
 $\text{dist}(s) = 0$   
for each  $v \in V \setminus \{s\}$ , in linearized order:  
   $\text{dist}(v) = \min_{(u,v) \in E} \{\text{dist}(u) + l(u,v)\}$ 
```

Subproblemas
Menores

Subestrutura
Ótima

Subsequência Crescente Mais Longa

- Problema: Dada uma sequência de números naturais, definir qual a subsequência crescente com mais elementos.

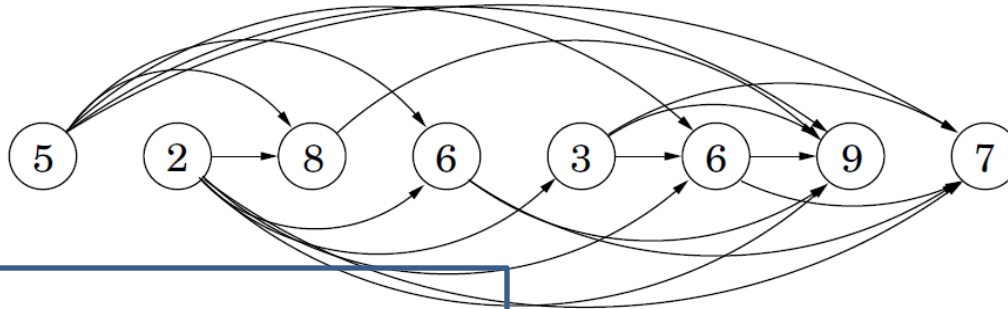
5 2 8 6 3 6 9 7

MSC: Recorrência e DAG implícito

5 2 8 6 3 6 9 7

SCML: Recorrência

Etapa1:



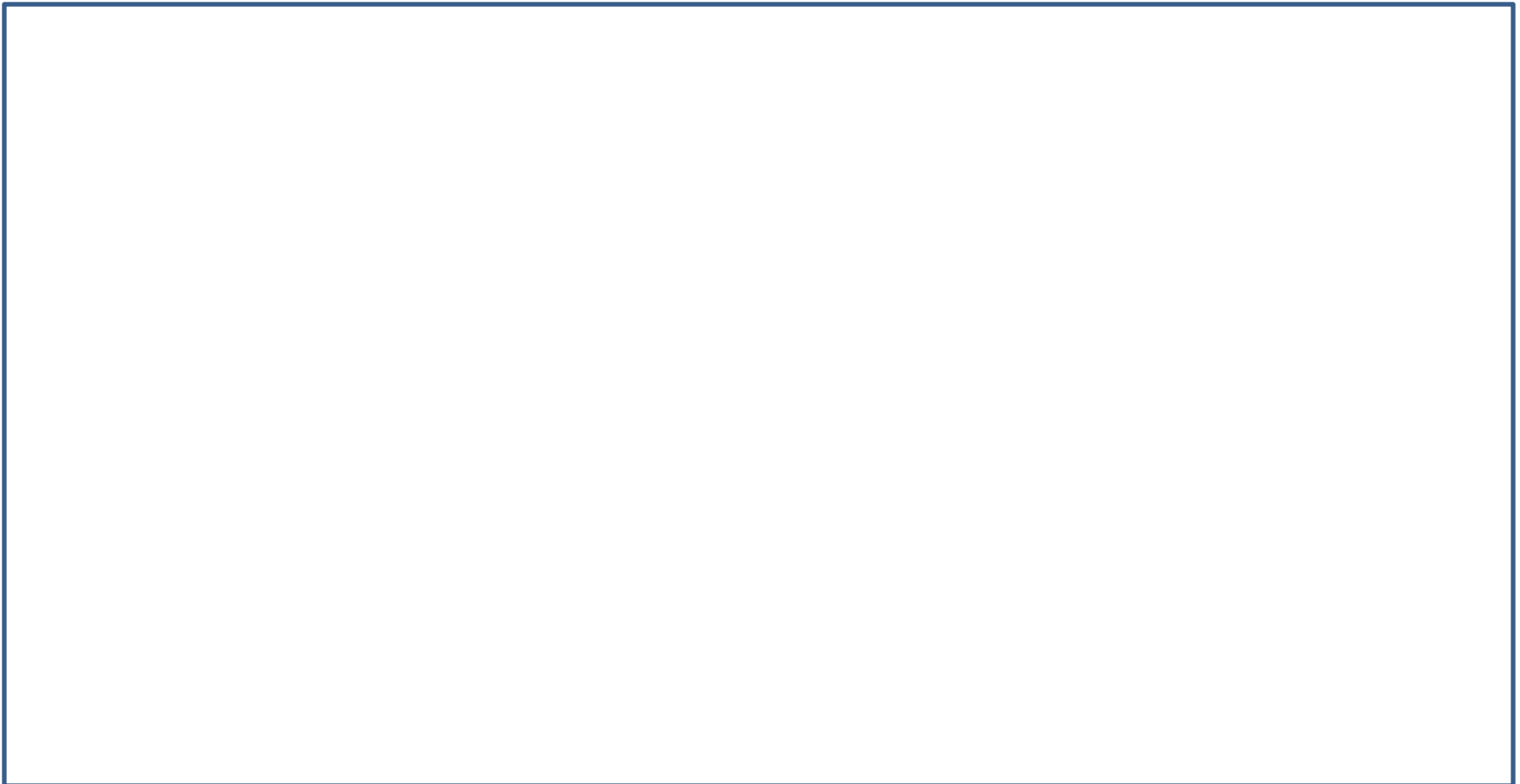
$$L(j) = 1 + \max\{L(i) : (i, j) \in E\}$$

Maior caminho em DAG

$$SCML = \max_{1 \leq j \leq n} \{L(j)\}$$

Etapa 2: Algoritmo Recursivo

$$L(j) = 1 + \max\{L(i) : (i, j) \in E\}$$



Etapa 3: Algoritmo Iterativo (Tabela)

- Vetor L, preenchido da menor posição para maior.



Etapa 3: Complexidade

```
for  $j = 1, 2, \dots, n$ :  
     $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$   
return  $\max_j L(j)$ 
```



Etapa 4: Construindo solução

```
for  $j = 1, 2, \dots, n$ :  
     $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$   
return  $\max_j L(j)$ 
```

5 2 8 6 3 6 9 7

Distância de Edição

- Transformar uma sequência em outra ao menor custo.
 - Casamento, substituição, inserção, remoção.

SITUADO

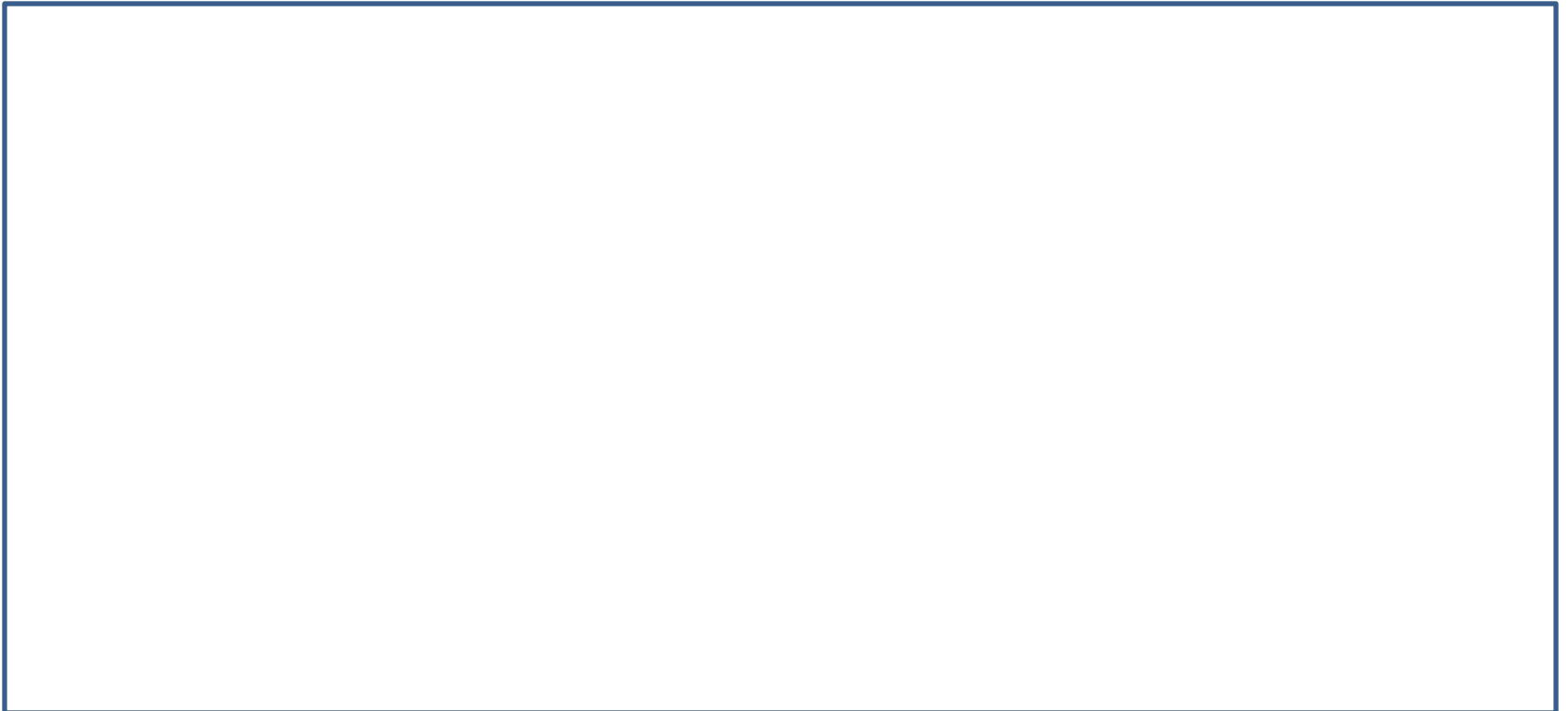
ESTUDO-

-SITUADO

ES-TU-DO

Edição: Subproblemas

- Problema: Alinhar duas sequências de caracteres $x[1 \cdots m]$ $y[1 \cdots n]$ $E(m, n)$



Etapa 1: Equação de Recorrência

- Problema: Alinhar duas sequências de caracteres $x[1 \cdots m]$ $y[1 \cdots n]$ $E(m, n)$

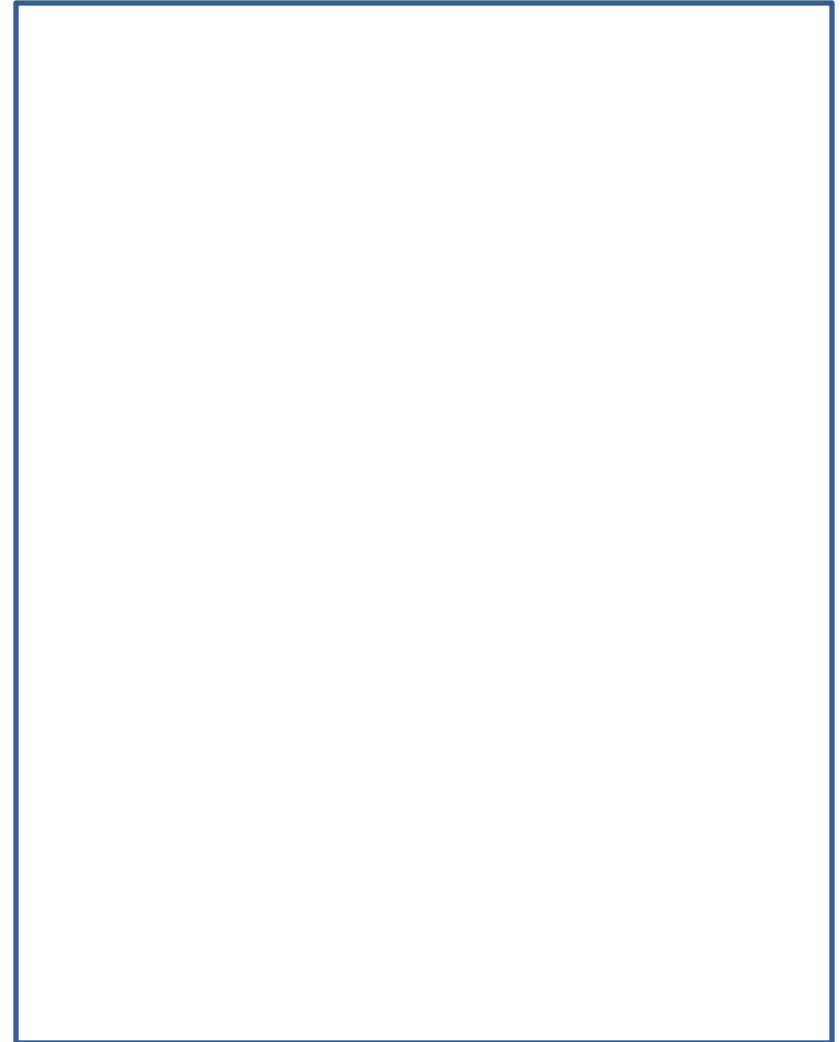
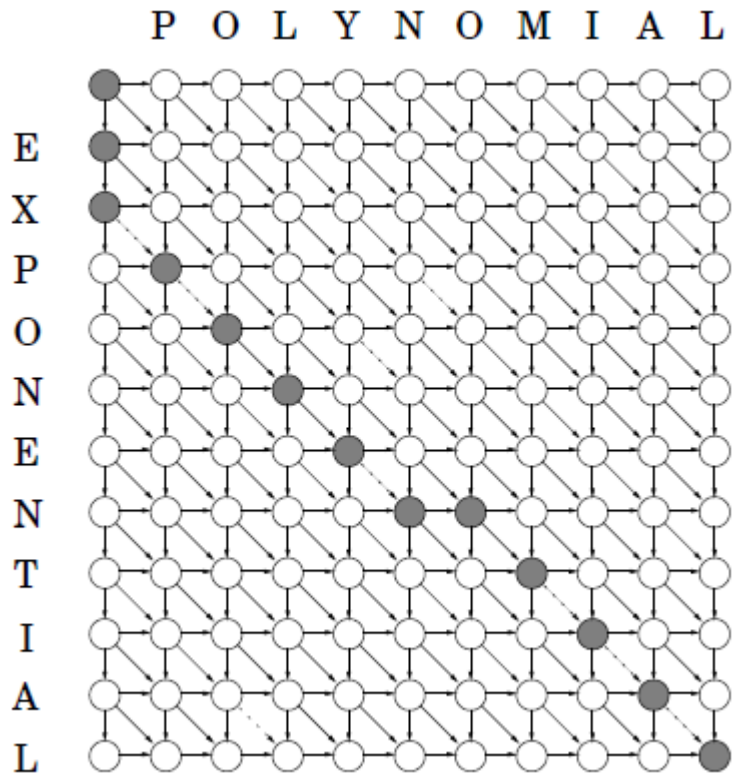
- Subproblema: alinhamento de prefixos

$$E(i, j) \quad \begin{array}{c} x[i] \\ - \end{array} \quad \text{or} \quad \begin{array}{c} - \\ y[j] \end{array} \quad \text{or} \quad \begin{array}{c} x[i] \\ y[j] \end{array}$$

- Composição: inserir, remover, casar

$$E(i, j) = \min\{1 + E(i - 1, j), 1 + E(i, j - 1), \text{diff}(i, j) + E(i - 1, j - 1)\}$$

Distância de Edição - DAG



Etapa 2: Algoritmo Recursivo

$$E(i, j) = \min\{1 + E(i - 1, j), 1 + E(i, j - 1), \text{diff}(i, j) + E(i - 1, j - 1)\}$$



Etapa 3: Algoritmo Iterativo

$$E(i, j) = \min\{1 + E(i - 1, j), 1 + E(i, j - 1), \text{diff}(i, j) + E(i - 1, j - 1)\}$$

Etapa 3: Complexidade

```
for  $i = 0, 1, 2, \dots, m$ :  
     $E(i, 0) = i$   
for  $j = 1, 2, \dots, n$ :  
     $E(0, j) = j$   
for  $i = 1, 2, \dots, m$ :  
    for  $j = 1, 2, \dots, n$ :  
         $E(i, j) = \min\{E(i - 1, j) + 1, E(i, j - 1) + 1, E(i - 1, j - 1) + \text{diff}(i, j)\}$   
return  $E(m, n)$ 
```

- $\Theta(mn)$ de tempo e espaço

Etapa 4: Solução I – armazenar

```
for  $i = 0, 1, 2, \dots, m$ :  
     $E(i, 0) = i$   
for  $j = 1, 2, \dots, n$ :  
     $E(0, j) = j$   
for  $i = 1, 2, \dots, m$ :  
    for  $j = 1, 2, \dots, n$ :  
         $E(i, j) = \min\{E(i - 1, j) + 1, E(i, j - 1) + 1, E(i - 1, j - 1) + \text{diff}(i, j)\}$   
return  $E(m, n)$ 
```

Etapa 4: Solução I – calcular

for $i = 0, 1, 2, \dots, m$:

$$E(i, 0) = i$$

for $j = 1, 2, \dots, n$:

$$E(0, j) = j$$

for $i = 1, 2, \dots, m$:

for $j = 1, 2, \dots, n$:

$$E(i, j) = \min\{E(i - 1, j) + 1, E(i, j - 1) + 1, E(i - 1, j - 1) + \text{diff}(i, j)\}$$

return $E(m, n)$

Problema da mochila

- Ladrão está com uma mochila que suporta no máximo 10 quilos e quer o maior lucro possível

Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

Problema da mochila

- Ladrão está com uma mochila que suporta no máximo 10 quilos e quer o maior lucro possível

Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

Etapa 1: Equação de Recorrência

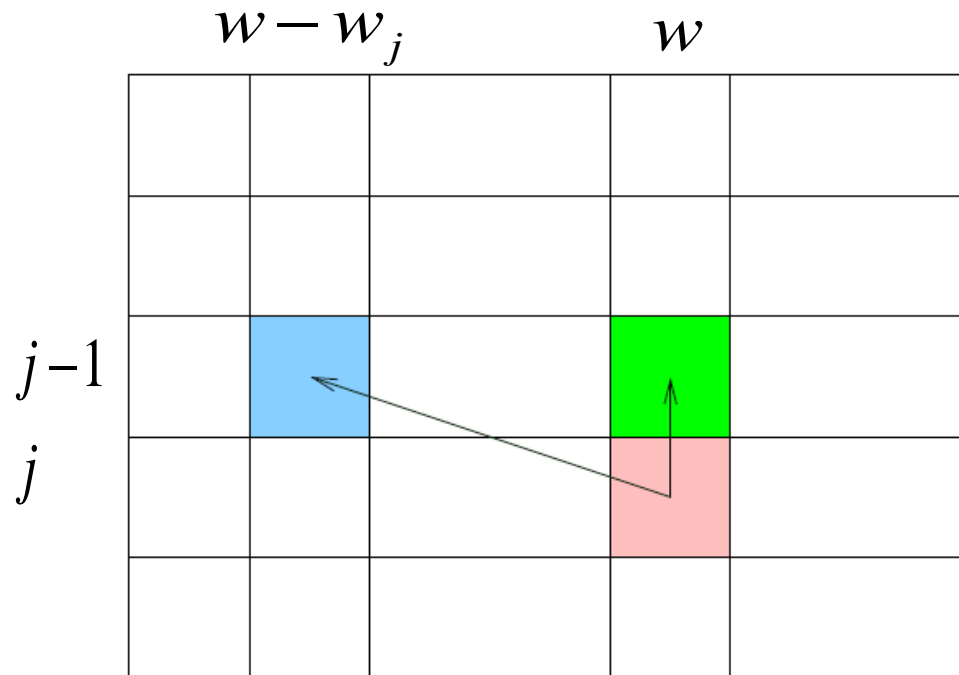
$K(w, j)$

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

Etapa 1: Equação de Recorrência

$K(w, j)$

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$



$$K(w, j) = \max \{ K(w, j-1), K(w - w_j, j-1) + v_j \}$$

Etapa 2: Algoritmo Recursivo

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

Etapa 3: Algoritmo Iterativo

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

Etapa 3: Complexidade

```
Initialize all  $K(0, j) = 0$  and all  $K(w, 0) = 0$ 
for  $j = 1$  to  $n$ :
    for  $w = 1$  to  $W$ :
        if  $w_j > w$ :  $K(w, j) = K(w, j - 1)$ 
        else:  $K(w, j) = \max\{K(w, j - 1), K(w - w_j, j - 1) + v_j\}$ 
return  $K(W, n)$ 
```

- Somente para valores inteiros
- $\Theta(Wn)$

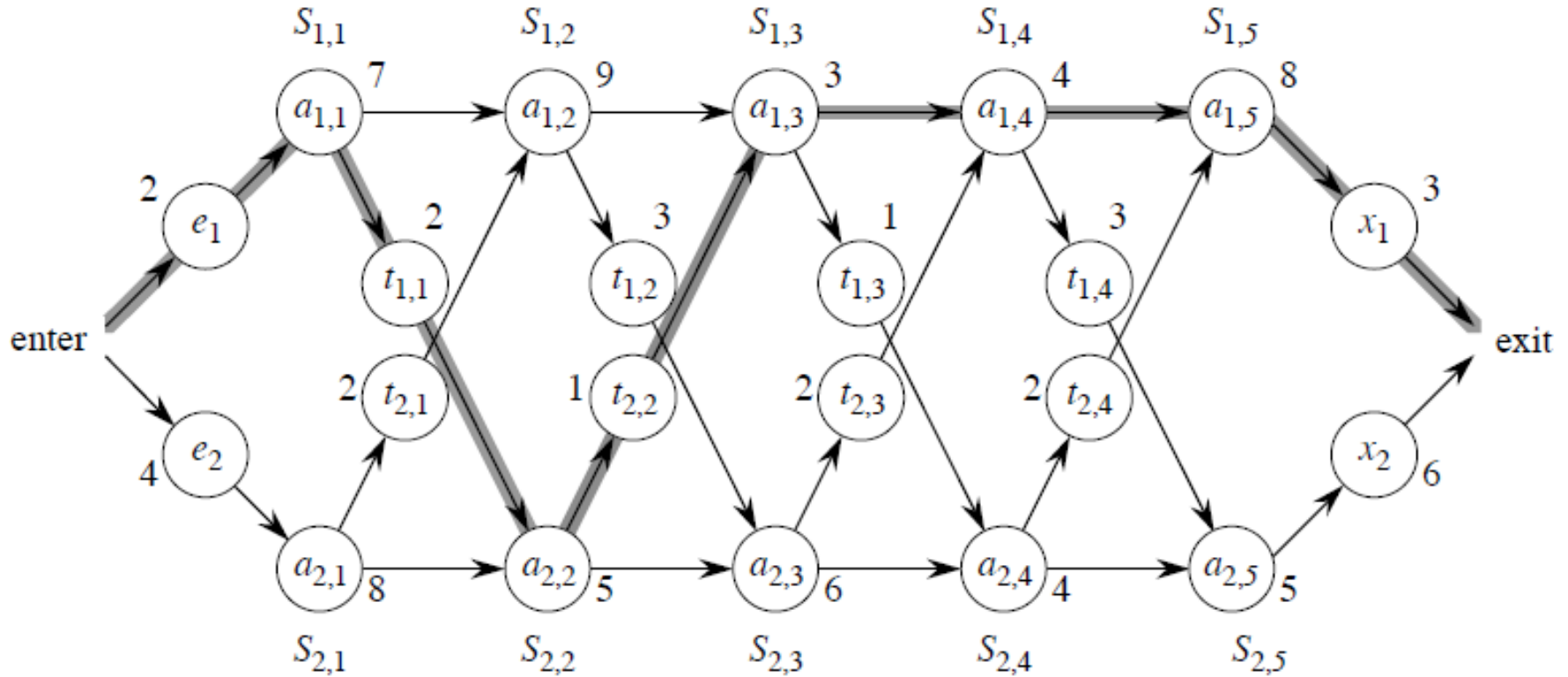
Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0							
2	0							
3	0							
4	0							

Etapa 4: Solução

```
Initialize all  $K(0, j) = 0$  and all  $K(w, 0) = 0$ 
for  $j = 1$  to  $n$ :
  for  $w = 1$  to  $W$ :
    if  $w_j > w$ :  $K(w, j) = K(w, j - 1)$ 
    else:  $K(w, j) = \max\{K(w, j - 1), K(w - w_j, j - 1) + v_j\}$ 
return  $K(W, n)$ 
```


Exemplo – Linha de montagem



Exemplo

- Montadora de veículos com duas linhas de montagem
 - Cada linha tem n estações: $S_{1,1}, \dots, S_{1,n}$ and $S_{2,1}, \dots, S_{2,n}$.
 - Estações correspondentes $S_{1,j}$ e $S_{2,j}$ possuem a mesma função, mas podem ter tempo de execução diferentes, $a_{1,j}$ e $a_{2,j}$.
 - Tempos de entrada: e_1 e e_2 .
 - Tempos de saída: x_1 e x_2 .
 - Para chegar a uma estação:
 - Ficando na mesma linha – nenhum custo
 - Transferido de outra linha – depois da estação $S_{i,j}$ é $t_{i,j}$.

Exemplo

- Problema
 - Que estações das linhas 1 e 2 devem ser escolhidas para ter uma fabricação mais rápida?
 - Tentar todas as possibilidades?
 - Cada candidato pode ser completamente especificado por quais estações da linha 1 são incluídas.
 - Linha 1 tem n estações
 - 2^n subconjuntos

Exemplo

- Estrutura de uma solução ótima
 - Pense em uma maneira rápida da entrada até a estação $S_{1,j}$.
 - Se $j=1$, fácil
 - Se $j \geq 2$, há duas opções
 - De $S_{1,j-1}$
 - De $S_{2,j-1}$

Exemplo

- **Observação chave:** Nós devemos ter um caminho rápido da entrada até $S_{i,j-1}$ nesta solução.
- Se uma maneira rápida de chegar é por meio de $S_{1,j-1}$, nós podemos usá-la para chegar a $S_{1,j}$.
- Se uma maneira rápida de chegar é por meio de $S_{2,j-1}$, nós podemos usá-la para chegar a $S_{1,j}$.

Exemplo

- **Geralmente:** Uma solução ótima para um problema (maneira rápida de chegar a $S_{1,j}$) contém com ele uma solução ótima para os subproblemas (maneiras rápidas de chegar a $S_{1,j-1}$ ou $S_{2,j-1}$).
- Isto é uma **subestrutura ótima**.

Subestrutura ótima

- Use subestruturas ótimas para construir soluções ótimas para problemas, por meio de soluções ótimas de subproblemas.
- Maneira rápida de chegar a $S_{i,j}$:
 - $S_{1,j-1}$, então vá diretamente para $S_{1,j}$, ou
 - $S_{2,j-1}$, transfira da linha 2 para a 1 e então vá para $S_{1,j}$

Solução Recursiva

- Seja $f_i[j] =$ o tempo mais rápido até a estação $S_{i,j}$, $i = 1, 2$ and $j = 1, \dots, n$.

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

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

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

For $j = 2, \dots, n$:

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

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

- f_* dá o valor da solução ótima.
- E se quisermos construir a solução ótima?
 - $l_i[j]$ = número da linha (1 ou 2) cuja estação $j - 1$ é usada para chegar a $S_{i,j}$.
- $S_{l_i[j], j-1}$ precede $S_{i,j}$.
- l^* = número da linha da estação n usada.

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 $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$

$l_1[j] \leftarrow 1$

else $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$

$l_1[j] \leftarrow 2$

if $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$

then $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$

$l_2[j] \leftarrow 2$

else $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$

$l_2[j] \leftarrow 1$

if $f_1[n] + x_1 \leq f_2[n] + x_2$

then $f^* = f_1[n] + x_1$

$l^* = 1$

else $f^* = f_2[n] + x_2$

$l^* = 2$

Construindo a solução ótima

PRINT-STATIONS(l, n)

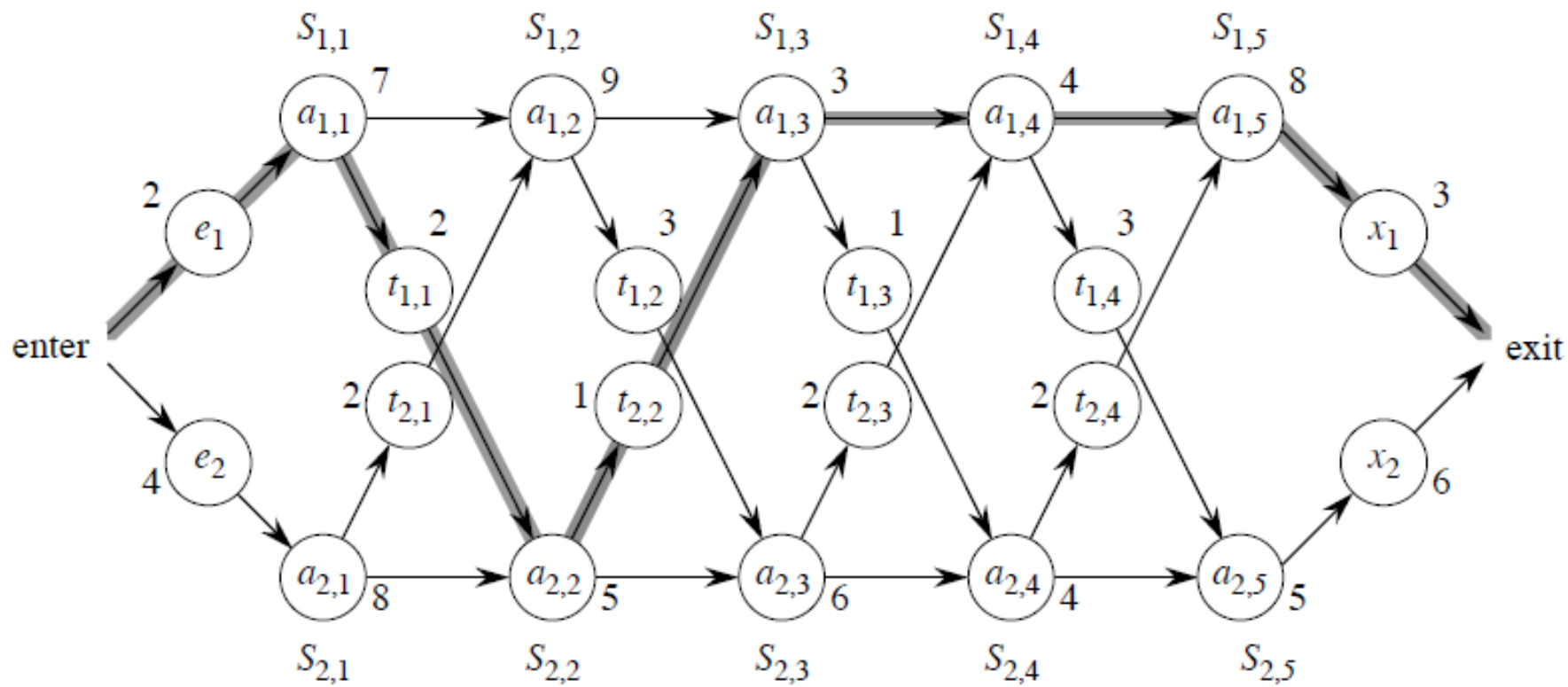
$i \leftarrow l^*$

print “linha” i “, estação” n

for $j \leftarrow n$ **downto** 2

do $i \leftarrow l_j[j]$

print “linha” i “, estação” $j - 1$



Considerações Finais

- Diferença entre PD e D&C
 - Sobreposição de problemas
- Definição da Equação de Recorrência
 - Grafo induzido
 - Automatização dos passos
- Memorização