Programação Dinâmica

Prof. Anderson Almeida Ferreira

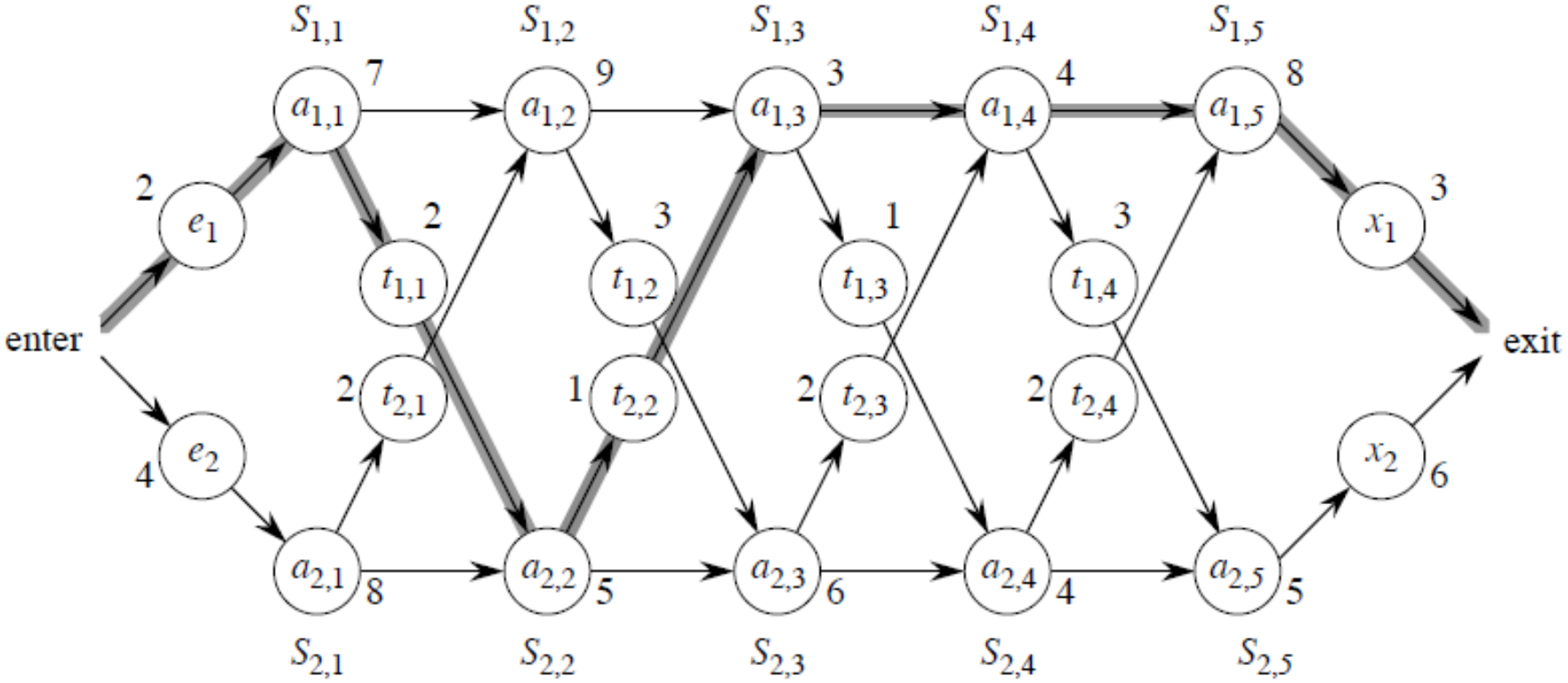
Programação Dinâmica

- 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.

Exemplo – Linha de montagem



Exemplo

- Montadora de veículos com duas linhas de montagem
  - Cada linha tem n estações: $S_{1,1}, \ldots, S_{1,n}$ *and* $S_{2,1}, \ldots, S_{2,n}$.
  - 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>= 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* $j = 1, \ldots, n$.
- ***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 $j = 2, \ldots, n$:

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

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

- 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* $S_{i,\,j}$.
- $S_{li}\,[\,j\,]$, *j−1 precede* $S_{i,\,j}$.
- l_* = número da linha da estação n usada.

FASTEST-WAY(a, t, e, x, n)

$f1[1] \leftarrow e1 + a1,1$

$f2[1] \leftarrow e2 + a2,1$

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)*

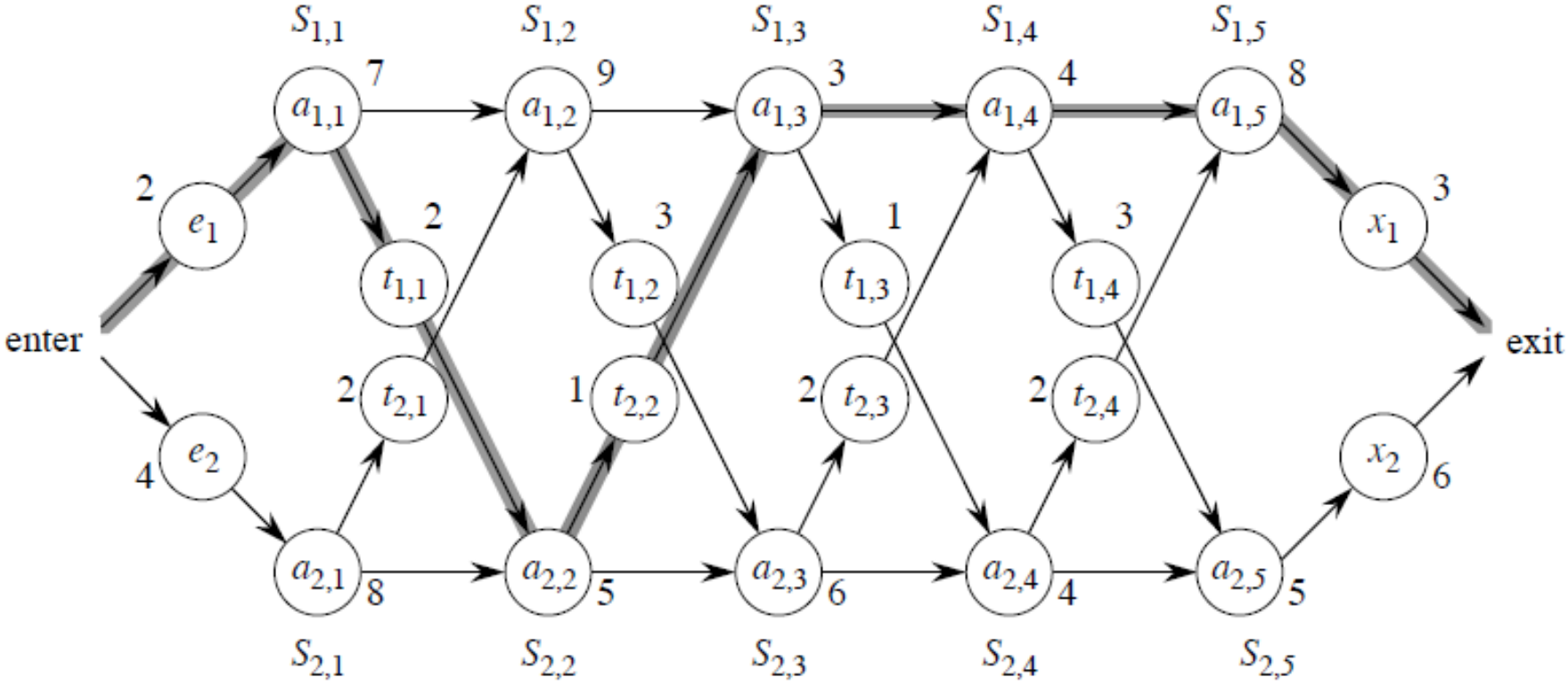
i ←*l*∗

print “linha” *i “, estação” n*

for *j* ← *n downto 2*

do *i* ←l_i *[j]*

print “linha” *i “, estação” j – 1*


$S_{1,1}$
$S_{1,2}$
$S_{1,3}$
$S_{1,4}$
$S_{1,5}$
$a_{1,1}$
$a_{1,2}$
$a_{1,3}$
$a_{1,4}$
$a_{1,5}$
7
9
3
4
8
2
e_1
2
3
1
3
3
x_1
$t_{1,1}$
$t_{1,2}$
$t_{1,3}$
$t_{1,4}$
enter
exit
2
$t_{2,1}$
1
$t_{2,2}$
2
$t_{2,3}$
2
$t_{2,4}$
4
e_2
x_2
6
$a_{2,1}$
$a_{2,2}$
$a_{2,3}$
$a_{2,4}$
$a_{2,5}$
8
5
6
4
5
$S_{2,1}$
$S_{2,2}$
$S_{2,3}$
$S_{2,4}$
$S_{2,5}$

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)
```

```
MemFibo(n):
  if (n < 2)
      return n
  else
      if F[n] is undefined
          F[n] ← MemFibo(n − 1) + MemFibo(n − 2)
      return F[n]
```

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

Fibonacci: definindo recursão

```
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] ← MemFibo(n − 1) + MemFibo(n − 2)
      return F[n]
```

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

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] ← MemFibo(n − 1) + MemFibo(n − 2)
      return F[n]

IterFibo(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
      F[i] ← F[i − 1] + F[i − 2]
  return F[n]
```

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

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] ← MemFibo(n − 1) + MemFibo(n − 2)
      return F[n]

IterFibo(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
      F[i] ← F[i − 1] + F[i − 2]
  return F[n]
```

- Grafo de recorrência
  - Subproblemas – nós
  - Dependência – arestas
- Memoização
- **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] ← MemFibo(n − 1) + MemFibo(n − 2)
      return F[n]
```

```
IterFibo(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
      F[i] ← F[i − 1] + F[i − 2]
  return F[n]
```

```
IterFibo2(n):
  prev ← 1
  curr ← 0
  for i ← 1 to n
      next ← curr + prev
      prev ← curr
      curr ← next
  return curr
```

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

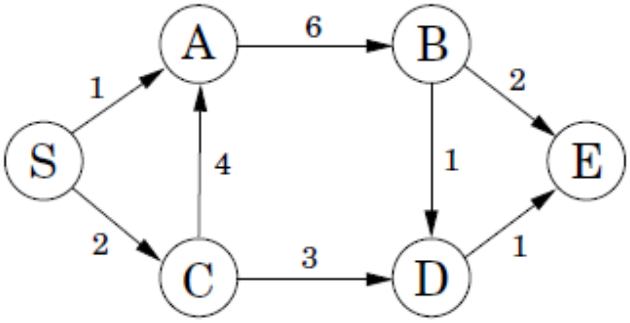
Tabela

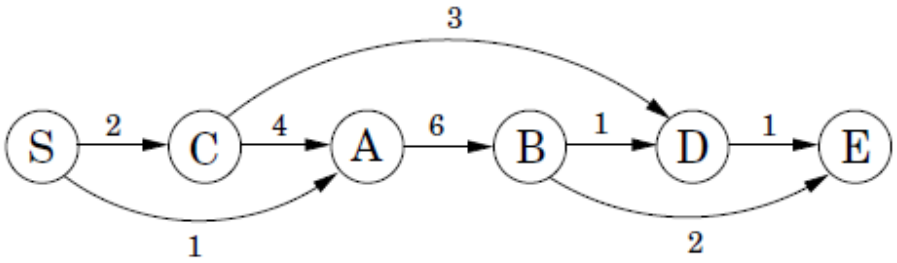
- Ordenação parcial
- Economizando memória

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.

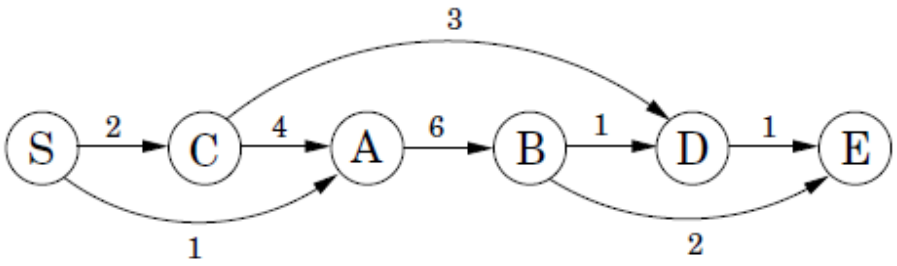
Linearização de Grafos Direcionados Acíclicos (DAGs)





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 dist$(\cdot)$ values to ∞
dist$(s) = 0$
for each $v \in V \backslash \{s\}$, in linearized order:
    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.

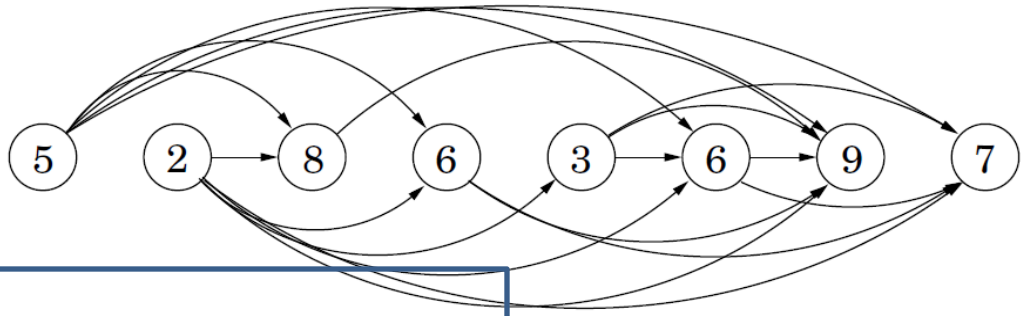
5 2 8 6 3 6 9 7

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

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

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

Etapa 4: Construindo solução

for $j = 1, 2, \ldots, n$:
    $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$
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 $x[1 \cdots m]$ $y[1 \cdots n]$ $E(m, n)$

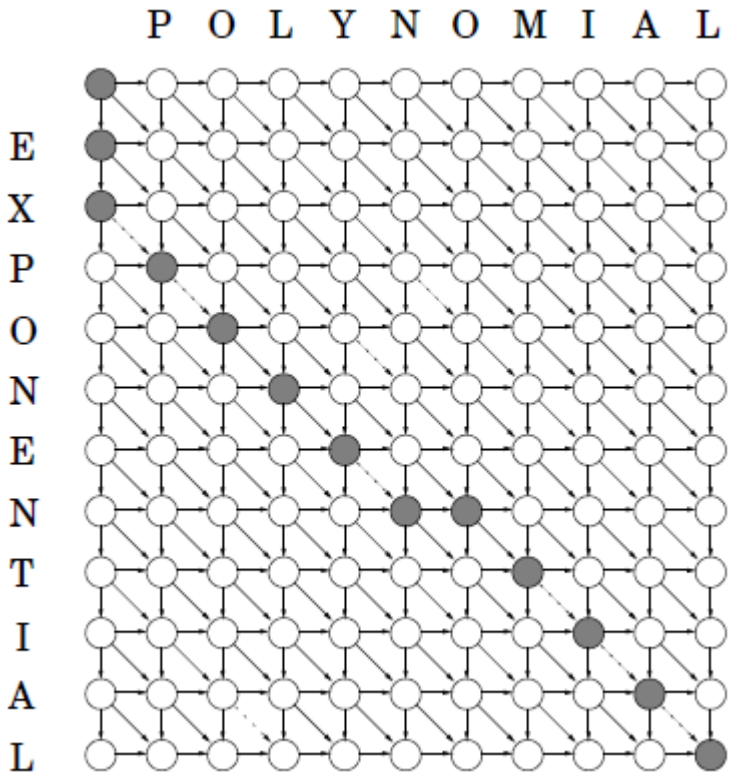
- Subproblema: alinhamento de prefixos

$E(i, j)$ $\begin{matrix} x[i] \\ - \end{matrix}$ **or** $\begin{matrix} - \\ y[j] \end{matrix}$ **or** $\begin{matrix} x[i] \\ y[j] \end{matrix}$

- Composição: inserir, remover, casar

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

Distância de Edição - DAG



Etapa 2: Algoritmo Recursivo

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

Etapa 2: Algoritmo Recursivo

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

Etapa 3: Algoritmo Iterativo

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

Etapa 3: Complexidade

```
for i = 0, 1, 2, ..., m:
    E(i, 0) = i
for j = 1, 2, ..., n:
    E(0, j) = j
for i = 1, 2, ..., m:
    for j = 1, 2, ..., n:
        E(i, j) = min{E(i − 1, j) + 1, E(i, j − 1) + 1, E(i − 1, j − 1) + diff(i, j)}
return E(m, n)
```

- $\Theta(mn)$ de tempo e espaço

Etapa 3: Como economizar espaço?

```
for i = 0, 1, 2, ..., m:
    E(i, 0) = i
for j = 1, 2, ..., n:
    E(0, j) = j
for i = 1, 2, ..., m:
    for j = 1, 2, ..., n:
        E(i, j) = min{E(i − 1, j) + 1, E(i, j − 1) + 1, E(i − 1, j − 1) + diff(i, j)}
return E(m, n)
```

- Θ(*mn*) de tempo e Θ(*m*) espaço

Distância de Edição - Exemplo

		P	O	L	I	N	O	M	I	A	L
E											
X											
P											
O											
N											
E											
N											
C											
I											
A											
L											

Etapa 4: Solução I – armazenar

```
for i = 0, 1, 2, ..., m:
    E(i, 0) = i
for j = 1, 2, ..., n:
    E(0, j) = j
for i = 1, 2, ..., m:
    for j = 1, 2, ..., n:
        E(i, j) = min{E(i − 1, j) + 1, E(i, j − 1) + 1, E(i − 1, j − 1) + diff(i, j)}
return E(m, n)
```

Etapa 4: Solução I – calcular

```
for i = 0, 1, 2, ..., m:
    E(i, 0) = i
for j = 1, 2, ..., n:
    E(0, j) = j
for i = 1, 2, ..., m:
    for j = 1, 2, ..., n:
        E(i, j) = min{E(i − 1, j) + 1, E(i, j − 1) + 1, E(i − 1, j − 1) + 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

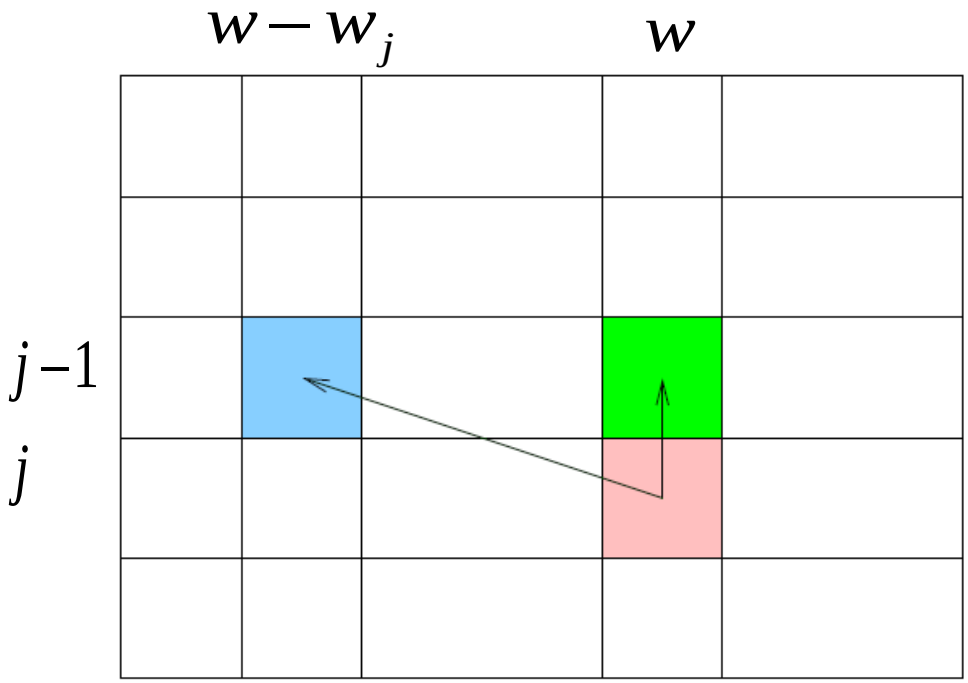
$K(w, j)$

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

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 $K(W,n)$

- Pseudo-polinomial (NP-completo)
- 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 $K(W,n)$

Considerações Finais

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