Programação Dinâmica

José Elias Claudio Arroyo

Departamento de Informática
Universidade Federal de Viçosa

INF 332 - 2022/2

Outline

Programação Dinâmica (PD)

- Inventado pelo matemático americano Richard Bellman anos 1950.
- A palavra “programação” se refere a fazer um **planejamento**, e não a programação de computadores.
- Após se tornar uma ferramenta importante para matemática aplicada é que a PD passou a ser considerada uma **técnica de construção de algoritmos**.

Programação dinâmica é uma técnica para solucionar problemas compostos por subproblemas menores (onde existe **sobreposição de subproblemas**).

Programação dinâmica é uma técnica para solucionar problemas compostos por subproblemas menores (onde existe **sobreposição de subproblemas**).

- Normalmente as soluções dos subproblemas menores e do problema principal estão relacionadas através de uma **relaçã de recorrência**.

Programação Dinâmica

Programação dinâmica é uma técnica para solucionar problemas compostos por subproblemas menores (onde existe **sobreposição de subproblemas**).

- Normalmente as soluções dos subproblemas menores e do problema principal estão relacionadas através de uma **relaçã de recorrência**.
- Ao invés de solucionar subproblemas idênticos várias vezes, com a PD soluciona-se cada subproblema **apenas uma vez** e as soluções são armazenadas em memória (**tabelas**), para serem reutilizadas no futuro.

Programação Dinâmica

Programação dinâmica é uma técnica para solucionar problemas compostos por subproblemas menores (onde existe **sobreposição de subproblemas**).

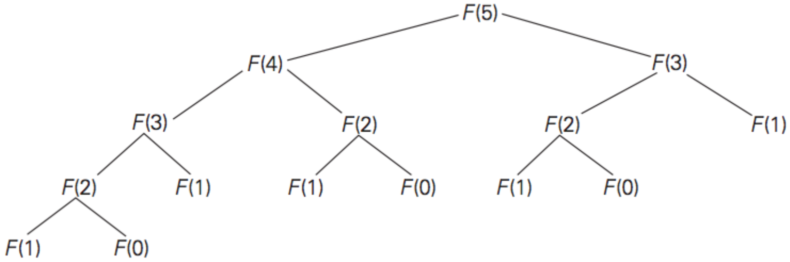
- Normalmente as soluções dos subproblemas menores e do problema principal estão relacionadas através de uma **relaçã de recorrência**.
- Ao invés de solucionar subproblemas idênticos várias vezes, com a PD soluciona-se cada subproblema **apenas uma vez** e as soluções são armazenadas em memória (**tabelas**), para serem reutilizadas no futuro.
- A solução do problema original, geralmente, é obtida de forma ascendente (buttom up). Ou seja, as soluções dos subproblemas, que são armazenadas numa tabela, são utilizadas para obter soluções de problemas cada vez maiores, até gerar a solução do problema original.

Exemplo: Série de Fibonacci

- Série de Fibonacci é a seguinte:
  $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$
- Os termos de desta série são gerados por uma simples **recorrência**:
  $F(n) = F(n-1) + F(n-2)$
  $F(1) = 1$
  $F(0) = 0$
- Note que, para calcular $F(n)$ precisa-se calcular $F(n-1)$ e $F(n-2)$.

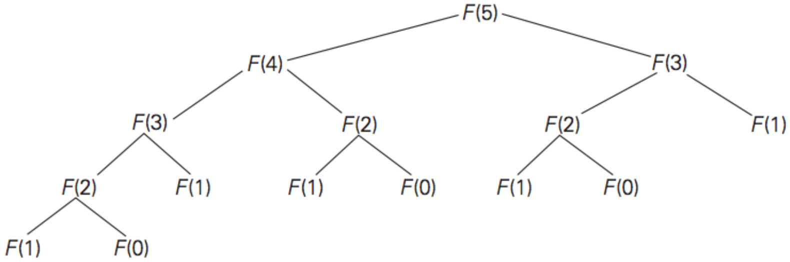
Exemplo: Série de Fibonacci

Árvore para calcular o n-ésimo termo das série de Fiboncci (por exemplo para calcular F(5)):



Exemplo: Série de Fibonacci

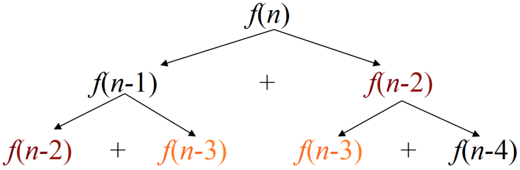
Árvore para calcular o n-ésimo termo das série de Fiboncci (por exemplo para calcular F(5)):



- Repare que o problema de calcular $F(n)$ é representado através de subproblemas sobrepostos.
- Note que F(2) é utilizado para calcular F(4) e F(3).

Exemplo: Série de Fibonacci

- Se utilizarmos essa recorrência diretamente (forma top-down), teríamos que recalcular um mesmo valor mais de uma vez.



...

Obtendo o *n*-ésimo número de Fibonacci de forma bottom-up:

- $f(0) = 0$
- $f(1) = 1$
- $f(2) = 0+1 = 1$
- $f(3) = 1+1 = 2$
- $f(4) = 1+2 = 3$
- $f(5) = 2+3 = 5$

. . . . .

- $f(n\text{-}2) =$
- $f(n\text{-}1) =$
- $f(n) = f(n\text{-}1) + f(n\text{-}2)$

Tabela:

0	1	1	2	3	5	8	...
0	1	2	3	4	5	6	

Ideia principal da técnica PD:

- Definir uma **relação de recorrência** entre um problema maior e alguns subproblemas menores;

Ideia principal da técnica PD:

- Definir uma **relação de recorrência** entre um problema maior e alguns subproblemas menores;
- Solucionar cada subproblema apenas uma vez;

Ideia principal da técnica PD:

- Definir uma **relação de recorrência** entre um problema maior e alguns subproblemas menores;
- Solucionar cada subproblema apenas uma vez;
- Armazenar as soluções em uma tabela;

Ideia principal da técnica PD:

- Definir uma **relação de recorrência** entre um problema maior e alguns subproblemas menores;
- Solucionar cada subproblema apenas uma vez;
- Armazenar as soluções em uma tabela;
- Obter a solução para o problema original da tabela (de forma **ascendente**).

- A PD, geralmente, é usado para resolver **problemas de otimização**, que consistem em procurar uma solução que minimize ou maximize um valor numérico (**solução ótima**).
- Na resolução de problemas de otimização é usado o seguinte **princípio de otimalidade**:
  "A solução ótima de um problema (ou subproblema) é obtida a partir de soluções ótimas de subproblemas menores"

Exemplo: Problema da fileira de moedas

- Dada um **fileira de n moedas** com valores inteiros positivos: $c_1, c_2, \cdots, c_n$.
- O objetivo é coletar o **maior valor** em moedas sendo que, **moedas adjacentes não podem ser coletadas.**
- **Exemplo**: Fileira 5, 1, 2, 10, 6, 2
  Alguma soluções:
  Coletar 5, 2 e 6. Valor coletado: 13.
  Coletar 5, 10 e 2. Valor coletado: 17 (melhor solução).
  Coletar 1, 10 e 2. Valor coletado: 13.

Exemplo: Problema da fileira de moedas

Relação de recorrência para calcular os valores das soluções (i.e., os valores coletados):

$$F(n) = \max\{c_n + F(n-2),\ F(n-1)\} \quad \text{for } n > 1,$$
$$F(0) = 0, \qquad F(1) = c_1.$$

"Para obter a solução ótima de um problema com n moedas, determinar as soluções ótimas dos subproblemas com $n-1$ e $n-2$ moedas"

A fórmula acima deve ser aplicada de forma bottom-up, ou seja, calcular $F(2), F(3), ..., F(n)$. Os valores calculados são guardados em uma tabela.

Exemplo: Problema da fileira de moedas

Exemplo: Fileira 5, 1, 2, 10, 6, 2. Tabela construida passo a passo:

$F[0] = 0$, $F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

$F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

$F[4] = \max\{10 + 5, 7\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

Exemplo: Problema da fileira de moedas

$F[5] = \max\{6 + 7, 15\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

$F[6] = \max\{2 + 15, 15\} = 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

Exemplo: Problema da fileira de moedas

ALGORITHM *CoinRow*$(C[1..n])$

```
//Applies formula (8.3) bottom up to find the maximum amount of money
//that can be picked up from a coin row without picking two adjacent coins
//Input: Array C[1..n] of positive integers indicating the coin values
//Output: The maximum amount of money that can be picked up
F[0] ← 0;   F[1] ← C[1]
for i ← 2 to n do
    F[i] ← max(C[i] + F[i − 2], F[i − 1])
return F[n]
```

Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

Exemplo: Problema da fileira de moedas

ALGORITHM *CoinRow*($C[1..n]$)

//Applies formula (8.3) bottom up to find the maximum amount of money
//that can be picked up from a coin row without picking two adjacent coins
//Input: Array $C[1..n]$ of positive integers indicating the coin values
//Output: The maximum amount of money that can be picked up

$F[0] \leftarrow 0; \quad F[1] \leftarrow C[1]$
for $i \leftarrow 2$ **to** n **do**
    $F[i] \leftarrow \max(C[i] + F[i-2], F[i-1])$
return $F[n]$

Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

Complexidade: $O(n)$

Exemplo: Problema da fileira de moedas

Exercício

Altere o algoritmo *CoinRow* para imprimir o valor $F(n)$ (o maior valor coletado) e as moedas coletadas.

Problema do troco:

- Dê o troco com valor n utilizando um número mínimo de moedas com valores $d_1 < d_2 < \cdots < d_m$.

Exemplo: Problema do troco

Problema do troco:

- Dê o troco com valor n utilizando um número mínimo de moedas com valores $d_1 < d_2 < \cdots < d_m$.
- Embora exista um algoritmo muito simples e eficiente para as denominações de moedas usadas na maioria dos países, aqui consideraremos o uso de PD para o caso geral.
- Suponha que existem quantidades ilimitadas para cada uma das m moedas $d_1 < d_2 < \cdots < d_m$, onde $d_1 = 1$.
- **Exemplo**: Com as moedas $d_1 = 1$, $d_2 = 3$, $d_3 = 4$, dar um troco de valor $n = 6$.

Exemplo: Problema do troco

- Seja $F(n)$ a quantidade mínima de moedas para gerar um troco de valor n ($\forall n \geq 1$).
- Se $n = 0$, $F(0) = 0$.

Exemplo: Problema do troco

- Seja $F(n)$ a quantidade mínima de moedas para gerar um troco de valor n ($\forall n \geq 1$).
- Se $n = 0$, $F(0) = 0$.
- O valor n só poderá ser obtida adicionando uma moeda d_j ($d_j \leq n$) ao valor $n - d_j$, para todo $j = 1, 2, ..., m$.

Exemplo: Problema do troco

- Seja $F(n)$ a quantidade mínima de moedas para gerar um troco de valor n ($\forall n \geq 1$).
- Se $n = 0$, $F(0) = 0$.
- O valor n só poderá ser obtida adicionando uma moeda d_j ($d_j \leq n$) ao valor $n - d_j$, para todo $j = 1, 2, ..., m$.
- Portanto, podemos considerar todas as moedas d_j, tal que $d_j \leq n$, e selecionar a moeda que minimiza $F(n - d_j)$.
- Ou, seja basta encontrar o valor mínimo de $F(n - d_j)$ e logo somar 1 (pois adiciona-se a moeda d_j que minimiza $F(n - d_j)$.).

Exemplo: Problema do troco

- Seja $F(n)$ a quantidade mínima de moedas para gerar um troco de valor n ($\forall n \geq 1$).
- Se $n = 0$, $F(0) = 0$.
- O valor n só poderá ser obtida adicionando uma moeda d_j ($d_j \leq n$) ao valor $n - d_j$, para todo $j = 1, 2, ..., m$.
- Portanto, podemos considerar todas as moedas d_j, tal que $d_j \leq n$, e selecionar a moeda que minimiza $F(n - d_j)$.
- Ou, seja basta encontrar o valor mínimo de $F(n - d_j)$ e logo somar 1 (pois adiciona-se a moeda d_j que minimiza $F(n - d_j)$.).

Relação de recorrência:

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

Exemplo: Problema do troco

Exemplo: moedas $d_1 = 1$, $d_2 = 3$, $d_3 = 4$, valor do troco $n = 6$. Tabela construida passo a passo:

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1-1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2-1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3-1], F[3-3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4-1], F[4-3], F[4-4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5-1], F[5-3], F[5-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

$F[6] = \min\{F[6-1], F[6-3], F[6-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

Exemplo: Problema do troco

```
ALGORITHM  ChangeMaking(D[1..m], n)
    //Applies dynamic programming to find the minimum number of coins
    //of denominations d1 < d2 < ··· < dm where d1 = 1 that add up to a
    //given amount n
    //Input: Positive integer n and array D[1..m] of increasing positive
    //       integers indicating the coin denominations where D[1] = 1
    //Output: The minimum number of coins that add up to n
    F[0] ← 0
    for i ← 1 to n do
        temp ← ∞;  j ← 1
        while j ≤ m and i ≥ D[j] do
            temp ← min(F[i − D[j]], temp)
            j ← j + 1
        F[i] ← temp + 1
    return F[n]
```

Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

Exemplo: Problema do troco

ALGORITHM *ChangeMaking*($D[1..m]$, n)

//Applies dynamic programming to find the minimum number of coins
//of denominations $d_1 < d_2 < \cdots < d_m$ where $d_1 = 1$ that add up to a
//given amount n
//Input: Positive integer n and array $D[1..m]$ of increasing positive
// integers indicating the coin denominations where $D[1] = 1$
//Output: The minimum number of coins that add up to n

$F[0] \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  $temp \leftarrow \infty$; $j \leftarrow 1$
  while $j \le m$ **and** $i \ge D[j]$ **do**
    $temp \leftarrow \min(F[i - D[j]], temp)$
    $j \leftarrow j + 1$
  $F[i] \leftarrow temp + 1$
return $F[n]$

Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

Complexidade: $O(nm)$

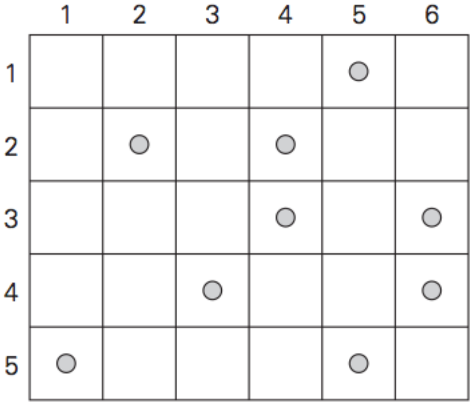
Exemplo: Problema do troco

Exercício

Altere o algoritmo *ChangeMaking* de tal maneira que sejam guardadas as moedas utilizadas. O algoritmo deve imprimir o valor $F(n)$ (a quantidade mínima de moedas utilizadas) e as moedas utilizadas para dar o troco.

Exemplo: Problema da Coleta de Moedas

- Várias moedas são colocadas em diferentes posições de um grid $n \times m$ com não mais de uma moeda por célula.



Exemplo: Problema da Coleta de Moedas

- Um robô, localizado na parte superior esquerda do grid (célula $(1, 1)$), precisa coletar o maior número possível de moedas e traze-las para a posição inferior direita do grid (célula (n, m)).
- O robô pode mover-se para uma posição à **direita** ou uma posição **abaixo** da atual.
- Quando o robô visita uma posição que tem uma moeda, ele automaticamente recolhe a moeda.
- Deseja-se encontrar o **número máximo de moedas** que o robô pode coletar e o **caminho** que ele precisa seguir para fazer isso.

Exemplo: Problema da Coleta de Moedas

- Seja $F(i, j)$ o maior número de moedas que o robô pode coletar e trazer para a célula (i, j).

Exemplo: Problema da Coleta de Moedas

- Seja $F(i, j)$ o maior número de moedas que o robô pode coletar e trazer para a célula (i, j).
- A célula (i, j) pode ser atingido a partir das **células adjacentes** $(i-1, j)$ ou $(i, j-1)$ (célula acima ou célula à esquerda).
- Os maiores números de moedas que podem ser trazidas para essas células são $F(i-1, j)$ e $F(i, j-1)$, respectivamente.

Exemplo: Problema da Coleta de Moedas

- Seja $F(i,j)$ o maior número de moedas que o robô pode coletar e trazer para a célula (i,j).
- A célula (i,j) pode ser atingido a partir das **células adjacentes** $(i-1,j)$ ou $(i,j-1)$ (célula acima ou célula à esquerda).
- Os maiores números de moedas que podem ser trazidas para essas células são $F(i-1,j)$ e $F(i,j-1)$, respectivamente.
- Portanto, o maior número de moedas que o robô pode trazer até a célula (i,j) é o máximo de $F(i-1,j)$ e $F(i,j-1)$ **mais uma moeda** possível na própria célula (i,j). Ou seja, temos a seguinte fórmula para $F(i,j)$:

$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \quad \text{for } 1 \le i \le n, \ \ 1 \le j \le m$$

$$F(0, j) = 0 \ \text{ for } 1 \le j \le m \quad \text{and} \quad F(i, 0) = 0 \ \text{ for } 1 \le i \le n,$$

Onde $c_{ij} = 1$ se existe uma moeda na célula (i,j), $c_{ij} = 0$ caso contrário.

Exemplo: Problema da Coleta de Moedas

ALGORITHM *RobotCoinCollection*$(C[1..n, 1..m])$

//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an $n \times m$ board by starting at (1, 1)
//and moving right and down from upper left to down right corner
//Input: Matrix $C[1..n, 1..m]$ whose elements are equal to 1 and 0
//for cells with and without a coin, respectively
//Output: Largest number of coins the robot can bring to cell (n, m)
$F[1, 1] \leftarrow C[1, 1]$; **for** $j \leftarrow 2$ **to** m **do** $F[1, j] \leftarrow F[1, j-1] + C[1, j]$
for $i \leftarrow 2$ **to** n **do**
    $F[i, 1] \leftarrow F[i-1, 1] + C[i, 1]$
    for $j \leftarrow 2$ **to** m **do**
        $F[i, j] \leftarrow \max(F[i-1, j], F[i, j-1]) + C[i, j]$
return $F[n, m]$

Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

Exemplo: Problema da Coleta de Moedas

ALGORITHM *RobotCoinCollection*$(C[1..n, 1..m])$

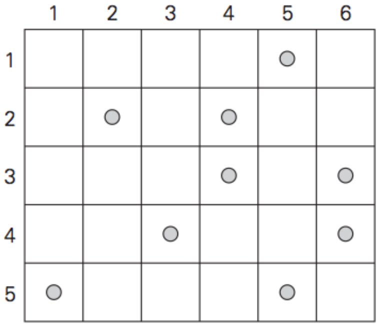
//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an $n \times m$ board by starting at (1, 1)
//and moving right and down from upper left to down right corner
//Input: Matrix $C[1..n, 1..m]$ whose elements are equal to 1 and 0
//for cells with and without a coin, respectively
//Output: Largest number of coins the robot can bring to cell (n, m)

$F[1, 1] \leftarrow C[1, 1]$; **for** $j \leftarrow 2$ **to** m **do** $F[1, j] \leftarrow F[1, j-1] + C[1, j]$
for $i \leftarrow 2$ **to** n **do**
  $F[i, 1] \leftarrow F[i-1, 1] + C[i, 1]$
  for $j \leftarrow 2$ **to** m **do**
    $F[i, j] \leftarrow \max(F[i-1, j], F[i, j-1]) + C[i, j]$
return $F[n, m]$

Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

Complexidade: $O(nm)$.

Exemplo: Problema da Coleta de Moedas



(a)

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

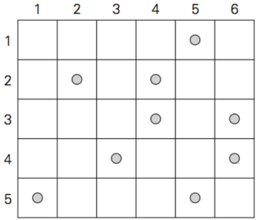
(b)

Exemplo: Problema da Coleta de Moedas

Exercício

Escreva um algoritmo para determinar um caminho ótimo que o robô precisa seguir, da posição $(1, 1)$ até a posição (n, m) do grid, para coletar o maior número possível de moedas.
Determine a complexidade do algoritmo.

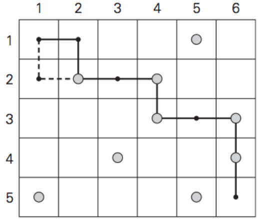
Exemplo: Problema da Coleta de Moedas



(a)

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

(b)



(c)

Neste exemplo são produzidos dois caminhos ótimos.

Exemplo: Subsequência Consecutiva Máxima

Dada uma sequência $X = [x_1, x_2, ..., x_n]$ de n números reais.
Encontrar uma subsequência consecutiva $Y = [x_i, x_{i+1}, ..., x_j]$ de X, $1 \leq i \leq j \leq n$, tal que a soma dos elementos de Y seja máxima.

Exemplos:

- $X = [4, 2, -7, 3, 0, -2, 1, 5, -2] \Rightarrow$ soma = 7.
- $X = [-2, 11, -4, 13, -5, 2] \Rightarrow$ soma = 20.
- $X = [-1, -2, 0] \Rightarrow$ soma = 0.
- $X = [4, 2, 8, 1] \Rightarrow$ soma = 15.

Exemplo: Subsequência Consecutiva Máxima

- Para projetar um algoritmo de PD, vamos definir uma função de recorrência para calcular o valor da soma de uma **subsequência consecutiva** $Y = [x_i, x_{i+1}, ..., x_j]$, $(1 \leq i \leq j \leq n)$.
- Seja $F(j)$ a soma da subsequência que finaliza em x_j:
  $Y = [x_i, x_{i+1}, ..., x_j]$
- $F(0) = 0$ (soma da subsequência vazia).
- Função de **recorrência**:

$$F(j) = max\{F(j-1) + x_j, x_j\}, j = 1, ..., n$$
$$F(0) = 0$$

Exemplo: Subsequência Consecutiva Máxima

$F(j) = max\{F(j-1) + x_j, x_j\}, j = 1, ..., n$
$F(0) = 0$

- Note que, com esta recorrência são calculadas a soma de $n+1$ subsequências: subsequência vazia, subsequências que finalizam em x_1, x_2, ... e x_n
- Se $F(j-1) < 0$, então o $F(j) = x_j$, isto significa que cria-se uma nova subsequñcia com um único elemento: $[x_j]$.
- Para que $[x_i, x_{i+1}, ..., x_j]$ seja a solução ótima, $F(j)$ deve ser máximo e $F(i-1) \leq 0$ (a soma da subsequência que finaliza em x_{i-1} deve ser ≤ 0).
- Exemplo de subsequências que podem ser determinadas:

$[], \quad [x_1], [x_1, x_2], [x_1, x_2, x_3], [x_4], [x_4, x_5], [x_4, x_5, x_6], ...$

$F(0), F(1), \quad F(2), \quad F(3), \quad F(4), \quad F(5), \quad F(6), ... \quad$ (suponha $F(3) < 0$)

Exemplo: Subsequência Consecutiva Máxima

- Os valores das $n + 1$ somas ($F(j)$, $j = 1, ..., n$) são armazenadas em uma tabela (array).
- O valor da Subsequência Consecutiva Máxima será o maior valor armazenado no array:
  $SomaMax = max\{F(j) : j = 1, ..., n\}$
- Para determinar os elementos da SCM, a partir da maior soma armazenada no array, procurar, de "forma regressiva", a posição da primeira soma ≤ 0.

Exemplo: $n = 9$, $X = [4, 2, -7, 3, 0, -2, 1, 5, -2]$

$F(0) = 0$

$F(1) = \max\{0 + 4, 4\} = 4$

$F(2) = \max\{4 + 2, 2\} = 6$

$F(3) = \max\{6 - 7, -7\} = -1$

$F(4) = \max\{-1 + 3, 3\} = 3$

$F(5) = \max\{3 + 0, 0\} = 3$

$F(6) = \max\{3 - 2, -2\} = 1$

$F(7) = \max\{1 + 1, 1\} = 2$

$F(8) = \max\{2 + 5, 5\} = 7$

$F(9) = \max\{7 - 2, -2\} = 5$

$F(0) = 0$
$F(j) = \max\ \{\ F(j\text{-}1) + x_j\ ,\ \ x_j\ \}$

F

0	1	2	3	4	5	6	7	8	9
0	4	6	-1	3	3	1	2	7	5

$Y = [x_4, x_5, x_6, x_7, x_8]$

$Y = [3,\ \ 0,\ \ -2,\ \ 1,\ \ 5]$

Exemplo: $n = 9$, $X = [4, 2, -7, 3, 0, -2, 1, 5, -2]$

$F(0) = 0$

$F(j) = \max \{ F(j\text{-}1) + x_j , \ x_j \}$

$F(0) = 0$
$F(1) = \max\{0 + 4, 4\} = 4$
$F(2) = \max\{4 + 2, 2\} = 6$
$F(3) = \max\{6 - 7, -7\} = -1$
$F(4) = \max\{-1 + 3, 3\} = 3$
$F(5) = \max\{3 + 0, 0\} = 3$
$F(6) = \max\{3 - 2, -2\} = 1$
$F(7) = \max\{1 + 1, 1\} = 2$
$F(8) = \max\{2 + 5, 5\} = 7$
$F(9) = \max\{7 - 2, -2\} = 5$

F

0	1	2	3	4	5	6	7	8	9
0	4	6	-1	3	3	1	2	7	5

$Y = [x_4, x_5, x_6, x_7, x_8]$

$Y = [3, \ 0, \ -2, \ 1, \ 5]$

Complexidade do algoritmo PD: $O(n)$

Exemplo: Subsequência Consecutiva Máxima

Exercício

Escreva o algoritmo de PD para determinar a subsequência consecutiva máxima de uma sequência X com *n* elementos.

Exemplo: Problema da Mochila

Dados n itens e uma mochila de capacidade W:

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$.

Notação: $Mochila(n, W)$

Solução por PD:

- Seja *Mochila*(i, j) um subproblema com i itens e capacidade da mochila j, $(1 \leq i \leq n, 1 \leq j \leq W)$.
- Seja $F(i, j)$ o valor da solução ótima do suproblema *Mochila*(i, j).
- Para o problema *Mochila*(i, j) com i itens $\{1, 2, ..., i-1, i\}$ e mochila de capacidade j,
  - Se a solução ótima **não inclui** o item i, então $F(i, j) = F(i-1, j)$
  - Se a solução ótima **inclui** o item i, então , $F(i, j) = v_i + F(i-1, j-w_i)$.

Exemplo: Problema da Mochila

Ou seja,

- Se $w_i > j$ (ou $j - w_i < 0$), o item i não cabe na mochila de capacidade j,
  $\Rightarrow F(i,j) = F(i-1,j)$.
- Se $w_i \leq j$ (ou $j - w_i \geq 0$), o item i cabe na mochila de capacidade j,
  $\Rightarrow F(i,j) = max\{F(i-1,j), v_i + F(i-1, j-w_i)\}$.
- Condições iniciais (caso base):
  $F(0,j) = 0, \forall j \geq 0;$
  $F(i,0) = 0, \forall i \geq 0.$
- Queremos encontrar (**objetivo**): $F(n, W)$.

Exemplo: Problema da Mochila

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

$$F(0, j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \geq 0.$$

Exemplo: Problema da Mochila

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

$$F(0, j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \geq 0.$$

Tabela a ser montada:

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

O objetivo é encontrar o valor de $F(n, W)$ (o custo máximo considerando n itens e uma mochila de capacidade W).

Exemplo: Problema da Mochila

Exercício 1

Resolva, passo a passo, o seguinte problema da mochila (montar a tabela F[0..n, 0..W]):
Número de itens $n = 4$
Capacidade da mochila: $W = 5$
Custos (valores) dos itens: $v_1 = 12$, $v_2 = 10$, $v_3 = 20$, $v_4 = 15$.
Pesos dos itens: $w_1 = 2$, $w_2 = 1$, $w_3 = 3$, $w_4 = 2$.

Exemplo: Problema da Mochila

$$F(i,j) = \begin{cases} F(i-1,j), & w_i > j \\ max\{F(i-1,j), F(i-1,j-w_i)+v_i\}, & w_i \leq j \end{cases}$$

$F(0, j) = 0$ for $j \geq 0$ and $F(i, 0) = 0$ for $i \geq 0$.

Cálculos para montar a tabela:

$F(0,j) = 0$, para $j = 0, 1, ...5$. $F(i,0) = 0$, para $i = 0, 1, ..., 4$

$F(1,1) = F(0,1)$, pois $w_1 > 1$.

$F(1,2) = max\{F(0,2), F(0,0) + v_1\} = max\{0, 12\} = 12$, $(w_1 \leq 2)$.

$F(1,3) = max\{F(0,3), F(0,1) + v_1\} = max\{0, 12\} = 12$, $(w_1 < 3)$.

$F(1,4) = max\{F(0,4), F(0,2) + v_1\} = max\{0, 12\} = 12$, $(w_1 < 4)$.

$F(1,5) = max\{F(0,5), F(0,3) + v_1\} = max\{0, 12\} = 12$, $(w_1 < 5)$.

$F(2,1) = max\{F(1,1), F(1,0) + v_2\} = max\{0, 10\} = 10$, $(w_2 \leq 1)$.

$F(2,2) = max\{F(1,2), F(1,1) + v_2\} = max\{12, 10\} = 12$, $(w_2 < 2)$.

$F(2,3) = max\{F(1,3), F(1,2) + v_2\} = max\{12, 12 + 10\} = 22$.

$F(2,4) = max\{F(1,4), F(1,3) + v_2\} = max\{12, 12 + 10\} = 22$.

$F(2,5) = max\{F(1,5), F(1,4) + v_2\} = max\{12, 12 + 10\} = 22$.

$F(3,1) = F(2,1) = 10$, $(w_3 > 1)$.
$F(3,2) = F(2,2) = 12$, $(w_3 > 2)$.
$F(3,3) = max\{F(2,3), F(2,0) + v_3\} = max\{22, 0 + 20\} = 22$.
$F(3,4) = max\{F(2,4), F(2,1) + v_3\} = max\{22, 10 + 20\} = 30$.
$F(3,5) = max\{F(1,5), F(1,2) + v_3\} = max\{12, 12 + 20\} = 32$.
$F(4,1) = F(2,1) = 10$, $(w_4 > 1)$.
$F(4,2) = max\{F(3,2), F(3,0) + v_4\} = max\{12, 0 + 15\} = 15$.
$F(4,3) = max\{F(3,3), F(3,1) + v_4\} = max\{22, 10 + 15\} = 25$.
$F(4,4) = max\{F(3,4), F(3,2) + v_4\} = max\{30, 12 + 15\} = 30$.
$F(4,5) = max\{F(3,5), F(3,3) + v_4\} = max\{32, 22 + 15\} = 37$.

A tabela obtida é apresentada a seguir.

Exemplo: Problema da Mochila

$W = 5$

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

$W = 5$

	i	capacity j 0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

Exercício 2

Escreva o algoritmo não recursivo que implementa a relacao de recorrência para construir a tabela $F[0..n, 0..W]$ e retorne o valor máximo $F[n, W]$ dos itens inseridos na mochila.

Exemplo: Problema da Mochila

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

Os itens a serem inseridos na mochila pode ser encontrado por retrocesso:

- Como $F(4,5) > F(3,5)$, o **item 4** deve ser incluído na mochila. Capacidade da mochila diminui: $5 - 2 = 3$.
- Uma vez que $F(3,3) = F(2,3)$, o **item 3** não será inserido na mochila.
- Como $F(2,3) > F(1,3)$, o **item 2** é inserido na mochila. Capacidade da mochila diminui: $3 - 1 = 2$.
- Como $F(1,2) > F(0,2)$, o **item 1** é inserido na mochila.
- Então, o conjunto de itens da solução ótima é: {item 1, item 2, item 4}.

Exemplo: Problema da Mochila

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

Os itens a serem inseridos na mochila pode ser encontrado por retrocesso:

- Como $F(4,5) > F(3,5)$, o **item 4** deve ser incluído na mochila. Capacidade da mochila diminui: $5 - 2 = 3$.
- Uma vez que $F(3,3) = F(2,3)$, o **item 3** não será inserido na mochila.
- Como $F(2,3) > F(1,3)$, o **item 2** é inserido na mochila. Capacidade da mochila diminui: $3 - 1 = 2$.
- Como $F(1,2) > F(0,2)$, o **item 1** é inserido na mochila.
- Então, o conjunto de itens da solução ótima é: {item 1, item 2, item 4}.

Exercício 3

Escreva um algoritmo/procedimento para imprimir os itens a serem inseridos na mochila a partir da tabela $F[0..n, 0..W]$.

Exemplo: Problema da Mochila

- Um aspecto insatisfatório da abordagem **bottom-up** é que algumas soluções de subproblemas menores não são utilizadas.
- O algoritmo pode ser melhorado, combinando as abordagens **top-down** e **bottom-up**. O objetivo é obter um método que resolva apenas uma vez os subproblemas necessários.
- O método, chamado *memory functions* (MF), resolve um problema na forma **top-down**, além disso, mantém uma **tabela** como é usada por um algoritmo **bottom-up**.
- Inicialmente, todas as entradas da tabela são inicializadas com um símbolo "nulo"indicando que elas ainda não foram calculadas.
- Posteriormente, sempre que um novo valor precisa ser calculado, o método verifica primeiro a entrada correspondente na tabela: Se essa entrada não for "nula", ela é simplesmente recuperada da tabela; Caso contrário, é calculado pela chamada recursiva cujo resultado é gravado na tabela.

Exemplo: Problema da Mochila

```
ALGORITHM  MFKnapsack(i, j)
    //Implements the memory function method for the knapsack problem
    //Input: A nonnegative integer i indicating the number of the first
    //       items being considered and a nonnegative integer j indicating
    //       the knapsack capacity
    //Output: The value of an optimal feasible subset of the first i items
    //Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
    //and table F[0..n, 0..W] whose entries are initialized with −1's except for
    //row 0 and column 0 initialized with 0's
    if F[i, j] < 0
        if j < Weights[i]
            value ← MFKnapsack(i − 1, j)
        else
            value ← max(MFKnapsack(i − 1, j),
                        Values[i] + MFKnapsack(i − 1, j − Weights[i]))
        F[i, j] ← value
    return F[i, j]
```

Exemplo: Problema da Mochila

Tabela determinada pelo método MF:

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3,\ v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2,\ v_4 = 15$	4	0	—	—	—	—	**37**

Exemplo: Problema da Mochila

Complexidade:

Claramente, a complexidade do algoritmo de PD para o problema da mochila é $O(nW)$.

É um algoritmo **pseudo-polinomial**: sua complexidade depende do valor de W (capacidade da mochila).

Fecho transitivo

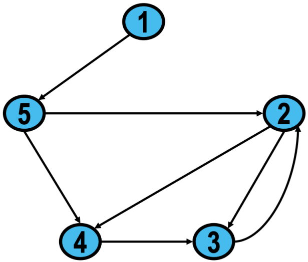
Um **grafo direcionado** (ou **dígrafo**) é uma estrutura $G = (V, E)$ formada por: um conjunto V, cujos elementos são chamados **vértices** ou **nodos**, e um conjunto E de pares ordenados de vértices, chamados **arcos**, ou **arestas direcionadas**.

Exemplo:

G = (V, E)

V = {1, 2, 3, 4, 5}

E = {(1, 5), (2, 3), (2, 4), (3, 2), (4, 3), (5, 2), (5, 4)}



Fecho transitivo

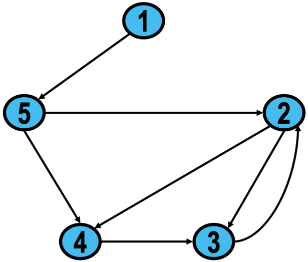
Um **grafo direcionado** (ou **dígrafo**) é uma estrutura $G = (V, E)$ formada por: um conjunto V, cujos elementos são chamados **vértices** ou **nodos**, e um conjunto E de pares ordenados de vértices, chamados **arcos**, ou **arestas direcionadas**.

Exemplo:

G = (V, E)

V = {1, 2, 3, 4, 5}

E = {(1, 5), (2, 3), (2, 4), (3, 2), (4, 3), (5, 2), (5, 4)}



Caminho direcionado: é uma sequência de arestas, todas dirigidas no mesmo sentido.

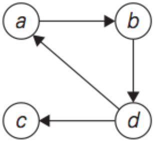
Exemplo. No grafo acima, um caminho direcionado de 1 para 3 é: $1 \to 5 \to 2 \to 3$. Não esixte caminho direcionado de 3 para 1.

Fecho transitivo

O **fecho transitivo** de um grafo direcionado com n vértices pode ser definido como uma matriz booleana T de tamanho $n \times n$ onde a entrada $t_{ij} = 1$ se existe um caminho direcionado entre os vértices i e j; $t_{ij} = 0$ caso contrário.

Fecho transitivo

O **fecho transitivo** de um grafo direcionado com n vértices pode ser definido como uma matriz booleana T de tamanho $n \times n$ onde a entrada $t_{ij} = 1$ se existe um caminho direcionado entre os vértices i e j; $t_{ij} = 0$ caso contrário.



(a)

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

(b)

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

(c)

(a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

Problema: determinar o fecho transitivo de um grafo direcionado.

Fecho transitivo

- O fecho transitivo pode ser computado através dos algoritmos de busca em profundidade (BP) e largura (BL).
- A BP, por exemplo, deve ser aplicado a partir de cada vértice v do grafo, obtendo os vértices alcançáveis a partir de v.
- Ou seja, este método percorre o mesmo grafo várias vezes.

Fecho transitivo

- O fecho transitivo pode ser computado através dos algoritmos de busca em profundidade (BP) e largura (BL).
- A BP, por exemplo, deve ser aplicado a partir de cada vértice v do grafo, obtendo os vértices alcançáveis a partir de v.
- Ou seja, este método percorre o mesmo grafo várias vezes.
- O **Algoritmo de Warshall**, baseado em Programação Din6amica, é usado para determinar o fecho transitivo de um grafo direcionado.

Fecho transitivo

- O algoritmo de Warshall constrói o fecho transitivo através de uma série de matrizes booleanas $n \times n$:
  $R^{(0)}, R^{(1)}, \ldots, R^{(k)}, \ldots, R^{(n)}$.

Fecho transitivo

- O algoritmo de Warshall constrói o fecho transitivo através de uma série de matrizes booleanas $n \times n$: $R^{(0)}, R^{(1)}, \ldots, R^{(k)}, \ldots, R^{(n)}$.
- Matrizes $R^{(k)}$ consideram uma ordem $\{1, 2, \cdots, n\}$ dos vértices.
- O elemento $r_{ij}^{(k)} = 1$ se e somente se existe um caminho direcionado entre i e j onde cada vértice intermediário (se houver algum) não é maior que k.

Fecho transitivo

- A série começa com $R^{(0)}$, onde não é permitido nenhum vértice intermediário entre os caminhos direcionados existentes. $R^{(0)} = MatrizAdjacencia(G)$.
- $R^{(1)}$ contém informação sobre os caminhos que podem utilizar o **primeiro vértice** como intermediário.
- $R^{(2)}$ contém informação sobre caminhos que podem utilizar **os dois primeiros vértices** como intermediários.
- De uma forma geral, $R^{(k)}$ contém informação sobre caminhos que podem utilizar **os** k **primeiros vértices** como intermediários.
- A matriz $R^{(n)}$ permite que todos os vértices sejam utilizados como intermediários; $R^{(n)}$ é o fecho transitivo.

Fecho transitivo

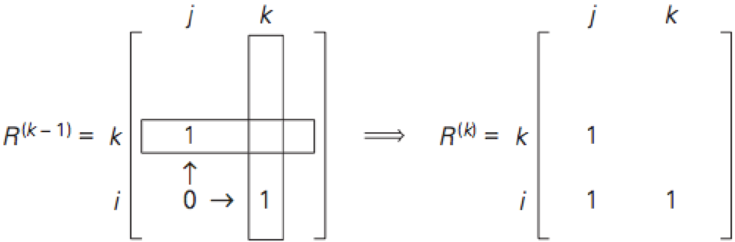
- Ideia central: $R^{(k)}$ pode ser calculado a partir de $R^{(k-1)}$.

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} or \left(r_{ik}^{(k-1)} and \; r_{kj}^{(k-1)} \right)$$

Fecho transitivo

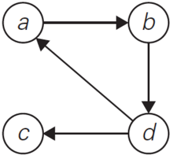
- Ideia central: $R^{(k)}$ pode ser calculado a partir de $R^{(k-1)}$.

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \, or \left(r_{ik}^{(k-1)} \, and \, r_{kj}^{(k-1)} \right)$$



Rule for changing zeros in Warshall's algorithm.

Fecho transitivo - Exemplo



$$R^{(0)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

$$R^{(1)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & \mathbf{1} & 1 & 0 \end{bmatrix} \end{array}$$

$$R^{(2)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 1 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & \mathbf{1} \end{bmatrix} \end{array}$$

$$R^{(3)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

$$R^{(4)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} \mathbf{1} & 1 & \mathbf{1} & 1 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

As linhas e colunas resaltadas em R^{k-1} são usadas para obter R^k

Fecho transitivo

ALGORITHM *Warshall*$(A[1..n, 1..n])$

//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph

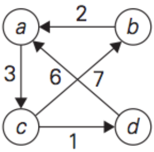
$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$

Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

Complexidade: $O(n^3)$

Todos os Pares de Caminhos Mínimos

A **matriz de distâncias** de um grafo direcionado ou não direcionado com n vértices é definida como uma matriz D de tamanho $n \times n$ onde o elemento d_{ij} é o custo do caminho mais curto entre i and j.



(a)

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

(b)

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

(c)

RE 8.14 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

$\Rightarrow$ Determinar a matriz de distâncias de um grafo direcionado ou não direcionado. Ou seja, determinar os caminhos de custo mínimo entre cada par de vértices.

Todos os Pares de Caminhos Mínimos

- O algoritmo de **Dijkstra** pode ser utilizado n vezes, caso não haja arestas de custo negativo.
  - Complexidade para grafos densos: $O(n^3 \log n)$;
- Algoritmo de **Bellman-Ford** pode ser aplicado caso haja arestas negativas: $O(n^4)$ para grafos densos.
- Algoritmo de **Floyd-Warshall** garante complexidade $O(n^3)$ mesmo com arestas negativas.

Todos os Pares de Caminhos Mínimos

- O algoritmo de **Floyd-Warshall** calcula todos os pares de caminhos mais curtos através de uma série de matrizes $n \times n$: $D^{(0)}, D^{(1)}, \ldots, D^{(k)}, \ldots, D^{(n)}$

- O algoritmo de **Floyd-Warshall** calcula todos os pares de caminhos mais curtos através de uma série de matrizes $n \times n$: $D^{(0)}, D^{(1)}, \ldots, D^{(k)}, \ldots, D^{(n)}$
- $D^{(0)} = W$ (matriz de custos), ou seja, $d_{ij}^{(0)} = w_{ij}, \forall i, j$
- O elemento $d_{ij}^{(k)}$ é o custo do menor caminho entre i e j considerando que cada vértice intermediário (se houver algum) é não é maior que k.

Todos os Pares de Caminhos Mínimos

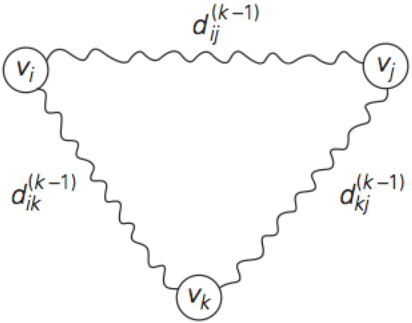
- Como no algoritmo de Warshall, os valores $d_{ij}^{(k)}$ podem ser calculados a partir dos valores $d_{ij}^{(k-1)}$:

$$d_{ij}^{(k)} = min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$

Todos os Pares de Caminhos Mínimos

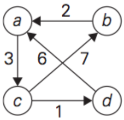
- Como no algoritmo de Warshall, os valores $d_{ij}^{(k)}$ podem ser calculados a partir dos valores $d_{ij}^{(k-1)}$:

$$d_{ij}^{(k)} = min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$



Underlying idea of Floyd's algorithm.

Todos os Pares de Caminhos Mínimos - Exemplo



$$D^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$$D^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$$D^{(2)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and *b* (note a new shortest path from *c* to *a*).

$$D^{(3)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$$D^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

ALGORITHM *Floyd*$(W[1..n, 1..n])$

//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
$D \leftarrow W$ //is not necessary if W can be overwritten
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$
return D

Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

Complexidade: $O(n^3)$