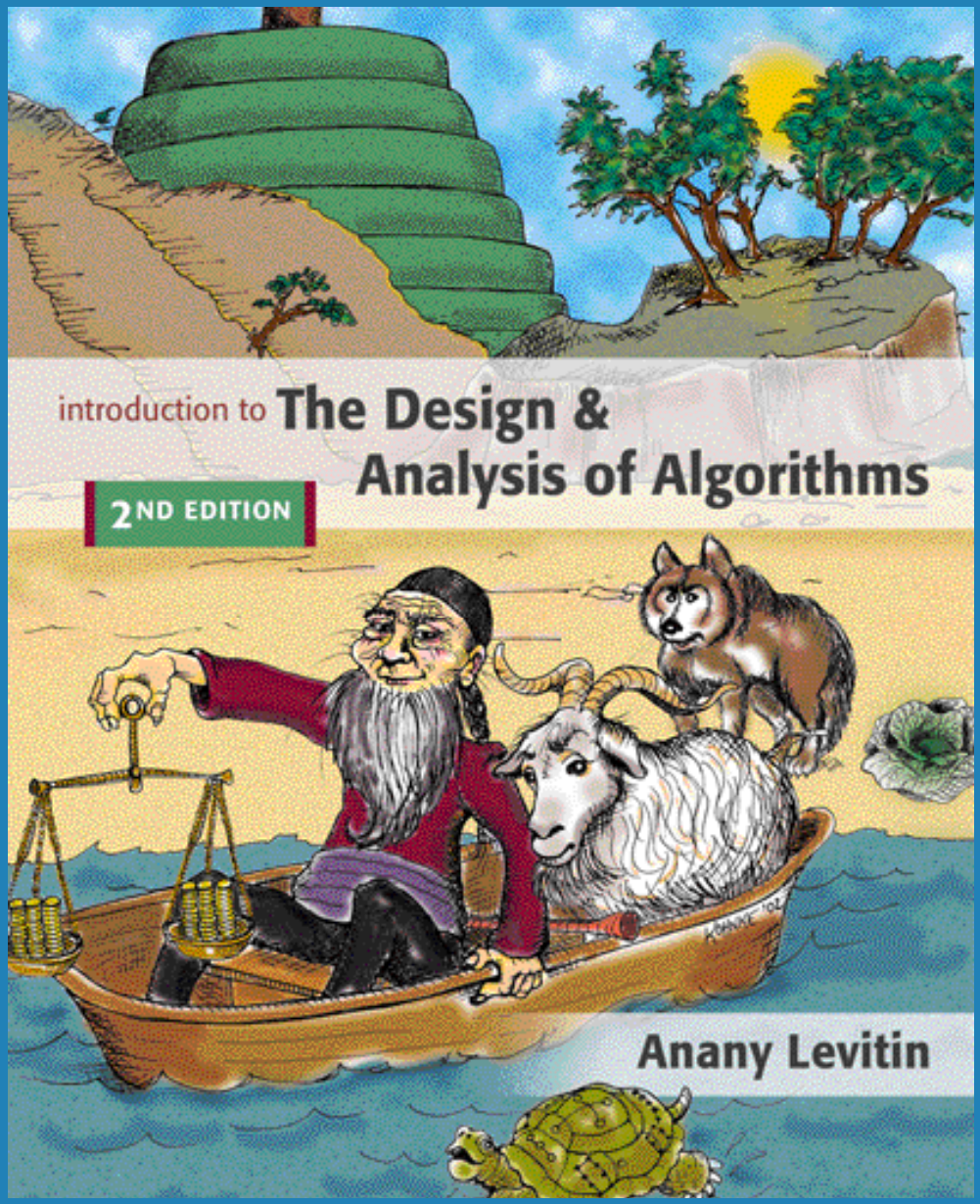
Chapter 8

Dynamic Programming



PEARSON Addison Wesley

Dynamic Programming

D*ynamic Programming* is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems

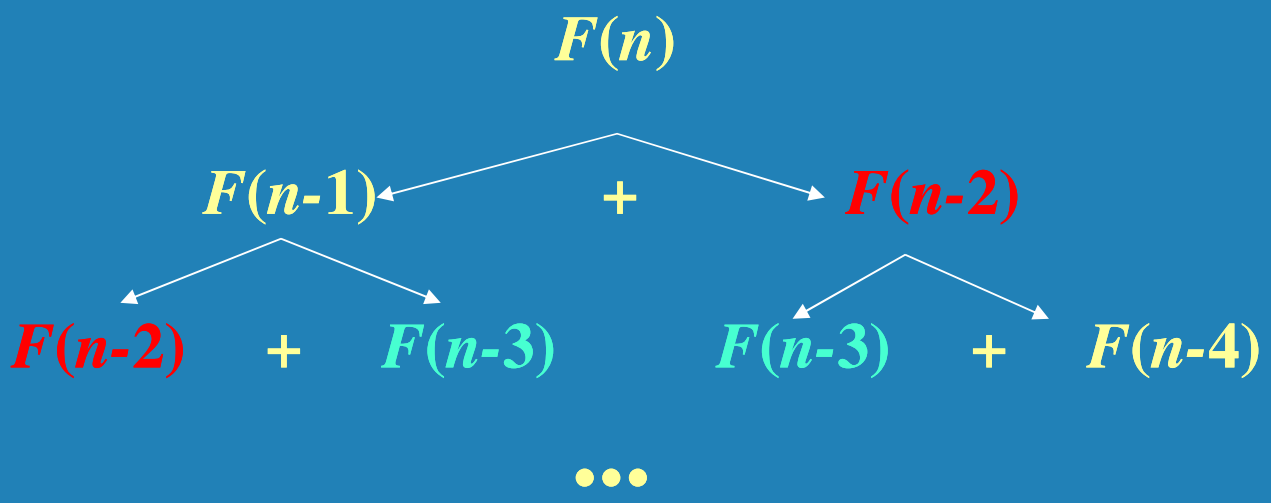
- **Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS**
- **"Programming" here means "planning"**
- **Main idea:**
  - **set up a recurrence relating a solution to a larger instance to solutions of some smaller instances**
  - **solve smaller instances once**
  - **record solutions in a table**
  - **extract solution to the initial instance from that table**

Example: Fibonacci numbers

- **Recall definition of Fibonacci numbers:**

$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$
$$F(0) = 0$$
$$F(1) = 1$$

- **Computing the n^{th} Fibonacci number recursively (top-down):**



Example: Fibonacci numbers (cont.)

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$F(0) = 0$
$F(1) = 1$
$F(2) = 1+0 = 1$
…
$F(n\text{-}2) =$
$F(n\text{-}1) =$
$F(n) = F(n\text{-}1) + F(n\text{-}2)$

0	1	1	. . .	$F(n\text{-}2)$	$F(n\text{-}1)$	$F(n)$

Efficiency:
- time
- space

Example: Fibonacci numbers (cont.)

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$F(0) = 0$
$F(1) = 1$
$F(2) = 1+0 = 1$
…
$F(n-2) =$
$F(n-1) =$
$F(n) = F(n-1) + F(n-2)$

```
Algorithm Fib(n)
 F[0] ← 0; F[1] ← 1;
 for i ← 2 to n
     do F[i] ← F[i-1] + F[i-2];
 return F[n];
```

0	1	1	. . .	$F(n-2)$	$F(n-1)$	$F(n)$

Efficiency:
- time
- space

Como melhorar Fib(n) para usar menos espaço?

Essência de DP algoritmos

- clássico *bottom-up*: resolve todos os sub-problemas menores de um dado problema

- variação *top-down*: evita resolver sub-problemas desnecessários

- o passo crucial sempre é: derivar uma recorrência que relaciona uma solução para o problema maior com as soluções de suas intâncias menores (e repetidas)

Examples of DP algorithms

- Computing a binomial coefficient
- Warshall's algorithm for transitive closure
- Floyd's algorithm for all-pairs shortest paths
- Constructing an optimal binary search tree
- Some instances of difficult discrete optimization problems:
  - travelling salesman
  - knapsack

Computing a binomial coefficient by DP

- *C(n,k) é o número de combinações (subconjuntos) de k elementos de um conjunto de n elementos (0 <= k <= n)*

Computing a binomial coefficient by DP

Binomial coefficients are coefficients of the binomial formula:

$$(a + b)^n =$$
$$C(n,0)a^n b^0 + \ldots + C(n,k)a^{n-k}b^k + \ldots + C(n,n)a^0 b^n$$

Recurrence:

$$C(n,k) = C(n\text{-}1,k) + C(n\text{-}1,k\text{-}1) \text{ for } n > k > 0$$
$$C(n,0) = 1, \quad C(n,n) = 1 \text{ for } n \geq 0$$

Value of $C(n,k)$ can be computed by filling a table

n linhas

k colunas

Computing a binomial coefficient by DP

Recurrence: $C(n,k) = C(n\text{-}1,k) + C(n\text{-}1,k\text{-}1)$ for $n > k > 0$

$C(n,0) = 1, \quad C(n,n) = 1$ for $n \geq 0$

	0	1	2	. . .	k-1	k
0	1					
1	1	1				
2	1	2	1			
...						
k	1					1
...						
n-1					$C(n\text{-}1,k\text{-}1)$	$C(n\text{-}1,k)$
n						$C(n,k)$

Pascal'sTriangle!

Computing $C(n,k)$: pseudocode and analysis

ALGORITHM *Binomial*(n, k)

```
//Computes C(n, k) by the dynamic programming algorithm
//Input: A pair of nonnegative integers n ≥ k ≥ 0
//Output: The value of C(n, k)
for i ← 0 to n do
    for j ← 0 to min(i, k) do
        if j = 0 or j = i
            C[i, j] ← 1
        else C[i, j] ← C[i − 1, j − 1] + C[i − 1, j]
return C[n, k]
```

Time efficiency: $\Theta(nk)$

Space efficiency: $\Theta(nk)$

Computing $C(n,k)$: pseudocode and analysis

Time efficiency:

$$A(n,k) = \sum_{i=1}^{k} \sum_{i=1}^{i-1} 1 + \sum_{i=k+1}^{n} \sum_{i=1}^{k} 1$$

$$= \sum_{i=1}^{k} (i-1) + \sum_{i=k+1}^{n} k$$

$$= ((k-1)k)/2 + k(n-k)$$

$$= \Theta(nk)$$

Space efficiency: $\Theta(nk)$

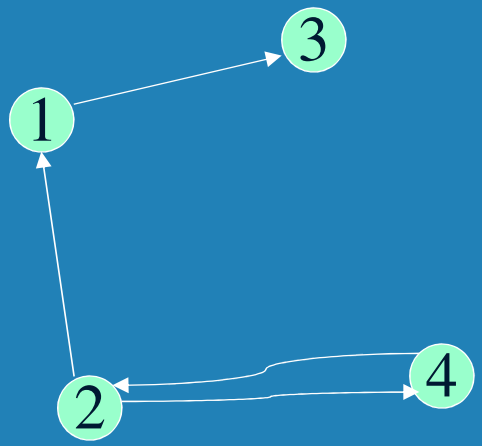
- *todo o espaço extra é realmente necessário?*

Warshall's Algorithm: Transitive Closure

- Computes the transitive closure of a relation
- Alternatively: existence of all nontrivial paths in a digraph

  fecho transitivo de um grafo direcionado:
  matriz *T=t{i,j} nxn* tal que o elemento [i,j] é 1 se existir um caminho não negativo entre os vértices *i* e *j*

- Example of transitive closure:

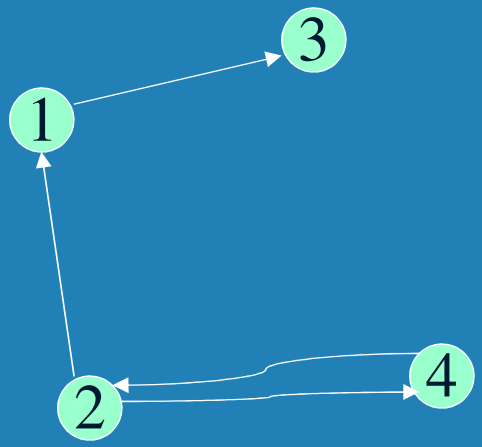


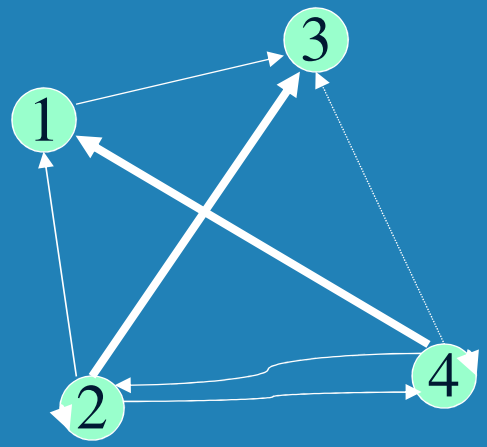
Warshall's Algorithm: Transitive Closure

- **Computes the transitive closure of a relation**
- **Alternatively: existence of all nontrivial paths in a digraph**

fecho transitivo de um grafo direcionado: matriz *T=t{i,j} nxn* tal que o elemento [i,j] é 1 se existir um caminho não negativo entre os vértices *i* e *j*

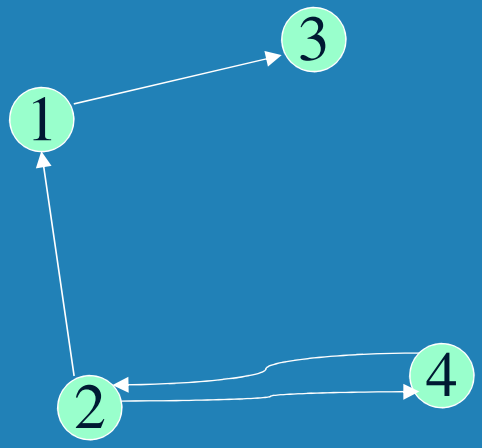
- **Example of transitive closure:**



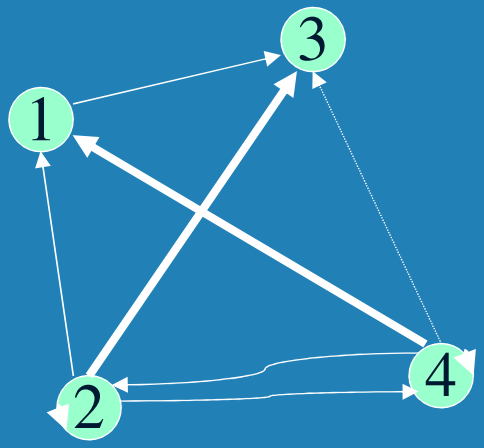


Warshall's Algorithm: Transitive Closure

- **Computes the transitive closure of a relation**
- **Alternatively: existence of all nontrivial paths in a digraph**
- **Example of transitive closure:**



0 0 1 0
1 0 0 1
0 0 0 0
0 1 0 0

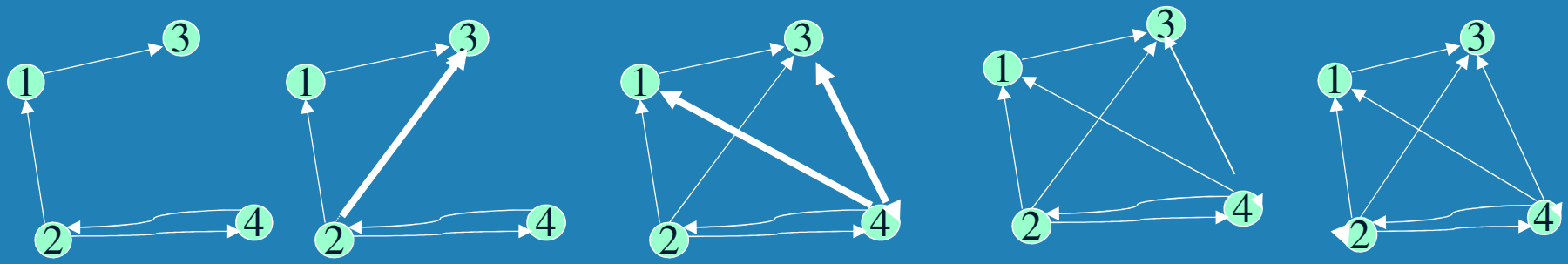


0 0 1 0
1 **1** **1** 1
0 0 0 0
1 1 **1** **1**

Como calcular o fecho transitivo usando DFS? e BFS?

Warshall's Algorithm [S.Warshall 1962]

Constructs transitive closure T as the last matrix in the sequence of n-by-n matrices $R^{(0)}, \ldots, R^{(k)}, \ldots, R^{(n)}$ where
$R^{(k)}[i,j] = 1$ iff there is nontrivial path from i to j with only the first k vertices allowed as intermediate
Note that $R^{(0)} = A$ (adjacency matrix), $R^{(n)} = T$ (transitive closure)



$R^{(0)}$

	1	2	3	4
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0
4	0	1	0	0

$R^{(1)}$

	1	2	3	4
1	0	0	1	0
2	1	0	**1**	1
3	0	0	0	0
4	0	1	0	0

$R^{(2)}$

	1	2	3	4
1	0	0	1	0
2	1	0	1	1
3	0	0	0	0
4	**1**	1	**1**	**1**

$R^{(3)}$

	1	2	3	4
1	0	0	1	0
2	1	0	1	1
3	0	0	0	0
4	1	1	1	1

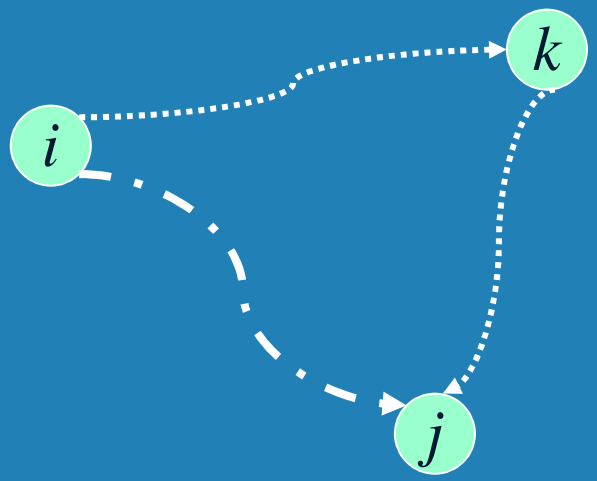
$R^{(4)}$

	1	2	3	4
1	0	0	1	0
2	1	**1**	1	1
3	0	0	0	0
4	1	1	1	1

Warshall's Algorithm (recurrence)

On the k-th iteration, the algorithm determines for every pair of vertices i, j if a path exists from i and j with just vertices $1,\ldots,k$ allowed as intermediate

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just } 1,\ldots,k-1) \\ \quad \text{or} & \\ R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j] & \text{(path from } i \text{ to } k \\ & \quad \text{and from } k \text{ to } i \\ & \quad \text{using just } 1,\ldots,k-1) \end{cases}$$



Warshall's Algorithm (matrix generation)

Recurrence relating elements $R^{(k)}$ to elements of $R^{(k-1)}$ is:

$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

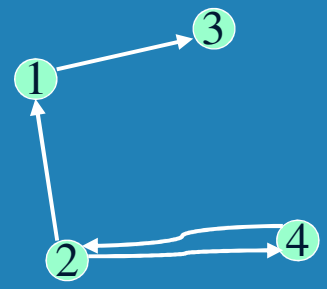
It implies the following rules for generating $R^{(k)}$ from $R^{(k-1)}$:

Rule 1 If an element in row *i* and column *j* is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$

Rule 2 If an element in row *i* and column *j* is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ *if and only if* the element in its row *i* and column *k* and the element in its column *j* and row *k* are both 1's in $R^{(k-1)}$

Warshall's Algorithm (example)

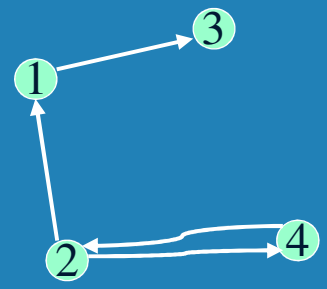
If an element in row i and column j is 0 in R(k-1), it has to be changed to 1 in R(k) if and only if the element in its row i and column k and the element in its column j and row k are both 1's in R(k-1)



$$R^{(0)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

Warshall's Algorithm (example)

If an element in row i and column j is 0 in R(k-1), it has to be changed to 1 in R(k) if and only if the element in its row i and column k and the element in its column j and row k are both 1's in R(k-1)



$R^{(0)} =$

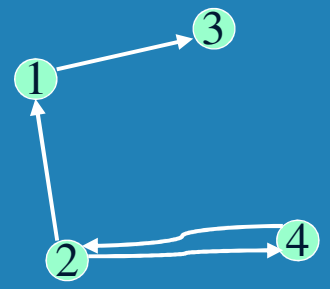
0	0	1	0
1	0	0	1
0	0	0	0
0	1	0	0

$R^{(1)} =$

0	0	1	0
1	0	1	1
0	0	0	0
0	1	0	0

Warshall's Algorithm (example)

If an element in row i and column j is 0 in R(k-1), it has to be changed to 1 in R(k) if and only if the element in its row i and column k and the element in its column j and row k are both 1's in R(k-1)



$R^{(0)} =$

0	0	1	0
1	0	0	1
0	0	0	0
0	1	0	0

$R^{(1)} =$

0	0	1	0
1	0	1	1
0	0	0	0
0	1	0	0

$R^{(2)} =$

0	0	1	0
1	0	1	1
0	0	0	0
1	1	1	1

$R^{(3)} =$

0	0	1	0
1	0	1	1
0	0	0	0
1	1	1	1

$R^{(4)} =$

0	0	1	0
1	1	1	1
0	0	0	0
1	1	1	1

Warshall's Algorithm (pseudocode and analysis)

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) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R^(k)[i, j] ← R^(k−1)[i, j] or (R^(k−1)[i, k] and R^(k−1)[k, j])
return R^(n)
```

Time efficiency: $\Theta(n^3)$
Space efficiency: Matrices can be written over their predecessors
Grafos esparsos: melhor que BFS/DFS?
Loop pode ser melhorado?
Pode usar *bitwise or?*

Idéia aplicada no Floyd's APSP

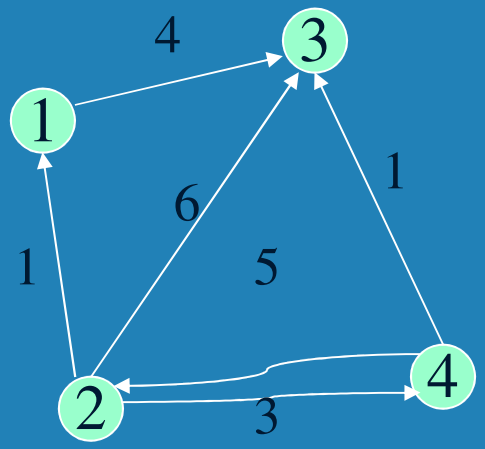
Floyd's Algorithm: All pairs shortest paths

[R.Floyd 1962]

Problem: In a weighted (di)graph, find shortest paths between every pair of vertices

Same idea: construct solution through series of matrices $D^{(0)}$, …, $D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

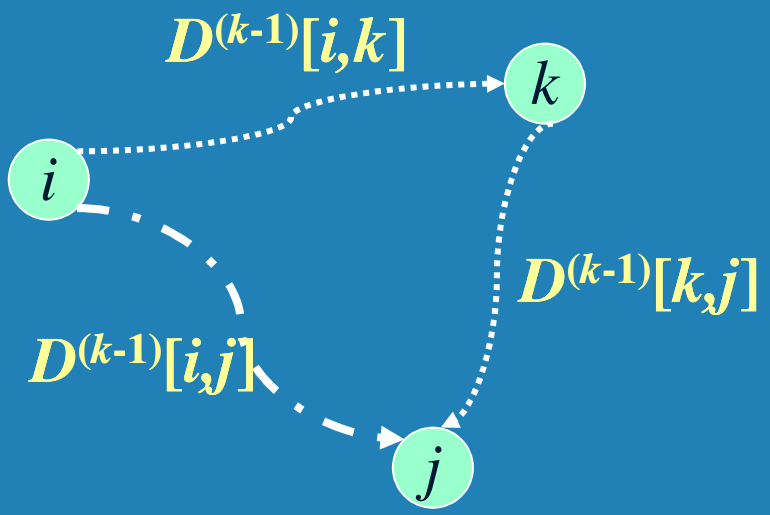
Example:



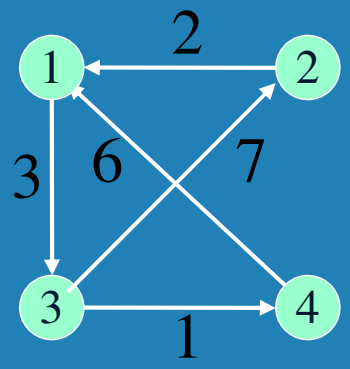
Floyd's Algorithm (matrix generation)

On the *k*-th iteration, the algorithm determines shortest paths between every pair of vertices *i, j* that use only vertices among 1,…,*k* as intermediate

$$D^{(k)}[i,j] = \min \{D^{(k-1)}[i,j],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$



Floyd's Algorithm (example)



$$D^{(0)} = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \qquad D^{(1)} = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \qquad D^{(3)} = \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \qquad D^{(4)} = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix}$$

Floyd's Algorithm (pseudocode and analysis)

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 ← W //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

Time efficiency: $\Theta(n^3)$

Space efficiency: Matrices can be written over their predecessors

Note: Shortest paths themselves can be found, too

como?

Criadores...

- Weisstein, Eric W. "Floyd-Warshall Algorithm." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/Floyd-WarshallAlgorithm.html

  - The Floyd-Warshall algorithm, also variously known as Floyd's algorithm, the Roy-Floyd algorithm, the Roy-Warshall algorithm, or the WFI algorithm, is an algorithm for efficiently and simultaneously finding the shortest paths (i.e., graph geodesics) between every pair of vertices in a weighted and potentially directed graph.

  - The Floyd algorithm is essentially equivalent to the transitive closure algorithm independently discovered by Roy (1959) and Warshall (1962) (Pemmaraju and Skiena 2003), which is the reason it is associated with all three authors.

  - Floyd, R. W. "Algorithm 97." Comm. ACM 5-6, 345, 1962.

  - Roy, B. "Transitivité et connexité." C. R. Acad. Sci. Paris 249, 216-218, 1959.

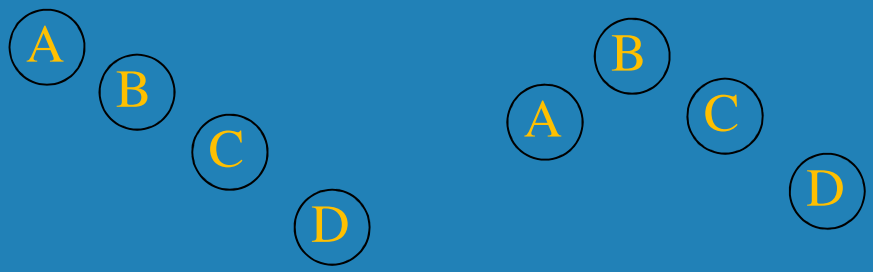
  - Warshall, S. "A Theorem on Boolean Matrices." J. ACM 9, 11-12, 1962.

Optimal Binary Search Trees

Problem: Given n keys $a_1 < \ldots < a_n$ and probabilities $p_1 \leq \ldots \leq p_n$ searching for them, find a BST with a minimum average number of comparisons in a *successful* search.

Since total number of BSTs with n nodes is *(Catalan number)* given by $C(2n,n)/(n+1)$, which grows exponentially, brute force is hopeless.

Example: What is an optimal BST for keys A, B, C, and D with search probabilities 0.1, 0.2, 0.4, and 0.3, respectively?

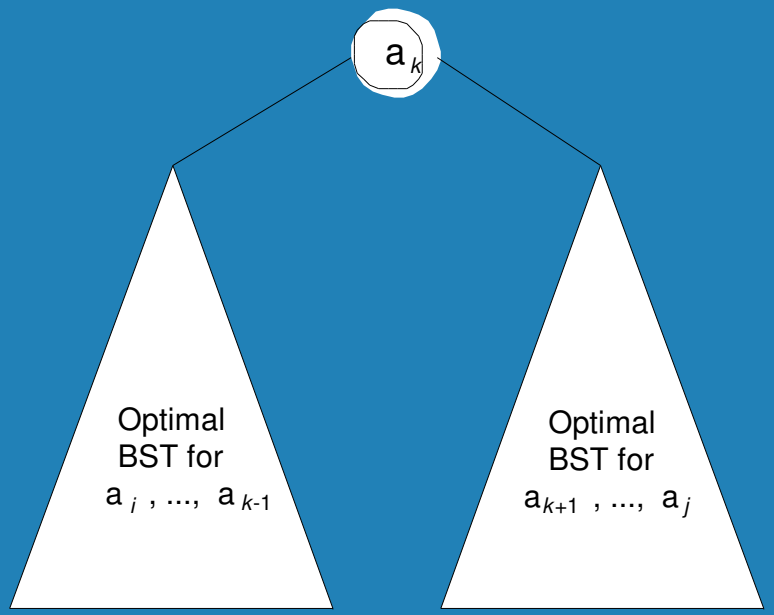


Força-bruta:
- ache as 14 árvores deste exemplo

DP for Optimal BST Problem

Let $a_1 \ldots a_n$ be distinct keys ordered from the smallest to the largest, and $p_1 \ldots p_n$ be the probabilities of (successfully) searching for them. Let C[i,j] be the smallest number of comparisons in BST T[i,j], composta pela chaves ai,.. aj (1 <= i <= l <= n). If we count tree levels starting with 1 (to make the number of comparisons equal to key's level) ....

Let $C[i,j]$ be minimum average number of comparisons made in T[i,j], optimal BST for keys $a_i < \ldots < a_j$, where $1 \le i \le j \le n$. Consider optimal BST among all BSTs with some a_k $(i \le k \le j)$ as their root; T[i,j] is the best among them.



$$C[i,j] = \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \text{ (level } a_s \text{ in T}[i,k\text{-}1] +1) + \sum_{s=k+1}^{j} p_s \text{ (level } a_s \text{ in T}[k+1,j] +1)\}$$

DP for Optimal BST Problem (cont.)

After simplifications, we obtain the recurrence for $C[i,j]$:

$$C[i,j] = \min_{i \le k \le j} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n$$

$$C[i,i] = p_i \quad \text{for } 1 \le i \le j \le n$$

- one node BT with a_i

$$C[i,i-1] = 0 \quad \text{for } 1 \le i \le n+1$$

- comparações em árvore vazia

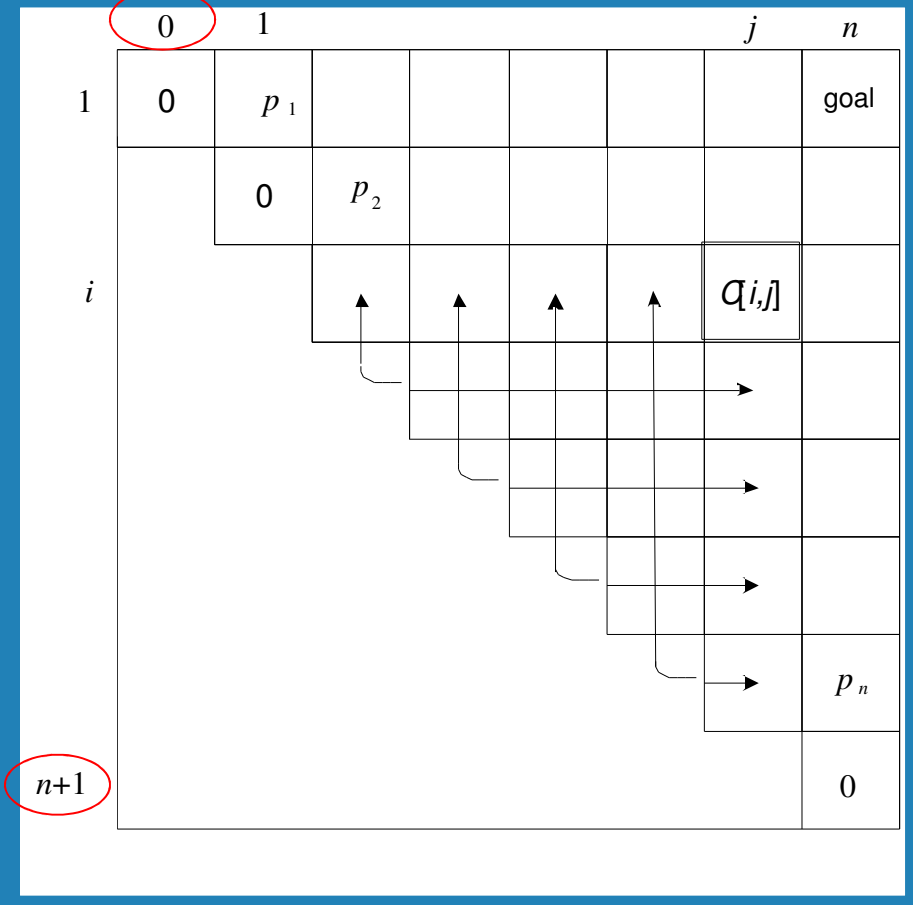
DP for Optimal BST Problem (cont.)

After simplifications, we obtain the recurrence for $C[i,j]$:

$$C[i,j] = \min_{i \le k \le j} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n$$

$$C[i,i] = p_i \quad \text{for } 1 \le i \le j \le n$$

$$C[i,i-1] = 0 \quad \text{for } 1 \le i \le n+1$$



Valores necessários:

- na linha ***i***, nas colunas à esquerda de ***j***
- na coluna ***j***, nas linhas abaixo de ***i***

Isso sugere preencher a tabela pelas diagonais

- começando com ***zeros*** na diagonal principal
- com as probabilidades p_i acima da diagonal
- e preencher a tabela em direção a posição [1,n]

Example:	key	*A*	*B*	*C*	*D*
	probability	0.1	0.2	0.4	0.3

The tables below are filled diagonal by diagonal: the left one is filled using the recurrence

$$C[i,j] = \min_{i \le k \le j} \{C[i,k\text{-}1] + C[k\text{+}1,j]\} + \sum_{s=i}^{j} p_s, \quad C[i,i] = p_i;$$

the right one, for trees' roots, records *k*'s values giving the minimal value

i \ *j*	0	1	2	3	4
1	0	.1			
2		0	.2		
3			0	.4	
4				0	.3
5					0

i \ *j*	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

Example:	key	*A*	*B*	*C*	*D*
	probability	0.1	0.2	0.4	0.3

$k=1$: $C[1,0] + C[2,2] + \sum_{s=1,2} p_s = 0 + 0.2 + (0.3) = 0.5$

$C[1,2] = \min$ $= 0.4$

$k=2$: $C[1,1] + C[3,2] + \sum_{s=1,2} p_s = 0.1 + 0 + (0.3) = 0.4$

i \ *j*	0	1	2	3	4
1	0	.1	.4		
2		0	.2		
3			0	.4	
4				0	.3
5					0

i \ *j*	0	1	2	3	4
1		1	2		
2			2		
3				3	
4					4
5					

Das duas BST possíveis com os dois primeiros nós (A e B)

- *a raiz da BST tem índice 2 (B) e*
- *o nro médio de comparações em caso de sucesso é 0.4*

Example:

key	*A*	*B*	*C*	*D*
probability	0.1	0.2	0.4	0.3

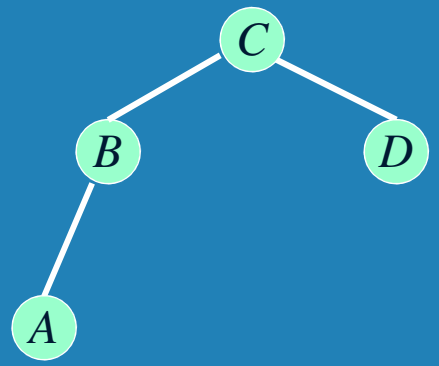
The tables below are filled diagonal by diagonal: the left one is filled using the recurrence

$$C[i,j] = \min_{i \le k \le j} \{C[i,k\text{-}1] + C[k\text{+}1,j]\} + \sum_{s=i}^{j} p_s, \quad C[i,i] = p_i;$$

the right one, for trees' roots, records *k*'s values giving the minimal value

i \ *j*	0	1	2	3	4
1	0	.1	.4	1.1	1.7
2		0	.2	.8	1.4
3			0	.4	1.0
4				0	.3
5					0

i \ *j*	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					



optimal BST

Optimal Binary Search Trees

ALGORITHM *OptimalBST*$(P[1..n])$

```
//Finds an optimal binary search tree by dynamic programming
//Input: An array P[1..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
//        optimal BST and table R of subtrees' roots in the optimal BST
for i ← 1 to n do
    C[i, i − 1] ← 0
    C[i, i] ← P[i]
    R[i, i] ← i
C[n + 1, n] ← 0
for d ← 1 to n − 1 do //diagonal count
    for i ← 1 to n − d do
        j ← i + d
        minval ← ∞
        for k ← i to j do
            if C[i, k − 1] + C[k + 1, j] < minval
                minval ← C[i, k − 1] + C[k + 1, j]; kmin ← k
        R[i, j] ← kmin
        sum ← P[i]; for s ← i + 1 to j do sum ← sum + P[s]
        C[i, j] ← minval + sum
return C[1, n], R
```

Analysis DP for Optimal BST Problem

Time efficiency: $\Theta(n^3)$ but can be reduced to $\Theta(n^2)$ by taking advantage of monotonicity of entries in the root table, i.e., $R[i,j]$ is always in the range between $R[i,j-1]$ and $R[i+1,j]$

Space efficiency: $\Theta(n^2)$

Method can be expanded to include unsuccessful searches

Knapsack Problem by DP

Given n items of

integer weights: $w_1 \quad w_2 \quad \dots \quad w_n$ *(positive)*

values: $v_1 \quad v_2 \quad \dots \quad v_n$

a knapsack of integer capacity W

find most valuable subset of the items that fit into the knapsack

Knapsack Problem by DP

Given n items of

integer weights: $w_1 \quad w_2 \quad \dots \quad w_n$ *(positive)*

values: $v_1 \quad v_2 \quad \dots \quad v_n$

a knapsack of integer capacity W

find most valuable subset of the items that fit into the knapsack

Consider instance defined by first i items and capacity j ($j \le W$).

Let $V[i, j]$ be optimal value of such instance. Then....

Knapsack Problem by DP

Given n items of

integer weights: $w_1 \quad w_2 \quad \dots \quad w_n$ *(positive)*

values: $v_1 \quad v_2 \quad \dots \quad v_n$

a knapsack of integer capacity W

find most valuable subset of the items that fit into the knapsack

Consider instance defined by first i items and capacity j ($j \le W$).

Let $V[i, j]$ be optimal value of such instance. Then

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1,j-w_i]\} & \text{if } j - w_i \ge 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

Initial conditions: $V[0,j] = 0$ and $V[i,0] = 0$

Completar linha-a-linha ou coluna-por-coluna

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1,j-w_i]\} & \text{if } j-w_i \geq 0 \\ V[i-1,j] & \text{if } j-w_i < 0 \end{cases}$$

$V[0,j] = 0$ and $V[i,0] = 0$

Para computar a entrada de V[i,j], computamos o máximo entre
- *linha anterior/mesma coluna*
- *soma de v_i e a entrada na linha anterior e w_i colunas à esquerda*

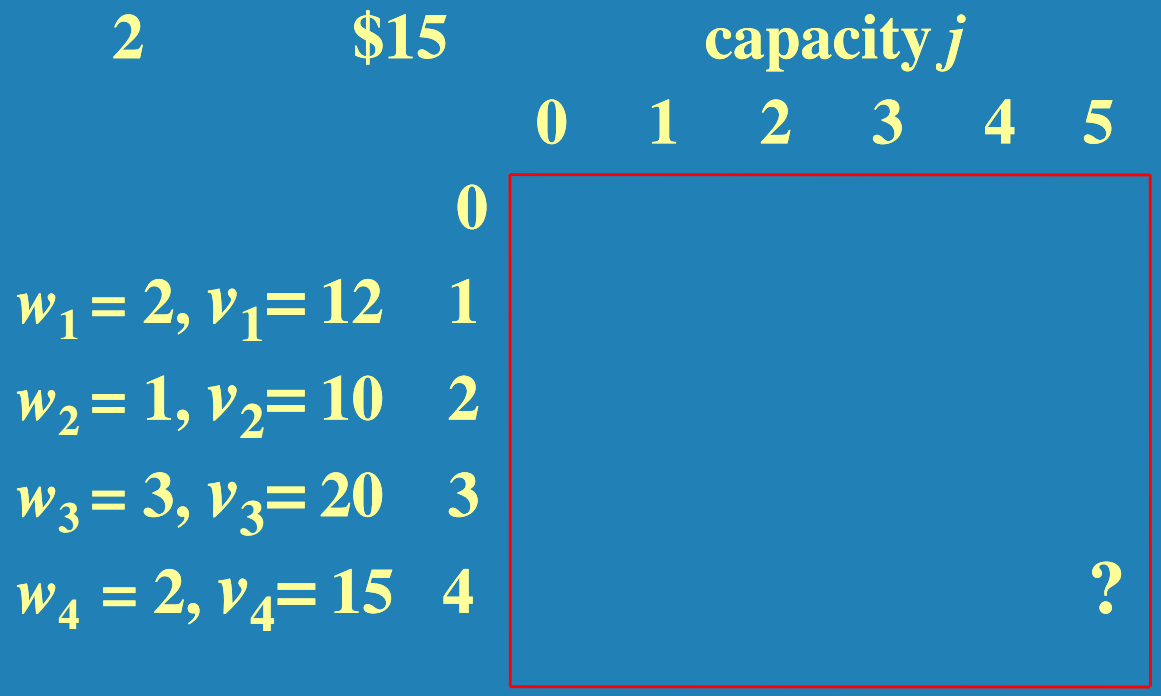
	0		$j-w_i$			j			W
0	0		0			0			0
i-1	0		$V[i-1, j-w_i]$			$V[i-1, j-w_i]$			
i	0					V[i,j]			
n	0								Goal

Knapsack Problem by DP (example)

Example: Knapsack of capacity $W = 5$

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

Para computar a entrada de V[i,j], computamos o máximo entre
- *linha anterior/mesma coluna*
- *soma de v_i e a entrada na linha anterior e w_i colunas à esquerda*



Knapsack Problem by DP (example)

Knapsack of capacity $W = 5$

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

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1,j-w_i]\} & \text{if } j-w_i \geq 0 \\ V[i-1,j] & \text{if } j-w_i < 0 \end{cases}$$

$V[0,j] = 0$ and $V[i,0] = 0$

- i=1; j- w_i : j=1; j-2=-1 < 0, V[1,1] = 0
- i=1; j- w_i : j=2.. 5, j-2=0..3, V[0,j]= max { 0; 12 + V[0,j-2] } = 12
- i=1; j=2; j-2=0; V[1,2]= max { V[1,2]; 10 + V[0, 2-2=0] } = 10
- i=2; j=2; j-2=0; V[2,2]= max { V[1,2]; 12 + V[1, 2-1=1] } = 12
- i=2; j=3; j-2=0; V[2,3]= max { V[1,3]; 10 + V[1, 3-1=2] } = 22

		0	1	2	3	4	5	capacity j
	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	

Memory functions

- Considerando problemas que satisfazem relação de recorrência top-down com sub-problemas sobrepostos, a solução (abordagem) direta top-down leva a soluções muito ineficientes (exponenciais ou pior) (e.g. Fib(n))

- A abordagem PD clássica trabalha bottom-up, preenchendo uma tabela com as soluções para *todos* os sub-problemas , mas resolvendo cada um deles apenas uma vez

- O problema com a abordagem PD clássica é que a solução de alguns dos subproblemas nem sempre é necessária, o que não acontece com abordagem top-down direta.. é natural tentar combinar as duas abordagens

Memory functions

- O objetivo é resolver apenas os sub-problemas necessários E resolvê-los apenas uma vez… memory functions , memoization)
- Método resolve usando abordagem top-down (em vez de chamadas linha-a-linha ou coluna-a-coluna para todos os ementos) mas mantém adicionalmente uma tabela do tipo utilizado na PD clássica bottom-up
- Inicialmente a tabela é inicializada com valor nulo para indicar que nada foi calculado. A partir daí, sempre que um valor precisar ser calculado o método checa se o valor é diferente de nulo (foi calculado antes); recupera valor se ele é valido (!nulo) ou calcula e guarda se for nulo

A memory function for the knapsack problem

```
Algorithm MFKnapsack (i,j)
// implements the memory function 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 arrays Weights[1..n], Values[1..n] and
//         table V[0..n, 0..W]  whose entries are initialized all with -1 except for
//         row 0 and column 0 which are initialized with 0

if V[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]))
  V[i,j] ← value
return V[i,j]
```

Knapsack Problem by DP (example)

```
if V[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]))
  V[i,j] ← value
return V[i,j]
```

0	1	2	3	4	5
0	0	0	0	0	0
0	0	12	12	12	12
0	10	12	22	22	22
0	10	12	22	30	32
0	10	15	25	30	37

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	---	12	22	---	22
$w_3 = 3, v_3 = 20$	3	0	---	---	22	---	32
$w_4 = 2, v_4 = 15$	4	0	---	---	---	---	37

- http://www.allisons.org/ll/AlgDS/Dynamic/Edit/
- http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/05-dynprog.pdf