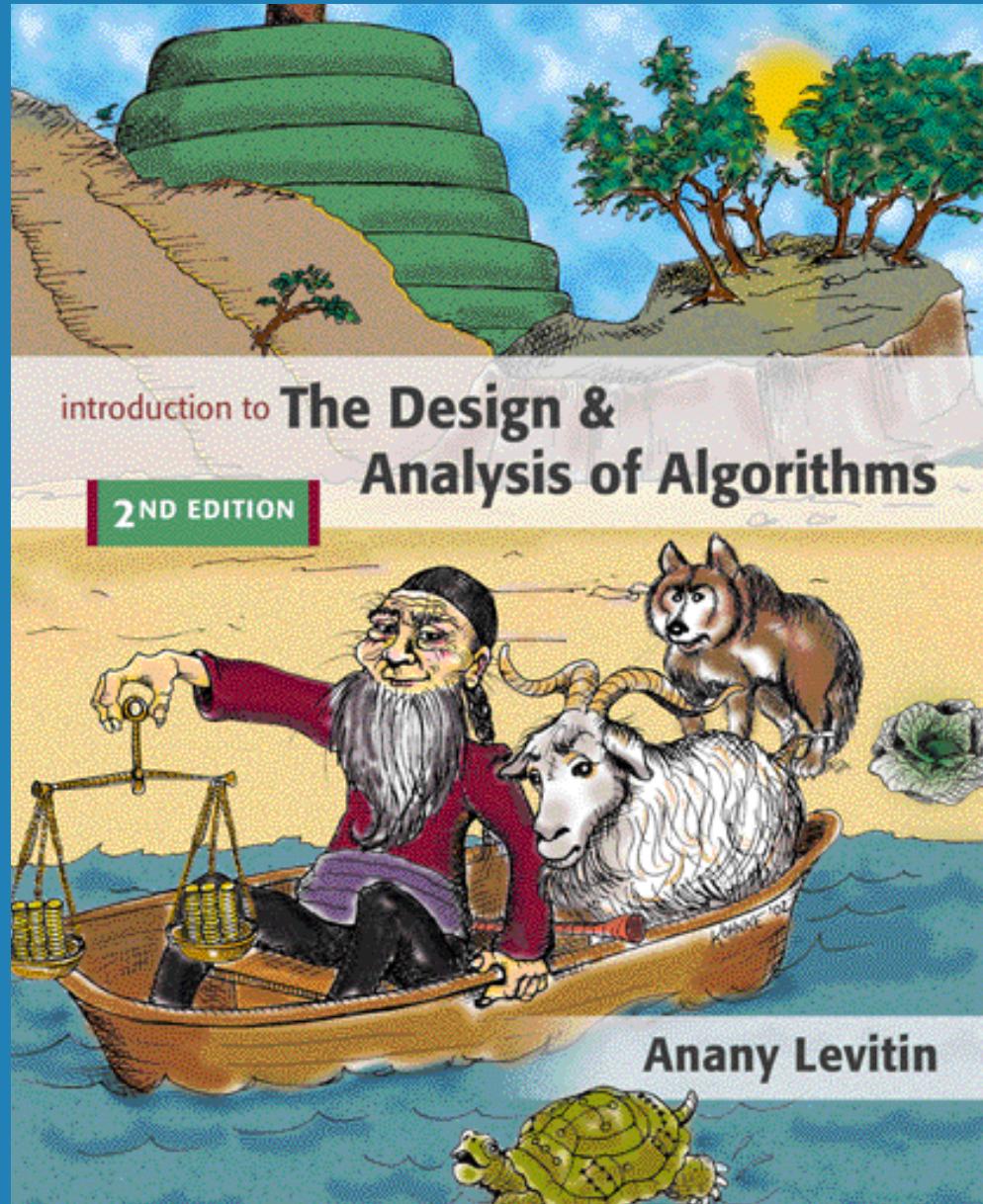


# Chapter 8

## Dynamic Programming



# Dynamic Programming



***Dynamic Programming*** is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping subinstances

- 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



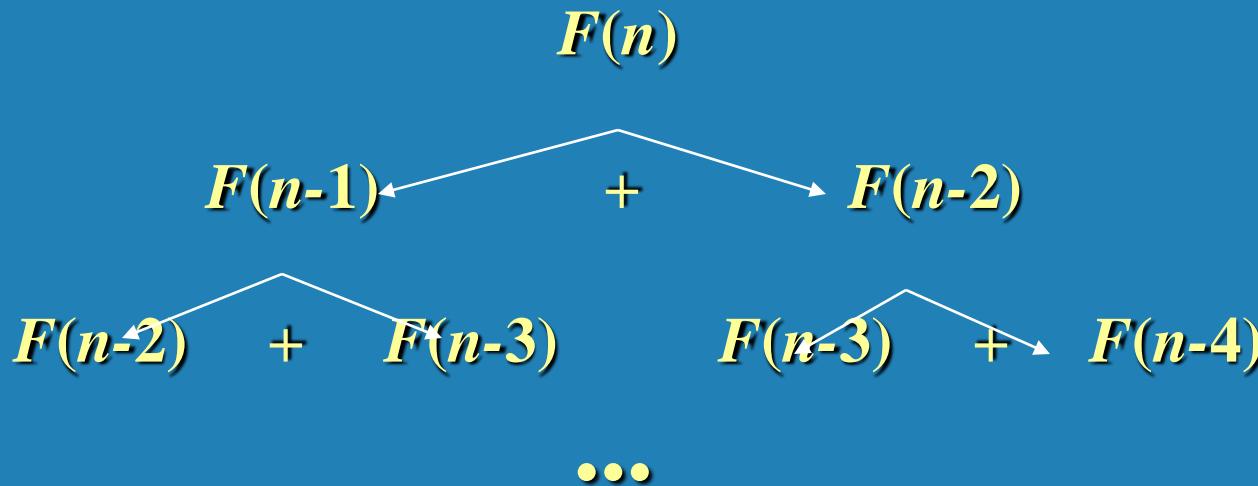
- Recall definition of Fibonacci numbers:

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

$$F(0) = 0$$

$$F(1) = 1$$

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



# Example: Fibonacci numbers (cont.)



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

$$F(0) = 0$$

$$F(1) = 1$$

$$F(2) = 1+0 = 1$$

...

$$F(n-2) =$$

$$F(n-1) =$$

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

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

Efficiency:

- time
- space

n  
n

What if we solve  
it recursively?

# Examples of DP algorithms



- Computing a binomial coefficient
- Longest common subsequence
- 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:
  - traveling salesman
  - knapsack



# Computing a binomial coefficient by DP



Binomial coefficients are coefficients of the binomial formula:

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

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

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

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

	0	1	2	...	$k-1$	$k$
0	1					
1	1	1				
.						
.						
.						
$n-1$			$C(n-1,k-1)$	$C(n-1,k)$		
$n$				$C(n,k)$		

# 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  $n \geq k \geq 0$ 
//Output: The value of  $C(n, k)$ 
for  $i \leftarrow 0$  to  $n$  do
    for  $j \leftarrow 0$  to  $\min(i, k)$  do
        if  $j = 0$  or  $j = i$ 
             $C[i, j] \leftarrow 1$ 
        else  $C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$ 
return  $C[n, k]$ 
```

**Time efficiency:**  $\Theta(nk)$

**Space efficiency:**  $\Theta(nk)$

# Knapsack Problem by DP



Given  $n$  items of

integer weights:  $w_1 \ w_2 \ \dots \ w_n$

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

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  $j$  ( $j \leq W$ ).

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

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

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

# 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

capacity  $j$

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

$w_1 = 2, v_1 = 12$     1

$w_2 = 1, v_2 = 10$     2

$w_3 = 3, v_3 = 20$     3

$w_4 = 2, v_4 = 15$     4

Backtracing finds the actual optimal subset, i.e. solution.

# Knapsack Problem by DP (pseudocode)



**Algorithm DPKnapsack( $w[1..n]$ ,  $v[1..n]$ ,  $W$ )**

**var**  $V[0..n, 0..W]$ ,  $P[1..n, 1..W]$ : int

**for**  $j := 0$  to  $W$  **do**

$V[0,j] := 0$

**for**  $i := 0$  to  $n$  **do**

$V[i,0] := 0$

**for**  $i := 1$  to  $n$  **do**

**for**  $j := 1$  to  $W$  **do**

**if**  $w[i] \leq j$  and  $v[i] + V[i-1, j-w[i]] > V[i-1, j]$  **then**

$V[i,j] := v[i] + V[i-1, j-w[i]]$ ;  $P[i,j] := j-w[i]$

**else**

$V[i,j] := V[i-1, j]$ ;  $P[i,j] := j$

**return**  $V[n, W]$  and the optimal subset by backtracing

Running time and space:

$O(nW)$ .

# Longest Common Subsequence (LCS)



- A subsequence of a sequence/string  $S$  is obtained by deleting zero or more symbols from  $S$ . For example, the following are **some** subsequences of “president”: pred, sdn, pre~~d~~nt. In other words, the letters of a subsequence of  $S$  appear in order in  $S$ , but they are not required to be consecutive.
- The longest common subsequence problem is to find a maximum length common subsequence between two sequences.

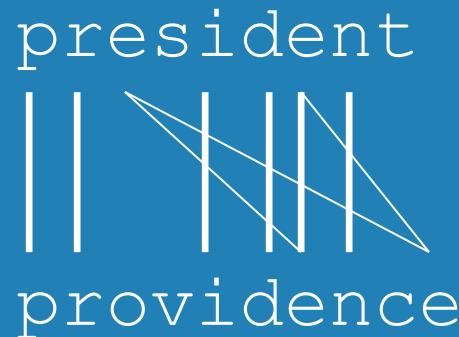


For instance,

**Sequence 1: president**

**Sequence 2: providence**

**Its LCS is priden.**



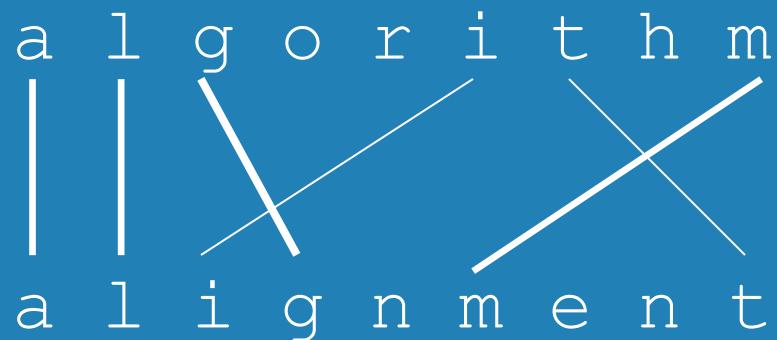


**Another example:**

**Sequence 1: algorithm**

**Sequence 2: alignment**

**One of its LCS is algm.**



# How to compute LCS?



- Let  $A=a_1a_2\dots a_m$  and  $B=b_1b_2\dots b_n$ .
- $\text{len}(i, j)$ : the length of an LCS between  $a_1a_2\dots a_i$  and  $b_1b_2\dots b_j$
- With proper initializations,  $\text{len}(i, j)$  can be computed as follows.

$$\text{len}(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \text{len}(i - 1, j - 1) + 1 & \text{if } i, j > 0 \text{ and } a_i = b_j, \\ \max(\text{len}(i, j - 1), \text{len}(i - 1, j)) & \text{if } i, j > 0 \text{ and } a_i \neq b_j. \end{cases}$$



**procedure** *LCS-Length*(*A*, *B*)

1. **for**  $i \leftarrow 0$  **to**  $m$  **do**  $len(i, 0) = 0$
2. **for**  $j \leftarrow 1$  **to**  $n$  **do**  $len(0, j) = 0$
3. **for**  $i \leftarrow 1$  **to**  $m$  **do**
4.     **for**  $j \leftarrow 1$  **to**  $n$  **do**
5.         **if**  $a_i = b_j$  **then**  $\begin{cases} len(i, j) = len(i - 1, j - 1) + 1 \\ prev(i, j) = " \blacktriangleright " \end{cases}$
6.         **else if**  $len(i - 1, j) \geq len(i, j - 1)$
7.             **then**  $\begin{cases} len(i, j) = len(i - 1, j) \\ prev(i, j) = " \uparrow " \end{cases}$
8.             **else**  $\begin{cases} len(i, j) = len(i, j - 1) \\ prev(i, j) = " \leftarrow " \end{cases}$
9.     **return**  $len$  and  $prev$



i	j	0	1	2	3	4	5	6	7	8	9	10
		p	r	o	v	i	d	e	n	c		e
0		0	0	0	0	0	0	0	0	0	0	0
1	p	0	1	1	1	1	1	1	1	1	1	1
2	r	0	1	2	2	2	2	2	2	2	2	2
3	e	0	1	2	2	2	2	2	3	3	3	3
4	s	0	1	2	2	2	2	2	3	3	3	3
5	i	0	1	2	2	2	3	3	3	3	3	3
6	d	0	1	2	2	2	3	4	4	4	4	4
7	e	0	1	2	2	2	3	4	5	5	5	5
8	n	0	1	2	2	2	3	4	5	6	6	6
9	t	0	1	2	2	2	3	4	5	6	6	6

Running time and memory:  $O(mn)$  and  $O(mn)$ .

# The backtracing algorithm



```
procedure Output-LCS( $A$ ,  $prev$ ,  $i$ ,  $j$ )
1 if  $i = 0$  or  $j = 0$  then return
2 if  $prev(i, j) = \text{''}\leftarrow\text{''}$  then  $\begin{cases} Output - LCS(A, prev, i - 1, j - 1) \\ \text{print } a_i \end{cases}$ 
3 else if  $prev(i, j) = \text{''}\uparrow\text{''}$  then  $Output - LCS(A, prev, i - 1, j)$ 
4 else  $Output - LCS(A, prev, i, j - 1)$ 
```



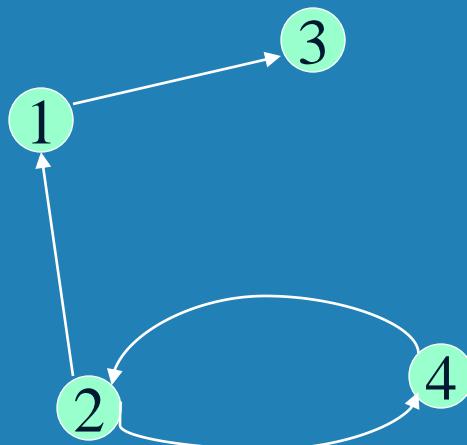
i	j	0	1	2	3	4	5	6	7	8	9	10
		p	r	o	v	i	d	e	n	c		e
0		0	0	0	0	0	0	0	0	0	0	0
1	p	0	1	1	1	1	1	1	1	1	1	1
2	r	0	1	2	2	2	2	2	2	2	2	2
3	e	0	1	2	2	2	2	2	3	3	3	3
4	s	0	1	2	2	2	2	2	3	3	3	3
5	i	0	1	2	2	2	3	3	3	3	3	3
6	d	0	1	2	2	2	3	4	4	4	4	4
7	e	0	1	2	2	2	3	4	5	5	5	5
8	n	0	1	2	2	2	3	4	5	6	6	6
9	t	0	1	2	2	2	3	4	5	6	6	6

Output: *priden*

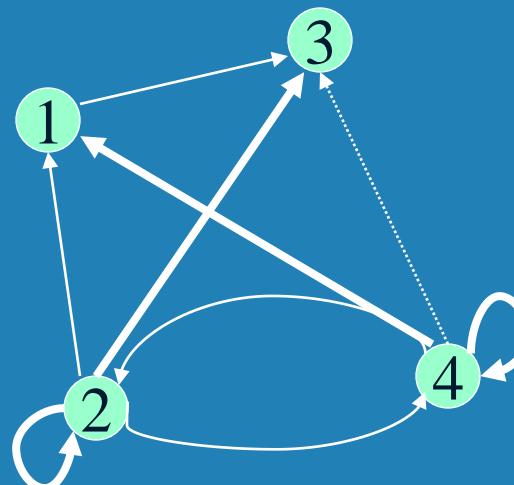
# Warshall's Algorithm: Transitive Closure



- 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

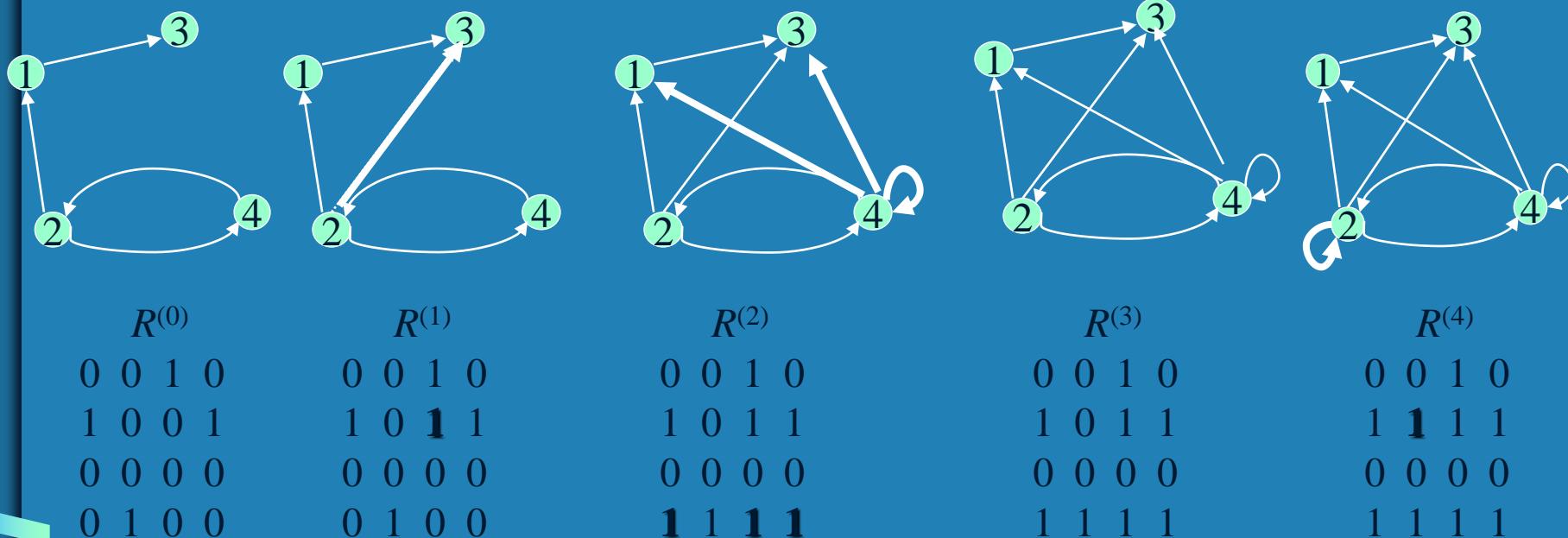


0	0	1	0
1	1	1	1
0	0	0	0
1	1	1	1

# Warshall's Algorithm



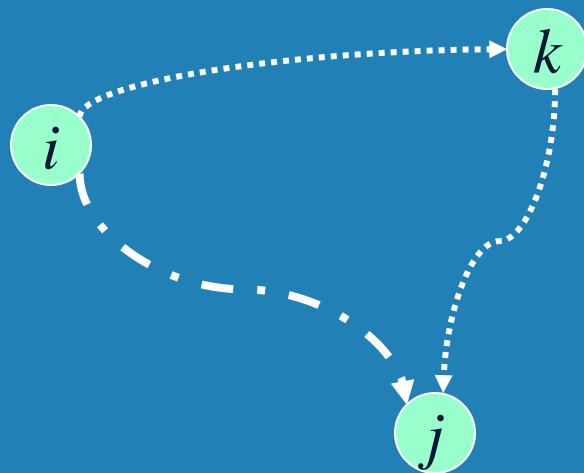
Constructs transitive closure  $T$  as the last matrix in the sequence of  $n$ -by- $n$  matrices  $R^{(0)}, \dots, R^{(k)}, \dots, R^{(n)}$  where  $R^{(k)}[i,j] = 1$  iff there is nontrivial path from  $i$  to  $j$  with only the first  $k$  vertices allowed as intermediate  
Note that  $R^{(0)} =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



# 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  $1, \dots, k$  allowed as intermediate

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



Initial condition?

# Warshall's Algorithm (matrix generation)

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

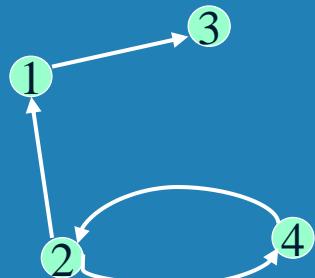
$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

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)



$$R^{(0)} =$$

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

$$R^{(2)} = \begin{matrix} & & 1 & 0 \\ & 1 & 0 & 1 & 1 \\ & \boxed{0} & 0 & \boxed{0} & 0 \\ 1 & 1 & \boxed{1} & 1 \end{matrix}$$

$$R^{(3)} =$$

$$\begin{matrix} & & 1 & 0 \\ & 1 & 0 & 1 & 1 \\ & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{matrix}$$

$$R^{(4)} = \begin{matrix} & & 1 & 0 \\ & 1 & 1 & 1 & 1 \\ & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{matrix}$$

# 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

$R^{(0)} \leftarrow A$

**for**  $k \leftarrow 1$  **to**  $n$  **do**

**for**  $i \leftarrow 1$  **to**  $n$  **do**

**for**  $j \leftarrow 1$  **to**  $n$  **do**

~~if  $(i, j)$  is connected~~    $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$  **or** ( $R^{(k-1)}[i, k]$  **and**  $R^{(k-1)}[k, j]$ )

**return**  $R^{(n)}$

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

**Space efficiency:** Matrices can be written over their predecessors  
(with some care), so it's  $\Theta(n^2)$ .

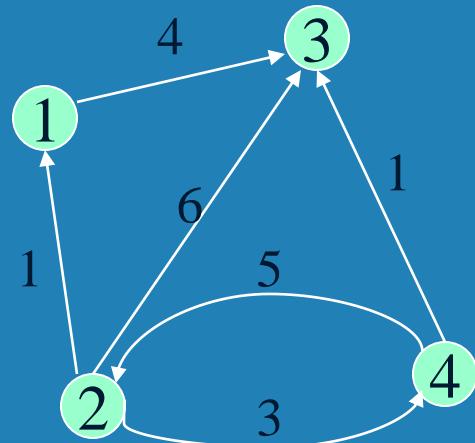
# Floyd's Algorithm: All pairs shortest paths



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

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

**Example:**

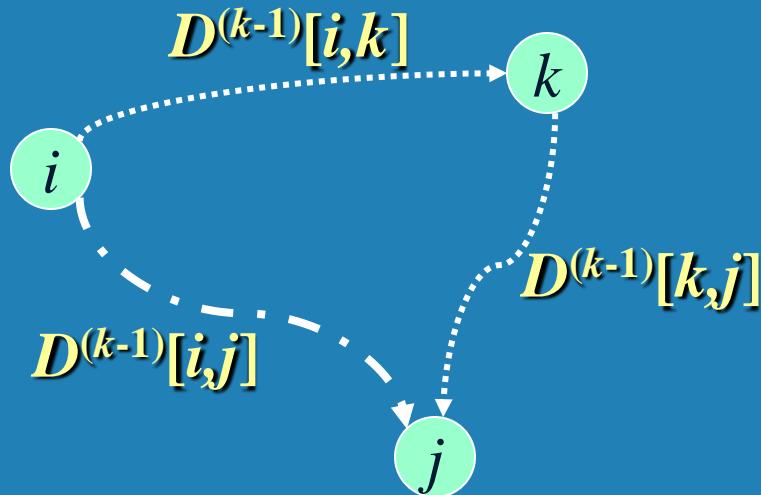


0	$\infty$	4	$\infty$
1	0	4	3
$\infty$	$\infty$	0	$\infty$
6	5	1	0

# Floyd's Algorithm (matrix generation)

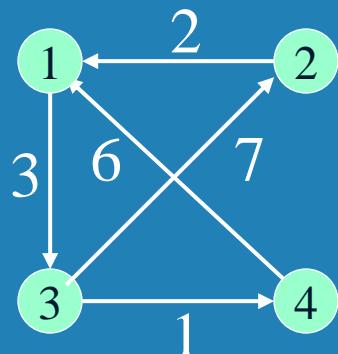
On the  $k$ -th iteration, the algorithm determines shortest paths between every pair of vertices  $i, j$  that use only vertices among  $1, \dots, k$  as intermediate

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



Initial condition?

# Floyd's Algorithm (example)



$$D^{(0)} =$$

0	$\infty$	3	$\infty$
2	0	$\infty$	$\infty$
$\infty$	7	0	1
6	$\infty$	$\infty$	0

$$D^{(1)} =$$

0	$\infty$	3	$\infty$
2	0	<b>5</b>	$\infty$
$\infty$	7	0	1
6	$\infty$	<b>9</b>	0

$$D^{(2)} =$$

0	$\infty$	<b>3</b>	$\infty$
2	0	<b>5</b>	$\infty$
<b>9</b>	7	0	1
6	$\infty$	<b>9</b>	0

$$D^{(3)} =$$

0	<b>10</b>	3	<b>4</b>
2	0	5	<b>6</b>
9	7	0	1
6	<b>16</b>	9	0

$$D^{(4)} =$$

0	10	3	4
2	0	5	6
7	7	0	1
6	16	9	0

# 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  $k \leftarrow 1$  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  $D$       If  $D[i,k] + D[k,j] < D[i,j]$  then  $P[i,j] \leftarrow k$ 
```

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

Since the superscripts  $k$  or  $k-1$  make no difference to  $D[i,k]$  and  $D[k,j]$ .

**Space efficiency:** Matrices can be written over their predecessors

**Note:** Works on graphs with negative edges but without negative cycles.  
Shortest paths themselves can be found, too. How?

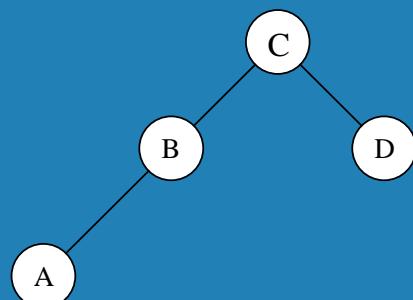
# Optimal Binary Search Trees



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

Since total number of BSTs with  $n$  nodes is 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?

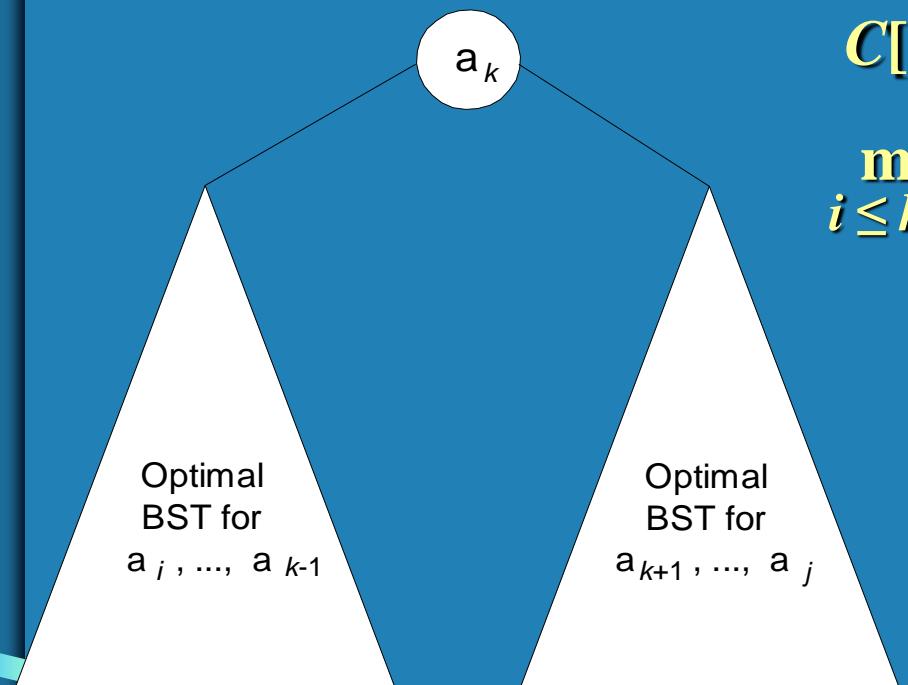


$$\begin{aligned}\text{Average \# of comparisons} \\ &= 1*0.4 + 2*(0.2+0.3) + 3*0.1 \\ &= 1.7\end{aligned}$$

# DP for Optimal BST Problem



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



$$C[i,j] =$$

$$\min_{i \leq k \leq j} \{ p_k \cdot 1 +$$

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

$$\sum_{s=k+1}^j p_s (\text{level } a_s \text{ in } T[k+1,j] + 1)\}$$

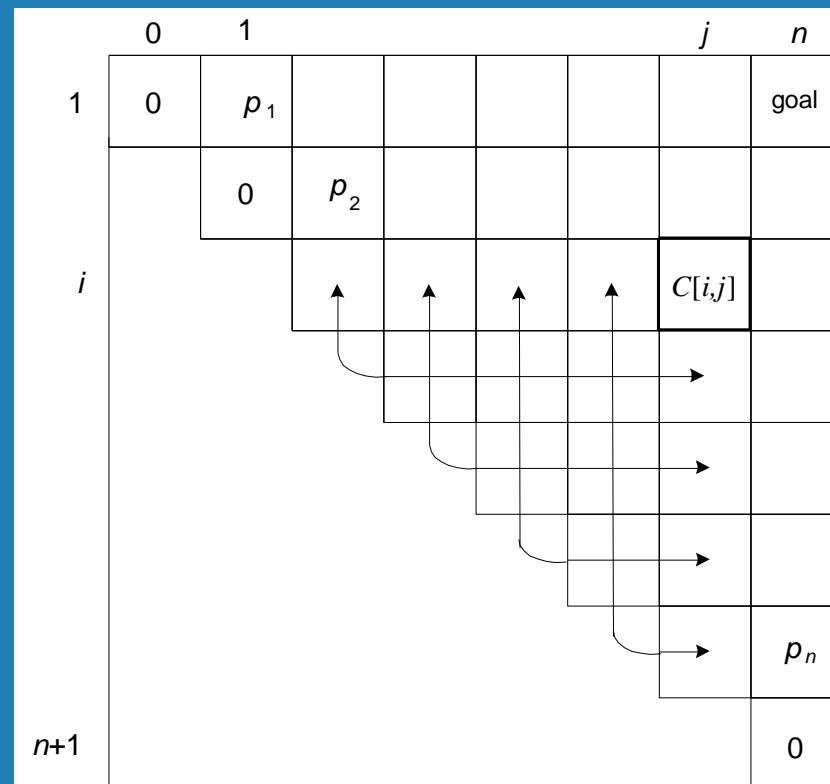
# DP for Optimal BST Problem (cont.)



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

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

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



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

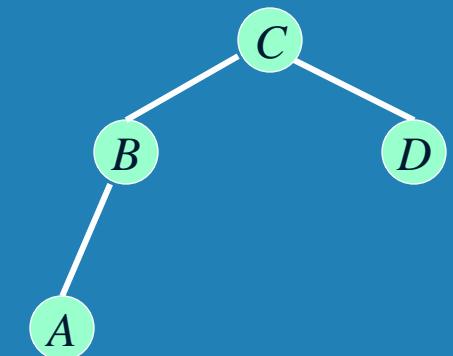
The tables below are filled diagonal by diagonal: the left one is filled using the recurrence

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

the right one, for trees' roots, records  $k$ 's values giving the minima

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

$i$	$j$	0	1	2	3	4
1	0			1	2	3
2	0			2	3	3
3	0				3	3
4	0					4
5	0					



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  $i \leftarrow 1$  to  $n$  do
     $C[i, i - 1] \leftarrow 0$ 
     $C[i, i] \leftarrow P[i]$ 
     $R[i, i] \leftarrow i$ 
 $C[n + 1, n] \leftarrow 0$ 
for  $d \leftarrow 1$  to  $n - 1$  do //diagonal count
    for  $i \leftarrow 1$  to  $n - d$  do
         $j \leftarrow i + d$ 
         $minval \leftarrow \infty$ 
        for  $k \leftarrow i$  to  $j$  do
            if  $C[i, k - 1] + C[k + 1, j] < minval$ 
                 $minval \leftarrow C[i, k - 1] + C[k + 1, j]; kmin \leftarrow k$ 
         $R[i, j] \leftarrow kmin$ 
         $sum \leftarrow P[i];$  for  $s \leftarrow i + 1$  to  $j$  do  $sum \leftarrow sum + P[s]$ 
         $C[i, j] \leftarrow minval + sum$ 
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**