Chapter 8: Dynamic Programming

Dynamic 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**

8.1 Example 0: 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):**

$$\begin{tikzcd}
 & & & F(n) \arrow[dll] \arrow[drr] & & & \\
 & F(n\text{-}1) \arrow[dl] \arrow[dr] & & + & & F(n\text{-}2) \arrow[dl] \arrow[dr] & \\
F(n\text{-}2) & + & F(n\text{-}3) & & F(n\text{-}3) & + & F(n\text{-}4)
\end{tikzcd}$$

•••

Example 0: Fibonacci numbers (cont.)

Computing the *n*th Fibonacci number using bottom-up iteration and recording results:

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

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

Example 1: Coin-row problem

There is a row of *n* coins whose values are some positive integers $c_1, c_2, \ldots, c_n$, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

E.g.: 5, 1, 2, 10, 6, 2. What is the best selection?

DP solution to the coin-row problem

Let F(n) be the maximum amount that can be picked up from the row of n coins. To derive a recurrence for F(n), we partition all the allowed coin selections into two groups:

those without last coin – the max amount is ?

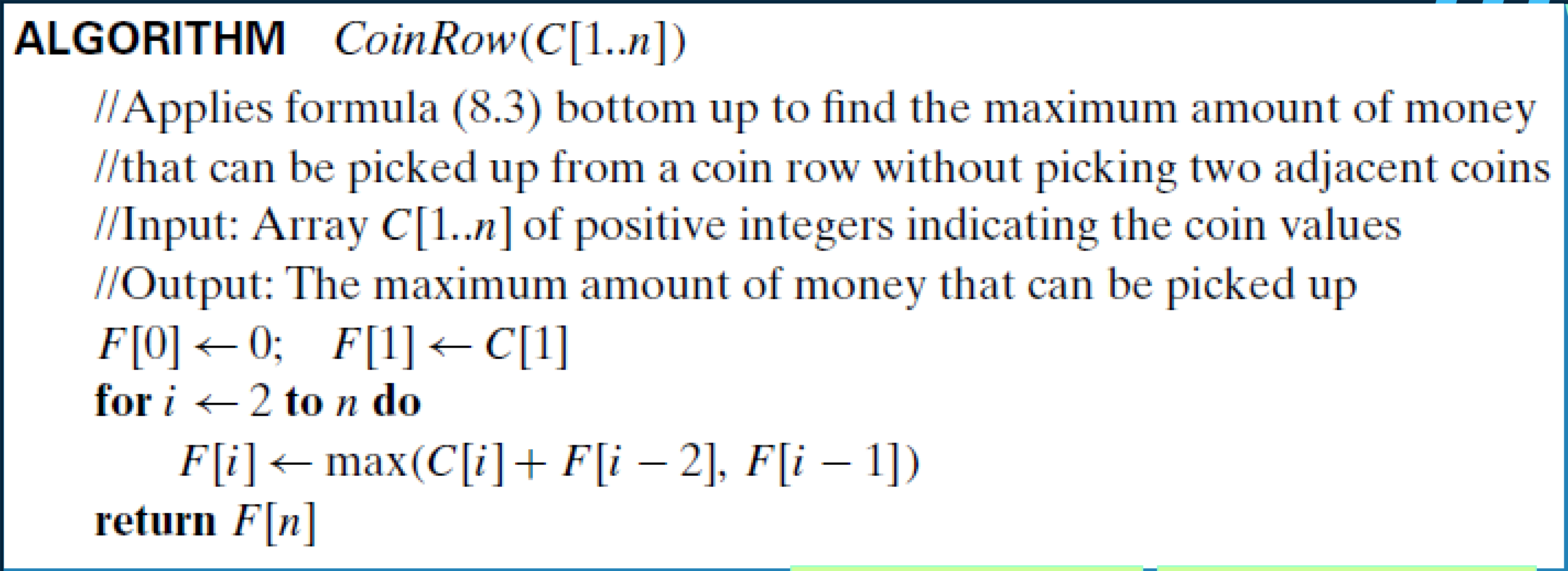
those with the last coin -- the max amount is ?

Thus we have the following recurrence

$$F(n) = \max\{c_n + F(n-2),\ F(n-1)\} \text{ for } n > 1,$$

$$F(0) = 0,\ F(1)=c_1$$

Pseudocode



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

Print coins of optimal solution

```
k=n;
while (k>0) {
   if (F(k)>F(k-1))
      print c[k]
      k = k-2
   else
      k = k-1
```

Example

$$F(n) = \max\{c_n + F(n-2),\ F(n-1)\} \text{ for } n > 1,$$

$$F(0) = 0,\ F(1)=c_1$$

index	0	1	2	3	4	5	6
coins	--	5	1	2	10	6	2
F( )							

Max amount:
Coins of optimal solution:

Time efficiency:
Space efficiency:

Note: All smaller instances were solved.

Example 2: Change-making problem

- Give change for amount *n* using the minimum number of coins of denominations $d_1<d_2 < \ldots <d_m$, where $d_1 = 1$
- Let $F(n)$ be the minimum number of coins whose values add up to *n* and define $F(0) = 0$.
- We have the following recurrence for $F(n)$:

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

- Example: amount $n = 6$ and coin denominations 1, 3, and 4.

n	0	1	2	3	4	5	6
F	0						

n	1	2	3	4	5	6
P						

Example 2: Pseudocode

```
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 $F[n]$
```

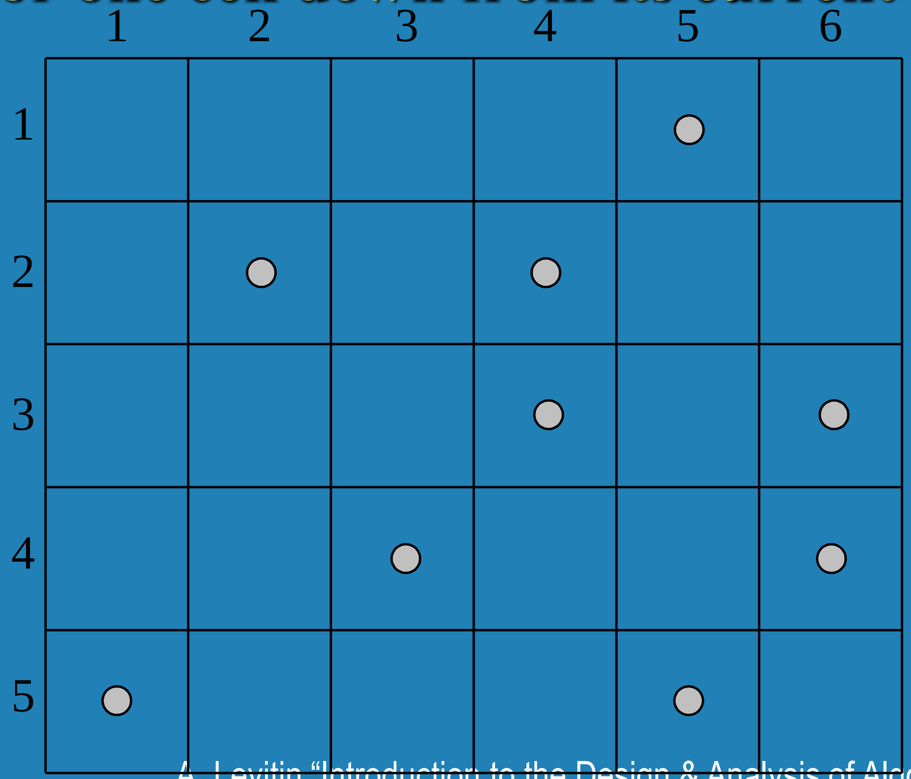
Print coins of optimal solution

```
k=n;
while (k>0) {
    print P[k]
    k = k – P[k]
}
```

```
if (F(i-D[j]) < temp)
    temp = F(i-D[j])
    P[i] = D[j]
j = j + 1
```

Example 3: Coin-collecting by robot

Several coins are placed in cells of an $n \times m$ board. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the right or one cell down from its current location.



Solution to the coin-collecting problem

Let F(i,j) be the largest number of coins the robot can collect and bring to cell (i,j) in the ith row and jth column.

The largest number of coins that can be brought to cell (i,j):

from the left neighbor ?

from the neighbor above?

The recurrence:

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

where $c_{ij} = 1$ if there is a coin in cell (i,j), and $c_{ij} = 0$ otherwise

$F(0, j) = 0$ for $1 \le j \le m$ and $F(i, 0) = 0$ for $1 \le i \le n$.

Speudocode

```
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 $F[n, m]$
```

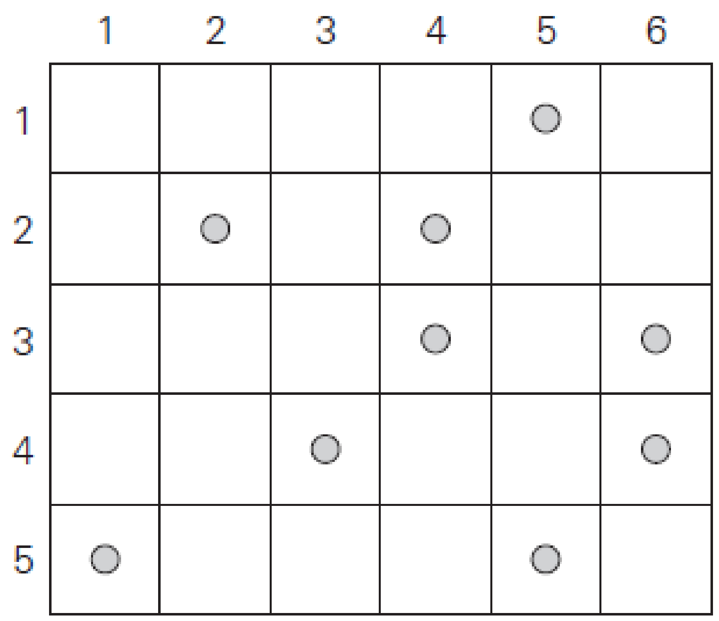
- **Time efficiency: Θ(mn). Space efficiency: Θ(mn)**

Dynamic programming algorithm results

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

where $c_{ij} = 1$ if there is a coin in cell (i,j), and $c_{ij} = 0$ otherwise

$F(0, j) = 0$ for $1 \le j \le m$ and $F(i, 0) = 0$ for $1 \le i \le n.$



	1	2	3	4	5	6
1					●	
2		●		●		
3				●		●
4			●			●
5	●				●	

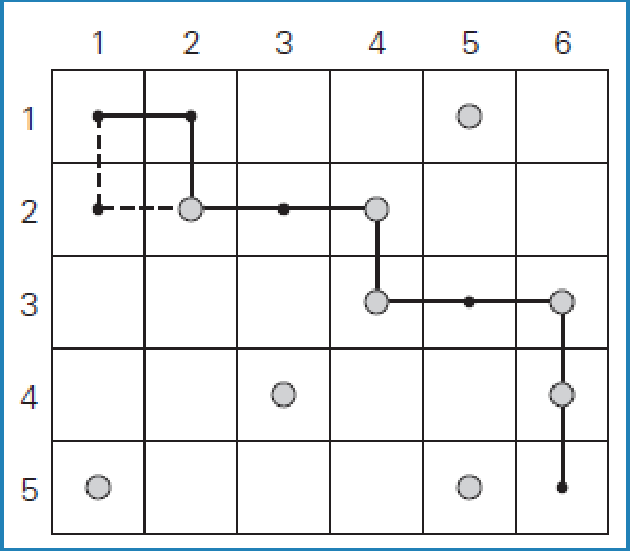
	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	**5**

Print the Optimal Path

- Pseudocode that prints the optimal path

```
i = n;  j = n
while ((i > 1) || (j > 1))
    if (F[i-1, j] > F[i, j-1]
        print '↑'
        i = i - 1
    else
        print '←'
        j = j - 1
```

- Example



Other examples of DP algorithms

- **Computing a binomial coefficient (# 9, Exercises 8.1)**
- **Some difficult discrete optimization problems:**
  - knapsack (Sec. 8.2)
  - traveling salesman
- **Constructing an optimal binary search tree (Sec. 8.3)**
- **Warshall's algorithm for transitive closure (Sec. 8.4)**
- **Floyd's algorithm for all-pairs shortest paths (Sec. 8.4)**

8.2 Knapsack Problem by DP

Given n items of

integer weights: $w_1 \quad w_2 \quad \ldots \quad w_n$

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

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 ($j \leq W$).

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

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

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

Knapsack Problem by DP (Bottom-up)

Example: Knapsack of capacity $W = 5$

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

Print optimal solution

```
i=n; j=W
while ((i>0) && (j>0)) {
   if(F(i, j) > F(i-1, j)
      print $v_i$
      $j = j - w_i$
   i = i – 1
}
```

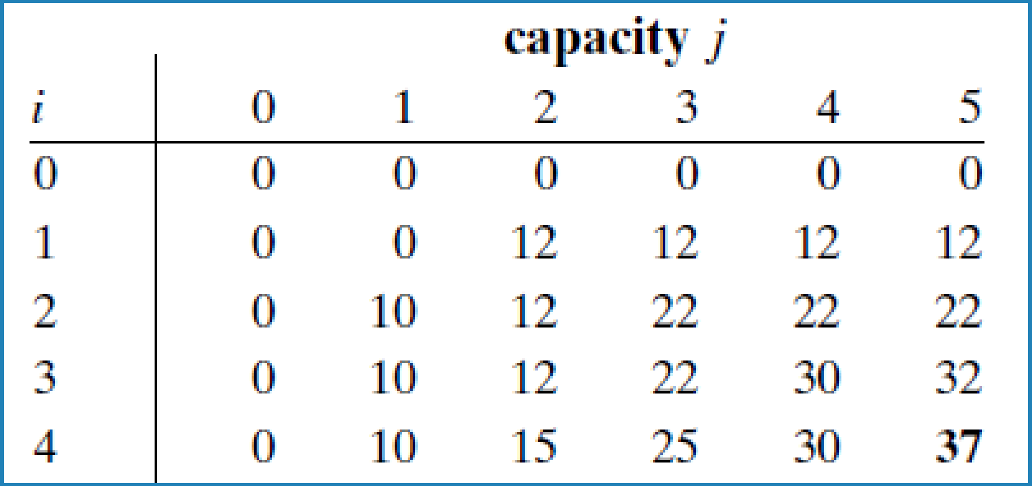
Bottom-up

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

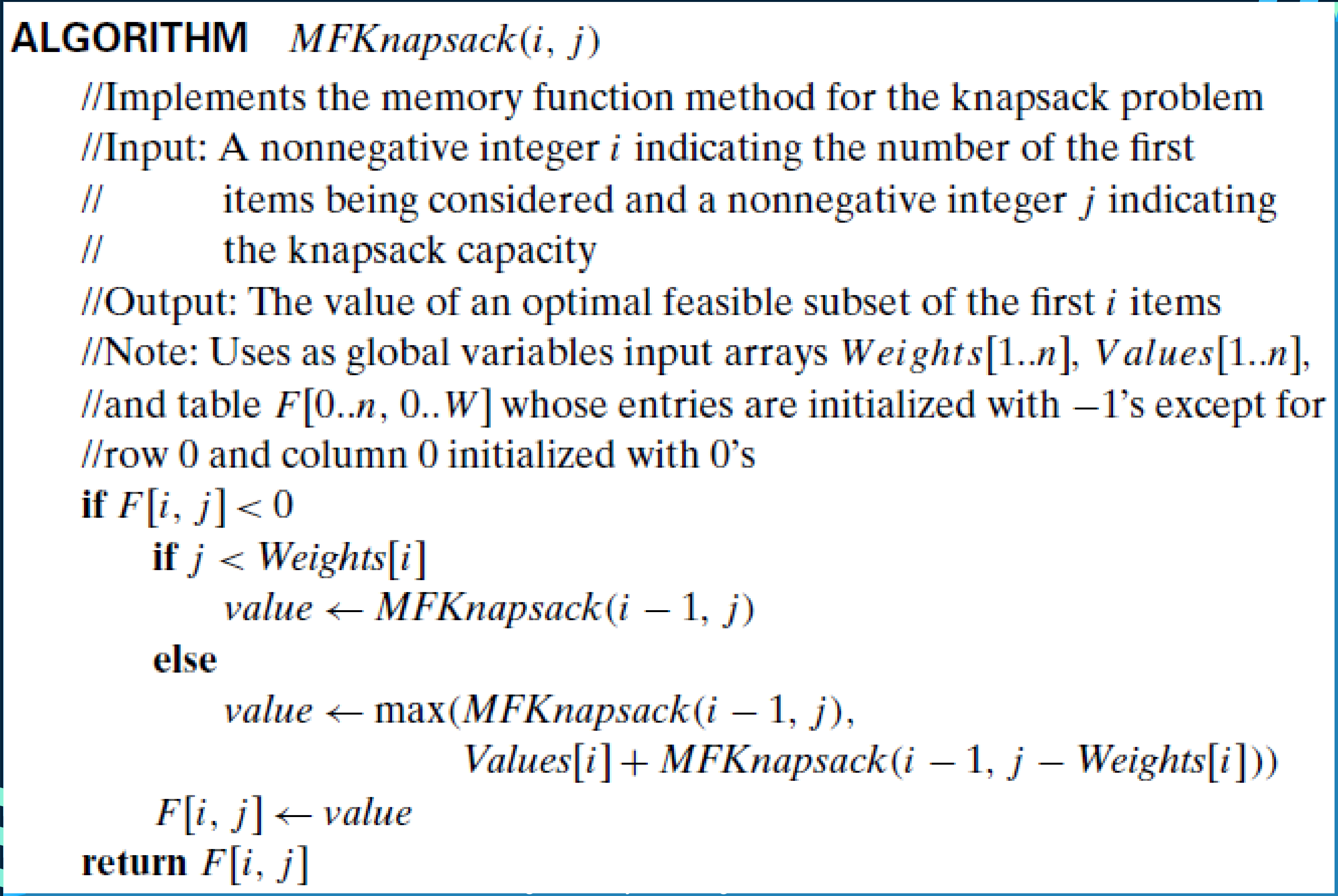
$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$



	capacity *j*					
i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	**37**

Knapsack Problem by DP (Top-down)

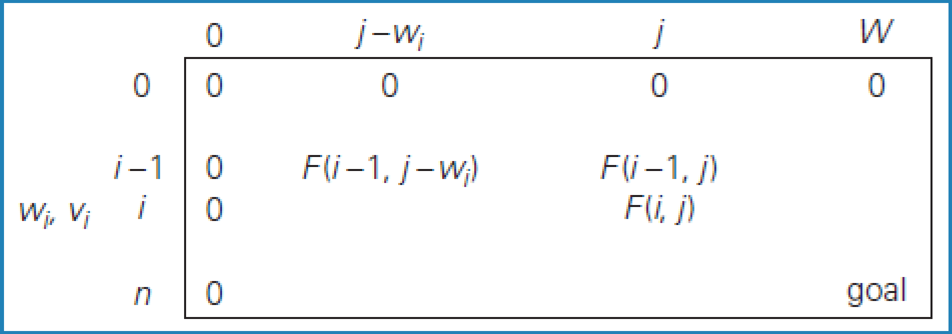


```
ALGORITHM  MFKnapsack(i, j)
    //Implements the memory function method for the knapsack problem
    //Input: A nonnegative integer i indicating the number of the first
    //        items being considered and a nonnegative integer j indicating
    //        the knapsack capacity
    //Output: The 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 $j < Weights[i]$
            $value \leftarrow MFKnapsack(i - 1, j)$
        else
            $value \leftarrow \max(MFKnapsack(i - 1, j),$
                        $Values[i] + MFKnapsack(i - 1, j - Weights[i]))$
        $F[i, j] \leftarrow value$
    return $F[i, j]$
```

Knapsack Problem by DP (Top-down)

Example: Knapsack of capacity *W* = 5

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$17



		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

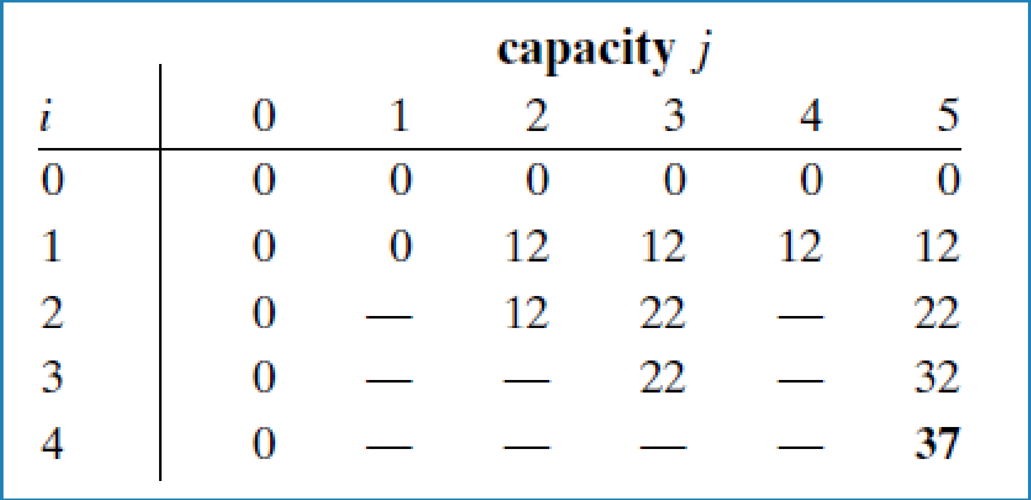
Top-down

$w_1 = 2, \; v_1 = 12$

$w_2 = 1, \; v_2 = 10$

$w_3 = 3, \; v_3 = 20$

$w_4 = 2, \; v_4 = 17$



	capacity j					
i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	—	12	22	—	22
3	0	—	—	22	—	32
4	0	—	—	—	—	**37**

8.3 Optimal Binary Search Trees

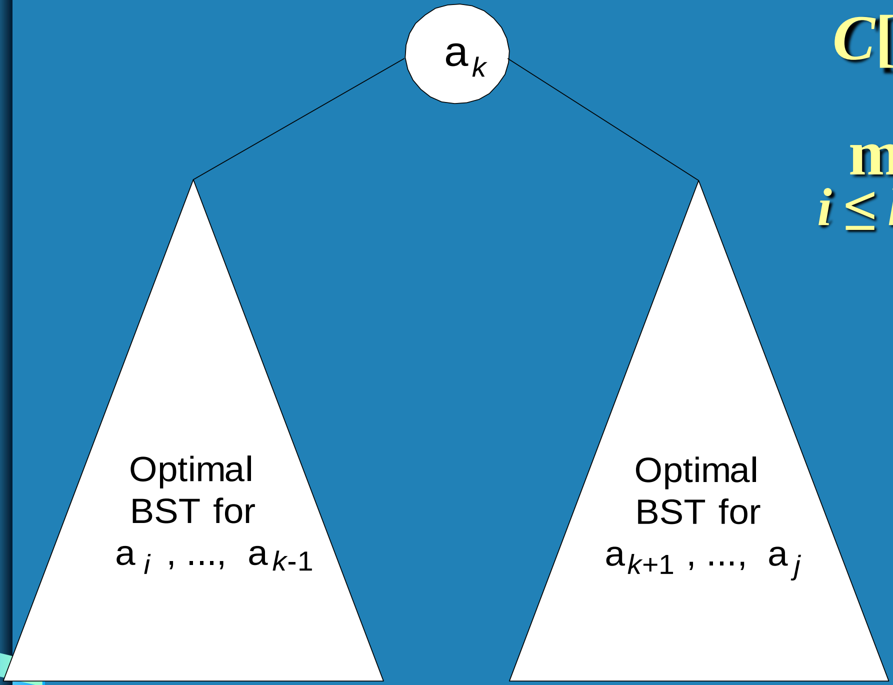
Problem: Given n keys $a_1 < \ldots < a_n$ and probabilities $p_1 \le \ldots \le p_n$ searching for them, find a BST with a minimum average number of comparisons in successful search.

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?

DP for Optimal BST Problem

Let $C[i,j]$ be minimum average number of comparisons made in T[i,j], optimal BST for keys $a_i < \ldots < a_j$, 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.



a_k
Optimal BST for $a_i, \ldots, a_{k-1}$
Optimal BST for $a_{k+1}, \ldots, a_j$

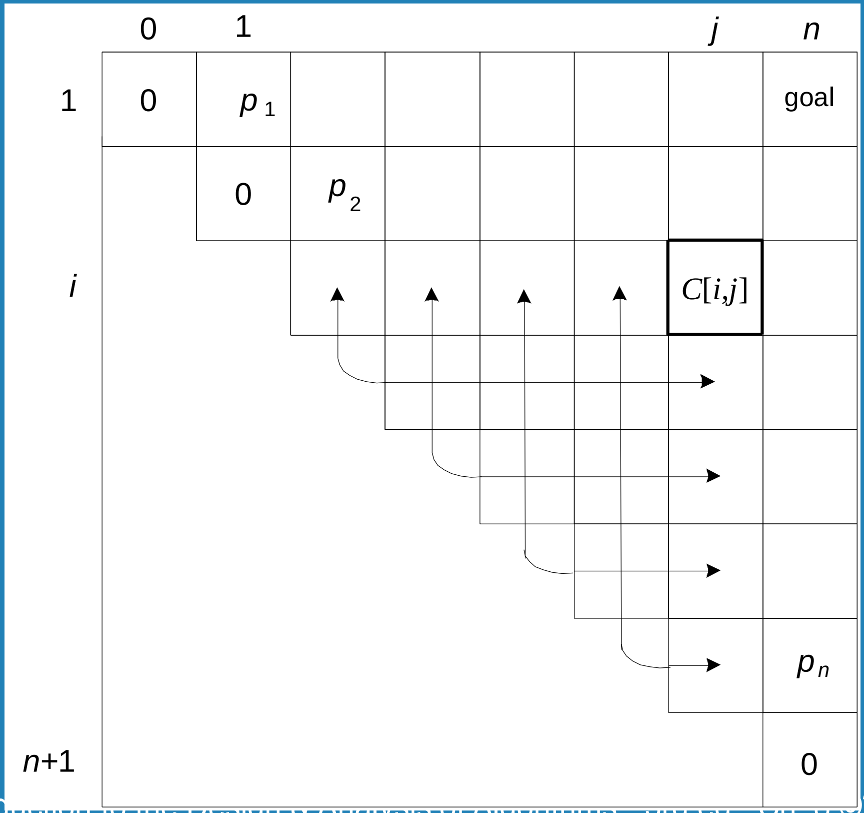
$$C[i,j] = \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \text{ (level } a_s \text{ in T}[i,k\text{-}1] + 1) + \sum_{s=k+1}^{j} p_s \text{ (level } a_s \text{ in T}[k+1,j] + 1)\}$$

DP for Optimal BST Problem (cont.)

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

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

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



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

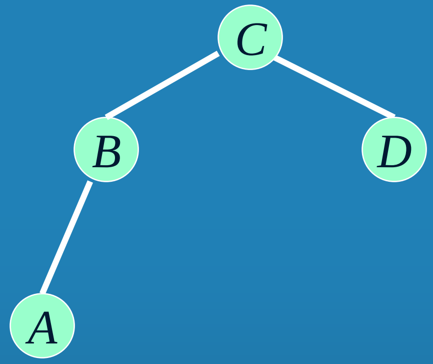
The tables below are filled diagonal by diagonal: the left one is filled using the recurrence

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

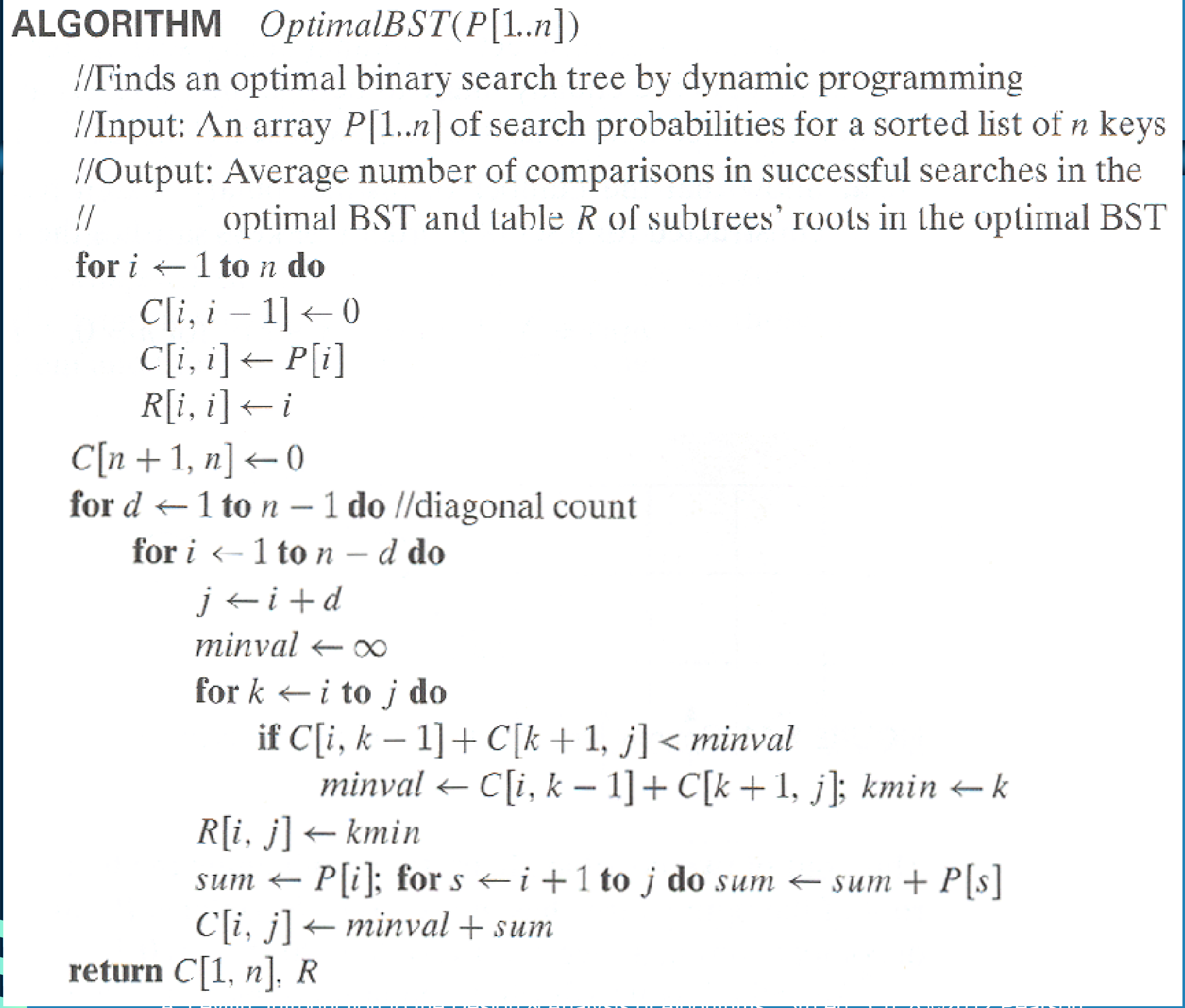
the right one, for trees' roots, records *k*'s values giving the minima

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



```
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: $\Theta(n^3)$

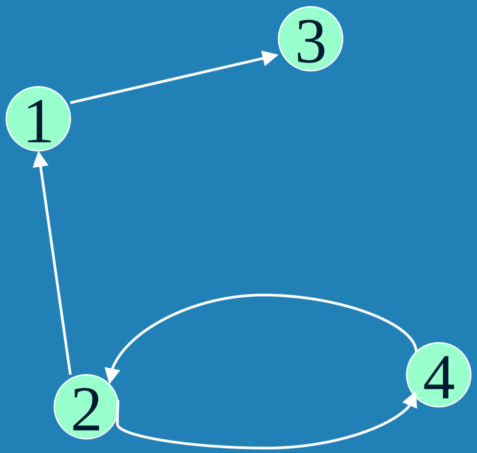
But can be reduced to $\Theta(n^2)$ by taking advantage of monotonicity of entries in the root table, i.e., entries in the root table are always nondecreasing along each row and column. This limits values for $R[i,j]$ is always in the range between $R[i,j-1]$ and $R[i+1,j]$

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

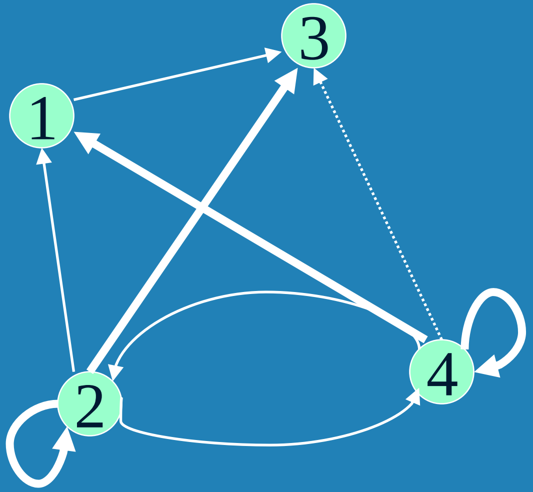
Method can be expended to include unsuccessful searches

8.4 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:



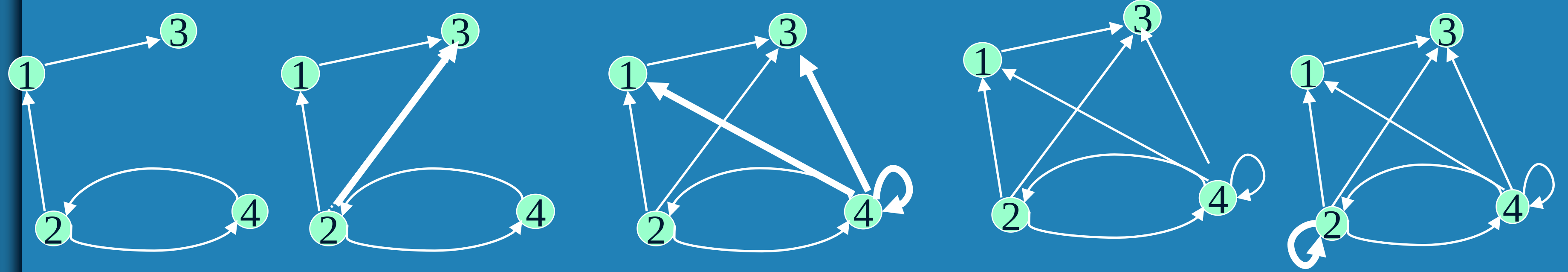
$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$



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

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 first k vertices allowed as intermediate
Note that $R^{(0)} = A$ (adjacency matrix), $R^{(n)} = T$ (transitive closure)

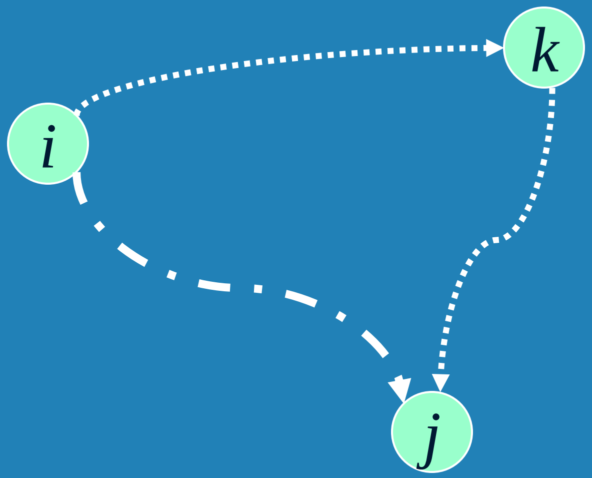


$$R^{(0)} \quad \begin{matrix} 0&0&1&0\\ 1&0&0&1\\ 0&0&0&0\\ 0&1&0&0 \end{matrix} \qquad R^{(1)} \quad \begin{matrix} 0&0&1&0\\ 1&0&\mathbf{1}&1\\ 0&0&0&0\\ 0&1&0&0 \end{matrix} \qquad R^{(2)} \quad \begin{matrix} 0&0&1&0\\ 1&0&1&1\\ 0&0&0&0\\ \mathbf{1}&1&\mathbf{1}&\mathbf{1} \end{matrix} \qquad R^{(3)} \quad \begin{matrix} 0&0&1&0\\ 1&0&1&1\\ 0&0&0&0\\ 1&1&1&1 \end{matrix} \qquad R^{(4)} \quad \begin{matrix} 0&0&1&0\\ 1&\mathbf{1}&1&1\\ 0&0&0&0\\ 1&1&1&1 \end{matrix}$$

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,…,*k* allowed as intermediate

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



Warshall's Algorithm (matrix generation)

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

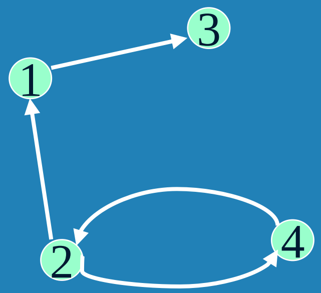
$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

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)



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

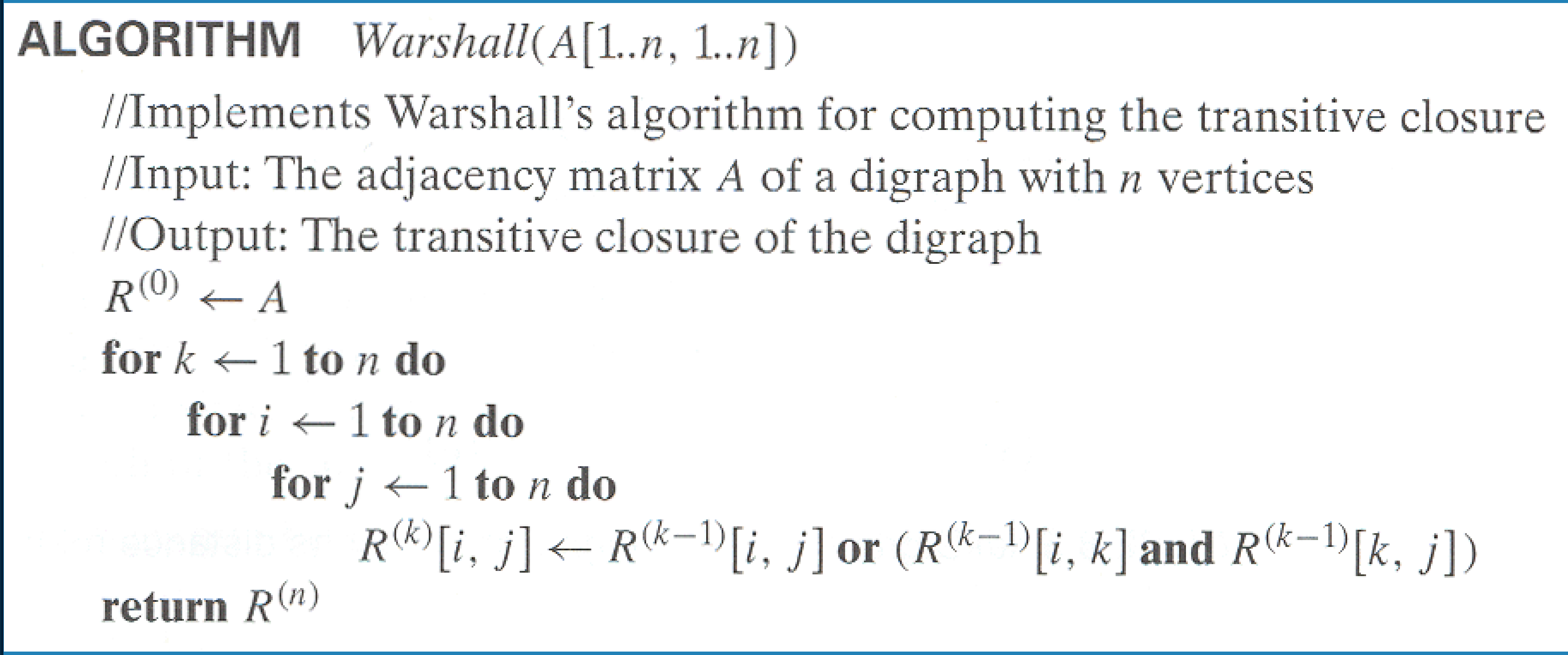
$$R^{(1)} = \begin{array}{cccc} 0 & 0 & 1 & 0 \\ 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}$$

$$R^{(2)} = \begin{array}{cccc} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{array}$$

$$R^{(3)} = \begin{array}{cccc} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{array}$$

$$R^{(4)} = \begin{array}{cccc} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{array}$$

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
                $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or $(R^{(k-1)}[i, k]$ and $R^{(k-1)}[k, j])$
    return $R^{(n)}$
```

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

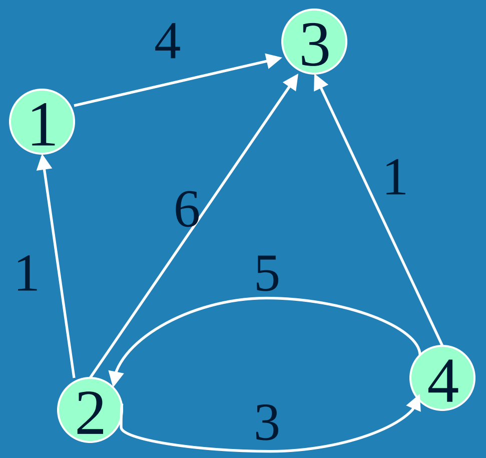
Space efficiency: Matrices can be written over their predecessors

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 $D^{(0)}$, …, $D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

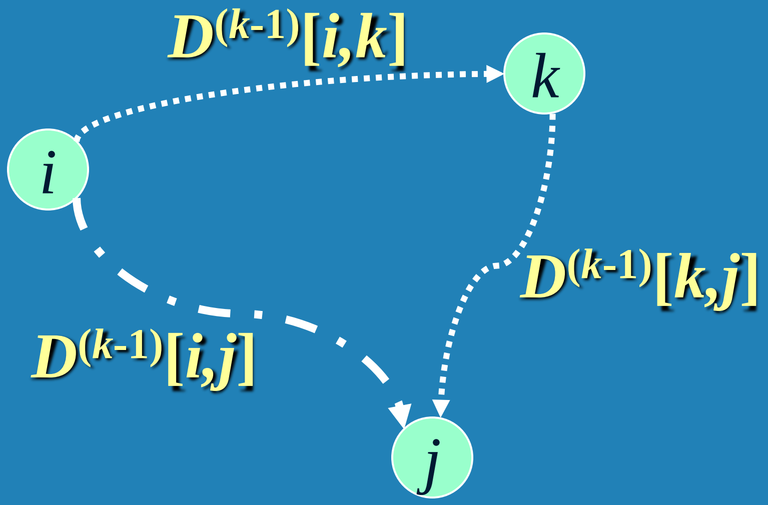
Example:



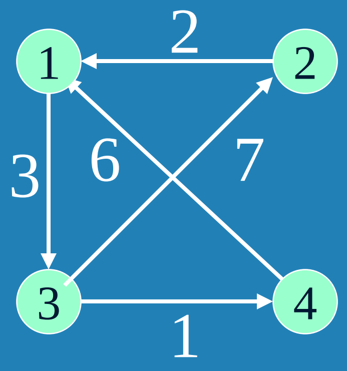
Floyd's Algorithm (matrix generation)

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)



$$D^{(0)} = \begin{array}{|c|ccc|}\hline 0 & \infty & 3 & \infty \\ \hline 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \\ \hline \end{array}$$

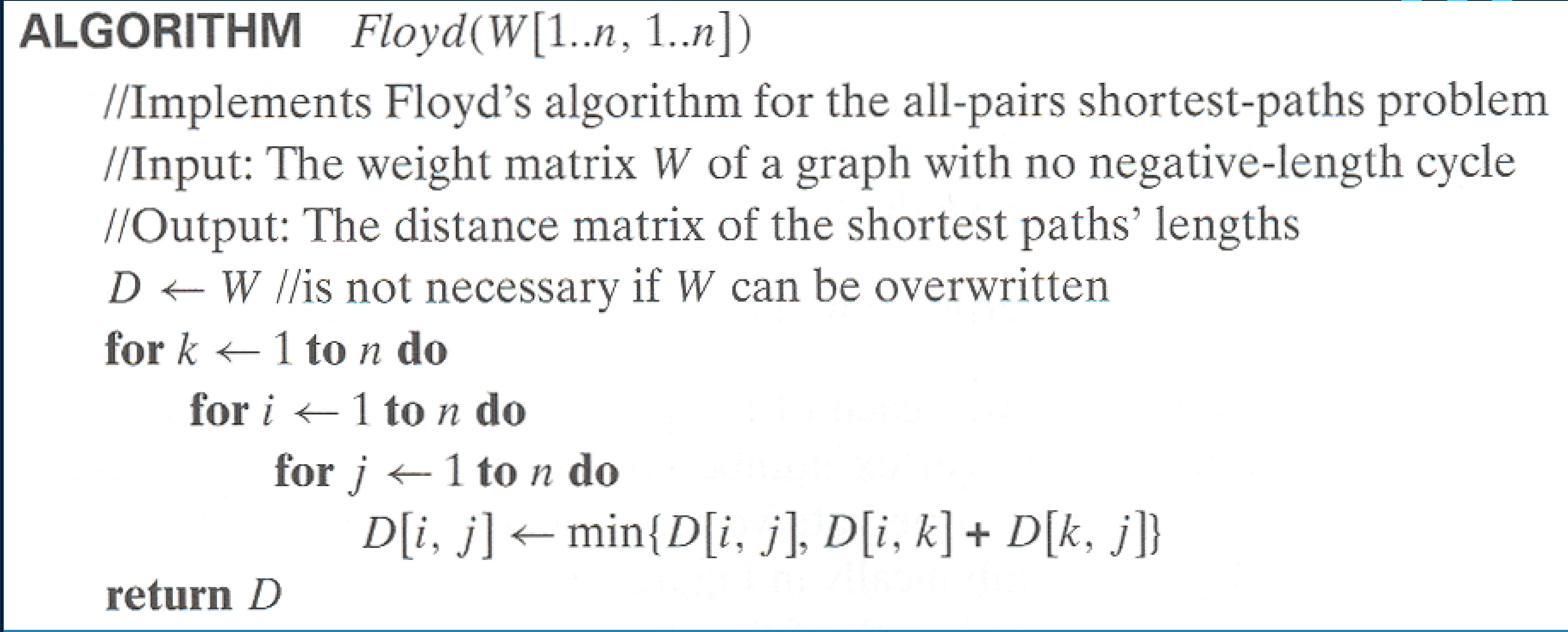
$$D^{(1)} = \begin{array}{cccc} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{array}$$

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

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

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

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
    $D \leftarrow W$ //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 $D$
```

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

Space efficiency: Matrices can be written over their predecessors

Note: Shortest paths themselves can be found, too (Problem 10)