Algorithms and Their Applications
- Dynamic Programming -

Won-Yong Shin

May 4th, 2020

연세대학교
YONSEI UNIVERSITY

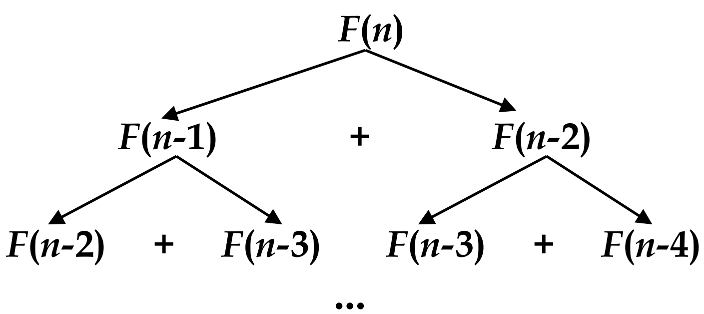
- Dynamic programming (DP)
  - A general algorithm design technique for solving problems defined by *recurrences with overlapping subproblems*
  - Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS
  - "Programming" here means "planning"

- 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**
  - Exact solution to the initial instance from that table

- Recall definition of Fibonacci numbers:

$$\boxed{\begin{array}{l} F(n) = F(n\text{-}1) + F(n\text{-}2) \\ F(0) = 0,\ F(1) = 1 \end{array}}$$

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



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

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

- Problem setup
  - 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

- 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** the last coin – the max amount is?
    - Those **with** the last coin – the max amount is?

$$\boxed{\begin{array}{l} F(n) = \max\{c_n + F(n-2),\ F(n-1)\} \quad \text{for } n > 1 \\ F(0) = 0, \qquad F(1) = c_1 \end{array}}$$

Coin-Row Problem - E.g.: 5, 1, 2, 10, 6, 2

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

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		

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

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**

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 $F[n]$

Time efficiency

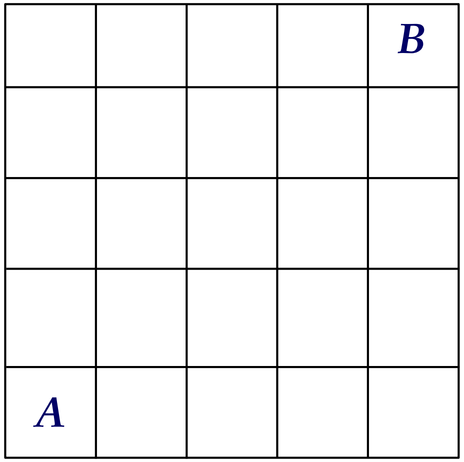
1) **Top-down approach**: Identical to the top-down computation of n^{th} Fibonacci number

⇨ $\Theta(\phi^n)$

2) **Bottom-up approach**

⇨ **$\Theta(n)$ time**

- Problem setup: shortest-path counting
  - Consider the problem of counting the number of shortest paths from point ***A*** to point ***B*** in a city with perfectly horizontal streets and vertical avenues
    - Let ***P*(*i*, *j*)** be the number of the shortest paths from square (1, 1) to square (*i*, *j*)

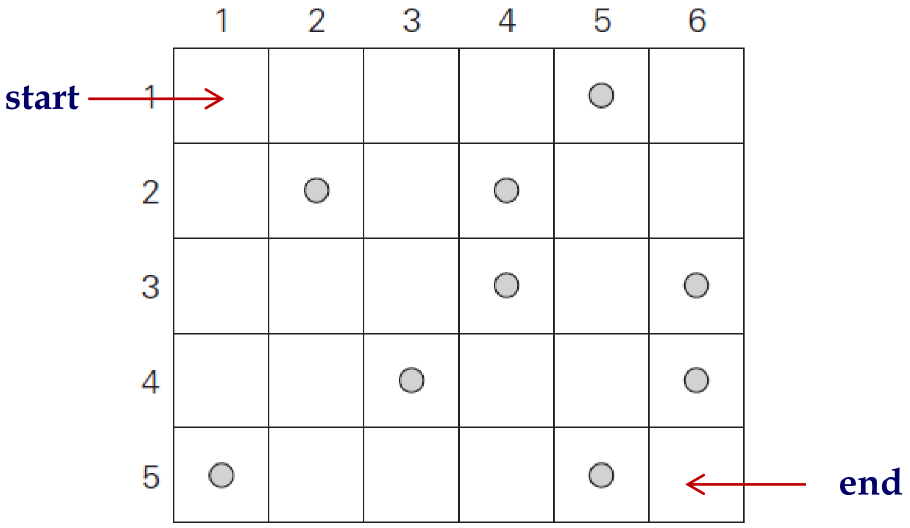


- DP solution to the path counting problem (*n*=8)

$$\begin{aligned} P(i,j) &= P(i,j-1)+P(i-1,j) \quad \text{for } 1 < i,j \le 8 \\ P(i,1) &= P(1,j) = 1 \quad \text{for } 1 \le i,j \le 8 \end{aligned}$$

1	8	36	120	330	792	1716	3432
1	7	28	84	210	462	924	1716
1	6	21	56	126	252	462	792
1	5	15	35	70	126	210	330
1	4	10	20	35	56	84	120
1	3	6	10	15	21	28	36
1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1

- Problem setup
  - 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



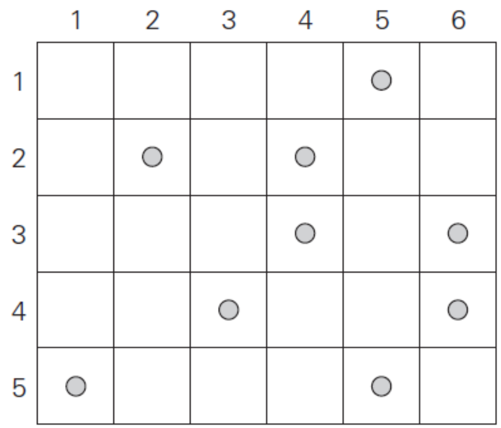
- DP solution
  - Let ***F*(*i*, *j*)** be the largest number of coins the robot can collect and bring to cell (*i*,*j*) in the *i*th row and *j*th column
  - The largest number of coins that can be brought to cell (*i*, *j*):
    - From the **left** neighbor?
    - From the neighbor **above**?

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

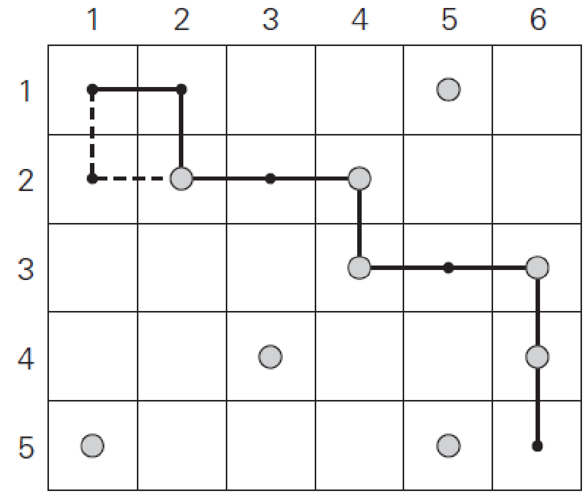
$$F(0, j) = 0 \ \text{ for } 1 \le j \le m \quad \text{and} \quad F(i, 0) = 0 \ \text{ for } 1 \le i \le n$$

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

Coin-Collecting Problem – E.g., 5×6 board



	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**



- Problem setup
  - Given n items of *integer* weights: $w_1 \ w_2 \ \ldots \ w_n$
    values: $v_1 \ v_2 \ \ldots \ v_n$

    a knapsack of integer capacity W, find most valuable subset of the items that fit into the knapsack

- Optimal solution
  - Consider instance defined by first i items and capacity j ($j \leq W$)
  - Let ***F(i,j)*** be optimal value of such instance
  - *Recurrence*:

Do not include the *i*th item **Do include the *i*th item**

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

  - Initial conditions:

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

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

capacity $W = 5$

	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**

- Time efficiency
  - Solution: $T(n) \in \Theta(Wn) = \Theta(n)$

- Memory function method
  - Combine the strengths of the *top-down* and *bottom-up* approaches
  - The example revisited

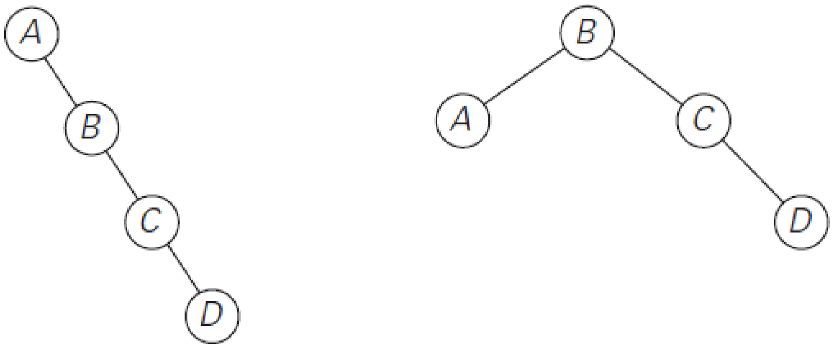
	i	capacity j = 0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3,\ v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2,\ v_4 = 15$	4	0	—	—	—	—	**37**

    - Only 11 out of 20 nontrivial values computed

- Other examples of DP algorithms
  - Constructing an optimal binary search tree (BST)
  - Warshall's algorithm for transitive closure
  - Floyd's algorithm for all-pairs shortest paths
  - Matrix-chain multiplication
  - Longest common subsequence

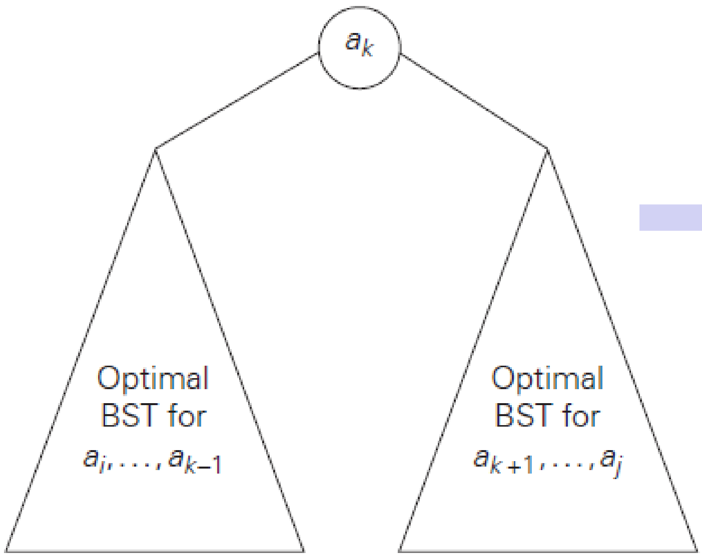
- Problem setup
  - Given n keys $a_1 < \ldots < a_n$ and probabilities $p_1 \le \ldots \le p_n$ searching for them, find a BST with a minimum average number of comparisons in successful search
  - Since the total number of BSTs with n nodes is given by $\frac{\binom{2n}{n}}{n+1}$, which grows *exponentially*, 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 the Optimal BST Problem (1/2)

- Overall procedure
  - 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 $\mathbf{1 \leq i \leq j \leq n}$
  - 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

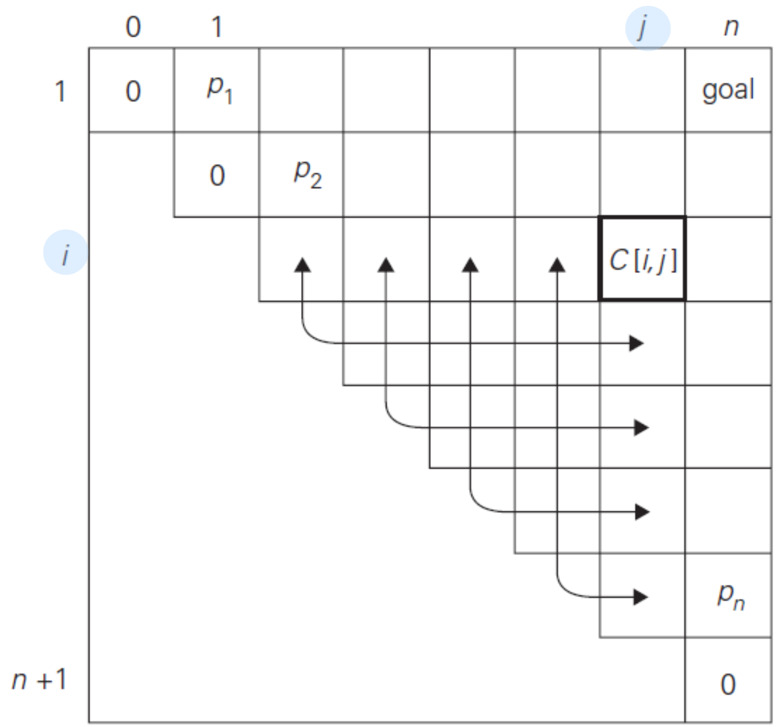


$$C(i, j) = \min_{i \leq k \leq j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1} + 1)$$
$$+ \sum_{s=k+1}^{j} p_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^{j} + 1)\}$$

- Overall procedure (Cont'd)
  - After simplifications, we obtain the recurrence for ***C(i,j)***:

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

$$C(i,i) = p_i \quad \text{for } 1 \le i \le n$$



Ex) Compute *C*(1, 3)

Optimal BST – Example #1 (1/2)

- The four-key set

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

- The initial tables

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

- Compute $C(1, 2)$:

$$C(1,2) = \min \begin{Bmatrix} k = 1: & C(1,0) + C(2,2) + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k = 2: & C(1,1) + C(3,2) + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{Bmatrix}$$
$$= 0.4$$

Optimal BST – Example #1 (2/2)

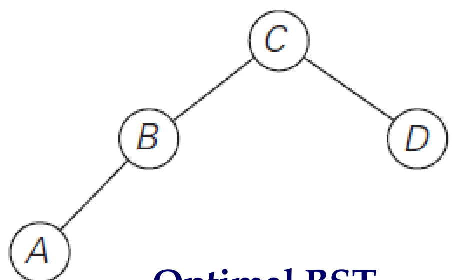
- The following final tables:

main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					



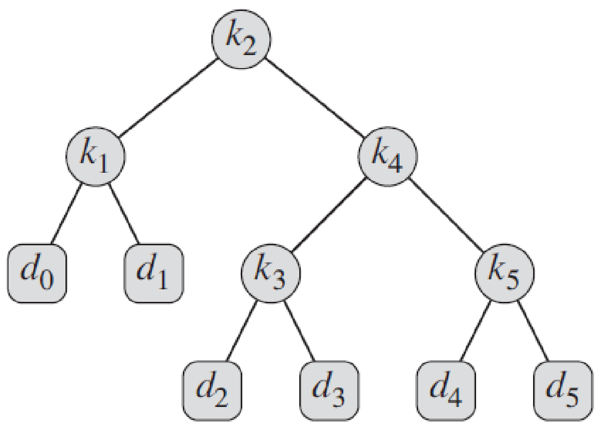
Optimal BST

Optimal BST – Example #2 (1/2)

- The five-key set (the use of "*dummy* keys")

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

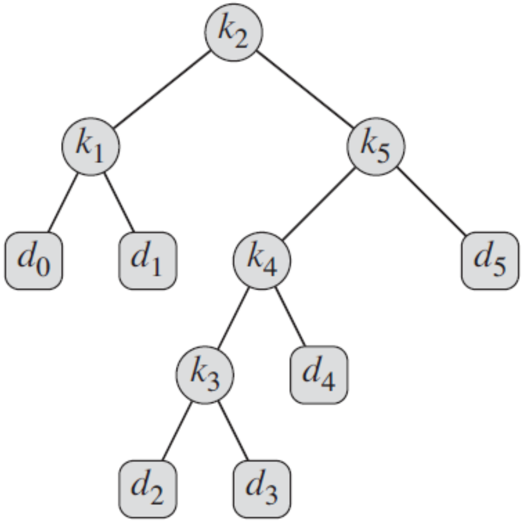
- Case 1



A BST with expected search cost 2.80

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

- Case 2



A BST with expected search cost 2.75

Optimal BST - Pseudocode

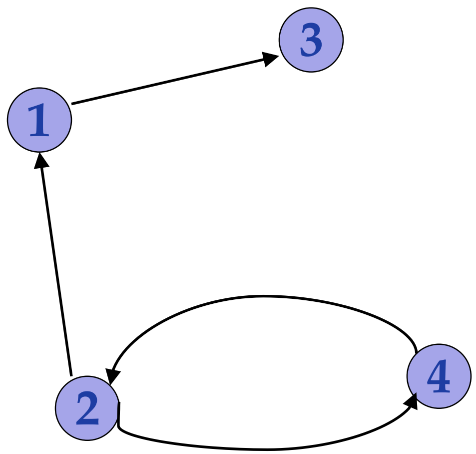
ALGORITHM *OptimalBST*($P[1..n]$)

```
//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
```

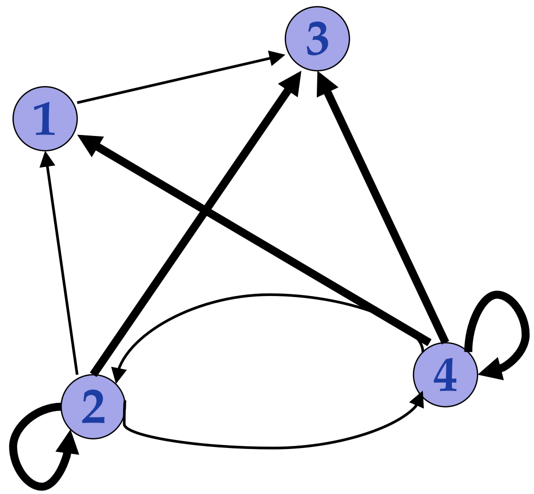
Recurrence

- Time efficiency
  - $\Theta(n^3)$ but can be reduced to $\mathbf{\Theta(n^2)}$ by taking advantage of *monotonicity of entries in the root table R(i, j)*
  - i.e., $R(i, j)$ is always in the range between $R(i, j\text{-}1)$ and $R(i\text{+}1, j)$
- Space efficiency
  - $\mathbf{\Theta(n^2)}$
- Method can be extended to include unsuccessful searches
  - E.g., the use of dummy keys

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

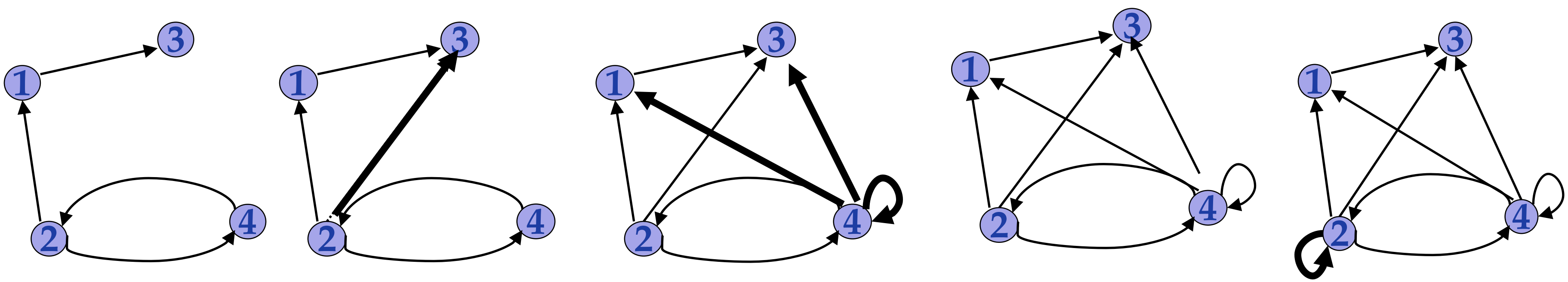


0	0	1	0
1	0	0	1
0	0	0	0
0	1	0	0



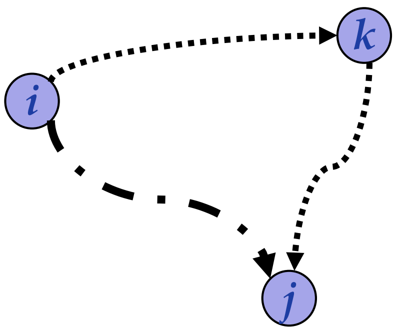
0	0	1	0
1	**1**	**1**	1
0	0	0	0
1	1	**1**	**1**

- Overall procedure
  - 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 $\boldsymbol{R^{(0)} = A}$ **(adjacency matrix)**, $\boldsymbol{R^{(n)} = T}$ **(transitive closure)**



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

- 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



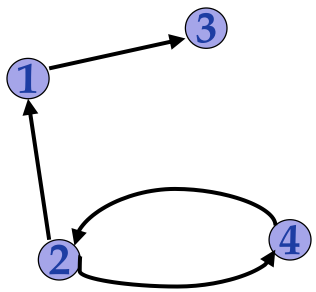
$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] \text{ (path using just } 1,\ldots,k\text{-}1) \\ \text{or} \\ R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j] \text{ (path from } i \text{ to } k \\ \text{and from } k \text{ to } j \text{ using just } 1,\ldots,k\text{-}1) \end{cases}$$

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

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

  - 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 $R^{(k-1)}$

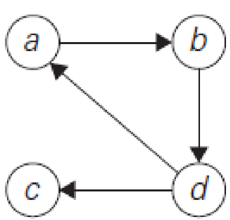
- Example 1



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

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

- Example 2



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

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 $R^{(n)}$

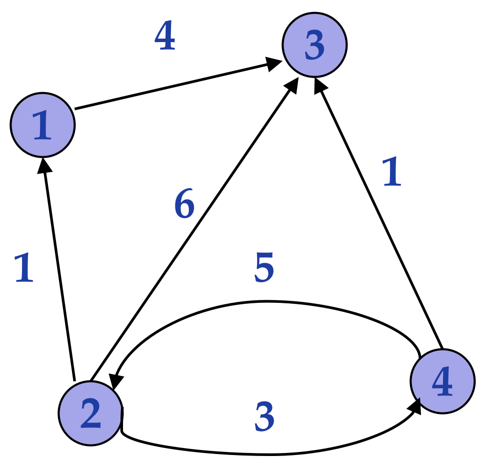
- Time efficiency
  - $T(n) \in \Theta(n^3)$
- Space efficiency
  - Matrices can be written over their predecessors

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

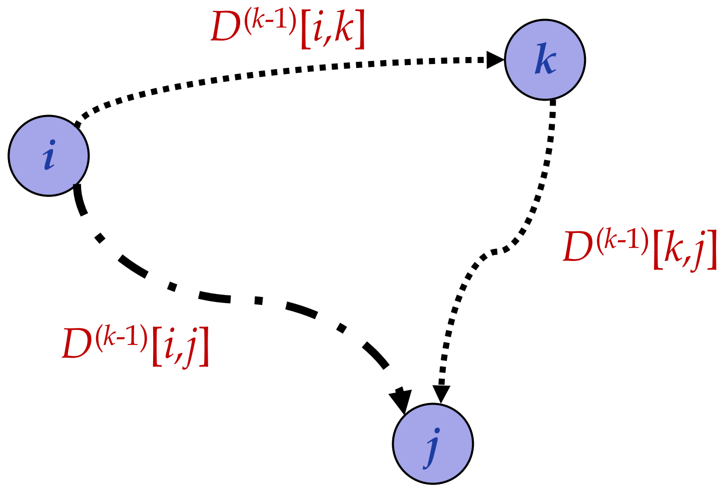
- Example:

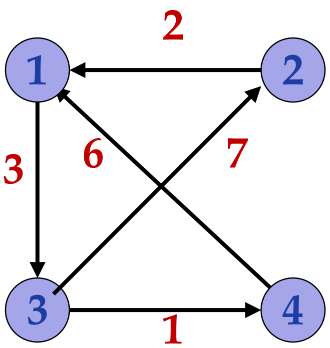


Floyd's Algorithm – Recurrence/Matrix Generation

- Recurrence
  - 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

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





$$D^{(0)} = \begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{matrix}$$

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

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

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

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

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
  - $T(n) \in \boldsymbol{\Theta}(n^3)$
- Space efficiency
  - Matrices can be written over their predecessors
- Note: Shortest paths themselves can be found