CPE231RC — Algorithms — Lecture7

Dynamic Programming

Dr. Prapong Prechaprapranwong

Computer Engineering

King Mongkut's University of Technology Thonburi

Dynamic Programming

planning

a technique for solving problems with overlapping subproblem by recurrence

overlap

For example: Fibonacci number $F(n) = \underline{F(n-1)} + \underline{F(n-2)}$ for $n > 1$ where $F(0) = 0$, $F(1) = 1$

Most Dynamic programming applications deal with optimization problems

- Basic examples
- Knapsack problem
- Warshall's and Floyd's algorithms

Example: Coin-Row Problem

Rules:

- Row of 6 coins whose values are 5, 1, 2, 10, 6, 2
- Pick up the maximum amount of money
- No two adjacent coins can be picked up.

Solve:

let F(n) be the maximum amount that can be picked up from the row of n coins

There are 2 groups:

1) groups of coin that include the last coin *n* → $c_n + F(n\text{-}2)$

2) groups of coin that not include the last coin *n* → $F(n\text{-}1)$

$$F(i) = \max\{c_i + F(i-2), F(i-1)\}, \quad 2 \le i \le n$$

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

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 $F[n]$

Time complexity Θ(n)
Space complexity Θ(n)

Solve: Coin-Row Problem

$$F(i) = \max\{c_i + F(i-2), F(i-1)\}, \quad 2 \le i \le n$$

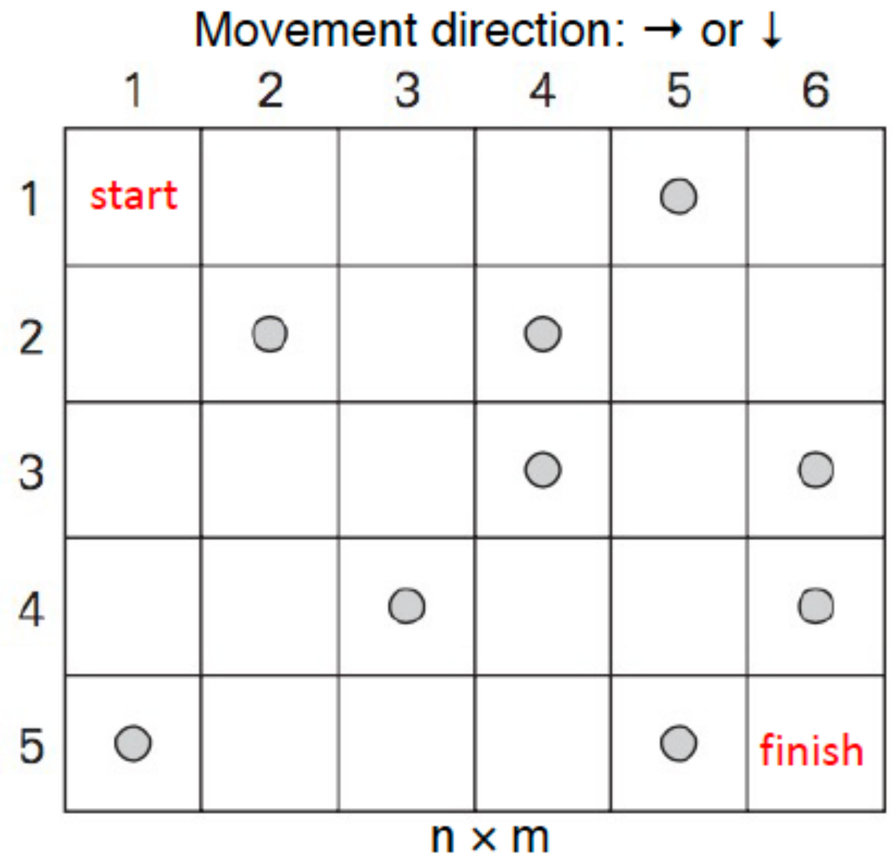
$$F(0) = 0, \quad F(1) = c_1$$

i	0	1	2	3	4	5	6
C_i							
F(i)							

Example: Coin Collecting Problem

Rules:

- Coins are placed in the cells of an *n* x *m* board, no more than 1 coin per cell
- start from upper left cell and end at the bottom right cell
- can move only right or down direction in each step
- find maximum coin and a path to collect the coins



Solve:

- Let $F(i,j)$ be the largest amount of coins that can collect and bring to cell (i,j)
  - (i,j) only reachable from left or above.
  - Largest # coins brought to these cells : $F(i,j-1)$ and $F(i-1,j)$

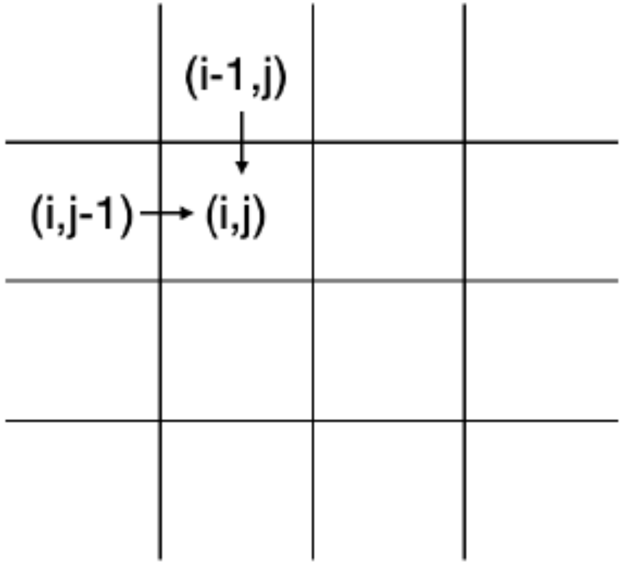
- Therefore, $F(i,j)$ satisfies the following formula:

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

$$F(0,j) = 0, \quad 1 \le j \le m$$ row 0 brings no coin

$$F(i,0) = 0, \quad 1 \le i \le n$$ column 0 brings no coin

$$c_{ij} = \begin{cases} 1 & \text{coin in cell } (i,j) \\ 0 & \text{no coin in cell } (i,j) \end{cases}$$



- Fill in n×m table of $F(i,j)$ values either row-by-row or column-by-column.

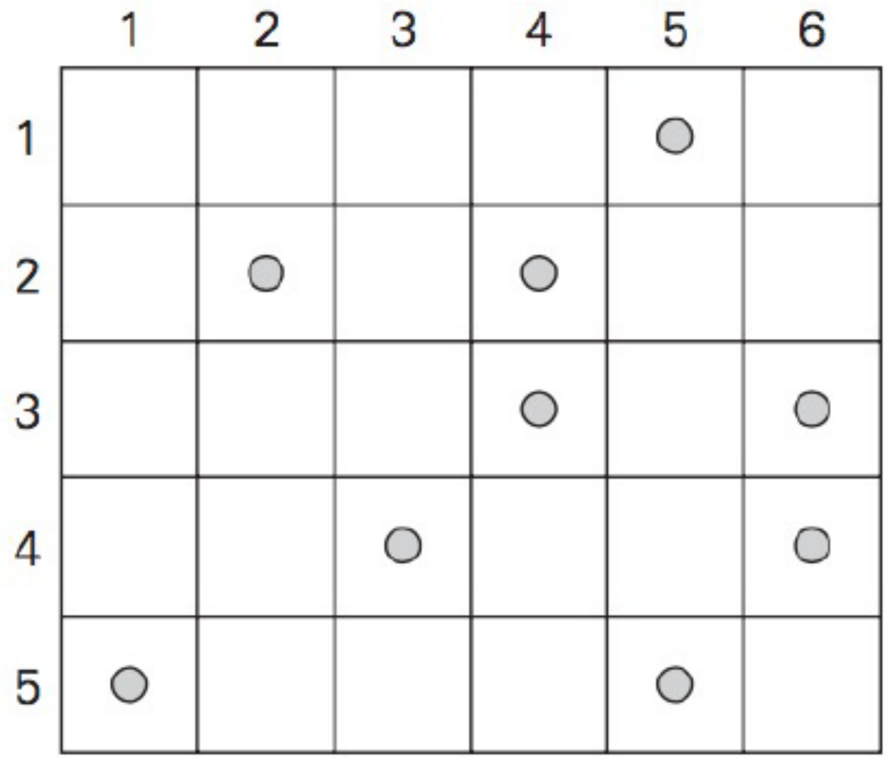
ALGORITHM *RobotCoinCollection*$(C[1..n, 1..m])$

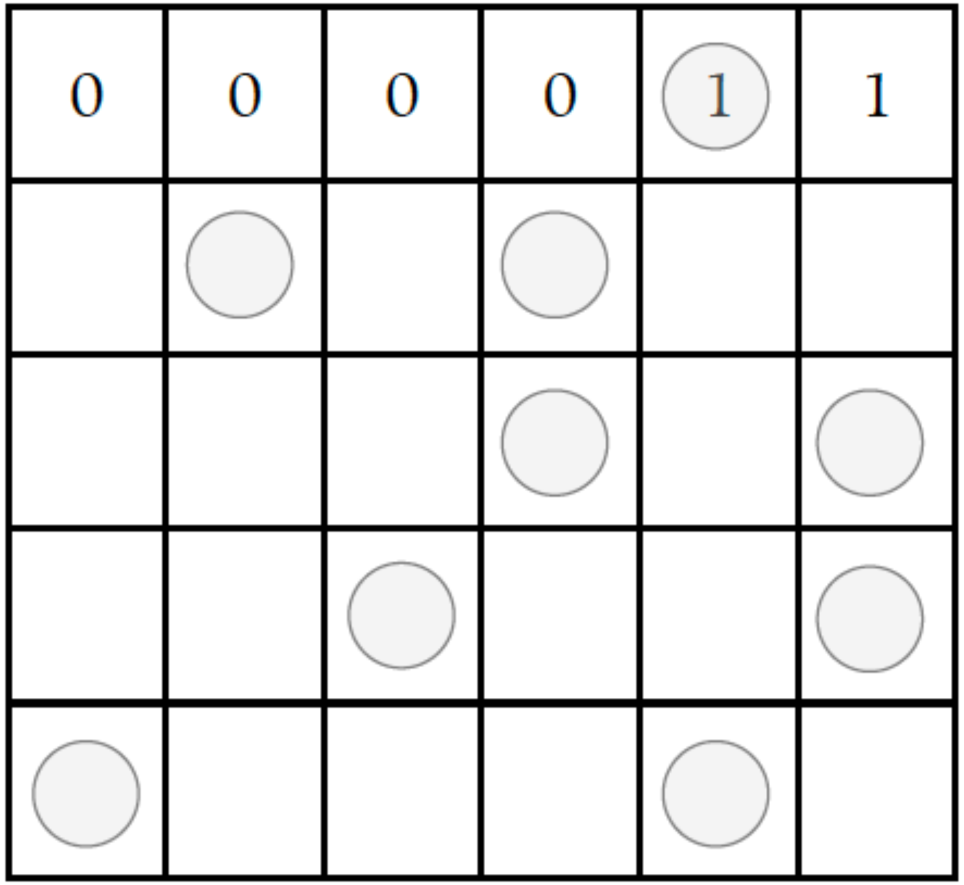
//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an $n \times m$ board by starting at (1, 1)
//and moving right and down from upper left to down right corner
//Input: Matrix $C[1..n, 1..m]$ whose elements are equal to 1 and 0
//for cells with and without a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

Solve: Coin Collection Problem



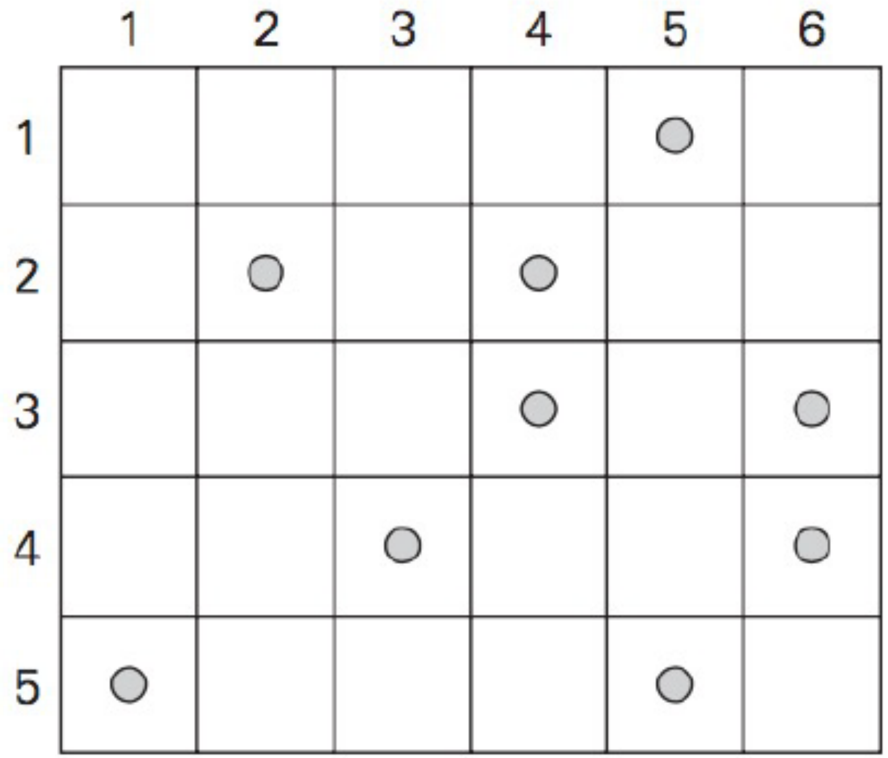


For i = 1,

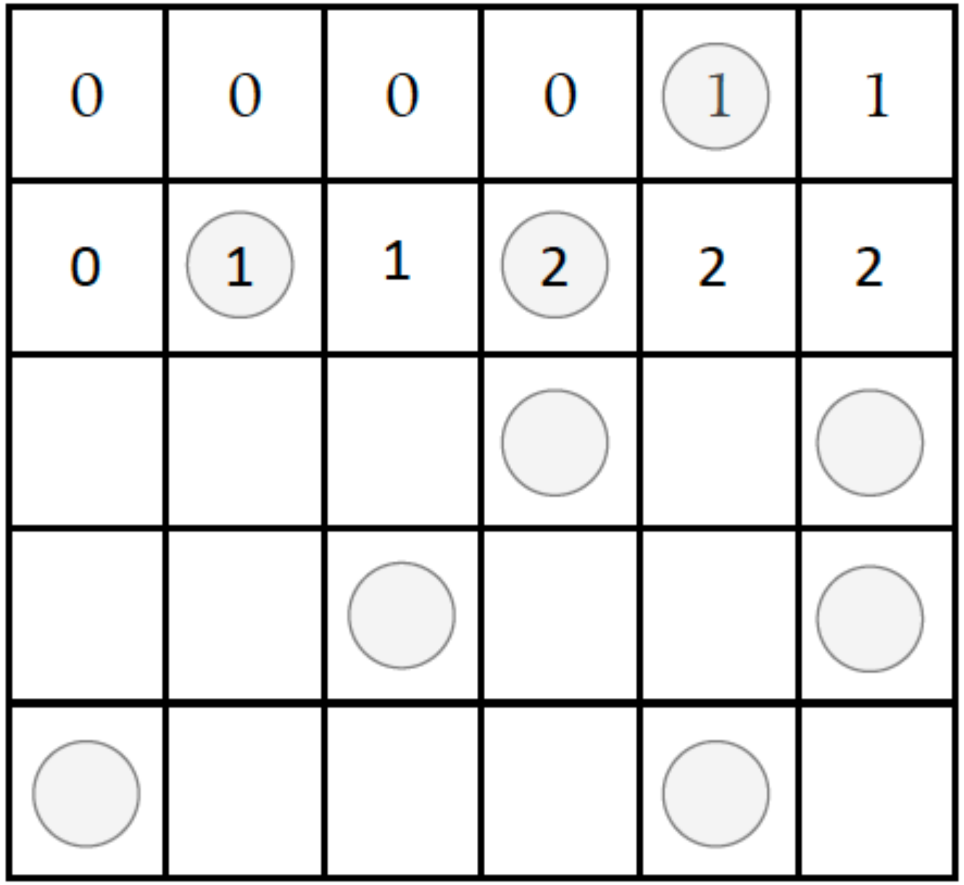
$F(1,1) = c_{11} = 0$

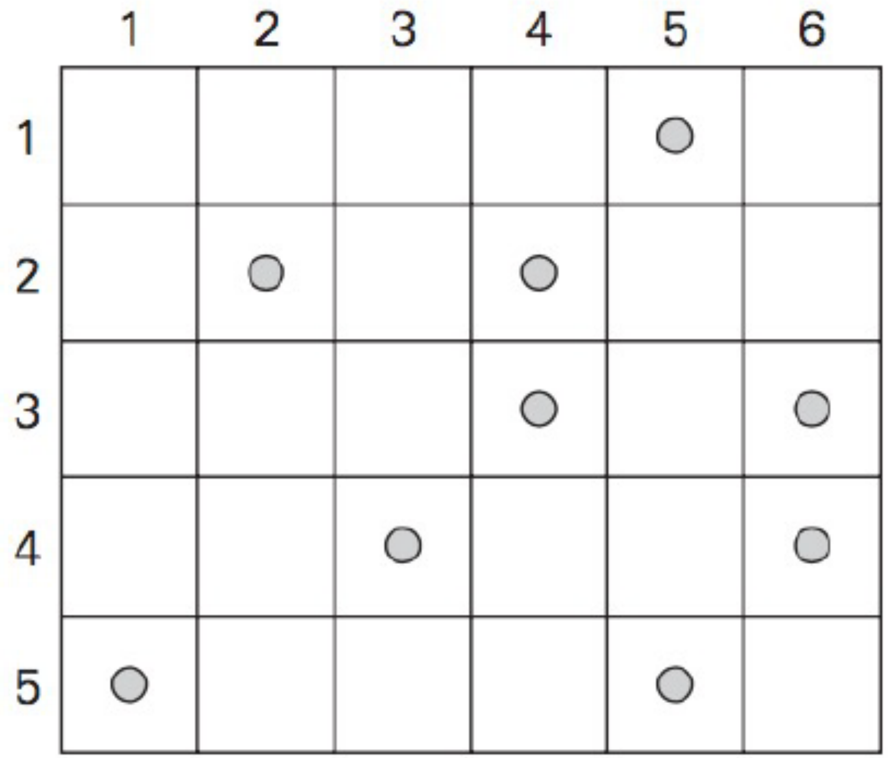
$$F(1, j) = \max\{\cancelto{0}{F(0,j)}, F(1,j-1)\} + c_{ij}$$
$$= F(1,j-1)+c_{ij},\ \ 2 <= j <= m$$

$$F(i, 1) = \max\{F(i-1, 1), \cancelto{0}{F(i,0)}\} + c_{ij}$$
$$= F(i-1,1)+c_{ij},\ \ 2 <= i <= n$$



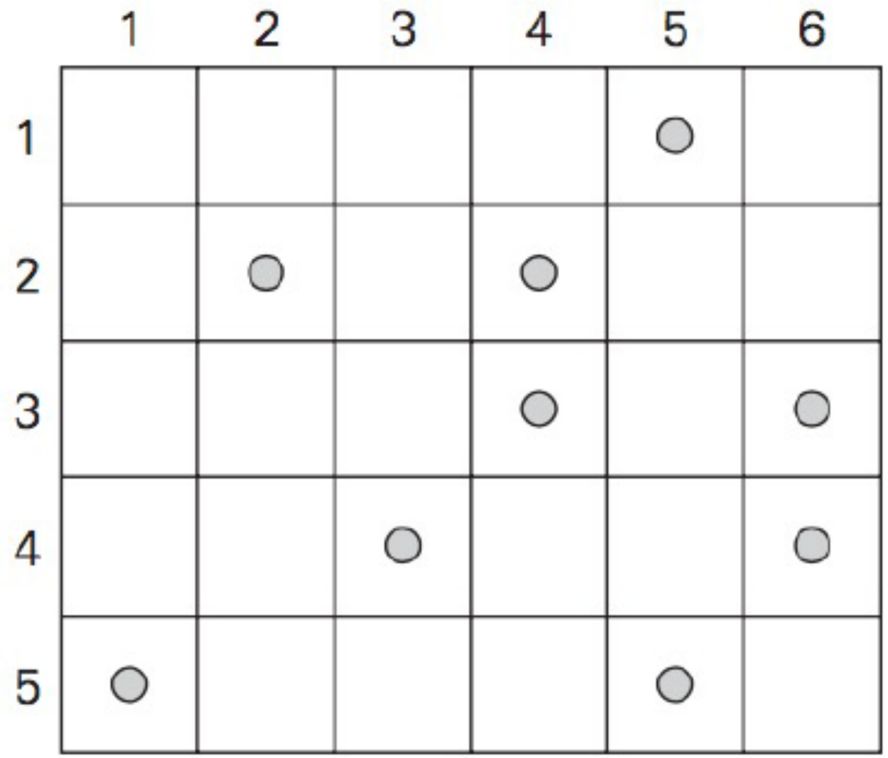
For i = 2





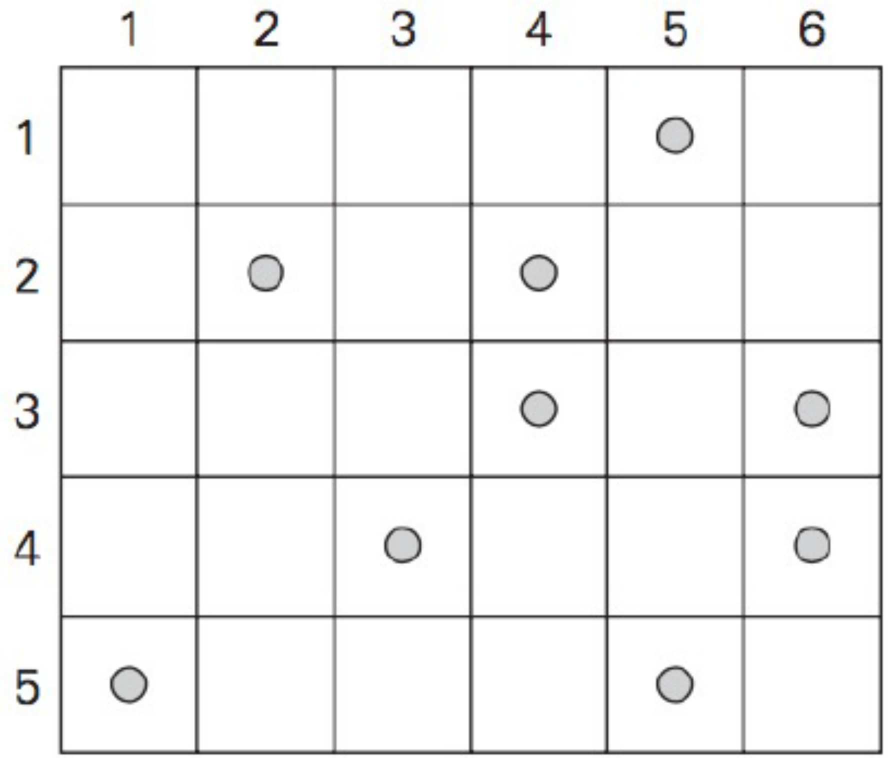
For i = 3

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



For i = 4

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

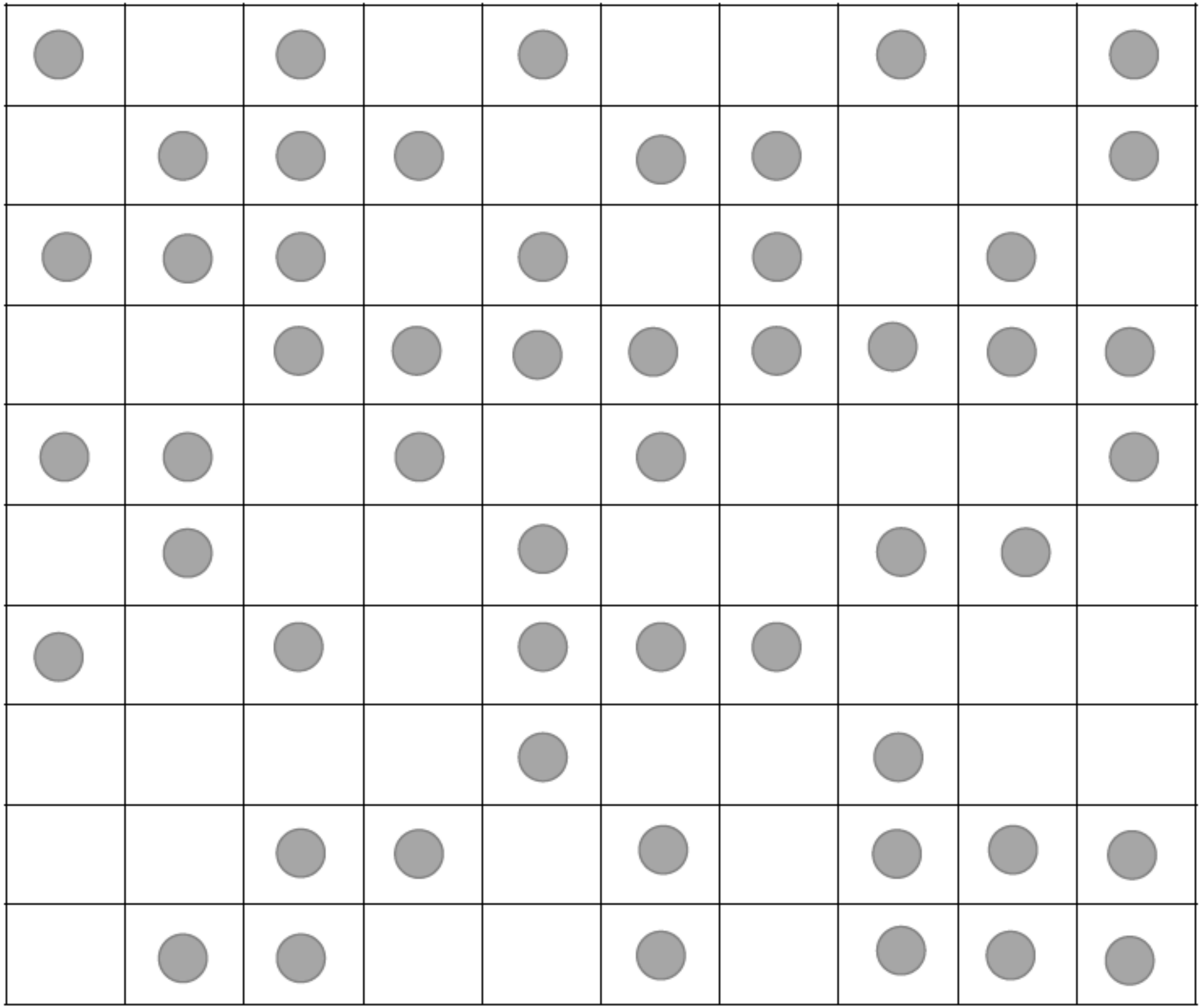


For i = 5

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

Max Coins = 5

Practice I) Find the maximum collected coins



Concept - Dynamic Programming

- Dynamic Programming = Dynamic Planning
- Technique to solve problems with overlapping subproblems

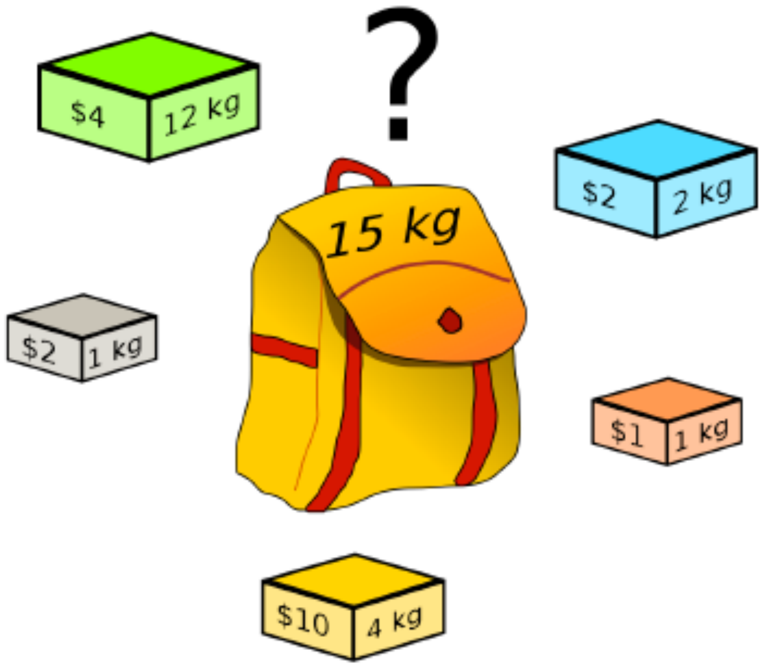
    - Subproblems typically arise from **recurrence relation** of the problem solution and subproblem solutions

    - Subproblem is solved as **a multistage decision process** only once and results recorded in a table like **space-time trade-off** design = Memoization, Tabulation

Knapsack Problem

Given n items of know weight $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity W

✓ find <u>the most value</u> subset of the items that fit in to the knapsack

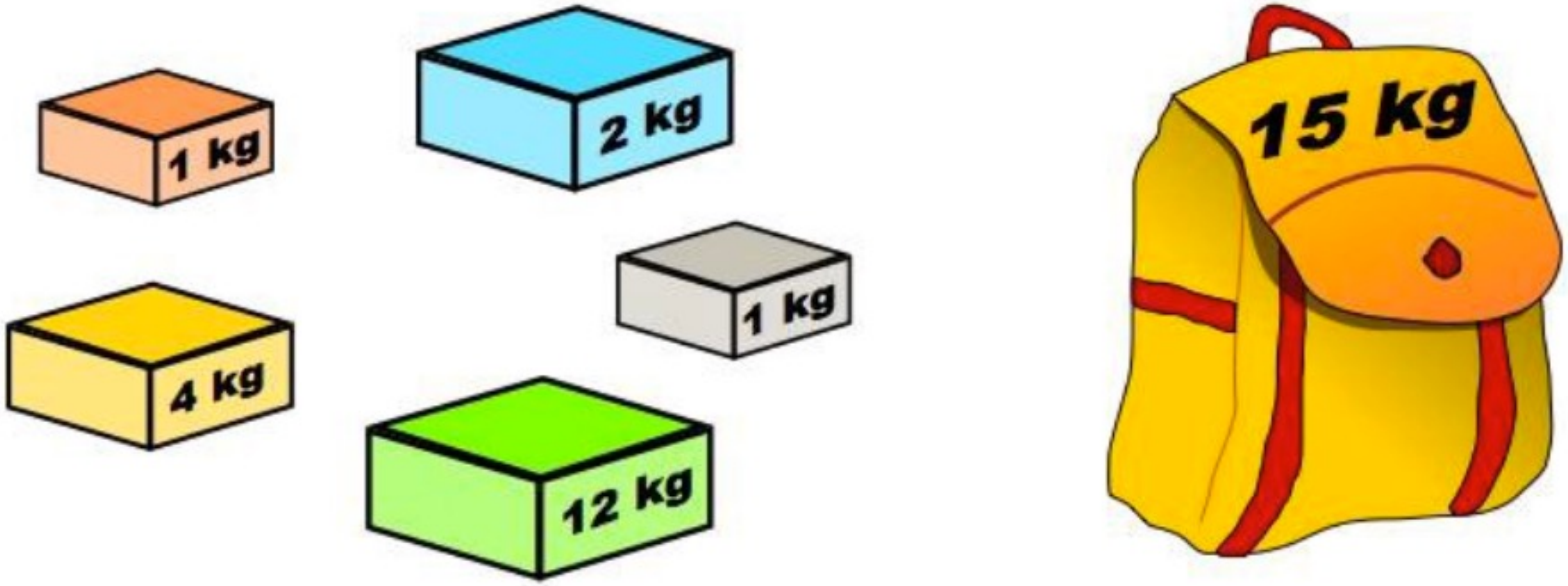


Step to solve (Brute force & exhaustive search approach)

1. Generating all possible subset of the items
2. Calculate all weight of those subset in knapsack
3. find the most value subset

- The first i items (**1,2,…,*i***) $1 \leq i \leq n$ with weights $\boldsymbol{w_1, w_2, \ldots w_i}$ and values $\boldsymbol{v_1, v_2, \ldots v_i}$ and knapsack capacity j , $1 \leq j \leq \boldsymbol{W}$
- let ***F(i,j)*** = the optimal value obtained from a subset of first i items that fit into the knapsack of capacity ***W***

“the value of the most valuable subset of the first *i* items that fit the knapsack of capacity *j*”



divide all the subset of the first *i* times that fit the knapsack of capacity *j* into 2 categories

1) do not include the i^{th} item — the value of an optional subset is *F(i-1,j)*

the value of the first $i-1$ items

2) include the i^{th} item — the value of an optional subset is $v_i + F(i-1, j-w_i)$

Weight before including item i^{th}

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

Objective : find F(*n*,*W*), the maximal value of a subset of the *n* given items that fit into the knapsack of capacity *W*

Objective : find F(n,W), the maximal value of a subset of the n given items that fit into the knapsack of capacity W

	0	$j\text{-}w_i$	j	W
0	0	0	0	0
$i\text{-}1$	0	$F(i\text{-}1,j\text{-}w_i)$	$F(i\text{-}1,j)$	
		compare		
w_i , v_i, i	0		$F(i,j)$	
n	0			goal

Example Knapsack Problem with dynamic programming approach

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

capacity W = 5

$w_1=2$, $v_1 = 12$

$w_2=1$, $v_2 = 10$

$w_3=3$, $v_3 = 20$

$w_4=2$, $v_4 = 15$

capacity j

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

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

Example Knapsack Problem with dynamic programming approach

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

capacity 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

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

Example Knapsack Problem with dynamic programming approach

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

capacity 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	$\boxed{37}$

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

Practice II) Knapsack Problem with dynamic programming approach (20 minutes)

item	weight	value
1	3	$25
2	2	$20
3	1	$15
4	4	$40
5	5	$50

capacity W = 6

$w_1=3,\ v_1 = 25$

$w_2=2,\ v_2 = 20$

$w_3=1,\ v_3 = 15$

$w_4=4,\ v_4 = 40$

$w_5=5,\ v_4 = 50$

capacity j

i	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0						
2	0						
3	0						
4	0						
5	0						

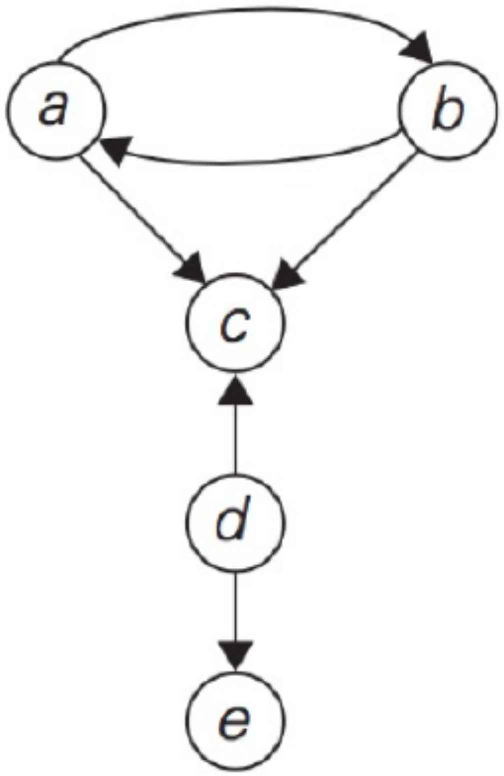
Warshall's and Floyd's Algorithms

directed path from the i^{th} vertex to the j^{th} vertex

- **Warshall's algorithm** → transitive closure of a directed graph
- **Floyd's algorithm** → all-pairs shortest path

Both method start from an <u>Adjacency Matrix</u>

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \left[\begin{array}{ccccc} \; & \; & \; & \; & \; \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{array} \right] \end{array}$$



if the i^{th} vertex connects to the j^{th} vertex then A_{ij} is 1 otherwise is 0

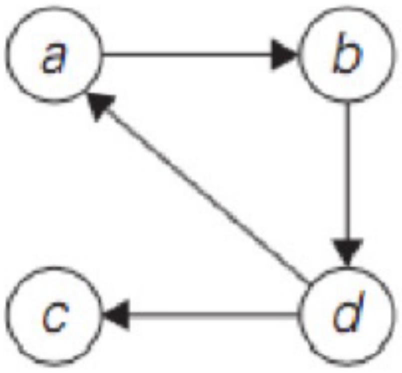
Warshall's algorithm

- Examples of Transitive closure application
  - Dependency of spreadsheet cells
  - Data flow in software design

- Depth-first-search or Breadth-first-search can be used to generate a transitive closure of a digraph.
  - Perform a traversal at i^{th} vertex and fill in columns in the i^{th} row
  - Ex: Try DFS starting at vertex a

However too many times of traversal for every vertex as a starting points and must transverse the same graph many times

Warshall's algorithm



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

$a \to b$

$b \to d$

$d \to a$ $\quad$ $d \to c$

Rules for applying Warshall's algorithm

- If r_{ij} is 1 in $\boldsymbol{R}^{(k-1)}$, it remains 1 in $\boldsymbol{R}^{(k)}$
- If r_{ij} is 0 in $\boldsymbol{R}^{(k-1)}$, it changes to 1 in $\boldsymbol{R}^{(k)}$ if and only if r_{ik} and r_{kj} in $\boldsymbol{R}^{(k-1)}$ are both 1.

$$\underset{\text{via vertex a}}{R^{(1)}} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

r_{da} and $r_{ab} = 1$

$d \to a \to b$

$$\underset{\text{via vertex b}}{R^{(2)}} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

r_{ab} and $r_{bd} = 1$

$a \to b \to d$

r_{db} and $r_{bd} = 1$

$d \to a \to b \to d$

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

via vertex c

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

via vertex d

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

//Wallshall'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 ← 1 **to** n **do**

  for i ← 1 **to** n **do** $O(n^3)$

    for j ← 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 $R^{(n)}$

Practice III) Apply Warshall's algorithm to find the transitive closure of the digraph defined by the following adjacency matrix and draw the graph (10 minutes)

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

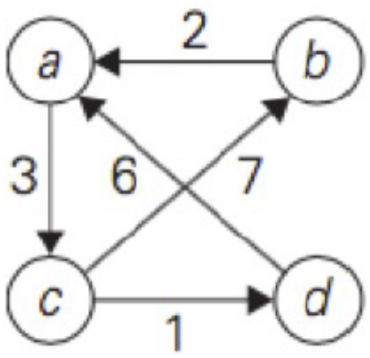
Floyd’s algorithm

All-pairs shortest paths

- applicable to both undirected and directed weighted graph
- consider a ***positive weighted*** connected graph W
- Find the shortest paths from each vertex to all the others.
- Example of application
  - precompute distances for motion planning in computer games
- Different from Dijkstra’s algorithm (single source shortest path)

- Distance matrix **D** contains the shortest path length from i^{th} vertex to j^{th} vertex

Floyd's algorithm



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

all self loop back = 0

Rules for applying Floyd's algorithm:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \text{ for } k \geq 1, d_{ij}^{(0)} = w_{ij}$$

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

via vertex a

$d_{ba} + d_{ac} = 2+3 = 5$

$d_{da} + d_{ac} = 6 +3 = 9$

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

via vertex b

Floyd's algorithm

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

via vertex c

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

via vertex d

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

//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 path's lengths

```
D ← W // is not necessary if W can be overwritten
for k ← 1 to n do
    for k ← 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
```

Practice IV) Solve the all-pair shortest path problem for the digraph with this weight matrix and draw this graph (20 minutes)

$$\begin{bmatrix} 0 & 3 & \infty & 2 & 6 \\ 5 & 0 & 4 & 2 & \infty \\ \infty & \infty & 0 & 5 & \infty \\ \infty & \infty & 1 & 0 & 4 \\ 5 & \infty & \infty & \infty & 0 \end{bmatrix}$$

Summary

- Dynamic programming solves problems whose final solution
  - expressed as recurrent relation of subproblem solutions
  - involved a multi-stage decision process

- Subproblem solutions stored and retrieved for later uses.

- Sample problems solved by dynamic programming
  - Simple ones like Coin-row and Coin-collection problems.
  - Integer version of the knapsack problem.
  - Transitive closure and all-pair shortest path problems.

Assignment: Research and Learning more about memoization and tabulation (using python’s decoration)