Name: Pattarapon Buathong Student ID 62070504012

M08 – 1 : Dynamic Programming I

1. Objectives / Outcome

1) Students will be able to recognize the concept of dynamic programming, especially three basic examples and knapsack problems.
2) Students will be able to recognize the concept of dynamic programming, especially warshall's and floyd's algorithms.
3) Students will be able to design and implement the warshall's algorithm for the simple problem with programming language.

2. Related Contents

Dynamic Programming (DP) is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping sub-problems. As a general method for optimizing multistage decision processes, the words "Programming" here means "Planning" and does not refer to computer programming. (Richard Bellman,1950) DP is a technique for solving problems with overlapping sub-problems. Typically, these sub-problems arise from a recurrence relating a given problem's solution to solutions of its smaller sub-problems. Rather than solving overlapping sub-problems again and again, DP suggests solving each of the smaller sub-problems only once and recording the results in a table from which a solution to the original problem can then be obtained.

The main idea of DP is 1) set up a recurrence relating a solution to a larger instance to solutions of some smaller instances 2) solve smaller instances once 3) record solutions in a table, and 4) extract solution to the initial instance from that table.

2.1. Basic Examples

The goal of this section is to introduce dynamic programming via three typical examples.

2.1.1. 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: 1) those that include the last coin and 2) those without it.

Name: Pattarapon Buathong Student ID 62070504012

The largest amount we can get from the first group is equal to c_n+ F (n – 2)—the value of the n^{th} coin plus the maximum amount we can pick up from the first n – 2 coins. The maximum amount we can get from the second group is equal to F(n – 1) by the

```
1. ALGORITHM CoinRow( C[1..n] )
2. // Input :  Array C[1..n] of positive integers indicating the coin values
3. //Output : The maximum amount of money that can be picked up
4. F[0] <- 0; F[1] <- C[1]
5. for i <- 2 to n do
6.     F[i] <- max(C[i] + F[i-2], F[i-1])
7.
8. return F[n]
```

definition of F(n). Thus, we have the following recurrence subject to the obvious initial conditions:

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \ for\ n > 1,$$
$$F(0) = 0, F(1) = C_1$$

For example, Solving the coin-row problem by dynamic programming for the coin row 5, 1, 2, 10, 6, 2. Work out the remaining values? How do you determine which coins are selected?

	n =	0	1	2	3	4	5	6
F[0] = 0 , F[1] = C_1 = 5	C_n		*5*	*1*	*2*	*10*	*6*	*2*
	F(n)	**0**	**5**					
F[2] = max{1+0, 5} = 5	C_n		*5*	*1*	*2*	*10*	*6*	*2*
	F(n)	0	5	**5**				
F[3] = max{2+5, 5} = 7	C_n		*5*	*1*	*2*	*10*	*6*	*2*
	F(n)	0	5	5	**7**			
F[4] = max{10+5, 7} = 15	C_n		*5*	*1*	*2*	*10*	*6*	*2*
	F(n)	0	5	5	7	15		
F[5] = max{6+7, 15} = 15	C_n		*5*	*1*	*2*	*10*	*6*	*2*
	F(n)	0	5	5	7	15	15	

Name: Pattarapon Buathong Student ID 62070504012

F[6] = max{2+15, 15} = 17	C_n		*5*	*1*	*2*	*10*	*6*	*2*
	F(n)	0	5	5	7	15	15	**17**

Using the CoinRow to find F(n), the largest amount of money that can be picked up, as well as the coins composing an optimal set, clearly takes **Θ (n) time** and **Θ (n) space**.

2.1.2. Coin-collection problem

Let F(i, j) be the largest number of coins the robot can collect and bring to the cell (i, j) in the ith row and jth column of the board. It can reach this cell either from the adjacent cell (i – 1, j) above it or from the adjacent cell (i, j – 1) to the left of it. The largest numbers of coins that can be brought to these cells are F(i – 1, j) and F(i, j – 1), respectively. Of course, there are no adjacent cells above the cells in the first row, and there are no adjacent cells to the left of the cells in the first column. For those cells, we assume that F(i – 1, j) and F(i, j – 1) are equal to 0 for their nonexistent neighbors. Therefore, the largest number of coins the robot can bring to cell (i, j) is the maximum of these two numbers plus one possible coin at cell (i, j) itself. In other words, we have the following formula for F(i, j):

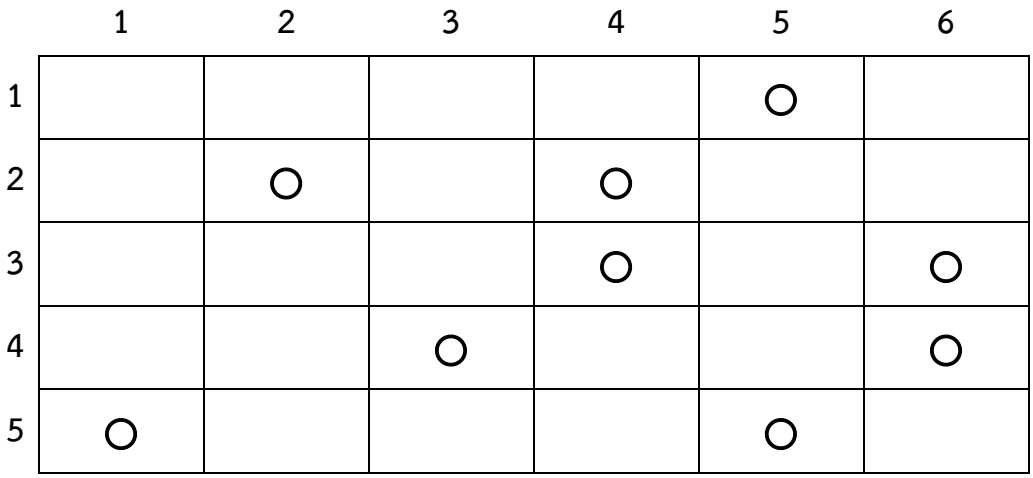
$$F(i,j) = \max\{F(n-1,j) + F(i,j-1)\} + C_{ij} \quad for\ 1 \le i \le n,\ 1 \le i \le m$$
$$F(0,j) = 0, for\ 1 \le i \le m \quad and\ F(i,0) = 0, for\ 1 \le i \le n$$

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

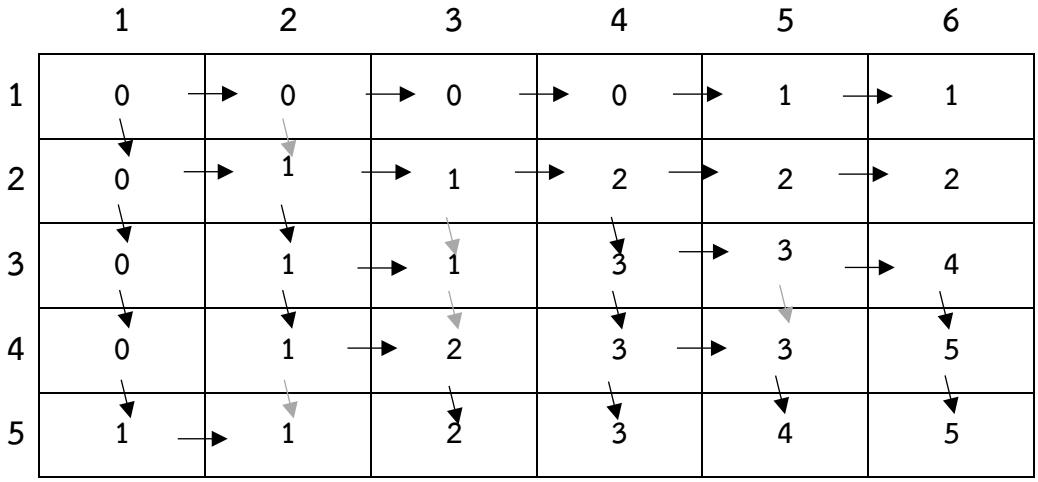
---

1. ***ALGORITHM*** *RobotCoinCollection(C[1..n, 1..m])*
2. *// Input : Matrix C[1..n, 1..m] whose elements are equal to 1 and 0 for cells with and without a coin, respectively*
3. *//Output : Largest number of coins the robot can bring to cell (n, m)*
4. *F[1, 1] <- C[1, 1]; for j <- 2 to m do F[1, j] <- F[1, j-1] + C[1, j]*
5. *For i <- 2 to n do*
6. *F[i, 1] <- F[i-1, 1] + C[i, 1]*
7. *For j <- 2 to m do*
8. *F[i, j] <- max(F[i-1, j], F[I, j-1]) + C[i, j]*
9. *Return F[n, m]*

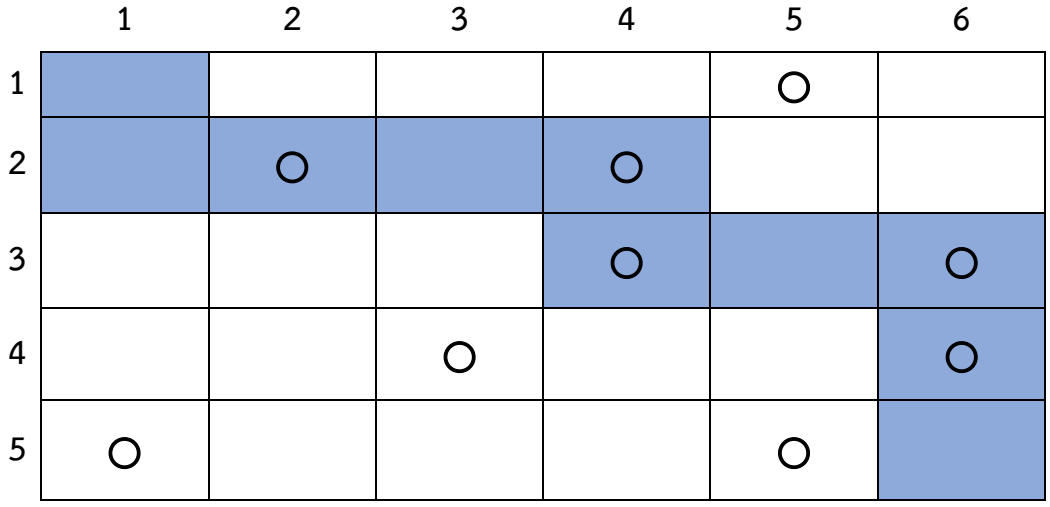
Name: Pattarapon Buathong Student ID 62070504012



a) Coins to collect.



b) Dynamic programming algorithm results.



c) Path Solution to collect 5 coins, the maximum number of coins possible.

Name: Pattarapon Buathong Student ID 62070504012

The time efficiency of the algorithm is **Θ (nm).** Its space efficiency is, obviously, also **Θ (nm).** Tracing the computations backward makes it possible to get an optimal path: if F(i − 1, j) > F(i, j − 1), an optimal path to cell (i, j) must come down from the adjacent cell above it; if F(i − 1, j) < F(i, j − 1), an optimal path to cell (i, j) must come from the adjacent cell on the left; and if F(i − 1, j) = F(i, j − 1), it can reach cell (i, j) from either direction. This yields two optimal paths for the instance in Table a), which are shown in Table c) as above. If ties are ignored, one optimal path can be obtained in Θ (n + m) time.

2.2. DP Knapsack Problem

To design a dynamic programming algorithm, we need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller sub-instances. Let us consider an instance defined by the first i items, 1≤ i ≤ n, with weights $w_1, \ldots, w_i$, values $v_1, \ldots, v_i$, and knapsack capacity w, 1 ≤ w ≤ W.

Let F (i, j) be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j. We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories: those that do not include the i^{th} item and those that do. Note the following:

- *Among the subsets that do not include the i^{th} item, the value of an optimal subset is, by definition, F(i − 1, w).*
- *Among the subsets that do include the i^{th} item (hence, w − wi ≥ 0), an optimal subset is made up of this item and an optimal subset of the first i − 1 items that fits* into the knapsack of capacity w − w_i . The value of such an optimal subset is vi + F(i − 1, w − w_i).

These observations lead to the following recurrence:

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

Name: Pattarapon Buathong Student ID 62070504012

	0	....	$w- w_i$	w	W
0	0		0	0	0
i-1	0		F(i – 1, w – w_i)	*F(i – 1, w)*	
w_i, v_i i	0			F (i, w)	
...					
n	0				F (n, W)

Goal for solving the knapsack problem by dynamic programming.

Example of solving an instance of the knapsack problem by the dynamic programming algorithm. **What is the maximum sum of values?**

		capacity W					
	i/w	0	1	2	3	4	5
No item	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

```
1.  ALGORITHM DPKnapsack(w[1..n], v[1..n], W)
2.  // Input :  integer V[0..n,0..W],  P[1..n,1..W]
3.  //Output : F[n,W] and the optimal subset by backtracing
4.  for w<-0 to W do
5.      F[0, w] <- 0
6.      for i<-0 to n do
7.          F[i, 0] <- 0
8.      for i<-1 to n do
9.          For w<-1 to W do
10.             if w[i] <= w then
11.                 F[i, w] <- max{F[i-1, w], v[i] + F[i-1, w-w[i]}
12.             else
13.                 F[i, w] <- F[i-1, w]
14. Return F[n, w] and the optimal subset by backtracing
```

2.3. Longest Common Subsequence (LCS)

Name: Pattarapon Buathong Student ID 62070504012

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, predent*. 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. The way to compute LCS is begin with define $A=a_1a_2...a_m$ and $B=b_1b_2...b_n$. *len*(*i, j*) is the length of an LCS between $a_1a_2...a_i$ and $b_1b_2...b_j$ With proper initializations, *len*(*i, j*) can be computed as follows:

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

The time and space efficiencies of the algorithm are obviously **O(mn)** and **O(mn)**, respectively.

```
9.  ALGORITHM LCSLength(A, B)
10. for i <- 0 to m do len(i, 0) = 0
11. for j <- 1 to n do len(0, j) = 0
12. for i <- 1 to m do
13.     for j <- 1 to n do
14.         If A_i = B_j then
15.             len(I, j) = len(i-1, j-1)+1
16.             prev(i, j) = go up left
17.         else if len(i-1, j) >= len(i, j-1) then
18.             len(i, j) = len(i-1, j)
19.             prev(i, j) = go up
20.         else len(i, j) = len(i, j-1)
21.             prev(i, j) = go left
22. return len and prev
```

```
1. ALGORITHM LCSOutput( A, prev, i, j )
2. If i=0 or j=0 then return
3. If prev(i, j) = go up left, then
4.     Output → LCS(A, prev, i-1, j-1) print a_i
5. else if prev(i, j) = go up, then Output → LCS(A, prev, i-1, j)
6. else Output → LCS(A, prev, i, j-1)
7. return
```

Name: Pattarapon Buathong Student ID 62070504012

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

Then when the back tracing algorithm is running, the result is '**p r i d e n**' as shown as follows:

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

2.4. Warshall's and Floyd's Algorithms

2.4.1. Warshall's Algorithm

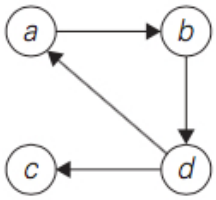
The transitive closure of a directed graph with n vertices can be defined as the **n–by-n boolean matrix T = {t_{ij}},** Constructs transitive closure T as the last matrix in the

Name: Pattarapon Buathong Student ID 62070504012

sequence of n-by-n *matrices* $R^{(0)},\ldots,R^{(k)},\ldots, R^{(n)}$ where $R^{(k)}[I,j] = 1$ if there exists a nontrivial path from i to j with only the first k vertices allowed as intermediate, and otherwise, t_{ij} is 0. **Note that $R^{(0)}$= A is adjacency matrix and $R^{(n)}$ = T is transitive closure.**

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. Define matrix r(k) whose elements are:

$$r_{ij}^{(k)} = \begin{cases} 1 & \text{there exists a directed path from } i \text{ to } j \text{ with} \\ & \text{intermediate vertices (if any) not higher than } k \\ 0 & \text{otherwise} \end{cases}$$



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

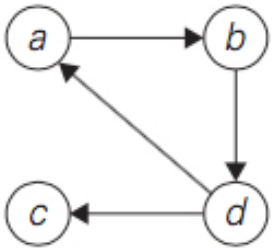
(a) Digraph. **(b) Its adjacency matrix.** **(c) Its transitive closure.**

<u>Rule for changing zeros in Warshall's algorithm:</u>

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 $R^{(k-1)}$

For example, the application of Warshall's algorithm to the digraph shown. New 1's is in bold.



$R^{(0)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

Name: Pattarapon Buathong Student ID 62070504012

$R^{(1)}$ =

a	0	0	1	0
b	1	0	1	1
c	0	0	0	0
d	0	1	0	0

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex a (note a new path from d to b); boxed row and column are used for getting $R^{(2)}$.

$R^{(2)}$ =

	a	b	c	d
a	0	0	1	0
b	1	0	1	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., a and b (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$R^{(3)}$ =

a	0	0	1	0
b	1	0	1	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., a, b, and c (no new paths); boxed row and column are used for getting $R^{(4)}$.

$R^{(4)}$ =

a	0	0	1	0
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., a, b, c, and d (note five new paths).

Name: Pattarapon Buathong Student ID 62070504012

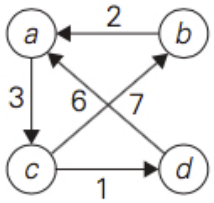
Here is pseudocode of Warshall's algorithm :

1. ***ALGORITHM*** *Warshall(A[1..n, 1..n])*
2. *//Input: The adjacency matrix A of a digraph with n vertices*
3. *//Output: The transitive closure of the digraph*
4. $R^{(0)}$*<- A*
5. *for k<-1 to n do*
6. *for i <- 1 to n do*
7. *for j<-1 to n do*
8. $R^{(k)}[i,j] \leftarrow R^{(k-1)}[i,j]$ *or* $(R^{(k-1)}[i,k]$ *and* $R^{(k-1)}[k,\ j])$
9. *Return* $R^{(n)}$

Time efficiency is Θ(n3) and Space efficiency is can be written over their predecessors (with some care), so it's Θ(n2). Note that, in software engineering, transitive closure can be used for investigating data flow and control flow dependencies as well as for inheritance testing of object-oriented software. In electronic engineering, it is used for redundancy identification and test generation for digital circuits.

2.4.2. Floyd's Algorithm

It is applicable to both undirected and directed weighted graphs provided that they do not contain a cycle of a negative length. The situation is that given a weighted connected graph, which is undirected or directed, **the all-pairs shortest paths problem** asks to find the distances—i.e., the lengths of the shortest paths—from each vertex to all other vertices. Consider a **positive weighted** connected graph W. We have to find the shortest paths from each vertex to all the others, that is distance matrix D contains the shortest path length from i^{th} vertex to j^{th} vertex.



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

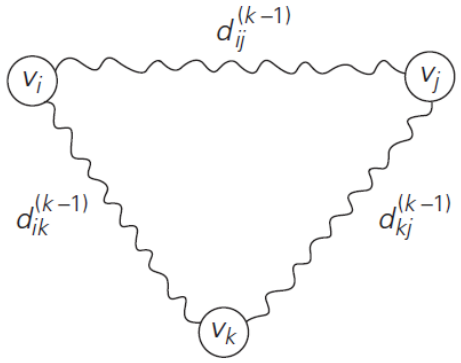
$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

(a) Digraph. **(b) Its weight matrix.** **(c) Its distance matrix.**

Name: Pattarapon Buathong Student ID 62070504012

When vi is a list of intermediate vertices each numbered not higher than k, vj .So, vi, vertices numbered ≤ k − 1, vk, vertices numbered ≤ k − 1, vj . The situation is depicted symbolically as below:

To generate the matrix, 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 as follows:

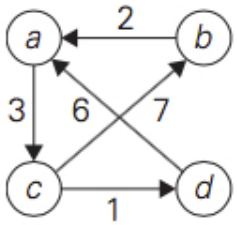


$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}, \quad k \geq 1$$

$$d_{ij}^{(0)} = w_{ij}$$

Here is pseudocode of Floyd's algorithm.

1. ***ALGORITHM*** *Floyd(W[1..n, 1..n])*
2. *//Input: The weight matrix W of a graph with no negative-length cycle*
3. *//Output: The distance matrix of the shortest paths' lengths*
4. *D <- W // is not necessary if W can be overwritten*
5. *for k <- 1 to n do*
6. *for i<-1 to n do*
7. *for j<-1 to n do*
8. *D[i, j] <- min{D[i, j], D[i, k] + D[k, j]}*
9. *return D*



$D^{(0)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

Name: Pattarapon Buathong Student ID 62070504012

$D^{(1)}$ =

a	0	∞	3	∞
b	2	0	5	∞
c	∞	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just a (note two new shortest paths from b to c and from d to c).

$D^{(2)}$ =

a	0	∞	3	∞
b	2	0	5	∞
c	9	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and b (note a new shortest path from c to a).

$D^{(3)}$ =

a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, b, and c (note four new shortest paths from a to b, from a to d, from b to d, and from d to b).

$D^{(4)}$ =

a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, b, c, and d (note a new shortest path from c to a).

2.5. Summary

Dynamic programming is a technique for solving problems with overlapping sub-problems. Typically, these sub-problems arise from a recurrence relating a solution to a given problem with solutions to its smaller sub-problems of the same type. Dynamic programming suggests solving each smaller sub-problem once and recording the results in a table from which a solution to the original problem can be then obtained.

Name: Pattarapon Buathong Student ID 62070504012

Warshall's algorithm for finding the transitive closure and Floyd's algorithm for the all-pairs shortest-paths problem are based on the idea that can be interpreted as an application of the dynamic programming technique.

3. Exercises (Group works)

3.1. Solve the instance 5, 1, 2, 10, 6 of the coin-row problems.

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

$$F(0) = 0, F(1) = C_1$$

	n =	0	1	2	3	4	5
F[0] = 0 , **F[1]** = C_1 = 5	C_n		*5*	*1*	*2*	*10*	*6*
	F(n)	**0**	**5**				
F[2] = max{1+0, 5} = 5	C_n		*5*	*1*	*2*	*10*	*6*
	F(n)	0	5	**5**			
F[3] = max{2+5, 5} = 7	C_n		*5*	*1*	*2*	*10*	*6*
	F(n)	0	5	5	**7**		
F[4] = max{10+5, 7} = 15	C_n		*5*	*1*	*2*	*10*	*6*
	F(n)	0	5	5	7	15	
F[5] = max{6+7, 15} = 15	C_n		*5*	*1*	*2*	*10*	*6*
	F(n)	0	5	5	7	15	15

Name: Pattarapon Buathong Student ID 62070504012

3.2. How would you modify the dynamic programming algorithm for the coincollecting problem if some cells on the board are inaccessible for the robot? Apply your algorithm to the board below, where the inaccessible cells are shown by X's.

How many optimal paths are there for this board?

	1	2	3	4	5	6
1		X		○		
2	○			X	○	
3		○		X	○	
4				○		○
	X	X	X		○	

Dynamic Programming Algorithm Result

	1	2	3	4	5	6
1	0	X	0	0	0	0
2	1	1	1	X	0	0
3	1	2	2	X	0	0
4	1	2	2	3	3	4
5	X	X	X	3	4	4

Name: Pattarapon Buathong Student ID 62070504012

	1	2	3	4	5	6
1	0	X	0	0	0	0
2	1	1	1	X	0	0
3	1	2	2	X	0	0
4	1	2	2	3	3	4
5	X	X	X	3	4	4

	1	2	3	4	5	6
1	0	X	0	0	0	0
2	1	1	1	X	0	0
3	1	2	2	X	0	0
4	1	2	2	3	3	4
5	X	X	X	3	4	4

	1	2	3	4	5	6
1	0	X	0	0	0	0
2	1	1	1	X	0	0
3	1	2	2	X	0	0
4	1	2	2	3	3	4
5	X	X	X	3	4	4

Name: Pattarapon Buathong Student ID 62070504012

3.3. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem: (capacity weight =6)

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

		capacity W						
	i/w	0	1	2	3	4	5	6
No item	0	0	0	0	0	0	0	0
$w_1 =$ 3 , $v_1 =$ 25	1	0	0	0	25	25	25	25
$w_2 =$ 2 , $v_2 =$ 20	2	0	0	20	25	25	45	45
$w_3 =$ 1, $v_3 =$ 15	3	0	15	20	35	40	45	60
$w_4 =$ 4, $v_4 =$ 40	4	0	15	20	35	40	55	60
$w_5 =$ 5, $v_5 =$ 50	5	0	15	20	35	40	55	65

Name: Pattarapon Buathong Student ID 62070504012

4. Lab Assignments (Ass04)

4.1. Implement the **Warshall's algorithm** that finds a transitive closure of a random graph.

The program performs the following steps:

(step 1) Accept the number of vertices V between 4 and 9 from the user.

(step 2) Generate an adjacency matrix of a random graph with V vertices.

(step 3) Run the Warshall's algorithm to find the transitive closure of the graph and print out the result.

In step 2, to determine if there should be a directed edge from vertex i to j, you can generate a random number from the range 0 and 1.

The following Python statement will generate a random number between 0 and 1:

```
import random as r
u = r.random()
if i != j and u < 0.3 :
    u = 1
else :
    u = 0
```

Submit this lab sheet include:

1) the screen snap picture of the output with V = 8 that is the directed graph generated by your Python Code,

2) .py file to LEB2.

Name: Pattarapon Buathong Student ID 62070504012

Sample Run

```
Enter the number of vertices (4 to 9):4
Generate an adjacency matrix of a random graph ..

[[0 0 1 0]
 [0 0 0 0]
 [0 0 0 1]
 [0 0 0 0]]

Transitive closure of the graph ..

[[0 0 1 1]
 [0 0 0 0]
 [0 0 0 1]
 [0 0 0 0]]
Enter the number of vertices (4 to 9):6
Generate an adjacency matrix of a random graph ..

[[0 0 1 0 0 0]
 [0 0 0 0 0 0]
 [1 1 0 1 0 0]
 [0 0 1 0 0 1]
 [1 0 1 0 0 0]
 [0 1 1 0 1 0]]

Transitive closure of the graph ..

[[1 1 1 1 1 1]
 [0 0 0 0 0 0]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]]
Enter the number of vertices (4 to 9):
```

Name: Pattarapon Buathong Student ID 62070504012

4.2. Show the step by step when V= 8

```
PS C:\Users\computer> & C:/Users/computer/AppData/Local/P
Input number of vertices between 4 to 9: 8
Generate an adjacency matrix of a random graph ...

0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
1 0 0 1 0 0 1 0
0 0 0 0 1 1 0 1
0 0 0 0 0 0 1 1
1 0 1 1 1 0 0 0
1 0 1 0 1 0 0 1
0 0 0 1 0 0 1 0
Transitive closure of the graph ...

1 0 1 1 1 1 1 1
1 0 1 1 1 1 1 1
1 0 1 1 1 1 1 1
1 0 1 1 1 1 1 1
1 0 1 1 1 1 1 1
1 0 1 1 1 1 1 1
1 0 1 1 1 1 1 1
1 0 1 1 1 1 1 1
```

Name: Pattarapon Buathong Student ID 62070504012

5. Additional Questions

1.1. What does dynamic programming have in common with divide-and-conquer?

Ans: Both algorithms are depended on dividing a problem into smaller instances of the same problem, then solve for the result.

1.2. What is a principal difference between them?

Ans: Dynamic programming works on problems which smaller instances overlap. While divide-and-conquer algorithm divides an instance into smaller instances, without any intersection. Thus, dynamic programming will store solutions explicitly to smaller instances, while divide-and-conquer algorithm do not do that.

1.3. True or false: A sequence of values in a row of the dynamic programming table for the knapsack problem is always non-decreasing?

Ans: $V[i, j-1] \leq V[i, j] \; for \; 1 \leq j \leq W$

True, since the maximum value of a subset that fits in knapsack with the capacity j-1, cannot surpass the maximum value of a subset that fits in knapsack with the capacity j.

1.4. True or false: A sequence of values in a column of the dynamic programming table for the knapsack problem is always non-decreasing?

Ans: $V[i-1, j] \leq V[i, j] \; for \; 1 \leq i \leq n$

True, since the maximum value of a subset of the first i-1 items that fits in knapsack with the capacity j, cannot surpass the maximum value of a subset of first i items that fits in knapsack with the same capacity j.

2. Reference

Levitin, A. (2012) Introduction to the Design & Analysis of Algorithms (3rd edition). Pearson. (Chapter 8.1 and 8.3)