Dynamic Programming

CS3230: Design and Analysis of Algorithms

Roger Zimmermann

National University of Singapore

Spring 2017

Introduction

- Dynamic Programming is a general algorithm design technique. “Programming” here means “planning.”
  - Invented by the American mathematician Richard Bellman in the 1950s to solve optimization problems.

Fundamentals

- Main idea
  - Solve several smaller (overlapping) subproblems
  - Record solutions in a table so that each subproblem is only solved once
  - Final state of the table will be (or contain) the solution
- Dynamic programming versus Divide-and-Conquer
  - Partition a problem into overlapping subproblems versus independent ones
  - Storing versus not storing of solutions to subproblems

Example: Fibonacci Numbers

- Recall the definition of Fibonacci numbers
  $f(0) = 0$
  $f(1) = 1$
  $f(n) = f(n-1) + f(n-2)$
- Computing the n^{th} Fibonacci number recursively (top-down):

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

Example: Fibonacci Numbers

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

$$
\begin{aligned}
&f(0) = 0\\
&f(1) = 1\\
&f(2) = 0 + 1 = 1\\
&f(3) = 1 + 1 = 2\\
&f(4) = 1 + 2 = 3\\
&f(5) = 2 + 3 = 5\\
&\ldots\\
&f(n-2) = \ldots\\
&f(n-1) = \ldots\\
&f(n) = f(n-1) + f(n-2)
\end{aligned}
$$

Example: Fibonacci Numbers

- Algorithm for computing the n^{th} Fibonacci number using bottom-up iteration:

  ALGORITHM FIB(n)

  $F[0] \leftarrow 0$

  $F[1] \leftarrow 1$

  for $i \leftarrow 2$ to n do

  $\quad F[i] \leftarrow F[i-1] + F[i-2]$

  return $F[n]$

- Uses extra space: Array $F[0..n]$

More Examples

- Computing binomial coefficients
- Warshall's algorithm for transitive closure
- Floyd's algorithms for all-pairs shortest paths
- Constructing an optimal binary search tree
- Some instances of difficult discrete optimization problems:
  - E.g.: Knapsack

Computing Binomial Coefficients

- A binomial coefficient, denoted $C(n, k)$ or $\binom{n}{k}$, is the number of combinations of k elements from an n-element set $(0 \leq k \leq n)$.
- Recurrence relation (a problem $\Rightarrow$ 2 overlapping problems):
  - $C(n, k) = C(n-1, k-1) + C(n-1, k)$, for $n > k > 0$, and
  - $C(n, 0) = C(n, n) = 1$

Computing Binomial Coefficients

- Dynamic programming solution
  - Record the values of the binomial coefficients in a table of $n+1$ rows and $k+1$ columns, numbered from 0 to n and 0 to k respectively.

	0	1	2	3	4	k
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
n	1	5	10	10	5	1

Computing Binomial Coefficients

- The first $k+1$ rows form a triangle, while the remaining rows form a rectangle.
- Time efficiency:

$$A(n,k) = \sum_{i=1}^{k}\sum_{j=1}^{i-1} 1 = \sum_{i=k+1}^{n}\sum_{j=1}^{k} 1 = \sum_{i=1}^{k}(i-1) + \sum_{i=k+1}^{n} k$$

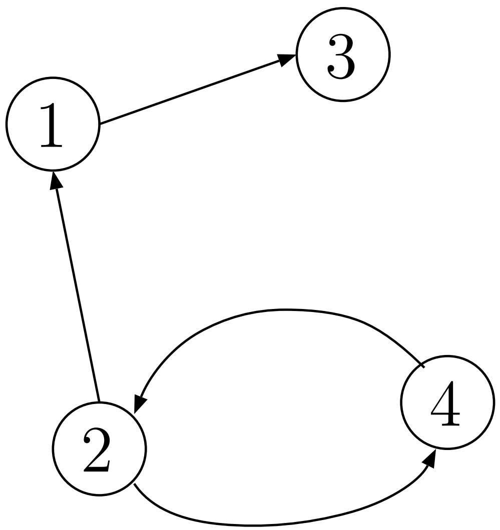
$$= \frac{(k-1)k}{2} + k(n-k) \in \Theta(nk).$$

Transitive Closure

- The transitive closure of a directed graph with n vertices can be defined as the $n \times n$ matrix T in which $t_{ij} = 1$ if there exists a *non-trivial directed path* (i.e., a directed path of a positive length) from the i^{th} to the j^{th} vertex; otherwise $t_{ij} = 0$.
- Solution: graph traversal-based algorithm and Warshall's algorithm.

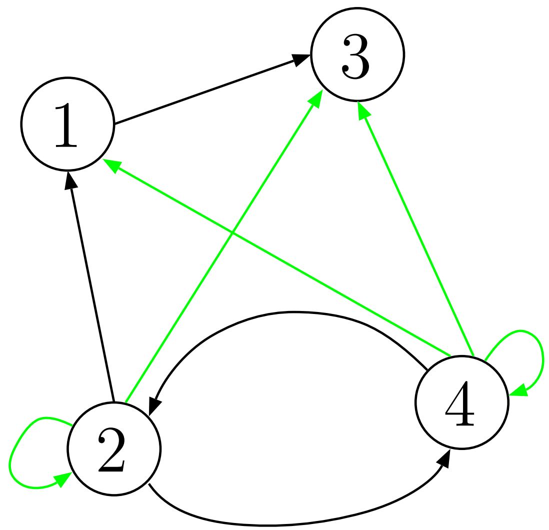
Transitive Closure

- From adjacency matrix to transitive closure:



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

Adjacency matrix



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

Transitive closure

Warshall's Algorithm

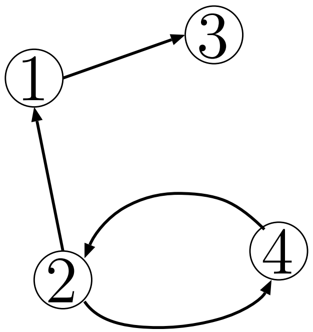
- Main idea: use a bottom-up method to construct the transitive closure of a given digraph with n vertices through a series of $n \times n$ boolean matrices: $R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}$.
- Q: how to obtain $R^{(k)}$ from $R^{(k-1)}$?
- $R^{(k)}$: $r_{ij}(k) = 1$ in $R^{(k)}$, iff
  there is an edge from i to j; or
  there is a path from i to j going through vertex 1; or
  there is a path from i to j going through vertex 1 and/or 2; or
  $\cdots$
  there is a path from i to j going through vertex 1,2, . . ., and or k

Illustration

- Rule for changing zeros in Warshall's algorithm:

$$R^{(k-1)} = \begin{array}{cc} & \begin{array}{cc} j & k \end{array} \\ \begin{array}{c} k \\ i \end{array} & \begin{bmatrix} 1 & \\ 0 & 1 \end{bmatrix} \end{array} \Rightarrow R^{(k)} = \begin{array}{cc} & \begin{array}{cc} j & k \end{array} \\ \begin{array}{c} k \\ i \end{array} & \begin{bmatrix} 1 & \\ 1 & 1 \end{bmatrix} \end{array}$$

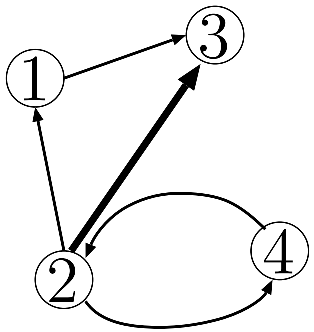
Illustration



$R^{(0)}$

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

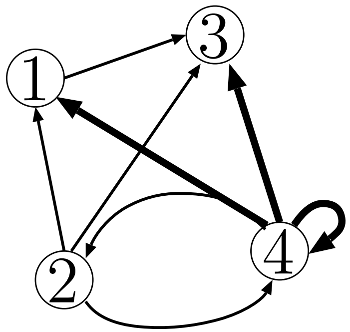
Do not allow an intermediate node



$R^{(1)}$

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

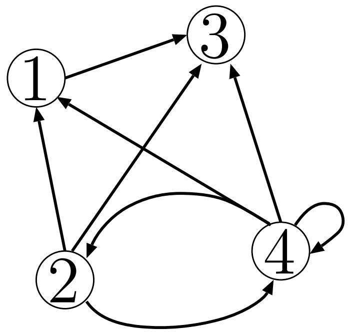
Allow 1 to be an intermediate node



$R^{(2)}$

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

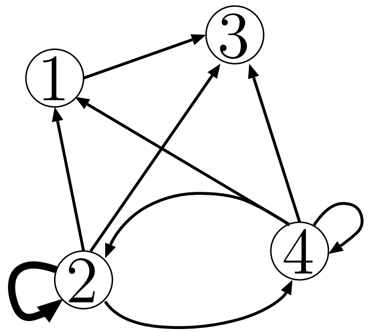
Allow 1,2 to be an intermediate node



$R^{(3)}$

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

Allow 1,2,3 to be an intermediate node



$R^{(4)}$

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

Allow 1,2,3,4 to be an intermediate node

Warshall's Algorithm

- In the k^{th} stage: to determine $R^{(k)}$ is to determine if a path exists between two vertices i, j using just vertices among $1, \ldots, k$.

$r_{ij}^{(k)} = 1 :$

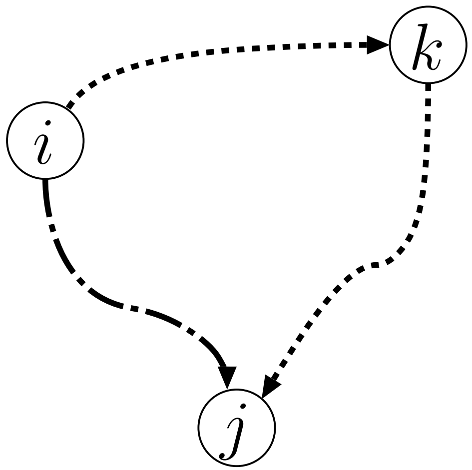
$$\begin{cases} r_{ij}^{(k-1)} = 1 & \text{path using just } 1, \ldots, k-1 \\ \text{or} & \\ (r_{ik}^{(k-1)} = 1 \text{ and } r_{kj}^{(k-1)} = 1) & \text{path from } i \text{ to } k \text{ and from } k \\ & \text{to } i \text{ using just } 1, \ldots, k-1 \end{cases}$$

Warshall's Algorithm

- Rule to determine whether $r_{ij}^{(k)}$ should be 1 in $R^{(k)}$:

a) If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.

b) If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ iff 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

- In a naive implementation it uses additional memory for all the matrices.
- Time efficiency: $\Theta(n^3)$.

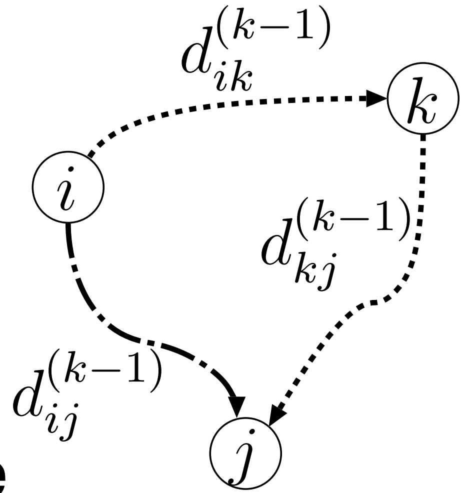
Floyd's Algorithm

- All pairs shortest paths problem: In a weighted graph, find shortest paths between every pair of vertices.
- Applicable to: undirected and directed weighted graphs; no cycles of negative length.
- Same idea as Warshall's algorithm: construct solution through a series of matrices $D^{(0)}, D^{(1)}, \ldots, D^{(N)}$.
  - $d_{ij}^{(k)} =$ length of the shortest path from i to j with each vertex numbered no higher than k.

Floyd's Algorithm

- $D^{(k)}$: allow $1, 2, \ldots, k$ to be intermediate vertices. In the k^{th} stage, determine whether the introduction of k as a new eligible intermediate vertex will bring about a shorter path from i to j.

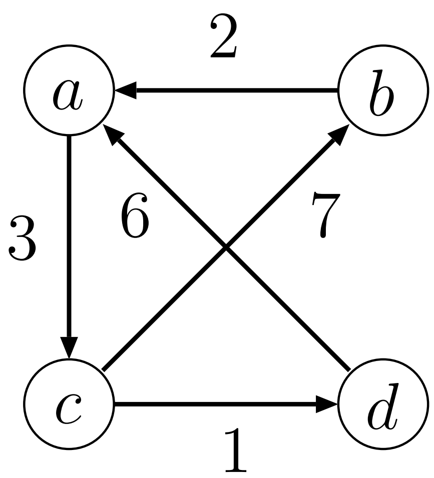
$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ for $k \geq 1$, $d_{ij}^{(0)} = w_{ij}$



k^{th} stage

Example: Floyd's Algorithm (1)

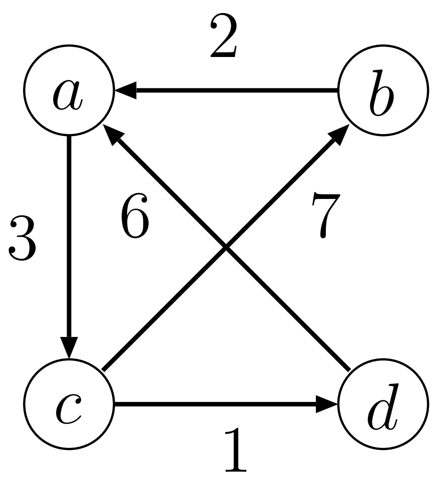
- Application of Floyd's algorithm to the graph shown. Updated elements are shown in red:



$$D^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left[\begin{array}{c|ccc} 0 & \infty & 3 & \infty \\ \hline 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{array}\right] \end{array}$$

Example: Floyd's Algorithm (2)

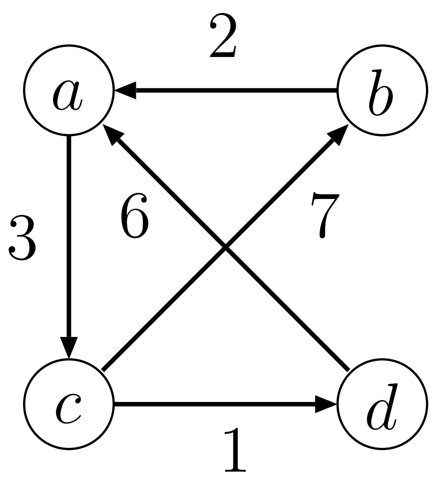
- Application of Floyd's algorithm to the graph shown. Updated elements are shown in red:



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

Example: Floyd's Algorithm (3)

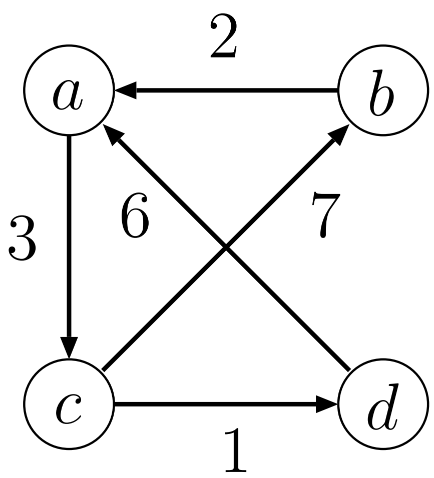
- Application of Floyd's algorithm to the graph shown. Updated elements are shown in red:



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

Example: Floyd's Algorithm (4)

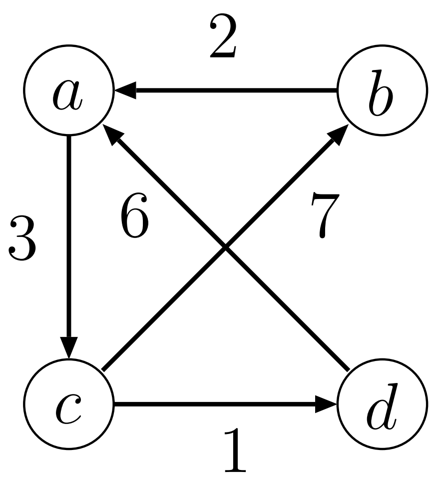
- Application of Floyd's algorithm to the graph shown. Updated elements are shown in red:



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

Example: Floyd's Algorithm (5)

- Application of Floyd's algorithm to the graph shown. Updated elements are shown in red:



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

General Comments

- The crucial step in designing a dynamic programming algorithm
  - Deriving a recurrence relating a solution to the problem's current instance with solutions of its smaller (and overlapping) subinstances.

The Knapsack Problem (1)

- The problem: Find the most valuable subset of n given items that fit into a knapsack of capacity W.
- Consider the following subproblem $P(i, j)$:
  - Find the most valuable subset of the first i items that fit into a knapsack of capacity j, where $1 \leq i \leq n$, and $1 \leq j \leq W$.
  - Let $V[i, j]$ be the value of an optimal solution to the above subproblem $P(i, j)$. Goal: $V[n, m]$.
  - The question: What is the recurrence relation that expresses a solution to this instance in terms of solutions to smaller subinstances?

The Knapsack Problem (2)

- The recurrence
  - Two possibilities for the most valuable subset for the subproblem $P(i, j)$:
    1. It does *not* include the i^{th} item: $V[i, j] = V[i-1, j]$.
    2. It includes the i^{th} item: $V[i, j] = v_i + V[i-1, j-w_i]$.

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

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$

Illustration

- Table for solving the knapsack problem by dynamic programming:

capacity j

		0	$j - w_j$	j	W
	0	0	0	0	0
	$i-1$	0	$V[i-1, j-w_j]$	$V[i-1, j]$	
w_j, v_j	i	0		$V[i, j]$	
	n	0			goal

Example

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

capacity j

	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	-	-	-	-	-
$w_2 = 1, v_2 = 10$	2	0	-	-	-	-	-
$w_3 = 3, v_3 = 20$	3	0	-	-	-	-	-
$w_4 = 2, v_4 = 15$	4	0	-	-	-	-	-

Example

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

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	-	-	-	-
$w_2 = 1, v_2 = 10$	2	0	-	-	-	-	-
$w_3 = 3, v_3 = 20$	3	0	-	-	-	-	-
$w_4 = 2, v_4 = 15$	4	0	-	-	-	-	-

Example

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

capacity j

	i	0	1	2	3	4	5
	0	$0+v_1$	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	-	-	-
$w_2 = 1, v_2 = 10$	2	0	-	-	-	-	-
$w_3 = 3, v_3 = 20$	3	0	-	-	-	-	-
$w_4 = 2, v_4 = 15$	4	0	-	-	-	-	-

Example

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

	i	capacity j 0	1	2	3	4	5
	0	0	$0+v_1$	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	-	-
$w_2 = 1, v_2 = 10$	2	0	-	-	-	-	-
$w_3 = 3, v_3 = 20$	3	0	-	-	-	-	-
$w_4 = 2, v_4 = 15$	4	0	-	-	-	-	-

Example

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

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	-	-	-	-	-
$w_3 = 3, v_3 = 20$	3	0	-	-	-	-	-
$w_4 = 2, v_4 = 15$	4	0	-	-	-	-	-

Example

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

capacity j

	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	$0+v_2$	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	-	-	-	-
$w_3 = 3, v_3 = 20$	3	0	-	-	-	-	-
$w_4 = 2, v_4 = 15$	4	0	-	-	-	-	-

Example

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

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+v_2	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	-	-	-
$w_3 = 3, v_3 = 20$	3	0	-	-	-	-	-
$w_4 = 2, v_4 = 15$	4	0	-	-	-	-	-

Example

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

		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

Example Observations

- Found maximal value $V[n,m] = V[4,5] = 37$ for capacity $W = 5$.
- Problem: not all values that we computed were really necessary to obtain the final solution.
- This happens because of bottom-up approach.

Memory Functions

- Memory functions: a combination of bottom-up and top-down method.
- Idea: solve the subproblems that are necessary and do it only once.
  - Top-down: solve common subproblems more than once.
  - Bottom-up: solve subproblems whose solutions are not necessary for solving the original problem.

MFKnapsack

ALGORITHM $MFKnapsack(i, j)$
if $V[i, j] < 0$ // if subproblem $P(i, j)$ has not been solved yet
    if $j < Weights[i]$
        $value = MFKnapsack(i - 1, j)$
    else
        $value$ = max$(MFKnapsack(i - 1, j),$
            $Values[i] + MFKnapsack(i - 1, j - Weights[i]))$
    // Store result in table
    $V[i, j] = value$
return $V[i, j]$

MF Example

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

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	-	12	22	-	22
$w_3 = 3, v_3 = 20$	3	0	-	-	22	-	32
$w_4 = 2, v_4 = 15$	4	0	-	-	-	-	37

Principle of Optimality

- An optimal solution to any instances of a problem must be made up of optimal solutions to its subinstances.
  – It underlies dynamic programming algorithms for optimization problems.
- Richard Bellman: an optimal solution to any instance of an optimization problem is composed of optimal solutions to its subinstances.

Take Away Message on Dynamic Programming

- The main idea of dynamic programming is to
  - solve several smaller (overlapping) subproblems;
  - record solutions in a table so that each subproblem is only solved once; and then
  - the final state of the table will be (or contain) the solution.
- Dynamic programming versus Divide-and-Conquer:
  - Partitioning a problem into overlapping subproblems versus independent ones.
  - Storing versus not storing of solutions to subproblems.