Module 5
Dynamic Programming

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Introduction to Dynamic Programming

- Dynamic Programming is an algorithm design technique for solving problems defined by recurrences with overlapping sub problems that exhibit the “optimal substructure” property.
- “Programming” here means “planning”
- <u>Main idea:</u>
    - 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
    - the solutions to the sub problems are optimal
    - extract solution to the initial instance from that table
    - Dynamic programming can be interpreted as a special variety of space-and-time tradeoff (store the results of smaller instances and solve a larger instance more quickly rather than repeatedly solving the smaller instances more than once).
- <u>Example:</u> Fibonacci series 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
- F(n) = F(n-1) + F(n-2), for n > 1. F(0)=0; F(1) = 1
    - F(6) = F(5) + F(4).
    - F(5) = F(4) + F(3). Note that we do not solve F(4) twice. We find F(4) only once and use that to compute F(5) and F(6).

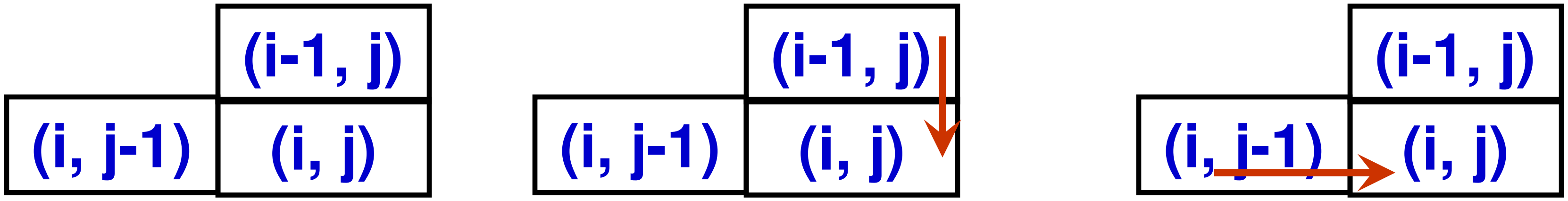
Coin-Collecting Problem

- **Problem Statement**: Several coins are placed in cells of an *n* x *m* board, no more than one coin per cell. 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. When the robot visits a cell with a coin, it always picks up that coin. Design an algorithm to find the maximum number of coins the robot can collect and a path it needs to follow to do this.

- **Solution:** 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 Fi, j-1) respectively. Of course, there are no adjacent cells to the left of the first column and above the first row. For such cells, we assume there are 0 neighbors.
- Hence, the largest number of coins the robot can bring to cell (i, j) is the maximum of the two numbers F(i-1, j) and F(i, j-1), plus the one possible coin at cell (i, j) itself.

Coin Collecting Problem

- Allowed Movements



$F(i, j) = F(i-1, j) + C_{ij}$

$F(i, j) = F(i, j-1) + C_{ij}$

$F(i, j) = \text{Max}\{F(i-1, j); F(i, j-1)\} + C_{ij}$

Coin-Collecting Problem

Recurrence

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

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

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

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)

Coin-Collecting Problem

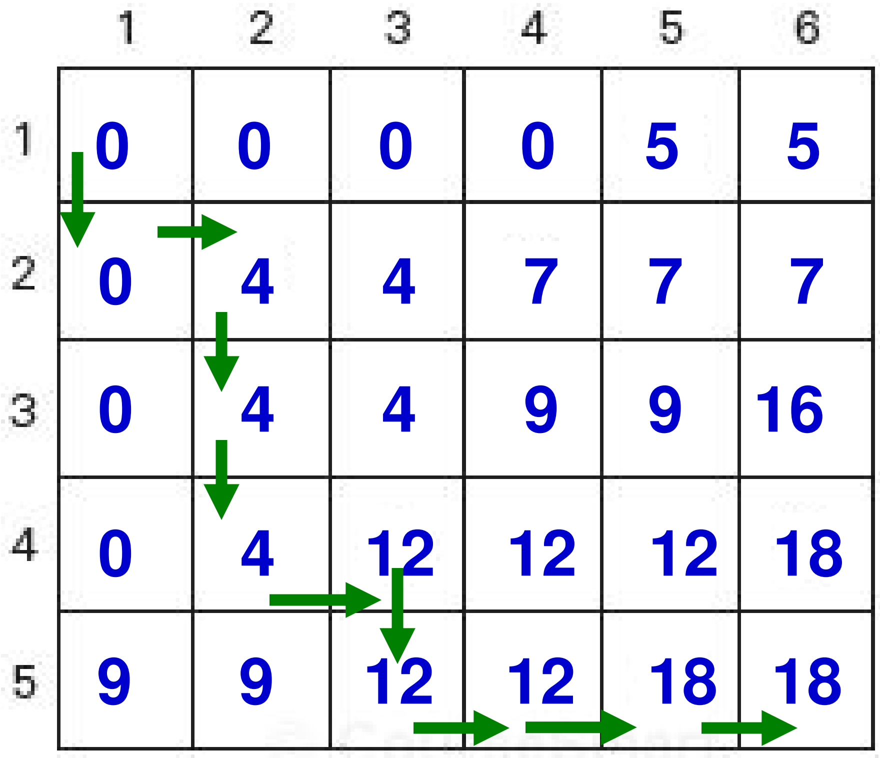
- <u>Tracing back the optimal path:</u>
- It is possible to trace the computations backwards to get an optimal path.
- If F(i-1, j) > F(i, j-1), an optimal path to cell (i, j) must come down from the adjacent cell above it;
- If F(i-1, j) < F(i, j-1), an optimal path to cell (i, j) must come from the adjacent cell on the left;
- If F(i-1, j) = F(i, j-1), it can reach cell (i, j) from either direction. Ties can be ignored by giving preference to coming from the adjacent cell above.
- If there is only one choice, i.e., either F(i-1, j) or F(i, j-1) are not available, use the other available choice.
- The optimal path can be obtained in Θ(n+m) time.

Coin-Collecting Problem: Ex-1

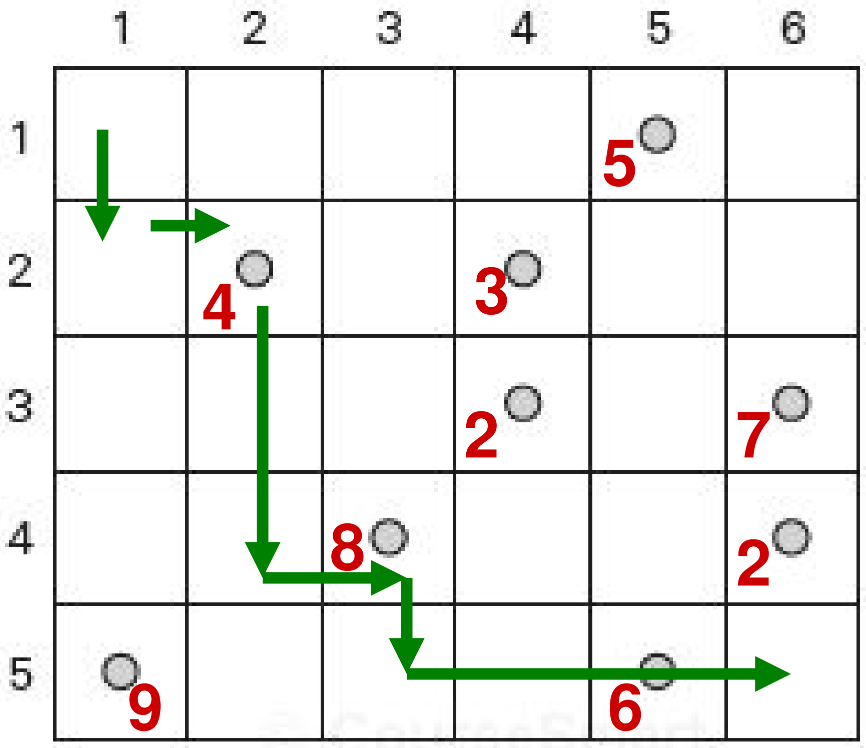
	1	2	3	4	5	6
1					5 ●	
2		4 ●		3 ●		
3				2 ●		7 ●
4			8 ●			2 ●
5	● 9				6 ●	

	1	2	3	4	5	6
1	0	0	0	0	5	5
2	0	4	4	7	7	7
3	0	4	4	9	9	16
4	0	4	12	12	12	18
5	9	9	12	12	18	18

Coin-Collecting Problem: Ex-1 (1)



	1	2	3	4	5	6
1	0	0	0	0	5	5
2	0	4	4	7	7	7
3	0	4	4	9	9	16
4	0	4	12	12	12	18
5	9	9	12	12	18	18



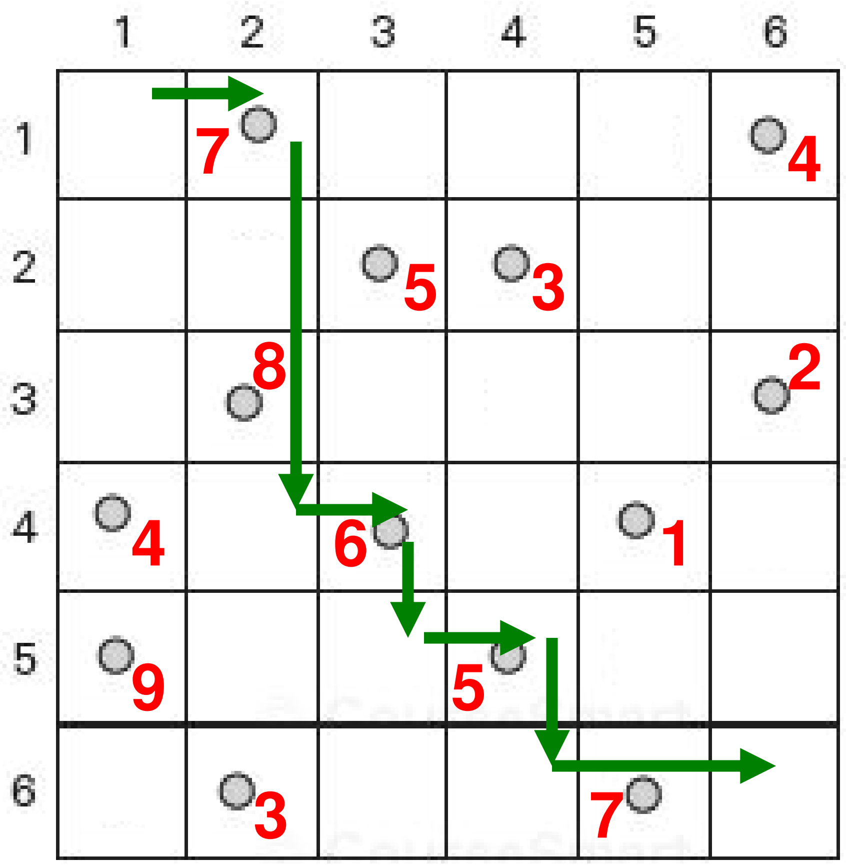
Coin-Collecting Problem: Ex-2

	1	2	3	4	5	6
1		7				4
2			5	3		
3		8				2
4	4		6		1	
5	9			5		
6		3			7	

	1	2	3	4	5	6
1	0	7	7	7	7	11
2	0	7	12	15	15	15
3	0	15	15	15	15	17
4	4	15	21	21	22	22
5	13	15	21	26	26	26
6	13	18	21	26	33	33

Coin-Collecting Problem: Ex-2 (1)

	1	2	3	4	5	6
1	0	7	7	7	7	11
2	0	7	12	15	15	15
3	0	15	15	15	15	17
4	4	15	21	21	22	22
5	13	15	21	26	26	26
6	13	18	21	26	33	33



	1	2	3	4	5	6
1		7				4
2			5	3		
3		8				2
4	4		6		1	
5	9			5		
6		3			7	

Computing a binomial coefficient

Binomial coefficients are coefficients of the binomial formula:

$$(a + b)^n = C(n,0)a^n b^0 + \ldots + C(n,k)a^{n-k}b^k + \ldots + C(n,n)a^0 b^n$$

Recurrence: $C(n,k) = C(n-1,k) + C(n-1,k-1)$ **for** $n > k > 0$

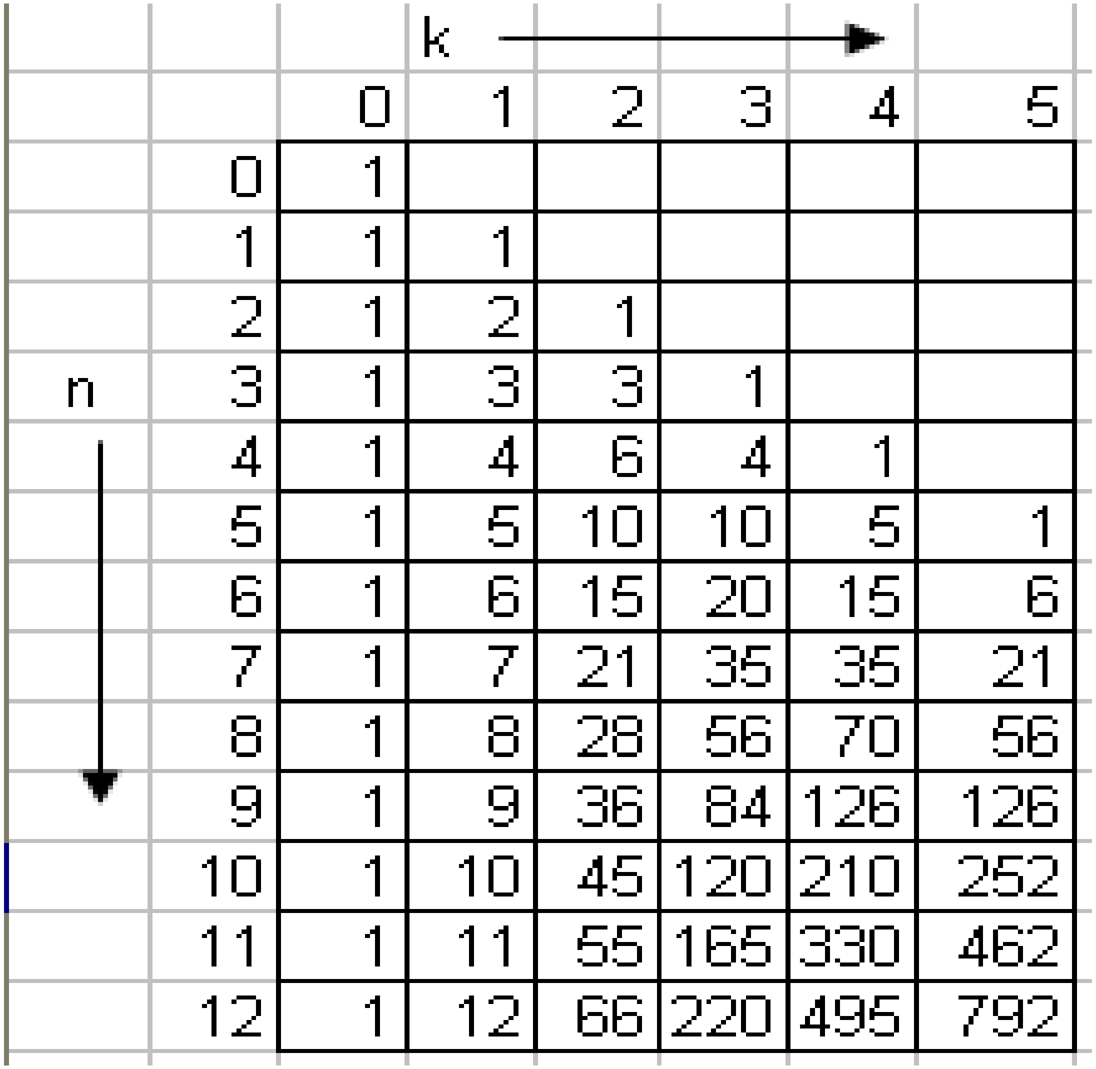
$C(n,0) = 1, \quad C(n,n) = 1$ **for** $n \geq 0$

Value of $C(n,k)$ **can be computed by filling a table:**

	0	1	2	. . .	k-1	k
0	1					
1	1	1				
.						
.						
.						
n-1					$C(n-1,k-1)$	$C(n-1,k)$
n						$C(n,k)$

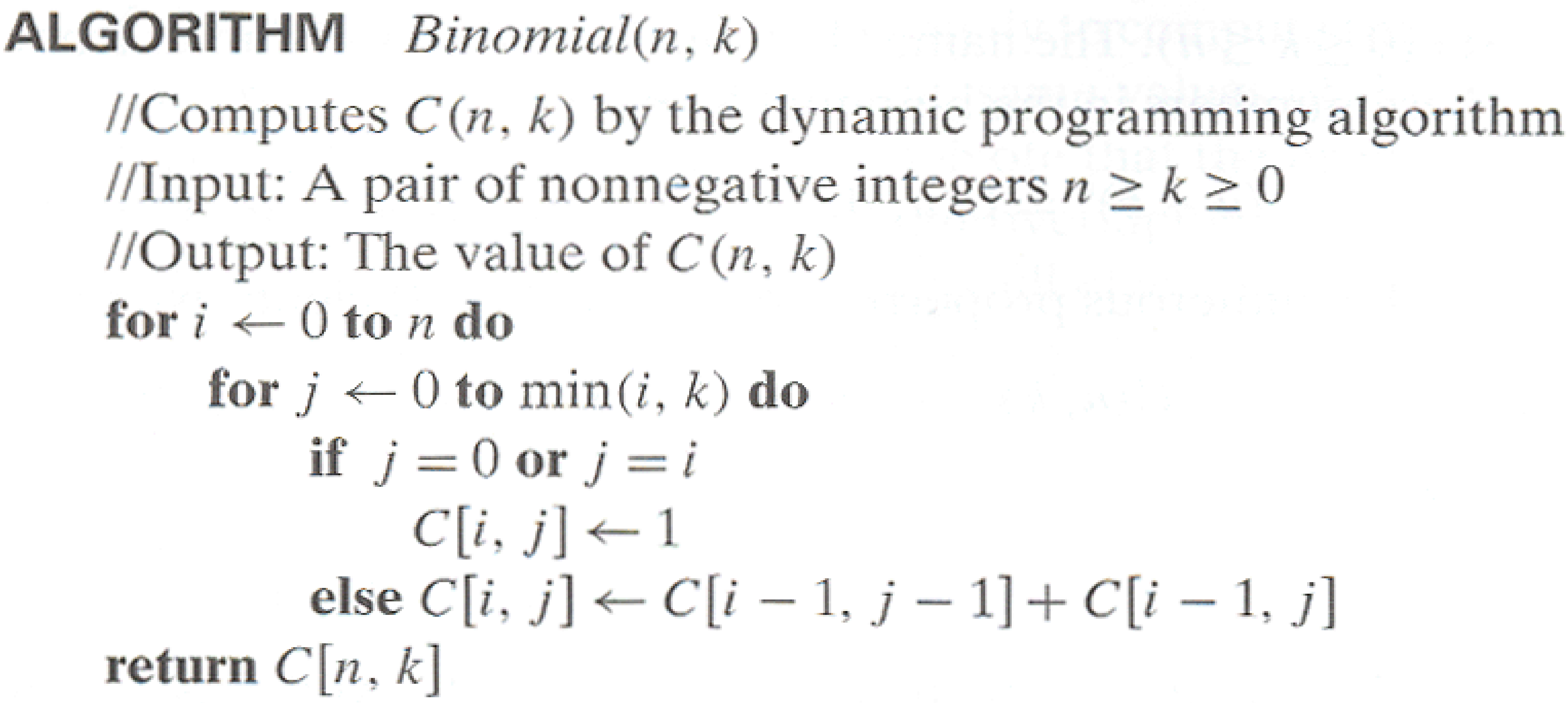
$$\boxed{nC_k = \frac{n!}{k!*(n-k)!}}$$

Computing *C*(12,*5*)



			k →				
		0	1	2	3	4	5
	0	1					
	1	1	1				
	2	1	2	1			
n	3	1	3	3	1		
↓	4	1	4	6	4	1	
	5	1	5	10	10	5	1
	6	1	6	15	20	15	6
	7	1	7	21	35	35	21
	8	1	8	28	56	70	56
	9	1	9	36	84	126	126
	10	1	10	45	120	210	252
	11	1	11	55	165	330	462
	12	1	12	66	220	495	792

Computing *C*(*n,k*): pseudocode and analysis



```
ALGORITHM  Binomial(n, k)
    //Computes $C(n, k)$ by the dynamic programming algorithm
    //Input: A pair of nonnegative integers $n \ge k \ge 0$
    //Output: The value of $C(n, k)$
    for $i \leftarrow 0$ to $n$ do
        for $j \leftarrow 0$ to $\min(i, k)$ do
            if $j = 0$ or $j = i$
                $C[i, j] \leftarrow 1$
            else $C[i, j] \leftarrow C[i-1, j-1] + C[i-1, j]$
    return $C[n, k]$
```

Time efficiency: Θ(*nk*)

Space efficiency: Θ(*nk*)

Longest Common Subsequence (LCS) Problem

LCS Problem: Overview

- The LCS problem is to find the longest subsequence common to all sequences in a set of sequences (often just two).
- Note that a subsequence is different from a substring in the sense that a subsequence need not be consecutive terms of the original sequence.
- An algorithm for the LCS problem could be used to find the longest common subsequence between the DNA strands of two organisms.
- For a given length of the two DNA strands, the longer the common subsequence, the more similar and closer (evolutionarily) are the two organisms.
- Example: X = ATGCAC Y = CAGATCCA
  - LCS(X, Y) = ATCA.

LCS Problem Idea

$X[1\ldots m] = \boxed{X_1\ X_2\ \ldots.\ X_{i-1}\ X_i}\ \ldots..\ X_m$

$Y[1\ldots n] = \boxed{Y_1\ Y_2\ \ldots\ldots\ldots\ldots\ Y_{j-1}\ Y_j}\ \ldots..\ Y_n$

LCS[i, j] = LCS[1…i][1…j]
= 1 + LCS[1…i-1][1….j-1]
= 1 + LCS[i-1, j-1]
if Xi = Yj

$X[1\ldots m] = \boxed{X_1\ X_2\ \ldots.\ X_{i-1}}\ X_i\ \ldots..\ X_m$

$Y[1\ldots n] = \boxed{Y_1\ Y_2\ \ldots\ldots\ldots\ldots\ Y_{j-1}\ Y_j}\ \ldots..\ Y_n$

LCS[i, j] = LCS[1…i-1][1….j]
= LCS[i-1, j]
if Xi ≠ Yj

OR

OR

$X[1\ldots m] = \boxed{X_1\ X_2\ \ldots.\ X_{i-1}\ X_i}\ \ldots..\ X_m$

$Y[1\ldots n] = \boxed{Y_1\ Y_2\ \ldots\ldots\ldots\ldots\ Y_{j-1}}\ Y_j\ \ldots..\ Y_n$

LCS[i, j] = LCS[1…i][1….j-1]
= LCS[i, j-1]
if Xi ≠ Yj

LCS Problem: Idea

- Let the two sequences to compare be X of length m and Y of length n. We want to find the LCS(X[1…m], Y[1…n]).
- If **X[m] = Y[n],** then we can simply discard the last character (that is common) from both the sequences and find the LCS of X[1…m-1] and Y[1…n-1], such that

  LCS(X[1…m], Y[1…n]) = LCS(X[1…m-1], Y[1…n-1]) + 1.
- If **X[m] ≠ Y[n],** then the longest common subsequence of the two sequences can be at most either X[m] or Y[n]; but not both. Hence, we can say that:

  LCS(X[1…m], Y[1…n])

  = Max {LCS(X[1…m-1], Y[1…n]), LCS(X[1…m], Y[1…n-1])}

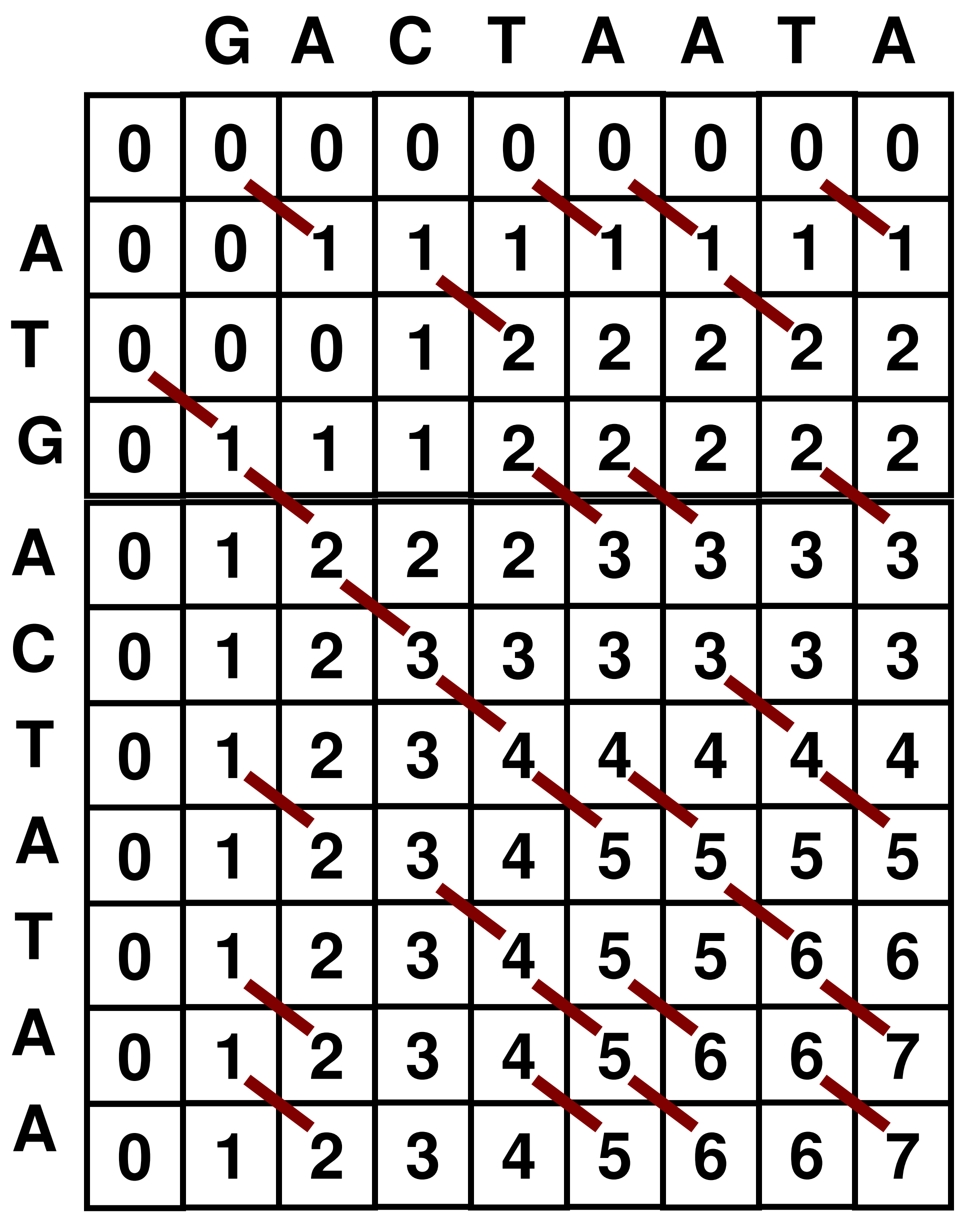
Dynamic Programming Formulation

Define: LCS[*i*][*j*] = Length of the LCS of sequence X[1…i] and Y[1…j]
Thus, LCS[i][0] = 0 for all i LCS[0][j] = 0 for all j
The goal is to find LCS[m][n]

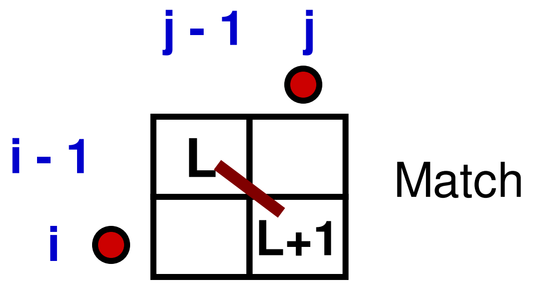
$$LCS[i][j] = \begin{cases} LCS[i-1][j-1]+1 & X[i]=Y[j] \\ Max\{LCS[i][j-1], LCS[i-1][j]\} & X[i]!=Y[j] \end{cases}$$

LCS Example 1 (1)

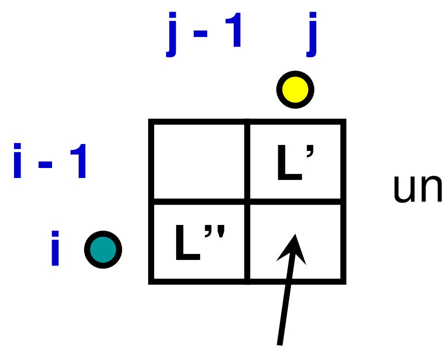
X = ATGACTATAA
Y = GACTAATA



		G	A	C	T	A	A	T	A
	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
T	0	0	0	1	2	2	2	2	2
G	0	1	1	1	2	2	2	2	2
A	0	1	2	2	2	3	3	3	3
C	0	1	2	3	3	3	3	3	3
T	0	1	2	3	4	4	4	4	4
A	0	1	2	3	4	5	5	5	5
T	0	1	2	3	4	5	5	6	6
A	0	1	2	3	4	5	6	6	7
A	0	1	2	3	4	5	6	6	7



Match

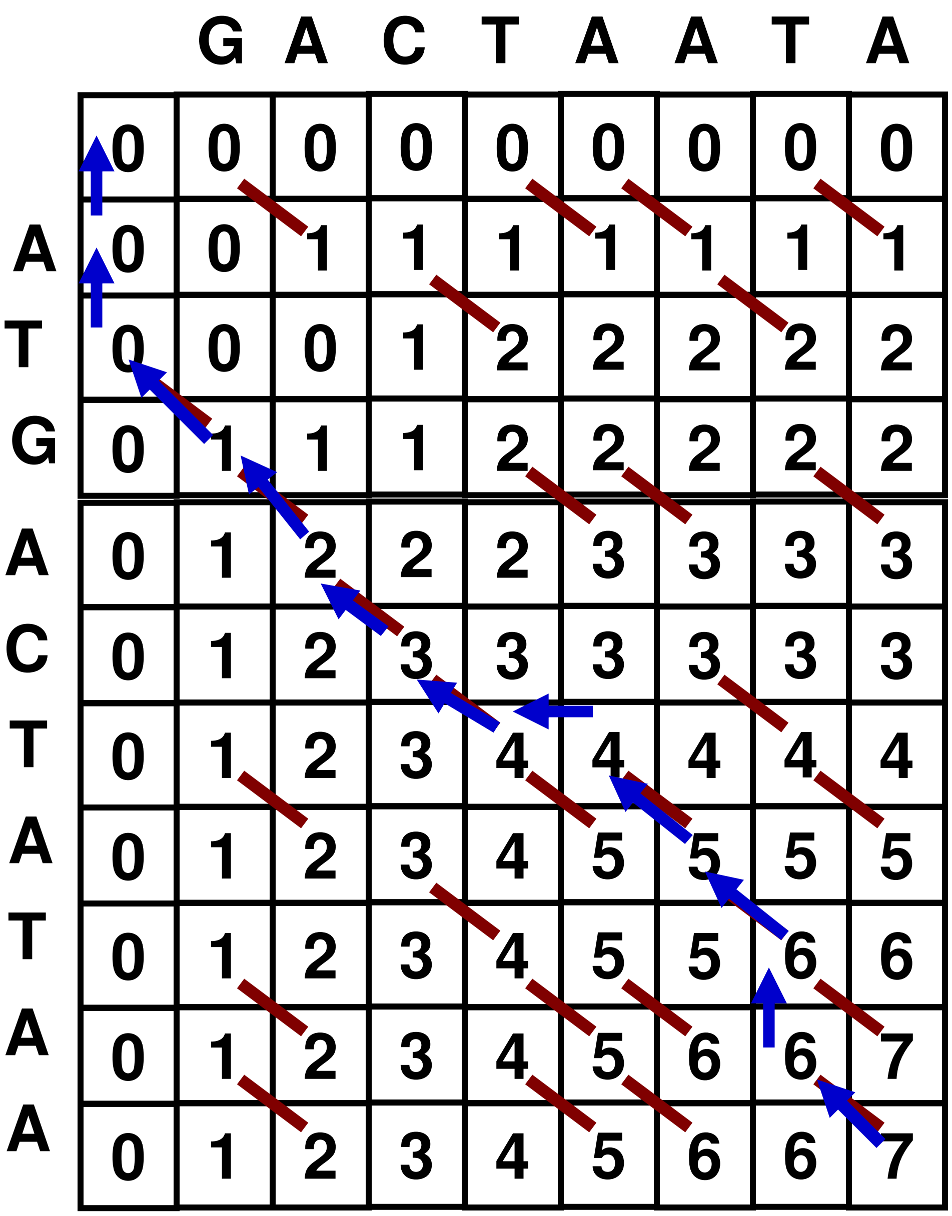


unmatch

Max(L’, L’’)

X = ATGACTATAA
Y = GACTAATA

LCS Example 1 (2)



		G	A	C	T	A	A	T	A
	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
T	0	0	0	1	2	2	2	2	2
G	0	1	1	1	2	2	2	2	2
A	0	1	2	2	2	3	3	3	3
C	0	1	2	3	3	3	3	3	3
T	0	1	2	3	4	4	4	4	4
A	0	1	2	3	4	5	5	5	5
T	0	1	2	3	4	5	5	6	6
A	0	1	2	3	4	5	6	6	7
A	0	1	2	3	4	5	6	6	7

Sequence Alignment

A T G A C T - A T A A
- - G A C T A A T - A

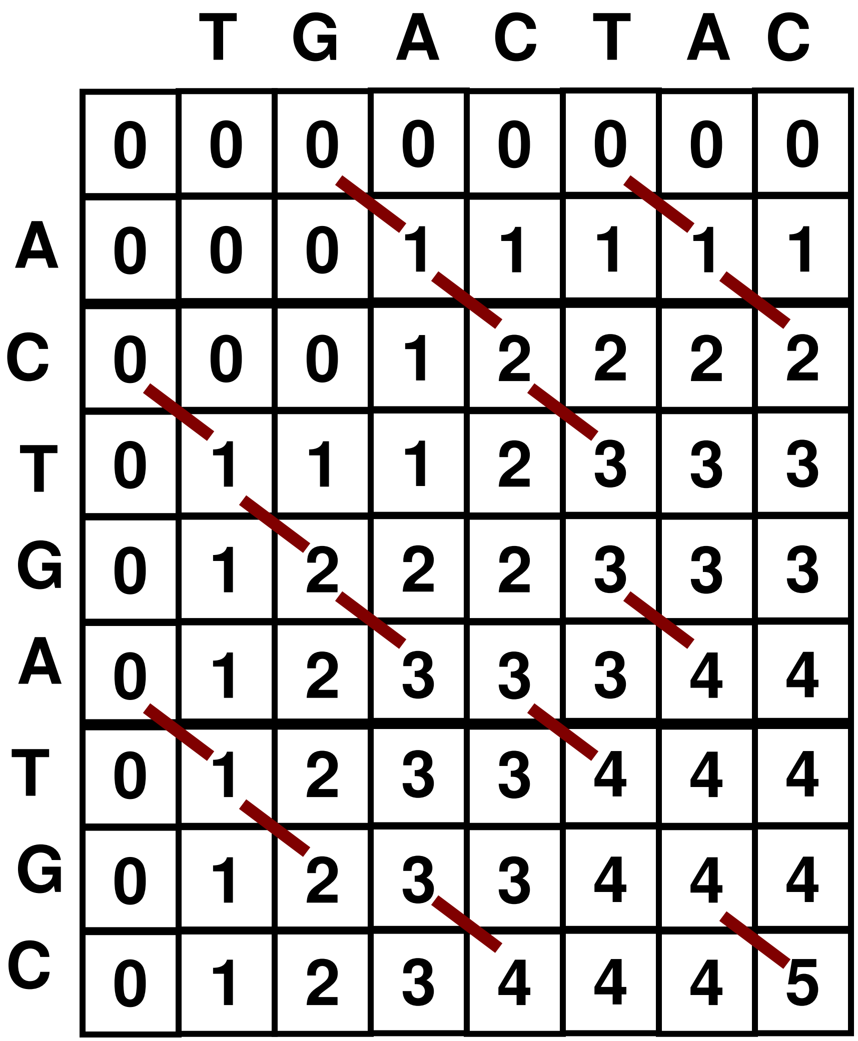
LCS: G A C T A T A

Ties are broken by going up

When you skip a symbol, it is aligned with a blank -.

X = TGACTAC
Y = ACTGATGC

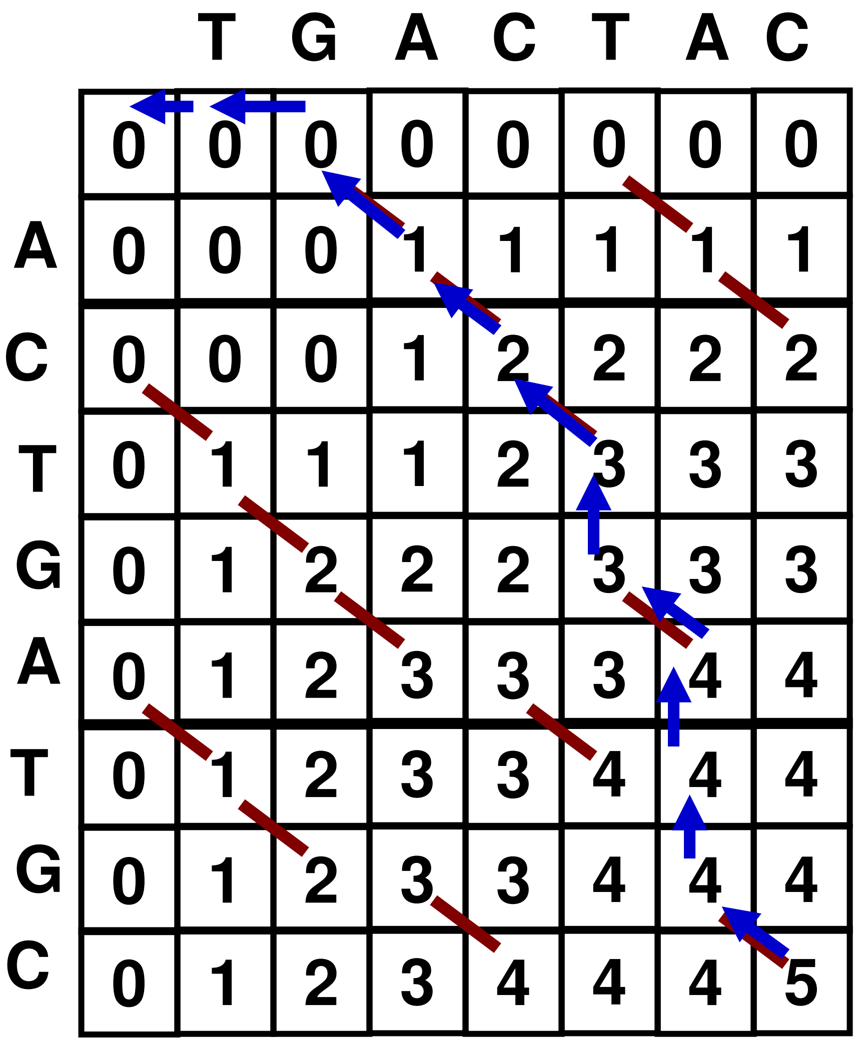
LCS Example 2 (1)



		T	G	A	C	T	A	C
	0	0	0	0	0	0	0	0
A	0	0	0	1	1	1	1	1
C	0	0	0	1	2	2	2	2
T	0	1	1	1	2	3	3	3
G	0	1	2	2	2	3	3	3
A	0	1	2	3	3	3	4	4
T	0	1	2	3	3	4	4	4
G	0	1	2	3	3	4	4	4
C	0	1	2	3	4	4	4	5

X = TGACTAC
Y = ACTGATGC

LCS Example 2 (2)

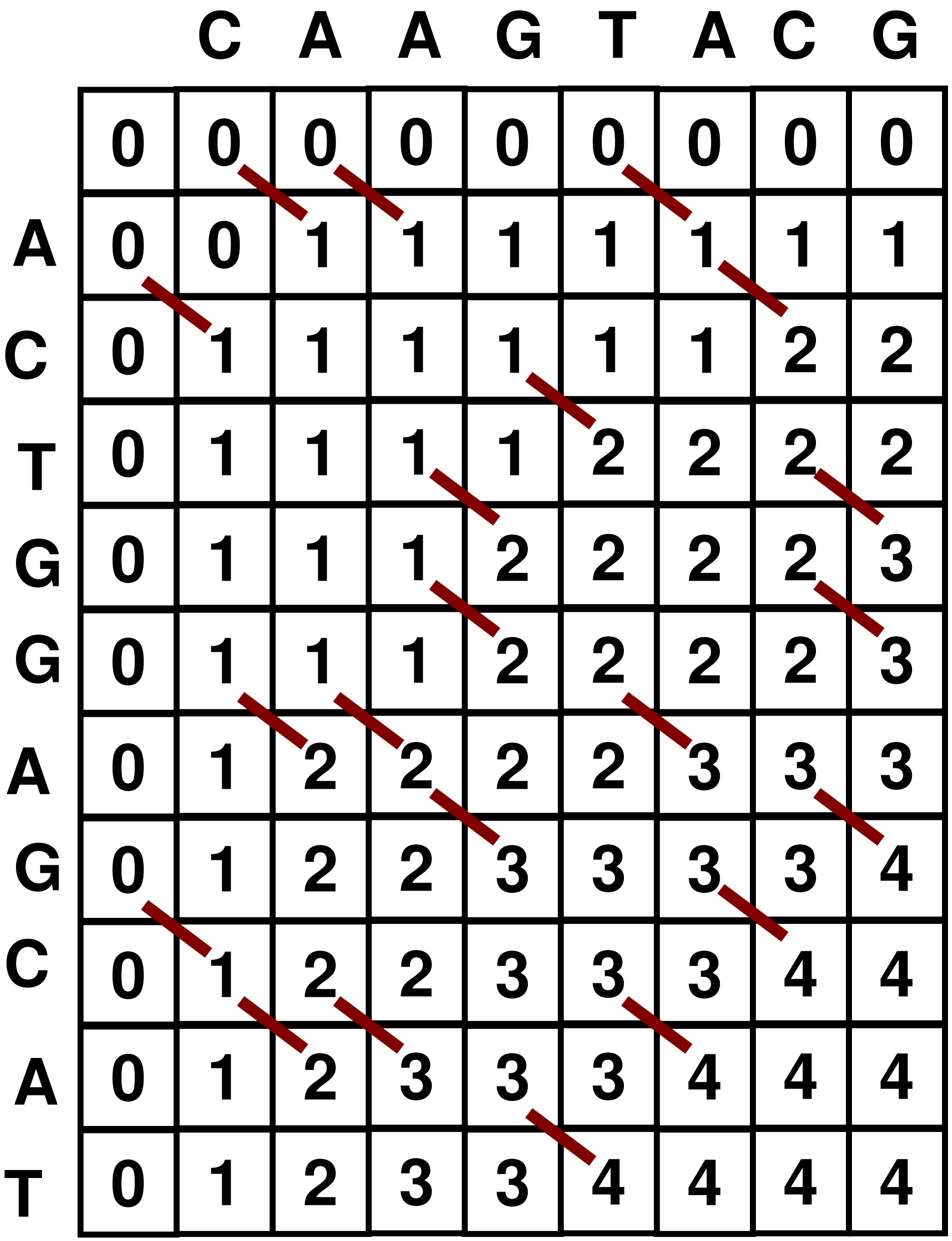


		T	G	A	C	T	A	C
	0	0	0	0	0	0	0	0
A	0	0	0	1	1	1	1	1
C	0	0	0	1	2	2	2	2
T	0	1	1	1	2	3	3	3
G	0	1	2	2	2	3	3	3
A	0	1	2	3	3	3	4	4
T	0	1	2	3	3	4	4	4
G	0	1	2	3	3	4	4	4
C	0	1	2	3	4	4	4	5

```
T G A C T - A - - C
- - A C T G A T G C
```

LCS: A C T A C

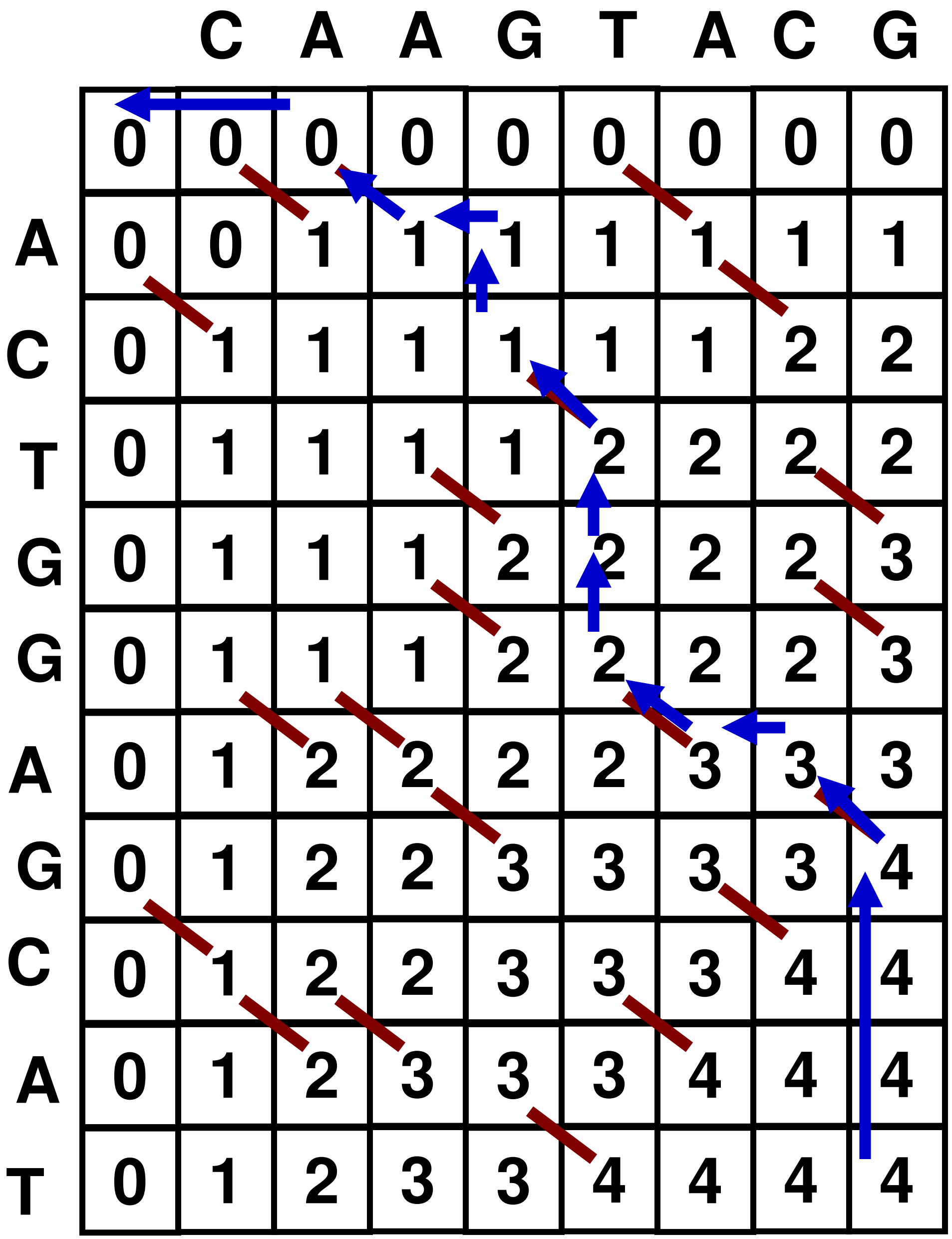
LCS Example 3 (1)



		C	A	A	G	T	A	C	G
	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
C	0	1	1	1	1	1	1	2	2
T	0	1	1	1	1	2	2	2	2
G	0	1	1	1	2	2	2	2	3
G	0	1	1	1	2	2	2	2	3
A	0	1	2	2	2	2	3	3	3
G	0	1	2	2	3	3	3	3	4
C	0	1	2	2	3	3	3	4	4
A	0	1	2	3	3	3	4	4	4
T	0	1	2	3	3	4	4	4	4

X = CAAGTACG
Y = ACTGGAGCAT

LCS Example 3 (2)



		C	A	A	G	T	A	C	G
	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
C	0	1	1	1	1	1	1	2	2
T	0	1	1	1	1	2	2	2	2
G	0	1	1	1	2	2	2	2	3
G	0	1	1	1	2	2	2	2	3
A	0	1	2	2	2	2	3	3	3
G	0	1	2	2	3	3	3	3	4
C	0	1	2	2	3	3	3	4	4
A	0	1	2	3	3	3	4	4	4
T	0	1	2	3	3	4	4	4	4

X = CAAGTACG
Y = ACTGGAGCAT

```
C A A G - T - - A C G - - -
- - A - C T G G A - G C A T
```

LCS: A T A G

Coin Change Problem

- Given a set of coin denominations CD[1…N] and a value S, we want to determine the optimal (minimum) number of coins that can be used so that the coin values add up to S.
- Assume there is an infinite supply of the coins for each value.

- Unlike the greedy approach, the dynamic programming solution will work for all coin denominations.
- Example: CD[1…3] = {3, 1, 4}; S = 6
  - Greedy approach will give a solution of picking 3 coins (values: 4, 1, 1)
  - Dynamic programming approach will give a solution of picking 2 coins (values: 3, 3)

Coin Change Problem

Recurrence Relation

- Given S and CD[1…N]
- Let $MNC^j[V]$ be the minimum number of coins that need to be picked up by considering coins at index 1…j so that the coins picked up add to a value of V, where $0 \leq V \leq S$.
- Let $LCP^j[V] = CD[j]$ if the jth coin needs to be picked up so that the total value of the coins picked is V.

LCP – Last Coin value Picked

$$MNC^j[V] = \text{Minimum} \begin{cases} MNC^{j-1}[V] \\ 1 + MNC^j[\,V - CD[j]] \end{cases}$$

// if coin index j should not
// be picked up for an optimal
// solution to add up to V

// if picking the coin at index j will reduce the
// number of coins from what is known currently

$0 \leq V \leq S$
$1 \leq j \leq N$
$V \geq CD[j]$

$MNC^j[X] = \infty$ for $X > 0$ and $j = 0$ (i.e., no coin is available for pickup)

$MNC^j[X] = 0$ for $X = 0$ and any j (i.e., value of the coins to add up to is 0)

Time complexity: $\Theta(N*S)$; **Space Complexity: $\Theta(S)$**

Iteration j-1

v →	0	1	2	3	4	5	6	....			S
MNC	0	…	…	…	…	Y	…	…	…	…	…
LCP	-	…	…	…	…	Z	…	…	…	…	…

Iteration j

Consider a value V (say, V = 5) that is greater than or equal to CD[j] (say, CD[j] = 3).

Look CD[j] = 3 cells backwards (i.e., V – CD[j])

←

v →	0	1	2	3	4	5	6	....			S
MNC	0	…	P	…	…	R	…	…	…	…	…
LCP	-	…	…	…	…	Q	…	…	…	…	…

R = Min (Y , 1 + P)

Q = Z if R = Y
Q = CD[j] if R = 1 + P

CD Array

j	CD[j]
1	3
2	1
3	4

Coin Change Problem

Example 1 (Initialization)

MNC Table

Coins/Denomination		Values, V						
j	CD[j]	0	1	2	3	4	5	6
0	-	0	∞	∞	∞	∞	∞	∞
1	3	0						
2	1	0						
3	4	0						

LCP Table

Coins/Denomination		Values, V						
j	CD[j]	0	1	2	3	4	5	6
0	-	-	-	-	-	-	-	-
1	3	-						
2	1	-						
3	4	-						

CD Array

j	CD[j]
1	3
2	1
3	4

Coin Change Problem

Example 1 (Iteration 1)

MNC Table

Coins/Denomination		Values, V						
j	CD[j]	0	1	2	3	4	5	6
0	-	0	∞	∞	∞	∞	∞	∞
1	3	0	∞	∞	1	∞	∞	2
2	1	0						
3	4	0						

LCP Table

Coins/Denomination		Values, V						
j	CD[j]	0	1	2	3	4	5	6
0	-	-	-	-	-	-	-	-
1	3	-	-	-	3	-	-	3
2	1	-						
3	4	-						

CD Array

j	CD[j]
1	3
2	1
3	4

Coin Change Problem

Example 1 (Iteration 2)

MNC Table

Coins/Denomination		Values, V						
j	CD[j]	0	1	2	3	4	5	6
0	-	0						
1	3	0	∞	∞	1	∞	∞	2
2	1	0	1	2	1	2	3	2
3	4	0						

LCP Table

Coins/Denomination		Values, V						
j	CD[j]	0	1	2	3	4	5	6
0	-	-						
1	3	-	-	-	3	-	-	3
2	1	-	1	1	3	1	1	3
3	4	-						

CD Array

j	CD[j]
1	3
2	1
3	4

Coin Change Problem

Example 1 (Iteration 3)

MNC Table

Coins/Denomination		Values, V						
j	CD[j]	0	1	2	3	4	5	6
0	-	0						
1	3	0						
2	1	0	1	2	1	2	3	2
3	4	0	1	2	1	1	2	2

LCP Table

Coins/Denomination		Values, V						
j	CD[j]	0	1	2	3	4	5	6
0	-	-						
1	3	-						
2	1	-	1	1	3	1	1	3
3	4	-	1	1	3	4	4	3

Coin Change Problem

Example 1 (Final Tables)

CD Array

j	CD[j]
1	3
2	1
3	4

MNC Table

Coins/Denomination		Values, V						
j	CD[j]	0	1	2	3	4	5	6
0	-	0	∞	∞	∞	∞	∞	∞
1	3	0	∞	∞	1	∞	∞	2
2	1	0	1	2	1	2	3	2
3	4	0	1	2	1	1	2	2

LCP Table

Coins/Denomination		Values, V						
j	CD[j]	0	1	2	3	4	5	6
0	-	-	-	-	-	-	-	-
1	3	-	-	-	3	-	-	3
2	1	-	1	1	3	1	1	3
3	4	-	1	1	3	4	4	3

Tracing the solution (V = 6): Coins picked: 3, 3

MNC[6] = 2 coins to be picked up for Value = 6

LCP[6] = 3; So, pick coin of value 3 and go to LCP[6-3] = LCP[3]

LCP[3] = 3; So, pick coin of value 3 and go to LCP[3-3] = LCP[0] = - // Done

Coin Change Problem Example 2

CD Array

j	1	2	3	4
CD[j]	5	1	2	4

Initialization

Let S = 17

MNC

Coins/ Denomination		Values, V																	
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	5	0																	
2	1	0																	
3	2	0																	
4	4	0																	

LCP

Coins/ Denomination		Values, V																	
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	5	-																	
2	1	-																	
3	2	-																	
4	4	-																	

Coin Change Problem Example 2

Iteration 1

CD Array

j	1	2	3	4
CD[j]	5	1	2	4

Let S = 17

MNC

Coins/ Denomination		Values, V																	
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	5	0	∞	∞	∞	∞	1	∞	∞	∞	∞	2	∞	∞	∞	∞	3	∞	∞
2	1	0																	
3	2	0																	
4	4	0																	

LCP

Coins/ Denomination		Values, V																	
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	5	-	-	-	-	-	5	-	-	-	-	5	-	-	-	-	5	-	-
2	1	-																	
3	2	-																	
4	4	-																	

Coin Change Problem Example 2

CD Array

j	1	2	3	4
CD[j]	5	1	2	4

Iteration 2

Let S = 17

MNC

Coins/ Denomination		Values, V																	
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	0																	
1	5	0	∞	∞	∞	∞	1	∞	∞	∞	∞	2	∞	∞	∞	∞	3	∞	∞
2	1	0	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5
3	2	0																	
4	4	0																	

LCP

Coins/ Denomination		Values, V																	
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	-																	
1	5	-	-	-	-	-	5	-	-	-	-	5	-	-	-	-	5	-	-
2	1	-	1	1	1	1	5	1	1	1	1	5	1	1	1	1	5	1	1
3	2	-																	
4	4	-																	

Coin Change Problem Example 2

CD Array

j	1	2	3	4
CD[j]	5	1	2	4

Iteration 3

Let S = 17

MNC

Coins/ Denomination		Values, V																	
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	0																	
1	5	0																	
2	1	0	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5
3	2	0	1	1	2	3	1	2	2	3	4	2	3	3	4	5	3	4	4
4	4	0																	

LCP

Coins/ Denomination		Values, V																	
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	-																	
1	5	-																	
2	1	-	1	1	1	1	5	1	1	1	1	5	1	1	1	1	5	1	1
3	2	-	1	2	2	2	5	1	2	2	2	5	1	2	2	2	5	1	2
4	4	-																	

Coin Change Problem Example 2

CD Array

j	1	2	3	4
CD[j]	5	1	2	4

Iteration 4

Let S = 17

MNC

Coins/ Denomination		Values, V																	
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	0																	
1	5	0																	
2	1	0																	
3	2	0	1	1	2	3	1	2	2	3	4	2	3	3	4	5	3	4	4
4	4	0	1	1	2	1	1	2	2	2	2	2	3	3	3	3	3	4	4

LCP

Coins/ Denomination		Values, V																	
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	-																	
1	5	-																	
2	1	-																	
3	2	-	1	2	2	2	5	1	2	2	2	5	1	2	2	2	5	1	2
4	4	-	1	2	2	4	5	1	2	4	4	5	1	2	4	4	5	1	2

Coin Change Problem Example 2

CD Array

j	1	2	3	4
CD[j]	5	1	2	4

Let S = 17

Final Tables

MNC

Coins/ Denomination		Values, V																	
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	5	0	∞	∞	∞	∞	1	∞	∞	∞	∞	2	∞	∞	∞	∞	3	∞	∞
2	1	0	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5
3	2	0	1	1	2	3	1	2	2	3	4	2	3	3	4	5	3	4	4
4	4	0	1	1	2	1	1	2	2	2	2	2	3	3	3	3	3	4	4

LCP

Coins/ Denomination		Values, V																	
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	5	-	-	-	-	-	5	-	-	-	-	5	-	-	-	-	5	-	-
2	1	-	1	1	1	1	5	1	1	1	1	5	1	1	1	1	5	1	1
3	2	-	1	2	2	2	5	1	2	2	2	5	1	2	2	2	5	1	2
4	4	-	1	2	2	4	5	1	2	4	4	5	1	2	4	4	5	1	2

CD Array

j	1	2	3	4
CD[j]	5	1	2	4

Final Table

V →	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
MNC	0	1	1	2	1	1	2	2	2	2	2	3	3	3	3	3	4	4
LCP	-	1	2	2	4	5	1	2	4	4	5	1	2	4	4	5	1	2

Tracing the solution (V = 17):

MNC[17] = 4 coins to be picked up for Value = 17

LCP[17] = 2; So, pick coin of value 2 and go to LCP[17 – 2] = LCP[15]

LCP[13] = 5; So, pick coin of value 5 and go to LCP[15 – 5] = LCP[10]

LCP[10] = 5; So, pick coin of value 5 and go to LCP[10 – 5] = LCP[5]

LCP[5] = 5; So, pick coin of value 5 and go to LCP[5 – 5] = LCP[0] = - // Done!!

Coins picked for Value = 17 are: 2, 5, 5, 5

Tracing the solution (V = 12):

MNC[12] = 3 coins to be picked up for Value = 12

LCP[12] = 2; So, pick coin of value 2 and go to LCP[12 – 2] = LCP[10]

LCP[10] = 5; So, pick coin of value 5 and go to LCP[10 – 5] = LCP[5]

LCP[5] = 5; So, pick coin of value 5 and go to LCP[5 – 5] = LCP[0] = - // Done!!

Coins picked for Value = 12 are: 2, 5, 5

Coin Change Problem
Example 3

Initialization

CD Array

j	1	2	3	4
CD[j]	5	6	1	9

Let S = 11

MNC

Coins/ Denomination		Values, V											
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11
0	-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	5	0											
2	6	0											
3	1	0											
4	9	0											

LCP

Coins/ Denomination		Values, V											
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11
0	-	-	-	-	-	-	-	-	-	-	-	-	-
1	5	-											
2	6	-											
3	1	-											
4	9	-											

Coin Change Problem Example 3

CD Array

j	1	2	3	4
CD[j]	5	6	1	9

Let S = 11

Iteration 1

MNC

Coins/ Denomination		Values, V											
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11
0	-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	5	0	∞	∞	∞	∞	1	∞	∞	∞	∞	2	∞
2	6	0											
3	1	0											
4	9	0											

LCP

Coins/ Denomination		Values, V											
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11
0	-	-	-	-	-	-	-	-	-	-	-	-	-
1	5	-	-	-	-	-	5	-	-	-	-	5	-
2	6	-											
3	1	-											
4	9	-											

Coin Change Problem Example 3

CD Array

j	1	2	3	4
CD[j]	5	6	1	9

Let S = 11

Iteration 2

MNC

Coins/ Denomination		Values, V											
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11
0	-	0											
1	5	0	∞	∞	∞	∞	1	∞	∞	∞	∞	2	∞
2	6	0	∞	∞	∞	∞	1	1	∞	∞	∞	2	2
3	1	0											
4	9	0											

LCP

Coins/ Denomination		Values, V											
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11
0	-	-											
1	5	-	-	-	-	-	5	-	-	-	-	5	-
2	6	-	-	-	-	-	5	6	-	-	-	5	6
3	1	-											
4	9	-											

Coin Change Problem
Example 3

CD Array

j	1	2	3	4
CD[j]	5	6	1	9

Let S = 11

Iteration 3

MNC

Coins/ Denomination		Values, V											
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11
0	-	0											
1	5	0											
2	6	0	∞	∞	∞	∞	1	1	∞	∞	∞	2	2
3	1	0	1	2	3	4	1	1	2	3	4	2	2
4	9	0											

LCP

Coins/ Denomination		Values, V											
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11
0	-	-											
1	5	-											
2	6	-	-	-	-	-	5	6	-	-	-	5	6
3	1	-	1	1	1	1	5	6	1	1	1	5	6
4	9	-											

Coin Change Problem
Example 3

CD Array

j	1	2	3	4
CD[j]	5	6	1	9

Let S = 11

Iteration 4

MNC

Coins/ Denomination		Values, V											
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11
0	-	0											
1	5	0											
2	6	0											
3	1	0	1	2	3	4	1	1	2	3	4	2	2
4	9	0	1	2	3	4	1	1	2	3	1	2	2

LCP

Coins/ Denomination		Values, V											
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11
0	-	-											
1	5	-											
2	6	-											
3	1	-	1	1	1	1	5	6	1	1	1	5	6
4	9	-	1	1	1	1	5	6	1	1	9	5	6

Final Tables

MNC

Coins/ Denomination		Values, V											
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11
0	-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	5	0	∞	∞	∞	∞	1	∞	∞	∞	∞	2	∞
2	6	0	∞	∞	∞	∞	1	1	∞	∞	∞	2	2
3	1	0	1	2	3	4	1	1	2	3	4	2	2
4	9	0	1	2	3	4	1	1	2	3	1	2	2

LCP

Coins/ Denomination		Values, V											
j	CD[j]	0	1	2	3	4	5	6	7	8	9	10	11
0	-	-	-	-	-	-	-	-	-	-	-	-	-
1	5	-	-	-	-	-	5	-	-	-	-	5	-
2	6	-	-	-	-	-	5	6	-	-	-	5	6
3	1	-	1	1	1	1	5	6	1	1	1	5	6
4	9	-	1	1	1	1	5	6	1	1	9	5	6

CD Array

j	1	2	3	4
CD[j]	5	6	1	9

Let S = 11

Tracing the Solution for V = 11

LCP[11] = 6; Pick 6
LCP[11 – 6] = LCP[5]
LCP[5] = 5; Pick 5
LCP[5 – 5] = - Done!!

Coins Picked for S = 11 are: 6, 5 (2 coins – optimal)
Greedy approach would have given a solution of 3 coins (9, 1, 1)

Integer (0-1) Knapsack Problem

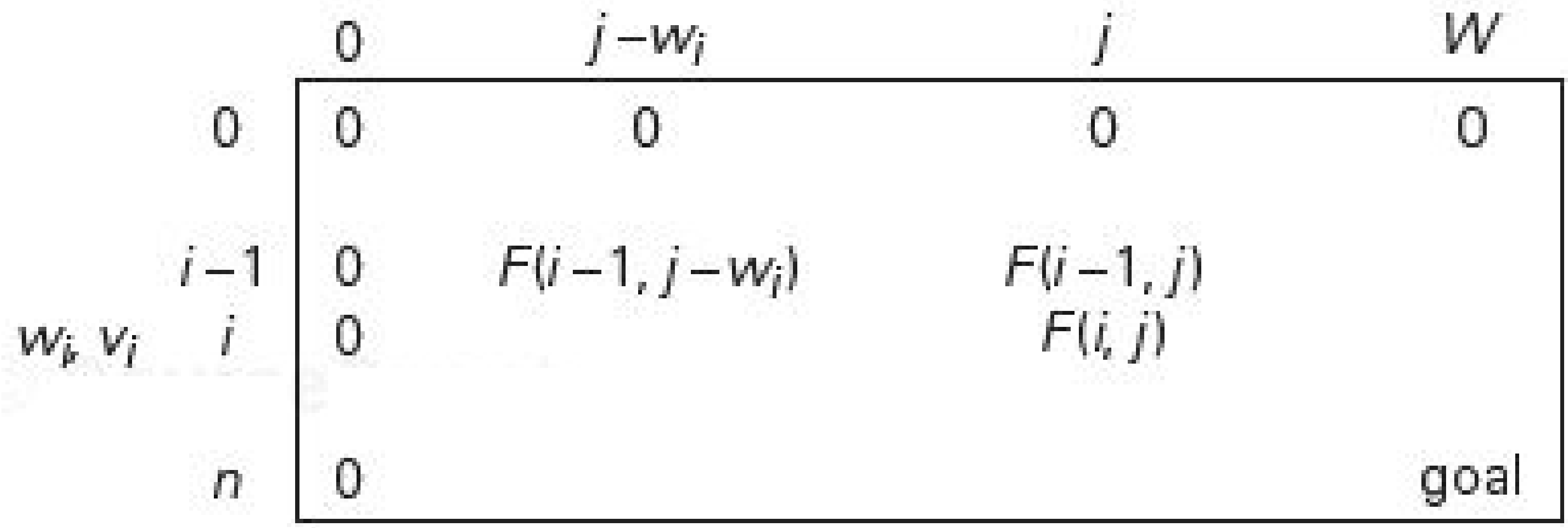
- **Problem Statement**: Design a dynamic programming algorithm for the integer-knapsack problem: given *n* items of known weights $w_1, w_2, \ldots, w_n$ (where all the weights are integers) and values $v_1, v_2, \ldots, v_n$ (the values need not be integers), and a knapsack capacity W (an integer), find the most valuable subset of the items that fit into the knapsack.
- **Solution**: Let F(*i*, *j*) be the value of the most valuable subset of the first *i* items ($1 \le i \le n$) that fit into the knapsack of capacity j ($1 \le j \le W$). We can divide all the subsets of the first *i* items that fit into the knapsack of capacity *j* into two categories: those that do not include the i^th item and those that do.
  - Among the subsets that do not include the i^th item, the value of an optimal subset is F(*i*-1, *j*).
  - Among the subsets that do include the i^th item (hence, $j - w_i >= 0$), 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 $j - w_i$. The value of such an optimal subset is $v_i + F(i-1, j - w_i)$.

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

Initial Condition: F(0, j) = 0 for 1 ≤ j ≤ W   F(i, 0) = 0 for 1 ≤ i ≤ n

Idea to Solve the Int. Knapsack Prob.

- The goal is to find F(n, W), the optimal value of a subset of the *n* given items that fit into the knapsack of capacity W, and an optimal subset itself.
- For i, j > 0, to compute the entry in the i^th row and j^th column, F(i, j), we compute the maximum of the entry in the previous row and the same column and the sum of v_i and the entry in the previous row and w_i columns to the left.
- The table can be filled either row-wise or column-wise.



		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

Example 1: Integer Knapsack Problem

- Find the composition of items that maximizes the value of the knapsack of integer capacity-weight 5.

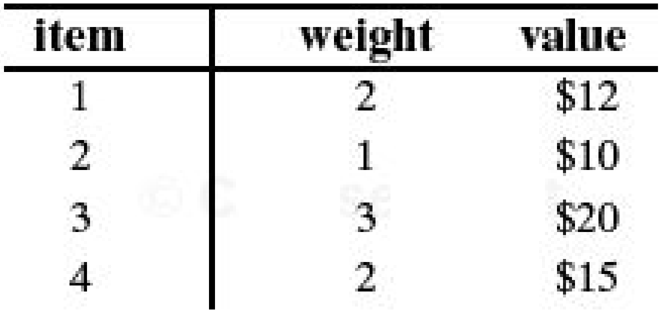
item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

		Capacity, j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
w1 = 2, v1 = 12	1	0					
w2 = 1, v2 = 10	2	0					
w3 = 3, v3 = 20	3	0					
w4 = 2, v4 = 15	4	0					

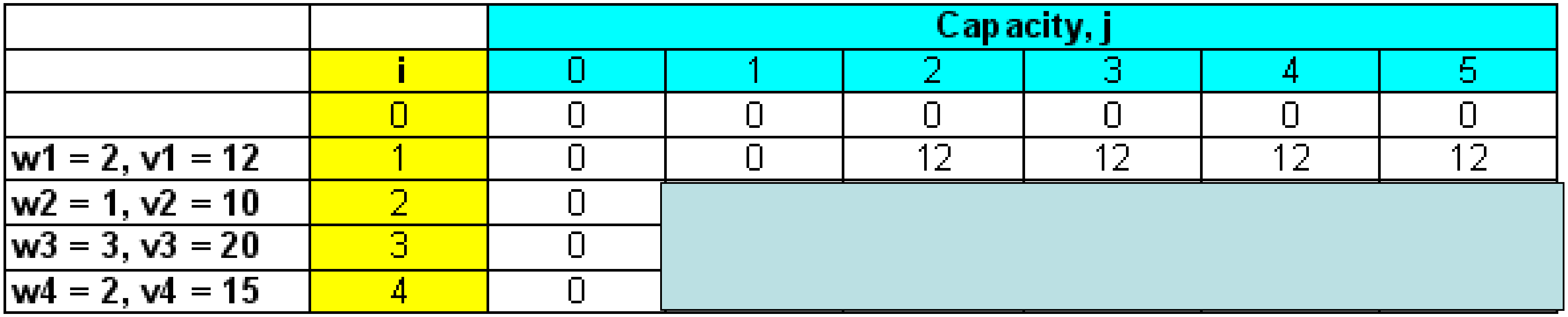
		Capacity, j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
w1 = 2, v1 = 12	1	0					
w2 = 1, v2 = 10	2	0					
w3 = 3, v3 = 20	3	0					
w4 = 2, v4 = 15	4	0					

Example 1: Integer Knapsack Problem

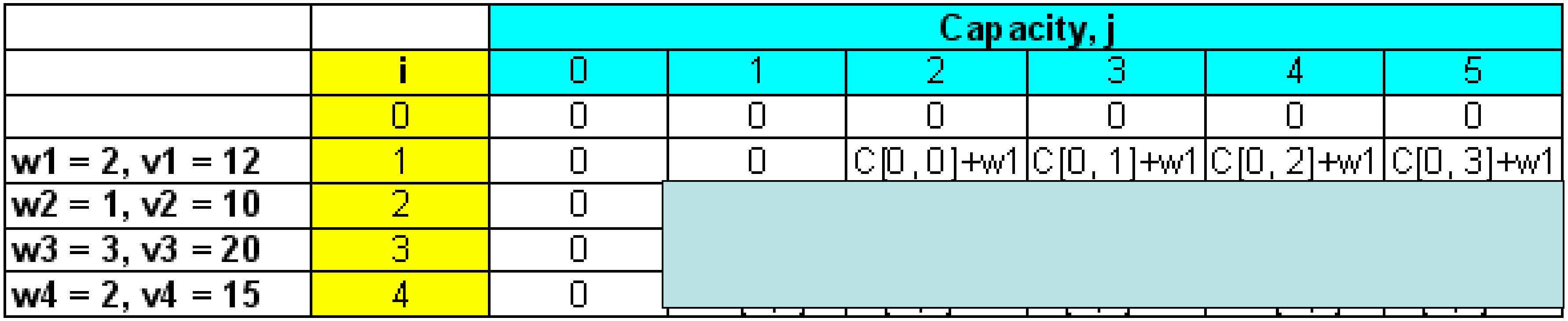
- Find the composition of items that maximizes the value of the knapsack of integer capacity-weight 5.



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



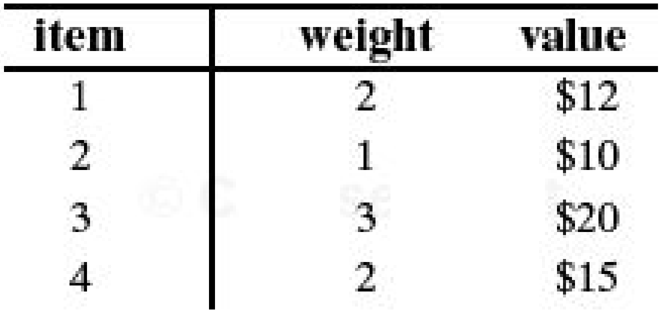
		Capacity, j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
w1 = 2, v1 = 12	1	0	0	12	12	12	12
w2 = 1, v2 = 10	2	0					
w3 = 3, v3 = 20	3	0					
w4 = 2, v4 = 15	4	0					



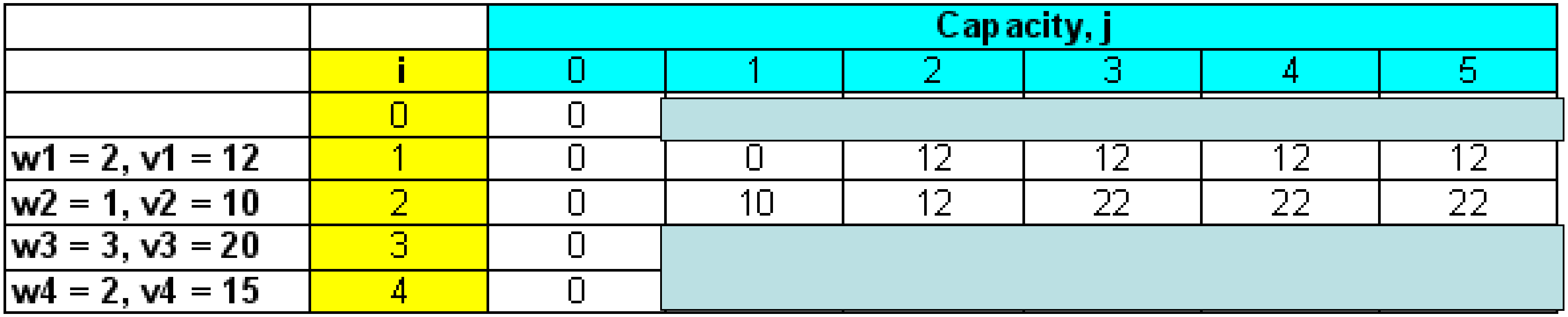
		Capacity, j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
w1 = 2, v1 = 12	1	0	0	C[0,0]+w1	C[0, 1]+w1	C[0, 2]+w1	C[0, 3]+w1
w2 = 1, v2 = 10	2	0					
w3 = 3, v3 = 20	3	0					
w4 = 2, v4 = 15	4	0					

Example 1: Integer Knapsack Problem

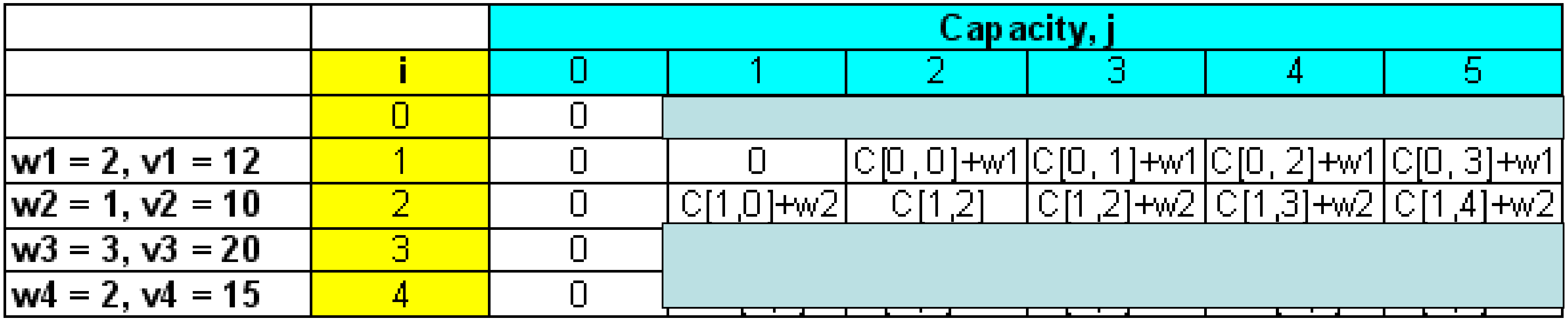
- Find the composition of items that maximizes the value of the knapsack of integer capacity-weight 5.



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



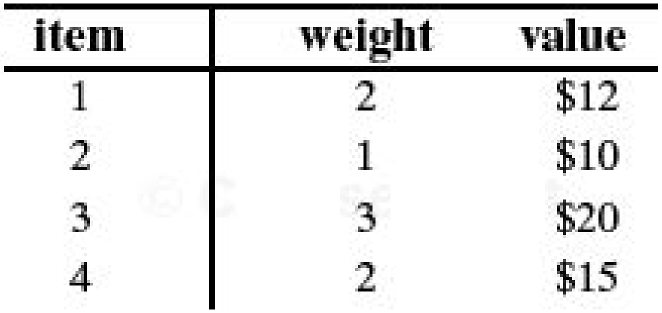
		Capacity, j					
	i	0	1	2	3	4	5
	0	0					
w1 = 2, v1 = 12	1	0	0	12	12	12	12
w2 = 1, v2 = 10	2	0	10	12	22	22	22
w3 = 3, v3 = 20	3	0					
w4 = 2, v4 = 15	4	0					



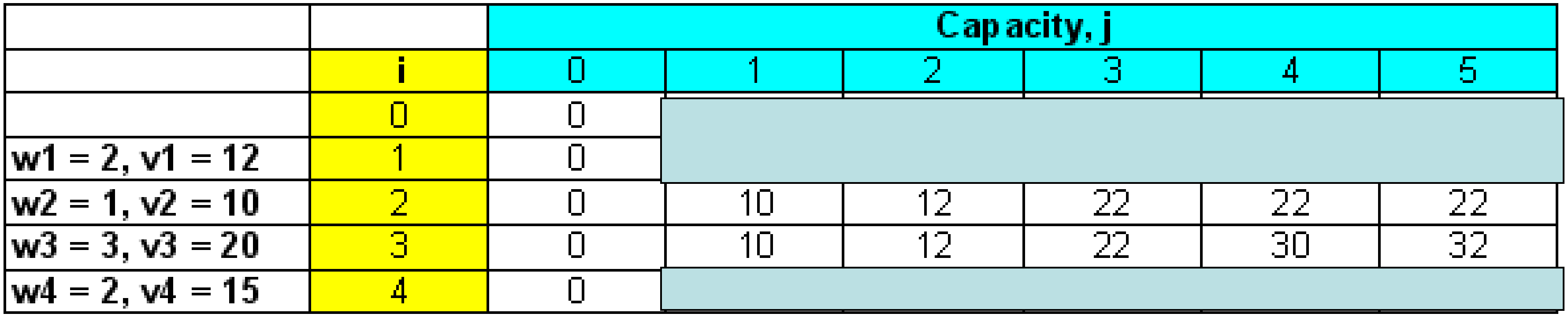
		Capacity, j					
	i	0	1	2	3	4	5
	0	0					
w1 = 2, v1 = 12	1	0	0	C[0,0]+w1	C[0, 1]+w1	C[0, 2]+w1	C[0, 3]+w1
w2 = 1, v2 = 10	2	0	C[1,0]+w2	C[1,2]	C[1,2]+w2	C[1,3]+w2	C[1,4]+w2
w3 = 3, v3 = 20	3	0					
w4 = 2, v4 = 15	4	0					

Example 1: Integer Knapsack Problem

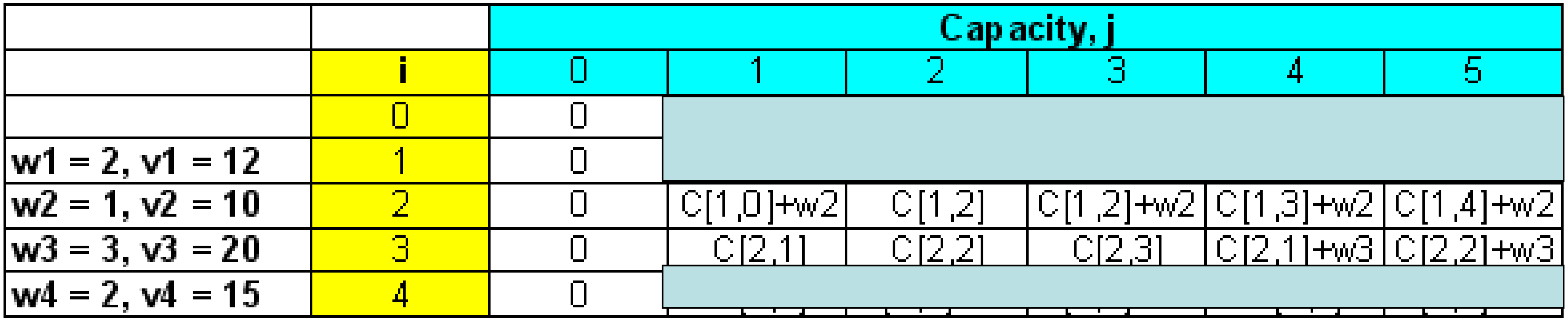
- Find the composition of items that maximizes the value of the knapsack of integer capacity-weight 5.



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



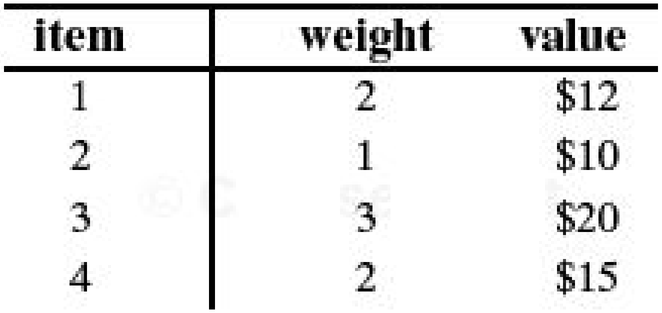
		Capacity, j					
	i	0	1	2	3	4	5
	0	0					
w1 = 2, v1 = 12	1	0					
w2 = 1, v2 = 10	2	0	10	12	22	22	22
w3 = 3, v3 = 20	3	0	10	12	22	30	32
w4 = 2, v4 = 15	4	0					



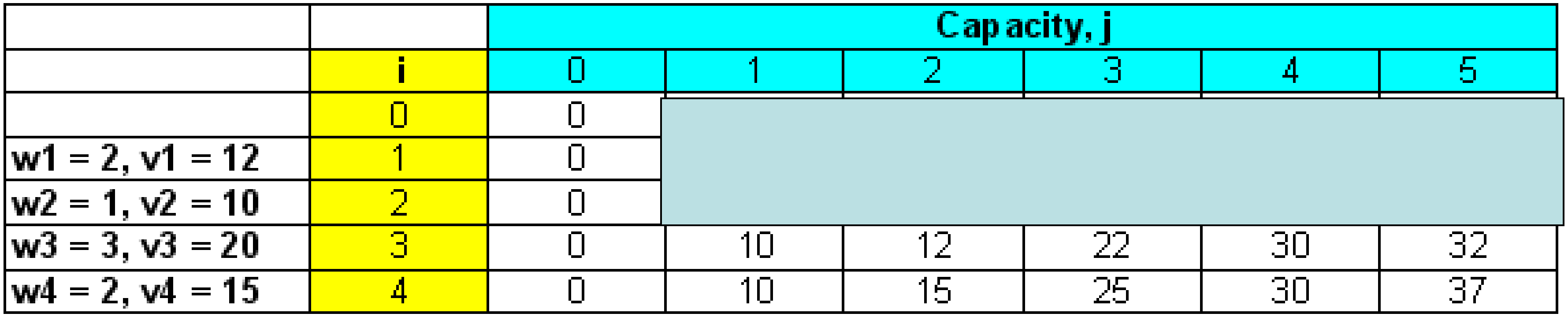
		Capacity, j					
	i	0	1	2	3	4	5
	0	0					
w1 = 2, v1 = 12	1	0					
w2 = 1, v2 = 10	2	0	C[1,0]+w2	C[1,2]	C[1,2]+w2	C[1,3]+w2	C[1,4]+w2
w3 = 3, v3 = 20	3	0	C[2,1]	C[2,2]	C[2,3]	C[2,1]+w3	C[2,2]+w3
w4 = 2, v4 = 15	4	0					

Example 1: Integer Knapsack Problem

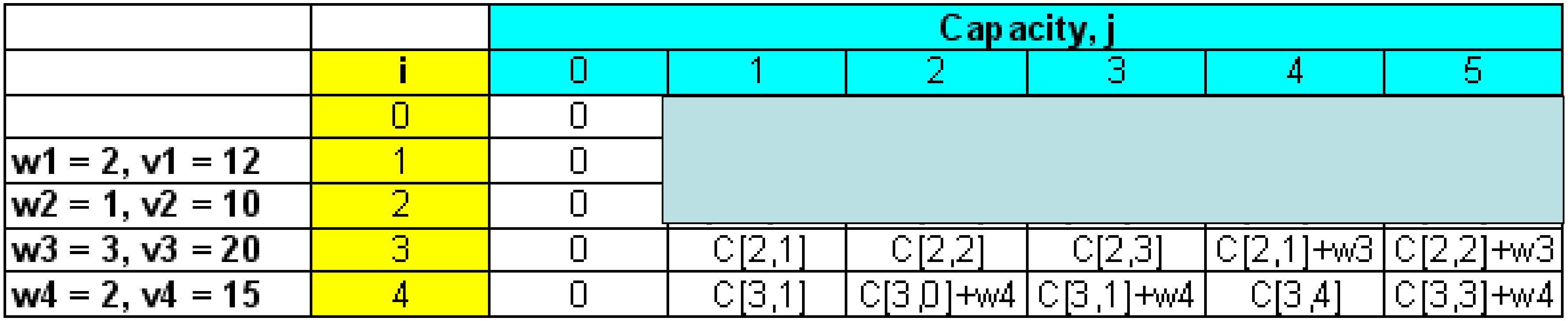
- Find the composition of items that maximizes the value of the knapsack of integer capacity-weight 5.



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



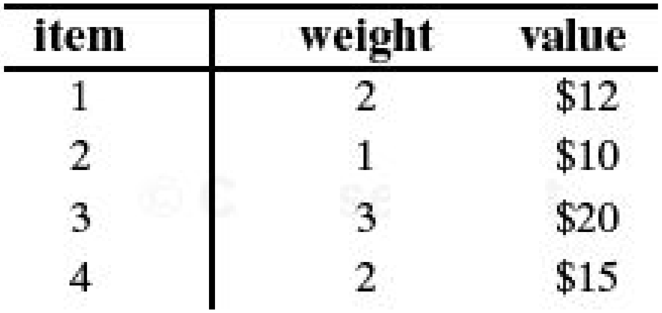
		Capacity, j					
	i	0	1	2	3	4	5
	0	0					
w1 = 2, v1 = 12	1	0					
w2 = 1, v2 = 10	2	0					
w3 = 3, v3 = 20	3	0	10	12	22	30	32
w4 = 2, v4 = 15	4	0	10	15	25	30	37



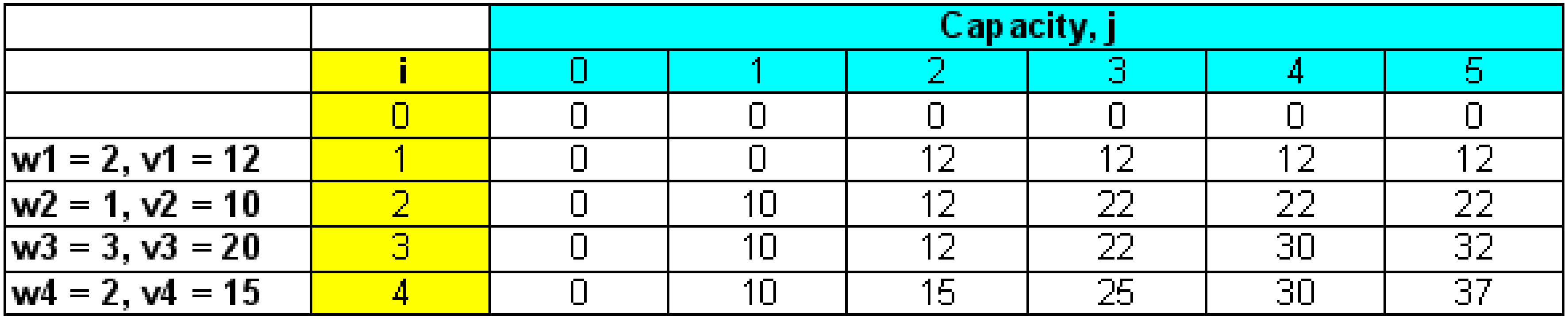
		Capacity, j					
	i	0	1	2	3	4	5
	0	0					
w1 = 2, v1 = 12	1	0					
w2 = 1, v2 = 10	2	0					
w3 = 3, v3 = 20	3	0	C[2,1]	C[2,2]	C[2,3]	C[2,1]+w3	C[2,2]+w3
w4 = 2, v4 = 15	4	0	C[3,1]	C[3,0]+w4	C[3,1]+w4	C[3,4]	C[3,3]+w4

Example 1: Integer Knapsack Problem

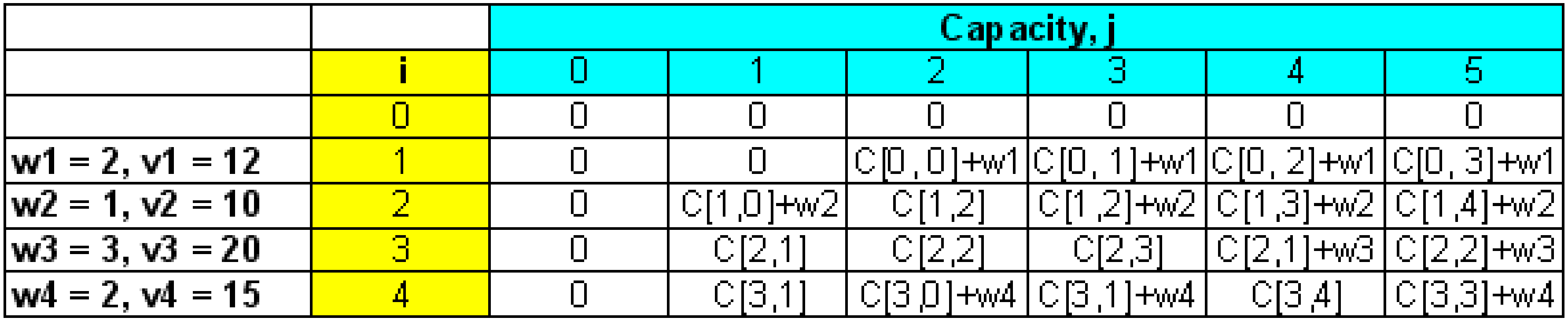
- Find the composition of items that maximizes the value of the knapsack of integer capacity-weight 5.



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



		Capacity, j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
w1 = 2, v1 = 12	1	0	0	12	12	12	12
w2 = 1, v2 = 10	2	0	10	12	22	22	22
w3 = 3, v3 = 20	3	0	10	12	22	30	32
w4 = 2, v4 = 15	4	0	10	15	25	30	37

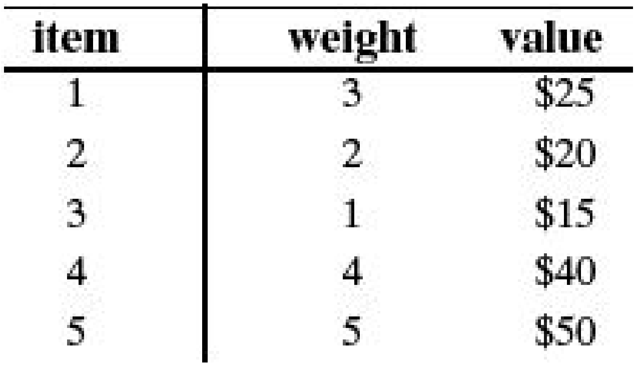


		Capacity, j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
w1 = 2, v1 = 12	1	0	0	C[0,0]+w1	C[0, 1]+w1	C[0, 2]+w1	C[0, 3]+w1
w2 = 1, v2 = 10	2	0	C[1,0]+w2	C[1,2]	C[1,2]+w2	C[1,3]+w2	C[1,4]+w2
w3 = 3, v3 = 20	3	0	C[2,1]	C[2,2]	C[2,3]	C[2,1]+w3	C[2,2]+w3
w4 = 2, v4 = 15	4	0	C[3,1]	C[3,0]+w4	C[3,1]+w4	C[3,4]	C[3,3]+w4

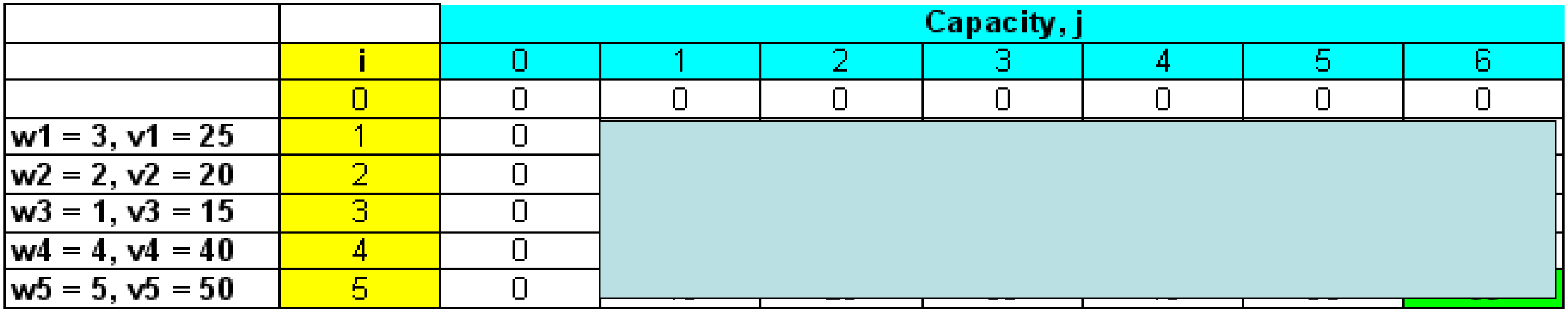
Choose W4(2), W2(1), W1(2), with values totaling to 37 and capacity 5.

Example 2: Integer Knapsack Problem

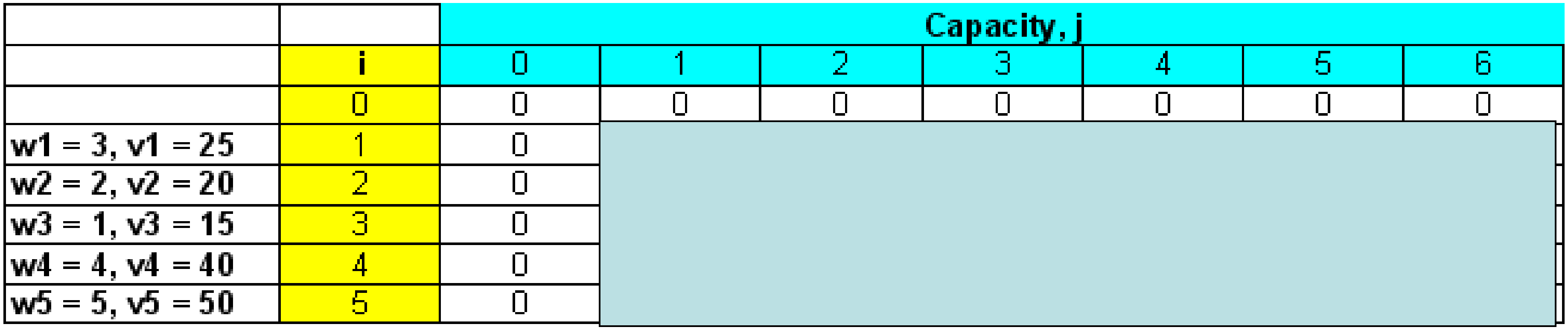
- Find the composition of items that maximizes the value of the knapsack of integer capacity-weight 6.



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



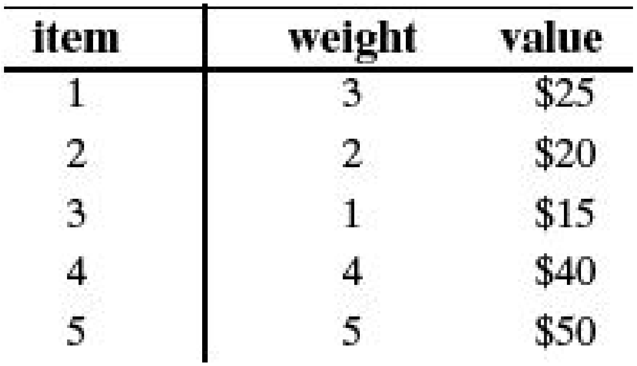
		Capacity, j						
	i	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
w1 = 3, v1 = 25	1	0						
w2 = 2, v2 = 20	2	0						
w3 = 1, v3 = 15	3	0						
w4 = 4, v4 = 40	4	0						
w5 = 5, v5 = 50	5	0						



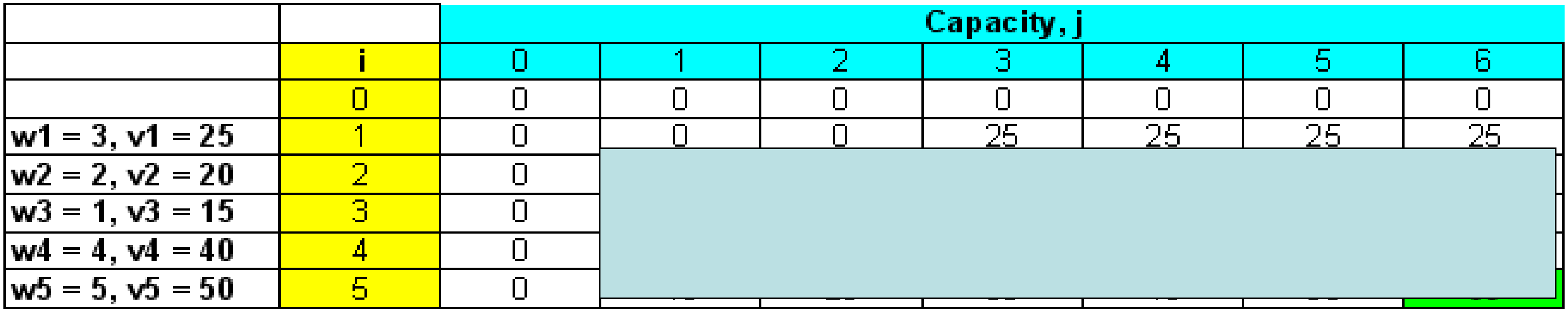
		Capacity, j						
	i	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
w1 = 3, v1 = 25	1	0						
w2 = 2, v2 = 20	2	0						
w3 = 1, v3 = 15	3	0						
w4 = 4, v4 = 40	4	0						
w5 = 5, v5 = 50	5	0						

Example 2: Integer Knapsack Problem

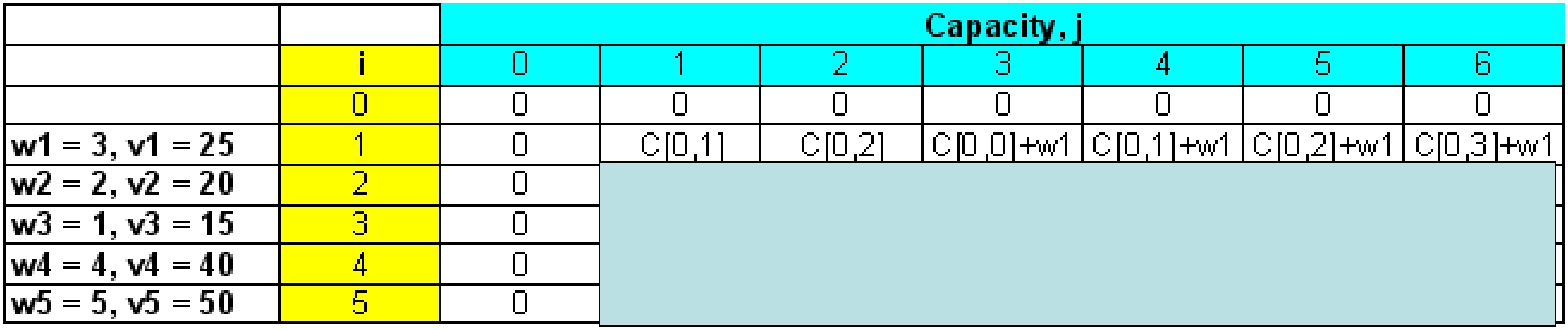
- Find the composition of items that maximizes the value of the knapsack of integer capacity-weight 6.



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



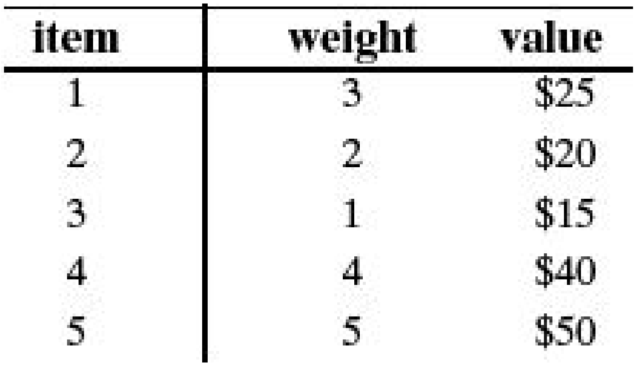
		Capacity, j						
	i	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
w1 = 3, v1 = 25	1	0	0	0	25	25	25	25
w2 = 2, v2 = 20	2	0						
w3 = 1, v3 = 15	3	0						
w4 = 4, v4 = 40	4	0						
w5 = 5, v5 = 50	5	0						



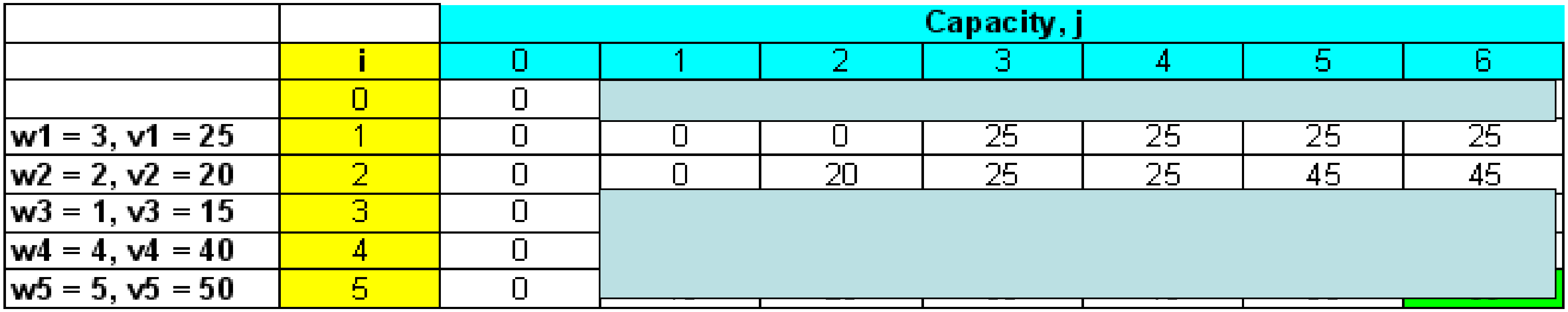
		Capacity, j						
	i	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
w1 = 3, v1 = 25	1	0	C[0,1]	C[0,2]	C[0,0]+w1	C[0,1]+w1	C[0,2]+w1	C[0,3]+w1
w2 = 2, v2 = 20	2	0						
w3 = 1, v3 = 15	3	0						
w4 = 4, v4 = 40	4	0						
w5 = 5, v5 = 50	5	0						

Example 2: Integer Knapsack Problem

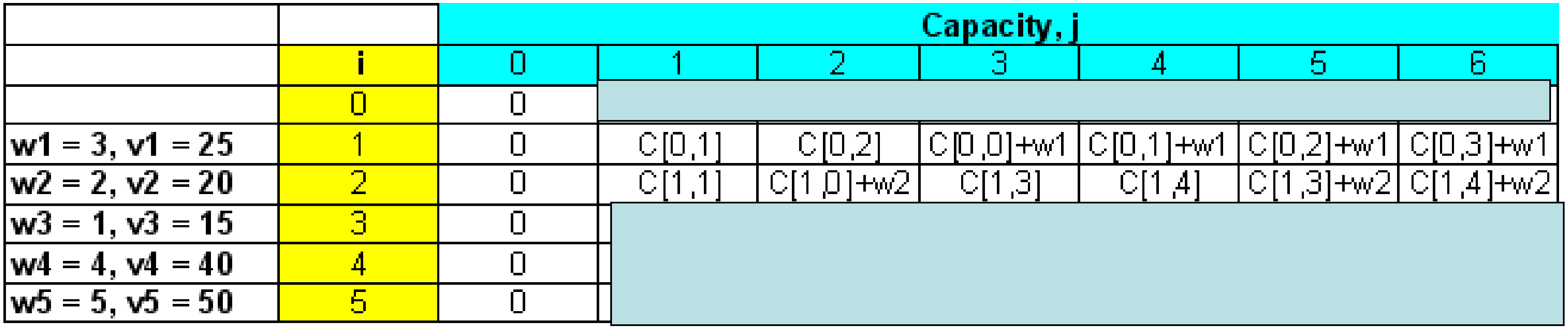
- Find the composition of items that maximizes the value of the knapsack of integer capacity-weight 6.



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



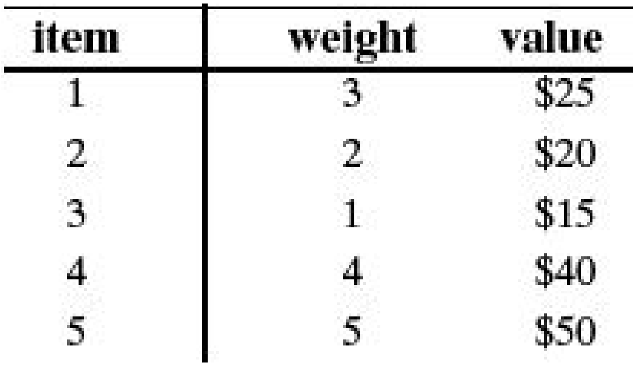
		Capacity, j						
	i	0	1	2	3	4	5	6
	0	0						
w1 = 3, v1 = 25	1	0	0	0	25	25	25	25
w2 = 2, v2 = 20	2	0	0	20	25	25	45	45
w3 = 1, v3 = 15	3	0						
w4 = 4, v4 = 40	4	0						
w5 = 5, v5 = 50	5	0						



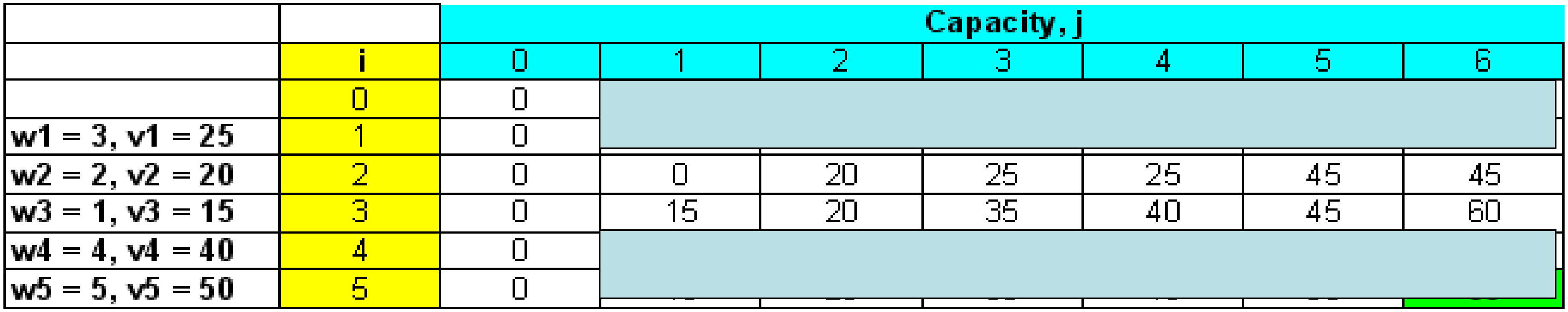
		Capacity, j						
	i	0	1	2	3	4	5	6
	0	0						
w1 = 3, v1 = 25	1	0	C[0,1]	C[0,2]	C[0,0]+w1	C[0,1]+w1	C[0,2]+w1	C[0,3]+w1
w2 = 2, v2 = 20	2	0	C[1,1]	C[1,0]+w2	C[1,3]	C[1,4]	C[1,3]+w2	C[1,4]+w2
w3 = 1, v3 = 15	3	0						
w4 = 4, v4 = 40	4	0						
w5 = 5, v5 = 50	5	0						

Example 2: Integer Knapsack Problem

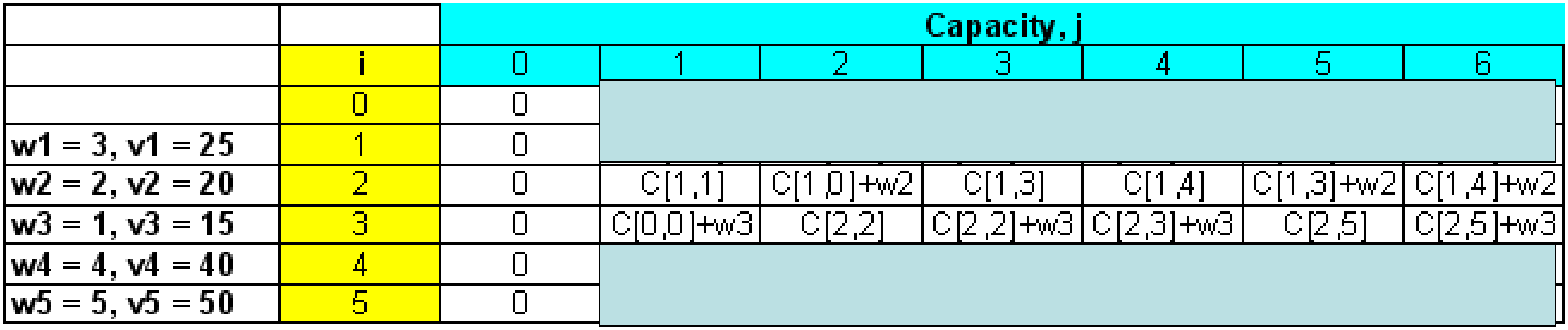
- Find the composition of items that maximizes the value of the knapsack of integer capacity-weight 6.



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



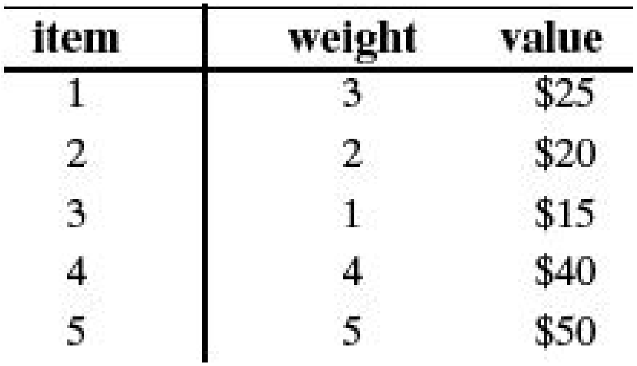
		Capacity, j						
	i	0	1	2	3	4	5	6
	0	0						
w1 = 3, v1 = 25	1	0						
w2 = 2, v2 = 20	2	0	0	20	25	25	45	45
w3 = 1, v3 = 15	3	0	15	20	35	40	45	60
w4 = 4, v4 = 40	4	0						
w5 = 5, v5 = 50	5	0						



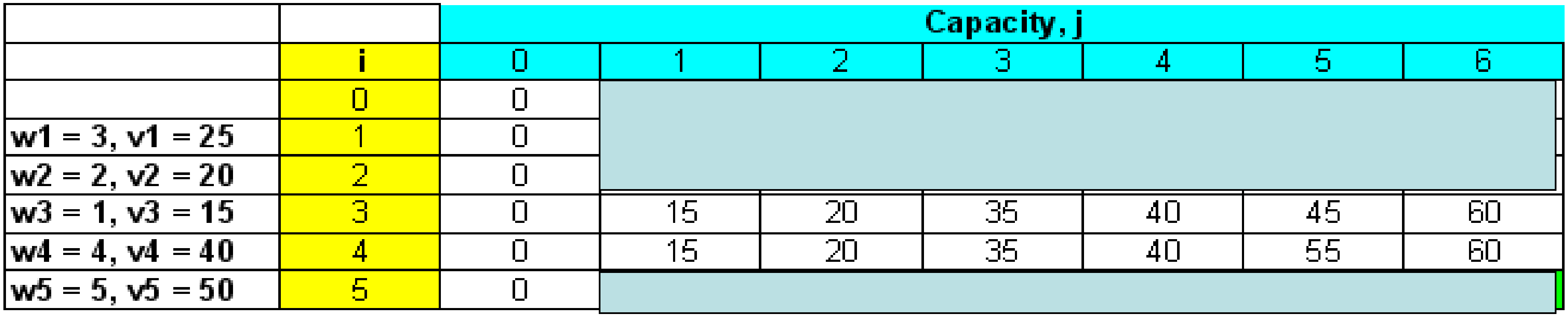
		Capacity, j						
	i	0	1	2	3	4	5	6
	0	0						
w1 = 3, v1 = 25	1	0						
w2 = 2, v2 = 20	2	0	C[1,1]	C[1 ,0]+w2	C[1,3]	C[1 ,4]	C[1,3]+w2	C[1,4]+w2
w3 = 1, v3 = 15	3	0	C[0,0]+w3	C[2,2]	C[2,2]+w3	C[2,3]+w3	C[2,5]	C[2,5]+w3
w4 = 4, v4 = 40	4	0						
w5 = 5, v5 = 50	5	0						

Example 2: Integer Knapsack Problem

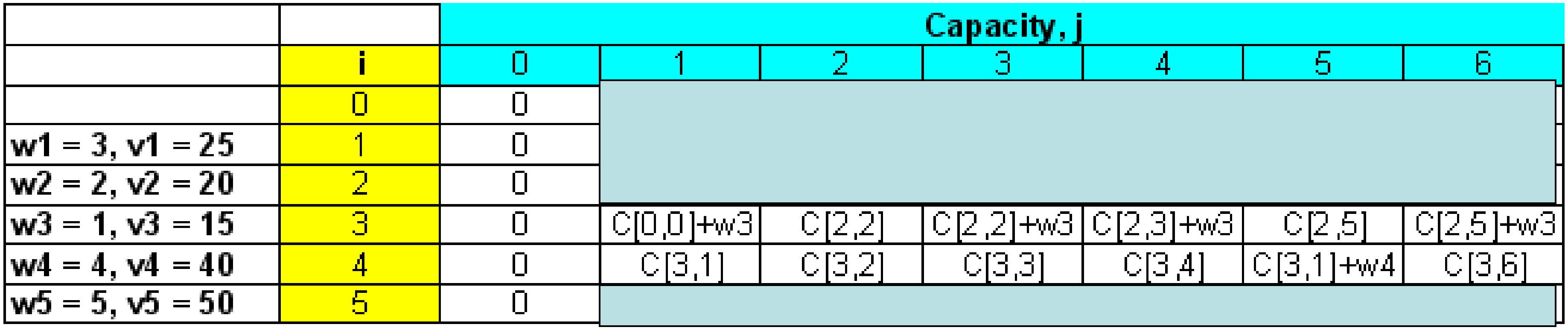
- Find the composition of items that maximizes the value of the knapsack of integer capacity-weight 6.



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



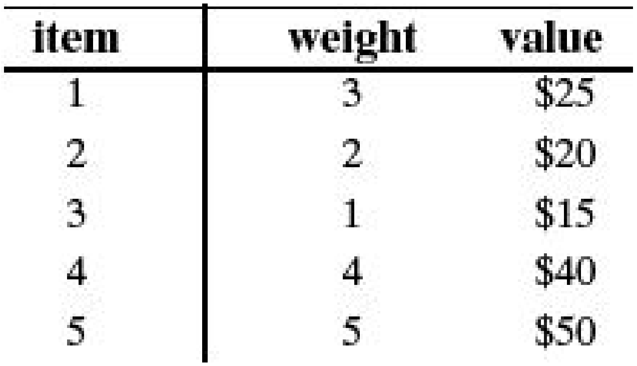
		Capacity, j						
	i	0	1	2	3	4	5	6
	0	0						
w1 = 3, v1 = 25	1	0						
w2 = 2, v2 = 20	2	0						
w3 = 1, v3 = 15	3	0	15	20	35	40	45	60
w4 = 4, v4 = 40	4	0	15	20	35	40	55	60
w5 = 5, v5 = 50	5	0						



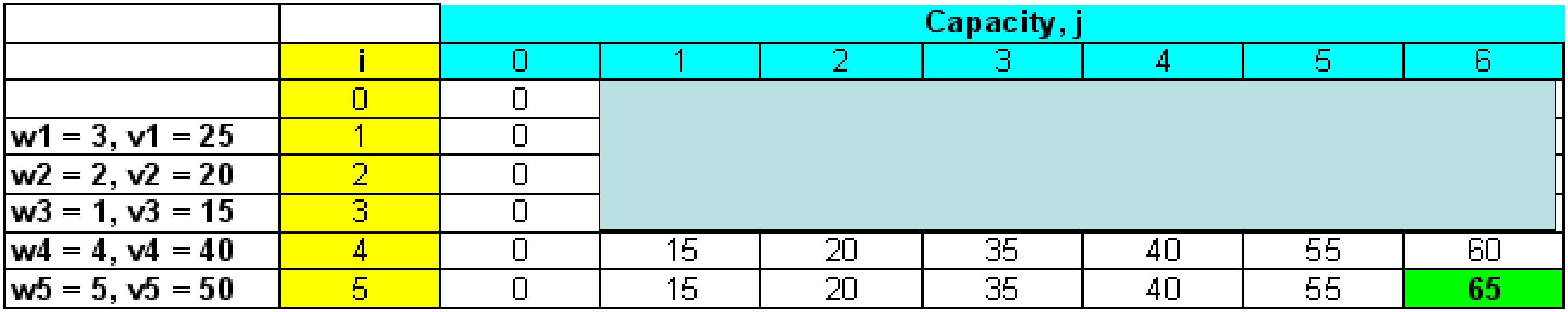
		Capacity, j						
	i	0	1	2	3	4	5	6
	0	0						
w1 = 3, v1 = 25	1	0						
w2 = 2, v2 = 20	2	0						
w3 = 1, v3 = 15	3	0	C[0,0]+w3	C[2,2]	C[2,2]+w3	C[2,3]+w3	C[2,5]	C[2,5]+w3
w4 = 4, v4 = 40	4	0	C[3,1]	C[3,2]	C[3,3]	C[3,4]	C[3,1]+w4	C[3,6]
w5 = 5, v5 = 50	5	0						

Example 2: Integer Knapsack Problem

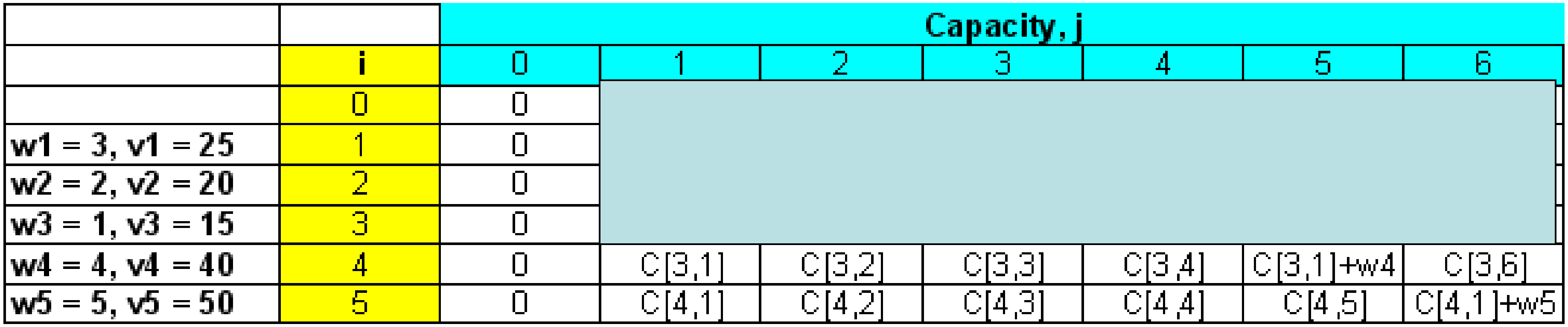
- Find the composition of items that maximizes the value of the knapsack of integer capacity-weight 6.



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



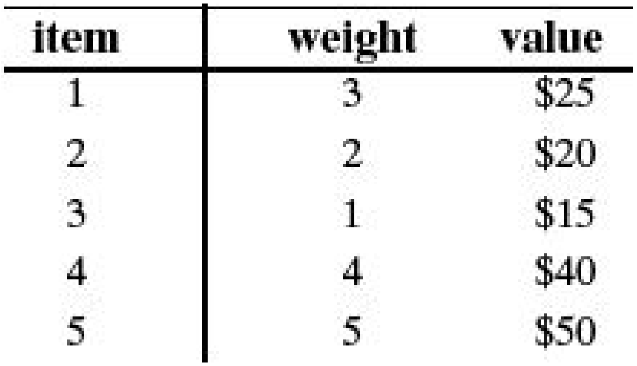
		Capacity, j						
	i	0	1	2	3	4	5	6
	0	0						
w1 = 3, v1 = 25	1	0						
w2 = 2, v2 = 20	2	0						
w3 = 1, v3 = 15	3	0						
w4 = 4, v4 = 40	4	0	15	20	35	40	55	60
w5 = 5, v5 = 50	5	0	15	20	35	40	55	65



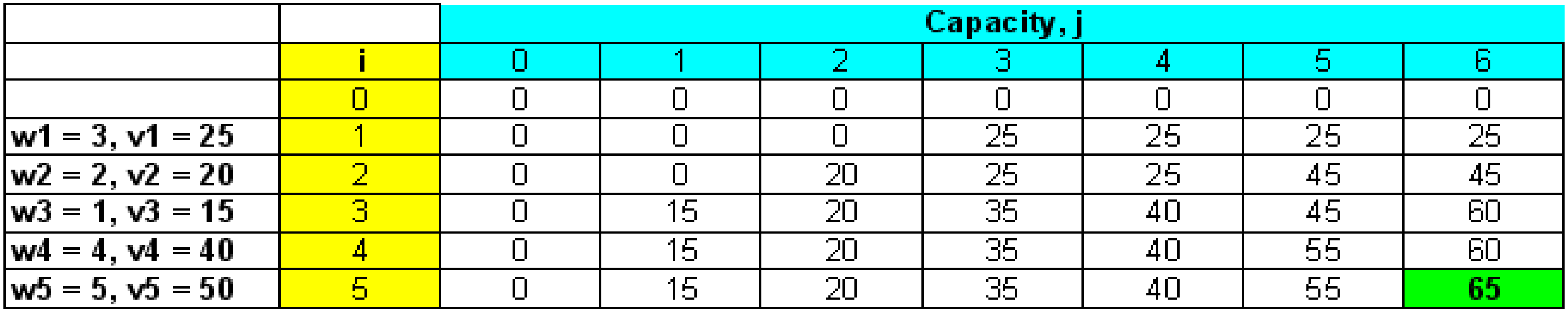
		Capacity, j						
	i	0	1	2	3	4	5	6
	0	0						
w1 = 3, v1 = 25	1	0						
w2 = 2, v2 = 20	2	0						
w3 = 1, v3 = 15	3	0						
w4 = 4, v4 = 40	4	0	C[3,1]	C[3,2]	C[3,3]	C[3,4]	C[3,1]+w4	C[3,6]
w5 = 5, v5 = 50	5	0	C[4,1]	C[4,2]	C[4,3]	C[4,4]	C[4,5]	C[4,1]+w5

Example 2: Integer Knapsack Problem

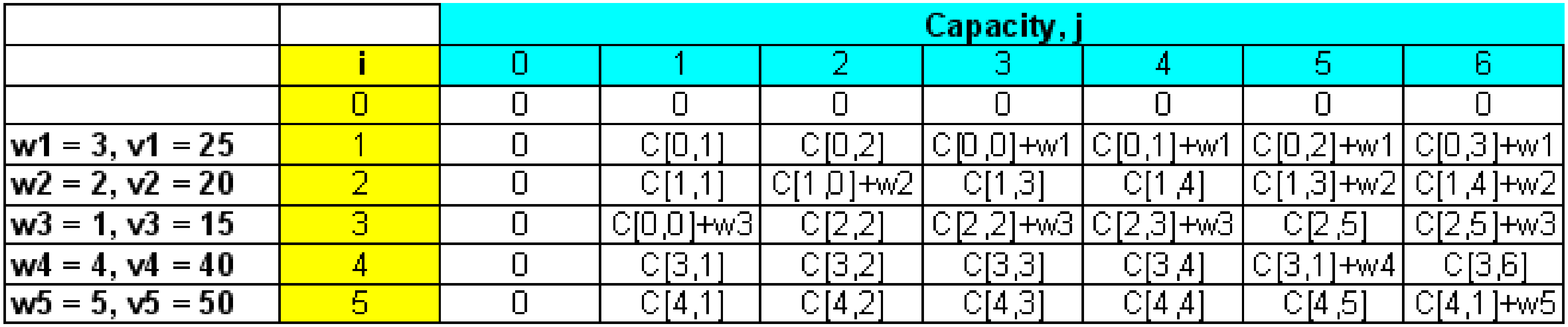
- Find the composition of items that maximizes the value of the knapsack of integer capacity-weight 6.



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



		Capacity, j						
	i	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
w1 = 3, v1 = 25	1	0	0	0	25	25	25	25
w2 = 2, v2 = 20	2	0	0	20	25	25	45	45
w3 = 1, v3 = 15	3	0	15	20	35	40	45	60
w4 = 4, v4 = 40	4	0	15	20	35	40	55	60
w5 = 5, v5 = 50	5	0	15	20	35	40	55	**65**



		Capacity, j						
	i	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
w1 = 3, v1 = 25	1	0	C[0,1]	C[0,2]	C[0,0]+w1	C[0,1]+w1	C[0,2]+w1	C[0,3]+w1
w2 = 2, v2 = 20	2	0	C[1,1]	C[1,0]+w2	C[1,3]	C[1,4]	C[1,3]+w2	C[1,4]+w2
w3 = 1, v3 = 15	3	0	C[0,0]+w3	C[2,2]	C[2,2]+w3	C[2,3]+w3	C[2,5]	C[2,5]+w3
w4 = 4, v4 = 40	4	0	C[3,1]	C[3,2]	C[3,3]	C[3,4]	C[3,1]+w4	C[3,6]
w5 = 5, v5 = 50	5	0	C[4,1]	C[4,2]	C[4,3]	C[4,4]	C[4,5]	C[4,1]+w5

Choose W5(5) and W3(1) with values totaling to $65 and capacity 6.

Matrix Multiplication

- Given two matrices A_{mxn} and B_{nxp}, the requirement to be able to multiply A x B is that the number of columns (n) in the first matrix (A) should be the same as the number of rows (n) in the second matrix (B). The resulting product matrix is of dimension mxp.
- $A_{mxn} * B_{nxp} = C_{mxp}$
- We do ‘n’ multiplications to fill up a cell in the product matrix C. As there are mxp such cells, the total number of multiplications encountered in the above case is **mxnxp**.
- Example:

4 multiplications / cell

$$\xleftrightarrow{\hspace{6em}}$$

$$3*1 + 4*4 + 1*3 + 2*5 = 32$$

$$\underset{\mathbf{3x4}}{\boxed{\begin{matrix}3&4&1&2\\5&2&2&1\\1&3&4&2\end{matrix}}} \; \mathbf{*} \; \underset{\mathbf{4x2}}{\boxed{\begin{matrix}1&2\\4&5\\3&2\\5&1\end{matrix}}} \; \mathbf{=} \; \underset{\mathbf{3x2}}{\boxed{\begin{matrix}x&x\\x&x\\x&x\end{matrix}}}$$

Total number of Multiplications
3x4x2 = 24

Motivating Example

- Matrix multiplication is commutative.
- For example: A*B*C can be multiplied as (A*B)*C = A*(B*C)
- Consider three matrices A^1_{3x4}, A^2_{4x5}, A^3_{5x2}. How should we parenthesize them so that we do the minimum number of multiplications?
- $A^1_{3x4} * A^2_{4x5} * A^3_{5x2}$

# Multipl.

$$= A^1_{3x4} * (A^2_{4x5} * A^3_{5x2}) = A^1_{3x4} * (\quad)_{4x2}$$

4x5x2 = 40 **3x4x2 = 24**

40 + 24 = 64

$$= (A^1_{3x4} * A^2_{4x5}) * A^3_{5x2} = (\ )_{3x5} * A^3_{5x2}$$

3x4x5 = 60 **3x5x2 = 30**

60 + 30 = 90

So, the best way to parenthesize is **A1 * (A2 * A3)**

Matrix Chain Multiplication

- Given a sequence of matrices, A1, …, An: the problem is to find the best way to parenthesize them so that we do the minimum number of multiplications.

A1 x A2 x A3 x …. x An

$p_0 x p_1$ $p_1 x p_2$ $p_2 x p_3$ ……… $p_{n-1} x p_n$

Optimal Substructure

Given a spread of matrices A_i x …. x A_j, we need to find a 'k' such that i ≤ k < j such that multiplying A_i x …. x A_j = (A_i x …. x A_k) * (A_{k+1} x …. x A_j) would result in the minimum number of multiplications for all possible values of k; i ≤ k < j

Let M[i, j] indicate the minimum number of multiplications needed to compute Ai x …. x Aj, where i ≤ j

$$M[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \left(M[i, k] + M[k+1, j] + p_{i-1} * p_k * p_j \right) & \end{cases}$$

Example to Illustrate the Recurrence

$$M[i, j] = \begin{cases} 0 & \text{if } i = j \\ \underset{i \le k < j}{\text{Min}} \left(M[i, k] + M[k+1, j] + p_{i-1} {}^{*} p_k {}^{*} p_j \right) \end{cases}$$

The dimension of the product matrix **(A_i x …. x A_k) is p_{i-1}xp_k**

p_{i-1}xp_i p_{k-1}xp_k

The dimension of the product matrix **(A_{k+1} x …. x A_j) is p_kxp_j**

P_kxp_{k+1} p_{j-1}xp_j

Let us say, we want to find A2….A7 and we decide to try for k = 4

That is, we want to multiply

A2 x … x A7 = (A2 x … x A4) * (A5 x … x A7)

Dimensions: 4 x 3 4 x 7 7 x 3

M[2, 7] for k = 4 is

M[2, 4] + M[5, 7] + $(\underset{\substack{\uparrow \\ p1 \\ p_{i-1}}}{4} \ {}^{*}\underset{\substack{\uparrow \\ p4 \\ p_k}}{7} \ {}^{*} \underset{\substack{\uparrow \\ p7 \\ p_j}}{3})$

A1	p0 x p1	3x4
A2	p1 x p2	4x5
A3	p2 x p3	5x2
A4	p3 x p4	2x7
A5	p4 x p5	7x3
A6	p5 x p6	3x5
A7	p6 x p7	5x3
A8	p7 x p8	3x6