# Module 4
# Dynamic Programming

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

# Introduction to Dynamic Programming

- Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping sub problems
- "Programming" here means "planning"
- Main idea:
    - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
    - solve smaller instances once
    - record solutions in a table
    - 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).
- Example: 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).

# 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:** $C(n,k) = C(n\text{-}1,k) + C(n\text{-}1,k\text{-}1)$ **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 | | | | | $C(n\text{-}1,k\text{-}1)$ | $C(n\text{-}1,k)$ |
| $n$ | | | | | | $C(n,k)$ |

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

# Computing $C(12,5)$

k →

| n ↓ | | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | | | | | |
| | 1 | 1 | 1 | | | | |
| | 2 | 1 | 2 | 1 | | | |
| | 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 ≥ k ≥ 0
    //Output: The value of C(n, k)
    for i ← 0 to n do
        for j ← 0 to min(i, k) do
            if j = 0 or j = i
                C[i, j] ← 1
            else C[i, j] ← C[i − 1, j − 1] + C[i − 1, j]
    return C[n, k]
```

**Time efficiency: Θ(*nk*)**

**Space efficiency: Θ(*nk*)**

# 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

**Recurrence**

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

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

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

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

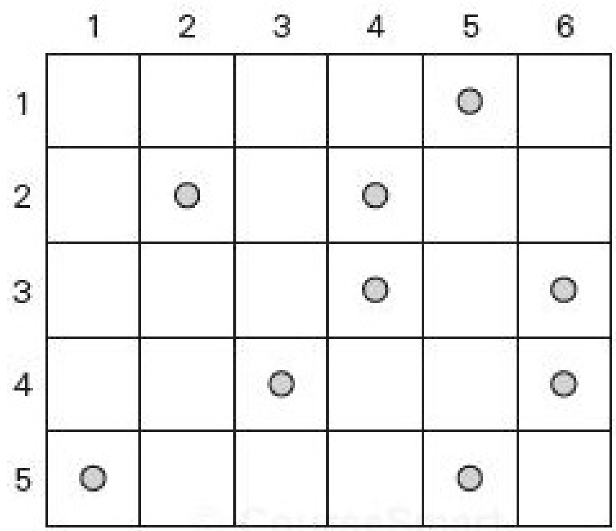
//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an $n \times m$ board by starting at (1, 1)
//and moving right and down from upper left to down right corner
//Input: Matrix $C[1..n, 1..m]$ whose elements are equal to 1 and 0
//for cells with and without a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

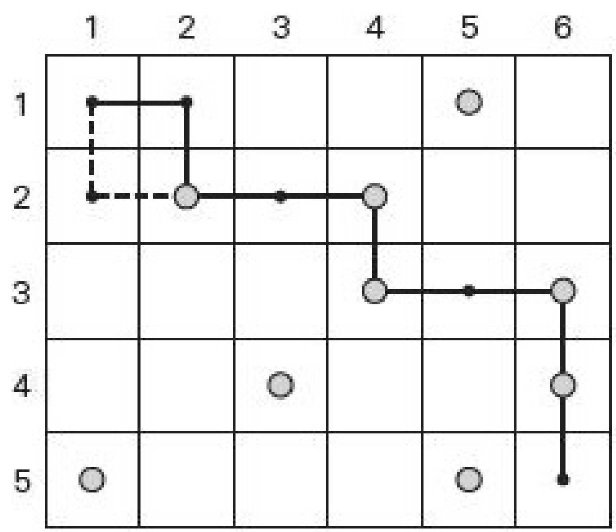
**Time Complexity: Θ(nm)**
**Space Complexity: Θ(nm)**

# Coin-Collecting Problem

- Tracing back the optimal path:
- 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 $\Theta(n+m)$ time.

# Example 1: Coin-Collecting Problem



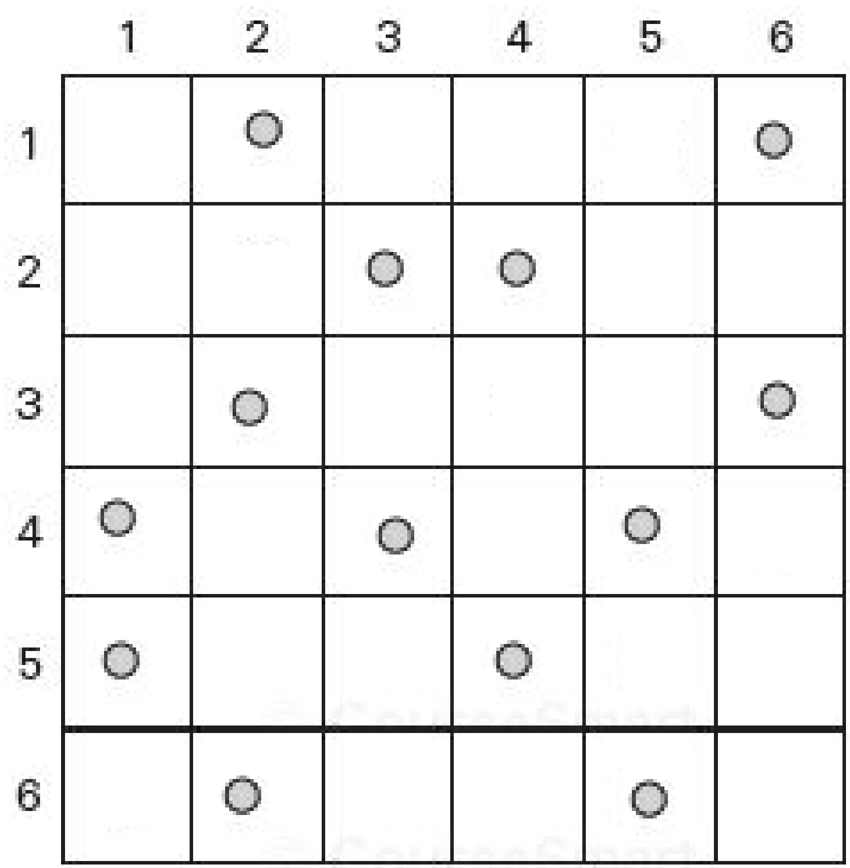




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 0 | 1 | 1 | 2 | 2 | 2 |
| 3 | 0 | 0 | 1 | 1 | 3 | 3 | 4 |
| 4 | 0 | 0 | 1 | 2 | 3 | 3 | 5 |
| 5 | 0 | 1 | 1 | 2 | 3 | 4 | 5 |

Note: The same table will be obtained even if the cells are filled column-wise.

# Example 2: Coin-Collecting Problem

**Given 6 x 6 Board**



**6 x 6 Table Filled up with F values**

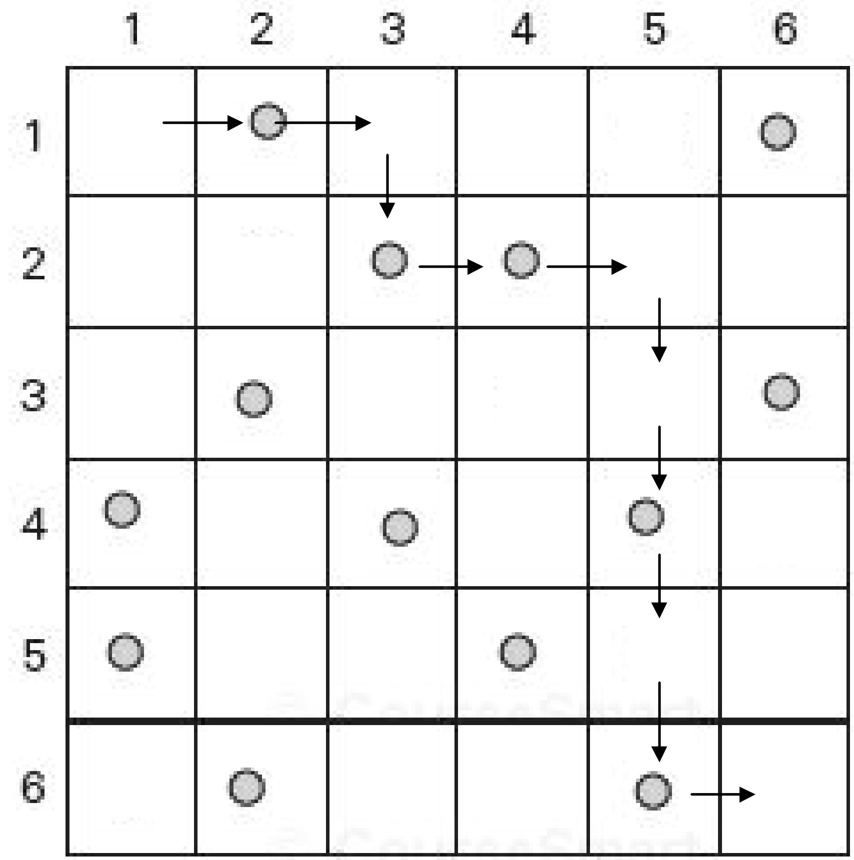
| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 2 |
| 2 | 0 | 1 | 2 | 3 | 3 | 3 |
| 3 | 0 | 2 | 2 | 3 | 3 | 4 |
| 4 | 1 | 2 | 3 | 3 | 4 | 4 |
| 5 | 2 | 2 | 3 | 4 | 4 | 4 |
| 6 | 2 | 3 | 3 | 4 | 5 | 5 |

# Example 2: Coin-Collecting Problem

**Tracing back the path through the F values**

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 2 |
| 2 | 0 | 1 | 2 | 3 | 3 | 3 |
| 3 | 0 | 2 | 2 | 3 | 3 | 4 |
| 4 | 1 | 2 | 3 | 3 | 4 | 4 |
| 5 | 2 | 2 | 3 | 4 | 4 | 4 |
| 6 | 2 | 3 | 3 | 4 | 5 | 5 |

**Optimal Path for the Robot to Collect the maximum # of coins**



# Coin-Row Problem

- **Problem Statement:** There is a row of *n* coins whose values are some positive integers $c_1, c_2, \ldots, c_n$, not necessarily distinct. The objective is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.
- **Recurrence:**
  - $F(n) = \text{Max}\{ c_n + F(n-2), F(n-1)\}$ for $n > 1$
  - $F(0) = 0$; $F(1) = c_1$.

**Θ(n) complexity for both time and space**

```
ALGORITHM  CoinRow(C[1..n])
    //Applies formula (8.3) bottom up to find the maximum amount of money
    //that can be picked up from a coin row without picking two adjacent coins
    //Input: Array C[1..n] of positive integers indicating the coin values
    //Output: The maximum amount of money that can be picked up
    F[0] ← 0;   F[1] ← C[1]
    for i ← 2 to n do
        F[i] ← max(C[i] + F[i − 2], F[i − 1])
    return F[n]
```

# Example 1 for Coin-Row Problem

**Solve for the coin-row problem for the instance {5, 1, 2, 10, 6, 2}**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| **C** | | 5 | 1 | 2 | 10 | 6 | 2 |
| **F** | 0 | 5 | 5 | 7 | 15 | 15 | **17** |
| **History** | | **C1** | - | C3 | **C4** | - | **C6** |
| | | - | **F[1]** | F[1] | **F[2]** | F[4] | **F[4]** |

| | |
|---|---|
| F[0] | 0 |
| F[1] | C1 |
| F[2] | Max (F[1], F[0]+C2) = Max(5, 0+1) = 5 = F[1] |
| F[3] | Max (F[2], F[1]+C3) = Max(5, 5+2) = 7 = F[1]+C3 |
| F[4] | Max (F[3], F[2]+C4) = Max(7, 5+10) = 15 = F[2]+C4 |
| F[5] | Max (F[4], F[3]+C5) = Max(15, 7+6) = 15 = F[4] |
| F[6] | Max (F[5], F[4]+C6) = Max(15, 15+2) = 17 = F[4]+C6 |

**The coins to be picked up are: C1 = 5, C4 = 10 and C6 = 2. The maximum value of the sum obtained is 17.**

# Example 2 for Coin-Row Problem

**Solve for the coin-row problem for the instance {3, 1, 5, 2, 5, 4, 2, 3}**

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| C | | 3 | 1 | 5 | 2 | 5 | 4 | 2 | 3 |
| F | 0 | 3 | 3 | 8 | 8 | 13 | 13 | 15 | **16** |
| History | | **C1** | - | **C3** | - | **C5** | - | C7 | **C8** |
| | | | F[1] | **F[1]** | F[3] | **F[3]** | **F[5]** | F[5] | **F[6]** |

| | |
|---|---|
| F[0] | 0 |
| F[1] | C1 |
| F[2] | Max (F[1], F[0]+C2) = Max(3 , 0 +1) = 3 = F[1] |
| F[3] | Max (F[2], F[1]+C3) = Max(3 , 3 +5) = 8 = F[1 ]+C3 |
| F[4] | Max (F[3], F[2]+C4) = Max(8 , 3 +2) = 8 = F[3] |
| F[5] | Max (F[4], F[3]+C5) = Max(8 , 8 +5) = 13 = F[3]+C5 |
| F[6] | Max (F[5], F[4]+C6) = Max(13, 8+4) = 13 = F[5] |
| F[7] | Max (F[6], F[5]+C7) = Max(13, 13+2) = 15 = F[5]+C7 |
| F[8] | Max (F[7], F[6]+C8) = Max(15, 13+3) = 16 = F[6]+C8 |

**The coins to be picked up are: C1 = 3, C3 = 5, C5 = 5 and C8 = 3.**
**The maximum value of the sum obtained is 16.**

# Integer 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.**

# 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 = CAGATCA
  - LCS(X, Y) = ATCA.

# 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

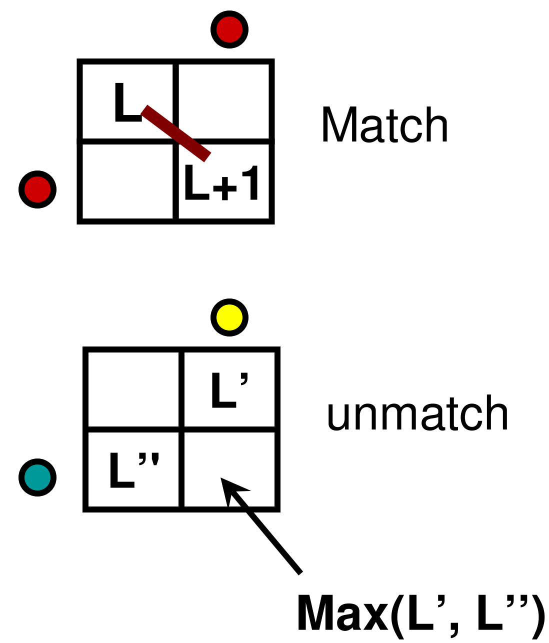
**Define: LCS[*i*][*j*] = Length of the LCS of sequence X[1…i] and Y[1…j]**
**Thus, LCS[i][0] = 0 for all i** **LCS[0][j] = 0 for all j**
**The goal is to find LCS[m][n]**

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

X = ATGACTATAA
Y = GACTAATA

# LCS Example 1 (1)

| | | 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 |



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 |

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

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