Dynamic Programming

All Pairs Shortest Paths | Transitive Closure

Longest Common Subsequence

Kazi Lutful Kabir

Spring 2023

Algorithmic Paradigms

- Greedy: Build up a global solution incrementally, myopically by optimizing some local criterion.

- Divide-and-conquer: Break up a problem into disjoint (non-overlapping) sub-problems, solve the sub-problems recursively, and then combine their solutions to form solution to the original problem. Brand-new sub-problems are generated at each step of the recursion.

- Dynamic programming: Break up a problem into a series of overlapping sub-problems and build up solutions to larger and larger sub-problems. Typically, same sub-problems are generated repeatedly.

Dynamic Programming (DP)

- DP is a method for solving certain kind of problems
- DP can be applied when the **solution of a problem includes solutions to subproblems**
- We need to find a recursive formula for the solution
- We can recursively solve subproblems, starting from the trivial case, and save their solutions in memory
- In the end we'll get the solution of the whole problem

Properties of a Problem that can be Solved with Dynamic Programming

- Simple Subproblems
  - We should be able to break the original problem to smaller subproblems that have the same structure
- Optimal Substructure of the Problems
  - The solution to the problem must be a composition of subproblem solutions
- Subproblem Overlap
  - Optimal subproblems to unrelated problems can contain subproblems in common

Optimal Substructure Property

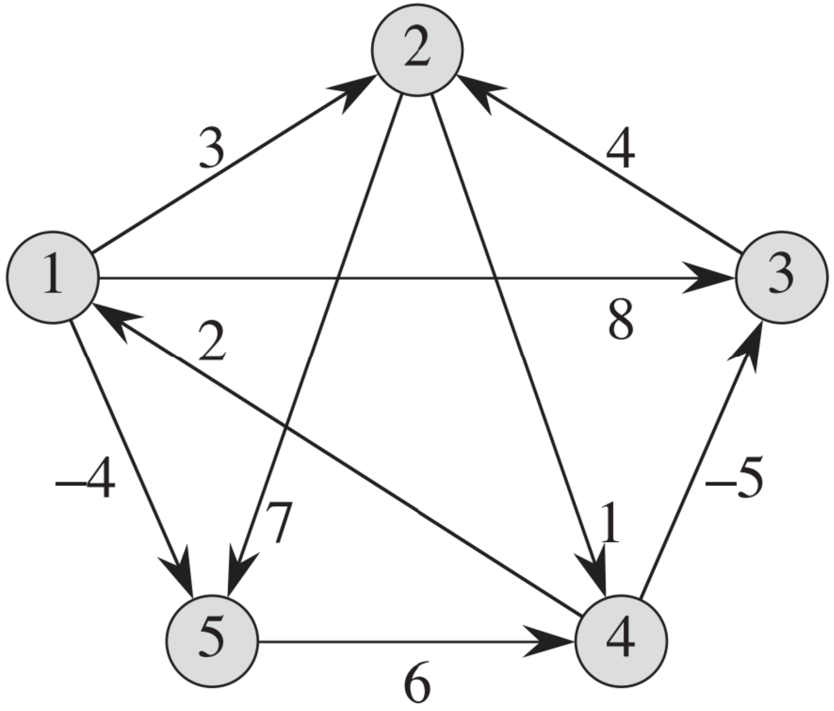
- A problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems.
- Whenever a problem exhibits optimal substructure, we have a good clue that dynamic programming might apply.
- Consequently, we must take care to ensure that the range of subproblems we consider includes those used in an optimal solution.
- We must also take care to ensure that the total number of distinct subproblems is a polynomial in the input size.

Steps

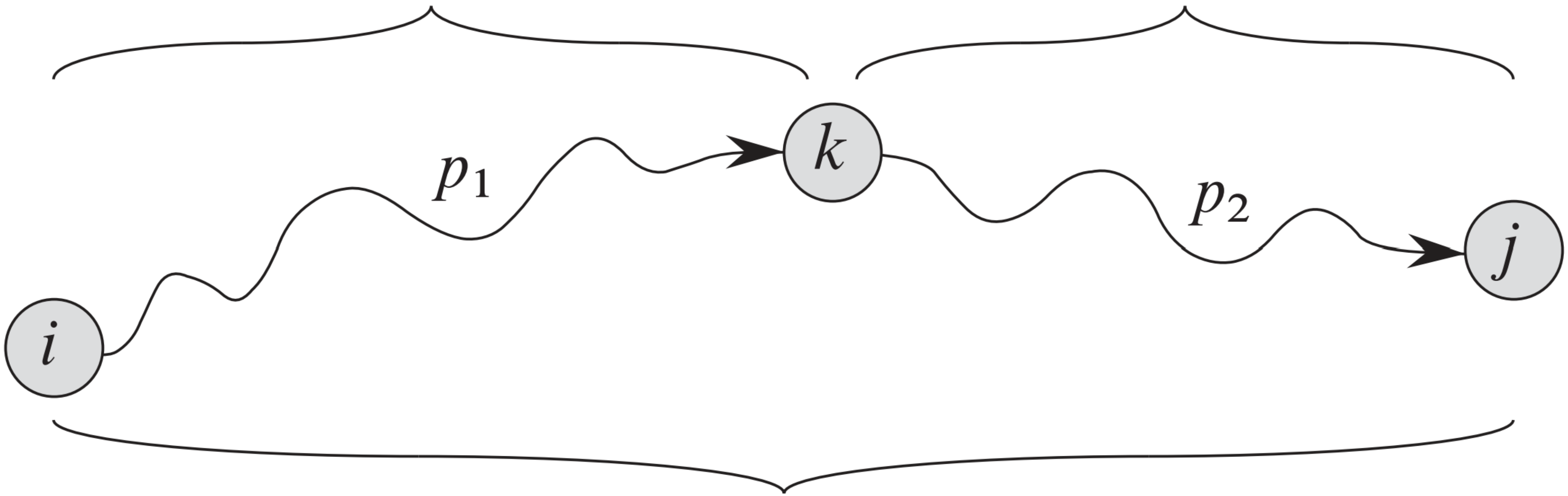
1. Find the optimal substructure property
2. Develop a recursive (can have iterative substitute) solution
3. Compute the optimal cost
4. Construct an optimal solution

All Pairs Shortest Paths : Floyd Warshall Algorithm

- **Problem:** Find the shortest distances between every pair of vertices in a given weighted directed Graph



Optimal Substructure Property



Building a Recursive Solution

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

Computing the Shortest Path: The Algorithm

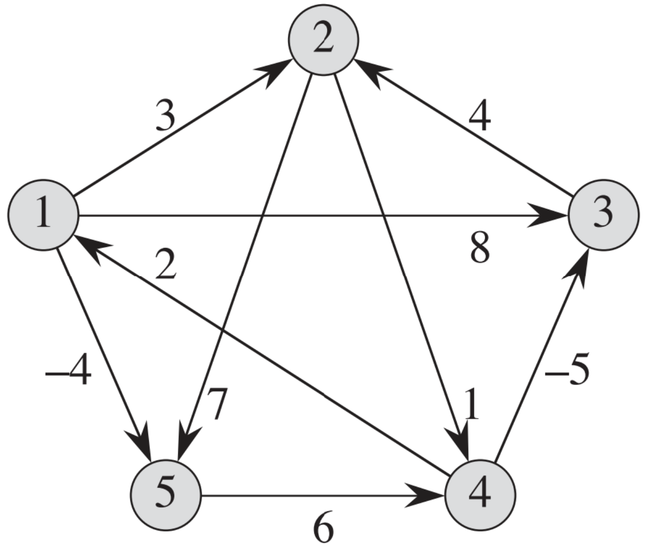
FLOYD-WARSHALL(W)

n = *Number of vertices*
$D^{(0)} = W$
for $k = 1$ **to** n
    let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
        for $j = 1$ **to** n
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Running Time: $\theta(n^3)$
Space Complexity: $\theta(n^2)$

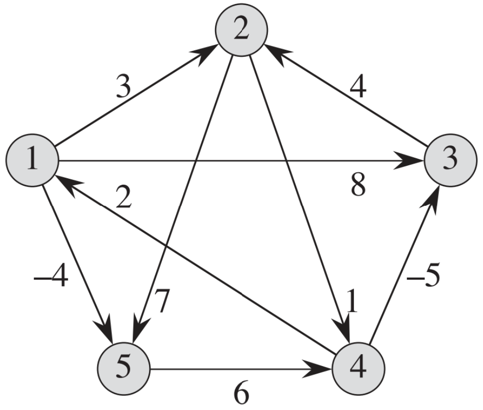
Constructing Shortest Path

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty , \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty . \end{cases}$$



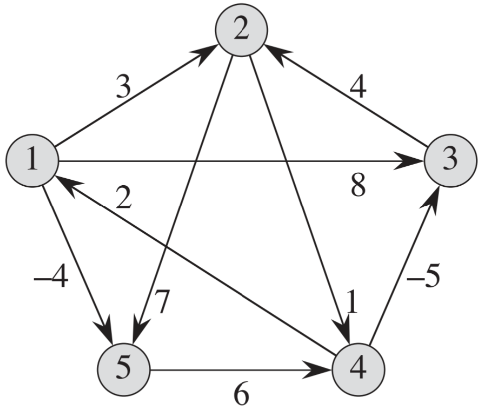
$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Adjacency Matrix for Weighted & Directed Graph



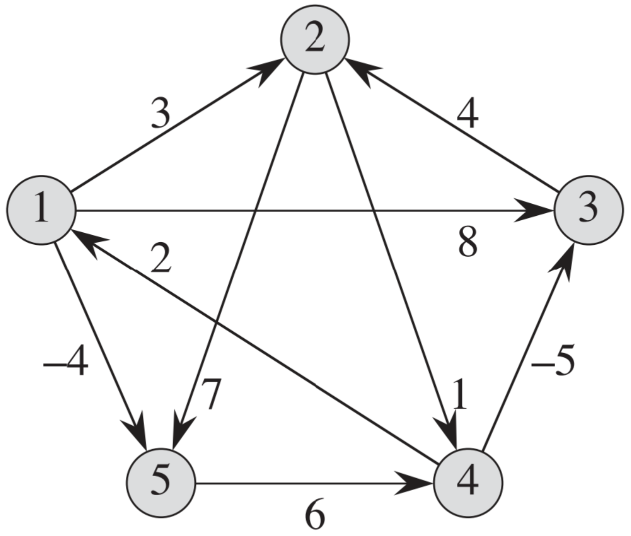
$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$



$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$



$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \mathrm{NIL} & 3 & 4 & 5 & 1 \\ 4 & \mathrm{NIL} & 4 & 2 & 1 \\ 4 & 3 & \mathrm{NIL} & 2 & 1 \\ 4 & 3 & 4 & \mathrm{NIL} & 1 \\ 4 & 3 & 4 & 5 & \mathrm{NIL} \end{pmatrix}$$

Transitive closure of a directed graph

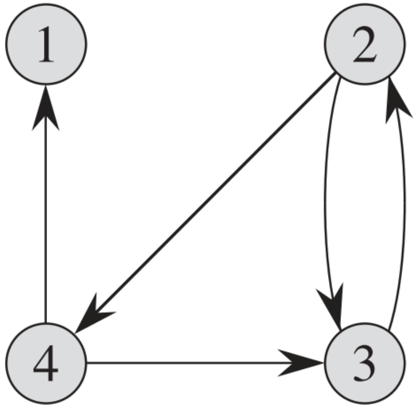
Given a directed graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$, we might wish to determine whether G contains a path from i to j for all vertex pairs $i, j \in V$. We define the ***transitive closure*** of G as the graph $G^* = (V, E^*)$, where

$$E^* = \{(i, j) : \text{there is a path from vertex } i \text{ to vertex } j \text{ in } G\} .$$

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E , \\ 1 & \text{if } i = j \text{ or } (i, j) \in E , \end{cases}$$

and for $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right) .$$



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

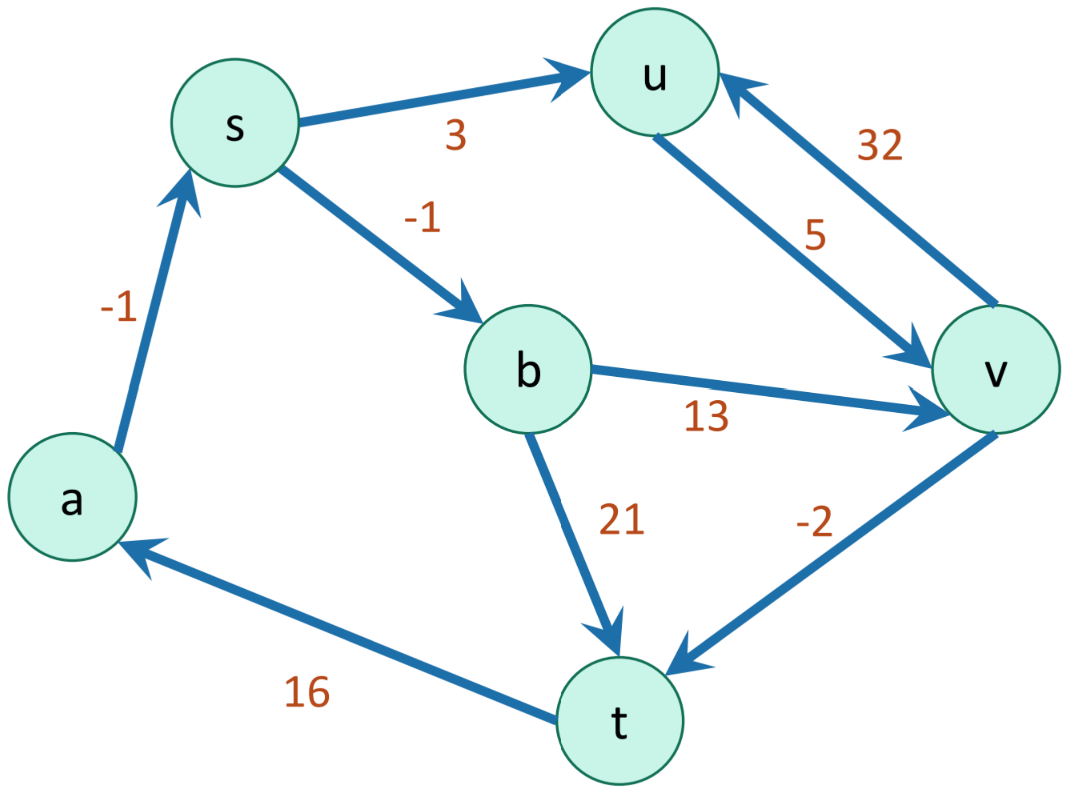
Running Time: θ(n³)
Space Complexity: θ(n²)

TRANSITIVE-CLOSURE(G)

$n = |G.V|$
let $T^{(0)} = (t_{ij}^{(0)})$ be a new $n \times n$ matrix
for $i = 1$ **to** n
  for $j = 1$ **to** n
    if $i == j$ or $(i, j) \in G.E$
      $t_{ij}^{(0)} = 1$
    else $t_{ij}^{(0)} = 0$
for $k = 1$ **to** n
  let $T^{(k)} = (t_{ij}^{(k)})$ be a new $n \times n$ matrix
  for $i = 1$ **to** n
    for $j = 1$ **to** n
      $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$
return $T^{(n)}$

Exercise for Practice

- Try Floyd-Warshall algorithm on the following graph (also find the transitive closure):



Longest Common Subsequence (LCS)

- Given two sequences

  $X = \langle x_1, x_2, \ldots, x_m \rangle$

  $Y = \langle y_1, y_2, \ldots, y_n \rangle$

  A subsequence of a given sequence is just the given sequence with zero or more elements left out

- A common subsequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ of X *and* Y
  - Z is a subsequence of both X and Y

- Example:

  X = A **B** **C** **B** D **A** B

  Y = **B** D **C** A **B** **A**

Goal: **Find the Longest Common Subsequence (LCS)**

Optimal Substructure Property of LCS

- The LCS problem has an *optimal substructure* property
  - solutions of subproblems are parts of the final solution
  - Subproblems: LCS of pairs of *prefixes* of X and Y

– An LCS of two sequences contains within it an LCS of prefixes of the two sequences.

Building the Solution

- Define $c[i, j]$ to be the length of an LCS of the sequences X_i and Y_j.
  - Goal: Find $c[m, n]$
  - Basis: $c[i, j] = 0$ if either $i = 0$ or $j = 0$
  - Recursion: How to define $c[i, j]$ recursively ?
- Finding an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$
  - If $x_m = y_n$, then we must find an LCS of X_{m-1} and Y_{n-1}.
    - Appending $x_m = y_n$ to this LCS yields an LCS of X and Y.
  - If $x_m \neq y_n$, then we must solve two subproblems:
    - Finding an LCS of X_{m-1} and Y
    - Finding an LCS of X and Y_{n-1}
    - Whichever of these two LCSs is longer is an LCS of X and Y.
- The recursive formula is

$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{if } i, j > 0 \text{ and } x[i] = y[j], \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x[i] \neq y[j] \end{cases}$$

Algorithm Pseudocode

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
        do c[i, 0] ← 0
for j ← 0 to n
        do c[0, j] ← 0
for i ← 1 to m
        do for j ← 1 to n
                do if x_i = y_j
                        then c[i, j] ← c[i − 1, j − 1] + 1
                             b[i, j] ← "↖"
                        else if c[i − 1, j] ≥ c[i, j − 1]
                                then c[i, j] ← c[i − 1, j]
                                     b[i, j] ← "↑"
                                else c[i, j] ← c[i, j − 1]
                                     b[i, j] ← "←"
return c and b
```

- The algorithm calculates the values of each entry of the array *c*[*m*, *n*].
- Each *c*[*i*, *j*] is calculated in constant time, and there are *m*·*n* elements in the array.
- So, the running time is *O*(*m*·*n*).

LCS Example

We'll see how LCS algorithm works on the following example:

X = ABCG

Y = BDCAG

LCS(X, Y) = BCG

X = A **B** **C** **G**

Y = **B** D **C** A **G**

LCS Example

ABCG
BDCAG

i	j	0	1	2	3	4	5
		y_j	**B**	**D**	**C**	**A**	**G**
0	x_i						
1	**A**						
2	**B**						
3	**C**						
4	**G**						

X = ABCG; $m = |X| = 4$
Y = BDCAG; $n = |Y| = 5$
Allocate array: $c[5, 4]$

LCS Example

ABCG
BDCAG

i	j	0	1	2	3	4	5
		y_j	**B**	**D**	**C**	**A**	**G**
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**G**	**0**					

for i = 0 to m $c[i, 0] = 0$

for j = 1 to n $c[0, j] = 0$

LCS Example

ABCG
BDCAG

	j	0	1	2	3	4	5
i		y_j	**B**	**D**	**C**	**A**	**G**
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**				
2	**B**	**0**					
3	**C**	**0**					
4	**G**	**0**					

if ($x_i == y_j$)
$c[i, j] = c[i-1, j-1] + 1$
else $c[i, j] = \max(c[i-1, j], c[i, j-1])$

LCS Example

ABCG
BDCAG

	j	0	1	2	3	4	5
i		y_j	**B**	**D**	**C**	**A**	**G**
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**G**	**0**					

if (x_i == y_j)
c[i, j] = c[i-1, j-1] + 1
else c[i, j] = max(c[i-1, j], c[i, j-1])

LCS Example

ABCG
BDCAG

i		j	0	1	2	3	4	5
			y_j	**B**	**D**	**C**	**A**	**G**
0	x_i		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	
2	**B**		**0**					
3	**C**		**0**					
4	**G**		**0**					

if ($x_i == y_j$)

c[i, j] = c[i-1, j-1] + 1

else c[i, j] = max(c[i-1, j], c[i, j-1])

LCS Example

ABCG

BDCAG

i	j	0	1	2	3	4	5
		y_j	**B**	**D**	**C**	**A**	**G**
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1** →	**1**
2	**B**	**0**					
3	**C**	**0**					
4	**G**	**0**					

if ($x_i == y_j$)

c[i, j] = c[i-1, j-1] + 1

else c[i, j] = max(c[i-1, j], c[i, j-1])

LCS Example

ABCG
BDCAG

	j	0	1	2	3	4	5
i		y_j	**B**	**D**	**C**	**A**	**G**
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**				
3	**C**	**0**					
4	**G**	**0**					

if ($x_i == y_j$)
$c[i, j] = c[i-1, j-1] + 1$
else $c[i, j] = \max(c[i-1, j], c[i, j-1])$

LCS Example

ABCG
BDCAG

i		j	0	1	2	3	4	5
			y_j	**B**	**D**	**C**	**A**	**G**
0	x_i		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	
3	**C**		**0**					
4	**G**		**0**					

if ($x_i == y_j$)
    c[i, j] = c[i-1, j-1] + 1
else c[i, j] = max(c[i-1, j], c[i, j-1])

LCS Example

ABCG

BDCAG

	j	0	1	2	3	4	5
i		y_j	**B**	**D**	**C**	**A**	**G**
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**1**
3	**C**	**0**					
4	**G**	**0**					

if (x_i == y_j)

c[i, j] = c[i-1, j-1] + 1

else c[i, j] = max(c[i-1, j], c[i, j-1])

LCS Example

ABCG
BDCAG

i		j	0	1	2	3	4	5
		y_j		**B**	**D**	**C**	**A**	**G**
0	x_i		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**1**
3	**C**		**0**	**1**	**1**			
4	**G**		**0**					

if ($x_i == y_j$)
 $c[i, j] = c[i-1, j-1] + 1$
else $c[i, j] = \max(c[i-1, j], c[i, j-1])$

LCS Example

ABCG
BDCAG

i		j	0	1	2	3	4	5
			y_j	**B**	**D**	**C**	**A**	**G**
0	x_i		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**1**
3	**C**		**0**	**1**	**1**	**2**		
4	**G**		**0**					

if ($x_i == y_j$)
    c[i, j] = c[i-1, j-1] + 1
else c[i, j] = max(c[i-1, j], c[i, j-1])

LCS Example

ABCG
BDCAG

i		j	0	1	2	3	4	5
			y_j	**B**	**D**	**C**	**A**	**G**
0	x_i		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**1**
3	**C**		**0**	**1**	**1**	**2** →	→ **2**	→ **2**
4	**G**		**0**					

if ($x_i == y_j$)
$c[i, j] = c[i-1, j-1] + 1$
else $c[i, j] = \max(c[i-1, j], c[i, j-1])$

LCS Example

ABCG
BDCAG

i		j	0	1	2	3	4	5
		y_j		**B**	**D**	**C**	**A**	**G**
0	x_i		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**1**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**G**		**0**	**1**				

if ($x_i == y_j$)

$c[i, j] = c[i-1, j-1] + 1$

else $c[i, j] = \max(c[i-1, j], c[i, j-1])$

LCS Example

ABCG
BDCAG

	j	0	1	2	3	4	5
i		y_j	**B**	**D**	**C**	**A**	**G**
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**1**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**G**	**0**	**1**	**1**	**2**	**2**	

if ($x_i == y_j$)

c[i, j] = c[i-1, j-1] + 1

else c[i, j] = max(c[i-1, j], c[i, j-1])

LCS Example

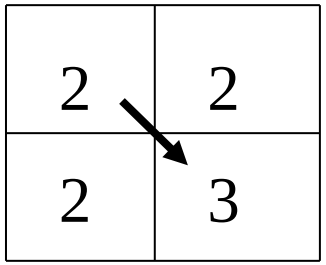
ABCG
BDCAG

i	j	0	1	2	3	4	5
		y_j	**B**	**D**	**C**	**A**	**G**
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**1**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**G**	**0**	**1**	**1**	**2**	**2**	**3**

if ($x_i == y_j$)
$c[i, j] = c[i-1, j-1] + 1$
else $c[i, j] = \max(c[i-1, j], c[i, j-1])$

How to Find Actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We can modify this algorithm to make it output an LCS of X and Y.
- Each $c[i, j]$ depends on $c[i\text{-}1, j\text{-}1]$, or $c[i\text{-}1, j]$ and $c[i, j\text{-}1]$.
- For each $c[i, j]$ we can say how it was acquired.



For example, here
$c[i, j] = c[i\text{-}1, j\text{-}1] +1 = 2+1=3$

How to Find Actual LCS

- Remember that

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- We can start from $c[m, n]$ and go backwards
- Whenever $c[i, j] = c[i\text{-}1, j\text{-}1]+1$, remember $x[i]$, because $x[i]$ is a part of LCS
- When i=0 or j=0 (we reached the beginning), output remembered letters in reverse order

Finding LCS: Example

i	j	0	1	2	3	4	5
		y_j	**B**	**D**	**C**	**A**	**G**
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**1**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**G**	**0**	**1**	**1**	**2**	**2**	**3**

Finding LCS: Example

	j	0	1	2	3	4	5
i		y_j	**B**	**D**	**C**	**A**	**G**
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**1**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**G**	**0**	**1**	**1**	**2**	**2**	**3**

LCS (reversed order): **G C B**

LCS (straight order): **B C G**

Another LCS Example

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Reference

Chapter-15 & 25, **Introduction to Algorithms** (3rd Ed.) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

STEVENS
INSTITUTE OF TECHNOLOGY
1870

THANK **YOU**

Stevens Institute of Technology