CE100 Algorithms and Programming II

Week-6 (Matrix Chain Order / LCS)

Spring Semester, 2021-2022

Download DOC, SLIDE, PPTX

Matrix Chain Order / Longest Common Subsequence

Outline

- Elements of Dynamic Programming
  - Optimal Substructure
  - Overlapping Subproblems

- Recursive Matrix Chain Order Memoization
    - Top-Down Approach
    - RMC
    - MemoizedMatrixChain
        - LookupC
    - Dynamic Programming vs Memoization Summary

- Dynamic Programming
  - Problem-2 : Longest Common Subsequence
    - Definitions
    - LCS Problem
    - Notations
    - Optimal Substructure of LCS
      - Proof Case-1
      - Proof Case-2
      - Proof Case-3

- A recursive solution to subproblems (inefficient)
- Computing the length of and LCS
  - LCS Data Structure for DP
  - Bottom-Up Computation
- Constructing and LCS
  - PRINT-LCS
  - Back-pointer space optimization for LCS length

- Most Common Dynamic Programming Interview Questions

Elements of Dynamic Programming

- When should we look for a DP solution to an optimization problem?
- Two key ingredients for the problem
  - Optimal substructure
  - Overlapping subproblems

DP Hallmark #1

- **Optimal Substructure**
  - A problem exhibits optimal substructure
    - if an optimal solution to a problem contains within it optimal solutions to subproblems
  - **Example:** *matrix-chain-multiplication*
    - Optimal parenthesization of $A_1 A_2 \ldots A_n$ that splits the product between A_k and A_{k+1}, contains within it **optimal soln's** to the problems of parenthesizing $A_1 A_2 \ldots A_k$ and $A_{k+1} A_{k+2} \ldots A_n$

Optimal Substructure

- Finding a suitable space of subproblems
  - Iterate on subproblem instances
  - **Example:** *matrix-chain-multiplication*
    - Iterate and look at the structure of optimal soln's to subproblems, sub-subproblems, and so forth
    - Discover that all subproblems consists of subchains of $\langle A_1, A_2, \dots, A_n \rangle$
    - Thus, the set of chains of the form $\langle A_i, A_{i+1}, \dots, A_j \rangle$ for $1 \leq i \leq j \leq n$
    - Makes a natural and reasonable space of subproblems

DP Hallmark #2

- **Overlapping Subproblems**
  - Total number of distinct subproblems should be **polynomial** in the input size
  - When a **recursive** algorithm revisits the same problem **over and over again**,
    - We say that the optimization problem has **overlapping subproblems**

Overlapping Subproblems

- **DP** algorithms typically take advantage of overlapping subproblems
  - by solving each problem once
  - then storing the solutions in a table
    - where it can be looked up when needed
  - using constant time per lookup

Overlapping Subproblems

- Recursive matrix-chain order

$$\text{RMC}(p, i, j)\{$$

$$\text{if } i = j \text{ then}$$

$$\text{return } 0$$

$$m[i, j] \leftarrow \infty$$

$$\text{for } k \leftarrow i \text{to } j - 1 \text{ do}$$

$$q \leftarrow \text{RMC}(p, i, k) + \text{RMC}(p, k + 1, j) + p_{i-1}p_kp_j$$

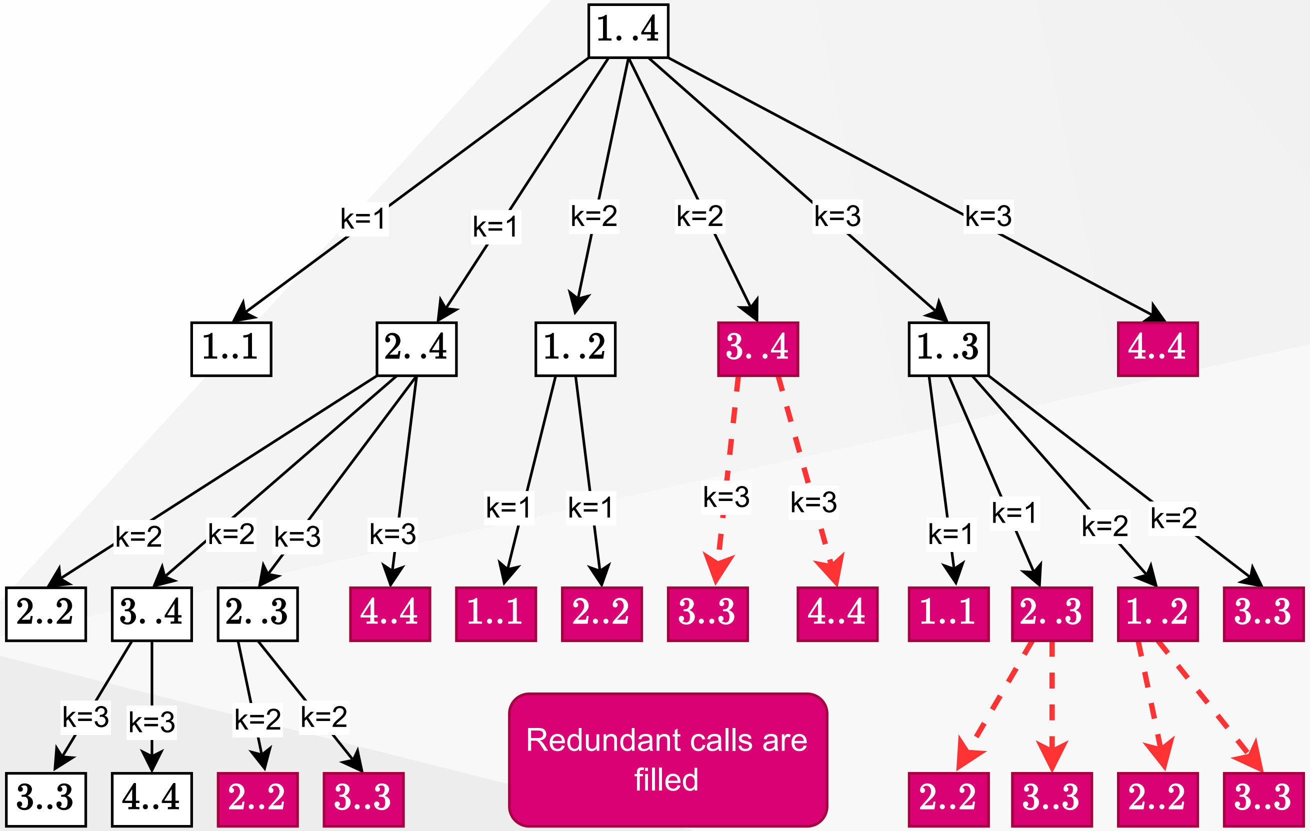
$$if\ q < m[i, j] \text{ then}$$

$$m[i, j] \leftarrow q$$

$$\text{return } m[i, j]\ \}$$

Direct Recursion: Inefficient!

- Recursion tree for $RMC(p, 1, 4)$
- Nodes are labeled with i and j values



Running Time of RMC

$T(1) \geq 1$

$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1)$ for $n > 1$

- For $i = 1, 2, \ldots, n$ each term $T(i)$ appears twice
  - Once as $T(k)$, and once as $T(n-k)$
- Collect $n-1$, 1's in the summation together with the front 1

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n$$

- Prove that $T(n) = \Omega(2n)$ using the **substitution method**

Running Time of RMC: Prove that $T(n) = \Omega(2n)$

- Try to show that $T(n) \geq 2^{n-1}$ (**by substitution**)
- **Base case:** $T(1) \geq 1 = 2^0 = 2^{1-1}$ for $n = 1$
- **Ind. Hyp.:**

$$T(i) \geq 2^{i-1} \text{ for all } i = 1, 2, \ldots, n-1 \text{ and } n \geq 2$$

$$T(n) \geq 2\sum_{i=1}^{n-1} 2^{i-1} + n$$

$$= 2\sum_{i=1}^{n-1} 2^{i-1} + n$$

$$= 2(2^{n-1} - 1) + n$$

$$= 2^{n-1} + (2^{n-1} - 2 + n)$$

$$\Rightarrow T(n) \geq 2^{n-1} \text{ Q.E.D.}$$

Running Time of RMC: $T(n) \geq 2^{n-1}$

- **Whenever**
  - a recursion tree for the natural recursive solution to a problem contains the same subproblem repeatedly
  - the total number of different subproblems is small
    - it is a good idea to see if $DP(Dynamic\ Programming)$ can be applied

Memoization

- Offers the efficiency of the usual DP approach while maintaining **top-down** strategy
- Idea is to **memoize** the natural, but inefficient, **recursive algorithm**

Memoized Recursive Algorithm

- Maintains an **entry** in a **table** for the soln to each subproblem
- Each table entry contains **a special value** to indicate that the entry has yet to be filled in
- When the subproblem is **first encountered** its solution is **computed** and then **stored** in the table
- Each **subsequent** time that the subproblem encountered the value stored in the table is simply **looked up** and **returned**

Memoized Recursive Matrix-chain Order

- Shaded subtrees are looked-up rather than recomputing

MemoizedMatrixChain(p)

$n \leftarrow length[p] - 1$

for $i \leftarrow 1$ to n do

  for $j \leftarrow 1$ to n do

    $m[i, j] \leftarrow \infty$

return LookupC$(p, 1, n) \Longrightarrow$

$\Longrightarrow$ LookupC(p, i, j)

if $m[i, j] = \infty$ then

  if $i = j$ then

    $m[i, j] \leftarrow 0$

  else

    for $k \leftarrow i$ to $j - 1$ do

      $q \leftarrow$ LookupC(p, i, k) + LookupC$(p, k + 1, j) + p_{i-1}p_kp_j$

      if $q < m[i, j]$ then

        $m[i, j] \leftarrow q$

return $m[i, j]$

Memoized Recursive Algorithm

- The approach assumes that
  - The set of **all possible subproblem parameters** are known
  - The relation between the **table positions** and **subproblems** is established
- Another approach is to memoize
  - by using **hashing** with subproblem parameters as key

Dynamic Programming vs Memoization Summary (1)

- Matrix-chain multiplication can be solved in $O(n^3)$ time
  - by either a top-down memoized recursive algorithm
  - or a bottom-up dynamic programming algorithm
- Both methods exploit the **overlapping subproblems** property
  - There are only $\Theta(n^2)$ different subproblems in total
  - Both methods **compute** the soln to **each problem once**
- **Without memoization** the natural **recursive** algorithm runs in **exponential time** since subproblems are solved repeatedly

Dynamic Programming vs Memoization Summary (2)

- **In general practice**
  - If all subproblems must be solved at once
    - a bottom-up **DP algorithm always outperforms** a top-down memoized algorithm by a constant factor
  - because, bottom-up **DP** algorithm
    - Has no overhead for recursion
    - Less overhead for maintaining the table
  - **DP: Regular** pattern of **table accesses** can be exploited to reduce the time and/or space requirements even further
  - **Memoized:** If some problems need not be solved at all, it has the advantage of avoiding solutions to those subproblems

Problem 3: Longest Common Subsequence

Definitions

- A **subsequence** of a given sequence is just the **given sequence** with **some elements** (possibly none) **left out**
- **Example:**
  - $X = \langle A, B, C, B, D, A, B \rangle$
  - $Z = \langle B, C, D, B \rangle$
    - Z is a subsequence of X

Problem 3: Longest Common Subsequence

Definitions

- **Formal definition:** Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a subsequence of X
  - if $\exists$ a **strictly increasing sequence** $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that $x_{i_j} = z_j$ for all $j = 1, 2, \ldots, k$, where $1 \leq k \leq m$
- **Example:** $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with the **index sequence** $\langle i_1, i_2, i_3, i_4 \rangle = \langle 2, 3, 5, 7 \rangle$

Problem 3: Longest Common Subsequence

Definitions

- If Z is a subsequence of both X and Y, we denote Z as a **common subsequence** of X and Y.
- **Example:**

$$X = \langle A, B^*, C^*, B, D, A^*, B \rangle$$
$$Y = \langle B^*, D, C^*, A^*, B, A \rangle$$

- $Z_1 = \langle B^*, C^*, A^* \rangle$ is a common subsequence (**of length 3**) of X and Y.
- **Two longest common subsequence (LCSs)** of X and Y?
  - $Z2 = \langle B, C, B, A \rangle$ of length 4
  - $Z3 = \langle B, D, A, B \rangle$ of length 4
    - *The optimal solution value = 4*

Longest Common Subsequence (LCS) Problem

- **LCS problem:** Given two sequences
  - $X = \langle x_1, x_2, \dots, x_m \rangle$ and
  - $Y = \langle y_1, y_2, \dots, y_n \rangle$, find the **LCS** of $X \& Y$
- **Brute force approach:**
  - Enumerate all subsequences of X
  - Check if each subsequence is also a subsequence of Y
  - Keep track of the **LCS**
  - What is the complexity?
  - There are 2^m subsequences of X
    - **Exponential runtime**

Notation

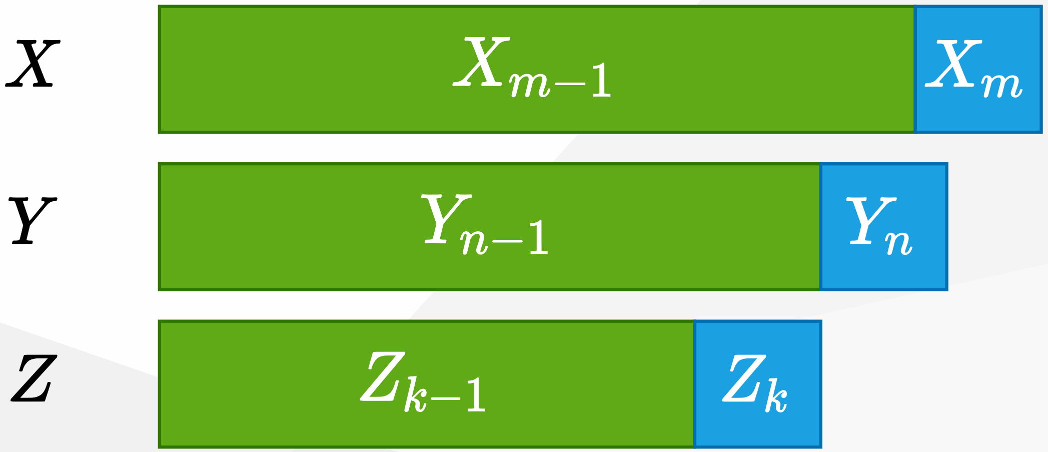
- **Notation:** Let X_i denote the i^{th} prefix of X
  - i.e. $X_i = \langle x_1, x_2, \ldots, x_i \rangle$
- **Example:**

$$X = \langle A, B, C, B, D, A, B \rangle$$

$$X_4 = \langle A, B, C, B \rangle$$
$$X_0 = \langle \rangle$$

Optimal Substructure of an LCS

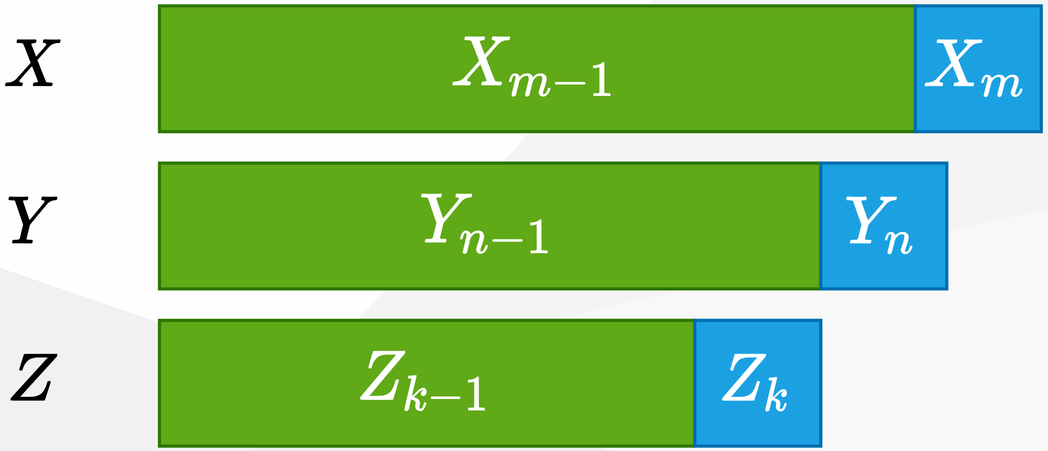
- Let $X =< x1, x2, \ldots, xm >$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ are given
- Let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be an **LCS** of X and Y



- **Question 1:** If $x_m = y_n$, how to define the optimal substructure?
  - We must have $z_k = x_m = y_n$ and
  - $Z_{k-1} = \text{LCS}(X_{m-1}, Y_{n-1})$

Optimal Substructure of an LCS

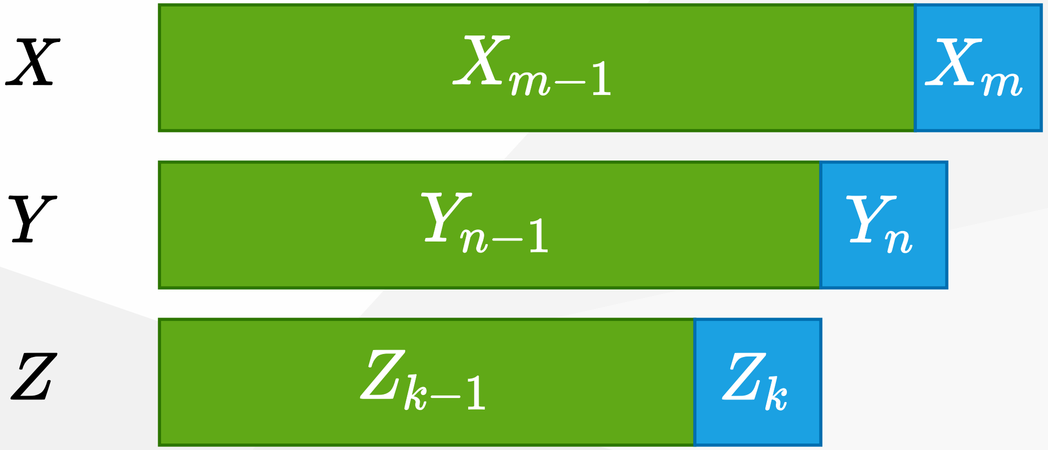
- Let $X =< x1, x2, \ldots, xm >$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ are given
- Let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be an **LCS** of X and Y



- **Question 2:** If $x_m \neq y_n$ and $z_k \neq x_m$, how to define the optimal substructure?
  - We must have $Z = \text{LCS}(X_{m-1}, Y)$

Optimal Substructure of an LCS

- Let $X =< x1, x2, \ldots, xm >$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ are given
- Let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be an **LCS** of X and Y



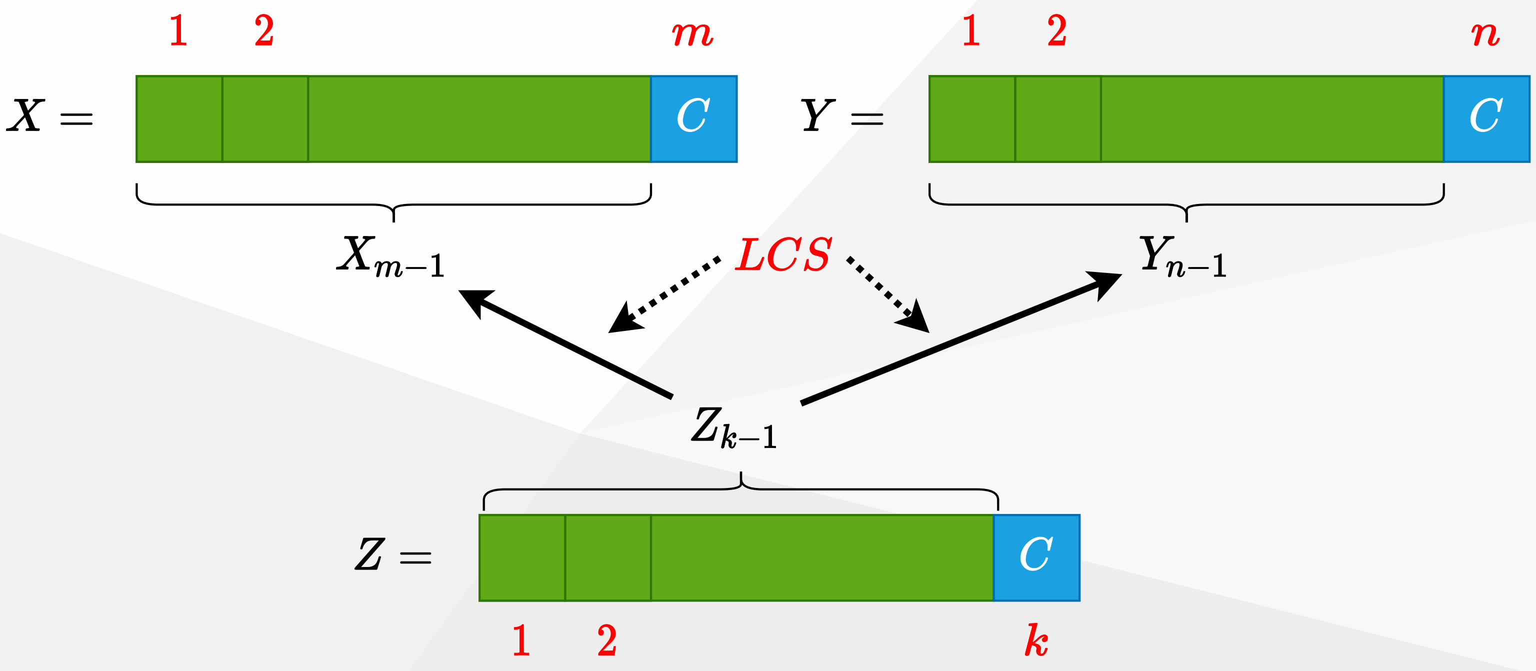
- **Question 3:** If $x_m \neq y_n$ and $z_k \neq y_n$, how to define the optimal substructure?
  - We must have $Z = \text{LCS}(X, Y_{n-1})$

Theorem: Optimal Substructure of an LCS

- Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and Y = <y1, y2, …, yn> are given
- Let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be an **LCS** of X and Y
- **Theorem:** Optimal substructure of an LCS:
  - If $x_m = y_n$
    - then $z_k = x_m = y_n$ and Z_{k-1} is an **LCS** of X_{m-1} and Y_{n-1}
  - If $x_m \neq y_n$ and $z_k \neq x_m$
    - then Z is an **LCS** of X_{m-1} and Y
  - If $x_m \neq y_n$ and $z_k \neq y_n$
    - then Z is an **LCS** of X and Y_{n-1}

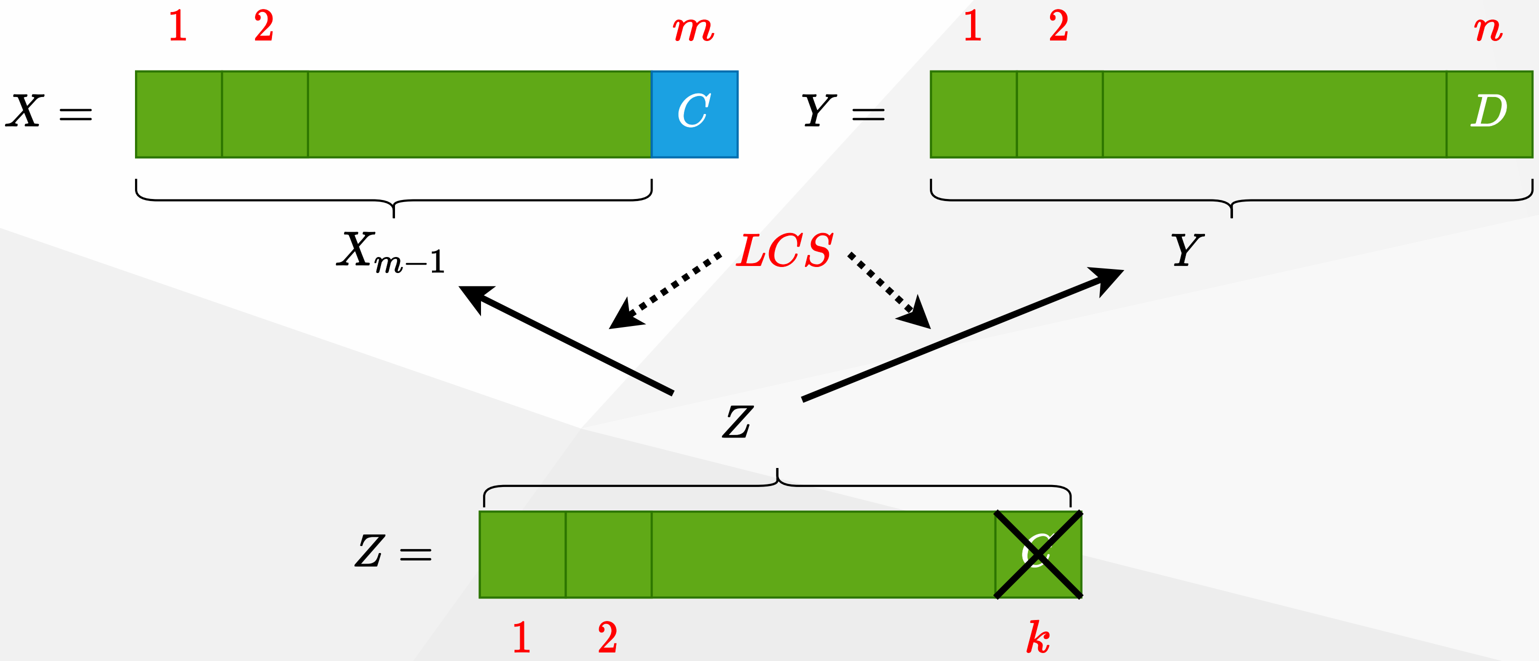
Optimal Substructure Theorem (case 1)

- If $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is an **LCS** of X_{m-1} and Y_{n-1}



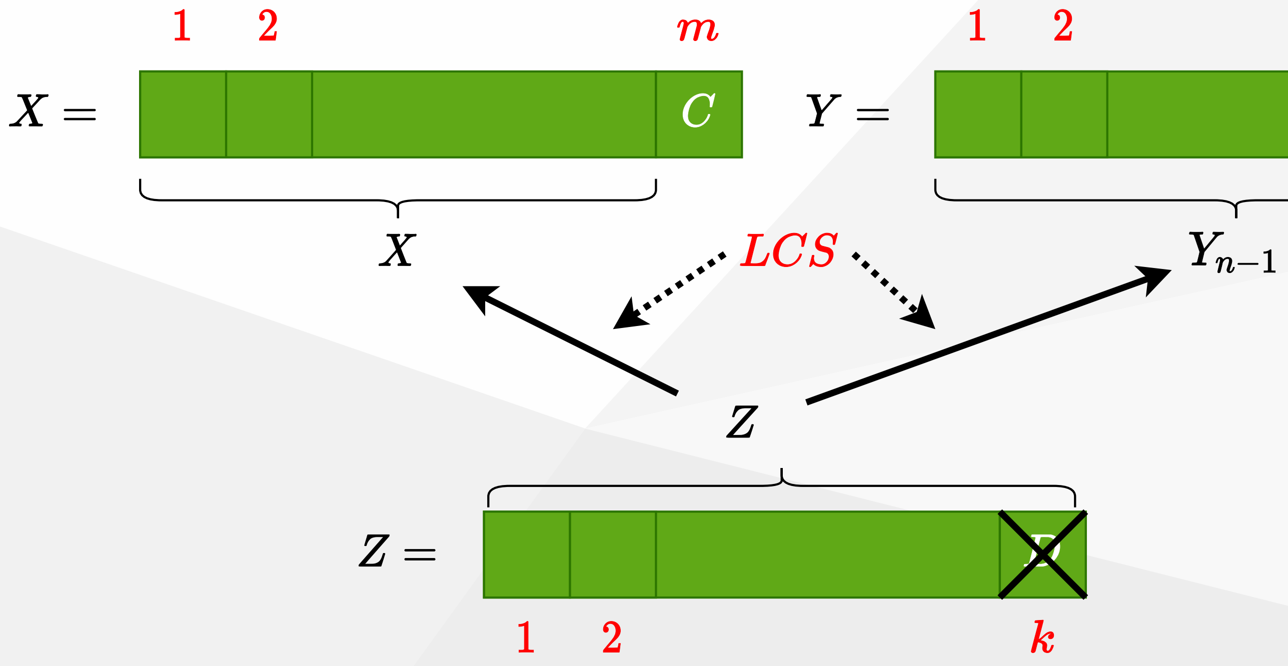
Optimal Substructure Theorem (case 2)

- If $x_m \neq y_n$ and $z_k \neq x_m$ then Z is an **LCS** of X_{m-1} and Y



Optimal Substructure Theorem (case 3)

- If $x_m \neq y_n$ and $z_k \neq y_n$ then Z is an **LCS** of X and Y_{n-1}



Proof of Optimal Substructure Theorem (case 1)

- If $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is an **LCS** of X_{m-1} and Y_{n-1}
- **Proof:** If $z_k \neq x_m = y_n$ then
  - we can append $x_m = y_n$ to Z to obtain a common subsequence of length $k+1 \implies$ **contradiction**
  - Thus, we must have $z_k = x_m = y_n$
  - Hence, the prefix Z_{k-1} is a **length-($k-1$) CS** of X_{m-1} and Y_{n-1}
- **We have to show that Z_{k-1}** is in fact an **LCS** of X_{m-1} and Y_{n-1}
- **Proof by contradiction:**
  - **Assume that** $\exists$ a CS W of X_{m-1} and Y_{n-1} with $|W| = k$
  - Then appending $x_m = y_n$ to W produces a **CS** of length $k+1$

Proof of Optimal Substructure Theorem (case 2)

- If $x_m \neq y_n$ and $z_k \neq x_m$ then Z is an **LCS** of X_{m-1} and Y
- **Proof :** If $z_k \neq x_m$ then Z is a CS of X_{m-1} and Y_n
  - **We have to show that** Z is in fact an **LCS** of X_{m-1} and Y_n
- **(Proof by contradiction)**
  - Assume that $\exists$ a CS W of X_{m-1} and Y_n with $|W| > k$
  - Then W would also be a CS of X and Y
  - Contradiction to the assumption that
    - Z is an LCS of X and Y with $|Z| = k$
- **Case 3:** Dual of the proof for (case 2)

A Recursive Solution to Subproblems

- Theorem implies that there are one or two subproblems to examine
- **if** $x_m = y_n$ **then**
  - we must solve the subproblem of finding an **LCS** of $X_{m-1} \& Y_{n-1}$
  - appending $x_m = y_n$ to this **LCS** yields an **LCS** of $X \& Y$
- **else**
  - we must solve **two subproblems**
    - finding an **LCS** of $X_{m-1} \& Y$
    - finding an **LCS** of $X \& Y_{n-1}$
  - longer of these two **LCS** s is an **LCS** of $X \& Y$
- **endif**

Recursive Algorithm (Inefficient)

$$
\begin{aligned}
&\text{LCS}(X, Y) \; \{ \\
&\quad m \leftarrow length[X] \\
&\quad n \leftarrow length[Y] \\
&\quad \text{if } x_m = y_n \text{ then} \\
&\qquad Z \leftarrow \text{LCS}(X_{m-1}, Y_{n-1}) \triangleright \text{solve one subproblem} \\
&\qquad \text{return } \langle Z, x_m = y_n \rangle \triangleright \text{append } x_m = y_n \text{ to } Z \\
&\quad else \\
&\qquad Z^{'} \leftarrow \text{LCS}(X_{m-1}, Y) \triangleright \text{solve two subproblems} \\
&\qquad Z^{''} \leftarrow \text{LCS}(X, Y_{n-1}) \\
&\qquad \text{return longer of } Z^{'} \text{ and } Z^{''} \\
&\}
\end{aligned}
$$

A Recursive Solution

- $c[i, j]$: length of an **LCS** of X_i and Y_j

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

Computing the Length of an LCS

- We can easily write an **exponential-time recursive algorithm** based on the given recurrence. $\Longrightarrow$ **Inefficient!**
- How many distinct subproblems to solve?
  - $\Theta(mn)$
- **Overlapping subproblems property:** Many subproblems share the same sub-subproblems.
  - **e.g.** Finding an **LCS** to $X_{m-1} \& Y$ and an **LCS** to $X \& Y_{n-1}$
  - has the sub-subproblem of finding an **LCS** to $X_{m-1} \& Y_{n-1}$
- Therefore, we can use **dynamic programming**.

Data Structures

- Let:
  - $c[i, j]$: length of an **LCS** of X_i and Y_j
  - $b[i, j]$: direction towards the table entry corresponding to the optimal subproblem solution chosen when computing $c[i, j]$.
  - Used to simplify the construction of an optimal solution at the end.
- Maintain the following tables:
  - $c[0 \ldots m, 0 \ldots n]$
  - $b[1 \ldots m, 1 \ldots n]$

Bottom-up Computation

- **Reminder:**

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

- How to choose the order in which we process $c[i,j]$ values?
- The values for $c[i-1,j-1]$, $c[i,j-1]$, and $c[i-1,j]$ must be computed before computing $c[i,j]$.

Bottom-up Computation

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

Need to process:

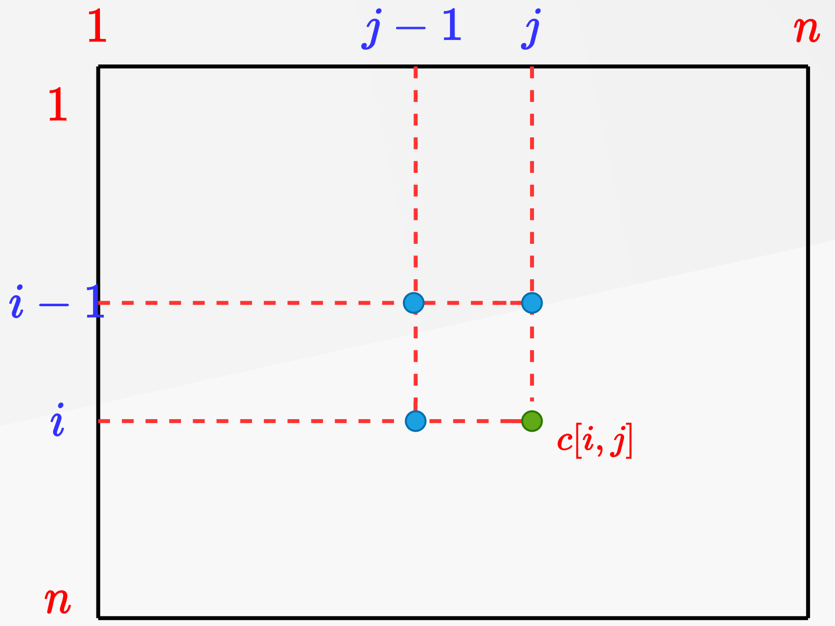
$c[i,j]$

after computing:

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

$c[i,j-1]$,

$c[i-1,j]$



Bottom-up Computation

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

$$\Downarrow$$

for $i \leftarrow 1$ to m

  for $j \leftarrow 1$ to n

    $\ldots$

    $\ldots$

    $c[i,j] = \cdots$

Computing the Length of an LCS

$$\frac{\text{Total Runtime} = \Theta(mn)}{\text{Total Space} = \Theta(mn)}$$

$LCS-LENGTH(X, Y)$

$m \leftarrow length[X]; n \leftarrow length[Y]$

for $i \leftarrow 0$ to m do $c[i, 0] \leftarrow 0$

for $j \leftarrow 0$ to n do $c[0, j] \leftarrow 0$

for $i \leftarrow 1$ to m do

  for $j \leftarrow 1$ to n do

    if $x_i = y_j$ then

      $c[i, j] \leftarrow c[i-1, j-1] + 1$

      $b[i, j] \leftarrow$ "$\nwarrow$"

    else if $c[i-1, j] \geq c[i, j-1]$

      $c[i, j] \leftarrow c[i-1, j]$

      $b[i, j] \leftarrow$ "$\uparrow$"

    else

      $c[i, j] \leftarrow c[i, j-1]$

      $b[i, j] \leftarrow$ "$\leftarrow$"

Computing the Length of an LCS-1

Operation of LCS-LENGTH on the sequences

$$X = \langle \overset{1}{A}, \overset{2}{B}, \overset{3}{C}, \overset{4}{B}, \overset{5}{D}, \overset{6}{A}, \overset{7}{B} \rangle$$

$$Y = \langle \overset{1}{B}, \overset{2}{D}, \overset{3}{C}, \overset{4}{A}, \overset{5}{B}, \overset{6}{A} \rangle$$

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0						
2 B	0						
3 C	0						
4 B	0						
5 D	0						
6 A	0						
7 B	0						

Computing the Length of an LCS-2

Operation of LCS-LENGTH on the sequences

$$X = \langle \overset{1}{A}, \overset{2}{B}, \overset{3}{C}, \overset{4}{B}, \overset{5}{D}, \overset{6}{A}, \overset{7}{B} \rangle$$

$$Y = \langle \overset{1}{B}, \overset{2}{D}, \overset{3}{C}, \overset{4}{A}, \overset{5}{B}, \overset{6}{A} \rangle$$

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0						
3 C	0						
4 B	0						
5 D	0						
6 A	0						
7 B	0						

Computing the Length of an LCS-3

Operation of LCS-LENGTH on the sequences

$$X = \langle \overset{1}{A}, \overset{2}{B}, \overset{3}{C}, \overset{4}{B}, \overset{5}{D}, \overset{6}{A}, \overset{7}{B} \rangle$$

$$Y = \langle \overset{1}{B}, \overset{2}{D}, \overset{3}{C}, \overset{4}{A}, \overset{5}{B}, \overset{6}{A} \rangle$$

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3 C	0						
4 B	0						
5 D	0						
6 A	0						
7 B	0						

Computing the Length of an LCS-4

Operation of LCS-LENGTH on the sequences

$$X = \langle \overset{1}{A}, \overset{2}{B}, \overset{3}{C}, \overset{4}{B}, \overset{5}{D}, \overset{6}{A}, \overset{7}{B} \rangle$$

$$Y = \langle \overset{1}{B}, \overset{2}{D}, \overset{3}{C}, \overset{4}{A}, \overset{5}{B}, \overset{6}{A} \rangle$$

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
$1\ A$	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
$2\ B$	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
$3\ C$	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
$4\ B$	0						
$5\ D$	0						
$6\ A$	0						
$7\ B$	0						

Computing the Length of an LCS-5

Operation of LCS-LENGTH on the sequences

$$X = \langle \overset{1}{A}, \overset{2}{B}, \overset{3}{C}, \overset{4}{B}, \overset{5}{D}, \overset{6}{A}, \overset{7}{B} \rangle$$

$$Y = \langle \overset{1}{B}, \overset{2}{D}, \overset{3}{C}, \overset{4}{A}, \overset{5}{B}, \overset{6}{A} \rangle$$

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3 C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4 B	0	$\nwarrow$ 1					
5 D	0						
6 A	0						
7 B	0						

Computing the Length of an LCS-6

Operation of LCS-LENGTH on the sequences

$$X = \langle \overset{1}{A}, \overset{2}{B}, \overset{3}{C}, \overset{4}{B}, \overset{5}{D}, \overset{6}{A}, \overset{7}{B} \rangle$$

$$Y = \langle \overset{1}{B}, \overset{2}{D}, \overset{3}{C}, \overset{4}{A}, \overset{5}{B}, \overset{6}{A} \rangle$$

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3 C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4 B	0	$\nwarrow$ 1	$\uparrow$ 1				
5 D	0						
6 A	0						
7 B	0						

Computing the Length of an LCS-7

Operation of LCS-LENGTH on the sequences

$$X = \langle \overset{1}{A}, \overset{2}{B}, \overset{3}{C}, \overset{4}{B}, \overset{5}{D}, \overset{6}{A}, \overset{7}{B} \rangle$$

$$Y = \langle \overset{1}{B}, \overset{2}{D}, \overset{3}{C}, \overset{4}{A}, \overset{5}{B}, \overset{6}{A} \rangle$$

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3 C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4 B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2			
5 D	0						
6 A	0						
7 B	0						

Computing the Length of an LCS-8

Operation of LCS-LENGTH on the sequences

$$X = \langle \overset{1}{A}, \overset{2}{B}, \overset{3}{C}, \overset{4}{B}, \overset{5}{D}, \overset{6}{A}, \overset{7}{B} \rangle$$

$$Y = \langle \overset{1}{B}, \overset{2}{D}, \overset{3}{C}, \overset{4}{A}, \overset{5}{B}, \overset{6}{A} \rangle$$

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
$1\ A$	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
$2\ B$	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
$3\ C$	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
$4\ B$	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2		
$5\ D$	0						
$6\ A$	0						
$7\ B$	0						

Computing the Length of an LCS-9

Operation of LCS-LENGTH on the sequences

$$X = \langle \overset{1}{A}, \overset{2}{B}, \overset{3}{C}, \overset{4}{B}, \overset{5}{D}, \overset{6}{A}, \overset{7}{B} \rangle$$

$$Y = \langle \overset{1}{B}, \overset{2}{D}, \overset{3}{C}, \overset{4}{A}, \overset{5}{B}, \overset{6}{A} \rangle$$

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3 C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4 B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	
5 D	0						
6 A	0						
7 B	0						

Computing the Length of an LCS-10

Operation of LCS-LENGTH on the sequences

$$X = \langle \overset{1}{A}, \overset{2}{B}, \overset{3}{C}, \overset{4}{B}, \overset{5}{D}, \overset{6}{A}, \overset{7}{B} \rangle$$

$$Y = \langle \overset{1}{B}, \overset{2}{D}, \overset{3}{C}, \overset{4}{A}, \overset{5}{B}, \overset{6}{A} \rangle$$

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3 C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4 B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5 D	0						
6 A	0						
7 B	0						

Computing the Length of an LCS-11

Operation of LCS-LENGTH on the sequences

$$X = \langle \overset{1}{A}, \overset{2}{B}, \overset{3}{C}, \overset{4}{B}, \overset{5}{D}, \overset{6}{A}, \overset{7}{B} \rangle$$

$$Y = \langle \overset{1}{B}, \overset{2}{D}, \overset{3}{C}, \overset{4}{A}, \overset{5}{B}, \overset{6}{A} \rangle$$

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
$1\ A$	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
$2\ B$	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
$3\ C$	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
$4\ B$	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
$5\ D$	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
$6\ A$	0						
$7\ B$	0						

Computing the Length of an LCS-12

Operation of LCS-LENGTH on the sequences

$$X = \langle \overset{1}{A}, \overset{2}{B}, \overset{3}{C}, \overset{4}{B}, \overset{5}{D}, \overset{6}{A}, \overset{7}{B} \rangle$$

$$Y = \langle \overset{1}{B}, \overset{2}{D}, \overset{3}{C}, \overset{4}{A}, \overset{5}{B}, \overset{6}{A} \rangle$$

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3 C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4 B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5 D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6 A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7 B	0						

Computing the Length of an LCS-13

Operation of LCS-LENGTH on the sequences

$$X = \langle \overset{1}{A}, \overset{2}{B}, \overset{3}{C}, \overset{4}{B}, \overset{5}{D}, \overset{6}{A}, \overset{7}{B} \rangle$$

$$Y = \langle \overset{1}{B}, \overset{2}{D}, \overset{3}{C}, \overset{4}{A}, \overset{5}{B}, \overset{6}{A} \rangle$$

- Running-time = $O(mn)$ since each table entry takes $O(1)$ time to compute

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3 C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4 B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5 D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6 A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7 B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

Computing the Length of an LCS-14

Operation of LCS-LENGTH on the sequences

$$X = \langle \overset{1}{A}, \overset{2}{B}, \overset{3}{C}, \overset{4}{B}, \overset{5}{D}, \overset{6}{A}, \overset{7}{B} \rangle$$

$$Y = \langle \overset{1}{B}, \overset{2}{D}, \overset{3}{C}, \overset{4}{A}, \overset{5}{B}, \overset{6}{A} \rangle$$

- Running-time = $O(mn)$ since each table entry takes $O(1)$ time to compute
- LCS of $X \& Y = \langle B, C, B, A \rangle$

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3 C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4 B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5 D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6 A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7 B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

Constructing an LCS

- The b table returned by **LCS-LENGTH** can be used to quickly construct an **LCS** of $X\&Y$
- Begin at $b[m, n]$ and trace through the table following arrows
- Whenever you encounter a "$\nwarrow$" in entry $b[i, j]$ it implies that $x_i = y_j$ is an element of **LCS**
- The elements of **LCS** are encountered in **reverse order**

Constructing an LCS

- The recursive procedure PRINT-LCS prints out LCS in proper order
- This procedure takes $O(m+n)$ time since at least one of i and j is decremented in each stage of the recursion

$$
\begin{aligned}
&\text{PRINT-LCS}(b, X, i, j) \\
&\quad \text{if } i = 0 \text{ or} j = 0 \text{ then} \\
&\quad \text{return} \\
&\quad \text{if } b[i, j] = \text{"} \nwarrow \text{" then} \\
&\qquad \text{PRINT-LCS}(b, X, i-1, j-1) \\
&\qquad \text{print } x_i \\
&\quad \text{else if } b[i, j] = \text{"} \uparrow \text{" then} \\
&\qquad \text{PRINT-LCS}(b, X, i-1, j) \\
&\quad \text{else} \\
&\qquad \text{PRINT-LCS}(b, X, i, j-1)
\end{aligned}
$$

- **The initial invocation:** $\text{PRINT-LCS}(b, X, length[X], length[Y])$

Do we really need the b table (back-pointers)?

- **Question:** From which neighbor did we expand to the highlighted cell?
- **Answer:** Upper-left neighbor,because $X[i] = Y[j]$.

$\downarrow i/j \rightarrow$	$0\ y_j$	$1\ B$	$2\ D$	$3\ C$	$4\ A$	$5\ B$	$6\ A$
$0\ x_i$	0	0	0	0	0	0	0
$1\ A$	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
$2\ B$	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
$3\ C$	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
$4\ B$	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
$5\ D$	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
$6\ A$	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
$7\ B$	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

Do we really need the b table (back-pointers)?

- **Question:** From which neighbor did we expand to the highlighted cell?
- **Answer:** Left neighbor, because $X[i] \neq Y[j]$ and $LCS[i, j-1] > LCS[i-1, j]$.

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3 C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4 B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5 D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6 A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7 B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

Do we really need the b table (back-pointers)?

- **Question:** From which neighbor did we expand to the highlighted cell?
- **Answer:** Upper neighbor,because $X[i] \neq Y[j]$ and $LCS[i, j-1] = LCS[i-1, j]$. *(See pseudo-code to see how ties are handled.)*

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3 C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4 B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5 D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6 A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7 B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

Improving the Space Requirements

- We can eliminate the b table altogether
  - each $c[i, j]$ entry depends only on 3 other c table entries: $c[i-1, j-1]$, $c[i-1, j]$ and $c[i, j-1]$
- Given the value of $c[i, j]$:
  - We can determine in $O(1)$ time which of these 3 values was used to compute $c[i, j]$ without inspecting table b
  - We save $\Theta(mn)$ space by this method
  - However, space requirement is still $\Theta(mn)$ since we need $\Theta(mn)$ space for the c table anyway

What if we store the last 2 rows only?

- To compute $c[i, j]$, we only need $c[i-1, j-1]$, $c[i-1, j]$,and $c[i-1, j-1]$
- So, we can store only the last two rows.

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3 C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4 B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5 D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6 A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7 B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

What if we store the last 2 rows only?

- To compute $c[i, j]$, we only need $c[i-1, j-1]$, $c[i-1, j]$, and $c[i-1, j-1]$
- So, we can store only the last two rows.

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow 0$	$\uparrow 0$	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\nwarrow 1$
2 B	0	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$
3 C	0	$\uparrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\uparrow 2$	$\uparrow 2$
4 B	0	$\nwarrow 1$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\leftarrow 3$
5 D	0	$\uparrow 1$	$\nwarrow 2$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\uparrow 3$
6 A	0	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\uparrow 3$	$\nwarrow 4$
7 B	0	$\nwarrow 1$	$\uparrow 2$	$\uparrow 3$	$\uparrow 3$	$\nwarrow 4$	$\uparrow 4$

What if we store the last 2 rows only?

- To compute $c[i, j]$, we only need $c[i-1, j-1]$, $c[i-1, j]$, and $c[i-1, j-1]$
- So, we can store only the last two rows.
- This reduces space complexity from $\Theta(mn)$ to $\Theta(n)$.
- Is there a problem with this approach?

$\downarrow i/j \rightarrow$	$0 y_j$	$1\ B$	$2\ D$	$3\ C$	$4\ A$	$5\ B$	$6\ A$
$0 x_i$	0	0	0	0	0	0	0
$1\ A$	0	$\uparrow 0$	$\uparrow 0$	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\nwarrow 1$
$2\ B$	0	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$
$3\ C$	0	$\uparrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\uparrow 2$	$\uparrow 2$
$4\ B$	0	$\nwarrow 1$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\leftarrow 3$
$5\ D$	0	$\uparrow 1$	$\nwarrow 2$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\uparrow 3$
$6\ A$	0	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\uparrow 3$	$\nwarrow 4$
$7\ B$	0	$\nwarrow 1$	$\uparrow 2$	$\uparrow 3$	$\uparrow 3$	$\nwarrow 4$	$\uparrow 4$

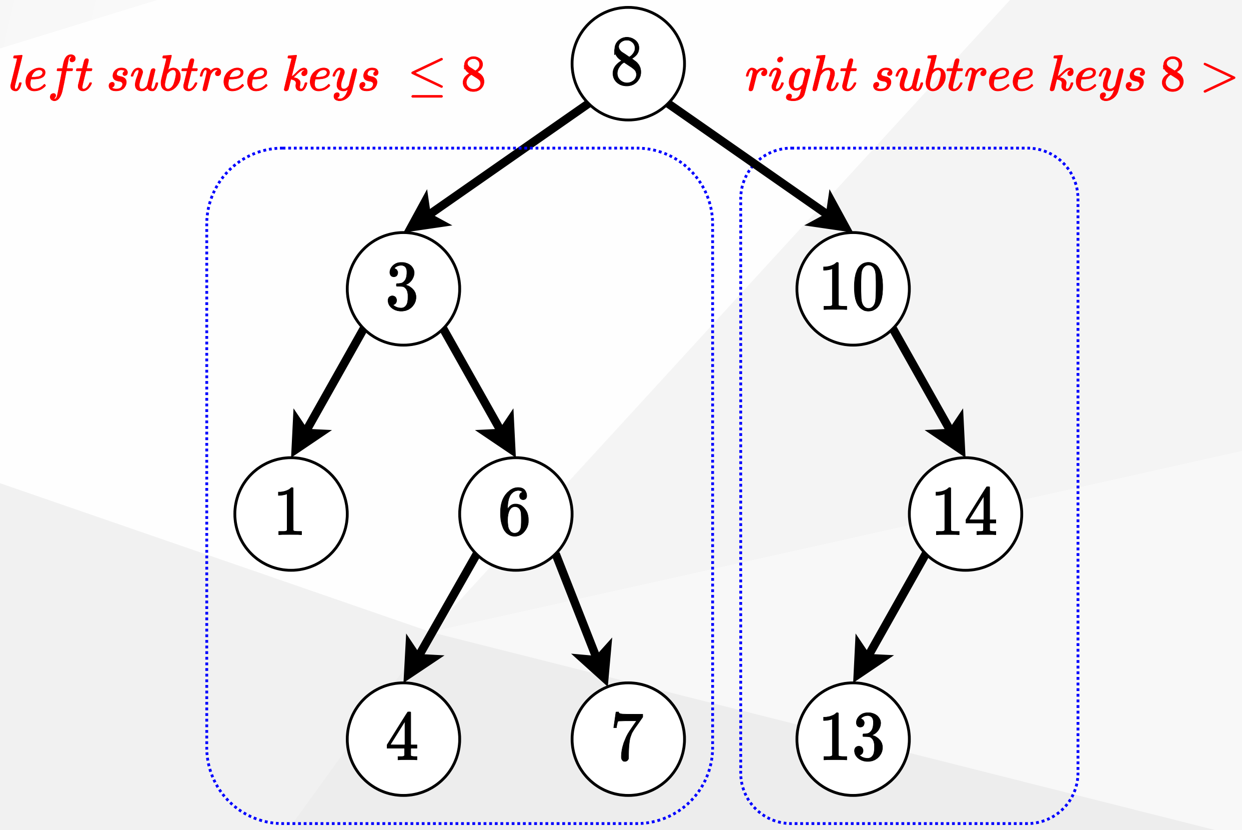
What if we store the last 2 rows only?

- **Is there a problem with this approach?**
  - We cannot construct the optimal solution because we cannot backtrace anymore.
  - This approach works if we only need the length of an LCS, not the actual LCS.

$\downarrow i/j \rightarrow$	$0 y_j$	1 B	2 D	3 C	4 A	5 B	6 A
$0 x_i$	0	0	0	0	0	0	0
1 A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2 B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3 C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4 B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5 D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6 A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7 B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

Problem 4 Optimal Binary Search Tree

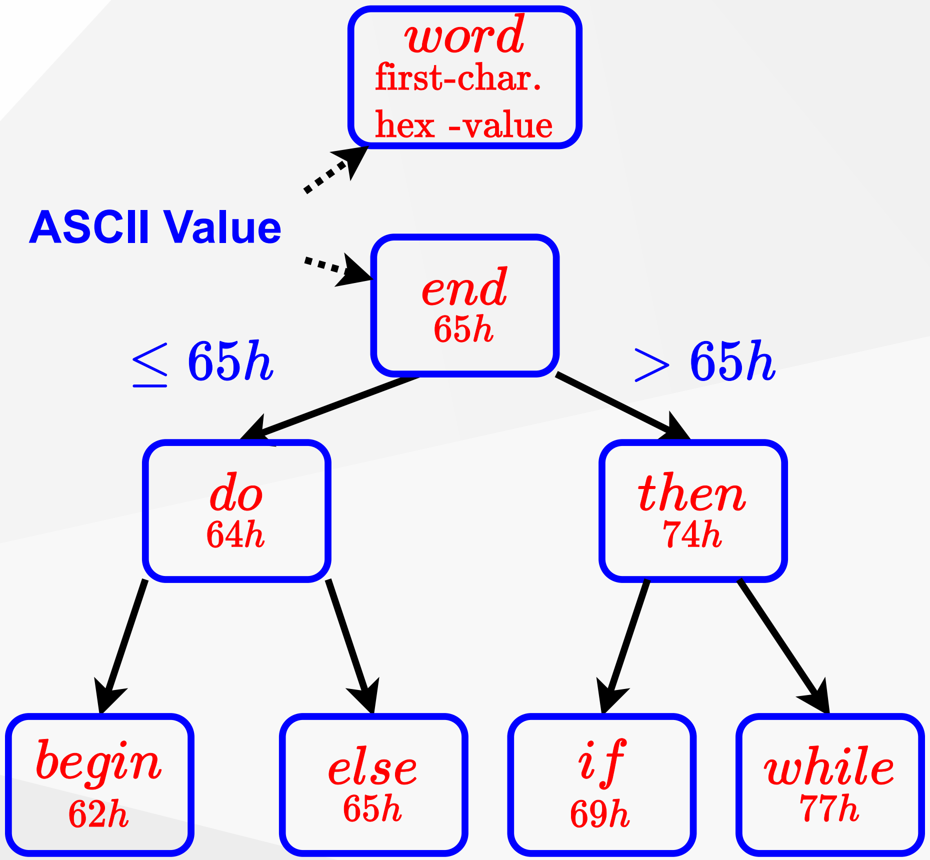
Reminder: Binary Search Tree (BST)



This property holds for all nodes

Binary Search Tree Example

- **Example:** English-to-French translation
  - Organize (English, French) word pairs in a BST
    - **Keyword:** English word
    - **Satellite Data:** French word
- We can search for an English word (node key) efficiently, and return the corresponding French word (satellite data).



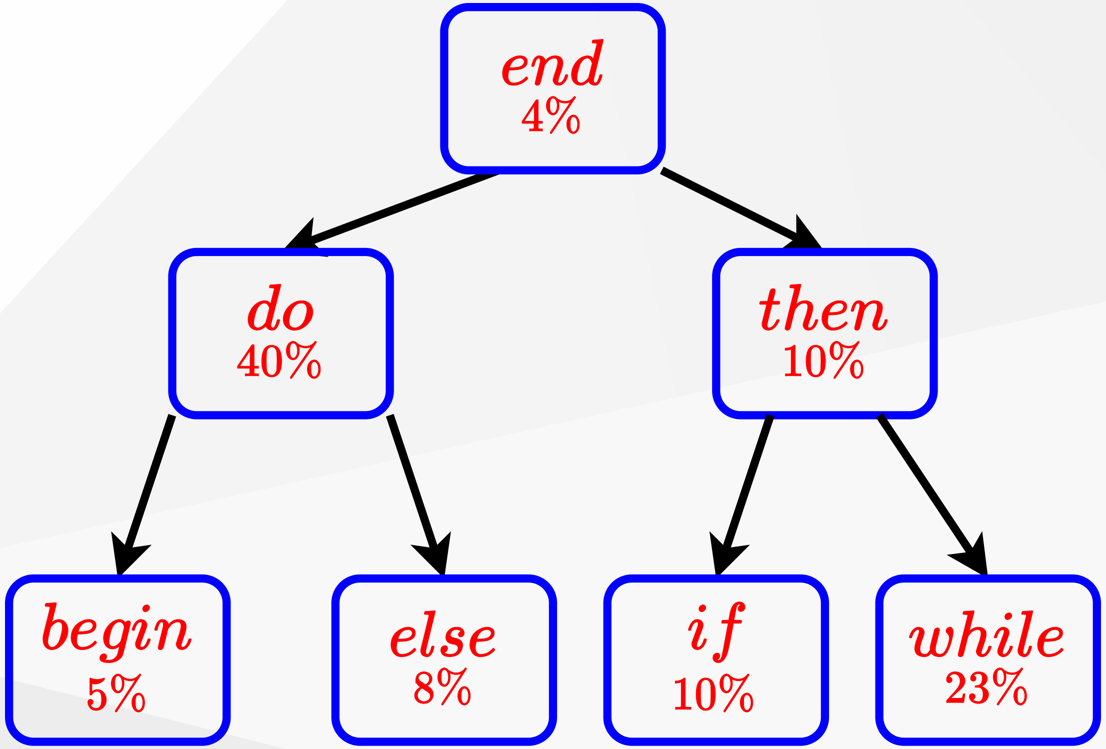
ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Binary Search Tree Example

Suppose we know the frequency of each keyword in texts:

$$\underbrace{begin}_{5\%}, \underbrace{do}_{40\%}, \underbrace{else}_{8\%}, \underbrace{end}_{4\%}, \underbrace{if}_{10\%}, \underbrace{then}_{10\%}, \underbrace{while}_{23\%},$$

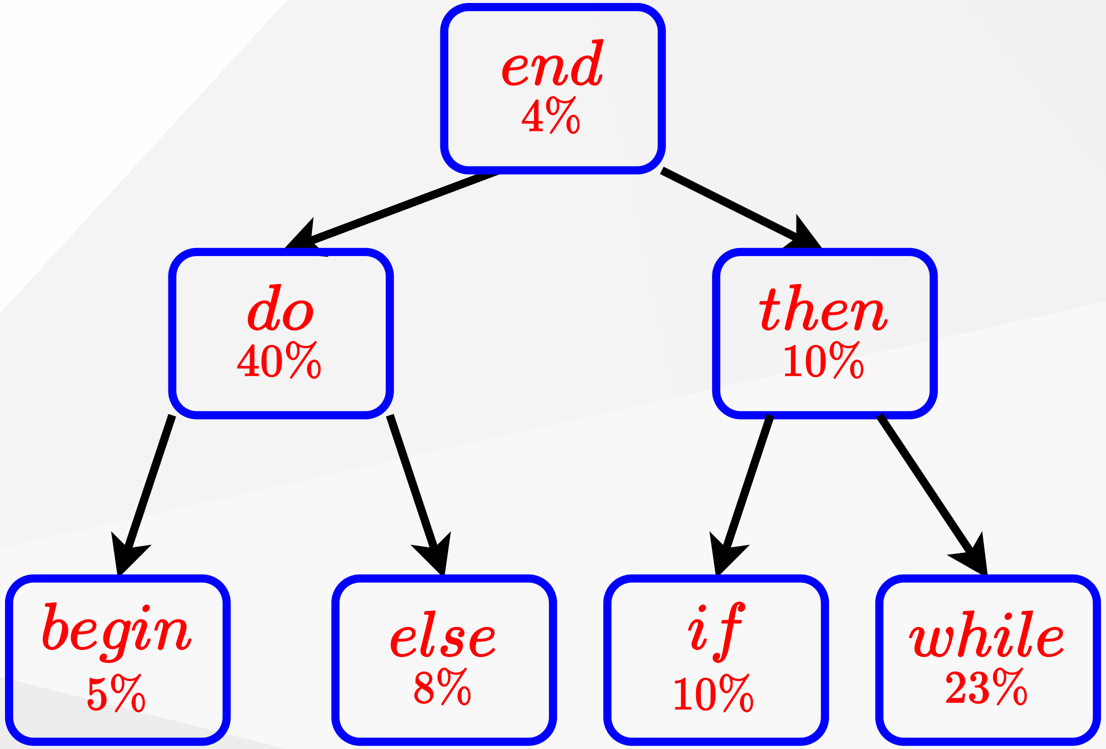


Cost of a Binary Search Tree

Example: If we search for keyword **"while"**, we need
to access 3 nodes. So, 23 of the queries will have cost of 3.

$$\text{Total Cost} = \sum_i (\text{depth}(i) + 1)\text{freq}(i)$$

$$= 1 \times 0.04 + 2 \times 0.4 +$$
$$2 \times 0.1 + 3 \times 0.05 +$$
$$3 \times 0.08 + 3 \times 0.1 +$$
$$3 \times 0.23$$
$$= 2.42$$

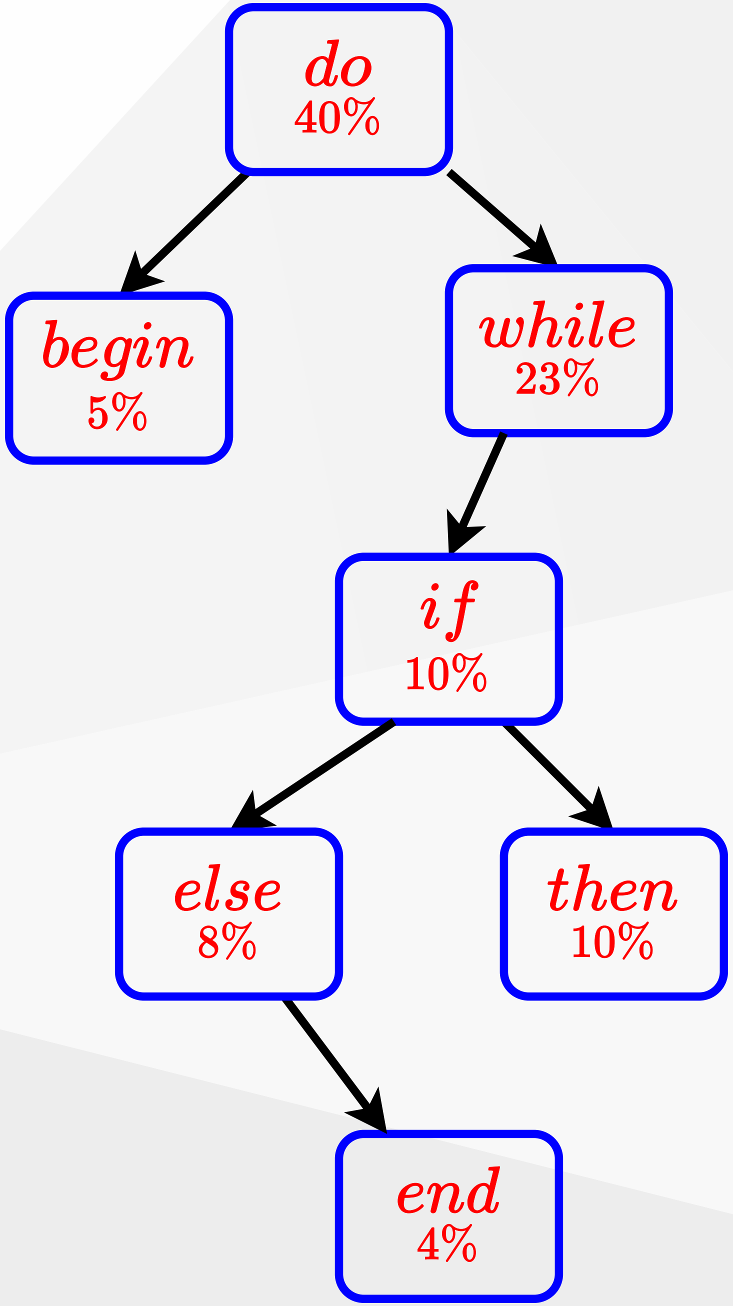


Cost of a Binary Search Tree

Example: If we search for keyword **"while"**, we need to access 3 nodes. So, 23 of the queries will have cost of 3.

$$
\begin{aligned}
\text{Total Cost} &= \sum_i (\text{depth}(i)+1)\text{freq}(i) \\
&= 1 \times 0.4 + 2 \times 0.05 + 2 \times 0.23 + \\
&3 \times 0.1 + 4 \times 0.08 + \\
&4 \times 0.1 + 5 \times 0.04 \\
&= 2.18
\end{aligned}
$$

- This is in fact an optimal BST.



Optimal Binary Search Tree Problem

- **Given:**
  - A collection of n keys $K_1 < K_2 < \dots K_n$ to be stored in a **BST**.
  - The corresponding p_i values for $1 \leq i \leq n$
    - p_i: probability of searching for key K_i
- **Find:**
  - An **optimal BST** with minimum total cost:

$$\text{Total Cost} = \sum_i (\text{depth}(i) + 1)\text{freq}(i)$$

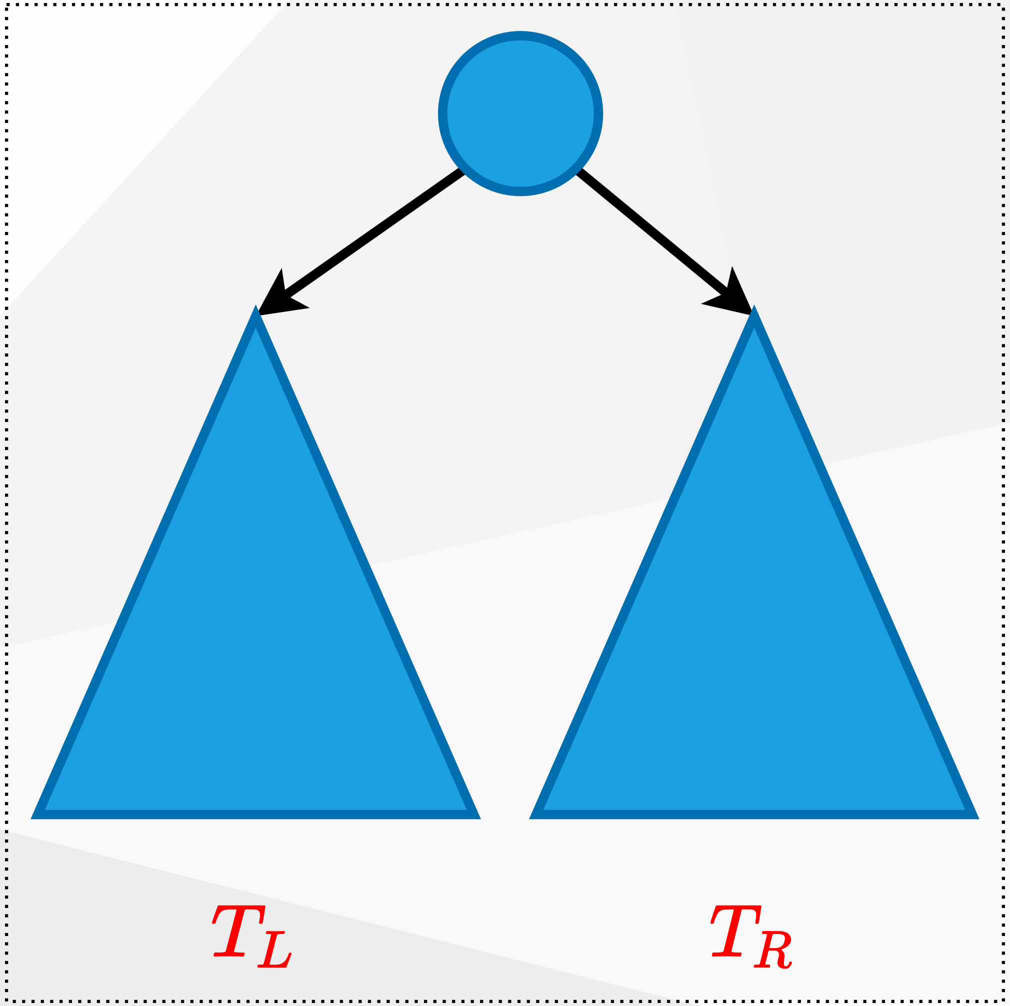
- **Note:** The BST will be static. Only search operations will be performed. No insert, no delete, etc.

Cost of a Binary Search Tree

- **Lemma 1**: Let Tij be a BST containing keys $K_i < K_{i+1} < \cdots < K_j$. Let T_L and T_R be the left and right subtrees of T. Then we have:

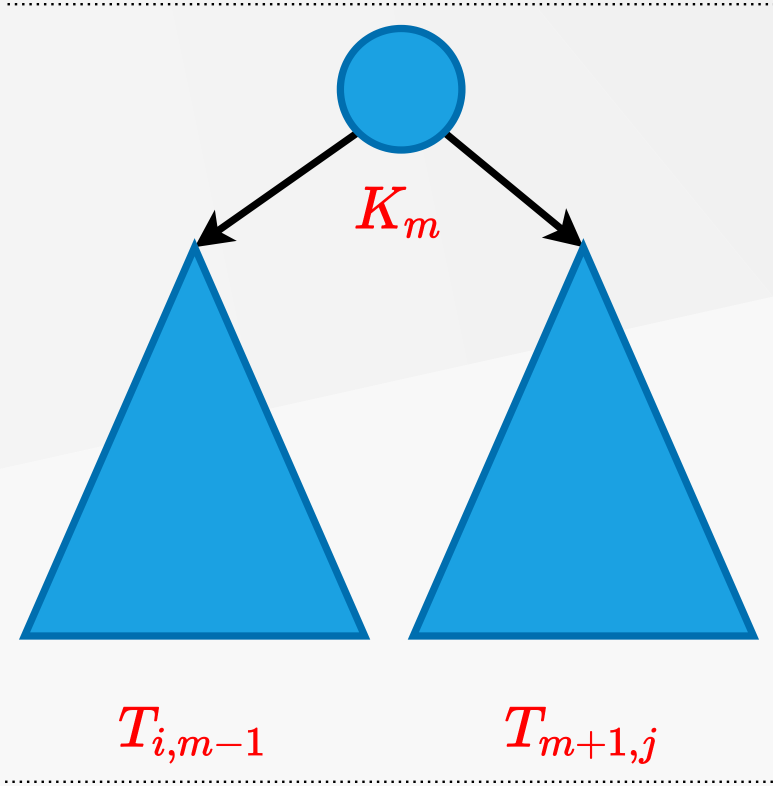
$$\text{cost}(T_{ij}) = \text{cost}(T_L) + \text{cost}(T_R) + \sum_{h=i}^{j} p_h$$

Intuition: When we add the root node, the depth of each node in T_L and T_R increases by 1. So, the cost of node h increases by p_h. In addition, the cost of root node r is p_r. That's why, we have the last term at the end of the formula above.



Optimal Substructure Property

- **Lemma 2:** Optimal substructure property
  - Consider an optimal **BST** T_{ij} for keys $K_i < K_{i+1} < \cdots < K_j$
  - Let K_m be the key at the root of T_{ij}
- **Then:**
  - $T_{i,m-1}$ is an **optimal BST** for subproblem containing keys:
    - $K_i < \cdots < K_{m-1}$
  - $T_{m+1,j}$ is an **optimal BST** for subproblem containing keys:
    - $K_{m+1} < \cdots < K_j$



$$\text{cost}(T_{ij}) = \text{cost}(T_{i,m-1}) + \text{cost}(T_{m+1,j}) + \sum_{h=i}^{j} p_h$$

Recursive Formulation

- **Note:** *We don't know which root vertex leads to the minimum total cost. So, we need to try each vertex m, and choose the one with minimum total cost.*
- $c[i, j]$: cost of an optimal BST T_{ij} for the subproblem $K_i < \cdots < K_j$

$$c[i, j] = \begin{cases} 0 & \text{if } i > j \\ \min\limits_{i \le r \le j} \{c[i, r-1] + c[r+1, j] + P_{ij}\} & \text{otherwise} \end{cases}$$

$$\text{where } P_{ij} = \sum_{h=i}^{j} p_h$$

Bottom-up computation

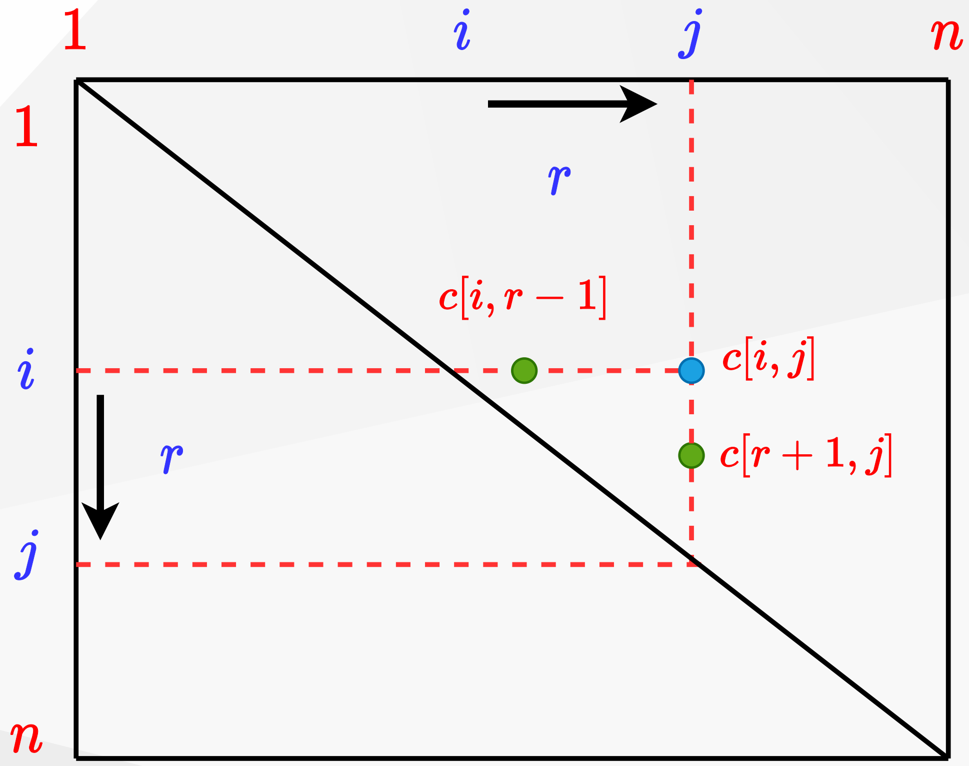
$$c[i,j] = \begin{cases} 0 & \text{if } i > j \\ \min\limits_{i \le r \le j} \{c[i, r-1] + c[r+1, j] + P_{ij}\} & \text{otherwise} \end{cases}$$

- How to choose the order in which we process $c[i,j]$ values?
- Before computing $c[i,j]$, we have to make sure that the values for $c[i, r-1]$ and $c[r+1, j]$ have been computed for all r.

Bottom-up computation

$$c[i,j] = \begin{cases} 0 & \text{if } i > j \\ \min\limits_{i \le r \le j} \{c[i, r-1] + c[r+1, j] + P_{ij}\} & \text{otherwise} \end{cases}$$

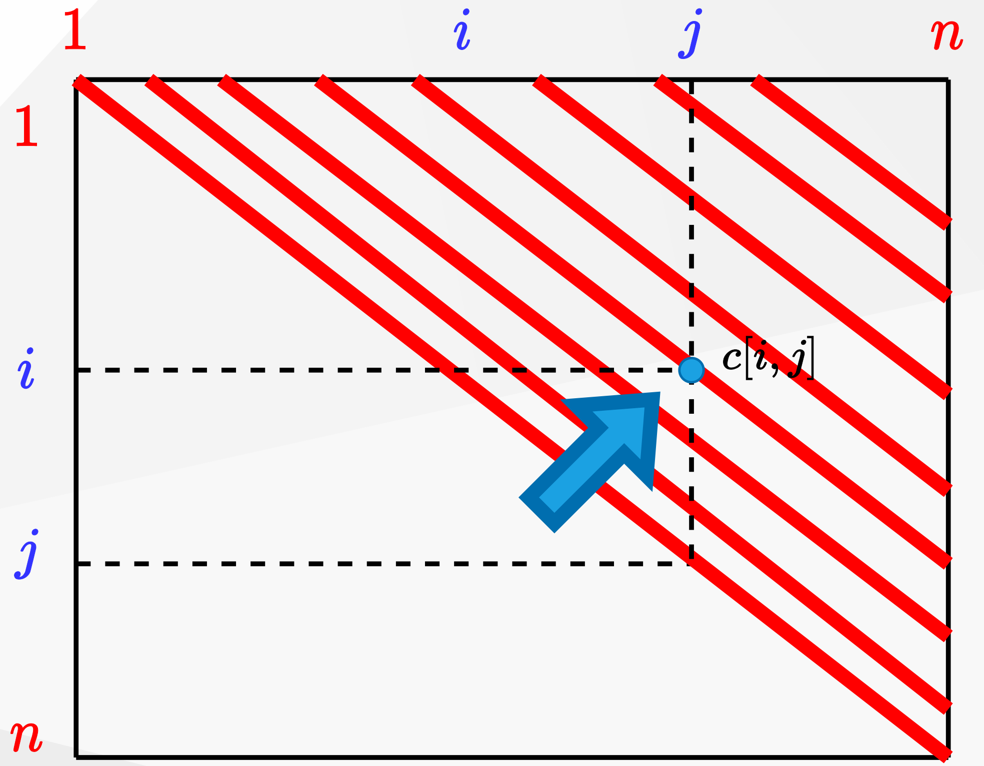
- $c[i, j]$ must be processed after $c[i, r-1]$ and $c[r+1, j]$



Bottom-up computation

$$c[i,j] = \begin{cases} 0 & \text{if } i > j \\ \min\limits_{i \le r \le j} \{c[i,r-1] + c[r+1,j] + P_{ij}\} & \text{otherwise} \end{cases}$$

- If the entries $c[i,j]$ are computed in the shown order, then $c[i,r-1]$ and $c[r+1,j]$ values are guaranteed to be computed before $c[i,j]$.



Computing the Optimal BST Cost

OPTIMAL-BST-COST(p, n)
  for $i \leftarrow 1$ to n do
    $c[i, i-1] \leftarrow 0$
    $c[i, i] \leftarrow p[i]$
    $R[i, j] \leftarrow i$
  $PS[1] \leftarrow p[1] \Longleftarrow PS[i] \rightarrow$ prefix-sum (i) : Sum of all $p[j]$ values for $j \leq i$
  for $i \leftarrow 2$ to n do
    $PS[i] \leftarrow p[i] + PS[i-1] \Longleftarrow$ compute the prefix sum
  for $d \leftarrow 1$ to $n-1$ do $\Longleftarrow$ BSTs with $d+1$ consecutive keys
    for $i \leftarrow 1$ to n–d do
      $j \leftarrow i + d$
      $c[i, j] \leftarrow \infty$
      for $r \leftarrow i$ to j do
        $q \leftarrow min\{c[i, r-1] + c[r+1, j]\} + PS[j]$–$PS[i-1]\}$
        if $q < c[i, j]$ then
          $c[i, j] \leftarrow q$
          $R[i, j] \leftarrow r$
  return $c[1, n], R$

Note on Prefix Sum

- We need P_{ij} values for each $i, j (1 \leq i \leq n \text{ and } 1 \leq j \leq n)$, where:

$$P_{ij} = \sum_{h=i}^{j} p_h$$

- If we compute the summation directly for every (i, j) pair, the runtime would be $\Theta(n^3)$.
- Instead, we spend $O(n)$ time in preprocessing to compute the prefix sum array **PS**. Then we can compute each P_{ij} in $O(1)$ time using **PS**.

Note on Prefix Sum

- In preprocessing, compute for each i:
  - $PS[i]$: the sum of $p[j]$ values for $1 \leq j \leq i$
- Then, we can compute P_{ij} in $O(1)$ time as follows:
  - $P_{ij} = PS[i]–PS[j-1]$
- **Example:**

$$
\begin{array}{r|cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
p: & 0.05 & 0.02 & 0.06 & 0.07 & 0.20 & 0.05 & 0.08 & 0.02
\end{array}
$$

$$
\begin{array}{r|cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
PS: & 0.05 & 0.07 & 0.13 & 0.20 & 0.40 & 0.45 & 0.53 & 0.55
\end{array}
$$

$$P_{27} = PS[7]–PS[1] = 0.53–0.05 = 0.48$$

$$P_{36} = PS[6]–PS[2] = 0.45–0.07 = 0.38$$

REVIEW

Overlapping Subproblems Property in Dynamic Programming

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.

Overlapping Subproblems Property in Dynamic Programming

Following are the two main properties of a problem that suggests that the given problem can be solved using Dynamic programming.

1. Overlapping Subproblems
2. Optimal Substructure

Overlapping Subproblems

- Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems.
- Dynamic Programming is mainly used when solutions of the same subproblems are needed again and again.
- In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed.
- So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again.

Overlapping Subproblems

- For example, Binary Search doesn’t have common subproblems.
- If we take an example of following recursive program for Fibonacci Numbers, there are many subproblems that are solved again and again.

Simple Recursion

- $f(n) = f(n-1) + f(n-2)$
- C sample code:

```c
#include <stdio.h>
// a simple recursive program to compute fibonacci numbers
int fib(int n)
{
    if (n <= 1)
        return n;
    else
        return fib(n-1) + fib(n-2);
}

int main()
{
    int n = 5;
    printf("Fibonacci number is %d ", fib(n));
    return 0;
}
```

Simple Recursion

- Output

```
Fibonacci number is 5
```

Simple Recursion

- $f(n) = f(n-1) + f(n-2)$

```java
/* a simple recursive program for Fibonacci numbers */
public class Fibonacci {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        System.out.println(fib(n));
    }

    public static int fib(int n) {
        if (n <= 1)
            return n;

        return fib(n - 1) + fib(n - 2);
    }
}
```

Simple Recursion

- $f(n) = f(n-1) + f(n-2)$

```
public class Fibonacci {
    public static void Main(string[] args) {
        int n = int.Parse(args[0]);
        Console.WriteLine(fib(n));
    }

    public static int fib(int n) {
        if (n <= 1)
            return n;

        return fib(n - 1) + fib(n - 2);
    }
}
```

Recursion tree for execution of fib(5)

```
                   fib(5)
                /            \
           fib(4)                fib(3)
         /      \                /      \
     fib(3)      fib(2)         fib(2)    fib(1)
    /     \       /    \       /    \
fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
 /    \
fib(1) fib(0)
```

- We can see that the function `fib(3)` is being called 2 times.
- If we would have stored the value of `fib(3)`, then instead of computing it again, we could have reused the old stored value.

Recursion tree for execution of fib(5)

There are following two different ways to store the values so that these values can be reused:

1. **Memoization (Top Down)**
2. **Tabulation (Bottom Up)**

Memoization (Top Down)

- The memoized program for a problem is similar to the recursive version with a small modification that looks into a lookup table before computing solutions.
- We initialize a lookup array with all initial values as `NIL`. Whenever we need the solution to a subproblem, we first look into the lookup table.
- If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

Memoization (Top Down)

- Following is the memoized version for the nth Fibonacci Number.
- C++ Version:

```cpp
/* C++ program for Memoized version
for nth Fibonacci number */
#include <bits/stdc++.h>
using namespace std;
#define NIL -1
#define MAX 100

int lookup[MAX];
```

Memoization (Top Down)

- C++ Version:

```
/* Function to initialize NIL
values in lookup table */
void _initialize()
{
    int i;
    for (i = 0; i < MAX; i++)
        lookup[i] = NIL;
}
```

Memoization (Top Down)

- C++ Version:

```
/* function for nth Fibonacci number */
int fib(int n)
{
    if (lookup[n] == NIL) {
        if (n <= 1)
            lookup[n] = n;
        else
            lookup[n] = fib(n - 1) + fib(n - 2);
    }

    return lookup[n];
}
```

Memoization (Top Down)

- C++ Version:

```
// Driver code
int main()
{
    int n = 40;
    _initialize();
    cout << "Fibonacci number is " << fib(n);
    return 0;
}
```

Memoization (Top Down)

- Java Version:

```java
/* Java program for Memoized version */
public class Fibonacci {
    final int MAX = 100;
    final int NIL = -1;

    int lookup[] = new int[MAX];

    /* Function to initialize NIL values in lookup table */
    void _initialize()
    {
        for (int i = 0; i < MAX; i++)
            lookup[i] = NIL;
    }
```

Memoization (Top Down)

- Java Version:

```
/* function for nth Fibonacci number */
int fib(int n)
{
    if (lookup[n] == NIL) {
        if (n <= 1)
            lookup[n] = n;
        else
            lookup[n] = fib(n - 1) + fib(n - 2);
    }
    return lookup[n];
}
```

Memoization (Top Down)

- Java Version:

```
    public static void main(String[] args)
    {
        Fibonacci f = new Fibonacci();
        int n = 40;
        f._initialize();
        System.out.println("Fibonacci number is"
                        + " " + f.fib(n));
    }
}
```

Memoization (Top Down)

- C# Version:

```
// C# program for Memoized versionof nth Fibonacci number
using System;

class FiboCalcMemoized {

    static int MAX = 100;
    static int NIL = -1;
    static int[] lookup = new int[MAX];

    /* Function to initialize NIL
    values in lookup table */
    static void initialize()
    {
        for (int i = 0; i < MAX; i++)
            lookup[i] = NIL;
    }
```

Memoization (Top Down)

- C# Version:

```csharp
/* function for nth Fibonacci number */
static int fib(int n)
{
    if (lookup[n] == NIL) {
        if (n <= 1)
            lookup[n] = n;
        else
            lookup[n] = fib(n - 1) + fib(n - 2);
    }
    return lookup[n];
}
```

Memoization (Top Down)

- C# Version:

```
    // Driver code
    public static void Main()
    {

        int n = 40;
        initialize();
        Console.Write("Fibonacci number is"
                      + " " + fib(n));
    }
}
```

Tabulation (Bottom Up)

- The tabulated program for a given problem builds a table in bottom-up fashion and returns the last entry from the table.
- For example, for the same Fibonacci number,
  - we first calculate `fib(0)` then `fib(1)` then `fib(2)` then `fib(3)`, and so on. So literally, we are building the solutions of subproblems bottom-up.

Tabulation (Bottom Up)

- C++ Version:

```
/* C program for Tabulated version */
#include <stdio.h>
int fib(int n)
{
    int f[n + 1];
    int i;
    f[0] = 0;
    f[1] = 1;
    for (i = 2; i <= n; i++)
        f[i] = f[i - 1] + f[i - 2];

    return f[n];
}
```

Tabulation (Bottom Up)

- C++ Version:

```
...
int main()
{
    int n = 9;
    printf("Fibonacci number is %d ", fib(n));
    return 0;
}
```

Output:

```
Fibonacci number is 34
```

Tabulation (Bottom Up)

- Java Version:

```java
/* Java program for Tabulated version */
public class Fibonacci {
  public static void main(String[] args)
  {
    int n = 9;
    System.out.println("Fibonacci number is " + fib(n));
  }
```

Tabulation (Bottom Up)

- Java Version:

```java
  /* Function to calculate nth Fibonacci number */
  static int fib(int n)
  {
    int f[] = new int[n + 1];
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++)
      f[i] = f[i - 1] + f[i - 2];

    return f[n];
  }
}
```

Tabulation (Bottom Up)

- C# Version:

```csharp
// C# program for Tabulated version
using System;

class Fibonacci {
    static int fib(int n)
    {
        int[] f = new int[n + 1];
        f[0] = 0;
        f[1] = 1;
        for (int i = 2; i <= n; i++)
            f[i] = f[i - 1] + f[i - 2];
        return f[n];
    }

    public static void Main()
    {
        int n = 9;
        Console.Write("Fibonacci number is"
                      + " " + fib(n));
    }
}
```

- Both Tabulated and Memoized store the solutions of subproblems.
- In Memoized version, the table is filled on demand while in the Tabulated version, starting from the first entry, all entries are filled one by one.
- Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version.

- To see the optimization achieved by Memoized and Tabulated solutions over the basic Recursive solution, see the time taken by following runs for calculating the 40th Fibonacci number:
- Recursive Solution:
  - https://ide.geeksforgeeks.org/vHt6ly
- Memoized Solution:
  - https://ide.geeksforgeeks.org/Z94jYR
- Tabulated Solution:
  - https://ide.geeksforgeeks.org/12C5bP

Optimal Substructure Property in Dynamic Programming

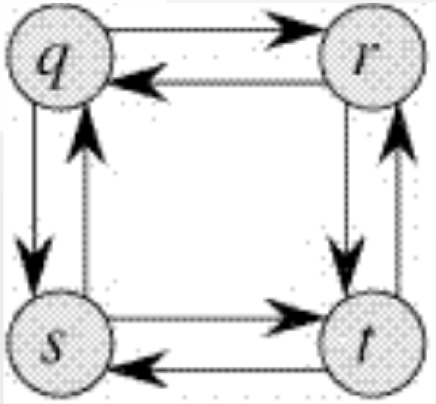
- A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.
- For example, the Shortest Path problem has following optimal substructure property:
  - If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. The standard All Pair Shortest Path algorithm like Floyd–Warshall and Single Source Shortest path algorithm for negative weight edges like Bellman–Ford are typical examples of Dynamic Programming.

Optimal Substructure Property in Dynamic Programming

- On the other hand, the Longest Path problem doesn’t have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes

Optimal Substructure Property in Dynamic Programming

- There are two longest paths from q to t: q→r→t and q→s→t. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path q→r→t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q→s→t→r and the longest path from r to t is r→q→s→t.

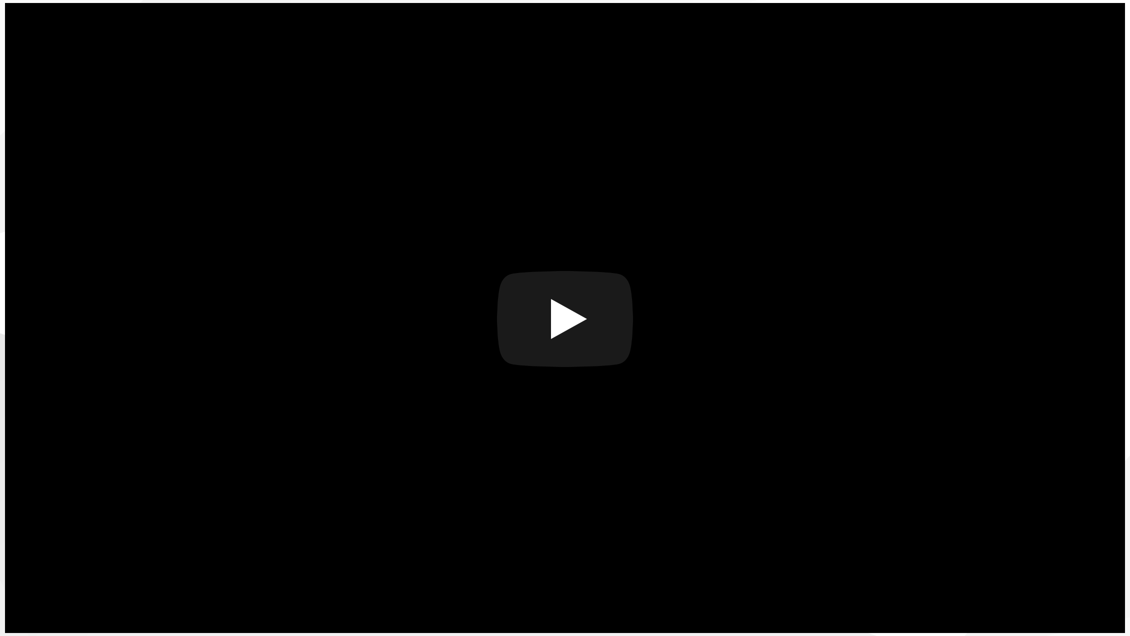


Most Common Dynamic Programming Interview Questions

Problem-1: Longest Increasing Subsequence

- Problem-1: Longest Increasing Subsequence

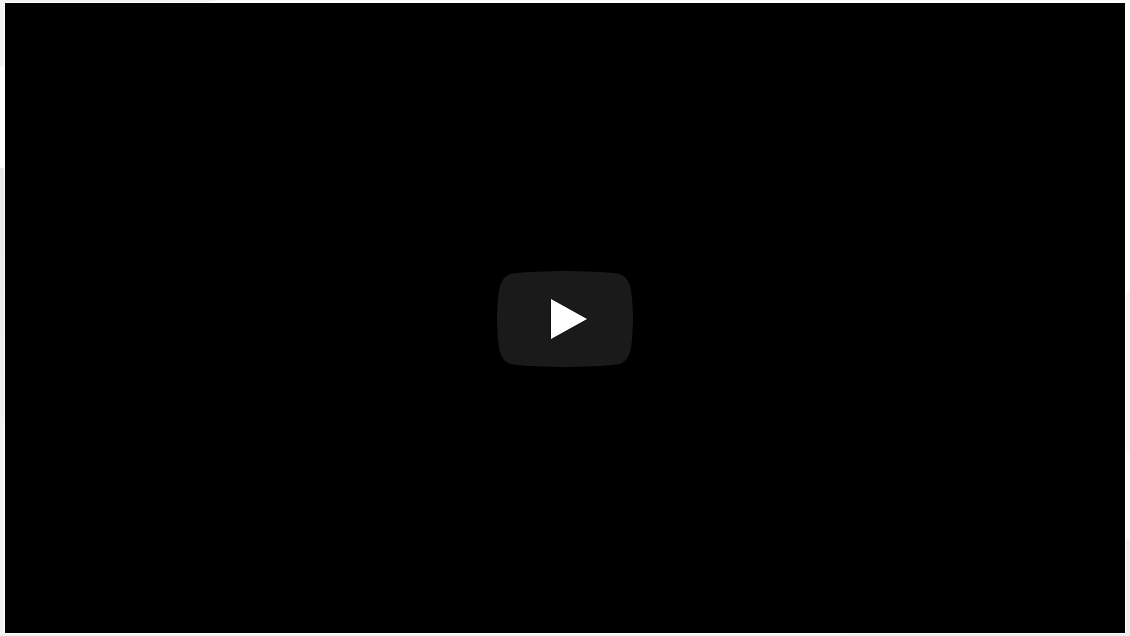
Problem-1: Longest Increasing Subsequence



Problem-2: Edit Distance

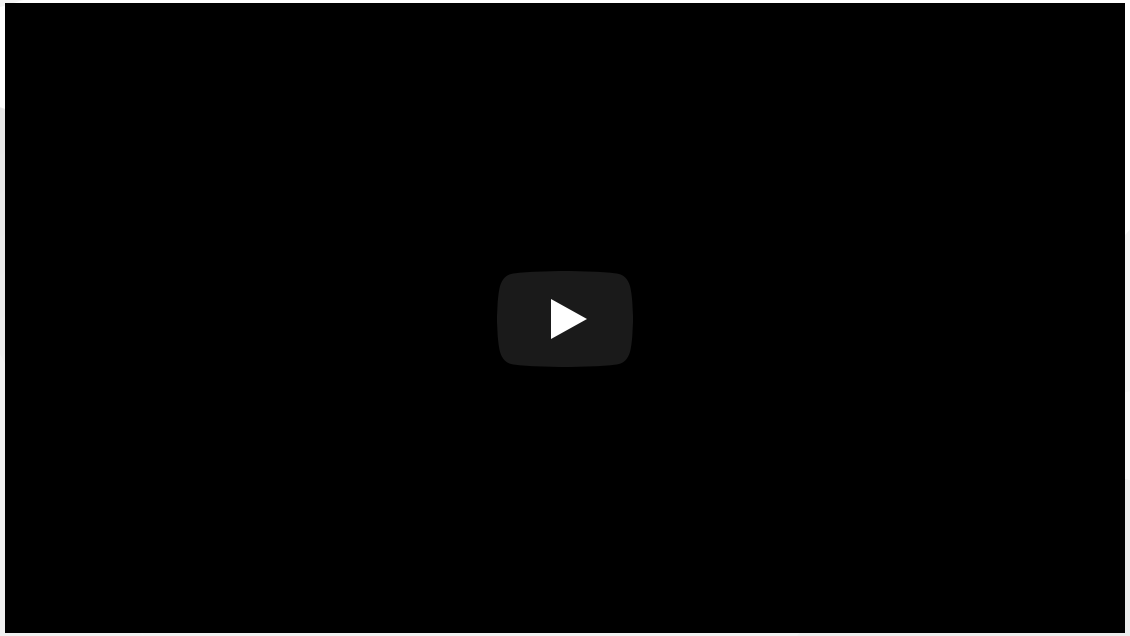
- Problem-2: Edit Distance

Problem-2: Edit Distance (Recursive)

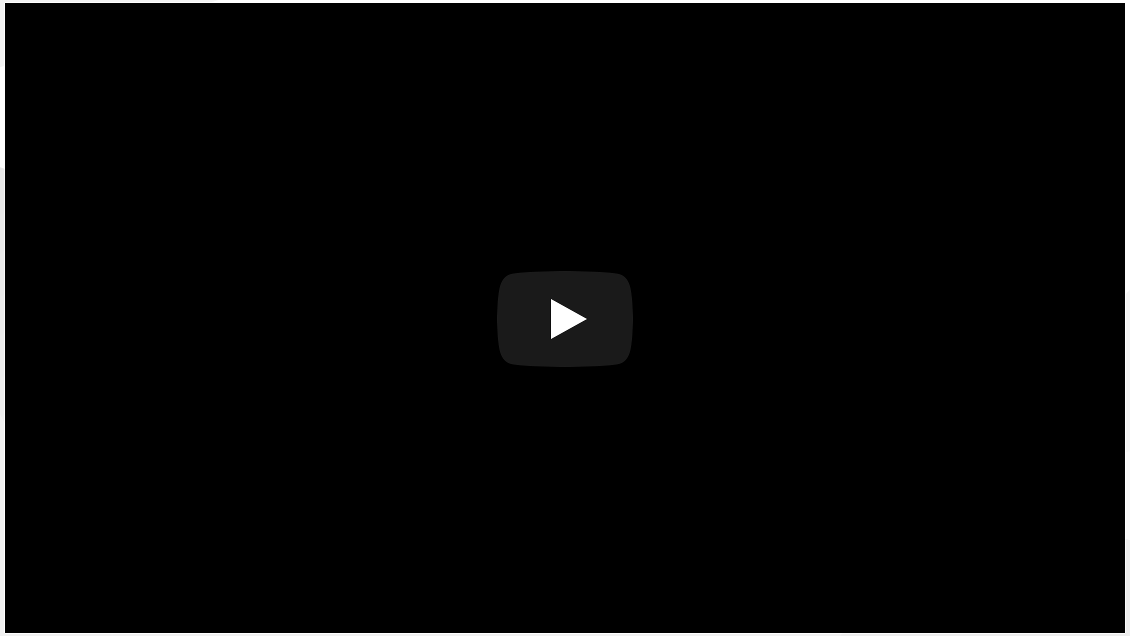


Problem-2: Edit Distance (DP)

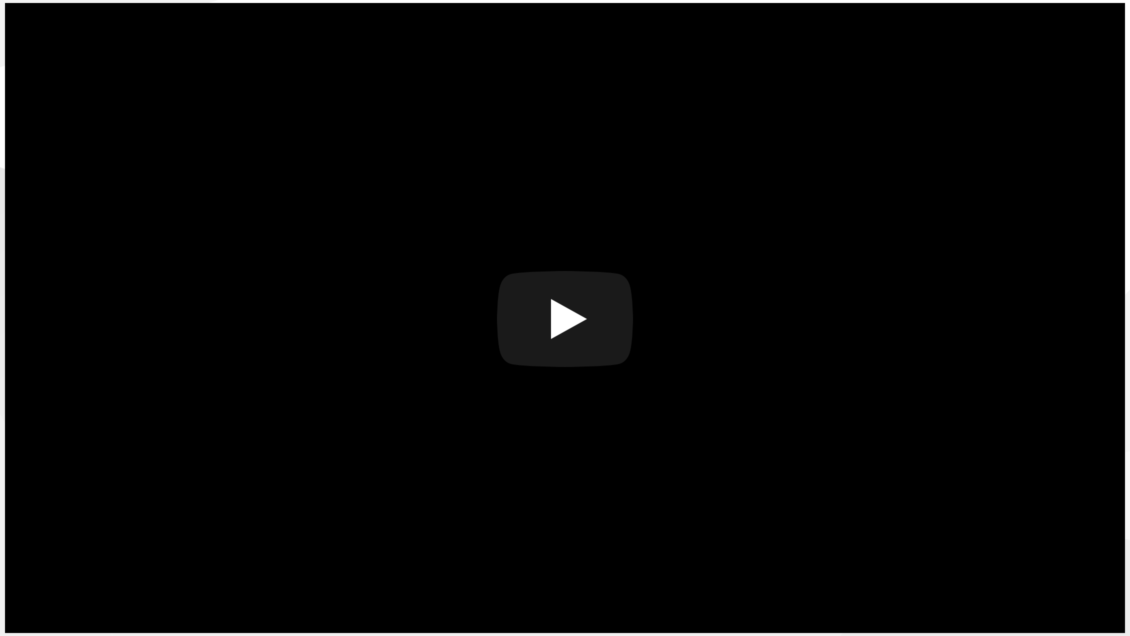
https://www.coursera.org/learn/dna-sequencing



Problem-2: Edit Distance (DP)



Problem-2: Edit Distance (Other)



Problem-3: Partition a set into two subsets such that the difference of subset sums is minimum

- Problem-3: Partition a set into two subsets such that the difference of subset sums is minimum

Problem-4: Count number of ways to cover a distance

- Problem-4: Count number of ways to cover a distance

Problem-5: Find the longest path in a matrix with given constraints

- Problem-5: Find the longest path in a matrix with given constraints

Problem-6: Subset Sum Problem

- Problem-6: Subset Sum Problem

Problem-7: Optimal Strategy for a Game

- Problem-7: Optimal Strategy for a Game

Problem-8: 0-1 Knapsack Problem

- Problem-8: 0-1 Knapsack Problem

Problem-9: Boolean Parenthesization Problem

- Problem-9: Boolean Parenthesization Problem

Problem-10: Shortest Common Supersequence

- Problem-10: Shortest Common Supersequence

Problem-11: Partition Problem

- Problem-11: Partition Problem

Problem-12: Cutting a Rod

- Problem-12: Cutting a Rod

Problem-13: Coin Change

- Problem-13: Coin Change

Problem-14: Word Break Problem

- Problem-14: Word Break Problem

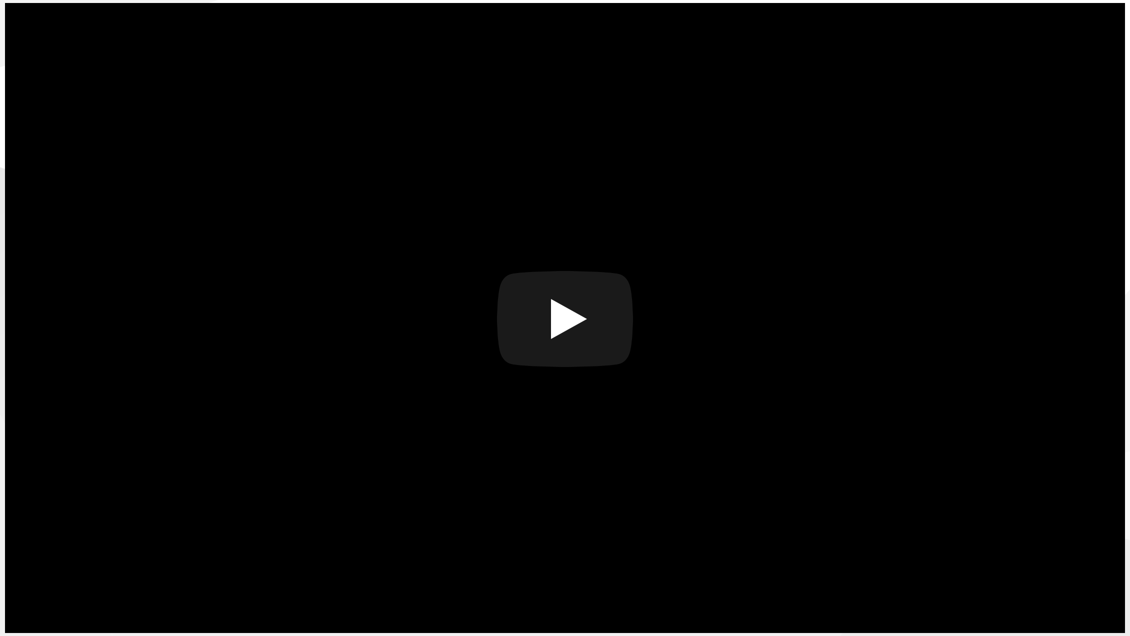
Problem-15: Maximum Product Cutting

- Problem-15: Maximum Product Cutting

Problem-16: Dice Throw

- Problem-16: Dice Throw

Problem-16: Dice Throw



Problem-17: Box Stacking Problem

- Problem-17: Box Stacking Problem

Problem-18: Egg Dropping Puzzle

- Problem-18: Egg Dropping Puzzle

References

- Introduction to Algorithms, Third Edition | The MIT Press
  - CLRS
- Bilkent CS473 Course Notes (new)
- Bilkent CS473 Course Notes (old)

$$-End - Of - Week - 6 - Course - Module-$$