CS211FZ: Data Structures and Algorithms II

11- Dynamic Programming

Longest Common Subsequence

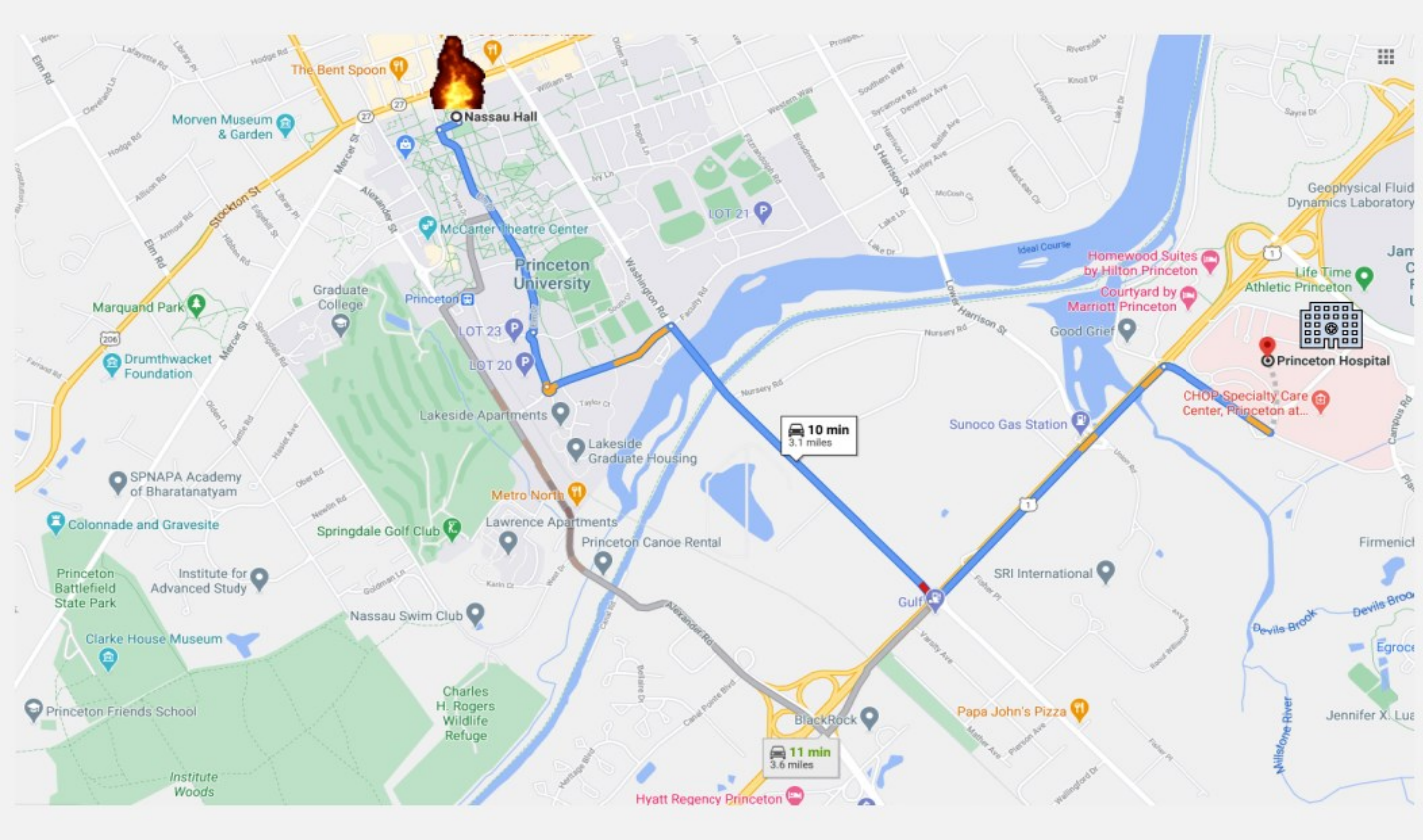
LECTURER: MAHWISH KUNDI & IRFAN AHMAD

MAHWISH.KUNDI@MU.IE , IRFAN.AHMAD@MU.IE

Introduction

Algorithms for Optimization Problems

- Designed to find the best solution among a set of possible solutions.
- For example:
  - Finding the shortest route or maximizing profits.



Algorithms for Optimization Problems

- Commonly used algorithms for solving optimization problems include:
  - Dynamic Programing
  - Greedy Algorithms

Dynamic Programming Vs Greedy Algorithm

- Both are employed to find best solution among various possibilities.

Dynamic programing

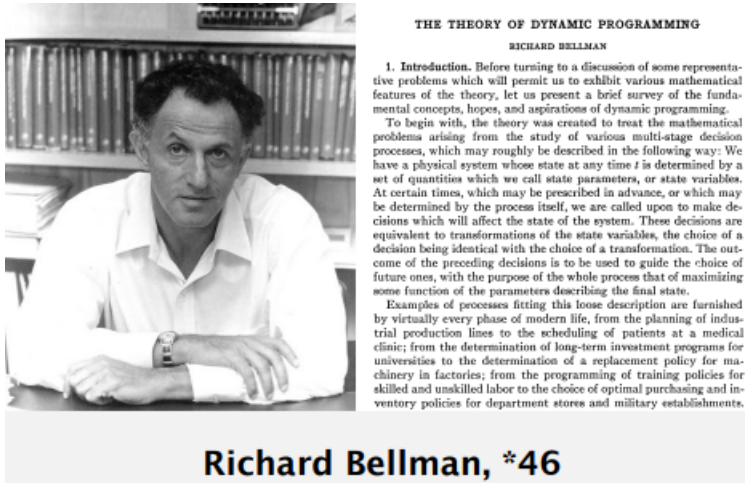
- Divides the complex problem into small subproblems.
- Make a choice at each step.
- Choice depends on knowing optimal solutions to subproblems. Solve subproblems first.
- Solve bottom-up. More optimal but slower.

Greedy algorithms

- Divides the complex problem into small subproblems.
- Make a choice at each step.
- Make the choice before solving the subproblems.
- Solve top-down. Less optimal but quicker.

Dynamic Programming - Definition

- Richard Bellman introduced the idea of dynamic programming in the 1950s.
- Algorithm design paradigm.
  - Divide a complex problem into a number of simpler overlapping subproblems.
  - Solve each subproblem only once.
  - Store the results of each solved sub-problem for later reuse.
  - Use the stored answers to find the overall solution.



THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time t is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

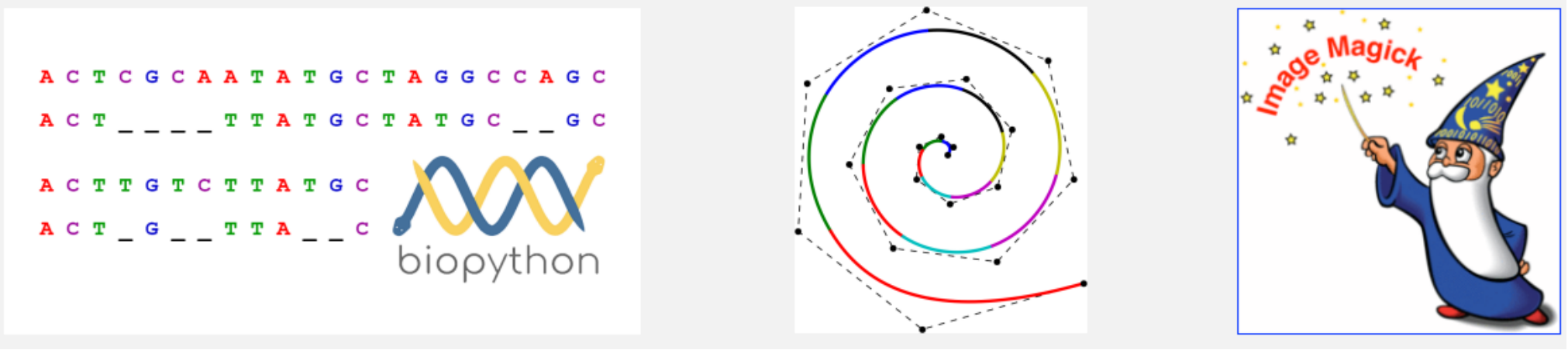
Richard Bellman, *46

Dynamic Programming - Application Areas

- Operations research: multistage decision processes, control theory, optimization, ...
- Computer science: AI, compilers, systems, graphics, databases, robotics, theory, ….
- Economics.
- Bioinformatics.
- Information theory.
- Tech job interviews.

Dynamic Programming - Algorithms

- Some famous algorithms:
  - System R algorithm for optimal join order in relational databases.
  - Needleman–Wunsch/Smith–Waterman for sequence alignment.
  - Bellman–Ford–Moore for shortest path.
  - De Boor for evaluating spline curves.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.

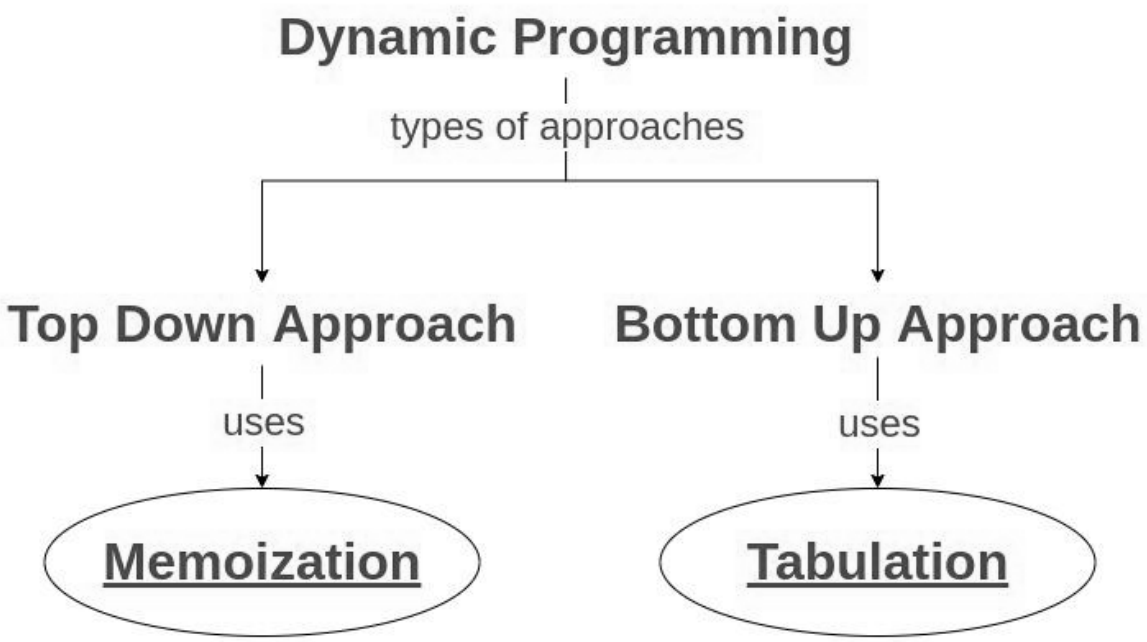


Dynamic Programming - Some Books

pp. 284–289

Dynamic Programming - Techniques to Solve Problems

- Dynamic programming is divided into two main approaches:



- Both help in solving complex problems more efficiently by storing and reusing solutions of overlapping subproblems, but they differ in the way they work.

Example - Fibonacci numbers

- Fibonacci numbers. 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

$$F_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ F_{i-1} + F_{i-2} & \text{if } i > 1 \end{cases}$$


Leonardo Fibonacci


3


5


8


13


21


34


55


89

Flower Patels follow Fibonacci numbers.

Fibonacci numbers: naïve recursive approach

- Goal: Given n, compute the nth Fibonacci number F_n.

- Naïve recursive approach:

```
public static long fib(int i)
{
   if (i == 0) return 0;
   if (i == 1) return 1;
   return fib(i-1) + fib(i-2);
}
```

Fibonacci numbers: naïve recursive approach

- How long to compute fib(80) using the naïve recursive algorithm?

A. Less than 1 second.

B. About 1 minute.

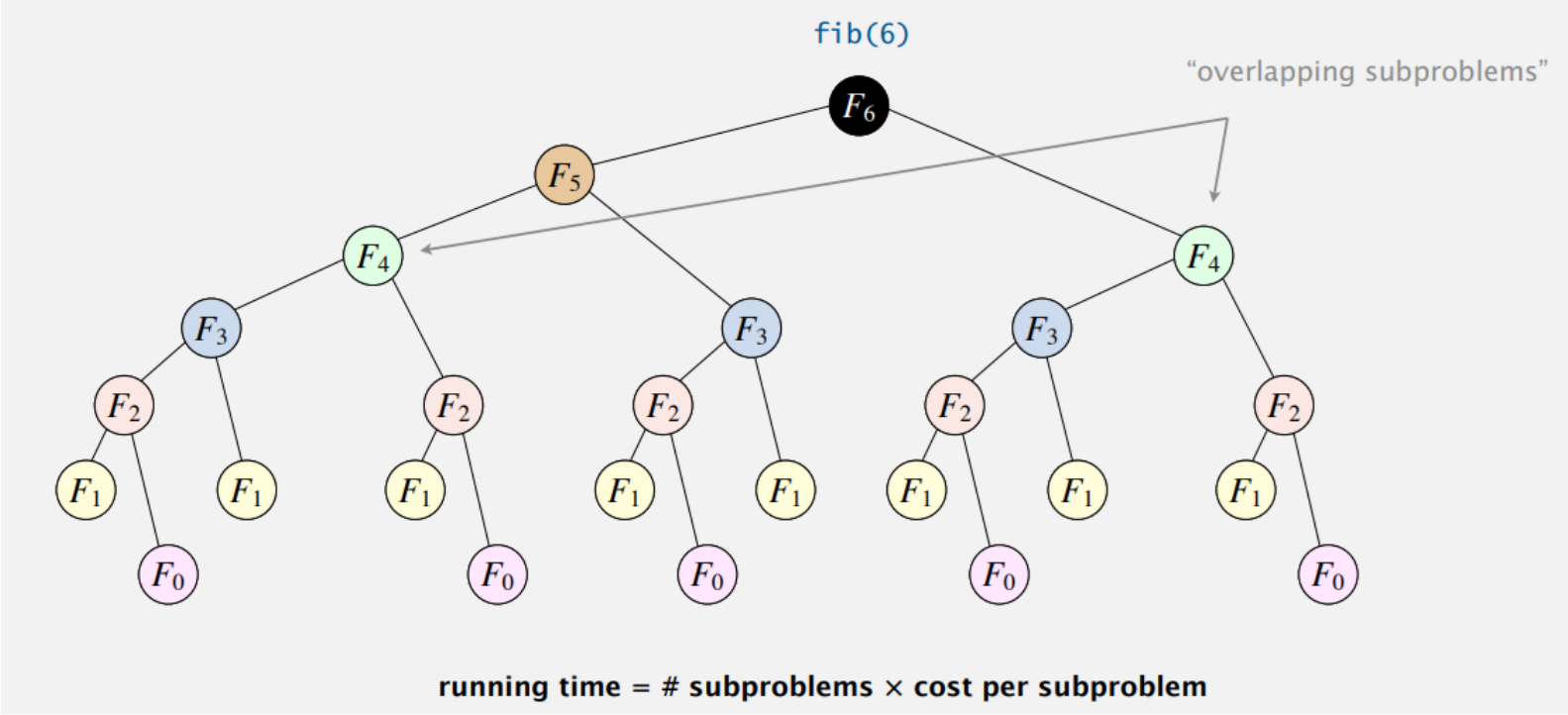
C. More than 1 hour.

D. Overflows a 64-bit long integer.

C. More than 1 hour. Computing **fib(80)** using the naive recursive algorithm would take an extremely long time, typically much more than one hour, due to the exponential increase in the number of recursive calls.

Fibonacci numbers: naïve recursive approach

- Exponential waste. Same overlapping subproblems are solved repeatedly.
- Ex. To compute fib(6):
  - fib(5) is called 1 time.
  - fib(4) is called 2 times.
  - fib(3) is called 3 times.
  - fib(2) is called 5 times.
  - fib(1) is called = 8 times.



Fibonacci numbers: top-down dynamic programming

- Memoization.
  - Top-down dynamic programming is also known as **memoization** because it avoids duplicating work by remembering the results of function calls.
  - Maintain an array (or symbol table) to remember all computed values.
  - If value to compute is known, just return it; otherwise, compute it; remember it; and return it.

```
public static long fib(int i)
{
   if (i == 0) return 0;
   if (i == 1) return 1;
   if (f[i] == 0) f[i] = fib(i-1) + fib(i-2);
   return f[i];
}
```

assume global long array f[], initialized to 0 (unknown)

Fibonacci numbers: top-down dynamic programming

- Explanation of the code.
  - Declares static array f for caching. Assume global long array f[], initialized to 0.
  - Handles base cases for n = 0 and n = 1.
  - Computes and caches Fibonacci values recursively.
  - Returns cached value if already computed.

- Impact:
  - Solves each subproblem F_i only once; $\Theta(n)$ time to compute F_n.

Fibonacci numbers: bottom-up dynamic programming

- Tabulation.
  - Build computation from the “bottom up.”
  - Solve small subproblems and save solutions.
  - Use those solutions to solve larger subproblems.

```
public static long fib(int n)
{
    long[] f = new long[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];
}
```

smaller subproblems

Fibonacci numbers: bottom-up dynamic programming

- Explanation of the code.
  - Creates array f of size n+1.
  - Sets f[0] = 0 and f[1] = 1.
  - Loops from 2 to n, calculating f[i].
  - Returns the value f[n].
- For example, to solve f[2] = f[1]+f[0], we already have solution of f[1] and f[0].

- Impact:
  - Solves each subproblem F_i only once; $\Theta(n)$ time to compute F_n. No recursion.

Fibonacci numbers: bottom-up dynamic programming

When the ordering of the subproblems is clear, and space is available to store all the solutions, bottom-up dynamic programming is a very effective approach.

Dynamic programming - summary

- Divide a complex problem into a number of simpler overlapping subproblems.
  - [define n + 1 subproblems, where subproblem i is computing the ith Fibonacci number]
- Define a recurrence relation to solve larger subproblems from smaller subproblems.
  - [easy to solve subproblem i if we know solutions to subproblems i − 1 and i − 2]
- Store solutions to each of these subproblems, solving each subproblem only once.
  - [use an array, storing subproblem i in f[i]]
- Use stored solutions to solve the original problem.
  - [subproblem n is original problem]

Longest Common Subsequence Problem

Longest Common Subsequence - Problem

- Let's consider an application of dynamic programming, where the order of solving the subproblems is not so clear, unlike Fibonacci numbers.

- Solving these problems will be difficult without recursive thinking and a bottom-up dynamic programming approach.

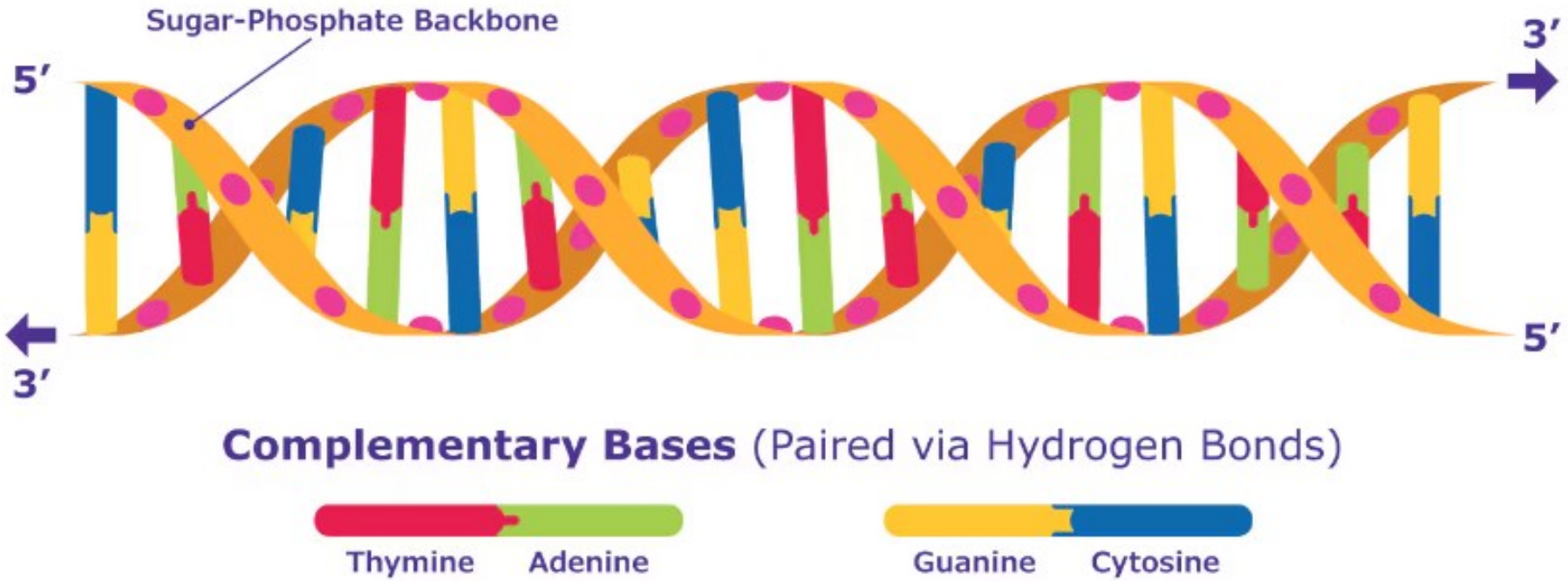
- Break down the problem into smaller subproblems, solve them in a specific order.

Longest Common Subsequence - Problem

- A fundamental string-processing problem that arises in computational biology and other domains.
- Given two strings x and y, we wish to determine how similar they are.
- Some example:
  - Comparing two DNA sequences for homology
  - Two English words for spelling
  - Two Java files for repeated code

Longest Common Subsequence - Problem

- Biological applications often need to compare the DNA of two (or more) different organisms.
- A strand of DNA consists of a string of molecules called bases.
- Possible bases are adenine{A}, cytosine{C}, guanine{G}, and thymine{T}.



Longest Common Subsequence - Problem

- We can express a strand of DNA as a string over the 4-element set $\{A, C, G, T\}$.

- For example, the DNA of one organism may be

$$S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA$$

- and the DNA of another organism may be

$$S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA$$

- Compare two strands of DNA to determine how "similar" or how "closely" related the two organisms are. How?

Longest Common Subsequence - Introduction

- A subsequence of a string S, is a set of characters that appear in left to-right order, but not necessarily consecutively.

- Example

$$ACTTGCG$$

- ACT, ATTC, T, ACTTGC are all subsequences.
- TTA is not a subsequence.

Longest Common Subsequence - Introduction

- A common subequence of two strings is a subsequence that appears in both strings.
- A longest common subequence (LCS) is a common subsequence of maximal length.
- Example 1:

$$S_1 = AAACCGTGAGTTATTCGTTCTAGAA$$
$$S_2 = CACCCCTAAGGTACCTTTGGTTC$$

- LCS is

$$S_1 = AAACCGTGAGTTATTCGTTCTAGAA$$
$$S_2 = CACCCCTAAGGTACCTTTGGTTC$$

$$ACCTAGTACTTTG$$

Longest Common Subsequence - Introduction

- Example 2: Two given sequences are

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

- Common subsequences of X and Y are

  $\langle B, C, A \rangle$ of length 3

  $\langle B, D, A, B \rangle$ of length 4

  $\langle B, C, B, A \rangle$ of length 4

- The sequence $\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ of length are LCS of X and Y, since X and Y have no common subsequence of length 5 or greater.

Longest Common Subsequence - Introduction

- In the longest-common-subsequence problem, the input is two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ and the goal is to find a maximum-length common subsequence of X and Y.

- This section shows how to efficiently solve the LCS problem using dynamic programming.

Longest Common Subsequence - Dynamic Programming

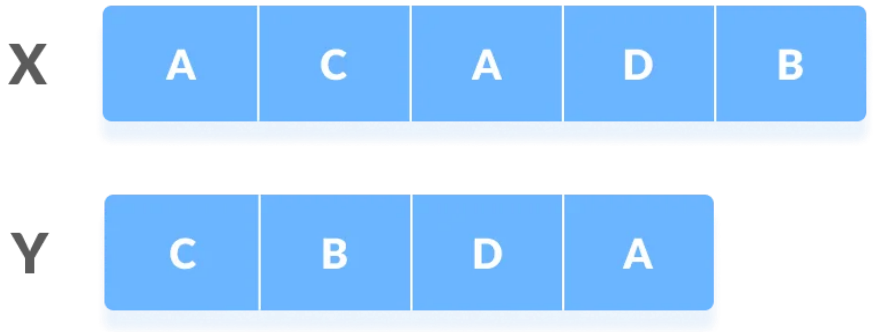
- In the bottom-up dynamic approach using the following steps and a table to track, we can find the longest common subsequence between two strings.
  - Step 1: Create a matrix of the size of the both strings $(n + 1) \times (m + 1)$ and fill the first column and row with 0.
  - Step 2: Fill each cell of the table using the following logic:
    - If the character corresponding to the current row and current column are matching, then fill the current cell by adding 1 to the value of its left top diagonal cell. Point an arrow to the diagonal cell.
    - If no match found, fill the current cell by taking max of previous row (top cell) and previous column (left cell), and Point an arrow to the cell with maximum value. If they are equal, then point to any of them (we shall point to previous row (top cell).

Longest Common Subsequence - Dynamic Programming

- Repeat Step 2 for all the rows and columns of the matrix.
- Step 3: The last cell in the matrix (last row and last column) will have the length of the longest common subsequence.
- Step 4: To find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to ("↖") symbol form the longest common subsequence.

Longest Common Subsequence - Dynamic Programming

- Let us take two sequences:



- Our goal is to find the LCS using bottom-up dynamic programming.

Longest Common Subsequence - Dynamic Programming

- **Step 1:** Create a matrix of the size of the both strings $(n + 1) \times (m + 1)$ and fill the first column and row with 0.

Initialize a table

		0 y_j	1 C	2 B	3 D	4 A
0	x_i	0	0	0	0	0
1	A	0				
2	C	0				
3	A	0				
4	D	0				
5	B	0				

Longest Common Subsequence - Dynamic Programming

- Step 2: Compare $x_i = 1, 2, 3, 4, 5$ with each column y_1, y_2, y_3, and y_4.
  - If a match found, fill the current cell by adding 1 to the value of its left top diagonal cell. Arrow points to diagonal cell.
  - If no match found, fill the current cell by taking max of previous row (top cell) and previous column (left cell), and point an arrow to the cell with maximum value. If they are equal, then point to any of them (we shall point to previous row (top cell).
  - Note, we need to see the top and left cell of the current cell.

		0 y_j	1 C	2 B	3 D	4 A
0	x_i	0	0	0	0	0
1	A	0				
2	C	0				
3	A	0				
4	D	0				
5	B	0				

Fill the values

Longest Common Subsequence - Dynamic Programming

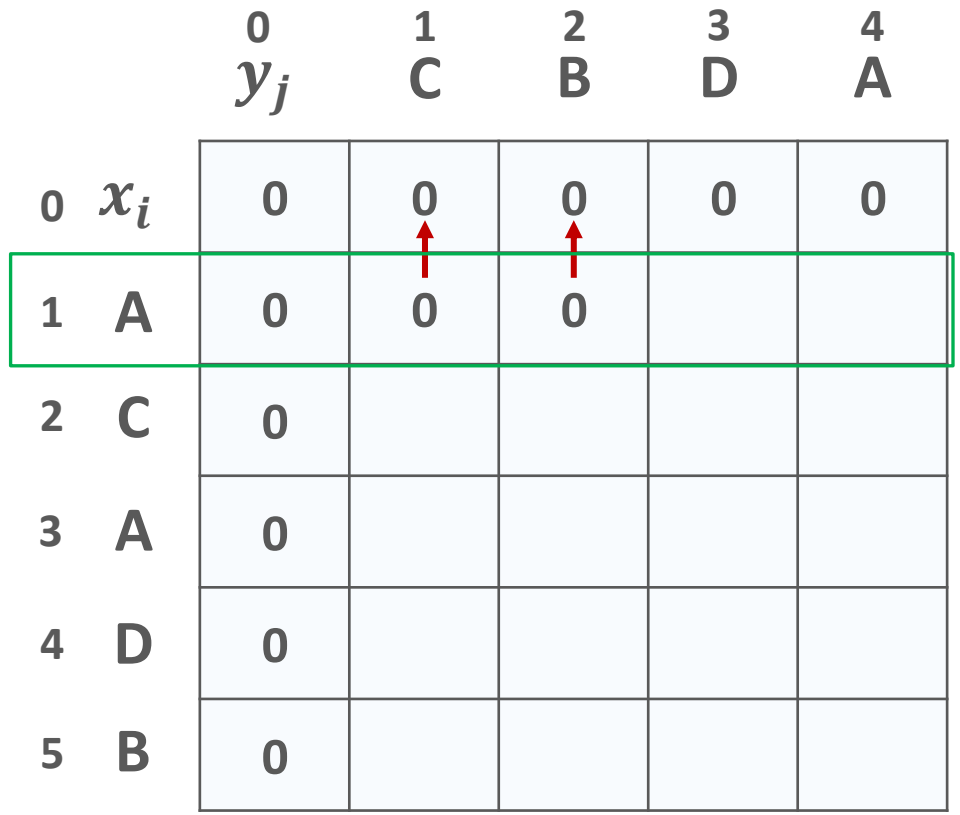
- Compare x_1 = A in row 1 with each column y_1, y_2, y_3, and y_4.
  - For example, x_1 = A and y_1= C do not match, so we take max value of top and left cell which is 0. The values are equal, so the arrow points to previous row (top cell).
  - A pointer is used for pointing to the cells.

		0 y_j	1 C	2 B	3 D	4 A
0	x_i	0	0	0	0	0
1	A	0	0			
2	C	0				
3	A	0				
4	D	0				
5	B	0				

Fill the values

Longest Common Subsequence - Dynamic Programming

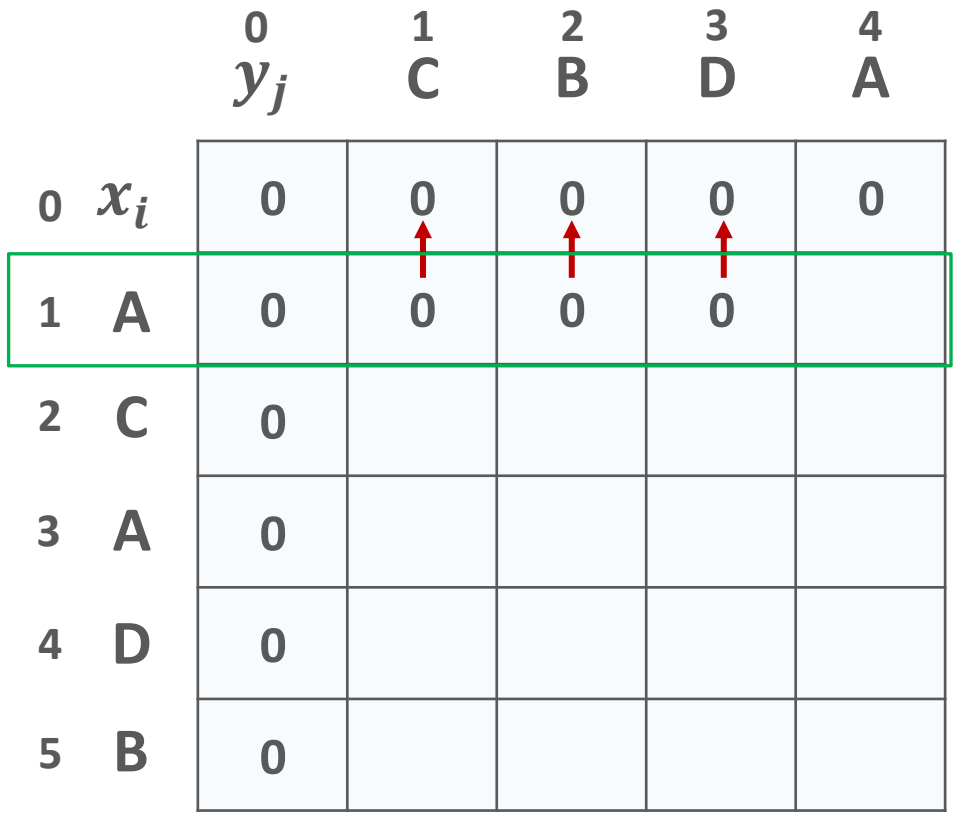
- Compare x_1 = A in row 1 with each column y_1, y_2, y_3, and y_4.
  - x_1 = A and y_2= B do not match, so we take max value of top and left cell which is 0. The values are equal, so the arrow points to previous row (top cell).



Fill the values

Longest Common Subsequence - Dynamic Programming

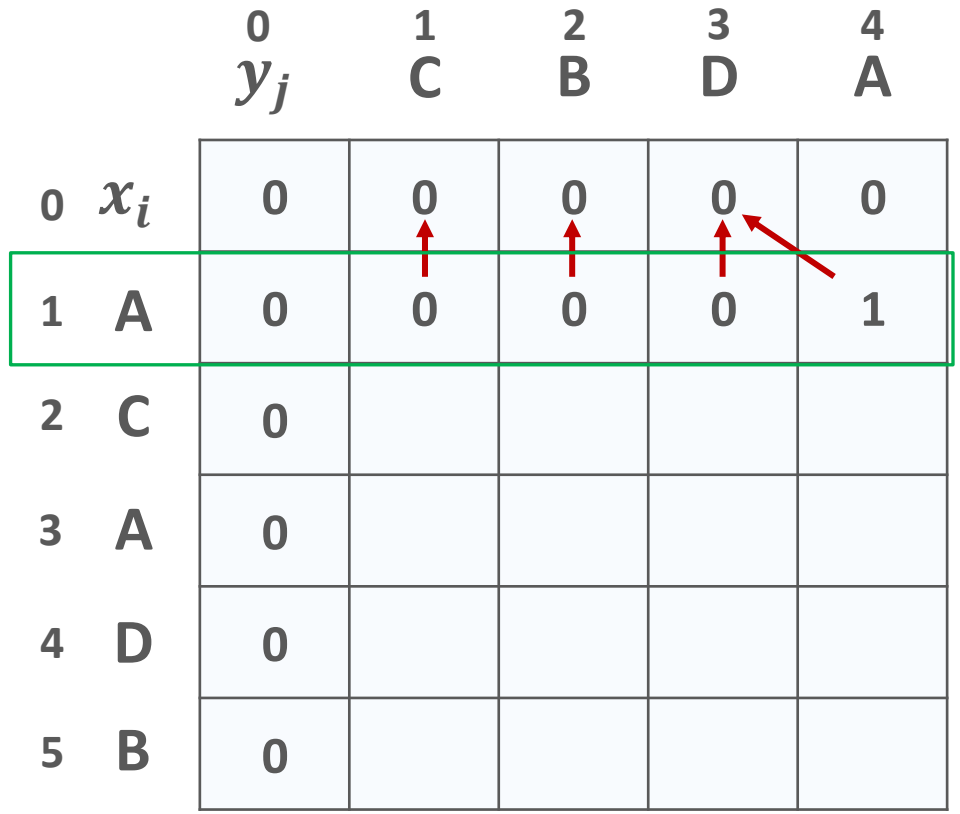
- Compare x_1 = A in row 1 with each column y_1, y_2, y_3, and y_4.
  - x_1 = A and y_3= D do not match, so we take max value of top and left cell which is 0. The values are equal, so the arrow points to previous row (top cell).



Fill the values

Longest Common Subsequence - Dynamic Programming

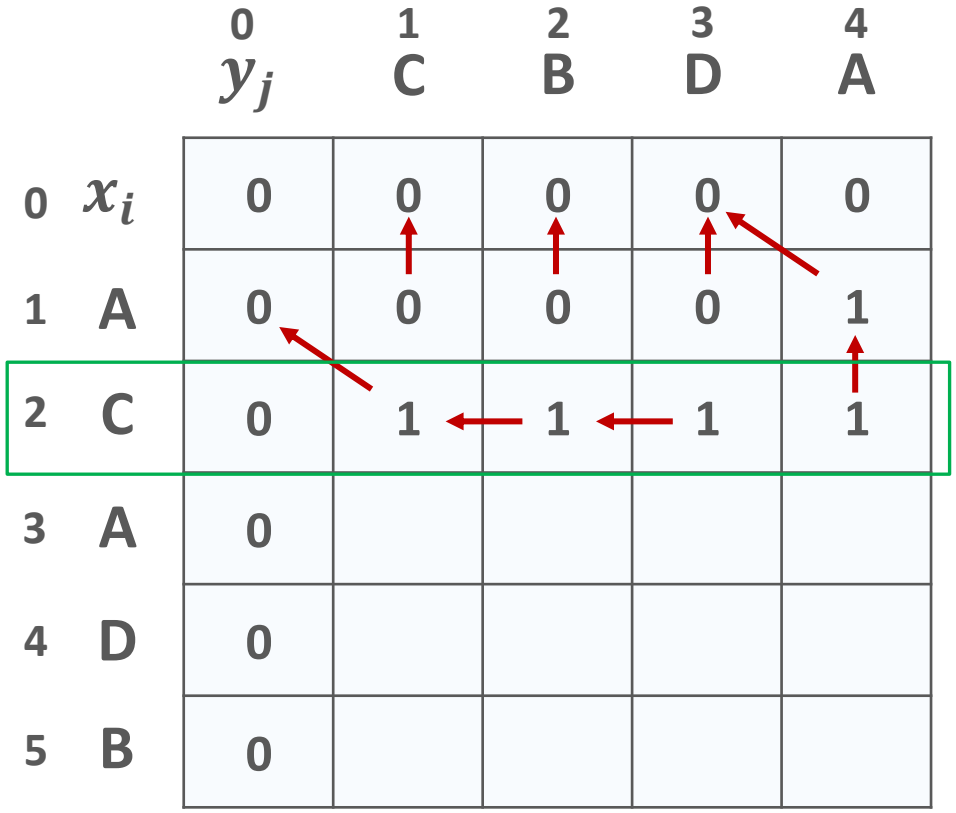
- Compare x_1 = A in row 1 with each column y_1, y_2, y_3, and y_4.
  - x_1 = A and y_4= A matches, so we add 1 to the left top diagonal cell (0+1=1) and fill it in the current cell. The arrow points to the diagonal cell.



Fill the values

Longest Common Subsequence - Dynamic Programming

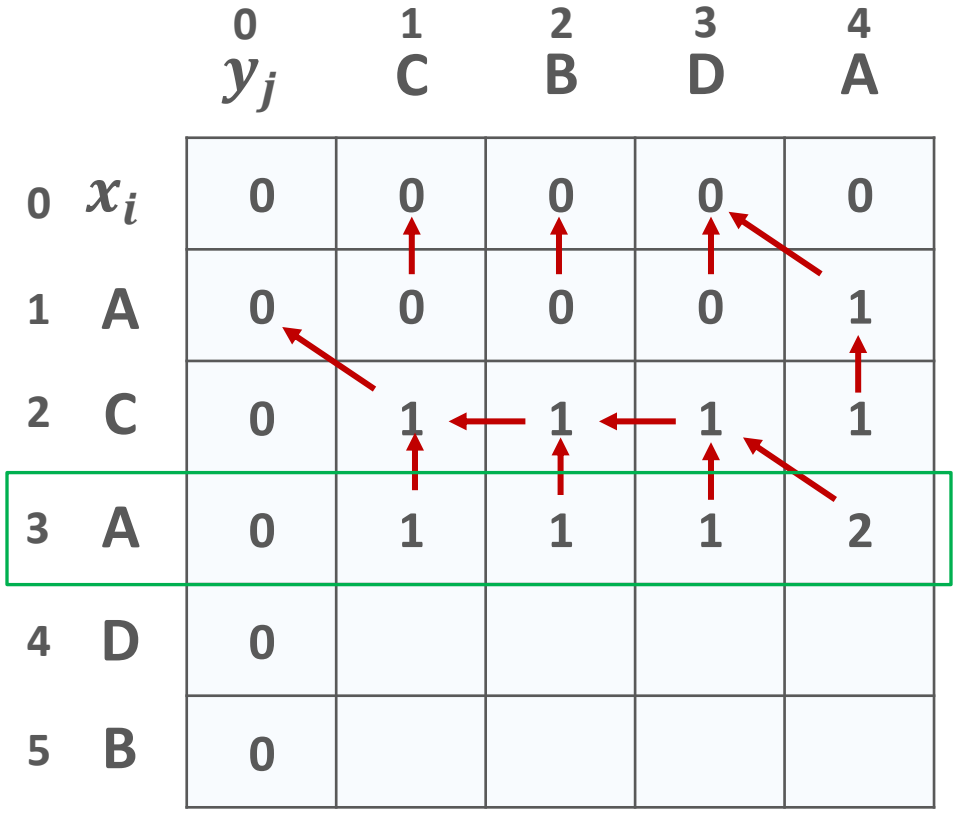
- Compare x_2 = C in row 2 with each column y_1, y_2, y_3, and y_4.
  - x_2 = C and y_1= C matches, so we add 1 to the left top diagonal cell (0+1=1) and fill it in the current cell. The arrow points to the diagonal cell.
  - x_2 = C and y_2= B do not match, so we take max value of top and left cell which is 1. The arrow points to the cell with maximum value.
  - x_2 = C and y_3= D do not match, so we take max value of top and left cell which is 1. The arrow points to the cell with maximum value.
  - x_2 = C and y_4= A do not match, so we take max value of top and left cell which is 1. The values are equal, so the arrow points to previous row (top cell).



Fill the values

Longest Common Subsequence - Dynamic Programming

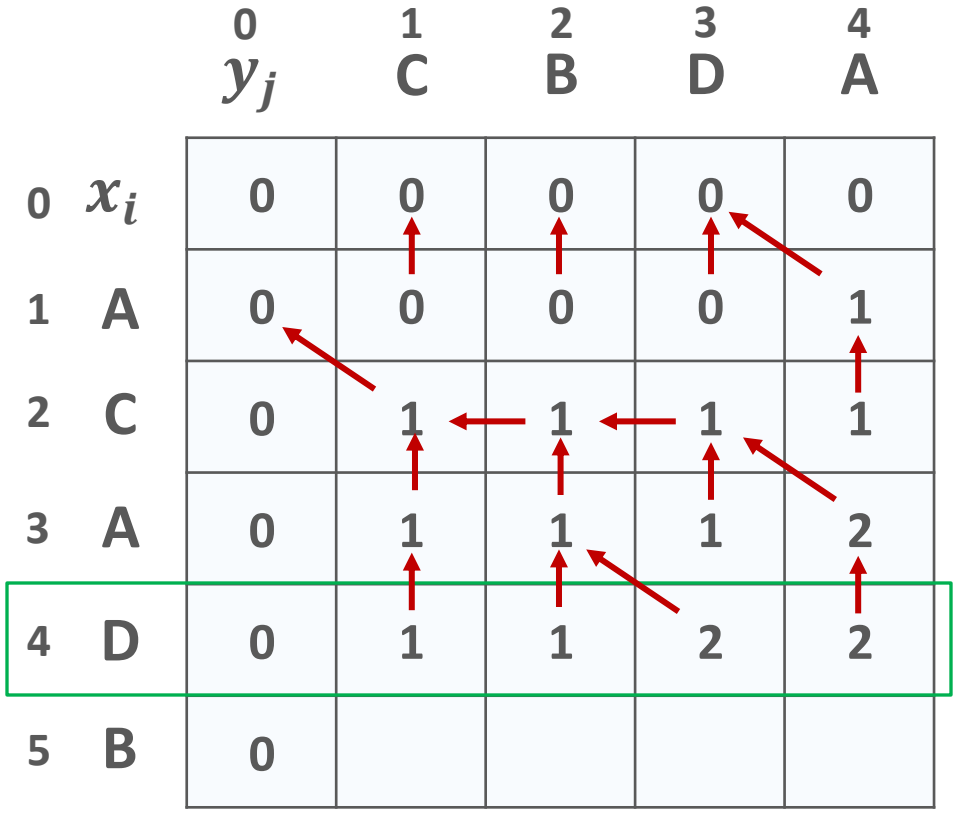
- Compare x_3 = A in row 3 with each column y_1, y_2, y_3, and y_4.
  - x_3 = A and y_1= C do not match, so we take max value of top and left cell which is 1. The arrow points to the cell with maximum value.
  - x_3 = A and y_2= B do not match, so we take max value of top and left cell which is 1. The values are equal, so the arrow points to previous row (top cell).
  - x_3 = A and y_3= D do not match, so we take max value of top and left cell which is 1. The values are equal, so the arrow points to previous row (top cell).
  - x_3 = A and y_4= A matches, so we add 1 to the left top diagonal cell (1+1=2) and fill it in the current cell. The arrow points to the diagonal cell.



Fill the values

Longest Common Subsequence - Dynamic Programming

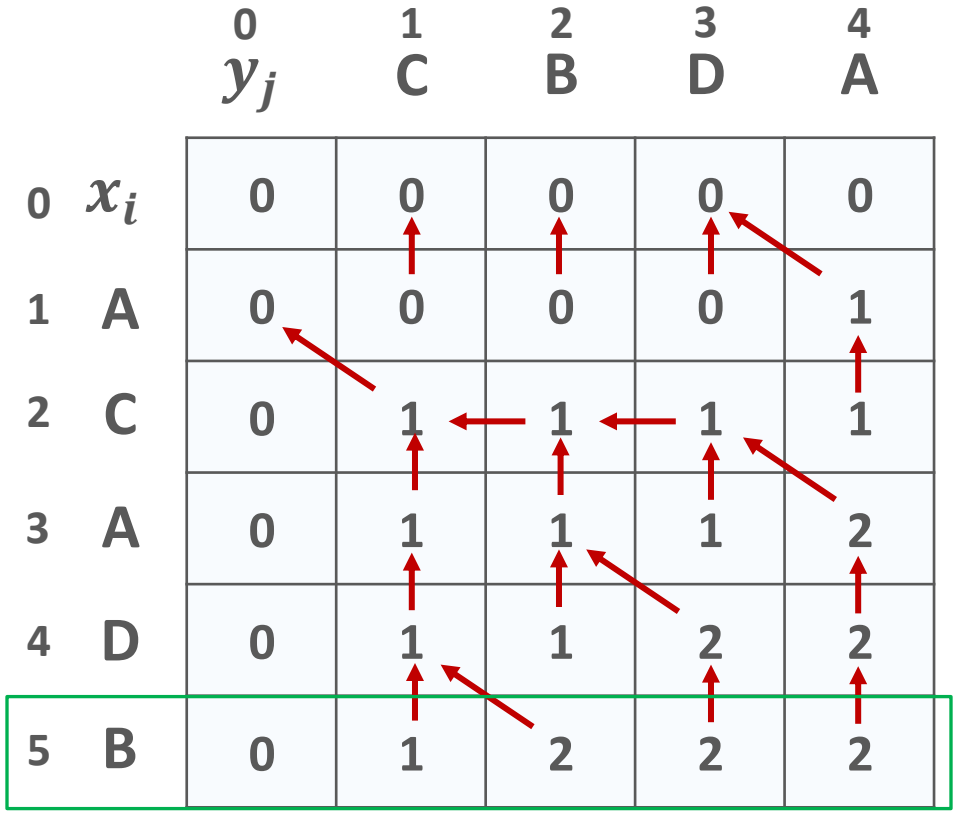
- Compare x_4 = D in row 4 with each column y_1, y_2, y_3, and y_4.
  - x_4 = D and y_1= C do not match, so we take max value of top and left cell which is 1. The arrow points to the cell with maximum value.
  - x_4 = D and y_2= B do not match, so we take max value of top and left cell which is 1. The values are equal, so the arrow points to previous row (top cell).
  - x_4 = D and y_3= D matches, so we add 1 to the left top diagonal cell (1+1=2) and fill it in the current cell. The arrow points to the diagonal cell.
  - x_4 = D and y_4= A do not match, so we take max value of top and left cell which is 2. The values are equal, so the arrow points to previous row (top cell).



Fill the values

Longest Common Subsequence - Dynamic Programming

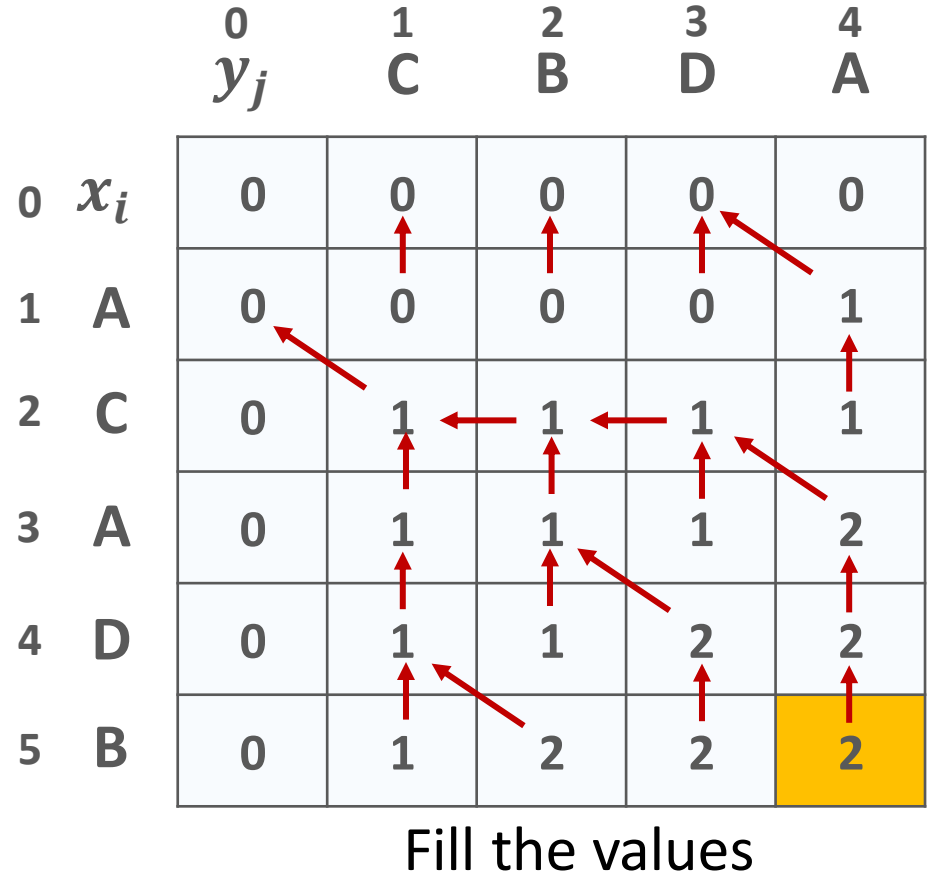
- Compare x_5 = B in row 5 with each column y_1, y_2, y_3, and y_4.
  - x_5 = B and y_1= C do not match, so we take max value of top and left cell which is 1. The arrow points to the cell with maximum value.
  - x_5 = B and y_2= B matches, so we add 1 to the left top diagonal cell (1+1=2) and fill it in the current cell. The arrow points to the diagonal cell.
  - x_5 = B and y_3= D do not match, so we take max value of top and left cell which is 2. The values are equal, so the arrow points to previous row (top cell).
  - x_5 = B and y_4= A do not match, so we take max value of top and left cell which is 2. The values are equal, so the arrow points to previous row (top cell).



Fill the values

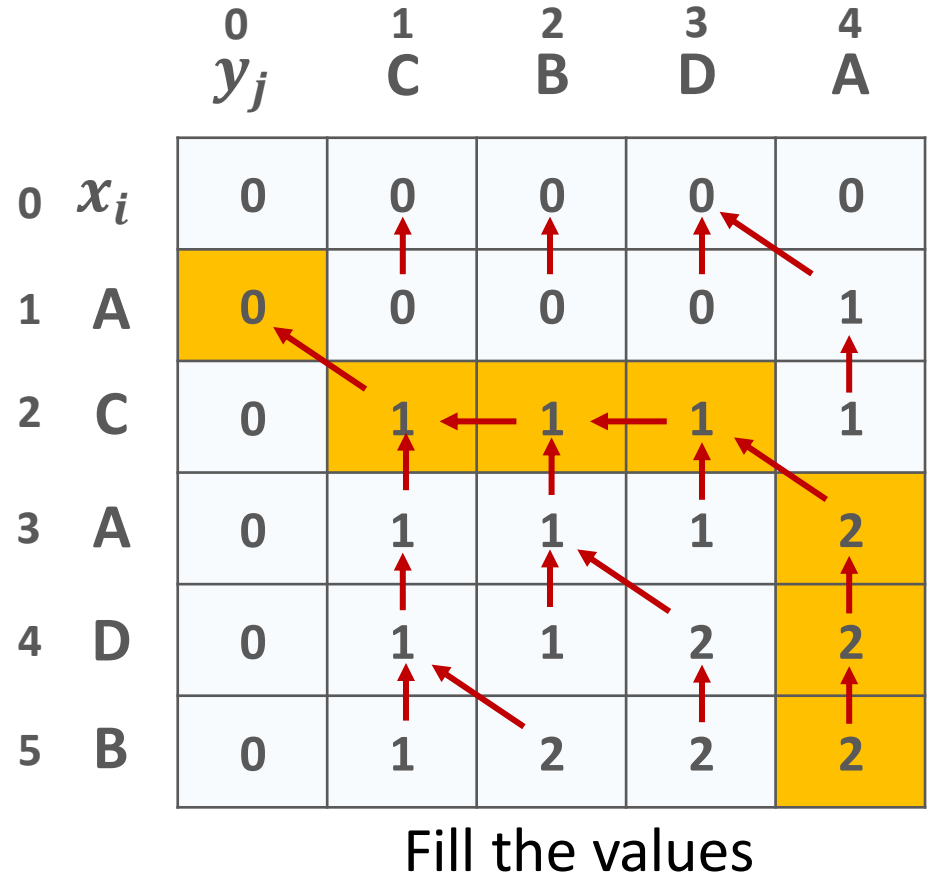
Longest Common Subsequence - Dynamic Programming

- Step 3: The value in the last row and the last column is the length of the longest common subsequence.
- Here the length of the longest common subsequence (LCS) is 2.



Longest Common Subsequence - Dynamic Programming

- Step 4: To find the longest common subsequence, start from the last element and follow the direction of the arrow.



Fill the values

Longest Common Subsequence - Dynamic Programming

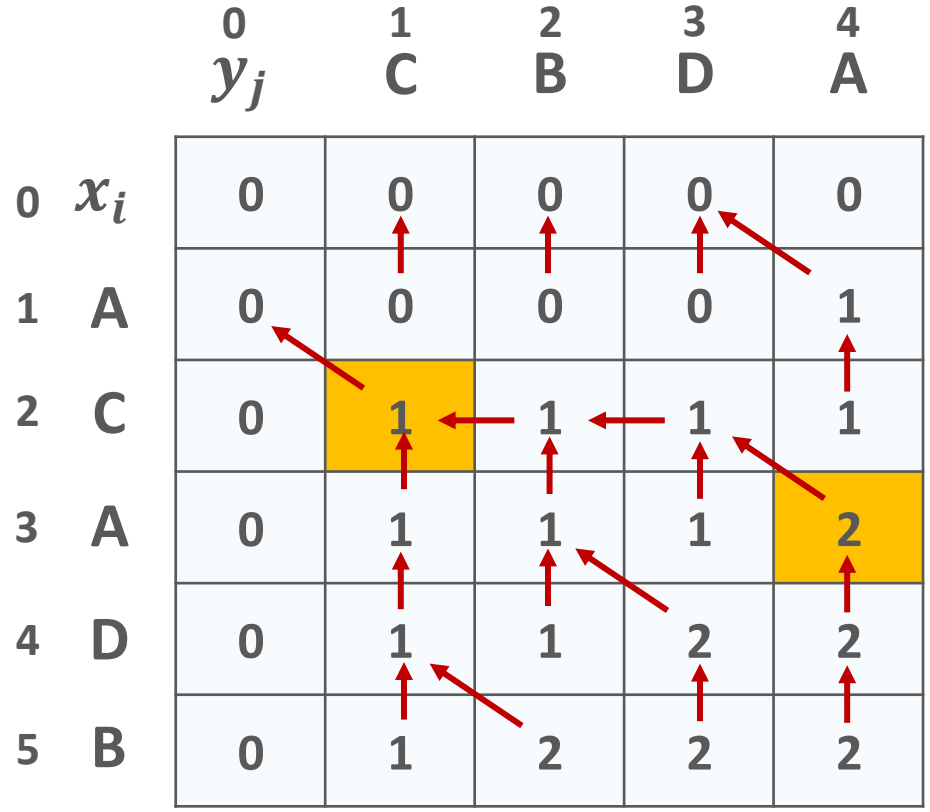
- The elements corresponding to ("↖") symbol form the longest common subsequence i.e., select the cells with diagonal arrow.

- Here C and A has diagonal arrows.

- The longest common sequence for

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

  is $\langle C, A \rangle$



Fill the values

LCS-LENGTH procedure

```
LCS-LENGTH(X, Y, m, n)
 1  let b[1 : m, 1 : n] and c[0 : m, 0 : n] be new tables
 2  for i = 1 to m
 3      c[i, 0] = 0
 4  for j = 0 to n
 5      c[0, j] = 0
 6  for i = 1 to m                  // compute table entries in row-major order
 7      for j = 1 to n
 8          if x_i == y_j
 9              c[i, j] = c[i − 1, j − 1] + 1
10              b[i, j] = "↖"
11          elseif c[i − 1, j] ≥ c[i, j − 1]
12              c[i, j] = c[i − 1, j]
13              b[i, j] = "↑"
14          else c[i, j] = c[i, j − 1]
15              b[i, j] = "←"
16  return c and b
```

LCS-LENGTH procedure – explanation (1)

- Initialization:
  - Inputs are sequences X and Y of lengths `m` and `n`.
  - Create tables `c[0..m, 0..n]` and `b[1..m, 1..n]` .

Table c for lengths and b for directions of LCS

- Base Case Setup:
  - Set the first column `c[i,0]` and the first-row `c[0,j]` to `0`.

- Filling the Table:
  - If characters match ($x_i == y_j$)
    - set `c[i, j] = c[i−1, j−1] + 1` and point `b[i, j]` diagonally.

LCS-LENGTH procedure – explanation (2)

- Filling the Table:
  - If characters do not match, compare values from the top `c[i−1, j]` and left `c[i, j−1]`:
    - If top is greater or equal, set `c[i, j] = c[i−1, j]` and point `b[i, j]` up.
    - Otherwise, set `c[i, j] = c[i, j−1]` and point `b[i, j]` left.

- Return Result:
  - Finally, return the tables c and b with the lengths and directions of LCS.

PRINT-LCS procedure

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return                    // the LCS has length 0
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i                 // same as y_j
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

PRINT-LCS procedure – explanation (1)

- Base Case Setup:
  - If either index i or j is zero, the function stops and returns because we've reached the end of one sequence.
- Handle Diagonal Arrow (↖):
  - If the current cell points diagonally `(b[i, j] == "↖")`, it means the characters x_i and y_j are part of the LCS.
  - Recursively call the function for the previous diagonal cell `(i-1, j-1)`, then print the character x_i.
- Handle Up Arrow (↑):
  - If the current cell points up `(b[i, j] == "↑")`, it means the character x_i is not part of the LCS.
  - Recursively call the function for the cell directly above `(i-1, j)`.

PRINT-LCS procedure – explanation (2)

- Handle Left Arrow (←):
  - If the current cell points left (`b[i, j] == "←"`), it means the character y_j is not part of the LCS.
  - Recursively call the function for the cell directly to the left (`i, j-1`).

LCS-LENGTH procedure – explanation (2)

- The figure in the next slide shows the tables produced by LCS-LENGTH on the Sequences $X=\langle A, B, C, B, D, A, B\rangle$ and $Y=\langle B, D, C, A, B, A\rangle$.
- The running time of the procedure is $\Theta(m\ n)$, since each table entry takes $\Theta(1)$ time to compute.

Tables produced by LCS-LENGTH - Example

The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $C[i, j]$ and the appropriate arrow for the value of $b[i, j]$.

The entry 4 in $c[7, 6]$ is the length of an $LCS\langle B, C, B, A \rangle$ of X and Y.

To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner, as shown by the sequence shaded blue. Each "↖" on the shaded-blue sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.