UVA DATA SCIENCE

Foundation of Computer Science for Data Science

Advanced algorithm II Dynamic Programming

Mai Dahshan

November 4, 2024

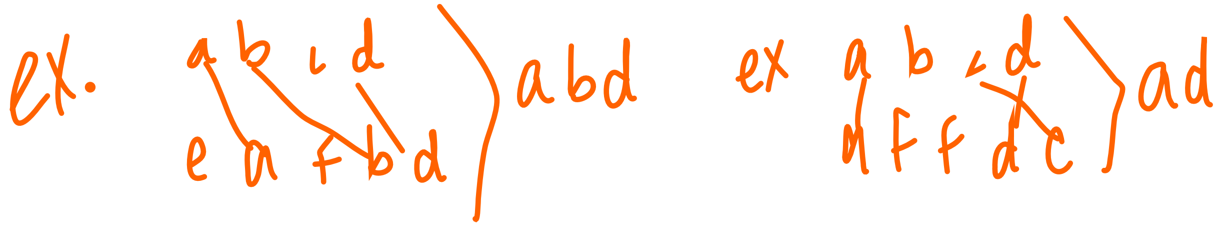
Two-dimensional Dynamic Programming

Two-dimensional Dynamic Programming

- **Two-dimensional dynamic programming (2D DP)** is a technique used when solving problems that require tracking two variables or indices.
- This approach is particularly useful for problems where a decision at one point depends on two previous subproblems, such as comparing two sequences or considering a grid layout.
- 2D array or matrix is needed to store computed result

Longest Common Subsequence (LCS)

- In **Longest Common Subsequence (LCS)** problem, the objective is to find the longest subsequence common to two strings.
- Given **two strings**: string S of length ***n***, and string T of length ***m***
- The goal is to produce their longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings



We are looking for strings that have letters appearing in the same consecutive order. Here, we did not include c b/c c did not come before d in both strings.

Longest Common Subsequence (LCS)

- LCS have many applications:
  - Molecular biology, DNA sequences
  - File comparison
  - more

Use of LCS in Molecular biology

- DNA sequences (genes) can be represented as sequences of four letters **ACGT**, corresponding to the four sub-molecules forming DNA
- Comparison of two sequences (human beings, animals or plants) by finding LCSs we can find
    - patterns of diseases
    - closeness of relationship among individuals
    - cultural heritage
    - relation of a crime scene to an individual
    - etc …

Use of LCS in File Comparison

- The Unix program "diff" is used to compare two different versions of the same file, to determine what changes have been made to the file
- It works by finding a longest common subsequence of the lines of the two files; any line in the subsequence has not been changed, so what it displays is the remaining set of lines that have changed

Longest Common Subsequence (LCS)

- Let there be string1: S = ABAZDC
- Let there be string 2: T = BACBAD

- What is the longest common subsequence? (and how long is it?)

UVA DATA SCIENCE

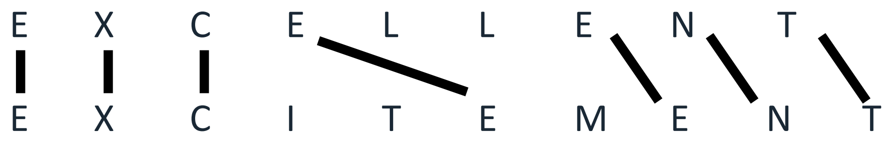
Longest Common Subsequence (LCS)

we want the LCS

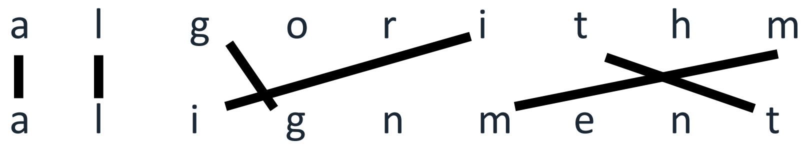
- Let there be string1: S = ABAZDC
- Let there be string 2: T = BACBAD
- In this case, the LCS has length 4 and is the string **ABAD**.
- Another way to look at it is we are finding a one-to-one matching between *some* of the letters in S and *some* of the letters in T

Longest Common Subsequence (LCS)

S 1: EXCELLENT, S 2: EXCITEMENT, LCS = EXCEENT



S 1: ALGORITHM, S 2: ALIGNMENT, LCS = ALGT



- Note the crossed lines. The 2nd symbol that cause the lines to be crossed is **not** added. In the example above, "**i**" and "**m**" are <u>not</u> added

Longest Common Subsequence (LCS)-Brute Force Solution

- For every subsequence of ***x***, check if it is a subsequence of ***y***

- Analysis :
  - There are 2^m subsequences of ***x***
  - Each check takes O(n) time, since we scan ***y*** for first element, and then scan for second element, etc.
  - The worst case running time is $O(n2^m)$ = *Exponential!*
  - Example – For S 1: algorithm, S 2: alignment = $9*2^9$ = 4608 comparisons!

Longest Common Subsequence (LCS)

- **Best Case**
    - If either string X or Y is empty, the LCS is zero because there are no common characters.
    - If m = 0 or n = 0, then LCS(X[0:m], Y[0:n]) = 0
    - This is the simplest case where no further recursion or calculation is needed, as an empty string has no subsequence

Longest Common Subsequence (LCS)

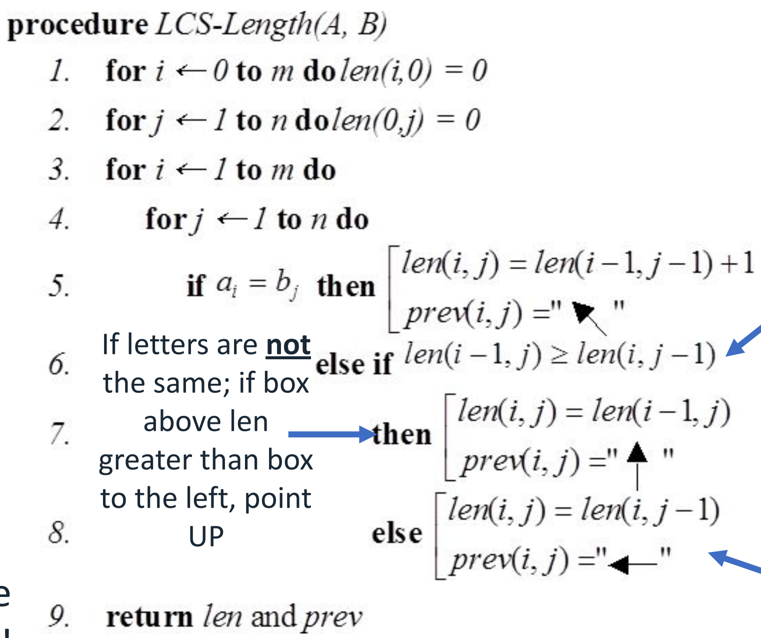
- **Recursive Case**
  - For each character in X and Y, we have two choices:
  - **Characters match**: If X[m-1] == Y[n-1], then this character is part of the LCS. We add 1 to the LCS length and look at the remaining parts of both strings
  - **Characters do not match**: If X[m-1] != Y[n-1], then the LCS could either include X[m-1] or Y[n-1], but not both. We then explore two subproblems:
    - Exclude the last character of X and find LCS(X[0:m-1], Y[0:n]).
    - Exclude the last character of Y and find LCS(X[0:m], Y[0:n-1]). We take the maximum result of these two subproblems

Longest Common Subsequence (LCS)

- The recursive relation can be defined as:

$$LCS(i,j) = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ 1 + LCS(i-1, j-1), & \text{if } S1[i-1] = S2[j-1] \\ \max(LCS(i-1, j), LCS(i, j-1)), & \text{if } S1[i-1] \neq S2[j-1] \end{cases}$$

Longest Common Subsequence (LCS)



```
procedure LCS-Length(A, B)
1.  for $i \leftarrow 0$ to $m$ do $len(i,0) = 0$
2.  for $j \leftarrow 1$ to $n$ do $len(0,j) = 0$
3.  for $i \leftarrow 1$ to $m$ do
4.      for $j \leftarrow 1$ to $n$ do
5.          if $a_i = b_j$ then $\left[\begin{array}{l} len(i,j) = len(i-1,j-1)+1 \\ prev(i,j) = "\nwarrow" \end{array}\right.$
6.          else if $len(i-1,j) \geq len(i,j-1)$
7.              then $\left[\begin{array}{l} len(i,j) = len(i-1,j) \\ prev(i,j) = "\uparrow" \end{array}\right.$
8.              else $\left[\begin{array}{l} len(i,j) = len(i,j-1) \\ prev(i,j) = "\leftarrow" \end{array}\right.$
9.  return len and prev
```

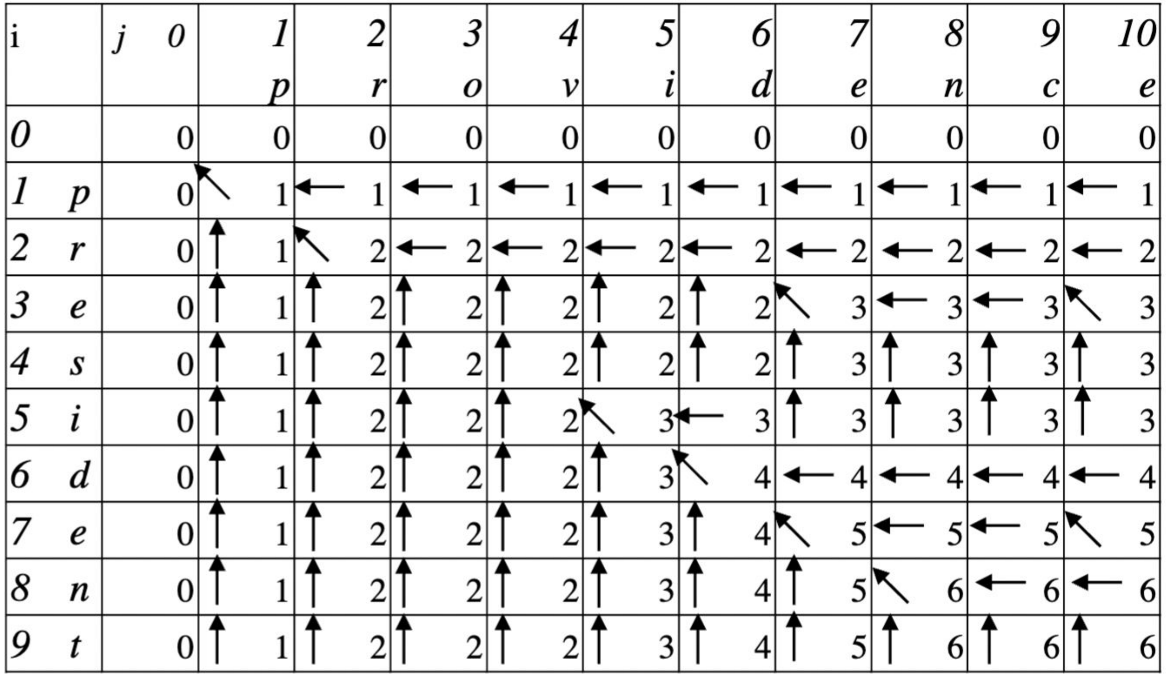
If letters are the same; **add 1** to upper left diagonal and draw arrow

If letters are **not** the same; if box above len greater than box to the left, point UP

If letters are **not** the same; if box to the left len greater than box to the top, point to the LEFT

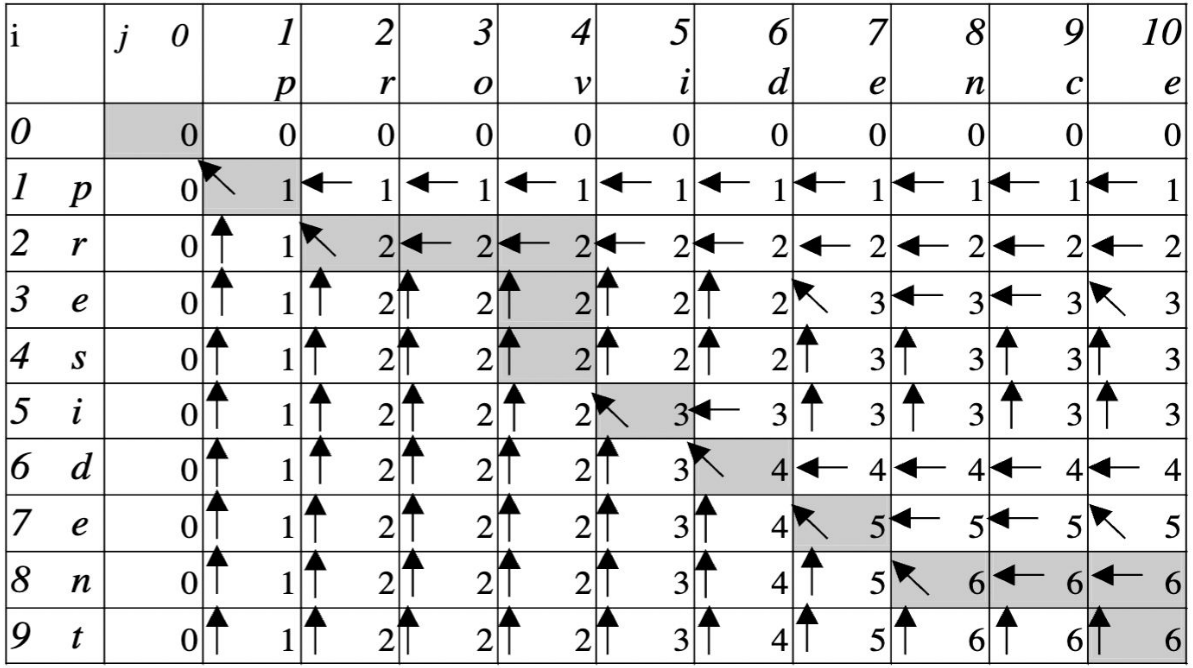
To break a tie (up and left have same len) then point UP!

Longest Common Subsequence (LCS) - Example



i	*j* *0*	*1* *p*	*2* *r*	*3* *o*	*4* *v*	*5* *i*	*6* *d*	*7* *e*	*8* *n*	*9* *c*	*10* *e*
0	0	0	0	0	0	0	0	0	0	0	0
1 *p*	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2 *r*	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4 *s*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5 *i*	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6 *d*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8 *n*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9 *t*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

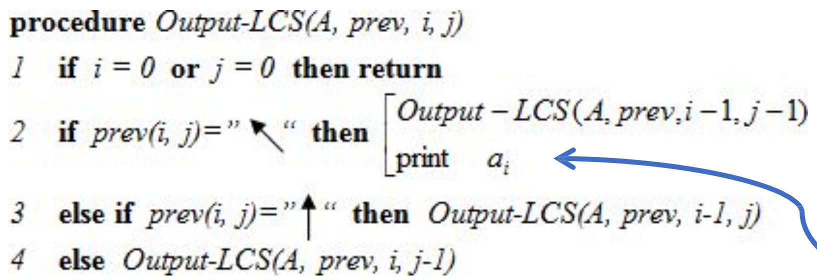
Longest Common Subsequence (LCS) - Example



i	j 0	1 p	2 r	3 o	4 v	5 i	6 d	7 e	8 n	9 c	10 e
0	0	0	0	0	0	0	0	0	0	0	0
1 p	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2 r	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3 e	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4 s	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5 i	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6 d	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7 e	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8 n	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9 t	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

Longest Common Subsequence (LCS) - Backtracking

- Apply the following algorithm to produce the output. Begin at the bottom right of the table, working backwards



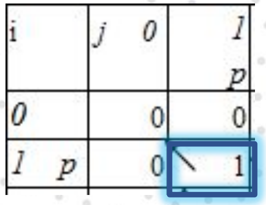
```
procedure Output-LCS(A, prev, i, j)
1  if $i = 0$ or $j = 0$ then return
2  if $prev(i, j)$ = "↖" then
       $Output-LCS(A, prev, i-1, j-1)$
       print $a_i$
3  else if $prev(i, j)$ = "↑" then Output-LCS(A, prev, i-1, j)
4  else Output-LCS(A, prev, i, j-1)
```

On Diagonals
Print the Letter!

- Final output is just imply reversed:
  - Utilizing this algorithm you'll get: **nedirp**
  - Reversing it, the final output is: **priden**

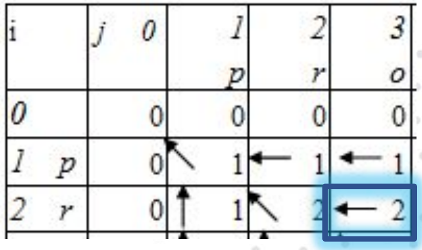
Longest Common Subsequence (LCS) - Example

- If the letters match ("p"&"p"), add one to the diagonal (0,0) and enter that value (**1**) in the box (1,1). Add a diagonal arrow



i	j 0	1 p
0	0	0
1 p	0	↖ 1

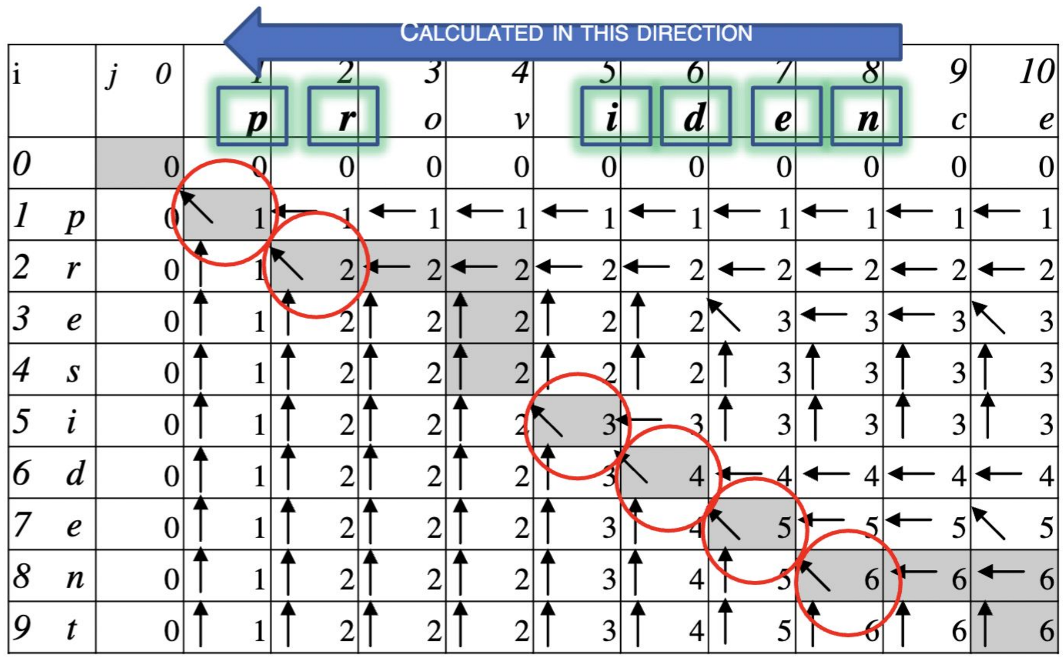
- If the letters do not match ("r"&"o"), pick the largest number of the diagonals (2,2) or (1,3) and draw an left arrow to the larger number. Will enter the value **2** in (3,3) (If diagonals are *equal* pick the top)



i	j 0	1 p	2 r	3 o
0	0	0	0	0
1 p	0	↖ 1	← 1	← 1
2 r	0	↑ 1	↖ 2	← 2

UVA DATA SCIENCE

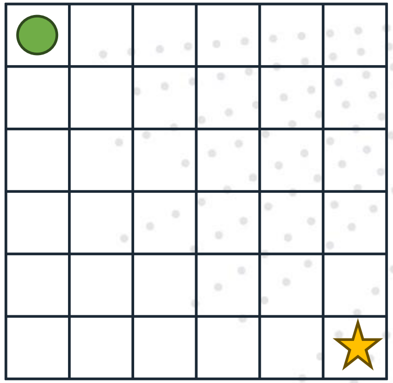
Longest Common Subsequence (LCS)


CALCULATED IN THIS DIRECTION

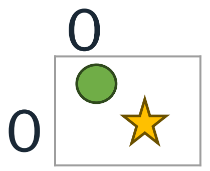
i	j	0	1	2	3	4	5	6	7	8	9	10
			p	r	o	v	i	d	e	n	c	e
0		0	0	0	0	0	0	0	0	0	0	0
1	p	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2	r	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3	e	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4	s	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5	i	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6	d	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7	e	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8	n	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9	t	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

Unique Pathfinding on Grid

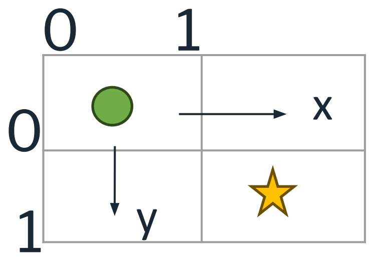
- Given grid of size **m x n** and need to find how many unique paths there are from the top-left corner **(0,0)** to the bottom-right corner **(m-1, n-1)**
- You can only move **down** or **right** at each step



Unique Pathfinding on Grid



1*1 Grid -> we only have one unique path



2*2 Grid

Number of unique paths = x+y
In this case, x=1 and y=1, # of unique paths =2

UVA DATA SCIENCE

Unique Pathfinding on Grid

- The best case occurs when you are already at the destination or when you only have one row or one column.
  - For example: If you're on the last row (m-1, j) or the last column (i, n-1), there is only one way to reach the destination because the only move available is to keep moving in one direction (right or down).
- In the **top-down approach**, when i == m - 1 or j == n - 1, the number of unique paths is 1
- In the **bottom-up approach**, we initialize the last row and last column as 1 because there's only one way to reach each cell from the destination

Unique Pathfinding on Grid

- **Recursive Case**:
  - The recursive case applies when you're not on the last row or column. For each cell (i, j), the number of unique paths is the sum of the unique paths from the cell below it (i+1, j) and the cell to the right of it (i, j+1).
    - This is expressed as: dp(i,j)=dp(i+1,j)+dp(i,j+1)

Unique Pathfinding on Grid

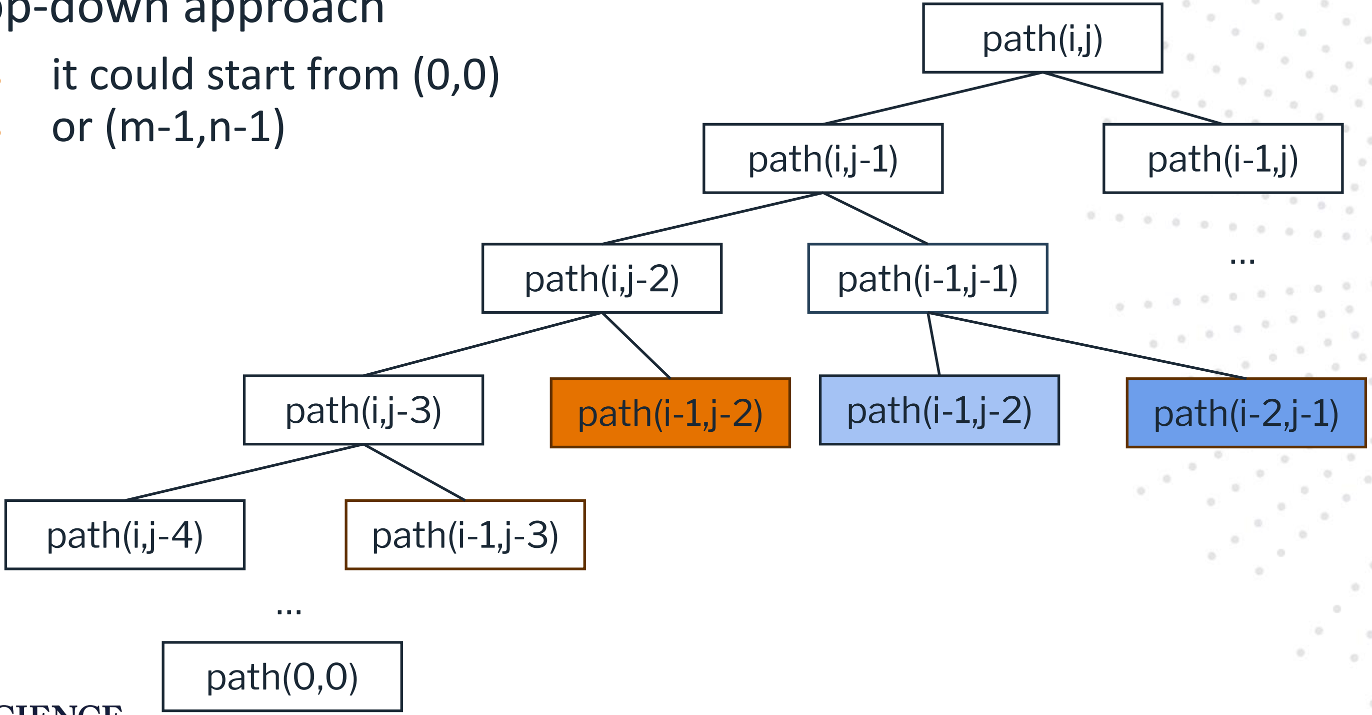
- **Overlapping Subproblems**:
  - In a brute force approach without memoization or tabulation, you would repeatedly calculate the same subproblem multiple times. For example, when calculating dp(i, j), you may need to compute dp(i+1, j) and dp(i, j+1) several times in the recursion tree
  - This is where **overlapping subproblems** come in: the same subproblems are being computed multiple times. Dynamic programming optimizes this by storing (memoizing) the results of each subproblem so that it does not recompute the same results

Unique Pathfinding on Grid

- **Optimal Substructure**:
  - The problem exhibits **optimal substructure** because the solution to the overall problem can be constructed from the solutions to its subproblems. Specifically, the number of unique paths to a cell (i, j) depends solely on the number of paths to its neighboring cells.

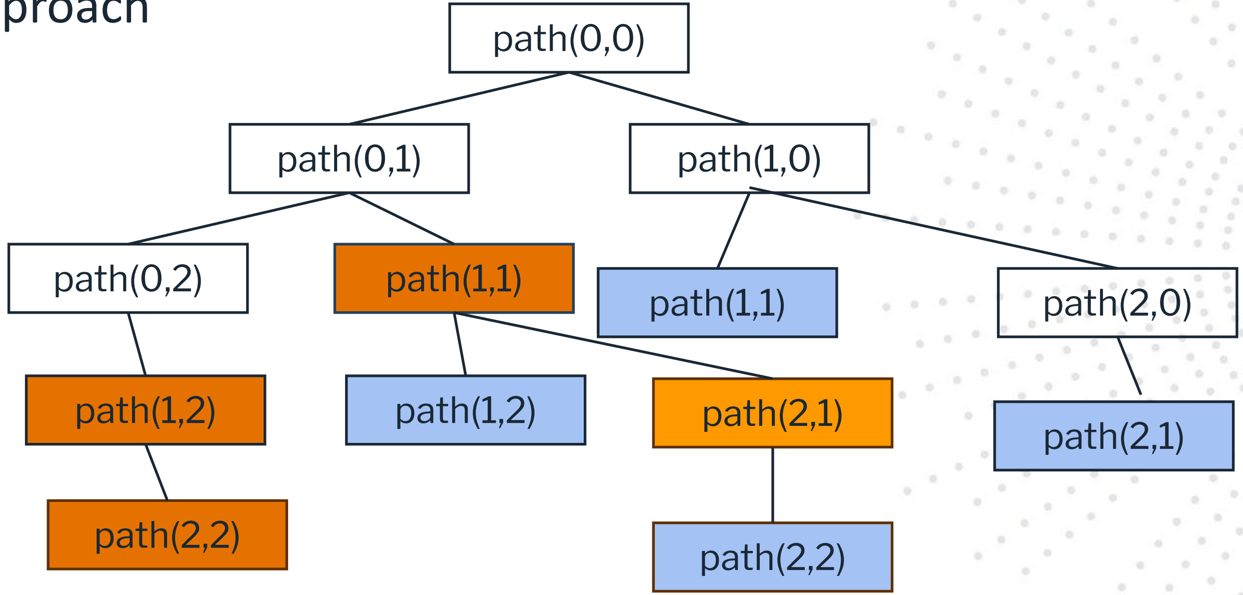
Unique Pathfinding on Grid

- Top-down approach
  - it could start from (0,0)
  - or (m-1,n-1)



Unique Pathfinding on Grid

- Top-down approach
  - 3*3 Grid



Unique Pathfinding on Grid

- **Steps to Solve**
  - **Initialize a DP table**: We define a DP table dp where dp[i][j] represents the number of unique paths to reach the cell (i, j).
  - **Base Case**:
    - There is **1 way** to reach the first row (dp[0][j]), because you can only move right
    - There is **1 way** to reach the first column (dp[i][0]), because you can only move down
  - **Recursive Relation**: For each cell (i, j), the number of unique paths to that cell is the sum of:
    - The paths coming from the cell directly above it (i-1, j) (if it exists).
    - The paths coming from the cell directly to the left of it (i, j-1) (if it exists).
    - So, the relation is:
      dp[i][j]=dp[i−1][j]+dp[i][j-1]

Unique Pathfinding on Grid

- Bottom - up
  - Initialize the table with 0 values, and set the base cases for the first row and first column to 1 (because there's only one way to reach these cells: moving either only right or only down)

1	1	1
1	0	0
1	0	0

Unique Pathfinding on Grid

Fill the rest of the DP table using the recursive relation.

1	1	1
1	2	3
1	3	6

Unique Pathfinding on Grid

- If you use the recursive relation
  - dp(i,j)=dp(i+1,j)+dp(i,j+1)
- **Base Case**
  - **Bottom-most row (i = m-1)**:
  - For any cell in the bottom row (dp[m-1][j]), there is only one way to reach the end: move **right**.
  - Therefore, all cells in the bottom row are initialized to 1.
  - **Right-most column (j = n-1)**:
  - For any cell in the right-most column (dp[i][n-1]), there is only one way to reach the end: move **down**.
  - Therefore, all cells in the right-most column are initialized to 1

Unique Pathfinding on Grid

- Bottom - up
  - Initialize the table with 0 values, and set the base cases for the last row and last column to 1 (because there's only one way to reach these cells: moving either only right or only down)

0	0	1
0	0	1
1	1	1

Unique Pathfinding on Grid

- Fill the rest of the DP table using the recursive relation.

6	3	1
3	2	1
1	1	1