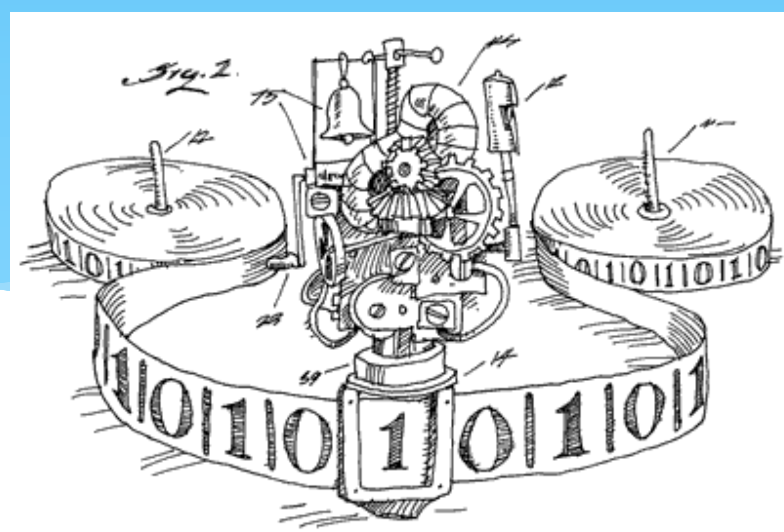
EECS 376: Foundations of Computer Science

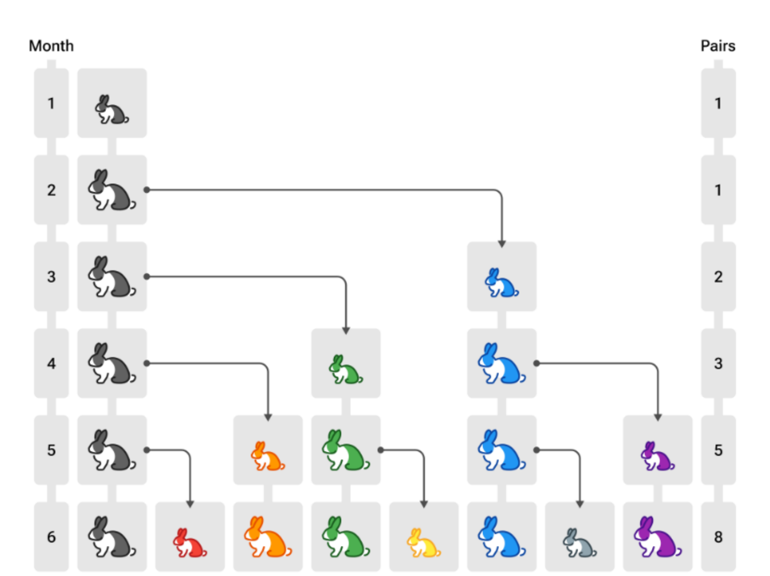
Seth Pettie

Lecture 4



“If you can solve it, it is an exercise; otherwise, it is a research problem”
-- Richard E. Bellman

Algorithmic Strategy: Dynamic Programming



Recap

* **Previously:** divide and conquer
  * A recurrence – break into smaller sub-problems and combine
  * Design goal is to minimize the number of recursive calls $\boldsymbol{k}$ and time to combine $O(\boldsymbol{n}^{\boldsymbol{d}})$
  * Examples: Closest pair, Karatsuba

Dynamic Programming

- **Today:** dynamic programming
  - A recurrence – break into smaller sub-problems and combine
  - ~~Design goal is to minimize the number of recursive calls~~ $\cancel{\boldsymbol{k}}$ ~~and time to combine~~ $\cancel{O(\boldsymbol{n}^{\boldsymbol{d}})}$
    - Don't worry about minimizing number of recursive calls!
  - **Idea: Maximize number of *repeated* recursive calls**

Dynamic Programming?

- Dynamic programming is *not*:
  - Dynamic, or programming!

"…It's impossible to use the word 'dynamic' in a pejorative sense…. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to." – Richard Bellman

Warm-Up: Fibonacci

- Recurrence for Fibonacci:

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

- Given a recurrence, three ways to <u>compute</u> its values:
- **Top-down recursive (naïve):** Starting at desired input, recurse down to base case(s)
- **Dynamic programming**
  - **Top-down with memoization:** Same as naïve, but save results as they're computed, reusing already-computed results
  - **Bottom-up table:** Start from base case(s), build up to desired result
- All these 'translate' the recurrence into an algorithm

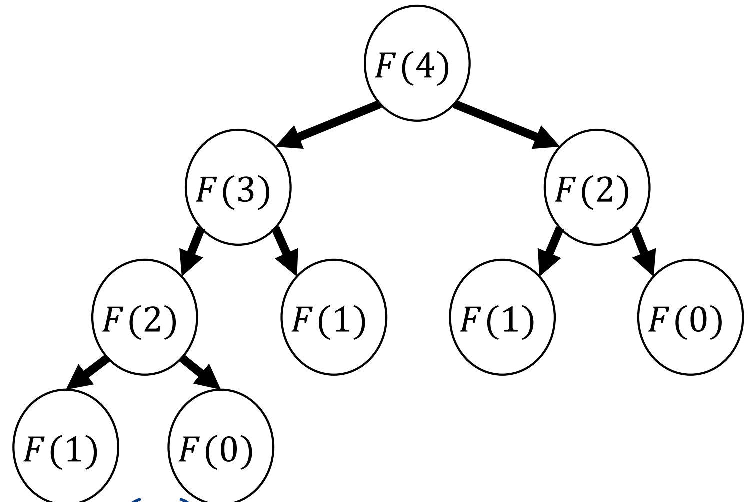
Fib: Naïve Implementation

- The xth Fibonacci number, for x a non-negative integer:

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

- **Top-down recursive (naïve):**

F(n): // n≥ 0 an integer
if $n = 0$ or $n = 1$ then return 1
return **F**$(n-1)$ + **F**$(n-2)$



- **Pro:** direct translation of recurrence
- **Con:** *exponential* runtime:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

$$= O\big(F(n)\big) = O(\varphi^n) = O(1.62^n)$$

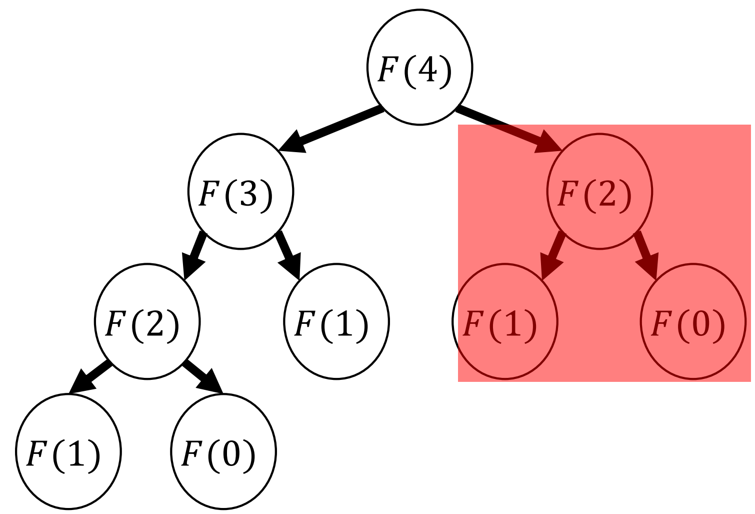
Fib: Memoization

- The xth Fibonacci number, for n a non-negative integer:

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

- **Top-down memoization:**

allocate $F[1..n]$ // entries initially NULL
$F[1] \leftarrow 1, F[2] \leftarrow 1$
M-F(n): // memoized implementation of F_n
if $F[n] = $ NULL then
    $F[n] \leftarrow$ **MF**$(n-1)$ + **MF**$(n-2)$
return $F[n]$



- **Pros:** much faster (but how much?)
- **Con:** requires accessing global memory, hard to analyze runtime

Fib: Bottom up (dynamic programming)

- Recurrence for Fibonacci:

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

- **Bottom-up Table:**

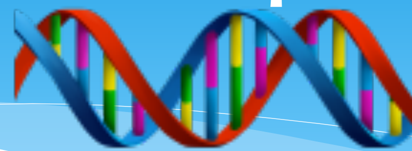
DP-F(x): // table implementation of F_n
allocate $F[1..n]$
$F[0] \leftarrow 1, F[1] \leftarrow 1$
for $i = 2..n$
  $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

x	F[x]
0	1
1	1
2	2
3	3
4	5
5	8
6	13
7	21

- **Q:** What is the runtime of this algorithm?
- $T(n) = O(n)$

- **Pro:** much faster, no globals, easier to analyze runtime
- **Cons:** must compute *entire* table of smaller results (but usually end up doing this anyway, in every strategy)

DNA Comparison



- Your DNA is a (*long*) string over {A, T, C, G}.
  - Small chance of random insertions, deletions, edits
- "Humans and chimps are 98.9% similar."
  - X: ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
  - Y: GTCGTTCGGAATGCCGTTGCTCTGTAA
- The length of the *longest common subsequence* between two genomes is a measure of *similarity*.
- How efficiently can we compute an LCS of X, Y?
  - |human genome| $\approx$ 3bil, |chimp genome| $\approx$ 2.8bil

Longest Common Subsequence

- Given strings $X[1..m]$ and $Y[1..n]$
- **Goal:** find the <u>*length*</u> of a ***longest common subsequence*** of X and Y
  - A **subsequence** of X is a string obtainable from X by deleting chars
  - A **common subsequence** of X and Y is a subsequence of both X and Y
- **Example:** "CT" is a common subsequence of "CGATG" and "CATGT". **Q:** What's the longest?
- **Q:** What's a brute force solution?
  - Each character of X and Y is either deleted or not: Runtime: $O(2^{m+n})$

Better way?

- Where to begin? If we can just find one pair of characters that *must* be matched in an LCS, then we can delete those characters and recurse
- **Idea 1:** Given $X[1..m]$ and $Y[1..n]$, suppose last characters are the same, i.e. $X[m] = Y[n]$
- **Q:** Should we always match X[m] and Y[n] ?
  - **Example:** X="***A***CTG", Y= "AT***A***G". **Q:** Is matching the blue G's a mistake? What about matching the red A's?
  - Yes, if $X[m] = Y[n]$. Why? (Can matching X[m] and Y[n] ever be a mistake?)

EECS ECE CSE

Better way?

- Given strings $X[1..m]$ and $Y[1..n]$
- Let $LCS(i, j)$ denote the *length* of a longest common subsequence of $X[1..i]$ and $Y[1..j]$. ($i = 0$ or $j = 0$ denotes the empty string)
- **Idea 2:** If $LCS(i, j - 1) = LCS(i, j) - 1$ or $LCS(i - 1, j) = LCS(i, j) - 1$, then the match $X[i]$ with $Y[j]$ is in a longest common subsequence of $X[1..i]$ and $Y[1..j]$.
  - We don't have to find the longest common subsequence directly, all we need is to find the *length* of the longest common subsequence, $LCS(i, j)$!
- **Example:** Suppose $X =$ "ATGCC" and $Y =$ "TAGC".
  - **Q:** What's $LCS(1,0)$?
  - **Q:** What's $LCS(5,3)$?
  - **Q:** What's $LCS(4,4)$?
  - **Q:** What's $LCS(5,4)$?

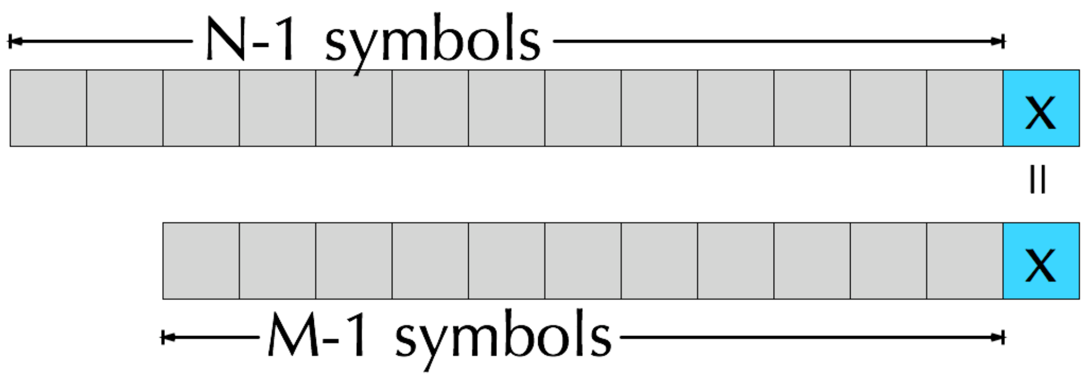
Recurrence for LCS

- **Step 1:** ***To find a dynamic programming algorithm, always start with a recurrence***
- We want a recurrence for $LCS(i, j)$
- **Q:** What should the base case be?
  - **Base case:** if $i = 0$ or $j = 0$ (empty string); $LCS(i, j) = 0$

- **Case 1:** $X[i] = Y[j]$ (ends with the same character)
  - **Example:** $X[1..i] =$ "CTGC**A**" and $Y[1..j] =$ "TCG**A**"
  - $LCS(i, j) = 1 + LCS(i - 1, j - 1)$

Recurrence for LCS

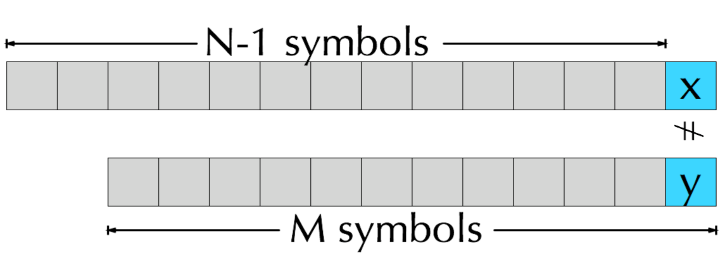
* **Case 1:** $X[i] = Y[j]$ (ends with the same character)
  * **Example:** $X[1..i] =$ "CTGCA" and $Y[1..j] =$ "TCGA"
  * $LCS(i, j) = 1 + LCS(i-1, j-1)$

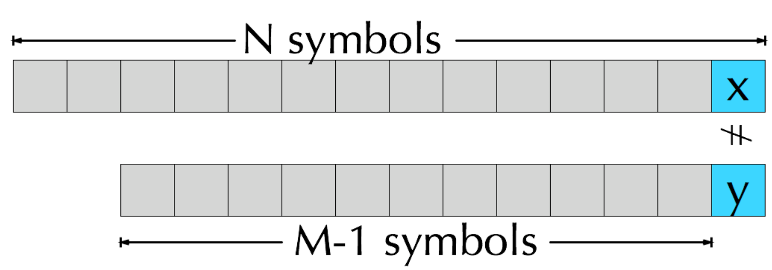


Recurrence for LCS

$$LCS(i,j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ 1 + LCS(i-1, j-1) & X[i] = Y[j] \\ ? & X[i] \neq Y[j] \end{cases}$$

- **Case 2:** $X[i] \neq Y[j]$ (end with different characters)
  - **Example:** $X[1..i] =$ "GTCA" and $Y[1..j] =$ "GTC"
  - *At least* one of the letters is *not* part of LCS
  - **Q:** How do we know which one?
    - Try both! $LCS(i,j) = \max\{LCS(i-1,j), LCS(i,j-1)\}$





Recurrence for LCS

- Given strings $X[1..m]$ and $Y[1..n]$
- Let $LCS(i,j)$ denote the length of a longest common subsequence of $X[1..i]$ and $Y[1..j]$

$$\boxed{LCS(i,j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ 1 + LCS(i-1, j-1) & X[i] = Y[j] \\ \max\begin{Bmatrix} LCS(i-1,j), \\ LCS(i, j-1) \end{Bmatrix} & X[i] \neq Y[j] \end{cases}}$$

Q: Given this recurrence, how do we find the length of an LCS of X and Y?

$LCS(m,n)$

Table Implementation of LCS

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

LCS$(X[1..m], Y[1..n])$: // table implementation of LCS
allocate $L[0..m][0..n]$
$L[0][1..n] \leftarrow 0, L[1..m][0] \leftarrow 0$
for $i = 1..m$ and $j = 1..n$: // fill columns of row i, one by one
  if $X[i] = Y[j]$ **then** $L[i][j] \leftarrow 1 + L[i-1][j-1]$
  else $L[i][j] \leftarrow \max\{L[i-1][j], L[i][j-1]\}$
return $L[m][n]$

Runtime: $O(mn)$

Q: How could we recover actual LCS, not just its length?
Store "prev" pointer in $L[i][j]$ depending on case.

Filling the table

* Try this visualization out! https://www.cs.usfca.edu/~galles/visualization/DPLCS.html

Longest Increasing Subsequence

(A classic coding problem)

* Given an array of integers $A[1..n]$
* **Goal:** Find the length of a ***longest increasing subsequence*** of A
  * largest inc. array obtainable by deleting parts of A
* **Example:** $[5,6,7]$ is an increasing subsequence of $[\mathbf{5}, \mathbf{6}, 0, \mathbf{7}, 1,2,0,4,0]$. **Q:** longest?
* **Q:** What's a brute force solution?
  * Each integer is either deleted or not, $O(2^n)$ time

Recurrence for LIS?

* Given an array of integers $A[1..n]$
* Let $LIS(i)$ be the length of a longest increasing subsequence of $A[1..i]$
* **Q:** What's $LIS(4)$ if $A = [1,1,2,1,3]$? $A = [5,6,8,2,3]$?
* Before: divided between last element(s) and rest of list(s). Can we do the same here?
* **Q:** Can we determine if $A[i]$ extends LIS of $A[1..i-1]$ by only looking at $A[i]$ and $A[i-1]$?
  * **Example:** $A[1..i-1] = [5,6,8,2]$, $A[i] = 3$
  * **No.** We *need more information* before we can conclude that $LIS(i) = 1 + LIS(i-1)$

Recurrence for LIS_{at}?

The subproblems we solve depend on how we solve the problem recursively

* Given an array of integers $A[1..n]$
* Let $LIS_{at}(i)$ be the length of a longest increasing subsequence of $A[1..i]$ that ends at $A[i]$
* **Q:** What's $LIS_{at}(4)$ if $A = [1,1,2,1,3]$? $A = [5,6,8,2,3]$?
* **Q:** Can we determine if $A[i]$ extends LIS of $A[1..j]$ ending at $A[j]$ by only looking at $A[i]$ and $A[j]$?
  * Example: $A[1..j] = [1,1,2], A[i] = 3$
  * **Yes.** If $A[i] > A[j]$, then

$$LIS_{at}(i) \geq 1 + LIS_{at}(j)$$

Recurrence for LIS_{at}

$$LIS_{at}(i) = \begin{cases} 0 & i = 0 \\ 1 + \max\left\{ LIS_{at}(j) \;\middle|\; \begin{array}{c} (A[j] < A[i] \text{ and } j < i) \\ \text{or } j = 0 \end{array} \right\} & i \neq 0 \end{cases}$$

LIS$(A[1..n])$: // table implementation of LCS
allocate $L[0..n]$
$L[0] \leftarrow 0$
for $i = 1..n$: // fill table
  $l \leftarrow 0$
  for $j = 1..i-1$:
    if $A[j] < A[i]$: $l \leftarrow \max\{l, L[j]\}$
  $L[i] \leftarrow l + 1$
return ?

- The conversion from recurrence to table is *mechanical*
- **Q:** Given this recurrence, how do we determine the length of a LIS?

Recurrence for LIS_{at}

$$LIS_{at}(i) = \begin{cases} 0 & i = 0 \\ 1 + \max\left\{ LIS_{at}(j) \;\middle|\; \begin{matrix} (A[j] < A[i] \text{ and } j < i) \\ \text{or } j = 0 \end{matrix} \right\} & i \neq 0 \end{cases}$$

LIS$(A[1..n])$: // table implementation of LCS
allocate $L[0..n]$
$L[0] \leftarrow 0$
for $i = 1..n$: // fill table
  $l \leftarrow 0$
  for $j = 1..i-1$:
    if $A[j] < A[i]$: $l \leftarrow \max\{l, L[j]\}$
  $L[i] \leftarrow l + 1$
return $\max_{1 \le i \le n} L[i]$

Runtime: $O(n^2)$

- The conversion from recurrence to table is *mechanical*
- **Q:** Given this recurrence, how do we determine the length of a LIS?
  - $LIS(n) = \max_{1 \le i \le n} LIS_{at}(i)$
- **Q:** Runtime?