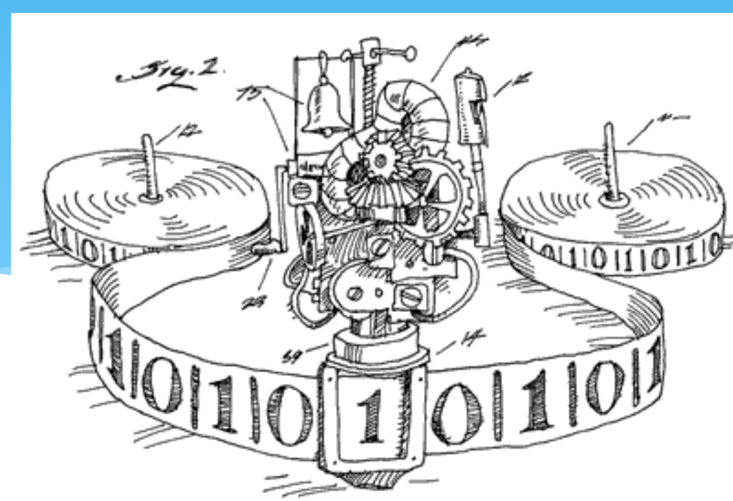
EECS 376: Foundations of Computer Science

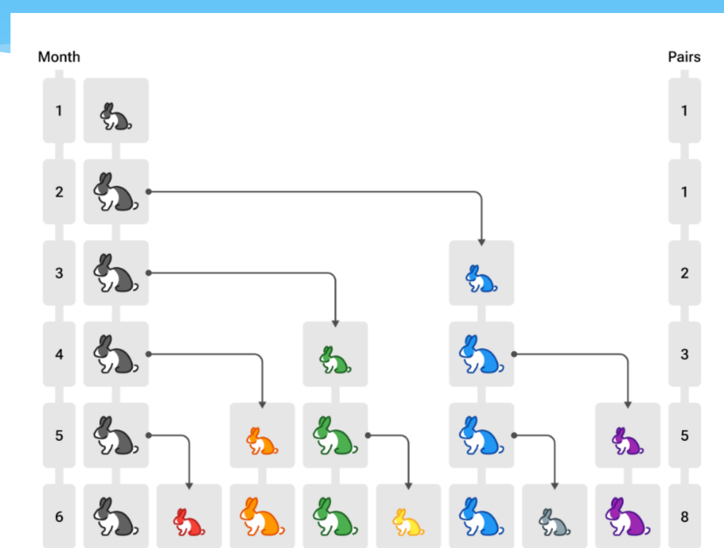
Chris Peikert

18 January 2023



"If you can solve it, it is an exercise; otherwise, it is a research problem"
- Richard E. Bellman,
The Primary Expositor of Dynamic Programming

Algorithmic Strategy:
Dynamic Programming



"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research…

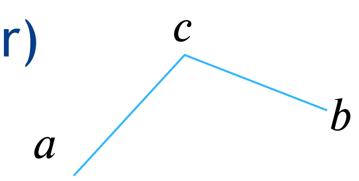
I had to do something to shield [SecDef] Wilson and the Air Force from the fact that I was really doing mathematics…

I was interested in planning, in decision making, in thinking. But planning is not a good word… I decided to use the word, '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."

Dynamic Programming



High-level Idea: Break a big problem into smaller (easier) subproblems—like D&C—but also exploiting:

1. Principle of "optimal substructure":
   any "piece" of an optimal structure is itself optimal.

Example: A subpath of any shortest path is itself a shortest path between its endpoints.

2. Overlapping sub-problems: "many" subproblems re-occur "many" times.

Example: When computing the Fibonacci sequence using the rule $F_n = F_{n-1} + F_{n-2}$, "many" recursive calls will be repeated.

Warm-Up: Fibonacci

* Recurrence for Fibonacci:

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

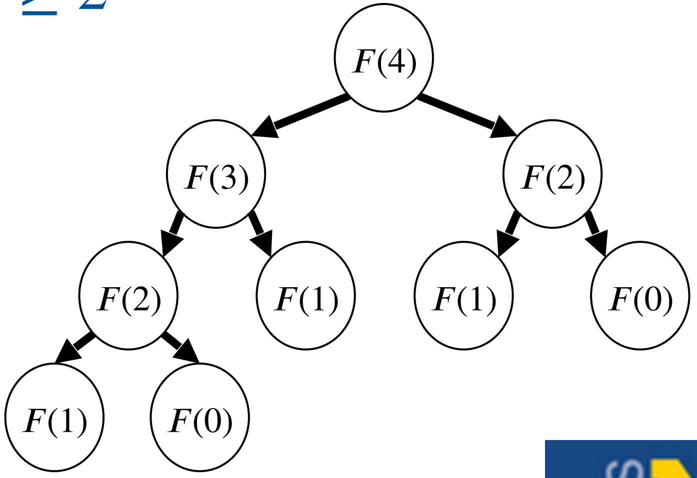
* Given a recurrence, three ways to <u>compute</u> its values:
* **Top-down Recursive (Naïve):** Starting at desired input, recurse down to base case(s)
* **Top-down w/ Memoization:** Same as naïve, but save results as they're computed, reusing already-computed results
* **Bottom-up Table:** Start from base case(s), build up to desired result
* All these are 'mechanical translations' of the recurrence

Fib: Naïve Implementation

* Recurrence for Fibonacci:

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

* **Top-down Recursive (Naïve):**
* **Algorithm:** $Fib(n)$
  * If $n = 0$ OR $n = 1$
    * **Return** 1
  * **Return** $Fib(n-1) + Fib(n-2)$
* Pro: direct translation of recurrence
* Con: *exponential* runtime: $T(n) = T(n-1) + T(n-2) + O(1)$

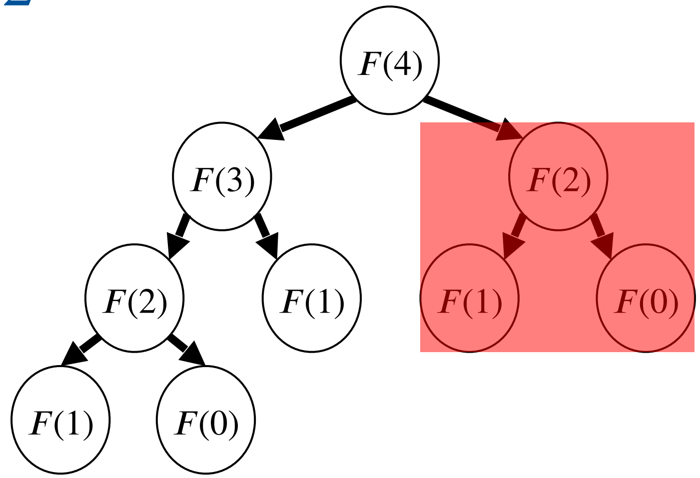


Fib: Top-Down w/Memoization

- Recurrence for Fibonacci:

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

- **Top-down Memoization:**
- $memo[0 \ldots n]$:= empty table
- **Algorithm:** $Fib(n)$
  - If $n = 0$ OR $n = 1$
    - **Return** 1
  - If $memo[n]$ undefined:
    - $memo[n] := Fib(n-1) + Fib(n-2)$
  - **Return** $memo[n]$
- **Pros:** much faster (but how much?)
- **Con:** global memo, clumsy impl., hard to analyze runtime



Fib: Bottom Up

* 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:**

1	1	2	3	5	8	13	21	34	55
0	1	2	3	4	5	6	7	8	9

* **Algorithm:** $Fib(n)$
  * allocate $table[0 \ldots n]$
  * $table(0) := 1$
  * $table(1) := 1$
  * For $i = 2$ to n:
    * $table(i) := table(i-1) + table(i-2)$
  * **Return** $table(n)$
* **Pro:** much faster, no globals, easy to analyze runtime
* **Cons(?):** must compute *entire* table of smaller results (but usually end up doing this anyway, in every strategy)

EECS ECE CSE

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

- **Definition:** A ***subsequence*** of a string s is a (not necessarily contiguous) subset of the characters of s, in their original order.
  - **Example:** for s = "Fibonacci sequence"
    - "Fun"
    - "seen"
    - "cse"
    - …
- **Goal:** Given strings $X[1..n]$ and $Y[1..m]$, find a ***longest common subsequence*** of X and Y.
  - Largest string obtainable from X *and* Y by deleting chars
- **Example:** "Gole" is an LCS of "Google" and "Go Blue".
- **Q:** What's a brute force solution?
  - Each character of X and Y is either deleted or not.

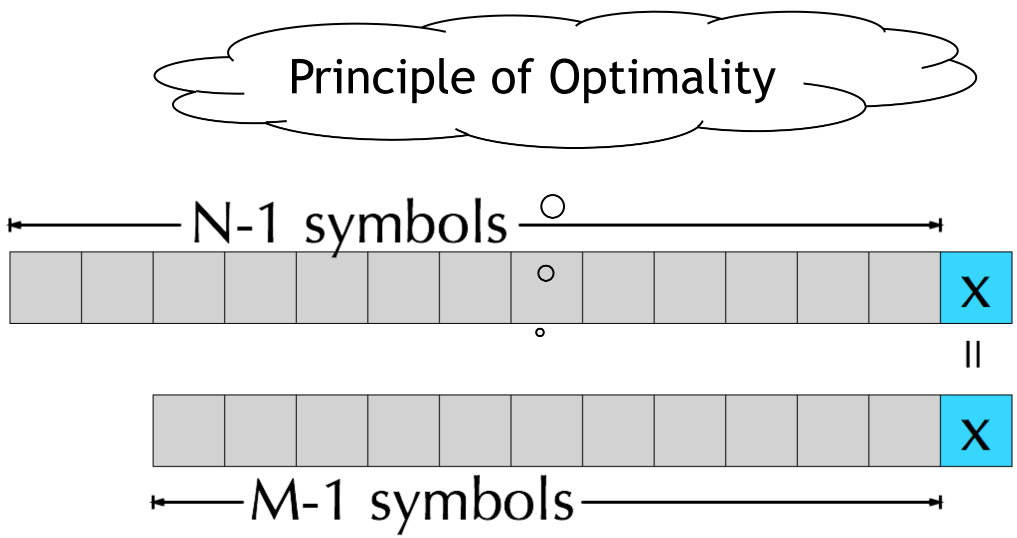
EECS ECE CSE

Dynamic Programming for LCS (and every other DP problem)

- Let $X[1 \ldots n]$ and $Y[1 \ldots m]$ be two given strings.
- **Key Idea #1:** to start, focus on the *length* of an LCS
  - (After finding length, finding an actual LCS will be easy!)
- **Key Idea #2:** discover a *recurrence* for LCS length, relative to suitable *substrings* (subproblems)
  - Which substrings to consider? How to relate them?
  - **An "art"!**
- Define $LCS(i, j) :=$ length of LCS of $X[1 \ldots i]$ and $Y[1 \ldots j]$.
  - Subproblems are (pairs of) *prefixes* of X and Y.

LCS Recurrence (Part 1)

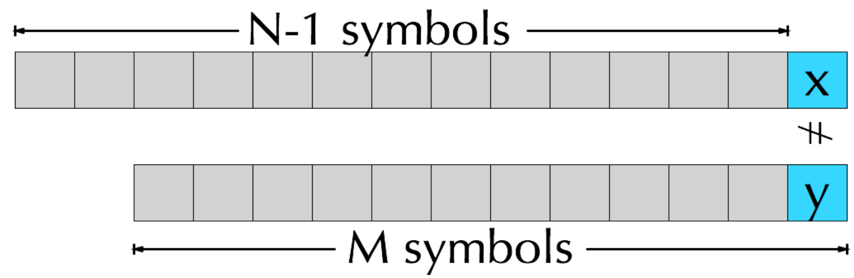
* $LCS(i, j) :=$ length of LCS of $X[1 \ldots i]$ and $Y[1 \ldots j]$.
* If the last characters are equal, i.e., $X[i] = Y[j]$:
  * $LCS(i, j) = 1 + LCS(i - 1, j - 1)$.



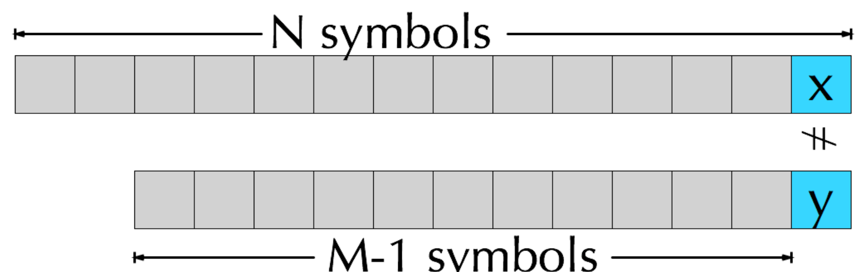
LCS Recurrence (Part 2)

* $LCS(i,j)$:= length of LCS of $X[1\ldots i]$ and $Y[1\ldots j]$.
* If the last characters are **not** equal, i.e., $X[i] \neq Y[j]$:
* $LCS(i,j)$ = Maximum of the only two options:

$LCS(i-1,j)$



and



$LCS(i,j-1)$

Full Recurrence for LCS

- $LCS(i, j)$ = LCS length of $X[1..i]$ and $Y[1..j]$.
- Then:

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

- **Naïve Implementation:** Exponential runtime!
- **Bottom up:** There are $O(nm)$ values of interest: $LCS(i, j)$ for $0 \leq i \leq n$ and $0 \leq j \leq m$ (overlapping sub-problems)

https://www.cs.usfca.edu/~galles/visualization/DPLCS.html

LCS Dynamic Programming in Action

* See hand-written notes (or visualization webpage):
  1. Bottom-up table filling
  2. Recovering an LCS itself from the lengths

Longest Increasing Subsequence

- Given: an array of numbers $A[1 \ldots n]$
- **Goal:** Find (the length of) a ***longest increasing subsequence*** of A.
  - Longest increasing array obtainable by deleting entries of A
- **Example:** $[5, 6, 7, 8]$ is an increasing subsequence of $[5, 6, 0, 7, 1, 2, 8, 4, 0, 5, 3]$.
  - **Q:** What's a longest one?
- **Q:** What's a brute force algorithm?
  - Each entry is either deleted or not: $\geq 2^n$ time!

Recurrence for LIS?

* Given an array of integers $A[1..n]$
* Let $LIS(i)$ be the length of a longest increasing subsequence of $A\left[1..i\right]$.
* **Q:** What's $LIS(4)$ if $A = [1,2,0,4,3]$? If $A = [1,2,3,0,5]$?
* **Q:** Can we determine whether $A[i]$ extends an LIS of $A[1..j]$ by only looking at $A[i]$ and $A[j]$?
  * **No.** We *need more information* to determine whether $LIS(i) \geq 1 + LIS(j)$.

Recurrence for LIS_{at}?

- 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 with $A[i]$.**
- **Q:** What's $LIS_{at}(4)$ if $A = [1,2,0,4,3]$?
  If $A = [1,2,3,0,5]$?
- **Q:** Can we determine if $A[i]$ extends an LIS of $A[1..j]$ that ends with $A[j]$ by only looking at $A[i]$ and $A[j]$?
  - **Yes.** If $A[i] > A[j]$, then $LIS_{at}(i) \geq 1 + LIS_{at}(j)$.

EECS ECE CSE

Recurrence for LIS_{at}

We don't know where the ***second-to-last*** element of the LIS ending at $A[i]$ is, so we consider every element that it *could* be!

* 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 with $A[i]$

$$LIS_{at}(i) = \begin{cases} 1 & \text{if } i = 1 \text{ or } A[j] \geq A[i] \text{ for all } j < i \\ 1 + \max\left\{ LIS_{at}(j) \mid A[j] < A[i] \text{ and } j < i \right\} & \text{otherwise} \end{cases}$$

Q: Given this recurrence, how to compute the length of an LIS of A? An LIS itself?

http://rosulek.github.io/vamonos/demos/lis.html

LIS Dynamic Programming in Action

* See hand-written notes (or visualization webpage):
  * A = [5,6,0,7,1,2,8,4,0,5,3]
  * Bottom-up table filling, with “back pointers”
  * Recovering an LIS itself from the table