Announcements

- HW 4 Due today

Today

- Dynamic Programming Introduction

Dynamic Programming (Ch 6)

- Background and past examples
- Longest Common Subsequence
- Knapsack
- Chain Matrix Multiplication
- All-Pairs Shortest Paths
- Independent Sets of Trees
- Travelling Salesman

Computing Fibonacci Numbers

Recall:

$$F_n = 1 \text{ if } n = 0 \text{ or } 1$$

$$F_n = F_{n-1} + F_{n-2} \text{ otherwise}$$

Naïve Algorithm

```
Fib(n)
  If n ≤ 1
     Return 1
  Else
     Return Fib(n-1)+Fib(n-2)
```

Far too slow!

Improved Algorithm

```
Fib2(n)
  Initialize A[0..n]
  A[0] = A[1] = 1
  For k = 2 to n
     A[k] = A[k-1] + A[k-2]
  Return A[n]
```

Tabulation of answers avoids runaway recursive calls.

Another Example

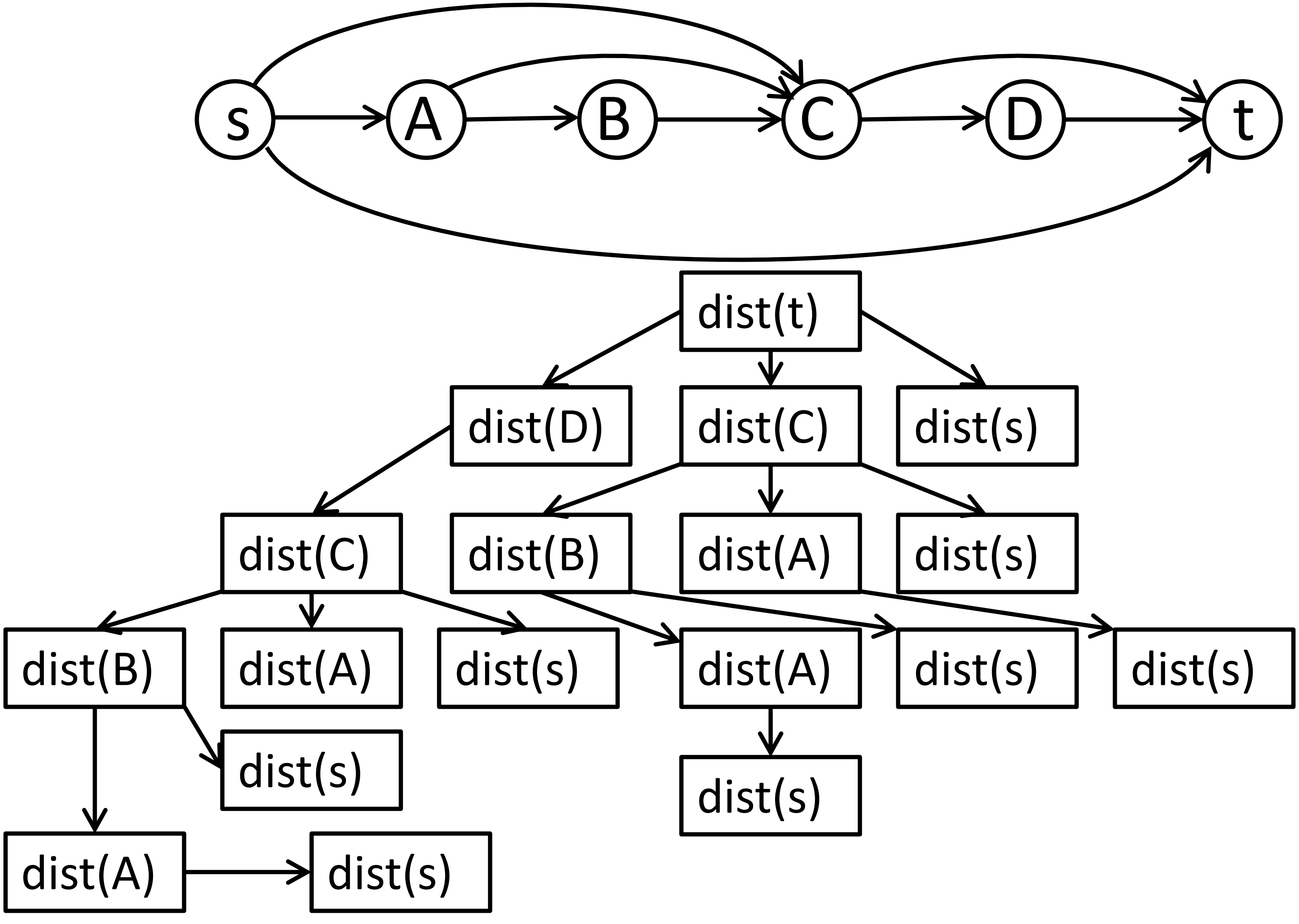
Something similar happens with our algorithm for shortest paths in DAGs.

This was based on the basic recursive formula

$$\mathrm{dist}(w) = \min_{(v,w)\in E} \mathrm{dist}(v) + \ell(v,w).$$

applied to vertices in topological order.

Example


s
A
B
C
D
t
dist(t)
dist(D)
dist(C)
dist(s)
dist(C)
dist(B)
dist(A)
dist(s)
dist(B)
dist(A)
dist(s)
dist(A)
dist(s)
dist(s)
dist(s)
dist(A)
dist(s)
dist(s)

Simplify by Tabulating

Instead of computing these values recursively, compute them one at a time, recording them. Then in the future, you only need to do table lookups.

Dynamic Programming

Our final general algorithmic technique:

1. Break problem into smaller subproblems.
2. Find recursive formula solving one subproblem in terms of simpler ones.
3. Tabulate answers and solve all subproblems.

Question: Dynamic Program

Which of the following algorithms that we have covered so far involves a dynamic program?

A) Bellman-Ford
B) Optimal Caching
C) Computing SCCs
D) Closest Pair of Points
E) Karatsuba Multiplication

$$\mathrm{dist}_k(w) = \min_{(v,w)\in E} \mathrm{dist}_{k-1}(v) + \ell(v,w).$$

Subsequences

Given a sequence, say `ABCBA`, a subsequence is the sequence obtained by deleting some letters and leaving the rest in the same order.

For example, `ABCBA` would have a subsequence `ABCBA` = `ACB`.

Longest Common Subsequence

We say that a sequence is a <u>common subsequence</u> of two others, if it is a subsequence of both.

For example `ABC` is a common subsequence of `ADBCA` and `AABBC`.

<u>**Problem:**</u> Given two sequences compute the <u>longest common subsequence</u>. That is the subsequence with as many letters as possible.

Question: LCSS

What is the length of the longest common subsequence of `ABCBA` and `ABACA`?

A) 1

B) 2

C) 3

D) 4

E) 5

$$\mathtt{ABCA} \;=\; \mathtt{\underline{ABC}B\underline{A}} \;=\; \mathtt{\underline{AB}A\underline{CA}}$$

Case Analysis

How do we compute LCSS($A_1A_2 \ldots A_n$, $B_1B_2 \ldots B_m$)?

Consider cases for the common subsequence:

1. It does not use A_n.
2. It does not use B_m.
3. It uses both A_n and B_m and these characters are the same.

Case 1

If the common subsequence does not use A_n, it is actually a common subsequence of

$A_1A_2\ldots A_{n-1}$, and $B_1B_2\ldots B_m$

Therefore, in this case, the longest common subsequence would be LCSS($A_1A_2\ldots A_{n-1}$, $B_1B_2\ldots B_m$).

Case 2

If the common subsequence does not use $\mathtt{B}_{\mathtt{m}}$, it is actually a common subsequence of

$\mathtt{A}_1\mathtt{A}_2\ldots\mathtt{A}_{\mathtt{n}}$, and $\mathtt{B}_1\mathtt{B}_2\ldots\mathtt{B}_{\mathtt{m-1}}$

Therefore, in this case, the longest common subsequence would be LCSS($\mathtt{A}_1\mathtt{A}_2\ldots\mathtt{A}_{\mathtt{n}}$, $\mathtt{B}_1\mathtt{B}_2\ldots\mathtt{B}_{\mathtt{m-1}}$).

Case 3

If a common subsequence uses both A_n and B_m…

- These characters must be the same.
- Such a subsequence is obtained by taking a common subsequence of:
  $A_1A_2\ldots A_{n-1}$, and $B_1B_2\ldots B_{m-1}$
  and adding a copy of $A_n = B_m$ to the end.
- The longest length of such a subsequence is
  LCSS($A_1A_2\ldots A_{n-1}$, $B_1B_2\ldots B_{m-1}$)+1.

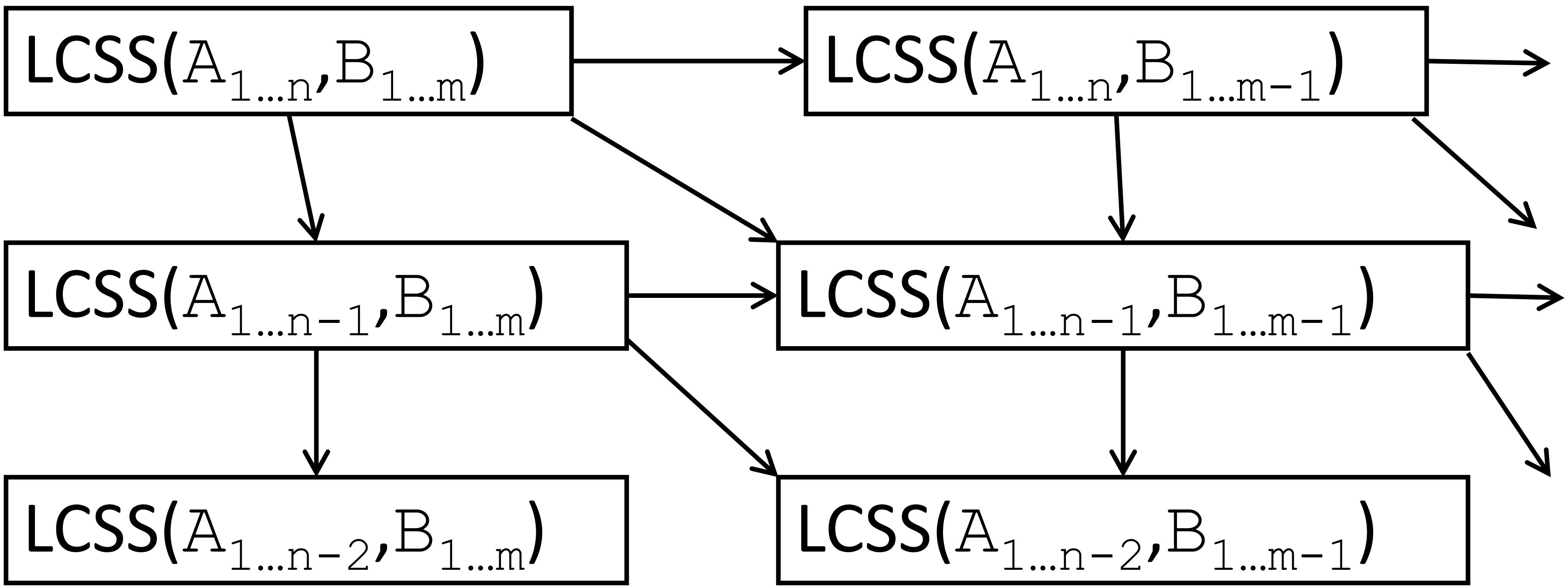
Recursion

On the other hand, the longest common subsequence must come from one of these cases. In particular, it will always be the one that gives the biggest result.

$$\mathrm{LCSS}(A_1A_2\ldots A_n, B_1B_2\ldots B_m) =$$
$$\mathrm{Max}(\mathrm{LCSS}(A_1A_2\ldots A_{n-1}, B_1B_2\ldots B_m),$$
$$\mathrm{LCSS}(A_1A_2\ldots A_n, B_1B_2\ldots B_{m-1}),$$
$$[\mathrm{LCSS}(A_1A_2\ldots A_{n-1}, B_1B_2\ldots B_{m-1})+1])$$

[where the last option is only allowed if $A_n = B_m$]

Recursion



Key Point: Subproblem reuse

Only ever see LCSS($A_1A_2…A_k$, $B_1B_2…B_\ell$)

Base Case

Our recursion also needs a base case.

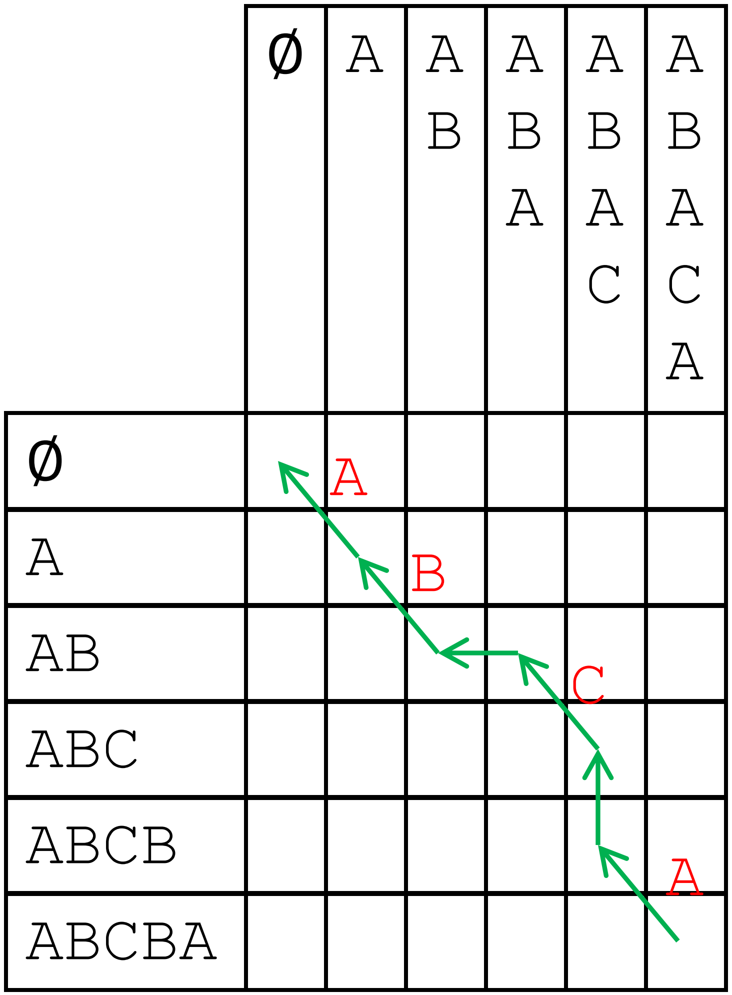
In this case we have:

LCSS(Ø,$B_1B_2 \ldots B_m$) = LCSS($A_1A_2 \ldots A_n$, Ø) = 0.

Algorithm

```
LCSS(A_1A_2…A_n,B_1B_2…B_m)
  Initialize Array T[0…n,0…m]
    \\ T[i,j] will store LCSS(A_1A_2…A_i,B_1B_2…B_j)
  For i = 0 to n
    For j = 0 to m                O(nm) iterations
      If (i = 0) OR (j = 0)
        T[i,j] ← 0
O(1)  Else If A_i = B_j
        T[i,j] ← max(T[i-1,j],T[i,j-1],T[i-1,j-1]+1)
      Else
        T[i,j] ← max(T[i-1,j],T[i,j-1])
  Return T[n,m]
```

Example



String:

ABCA

Proof of Correctness

Prove by induction that each value assigned to T[i,j] is the correct value for LCSS($A_1A_2 \ldots A_i, B_1B_2 \ldots B_j$).

Base Case: When i or j is 0 we assign 0.

Inductive Step: Assuming that previous values are assigned correctly, T[i,j] gets correct value because of recursion for LCSS and inductive hypothesis (and that we have previously filled in T[i-1,j], T[i,j-1] and T[i-1,j-1]).

Notes about DP

- General Correct Proof Outline:
  - Prove by induction that each table entry is filled out correctly
  - Use base-case and recursion
- Runtime of DP:
  - Usually
    [Number of subproblems]x[Time per subproblem]

More Notes about DP

- Finding Recursion
  - Often look at first or last choice and see what things look like without that choice
- Key point: Picking right subproblem
  - Enough information stored to allow recursion
  - Not too many