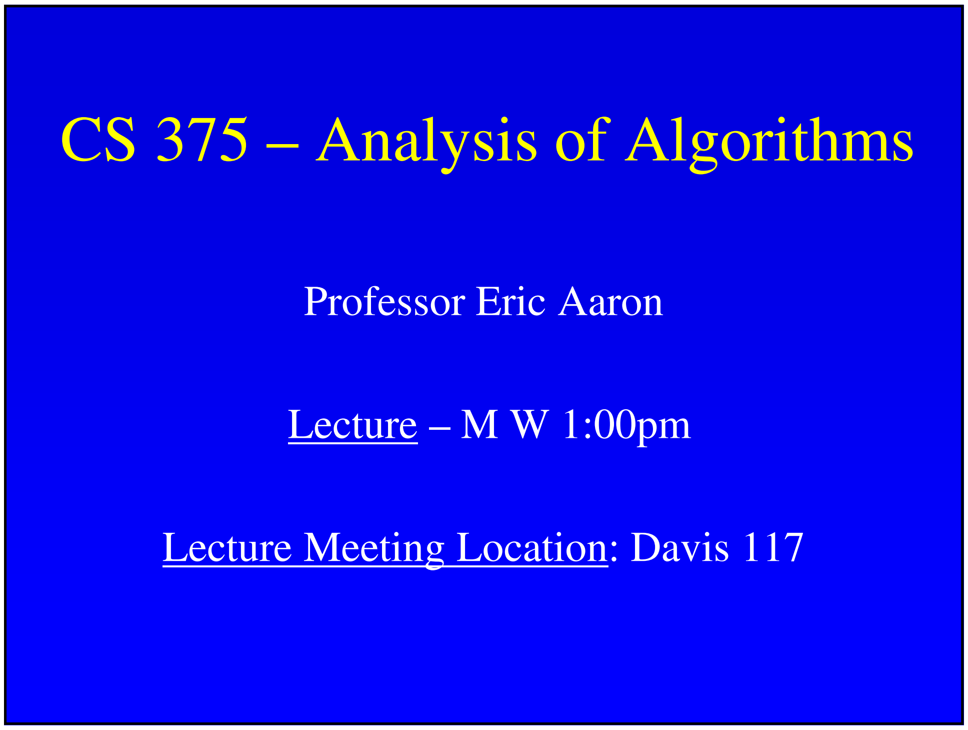
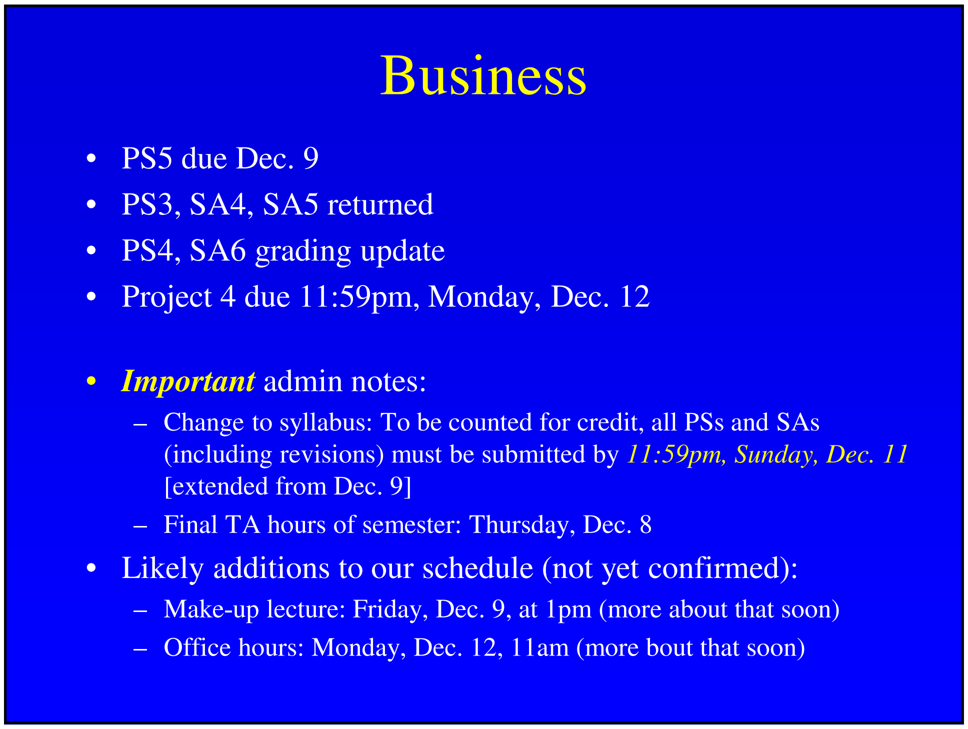


CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

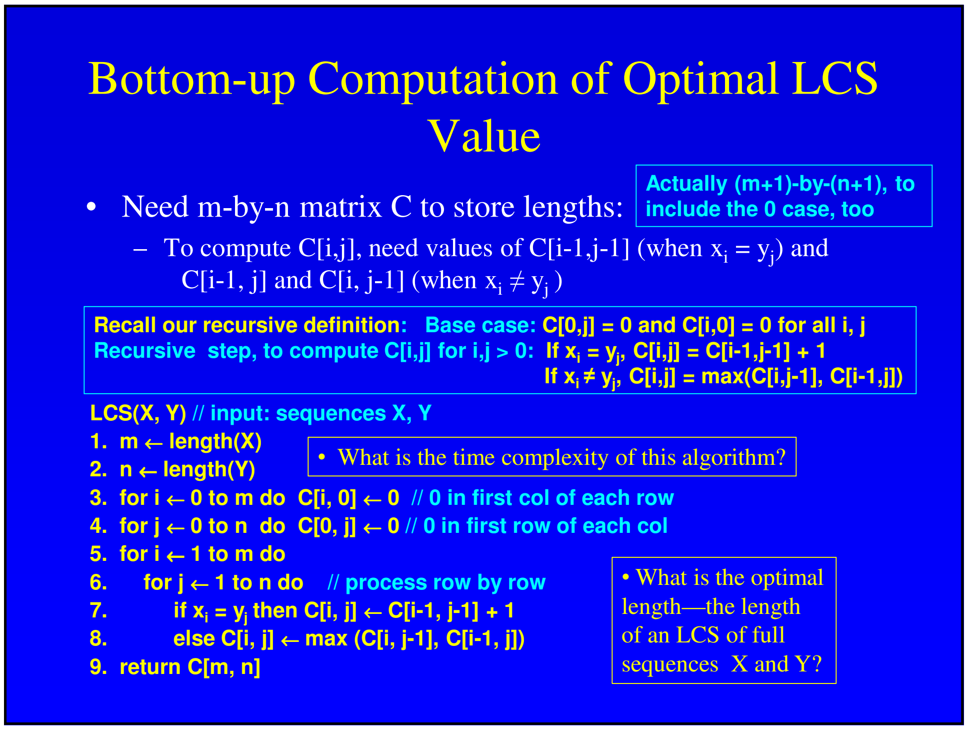
Lecture Meeting Location: Davis 117



Business

- PS5 due Dec. 9
- PS3, SA4, SA5 returned
- PS4, SA6 grading update
- Project 4 due 11:59pm, Monday, Dec. 12

- ***Important*** admin notes:
  - Change to syllabus: To be counted for credit, all PSs and SAs (including revisions) must be submitted by *11:59pm, Sunday, Dec. 11* [extended from Dec. 9]
  - Final TA hours of semester: Thursday, Dec. 8
- Likely additions to our schedule (not yet confirmed):
  - Make-up lecture: Friday, Dec. 9, at 1pm (more about that soon)
  - Office hours: Monday, Dec. 12, 11am (more bout that soon)



Bottom-up Computation of Optimal LCS Value

- Need m-by-n matrix C to stores lengths:
  - To compute C[i,j], need values of C[i-1,j-1] (when $x_i = y_j$) and C[i-1, j] and C[i, j-1] (when $x_i \neq y_j$)

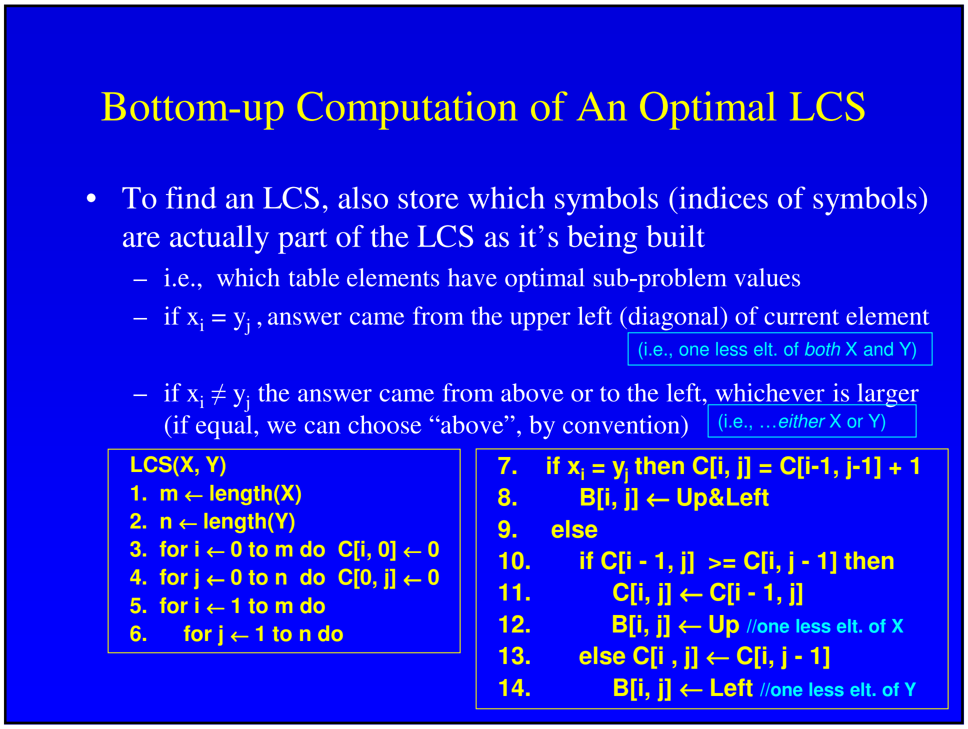
Actually (m+1)-by-(n+1), to include the 0 case, too

Recall our recursive definition: Base case: C[0,j] = 0 and C[i,0] = 0 for all i, j
Recursive step, to compute C[i,j] for i,j > 0: If $x_i = y_j$, C[i,j] = C[i-1,j-1] + 1
If $x_i \neq y_j$, C[i,j] = max(C[i,j-1], C[i-1,j])

```
LCS(X, Y) // input: sequences X, Y
1. m ← length(X)
2. n ← length(Y)
3. for i ← 0 to m do  C[i, 0] ← 0  // 0 in first col of each row
4. for j ← 0 to n  do  C[0, j] ← 0 // 0 in first row of each col
5. for i ← 1 to m do
6.     for j ← 1 to n do    // process row by row
7.         if $x_i = y_j$ then C[i, j] ← C[i-1, j-1] + 1
8.         else C[i, j] ← max (C[i, j-1], C[i-1, j])
9. return C[m, n]
```

- What is the time complexity of this algorithm?

- What is the optimal length—the length of an LCS of full sequences X and Y?



Bottom-up Computation of An Optimal LCS

- To find an LCS, also store which symbols (indices of symbols) are actually part of the LCS as it's being built
  - i.e., which table elements have optimal sub-problem values
  - if $x_i = y_j$, answer came from the upper left (diagonal) of current element

    (i.e., one less elt. of *both* X and Y)
  - if $x_i \neq y_j$ the answer came from above or to the left, whichever is larger (if equal, we can choose "above", by convention)

    (i.e., …*either* X or Y)

```
LCS(X, Y)
1. m ← length(X)
2. n ← length(Y)
3. for i ← 0 to m do  C[i, 0] ← 0
4. for j ← 0 to n  do  C[0, j] ← 0
5. for i ← 1 to m do
6.     for j ← 1 to n do
7.         if $x_i = y_j$ then C[i, j] = C[i-1, j-1] + 1
8.             B[i, j] ← Up&Left
9.         else
10.            if C[i - 1, j]  >= C[i, j - 1] then
11.                C[i, j] ← C[i - 1, j]
12.                B[i, j] ← Up //one less elt. of X
13.            else C[i , j] ← C[i, j - 1]
14.                B[i, j] ← Left //one less elt. of Y
```

- LCS example:

LCS of

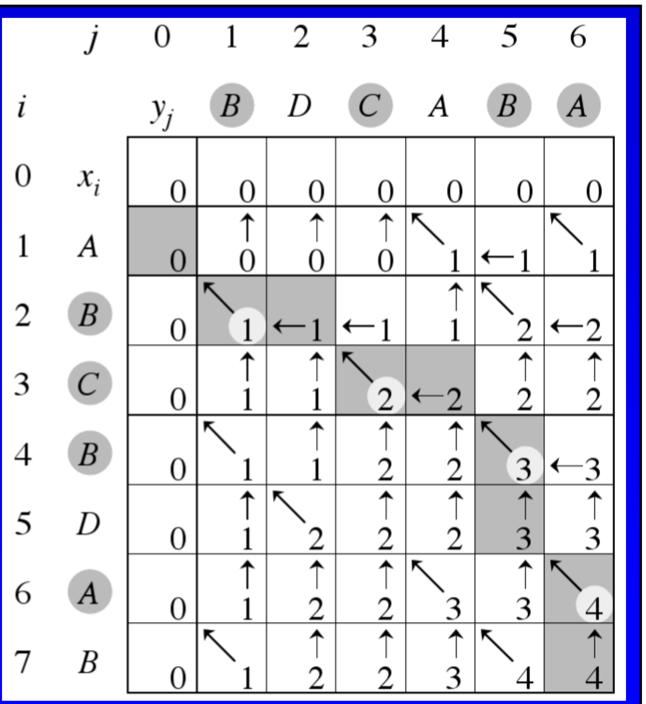
X = <A,B,C,B,D,A,B>

and

Y = <B,D,C,A,B,A>

The algorithm finds an LCS: *BCBA*

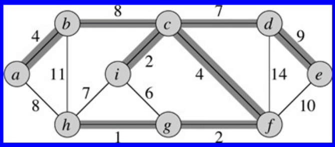
Are there others?



	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

A few kinds of graph problems

- Graph representations are very useful, and broadly applicable!
- Examples: Robot *navigation* / *inspection* problems
  - *Patrol*: Team of robots repeatedly inspects all critical points on a map (e.g., locations of valuable items), to check for intruders or other problems
  - *Boundary coverage*: Team of robots completely inspects the boundaries of all items (in a 2D-mapped environment)
  - *Pursuit-evasion*: Team of robots corners / captures intruders that are trying to avoid capture
  - *Map visitation*: Team of robots visits every critical point on a map
- Typically, a *map* underlies these problems / solutions, and that map is represented by a graph



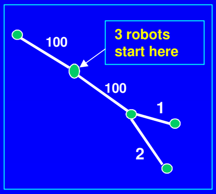
Map Visitation Problem (MVP)

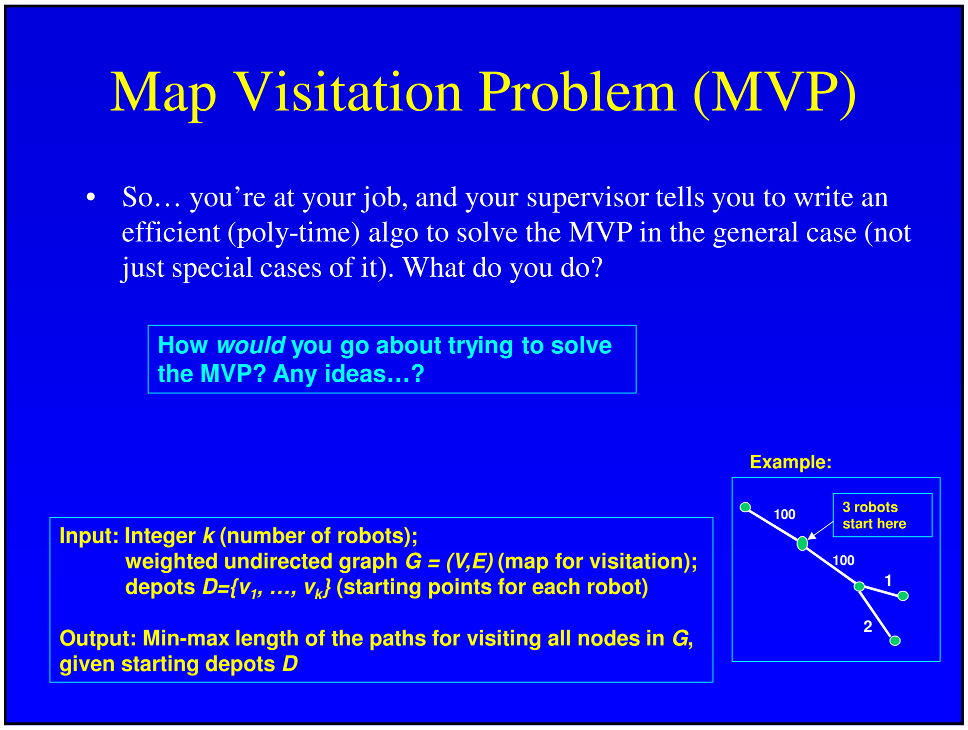
- There are multiple variations on the MVP, but we'll look at the *multi-robot*, *multi-depot* MVP
  - Optimization problem: Fastest way for a team of robots, starting from a collection of depot locations on a map, to visit all nodes on that map
  - MVP looks at this as: How do we make sure that the last node visited is visited as soon as possible—to get information about every node as soon as possible?

Note: This isn't the same as, say, minimizing total distance traveled by all robots combined—it's a different metric for optimization

Properties of map / robots for MVP:
1. **Each robot starts at a specified depot**
2. **Navigation on the map is pre-computed, not figured out by the robot as it moves**
3. **A solution to MVP means at least one robot visits each point on the map**

Example:



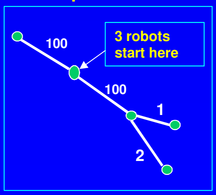


Map Visitation Problem (MVP)

- So… you're at your job, and your supervisor tells you to write an efficient (poly-time) algo to solve the MVP in the general case (not just special cases of it). What do you do?

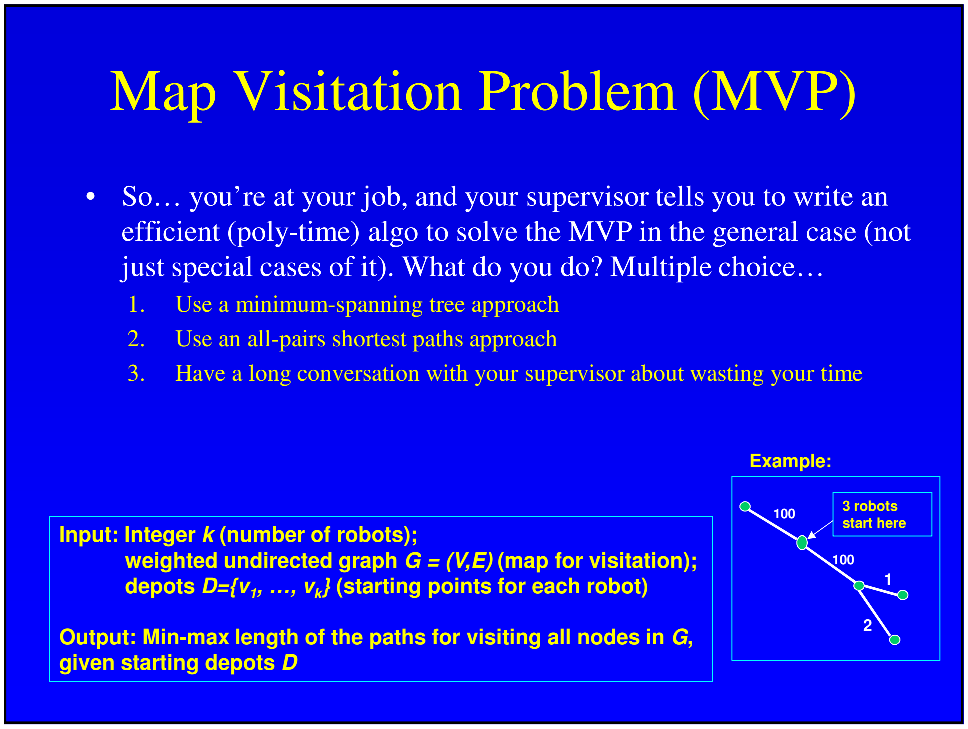
How *would* you go about trying to solve the MVP? Any ideas…?

Example:



Input: Integer *k* (number of robots);
weighted undirected graph *G = (V,E)* (map for visitation);
depots $D=\{v_1, \ldots, v_k\}$ (starting points for each robot)

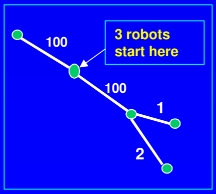
Output: Min-max length of the paths for visiting all nodes in *G*, given starting depots *D*



Map Visitation Problem (MVP)

- So… you're at your job, and your supervisor tells you to write an efficient (poly-time) algo to solve the MVP in the general case (not just special cases of it). What do you do? Multiple choice…
  1. Use a minimum-spanning tree approach
  2. Use an all-pairs shortest paths approach
  3. Have a long conversation with your supervisor about wasting your time

Example:



Input: Integer *k* (number of robots);
weighted undirected graph *G = (V,E)* (map for visitation);
depots $D=\{v_1, \ldots, v_k\}$ (starting points for each robot)

Output: Min-max length of the paths for visiting all nodes in *G*, given starting depots *D*

Map Visitation Problem (MVP)

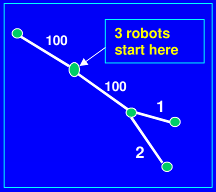
- So… you're at your job, and your supervisor tells you to write an efficient (poly-time) algo to solve the MVP in the general case (not just special cases of it). What do you do? Multiple choice…
  1. Use a minimum-spanning tree approach
  2. Use an all-pairs shortest paths approach
  3. Have a long conversation with your supervisor about ~~wasting~~ better uses of your time

Options 1 and 2 won't work! But before you do option 3, you should *know* you're right…

Input: Integer *k* (number of robots);
weighted undirected graph *G = (V,E)* (map for visitation);
depots $D=\{v_1, \ldots, v_k\}$ (starting points for each robot)

Output: Min-max length of the paths for visiting all nodes in *G*, given starting depots *D*

Example:



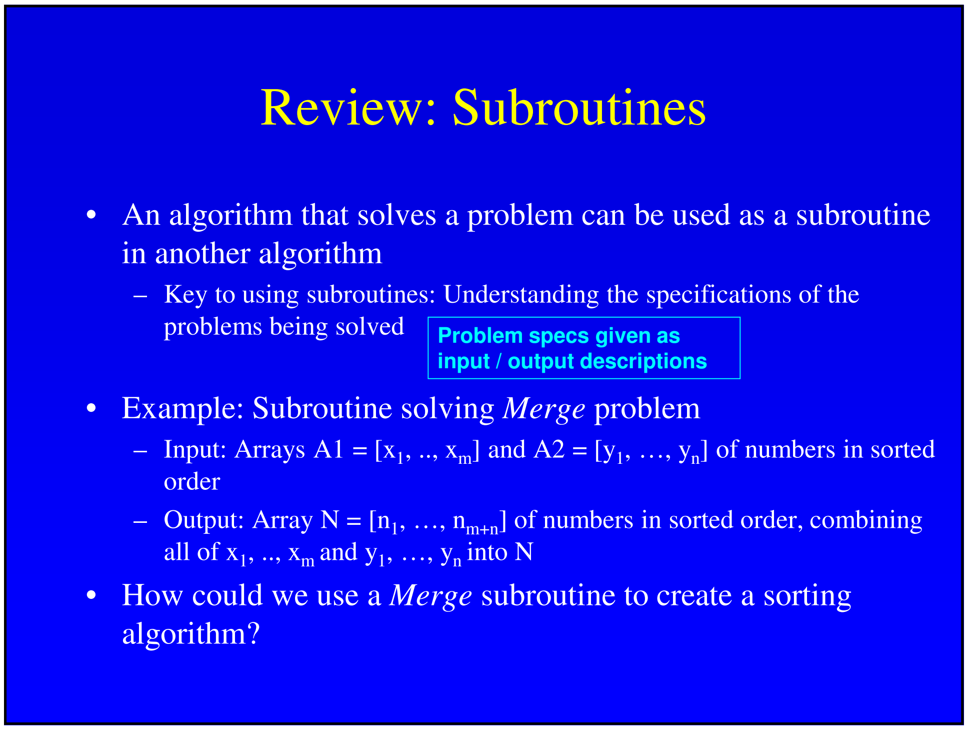
Headlines!

Coming Up on CS375…

- *What's the fastest sorting algorithm?*
  - Is it quicksort?
  - The answer might surprise you! (Or it might not!)
- *All the ~~hottest fashion trends for programmers~~ most important complexity classes for programmers!*
  - With essential algos from each one!
- *Common myths about recursion, debunked!*
  - (Some of the answers here *actually* might surprise you!)
- *How to make an algorithm not just a little faster, but a lot faster!*
- *Could **your next problem** be **NP-Complete**? And how would you know?*

Have you heard of NP-Completeness before? What do you know about it?

- Plus… *language* / *NLP applications*, a little peek into *what a compiler / interpreter does*, and *more*!

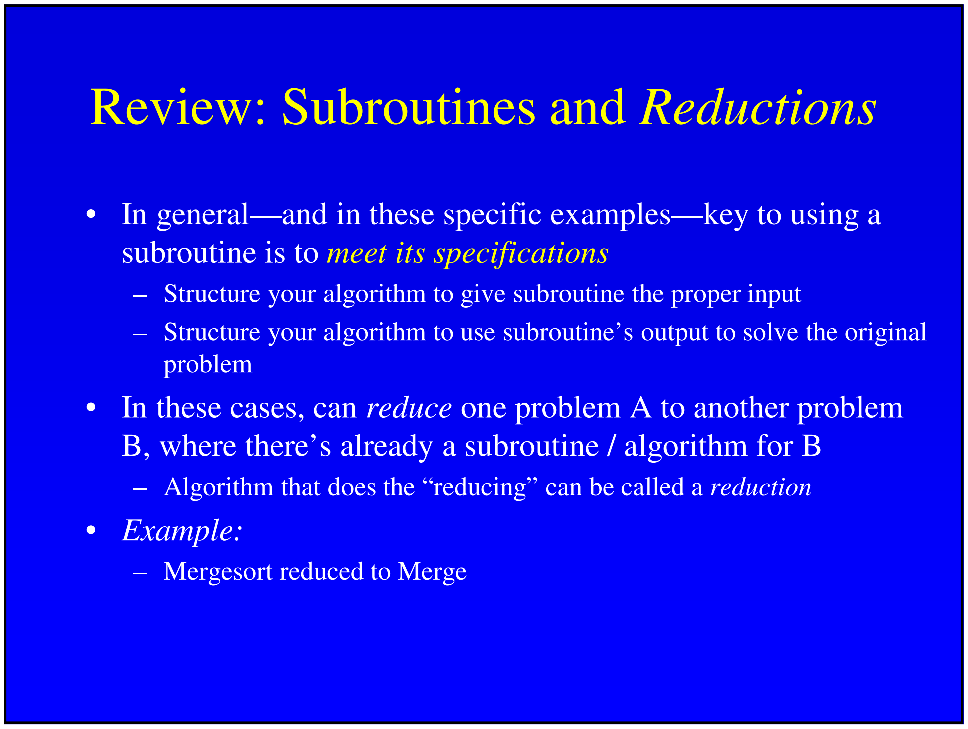


Review: Subroutines

- An algorithm that solves a problem can be used as a subroutine in another algorithm
  - Key to using subroutines: Understanding the specifications of the problems being solved

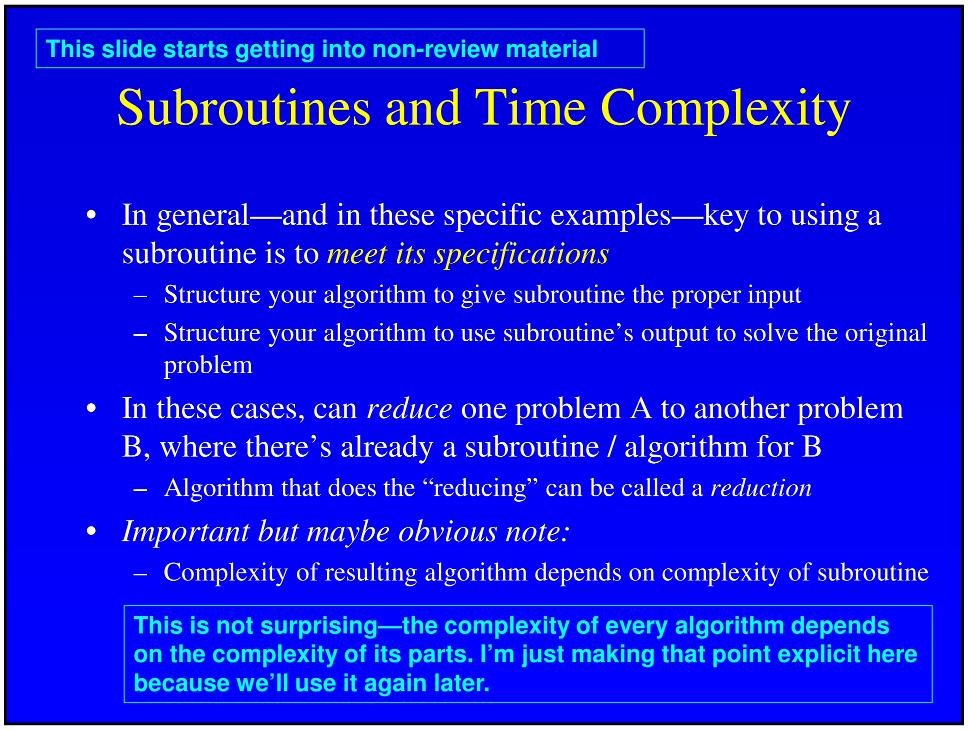
Problem specs given as input / output descriptions

- Example: Subroutine solving *Merge* problem
  - Input: Arrays A1 = [x_1, .., x_m] and A2 = [y_1, …, y_n] of numbers in sorted order
  - Output: Array N = [n_1, …, n_{m+n}] of numbers in sorted order, combining all of x_1, .., x_m and y_1, …, y_n into N
- How could we use a *Merge* subroutine to create a sorting algorithm?



Review: Subroutines and *Reductions*

- In general—and in these specific examples—key to using a subroutine is to *meet its specifications*
  - Structure your algorithm to give subroutine the proper input
  - Structure your algorithm to use subroutine's output to solve the original problem
- In these cases, can *reduce* one problem A to another problem B, where there's already a subroutine / algorithm for B
  - Algorithm that does the "reducing" can be called a *reduction*
- *Example:*
  - Mergesort reduced to Merge



This slide starts getting into non-review material

Subroutines and Time Complexity

- In general—and in these specific examples—key to using a subroutine is to *meet its specifications*
  - Structure your algorithm to give subroutine the proper input
  - Structure your algorithm to use subroutine's output to solve the original problem
- In these cases, can *reduce* one problem A to another problem B, where there's already a subroutine / algorithm for B
  - Algorithm that does the "reducing" can be called a *reduction*
- *Important but maybe obvious note:*
  - Complexity of resulting algorithm depends on complexity of subroutine

This is not surprising—the complexity of every algorithm depends on the complexity of its parts. I'm just making that point explicit here because we'll use it again later.