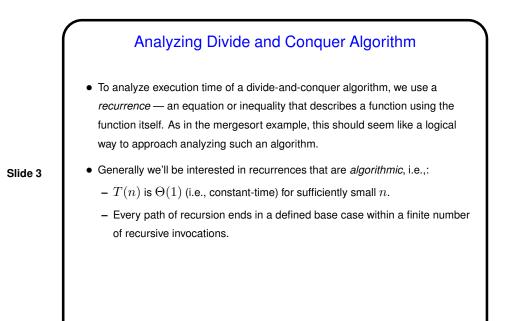


**Divide and Conquer Revisited** • "Divide and conquer", in context, is a strategy for problem-solving that consists of three steps: • Split the problem into one or more smaller problems that are instances of the same problem, unless it's so small and simple that it can be solved directly (in which case it's called a *base case*). • Solve the subproblems by solving them recursively. • *Merge* the solutions to the subproblems into a solution to the whole problem. • Note that of course this can only work if this recursive splitting eventually always reaches a base case.



## Solving Recurrences • Several ways to solve recurrences: • Substitution method — guesses a solution and proves that it is correct via induction. • Recursion-tree method — models the recurrence as a tree, as the textbook does for MERGE-SORT. • Master method — applies to recurrences of the form T(n) = aT(n/b) + f(n)by applying a "master theorem". • Akra-Bazzi method — a general method involving calculus.

Slide 4

 $\mathbf{2}$ 

### Examples — Multiplying Square Matrices

• Matrix multiplication occurs frequently in numerical work. You may remember how this works, and if not, review summary in Appendix D. Given two N by N matrices A and B, their product is also N by N, and its elements are defined as:

Slide 5

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$$

(I find it useful to model this as a dot product of the i-th row of A and the j-th column of B.)

Generally we'll look at *dense* matrices, in which most elements are nonzero. (In *sparse* matrices, most elements are zero; such matrices can be more compactly represented.)

### **Block-Based Matrix Multiplication**

• We'll first observe that another way to compute the product  $C = A \cdot B$  is by first partitioning each of the matrices into submatrices ("blocks", also identified by two indices, e.g.,  $C_{11}$  for the top left block) and applying the basic idea of the algorithm — sum of products — to these blocks:

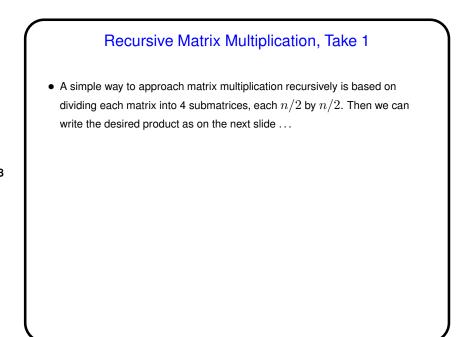
$$C_{ij} = \sum_{k=1}^{N} A_{ik} \cdot B_{kj}$$

- (Showing that this works is not difficult basic algebra though tedious.)
- To put this to use, it's helpful to define and implement an algorithm to compute

$$C = C + A \cdot B$$

# • As noted, it's more generally useful to define an algorithm for computing $C = C + A \cdot B$ (where A, B, and C are n by n dense matrices). The naive algorithm is just: MATRIX-MULTIPLY(A, B, C, n)for i = 1 to nfor j = 1 to nfor k = 1 to n $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ It should be fairly clearly that this algorithm's running time is $\Theta(n^3)$ .

Slide 7



Precursive Matrix Multiplication, Take 1, Continued • In matrix notation:  $\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$ • In equation form, one for each submatrix of C:  $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$ 

 $C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$  $C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$ 

Slide 9

| ļ | An algorithm based on this approach (assuming $C$ is initialized to 0): |
|---|---|
| I | MATRIX-MULTIPLY-RECURSIVE $(A,B,C,n)$                                   |
| i | if $n == 1$   |
|   | $c_{11}+=a_{11}\cdot b_{11}$ // Base case                               |
|   | return  |
| I | Matrix-Multiply-Recursive $(A_{11},B_{11},C_{11},n/2)$                  |
| I | MATRIX-MULTIPLY-RECURSIVE $(A_{11},B_{12},C_{12},n/2)$                  |
| I | Matrix-Multiply-Recursive $(A_{21},B_{11},C_{21},n/2)$                  |
| I | Matrix-Multiply-Recursive $(A_{21},B_{12},C_{22},n/2)$                  |
| I | Matrix-Multiply-Recursive $(A_{12},B_{21},C_{11},n/2)$                  |
| I | Matrix-Multiply-Recursive $(A_{12},B_{22},C_{12},n/2)$                  |
| I | Matrix-Multiply-Recursive $(A_{22},B_{21},C_{21},n/2)$                  |
| I | Matrix-Multiply-Recursive $(A_{22},B_{22},C_{22},n/2)$                  |

# Recursive Matrix Multiplication, Take 1, Implementation Details

• It's worth noting that in order to make access to the submatrices work we have to either copy the submatrices to and from temporary storage, incurring a runtime cost of  $\Theta(n^2)$ , or manipulate indices appropriately, which takes constant time. The latter is more efficient and practical, though the details are a little tedious to work out.

Slide 11

#### Sidebar: Block-Based Matrix Multiplication, Revisited

- Total tangent, but I think maybe interesting:
- Matrix multiplication turns out to be an operation that lends itself to "parallelization" (solving in a way that uses multiple threads or processes at the same time), and often block-based algorithms work well for such implementation.
- So ... I teach an elective (CSCI 3366) in parallel computing, and I use matrix multiplication as an example. I used to do research on design patterns for parallel computing, and one of my long-term research collaborators claimed that block-based matrix multiplication was faster even without multiple processes or threads, because it "made better use of the memory hierarchy" (cache etc.). I was skeptical, but when I wrote code for it ... (Short demo. Note that code is available on course Web site under "sample programs".)

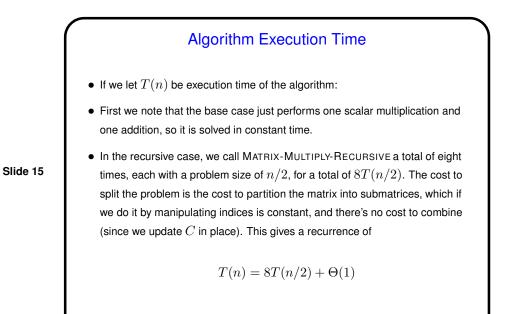
Slide 13

### Sidebar, Continued

- Sample code accesses submatrices using the method of index calculation, based on a slight extension to how straight C represents multidimensional arrays:
- Straight C represents an *m* by *n* 2D array A in "row-major" order, i.e., as a 1D array AA of size *m* · *n* consisting of row 0 of A followed by row 1, then row 2, etc. A [i] [j] is then accessed as [(i\*n)+j]. (A diagram helps here!) How then to access elements of submatrices? well, each row of a submatrix is contiguous in memory, just as a full row is, but while the rows of the full matrix are laid out one after another, the rows of a submatrix aren't, but are spaced out at regular intervals, based on the dimensions of the full matrix. An index relative to a submatrix is not too hard to compute using the dimensions of the full matrix rows in the full 1D array. Here too a diagram is helpful, and see example code.

#### Algorithm Correctness

- The base case is okay in this case each of the submatrices is a single element.
- It might not be 100% obvious why this algorithm correctly translates the block-based equations for the multiplication, but: Note that each of the equations does two multiplications and adds the result, for a total of 8 calculations of the form  $C_{ij} + = A_{ik} \cdot B_{kj}$ . (This also explains why it's useful to have as a basic operation  $C = C + A \cdot B$ .)



### Algorithm Execution Time, Continued

• The textbook uses the "master method" (more later) to get an order of growth of  $\Theta(n^3)$  for this recurrence. It's at least a bit reassuring that this is the same as the order of growth of the naive algorithm, since this does seem like it's doing the same basic operations, just in a different order, no?

