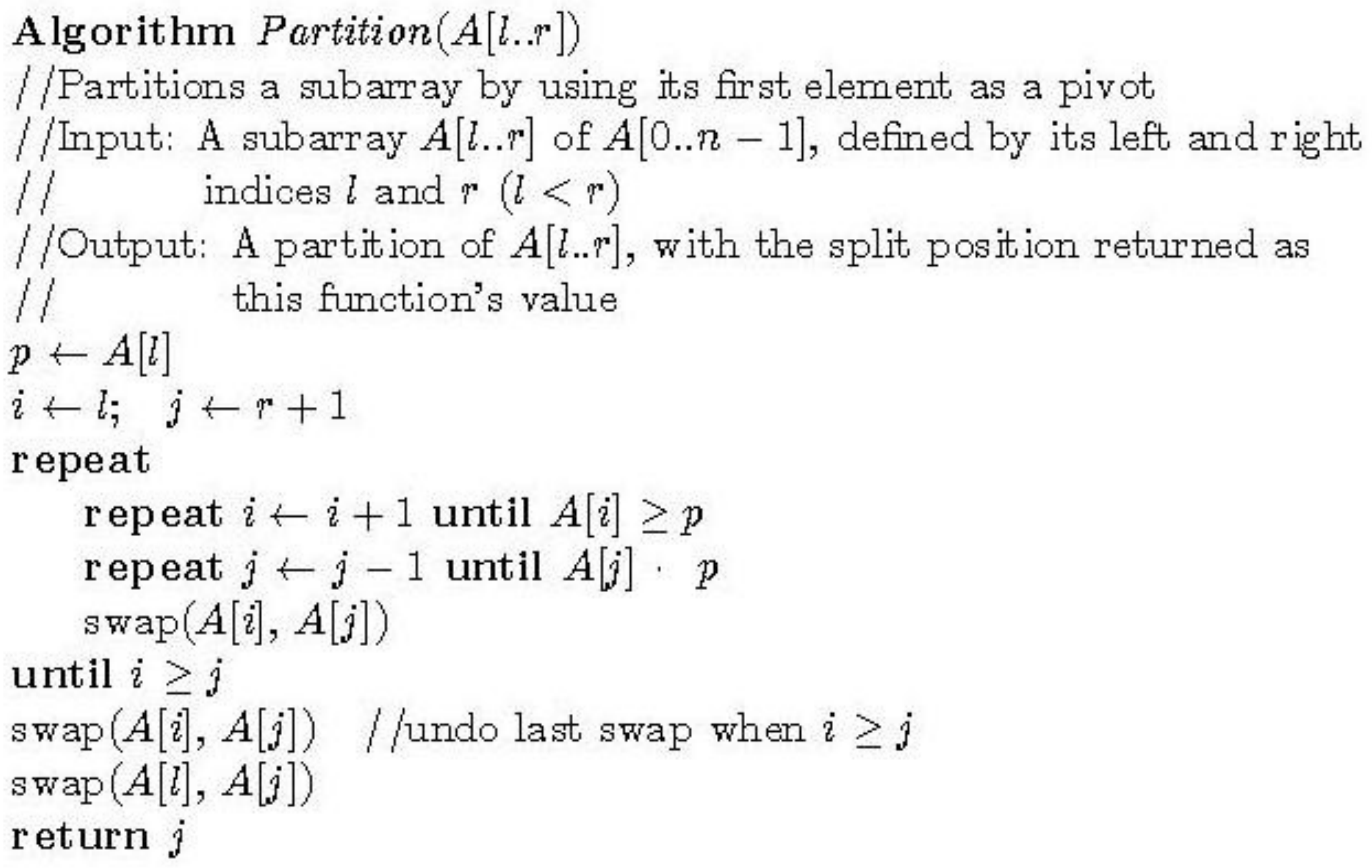
Divide and Conquer (II)

CSC 411

Richard Kelley

Partitioning Algorithm



```
Algorithm Partition($A[l..r]$)
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
//          indices $l$ and $r$ $(l < r)$
//Output: A partition of $A[l..r]$, with the split position returned as
//           this function's value
$p \leftarrow A[l]$
$i \leftarrow l$;   $j \leftarrow r + 1$
repeat
    repeat $i \leftarrow i + 1$ until $A[i] \geq p$
    repeat $j \leftarrow j - 1$ until $A[j] \cdot p$
    swap($A[i], A[j]$)
until $i \geq j$
swap($A[i], A[j]$)   //undo last swap when $i \geq j$
swap($A[l], A[j]$)
return $j$
```

Procedure for Analysis

1. Define a variable (or *variables*) for **input size**.
   - This is frequently something like *n*.
2. Choose a notation for the time complexity.
   - This is frequently something like *T*.
   - It will be a function of the input size, so *T*(*n*).
3. Write down an expression for *T*(*n*) in terms of *n*.
   - An explicit formula if you can, or
   - A recurrence.
4. Simplify the formula as much as possible.
   - Using algebra or the Master Theorem.
5. Convert to big-oh notation to summarize the simplified result.

Master Theorem

Let T be an increasing function that satisfies the recurrence relation:

$$T(n) = a\,T(n/b) + cn^d$$

whenever $n = b^k$, where k is a positive integer, a >= 1, b is an integer greater than 1, c is a positive real number, and d is a non-negative real number. Then:

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \quad \text{case 1} \\ O(n^d \log_b n) & \text{if } a = b^d \quad \text{case 2} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{case 3} \end{cases}$$

a >= 1: The number of subproblems

b > 1: Amount by which problems shrink

cn^d: Amount of nonrecursive work at each level of recursion.

The cases are (effectively) comparing cn^d with n^(log_b(a))

Analysis of Mergesort

- All cases have same efficiency: $\Theta(n \log n)$

- Why? What is its growth function?
  - $T(n) = 2\,T(n/2) + \boxed{f(n)}$ =? The time needed for merging
  - $T_{worst}(n) = 2\,T_{worst}(n/2) + n\text{-}1$ *for* $n>1$, $T(1)=0$
  - *Using* **Master Theorem**

1. If $a < b^d$, $T(n) \in \Theta(n^d)$
2. If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
3. If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

$a = 2$, $b=2$, $d=1$

=> Case 2, $T(n) \in \Theta(n \log n)$

Analysis of Quicksort

- Best case: split in the middle $\Theta(n \log n)$

- Why?
  - $T_{best}(n) = 2\,T_{best}(n/2) + \bigcirc\!\!\!\!f(n)$ =? The time needed for partition
  - $T_{best}(n) = 2\,T_{best}(n/2) + n-1$ *for $n>1$, $T(1)=0$*
  - *Using* **Master Theorem**

1. If $a < b^d$, $T(n) \in \Theta(n^d)$
2. If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
3. If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

$a = 2$, $b=2$, $d=1$

=> Case 2, $T(n) \in \Theta(n \log n)$

Analysis of Quicksort

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$
- Average case: random arrays — $\Theta(n \log n)$

Karatsuba Multiplication

What is the complexity of arithmetic?

- Think in terms of single-digit operations.
- How many one-digit additions are involved in adding two *n*-digit numbers?
- What about subtraction?
- What about multiplication?

Complexity of multiplication

Consider the problem of multiplying two (large) *n*-digit integers represented by arrays of their digits such as:

A = 12345678901357986429 B = 87654321284820912836

The grade-school algorithm:

$$\begin{array}{rrrrrrr}
 & & & a_1 & a_2 \ldots & a_n \\
 & & & b_1 & b_2 \ldots & b_n \\
 & (d_{10}) & d_{11} & d_{12} & \ldots & d_{1n} \\
(d_{20}) & d_{21} & d_{22} \ldots & & d_{2n} & \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots \ \ldots \\
\end{array}$$

$(d_{n0})\ d_{n1} d_{n2} \ldots d_{nn}$

Efficiency: n^2 one-digit multiplications

Let's assume that one-digit operations all have the same cost. The problem with multiplication is that each digit of one number has to interact with every digit of the other number, making the situation quadratic.

Can we multiply *n*-digit numbers using divide and conquer?

- Consider $A * B$ where $A = 2135$ and $B = 4014$
  - $A = (21 \cdot 10^2 + 35)$
  - $B = (40 \cdot 10^2 + 14)$
- $A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$
  $= (21 * 40) \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + (35 * 14) \cdot 10^0$
- We’ve made progress, and we’re left with four smaller multiplications…

Divide and Conquer for Multiplication

- Let's assume *n* is a power of two (just to make things easier, not necessary in general)
- Write each number in base B as
  - $x = x_1 \cdot B^{\wedge}\{n/2\} + x_0$
  - $y = y_1 \cdot B^{\wedge}\{n/2\} + y_0$
  - So that x_1, x_0, y_1, y_0 are each n/2-digit integers
- Then $xy = x_1y_1 \cdot B^n + (x_1y_0 + x_0y_1) \cdot B^{\wedge}\{n/2\} + x_0y_0$
- This gives us four recursive multiplications:
  - x_1y_1, x_1y_0, x_0y_1, x_0y_0, each half the size of the original input.
- Plus work to add results and shift digits.
  - This is a linear amount of work.

Divide and Conquer for Multiplication

- Four subproblems,
- Each half the size of the input,
- With a linear amount of work to combine.

- What is the recurrence for *T(n)*?
- What does the Master Theorem tell us about *T(n)*?

Divide and Conquer

- Recurrence
  - $T(n) = 4\, T(n/2) + c * n$
- Master theorem:
  - Compare a to b^d: $a = 4$, $b^d = 2^1 = 2$.
  - $4 > 2$, so case 3: $T(n) = O(n^{log\_b(a)})$, or $O(n^2)$.
- Sad face.

Back to Basics

- $A * B$ where $A = 23$ and $B = 14$
  - $A = (2 \cdot 10^1 + 3 \cdot 10^0)$
  - $B = (1 \cdot 10^1 + 4 \cdot 10^0)$
- $A * B = (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0)$
  $= (2*1) \cdot 10^2 + \underline{(2*4 + 3*1)} \cdot 10^1 + (3*4) \cdot 10^0$
- BUT
  - $2*4 + 3*1 = (2+3)*(1+4) - (2*1) - (3*4)$
- The trick is that we can ***reuse*** multiplications, at the cost of more addition and subtraction.
  - We reduce from 4 multiplications to 3. Does this make a difference?

Back to Basics

- (Re)consider $A * B$ where $A = 2135$ and $B = 4014$
  - $A = (21 \cdot 10^2 + 35)$
  - $B = (40 \cdot 10^2 + 14)$
- $A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$
  $= (21 * 40) \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + (35 * 14) \cdot 10^0$
- Can we use the same “trick” in the *n*-digit case?

Karatsuba Multiplication

- Given
  - $x = x_1 \cdot B^{n/2} + x_0$
  - $y = y_1 \cdot B^{n/2} + y_0$
- Recursively compute three products:
  - $z_2 = x_1 y_1$
  - $z_0 = x_0 y_0$
  - $z_1 = (x_1 + x_0)(y_1 + y_0)$
- Notices that $x_1 y_0 + x_0 y_1 = z_1 - z_2 - z_0$.
- So write $xy = z_2 \cdot B^n + (z_1 - z_2 - z_0) \cdot B^{n/2} + z_0$,
- Same result, but with 3 multiplications instead of 4!!

Complexity of Karatsuba Multiplication

- ***Three*** subproblems,
- Each half the size of the input problem,
- With a linear amount of extra work.

- What's the recurrence for *T(n)*?
- What does the Master Theorem tell us?

Karatsuba Complexity

- Recurrence
  - *$T(n) = 3\,T(n/2) + c * n$*
- Master Theorem
  - Compare a to b^d: 3 vs 2^1: 3 > 2.
  - 3 > 2, so still in case 3, BUT
  - *$T(n) = O(n^{log\_2(3)})$* now. What is $\log_2 3$? About 1.58.
  - So *$T(n) = O(n^{1.58})$* instead of *$O(n^2)$*, which is asymptotically great.
  - Doubling the input *less than triples the cost*, vs *quadrupling* the cost.

Where does the gain come from?

- Not from the fact that addition is faster than multiplication.
- Entirely from restructuring the the tree of computations we perform recursively.

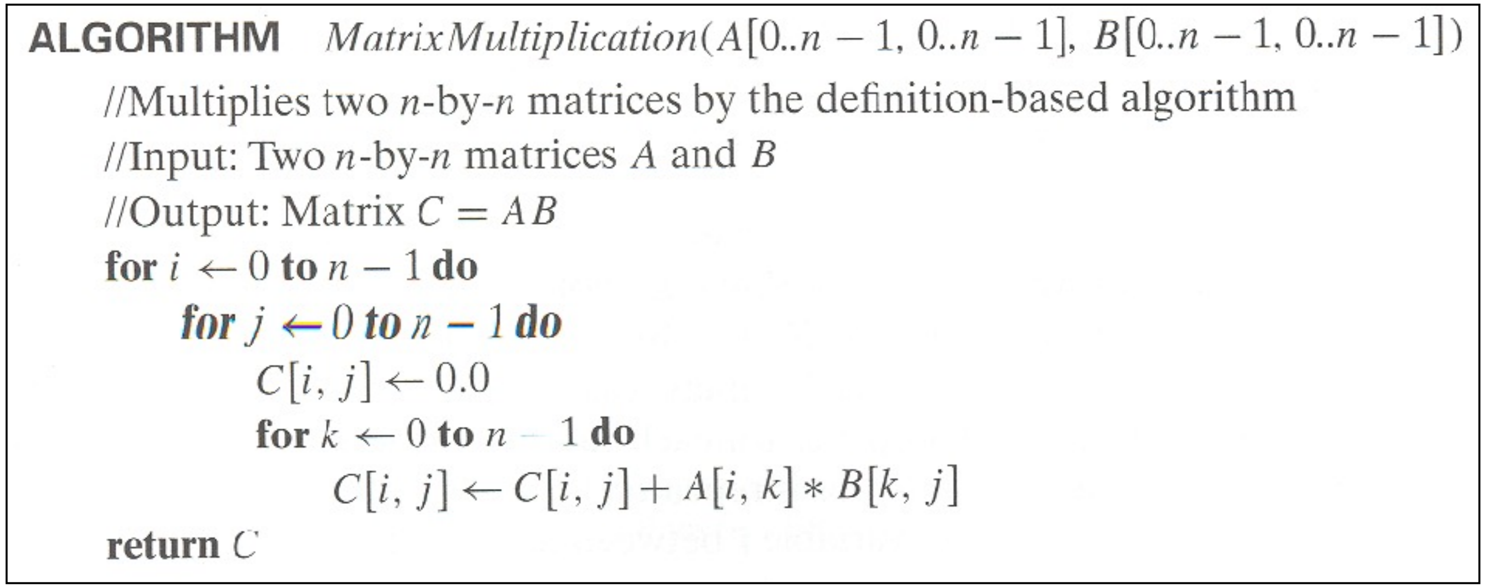
Practical Notes

- Karatsuba multiplication always works, but practically is only useful for large *n*.
- For small *n*, the “grade school” algorithm works.
- So “big integer” libraries that use Karatsuba multiplication don’t recurse all the way down – they switch to grade school at some threshold.
- If *n* is odd, then pad with leading zeros.

Strassen Multiplication

We can apply a similar idea to Matrix Multiply

- First let’s recall the standard algorithm for matrix multiplication



```
ALGORITHM  MatrixMultiplication($A[0..n-1, 0..n-1]$, $B[0..n-1, 0..n-1]$)
    //Multiplies two $n$-by-$n$ matrices by the definition-based algorithm
    //Input: Two $n$-by-$n$ matrices $A$ and $B$
    //Output: Matrix $C = AB$
    for $i \leftarrow 0$ to $n-1$ do
        for $j \leftarrow 0$ to $n-1$ do
            $C[i, j] \leftarrow 0.0$
            for $k \leftarrow 0$ to $n-1$ do
                $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
    return $C$
```

- The trick is to think about this at a different level of granularity.
- Easier to write at the board…