Lectures 7-8
More Divide and Conquer
• Multiplication
John McCarthy (1927 –2011)

• Pioneer in artificial intelligence
  – In fact, he coined the term

• Lots of work on how reasoning can be automated
  – Meta-lesson: you learn a lot about something when you teach a computer to do it.
McCarthy’s “91” function

- Why be so careful about correctness with recursive procedures?

\[ M(n) = \begin{cases} 
  n - 10, & \text{if } n > 100 \\
  M(M(n + 11)), & \text{if } n \leq 100 
\end{cases} \]

- This function is tricky to reason about because it doesn’t always make recursive calls on smaller inputs
  - Programmer’s goal: code that is easy to understand
  - Do not write circular programs!
Reminder: geometric series

\[
1 + x + x^2 + \cdots + x^n = \frac{1 - x^{n+1}}{1 - x} \quad \text{for} \quad x \neq 1
\]

\[
1 + x + x^2 + \cdots = \frac{1}{1 - x} \quad \text{for} \quad |x| < 1
\]
Review

• For each tree:
  – How many leaves?
  – How many nodes in total? (big-theta)
Multiplying large integers

- Given \( n \)-bit integers \( a, b \) (in binary), compute \( c = ab \)

- **Naive (grade-school) algorithm:**
  - Write \( a,b \) in binary
  - Compute \( n \) intermediate products
  - Do \( n \) additions

- **Running time?**
  - Total work: \( \Theta(n^2) \)

\[
\begin{array}{cccc}
  & a_{n-1} & a_{n-2} & \ldots & a_0 \\
\times & b_{n-1} & b_{n-2} & \ldots & b_0 \\
\hline
& & & & & n \text{ bits} \\
& & & & 0 & 0 \ldots 0 \\
\hline
& \cdots & \cdots & \cdots & \cdots & \cdots \\
& & & & & 2n \text{ bit output} \\
\end{array}
\]
Divide-and-conquer design

1. **Divide** the problem (instance) into subproblems.

2. **Conquer** the subproblems by solving them recursively.

3. **Combine** subproblem solutions.
Multiplying large integers

• **Divide and Conquer** (Attempt #1):
  
  – Write \( a = A_1 2^{n/2} + A_0 \)
  \( b = B_1 2^{n/2} + B_0 \)
  
  – We want \( ab = A_1 B_1 2^n + (A_1 B_0 + B_1 A_0) 2^{n/2} + A_0 B_0 \)
  
  – Multiply \( n/2 \)–bit integers recursively
  
  – \( T(n) = 4T(n/2) + \Theta(n) \)
  
  – Alas! this is still \( \Theta(n^2) \).

  (Exercise: write out the recursion tree.)
• **Divide and Conquer** (Attempt #1):  
  – Write \( a = A_1 2^{n/2} + A_0 \)  
    \[ b = B_1 2^{n/2} + B_0 \]  
  – We want \( ab = A_1B_1 2^n + (A_1B_0 + B_1A_0) 2^{n/2} + A_0B_0 \)  
  – Multiply \( n/2 \)–bit integers recursively  

– Consider the expression  
\[
(A_0+A_1) (B_0 + B_1) = A_0B_0 + A_1B_1 + (A_0B_1 + B_1A_0)
\]

– We can get away with 3 multiplications! (in yellow)  
  \[ x = A_1B_1 \quad y = A_0B_0 \quad z = (A_0+A_1)(B_0+B_1) \]  

– Now we use \( ab = A_1B_1 2^n + (A_1B_0 + B_1A_0) 2^{n/2} + A_0B_0 \)

\[
= x 2^n + (z-x-y) 2^{n/2} + y
\]
Multiplying large integers

**MULTIPLY** \((n, a, b)\)

- \(a\) and \(b\) are \(n\)-bit integers
- Assume \(n\) is a power of 2 for simplicity

1. **If** \(n \leq 2\) **then** use grade-school algorithm **else**
2. \(A_1 \leftarrow a \div 2^{n/2} ; B_1 \leftarrow b \div 2^{n/2} ;\)
3. \(A_0 \leftarrow a \mod 2^{n/2} ; B_0 \leftarrow b \mod 2^{n/2} .\)
4. \(x \leftarrow \text{MULTIPLY} (n/2 , A_1 , B_1)\)
5. \(y \leftarrow \text{MULTIPLY} (n/2 , A_0 , B_0)\)
6. \(z \leftarrow \text{MULTIPLY} (n/2 , A_1+A_0 , B_1+B_0)\)
7. **Output** \(x \cdot 2^n + (z-x-y)2^{n/2} + y\)
Multiplying large integers

• The resulting recurrence
  \[ T(n) = 3T(n/2) + \Theta(n) \]

• We will see that \[ T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.59\ldots}) \]

• Note: There is a \( \Theta(n \log n) \) algorithm for multiplication (assumes constant-time word ops)
  – Actually, \( \Theta(n \log(n) \ 2^{\log^*(n)}) \) bit-level operations (due to Martin Fürer, 2007)
Reminder: recursion trees

- Technique for guessing solution to recurrences
  - Write out tree of recursive calls
  - Each node gets assigned the work done during that call to the procedure (dividing and combining)
  - Total work is sum of work at all nodes
Solve $T(n) = 3T(n/2) + cn$, where $c > 0$ is constant.

At level $k$: $(3/2)^k cn$

At bottom level: $(3/2)^{\log n} cn = c n^{\log_2 3}$
Recursion tree for multiplication

Solve $T(n) = 3T(n/2) + cn$, where $c > 0$ is constant.

Geometric series: When base of exponent is constant (e.g. 3/2), the largest term is a constant fraction of total.

Total work:

$$
\text{Total work} = \sum_{k=0}^{\log_2 n} \left( \frac{3}{2} \right)^k \cdot cn = \Theta(3^{\log_2 n}) = \Theta(n^{\log_2 3})
$$
Divide-and-conquer design

1. **Divide** the problem (instance) into subproblems.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** subproblem solutions.
Binary search

Find an element in a sorted array:

1. Divide: Check middle element.
2. Conquer: Recursively search 1 subarray.

Example: Find 9

3 5 7 8 9 12 15
Binary search

Find an element in a sorted array:

1. **Divide**: Check middle element.
2. **Conquer**: Recursively search 1 subarray.
3. **Combine**: Trivial.

**Example**: Find 9

<table>
<thead>
<tr>
<th>3</th>
<th>5</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>12</th>
<th>15</th>
</tr>
</thead>
</table>

S. Raskhodnikova and A. Smith. Based on notes by E. Demaine and C. Leiserson
Binary search

Find an element in a sorted array:

1. **Divide**: Check middle element.
2. **Conquer**: Recursively search 1 subarray.
3. **Combine**: Trivial.

**Example**: Find 9

3  5  7  8  9  12  15
Binary search

Find an element in a sorted array:

1. Divide: Check middle element.
2. Conquer: Recursively search 1 subarray.

Example: Find 9

3 5 7 8 9 12 15
Binary search

Find an element in a sorted array:

1. Divide: Check middle element.
2. Conquer: Recursively search 1 subarray.

Example: Find 9

3 5 7 8 9 12 15
Binary search

Find an element in a sorted array:

1. Divide: Check middle element.
2. Conquer: Recursively search 1 subarray.

Example: Find 9

3 5 7 8 9 12 15
**Binary Search**

\[
\text{\textsc{BinarySearch}}(b, A[1 \ldots n]) \triangleright \text{find } b \text{ in sorted } (\text{increasing order}) \text{ array } A
\]

1. If \( n=0 \) then return “not found”
2. If \( A[\lfloor n/2 \rfloor] = b \) then return \( \lceil n/2 \rceil \)
3. If \( A[\lfloor n/2 \rfloor] > b \) then
4. \hspace{1em} return \text{\textsc{BinarySearch}}(A[1 \ldots \lfloor n/2 \rfloor])
5. Else
6. \hspace{1em} return \( \lfloor n/2 \rfloor + \text{\textsc{BinarySearch}}(A[\lceil n/2 \rceil + 1 \ldots n]) \)
Proof of correctness

• Omitted here, similar to Merge-Sort
Recurrence for binary search

\[ T(n) = 1 \cdot T(n/2) + \Theta(1) \]

# subproblems
subproblem size
work dividing and combining
Recurrence for binary search

\[ T(n) = 1 \cdot T(n/2) + \Theta(1) \]

\[ = c \lceil \log n \rceil = \Theta(\lg n) . \]
Exponentiation

**Problem:** Compute $a^b$, where $b \in \mathbb{N}$ is $n$ bits long.

**Question:** How many multiplications?

**Naive algorithm:** $\Theta(b) = \Theta(2^n)$ (exponential in the input length!)

**Divide-and-conquer algorithm:**

$$a^b = \begin{cases} a^{b/2} \cdot a^{b/2} & \text{if } b \text{ is even;} \\ a^{(b-1)/2} \cdot a^{(b-1)/2} \cdot a & \text{if } b \text{ is odd.} \end{cases}$$

$$T(b) = T(b/2) + \Theta(1) \implies T(b) = \Theta(\log b) = \Theta(n) .$$
So far: 3 recurrences

- **Merge Sort**
  \[ T(n) = 2 \ T(n/2) + \Theta(n) = \Theta(n \log n) \]

- **Binary Search / Exponentiation**
  \[ T(n) = 1 \ T(n/2) + \Theta(1) = \Theta(\log n) \]

- **Multiplication**
  \[ T(n) = 3 \ T(n/2) + \Theta(n) = \Theta(n^{\log_2 3}) \]

**Coming soon:** systematic ways to solve these.