Lecture 4
The Divide-and-Conquer Design Paradigm

View in slide-show mode
Reminder: Merge Sort

Input array A

Divide

sort this half

Conquer

sort this half

Combine

merge two sorted halves
The Divide-and-Conquer Design Paradigm

1. Divide the problem (instance) into subproblems.

2. Conquer the subproblems by solving them recursively.

3. Combine subproblem solutions.
Example: Merge Sort

1. **Divide**: Trivial.
2. **Conquer**: Recursively sort 2 subarrays.
3. **Combine**: Linear-time merge.

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

- # subproblems
- subproblem size
- work dividing and combining
Master Theorem: Reminder

\[ T(n) = aT(n/b) + f(n) \]

**Case 1:**

\[ \frac{n^{\log_b a}}{f(n)} = \Omega(n^{\mathcal{E}}) \]

\[ T(n) = \Theta(n^{\log_b a}) \]

**Case 2:**

\[ \frac{f(n)}{n^{\log_b a}} = \Theta(\log^k n) \]

\[ T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \]

**Case 3:**

\[ \frac{f(n)}{n^{\log_b a}} = \Omega(n^{\mathcal{E}}) \]

\[ T(n) = \Theta(f(n)) \]

and \( a f(n/b) \leq c f(n) \) for \( c < 1 \)
Merge Sort: Solving the Recurrence

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

\[ a = 2, \quad b = 2, \quad f(n) = \Theta(n), \quad n^{\log_b a} = n \]

Case 2:

\[ \frac{f(n)}{n^{\log_b a}} = \Theta(\log^k n) \]

\[ T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \]

holds for \( k = 0 \)

\[ T(n) = \Theta(n \log n) \]
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
Recurrence for Binary Search

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

- # subproblems
- subproblem size
- work dividing and combining
Binary Search: Solving the Recurrence

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

\[ a = 1, \quad b = 2, \quad f(n) = \Theta(1), \quad n^{\log_b a} = n^0 = 1 \]

Case 2:
\[
\frac{f(n)}{n^{\log_b a}} = \Theta(\lg^k n)
\]
\[ T(n) = \Theta(n^{\log_b a} \lg^{k+1} n) \]

holds for \( k = 0 \)

\[ T(n) = \Theta(\lg n) \]
Problem: Compute $a^n$, where $n$ is a natural number

```python
Naive-Power(a, n)
    powerVal ← 1
    for i ← 1 to n
        powerVal ← powerVal . a
    return powerVal
```

What is the complexity? $T(n) = \Theta(n)$
Powering a Number: Divide & Conquer

Basic idea:

\[ a^n = \begin{cases} 
  a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \\
  a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd} 
\end{cases} \]
Powering a Number: Divide & Conquer

\[
\text{POWER}(a, n) \\
\text{if } n = 0 \text{ then return } 1 \\
\text{else if } n \text{ is even then} \\
\quad \text{val } \leftarrow \text{POWER}(a, \frac{n}{2}) \\
\quad \text{return } \text{val} \times \text{val} \\
\text{else if } n \text{ is odd then} \\
\quad \text{val } \leftarrow \text{POWER}(a, \frac{n-1}{2}) \\
\quad \text{return } \text{val} \times \text{val} \times a
\]
Powering a Number: Solving the Recurrence

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

Case 2: \[ \frac{f(n)}{n^{\log_b a}} = \Theta(\lg^k n) \]

\[ T(n) = \Theta(n^{\log_b a \lg^{k+1} n}) \]

holds for \( k = 0 \)

\[ T(n) = \Theta(\lg n) \]
Matrix Multiplication

**Input**: $A = [a_{ij}]$, $B = [b_{ij}]$

**Output**: $C = [c_{ij}] = A \cdot B$

$$
\begin{pmatrix}
  c_{11} & c_{12} & \ldots & c_{1n} \\
  c_{21} & c_{22} & \ldots & c_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  c_{n1} & c_{n2} & \ldots & c_{nn}
\end{pmatrix} = \begin{pmatrix}
  a_{11} & a_{12} & \ldots & a_{1n} \\
  a_{21} & a_{22} & \ldots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{n1} & a_{n2} & \ldots & a_{nn}
\end{pmatrix} \cdot \begin{pmatrix}
  b_{11} & b_{12} & \ldots & b_{1n} \\
  b_{21} & b_{22} & \ldots & b_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  b_{n1} & b_{n2} & \ldots & b_{nn}
\end{pmatrix}
$$

$$
c_{ij} = \sum_{1 \leq k \leq n} a_{ik} \cdot b_{kj}
$$
**Standard Algorithm**

\[
\begin{align*}
\text{for } i & \leftarrow 1 \text{ to } n \\
\text{ do for } j & \leftarrow 1 \text{ to } n \\
\text{ do } c_{ij} & \leftarrow 0 \\
\text{ for } k & \leftarrow 1 \text{ to } n \\
\text{ do } c_{ij} & \leftarrow c_{ij} + a_{ik} \cdot b_{kj}
\end{align*}
\]

**Running time** = $\Theta(n^3)$
Matrix Multiplication: Divide & Conquer

IDEA: **Divide** the $n \times n$ matrix into

$2 \times 2$ matrix of $(n/2) \times (n/2)$ submatrices

\[
\begin{pmatrix}
    c_{11} & c_{12} \\
    c_{21} & c_{22}
\end{pmatrix}
= \begin{pmatrix}
    a_{11} & a_{12} \\
    a_{21} & a_{22}
\end{pmatrix}
\cdot
\begin{pmatrix}
    b_{11} & b_{12} \\
    b_{21} & b_{22}
\end{pmatrix}
\]

\[
c_{11} = a_{11} b_{11} + a_{12} b_{21}
\]
Matrix Multiplication: Divide & Conquer

IDEA: **Divide** the $n \times n$ matrix into

2x2 matrix of $(n/2) \times (n/2)$ submatrices

$$
\begin{bmatrix}
    c_{11} & c_{12} \\
    c_{21} & c_{22}
\end{bmatrix} = \begin{bmatrix}
    a_{11} & a_{12} \\
    a_{21} & a_{22}
\end{bmatrix} \cdot \begin{bmatrix}
    b_{11} & b_{12} \\
    b_{21} & b_{22}
\end{bmatrix}
$$

$$
c_{12} = a_{11}b_{12} + a_{12}b_{22}
$$
Matrix Multiplication: Divide & Conquer

**IDEA:** *Divide* the $n \times n$ matrix into

2x2 matrix of $(n/2) \times (n/2)$ submatrices

\[
\begin{pmatrix}
    c_{11} & c_{12} \\
    c_{21} & c_{22}
\end{pmatrix}
= 
\begin{pmatrix}
    a_{11} & a_{12} \\
    a_{21} & a_{22}
\end{pmatrix}
\cdot
\begin{pmatrix}
    b_{11} & b_{12} \\
    b_{21} & b_{22}
\end{pmatrix}
\]

\[c_{21} = a_{21}b_{11} + a_{22}b_{21}\]
Matrix Multiplication: Divide & Conquer

IDEA: **Divide** the $n \times n$ matrix into

2x2 matrix of $(n/2) \times (n/2)$ submatrices

$$
\begin{bmatrix}
c_{11} & c_{12} \\
c_{21} & c_{22}
\end{bmatrix} =
\begin{bmatrix}
a_{11} & a_{12} \\
a_{21} & a_{22}
\end{bmatrix} \cdot
\begin{bmatrix}
b_{11} & b_{12} \\
b_{21} & b_{22}
\end{bmatrix}
$$

$c_{22} = a_{21}b_{12} + a_{22}b_{22}$
Matrix Multiplication: Divide & Conquer

\[
\begin{pmatrix}
  c_{11} & c_{12} \\
  c_{21} & c_{22}
\end{pmatrix} =
\begin{pmatrix}
  a_{11} & a_{12} \\
  a_{21} & a_{22}
\end{pmatrix} \cdot
\begin{pmatrix}
  b_{11} & b_{12} \\
  b_{21} & b_{22}
\end{pmatrix}
\]

\[
\begin{align*}
  c_{11} &= a_{11} b_{11} + a_{12} b_{21} \\
  c_{12} &= a_{11} b_{12} + a_{12} b_{22} \\
  c_{21} &= a_{21} b_{11} + a_{22} b_{21} \\
  c_{22} &= a_{21} b_{12} + a_{22} b_{22}
\end{align*}
\]

- 8 mults of \((n/2)\times(n/2)\) submatrices
- 4 adds of \((n/2)\times(n/2)\) submatrices
MATRIX-MULTIPLY (A, B)

// Assuming that both A and B are nxn matrices

if n = 1 then return A * B

else

partition A, B, and C as shown before

c_{11} = MATRIX-MULTIPLY(a_{11}, b_{11}) + MATRIX-MULTIPLY(a_{12}, b_{21})
c_{12} = MATRIX-MULTIPLY(a_{11}, b_{12}) + MATRIX-MULTIPLY(a_{12}, b_{22})
c_{21} = MATRIX-MULTIPLY(a_{21}, b_{11}) + MATRIX-MULTIPLY(a_{22}, b_{21})
c_{22} = MATRIX-MULTIPLY(a_{21}, b_{12}) + MATRIX-MULTIPLY(a_{22}, b_{22})

return C
Matrix Multiplication: Divide & Conquer Analysis

\[ T(n) = 8 \ T(n/2) + \Theta(n^2) \]

- 8 recursive calls
- each subproblem has size n/2
- submatrix addition
Matrix Multiplication: Solving the Recurrence

\[ T(n) = 8 \ T(n/2) + \Theta(n^2) \]

- \( a = 8, \ b = 2, \ f(n) = \Theta(n^2), \ n^{\log_b a} = n^3 \)

**Case 1:**

\[ \frac{n^{\log_b a}}{f(n)} = \Omega(n^\epsilon) \]

\[ T(n) = \Theta(n^{\log_b a}) \]

\[ T(n) = \Theta(n^3) \]

*No better than the ordinary algorithm!*

CS 473 – Lecture 4

Cevdet Aykanat and Mustafa Ozdal

Computer Engineering Department, Bilkent University
Matrix Multiplication: Strassen’s Idea

\[
\begin{pmatrix}
  c_{11} & c_{12} \\
  c_{21} & c_{22}
\end{pmatrix}
= 
\begin{pmatrix}
  a_{11} & a_{12} \\
  a_{21} & a_{22}
\end{pmatrix} \cdot 
\begin{pmatrix}
  b_{11} & b_{12} \\
  b_{21} & b_{22}
\end{pmatrix}
\]

Compute \( c_{11}, c_{12}, c_{21}, \) and \( c_{22} \) using 7 recursive multiplications
Matrix Multiplication: Strassen’s Idea

\[ P_1 = a_{11} \cdot x (b_{12} - b_{22}) \]
\[ P_2 = (a_{11} + a_{12}) \cdot x b_{22} \]
\[ P_3 = (a_{21} + a_{22}) \cdot x b_{11} \]
\[ P_4 = a_{22} \cdot x (b_{21} - b_{11}) \]
\[ P_5 = (a_{11} + a_{22}) \cdot x (b_{11} + b_{22}) \]
\[ P_6 = (a_{12} - a_{22}) \cdot x (b_{21} + b_{22}) \]
\[ P_7 = (a_{11} - a_{21}) \cdot x (b_{11} + b_{12}) \]

**Reminder:** Each submatrix is of size \((n/2) \times (n/2)\)

Each add/sub operation takes \(\Theta(n^2)\) time

Compute \(P_1 \ldots P_7\) using 7 recursive calls to matrix-multiply

**How to compute \(c_{ij}\) using \(P_1 \ldots P_7\)?**
Matrix Multiplication: Strassen’s Idea

\[
P_1 = a_{11} \times (b_{12} - b_{22})
\]
\[
P_2 = (a_{11} + a_{12}) \times b_{22}
\]
\[
P_3 = (a_{21} + a_{22}) \times b_{11}
\]
\[
P_4 = a_{22} \times (b_{21} - b_{11})
\]
\[
P_5 = (a_{11} + a_{22}) \times (b_{11} + b_{22})
\]
\[
P_6 = (a_{12} - a_{22}) \times (b_{21} + b_{22})
\]
\[
P_7 = (a_{11} - a_{21}) \times (b_{11} + b_{12})
\]

\[
c_{11} = P_5 + P_4 - P_2 + P_6
\]
\[
c_{12} = P_1 + P_2
\]
\[
c_{21} = P_3 + P_4
\]
\[
c_{22} = P_5 + P_1 - P_3 - P_7
\]

7 recursive multiply calls
18 add/sub operations

Does not rely on commutativity of multiplication
Matrix Multiplication: Strassen’s Idea

\[ P_1 = a_{11} \times (b_{12} - b_{22}) \]
\[ P_2 = (a_{11} + a_{12}) \times b_{22} \]
\[ P_3 = (a_{21} + a_{22}) \times b_{11} \]
\[ P_4 = a_{22} \times (b_{21} - b_{11}) \]
\[ P_5 = (a_{11} + a_{22}) \times (b_{11} + b_{22}) \]
\[ P_6 = (a_{12} - a_{22}) \times (b_{21} + b_{22}) \]
\[ P_7 = (a_{11} - a_{21}) \times (b_{11} + b_{12}) \]

\[ e.g. \text{ Show that } c_{12} = P_1 + P_2 \]

\[ c_{12} = P_1 + P_2 \]
\[ \quad = a_{11}(b_{12} - b_{22}) + (a_{11} + a_{12})b_{22} \]
\[ \quad = a_{11}b_{12} - a_{11}b_{22} + a_{11}b_{22} + a_{12}b_{22} \]
\[ \quad = a_{11}b_{12} + a_{12}b_{22} \]
Strassen’s Algorithm

1. **Divide**: Partition $A$ and $B$ into $(n/2) \times (n/2)$ submatrices. Form terms to be multiplied using $+$ and $-$.  

2. **Conquer**: Perform 7 multiplications of $(n/2) \times (n/2)$ submatrices recursively.

3. **Combine**: Form $C$ using $+$ and $-$ on $(n/2) \times (n/2)$ submatrices.

**Recurrence**: $T(n) = 7 \cdot T(n/2) + \Theta(n^2)$
Strassen’s Algorithm: Solving the Recurrence

\[ T(n) = 7 \cdot T(n/2) + \Theta(n^2) \]

\[ a = 7, \quad b = 2, \quad f(n) = \Theta(n^2), \quad n^{\log_b a} = n^{\lg 7} \]

**Case 1:**

\[ \frac{n^{\log_b a}}{f(n)} = \Omega(n^\varepsilon) \]

\[ T(n) = \Theta(n^{\log_b a}) \]

\[ T(n) = \Theta(n^{\lg 7}) \]

*Note:* \( \lg 7 \approx 2.81 \)
Strassen’s Algorithm

- The number **2.81** may not seem much smaller than **3**

- But, it is significant because the difference is in the exponent.

- Strassen’s algorithm **beats** the ordinary algorithm on today’s machines for **n ≥ 30** or so.

- Best to date: **Θ(n^{2.376...})** *(of theoretical interest only)*
VLSI Layout: Binary Tree Embedding

- **Problem**: Embed a complete binary tree with \( n \) leaves into a 2D grid with minimum area.

- **Example**: 

![Binary Tree Embedding Diagram](image)
Binary Tree Embedding

- Use divide and conquer

1. Embed the root node
2. Embed the left subtree
3. Embed the right subtree

What is the min-area required for n leaves?
Binary Tree Embedding

\[
W(n) = 2W(n/2) + 1
\]

\[
H(n) = H(n/2) + 1
\]
Binary Tree Embedding

- Solve the recurrences:
  \[ W(n) = 2W(n/2) + 1 \]
  \[ H(n) = H(n/2) + 1 \]

- \[ W(n) = \Theta(n) \]
- \[ H(n) = \Theta(\log n) \]

- \[ \text{Area}(n) = \Theta(n \log n) \]
Binary Tree Embedding

Example:
Binary Tree Embedding: H-Tree

- Use a different divide and conquer method

1. Embed root, left, right nodes
2. Embed subtree 1
3. Embed subtree 2
4. Embed subtree 3
5. Embed subtree 4

What is the min-area required for n leaves?
Binary Tree Embedding: H-Tree

\[ W(n) = 2W(n/4) + 1 \]

\[ H(n) = 2H(n/4) + 1 \]
Binary Tree Embedding: H-Tree

- Solve the recurrences:
  
  \[
  W(n) = 2W(n/4) + 1 \\
  H(n) = 2H(n/4) + 1
  \]

- \( W(n) = \Theta(\sqrt{n}) \)
- \( H(n) = \Theta(\sqrt{n}) \)

- \( \text{Area}(n) = \Theta(n) \)
Binary Tree Embedding: H-Tree

Example:

\[ W(n) \]

\[ H(n) \]
Correctness Proofs

- **Proof by induction** commonly used for D&C algorithms

- **Base case**: Show that the algorithm is correct when the recursion bottoms out (i.e., for sufficiently small $n$)

- **Inductive hypothesis**: Assume the alg. is correct for any recursive call on any smaller subproblem of size $k$ ($k < n$)

- **General case**: Based on the inductive hypothesis, prove that the alg. is correct for any input of size $n$
Example Correctness Proof: Powering a Number

```
POWER (a, n)
    if n = 0 then return 1

    else if n is even then
        val ← POWER (a, n/2)
        return val * val

    else if n is odd then
        val ← POWER (a, (n-1)/2)
        return val * val * a
```
Example Correctness Proof: Powering a Number

- **Base case**: POWER (a, 0) is correct, because it returns 1
- **Ind. hyp**: Assume POWER (a, k) is correct for any k < n
- **General case**:
  
  In POWER (a, n) function:
  
  If n is even:
  
  \[ \text{val} = a^{n/2} \ (\text{due to ind. hyp.}) \]
  
  it returns \( \text{val} \cdot \text{val} = a^n \)
  
  If n is odd:
  
  \[ \text{val} = a^{(n-1)/2} \ (\text{due to ind. hyp.}) \]
  
  it returns \( \text{val} \cdot \text{val} \cdot a = a^n \)

\[ \square \text{The correctness proof is complete} \]
Maximum Subarray Problem

- **Input**: An array of values
- **Output**: The contiguous subarray that has the largest sum of elements

Input array:

```
13  -3  -25  20  -3  -16  -23  18  20  -7  12  -22  -4  7
```

the maximum contiguous subarray
Maximum Subarray Problem: Divide & Conquer

- **Basic idea:**
  - Divide the input array into 2 from the middle
  - Pick the best solution among the following:
    1. The max subarray of the left half
    2. The max subarray of the right half
    3. The max subarray crossing the mid-point

![Diagram showing subarray divisions](image)
Maximum Subarray Problem: Divide & Conquer

- **Divide**: Trivial (divide the array from the middle)
- **Conquer**: Recursively compute the max subarrays of the left and right halves
- **Combine**: Compute the max-subarray crossing the mid-point \((can be done in \Theta(n) \text{ time})\). Return the max among the following:
  1. the max subarray of the left subarray
  2. the max subarray of the right subarray
  3. the max subarray crossing the mid-point

See textbook for the detailed solution.
Conclusion

- Divide and conquer is just one of several powerful techniques for algorithm design.
- Divide-and-conquer algorithms can be analyzed using recurrences and the master method (so practice this math).
- Can lead to more efficient algorithms