

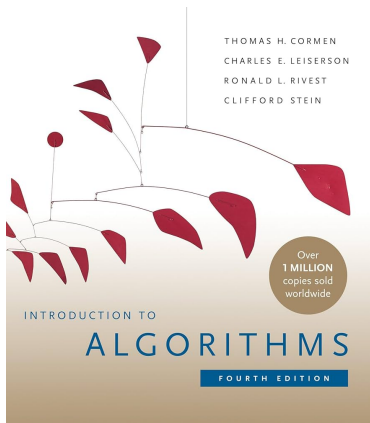
# Introduction to Algorithms

## Lecture 3: Divide and Conquer

Prof. Charles E. Leiserson and Prof. Erik Demaine  
Massachusetts Institute of Technology

August 5, 2025

# Introduction to Algorithms



Content has been extracted from *Introduction to Algorithms*, Fourth Edition, by Cormen, Leiserson, Rivest, and Stein. MIT Press. 2022.

Visit <https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/>.

Original slides from *Introduction to Algorithms 6.046J/18.401J*, Fall 2005 Class by Prof. Charles Leiserson and Prof. Erik Demaine. MIT OpenCourseWare Initiative available at <https://ocw.mit.edu/courses/6-046j-introduction-to-algorithms-sma-5503-fall-2005/>.

# The Divide & Conquer Design Paradigm

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

# Merge-sort

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

# Merge-sort

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

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

# Merge-sort

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

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Number of subproblems.



# Merge-sort

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

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Number of subproblems.

Subproblem size.

# Merge-sort

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

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Number of subproblems.

Subproblem size.

Work dividing and combining.



# Merge-sort

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

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Number of subproblems.

Subproblem size.

Work dividing and combining.

The diagram illustrates the recurrence relation  $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ . A red arrow points from the text "Number of subproblems." to the coefficient "2". A blue arrow points from the text "Subproblem size." to the fraction  $\frac{n}{2}$ . An orange arrow points from the text "Work dividing and combining." to the  $\Theta(n)$  term.

# Master Theorem (reprise)

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

# Master Theorem (reprise)

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

## Case 1

$$\begin{aligned} f(n) &= O\left(n^{\log_b a - \varepsilon}\right), \text{ constant } \varepsilon > 0 \\ \implies T(n) &= \Theta(n^{\log_b a}). \end{aligned}$$

# Master Theorem (reprise)

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Case 1

$$\begin{aligned} f(n) &= O\left(n^{\log_b a - \varepsilon}\right), \text{ constant } \varepsilon > 0 \\ \implies T(n) &= \Theta(n^{\log_b a}). \end{aligned}$$

Case 2

$$\begin{aligned} f(n) &= \Theta\left(n^{\log_b a} \lg^k n\right), \text{ constant } k \geq 0 \\ \implies T(n) &= \Theta(n^{\log_b a} \lg^{k+1} n). \end{aligned}$$

# Master Theorem (reprise)

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Case 1

$$f(n) = O\left(n^{\log_b a - \varepsilon}\right), \text{ constant } \varepsilon > 0 \\ \implies T(n) = \Theta(n^{\log_b a}).$$

Case 2

$$f(n) = \Theta\left(n^{\log_b a} \lg^k n\right), \text{ constant } k \geq 0 \\ \implies T(n) = \Theta(n^{\log_b a} \lg^{k+1} n).$$

Case 3

$$f(n) = \Omega\left(n^{\log_b(a) + \varepsilon}\right), \text{ constant } \varepsilon > 0, \text{ and regularity condition} \\ \implies T(n) = \Theta(f(n)).$$

# Master Theorem (reprise)

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Case 1

$$f(n) = O\left(n^{\log_b a - \varepsilon}\right), \text{ constant } \varepsilon > 0 \\ \implies T(n) = \Theta(n^{\log_b a}).$$

Case 2

$$f(n) = \Theta\left(n^{\log_b a} \lg^k n\right), \text{ constant } k \geq 0 \\ \implies T(n) = \Theta(n^{\log_b a} \lg^{k+1} n).$$

Case 3

$$f(n) = \Omega\left(n^{\log_b(a) + \varepsilon}\right), \text{ constant } \varepsilon > 0, \text{ and regularity condition} \\ \implies T(n) = \Theta(f(n)).$$

# Master Theorem (reprise)

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Case 1

$$f(n) = O\left(n^{\log_b a - \varepsilon}\right), \text{ constant } \varepsilon > 0 \\ \implies T(n) = \Theta(n^{\log_b a}).$$

Case 2

$$f(n) = \Theta\left(n^{\log_b a} \lg^k n\right), \text{ constant } k \geq 0 \\ \implies T(n) = \Theta(n^{\log_b a} \lg^{k+1} n).$$

Case 3

$$f(n) = \Omega\left(n^{\log_b(a) + \varepsilon}\right), \text{ constant } \varepsilon > 0, \text{ and regularity condition} \\ \implies T(n) = \Theta(f(n)).$$

$$\text{MERGE-SORT: } a = 2, b = 2 \implies n^{\log_b a} = n^{\log_2 2} = n \\ \implies \text{Case 2 } (k = 0) \implies T(n) = \Theta(n \lg n).$$

# Plan

Binary Search

Powering a Number

Fibonacci Numbers

Matrix Multiplication

Strassen's Algorithm

VLSI Tree Layout



# Binary Search

Find an element in a sorted array:

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

# 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.
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.
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.
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.
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.
3. **Combine:** Trivial.

Example:

Find 9

3	5	7	8	9	12	15
---	---	---	---	---	----	----

# Recurrence for Binary Search

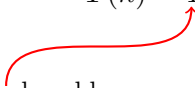
$$T(n) = 1T\left(\frac{n}{2}\right) + \Theta(1)$$



## Recurrence for Binary Search

$$T(n) = 1T\left(\frac{n}{2}\right) + \Theta(1)$$

Number of subproblems.



# Recurrence for Binary Search

$$T(n) = 1T\left(\frac{n}{2}\right) + \Theta(1)$$

Number of subproblems.

Subproblem size.

# Recurrence for Binary Search

$$T(n) = 1T\left(\frac{n}{2}\right) + \Theta(1)$$

Number of subproblems.

Subproblem size.

Work dividing and combining.

# Recurrence for Binary Search

$$T(n) = 1T\left(\frac{n}{2}\right) + \Theta(1)$$

Number of subproblems.

Subproblem size.

Work dividing and combining.

The diagram illustrates the recurrence relation  $T(n) = 1T\left(\frac{n}{2}\right) + \Theta(1)$ . A red arrow points from the text "Number of subproblems." to the coefficient "1". A blue arrow points from the text "Subproblem size." to the term  $\left(\frac{n}{2}\right)$ . An orange arrow points from the text "Work dividing and combining." to the term  $\Theta(1)$ .

# Recurrence for Binary Search

$$T(n) = 1T\left(\frac{n}{2}\right) + \Theta(1)$$

Number of subproblems.

Subproblem size.

Work dividing and combining.

$$\begin{aligned}\text{BINARY SEARCH: } a = 1, b = 2 &\implies n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \\ &\implies \text{Case 2 } (k = 0) \implies T(n) = \Theta(\lg n).\end{aligned}$$

# Plan

Binary Search

Powering a Number

Fibonacci Numbers

Matrix Multiplication

Strassen's Algorithm

VLSI Tree Layout

# Powering a Number

## Problem:

Compute  $a^n$ , where  $n \in \mathbb{N}$ .

## Naive algorithm:

$\Theta(n)$ .

# Powering a Number

## Problem:

Compute  $a^n$ , where  $n \in \mathbb{N}$ .

## Naive algorithm:

$\Theta(n)$ .

## Divide-and-conquer algorithm:

$$a^n = \begin{cases} a^{\frac{n}{2}} \cdot a^{\frac{n}{2}} & \text{if } n \text{ is even;} \\ a^{\frac{n-1}{2}} \cdot a^{\frac{n-1}{2}} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$



# Powering a Number

## Problem:

Compute  $a^n$ , where  $n \in \mathbb{N}$ .

## Naive algorithm:

$\Theta(n)$ .

## Divide-and-conquer algorithm:

$$a^n = \begin{cases} a^{\frac{n}{2}} \cdot a^{\frac{n}{2}} & \text{if } n \text{ is even;} \\ a^{\frac{n-1}{2}} \cdot a^{\frac{n-1}{2}} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1) \dots$$

# Powering a Number

## Problem:

Compute  $a^n$ , where  $n \in \mathbb{N}$ .

## Naive algorithm:

$\Theta(n)$ .

## Divide-and-conquer algorithm:

$$a^n = \begin{cases} a^{\frac{n}{2}} \cdot a^{\frac{n}{2}} & \text{if } n \text{ is even;} \\ a^{\frac{n-1}{2}} \cdot a^{\frac{n-1}{2}} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1) \implies T(n) = \Theta(\lg n).$$

# Plan

Binary Search

Powering a Number

Fibonacci Numbers

Matrix Multiplication

Strassen's Algorithm

VLSI Tree Layout

# Fibonacci Numbers

Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

0   1   1   2   3   5   8   13   21   34   ...

# Fibonacci Numbers

Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

0   1   1   2   3   5   8   13   21   34   ...

Naive recursive algorithm:

$$\Omega(\phi^n)$$

(exponential time), where  $\phi = \frac{1+\sqrt{5}}{2}$  is the golden ratio.

# Computing Fibonacci Numbers

## Bottom-up:

- ▶ Compute  $F_0, F_1, F_2, \dots, F_n$  in order, forming each number by summing the two previous.
- ▶ Running time:  $\Theta(n)$ .

---

<sup>1</sup>Computer Floating-Point Arithmetic and round-off errors, Kaluarachchi, 2022.

# Computing Fibonacci Numbers

## Bottom-up:

- ▶ Compute  $F_0, F_1, F_2, \dots, F_n$  in order, forming each number by summing the two previous.
- ▶ Running time:  $\Theta(n)$ .

## Naïve recursive squaring:

$F_n = \frac{\phi^n}{\sqrt{5}}$  rounded to the nearest integer.

---

<sup>1</sup>Computer Floating-Point Arithmetic and round-off errors, Kaluarachchi, 2022.

# Computing Fibonacci Numbers

## Bottom-up:

- ▶ Compute  $F_0, F_1, F_2, \dots, F_n$  in order, forming each number by summing the two previous.
- ▶ Running time:  $\Theta(n)$ .

## Naïve recursive squaring:

$F_n = \frac{\phi^n}{\sqrt{5}}$  rounded to the nearest integer.

- ▶ Recursive squaring:  $\Theta(\lg n)$  time.

---

<sup>1</sup>Computer Floating-Point Arithmetic and round-off errors, Kaluarachchi, 2022.



# Computing Fibonacci Numbers

## Bottom-up:

- ▶ Compute  $F_0, F_1, F_2, \dots, F_n$  in order, forming each number by summing the two previous.
- ▶ Running time:  $\Theta(n)$ .

## Naïve recursive squaring:

$F_n = \frac{\phi^n}{\sqrt{5}}$  rounded to the nearest integer.

- ▶ Recursive squaring:  $\Theta(\lg n)$  time.
- ▶ This method is unreliable, since floating-point arithmetic is prone to round-off errors<sup>1</sup>.

---

<sup>1</sup>Computer Floating-Point Arithmetic and round-off errors, Kaluarachchi, 2022.

# Recursive Squaring

Theorem:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n.$$

# Recursive Squaring

Theorem:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n.$$

Algorithm:

Recursive squaring.

Time =  $\Theta(\lg n)$ .

# Recursive Squaring

Theorem:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n.$$

Algorithm:

Recursive squaring.

Time =  $\Theta(\lg n)$ .

**Proof of theorem.** (Induction on  $n$ .)

Base ( $n = 1$ ):

# Recursive Squaring

Theorem:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n.$$

Algorithm:

Recursive squaring.

Time =  $\Theta(\lg n)$ .

**Proof of theorem.** (Induction on  $n$ .)

Base ( $n = 1$ ):

$$\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1$$

# Recursive Squaring

**Proof of theorem.** (Induction on  $n$ .)

Inductive step ( $n \geq 2$ ):

# Recursive Squaring

**Proof of theorem.** (Induction on  $n$ .)

Inductive step ( $n \geq 2$ ):

$$\begin{aligned} \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} &= \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \quad \blacksquare \end{aligned}$$

# Plan

Binary Search

Powering a Number

Fibonacci Numbers

Matrix Multiplication

Strassen's Algorithm

VLSI Tree Layout



# Matrix Multiplication

**Input:**  $A = [a_{ij}], B = [b_{ij}].$   $\left. \vphantom{\begin{matrix} \text{Input:} \\ \text{Output:} \end{matrix}} \right\} i, j = 1, 2, \dots, n.$   
**Output:**  $C = [c_{ij}] = A \cdot B.$

# Matrix Multiplication

**Input:**  $A = [a_{ij}], B = [b_{ij}].$   
**Output:**  $C = [c_{ij}] = A \cdot B.$   $\left. \vphantom{\begin{matrix} \text{Input} \\ \text{Output} \end{matrix}} \right\} i, j = 1, 2, \dots, n.$

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

# Matrix Multiplication

**Input:**  $A = [a_{ij}], B = [b_{ij}]$ .  
**Output:**  $C = [c_{ij}] = A \cdot B$ .  
 $\left. \vphantom{\begin{matrix} \text{Input} \\ \text{Output} \end{matrix}} \right\} i, j = 1, 2, \dots, n.$

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

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

# Standard Algorithm

---

```
for  $i \leftarrow 1$  to  $n$  do
  for  $j \leftarrow 1$  to  $n$  do
     $c_{ij} \leftarrow 0$ 
    for  $k \leftarrow 1$  to  $n$  do
       $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$ 
    end for
  end for
end for
```

---

**Running time**  $= \Theta(n^3)$

# Divide-and-Conquer Algorithm

## IDEA:

$n \times n$  matrix =  $2 \times 2$  matrix of  $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$  submatrices:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$
$$C = A \cdot B$$

$$\left. \begin{array}{lclcl} r & = & ae & + & bg \\ s & = & af & + & bh \\ t & = & ce & + & dg \\ u & = & cf & + & dh \end{array} \right\}$$

# Divide-and-Conquer Algorithm

## IDEA:

$n \times n$  matrix =  $2 \times 2$  matrix of  $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$  submatrices:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$
$$C = A \cdot B$$

$$\left. \begin{array}{lcl} r & = & ae + bg \\ s & = & af + bh \\ t & = & ce + dg \\ u & = & cf + dh \end{array} \right\} \begin{array}{l} \text{recursive} \\ \uparrow \\ 8 \text{ mults of } \left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right) \text{ submatrices.} \\ 4 \text{ adds of } \left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right) \text{ submatrices.} \end{array}$$

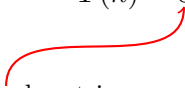
# Analysis of D&C Algorithm

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

# Analysis of D&C Algorithm

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Number of submatrices.





# Analysis of D&C Algorithm

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Number of submatrices.

Submatrix size.

# Analysis of D&C Algorithm

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Number of submatrices.

Submatrix size.

Work adding submatrices.

# Analysis of D&C Algorithm

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Number of submatrices.

Submatrix size.

Work adding submatrices.

# Analysis of D&C Algorithm

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Number of submatrices.      Submatrix size.      Work adding submatrices.

$$n^{\log_b a} = n^{\log_2 8} = n^3 \implies \text{Case 1} \implies T(n) = \Theta(n^3).$$

# Analysis of D&C Algorithm

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Number of submatrices.      Submatrix size.      Work adding submatrices.

$$n^{\log_b a} = n^{\log_2 8} = n^3 \implies \text{Case 1} \implies T(n) = \Theta(n^3).$$

**No better than the ordinary algorithm.**

# Plan

Binary Search

Powering a Number

Fibonacci Numbers

Matrix Multiplication

Strassen's Algorithm

VLSI Tree Layout

# Strassen's Idea

- Multiply 2 matrices with only 7 recursive multiplications.

$$P_1 = a \cdot (f - h)$$

$$P_2 = (a + b) \cdot h$$

$$P_3 = (c + d) \cdot e$$

$$P_4 = d \cdot (g - e)$$

$$P_5 = (a + d) \cdot (e + h)$$

$$P_6 = (b - d) \cdot (g + h)$$

$$P_7 = (a - c) \cdot (e + f)$$

# Strassen's Idea

- ▶ Multiply 2 matrices with only 7 recursive multiplications.

$$P_1 = a \cdot (f - h)$$

$$P_2 = (a + b) \cdot h$$

$$P_3 = (c + d) \cdot e$$

$$P_4 = d \cdot (g - e)$$

$$P_5 = (a + d) \cdot (e + h)$$

$$P_6 = (b - d) \cdot (g + h)$$

$$P_7 = (a - c) \cdot (e + f)$$

$$r = P_5 + P_4 - P_2 + P_6$$

$$s = P_1 + P_2$$

$$t = P_3 + P_4$$

$$u = P_5 + P_1 - P_3 - P_7$$

**7 mults, 18 adds/subs.**

## NOTE:

No reliance on commutativity of multiplication!



# Strassen's Idea

- Multiply 2 matrices with only 7 recursive multiplications.

$$P_1 = a \cdot (f - h)$$

$$P_2 = (a + b) \cdot h$$

$$P_3 = (c + d) \cdot e$$

$$P_4 = d \cdot (g - e)$$

$$P_5 = (a + d) \cdot (e + h)$$

$$P_6 = (b - d) \cdot (g + h)$$

$$P_7 = (a - c) \cdot (e + f)$$

$$r = P_5 + P_4 - P_2 + P_6$$

$$= (a + d)(e + h)$$

$$+ d(g - e) - (a + b)h$$

$$+ (b - d)(g + h)$$

$$= ae + ah + de + dh$$

$$+ dg - de - ah - bh$$

$$+ bg + bh - dg - dn$$

$$= ae + bg$$

# Strassen's Algorithm

1. **Divide:** Partition  $A$  and  $B$  into  $\frac{n}{2} \times \frac{n}{2}$  submatrices. Form terms to be multiplied using  $+$  and  $-$ .
2. **Conquer:** Perform 7 multiplications of  $\frac{n}{2} \times \frac{n}{2}$  submatrices recursively.
3. **Combine:** Form  $C$  using  $+$  and  $-$  on  $\frac{n}{2} \times \frac{n}{2}$  submatrices.

# Strassen's Algorithm

1. **Divide:** Partition  $A$  and  $B$  into  $\frac{n}{2} \times \frac{n}{2}$  submatrices. Form terms to be multiplied using  $+$  and  $-$ .
2. **Conquer:** Perform 7 multiplications of  $\frac{n}{2} \times \frac{n}{2}$  submatrices recursively.
3. **Combine:** Form  $C$  using  $+$  and  $-$  on  $\frac{n}{2} \times \frac{n}{2}$  submatrices.

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

# Analysis of Strassen

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

# Analysis of Strassen

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7} = n^{2.81} \implies \text{Case 1} \implies T(n) = \Theta(n^{\lg 7}).$$

# Analysis of Strassen

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7} = n^{2.81} \implies \text{Case 1} \implies T(n) = \Theta(n^{\lg 7}).$$

## Note:

The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for  $n \geq 32$  or so.

# Analysis of Strassen

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7} = n^{2.81} \implies \text{Case 1} \implies T(n) = \Theta(n^{\lg 7}).$$

## Note:

The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for  $n \geq 32$  or so.

**Best to date** (of theoretical interest only):  $\Theta(n^{2.376\dots})$ .

# Plan

Binary Search

Powering a Number

Fibonacci Numbers

Matrix Multiplication

Strassen's Algorithm

VLSI Tree Layout



# VLSI Layout

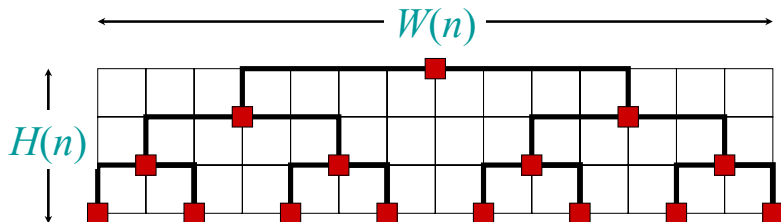
## Problem:

Embed a complete binary tree with  $n$  leaves in a grid using minimal area.

# VLSI Layout

## Problem:

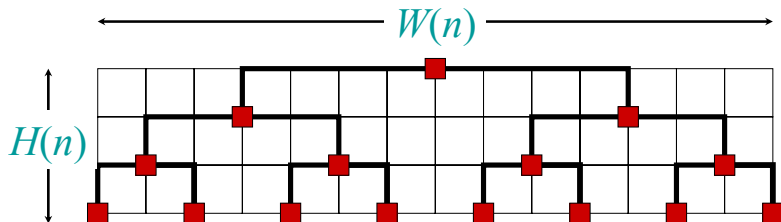
Embed a complete binary tree with  $n$  leaves in a grid using minimal area.



# VLSI Layout

## Problem:

Embed a complete binary tree with  $n$  leaves in a grid using minimal area.

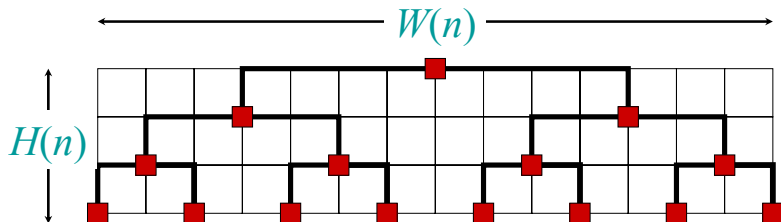


$$\begin{aligned} H(n) &= H\left(\frac{n}{2}\right) + \Theta(1) \\ &= \Theta(\lg n) \end{aligned}$$

# VLSI Layout

## Problem:

Embed a complete binary tree with  $n$  leaves in a grid using minimal area.



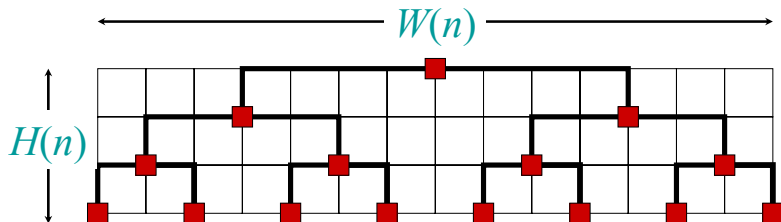
$$\begin{aligned} H(n) &= H\left(\frac{n}{2}\right) + \Theta(1) \\ &= \Theta(\lg n) \end{aligned}$$

$$\begin{aligned} W(n) &= 2W\left(\frac{n}{2}\right) + \Theta(1) \\ &= \Theta(n) \end{aligned}$$

# VLSI Layout

## Problem:

Embed a complete binary tree with  $n$  leaves in a grid using minimal area.

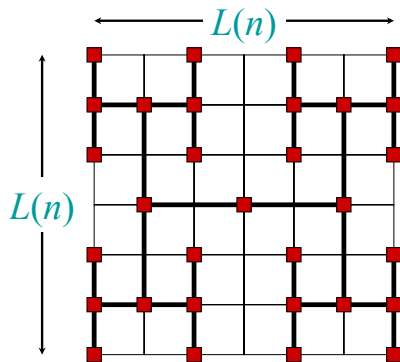


$$\begin{aligned} H(n) &= H\left(\frac{n}{2}\right) + \Theta(1) \\ &= \Theta(\lg n) \end{aligned}$$

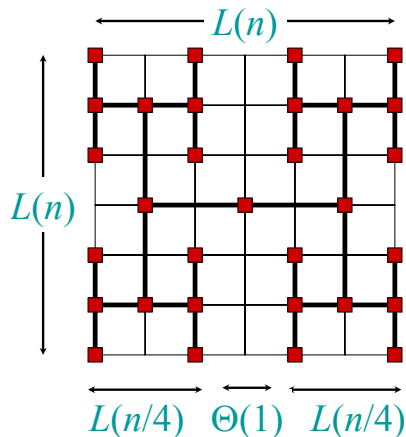
$$\begin{aligned} W(n) &= 2W\left(\frac{n}{2}\right) + \Theta(1) \\ &= \Theta(n) \end{aligned}$$

$$\mathbf{Area:} = \Theta(n \ln n)$$

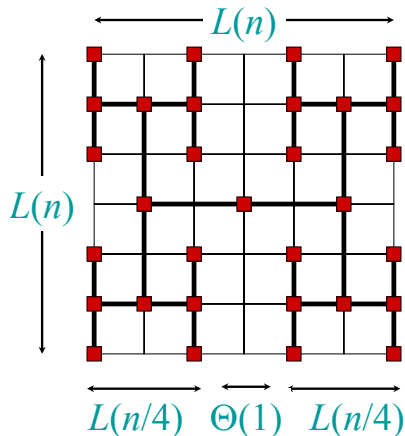
# H-tree Embedding



# H-tree Embedding



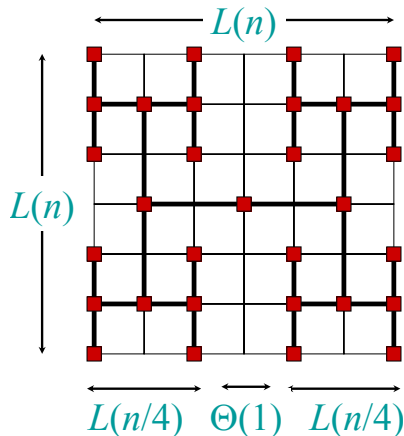
# H-tree Embedding



$$\begin{aligned} L(n) &= 2L\left(\frac{n}{4}\right) + \Theta(1) \\ &= \Theta(\sqrt{n}) \end{aligned}$$



# H-tree Embedding



$$\begin{aligned} L(n) &= 2L\left(\frac{n}{4}\right) + \Theta(1) \\ &= \Theta(\sqrt{n}) \end{aligned}$$

$$\text{Area} = \Theta(n)$$

# Conclusions

- ▶ 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).
- ▶ The divide-and-conquer strategy often leads to efficient algorithms.

End of Lecture 3.



# Top 5 Fundamental Takeaways

# Top 5 Fundamental Takeaways

- 5 **Applications Beyond Sorting and Searching:** Techniques like VLSI tree layout and H-tree embedding optimize spatial and computational complexity in fields like circuit design and geometry.

# Top 5 Fundamental Takeaways

- 5 **Applications Beyond Sorting and Searching:** Techniques like VLSI tree layout and H-tree embedding optimize spatial and computational complexity in fields like circuit design and geometry.
- 4 **Optimized Computation Techniques:** Problems such as exponentiation ( $O(\log n)$ ) and Fibonacci computation ( $O(\log n)$ ) benefit from divide-and-conquer methods that replace naive exponential-time approaches.

# Top 5 Fundamental Takeaways

- 5 **Applications Beyond Sorting and Searching:** Techniques like VLSI tree layout and H-tree embedding optimize spatial and computational complexity in fields like circuit design and geometry.
- 4 **Optimized Computation Techniques:** Problems such as exponentiation ( $O(\log n)$ ) and Fibonacci computation ( $O(\log n)$ ) benefit from divide-and-conquer methods that replace naive exponential-time approaches.
- 3 **Master Theorem for Complexity Analysis:** A formulaic method to determine the time complexity of divide-and-conquer algorithms based on recurrence relations.



# Top 5 Fundamental Takeaways

- 5 **Applications Beyond Sorting and Searching:** Techniques like VLSI tree layout and H-tree embedding optimize spatial and computational complexity in fields like circuit design and geometry.
- 4 **Optimized Computation Techniques:** Problems such as exponentiation ( $O(\log n)$ ) and Fibonacci computation ( $O(\log n)$ ) benefit from divide-and-conquer methods that replace naive exponential-time approaches.
- 3 **Master Theorem for Complexity Analysis:** A formulaic method to determine the time complexity of divide-and-conquer algorithms based on recurrence relations.
- 2 **Efficient Algorithms Using Divide and Conquer:** Algorithms like merge sort ( $O(n \log n)$ ), binary search ( $O(\log n)$ ), and Strassen's matrix multiplication ( $O(n^{2.81})$ ) leverage this paradigm for improved efficiency.

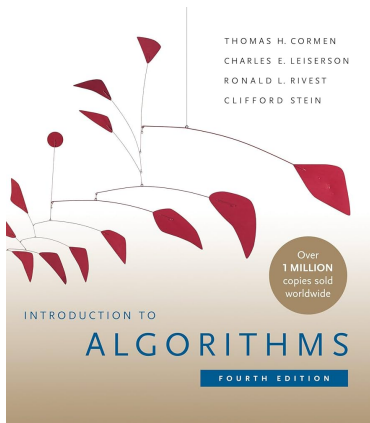
# Top 5 Fundamental Takeaways

- 5 **Applications Beyond Sorting and Searching:** Techniques like VLSI tree layout and H-tree embedding optimize spatial and computational complexity in fields like circuit design and geometry.
- 4 **Optimized Computation Techniques:** Problems such as exponentiation ( $O(\log n)$ ) and Fibonacci computation ( $O(\log n)$ ) benefit from divide-and-conquer methods that replace naive exponential-time approaches.
- 3 **Master Theorem for Complexity Analysis:** A formulaic method to determine the time complexity of divide-and-conquer algorithms based on recurrence relations.
- 2 **Efficient Algorithms Using Divide and Conquer:** Algorithms like merge sort ( $O(n \log n)$ ), binary search ( $O(\log n)$ ), and Strassen's matrix multiplication ( $O(n^{2.81})$ ) leverage this paradigm for improved efficiency.
- 1 **Divide and Conquer Paradigm:** This algorithmic approach breaks a problem into smaller subproblems, solves them recursively, and combines the results efficiently.

# Top 5 Fundamental Takeaways

- 5 **Applications Beyond Sorting and Searching:** Techniques like VLSI tree layout and H-tree embedding optimize spatial and computational complexity in fields like circuit design and geometry.
- 4 **Optimized Computation Techniques:** Problems such as exponentiation ( $O(\log n)$ ) and Fibonacci computation ( $O(\log n)$ ) benefit from divide-and-conquer methods that replace naive exponential-time approaches.
- 3 **Master Theorem for Complexity Analysis:** A formulaic method to determine the time complexity of divide-and-conquer algorithms based on recurrence relations.
- 2 **Efficient Algorithms Using Divide and Conquer:** Algorithms like merge sort ( $O(n \log n)$ ), binary search ( $O(\log n)$ ), and Strassen's matrix multiplication ( $O(n^{2.81})$ ) leverage this paradigm for improved efficiency.
- 1 **Divide and Conquer Paradigm:** This algorithmic approach breaks a problem into smaller subproblems, solves them recursively, and combines the results efficiently.

# Introduction to Algorithms



Content has been extracted from *Introduction to Algorithms*, Fourth Edition, by Cormen, Leiserson, Rivest, and Stein. MIT Press. 2022.

Visit <https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/>.

Original slides from *Introduction to Algorithms 6.046J/18.401J*, Fall 2005 Class by Prof. Charles Leiserson and Prof. Erik Demaine. MIT OpenCourseWare Initiative available at <https://ocw.mit.edu/courses/6-046j-introduction-to-algorithms-sma-5503-fall-2005/>.