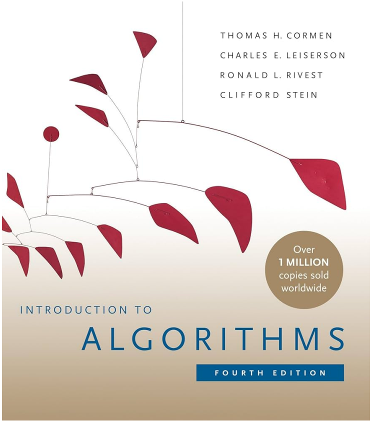
Introduction to Algorithms
Lecture 3: Divide and Conquer

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

August 12, 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.

Work dividing and combining.

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.

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

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

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

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

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

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

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

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

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

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

Merge-sort: $a = 2, b = 2 \implies n^{log_b a} = n^{log_2 2} = n$

$\implies$ 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.

Work dividing and combining.

Subproblem size.

Recurrence for Binary Search

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

Number of subproblems.

Work dividing and combining.

Subproblem size.

Recurrence for Binary Search

$$T(n) = \underset{\text{Number of subproblems.}}{1}T\left(\frac{n}{\underset{\text{Subproblem size.}}{2}}\right) + \underset{\text{Work dividing and combining.}}{\Theta(1)}$$

Number of subproblems.

Subproblem size.

Work dividing and combining.

$$\textsc{Binary Search}: \; a = 1, b = 2 \implies n^{log_b a} = n^{log_2 1} = n^0 = 1$$
$$\implies \text{Case 2 } (k = 0) \implies T(n) = \Theta(\lg n).$$

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:

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

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

Powering a Number

Problem:

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

Naive algorithm:

$\Theta(n)$.

Divide-and-conquer algorithm:

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

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

Plan

Binary Search

Powering a Number

Fibonacci Numbers

Matrix Multiplication

Strassen's Algorithm

VLSI Tree Layout

Fibonacci Numbers

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \ldots$$

Fibonacci Numbers

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \ldots$$

Recursive definition:

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

Fibonacci Numbers

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \ldots$$

Recursive definition:

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

Naive recursive algorithm:

Fibonacci Numbers

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \ldots$$

Recursive definition:

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

Naive recursive algorithm:

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

Fibonacci Numbers

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \ldots$$

Recursive definition:

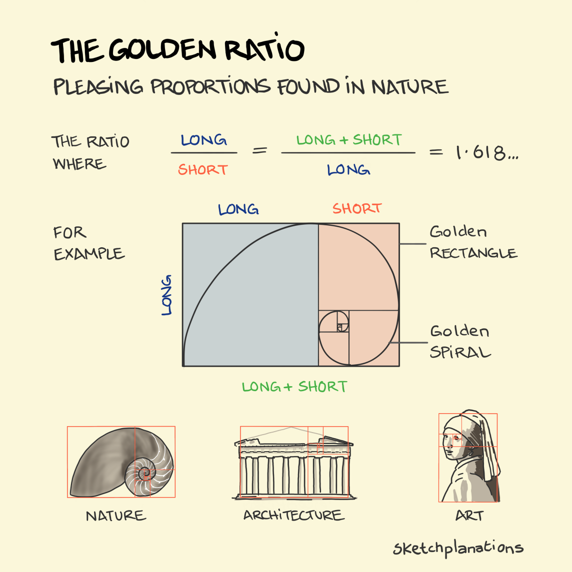
$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

Naive recursive algorithm:

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

where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

The Golden Ratio (1.61803398875...)



sketchplanations.com/the-golden-ratio

Fibonacci Numbers

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \ldots$$

Recursive definition:

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

Naive recursive algorithm:

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

where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Fibonacci Numbers

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \ldots$$

Recursive definition:

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

Naive recursive algorithm:

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

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

Computing Fibonacci Numbers

Bottom-up:

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

[1] Computer Floating-Point Arithmetic and round-off errors, Kaluarachchi, 2022.

Computing Fibonacci Numbers

Bottom-up:

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

Naive recursive squaring:

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

[1] Computer Floating-Point Arithmetic and round-off errors, Kaluarachchi, 2022.

Computing Fibonacci Numbers

Bottom-up:

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

Naive recursive squaring:

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

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

[1] Computer Floating-Point Arithmetic and round-off errors, Kaluarachchi, 2022.

Computing Fibonacci Numbers

Bottom-up:

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

Naive 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[1].

[1] Computer Floating-Point Arithmetic and round-off errors, Kaluarachchi, 2022.

Recursive Squaring

Theorem:

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.

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.)

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 case ($n = 1$).

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 case ($n = 1$).

Inductive Hypothesis ($n = k$).

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 case ($n = 1$).

Inductive Hypothesis ($n = k$).

Inductive Step ($n = k + 1$).

Recursive Squaring

Theorem:

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

Proof of theorem. (Induction on n.)

Base case ($n = 1$):

Recursive Squaring

Theorem:

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

Proof of theorem. (Induction on n.)

Base case ($n = 1$):

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

Recursive Squaring

Theorem:

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

Proof of theorem. (Induction on n.)

Inductive Hypothesis ($n = k$):

Recursive Squaring

Theorem:

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

Proof of theorem. (Induction on n.)

Inductive Hypothesis ($n = k$):

$$A^k = \begin{bmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{bmatrix}$$

Recursive Squaring

Theorem:

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

Proof of theorem. (Induction on n.)

Inductive Step ($n = k + 1$):

Recursive Squaring

Theorem:

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

Proof of theorem. (Induction on n.)

Inductive Step ($n = k + 1$):

$$A^{k+1} = \begin{bmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{bmatrix}$$

$$A^k \cdot A = \begin{bmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{bmatrix}$$

$$\begin{bmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{bmatrix}$$

Recursive Squaring

$$F_0, F_1, F_2, F_3, \ldots, F_{k-2}, F_{k-1}, F_k, F_{k+1}, F_{k+2}, \ldots$$

Proof of theorem. (Induction on n.)

Inductive Step ($n = k + 1$):

$$A^{k+1} = \begin{bmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{bmatrix}$$

$$A^k \cdot A = \begin{bmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{bmatrix}$$

$$\begin{bmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{bmatrix}$$

$$(1,1)\colon\ F_{k+1} + F_k = F_{k+2}$$

$$(1,2)\colon\ F_{k+1} + 0 = F_{k+1}$$

$$(2,1)\colon\ F_k + F_{k-1} = F_{k+1}$$

$$(2,2)\colon\ F_k + 0 = F_k$$

■

Plan

Binary Search

Powering a Number

Fibonacci Numbers

Matrix Multiplication

Strassen's Algorithm

VLSI Tree Layout

Matrix Multiplication

$$\left.\begin{array}{rl} \textbf{Input:} & A = [a_{ij}], B = [b_{ij}]. \\ \textbf{Output:} & C = [c_{ij}] = A \cdot B \end{array}\right\} i, j = 1, 2, \ldots, n.$$

Matrix Multiplication

$$\left.\begin{array}{rl}\textbf{Input:} & A = [a_{ij}], B = [b_{ij}]. \\ \textbf{Output:} & C = [c_{ij}] = A \cdot B\end{array}\right\} i, j = 1, 2, \ldots, n.$$

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

Matrix Multiplication

$$\left.\begin{array}{rl} \textbf{Input:} & A = [a_{ij}], B = [b_{ij}]. \\ \textbf{Output:} & C = [c_{ij}] = A \cdot B \end{array}\right\} i, j = 1, 2, \ldots, n.$$

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

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

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:

$$\left[\begin{array}{cccccccc:cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\
33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\
49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 & 60 & 61 & 62 & 63 & 64 \\
65 & 66 & 67 & 68 & 69 & 70 & 71 & 72 & 73 & 74 & 75 & 76 & 77 & 78 & 79 & 80 \\
81 & 82 & 83 & 84 & 85 & 86 & 87 & 88 & 89 & 90 & 91 & 92 & 93 & 94 & 95 & 96 \\
97 & 98 & 99 & 100 & 101 & 102 & 103 & 104 & 105 & 106 & 107 & 108 & 109 & 110 & 111 & 112 \\
113 & 114 & 115 & 116 & 117 & 118 & 119 & 120 & 121 & 122 & 123 & 124 & 125 & 126 & 127 & 128 \\
\hdashline
129 & 130 & 131 & 132 & 133 & 134 & 135 & 136 & 137 & 138 & 139 & 140 & 141 & 142 & 143 & 144 \\
145 & 146 & 147 & 148 & 149 & 150 & 151 & 152 & 153 & 154 & 155 & 156 & 157 & 158 & 159 & 160 \\
161 & 162 & 163 & 164 & 165 & 166 & 167 & 168 & 169 & 170 & 171 & 172 & 173 & 174 & 175 & 176 \\
177 & 178 & 179 & 180 & 181 & 182 & 183 & 184 & 185 & 186 & 187 & 188 & 189 & 190 & 191 & 192 \\
193 & 194 & 195 & 196 & 197 & 198 & 199 & 200 & 201 & 202 & 203 & 204 & 205 & 206 & 207 & 208 \\
209 & 210 & 211 & 212 & 213 & 214 & 215 & 216 & 217 & 218 & 219 & 220 & 221 & 222 & 223 & 224 \\
225 & 226 & 227 & 228 & 229 & 230 & 231 & 232 & 233 & 234 & 235 & 236 & 237 & 238 & 239 & 240 \\
241 & 242 & 243 & 244 & 245 & 246 & 247 & 248 & 249 & 250 & 251 & 252 & 253 & 254 & 255 & 256
\end{array}\right]$$

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:

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

$$C = A \cdot B$$

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

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:

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

$$C = A \cdot B$$

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

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

$$T(n) = \underset{\text{Number of submatrices.}}{8}T\left(\underset{\text{Submatrix size.}}{\frac{n}{2}}\right) + \Theta(\underset{\text{Work adding submatrices.}}{n^2})$$

Analysis of D&C Algorithm

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

Number of submatrices.

Work adding submatrices.

Submatrix size.

Analysis of D&C Algorithm

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

Number of submatrices.

Work adding submatrices.

Submatrix size.

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

Analysis of D&C Algorithm

$$T(n) = \underbrace{8}_{\text{Number of submatrices.}} T\left(\underbrace{\frac{n}{2}}_{\text{Submatrix size.}}\right) + \underbrace{\Theta(n^2)}_{\text{Work adding submatrices.}}$$

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

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.

$$
\begin{aligned}
P_1 &= a \cdot (f - h) \\
P_2 &= (a + b) \cdot h \\
P_3 &= (c + d) \cdot e \\
P_4 &= d \cdot (g - e) \\
P_5 &= (a + d) \cdot (e + h) \\
P_6 &= (b - d) \cdot (g + h) \\
P_7 &= (a - c) \cdot (e + f)
\end{aligned}
$$

Strassen's Idea

- Multiply 2 matrices with only 7 recursive multiplications.

$$
\begin{aligned}
P_1 &= a \cdot (f - h) \\
P_2 &= (a + b) \cdot h \\
P_3 &= (c + d) \cdot e \\
P_4 &= d \cdot (g - e) \\
P_5 &= (a + d) \cdot (e + h) \\
P_6 &= (b - d) \cdot (g + h) \\
P_7 &= (a - c) \cdot (e + f)
\end{aligned}
$$

$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
s &= P_1 + P_2 \\
t &= P_3 + P_4 \\
u &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

7 mults, 18 adds/subs.

NOTE:
No reliance on commutativity of multiplication!

Strassen's Idea

- Multiply 2 matrices with only 7 recursive multiplications.

$$
\begin{aligned}
P_1 &= a \cdot (f - h) \\
P_2 &= (a + b) \cdot h \\
P_3 &= (c + d) \cdot e \\
P_4 &= d \cdot (g - e) \\
P_5 &= (a + d) \cdot (e + h) \\
P_6 &= (b - d) \cdot (g + h) \\
P_7 &= (a - c) \cdot (e + f)
\end{aligned}
$$

$$
\begin{aligned}
r =& P_5 + P_4 - P_2 + P_6 \\
=& (a + d)(e + h) \\
& + d(g – e) – (a + b)h \\
& + (b – d)(g + h) \\
=& ae + ah + de + dh \\
& + dg - de - ah - bh \\
& + bg + bh - dg - dn \\
=& ae + bg
\end{aligned}
$$

Strassen's Idea

- Multiply 2 matrices with only 7 recursive multiplications.

$$
\begin{aligned}
P_1 &= a \cdot (f - h) \\
P_2 &= (a + b) \cdot h \\
P_3 &= (c + d) \cdot e \\
P_4 &= d \cdot (g - e) \\
P_5 &= (a + d) \cdot (e + h) \\
P_6 &= (b - d) \cdot (g + h) \\
P_7 &= (a - c) \cdot (e + f)
\end{aligned}
$$

$$
\begin{aligned}
t =& P_3 + P_4 \\
=& (c + d)e + d(g - e) \\
=& ce + de + dg - de \\
=& ce + dg
\end{aligned}
$$

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.

Theoretical Notes

- Strassen's algorithm was the first to beat $O(n^3)$ time.
- Coppersmith–Winograd algorithm runs in $O(n^{2.376})$ time.
- Current best asymptotic bound (not practical): $O(n^{2.37286})$.

Practical Issues with Strassen's Algorithm

- Higher constant factor than the naive $O(n^3)$ method.
- Performs poorly on sparse matrices.
- Not numerically stable — larger error accumulation.
- Submatrices consume extra space, especially with copying.

Additional Considerations

- Numerical stability problem is less severe than previously thought.
- Index calculations can reduce space requirements.
- Researchers have sought the crossover point where Strassen outperforms the naive method — results vary.

Plan

Binary Search

Powering a Number

Fibonacci Numbers

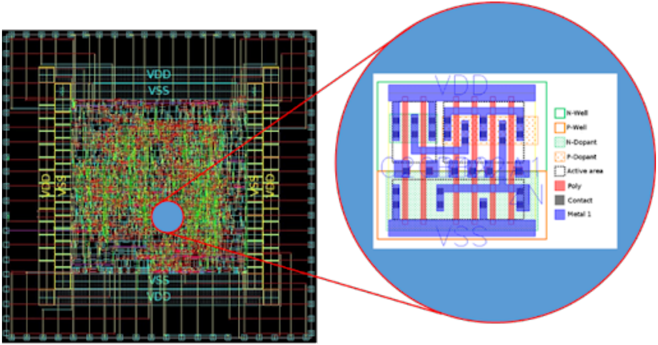
Matrix Multiplication

Strassen's Algorithm

VLSI Tree Layout

VLSI Layout

- VLSI – Very Large Scale Integration.



maven-silicon.com

What is VLSI Layout? – The Blueprint of a Chip

- **Definition:** Physical geometric representation of an IC design used for fabrication.
- **Purpose:** Translates logic/schematic into manufacturable mask patterns.
- **Key Elements:**
  - Active regions (transistors), polysilicon gates
  - Metal interconnect layers (wiring)
  - Contacts and vias (vertical connections)
  - Isolation and well structures

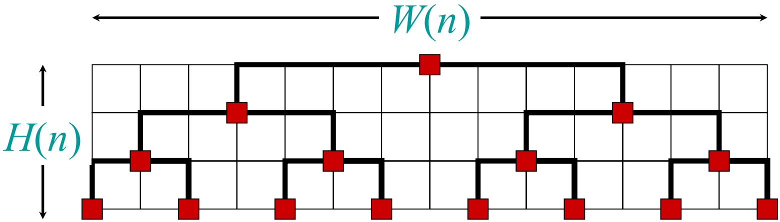
VLSI Layout

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

VLSI Layout

Problem:

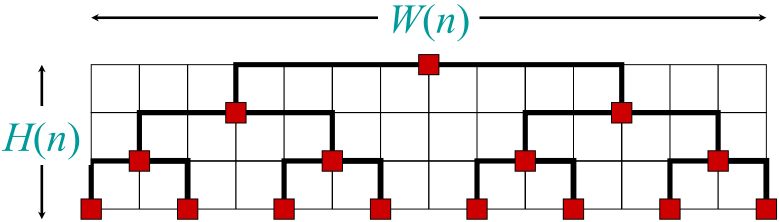
Embed a complete binary tree with n leaves in a grid using minimal area.



VLSI Layout

Problem:

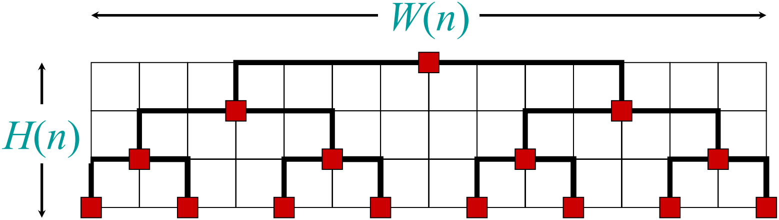
Embed a complete binary tree with n leaves in a grid using minimal area.



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

VLSI Layout

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



$$H(n) = H\left(\frac{n}{2}\right) + \Theta(1) \qquad W(n) = 2W\left(\frac{n}{2}\right) + \Theta(1)$$

VLSI Tree Layout Recurrences

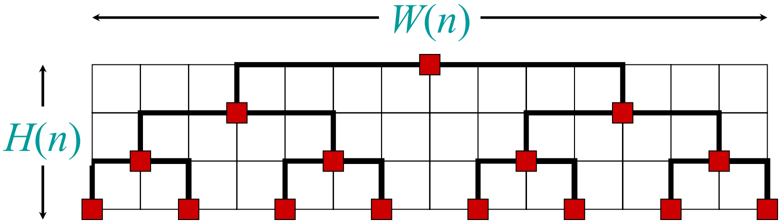
- Why $\Theta(1)$ non-recursive cost?
  - Combining two $n/2$ sublayouts adds only a *fixed* vertical channel/spacing and a level of connectors.
  - This overhead does *not* depend on n (no n-sized merge), hence constant per level.
- Height recurrence: $H(n) = H(\frac{n}{2}) + \Theta(1) \Rightarrow Case?$
- Width recurrence: $W(n) = 2\,W(\frac{n}{2}) + \Theta(1) \Rightarrow Case?$

VLSI Tree Layout Recurrences

- Why $\Theta(1)$ non-recursive cost?
  - Combining two $\frac{n}{2}$ sublayouts adds only a *fixed* vertical channel/spacing and a level of connectors.
  - This overhead does *not* depend on n (no n-sized merge), hence constant per level.
- Height recurrence:
  $H(n) = H(\frac{n}{2}) + \Theta(1) \Rightarrow \text{Case } 2 : H(n) = \Theta(\lg n)$
- Width recurrence:
  $W(n) = 2\,W(\frac{n}{2}) + \Theta(1) \Rightarrow \text{Case } 1 : W(n) = \Theta(n)$

VLSI Layout

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

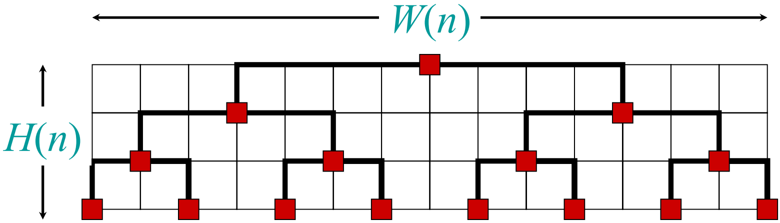


$$\begin{aligned} H(n) &= H\left(\frac{n}{2}\right) + \Theta(1) \\ &= \Theta(\lg n) \end{aligned}$$

VLSI Layout

Problem:

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



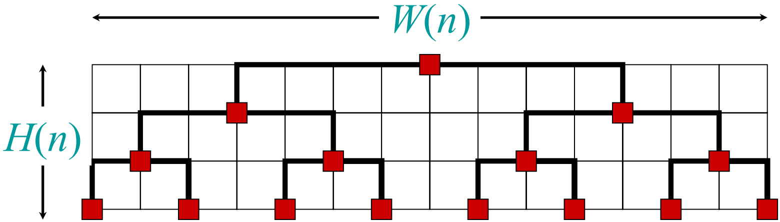
$$H(n) = H\left(\frac{n}{2}\right) + \Theta(1) = \Theta(\lg n)$$

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

VLSI Layout

Problem:

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

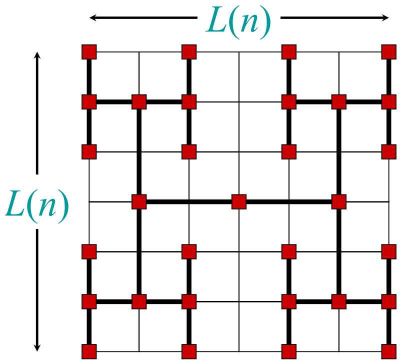


$$H(n) = H\left(\frac{n}{2}\right) + \Theta(1) = \Theta(\lg n)$$

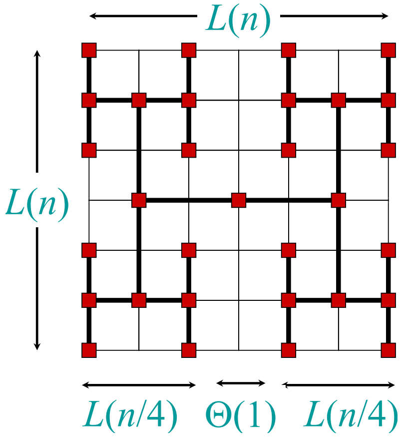
$$W(n) = 2W\left(\frac{n}{2}\right) + \Theta(1) = \Theta(n)$$

Area: $= \Theta(n \lg n)$

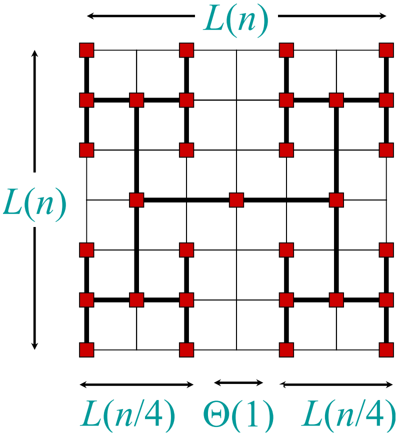
H-tree Embedding



H-tree Embedding

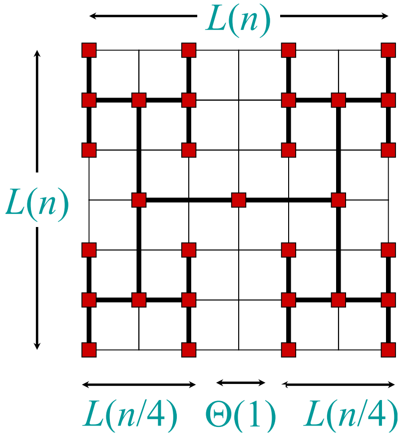


H-tree Embedding



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

H-tree Embedding



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

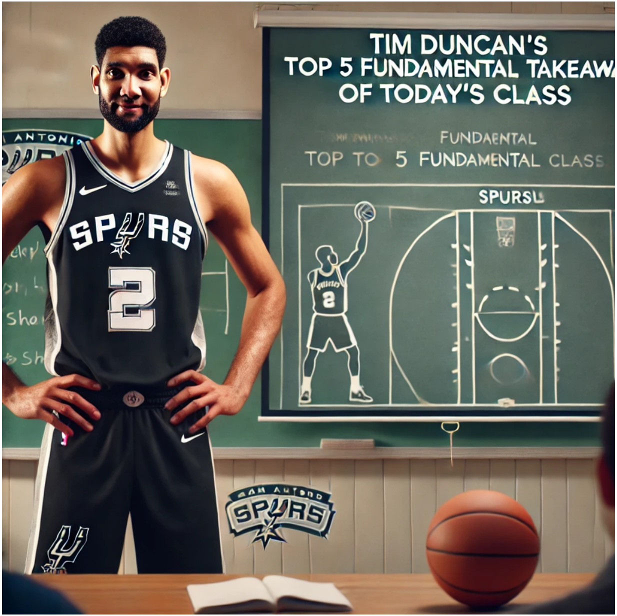
Area: $= \Theta(n)$

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.

TDT5FTOTTC


TIM DUNCAN'S
TOP 5 FUNDAMENTAL TAKEAW
OF TODAY'S CLASS
FUNDAENTAL
TOP TO 5 FUNDAMENTAL CLASS
SPURS
2
SPURS

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.

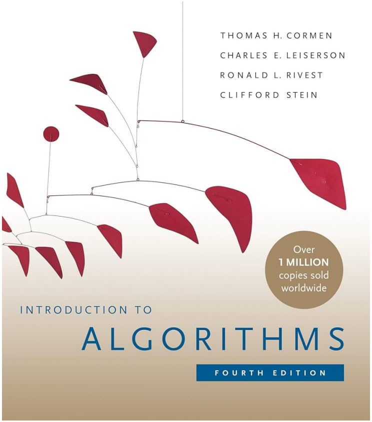
4 **Optimized Computation Techniques**: Problems such as exponentiation ($O(\log n)$) and Fibonacci computation ($O(\log n)$) benefit from divide-and-conquer methods that replace naive exponential-time approaches.

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/.