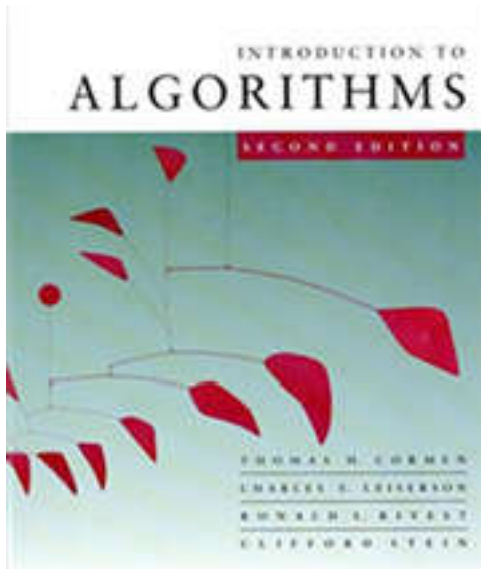


Introduction to Algorithms



Divide and Conquer

- Binary search
- Powering a number
- Fibonacci numbers
- Matrix multiplication
- Strassen's algorithm
- VLSI tree layout

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.

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.

$$T(n) = 2T(n/2) + \Theta(n)$$

subproblems *subproblem size* *work dividing and combining*

Master theorem (reprise)

$$T(n) = aT(n/b) + f(n)$$

CASE 1: $f(n) = O(n \log_b a - \Sigma)$, constant $\Sigma > 0$

Ⓜ $T(n) = \Theta(n \log_b a)$.

CASE 2: $f(n) = \Theta(n \log_b a \lg_k n)$, constant $k \in \mathbb{N}$

Ⓜ $T(n) = \Theta(n \log_b a \lg_k n)$.

CASE 3: $f(n) = \Omega(n \log_b a + \Sigma)$, constant $\Sigma > 0$,

and regularity condition

Ⓜ $T(n) = \Theta(f(n))$.

Master theorem (reprise)

$$T(n) = aT(n/b) + f(n)$$

CASE 1: $f(n) = O(n \log b a - \Sigma)$, constant $\Sigma > 0$

Ⓜ $T(n) = \Theta(n \log b a)$.

CASE 2: $f(n) = \Theta(n \log b a \lg k n)$, constant $k \varepsilon 0$

Ⓜ $T(n) = \Theta(n \log b a \lg k+1 n)$.

CASE 3: $f(n) = \Theta(n \log b a + \Sigma)$, constant $\Sigma > 0$,
and regularity condition

Ⓜ $T(n) = \Theta(f(n))$.

Merge sort: $a = 2, b = 2$ Ⓜ $n \log b a = n \log 2 2 = n$

Ⓜ **CASE 2** ($k = 0$) Ⓜ $T(n) = \Theta(n \lg n)$.

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

$$T(n) = 1T(n/2) + \Theta(1)$$

subproblems *subproblem size* *work dividing and combining*

Recurrence for binary search

$$T(n) = 1T(n/2) + \Theta(1)$$

subproblems *subproblem size* *work dividing and combining*

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \Rightarrow \text{CASE 2 } (k=0)$$
$$\Rightarrow T(n) = \Theta(\lg n) .$$

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:

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

Powering a number

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

Naive algorithm: $\Theta(n)$.

Divide-and-conquer algorithm:

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

$$T(n) = T(n/2) + \Theta(1) \quad \text{Ⓜ} \quad T(n) = \Theta(\lg n) .$$

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 $\phi = (1 + \sqrt{5})/2$
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: $\Theta(n)$.

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: $\Theta(n)$.

Naive recursive squaring:

$F_n = \varphi^n / 5$ 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.

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 ($n = 1$):
$$\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1 .$$

Recursive squaring

Inductive step ($n \geq 2$):

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

Matrix multiplication

Input: $A = [a_{ij}], B = [b_{ij}].$ } $i, j = 1, 2, \dots, n.$
Output: $C = [c_{ij}] = A \cdot B.$

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

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

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  $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$ 
```

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  $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$ 
```

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

Divide-and-conquer algorithm

IDEA:

$n \times n$ matrix = 2×2 matrix of $(n/2) \times (n/2)$ submatrices:

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

$$C = A \cdot B$$

$$r = ae + bg$$

$$s = af + bh$$

$$t = ce + dg$$

$$u = cf + dh$$

8 mults of $(n/2) \times (n/2)$ submatrices

4 adds of $(n/2) \times (n/2)$ submatrices

Divide-and-conquer algorithm

IDEA:

$n \times n$ matrix = 2×2 matrix of $(n/2) \times (n/2)$ submatrices:

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

$$C = A \cdot B$$

$$r = ae + bg$$

$$s = af + bh$$

$$t = ce + dg$$

$$u = cf + dh$$

recursive

8 mults of $(n/2) \times (n/2)$ submatrices

4 adds of $(n/2) \times (n/2)$ submatrices

Analysis of D&C algorithm

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

submatrices *submatrix size* *work adding submatrices*

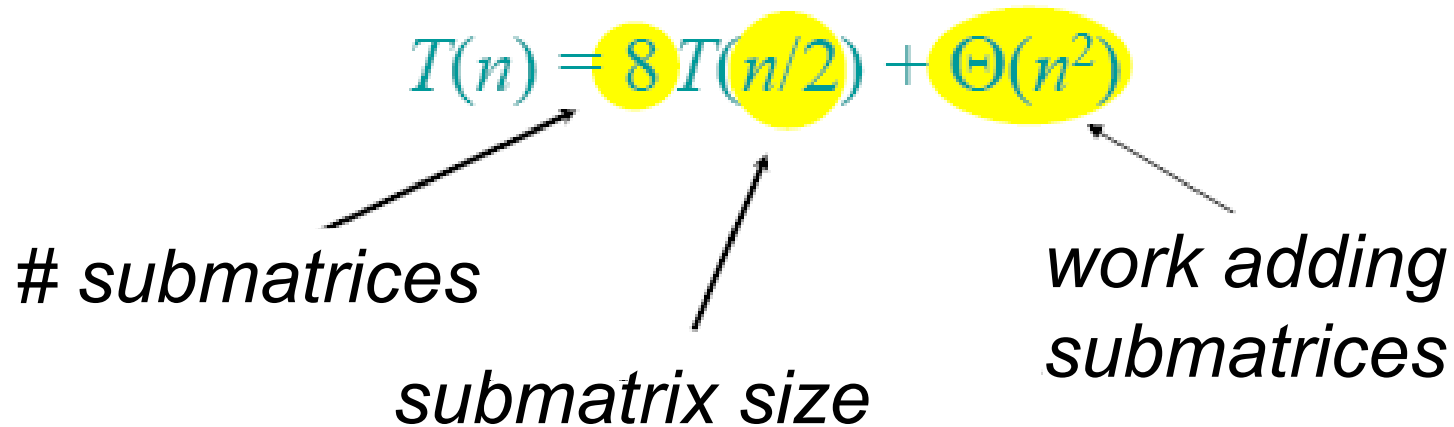
Analysis of D&C algorithm

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

submatrices

submatrix size

work adding submatrices



$$n \log b a = n \log_2 8 = n^3 \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^3).$$

Analysis of D&C algorithm

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

submatrices submatrix size work adding submatrices

$$n \log_b a = n \log_2 8 = n \cdot 3 \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^3).$$

No better than the ordinary algorithm.

Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$P1 = a \cdot (f - h)$$

$$P2 = (a + b) \cdot h$$

$$P3 = (c + d) \cdot e$$

$$P4 = d \cdot (g - e)$$

$$P5 = (a + d) \cdot (e + h)$$

$$P6 = (b - d) \cdot (g + h)$$

$$P7 = (a - c) \cdot (e + f)$$

Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$P1 = a \cdot (f - h)$$

$$P2 = (a + b) \cdot h$$

$$P3 = (c + d) \cdot e$$

$$P4 = d \cdot (g - e)$$

$$P5 = (a + d) \cdot (e + h)$$

$$P6 = (b - d) \cdot (g + h)$$

$$P7 = (a - c) \cdot (e + f)$$

$$r = P5 + P4 - P2 + P6$$

$$s = P1 + P2$$

$$t = P3 + P4$$

$$u = P5 + P1 - P3 - P7$$

Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$P1 = a \cdot (f - h)$$

$$P2 = (a + b) \cdot h$$

$$P3 = (c + d) \cdot e$$

$$P4 = d \cdot (g - e)$$

$$P5 = (a + d) \cdot (e + h)$$

$$P6 = (b - d) \cdot (g + h)$$

$$P7 = (a - c) \cdot (e + f)$$

$$r = P5 + P4 - P2 + P6$$

$$s = P1 + P2$$

$$t = P3 + P4$$

$$u = P5 + P1 - P3 - P7$$

7 mults, 18 adds/subs.

Note: No reliance on commutativity of mult!

Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$P1 = a \cdot (f - h)$$

$$P2 = (a + b) \cdot h$$

$$P3 = (c + d) \cdot e$$

$$P4 = d \cdot (g - e)$$

$$P5 = (a + d) \cdot (e + h)$$

$$P6 = (b - d) \cdot (g + h)$$

$$P7 = (a - c) \cdot (e + f)$$

$$r = P5 + P4 - P2 + P6$$

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

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

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

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

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

$$+ bg + bh - dg - dh$$

$$= ae + bg$$

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.

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.

$$T(n) = 7 T(n/2) + \Theta(n^2)$$

Analysis of Strassen

$$T(n) = 7 T(n/2) + \Theta(n^2)$$

Analysis of Strassen

$$T(n) = 7 T(n/2) + \Theta(n^2)$$

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

Analysis of Strassen

$$T(n) = 7 T(n/2) + \Theta(n^2)$$

$$n \log b a = n \log_2 7 \approx n^{2.81} \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^{\lg 7}).$$

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

$$T(n) = 7 T(n/2) + \Theta(n^2)$$

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

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.

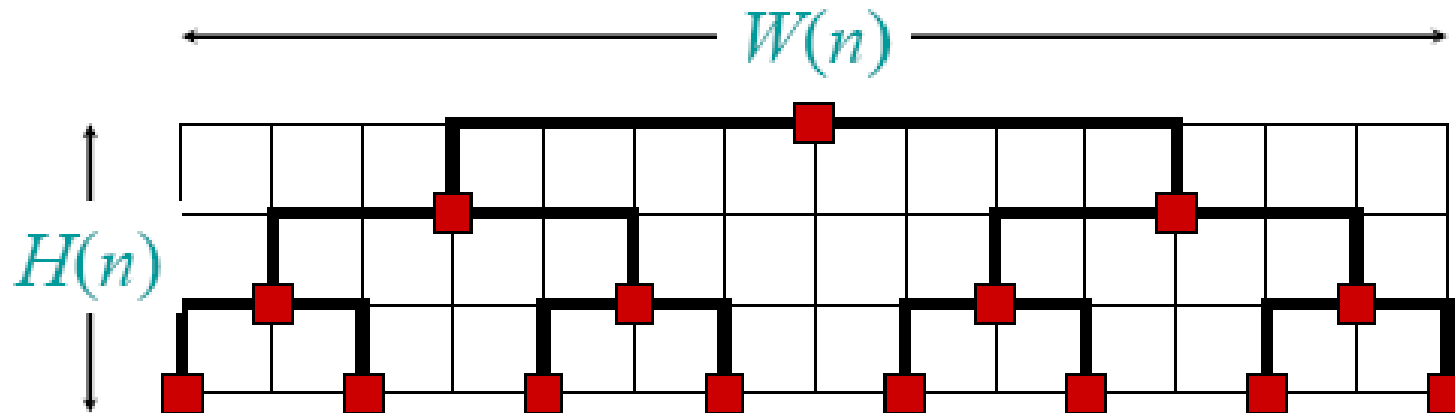
Best to date (of theoretical interest only): $\Theta(n^{2.376\dots})$.

VLSI layout

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

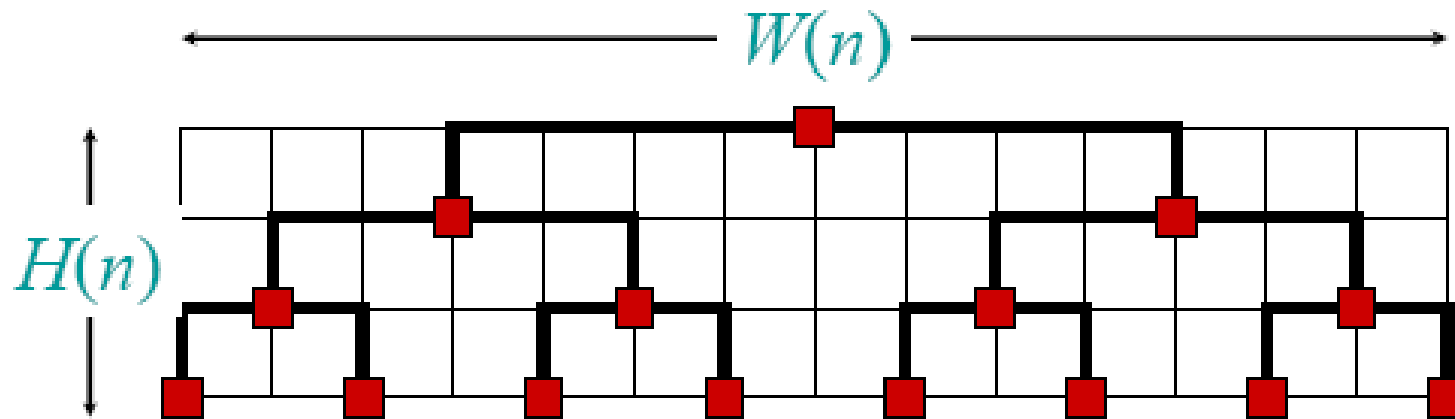
VLSI layout

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



VLSI layout

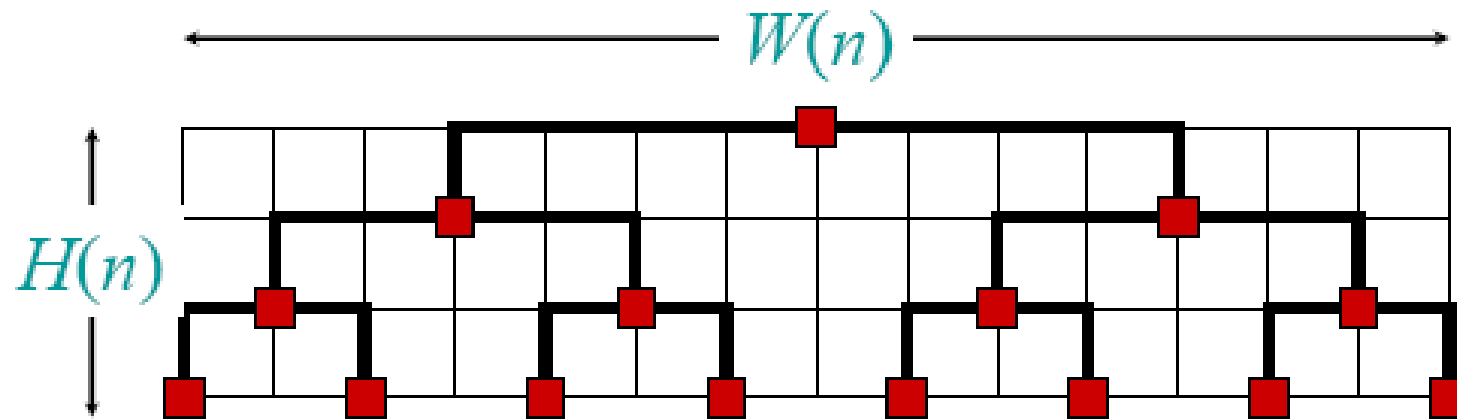
Problem: Embed a complete binary tree with n leaves in a grid using minimal area.



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

VLSI layout

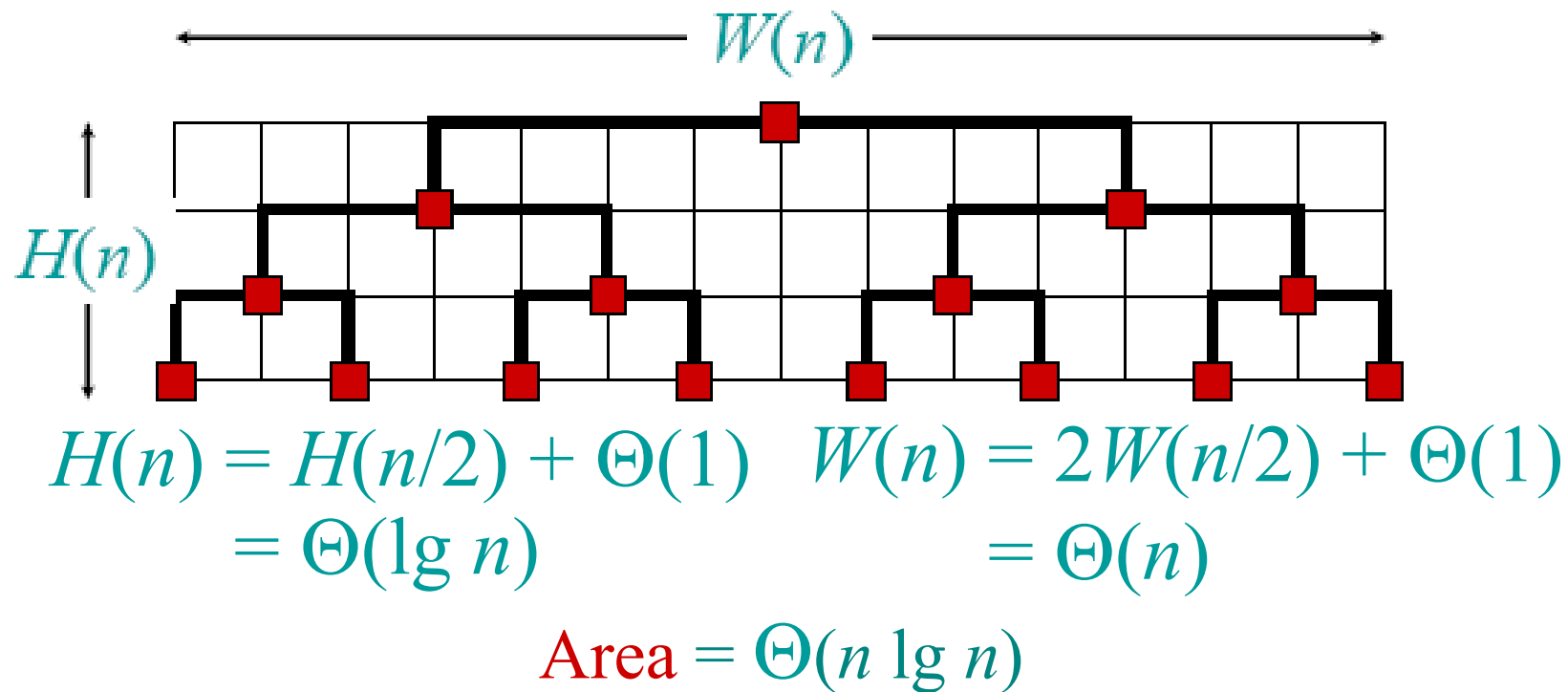
Problem: Embed a complete binary tree with n leaves in a grid using minimal area.



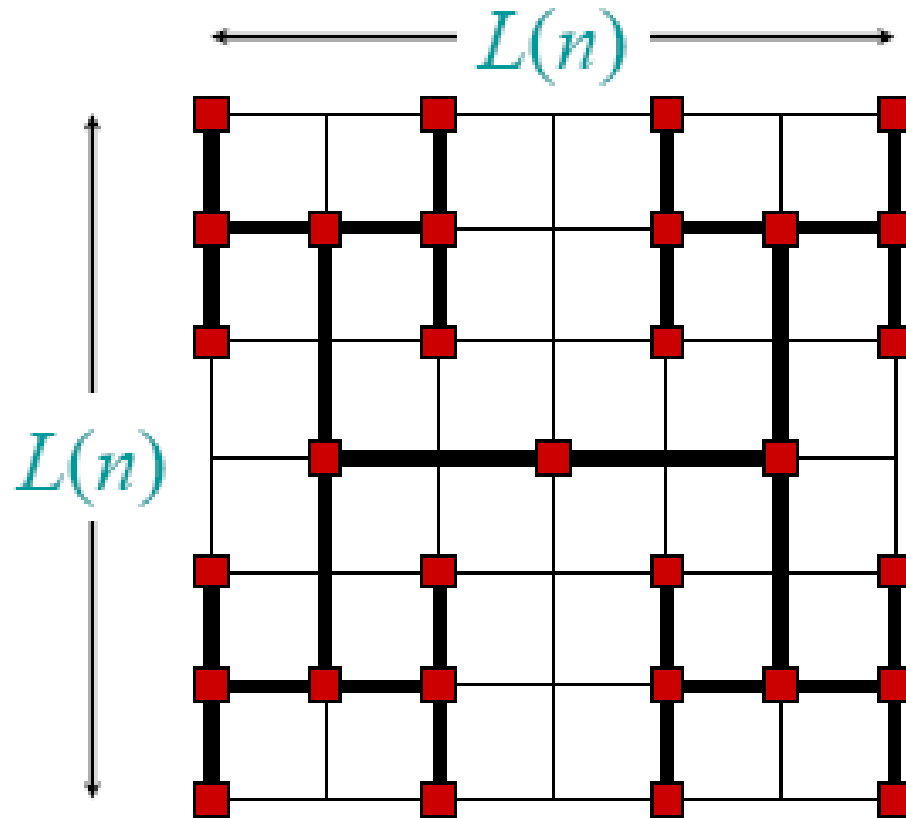
$$\begin{aligned} H(n) &= H(n/2) + \Theta(1) & W(n) &= 2W(n/2) + \Theta(1) \\ &= \Theta(\lg n) & &= \Theta(n) \end{aligned}$$

VLSI layout

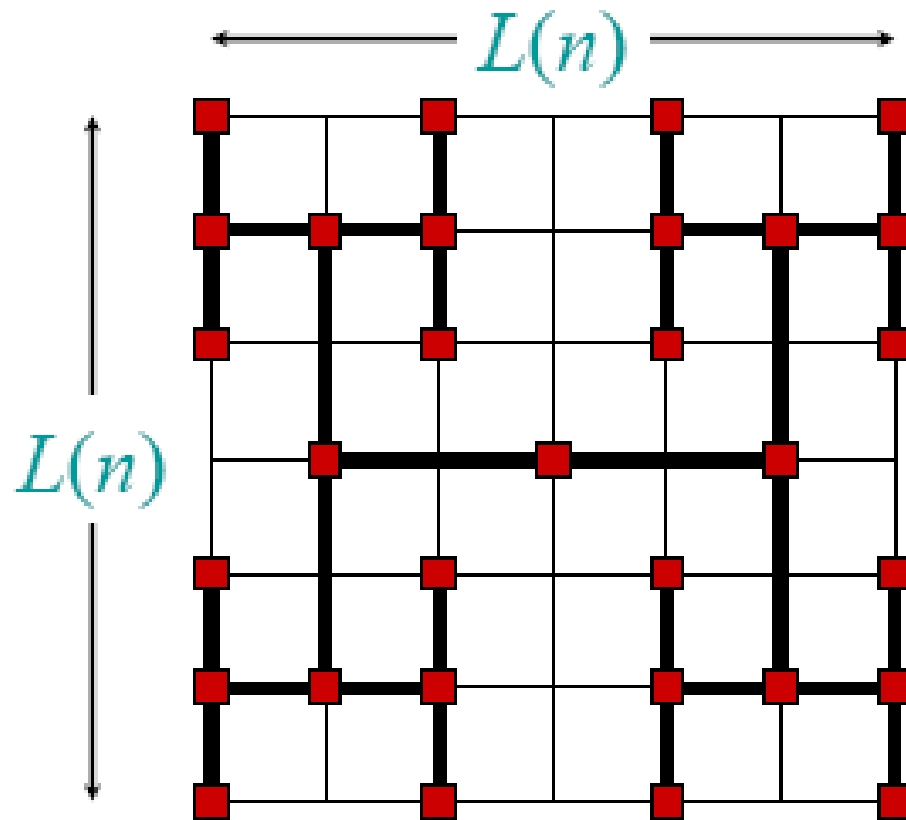
Problem: Embed a complete binary tree with n leaves in a grid using minimal area.



H-tree embedding

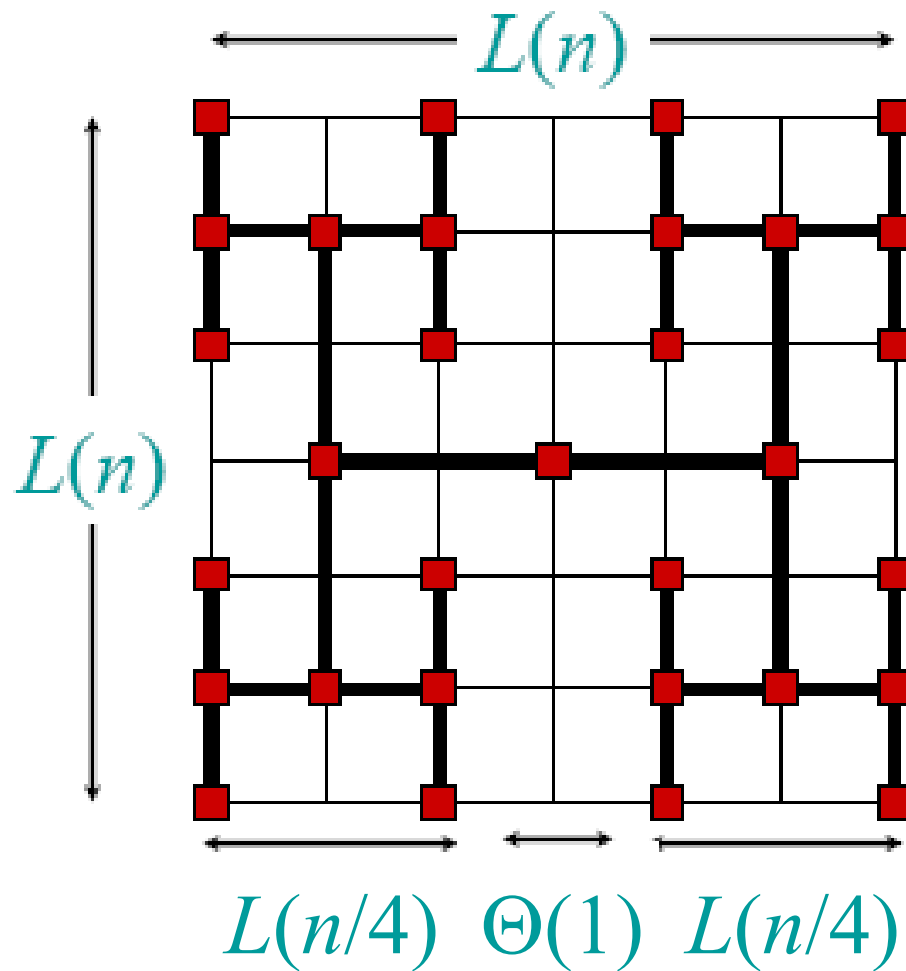


H-tree embedding



$$L(n/4) \Theta(1) L(n/4)$$

H-tree embedding



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

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

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