CENG 218
Design and Analysis of Algorithms

Izmir Institute of Technology

Lecture 5: Analysis of divide-and-conquer algorithms

Slides were mostly prepared using the material provided by Prof. Charles E. Leiserson and Prof. Erik Demaine from MIT

The divide-and-conquer design paradigm

Remember divide-and-conquer approach:

1. **Divide** the problem into several subproblems.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** the solutions of subproblems.

Master method (refresher)

$$T(n) = a\,T(n/b) + f(n)$$

CASE 1: $f(n) = O(n^{\log_b a - \varepsilon})$

$\Rightarrow T(n) = \Theta(n^{\log_b a})$.

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$

$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$

$\Rightarrow T(n) = \Theta(f(n))$.

Master method for Merge sort

1. **Divide** the list of length n into 2 (constant time).
2. **Conquer:** Recursively sort two sublists, each of size $\leq \lceil n/2 \rceil$.
3. **Combine:** Merge sublists in linear time.

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

 # subproblems

subproblem size

work for dividing and combining

Master method for Merge sort

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

a=2, b=2 $\Rightarrow n^{\log_b a} = n$; $f(n) = \Theta(n)$.

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n) = \Theta(n \lg^0 n)$, ($k$ = 0).

$$\therefore\ T(n) = \Theta(n \lg n).$$

Powering a number

Problem: Compute a^n.

Naive algorithm: $a^n = a^{n-1} \cdot a \quad \rightarrow \quad \Theta(n)$.

Divide-and-conquer algorithm:

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

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

$a=1,\ b=2 \Rightarrow n^{\log_b a} = 1\ ;\ f(n) = 1.$

CASE 2: $f(n) = \Theta(\lg^0 n)$, $\quad T(n) = \Theta(\lg n)$

Computing Fibonacci numbers

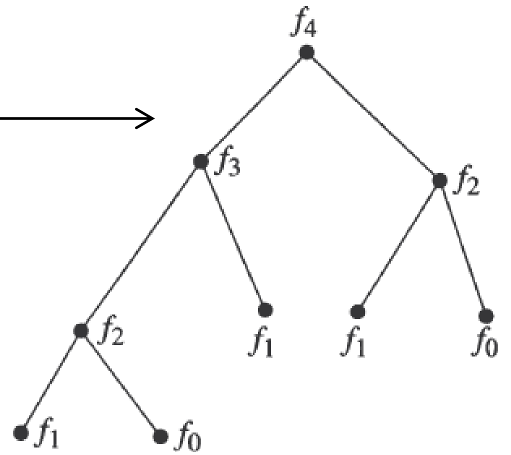
$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n\text{-}1}+F_{n\text{-}2} & \text{if } n \geq 2 \end{cases}$$

0 1 1 2 3 5 8 13 21 34 …

1) Naive recursive algorithm $\longrightarrow$

is $\Omega(\varphi^n)$ where $\varphi=(1+\sqrt{5})/2$

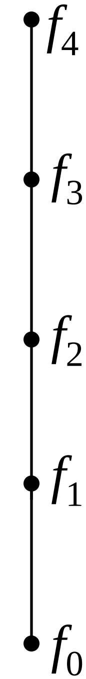
i.e. exponential time.



Computing Fibonacci numbers

2) Bottom-up (iterative) approach:

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



Computing Fibonacci numbers

3) 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's running time = $\Theta(\lg n)$. How?

By divide-and-conquer powering, i.e. $a^n = a^{n/2} \cdot a^{n/2}$

Computing Fibonacci numbers

Proof of theorem: By mathematical induction

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

Inductive step ($n \geq 2$):

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

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

By inductive hypothesis P(n-1)

Matrix multiplication

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

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

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

Standard algorithm

MATRIX-MULTIPLICATION (A, B: matrices)

for $i \leftarrow 1$ **to** n

  for $j \leftarrow 1$ **to** n

    $c_{ij} \leftarrow 0$

    for $k \leftarrow 1$ **to** n

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

return C {$C = [c_{ij}]$ is the product of A and B}

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

Divide-and-conquer algorithm

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

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

$$C \quad = \quad A \quad \cdot \quad B$$

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

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

$\uparrow$
What is the recurrence here?

Divide-and-conquer algorithm

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

 # submatrix multip.

submatrix size

work for adding submatrices

$n^{\log_b a} = n^{\log_2 8} = n^3$

CASE 1: $f(n) = O(n^{\log_b a - \varepsilon})$

$T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$

No better than the standard algorithm!

Strassen's algorithm

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

Multiply 2 × 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 multiplications,
18 adds/subs.

Strassen's algorithm

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

 # submatrices *submatrix size* *work for adding submatrices*

$n^{\log_b a} = n^{\log_2 7} = n^{2.81}$

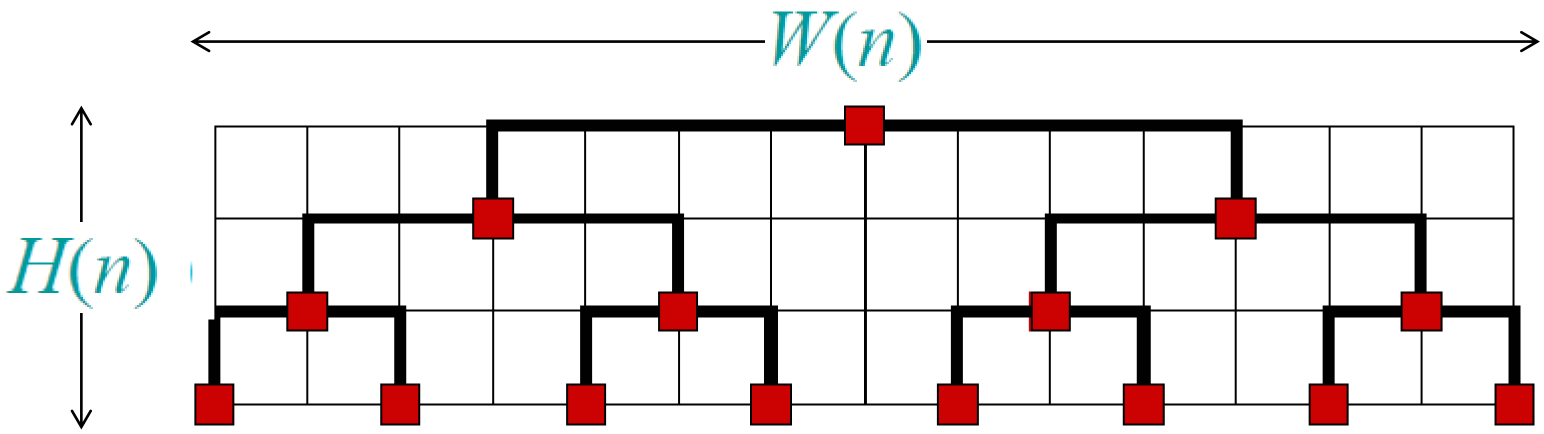
CASE 1: $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{2.81})$

Note: 2.81 may not seem much smaller than 3, but the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for $n \geq 30$ or so.

Another example: VLSI layout

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

Naive embedding:



What is the recurrence for $H(n)$?

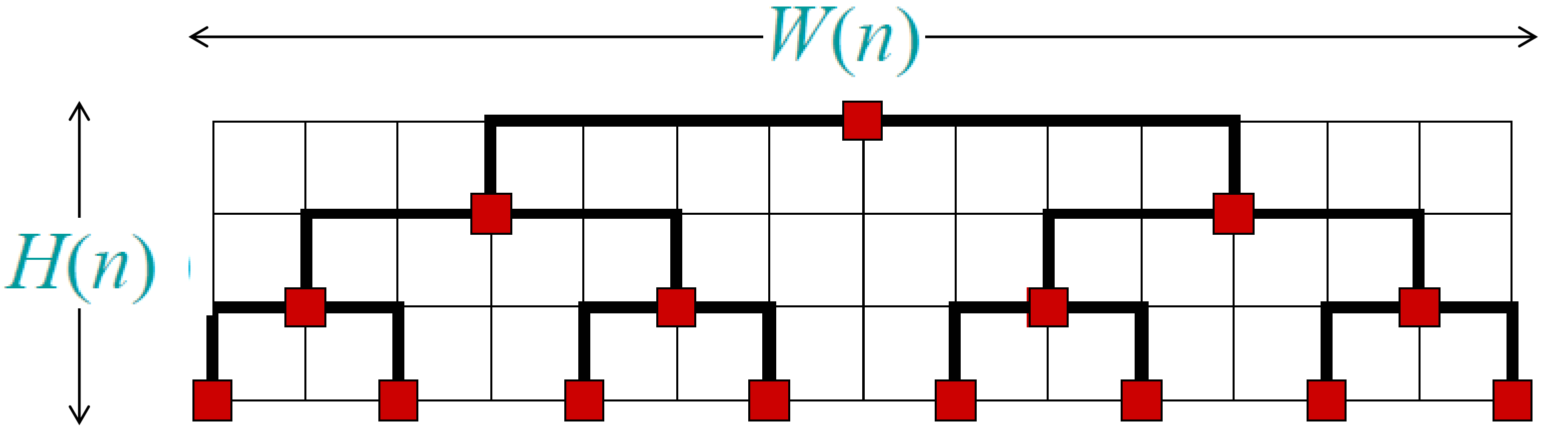
$H(n) = H(n/2) + \Theta(1)$

$\quad\quad = \Theta(\lg n)$

$H(1)=1$
$H(3)=2$
$H(7)=3$
$H(15)=4$

Another example: VLSI layout

Naive embedding:



What is the recurrence for $H(n)$?

$$H(n) = H(n/2) + \Theta(1)$$
$$= \Theta(\lg n)$$

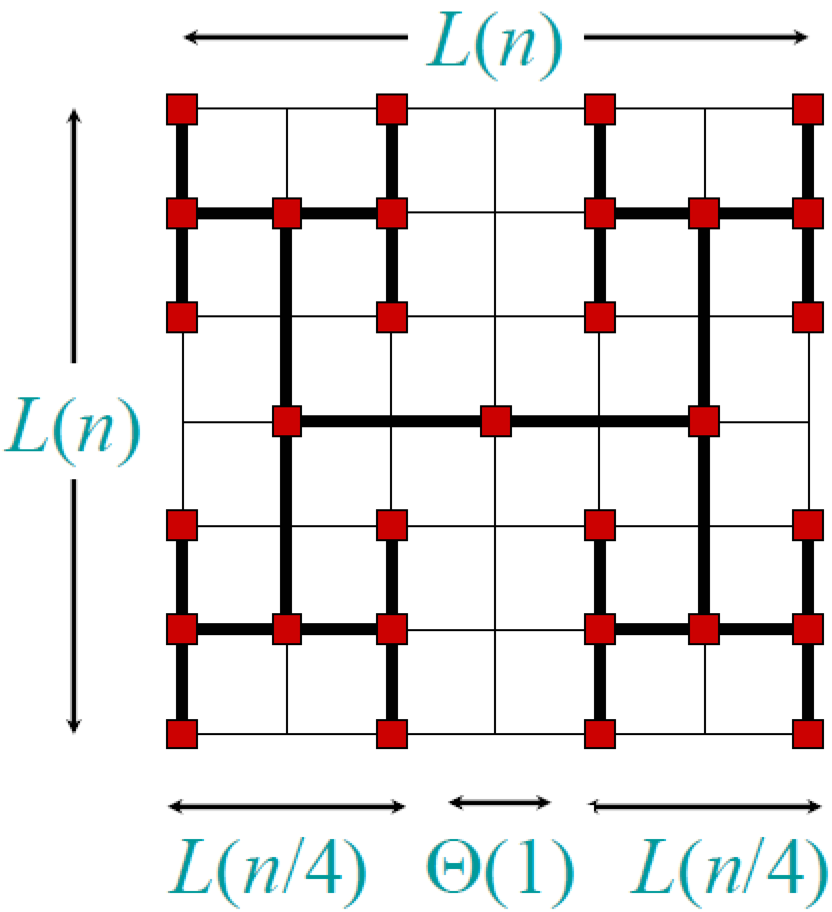
What is the recurrence for $W(n)$?

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

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

Another example: VLSI layout

H-Tree embedding:



$$L(n) = 2L(n/4) + \Theta(1)$$

$$L(n) = \Theta(\sqrt{n})$$

$$W(n) = \Theta(\sqrt{n})$$

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