What is an Algorithm?

(And how do we analyze one?)

Algorithms

- ***Informally***,
  - A tool for solving a well-specified computational problem.

- **Example: sorting**

  input: A sequence of numbers.

  output: An ordered permutation of the input.

  issues: correctness, efficiency, storage, etc.

Strengthening the Informal Definiton

- An algorithm is a **finite** sequence of **unambiguous** instructions for solving a well-specified computational problem.
- Important Features:
  - Finiteness.
  - Definiteness.
  - Input.
  - Output.
  - Effectiveness.

Algorithm Analysis

- Determining performance characteristics. (Predicting the resource requirements.)
  - Time, memory, communication bandwidth etc.
  - **Computation time** (running time) is of primary concern.
- Why analyze algorithms?
  - **Choose** the **most efficient** of several possible algorithms for the same problem.
  - Is the best possible **running time** for a problem ***reasonably finite*** for practical purposes?
  - Is the algorithm **optimal** (best in some sense)? – Is something better possible?

Running Time

- Run time expression should be machine-independent.
  - Use a model of computation or “hypothetical” computer.
  - Our choice – **RAM model** (most commonly-used).
- Model should be
  - Simple.
  - Applicable.

RAM Model

- Generic single-processor model.
- **Supports simple constant-time instructions** found in real computers.
  - Arithmetic (+, –, *, /, %, floor, ceiling).
  - Data Movement (load, store, copy).
  - Control (branch, subroutine call).
- Run time (**cost**) is uniform (**1 time unit**) for all simple instructions.
- Memory is unlimited.
- Flat memory model – no hierarchy.
- Access to a word of memory takes **1 time unit**.
- Sequential execution – **no concurrent operations**.

Running Time – Definition

- Call each simple instruction and access to a word of memory a "primitive operation" or "step."
- Running time of an algorithm for a given input is
  - The **number of steps** executed by the algorithm on that **input**.
- Often referred to as the ***complexity*** of the algorithm.

Complexity and Input

- Complexity of an algorithm generally depends on
  - **Size of input**.
    - Input size depends on the problem.
      - Examples: No. of items to be sorted.
      - No. of vertices and edges in a graph.
  - **Other characteristics of the input data**.
    - Are the items already sorted?
    - Are there cycles in the graph?

Worst, Average, and Best-case Complexity

- Worst-case Complexity
  - **Maximum** steps the algorithm takes for any possible input.
  - Most tractable measure.
- Average-case Complexity
  - **Average** of the running times of all ***possible* inputs.**
  - Demands a definition of probability of each input, which is usually difficult to provide and to analyze.
- Best-case Complexity
  - **Minimum** number of steps for any possible input.
  - Not a useful measure. Why?

A Simple Example – *Linear Search*

INPUT: a sequence of *n* numbers, *key* to search for.

OUTPUT: *true* if *key* occurs in the sequence, *false* otherwise.

LinearSearch(A, *key*)	*cost*	*times*
$i \leftarrow 1$	c_1	1
while $i \leq n$ **and** A[i] != *key*	c_2	x
do i++	c_3	x-1
if $i \leq n$	c_4	1
then return *true*	c_5	1
else return *false*	c_6	1

x ranges between 1 and n+1.

So, the running time ranges between

$c_1 + c_2 + c_4 + c_5$ – **best case**

and

$c_1 + c_2(n+1) + c_3 n + c_4 + c_6$ – **worst case**

A Simple Example – *Linear Search*

INPUT: a sequence of *n* numbers, *key* to search for.

OUTPUT: *true* if *key* occurs in the sequence, *false* otherwise.

LinearSearch(A, *key*)	*cost*	*times*
$i \leftarrow 1$	1	1
while $i \le n$ **and** A[i] != *key*	1	x
do i++	1	x-1
if $i \le n$	1	1
then return *true*	1	1
else return *false*	1	1

Assign a cost of 1 to all statement executions.

Now, the running time ranges between

$1+ 1+ 1 + 1 = 4$ – **best case**

and

$1+ (n+1)+ n + 1 + 1 = 2n+4$ – **worst case**

A Simple Example – *Linear Search*

INPUT: a sequence of *n* numbers, *key* to search for.

OUTPUT: *true* if *key* occurs in the sequence, *false* otherwise.

LinearSearch(A, *key*)	*cost*	*times*
1 $i \leftarrow 1$	1	1
2 **while** $i \leq n$ **and** A[*i*] != *key*	1	x
3 **do** *i*++	1	x-1
4 **if** $i \leq n$	1	1
5 **then return** *true*	1	1
6 **else return** *false*	1	1

If we assume that we search for a random item in the list, on an average, Statements 2 and 3 will be executed *n*/2 times. Running times of other statements are independent of input. Hence, **average-case complexity** is

$$1 + n/2 + n/2 + 1 + 1 = n+3$$

Order of growth

- Principal interest is to determine
  - how running time grows with input size – **Order of growth**.
  - the running time for large inputs – **Asymptotic complexity**.
- In determining the above,
  - **Lower-order terms and coefficient of the highest-order term are insignificant.**
  - **Ex: In $7n^5+6n^3+n+10$, which term dominates the running time for very large n?**
- Complexity of an algorithm is denoted by the highest-order term in the expression for running time.
  - **Ex:** $O(n)$, $\Theta(1)$, $\Omega(n^2)$, etc.
  - Constant complexity when running time is independent of the input size – denoted $O(1)$.
  - **Linear Search**: **Best case Θ(1), Worst and Average cases: Θ(n).**
- More on O, Θ, and Ω in next class. Use Θ for the present.

Comparison of Algorithms

- Complexity function can be used to compare the performance of algorithms.
- Algorithm *A* is more efficient than Algorithm *B* for solving a problem, if the complexity function of *A* is of lower order than that of *B*.
- Examples:
  - **Linear Search** – $\Theta(n)$ vs. **Binary Search** – $\Theta(\lg n)$
  - **Insertion Sort** – $\Theta(n^2)$ vs. **Quick Sort** – $\Theta(n \lg n)$

Asymptotic Notation, Review of Functions & Summations

Asymptotic Complexity

- Running time of an algorithm as a function of input size *n* **for large *n***.
- Expressed using only the **highest-order term** in the expression for the exact running time.
  - Instead of exact running time, say $\Theta(n^2)$.
- Describes behavior of function in the limit.
- Written using ***Asymptotic Notation***.

Asymptotic Notation

- **Θ, *O*, Ω, *o*, ω**
- Defined for functions over the natural numbers.
  - **Ex:** $f(n) = \Theta(n^2)$.
  - Describes how $f(n)$ grows in comparison to n^2.
- Define a ***set*** of functions; in practice used to compare two function sizes.
- The notations describe different rate-of-growth relations between the defining function and the defined set of functions.

Θ-notation

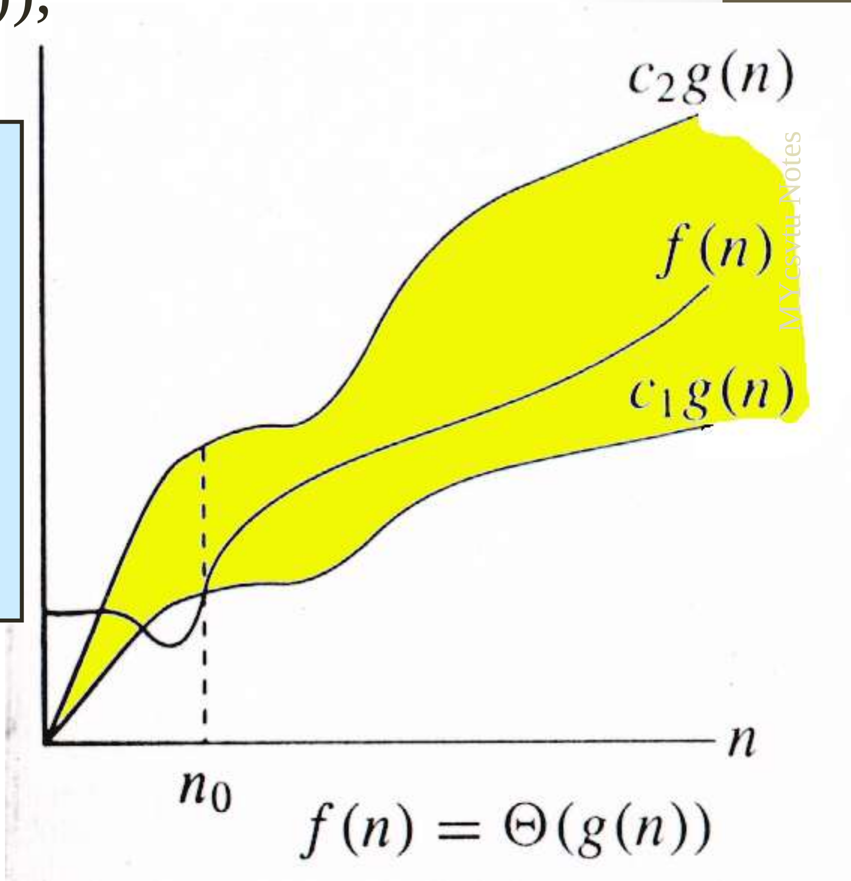
For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{f(n)$:

∃ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$,

we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

}



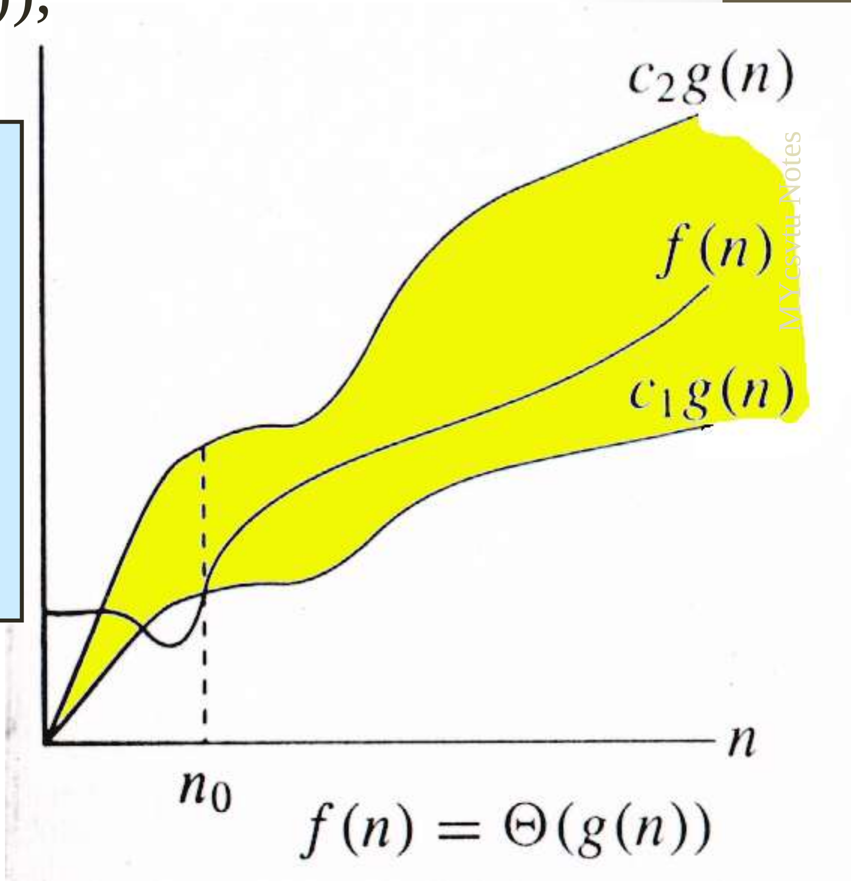
Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.

$g(n)$ is an *asymptotically tight bound* for $f(n)$.

Θ-notation

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{f(n)$:
$\exists$ positive constants c_1, c_2, and n_0,
such that $\forall n \geq n_0$,
we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
$\}$



Technically, $f(n) \in \Theta(g(n))$.
Older usage, $f(n) = \Theta(g(n))$.
I'll accept either…

$f(n)$ and $g(n)$ are nonnegative, for large n.

Example

$\Theta(g(n)) = \{f(n) :$ $\exists$ **positive constants** $c_1, c_2,$ **and** $n_0,$ **such that** $\forall n \geq n_0, \quad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

- $10n^2 - 3n = \Theta(n^2)$
- What constants for n_0, c_1, and c_2 will work?
- Make c_1 a little smaller than the leading coefficient, and c_2 a little bigger.
- ***To compare orders of growth, look at the leading term.***
- Exercise: Prove that $n^2/2-3n= \Theta(n^2)$

Example

$\Theta(g(n)) = \{f(n) :$ **$\exists$ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$,** $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

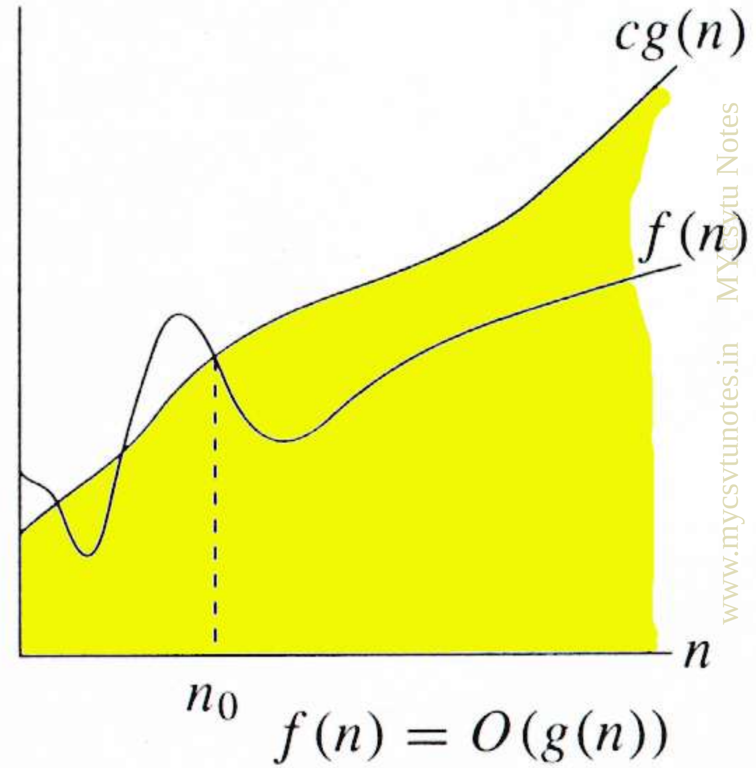
- Is $3n^3 \in \Theta(n^4)$??
- How about $2^{2n} \in \Theta(2^n)$??

O-notation

For function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$O(g(n)) = \{f(n)$:
$\exists$ **positive constants** c **and** n_0,
such that $\forall n \geq n_0$,
we have $0 \leq f(n) \leq cg(n)$ $\}$

Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.



$g(n)$ is an *asymptotic upper bound* for $f(n)$.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$.
$\Theta(g(n)) \subset O(g(n))$.

Examples

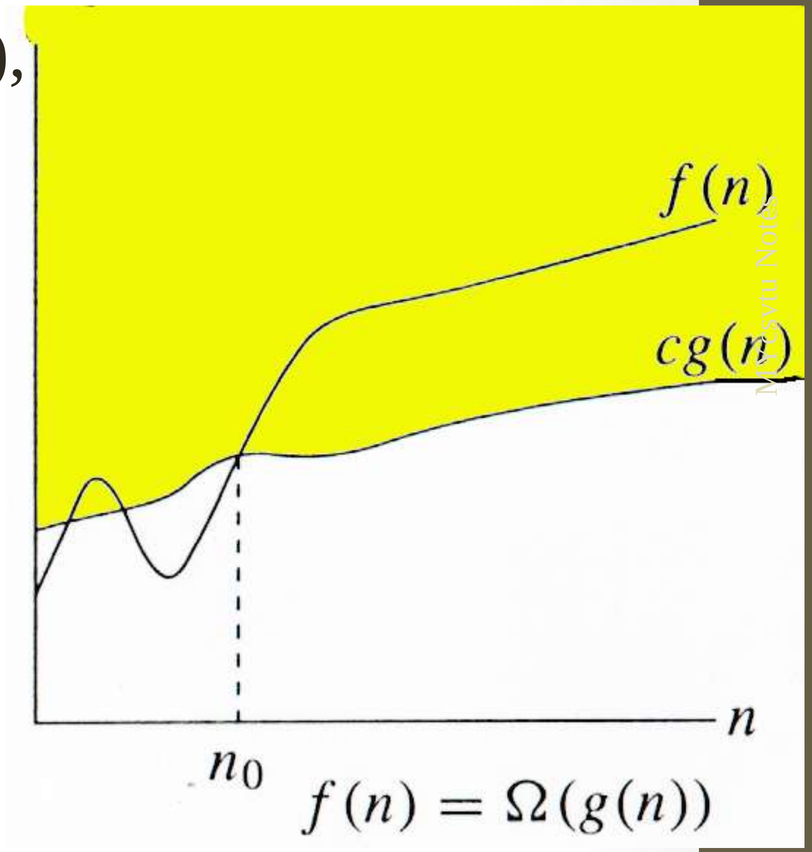
$O(g(n)) = \{f(n) : \exists$ **positive constants** c **and** n_0, **such that** $\forall n \geq n_0$, **we have** $0 \leq f(n) \leq cg(n)\ \}$

- Any linear *function an* + *b* is in $O(n^2)$.
- Show that $3n^3 = O(n^4)$ for appropriate c and n_0.

Ω -notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

$\Omega(g(n)) = \{f(n) :$
$\exists$ **positive constants** c **and** n_0**,**
such that $\forall n \geq n_0$**,**
we have $0 \leq cg(n) \leq f(n)\}$



Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

$g(n)$ **is an *asymptotic lower bound* for** $f(n)$.

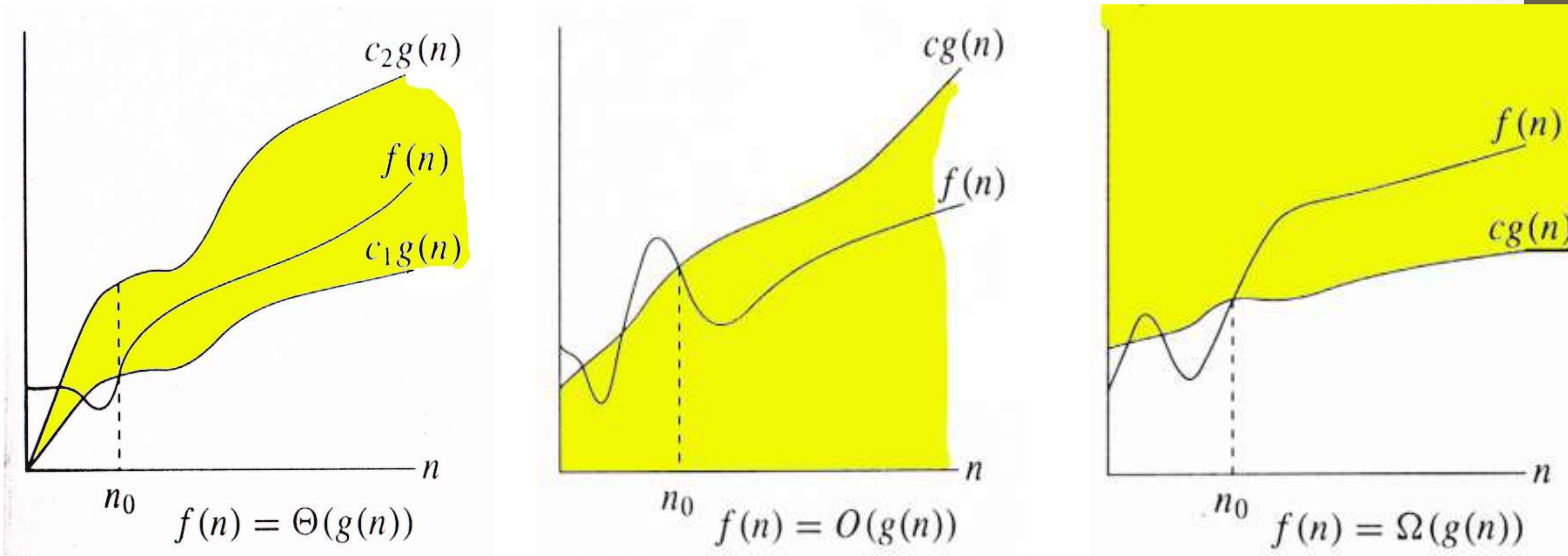
$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$.
$\Theta(g(n)) \subset \Omega(g(n))$.

Example

$\Omega(g(n)) = \{f(n)$: $\exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq cg(n) \leq f(n)\}$

- $\sqrt{n} = \Omega(\lg n)$. Choose c and n_0.

Relations Between Θ, *O*, Ω



Relations Between Θ, Ω, *O*

Theorem : For any two functions $g(n)$ and $f(n)$,

$f(n) = \Theta(g(n))$ iff

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

- I.e., $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
- In practice, asymptotically tight bounds are obtained from asymptotic upper and lower bounds.

Running Times

- "Running time is $O(f(n))$" $\Rightarrow$ Worst case is $O(f(n))$
- $O(f(n))$ bound on the worst-case running time $\Rightarrow$ $O(f(n))$ bound on the running time of every input.
- $\Theta(f(n))$ bound on the worst-case running time $\not\Rightarrow$ $\Theta(f(n))$ bound on the running time of every input.
- "Running time is $\Omega(f(n))$" $\Rightarrow$ Best case is $\Omega(f(n))$
- Can still say "Worst-case running time is $\Omega(f(n))$"
  - Means worst-case running time is given by some unspecified function $g(n) \in \Omega(f(n))$.

Example

- ***Insertion sort*** takes $\Theta(n^2)$ in the worst case, so sorting (as a *problem*) is $O(n^2)$.
- Any sort algorithm must look at each item, so sorting is $\Omega(n)$.
- In fact, using (e.g.) merge sort, sorting is $\Theta(n \lg n)$ in the worst case.
  - Later, we will prove that we cannot hope that any comparison sort to do better in the worst case.

Asymptotic Notation in Equations

- Can use asymptotic notation in equations to replace expressions containing lower-order terms.
- For example,

  $4n^3 + 3n^2 + 2n + 1 = 4n^3 + 3n^2 + \Theta(n)$

  $= 4n^3 + \Theta(n^2) = \Theta(n^3)$.
- In equations, $\Theta(f(n))$ always stands for an $g(n) \in \Theta(f(n))$
  - In the example above, $\Theta(n^2)$ stands for $3n^2 + 2n + 1$.

o-notation

For a given function $g(n)$, the set little-o:

$$o(g(n)) = \{f(n): \forall\, c > 0, \exists\, n_0 > 0 \text{ such that } \forall\, n \geq n_0, \text{ we have } 0 \leq f(n) < cg(n)\}.$$

$f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity:

$$\lim_{n \to \infty} [f(n) / g(n)] = 0$$

$g(n)$ is an ***upper bound*** for $f(n)$ that is not asymptotically tight.

Observe the difference in this definition from previous ones.

ω-notation

For a given function $g(n)$, the set little-omega:

$$\omega(g(n)) = \{f(n): \forall\ c > 0, \exists\ n_0 > 0 \text{ such that } \forall\ n \geq n_0, \text{ we have } 0 \leq cg(n) < f(n)\}.$$

$f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity:

$$\lim_{n \to \infty} [f(n) / g(n)] = \infty.$$

$g(n)$ is a ***lower bound*** for $f(n)$ that is not asymptotically tight.

Comparison of Functions

$$f \leftrightarrow g \approx a \leftrightarrow b$$

$$f(n) = O(g(n)) \approx a \leq b$$

$$f(n) = \Omega(g(n)) \approx a \geq b$$

$$f(n) = \Theta(g(n)) \approx a = b$$

$$f(n) = o(g(n)) \approx a < b$$

$$f(n) = \omega(g(n)) \approx a > b$$

Limits

- $\lim_{n\to\infty} [f(n) / g(n)] = 0 \Rightarrow f(n) \in o(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)] = \infty \Rightarrow f(n) \in \omega(g(n))$

Properties

- **Transitivity**

$$f(n) = \Theta(g(n)) \;\&\; g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \;\&\; g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \;\&\; g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$
$$f(n) = o\,(g(n)) \;\&\; g(n) = o\,(h(n)) \Rightarrow f(n) = o\,(h(n))$$
$$f(n) = \omega(g(n)) \;\&\; g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$$

- **Reflexivity**

$$f(n) = \Theta(f(n))$$
$$f(n) = O(f(n))$$
$$f(n) = \Omega(f(n))$$

Properties

- **Symmetry**

  $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$

- **Complementarity**

  $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

  $f(n) = o(g(n))$ iff $g(n) = \omega((f(n))$

Common Functions

Monotonicity

- $f(n)$ is
  - **monotonically increasing** if $m \leq n \Rightarrow f(m) \leq f(n)$.
  - **monotonically decreasing** if $m \geq n \Rightarrow f(m) \geq f(n)$.
  - **strictly increasing** if $m < n \Rightarrow f(m) < f(n)$.
  - **strictly decreasing** if $m > n \Rightarrow f(m) > f(n)$.

Exponentials

- **Useful Identities:**

$$a^{-1} = \frac{1}{a}$$

$$(a^m)^n = a^{mn}$$

$$a^m a^n = a^{m+n}$$

- **Exponentials and polynomials**

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0$$

$$\Rightarrow n^b = o(a^n)$$

Logarithms

$x = \log_b a$ is the exponent for $a = b^x$.

Natural log: $\ln a = \log_e a$

Binary log: $\lg a = \log_2 a$

$\lg^2 a = (\lg a)^2$

$\lg \lg a = \lg (\lg a)$

$$a = b^{\log_b a}$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b(1/a) = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

Polylogarithms

- **For $a \geq 0$, $b > 0$,** $\lim_{n \to \infty} (\lg^a n / n^b) = 0$, so $\lg^a n = o(n^b)$, and $n^b = \omega(\lg^a n)$
  - Prove using L'Hopital's rule repeatedly
- $\lg(n!) = \Theta(n \lg n)$
  - Prove using Stirling's approximation (in the text) for $\lg(n!)$.

Exercise

Express functions in A in asymptotic notation using functions in B.

A	B	
$5n^2 + 100n$	$3n^2 + 2$	$A \in \Theta(B)$

$A \in \Theta(n^2), n^2 \in \Theta(B) \Rightarrow A \in \Theta(B)$

$\log_3(n^2)$	$\log_2(n^3)$	$A \in \Theta(B)$

$\log_b a = \log_c a / \log_c b$; A = 2lg$n$ / lg3, B = 3lg$n$, A/B =2/(3lg3)

$n^{\lg 4}$	$3^{\lg n}$	$A \in \omega(B)$

$a^{\log b} = b^{\log a}$; B $=3^{\lg n}=n^{\lg 3}$; A/B $=n^{\lg(4/3)} \rightarrow \infty$ as $n \rightarrow \infty$

$\lg^2 n$	$n^{1/2}$	$A \in o(B)$

$\lim_{n\rightarrow\infty} (\lg^a n / n^b) = 0$ (here $a = 2$ and $b = 1/2$) $\Rightarrow A \in o(B)$

Recurrences

The Master Method

- Based on the Master theorem.
- "Cookbook" approach for solving recurrences of the form

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

  - $a \geq 1$, $b > 1$ are constants.
  - $f(n)$ is asymptotically positive.
  - n/b may not be an integer, but we ignore floors and ceilings. Why?
- Requires memorization of three cases.

The Master Theorem

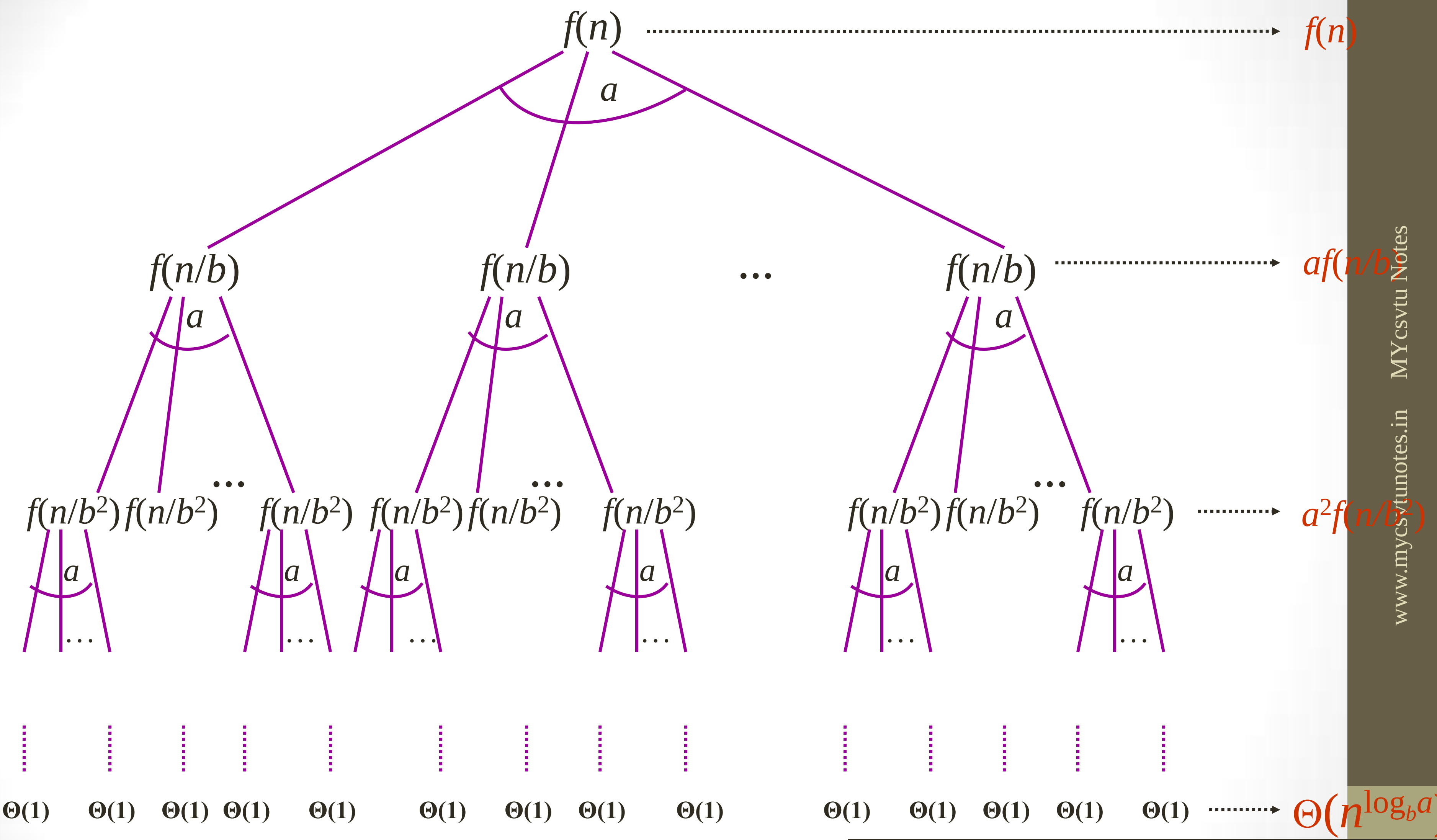
Theorem 4.1

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and

Let $T(n)$ be defined on nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where we can replace n/b by $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. $T(n)$ can be bounded asymptotically in three cases:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if, for some constant $c < 1$ and all sufficiently large n, we have $a \cdot f(n/b) \leq c\, f(n)$, then $T(n) = \Theta(f(n))$.

Recursion tree view



Total: $T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$

The Master Theorem

Theorem 4.1

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and

Let $T(n)$ be defined on nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where we can replace n/b by $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. $T(n)$ can be bounded asymptotically in three cases:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if, for some constant $c < 1$ and all sufficiently large n, we have $a \cdot f(n/b) \leq c\, f(n)$, then $T(n) = \Theta(f(n))$.

Master Method – Examples

- $T(n) = 16T(n/4)+n$
  - $a = 16$, $b = 4$, $n^{\log_b a} = n^{\log_4 16} = n^2$.
  - $f(n) = n = O(n^{\log_b a-\varepsilon}) = O(n^{2-\varepsilon})$, where $\varepsilon = 1 \Rightarrow$ **Case 1.**
  - Hence, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

- $T(n) = T(3n/7) + 1$
  - $a = 1$, $b=7/3$, and $n^{\log_b a} = n^{\log_{7/3} 1} = n^0 = 1$
  - $f(n) = 1 = \Theta(n^{\log_b a}) \Rightarrow$ **Case 2**.
  - Therefore, $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(\lg n)$

Master Method – Examples

- $T(n) = 3T(n/4) + n \lg n$
  - $a = 3$, $b=4$, thus $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$
  - $f(n) = n \lg n = \Omega(n^{\log_4 3 + \varepsilon})$ where $\varepsilon \approx 0.2 \Rightarrow$ **Case 3**.
  - Therefore, $T(n) = \Theta(f(n)) = \Theta(n \lg n)$.

- $T(n) = 2T(n/2) + n \lg n$
  - $a = 2$, $b=2$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_2 2} = n$
  - **$f(n)$** is asymptotically larger than $n^{\log_b a}$, but not polynomially larger. The ratio $\lg n$ is asymptotically less than n^{ε} for any positive ε. Thus, the Master Theorem ***doesn't*** apply here.

Master Theorem – What it means?

- **Case 1:** If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
  - $n^{\log_b a} = a^{\log_b n}$: Number of leaves in the recursion tree.
  - $f(n) = O(n^{\log_b a - \varepsilon}) \Rightarrow$ Sum of the cost of the nodes at each internal level asymptotically smaller than the cost of leaves by a *polynomial* factor.
  - Cost of the problem dominated by leaves, hence cost is $\Theta(n^{\log_b a})$.

Master Theorem – What it means?

- **Case 2:** If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
  - $n^{\log_b a} = a^{\log_b n}$: Number of leaves in the recursion tree.
  - $f(n) = \Theta(n^{\log_b a}) \Rightarrow$ Sum of the cost of the nodes at each level asymptotically the same as the cost of leaves.
  - There are $\Theta(\lg n)$ levels.
  - Hence, total cost is $\Theta(n^{\log_b a} \lg n)$.

Master Theorem – What it means?

- **Case 3:** If $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$, and if, for some constant $c < 1$ and all sufficiently large n, we have $a \cdot f(n/b) \leq c\, f(n)$, then $\mathbf{T(n) = \Theta(f(n))}$.

- $n^{\log_b a} = a^{\log_b n}$: Number of leaves in the recursion tree.
- $f(n) = \Omega(n^{\log_b a+\varepsilon}) \Rightarrow$ Cost is dominated by the root. Cost of the root is asymptotically larger than the sum of the cost of the leaves by a polynomial factor.
- Hence, cost is $\Theta(f(n))$.

Master Theorem – Proof for exact powers

- Proof when *n* is an exact power of *b*.
- Three steps.
  1. Reduce the problem of solving the recurrence to the problem of evaluating an expression that contains a summation.
  2. Determine bounds on the summation.
  3. Combine 1 and 2.

Proof for exact powers – Step 1

Lemma 4.2

Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a nonnegative function defined on exact powers of b. Define $T(n)$ on exact powers of b by the recurrence

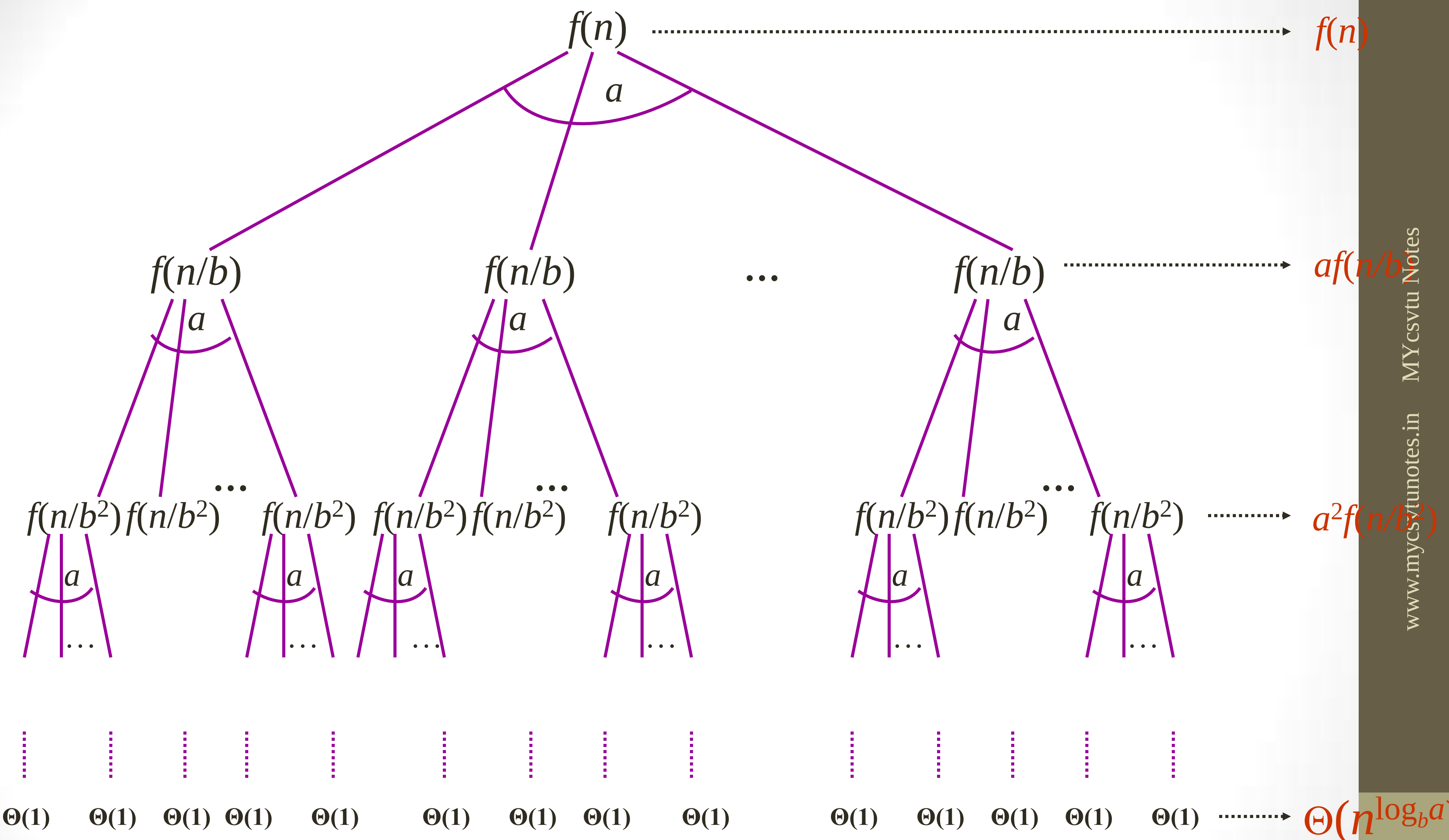
$T(n) = \Theta(1)$ if $n = 1$,

a +ve integer.

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \qquad (4.6)$$

Then

Proof of Lemma 4.2



Total: $T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$

Proof of Lemma 4.2 – Contd.

- Cost of the root – $f(n)$
- Number of children of the root = Number of nodes at distance 1 from the root = a.
- Problem size at depth 1 = Original Size/b = n/b.
- Cost of nodes at depth 1 = $f(n/b)$.

- Each node at depth 1 has a children.
- Hence, number of nodes at depth 2

  = # of nodes at depth 1 $\times$ # of children per depth 1 node,

  = $a \times a = a^2$
- Size of problems at depth 2 = ((Problem size at depth 1)/b) = $n/b/b = n/b^2$.
- Cost of problems at depth 2 = $f(n/b^2)$.

Proof of Lemma 4.2 – Contd.

- Continuing in the same way,
- number of nodes at depth j

  $= a^j$
- Size of problems at depth $j = n/b^j$.
- Cost of problems at depth $j = f(n/b^j)$.
- Problem size reduces to 1 at leaves.
- Let x be the depth of leaves. Then x is given by $n/b^x = 1$
- Hence, depth of leaf level is $\log_b n$.
- number of leaves = number of nodes at level $\log_b n$ = $a^{\log_b n} = n^{\log_b a}$.

Proof of Lemma 4.2 – Contd.

- Cost of a leaf node = $\Theta(1)$.
- So, total cost of all leaf nodes = $\Theta(n^{\log_b a})$. → (4.2 a)
- Total cost of internal nodes = Sum of total cost of internal nodes at all levels (from depth 0 (root) to depth $\log_b n - 1$).

$$= \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$ → (4.2 b)

- Total problem cost = Cost of leaves + Cost of internal nodes =

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$ (from 4.2 a and 4.2 b)

Step 2 – Bounding the Summation in Eq. (4.6)

Lemma 4.3

Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a nonnegative function defined on exact powers of b. A function $g(n)$ defined over exact powers of b by

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

can be bounded asymptotically for exact powers of b as follows.

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $g(n) = O(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $g(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $a f(n/b) \leq c f(n)$ for some constant $c < 1$ and all $n \geq b$, then $g(n) = \Theta(f(n))$.

Proof of Lemma 4.3

Case 1

$f(n) = O(n^{\log_b a-\varepsilon}) \Rightarrow f(n/b^j) = O((n/b^j)^{\log_b a-\varepsilon})$

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

$$= O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon}\right)$$

Factoring out terms and simplifying the summation within *O*-notation leaves an increasing geometric series.

$$\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon} = n^{\log_b a-\varepsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^{\varepsilon}}{b^{\log_b a}}\right)^j$$

$$= n^{\log_b a-\varepsilon} \sum_{j=0}^{\log_b n-1} \left(b^{\varepsilon}\right)^j$$

Proof of Lemma 4.3 – Contd.

$$\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon} = n^{\log_b a-\varepsilon} \sum_{j=0}^{\log_b n-1} \left(b^{\varepsilon}\right)^j$$

$$= n^{\log_b a-\varepsilon} \left(\frac{b^{\varepsilon \log_b n} - 1}{b^{\varepsilon} - 1}\right)$$

$$= n^{\log_b a-\varepsilon} \left(\frac{n^{\varepsilon} - 1}{b^{\varepsilon} - 1}\right)$$

$$= n^{\log_b a-\varepsilon} O(n^{\varepsilon})$$;because ε and *b* are constants.

$$= O(n^{\log_b a})$$

$$g(n) = O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon}\right) = O(n^{\log_b a})$$

Proof of Lemma 4.3 – Contd.

Case 2

$f(n) = \Theta(n^{\log_b a}) \Rightarrow f(n/b^j) = \Theta((n/b^j)^{\log_b a})$

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

$$= \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right)$$

Factoring out terms and simplifying the summation within Θ-notation leaves a constant series.

$$\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} = n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j$$

$$= n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1$$

$$= n^{\log_b a} \log_b n$$

Proof of Lemma 4.3 – Contd.

Case 2 – Contd.

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j} \right)^{\log_b a} \right)$$

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

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

Proof of Lemma 4.3 – Contd.

Case 3

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

$$= f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \cdots + a^{\log_b n-1} f$$

- $f(n)$ is nonnegative, by definition.
- a (number of subproblems) and b (factor by which the problem size is reduced at each step) are nonnegative.
- Hence, each term in the above expression for $g(n)$ is nonnegative. Also, $g(n)$ contains $f(n)$.
- Hence $g(n) = \Omega(f(n))$, for exact powers of b.

Proof of Lemma 4.3 – Contd.

Case 3 – Contd.

- By assumption, $a\, f(n/b) \leq c\, f(n)$, for $c < 1$ and all $n \geq b$.
- $\Rightarrow f(n/b) \leq (c/a)\, f(n)$.
- Iterating j times, $f(n/b^j) \leq (c/a)^j\, f(n)$.
- $\Rightarrow a^j\, f(n/b) \leq c^j\, f(n)$.

Proof of Lemma 4.3 – Contd.

Case 3 – Contd.

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

$$\leq \sum_{j=0}^{\log_b n - 1} c^j f(n)$$

$$\leq \sum_{j=0}^{\infty} c^j f(n)$$

$$= f(n)\left(\frac{1}{1-c}\right) \quad = O(f(n))$$

Substituting $a^j f(n/b) \leq c^j f(n)$ and simplifying yields a decreasing geometric series since $c < 1$.

Thus, $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ (proved earlier).

$\therefore\ g(n) = \Theta(f(n))$.

Master Theorem – Proof – Step 3

Lemma 4.4

Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a nonnegative function defined on exact powers of b. Define $T(n)$ on exact powers of b by the recurrence

$T(n) = \Theta(1)$ if $n = 1$,

$T(n) = aT(n/b) + f(n)$ if $n = b^i$, i is a +ve integer.

Then $T(n)$ can be bounded asymptotically for exact powers of b as follows.

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and $af(n/b) \leq c f(n)$ for some constant $c < 1$ and large n, then $T(n) = \Theta(f(n))$.

Lemma 4.4 – Proof

By Lemma 4.2,

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

Bounds obtained for all 3 cases in Lemma 4.3. Use them.

Case 1:

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

$$= \Theta(n^{\log_b a}) \quad \textbf{Why?}$$

Case 2:

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

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

Case 3:

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

$$= \Theta(f(n)) \qquad ; \because f(n) = \Omega(n^{\log_b a + \varepsilon})$$

Proof for when n is not an exact power of

- To complete the proof for Master Theorem in general,
  - Extend analysis to cases where floors and ceilings occur in the recurrence.
  - I.e., consider recurrences of the form

$$T(n) = aT(\lceil n/b \rceil) + f(n)$$

and

$$T(n) = aT(\lfloor n/b \rfloor) + f(n)$$

- Go through Sec. 4.4.2 in the text.