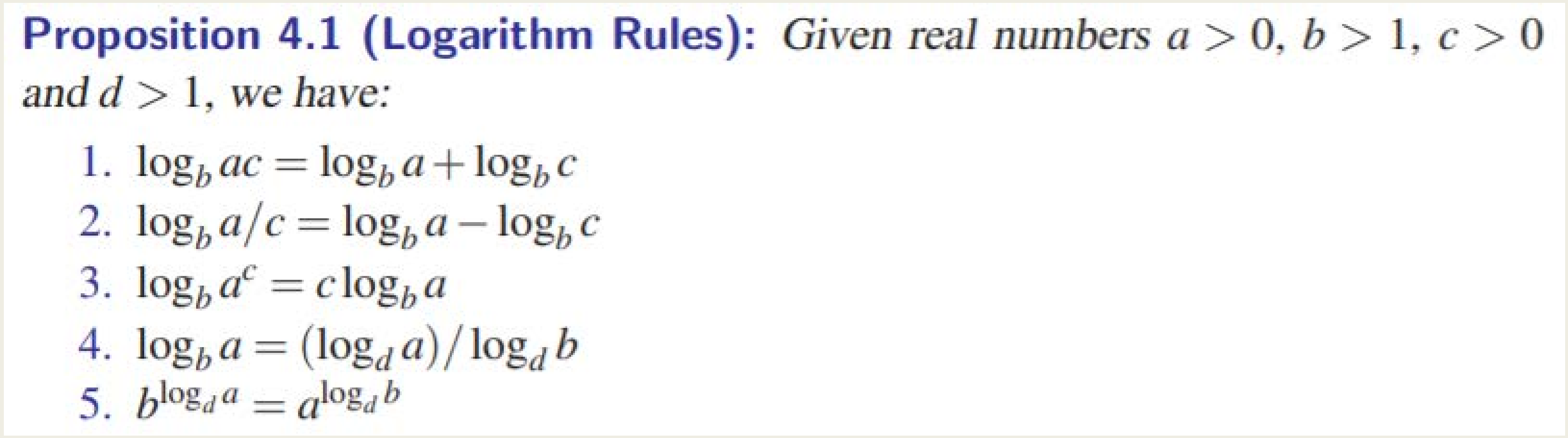
CH4. ANALYSIS TOOLS

CSED233 Data Structure
Prof. Hwanjo Yu
POSTECH

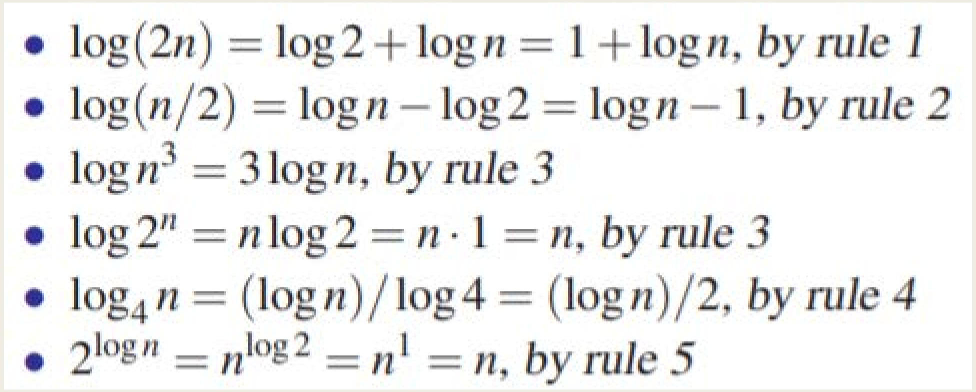
The seven functions (§ 4.1)

1. The Constant Function: f(n) = c
2. The Logarithm Function: f(n) = log n
   - *The base could be anything but commonly 2*



Proposition 4.1 (Logarithm Rules): *Given real numbers $a > 0$, $b > 1$, $c > 0$ and $d > 1$, we have:*

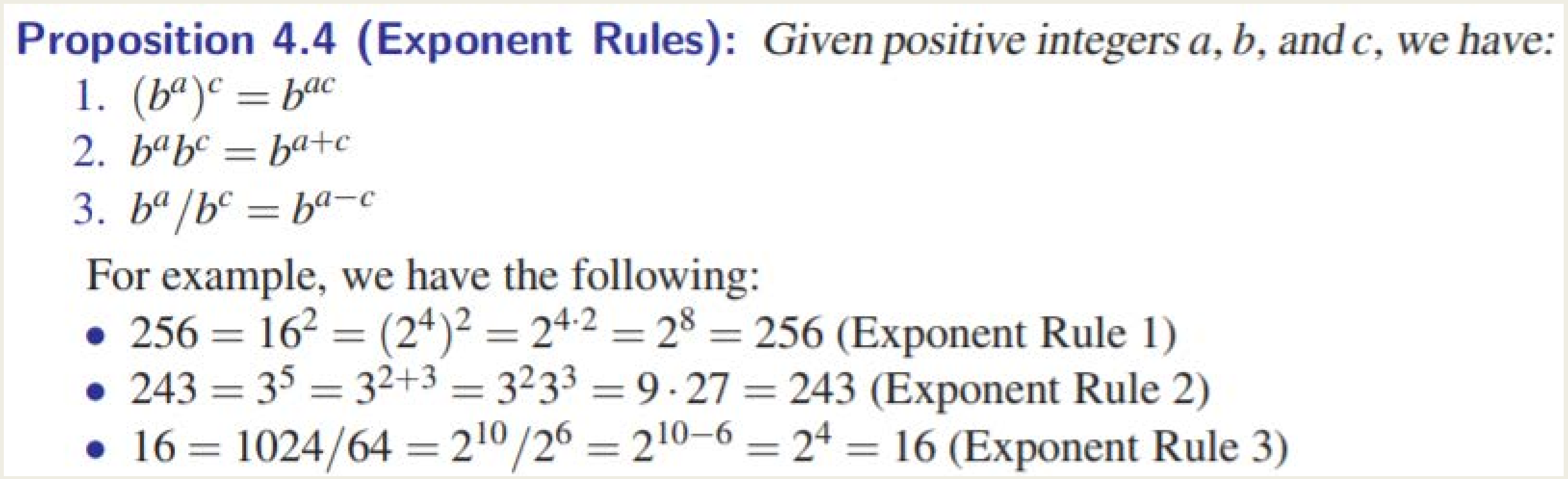
1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c \log_b a$
4. $\log_b a = (\log_d a)/\log_d b$
5. $b^{\log_d a} = a^{\log_d b}$



- $\log(2n) = \log 2 + \log n = 1 + \log n$, *by rule 1*
- $\log(n/2) = \log n - \log 2 = \log n - 1$, *by rule 2*
- $\log n^3 = 3 \log n$, *by rule 3*
- $\log 2^n = n \log 2 = n \cdot 1 = n$, *by rule 3*
- $\log_4 n = (\log n)/\log 4 = (\log n)/2$, *by rule 4*
- $2^{\log n} = n^{\log 2} = n^1 = n$, *by rule 5*

The seven functions (§ 4.1)

3. The Linear Function: $f(n) = n$
4. The N-Log-N Function: $f(n) = n \log n$
5. The Quadratic Function: $f(n) = n^2$
6. The Cubic Function and Other Polynomials
   - *Cubic:* $f(n) = n^3$
   - *Polynomial:* $f(n) = a_0+a_1n+a_2n^2+a_3n^3+\ldots+a_dn^d$
7. The Exponential Function: $f(n) = b^n$



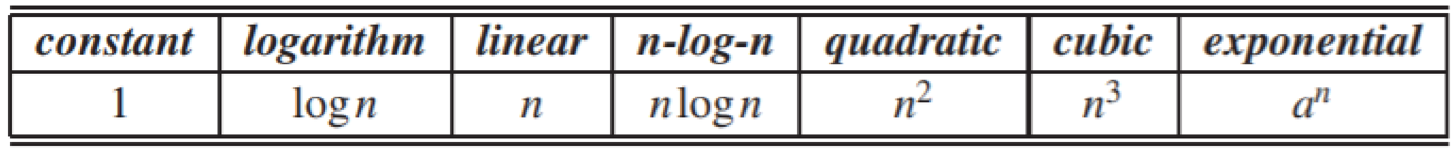
Proposition 4.4 (Exponent Rules): *Given positive integers a, b, and c, we have:*

1. $(b^a)^c = b^{ac}$
2. $b^a b^c = b^{a+c}$
3. $b^a / b^c = b^{a-c}$

For example, we have the following:

- $256 = 16^2 = (2^4)^2 = 2^{4\cdot 2} = 2^8 = 256$ (Exponent Rule 1)
- $243 = 3^5 = 3^{2+3} = 3^2 3^3 = 9 \cdot 27 = 243$ (Exponent Rule 2)
- $16 = 1024/64 = 2^{10}/2^6 = 2^{10-6} = 2^4 = 16$ (Exponent Rule 3)

Comparing Growth Rates (§ 4.1.8)



constant	*logarithm*	*linear*	*n-log-n*	*quadratic*	*cubic*	*exponential*
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Table 4.1: Classes of functions. Here we assume that $a > 1$ is a constant.

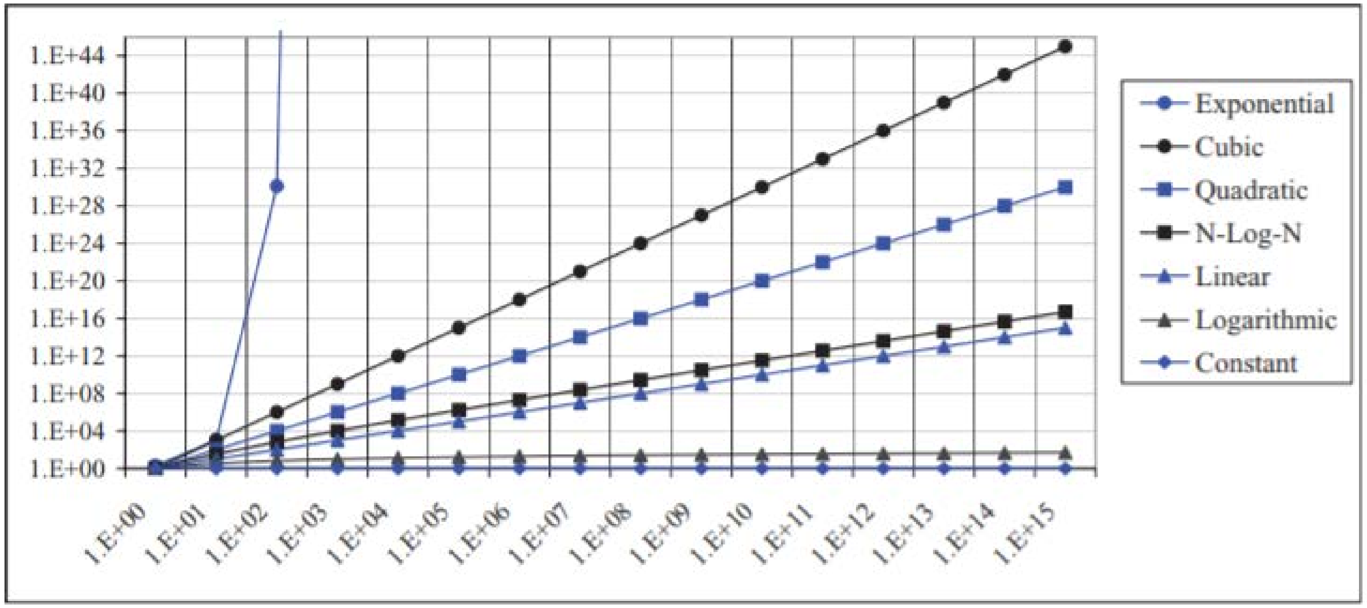


Figure 4.2: Growth rates for the seven fundamental functions used in algorithm analysis. We use base $a = 2$ for the exponential function. The functions are plotted in a log-log chart, to compare the growth rates primarily as slopes. Even so, the exponential function grows too fast to display all its values on the chart. Also, we use the scientific notation for numbers, where aE+b denotes $a10^b$.

Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program

Example: find max element of an array

```
Algorithm arrayMax(A, n)
    Input array A of n integers
    Output maximum element of A

    currentMax ← A[0]
    for i ← 1 to n − 1 do
        if A[i] > currentMax then
            currentMax ← A[i]
    return currentMax
```

Pseudocode Details

- Control flow
  - ***if** … **then** … [**else** …]*
  - ***while** … **do** …*
  - ***repeat** … **until** …*
  - ***for** … **do** …*
  - *Indentation replaces braces*
- Method declaration

  ***Algorithm** method (arg [, arg…])*

  ***Input** …*

  ***Output** …*

- Method call

  ***var.method (arg** [, **arg**…])*

- Return value

  ***return** expression*

- Expressions

$\leftarrow$	*Assignment (like $=$ in C++)*
$=$	*Equality testing (like $==$ in C++)*
n^2	*Superscripts and other mathematical formatting allowed*

Counting primitive operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
Algorithm arrayMax(A, n)                        # operations
    currentMax ← A[0]                           2
    for i ← 1 to n − 1 do                       2n
        if A[i] > currentMax then               2(n − 1)
              currentMax ← A[i]                 2(n − 1)
        { increment counter i }                 2(n − 1)
    return currentMax                           1

                                          Total 8n − 2
```

Estimating Running Time

- Algorithm ***arrayMax*** executes $8n - 2$ primitive operations in the worst case. Define:

  a = *Time taken by the fastest primitive operation*

  b = *Time taken by the slowest primitive operation*

- Let $T(n)$ be worst-case time of ***arrayMax.*** Then

$$a\,(8n - 2) \le T(n) \le b(8n - 2)$$

- Hence, the running time $T(n)$ is bounded by two linear functions

Analysis of Algorithms (§ 4.2)

- How do we measure the “goodness” of an algorithm?
  - *Running time, space usage*
  - *Measure the running time of an algorithm w.r.t. the input size*
  - *We want the measure independent of environmental factors such as processor speed, disk speed, OS, programming language, etc.*
- Counting primitive operations
  - *Assigning a value to a variable*
  - *Calling a function*
  - *Performing an arithmetic operation*
  - *Comparing two numbers*
  - *Indexing into an array*
  - *Following an object reference*
  - *Returning from a function*
- Worse case analysis
  - *Average case analysis requires to compute input probabilities*
  - *Worst case analysis is easier and requires algorithms to do well on every input*

Asymptotic Notation (§ 4.2.3)

- Focus on growth rate of running time w.r.t. the input size n
- "big-Oh" notation:
  - *$f(n)$ is $O\big(g(n)\big)$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$, for $n \geq n_0$*
  - *"$f(n)$ is big-Oh of $g(n)$" or "$f(n)$ is order of $g(n)$"*

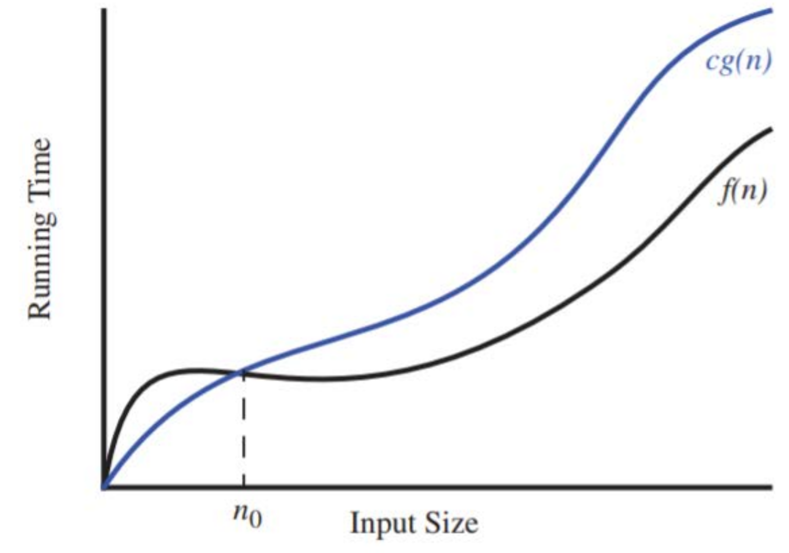


Figure 4.5: The "big-Oh" notation. The function $f(n)$ is $O(g(n))$, since $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

Big-Oh

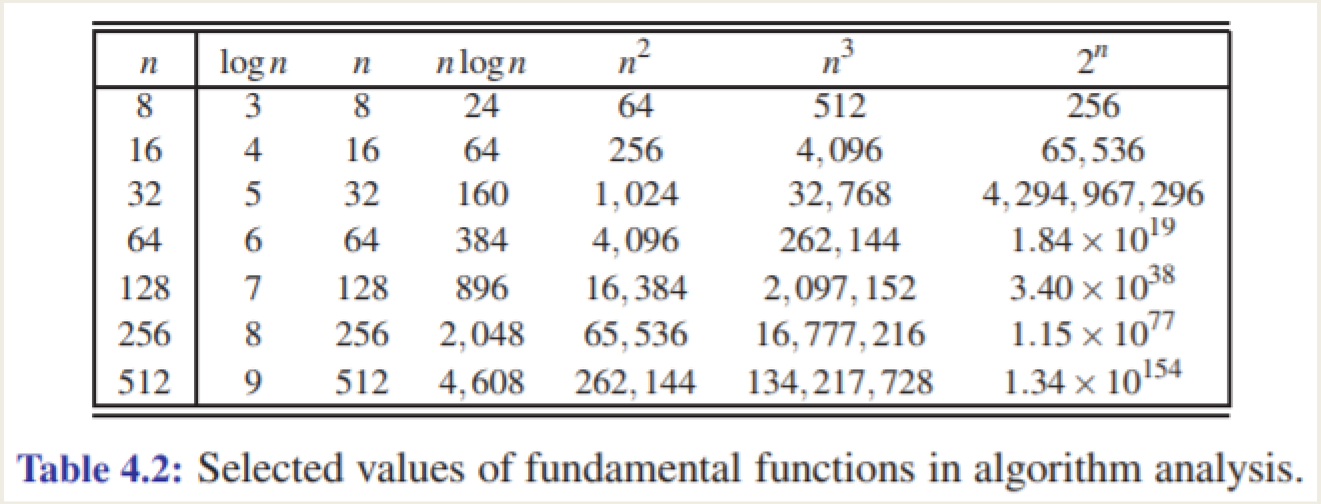
- Examples
  - $5n^4 + 3n^3 + 2n^2 + 4n + 1$ *is* $O(n^4)$
  - $5n^2 + 3n \log n + 2n + 5$ *is* $O(n^2)$
  - $3 \log n + 2$ *is* $O(\log n)$
  - 2^{n+2} *is* $O(2^n)$
  - $2n + 100 \log n$ *is* $O(n)$

Big-Omega and Big-Theta

- Big-Omega
  - $f(n)$ *is* $\Omega\big(g(n)\big)$ *if there is a real constant* $c > 0$ *and an integer constant* $n_0 \geq 1$ *such that* $f(n) \geq cg(n)$, *for* $n \geq n_0$
  - $3n \log n + 2n$ *is* $\Omega(n \log n)$
- Big-Theta
  - $f(n)$ *is* $\Theta\big(g(n)\big)$ *if there is a real constant* $c' > 0$ *and* $c'' > 0$ *an integer constant* $n_0 \geq 1$ *such that* $c'g(n) \leq f(n) \leq c''g(n)$, *for* $n \geq n_0$
  - $3n \log n + 4n + 5 \log n$ *is* $\Theta(n \log n)$

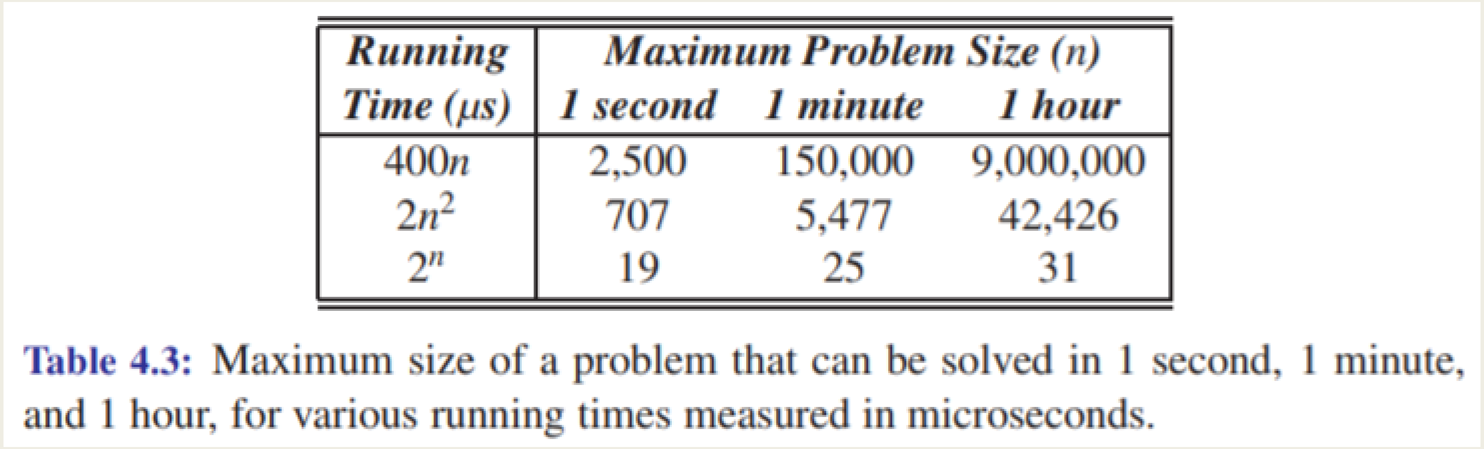
Asymptotic Analysis

■ "Algorithm A is *asymptotically better* than algorithm B" means..



n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Table 4.2: Selected values of fundamental functions in algorithm analysis.



Running Time (μs)	***Maximum Problem Size (n)***		
	1 second	***1 minute***	***1 hour***
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

Table 4.3: Maximum size of a problem that can be solved in 1 second, 1 minute, and 1 hour, for various running times measured in microseconds.

Asymptotic Analysis and Time Complexity

- Generally speaking,
  - $O(n \log n)$ *is efficient*
  - $O(n^2)$ *is fast enough when n is small*
  - $O(2^n)$ *is never considered efficient*
- Examples
  - *Insertion sort is* $O(n^2)$
  - *Quick sort is* $O(n \log n)$
  - *Find max element of an array is* $O(n)$
  - *Search with indexing is* $O(\log n)$
  - *Hashing is* $O(1)$
- A recursive algorithm for computing powers

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{otherwise} \end{cases}$$

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, (n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x, n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$