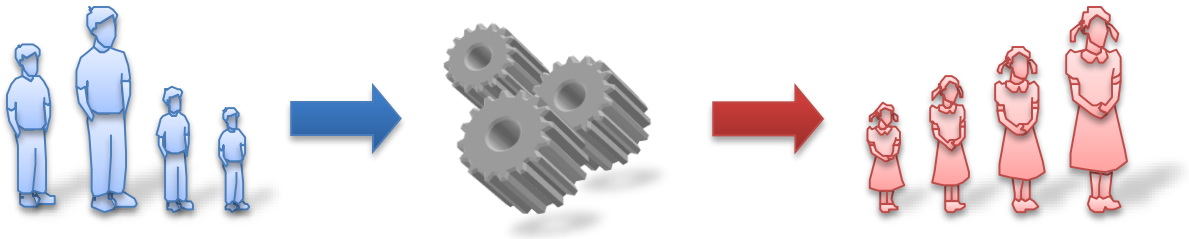
Data Structure:
Analysis of Algorithms

2017

Instructor: Prof. Young-guk Ha
Dept. of Computer Science & Engineering

Recall What is Algorithm

- An ***algorithm*** is a set of instructions (i.e., a step-by-step procedure) for solving a problem in a finite amount of time
  - Most algorithms transform input objects into output objects



Running Time of Algorithm

- Running time of an algorithm typically grows not only with the size but with the composition of the input
- ***Average case*** running time is often difficult to determine
  - Must consider probability of every possible input composition
- ***Best case*** running time is actually meaningless
  - Provides little information on algorithm characteristic
  - Best case rarely occurs
- Thus, we mainly focus on the ***worst case*** running time
  - Easier to analyze
  - Determines the ***upper bound*** of running time
  - Crucial to some applications (e.g., real-time applications such as atomic control, airplane control, …)

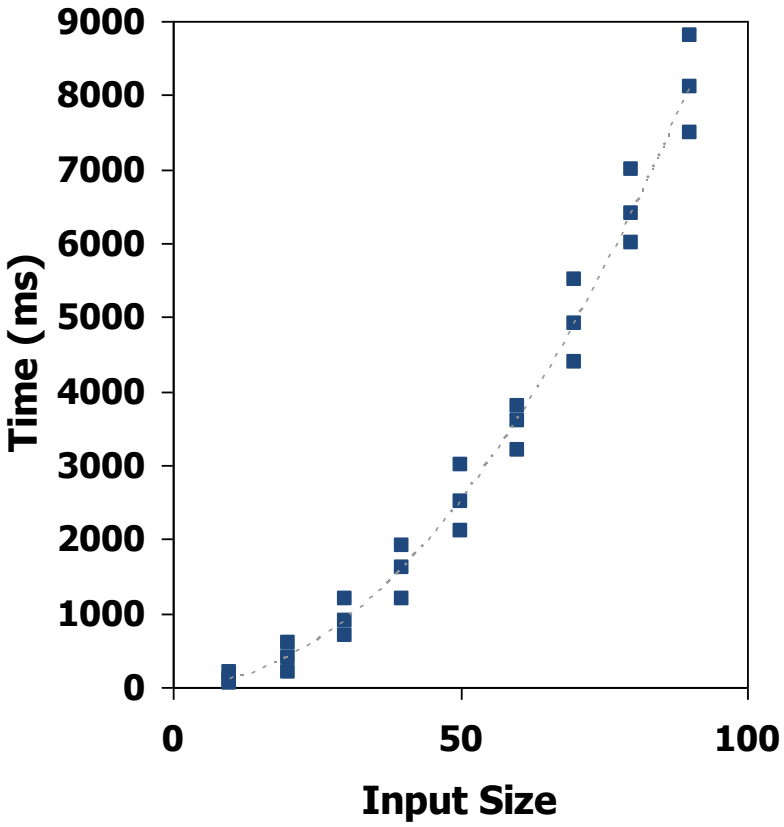
How to Determine Performance of Algorithm

1. Experimental measurement

2. Theoretical analysis

Experimental Measurement of Running Time

1) Write a complete program that implements the algorithm
   - E.g., function **`sum`** that adds a list of numbers
2) Run the program with inputs of varying size and composition
3) Use a Java method such as **`System.currentTimeMillis()`** to get an accurate measure of the actual running time
   - We can also use system function **`clock()`** in C or C++
4) Plot the results



Limitations of the Experimental Measurement

- Experimental performance measurement is valuable but …
- It is necessary to implement the complete algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment
- In order to compare two algorithms, exactly the same H/W and S/W environment must be used

→ ***Machine-Dependent Scheme***

Theoretical Performance Analysis

- Using a high-level specification of the algorithm (e.g., pseudo-code) instead of a complete implementation
  - Counts ***primitive operations*** of the algorithm to determine the running time
  - Characterizes running time as a function of the input size n, e.g., $T(n) = n \log n$, $T(n) = n^2 + n$
- Takes into account all possible inputs
- Allows us to evaluate the performance of an algorithm independently of the H/W or S/W environment

→ ***Machine-Independent Scheme***

Primitive Operations

- Basic computations performed by an algorithm, which are assumed to take a *constant amount of time*
- Identifiable in pseudo-code and largely independent from the programming language
- Examples
  - Evaluating an expression (e.g., +, -, /, *, <, >, ==, …)
  - Assigning a value to a variable (e.g., =)
  - Indexing into an array (e.g., arr[i])
  - Comparison (e.g., "if" statement)
  - Calling or returning from a function or method
  - And so on

Pseudo-code

- To specify an algorithm, we will use a pseudo-code together with Java
  - High-level description of an algorithm
  - More structured than English prose and less detailed than a programming language
  - Preferred notation for describing algorithms
  - Hides detailed program design issues

[Pseudo-Code Example]
Find the largest element from 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
```

Pseudo-code Details

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

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

  Input …

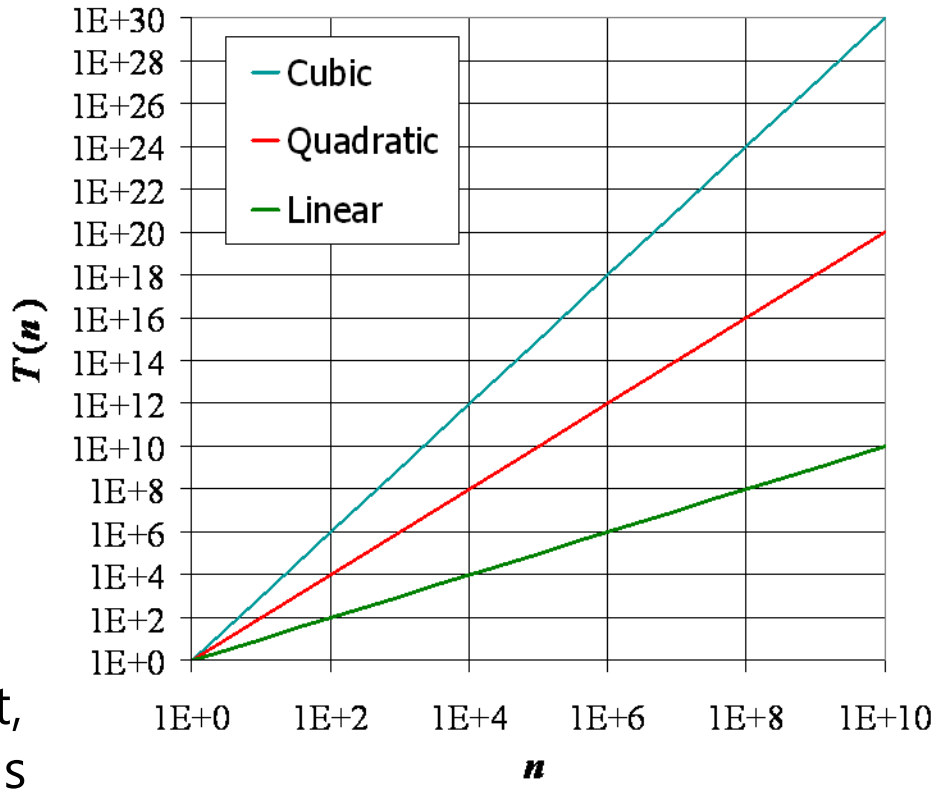
  Output …
- Function call

  function(***arg*** [, ***arg*** …])
- Return value

  return ***expression***
- Expressions
  - ← Assignment (like "=" in Java)
  - = Equality testing (like "==" in Java)
  - n^2 Superscripts and other mathematical formatting allowed

Important Functions for Algorithm Analysis

- Seven functions of input size n that often appear in algorithm analysis
  - Constant: $T(n) = C$
  - Logarithmic: $T(n) = \log n$
  - Linear: $T(n) = n$
  - Log linear: $T(n) = n \log n$
  - Quadratic: $T(n) = n^2$
  - Cubic: $T(n) = n^3$
  - Exponential: $T(n) = 2^n$
  - Factorial: $T(n) = n!$
- For example, in a log-scale chart, the slope of the line corresponds to the ***growth rate*** of the function



Counting Primitive Operations

- By inspecting a pseudo-code, we can determine the number of primitive operations executed by an algorithm as a function of the input size ***n***

Algorithm ***arrayMax***(*A*, *n*)	# Primitive operations	
currentMax ← *A*[0]	2	: array indexing, assignment
for *i* ← 1 to *n* – 1 do	2*n*	: assignment, comparison
if *A*[*i*] > *currentMax* then *currentMax* ← *A*[*i*]	2(*n* – 1)	: array indexing, comparison
// replaces *currentMax*	2(*n* – 1)	: array indexing, assignment
// (*i* = *i* + 1) in the for loop	2(*n* – 1)	: addition, assignment
return *currentMax*	1	: return from a function
Call ***arrayMax***(…)	1	: call a function
Total:	**8*n* – 2**	

- Algorithm **`arrayMax`** executes exactly $8n - 2$ primitive operations in the worst case (when the input is sorted in ascending order)
  - I.e., $T(n) = 8n - 2$
  - What is the minimum number of primitive OPs (best case)?

Determining Exact Running Time

- Exact counting of primitive operations
  → *Is it worth the effort?*
  - Exact counting of primitive operations is an exceedingly difficult task as the algorithm becomes larger and more complicated
  - Actually not very exact because of the inexactness of what a primitive operation stands for
  - Furthermore, primitive operations are assumed to take a constant amount of time, but may not actually

Asymptotic Algorithm Analysis

- Refers to analyzing performance of an algorithm based on the ***growth rate*** of its running time as n grows
  - Focus on a "*big-picture aspect*" of algorithm performance (not a "snapshot") by disregarding *constant factors* and *lower-order terms* of the running time function $T(n)$
  - Expressing running time functions with the "***Big-Oh*** (O)," "***Big-Omega*** (Ω)," or "***Big-Theta*** (Θ)" notation

- Asymptotic analysis example of the **`arrayMax`**
  - Algorithm **`arrayMax`** executes exactly "$8n - 2$" primitive operations in the worst case, i.e., "$T(n) = 8n - 2$"
  - Instead, we say "$T(n) = O(n)$" by asymptotic notation
    - This implies that "the growth rate of $8n - 2$ is not greater than the growth rate of n as n grows"
    - For reference, $O(n)$ is read "big-oh of n" or "order of n"

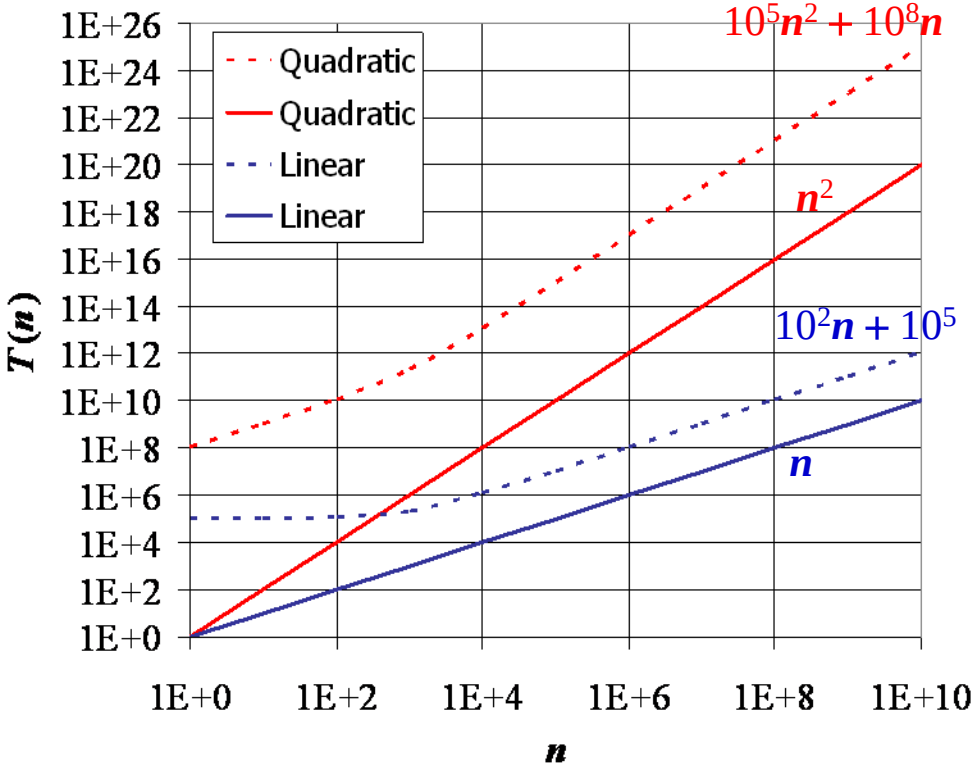
Growth Rate of Running Time

- Growth rate of $T(n)$
  - Does ***NOT*** affected by
    - Constant factors
    - Lower-order terms

    as n grows

- Examples
  - Linear functions
    - $T(n) = n$
    - $T'(n) = 10^2 n + 10^5$
  - Quadratic functions
    - $T(n) = n^2$
    - $T'(n) = 10^5 n^2 + 10^8 n$

[Example]
Growth rate of functions



Big-Oh (O) Notation

- Formal definition
  - Given functions $f(n)$ and $g(n)$, $f(n) = O(g(n))$ iff there are a real constant $c > 0$ and an integer constant $n_0 \geq 1$ s.t.

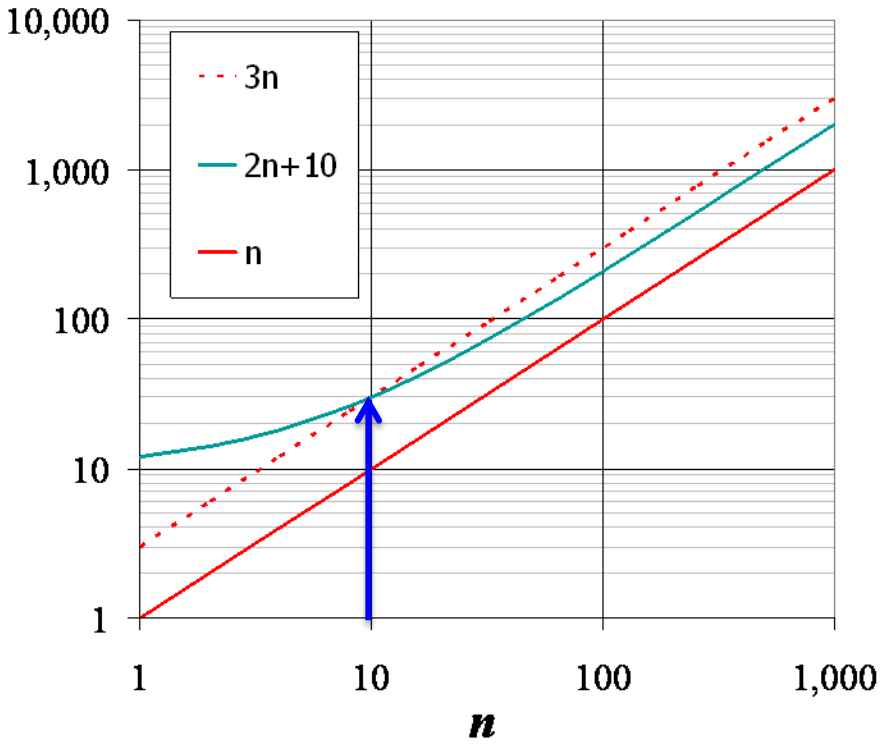
    $f(n) \leq cg(n)$ for $n \geq n_0$

  - Most commonly used for asymptotic analysis
- Example
  - Let $f(n) = 2n+10$ then we can say $f(n) = O(n)$
  - Justification by the definition
    - $2n + 10 \leq cn$ for $n \geq n_0$
    - $2n + 10 \leq 3n$ for $n \geq 10$
    - We can determine $c = 3$ and $n_0 = 10$

[Example]

"$2n + 10 \leq 3n$ holds if $n \geq 10$"

(i.e., $c = 3$, $n_0 = 10$)



Intuition for Big-Oh Notation

- The big-Oh notation informs us of the ***upper bound*** of the growth rate of an algorithm's running time

- The statement "$\boldsymbol{f(n) = O(g(n))}$" means that
  - The growth rate of the algorithm's running time $\boldsymbol{f(n)}$ is *no greater than* the growth rate of $\boldsymbol{g(n)}$
  - It is guaranteed that the algorithm is completed in at most a constant times of $\boldsymbol{g(n)}$ as the input size $\boldsymbol{n}$ grows

Big-Oh Rules

1. If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$
   - Drop all the *lower-order* terms
   - Drop all the *constant* factors

2. Use the *smallest* possible class of functions
   - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"

3. Use the *simplest* expression of the class
   - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Big-Oh Examples (1)

- $7n - 2 = O(n)$

  : need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq cn$ for $n \geq n_0$

  This is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5 = O(n^3)$

  : need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$

  This is true for $c = 4$ and $n_0 = 21$

- $3 \log n + 5 = O(\log n)$

  : need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \log n$ for $n \geq n_0$

  This is true for $c = 8$ and $n_0 = 2$

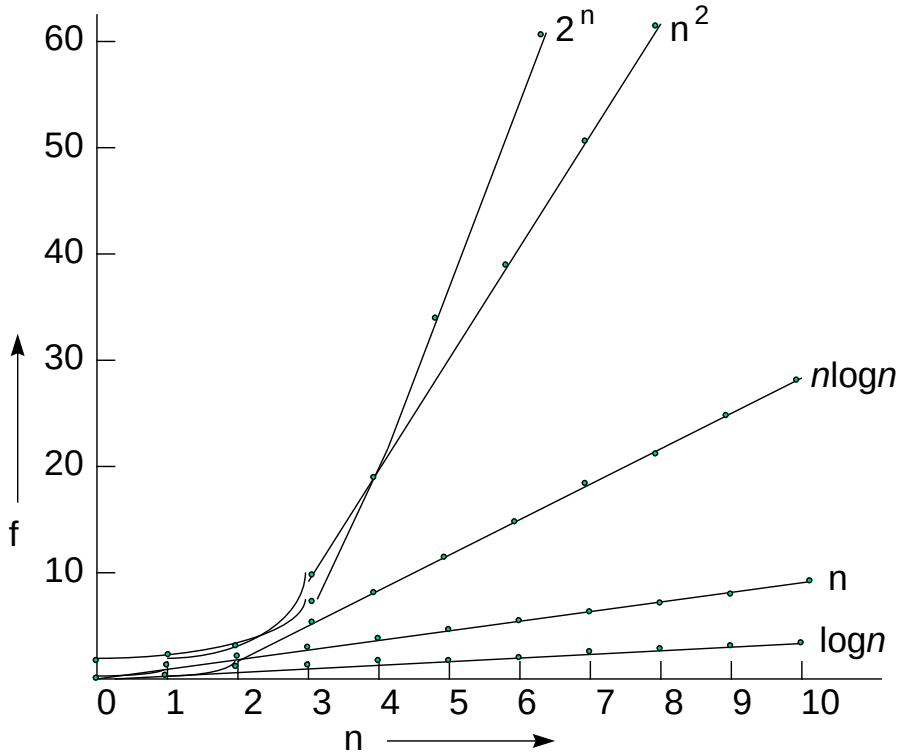
Big-Oh and Growth Rate

- We can use the big-Oh notation to rank algorithms according to their growth rate of running time

Better

1. Constant time: $O(1)$
2. Logarithm time: $O(\log n)$
3. Linear time: $O(n)$
4. Log linear time: $O(n \log n)$
5. Polynomial time
   - Quadratic: $O(n^2)$
   - Cubic: $O(n^3)$
   - …
6. Exponential time: $O(2^n)$
7. Factorial time: $O(n!)$

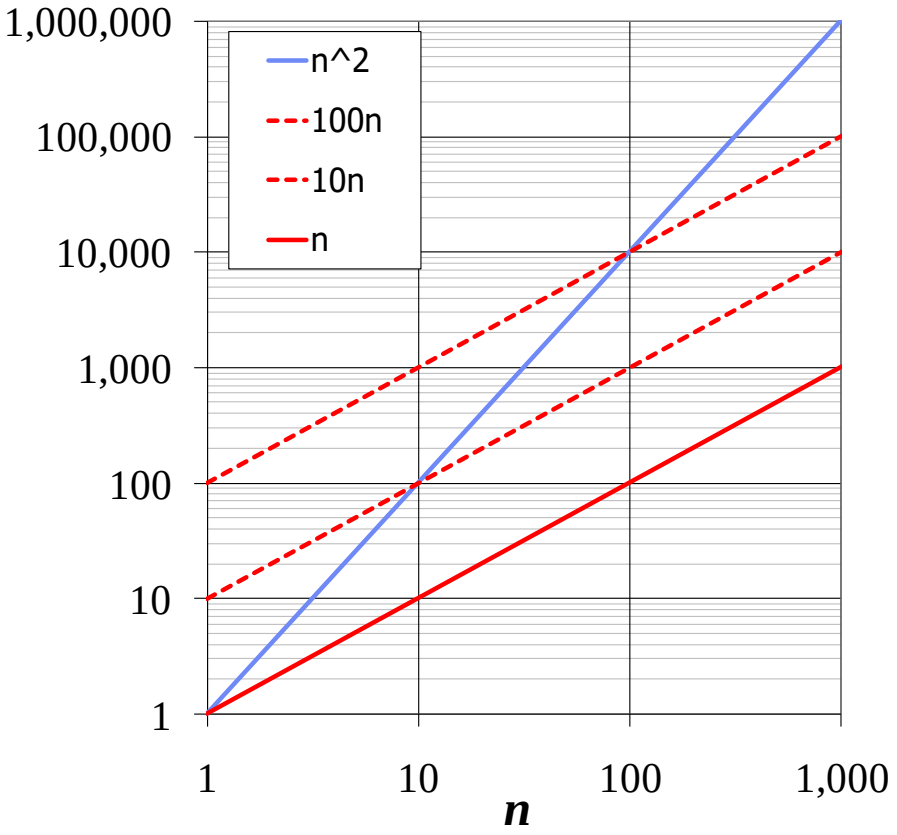
Worse



Big-Oh Examples (2)

- $n^2 \neq O(n)$
  - $n^2 \leq cn, n \leq c$
  - The above inequality cannot be satisfied since c must be a constant
  - Growth rate of n cannot be an upper bound of growth rate of n^2 as n grows

- $n = O(n^2)$
  - $n \leq cn^2, 1 \leq cn$
  - The above inequality is satisfied when $c = 1$ and $n \geq 1$ ($n_0 = 1$)
  - Growth rate of n^2 can be an upper bound of growth rate of n as n grows (however, better use $O(n)$ instead)



Big-Oh Examples (3)

- $7 = O(1)$
- $3\boldsymbol{n} + 2 = O(\boldsymbol{n})$
- $2^{\boldsymbol{n}+2} = 4 * 2^{\boldsymbol{n}} = O(2^{\boldsymbol{n}})$
- $10\boldsymbol{n}^2 + 4\boldsymbol{n} + 2 = O(\boldsymbol{n}^2)$
- $1000\boldsymbol{n}^2 + 100\boldsymbol{n} - 6 = O(\boldsymbol{n}^2)$
- $6 * 2^{\boldsymbol{n}} + \boldsymbol{n}^2 = O(2^{\boldsymbol{n}})$
- $2\boldsymbol{n} + 1000 \log \boldsymbol{n} = O(\boldsymbol{n})$
- $\boldsymbol{n} \log \boldsymbol{n} + 10\boldsymbol{n} + 5 = O(\boldsymbol{n} \log \boldsymbol{n})$
- $3 \log \boldsymbol{n} + 2 = O(\log \boldsymbol{n})$
- $3\boldsymbol{n} + 3 = O(\boldsymbol{n}^2)$ → *Not wrong, but better use $O(\boldsymbol{n})$ according to the rule #2*
- $10\boldsymbol{n}^2 + 4\boldsymbol{n} + 2 = O(10\boldsymbol{n}^2)$ → *Not wrong, but better use $O(\boldsymbol{n}^2)$ instead*

Calculating Running Time

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$f(n)=n\log n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01μs	.03μs	.1μs	1μs	10μs	10sec	1μs
20	.02μs	.09μs	.4μs	8μs	160μs	2.84hr	1ms
30	.03μs	.15μs	.9μs	27μs	810μs	6.83d	1sec
40	.04μs	.21μs	1.6μs	64μs	2.56ms	121.36d	18.3min
50	.05μs	.28μs	2.5μs	125μs	6.25ms	3.1yr	13d
100	.10μs	.66μs	10μs	1ms	100ms	3171yr	$4*10^{13}$yr
1,000	1.00μs	9.96μs	1ms	1sec	16.67min	$3.17*10^{13}$yr	$32*10^{283}$yr
10,000	10.00μs	130.03μs	100ms	16.67min	115.7d	$3.17*10^{23}$yr	
100,000	100.00μs	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$yr	
1,000,000	1.00ms	19.92ms	16.67min	31.71yr	$3.17*10^{7}$yr	$3.17*10^{43}$yr	

μs = microsecond = 10^{-6} seconds
ms = millisecond = 10^{-3} seconds
sec = seconds
min = minutes
hr = hours
d = days
yr = years

[Running times on a computer with 1.0 GHz CPU]

Relatives of Big-Oh

- *Big-Omega* (Ω) notation
  - **Formal definition**: $f(n)$ is $\Omega(g(n))$ iff there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ s.t.

$$f(n) \geq cg(n) \text{ for } n \geq n_0$$

- *Big-Theta* (Θ) notation
  - **Formal definition**: $f(n)$ is $\Theta(g(n))$ iff there are real constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ s.t.

$$c'g(n) \leq f(n) \leq c''g(n) \text{ for } n \geq n_0$$

i.e., $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Intuition of Big-Omega and Big-Theta

- Big-Omega notation
  - Informs us of the ***lower bound*** of the growth rate of an algorithm's running time
    - $f(n)$ is $\Omega(g(n))$ if the growth rate of $f(n)$ is ***greater than or equal to*** the growth rate of $g(n)$
    - It is guaranteed that the algorithm is completed in at least a constant times of $g(n)$ as the input size n grows
- Big-Theta notation
  - Informs us of the ***tight bound*** of the growth rate of an algorithm's running time (i.e., exact growth rate)
    - $f(n)$ is $\Theta(g(n))$ if the growth rate of $f(n)$ is ***equal to*** the growth rate of $g(n)$
    - It is guaranteed that the algorithm is completed in a constant times of $g(n)$ as the input size n grows

Big-Omega and Big-Theta Examples (1)

- $5n^2 = \Omega(n^2)$

  : $5n^2$ is $\Omega(n^2)$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $5n^2 \geq cn^2$ for $n \geq n_0$

  → $c = 5$ and $n_0 = 1$

- $5n^2 = \Omega(n)$

  : $5n^2$ is $\Omega(n)$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $5n^2 \geq cn$ for $n \geq n_0$

  → $c = 1$ and $n_0 = 1$

- $5n^2 = \Theta(n^2)$

  : $5n^2 = \Theta(n^2)$ because both "$5n^2 = \Omega(n^2)$" and "$5n^2 = O(n^2)$" hold

  We have already seen $5n^2 = \Omega(n^2)$;

  for $5n^2 = O(n^2)$, there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $5n^2 \leq cn^2$ for $n \geq n_0$

  → $c = 5$ and $n_0 = 1$

Big-Omega and Big-Theta Examples (2)

- $3n + 2 = \Omega(n)$

  → as $3n + 2 \geq 3n$ for all $n \geq 1$

- $100n + 6 = \Omega(n)$
- $10n^2 + 4n + 2 = \Omega(n^2)$
- $6 * 2^n + n^2 = \Omega(2^n)$
- $6 * 2^n + n^2 = \Omega(n^2)$
- $6 * 2^n + n^2 = \Omega(n)$
- $6 * 2^n + n^2 = \Omega(1)$

- $3n + 2 = \Theta(n)$

  → as $3n \leq 3n + 2 \leq 4n$ for all $n \geq 2$

- $10n^2 + 4n + 2 = \Theta(n^2)$
- $6 * 2^n + n^2 = \Theta(2^n)$
- $10n^2 + 4n + 2 \neq \Theta(n)$
- $6 * 2^n + n^2 \neq \Theta(n^2)$
- $6 * 2^n + n^2 \neq \Theta(n^{100})$
- $6 * 2^n + n^2 \neq \Theta(1)$