Design and Analysis of Algorithms

Growth of Functions

Prof. Chuhua Xian

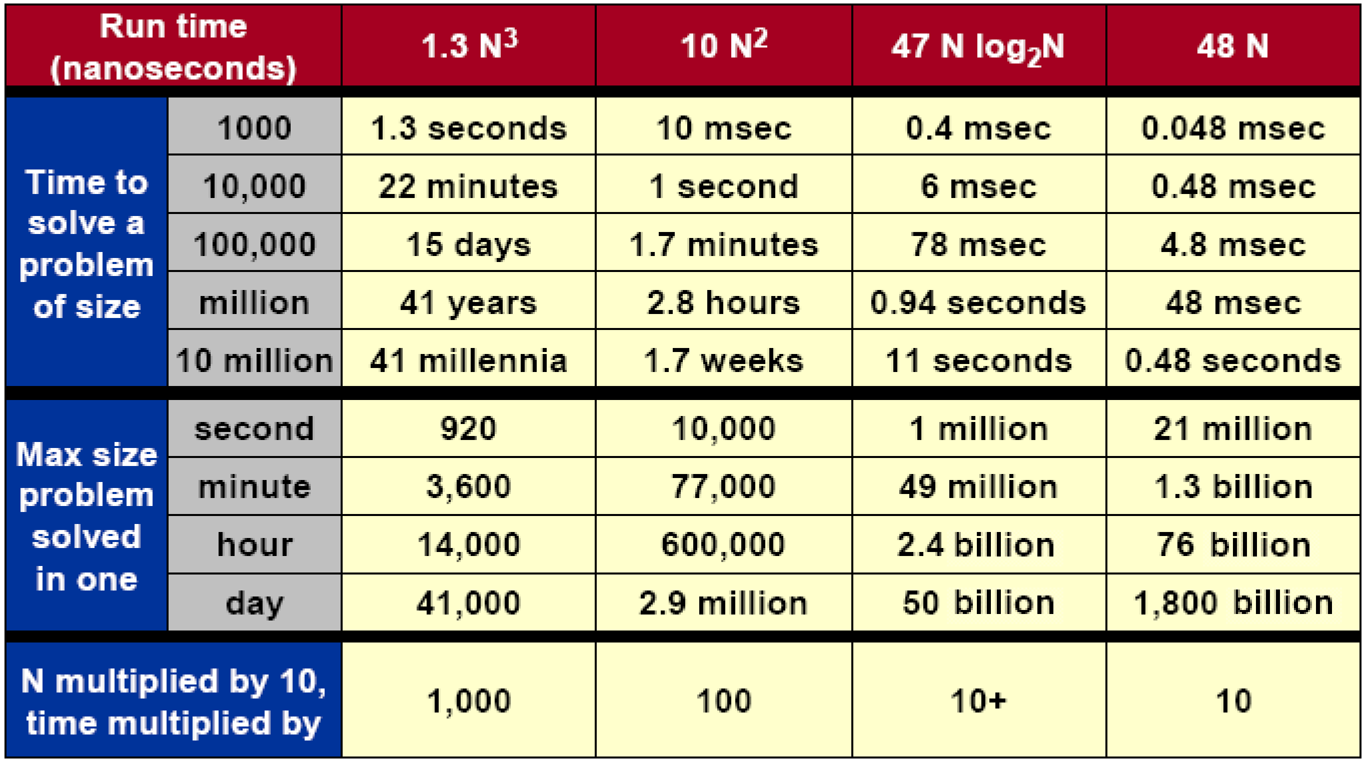
Email: chhxian@scut.edu.cn

School of Computer Science and Engineering

Growth of Functions

- **Topics:**
  - **Growth of functions**
  - **O/Θ/Ω notations**

What Does It Matter?



Run time (nanoseconds)		$1.3 N^3$	$10 N^2$	$47 N \log_2 N$	$48 N$
Time to solve a problem of size	1000	1.3 seconds	10 msec	0.4 msec	0.048 msec
	10,000	22 minutes	1 second	6 msec	0.48 msec
	100,000	15 days	1.7 minutes	78 msec	4.8 msec
	million	41 years	2.8 hours	0.94 seconds	48 msec
	10 million	41 millennia	1.7 weeks	11 seconds	0.48 seconds
Max size problem solved in one	second	920	10,000	1 million	21 million
	minute	3,600	77,000	49 million	1.3 billion
	hour	14,000	600,000	2.4 billion	76 billion
	day	41,000	2.9 million	50 billion	1,800 billion
N multiplied by 10, time multiplied by		1,000	100	10+	10

Orders of Magnitude

Seconds	Equivalent
1	1 second
10	10 seconds
10^2	1.7 minutes
10^3	17 minutes
10^4	2.8 hours
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades
10^{10}	3.1 centuries
. . .	forever
10^{21}	age of universe

Meters Per Second	Imperial Units	Example
10^{-10}	1.2 in / decade	Continental drift
10^{-8}	1 ft / year	Hair growing
10^{-6}	3.4 in / day	Glacier
10^{-4}	1.2 ft / hour	Gastro-intestinal tract
10^{-2}	2 ft / minute	Ant
1	2.2 mi / hour	Human walk
10^2	220 mi / hour	Propeller airplane
10^4	370 mi / min	Space shuttle
10^6	620 mi / sec	Earth in galactic orbit
10^8	62,000 mi / sec	1/3 speed of light

Powers of 2		
	2^{10}	thousand
	2^{20}	million
	2^{30}	billion

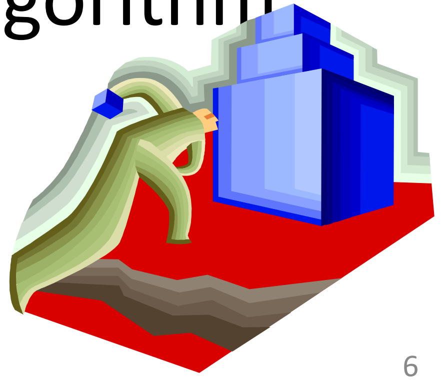
Asymptotic Growth

- **In the insertion-sort example we discussed that when analyzing algorithms we are**
  --interested in worst-case running time as function of input size *n*.
  --not interested in exact constants in bound.
  --not interested in lower order terms

- **A good reason for not caring about constants and lower order terms is that the RAM model is not completely realistic anyway (not all operations cost the same).**

Growth Rate of Running Time

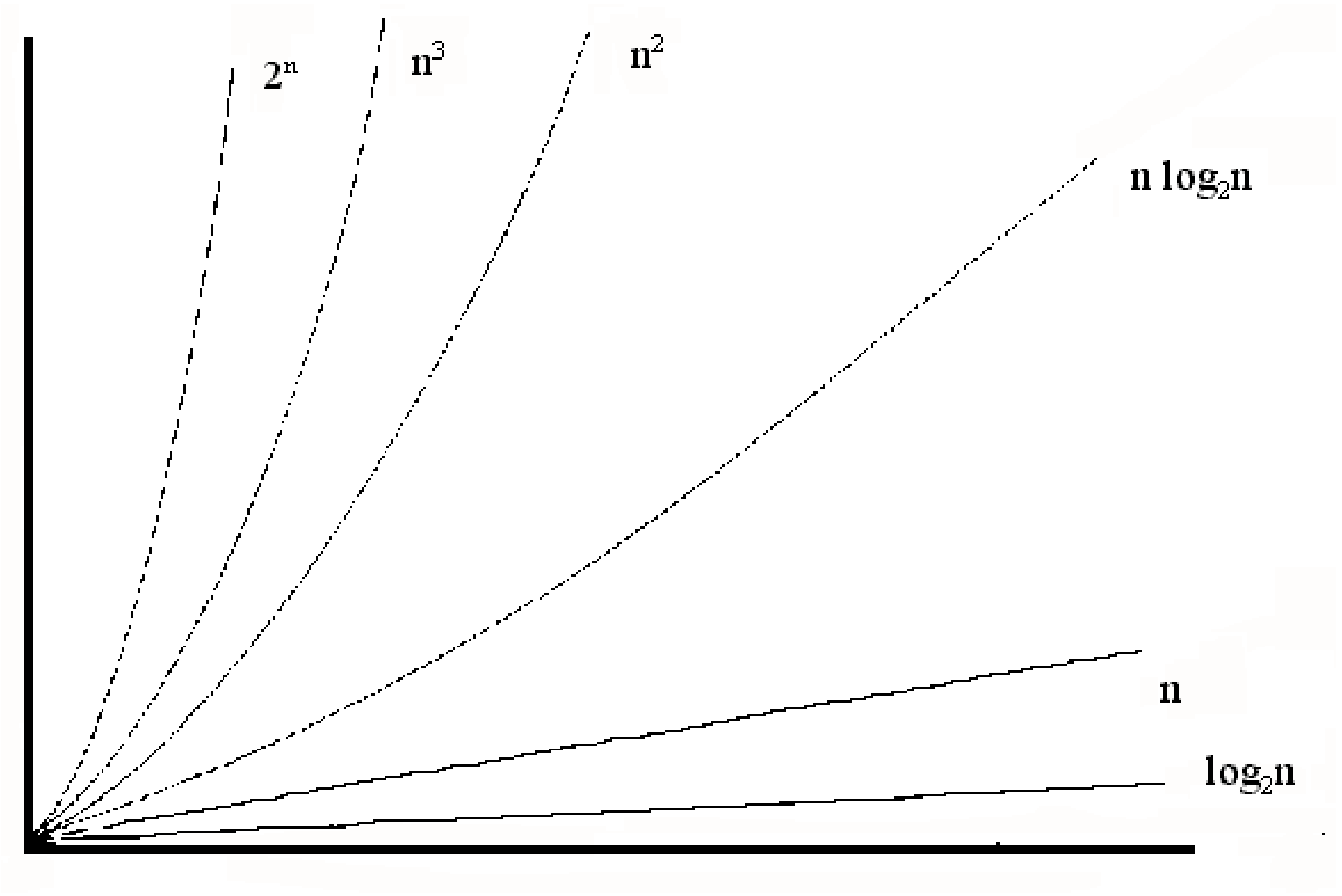
- Changing the hardware/ software environment
  - Affects $\boldsymbol{T(n)}$ by a constant factor, but
  - Does not alter the growth rate of $\boldsymbol{T(n)}$
- The linear growth rate of the running time $\boldsymbol{T(n)}$ is an intrinsic property of algorithm ***arrayMax***



n	$\log n$	n	$n\log n$	n^2	n^3	2^n
4	2	4	8	16	64	16
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	4,094	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	$1.34 * 10^{154}$
1024	10	1,024	10,240	1,048,576	1,073,741,824	$1.79 * 10^{308}$

The Growth Rate of the Six Popular functions

Common growth rates



Simple Review

- Merge Sort
  - MERGE-SORT(A,p,r)
  - MERGE(A,p,q,r)
- Analysis of Merge Sort—Θ(nlgn), by
  - Picture of Recursion Tree
  - Telescoping
  - Mathematical Induction
- Asymptotic Growth
  - O-notation

Big-Oh Notation

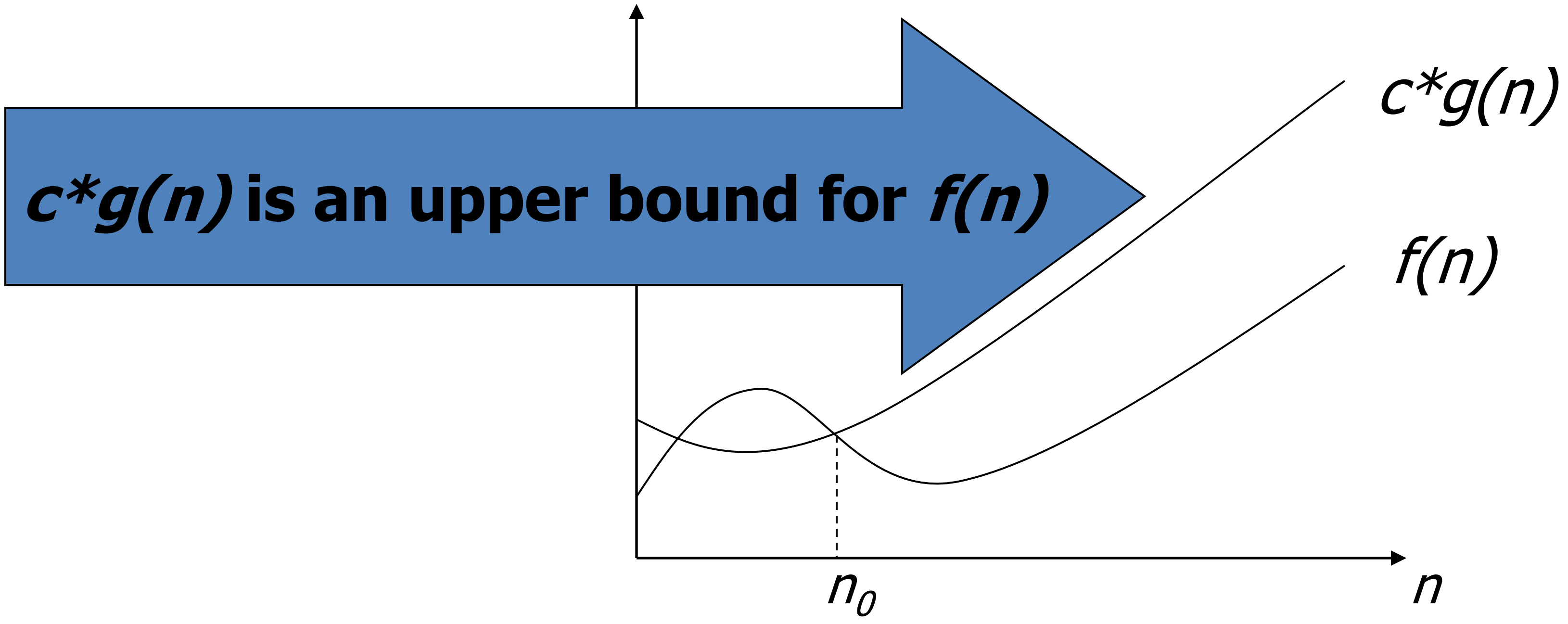
- To simplify the running time estimation, for a function *f(n),* we drop the leading constants and delete lower order terms.

Example: $10n^3+4n^2-4n+5$ is $O(n^3)$.

Big-Oh Defined

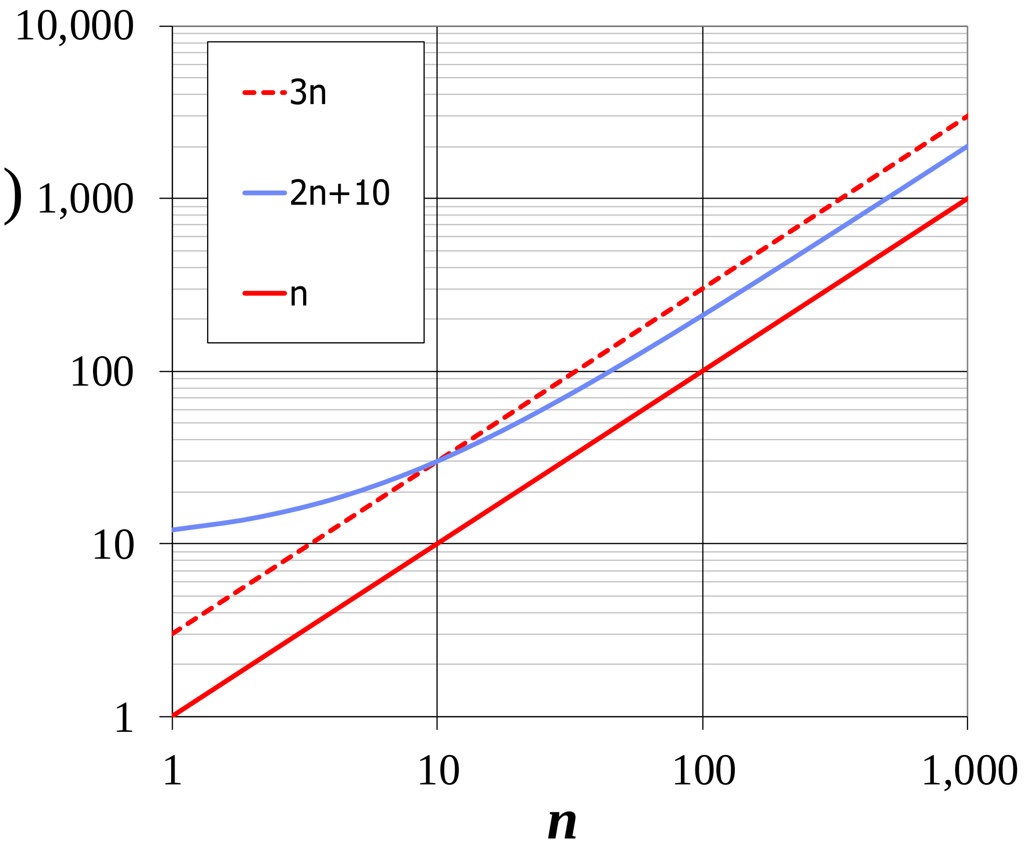
- The O symbol was introduced in 1927 to indicate relative growth of two functions based on asymptotic behavior of the functions now used to classify functions and families of functions

*f(n) = O(g(n)) if there are constants c and n0 such that f(n) < c*g(n) when* $n \geq n_0$



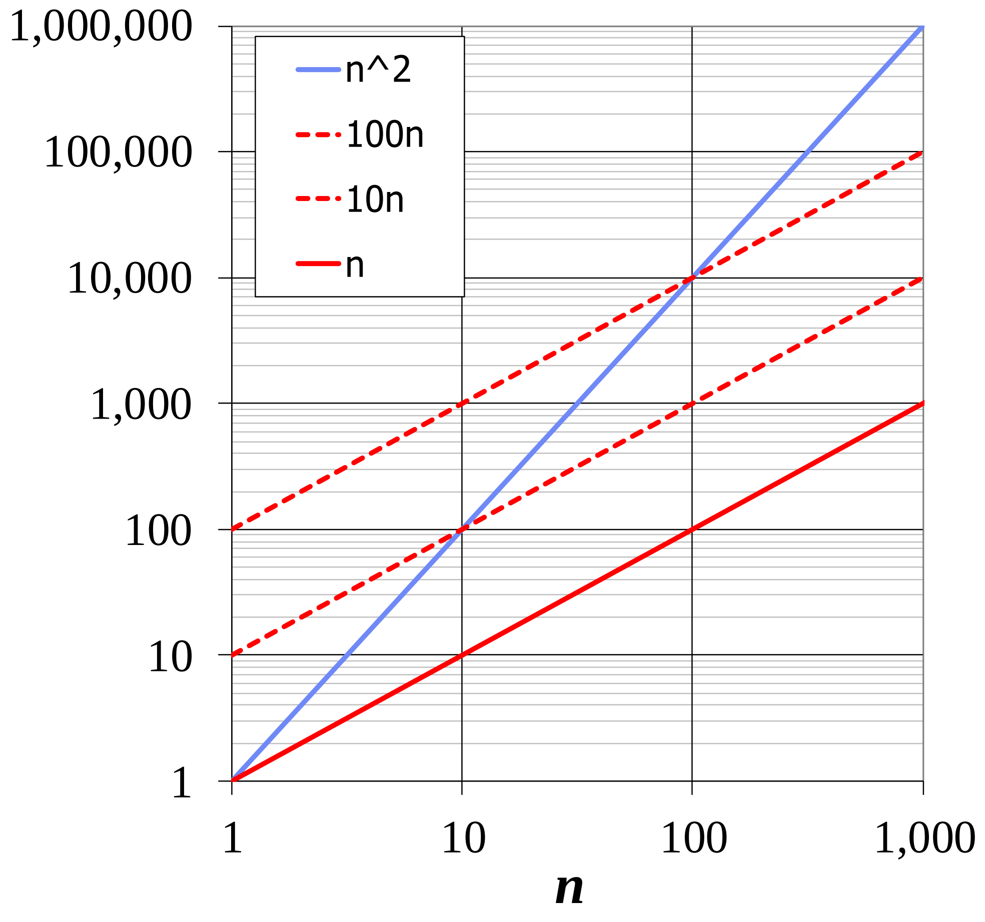
Big-Oh Example

- Example: $2\boldsymbol{n} + 10$ is $\boldsymbol{O}(\boldsymbol{n})$
  - $2\boldsymbol{n} + 10 \le \boldsymbol{cn}$
  - $(\boldsymbol{c} - 2)\,\boldsymbol{n} \ge 10$
  - $\boldsymbol{n} \ge 10/(\boldsymbol{c} - 2)$
  - Pick $\boldsymbol{c} = 3$ and $\boldsymbol{n_0} = 10$

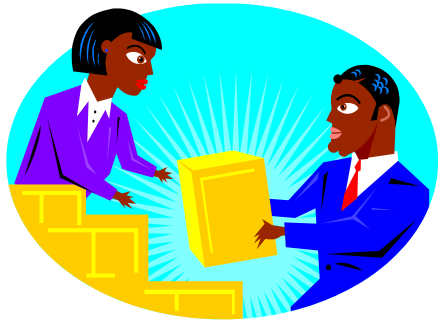


Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant
  - n^2 is $O(n^2)$.



More Big-Oh Examples



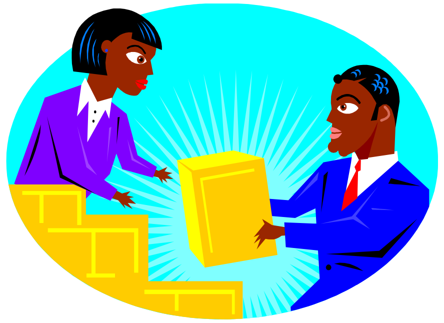
- 7n-2

  7n-2 is O(n)

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

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

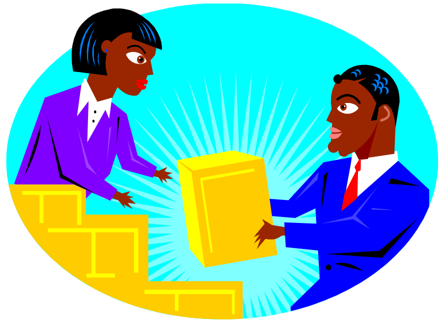
More Big-Oh Examples



- $3n^3 + 20n^2 + 5$

$3n^3 + 20n^2 + 5$ is $O(n^3)$
need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$
this is true for $c = 4$ and $n_0 = 21$

More Big-Oh Examples



- $3 \log n + 5$

  $3 \log n + 5$ is $O(\log n)$

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

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

More Big-Oh Examples

- 10000n + 5

10000n is O(n)

f(n)=10000n and g(n)=n, n_0 = 10000 and c = 1 then f(n) $\leq$ 1*g(n) where n $\geq$ n_0 and we say that f(n) = O(g(n))

More examples

- What about $f(n) = 4n^2$? Is it $O(n)$?
  - Find a c such that $4n^2 < cn$ for any $n > n0$
  - $4n<c$ and thus c is not a constant.
- $50n^3 + 20n + 4$ is $O(n^3)$
  - Would be correct to say is $O(n^3+n)$
    - Not useful, as n^3 exceeds by far n, for large values
  - Would be correct to say is $O(n^5)$
    - OK, but g(n) should be as closed as possible to f(n)
- $3\log(n) + \log(\log(n)) = O(\ ?\ )$

Big-Oh Examples

Suppose a program P is $O(n^3)$, and a program Q is $O(3^n)$, and that currently both can solve problems of size 50 in 1 hour. If the programs are run on another system that executes exactly 729 times as fast as the original system, what size problems will they be able to solve?

Big-Oh Examples

$n^3 = 50^3 * 729$ $3^n = 3^{50} * 729$

$n = \boxed{\sqrt[3]{50^3} * \sqrt[3]{729}}$ $n = \log_3 (729 * 3^{50})$

$n = \log_3(729) + \log_3 3^{50}$

$n = 50 * 9$ $n = 6 + \log_3 3^{50}$

$n = 50 * 9 = 450$ $n = 6 + 50 = 56$

- Improvement: problem size increased by 9 times for n^3 algorithm but only a slight improvement in problem size (+6) for exponential algorithm.

Problems

N^2	=	$O(N^2)$	true
$2N$	=	$O(N^2)$	true
N	=	$O(N^2)$	true
N^2	=	$O(N)$	false
$2N$	=	$O(N)$	true
N	=	$O(N)$	true

Big-Oh Rules

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Delete lower-order terms
  2. Drop leading constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement “$f(n)$ is $O(g(n))$” means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

Growth Rate of Running Time

- Consider a program with time complexity $O(n^2)$.
- For the input of size n, it takes 5 seconds.
- If the input size is doubled (2n), then it takes 20 seconds.

- Consider a program with time complexity O(n).
- For the input of size n, it takes 5 seconds.
- If the input size is doubled (2n), then it takes 10 seconds.

- Consider a program with time complexity $O(n^3)$.
- For the input of size n, it takes 5 seconds.
- If the input size is doubled (2n), then it takes 40 seconds.

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm ***arrayMax*** executes at most $6\boldsymbol{n} - 1$ primitive operations
  - We say that algorithm ***arrayMax*** "runs in $\boldsymbol{O}(\boldsymbol{n})$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Common time complexities

BETTER

$O(1)$	constant time
$O(\log n)$	log time
$O(n)$	linear time
$O(n \log n)$	log linear time
$O(n^2)$	quadratic time
$O(n^3)$	cubic time
$O(2^n)$	exponential time

WORSE

Important Series

$$S(N) = 1 + 2 + \ldots + N = \sum_{i=1}^{N} i = N(1+N)/2$$

- Sum of squares: $\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$ for large N

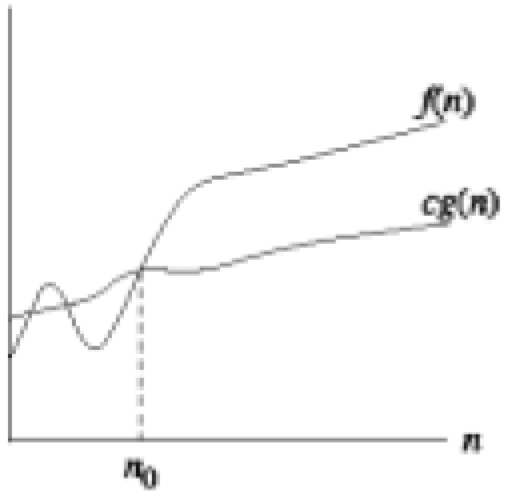
- Sum of exponents: $\sum_{i=1}^{N} i^k \approx \frac{N^{k+1}}{|k+1|}$ for large N and k $\neq$ -1

- Geometric series: $\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$
  - Special case when A = 2
    - $2^0 + 2^1 + 2^2 + \ldots + 2^N = 2^{N+1} - 1$

Ω-notation

- ***Ω(g(n)) = {f(n):*** **There exist positive constants *c* and n_0 such that** $0 \le cg(n) \le f(n)$ **for all** $n \ge n_0$**}**

  --We use *Ω*-notation to give a lower bound on a function.



Ω-notation

- **Examples:**

--$1/3n^2 - 3n \in \Omega(n^2)$ because $1/3n^2 - 3n \geq cn^2$ if $c \leq 1/3-3/n$ which holds for $c = 1/6$ and $n > 18$

--$k_1n^2+k_2n+k_3 \in \Omega(n^2)$

--$k_1n^2+k_2n+k_3 \in \Omega(n)$ (lower bound)

Ω-notation

- **Note:**

--when we say "the running time is *$\Omega(n^2)$*" we mean that the best-case running time is *$\Omega(n^2)$* – the worst case might be worse.

--Insertion-sort:

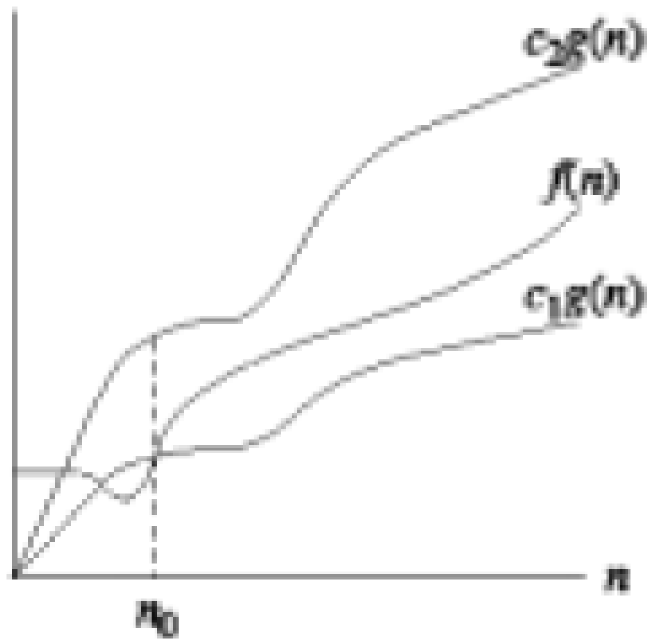
>> Best case: *$\Omega(n)$* – when the input array is already sorted.
>> Worst case: *$O(n^2)$*– when the input array is reverse sorted.
>>We can also say that the worst case running time is *$\Omega(n^2)$*

Θ-notation

- $\Theta(g(n)) = \{f(n)$: There exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$

  --We use Θ-notation to give a tight bound on a function.

  -- $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$



Θ-notation

- **Examples:**

-- $k_1n^2+k_2n+k_3 \in \Theta(n^2)$

--The worst case running time of insertion-sort is $\Theta(n^2)$

--$6nlgn + \sqrt{n}lg^2n=\Theta(nlgn)$

>>We need to find $c_1, c_2, n_0 > 0$ **such that** $c_1nlgn \leq 6nlgn + \sqrt{n}lg^2n \leq c_2nlgn$ **for** $n \geq n_0$**.**

>> $c_1nlgn \leq 6nlgn + \sqrt{n}lg^2n \rightarrow c_1 \leq 6 + lgn/\sqrt{n}$**, which is true if we choose** $c_1 = 6$ **and** $n_0 = 1$**.** $6nlgn + \sqrt{n}lg^2n \leq c_2nlgn \rightarrow 6 + lgn/\sqrt{n} \leq c_2$**, which is true if we choose** $c_2 = 7$ **and** $n_0 = 2$**. This is because** $lgn \leq \sqrt{n}$ **if** $n \geq 2$**. So** $c_1 = 6, c_2 = 7$ **and** $n_0 = 2$ **works.**

Θ-notation

- **Note:**

--We often think of $f(n) = O(g(n))$ **as corresponding to** $f(n) \leq g(n)$.

--Similarly, $f(n) = \Theta(g(n))$ **corresponds to** $f(n) = g(n)$

--Similarly, $f(n) = \Omega(g(n))$ **corresponds to** $f(n) \geq g(n)$

Asymptotic Notation in Equations

- **Used to replace functions of lower-order terms to simplify equations/expressions.**

- **For example,**

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

Or we can do the following: $4n^3 + 3n^2 + 2n + 1 = 4n^3 + f(n^2)$
Where $f(n^2)$ **simplifies the equation**