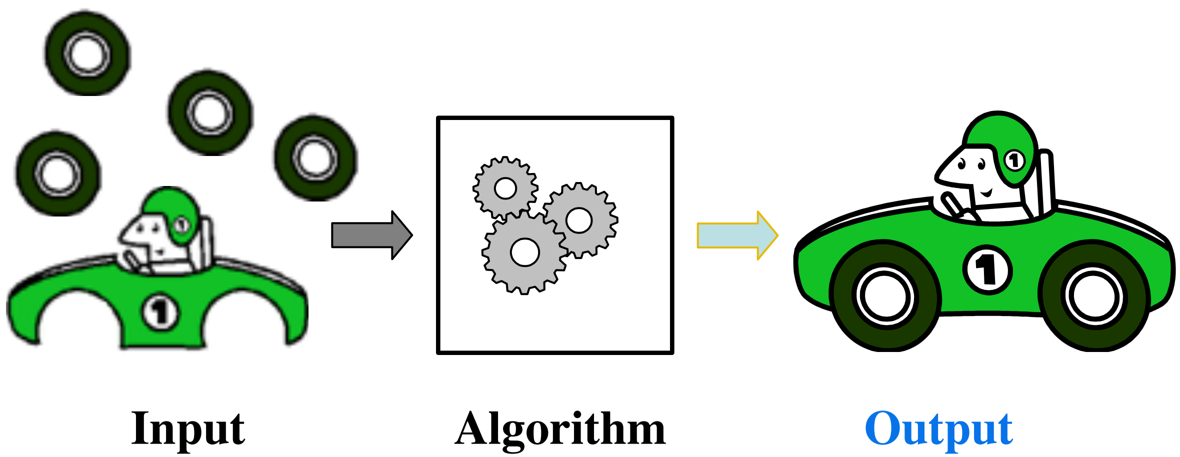
CSI2110

Data Structures and Algorithms

Prof. WonSook Lee

Algorithms



An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

Analyze an algorithm = determine its efficiency

Analyze an algorithm

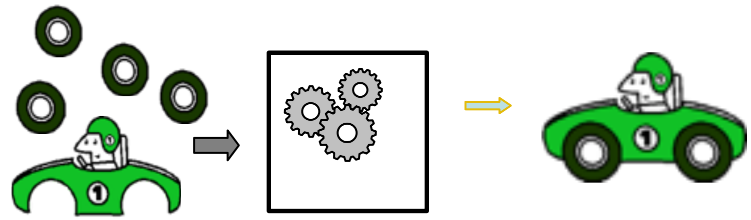
Analyze an algorithm = determine its efficiency

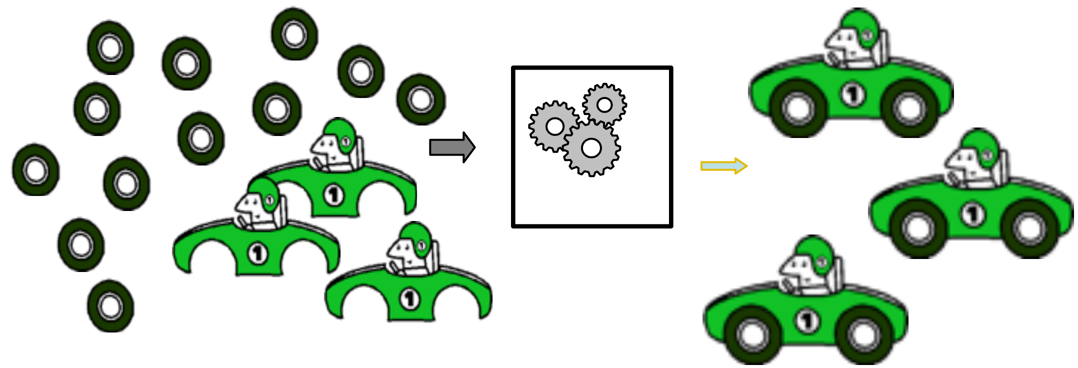
Efficiency ?

- Execution time …
- Memory …
- Quality of the result ….
- Simplicity ….

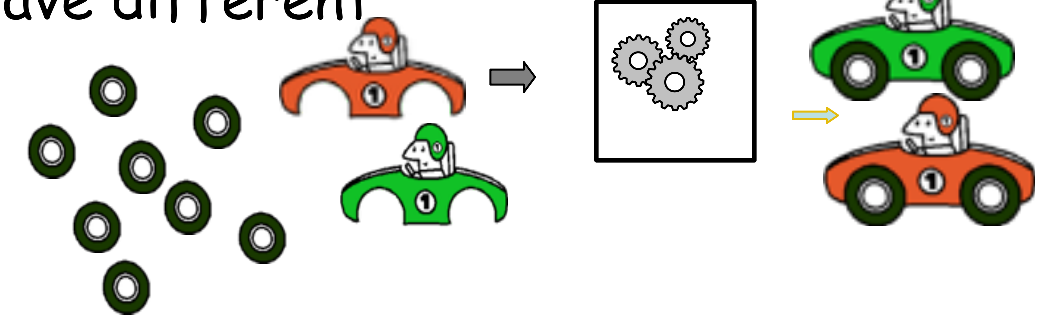
Running Time

The running time depends on the input size

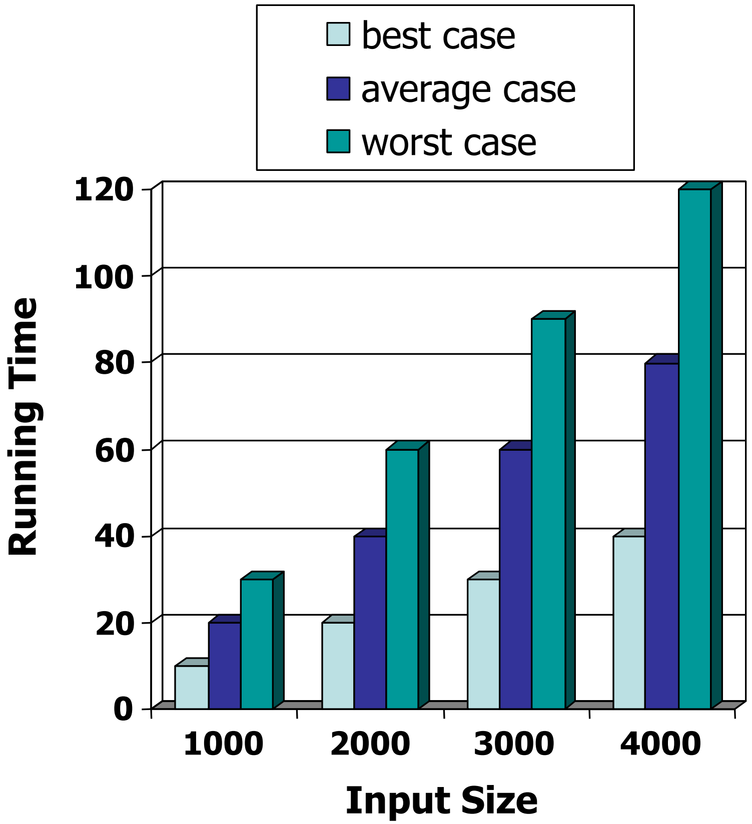




It also depends on the input data: Different inputs can have different running times

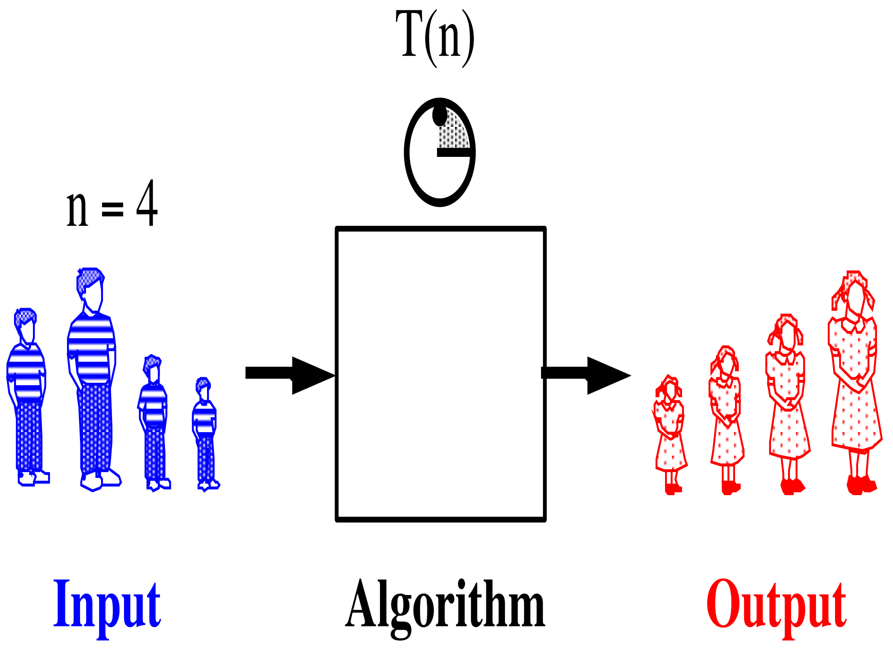


Running time



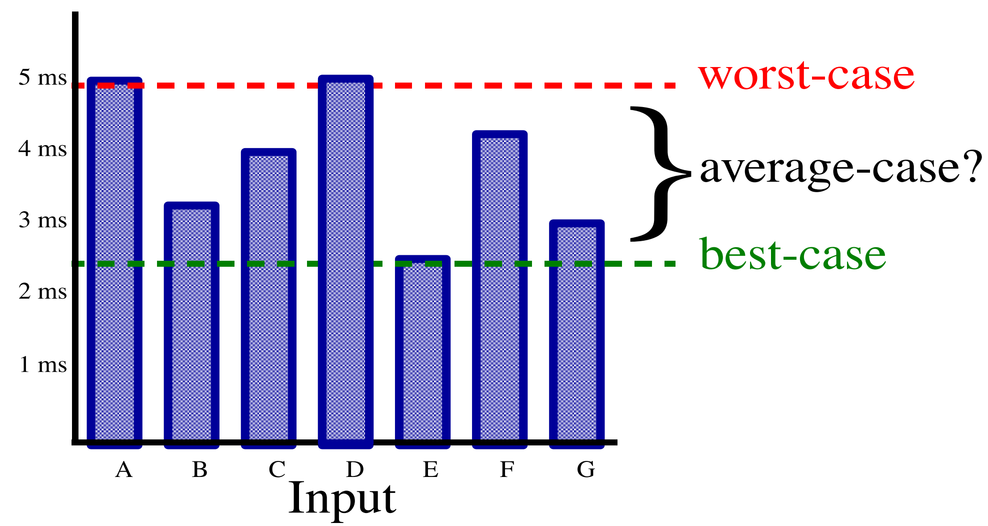
Analysis of Algorithms

- Running Time
- Upper Bounds
- Lower Bounds
- Examples
- Mathematical facts

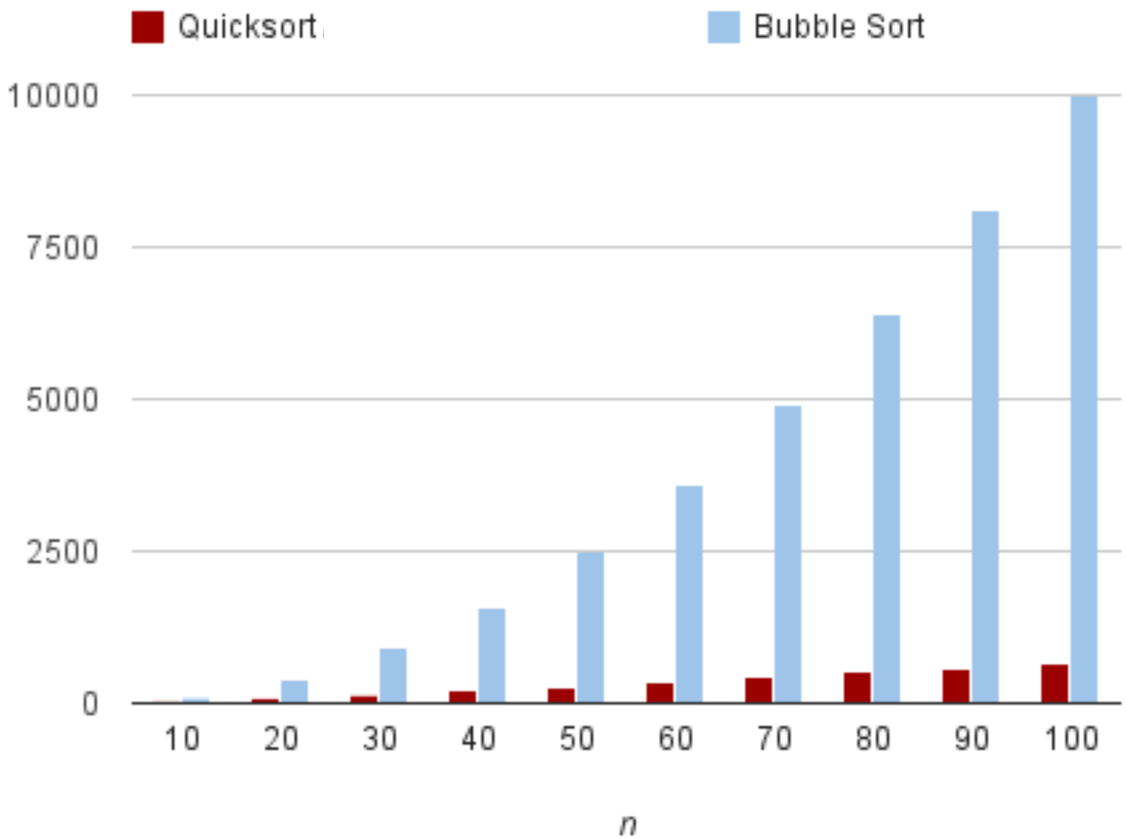


Average Case vs. Worst Case Running Time of an algorithm

- Finding the average case can be very difficult
- Knowing the worst-case time complexity can be important. We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games and robotics

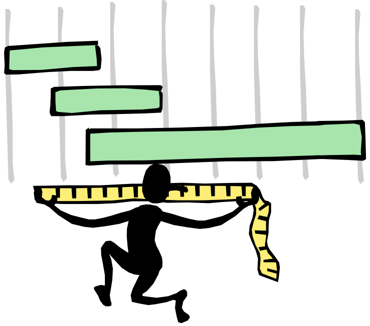


example: average case - different



example: worst case - same

Measuring the Running Time

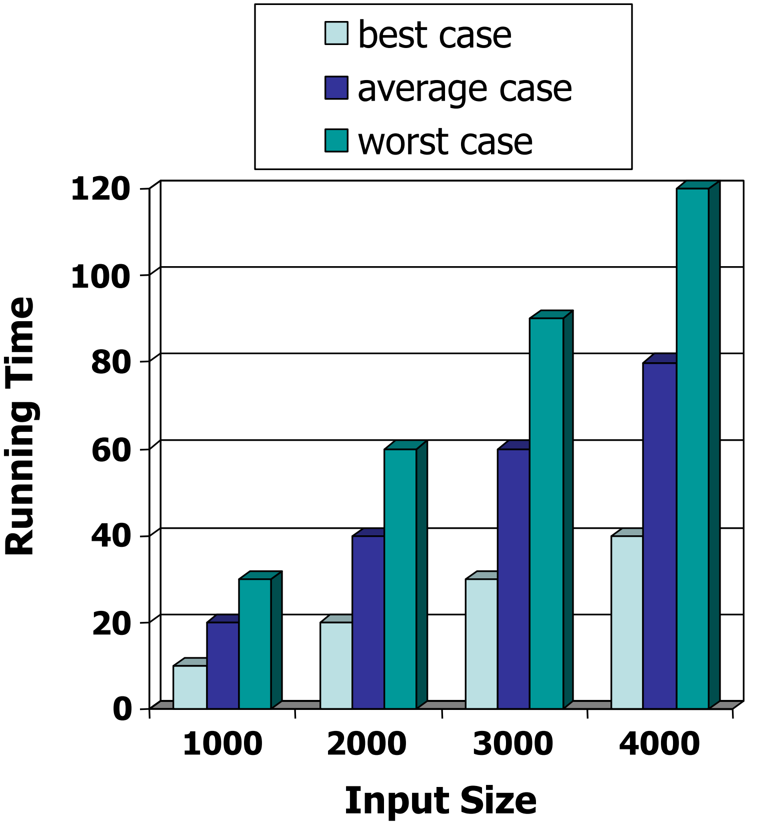


- How should we measure the running time of an algorithm?

- Approach 1: Experimental Study
  - Write a program which implement the algorithm
  - Run the program with all possible input sets of data of various size and contents
  - Use a method (system.currentTimeMillis()) to measure exact running time

Measuring the Running Time

- Approach 1: Experimental Study
- The measurement of the experiment can be like this…



Beyond Experimental Studies

- Experimental studies have several limitations:
  - need to implement
  - limited set of inputs
  - hardware and software environments.

Theoretical Analysis

- We need a general methodology which:
  - is independent of implementation.
    - Uses a high-level description of the algorithm
  - takes into account all possible inputs.
    - Characterizes running time as a function of the input size.
  - is independent of the hardware and software environment.

Analysis of Algorithms

- **Primitive Operations:** Low-level computations independent from the programming language can be identified in pseudocode.

- Examples:
  - calling a method and returning from a method
  - arithmetic operations (e.g. addition)
  - comparing two numbers, etc.

- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm.

Pseudo-code

- Mixture of natural language and high-level programming concept
  - to tell the general idea behind the data structure or algorithm implementation
- The pseudo-code is an description of algorithm which is
  - more structured than ordinary prose, but
  - less definite than programming languages

Pseudo-code

- Expressions: use standard mathematical symbols to describe Boolean and numerical expressions
  - use ← for allocations ("=" en Java)
  - use = for equal relation ("==" en Java)
- Method declaration
  - **Algorithm** nom(param1, param2)

Pseudo-code

- Programming element:
  - decision: if.. Then.. [else..]
  - Loop while: while…do
  - Loop repeat: repeat … until..
  - Loop for: for.. Do
  - Vector index: A[i]
- methods:
  - call: object method (args)
  - return: return value

Example:

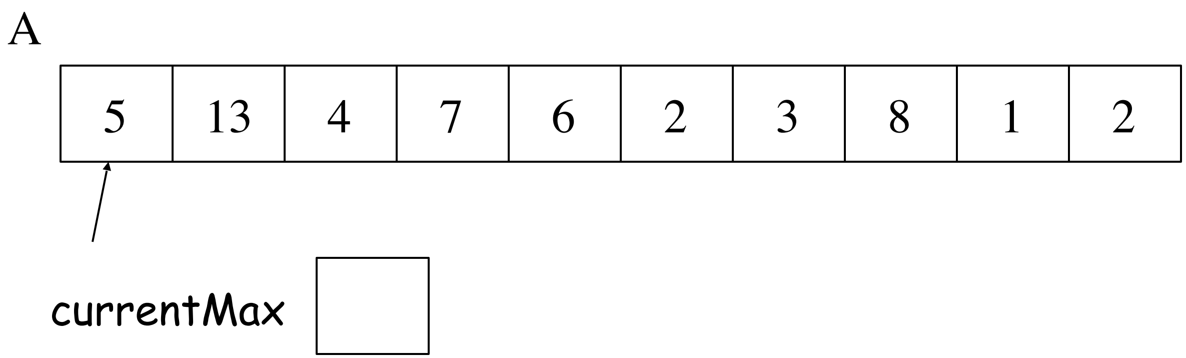
Find the maximum element of a vector (array)

Algorithm arrayMax(A, n):

input: A vector A containing n entries

output: the maxim element of A

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



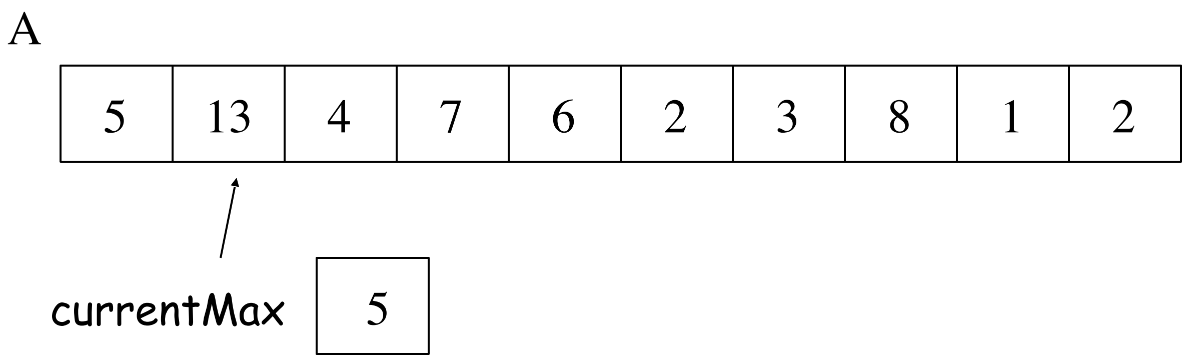

```
currentMax ← A[0]
for i ← 1 to n -1 do
if  currentMax < A[i] then
            currentMax ← A[i]

return currentMax
```

A

5	13	4	7	6	2	3	8	1	2

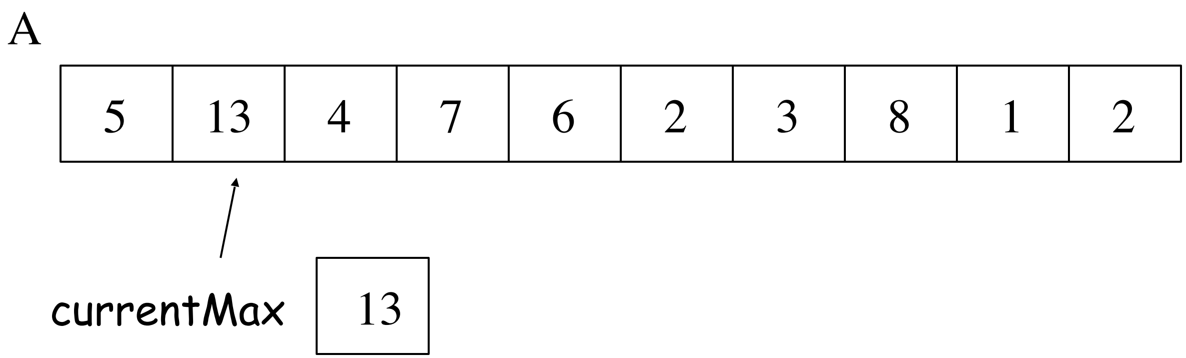
currentMax	5

```
currentMax ← A[0]
for i ← 1 to n -1 do
if  currentMax < A[i] then
            currentMax ← A[i]


return currentMax
```




```
currentMax ← A[0]
for i ← 1 to n -1 do
if  currentMax < A[i] then
            currentMax ← A[i]


return currentMax
```

What are the primitive operations to count



Comparisons
Assignments to currentMax

```
currentMax ← A[0]
for i ← 1 to n -1 do
if currentMax < A[i] then
        currentMax ← A[i]

return currentMax
```

```
currentMax ← A[0]                      1 assignment
for i ← 1 to n -1 do
if currentMax < A[i] then              n-1 comparisons
        currentMax ← A[i]              n-1   assignments (worst case)

return currentMax
```

5	7	8	10	11	12	14	16	17	20

In the best case ?

15	1	12	3	9	7	6	4	2	11

```
currentMax ← A[0]                 1 assign
for i ← 1 to n -1 do
if currentMax < A[i] then         n-1 comparaisons
        currentMax ← A[i]         0 assign

return currentMax
```

Summarizing:

Worst Case:	Best Case:
n-1 comparisons	n-1 comparisons
n assignments	1 assignment

Another Example

Looking for the rank of an element in *A*
(size of *A* is size*A*)

i ← 0 - - - - - - - - - - - - - - → 1 assignment

while (A[i] ≠ element)

i ← i+1 - - - - - - - - - - - - - - → size*A* checks & assignment (if we are not lucky)

return i

Worst Case

Upper Bound

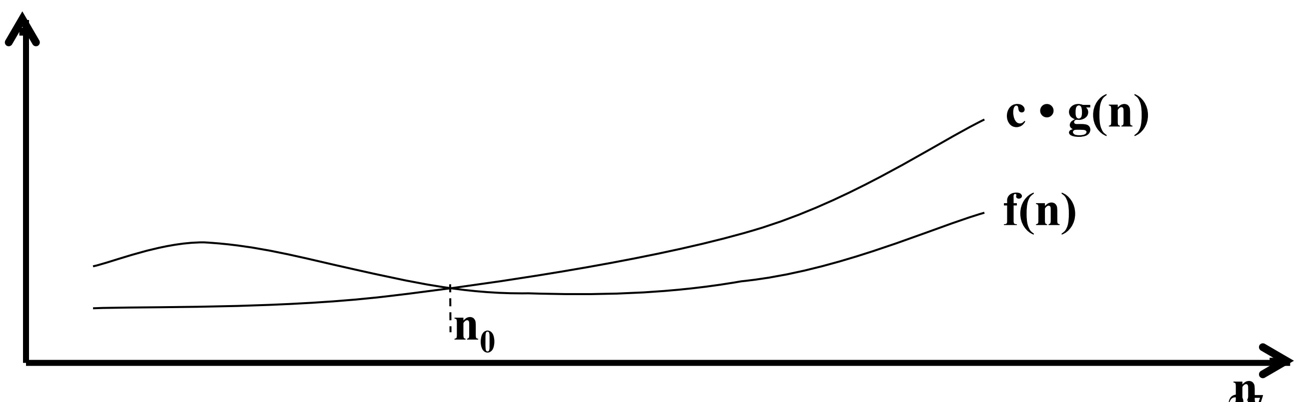
The "Big-Oh" Notation:

- given functions $f(n)$ and $g(n)$, we say that

$$f(n) \text{ is } O(g(n))$$

if and only if there are positive constants c and n_0 such that

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



prove that $f(n) \leq c\, g(n)$ for some $n \geq n_0$

An Example

$$f(n) = 60n^2 + 5n + 1 \qquad g(n) = n^2$$

prove that $f(n) \leq c\, n^2$

$$\leq$$

$$\underbrace{60n^2 + 5n^2 + n^2}_{} \qquad \text{for } n \geq 1$$

$$= 66n^2$$

$$\Rightarrow \quad c = 66 \qquad n_0 = 1$$

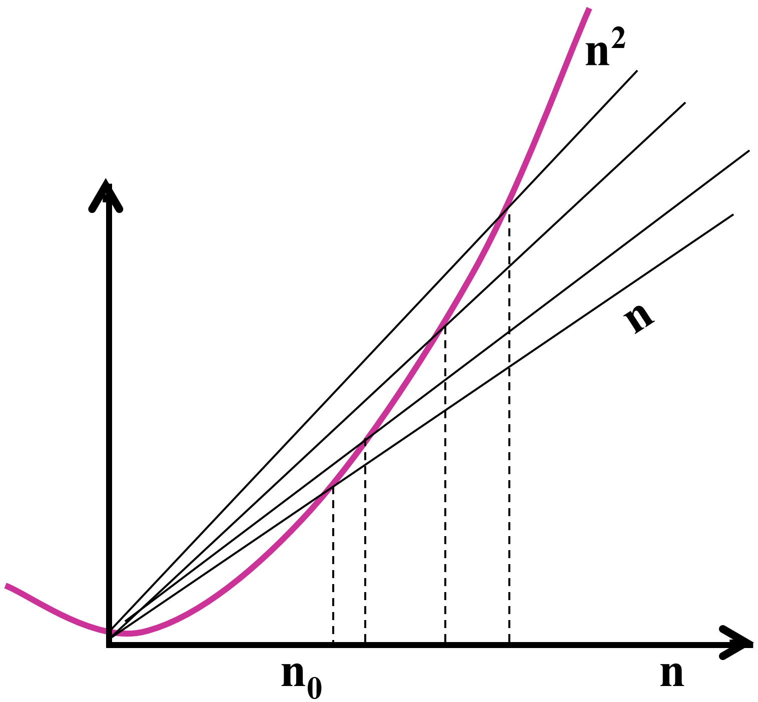
$$f(n) \leq c\, n^2 \quad \forall \quad n \geq n_0$$

$\Rightarrow$ **$f(n) = O(n^2)$**

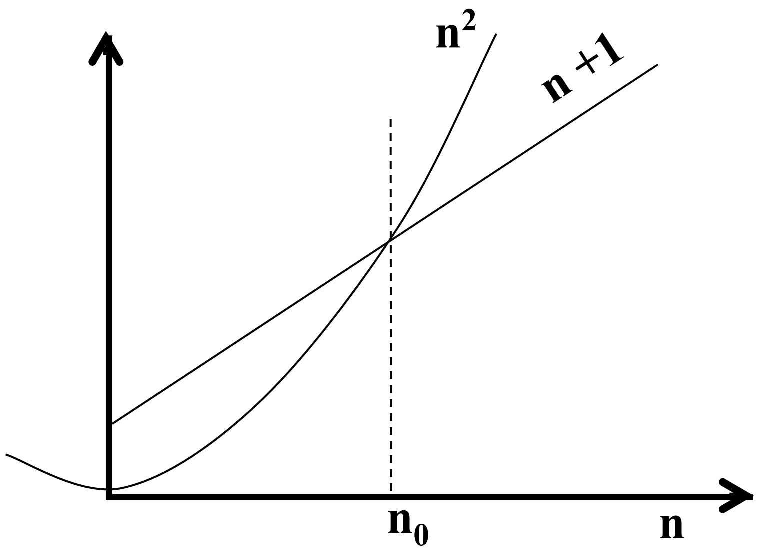
On the other hand…

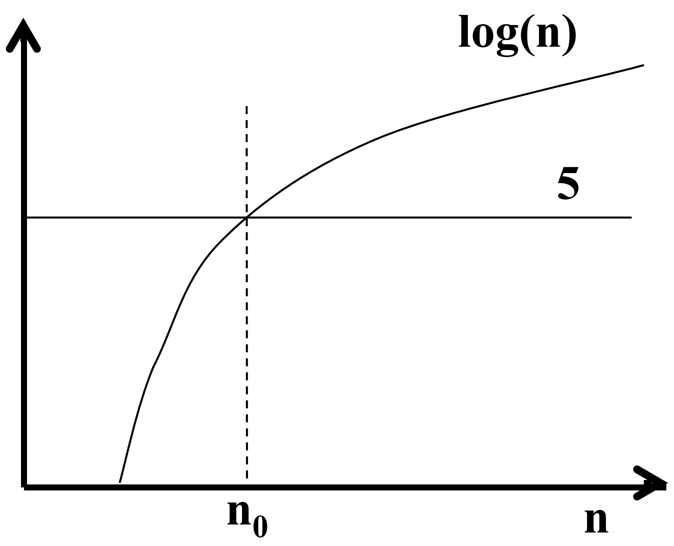
n^2 is not $O(n)$ because there is no c and n_0 such that: $n^2 \le cn$ for $n \ge n_0$

(no matter how large a c is chosen there is an n big enough that $n^2 > c\,n$) .



$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$ … remember !!





n =	2	16	256	1024
log log n	0	2	3	3.32
log n	1	4	8	10
n	2	16	256	1024
n log n	2	64	448	10 200
n^2	4	256	65 500	$1.05 * 10^6$
n^3	8	4 100	16 800 800	$1.07 * 10^9$
2^n	4	35 500	$11.7 * 10^6$	$1.80 * 10^{308}$

Asymptotic Notation (cont.)

- Note: Even though it is correct to say
  - "$7n - 3$ is $O(n^3)$",
- a better statement is
  - "$7n - 3$ is $O(n)$", that is,
- one should make the approximation as tight as possible

Theorem:

If $g(n)$ is $O(f(n))$, then for any constant $c > 0$

$g(n)$ is also $O(c\ f(n))$

Theorem:

$O(f(n) + g(n)) = O(\max(f(n), g(n)))$

Ex 1:

$$2n^3 + 3n^2 = O(\max(2n^3, 3n^2))$$
$$= O(2n^3) = O(n^3)$$

Ex 2:

$$n^2 + 3 \log n - 7 = O(\max(n^2, 3 \log n - 7))$$
$$= O(n^2)$$

Simple Big Oh Rule:

Drop lower order terms and constant factors

$7n-3$ is $O(n)$

$8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

$12n^3 + 5000n^2 + 2n^4$ is $O(n^4)$

Other Big Oh Rules:

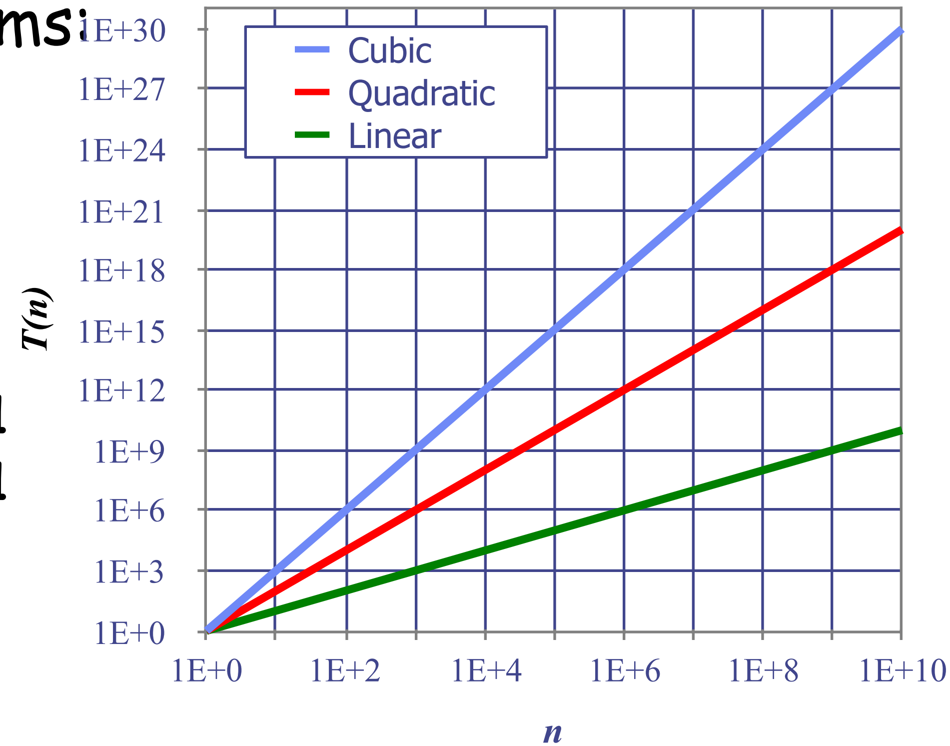
- Use the smallest possible class of functions
  - Say "2**n** is **O**(**n**)" instead of "2**n** is **O**(**n**^{2})"

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

    "3**n** + 5 is **O**(3**n**)"

Asymptotic Notation (terminology)

- Special classes of algorithms:
  - constant: $O(1)$
  - logarithmic: $O(\log n)$
  - linear: $O(n)$
  - quadratic: $O(n^2)$
  - cubic: $O(n^3)$
  - polynomial: $O(n^k)$, $k \geq 1$
  - exponential: $O(a^n)$, $n > 1$



Asymptotic Analysis and execution time

- Use the Big-O notation to indicate the number of primitive operations executed according to the entry size
- For example, we say that algorithm arrayMax has an execution time O(n)
- While comparing the asymptotic execution times
  - O(log n) is better than O(n)
  - O(n) is better than O(n^2)
  - **$\log n << n^{-2} << n << n \log n << n^2 << n^3 << 2^n$**

Example of Asymptotic Analysis

An algorithm for computing prefix averages

The $\boldsymbol{i}$-th prefix average of an array $\boldsymbol{X}$ is average of the first $(\boldsymbol{i} + 1)$ elements of $\boldsymbol{X}$

$\boldsymbol{A}[\boldsymbol{i}] = \boldsymbol{X}[0] + \boldsymbol{X}[1] + \ldots + \boldsymbol{X}[\boldsymbol{i}]$

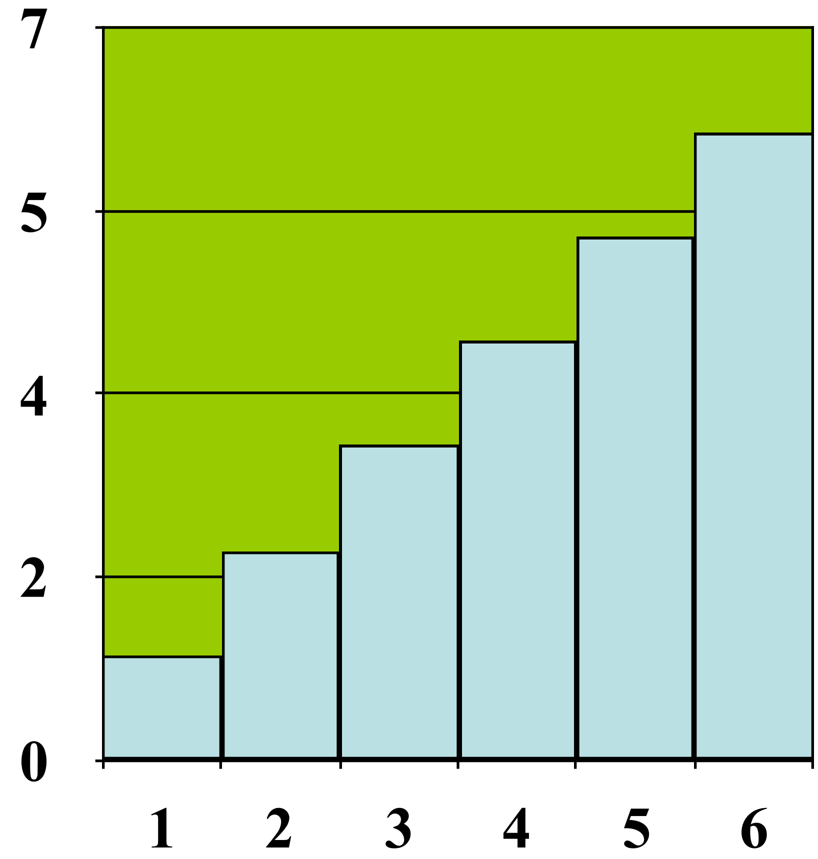
Example of Asymptotic Analysis

Algorithm ***prefixAverages1***(X, n)

Input array X of n integers
Output array A of prefix averages of X #operations

$A \leftarrow$ new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	
$\quad s \leftarrow X[0]$	n
$\quad$ **for** $j \leftarrow 1$ **to** i **do**	
$\qquad s \leftarrow s + X[j]$	$1 + 2 + \ldots + (n - 1)$
$\quad A[i] \leftarrow s / (i + 1)$	n
return A	1

- The running time of **prefixAverages1** is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm **prefixAverages1** runs in $O(n^2)$ time



Another Example

- A better algorithm for computing prefix averages:

Algorithm prefixAverages2(X):

Input: An n-element array X of numbers.

Output: An n -element array A of numbers such that A[i] is the average of elements X[0], ... , X[i].

Let X be an array of $\boldsymbol{n}$ numbers. # operations

$s \leftarrow 0$	1
for $i \leftarrow 0$ **to** $\boldsymbol{n}$ **do**	n
$\quad s \leftarrow s + X[\boldsymbol{i}]$	n
$\quad A[\boldsymbol{i}] \leftarrow s/(\boldsymbol{i}+ 1)$	n
return array A	1

O(n) time

Lower Bound

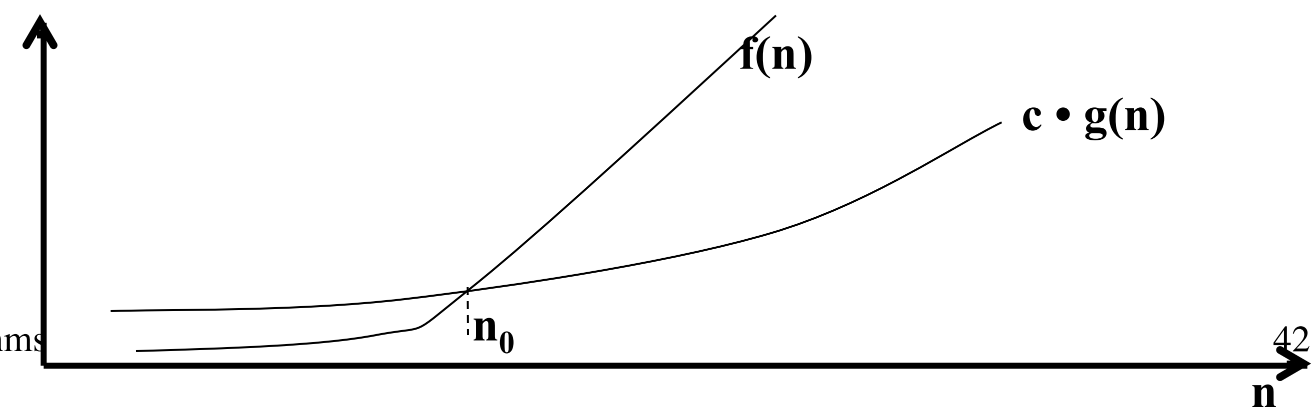
… is big omega …

f(n) is $\Omega(g(n))$

if there exist $c > 0$ and $n_0 > 0$ such that

$f(n) \geq c \cdot g(n)$ for all $n \geq n_0$

(thus, f(n) is $\Omega(g(n))$ iff g(n) is $O(f(n))$)



Tight Bound

... is big theta ...

$g(n)$ is $\Theta(f(n))$

<===>

if $g(n) \in O(f(n))$

AND

$f(n) \in O(g(n))$

is an element of (set membership)

Mathematical notation instead of "is"

An Example

We have seen that

$f(n) = 60n^2 + 5n + 1$ is $O(n^2)$

but $60n^2 + 5n + 1 \geq 60n^2$ for $n \geq 1$

So: with $c = 60$ and $n_0 = 1$

$f(n) \geq c \bullet n^2$ for all $n \geq 1$ $\Rightarrow$ $f(n)$ is $\Omega(n^2)$

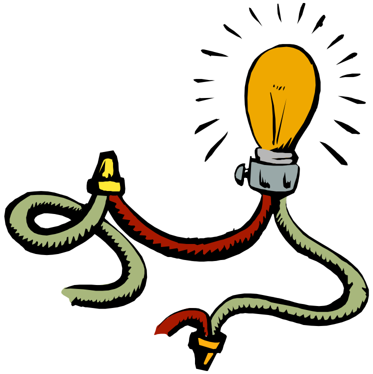
Therefore:

$$\boxed{\begin{array}{l} f(n) \text{ is } O(n^2) \\ \quad \text{AND} \\ f(n) \text{ is } \Omega(n^2) \end{array}}$$

$$\Downarrow$$

$f(n)$ is $\Theta(n^2)$

Intuition for Asymptotic Notation



Big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

Math You Need to Review

Logarithms and Exponents

properties of logarithms:

$\log_b(xy) = \log_b x + \log_b y$

$\log_b (x/y) = \log_b x - \log_b y$

$\log_b xa = a\log_b x$

$\log_b a = \log_x a / \log_x b$

properties of exponentials:

$a^{(b+c)} = a^b a^c$

$a^{bc} = (a^b)^c$

$a^b / a^c = a^{(b-c)}$

$b = a^{\log_a b}$

$b^c = a^{c*\log_a b}$

More Math to Review

- Floor: $\lfloor x \rfloor$ = the largest integer $\leq x$ $\lfloor 2.3 \rfloor = 2$
- Ceiling: $\lceil x \rceil$ = the smallest integer $\geq x$ $\lceil 2.3 \rceil = 3$
- Summations:
  - General definition:

$$\sum_{t=s}^{t} f(i) = f(s) + f(s+1) + f(s+2) + .. + f(t)$$

  - where f is a function, s is the starting index, and t is the ending index

More Math to Review

- Arithmetic Progression : f(i) = i a

$$S = \sum_{i=0}^{n} di = 0 + d + 2d + \ldots + nd$$

$$= nd+(n-1)d+(n-2)d + \ldots + 0$$

$$2S = nd + nd + nd + \ldots + nd$$

$$= (n+1)\, nd$$

$$S = d/2\; n(n+1)$$

for d=1, $S = 1/2\; n(n+1)$

More Math to Review

- Geometric Sum : $f(i) = a^i$
- The geometric progressions have an exponential growth

$$S = \sum_{i=0}^{n} r^i = 1 + r + r^2 + \ldots + r^n$$

$$rS = \quad r + r^2 + \ldots + r^n + r^{n+1}$$

$$rS - S = (r-1)S = r^{n+1} - 1$$

$$S = (r^{n+1}-1)/(r-1)$$

If $r=2$, $S = (2^{n+1}-1)$

"Dear Andy: How have you been?
Your mother and I are fine. We miss you.
Please sign off your computer and come
downstairs for something to eat. Love, Dad."