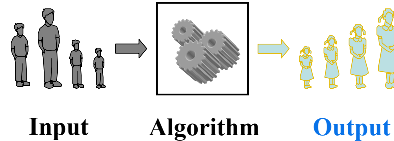
Analysis of Algorithms

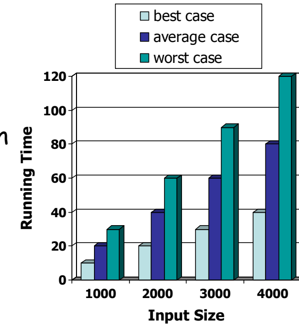
- Running Time
- Upper Bounds
- Lower Bounds
- Examples
- Mathematical facts



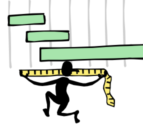
An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

Running Time of an algorithm

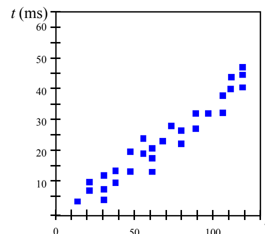
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



Measuring the Running Time



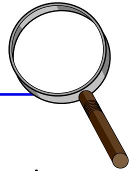
- How should we measure the running time of an algorithm?
- Approach 1: Experimental Study



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 that:

Uses a high-level description of the algorithm (independent of implementation).

Characterizes running time as a function of the input size.

Takes into account all possible inputs.

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.

Example:

Algorithm arrayMax(A, n):
Input: An array A storing n integers.
Output: The maximum element in A.
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**
currentMax ← A[i] - - → n-1 check
n-1 assignments (if we are not lucky)
return currentMax - - - - → 1 return value

Looking for the rank of an element in A of size sizeA

```
i ← 0                  ---------------> 1 assignment
while (A[i] ≠ element)
        i ← i+1         ---------------> sizeA checks & assignment
                                         (if we are not lucky)
return i
```

Worst Case

Big-Oh

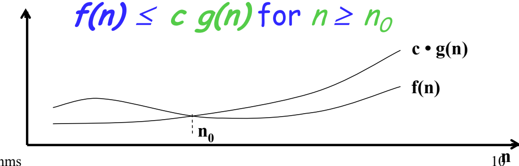
(upper bound)

- 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

$$f(n) \le c\, g(n) \text{ for } n \ge n_0$$



An Example

prove that $f(n) \le c\, g(n)$ for all $n \ge n_0$

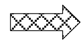
$f(n) = 60n^2 + 5n + 1$ $\qquad g(n) = n^2$

$\le$

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

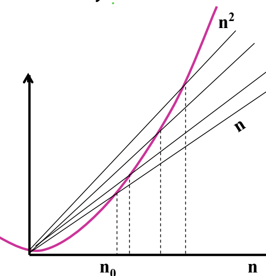
$60n^2 + 5n^2 + n^2$ $\qquad$ for $n \ge 1$

$= 66n^2$



$c = 66 \qquad n_0 = 1$

$f(n) \le c\, n^2 \qquad \forall\ n \ge n_0$



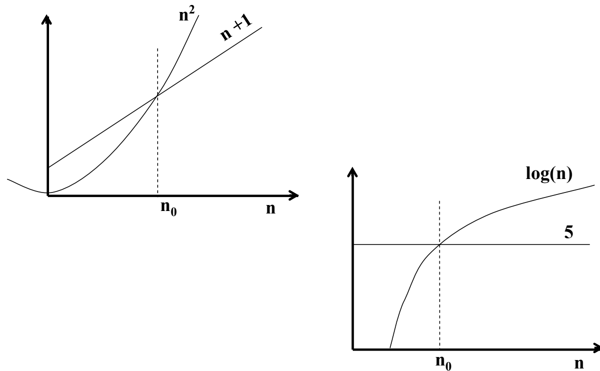
On the other hand…

n^2 is not $O(n)$ because there is no c and n_0 such that:

$$n^2 \le cn \text{ for } n \ge n_0$$

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

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) \ldots$



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 "$2n$ 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 $O(3n)$"

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:	$O(n^k)$, $k > 0$
exponential:	$O(a^n)$, $n > 1$

Example of Asymptotic Analysis

An algorithm for computing prefix averages

The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X.

That is, $A[i] = X[0] + X[1] + \ldots + X[i]$

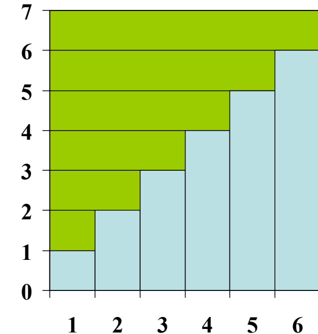
Example of Asymptotic Analysis

Algorithm *prefixAverages1*(*X, n*)

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

	#operations
A ← new array of ***n*** integers	***n***
for ***i*** ← 0 **to** ***n*** − 1 **do**	***n***
s ← ***0***	***n***
for ***j*** ← 0 **to** ***i*** **do**	1 + 2 + …+ ***n***
s ← ***s*** + ***X***[***j***]	1 + 2 + …+ ***n***
A[***i***] ← ***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 ***n*** numbers.	# operations
s← 0	1
for i ← 0 **to** *n-1* **do**	n
s ← s + X[***i***]	n
A[***i***] ← s/(***i***+ 1)	n
return array A	1

O(n) time

big-Omega

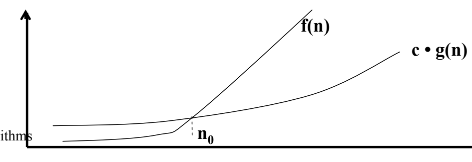
(lower bound)

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 $O(f(n))$)



big-Theta

... is big theta ...

g(n) is Θ(f(n))

<===>

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

AND

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

Big Theta Θ notation allows us to say that two functions grow at the same rate, up to constant factors.

When we say g(n) is Θ(f(n)), it means

there are two constants c_1 and c_2, and $n_0 \geq 1$ such that

$$c_1 f(n) \leq g(n) \leq c_2 f(n), \text{ for } n \geq n_0.$$

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 $n_0 = 1$

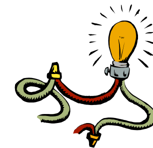
$f(n) \geq c \cdot n^2$ for all $n \geq 1$

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

f(n) is $O(n^2)$
AND
f(n) is $\Omega(n^2)$

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 (Appendix A)

properties of logarithms:

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

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

$\log_b x^a = a\log_b x$

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

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 $\leq x$
- Ceiling: $\lceil x \rceil$ = the smallest integer $\geq x$
- Summations: (see Appendix A)
- Geometric progression: (see Appendix A)

More Math to Review
Arithmetic Progression

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

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

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

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

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

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

More Math to Review
Geometric Progression

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

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

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

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

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