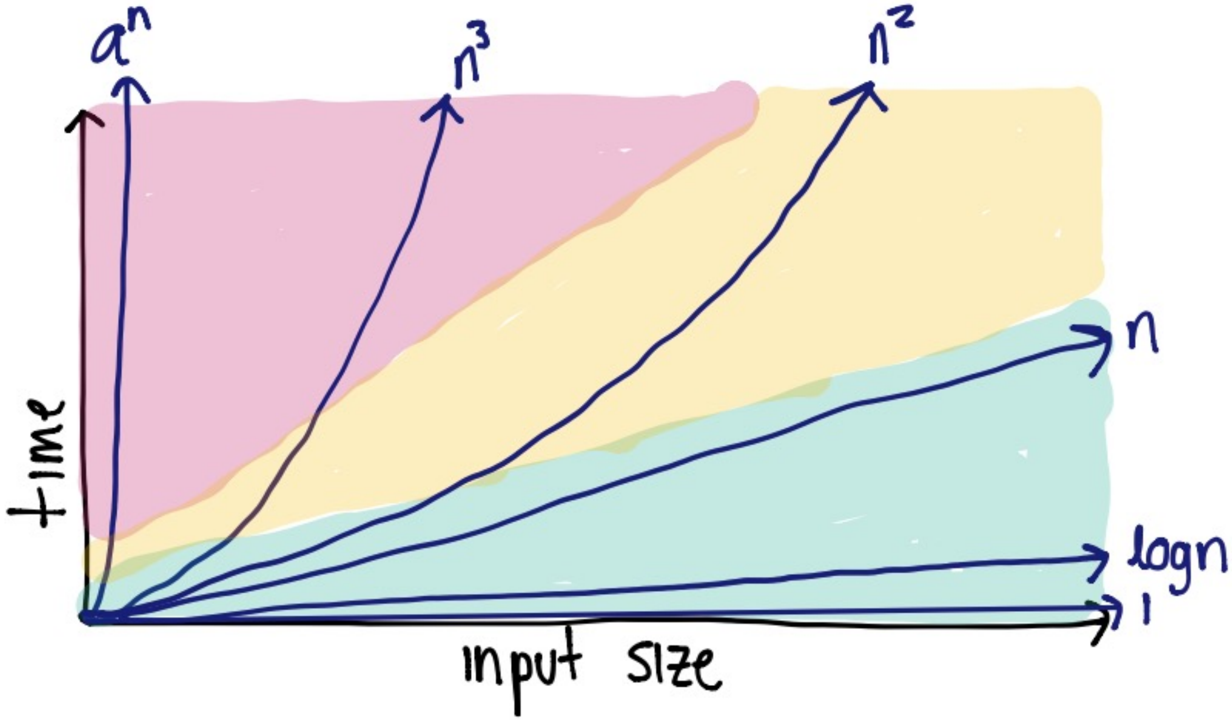
Growth of functions

CLRS 3.1 & 3.2



Algorithmic Purpose

- To determine the worst-case running time, we count the maximum number of instructions an algorithm requires, as a function of the input size

Algorithm ***arrayMax***(***A***, ***n***)	# instructions
currentMax = *A*[1]	2
for ***i*** = 2 **to** *n* **do**	2 + ***n***
if *A*[*i*] > *currentMax* **then**	2(***n*** - 1)
currentMax = *A*[*i*]	2(***n*** - 1)
{ increment counter ***i*** }	2(***n*** - 1)
return *currentMax*	1

	7*n* - 1

- However, rather than expressing the exact number of instructions, we use **asymptotic complexity** to express it in terms of growth rate.
  - ”The algorithm *arrayMax* has a worst-case running time of *O(n)*."

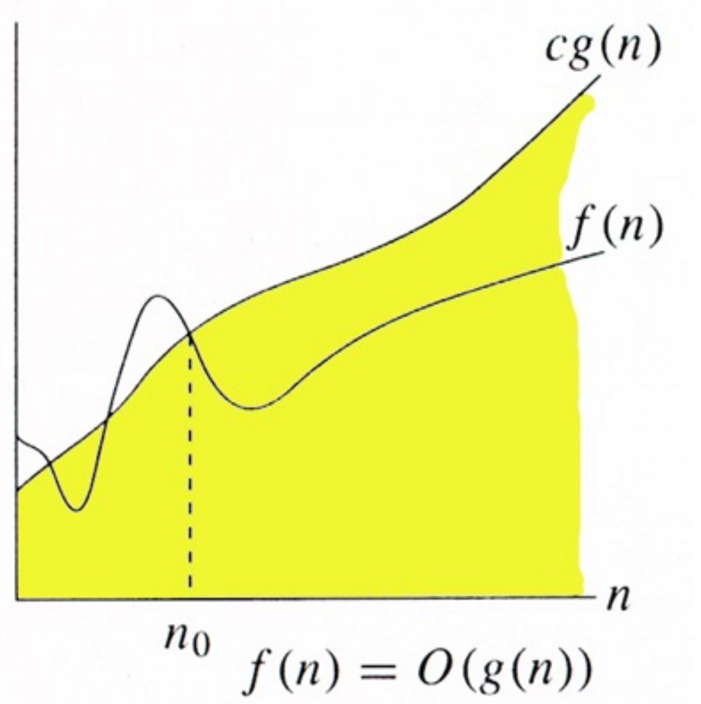
Asymptotic Complexity

- Worst case running time of an algorithm as a function of input size n **for large n.**
- Expressed using only the highest-order term in the expression for the exact running time
  - Instead of exact running time, say $O(n^2)$
- Written using asymptotic notation (***O***, Ω, Θ, *o*, ω)
  - Ex: $f(n) = O(n^2)$
  - Describes how $f(n)$ grows in comparison to n^2
- The notations describe different rate-of-growth relations between the defining function and the defined **set** of functions
- Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs

O-notation ("Big Oh")

For functions $g(n)$, we define $O(g(n))$ as the set:

$$O\big(g(n)\big) = \{\, f(n) : \exists \text{ positive constants } c \text{ and } n_0, \text{ such that } \forall n \geq n_0, \text{ we have } 0 \leq f(n) \leq cg(n) \,\}$$



- Technically, we would write $f(n) \in O\big(g(n)\big)$
- Often, you will see equivalently the notation $f(n) = O(g(n))$

- Intuitively: $O\big(g(n)\big)$ is the set of functions whose *rate of growth* is the **same as or lower** than $g(n)$

- $g(n)$ is an **asymptotic upper bound** for $f(n)$

O-notation: Examples

For functions $g(n)$, we define $O(g(n))$ as the set:

$$O\big(g(n)\big) = \{\, f(n) : \exists \text{ positive constants } c \text{ and } n_0, \text{ such that } \forall n \geq n_0, \text{ we have } 0 \leq f(n) \leq cg(n) \,\}$$

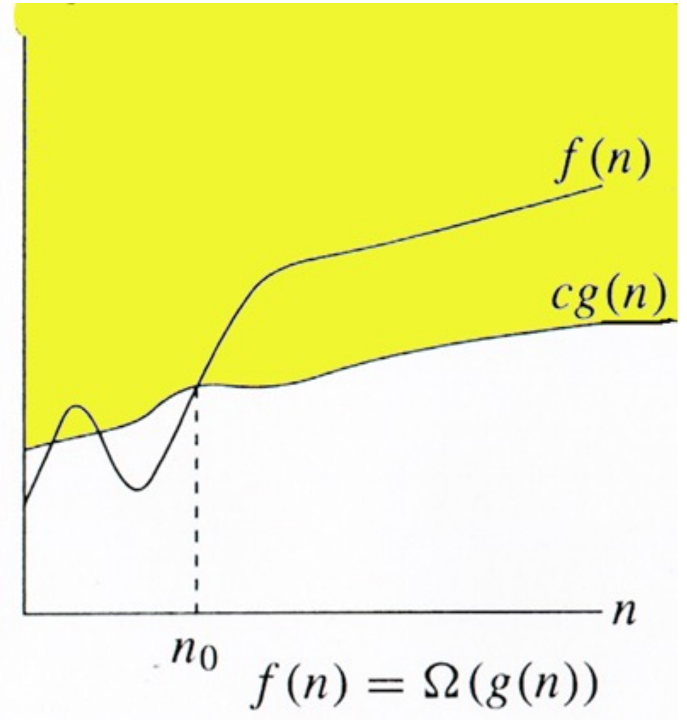
- $O(n)$ includes:
  - $f(n) = 2n + 10$
  - $f(n) = n + 1$
  - $f(n) = 10000n$
  - $f(n) = 10000n + 300$
- $O(n^2)$ includes:
  - $f(n) = n^2 + 1$
  - $f(n) = n^2 + n$
  - $f(n) = 10000n^2 + 10000n + 300$
  - $f(n) = n^{1.99}$

- The function n^2 is **not** $O(n)$
  - the inequality $n^2 \leq cn$ cannot be satisfied since c is a constant
- Technically, n is $O(n^2)$, but…
  - We **would not use this** to express the run time of an algorithm
  - We want to use tight upper bounds to be precise

Ω-notation ("Big Omega")

For functions $g(n)$, we define $\Omega(g(n))$ as the set:

$$\Omega(g(n)) = \{\, f(n) : \exists \text{ positive constants } c \text{ and } n_0, \text{ such that } \forall n \geq n_0, \text{ we have } 0 \leq cg(n) \leq f(n) \,\}$$



- Intuitively: $\Omega(g(n))$ is the set of functions whose *rate of growth* is the **same as or higher** than $g(n)$

- $g(n)$ is an **asymptotic lower bound** for $f(n)$

Ω-notation: Examples / notes

For functions $g(n)$, we define $\Omega(g(n))$ as the set:

$$\Omega(g(n)) = \{\, f(n) : \exists \text{ positive constants } c \text{ and } n_0, \text{ such that } \forall n \geq n_0, \text{ we have } 0 \leq cg(n) \leq f(n) \,\}$$

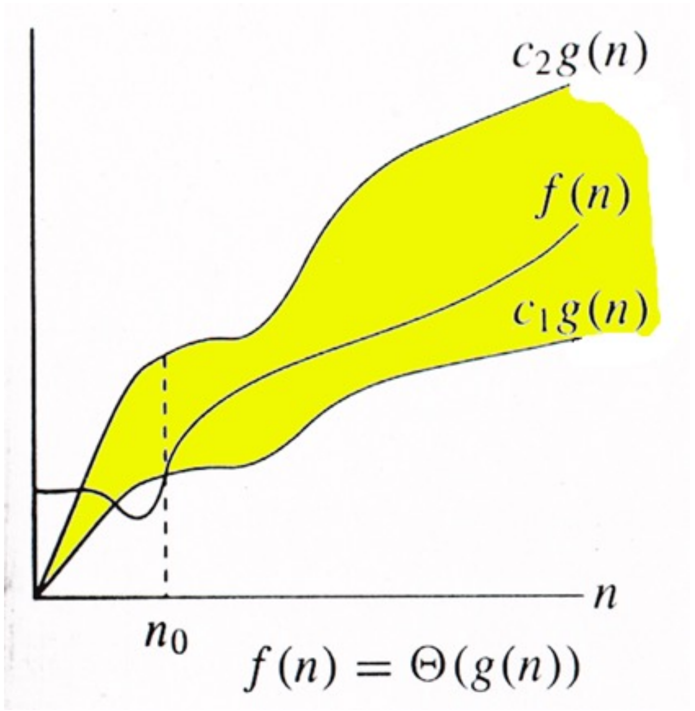
- When we say the *running time* (no modifier) of an algorithm is $\Omega(g(n))$, it applies to every input
    - So, we are giving a lower bound on the best-case running time.

- Example: **insertion sort**
    - running time belongs to both $\Omega(n)$ and $O(n^2)$
    - running time is **not** $\Omega(n^2)$
    - **worst-case** running time is $\Omega(n^2)$

Θ-notation ("Theta")

For functions $g(n)$, we define $\Theta\big(g(n)\big)$ as the set:

$$\Theta\big(g(n)\big) = \{\, f(n) : \exists \text{ positive constants } c_1, c_2, n_0, \text{ such that } \forall n \geq n_0, \text{ we have } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \,\}$$

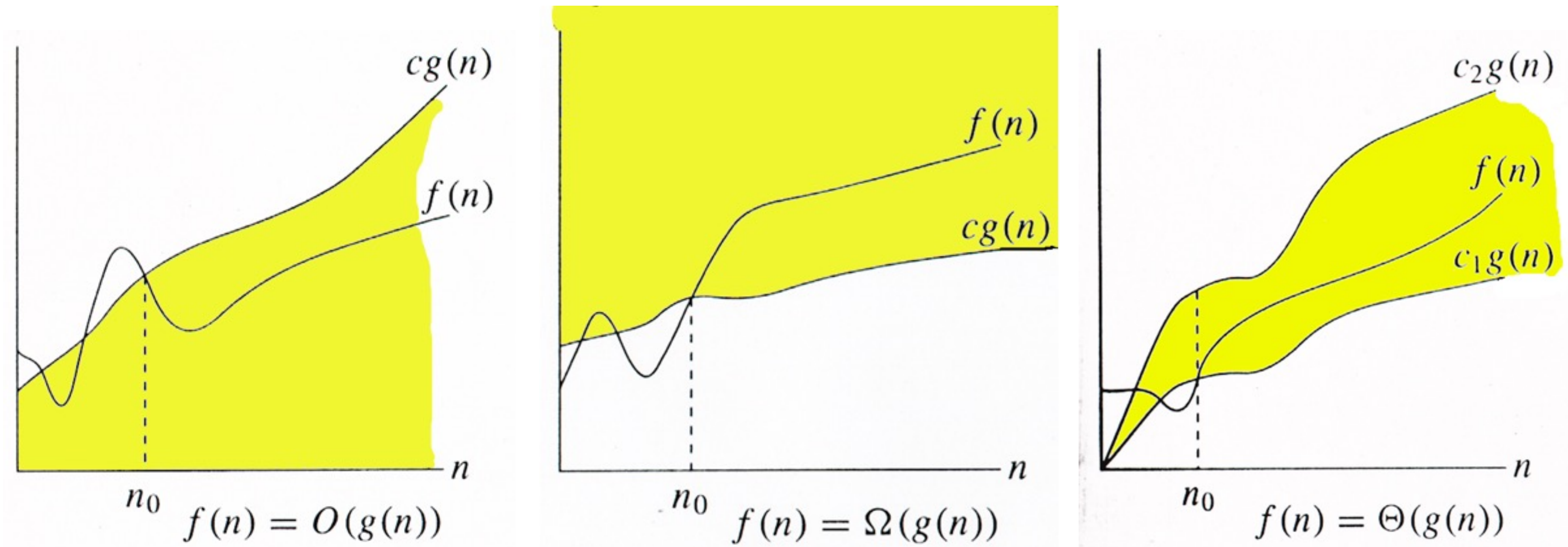


Theorem 3.1
For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

- Intuitively: $\Theta\big(g(n)\big)$ is the set of functions that have the **same** *rate of growth* as $g(n)$

- $g(n)$ is an **asymptotically tight bound** for $f(n)$

Relationship between O, Ω, Θ



Relatives of O and Ω

"Little oh"

$$\mathrm{o}\big(g(n)\big) = \{\, f(n) : \ \forall\, c > 0, \exists\, n_0 \geq 0, \text{ such that } \forall n \geq n_0, \text{ we have } 0 \leq f(n) \leq cg(n) \,\}$$

"Little omega"

$$\omega\big(g(n)\big) = \{\, f(n) : \ \forall\, c > 0, \exists\, n_0 \geq 0, \text{ such that } \forall n \geq n_0, \text{ we have } 0 \leq cg(n) < \ f(n) \,\}$$

Analogy between comparing functions *f* and *g* and comparing numbers *a* and *b:*

- $f(n) = O\big(g(n)\big)$ is like $a \leq b$
- $f(n) = \Omega\big(g(n)\big)$ is like $a \geq b$
- $f(n) = \Theta\big(g(n)\big)$ is like $a = \ b$
- $f(n) = o\big(g(n)\big)$ is like $a < b$
- $f(n) = \omega\big(g(n)\big)$ is like $a > b$

Properties

- **Transitivity:**
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ implies $f(n) = \Theta(h(n))$
  - $f(n) = O(g(n))$ and $g(n) = O(h(n))$ implies $f(n) = O(h(n))$
  - $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ implies $f(n) = \Omega(h(n))$
  - $f(n) = o(g(n))$ and $g(n) = o(h(n))$ implies $f(n) = o(h(n))$
  - $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ implies $f(n) = \omega(h(n))$

- **Reflexivity:**
  - $f(n) = \Theta(f(n))$
  - $f(n) = O(f(n))$
  - $f(n) = \Omega(f(n))$

- **Symmetry:**
  - $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$

- **Transpose symmetry:**
  - $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$
  - $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

Math to review

- Logarithms & Exponentials (3.2)

$\log_b a = c \quad \text{if} \quad a = b^c$

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:

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

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

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

$b = a^{\log_a b}$

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

- Summations (Appendix A)

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n$$

$$= \frac{1}{2} n(n+1) = \Theta(n^2)$$

- Sets and relations (Appendix B)
- Counting and probability (Appendix C)
- Proof techniques

Relationship between standard functions (3.2)

- When we discuss logarithms, we usually mean binary logarithm (base 2)

- **Fact 1**: $n^b = o(a^n)$ for all constants a and b such that $a > 1$
  - Any exponential function with a base strictly greater than 1 grows faster than any polynomial function

- **Fact 2**: $\log^b n = o(n^a)$ for any positive constant a and b
  - Any positive polynomial function grows faster than any polylogarithmic function.

- Examples which apply Fact 1 or Fact 2:
  - $\log n = o(n)$
  - $n \log n = o(n^2)$
  - $n^5 = o(2^n)$

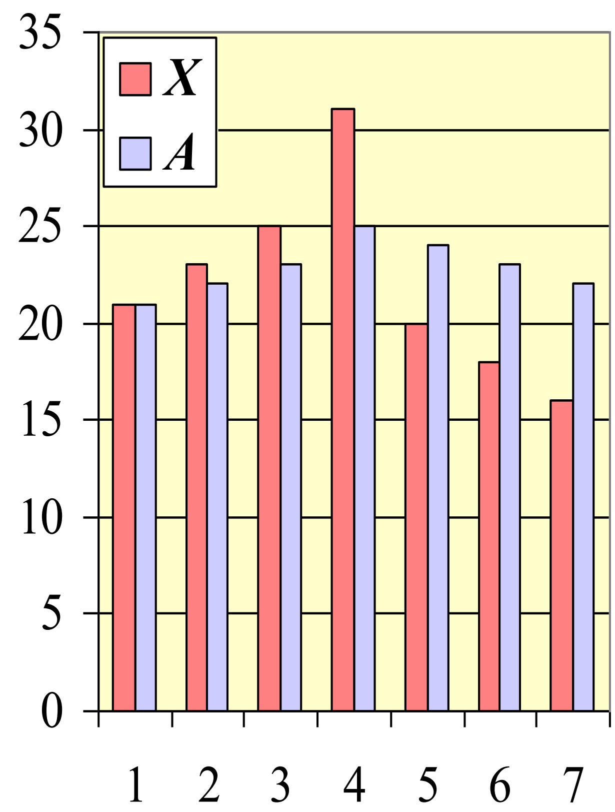
Example algorithm analysis: computing prefix average

We give two algorithms for computing prefix averages

- the ***i*-th prefix average** of an array ***X*** is the average of the first *i* elements of ***X*:**

$$\boldsymbol{A}[\boldsymbol{i}] = \frac{\boldsymbol{X}[\mathbf{1}] + \boldsymbol{X}[\mathbf{2}] + \ldots + \boldsymbol{X}[\boldsymbol{i}]}{\boldsymbol{i}}$$

- Prefix average has applications in economic and statistics



Example algorithm analysis: computing prefix average

Each algorithm takes as input an array *X* of *n* integers, and outputs an array *A* of prefix averages of *X*

```
Algorithm prefixAvgV1(X, n)
Let A be an array of n integers
for i = 1 to n  do
    s = X[1]
    for j = 2 to i do
        s = s + X[j]
    A[i] = s / i
return A
```

```
Algorithm prefixAvgV2(X, n)
Let A be an array of n integers
s = 0
for i = 1 to n  do
    s = s + X[i]
    A[i] = s / i
return A
```

What is the running time of each algorithm? Which is better?

In-class example: algorithm analysis

What is the run time of each algorithm?

Algorithm ***Foo*(*n*)**
s = 0
for $i = 1$ **to** n **do**
    s = s + 1
return s

Algorithm ***Bar*(*n*)**
s = 0
for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
        s = s + 1
return s

Algorithm ***Cow*(*n*)**
s = 0
for $i = 1$ **to** n **do**
    for $j = 1$ **to** 5 **do**
        s = s + 1
return s

Algorithm ***Cat*(*n*)**
s = 0
for $i = 1$ **to** 5 **do**
    s = s + 1
return s

Algorithm ***Bird*(*n*)**
s = 0
for $i = 1$ **to** 5 **do**
    for $j = 1$ **to** 5 **do**
        s = s + 1
return s

Algorithm ***Dog*(*n*)**
s = n
while s > 1
    $s = s / 2$
return s