CS 350 Algorithms and Complexity

Fall 2018

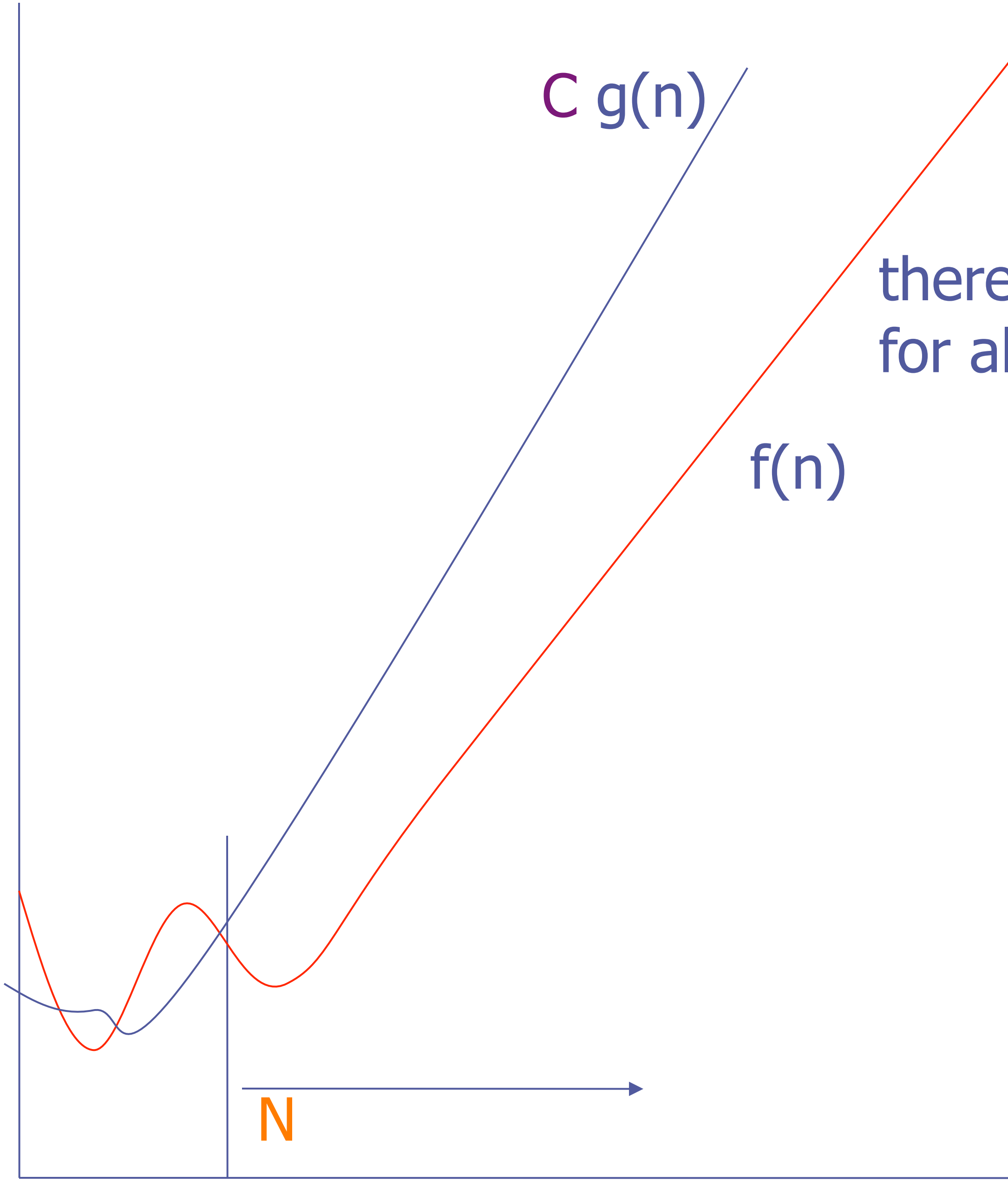
Lecture 3: Asymptotic Notation, and Analyzing Non-Recursive Algorithms

Andrew P. Black

Department of Computer Science

Portland State University

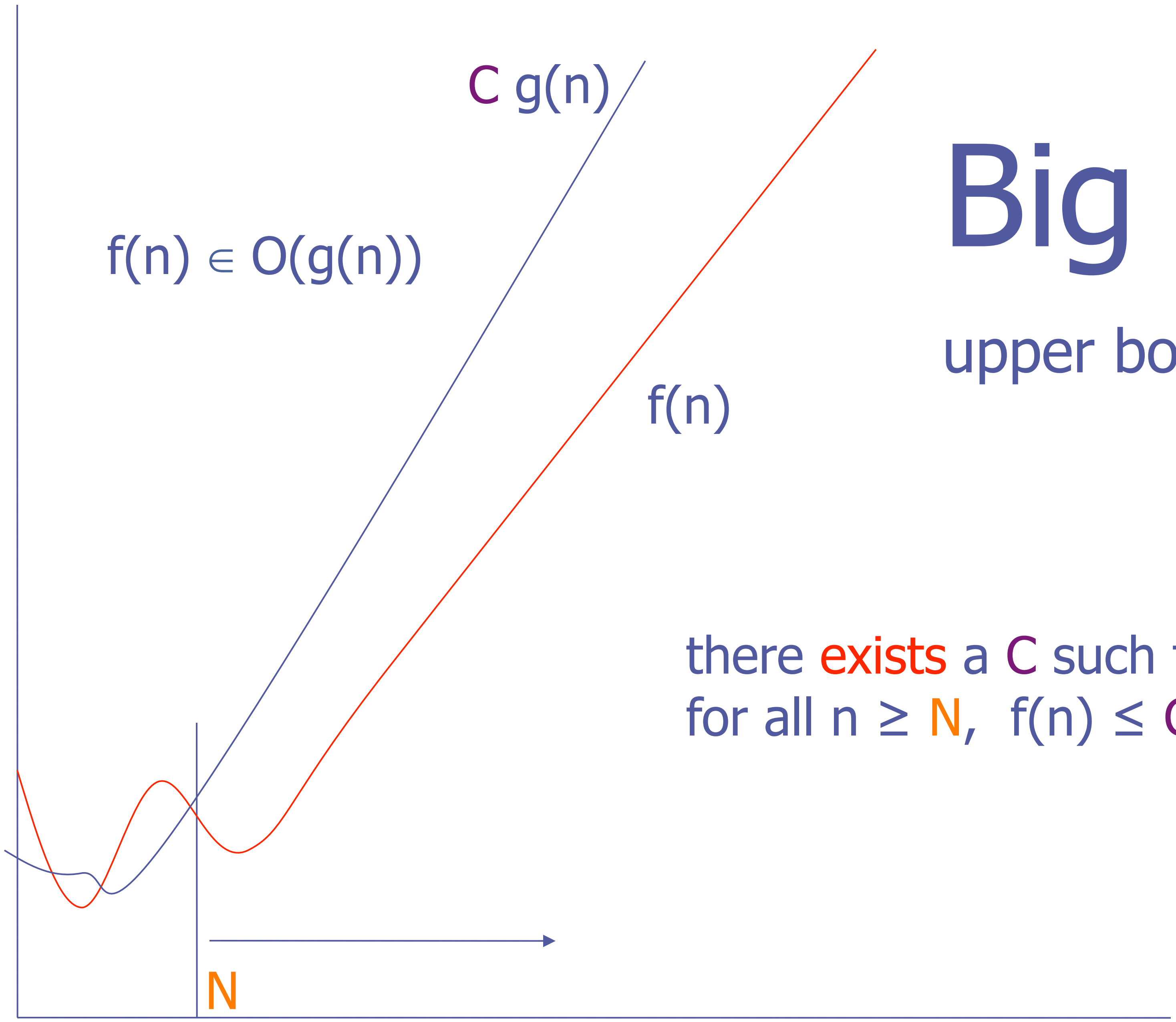
upper bounds



there exists a C such that,
for all n > N, f(n) ≤ C g(n).

Big Oh

upper bound



there exists a C such that,
for all $n \geq N$, $f(n) \leq C\, g(n)$.

Formalizing Asymptotic Nota

"Big Oh"

- A function f(n) is said to be O(g(n)) if there are constants c>0 and N>0 such that:

$$f(n) \leq c\, g(n) \quad \text{for all } n \geq N$$

- In other words: for large enough input n, f(n) is no more than a constant multiple of g(n).
- Big Oh is used for stating upper bounds.

Which of the following is true:

A. $3n^2 + 500 \in O(n)$

B. $3n^2 + 500 \in O(n^2)$

C. $3n^2 + 500 \in O(n^3)$

D. A & B

E. B & C

F. none of the above

Which of the following is true:

A. $3n^3/500 \in O(n)$

B. $3n^3/500 \in O(n^2)$

C. $3n^3/500 \in O(n^3)$

D. A & B

E. B & C

F. none of the above

Examples:

- $4n^2 + 3 \in O(n^2)$
- $4n^3 + 3 \in O(n^3)$
- $n^2/1000 + 3000n \in O(n^2)$

In general:

- Can ignore all but the highest power
- Can ignore coefficients

Logarithms

- Which of the following is true?

  A. $O(\ln n) = O(\log_{10} n)$

  B. $O(\lg n) = O(\ln n)$

  C. $\lg n = \ln n$

  D. all of the above are true

  E. none of the above is true

  F. A and B are true

  G. B and C are true

Powers

- Which of the following is true

A. $O(4^n) = O(2^n)$

B. $O(2 \times 2^n) = O(10 \times 2^n)$

C. both of the above are true

D. neither of the above is true

More Examples:

Logarithms:

- Can ignore base because:

$$\log_a b = \log_c b / \log_c a.$$

- Thus $O(\log_2 n)$ is the same as $O(\log_{10} n)$.

Exponents:

- Can ignore non-exponential terms
- Base of exponentiation is important; for example, $O(4^n)$ is bigger than $O(2^n)$.

Properties of Big Oh:

- For all constants $c>0$ and $a>1$, and monotonically increasing functions $f(n)$:

$$f(n)^c \text{ is } O(a^{f(n)})$$

- For example:
  - n^c is $O(a^n)$
  - n^{256} is $O(1.0001^n)$
  - $(\log_a n)^c$ is $O(a^{\log_a n})$, which is $O(n)$.

More Properties of Big Oh:

- $\mathcal{O}$ notation is additive and multiplicative:

  If $f(n) \in \mathcal{O}(s(n))$ and $g(n) \in \mathcal{O}(t(n))$, then:

  - $f(n) + g(n) \in \mathcal{O}(s(n) + t(n))$;
  - $f(n)g(n) \in \mathcal{O}(s(n)t(n))$.

- $\mathcal{O}$ notation is transitive:

  If $f(n) \in \mathcal{O}(g(n))$, and $g(n) \in \mathcal{O}(h(n))$, then $f(n) \in \mathcal{O}(h(n))$.

Classes of Algorithm:

There are standard names for some of the most common complexity classes:

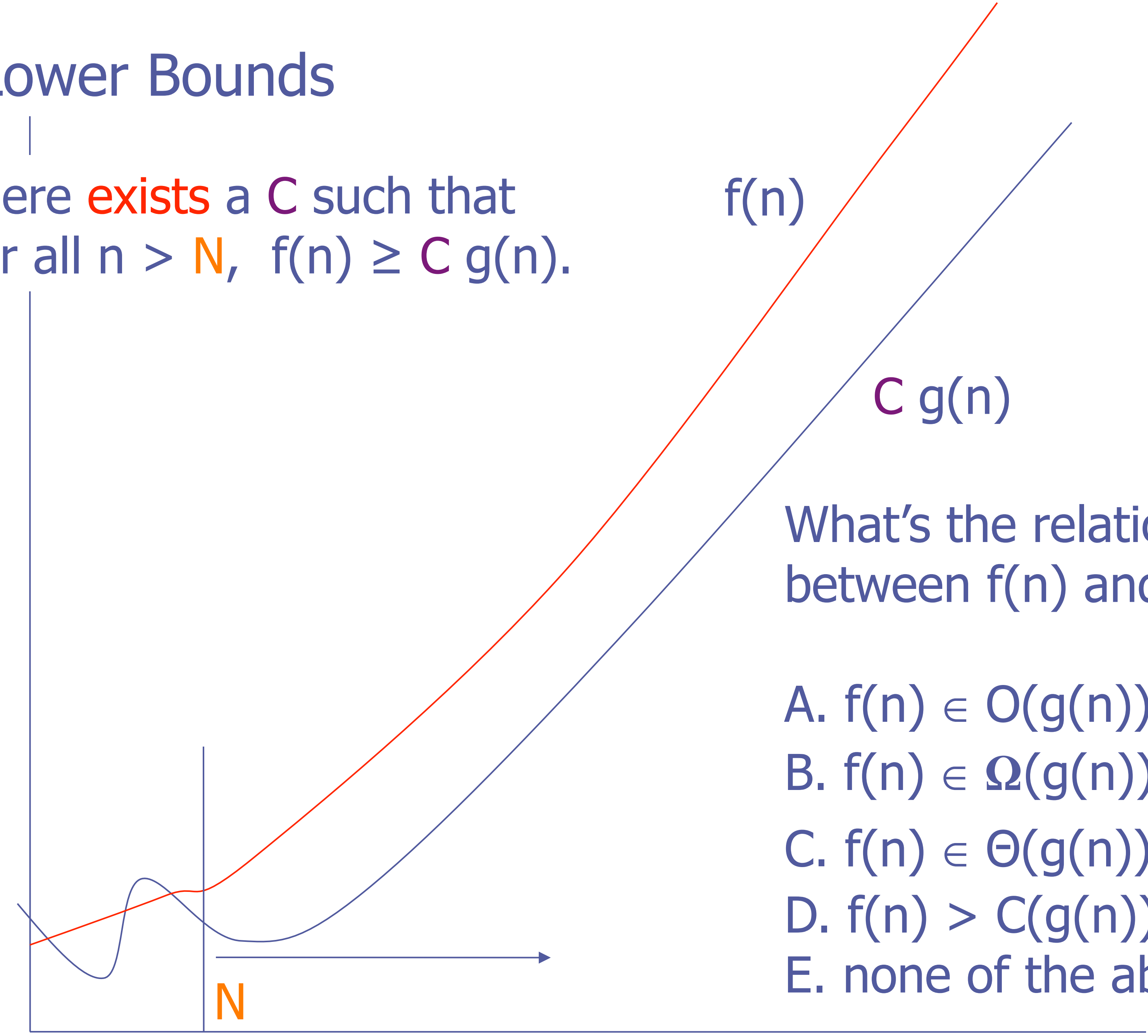
- Constant: $O(1)$
- Logarithmic: $O(\log n)$
- Linear: $O(n)$
- Linearithmic: $O(n \log n)$
- Quadratic: $O(n^2)$
- Exponential: $O(2^n)$
- Double Exponential: $O(2^{2^n})$

Polynomial Algorithms:

- An algorithm is said to be <u>polynomial</u> if it is $O(n^p)$ for some integer p.
- Terminology:
  - Problems with polynomial algorithms are generally considered to be <u>tractable</u>.
  - Problems for which no polynomial algorithm has been found are often considered <u>intractable</u>.

Lower Bounds

there exists a C such that
for all n > N, f(n) ≥ C g(n).



What's the relationship between f(n) and g(n)?

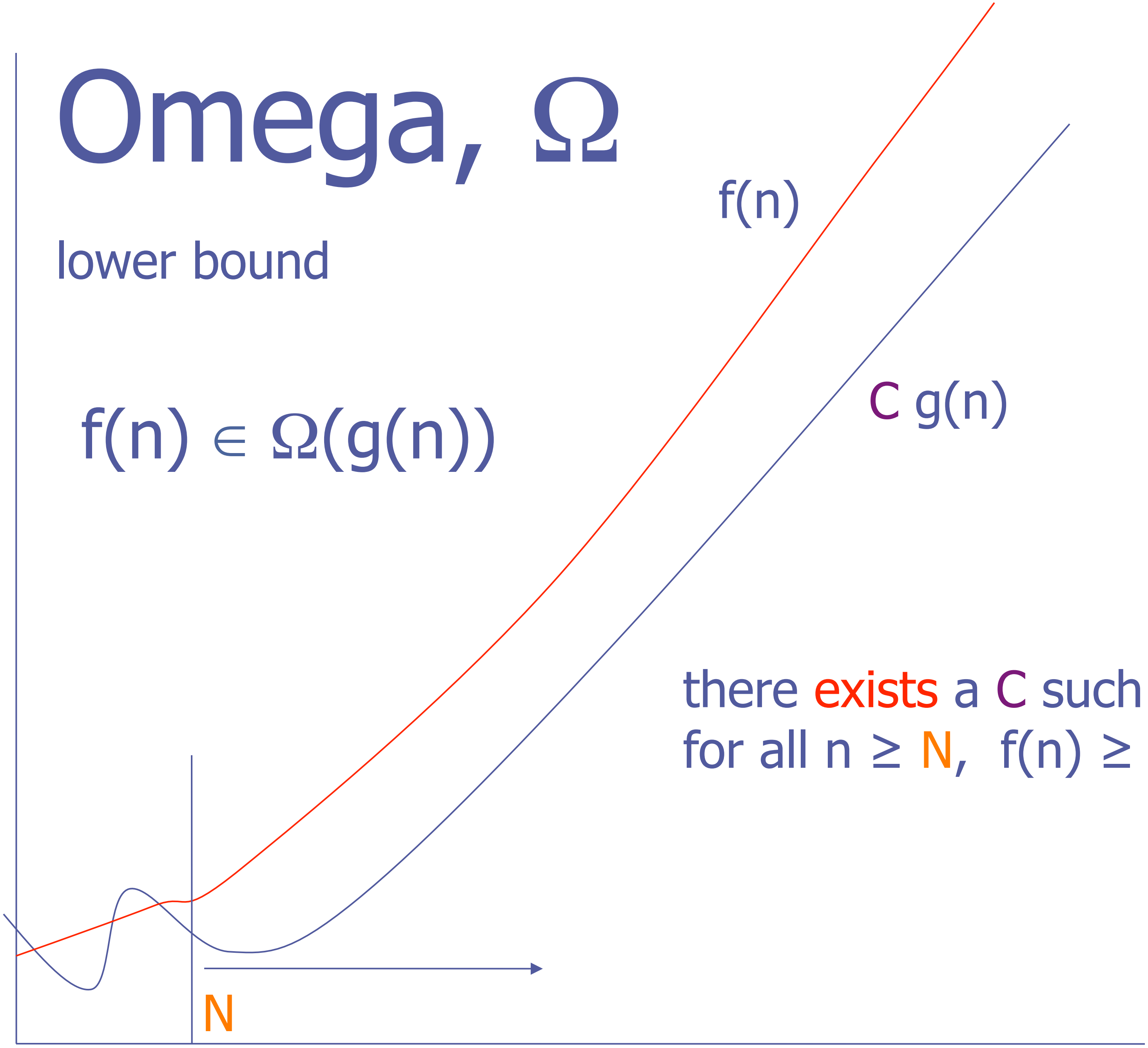
A. $f(n) \in O(g(n))$
B. $f(n) \in \Omega(g(n))$
C. $f(n) \in \Theta(g(n))$
D. $f(n) > C(g(n))$
E. none of the above

Omega, Ω

lower bound

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

there exists a C such that
for all $n \geq N$, $f(n) \geq C\, g(n)$.



Dealing with Lower Bounds:

- "This algorithm takes at least …"

Omega

- A function $f(n)$ is said to be in $\Omega(g(n))$ if there are constants c>0 and N>0 such that:

$$f(n) \geq c\ g(n) \quad \text{for all } n \geq N$$

- Note that $f(n) \in \Omega(g(n))$ if and only if $g(n) \in \mathrm{O}(f(n))$.

Mnemonics

- Big Oh is really a Capital greek letter Omicron; pronounce it O-**micron**. Pronounce **Ω** O-**mega.**
- Read $f(n) \in O(g(n))$ as f is O-smaller-than g
- Read $f(n) \in \Omega(g(n))$ as f is O-larger-than g
  - The large O (O, **Ω**) says: f may be equal to g

Tight Bounds:

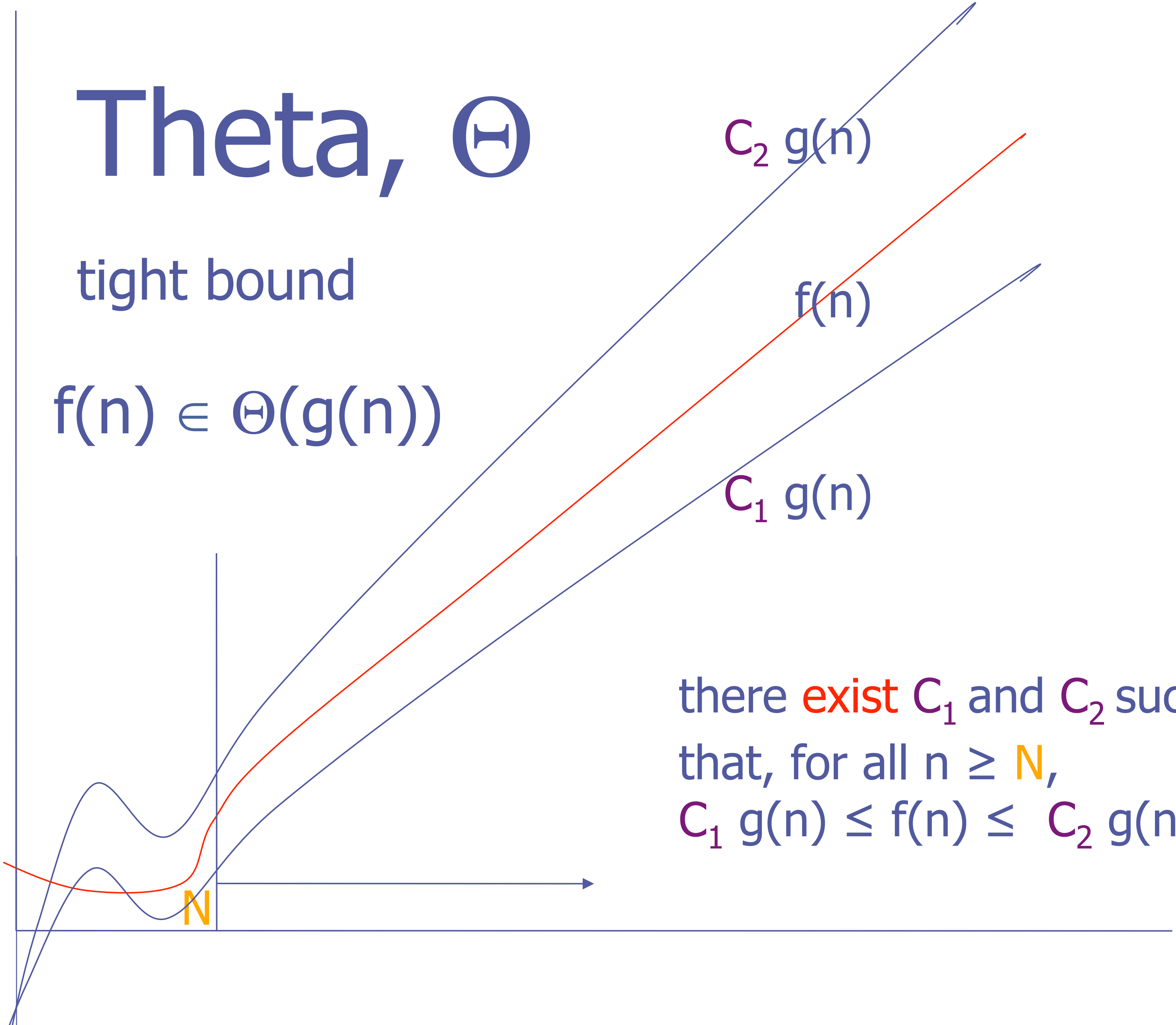
- A function f(n) is said to be in Θ(g(n)) if it is in both O(g(n)) and Ω(g(n)).
  - If f(n) $\in$ Θ(g(n)), then it is eventually “sandwiched” between constant multiples of g(n).

- $f(n) \in \Theta(g(n))$ if and only if $\lim_{n \to \infty} \frac{g(n)}{f(n)} = c$

Theta, Θ

tight bound

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



there exist C_1 and C_2 such that, for all $n \geq N$,
$C_1\ g(n) \leq f(n) \leq C_2\ g(n)$.

Simple laws of $\Theta(..)$ notation:

- Addition:

  $$\Theta(f(n) + g(n)) = \Theta(f(n)) + \Theta(g(n))$$

- Scaling: for any constant $c>0$,

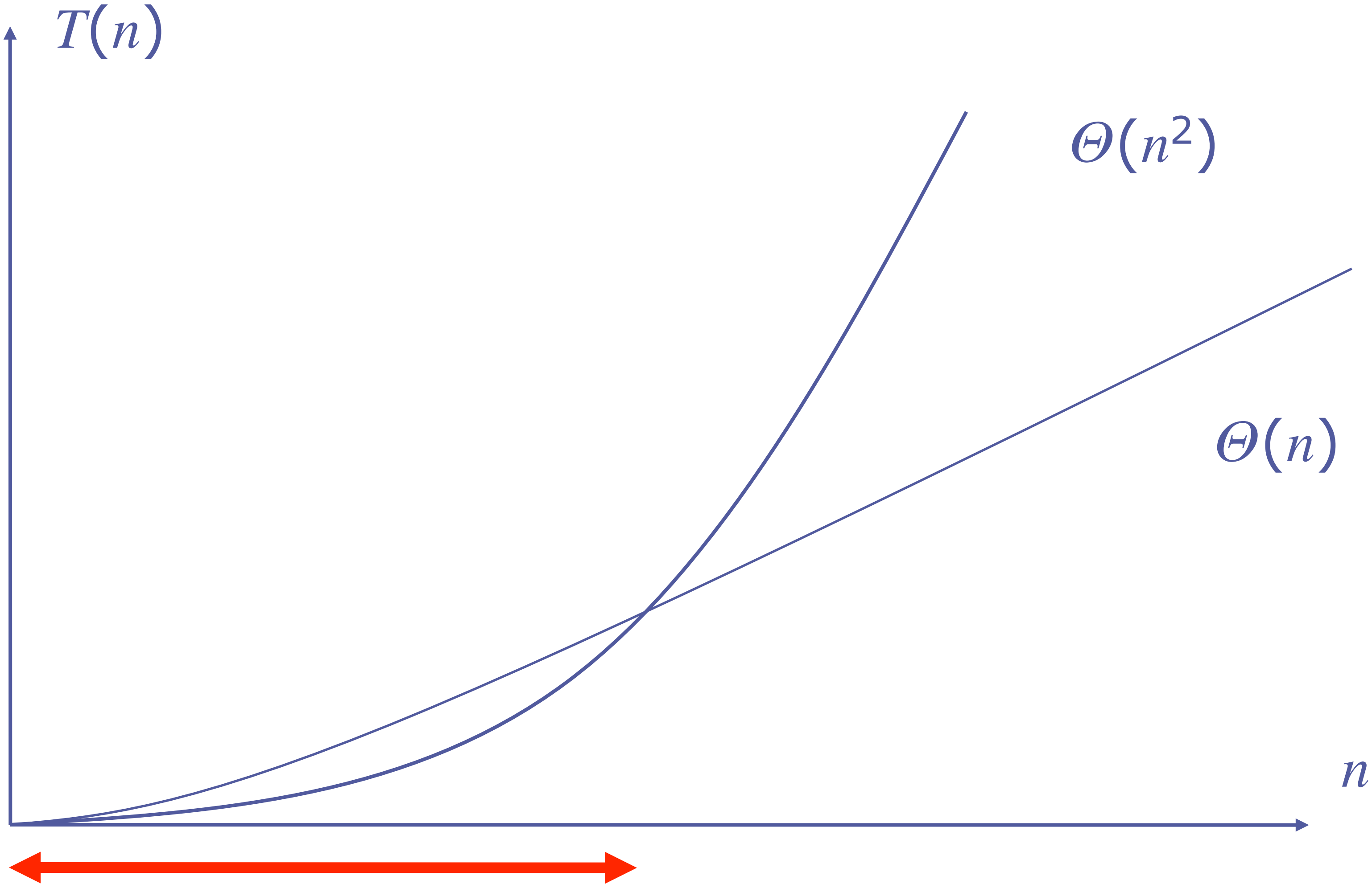
  $$\Theta(cf(n)) = c\,\Theta(f(n)) = \Theta(f(n))$$

True or False

- You have two sorting algorithms:
  B is $O(n^2)$, while
  Q is $O(n \lg n)$.
- True or false: Q is <u>always</u> faster than B
  - A. True
  - B. False

Beware Constant Factors!

- Use complexity measures with care!
- A $\Theta(n^2)$ algorithm might actually be faster than a $\Theta(n)$ algorithm for all values of n encountered in some real application!



The $\Theta(n^2)$ algorithm is faster than the $\Theta(n)$ alternative if we're working within this particular range …

Comparing Orders of Growth

- If you need to compare the rates of growth of two functions, t and g, the easiest way is often to take limits:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \Rightarrow t(n) \text{ has a smaller order of growth than } g(n) \\ c > 0 & \Rightarrow t(n) \text{ has the same order of growth as } g(n) \\ \infty & \Rightarrow t(n) \text{ has a larger order of growth than } g(n) \end{cases}$$

Analysis of time efficiency

- Time efficiency is analyzed by determining the number of repetitions of the "basic operation"
- Almost always depends on the size of the input
- "Basic operation": the operation that contributes most towards the running time of the algorithm

cost of basic op: constant

number of times basic op is executed

run time

$$T(n) \approx c_{op} \times C(n)$$

Problem	Input size measure	Basic operation
Searching for key in a list of *n* items	A: Number of list's items, i.e. n	
Multiplication of two matrices	B: Matrix dimension, or total number of elements	
Checking primality of a given integer *n*	C: size of n = number of digits (in binary rep)	
Shortest path through a graph	D: #vertices and/or edges	

Problem	Input size measure	Basic operation
Searching for key in a list of n items	A: Number of list's items, i.e. n	A: Key comparison
	B: Matrix dimension, or total number of elements	B: Multiplication of two numbers
	C: size of n = number of digits (in binary rep)	C: Division
	D: #vertices and/or edges	D: Visiting a vertex or traversing an edge

Best-case, average-case, worst-case

- For some algorithms, efficiency depends on the input:
- Worst case: $C_{worst}(n)$ – maximum over inputs of size n
- Best case: $C_{best}(n)$ – minimum over inputs of size n
- Average case: $C_{avg}(n)$ – "average" over inputs of size n
  - Number of times the basic operation will be executed on typical input
    - <u>Not</u> the average of worst and best case
  - Expected number of basic operations treated as a random variable under some assumption about the probability distribution of all possible inputs

Discuss:

ALGORITHM *UniqueElements*$(A[0..n-1])$

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i] = A[j]$ **return false**
return true

- What's the best case, and its running time?
  A. constant — $O(1)$
  B. linear — $O(n)$
  C. quadratic — $O(n^2)$

Discuss:

ALGORITHM *UniqueElements*($A[0..n-1]$)
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

- What's the worst case, and its running time?
  A. constant — $O(1)$
  B. linear — $O(n)$
  C. quadratic — $O(n^2)$

Discuss:

ALGORITHM *UniqueElements*$(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise

```
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

- What's the average case, and its running time?
  - A. constant — $O(1)$
  - B. linear — $O(n)$
  - C. quadratic — $O(n^2)$

General Plan for Analysis of non-recursive algorithms

1. Decide on parameter n indicating input size
2. Identify algorithm's basic operation
3. Determine worst, average, and best cases for input of size n
4. Set up a sum for the number of times the basic operation is executed
5. Simplify the sum using standard formulae and rules (see Levitin Appendix A)

“Basic Operation”

ALGORITHM $MaxElement(A[0..n-1])$

//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $A[i] > maxval$
    $maxval \leftarrow A[i]$
return $maxval$

- Why choose > as the basic operation?
  - Why not $i \leftarrow i + 1$?
  - Or [ ] ?

Same Algorithm:

ALGORITHM *MaxElement* (*A*: *List*)
// Determines the value of the largest element in the list A
// Input: a list A of real numbers
// Output: the value of the largest element of A
maxval ← *A.first*
for *each* **in** A **do**
  if *each* > *maxval*
    maxval ← *each*
return *maxval*

- Why choose > as the basic operation?
  - Why not $i \leftarrow i + 1$?
  - Or [ ] ?

Useful Summation Formulae

$\Sigma_{1 \leq i \leq u} 1 =$

In particular, $\Sigma_{1 \leq i \leq n} 1 = n$

$\Sigma_{1 \leq i \leq n}\, i =$

$\Sigma_{1 \leq i \leq n}\, i^2 =$

$\Sigma_{0 \leq i \leq n}\, a^i =$

In particular, $\Sigma_{0 \leq i \leq n}\, 2^i =$

$\Sigma(a_i \pm b_i) =$

$\Sigma c\, a_i =$

$\Sigma_{l \leq i \leq u}\, a_i =$

Useful Summation Formulae

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n}\, a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \ne 1$

In particular, $\Sigma_{0 \le i \le n}\, 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma\, a_i \pm \Sigma\, b_i$ $\qquad$ $\Sigma c\, a_i = c\, \Sigma\, a_i$

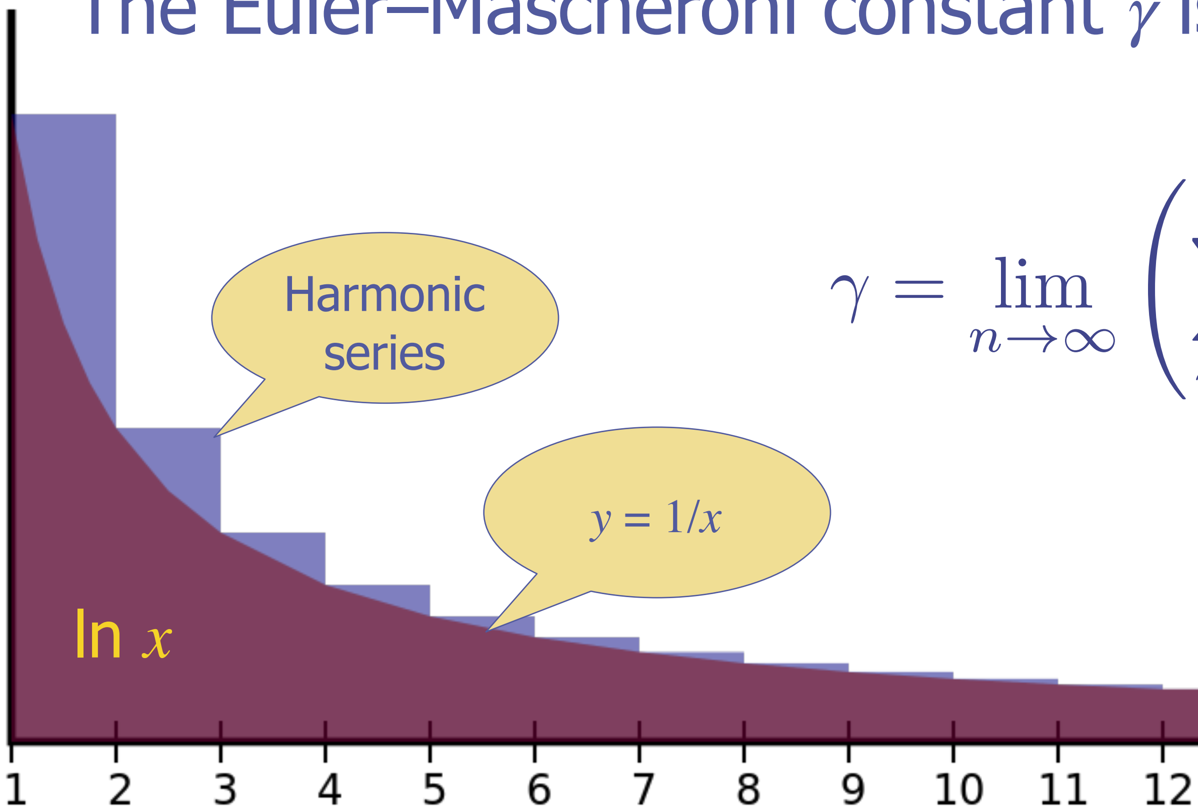
$\Sigma_{l \le i \le u}\, a_i = \Sigma_{l \le i \le m}\, a_i + \Sigma_{m+1 \le i \le u}\, a_i$

Where do the Summation formulae come from?

- Answer: mathematics.
- Example:

The Euler–Mascheroni constant γ is <u>defined</u> as:

$$\gamma = \lim_{n\to\infty}\left(\sum_{i=1}^{n}\frac{1}{i} - \ln n\right)$$



What does Levitin's $\approx$ mean?

- "becomes almost equal to as $n \to \infty$"
- So formula 8

$$\sum_{i=1}^{n} \lg i \approx n \lg n$$

  - means

$$\lim_{n\to\infty} \left(\sum_{i=1}^{n} \lg i - n \lg n \right) = 0$$

Example: Counting Binary Digits

ALGORITHM *Binary*(n)

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
$count \leftarrow 1$
while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
return $count$

- How many times is the basic operation executed?
- Why is this algorithm harder to analyze than the earlier examples?

Ex 2.3, Problem 1

- Working with a *partner*:

1. Compute the following sums.

   a. $1 + 3 + 5 + 7 + ... + 999$

   b. $2 + 4 + 8 + 16 + ... + 1024$

   c. $\sum_{i=3}^{n+1} 1$

   d. $\sum_{i=3}^{n+1} i$

   e. $\sum_{i=0}^{n-1} i(i+1)$

   f. $\sum_{j=1}^{n} 3^{j+1}$

   g. $\sum_{i=1}^{n} \sum_{j=1}^{n} ij$

   h. $\sum_{i=1}^{n} 1/i(i+1)$

Ex 2.3, Problem 2

2. Find the order of growth of the following sums.

   a. $\sum_{i=0}^{n-1}(i^2+1)^2$ b. $\sum_{i=2}^{n-1}\lg i^2$

   c. $\sum_{i=1}^{n}(i+1)2^{i-1}$ d. $\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}(i+j)$

   Use the $\Theta(g(n))$ notation with the simplest function $g(n)$ possible.

Ex 2.3, Problem 3

3. The sample variance of n measurements $x_1, x_2, ..., x_n$ can be computed as

$$\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1} \text{ where } \bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

or

$$\frac{\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2/n}{n-1}.$$

Find and compare the number of divisions, multiplications, and additions/subtractions (additions and subtractions are usually bunched together) that are required for computing the variance according to each of these formulas.

Ex 2.3, Problem 4

4. Consider the following algorithm.

Algorithm *Mystery*(n)
//Input: A nonnegative integer n
$S \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $S \leftarrow S + i * i$
return S

What does this algorithm compute?

A. n^2

B. $\sum_{i=1}^{n} i$

C. $\sum_{i=1}^{n} i^2$

D. $\sum_{i=1}^{n} 2i$

Ex 2.3, Problem 4

4. Consider the following algorithm.

Algorithm *Mystery*(n)
//Input: A nonnegative integer n
$S \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $S \leftarrow S + i * i$
return S

What is the basic operation?

A. multiplication

B. addition

C. assignment

D. squaring

Ex 2.3, Problem 4

4. Consider the following algorithm.

Algorithm *Mystery*(n)
//Input: A nonnegative integer n
$S \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $S \leftarrow S + i * i$
return S

How many times is the basic operation executed?

A. once

B. n times

C. $\lg n$ times

D. none of the above

Ex 2.3, Problem 4

4. Consider the following algorithm.

Algorithm *Mystery*(n)
//Input: A nonnegative integer n
$S \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $S \leftarrow S + i * i$
return S

What is the efficiency class of this algorithm? [b is # of bits needed to represent n]

A. $\Theta(1)$

B. $\Theta(n)$

C. $\Theta(b)$

D. $\Theta(2^b)$

Ex 2.3, Problem 4 (cont)

e. Suggest an improvement or a better algorithm altogether and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

Ex 2.3, Problem 9

Prove the formula

$$\sum_{i=1}^{n} i = 1 + 2 + ... + n = \frac{n(n+1)}{2}$$

either by mathematical induction or by following the insight of a 10-year old schoolboy named Karl Friedrich Gauss (1777–1855) who grew up to become one of the greatest mathematicians of all times.

Ex 2.3, Problem 11

Algorithm $GE(A[0..n-1, 0..n])$
//Input: An n-by-$n+1$ matrix $A[0..n-1, 0..n]$ of real numbers
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        for $k \leftarrow i$ **to** n **do**
            $A[j,k] \leftarrow A[j,k] - A[i,k] * A[j,i] \;/\; A[i,i]$

a. Find the time efficiency class of this algorithm
b. What glaring inefficiency does this code contain, and how can it be eliminated?
c. Estimate the reduction in run time.