Lecture 1
Part 2

This starts where we left off last week.

I made a few minor changes to a few of the slides.

Pause and Reflect

- SO FAR: we learned how to determine the *running time* aka the *efficiency* of an algorithm
  - Non-recursive algorithms only
  - Count the statements, or the basic operations
  - The result is a function of n (input size)

Running times of algorithms are functions.

$$f(n) = n$$

$$f(n) = n^2$$

$$f(n) = \frac{(n-1)n}{2}$$

There are LOTS of functions in the world.

8675309

$4n^2+10$

3^n

$\log_2 n$

$50n^3+20n+4$

$4n\log_2 n$

2^n-1

$1+\log_2 6$

$3\log_2 n+1$

$3737n$

$n(2n+1)$

$3\log_2 n+n$

$5!+3^2$

n^2+3n^3

$\log_2 n+9n!$

Comparing functions

- Some functions are bigger than others
- What does “bigger” mean?
- We need a formalized way to talk about this

From earlier:

1. Efficiency of an algorithm depends on **input size**
2. Efficiency of an algorithm also depends on **basic operation**
3. Efficiency can be expressed by **counting** the basic operation

This is the algorithm from “Example 3” in Part 1

```
Loops(A[0..n-1])
  for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j≥0 and A[j]>v do
      A[j+1] ← A[j]
      j ← j-1
    A[j+1] ← v
```

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \approx \frac{n^2}{2}$$

Example 1

- Problem: find the largest element in a list
- Input size measure:
  - *Number of list items, i.e. n*
- Basic operation:
  - *If statement / comparison*

```
ALGORITHM MaxElement(A[0..n − 1])
    maxval ← A[0]
    for i ← 1 to n − 1 do
        if A[i] > maxval
            maxval ← A[i]
    return maxval
```

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1$$

Example 2

- Problem: Multiplication of two matrices
- Input size measure:
  - *Matrix* dimension *(elements per row/col)*
- Basic operation:
  - *Innermost expression and assignment*

```
ALGORITHM MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            C[i, j] ← 0.0
            for k ← 0 to n − 1 do
                C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
    return C
```

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = n^3$$

Example 3

- Problem: calculating an unusual sum
- Input size measure:
  - *Number n*
- Basic operation*:*
  - *Division & assignment on line 6*
    - (but note that div-by-2 is actually a super-fast op)

```
1. Example3(n)
2.  sum ← 0
3.  i ← n
4.  while i ≥ 1
5.      sum ← sum + 1
6.      i ← i/2
7.  return sum
```

$$C(n) = \log n$$

Example 4

- Problem: Searching for key in a list of n items
- Input size measure:
  - *Number of list items, i.e. n*
- Basic operation:
  - *Key comparison / while loop*

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
    $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

Depends on order of input

$C_{worst}(n) = n$
$C_{best}(n) = 1$

Worst case, average case, best case

- Worst case:
  - Most possible number of steps needed by an algorithm
- Average case:
  - Number of steps needed “on average”
- Best case:
  - Number of steps needed if you “get lucky” with a particular input

- Consider the problem of finding an element in an unsorted list

Which to use: best, worst, average?

- We will focus on worst-case analysis in this course
  - Unless otherwise specified, you should always analyze the worst case

- There are many situations where best case = worst case
  - Example: find the *largest* element in an unsorted list

Running time/efficiency can be many different functions

- $C(n) = n(n\text{-}1)/2$
- $C(n) \approx 0.5n^2$
- $C(n) = log\ n + 5$
- $C(n) = n!$

8675309

$4n^2+10$

3^n

$\log_2 n$

$50n^3+20n+4$

$4n\log_2 n$

2^n-1

$1+\log_2 6$

$3\log_2 n+1$

$3737n$

$n(2n+1)$

$3\log_2 n+n$

$5!+3^2$

n^2+3n^3

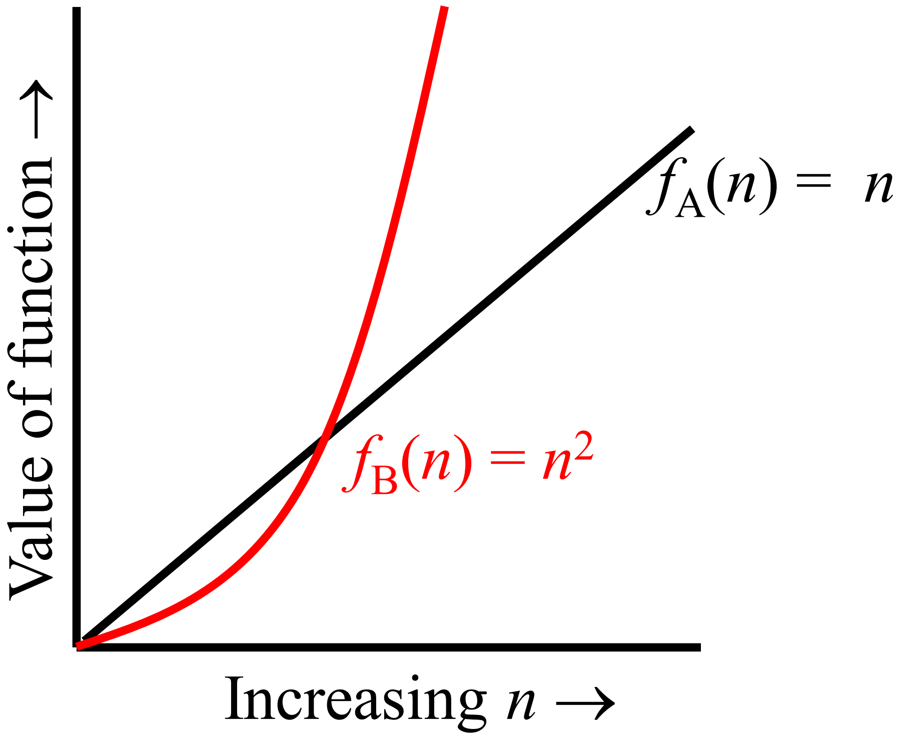
$\log_2 n+9n!$

- Which one is the better algorithm?

Let’s look at some functions

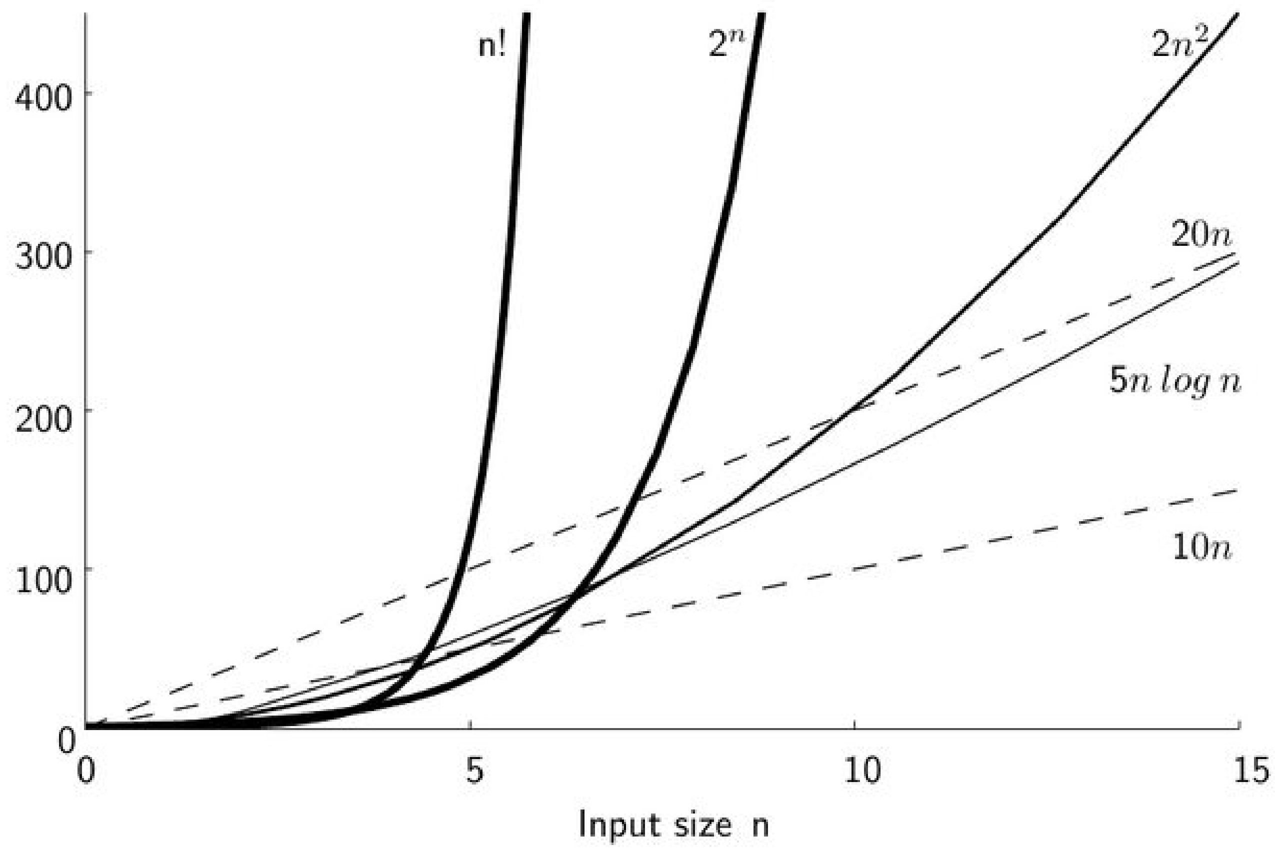
- DESMOS

Order of growth



Order of growth

- What we really care about:
  - Order of growth as $n \rightarrow \infty$



Orders of growth

these represent possible functions that classify basic ops counts

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$\boxed{9.3 \cdot 10^{157}}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

1.5×10^{133} years on the world's fastest supercomputer

Common efficiency classes

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 5.5). Note that a logarithmic algorithm cannot take into account all its input (or even a fixed fraction of it): any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	"*n-log-n*"	Many divide-and-conquer algorithms (see Chapter 4), including mergesort and quicksort in the average case, fall into this category.

Common efficiency classes (cont.)

n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on n-by-n matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

General strategy for analysis of non-recursive algorithms

From the textbook (p62):

1. Decide on a parameter indicating the input's size.
2. Identify the algorithm's basic operation.
3. Be sure the number of times the basic operation is executed depends only on the size of the input.
   - If it depends on some other property, the best/worst/average case efficiencies must be investigated separately
4. Set up a sum expressing the number of times the basic operation is executed.
5. Use summation algebra to find a closed-form expression for the sum from step 4 above.
6. Determine the efficiency class of the algorithm using asymptotic notations

Asymptotic order of growth

A way of comparing functions

- Big O (Pronounced “big oh”)
- Big Ω
- Big Θ

Some functions are essentially *the same*

8675309

$4n^2+10$

3^n

$\log_2 n$

$50n^3+20n+4$

$4n\log_2 n$

2^n-1

$1+\log_2 6$

$3\log_2 n+1$

$3737n$

$n(2n+1)$

$3\log_2 n+n$

$5!+3^2$

n^2+3n^3

$\log_2 n+9n!$

3^n

2^n-1

$50n^3+20n+4$
n^2+3n^3

$4n^2+10$
$n(2n+1)$

$\log_2 n+9n!$

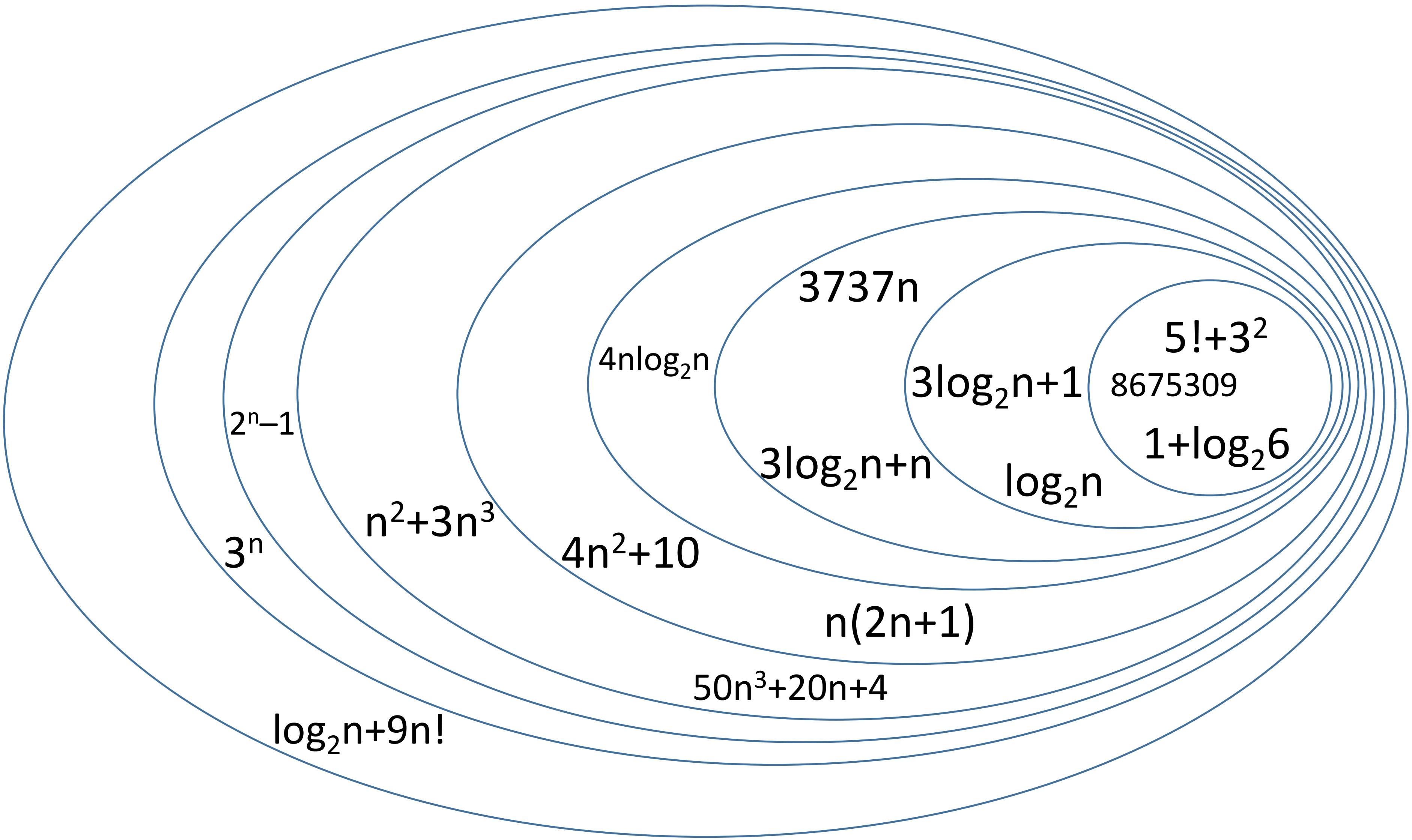
$5!+3^2$
8675309
$1+\log_2 6$

$4n\log_2 n$

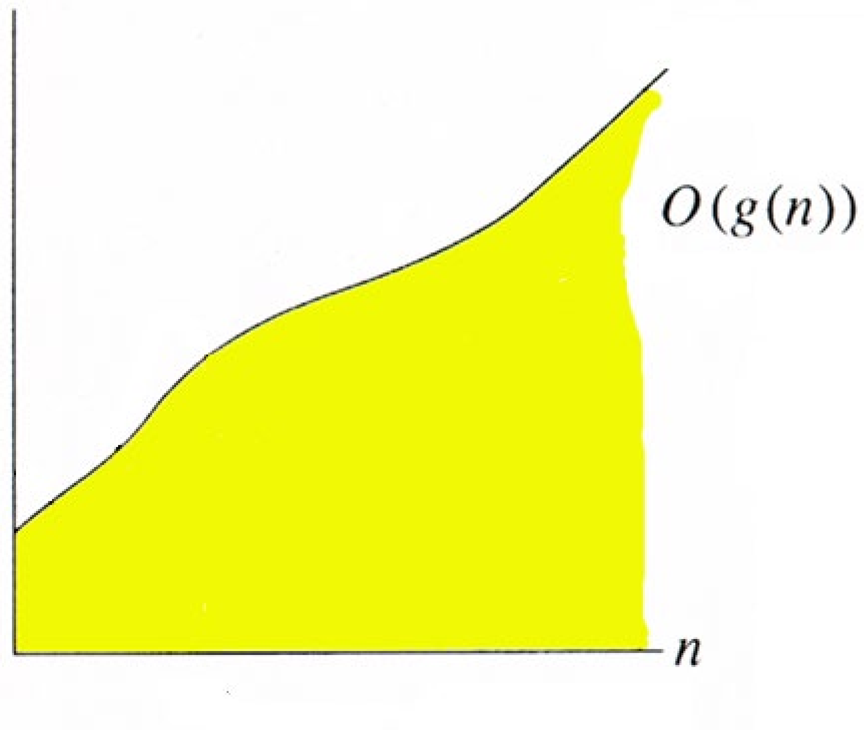
$3\log_2 n+1$
$\log_2 n$

$3737n$
$3\log_2 n+n$

Even better


3737n
$5!+3^2$
$4n\log_2 n$
$3\log_2 n+1$
8675309
2^n-1
$1+\log_2 6$
$3\log_2 n+n$
$\log_2 n$
n^2+3n^3
3^n
$4n^2+10$
$n(2n+1)$
$50n^3+20n+4$
$\log_2 n+9n!$

Big-O in pictures

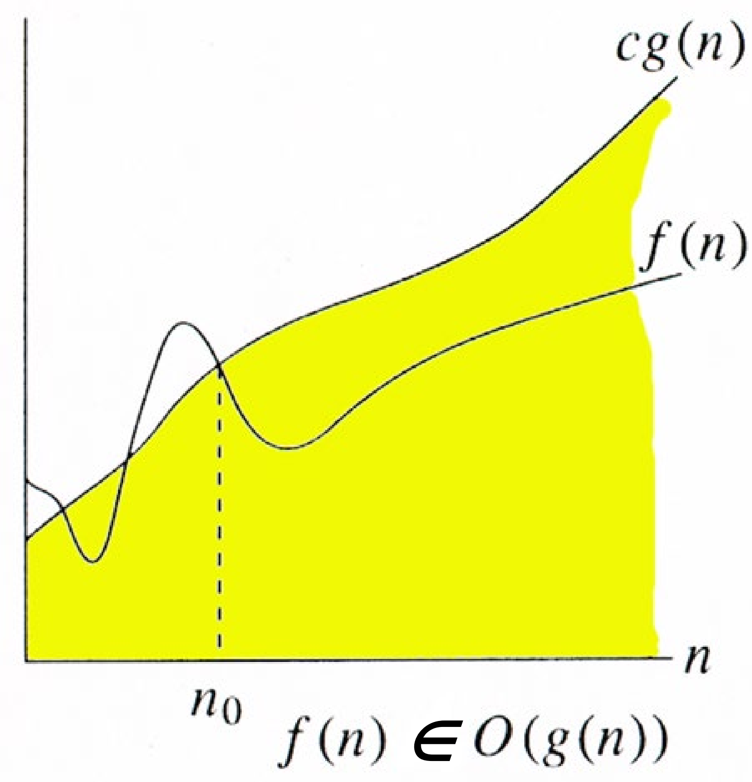


Set of all functions whose *rate of growth* is the same as or lower than that of *g*(*n*).

We also say "f(n) is *bounded above* by a constant multiple of g(n)"

or (carelessly) just "f(n) is bounded by g(n)"

Big-O in pictures



$f(n) \leq c * g(n)$, for all $n \geq n_0$

Big-O (formal definition)

<u>Definition:</u>

- a function **f(n)** is in the set **O(g(n))** [*denoted: $f(n) \in O(g(n))$*] if there is a constant **c** and a positive integer $\mathbf{n_0}$ such that

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

i.e. f(n) is bounded above by some constant multiple of g(n)

Example

- Is $f(n) = 2n+6 \in O(n)$?
- By the definition:
  - Need to find a constant c and a constant n_0 such that $f(n) \leq cg(n)$ for all $n > n_0$
- Many will work
  - Use $c = 4$ and $n_0 = 3$
- $\rightarrow$ $f(n)$ is $\in O(n)$

n	f(n)	c*g(n)
1	8	4
2	10	8
3	12	12
4	14	16
5	16	20
6	18	24
…	…	…

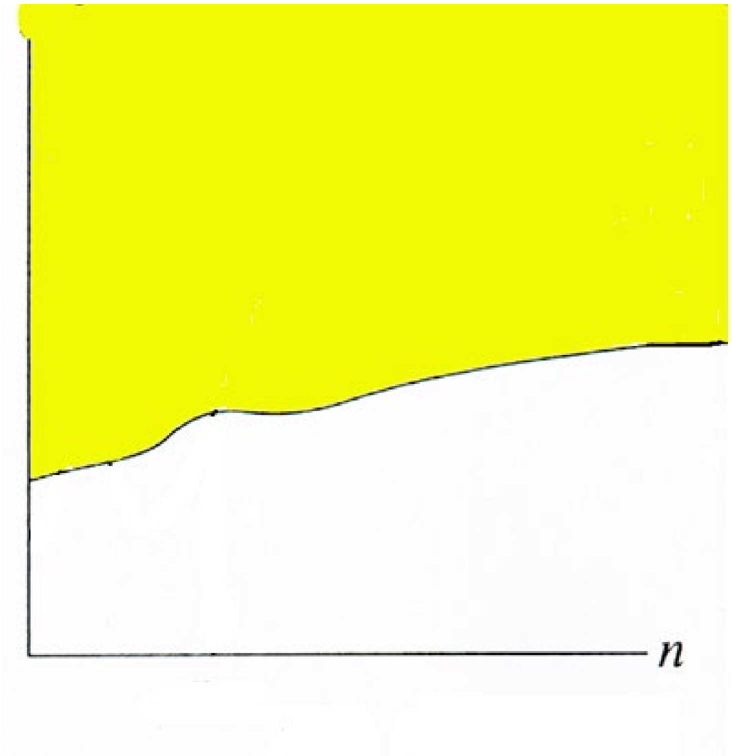
Looks good from here down

Big-O

- Simple Rule: Drop lower order terms and constant factors

1. $50n^3 + 20n + 4 \in O(n^3)$
2. $4n^2 + 10 \in O(n^2)$
3. $n(2n + 1) \in O(n^2)$
4. $3\log n + 1 \in O(\log n)$
5. $3\log n + n \in O(n)$
6. $1 + \log 6 \in O(1)$
7. $5! + 3^2 \in O(1)$

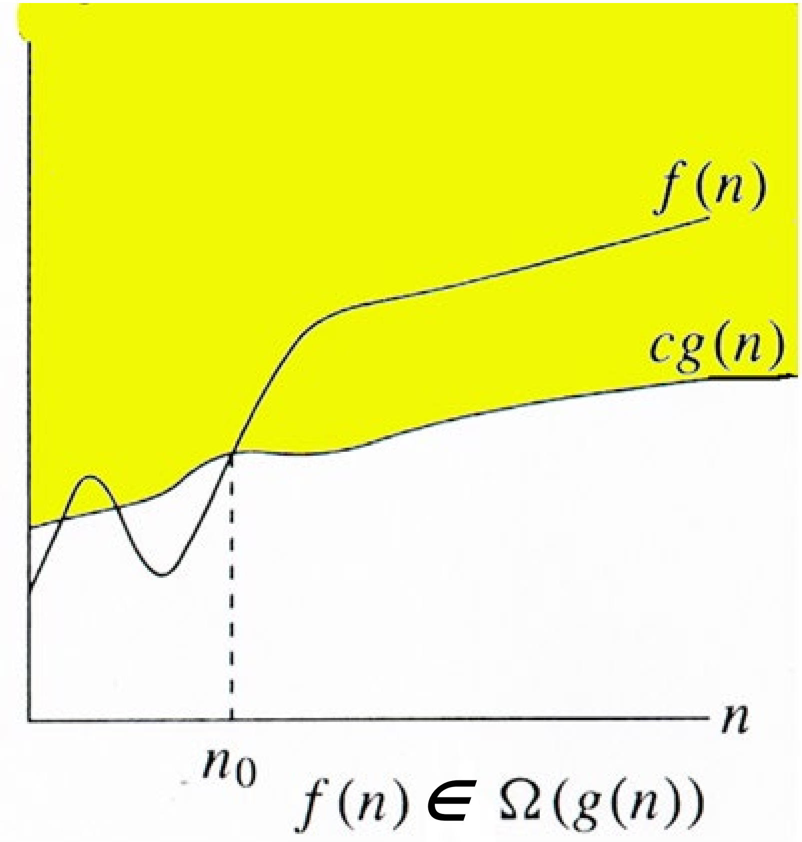
Big Omega



$\Omega(g(n))$

Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

Big Omega



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

Big Omega

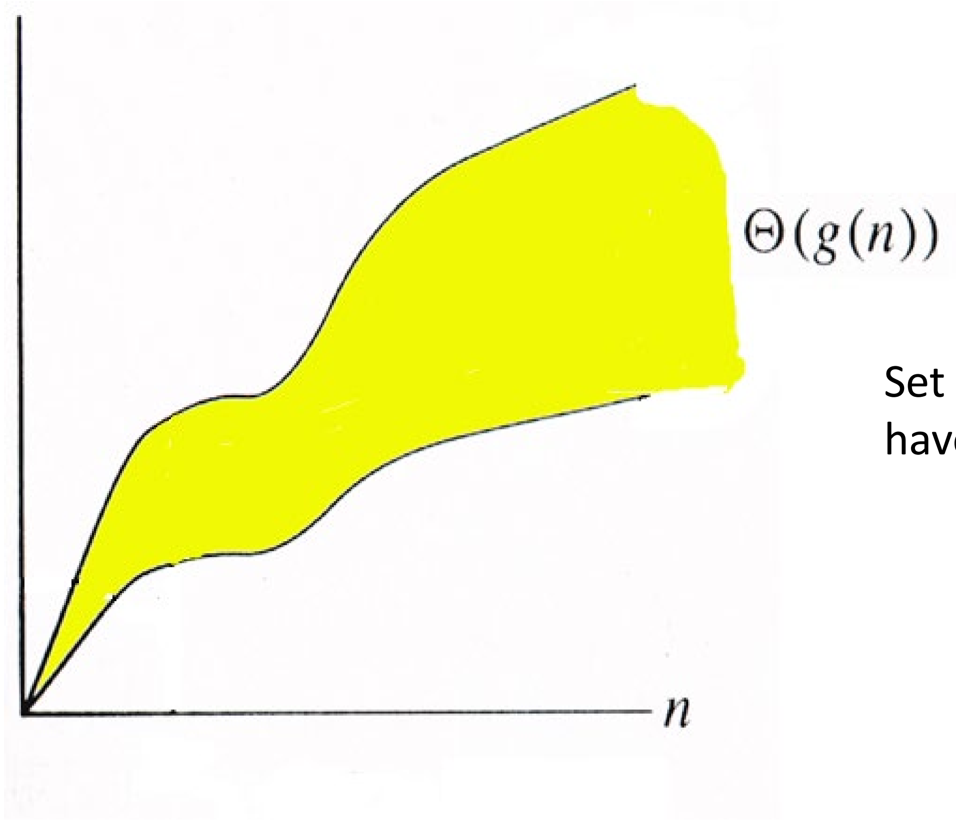
Definition:

- a function **f(n)** is in the set **Ω(g(n))** [*denoted: $f(n) \in \Omega(g(n))$*] if there is a constant **c** and a positive integer $\mathbf{n_0}$ such that

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

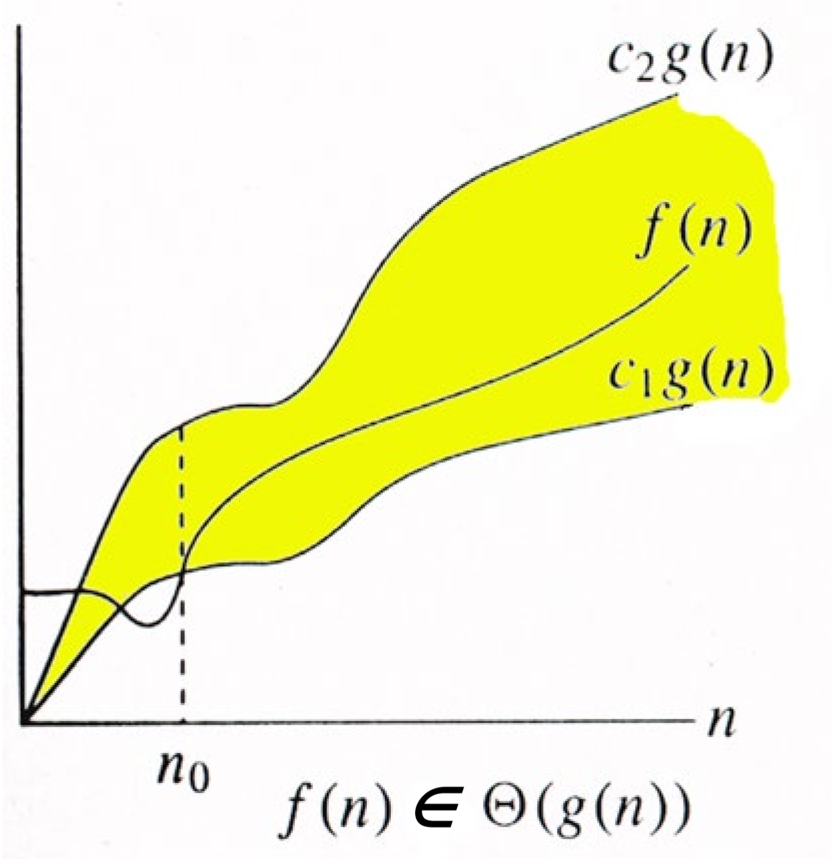
- *i.e.* f(n) is bounded below by some constant multiple of g(n)

Big Theta



Set of all functions that
have the same *rate of growth* as *g*(*n*).

Big Theta



$c_2\, g(n) \leq f(n) \leq c_1\, g(n)$, for all $n \geq n_0$

Big Theta

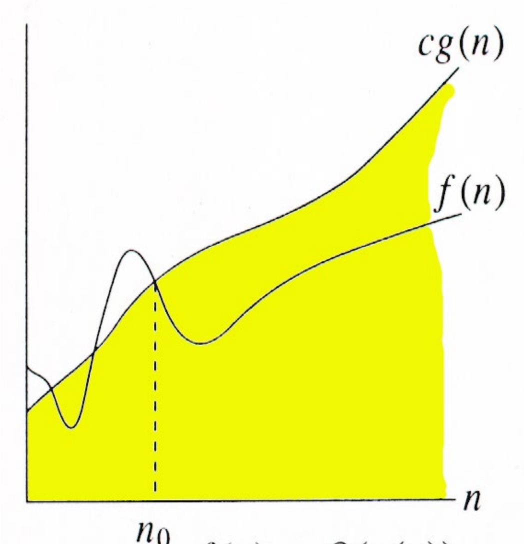
<u>Definition:</u>

- a function **$f(n)$** is in the set **$\Theta(g(n))$** [*denoted: $f(n) \in \Theta(g(n))$*] if there are constants **c_1 and c_2**, and a positive integer **n_0** such that

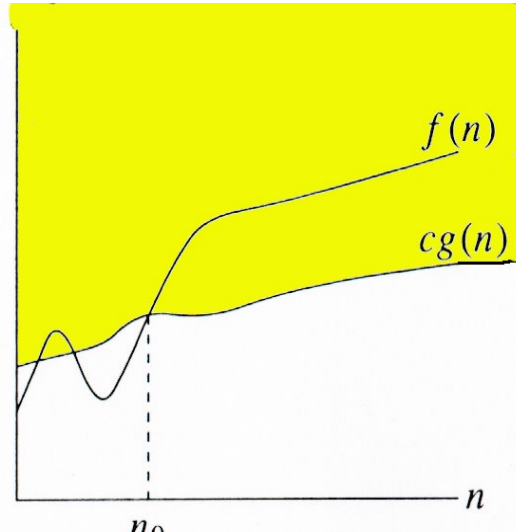
$$c_2\, g(n) \le f(n) \le c_1\, g(n)\ ,\ \text{for all } n \ge n_0$$

- *i.e.* f(n) is bounded both above and below by constant multiples of g(n)

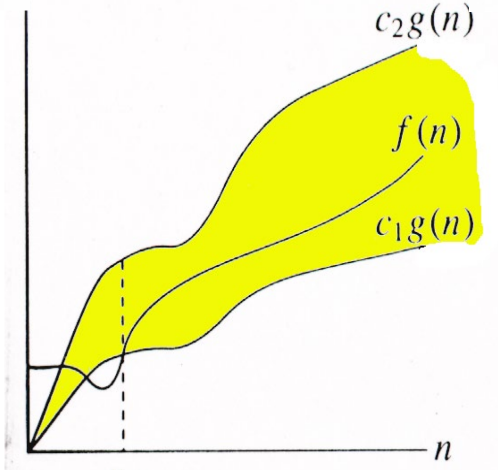
Summary of notations - pictorial



$f(n) = O(g(n))$



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



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

Summary of notations - intuition

- Big-O → execution will take *at MOST that long*
- Big-Ω → execution will take *at LEAST that long*
- Big-Θ → execution *will take THAT long*

In general…

- We will usually focus on Big-O
- Why?
  - Focuses on worst case efficiency
  - Most common when people talk about algorithms

Examples

What is the efficiency class of the following functions?

- $10n$ O(n)
- $5n^2 + 20$ $O(n^2)$
- $10000n + 2^n$ $O(2^n)$
- log(n) * (1 + n) O(nlog(n))

Example 1

- Problem: find the max element in a list
- Input size measure:
  - *Number of list items, i.e. n*
- Basic operation:
  - *Comparison*

```
ALGORITHM  MaxElement(A[0..n − 1])
    maxval ← A[0]
    for i ← 1 to n − 1 do
        if A[i] > maxval
            maxval ← A[i]
    return maxval
```

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in O(n)$$

Example 2

- Problem: *Multiplication of two matrices*
- Input size measure:
  - *Matrix dimensions or total number of elements*
- Basic operation:
  - *Multiplication of two numbers*

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$
    for $i \leftarrow 0$ **to** $n-1$ **do**
        for $j \leftarrow 0$ **to** $n-1$ **do**
            $C[i, j] \leftarrow 0.0$
            for $k \leftarrow 0$ **to** $n-1$ **do**
                $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
    return C

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = n^3 \in O(n^3)$$

Example 3: Element uniqueness problem

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

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
```

Example 3

- Parameter for input size:

  n, the size of the array

- Basic operation:

  Comparison in the innermost loop

- Worst case efficiency count… nested loop:

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

$$= \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i = n(n-1) - (n-1) - (n-2)(n-1)/2$$

$$= n^2 - n - n + 1 - n^2/2 + 3n/2 - 1$$

$$= n^2/2 - n/2 \in O(n^2)$$

Practice problems

- Chapter 1.1 page 8, question 5
- Chapter 1.2 page 18, question 9
- Chapter 1.3 page 23, question 1
- Chapter 2.1, page 50, question 2
- Chapter 2.2, page 60, question 5
- Chapter 2.3, page 68, questions 5, 6

More practice problems

For each of the following simple algorithms determine:

a. its basic operation
b. basic operation count
c. if basic op count depends on input form

1. Computing the sum of a set of numbers
2. Computing n! (n factorial)
3. Checking whether all elements in a given array are distinct