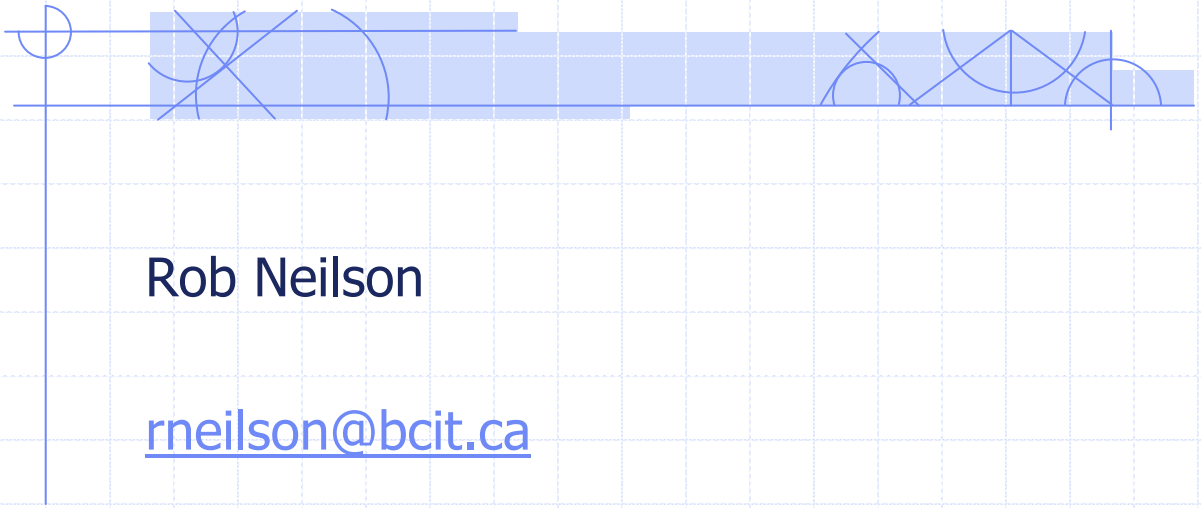
COMP 3760: Algorithm Analysis and Design

Lesson 3: Orders of Growth



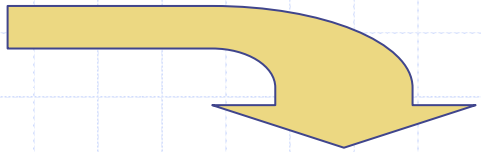
Rob Neilson

rneilson@bcit.ca

What did we learn last lesson?

1. the **size of input** is what (typically) determines how fast an algorithm will run
2. for any algorithm we can identify a **basic operation** which is the operation on which the algorithms performance depends
3. we can study an algorithm and express the count of its basic operations as a polynomial equation, for example:

```
1. Loops(A[0..n-1])
2.  for i ← 1 to n-1 do
3.    v ← A[i]
4.    j ← i-1
5.    while j≥0 and A[j]>v do
6.      A[j+1] ← A[j]
7.      j ← j-1
8.    A[j+1] ← v
```



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

General Strategy for Analysis of Non-recursive Algorithms

This strategy is taken from page 62 of your textbook:

1. Decide on a parameter indicating the inputs size.
2. Identify the algorithms basic operation.
3. Check whether 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 standard formulas and rules of sum manipulation to find a closed form formula f(n) for the sum from step 4 above.
6. Determine the efficiency class of the algorithm by bounding its order of growth in the worst case, best case, and/or average case.

Orders of Growth

- for small input sizes, any old algorithm should do; it does not necessarily have to be efficient
- but, algorithms tend to show their limitations as the size of the input gets large
- consider table 2.1 from your textbook:

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	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
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}		

these represent possible functions that count basic ops in an algo

it would take more than 5 billion years to make 100! calculations*

* - on at 1000 MHz CPU

Action Break 1: Basic Operation Counts

1. With a partner, discuss and answer question 1, page 50 ... ie:

For each of the following algorithms determine:

- a natural size metric for its input
- its basic operation
- if basic op count can be the different for inputs of same size

(a) computing the sum of n numbers

(b) computing n! (n factorial)

(c) determining if a specific number exists in an array of n numbers

Action Break 1 Answers

a. computing the sum of n numbers

natural size metric = n;
basic op = addition of two numbers;
op count differs for same size inputs = no

b. computing n!

natural size metric = the magnitude of n;
basic op = multiplication of two integers;
op count differs for same size inputs = no

c. finding the largest element in a list of n numbers

natural size metric = n;
basic op = comparison of two numbers;
op count differs for same size inputs = yes

What are: Worst Case, Best Case, Average Case Analysis ?

- from the previous question ...
  - it asked '***does the count of basic operations differ for inputs of the same size***'

Why is this important?

  - the performance of many algorithms depends on the actual input data that is used
  - the textbook uses an example of sequential search
    - (ie: a linear scan of an unsorted array for a specific target)

Consider Sequential Search ...

  - What is the **best case performance** for an input size of, say, 100?
  - What type of input induces this behaviour?
    - best case performance is ***1 comparison***, ie, when the input is organized such that the ***search target is in the first element of the array***

Worst, Best, Average Case Analysis (cont)

- still considering sequential search ...
  - What is the **worst case performance** for an input size of, say, 100?
  - What type of input induces this behaviour?
    - worst case performance is ***100 comparisons***, ie, when the input is organized such that the ***search target does not exist in the array***
    - *Note: we can generalize the worst case for inputs of size n, and say that it will take n comparisons in the worst case*
  - What is the **average case performance** for an input size of, say, 100?
    - we assume the probability is equal that the target will be in any position in the array
    - in this case, the average # comparisons is ***(1+2+3+…+99+100)/100***
    - *Note: we can generalize the average case for inputs of size n, and say that it will take on average (n+1)/2 comparisons*

Which to use: best, worst, average

- when analyzing an algorithm, we need to consider the specifics of how it is to be used to determine the relative importance of best/worst/average case
- probably average case analysis is the most useful for **non-mission-critical** applications
- for **mission-critical stuff**, we always want to look at the worst case
- since average case analysis can get quite complicated, ***we will focus on worst-case analysis in this course***
  - unless otherwise specified, you should always analyze the worst case

A final note ...

... on best/worst/average case ...

- there are many situations where:

  best case == worst case == average case

- consider the algorithm to find the largest element in an unordered array

  – if the input size is 1000, the algorithm will always need to make 999 comparisons, no matter what the 1000 elements are or how they are arranged

Action Break 2

1. Working with someone or by yourself, answer question 3 on page 51, ie:

   Consider a variation of sequential search that scans a list to return the number of occurrences of a given search key in the list.

   What is the best/average/worst case efficiency for this algorithm and how do these values compare to the efficiency of classic sequential search?

Action Break 2 Answer

This "count number of occurrences" algorithm will always make n key comparisons on every input of size n, whereas this number may vary between n and 1 for the classic version of sequential search.

Count Occurrences: best = worst = average = n

Sequential Search: best = 1 worst = n average = (n+1)/2

Asymptotic Notations

- the asymptotic notations O(*g(n)*), Ω(*g(n)*), and Θ(*g(n)*) provide us with a generalized classification for algorithms
- these notations work like this:
  1. each notation (eg: O(*g(n)*)) is a set of functions
  2. all the functions that belong to a set have their order of growth bounded by *g(n)*, as n goes to infinity

  – for example: consider $g(n)=n^2$ (ie: $O(n^2)$)

    - the following functions are all members of the set $O(n^2)$

      $3n+1$ $6n^2$ $3n\log n$ $n^2+8n+16$

  – we could say ...
    - $O(n^2)$ = {3n+1, $6n^2$, 3nlogn, $n^2+8n+16$ ... <plus many more functions> }

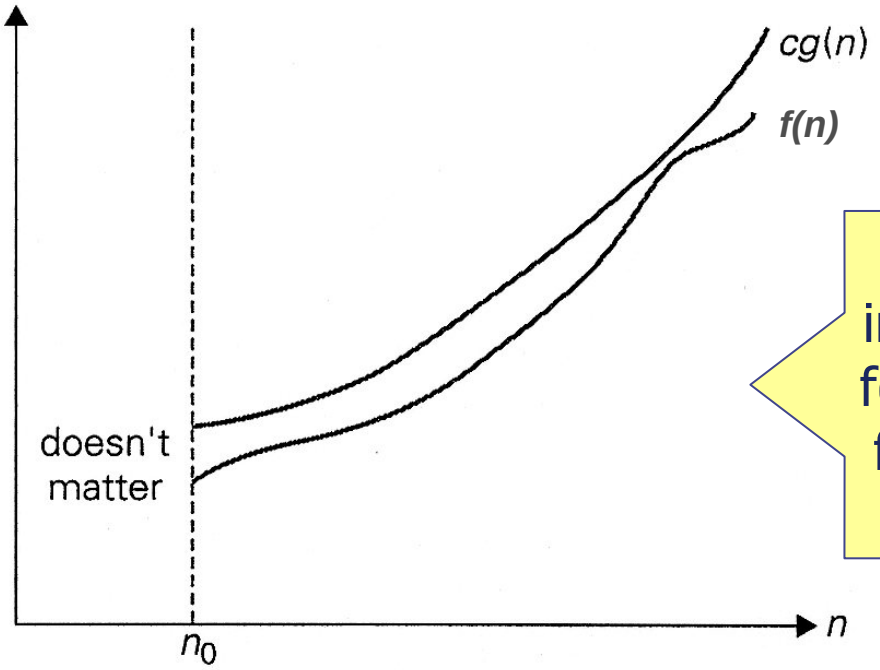
Big-Oh (formal definition)

Definition:

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

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

- ie: f(n) is bounded above by some constant multiple of g(n)



this tells us:
in the worst case
f(n) will not grow
faster than cg(n)
for large n

FIGURE 2.1 Big-oh notation: $f(n) \in O(g(n))$

Big-Oh (formal definition - example)

Definition:

- 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

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

- ie: f(n) is bounded above by some constant multiple of g(n)

Consider an example:

- assume that a count of basic operations, f(n), is:

  $\mathbf{f(n) = 3n^2 - 5n + 20}$

  - we might ask: **is $f(n) \in O(n^2)$?** ie, **is $3n^2 - 5n + 20 \in O(n^2)$**
  - we can determine this by applying the definition of O(g(n)) ...

$$\mathbf{3n^2 - 5n + 20 \leq cn^2 \text{ , for all } n \geq n_0}$$

continued next slide ...

Big-Oh (applying the formal definition)

$3n^2 - 5n + 20 \leq cn^2$, for all $n \geq n_0$

- to test if the given functions satisfy the formal definition, we have to find a **c** and a **n_0** that make the equation true

- assume **c=3**, then we have: $3n^2 - 5n + 20 \leq 3n^2$

  Is this equality true?

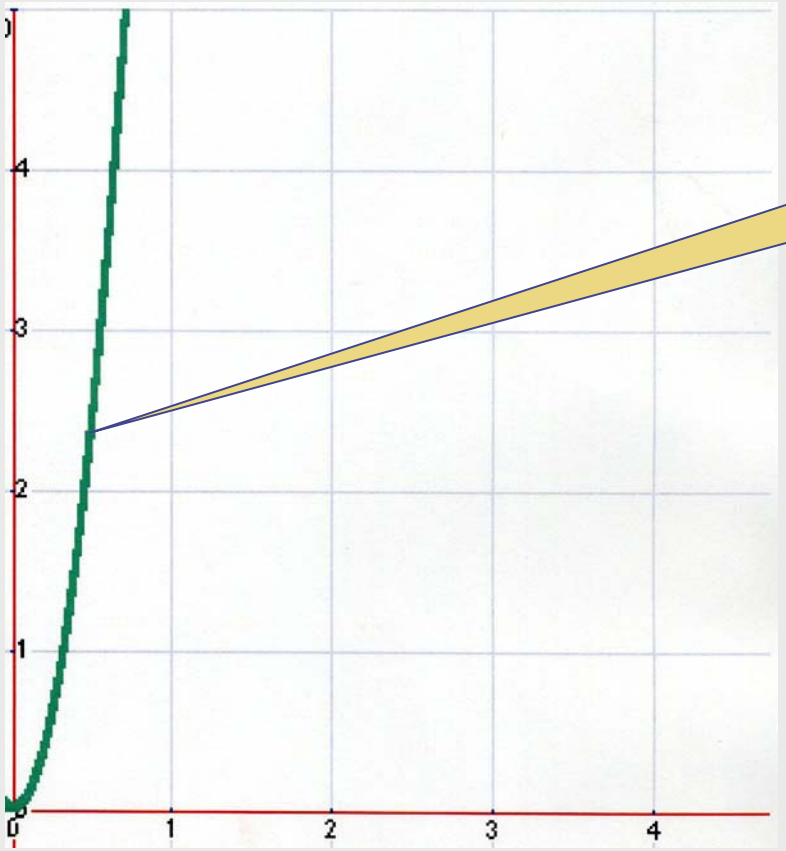
  - YES, it is true, as long as $n \geq 4$
  - therefore we can say:

    $3n^2 - 5n + 20 \leq cn^2$ for c=3 and $n \geq 4$

    Therefore, by definition,

    $3n^2 - 5n + 20 \in O(n^2)$

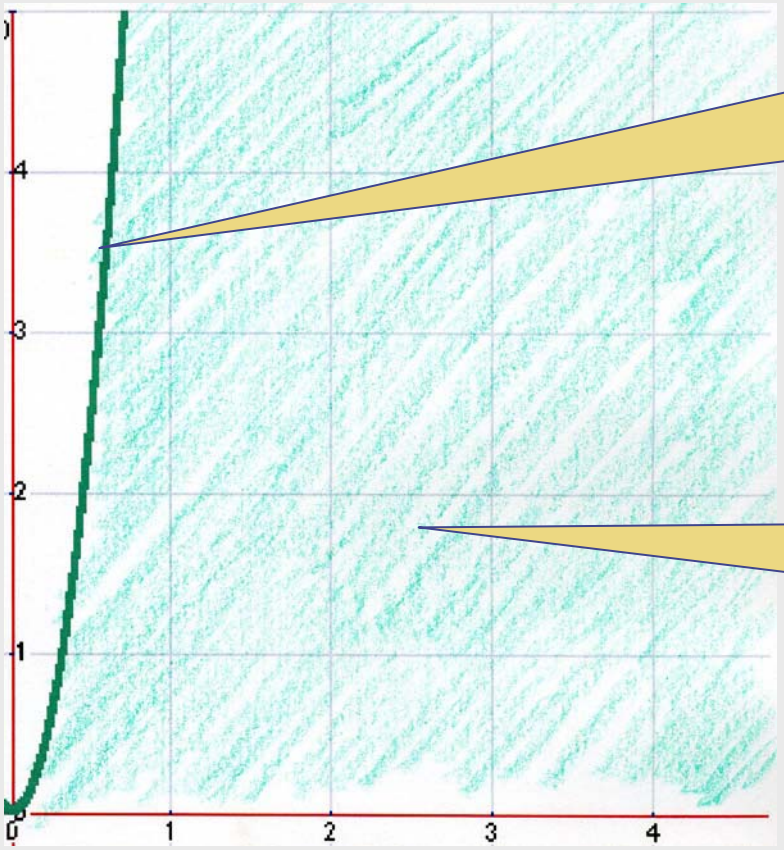
Big-Oh (graphically)



f(n) = n^2, with c=10

(ie: this is the graph of $10n^2$)

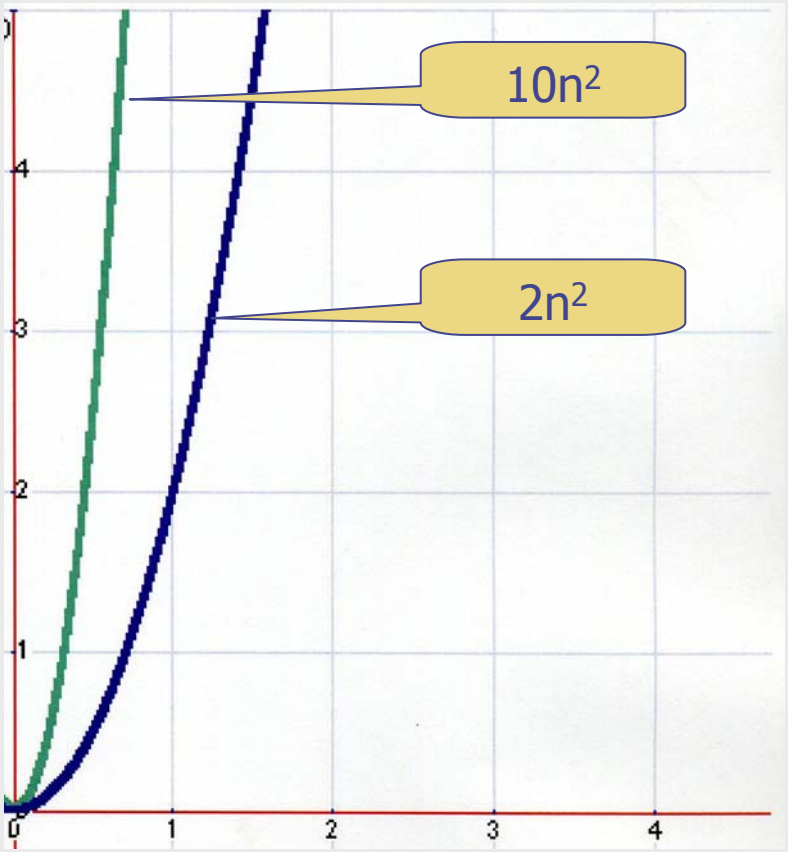
Big-Oh (graphically)



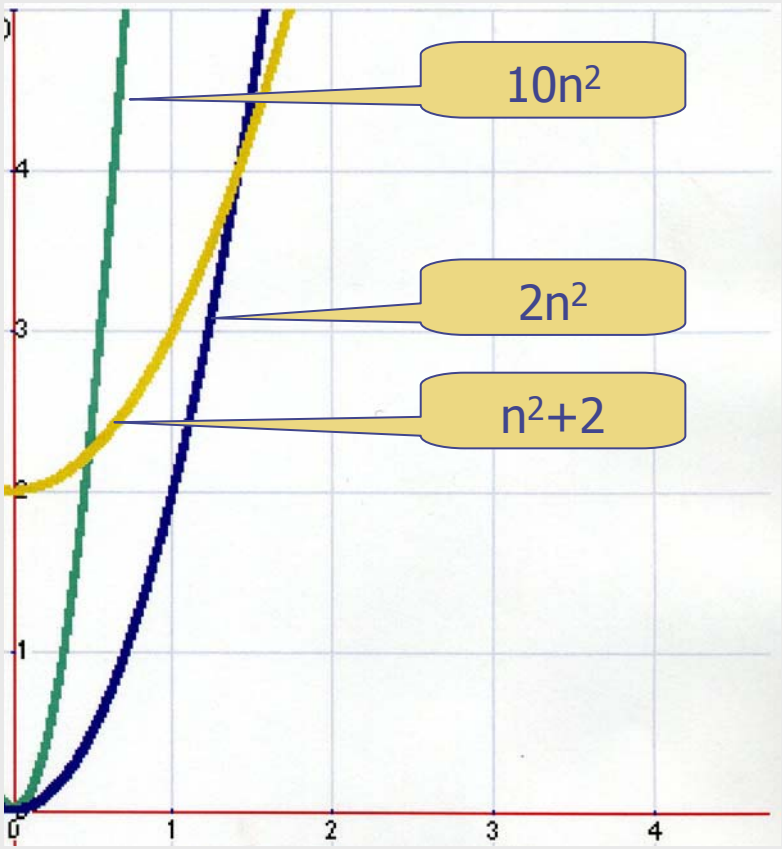
we can visualize the set $O(n^2)$ as all functions whose order of growth is bounded by this line

any f(n) in this shaded area is also a member of the set $O(n^2)$

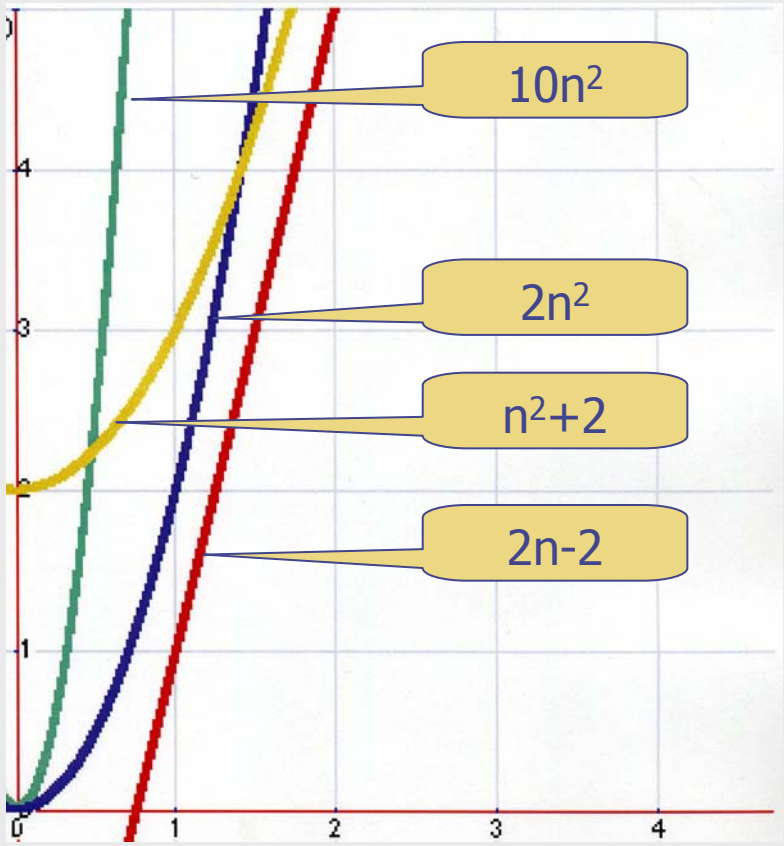
Big-Oh (graphically)



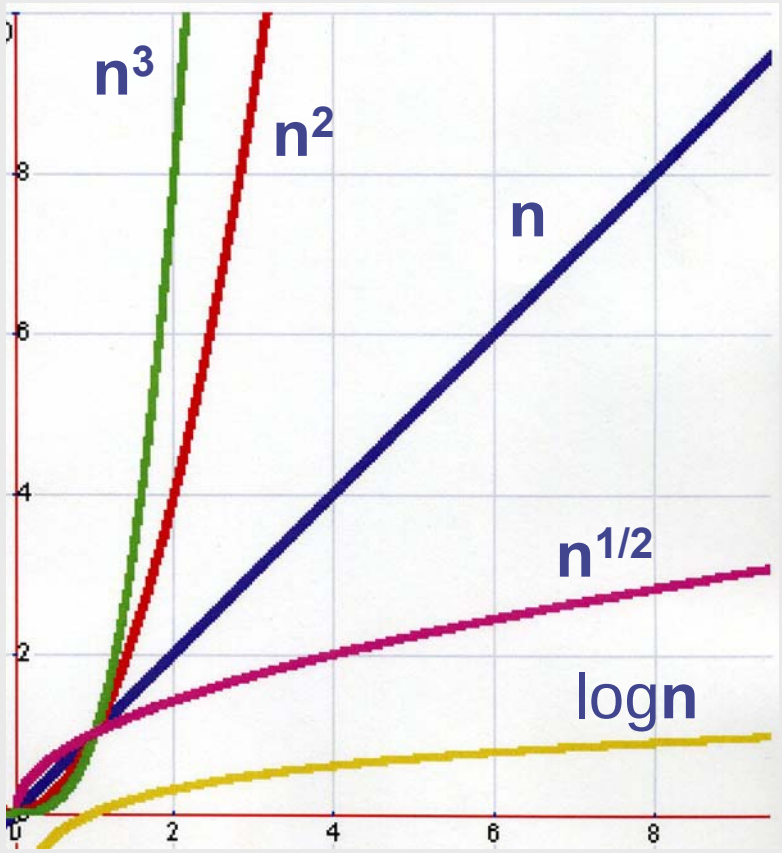
Big-Oh (graphically)



Big-Oh (graphically)



Big-Oh – rates of growth



you can see how fast the number of basic operations increases, even for quadratic and cubic functions.

the growth of the exponential functions is too steep to show here!

Base Efficiency Classes (part 1)

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.

Base Efficiency Classes (part 2)

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.

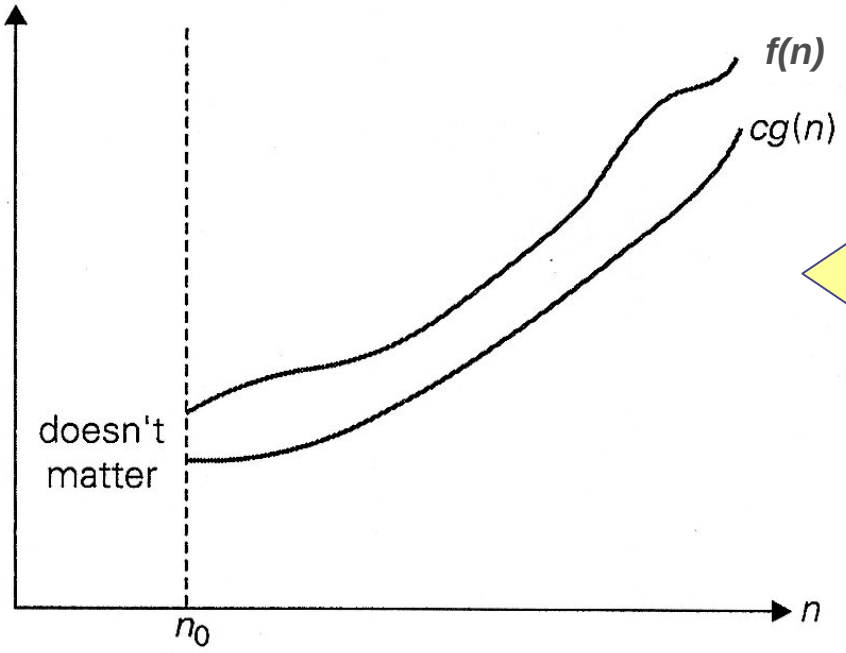
Big Omega

Definition:

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

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

- ie: f(n) is bounded below by some constant multiple of g(n)



this tells us:
in the best case
f(n) will not grow
slower than cg(n)

FIGURE 2.2 Big-omega notation $f(n) \in \Omega(g(n))$

Big Theta

Definition:

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

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

- ie: f(n) is bounded both above and below by constant multiples of g(n)

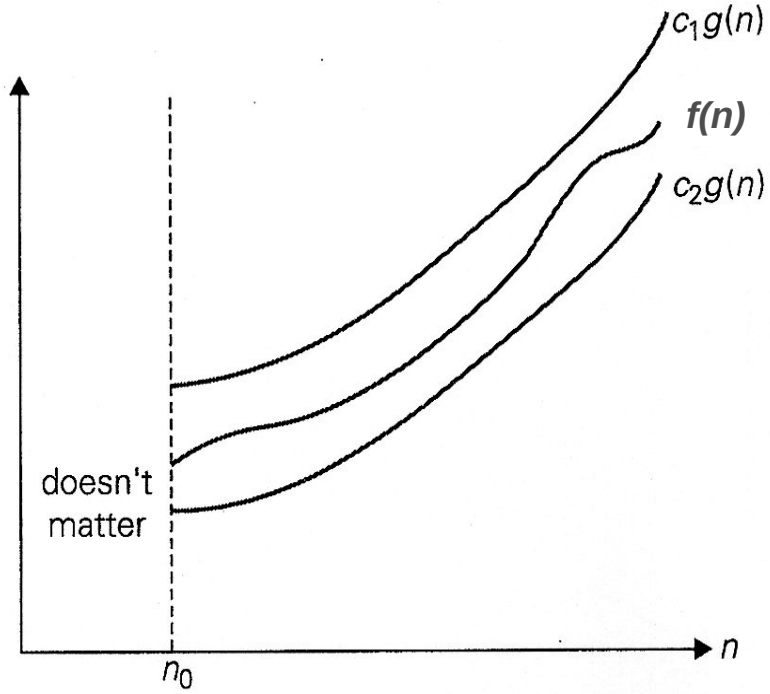


FIGURE 2.3 Big-theta notation: $f(n) \in \Theta(g(n))$

this tells us that the best and worst case efficiencies of the algorithm are roughly the same

Homework from this week's lessons

(due start of lab next week)

Reading:

- review chapters 2.1, 2.2, 2.3

Homework:

- chapter 2.1, page 51, question 6
- chapter 2.2, page 60, question 2
- chapter 2.3, page 67, question 1, parts a, b, c, d
- chapter 2.3, page 67, question 2, parts a, b
- chapter 2.3, page 67, question 10, part a

The End

The End!