CSC 172– Data Structures and Algorithms

Lecture #5

Spring 2018

STOP

Please go through Chapter 4 before you proceed

http://lti.cs.vt.edu/LTI_ruby/Books/CS172/html/#algorithm-analysis

Asymptotic Analysis

we will look at:

- Justification for analysis
- Concept of a growth rate
  - the rate at which the cost of an algorithm grows as the size of its input grows
- Asymptotic Notation symbols
- The concept of an upper bound and lower bound for a growth rate
  - How to estimate these bounds for a simple program
    - Counting machine instructions
- Examples

Background

Suppose we have two algorithms, how can we tell which is better?

We could implement both algorithms, run them both

– Expensive and error prone

Preferably, we should analyze them mathematically

– *Algorithm analysis*

Asymptotic Analysis

In general, we will always analyze algorithms with respect to one or more variables

We will begin with one variable:

- The number of items n currently stored in an array or other data structure
- The number of items expected to be stored in an array or other data structure
- The dimensions of an $n \times n$ matrix

Examples with multiple variables:

- Dealing with n objects stored in m memory locations
- Multiplying a $k \times m$ and an $m \times n$ matrix
- Dealing with sparse matrices of size $n \times n$ with m non-zero entries
- Dealing with Graphs

Asymptotic Analysis

Given an algorithm:

- We need to be able to describe these values mathematically
- We need a systematic means of using the description of the algorithm together with the properties of an associated data structure
-
- We need to do this in a machine-independent way

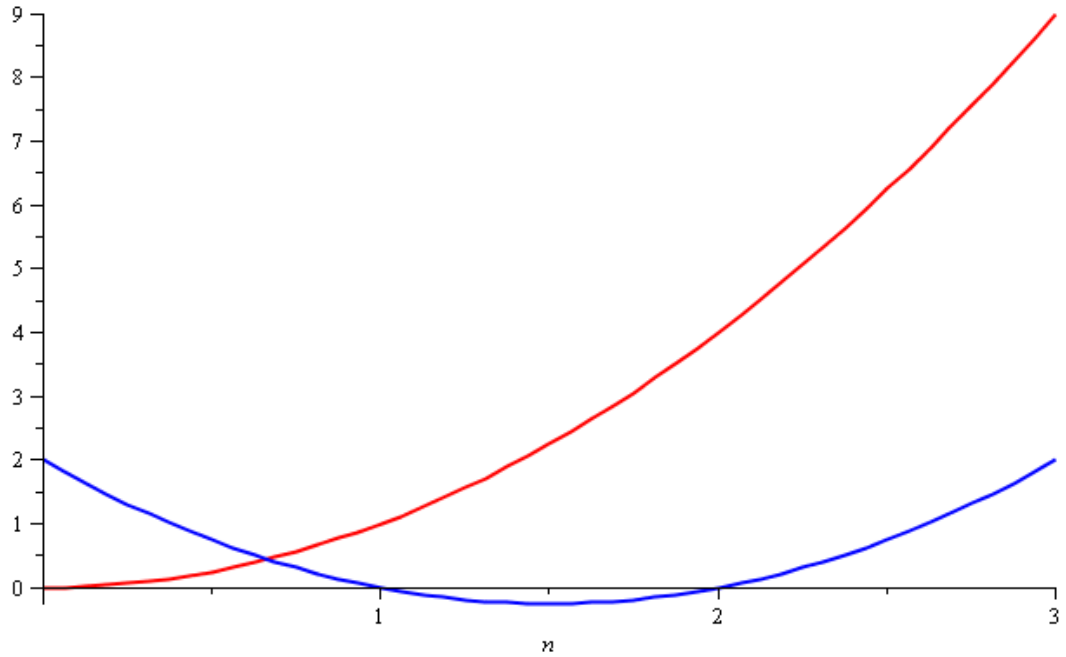
For this, we need Landau symbols and the associated asymptotic analysis

Quadratic Growth

Consider the two functions

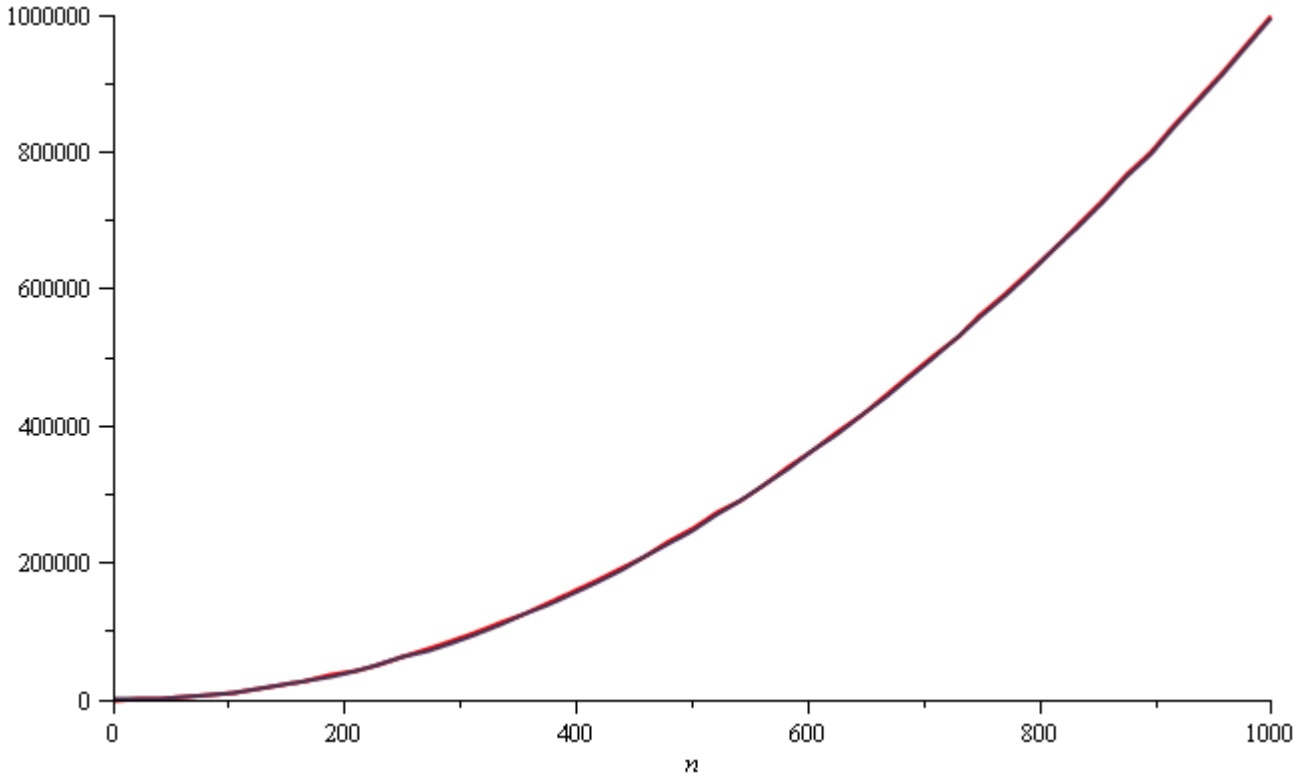
$f(n) = n^2$ and $g(n) = n^2 - 3n + 2$

Around $n = 0$, they look very different



Quadratic Growth

Yet on the range $n = [0, 1000]$, they are (relatively) indistinguishable:



Quadratic Growth

The absolute difference is large, for example,

$$f(1000) = 1\ 000\ 000$$

$$g(1000) = \ \ 997\ 002$$

but the relative difference is very small

$$\left|\frac{f(1000) - g(1000)}{f(1000)}\right| = 0.002998 < 0.3\%$$

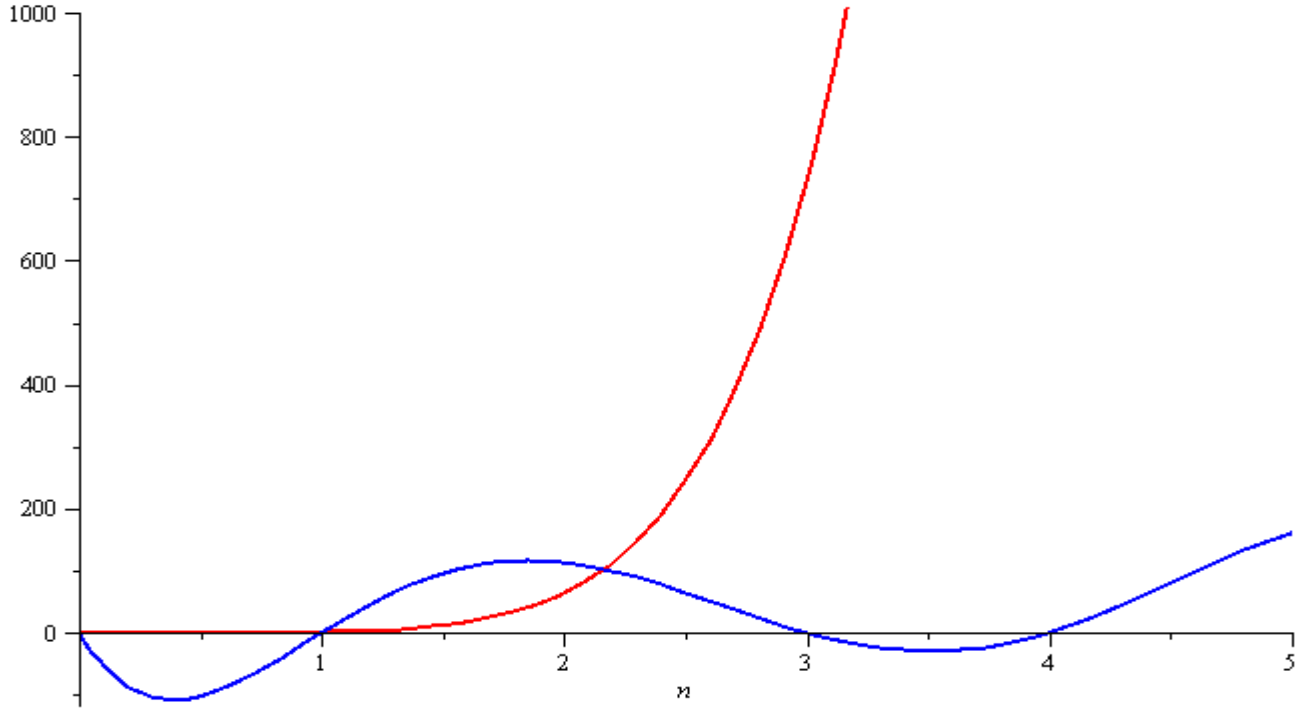
and this difference goes to zero as $n \to \infty$

Polynomial Growth

To demonstrate with another example,

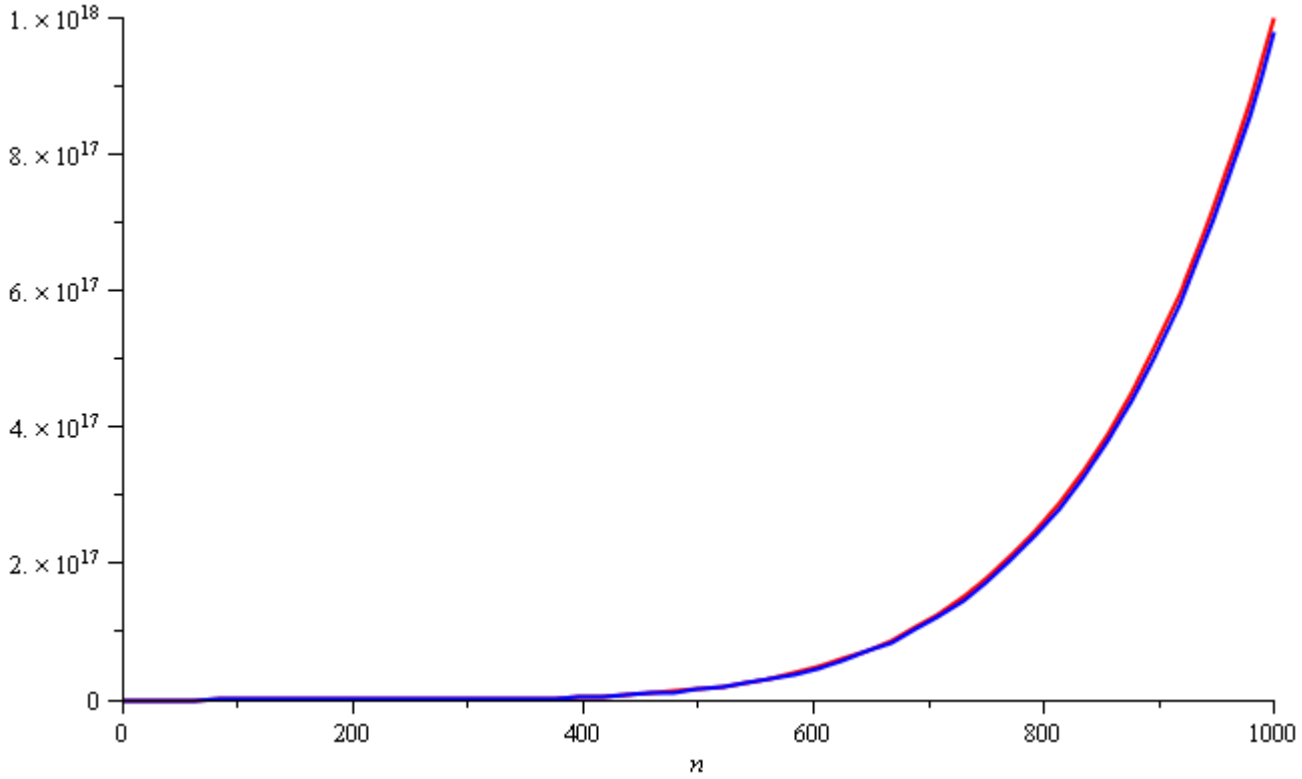
$f(n) = n^6$ and

$g(n) = n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$



Polynomial Growth

Still, around $n = 1000$, the relative difference is less than 3%



Polynomial Growth

The justification for both pairs of polynomials being similar is that, in both cases, they each had the same leading term:

n^2 in the first case, n^6 in the second

Suppose however, that the coefficients of the leading terms were different

– In this case, both functions would exhibit the same rate of growth, however, one would always be proportionally larger

Counting Instructions

Because we can count the number instructions, we can also estimate how much time is required to run one of these algorithms on a computer

Example

For example, the time taken to find the difference between the maximum and minimum element in an array of n random integers will take $n + b$ operations, where b is a constant.

```
public static int diffMaxMin(int[] anArray) {

        if (anArray.length <= 1)
            return 0;

        int max,min;
        max =  min = anArray[0];
        for (int val: anArray) {
            if (val > max) max = val;
            if (val < min) min = val;
        }
        return max - min;
    }
```

T(n) = c (n + b)
= cn + b'
(b' is another constant)

Most of the times, we ignore constant terms.
So,
T(n) = cn

Example 1

```
a = b;
```

$T(n) = c_1$
Constant Running time

Example 2

```
sum = 0;
for (i=1; i<=n; i++)
    sum += n;
```

$T(n) = cn$
Linear Running Time

Example 3

```
sum = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    sum++;
```

$T(n) = cn^2$

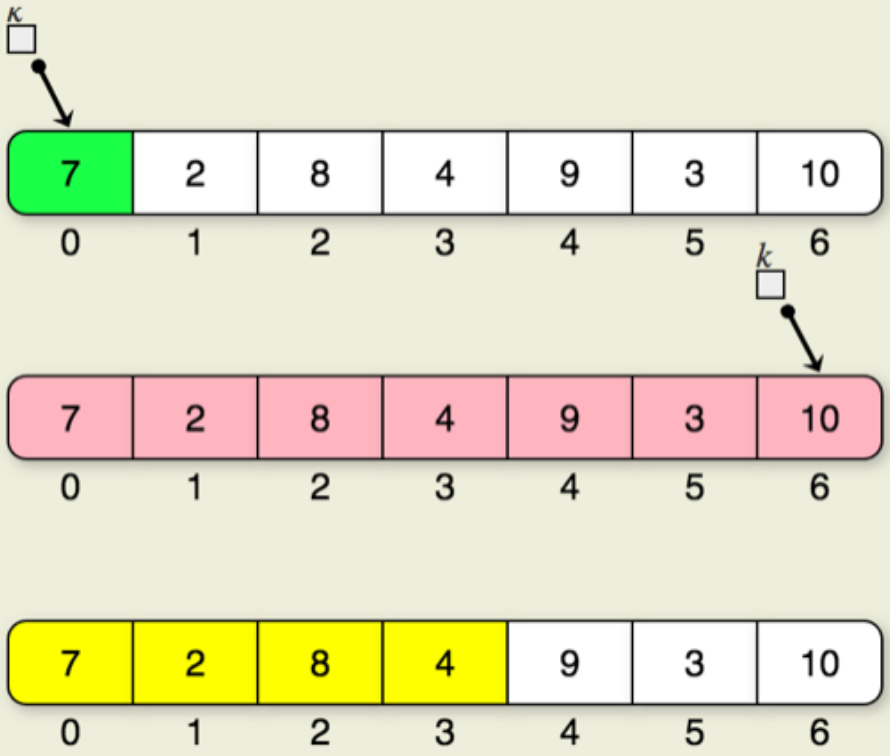
Quadratic Running Time

Example 4

```
sum = 0;
for (j=1; j<=n; j++)        // First for loop
   for (i=1; i<=j; i++)     //   is a double loop
      sum++;
for (k=0; k<n; k++)         // Second for loop
   A[k] = k;
```

$$T(n) = c_1 \frac{n(n+1)}{2} + c_2 n$$

Linear Search: Best, Worst, and Average Case



Best Case. A single comparison is performed.

Worst Case. n comparisons are performed.

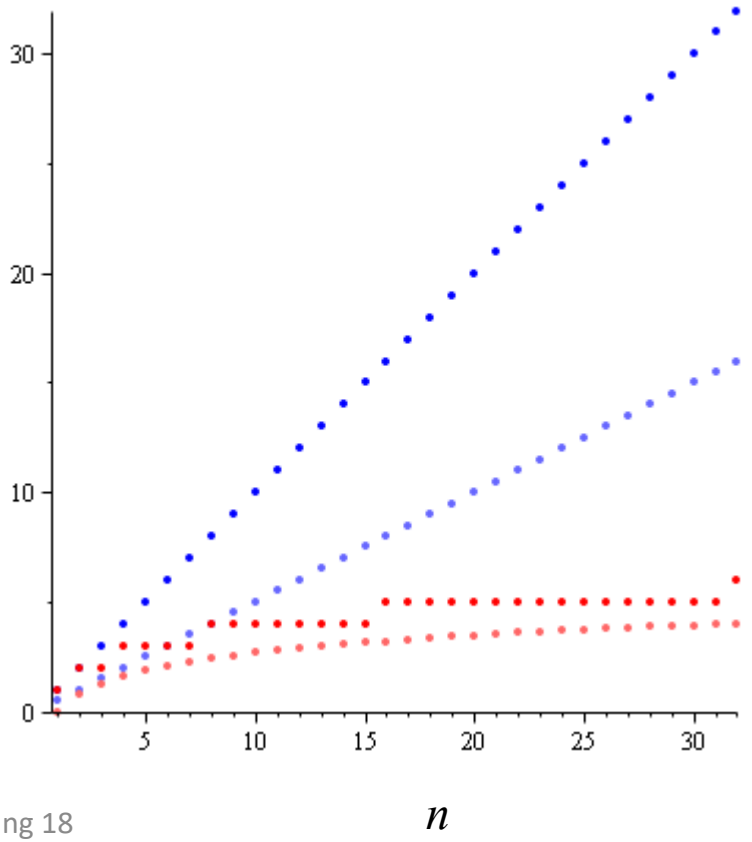
Average Case. $\frac{n+1}{2}$ comparisons are performed.

Linear and binary search

There are other algorithms which are significantly faster as the problem size increases

This plot shows maximum and average number of comparisons to find an entry in a sorted array of size n

- Linear search
- Binary search



Algorithms Analysis

We will use Landau symbols to describe the complexity of algorithms

– E.g., adding a list of n doubles will be said to be a $\Theta(n)$ algorithm

ASYMPTOTIC ANALYSIS

- Back of the envelope time/space estimation
- Independent of whether our computer is fast
- Big-o, big-omega, theta

ASYMPTOTIC ANALYSIS

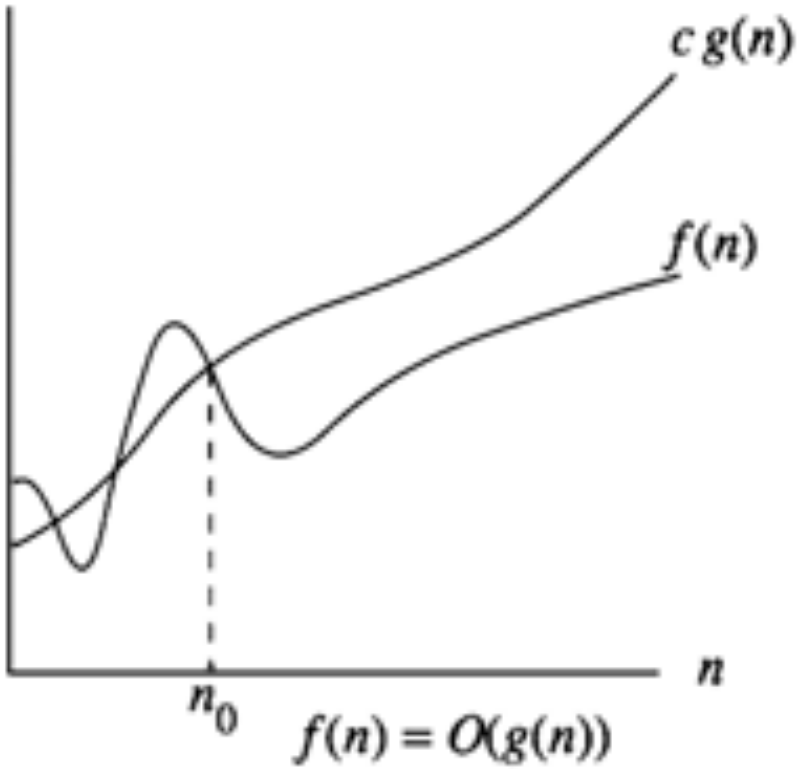
Big-O

$$f, g : \mathbb{N} \to \mathbb{R}^+$$

$$f(n) = O(g(n)) \text{ iff } \exists \text{ constants } C, n_0 > 0$$

$$\text{such that } f(n) \leq Cg(n), \forall n \geq n_0$$

Intuition



In English

- *f(n) = O(g(n))* means: for n sufficiently large, *f(n)* is bounded above by a constant scaling of *g(n)*
  - Does the "English translation" make things worse?

- An algorithm with runtime *f(n)* is at least as good as an algorithm with runtime *g(n)*, asymptotically

Examples

$$n^2 = O(n^2)$$

$$n^2 = O(n^2/10^6)$$

$$n = O(n^2)$$

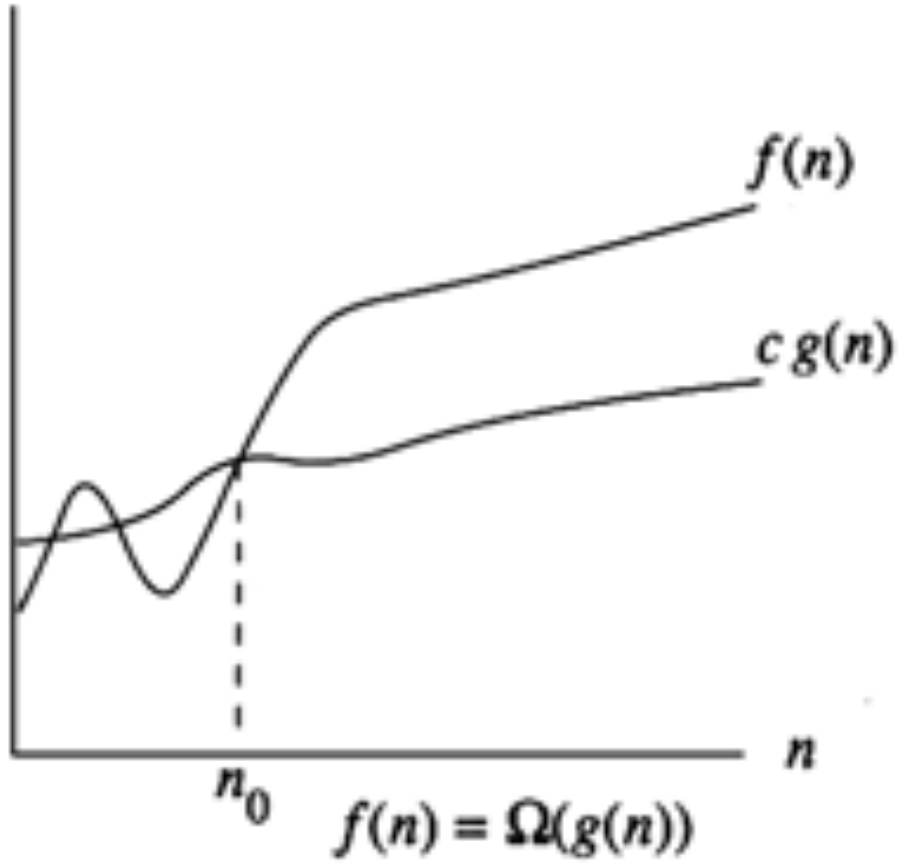
Big-Omega

$$f, g : \mathbb{N} \to \mathbb{R}^+$$

$$f(n) = \Omega(g(n)) \text{ iff } \exists \text{ constants } C, n_0 > 0$$

$$\text{such that} f(n) \geq Cg(n), \forall n \geq n_0$$

In picture



Examples

$$n \log n = \Omega(n)$$

$$2^n / 10^6 = \Omega(n^{100})$$

Equivalence

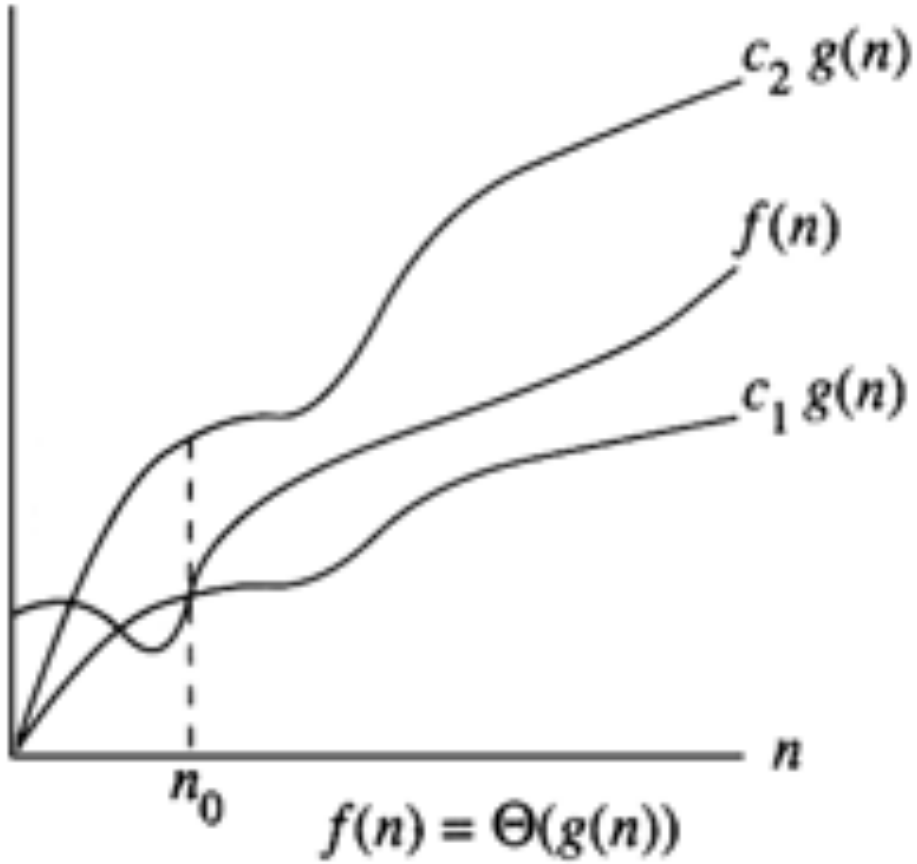
$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

Theta

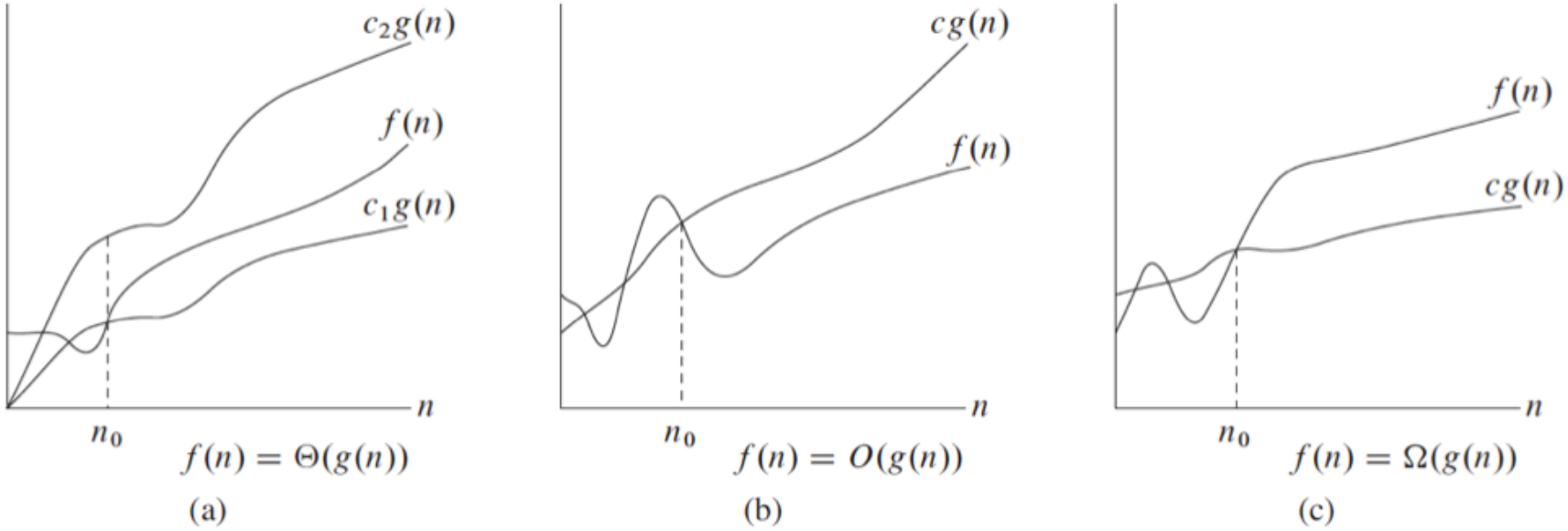
$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n) \text{ and } g(n) = O(f(n))$$

We say they "have the same growth rate"

In picture



$O, \Omega, and\ \Theta$



What is the runtime?

```
// Return position of largest value in integer array A
static int largest(int[] A) {
  int currlarge = 0;                // Position of largest element seen
  for (int i=1; i<A.length; i++) // For each element
    if (A[currlarge] < A[i])        //   if A[i] is larger
      currlarge = i;                //     remember its position
  return currlarge;                 // Return largest position
}
```

O(n) or $\theta(n)$

From now on, we will talk about run-time in asymptotic notation!

Example

For example, the time taken to find the difference between the maximum and minimum element in an array of n random integers will take $n + b$ operations, where b is a constant.

```
public static int diffMaxMin(int[] anArray) {

        if (anArray.length <= 1)
            return 0;

        int max,min;
        max =  min = anArray[0];
        for (int val: anArray) {
            if (val > max) max = val;
            if (val < min) min = val;
        }
        return max - min;
    }
```

Most of the times, we ignore constant terms. So,

T(n) = O(n)

Example 1

```
a = b;
```

$T(n) = O(1)$
Constant Running time

Example 2

```
sum = 0;
for (i=1; i<=n; i++)
    sum += n;
```

$T(n) = O(n)$
Linear Running Time

Example 3

```
sum = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    sum++;
```

$T(n) = O(n^2)$
Quadratic Running Time

Example 4

```
sum = 0;
for (j=1; j<=n; j++)        // First for loop
   for (i=1; i<=j; i++)     //   is a double loop
      sum++;
for (k=0; k<n; k++)         // Second for loop
   A[k] = k;
```

$T(n) = O(n^2)$

Program Efficiency

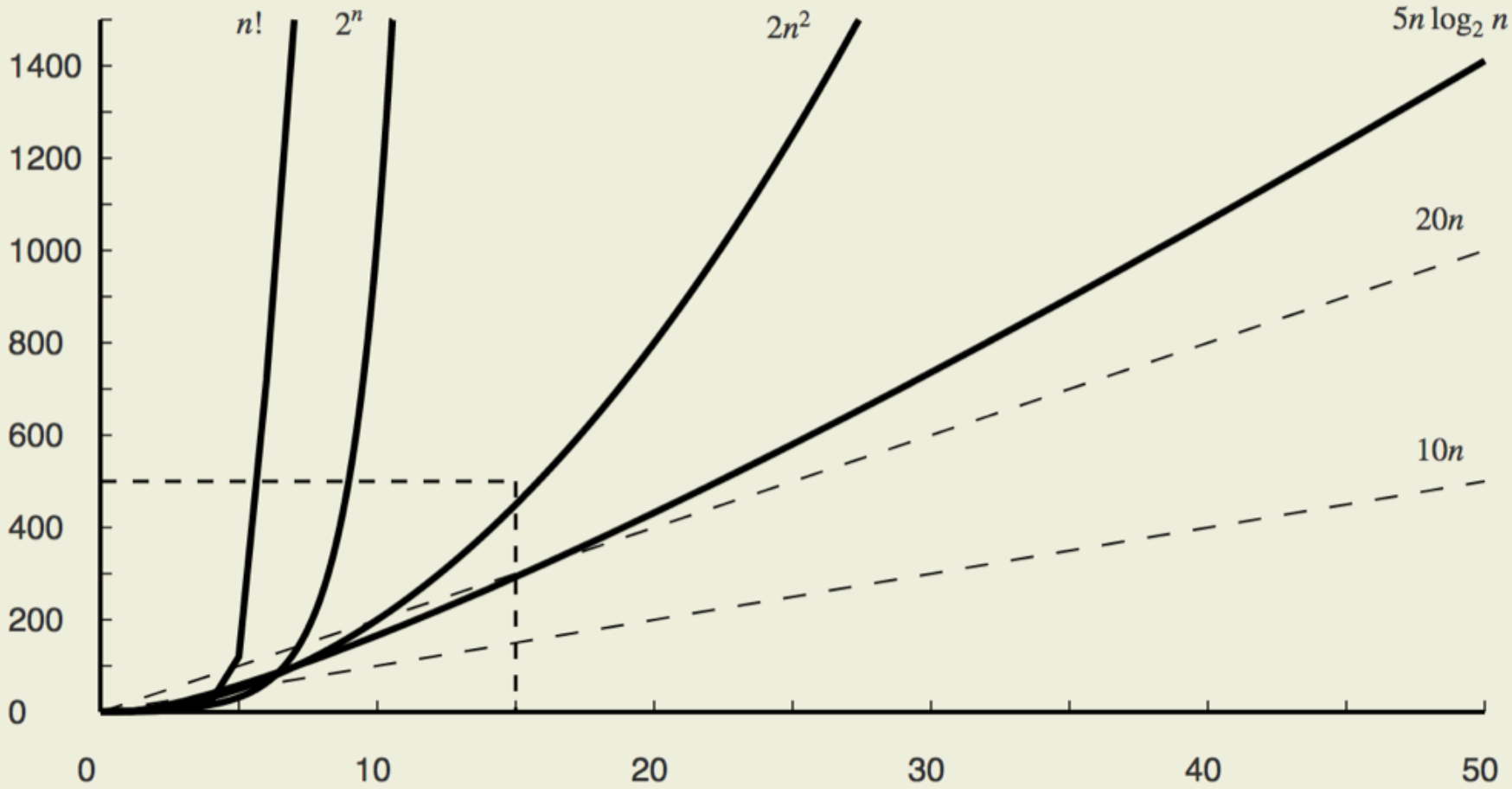
- Faster Computer, or Faster Algorithm?

- Insertion Sort:
  - Best Case: $O(n)$
  - Worst Case: $O(n^2)$

Runtime Matters

n	$\log \log n$	$\log n$	n	$n \log n$	n^2	n^3	2^n
16	2	4	2^4	$4 \cdot 2^4 = 2^6$	2^8	2^{12}	2^{16}
256	3	8	2^8	$8 \cdot 2^8 = 2^{11}$	2^{16}	2^{24}	2^{256}
1024	≈ 3.3	10	2^{10}	$10 \cdot 2^{10} \approx 2^{13}$	2^{20}	2^{30}	2^{1024}
64K	4	16	2^{16}	$16 \cdot 2^{16} = 2^{20}$	2^{32}	2^{48}	2^{64K}
1M	≈ 4.3	20	2^{20}	$20 \cdot 2^{20} \approx 2^{24}$	2^{40}	2^{60}	2^{1M}
1G	≈ 4.9	30	2^{30}	$30 \cdot 2^{30} \approx 2^{35}$	2^{60}	2^{90}	2^{1G}

Runtime Matters



Let's check our understanding

2^n	$5 \log \log n$	$\log^2 n$	$n^{4/3}$	2^{n^2}	$2 \log n$
$n^{4/3}$	$n \log^2 n$	n^4	$7\sqrt{n}$	$2 \log^3 n$	2^n
$\log^2 n$	$n \log^2 n$	2^n	$n \log \log n$	$2^{\sqrt{n}}$	$2 \log n$
$n^{4/3}$	$n \log^2 n$	n^4	$7\sqrt{n}$	$2 \log^3 n$	2^n

Sort them in ascending order (fastest to slowest)

Question

What is the smallest integer k such that $n \log n$ is in $O(n^k)$?

k = 2

True / False

The Sequential Search algorithm is in $O(n^2)$.

TRUE. A better (or more informative) answer would be O(n)

Something you must know!

$$\sum_{i=0}^{\log n} n = n \log n.$$

$$\sum_{i=0}^{\log n} 2^i = \Theta(n)$$

Example 5

```
sum1 = 0;
for (k=1; k<=n; k*=2)     // Do log n times
   for (j=1; j<=n; j++)   // Do n times
      sum1++;
```

$$\sum_{i=0}^{\log n} n = n \log n.$$

$T(n) = O(n \log n)$

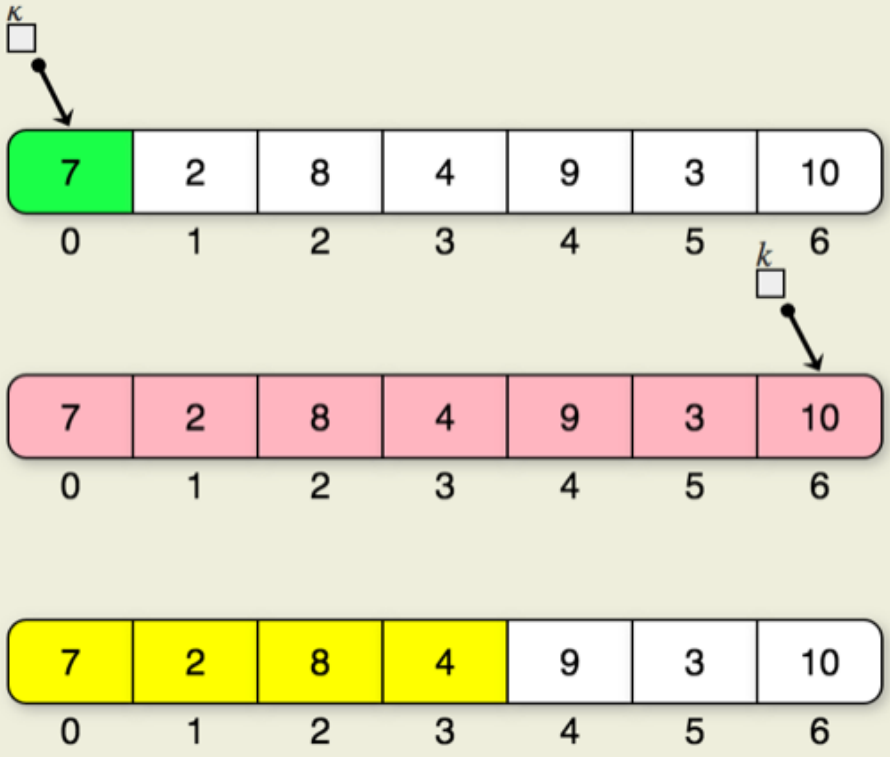
Example 6

```
sum2 = 0;
for (k=1; k<=n; k*=2)     // Do log n times
   for (j=1; j<=k; j++)   // Do k times
      sum2++;
```

$$\sum_{i=0}^{\log n} 2^i = \Theta(n)$$

$T(n) = O(n)$

Linear Search: In Big Oh notation ?



Best Case. A single comparison is performed.

Worst Case. n comparisons are performed.

Average Case. $\frac{n+1}{2}$ comparisons are performed.

ADDITIONAL SLIDES

Faster Computer, or Faster Algorithm?

f(n)	n	n′	Change	n′/n
$10n$	1000	10,000	$n' = 10n$	10
$20n$	500	5000	$n' = 10n$	10
$5n \log n$	250	1842	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	−−

Old Machine vs New Machine (10 times faster):
10,000 operations vs 100,000 operations in one hour

1st Column: Five growth rate equations.
2nd Column: Maximum value of n supported
3rd Column: n′, the new maximum size for the problem supported
4th Column: Change in size
5th Column: increase the increase in the problem size as the ratio of n′ to n.

Asymptotic Analysis for Two functions

1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
2. If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n) + f_2(n)$ is in $O(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.

Summary

- Faster running time does not imply faster algorithm
- For describing the efficiency of an algorithm:
  - You should provide the:
    - Upper Bound (Big Oh) and Lower bound (Big Omega)
- You can ignore lower order terms and constants in asymptotic notations.
- From Now on, we will use only these term describing efficiency of any algorithms!

Acknowledgement

- Douglas Wilhelm Harder.
  - Thanks for making an excellent set of slides for ECE 250 *Algorithms and Data Structures* course

- Prof. Hung Q. Ngo:
  - Thanks for those beautiful slides created for CSC 250 (Data Structures) course at UB.

- Many of these slides are taken from these two sources.