Another Sorting Algorithm

- What was the running time of insertion sort?

- Can we do better?

Designing Algorithms

- Many ways to design an algorithm:
  - Incremental:
  - Divide and Conquer:

Divide and Conquer

- *Divide*
- *Conquer*
- *Combine*

Merge Sort

- Merge Sort is an example of a divide and conquer algorithm

```
MERGE-SORT(A, p, r)
∇ p & r are indices into the array (p < r)
if p < r                         ∇Check for base case
   then q ← ⌊(p + r) / 2⌋        ∇Divide
      MERGE-SORT(A, p, q)        ∇Conquer
      MERGE-SORT(A, q + 1, r)    ∇Conquer
      MERGE(A, p, q, r)          ∇Combine
```

Example

- How would the following array (n=11) be sorted? Since we are sorting the full array, p=1 and r = 11.

4	7	2	6	1	4	7	3	5	2	6

- What would the initial call to MERGE-SORT look like?
- What would the next call to MERGE-SORT look like?
- What would the one after that look like?

The Merge Procedure

- **Input:** Array A and indices p, q, r such that
  - $p \leq q < r$
  - Subarray A[p..q] is sorted and subarray A[q+1..r] is sorted. Neither subarray is empty
- **Output:** The two subarrays are merged into a single sorted subarray in A[p..r]

```
MERGE(A, p, q, r)
n_1 ← q − p + 1
n_2 ← r − q
create arrays L[1..n_1 + 1] and R[1..n_2 + 1]
for i ← 1 to n_1
   do L[i] ← A[p + i − 1]
for j ← 1 to n_2
   do R[j] ← A[q + j]
L[n_1 + 1] ← ∞
R[n_2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
   do if L[i] ≤ R[j]
      then A[k] ← L[i]
         i ← i + 1
      else A[k] ← R[j]
         j ← j + 1
```

Example

- A call of MERGE(A, 1, 3, 5) where the array is:

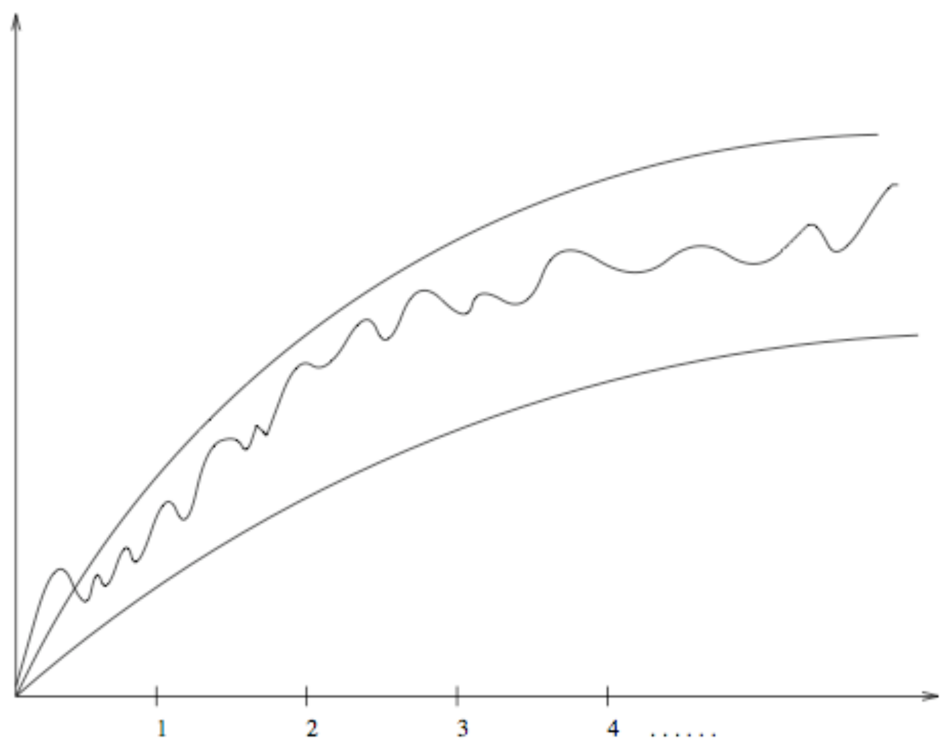
3	5	7	2	6

Analysis

- **Best Case**: Too easy to cheat with best case. We do not rely it on much
- **Average Case**: Usually *very hard* to compute the average running time. Very time consuming.
- **Worst Case**: Fairly easy to analyze. Often close to the average running time. More informative.

Exact Analysis is Hard

- Best, average, and worst case complexity of an algorithm is a numerical function of the size of the instances.



Exact Analysis is Hard

- It is difficult to work with ***exactly*** because it is typically very complicated.
- It is cleaner and easier to talk about *upper and lower bounds* of the function.
- Remember that we ignore constants.
  - This makes sense since running our algorithm on a machine that is twice as fast will affect the running time by a multiplicative constant of 2, we are going to have to ignore constant factors anyway.

Asymptotic Notation

- Asymptotic notation (O, Θ, Ω) are the best that we can practically do to deal with the complexity of functions.

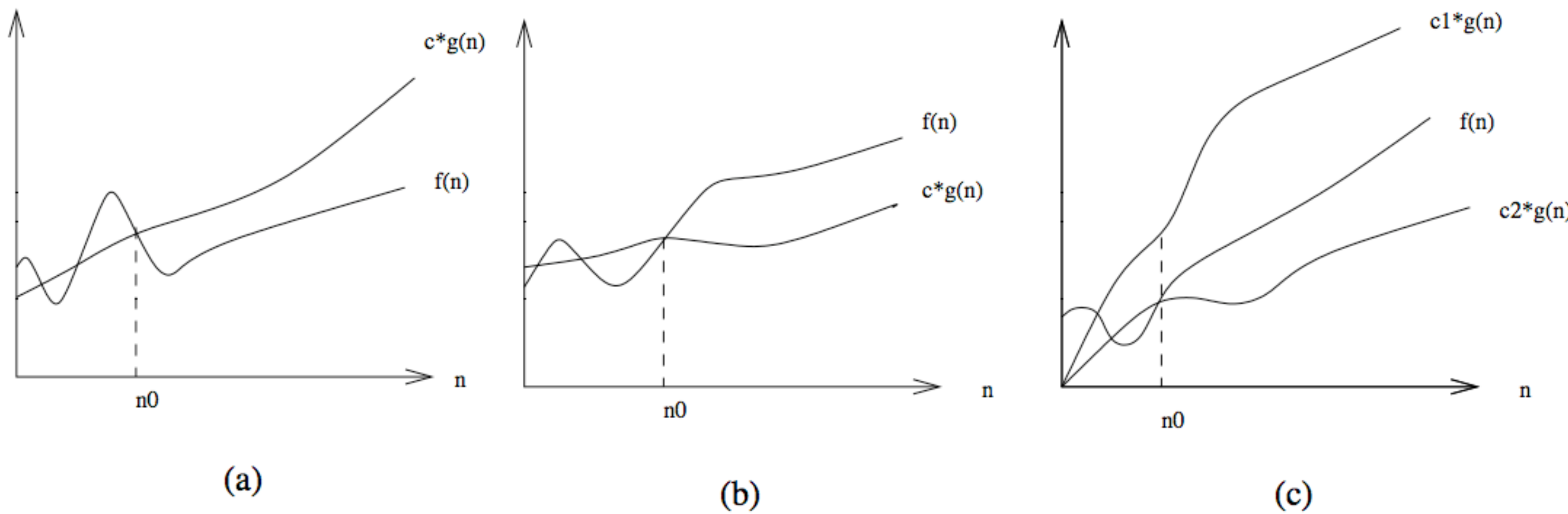
Bounding Functions

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

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

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

Examples of O, Ω, and Θ



Formal Definitions – Big Oh

- $f(n) = \mathrm{O}(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $c.g(n)$.

- Think of the equality (=) as meaning *in the set of functions*.

Formal Definitions – Big Omega

Formal Definitions – Big Theta

Logarithms

- It is important to understand deep in your bones what logarithms are and where they come from.
- A logarithm is simply an inverse exponential function. Saying $b^x = y$ is equivalent to saying that $x = \log_b y$.

Logarithms

- Exponential functions, like the amount owed on a n year mortgage at an interest rate of c % per year, are functions which grow distressingly fast, as anyone who has tried to pay off a mortgage knows.
- Thus inverse exponential functions, ie. logarithms, grow refreshingly slowly.

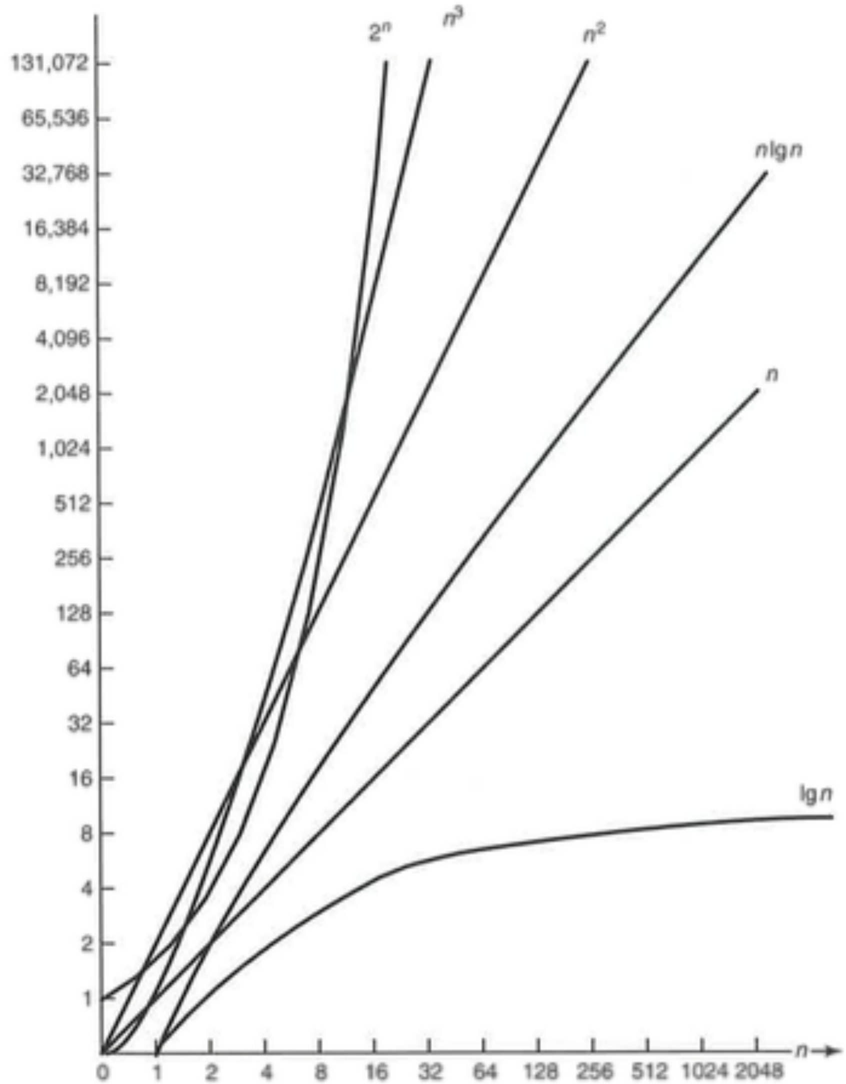
Examples of Logarithmic Functions

- Binary search is an example of an O(lg n) algorithm. After each comparison, we can throw away half the possible number of keys.
- Thus twenty comparisons suffice to find any name in the million-name Manhattan phone book!
- If you have an algorithm which runs in O(lg n) time, take it, because this is blindingly fast even on very large instances.

Asymptotic Dominance in Action

	O(lg n)	O(n)	O(n lg n)	n^2	2^n	n!
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4*1015 yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	$4*10^{13}$ yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		

Growth rate of complexity functions



Readings

- Read chapters 1, 2, 3 from the book