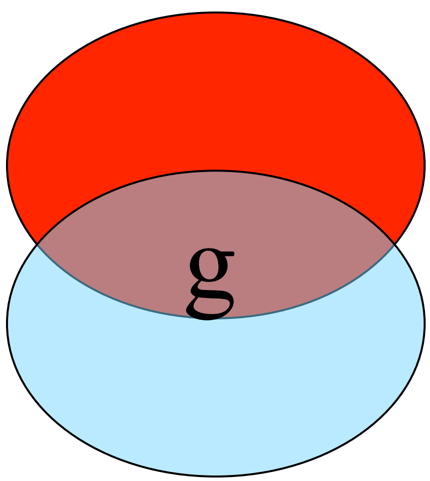
CS 4102, Algorithms: Chapter 2

- Measuring time complexity
  - Order classes: Big-Oh etc.
  - Proving order-class membership
  - Properties of order-classes
- More on optimality (not in text)
  - Improving searching of lists
  - Binary Search: W(n), A(n)
  - Decision Trees for lower-bounds arguments

Classifying functions by their Asymptotic Growth Rates

- asymptotic growth rate, asymptotic order, or order of functions
  - Comparing and classifying functions that ignores *constant factors* and *small inputs*.
- The Sets big oh O(g), big theta Θ(g), big omega Ω(g)



Ω(g): functions that grow **at least as fast** as g

Θ(g): functions that grow **at the same rate** as g

O(g): functions that grow **no faster** than g

The Sets $O(g)$, $\Theta(g)$, $\Omega(g)$

- Let *g* and *f* be a functions from the nonnegative integers into the positive real numbers
- For some real constant $c > 0$ and some nonnegative integer constant N_0

- $O(g)$ is the set of functions f, such that
- $f(n) \leq c\, g(n)$ for all $n \geq N_0$
- $\Omega(g)$ is the set of functions f, such that
- $f(n) \geq c\, g(n)$ for all $n \geq N_0$
- $\Theta(g) = O(g) \cap \Omega(g)$
  - asymptotic order of g
  - $f \in \Theta(g)$ read as "f is asymptotic order g" or "f is order g"

Asymptotic Bounds

- The Sets big oh $O(g)$, big theta $\Theta(g)$, big omega $\Omega(g)$ – remember these meanings:
  - $O(g)$: functions that grow **no faster** than g, or **asymptotic upper bound**
  - $\Omega(g)$: functions that grow **at least as fast** as g, or **asymptotic lower bound**
  - $\Theta(g)$: functions that grow **at the same rate** as g, or **asymptotic tight bound**

Comparing asymptotic growth rates

- Comparing f(n) and g(n) as n approaches infinity,
- IF $$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$
- $< \infty$, including the case in which the limit is 0 then $f \in O(g)$
- > 0, including the case in which the limit is ∞ then $f \in \Omega(g)$
- $= c$ and $0 < c < \infty$ then $f \in \Theta(g)$
- $= 0$ then $f \in o(g)$ read as "little oh of g"
- $= \infty$ then $f \in \omega(g)$ read as "little omega of g"

Properties of $O(g)$, $\Theta(g)$, $\Omega(g)$

- Transitive: If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$
  O is transitive. Also Ω, Θ, o, ω are transitive.
- Reflexive: $f \in \Theta(f)$
- Symmetric: If $f \in \Theta(g)$, then $g \in \Theta(f)$
- Θ defines an equivalence relation on the functions.
  - Each set $\Theta(f)$ is an equivalence class (complexity class).
- $f \in O(g) \Leftrightarrow g \in \Omega(f)$
- $O(f + g) = O(\max(f, g))$
  similar equations hold for Ω and Θ

Classification of functions (1)

- O(1) denotes the set of functions bounded by a *constant* (for large n)
- $f \in \Theta(n)$, f is *linear*
- $f \in \Theta(n^2)$, f is *quadratic*; $f \in \Theta(n^3)$, f is *cubic*
- $\lg n \in o(n^\alpha)$ for any $\alpha > 0$, including fractional powers

$$\sum_{i=1}^{n} i^d \in \Theta(n^{d+1}) \qquad \sum_{i=1}^{n} \log(i) \in \Theta(n \log(n))$$

$$\sum_{i=a}^{b} r^i \in \Theta(r^b) \text{ for } r > 0,\ r \neq 1,\ b \text{ may be some function of } n$$

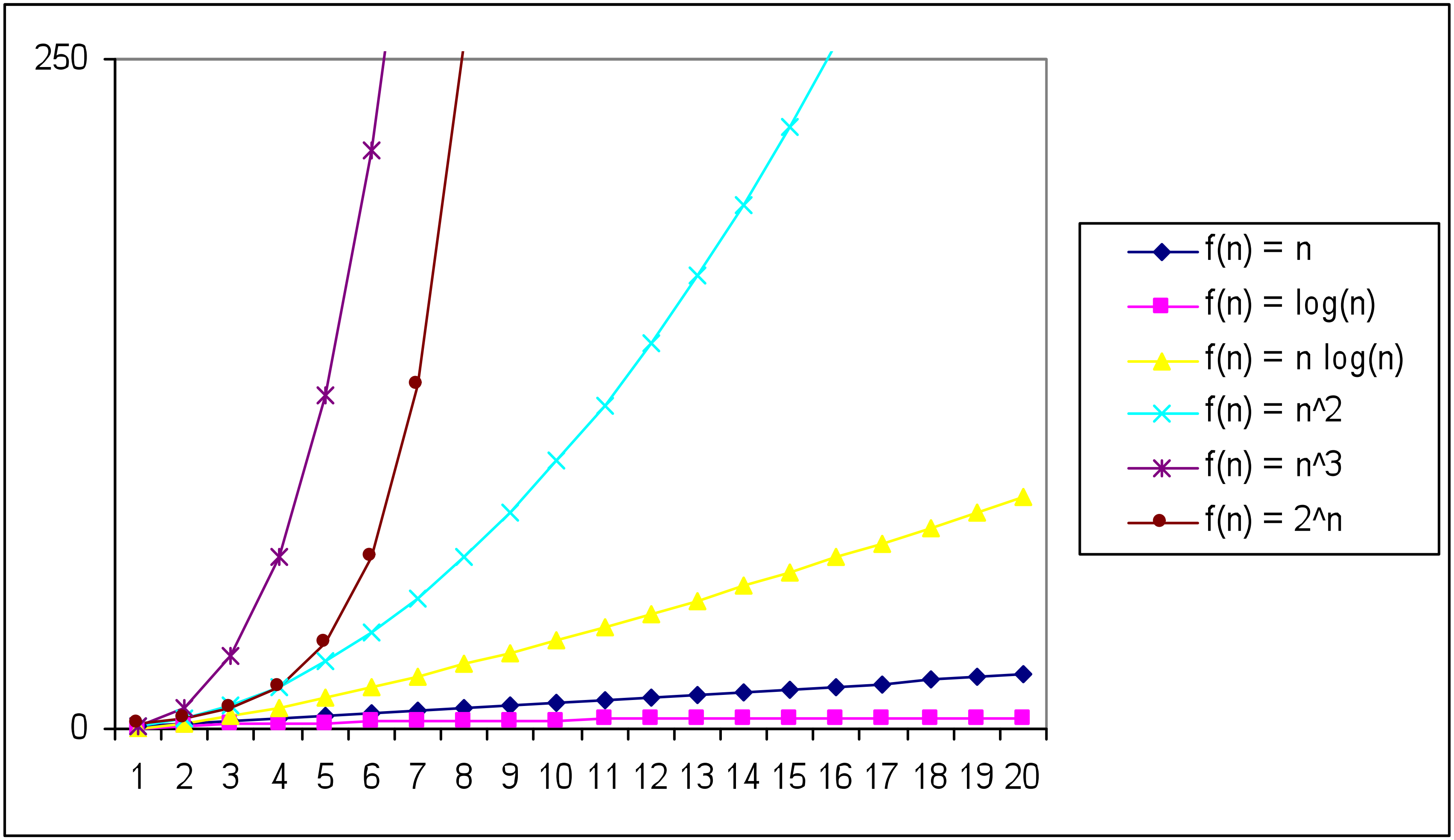
Classification of functions (2)

- $n^k \in o(c^n)$ for any $k > 0$ and any $c > 1$
  - powers of n grow more slowly than any exponential function c^n

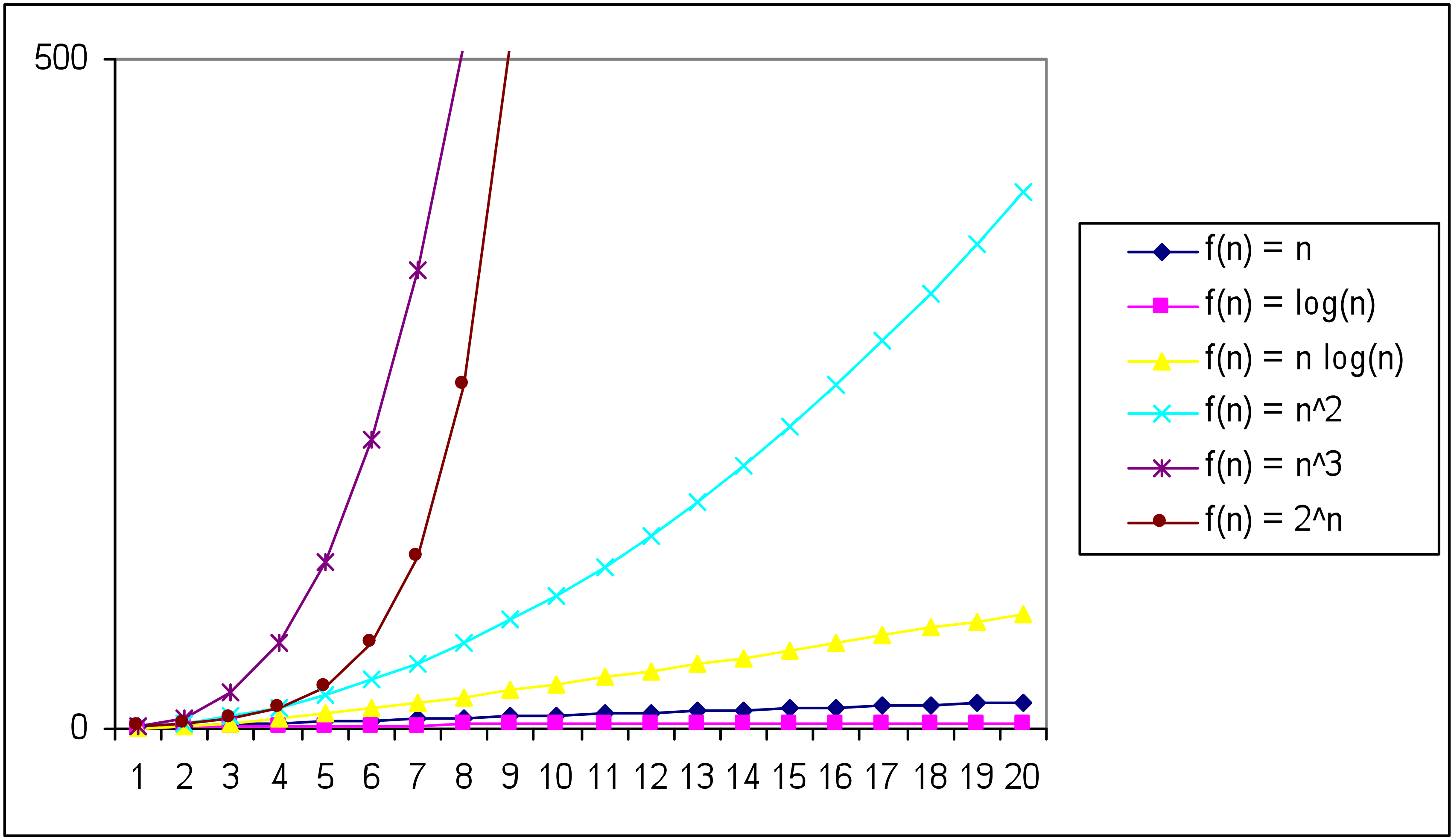
Does Order Class Matter?

- No, not for small inputs
- Yes, for many real problems

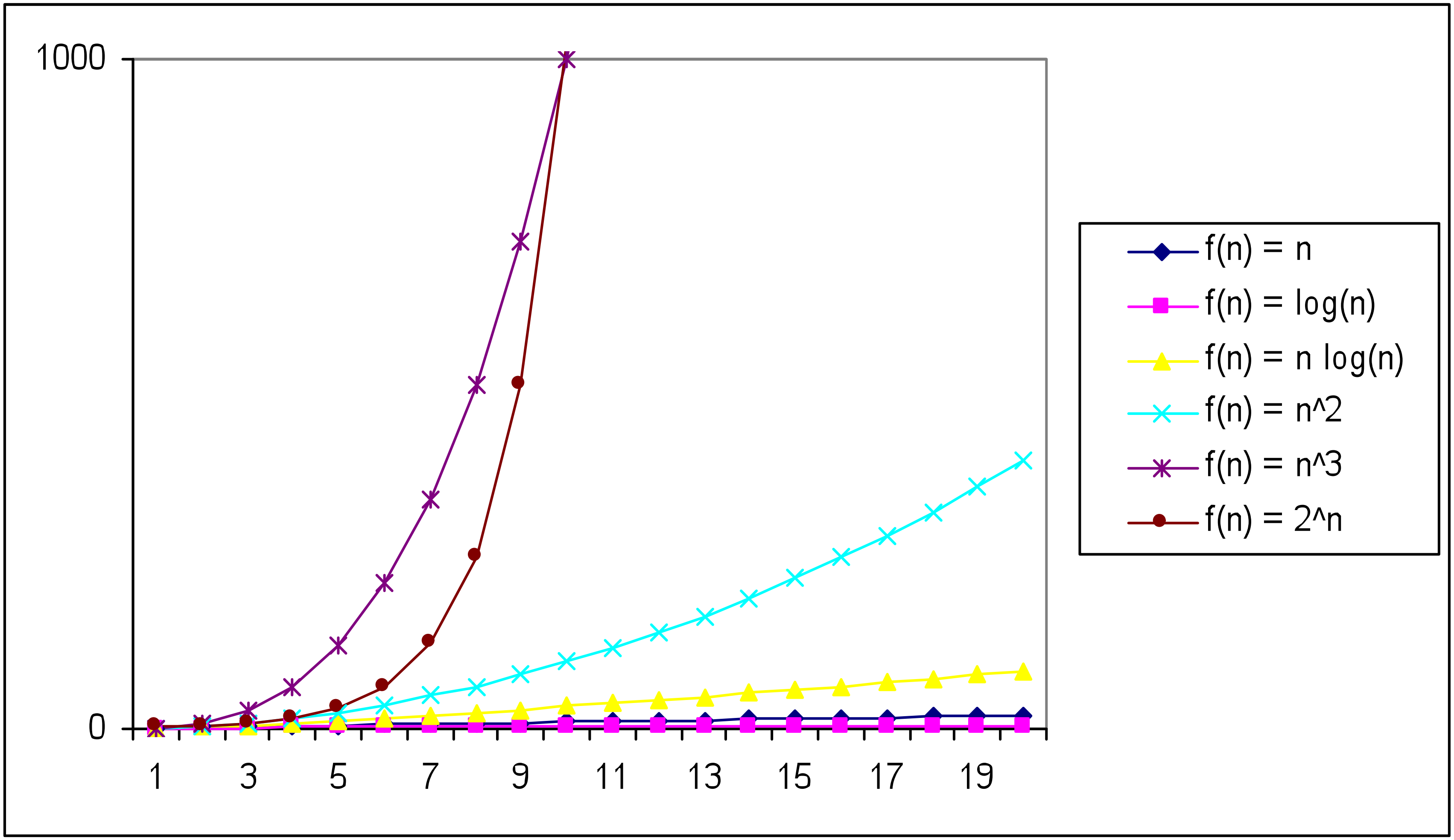
Practical Complexity



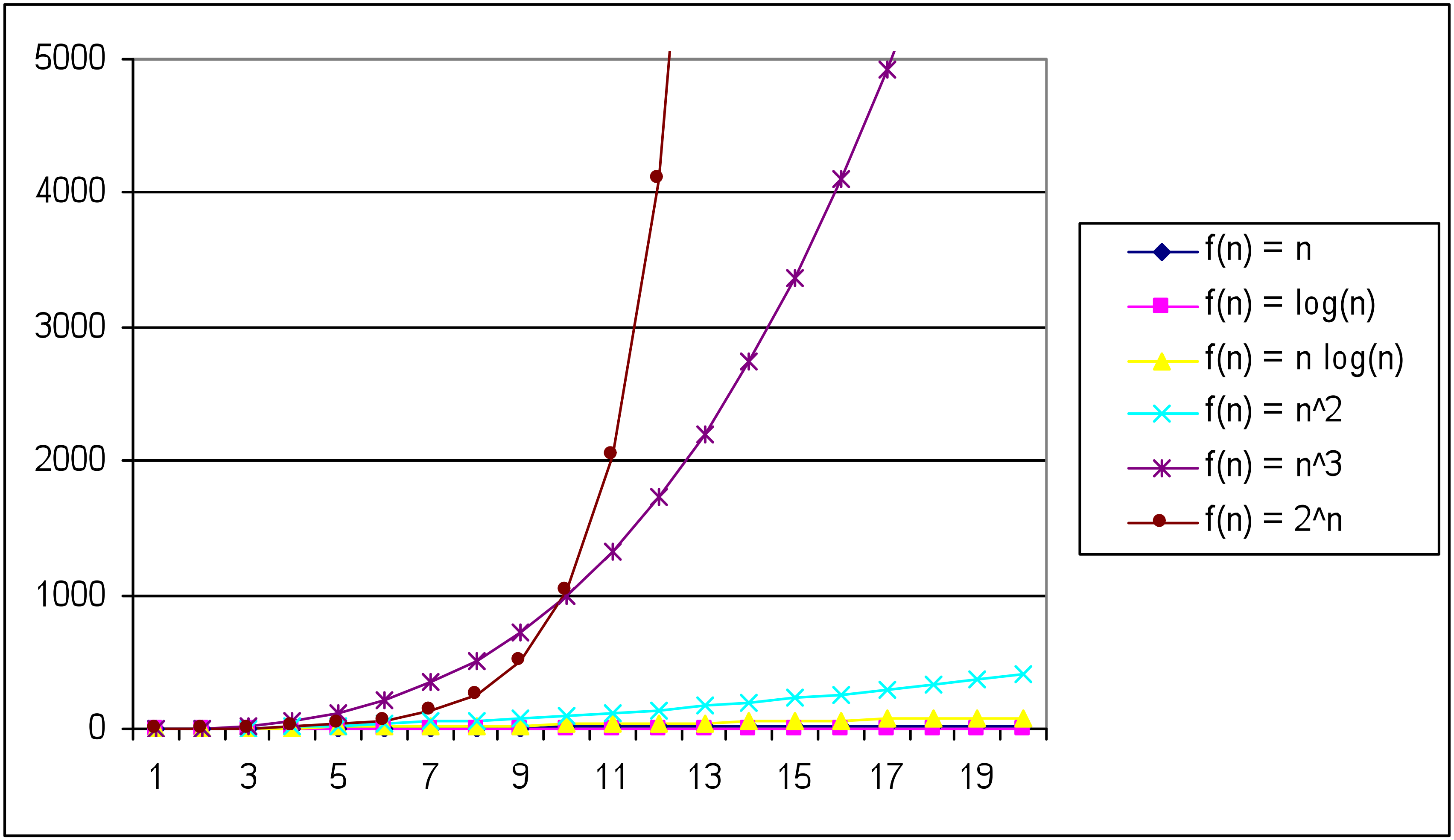
Practical Complexity



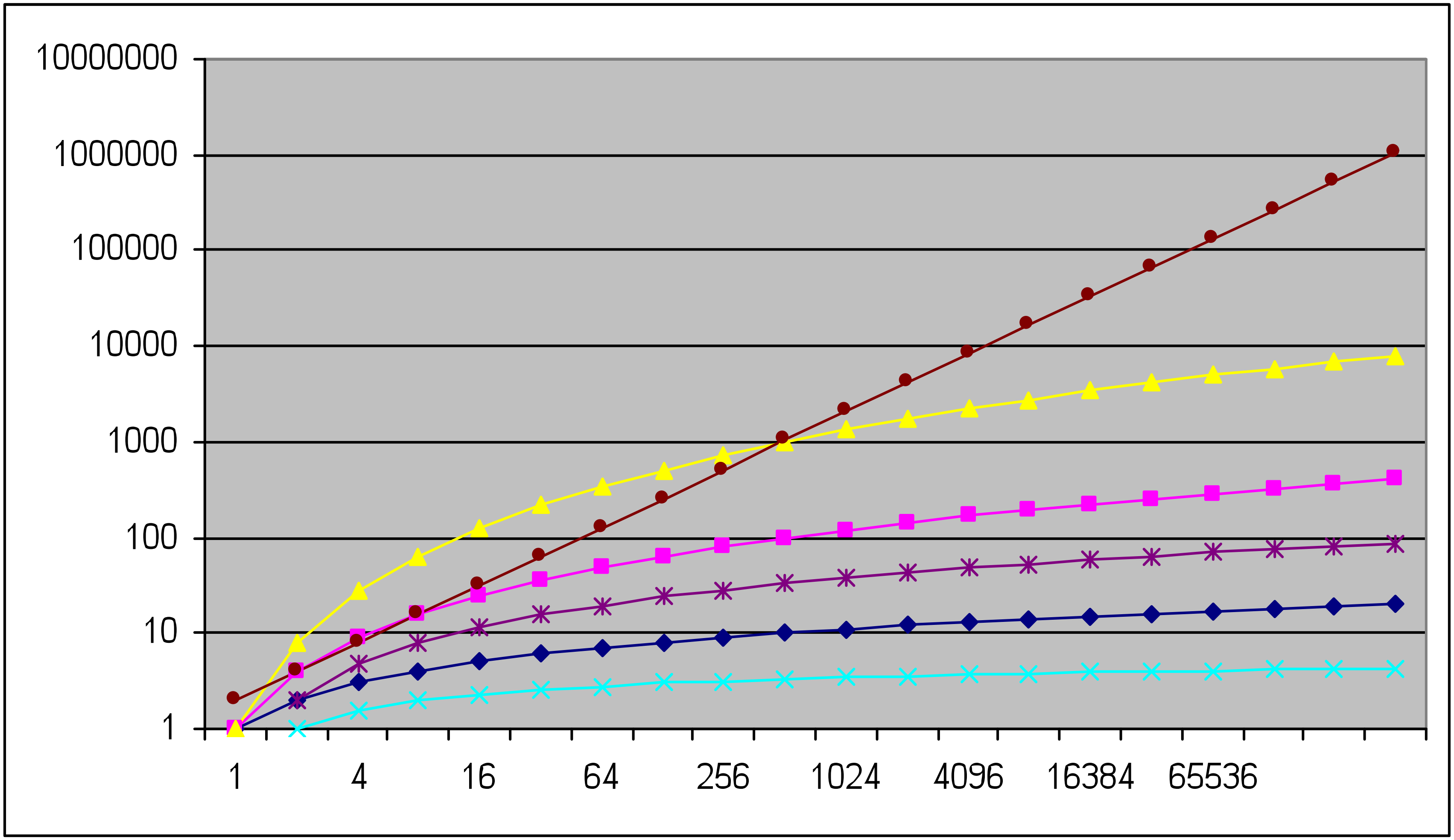
Practical Complexity



Practical Complexity



Practical Complexity



More on Optimality

- Binary Search
- Decision tree arguments for Search Algorithms

Searching Revisited

- Notes about slides vs. code
  - K is variable name in slides (“key”)
    - We use *target* in code
  - E is variable name in slides (“elements”?)
    - We use *list* in code

Searching Revisited

- Problem: array search
  - Given an array E containing n and given a value K, find an index for which K = E[index] or, if K is not in the array, return –1 as the answer.
  - Sometimes we know E is sorted, so we can use that
- Design Trade-off: a more organized data structure with more efficient operations vs. cost of keeping it organized

- If unsorted, standard sequential search (see earlier)
- If sorted, two strategies:
  - Quit when we know we've passed where it should be
  - Binary Search

Sequential Search, Optimality

- Reminder: time complexity for standard sequential search
  - $W(n) = n$
  - $A(n) = q\,[(n+1)/2] + (1-q)\,n$
    - where q is the probability it's in the list

Better Algorithm If "Better" Input

- Modify sequential search:
  As soon as an entry larger than K is encountered, the algorithm can terminate with the answer –1.

- Clearly better. Or is it?
  - In what sense?
  - Same order-class, same worst-case

Modified sequential search

```
def seq_search_mod(list, target):
    ans = -1
    i = 0
    for cur in list:
        if cur < target:    # could be later
            i = i + 1
        elif cur > target:  # not there
            break
        else:               # found it
            ans = i
            break
    return ans
```

Binary Search:

- Strategy
  - compare K first to the entry in the middle of the array
    - eliminates half of the entry with one comparison
  - apply the same strategy recursively
    - but note that this can be implemented using a loop
- Algorithm: Binary Search
  - Input: E, first, last, and K, all integers, where E is an ordered array in the range first, …, last, and K is the key sought.
  - Output: index such that E[index] = K if K is in E within the range first, …, last, and index = -1 if K is not in this range of E

Binary Search

```
def do_binsearch_rec(list, target, first, last):
    if last < first:
        ans = -1
    else:
        mid = (first + last)/2
        if target == list[mid]:
            ans = mid
        elif target < list[mid]:
            ans = do_binsearch_rec(list, target, first, mid-1)
        else:
            ans = do_binsearch_rec(list, target, mid+1, last)
    return ans
```

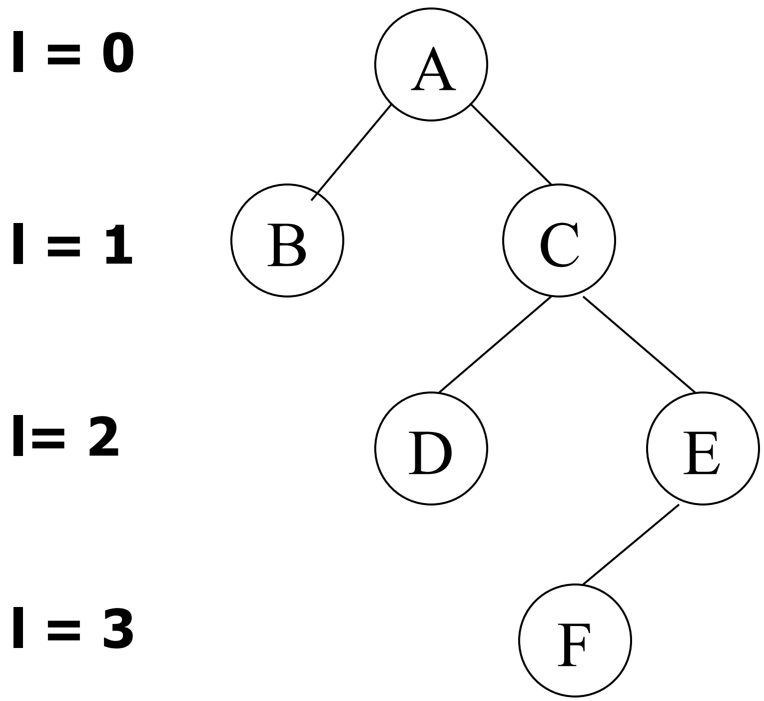
Recursive vs. non-recursive?

- Can you write this code as a non-recursive algorithm?

An Aside on Binary Trees….

- Review section 2.6, Trees
- Definition of level of a node in a tree
  - Root: level 0
  - Other nodes: one more than level of parent
  - In other words, level is the number of “levels” above a given node, or length of path back to the root
- Definition of height
  - Height of a tree: maximum level of a tree’s leaves

Level and Height Illustrated



- Level applies to all nodes at that "level"
- Number of "levels" is one more than tree's level
- Height of tree is 3

h =
h =
h =
h =

Properties of Binary Trees

- *Lemma 1*
  At level d in a binary tree, there are at most 2^d nodes
- *Lemma 2*
  A binary tree with height h has at most $2^{h+1}-1$ nodes
  - Examples: h=0, 1 node. h=1, 3 nodes. h=2, 7 nodes.
- *Lemma 3*
  A binary tree with n nodes has height at least: Ceiling(lg(n+1)) - 1
  - Examples: 7 nodes? Shortest tree has h=2 (3 levels)
    8 nodes? Shortest tree has h=3 (4 levels)

Worst-Case Analysis of Binary Search

- Assumptions:
  - Let the problem size be n = last – first + 1; n>0
  - Basic operation is a comparison of K to an array entry
  - Assume one comparison is done with the three-way branch
- Analysis
  - First comparison, assume K != E[mid], divides the array into two sections, each section has at most Floor[n/2] entries.
  - Estimate that the size of the range is divided by 2 with each recursive call.
  - How many times can we divide n by 2 without getting a result less than 1 (i.e. $n/(2^d) >= 1$) ?
  - d <= lg(n), therefore we do Floor[lg(n)] comparison following recursive calls, and one before that.
- $W(n) = \text{Floor}[\lg(n)] + 1 = \text{Ceiling}[\lg(n + 1)] \in \Theta(\log n)$

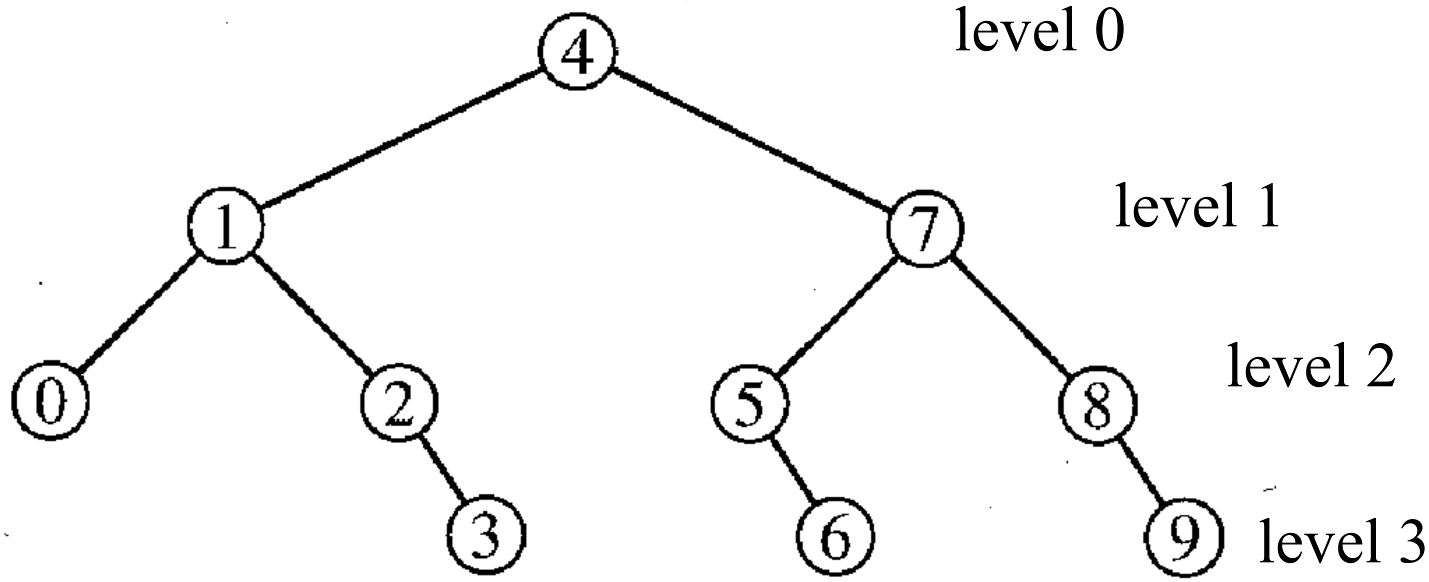
Average Case Analysis of Binary Search

- Analysis is, well, ugly (can I say that?)
- But, consider the decision tree and note:
  - For a complete binary tree, more than half the nodes are at the bottom level. The worst case!
  - Also, if the key is not there, we don't know until we "reach" the bottom level of the tree.
  - Therefore, you can imagine that the average case is very close to the worst-case.
  - But, that's OK, cause $W(n) = \Theta(\lg n)$ is pretty darn good!
- $A(n) \approx \lg(n+1) - q$, where q is probability of successful search
- Recall $W(n) = \text{Ceiling}[\lg(n + 1)]$

Optimality of Binary Search

- So far we improve from $\theta(n)$ algorithm to $\theta(\log n)$
  - Can more improvements be possible?
- For optimality and such questions, we must make a proof for a *class of algorithm*
  - Here, the class is: the set of search algorithms for sequences where a comparison is the basic operation
- Such algorithms can be modeled with a **decision tree**:
  - Root contains index of the first item compared to the target
  - If equal, we'd stop
  - If target less than that item, next comparison is the left-child
  - If target greater than item, next comparison is the right-child
  - Etc.

Example of Decision Tree



- Height of tree is 3 (max level of a leaf)
- Number of levels? height+1
- W(n) number of comparisons? number of nodes on path from root to leaf. I.e., num. levels or height+1

What Decision Trees Tell Us about Search

- Shows a trace of the order and number of comparisons made
  - Path from root to “deepest” node is W(n)
  - Average path length is A(n)
- If we find properties for decisions trees in general, these are true of any algorithm in this class

Decision Trees, Search Algorithms

- How "short" can a decision tree be?
- Let N be the number of nodes in a decision tree
  - Different than n (number of items in list)
- By Lemma 3:
  - height >= Ceiling(lg(N+1)) - 1
- From previous slide, number of nodes on path is height+1
- So, max number of nodes >= Ceiling(lg(N+1))
  - Max number of nodes is W(n)
- W(n) >= Ceiling(lg(N+1))
  - But this is N not n

Decisions Trees, Search Algorithms (2)

- We claim N >= n if an algorithm A works correctly in all cases
  - For argument, see next slide
- If N >= n then
  Ceiling(lg(N+1)) >= Ceiling(lg(n+1))
- Therefore…
  - Any search algorithm that uses comparisons can be represented by a decision tree
  - W(n) >= Ceiling(lg(N+1)) >= Ceiling(lg(n+1))

Prove by contradiction that N >= n

- Suppose there is no node labeled i for some i in the range from 0 through n-1
  - Make up two input arrays E1 and E2 such that
  - E1[i] = K but E2[i] = K' > K
  - For j < i, make E1[j] = E2[j] using some key values less than K
  - For j > i, make E1[j] = E2[j] using some key values greater than K' in sorted order
  - Since no node in the decision tree is labeled i, the algorithm A never compares K to E1[i] or E2[i],
but it gives same output for both
  - Such algorithm A gives wrong output for at least one of the array and it is not a correct algorithm
- Conclude that the decision has at least n nodes

Binary Search is Optimal

- W(n) >= Ceiling(lg(n+1)) for any seach algorithm using key comparisons

- Binary search has this W(n)
  - No algorithm can have a lower W(n)
  - It's optimal