

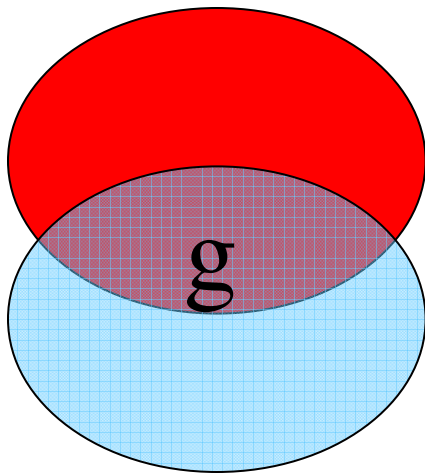
CS 216, Fall 2005: Chapter 2

- These slides were used in the lecture on Chapter 2 during Fall 2005.
 - Consult these with notes taken in class.

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 $\Theta(g)$, big omega $\Omega(g)$

$\Omega(g)$: functions that grow **at least as fast** as g



$\Theta(g)$: 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 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 $N \geq n_0$
- $\Theta(g) = O(g) \cap \Omega(g)$
 - $\Theta(g)$ is the asymptotic order of g or the order of g
 - $f \in \Theta(g)$ read as

“ f is of the same order as g ” or “ f is $\Theta(g)$ ”

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**

Another Way to Define Order Classes

- Comparing $f(n)$ and $g(n)$ as n approaches infinity,
- IF
$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$
- $< \infty$, 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"

Some 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

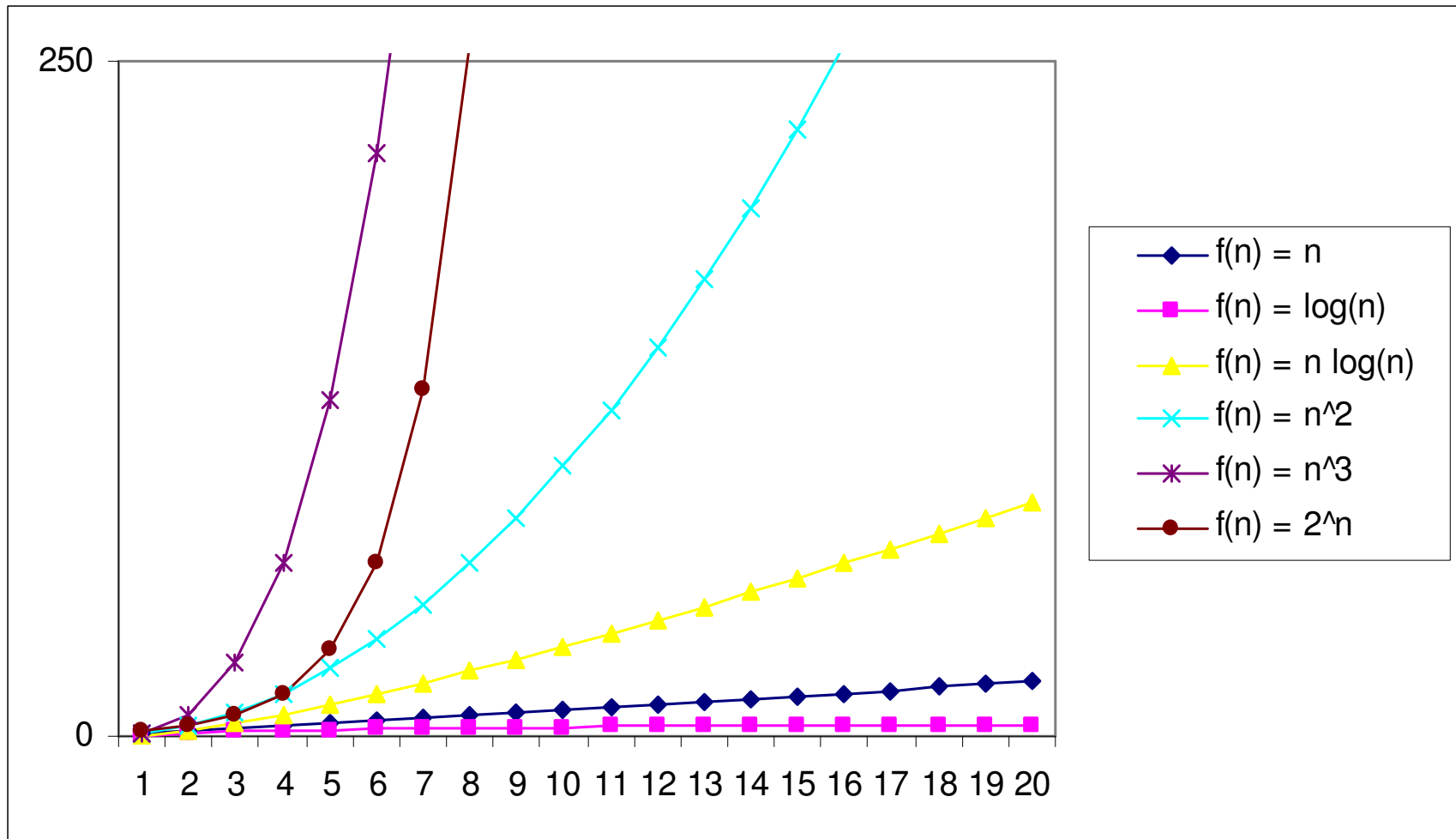
Classification of functions (1)

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

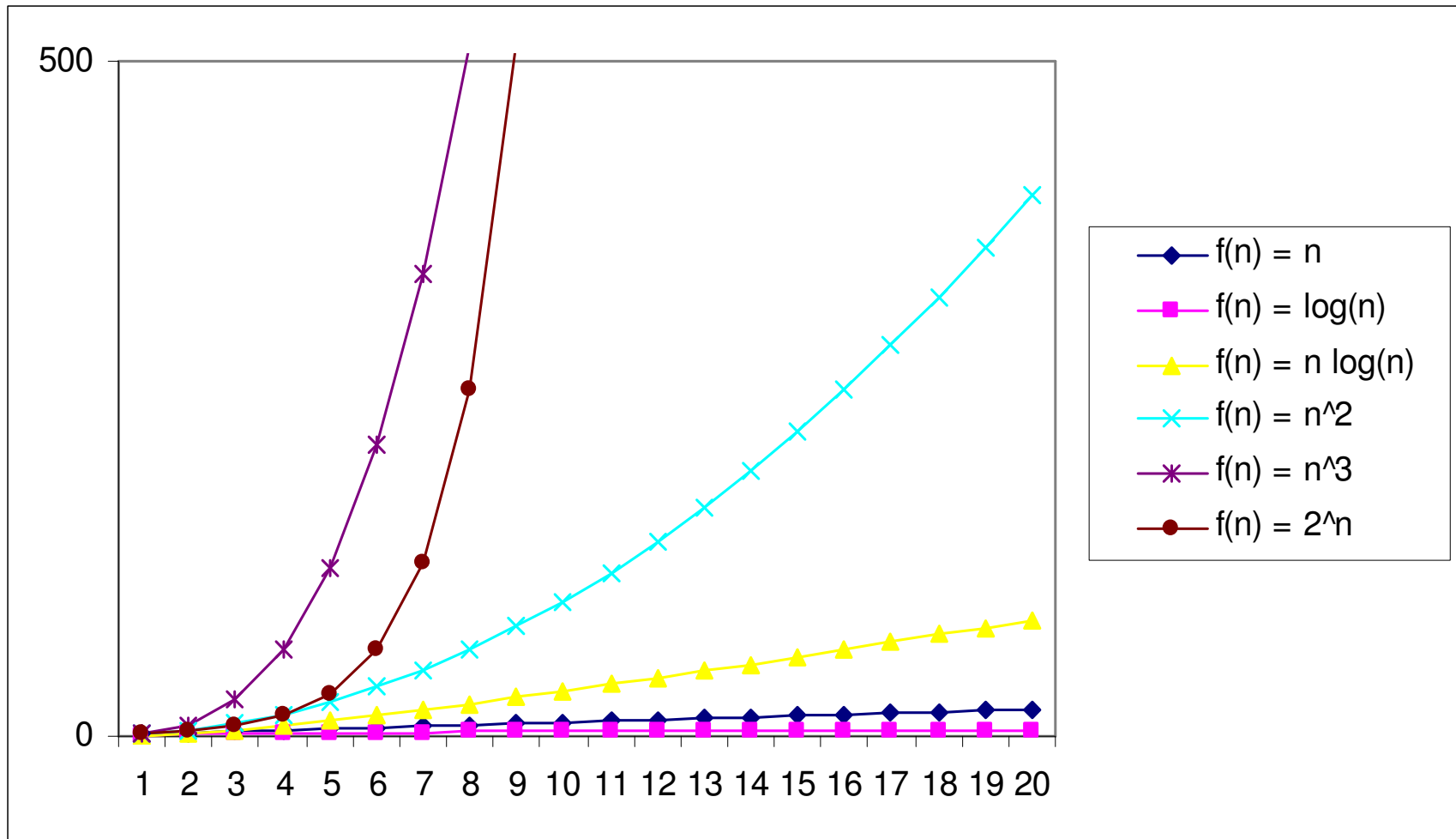
Does Order Class Matter?

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

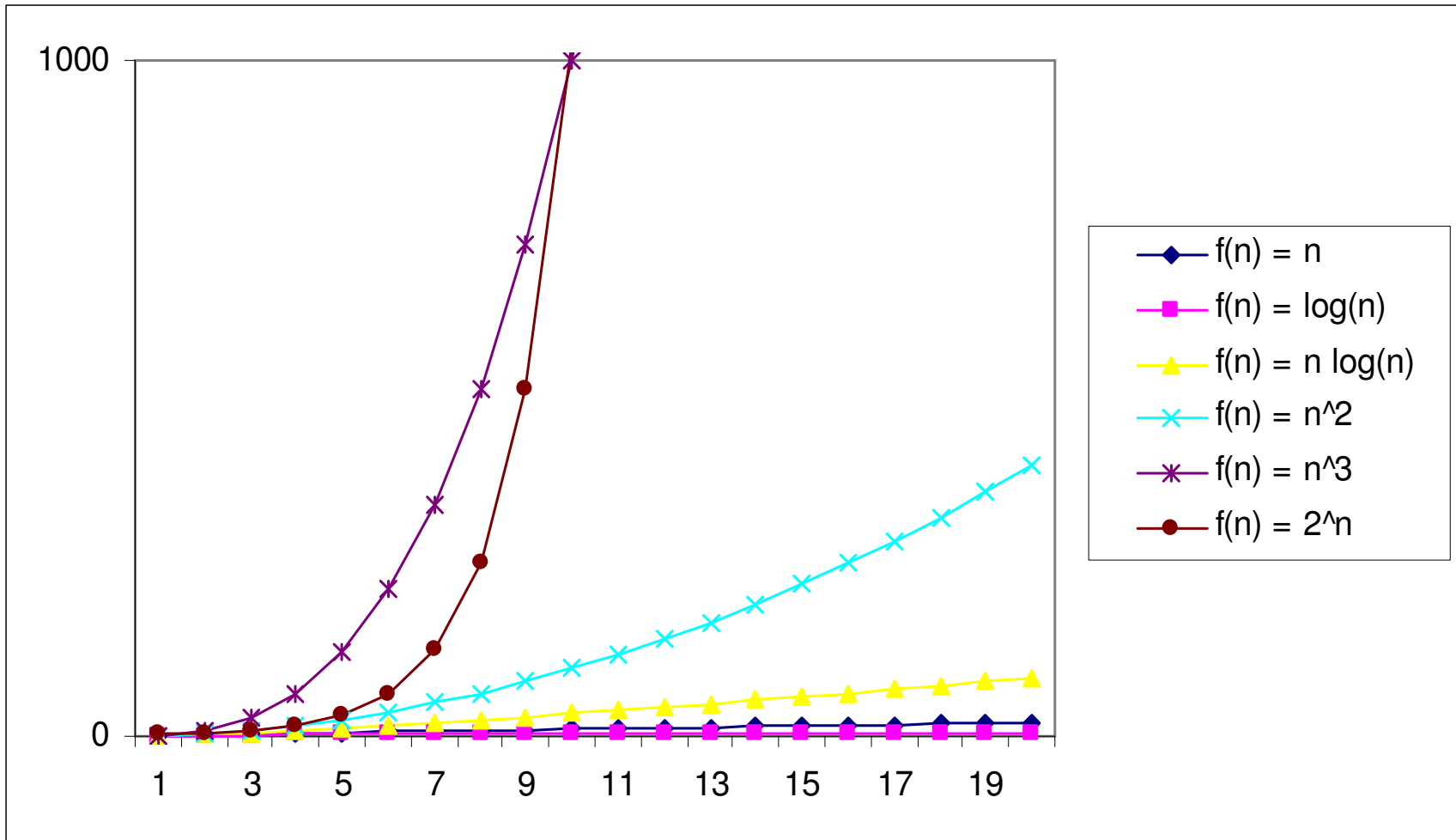
Practical Complexity



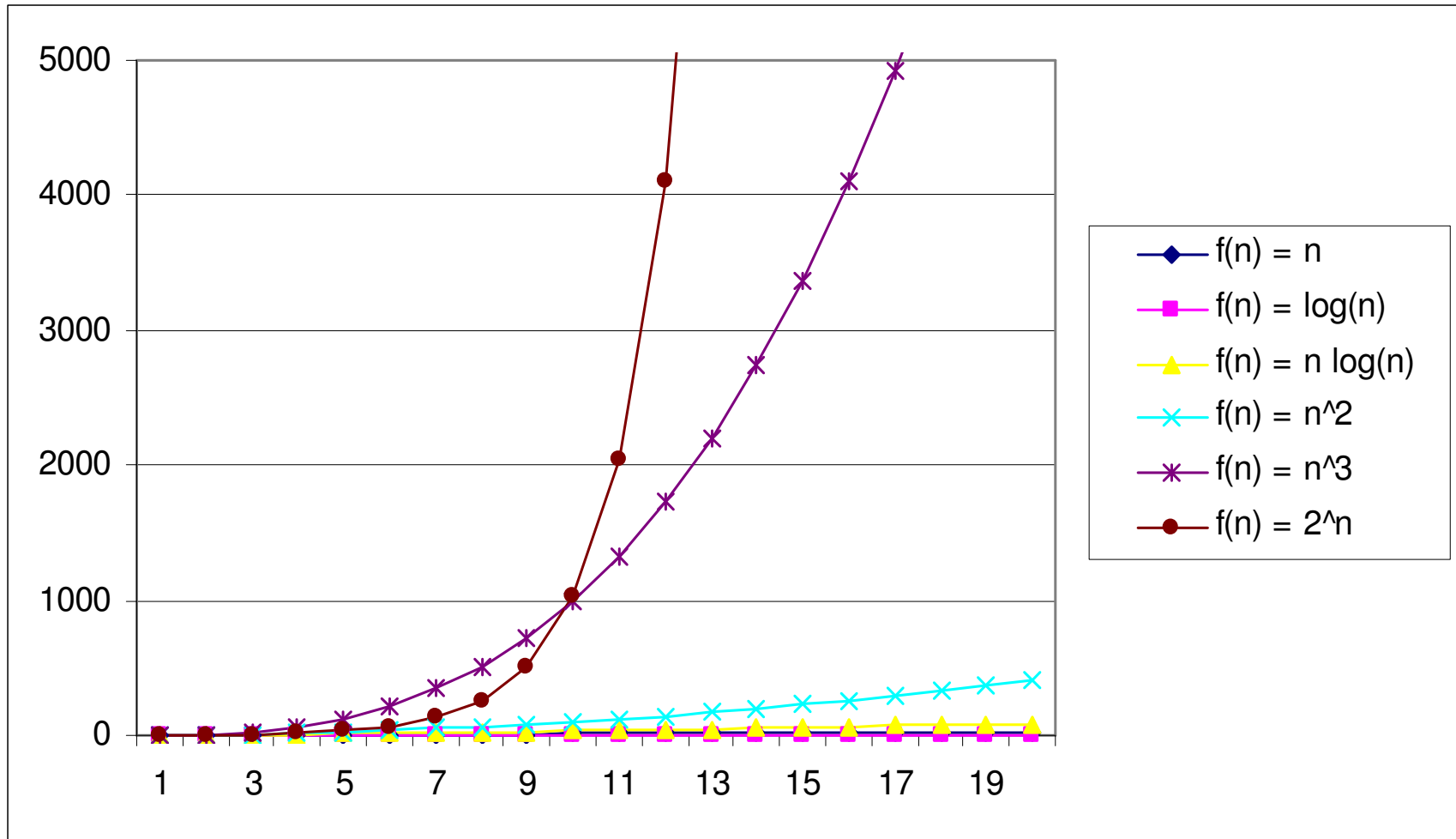
Practical Complexity



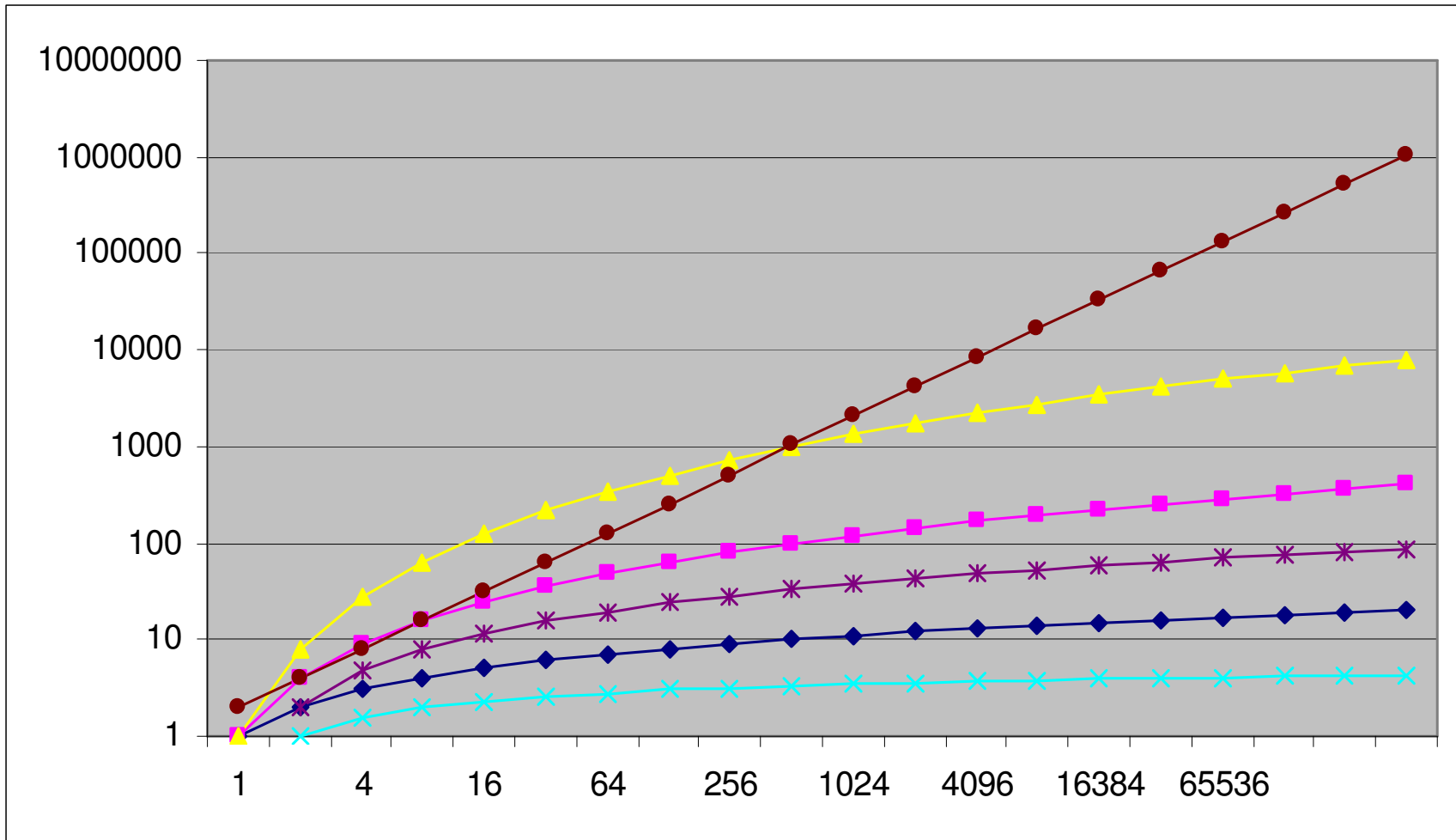
Practical Complexity



Practical Complexity



Practical Complexity



Example Done on 9/27/2005

Another Example: Fibonacci Numbers

- Recursive mathematical definition
 - Fibonacci numbers:
 $F(0) = F(1) = 1$
 $F(n) = F(n-1) + F(n-2)$ for $n > 1$
 - Note base case
- How to implement? Can you name two different ways?
 - Loop. Complexity: $O(n)$
 - Recursively. Complexity: ?

Implement Fibonacci numbers

- It's beautiful code, no?

```
long fib(int n) {  
    assert(n >= 0);  
    if ( n == 0 ) return 1;  
    if ( n == 1 ) return 1;  
    return fib(n-1) + fib(n-2);  
}
```

- Is there a problem here? (Yes, inefficient.)
 - Run and time it.
 - Trace it out
 - Show what recursive calls are made for smaller inputs

Recursion: Good or Evil?

- It depends...
- Sometimes recursion is an efficient design strategy, sometimes not
 - Important! we can define recursively and implement non-recursively
- Note that many recursive algorithms can be re-written non-recursively
 - Use an explicit stack
 - Remove tail-recursion (compilers often do this for you)
- For Fibonacci, the answer is: Bad Idea!

Time for Recursive Fibonacci

