Algorithms

Lec#1

Fall2014

The Course

- Course Goal: a rigorous introduction to the design and analysis of algorithms
  - Not a lab or programming course
  - Not a math course, either
- Textbook: *Introduction to Algorithms*, Cormen, Leiserson, Rivest, Stein
  - An excellent reference you should own

The Course

- Grading policy:
  - Homework & Quizzes: 15%
  - Exam 1: Sep,25th 3:45-5:00 pm 25%
  - Exam 2: Oct 30th 3:45-5:00 pm 25%
  - Final: Dec 12th 7:45-10:00 am 35%

The Course

- Format
  - Two lectures/week
  - Homework most weeks
    - Problem sets
    - Maybe occasional programming assignments
  - Two tests + final exam

Algorithms

Algorithm: give a language for talking about program behavior.

- a set of step by step instructions a program follows to do certain task.

Example 1

How to get to work in the morning

- Different ways with same start and end

Algorithm 1: walking

1. Walk out the front door and lock it.
2. Walk 3 miles.
3. Enter the department building.

Example 2

Algorithm 2: Bicycle

1. Walk out front door and lock it
2. Unlock bicycle , put on helmet.
3. Ride bicycle for 3 miles.
4. Lock up bicycle, take off helmet.
5. Enter department building.

Example 3

Algorithm 3: bus

1. Walk out front door and lock it.
2. Walk half a mile to the bus stop.
3. Ride the bus.
4. Walk to the office.
5. Enter office building.

Example 4

Algorithm 4: Taxi

1. Call taxi company.
2. Walk out front door and lock it.
3. Ride the taxi for miles.
4. Enter the department building.

Compare Algorithms

Walking	Bicycle	Bus	Taxi
free	cheap	cheap	expensive

Compare Algorithms…

Walking	**Bicycle**	**Bus**	**Taxi**
slow	**Medium**	**Medium**	**fast**

Analysis of Algorithms

Analysis of Algorithms: is the theoretical study of computer program performance and resource usage.

- Study how to make things fast.
- In programming …What is more important than performance?
    1. Correctness
    2. Simplicity
    3. Maintainability
    4. Robustness of the software
    5. Security…etc.

Asymptotic Performance

- In this course, we care most about *asymptotic performance*
    - How does the algorithm behave as the problem size gets very large?
        - Running time
        - Memory/storage requirements
        - Bandwidth/power requirements/logic gates/etc.

Running Time

- Number of primitive steps that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
  - We can be more exact if need be
- Worst case vs. average case

  (best case is ….)

Insertion Sort

Statement	Effort
`InsertionSort(A, n) {`	
`  for i = 2 to n {`	$c_1 n$
`      key = A[i]`	$c_2(n-1)$
`      j = i - 1;`	$c_3(n-1)$
`      while (j > 0) and (A[j] > key) {`	$c_4 T$
`            A[j+1] = A[j]`	$c_5(T-(n-1))$
`            j = j - 1`	$c_6(T-(n-1))$
`      }`	0
`      A[j+1] = key`	$c_7(n-1)$
`  }`	0
`}`	

$T = t_2 + t_3 + \ldots + t_n$ where t_i is number of while expression evaluations for the i^{th} for loop iteration

Analyzing Insertion Sort

- $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 T + c_5(T - (n-1)) + c_6(T - (n-1)) + c_7(n-1)$

  $= c_8 T + c_9 n + c_{10}$
- What can T be?
  - Best case -- inner loop body never executed
    - $t_i = 1 \Rightarrow T(n)$ is a linear function
  - Worst case -- inner loop body executed for all previous elements
    - $t_i = i \Rightarrow T(n)$ is a quadratic function
  - Average case
    - ???

Analysis

- Simplifications
    - Ignore actual and abstract statement costs
    - *Order of growth* is the interesting measure:
        - Highest-order term is what counts
            - Remember, we are doing asymptotic analysis
            - As the input size grows larger it is the high order term that dominates

Upper Bound Notation

- We say InsertionSort's run time is *$O(n^2)$*
  - Properly we should say run time is *in* $O(n^2)$
  - Read O as "Big-O" (you'll also hear it as "order")
- In general a function
  - f(n) is O(g(n)) if there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
- Formally
  - $O(g(n)) = \{ f(n)\text{: } \exists$ positive constants c and n_0 such that $f(n) \leq c \cdot g(n)\ \forall\ n \geq n_0$

Insertion Sort Is $O(n^2)$

- Proof
  - Suppose runtime is $an^2 + bn + c$
    - If any of a, b, and c are less than 0 replace the constant with its absolute value
  - $an^2 + bn + c \quad \leq (a + b + c)n^2 + (a + b + c)n + (a + b + c)$
  - $\qquad\qquad\qquad \leq 3(a + b + c)n^2$ for $n \geq 1$
  - Let $c' = 3(a + b + c)$ and let $n_0 = 1$
- Question
  - Is InsertionSort $O(n^3)$?
  - Is InsertionSort $O(n)$?

Big O Fact

- A polynomial of degree k is $O(n^k)$
- Proof:
  - Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$
    - Let $a_i = | b_i |$
  - $f(n) \leq a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

$$\leq \quad n^k \sum a_i \frac{n^i}{n^k} \quad \leq \quad n^k \sum a_i \quad \leq \quad cn^k$$

Lower Bound Notation

- We say InsertionSort's run time is $\Omega(n)$
- In general a function
  - f(n) is $\Omega(g(n))$ if $\exists$ positive constants *c* and n_0 such that $0 \leq c \cdot g(n) \leq f(n) \quad \forall\ n \geq n_0$
- Proof:
  - Suppose run time is an + b
    - Assume a and b are positive (what if b is negative?)
  - $an \leq an + b$

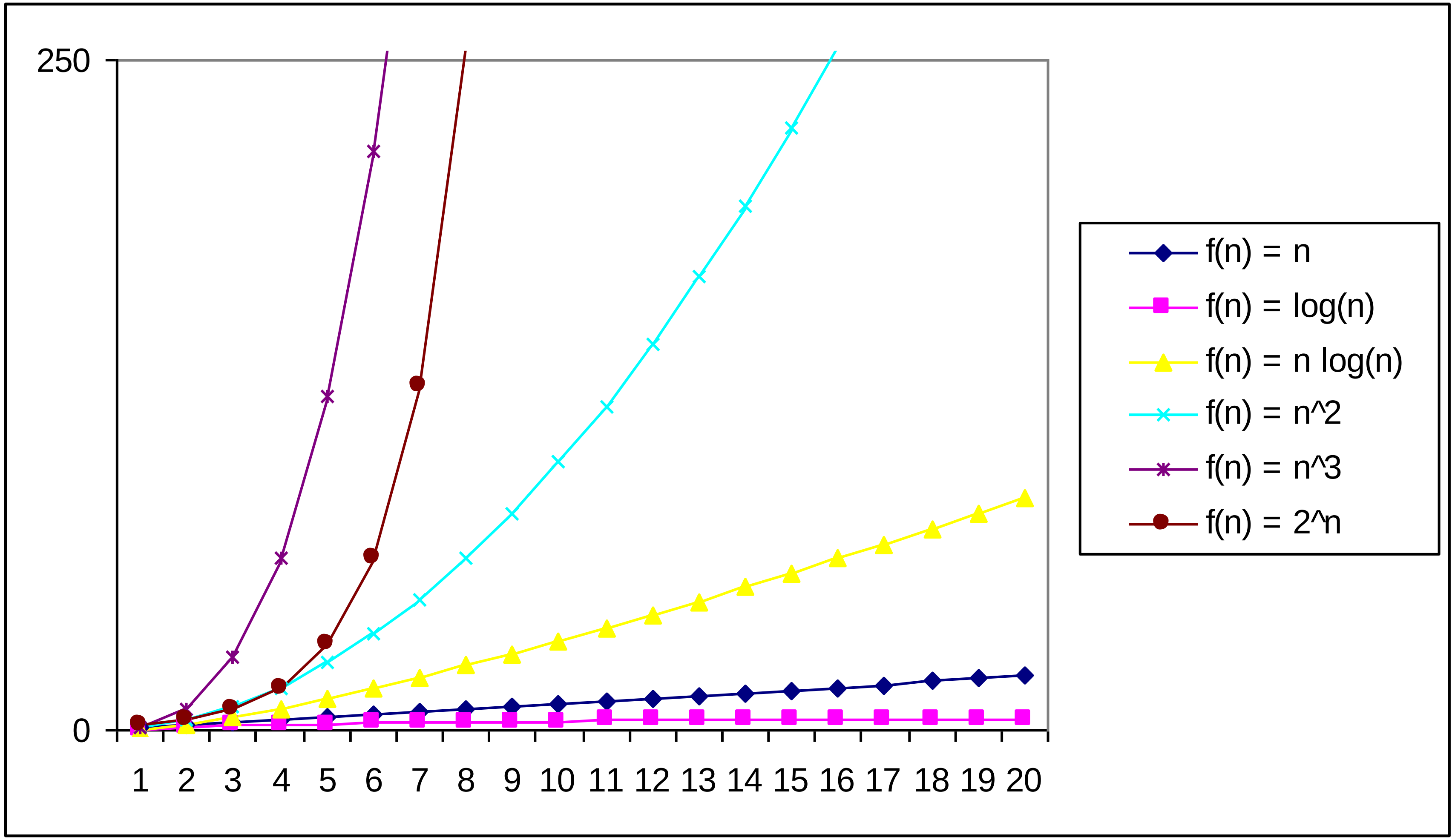
Asymptotic Tight Bound

- A function f(n) is Θ(g(n)) if $\exists$ positive constants c_1, c_2, and n_0 such that

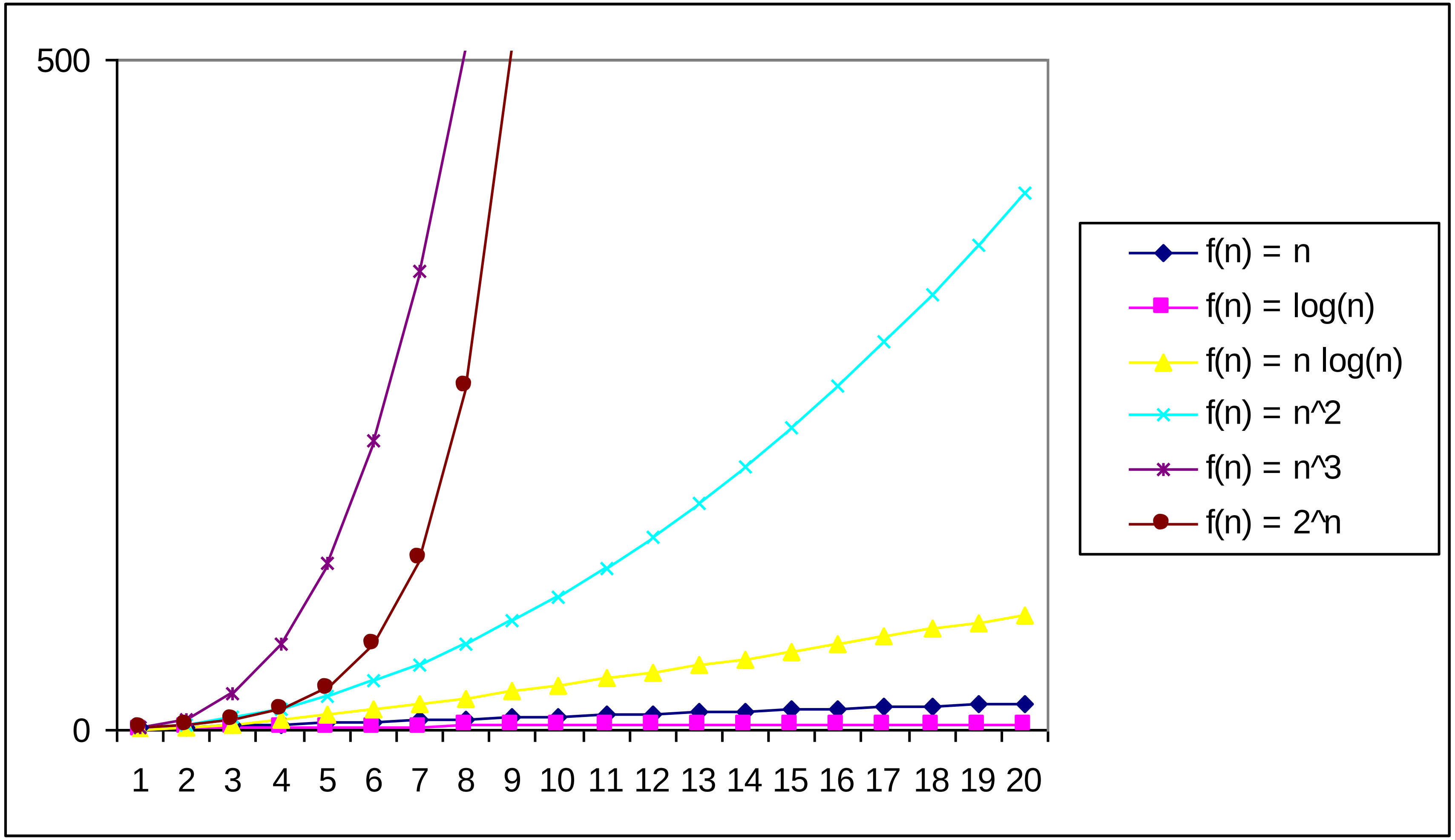
$$c_1\, g(n) \leq f(n) \leq c_2\, g(n)\ \forall\ n \geq n_0$$

- Theorem
  - f(n) is Θ(g(n)) iff f(n) is both O(g(n)) and Ω(g(n))
  - Proof: someday

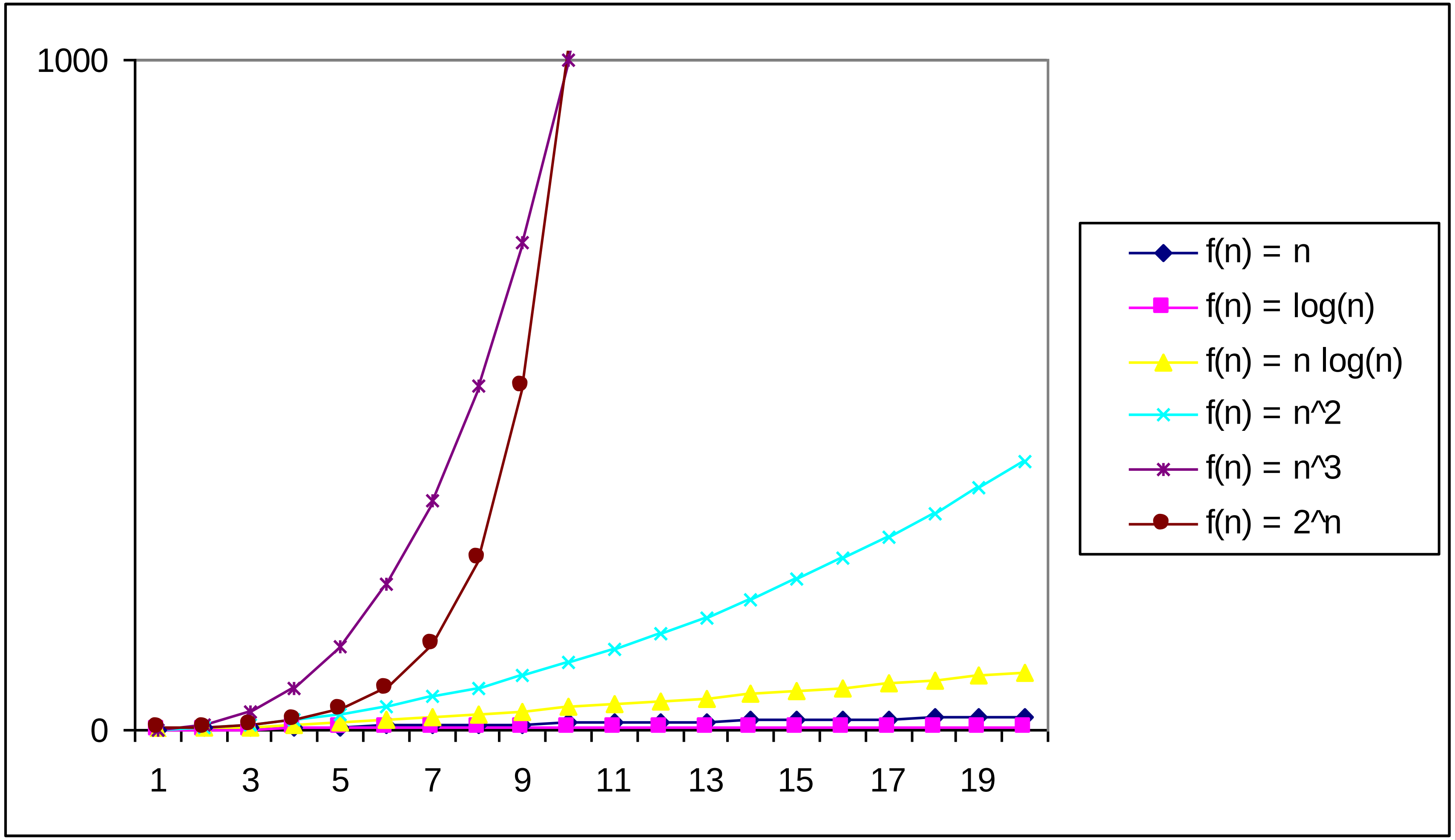
Practical Complexity



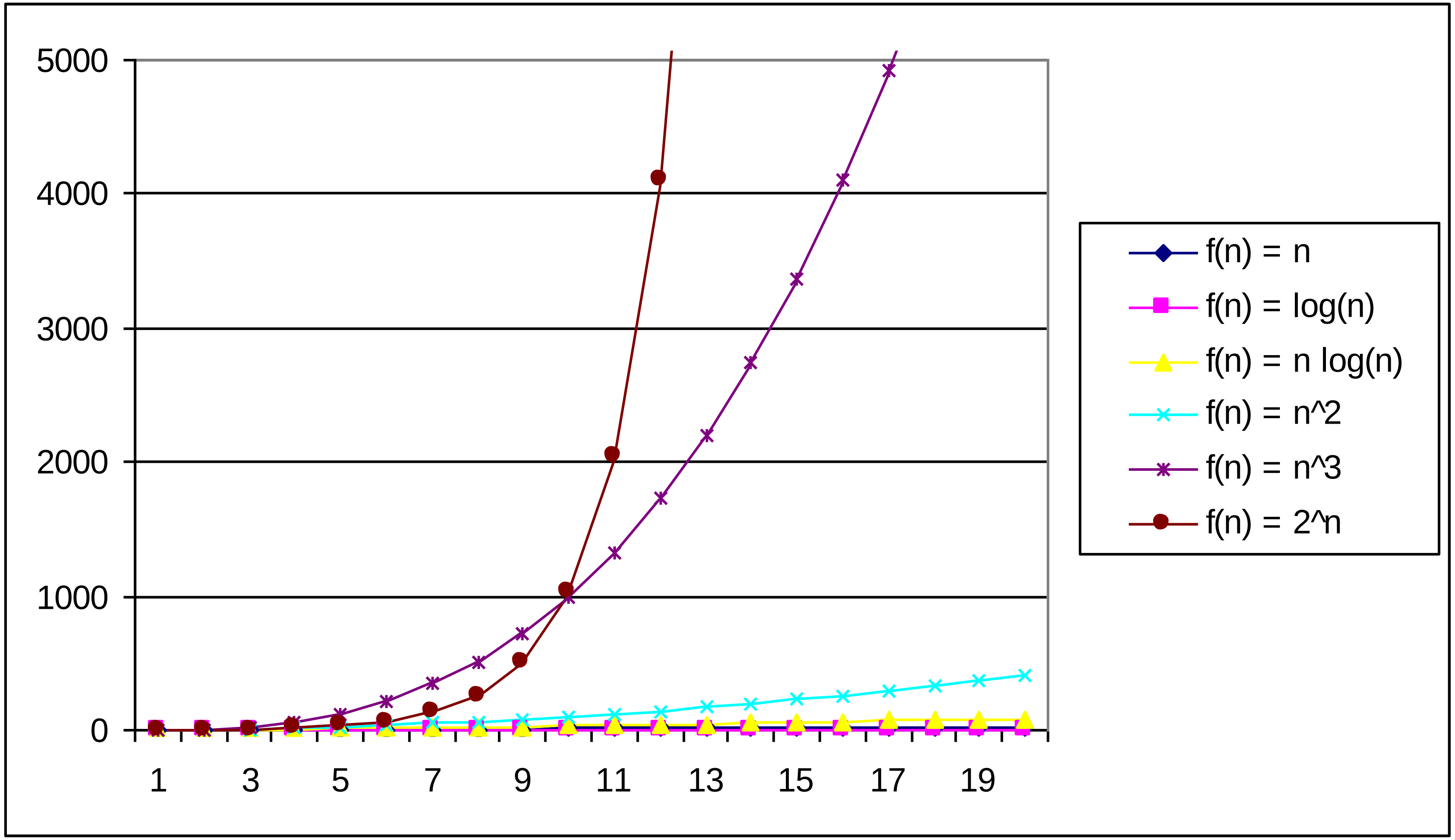
Practical Complexity



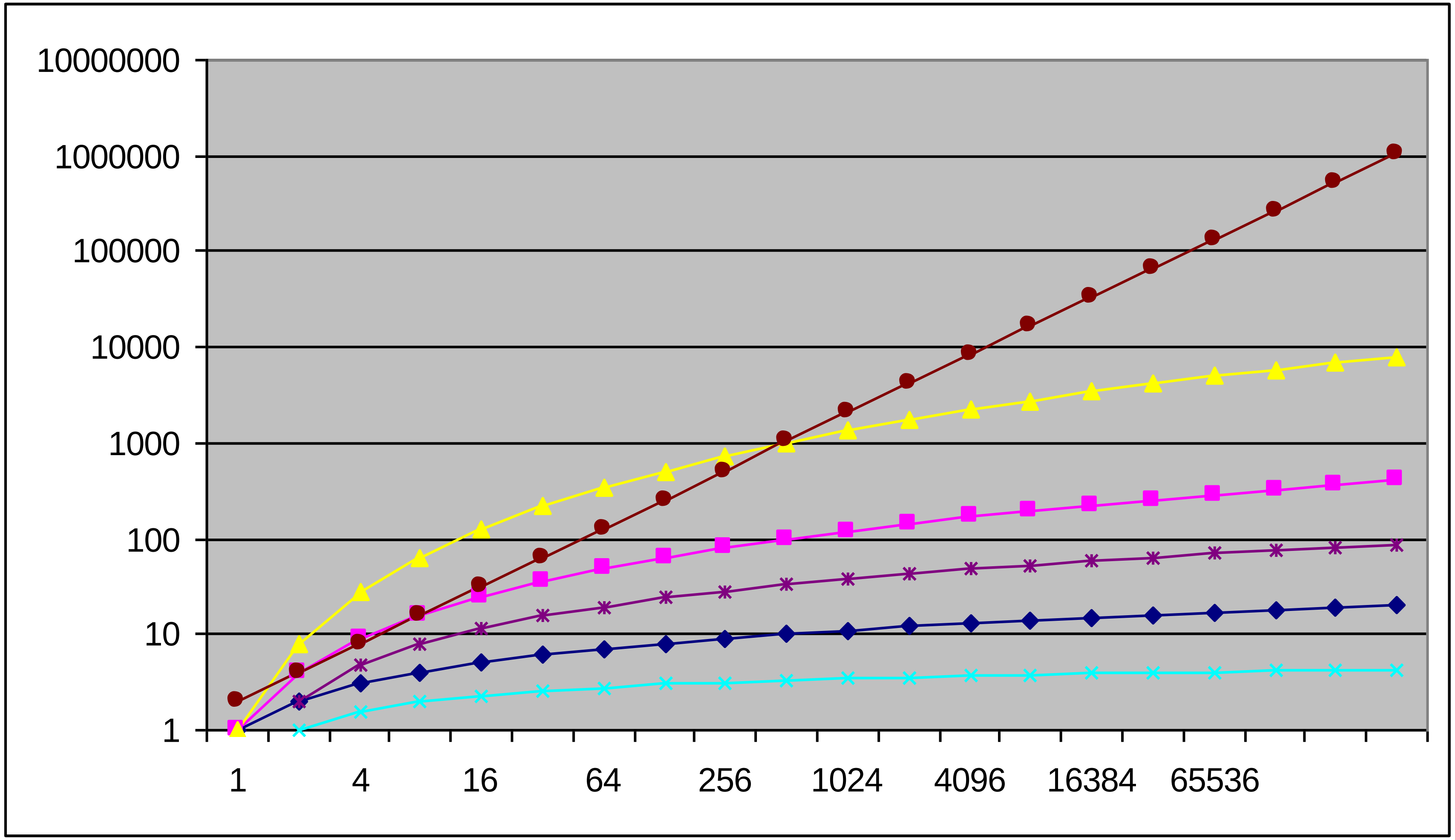
Practical Complexity



Practical Complexity



Practical Complexity



Other Asymptotic Notations

- A function f(n) is o(g(n)) if $\exists$ positive constants c and n_0 such that
  $f(n) < c\, g(n)\ \forall\ n \geq n_0$
- A function f(n) is ω(g(n)) if $\exists$ positive constants c and n_0 such that
  $c\, g(n) < f(n)\ \forall\ n \geq n_0$
- Intuitively,

  - o() is like <
  - O() is like $\leq$
  - ω() is like >
  - Ω() is like $\geq$
  - Θ() is like =

Up Next

- Solving recurrences
  - Substitution method
  - Master theorem