Runtime Efficiency
and
Asymptotic Analysis

Algorithm Efficiency

There are often many approaches (algorithms) to solve a problem.

How do we choose between them?

Two goals for best design practices:

1. To design an algorithm that is easy to understand, code, debug.
2. To design an algorithm that makes efficient use of the computer’s resources.

Algorithm Efficiency (cont)

Goal (1) is the concern of Software Engineering.

Goal (2) is the concern of data structures and algorithm analysis.

When goal (2) is important, how do we measure an algorithm’s cost?

How to Measure Efficiency?

1. Empirical comparison (run programs)
2. Asymptotic Algorithm Analysis

Critical resources:

Factors affecting running time:

For most algorithms, running time depends on “size” of the input.

Running time is expressed as **T**(*n*) for some function **T** on input size *n*.

Efficiency (Complexity)

- The rate at which storage or time grows as a function of the problem size.

- Two types of efficiency:
  - Time efficiency
  - Space efficiency

- There are always tradeoffs between these two efficiencies.
  - Example: singly vs. doubly linked list

Space/Time Tradeoff Principle

- There are always tradeoffs between these two efficiencies.

One can often reduce time if one is willing to sacrifice space, or vice versa.

- Example:
  - singly vs. doubly linked list

Disk-based Space/Time Tradeoff Principle:

The smaller you make the disk storage requirements, the faster your program will run.

Time Efficiency

- How do we improve the time efficiency of a program?

- **The 90/10 Rule**

  90% of the execution time of a program is spent in executing 10% of the code

- So, how do we locate the **critical 10%**?

  - software metrics tools
  - global counters to locate bottlenecks (loop executions, function calls)

Time Efficiency Improvements

Possibilities (some better than others!)

- Move code **out of loops** that does not belong there (just good programming!)

- Remove any **unnecessary I/O** operations (I/O operations are expensive timewise)

- Code so that the compiled code is more efficient

- Replace an inefficient algorithm (best solution)

Moral - Choose the most appropriate algorithm(s) BEFORE program implementation

Real Time Vs Logical units

- To Evaluate an algorithm efficiency real time unit such as microsecond and nanosecond should no be used.

- Logical unit that express a relationship between the **size n** of data and the amount of **time t** required to process the data should be used.

Complexity Vs Input Size

- Complexity function that express relationship between n and t is usually much more complex, and calculating such a function only in regard to **large amount of data**.
- Normally, we are not so much interested in the **time** and **space** complexity for small inputs.

- For example, while the difference in time complexity between linear and binary search is meaningless for a sequence with **n = 3**, it is more significant for for **n = 2^{30}.**

Example

- N^2 is less than 2^N
  - $N=2$ $2^2=4$ $2^2= 4$ Both are same
  - $N=3$ $3^2=9$ $2^3= 8$ N^2 is greater than 2^N
  - $N=4$ $4^2=16$ $2^4= 16$ both are same
  - $N=5$ $5^2=25$ $2^5=32$ 2^N is greater than N^2

  $N^2 < 2^N$ for all $N >=N^o$ where $N^o =4$

Complexity

- Comparison: time complexity of algorithms A and B

Input Size	Algorithm A	Algorithm B
n	5,000n	$\lceil 1.1^n \rceil$
10	50,000	3
100	500,000	13,781
1,000	5,000,000	$2.5 \cdot 10^{41}$
1,000,000	$5 \cdot 10^9$	$4.8 \cdot 10^{41392}$

Complexity

- This means that **algorithm B** cannot be used for large inputs, while running **algorithm A** is still feasible.

- So what is important is the **growth** of the complexity functions.

- The growth of time and space complexity with increasing input size n is a suitable measure for the comparison of algorithms.

Best, Worst, Average Cases

Not all inputs of a given size take the same time to run.

Sequential search for *K* in an array of *n* integers:

- Begin at first element in array and look at each element in turn until *K* is found

Best case:

Worst case:

Average case:

Best Case

- Best case is defined as which input of size *n* is cheapest among all inputs of size *n*.
- Example

1- Already sorted array in Insertion Sort

2- Key element exist at first position in Array for linear search

Worst Case

- Worst case refers to the worst input from among the choices for possible inputs of a given size.
- Example

1- Linear Search

key element is the last element in array

2- Insertion Sort

Array is in reverse order sorted

Average Case

- Average case appears to be the fairest measure.
- Data is equally likely to occur at any position in any order
- It may be difficult to determine
- Probabilistic Analysis are used
- Example

1- Randomly choose n numbers and apply insertion sort

Asymptotic Analysis (Runtime Analysis)

- Independent of any specific hardware or software
- Expresses the complexity of an algorithm in terms of its relationship to some known function
- The efficiency can be expressed as “proportional to” or
  “on the order of” some function
- A common notation used is called Big-Oh:

$$T(n) = O(f(n))$$

Said: T of n is “on the order of” f(n)

Asymptotic Analysis

- This idea is incorporated in the "Big Oh" notation for asymptotic performance.

- **Definition**: $T(n) = O(f(n))$
- if and only if there are constants $c0$ and $n0$ such that $T(n) <= c0\ f(n)$ for all $n >= n0$.

Example Algorithm and Corresponding Big-Oh

Algorithm:

- **prompt the user for a filename and read it (400 “time units”)**
- **open the file (50 “time units”)**
- **read 15 data items from the file into an array (10 “time units” per read)**
- **close the file (50 “time units”)**

Formula describing algorithm efficiency:

500 + 10n where n=number of items read (here, 15)

Which term of the function really describes the growth pattern as n increases?

Therefore, the Big-Oh for this algorithm is O(n).

Example (Con’t)

- Function n will *never* be larger than the function $500 + 10\ n$, no matter how large n gets.
- However, there are constants c0 and n0 such that $500 + 10n <= c0\ n$ when $n >= n0$. One choice for these constants is $c0 = 20$ and $n0 = 50$.
- Therefore, $500 + 10n = O(n)$.
- Other choices for c0 and n0. For example, any value of $c0 > 20$ will work for $n0 = 50$.

Common Functions in Big-Oh
(Most Efficient to Least Efficient)

- **O(1) or O(c)**

  Constant growth. The runtime does not grow at all as a function of n. It is a constant. Basically, it is any operation that does not depend on the value of n to do its job. Has the slowest growth pattern (none!).

  Examples:

  - Accessing an element of an array containing n elements
  - Adding the first and last elements of an array containing n elements
  - Hashing

Common Functions in Big-Oh (con’t)

- **O(lg(n))**

  Logarithmic growth. The runtime growth is proportional to the base 2 logarithm (lg) of n.

  Example:

  - Binary search

Common Functions in Big-Oh (con’t)

- **O(n)**

  Linear growth. Runtime grows proportional to the value of n.

  Examples:

  - Sequential (linear) search
  - Any looping over the elements of a one-dimensional array (e.g., summing, printing)

Common Functions in Big-Oh (con’t)

- **O(n lg(n))**

  n log n growth. Any sorting algorithm that uses comparisons between elements is O(n lg n), based on divide an conquer approach.

  Examples

  - Merge sort
  - Quicksort

Common Functions in Big-Oh (con’t)

- **$O(n^k)$**

  Polynomial growth. Runtime grows very rapidly.

  Examples:

  - Bubble sort ($O(n^2)$)
  - Selection (exchange) sort ($O(n^2)$)
  - Insertion sort ($O(n^2)$)

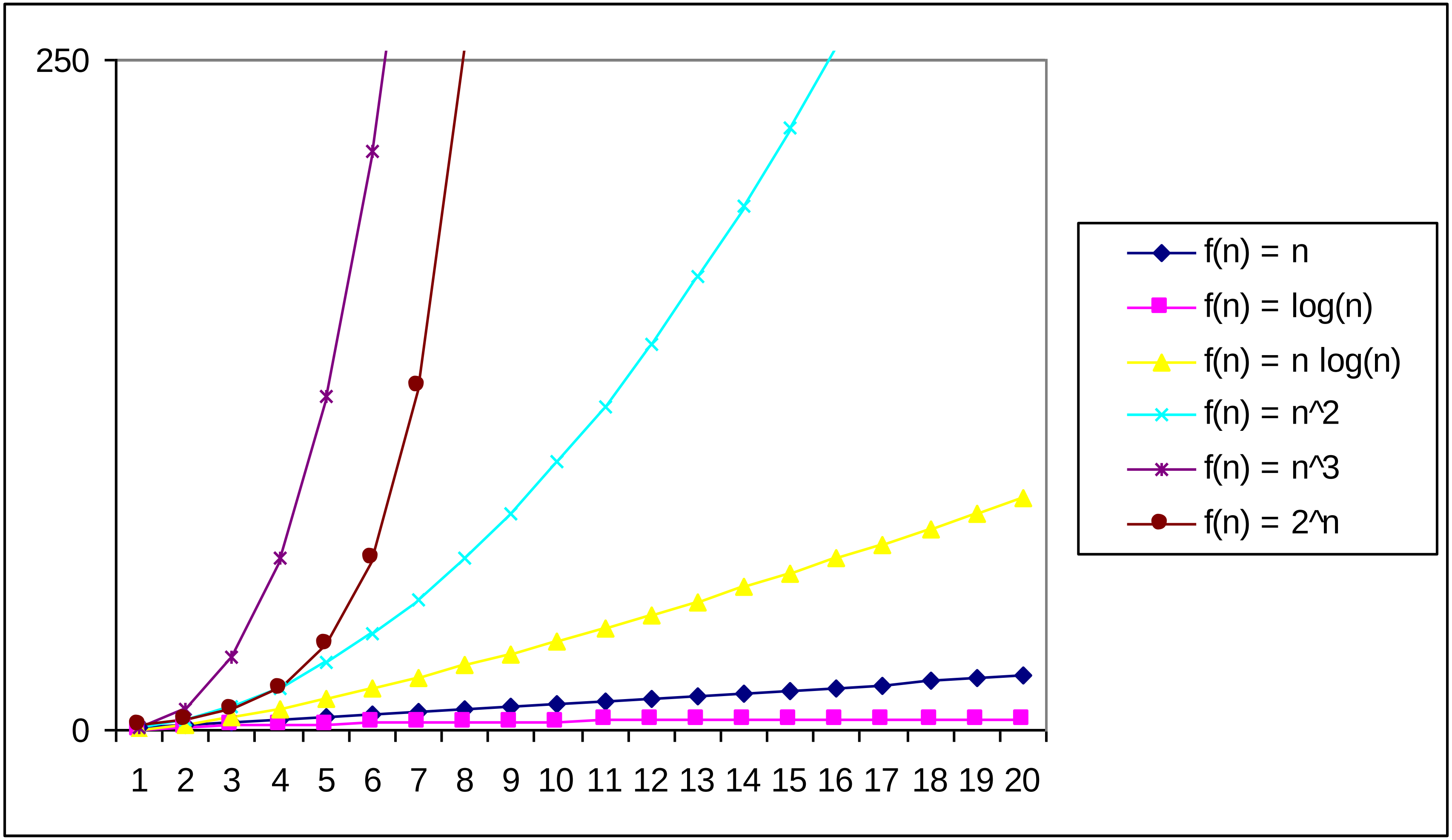
Common Functions in Big-Oh (con't)

- **O(2^n)**

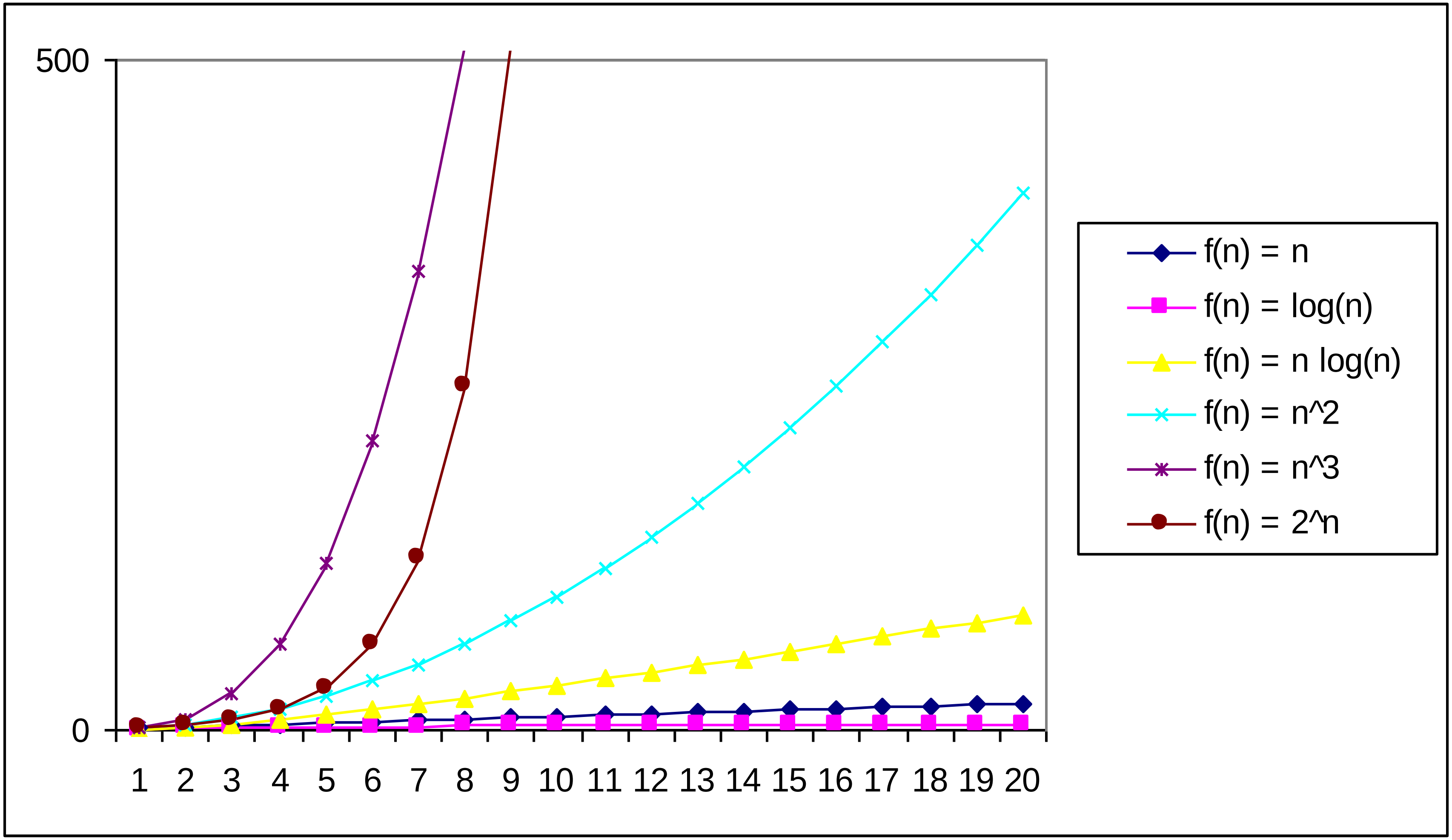
  Exponential growth. Runtime grows extremely rapidly as n increases. Are essentially useless except for very small values of n.

- **Others**
  - O(sqrt(n))
  - O(n!)

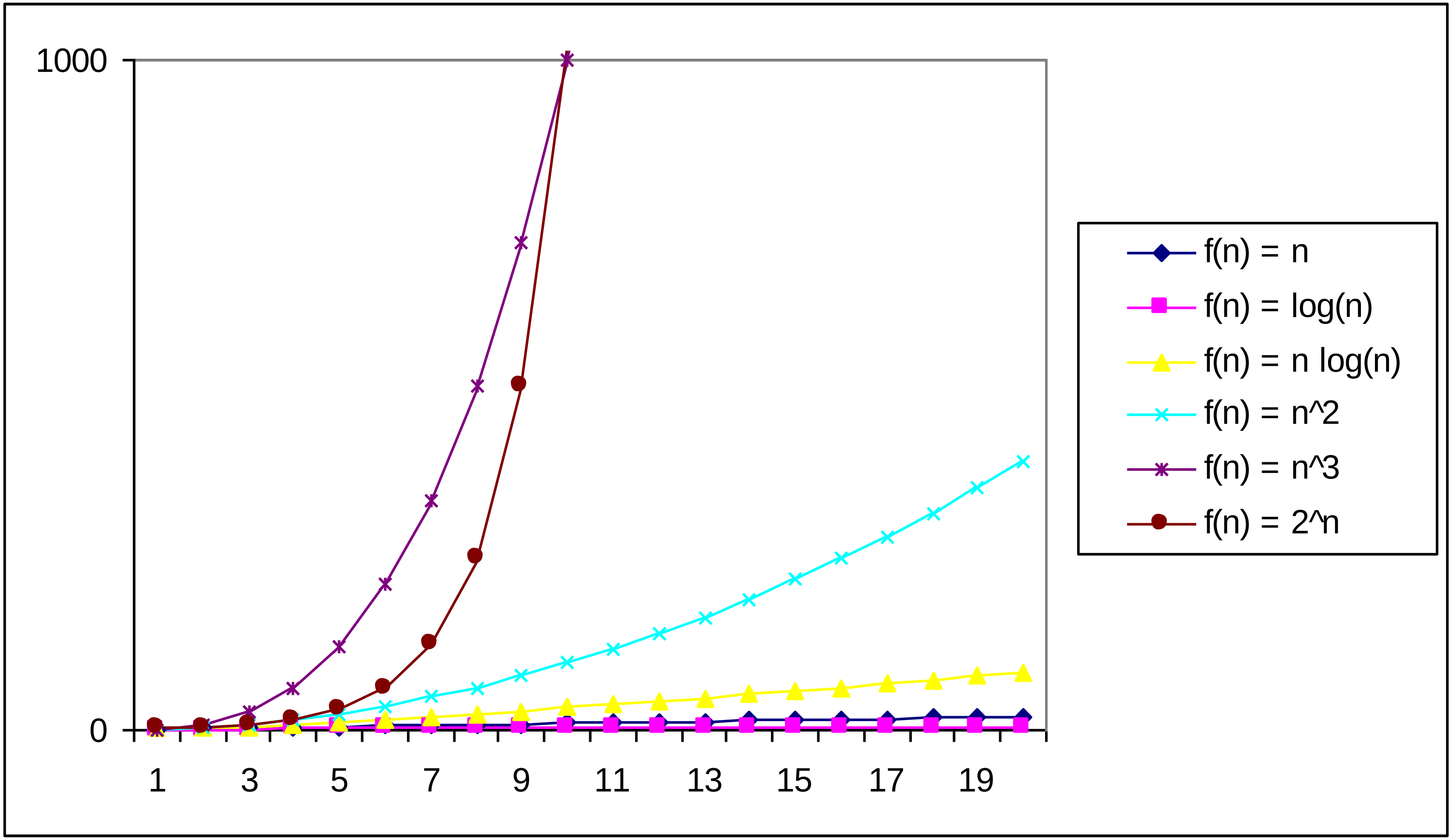
Practical Complexity



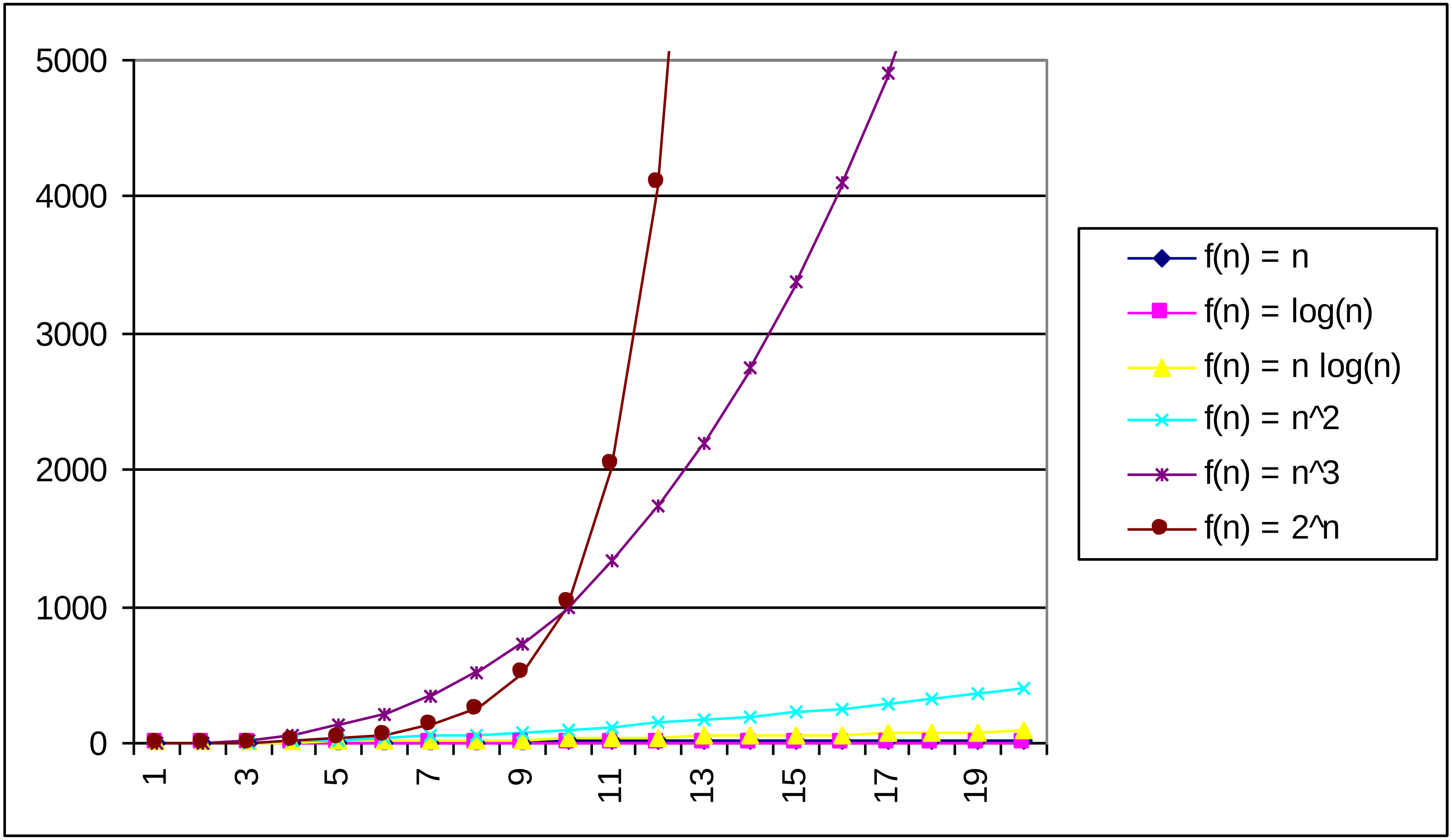
Practical Complexity



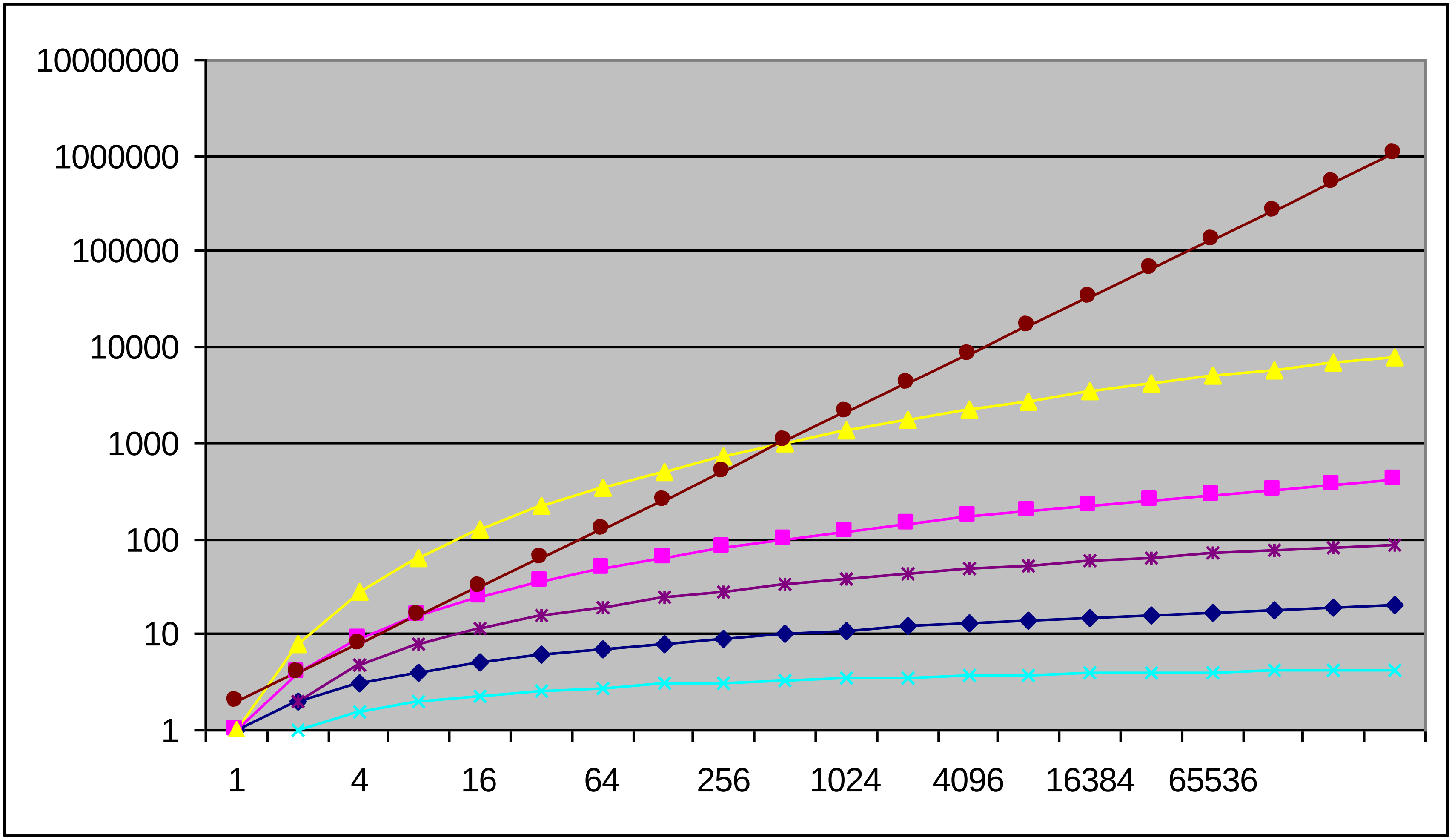
Practical Complexity



Practical Complexity



Practical Complexity



Asymptotic Notations

- These notations are used to describe the asymptotic running time of an algorithm, are defined in terms of function whose domain are set of natural number N={0,1,2,……}

Asymptotic Time Complexity

- The limiting behavior of the **execution time** of an ***algorithm*** when the size of the problem goes to infinity.

Asymptotic Space Complexity

- The limiting behavior of the use of **memory space** of an ***algorithm*** when the size of the problem goes to infinity.

Asymptotic Tight Bound

Θ- Notation

- f(n) = Θ(g(n)) where Θ(g(n)) are the set of functions
- A function f(n) is Θ(g(n)) if there exist positive constants c_1, c_2, and n_0 such that

$$0 \leq c_1\, g(n) \leq \mathbf{f(n)} \leq c_2\, g(n) \quad \text{for all } n \geq n_0$$

- Theorem
  - f(n) is Θ(g(n)) if f(n) is both O(g(n)) and Ω(g(n))

Θ (Cont.)

- **A function f(n) belongs to the set** $\Theta(g(n))$ if there exist positive constants c1 and c2 such that it can be "sandwiched" between c1g(n) and c2g(n), for all large n.

Example: $3n+2 = \Theta(n)$

as $3n+2 \geq 3n$ for all $n \geq 2$,

and $3n+2 \leq 4n$ for all $n \geq 2$

so c1=3, c2=4 and n0=2.

$3n + 3 = \Theta(n)$

-

asymptotic upper bound
big-O notation

- An *asymptotic bound*, as function of the size of the input, on the worst (slowest, most amount of space used, etc.) an *algorithm* will do to solve a problem.
- That is, no input will cause the algorithm to use more resources than the bound
  - 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): there exist positive constants *c* and n_0 such that
  - $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

Example 3n+2 = O(n) as 3n+2 <= 4n for al n>=2 i.e. 3n+3=O(n)

Big-Oh (Cont.)

- The idea behind the big-O notation is to establish an **upper boundary** for the growth of a function f(n) for large .
- This boundary is specified by a function g(x) that is usually much **simpler** than f(x).
- We accept the constant C in the requirement
- $f(n) \le C \cdot g(n)$ whenever $n > n0$,
- We are only interested in large x, so it is OK if
  $f(x) > C \cdot g(x)$ for $x \le k$.
- The relationship between f and g can be expressed by stating either that g(n) is an upper bound on the value of f(n) or that in the long run , f grows at most as fast as g.

Big-Oh (Cont.)

- Question: If f(x) is O(x^2), is it also O(x^3)?

- **Yes.** x^3 grows faster than x^2, so x^3 grows also faster than f(x).

- Therefore, we always have to find the **smallest** simple function g(x) for which f(x) is O(g(x)).

Useful Rules for Big-O

- For any **polynomial** $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$, where $a_0, a_1, \ldots, a_n$ are real numbers,
- $f(x)$ is $O(x^n)$.

- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then $(f_1 + f_2)(x)$ is $O(\max(g_1(x), g_2(x)))$

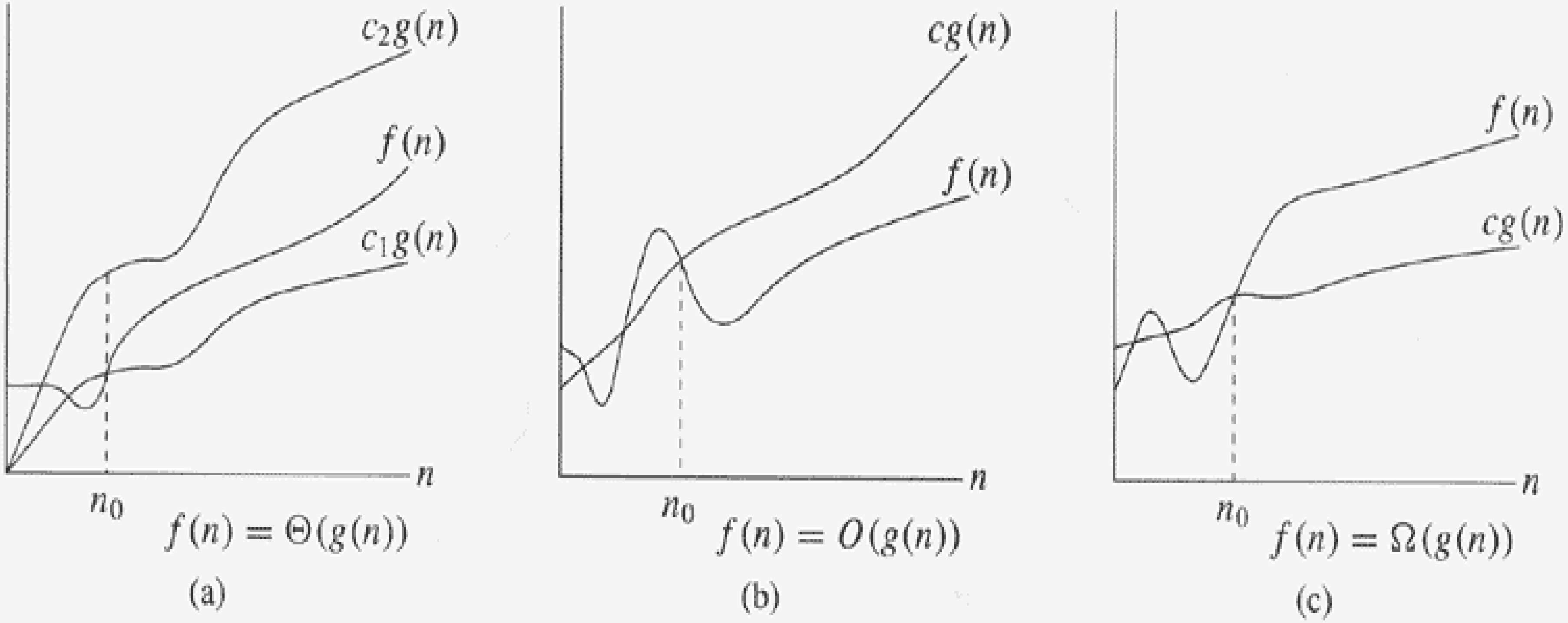
- If $f_1(x)$ is $O(g(x))$ and $f_2(x)$ is $O(g(x))$, then $(f_1 + f_2)(x)$ is $O(g(x))$.

- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then $(f_1 f_2)(x)$ is $O(g_1(x)\, g_2(x))$.

Asymptotic Lower bound

Ω notation

- **An *asymptotic bound*, as function of the size of the input, on the best (fastest, least amount of space used, etc.) an *algorithm* can possibly achieve to solve a problem.**
- **Ω notation provides an asymptotic lower bound on a function**
- **That is, no algorithm can use fewer resources than the bound**
  - f(n) is $\Omega(g(n))$ if there exists positive constants c and n_0 such that
  - $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$
- Theorem
  - f(n) is Θ(g(n)) if f(n) is both O(g(n)) and Ω(g(n))



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

(a)

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

(b)

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

(c)

Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. **(b)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ 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 $cg(n)$. **(c)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ 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 above $cg(n)$.