Data Structure and Algorithms-I

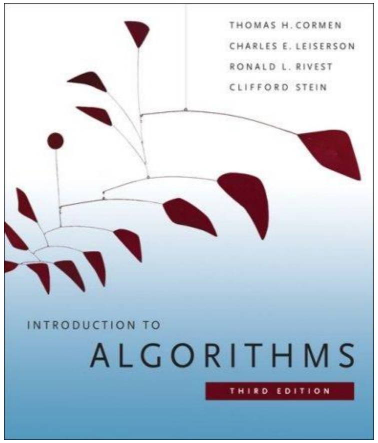
Introduction to Asymptotic Analysis

Lecturer Saifur Rahman, Dept. of CSE, United International University

The Course

- Purpose: a rigorous introduction to the design and analysis of algorithms
  - Not a programming course
  - Not a math course, either

- Textbook: *Introduction to Algorithms* (3rd edition) Cormen, Leiserson, Rivest, and Stein
  - An excellent reference you should own



What is a Data Structure?

- Data is a collection of facts, such as values, numbers, words, measurements, or observations.
- Structure means a set of rules that holds the data together.
- A data structure is a particular way of storing and organizing data in a computer so that it can be used **efficiently**.
  - Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks.
  - Data Structures provide a means to manage huge amount of data efficiently.
  - Usually, efficient data structures are a key to designing efficient algorithms.
  - Data structures can be nested.

Types of Data Structures

- Data structures are classified as either
  - Linear (*e.g*, arrays, linked lists), or
  - Nonlinear (*e.g*, trees, graphs, etc.)

- A data structure is said to be linear if it satisfies the following four conditions
  - There is a unique element called the first
  - There is a unique element called the last
  - Every element, except the last, has a unique successor
  - Every element, except the first, has a unique predecessor

- There are two ways of representing a linear data structure in memory
  - By means of sequential memory locations (arrays)
  - By means of pointers or links (linked lists)

What is an Algorithm?

- An algorithm is a sequence of computational steps that solves a well-specified computational problem.
  - An algorithm is said to be correct if, for every input instance, it halts with the correct output
  - An incorrect algorithm might not halt at all on some input instances, or it might halt with other than the desired output.

What is a Program?

- A program is the expression of an algorithm in a programming language
- A set of instructions which the computer will follow to solve a problem

Define a Problem, and Solve It

- **Problem**:
  - Description of Input-Output relationship
- **Algorithm**:
  - A sequence of computational steps that transform the input into the output.
- **Data Structure:**
  - An organized method of storing and retrieving data.
- **Our Task:**
  - Given a problem, design a ***correct*** and ***good*** algorithm that solves it.

Define a Problem, and Solve It

Problem: Input is a sequence of integers stored in an array.
Output the minimum.

Algorithm

INPUT

instance

25, 90, 53, 23, 11, 34

```
m:= a[1];
for I:=2 to size of input
     if  m > a[i]  then
          m:=a[I];
return s
```

OUTPUT

11

m, a[i]

Data-Structure

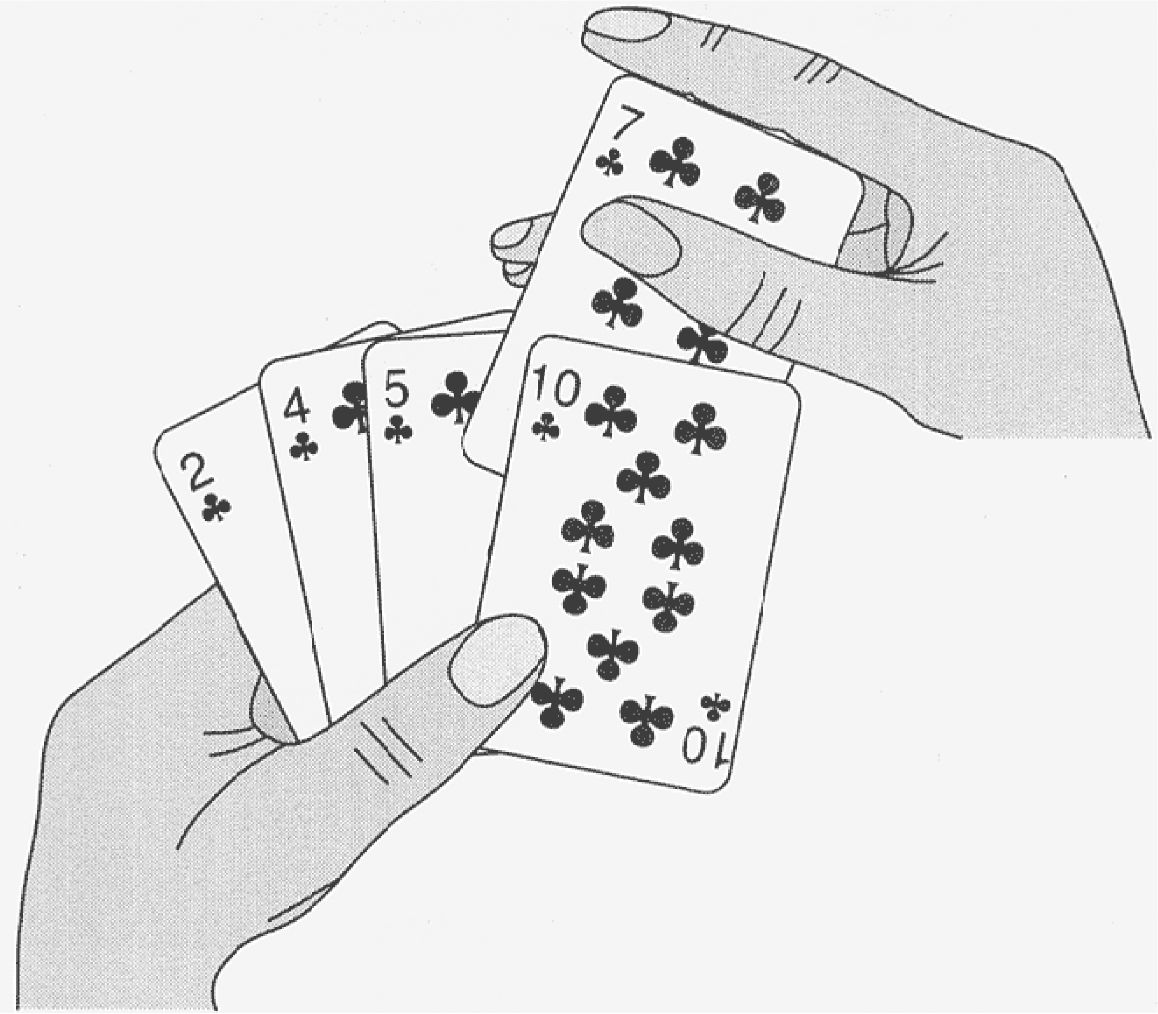
What do we Analyze?

- Correctness
  - Does the input/output relation match algorithm requirement?
- Amount of work done (complexity)
  - Basic operations to do task
- Amount of space used
  - Memory used
- Simplicity, clarity
  - Verification and implementation.
- Optimality
  - Is it impossible to do better?

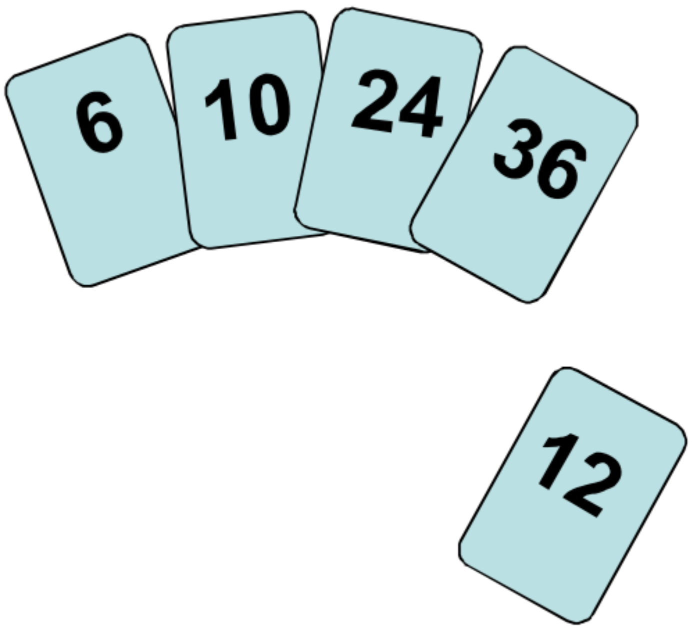
Running Time

- Number of primitive steps that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
    - y = m * x + b
    - c = 5 / 9 * (t - 32)
    - z = f(x) + g(y)
- We can be more exact if need to be

An Example: Insertion Sort

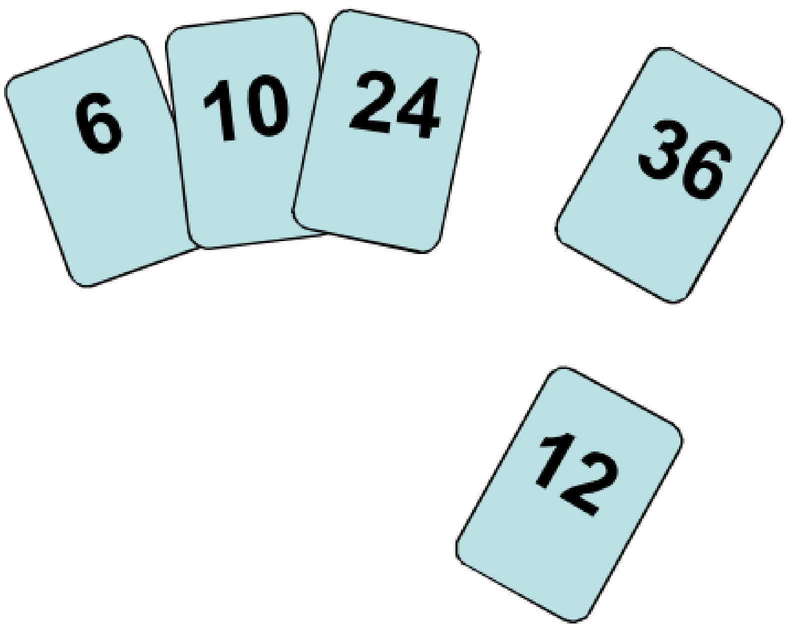


An Example: Insertion Sort



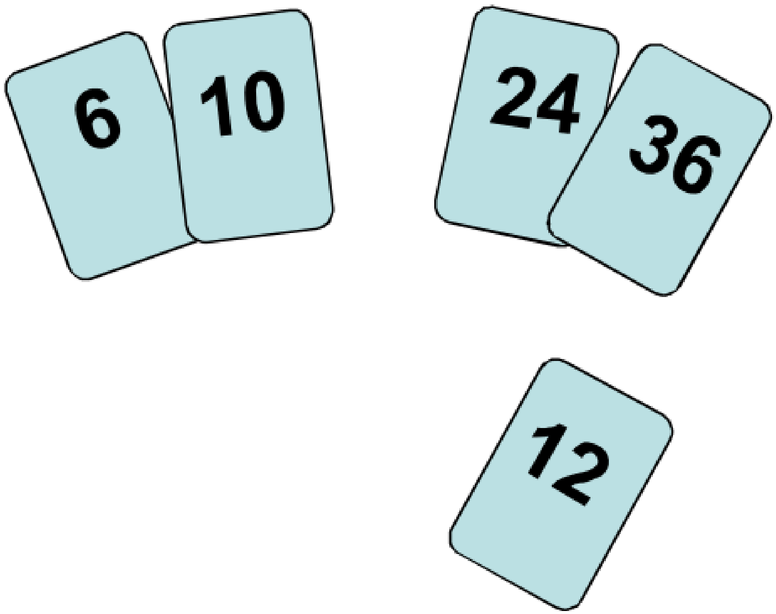
To insert 12, we need to make room for it by moving first 36 and then 24.

An Example: Insertion Sort



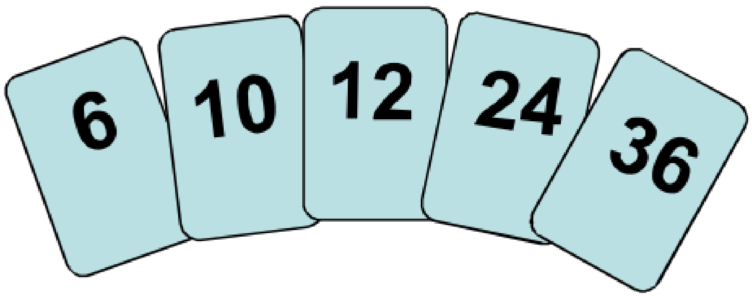
To insert 12, we need to make room for it by moving first 36 and then 24.

An Example: Insertion Sort



To insert 12, we need to make room for it by moving first 36 and then 24.

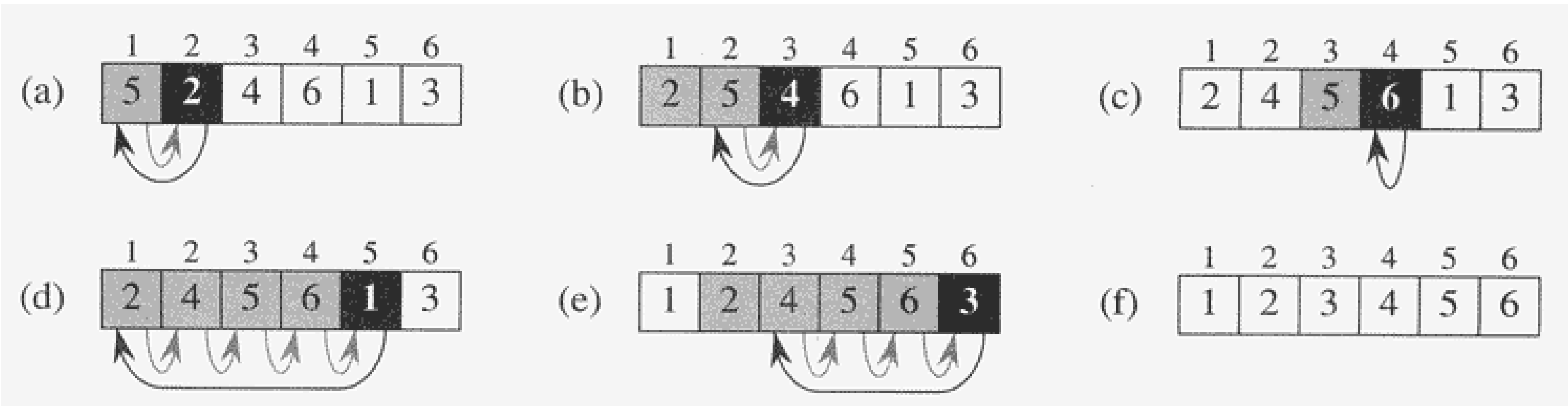
An Example: Insertion Sort



To insert 12, we need to make room for it by moving first 36 and then 24.

An Example: Insertion Sort

A = {5, 2, 4, 6, 1, 3}



An Example: Insertion Sort

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```

An Example: Insertion Sort

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```

How many times will this loop execute?

Analyzing Insertion Sort

Statement	Cost	Times
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	$(T-(n-1))$
A[j+1] = key	c_7	$(n-1)$
}		
}		

$T = t_2 + t_3 + \ldots + t_n$, where t_i is the 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:** the array is sorted (inner loop body never executed)
    - $t_i = 1 \rightarrow T = n$
    - $T(n) = an + b$, a linear function of n
  - **Worst case:** the array is reverse sorted (inner loop body executed for all previous elements)
    - $t_i = i \rightarrow T = n(n + 1)/2 - 1$
    - $T(n) = an^2 + bn + c$, a quadratic function of n
  - **Average case:**
    - ???

Asymptotic Performance

- 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.

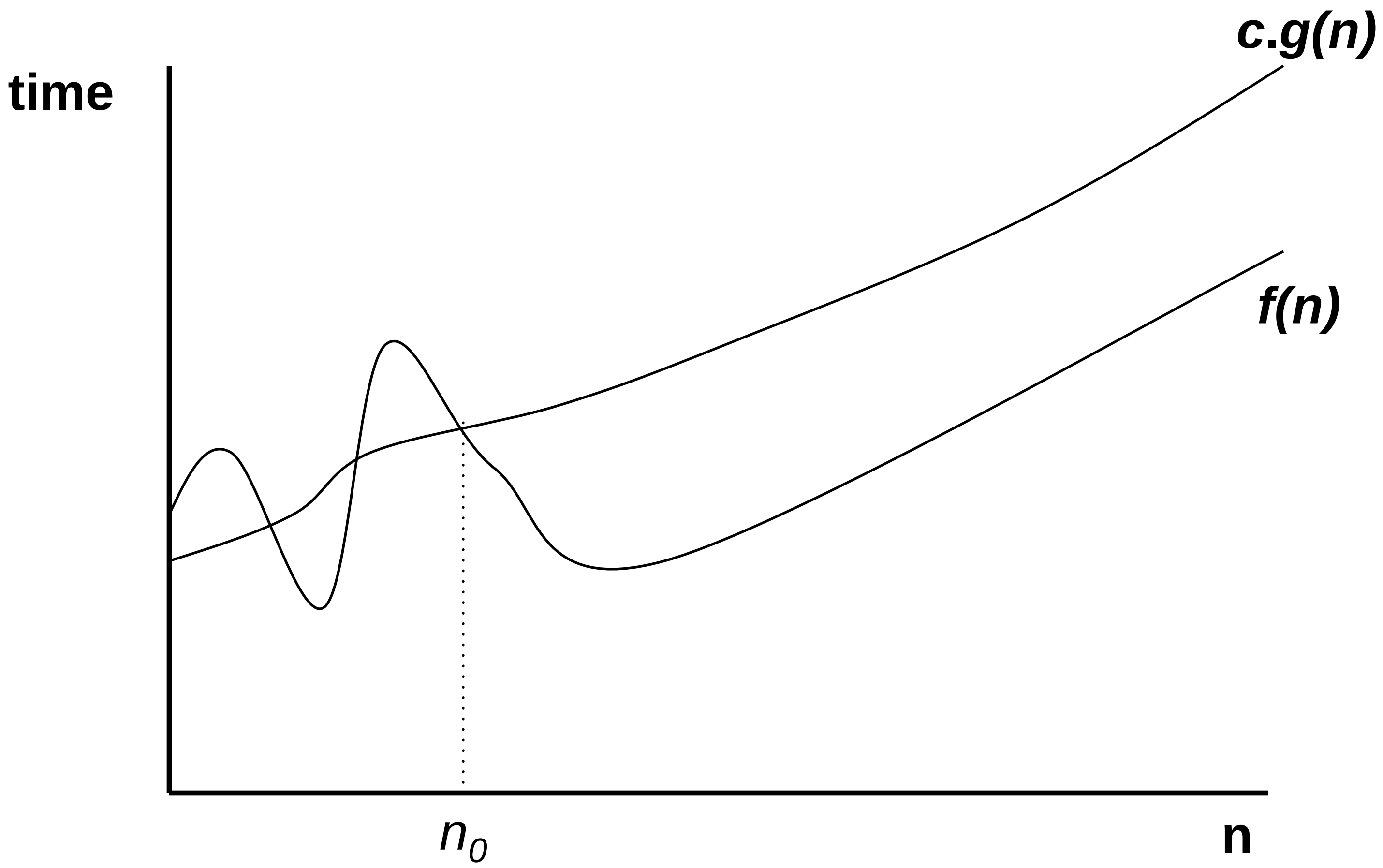
Asymptotic Analysis

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee of required resources
- Average case
  - Provides the expected running time
  - Very useful, but treat with care: what is “average”?
    - Random (equally likely) inputs
    - Real-life inputs
- Best case

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 $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
- Formally
  - $O(g(n)) = \{ f(n)\text{: } \exists$ positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n) \ \forall\ n \geq n_0 \}$

Upper Bound Notation



We say $g(n)$ is an *asymptotic upper bound* for $f(n)$

Insertion Sort is $O(n^2)$

- Proof
  - The run-time is $an^2 + bn + c$
    - If any of a, b, and c are less than 0, replace the constant with its absolute value
  - $an^2 + bn + c \quad \le (a + b + c)n^2 + (a + b + c)n + (a + b + c)$

    $\le 3(a + b + c)n^2$ for $n \ge 1$

    Let $c' = 3(a + b + c)$ and let $n_0 = 1$. Then

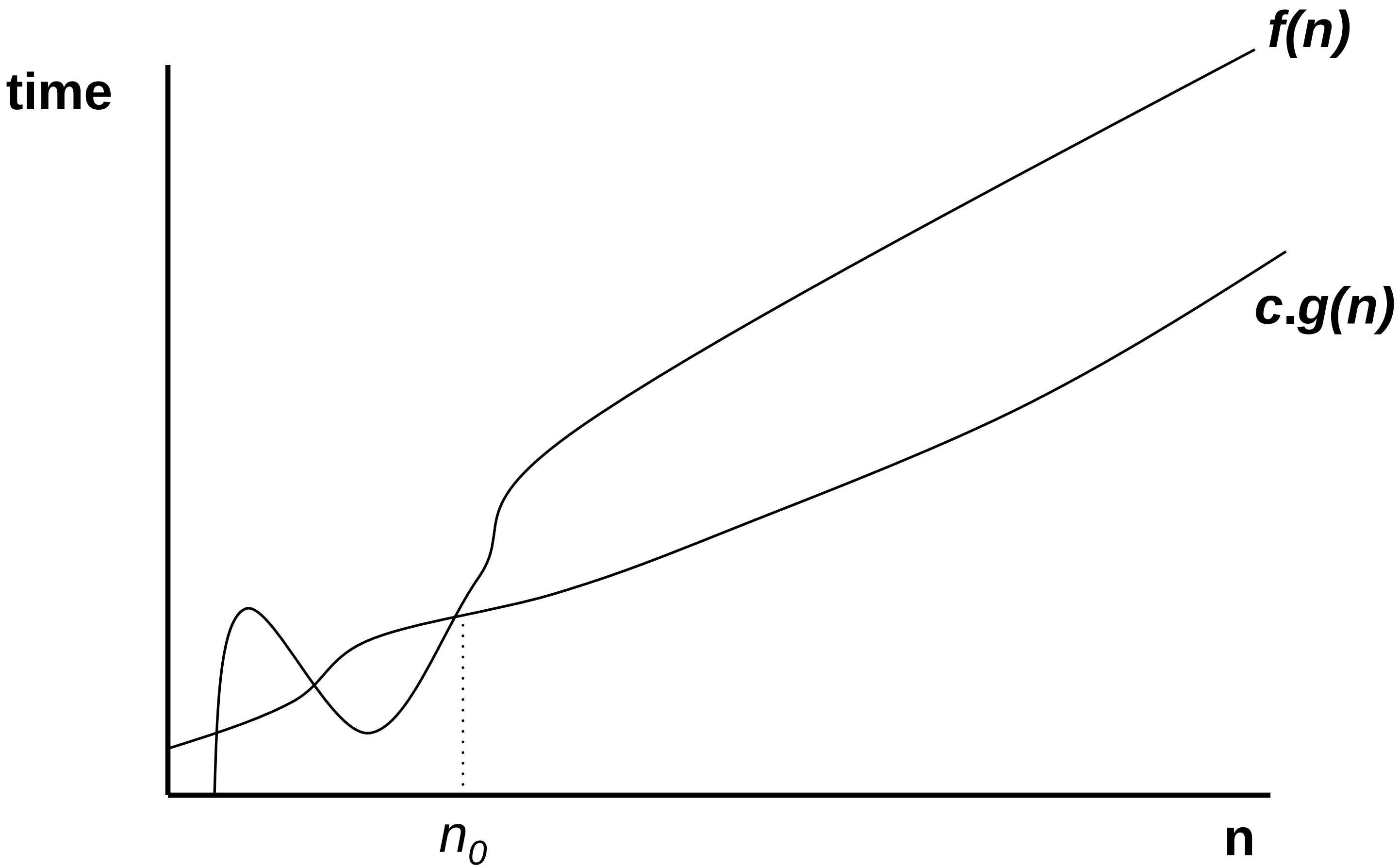
    $an^2 + bn + c \le c' n^2$ for $n \ge 1$

    Thus $an^2 + bn + c = O(n^2)$.
- Question
  - Is InsertionSort $O(n^3)$?
  - Is InsertionSort $O(n)$?

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 $0 \le c \cdot g(n) \le f(n) \;\; \forall\, n \ge n_0$
- Proof:
  - Suppose run time is an + b
    - Assume a and b are positive
  - $an \le an + b$

Lower Bound Notation



We say $g(n)$ is an *asymptotic lower bound* for $f(n)$

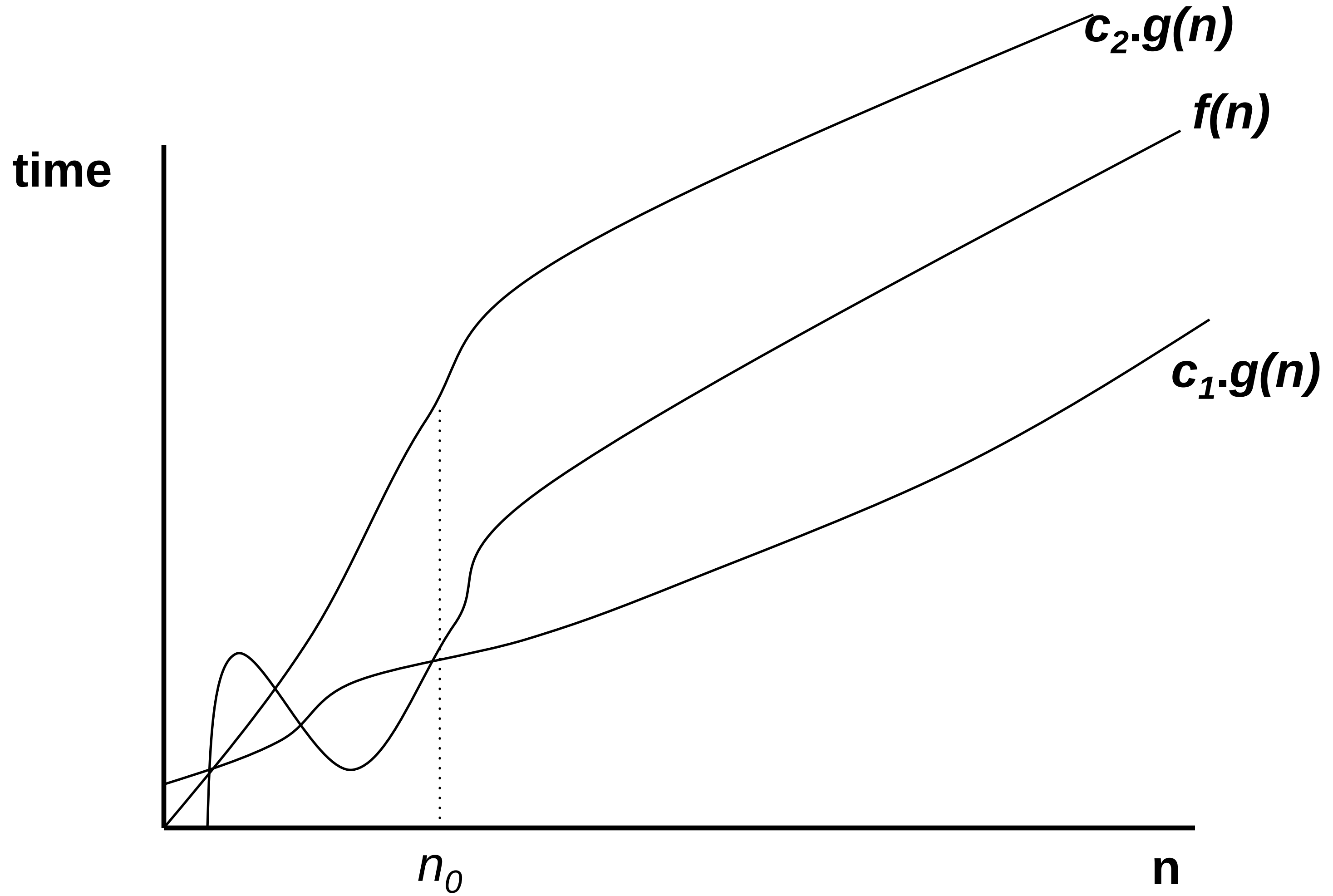
Asymptotic Tight Bound

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

$$0 \le c_1\, g(n) \le f(n) \le c_2\, g(n)\ \forall\ n \ge n_0$$

- Theorem
  - $f(n)$ is $\Theta(g(n))$ iff $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$
  - Proof:

Asymptotic Tight Bound



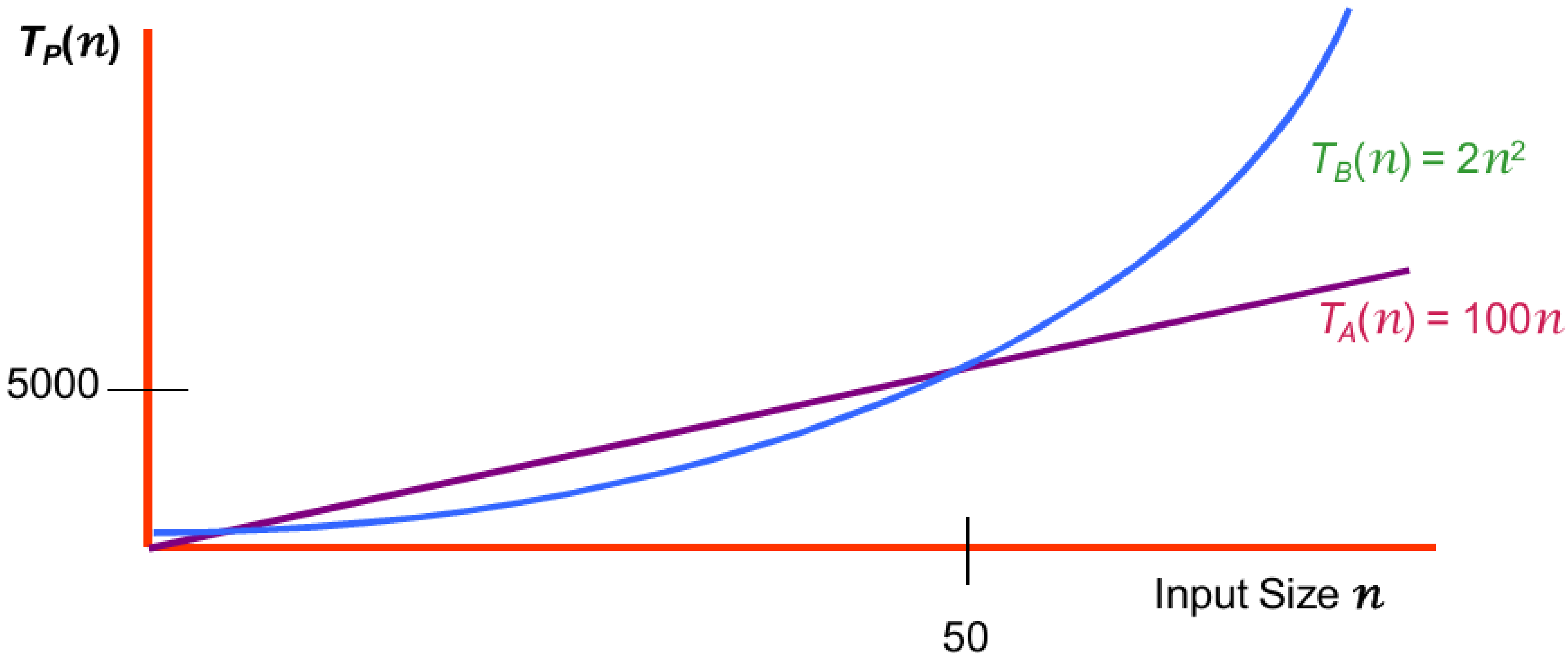
We say $g(n)$ is an *asymptotic tight bound* for $f(n)$

Practical Complexity

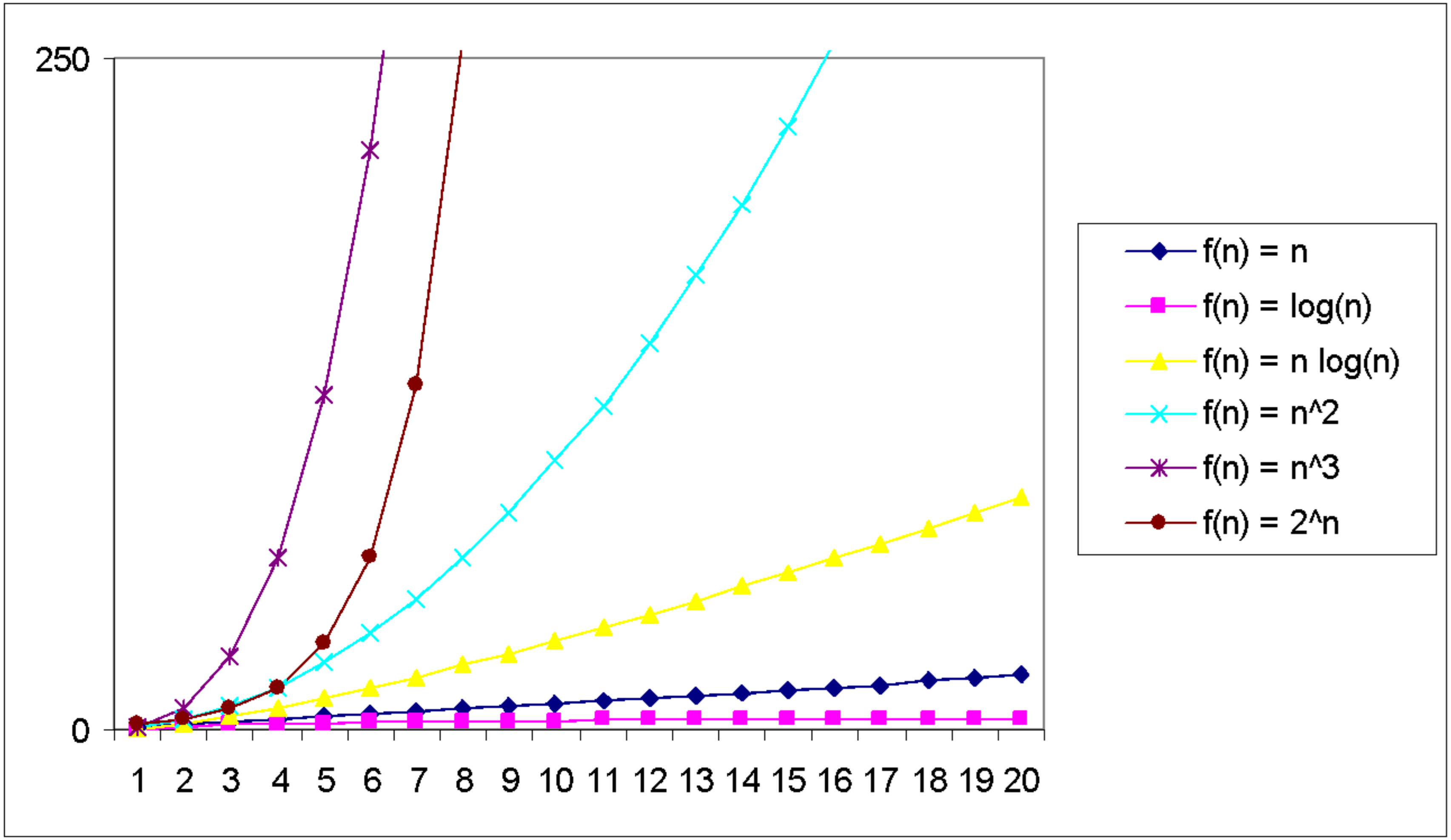
For large input sizes, constant terms are insignificant

Program A with running time $T_A(n) = 100n$

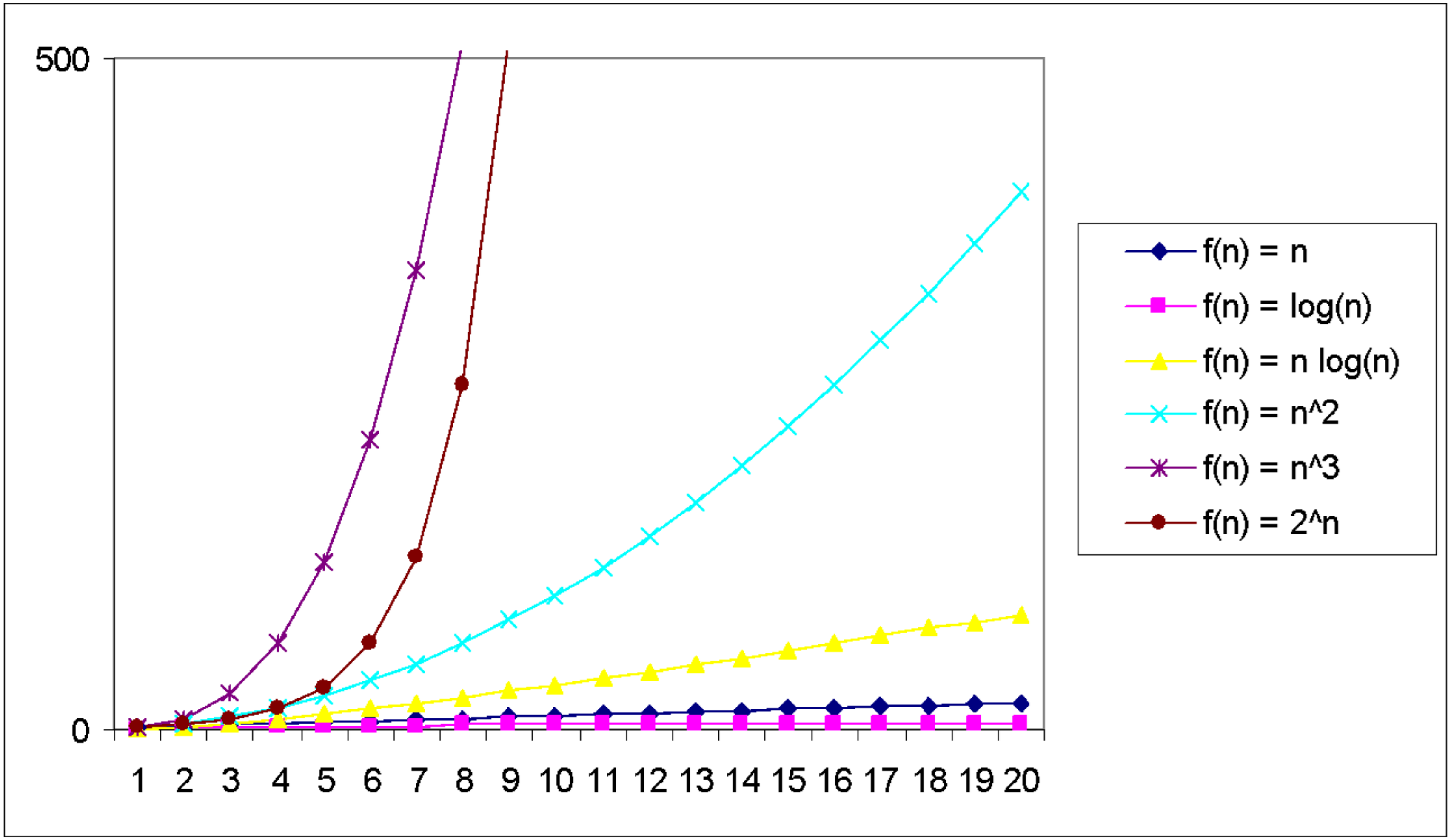
Program B with running time $T_B(n) = 2n^2$



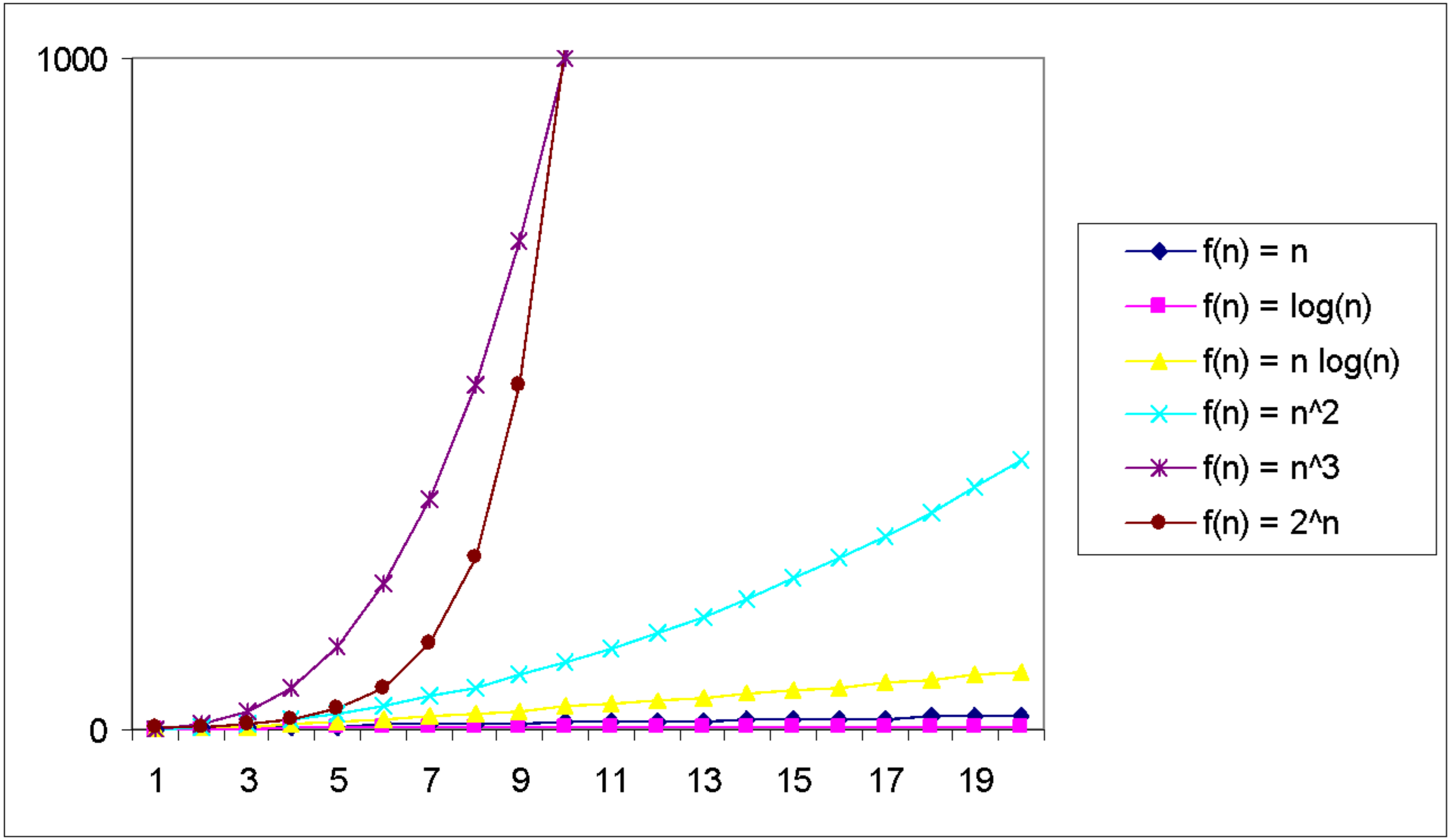
Practical Complexity



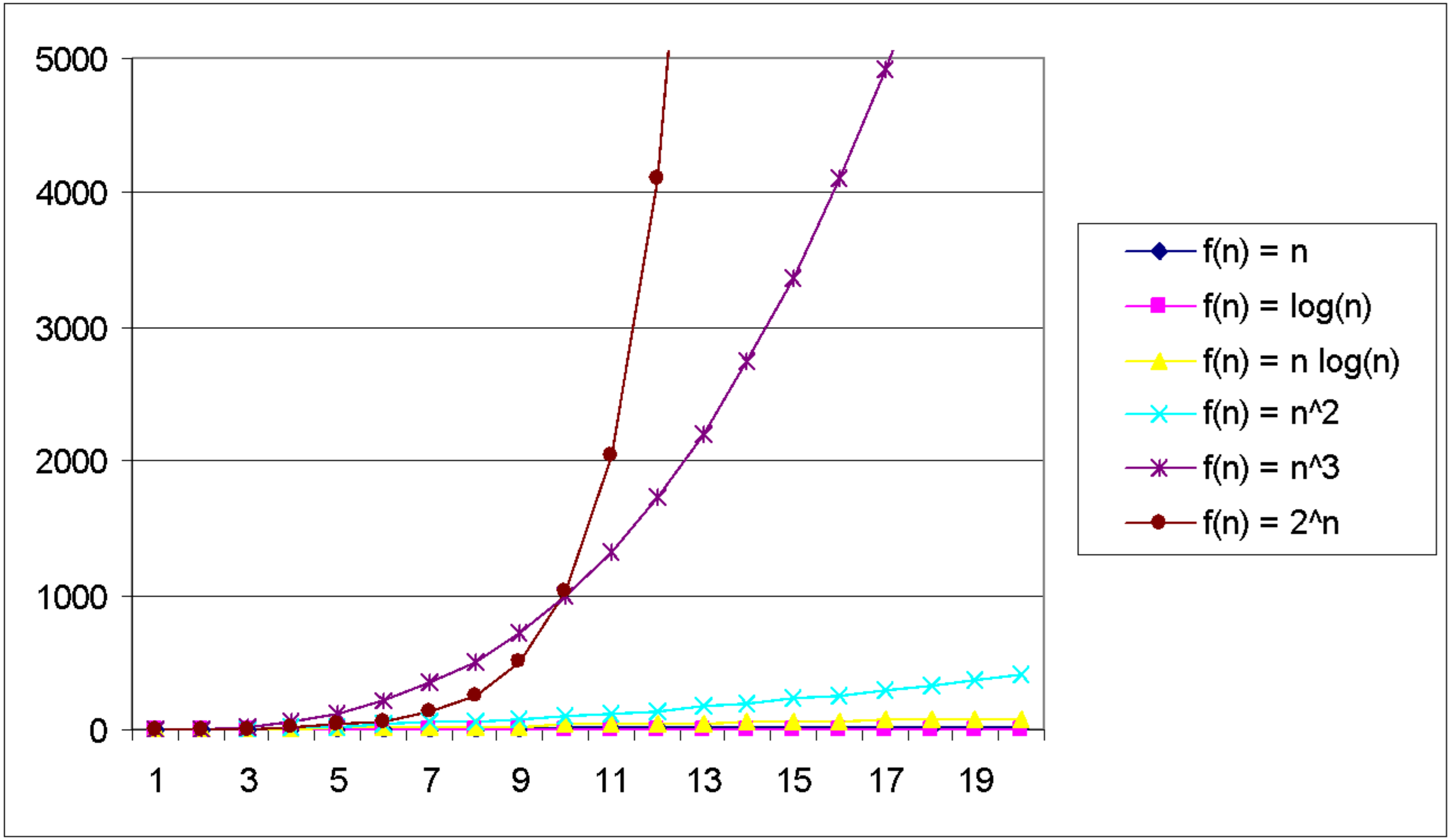
Practical Complexity



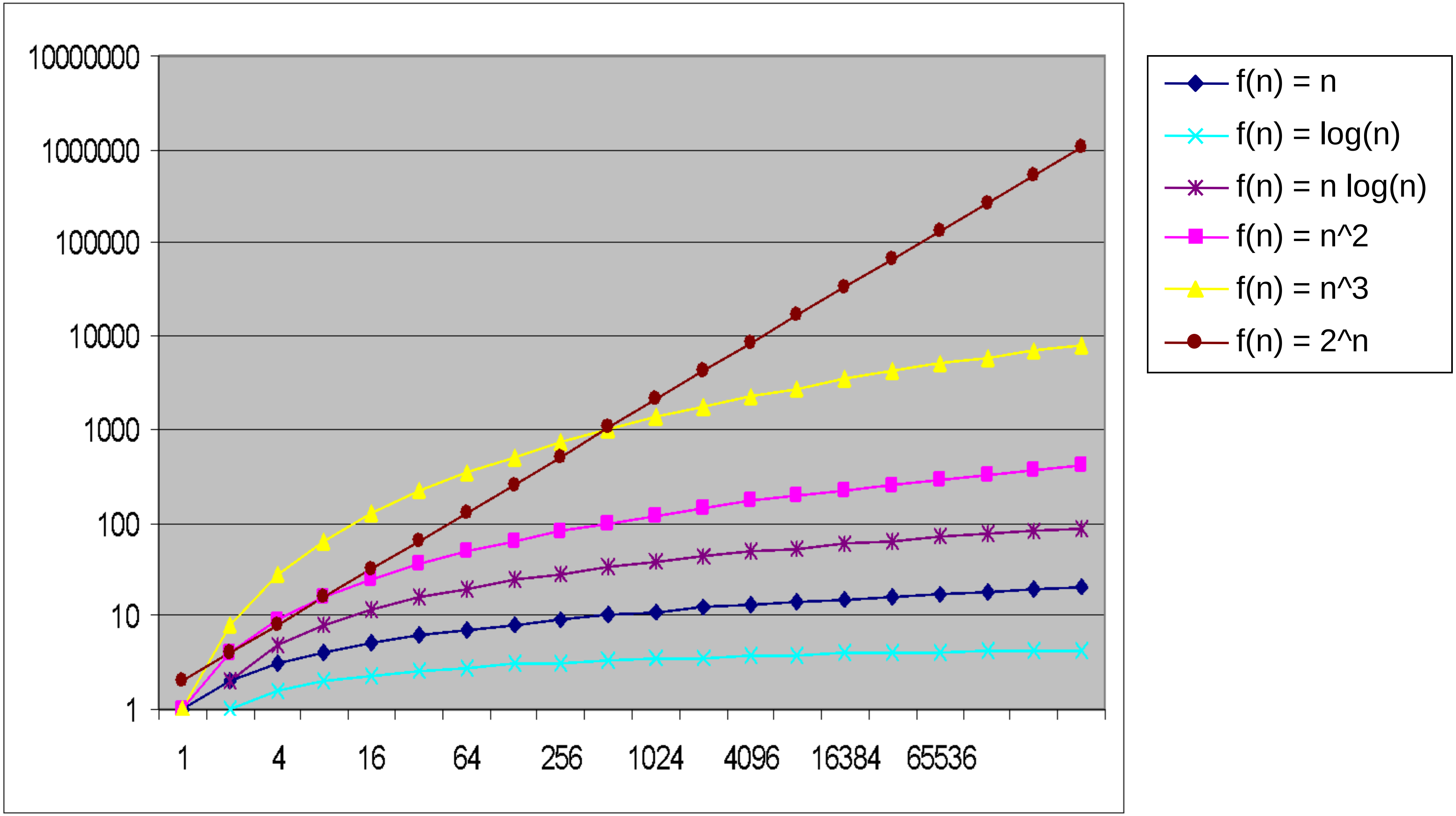
Practical Complexity



Practical Complexity



Practical Complexity



Practical Complexity

Function	Descriptor	Big-Oh
c	Constant	$O(1)$
$\log n$	Logarithmic	$O(\log n)$
n	Linear	$O(n)$
$n \log n$	$n \log n$	$O(n \log n)$
n^2	Quadratic	$O(n^2)$
n^3	Cubic	$O(n^3)$
n^k	Polynomial	$O(n^k)$
2^n	Exponential	$O(2^n)$
$n!$	Factorial	$O(n!)$

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 $\omega(g(n))$ 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 $\leq$
  - $\omega(\ )$ is like $>$
  - $\Omega(\ )$ is like $\geq$
  - $\Theta(\ )$ is like $=$