Data Structure and Algorithms-II

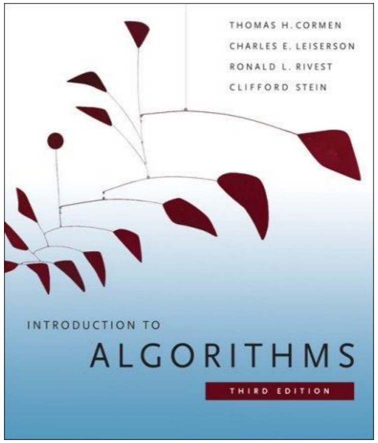
Analyzing Algorithms

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 <u>two ways</u> 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?

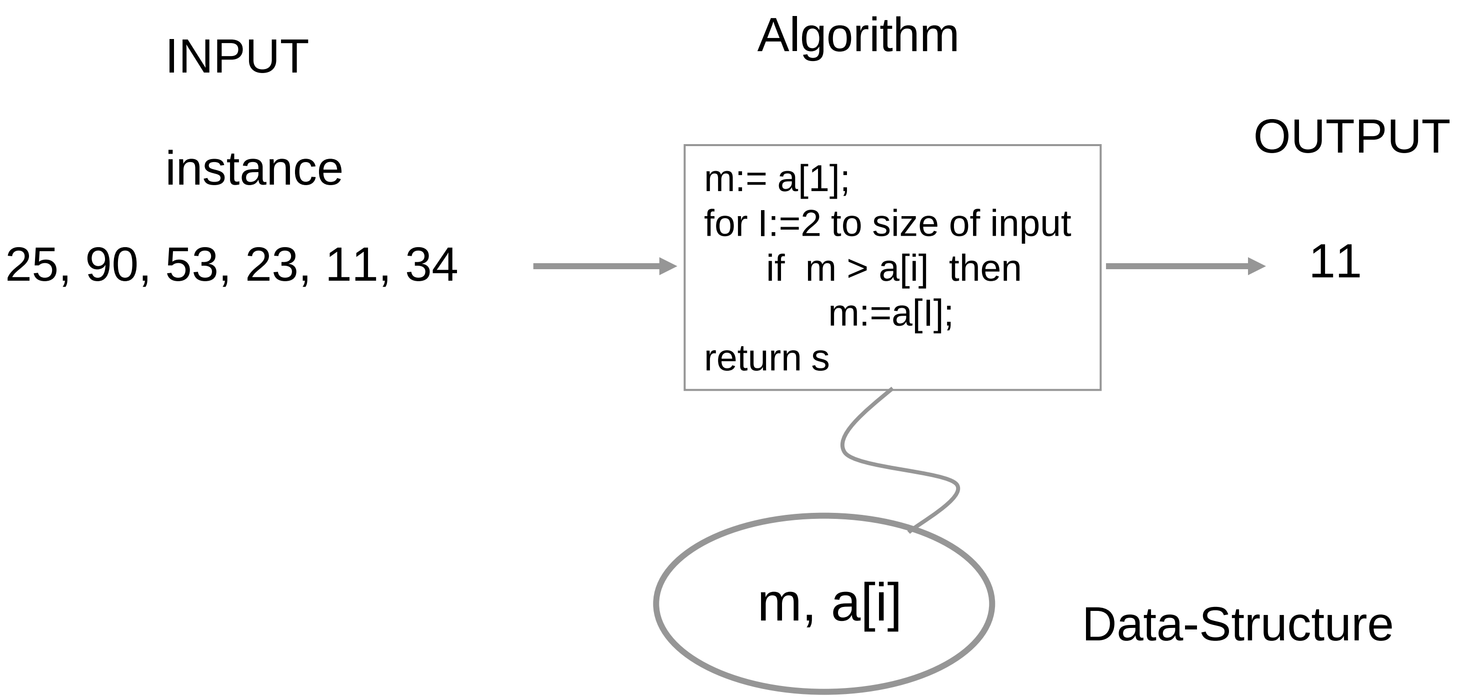
- A program is the expression of an algorithm in a programming language
- A set of instructions which the computer will follow to solve a problem

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.



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?

Analyzing Algorithms

- Asymptotic Notation
- Analyzing Runtime

Asymptotic Analysis

- The term asymptotic means approaching a value (e.g. infinity).
  - $T_1(n) = 10^{10}n^2$
  - $T_2(n) = 10^{-8}n^3$
  - If the max value of n is 10^8 then T_2 is cheaper than T_1
  - However if $n \to \infty$, T_1 is cheaper [Asymptotic]
  - Therefore, asymptotically, $T_1(n) \leq T_2(n)$

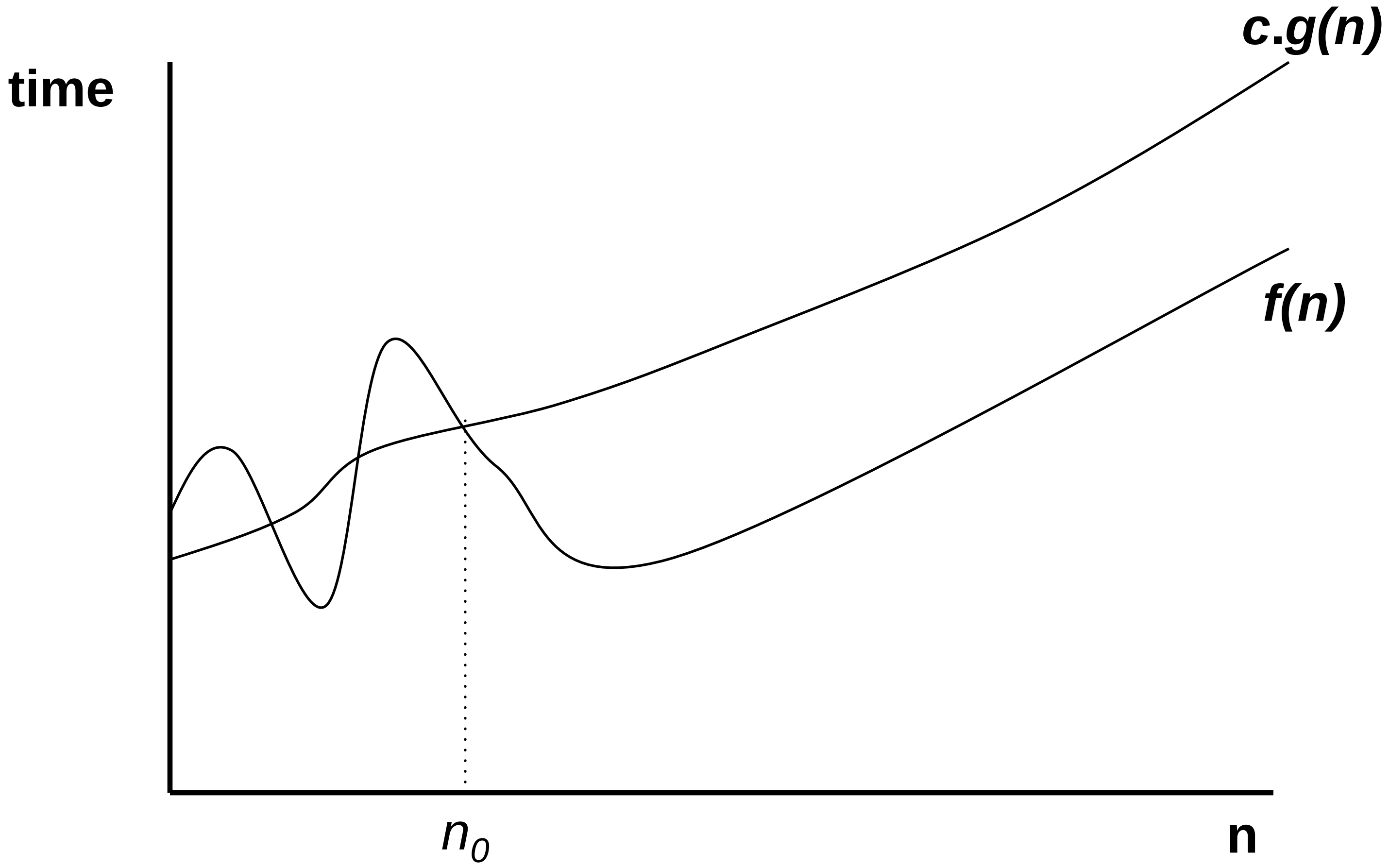
Asymptotic Analysis

- 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
  - $O(g(n)) = \{ f(n)$: $\exists$ 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
  - $an^2 + bn + c \le (a + b + c)n^2 + (a + b + c)n + (a + b + c)$

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

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

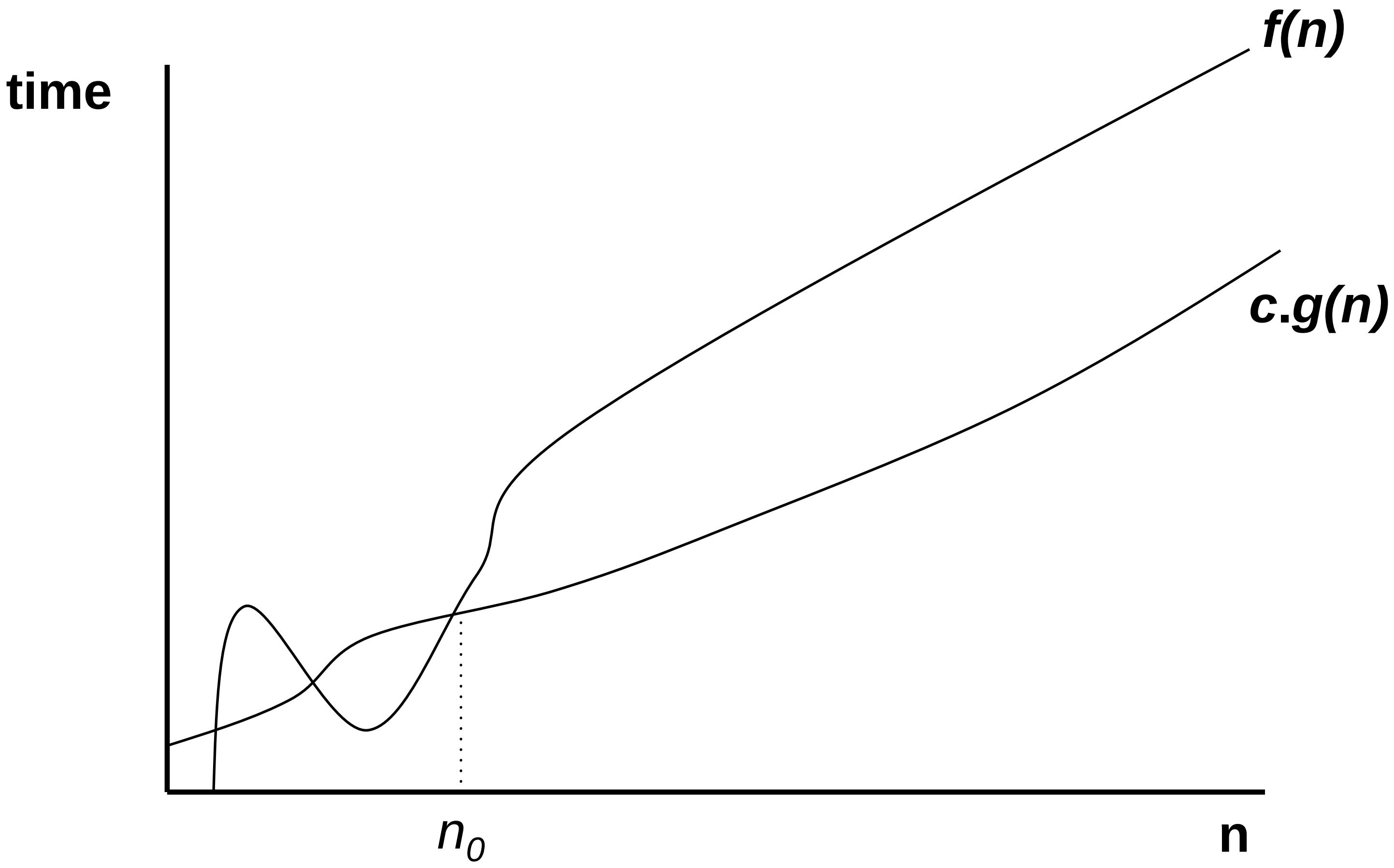
    $an^2 + bn + c \le c' n^2$ for $n \ge 1$

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

Lower Bound Notation



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

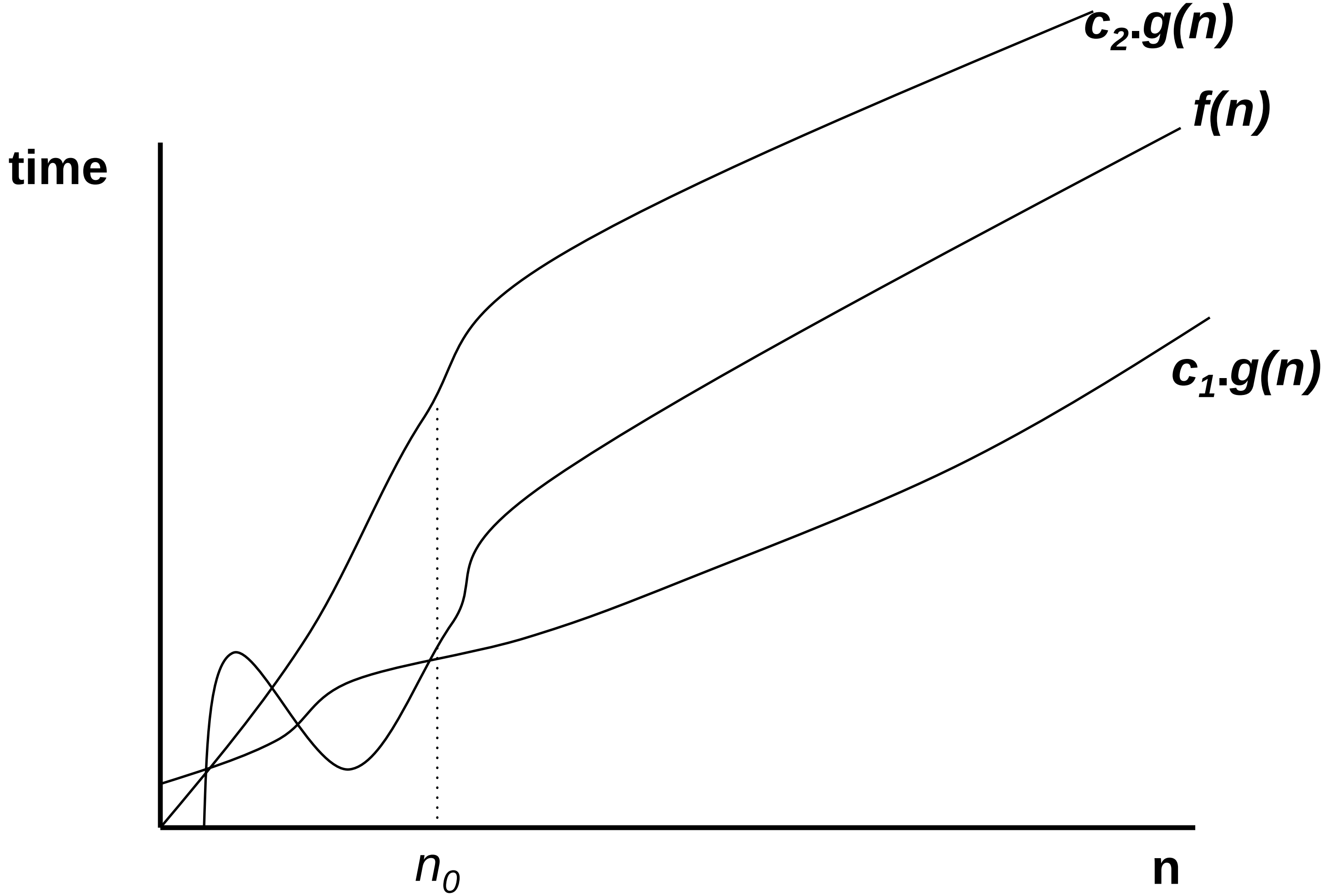
Asymptotic Tight Bound

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

$$0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)\ \forall\ n \geq n_0$$

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

Asymptotic Tight Bound



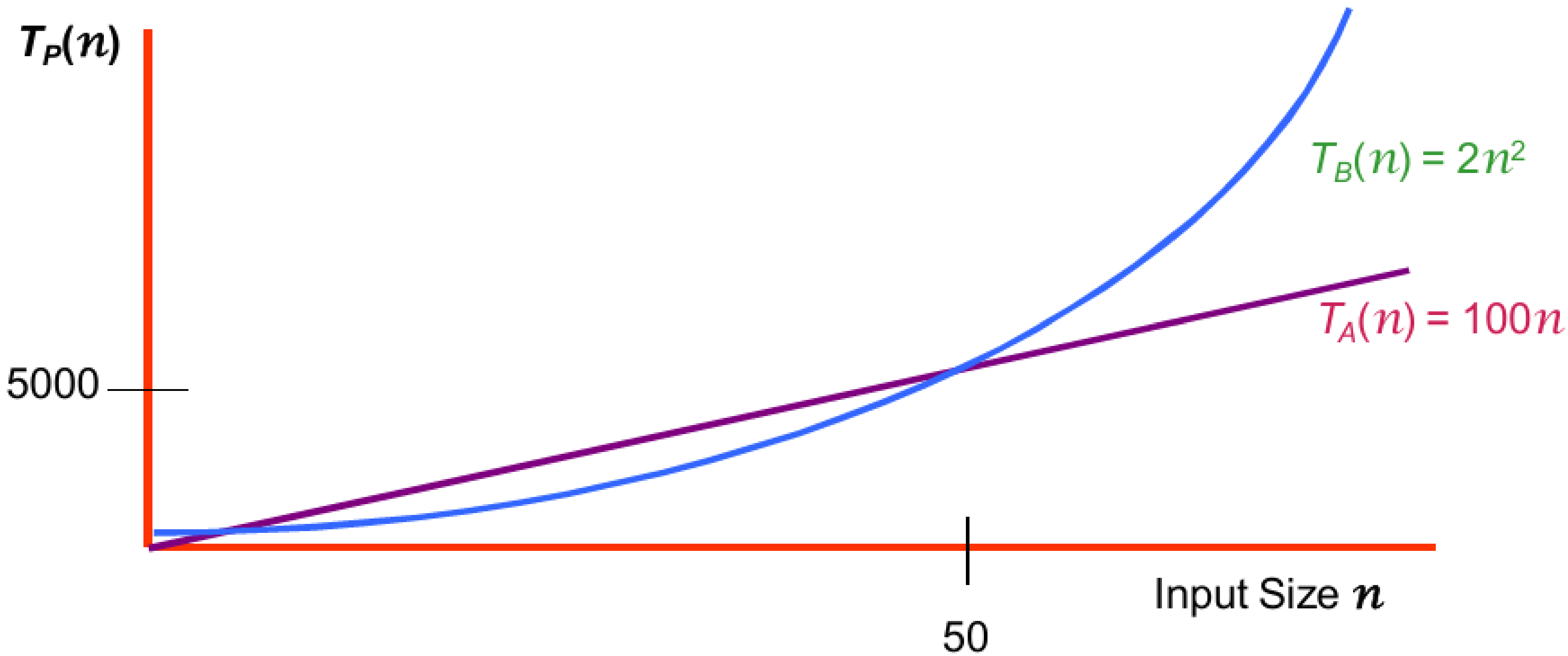
We say $g(n)$ is an *asymptotic tight bound* for $f(n)$

Practical Complexity

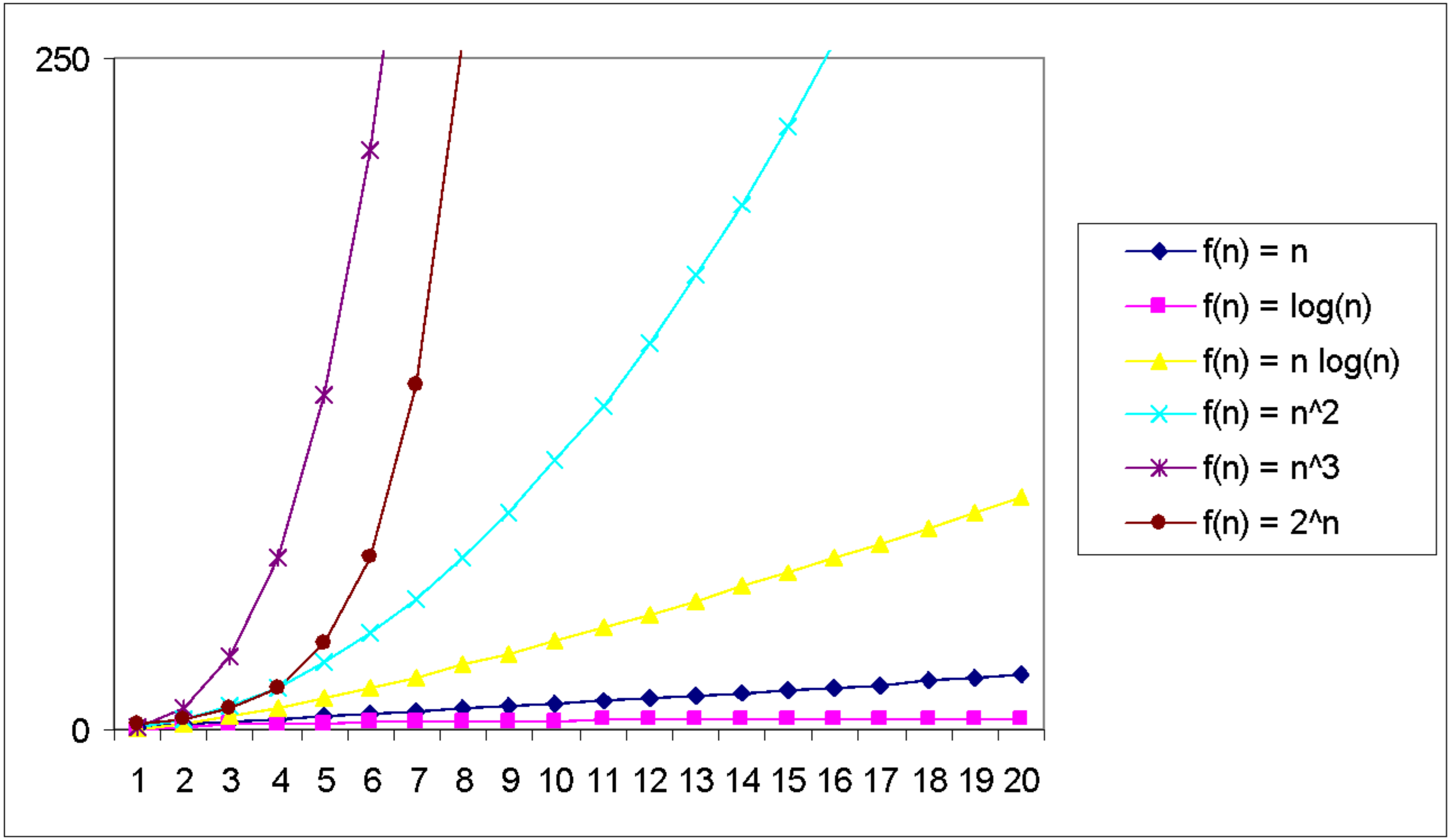
For large input sizes, constant terms are insignificant

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

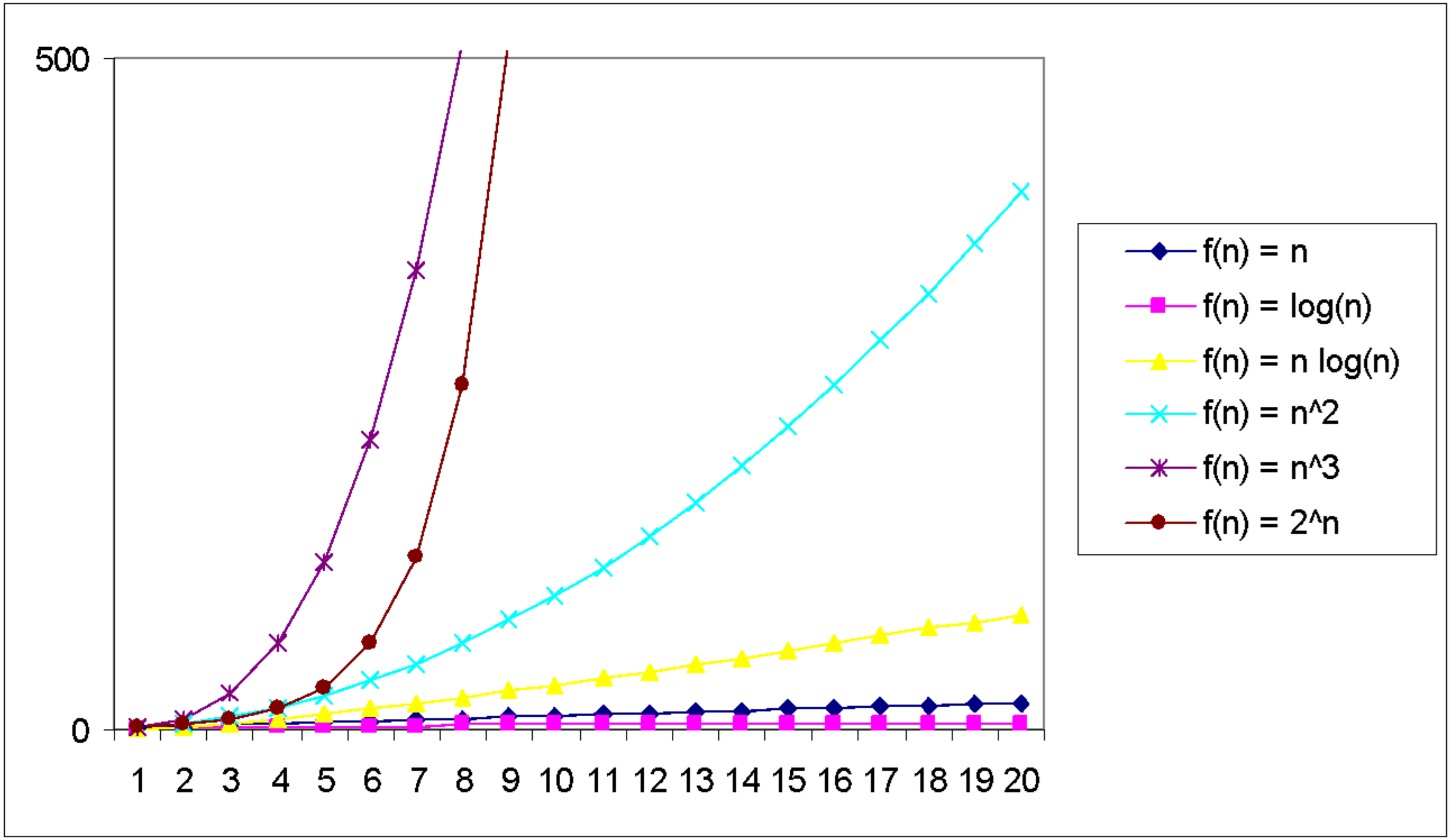
Program B with running time $T_B(n) = 2n^2$



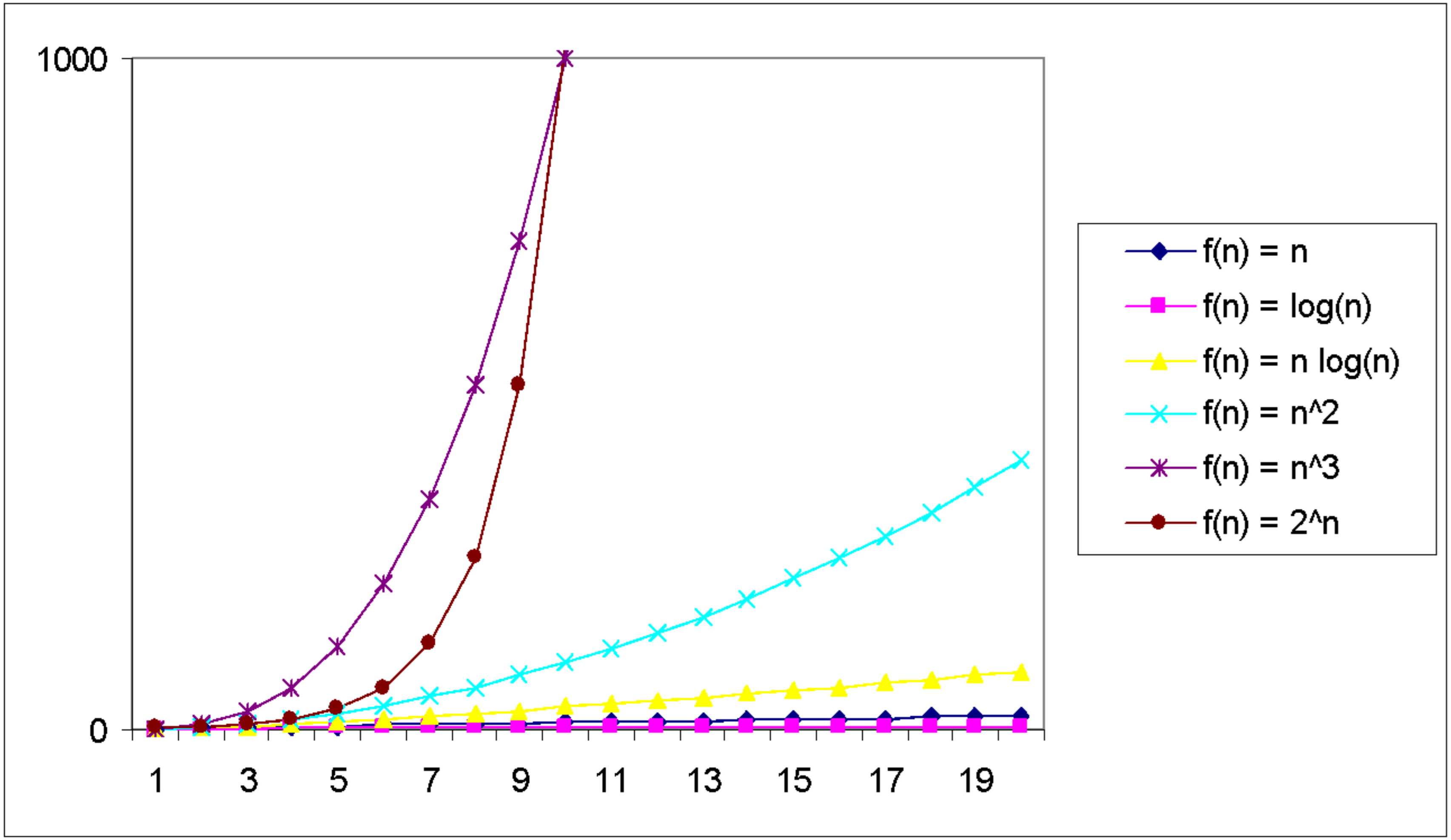
Practical Complexity



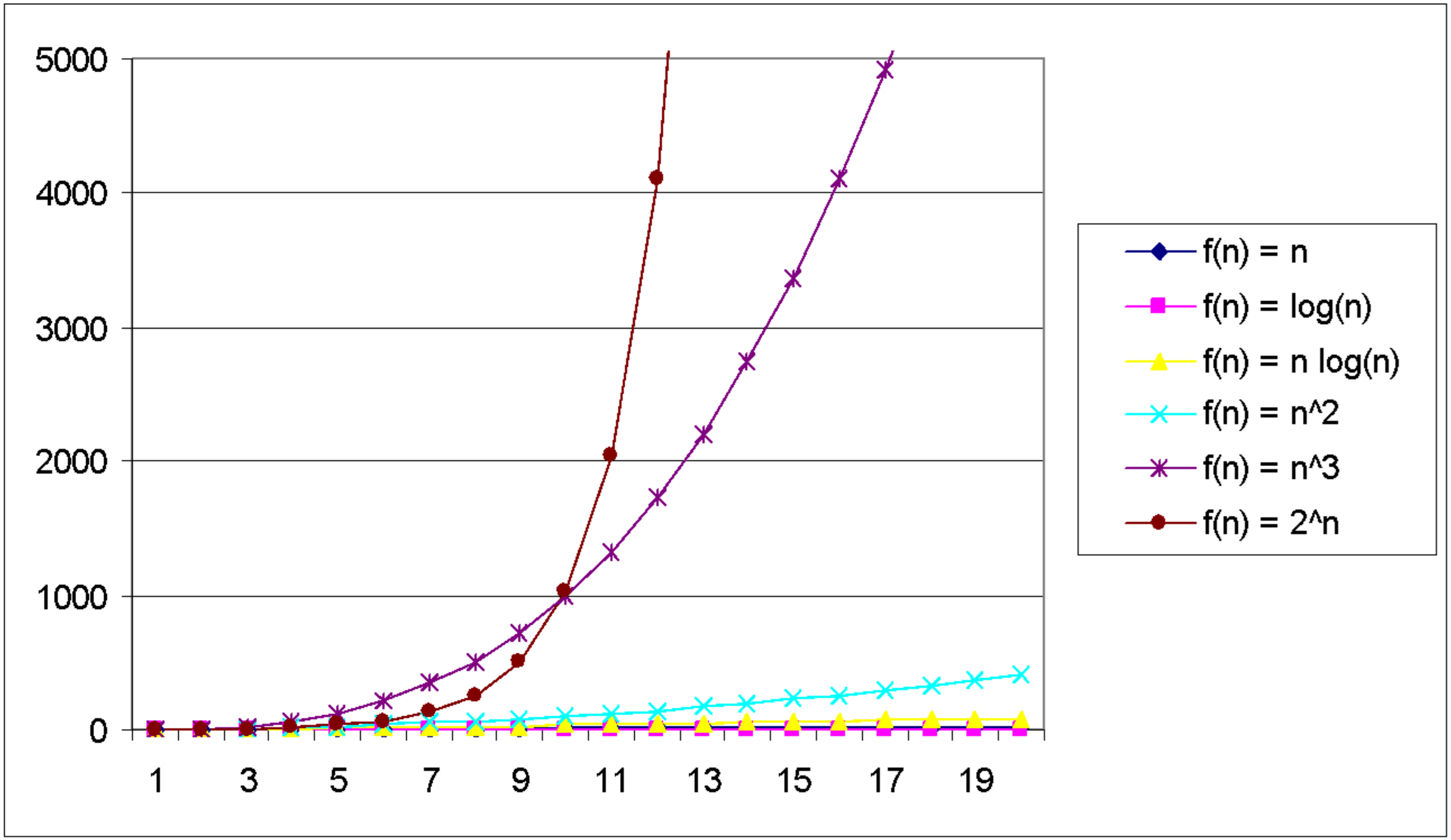
Practical Complexity



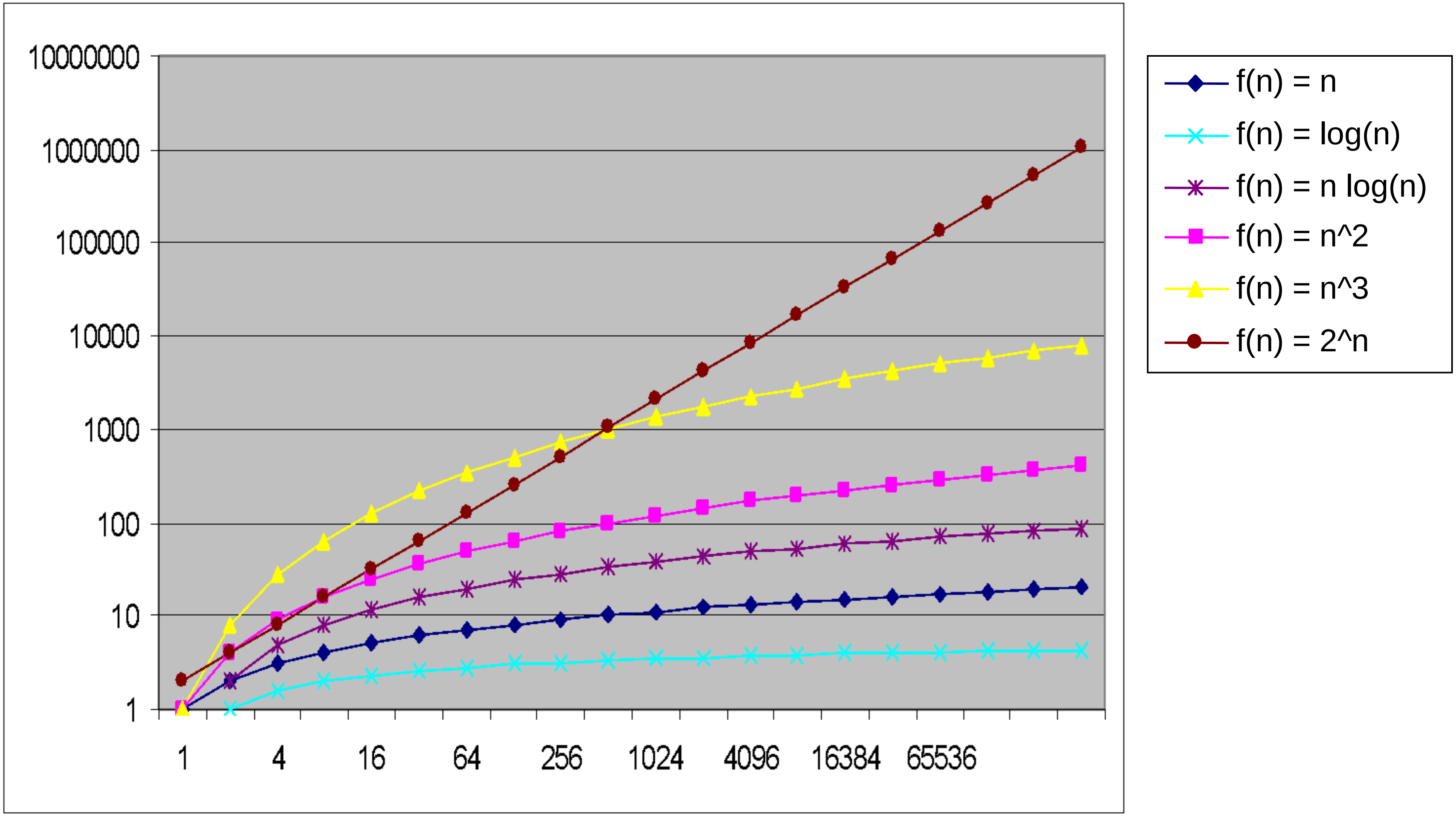
Practical Complexity



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 $=$

Other Asymptotic Notations

- Assume: $T(n) = 5n^3 + 4n + 1$, $g(n) = n^3$
  - $T(n)$ is $O(n^3)$
  - $T(n)$ is $\Omega(n^3)$
  - $T(n)$ is $\Theta(n^3)$
  - $T(n)$ is $O(n^7)$
  - $T(n)$ is $\Theta(n^2)$

Exact cost analysis

best and worst case analysis

Exact Cost Analysis: Example 1

- Consider Line 3. How many times the line 3 executes?
  - Best case: 0
  - Worst case: n
  - Average case:

```
1 for i in 1 to n:
2     if array[i]%3 == 0:
3         print(array[i])
```

$$\frac{1}{n}\sum_{1}^{n} i = \frac{1}{n}\frac{n(n-1)}{2} = \frac{n-1}{2}$$

Exact Cost Analysis: Example 1

The running time of this algorithm therefore belongs to both $\boldsymbol{\Omega(n)}$ and $\boldsymbol{O(n)}$. which means it is in $\boldsymbol{\Theta(n)}$

```
1 for i in 1 to n:
2     if array[i]%3 == 0:
3         print(array[i])
```

- Consider Line 3. How many times the line 3 executes?
  - Best case: 0
  - Worst case: n
  - Average case:

$$\frac{1}{n}\sum_{1}^{n} i = \frac{1}{n}\frac{n(n-1)}{2} = \frac{n-1}{2}$$

Exact Cost Analysis: Example 2

```
1 for (i ← n; i>=0; i ← i-5) do {
2       if ( A[i] < 100) then
3             break;
4       for (k ← 1; k<=n; k ← k*2) do
5             print A[k];
6 }
```

What is the time complexity of the code? Derive the best and worst case run-time and express in O notation.

Line	Worst	Best
1		
2		
3		
4		
5		
Asymptotic		

Exact Cost Analysis: Example 2

```
1  for (i ← n; i>=0; i ← i-5) do {
2       if ( A[i] < 100) then
3            break;
4       for (k ← 1; k<=n; k ← k*2) do
5            print A[k];
6  }
```

Line	Worst	Best
1	$c_1.(\frac{n}{5}+1)$	$c_1.1$
2	$c_2.\frac{n}{5}$	$c_2.1$
3	$c_3.0$	$c_3.1$
4	$c_4.\frac{n}{5}*(\log_2 n+1)$	$c_4.0$
5	$c_5.\frac{n}{5}*\log_2 n$	$c_5.0$
Asymptotic	$O(n\, log_2 n)$	$O(1)$

Exact Cost Analysis: Example 2

```
1 for (i ← n; i>=0; i ← i-5) do {
2     if ( A[i] < 100) then
3         break;
4     for (k ← 1; k<=n; k ← k*2) do
5         print A[k];
6 }
```

- Observe Line 1
  - value of i: $n, n-5, n-10, \ldots$ until less than 0
  - therefore, runs $\frac{n}{5}+1$ times

Exact Cost Analysis: Example 2

```
1 for (i ← n; i>=0; i ← i-5) do {
2     if ( A[i] < 100) then
3         break;
4     for (k ← 1; k<=n; k ← k*2) do
5         print A[k];
6 }
```

- Observe Line 4
  - value of i: $1, 2, 4, 8, \ldots, \boldsymbol{n}$
  - value of i: $2^0, 2^1, 2^2, 2^3, \ldots, \boldsymbol{2^x}$
  - $2^x = n$
  - $x = \log_2 n$
  - Therefore, inner statements of loop in line 4 runs $\log_2 n + 1 + 1$ times

Exact Cost Analysis: Example 2

```
for (i ← n; i>=0; i ← i-5) do {
    if ( A[i] < 100) then
        break;
    for (k ← 1; k<=n; k ← k*2) do
        print A[k];
}
```

- Best case: $\Omega(1)$
- Worst case: $O(n \log_2 n)$ or $O(nlgn)$

The running time of this algorithm therefore belongs to both $\boldsymbol{\Omega(1)}$ and $\boldsymbol{O(nlgn)}$

Exact Cost Analysis: Example 3

$$\log_4 n = \log_{2^2} n = \frac{1}{2} * \log_2 n$$

```
1  for (i ← n; i>=0; i ← i-3) do {
2      if ( A[i] < 100) then
3          break;
4      for (k ← n; k>=1; k ← k/4) do
5          print A[k];
6  }
```

Derive the running-time equations and express in "O" notation

Line	Worst	Best
1		
2		
3		
4		
5		
Asymptotic		

Exact Cost Analysis: Example 3

$$\log_4 n = \log_{2^2} n = \frac{1}{2} * \log_2 n$$

```
1  for (i ← n; i>=0; i ← i-3) do {
2      if ( A[i] < 100) then
3          break;
4      for (k ← n; k>=1; k ← k/4) do
5          print A[k];
6  }
```

Line	Worst	Best
1	n/3+1	
2	n/3	
3	0	
4	$\frac{n}{3}.(\log_4 n + 1)$	
5	$\frac{n}{3}.\log_4 n$	
Asymptotic	$O(n\, log_2 n)$	$O(1)$

Exact Cost Analysis: Example 3

```
1 for (i ← n; i>=0; i ← i-3) do {
2     if ( A[i] < 100) then
3         break;
4     for (k ← n; k>=1; k ← k/4) do
5         print A[k];
6 }
```

- Observe Line 4
  - value of i: $\frac{n}{4^0}, \frac{n}{4}, \frac{n}{4^2}, \frac{n}{4^3}, \ldots, 1(\frac{n}{4^x})$
  - $\frac{n}{4^x} = 1$
  - $x = \log_4 n$
  - Therefore, inner statements of loop in line 4 runs $\log_4 n + 1 + 1$ times

Exact Cost Analysis: Example 3

```
for (i ← n; i>=0; i ← i-3) do {
    if ( A[i] < 100) then
        break;
    for (k ← n; k>=1; k ← k/4) do
        print A[k];
}
```

- Best case: $\Omega(1)$
- Worst case: $O(n \log_2 n)$ or $O(nlgn)$

The running time of this algorithm therefore belongs to both $\boldsymbol{\Omega(1)}$ and $\boldsymbol{O(nlgn)}$

Exact Cost Analysis: Example 4

```
1  for (i=0; i<=n; i++){
2      for (j=2; j<=i; j=j++){
3          print(j)
4      }
5  }
```

Derive the running-time equations and express in "O" notation

Line	Worst	Best
1		
2		
3		
Asymptotic		

Exact Cost Analysis: Example 5

```
1  c = 0;
2  for (i=n/2; i<=n; i++){
3      for (j=2; j<=n; j=j*2){
4          c += k + n/2;
5      }
6  }
7  for (i=0; i<=m; i++){
8      for (j=2; j<=n; j=j+=2){
9          c += k + n/2;
10     }
11 }
```

Derive the running-time equations and express in "O" notation

Line	Worst	Best
1		
2		
3		
4		
5		
Asym ptotic		

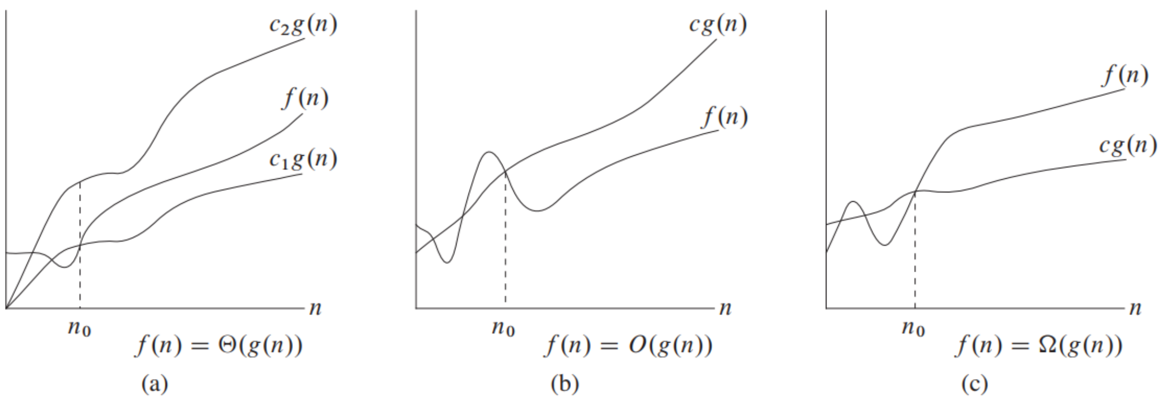
Practice

Question Patterns

- Derive the best and worst-case running-time equations and express them in O notation.
- Derive the exact cost equation and express it in O notation
- Provide best and worst-case examples

Quick Evaluation 1

- Which picture shows the **asymptotic tight bound**?
- Show that $f(n) = an^3 + bn^2 + cn + d$ is $O(n^3)$
- Show that $f(n) = an^2 + bn + c$ is not $O(n)$
- Show that $f(n) = an^2 + bn + c$ is $O(n^3)$
- Show that $f(n) = an^2 + bn + c$ is $\Theta(n^2)$



Quick Evaluation 2

- What is the time complexity of the code?
- Derive the **exact cost equation** and express in *O* notation

```
int i, j, k = 0;
for (i=n/2; i<=n; i++){
    for (j=2; j<=n; j=j*2){
        k = k + n/2;
    }
}
```

Quick Evaluation 3

- What is the time complexity of the code?
- Derive the **exact cost equation** and express in *O* notation

```
c = 0;
for (k=0;k<10;k=k*2){
    for (i=n/2; i<=n; i++){
        for (j=2; j<=n; j=j*2){
            c += k + n/2;
        }
    }
}
```

Quick Evaluation 4

- What is the time complexity of the code?
- Derive the **exact cost equation** and express in *O* notation

```
for (i=n/2; i<=n; i++){
    for (j=2; j<=n; j=j*2){
        k = k + n/2;
    }
}
for (i ← n; i>=0; i=i-5) do {
    if ( A[i] < 100) then
        break;
    for (k ← 1; k<=n; k=k*2) do
        print A[k];
}
for (i ← n; i>=0; i=i-3) do {
    if ( A[i] < 100) then
        break;
    for (k ← n; k>=1; k=k/4) do
        print A[k];
}
```

Resources

- https://www.cs.auckland.ac.nz/courses/compsci220s1t/lectures/lecturenotes/GG-lectures/BigOhexamples.pdf
- http://www.cs.utsa.edu/~bylander/cs3233/big-oh.pdf
- https://youtu.be/FEnwM-iDb2g
- https://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-processing-an-unsorted-array/11227902#11227902