Review:
Complexity Analysis
(Some topics covered in earlier course: Data Structure and Algorithms)

Important Properties of Algorithms

- Correct
  - *always* returns the desired output for all legal instances of the problem.
- Efficient
  - Measured in terms of time or space
  - Time tends to be more important
  - The running time analysis allows us to improve our algorithms

Analysis of Algorithms

- Why analyze algorithms?
  - evaluate algorithm performance
  - compare different algorithms
- Analyze what about them?
  - running time, memory usage
  - worst-case and “typical” case
- Analysis of algorithms compare algorithms and not programs

What is Running Time? Actual Time or Number of Steps?

- Actual time (seconds, minutes, hours) different for different computers
- Actual time different for different Operating System, Programming Language
- But, number of steps (**addition**, **comparison**, **assignment**, etc…) is same for a program
- So, **number of steps** is the running time

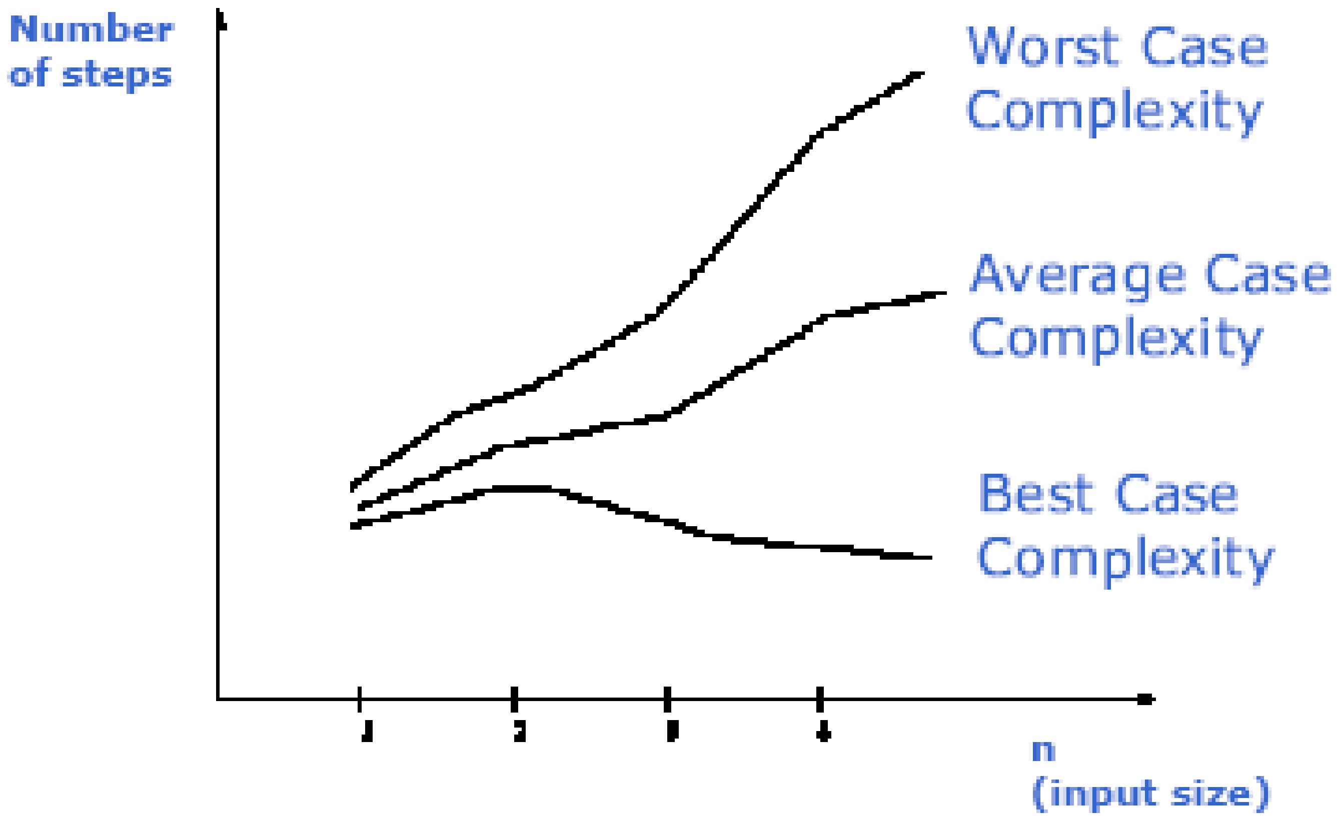
Analysis of Algorithms (cont.)

- Efficiency of the algorithm is always based on the **input size**
  - Input size usually represented by ***n***

Algorithm Complexity

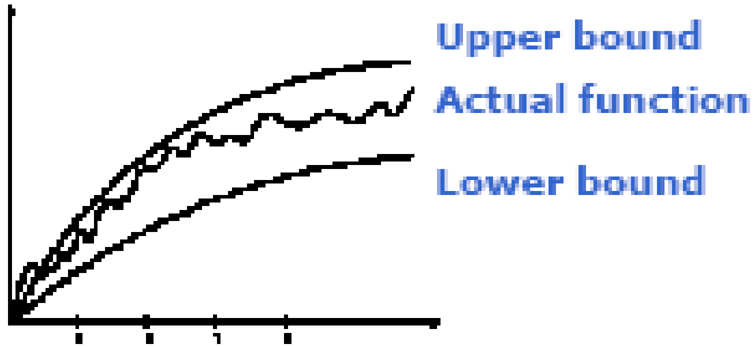
- Worst Case Complexity:
  - the function defined by the *maximum* number of steps taken on any instance of size *n*
- Best Case Complexity:
  - the function defined by the *minimum* number of steps taken on any instance of size *n*
- Average Case Complexity:
  - the function defined by the *average* number of steps taken on any instance of size *n*

Best, Worst, and Average Case Complexity

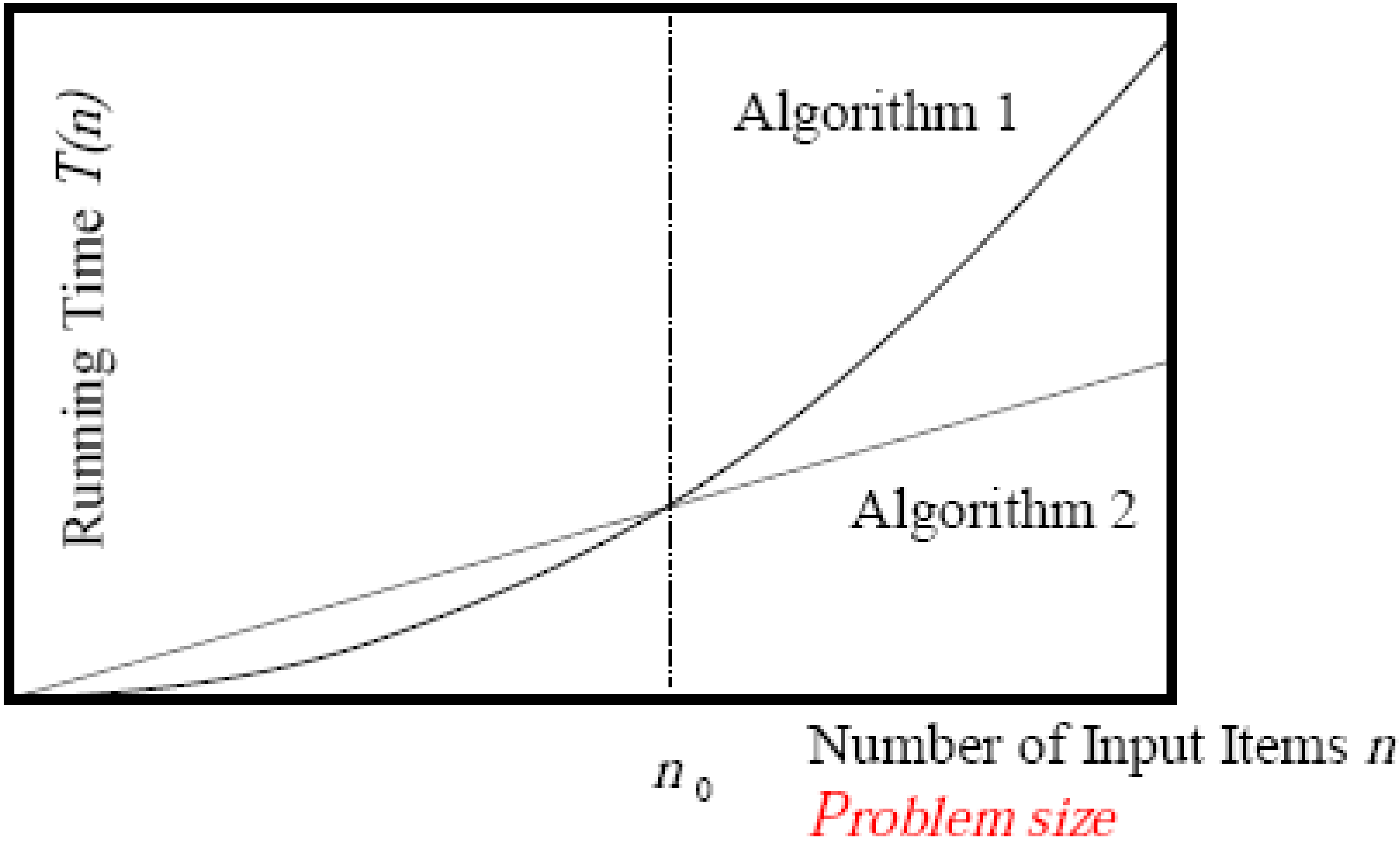


Doing the Analysis

- It's hard to estimate the running time exactly
  - Best case depends on the input
  - Average case is difficult to compute
  - So we usually focus on worst case analysis
    - Easier to compute
    - Usually close to the actual running time
- Strategy: try to find upper and lower bounds of the worst case function.



Analysis of Running Time



Comparing Algorithms

- Establish a relative order among different algorithms, in terms of their relative rates of growth.
- The rates of growth are expressed as functions, which are generally in terms of the number of inputs n.

Asymptotic Analysis

- Asymptotic analysis of an algorithm describes the relative efficiency of an algorithm as n get very large.
- When you're dealing with small input size, most of algorithms will do good
- When the input size is very large things change
- Eg: For very large *n*, algorithm 1 grows faster than algorithm 2.

An simple comparison

- Let's assume that you have 3 algorithms to sort a list
  - $f(n) = n \log_2 n$
  - $g(n) = n^2$
  - $h(n) = n^3$
- Let's also assume that each step takes 1 microsecond (10^{-6})

n	*n log n*	*n^2*	*n^3*
10	33.2	100	1000
100	664	10000	1seg
1000	9966	1seg	16min
100000	1.7s	2.8 hours	31.7 years

But there can be more than one term

- $f(n) = n^2 + 2n + 5\mathbf{log}\ n$

  $f(n) = n^3 + n^2 + n\mathbf{log}n + 1000$
- Example:

```
for i = 1 to n do
    for j = 1 to i do
        do some work
```

How many **work** here?

$1+2+3+\ldots+n = n(n+1)/2 = n^2/2 + n/2$

- So, we go for **approximation** to the **dominating term** (the **highest term without constant**)

Who dominates? When?

FIGURE 2.1 The growth rate of all terms of function $f(n) = n^2 + 100n + \log_{10}n + 1{,}000$.

n	$f(n)$	n^2		$100n$		$\log_{10}n$		1,000	
	Value	Value	%	Value	%	Value	%	Value	%
1	1,101	1	0.1	100	9.1	0	0.0	1,000	90.83
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	10,000	47.6	10,000	47.6	2	0.001	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

Theoretical Framework

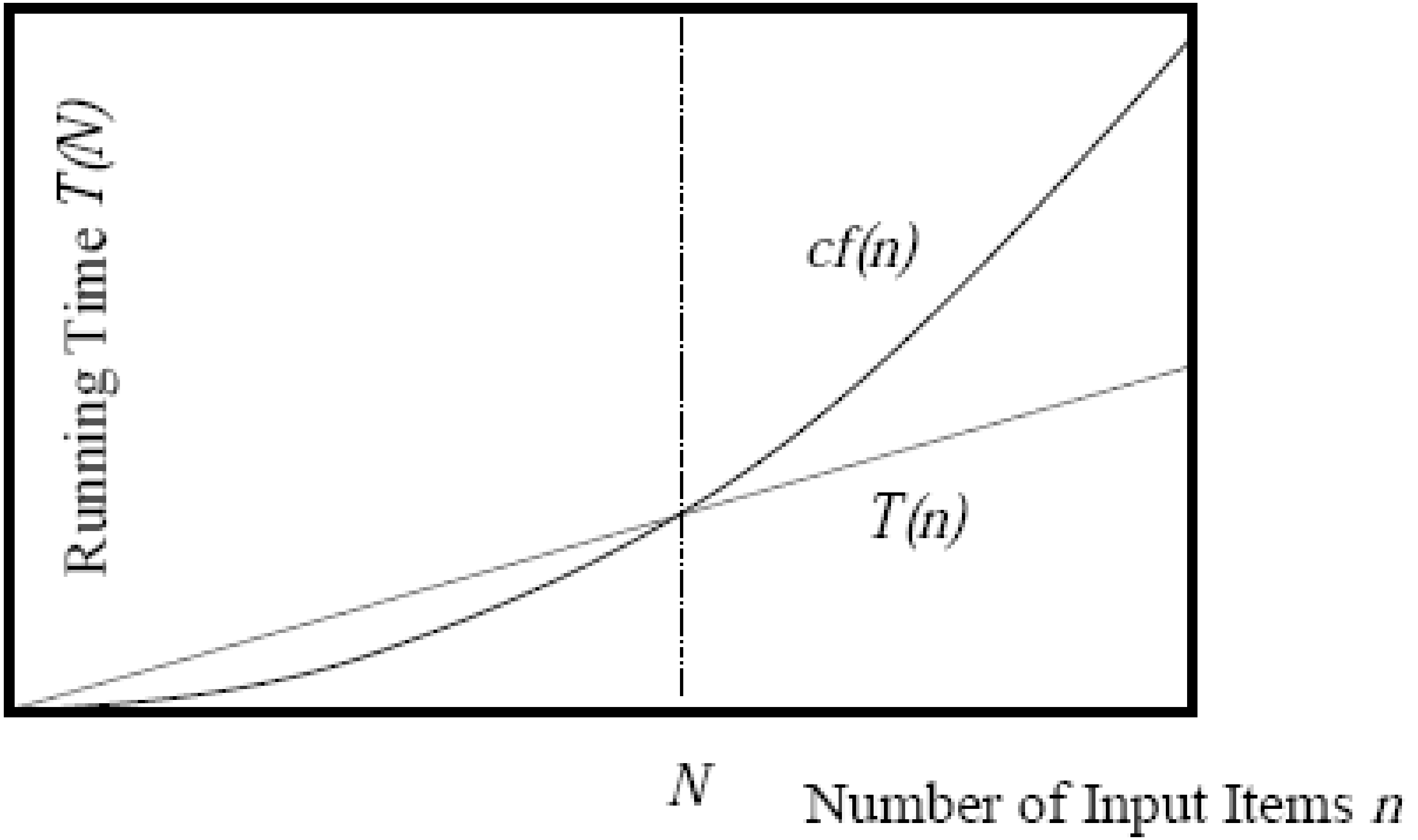
- "Big-Oh"
  - Definition:

$T(n) = O(f(n))$ if there are positive constants c and N such that $T(n) \leq c\,f(n)$ when $n \geq N$

  - This says that function *T(n)* grows at a rate no faster than *f(n);* thus *f(n)* is an upper bound on *T(n).*

Big-Oh Upper Bound

$T(n) = O(f(n))$ if there are positive constants c and N such that $T(n) \leq c\,f(n)$ when $n \geq N$



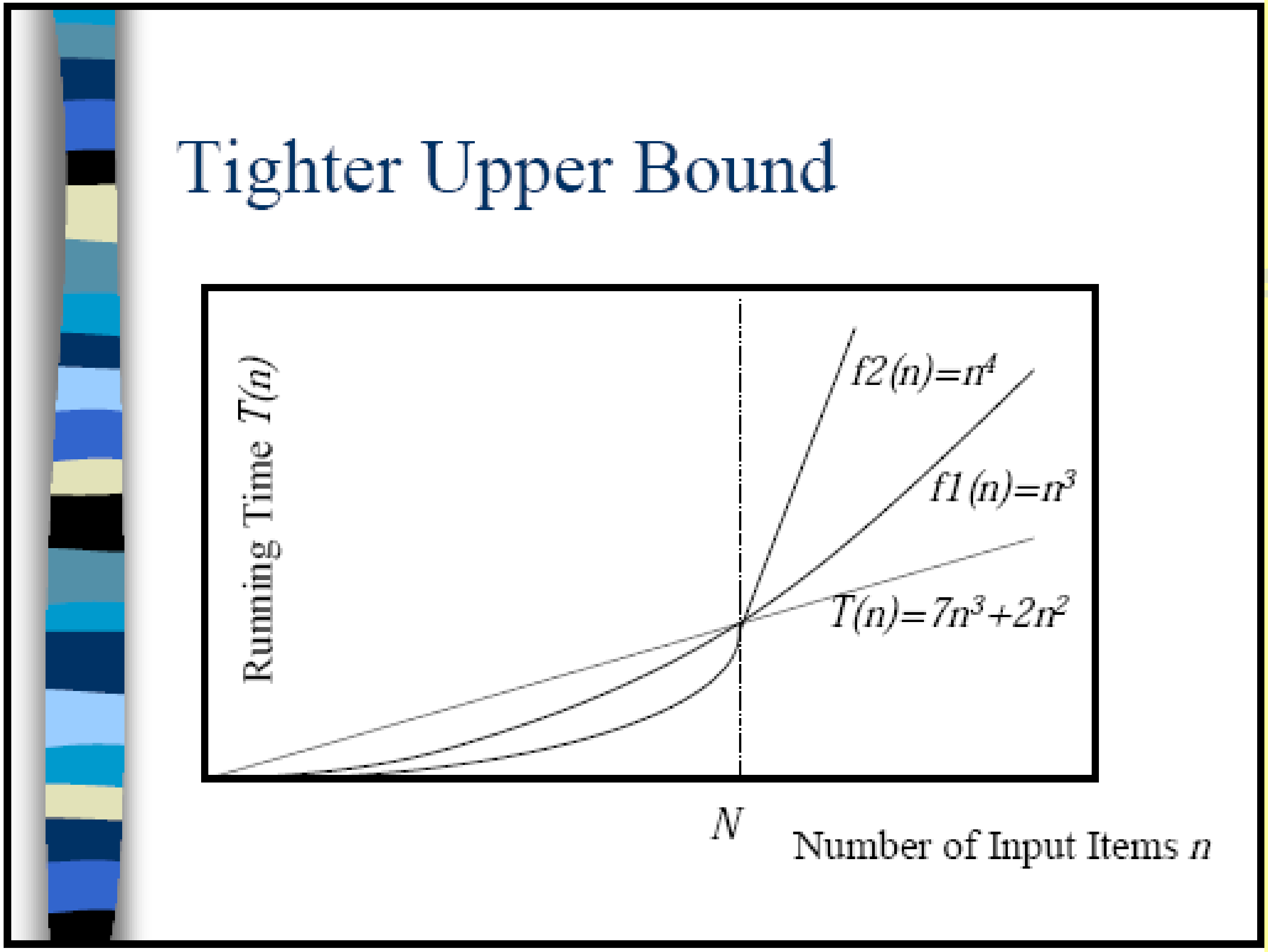
An Example

- Prove that $7n^3 + 2n^2 = O(n^3)$
  - Since $7n^3 + 2n^2 < 7n^3 + 2n^3 = 9n^3 \,(\,for\ n \ge 1)$
  - Then $7n^3 + 2n^2 = O(n^3)$ with $c = 9$ and $N = 1$

- Similarly, we can prove that $7n^3 + 2n^2 = O(n^4)$

  The first bound is tighter.

- **We take the tighter/closest bound (which is the best answer)**



So, we take T(n) = O(n^3)

A Simple Example

$$\sum_{i=1}^{n} i^2$$

Time Units to Compute

- 1 for the assignment.
- 1 assignment, $n+1$ tests, and n increments.
- n loops of 3 units for an assignment, an addition, and two multiplications.
- 1 for the return statement.

Total: $1+(1+n+1+n)+3n+1$
$= 5n+4 = O(n)$

```
int sum (int n)
{
   int partial_sum = 0;
   int i;
   for (i = 1; i <= n; i++)
      partial_sum = partial_sum + (i * i );
   return partial_sum;
}
```

```
   i = 1;
   if (i <= n) {
      partial_sum = partial_sum + (i * i );
      i++; /* i = i + 1 */
   }
```

Analysis too complex

General Rules

- Loops
  - The running time of a “for” loop is at most the running time of the statements inside the “for” loop (including tests) times the number of iterations.

```
for (i = 1; i <= n; i++) {
      sum = sum + i;
}
```

- The above example is *O(n)*.

General Rules (cont.)

- Nested loops
  - The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= m; j++) {
        sum = sum + i + j;
    }
}
```

$3mn = O(mn)$

- The above example is $O(mn)$.

General Rules (cont.)
A Question:

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= m; j++) {
        for (k = 1; k <= p; k++) {
            sum = sum + i + j + k;
        }
    }
}
```

- $4pmn = O(pmn)$.

General Rules (cont.)

- Consecutive statements
  – These just add, and the maximum is the one that counts.

```
for (i = 1; i <= n; i++) {
      sum = sum + i;
}
for (i = 1; i <= n; i++) {
      for (j = 1; j <= n; j++) {
            sum = sum + i + j;
      }
}
```

← $O(n)$

← $O(n^2)$

- The above example is $O(n^2+n) = O(n^2)$.

General Rules (cont.)
A Question:

```
for (i = 1; i <= n; i++) {
      for (j = 1; j <= n; j++) {
            sum = sum + i + j;
      }
}
sum = sum / n;
for (i = 1; i <= n; i++) {
      sum = sum + i;
}
for (j = 1;  j<= n; j++) {
      sum = sum + j*j;
}
```

- $n^2+1+n+n=O(n^2+2n+1) = O(n^2)$.

General Rules (Cont.)

- If (test) s1 else s2
  - The running time is never more than the running time of the test plus the larger of the running times of s1 and s2.

```
if (test == 1) {
    for (i = 1; i <= n; i++) {
        sum = sum + i;
    }
}
else for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
            sum = sum + i + j;
        }
    }
```

- The running time = 1 + $max(n,n^2) = O(n^2)$.

General Rules (Cont.)

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++) {
        for (k = 1; k <= n; k++) {
            sum = sum + i + j + k;
        }
    }
}

if (test == 1) {
   for (i = 1; i <= n; i++) {
      for (j = 1; j <= n; j++) {
         sum = sum + i;
      }
   }
}
else for (i = 1; i <= n; i++) {
         sum = sum + i + j;
      }
```

- The running time $= O(n^3) + O(n^2)$ $=O(n^3)$.

General Rules (Cont.)

- Recursion:
  - Analyze from the inside (or deepest part) first and work outwards. If there are function calls, these must be analyzed first. This even works for recursive functions:

```
long factorial (int n) {
    if (n <= 1)
        return 1;
   else
      return n * factorial(n-1);
}
```

Time Units to Compute

----

1 for the test.

1 for the multiplication statement.

What about the function call?

- The running time of *factorial(n)* = *T(n)* = 2+*T(n-1)* = 4+*T(n-2)* = 6+*T(n-3)* = … = 2*n* = *O(n)*.

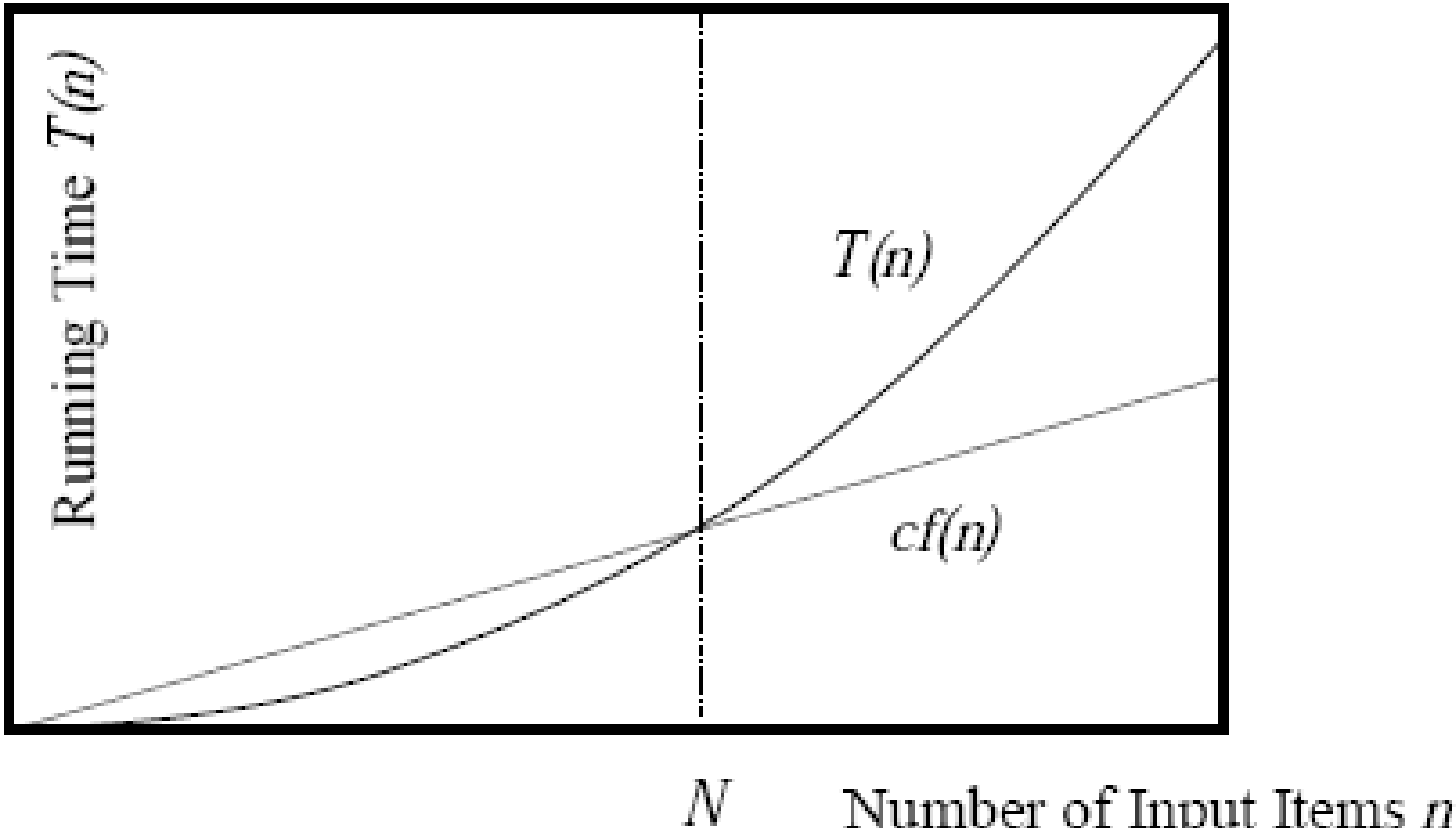
Big-Omega

Definition:

$T(n) = \Omega(f(n))$ if there are positive constants c and N such that $T(n) \geq c\, f(n)$ when $n \geq N$

– This says that function *T(n)* grows at a rate no slower than *f(n);* thus *f(n)* is a lower bound on *T(n)*.

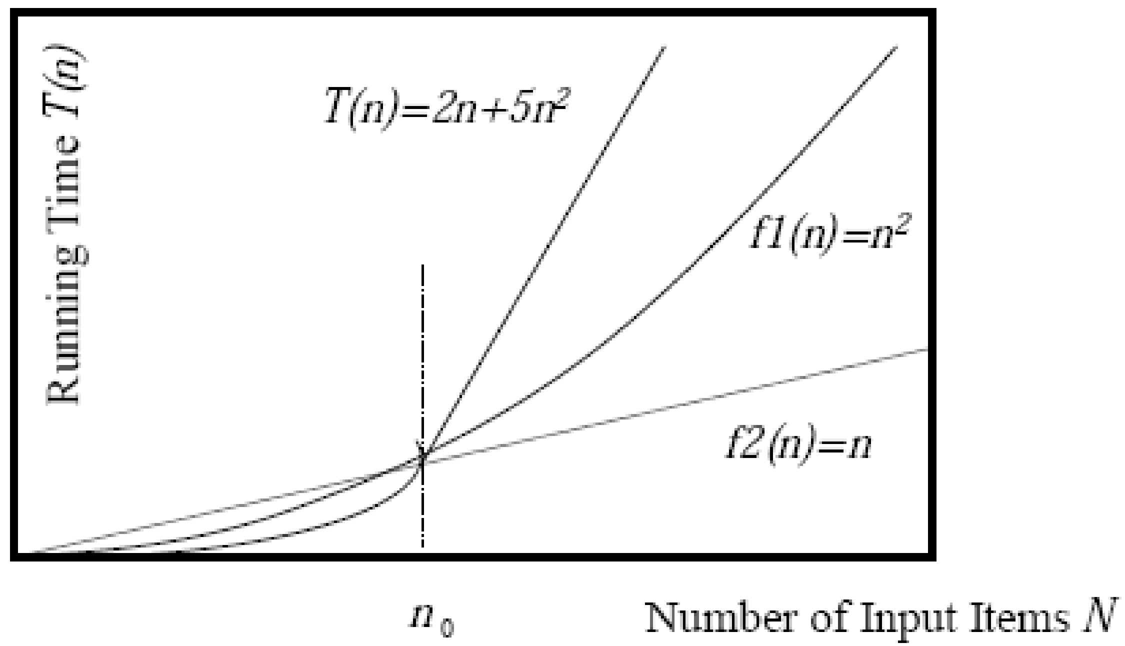
Big Omega Lower Bound



Big Omega Example

- Prove that $2n+5n^2=\Omega(n^2)$
  - Since $2n+5n^2>5n^2>1n^2 (\,for\ n\geq 1)$
  - Then $2n+5n^2=\Omega(n^2)\ with\ c=1\ and\ N=1$
- Similarly, we can prove that $2n+5n^2=\Omega(n)$
- The above bound is tighter.

Tighter Lower Bound



So, we take T(*n*) = Ω(*n*^{2})

Big Theta

- Definition:

$$T(n) = \Theta(f(n)) \text{ if and only if}$$

$$T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n))$$

- This says that function *T(n)* grows at the same rate as *f(n)*.

- Put another way:

$T(n) = \Theta(f(n))$ if there are positive constants c, d, and N such that $cf(n) \le T(n) \le d\, f(n)$ when $n \ge N$

Big Theta Example

- If two functions *f* and *g* are proportional then $f(N) = \Theta(g(n))$

Constant

- Since $log_A n = log_B n / log_B A$
  - Then: $\log_A(n) = \Theta(\log_B n)$
  - The base of the log is irrelevant.

little-oh

- Definition:

$$T(n) = o(f(n)) \text{ if and only if}$$
$$T(n) = O(f(n)) \text{ and } T(n) \neq \Theta(f(n))$$

– This says that function *T(n)* grows at a rate strictly less than *f(n)*.

- Example:
  - $n^2+3n+100 = o(n^3)$
  - $n \log n = o(n^2)$

A Hierarchy of Growth Rates

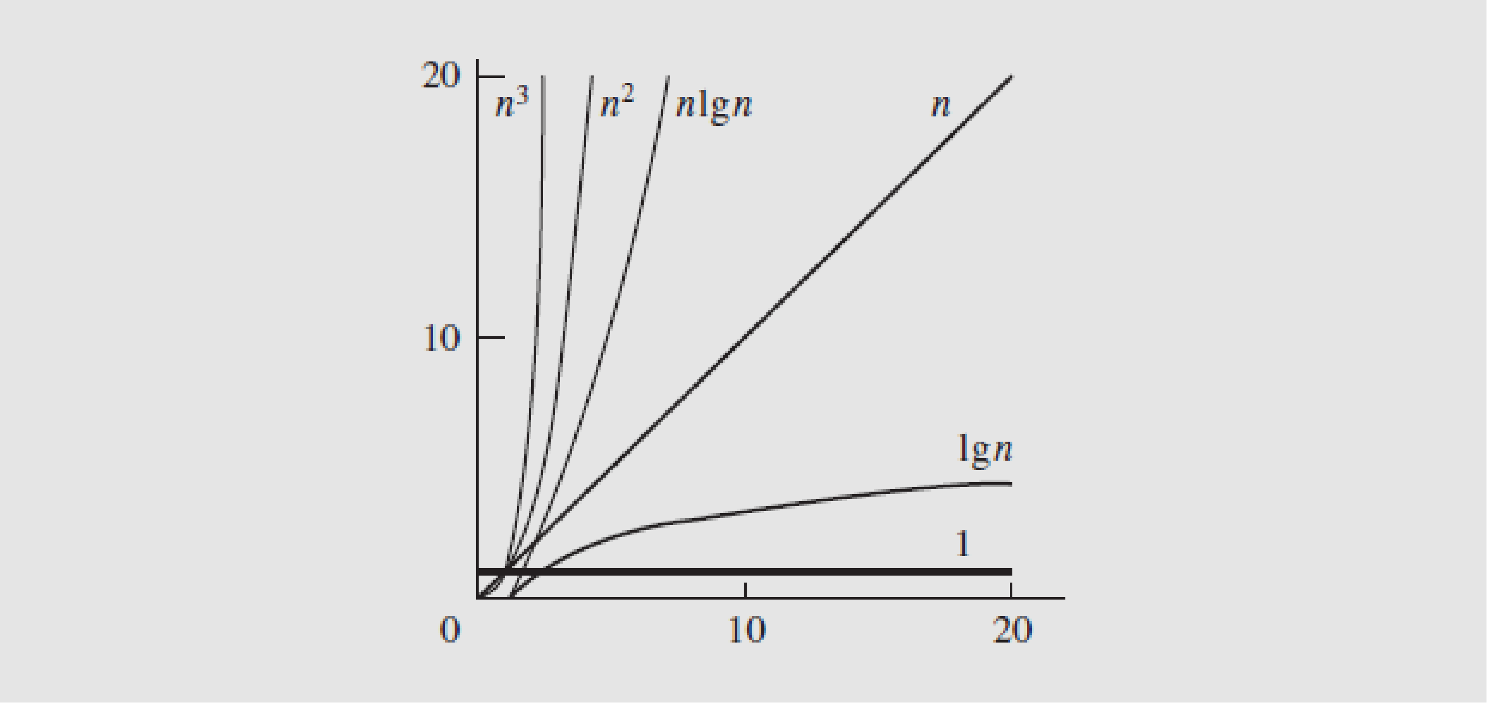
$$c < \log n < \log^2 n < \log^k n < n < n \log n <$$
$$n^2 < n^3 < 2^n < 3^n < n! < n^n$$

$\log^k n$ means log n * log n * log n …. k times

log n < n^c, for any c. For example, log n < $n^{0.2}$

Comparison of growth rates

Typical functions applied in big-O estimates.



Properties of O

1. If $f(n) = O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n) = O(h(n))$
2. If $f(n) = O(h(n))$ and $g(n)$ is $O(h(n))$, then $f(n)+g(n) = O(h(n))$
3. If $f(n) = O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n) = O(h(n))$
4. $an^k = O(n^k)$
5. $n^k = O(n^{k+j})$
6. $\log_a n = O(\log_b n)$

Proofs are easy, see the text book.

General Rules

If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, then

(a) $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$

(b) $T_1(n) * T_2(n) = O(f(n)) * O(g(n))$

Example: Algorithm A:

Step 1: Run algorithm A1 that takes $O(n^3)$ time

Step 2: Run algorithm A2 that takes $O(n^2)$ time

$T_A(n) = T_{A1}(n)+T_{A2}(n)= O(n^3)+O(n^2)$

$=\max(O(n^3), O(n^2)) = O(n^3)$

General Rules (cont.)

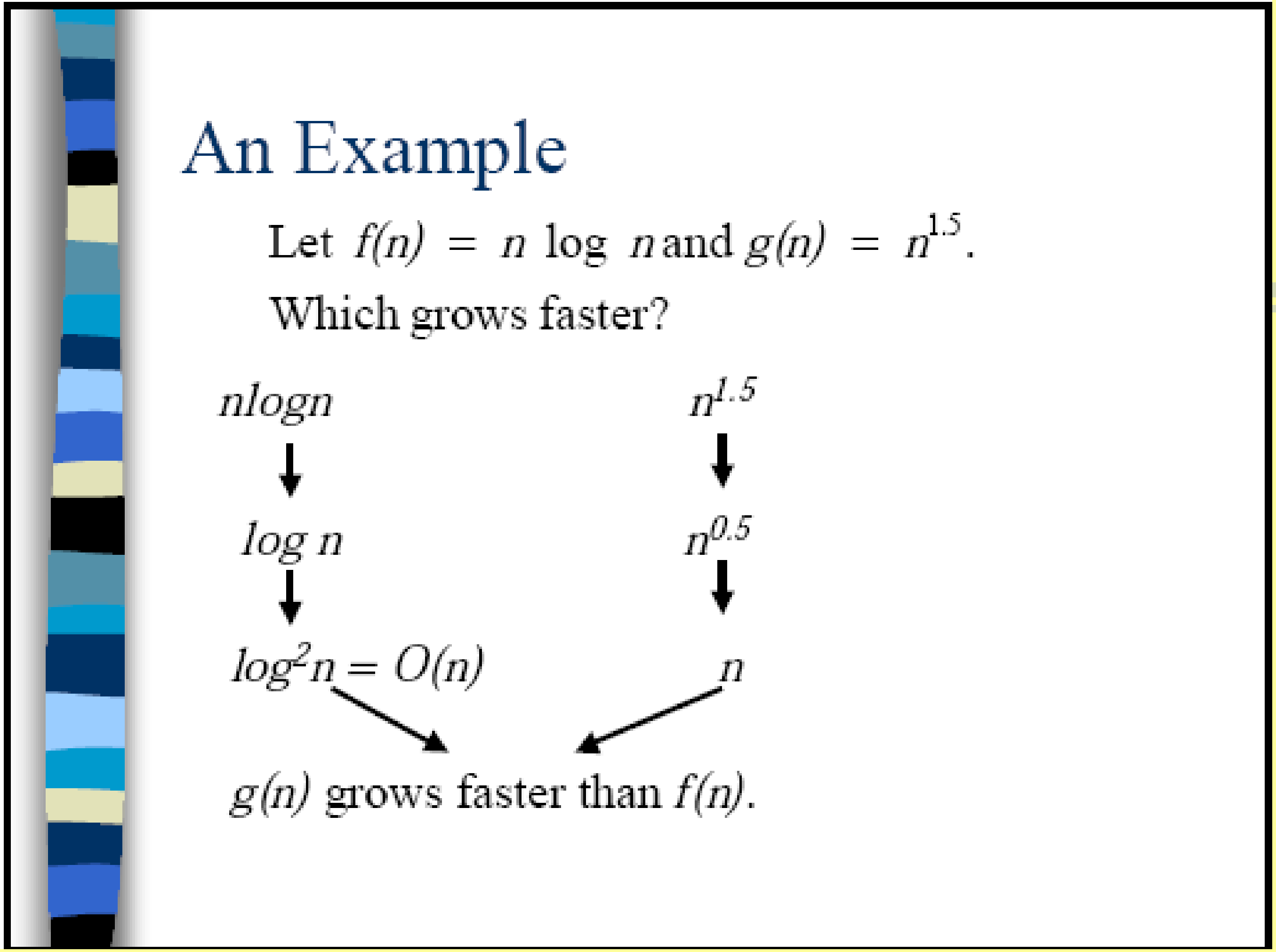
If $T(n)$ is a polynomial of degree k, then

$T(n) = \Theta(n^k)$

Example:

$T(n) = n^8 + 3n^5 + 4n^2 + 6 \qquad = \Theta(n^8)$

$\log^k(n) = O(n)$ for any constant k.



So, **$n \log n < n^{1.5}$**, and **$\log n < n^{0.5}$**.
Actually, we can prove similarly that **$\log n < n^c$** for any c

Example of Best/Average/Worst Case Complexity

- Program for searching an item in an array of n items

```
found = false
For i = 1 to n
     If item = A[i] then found = true
```

- What are the number of steps (mainly, comparison) required by this program?
  - Best case: 1, because the program finds at the beginning
  - Worst Case: n or n+1., because the program finds it at the end (so n comparisons) or does not find (n+1 comparison)
  - Average case: Not easy, needs probability. An easy way for this program is: (best + worst)/2 = (n+1)/2