Design and Analysis of Algorithms

Week 2: Fundamentals of Algorithm Analysis

Richard Kelley

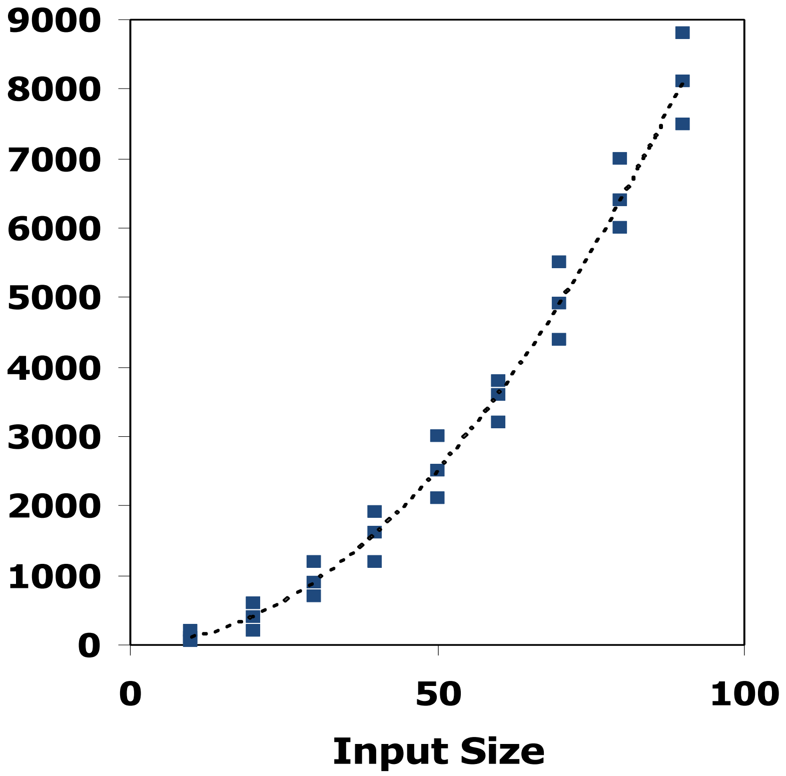
Analysis of Algorithms

- Issues:
  - correctness
  - time efficiency
  - space efficiency
  - optimality

- Approaches:
  - theoretical analysis
  - empirical analysis

Experimental Evaluation of Running Time

- Write a program implementing the algorithm
- Run the program with some inputs
    - varying size and composition
- You can use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Empirical Analysis of Time Efficiency

- Select a specific (typical) sample of inputs

- Use physical unit of time (e.g., milliseconds)

  or

  Count actual number of basic operation's executions

- Analyze the empirical data

Limitations of Experiments

- Experimental evaluation of running time is very useful but
  - It is necessary to implement the algorithm, which may be difficult (in terms of time) and can be expensive
  - Results may not be indicative of the running time on other inputs not included in the experiment
  - In order to compare two algorithms, the same hardware and software environments must be used

How to (theoretically) calculate the running time?

- Most algorithms transform input objects into output objects

  - The running time of an algorithm typically grows with the input size
    - idea: analyze running time as a function of input size

How to Calculate Running Time

- Problem: finds the first prime number in an array by scanning it left to right
  - Given an algorithm, running time can be very different even on inputs of the same size,

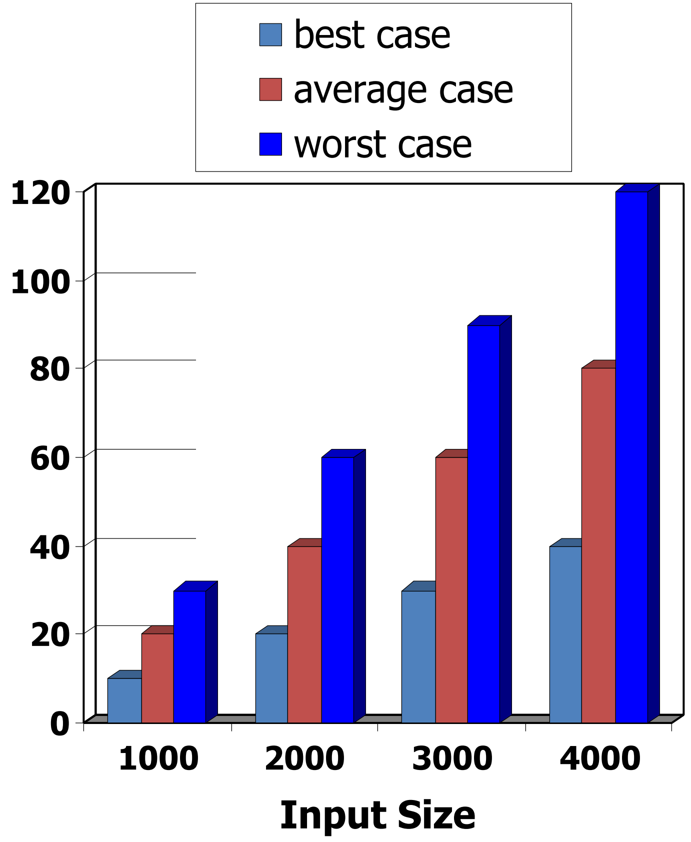
5	3	1	2	8	4	7	6

1	4	6	8	5	3	2	7

- Idea: analyze running time in the
  - best case
  - worst case
  - average case

How to Calculate Running Time

- Best case running time is usually useless
- Average case time is very useful but often difficult to determine
- We focus on the worst case running time
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



Theoretical Analysis of Time Efficiency

- Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*
- *Basic operation*: the operation that contributes most towards the running time of the algorithm

input size

$$T(n) \approx c_{op}C(n)$$

running time

execution time for the basic operation

Number of times basic operation is executed

Input size and basic operation examples

Problem	*Input size measure*	*Basic operation*
Searching for a key in a list of n items	Number of list's items, i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	n'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Review (slides from CSC 280):

- The following topics were covered in the past:
    - Discuss the goals of software development with respect to efficiency
    - Introduce the concept of algorithm analysis
    - Explore the concept of asymptotic complexity
    - Compare various growth functions

- We will review those topics and add depth to them

Analysis of Algorithms

- An aspect of software quality is the efficient use of resources, including the CPU time and memory
- Algorithm analysis is a core computing topic
- It gives us a basis to compare the efficiency of algorithms
- Example: which sorting algorithm is more efficient?

Growth Functions

- Analysis is defined in general terms, based on:
  - the problem size (ex: number of items to sort)
  - key operation (ex: comparison of two values)

- A *growth function* shows the relationship between the size of the problem (n) and the time it takes to solve the problem

- For example:

$$t(n) = 15n^2 + 45\,n$$

Growth of Functions

How much (unit) time is needed for a problem size of N if you have the growth function: $t(n) = 15n^2 + 45n$

Number of dishes (n)	$15n^2$	45n	$15n^2 + 45n$
1	15	45	60
2	60	90	150
5	375	225	600
10	1,500	450	1,950
100	150,000	4,500	154,500
1,000	15,000,000	45,000	15,045,000
10,000	1,500,000,000	450,000	1,500,450,000
100,000	150,000,000,000	4,500,000	150,004,500,000
1,000,000	15,000,000,000,000	45,000,000	15,000,045,000,000
10,000,000	1,500,000,000,000,000	450,000,000	1,500,000,450,000,000

FIGURE 2.1 Comparison of terms in growth function

Growth Functions

- It's not usually necessary to know the exact growth function
- The key issue is the *asymptotic complexity* of the function – how it grows as n increases
- Determined by the dominant term in the growth function
- This is referred to as the *order* of the algorithm
- We often use *Big-Oh notation* to specify the order, such as $O(n^2)$

Some growth functions and their asymptotic complexity

Growth Function	Order	Label
$t(n) = 17$	$O(1)$	constant
$t(n) = 3\log n$	$O(\log n)$	logarithmic
$t(n) = 20n - 4$	$O(n)$	linear
$t(n) = 12n \log n + 100n$	$O(n \log n)$	n log n
$t(n) = 3n^2 + 5n - 2$	$O(n^2)$	quadratic
$t(n) = 8n^3 + 3n^2$	$O(n^3)$	cubic
$t(n) = 2^n + 18n^2 + 3n$	$O(2^n)$	exponential

FIGURE 2.2 Some growth functions and their asymptotic complexity

Key: ignore multiplicative constants
and the lower order terms

Do the growth functions really matter?

Is the following statement true?

- With the advances in the speed of processors and the availability of large amounts of inexpensive memory, one can simply find a faster CPU to overcome the inefficiency of algorithm.

Increase in problem size with a ten-fold increase in processor speed

Algorithm	Time Complexity	Max Problem Size Before Speedup	Max Problem Size After Speedup
A	n	s_1	$10s_1$
B	n^2	s_2	$3.16s_2$
C	n^3	s_3	$2.15s_3$
D	2^n	s_4	$s_4 + 3.3$

FIGURE 2.3 Increase in problem size with a tenfold increase in processor speed

Comparison of typical growth functions for small values of N

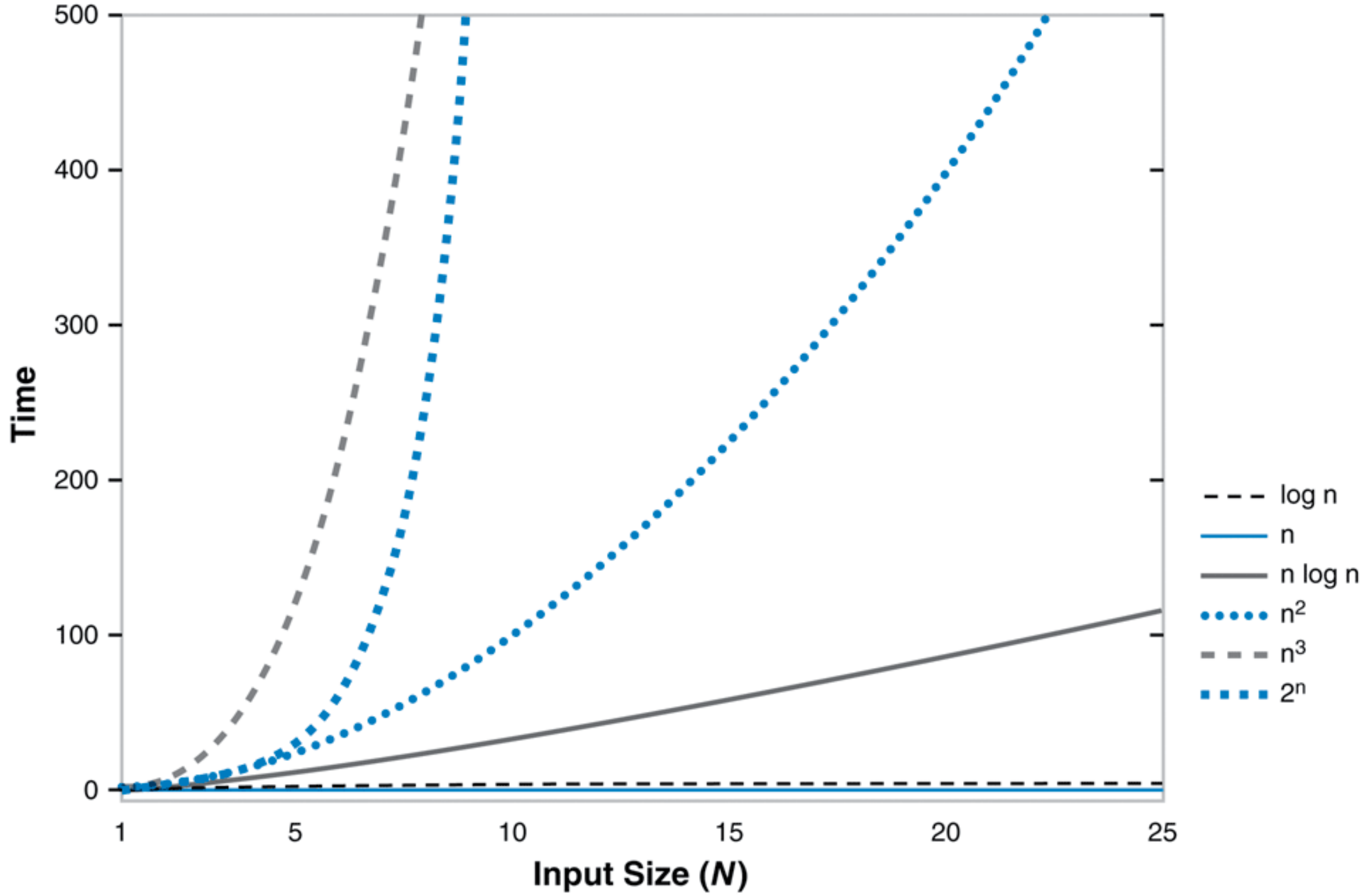


FIGURE 2.4 Comparison of typical growth functions for small values of n

Comparison of typical growth functions for large values of N

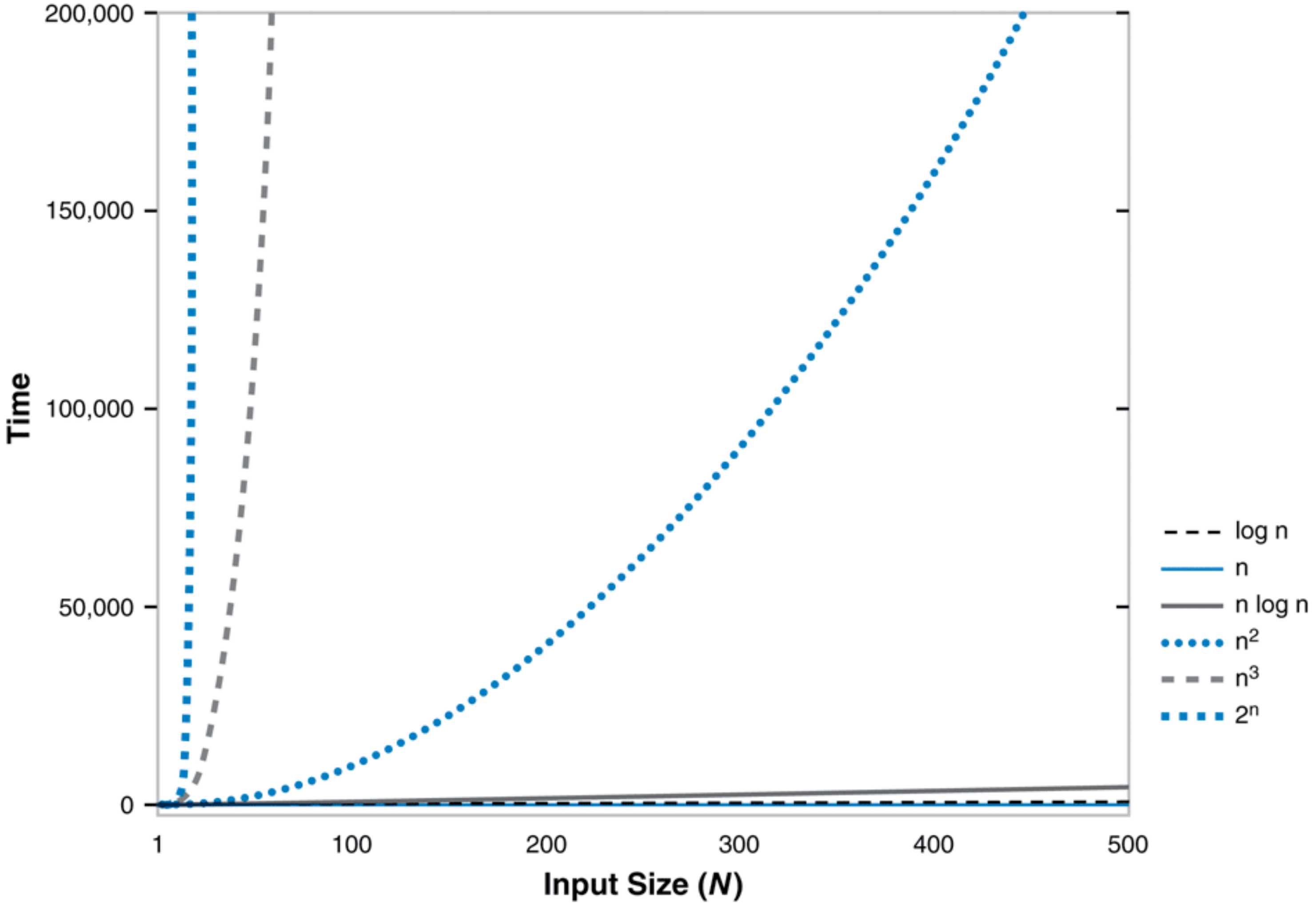


FIGURE 2.5 Comparison of typical growth functions for large values of n

Analyzing Loop Execution

- A loop executes a certain number of times (say n)
- Thus the complexity of a loop is n times the complexity of the body of the loop
- When loops are nested, the body of the outer loop includes the complexity of the inner loop

Analyzing Loop Execution

- What is the time complexity of the following loop?

```
x=0;
for (int i = 0; i < n; i++){
    x = x + 1;
}
```

- The time complexity of the loop is O(n) because the loop executes n times and the body of the loop is O(1)

Analyzing Loop Execution

- What is the time complexity of the following loop?

```
for (int i=0; i<n; i++) {
     x = x + 1;
     for (int j=0; j<n; j++){
          y = y - 1;
     }
}
```

O(n) O(n)

- The time complexity of the loop is $O(n^2)$ because the loop executes n times and the body of the loop, including a nested loop, is O(n)

Examples

- Find the sum of 1 to n.

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

- The time complexity of the for loop is O(n)

Does there exist a better algorithm?

Examples

- Find the sum of 1 to n.

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

Does there exist a better algorithm?

```
int sum = n*(n+1)/2;
```

- The time complexity of the for loop is O(n)
- The time complexity of the formula is O(1)

Analyzing Method Calls

- To analyze method calls, we simply replace the method call with the order of the body of the method
- A call to the following method is O(1)

```
public void printsum(int count)
{
    sum = count*(count+1)/2;
    System.out.println(sum);
}
```

More examples

- What is the time complexity of the following while loop?

```
while (count < n) {
    x = x + 1;
    count++;
}
```

```
while (count < 2n) {
    x = x + 2;
    count++;
}
```

- The time complexity of the either while loop is O(n)

More examples

```
for (int count=0; count<n; count++){
   printsum(count);
 }

public void printsum(int count){
   int sum=0;
   for (int i=0; i<count; i++) {
       sum = sum + i;
    }
    System.out.println(sum);
}
```

More examples

```
for (int count=0; count<n; count++){
    printsum(count);
 }

public void printsum(int count){
   int sum=0;
   for (int i=0; i<count; i++) {
       sum = sum + i;
    }
    System.out.println(sum);
}
```

The time complexity is $O(n^2)$

Two Broad Classes of Analysis

- Nonrecursive Algorithms
- Recursive Algorithms

- These are "equivalent" in the sense that we can convert between them:
  - Recursive -> Nonrecursive: Simulate the recursion in a loop.
  - Nonrecursive -> Recursive: Study functional programming.

Analyze the time efficiency of <u>non-recursive</u> algorithms

- General Plan for Analysis

  - Decide on parameter *n* indicating <u>*input size*</u>
  - Identify algorithm's <u>*basic operation*</u>
  - Determine <u>*worst*</u>, <u>*average*</u>, and <u>*best*</u> cases for input of size *n*
  - Set up a sum for the number of times the basic operation is executed
  - Simplify the sum using standard formulas and rules

Example: Sequential search

- Worst case?
- Best case?
- Average case?

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Solution

- $C_{worst}(n) = n$

- $C_{best}(n) = 1$

- $C_{avg}(n) = 1 * \frac{p}{n} + 2 * \frac{p}{n} + \cdots + i * \frac{p}{n} + \cdots + n * \frac{p}{n} + n(1 - p)$

  $= (1 + n) * \frac{p}{2} + n(1 - p)$

  $= (1 - \frac{p}{2}) * n + \frac{p}{2}$

  - p: the probability the key is in array A[1..n]

Useful summation formulas and rules

$\Sigma_{l\le i\le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular $l = 1$, $u = n$, $\Sigma_{l\le i\le u} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1\le i\le n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1\le i\le n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0\le i\le n}\, a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \ne 1$

In particular, $\Sigma_{0\le i\le n}\, 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i$

$\Sigma c a_i = c \Sigma a_i$

$\Sigma_{l\le i\le u} a_i = \Sigma_{l\le i\le m} a_i + \Sigma_{m+1\le i\le u} a_i$

Asymptotic order of growth

- O(g(n)): big oh
  - The set of all functions with <u>a smaller or same order</u> of growth as g(n)
  - Class of functions $f(n)$ that grow <u>no faster</u> than $g(n)$

- Ω(g(n)): big omega
  - The set of functions with <u>a larger or same order</u> of growth as g(n)
  - Class of functions $f(n)$ that grow <u>at least as fast</u> as $g(n)$

- Θ(g(n)): big theta
  - The set of all functions that have <u>the same order</u> of growth as g(n)
  - class of functions $f(n)$ that grow <u>at same rate</u> as $g(n)$

Big-oh

- t(n) <= cg(n)   for all n >= n_0

f(*n*) that grow no faster than *g*(*n*)

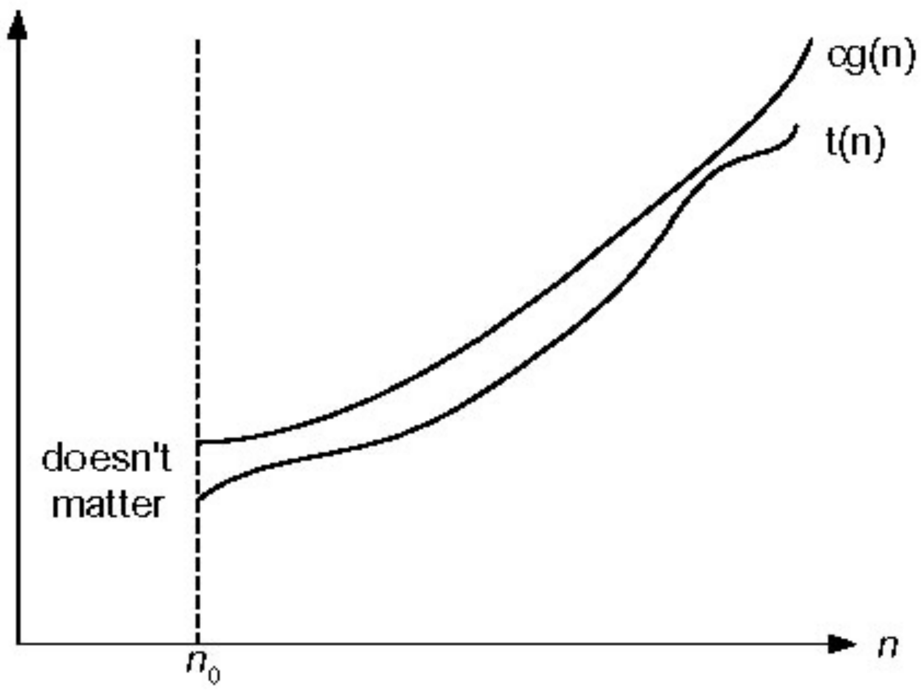
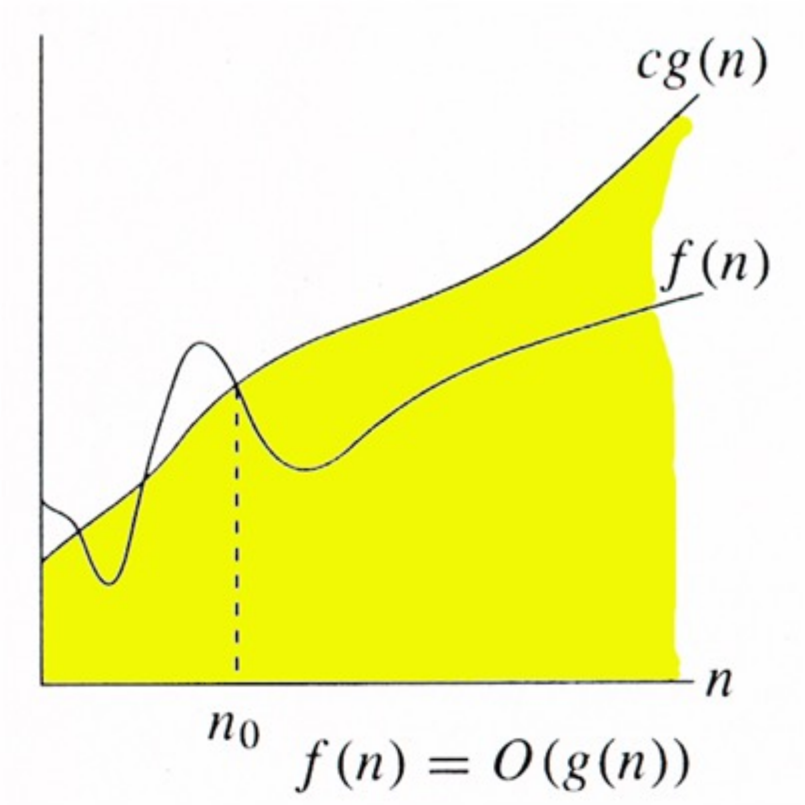


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$



Big-omega

- $t(n) >= cg(n)$ for all $n >= n_0$

f(*n*) that grow <u>at least as fast</u> as *g*(*n*)

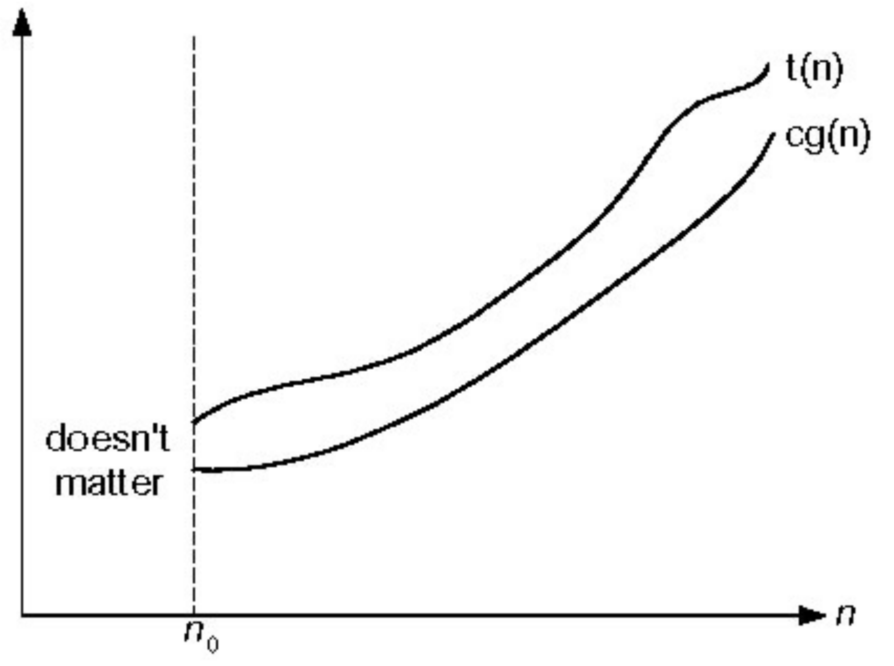
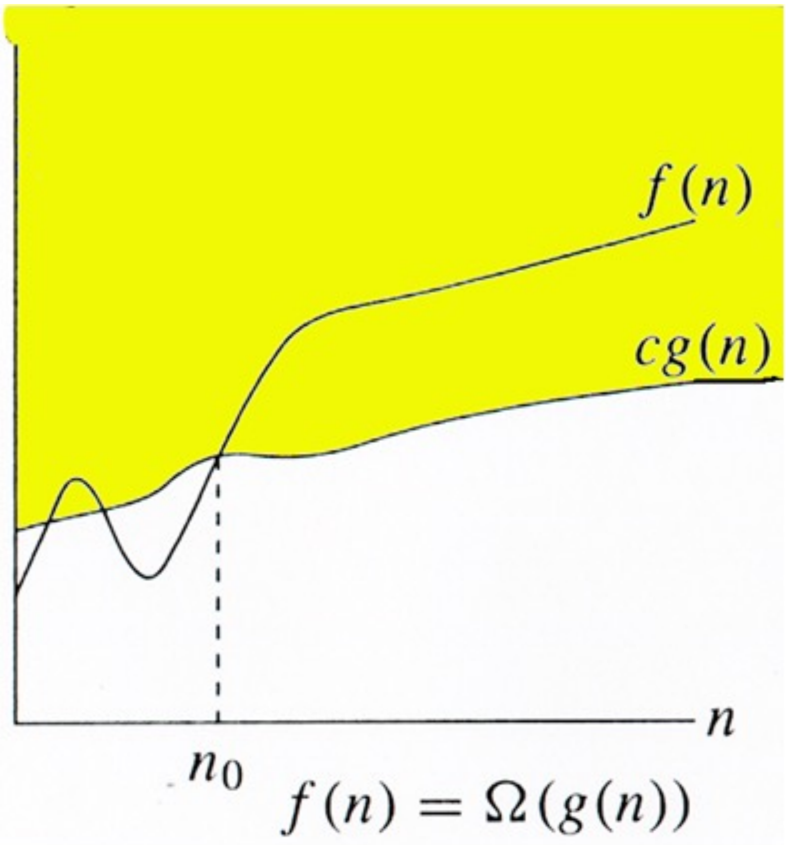


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$



Big-theta

- $c_2g(n) <= t(n) <= c_1g(n)$ for all $n >= n_0$

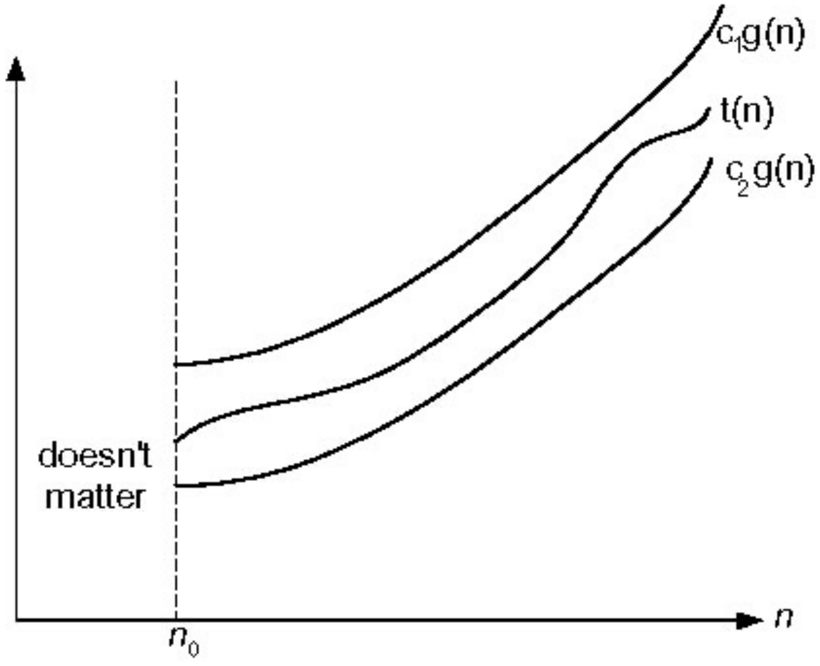
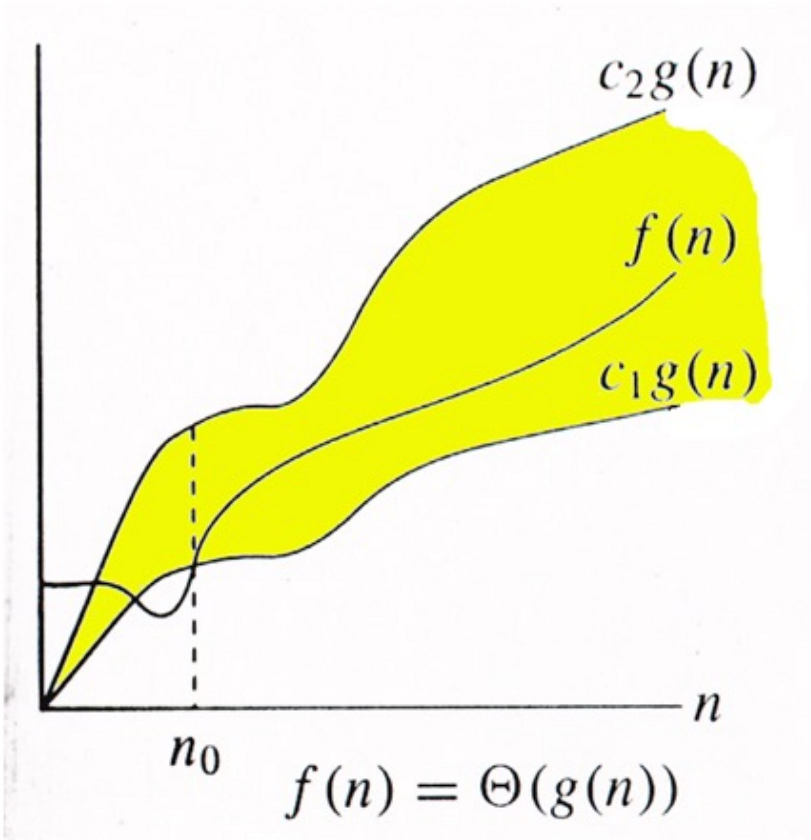


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

f(*n*) that grow <u>at same rate</u> as *g*(*n*)



Example: Element uniqueness problem

ALGORITHM *UniqueElements*$(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

What is t(n) = ?

What is the big theta of the algorithm? $\Theta(n^2)$

Example: Matrix multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1,\ 0..n-1],\ B[0..n-1,\ 0..n-1])$

//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix $C = AB$

$$\begin{array}{r} \\ \\ \text{row } i \hookrightarrow \\ \\ \\ \end{array} \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & a_{i3} & \dots & a_{in} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} \cdot \begin{array}{c} \text{column } j \\ \downarrow \\ \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1j} & \dots & b_{1n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{i1} & b_{i2} & \dots & b_{ij} & \dots & b_{in} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nj} & \dots & b_{nn} \end{bmatrix} \end{array} =$$

$$= \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1j} & \dots & c_{1n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ c_{i1} & c_{i2} & \dots & \boxed{\mathbf{c_{ij}}} & \dots & c_{in} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \dots & c_{nj} & \dots & c_{nn} \end{bmatrix} \quad \begin{array}{c} \text{entry on row } i \\ \text{column } j \end{array}$$

Example: Matrix multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$
    //Multiplies two n-by-n matrices by the definition-based algorithm
    //Input: Two n-by-n matrices A and B
    //Output: Matrix $C = AB$
    for $i \leftarrow 0$ **to** $n-1$ **do**
        for $j \leftarrow 0$ **to** $n-1$ **do**
            $C[i, j] \leftarrow 0.0$
            for $k \leftarrow 0$ **to** $n-1$ **do**
                $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
    return C

What is $t(n)$ = ?

What is the big theta of the algorithm? $\Theta(n^3)$

Common time complexities

- Arranging the following time complexities from better to worst.
- $O(n)$, $O(n^2)$, $O(n^3)$, $O(1)$, $O(\log n)$, $O(n \log n)$, $O(2^n)$

Common time complexities

BETTER

- $O(1)$ constant time
- $O(\log n)$ log time
- $O(n)$ linear time
- $O(n \log n)$ log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(2^n)$ exponential time

WORSE

Math you need to review

- Summations (see CSC 210)
- Logarithms and Exponents
  - **properties of logarithms:**

    $\log_b(xy) = \log_b x + \log_b y$

    $\log_b (x/y) = \log_b x - \log_b y$

    $\log_b x^a = a\log_b x$

    $\log_b a = \log_x a/\log_x b$

  - **properties of exponentials**:

    $a^{(b+c)} = a^b a^c$

    $a^{bc} = (a^b)^c$

    $a^b / a^c = a^{(b-c)}$

    $b = a^{\log_a b}$

    $b^c = a^{c*\log_a b}$

When Do Logarithms Occur?

- Algorithms have a logarithmic term when they use a divide and conquer technique
- the data keeps getting “divided by 2”

```
// input integer: n > 0
int foo(int n)
{
    int total = 0;
    while( n > 0 )
    {    n = n / 2;
         total++;
    }
    return total;
}
```

Summary: Time efficiency of non-recursive algorithms

- General Plan for Analysis
  - Decide on parameter *n* indicating ***input size***
  - Identify algorithm's ***basic operation***
  - Determine ***worst***, ***average***, and ***best*** cases for input of size *n*
  - Set up a sum for the number of times the basic operation is executed
  - Simplify (or evaluate) the sum using standard formulas and rules.