Complexity Analysis

Lecture 02

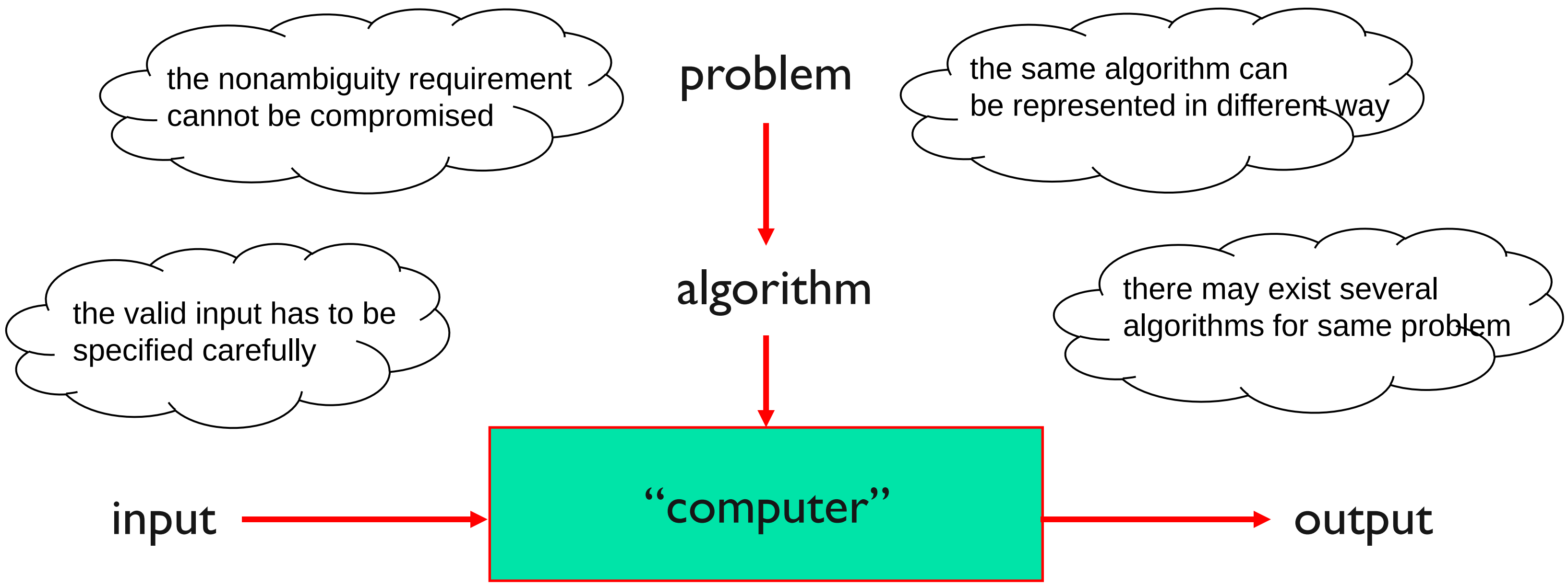
Instructor: Dr. Cong Pu, Ph.D.

cong.pu@okstate.edu

Adapted partially from Data Structures and Algorithms in Java, M.T. Goodrich, R.Tamassia and M. H. Goldwasser, Sixth Edition, Wiley; Data Structures and Algorithms in C++, Adam Drozdek, 4th Edition, Cengage Learning

What is an Algorithm?

- *Algorithm*
  - a sequence of unambiguous instructions for solving a problem
  - e.g., obtaining a required output for any valid input in a finite amount of time.



Why Study Algorithms?

- Theoretical importance
  - the core of computer science

- Practical importance
  - a practitioner's toolkit of known algorithms
  - framework for designing and analyzing algorithms for new problems
  - algorithm design techniques → problem solving strategies

Two Main Issues Related to Algorithms

- *Algorithm design techniques*
  - a general approach to solving problems algorithmically
    - applicable to a variety of problems from different areas
  - guidance for designing algorithms for new problems

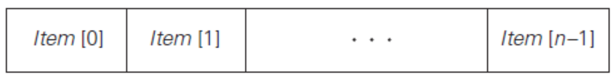
- How to **analyze algorithm efficiency**?
  - How good is the algorithm?
    - time efficiency
    - space efficiency
  - Does there exist a better algorithm?
    - lower bounds
    - optimality

Important Problem Types

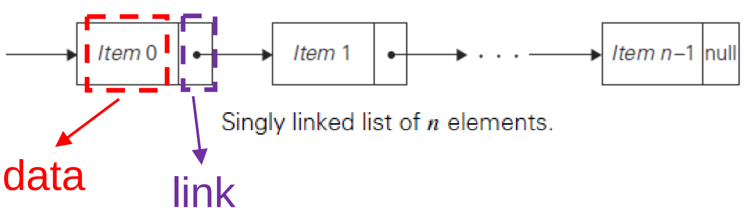
- sorting
- searching
- string processing
- graph problems
- numerical problems
- etc.

Fundamental Data Structures

- ***Data structure***: a particular scheme of organizing related data items
- Linear data structures:
  - array



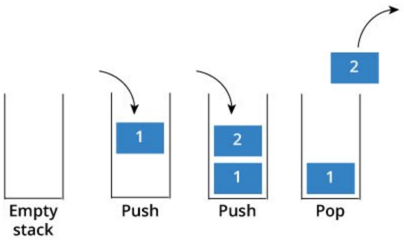
  - linked list
    - a sequence of 0 or more elements (called nodes)
    - node:
      - data
      - link to another node



Singly linked list of n elements.

Fundamental Data Structures

- ***Data structure***: a particular scheme of organizing related data items
- Linear data structures:
  - two special lists

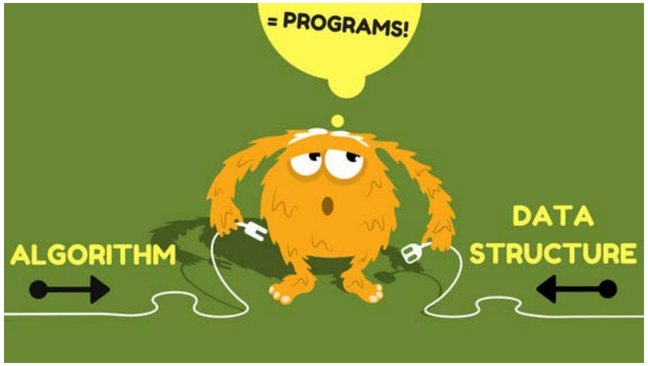


    - stack
      - operations (insert and delete) can be done only at the end (called *top*)
      - last-in-first-out fashion

    - queue
      - deletion at one end (called *front*)
      - insertion at the other end (called *rear*)
      - first-in-first-out fashion

Algorithm vs. Data Structure

- Suppose we have two algorithms, how can we tell which one is better?
  - could implement both algorithms, run them both
    - expensive and error prone…
  - preferably, analyze them mathematically
    - ***algorithm analysis***
- ***Algorithms*** ←→ **Data Structures**
  - data structures are implemented using algorithms



Computational and Asymptotic Complexity

- The same problem can be solved by multiple algorithms
  - but, differ in efficiency
    - for small amount of data, differences are not significant
    - differences grow with the amount of data
- Compare the efficiency of algorithms → *computational complexity*
- *Computational complexity* indicates
  - how much effort needed to apply the algorithm, or
  - how costly it is
    - cost is interpreted in different ways
    - depending on the context

Computational and Asymptotic Complexity

- *Two efficiency criteria*
  - **time**, the amount of **time** an algorithm takes in terms of the amount of input
  - **space**, the amount of **memory (space)** an algorithm takes in terms of the amount of input
  - the factor of time is usually more important than that of space
    - *running time is system-dependent and language-dependent*
- Algorithm's ***asymptotic complexity***
  - when ***n* (number of input items)** goes to infinity, what happens to the algorithm's performance?

Computational and Asymptotic Complexity

- When evaluating algorithm's efficiency, we DO NOT use real-time units (e.g., microseconds, …)
- *Logical units*
  - expressing a *relationship* between the size ***n*** of input and the amount of time ***t*** required to process the input
  - e.g.,
    - suppose a linear relationship between the size ***n*** and the time ***t***

$$t = cn$$

      - an increase of input by a factor of 5 → the increase of time by the same factor

$$n_2 = 5n_1 \rightarrow t_2 = 5t_1$$

Computational and Asymptotic Complexity

- The relationship function between ***n*** and ***t*** is usually complex
  - discard the terms that do not substantially change function’s magnitude
  - the resulting function provides an approximate measure of efficiency
    - sufficiently ***close*** to the original, especially with large quantities of data
- ***Asymptotic* complexity**
  - used when
    - discarding certain terms to express the efficiency
    - approximations are acceptable

Computational and Asymptotic Complexity (cont.)

- e.g., $f(n) = n^2 + 100n + \log_{10}n + 1000$
- As the value of n increases, only the **n^2 term** is significant

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

quadratic growth

Big-O Notation

- The most commonly used notation for asymptotic complexity
  - estimate the rate of function growth
  - e.g., $n^2 + 100n + \log_{10} n + 1000 = O(n^2)$

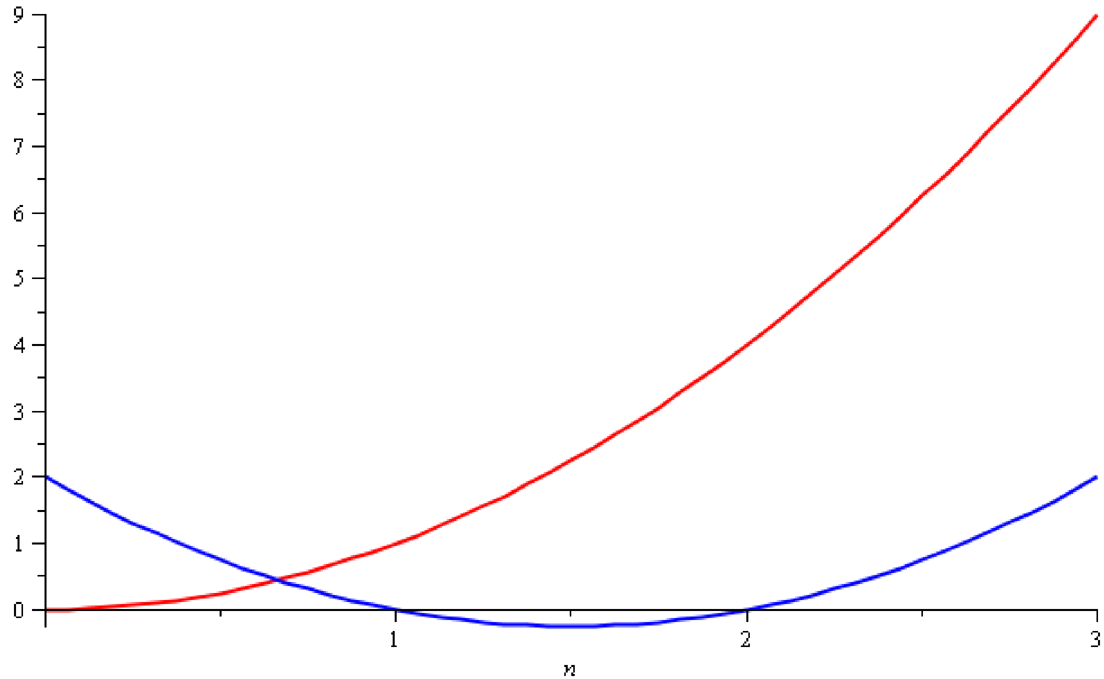
big-O notation

- **Definition**:
  - Let $f(n)$ and $g(n)$ be positive-valued functions, where n is a positive integer.
  - We write $\boldsymbol{f(n) = O(g(n))}$ if and only if there exists a real number c and positive integer N satisfying $\boldsymbol{0 \leq f(n) \leq cg(n)}$ for all $n \geq N$.

- Examples:
  - $f(n) = 3n + 2$

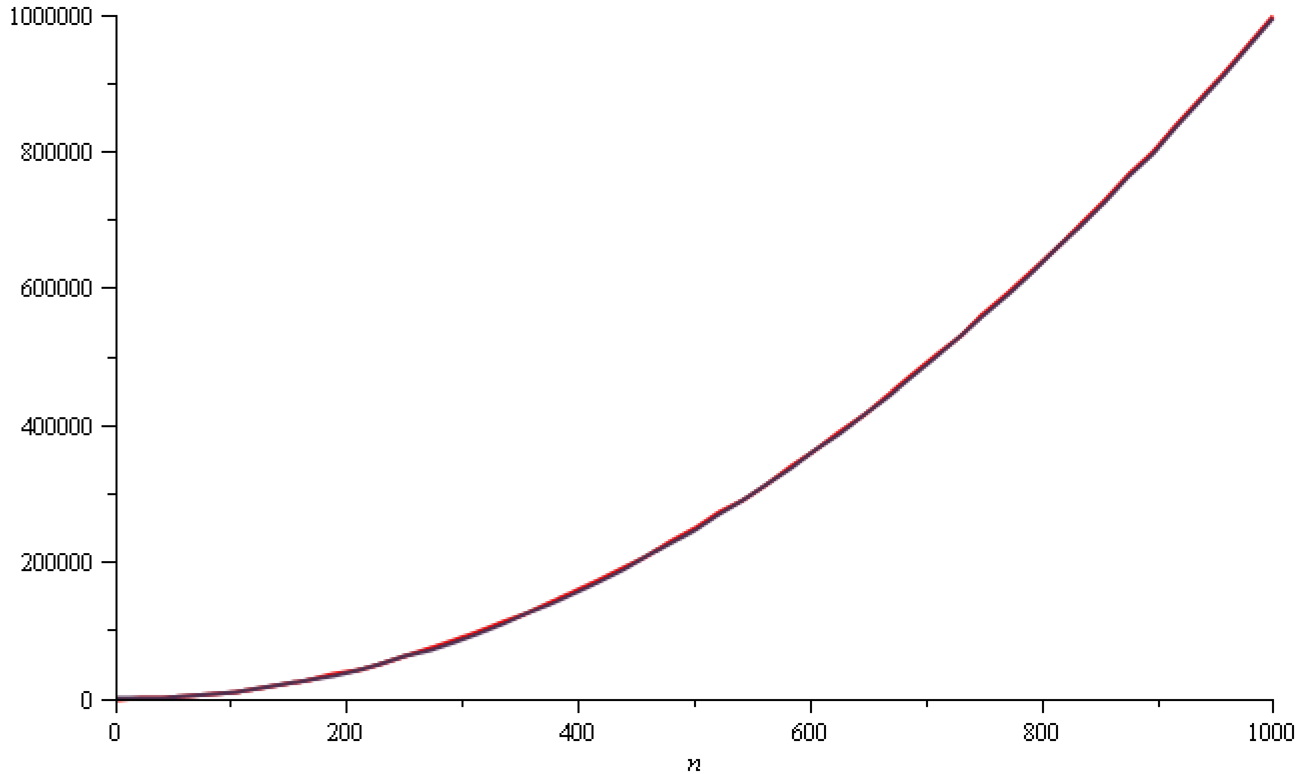
Quadratic Growth

- Consider the two functions
  - $f(n) = n^2$ and $g(n) = n^2 - 3n + 2$
  - Around $n = 0$, they look very different



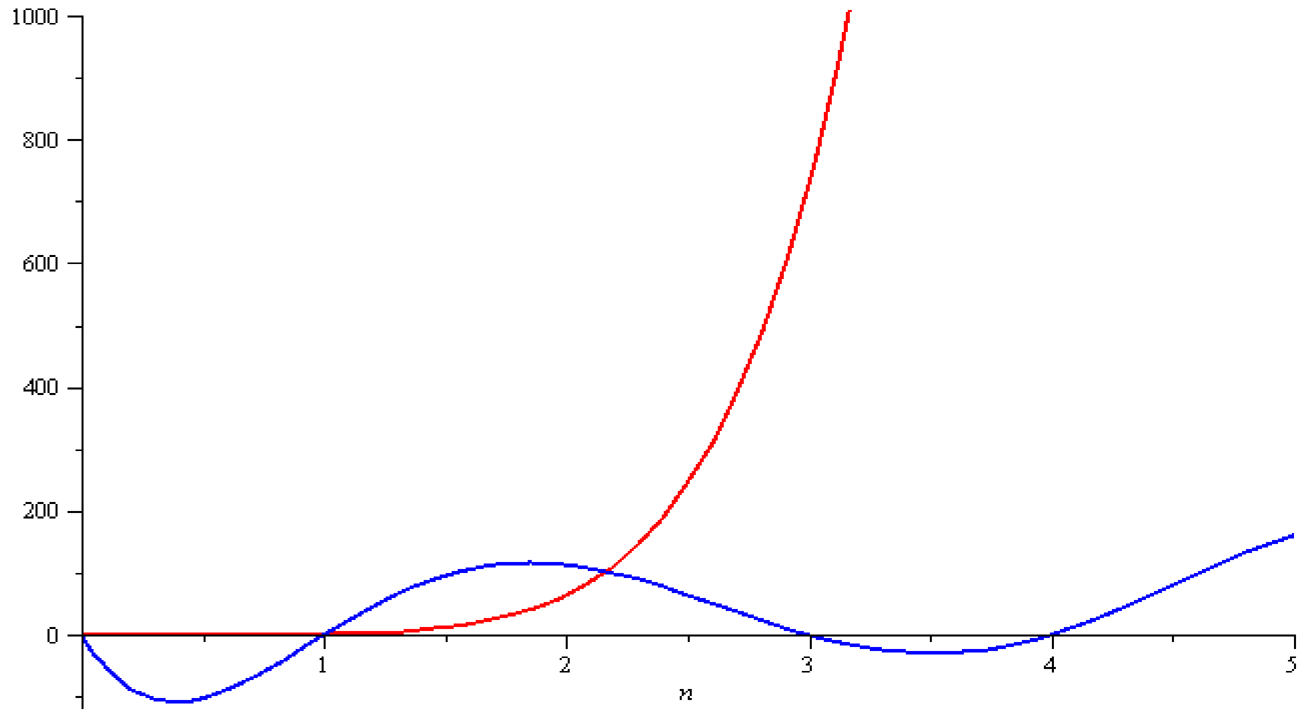
Quadratic Growth (cont.)

- Yet on the range $n = [0, 1000]$, they are (relatively) indistinguishable:



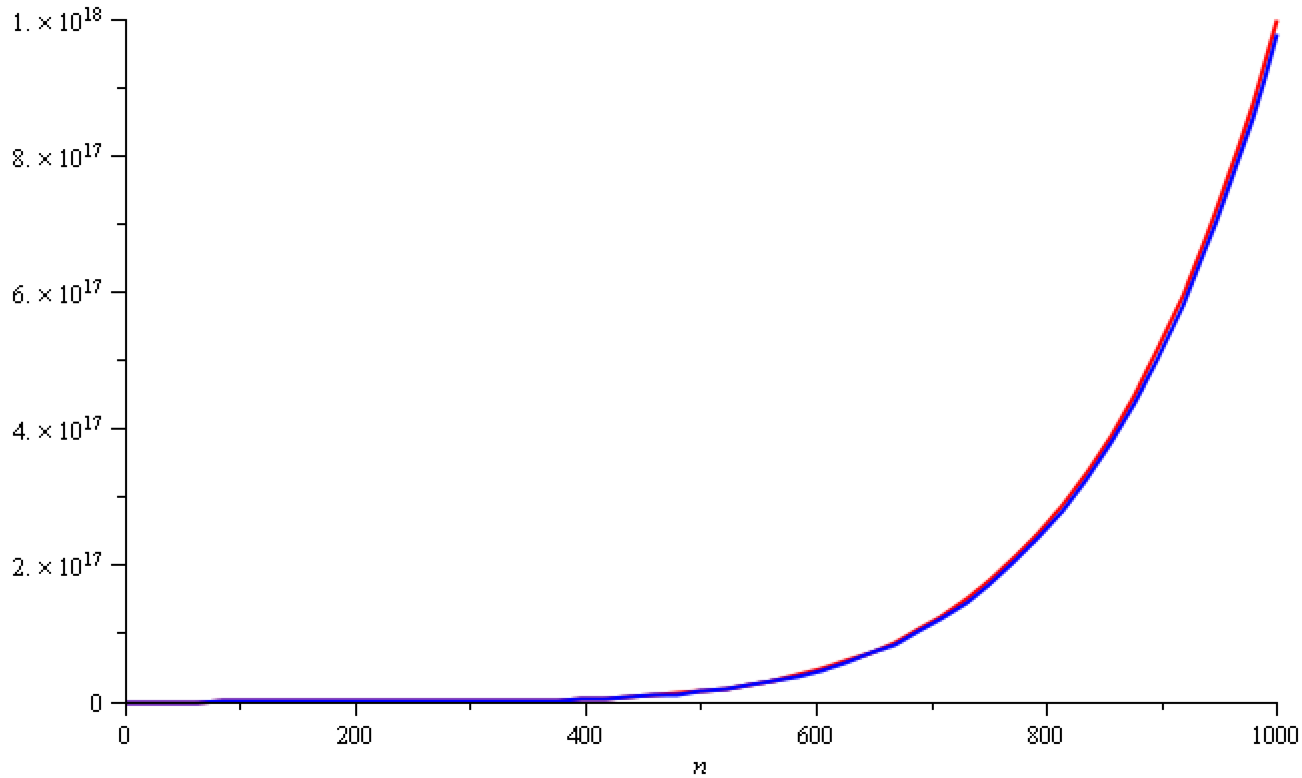
Polynomial Growth

- To demonstrate with another example,
  - $f(n) = n^6$ and $g(n) = n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$
  - Around $n = 0$, they are very different



Polynomial Growth (cont.)

- Still, around n = 1000, the relative difference is less than 3%



Big-O Notation (cont.)

- While ***c*** and ***N*** exist,
  - how to calculate them? or what to do if multiple candidates exist?
- e.g., the function *f*:

$$f(n) = 2n^2 + 3n + 1$$

and *g*:

$$g(n) = n^2$$

- Clearly $f(n)$ is $O(n^2)$; **possible candidates** for *c* and *N*

c	≥ 6	$\geq 3\frac{3}{4}$	$\geq 3\frac{1}{9}$	$\geq 2\frac{13}{16}$	$\geq 2\frac{16}{25}$	$\ldots$	$\rightarrow$	2
N	1	2	3	4	5	$\ldots$	$\rightarrow$	∞

Big-O Notation (cont.)

- Solving the inequality from the definition of big-O:

$$f(n) \leq cg(n)$$

- Substituting for $f(n)$ and $g(n)$,

$$2n^2 + 3n + 1 \leq cn^2 \quad \text{or} \quad 2 + \frac{3}{n} + \frac{1}{n^2} \leq c$$

- Since $n \geq N$, and N is a positive integer, start with $N = 1$ and substitute in either expression to obtain c

c	≥ 6	$\geq 3\frac{3}{4}$	$\geq 3\frac{1}{9}$	$\geq 2\frac{13}{16}$	$\geq 2\frac{16}{25}$	$\ldots$	$\rightarrow$	2
N	1	2	3	4	5	$\ldots$	$\rightarrow$	∞

Big-O Notation (cont.)

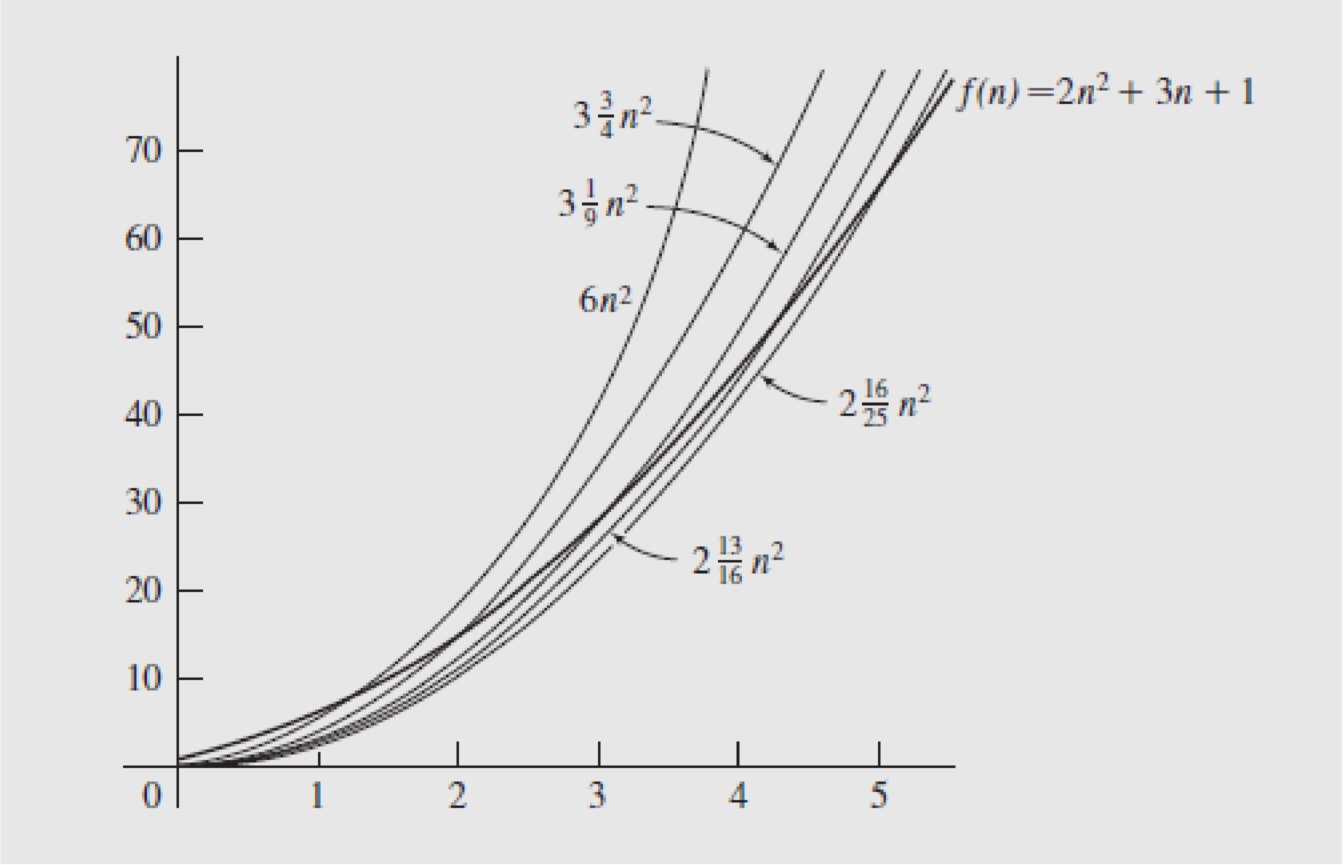
$$f(n) = 2n^2 + 3n + 1$$

- Generally, choose an N that allows **one term of *f*** to **dominate** the expression
  - only two terms to consider: **$2n^2$ and $3n$**, since the last term is a constant
  - as long as n is greater than 1.5, **$2n^2$** dominates the expression
  - N must be 2 or more, and c is greater than 3.75
- The choice of c depends on the choice of N and vice-versa

c	≥ 6	$\geq 3\frac{3}{4}$	$\geq 3\frac{1}{9}$	$\geq 2\frac{13}{16}$	$\geq 2\frac{16}{25}$	$\dots$	$\rightarrow$	2
N	1	2	3	4	5	$\dots$	$\rightarrow$	∞

Big-O Notation (cont.)

- Different values of c and N:



Examples of Complexities

- Classes of algorithms and their execution times
  - Use a computer executing 1 million operations per second

Class	Complexity	Number of Operations and Execution Time (1 instr/μsec)					
n		10		10^2		10^3	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	$O(\lg n)$	3.32	3 μsec	6.64	7 μsec	9.97	10 μsec
linear	$O(n)$	10	10 μsec	10^2	100 μsec	10^3	1 msec
$O(n \lg n)$	$O(n \lg n)$	33.2	33 μsec	664	664 μsec	9970	10 msec
quadratic	$O(n^2)$	10^2	100 μsec	10^4	10 msec	10^6	1 sec
cubic	$O(n^3)$	10^3	1 msec	10^6	1 sec	10^9	16.7 min
exponential	$O(2^n)$	1024	10 msec	10^{30}	$3.17 * 10^{17}$ yrs	10^{301}	

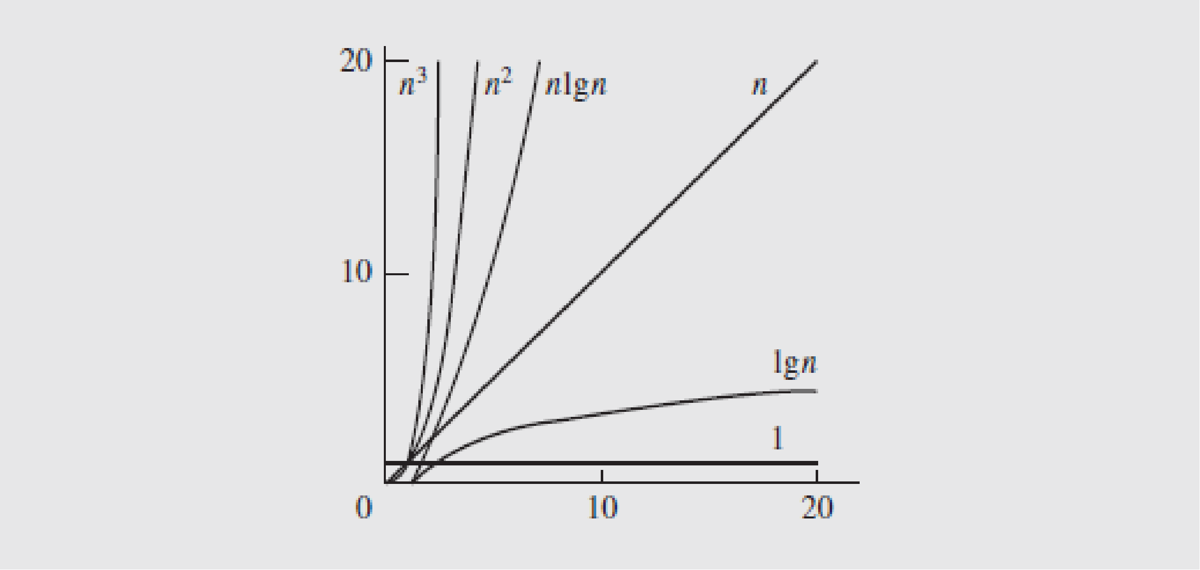
Examples of Complexities (cont.)

- The ***class*** of an algorithm based on big-O notation
  - a convenient way to describe its behavior
- e.g., a **linear function** is $O(n)$;
  - its time increases in direct proportion to the amount of data processed

n		10^4		10^5		10^6	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	$O(\lg n)$	13.3	13 μsec	16.6	7 μsec	19.93	20 μsec
linear	$O(n)$	10^4	10 msec	10^5	0.1 sec	10^6	1 sec
$O(n \lg n)$	$O(n \lg n)$	$133 * 10^3$	133 msec	$166 * 10^4$	1.6 sec	$199.3 * 10^5$	20 sec
quadratic	$O(n^2)$	10^8	1.7 min	10^{10}	16.7 min	10^{12}	11.6 days
cubic	$O(n^3)$	10^{12}	11.6 days	10^{15}	31.7 yr	10^{18}	31,709 yr
exponential	$O(2^n)$	10^{3010}		10^{30103}		10^{301030}	

Examples of Complexities (cont.)

- Relationships expressed graphically:



- With today's supercomputers..
  - cubic order algorithms or higher are impractical for large numbers of elements

Finding Asymptotic Complexity

- Asymptotic bounds
  - used to determine the time and space efficiency of algorithms
  - generally, we are interested in **time complexity**!!
- Consider a simple loop:

```
for (i = sum = 0; i < n; i++)
        sum = sum + a[i]
```

  - in initialization, execute two assignments ***once***
    - <u>sum = 0</u> and <u>i = sum</u>
  - in the loop, iterates ***n times***
    - <u>update</u> sum (sum = sum + a[i]) and <u>increment</u> i (e.g., i++)
  - 2 + 2n assignments → O(n)   /* asymptotic complexity */

Finding Asymptotic Complexity (cont.)

- A nested loop case,
  - the complexity grows by a factor of ***n***, although this isn't always the case
- Consider,

```
for (i = 0; i < n; i++) {
   for (j = 1, sum = a[0]; j <= i; j++)
      sum += a[j];
   cout << ”sum for subarray 0 through “ << i
        <<” is “<<sum<<end1;
}
```

Finding Asymptotic Complexity (cont.)

```
for (i = 0; i < n; i++) {
        for (j = 1, sum = a[0]; j <= i; j++)
            sum += a[j];
        cout << ”sum for subarray 0 through “ << i
              << ” is “ << sum << end1;
}
```

- In the outer loop, initialize i; execute ***n times***
  - increment i, and execute the inner loop and cout statement
- In the inner loop, initialize j and sum each time,
  - the number of assignments so far, 1 + 3n
  - execute i times, where i ranges from 1 to n – 1
  - each time the inner loop executes, increment j and update sum
  - the inner loop executes $\sum_{i=1}^{n-1} 2i = 2(1 + 2 + \cdots + n - 1) = n(n - 1)$ assignments
- The total number of assignments, 1 + 3n + n(n – 1) → O(n^2)