Asymptotic Analysis of Algorithms

Chapter 4

Overview

- Motivation
- Definition of Running Time
- Classifying Running Time
- Asymptotic Notation & Proving Bounds
- Algorithm Complexity vs Problem Complexity

Overview

- **Motivation**
- Definition of Running Time
- Classifying Running Time
- Asymptotic Notation & Proving Bounds
- Algorithm Complexity vs Problem Complexity

The Purpose of Asymptotic Analysis

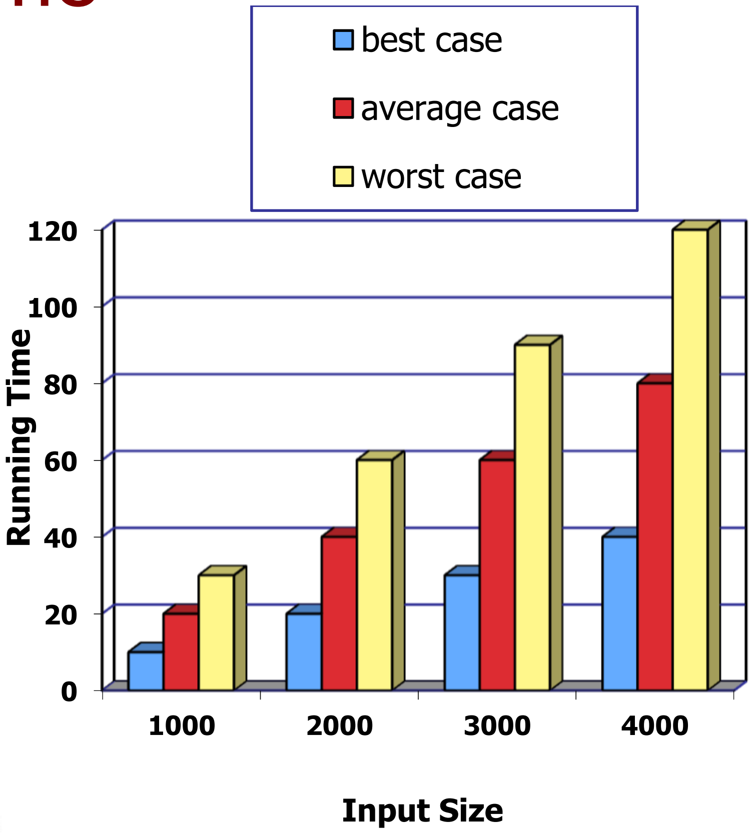
- To estimate how long a program will run.
- To estimate the largest input that can reasonably be given to the program.
- To compare the efficiency of different algorithms.
- To help focus on the parts of code that are executed the largest number of times.
- To choose an algorithm for an application.

Overview

- Motivation
- **Definition of Running Time**
- Classifying Running Time
- Asymptotic Notation & Proving Bounds
- Algorithm Complexity vs Problem Complexity

Running Time

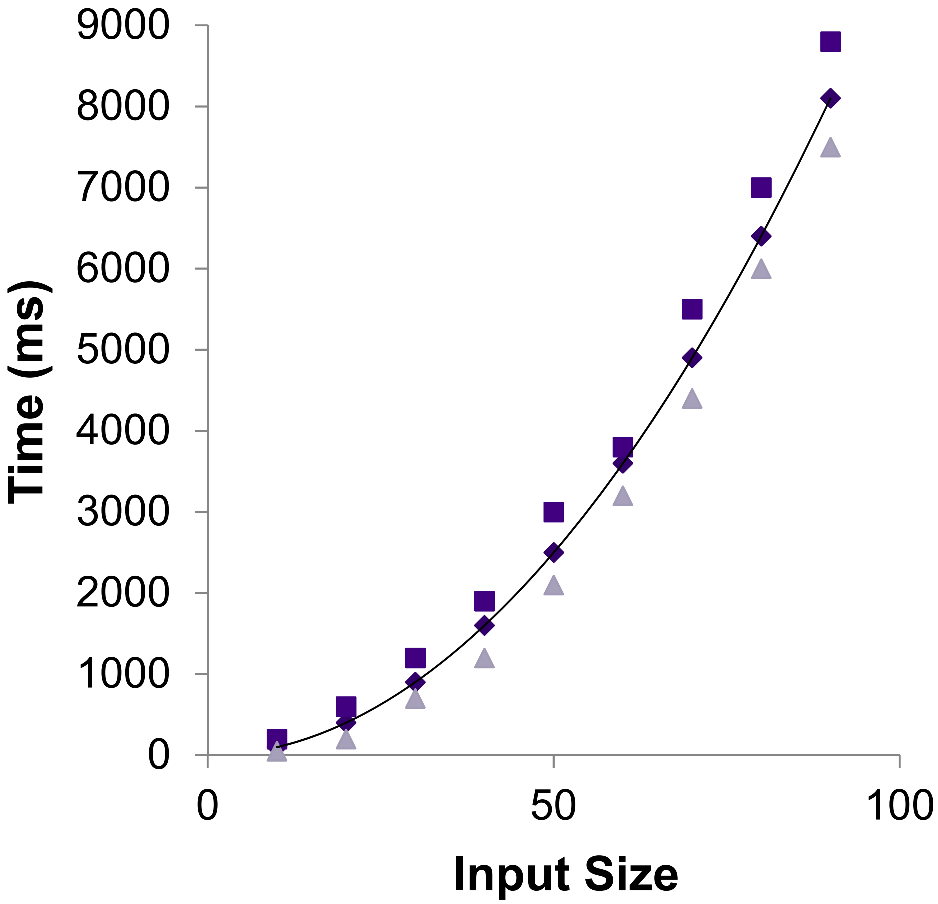
- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Reduces risk



“Level with me, Colonel. What kind of worst-case scenario are we talking about here?”

Experimental Studies

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



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- 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

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, ***n***.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Assumed to take a constant amount of time

- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

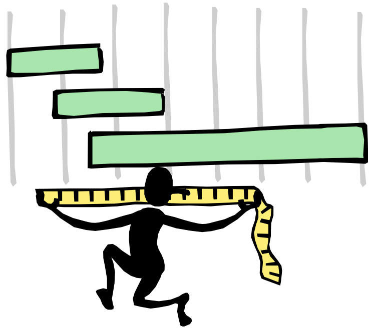
```
Algorithm arrayMax(A, n)
                                                   # operations

    currentMax ← A[0]                                   ?

    for i ← 1 to n - 1 do                               ?
        if A[i] > currentMax then                       ?
            currentMax ← A[i]                           ?
    return currentMax                                   ?

                                               Total    ?
```

Estimating Running Time



- Algorithm ***arrayMax*** executes $6n - 1$ primitive operations in the worst case. Define:

  a = Time taken by the fastest primitive operation

  b = Time taken by the slowest primitive operation

- Let $T(n)$ be worst-case time of ***arrayMax.*** Then

$$a\,(6n - 1) \le T(n) \le b(6n - 1)$$

- Hence, the running time $T(n)$ is bounded by two linear functions

Growth Rate of Running Time

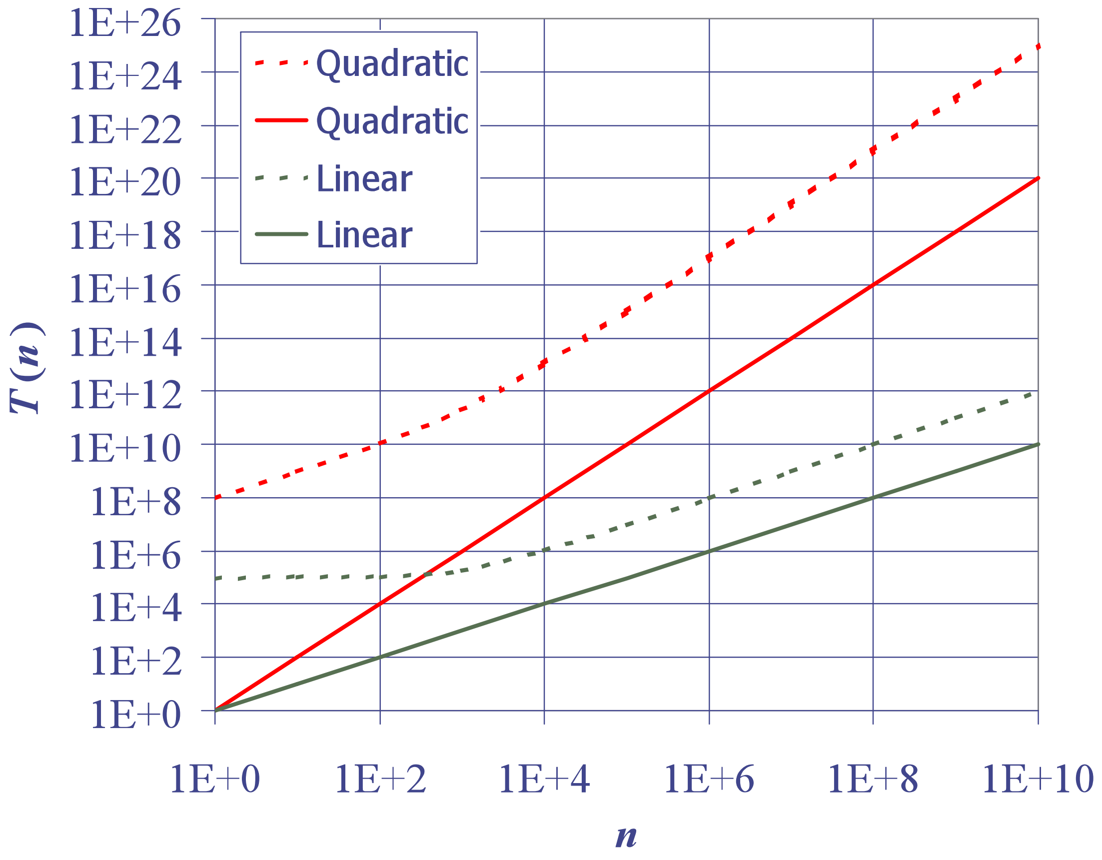
- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not qualitatively alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm ***arrayMax***

Overview

- Motivation
- Definition of Running Time
- **Classifying Running Time**
- Asymptotic Notation & Proving Bounds
- Algorithm Complexity vs Problem Complexity

Constant Factors

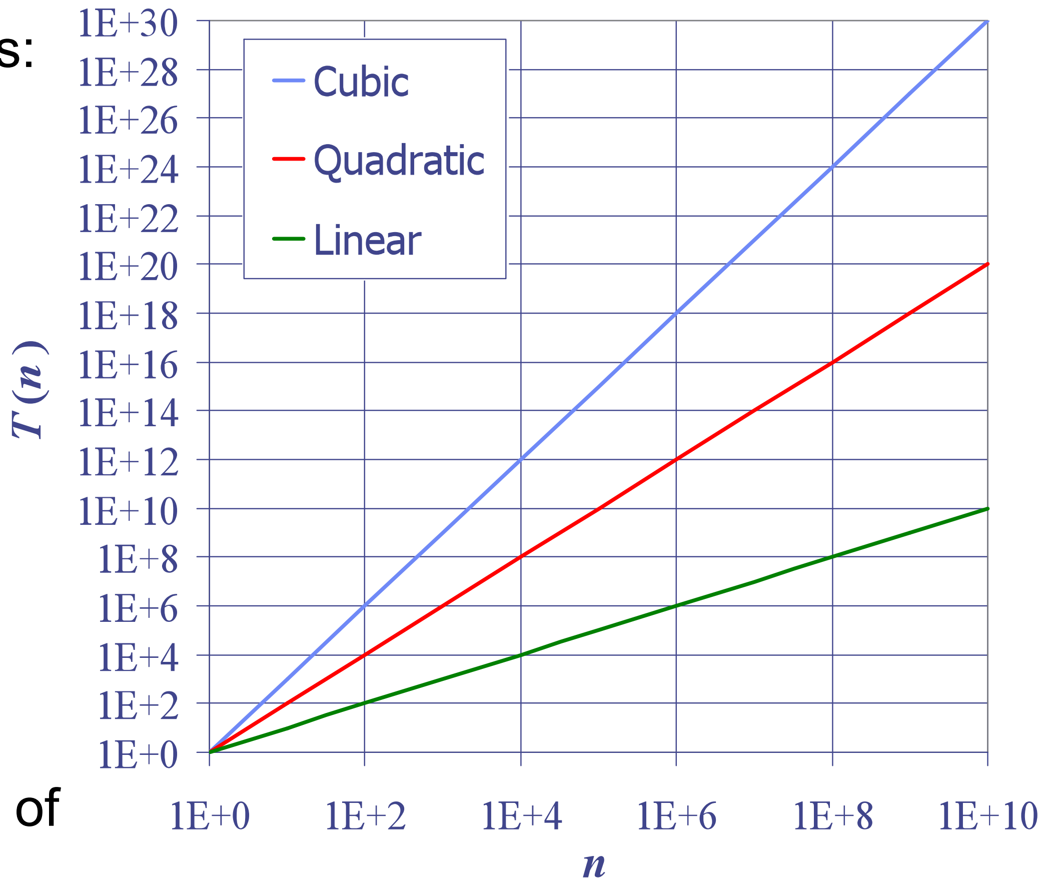
- On a logarithmic scale, the growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function



We will follow the convention that $\log n \equiv \log_2 n$.

Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic ≈ $\log n$
  - Linear ≈ n
  - N-Log-N ≈ $n \log n$
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n



- In a log-log chart, the slope of the line corresponds to the growth rate of the function.

Classifying Functions

	n			
$T(n)$	10	100	1,000	10,000
$\log n$	3	6	9	13
$n^{1/2}$	3	10	31	100
n	10	100	1,000	10,000
$n \log n$	30	600	9,000	130,000
n^2	100	10,000	10^6	10^8
n^3	1,000	10^6	10^9	10^{12}
2^n	1,024	10^{30}	10^{300}	10^{3000}

Note: The universe is estimated to contain $\sim 10^{80}$ particles.

Let’s practice classifying functions

Which are more alike?

n^{1000} n^{2} 2^{n}

Which are more alike?

$$\underbrace{n^{1000} \qquad n^{2}}_{\text{Polynomials}} \qquad 2^{n}$$

Which are more alike?

$1000n^2$ $3n^2$ $2n^3$

Which are more alike?

$$\underbrace{1000n^2 \qquad 3n^2}_{\text{Quadratic}} \qquad 2n^3$$

Overview

- Motivation
- Definition of Running Time
- Classifying Running Time
- **Asymptotic Notation & Proving Bounds**
- Algorithm Complexity vs Problem Complexity

Some Math to Review

- Summations
- Logarithms and Exponents
- Existential and universal operators
- Proof techniques

- **existential and universal operators**

$$\exists g \forall b \; Loves(b, g)$$

$$\forall g \exists b \; Loves(b, g)$$

- **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**:

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

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

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

$b = a^{\log_a b}$

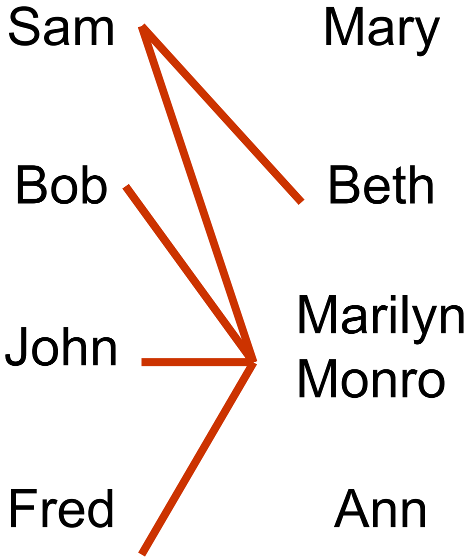
$b^c = a^{c*\log_a b}$

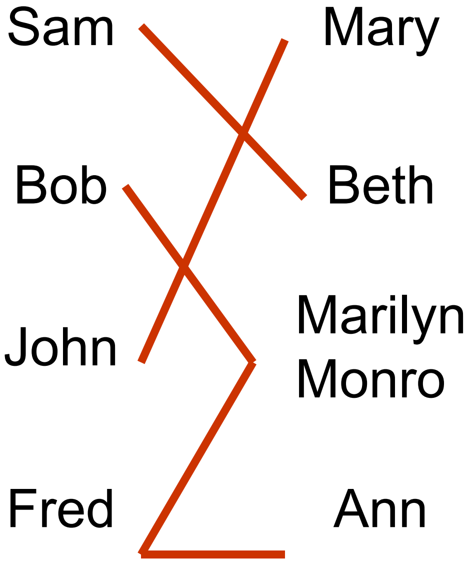
Understand Quantifiers!!!

$$\exists g, \forall b, \mathit{loves}(b,g) \qquad \forall g, \exists b, \mathit{loves}(b,g)$$

One girl

Could be a separate girl
for each boy.


Sam
Mary
Bob
Beth
John
Marilyn
Monro
Fred
Ann


Sam
Mary
Bob
Beth
John
Marilyn
Monro
Fred
Ann

Asymptotic Notation
($\mathrm{O}, \Omega, \Theta$ and all of that)

- The notation was first introduced by number theorist Paul Bachmann in 1894, in the second volume of his book *Analytische Zahlentheorie* ("analytic number theory”).
- The notation was popularized in the work of number theorist Edmund Landau; hence it is sometimes called a Landau symbol.
- It was popularized in computer science by Donald Knuth, who (re)introduced the related Omega and Theta notations.
- Knuth also noted that the (then obscure) Omega notation had been introduced by Hardy and Littlewood under a slightly different meaning, and proposed the current definition.

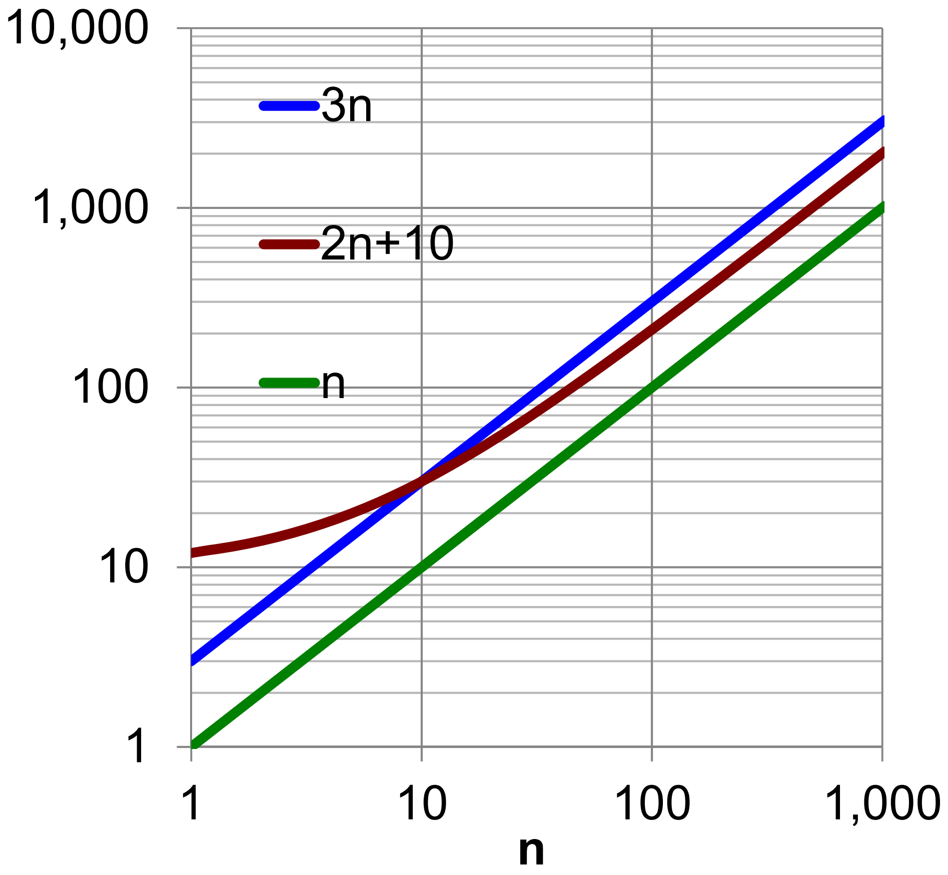
Source: Wikipedia

Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

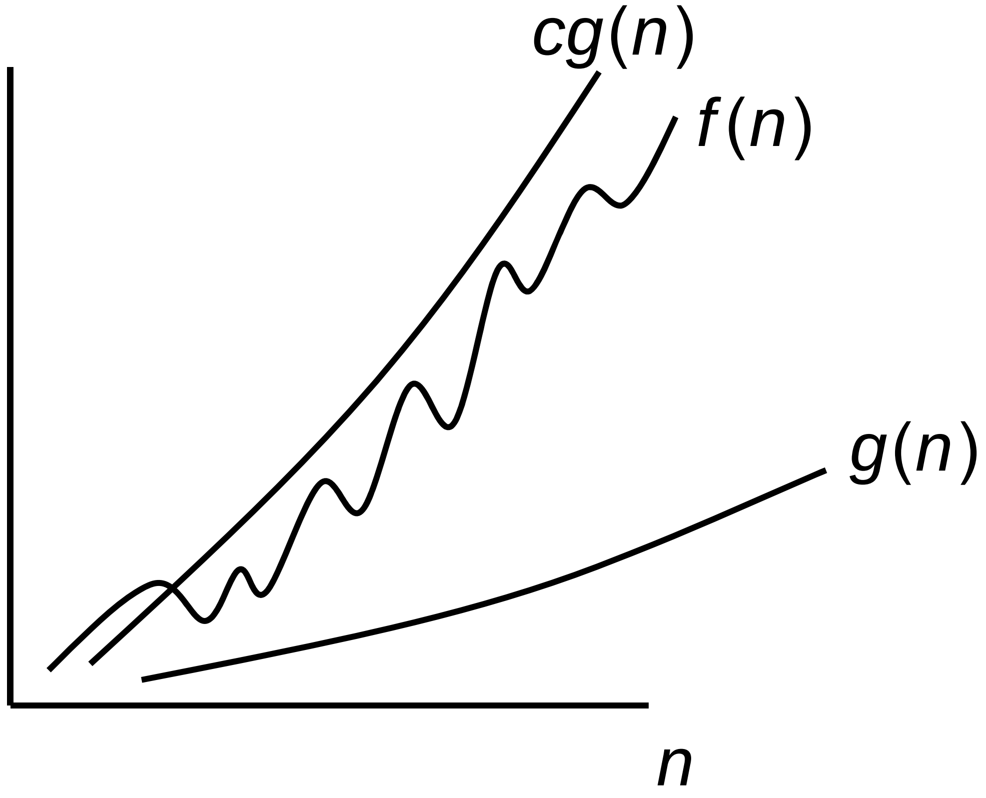
  $f(n) \leq cg(n)$ for $n > n_0$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n > 10$
  - $n > 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



Definition of "Big Oh"

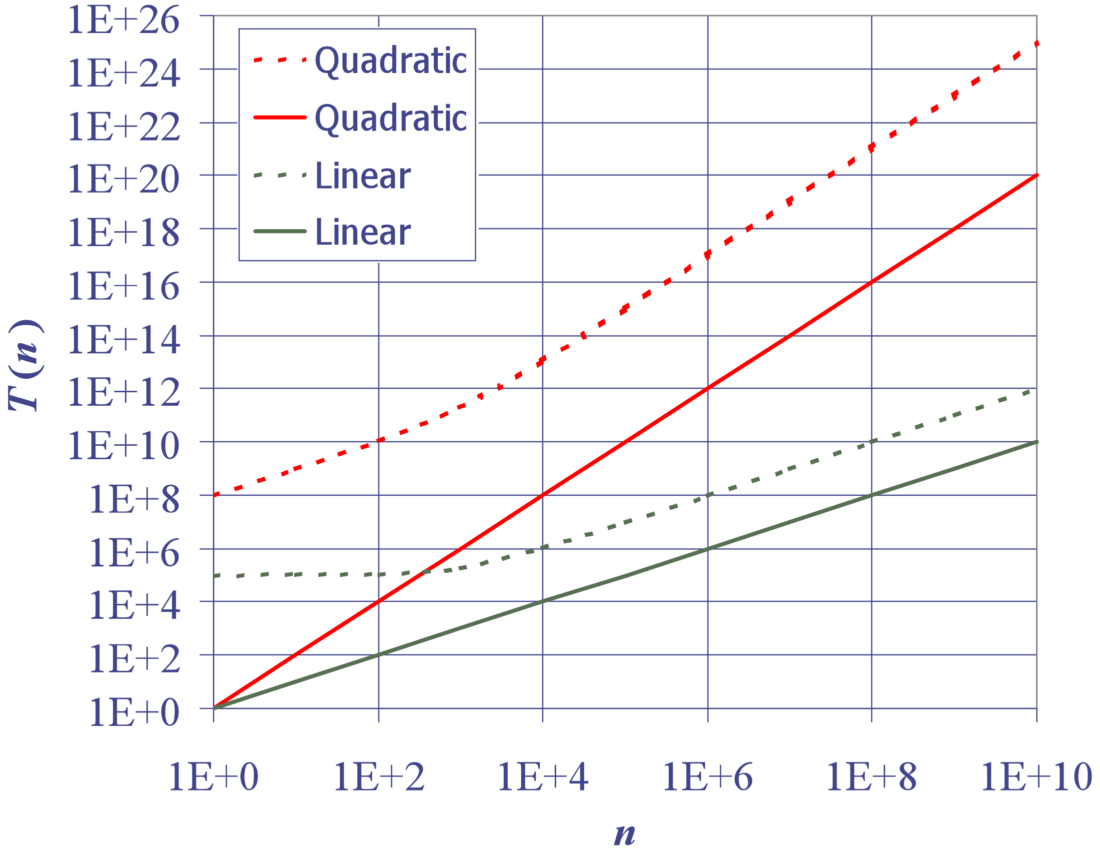
$$f(n) \in O(g(n))$$



$$\exists c, n_0 > 0 : \forall n \geq n_0, f(n) \leq cg(n)$$

Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \le cn$
  - $n < c$
  - The above inequality cannot be satisfied since c must be a constant



More Big-Oh Examples

- 7n-2

  7n-2 is O(n)

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \cdot n$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$

  this is true for $c = 5$ and $n_0 = 20$

- 3 log n + 5

  3 log n + 5 is O(log n)

  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \cdot \log n$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 32$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- We generally specify the tightest bound possible
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Asymptotic Algorithm Analysis

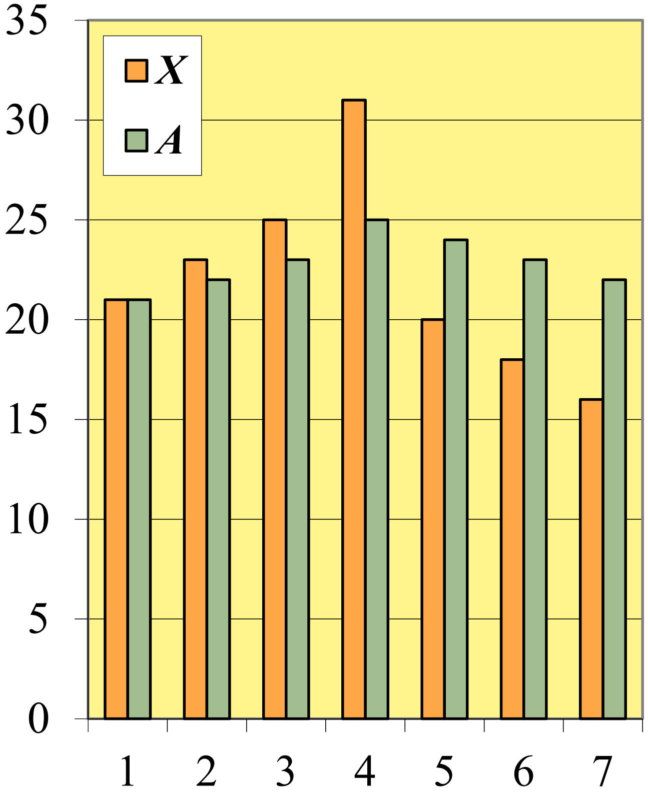
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm ***arrayMax*** executes at most $6n - 1$ primitive operations
  - We say that algorithm ***arrayMax*** "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages
- The $\boldsymbol{i}$-th prefix average of an array $\boldsymbol{X}$ is the average of the first $(\boldsymbol{i}+1)$ elements of $\boldsymbol{X}$:

$$\boldsymbol{A}[\boldsymbol{i}] = (\boldsymbol{X}[0] + \boldsymbol{X}[1] + \ldots + \boldsymbol{X}[\boldsymbol{i}])/(i+1)$$

- Computing the array $\boldsymbol{A}$ of prefix averages of another array $\boldsymbol{X}$ has applications to financial analysis, for example.



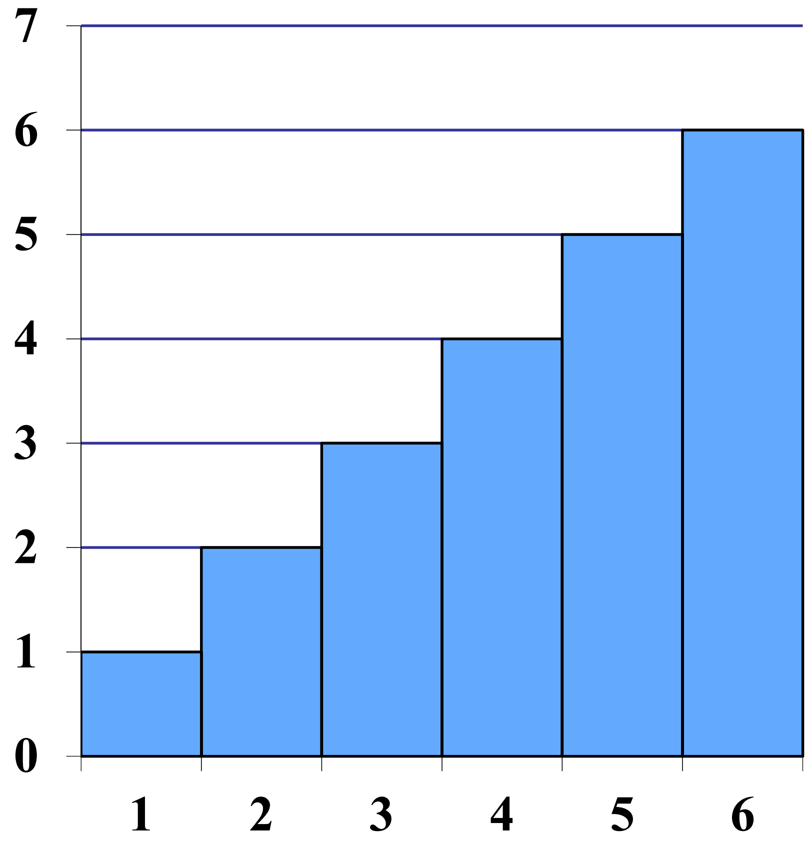
Prefix Averages (v1)

- The following algorithm computes prefix averages by applying the definition

```
Algorithm prefixAverages1(X, n)
    Input array X of n integers
    Output array A of prefix averages of X    #operations
    A | new array of n integers               n
    for i | 0 to n – 1 do                     n
        s | X[0]                              n
        for j | 1 to i do                     1 + 2 + …+ (n – 1)
            s | s + X[j]                      1 + 2 + …+ (n – 1)
        A[i] | s / (i + 1)                    n
    return A                                  1
```

Arithmetic Progression

- The running time of ***prefixAverages1*** is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm ***prefixAverages1*** runs in $O(n^2)$ time



Prefix Averages (v2)

- The following algorithm computes prefix averages efficiently by keeping a running sum

```
Algorithm prefixAverages2(X, n)
    Input array X of n integers
    Output array A of prefix averages of X      #operations
    A ← new array of n integers                      n
    s ← 0                                            1
    for i ← 0 to n − 1 do                            n
        s ← s + X[i]                                 n
        A[i] ← s / (i + 1)                           n
    return A                                         1
```

- Algorithm ***prefixAverages2*** runs in ***O***(***n***) time

Relatives of Big-Oh

- **Big-Omega**
  - f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
- **Big-Theta**
  - f(n) is $\Theta(g(n))$ if there are constants $c_1 > 0$ and $c_2 > 0$ and an integer constant $n_0 \geq 1$ such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for $n \geq n_0$

Intuition for Asymptotic Notation

Big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

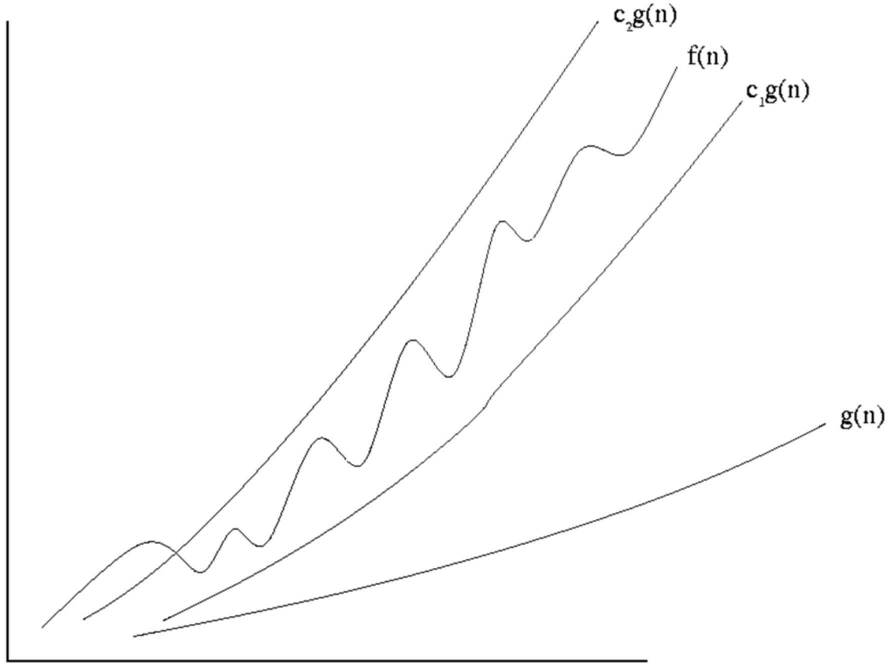
big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

Note that $f(n) \in \Theta\left(g(n)\right) \equiv \left(f(n) \in O\left(g(n)\right) \text{ and } f(n) \in \Omega\left(g(n)\right)\right)$

Definition of Theta

$$f(n) = \theta(g(n))$$



$$\exists c_1, c_2, n_0 > 0 : \forall n \geq n_0, \underbrace{c_1 g(n) \leq f(n) \leq c_2 g(n)}$$

f(n) is sandwiched between $c_1g(n)$ and $c_2g(n)$

Overview

- Motivation
- Definition of Running Time
- Classifying Running Time
- Asymptotic Notation & Proving Bounds
- **Algorithm Complexity vs Problem Complexity**

Time Complexity of an Algorithm

The time complexity of an algorithm is the *largest* time required on *any* input of size n. (Worst case analysis.)

- $O(n^2)$: For any input size $n \geq n_0$, the algorithm takes no more than cn^2 time on every input.
- $\Omega(n^2)$: For any input size $n \geq n_0$, the algorithm takes at least cn^2 time on at least one input.
- $\theta(n^2)$: Do both.

What is the height of tallest person in the class?

Bigger than this?

Need to find only one person who is taller

Smaller than this?

Need to look at every person

Time Complexity of a Problem

The time complexity of a problem is the time complexity of the *fastest* algorithm that solves the problem.

- $O(n^2)$: Provide an algorithm that solves the problem in no more than this time.
  - Remember: for every input, i.e. worst case analysis!
- $\Omega(n^2)$: Prove that no algorithm can solve it faster.
  - Remember: only need one input that takes at least this long!
- $\theta(n^2)$: Do both.

Overview

- Motivation
- Definition of Running Time
- Classifying Running Time
- Asymptotic Notation & Proving Bounds
- Algorithm Complexity vs Problem Complexity