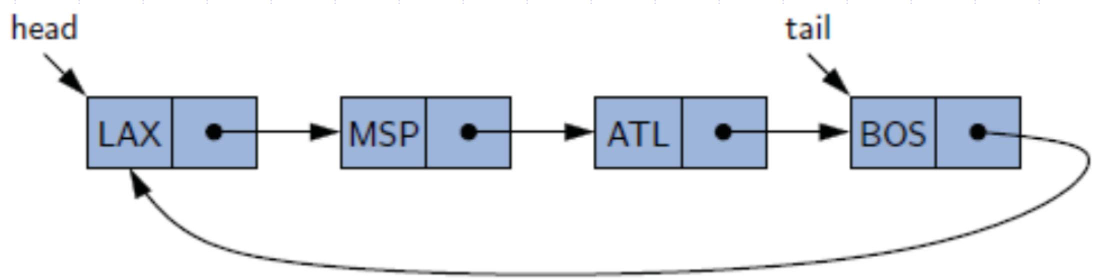
Fundamental Data Structures - Review

- **Circularly Linked Lists**
  - a singularly linked list in which the **next reference of the tail node is set to refer back to the head of the list** (rather than null)
  - good for cyclic-order systems (round-robin scheduler, etc.)



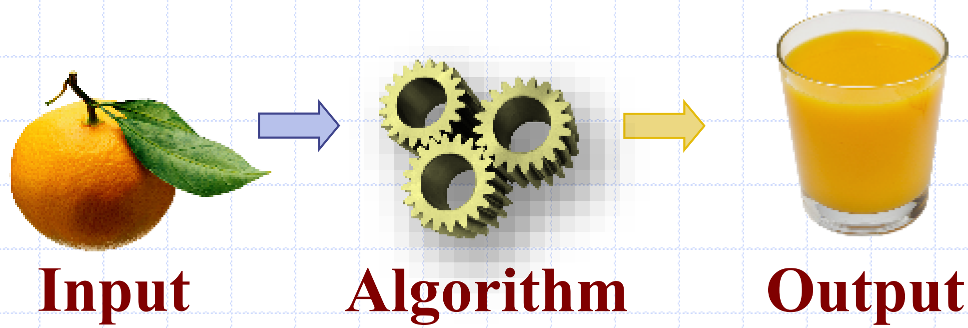
  - No need to keep track of head node
  - Add **rotate**() method to move the first element to the end of the list.

Fundamental Data Structures - Review

- **Equivalency Testing**
  - Arrays – use **Arrays.equals** method for one dimensional arrays, **Arrays.deepEquals** for two dimensional arrays
  - Singly Linked Lists – implement equals method to verify **lengths and equivalency element-by-element** using **equals** method.
- **Cloning Data Structures**
  - Object's *clone* method returns a shallow copy
  - Implement your own *clone* method for a deeper cloning
    - Implement *Cloneable* interface
    - clone individual elements

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Analysis of Algorithms

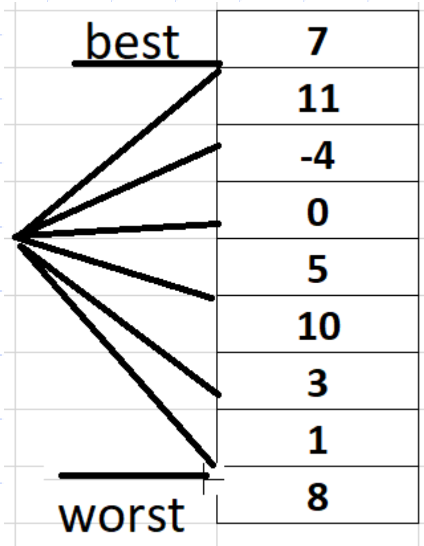


Analysis of Algorithms

- Recall:
  - a ***data structure*** is a systematic way of organizing and accessing data.
  - an ***algorithm*** is a step-by-step procedure for performing some task in a finite amount of time.
- To be able to classify some data structures and algorithms as "good," we must have **precise ways of analyzing them**.
- **Running times** of algorithms and data structure operations is is a natural measure of "goodness", since **time is a precious resource**.
- We are interested in characterizing an algorithm's **running time as a function of the input size**.

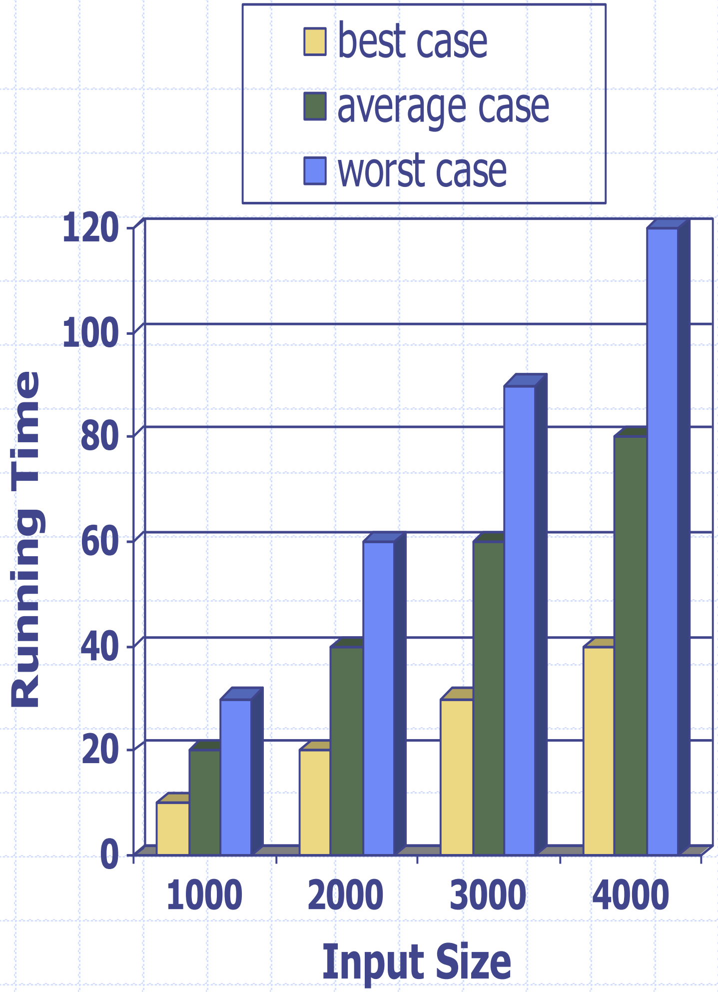
Running Time Scenarios

- **Best-case**: the case with the **shortest** running time
- **Worst-case**: the case with the **longest** running time
- **Average-case**: this is the running time used by the algorithm **averaged over all possible inputs**.
- Example: do a linear search in an array - the element you are searching for could be at:
  - the beginning of the array
  - the end of the array
  - somewhere between



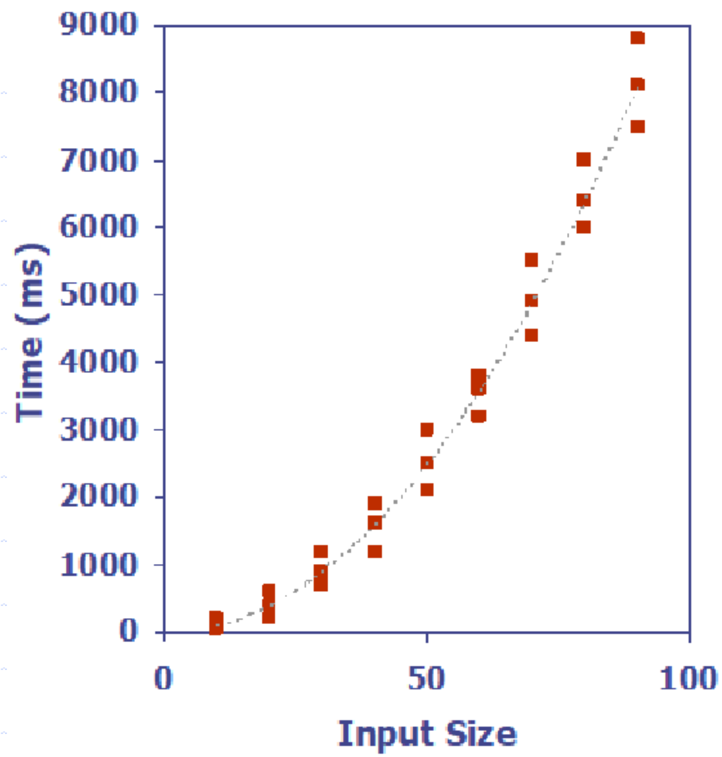
Running Time

- 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
  - Crucial to applications such as games, finance and robotics



Experimental Studies

- Write a program implementing the algorithm.
- Run the program with inputs of varying size and composition, noting the time needed:
- Plot the results.




```
long startTime = System.currentTimeMillis( );     // record the starting time
/* (run the algorithm) */
long endTime = System.currentTimeMillis( );       // record the ending time
long elapsed = endTime − startTime;               // compute the elapsed time
```

Experimental Studies - Example

- Consider two algorithms for constructing long strings in Java (StringExperiment.java):

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135

Table 4.1: Results of timing experiment on the methods from Code Fragment 4.2.

- There is an **order of magnitude difference** in the growth of the running times!

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** (either in the form of an actual code fragment, or language-independent pseudocode) **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**.

Theoretical Analysis

- Things to review and/or define:
  - Pseudocode
  - Random Access Machine (RAM) model
  - Counting Primitive operations
  - Seven important functions to express the growth rate of algorithm's running time
  - Big O notation to give an upper bound on the growth rate of a function

Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues
- Example:
- Algorithm **sum**(*arr*):

  Input: an array of integers *arr*

  Output: the sum of array elements *sum*

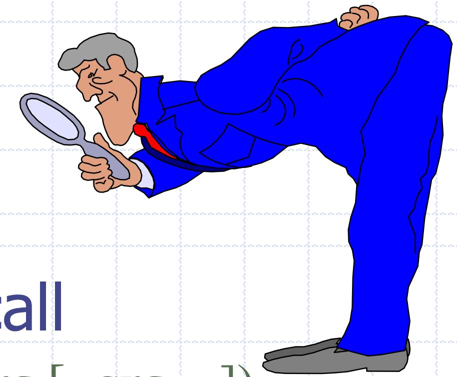
  sum = 0

  for i=0 to arr.length-1 ***do***

      sum ← *sum* + *arr[i]*

  return *sum*

Pseudocode Details



- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***…])

  Input …

  Output …
- Method call

  method (***arg*** [, ***arg***…])
- Return value

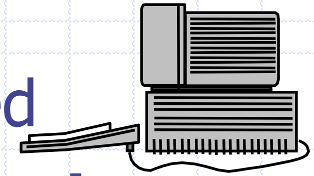
  return ***expression***
- Expressions:

  $\leftarrow$ Assignment

  $=$ Equality testing

  n^2 Superscripts and other mathematical formatting allowed

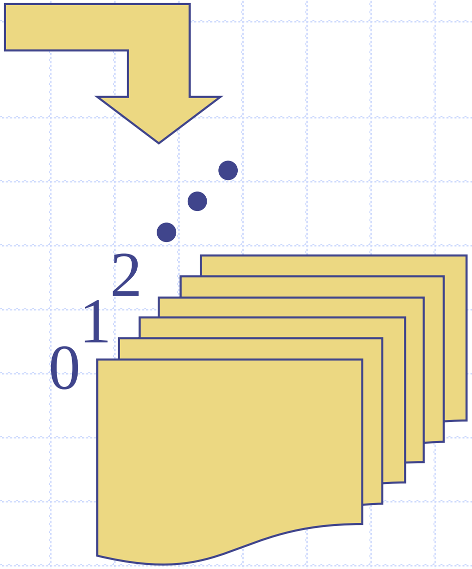
The Random Access Machine (RAM) Model



- Algorithms can be measured in a **machine-independent way** using the Random Access Machine (RAM) model.

A RAM consists of:

- A CPU
- A potentially unbounded bank of memory cells, each of which can hold an arbitrary number or character
- Memory cells are numbered and **accessing any cell in memory takes unit time**.



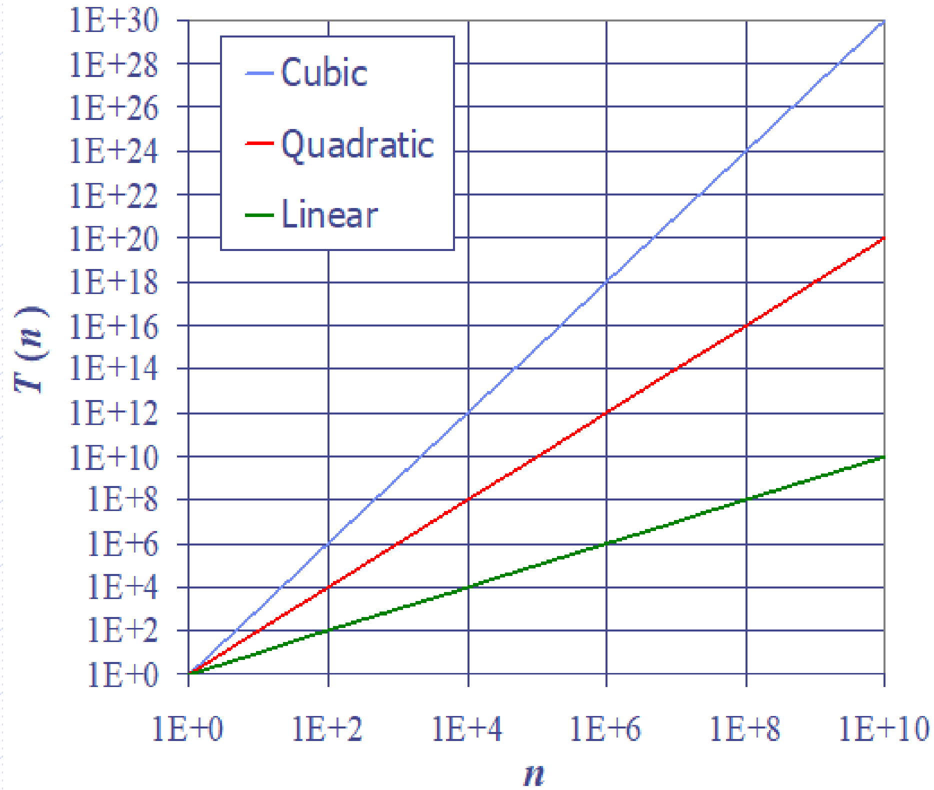
The Random Access Machine (RAM) Model

- RAM is an abstraction that **allows us to compare algorithms on the basis of performance**.
- This model assumes **a single processor**.
- In the RAM model, **instructions are executed one after the other**, with **no concurrent operations**.
- The assumptions made in the RAM model to accomplish this are:
  - Each **simple operation** takes **1** time step.
  - Loops and subroutines are not simple operations.
  - Each **memory access** takes one time step, and there is **no shortage of memory**.
- For any given problem the **running time of an algorithms is assumed to be the number of time steps/units**.
- The space used by an algorithm is assumed to be the **number of RAM memory cells**.

https://courses.cs.washington.edu/courses/csep521/97au/notes/lect1-scribe/node3.html

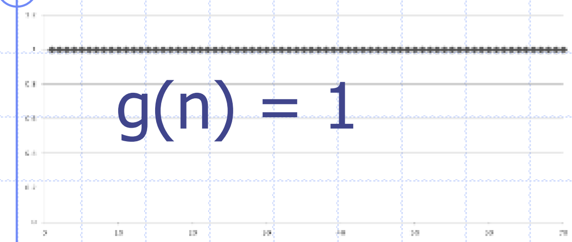
Seven Important Functions

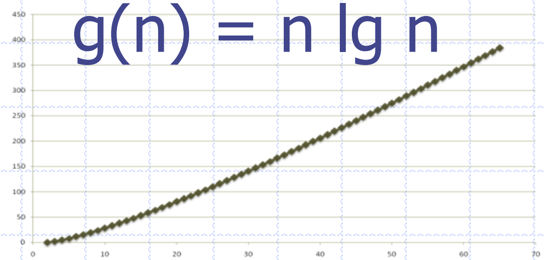
- Seven functions that often appear in algorithm analysis:
  - **Constant** ≈ 1
  - **Logarithmic** $\approx \log n$
  - **Linear** $\approx n$
  - **N-Log-N** $\approx n \log n$
  - **Quadratic** $\approx n^2$
  - **Cubic** $\approx n^3$
  - **Exponential** $\approx 2^n$
- In a log-log chart, the slope of the line corresponds to the growth rate

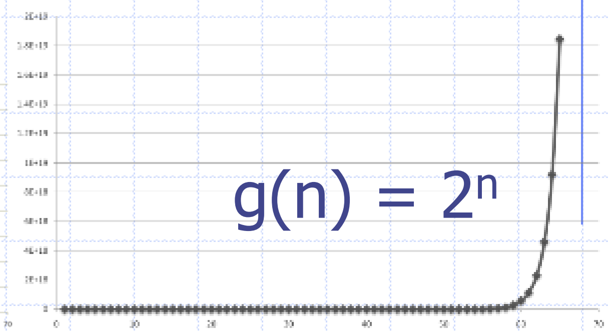


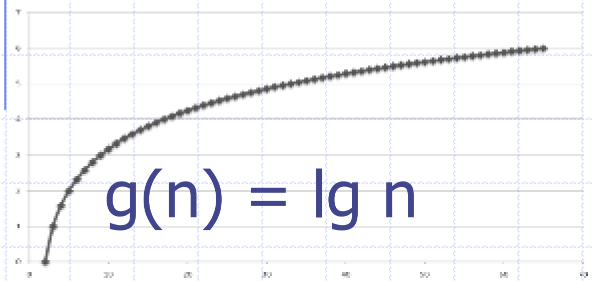
Functions Graphed Using "Normal" Scale

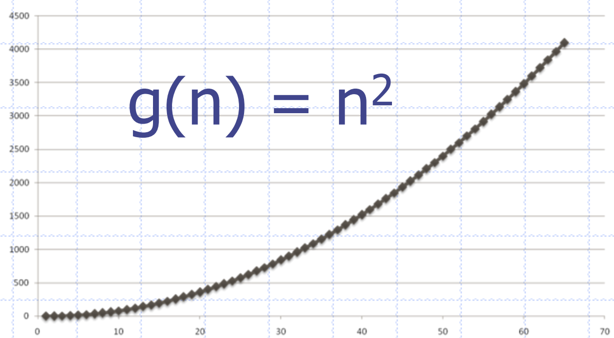
Slide by Matt Stallmann included with permission.

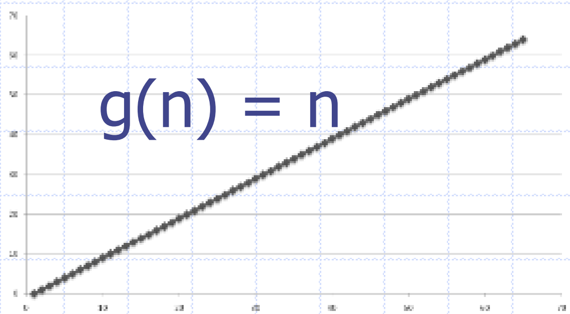


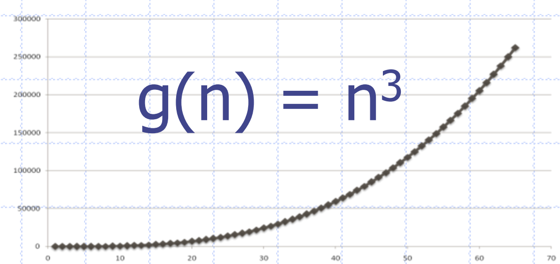












Primitive Operations

- Primitive operations are:
  - Basic computations performed by an algorithm
  - Identifiable in pseudocode
  - Largely independent from the programming language
  - Exact definition not important (we will see why later)
  - Assumed to take a **constant amount of time** in the RAM model

- Examples:
- Evaluating an **expression**
- **Assigning** a value to a variable
- **Indexing** into an array
- **Calling** a method
- **Returning** from a method
- **Comparing** two numbers
- Following an object reference

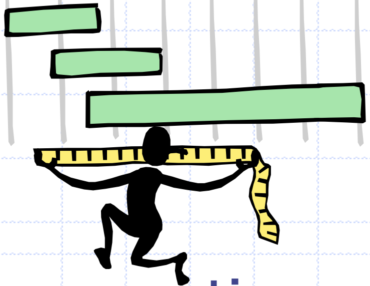
Counting Primitive Operations

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

```
1 /** Returns the maximum value of a nonempty array of numbers. */
2 public static double arrayMax(double[ ] data) {
3   int n = data.length;
4   double currentMax = data[0];        // assume first entry is biggest (for now)
5   for (int j=1; j < n; j++)           // consider all other entries
6     if (data[j] > currentMax)         // if data[j] is biggest thus far...
7       currentMax = data[j];           // record it as the current max
8   return currentMax;
9 }
```

- Step 3: 2 ops, 4: 2 ops, 5: 2n ops, 6: 2n ops, 7: 0 to n ops, 8: 1 op

Estimating Running Time



- Algorithm arrayMax executes $5n + 5$ primitive operations in the worst case, $4n + 5$ in the best 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

$$a\,(4n + 5) \le T(n) \le b(5n + 5)$$

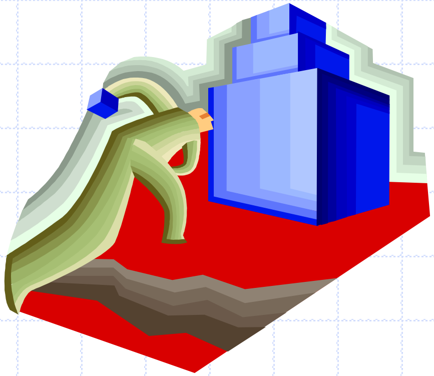
- Hence, the running time $T(n)$ is bounded by two linear functions.

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $\boldsymbol{T(n)}$ by a constant factor, but
  - Does not alter the growth rate of $\boldsymbol{T(n)}$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm arrayMax
- Constant factors don't matter
- For example, c*$\mathbf{n^2}$ and $\mathbf{n^2}$ have the same growth rate
  - both quadruple when **n** is doubled,

    $c*(2n)^2 = \mathbf{4}*c*n^2$ and

    $(2n)^2 = \mathbf{4}*n^2$



Why Growth Rate Matters

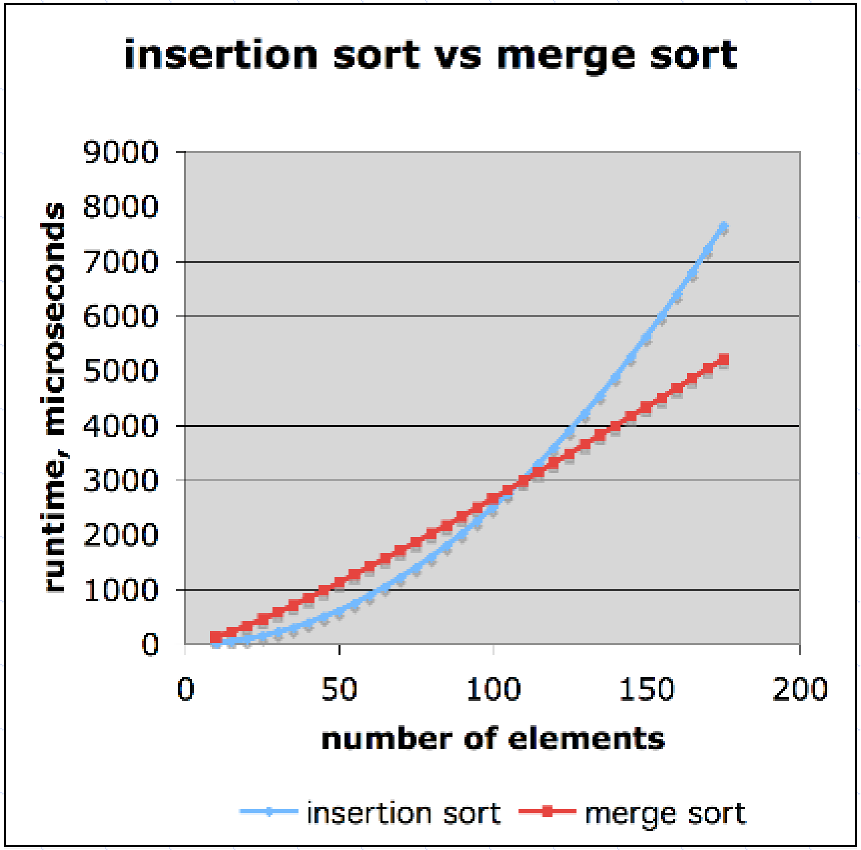
Slide by Matt Stallmann included with permission.

if runtime is...	time for **n + 1**	time for **2 n**	time for **4 n**
$c \lg n$	$c \lg (n + 1)$	$c (\lg n + 1)$	$c(\lg n + 2)$
$c\ n$	$c\ (n + 1)$	$2c\ n$	$4c\ n$
$c\ n \lg n$	$\sim c\ n \lg n + c\ n$	$2c\ n \lg n + 2cn$	$4c\ n \lg n + 4cn$
$c\ n^2$	$\sim c\ n^2 + 2c\ n$	$\mathbf{4c\ n^2}$	$16c\ n^2$
$c\ n^3$	$\sim c\ n^3 + 3c\ n^2$	$8c\ n^3$	$64c\ n^3$
$c\ 2^n$	$c\ 2^{n+1}$	$c\ 2^{2n}$	$c\ 2^{4n}$

runtime quadruples when problem size doubles

Slide by Matt Stallmann included with permission.

Comparison of Two Algorithms



insertion sort is

n^2 / 4

merge sort is

2 n lg n

sort a million items?

insertion sort takes roughly 70 hours

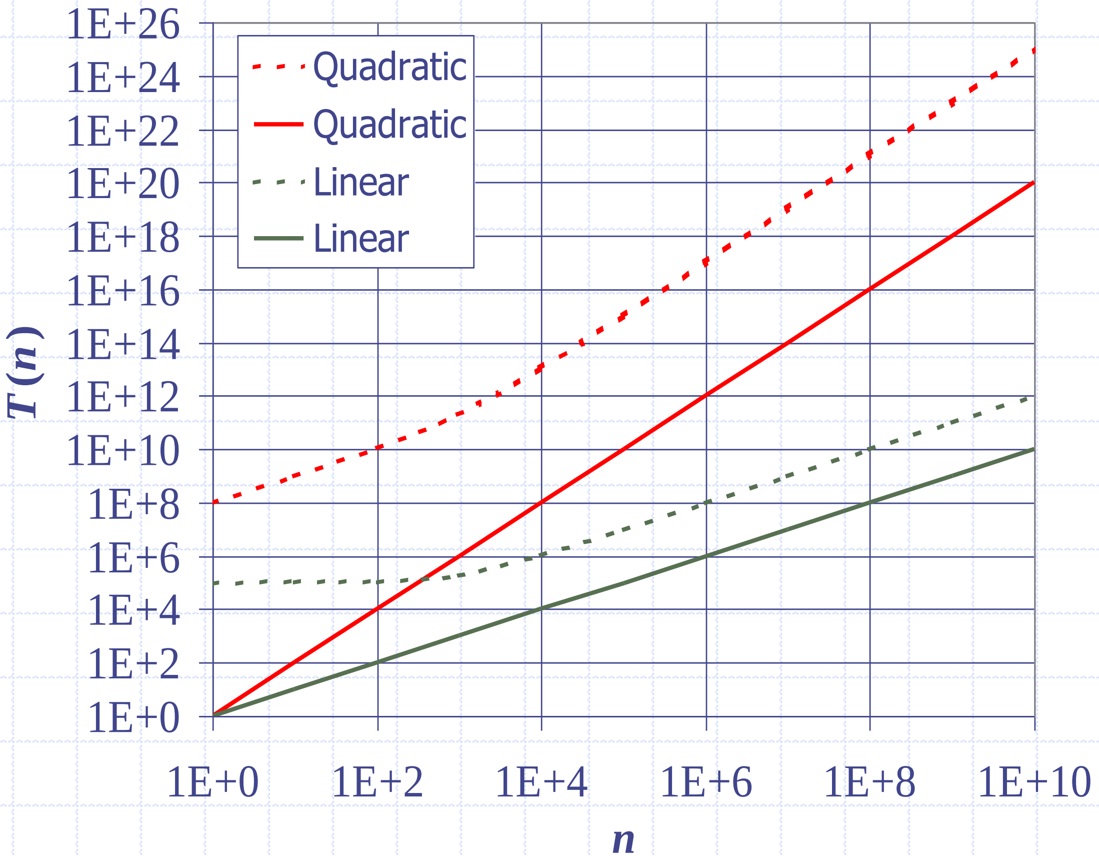
while

merge sort takes roughly 40 seconds

This is a slow machine, but if 100 x as fast then it's 40 minutes versus less than 0.5 seconds

Constant Factors

- 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



Big-Oh Notation

- In many situations we would like to **give an upper bound on the growth rate of a function f(n)**.
- Example: **sequential search in arrays**
  - if the target value is just the first element, then the running time is a constant function.
  - however, in worst-case scenario, the running time of sequential search algorithm f(n) **grows at most as fast as a linear function**.
- We can say that the **growth rate of running time function f(n)**, **is bounded above by a linear function.**

Big-Oh Notation

- Another example:
  - Let the growth rate of running time function be:

    $$f(n) = 10n + 10$$
  - We know 10n+10 > n, so f(n) is not bounded by n.
  - Also, 10n + 10 > 10n, so fn() is not bounded by 10n.
  - However, 10n +10 < 11n for n > 10
- In this case, we say that f(n) is **asymptotically bounded above** by 11n.
- This means that the growth rate of $f(n)$ is no more than the growth rate of function **11n.**
- The Big-Oh notation can express well just that.

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 $n \geq n_0$

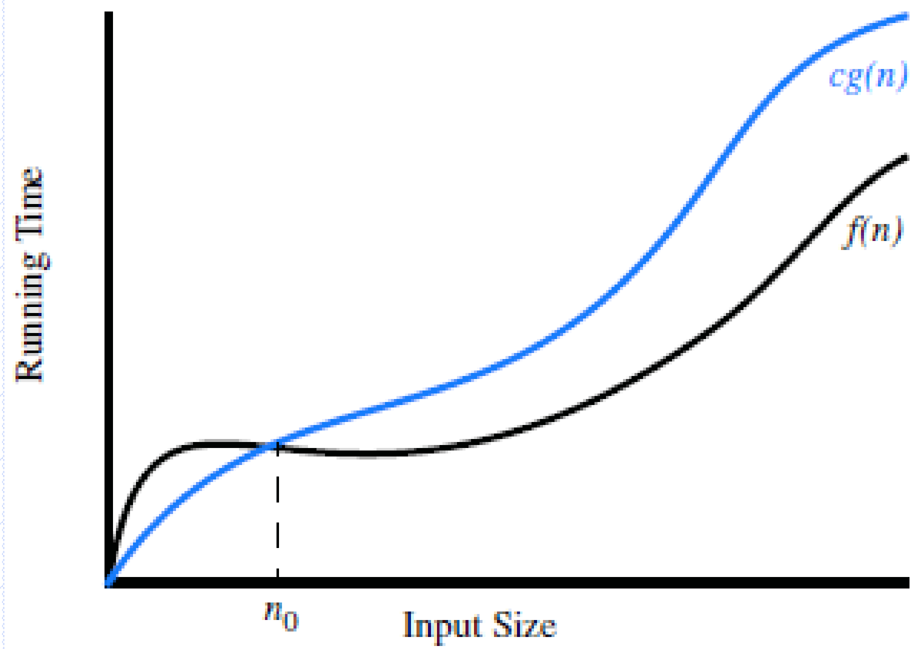
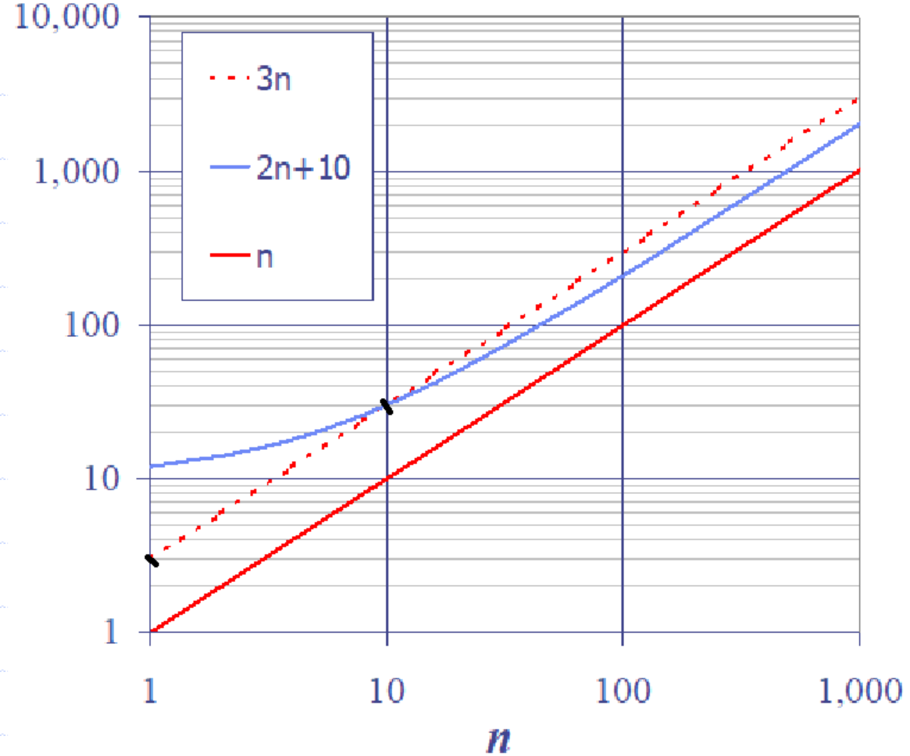


Figure 4.5: Illustrating the "big-Oh" notation.

- The function $f(n)$ is $O(g(n))$, since $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

Big-Oh Notation

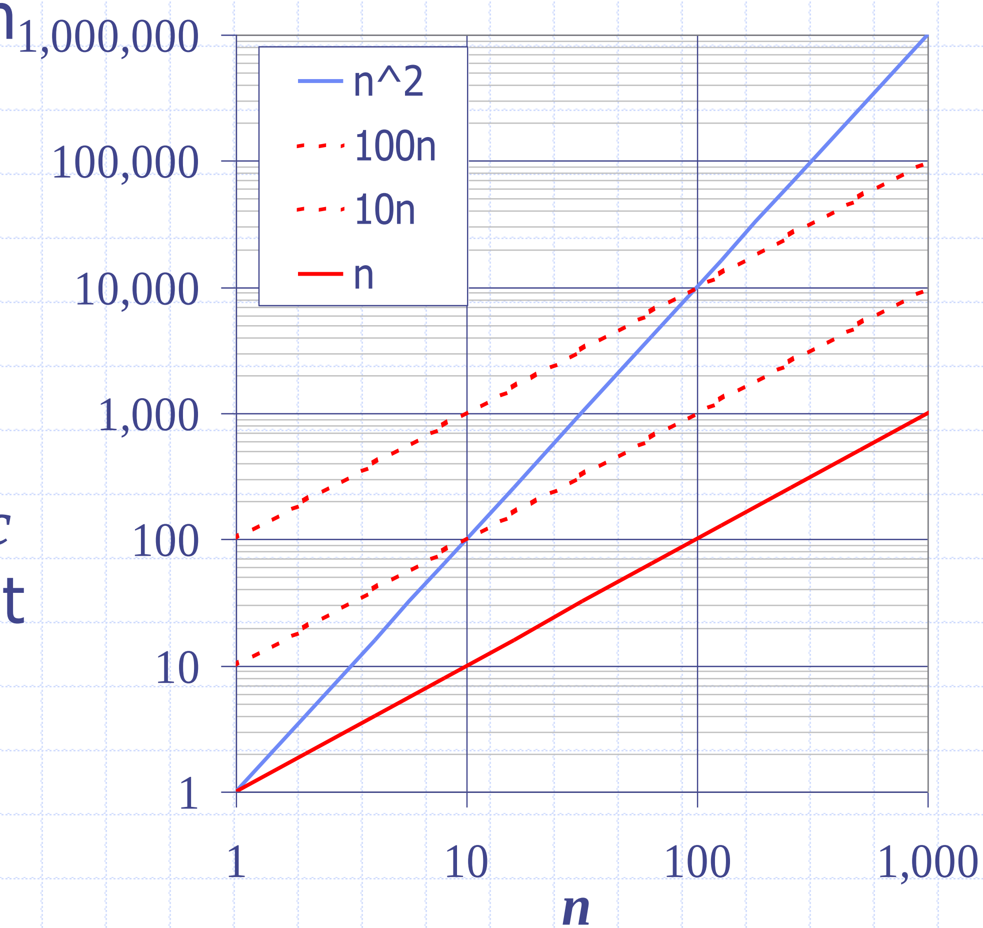
- Example: $2n + 10$ is $O(n)$
- We have:
  - **g(n)** = n
  - **f(n)** = 2n + 10
- We need to find **c** and n_0 s.t.
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$



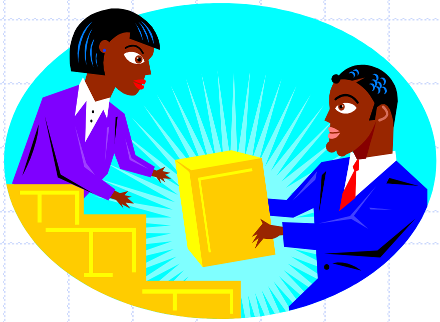
  - Pick $c = 3$, so that the inequality holds.
    - *this gives* $n \geq 10$
  - Picking $n_0 = 10$, this inequality will hold for $n \geq n_0$.

Big-Oh Example

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



More Big-Oh Examples



- $7n - 2$ is $O(n)$

need to find $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq \mathbf{c\,n}$ for $n \geq n_0$

$7n - 2 \leq \mathbf{7}n$ for any $\mathbf{n}$. By picking $\mathbf{c} = 7$, this is true for $\mathbf{n_0} = 1$

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

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

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

this is true for $c = 4$ and $n_0 = 21$

- $3 \log n + 5$

$3 \log n + 5$ is $O(\log n)$

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

this is true for $c = 8$ and $n_0 = 2$

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 is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - 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

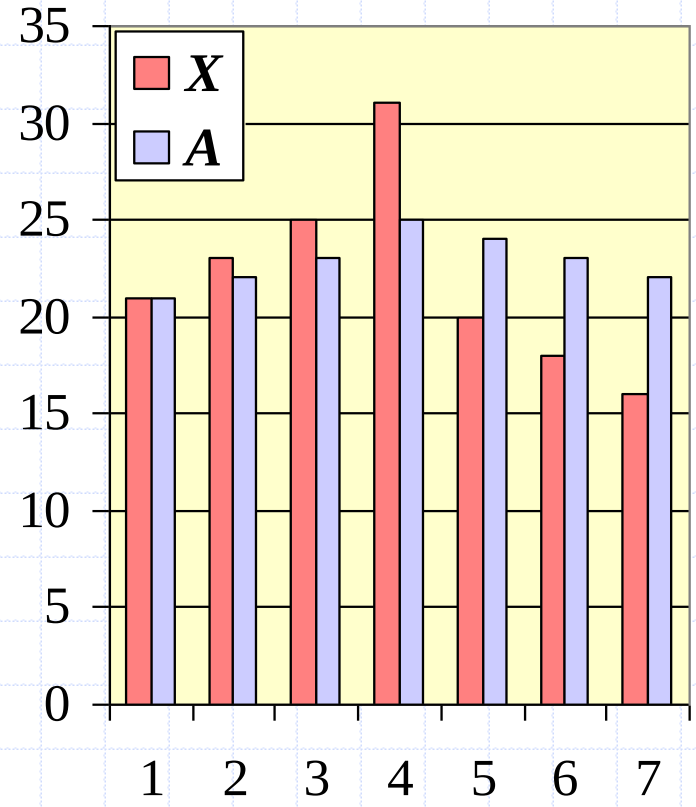
- The asymptotic analysis of an algorithm **determines the running time in big-Oh notation**.
- To perform the asymptotic analysis:
  - We find the **worst-case number of primitive operations** executed as a function of the input size.
  - We express this function with **big-Oh notation**.
- Example:
  - We 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 ***i*-th prefix average of an array** X is average of the first $(i + 1)$ elements of X:

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

- Computing the array A of prefix averages of another array X has applications to financial analysis



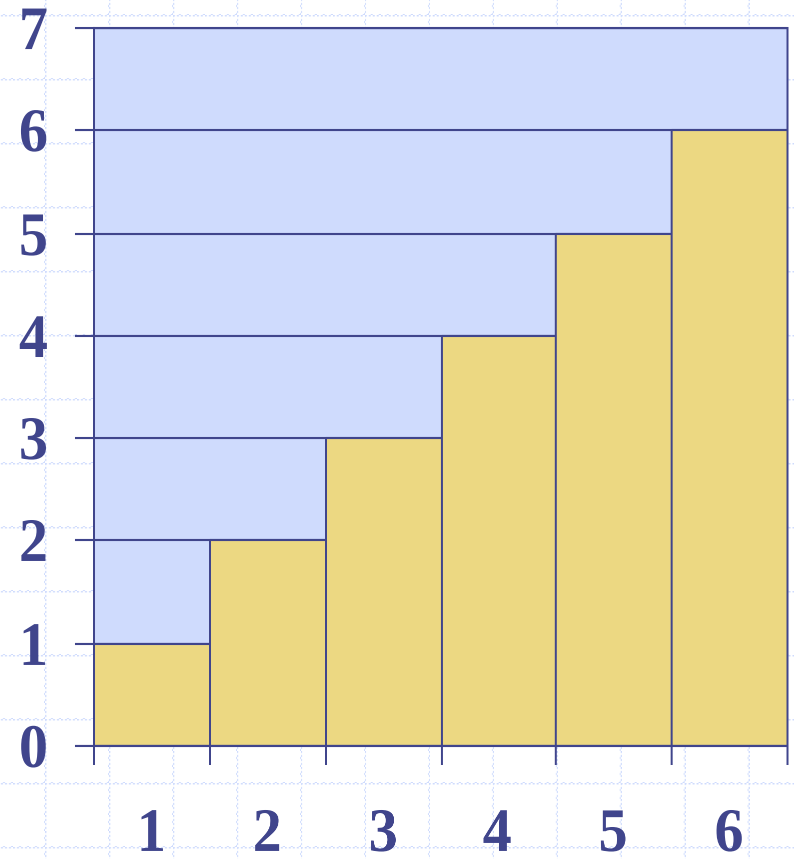
Prefix Averages (Quadratic)

The following algorithm computes prefix averages in **quadratic time** by applying the definition

```
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
public static double[ ] prefixAverage1(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n];                    // filled with zeros by default
  for (int j=0; j < n; j++) {
    double total = 0;                             // begin computing x[0] + ... + x[j]
    for (int i=0; i <= j; i++)
      total += x[i];
    a[j] = total / (j+1);                         // record the average
  }
  return a;
}
```

Arithmetic Progression

- The running time of prefixAverage1 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 prefixAverage1 runs in $O(n^2)$ time



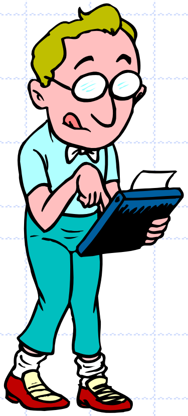
Prefix Averages 2 (Linear)

- The following algorithm uses a **running summation** to improve the efficiency

```
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
public static double[ ] prefixAverage2(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n];                // filled with zeros by default
  double total = 0;                           // compute prefix sum as x[0] + x[1] + ...
  for (int j=0; j < n; j++) {
    total += x[j];                            // update prefix sum to include x[j]
    a[j] = total / (j+1);                     // compute average based on current sum
  }
  return a;
}
```

Algorithm prefixAverage2 runs in $O(n)$ time!

Math you need to Review



- Summations
- Powers
- Logarithms
- Proof techniques
- Basic probability

- Properties of powers:

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

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

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

  $b = a^{\log_a b}$

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

- Properties of logarithms:

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

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

  $\log_b xa = a\log_b x$

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

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

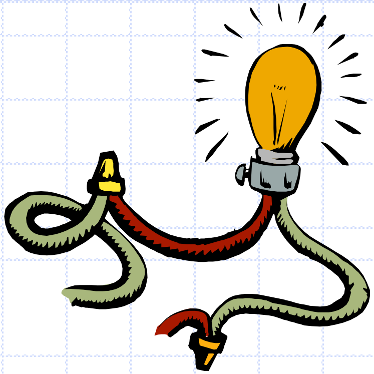
$$f(n) \geq c\, g(n) \text{ for } n \geq n_0$$

big-Theta

- $f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that

$$c'g(n) \leq f(n) \leq c''g(n) \text{ for } n \geq n_0$$

Intuition for Asymptotic Notation



big-Oh

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

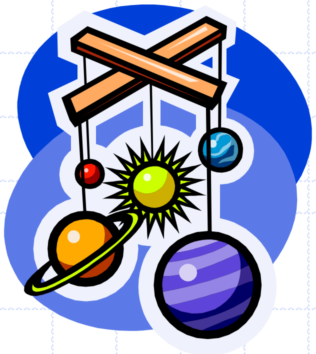
big-Omega

- 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)

Example Uses of the Relatives of Big-Oh



- **$5n^2$ is $\Omega(n^2)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c\, g(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

- **$5n^2$ is $\Omega(n)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $\mathrm{f}(n) \geq c\, g(n)$ for $n \geq n_0$

  let $c = 1$ and $n_0 = 1$

- **$5n^2$ is $\Theta(n^2)$**

  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $\mathrm{f}(n) \leq c\, g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$