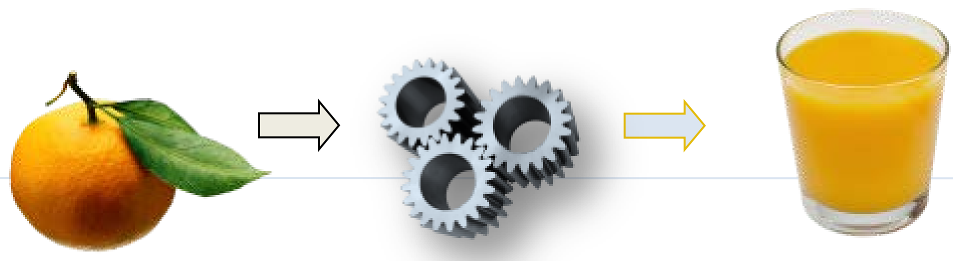
ANALYSIS OF ALGORITHMS



Input **Algorithm** **Output**

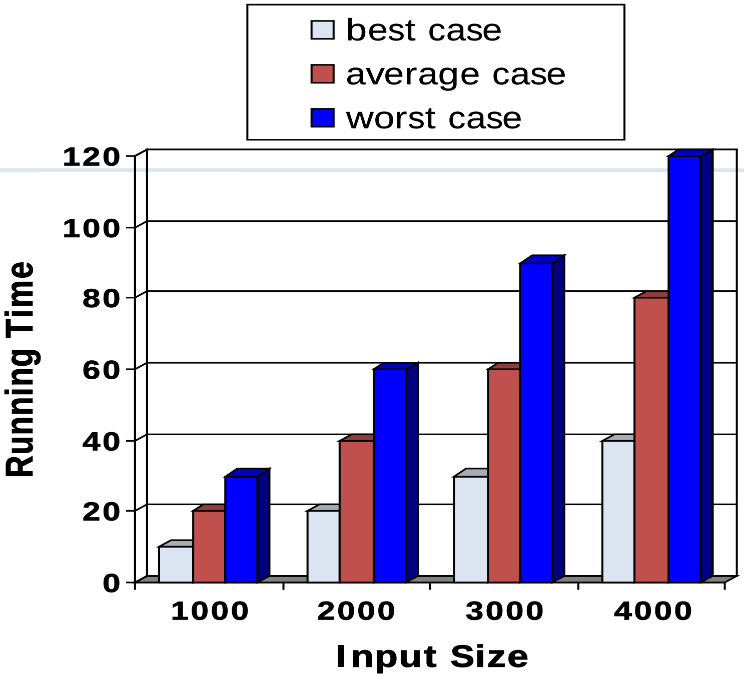
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

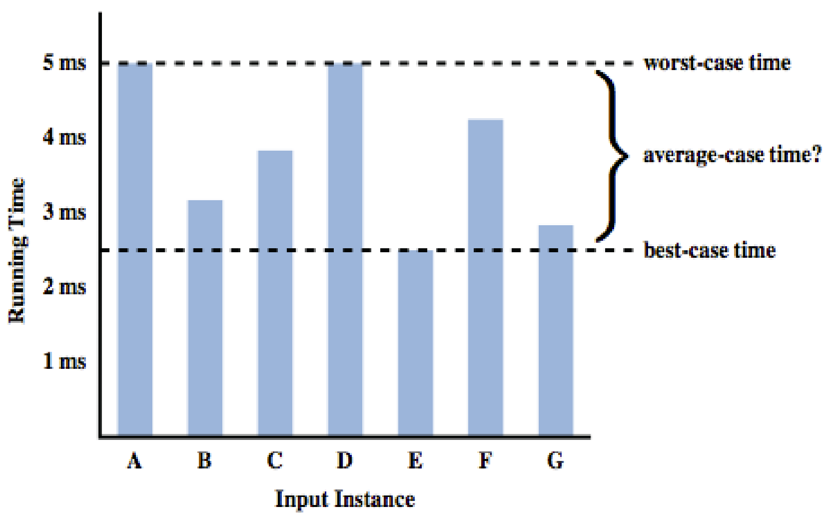
EVALUATING PROGRAM EFFICIENCY

- What's the best way to program a solution to a problem ?
- Efficiency in terms of
  - Time (running time)
  - Space (memory requirements)
  - Resources (Input/Output such as disk I/O)
  - Energy consumption
- Most analysis focuses on time efficiency

RUNNING TIME

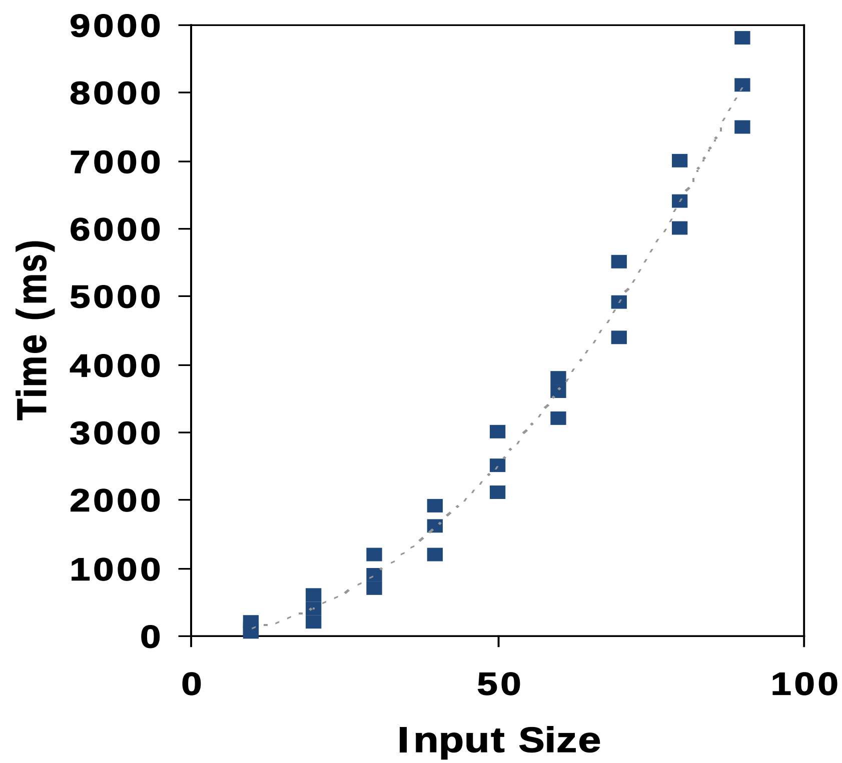
- The running time of an algorithm typically grows with the **input size**.
- Average case time is often difficult to determine.
- We focus on the **worst case running time.**
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics





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
```

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.
- Measured times reported by the system may likely vary from trial to trial, even on the same machine and input.
  - Processes share CPU and memory.
- In order to compare two algorithms, the same hardware and software environments must be used

EXAMPLE

```
/** Uses repeated concatenation to compose a String with n copies of character c. */
public static String repeat1(char c, int n) {
  String answer = "";
  for (int j=0; j < n; j++)
    answer += c;
  return answer;
}

/** Uses StringBuilder to compose a String with n copies of character c. */
public static String repeat2(char c, int n) {
  StringBuilder sb = new StringBuilder( );
  for (int j=0; j < n; j++)
    sb.append(c);
  return sb.toString( );
}
```

Code Fragment 4.2: Two algorithms

repeat('*', 40)

"**"

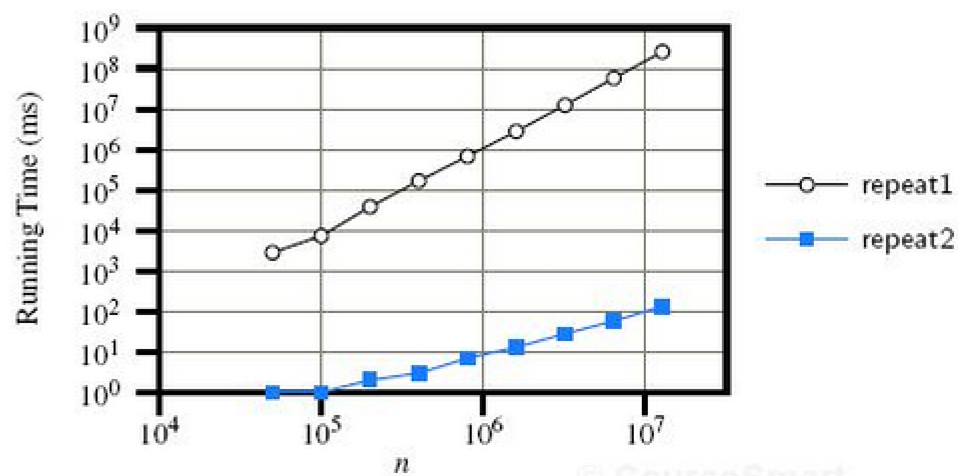
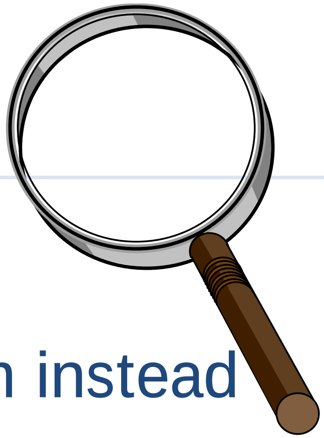


Figure 4.1: Chart of the results of the timing experiment from Code Fragment 4.2, displayed on a log-log scale. The divergent slopes demonstrate an order of magnitude difference in the growth of the running times.

GOALS OF ANALYZING THE EFFICIENCY OF ALGORITHMS

1. Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.
2. Is performed by studying a high- level description of the algorithm without need for implementation.
3. Takes into account all possible inputs.

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 <u>evaluate the speed of an algorithm independent of the hardware/software environment</u>

F.Y.I.: PRIMITIVE OPERATIONS

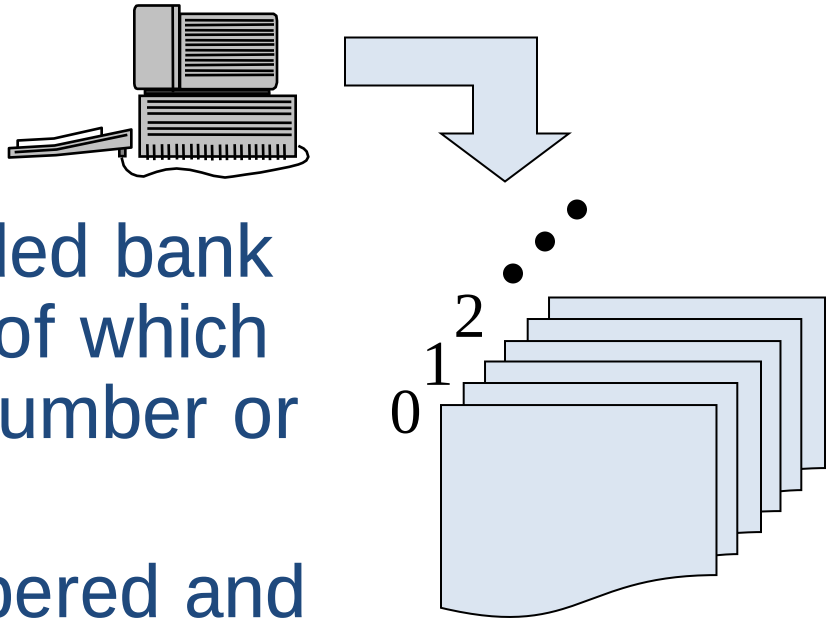
- Basic computations performed by an algorithm
- Identifiable in **pseudocode**
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the **RAM model**

- Examples:
  - Assigning a value to a variable
  - Following an object reference
  - Performing an arithmetic operation (for example, adding two numbers)
  - Comparing two numbers
  - Accessing a single element of an array by index
  - Calling a method
  - Returning from a method

F.Y.I.: THE RANDOM ACCESS MACHINE (RAM)

A RAM consists of

- A CPU
- An 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



F.Y.I.: 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 of pseudocode (for the mathematical game fizz buzz):

Fortran style pseudo code

```
program fizzbuzz
Do i = 1 to 100
    set print_number to true
    If i is divisible by 3
        print "Fizz"
        set print_number to false
    If i is divisible by 5
        print "Buzz"
        set print_number to false
    If print_number, print i
    print a newline
end do
```

Pascal style pseudo code

```
procedure fizzbuzz
For i := 1 to 100 do
    set print_number to true;
    If i is divisible by 3 then
        print "Fizz";
        set print_number to false;
    If i is divisible by 5 then
        print "Buzz";
        set print_number to false;
    If print_number, print i;
    print a newline;
end
```

C style pseudo code:

```
void function fizzbuzz
For (i = 1; i <= 100; i++) {
    set print_number to true;
    If i is divisible by 3
        print "Fizz";
        set print_number to false;
    If i is divisible by 5
        print "Buzz";
        set print_number to false;
    If print_number, print i;
    print a newline;
}
```

http://en.wikipedia.org/wiki/Pseudocode

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

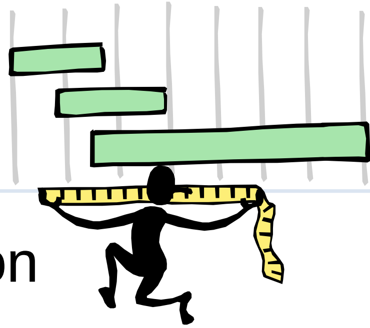
COUNTING PRIMITIVE OPERATIONS

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

```
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



Focus on the growth rate of the running time as a function of the input size n, taking a "big-picture" approach.

- 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

$$a(4n + 5) \leq T(n) \leq b(5n + 5)$$

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

GROWTH RATE OF RUNNING TIME

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$

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

- The linear growth rate of the running time $T(n)$ is an <u>intrinsic property of algorithm</u> arrayMax

Slide by Matt Stallmann included with permission.

WHY GROWTH RATE MATTERS

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	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
$c\ n^2$	$\sim c\ n^2 + 2c\ n$	**$4c\ n^2$**	$16c\ n^2$
$c\ n^3$	$\sim c\ n^3 + 3c\ n^2$	$8c\ n^3$	$64c\ n^3$
$c\ 2^n$	$c\ 2^{n+1}$	$c\ 2^{2n}$	$c\ 2^{4n}$

runtime quadruples when problem size doubles

F.Y.I.: 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 ≈ b^n

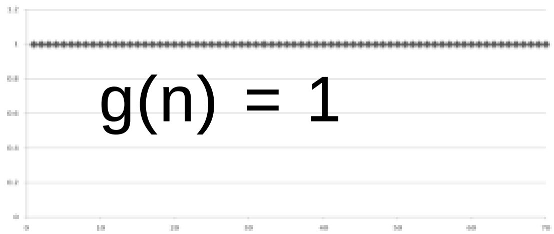
$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d$$

Polynomials

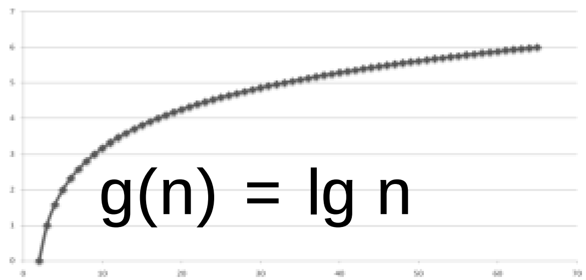
- In a log-log chart, the slope of the line corresponds to the growth rate

log-log chart

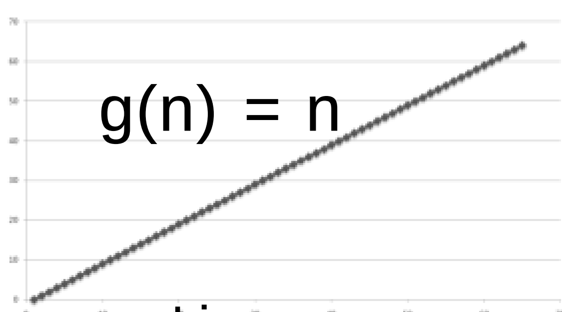
THE SEVEN FUNCTIONS GRAPHED USING "NORMAL" SCALE



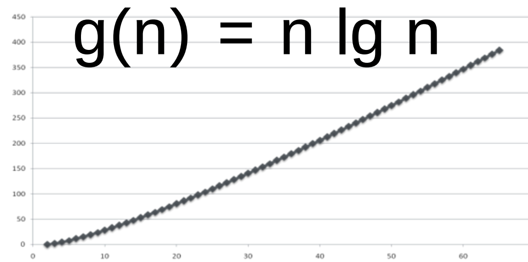
Constant



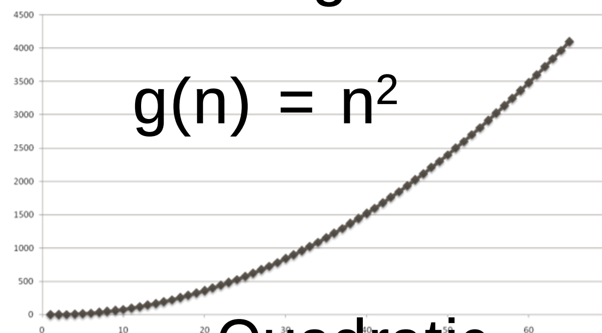
Logarithmic



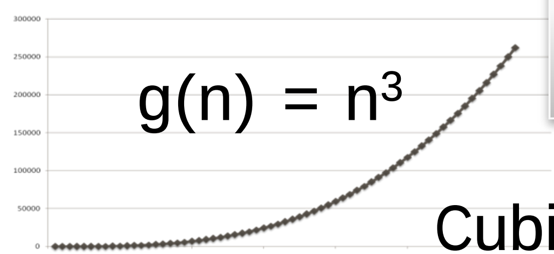
Linear



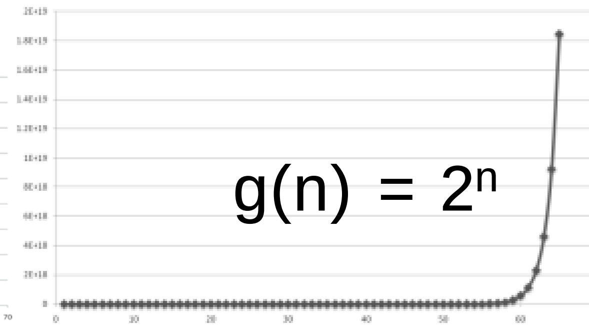
N-Log-N



Quadratic

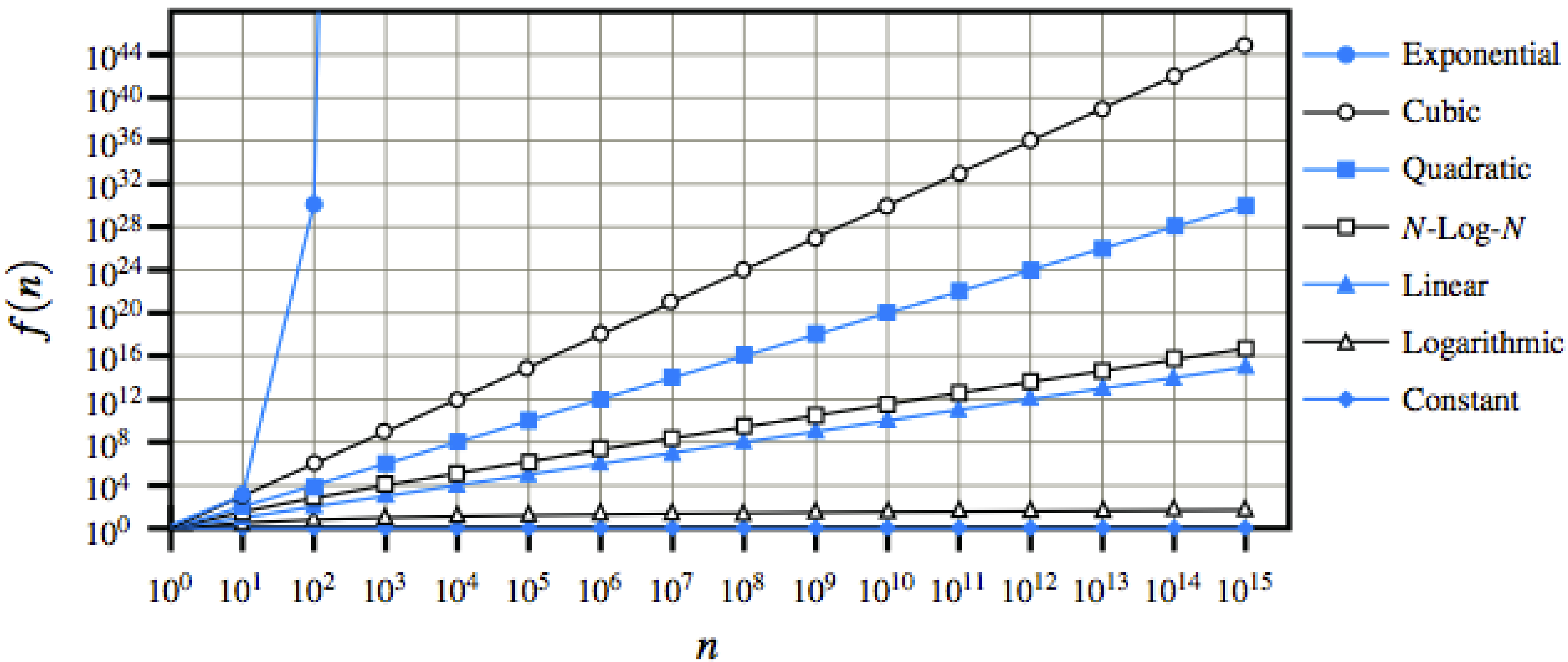


Cubic



Exponential

Slide by Matt Stallmann included with permission.



log-log chart

F.Y.I.: SUMMATIONS

Running times of loops with increasing terms

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b)$$

where a and b are integers and $a \leq b$

Proposition 4.3: *For any integer $n \geq 1$, we have:*

$$1 + 2 + 3 + \cdots + (n-2) + (n-1) + n = \frac{n(n+1)}{2}.$$

Loop for which each iteration takes a multiplicative factor longer than the previous one.

Proposition 4.5: *For any integer $n \geq 0$ and any real number a such that $a > 0$ and $a \neq 1$, consider the summation*

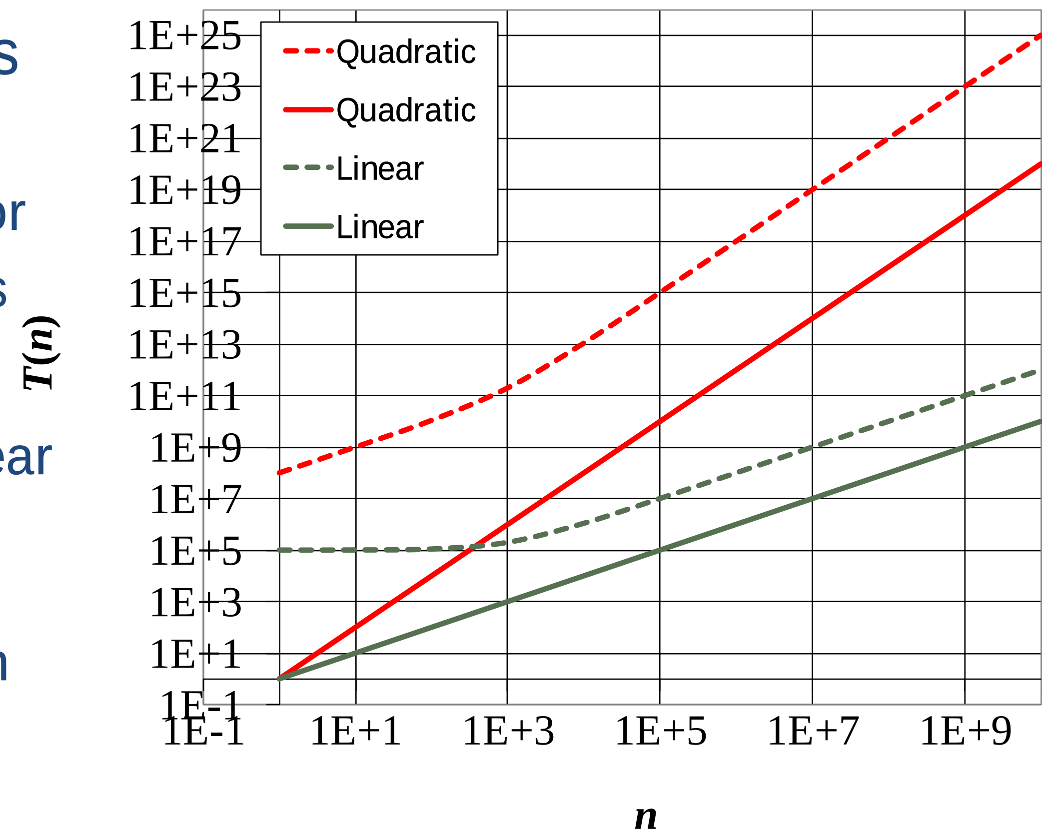
$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n$$

(remembering that $a^0 = 1$ if $a > 0$). This summation is equal to

$$\frac{a^{n+1} - 1}{a - 1}.$$

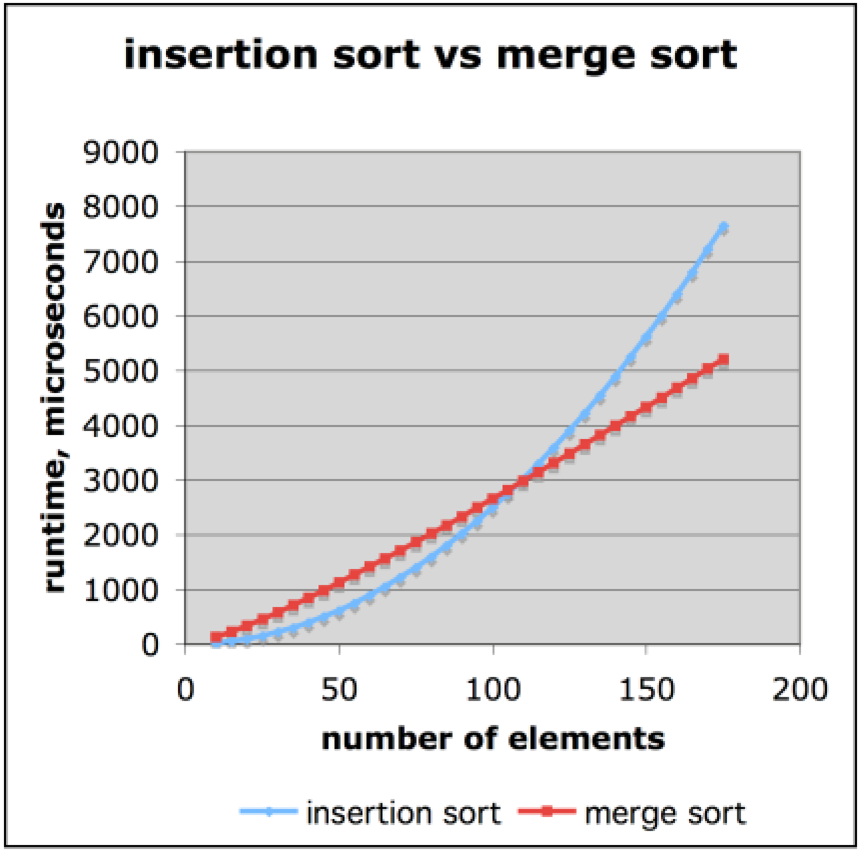
CONSTANT FACTORS

- 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



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

BIG-OH NOTATION

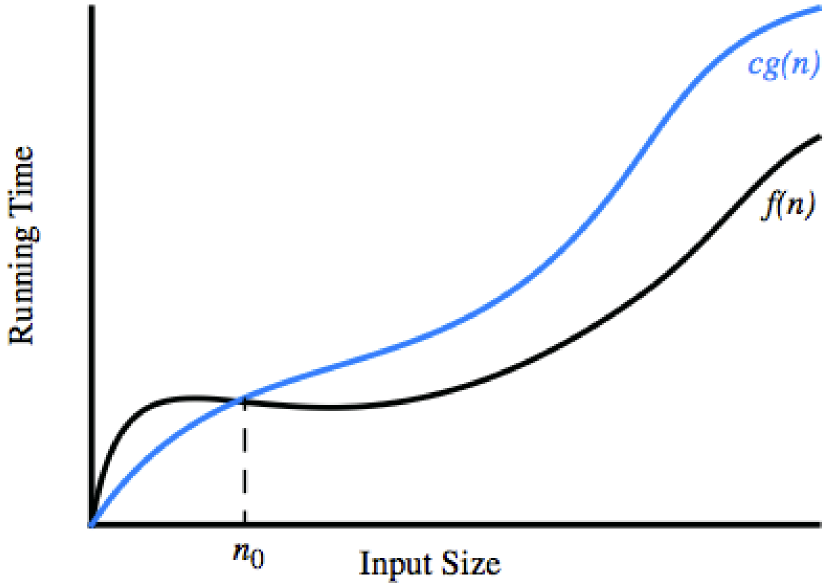
Focus on the growth rate of the running time as a function of the input size *n*, taking a "big-picture" approach.

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \quad \text{for} \quad n \geq n_0.$$

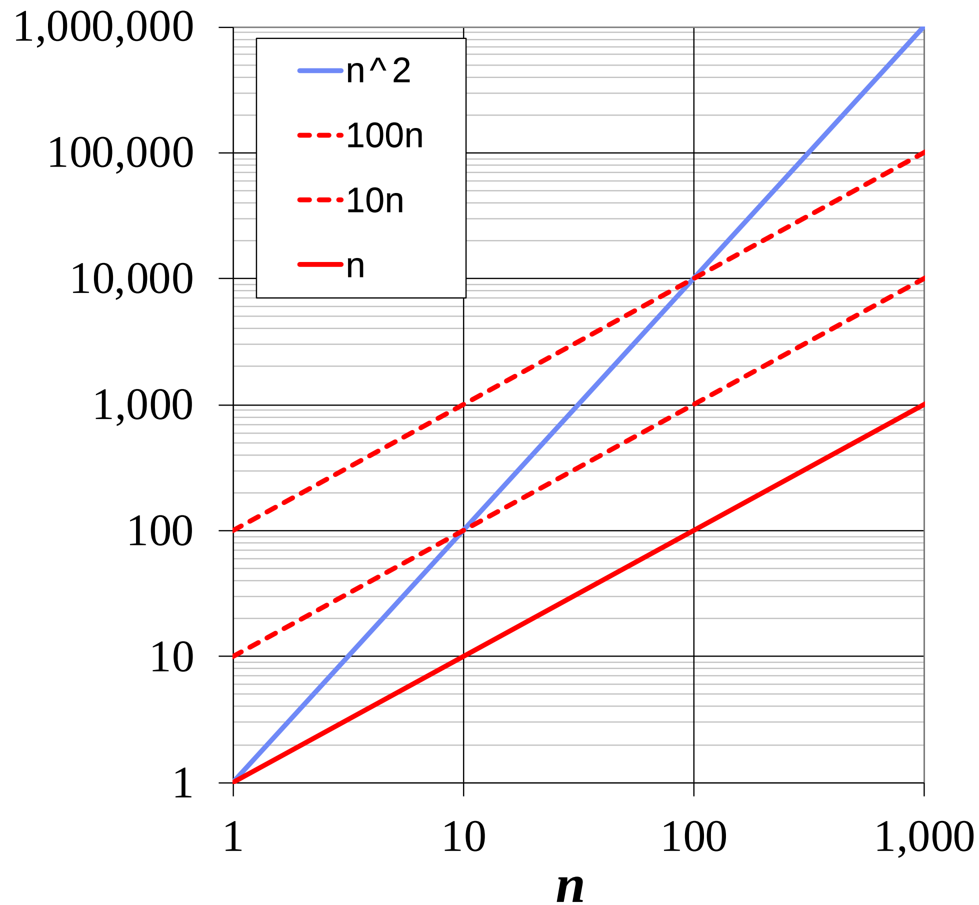
This definition is often referred to as the "big-Oh" notation, for it is sometimes pronounced as "$f(n)$ is ***big-Oh*** of $g(n)$."

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



BIG-OH EXAMPLE

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality c annot 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 $7n - 2 \leq cn$ 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 $3n^3 + 20n^2 + 5 \leq cn^3$ 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 $3 \log n + 5 \leq c \log n$ 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

Proposition 4.8: *If $f(n)$ is a polynomial of degree d, that is,*

$$f(n) = a_0 + a_1 n + \cdots + a_d n^d,$$

and $a_d > 0$, then $f(n)$ is $O(n^d)$.

- 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

COMPARATIVE ANALYSIS

- What's considered a better algorithm?
  - Asymptotically faster algorithm that solves the same proble m.
  - NOTE: Although we ignore constants, large constants can be effect the time significantly in real programs.
- What's considered fast?
  - Generally, algorithms that runs within O(nlogn) is considered fast.
  - If we need to divide algorithms to tractable and intractable, t he borderline will be polynomial (n^k) vs exponential (a^n)

EXAMPLE: TREE-WAY SET DISJOINTNESS

PROBLEM(three-way set disjointness): Given three sets, *A*, *B*, and *C*, that contains no duplicate values, determine if the intersection of the three sets is empty, namely, that there is no element *x* such that $x \in A, x \in B, \text{and} x \in C$.

```
/** Returns true if there is no element common to all three arrays. */
public static boolean disjoint1(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
  for (int a : groupA)
    for (int b : groupB)
      for (int c : groupC)
        if ((a == b) && (b == c))
          return false;                    // we found a common value
  return true;                             // if we reach this, sets are disjoint
}
```

$O(n^3)$

```
/** Returns true if there is no element common to all three arrays. */
public static boolean disjoint2(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
  for (int a : groupA)
    for (int b : groupB)
      if (a == b)                    // only check C when we find match from A and B
        for (int c : groupC)
          if (a == c)                // and thus b == c as well
            return false;            // we found a common value
  return true;                       // if we reach this, sets are disjoint
}
```

HINT: There are quadratically many pairs (a,b) to consider. However, if A and B are each sets of distinct elements, there can be at most $O(n)$ such pairs with a equal to b. Therefore, the inner most loop, over C, executes at most n times.

$O(n^2)$

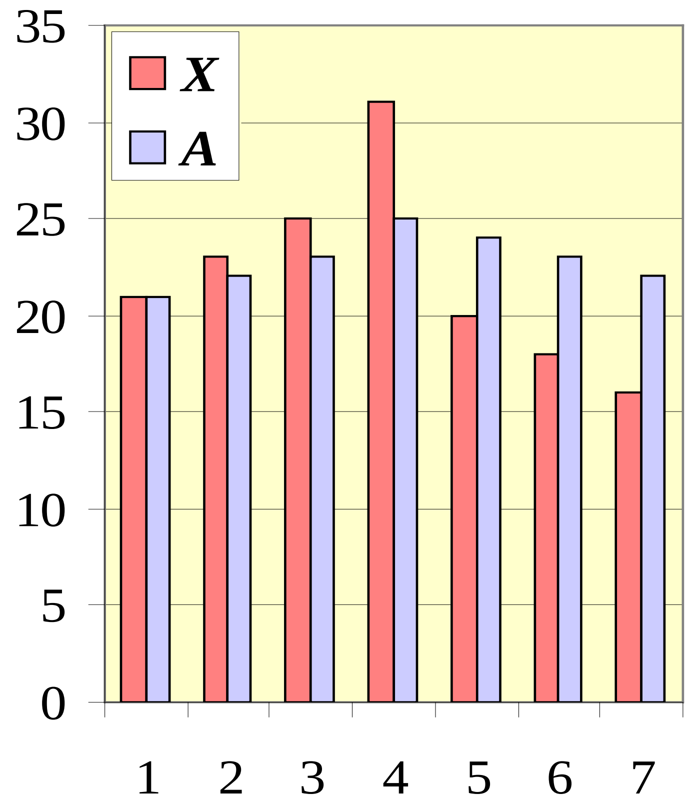
EXAMPLE: COMPUTING PREFIX AVERAGES

PROBLEM: The ***i***-th prefix aver age of an array ***X*** is average o f the first $(i+1)$ elements of ***X***:

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

Computing the array ***A*** of prefi x averages of another array ***X***

APPLICATIONS: Given a stream of daily Web usage logs, a websi te manager may wish to track av erage usage trends over various time periods.



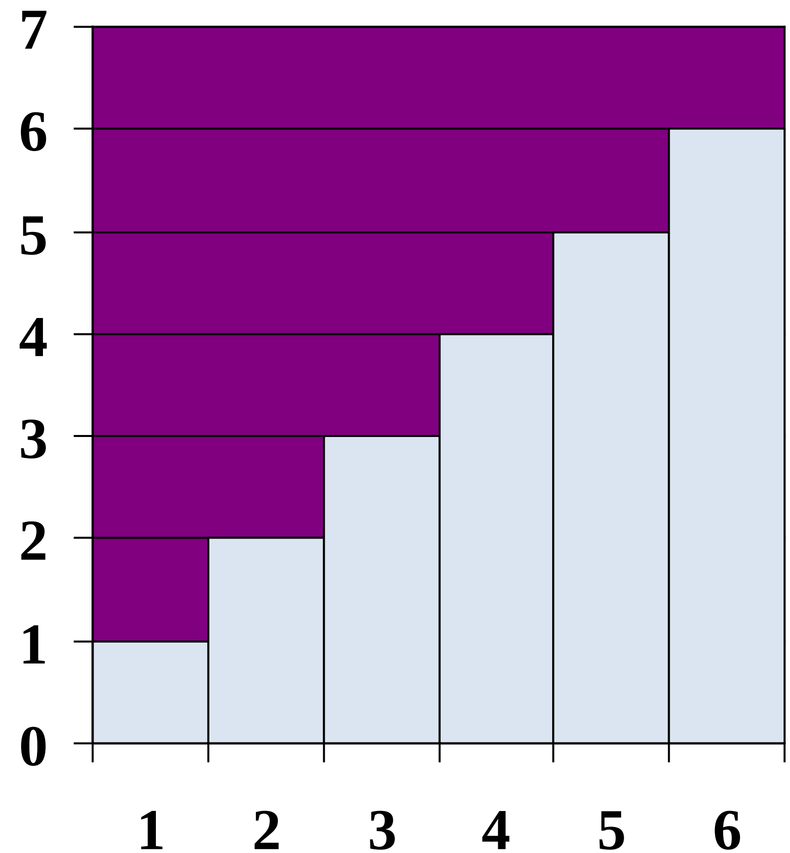
Prefix Averages (Quadratic time)

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 prefi xAverage1 is $O(1 + 2 + \ldots + n)$
- The sum of the first n in tegers is $n(n + 1) / 2$
  - There is a simple visual pr oof of this fact
- Thus, algorithm prefixAv erage1 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!

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

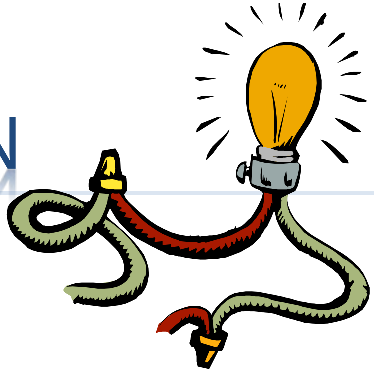
$$f(n) \geq c\, g(n) \text{ for } n \geq n_0$$

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)

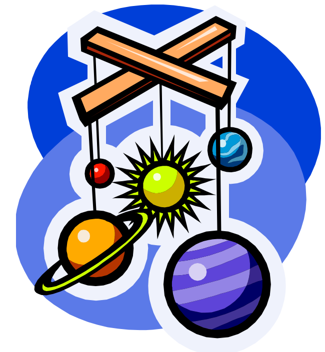
big-Omega

- 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 $n \geq n_0$

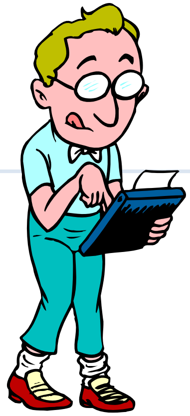
  let $c = 1$ and $n_0 = 1$

- **$5n^2$ 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 $n_0 = 1$

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:
  $\log_b(xy) = \log_b x + \log_b y$
  $\log_b (x/y) = \log_b x - \log_b y$
  $\log_b xa = a\log_b x$
  $\log_b a = \log_x a / \log_x b$