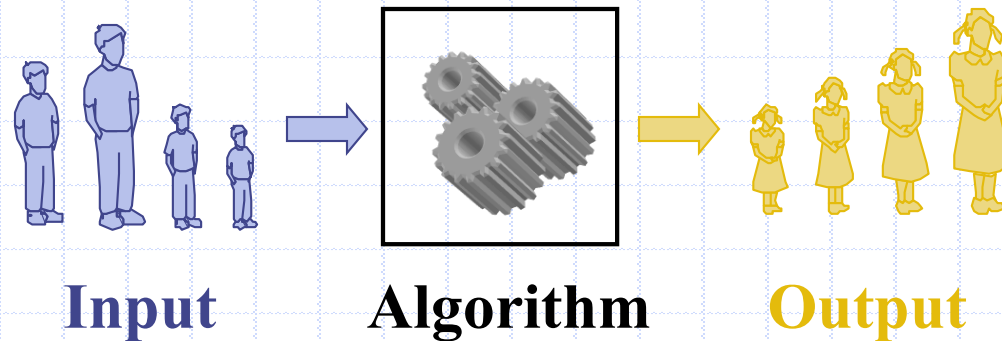


Lesson 2: Introduction to Analysis of Algorithms: *Discovering the Laws Governing Nature's Computation*



Wholeness of the Lesson

An algorithm is a procedure for performing a computation or deriving an output from a given set of inputs according to a specified rule. By representing algorithms in a neutral language, it is possible to determine, in mathematical terms, the efficiency of an algorithm and whether one algorithm typically performs better than another. Efficiency of computation is the hallmark of Nature's self-referral performance. Contact with the home of all the laws of nature at the source of thought results in action that is maximally efficient and less prone to error.

Natural Things to Ask

- ◆ How can we determine whether an algorithm is *efficient*?
- ◆ *Correct*?
- ◆ Given two algorithms that achieve the same goal, how can we decide which one is better? (E.g. sorting) Can our analysis be independent of a particular operating system or implementation in a language?
- ◆ How can we express the steps of an algorithm without depending on a particular implementation?

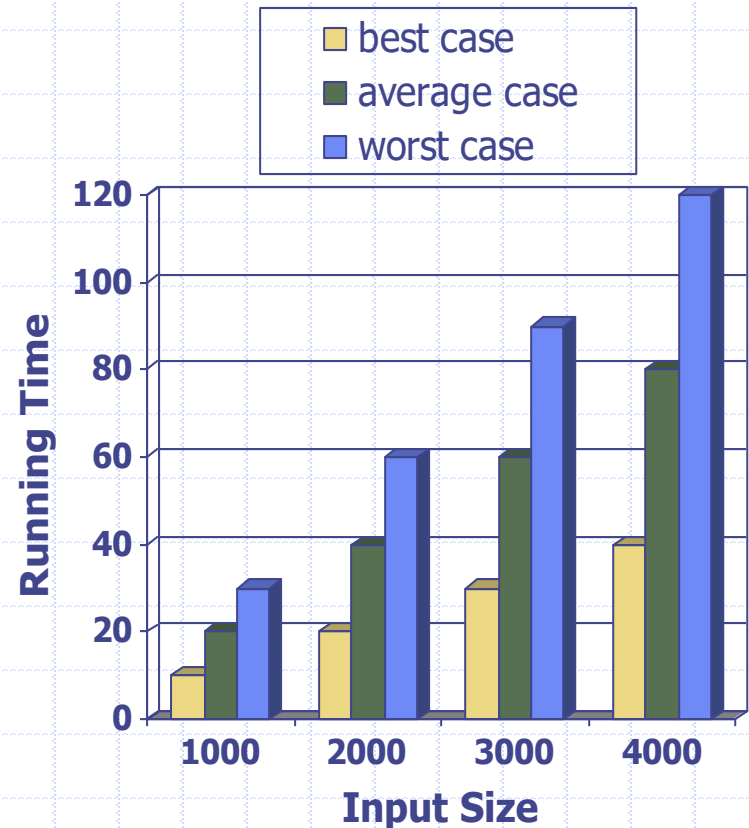
A Framework For Analysis of Algorithms

We will specify:

- ◆ A simple neutral language for describing algorithms
- ◆ A simple, general computational model in which algorithms execute
- ◆ Procedures for measuring running time
- ◆ A classification system that will allow us to categorize algorithms (a precise way of saying “fast”, “slow”, “medium”, etc)
- ◆ Techniques for proving correctness of an algorithm

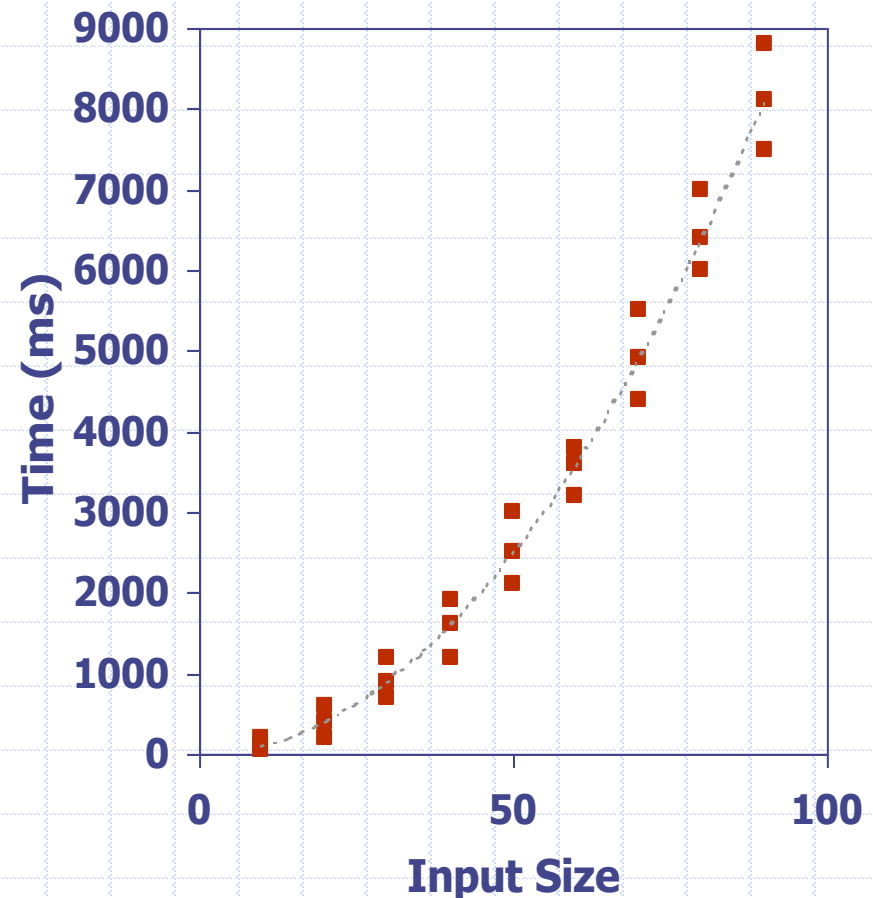
Running Time

- ◆ 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.
- ◆ Often, we focus on the worst case running time.
 - Easier to analyze
 - Crucial to applications such as games, finance and robotics



Running Time by Experiment

- ◆ 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
- ◆ Optionally, plot the results

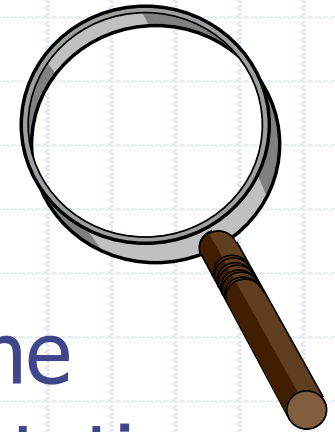


Limitations of Experiments

- ◆ 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

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: find max element of an array

```
Algorithm arrayMax(A, n)  
  Input array A of n integers  
  Output maximum element of A  
  
  currentMax  $\leftarrow A[0]$   
  for i  $\leftarrow 1$  to n - 1 do  
    if A[i] > currentMax then  
      currentMax  $\leftarrow A[i]$   
  return currentMax
```


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

var.method (*arg* [, *arg*...])

◆ Return value

return *expression*

◆ Expressions

← Assignment
(like = in Java)

= Equality testing
(like == in Java)

*n*² Superscripts and other
mathematical
formatting allowed

Exercises

◆ Sorting algorithm: Translate into pseudo-code:

Given an array arr, create a second array arr2 of the same length.

To begin, find the min of arr, place it in position 0 of arr2, and remove min from arr

At the ith step, find the min of arr, place it in the next available position in arr2, and remove min from arr

Return arr2

(Assume that min and remove functions are available)

◆ Palindrome algorithm: Transform into pseudo-code:

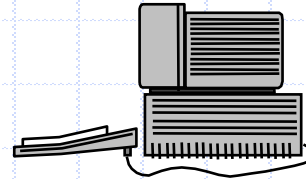
```
public boolean isPal(String s) {  
    if(s==null || s.length() <= 1) {  
        return true;  
    }  
    while(s.length() > 1){  
        if(s.charAt(0) !=  
            s.charAt(s.length()-1)){  
            return false;  
        }  
        s = s.substring(1,s.length()-1);  
    }  
    return true;  
}
```

Main Point

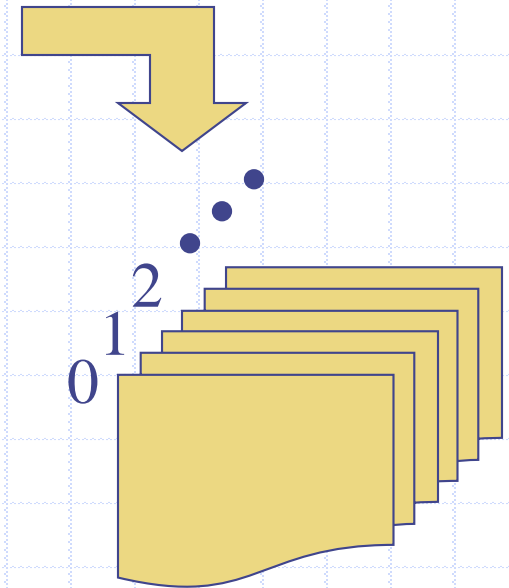
For purposes of examining, analyzing, and comparing algorithms, a neutral algorithm language is used, independent of the particularities of programming languages, operating systems, and system hardware. Doing so makes it possible to study the inherent performance attributes of algorithms, which are present regardless of implementation details. This illustrates the SCI principle that more abstract levels of intelligence are more comprehensive and unifying.

The Random Access Machine (RAM) Model

- ◆ A CPU



- ◆ A potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or characters



- ◆ Memory cells are numbered and accessing any cell in memory takes unit time.

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

Primitive Operations In This Course

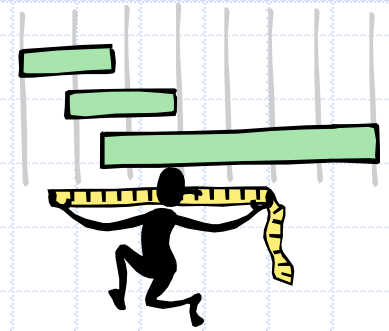
- Performing an arithmetic operation (+, *, etc)
- Comparing two numbers
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method
- Following an object reference

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

Algorithm <i>arrayMax</i> (<i>A</i> , <i>n</i>)	# operations
<i>currentMax</i> $\leftarrow A[0]$	2
<i>m</i> $\leftarrow n - 1$	2
for <i>i</i> $\leftarrow 1$ to <i>m</i> do	$1 + n$
if <i>A</i> [<i>i</i>] > <i>currentMax</i> then	$2(n - 1)$
<i>currentMax</i> $\leftarrow A[i]$	$2(n - 1)$
{ increment counter <i>i</i> }	$2(n - 1)$
return <i>currentMax</i>	1
Total	$7n$

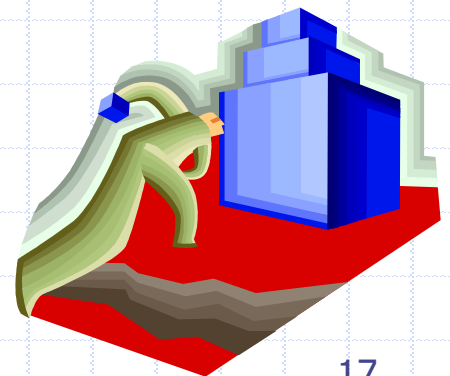
Estimating Running Time



- ◆ Algorithm *arrayMax* executes $7n$ 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
$$a * (7n) \leq T(n) \leq b * (7n)$$
- ◆ 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 $T(n)$
- ◆ The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm *arrayMax*



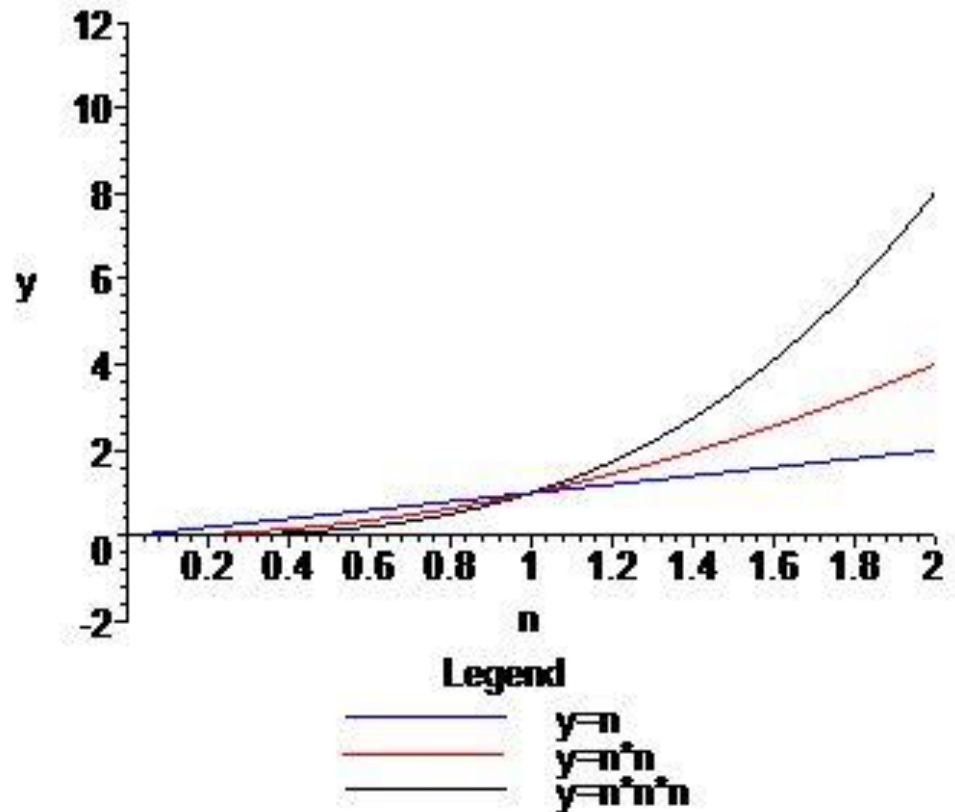
Growth Rates

◆ Growth rates of functions:

- Linear $\approx n$
- Quadratic $\approx n^2$
- Cubic $\approx n^3$

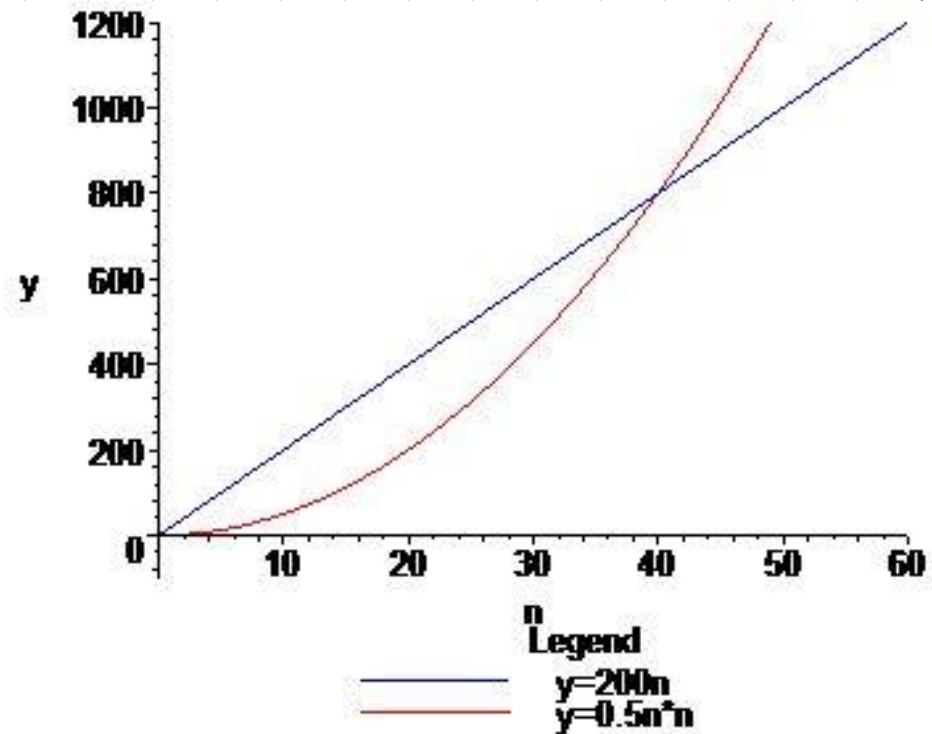
◆ The graph of the cubic begins as the slowest but eventually overtakes the quadratic and linear graphs

◆ Important factor for growth rates is the behavior as n gets large



Constant Factors & Lower-order Terms

- ◆ The growth rate is not affected by
 - constant factors
 - or
 - lower-order terms
- ◆ Example
 - Compare $200 \cdot n$ with $0.5n^2$
 - Quadratic growth rate must eventually dominate linear growth



Big-Oh Notation (§ 1.2)

◆ Given functions $f(n)$ and $g(n)$ defined on non-negative integers n , we say that $f(n)$ is $O(g(n))$ (or “ $f(n)$ belongs to $O(g(n))$ ”) if there are positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$

◆ Example: $2n + 10$ is $O(n)$

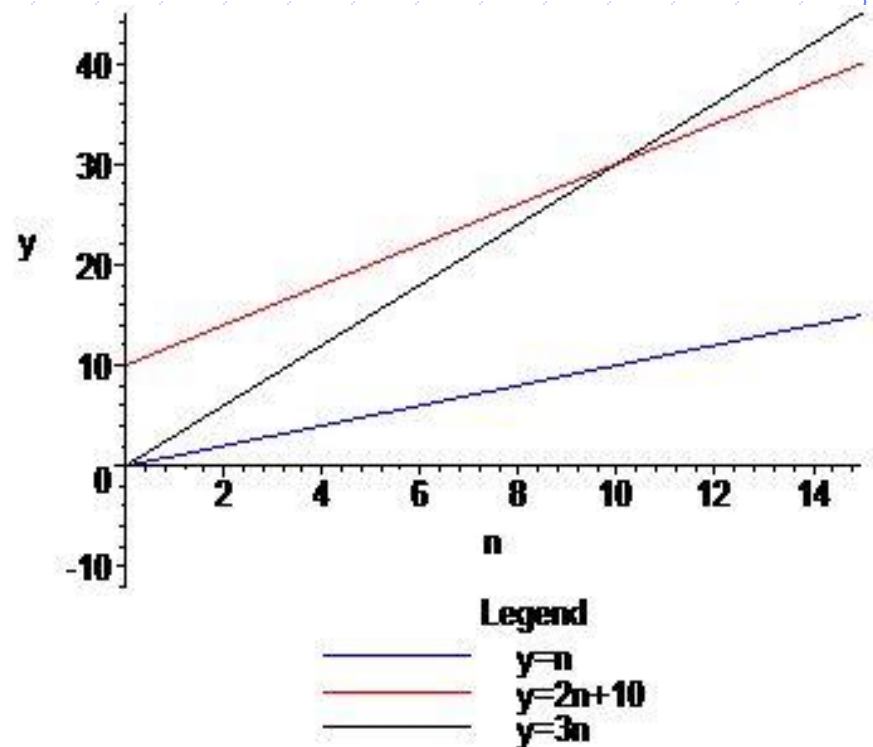
$$f(n) = 2n + 10$$

If $g(n) = n$, $3g(n)$ will eventually get bigger than $f(n)$. We look for n_0 , the point where the two graphs meet:

$$3n = 2n + 10$$

$$n = 10$$

It follows that for all $n \geq 10$, $f(n) \leq 3g(n)$



Big-Oh Example

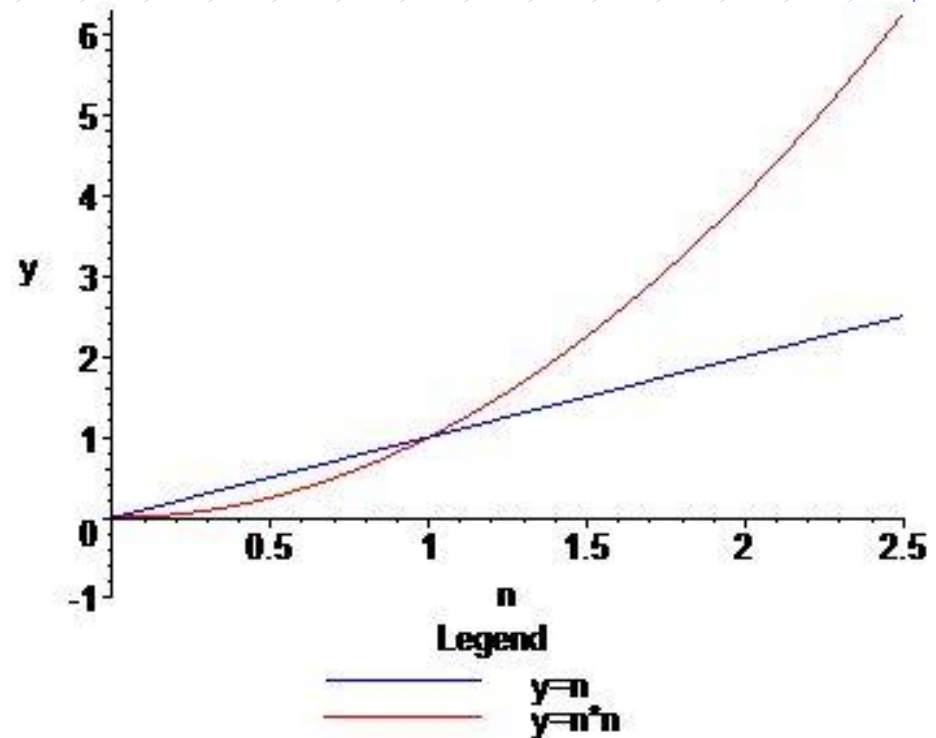
◆ Example: n^2 is not $O(n)$

Proof

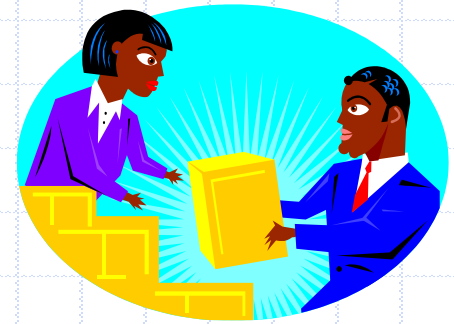
For each c and n_0 , we need to find an $n \geq n_0$ such that $n^2 > cn$.

Can do this by letting n be any integer bigger than both n_0 and c . Then

$$n^2 = n * n > c * n$$



More Big-Oh Examples



- ◆ n is $O(2n+1)$
- ◆ $n \log n + n$ is $O(n \log n)$
- ◆ Fact (for students who are familiar with “limits”):

If

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

exists and is finite then

f is $O(g)$.

- Example:

$$3n^2 + 1 \text{ is } O(2n^2 + n)$$

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 $g(n)$
- ◆ Example: Neither of the functions $2n$ nor $2n^2$ grows any faster (asymptotically) than n^2 . Therefore, both functions belong to $O(n^2)$

Standard Complexity Classes

- ◆ The most common complexity classes used in analysis of algorithms are, in increasing order of growth rate:

$O(1)$, $O(\log n)$, $O(n^{1/k})$, $O(n)$, $O(n \log n)$, $O(n^k)$ ($k > 1$),

$O(2^n)$, $O(n!)$, $O(n^n)$

- ◆ Functions that belong to classes in the first row are known as *polynomial time bounded*.
- ◆ Verification of the relationships between these classes can be done most easily using limits, sometimes with L' Hopital's Rule

L'Hopital's Rule. Suppose f and g have derivatives (at least when x is large) and their limits as $x \rightarrow \infty$ are either both 0 or both infinite. Then

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}$$

as long as these limits exist.

Optional Examples Using Limits

- ◆ $\lim_n (3n^2 + 2n - 4 / n^2)$ is finite ($= 3$), so $3n^2 + 2n - 4$ is $O(n^2)$.
- ◆ $\lim_n (n^{1/2} / n) = 0$ so $n^{1/2}$ is $O(n)$
but $\lim_n (n / n^{1/2})$ is infinite, so n is not $O(n^{1/2})$
- ◆ $\lim_n (\log n / n^{1/2}) = 0$, so $\log n$ is $O(n^{1/2})$
but $\lim_n (n^{1/2} / \log n)$ is infinite, so $n^{1/2}$ is not $O(\log n)$
- ◆ $\lim_n (n^k / 2^n) = 0$, so n^k is $O(2^n)$ (for any $k > 0$)
but $\lim_n (2^n / n^k)$ is infinite, so 2^n is not $O(n^k)$

Big-Oh Rules



◆ If $f(n)$ is a polynomial of degree d , say,
 $f(n) = a_0 + a_1 n + \dots + a_d n^d$, then $f(n)$ is $O(n^d)$:

1. Drop lower-order terms (those of degree less than d)
2. Drop constant factors (in this case, a_d)
3. See first example on previous slide

◆ Guidelines:

- Use the smallest possible class of functions
- E.g. Say “ $2n$ is $O(n)$ ” instead of “ $2n$ is $O(n^2)$ ”
- Use the simplest expression of the class
- E.g. 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 (worst-case) 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 determined that algorithm *arrayMax* executes at most $7n$ primitive operations
 - Since $7n$ is $O(n)$, we say that algorithm *arrayMax* “runs in $O(n)$ time”

Basic Rules For Computing Asymptotic Running Times

◆ Rule-1: For Loops

The running time of a for loop is at most the running time of the statements inside the loop (including tests) times the number of iterations (see *arrayMax*)

◆ Rule-2: Nested Loops

Analyze from inside out. The total running time of a statement inside a group of nested loops is the running time of the statement times the sizes of all the loops

```
for i ← 0 to n-1 do
  for j ← 0 to n-1 do
    k ← i + j
```

(Runs in $O(n^2)$)

(continued)

◆ Rule-3: Consecutive Statements

Running times of consecutive statements should be added in order to compute running time of the whole

```
for i ← 0 to n-1 do  
    a[i] ← 0  
for i ← 0 to n-1 do  
    for j ← 0 to i do  
        a[i] ← a[i] + i + j
```

(Running time is $O(n) + O(n^2)$. By an exercise, this is $O(n^2)$)

(continued)

◆ Rule-4: If/Else

For the fragment

if *condition* **then**

S1

else

S2

the running time is never more than the running time of the *condition* plus the larger of the running times of S1 and S2.

Example: Removing Duplicates From An Array

The problem: Given an array of n integers that lie in the range $0..2n - 1$, return an array in which all duplicates have been removed.

Remove Dups, Algorithm #1

Algorithm removeDups1(A,n)

Input: An array A with $n > 0$ integers in the range $0..2n-1$

Output: An array B with all duplicates in A removed

for $i \leftarrow A.length-1$ **to** 0 **do**

for $j \leftarrow A.length-1$ **to** $i+1$ **do**

if $A[j] = A[i]$ **then**

$A \leftarrow \text{removeLast}(A, A[i])$

Algorithm removeLast(A,a)

Input: An array A of integers and an array element a

Output: The array A modified by removing last occurrence of a

$pos \leftarrow -1$

$k \leftarrow A.length$

$B \leftarrow \text{new Array}(k-1)$

$i \leftarrow k-1$

while $pos < 0$ **do** //must eventually terminate

if $a = A[i]$ **then** $pos \leftarrow i$

else $i \leftarrow i-1$

for $j \leftarrow 0$ **to** $k-2$ **do**

if $j < pos$ **then** $B[j] \leftarrow A[j]$

else $B[j] \leftarrow A[j+1]$

return B

Analysis

T_{rl} = running time of removeLast

T_{rd} = running time of removeDups1

- $T_{rl}(k)$ is $O(2k) = O(k)$

- $T_{rd}(n)$ is $O(n^3)$

Therefore, the running time of Algorithm #1 is $O(n^3)$

One way to improve: Insert non-dups into an auxiliary array.

Remove Dups, Algorithm #2

Algorithm removeDups2(A,n)

Input: An array A with $n > 0$ integers in the range $0..2n-1$

Output: An array B with all duplicates in A removed

$B \leftarrow \text{new Array}(n)$ //assume initialized with 0's

$\text{index} \leftarrow 0$

for $i \leftarrow 0$ **to** $n-1$ **do**

$\text{dupFound} \leftarrow \text{false}$

for $j \leftarrow 0$ **to** $i-1$ **do**

if $A[j] = A[i]$ **then**

$\text{dupFound} \leftarrow \text{true}$

break //exit to outer loop

 //if no dup found up to i, add $A[i]$ to new array

if !dupFound **then**

$B[\text{index}] \leftarrow A[i]$

$\text{index} \leftarrow \text{index} + 1$

 //end outer for loop

 //next: eliminate extra 0's at the end

$C \leftarrow \text{new Array}(\text{index})$

for $j \leftarrow 0$ **to** index **do**

$C[j] \leftarrow B[j]$

return C

Analysis

T = running time of removeDups2

$O(n)$ initialization +
 nested for loops bound to n +
 $O(n)$ copy operation

\Rightarrow

$T(n)$ is $O(n) + O(n^2) + O(n)$
 $= O(n^2)$

Therefore, the running time of
Algorithm #2 is $O(n^2)$

One way to improve: Use bookkeeping
device to keep track of duplicates and
eliminate inner loop

Remove Dups, Algorithm #3

Algorithm removeDups3(A,n)

Input: An array A with $n > 0$ integers in the range $0..2n-1$

Output: An array with all duplicates in A removed

```
W ← new Array(2n)    //for bookkeeping
B ← new Array(n)      //assume both initialized with 0's
index ← 0
for i ← 0 to n-1 do
    u ← A[i]
    if W[u] = 0 then //means a new value
        B[index] ← A[i]
        index ← index + 1
        W[u] ← 1
```

//next: eliminate extra 0s at the end

```
C ← new Array(index)
```

```
for j ← 0 to index do
```

```
    C[j] ← B[j]
```

```
return C
```

Analysis

T = running time of removeDups3

$O(n)$ initialization +
single for loop of size n +
 $O(n)$ copy operation

=>

$T(n)$ is $3 * O(n) = O(n)$

Therefore, the running time of Algorithm #3 is $O(n)$

Main Point

One can improve the running time of the "remove duplicates" algorithm from $O(n^2)$ to $O(n)$ by introducing a tracking array as a means to encapsulate "bookkeeping" operations (this can be done more generally using a hashtable). This technique is reminiscent of the Principle of the Second Element from SCI: To remove the darkness, struggling at the level of darkness is ineffective; instead, introduce a *second element* – namely, *light*. As soon as the light is introduced, the problem of darkness disappears.

Relatives of Big-Oh



◆ **big-Omega**

- $f(n)$ is $\Omega(g(n))$ if $g(n)$ is $O(f(n))$.

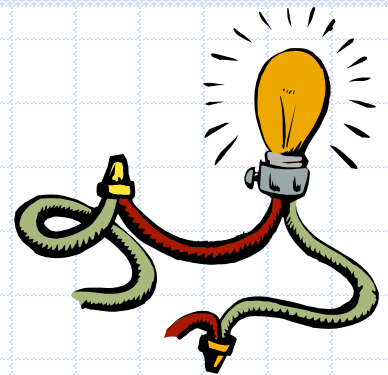
◆ **big-Theta**

- $f(n)$ is $\Theta(g(n))$ if $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$.

◆ **little-oh**

- $f(n)$ is $o(g(n))$ if, for any constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$
- In case $\lim_n(f(n)/g(n))$ exists, $f(n)$ is $o(g(n))$ iff the limit = 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 $\Omega(g(n))$ if $f(n)$ is **asymptotically greater than or equal** to $g(n)$

big-Theta

- $f(n)$ is $\Theta(g(n))$ if $f(n)$ is **asymptotically equal** to $g(n)$

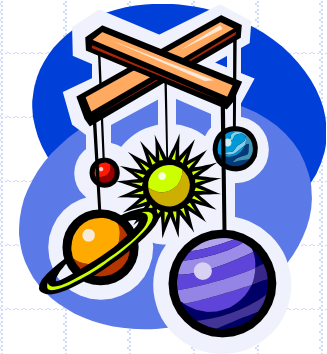
little-oh

- $f(n)$ is $o(g(n))$ if $f(n)$ is **asymptotically strictly less** than $g(n)$

little-omega

- $f(n)$ is $\omega(g(n))$ if $f(n)$ is **asymptotically strictly greater** than $g(n)$

Examples of the Relatives of Big-Oh



- **$5n^2$ is $\Omega(n^2)$ and therefore, $5n^2$ is $\Theta(n^2)$**

$f(n)$ is $\Omega(g(n))$ iff $g(n)$ is $O(f(n))$ iff there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$

So $5n^2$ is $\Omega(n^2)$ iff n^2 is $O(5n^2)$, which is obviously true.

To show $5n^2$ is $\Theta(n^2)$, must show also that $5n^2$ is also $O(n^2)$ – this is also obvious.

Therefore $5n^2$ is $\Theta(n^2)$

- **$5n$ is $o(n^2)$ but $5n$ is not $o(n)$**

Need to show that for any positive c , $5n \leq cn^2$ for large enough n .

This inequality holds whenever $n \geq 5/c$.

Therefore, to prove that $5n$ is $o(n^2)$, given any positive c , pick n_0 bigger than $5/c$.

Then for all $n \geq n_0$, $5n \leq cn^2$

To show $5n$ is not $o(n)$, we must find positive c so that for every choice of n_0 , there is an $n \geq n_0$ for which $5n > cn$.

This is obviously true: let $c = 1$, and given n_0 , choose $n = n_0$.

Running Time of Recursive Algorithms

- ◆ Problem: Given an array of integers in sorted order, is it possible to perform a search for an element in such a way that no more than half the elements of the array are examined? (Assume the array has 8 or more elements.)

Binary Search

Algorithm search(A,x)

Input: An already sorted array A with n elements and search value x

Output: true or false

return binSearch(A, x, 0, A.length-1)

(continued)

Algorithm binSearch(A, x, lower, upper)

Input: Already sorted array A of size n, value x to be searched for in array section A[lower]..A[upper]

Output: true or false

```
if lower > upper then return false
mid ← (upper + lower)/2
if x = A[mid] then return true
if x < A[mid] then
    return binSearch(A, x, lower, mid - 1)
else
    return binSearch(A, x, mid + 1, upper)
```

For the worst case (x is above all elements of A and n a power of 2), running time is given by the **Recurrence Relation:** (In this case, right half is always half the size of the original.)

$$T(1) = d; \quad T(n) = c + T(n/2)$$

Solving A Recurrence Relation: The Guessing Method

$$T(1) = d$$

$$T(2) = c + d$$

$$T(4) = c + c + d$$

$$T(8) = c + c + c + d$$

$$T(2^m) = c * m + d$$

$$T(n) = c * \log n + d, \text{ which is } O(\log n)$$

- ◆ Guessing method requires us to guess the general formula from a few small values. This is not always possible to do (we will discuss an alternative method in a later lecture)
- ◆ When using the guessing method, final formula should be verified. Often this is done by induction, though in simple cases, a direct verification is possible.

Verification of Formula

Claim: $T(n) = c * \log n + d$ for all n of the form $n = 2^m$.

In other words, $c * \log n + d$ satisfies the recurrence relation:

$$T(1) = d; \quad T(n) = T(n/2) + c \text{ for all } n \text{ of the form } n = 2^m.$$

Proof: We prove the result for all non-negative integers of the form $n = 2^m$. The proof is by induction on m .

Base case:

Let $m = 0$. Then $n = 1$. $LHS = T(1) = d$. $RHS = c * \log 1 + d = d$. Thus $LHS = RHS$.

Induction hypothesis:

The result is true for $m = k$. That is, $n = 2^k$ and $T(2^k) = c * \log (2^k) + d$.

$$\text{Thus } T(2^k) = c * k + d \quad \text{eq. (1)}$$

Inductive step:

We need to prove $T(2^{k+1}) = c * (k + 1) + d$.

$$\begin{aligned} LHS = T(2^{k+1}) &= T(2^k) + c && \text{(by } T(n) = T(n/2) + c \text{)} \\ &= (c * k + d) + c && \text{(by eq. (1))} \\ &= c * (k + 1) + d = RHS. \end{aligned}$$

Note: base of the log is 2 in our algorithm class unless otherwise stated.

Additional Points About the Guessing Method

- ◆ To state results (so far) correctly, it's necessary for n to be a power of 2. But we wish to have a bound on running time for any n – what can be done?
- ◆ In all cases that concern us, T is well enough behaved (as is the complexity function $\log n$) between successive powers of 2 to permit us to conclude that, if we know the asymptotic running time, assuming n is a power of 2, the running time for all n , not necessarily a power of 2, must belong to this same complexity class.

The Divide and Conquer Algorithm Strategy

- ◆ The binary search algorithm is an example of a “Divide And Conquer” algorithm, which is typical strategy when recursion is used.
- ◆ The method:
 - **Divide** the problem into subproblems (divide input array into left and right halves)
 - **Conquer** the subproblems by solving them recursively (search recursively in whichever half could potentially contain target element)
 - **Combine** the solutions to the subproblems into a solution to the problem (return value found or indicate not found)

Main Point

Recurrence relations are used to analyze recursively defined algorithms. Just as recursion involves repeated self-calls by an algorithm, so the complexity function $T(n)$ is defined in terms of itself in a recurrence relation. Recursion is a reflection of the self-referral dynamics at the basis of Nature's functioning. Ultimately, there is just one field of existence; everything that happens therefore is just the dynamics of this one field interacting with itself. When individual awareness opens to this level of Nature's functioning, great accomplishments become possible.

Another Technique To Solve Recurrences: Counting Self-Calls

- ◆ To determine the running time of a recursive algorithm, another often-used technique is *counting self-calls*.
- ◆ Often, processing time in a recursion, apart from self-calls, is constant. In such cases, running time is proportional to the number of self-calls.

Example of Counting Self-Calls: The Fib Algorithm

- ◆ The Fibonacci numbers are defined recursively by:
 $F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)$
- ◆ This is a recursive algorithm for computing the n th Fibonacci number:

Algorithm fib(n)

Input: a natural number n

Output: $F(n)$

if ($n = 0 \parallel n = 1$) **then return** n

return fib($n-1$) + fib($n-2$)

(continued)

Lemma. For $n > 1$, the number $S(n)$ of self-calls in $\text{fib}(n)$ is $\geq F(n)$

Proof. By (strong) induction on n .

Base Cases:

$n=2$. In this case $S(2) = 2 \geq 1 = F(2)$. Thus $S(n) \geq F(n)$.

$n=3$. In this case $S(3) = 4 \geq 2 = F(3)$. Thus $S(n) \geq F(n)$.

Induction Hypothesis:

Assume the result for all values of n less than or equal to k .

Therefore, $S(k) \geq F(k)$ and $S(k - 1) \geq F(k - 1)$

$$\begin{aligned} S(k+1) &= 2 + S(k) + S(k - 1) \\ &\geq 2 + F(k) + F(k - 1) \\ &\geq F(k + 1) \end{aligned}$$

Lemma. For all $n > 4$, $F(n) > (4/3)^n$ **Proof.** Exercise!

=====

Therefore, the running time of the fib algorithm is $\Omega(r^n)$ for some $r > 1$. In other words, fib is an *exponentially slow* algorithm!

Binary Search, Counting Self-Calls

- ◆ We can compute asymptotic running time of Binary Search using the technique of counting self-calls.
- ◆ Observation: Suppose n is a power of 2 – say $n = 2^m$. Then the number of terms in the sequence $2^m, 2^{m-1}, 2^{m-2}, \dots, 2^1, 2^0=1$ is $m + 1 = 1 + \log n$.
- ◆ In the worst case for BinarySearch, input size $n = 2^m$ is a power of 2 and is cut exactly in half with each successive self-call. Therefore, the successive input sizes of self-calls is $2^{m-1}, 2^{m-2}, \dots, 2^1, 2^0=1$, so $\log n$ self-calls are made. Running time is therefore $O(\log n)$

Descending Sequences

- ◆ We observed that if n is a power of 2 – say $n = 2^m$. Then the number of terms in the sequence

$2^m, 2^{m-1}, 2^{m-2}, \dots, 2^1, 2^0=1$
is $m + 1 = 1 + \log n$.

- ◆ A more general fact that can be proved using this observation is if n is any positive integer, the sequence of terms

$$n, n/2, n/4, \dots, n/2^m = 1$$

where m is the largest power of 2 that is $\leq n$, has exactly $1 + m = 1 + \lfloor \log n \rfloor$ terms

Proving Correctness: Iterative Algorithms

- ◆ To prove an algorithm is correct, we view it as a machine that transforms input data to output data. The input data are expected to satisfy certain requirements (called *preconditions*) and the output data are also expected to satisfy certain requirements (called *postconditions*).
- ◆ A proof of correctness is a proof that, whenever input data for an algorithm satisfy the preconditions, the output data satisfy the postconditions.

- ◆ Example

Algorithm add(m,n)

Input: Two integers m, n

Output: $m + n$

$z \leftarrow m + n$

return z

- ◆ Here, the precondition is: m and n are integers. The postcondition is: returned value is the sum of m and n.
- ◆ *Proof of Correctness:* Assume m and n are integers, so that the precondition is satisfied. As the algorithm executes, z is populated with the value $m + n$. The return value is z, which is the sum $m+n$, so the postcondition is satisfied.

(continued)

- ◆ In a careful demonstration of correctness, the algorithm is broken down into sections. Postconditions for one section become preconditions of next section. The proof of correctness will then show that within each section, under the assumption that preconditions hold for the section, the algorithm's output for that section satisfies the postconditions.
- ◆ In practice, for iterative algorithms, all that is necessary is to establish that the "goal" of each loop in the algorithm is achieved. This is done with the technique of *loop invariants*.

Loop Invariants

Algorithm iterativeFactorial(n)

Input: A non-negative integer n

Output: n!

```
if (n = 0 || n = 1) then
    return 1
accum ← 1
for i ← 1 to n do
    accum ← accum * i
return accum
```

Example: Loop invariant here is:
 $I(i)$: accum = i!

Definition of Loop Invariant: A loop invariant $I(i)$ is a statement depending on the iterator i , which holds true at the completion of the i th pass through the loop.

Theorem: A loop will be correct relative to its pre- and post-conditions if each of the following is true about its loop invariant $I(i)$

1. $I(k)$ is true at end of the $i=k$ pass for each k in the range of i . (This is verified by induction.)
2. If the loop terminates after finitely many iterations, the truth of the loop invariant ensures the truth of the postcondition of the loop

Proof of Correctness

Algorithm iterativeFactorial(n)

Input. A non-negative integer n

Output. n!

```
if (n = 0 || n = 1) then
    return 1
```

```
accum ← 1
```

```
for i ← 1 to n do
```

```
    accum ← accum * i
```

```
return accum
```

Proof of correctness:

We identify the loop invariant for the for loop as before: **I(i): accum = i!**

We verify by induction on i that I(k) holds at the end of the i = k pass, for $1 \leq k \leq n$.

Base Case. After the iteration i = 1 completes, accum has value $1 = 1!$.

Induction Step. Assuming I(k) holds, show I(k+1) holds. Because I(k) is true, at the end of the i = k pass, accum = k!. Then, at the end of the i = k+1 pass, we have:

$$\text{accum} = \text{accum} * i = \text{accum} * (k+1) = k! * k+1 = (k+1)!$$

This shows that the loop invariant is true for all k for which $1 \leq k \leq n$. In particular, the value stored in accum after the k = n pass is n!, and this is the value that the algorithm returns. Therefore, iterativeFactorial correctly computes n! on input n.

Proving Correctness: Recursive Algorithms

The strategy for proving correctness of a recursive algorithm involves the following steps:

1. Verify that the recursion is valid: there should be a base case and recursive calls must eventually lead to the base case
2. Show that the values given by the base case are correct outputs for the function
3. Show that, if you assume the output value of the algorithm on input j is correct, for all $j < n$, then output value on input n is correct.

Example of Correctness Proof for a Recursive Algorithm

Algorithm recursiveFactorial(n)

Input. A non-negative integer n

Output. n!

```
if (n = 0 || n = 1) then
    return 1
```

```
return n * recursiveFactorial(n-1)
```

Proof of Correctness

Valid Recursion Base case is “n= 0 or n=1”. Each self-call reduces input size by 1, so eventually base case is reached

Base The values 0! and 1! are correctly computed by the base case

Recursion Assuming recursiveFactorial(n-1) correctly computes n-1!, we must show output of recursiveFactorial(n) is correct. But output of recursiveFactorial(n) is $n * \text{recursiveFactorial}(n-1) = n!$, as required.

Therefore, the algorithm correctly computes n! for every n.

Another Example: McCarthy's 91

Algorithm mc91(n)

Input. A positive integer n

Output. 91 if $n \leq 101$,
n-10 otherwise

```
if n > 100 then
    return n - 10
return mc91(mc91(n+11))
```

Here, proof that this is a valid recursion is not obvious. It is necessary to prove the output is right in order to prove that the recursion is valid – not easy to separate these two validation points.

Optional Exercise: Prove mc91 is correct.

The Master Formula

For recurrences that arise from Divide-And-Conquer algorithms (like Binary Search), there is a general formula that can be used.

Theorem. Suppose $T(n)$ satisfies

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ aT(\lceil \frac{n}{b} \rceil) + cn^k & \text{otherwise} \end{cases}$$

where k is a nonnegative integer and a, b, c, d are constants with $a > 0, b > 1, c > 0, d \geq 0$. Then

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } a < b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^{\log_b a}) & \text{if } a > b^k \end{cases}$$

Master Formula (continued)

Notes.

- (1) The result holds if $\lceil \frac{n}{b} \rceil$ is replaced by $\lfloor \frac{n}{b} \rfloor$.
- (2) Whenever T satisfies this “divide-and-conquer” recurrence, it can be shown that the conclusion of the theorem holds for *all* natural number inputs, not just to powers of b .

Master Formula (continued)

Example. A particular divide and conquer algorithm has running time T that satisfies:

$$T(1) = d \quad (d > 0)$$

$$T(n) = 2T(n/3) + 2n$$

Find the asymptotic running time for T .

Master Formula (continued)

Solution. The recurrence has the required form for the Master Formula to be applied. Here,

$$a = 2$$

$$b = 3$$

$$c = 2$$

$$k = 1$$

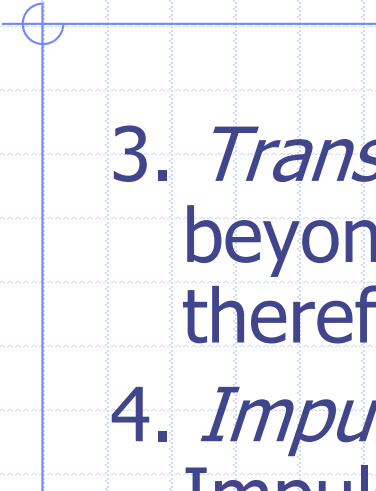
$$b^k = 3$$

Therefore, since $a < b^k$, we conclude by the Master Formula that

$$T(n) = \Theta(n).$$

Connecting the Parts of Knowledge With The Wholeness of Knowledge

1. There are many techniques for analyzing the running time of a recursive algorithm. Most require special handling for special requirements (e.g. n not a power of 2, counting self-calls, verifying a guess).
2. The Master Formula combines all the intelligence required to handle analysis of a wide variety of recursive algorithms into a single simple formula, which, with minimal computation, produces the exact complexity class for algorithm at hand.

- 
3. *Transcendental Consciousness* is the field beyond diversity, beyond problems, and therefore is the field of solutions.
 4. *Impulses Within The Transcendental Field.* Impulses within this field naturally form the blueprint for unfoldment of the highly complex universe. This blueprint is called *Ved.*
 5. *Wholeness Moving Within Itself.* In Unity Consciousness, solutions to problems arise naturally as expressions of one's own unbounded nature.