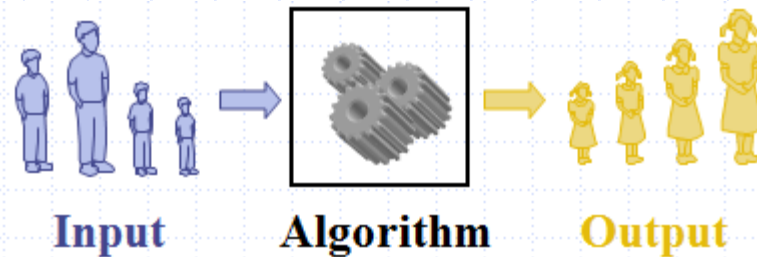# Lesson 2: Introduction to Analysis of Algorithms: *Discovering the Laws Governing Nature's Computation*

Input **Algorithm** **Output**

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

Overview of the Lesson

1. Issues in Determining Running Time
2. Pseudo-code to Describe Algorithms
3. RAM Model and Counting Primitive Operations
4. Growth Rates and Limits at Infinity
5. Theta (Θ), little-oh (o), little-omega (ω)
6. Big-oh (O) and big-omega (Ω)
7. Asymptotic Algorithm Analysis
8. The Sum of Two Problem
9. Running Time of Recursive Algorithms: Guessing, Counting Self-Calls and The Master Formula
10. Proving Correctness: Iterative Algorithms
11. Proving Correctness: Recursive Algorithms

Issues in Determining Running Time

- 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
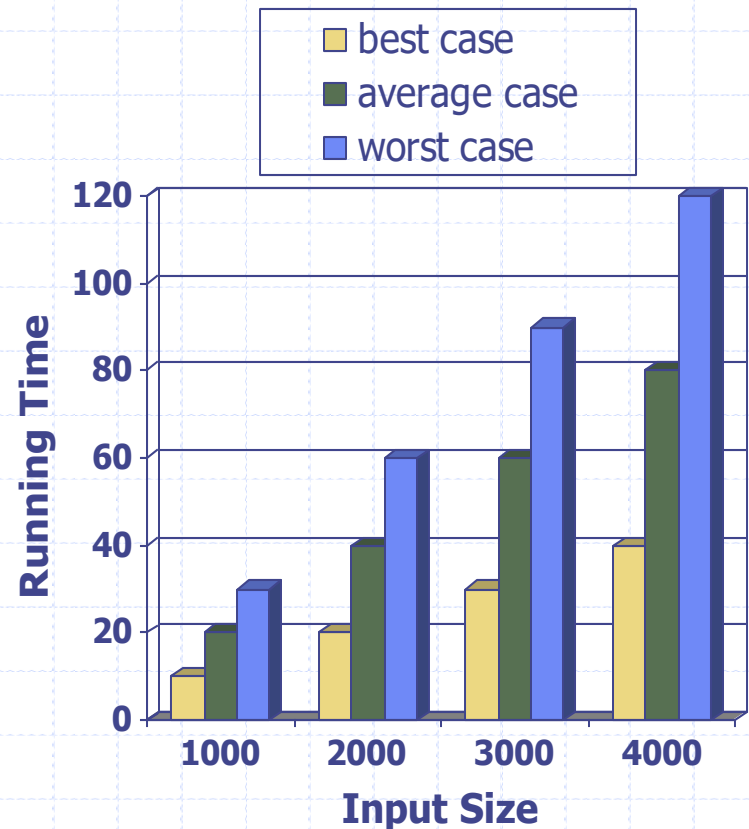
Direct Measurement

- Write a program implementing the algorithm
- Run the program with inputs of varying size
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
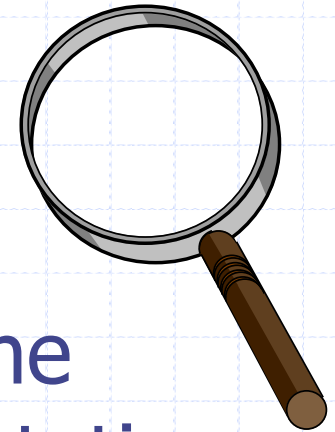- Optionally, plot the results
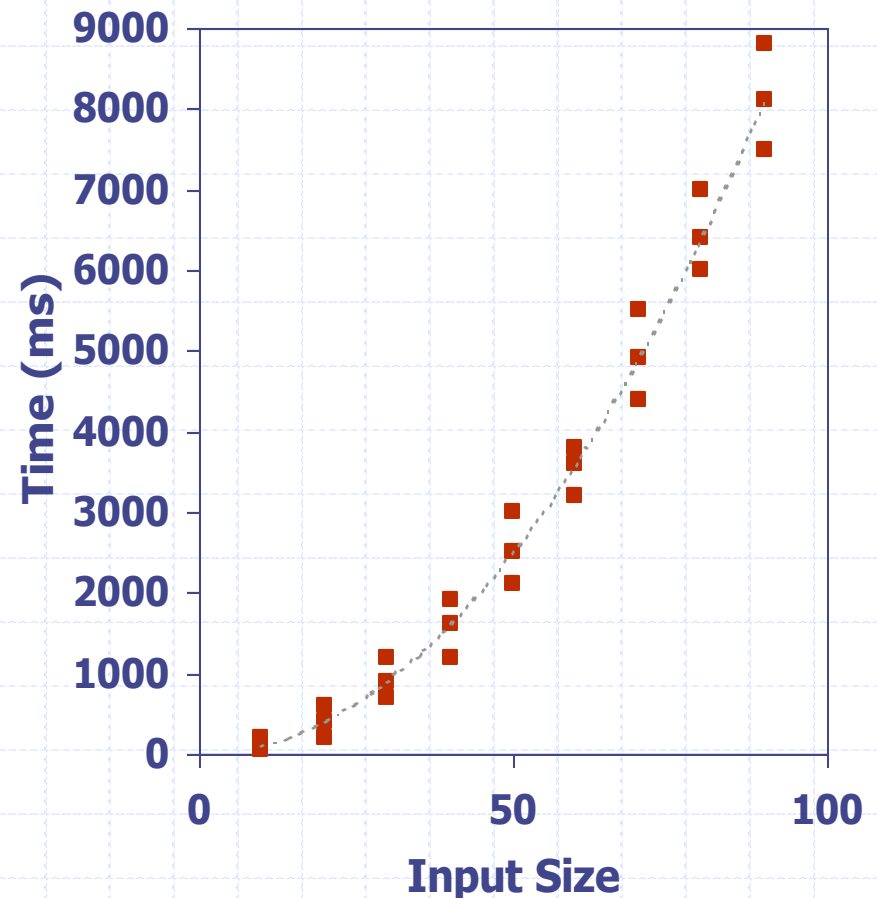
Limitations of Direct Measurement

- 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

Overview of the Lesson

1. Issues in Determining Running Time
2. Pseudo-code to Describe Algorithms
3. RAM Model and Counting Primitive Operations
4. Growth Rates and Limits at Infinity
5. Theta (Θ), little-oh (o), little-omega (ω)
6. Big-oh (O) and big-omega (Ω)
7. Asymptotic Algorithm Analysis
8. The Sum of Two Problem
9. Running Time of Recursive Algorithms: Guessing, Counting Self-Calls and The Master Formula
10. Proving Correctness: Iterative Algorithms
11. Proving Correctness: Recursive Algorithms

Pseudo-code to Describe Algorithms

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms

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 Syntax

- ◆ 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

Exercise

Remove duplicates algorithm: Translate into pseudo-code:

Given a list L, return a list M of the distinct elements of L by doing the following:
For each i less than size of L, if L[i] is not yet in M, then add L[i] to M.

Exercise

Remove duplicates algorithm: Translate into pseudo-code:

Given a list L, return a list M of the distinct elements of L by doing the following:
For each i less than size of L, if L[i] is not yet in M, then add L[i] to M.

Algorithm *removeDups* **(L)**
   Input a list L
   Output a list M containing the distinct elements of L

   M ← new List
   for *i ← 0* **to** *L.size() -1* **do**
     if not *M.contains* (*L[i]*) **then**
   *M.add* (*L[i]*)
   return *M*

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.

Overview of the Lesson

1. Issues in Determining Running Time
2. Pseudo-code to Describe Algorithms
3. RAM Model and Counting Primitive Operations
4. Growth Rates and Limits at Infinity
5. Theta (Θ), little-oh (o), little-omega (ω)
6. Big-oh (O) and big-omega (Ω)
7. Asymptotic Algorithm Analysis
8. The Sum of Two Problem
9. Running Time of Recursive Algorithms: Guessing, Counting Self-Calls and The Master Formula
10. Proving Correctness: Iterative Algorithms
11. Proving Correctness: Recursive Algorithms

The Random Access Machine (RAM) Model

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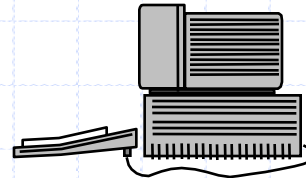
- A potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or characters
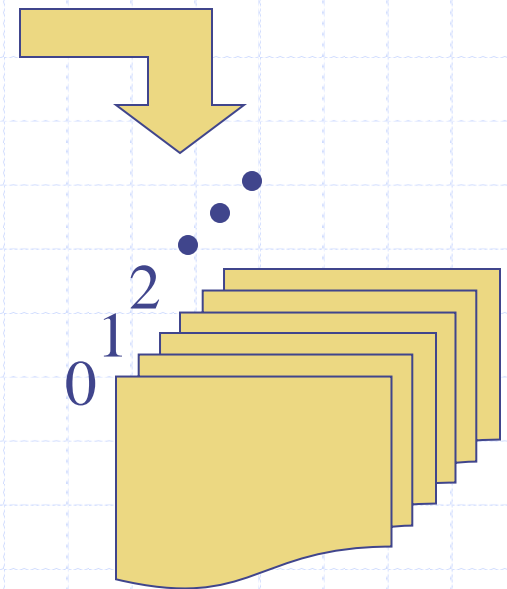
- Memory cells are numbered and accessing any cell in memory takes unit time.

Primitive Operations

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

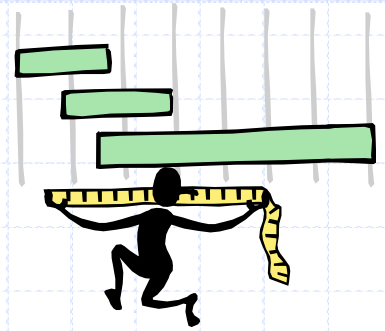# Primitive Operations in This Course

- 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 *arrayMax*(*A*, *n*)	# operations
currentMax ← *A*[0]	2
m ← *n* − 1	2
for *i* ← 1 **to** *m* **do**	$1 + n$
if *A*[*i*] > *currentMax* **then**	$2(n-1)$
currentMax ← *A*[*i*]	$2(n-1)$
{ increment counter *i* }	$2(n-1)$
return *currentMax*	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
$$a\,(7n) \le T(n) \le b(7n)$$
- 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*
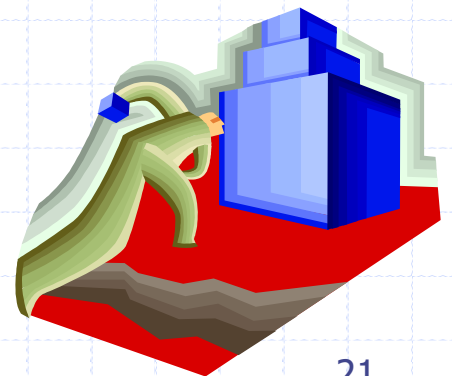
Overview of the Lesson

1. Issues in Determining Running Time
2. Pseudo-code to Describe Algorithms
3. RAM Model and Counting Primitive Operations
4. Growth Rates and Limits at Infinity
5. Theta (Θ), little-oh (o), little-omega (ω)
6. Big-oh (O) and big-omega (Ω)
7. Asymptotic Algorithm Analysis
8. The Sum of Two Problem
9. Running Time of Recursive Algorithms: Guessing, Counting Self-Calls and The Master Formula
10. Proving Correctness: Iterative Algorithms
11. Proving Correctness: Recursive Algorithms

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

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

Legend
y=n
y=n*n
y=n*n*n
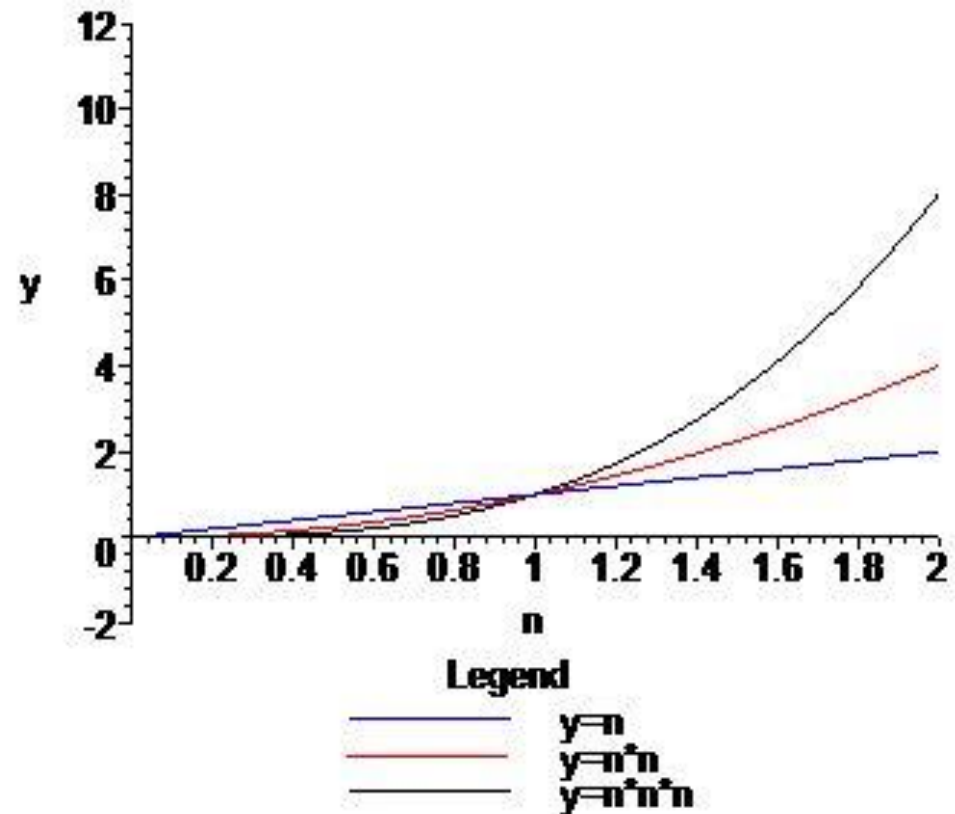
Detecting Growth Rates Using Limits

We can tell linear functions $f(n) = an + b$ always grow more slowly than the quadratic $g(n) = n^2$ because the quotient $f(n)/g(n)$ tends to 0 as n becomes large:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{an + b}{n^2} = \lim_{n \to \infty} \frac{\frac{a}{n} + \frac{b}{n^2}}{1} = 0.$$

Example: Show that 5n + 3 grows more slowly than n²

Solution: $\displaystyle \lim_{n \to \infty} \frac{5n + 3}{n^2} = \lim_{n \to \infty} \frac{\frac{5}{n} + \frac{3}{n^2}}{1} = 0$

(continued)

On the other hand, all quadratic functions always grow at the same rate. We can see this using limits: If $f(n) = an^2 + bn + c$ and $g(n) = dn^2 + en + r$, where $a \neq 0$ and $d \neq 0$, then the quotient $f(n)/g(n)$ tends to a nonzero number:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{an^2 + bn + c}{dn^2 + en + r} = \lim_{n \to \infty} \frac{a + \frac{b}{n} + \frac{c}{n^2}}{d + \frac{e}{n} + \frac{r}{n^2}} = \frac{a}{d} \neq 0.$$

Example: Show that $3n^2 + 7$ grows at the same rate as $5n^2 - n$.

Solution:
$$\lim_{n \to \infty} \frac{3n^2 + 7}{5n^2 - n} = \lim_{n \to \infty} \frac{3 + \frac{7}{n^2}}{5 - \frac{1}{n}} = \frac{3}{5} \neq 0$$

Overview of the Lesson

1. Issues in Determining Running Time
2. Pseudo-code to Describe Algorithms
3. RAM Model and Counting Primitive Operations
4. Growth Rates and Limits at Infinity
5. Theta (Θ), little-oh (o), little-omega (ω)
6. Big-oh (O) and big-omega (Ω)
7. Asymptotic Algorithm Analysis
8. The Sum of Two Problem
9. Running Time of Recursive Algorithms: Guessing, Counting Self-Calls and The Master Formula
10. Proving Correctness: Iterative Algorithms
11. Proving Correctness: Recursive Algorithms

Theta, Little-oh, Little-omega

Suppose f(n) and g(n) are functions. Then

- f(n) is o(g(n)) ("f(n) is little-oh of g(n)") if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$ [f(n) grows much more slowly than g(n)]

- f(n) is $\omega(g(n))$ ("f(n) is little-omega of g(n)") if g(n) is o(f(n)) [f(n) grows much faster than g(n)]

- f(n) is $\Theta(g(n))$ ("f(n) is theta of g(n)") if for some <u>nonzero</u> number **r** $\lim_{n \to \infty} \frac{f(n)}{g(n)} = r$ [f(n) grows at the same rate as g(n)]

The classes of functions represented by o, ω, and Θ are called <u>complexity classes</u>.

<u>Note</u>: It is theoretically possible that limits of this kind may not exist. This situation almost never arises in the context of determining running times of algorithms, and so we do not attempt to handle this special case in this course.

Examples

- 5n + 3 is $o(n^2)$

- $3n^2 + 7$ is $\Theta(n^2)$

- n^2 is $\omega(n)$.

Standard Complexity Classes

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

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

$$\Theta(2^n), \Theta(n!), \Theta(n^n)$$

Functions that belong to classes in the first row are known as *polynomial time bounded.*

◆ Verification of the relationships between these classes sometimes requires the use of *L'Hopital's Rule*

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

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

as long as these limits exist.

Example using L'Hopital

Problem. Show that $\log n$ is $\mathrm{o}(\sqrt{n})$.

Solution. We show that the limit of the quotient is 0.

$$\lim_{n \to \infty} \frac{\log n}{n^{1/2}} = \lim_{n \to \infty} \frac{\frac{1}{n} \cdot \log e}{(1/2)n^{-(1/2)}} = \lim_{n \to \infty} \frac{2 \log e}{n^{1/2}} = 0.$$

Overview of the Lesson

1. Issues in Determining Running Time
2. Pseudo-code to Describe Algorithms
3. RAM Model and Counting Primitive Operations
4. Growth Rates and Limits at Infinity
5. Theta (Θ), little-oh (o), little-omega (ω)
6. Big-oh (O) and big-omega (Ω)
7. Asymptotic Algorithm Analysis
8. The Sum of Two Problem
9. Running Time of Recursive Algorithms: Guessing, Counting Self-Calls and The Master Formula
10. Proving Correctness: Iterative Algorithms
11. Proving Correctness: Recursive Algorithms

Big-oh and Big-omega

- If f(n) *grows no faster* than g(n), we say f(n) is O(g(n)) ("big-oh")

- If f(n) *grows at least as fast* as g(n), we say f(n) is Ω(g(n)) ("big-omega")

- The Big-oh notation gives an upper bound on the growth rate of a function; The Big-omega notation gives an lower bound on the growth rate of a function.

- Limit criterion:

$$\text{f(n) is O(g(n)) if}$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \text{ is finite}$$

Then, f(n) is Ω(g(n)) if g(n) is O(f(n)).

Examples

- Both $2n + 1$ and $3n^2$ are $O(n^2)$
- Both $2n^2 - 1$ and $4n^3$ are $\Omega(n^2)$

Relationships Between the Complexity Classes
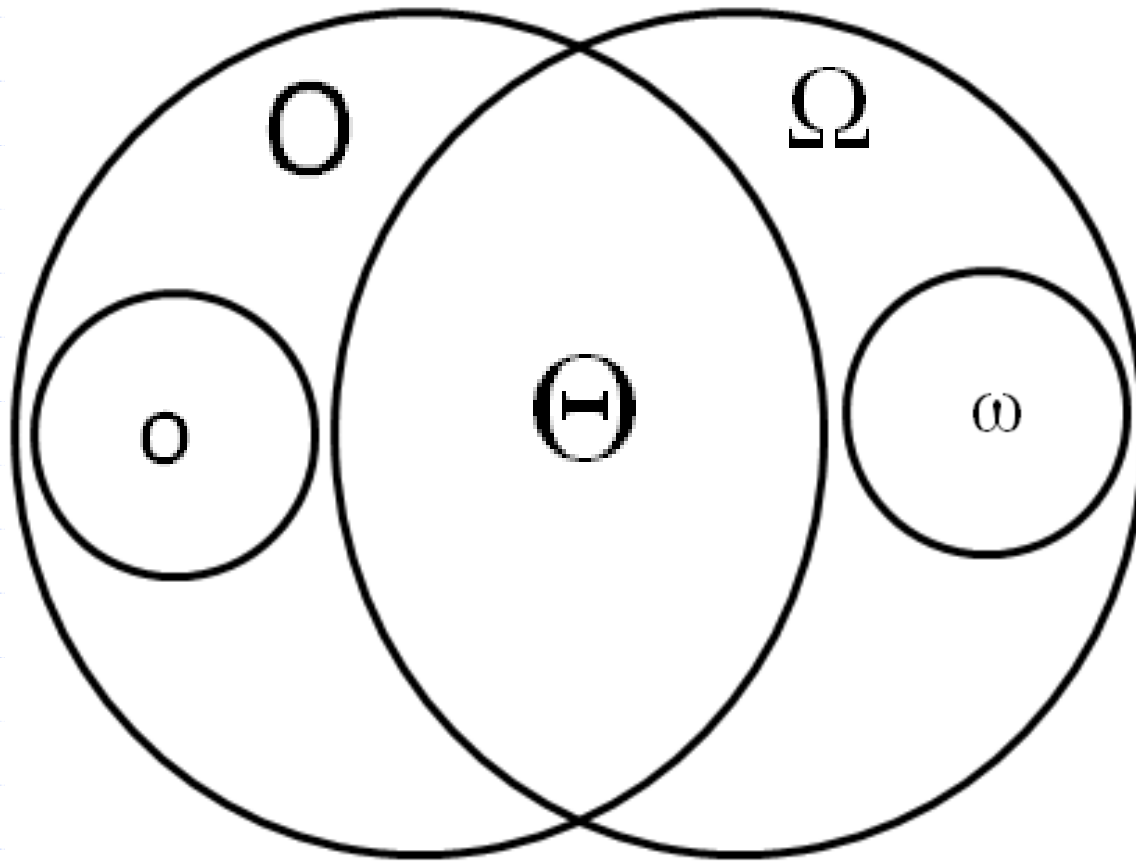
- Whenever f(n) is o(g(n)), f(n) is O(g(n)).

- Whenever f(n) is ω(g(n)), f(n) is Ω(g(n)).

- No function is in both o and ω

- If f(n) is in both O(g(n)) and Ω(g(n)), it is in Θ(g(n)).

Summary of Criteria for Determining Complexity

$f(n)$ is $O(g(n))$	if	$\displaystyle\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is finite
$f(n)$ is $\Omega(g(n))$	if	$\displaystyle\lim_{n\to\infty} \frac{g(n)}{f(n)}$ is finite
$f(n)$ is $\Theta(g(n))$	if	$\displaystyle\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is nonzero
$f(n)$ is $o(g(n))$	if	$\displaystyle\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n)$ is $\omega(g(n))$	if	$\displaystyle\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Overview of the Lesson

1. Issues in Determining Running Time
2. Pseudo-code to Describe Algorithms
3. RAM Model and Counting Primitive Operations
4. Growth Rates and Limits at Infinity
5. Theta (Θ), little-oh (o), little-omega (ω)
6. Big-oh (O) and big-omega (Ω)
7. Asymptotic Algorithm Analysis
8. The Sum of Two Problem
9. Running Time of Recursive Algorithms: Guessing, Counting Self-Calls and The Master Formula
10. Proving Correctness: Iterative Algorithms
11. Proving Correctness: Recursive Algorithms

Asymptotic Algorithm Analysis

- Asymptotic analysis of an algorithm looks at the growth rate of T(n) as n approaches infinity.

- Asymptotic analysis of an algorithm determines which complexity class the running time belongs to.

- 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 big-Oh notation (or one of its variants)

- 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". Or we could say *arrayMax* runs in $\Theta(n)$ 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 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 $\Theta(n^2)$ – often we say 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) = 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.

Overview of the Lesson

1. Issues in Determining Running Time
2. Pseudo-code to Describe Algorithms
3. RAM Model and Counting Primitive Operations
4. Growth Rates and Limits at Infinity
5. Theta (Θ), little-oh (o), little-omega (ω)
6. Big-oh (O) and big-omega (Ω)
7. Asymptotic Algorithm Analysis
8. The Sum of Two Problem
9. Running Time of Recursive Algorithms: Guessing, Counting Self-Calls and The Master Formula
10. Proving Correctness: Iterative Algorithms
11. Proving Correctness: Recursive Algorithms

Example:
The Sum of Two problem

The problem: Given an array A of distinct integers and another integer z, determine whether A contains numbers x, y so that x + y = z.

Sum of Two, Algorithm #1

Algorithm sumOfTwo_1(A, z)

    Input: An array A of distinct integers, and
          an integer z

    Output: true if there are x, y in A
          with x + y = z, else false

    for i ← 0 **to** A.length -1 **do**

        for j ← 0 **to** i − 1 **do**

            if A[i] + A[j] = z **then**

                return true

    return false

Analysis

- Nested for-loops with inner variable dependent on outer variable:

- The *running time of sumOfTwo_1 is* $\Theta(n^2)$

How to Improve?
Notice that the inner loop examines the same elements many times. Find a way to keep hold of these elements after they have been examined once.

Sum of Two, Algorithm #2

Algorithm sumOfTwo_2(A, z)

    Input: An array A of distinct integers, and

        an integer z

    Output: true if there are x, y in A

        with x + y = z, else false

    h $\leftarrow$ new HashMap

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

       val $\leftarrow$ A[i]

      if h.containsKey(z - val) **then**

         return true

      h.put(val, 0)

   return false;

Analysis

<u>Note</u>: As we will show in a later lesson, the operations "put" and "containsKey" on a properly defined hashtable have O(1) running times.

- One for loop

- *Running time of* sumOfTwo_2 *is* O(n)

Main Point

One can improve the running time of a solution to the Sum of Two problem from $O(n^2)$ to $O(n)$ by introducing a hashtable as a means to encapsulate "bookkeeping" operations. 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.

Overview of the Lesson

1. Issues in Determining Running Time
2. Pseudo-code to Describe Algorithms
3. RAM Model and Counting Primitive Operations
4. Growth Rates and Limits at Infinity
5. Theta (Θ), little-oh (o), little-omega (ω)
6. Big-oh (O) and big-omega (Ω)
7. Asymptotic Algorithm Analysis
8. The Sum of Two Problem
9. <u>Running Time of Recursive Algorithms: Guessing, Counting Self-Calls and The Master Formula</u>
10. Proving Correctness: Iterative Algorithms
11. Proving Correctness: Recursive Algorithms

Running Time of Recursive Algorithms: Guessing

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)

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 $\leftarrow$ (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)

Example

Search key x = 7

(1 2 5 7 12 **14** 21 24 25 38 52)

search left

(1 2 **5** 7 12)

search right

(**7** 12)

A[mid] = 7 $\Rightarrow$ return true

Example

Search key x = 20

(1 2 5 7 **12** 14 21 24 25 38)

search right

(14 21 **24** 25 38)

search left

(**14** 21)

search right

(**21**)

search left

lower > upper $\Rightarrow$ return false

Binary Search Running Time

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	+1
mid ← (upper + lower)/2	+3
if x = A[mid] **then** **return** true	+2
if x < A[mid] **then**	+2
return binSearch(A, x, lower, mid − 1)	
else	
return binSearch(A, x, mid + 1, upper)	+3 + T(n/2)

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 exactly half the size of the original.)

$$T(1) = 13; \quad T(n) = 11 + T(n/2)$$

Solving A Recurrence Relation: The Guessing Method

T(1) = 13

T(2) = 11 + 13

T(4) = 11 + 11 + 13

T(8) = 11 + 11 + 11 + 13

$T(2^m)$ = 11 * m + 13

T(n) = 11 * log n + 13, which is O(log n)

- ◆ Guessing method requires us to guess the general formula from a few small values.
- ◆ When using the guessing method, final formula should be verified. Sometimes induction is needed for this, though in simple cases, a direct verification is possible.

Verification of Formula

Claim The function f(n) = 11 * log n + 13 is a solution to the recurrence

$$T(1) = 13; \quad T(n) = T(n/2) + 11 \quad (n \text{ a power of 2})$$

Proof: Prove that whenever n is a power of 2, f(n) satisfies the recurrence. Since n is a power of 2, we write $n = 2^m$. Treating n as a power of 2, f(n) can be written as

$f(2^m) = 11 * m + 13.$

We must show that f(1) = 13 and $f(2^m) = f(2^{m-1}) + 11$

For n = 1, we have:

f(1) = 11 * log 1 + 13 = 13

In general,

$$
\begin{aligned}
f(2^m) &= 11 * m + 13 \\
       &= 11 * (m-1) + 11 + 13 \\
       &= (11 * (m-1) + 13) + 11 \\
       &= f(2^{m-1}) + 11,
\end{aligned}
$$

as required.

(Note: Induction is not required in this case.)

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?

Optional: Not a Power of 2 – Main Result

◆ A number-theoretic function f(n) is <u>smooth</u> if f(n) is eventually nondecreasing and if f(2n) is $\Theta(f(n))$

◆ <u>Theorem</u>. Suppose f(n) is smooth and

T(n) is eventually nondecreasing. Then

if T(n) is $\Theta(f(n))$ for n a power of 2, T(n)

is $\Theta(f(n))$.

Note: All complexity functions f(n) we consider (except exponential functions) will be smooth, and running time functions T(n) will always be eventually nondecreasing. We will not verify this point in the future.

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:

- **<u>Divide</u>** the problem into subproblems (divide input array into left and right halves)
- **<u>Conquer</u>** the subproblems by solving them recursively (search recursively in whichever half could potentially contain target element)
- **<u>Combine</u>** 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.

Overview of the Lesson

1. Issues in Determining Running Time
2. Pseudo-code to Describe Algorithms
3. RAM Model and Counting Primitive Operations
4. Growth Rates and Limits at Infinity
5. Theta (Θ), little-oh (o), little-omega (ω)
6. Big-oh (O) and big-omega (Ω)
7. Asymptotic Algorithm Analysis
8. The Sum of Two Problem
9. Running Time of Recursive Algorithms: Guessing, Counting Self-Calls and The Master Formula
10. Proving Correctness: Iterative Algorithms
11. Proving Correctness: Recursive Algorithms

Running Time of Recursive Algorithms: 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.

Running time for Binary Search, Counting Self-Calls

◆ We can compute asymptotic running time of Binary Search using the technique of counting self-calls.

◆ Observation1: 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}, . . ., 2^1, $2^0 = 1$

is $m + 1 = 1 + \log n$.

◆ Observation2: 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, . . . , $n/2^m = 1$

where 2^m is the largest power of 2 that is $\leq n$, has

exactly $1 + m = 1 + \lfloor \log n \rfloor$ terms

◆ In the worst case for BinarySearch, input size n is cut in half with each successive self-call. Therefore, the successive input sizes of self-calls is n/2, n/4, . . . , $n/2^m = 1$, 0, so $1 + \lfloor \log n \rfloor$ self-calls are made. Running time is therefore O(log n).

Descending Sequences

Here are facts used in the last slide. We will revisit these in later lectures.

◆ If n is a power of 2, say $n = 2^m$, then the number of terms in the sequence

$$2^m, 2^{m-1}, 2^{m-2}, \ldots, 2^1, 2^0 = 1$$

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

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

where 2^m is the largest power of 2 that is $\leq n$, has

exactly $1 + m = 1 + \lfloor \log n \rfloor$ terms

Overview of the Lesson

1. Issues in Determining Running Time
2. Pseudo-code to Describe Algorithms
3. RAM Model and Counting Primitive Operations
4. Growth Rates and Limits at Infinity
5. Theta (Θ), little-oh (o), little-omega (ω)
6. Big-oh (O) and big-omega (Ω)
7. Asymptotic Algorithm Analysis
8. The Sum of Two Problem
9. <u>Running Time of Recursive Algorithms: Guessing, Counting Self-Calls and The Master Formula</u>
10. Proving Correctness: Iterative Algorithms
11. Proving Correctness: Recursive Algorithms

The Master Formula

For recurrences that arise from Divide-and-Conquer algorithms (like Binary Search), there is a general formula for finding a closed-form solution:

Theorem. Suppose $T(n)$ satifies

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

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

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

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

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

Master Formula - Exercise

- Use Master Formula to compute the running time of BinarySearch.

Another example : The Fib Algorithm

- Next we will look at an algorithm that requires more advanced methods to determine the running time.
- To do this, we will make use of the techniques we have learned so far and develop them further.

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 nth Fibonacci number:

    Algorithm fib(n)

        Input: a natural number n
        Output: F(n)

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

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

The running time of recursive Fib Algorithm is given by the following ***Recurrence Relation:***
$$T(1) = c; \quad T(n) = T(n-1) + T(n-2) + d$$
(c, d are some constants)

The Fib Algorithm

- How to compute T(n)?

$$T(n) = T(n-1) + T(n-2) + d$$
$$\geq T(n-2) + T(n-2) + d$$
$$\geq 2T(n-2)$$

Lemma. Suppose $T(1) = c$, $T(n) \geq 2T(n-2)$. Define a recurrence $S(1) = c$, $S(n) = 2S(n-2)$. Then for all n,

$$T(n) \geq S(n)$$

Proof. Proceed by induction on n to show $T(n) \geq S(n)$. This is obvious for $n = 1$. Assume $T(k) \geq S(k)$ whenever $k < n$. Then

$$T(n) \geq 2T(n-2) \geq 2S(n-2) = S(n)$$

In particular, if it can be shown that $S(n)$ is $\Theta(g(n))$, then $T(n)$ is $\Omega(g(n))$.

Solving A Recurrence Relation: The Guessing Method

Question: How to compute $S(n)$ given that $S(1) = c$, $S(n) = 2S(n-2)$?

$S(1) = c$

$S(3) = 2*S(1) = 2*c$

$S(5) = 2*S(3) = 2*2*c = 2^2 c$

$S(7) = 2*S(5) = 2*2*2*c = 2^3 c$

$S(9) = 2*S(7) = 2*2*2*2*c = 2^4 c$

$S(n) = 2^{n/2} * c = \left(\sqrt{2}\right)^n * c$, which is $\Theta\left(\left(\sqrt{2}\right)^n\right)$

Verification of Formula

Claim The function $f(n) = 2^{n/2} * c$ is a solution to the recurrence
$$S(1) = c, \quad S(n) = 2S(n-2).$$

Proof:

For n = 1, we have

$f(1) = 2^0 * c = c$

In general,

$f(n) = 2^{n/2} * c = 2 * 2^{(n-2)/2} * c = 2f(n-2)$

as required.

The Fib Algorithm

By guessing method, we know that S(n) is $\Theta(\sqrt{2})^n$, therefore T(n) is $\Omega(\sqrt{2})^n$
This shows that fib is an *exponentially slow* algorithm!

An algorithm is said to have an *exponential running time* if its running time is $\Theta(r^n)$ for some r > 1. We have shown here that fib is either exponential or worse! But it can be shown there is a number φ for which T(n) is $\Theta(\varphi^n)$
[φ is called the *Golden Ratio*]

DEMO: See the code that accompanies this lecture.

==

Fact: Not only recursive fib algorithm runs exponentially, the Fibonacci numbers grow exponentially too. We will use this fact in later lecture.

In-class Exercise: For all n > 7, F(n) > $(\sqrt{2})^n$

Overview of the Lesson

1. Issues in Determining Running Time
2. Pseudo-code to Describe Algorithms
3. RAM Model and Counting Primitive Operations
4. Growth Rates and Limits at Infinity
5. Theta (Θ), little-oh (o), little-omega (ω)
6. Big-oh (O) and big-omega (Ω)
7. Asymptotic Algorithm Analysis
8. The Sum of Two Problem
9. Running Time of Recursive Algorithms: Guessing, Counting Self-Calls and The Master Formula
10. Proving Correctness: Iterative Algorithms
11. Proving Correctness: Recursive Algorithms

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.

◆ 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: A 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 ith pass through the loop, for each i.

Typically, showing that a statement I(i) is a loop invariant requires an argument by induction on i.

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 of iterativeFactorial:

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 (for k < n), 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:

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

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.

Overview of the Lesson

1. Issues in Determining Running Time
2. Pseudo-code to Describe Algorithms
3. RAM Model and Counting Primitive Operations
4. Growth Rates and Limits at Infinity
5. Theta (Θ), little-oh (o), little-omega (ω)
6. Big-oh (O) and big-omega (Ω)
7. Asymptotic Algorithm Analysis
8. The Sum of Two Problem
9. Running Time of Recursive Algorithms: Guessing, Counting Self-Calls and The Master Formula
10. Proving Correctness: Iterative Algorithms
11. Proving Correctness: Recursive Algorithms

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(j) correctly computes j! Whenever j < n, we must show output of recursiveFactorial(n) is correct. But output of recursiveFactorial(n) is n * recursiveFactorial(n-1) = n*(n-1)!, 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 <= 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.
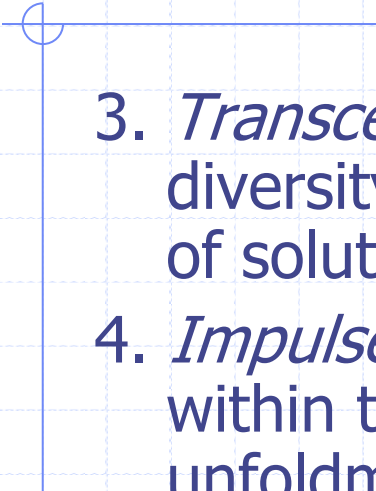
Optional Exercise: Prove mc91 is correct.

Overview of the Lesson

1. Issues in Determining Running Time
2. Pseudo-code to Describe Algorithms
3. RAM Model and Counting Primitive Operations
4. Growth Rates and Limits at Infinity
5. Theta (Θ), little-oh (o), little-omega (ω)
6. Big-oh (O) and big-omega (Ω)
7. Asymptotic Algorithm Analysis
8. The Sum of Two Problem
9. Running Time of Recursive Algorithms: Guessing, Counting Self-Calls and The Master Formula
10. Proving Correctness: Iterative Algorithms
11. Proving Correctness: Recursive Algorithms

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.