Maharishi University of Management

Computer Science Department

CS 435

Algorithms:
The Principle of Least Action

Clyde D. Ruby, Ph.D.

CS 435
Algorithms: Analysis And Design

Professor
Clyde D. Ruby, Ph.D.
cruby@miu.edu
x4324

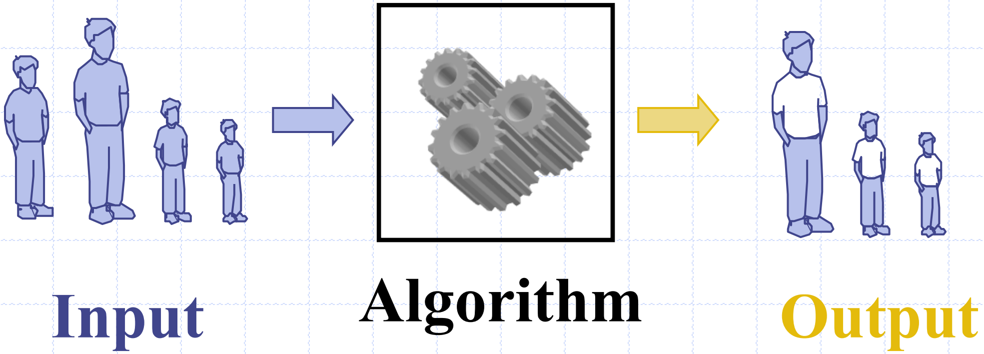
Lecture 1: Theoretical Computer Science or, What problems can computers solve?

Locating infinity in the study of algorithms.

What is an algorithm?

What is an algorithm?

- An **algorithm** is simply a step-by-step procedure for solving a problem in a finite amount of time.
  - Has a unique first step
  - Each step has a unique successor step
  - Sequence of steps terminate with a final result (or …)



Wholeness Statement

The study of algorithms is a core part of computer science and brings the scientific method to the discipline; it has its theoretical aspects (a systematic expression in mathematics), can be verified experimentally, has a wide range of applications, and has a record of achievements.

Science of Consciousness: SCI also has theoretical and experimental aspects, and can be applied and verified universally by anyone.

Overview

- Schedule and Evaluation Criteria
- Origins of Algorithms
- Analysis of Algorithms
  - Computational Complexity
  - Pseudocode
  - Growth Rate of Running Time
  - Asymptotic analysis
    - Big Oh notation
- Math you need to review
  - Exponents
  - Logarithms
  - Probability

Schedule
CS 435: *Algorithms*

Theme	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Foundations of Analysis and Sorting	Introduction and Overview	Stacks, Queues, Vectors, Lists, and Sequences	Trees and Amortized Analysis	Priority Queues, Selection-sort, Insertion-sort, and Heap-sort	Divide-and-Conquer Paradigm: Merge Sort and Quick Sort	Shell Sort, Divide-and-Conquer Analysis & Lower Bound on Sorting by Key Comparison
	Algorithm Analysis	Reading & Homework	Reading & Homework	Reading & Homework	Reading & Homework	Reading & Homework
Searching	Unordered Dictionaries, and Ordered Lookup Tables	Ordered Dictionaries: Binary Search Trees, AVL, and 2-4 Trees	Red-Black Trees	Quick Selection and Linear Sorting Algorithms	Review for Exam	Written Mid-term Exam
	Reading & Homework	Reading & Homework	Reading & Homework	Reading & Homework	Quiz1 & Study	
Techniques and Graphs	Greedy Algorithms, Dynamic Programming, and Memoization	Intro to Graphs and Graph Traversal (DFS)	Graph Traversal (BFS) and Template Methods	Weighted Graphs, & Shortest Paths	Minimum Spanning Trees	P vs. NP Is P = NP?
	Reading & Homework	Lab, Reading & Homework	Reading & Homework	Reading & Homework	Reading & Homework	Reading & Homework
Computability	Complexity Classes NP-Hard and NP-Complete	Approximation Algorithms & P, NP, NPH, NPC Review	Review for Exam	Written Final Exam		
	Reading & Homework	Reading & Homework	Quiz 2 & Study			

- Algorithms in the Computer Science Unified Field Chart

Course Syllabus

Course Goal

- The goal of the course is to learn how to design and analyze various algorithms to solve a computational problem.
- We will achieve this through in-class and homework exercises to help us learn how to
  - break a problem into subtasks,
  - select from a range of possible design strategies,
  - select from a set of abstract data types,
  - Then design an algorithmic solution using pseudo code,
  - evaluate the efficiency of our algorithm to justify the selections used in the design of our solution.
- This goal will also be achieved by studying examples from a range of algorithms, including their design, analysis, implementation, and experimentation. This is a course in problem solving.

Course Objectives

Students should be able to:

1. **Design an algorithm to solve a computational problem based on one or more of the basic design strategies: exhaustive search, divide-and-conquer, greedy, dynamic programming, memoization, randomization, and/or prune-and-search.**
2. Explain and use big O notation to specify the asymptotic space and time complexity (upper bounds) of some specific algorithm’s, e.g., the computational complexity of the principal algorithms for sorting, searching, selection, and hashing.
3. **Create complex algorithms by breaking a problem into subtasks and then using various abstract data structures as building blocks to create efficient solutions.**
4. Explain factors other than computational efficiency that influence the choice of algorithms, such as programming time, simplicity, maintainability, and the use of application-specific patterns in the input data.
5. Understand the importance of the basic complexity classes P, NP, NP-Hard, and NP-Complete and why certain problems will likely remain intractable unless P=NP.
6. Design solutions to graph problems by incorporating the fundamental graph algorithms, including depth-first and breadth-first search, single-source shortest paths, and minimum spanning tree algorithms.
7. Explain the connection between the Science of Consciousness and Algorithm Analysis and Design.

EVALUATION CRITERIA

The course grade will be based on two examinations, several quizzes, lab assignments, class participation, and the Professional Etiquette evaluation with the following weights:

Class Participation and Attendance	5%
Homework, Labs & Quizzes	10%
Midterm Exam	40%
Final Exam	45%

Attendance at all class sessions including labs is required. Unexcused absences or tardiness will reduce a student's final grade.

APPROXIMATE GRADING SCALE

Percent	*Grade*
97 – 100	A+
90 – 97	A
87 – 90	A-
84 – 87	B+
76 – 84	B
73 – 76	B-
70 – 73	C+
62 – 70	C
0 – 62	NC

COURSE TEXTBOOK

The following textbook is was used as the basis for this course. Reading assignments will be made from this text.

- *Algorithm Design: Foundations, Analysis, and Internet Examples*, by M. Goodrich & R. Tamassia, published by Wiley & Sons, 2002.

OTHER REFERENCES

- *An Introduction to Algorithms* by T.H. Cormen, C.E. Leiserson, R.L. Rivest published by McGraw-Hill (1000 pages, difficult reading but a great reference.)

- *The Algorithm Design Manual* by Steve S. Skiena published by Springer-Verlag 1998 (500 pages, a unique and excellent book containing an outstanding collection of real-life challenges, a survey of problems, solutions, and heuristics, and references help one find the code one needs.)

- *Data Structures, Algorithms, and Applications in Java* by Sartaj Sahni published by McGraw-Hill Companion website: http://www.mhhe.com/engcs/compsci/sahnijava/ (Java code for many algorithms.)

- *Foundations of Algorithms, Using Java Pseudocode* by Richard Neapolitan and Kumarss Naimipour published by Jones and Bartlett Publishers, 2004 (600 pages, all mathematics is fully explained; clear analysis)

Daily Schedule

Morning:

10am-12:15pm	lecture (with a break)
12:15-12:30pm	morning meditation

Afternoon:

12:30-1:30pm	lunch
1:30-2:45pm	lecture or homework
2:55-3:20pm	group meditation
3:30-4:00pm	class as needed

Evening:

dinner, homework, rest

Reading Assignments Week 1

Lesson 1a: Overview & Algorithm Analysis	Read pages 4-20, 31-33, 42-46
Lesson 1b: Mathematical Review	Read pages 21-28
Lesson 2: Stacks, Queues, Vectors, Lists, and Sequences	Read pages 56-74
Lesson 3: Amortization and Trees	Read pages 34-41, 75-93
Lesson 4: Priority Queues and Sorting	Read pages 94-113
Lesson 5: Divide-and-Conquer: Merge-Sort and Quick-Sort	Read pages 218-224, Read pages 235-238, 263-267
Lesson 6: Master Method, Sorting Lower Bound	Read pages 268-270,

Reading Assignments Week 2

Lesson 7: Dictionaries: Unordered and Ordered	Read pages 114-127, 140-151
Lesson 8: AVL Trees and 2-4 Trees	Read pages 152-169
Lesson 9: Red-Black Trees	Read pages 170-184, 195-202
Lesson 10: Quick Selection and Linear Time Sorting	Read pages 245-247 Read pages 239-244

Reading Assignments Weeks 3 and 4

Lesson 11: Greedy Method and Dynamic Programming	Read pages 258-262 Read pages 274, 278-281
Lesson 12: Graphs & Graph Traversal	Read pages 288-315
Lesson 13: Weighted Graphs & Shortest Paths	Read pages 340-359
Lesson 14: Minimum Spanning Trees	Read pages 360-375
Lesson 15: P vs. NP or Is P = NP?	Read pages 592-599
Lesson 16: NP-Completeness and Approximation Algorithms	Read pages 499-617, Read pages 618-626
Lesson 17: Directed Graphs	Read pages 316-334

Origins of algorithms

Once upon a time…

- In 1928 in the world of mathematics
  - There existed the hope that a set of axioms (rules) could be identified that could unlock all the truths of mathematics
  - Properly applied, these rules could be applied to solve/prove the validity of any math problem/proposition
  - …and the world would be a better place.

Key figures in the story…

- David Hilbert
  - Tried to find a general algorithmic procedure (a set of rules) for answering all mathematical inquiries

Hilbert's agenda

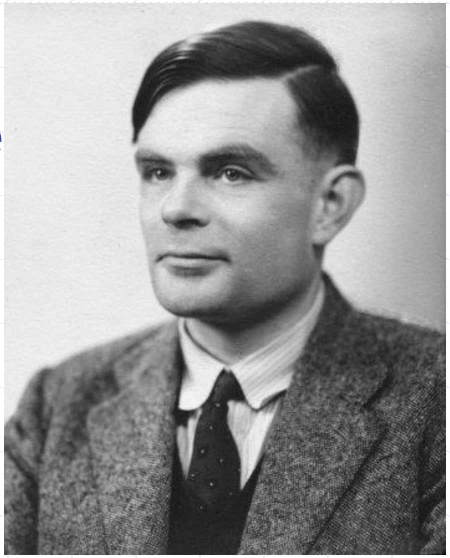
- Three questions at the heart of his agenda were:
  - Is mathematics *consistent*?
    - Can no statement ever be proven both true and false with the rules of math?
  - Is mathematics *complete*?
    - Can every assertion either be proven or disproven with the rules of math?
  - Is mathematics *decidable*?
    - Are there definite steps that would prove or disprove any assertion?

Incompleteness Theorem

- Kurt Gödel wrote a paper in 1931 that shook the math world.
  - Proved that consistency and completeness in math could not be attained.
    - There is no consistent and complete system of formal rules that is comprehensive enough to include arithmetic.
  - SCI point: Total Knowledge cannot be fully described in finite relative terms.
  - Another important math question:
    - Is mathematics *sound*?
      - Is every theorem (proven statement) also true in the relevant model/structure (e.g., in the real world of arithmetic)

Decidability problem

- Alan Turing, sometimes called the "father of computer science".
  - Studied Godel's work in 1935
  - Studied the Problem:
    - Can one find an algorithm to determine whether a mathematical proposition is true or false or are some propositions undecidable?



Halting problem

- *Turing* formalized the concept of an algorithm/program using Turing Machines.
- *Proposition: Given a description of a Turing machine and its initial input, determine whether the program, when executed on this input, ever halts (completes). The alternative is that it runs forever without halting.*

- Self-referral – one Turing machine analyzes another Turing machine

Does this program ever halt?

A perfect number is an integer that is the sum of its positive factors (divisors), not including original number: 6 = 1 + 2 + 3

Algorithm FindOddPerfectNumber()

Input: none

Output: Returns an odd perfect number

n := 1
sum := 0
while *sum* ≠ *n* **do**
  n := *n* + 2
  sum := 0
  for *fact* := 1 to *n/2* **do**
    if *fact* is a factor of *n* **then**
      sum := *sum* + *fact*
return n

What *can* computers do?

- What problems are computable?
  - Theory of computation
- What is the time and space complexity of a problem?
  - Complexity analysis (an important focus of this course)

- Computer models (theory of computation)
  - Deterministic finite state machine
  - Push-down automata
  - “Turing machine” – a tape of instructions
  - Random-access machine (the model we will use)

Main Point

1. The Halting Problem as well as other problems are provably non-computable, i.e., undecidable. That is, there cannot exist a universal method (algorithm) that can be used to determine whether every given mathematical proposition is true or false, such as whether or not a given program halts on a specific input. Such an algorithm would be finite but it would have to answer the question for an infinite number of propositions or programs.

   Science of Consciousness: Total Knowledge, which is infinite, cannot be fully described in the finite relative. However, as we develop higher consciousness, we can experience and thus verify the existence of Total Knowledge as the unbounded silence at the source of thought; this is the basis of human creativity so we can apply it for the good of everything and everyone.

Analysis of Algorithms

Algorithm Analysis

- Answers the questions:
  - Can a problem be solved by algorithm?
  - Can it be solved efficiently?

- Algorithms may look very similar, but be very different

Or

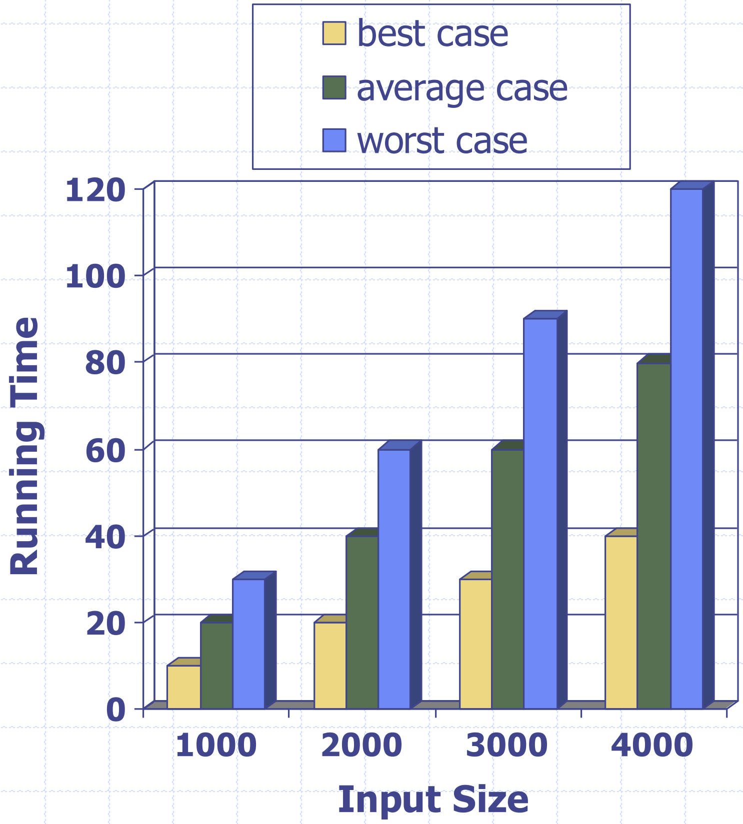
- They may look very different, but be equivalent (in running time)

Computational Complexity

- The theoretical study of time and space requirements of algorithms
- Time complexity is the amount of work done by an algorithm
  - Roughly proportional to the critical (inherently important) operations

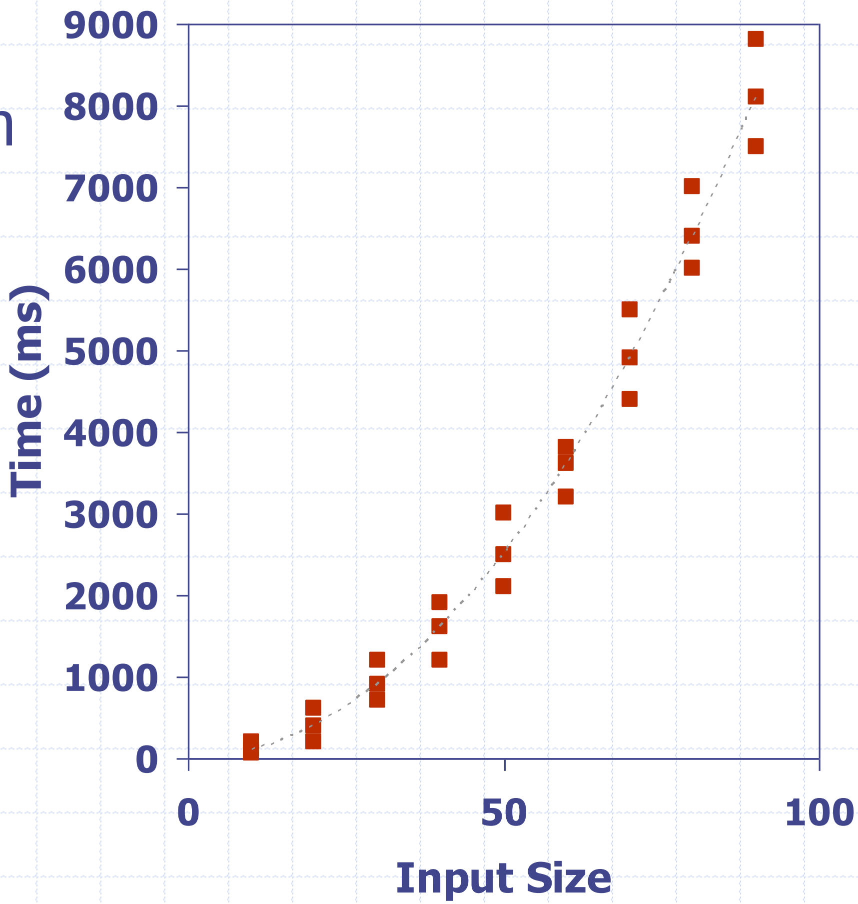
Running Time (§1.1)

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



Experimental Studies (§ 1.6)

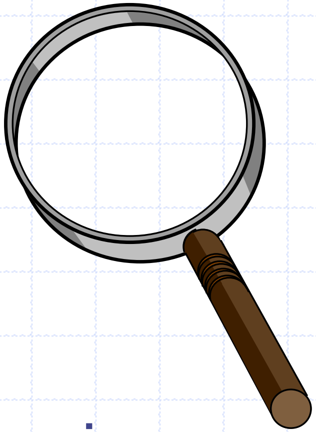
- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- Requires implementation of the algorithm,
  - which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- To compare two algorithms,
  - the same hardware and software environments must be used

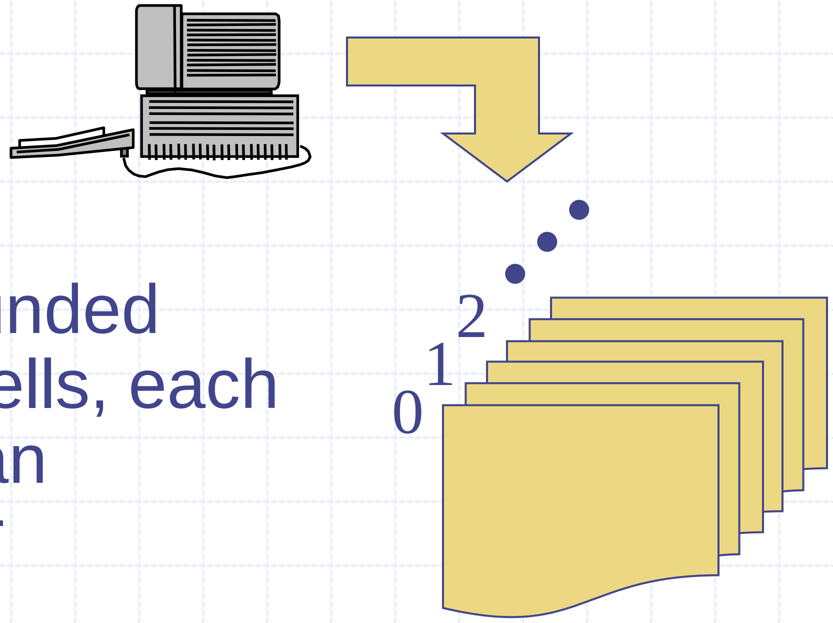
Theoretical Analysis



- How we do it:
  - A high-level description of the algorithm is used
    - instead of an implementation
  - Running time is characterized as a function of the input size, *n*
- This takes into account all possible inputs
- Allows evaluation of the speed of an algorithm independent of the hardware/software environment

The Random Access Machine (RAM) Model

- A **CPU**



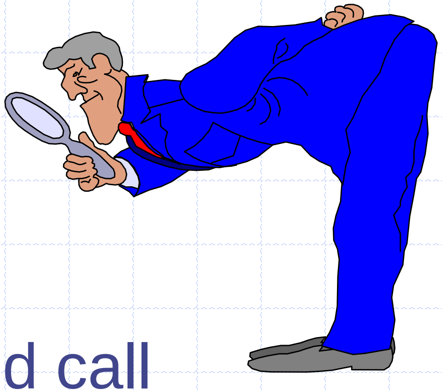
- A potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character

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

What to count and consider

- Significant operations
  - Is it integral to the algorithm or is it overhead or bookkeeping?
  - What are some Examples?
    - Comparison operations
    - Arithmetic operations (evaluating an expression)
    - Assigning a value to a variable
    - Function calls
    - Indexing into an array
    - Following a reference
    - Returning from a method

Pseudocode Details



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

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

  Input …

  Output …

- Method call

  var.method (*arg* [, *arg*…])

- Return value

  return *expression*

- Expressions

  $\leftarrow$ or **:=** Assignment (like $=$ in Java)

  $=$ Equality testing (like $==$ in Java)

  n^2 Superscripts and other mathematical formatting allowed

Pseudocode Example (§1.1)

- 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 := $A[0]$
  for i := 1 **to** $n - 1$ **do**
    if $A[i] >$ *currentMax* **then**
      currentMax := $A[i]$
  return *currentMax*

Primitive Operations

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

Counting Primitive Operations (§1.1)

- Inspect the pseudocode to determine the maximum number of primitive operations executed by an algorithm as a function of the input size

Algorithm *arrayMax*(*A*, *n*) # operations
    currentMax := *A*[0]
    for *i* := 1 **to** *n* – 1 **do**
        if *A*[*i*] > *currentMax* **then**
            currentMax := *A*[*i*]
    { increment counter ***i*** (add & assign) }
    return *currentMax*

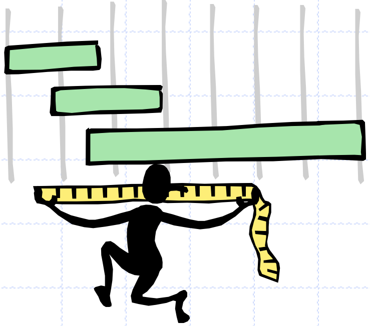
Total

Counting Primitive Operations (§1.1)

- Inspect the pseudocode to 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
for i := 1 **to** $n - 1$ **do**	$1 + n$
if $A[i] >$ ***currentMax*** **then**	$2(n - 1)$
currentMax := $A[i]$	$2(n - 1)$
{ increment counter i (add & assign) }	$2(n - 1)$
return ***currentMax***	1
Total	$7n - 2$

Estimating Running Time



- Algorithm ***arrayMax*** executes $7n - 2$ 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

$$a(7n - 2) \le T(n) \le b(7n - 2)$$

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

Main Point

2. Complexity analysis determines the resources (time and space) needed by an algorithm so we can compare the relative efficiency of various algorithmic solutions. To design an efficient algorithm, we need to be able to determine its complexity so we can compare any refinements of that algorithm so we know when we have created a better, more efficient solution.

   Science of Consciousness: Through regular deep rest (transcending) and dynamic activity we refine our mind and body until our thoughts and actions become most efficient; in the state of enlightenment, the conscious mind operates at the level of pure consciousness, which always operates with maximum efficiency, according to the natural law of least action, so we can spontaneously fulfill our desires and solve even non-computable problems.

Asymptotic Analysis

- **Asymptote**:
  - A line whose distance from a given curve approaches zero as they tend to infinity
    - A term derived from analytic geometry
  - Originates from the Greek word *asumptotos* which means not intersecting
  - Thus an asymptote is a limiting line
- **Asymptotic**:
  - Relating to or having the nature of an asymptote
- **Asymptotic analysis in complexity theory**:
  - Describes the upper (or lower) bounds of an algorithm's behavior in terms of its usage of time and space
  - Used to classify computational problems and algorithms according to their inherent difficulty

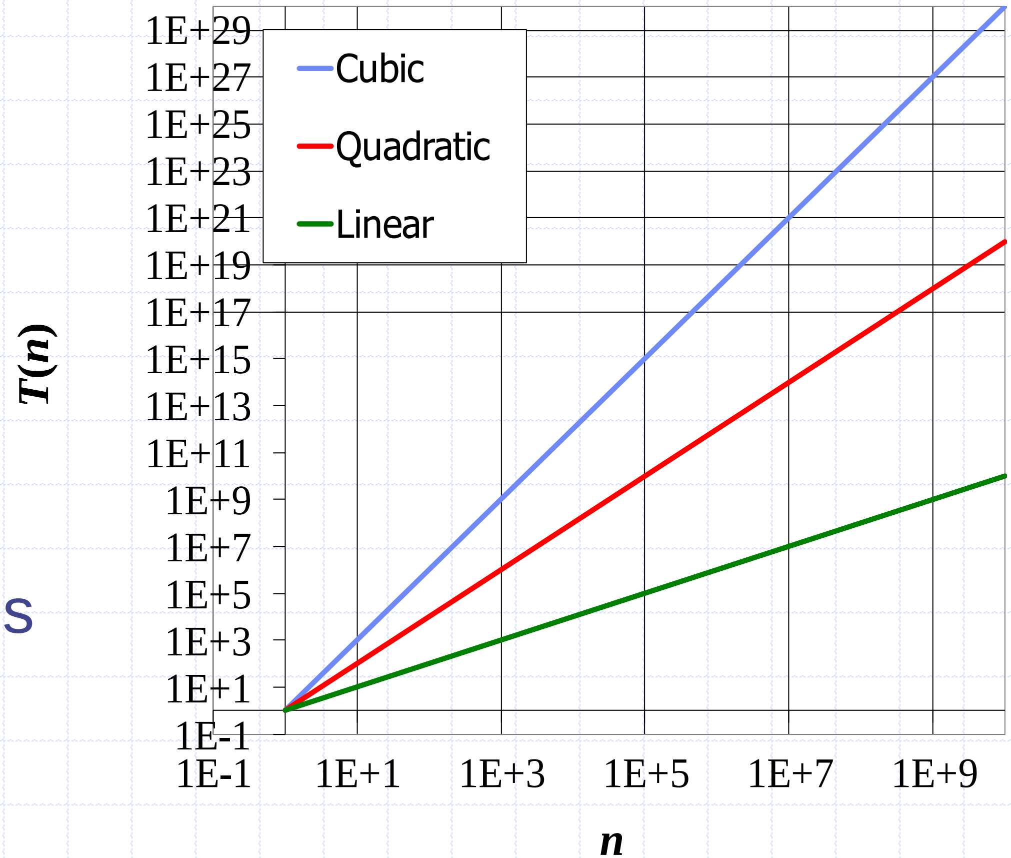
- We are going to classify algorithms in terms of functions of their input size
  - Therefore, how can we draw graphs of quadratic or cubic functions so the graphs look and behave like asymptotes (a limiting line)?

Log-Log Graph

- A two-dimensional graph that uses ***logarithmic scales*** on both the horizontal and vertical axes.
- The scaling of the axes is nonlinear
  - So a function of the form $y = ax^b$ will appear as a straight line
  - Note that
    - b is the slope of the line (*gradient*)
    - a is the y value when $x = 1$

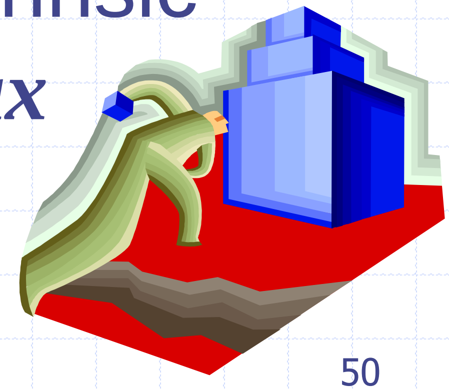
Growth Rates on a Log-Log Graph

- Growth rates of functions:
  - Linear ≈ n
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
- In a log-log chart, the slope of the line (gradient) corresponds to the growth rate of the function



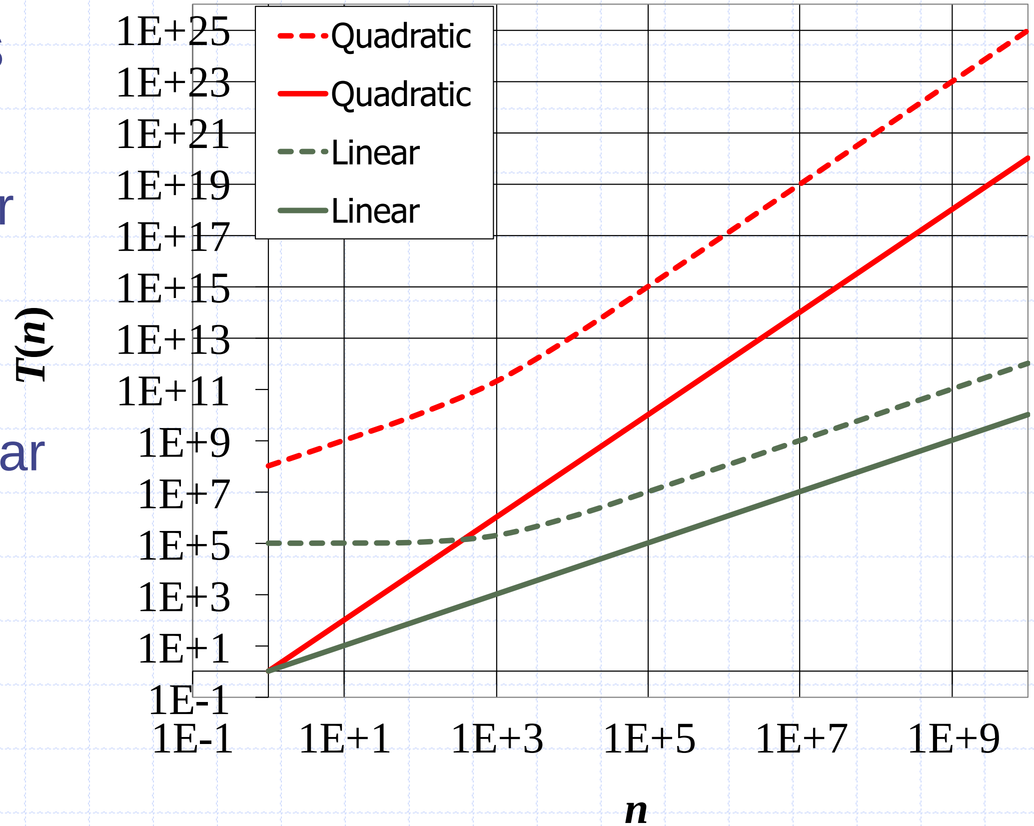
Growth Rate of Running Time

- The hardware/software environment
  - Affects $T(n)$ by a constant factor,
  - But does not alter the asymptotic growth rate of $T(n)$
- For example: The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm ***arrayMax***



Constant Factors

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



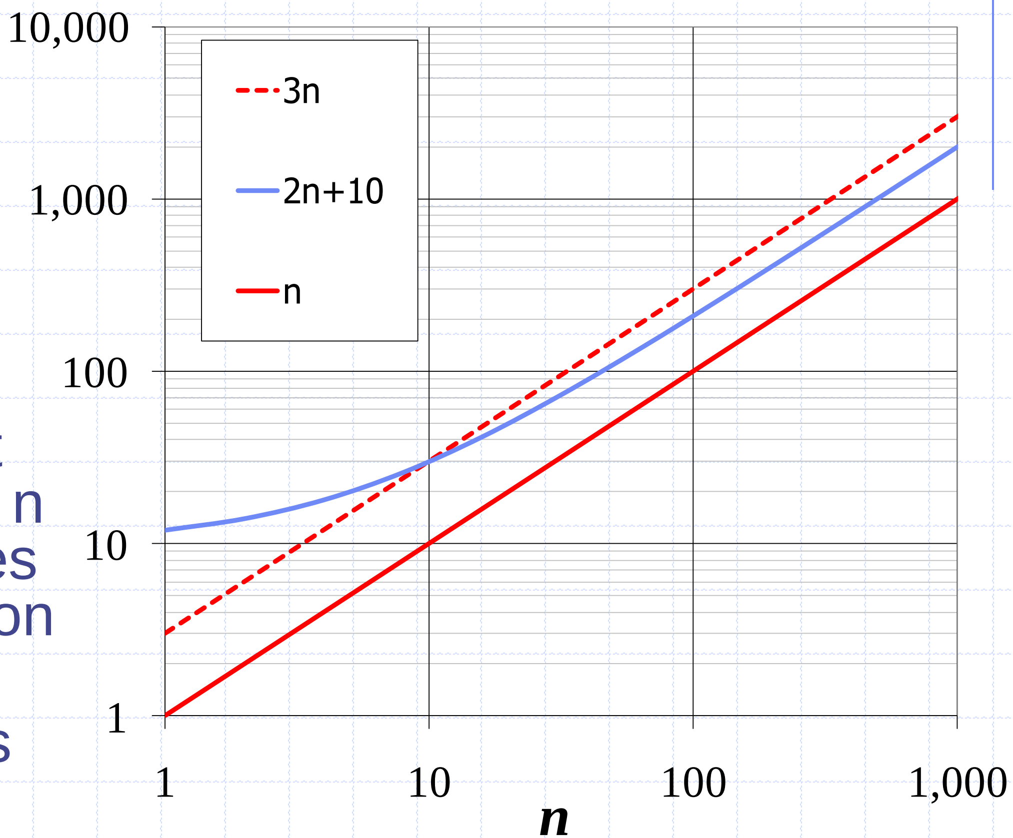
Big-Oh Notation (§1.2)

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

    $$f(n) \leq cg(n) \text{ for } n \geq n_0$$
- Example:
  - prove that $2n + 10$ is $O(n)$

Big-Oh Notation (§1.2)

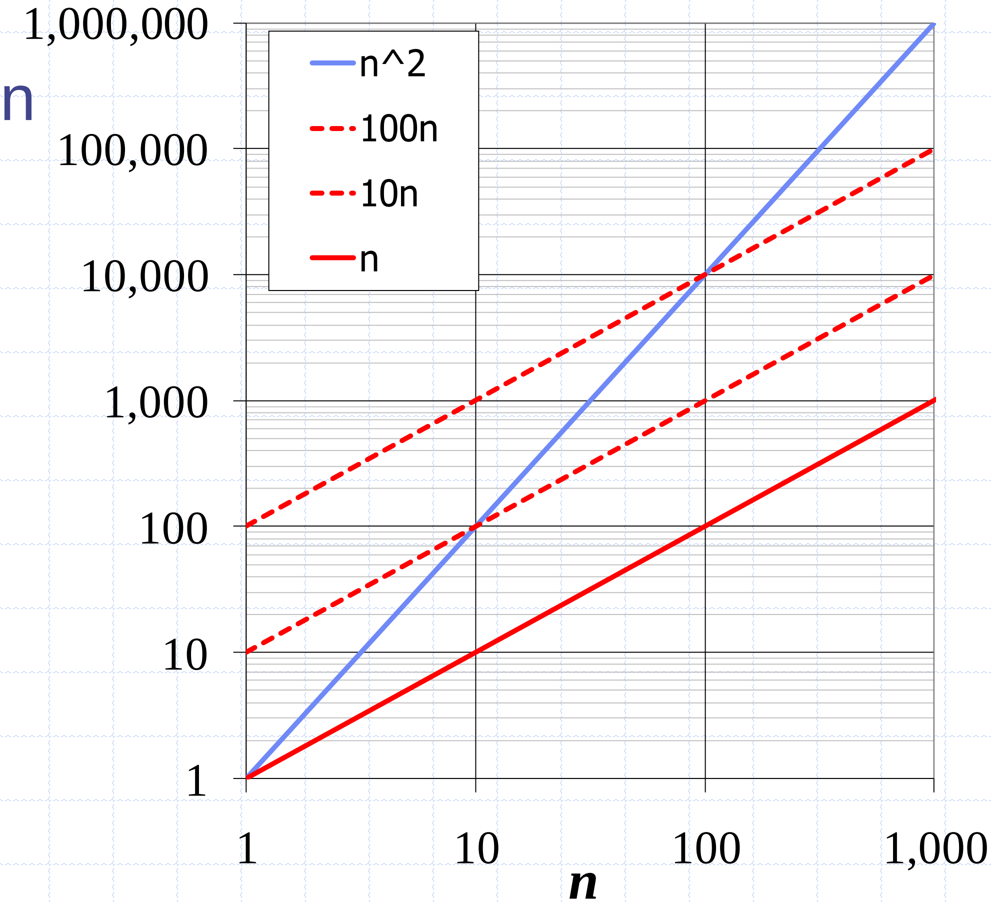
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \le cn$
  - $(c - 2)\, n \ge 10$
  - $n \ge 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



- The graph illustrates that when $c = 3$, the graph for n is shifted up and becomes an upper bound of function $2n$+10
- Note also that the graphs cross when $n = 10$
  - From then on $3n$ is an upper bound of $2n$+10

Big-Oh Example

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



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 Examples

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

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

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

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

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

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

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

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

  this is true for $c = 4$ and $n_0 = 2$

Big-Oh Rules

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- 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 determine that algorithm ***arrayMax*** executes at most $7n-2$ primitive operations
  - We say that algorithm ***arrayMax*** "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped, we can disregard them when counting primitive operations

Counting Primitive Operations using Big-oh Notation

- Why don't we need to precisely count every primitive operation like we did previously?

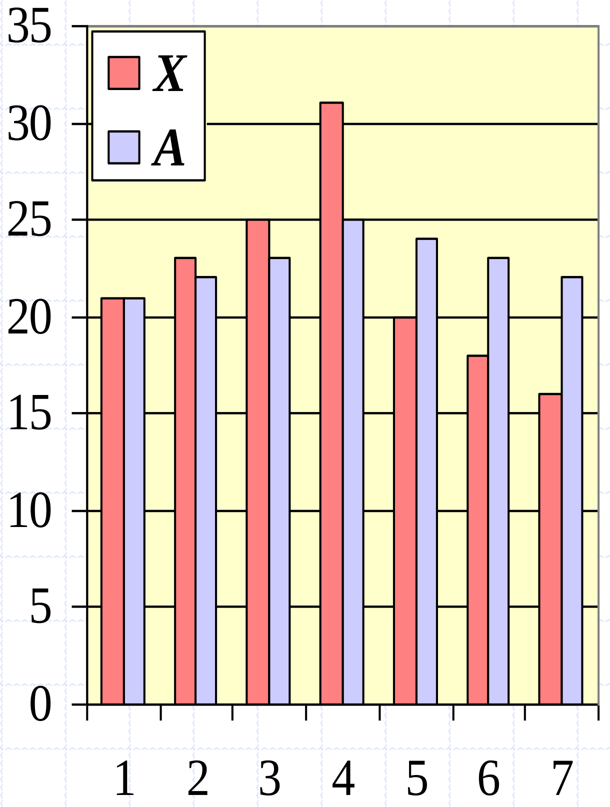
Algorithm *arrayMax*(*A*, *n*)	# operations
currentMax := *A*[0]	$O(1)$
for *i* := 1 **to** *n* – 1 **do**	$O(n)$
if *A*[*i*] > *currentMax* **then**	$O(n)$
currentMax := *A*[*i*]	$O(n)$
{ increment counter *i* (add & assign) }	$O(n)$
return *currentMax*	$O(1)$
Total	$O(n)$

Computing Prefix Averages

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

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

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



Prefix Averages (Quadratic)

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

Algorithm *prefixAverages1*(*X*, *n*)	
Input array X of n integers	
Output array A of prefix averages of X	#operations
A := new array of n integers	n
for $i := 0$ **to** $n - 1$ **do**	n
$s := X[0]$	n
for $j := 1$ **to** i **do**	$1 + 2 + \ldots + (n - 1)$
$s := s + X[j]$	$1 + 2 + \ldots + (n - 1)$
$A[i] := s / (i + 1)$	n
return A	1

Arithmetic Progression

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

Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum

Algorithm ***prefixAverages2*(*X*, *n*)**	
Input array X of n integers	
Output array A of prefix averages of X	#operations
A := new array of n integers	n
$s := 0$	1
for $i := 0$ **to** $n - 1$ **do**	n
$\quad s := s + X[i]$	n
$\quad A[i] := s / (i + 1)$	n
return A	1

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

Optimality

- Can be proven by showing that every possible algorithm has to do at least some number of critical operations to solve the problem
- Then prove that a specific algorithm attains this lower bound

- Simplicity is an important practical consideration!!

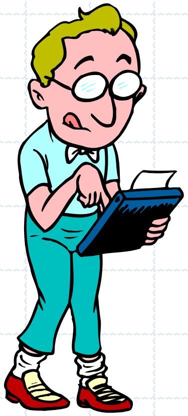
- Course motto: consider efficiency, but favor simplicity

Main Point

3. An algorithm is “optimal” if its computational complexity is equal to the “maximal lower bound” of all algorithmic solutions to that problem; that is, an algorithm is optimal if it can be proven that no algorithmic solution can do asymptotically better.

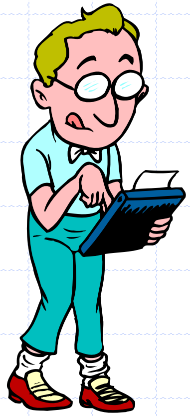
   Science of Consciousness: An individual’s actions are optimal if they are the most effective and life-supporting. Development of higher states of consciousness results in optimal action because thoughts are performed while established in the silent state of pure consciousness, the source of creativity and intelligence in nature.

Math you need to Review



- Summations (Sec. 1.3.1)
- Logarithms and Exponents (Sec. 1.3.2)
- Proof techniques (Sec. 1.3.3)
- Basic probability (Sec. 1.3.4)

Math you need to Review



- Summation Formulas (Sec. 1.3.1)
  - $\Sigma_{i=1}^{n} i = (1 + 2 + \ldots + n\text{-}1 + n) = n(n+1)/2$
  - $\Sigma_{i=n}^{1} i = (n + n\text{-}1 + \ldots + 2 + 1) = n(n+1)/2$
  - Constant factors: $\Sigma_{i=0}^{n} c\, f(i) = c\, \Sigma_{i=0}^{n} f(i)$
  - Summing constants: $\Sigma_{i=1}^{n} c = cn$
  - Sum of powers of 2:

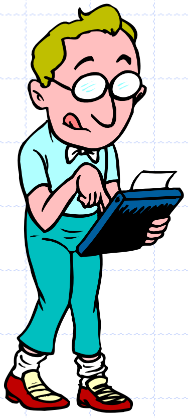
    $$\Sigma_{i=0}^{n} 2^i = 2^{n+1} - 1 = 2 * 2^n - 1 \text{ which is } O(2^n)$$

    (how to remember: consider each power of two is a bit in a binary number)

More summation formulas

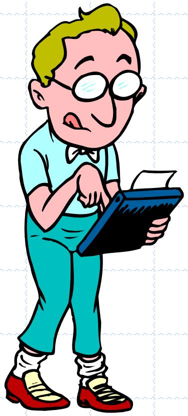
- $\Sigma_{i=0}^{n} 1/2^i = 2 - 1/2^n$
  - What if i starts at 1 instead of 0?
- $\Sigma_{i=1}^{n} i*2^i = (n-1)\,2^{n+1} + 2$
- Sum of squares:
  - $\Sigma_{i=0}^{n} i^2 = (2n^3 + 3n^2 + n) / 6$
- Geometric progressions
  - $\Sigma_{i=0}^{n} a^i = (a^{n+1} - 1)/(a - 1)$

Math you need to Review



- Floor of x
  - The largest integer less than or equal x
- Ceiling of x
  - The smallest integer greater than or equal to x

Math you need to Review



- Exponents (Sec. 1.3.2)

  $a^0 = 1$

  $a^1 = a$

  $a^{-1} = 1/a$

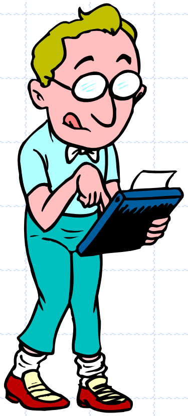
  $a^{-b} = 1/a^{b}$

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

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

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

Math you need to Review



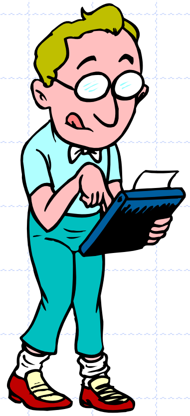
Logarithms and Exponents (Sec. 1.3.2)

$\log_b a = x$ iff $b^x = a$

$a = b^{\log_b a}$

- These are derived from the definition of logarithms;
- all other equalities can be derived from these two rules and the rules for exponents (previous slide)

Math you need to Review



Logarithms and Exponents (Sec. 1.3.2)

$\log_b 1 = 0$

$a^c = b^{c \log_b a} = (b^{\log_b a})^c$

$\log_b b^x = x$

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

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

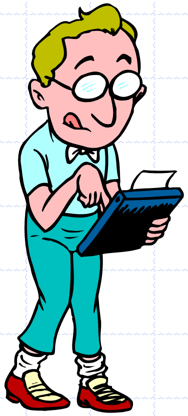
$\log_b (a^c) = c \log_b a$

$(\log_c b)(\log_b x) = (\log_c x)$ (base conversion)

$\log^k n = (\log n)^k$ (exponentiation)

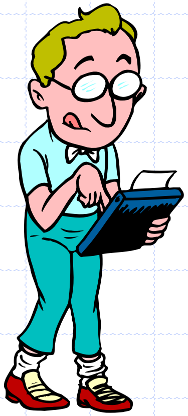
$\log \log n = \log (\log n)$ (composition)

Math you need to Review



- Proof techniques (Sec. 1.3.3)
  - Logic (DeMorgan's Law, etc.)
  - Counterexample
  - Contrapositive
  - Contradiction
  - Induction
  - Vacuous proofs
  - Loop invariants

Math you need to Review



- Basic probability (Sec. 1.3.4)
  - Events
    - Independent
    - Mutually independent
  - Probability space
  - Random variables (independent)
    - A function that maps outcomes from some sample space S to real numbers (usually the interval {0, 1} to indicate the probability)
  - Expected values
  - Motivation (need to know the likelihood of certain sets of input)
    - Usually assume all are equally likely
    - E.g., if N possible sets, then 1/N is the probability

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. An algorithm is like a recipe to solve a computable problem starting with an initial state and terminating in a definite end state.

2. To help develop the most efficient algorithms possible, mathematical techniques have been developed for formally expressing algorithms (pseudocode) so their complexity can be measured through mathematical reasoning and analysis; these results can be further tested empirically.

3. **<u>Transcendental Consciousness</u>** is the home of all knowledge, the source of thought. The TM technique is like a recipe we can follow to experience the home of all knowledge in our own awareness.
4. **<u>Impulses within Transcendental Consciousness</u>**: Within this field, the laws of nature continuously calculate and determine all activities and processes in creation.
5. **<u>Wholeness moving within itself</u>** : In unity consciousness, all expressions are seen to arise from pure simplicity--diversity arises from the unified field of one's own Self.