Maharishi University of Management

Computer Science Department

CS 421

Algorithms:
The Principle of Least Action

Clyde D. Ruby, Ph.D.

1

CS 421
Algorithms: Analysis And Design

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

2

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

Locating infinity in the study of algorithms.

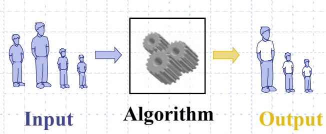
3

What is an algorithm?

4

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 …)



5

Why study algorithms?

- You might ask, “why would I want to know any of the details of the sorting algorithms?”
  - If I want to sort, I’ll just call a library sort routine.
- This is fine if you’re going to remain a programmer in a simplistic since,
  - i.e., just get the job done without regard to efficiency, best practices, or high-level knowledge of how things are done by API’s

6

Computer Science Majors

- CS majors should at least be exposed to and have a basic understanding of what the different sorting algorithms do because
  - there are tradeoffs between the different sorting algorithms
  - they illustrate important algorithm design strategies
    - We need people who can develop and implement new algorithms based on exposure to well designed algorithms
    - Also, we don't have to be great at math, but we need some math to push computer science forward (particularly in algorithms)

Decomposition & Abstraction

- What do we mean by decomposition?
  - Divide the problem into small programs (components) that interact in simple, well-defined ways
  - In theory, different people should be able to work on different components (subprograms) independently
- What do we mean by abstraction?
  - It means ignoring details (how) and focusing on what needs to be done to simplify the original larger problem and its solution

Abstraction Mechanisms

- Abstraction by parameterization
  - Using parameters to represent a potentially infinite set of different computations
  - Parameterized functions or types, for example:

```
square (x) { return x * x; }
```

- Abstraction by specification
  - Associating a specification with a function/operation/type
  - Then using the specification rather than the code to reason about what happens when a function is called
  - Example: using the behavioral specification of an API rather than knowing/seeing the program code written for the API

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.

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

Overview

- Schedule and Evaluation Criteria
- Importance of Abstraction
- 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 421
Algorithms

Schedule

Theme	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Foundations, Analysis, and Abstract Data Types	Introduction and Overview: Abstraction and Complexity	Stacks, Queues, and Lists	Sequences: Array-Based Containers	Recursion	Binary Trees	The Heap
	Algorithm Analysis	Study & Homework	Study & Homework	Study & Homework	Reading & Homework	Reading & Homework
Sorting	Selection-Sort, Insertion-Sort Heap-Sort, and Shell-Sort	Priority Queues, PQ-sorting, and Radix-sort	Merge-Sort	Quick-Sort	Review for Exam	Mid-term Exam
	Reading & Homework	Reading & Homework	Reading & Homework	Reading & Homework	Study	
Searching Techniques and Strategies	Dictionaries: Log files, Hash Tables, and Lookup Tables	Ordered Dictionaries:, Balanced BST's, and AVL Trees	2-4 Trees	Ordered Dictionaries: Red-Black Trees	Introduction to Graphs and Graph Traversal: DFS	Graph Traversal: BFS and Template Methods
	Reading & Homework	Lab, Reading & Homework	Reading & Homework	Reading & Homework	Reading & Homework	Reading & Homework
Graphs	Weighted Graphs & Shortest Paths	Minimum Spanning Trees	Review for Exam and Short Answer Quiz	Final Exam		
	Reading & Homework	Reading & Homework	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, including how to evaluate algorithm efficiency, select from a range of possible design strategies and/or abstract data types, and justify those selections in the design of a solution. This goal will be achieved by exploring a range of algorithms, including their design, analysis, implementation (in JavaScript), and experimentation.

Course Objectives
Students should be able to:

1. Design a pseudo code algorithm to solve a computational problem using one or more of the basic design strategies: exhaustive search, divide-and-conquer, greedy, randomization, and/or decrease-and-conquer.
2. Create complex algorithms using various abstract data structures as building blocks in pseudo code and JavaScript algorithms.
3. Translate a pseudo code algorithm into a JavaScript program.
4. Understand through direct experience why it is **important to design the algorithm** before coding in JavaScript (or any programming language).
5. Explain and use big O notation to specify the asymptotic bounds of an algorithm's space and time complexity, e.g., the computational complexity of the principal algorithms for sorting, searching, selection, and hashing.
6. 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.
7. 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.
8. Explain the connection between the Science and Technology 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*
90 – 100	A
87 – 90	A-
84 – 87	B+
76 – 84	B
73 – 76	B-
70 – 73	C+
62 – 70	C
0 – 62	NC

NO COURSE TEXTBOOK

The data structures that I provide will be variations and adaptations from the ones in the following book.

- *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, C. Stein published by The MIT Press, 2009 (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 and Algorithms in Java, 4th Ed.* by M. Goodrich & R. Tamassia, published by Wiley & Sons, 2006.
- *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

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)

Natural Things to Ask

- How can we determine whether an algorithm is *efficient* ?
- 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 or programming language?

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)

25

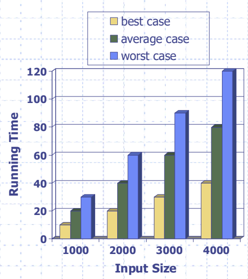
Computational Complexity

- 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

26

Running Time (§1.1)

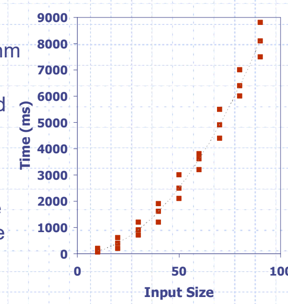
- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- So we usually focus on worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



27

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



28

Limitations of Experiments

- Requires implementation of the algorithm,
  - which may be difficult and/or time consuming
- 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
  - better to compare before actually implementing them (to save time)

29

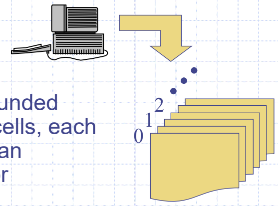
Theoretical Analysis

- 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*
- Takes into account all possible inputs
- Allows evaluation of the speed of an algorithm independent of the hardware/software environment

30

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 unit 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…])

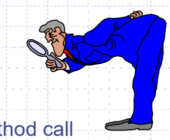
- Return value

  return *expression*

- Expressions

  ← Assignment (like = in Java)
  = Equality testing (like == in JavaScript)
  n^2 Superscripts and other mathematical formatting allowed

Exercise



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

  Algorithm *arrayMax*(A, n)
  Input array A of $n > 0$ integers
  Output maximum element of A

- Method call

  var.method (arg [, arg…])

- Return value

  return *expression*

- Expressions

  ← Assignment (like = in JavaScript)
  = Equality testing (like == in JavaScript)
  n^2 Superscripts and other mathematical formatting allowed

Pseudocode Example

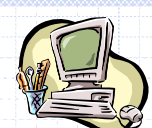
- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

```
Algorithm arrayMax(A, n)
  Input array A of n > 0 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

- 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
```


37

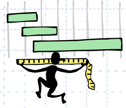
Counting Primitive Operations (§1.1)

- 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
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$1 + n$
if $A[i] > currentMax$ **then**	$2(n - 1)$
$currentMax \leftarrow A[i]$	$2(n - 1)$
{ increment counter i (add & assign) }	$2(n - 1)$
return $currentMax$	1
Total	$7n - 2$

38

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

39

Main Point

1. 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.

40

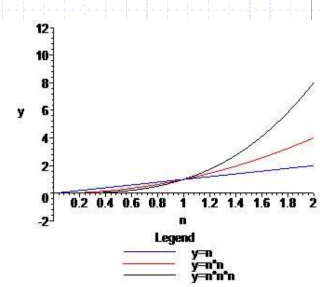
Asymptotic Analysis

- **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)?

41

Growth Rates

- Growth rates of functions:
  - Linear ≈ n
  - Quadratic ≈ n^2
  - Cubic ≈ 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



42

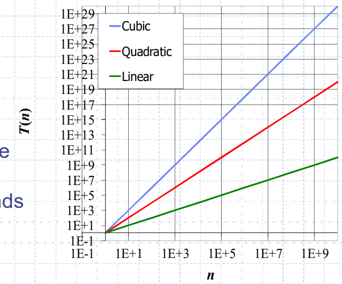
Log-Log Graph

- 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 $x = 1$

43

Growth Rates on a Log-Log Graph

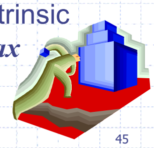
- Growth rates of functions:
  - Linear $\approx n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$
- In a log-log chart, the slope of the line (gradient) corresponds to the growth rate of the function



44

Growth Rate of Running Time

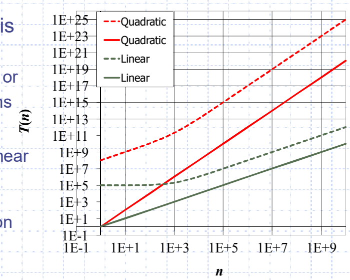
- The hardware/software environment
  - Affects $T(n)$ by a constant factor,
  - But does not alter the asymptotic growth rate of $T(n)$
- For example: The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm ***arrayMax***



45

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



46

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

    $f(n) \le cg(n)$ for $n \ge n_0$
- Example:
  - prove that $2n + 10$ is $O(n)$

47

Big-oh

- If f(n) *grows no faster* than g(n), we say f(n) is O(g(n)) ("big-oh")
- We also say that g(n) is an ***asymptotic upper bound*** on f(n)
- Big-oh is a set of functions
  - O(g(n)) = { f(n) | ∃ c, n_0 : $0 \le f(n) \le cg(n)$ for all $n \ge n_0$ }
- Limit criterion:

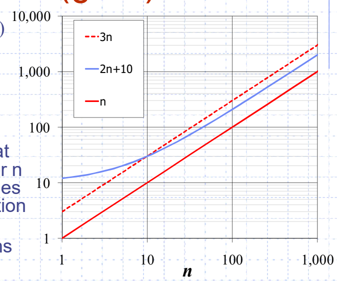
  f(n) is O(g(n)) if

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

48

Big-Oh Notation (§1.2)

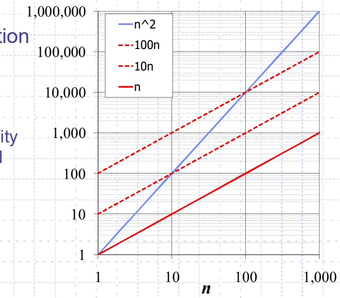
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \le cn$
  - $(c - 2)\, n \ge 10$
  - $n \ge 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$
- The graph illustrates that when $c = 3$, the graph for n is shifted up and becomes an upper bound of function $2n+10$
- Note also that the graphs cross when $n = 10$
  - From then on $3n$ is an upper bound of $2n+10$



49

Big-Oh Example

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



50

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

51

Big-Oh Examples

- 7n-2 is O(n)

need $c > 0$ and $n_0 \ge 1$ such that $7n\text{-}2 \le c\bullet n$ for $n \ge n_0$
this is true for $c = 7$ and $n_0 = 1$

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

need $c > 0$ and $n_0 \ge 1$ such that $3n^3 + 20n^2 + 5 \le c\bullet n^3$ for $n \ge n_0$
this is true for $c = 4$ and $n_0 = 21$

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

need $c > 0$ and $n_0 \ge 1$ such that $3 \log n + \log \log n \le c\bullet \log n$ for $n \ge n_0$
this is true for $c = 4$ and $n_0 = 2$

52

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 $O(3n)$"

53

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

54

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 $\leftarrow A[0]$	$O(1)$
for $i \leftarrow 1$ **to** $n - 1$ **do**	$O(n)$
if $A[i] >$ *currentMax* **then**	$O(n)$
currentMax $\leftarrow A[i]$	$O(n)$
{ increment counter i (add & assign) }	$O(n)$
return *currentMax*	$O(1)$
Total	$O(n)$

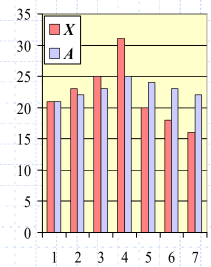
55

Computing Prefix Averages

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

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

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



56

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
$A \leftarrow$ new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow X[0]$	n
for $j \leftarrow 1$ **to** i **do**	$1 + 2 + \ldots + (n - 1)$
$s \leftarrow s + X[j]$	$1 + 2 + \ldots + (n - 1)$
$A[i] \leftarrow s / (i + 1)$	n
return A	1

57

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

58

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
$A \leftarrow$ new array of n integers	n
$s \leftarrow 0$	1
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow s + X[i]$	n
$A[i] \leftarrow s / (i + 1)$	n
return A	1

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

59

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

60

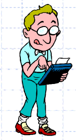
Main Point

2. 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.

61

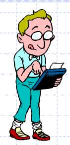
Math you need to Review



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

62

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:

    $\Sigma_{i=0}^{n} 2^i = 2^{n+1} - 1$

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

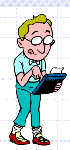
63

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
  - $\Sigma_{i=0}^{n} a^i = (a^{n+1} - 1)/(a - 1)$

64

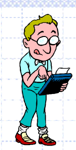
Math you need to Review



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

65

Math you need to Review



- Exponents (Sec. 1.3.2)

  $a^0 = 1$

  $a^1 = a$

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

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

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

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

66

Math you need to Review

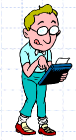
Logarithms and Exponents (Sec. 1.3.2)

$\log_b a = x$ 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)

$\log_b 1 = 0$

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

$\log_b b^x = x$

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

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

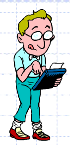
$\log_b (a^c) = c \log_b a$

$\log_b a = (\log_x a) / \log_x b$ (base conversion)

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

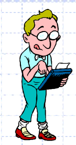
$\log \log n = \log (\log n)$ (composition)

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. **Transcendental Consciousness** 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. **Impulses within Transcendental Consciousness**: Within this field, the laws of nature continuously calculate and determine all activities and processes in creation.
5. **Wholeness moving within itself** : In unity consciousness, all expressions are seen to arise from pure simplicity--diversity arises from the unified field of one's own Self.