CS 2200: Algorithms

Fall, 2018

Administrative Information

- Course Webpage:
  - http://www.bowdoin.edu/~smajerci/teaching/cs2200/2018fall/index.html
- Textbook: Coren, Leiserson, Rivest, and Stein. *Introduction to Algorithms, 3rd edition,* MIT Press, 2009.
- My Office Hours:
  - Monday, 6:00-8:00 pm, Searles 224
  - Tuesday, 1:00-2:30 pm, Searles 222
- TAs (Office Hours TBA):
  - Zoe Aarons
  - Luca Ostertag-Hill
  - Jack Ward
  - Erik Wurman

What you can expect from me

- Strategies for designing algorithms
- When to use those strategies
- Tools for analyzing algorithm efficiency
- Techniques for arguing algorithm correctness (a little)
- Specific algorithms
- Improved problem solving skills
- Improved ability to think abstractly

What I will expect from you

- Labs and Homework Problems (25%):
  - Generally after every two classes
  - In-Lab Problems
  - Homework Problems
  - More a learning tool than a testing tool
- 3 Exams (75%):
  - In class
  - Closed book, closed notes
    - except for one 8.5 x 11 sheet of notes (both sides)

Collaboration Levels

- Level 0 (In-Lab and In-Class Problems)
  - No restrictions on collaboration
- Level 1 (Homework Problems)
  - Verbal collaboration without code sharing
  - But many details about what is allowed
- Level 2 (Not used in this course)
  - Discussions with TAs only

Collaboration Levels

- Level 0 (In-Lab and In-Class Problems)
  - No restrictions on collaboration
- Level 1 (Homework Problems)
  - Verbal collaboration without code sharing
  - But many details about what is allowed
- ~~Level 2 (Not used in this course)~~
  - ~~Discussions with TAs only~~

Collaboration Levels

- Level 0 (In-Lab and In-Class Problems)
  - No restrictions on collaboration
- Level 1 (Homework Problems)
  - Verbal collaboration without code sharing
  - But many details about what is allowed
- ~~Level 2 (Not used in this course)~~
  - ~~Discussions with TAs only~~
- Level 3 (Exams)
  - Professor clarifications ***only***

Algorithms is a Difficult Class!

- Much more abstract than Data Structures:
  - emphasis is on *designing* the solution technique, not *implementing* a solution
- What to do:
  - Allow plenty of time to read the materials and do the homework
  - Solve all problems (even the optional ones)
  - Go to the study groups (TA hours)
  - Form a group to work with
  - Spaced study

Learning

- What helps you?
- What hinders you?

Algorithms and Programs

- An algorithm is a computational recipe designed to solve a particular problem
- Must be implemented as a program in a particular programming language
- Data structures are critical...
- ...but you already know that.

Making a telephone call to Jill

pick up the phone;
dial Jill's number;
wait for person to answer;
talk;

Correctness

Waiting at a traffic light

```
if (light is red) {
        wait a while;
        accelerate;
}
```

Definiteness

Looking for an integer >= 0 with property P.

```
i = 0;
foundIt = testForP(i);
while (!foundIt) {
        i++;
        foundIt = testForP(i);
}
```

Finite number of steps

Packing for vacation

```
flip coin;
if (heads)
        pack paraglider;
else
        pack scuba gear;
```

Predictability

Desirable Characteristics

- THEORY suggests/requires:
  - Correctness
  - Definiteness
  - Finiteness
  - Predictability
- Practice suggests:
  - Efficiency
  - Clarity
  - Brevity

An algorithm is:

…a list of **precisely** defined steps that can be done by a computer in a **finite** (and, hopefully, relatively **short**) amount of **time** to **correctly** solve a particular type of **problem**.

Types of Problems

- STRUCTURING: transform input to satisfy Y (SORT)
- CONSTRUCTION: build X to satisfy property Y (MST)
- OPTIMIZATION: find best X satisfying property Y (TSP)
- DECISION: does the input satisfy property Y (SAT)
- APPROXIMATION: find X that almost satisfies property P and has bounded error (TSP)
- RANDOMIZED: make random choices (QuickSort)
- PARALLEL ALGORITHMS (Factoring)
- ON-LINE ALGORITHMS (Job Scheduling)

Pseudocode

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

Example: Find the maximum element of an array

arrayMax(A, n)

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

Pseudocode Details

- Control flow
  - **if**…[**else**…]
  - **while**…
  - **repeat**…**until** …
  - **for**…**to** and **for**…**downto**
  - Indentation replaces braces
- Method declaration

  method (***arg*** [, ***arg***…])
- Method call (pass by value)

  method (***arg*** [, ***arg***…])
- Return value

  return ***expression***
- Java expressions
  - Also: i = j = k
  - Booleans "short circuit"
- NOTE:
  - Will use 0-based indexing, BUT
  - CLRS uses 1-based indexing!
- Usual OOP notation
  - x.f is the attribute f of object x

Sorting

- Pervasive problem
  - Data processing
  - Efficient search
  - Operations research (e.g. shortest jobs first)
  - Event-driven simulation (e.g. what happens first?)
  - Sub-routine for other algorithms (e.g. Kruskal's MST)
- Informally
  - Bunch of items
  - Each has a "key" that allows "<=" comparison
  - Put items in ascending (or descending) order according to key comparisons

Sorting

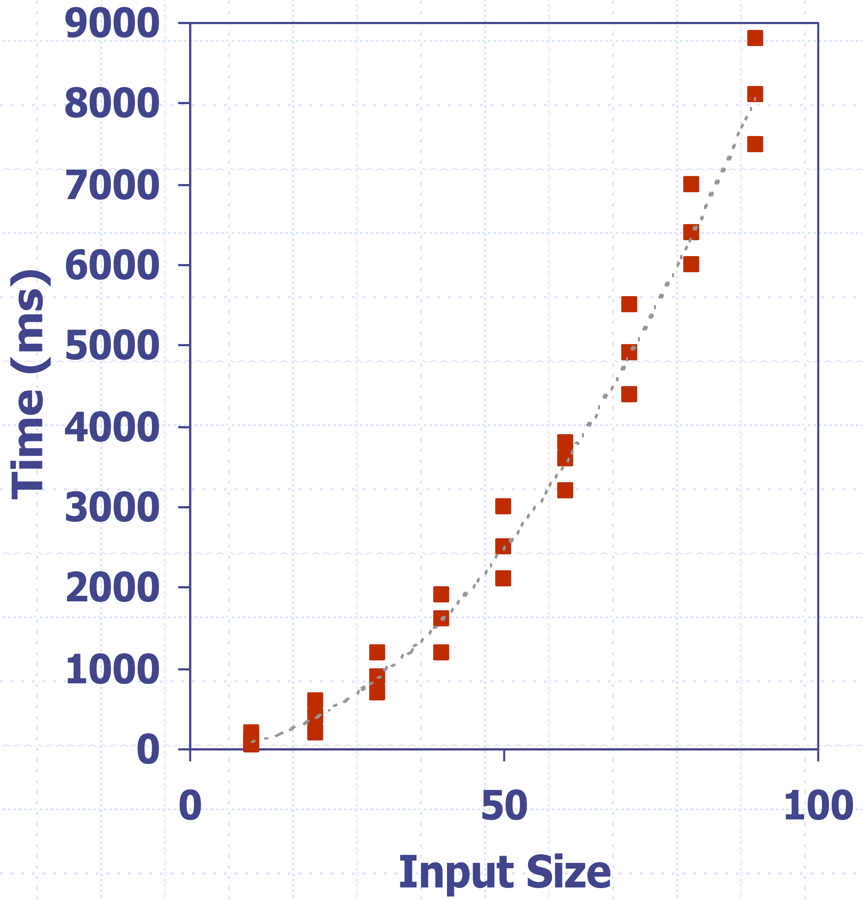
- Bubble Sort
- Selection Sort
- Insertion Sort

What About Efficiency?

- Time
- Space

Experimental Studies

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



Not Okay

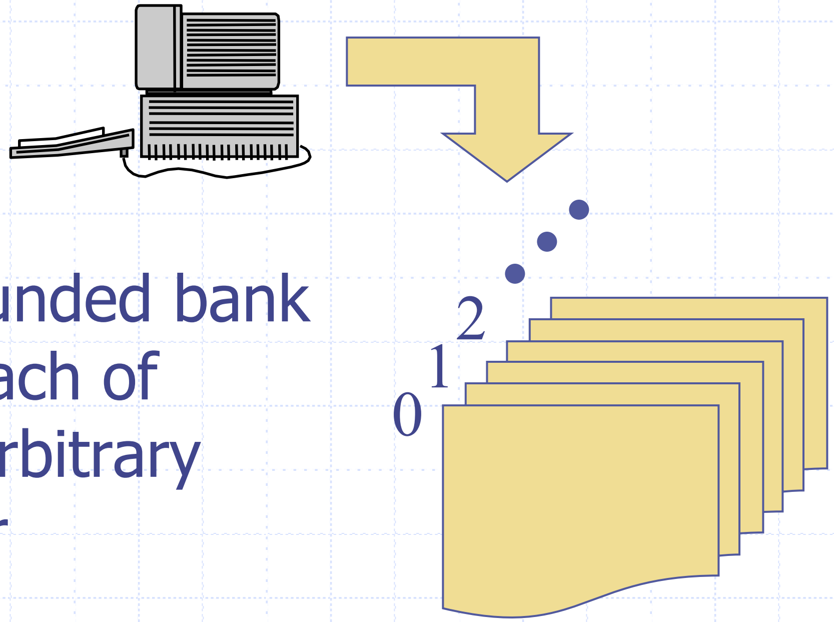
- Implementation can be difficult
- Results depend on:
  - quality of the implementation
  - language used
  - computer used
- Can only run on a limited number of inputs, which may not be representative
- Difficult to test on very large inputs
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

- Use a pseudocode description of the algorithm instead of an implementation
- Equate running time with the number of instructions executed
- Characterize this measure of running time as a function of the input size, *n*.
- Advantages:
  - Takes into account all possible inputs
  - Can analyze and compare algorithms independently of hardware and software

The Random Access Machine (RAM) Model

- A **CPU**



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

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

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent of any programming language
- Exact definition not important
- Each assumed to take a constant amount of time
- Each assumed to take the *same* constant amount of time

- Examples:
  - Evaluating a binary expression,
    e.g. (a + b)
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Really?

- Ignores many things, e.g.
  - Memory hierarchy
  - Processor load
  - “Tricks” like:
    - Pipelining
    - Speculative execution (e.g. branch prediction)
  - Some operations really are a lot more expensive
- But, in practice, it works:
  - It accurately characterizes the running time.
  - It allows us to compare different algorithms.

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

arraySum(A, n)		#operations
$sum = 0$		1
for $i = 0$ **to** $n - 1$		$3n + 2$
$\quad sum = sum + A[i]$		$3n$
return sum		1
	Total	$6n + 4$

Growth Rate of Running Time

- Algorithm ***arraySum*** executes $6n + 4$ primitive operations in the worst case (and the best case).
- Changing the hardware/software environment
  - Affects this by a constant factor, but
  - Does not alter the growth ***rate***
- The fact that the running time grows at the same ***rate*** as the input size is an intrinsic property of algorithm ***arraySum***

Focus on the ***Rate*** of Growth: Big-O Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 > 0$ such that:

  $$f(n) \leq cg(n) \text{ for } n \geq n_0$$

- Example 1: $6n + 4$ is $O(n)$
  - $6n + 4 \leq cn$
  - $6 + 4/n \leq c$
  - Pick $c = 7$ and $n_0 = 4$

- Example 2: n^2 is *not* $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$

More Big-O Examples

- $7n - 2$

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

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

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

- $3n^3 + 20n^2 + 5$

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

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

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

Big-O Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  - Drop lower-order terms
  - Drop constant factors
- Use the "smallest" possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"

Two Relative Growth Rate Rules

- Any positive polynomial function with degree greater than 0 grows faster than any poly-log function:

  $$\lg^a n = O(n^b),\ a > 0,\ b > 0$$

- Any exponential with base greater than 1 grows faster than any polynomial function with degree greater than 0:

  $$n^b = O(c^n),\ b > 0 \text{ and } c > 1$$

Relative Growth Rates

- lg lg n
- lg n
- $\lg^2 n$ also written as $(\lg n)^2$
- $\sqrt{n}$
- n
- n lg n
- n^2
- n^3
- 2^n

Array Sum

- Ignoring constant factors makes things easier!

***arraySum*(*A*, *n*)**	
	#operations
$sum = 0$	1
for $i = 0$ **to** $n - 1$	n
$\quad sum = sum + A[i]$	n
return sum	1
Total	$2n + 2$

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

Relatives of Big-O

- **big-Omega**
  - $f(n)$ is $\Omega(g(n))$ if there is:
    - a constant $c > 0$, and
    - an integer constant $n_0 > 0$

    such that:

    $f(n) \geq c\, g(n)$ for $n \geq n_0$
- **big-Theta**
  - $f(n)$ is $\Theta(g(n))$ if there are:
    - constants $c' > 0$ and $c'' > 0$, and
    - an integer constant $n_0 > 0$

    such that:

    $c'\, g(n) \leq f(n) \leq c''\, g(n)$ for $n \geq n_0$

  - Notice that the two constants, c' and c'', can be different, but n_0 must be the same for both. Just use the max!

Intuition for Asymptotic Notation

big-O

- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)
- usually used to describe worst case

big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)
- can be used to describe best case

big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)
- if best and worst case are the same

Exercises

- Is $n^2/3 + 3n$ $O(n)$?
- Is $n^2/3 + 3n$ $\Omega(n)$?
- Is $n^2/3 + 3n$ $\Theta(n)$?
- Is $n^2/3 + 3n$ $\Theta(n^2)$?

Asymptotic Analysis is Powerful

- An $O(n^{4/3} \log n)$ algorithm to test a conjecture about pyramid numbers ran about 30,000 times faster than an $O(n^2)$ algorithm at $n = 10^9$, finishing in 20 minutes instead of just over a year.

Asymptotic Analysis is Powerful

- In a race between two algorithms to solve the maximum-sum subarray problem:
  - A $\Theta(n^3)$ algorithm was implemented in tuned C code on a 533MHz Alpha 21164 (this was 2000…)
  - A $\Theta(n)$ algorithm was implemented in interpreted Basic on a 2.03 Radio Shack TRS-80 Model II
- The winner?
  - The horribly implemented, but asymptotically faster, algorithm started beating the beautifully implemented algorithm at $n = 5{,}800$.
  - At $n = 10{,}000$, the $\Theta(n^3)$ algorithm took 7 days compared to 32 minutes for the $\Theta(n)$ algorithm.

How Efficient Are Our Sorting Algorithms?

- Bubble Sort
  - worst case?
  - best case?
- Selection Sort
  - worst case?
  - best case?
- Insertion Sort
  - worst case?
  - best case?

Algorithm Design Principle 1

Sometimes we can devise a new
(possibly better) algorithm
by reallocating our computational efforts.

Reallocate Computational Effort: Example 1: Sorting

- Selection Sort
  - Picking next element to place is harder (always)
  - Placing it is easier
- Insertion Sort
  - Picking next element to place is easier
  - Placing it is harder (only sometimes)

Reallocate Computational Effort: Example 2: Searching

- Unsorted list
  - Easy to add items
  - Much harder to find an item
- Sorted list
  - Extra effort to add items (need to keep sorted)
  - Much easier to find an item

Algorithm Design Principle 2

DIVIDE AND CONQUER!

Better Sorting Through Recursion

- Selection Sort → Quick Sort
- Insertion Sort → Merge Sort