CSE 3101: Introduction to the Design and Analysis of Algorithms

Instructor: Suprakash Datta (datta[at]cse.yorku.ca) ext 77875

Lectures: Tues (PSE 321), 7–10 PM

Office hours: Wed 4-6 pm (CSEB 3043), or by appointment.

TA: TBA

Textbook: Cormen, Leiserson, Rivest, Stein.
Introduction to Algorithms (3nd Edition)

CSE 3101: Administrivia

Described in more detail on webpage
http://www.cse.yorku.ca/course/3101

Grading:
Tests: 50% (best 4 out of 5 tests)
Final: 50%
HW: 0%

Notes:
1. All assignments are individual.
2. There MAY be an extra credit test.

Topics: Listed on webpage.

CSE 3101: More administrivia

Plagiarism: Will be dealt with very strictly. Read the detailed policies on the webpage.

Handouts (including solutions): in /cs/course/3101

Grades: will be on *ePost*.

Slides: Will usually be on the web the morning of the class. The slides are for MY convenience and for helping you recollect the material covered. They are not a substitute for, or a comprehensive summary of, the book.

Webpage: All announcements/handouts will be published on the webpage -- check often for updates)

CSE 3101: resources

- We will follow the textbook closely.
- There are more resources than you can possibly read – including books, lecture slides and notes, online texts.
- Jeff Edmonds' (www.cse.yorku.ca/~jeff) textbook has many, many worked examples.
- Andy Mirzaian (www.cse.yorku.ca/~andy) has very good notes and slides for this course
- The downloadable text by Parberry on Problems in Algorithms (http://www.eng.unt.edu/ian/books/free/) is an invaluable resource for testing your understanding

Recommended strategy

- Practice instead of reading.
- Try to get as much as possible from the lectures.
- Try to listen more and write less in class.
- If you need help, get in touch with me early.
- If at all possible, try to come to the class with a fresh mind.
- Keep the big picture in mind. ALWAYS.

The Big Picture

- The design and analysis of algorithms is a FOUNDATIONAL skill -- needed in almost every field in Computer Science and Engineering.
- Programming and algorithm design go hand in hand.
- Coming up with a solution to a problem is not of much use if you cannot argue that the solution is
  - Correct, and
  - Efficient

The Big Picture - 2

- Computation is a “natural” phenomenon – there is a fairly developed science behind it and it studies what can and cannot be done by a given computational model.
- Typical questions are “what cannot be solved by a Turing machine?” and “does there exist an efficient algorithm for this problem?”

Imagine - 0

- You take up a job at a bank. Your group leader defines the problem you need to solve. Your job is to design an algorithm for a financial application that you did not encounter in your classes.
- How do you go about this task?

Designing algorithms – knowledge of paradigms.

Imagine - 1

You want your team to implement your idea – one of them brings you this code and argues that anyone should see that this sorts an array of numbers correctly.

```
for j=2 to length(A)
  do key=A[j]
    i=j-1
    while i>0 and A[i]>key
      do A[i+1]=A[i]
        i--
    A[i+1]:=key
```

How can you be sure?

Correctness proofs – reasoning about algorithms

Imagine - 2

- Two members of your team have designed alternative solutions to the problem you wanted them to solve. Your job is to select the better solution and reward the designer. There are serious consequences for the company as well for the designer.

Efficiency of algorithms – algorithm analysis

Imagine - 3

- Your boss asks you to solve a problem – the best algorithm you can come up with is very slow. He wants to know why you cannot do better.

Intractability : reasoning about problems

Primary Objectives

Previous courses (1020,1030,2011):

1. Figure out an algorithm.
2. Code it, debug, test with “good” inputs.
3. Some idea of running time.
4. Some well known algorithms:
   e.g. QuickSort, Prim(MST), Dijkstra’s algorithm
5. Possibly: some idea of lower bounds.

3101: Problem-solving, Reasoning about ALGORITHMS

1. Design of algorithms
2. Correctness proofs.
   Loop invariants
3. Efficiency analysis.
4. Comparison of algorithms (Better? Best?)
5. Intractability

Machine-independence
Rigorous

Primary objectives - continued

Reasoning about PROBLEMS:

1. Some design paradigms.
   Divide-and-Conquer, Greedy, Dynamic Programming
2. Very simple data structures
   Heaps
3. Lower bounds.
   "Is your algorithm the *best possible*?"
   "No comparison-based sorting algorithm can have running time better than $\theta(n \log n)$".
4. Complexity classes.
   "Are there inherently **hard** problems?"
   P vs NP

Secondary objectives

A new way of thinking -- abstracting out the algorithmic problem(s):

-- Extract the algorithmic problem and ignore the "irrelevant" details

-- Focuses your thinking, more efficient problem solving.

Role of mathematics

1. Needed for correctness proofs
   Pre-condition – post-condition framework; similar ideas used in program verification, Computer-aided design.

2. Needed for performance analysis
   Computation of running time

Specific topics
1. (Very) elementary logic.
2. Elementary calculus.
3. Summation of series.
4. Simple counting techniques.
5. Simple proof techniques: Induction, proof by contradiction
6. Elementary graph theory

Why you should learn algorithms,
Or, Why this is a core course.

1. Fact: Algorithms are always crucial
   Applications: Computational Biology, Genomics
   Data compression
   Indexing and search engines
   Cryptography
   Web servers: placement, load balancing, mirroring
   Optimization (Linear programming, Integer Programming)
2. Fact: Real programmers may not need algorithms…… but architects do!
3. Much more **important** fact: you must be able to REASON about algorithms designed by you and others. E.g., convincingly argue "my algorithm is correct", "my algorithm is fast", "my algorithm is better than the existing one", "my algorithm is the best possible", "our competitor cannot possibly have a fast algorithm for this problem",…

Some examples

1. Sorting a set of numbers (seen before)
2. Finding minimal spanning trees (seen before)
3. Matrix multiplication – compute $A_1A_2A_3A_4\ldots A_n$ using the fewest number of multiplications
   e.g.: $A_1 = 20 \times 30$, $A_2 = 30 \times 60$, $A_3 = 60 \times 40$,
   $(A_1\ A_2)\ A_3$ => 20x 30 x 60 + 20 x 60 x 40 = 84000
   $A_1\ (A_2\ A_3)$ => 20x 30 x 40 + 30 x 60 x 40 = 96000

4. Traveling Salesman Problem: Find the minimum weight cycle in an weighted undirected graph which visits each vertex exactly once and returns to the starting vertex
   Brute force: find intersections of all pairs of sides, include points of each polygon that are inside the other.
   Can we do better?

Reasoning (formally) about algorithms

1. I/O specs: Needed for correctness proofs, performance analysis.
   E.g. for sorting:
   INPUT: A[1..n] - an array of integers
   OUTPUT: a permutation B of A such that
   $B[1] \le B[2] \le \ldots \le B[n]$

2. CORRECTNESS: The algorithm satisfies the output specs for EVERY valid input.

3. ANALYSIS: Compute the running time of the algorithm, the space requirements, number of cache misses, disk accesses, network accesses,….

Pseudocode

- Machine/language independent statements.
- Very simple commands: assignment, equality tests, branch statements, for/while loops, function calls.
- No objects/classes (usually).
- Comments, just like in real programs.
- Should be at a level that can be translated into a program very easily.
- **As precise as programs, without the syntax headaches**
- My notation can vary slightly from the book.

You can use pseudocode, English or a combination.

A simple example

Q1. Find the max of n numbers (stored in array A)
Formal specs:
INPUT: A[1..n] - an array of integers
OUTPUT: an element m of A such that A[j] ≤ m, $1 \le j \le$ length(A)

Find-max (A)

```
1. max ← A[1]
2. for j ← 2 to length(A)
3.    do if (max < A[j])
4.          max ← A[j]
5. return max
```

How many comparisons?

Q2. Can you think of another algorithm? Take a minute….
How many comparisons does it take?

Aside

- How many nodes does a binary tree with n leaves have?

Finding the maximum – contd.

- Proposition: Every full binary tree with n leaves has n-1 internal nodes.
- Corollary: Any "tournament algorithm" to find the maximum uses n-1 comparisons, as long it uses each element exactly once in comparisons.

Correctness

- How can we show that the algorithm works correctly for all possible inputs of all possible sizes?
- Exhaustive testing not feasible.
- Analytical techniques are ~~useful~~ essential here.

Correctness Proof 1

INPUT: A[1..n] - an array of integers
OUTPUT: an element max of A such that A[j] ≤ max, $1 \le j \le$ length(A)

Find-max (A)

```
1. max ← A[1]
2. for j ← 2 to length(A)
3.    do if (max < A[j])
4.          max ← A[j]
5. return max
```

Prove that for any valid Input, the output of Find-max satisfies the output condition.

Proof 1 [by contradiction]: Suppose the algorithm is incorrect. Then for some input A,
(a) max is not an element of A or
(b) (∃j | max < A[j]).
max is initialized to and assigned to elements of A – so (a) is impossible. WHY?
(b) After the j^{th} iteration of the for-loop (lines 2 – 4), max ≥ A[j].
From lines 3,4, max only increases.
Therefore, upon termination, max ≥ A[j], which contradicts (b).

Correctness Proof 1 - comments

- The preceding proof reasons about the whole algorithm
- It is possible to prove correctness by induction as well: this is left as an exercise for you.
- What if the algorithm/program was very big and had many function calls, nested loops, if-then's and other standard features?
- Need a simpler, more "modular" strategy.

Correctness Proof - 2

INPUT: A[1..n] - an array of integers
OUTPUT: an element m of A such that m ≤ A[j],
1 ≤ j ≤ length(A)

```
Find-max (A)
 1. max ← A[1]
 2. for j ← 2 to length(A)
 3.    do if (max < A[j])
 4.          max ← A[j]
 5. return max
```

Prove that for any valid Input, the output of Find-max satisfies the output condition.

Proof 2 [use loop invariants]:

(identify invariant) At the beginning of iteration j of for loop, max contains the maximum of A[1..j-1].

(Proof) Clearly true for j=2. For j > 2, assume that invariant holds for j-1. So at the beginning of iteration j-1 max contains the maximum of A[1..j-2]. Case (a) A[j] is the maximum of A[1..j]. In lines 3,4, max is set to A[j]. Case (b) A[j] is not the maximum of A[1..j], so the maximum of A[1..j] is in A[1..j-1]. By our assumption max already has this value and by lines 3-4 max is unchanged in this iteration.

Correctness Proof – continued

INPUT: A[1..n] - an array of integers
OUTPUT: an element m of A such that m ≤ A[j],
1 ≤ j ≤ length(A)

```
Find-max (A)
 1. max ← A[1]
 2. for j ← 2 to length(A)
 3.    do if (max < A[j])
 4.          max ← A[j]
 5. return max
```

Proof using loop invariants - continued:
We proved that the invariant holds at the beginning of iteration j for each j used by Find-max.

Upon termination, j = length(A)+1. (WHY?)
The invariant holds, and so max contains the maximum of A[1..n]
-- STRUCTURED PROOF TECHNIQUE!
-- VERY SIMILAR TO INDUCTION!
We will see more non-trivial examples later.

Analysis of Algorithms

- Measures of efficiency:
  - –Running time
  - –Space used
  - – others
- Efficiency as a function of input size (NOT value!)
  - –Number of data elements (numbers, points)
  - –Number of bits in an input number

  e.g. Find the factors of a number n,
  Determine if an integer n is prime

Model: What machine do we assume? Intel? Motorola? P4? Atom? GPU?