COMP 200: Elements of Computer Science
Fall 2004
Lecture 35: November 24, 2004

Review and Conclusions

From the Last Lecture

The take-away point from Monday's lecture should be that the set of languages and grammars have a natural hierarchical structure, analogous to the hierarchy of complexities that exist in world of algorithms. In algorithms, we have a collection of problems that can be solved in $\mathbf{O}(n)$ time, then more algorithms that can be solved in low-order polynomial time [for example, sorting has lower and upper bounds of $\mathbf{O}(n \log_2 n)$], then polynomial time (any fixed exponent on the polynomial), then those problems where we do not know of a polynomial algorithm. The latter class of problems includes many where the algorithmic gap (the distance between the lower bound and the best known algorithm — the upper bound) is large.

Similarly, for languages, regular languages (defined by finite automata and regular expressions — which are equivalent) are a strict subset of the context-free languages (defined by context-free grammars) which are a subset of the set of context-sensitive languages (defined by context-sensitve grammars).

Regular languages correspond to languages that can be recognized by a Turing machine that can only move left on its tape. Context-free languages correspond to languages that can be recognized by a Turing machine with a stack rather than a tape. Context-sensitive languages require a tape with two directions of movement.

Regular languages can be recognized in time that is linear in the length of the input string (not the grammar or the expression). Several important subsets of the context-free languages can be recognized in time that grows linearly in the input string, too. (These are called deterministic context-free languages and include the important subsets the LL(1) and LR(1) languages. The details are arcane and covered, at best, in COMP 412). General context-free grammars require $\mathbf{O}(n^3)$ time to recognize (Cocke-Younger algorithm, Kasami's algorithm, Earley's algorithm).

One Last Algorithm – *Monte Carlo Approximation of PI*

Assume that you have a function that can return you a random number between zero and one. (You can always use a function that returns a random number between 0 and n — just divide the result by n.)

We can use the random number generator to approximate the value of PI.

Consider the unit square — from (0,0) to (1,1) through corners at (1,0) and (0,1). If we throw darts at the square — pairs of random numbers between zero and one, inclusive — then the ratio of those darts that fall inside the unit circle to those inside the unit square should be PI/4 to 1.

Throw 100 darts. For each dart, compute the distance between the dart and (0,0) — using the LineLength function from back when we studied structures. (If the dart is (x,y), the distance is just the square root of

```
x*x + y * y
```

which we can compute pretty quickly. If we count the number of darts for which this distance is less than or equal to 1, and multiply by four, we get an approximate value for PI. To improve our approximation, we can look at more darts.

We did not use this as a lab because Dr.Scheme has a lousy random number generator. My approximator converged to about 3.2.

Review — *or, what have we done in 35 lectures …*

Back in Lecture 1, I asked you to define an algorithm. After I got tired of the blank stares, I proposed two definitions from classes that I took as an undergrad: an effective procedure for solving a problem ***or*** a Turing machine that always halts. You were mystified by these definitions; today we can have a deep discussion on them.

Algorithms and how to express them

1. Simple programming in Scheme
   - → Recursion, conditionals, working with structures and lists, set! for changing the value associated with a Scheme name (or object), structural induction over the counting numbers
   - → Hand evaluation of Scheme programs (the rewriting rules)
2. Data structures
   - → Scheme's struct, vectors, arrays, lists, stacks, queues, trees, graphs,
3. Algorithmic strategies
   - → Traversing a graph, divide and conquer, greedy, dynamic planning
4. Complexity of Algorithms
   - → Upper bounds, lower bounds, algorithmic gaps, NP-complete problems
5. Computing via biological analogy
   - → Viruses, worms, Trojan horses, genetic algorithms, artificial intelligence, and the Turing test
6. Models of computation
   - → Turing machines, universal Turing machines, Church-Turing thesis
   - → Finite automata, regular languages, context-free languages, and the limited versions of Turing machines that correspond to these models
7. Cryptography
   - → Substitution codes, one-way ciphers, public-key encryption, RSA algorithm (and the way that it uses complexity theory to defeat attackers — hiding behind factorization)
8. How programs execute

→ From Scheme code to interpreters and compilers

→ How system architects build up a processor from things that (intellectually) resemble a Turing machine

→ From a Turing machine to a modern PC

Algorithms:

- Searching a phonebook (linear search, binary search, and some allusion to encoding names into integers so you can find any name in **O**(1) time.)
- PointAdd, PointSubtract, Euclidean distance (line length)
- Lists: List length, sum a list of numbers, take their average, Max, Min, and MaxMin (divide-and-conquer example from every algorithms textbook ever written)
- Induction on integers: Sum 1 to n, Factorial of n (product from 1 to n)
- Making change with the greedy algorithm
- Insertion sort, bubblesort, mergesort, quicksort
- Minimum cost spanning trees (18)
- Class scheduling (18)
- Manhattan grid problem or Rice to the Galleria by car
- Recognizing a palindrome (or parentheses languages)

COMP 200 Final Project Presentations	
Monday, 1st	
Monday, 2nd	
Monday, 3rd	
Monday 4th	
Wednesday, 1st	Ian / Elizabeth
Wednesday, 2nd	
Wednesday, 3rd	
Wednesday, 4th	
Friday, 1st	Kira / Griffith
Friday, 2nd	
Friday, 3rd	
Friday, 4th	
Other	
Other	
Other	
Other	