COMP 200: Elements of Computer Science
Fall 2004
Lecture 27: November 3, 2004

Exam Review & Intro to Models of Computing

On the Board

2nd Exam: out Friday, due 7 days later (2 hours, closed book, notes, computer)

Review

Since the last exam, we have looked at:

- Algorithms and data structures
  - Bubblesort & using Vectors (*Keys* vector) to hold a list
  - Bounded iteration & loops
  - Stacks & queues as example abstractions
  - Trees (ancestor & descendant trees)
  - General graphs
  - Structural recursion over the counting numbers
    - Sum from 1 to n
    - Product from 1 to n (n!)
  - Algorithmic paradigms
    - Search on trees and graphs
    - Divide and conquer
      - Max, min, mergesort
    - Greedy algorithms
      - Making change for a dollar while using the minimal number of coins
      - Rice to the Galleria — greedy style
      - Classroom scheduling
        - Simple problem is easy in greedy form; Adding constraints makes it quite hard
      - Minimum cost spanning trees (greedy works)
    - Dynamic Planning
      - Tabular form of Rice to Galleria
- Algorithmic complexity, efficiency, and **O** notation
  - Worst case versus average case
  - Notion of an upper and a lower bound

In MergeSort, we saw that a single abstraction may not work well for all the parts of a problem. For example, representing the list as a vector with pairs (low,high) to represent subranges is natural in the divide step, but complicates the combine step. Similarly, a Scheme list representation makes the combine step easy, but complicates the divide

Example where lower bound is still too large
— checking file system on terabyte-sized disk array
— need a better idea (journalled file system)
Computational gap between upper and lower bound
P vs NP
Biological Analogies in algorithms and computing
Viruses, worms, and Trojan horses…

Whew, that's enough.

Models of Computation

You have all grown up with computers. They are familiar objects and all of you have some understanding of how they work. For the next week or so, we will look at some models of computation — highly simplified models — that have the same power as the most advanced computers. That is, any computation that can be done effectively on the most complex computers can also be formulated so that it runs effectively on these simple models.

The fundamental model of computation that is most widely used dates back to 1936, when Alan Turing published a paper that is the cornerstone of the science involved in algorithms. Turing's paper proposed a simple computational model and showed that fundamental questions about what is computable could be posed in terms of this model. His "Turing machine" elucidated a number of critical issues in the theory of computing. It sets a fairly low standard for what is required to achieve "universal" computation — that is, anything that can be effectively computed can be computed on a universal machine. It shows that some important questions cannot be answered — such as, will this procedure ever halt? (More precisely, we cannot devise an algorithm that will examine an arbitrary algorithm or program and correctly report whether or not it halts.)

Turing's model is simple. In fact, it requires a number of principled rules to simplify our thinking.

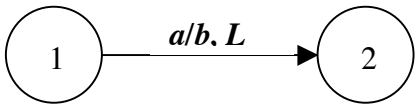
1. Every piece of data can be represented with a finite string of symbols, drawn from a finite alphabet. For example, arbitrary numbers are represented with the digits 0 through 9, a comma, and a decimal point or period. English words are represented with 52 letters (upper and lower case) plus the hyphen.
2. We can treat all the storage in a computer as a single infinite capacity tape, where each cell on the tape can record one symbol from our

alphabet, or a marker to signify that it is empty. (We often use the hash mark, '#', to signify an empty cell.

3. Any effective algorithm can be represented in a finite program with a finite number of commands. (Using looping constructs, we can run a small number of commands for a very long time, but the number of commands in the written form of the program is finite.)
4. Any effective algorithm can be represented with an extremely simple set of commands — the ability to read a symbol from the tape, to replace it with another symbol, and to move right or left — ***combined with*** a finite state controller — a controller that can move from state to state based on the symbols that it sees on the tape. The number of such states is finite.

Notation

We represent a state with a circle (a ***rountangle*** in the book) and a transition between two states as an arrow with some cryptic notation describing when it triggers (or is taken) and what actions it performs. For example:



Specifies that, when the machine is in state 1 and the tape symbol is ***a***, the machine should replace ***a*** with ***b*** and move its view one position ***L***eft on the tape. A state can have many outgoing transitions; each transition must have a unique triggering label. (The letter ***a*** can appear on at most one transition leaving the state. This restriction ensures that the finite state controller is deterministic — it never has to make a choice between two possible actions.)

Turing Machines

This simple machine model suffices to encode any effective algorithm. The book gives a finite state controller for a Turing machine that recognizes palindromes — strings that read the same forward and backward.

The machine assumes that the palindrome is written on the tape and that the viewport on the tape starts at the left end of the string. It uses '#' to denote an empty cell on the tape.

See Figure 9.4, page 224 in the book.