Math/CCS 103 Professor: Padraic Bartlett

Homework 2: Algorithms

Due Friday, week 1 *UCSB 2014*

Homework Problems.

Pick **two** of the following **three** problems to solve!

1. Consider the following algorithm (Stoogesort[1]!) for sorting a list:

   Algorithm. Take as input a list $L = (l_1, \dots l_n)$.

   (a) If your list contains one or two elements, sort it by just looking at the list.
   (b) Otherwise, the list contains ≥ 3 elements. Let $M = \lceil 2n/3 \rceil$.
   (c) Stoogesort the list $(l_1, \dots l_m.)$
   (d) Stoogesort the list $(l_{n-m+1}, \dots l_n)$.
   (e) Stoogesort the list $(l_1, \dots l_m.)$

   Create two lists of 7 or so elements and run this algorithm on those lists. Prove that this algorithm sorts any list. Can you find a polynomial $p(n)$, like n^2 or n^3, such that this algorithm has runtine $O(p(n))$?

2. Spaghetti sort is a sorting algorithm that uses **spaghetti**! We define it here:

   Algorithm. Take as input a list $L = (l_1, \dots l_n)$. Also, a box of dried spaghetti and a hand of appropriate size.

   (a) Given this list of integers, find the largest integer m in our list.
   (b) Define a full piece of spaghetti as m spaghetti-units. Using these units and your box of spaghetti, create a piece of spaghetti of length k for every number k in our list.
   (c) Take all of your newly-formed spaghetti-integers, put them in your hand, take them loosely in your hand and lower them to the table, so that they all stand upright, resting on the table surface. They are now sorted by height!
   (d) One by one, pick out the shortest rod of spaghetti and write it down on our list. This process orders our list.

   What is the runtime of this algorithm? Include your assumptions on how much time it takes you to perform the various steps in this algorithm; i.e. how much effort does each step take? What assumptions are you making about the device that is implementing your sort? Should some steps be regarded as taking longer than others?

[1] Named after the comedy routines of the Three Stooges; specifically, the ones where each stooge hits the other two.

3. Here is a third sorting algorithm, called CycleSort. It works on permutations of $\{1, \ldots n\}$:

   Algorithm. Take as input a list $L = (l_1, \ldots l_n)$, consisting of the numbers $\{1, \ldots n\}$ under some permutation.

   (a) Initialize our algorithm by defining a pair of useful placeholder values: *curr*, the place in the list we're currently sorting, and *temp*, which will hold a value we are moving in our list. Set $curr = 1$, and don't worry about *temp* yet.

   (b) Look at l_{curr}.

   (c) If l_{curr} is equal to *curr* itself, then the list element l_{curr} is in the right place. Proceed according to one of the following two options:

      i. If $curr = n$, our list is sorted and we are done.

      ii. Otherwise, set $curr = curr + 1$ and return to (b).

   (d) Otherwise, $l_{curr} \neq curr$. In this situation, enter the following loop:

      i$\star$. Set $temp = l_{curr}$. Then set $l_{curr} = curr$. Finally, set $curr = temp$.

      ii$\star$. If $l_{curr} = curr$, quit this loop and go to (b).

      iii$\star$. Otherwise, return to i$\star$.

   Write out a few permutations of $\{1, \ldots 9\}$ and run this algorithm on those lists. What is the runtime of this algorithm? How many times does our algorithm change the value of any entry in our list? (This third property is the cool feature of this algorithm, and is why it sometimes sees use in practical settings.)