Algorithms
Computer Science 140 & Mathematics 168
Fall 2003
Instructor: B Thom
Homework 2b
Due Tuesday, Sept. 16

1. **[15 Points] Heapsort!** Implement Heapsort in either Java or C++. Your program should take the length of the array, n, followed by a list of n integers as input. (either from the command line or from standard input). The program should then implement the heapsort algorithm described in class. Turn in a printout of your program and a sample session (a script file if you are on a Unix machine – do `man script` to learn more about scripts) that shows you running the program on some test input. Try to do this using only your lecture notes and not looking at the book. The book uses recursion to build the heap and our discussion did not require recursion.

2. **[10 Points] Heapsort Revisited.** Recall that Heapsort used two functions: buildheap and heapify. The buildheap function took the initial array and turned it into a heap with n elements. We only used buildheap once. Afterwards, we repeatedly swapped the element at the root of the heap with the element at location n, decrement the value of n, and called heapify to restore the heap property for the remaining $n-1$ elements.

   Professor I. Lai of the Massachusetts Institute of Tautologies has proposed the following variant of Heapsort: "Here's what you do," he says excitedly, while gnawing on a foot-long tootsie roll. "You just use buildheap and you don't even need the heapify function! First you call buildheap to get a good heap. Then, you swap the element at the root with the element at location n and decrement the value of n. Now, rather than calling heapify, you just call buildheap all over again on the remaining heap of size $n-1$. You repeat this process until the array is sorted. This is so simple and elegant, I can't believe that nobody ever thought of it before."

   Professor Lai's variant of heapsort works but it's running time isn't so great. How bad is it? Explain.

3. **[20 Points] Curly, Mo, and Larry's Totally Excellent (?) Sorting Algoithm!** Professors Curly, Mo, and Larry of the Pasadena Institute of Technology have proposed the following sorting algorithm: First sort the first two-thirds of the elements in the array. Next sort the last two thirds of the array. Finally, sort the first two thirds again. Notice that this algorithm does not allocate any extra memory; all the sorting is done inside array A. Here's the code:

   Stooge-sort (A,i,j)
   begin
       if $A[i] > A[j]$ **then**

```
        swap A[i] and A[j].
    if i + 1 ≥ j then
        return.
    k = ⌊(j − i + 1)/3⌋.
    Stooge-sort(A, i, j − k).     Comment: Sort first two-thirds.
    Stooge-sort(A, i + k, j).     Comment: Sort last two-thirds.
    Stooge-sort(A, i, j − k).     Comment: Sort first two-thirds again!
end
```

(a) Explain why this algorithm correctly sorts its input. To do so, just explain briefly why sorting the first two-thirds, then the last two-thirds, and then the first two-thirds again will result in a sorted array. You may use pictures to help explain your argument if you like. You're not being asked to provide a formal proof of correctness here, just a brief argument indicating why we should believe that this approach will result in a sorted array.

(b) Find a recurrence relation for the worst-case running time of Stooge-sort.

(c) Next, solve the recurrence relation using the work tree method. **Show all of your work.** Do not use the "Master Theorem" in the book - it's not as general as the work tree method. In your analysis, it will be convenient to choose n to be c^k for some fixed constant c. (For example, we used $c = 2$ when analyzing Mergesort. Here you will want to use a different value of c. This value of c might not even be an integer!)

(d) Explain why the asymptotic running time is still the same even if n is not exactly equal to c^k. (For example, we showed that Mergesort runs in time $\Theta(n \lg n)$ even if n is not a power of 2.)

(e) How does the worst-case running time of Stooge-Sort compare with the worst-case running times of Insertion Sort, Selection Sort, Quicksort, Heapsort, and Mergesort?

4. **[15 Points] Sorting Partially Sorted Data.**

(a) Exercise 8.1-4, page 168. (*Note:* The reason that the book states that it is "not rigorous to simply combine the lower bounds for the individual subsequences" is that this would only show that the lower bound is $\Omega(n \lg k)$ under the assumption that the algorithm just sorted each of those groups. We want to show that **no matter how the algorithm works**, $\Omega(n \lg k)$ is the lower bound - even for algorithms that might do something entirely different from just sorting the individual groups!)

(b) Now briefly describe how such an array (an array of length n with n/k subsequences such that the elements in each subsequence are all smaller than the elements in the next subsequence) can be sorted in time $O(n \log k)$. From the first part of this problem, you can now conclude that your algorithm is asymptotically optimal! Yeehaw!

5. **[15 Points] The Index Problem!** Let A be an array of n distinct integers where A is already **sorted** in ascending order. Our problem is to find an index $i, 1 \leq i \leq n$, such that $A[i] = i$ or determine that no such i exists.

   (a) Find a $\Theta(\log n)$ algorithm for this problem.

   (b) Show that any comparison-based algorithm for this problem must use $\Omega(\log n)$ comparisons. (*Note:* Use an argument very similar to the $\Omega(n \log n)$ lower bound we achieved for comparison-based sorting.)