Assignment 3—Due Monday, November 20

"A Modern Sort of Sort"

Background

You might think that, after so many years of investigation, a problem as well-known as sorting would have been worked to death. Not true! Old problems never die.

Just last year there was a most interesting paper on sorting.[1] It gives a new sorting algorithm which the authors call *Forward Radix Sort* (FRS). I've xeroxed one section from their paper and attached it.

Now Anderson an Nilsson say that FRS is wonderful— simple and fast. On the other hand, they invented the thing and seem a bit brash– who can believe them! In this assignment you'll help us gauge the veracity of these guys (and, by association, the veracity of Swedish people everywhere).

Start off by reading about radix sort in your text, and then carefully studying the algorithm in the attached excerpt of a paper. It's well-described, but it's certainly terse. Work examples as a way of understanding the algorithm. Expect to have to read the description many times. Don't give up; you can't read scientific papers in the same way you'd read a book.

What you must do

Implement FRS. Do a nice, clean implementation. Make it run as fast as you can realistically make it run.

Call your program `fsort`. Give your program the same basic call convention as the UNIX sort, `sort (1)` (but without all the fancy options). Namely, read from a named file and write to standard output: you'll type

```
fsort myfile > mysortedfile
```

to sort the lines of file `myfile`, writing the (sorted) result to standard output. Since you'll be writing the sorted file to standard output, write the descriptive information requested below to standard error.

As mentioned, the file you are sorting is to be regarded as a sequence of *lines*. A ⟨newline⟩ character marks the end of each line. You output the same sequence of lines, but in non-decreasing order. You can be sure something is wrong with your program if the number of bytes in your sorted file differs from the the number of bytes in your input file. (Use `wc` or `ls -l` to see this.)

[1] A. Anderson and S. Nilsson, "A New Efficient Radix Sort." *Proc. 35th Annual IEEE Symp. on the Foundations of Computer Science*, 714–721, 1994.

The lines of your file are to be sorted in the usual dictionary order. Let $x = x_1 \cdots x_k$ and $y = y_1 \ldots y_l$, where each x_i is a byte. Then $x < y$ if there is an i, $0 \le i \le k$, such that $x_j = y_j$ for $1 \le j \le i$ and either $i = k < l$ or $i < k$, $i < l$ and $x_{i+1} < y_{i+1}$. Thus $\varepsilon <$ `a` $<$ `az` $<$ `b` $<$ `bb` $<$ `c` $<$ `xx`, where ε is the empty string (the line which would exist between consecutive $\langle$newlines$\rangle$.

You'll probably want to use $2^{16} = 65,536$ buckets. Experiment with 2^8 buckets and report your findings in the writeup.

Support an argument of `-v`. This selects *verbose* execution. When you run with verbose execution your program will print how many lines got sorted, and in how much time. You may print any further statistics you deem useful.

Timing information can be obtained using `getrusage (3)`. Watch the newsgroup for a pointer to a cleaned-up interface. Other helpful code may be posted; keep watch.

Test your program with small test cases and with `rogaway/public_html/classes/110/big`. Tell me what is line number 5000 of your sorted file. Compare (use `diff`) your sorted file and the one which `sort (1)` generates.

Do your final work on the instructional machines. Your grade will take consideration of how well you've implemented the algorithm. Do your final compile with `g++ -O6`. Run your code on one of the DEC 5000/25 and report the time to sort `big` on this machine.

Please don't make multiple copies of `big` or any other huge files. Clean up any sorted file after you create it. Don't forget that you can write your output to `/tmp` if writing to your own directory gives you disk quota problems.

Include in your writeup any ideas about how to make your program faster. Compare the speed of your `fsort` with the speed of the native `sort`. If your program is faster for large files, find the approximate cross-over point using a prefix of the file `big`.