COMS 3136 Essential Data Structures in C/C++, Lab #1 (100 points total)

Please read this assignment carefully and follow the instructions EXACTLY.

Submission

Please refer to the lab retrieval and submission instruction, which outlines the only way to submit your lab assignments. Please do not email me your code.

If a lab assignment consists of multiple parts, your code for each part must be in a subdirectory (named "part1", "part2", etc.)

Please include README.txt in the top level directory. At a minimum, README.txt should contain the following info:

- your name
- your UNI
- lab assignment number
- description of your solution

The description can be as short as "My code works exactly as specified in the lab." You may also want to include anything else you would like to communicate to the grader such as extra functionalities you implemented or how you tried to fix your non-working code.

Makefile

If a part asks for a program, you must provide a Makefile. The TAs will make no attempt to compile your program other than typing "make".

The sample Makefile in the lecture note is a good starting point for the Makefile you need to write for this lab. If you understand everything about the sample Makefile, you should have no problem writing this lab's Makefile.

Here are some additional documentations on Make:

- Stanford CS Education Library has a nice short tutorial on Makefiles. See Unix Programming Tools, Section 2, available at http://cslibrary.stanford.edu/107/.

- The manual for make is available at http://www.gnu.org/software/make/manual/make.html. Reading the first couple of chapters should be sufficient for basic understanding.

There are a few rules that we will follow when writing makefiles in this class:

- Always provide a "clean" target to remove the intermediate and executable files.

- Compile with gcc rather than cc.

- Use "-Wall" option for all compilations.

- Use "-g" option for compiling and linking.

The sample Makefile follows all these rules.

Valgrind

You will be heavily penalized if your program contains memory errors. Memory errors include (among other things) failure to call free() on the memory you obtained through malloc(), accessing past array bounds, dereferencing uninitialized pointers, etc.

You can use a debugging tool called "valgrind" to check your program:

```
valgrind --leak-check=yes ./your_executable
```

It will tell you if your program has any memory error. See "The Valgrind Quick Start Guide" at http://valgrind.org/docs/manual/quick-start.html for more info.

You must include the output of the valgrind run for EACH PART in your README.txt. In addition, TAs will run valgrind on your program when grading.

Part 1: Selection sort (20 points)

In lab1/part1, I gave you selectionsort_main.c. It reads the size of an array from the user, allocates a char array, randomly fills the array with '0' - '9', and sorts it by calling selection_sort().

Write Makefile, selectionsort.h, and selectionsort.c. Do NOT modify selectionsort_main.c. You may want to modify selectionsort_main.c during your development (to put printf() for debugging, for example), but do not submit any modification to selectionsort_main.c. The easiest way to ensure it is to never git-commit any changes you made to the file.

Implement the iterative selection sort, not the recursive one. You are welcome to take the code from my lecture note (which is basically from the textbook.)

Your executable must be called "selectionsort".

Part 2: Merge sort (30 points)

The lab1/part2 directory contains a partial implementation of merge sort. What's missing is the implementation of merge(). Write it in merge.c. The behavior of the function is documented in merge.h. Again, do NOT modify the provided merge.h and mergesort_main.c.

Your executable must be called "mergesort".

Part 3: Merge sort with recursion trace (30 points)

I put my selectionsort and mergesort executables in /home/jae/cs3136-pub/bin directory. There is also "mergesort-trace" in there. The mergesort-trace program modifies mergesort so that you can see what's going on at each recursion step. Here is the output of mergesort-trace for 8 numbers:

```
$ echo 8 | /home/jae/cs3136-pub/bin/mergesort-trace
36753562
sorting 36753562
  sorting 3675
    sorting 36
      sorting 3
      sorting 6
    merging 3 & 6 into 36
    sorting 75
      sorting 7
      sorting 5
    merging 7 & 5 into 57
  merging 36 & 57 into 3567
  sorting 3562
    sorting 35
      sorting 3
      sorting 5
    merging 3 & 5 into 35
    sorting 62
      sorting 6
      sorting 2
    merging 6 & 2 into 26
  merging 35 & 26 into 2356
merging 3567 & 2356 into 23355667
23355667
```

Note that you can use "echo 8 |" to feed an input to a program instead of waiting for a prompt and typing it in, which will come in handy when you do it over and over.

Your job for part 3 is to modify mergesort_main.c to produce the trace output shown above. You can modify merge.c if you want, but you don't need to. I did all my modification in mergesort_main.c.

You must produce the trace output EXACTLY as shown above, down to the number of spaces in the indentations. In fact, we will use a script to compare your output to that of /home/jae/cs3136-pub/bin/mergesort-trace when grading. This is possible because mergesort_main.c uses a fixed seed for random numbers, so the numbers are always the same in the same machine. Use my mergesort-trace to make sure yours produces the same output for different array sizes.

Part 4: Running time analysis (20 points)

In this part of the lab, we will do some experiments with the selectionsort

and mergesort (don't use mergesort-trace) executables. You should use your own executables, but if you couldn't get it working bug-free before the deadline, you can do this part with my executables.

(a) Running time of selection sort

You can use "/usr/bin/time" program to take various measurements of a running program. You can run it like this:

```
/usr/bin/time ./selectionsort
```

At the end of the program run, the time program will report the time it took to run your selectionsort (along with many other useful info, which you can learn about from "man time" if you're curious.)

You actually need to type "/usr/bin/time" rather than "time". That's because if you type "time", bash shell will run its own built-in command called time rather than the program located at /usr/bin directory.

You will need to give it a large array size to see it run for a meaningful amount of time, in which case you can suppress the output of the large array like this:

```
echo 10000 | /usr/bin/time ./selectionsort > /dev/null
```

Saying ">/dev/null" suppresses the output of the printf() statements.

The time that we are interested in will be the sum of what /usr/bin/time reports as "user" and "system" time. The "user" time will be the amount of time your code has spent doing calculations, and the "system" time will be the amount of time your code has spent inside system calls, like reading files or printing to screen. Since you suppress printf()s, the system time will be negligible.

Measure the running time of your selection sort on a CLIC machine for various input sizes like 10,000 (10k), 100k, 200k, etc. (I measured up to about 400,000, after which I stopped trying because it was getting too boring...)

(b) Running time of merge sort

Do the same experiment with mergesort. You should be able to go to much larger array size. But at some point, you will hit the stack size limit imposed by the bash shell (it was 8 MB for my session), and mergesort will crash. (You will get some obnoxious message like "Command terminated by signal 11".)

You can lift the limit by typing:

```
ulimit -s unlimited
```

which will make your stack grow to infinity and beyond.

But do refrain from running it with more than 10,000,000 (10m) numbers. Beyond that level, the amount of memory that your (and your classmates') mergesort consume might become an issue if many other people are using the

machine at the same time. The CRF might come after you (or even worse, after me!)

If you really want to, you can try 100m (or even 1g) during off-peak hours (i.e., morning), but run "w" command beforehand to make sure the machine is not being heavily used.

(c) Analysis of running time

Does the running time of selection sort fit the expectation that it will be roughly proportional to the input size squared, i.e., O(n^2)?

What about mergesort's O(n * log n)?

From your measurements, can you extrapolate and estimate the running times of the two sorting programs given a billion numbers?

Would you say that these classifications of algorithm running times are just theoretical or they actually have huge impact on real-world performance of computer software?

--

Good luck!