Project 1: MergeSort

Submission deadline: 9pm on 9/20, 2023
(6% of the total grade + 1% extra credit)

This assignment consists of two parts – Part A and Part B

Part A (3% of the total grade): Implement the MergeSort algorithm that sorts an array of arbitrary records. More specifically, you need to implement the body of

```
void msort(void* base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

The arguments to msort() are identical to those of qsort() in the C runtime library (i.e., read the man page of qsort( )). base holds the input elements of size nmemb (unsorted), and after msort(), the elements in base need to be sorted. Like qsort(), this function allows sorting array elements of any type (e.g., built-in or custom type) as long as one provides the compar() function for comparing two elements. Here is some sample code that shows the usage of msort().

```
struct student {int id; char name[100]; int score};
struct student unsorted1[100], unsorted2[100];
…
int compare_id(const void*A, const void *B) {
    const struct student *pA = (const struct student *)A,
                         *pB = (const struct student *)B;
    return (pA->id > pB->id) ? 1 : ((pA->id < pB->id) ? -1: 0);
}
int compare_score(const void*A, const void *B) {
    const struct student *pA =(const struct student *)A,
                          *pB = (const struct student *)B;
    return (pA->score > pB->score) ? 1 : ((pA->score < pB->score) ? -1: 0);
}
…
msort(unsorted1, 100, sizeof(struct student), compare_id);    // sort by the id
msort(unsorted2, 100, sizeof(struct student), compare_score); // sort by the score
…
```

Logistics for Part A:

- Write the code in C – your code should compile with gcc installed on eelab5 or eelab6
- We provide "msort.c", "msort.h", "test.c" "Makefile" in the KLMS page. All you need to do for Part A is to update and submit "msort.c" that includes the complete implementation of msort() described above.
- You can use "test.c" for your own testing & debugging. The current version accepts integers from stdin, sorts the input by msort() and qsort(), and compare the results by printing each element.
  The following commands should build "test", and feed in 100 random integers to "test"

  ```
  $ make
  $ shuf -i 0-10000000 -n 100 | ./test
  ```

- Feel free to modify "test.c" for your own testing and debugging. Note that msort() should work for an input array of an arbitrary custom type.
- We will test your submitted "msort.c" by compiling and linking it to our own test program, so don't assume "test.c" is all that you need to test with.

Part B (3% of the total grade + 1% of extra credit): You are implementing a sorting algorithm for a data set of many thousands of lists of varying lengths, including both a very high proportion of short lists (e.g., with between 2 and 64 items) and a lesser number of much longer lists (e.g., > 10000 items). In addition, around half of the lists are in a random order, and the remaining half are in almost completely sorted order (e.g., with only a few disordered items). You are using MergeSort and have been asked to extend it to improve running time in these two situations: 1) with *short lists* and 2) with *partially sorted* lists. Extend your MergeSort algorithm with, for example, *alternative sorting approaches* and *test conditions* to meet these requirements. Your revised algorithm should not change how MergeSort is invoked – all the optimizations should be under the hood.

Implicit in the first optimization is selecting an appropriate threshold for what is considered a *short list*. You can use any value (e.g., 16 items in length) but you can gain extra credit if you explicitly make a choice and provide evidence for its appropriateness (e.g., show that it provides optimal performance) in your readme file.

Logistics for Part B:

- Write the code in C – your code should compile with gcc installed on eelab5 or eelab6
- We provide "msort_opt.c", "msort_opt.h", "test_opt.c" "Makefile" in the KLMS page. All you need to do for Part B is to update and submit "msort.c" that includes the complete implementation of the optimized msort() described above.
- You can use "test_opt.c" for your own testing & debugging. Its basically a silent version of test code used in Part A. You may use and extend this freely to support testing the performance of your algorithm (e.g., by integrating repetition and measurement features for extra credit). Alternatively, the following bash commands may help:

  ```
  # Generate and sort numLists lists of length numItems. Items are random. Time the process.
  numLists=100000;
  numItems=10;
  startTime=$(date +%s.%6N);              # gdate on osx; date on linux
  for i in {1..$numLists}; do shuf -i 0-10000000 -n $numItems | ./test_opt; done;
  endTime=$(date +%s.%6N);                # gdate on osx; date on linux
  diffTime=$(($endTime-$startTime));
  echo "Sorted $numLists lists of $numItems random items in $diffTime seconds";
  ```

```
# Generate and sort numLists lists of length numItems. Items are a sequence. Time the process.
numLists=100000;
numItems=100;
startTime=$(date +%s.%6N);              # gdate on osx; date on linux
for i in {1..$numLists}; do seq 1 $numItems | ./test_opt; done;
endTime=$(date +%s.%6N);                # gdate on osx; date on linux
diffTime=$(($endTime-$startTime));
echo "Sorted $numLists lists of $numItems ordered items in $diffTime seconds";
```

- We will test your program by compiling and linking with our own test program.

Collaboration policy:

- You can discuss algorithms and implementation strategies with other students, but you should write the code on your own. That is, you should NOT look at the code of anyone else.
- You can use the code in the textbook or in the slides, but you are NOT allowed to look up the Internet for the merge sort algorithm. You can study the usage of qsort() or command line processing, etc. in the Internet, though.

Submission:

- You need to submit 3 files – your ”msort.c” (part A), “msort_opt.c” (part B), and “readme” in a tarred gzipped file with the name as YourID.tar.gz. If your student ID is 20211234, then 20211234.tar.gz should be the file name.
- ‘make’ should build both binaries, “test” and “test_opt”.
- You will get **ZERO** point for each part if “msort.c” or “msort_opt.c” doesn’t compile (due to compiling errors). Make sure that your code compiles before submission.
- Suppress all warnings at compiling by turning on the “-W -Wall” options in the CFLAGS (see Makefile). You will get **some penalty** if gcc produces *any* warnings.
- “readme” (a text file with “readme” as the file name) should contain the followings. (1) Explain how your standard (Part A) msort() works briefly (2) Explain your optimizations for part B. (3) (Optional) provide evidence for your selected threshold (between long and short lists) leading to optimal performance (**extra credit of 1%**). (3) List all collaborators with whom you discussed with in this file.

Grading:

- We will evaluate the accuracy for both Part A and Part B.
- We will evaluate your readme file
- We will evaluate the clarity of your code – coding style, comments, etc.