Assignment 5: Word structure merge sort in ANSI C

Due: Mon, Sep 7.

On `silo.soic.indiana.edu`, change into a directory you create for this assignment, and issue the following command to get the files you need:

```
cp ~rcwhaley/classFiles/ARMF20/p0_mrgSort/* .
```

You will hand in two files implementing least-to-greatest merge sort on structures, and I have given you files with some code to get started for both. You are not allowed to change compiler flags, `word.h` or the API of any file given in either merge sort (and you must actually use these routines to do the work). Note that if you add even more parameters to the recursion over those provided in `mrgSortFast` you can speed up some sorts even more! If you want to play with that, just don't have it be in the required handin file (since you can test any name with provided framework). The routines you must hand in are:

1. **mrgSortSlow.c**: In this file you will implement standard merge sort using the standard API, which does recursion using only a base pointer to a NULL-terminated singly-linked list. All recursion must be in terms of the routine `mrgSortSlow`, and you cannot change its ABI. The purpose of writing this is to make sure you understand merge sort, and perhaps find an optimization or two. You are also required to use the provided routine's (`mergeLists`) API for merging your sorted lists during the recursion. Comparing the timing of this routine to the selection sort routines will show you the importance of proper algorithm selection for resource constrained systems. The trick making my already-sorted time so good may not occur to you, but it is possible using this ABI, and involves doing multiple things when forced to traverse the list, which is not natural if you haven't paid attention to optimization when programming before..
2. **mrgSortFast.c**: In this file you will implement an optimized routine that uses a non-recursive wrapper routine to supply the standard merge-sort API, but does recursion using an API with more parameters so that we can avoid redundant traversals of the list. Getting this working will show you the importance of the high level part of systems optimizations once an algorithm is selected. Low level assembly code systems optimization is then used once proper a proper high-level systems implementation is roughed out. You will probably want to write a succession of more optimized versions. In particular, one required thing is to avoid the unnecessary traverse (needed to split the unsorted array in half) by having the recursion itself provide the first unsorted node to start subsequent recursive calls at.

I have also provided you with two files implementing selection sort:

1. `mrgSortSlow.c`: This file is implemented in a way that many would right it straight out of learning about structures, and it therefore does redundant traversals of the structure (and when you consider a memory load may take 6 orders of magnitude longer than an integer operation, you realize how bad that is on a resource constrained architecture).
2. `mrgSort.c`: This one has been mildly optimized to avoid redundant traversals.

In addition to allowing you to be sure the sanity tester and timer work given valid searches,

scoping the difference between `Slow` and normal versions may help you think about ways to do similar things in merge sort. It is possible to produce a `Fast` version that will sort an already-sorted list in `O(N)` time without doing any extra work not required by the algorithm, but I'm not showing that to you since its very similar to the technique you should figure out in merge sort!

Even more important, scoping the timing difference between these selection sorts and the merge sorts for any large list will allow you to understand the **vast gulf** between $O(N^2)$ and $O\left(Nlog_2(N)\right)$!

I have also provided you with a tester and a timer that can be used with any sort suing the standard API, where the filename and API name are the same. To build a tester/timer use these targets:

1. `make xtstSort mySort=XXX`: Eg. `make xtstSort mySort=selSortSlow`
2. `make xtimSort mySort=XXX`: Eg. `make xtimSort mySort=selSortSlow`

Both the tester and timer take any number of test/timed node counts as arguments. For instance, I can sanity check the provided selection sort with something like:

```
valgrind --leak-check=full --malloc-fill=0x88 --track-origins=yes \
   ./xtstSort 0 1 2 3 4 5 6 7 8 9 10 11 777
```

This will invoke your sort on the given 13 different lengths. My code produces no valgrind errors or warnings.

To give you an idea of what to shoot for, here are timings of all my implementations at size $N = 50,000$ (note you want larger timings for merge sort, but that they take too long using selection sort)

```
silo>make xtimSort mySort=selSortSlow
silo>./xtimSort 50000
   INIT   N=50000 took 1.634336e-03 seconds
   TIMING UNSORTED SORT N=50000 case, word_t
   SORT UNSORTED N=50000 took 3.299506e+01sec, sort/init=20188.7
   SORT   SORTED N=50000 took 2.266664e+01sec, sort/init=13869.0, unsort/sort=1.5

silo>make xtimSort mySort=selSort
silo>./xtimSort 50000
   INIT   N=50000 took 1.507147e-03 seconds
   TIMING UNSORTED SORT N=50000 case, word_t
   SORT UNSORTED N=50000 took 2.162253e+01sec, sort/init=14346.7
   SORT   SORTED N=50000 took 2.170070e+01sec, sort/init=14398.5, unsort/sort=1.0
```

```
silo>make xtimSort mySort=mrgSortSlow ; mv xtimSort xtimeSlwMrg
silo>./xtimeSlwMrg 50000
   INIT   N=50000 took 1.664997e-03 seconds
   TIMING UNSORTED SORT N=50000 case, word_t
   SORT UNSORTED N=50000 took 1.313421e-02sec, sort/init=7.9
   SORT   SORTED N=50000 took 1.067910e-03sec, sort/init=0.6, unsort/sort=12.3

silo>make xtimSort mySort=mrgSortFast ; mv xtimSort xtimeFstMrg
silo>./xtimeFstMrg  50000
   INIT   N=50000 took 1.959106e-03 seconds
   TIMING UNSORTED SORT N=50000 case, word_t
   SORT UNSORTED N=50000 took 9.572926e-03sec, sort/init=4.9
   SORT   SORTED N=50000 took 9.512450e-04sec, sort/init=0.5, unsort/sort=10.1
```

Since these small-case timings jump around a lot, here are some timings for larger problems that can only be done via $O\left(Nlog_2(N)\right)$ algorithms:

```
silo>./xtimeSlwMrg 5000000
   INIT   N=5000000 took 4.637860e-01 seconds
   TIMING UNSORTED SORT N=5000000 case, word_t
   SORT UNSORTED N=5000000 took 4.356770e+00sec, sort/init=9.4
   SORT   SORTED N=5000000 took 3.721241e-01sec, sort/init=0.8, unsort/sort=11.7

silo>./xtimeFstMrg 5000000
   INIT   N=5000000 took 4.637101e-01 seconds
   TIMING UNSORTED SORT N=5000000 case, word_t
   SORT UNSORTED N=5000000 took 2.420972e+00sec, sort/init=5.2
   SORT   SORTED N=5000000 took 3.725983e-01sec, sort/init=0.8, unsort/sort=6.5
```

Please post progress on piazza: when you first get mrgSort working, post your initial timing for N=5,000,000, and then as you improve, post updates. This will make it more fun for everyone, and inspire people to keep looking for tricks (maybe even me!).

NOTE: do not worry at all about timing until you get mrgSort understood and working: it is very hard to optimize a problem you only have a hazy view of!