

Threaded Merge Sort

Cpt S 321 Homework Assignment, WSU

Submission Instructions:

Submit via Git.

Directory name: MergeSort

Git tag for submission: MergeSort-v1.0

Assignment Instructions:

Read each step's instructions carefully before you write *any* code.

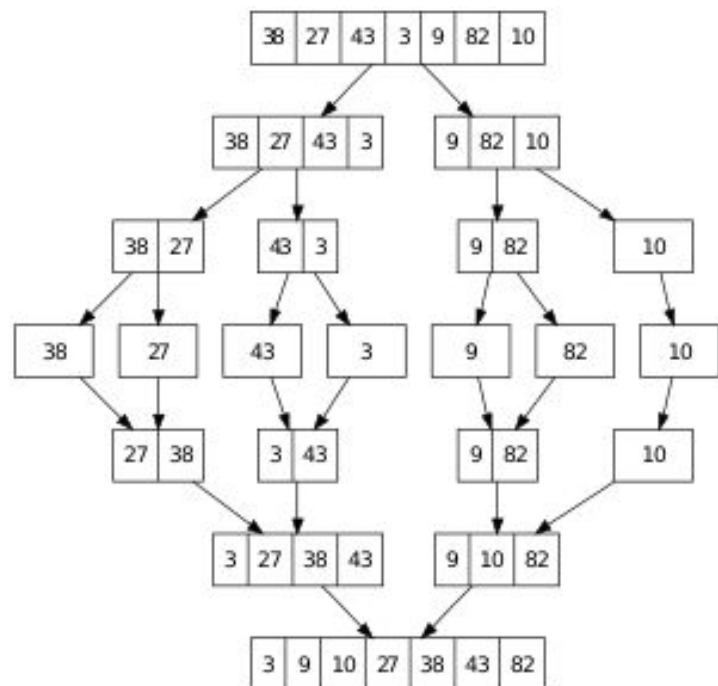
For this assignment we'll be implementing a classic merge sort algorithm in C#, and then doing a threaded version of it. Then with the two versions on hand we'll sort the same list on both of them and look at the difference in time to execute.

Merge Sort

First, if you remember merge sort from your introduction to algorithms class we'll be making that in C#. This is an $O(n \log n)$ time [algorithm](#) that can be done in place if you pass around the array by reference to sort.

The image to the right is my general reminder of how the algorithm works. First, it divides the list it's provided in half and calls merge sort on each half. If it's handed a list with one or zero items, it just returns because the list is already sorted.

Second, it takes the two sublists it was provided and merges them together, which can be done in $O(n)$ time. Then it returns.



Threaded Merge Sort

You'll note that the drawing of the merge sort algorithm has a big divide down the middle as each half of the initial list is split up and then merged back together. This indicates that we can do both sides of the algorithm in parallel with threads. Once both threads have completed their work, then we can merge the two lists and return. The underlying question is whether using threads like this will speed up the algorithm at all.

Issues to remember:

- If you're passing lists around, make sure you know whether your list is going to be modified directly by the function call
 - You can also pass around the original array plus the low and high indices that given function call should be sorting between and all functions just work on the same array instead of copying sub-arrays to the recursive calls.
- Make sure you're not merging until both threads doing merge sort are actually done
 - Start both threads (one for the left subarray, one for the right)
 - Then use the `Thread.Join()` call to wait until they've completed.

Basically, where your MergeSort function does the merge sort calls, our code should thread the MergeSorts and wait for them to return.

Algorithm Comparison

Once we have both a classic merge sort and a threaded merge sort on hand it's time to compare them!

In your main, make a function to run your tests. It should use Random to generate lists of varying lengths (2 copies of each list, one for each sort), then use the DateTime library to measure how many milliseconds each of your algorithms take to sort their copy of the list.

The lists should have varying lengths:

```
[ 8, 64, 256, 1024 ] // Go higher if you like, but I don't think my computer can do it...
```

The random numbers you generate should be from 0 to `Int32.MaxValue` so that there's not too many collisions and that data is nice and random.

Note: You'll have plenty of threads going here, but shouldn't hit any default limits in the CLR given these numbers.

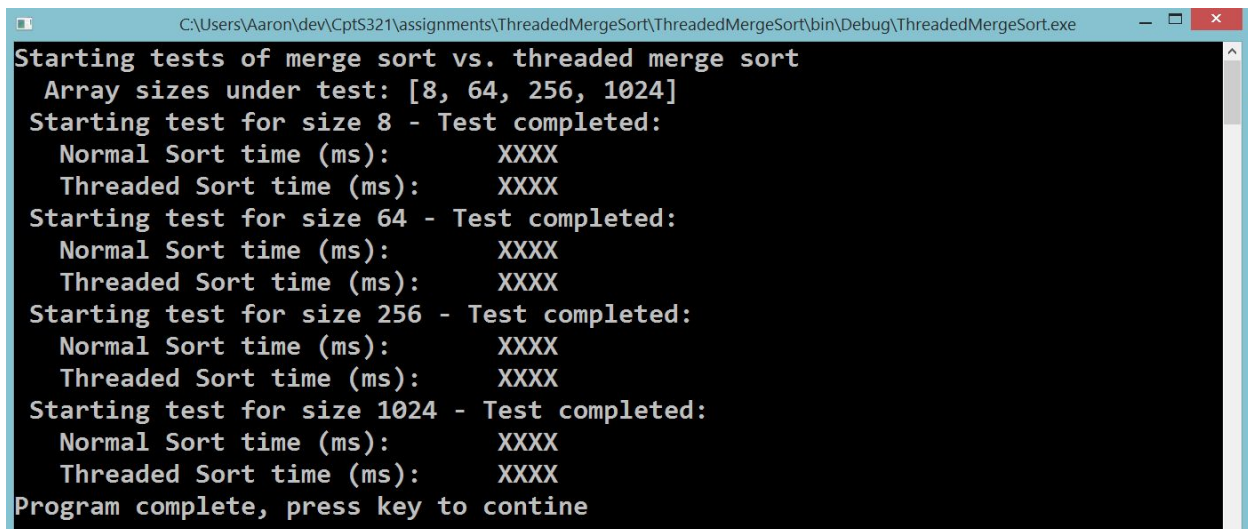
Take in the UNIX time when you call the function to start sorting, then again when it ends. Output the size of the list, the time for the normal merge sort, and the time for the threaded one.

```
long milliseconds = DateTimeOffset.Now.ToUnixTimeMilliseconds();
```

Output:

Run your normal merge sort function (non-threaded) on the lists, and snag the UNIX time before and after execution. Then do it on the other copy of that list with your threaded merge sort algorithm. Again, get the start and end times of the execution. The difference between the timestamps is a rough estimate of how long it took to run the sort function.

Do this for each of the list sizes and output them all to the terminal. Below is the output from my implementation of this, though the timing values are hidden. Your numbers will be based on the efficiency of your implementation.

A screenshot of a Windows terminal window with a black background and white text. The window title bar shows the file path: C:\Users\Aaron\dev\CptS321\assignments\ThreadedMergeSort\ThreadedMergeSort\bin\Debug\ThreadedMergeSort.exe. The terminal output reads: 'Starting tests of merge sort vs. threaded merge sort', 'Array sizes under test: [8, 64, 256, 1024]', 'Starting test for size 8 - Test completed:', 'Normal Sort time (ms): XXXX', 'Threaded Sort time (ms): XXXX', 'Starting test for size 64 - Test completed:', 'Normal Sort time (ms): XXXX', 'Threaded Sort time (ms): XXXX', 'Starting test for size 256 - Test completed:', 'Normal Sort time (ms): XXXX', 'Threaded Sort time (ms): XXXX', 'Starting test for size 1024 - Test completed:', 'Normal Sort time (ms): XXXX', 'Threaded Sort time (ms): XXXX', and 'Program complete, press key to continue'.

```
C:\Users\Aaron\dev\CptS321\assignments\ThreadedMergeSort\ThreadedMergeSort\bin\Debug\ThreadedMergeSort.exe
Starting tests of merge sort vs. threaded merge sort
Array sizes under test: [8, 64, 256, 1024]
Starting test for size 8 - Test completed:
Normal Sort time (ms):      XXXX
Threaded Sort time (ms):   XXXX
Starting test for size 64 - Test completed:
Normal Sort time (ms):      XXXX
Threaded Sort time (ms):   XXXX
Starting test for size 256 - Test completed:
Normal Sort time (ms):      XXXX
Threaded Sort time (ms):   XXXX
Starting test for size 1024 - Test completed:
Normal Sort time (ms):      XXXX
Threaded Sort time (ms):   XXXX
Program complete, press key to continue
```

Requirement Scoring Details:

- Merge sort (4 pts)
- Threaded Merge Sort (7 pts)
- Timing results output (3 pts)
- Code overall quality (1 pt)