Administrative instructions:
1. For any clarification or doubts, the TA Hangjin Liu (hliu25 AT ncsu DOT edu) is in charge of homeworks. She should be your first point of contact on homework-related issues.
2. The homework should be submitted in pairs or triples. Not individually.
3. You should submit electronically through Moodle by midnight the day that the homework is due.
4. Your report should describe any mathematical derivations, responses to questions, and results including any plots. Please justify your answers carefully, including providing interesting insights when relevant.
5. Please attach your code wherever applicable. Please properly comment and indent your code to make it readable. Also include the output in your report wherever applicable.

Improving our merge sort implementation: Recall the merge sort algorithm. (Matlab and Python implementations are available on the course webpage.) Here are different possible ways to accelerate the implementation.

1. **Merging in place** – the program is quite wasteful in terms of moving arrays back and forth between routines, which also requiring plenty of memory allocations and deallocations. Instead, you can implement mergesort by creating an array where the numbers are stored, and then performing all the work using the same array. This implementation style requires to (i) maintain the array as a global variable (you can read about how Matlab / Python handle global variables) and (ii) function calls will have the style mergesort(X,p,q) and merge(X,p,q,r), where X is the array and p, q, and r are indices indicating what ranges in X to sort or what ranges to merge.

2. **Functions in the same file** – it is possible that Matlab (and possibly also the Python implementation) is wasting runtime by moving between the mergesort and merge files, and various overheads are involved with numerous function calls. It could speed things up to put both functions in the same file. Moreover, it might help even more to inline the merge routine into the mergesort routine. (Inlining in this case means copying the code for merge directly into the location in mergesort where it is used; this step will conserve on overheads involved in calling the merge routine, including allocating and deallocating memory.)

3. **Larger basis case** – our implementation performs the “basis case” for sorting problems of length 1; these are sorted within the routine and no further recursive calls are needed. However, sorting problems of length 2 (or a bit larger) are also trivial, and implementing these (or somewhat larger ones) as the basis case would nicely reduce the total number of function calls.

Please see how you can accelerate our implementations. Please explore at least two among options 1-3. You are certainly welcome to improvise and try other accelerations. For example, a direction that could turn into a final project appears below.
To show that your approach indeed works well, please learn about profiling (Module 19) and discuss results that led you to make some of these changes. Show (in a plot or table) your runtimes as a function of the input size for our implementation and yours. To reduce noisiness in measuring the runtime, you may want to run the function multiple times and average over their running times. Finally, please compare the running times over a wide range of problem sizes (a log plot may help illustrate your findings).

**Possible direction for final project:** (Work on this direction if you become enthusiastic about sorting algorithms.) As a nice extension of part 3 above (improving the basis case), you could use insertion sort as the basis case for small sorting problems up to size $k$. To see why this makes sense, consider that running insertion sort on a problem of size $k$ is $\Theta(k^2)$, and because we would have $\Theta(n/k)$ such basis cases to process, they would be solved in $\Theta(n/k \times k^2) = \Theta(nk)$ time. Moreover, merging these $\Theta(n/k)$ subproblems can be done in $\Theta(n \times \log(n/k))$ time. Choosing $k = \log_2(n)$ ensures that the total running time is $\Theta(nk + n \times \log(n/k)) = \Theta(n \times \log(n))$. (In practice, $k$ should be chosen with care. Tinker with it!) Indeed, if you want to pursue this type of thing as your final project, you can read about other sorting algorithms and try to mix and match ideas.

**And another one:** Those of you who become interested in sorting may also want to learn how to compute the median of an array (or list) of values. Recall that you can sort the $n$ numbers in $\Theta(n \times \log(n))$, and extracting the median then requires constant time. However, much of the work performed when implementing a complete sort is redundant in finding the median, for example if you’ve already determined that the first three numbers are all greater than the median, then there is no need to sort them. Fast (linear time!) algorithms for computing the median have been developed. Feel free to read about them and implement accordingly. As above, this topic might be better for your final project.