Objective: Become more proficient at implementing sorting algorithms.

Start by downloading: hw5.zip from http://www.cs.uni.edu/~fienup/cs1520f18/homework/

Part A: Recall that after several iterations of insertion sort’s outer loop, a list might look like:

```
<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unsorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5</td>
<td>6 7 8</td>
</tr>
<tr>
<td>10 20 35 40 45 60</td>
<td>25 50 90</td>
</tr>
</tbody>
</table>
```

In insertion sort the inner-loop takes the "first unsorted item" (25 at index 6 in the above example) and "inserts" it into the sorted part of the list "at the correct spot." After 25 is inserted into the sorted part, the list would look like:

```
<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unsorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5</td>
<td>6 7 8</td>
</tr>
<tr>
<td>10 20 25 35 40 45 60</td>
<td>50 90</td>
</tr>
</tbody>
</table>
```

Code for insertion sort discussed in class is given below:

```python
def insertionSort(myList):
    """Rearranges the items in myList so they are in ascending order""
    for firstUnsortedIndex in range(1, len(myList)):
        itemToInsert = myList[firstUnsortedIndex]
        testIndex = firstUnsortedIndex - 1
        while testIndex >= 0 and myList[testIndex] > itemToInsert:
            myList[testIndex+1] = myList[testIndex]
            testIndex = testIndex - 1
        # Insert the itemToInsert at the correct spot
        myList[testIndex + 1] = itemToInsert
```

The inner-loop combines finding the correct spot to insert with making room to insert by scanning the sorted part from right-to-left and shifting items right one spot to make room until the correct insertion spot is found.

For Part A, I want you to decouple the finding of the correct spot from making room by:

- finding the right spot in the sorted part for the item to be inserted by doing a binary-search of the sorted part,
- make room to insert at the right spot and insert the itemToInsert

Compare the performance of your code with the original insertion sort on "large" random integer arrays. Turn in a timing comparison between your code and the original insertion sort by timing both insertion sorts on random data of size 10,000 and 15,000 items.

Part B: The advanced sorts like merge sort (\(O(n \log_2 n)\) in the worst-case) are faster than the simple sorts (\(O(n^2)\)) when \(n\) is large, but the simple sorts are actually faster when \(n\) is “small”. Use the hw5/compareSorts.py program to experimenting determine the threshold of when the selection sort (a simple sort) is faster on randomly ordered lists than merge sort on your computer. THEN, implement an improved merge sort that utilizes selection sort on small (i.e., less than your threshold) length lists. See the picture on the backside of this handout. In addition to your code, include a report of the timing comparisons between your “improved merge sort” over the original merge sort from lab 8. Time both merge sorts on random data of size 200,000, 400,000, and 800,000.

SUBMISSION
Submit ALL necessary files to run your sorts and your timing “report” for parts A and B as a single zipped file (called hw5.zip) electronically at

https://www.cs.uni.edu/~schafer/submit/which_course.cgi
Diagram illustrating Part B of the assignment:

Unsorted size $n$

Unsorted size $n/2$

$\text{n/4}$

$\text{n/4}$

Unsorted size $n/2$

$\text{n/4}$

$\text{n/4}$

Recursively divide down to the threshold where selection sort is faster

Perform selection sort(s) to sort the smaller array(s)

Recursively merge sorted arrays to finish the sort

$\text{n/4}$

$\text{n/4}$

Sorted size $n/2$

$\text{n/4}$

$\text{n/4}$

Sorted size $n/2$

$\text{n/4}$

$\text{n/4}$

Sorted size $n$