**Objective:** Become more proficient at implementing sorting algorithms.

**Part A:** Recall the insertion sort code from lecture 16 and lab 8:

```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
```

It sorts in ascending order by inserting the first item from the unsorted part into the sorted part on the left end of the list. For example, the inner-loop takes 25 and "inserts" it into the sorted part of the list "at the correct spot."

```
<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</td>
<td>45 50 90 . . .</td>
</tr>
<tr>
<td>60 25 50 90</td>
<td>70</td>
</tr>
</tbody>
</table>
```

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</td>
<td>40 45 60 50 90</td>
</tr>
<tr>
<td>70</td>
<td></td>
</tr>
</tbody>
</table>
```

For Part A, I want you to modify and improve insertion sort (still sorting in ascending order) by:

- build the sorted part on the right end of the list and insert the last unsorted item into it repeatedly,
- speed up the inner-while-loop by:
  - starting the sort by appending `sys.maxsize` to the list
  - simplify the while-loop condition since the `itemToInsert` will always be less than `sys.maxsize`
  - end the sort by popping the `sys.maxsize` value from the list

Include a timing program as in lab8.zip for your improved insertion sort that times the sorting algorithm several time with different initial orderings of 10,000 list items. The initial orderings of items are: descending order, ascending order, random order, and random order again to check for consistence. Write a brief report comparing the timings of your improved insertion sort to the original insertion sort from lab 8 -- include a table of your timings and a brief explanation of the timing results.

**Part B:** In lecture 17 and lab 8 we wrote a simple heap sort that used the BinHeap class. Unfortunately, my lab 8 timings for 400,000 random integers where: 10.5 seconds for heap sort, 3.37 seconds for merge sort, and 2.13 seconds for quick sort. In lab I claimed that heap sort was slower because it used a BinHeap object (e.g., `myHeap=BinHeap()`) and did an extra layer of method calls (e.g., `myHeap.insert(item) and myHeap.delMin()`) which merge sort and quick sort did not have to do. While this claim is true, it is only part of the reason for the “slow” heap sort timing.
For Part B, I want you to modify and improve heap sort (still sorting in ascending order) by:

- eliminating the usage of the BinHeap class by using the list parameter (e.g., myList) to store items rearranged as a heap and “in-line” the code for heap methods right into the heap-sort code, so your heap-sort code calls no methods.
- improve the building of the initial “max-heap” containing all of the list items (called heapifying the list) by:
  - use index 0 to hold the root of the max-heap. This changes the relationship of parent and child indexes. A node at index i has: a left child at i*2+1, a right child at i*2+2, and a parent at (i-1)//2
  - use the concept employed in the “buildHeap” method in the BinHeap class: (1) think of the leaves as mini max-heaps of size 1 and (2) percolate down the non-leaves starting at index (len(myList)//2 - 1) through index 0 into the smaller heaps below to form bigger-and-bigger heaps.
  - while percolating down the non-leaves, don’t repeatedly “swap” down a branch of the heap (e.g., 3 moves per swap), but instead “insert” the non-leaf item into the correct spot in the branch by shifting larger children up the branch (one move per shift) before inserting at the correct spot.
- use the max-heap of all list items in the unsorted part to perform a selection-sort like sort. After heapifying the list in the above step we have:

![Diagram showing heapify process]

Include a timing program as in lab8.zip for your improved heap sort that times the sorting of a randomly generated list of a user-specified size. Write a brief report comparing the timings of your improved heap sort to the original heap sort from lab 8 on 100,000 items, 200,000 items, 400,000 items, and 800,000 items -- include a table of your timings and a brief explanation of the timing results.

**SUBMISSION**

Submit **ALL necessary files** to run your sorts and your timing “reports” 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