After implementing the basic sorting algorithms in the previous lab, we are now ready to implement some *faster* sorting algorithms. These faster algorithms are more commonly used in professional applications. We shouldn't forget about insertion or selection sort though, since they can be faster given a sufficiently small amount of data.

Merge sort

A significant difference between merge sort and the sorts from the previous lab is that it will be using an additional array rather than sorting in place. We can mostly use the code from lectures with a couple of tweaks to fit our sorting application. Merge sort uses a divide and conquer strategy. The basic idea is to split the array into two nearly even pieces (or sub arrays), sort the pieces, and then merge the sorted results together. This fits well with a recursive solution. The base case is when a sub array is of length less than 2. This sub array is already sorted and can be *merged* with another sorted sub array to produce a larger sorted sub array. The recursion continues to unwind, making larger and larger sorted sub arrays until the entire array is sorted.

In pseudocode merge sort can look like this:

```
mergeSort(left, right)
   if left < right
      mid = (left + right) / 2
      mergeSort(left, mid)
      mergeSort(mid + 1, right)
      merge(left, mid + 1, right)
```

And the merge itself can look like this:

```
merge(left, mid, right)
   copy nums[left...right] to temp[left...right]
   i = left, j = left, k = mid
   while i < mid and k <= right
      if temp[i] < temp[k] then nums[j++] = temp[i++]
      else nums[j++] = temp[k++]
   while i < mid
      nums[j++] = temp[i++]
   while j <= right
      nums[j++] = temp[k++]
```

Normally the merge would be done from the original array (`nums`) to the temporary array and then the result copied back at the end. In this case we do the copying first and then merge back into the original array so that we can observe it happening.

Quick sort

Although quick sort has a worst case of $O(n^2)$ the average case, like merge sort, is $O(nlogn)$. It has the advantage of sorting in place without the need for an auxillary array. The basic idea of quick sort is to take an item from the array to be a pivot. Put everything that is smaller than the pivot on the left hand side of the array and everything than is greater than or equal to the pivot on the right hand side. Recursively sort both parts of the array using quick sort.

There are different ways of partitioning the array and of choosing a pivot. A pseudocode implementation of quicksort which uses the first item as the pivot could look like this:

```
quickSort(left, right) {
    if left < right
        p = partition(left, right)
        quickSort(left, p)
        quickSort(p+1, right)

partition(left, right)
    pivot = nums[left]
    hole = left, i = left+1, j = right
    loop forever
        while j > hole && nums[j] >= pivot
           j--
        if j == hole then exit loop
        nums[hole] = nums[j]
        hole = j
        while i < hole && nums[i] < pivot
           i++
        if i == hole then exit loop
        nums[hole] = nums[i]
        hole = i
    nums[hole] = pivot
    return hole
```

Heap sort

This is another in place sorting algorithm that is guaranteed to be $O(nlogn)$ in the worst case. Heap sort sort organises the items of an array into a heap structure, and then repeatedly performs two steps until the array is sorted. The first step is to to swap the largest item with the last item in the array (shrinking the heap by one item), and the second step is to sift the new root of the heap down to restore the heap structure (each node being greater than or equal to its children).

We won't give you any pseudocode for heap sort, but will let you work it out yourself. It's helpful to know that when using an array to represent a heap the children of item i have indexes $2 * i + 1$ and $2 * i + 2$. In order to sort the array using heap sort you first *heapify* the array by calling `siftDown()` on each index from $array.length/2 - 1$ back to the root index (0). After the array has become a heap you then sort it by swapping the root with the last item (shrinking the heap by 1) and *sift down* the new root to the correct place. The `siftDown()` method just swaps an item with the largest child that is bigger than it (if such a child exists) and then calls `siftDown()` on the new index to continue sifting downwards until it finds its correct place.

We haven't provided you with any files for this lab. You can just copy all of your files from the week 9 lab (changing every reference to `week09` into `week11`) and then implement merge sort, quick sort, and heap sort using one of your previous sort implementations as a basis. You should uncomment the appropriate lines in `Sorter.Type` and `SortApp.createSorter()` so that your program creates and uses the new sort implementations that you write. You might also want to slightly increase the time argument given to `Thread.sleep` in the `update()` method so that you can see the faster sorts more clearly.

As usual you can compile your code like this:

```
javac -d .  -Xlint *.java
```

And you can run your code like this:

```
java week11.SortApp 3 < 400-nums
```

The argument 3 selects merge sort as the sorting method (4 and 5 select quick sort and heap sort respectively). You should also check that your sort behaves as expected when using the `-g` argument to display a graphical view of your sort. You can increment the comparison counter inside the condition of a loop by adding something like `(++comparisons > 0)` to the loop conditions immediately before a comparison is done.

Marking

In the first part of the lab (worth 1%) you must implement merge sort and quick sort (both are required to get the 1%).

In the second part of the lab (also worth 1%) you must implement heap sort.

When all of your sorts are working correctly you can use the command `241-check` to make sure they pass all of our tests. If all is well then you can submit them using `241-submit` as usual.