# Programming Assignment 2:
# Out of Sorts

Due date: October 9 at the beginning of lecture via Dropbox.

**Overview:**

In this lab, you will implement two sorting algorithms and compare their performance for a variety of inputs. You may also need to write a program to generate a sequence of integers for testing.

**Program:**

In your chosen programming language, implement the following program:

1. Ask the user to select one of three arrays to sort:
   - `sorted`: containing the numbers 1 through 100,000 in increasing order.
   - `random`: containing a random sequence of 100,000 integers in the range from 1 to 100,000. Duplicate values are fine.
   - `reverse`: containing the numbers 1 through 100,000 in **decreasing** order.

   Call this array `original`.
2. Make a **duplicate** of `original` and then sort it using Insertion Sort (which you must write – do **not** use any built-in sorting algorithms for this assignment). Run the sorted array through your `isOrdered` method from Group Assignment 2 and **validate that the array is actually sorted**. Print the time it took to sort the array.
   (a) Make sure your timing calculation does not include the time to duplicate the `original` array, or to validate the array with `isOrdered`.
   (b) Record the elapsed time using appropriate units.
3. Make a second duplicate of `original` and then sort it using Quicksort (again, you must write this) using the median-of-three pivot selection, using the partitioning algorithm from lecture, and using Insertion Sort to sort any sub-sequence of **10 elements or fewer**. Again, validate the sorted array and output the timing.

**Analysis:**

Complete the following analysis:

1. Knowing how Insertion Sort works, predict which of the three files above will be sorted the fastest by your insertion sort. **Explain your educated prediction**.
2. Predict which of the three files above will be sorted the slowest by your insertion sort, and explain.
3. Predict which of the three files will be sorted the fastest by Quicksort, and explain.
4. Predict whether Insertion Sort or Quicksort will run faster for the `sorted` array, and explain.
5. Predict whether Insertion Sort or Quicksort will run faster for the `random` array, and explain.
6. Run your program on each of the files above, **10 times for each algorithm/file pair**. Record the running times and calculate the mean running time for each. As in Programming Assignment 1, if you are using Java or any other interpreted/virtual machine language (Python, C#, etc. – ask me if unsure), you must run your sort algorithms 5 times prior to recording any data on execution times.
7. Compare your predictions with your results. For any incorrect predictions, **explain the discrepency**, and convince me you now know why you got the results you did.

**Extra Credit:**

For 10 points (out of 100) extra credit, implement a second Quicksort variant:

- In normal Quicksort's `partition`, we swap `i` and `s` whenever `a[i] <= M`. If an array contains entirely identical elements – like a list of 1's, for example – then the pivot will get swapped to the last element of the array and stay there, leaving one side of partition with $n-1$ elements, and the other side of 0 elements.
- We can improve this if we arrange for only *some* of the elements equal to M to end up on its left, and some to end up on its right.
- Our first thought is to randomize which side an element goes on when it equals M, but this would be silly and unpredictable.
- Instead, we'll change partition to swap `i` and `s` when:
  - `a[i] < M` (strictly less than)
  - OR `a[i] == M and i % 2 == 0`
- Since half of all indices in the list are even, we would expect this to – on average – arrange half of the elements equal to M on its left, and half on its right.

So do it. Write an Improved Quicksort that uses this tweak. Then create a fourth array pattern: `identical`, which is the number 1 repeated 100,000 times. Predict whether Classic Quicksort or Improved Quicksort will be faster, and justify your prediction. Then measure **all four arrays** for Improved Quicksort, and the `identical` array for Classic Quicksort, then compare.

**Deliverables:**

Turn in:

1. A printed copy of your program's code. I do not need the code which generates the sequences of numbers.
2. Your thorough analysis :).