Sorting

Sorting is one of the most useful things a computer can do, along with searching. However, it is also one of the most computationally demanding things a computer must do.

We saw two sorts in class, SelectionSort and MergeSort. Here they are again described in psuedocode—don't worry about actual code too much:

SelectionSort:

```
Function SelectionSort(array data, int size) {
	for i = 0 until size {
		for j = i + 1 until size {
			if array[j] < array[i] {
				if j != i {
					swap array[j], array[j]
				}
			}
		}
	}
}
```

MergeSort (first called with 0 going into *low*, and array size into *high*):

```
Function MergeSort(array data, int low, int high) {
	if low < high {
		middle = (low + high) / 2
		MergeSort(array, low, middle)
		MergeSort(array, middle + 1, high)
		MERGE(A, low, middle, high) // the money's all here! Merge 2 arrays
	}
}
```

What are the primary differences between the two?

What are the worst case O( ) of these two?

Note the two different ways the problem of sorting a series is approached. MergeSort is a lot faster because a divide-and-conquer mentality is used.

<u>Sorting, part 2</u>

There is another sort that uses the same mentality that is often even faster, and is much more efficient with memory: QuickSort.

The idea behind QuickSort is basically the same as the idea behind MergeSort. We *divide and conquer*. However, QuickSort doesn't use as much memory as MergeSort and if implemented right is often just as good.

However, the code is complicated!! Let's just look at how QuickSort works:

<u>*Steps in QuickSort (read carefully and DRAW this process!)*</u>

1. Choose a **pivot** (easiest way is to just choose first element)
    - If not the first element, swap **pivot** with first element.

2. Maintain two pointers to elements, one at element directly after pivot—we'll call it **start**—and the other at the end of the data array—we'll call it **end**.

3. As long as the element pointed two by **start** is less than the pivot, move start up the array.

4. If an element is found that is greater than the pivot, stop moving **start**.

5. Look at **end**. Move **end** down the array until an element less than the pivot is found.

6. Swap **start** and **end**. Move **end** to next value.

*7. Repeat steps 3 to 6 until **start** and **end** cross.*

8. Swap **end** and **pivot**.

9. Recursively run again on half the array—go to step 1 on both halves.

Best case for QuickSort: O (   ) ?

Worst case for QuickSort: O (   ) ?

Could we make this better? How?