CSE12F20-Nov4-1

	Insertion	**Selection**	**Merge**	**Quick**
Best case time				
Worst case time				

Key operations	swap(a, j, j - 1)
	(until in the right place)
		swap(a, i, indexOfMin)
		(after finding minimum value)
			l = copy(a, 0, len/2)
			r = copy(a, len/2, len)
			ls = sort(l)
			rs = sort(r)
			merge(ls, rs)
				p = partition(a, l, h)
				sort(a, l, p)
				sort(a, p + 1, h)

```
import java.util.Arrays;
public class Sort {
static void selectionSort(int[] arr) {
    for(int i = 0; i < arr.length; i += 1) {
        int minIndex = i;
        for(int j = i; j < arr.length; j += 1) {
            if(arr[minIndex] > arr[j]) { minIndex = j; }
        }
        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}

static void insertionSort(int[] arr) {
    for(int i = 0; i < arr.length; i += 1) {
        for(int j = i; j > 0; j - = 1) {
            if(arr[j] < arr[j - 1]) {
                int temp = arr[j - 1];
                arr[j - 1] = arr[j];
                arr[j] = temp;
            }
            else { break; } // new! exit inner loop early
        }
    }
}
}
```

```
import java.util.Arrays;
public class SortFaster {

  static int[] combine(int[] p1, int[] p2) {...}

  static int[] mergeSort(int[] arr) {
    int len = arr.length
    if(len <= 1) { return arr; }
    else {
      int[] p1 = Arrays.copyOfRange(arr, 0, len / 2);
      int[] p2= Arrays.copyOfRange(arr, len / 2, len);
      int[] sortedPart1 = mergeSort(p1);
      int[] sortedPart2 = mergeSort(p2);
      int[] sorted = combine(sortedPart1, sortedPart2);
      return sorted;
    }
  }








  static int partition(String[] array, int l, int h) {...}

  static void qsort(String[] array, int low, int high) {
    if(high - low <= 1) { return; }
    int splitAt = partition(array, low, high);
    qsort(array, low, splitAt);
    qsort(array, splitAt + 1, high);
  }

  public static void sort(String[] array) {
    qsort(array, 0, array.length);
  }

}
```