Insertion Sort

```
public void insertionSort(T[] items) {

  for (int i = 1; i < items.length; i++) { // Insert i'th record
    for (int j = i; (j > 0) && (items[j].compareTo(items[j - 1]) < 0); j--) {
      swap(items, j, j - 1);
    }
  }
}
```

- Trace the execution of insertion sort as it sorts the following sequence of records. In this case, the key is the number, and the superscript indicates the record.

  $[7^A, 1^B, 11^C, 7^D, 1^E, 1^F]$

  After first iteration of outer loop:

  $[\mathbf{1^B}, \mathbf{7^A}, 11^C, 7^D, 1^E, 1^F]$

  2nd:

  $[\mathbf{1^B}, \mathbf{7^A}, \mathbf{11^C}, 7^D, 1^E, 1^F]$

  3rd:

  $[\mathbf{1^B}, \mathbf{7^A}, \mathbf{7^D}, \mathbf{11^C}, 1^E, 1^F]$

  4th:

  $[\mathbf{1^B}, \mathbf{1^E}, \mathbf{7^A}, \mathbf{7^D}, \mathbf{11^C}, 1^F]$

  5th:

  $[\mathbf{1^B}, \mathbf{1^E}, \mathbf{1^F}, \mathbf{7^A}, \mathbf{7^D}, \mathbf{11^C},]$

- Was insertion sort stable in this case?

  Yes.

- If so, is insertion sort *always* stable? Justify your answer.

  Yes. The insertion operation will never move a key past a key with an equal value.

Selection Sort

```
public void selectionSort(T[] items) {

  for (int i = 0; i < items.length - 1; i++) { // Select i'th biggest record
    int bigindex = 0; // Current biggest index
    for (int j = 1; j < items.length - i; j++) { // Find the max value
      if (items[j].compareTo(items[bigindex]) > 0) { // Found something bigger
        bigindex = j; // Remember bigger index
      }
    }
    swap(items, bigindex, items.length - i - 1); // Put it into place
  }
}
```

- Trace the execution of selection sort as it sorts the following sequence of records. In this case, the key is the number, and the superscript indicates the record.

  $[7^A, 1^B, 11^C, 7^D, 1^E, 1^F]$

  After first iteration of outer loop:

  $[7^A, 1^B, 1^F, 7^D, 1^E, \mathbf{11^C}]$

  2nd:

  $[1^E, 1^B, 1^F, 7^D, \mathbf{7^A}, \mathbf{11^C}]$

  3rd:

  $[1^E, 1^B, 1^F, \mathbf{7^D}, \mathbf{7^A}, \mathbf{11^C}]$

  4th:

  $[1^F, 1^B, \mathbf{1^E}, \mathbf{7^D}, \mathbf{7^A}, \mathbf{11^C}]$

  5th:

  $[\mathbf{1^B}, \mathbf{1^F}, \mathbf{1^E}, \mathbf{7^D}, \mathbf{7^A}, \mathbf{11^C}]$

- Was selection sort stable in this case?

  No.

- If so, is selection sort *always* stable? Justify your answer.

Comparing Sorts

- On average, insertion sort is faster than selection sort by a constant factor. Why?

For insertion sort the inner for loop will terminate early when the item being inserted reaches the correct position. On average, we only need to insert half-way to the start of the array. The inner loop for selection sort never terminates early.

- Provide an example of a length-three worst case-input for insertion sort. How many comparisons will insertion sort perform on your input? How many swaps?

  $[3, 2, 1]$

  The worst case for insertion sort is reverse order. In this case, it will require 1 comparison to insert 2 *and 2 comparisons to insert* 1*, for a total of 3.*

  $1 + 2 = 3$

  In the worst case, insertion sort performs one swap after every comparisons. Since each swap requires two assignmnets, the total number of assignments will be 6

- For the input you created above, how many comparisons will selection sort perform? How many swaps?

  Comparisons: two comparisons against 3, then one comparison between 1 and 2, for a total of three comparisons.

  $2 + 1 = 3$

  Two swaps for a total of four array assignments.

- Our textbook presents a worst-case analysis of both insertion sort and selection sort through an illustration of stacked boxes. Instead of this approach, analyze the sorts above by developing summations that describe the number of comparisons performed. Solve the summations.

  Insertion sort: As in the 3-element example above, the first element is involved in one comprasion, the second two, until the final element to be inserted is involved in n-1 comparisons:

  $1 + 2 + 3 + ... + n - 1 = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$

  Selection sort: As in the 3-element example above, the first pass through the inner loop involves $n-1$ comparisons to find the largest element overall, the second pass requires $n - 2$ to find the second largest, etc. The full summation is:

  $(n - 1) + (n - 2) + ... + 1 = \sum_{i=1}^{n} n - i = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$

- Here is the implementation of the swap operation used by the two algorithms above:

```java
public void swap(T[] items, int i, int j) {
   T tmp = items[i];
   items[i] = items[j];
   items[j] = tmp;
}
```

Notice that each swap operation requires three assignments. For this reason, most insertion sort implementations *don't* call a swap method. Instead, they follow the following steps for each sorting pass:

```
Place the item being inserted into a temporary variable.

Iterate to the left, shifting each element to the right if it is
greater than the element being inserted. (This requires only one
assignment per item.)

Place the item into its final sorted position.
```

Rewrite the insertion sort algorithm to use this approach.

```java
public void insertionSort(T[] items) {

    for (int i = 1; i < items.length; i++) { // Insert i'th record
        T tmp = items[i];
        int j;
        for (j = i; (j > 0) && (tmp.compareTo(items[j - 1]) < 0); j--) {
            items[j] = items[j - 1];
        }
        items[j] = tmp;
    }
}
```