Sorting Algorithms

The 3 sorting methods discussed here all have wild signatures. For example,

public static <E extends Comparable<? super E>>void BubbleSort(E[] array)

The underlined portion is a *type bound*. This says that the generic type E used as the base type of the array must implement or extend a superclass that implements, the Comparable interface (which says that E has a compareTo(E) method. See the discussion in Weiss of wildcards and type bounds,p. 151-154.

In less generic examples you probably don't need this. If you are writing BubbleSort to sort strings its signature could just be

public static void BubbleSort(String[] array)

BubbleSort

BubbleSort makes repeated passes through the array, interchanging successive elements that are out of order. When no changes are made in a pass the array is sorted.

```java
public static <E extends Comparable<? super E>>void BubbleSort(E[] array ) {
        boolean sorted = false;
        int highest = array.length-1;
        while (!sorted) {
                sorted = true;
                for (int i = 0; i < highest; i++) {
                        if (array[i].compareTo(array[i+1]) > 0) {
                                E buffer = array[i];
                                array[i] = array[i+1];
                                array[i+1] = buffer;
                                sorted = false;
                        }
                }
                highest -= 1;
        }
}
```

Original data

33	**12**	**45**	**17**	**23**	**52**	**24**

12	**33**	**17**	**23**	**45**	**24**	**52**
12	**17**	**23**	**33**	**24**	**45**	**52**
12	**17**	**23**	**24**	**33**	**45**	**52**
12	**17**	**23**	**24**	**33**	**45**	**52**

Each row shows the result of a pass through the previous row, flipping consecutive elements that are out of order.

The first pass through the list does (n-1) comparisons. That pass puts the largest element into its proper location at the last spot in the list, so the next pass does (n-2) comparisons. Altogether we do at most

$$(n-1)+(n-2)+...+1 = n(n-1)/2$$

comparisons. For each comparison we do at most 1 interchange, which takes 3 assignment statements. This means BubbleSort is worst-case $O(n^2)$.

Note that the best case for BubbleSort is when the data is already sorted; only one pass is then needed and the running time is O(n). Of course, if you knew the data was already sorted there wouldn't be a lot of point in calling BubbleSort at all. If the data is nearly sorted (which is not uncommon in real data) BubbleSort might terminate after just a few passes.

SelectionSort

SelectionSort finds the smallest element and puts it at position 0, the smallest remaining element and puts it at position 1, etc.

```java
public static <E extends Comparable<? super E>>void
                                        SelectionSort(E[] array ) {
      for (int i=0; i < array.length-1; i++ ) {
            // find the index of the smallest remaining element
            int small = i;
            for (int j = i+1; j < array.length; j++) {
                  if (array[j].compareTo(array[small]) < 0)
                        small = j;
            }
            // put the smallest remaining element at position i
            E buffer = array[i];
            array[i] = array[small];
            array[small]= buffer;
      }
}
```

Original data

33	12	45	17	23	52	24
12	33	45	17	23	52	24
12	17	45	33	23	52	24
12	17	23	33	45	52	24
12	17	23	24	45	52	33
12	17	23	24	33	52	45
12	17	23	24	33	45	52

The element put in its final location is in blue.

Selection sort does (n-1) passes. The first one does (n-1) comparisons; the second (n-2) comparisons, and so forth. There are a total of

$$(n-1) + (n-2) + (n-3) + ... + 1 = n(n-1)/2$$

comparisons. This is very similar to BubbleSort, only instead of interchanging elements of the array, which takes 3 assignments, here each comparison results in at most one integer assignment. Both are worst-case $O(n^2)$, but in specific examples SelectionSort usually runs somewhat faster.

BubbleSort wins if the data is sorted or nearly sorted. Unlike BubbleSort, SelectionSort doesn't have a quick way out if the data is already sorted; it always does n*(n-1)/2 comparisons.

InsertionSort

InsertionSort maintains a sorted portion of the array (the front) and inserts elements from the unsorted portion into it.

```
public static <E extends Comparable<? super E>>void
                                        InsertionSort(E[] array ) {
      for (int p = 1; p < array.length; p++) {
                  // p is the index of the start of the unsorted part
            E item = array[p];
            int j;
            for (j=p ; j > 0 && item.compareTo(array[j-1]) < 0; j--)
                  array[j] = array[j-1];
            array[j]= item;
      }
}
```

Original data

33	12	45	17	23	52	24
12	33	45	17	23	52	24
12	33	45	17	23	52	24
12	17	33	45	23	52	24
12	17	23	33	45	52	24
12	17	23	33	45	52	24
12	17	23	24	33	45	52

The sorted portion of the array is in blue.

It is easy to see that InsertionSort is no worse than $O(n^2)$ -- the outer loop runs n times, and the inner loop also takes at most n steps -- n steps done n times gives a total of n^2 steps.

The worst case is when the data is reverse-sorted (biggest to smallest); the first pass does 1 comparison, the second 2, and so forth. Altogether this does $1+2+3+...+(n-1) = n(n-1)/2$ comparisons.

But InsertionSort has some saving graces. Note that if the data is already sorted, each pass does only one comparison and no assignment statements, so the algorithm runs in O(n) steps.

We can say even more. Call an *inversion* an instance of two entries of the array being in the wrong order: indices i and j with $i < j$ but $A[i] > A[j]$

The array

33	**12**	**45**	**17**	**23**	**52**	**24**

has 8 inversions: (33 12) (33 17) (33 23) (33 24) (45 17) (45 23) (45 24) and (52 24)

Each time InsertionSort does an assignment it removes one inversion. So if you have an array that is nearly sorted in that it has only a small number of inversions, InsertionSort can sort it quickly.

InsertionSort is a good choice if you have a small amount of data to sort; it tends to be faster than the other simple sorts and is easy to implement.

If you want to sort data the size of the NY phone book, InsertionSort is a terrible choice. There are sorting algorithms that are O(n*log(n)), which is vastly better than $O(n^2)$ when n is large.