1. **Bubble Sort**The bubble sort is notoriously slow, but it's conceptually the simplest of the sorting Algorithms.
2. **Efficiency of the Bubble Sort**O(N2)
3. **Selection Sort**The selection sort improves on the bubble sort by reducing the number of swaps necessary from O(N2) to O(N). Unfortunately, the number of comparisons remains O(N2). However, the selection sort can still offer a significant improvement for large records that must be physically moved around in memory, causing the swap time to be much more important than the comparison time. (Typically this isn't the case in Java, where references are moved around, not entire objects.)
4. **Efficiency of the Selection Sort**O(N2)
5. **Insertion Sort**In most cases the insertion sort is the best of the elementary sorts described in this

chapter. It still executes in O(N2) time, but it's about twice as fast as the bubble sort and

somewhat faster than the selection sort in normal situations. It's also not too complex,

although it's slightly more involved than the bubble and selection sorts. It's often used as the final stage of more sophisticated sorts, such as quicksort.

**Efficiency of the Insertion Sort**How many comparisons and copies does this algorithm require? On the first pass, it

compares a maximum of one item. On the second pass, it's a maximum of two items, and

so on, up to a maximum of N–1 comparisons on the last pass. This is

1 + 2 + 3 + ... + N–1 = N\*(N–1)/2

However, because on each pass an average of only half of the maximum number of

items are actually compared before the insertion point is found, we can divide by 2, which

gives:

N\*(N–1)/4

The number of copies is approximately the same as the number of comparisons.

However, a copy isn't as time-consuming as a swap, so for random data this algorithm

runs twice as fast as the bubble sort and faster than the selection sort.

In any case, like the other sort routines in this chapter, the insertion sort runs in O(N2)

time for random data.

For data that is already sorted or almost sorted, the insertion sort does much better.

When data is in order, the condition in the while loop is never true, so it becomes a

simple statement in the outer loop, which executes N–1 times. In this case the algorithm

runs in O(N) time. If the data is almost sorted, insertion sort runs in almost O(N) time,

which makes it a simple and efficient way to order a file that is only slightly out of order.

However, for data arranged in inverse sorted order, every possible comparison and shift is carried out, so the insertion sort runs no faster than the bubble sort.

1. **Comparing the Simple Sorts**There's probably no point in using the bubble sort unless you don't have your algorithm

book handy. The bubble sort is so simple you can write it from memory. Even so, it's

practical only if the amount of data is small. (For a discussion of what "small" means, see

Chapter 15, "When to Use What.")

The selection sort minimizes the number of swaps, but the number of comparisons is still

high. It might be useful when the amount of data is small and swapping data items is very

time-consuming compared with comparing them.

The insertion sort is the most versatile of the three and is the best bet in most situations,

assuming the amount of data is small or the data is almost sorted. For larger amounts of

data, quicksort is generally considered the fastest approach; we'll examine quicksort in

Chapter 7.

We've compared the sorting algorithms in terms of speed. Another consideration for any

algorithm is how much memory space it needs. All three of the algorithms in this chapter

carry out their sort *in place*, meaning that, beside the initial array, very little extra memory

is required. All the sorts require an extra variable to store an item temporarily while it's

being swapped.

1. **Summary**

The sorting algorithms in this chapter all assume an array as a data storage structure.

• Sorting involves comparing the keys of data items in the array and moving the items

(actually references to the items) around until they're in sorted order.

• All the algorithms in this chapter execute in O(N2) time. Nevertheless, some can be

substantially faster than others.

• An invariant is a condition that remains unchanged while an algorithm runs.

The bubble sort is the least efficient, but the simplest, sort.

• The insertion sort is the most commonly used of the O(N2) sorts described in this

chapter.

• A sort is stable if the order of elements with the same key is retained.

• None of the sorts in- this chapter require more than a single temporary variable in addition to the original array.