# Insertion and Selection Sorts

Insertion and selection sorts are two different ways of sorting lists and collections. To begin with, selection sort use a combination of both searching and sorting. Each time a selection sort passes through a list, its goal is to find the lowest remaining elements and switch it with whatever value is currently in its proper location in the list. A selection sort will always end up passing through an array one less time than the number of items it contains. A selection sort uses nested for loops; the inner loop finds the next smallest value (or largest if sorting in decreasing order) and the outer loop swaps that found value with the value currently its proper location. The Big-Oh of a selection sort is $O(n^2)$.

Here's a table of the contents of an array after every pass of a selection sort in ascending order:

| Array at beginning: | 52 | 72 | 21 | 46 | 27 | 92 |
|---|---|---|---|---|---|---|
| After Pass #1: | 21 | 72 | 52 | 46 | 27 | 92 |
| After Pass #2: | 21 | 27 | 52 | 46 | 72 | 92 |
| After Pass #3: | 21 | 27 | 46 | 52 | 72 | 92 |
| After Pass #4: | 21 | 27 | 46 | 52 | 72 | 92 |
| After Pass #5 (done): | 21 | 27 | 46 | 52 | 72 | 92 |

The insertion sort is often compared to putting playing cards in their proper order. You pick up the closest card to the top of the deck and as you pick it up, you insert it into its correct position in your current hand of organized cards. The insertion sort splits an array into two sub-arrays. The first sub-array (the cards in your hand) is the sorted array and it increases in size as the sort goes on. The second sub-array (the cards that have yet to be picked up) is unsorted, and it contains all the elements yet to be inserted into the first sub-array, and decreases in size as the sort goes on. The insertion sort maintains the two sub-arrays within the same array. When the sort begins, the first element of array becomes the first element of the first sub-array and is considered the "sorted array". With each pass through the sort, the next element in the unsorted second sub-array is placed into the first sub-array in its proper sorted location. The insertion sort can be very efficient when used on smaller arrays. Unfortunately, it becomes slower and less efficient when dealing with large amounts of data. The Big-Oh of an insertion sort is also $O(n^2)$. An example of an ascending-order sort is as follows:

= 1st sub-array  = 2nd sub-array

| Array at beginning: | 53 | 75 | 14 | 73 | 24 | 55 |
|---|---|---|---|---|---|---|
| | 53 | 75 | 14 | 73 | 24 | 55 |
| | 53 | 75 | 14 | 73 | 24 | 55 |
| | 14 | 53 | 75 | 73 | 24 | 55 |
| | 14 | 53 | 73 | 75 | 24 | 55 |
| | 14 | 24 | 53 | 73 | 75 | 55 |
| Array after the only pass: | 14 | 24 | 53 | 55 | 73 | 75 |

Example code:

```
public static void selectionSort(int[] numbers) {
    for(int i = 0; i < numbers.length-1; i++) {
        int minIndex = i;      // Index of smallest remaining value.
        for(int j = i + 1; j < numbers.length; j++) {
            if(numbers[minIndex] > numbers[j]) {
                minIndex = j;  // Remember index of new minimum
            }
        }
        if(minIndex != i) {
            //...  Exchange current element with smallest remaining.
            int temp = numbers[i];
            numbers[i] = numbers[minIndex];
            numbers[minIndex] = temp;
        }
    }
}


void insertionSort(int[] numbers) {
      int j, newValue;
      for(int i = 1; i < numbers.length; i++) {
            newValue = numbers[i]; // Gets next value to be moved
            j = i;
            while(j > 0 && numbers[j - 1] > newValue) {
                  // Finds where the new value belongs in the first
                  sub-array
                  numbers[j] = numbers[j - 1];
                  j--;
            }
            numbers[j] = newValue; // Adds the new value to the first
            sub-array
      }
}
```

Source:

Big Java Textbook