The purpose of sorting is to place the elements in a data structure in a particular order. Usually, we design sorting algorithms to sort elements in ascending order (from smallest to largest). Variations on the order are generally easy to implement once you understand how the sorting algorithm works.

**Insertion Sort:**

Insertion sort works by examining each element one at a time, determining where it belongs among the elements that are already in order and placing it there. Think of re-shelving books in a library as a visualization of insertion sort.

**Algorithm InsertionSort(A):**

*Input: An array A of n comparable elements*

*Output: The array A with elements rearranged in non-decreasing order*

{ loop through each element (the current element), one at a time }

for \( i \leftarrow 1 \) to \( n-1 \) do

\{ make a temporary copy of the current element \}

\( \textit{cur} \leftarrow A[i] \)

\{ move everything bigger than the current element one position to the right \}

\( j \leftarrow i-1 \)

while \( j \geq 0 \) and \( A[j] > \textit{cur} \) do

\( A[j+1] \leftarrow A[j] \)

\( j \leftarrow j-1 \)

{ put the current element in its place }

\( A[j+1] \leftarrow \textit{cur} \)

This is an \textit{in-place} sorting algorithm, since it keeps all elements in the original array as they are rearranged.

The key to the algorithm is the following \textit{invariant}:

As \( i \) goes from 0 to \( n-1 \), the elements in positions \( A[0] \ldots A[i-1] \) are in order, while the elements in positions \( A[i] \ldots A[n-1] \) are not yet ordered.

We can say that \( i \) partitions the array into ordered and unordered segments.

Here’s the insertion sort implemented as a Java method:

```java
public void insertionSort()
{
    int i;
    for (i=1; i<num_values; i++)
    {
        int curr = data[i];
        int j = i-1;
        while ((j >= 0) && (data[j] > curr))
            data[j+1] = data[j--];
        data[j+1] = curr;
    }
}
```

This method accesses the following class fields:

- \textit{data}: the array of elements to be sorted.
- \textit{num_values}: the number of elements in the array to be sorted.
Insertion Sort Example:

We can always start by assuming that the first element is in place. Green indicates the ordered segment.

\[ \begin{array}{cccccc}
45 & 32 & 12 & 19 & 57 & 20 \\
\end{array} \]

**i=1**, place A[1], among A[0] … A[0]

\[ \begin{array}{cccccc}
45 & 32 & 12 & 19 & 57 & 20 \\
32 & 45 & 12 & 19 & 57 & 20 \\
\end{array} \]


\[ \begin{array}{cccccc}
32 & 45 & 12 & 19 & 57 & 20 \\
12 & 32 & 45 & 19 & 57 & 20 \\
\end{array} \]


\[ \begin{array}{cccccc}
12 & 32 & 45 & 19 & 57 & 20 \\
12 & 19 & 32 & 45 & 57 & 20 \\
\end{array} \]


\[ \begin{array}{cccccc}
12 & 19 & 32 & 45 & 57 & 20 \\
12 & 19 & 32 & 45 & 57 & 20 \\
\end{array} \]


\[ \begin{array}{cccccc}
12 & 19 & 32 & 45 & 57 & 20 \\
12 & 19 & 32 & 45 & 57 & 20 \\
\end{array} \]
Selection Sort:

Selection sort works by continually looking for the smallest of elements which have not yet been ordered, then placing that element at the end of the elements that are already in order.

The key to the selection sort is the following invariant:

As partition goes from 0 to n-1, its splits the array into two segments:

- segment 1: A[0] ... A[partition-1] in which all elements are in order, and
- segment 2: A[partition] ... A[n-1] in which elements have not yet been ordered.

In addition, all elements in segment 1 are smaller than or equal to all elements that are in segment 2.

Algorithm SelectionSort(A):

Input: An array A of n comparable elements
Output: The array A with elements rearranged in non-decreasing order

{ partition moves through the array, dividing it into ordered and unordered segments }
for partition ≜ 0 to n-2 do
  { find the smallest element in the unordered segment }
  selection ≜ partition
  for i ≜ partition to n-1 do
      selection ≜ i
  { move next smallest element into position at end of ordered segment and move the element that was in that position out of the way. }

Here’s the selection sort implemented as a Java method:

```java
public void selectionSort()
{
  int partition;
  for (partition=0; partition<num_values-1; partition++)
  {
    int i, selection=partition;
    for (i=partition+1; i<num_values; i++)
      if (data[i] < data[selection]) selection = i;
    swap(partition, selection);
  }
}
```

This method accesses the following class fields:

data: the array of elements to be sorted.
num_values: the number of elements in the array to be sorted.
Selection Sort Example:

Green indicates the ordered segment.

partition=0, find smallest among A[0] … A[5] and move it to A[0]


