Insertion Sort

Insertion Sort – Algorithm that uses one part of the array to hold the sorted values, and the other part of the array to hold values that are not sorted yet.

* The algorithm takes one value at a time from the unsorted part of the array and puts it into the right place in the sorted array, until the array is sorted.

Algorithm

1) Take the first value from the unsorted part of the array.
2) Move the value into the correct place in the sorted part of the array
3) Go through the unsorted part of the array again as many times as there are values.

Example)

Step 1: Start w/ unsorted array:

7, 12, 9, 11, 3

Step 2: We consider the first value as the initial sorted part of the array:

7, 12, 9, 11, 3

Step 3: The next value 12 should now be moved into the correct position in the sorted part of the array, but 12 > 7, so it's already sorted.

7, 12, 9, 11, 3

Step 4: Consider the next value 9. It must be moved to the correct position in the sorted part of the array. So move it in between 7 and 12.

7, 12, 9, 11, 3 ⟶ 7, 9, 12, 11, 3

Step 5: The next value is 11. We move it between 9 and 12 in the sorted part of the array.

7, 9, 12, 11, 3 ⟶ 7, 9, 11, 12, 3

Step 6: The last value to insert into the correct position is 3, so we insert it in front of all other values b/c it is the lowest value.

7, 9, 11, 12, 3 ⟶ 3, 7, 9, 11, 12

Explanation

The first value is considered to be the initial sorted part of the array.

Every value after the first value must be compared to the values in the sorted portion to be inserted into the correct position.

The Insertion Sort Algorithm must run through the array 4 times to sort the array of 5 values b/c we don't need to sort the first value.

Each iteration makes the unsorted part shorter.

Implementation

1) Start w/ an array to sort.
2) An outer loop that picks a value to be sorted. For an array of n values, this outer loop skips the first value, and must run n-1 times.
3) An inner loop that goes through the sorted part of the array, to find where to insert the value. If the value to be sorted is at index i, the sorted part of the array starts at index 0 and ends at index i-1.

Insertion Sort Improvement

The way we are doing it now is finding where to insert the key and then shifting the necessary elements. However, instead of doing these two steps separately, we can do it simultaneously. The psuedocode looks like the following:

```
for (i = 1 ; i < array length ; i++) {
    key = array[i]
    j = i-1;          <-- element to the left of
                          the key (already sorted)

    while ( j ≥ 0 & array[j] > key )   <-- if j = 0, then we
                                           just compare one
                                           element
        array (j+1) = array[ j ]  - shifts to
                                    right into
                                    old key
                                    index
        j--;                               if the key is
                                           smaller, we can shift

    array[j+1] = key  <--
        if j = -1, means insert
        at j = 0 , so it's the smallest element.
```

Time Complexity: $O(n^2)$

Worst Case is when the array is sorted in descending order. This means every new value must "move through" the whole sorted part.

These are the operations done by Insertion Sort for the first elements:

- The 1st element is already in the correct position
- The 2nd element must be compared and moved past the first value
- The 3rd element must be compared and moved past two values
- The 4th element must be compared and moved past three values

So we get the recurrence for n values

$$1+2+3+\ldots+(n-1)$$

We know this equals

$$\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \in O(n^2)$$