Algorithms

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today's Lecture

- Insertion sort

Insertion Sort

- Elementary sorting algorithm.
- Think of the array as being divided into a sorted portion (left side) and an unsorted portion (right side).
- As the algorithm proceeds the sorted portion keeps growing while the unsorted portion shrinks.
- The basic focus of the algorithm is to take an item from the unsorted piece and move that item into its proper place in the sorted piece (insert the item into the sorted piece).
- Similar to selection sort in that they both maintain a portion of the array that is sorted and a portion that is unsorted.

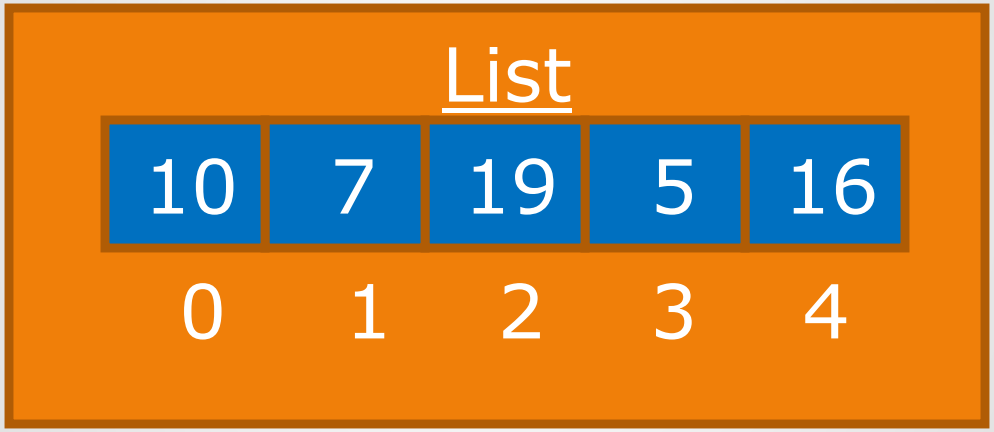
Insertion Sort Overview

Insertion Sort – Comparison Based

- Comparison Based Sort - Works by comparing the items being sorted against each other to determine the order.
- Insertion sort is a comparison based sort.

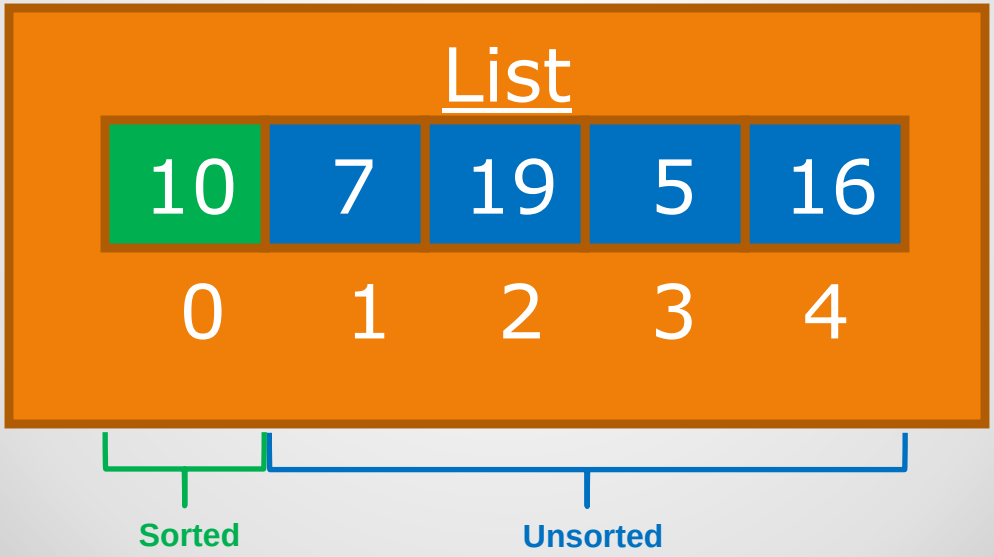
Insertion Sort – Comparison Based Sort

- Sort the following list using insertion sort.



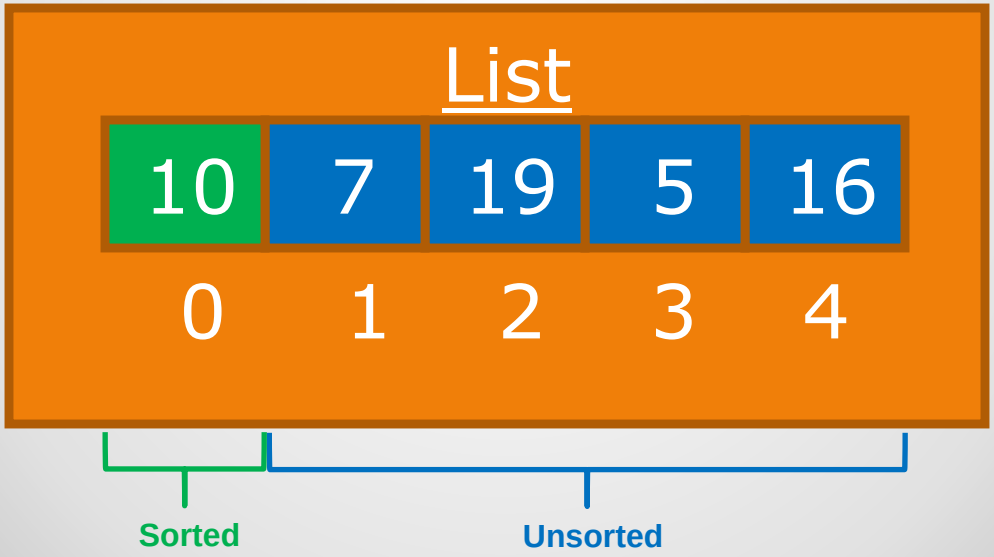
Insertion Sort

- Insertion sort.
- The list is divided into a sorted section and an unsorted section.
- Initially, only the first element is in the sorted section (10 is the only element in sorted below).



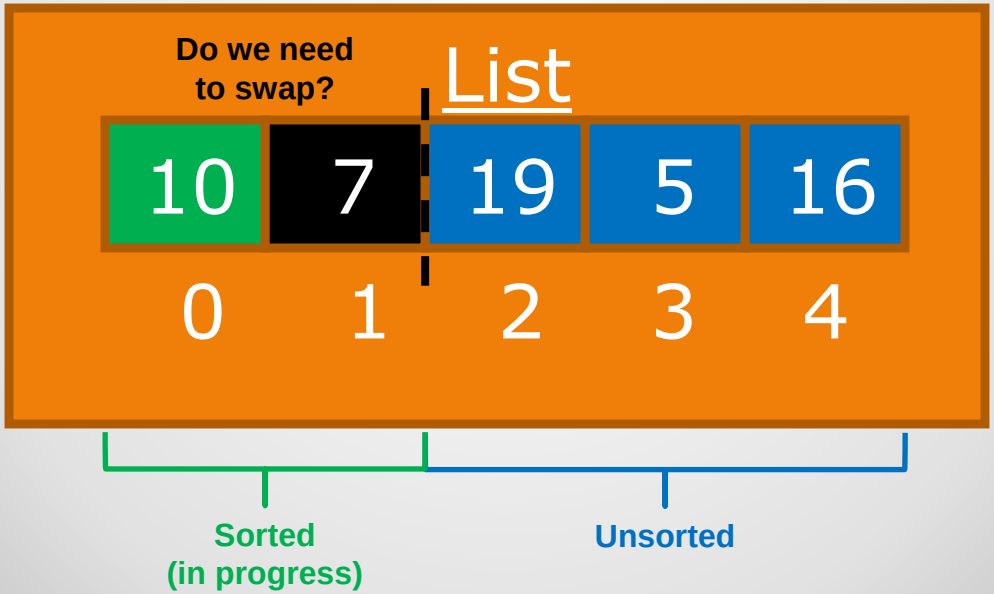
Insertion Sort

- Insertion sort.
- General Procedure.
  - Take an element from the unsorted section.
  - Put that element in its correct place in the sorted section.
  - Keep doing this until there are no more elements in the unsorted section



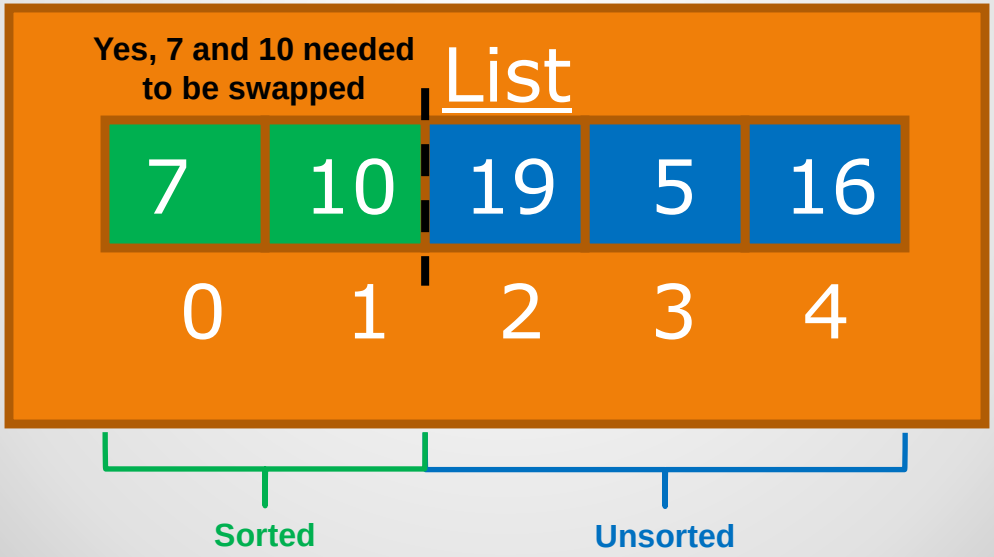
Insertion Sort

- Insertion sort.
- The next element in unsorted is 7.
- Need to find its correct place in the sorted section.
- Start from the end of the sorted section and keep swapping elements until you find the place where 7 belongs.



Insertion Sort

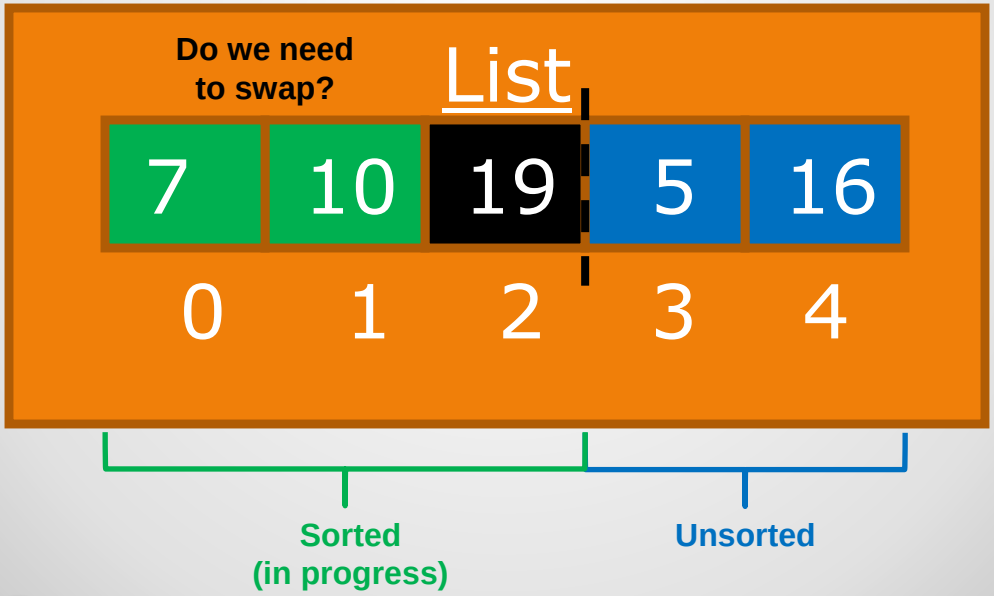
- Insertion sort.
- 7 and 10 needed to be swapped.
- Now the sorted section has two elements.
- We can now move on to the next element in the unsorted section (on next slide).



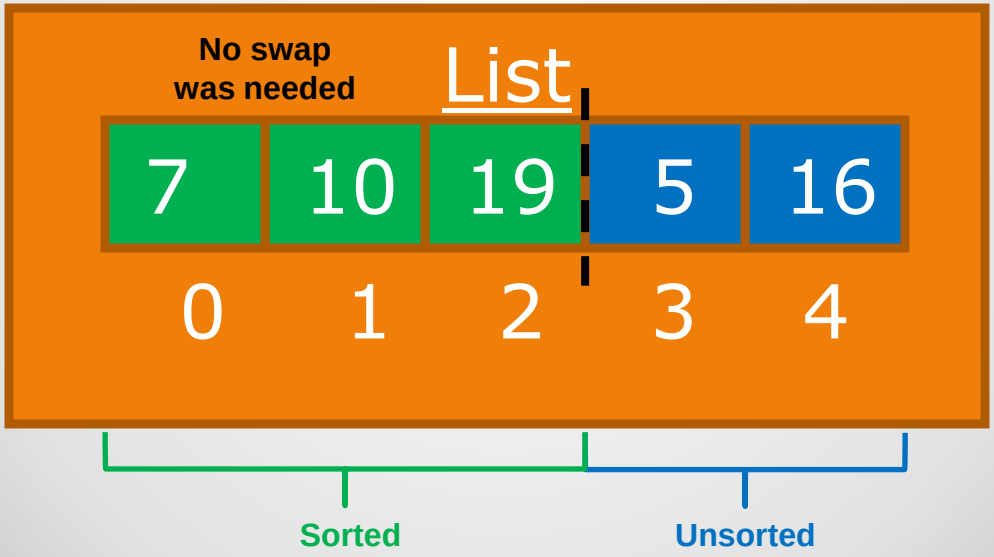
Insertion Sort

Insertion Sort

- Insertion sort.
- The next element in unsorted is 19.
- Find its correct place in unsorted.
- First compare 19 with the element to its left (10).
- Do we need to do any swaps?



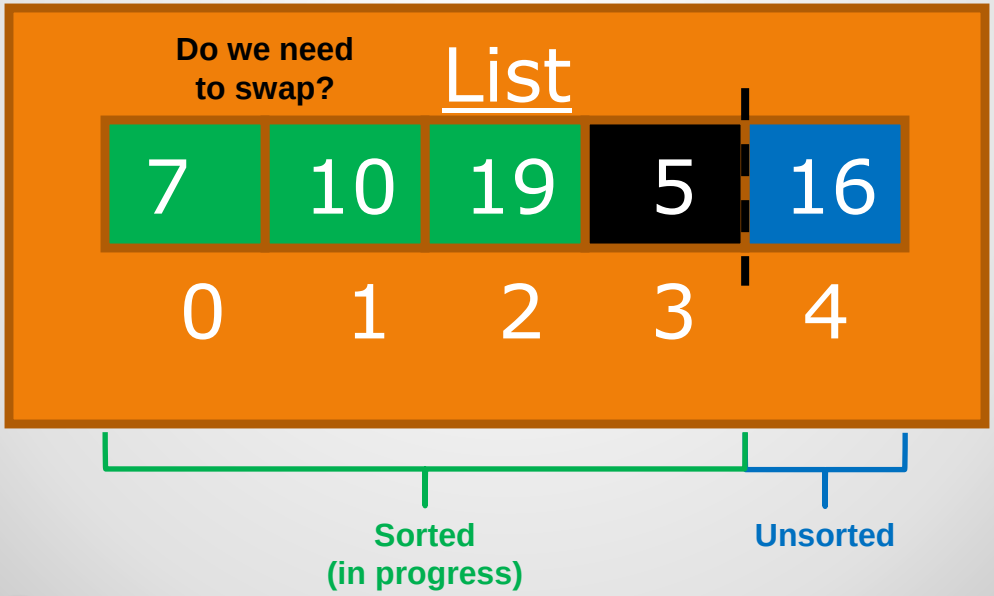
- Insertion sort.
- No swaps are necessary.
- Now there are three elements in the sorted section.
- Go to next slide…



Insertion Sort

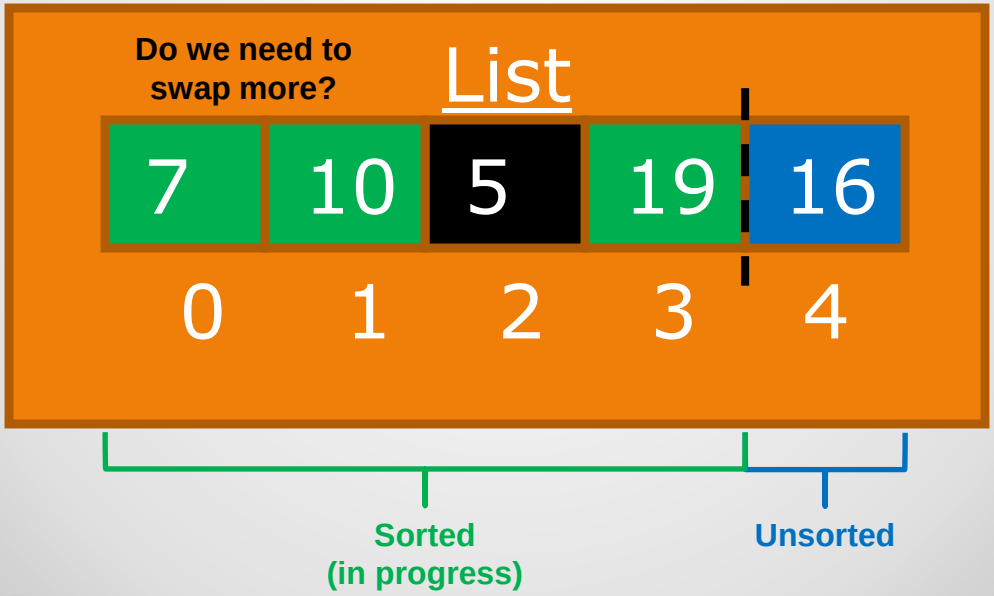
Insertion Sort

- Insertion sort.
- The next element in unsorted is 5.
- First compare 5 with the element to its left (19).
- Do we need to do any swaps?

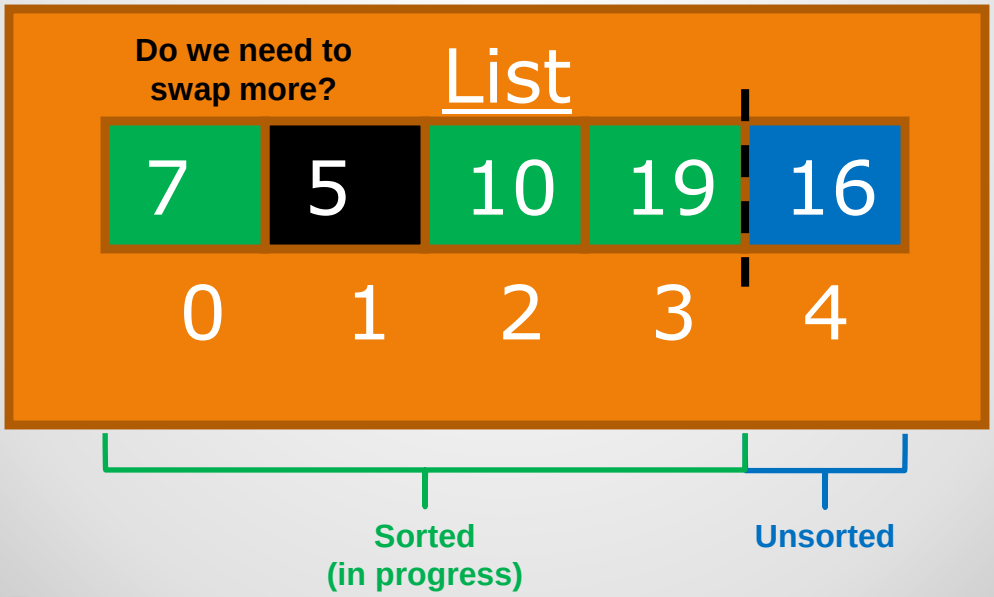


Insertion Sort

- Insertion sort.
- Had to swap 5 and 19.
- Do we need to swap 10 and 5?

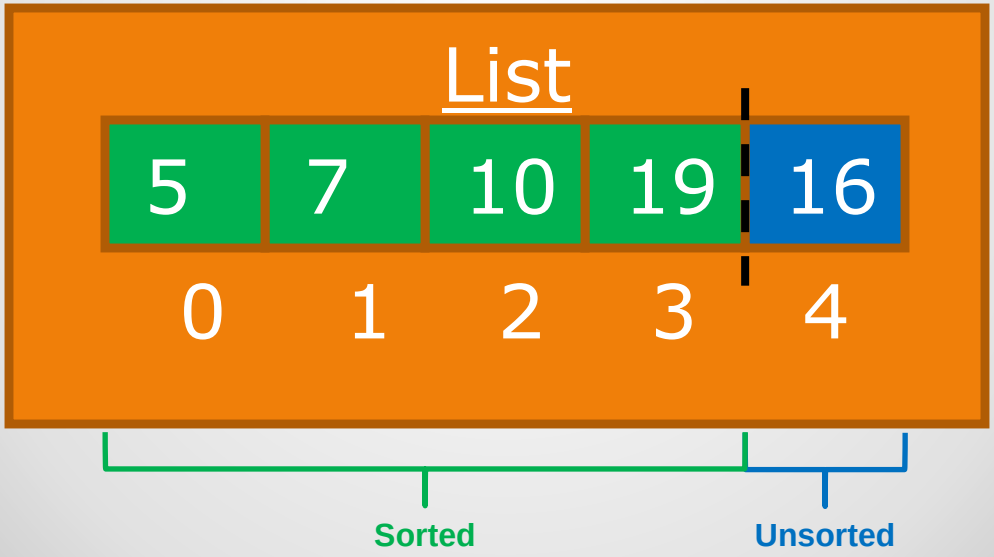


- Insertion sort.
- Had to swap 5 and 10.
- Do we need to swap 7 and 5?



Insertion Sort

- Insertion sort.
- Had to swap 5 and 7.
- We are now done with inserting 5 in the sorted section (it had to be shifted all the way through the sorted section to the beginning).
- Go to next slide…



Insertion Sort

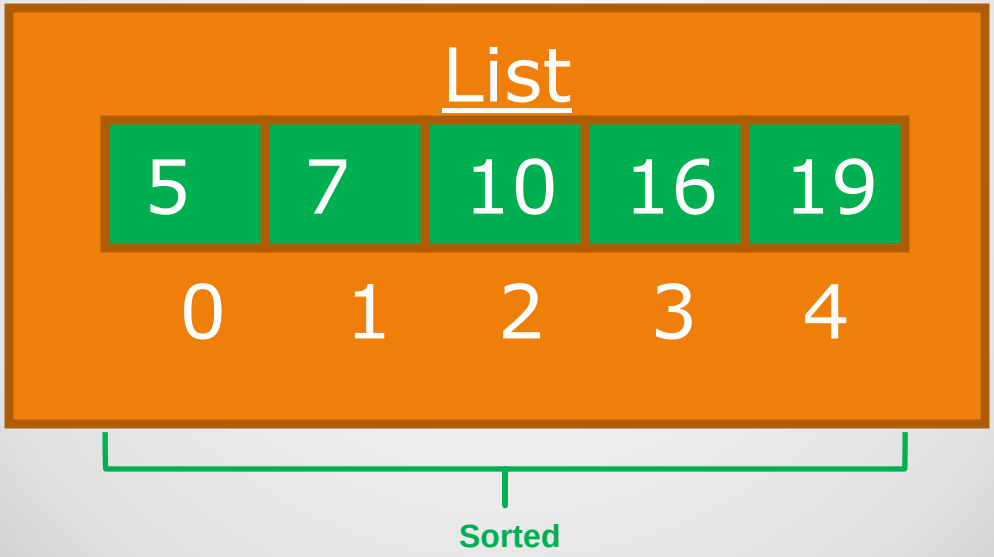
Insertion Sort

- Insertion sort.
- The next element in unsorted is 16.
- First compare 16 with the element to its left (19).
- Do we need to do any swaps?

Insertion Sort

- Insertion sort.
- Had to swap 16 and 19.
- Do we need to swap 10 and 16?

- Insertion sort.
- No more swaps.
- The unsorted section is now empty.
- DONE! The list is sorted.



Insertion Sort

- All passes and swaps shown here…

Start	10	7	19	5	16	
Process 7	7	10	19	5	16	← Swap 7 and 10
Done with 7	7	10	19	5	16	
Process 19	7	10	19	5	16	
Done with 19	7	10	19	5	16	← No swaps done
Process 5	7	10	19	5	16	
	7	10	5	19	16	← Swap 5 and 19
	7	5	10	19	16	← Swap 5 and 10
	5	7	10	19	16	← Swap 5 and 7
Done with 5	5	7	10	19	16	
Process 16	5	7	10	16	19	
	5	7	10	16	19	← Swap 16 and 19
Done	5	7	10	16	19	

Green – Element in sorted section

Blue – Element in unsorted section

Red – Swap was done

All Passes and Swaps

Insertion Sort

```
void insertionSort(int[] list)
   Declare int temp, outer, i

   For outer = 1 to (list.length – 1)
      i = outer

      While (i > 0) && (list[i - 1] > list[i])
               // Swap elements
               temp = list[i]
               list[i] = list[i - 1]
               list[i - 1] = temp

               Decrement i
      endWhile
   endFor
```

Outer loop visits all elements in the array

The inner loop moves the current unsorted element (list[i]) into its place in the sorted portion of the array

These three lines swap adjacent elements

Go to the next element (moving towards beginning of the array)

- Now on to the analysis of insertion sort…

Insertion Sort Analysis

Insertion Sort Analysis

- Runtimes for different cases of input data sets.
- Average (randomized data) is $O(n^2)$ – In practice it will be better than selection sort (inner loop will not always visit all elements like selection sort must do). Does $\frac{1}{4}(n^2 - n)$ comparisons.
- Worst (sorted in reverse order) is $O(n^2)$ – It will always visit all elements in the sorted portion of the list, and it will always do the maximum number of swaps. Does $\frac{1}{2}(n^2 - n)$ comparisons.
- Best (data is already sorted) is $O(n)$ – The reason for this speed up is that the inner loop will never run (no swaps are necessary). It just iterates through the outer loop without doing any swaps. Does (n-1) comparisons.

Insertion Sort – Big O (Worst)

- There are $\frac{n(n-1)}{2}$ comparisons.

Simplify:

- $f(n) = \frac{n(n-1)}{2}$
- $= \frac{n^2}{2} - \frac{n}{2}$ ← Distribute n
- $= ½ (n^2 - n)$ ← Factor out 1/2
- $= n^2 - n$ ← Remove lower order terms
- $= n^2$

- $f(n) \in O(n^2)$

Insertion Sort – Big O (Average)

- The worst case is (from previous slide):
- = ½ (n^2 – n)

- In the worst case it will compare the current element against all elements of the sorted section.
- In the average case it will only do half of those comparisons (only needs to visit half of the items in the sorted section on average).

- Therefore, the average case will be:
- = ½ * ½ (n^2 – n)
- = ¼ (n^2 – n)
- = n^2 – n
- f(n) ∈ O(n^2)

- Runtimes for different cases of input data.

Input Data	$\Omega(g(n))$	$O(g(n))$	$\Theta(g(n))$
Average Case	n^2	n^2	n^2
Best Case (already sorted low to high)	n	n	n
Worst Case (sorted high to low)	n^2	n^2	n^2

- Best case input data speeds up the algorithm.
- Data that is basically sorted will also run very fast (only a few extra swaps are necessary).

Insertion Sort Analysis

- **End of Slides**

End of Slides