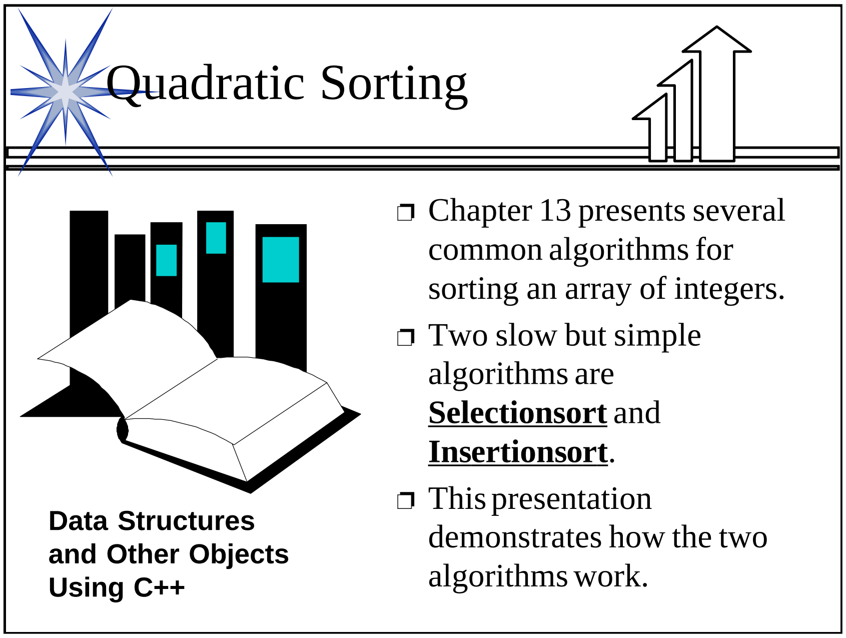


Quadratic Sorting

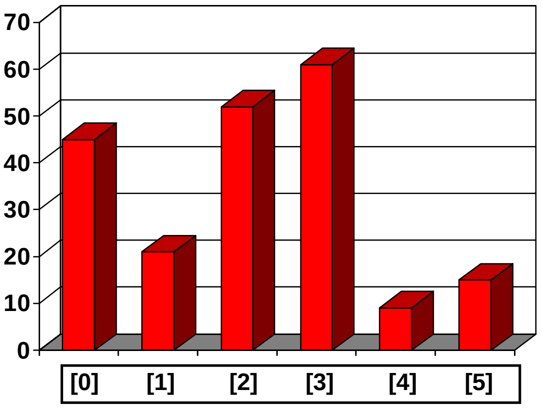
- Chapter 13 presents several common algorithms for sorting an array of integers.
- Two slow but simple algorithms are **Selectionsort** and **Insertionsort**.
- This presentation demonstrates how the two algorithms work.

Data Structures and Other Objects Using C++

The presentation illustrates two quadratic sorting algorithms: Selectionsort and Insertionsort. Before this lecture, students should know about arrays, and should have seen some motivation for sorting (such as binary search of a sorted array).

Sorting an Array of Integers

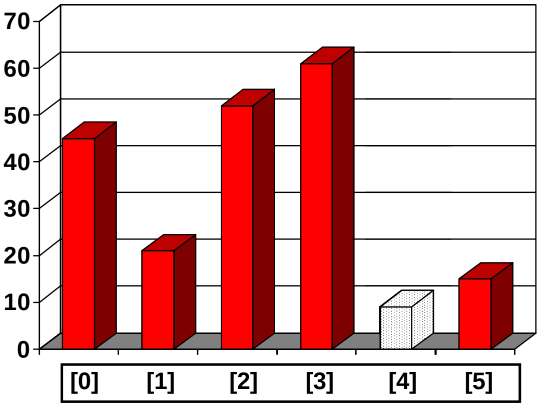
- The picture shows an array of six integers that we want to sort from smallest to largest



The picture shows a graphical representation of an array which we will sort so that the smallest element ends up at the front, and the other elements increase to the largest at the end. The bar graph indicates the values which are in the array before sorting--for example the first element of the array contains the integer 45.

The Selectionsort Algorithm

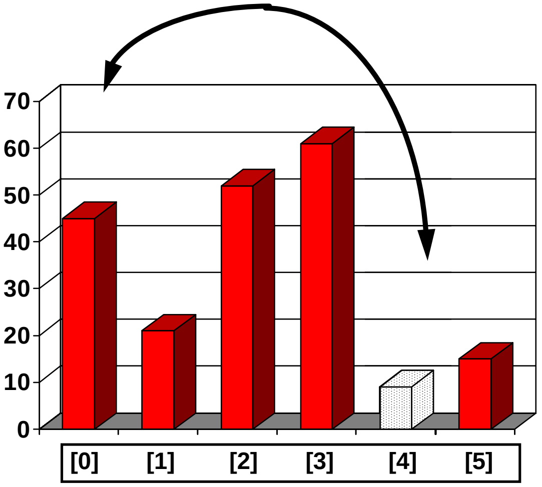
- ❒ Start by finding the **smallest** entry.



The first sorting algorithm that we'll examine is called Selectionsort. It begins by going through the entire array and finding the smallest element. In this example, the smallest element is the number 8 at location [4] of the array.

The Selectionsort Algorithm

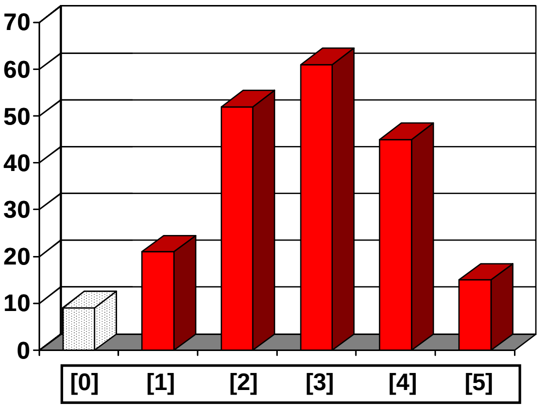
- ❐ Start by finding the **smallest** entry.
- ❐ Swap the smallest entry with the **first entry**.



Once we have found the smallest element, that element is swapped with the first element of the array...

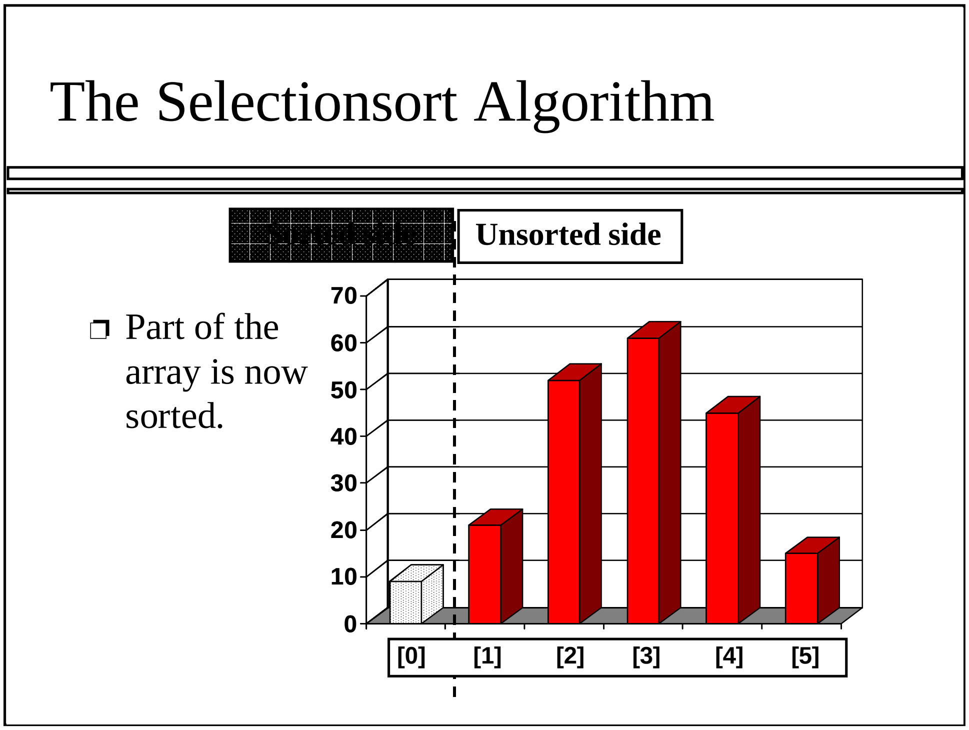
The Selectionsort Algorithm

- Start by finding the **smallest** entry.
- Swap the smallest entry with the **first entry**.



...like this.

The smallest element is now at the front of the array, and we have taken one small step toward producing a sorted array.



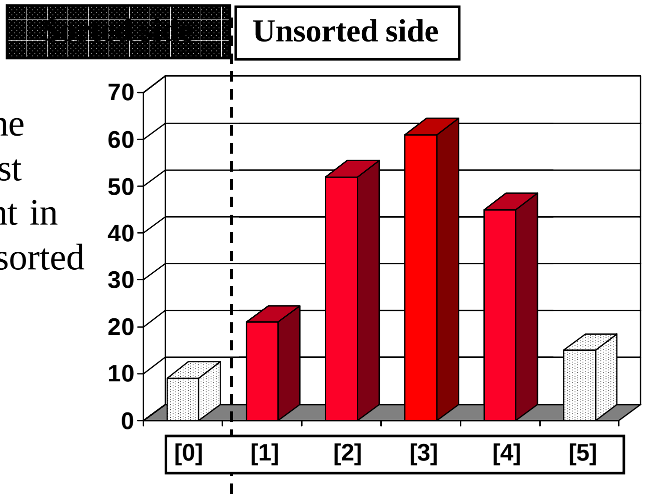
The Selectionsort Algorithm

- Part of the array is now sorted.

At this point, we can view the array as being split into two sides: To the left of the dotted line is the "sorted side", and to the right of the dotted line is the "unsorted side". Our goal is to push the dotted line forward, increasing the number of elements in the sorted side, until the entire array is sorted.

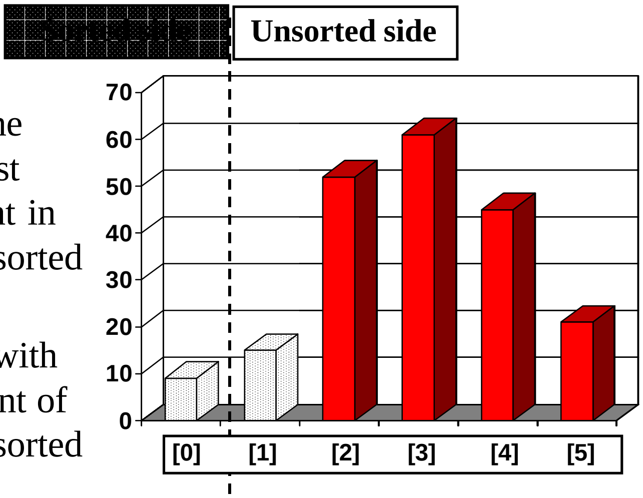
The Selectionsort Algorithm

- Find the smallest element in the unsorted side.



Each step of the Selectionsort works by finding the smallest element in the unsorted side. At this point, we would find the number 15 at location [5] in the unsorted side.

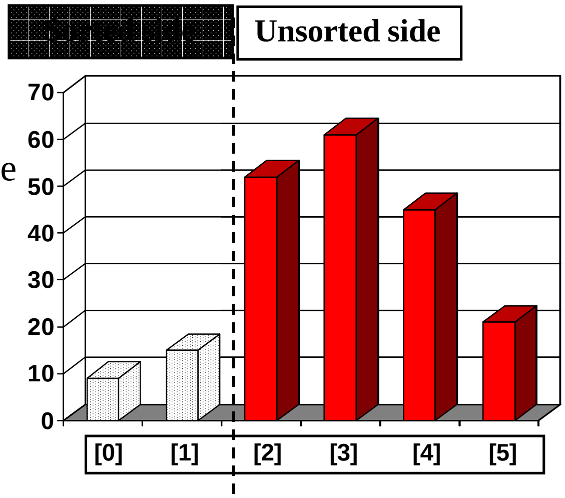
The Selectionsort Algorithm



- ❐ Find the smallest element in the unsorted side.
- ❐ Swap with the front of the unsorted side.

This small element is swapped with the number at the front of the unsorted side, as shown here...

The Selectionsort Algorithm



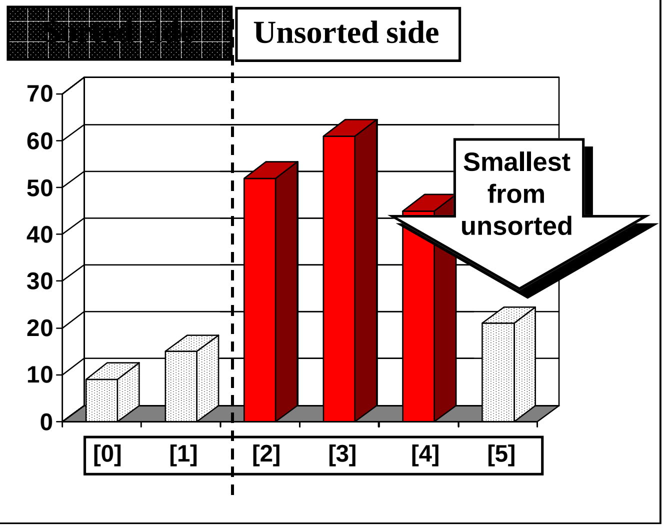
- ❐ We have increased the size of the sorted side by one element.

...and the effect is to increase the size of the sorted side by one element.

As you can see, the sorted side always contains the smallest numbers, and those numbers are sorted from small to large. The unsorted side contains the rest of the numbers, and those numbers are in no particular order.

The Selectionsort Algorithm

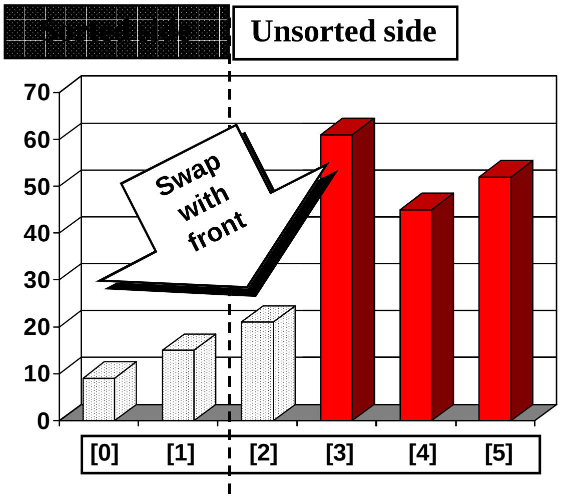
- The process continues...



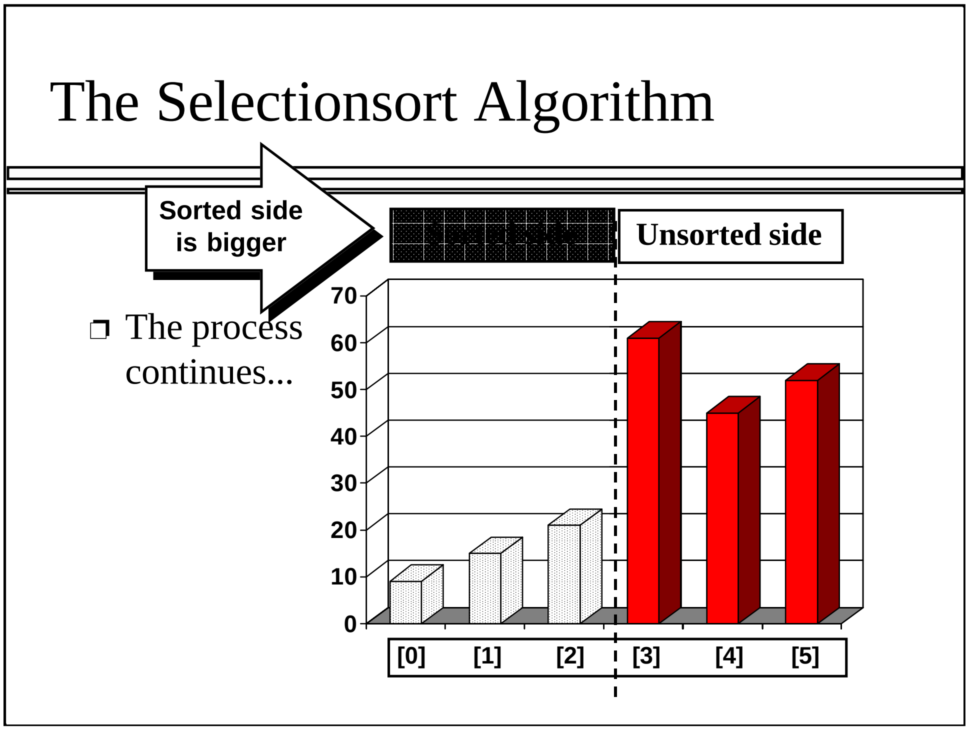
Again, we find the smallest entry in the unsorted side...

The Selectionsort Algorithm

- ❐ The process continues...



...and swap this element with the front of the unsorted side.



The Selectionsort Algorithm

Sorted side is bigger

Sorted side

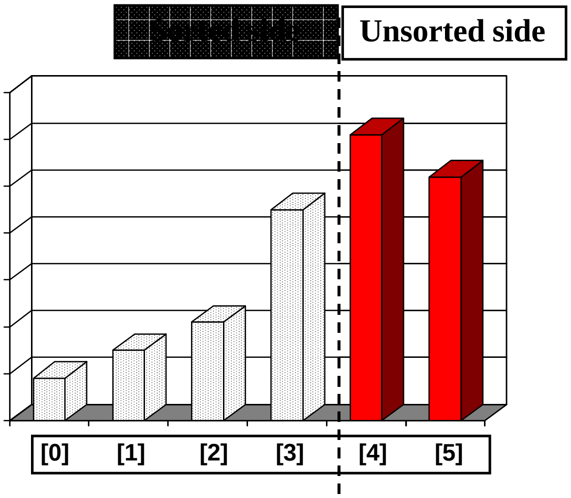
Unsorted side

- The process continues...

The sorted side now contains the three smallest elements of the array.

The Selectionsort Algorithm

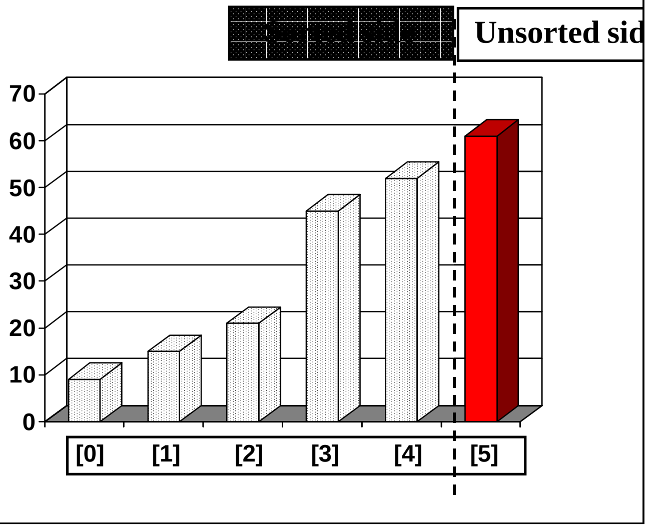
- The process keeps adding one more number to the sorted side.
- The sorted side has the smallest numbers, arranged from small to large.



Here is the array after increasing the sorted side to four elements.

The Selectionsort Algorithm

- We can stop when the unsorted side has just one number, since that number must be the largest number.



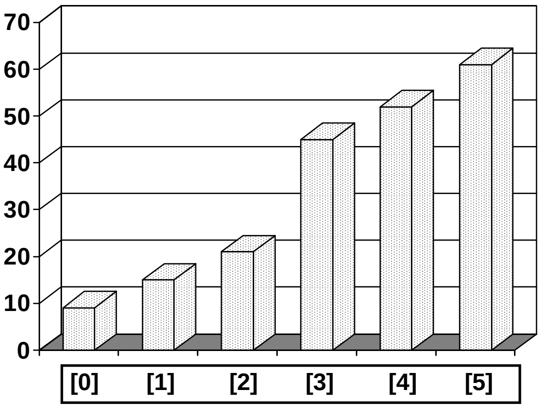
And now the sorted side has five elements.

In fact, once the unsorted side is down to a single element, the sort is completed. At this point the 5 smallest elements are in the sorted side, and so the the one largest element is left in the unsorted side.

We are done...

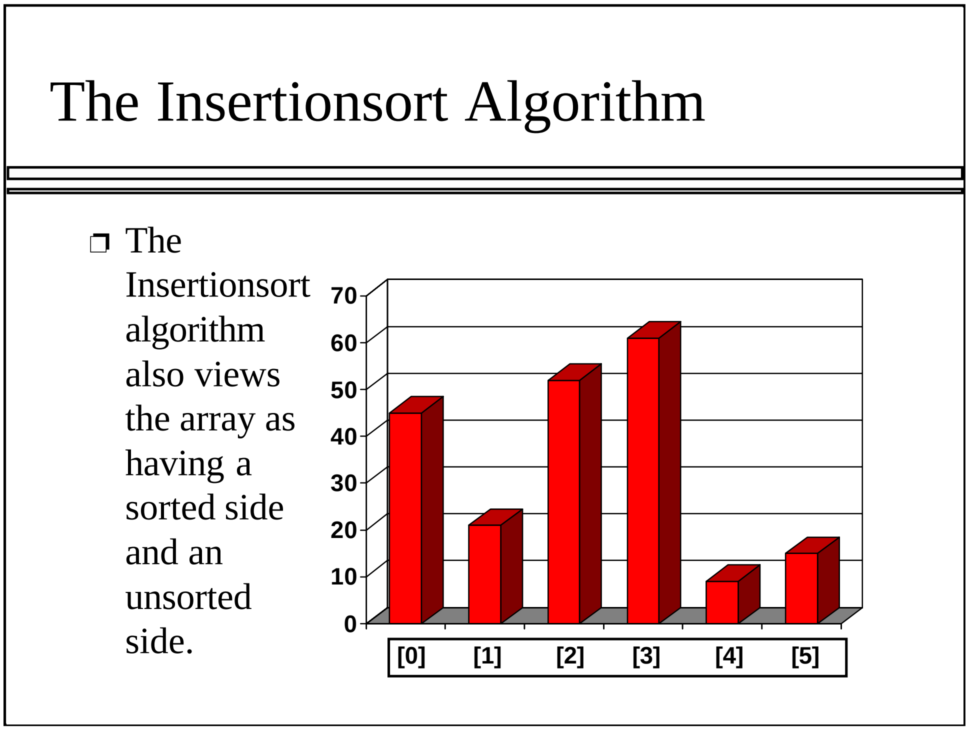
The Selectionsort Algorithm

- ❒ The array is now sorted.
- ❒ We repeatedly **selected** the smallest element, and moved this element to the front of the unsorted side.



...The array is sorted.

The basic algorithm is easy to state and also easy to program.

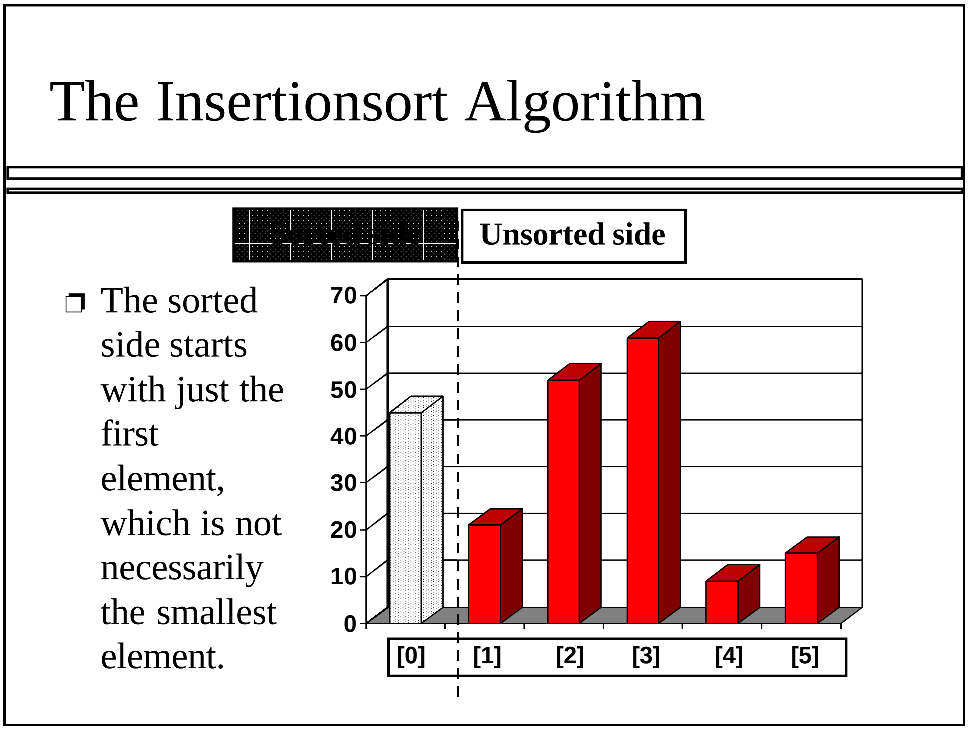


The Insertionsort Algorithm

- The Insertionsort algorithm also views the array as having a sorted side and an unsorted side.

Now we'll look at another sorting method called Insertionsort. The end result will be the same: The array will be sorted from smallest to largest. But the sorting method is different.

However, there are some common features. As with the Selectionsort, the Insertionsort algorithm also views the array as having a sorted side and an unsorted side, ...



The Insertionsort Algorithm

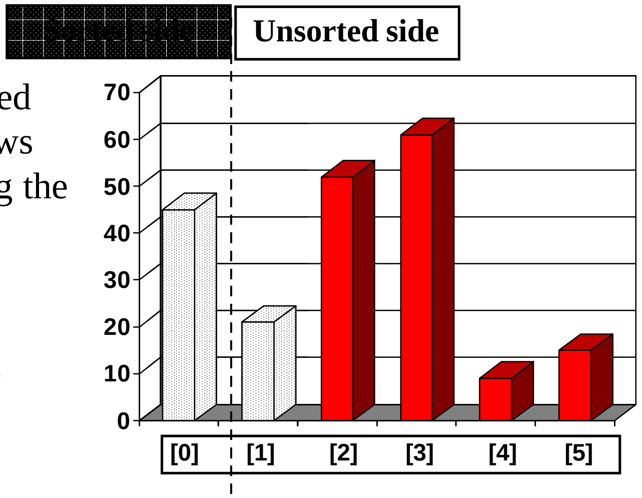
- The sorted side starts with just the first element, which is not necessarily the smallest element.

...like this.

However, in the Selectionsort, the sorted side always contained the smallest elements of the array. In the Insertionsort, the sorted side will be sorted from small to large, but the elements in the sorted side will not necessarily be the smallest entries of the array.

Because the sorted side does not need to have the smallest entries, we can start by placing one element in the sorted side--we don't need to worry about sorting just one element. But we do need to worry about how to increase the number of elements that are in the sorted side.

The Insertionsort Algorithm

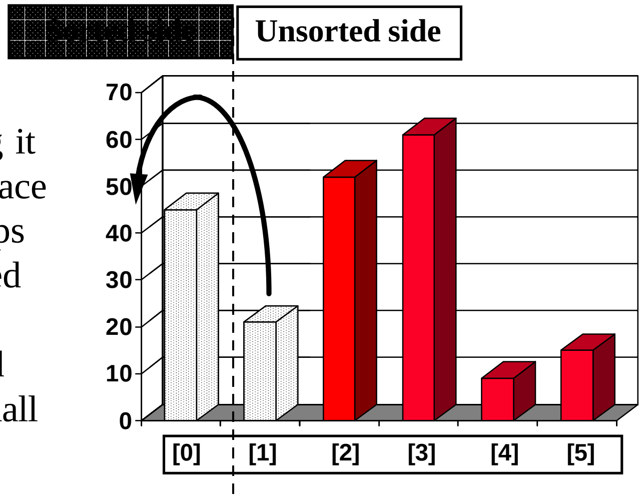


❒ The sorted side grows by taking the front element from the unsorted side...

The basic approach is to take the front element from the unsorted side...

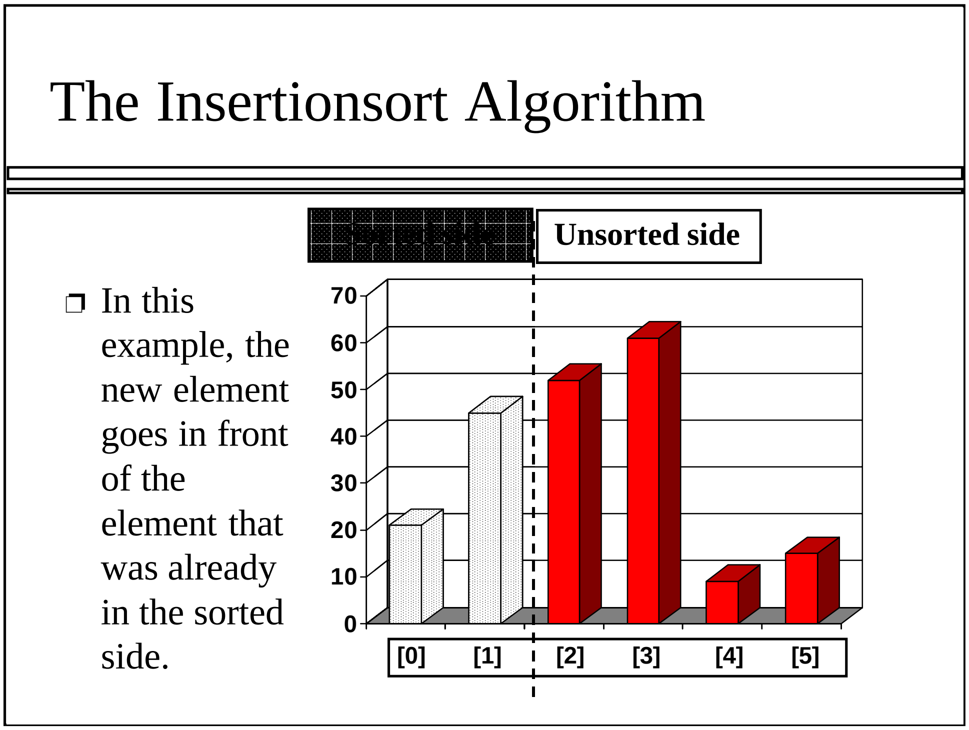
The Insertionsort Algorithm

- ...and inserting it in the place that keeps the sorted side arranged from small to large.



...and insert this element at the correct spot of the sorted side.

In this example, the front element of the unsorted side is 20. So the 20 must be inserted before the number 45 which is already in the sorted side.

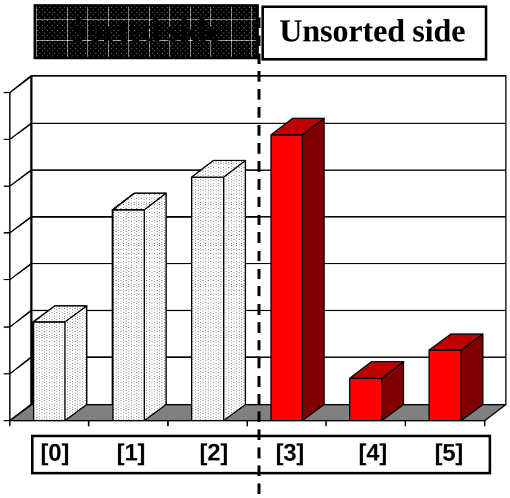


The Insertionsort Algorithm

- In this example, the new element goes in front of the element that was already in the sorted side.

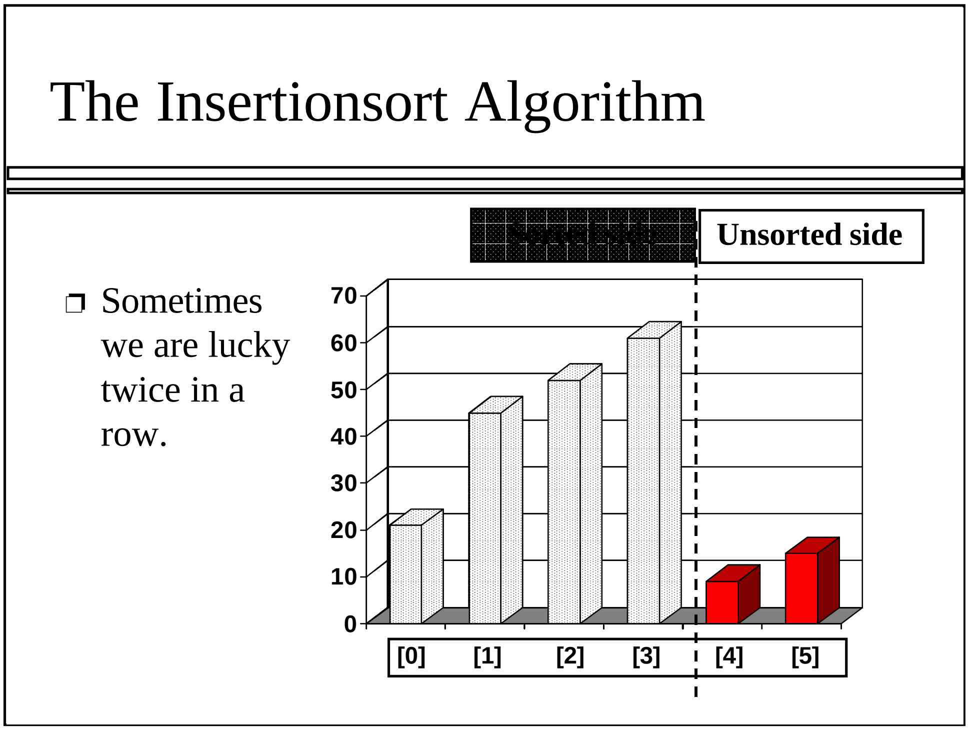
After the insertion, the sorted side contains two elements. These two elements are in order from small to large, although they are not the smallest elements in the array.

The Insertionsort Algorithm



- ❐ Sometimes we are lucky and the new inserted item doesn't need to move at all.

Sometimes we are lucky and the newly inserted element is already in the right spot. This happens if the new element is larger than anything that's already in the array.



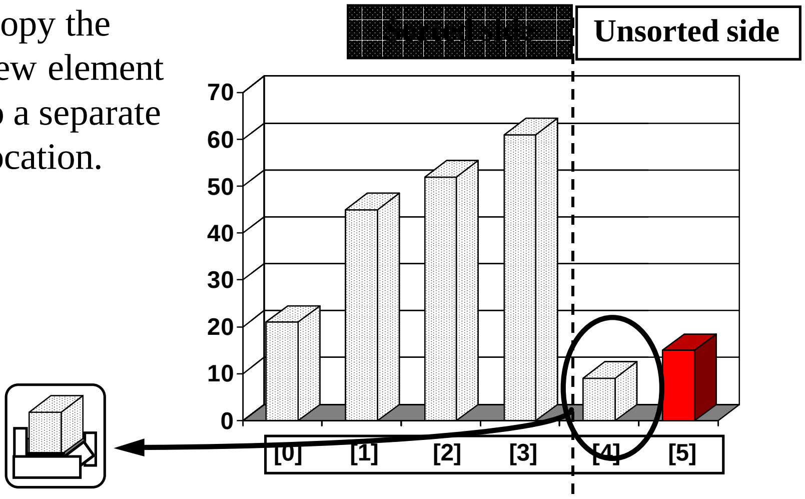
The Insertionsort Algorithm

- Sometimes we are lucky twice in a row.

Sometimes we are lucky twice in a row.

How to Insert One Element

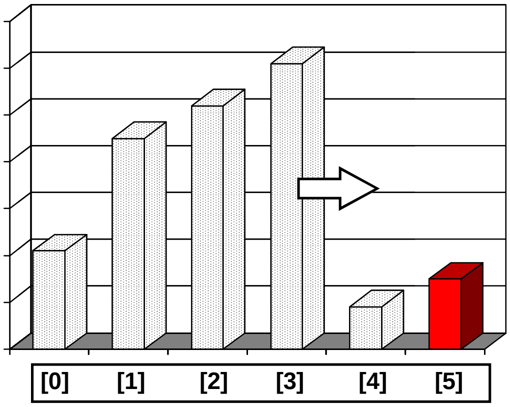
❶ Copy the new element to a separate location.



The actual insertion process requires a bit of work that is shown here. The first step of the insertion is to make a copy of the new element. Usually this copy is stored in a local variable. It just sits off to the side, ready for us to use whenever we need it.

How to Insert One Element

❷ Shift elements in the sorted side, creating an open space for the new element.

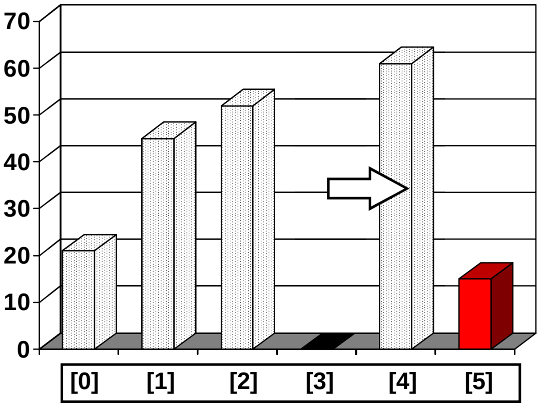


After we have safely made a copy of the new element, we start shifting elements from the end of the sorted side. These elements are shifted rightward, to create an "empty spot" for our new element to be placed.

In this example we take the last element of the sorted side and shift it rightward one spot...

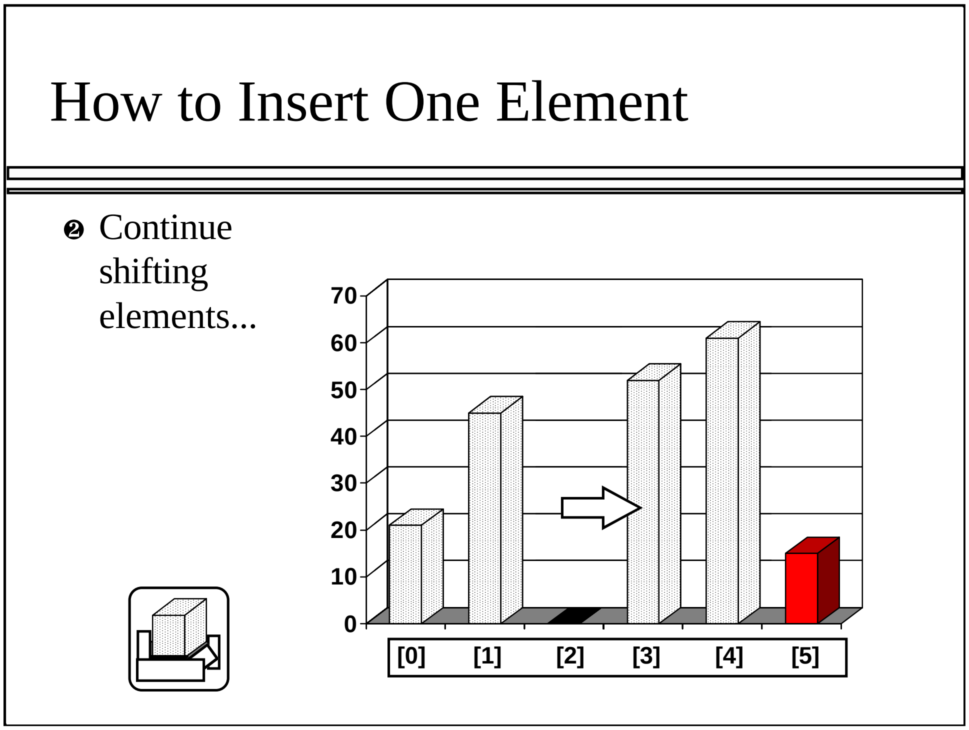
How to Insert One Element

❷ Shift elements in the sorted side, creating an open space for the new element.



...like this.

Is this the correct spot for the new element? No, because the new element is smaller than the next element in the sorted section. So we continue shifting elements rightward...



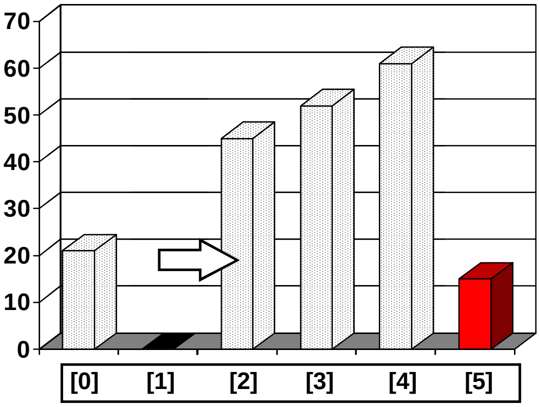
How to Insert One Element

❷ Continue shifting elements...

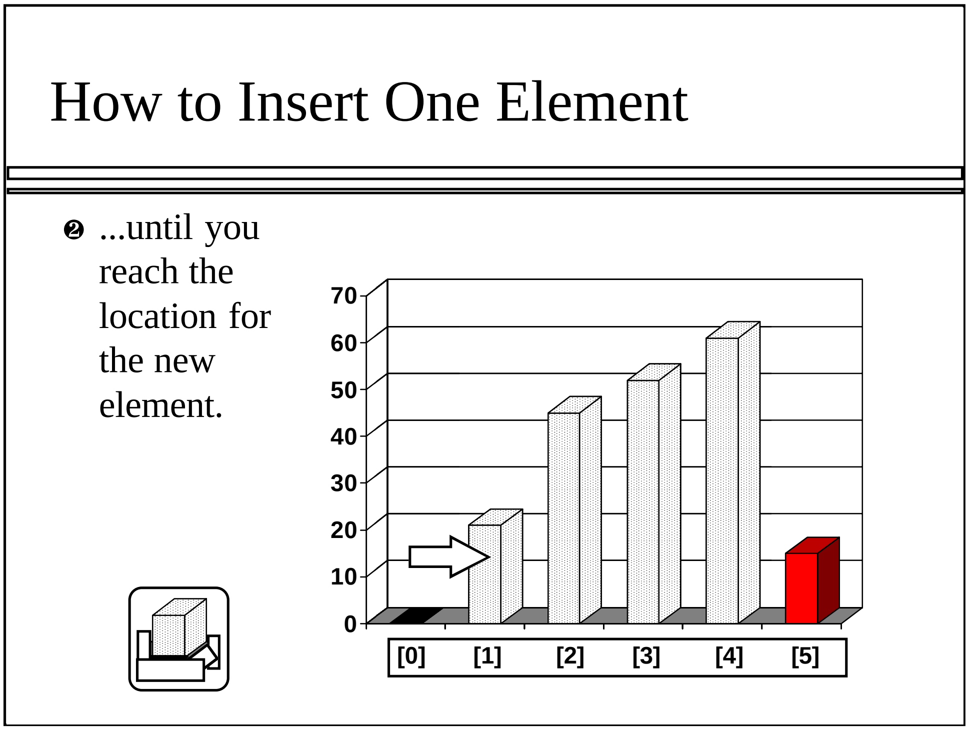
This is still not the correct spot for our new element, so we shift again...

How to Insert One Element

❷ Continue shifting elements...



...and shift one more time...



How to Insert One Element

❷ ...until you reach the location for the new element.

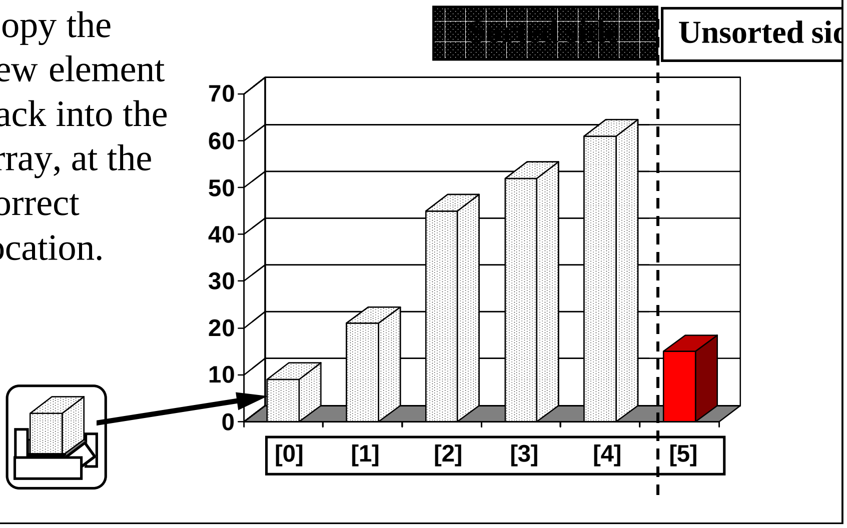
Finally, this is the correct location for the new element. In general there are two situations that indicate the "correct location" has been found:

1. We reach the front of the array (as happened here), or

2. We reached an element that is less than or equal to the new element.

How to Insert One Element

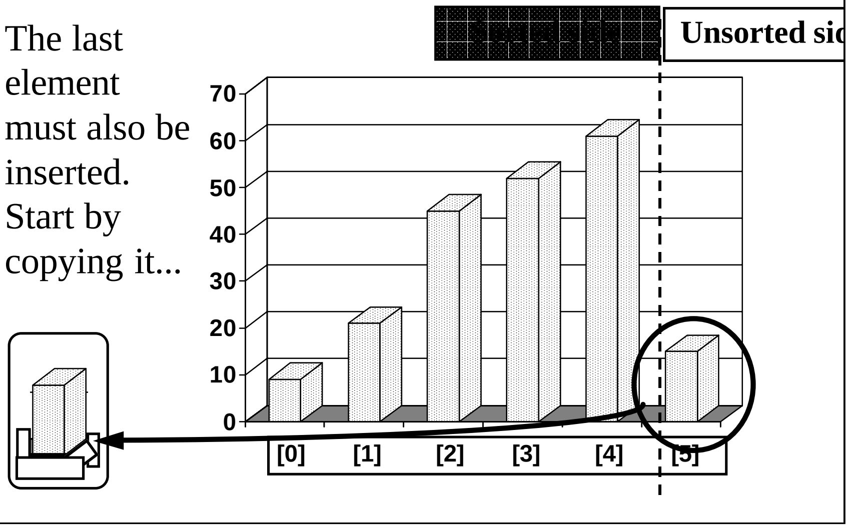
❸ Copy the new element back into the array, at the correct location.



Once the correct spot is found, we copy the new element back into the array. The number of elements in the sorted side has increased by one.

How to Insert One Element

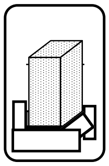
- The last element must also be inserted. Start by copying it...

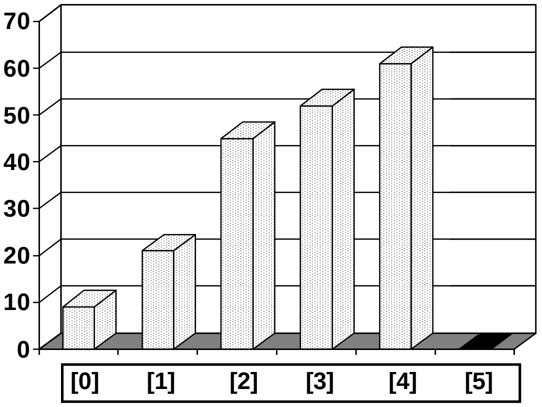


The last element of the array also needs to be inserted. Start by copying it to a safe location.

A Quiz

How many shifts will occur before we copy this element back into the array?

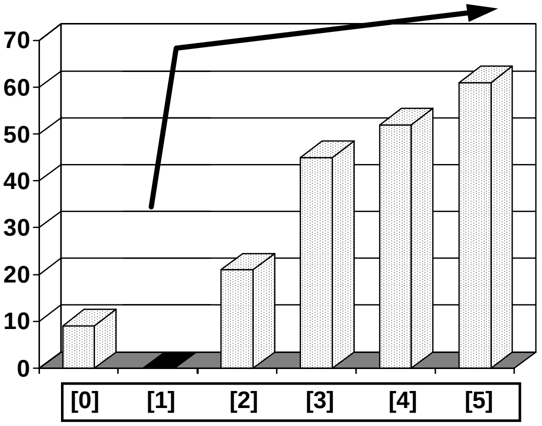




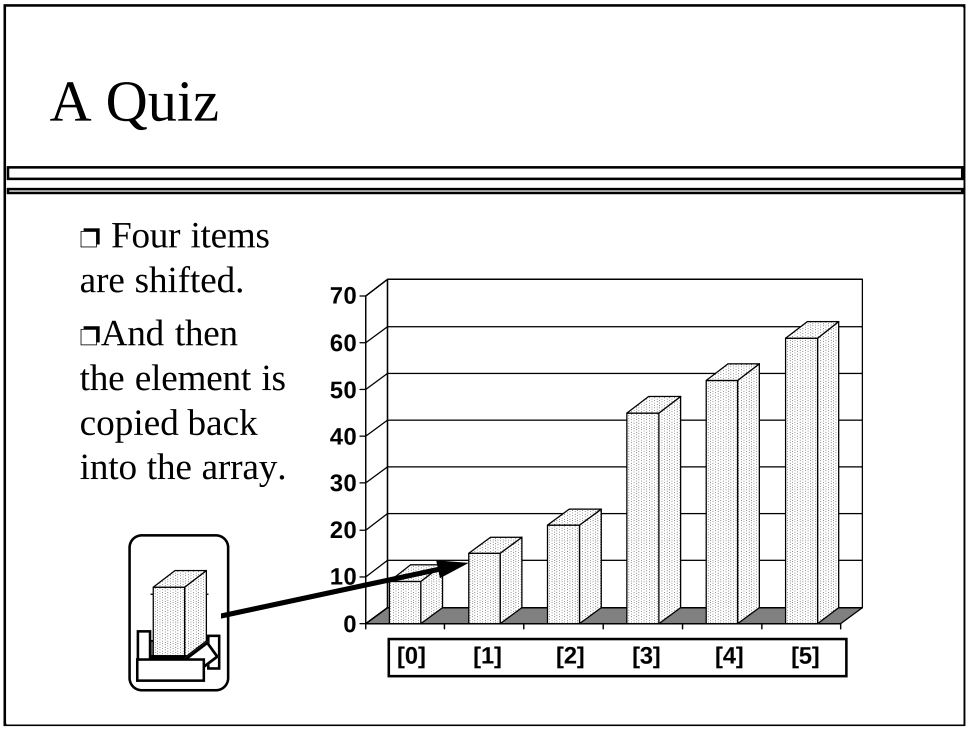
Now we will shift elements rightward. How many shifts will be done?

A Quiz

❐ Four items are shifted.



These four elements are shifted. But the 7 at location [0] is not shifted since it is smaller than the new element.



A Quiz

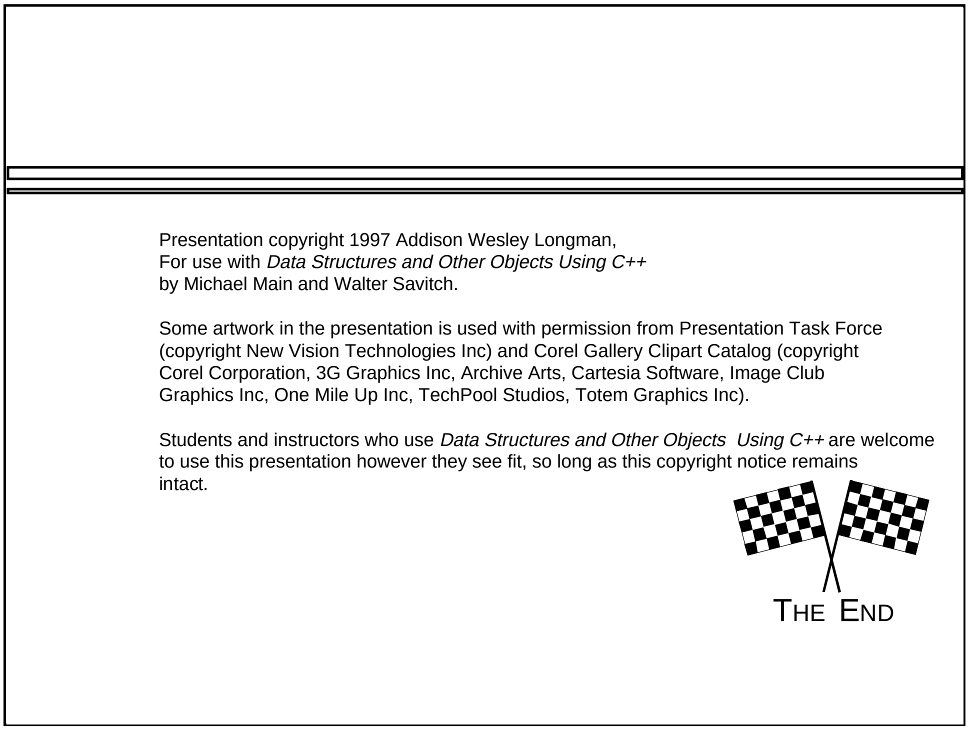
- ❒ Four items are shifted.
- ❒And then the element is copied back into the array.

The new element is inserted into the array.

Timing and Other Issues

- ❒ Both Selectionsort and Insertionsort have a worst-case time of $O(n^2)$, making them impractical for large arrays.
- ❒ But they are easy to program, easy to debug.
- ❒ Insertionsort also has good performance when the array is nearly sorted to begin with.
- ❒ But more sophisticated sorting algorithms are needed when good performance is needed in all cases for large arrays.

A quick summary of these algorithms' timing properties.



THE END

Feel free to send your ideas to:

Michael Main
main@colorado.edu