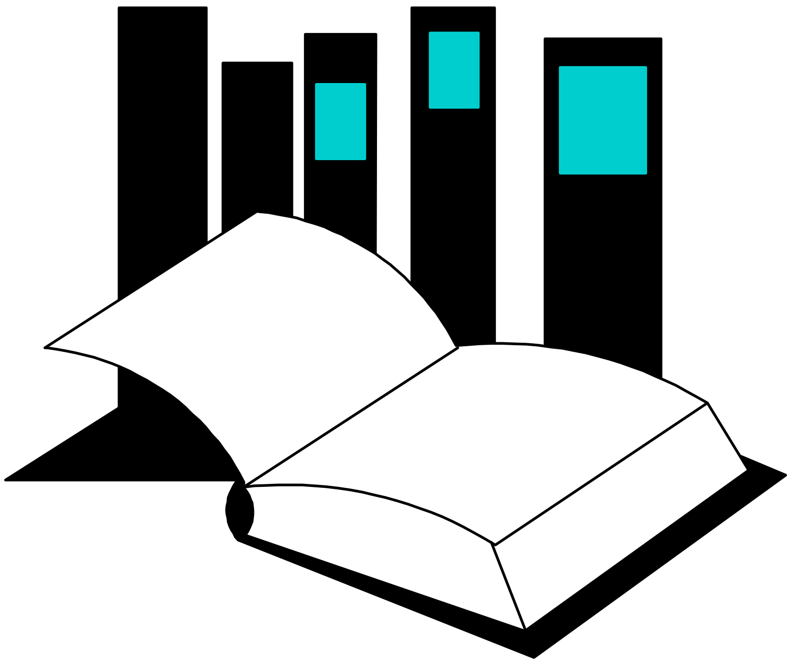
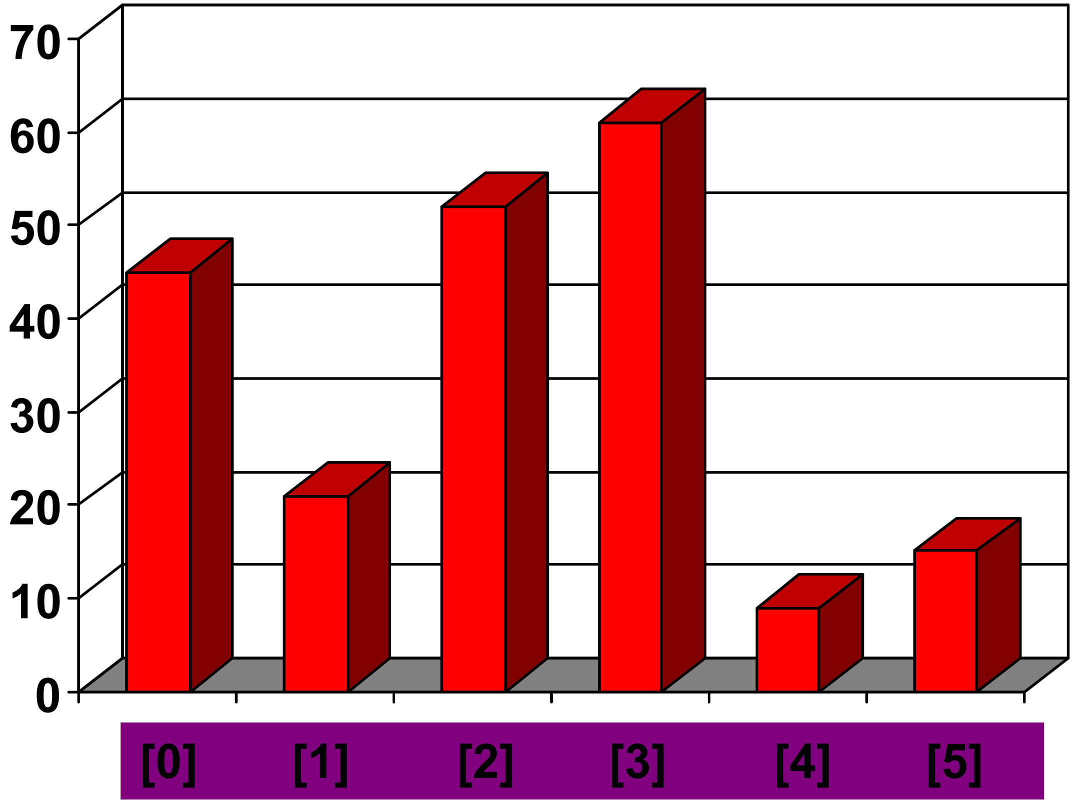
Quadratic Sorting



- 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.

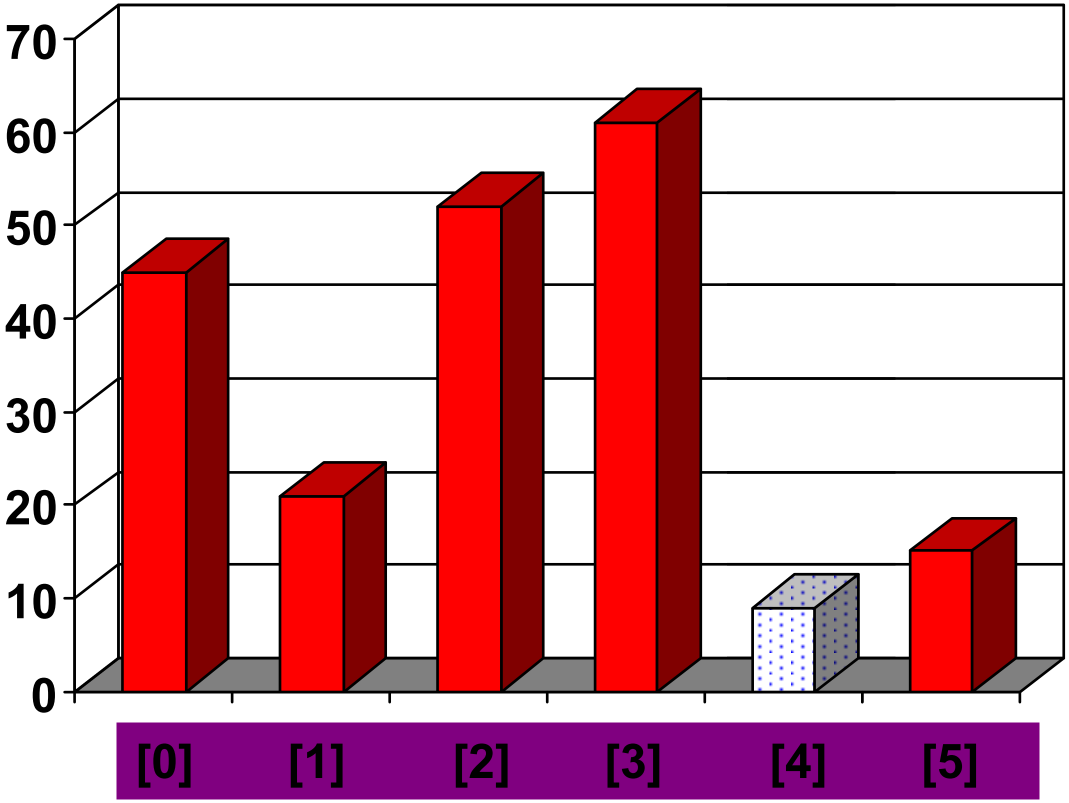
Sorting an Array of Integers

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



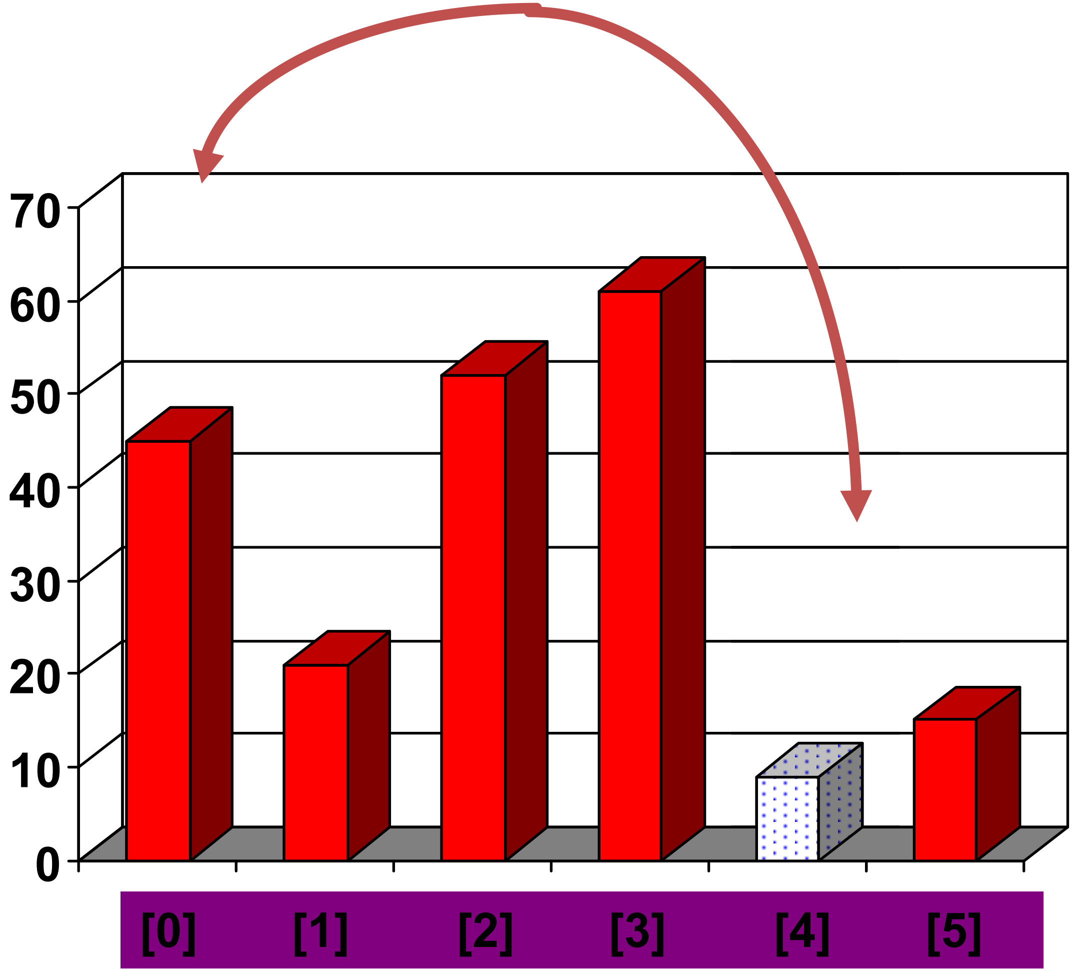
The Selectionsort Algorithm

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



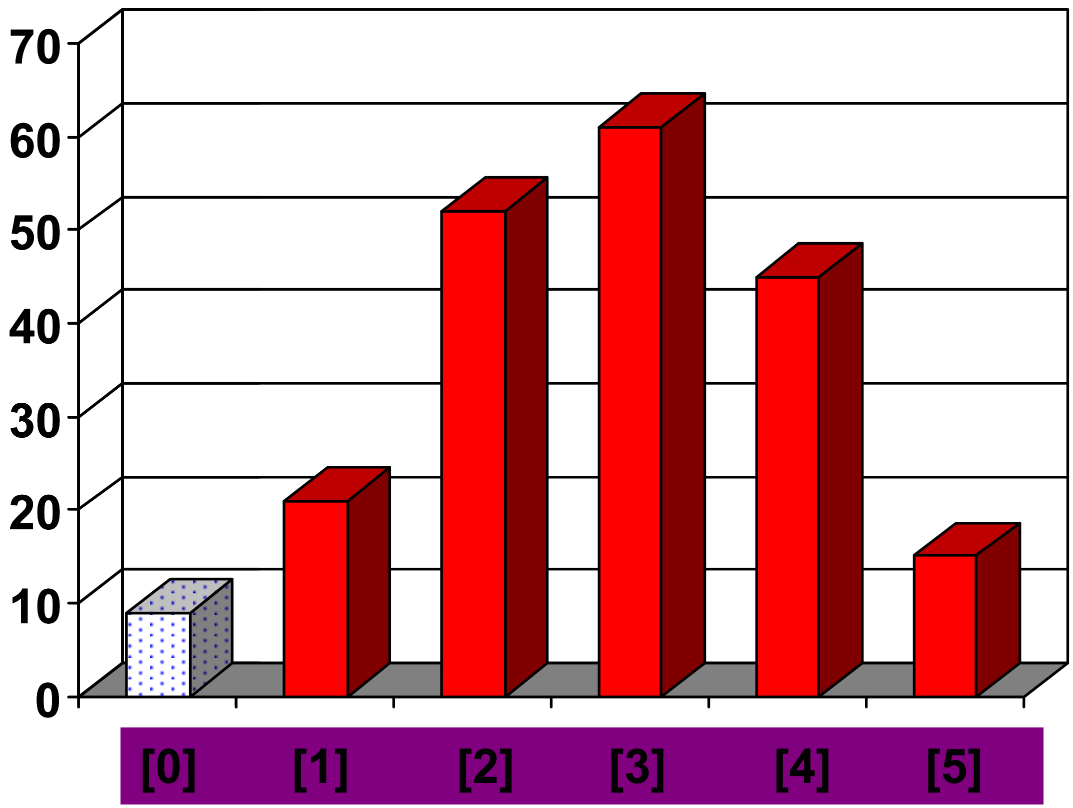
The Selectionsort Algorithm

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



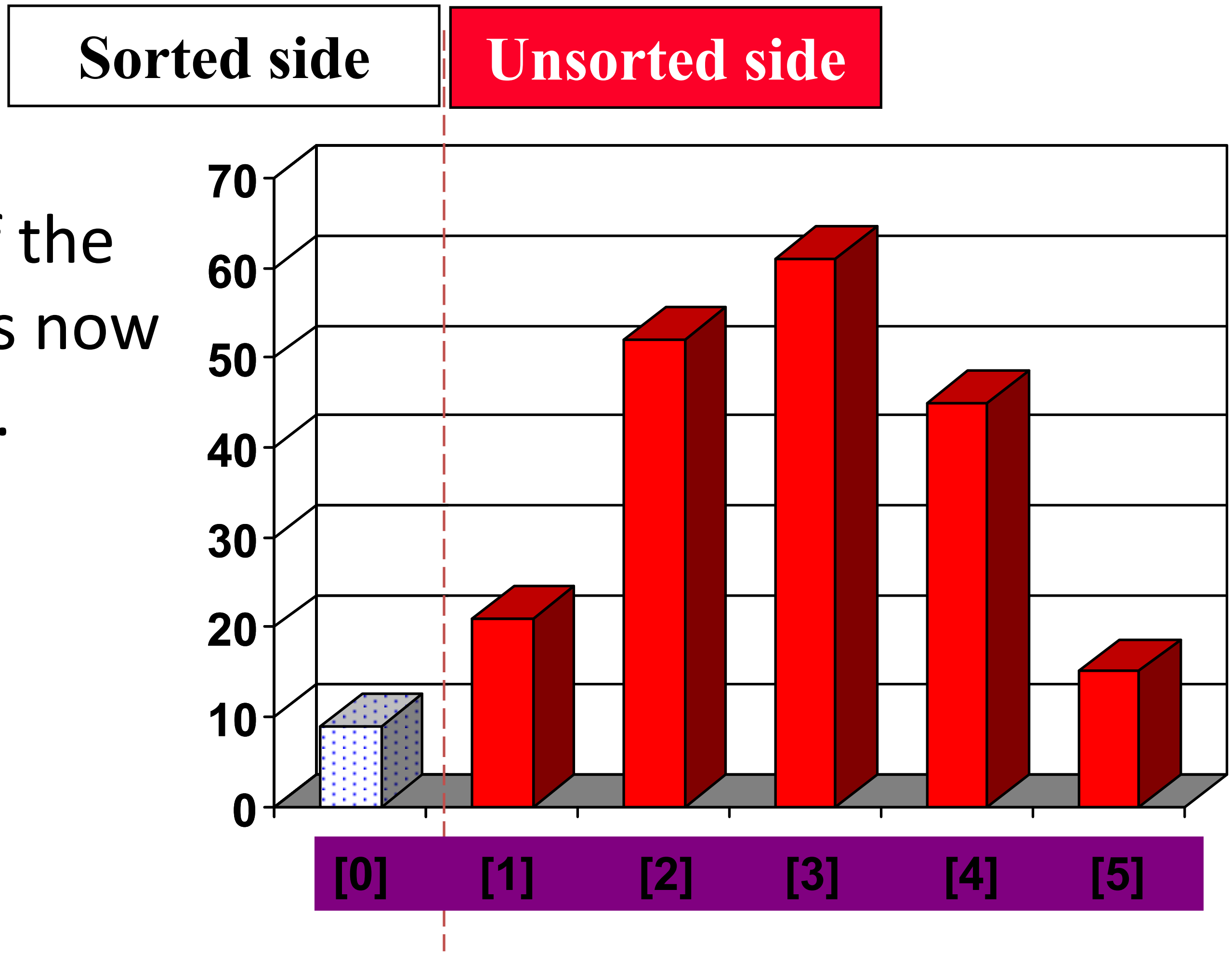
The Selectionsort Algorithm

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



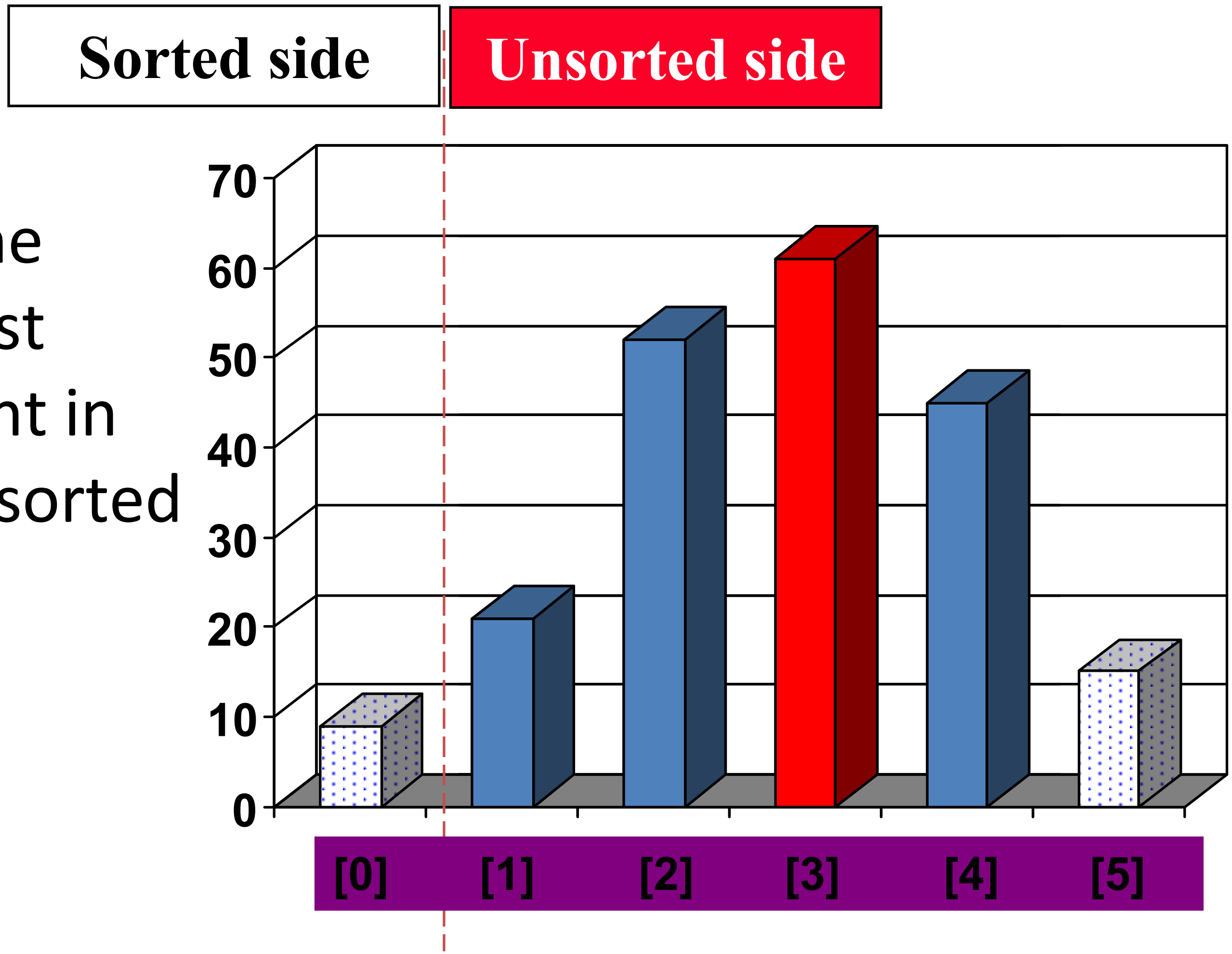
The Selectionsort Algorithm

- Part of the array is now sorted.



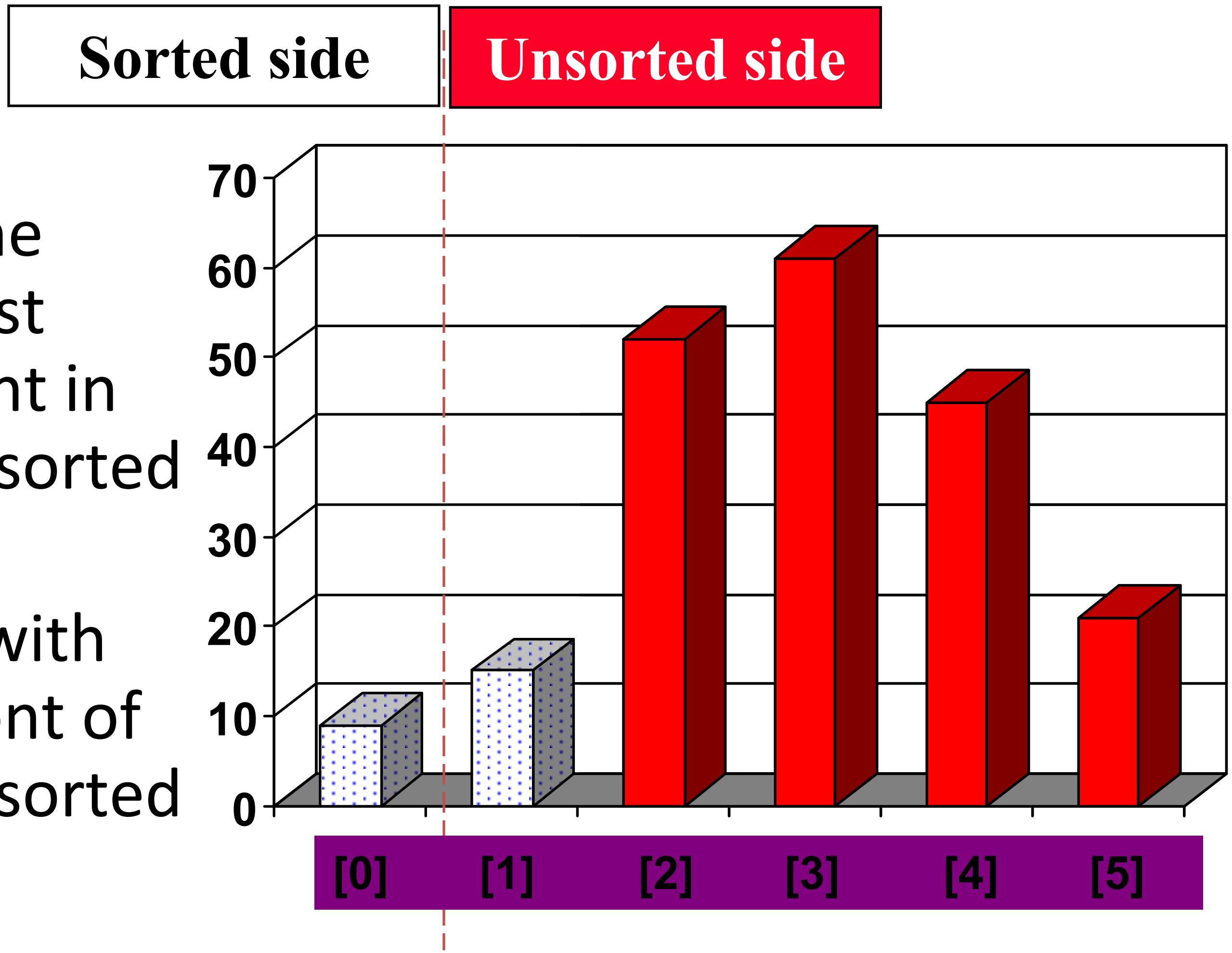
The Selectionsort Algorithm

- Find the smallest element in the unsorted side.



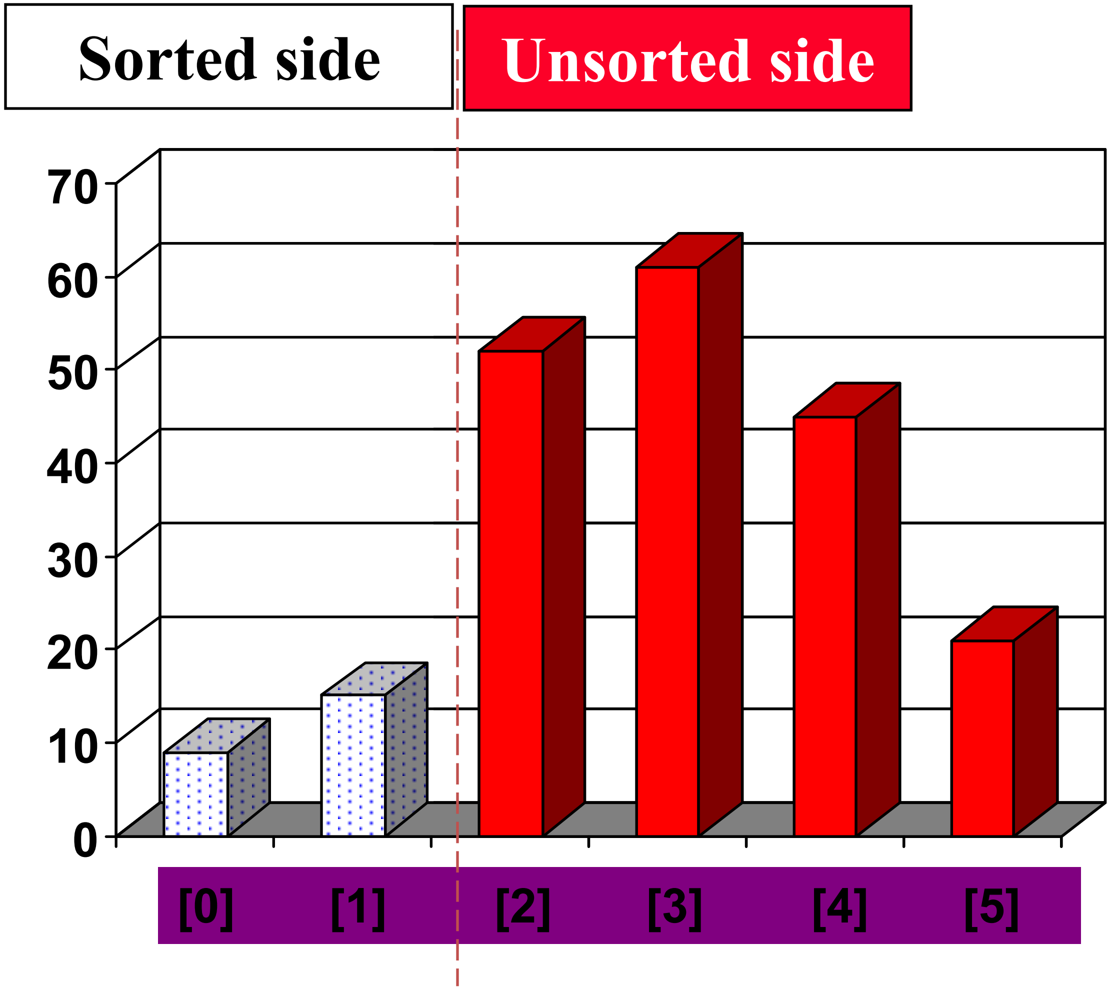
The Selectionsort Algorithm

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



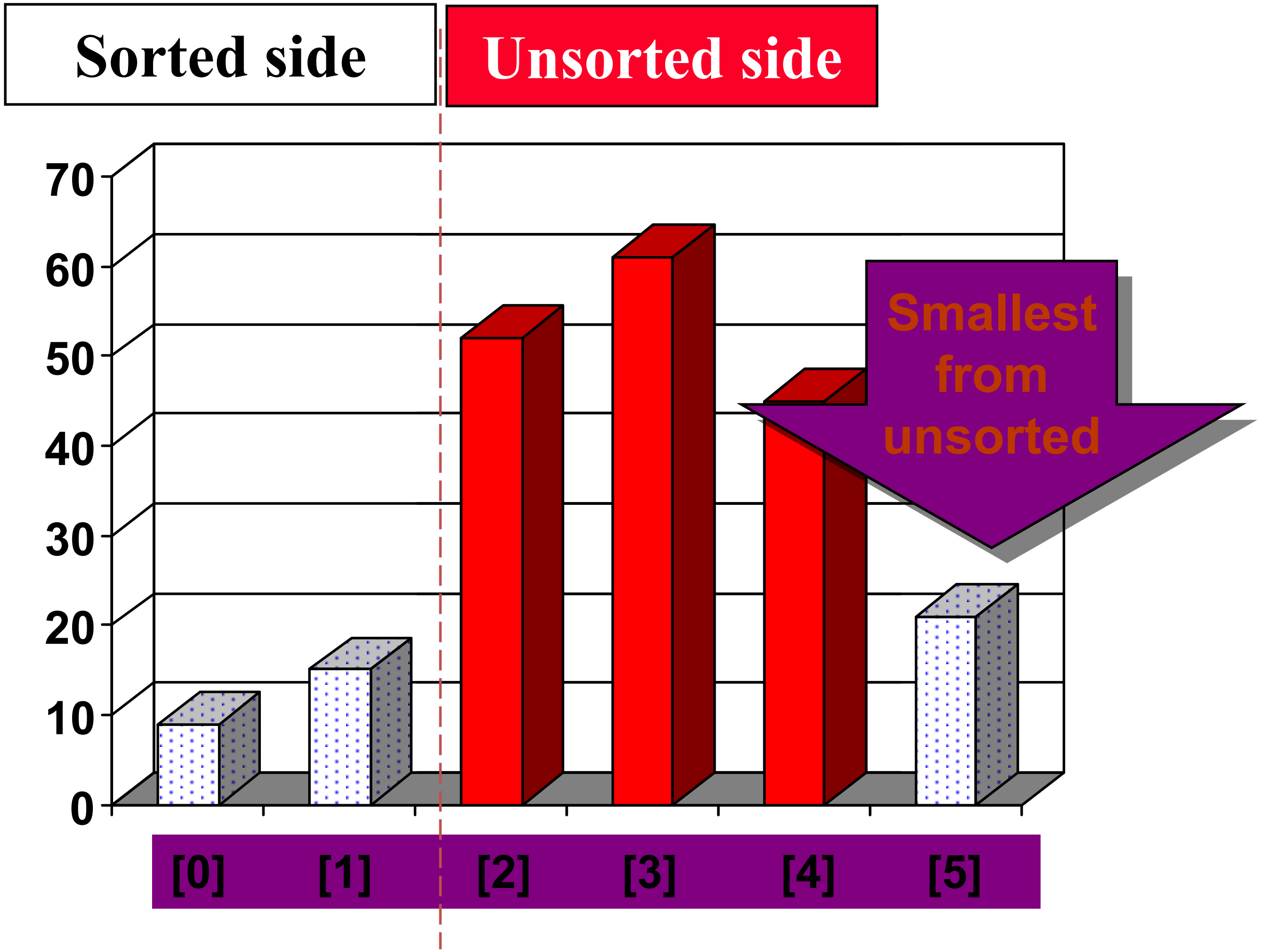
The Selectionsort Algorithm

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



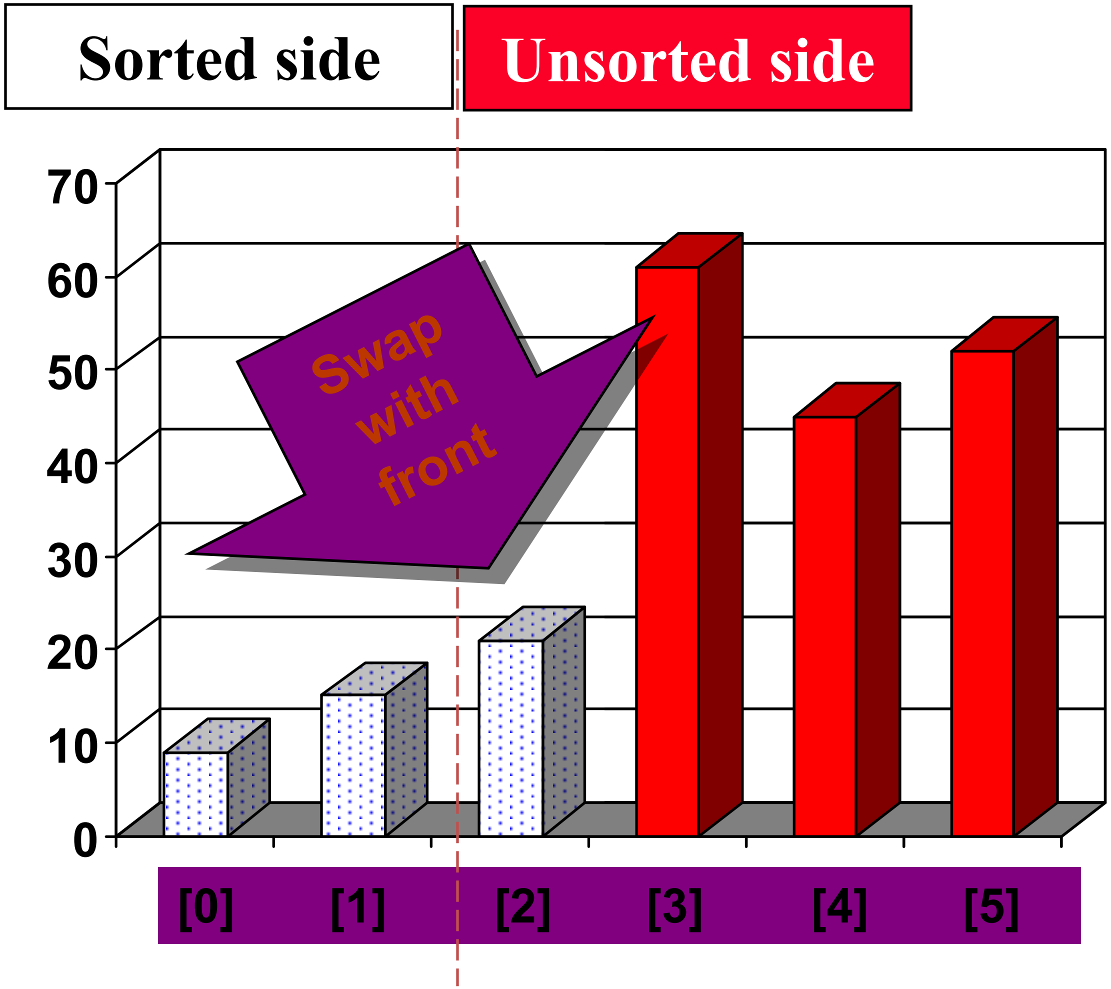
The Selectionsort Algorithm

- The process continues...

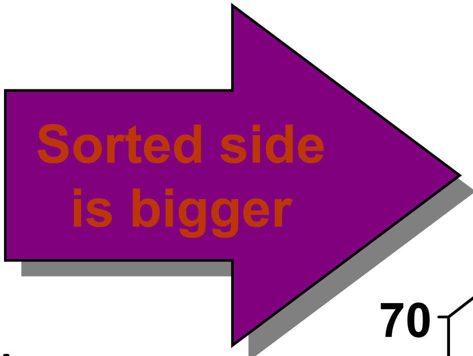


The Selectionsort Algorithm

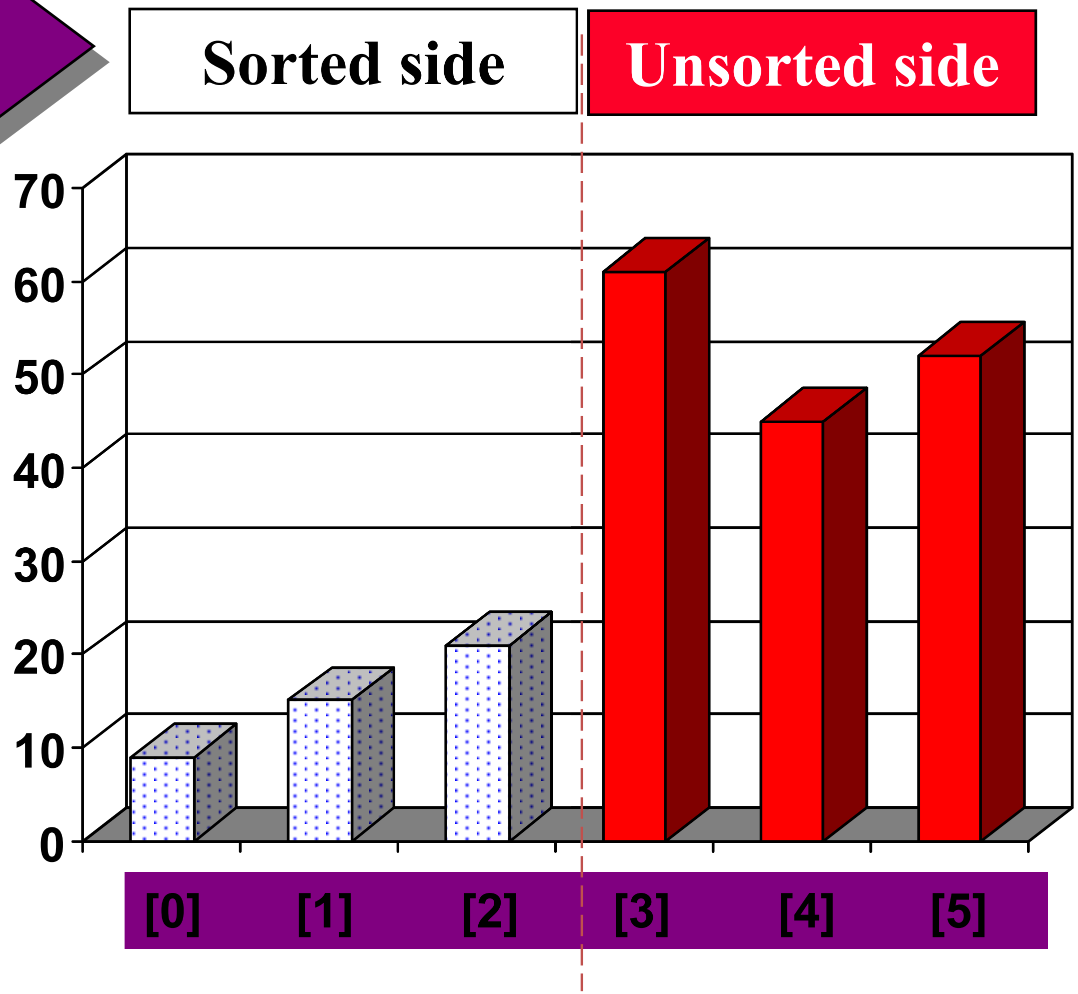
- The process continues...



The Selectionsort Algorithm

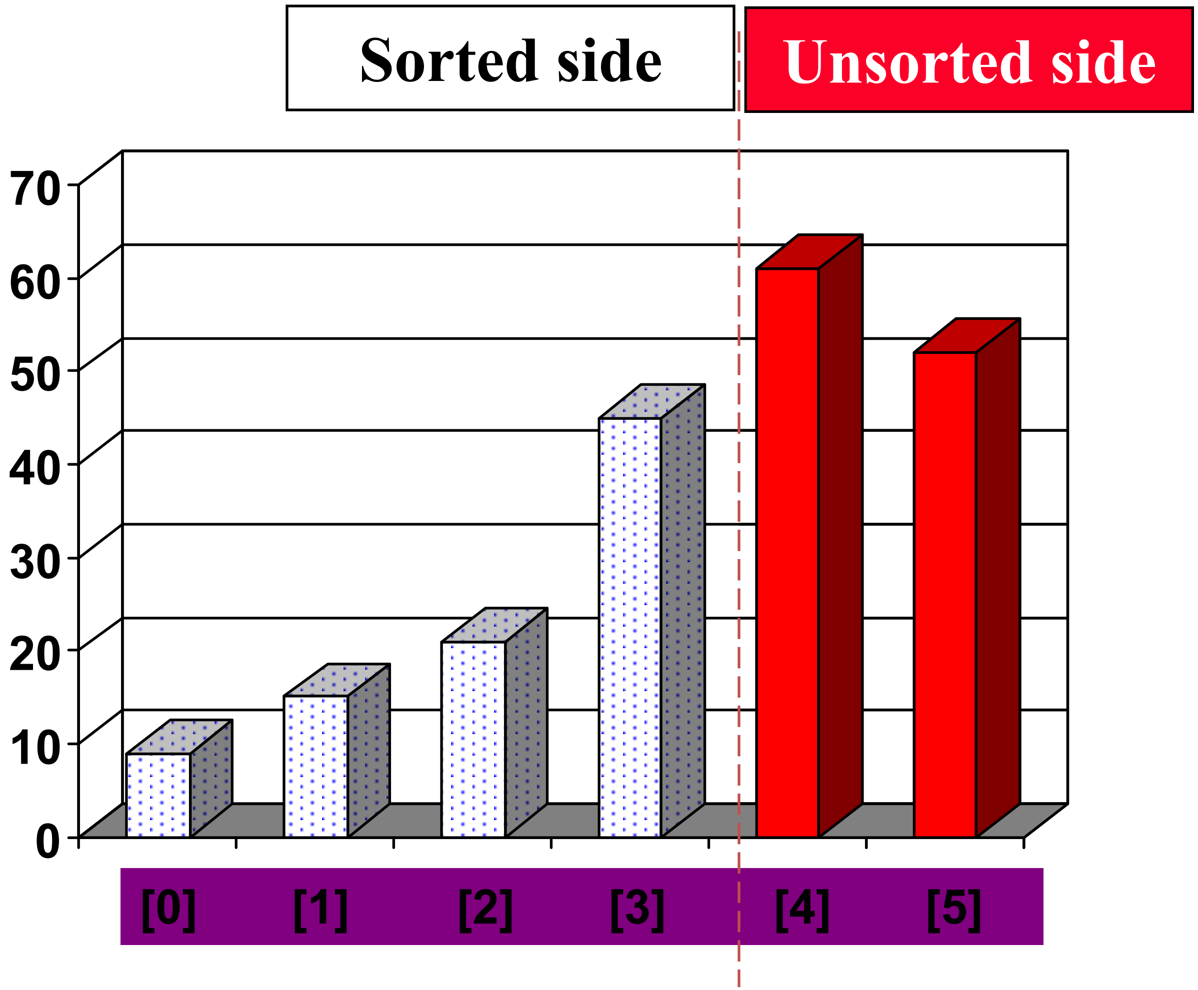


- The process continues...



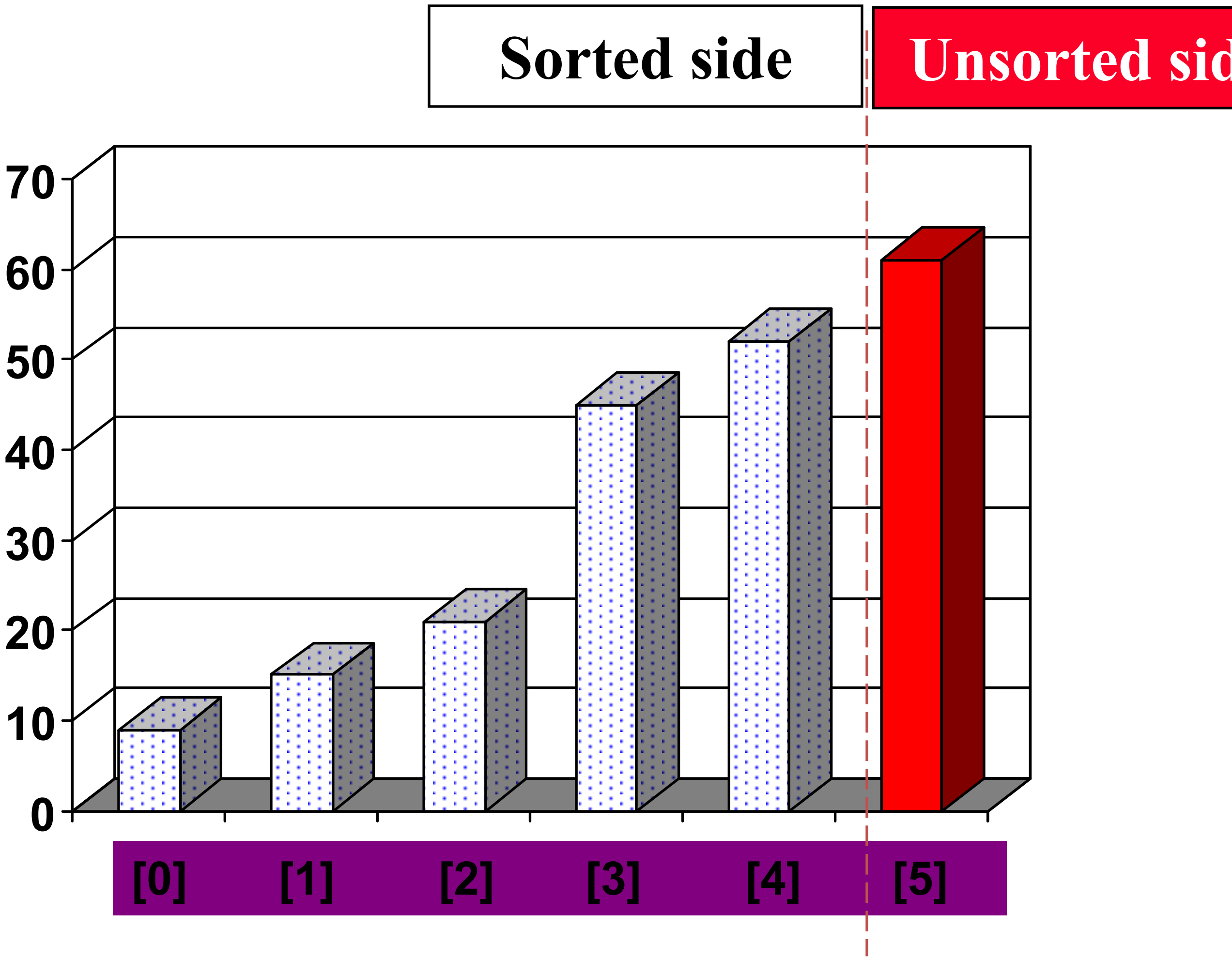
The Selectionsort Algorithm

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



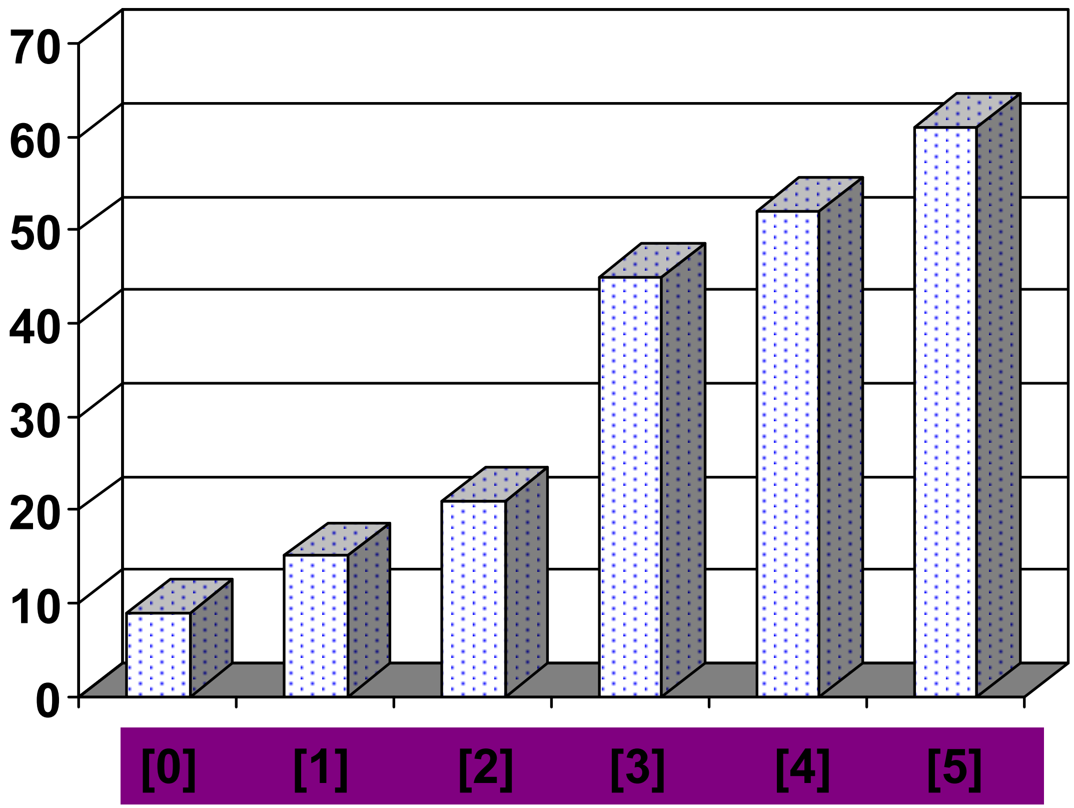
The Selectionsort Algorithm

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



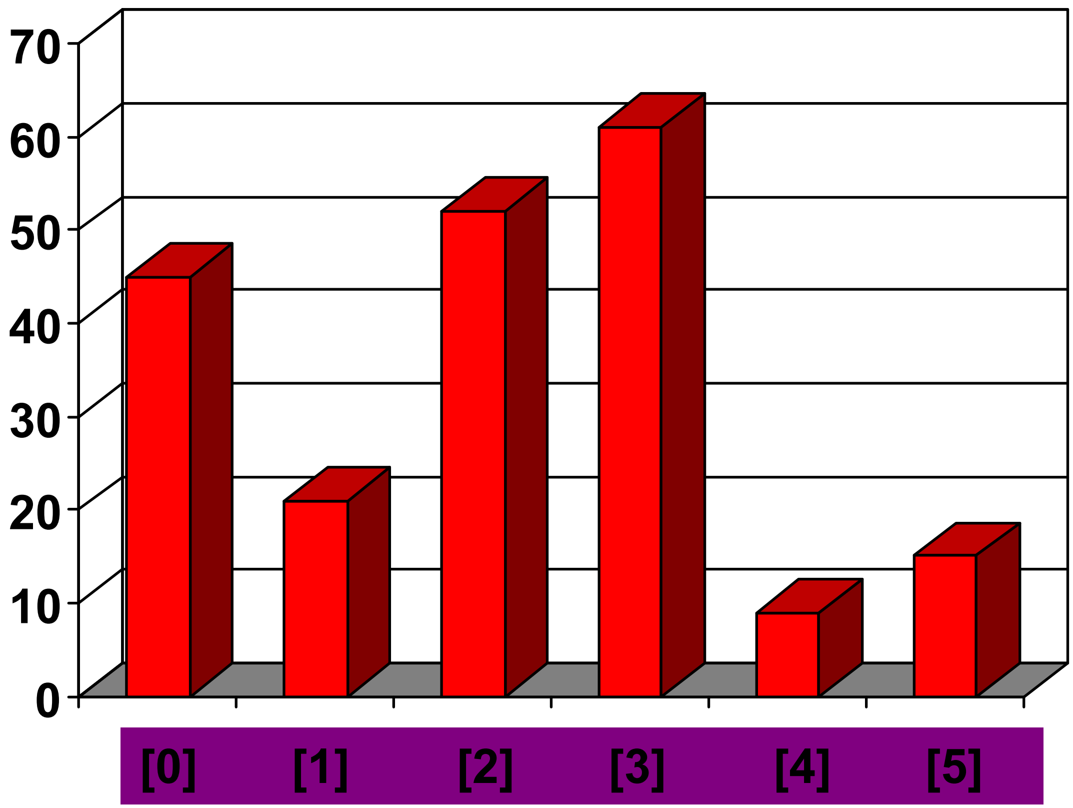
The Selectionsort Algorithm

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



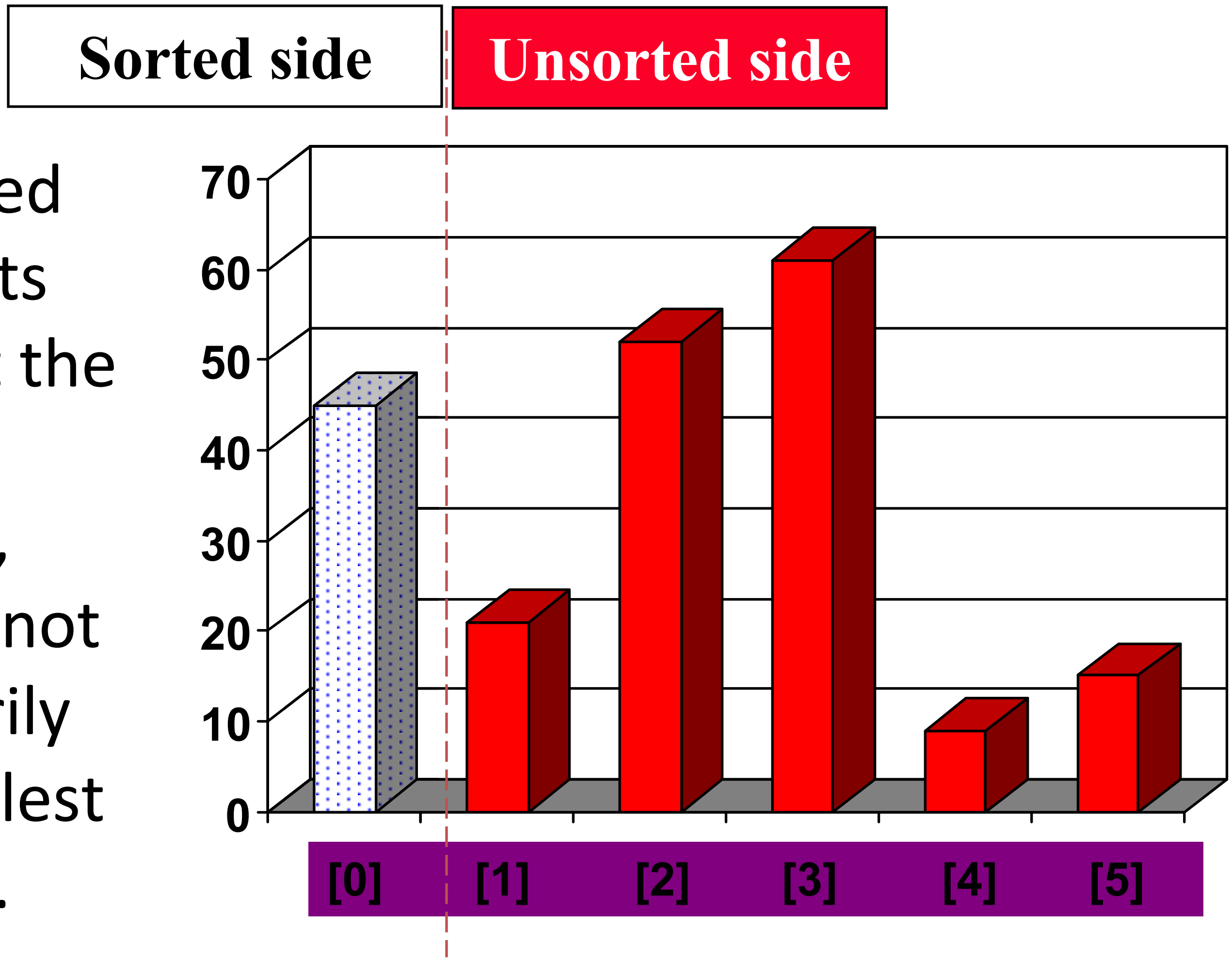
The Insertionsort Algorithm

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



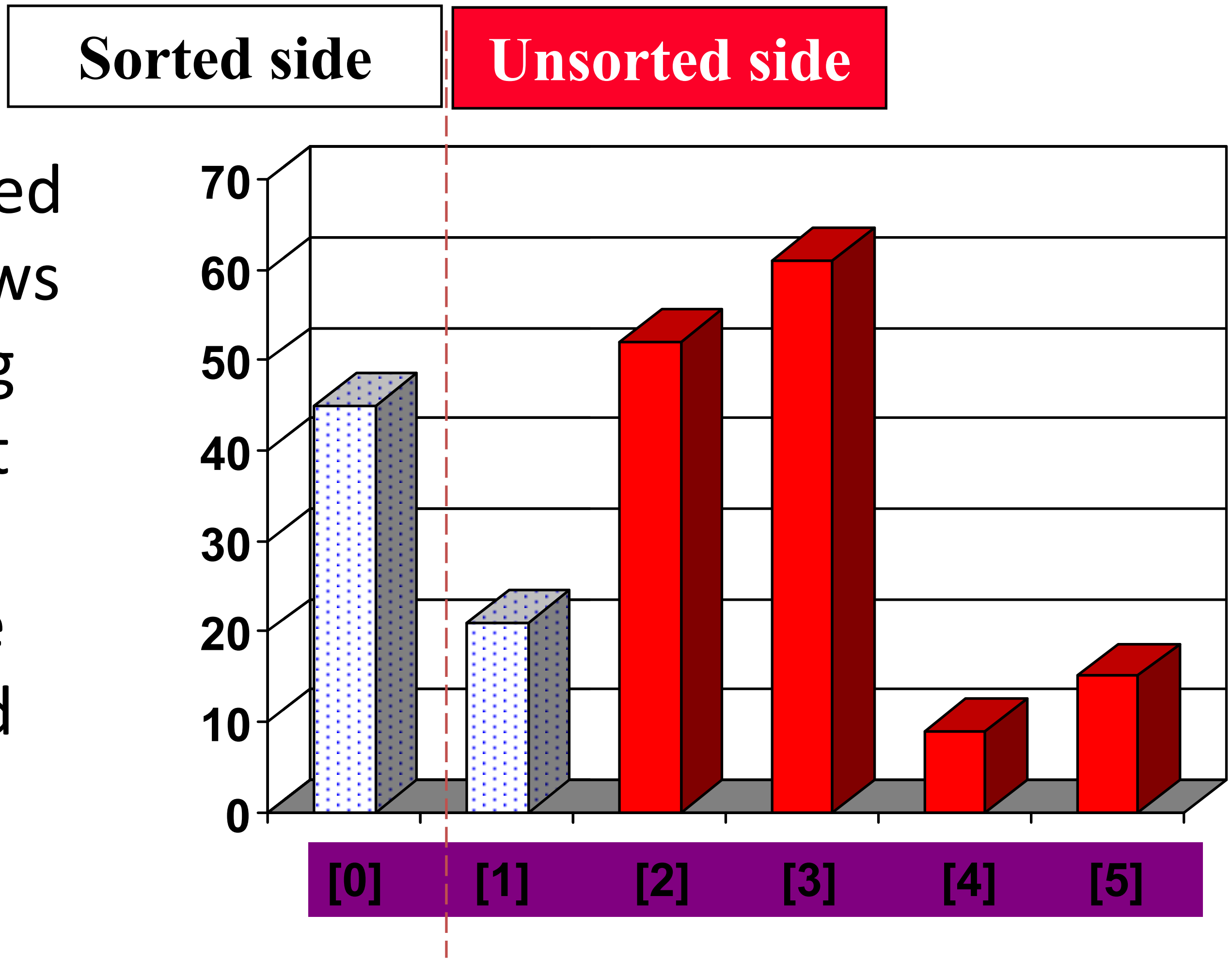
The Insertionsort Algorithm

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



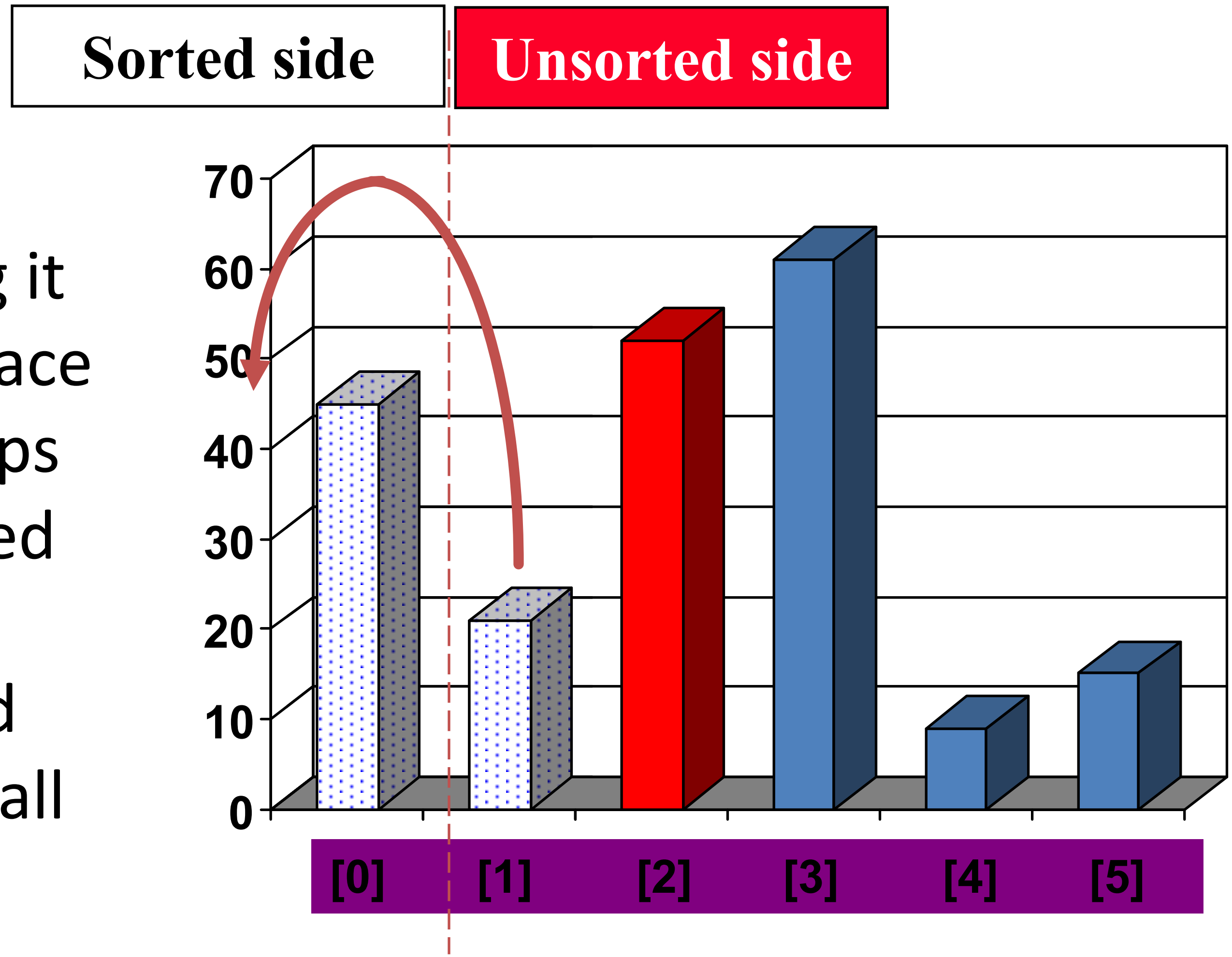
The Insertionsort Algorithm

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



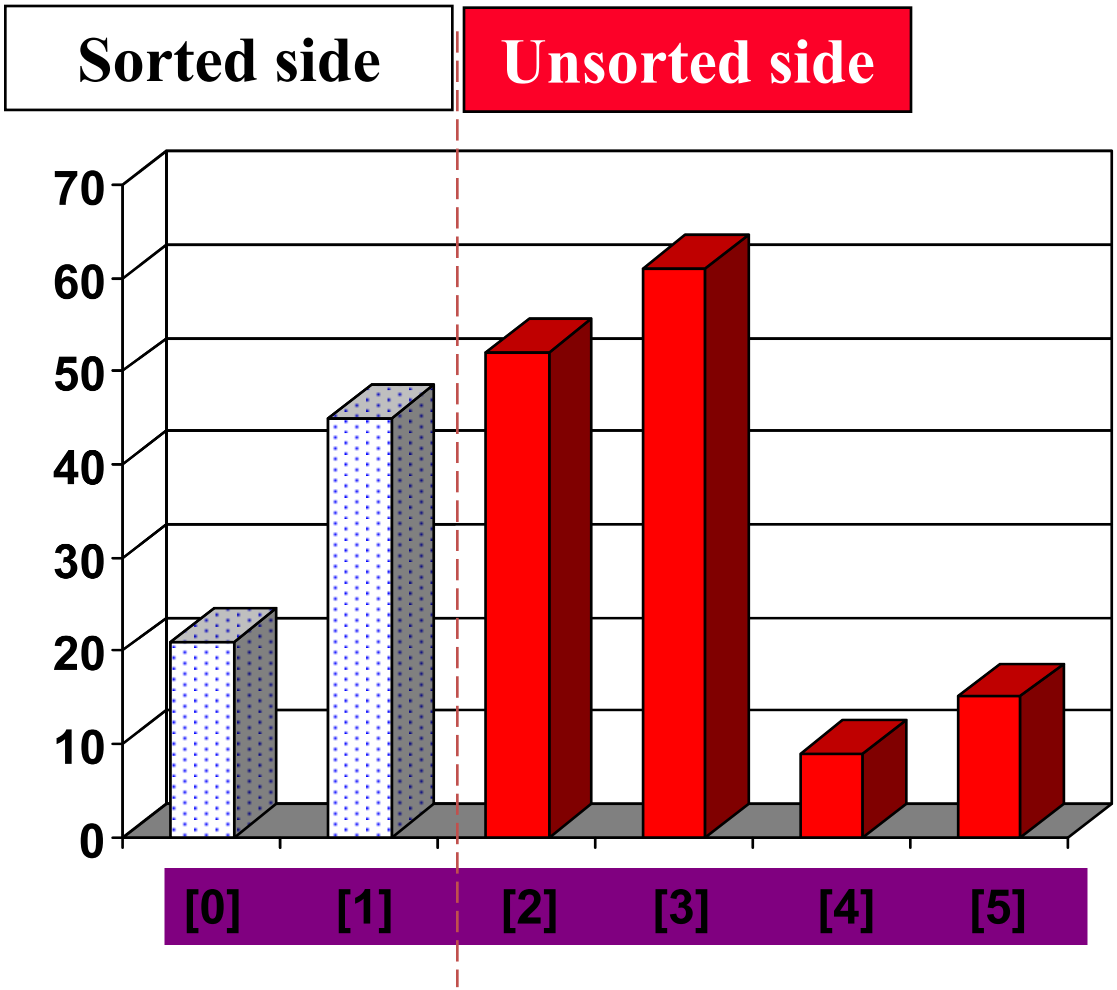
The Insertionsort Algorithm

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



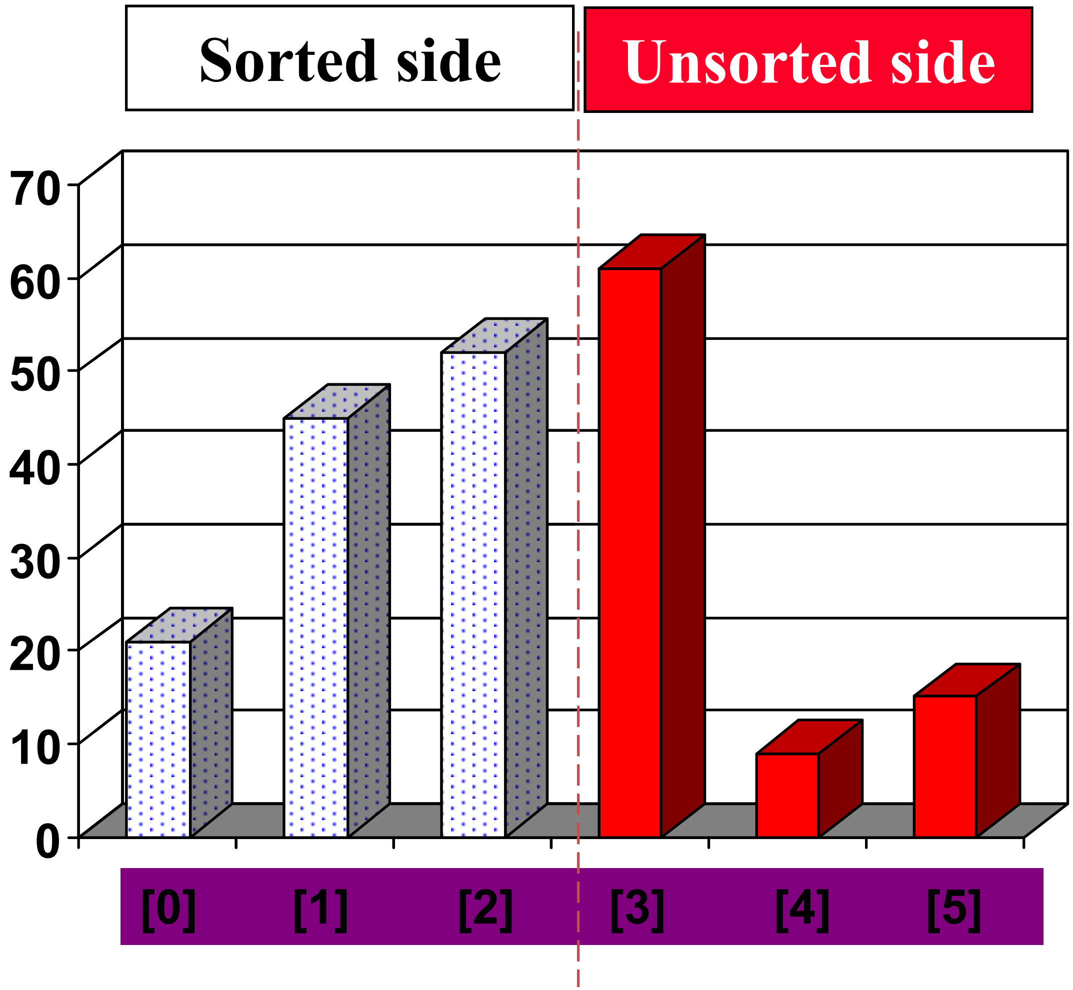
The Insertionsort Algorithm

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



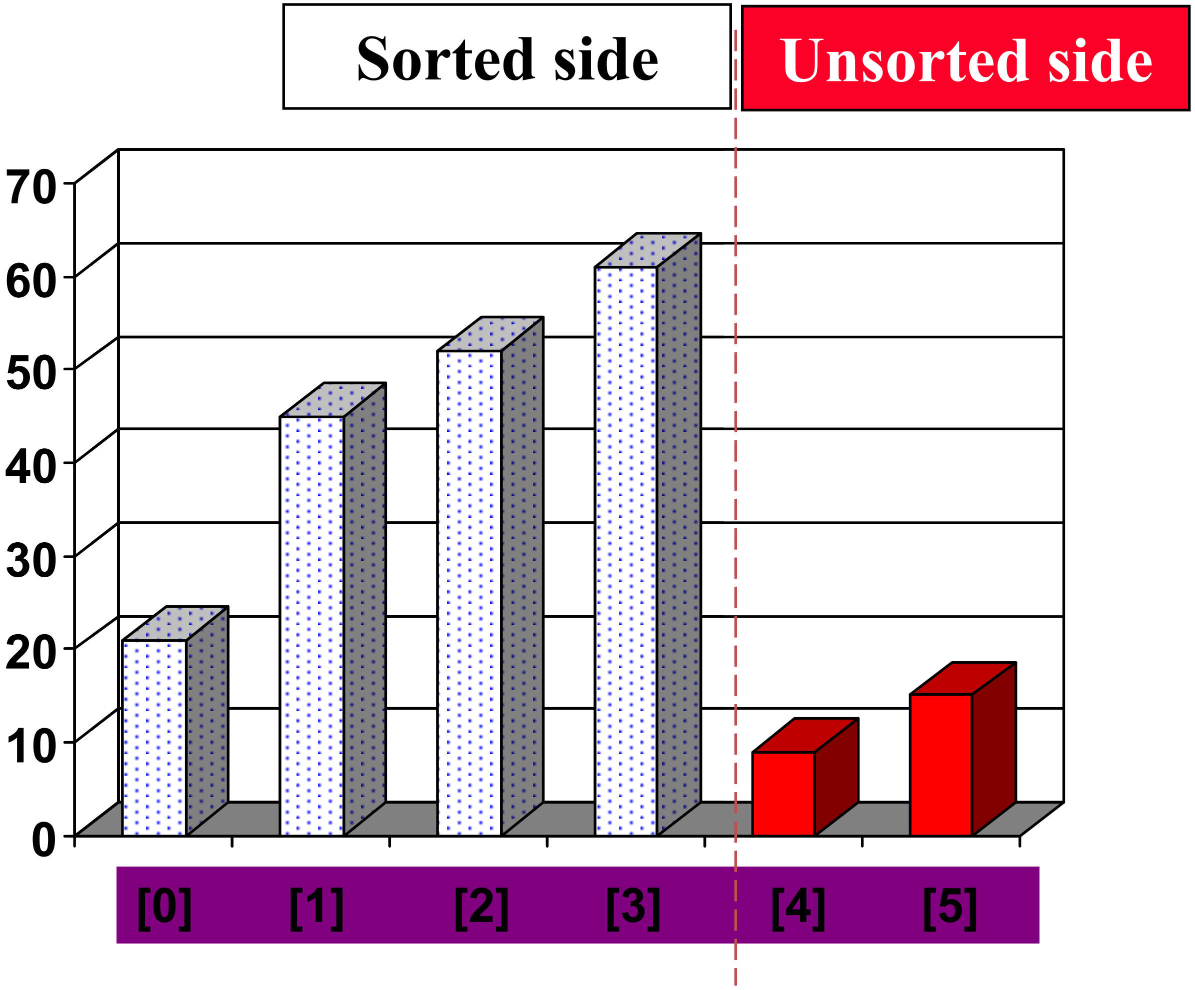
The Insertionsort Algorithm

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



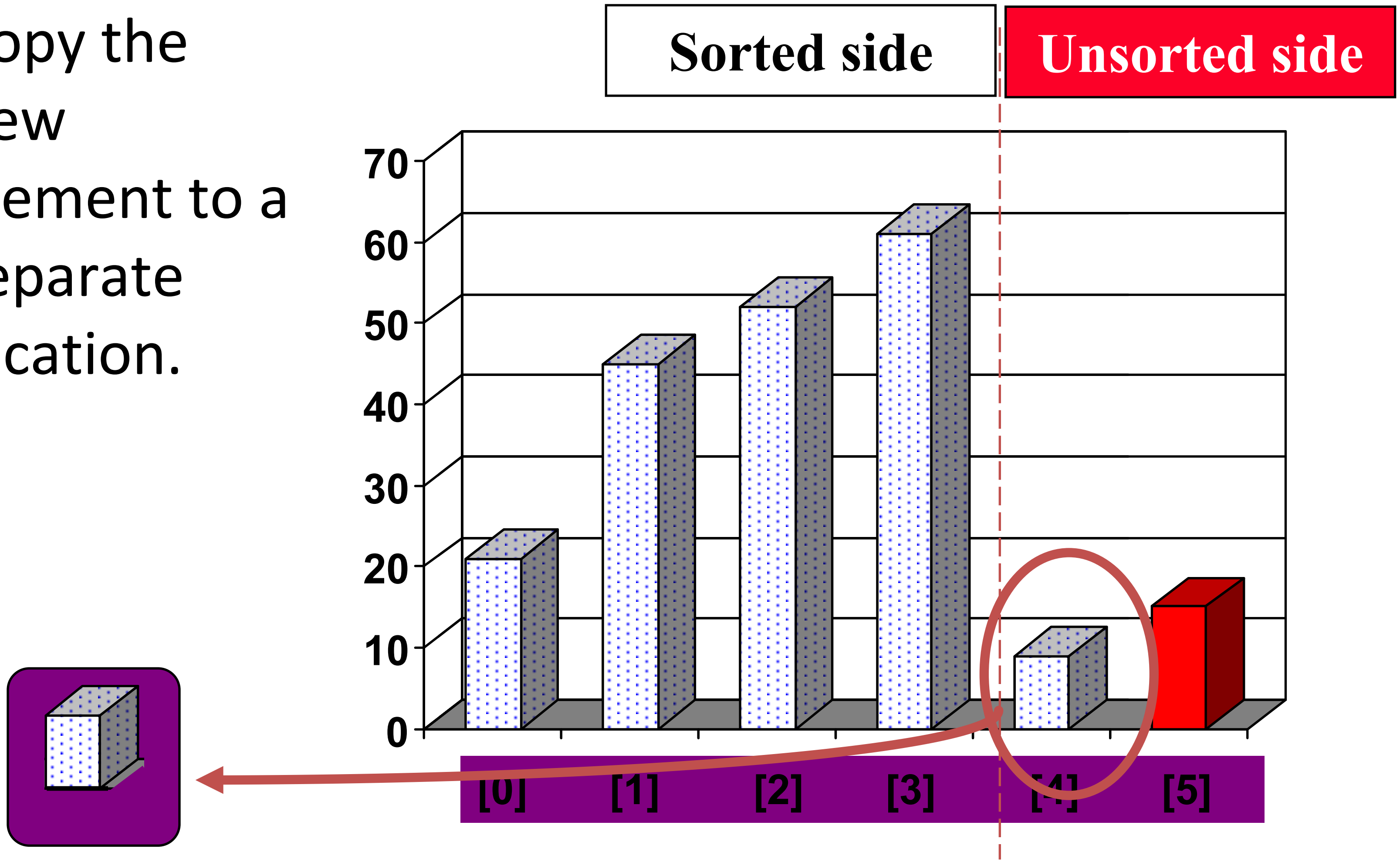
The Insertionsort Algorithm

- Sometimes we are lucky twice in a row.



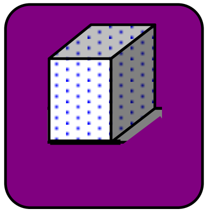
How to Insert One Element

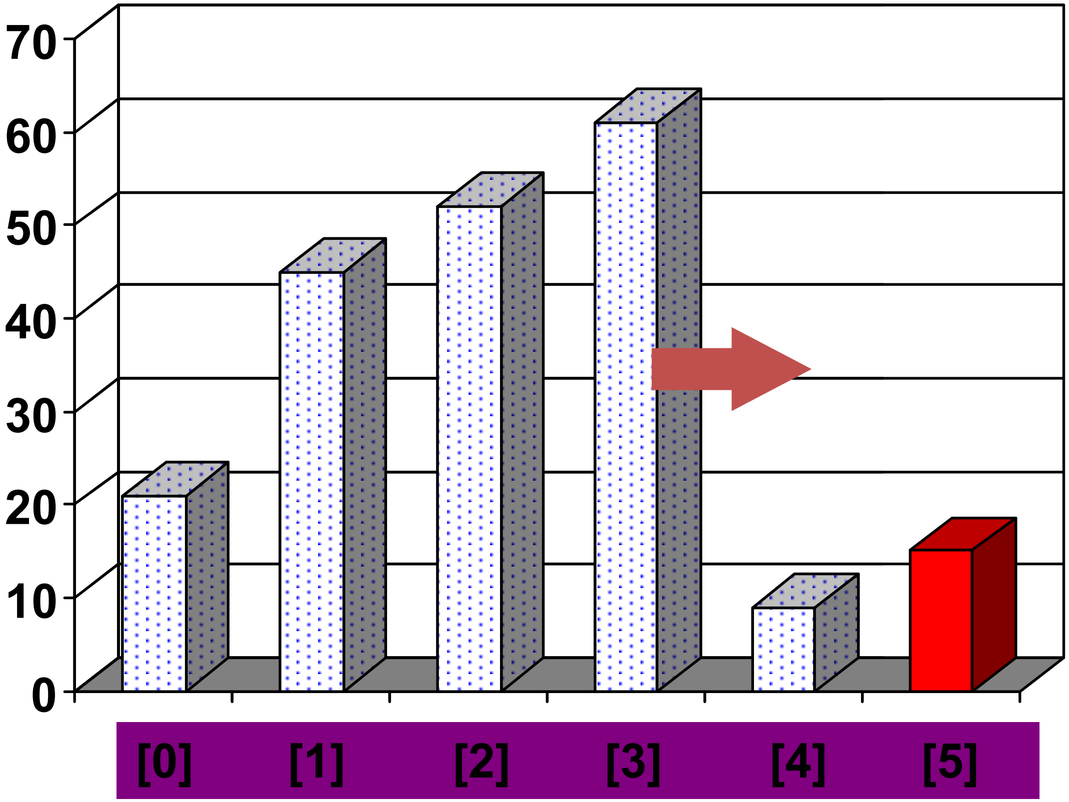
- Copy the new element to a separate location.



How to Insert One Element

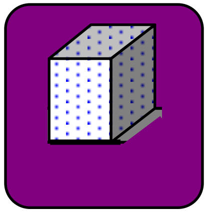
- Shift elements in the sorted side, creating an open space for the new element.

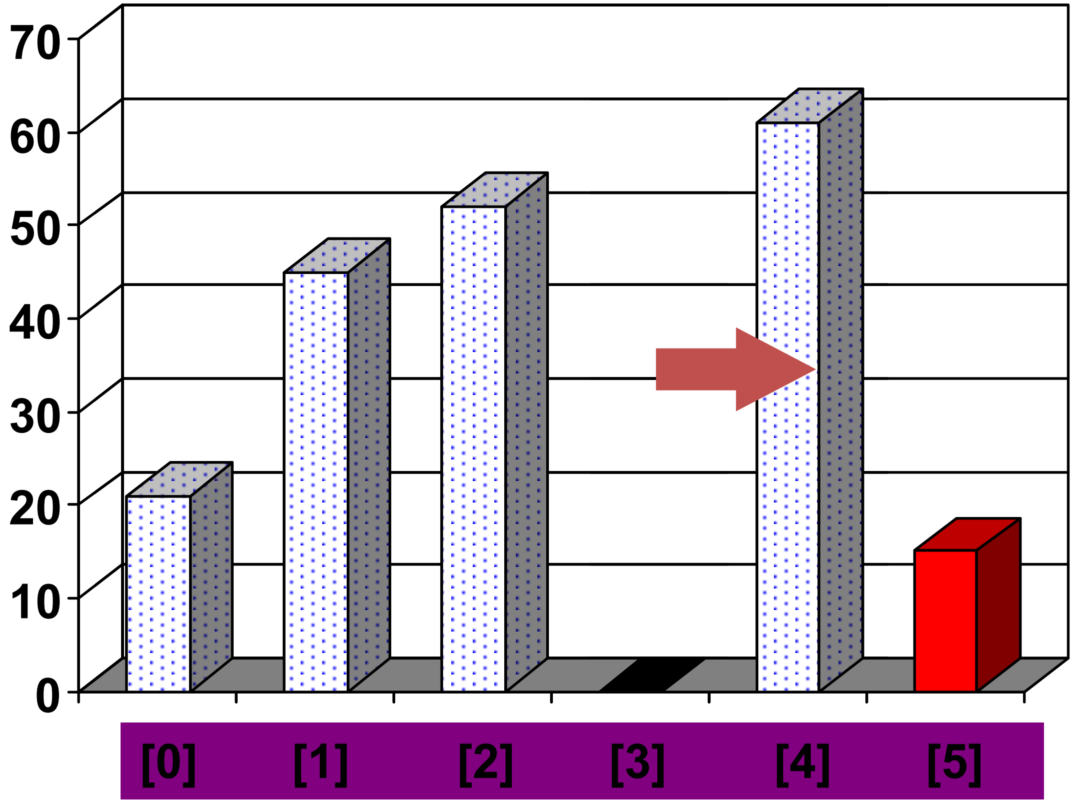




How to Insert One Element

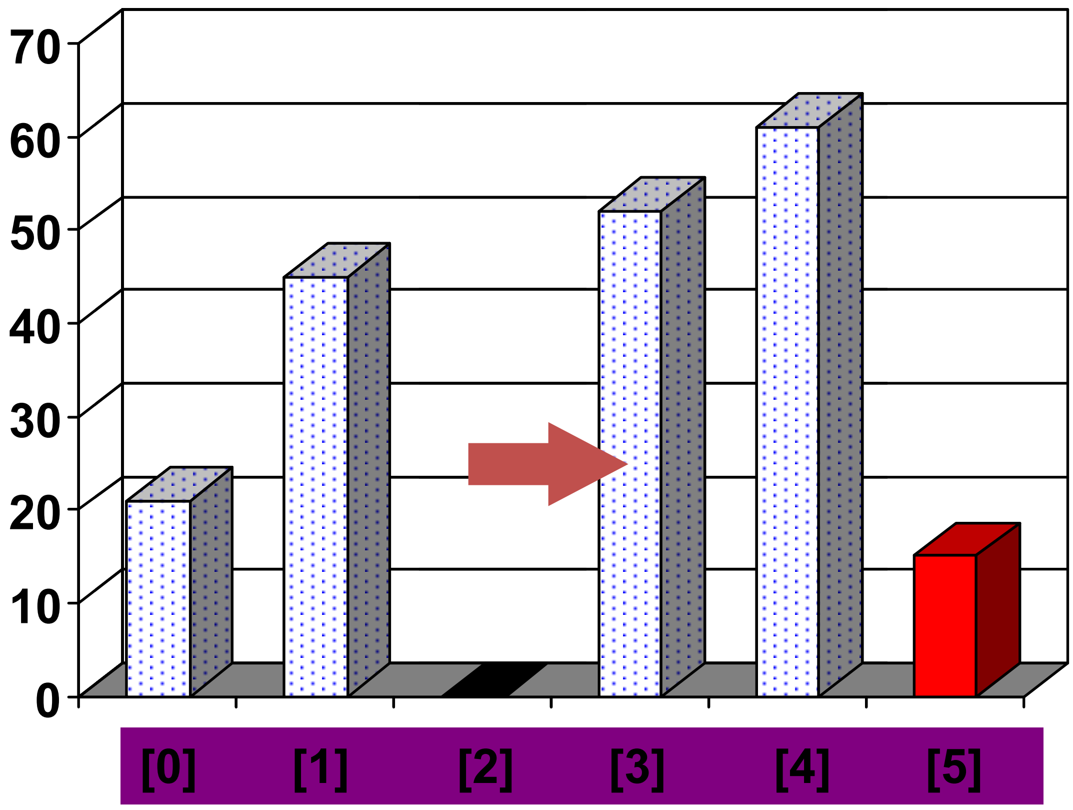
- Shift elements in the sorted side, creating an open space for the new element.

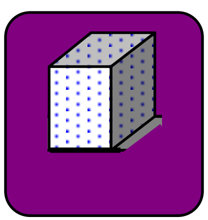




How to Insert One Element

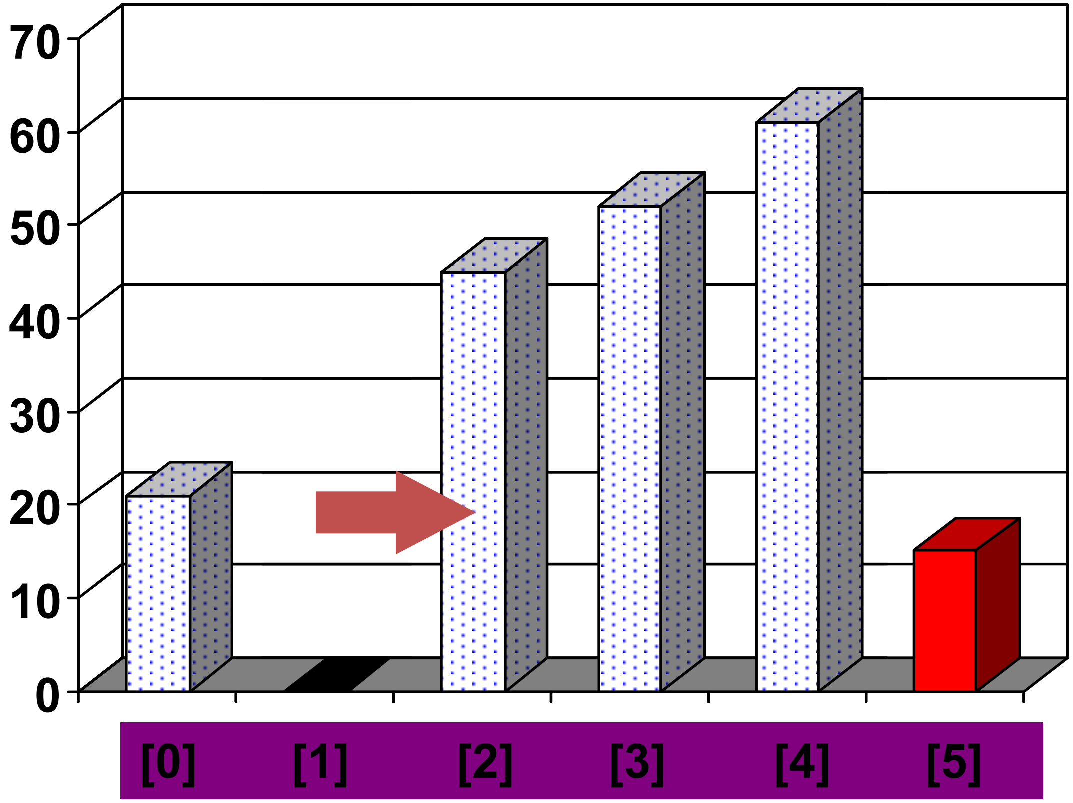
- Continue shifting elements...

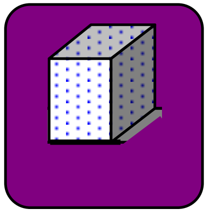




How to Insert One Element

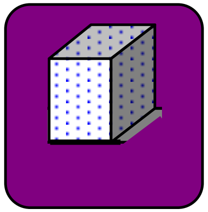
- Continue shifting elements...

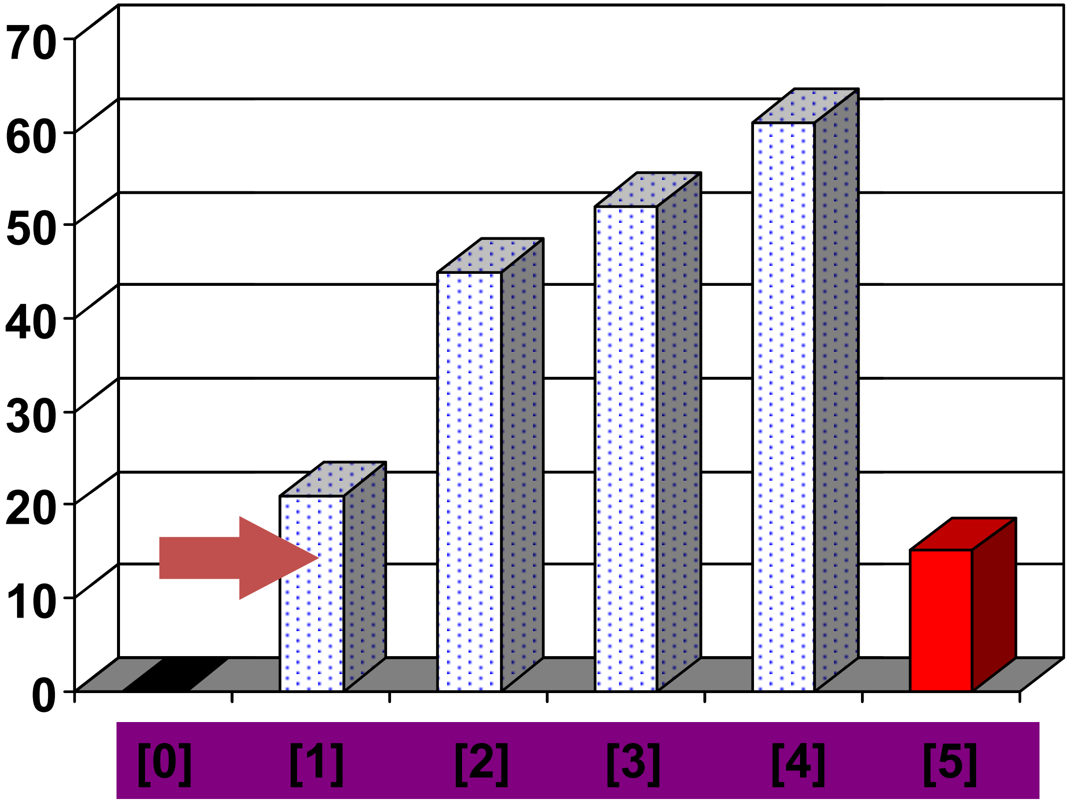




How to Insert One Element

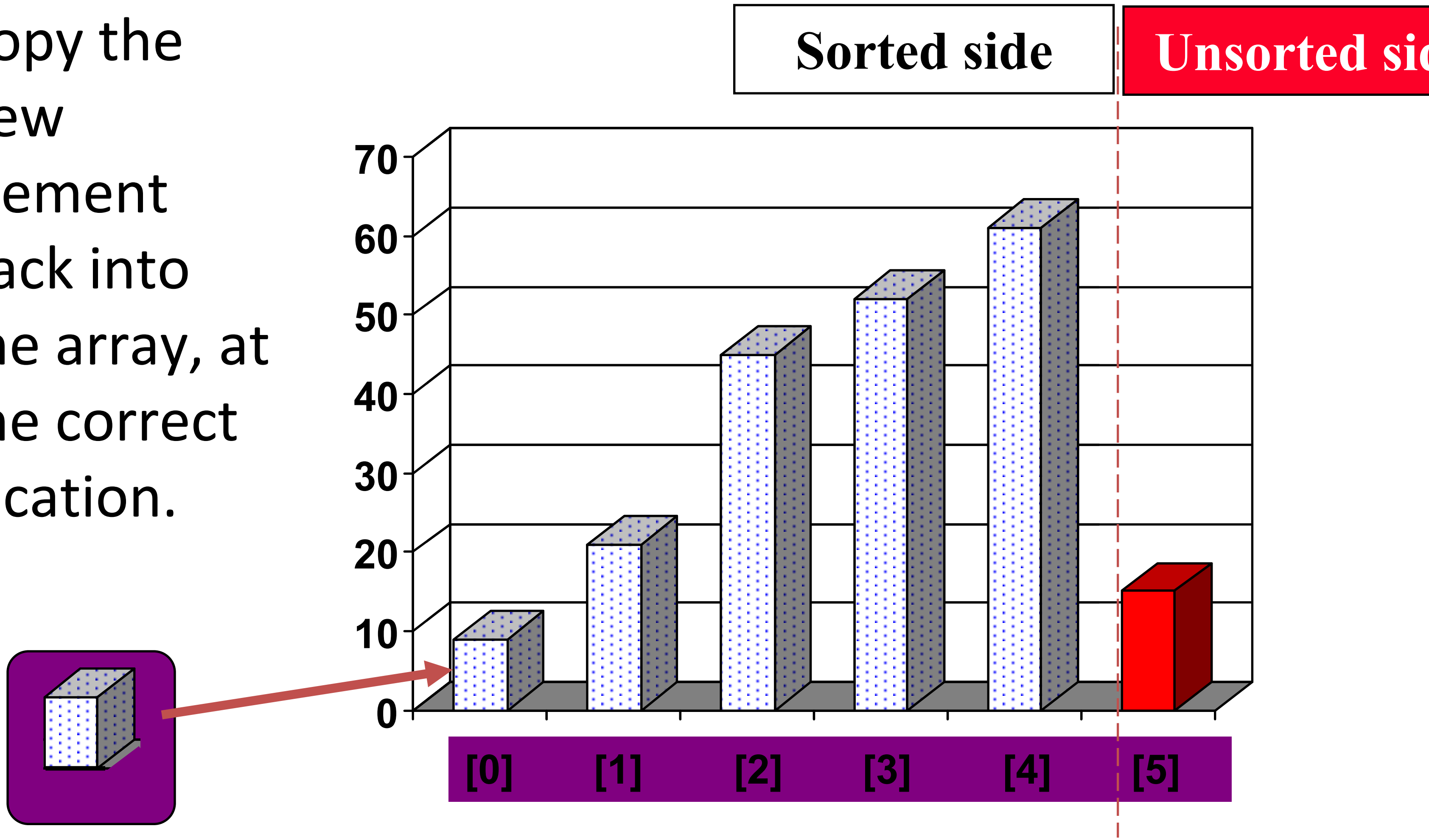
- ...until you reach the location for the new element.





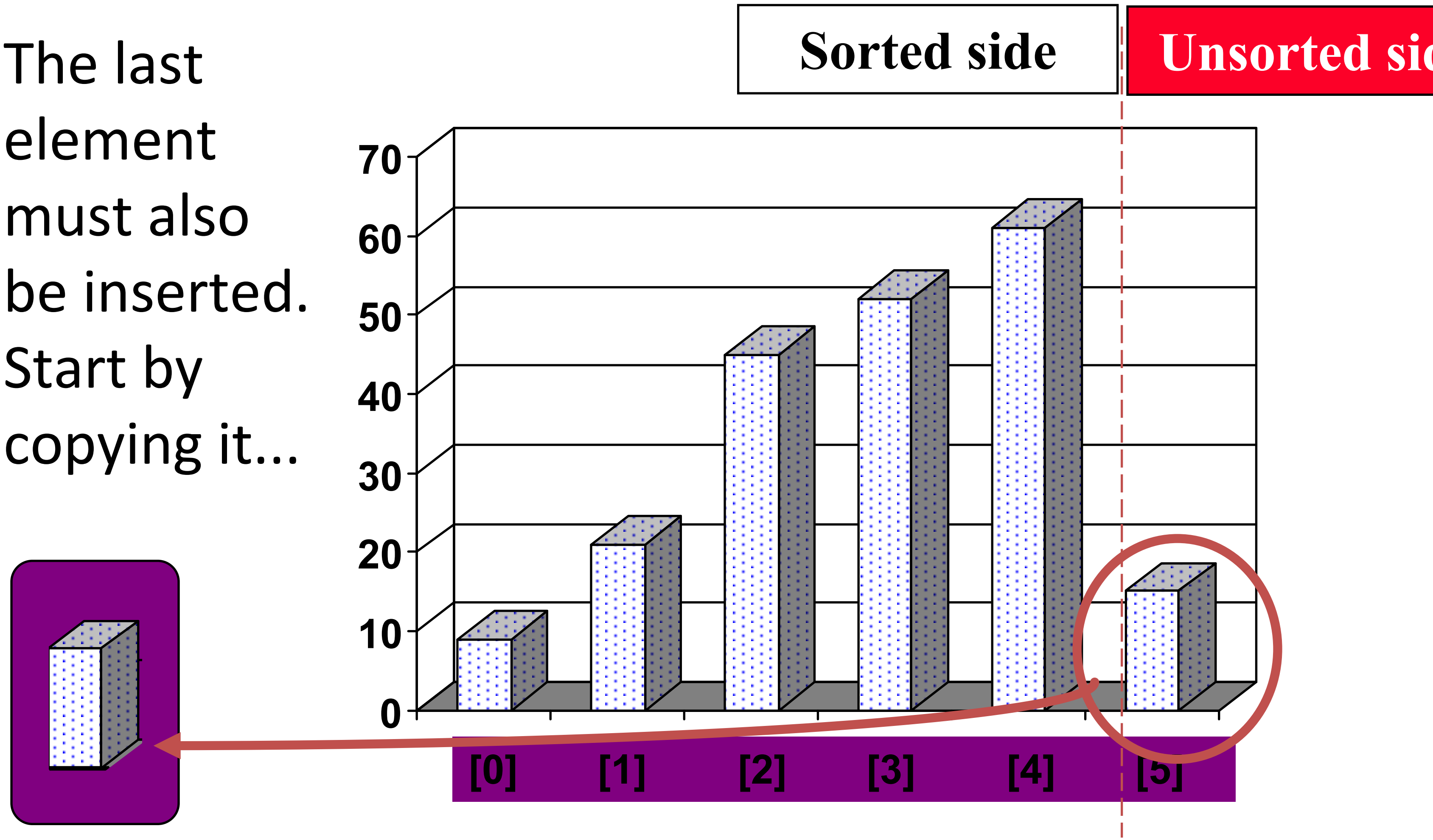
How to Insert One Element

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



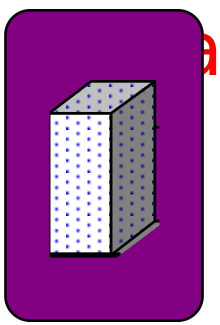
How to Insert One Element

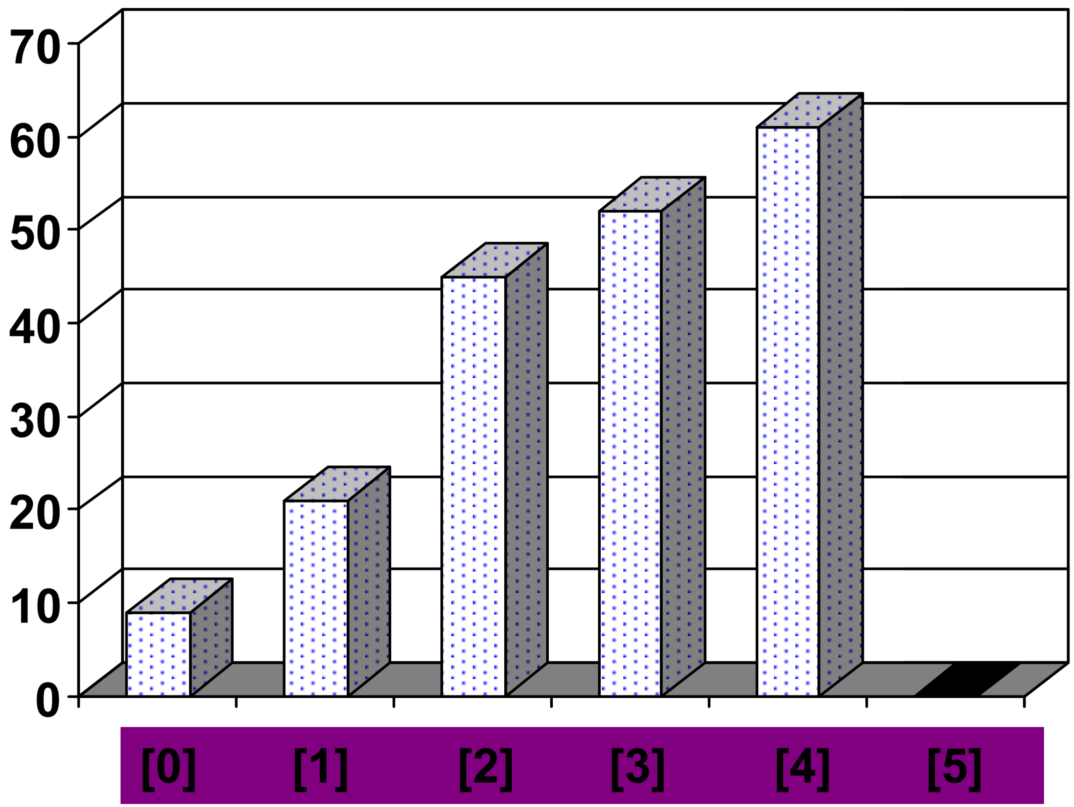
- The last element must also be inserted. Start by copying it...



A Quiz

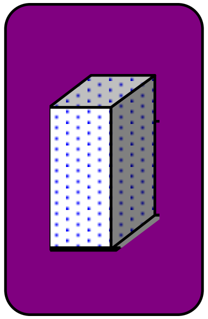
How many shifts will occur before we copy this element back into th ay?

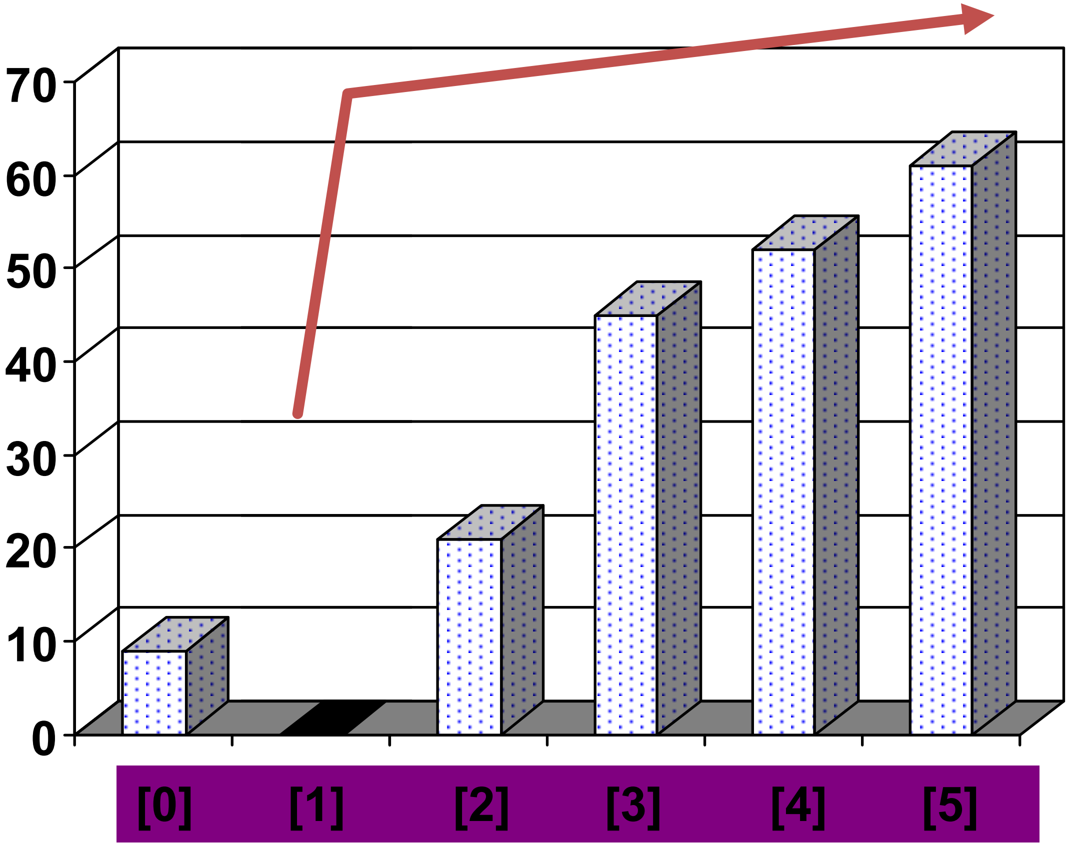




A Quiz

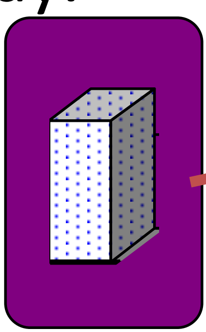
- Four items are shifted.

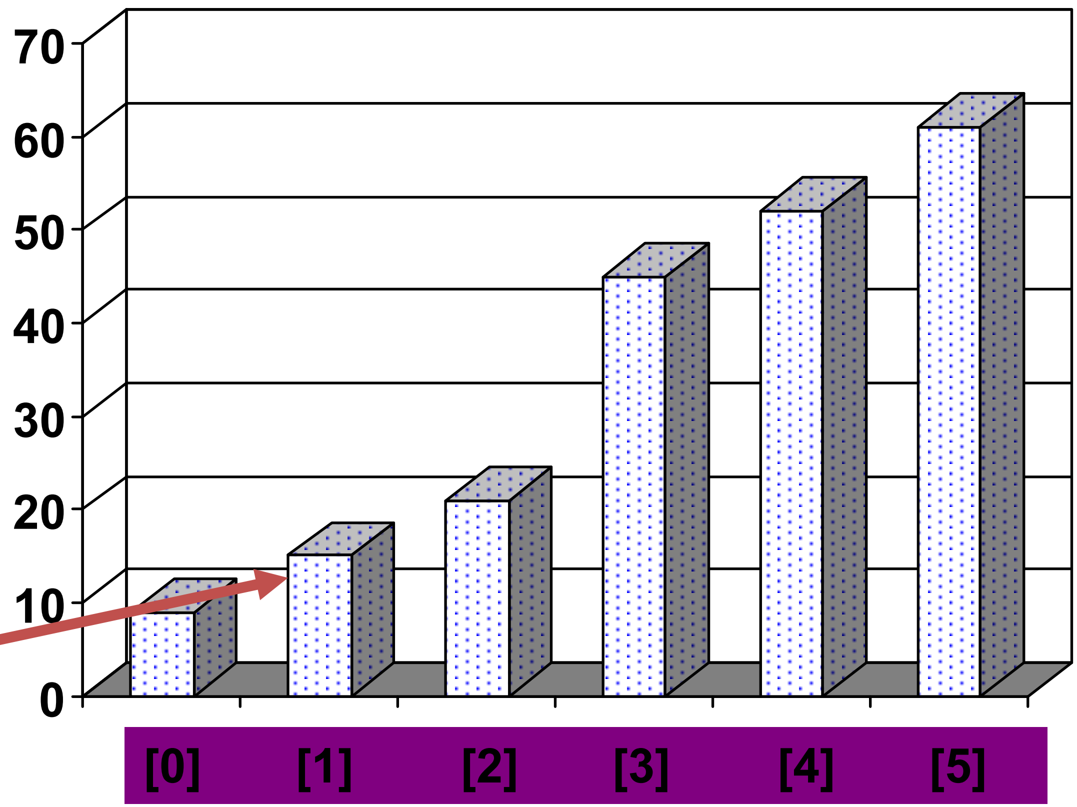




A Quiz

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





SelectionSort Code

```
void selectionsort(int data[ ], size_t n)
{
    size_t i, j, index_of_largest;
    int largest;
    if (n == 0)
      return; // No work for an empty array.
    for (i = n-1; i > 0; --i)  {
        largest = data[0];
        index_of_largest = 0;
        for (j = 1; j <= i; ++j)   {
            if (data[j] > largest)   {
                largest = data[j];
                index_of_largest = j;
            }
        }
        swap(data[i], data[index_of_largest]);
    }
}
```

Timing and Other Issues

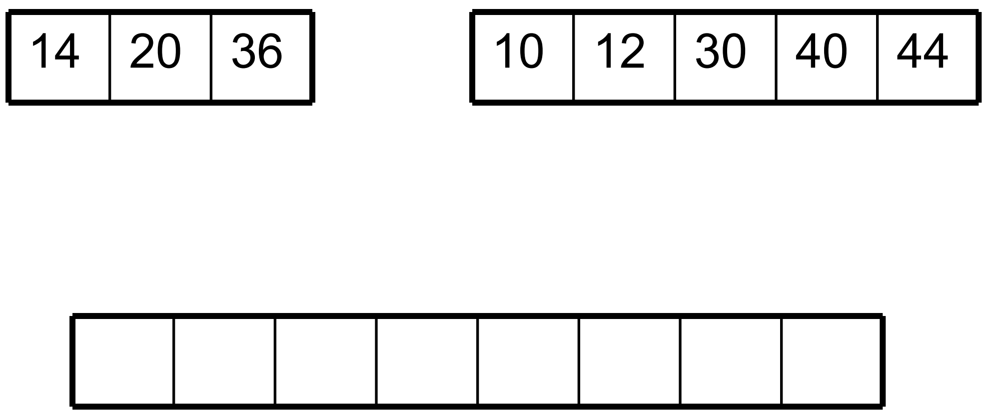
- Both Selectionsort and Insertionsort have a worst-case time of $O(n^2)$, making them impractical for large arrays.
- But they are easy to program, easy to debug.
- Insertionsort also has good performance when the array is nearly sorted to begin with.
- But more sophisticated sorting algorithms are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

Mergesort

- Mergesort:
  a divide and conquer algorithm for sorting
- Small cases:
  a single element array, which is already sorted
- Large cases:
  – split the array into two “equal” size parts
  – recursively sort the two parts
  – merge the two sorted subarrays into a sorted array
- The key to mergesort is the merge algorithm

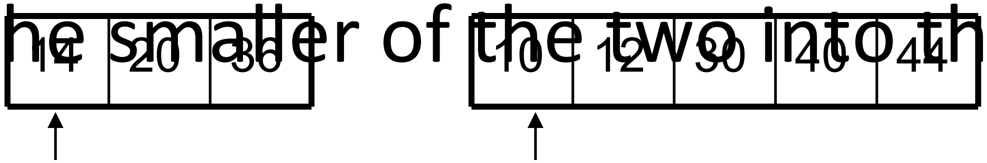
Mergesort

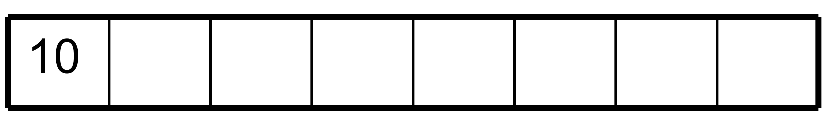
- Merging two sorted arrays into an output array



Mergesort

- Merging two sorted arrays into an output array
- Start by examining the first elements of each array
- Copy the smaller of the two into the output array





Mergesort

- Merging two sorted arrays into an output array
- Start by examining the first elements of each array
- Copy the smaller of the two into the output array
- Move to the right in the array from which the value was copied

14	20	36

10	12	30	40	44

10							

Mergesort

- Merging two sorted arrays into an output array
- Start by examining the first elements of each array
- Copy the smaller of the two into the output array
- Move to the right in the array from which the value was copied
- Copy the smaller of the two values into the output array

14	20	36

10	12	30	40	44

10	12						

Mergesort

- Merging two sorted arrays into an output array
- Start by examining the first elements of each array
- Copy the smaller of the two into the output array

14	20	36

10	12	30	40	44

- Move to the right in the array from which the value was copied

10	12						

Mergesort

- Merging two sorted arrays into an output array
- Start by examining the first elements of each array
- Copy the smaller of the two into the output array

14	20	36

10	12	30	40	44

- Move to the right in the array from which the value was copied
- Copy the smaller of the two values into the output array

10	12	14					

Mergesort

- Merging two sorted arrays into an output array
- Start by examining the first elements of each array
- Copy the smaller of the two into the output array

14	20	36

10	12	30	40	44

- Move to the right in the array from which the value was copied

10	12	14					

Mergesort

- Merging two sorted arrays into an output array
- Start by examining the first elements of each array
- Copy the smaller of the two into the output array

14	20	36

10	12	30	40	44

- Move to the right in the array from which the value was copied
- Copy the smaller of the two values into the output array

10	12	14	20				

Mergesort

- Merging two sorted arrays into an output array
- Start by examining the first elements of each array
- Copy the smaller of the two into the output array
- Move to the right in the array from which the value was copied

14	20	36

10	12	30	40	44

10	12	14	20				

Mergesort

- Merging two sorted arrays into an output array
- Start by examining the first elements of each array
- Copy the smaller of the two into the output array

14	20	36

10	12	30	40	44

- Move to the right in the array from which the value was copied
- Copy the smaller of the two values into the output array

10	12	14	20	30			

Mergesort

- Merging two sorted arrays into an output array
- Start by examining the first elements of each array
- Copy the smaller of the two into the output array
- Move to the right in the array from which the value was copied

14	20	36

10	12	30	40	44

10	12	14	20	30			

Mergesort

- Merging two sorted arrays into an output array
- Start by examining the first elements of each array
- Copy the smaller of the two into the output array

14	20	36

10	12	30	40	44

- Move to the right in the array from which the value was copied
- Copy the smaller of the two values into the output array

10	12	14	20	30	36		

Mergesort

- Merging two sorted arrays into an output array
- Start by examining the first elements of each array
- Copy the smaller of the two into the output array

14	20	36

10	12	30	40	44

- Move to the right in the array from which the value was copied
- If this is not possible, copy the remaining values from the other array into the output

10	12	14	20	30	36	40	

Mergesort

- Merging two sorted arrays into an output array
- Start by examining the first elements of each array
- Copy the smaller of the two into the output array

14	20	36

10	12	30	40	44

- Move to the right in the array from which the value was copied
- If this is not possible, copy the remaining values from the other array into the output

10	12	14	20	30	36	40	44

Mergesort Code

```
void mergesort(int data[ ], size_t n)
{
   size_t  n1; // Size of the first subarray
   size_t  n2; // Size of the second subarray

   if (n > 1)
   {
      // Compute sizes of the subarrays.
      n1 = n / 2;
      n2 = n - n1;

      mergesort(data, n1);         // Sort from data[0] through data[n1-
   1]
      mergesort((data + n1), n2);  // Sort from data[n1] to the end

      // Merge the two sorted halves.
```

Merge Code

```
void merge(int data[ ], size_t n1, size_t n2)
{
   int *temp;          // Points to dynamic array to hold the sorted
   elements
   size_t copied  = 0; // Number of elements copied from data to temp
   size_t copied1 = 0; // Number copied from the first half of data
   size_t copied2 = 0; // Number copied from the second half of data
   size_t i;           // Array index to copy from temp back into data

   // Allocate memory for the temporary dynamic array.
   temp = new int[n1+n2];
   // Merge elements, copying from two halves of data to the
   temporary array.
   while ((copied1 < n1) && (copied2 < n2)) {
      if (data[copied1] < (data + n1)[copied2])
         temp[copied++] = data[copied1++];        // Copy from first half
      else
         temp[copied++] = (data + n1)[copied2++]; // Copy from second
```

Merge Code

```
// Copy any remaining entries in the left and right subarrays.
   while (copied1 < n1)
      temp[copied++] = data[copied1++];
   while (copied2 < n2)
      temp[copied++] = (data+n1)[copied2++];
   // Copy from temp back to the data array, and release temp's memory.
   for (i = 0; i < n1+n2; i++)
      data[i] = temp[i];
   delete [ ] temp;
}
```

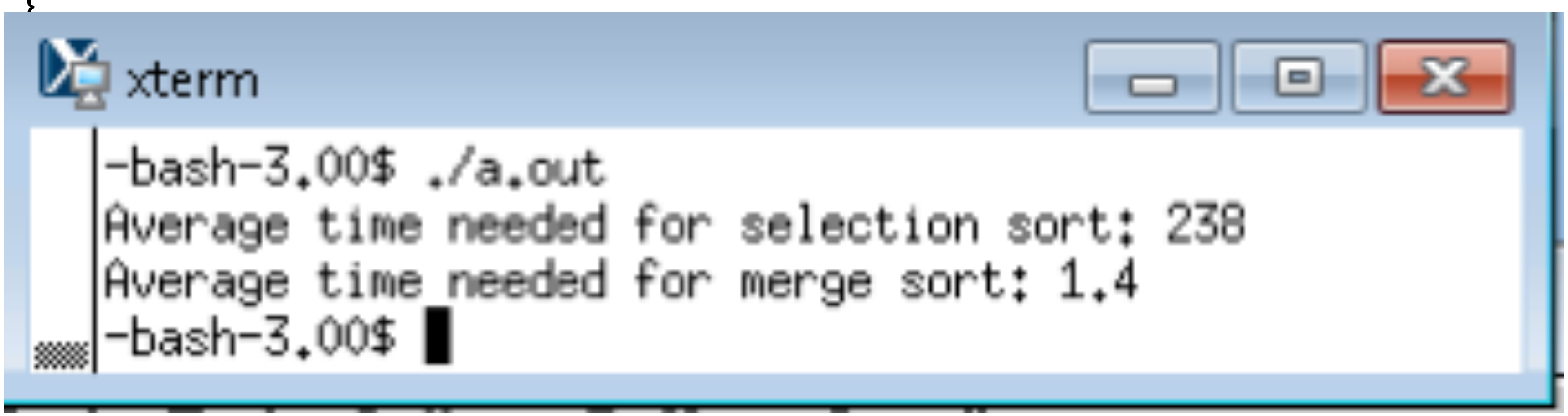
Timing Study

```
int main()
{
  int *data = NULL;
  double stotal = 0;            // Total time used in all runs of SelectionSort
  double mtotal = 0;            // Total time used in all runs of MergeSort
  for (int j = 0; j < 50; j++)
  {
    data = randomArray(aSize);   // aSize constant int , set to 10000
    clock_t beginSSort = clock(); // clock function found in <time.h>
    selectionsort(data, aSize);
    clock_t endSSort =clock();
    stotal += diffclock(endSSort,beginSSort); // difference in ms
    clock_t beginMSort = clock();
    mergesort(data, aSize);
    clock_t endMSort =clock();
    mtotal += diffclock(endMSort,beginMSort);
    delete [] data;
  }
```

Timing Study

```
  double s_avg = stotal/50.0;
  double m_avg = mtotal/50.0;

  cout << "Average time needed for selection sort: " << s_avg << endl;
  cout << "Average time needed for merge sort: " << m_avg << endl;
  return EXIT_SUCCESS;
}
```



Mergesort Asymptotic Runtime

- Running time of mergesort is ***O***(nlog n)