C++ Object-Oriented Prog.
Unit 6: Generic Programming

CHAPTER 24: BASIC ALGORITHM 1: GENERIC SORTING

DR. ERIC CHOU IEEE SENIOR MEMBER

Objectives:

Basic searching and sorting algorithms:

- Linear Search
- Binary Search
- Selection Sort
- Insertion Sort
- Bubble Sort
- Merge Sort

More advanced sorting algorithms will be introduced in C++ 3 course.

LECTURE 1

Overview

Introduction

Common problem: sort a list of values, starting from lowest to highest.

- List of exam scores
- Words of dictionary in alphabetical order
- Students names listed alphabetically
- Student records sorted by ID#

Generally, we are given a list of records that have *keys.* These keys are used to define an ordering of the items in the list.

C++ Implementation of Sorting

- Use C++ templates to implement a generic sorting function.
- This would allow use of the same function to sort items of any class.
- However, class to be sorted must provide the following overloaded operators:
  - Assignment: =
  - Ordering: >, <, ==
- Example class: C++ STL string class
- In this lecture, we'll talk about sorting integers; however, the algorithms are general and can be applied to any class as described above.

Quadratic Sorting Algorithms

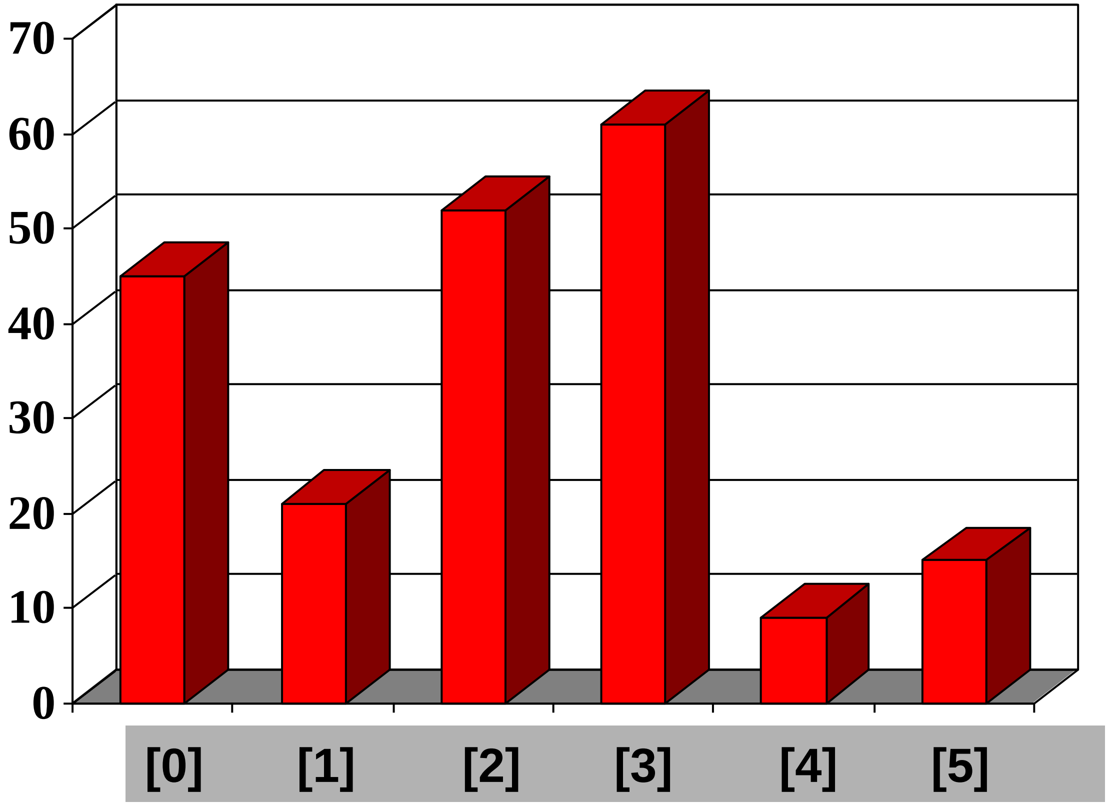
- We are given *n* records to sort.
- There are a number of simple sorting algorithms whose worst and average case performance is quadratic $O(n^2)$:
  - Selection sort
  - Insertion sort
  - Bubble sort

LECTURE 2

Selection Sort

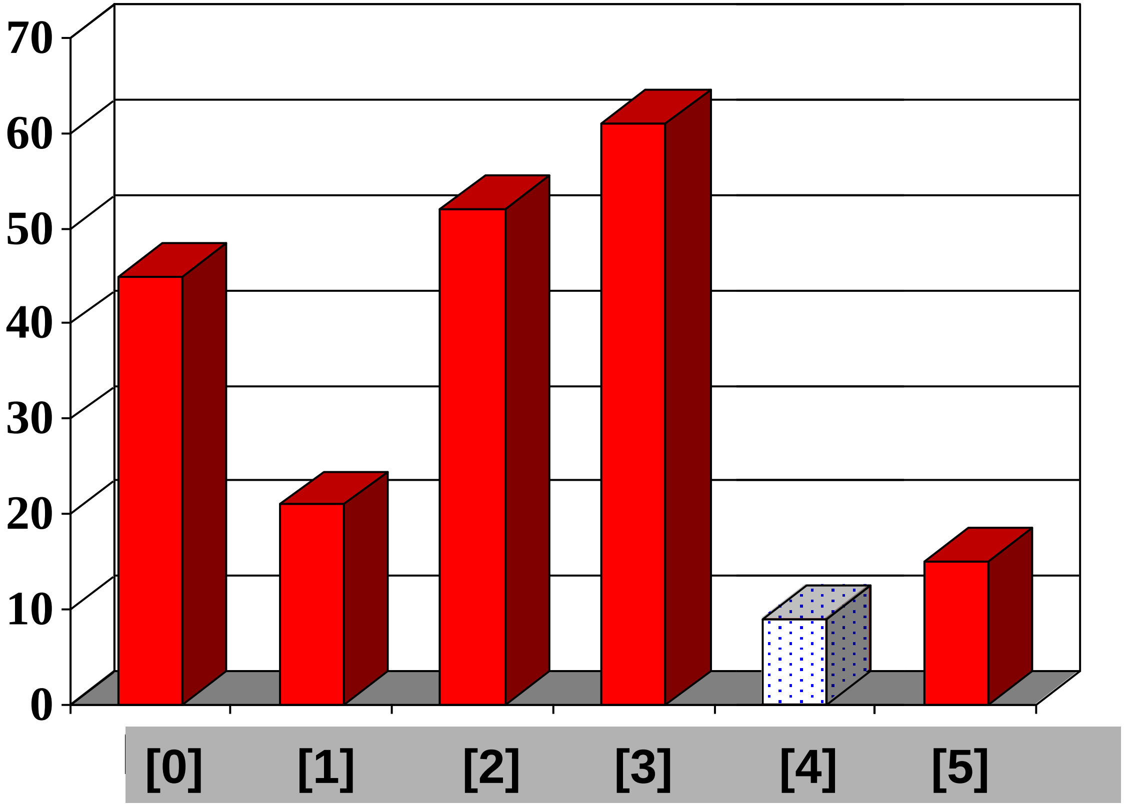
Sorting an Array of Integers

Example: we are given an array of six integers that we want to sort from smallest to largest



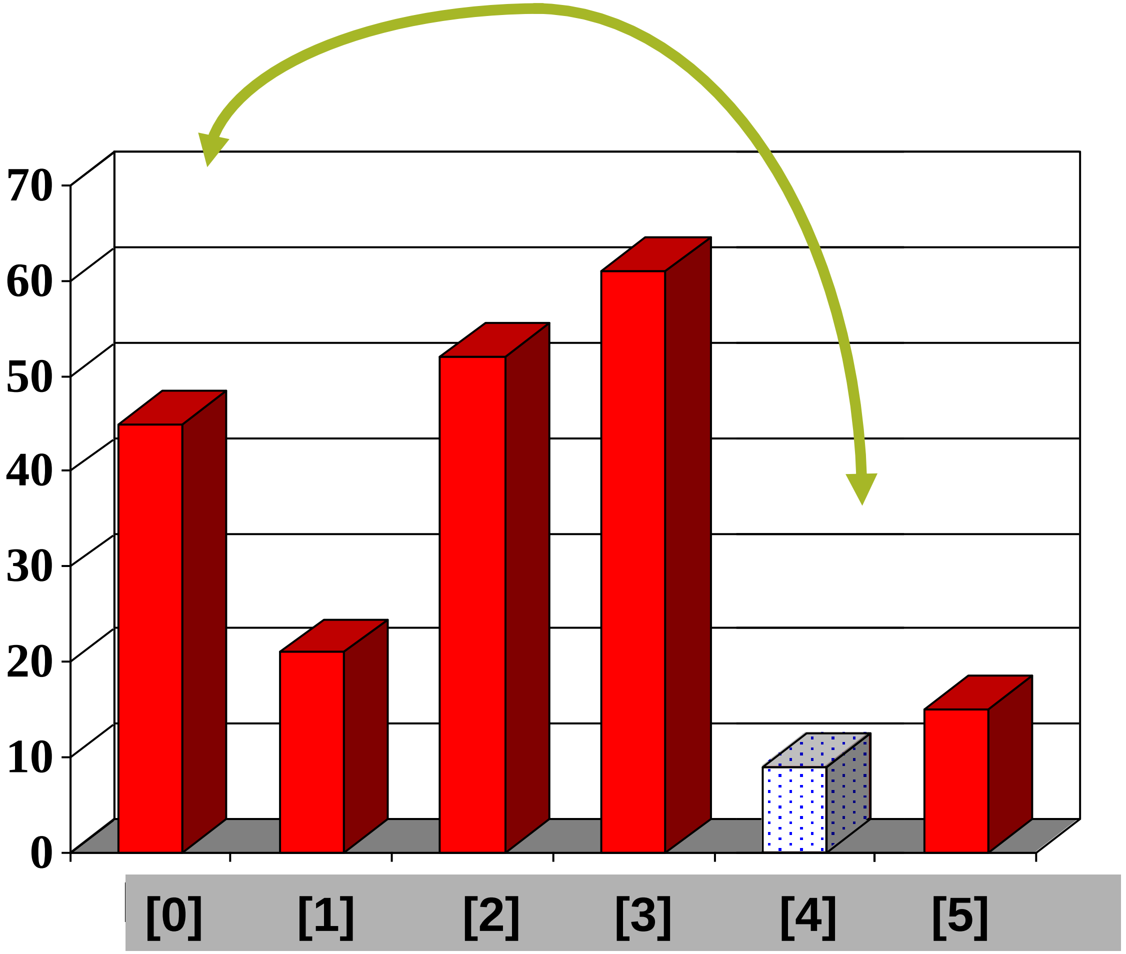
The Selection Sort Algorithm

Start by finding the **smallest** entry.



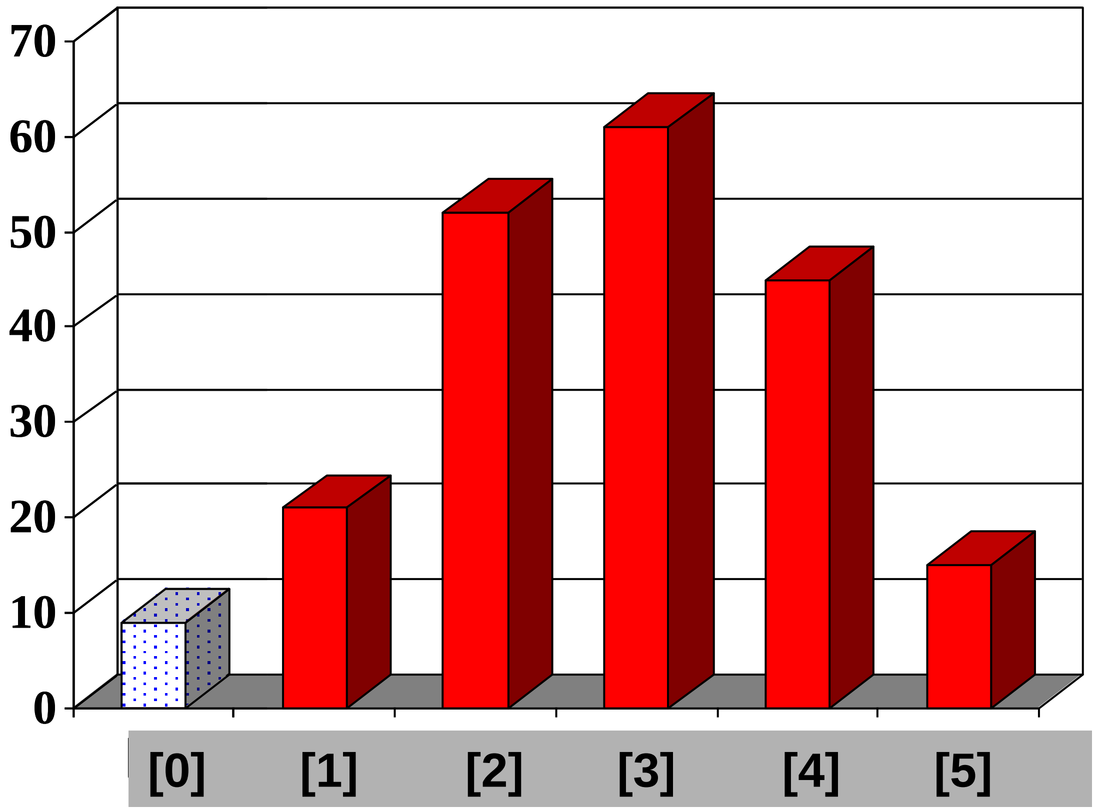
The Selection Sort Algorithm

Swap the smallest entry with the **first entry**.



The Selection Sort Algorithm

Swap the smallest entry with the **first entry**.

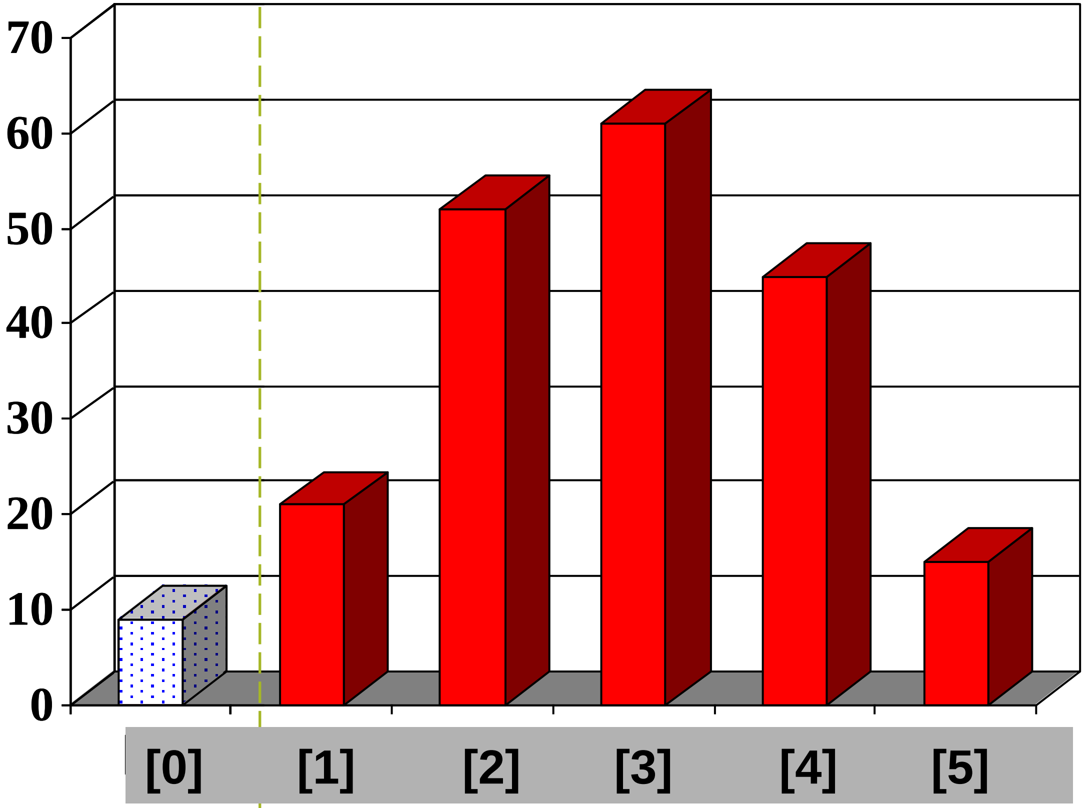


The Selection Sort Algorithm

Sorted side

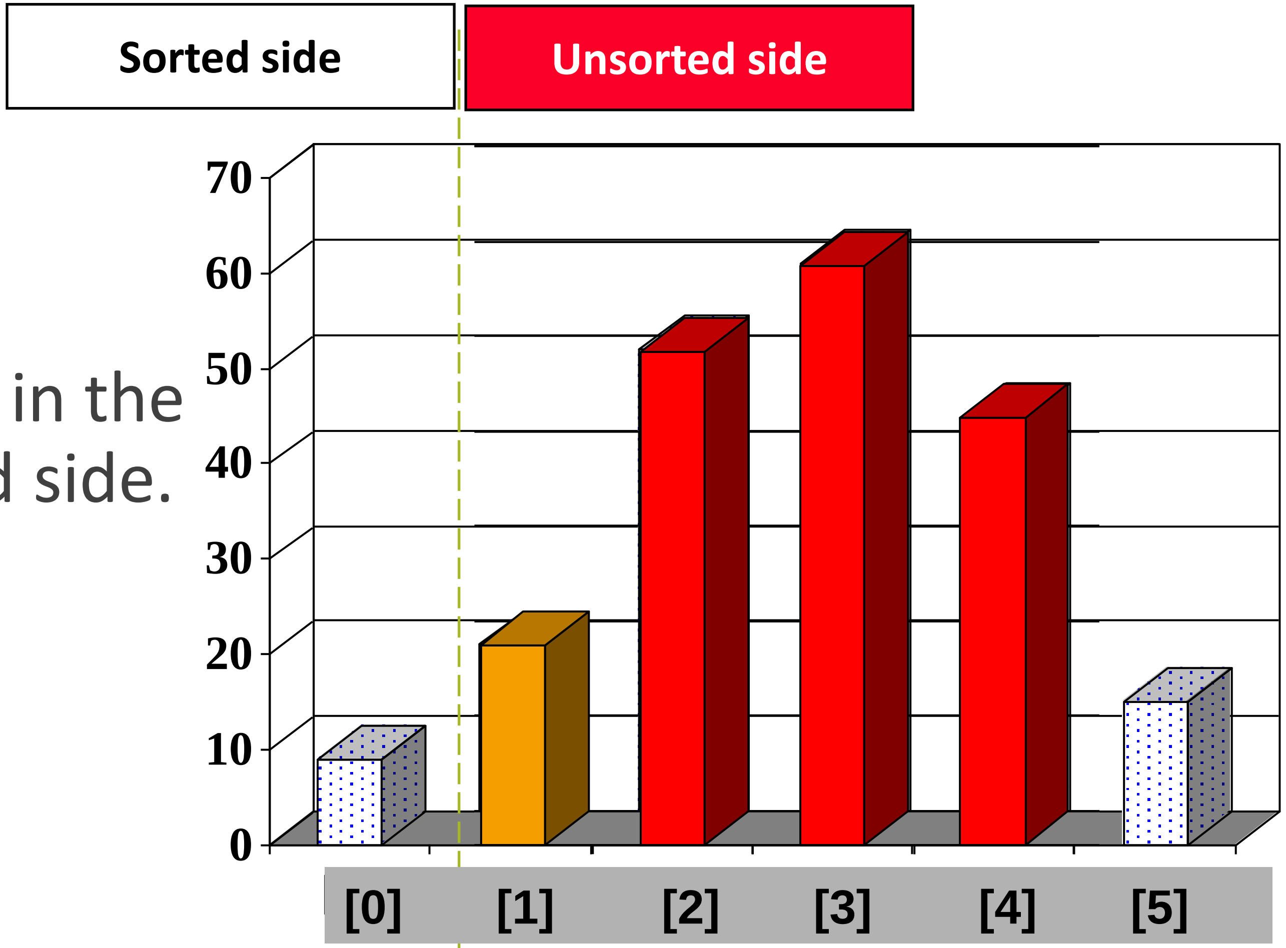
Unsorted side

Part of the array is now sorted.



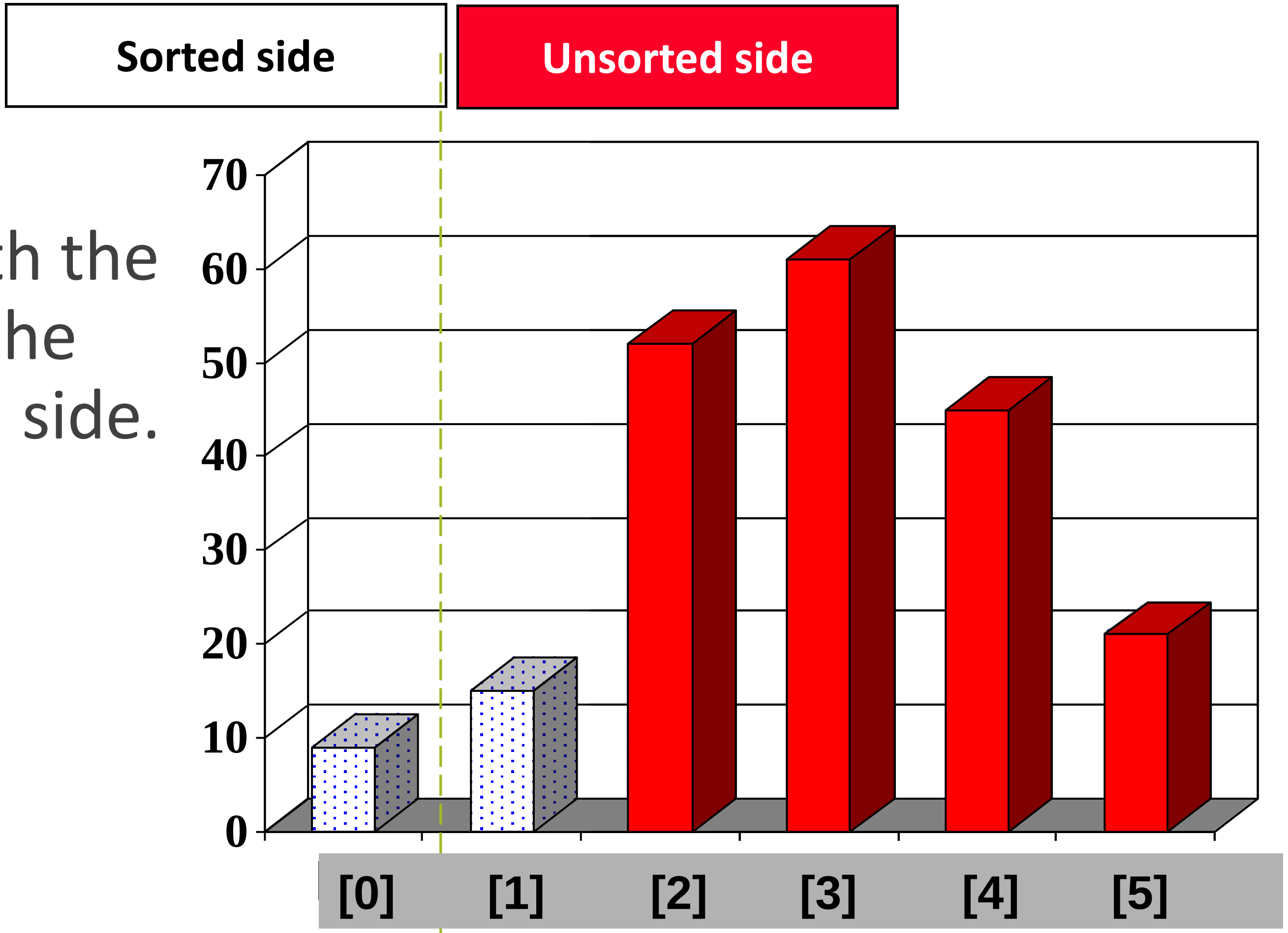
The Selection Sort Algorithm

Find the smallest element in the unsorted side.



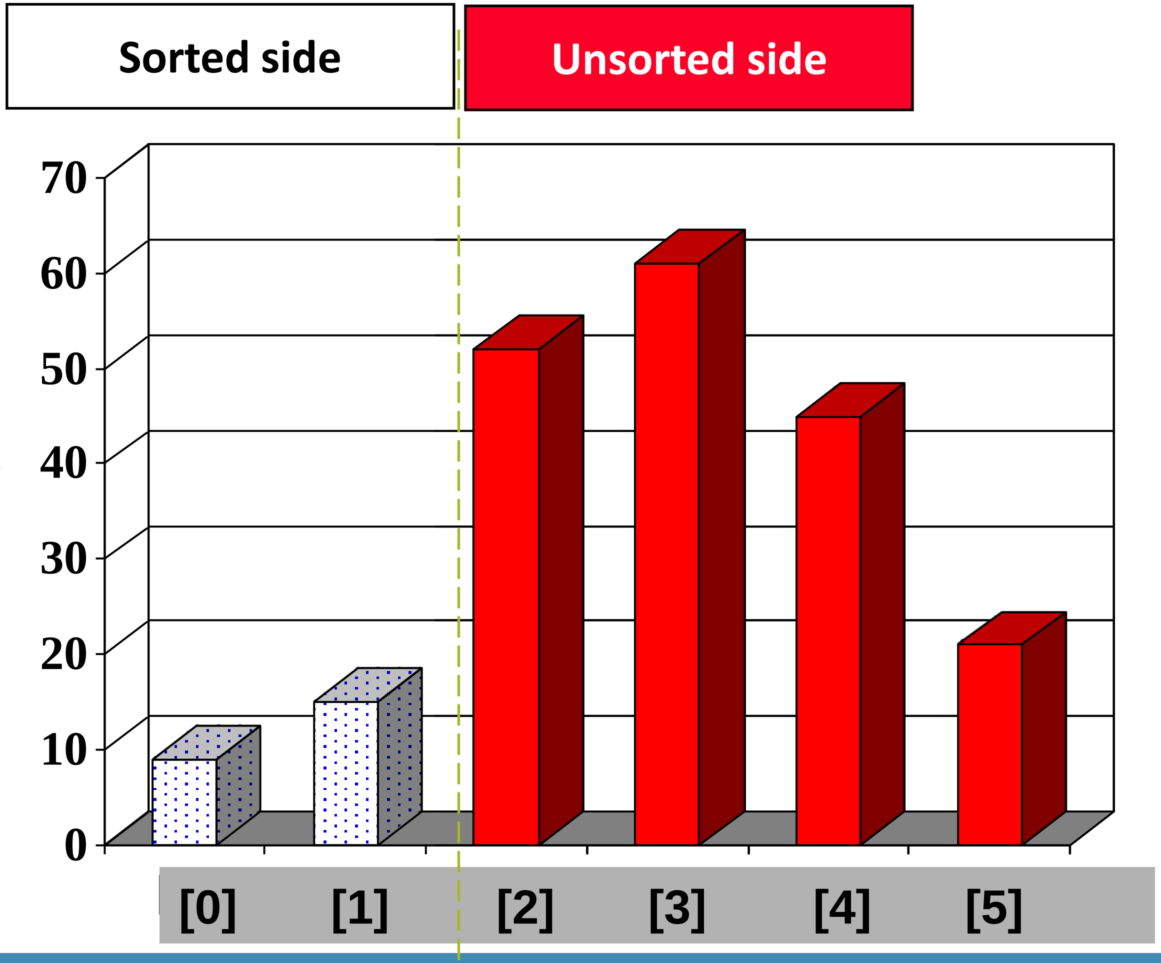
The Selection Sort Algorithm

Swap with the front of the unsorted side.



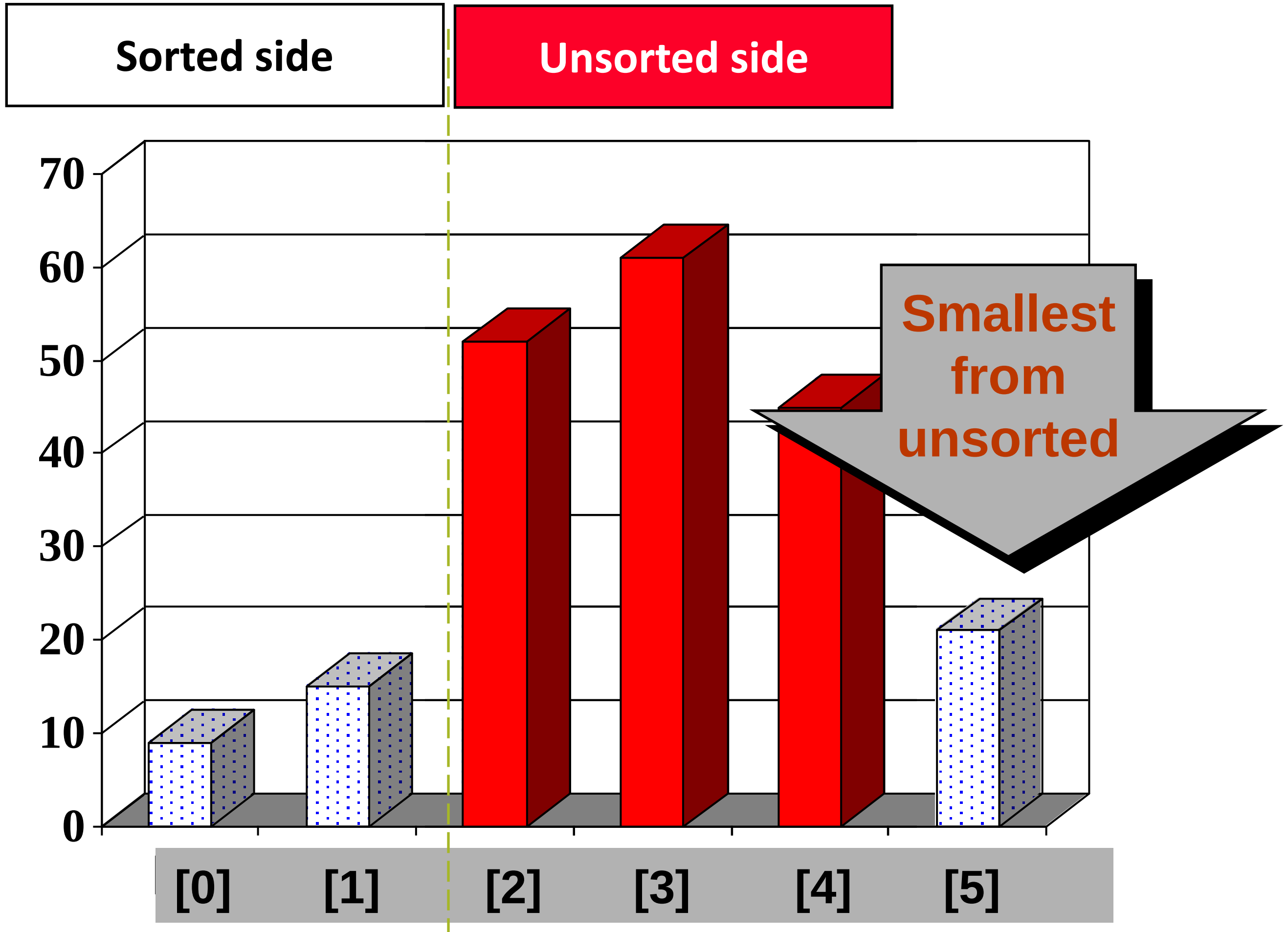
The Selection Sort Algorithm

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



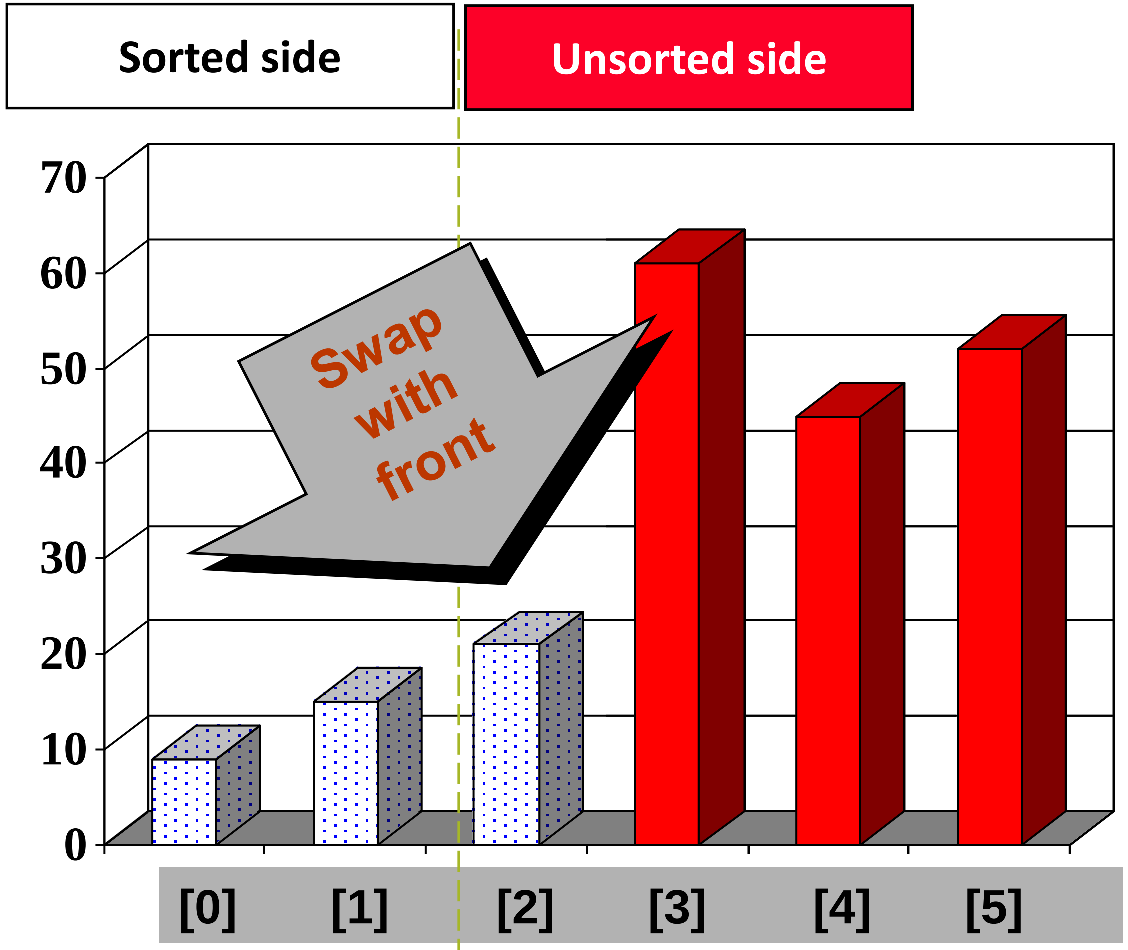
The Selection Sort Algorithm

The process continues...

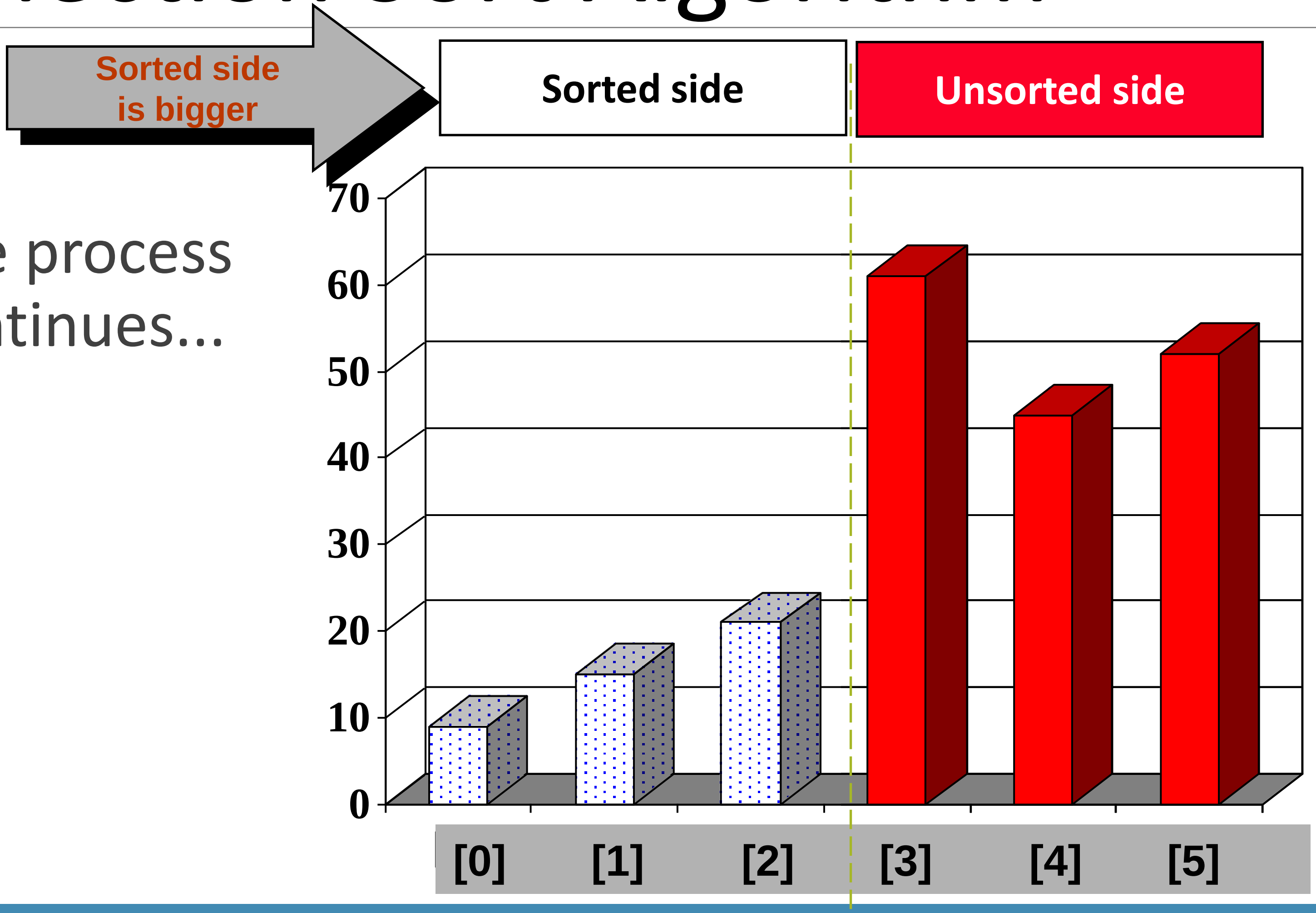


The Selection Sort Algorithm

The process continues...



The Selection Sort Algorithm

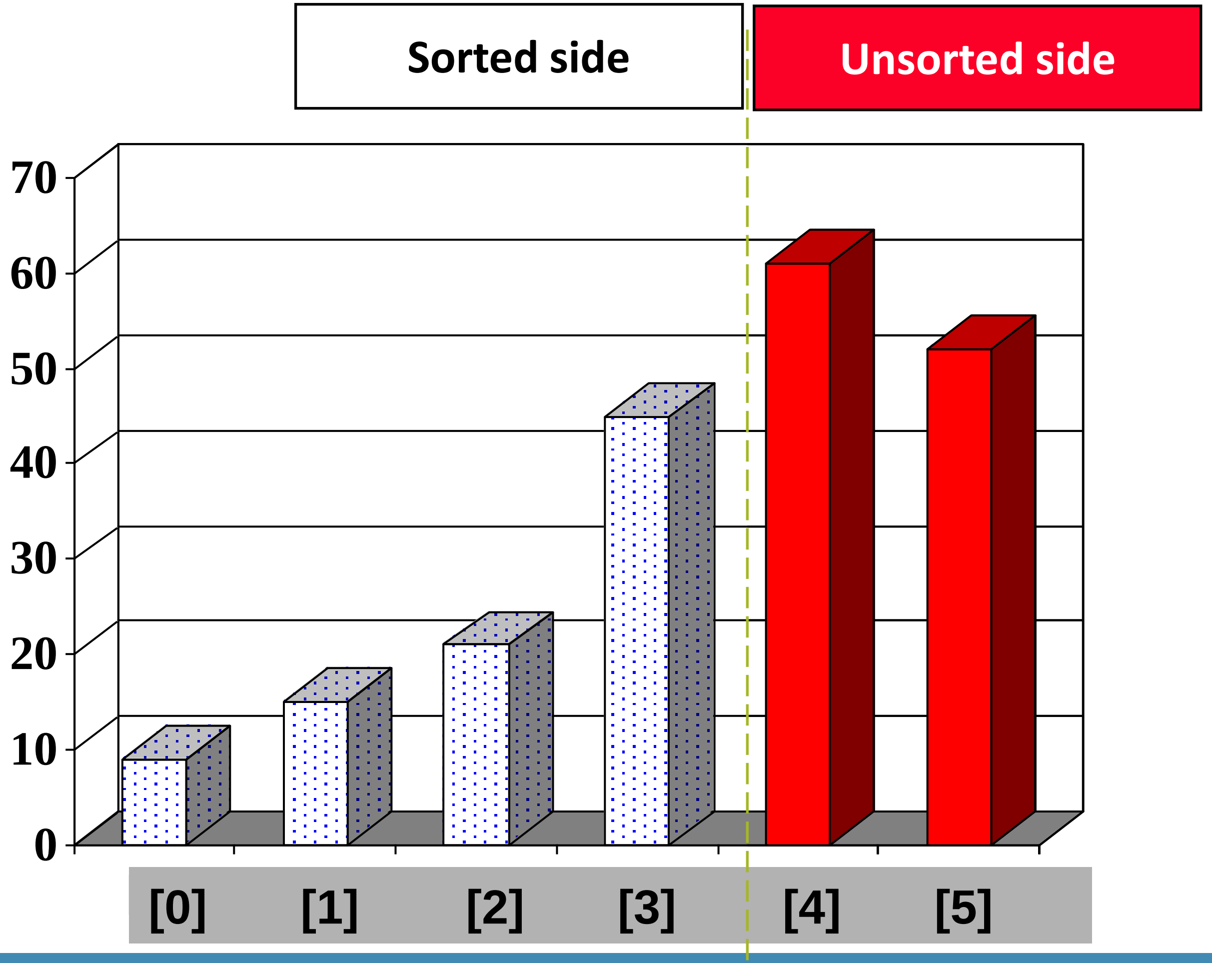


The process continues...

The Selection Sort Algorithm

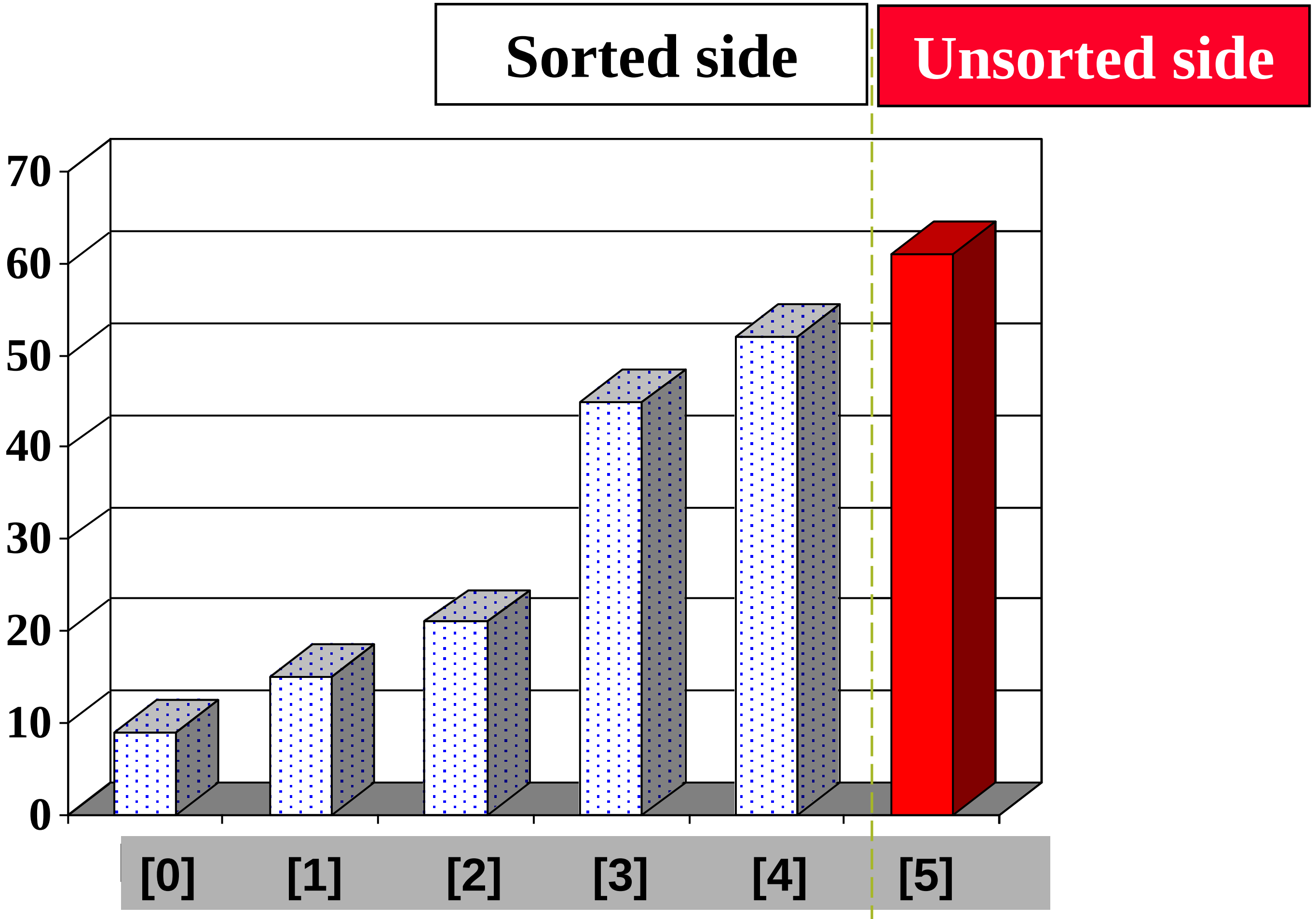
The process keeps adding one more number to the sorted side.

The sorted side has the smallest numbers, arranged from small to large.



The Selection Sort Algorithm

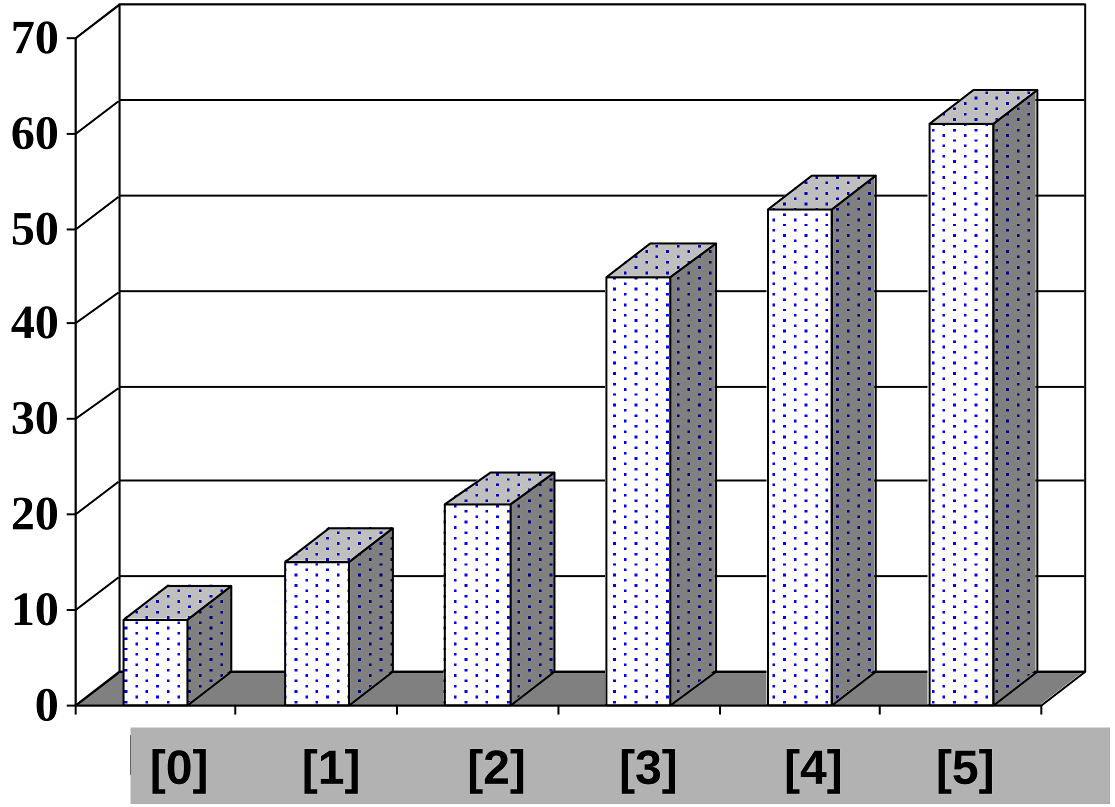
We can stop when the unsorted side has just one number, since that number must be the largest number.



The Selection Sort Algorithm

The array is now sorted.

We repeatedly **selected** the smallest element, and moved this element to the front of the unsorted side.



```
template <class Item>
void selection_sort(Item data[ ], size_t n){
    size_t i, j, smallest;
    Item temp;

    if(n < 2) return;  // nothing to sort!!

    for(i = 0; i < n-1 ; ++i){
        // find smallest in unsorted part of array
        smallest = i;
        for(j = i+1; j < n; ++j)
            if(data[smallest] > data[j]) smallest = j;

        // put it at front of unsorted part of array (swap)
        temp = data[i];
        data[i] = data[smallest];
        data[smallest] = temp;
    }
}
```

Selection Time Sort Analysis

In O-notation, what is:

- Worst case running time for *n* items?
- Average case running time for *n* items?

Steps of algorithm:

for i = 1 to n-1

find smallest key in unsorted part of array

swap smallest item to front of unsorted array

decrease size of unsorted array by 1

Selection Time Sort Analysis

In O-notation, what is:

- Worst case running time for *n* items?
- Average case running time for *n* items?

Steps of algorithm:

for i = 1 to n-1   O(n)

find smallest key in unsorted part of array   O(n)

swap smallest item to front of unsorted array

decrease size of unsorted array by 1

Selection sort analysis: $O(n^2)$

```
template <class Item>
void selection_sort(Item data[ ], size_t n){
    size_t i, j, smallest;
    Item temp;

    if(n < 2) return;  // nothing to sort!!

    for(i = 0; i < n-1 ; ++i)    {
       // find smallest in unsorted part of array
      smallest = i;
      for(j = i+1; j < n; ++j)
          if(data[smallest] > data[j]) smallest = j;

      // put it at front of unsorted part of array (swap)
      temp = data[i];
      data[i] = data[smallest];
      data[smallest] = temp;
    }
}
```

Outer loop: O(n)

```
template <class Item>
void selection_sort(Item data[ ], size_t n){
    size_t i, j, smallest;
    Item temp;

    if(n < 2) return;  // nothing to sort!!

    for(i = 0; i < n-1 ; ++i)    {
        // find smallest in unsorted part of array
        smallest = i;
        for(j = i+1; j < n; ++j)
            if(data[smallest] > data[j]) smallest = j;

        // put it at front of unsorted part of array (swap)
        temp = data[i];
        data[i] = data[smallest];
        data[smallest] = temp;
    }
}
```

Outer loop: O(n)

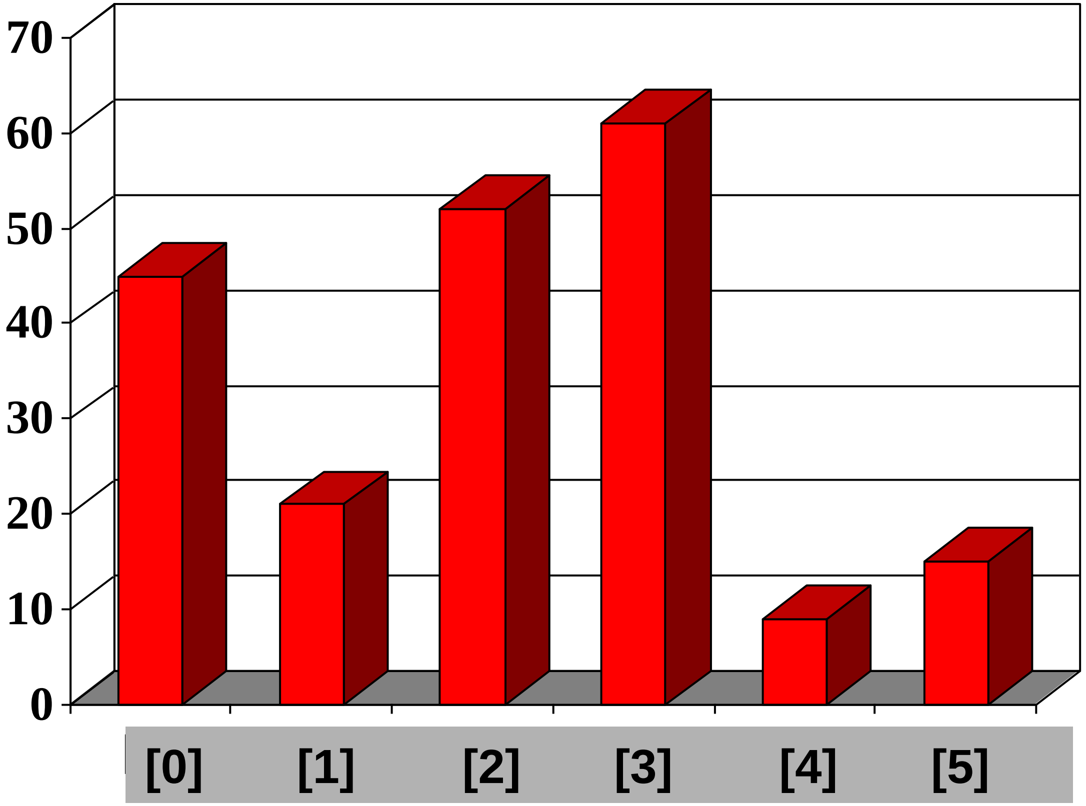
Inner loop: O(n)

LECTURE 3

Insertion Sort

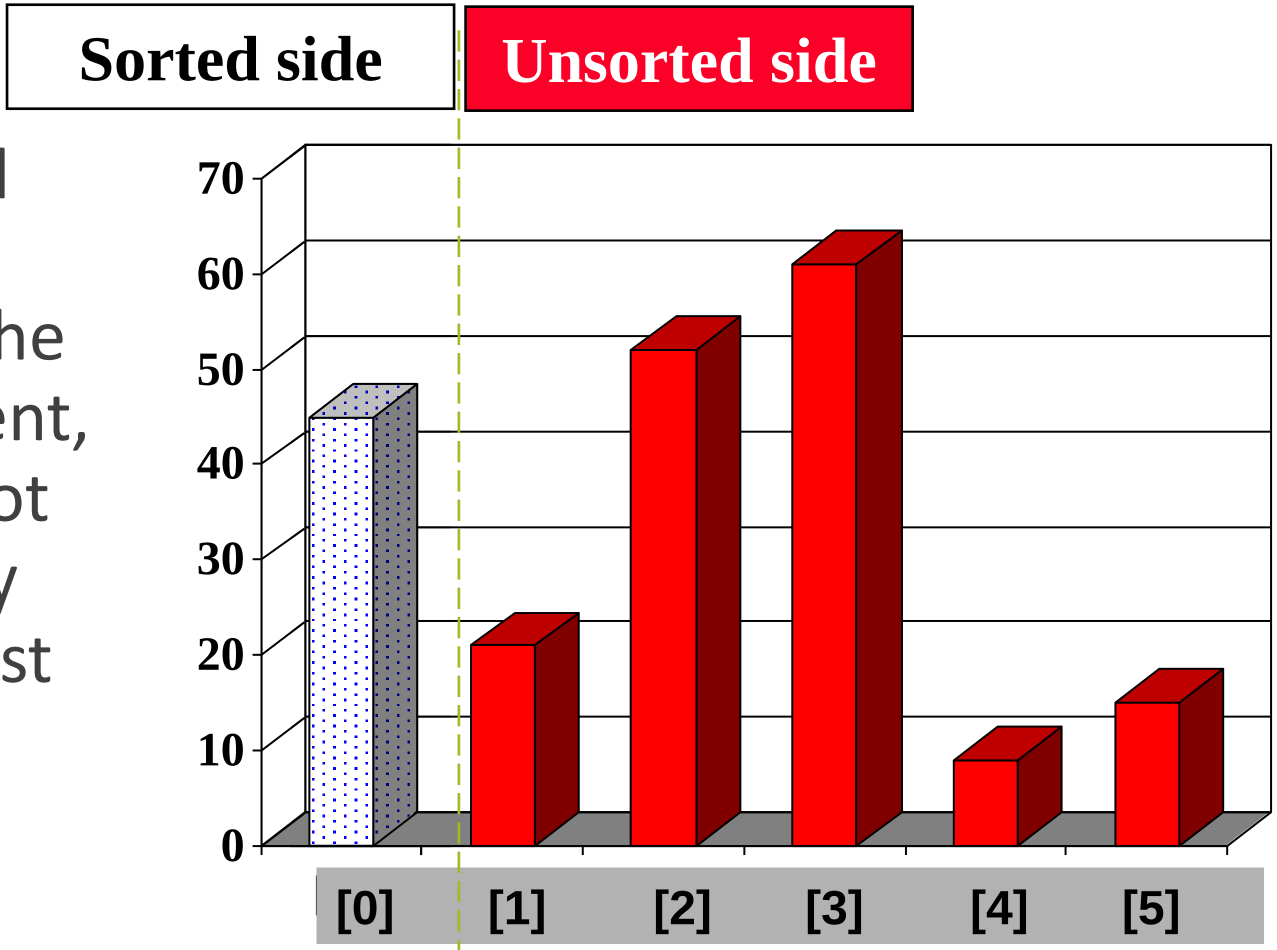
The Insertion Sort Algorithm

The Insertion Sort algorithm also views the array as having a sorted side and an unsorted side.



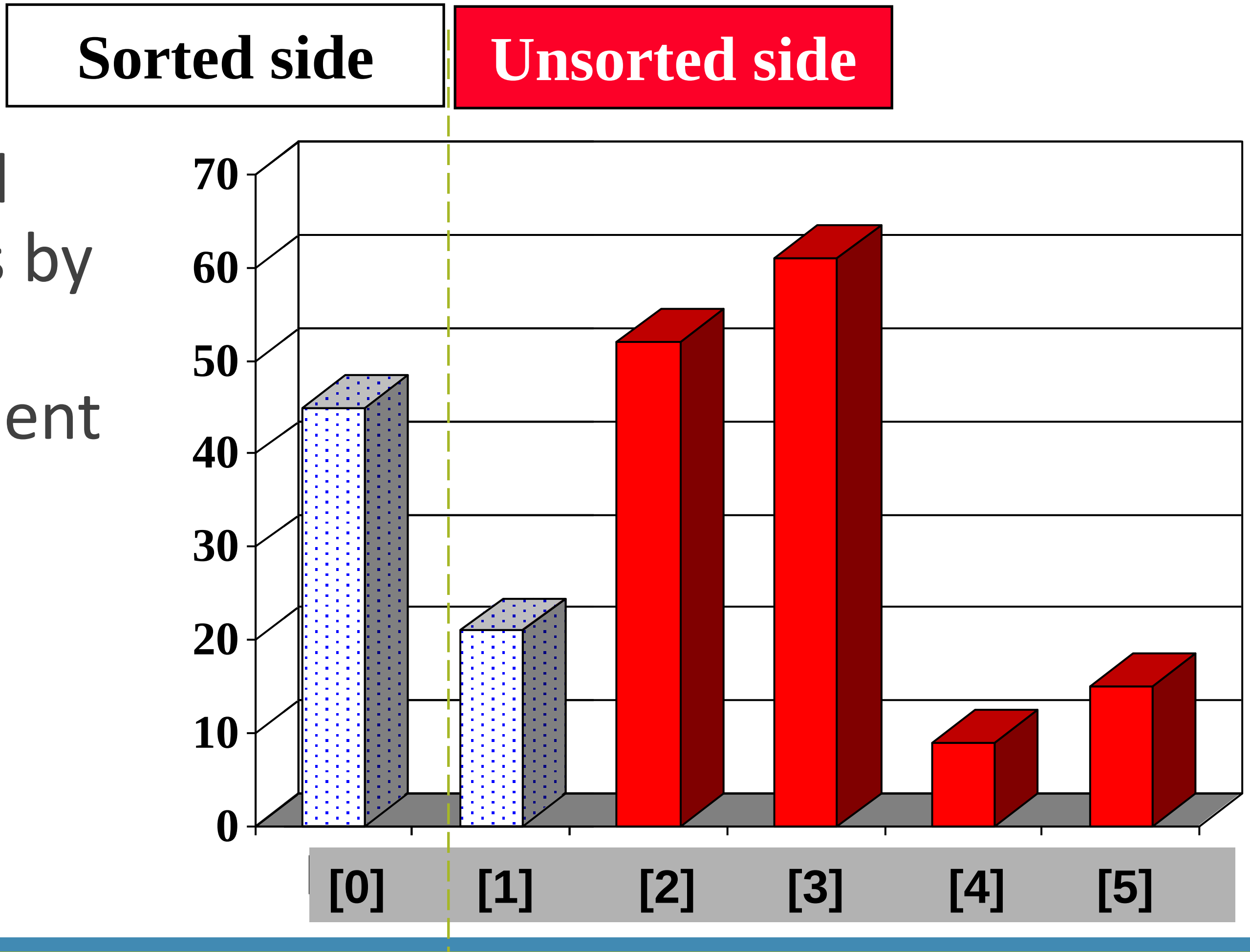
The Insertion Sort Algorithm

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



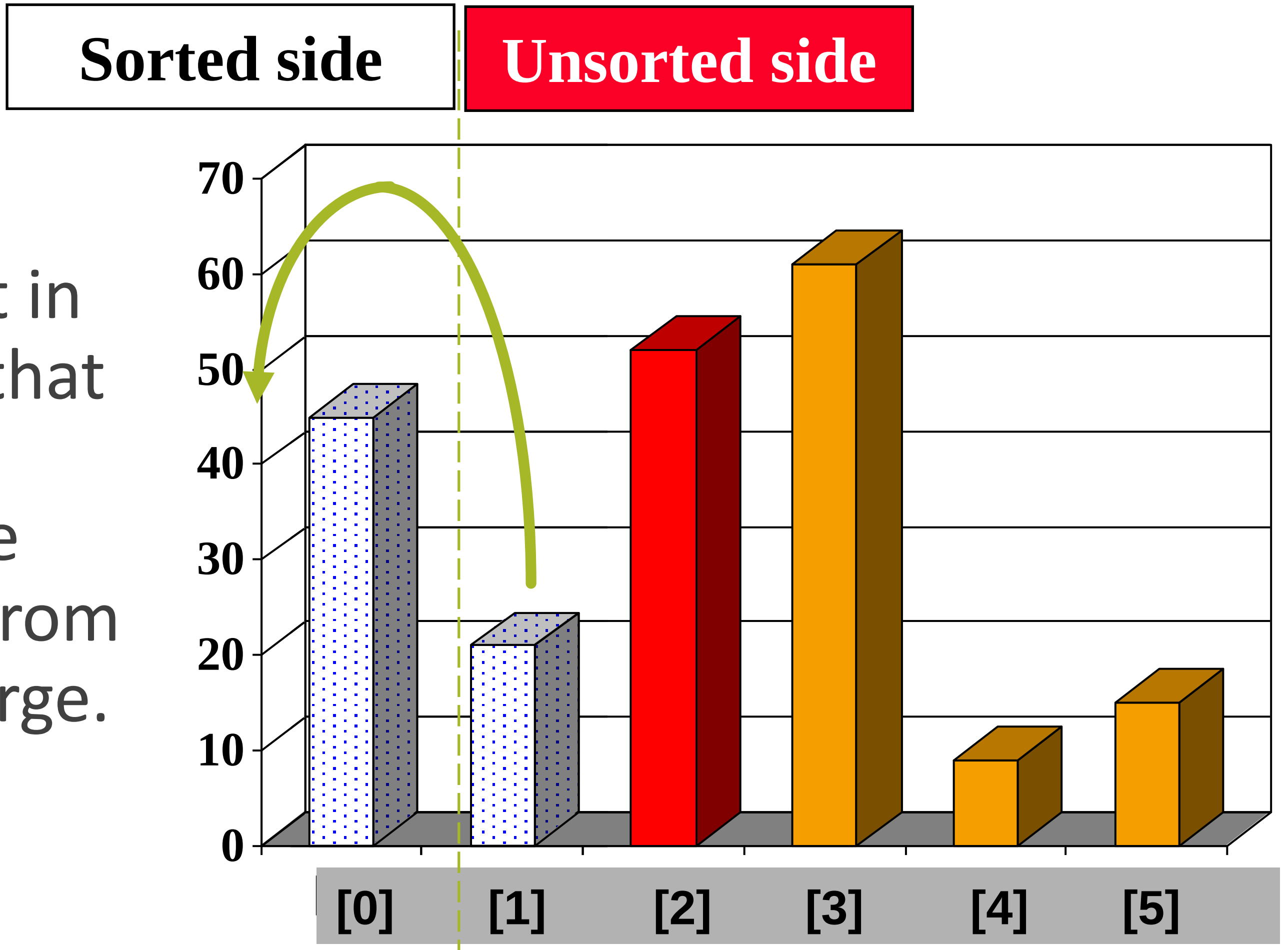
The Insertion Sort Algorithm

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

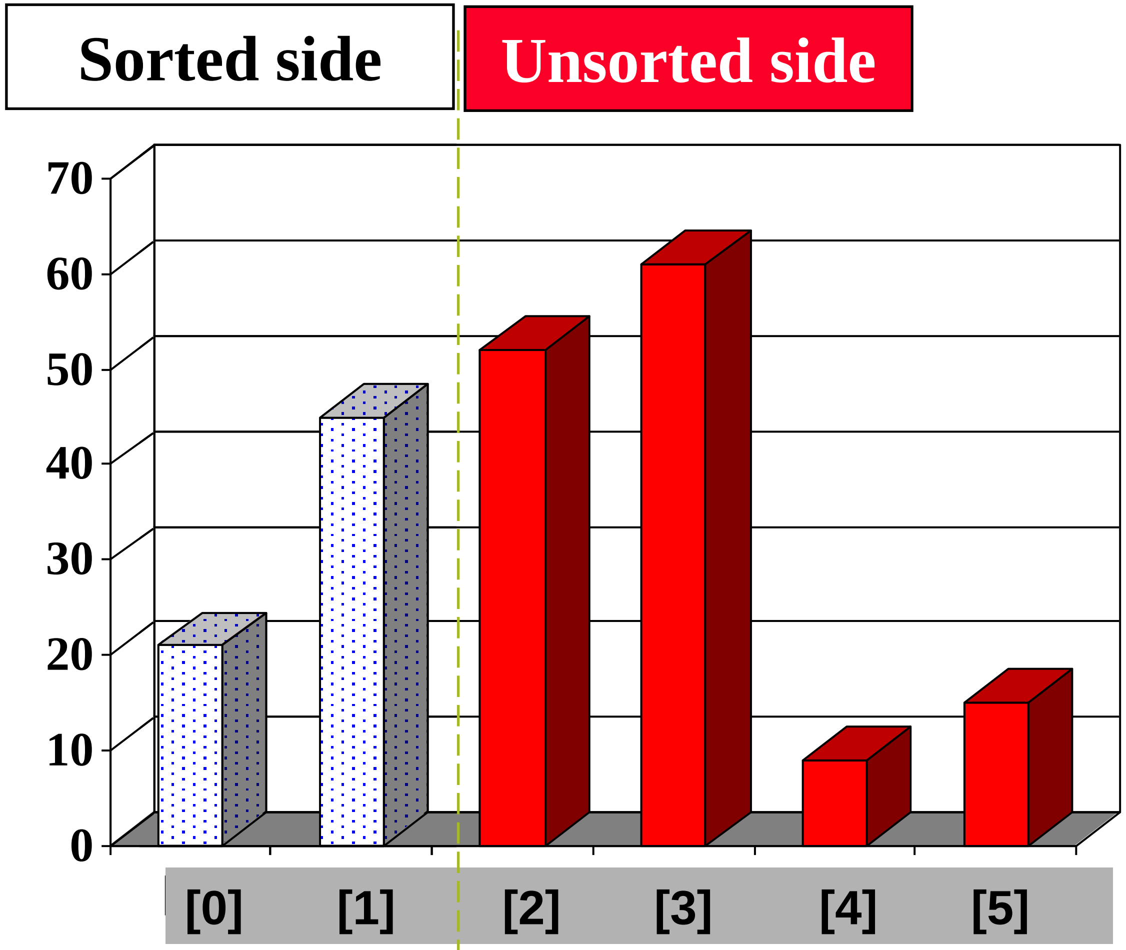


The Insertion Sort Algorithm

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

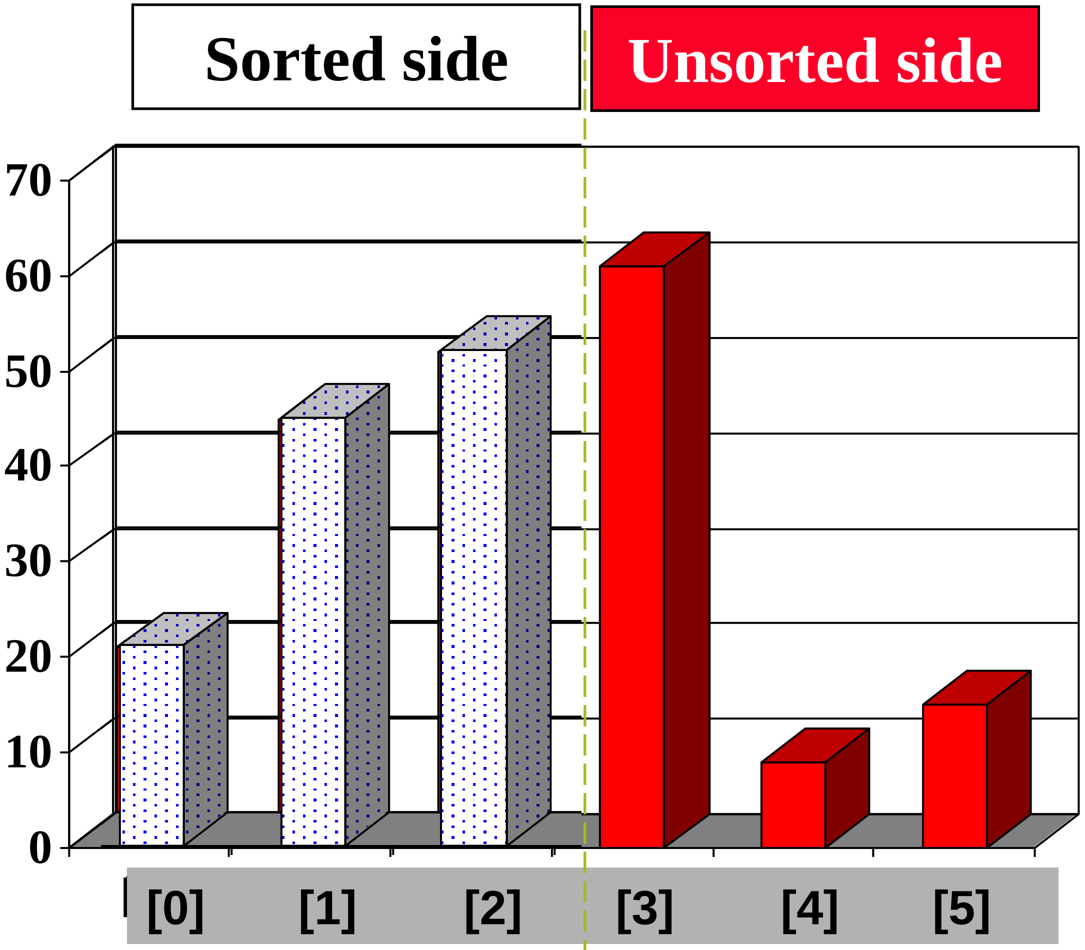


The Insertion Sort Algorithm



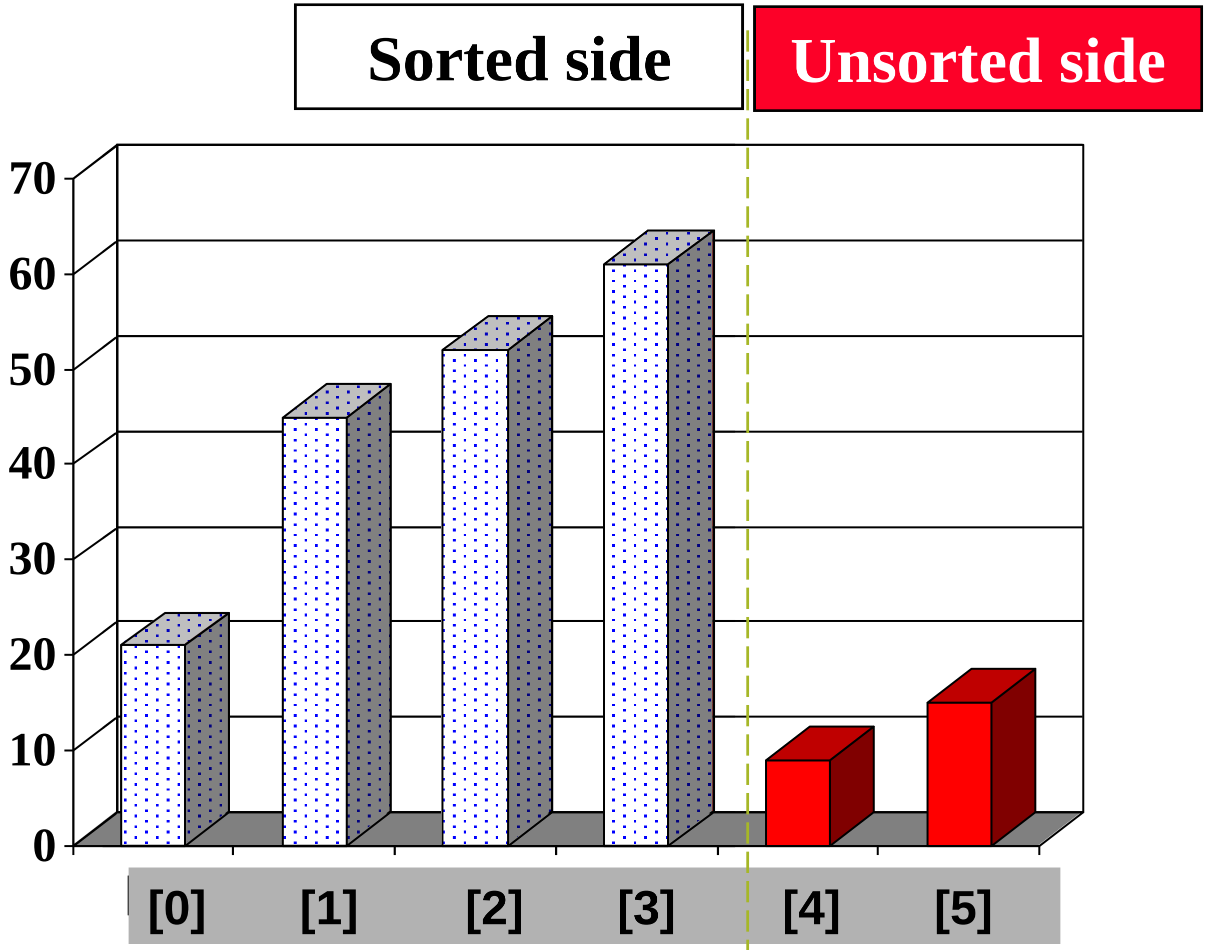
The Insertion Sort Algorithm

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



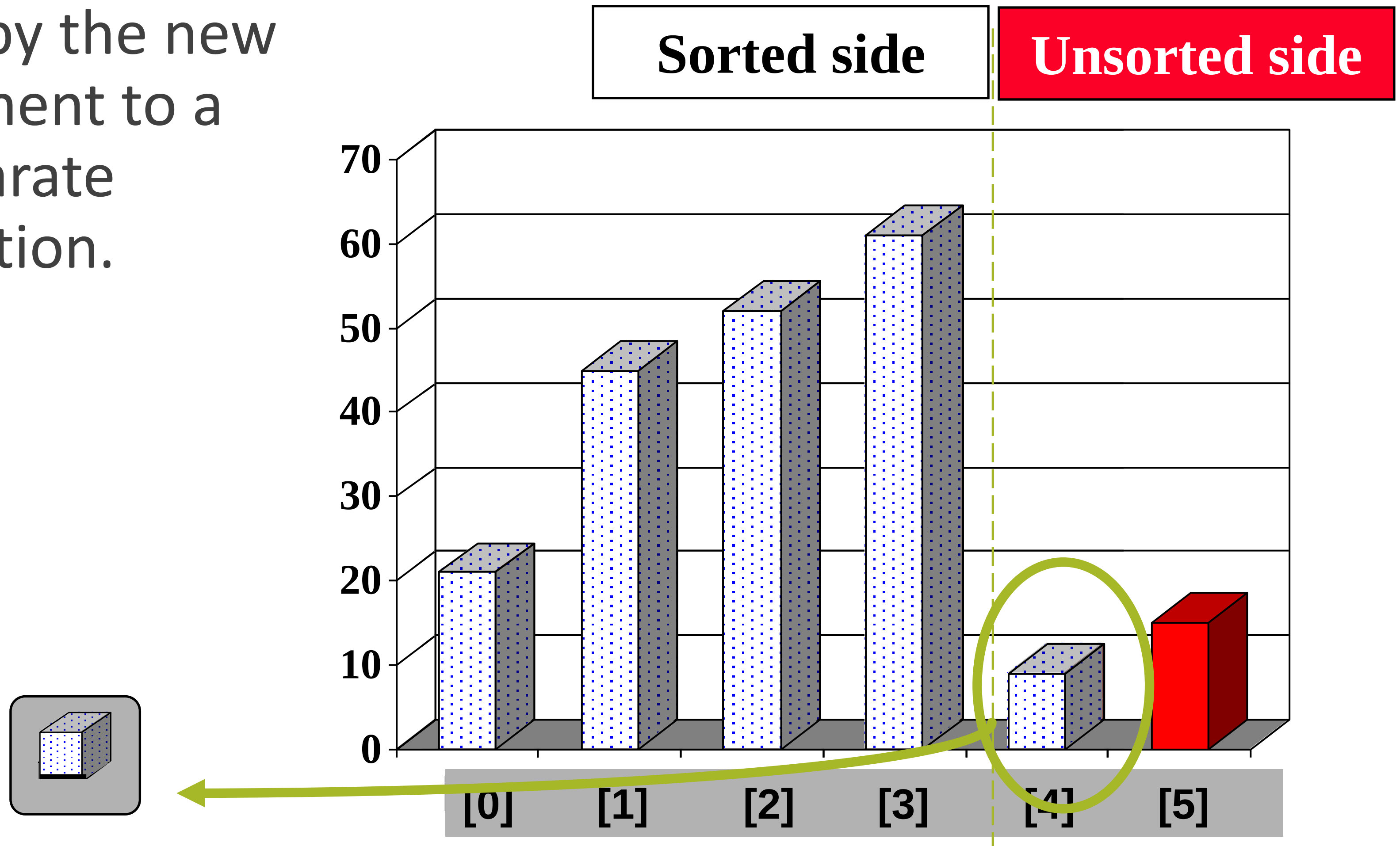
The Insertion Sort Algorithm

Sometimes we are lucky twice in a row.



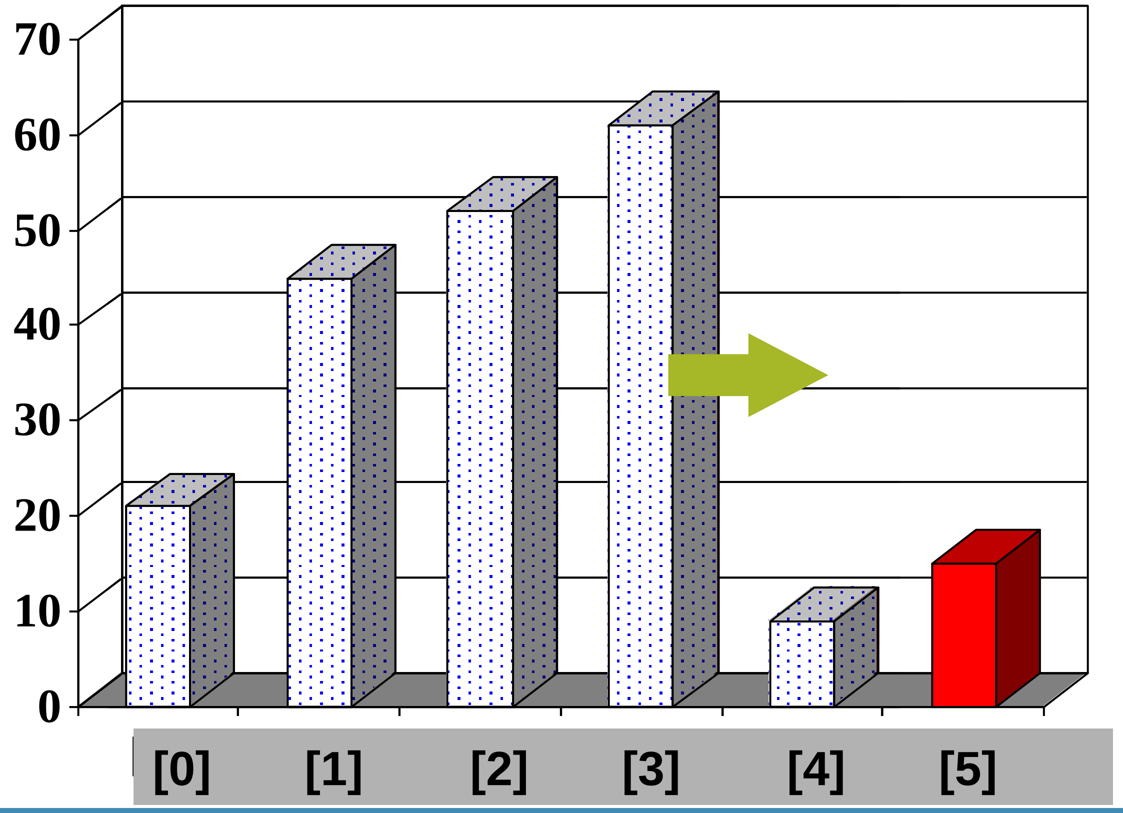
How to Insert One Element

- Copy the new element to a separate location.



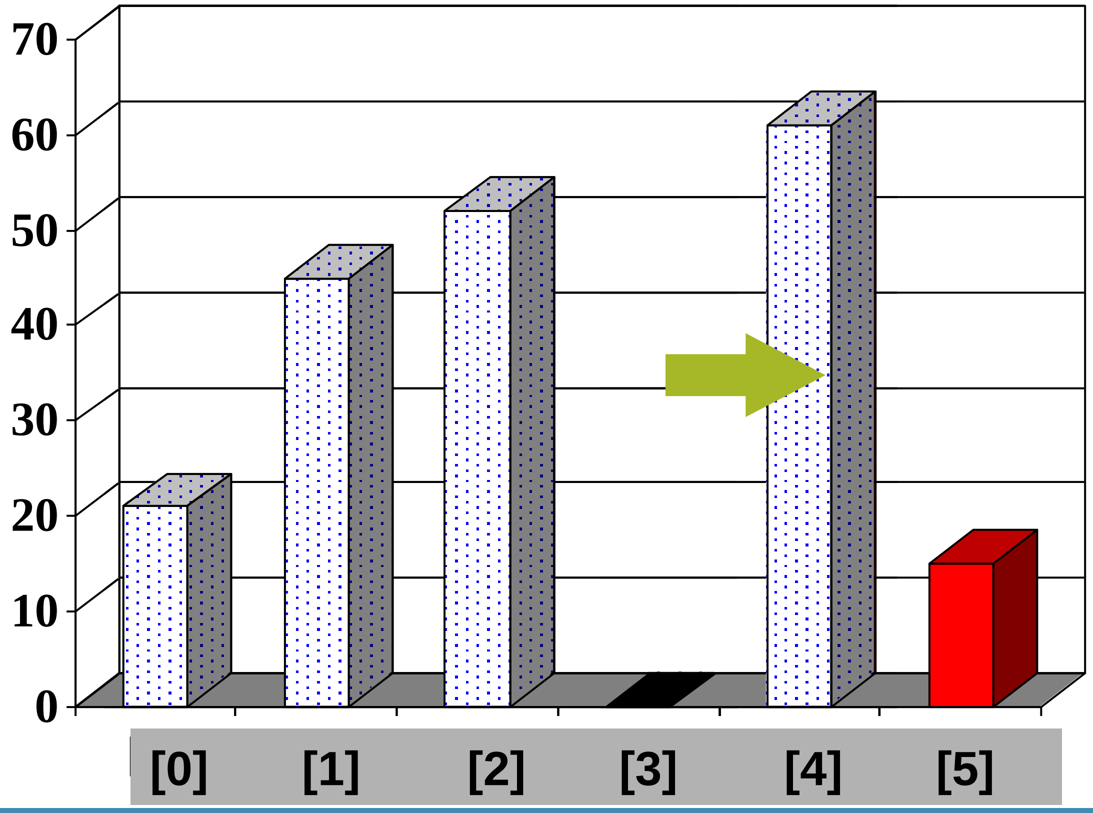
How to Insert One Element

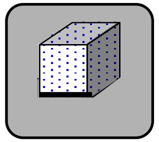
- Shift elements in the sorted side, creating an open space for the new element.



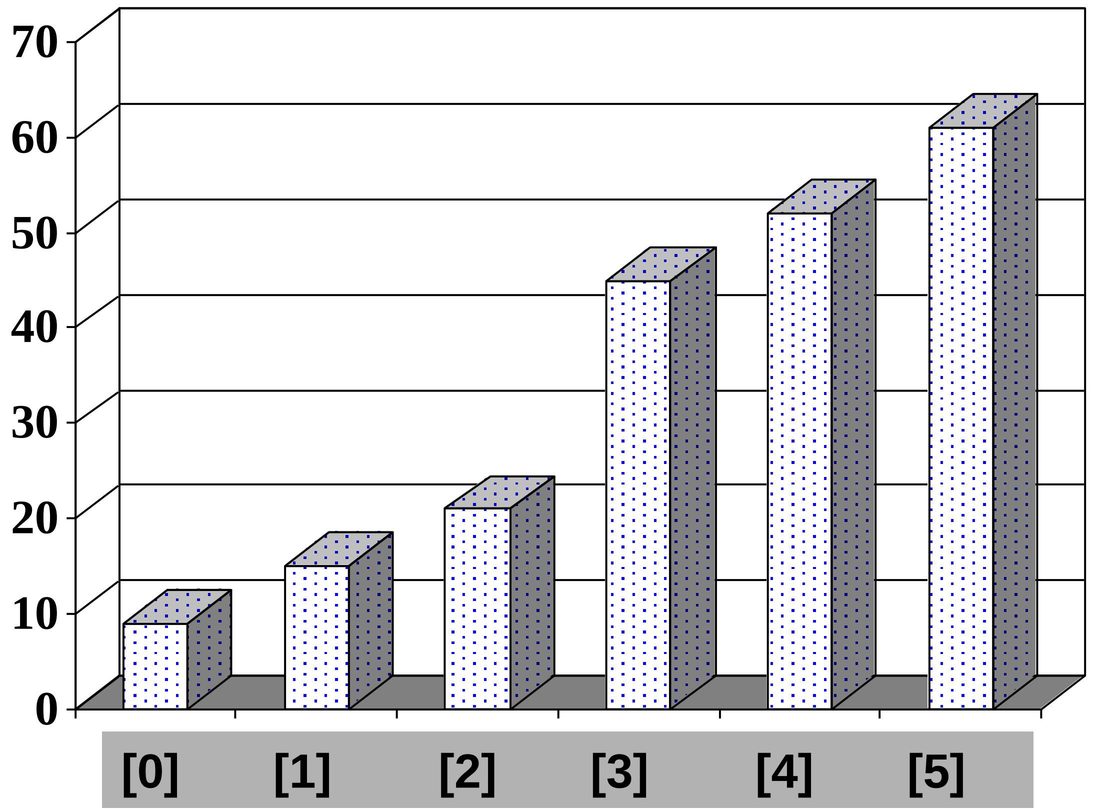
How to Insert One Element

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





Sorted Result



```
template <class Item>
void insertion_sort(Item data[ ], size_t n) {
    size_t i, j;
    Item temp;

    if(n < 2) return;  // nothing to sort!!

    for(i = 1; i < n; ++i)
    {
      // take next item at front of unsorted part of array
      // and insert it in appropriate location in sorted part of array
      temp = data[i];
      for(j = i; data[j-1] > temp and j > 0; --j)
        data[j] = data[j-1];  // shift element forward

      data[j] = temp;
    }
}
```

Insertion Sort Time Analysis

In O-notation, what is:

- Worst case running time for *n* items?
- Average case running time for *n* items?

Steps of algorithm:

```
for i = 1 to n-1
   take next key from unsorted part of array
   insert in appropriate location in sorted part of array:
               for j = i down to 0,
                           shift sorted elements to the right if key > key[i]
   increase size of sorted array by 1
```

Insertion Sort Time Analysis

In O-notation, what is:

- Worst case running time for *n* items?
- Average case running time for *n* items?

Steps of algorithm:

```
for i = 1 to n-1
   take next key from unsorted part of array
   insert in appropriate location in sorted part of array:
            for j = i down to 0,
                        shift sorted elements to the right if key > key[i]
   increase size of sorted array by 1
```

Outer loop: O(n)

Insertion Sort Time Analysis

In O-notation, what is:

- Worst case running time for *n* items?
- Average case running time for *n* items?

Steps of algorithm:

```
for i = 1 to n-1
   take next key from unsorted part of array
   insert in appropriate location in sorted part of array:
            for j = i down to 0,
                    shift sorted elements to the right if key > key[i]
   increase size of sorted array by 1
```

Outer loop: O(n)

Inner loop: O(n)

```
template <class Item>
void insertion_sort(Item data[ ], size_t n)
{
    size_t i, j;
    Item temp;

    if(n < 2) return;  // nothing to sort!!

    for(i = 1; i < n; ++i)
    {
      // take next item at front of unsorted part of array
      // and insert it in appropriate location in sorted part of array
      temp = data[i];
      for(j = i; data[j-1] > temp and j > 0; --j)
        data[j] = data[j-1];  // shift element forward

      data[j] = temp;
    }
}
```

O(n)

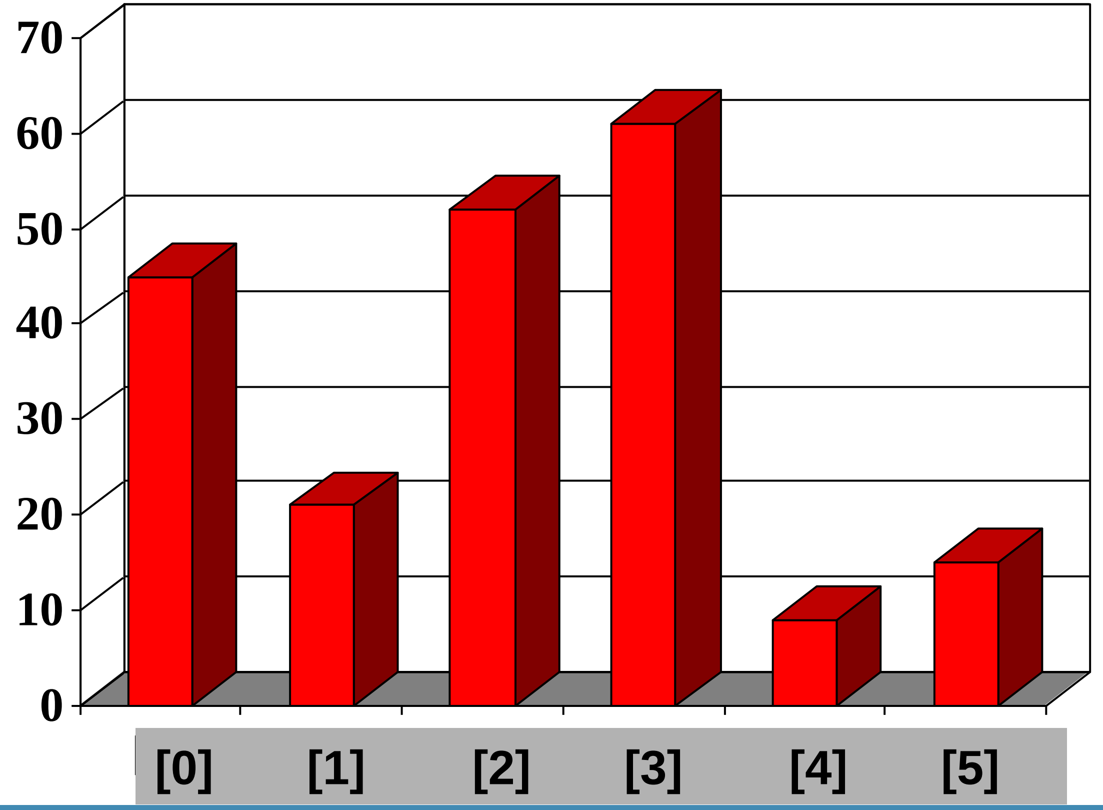
O(n)

LECTURE 4

Bubble Sort

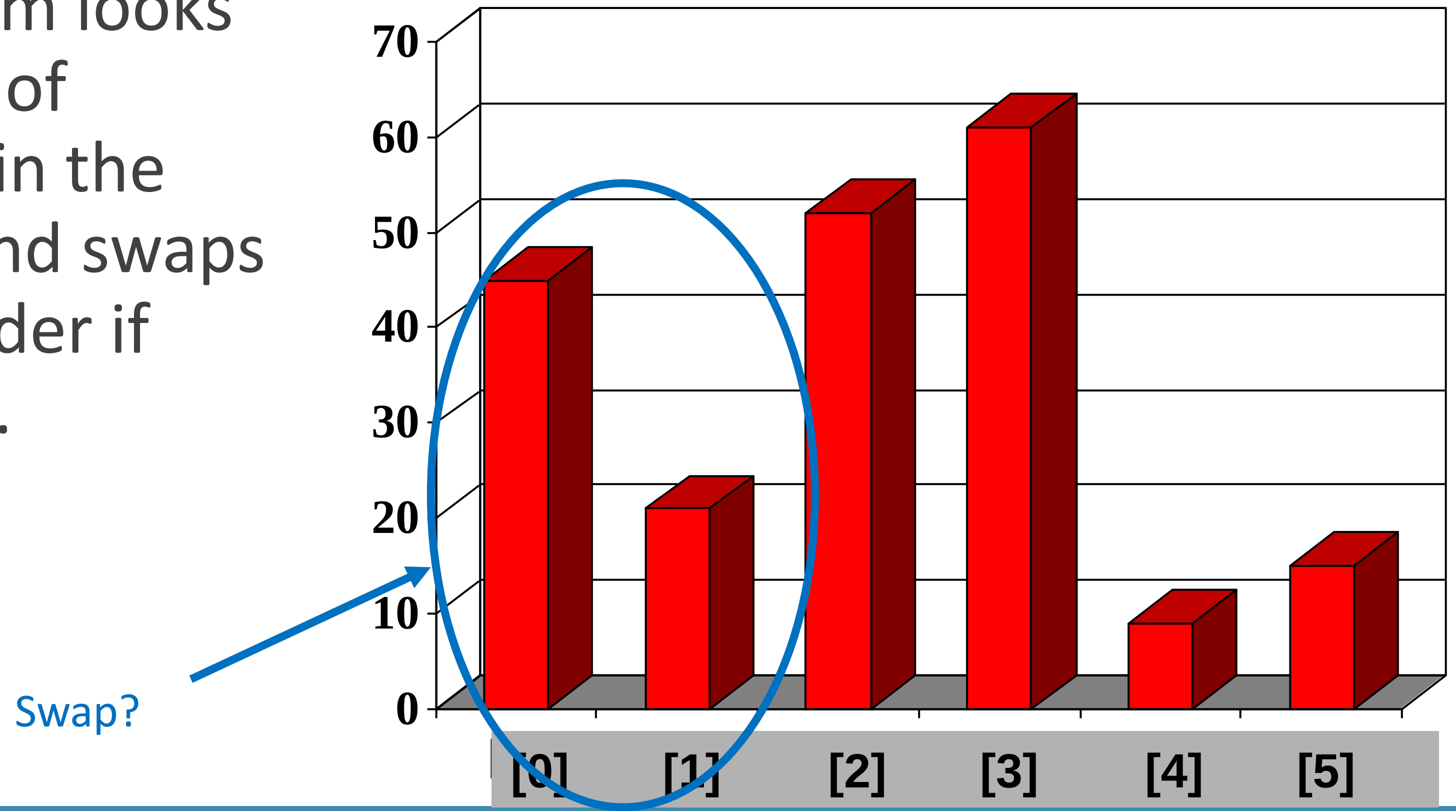
The Bubble Sort Algorithm

The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



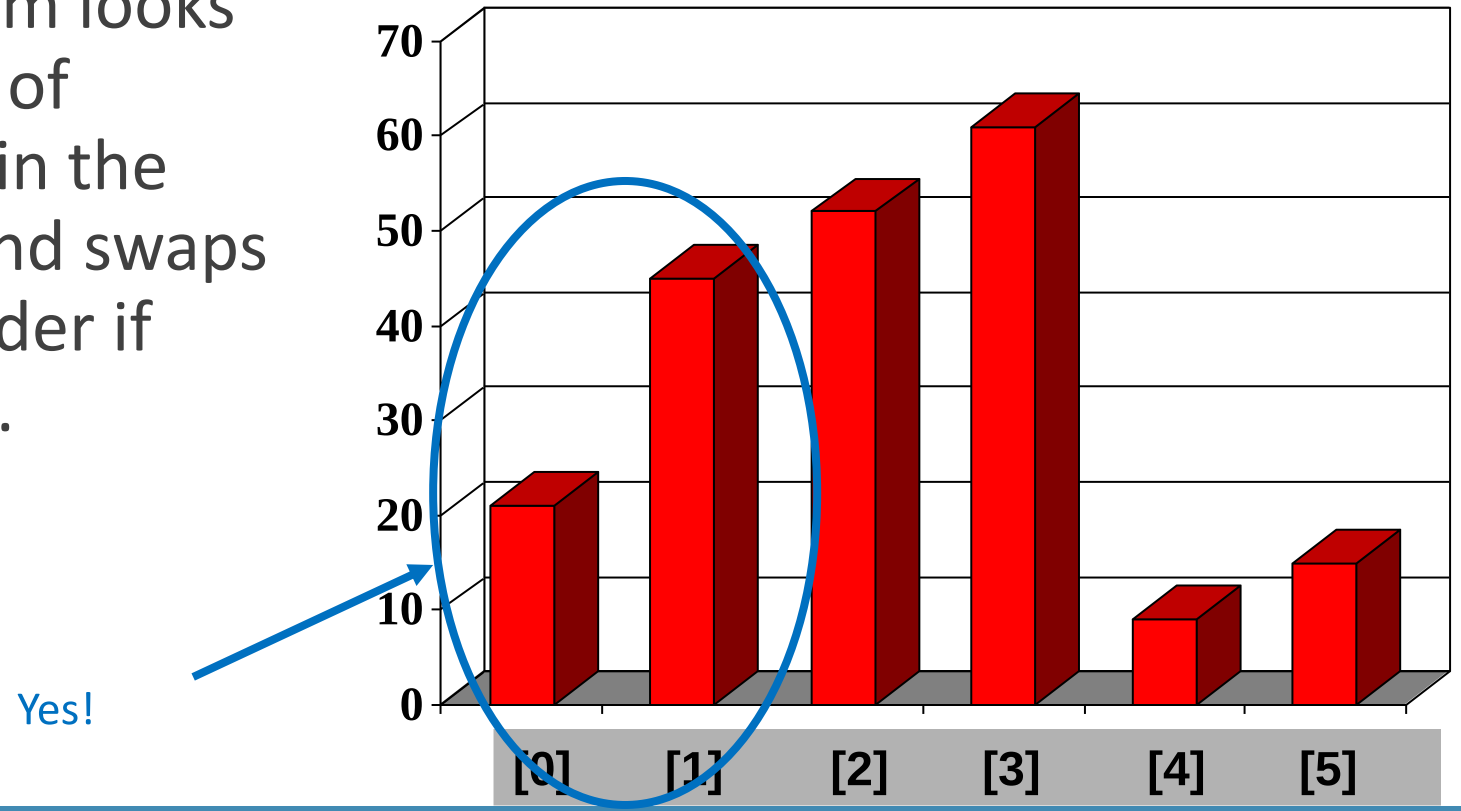
The Bubble Sort Algorithm

The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



The Bubble Sort Algorithm

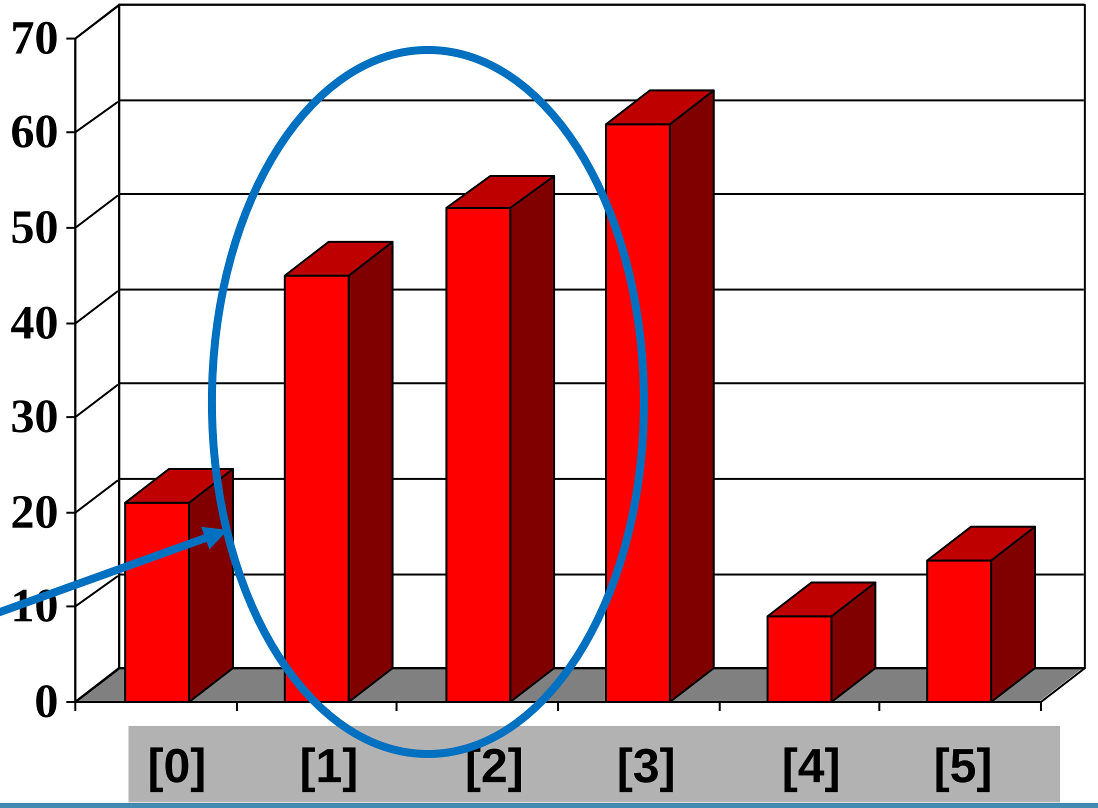
The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



The Bubble Sort Algorithm

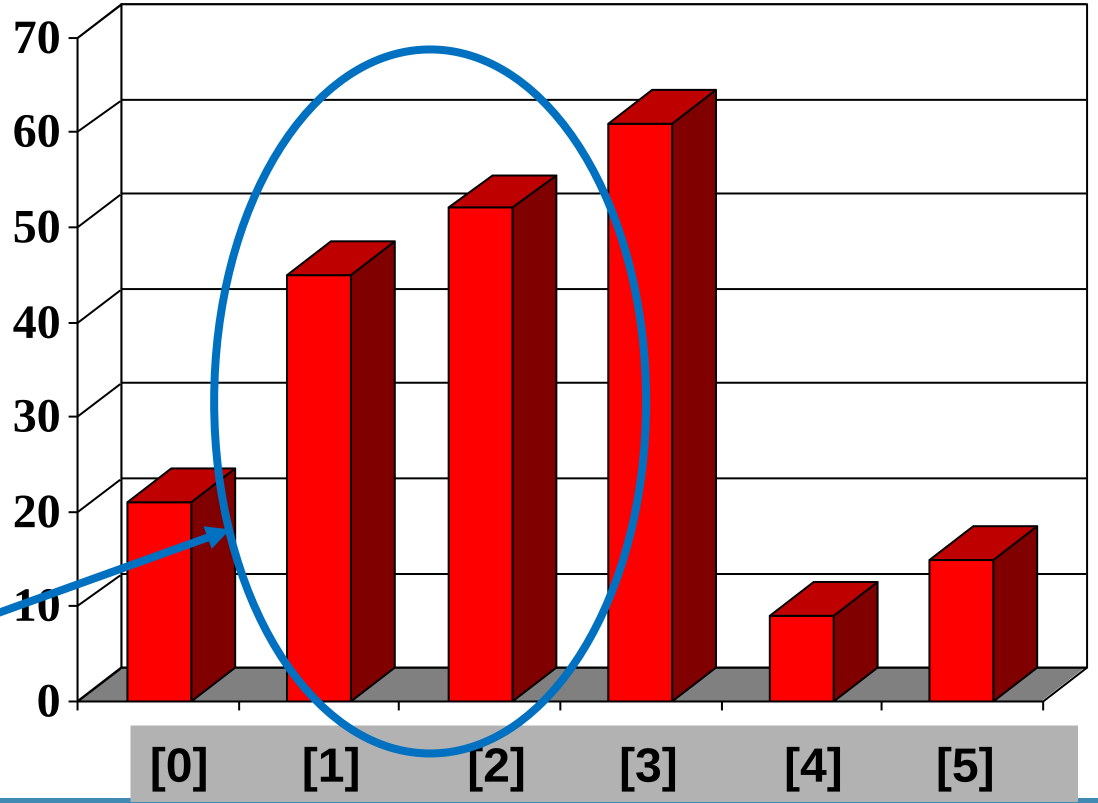
The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.

Swap?



The Bubble Sort Algorithm

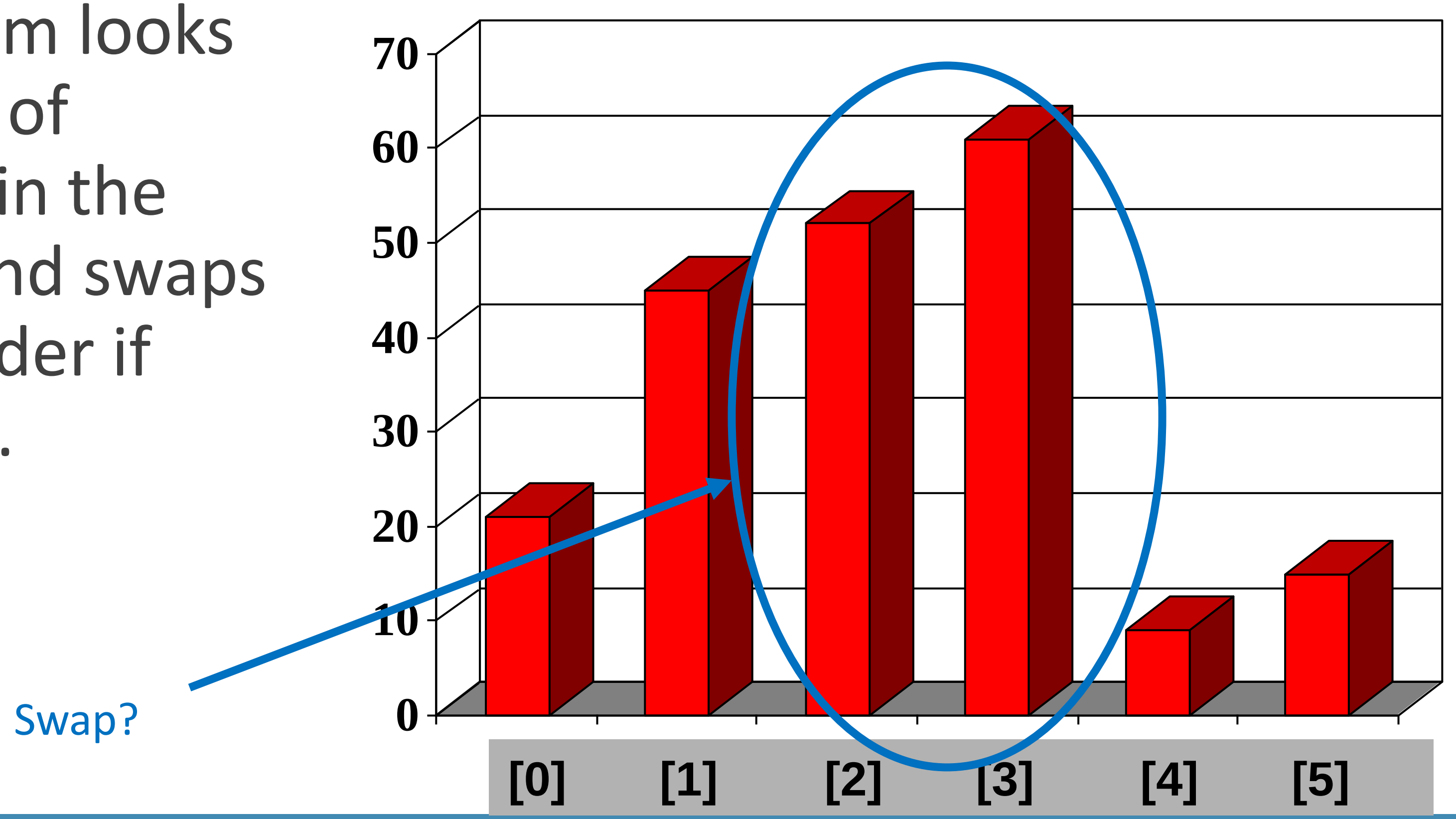
The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



No.

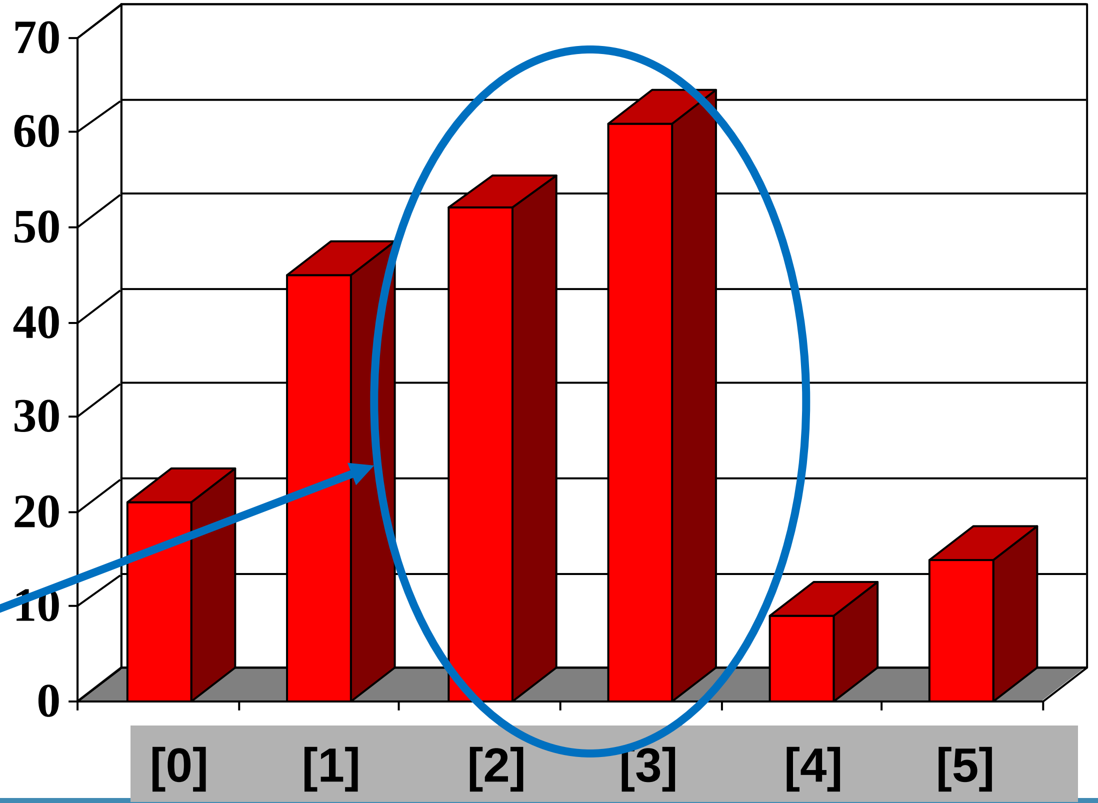
The Bubble Sort Algorithm

The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



The Bubble Sort Algorithm

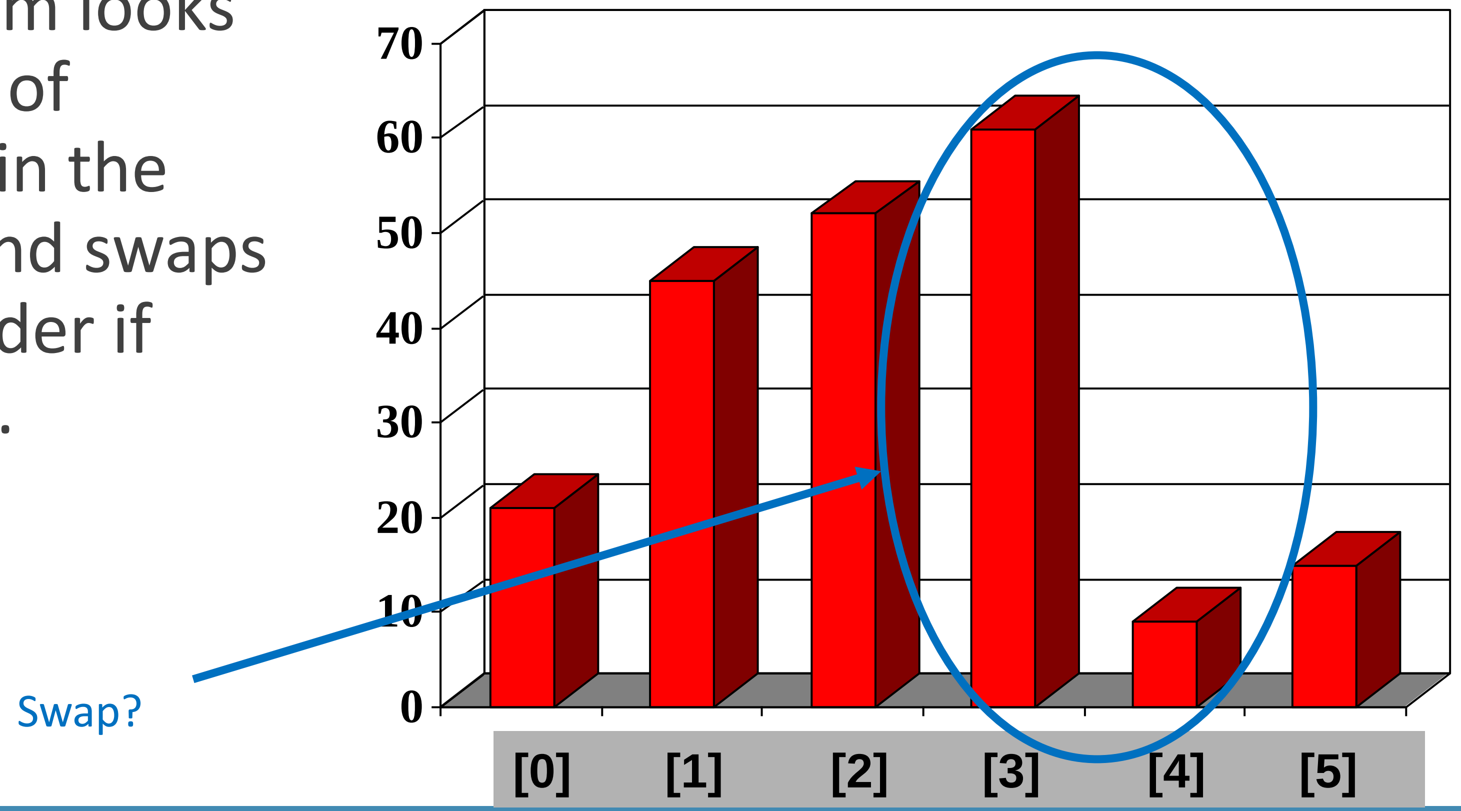
The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



No.

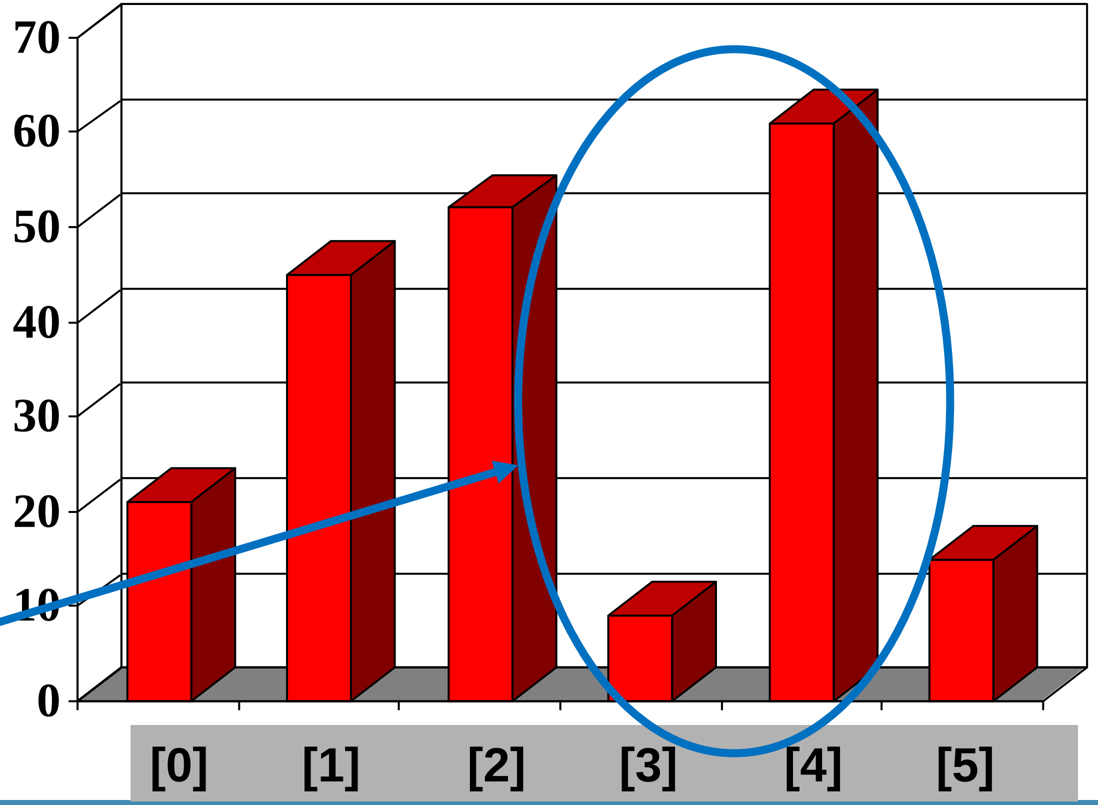
The Bubble Sort Algorithm

The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



The Bubble Sort Algorithm

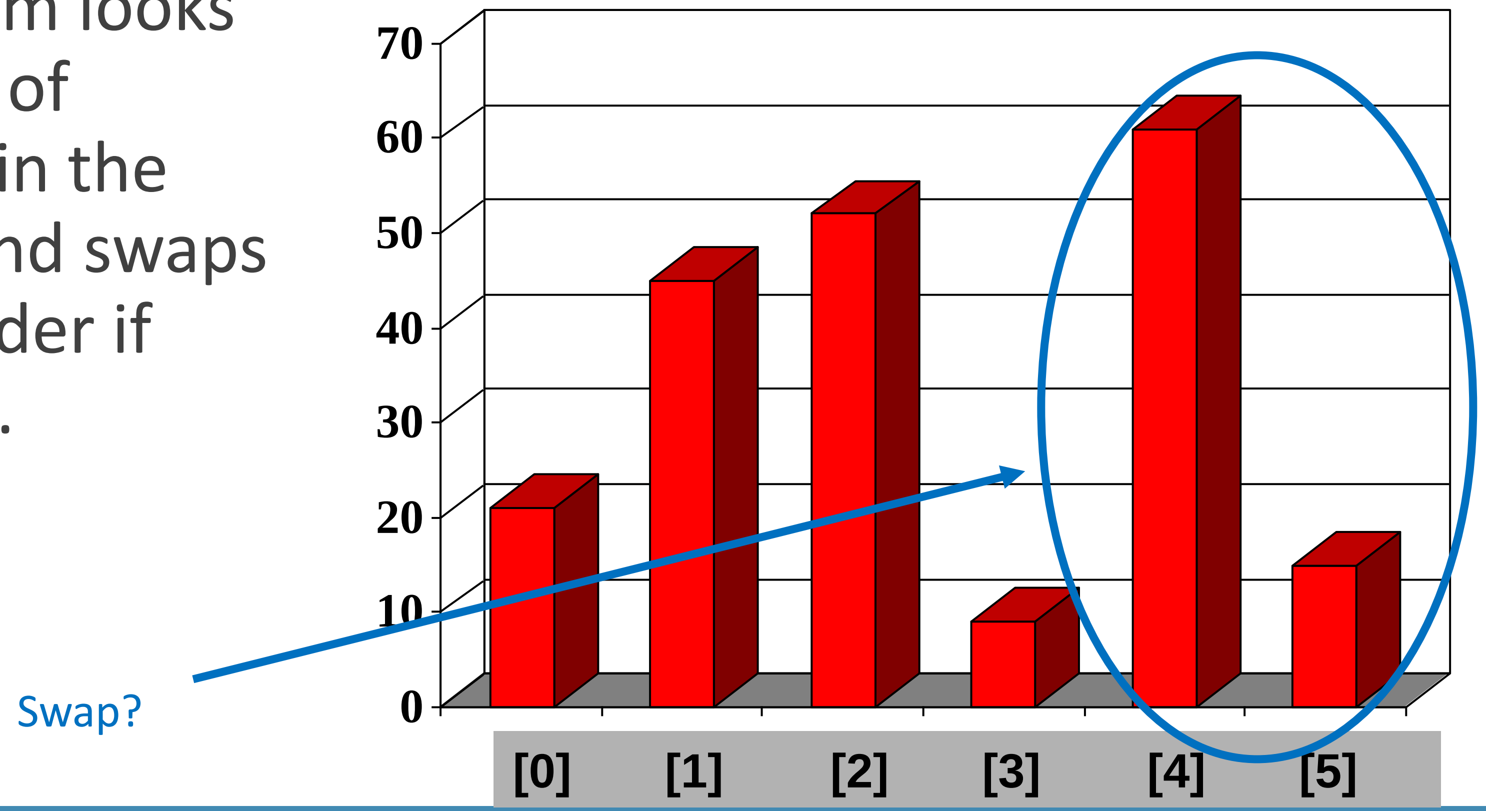
The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



Yes!

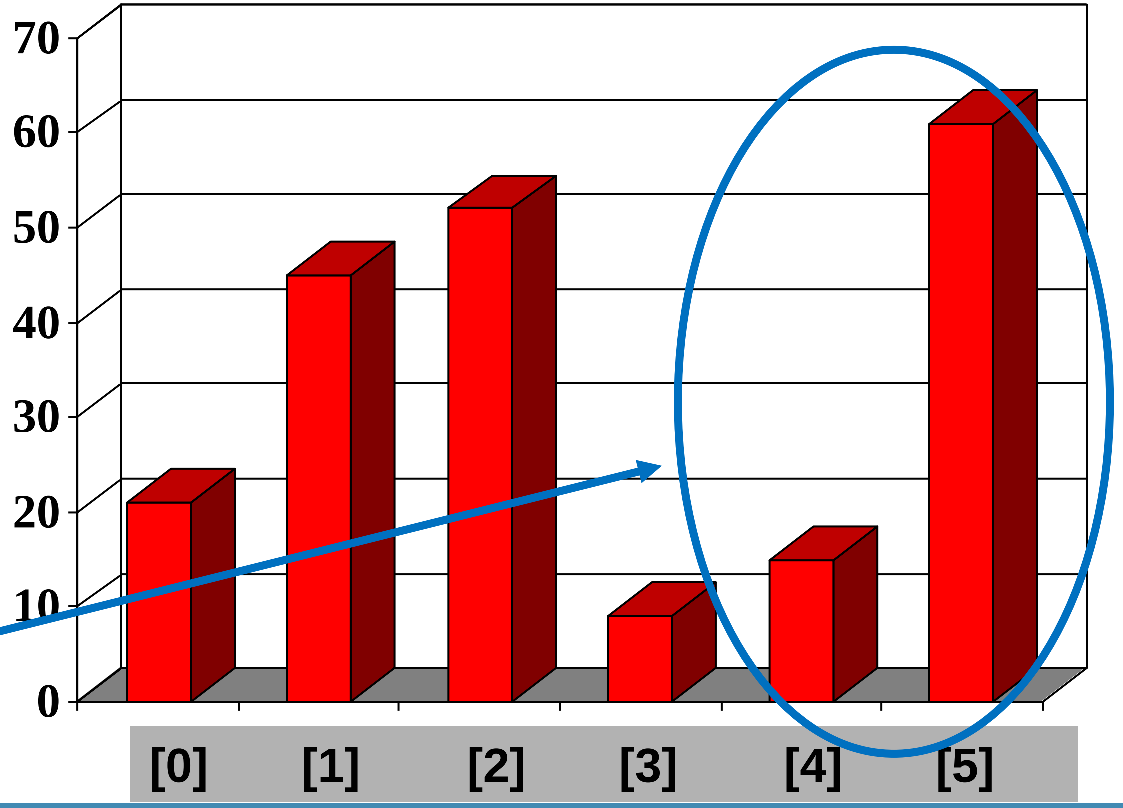
The Bubble Sort Algorithm

The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



The Bubble Sort Algorithm

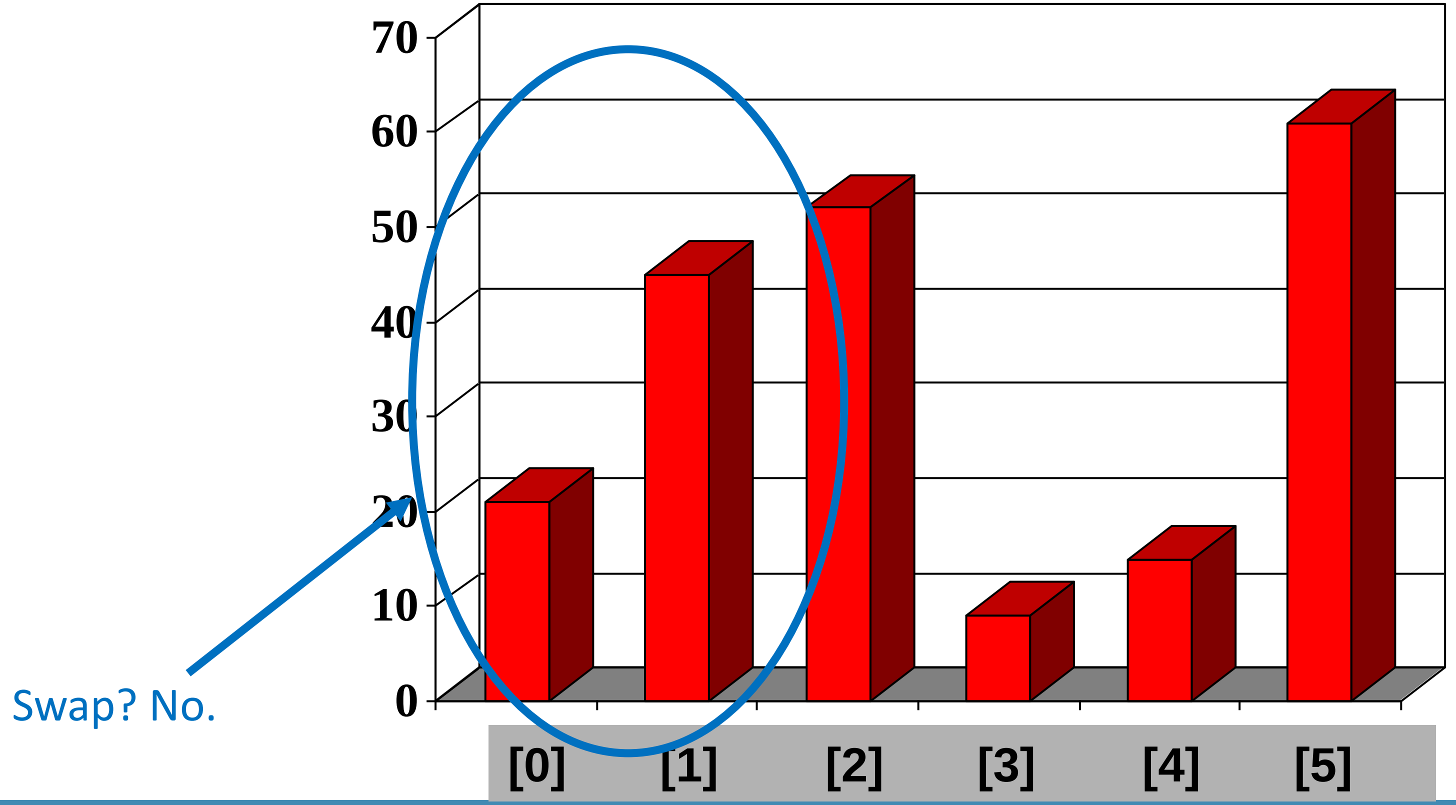
The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



Yes!

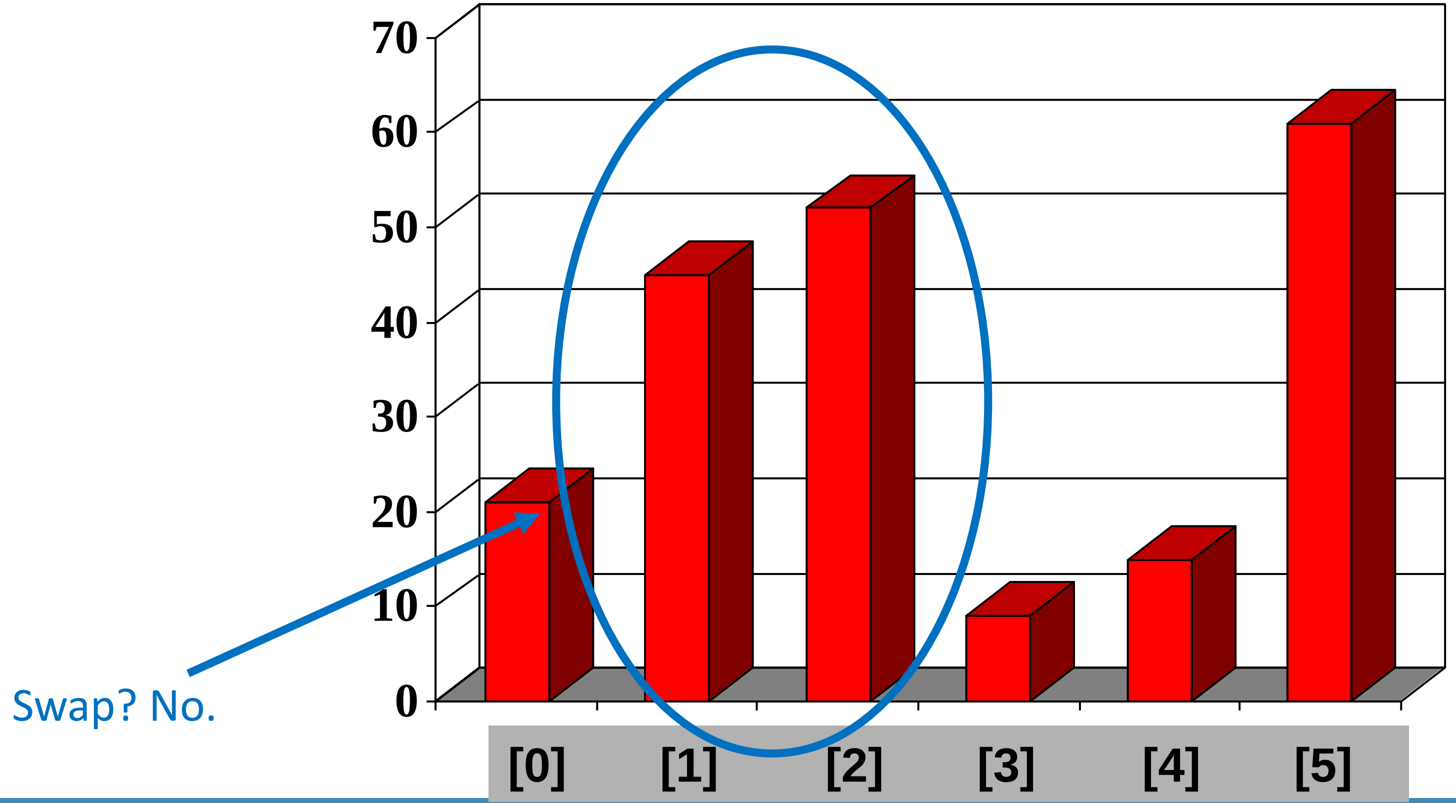
The Bubble Sort Algorithm

Repeat.



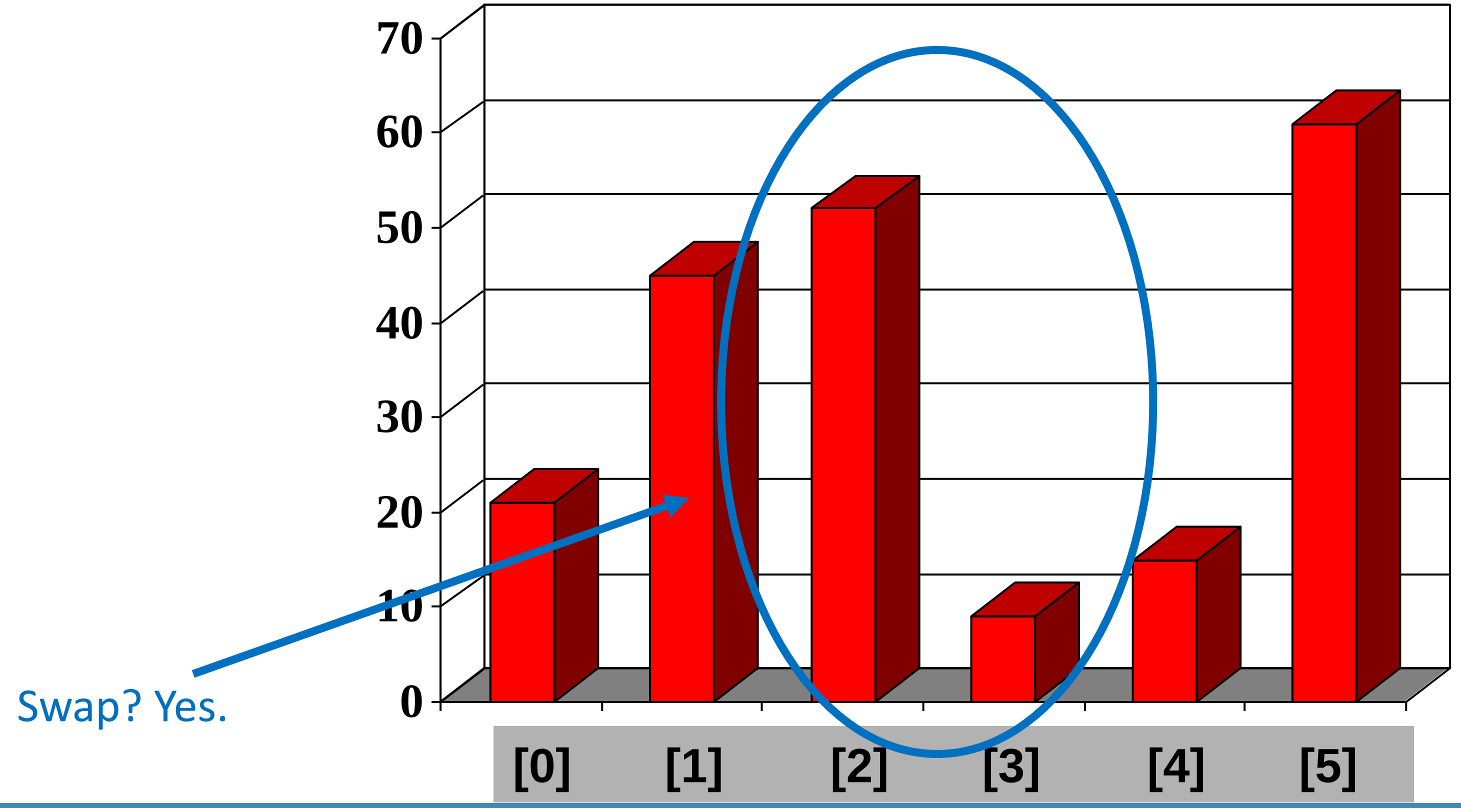
The Bubble Sort Algorithm

Repeat.



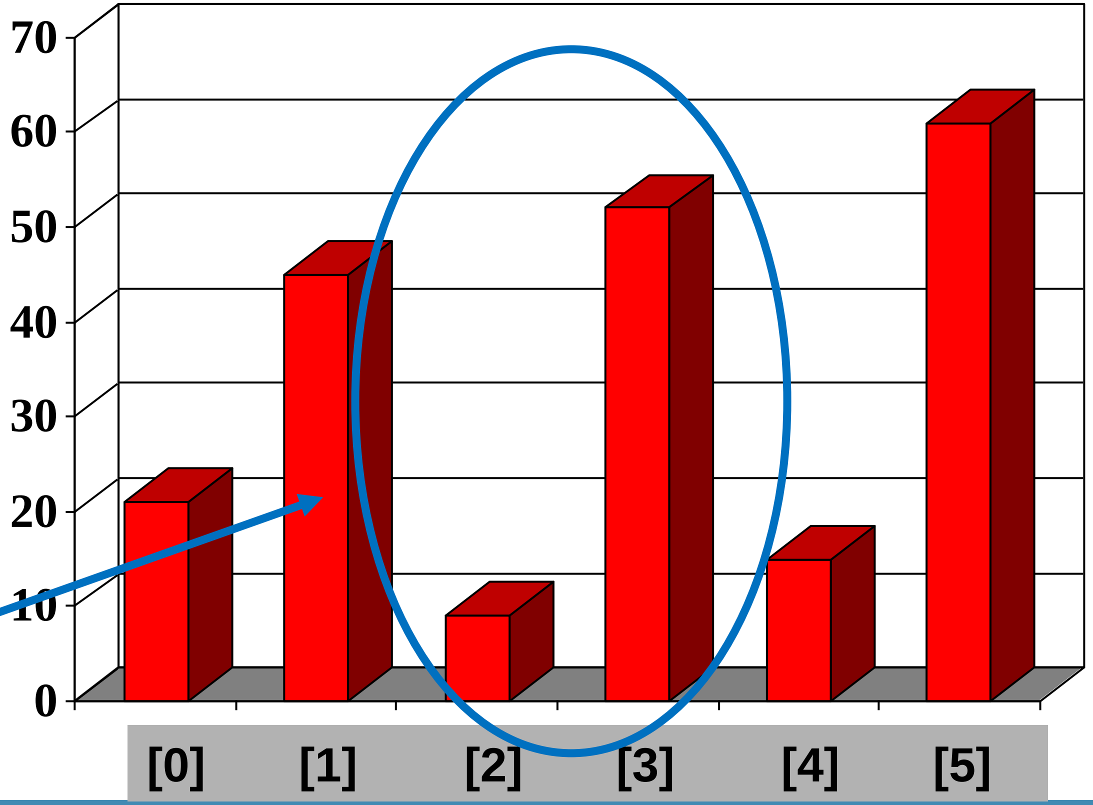
The Bubble Sort Algorithm

Repeat.



The Bubble Sort Algorithm

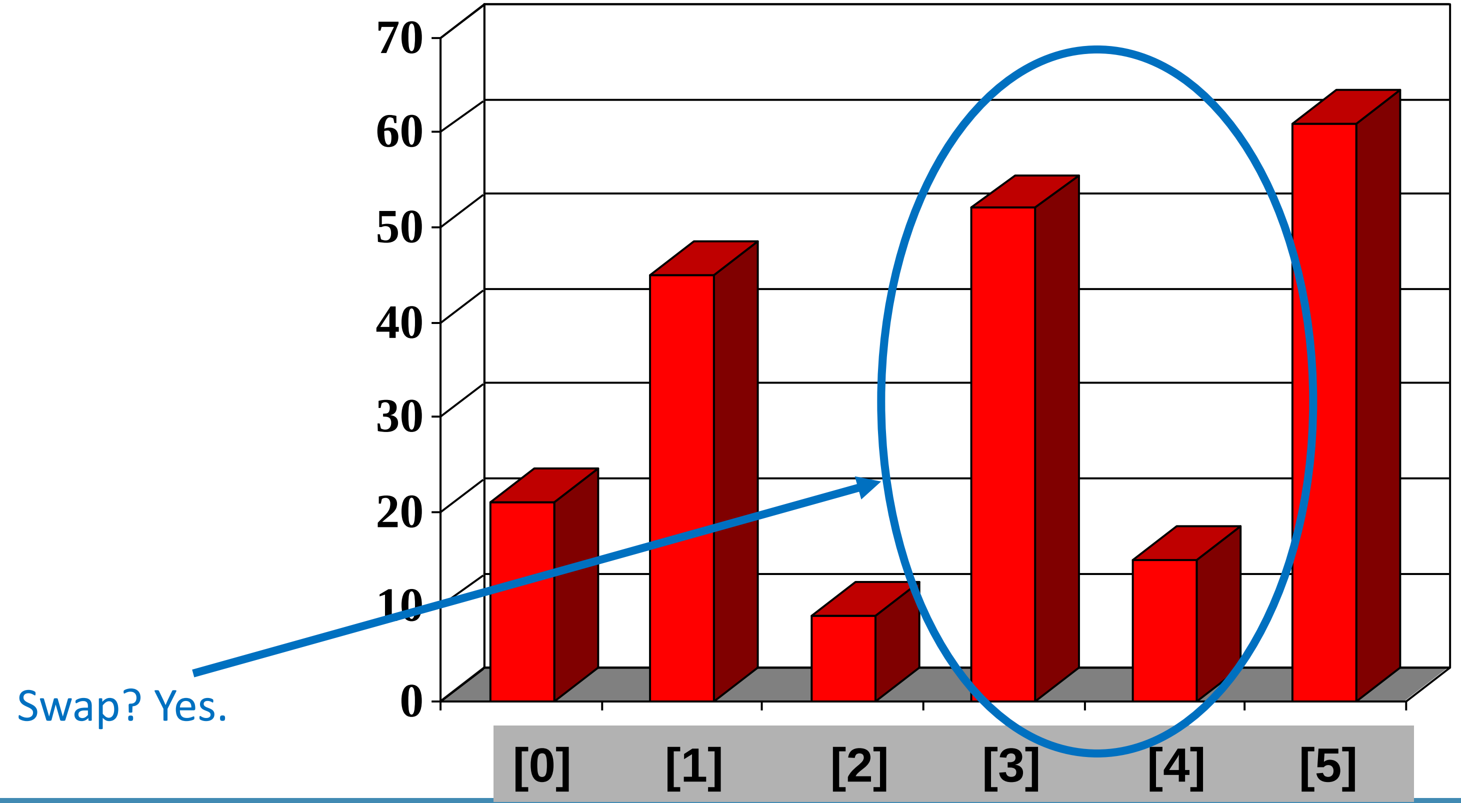
Repeat.



Swap? Yes.

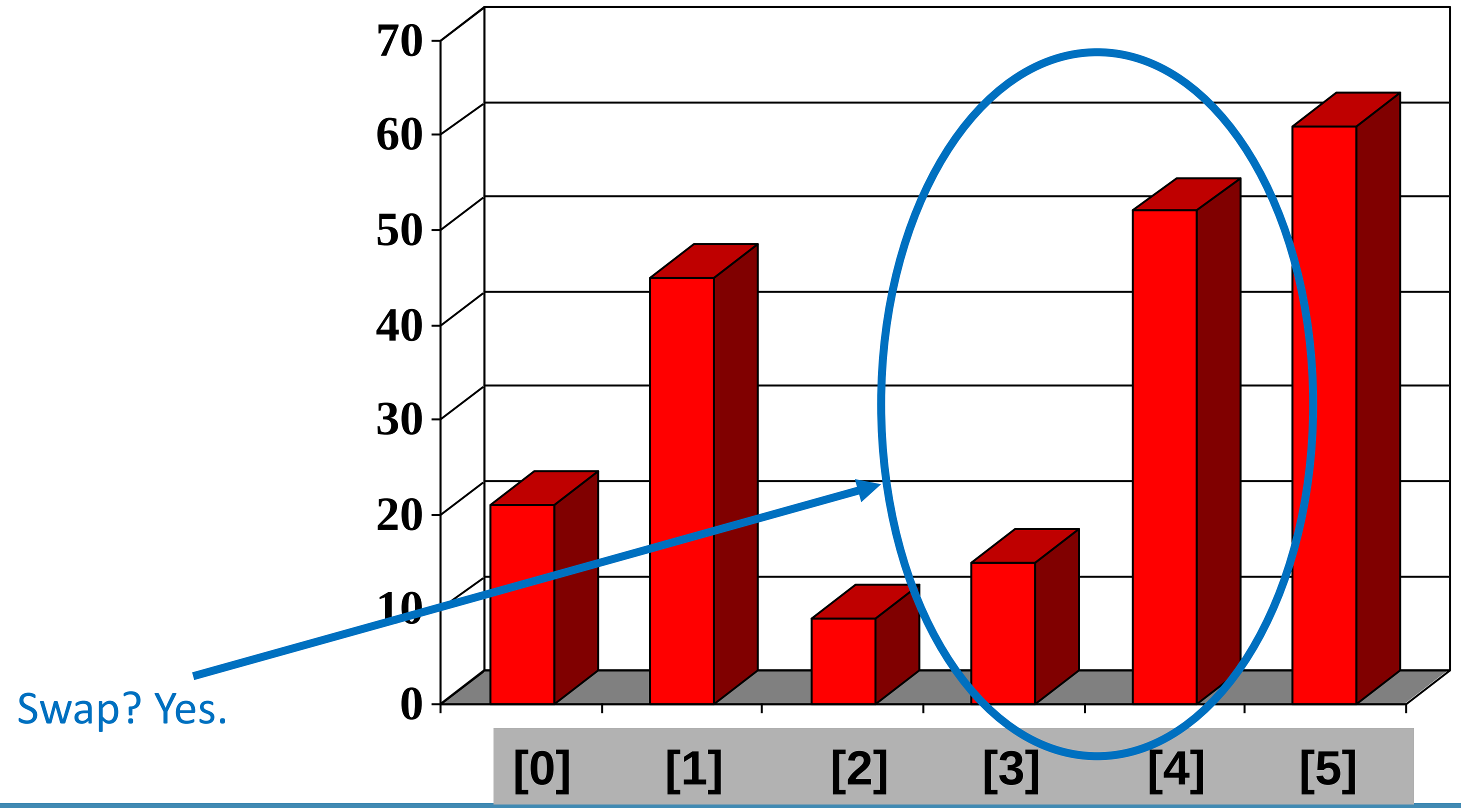
The Bubble Sort Algorithm

Repeat.



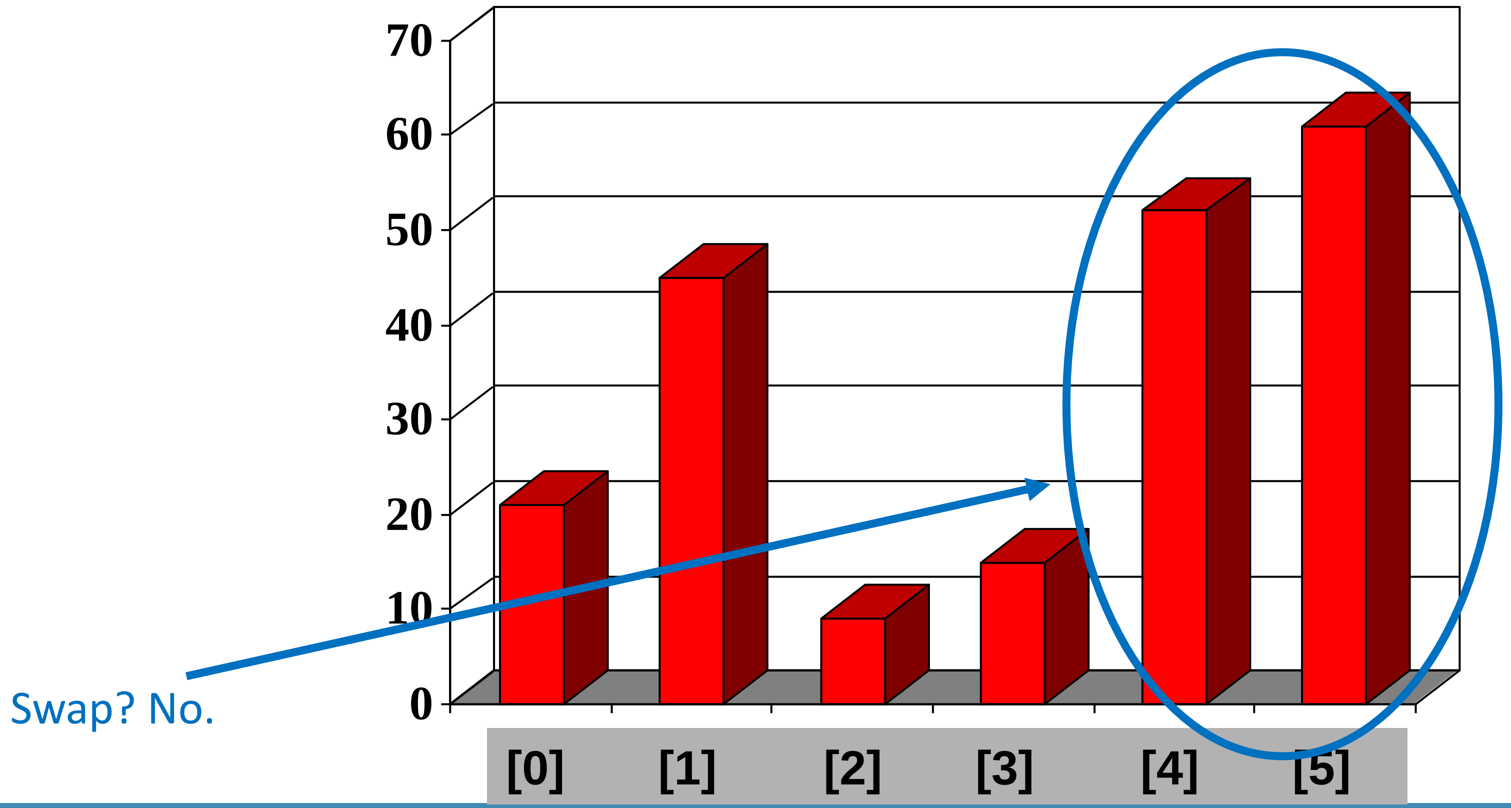
The Bubble Sort Algorithm

Repeat.



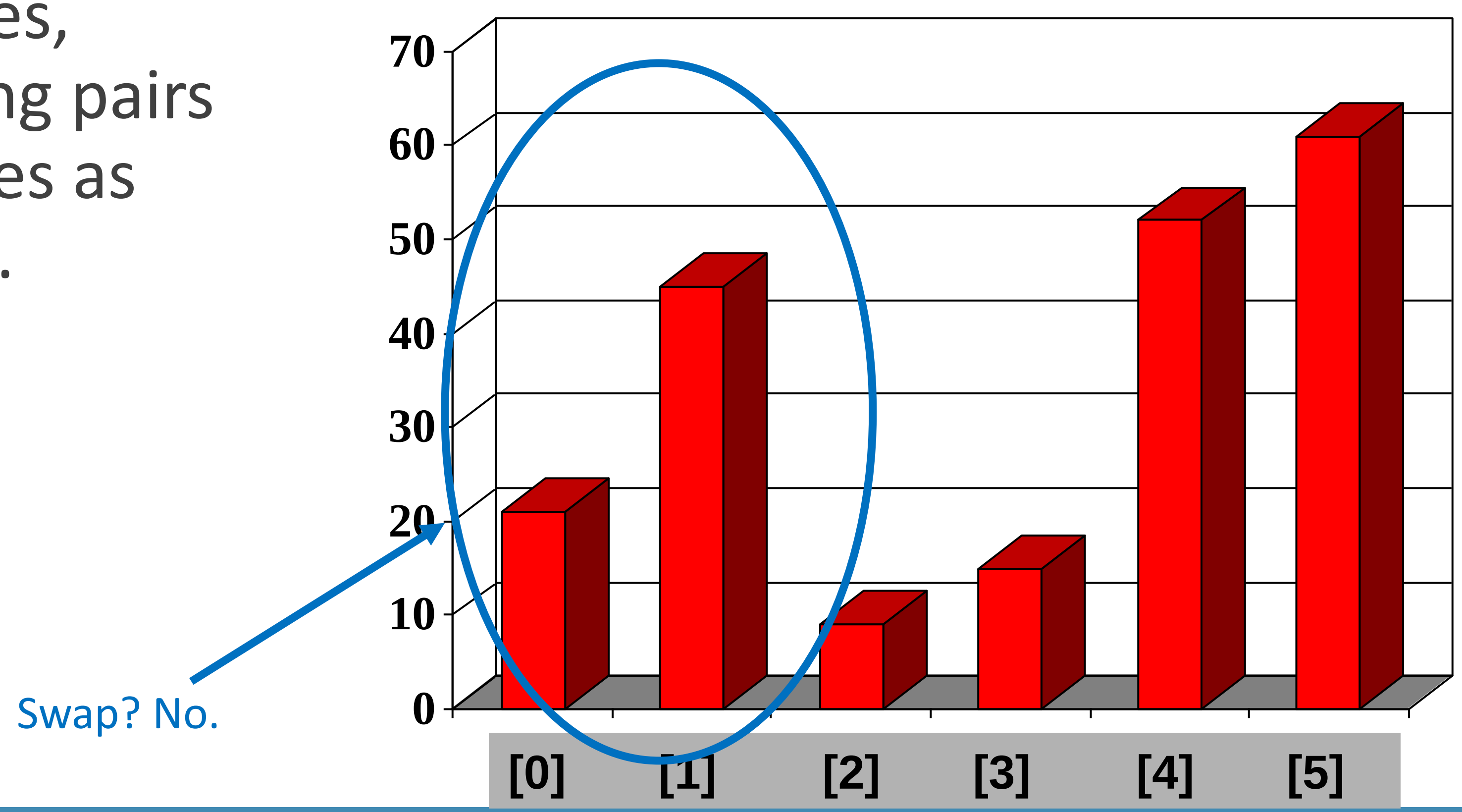
The Bubble Sort Algorithm

Repeat.



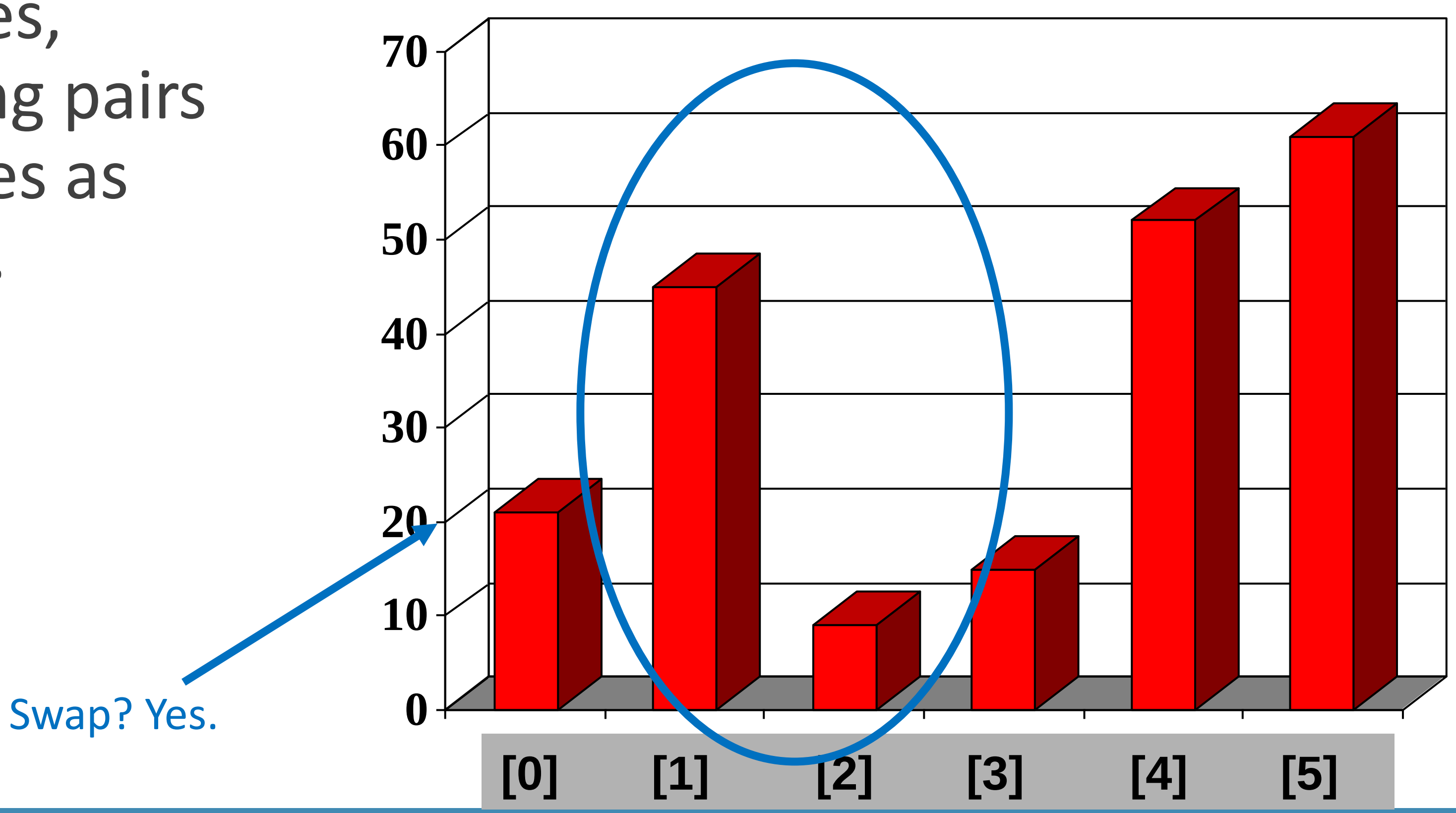
The Bubble Sort Algorithm

Loop over array n-1 times, swapping pairs of entries as needed.



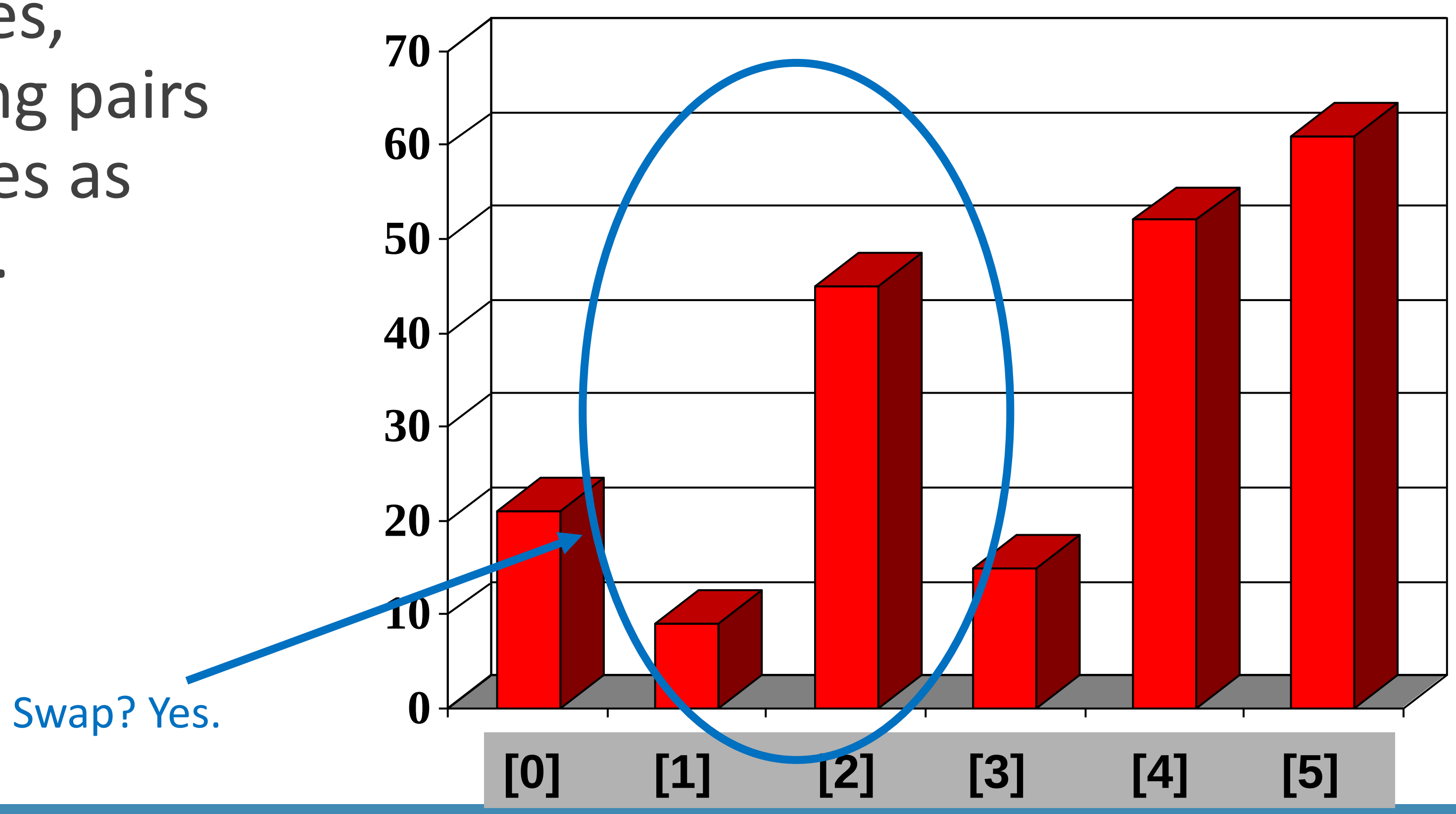
The Bubble Sort Algorithm

Loop over array n-1 times, swapping pairs of entries as needed.



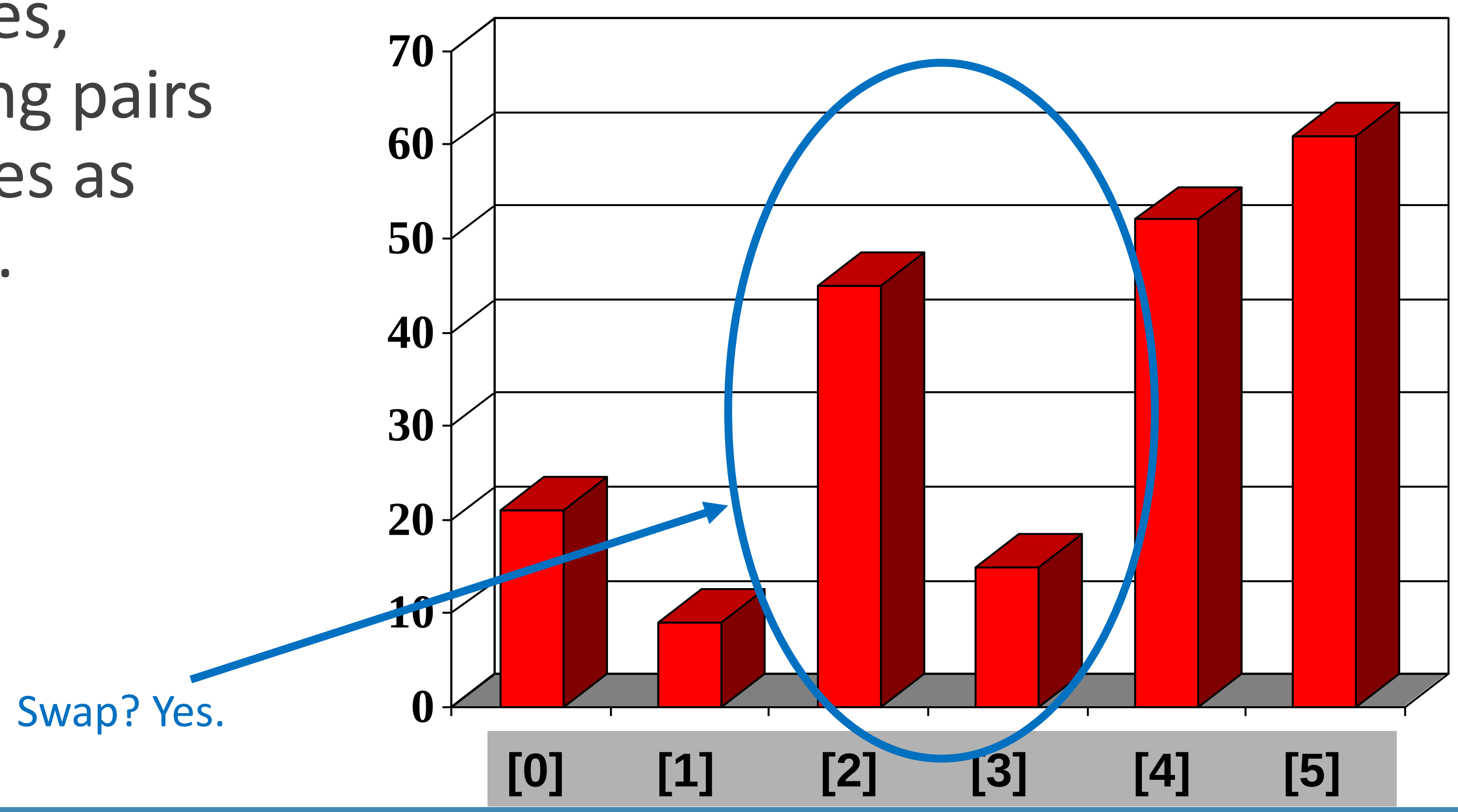
The Bubble Sort Algorithm

Loop over array n-1 times, swapping pairs of entries as needed.



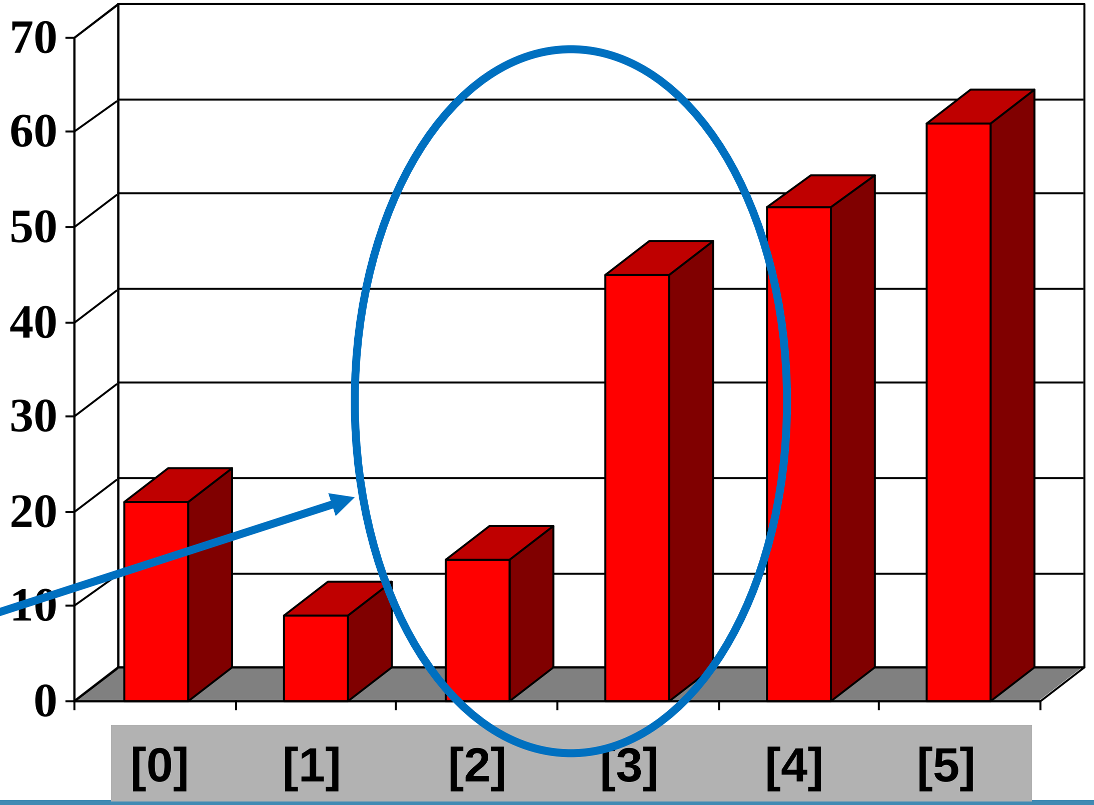
The Bubble Sort Algorithm

Loop over array n-1 times, swapping pairs of entries as needed.



The Bubble Sort Algorithm

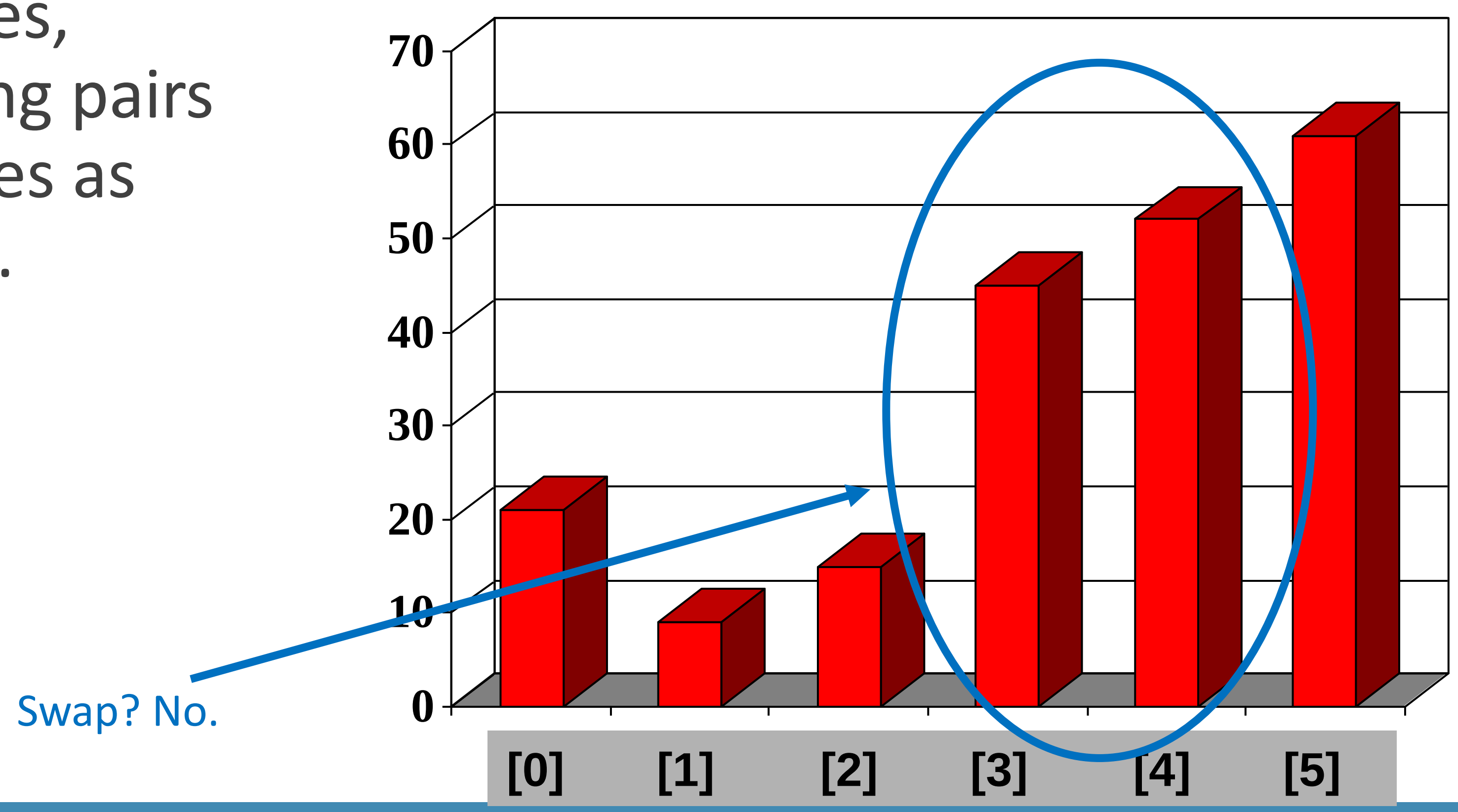
Loop over array n-1 times, swapping pairs of entries as needed.



Swap? Yes.

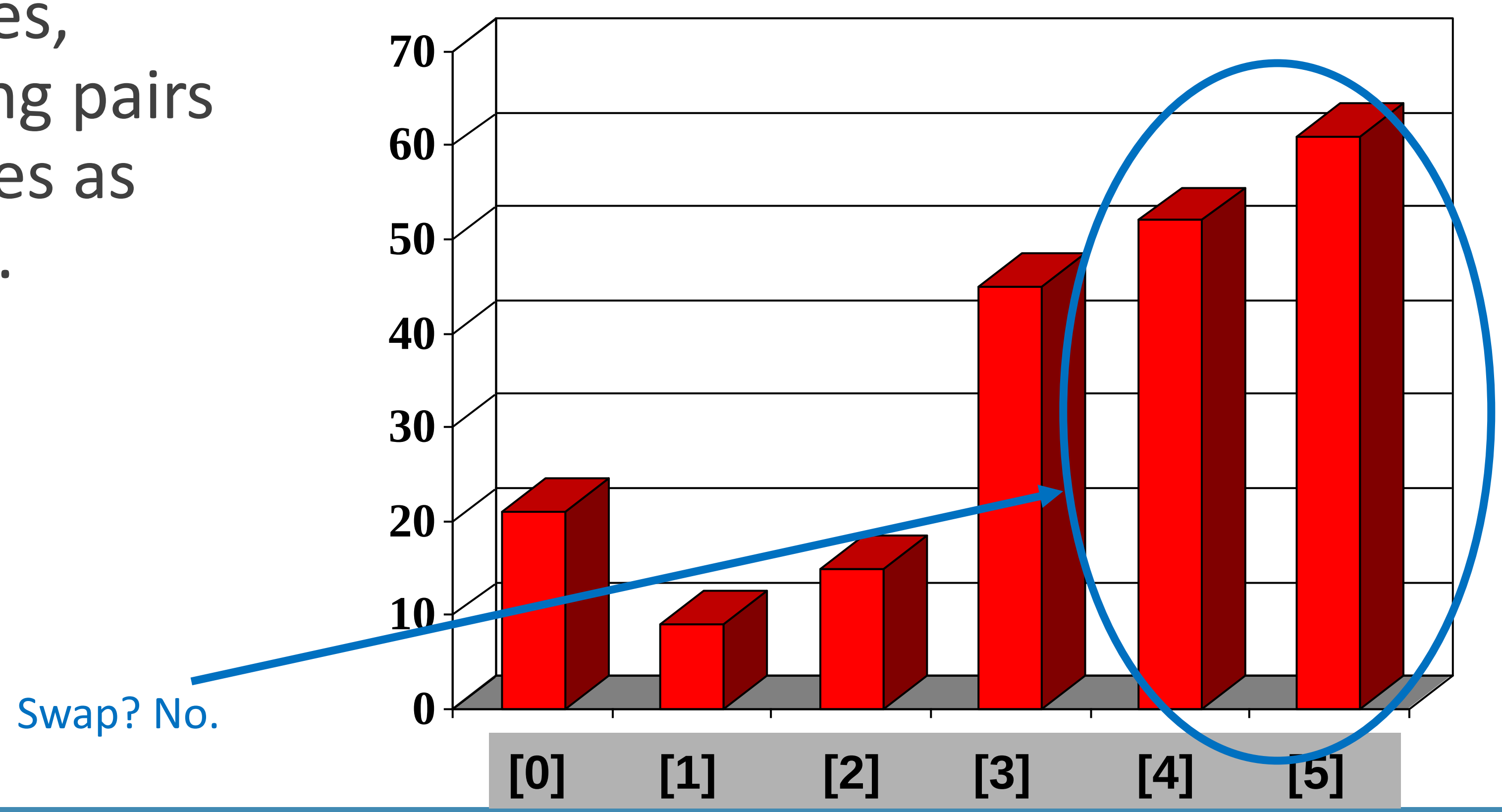
The Bubble Sort Algorithm

Loop over array n-1 times, swapping pairs of entries as needed.



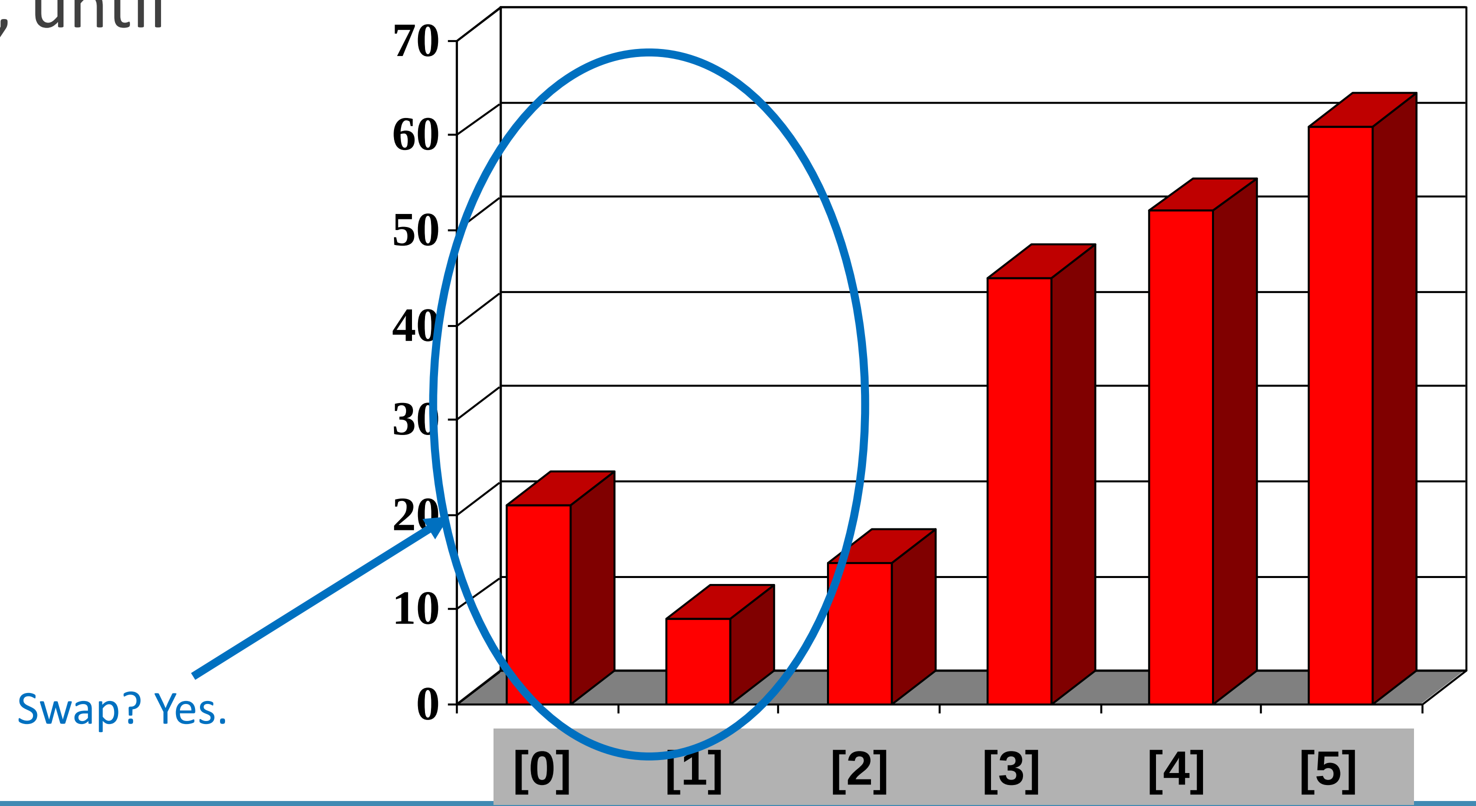
The Bubble Sort Algorithm

Loop over array n-1 times, swapping pairs of entries as needed.



The Bubble Sort Algorithm

Continue looping, until done.



```
template <class Item>
void bubble_sort(Item data[ ], size_t n){
    size_t i, j;
    Item temp;

    if(n < 2) return;  // nothing to sort!!

    for(i = 0; i < n-1; ++i)
    {
      for(j = 0; j < n-1;++j)
          if(data[j] > data[j+1])      // if out of order, swap!
          {
                temp = data[j];
                data[j] = data[j+1];
                data[j+1] = temp;
          }
    }
}
```

```
template <class Item>
void bubble_sort(Item data[ ], size_t n){
    size_t i, j;
    Item temp;
    bool swapped = true;

    if(n < 2) return;  // nothing to sort!!
    for(i = 0; swapped and i < n-1; ++i)
    {  // if no elements swapped in an iteration,
       // then elements are in order: done!
       for(swapped = false, j = 0; j < n-1;++j)
           if(data[j] > data[j+1])      // if out of order, swap!
           {
                   temp = data[j];
                   data[j] = data[j+1];
                   data[j+1] = temp;
             swapped = true;
            }
     }
}
```

Bubble Sort Time Analysis

In O-notation, what is:

- Worst case running time for *n* items?
- Average case running time for *n* items?

Steps of algorithm:

```
for i = 0 to n-1
   for j =0 to n-2
            if key[j] > key[j+1] then swap
   if no elements swapped in this pass through array, done.
   otherwise, continue
```

Bubble Sort Time Analysis

In O-notation, what is:

- Worst case running time for *n* items?
- Average case running time for *n* items?

Steps of algorithm:

```
for i = 0 to n-1                                              O(n)
   for j =0 to n-2                       O(n)
          if key[j] > key[j+1] then swap
   if no elements swapped in this pass through array, done.
   otherwise, continue
```

Timing and Other Issues

- Selection Sort, Insertion Sort, and Bubble Sort all have a worst-case time of $O(n^2)$, making them impractical for large arrays.
- But they are easy to program, easy to debug.
- Insertion Sort 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.

Next time: Merge Sort, Quick Sort, and Radix Sort.

LECTURE 5

Merge Sort

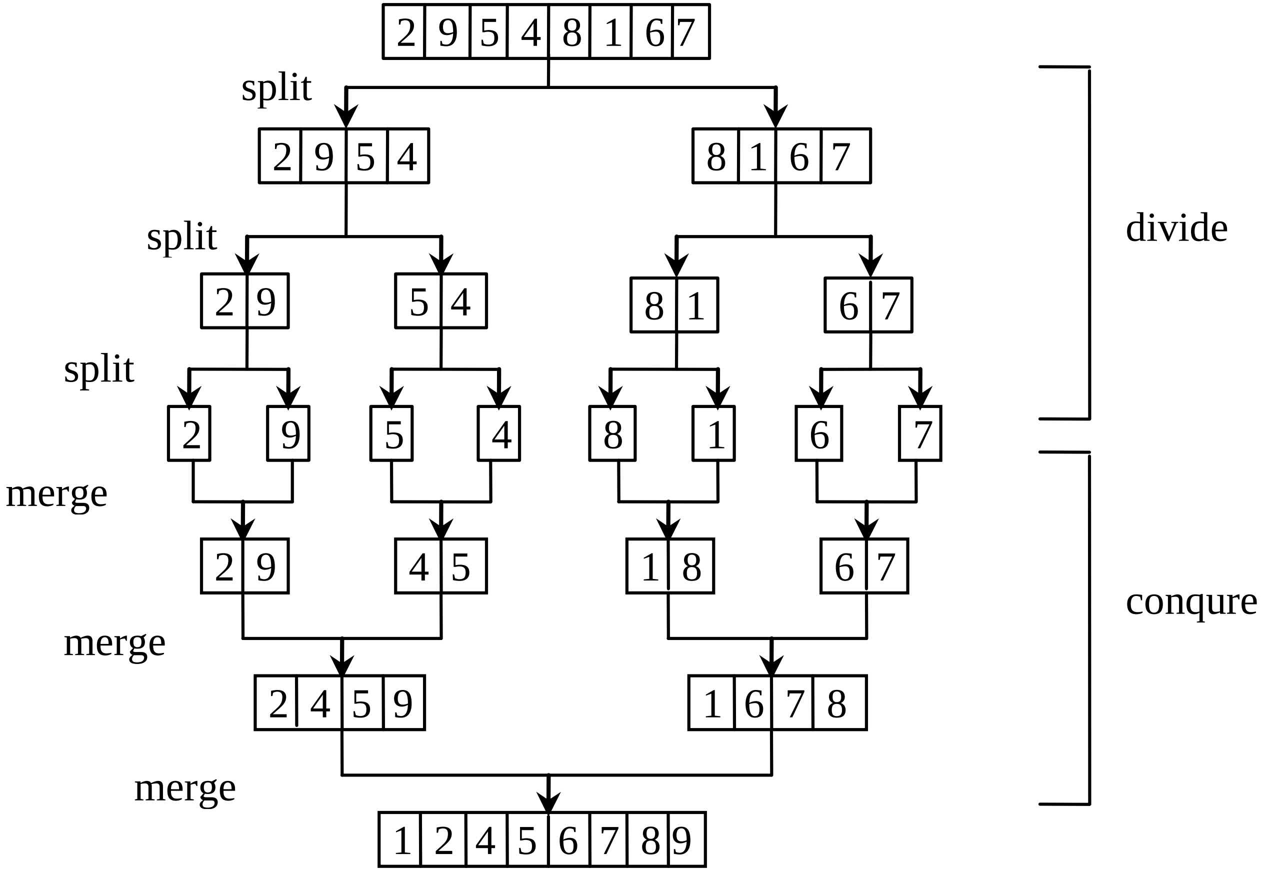
Merge Sort

The merge sort algorithm can be described recursively as follows: The algorithm divides the array into two halves and applies a merge sort on each half recursively. After the two halves are sorted, merge them.

Merge Sort Algorithm

```
public static void mergeSort(int[] list) {
    if (list.length > 1) {
        int low = 0, high = list.length-1;
        int mid = (low+high)/2;
        mergeSort(list[0 ... mid+ 1]);
        mergeSort(list[mid + 1 ... list.length]);
        merge list[0 ... mid+ 1] with
        list[mid + 1 ... list.length];
    }
} // First half will be a little bit longer (if length is odd)
```

Merge Sort



2 9 5 4 8 1 6 7
split
2 9 5 4
8 1 6 7
split
2 9
5 4
8 1
6 7
split
2
9
5
4
8
1
6
7
merge
2 9
4 5
1 8
6 7
merge
2 4 5 9
1 6 7 8
merge
1 2 4 5 6 7 8 9
divide
conqure

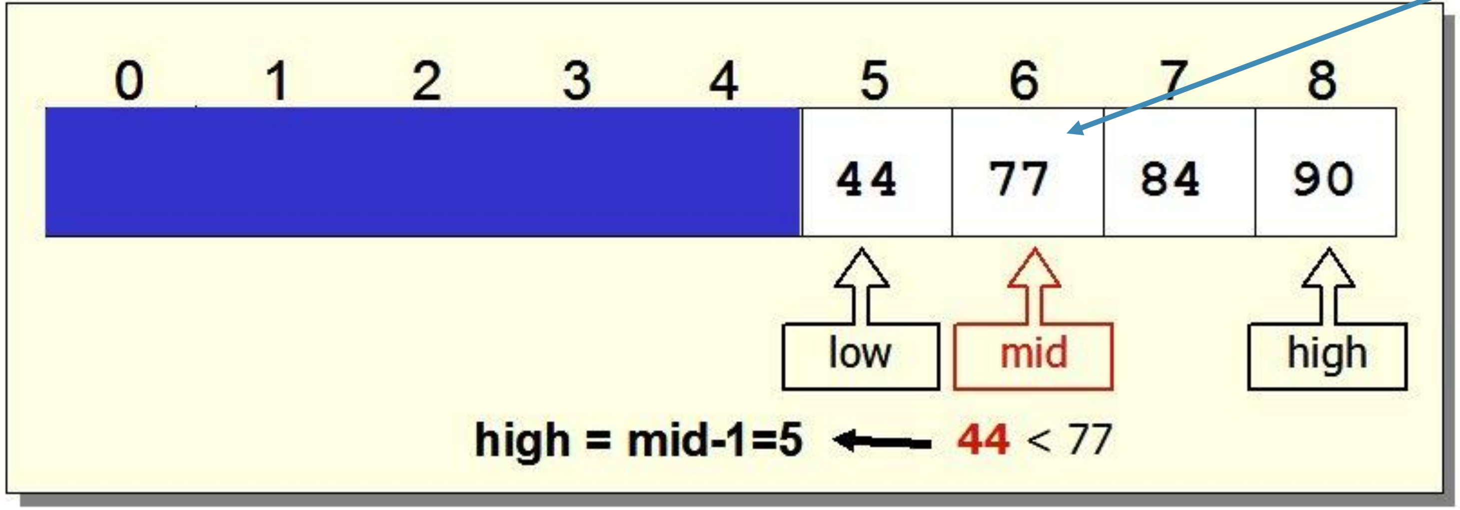
We use the same low, mid, high system as the Binary Search

	low	high	mid
#1	0	8	4
#2	5	8	6

search(44)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

mid belongs to first half



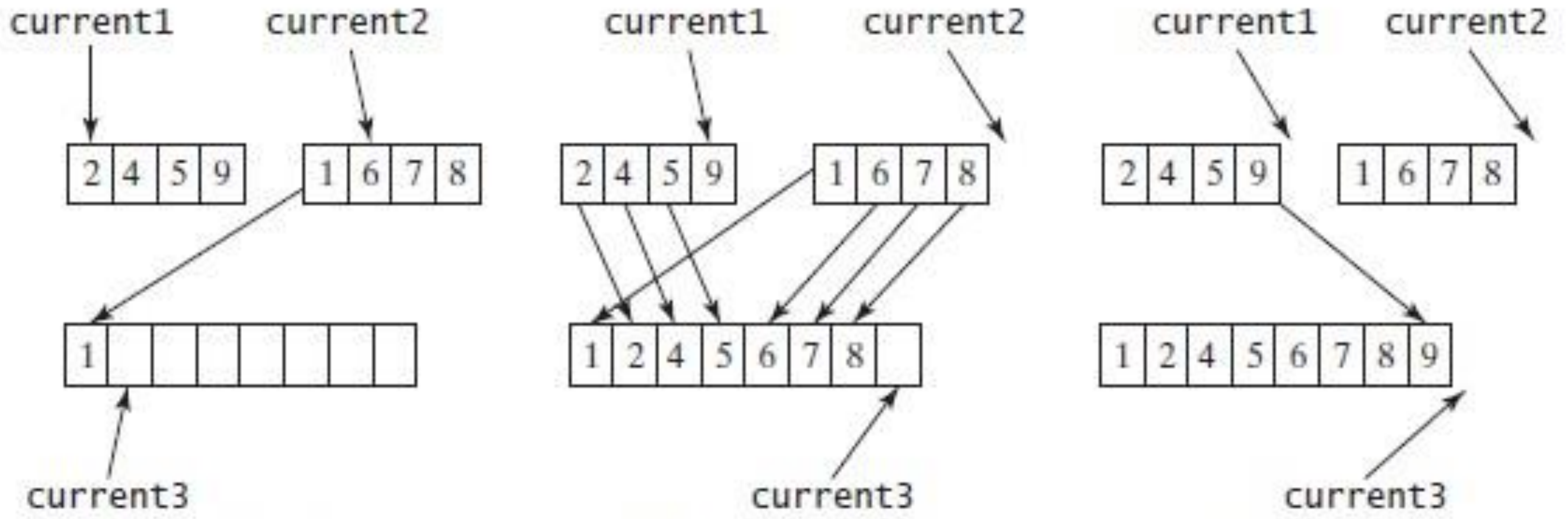
Top Level mergeSort Method

```
public static void mergeSort(int[] list) {
  if (list.length > 1) {
    // Merge sort the first half
    int low = 0, high = list.length-1;
    int mid = (low+high)/2;
    int[] firstHalf = new int[mid+1];
    System.arraycopy(list, 0, firstHalf, 0, mid+1);
    mergeSort(firstHalf);

    // Merge sort the second half
    int secondHalfLength = list.length - (mid+1);
    int[] secondHalf = new int[secondHalfLength];
    System.arraycopy(list, mid+1,
      secondHalf, 0, secondHalfLength);
    mergeSort(secondHalf);

    // Merge firstHalf with secondHalf into list
    merge(firstHalf, secondHalf, list);
  }
}
```

Merge



(a) After moving 1 to `temp`

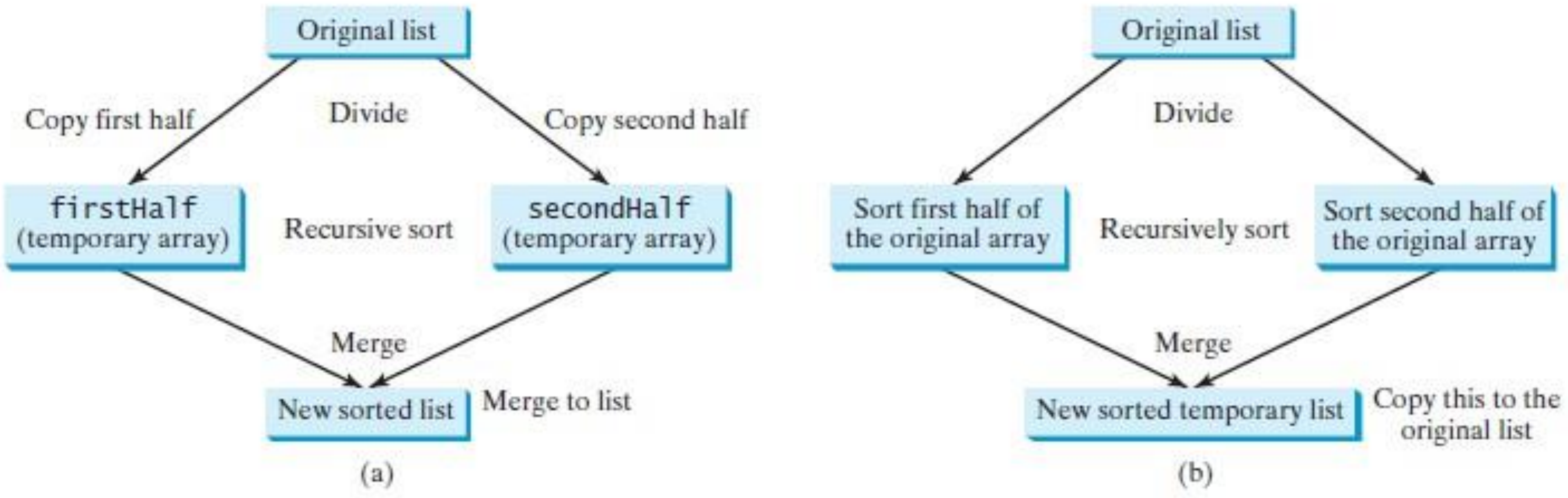
(b) After moving all the elements in `list2` to `temp`

(c) After moving 9 to `temp`

merge Method

```
int *merge(int *a, int *b, int na, int nb){
    int *c = new int[na+nb];
    int p=0, q=0, r=0;
    while (p<na && q<nb){
        if (a[p]<b[q]) c[r++] = a[p++];
        else c[r++] = b[q++];
    }
    while (p<na) c[r++] = a[p++];
    while (q<nb) c[r++] = b[q++];
    return c;
}
```

Data Structures



Merge Sort Time

Let T(n) denote the time required for sorting an array of n elements using merge sort. Without loss of generality, assume n is a power of 2. The merge sort algorithm splits the array into two subarrays, sorts the subarrays using the same algorithm recursively, and then merges the subarrays. So,

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + mergetime$$

Merge Sort Time

The first *T(n/2)* is the time for sorting the first half of the array and the second *T(n/2)* is the time for sorting the second half. To merge two subarrays, it takes at most *n-1* comparisons to compare the elements from the two subarrays and *n* moves to move elements to the temporary array. So, the total time is *2n-1*. Therefore,

$$T(n) = 2T(\frac{n}{2}) + 2n - 1 = 2(2T(\frac{n}{4}) + 2\frac{n}{2} - 1) + 2n - 1 = 2^2 T(\frac{n}{2^2}) + 2n - 2 + 2n - 1$$

$$= 2^k T(\frac{n}{2^k}) + 2n - 2^{k-1} + \ldots + 2n - 2 + 2n - 1$$

$$= 2^{\log n} T(\frac{n}{2^{\log n}}) + 2n - 2^{\log n - 1} + \ldots + 2n - 2 + 2n - 1$$

$$= n + 2n \log n - 2^{\log n} + 1 = 2n \log n + 1 = O(n \log n)$$

Demonstration Program

MERGESORT.CPP

Demo Program:

MergeSort.cpp

- MergeSort’s core is Divide and merge on integration.
- Arrays.sort uses merge sort algorithm. Use extra memory space to cut time.

MergeSort.Cpp

[3, 7, 1, 9, 5, 4, 8, 10, 2, 6]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```
int *sort(int *a, int len){
    if (len<=1) return a;
    int low = 0;
    int high = len-1;
    int mid = (low+high)/2;
    int esize = mid+1;
    int *e = new int[esize];
    int fsize = len-mid-1;
    int *f = new int[fsize];
    int p =0;
    for (int i=0; i<esize; i++){
        e[i] = a[p++];
    }
    for (int i=0; i<fsize; i++){
        f[i] = a[p++];
    }
    e = sort(e, esize);
    f = sort(f, fsize);
    a = merge(e, f, esize, fsize);
    delete[] e;
    delete[] f;
    return a;
}
```