Sorting

Simple Sorting Methods

Introduction

- Common problem: sort a **list** of values, starting from lowest to highest (ascending order), or from highest to lowest (descending order).
- Example Lists:
  - Exam scores
  - Words of dictionary in alphabetical order
  - Student names listed alphabetically
  - Student records sorted by ID#
- Generally, we are given a list of records that have *keys*. These keys (*sort fields*) are used to define an ordering of the elements in the list.

Contiguous vs. Non-Contiguous list

- The list may be:
  - contiguous and randomly accessible (like an array), or
  - dispersed and only sequentially accessible (like a linked list).

- The implementation details will differ in both cases, but the same logic applies.

Internal vs. External Sorting

- In an internal sort, the list of elements is small enough to be maintained entirely in physical memory for the duration of the sort.

- In an external sort, the list of elements will not fit entirely into physical memory at once. In that case, the elements are kept in disk files and only a selection of them are made resident in physical memory at any given time.

- We will consider only internal sorting in this course.

Internal Sorting Analysis

- When analyzing the performance of various sorting algorithms, there are two factors:
  - The number of comparisons that are required
  - The number of element moves that are required
- Both worst-case and average-case performance measures are significant.

Java Implementation of Sorting

- Use Java generics to implement a generic function for sorting a list of elements of any class.
- The class of the elements to be sorted must either:
  - Implement the Comparable interface or
  - Provide a suitable element Comparator.
    (See: the java.util.Comparator interface)

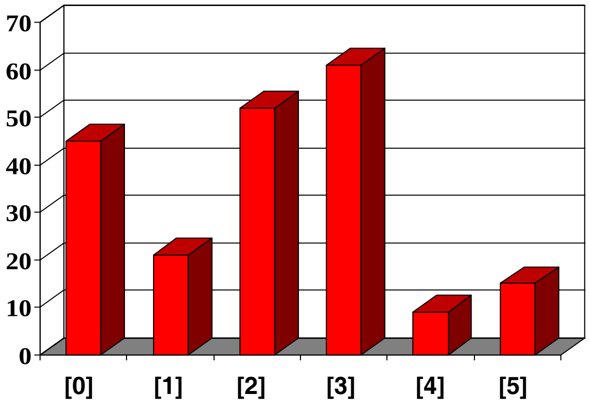
Quadratic Sorting Algorithms

- The Problem:
  We are given a list of *n* comparable elements to sort.

- There are a number of simple sorting algorithms whose worst and average case time performance is quadratic O(n^2):
  - Selection sort
  - Insertion sort
  - Bubble sort

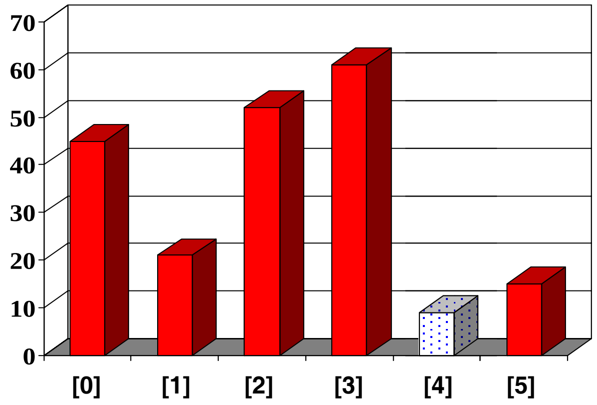
Sorting an Array of Integers

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



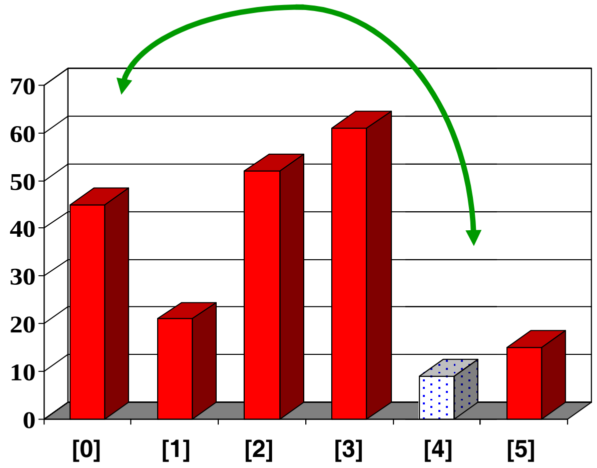
1. The Selection Sort Algorithm

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



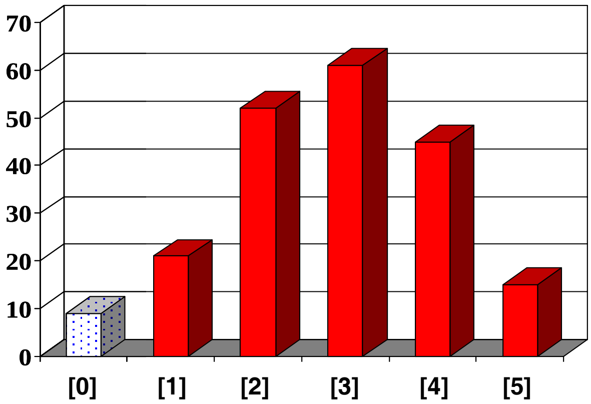
1. The Selection Sort Algorithm

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

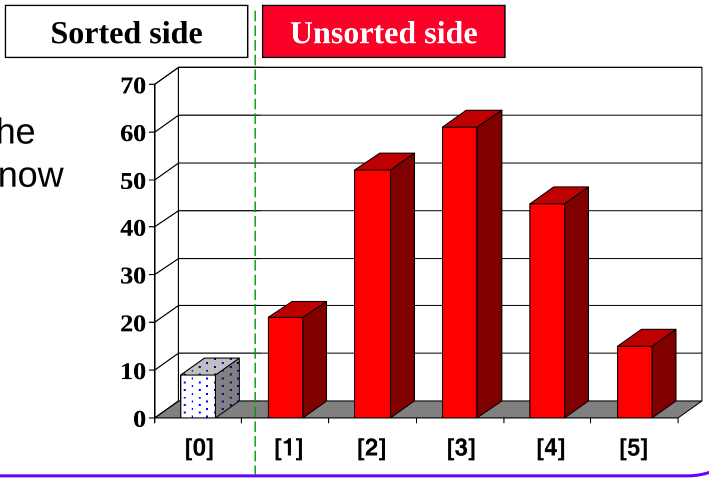


1. The Selection Sort Algorithm

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

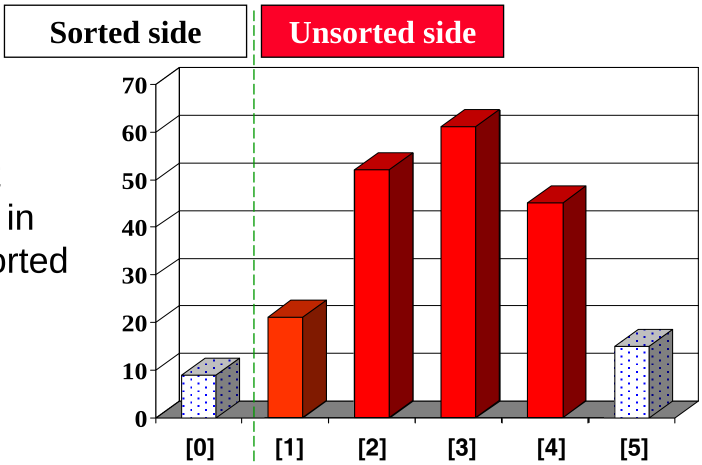


1. The Selection Sort Algorithm



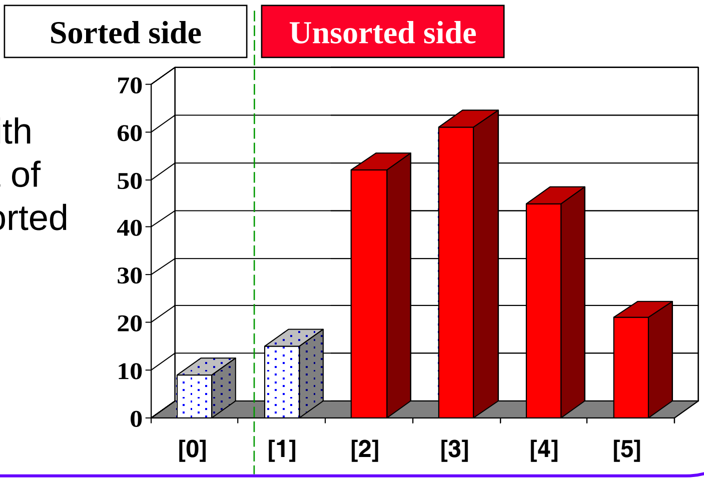
- Part of the array is now sorted.

1. The Selection Sort Algorithm



- Find the smallest element in the unsorted side.

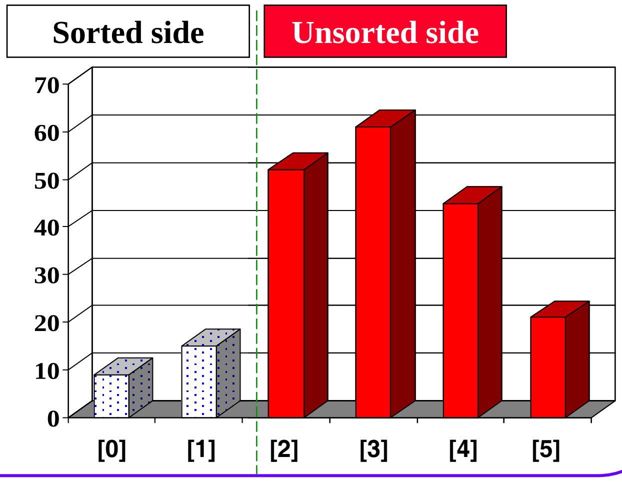
1. The Selection Sort Algorithm



- Swap with the front of the unsorted side.

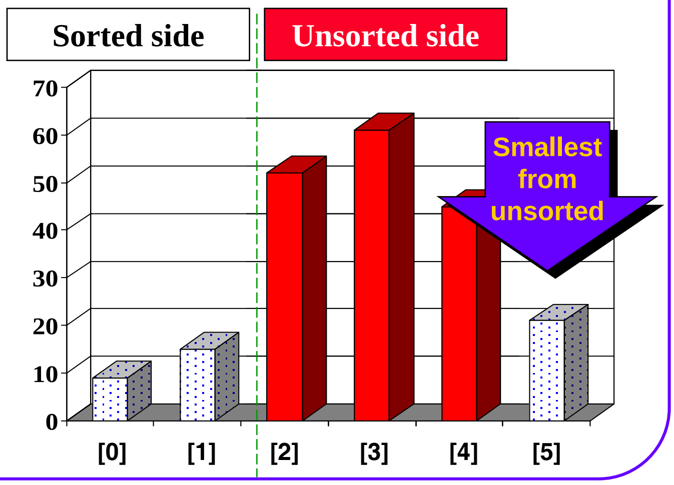
1. The Selection Sort Algorithm

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



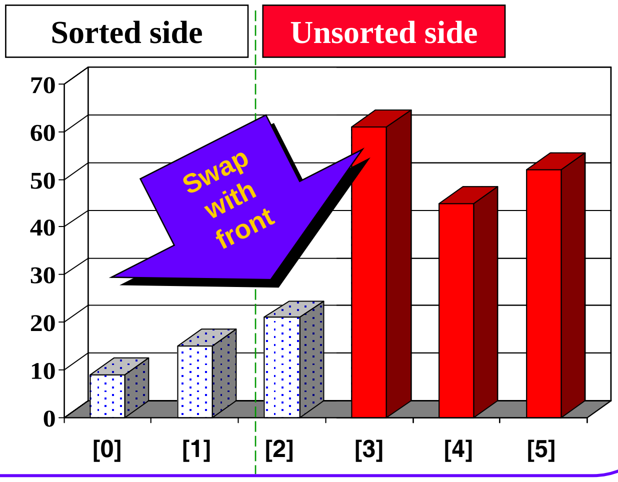
1. The Selection Sort Algorithm

- The process continues...



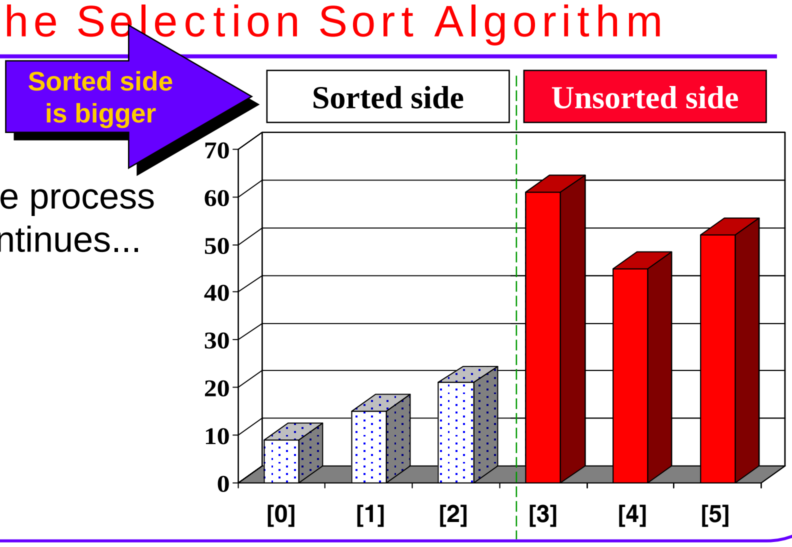
1. The Selection Sort Algorithm

- The process continues...



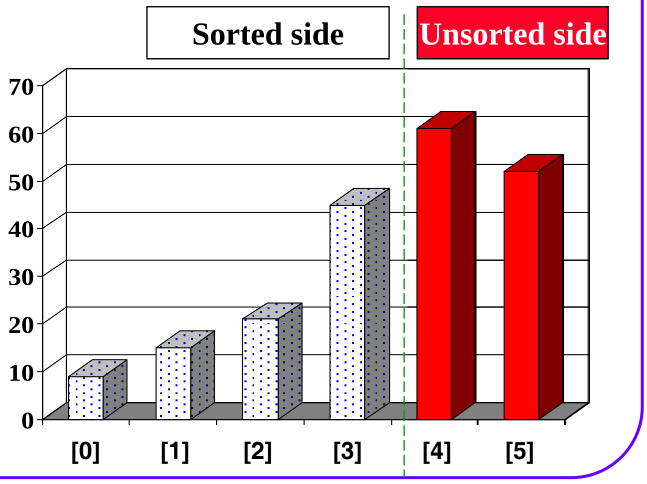
1. The Selection Sort Algorithm

- The process continues...



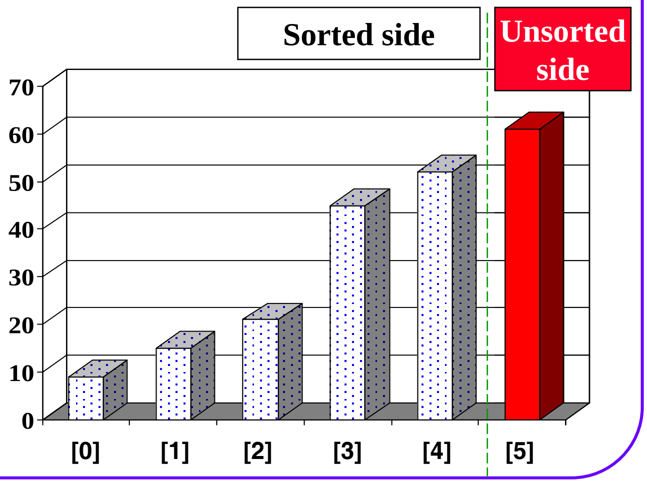
1. The Selection Sort Algorithm

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



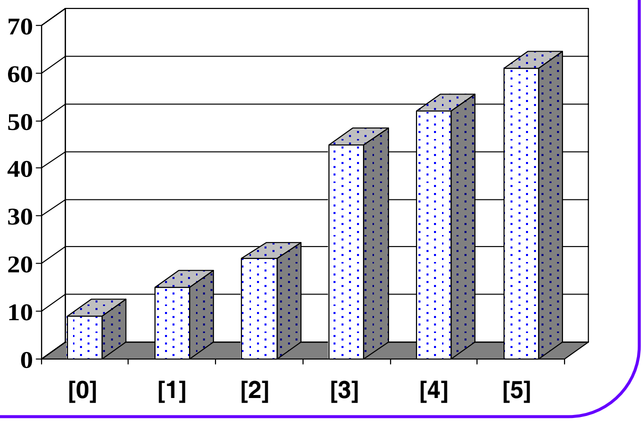
1. The Selection Sort Algorithm

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



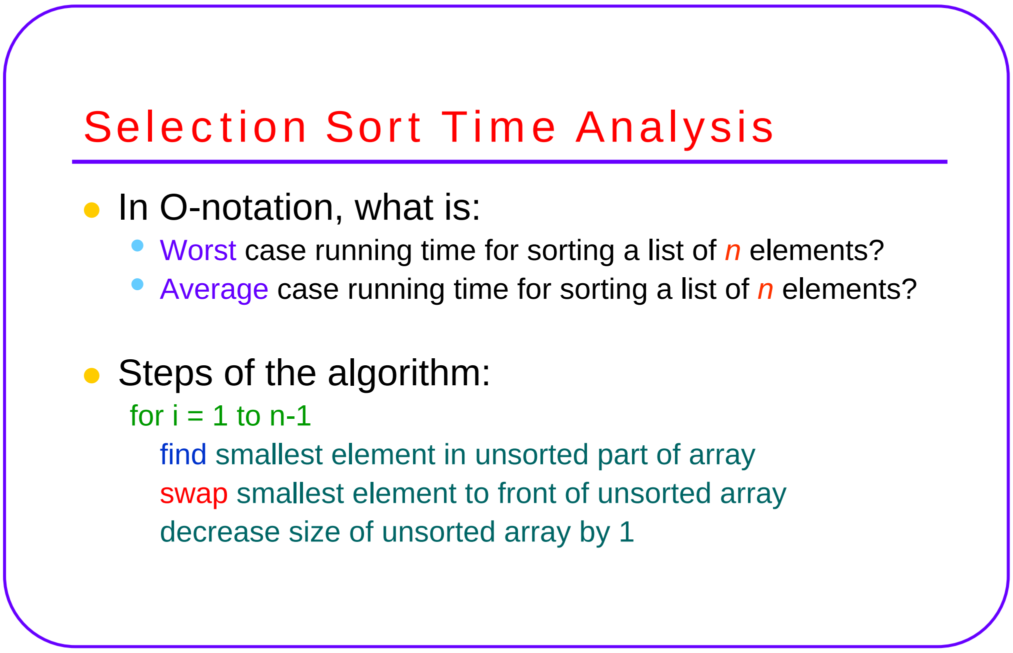
1. The Selection Sort Algorithm

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



1. The Selection Sort Function

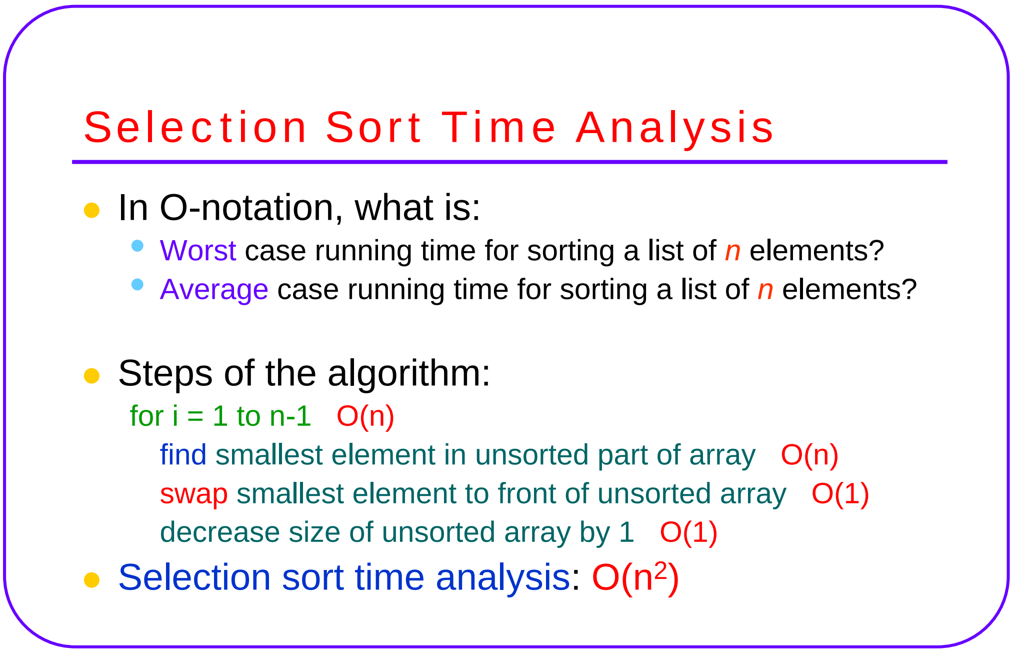
```
public void selectionSort(E[ ] data, EComparator c)  {
    int n = data.length;
    int i, j, smallest;

    if (n < 2) return;                          // nothing to sort!!
    for (i = 0; i < n - 1 ; ++i)  {
        smallest = i;                           // find smallest in the unsorted part
        for (j = i + 1; j < n; ++j)
            if (c.compare(data[smallest], data[j]) > 0)  smallest = j;

        swap (data[i], data[smallest]);         // swap it with front of unsorted part
    }
}
```



Selection Sort Time Analysis

- In O-notation, what is:
  - Worst case running time for sorting a list of *n* elements?
  - Average case running time for sorting a list of *n* elements?
- Steps of the algorithm:

```
for i = 1 to n-1
    find smallest element in unsorted part of array
    swap smallest element to front of unsorted array
    decrease size of unsorted array by 1
```

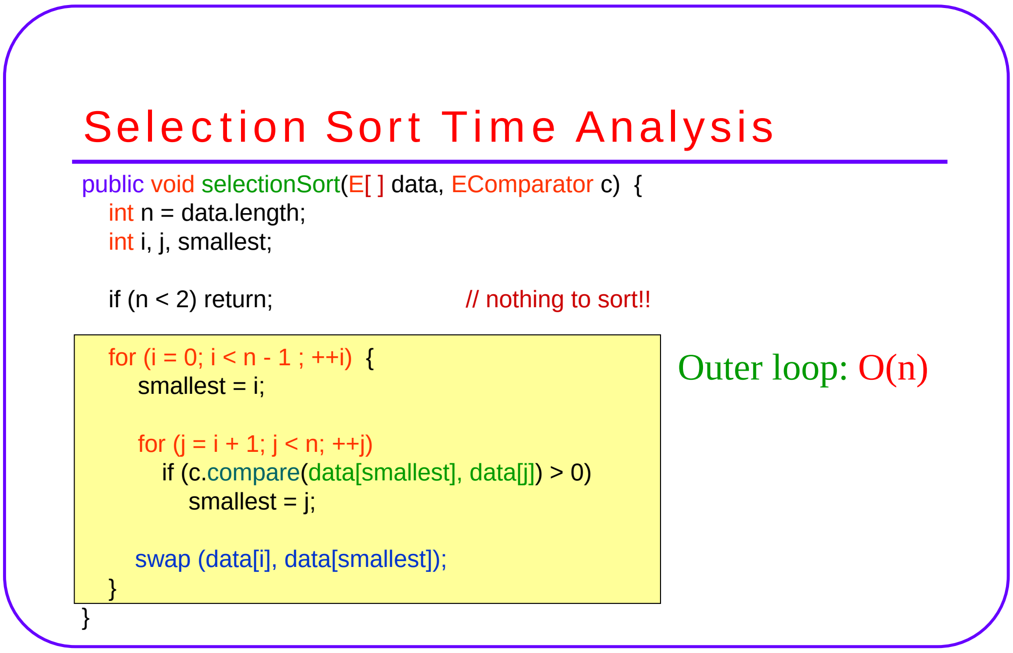


Selection Sort Time Analysis

- In O-notation, what is:
  - Worst case running time for sorting a list of *n* elements?
  - Average case running time for sorting a list of *n* elements?
- Steps of the algorithm:

```
for i = 1 to n-1   O(n)
    find smallest element in unsorted part of array   O(n)
    swap smallest element to front of unsorted array   O(1)
    decrease size of unsorted array by 1   O(1)
```

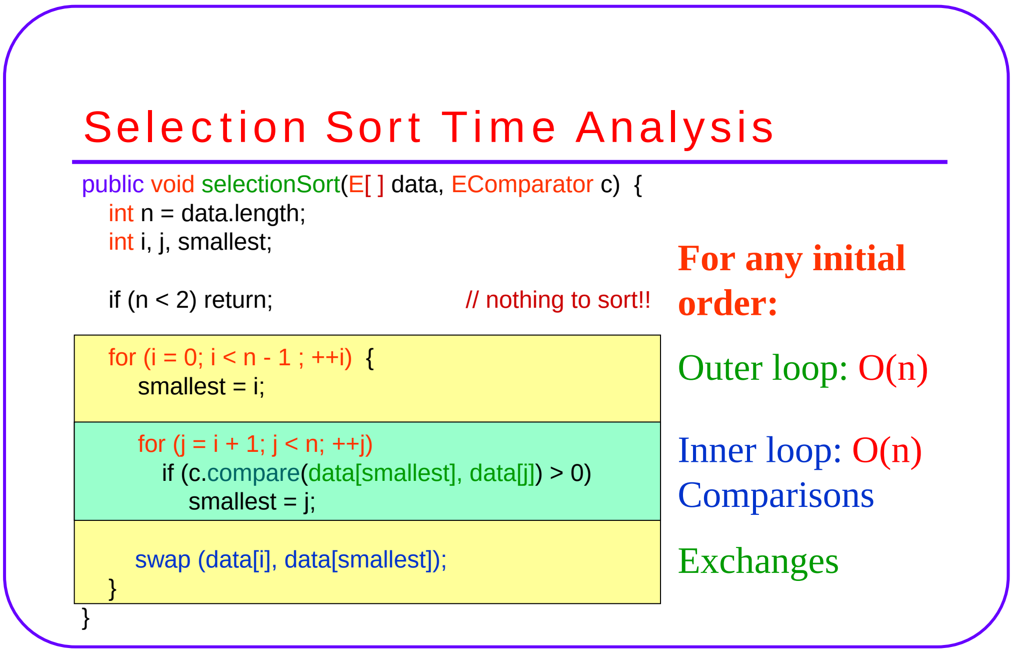
- Selection sort time analysis: $O(n^2)$



Selection Sort Time Analysis

```
public void selectionSort(E[ ] data, EComparator c)  {
    int n = data.length;
    int i, j, smallest;

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

    for (i = 0; i < n - 1 ; ++i)  {
        smallest = i;

        for (j = i + 1; j < n; ++j)
            if (c.compare(data[smallest], data[j]) > 0)
                smallest = j;

        swap (data[i], data[smallest]);
    }
}
```

Outer loop: O(n)



Selection Sort Time Analysis

```
public void selectionSort(E[ ] data, EComparator c)  {
    int n = data.length;
    int i, j, smallest;

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

    for (i = 0; i < n - 1 ; ++i)  {
        smallest = i;

        for (j = i + 1; j < n; ++j)
            if (c.compare(data[smallest], data[j]) > 0)
                smallest = j;

        swap (data[i], data[smallest]);
    }
}
```

For any initial order:

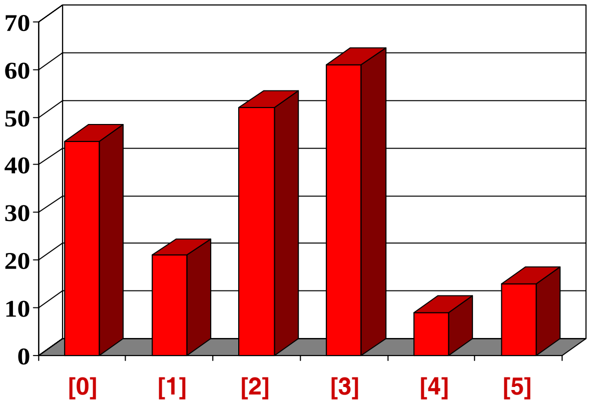
Outer loop: O(n)

Inner loop: O(n)
Comparisons

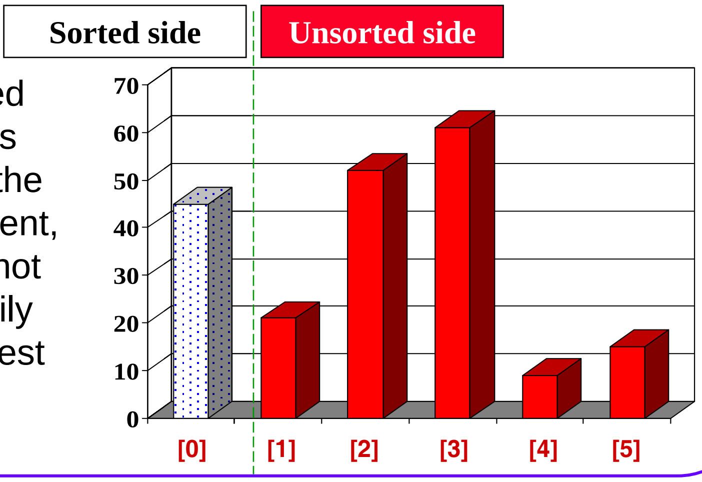
Exchanges

2. The Insertion Sort Algorithm

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



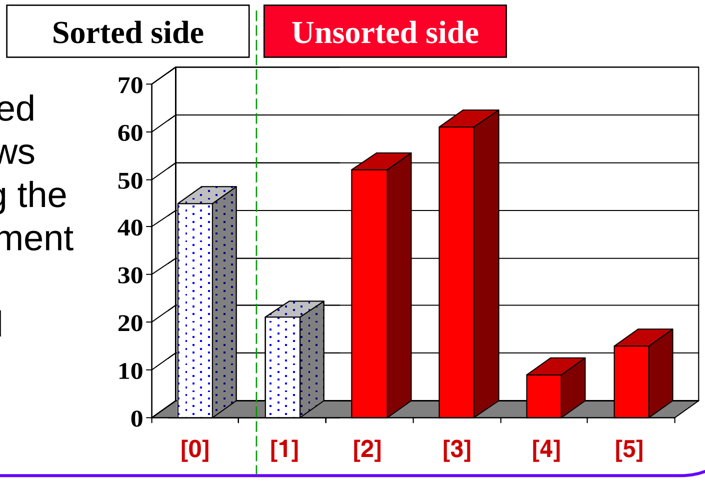
2. The Insertion Sort Algorithm



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

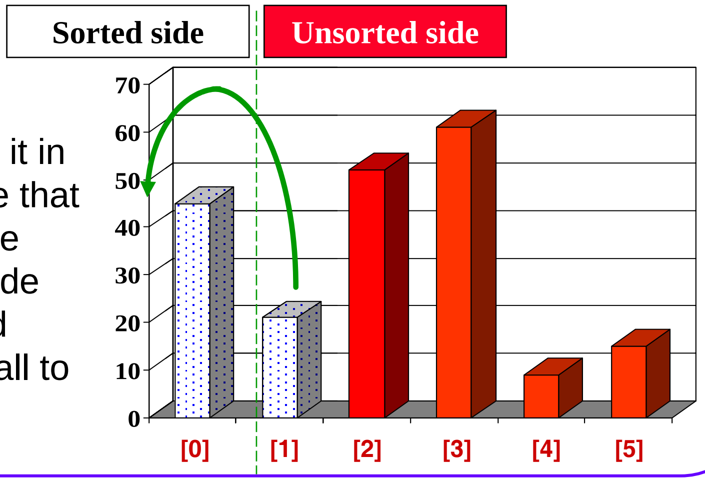
2. The Insertion Sort Algorithm

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



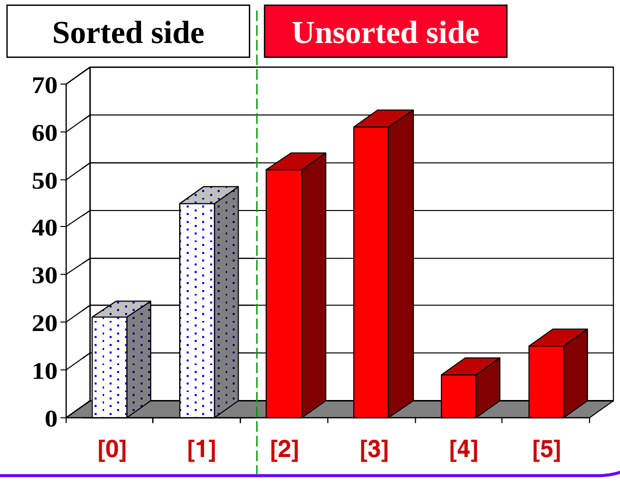
2. The Insertion Sort Algorithm

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



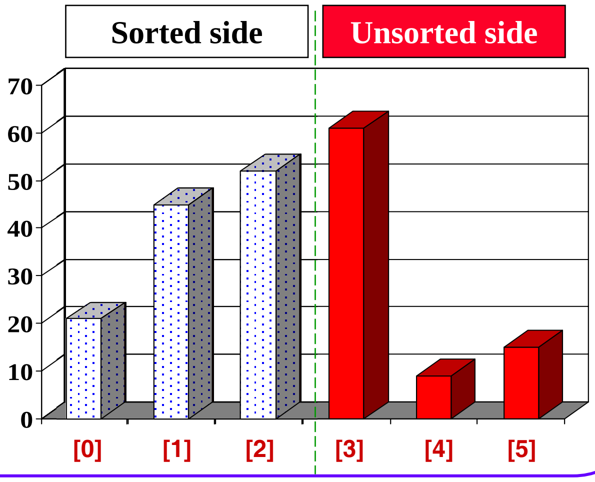
2. The Insertion Sort Algorithm

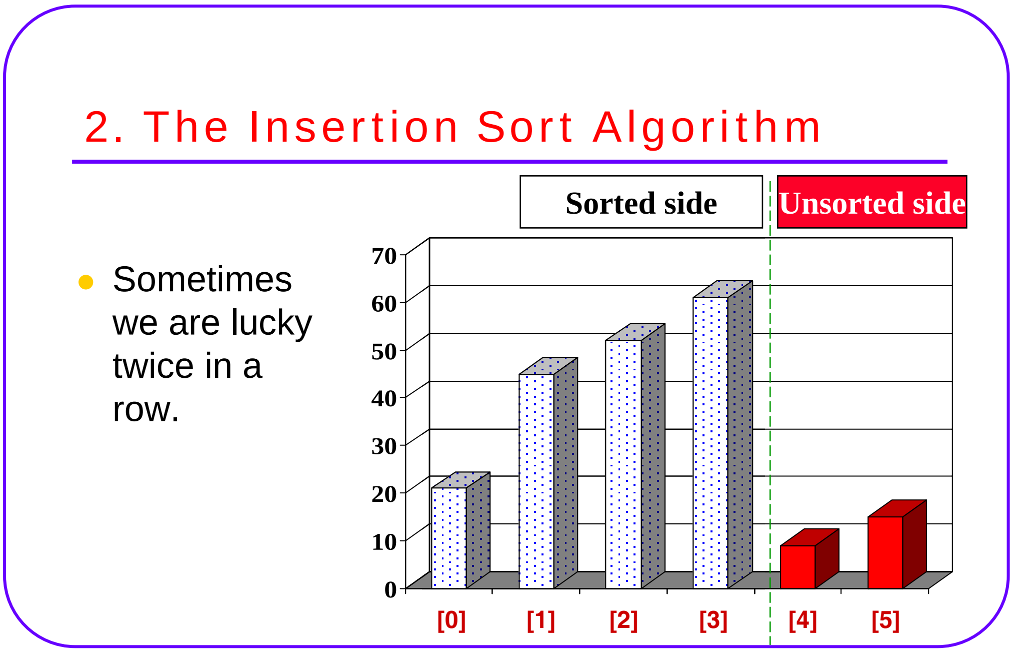
- After the insertion, the sorted side contains two elements



2. The Insertion Sort Algorithm

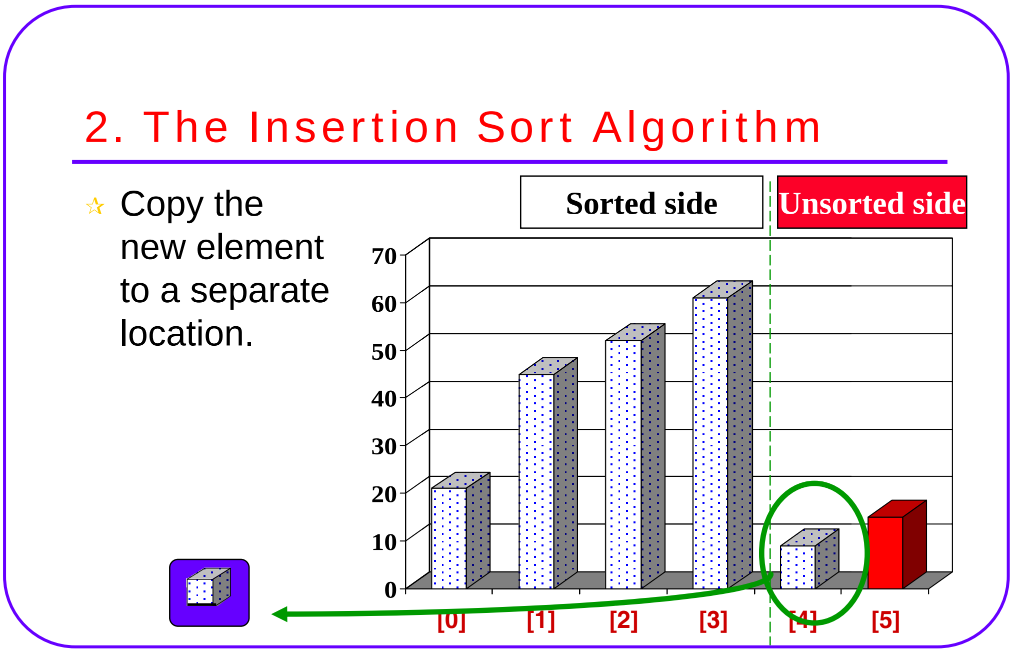
- Sometimes we are lucky and the new inserted item doesn't need to move at all.





2. The Insertion Sort Algorithm

- Sometimes we are lucky twice in a row.

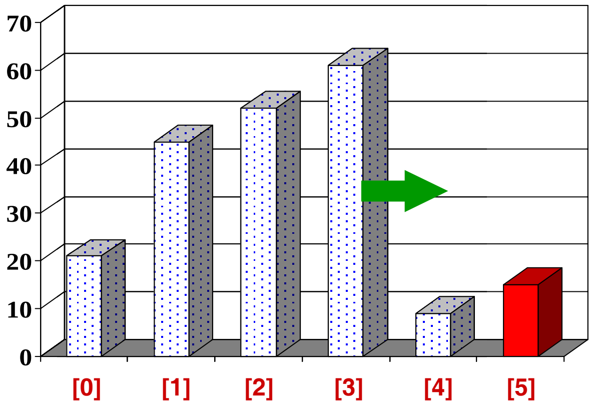


2. The Insertion Sort Algorithm

☆ Copy the new element to a separate location.

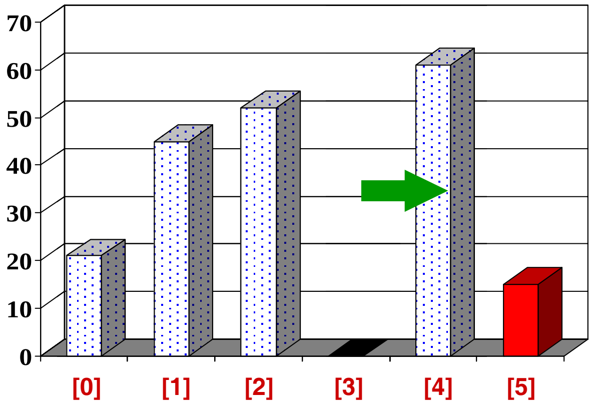
2. The Insertion Sort Algorithm

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



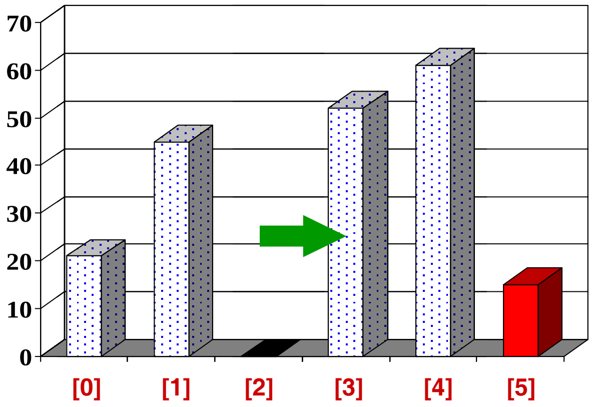
2. The Insertion Sort Algorithm

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



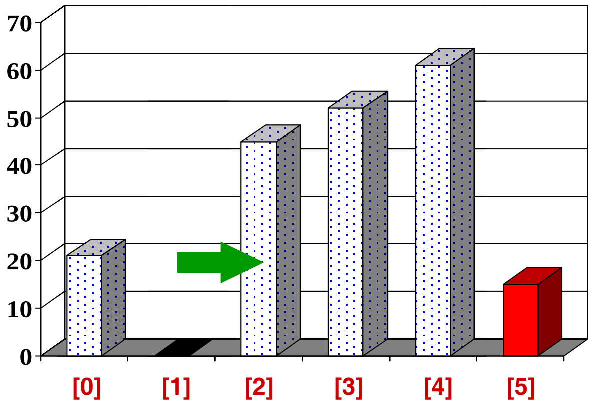
2. The Insertion Sort Algorithm

- Continue shifting elements...



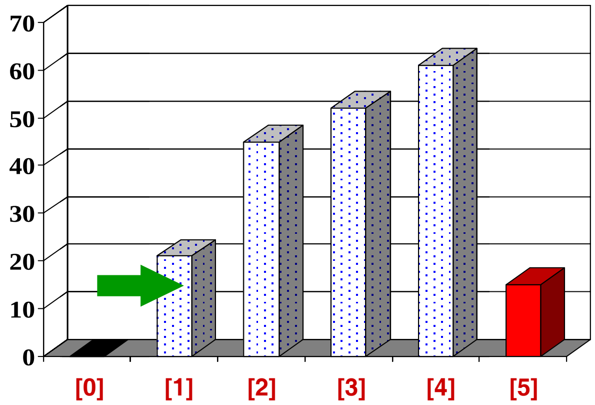
2. The Insertion Sort Algorithm

- Continue shifting elements...



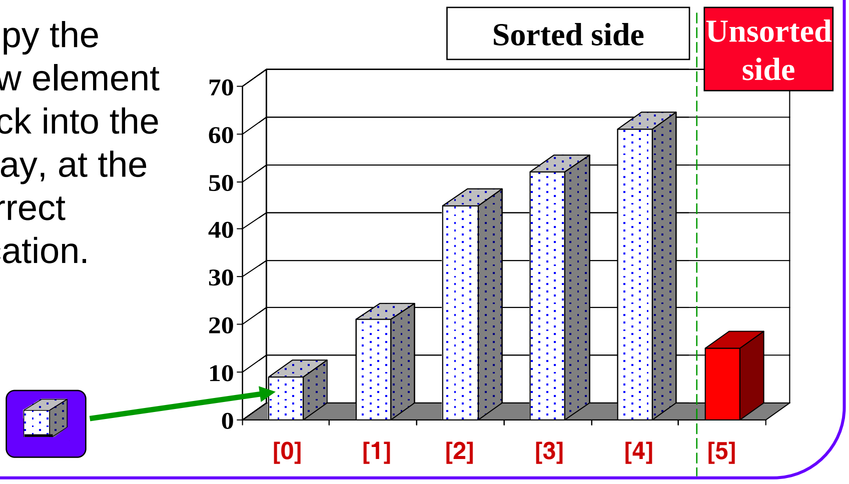
2. The Insertion Sort Algorithm

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



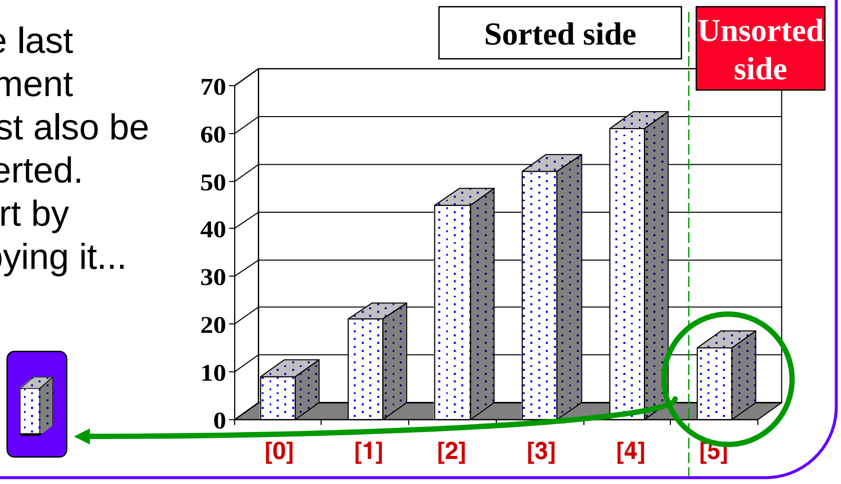
2. The Insertion Sort Algorithm

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



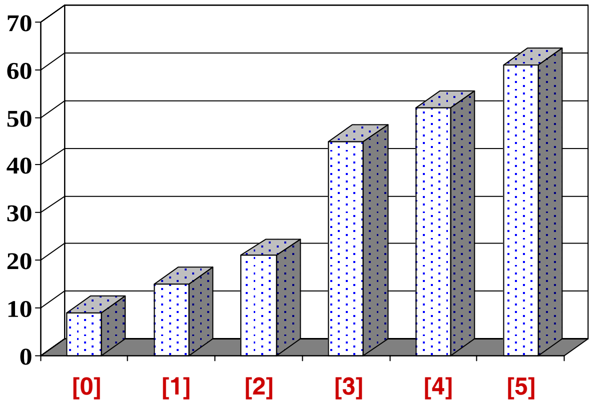
2. The Insertion Sort Algorithm

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



2. The Insertion Sort Algorithm

- The new element is inserted into the array
- We have a sorted array



2. The Insertion Sort Function

```
public void insertionSort(E[ ] data, EComparator c)  {
    int n = data.length;
    int i, j;
    E temp;

    if (n < 2) return;                        // nothing to sort!!
    for (i = 1; i < n; ++i)  {
        // take next element at front of unsorted part of array
        // and insert it in appropriate location in sorted part of array
            temp = data[i];
            for (j = i; (c.compare(data[j-1], temp) > 0) && (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 sorting a list of *n* elements?
  - Average case running time for sorting a list of *n* elements?

- Steps of the algorithm:

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

Insertion Sort Time Analysis

- In O-notation, what is:
  - Worst case running time for sorting a list of *n* elements?
  - Average case running time for sorting a list of *n* elements?

- Steps of the algorithm:

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

- Insertion sort time analysis: $O(n^2)$

Insertion Sort Time Analysis

```
public void insertionSort(E[ ] data, EComparator c)  {
    int n = data.length;
    int i, j;
    E 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; (c.compare(data[j-1], temp) > 0) && (j > 0); --j)
            data[j] = data[j-1];    // shift element forward
        data[j] = temp;
    }
}
```

Outer loop: O(n)
Exchanges

Inner loop: O(n)
Comparisons and
may be exchanges

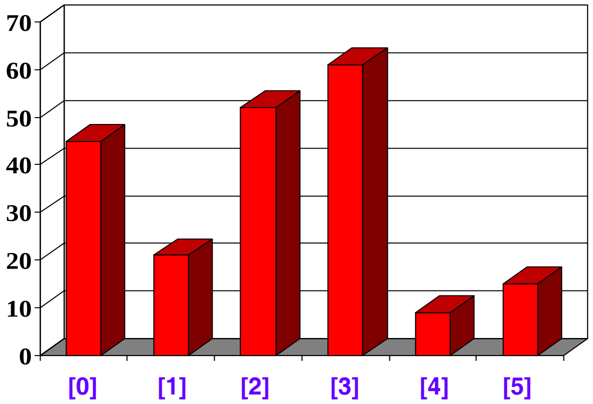
Insertion Sort Time Analysis

Initial Data Order	Comparisons	Assignments
Sorted Order	n-1 = O(n)	2(n-1) = O(n)
Random Order	n(n-1)/4 = O((n^2)/4)	n(n-1)/4 = O((n^2)/4)
Inverse Order	n(n-1)/2 = O((n^2)/2)	n(n-1)/2 = O((n^2)/2)

Empty Slide

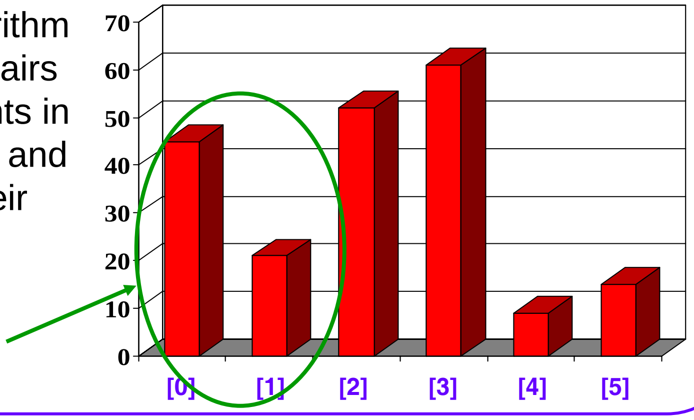
3. The Bubble Sort Algorithm

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



3. The Bubble Sort Algorithm

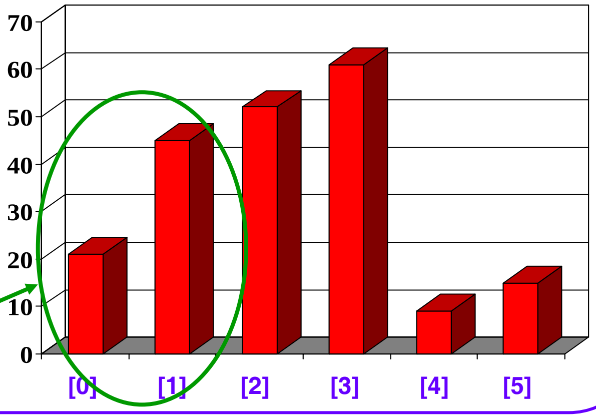
- The Bubble Sort algorithm looks at pairs of elements in the array, and swaps their order if needed.



Swap?

3. The Bubble Sort Algorithm

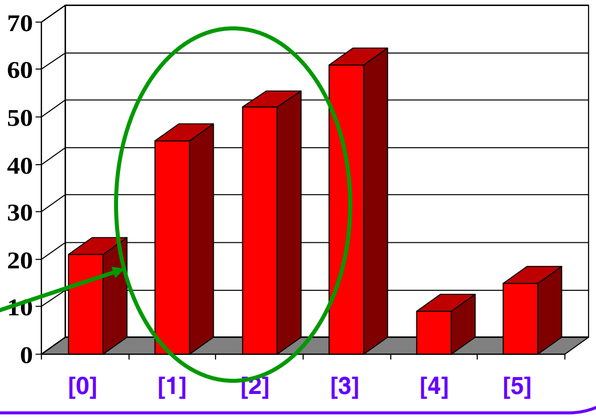
- The Bubble Sort algorithm looks at pairs of elements in the array, and swaps their order if needed.



Yes!

3. The Bubble Sort Algorithm

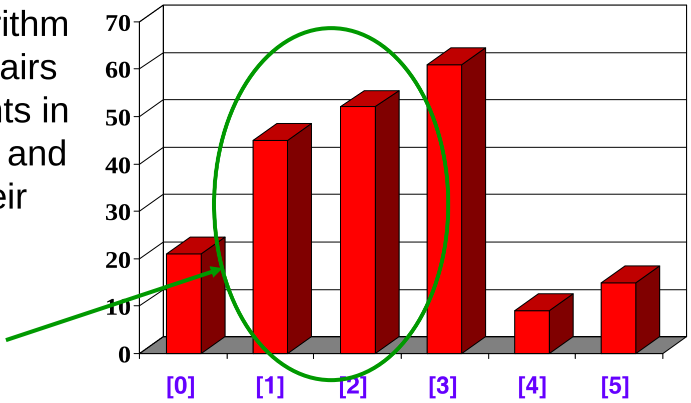
- The Bubble Sort algorithm looks at pairs of elements in the array, and swaps their order if needed.



Swap?

3. The Bubble Sort Algorithm

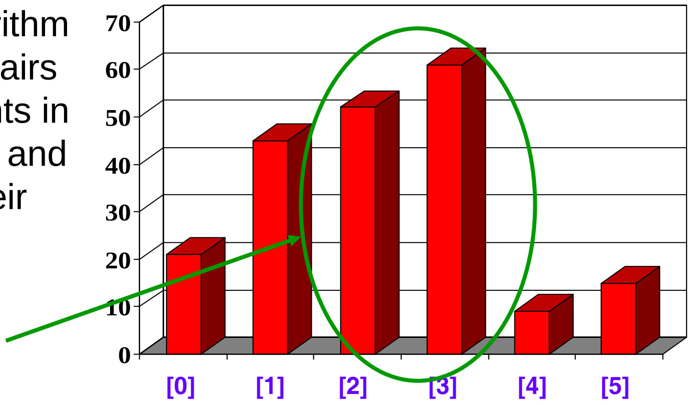
- The Bubble Sort algorithm looks at pairs of elements in the array, and swaps their order if needed.



No.

3. The Bubble Sort Algorithm

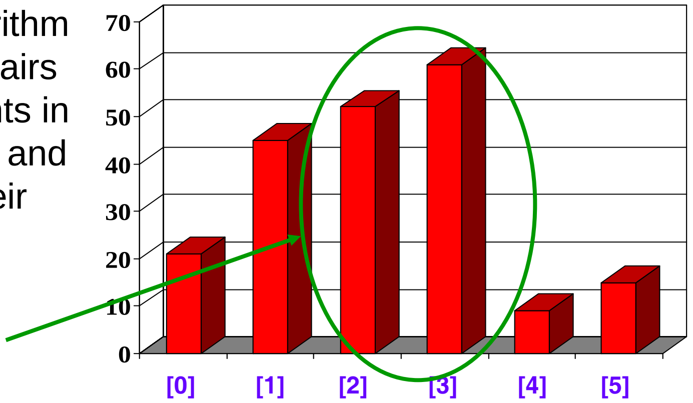
- The Bubble Sort algorithm looks at pairs of elements in the array, and swaps their order if needed.



Swap?

3. The Bubble Sort Algorithm

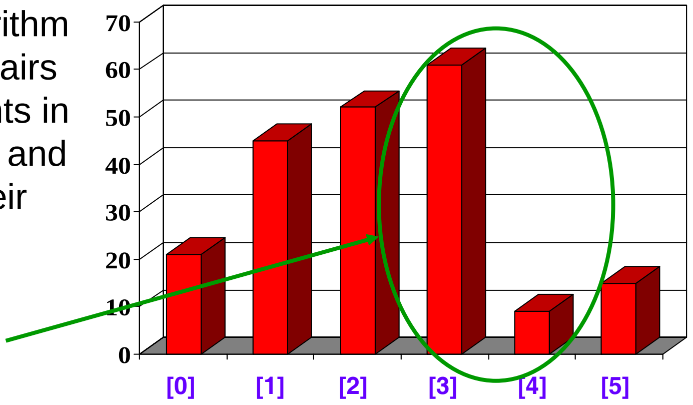
- The Bubble Sort algorithm looks at pairs of elements in the array, and swaps their order if needed.



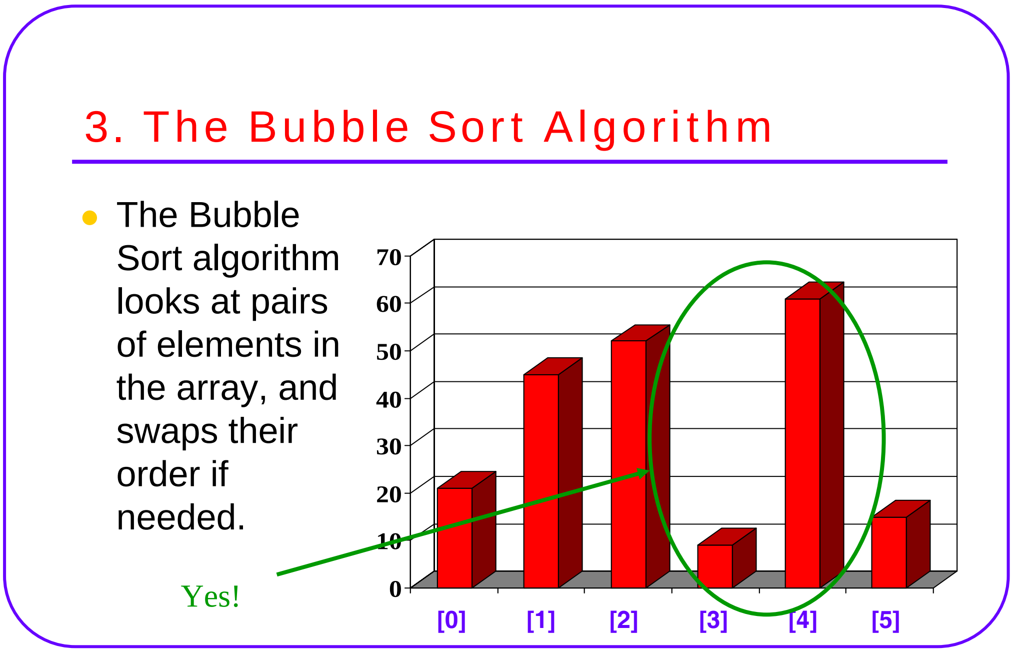
No.

3. The Bubble Sort Algorithm

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



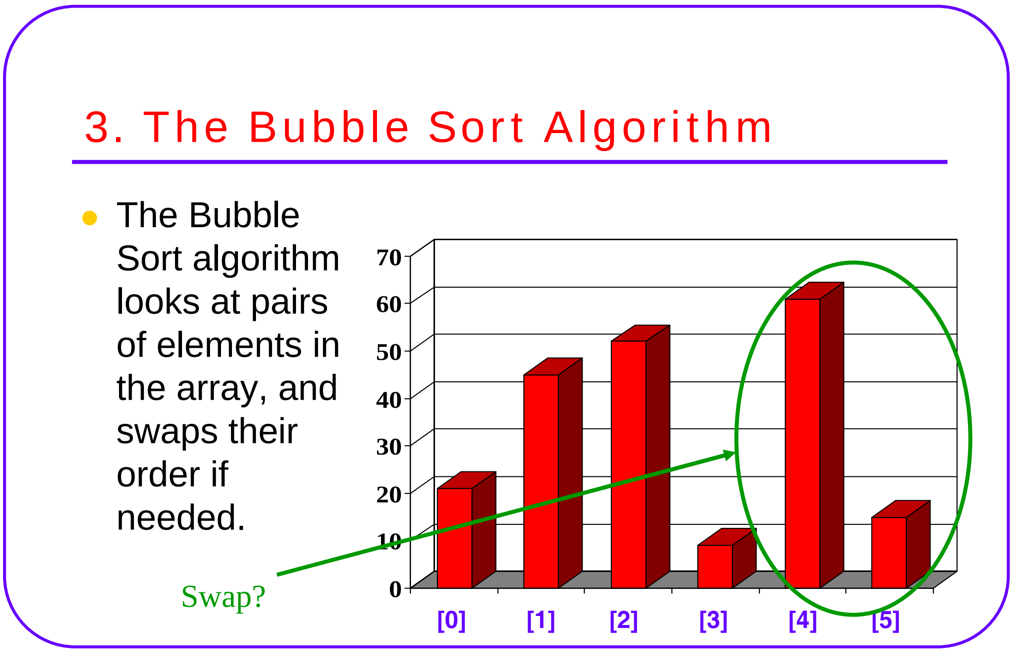
Swap?



3. The Bubble Sort Algorithm

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

Yes!



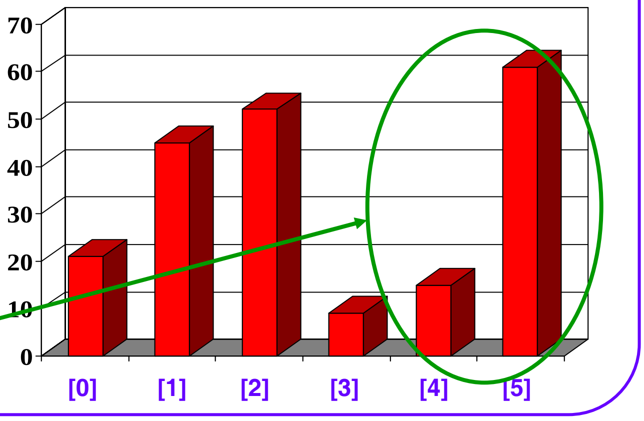
3. The Bubble Sort Algorithm

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

Swap?

3. The Bubble Sort Algorithm

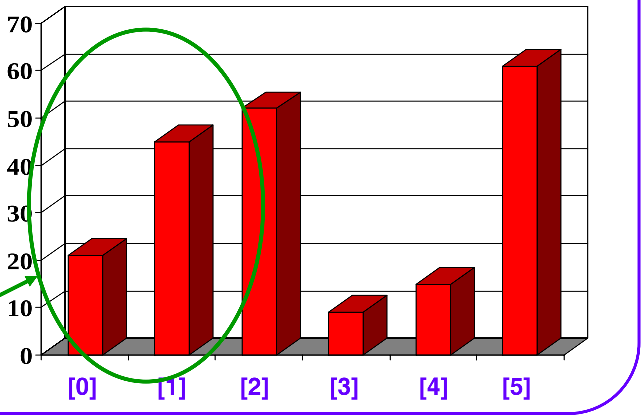
- The Bubble Sort algorithm looks at pairs of elements in the array, and swaps their order if needed.



Yes!

3. The Bubble Sort Algorithm

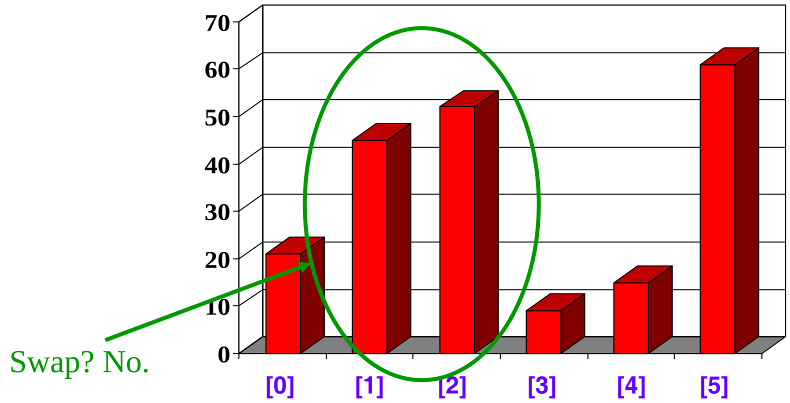
- Repeat.



Swap? No.

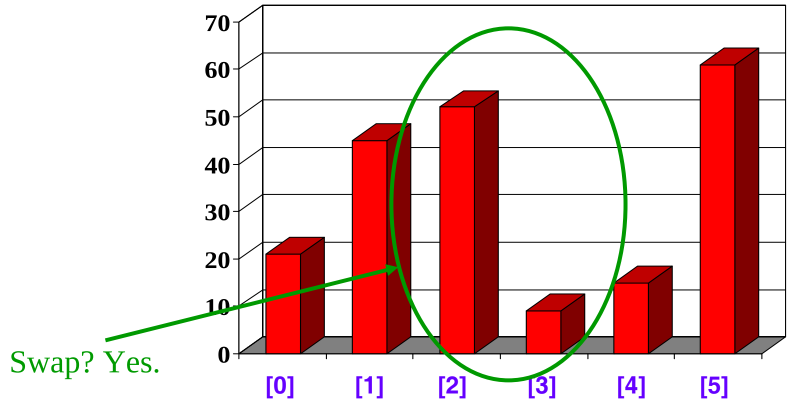
3. The Bubble Sort Algorithm

- Repeat.



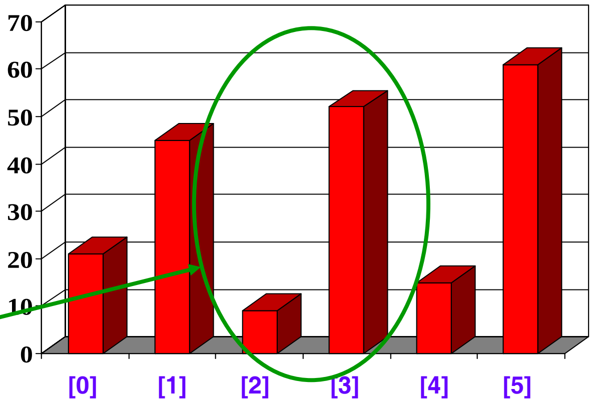
3. The Bubble Sort Algorithm

- Repeat.



3. The Bubble Sort Algorithm

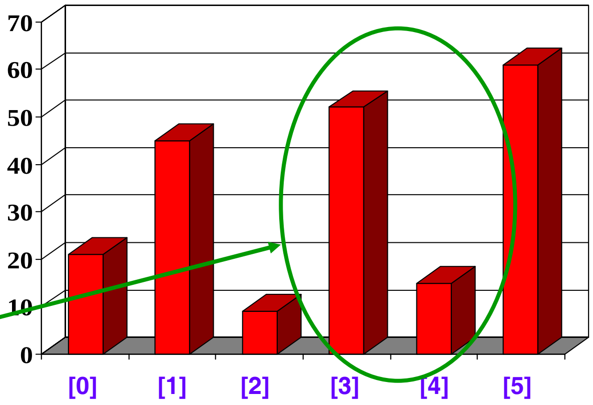
- Repeat.



Swap? Yes.

3. The Bubble Sort Algorithm

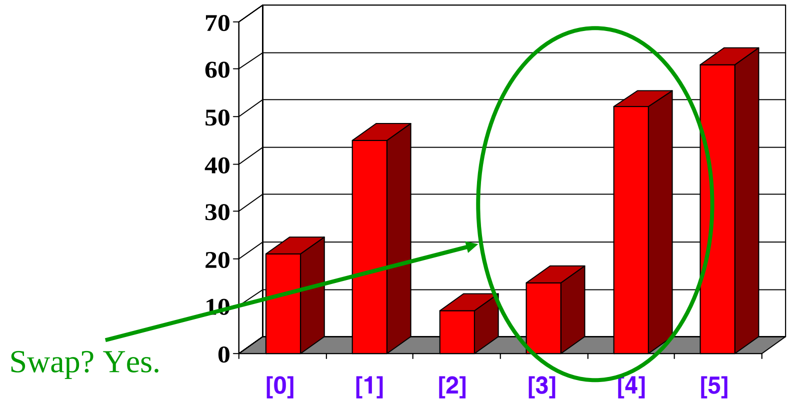
- Repeat.



Swap? Yes.

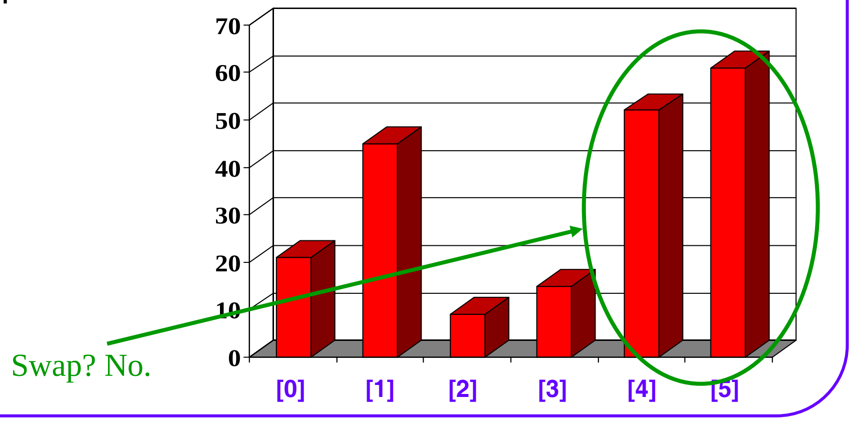
3. The Bubble Sort Algorithm

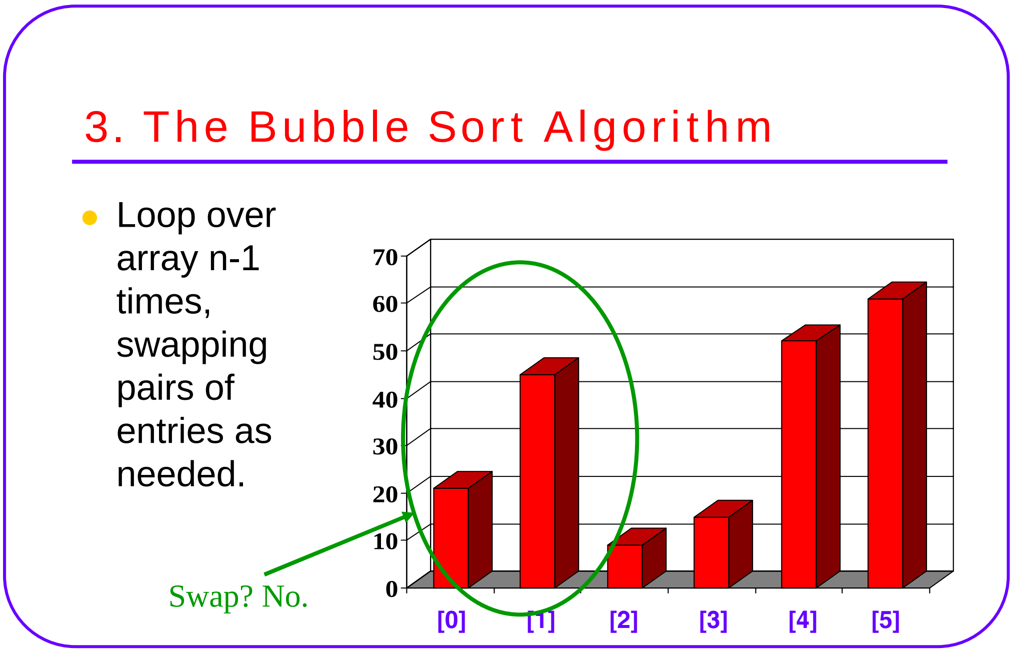
- Repeat.



3. The Bubble Sort Algorithm

- Repeat.

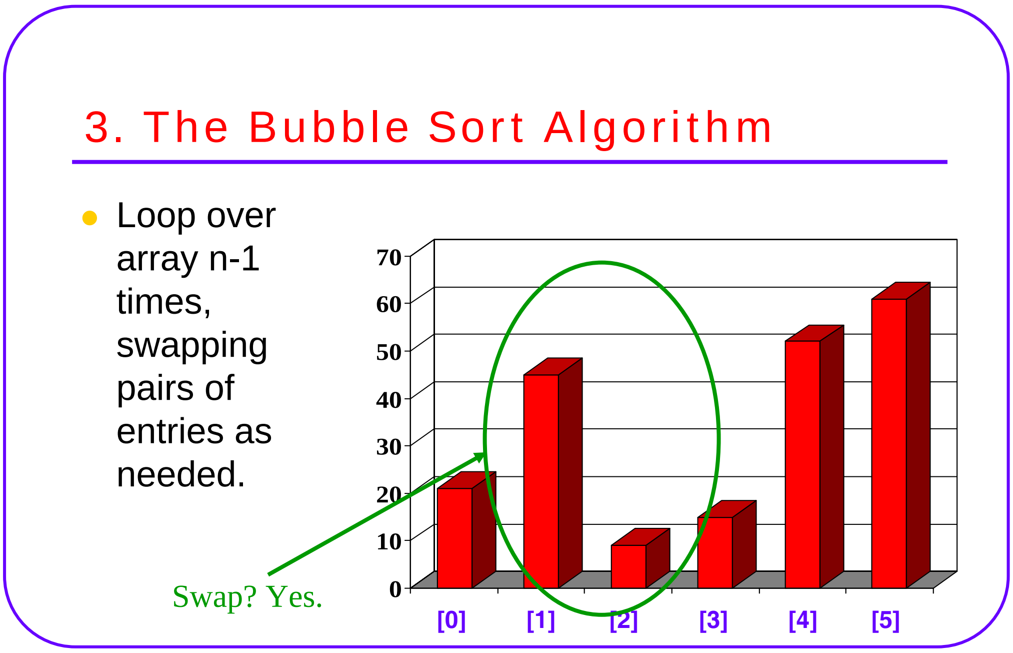




3. The Bubble Sort Algorithm

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

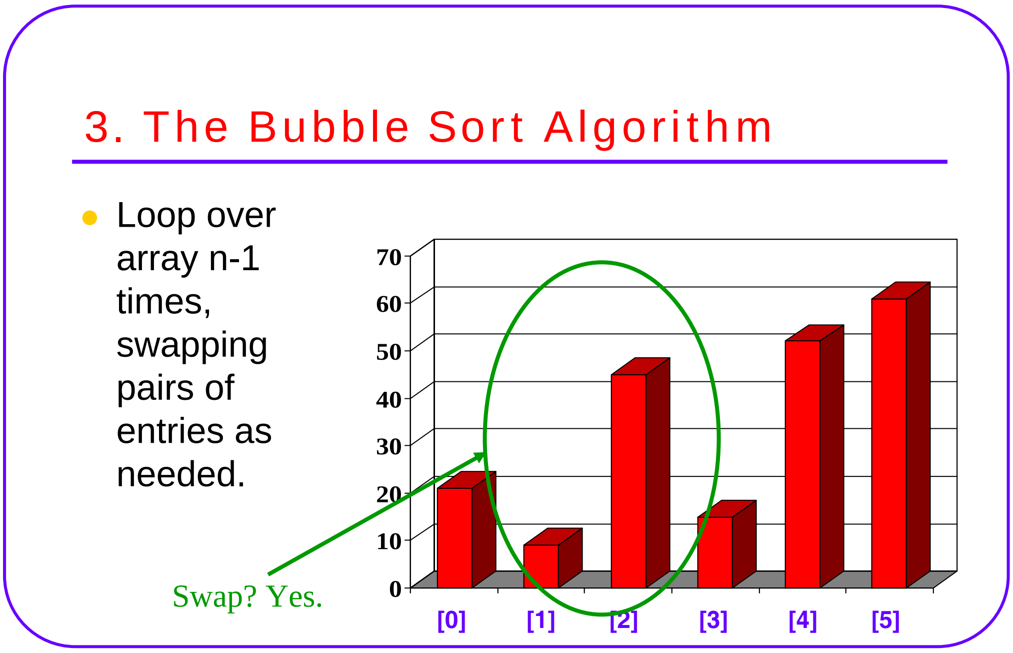
Swap? No.



3. The Bubble Sort Algorithm

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

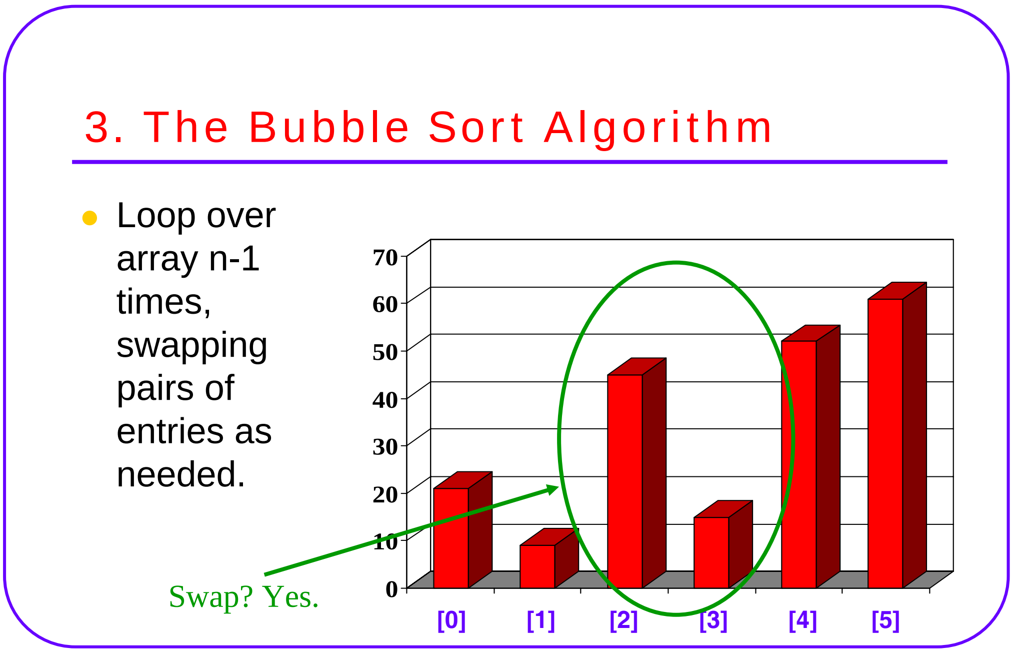
Swap? Yes.



3. The Bubble Sort Algorithm

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

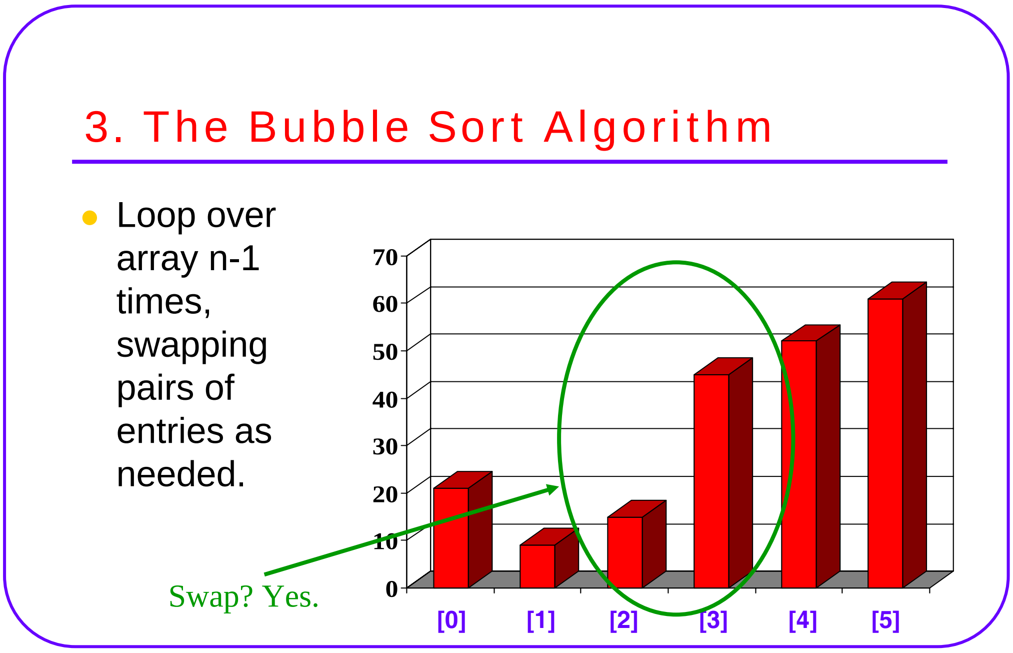
Swap? Yes.



3. The Bubble Sort Algorithm

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

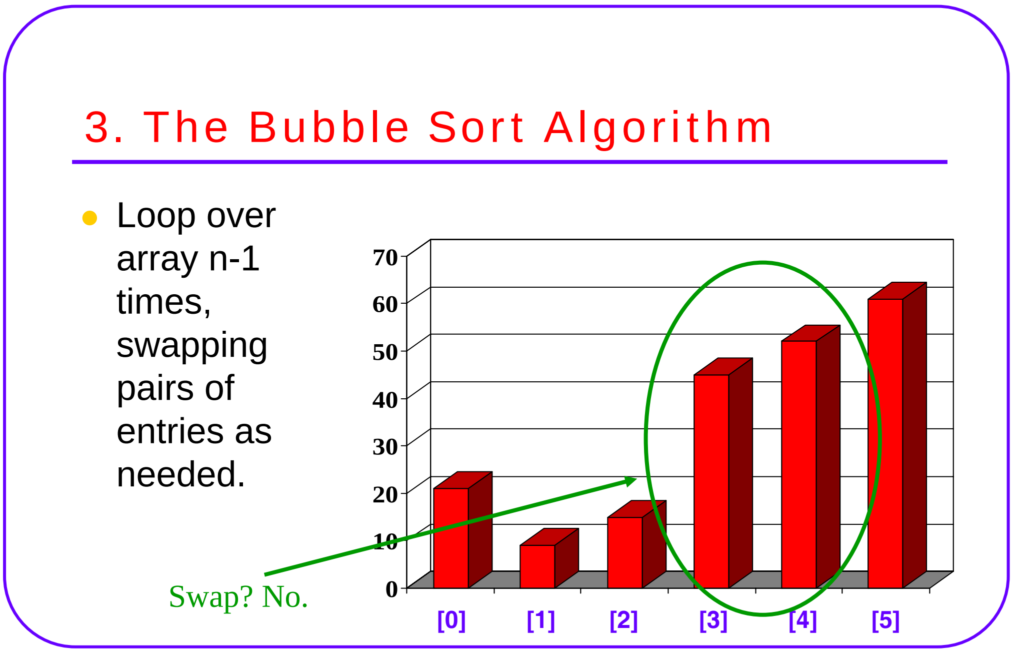
Swap? Yes.



3. The Bubble Sort Algorithm

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

Swap? Yes.



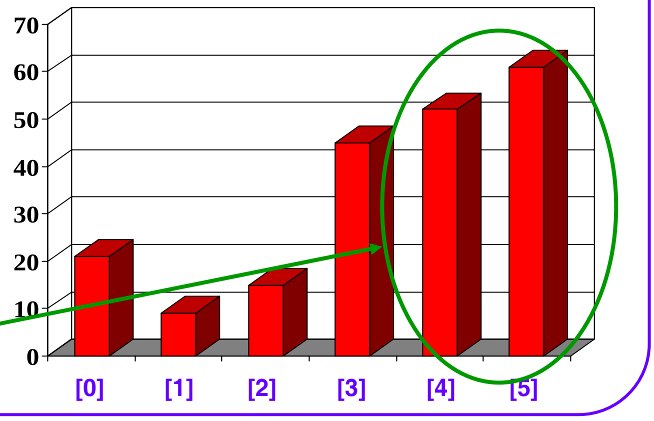
3. The Bubble Sort Algorithm

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

Swap? No.

3. The Bubble Sort Algorithm

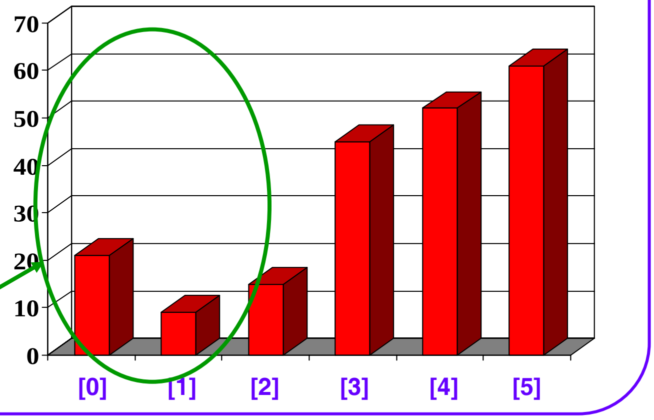
- Loop over array n-1 times, swapping pairs of entries as needed.



Swap? No.

3. The Bubble Sort Algorithm

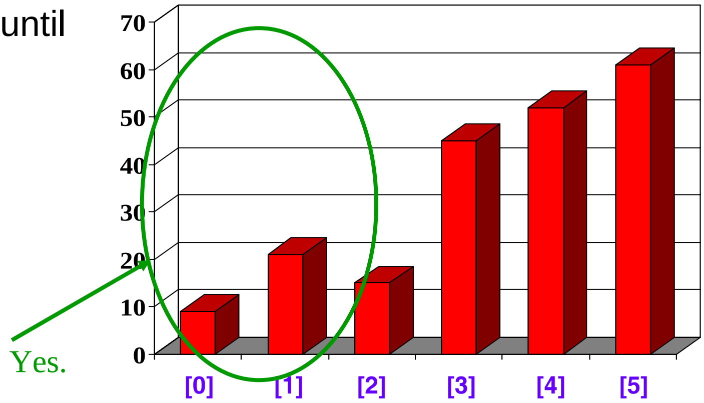
- Continue looping, until done.



Swap? Yes.

3. The Bubble Sort Algorithm

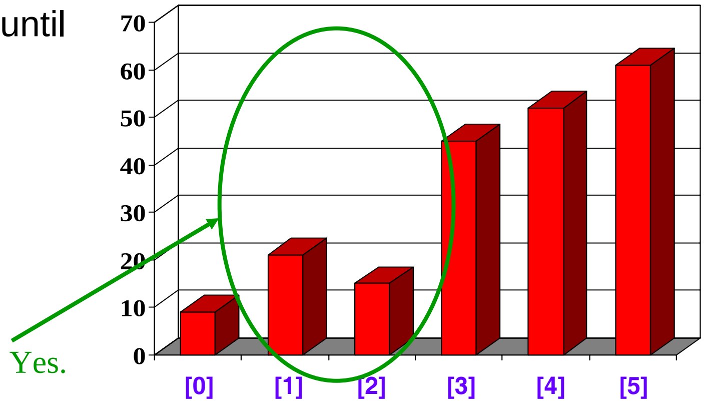
- Continue looping, until done.



Swap? Yes.

3. The Bubble Sort Algorithm

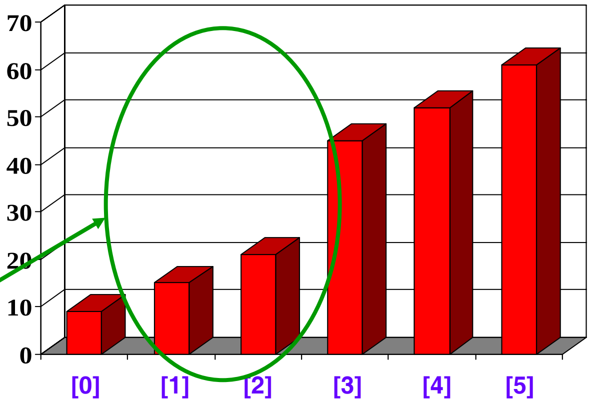
- Continue looping, until done.



Swap? Yes.

3. The Bubble Sort Algorithm

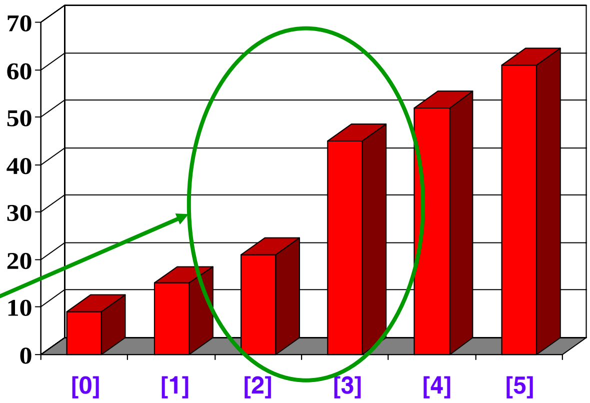
- Continue looping, until done.



Swap? Yes.

3. The Bubble Sort Algorithm

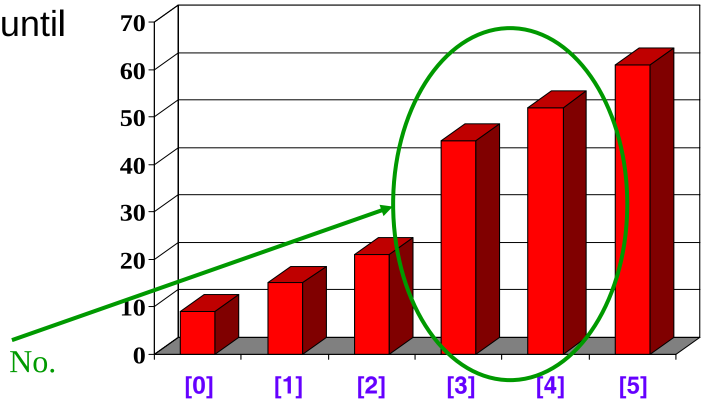
- Continue looping, until done.



Swap? No.

3. The Bubble Sort Algorithm

- Continue looping, until done.

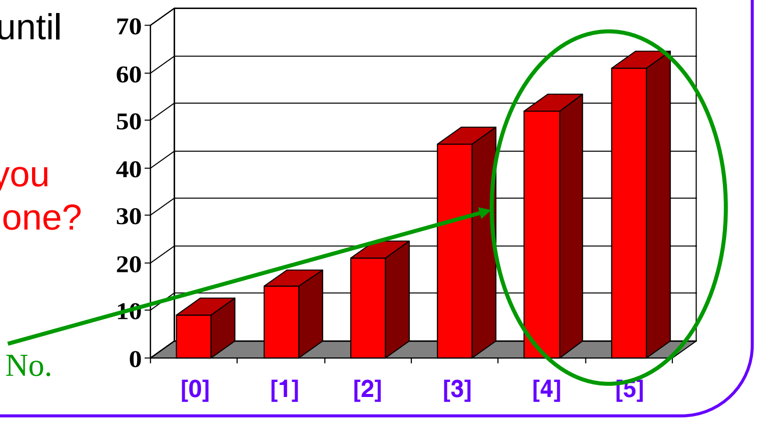


Swap? No.

3. The Bubble Sort Algorithm

- Continue looping, until done.

How do you know if done?



Swap? No.

3. The Bubble Sort Function

```
public void bubbleSort(E[ ] data, EComparator c)  {
    int n = data.length;
    int i, j;

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

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

An Improved Bubble Sort Function

```
public void bubbleSort(E[ ] data, EComparator c)  {
    int n = data.length;
    int i, j;
    boolean sorted = false;

    for (i = n-1; (i > 0)  && !sorted; --i)  {
        // if no elements swapped in a whole iteration, then elements are in order.
        for (sorted = true, j = 0; j < i; ++j)
            if (c.compare(data[j], data[j+1]) > 0)  {             // if out of order, swap!
                swap(data[j], data[j+1]);
                sorted = false;
            }
    }
}
```

Bubble Sort Time Analysis

- In O-notation, what is:
  - Worst case running time for sorting a list of *n* elements?
  - Average case running time for sorting a list of *n* elements?
- Steps of the algorithm:

```
for i = 0 to n-1
    for j =0 to n-2
         if element[j] > element[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 sorting a list of *n* elements?
  - Average case running time for sorting a list of *n* elements?
- Steps of the algorithm:

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

- Bubble sort time analysis: $O(n^2)$

Bubble Sort Time Analysis

Initial Data Order	Comparisons	Assignments
Sorted Order	n-1 = O(n)	0 = O(1)
Random Order	n(n-1)/4 = O((n^2)/4)	n(n-1)/4 = O((n^2)/4)
Inverse Order	n(n-1)/2 = O((n^2)/2)	n(n-1)/2 = O((n^2)/2)

Summary

- 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 for good performance in sorting large arrays.