

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

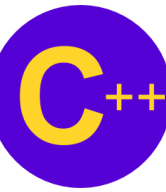


Introduction

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.

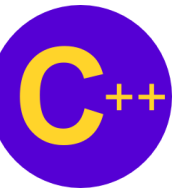


Quadratic Sorting Algorithms

- 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

Iterator



How to iterate through a vector in C++

1. Use a for loop and reference pointer
2. Use an iterator
3. Use the auto keyword

Use a for loop and reference pointer

- In C++, vectors can be indexed with **[]operator**, similar to arrays.
- To iterate through the vector, run a **for** loop from **i = 0** to **i = vec.size()**.
- Instead of **[]operator**, the **at()** function can also be used to fetch the value of a vector at a specific index:

```
vec.at(i)
```

Where **i** is the index.

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    //declaration of int vector
    vector<int> vec = {1, 2, 3, 4, 5};

    // returns length of vector as unsigned int
    unsigned int vecSize = vec.size();

    // run for loop from 0 to vecSize
    for(unsigned int i = 0; i < vecSize; i++)
    {
        cout << vec[i] << " ";
    }
    cout << endl;

    return 0;
}
```

forloop.cpp

Use an iterator

- An **iterator** can be generated to traverse through a vector.

```
vector<int>::iterator iter;
```

- An **iterator** is used as a **pointer** to iterate through a sequence such as a **string** or **vector**.
- The pointer can then be incremented to access the next element in the sequence.
- To access the value in the memory space to which the iterator is pointing, you must use `*`.

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
    vector<int> vec = {1, 2, 3, 4, 5};
    int vecSize = vec.size(); // returns length of vector

    vector<int>::iterator iter = vec.begin();

    cout << "Vector: ";

    for(iter; iter < vec.end(); iter++)
    {
        // access value in the memory to which the pointer
        // is referencing
        cout << *iter << " ";
    }
    cout << endl;
    return 0;
}
```

Iterator.cpp



Use the auto keyword

- The **auto** keyword can also be used to traverse through a vector.
- **auto** must be given a variable to store the accessed element and the sequence that needs to be iterated.
- It must be used in conjunction with a for loop, as shown below.

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    //declearation of int vector
    vector<int> vec = {1, 2, 3, 4, 5};
    int vecSize = vec.size(); // returns length of vector

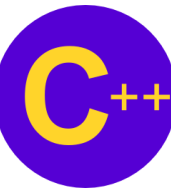
    cout << "Vector: ";

    for (auto & element : vec)
    {
        cout << element << " ";
    }
    return 0;
}
```

auto.cpp

LECTURE 3

C++ Standard Algorithms



C++ Standard Library algorithms

- The ***C++ Standard Library algorithms*** are components that perform algorithmic operations on containers and other sequences.
- Because searching, counting, and sorting are such common operations, the C++ Standard Library comes with a set of algorithms to easily implement these common functions in just a few lines of code. The Standard Library functions are well tested, optimized and efficient. They work on a variety of different container types, and many support parallelization (large problems can be divided into smaller ones and solved simultaneously, in order to complete tasks faster).



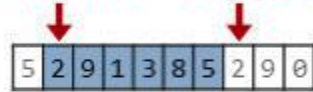
C++ Standard Library algorithms

- The algorithms library functions operate on ranges of elements. Note that a range is defined as `[first, last)` where `last` refers to the element past the last element to inspect.
- The C++ Standard algorithms are located in the `<algorithm>` header.

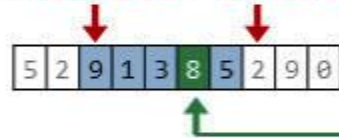
C++ Standard Library Algorithms

Non-Modifying Sequence Operations <algorithm>

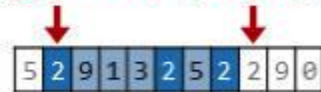
`any_of (@first, @last, f()→bool) →` • true, if *f* yields true for *any*, *all* or *none* elements in the input range
`all_of (@first, @last, f()→bool) →`
`none_of (@first, @last, f()→bool) →` • false otherwise



`find (@first, @last, value) →` • @1st match
`find_if (@first, @last, f()→bool) →` • @last if no match



`count (@first, @last, value) →` number of occurrences
`count_if (@first, @last, f()→bool) →` (here: 3 if value = 2)

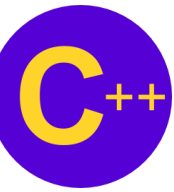


`search (@fst1, @lst1, @fst2, @lst2) →` • @1st occurrence of sequence 2 inside sequence 1
• @last1 otherwise



LECTURE 4

C++ Lambda Expressions



C++ Lambda Expressions

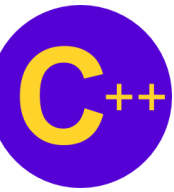
- C++ 11 introduced lambda expressions to allow inline functions which can be used for short snippets of code that are not going to be reused and therefore do not require a name. In their simplest form a lambda expression can be defined as follows:

```
[ capture clause ] (parameters) -> return-type  
{  
    definition of method  
}
```



C++ Lambda Expressions

- Generally, the return-type in lambda expressions is evaluated by the compiler itself and we don't need to specify it explicitly. Also, the -> return-type part can be ignored. However, in some complex cases e.g. conditional statements, the compiler can't determine the return type and explicit specification is required.



C++ Lambda Expressions

- A lambda expression can have more power than an ordinary function by having access to variables from the enclosing scope. We can capture external variables from the enclosing scope in three ways :
 - Capture by reference
 - Capture by value
 - Capture by both (mixed capture)
- Syntax used for capturing variables :
 - [&] : capture all external variables by reference
 - [=] : capture all external variables by value
 - [a, &b] : capture a by value and b by reference
- A lambda with an empty capture clause [] can only access variables which are local to it.



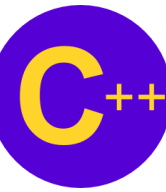
Demo Program

- lambda1.cpp
- lambda2.cpp

LECTURE 5

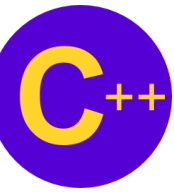
C++ STL Functional

Sr.No.	Operator classes	Definition
1	<u>bit_and</u>	It is a bitwise AND function object class
2	<u>bit_or</u>	It is a bitwise OR function object class
3	<u>bit_xor</u>	It is a bitwise XOR function object class
3	<u>divides</u>	It is a division function object class
4	<u>equal_to</u>	It is a function object class for equality comparison
5	<u>greater</u>	It is a function object class for greater-than inequality comparison
6	<u>greater_equal</u>	It is a function object class for greater-than-or-equal-to comparison
7	<u>less</u>	It is a function object class for less-than inequality comparison
8	<u>less_equal</u>	It is a function object class for less-than-or-equal-to comparison
9	<u>logical_and</u>	It is a logical AND function object class
10	<u>logical_not</u>	It is a logical NOT function object class
11	<u>logical_or</u>	It is a logical OR function object class
12	<u>minus</u>	It is a subtraction function object class
13	<u>modulus</u>	It is a modulus function object class
14	<u>multiplies</u>	It is a multiplication function object class
15	<u>negate</u>	It is a negative function object class
16	<u>not_equal_to</u>	It is a function object class for non-equality comparison
17	<u>plus</u>	It is an addition function object class



Algorithm max_element / min_element

std::max_element is defined inside the header file and it returns an iterator pointing to the element with the largest value in the range [first, last). **std::max_element** can be used in two ways. The comparisons can be performed either using **operator <** (first version), or using a **pre-defined function** (second version). If more than one element satisfies the condition of being the largest, the iterator returned points to the first of such elements.



Algorithm sort

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<int> v{ 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };

    sort(v.begin(), v.end());

    cout << "Sorted \n";
    for (auto x : v)
        cout << x << " ";

    return 0;
}
```

sort.cpp



Algorithm sort2

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<int> v{ 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };

    sort(v.begin(), v.end(), greater<int>());

    cout << "Sorted \n";
    for (auto x : v)
        cout << x << " ";

    return 0;
}
```

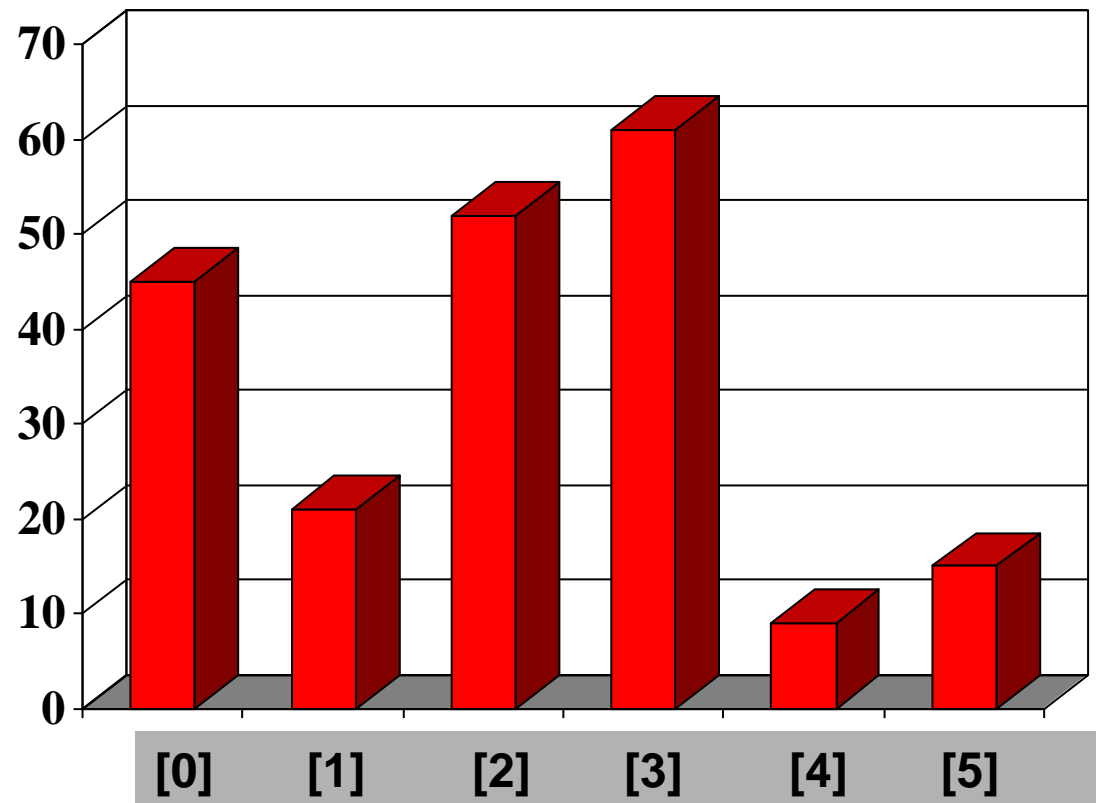
sort2.cpp

LECTURE 4

Selection Sort

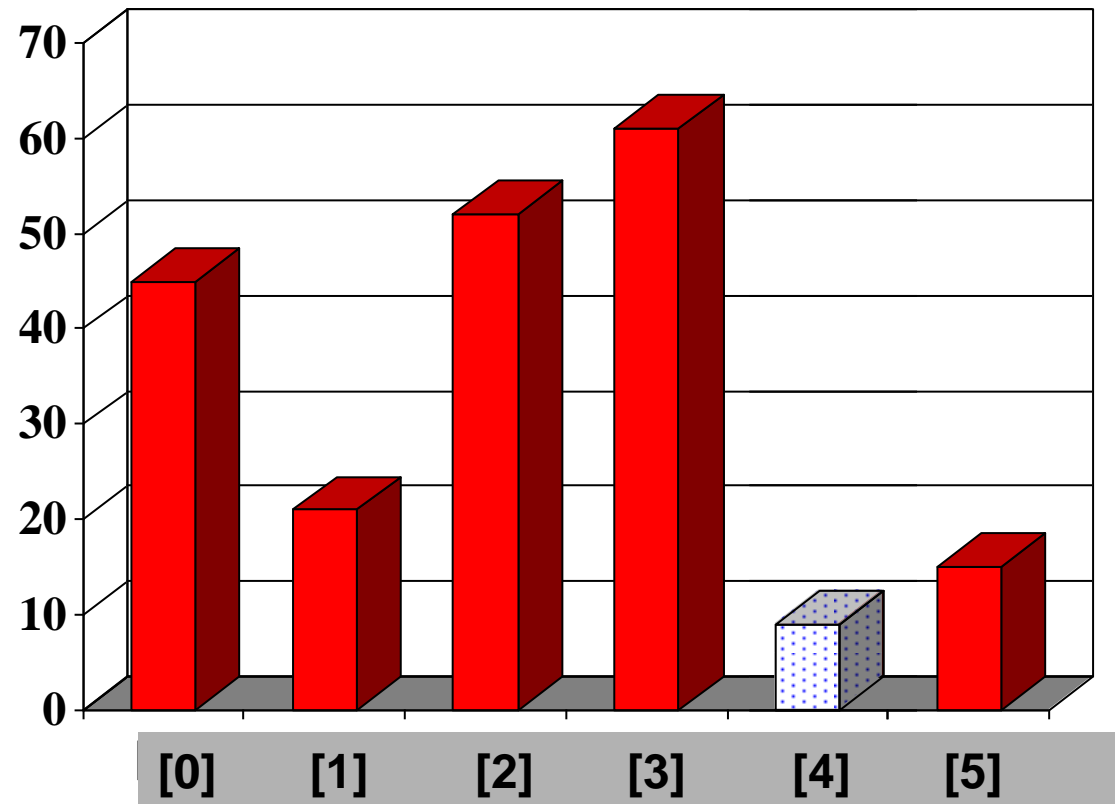
Sorting an Array of Integers

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



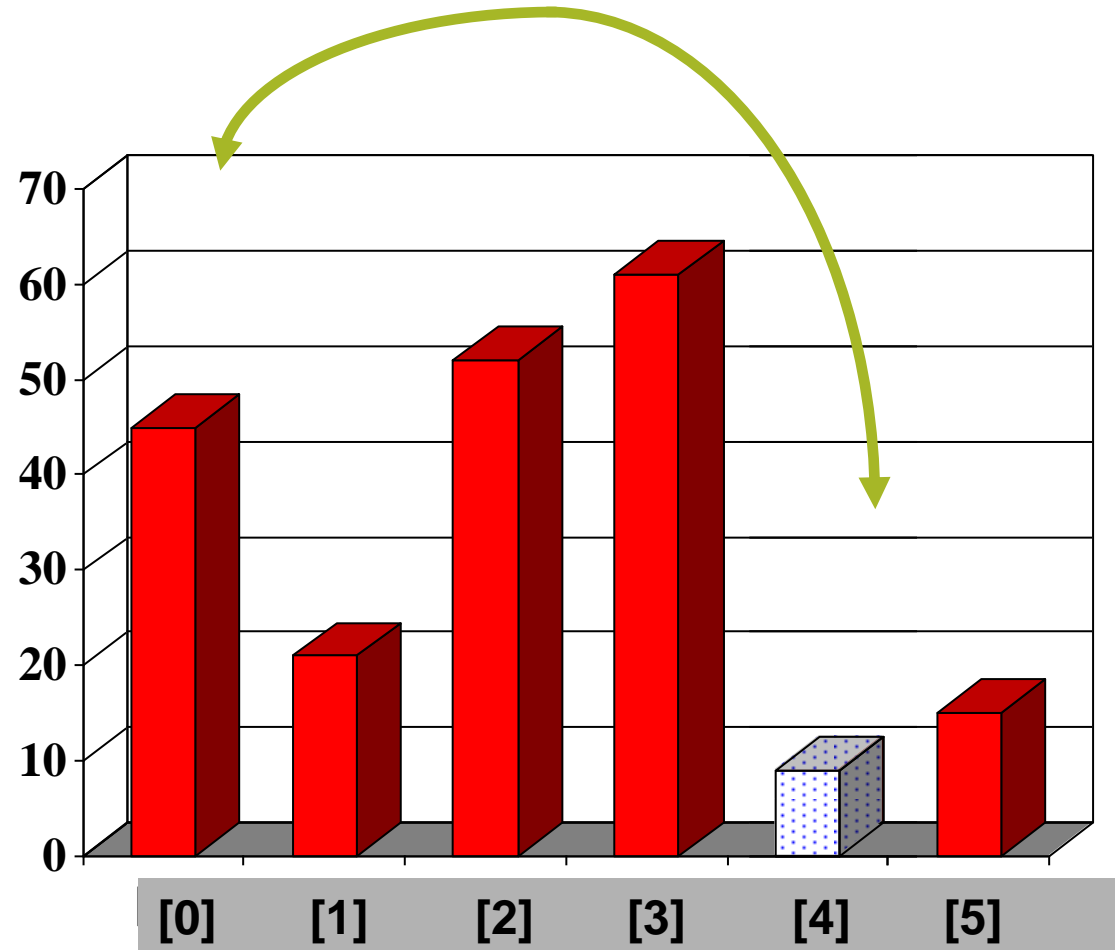
The Selection Sort Algorithm

Start by
finding the
smallest entry.



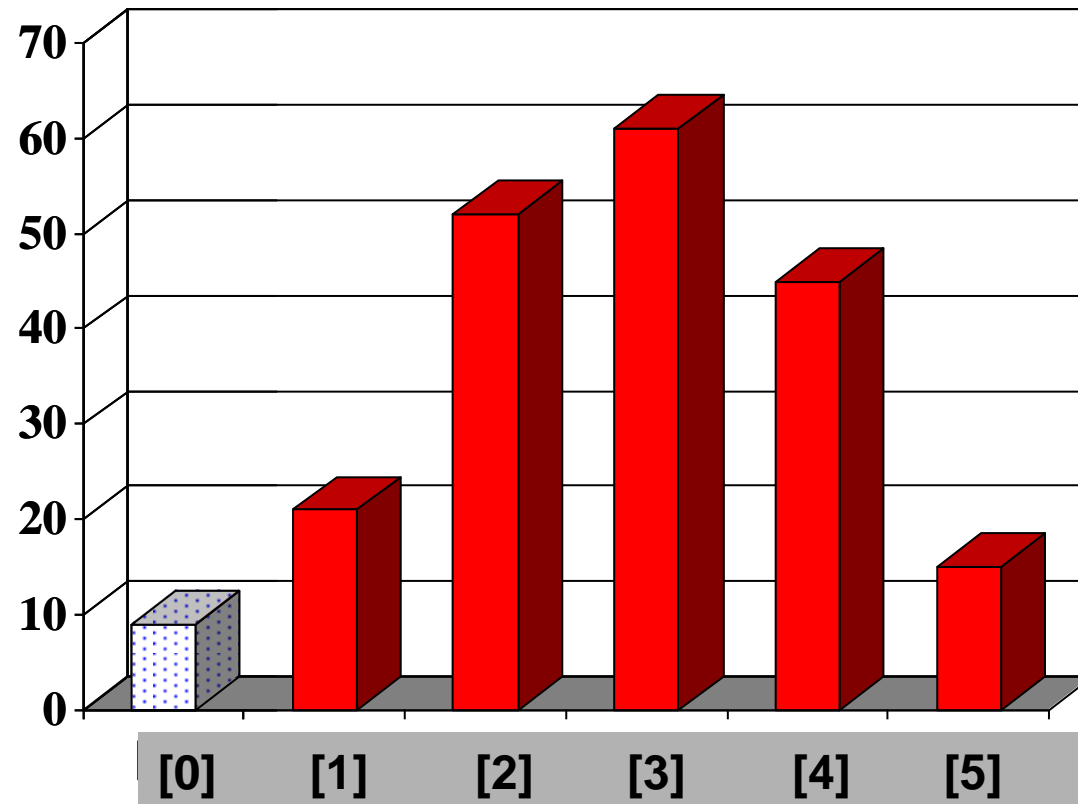
The Selection Sort Algorithm

Swap the smallest entry with the first entry.



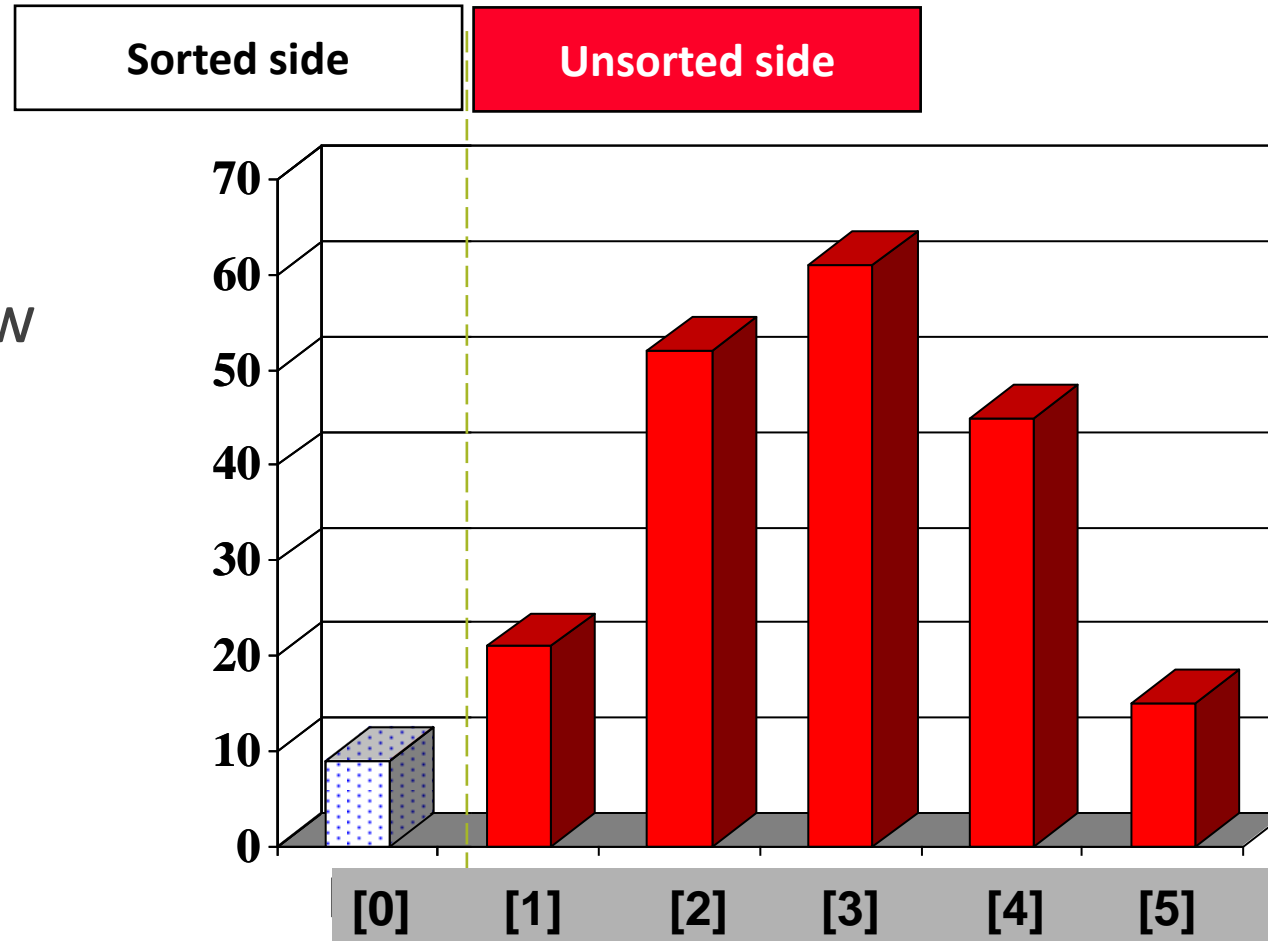
The Selection Sort Algorithm

Swap the smallest entry with the first entry.

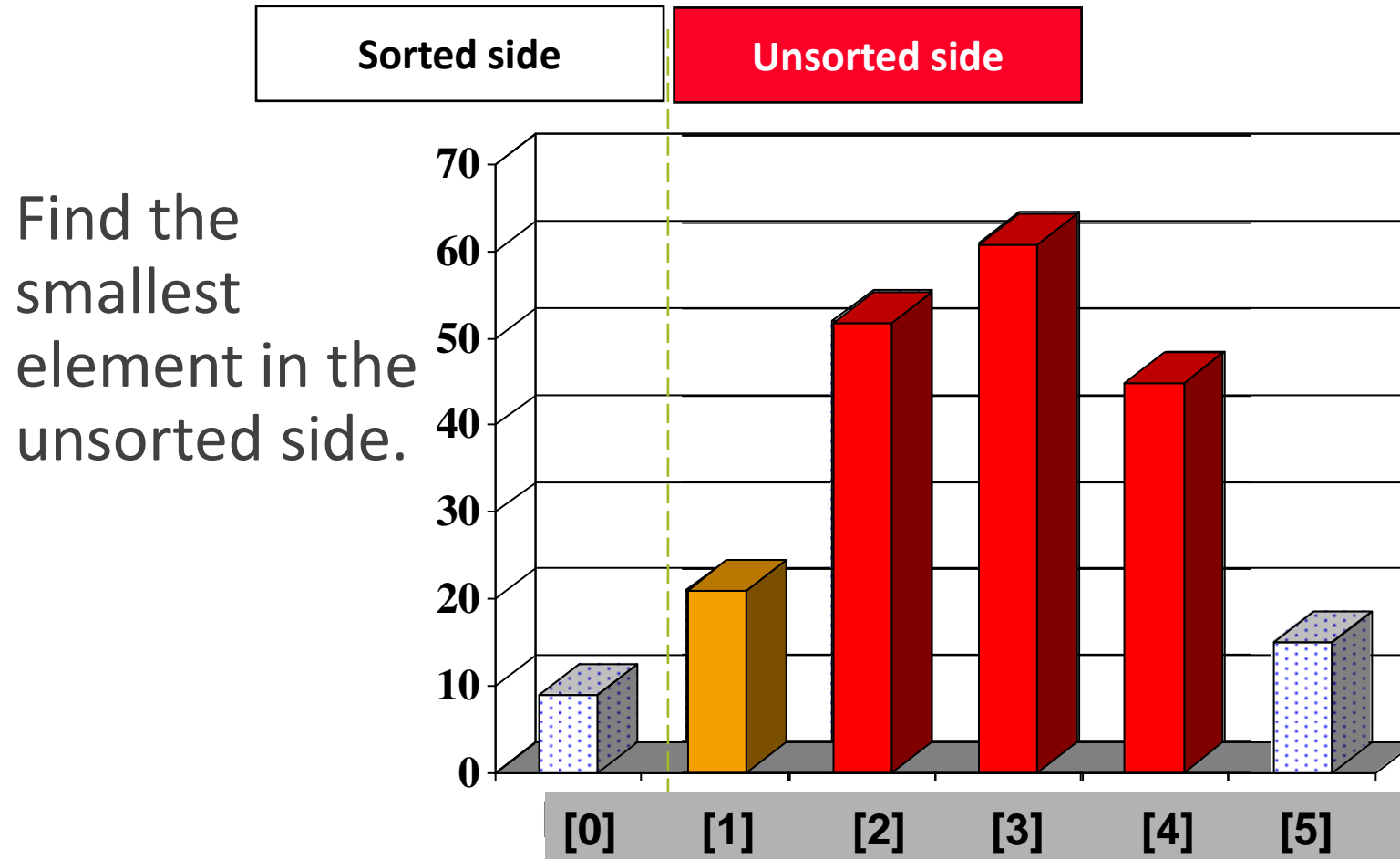


The Selection Sort Algorithm

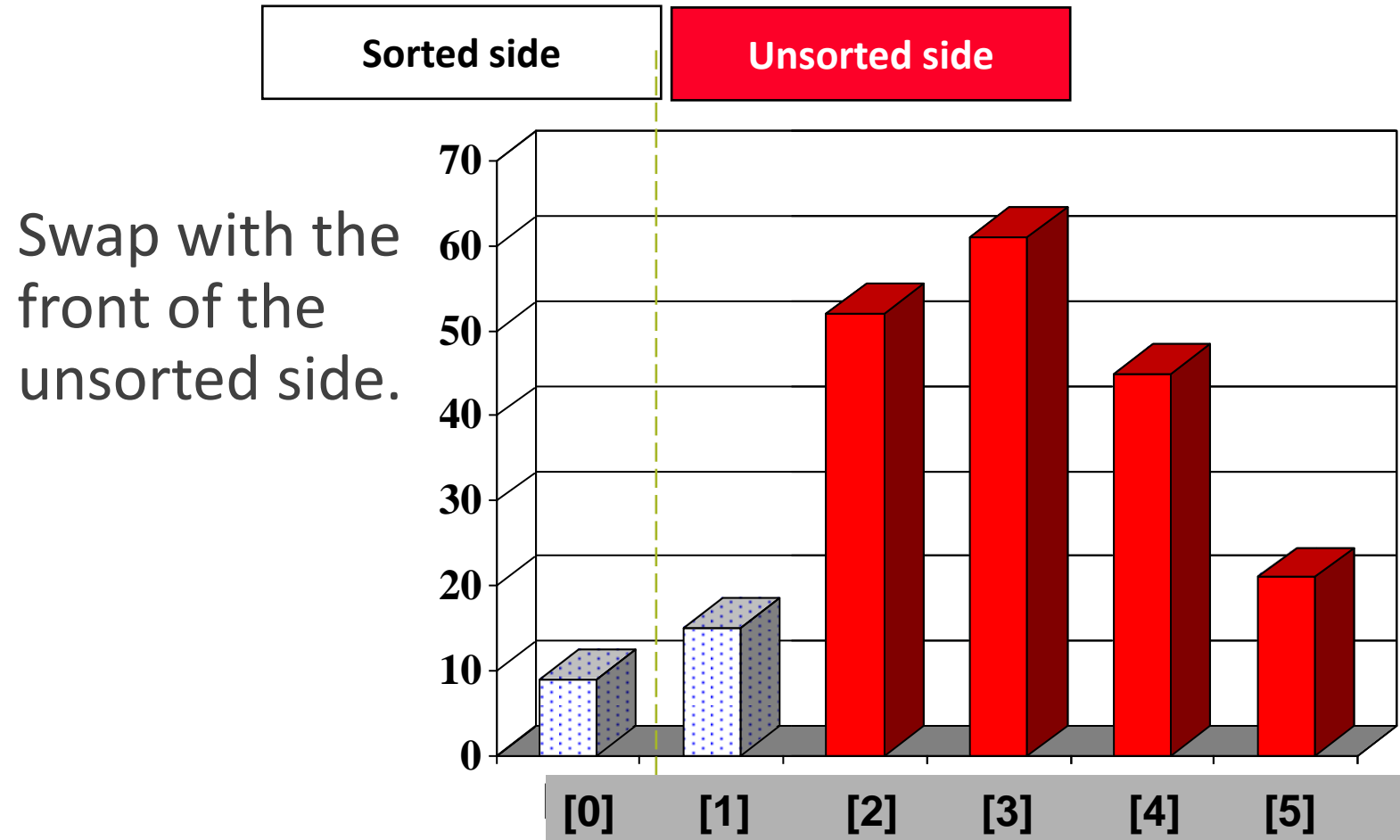
Part of the array is now sorted.



The Selection Sort Algorithm

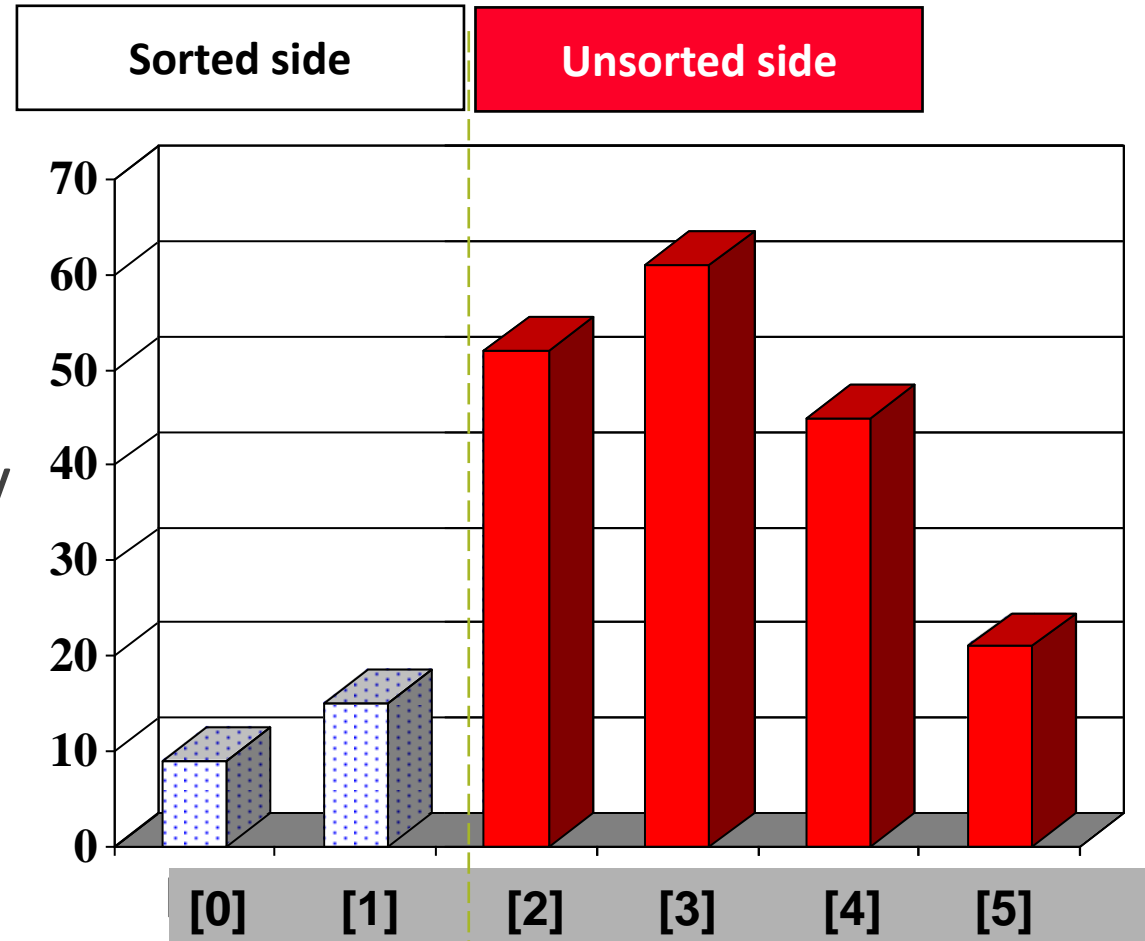


The Selection Sort Algorithm



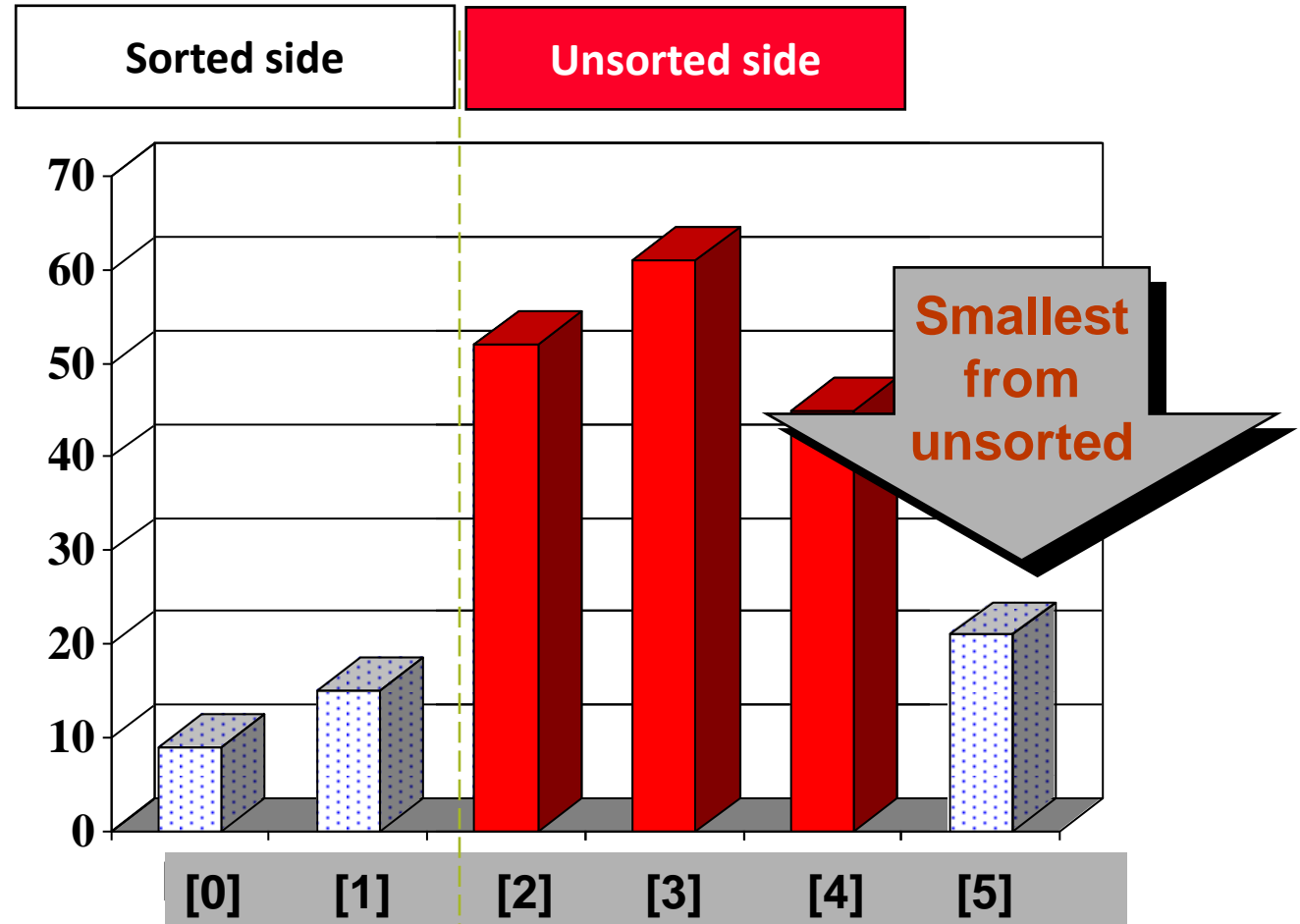
The Selection Sort Algorithm

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



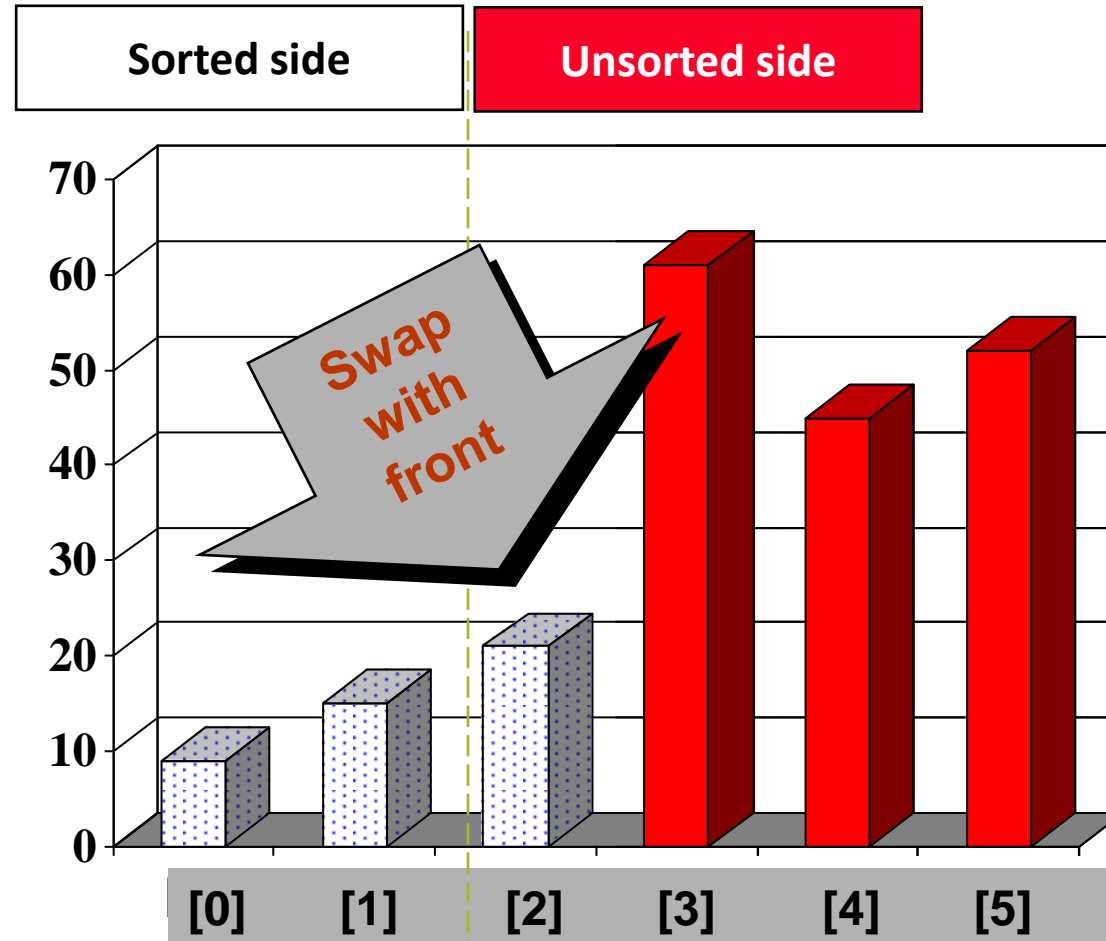
The Selection Sort Algorithm

The process continues...

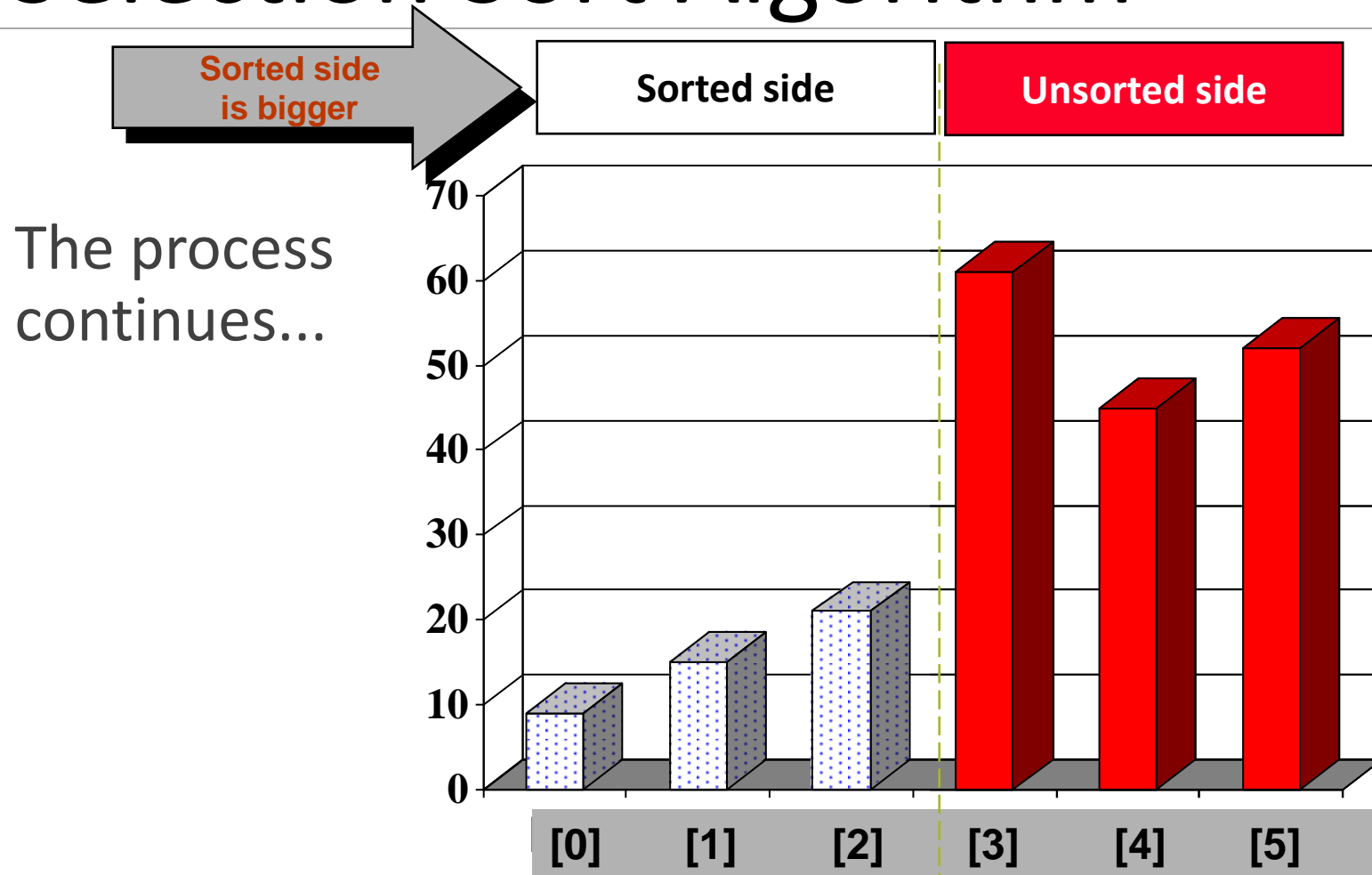


The Selection Sort Algorithm

The process continues...



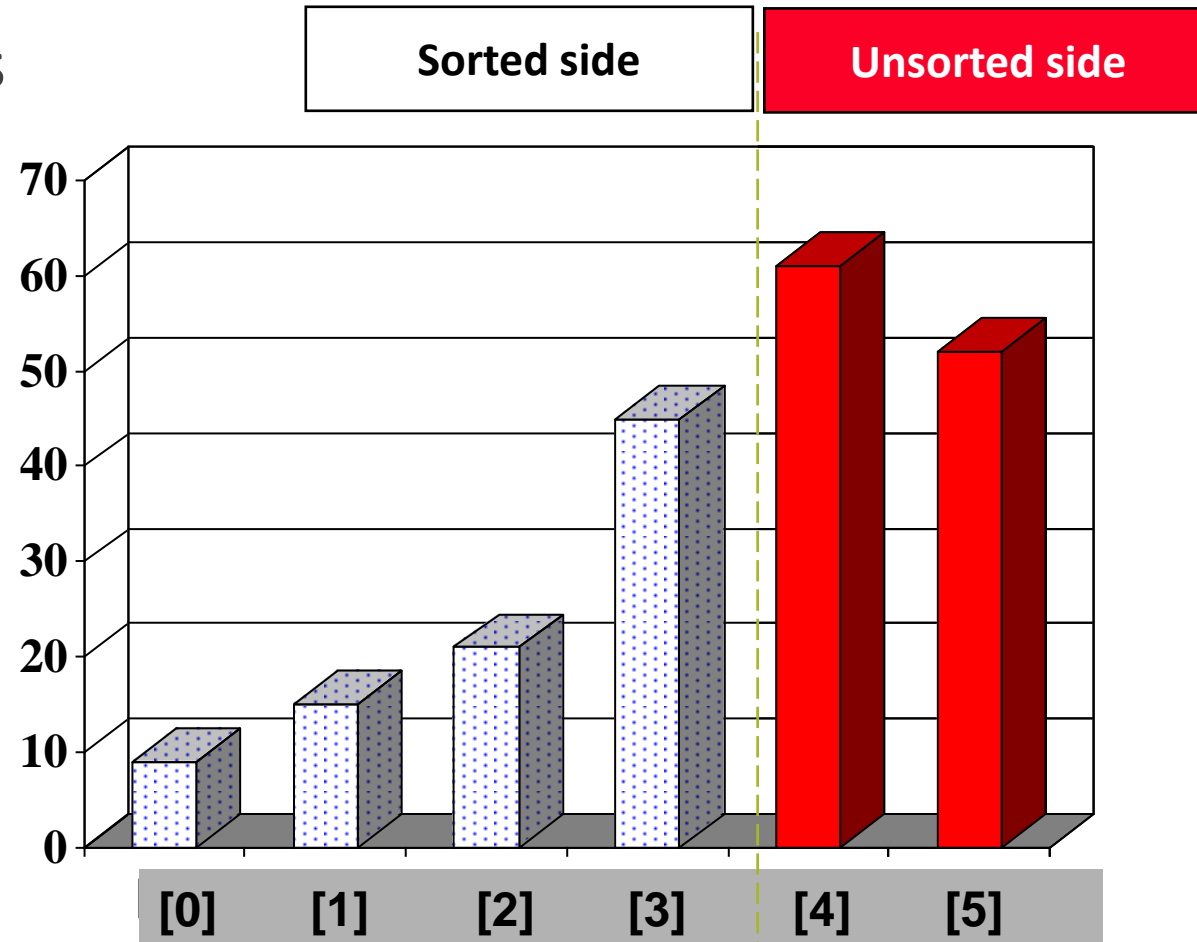
The Selection Sort Algorithm



The Selection Sort Algorithm

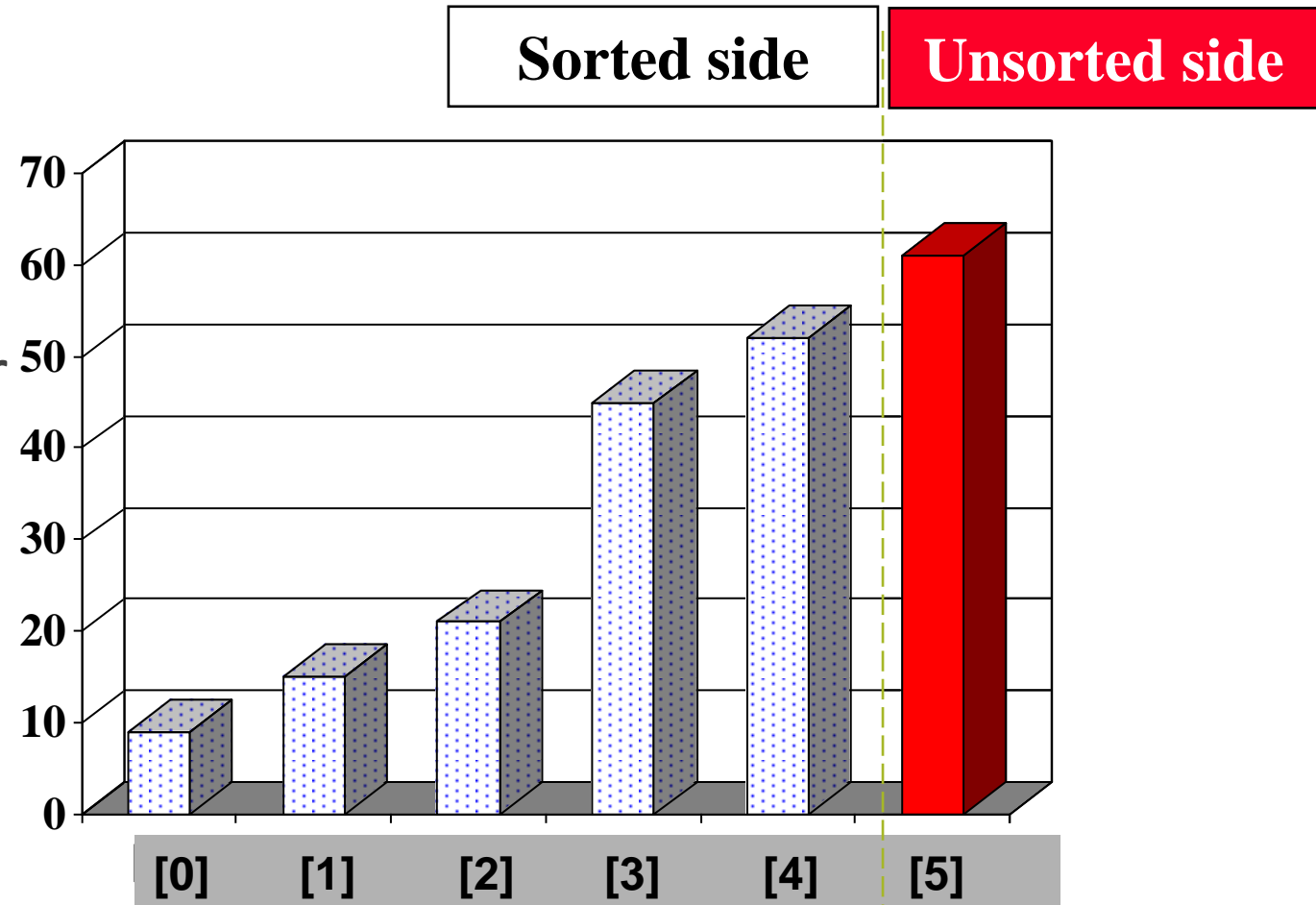
The process keeps adding one more number to the sorted side.

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



The Selection Sort Algorithm

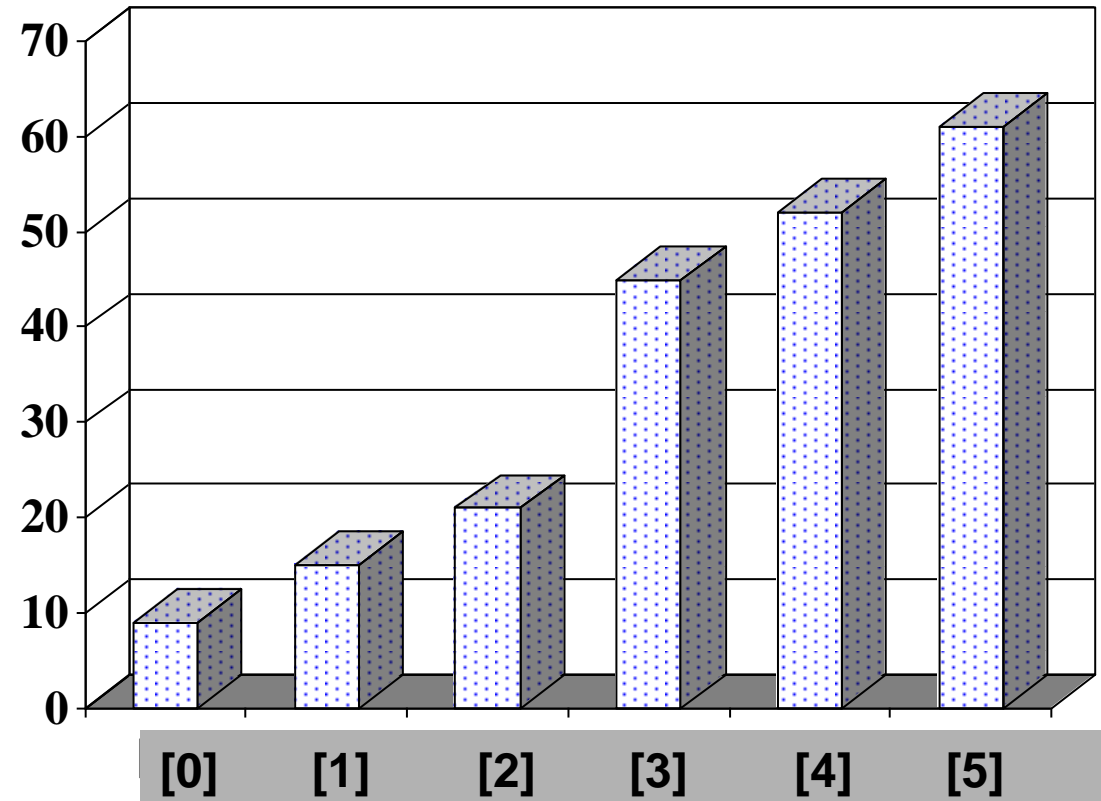
We can stop when the unsorted side has just one number, since that number must be the largest number.



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.

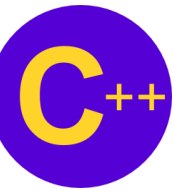


```
template <class Item>
void selection_sort(Item data[ ], size_t n){
    size_t i, j, smallest;
    Item temp;

    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: $O(n)$

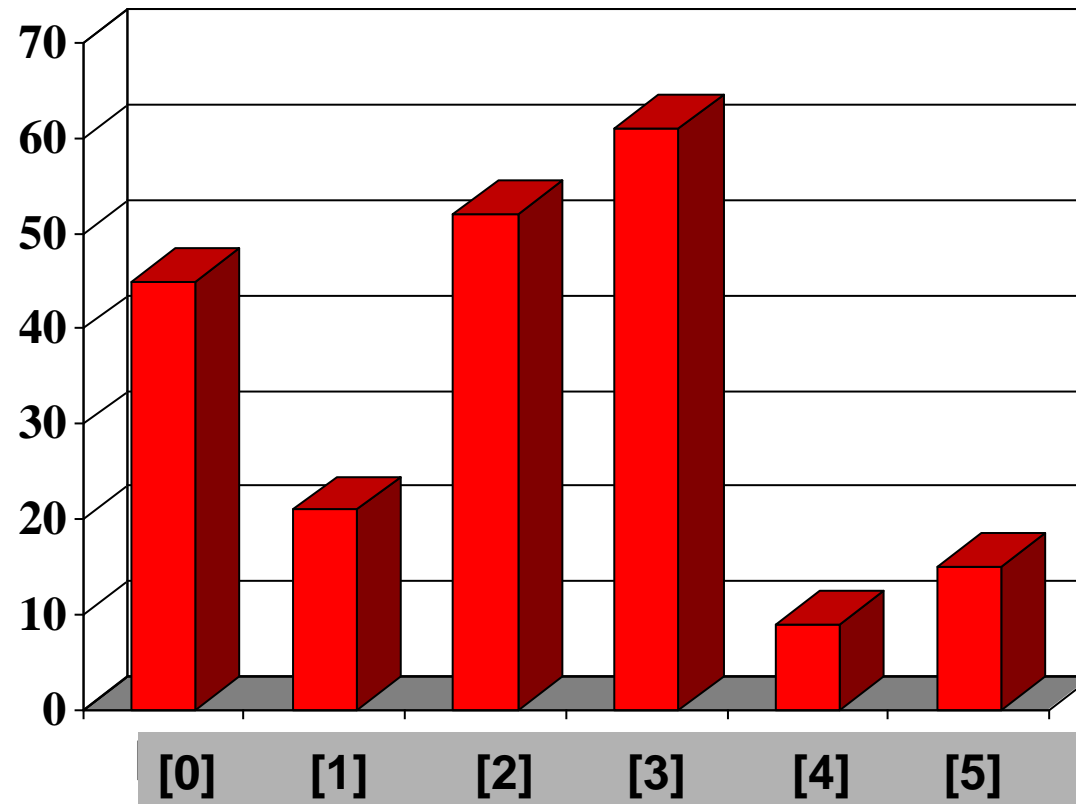
Inner loop: $O(n)$

LECTURE 5

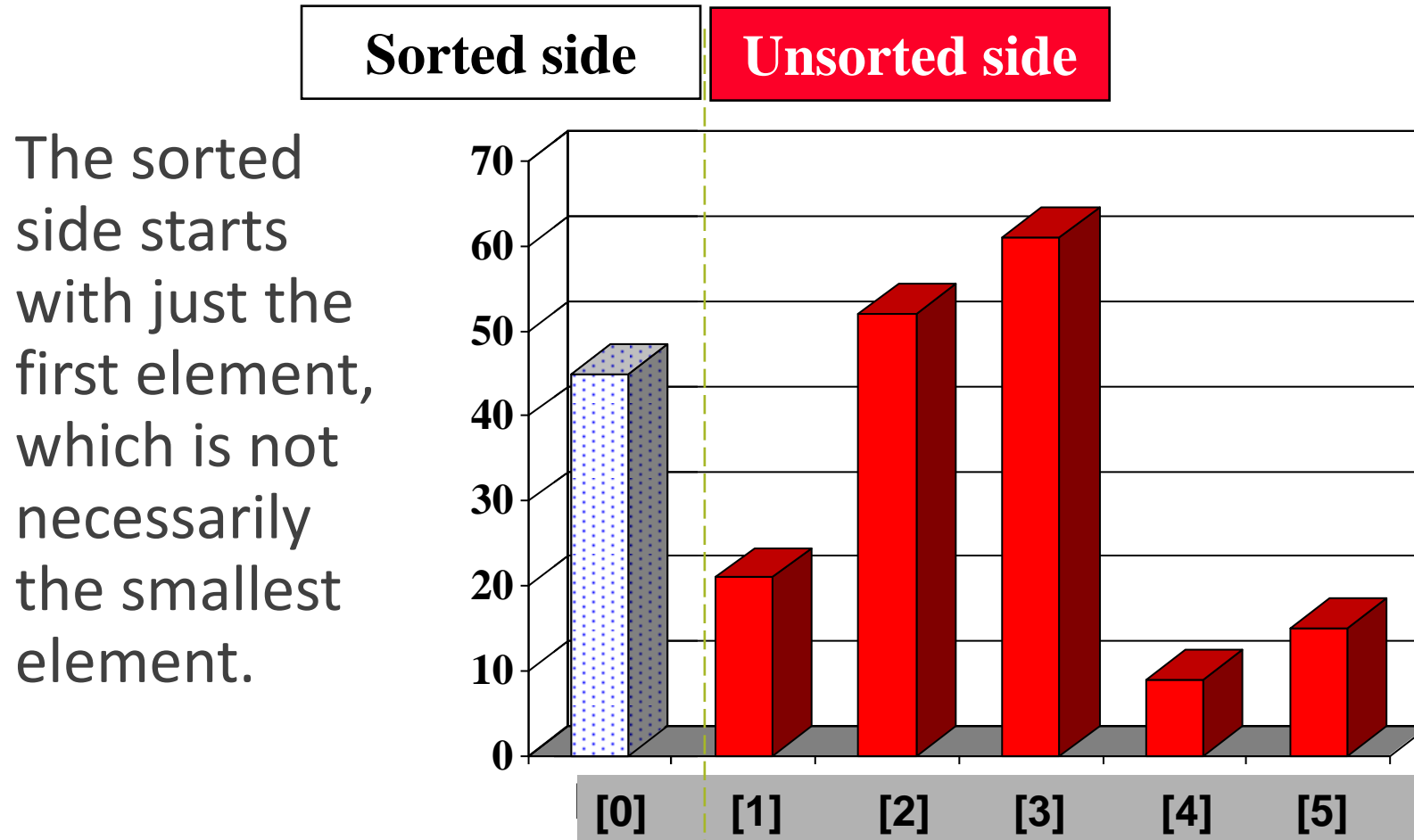
Insertion Sort

The Insertion Sort Algorithm

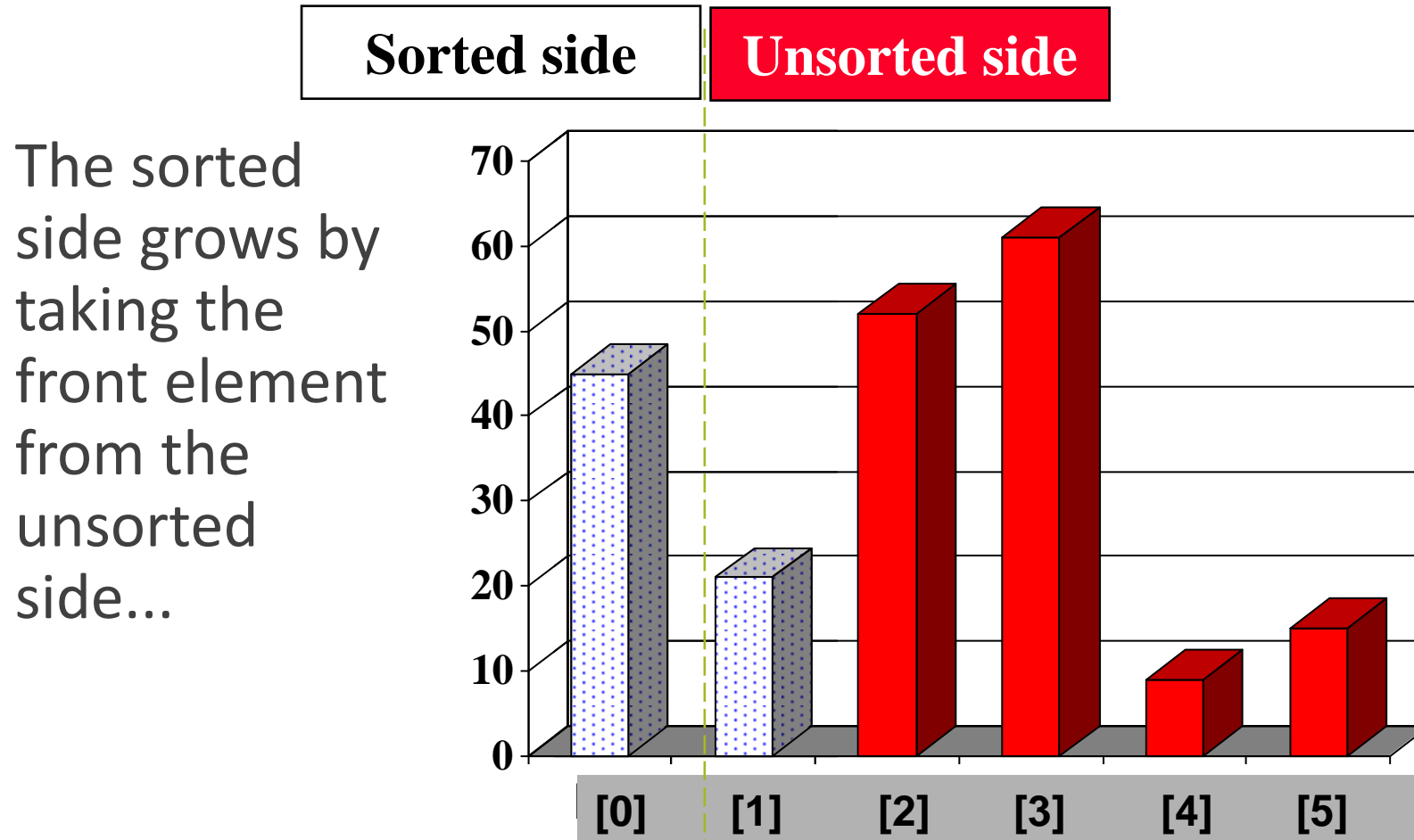
The Insertion Sort algorithm also views the array as having a sorted side and an unsorted side.



The Insertion Sort Algorithm

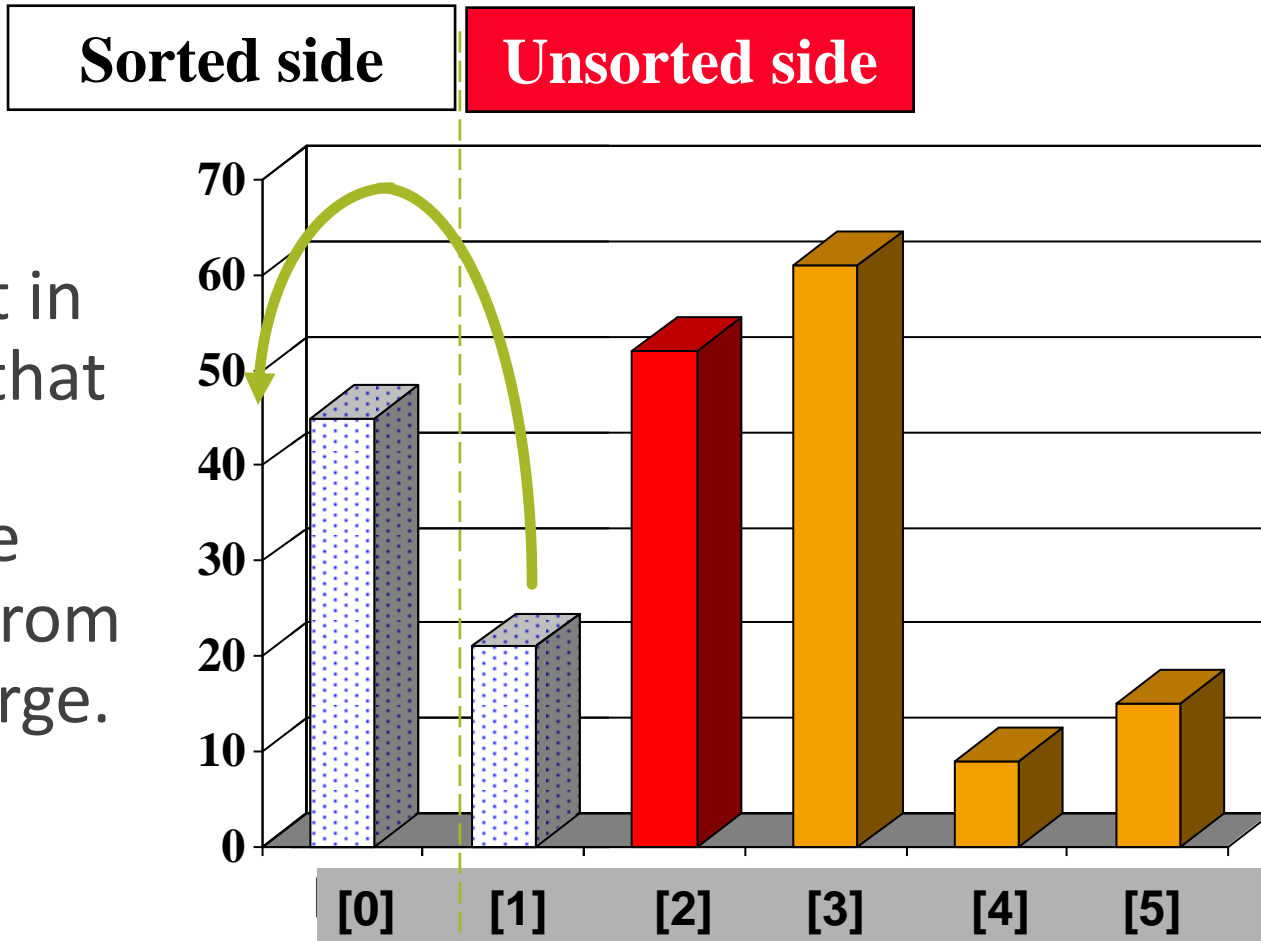


The Insertion Sort Algorithm

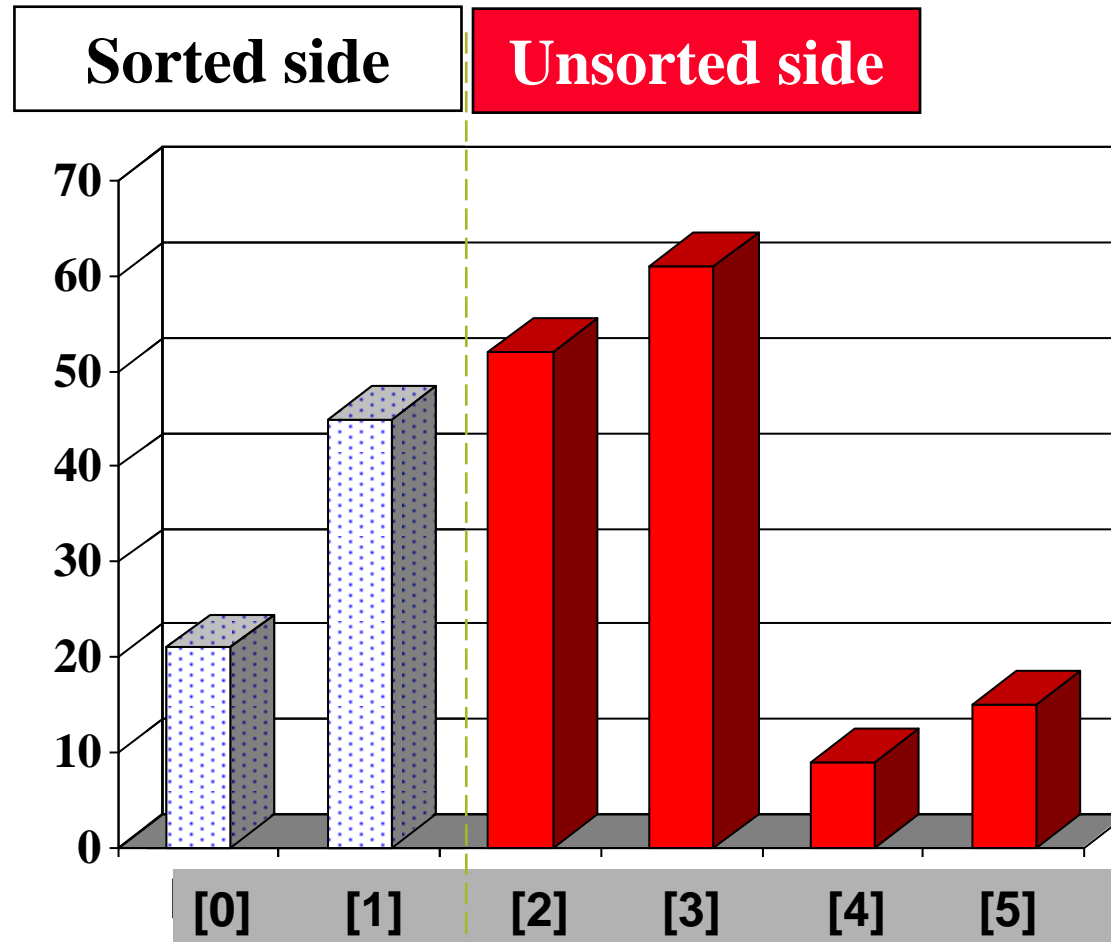


The Insertion Sort Algorithm

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

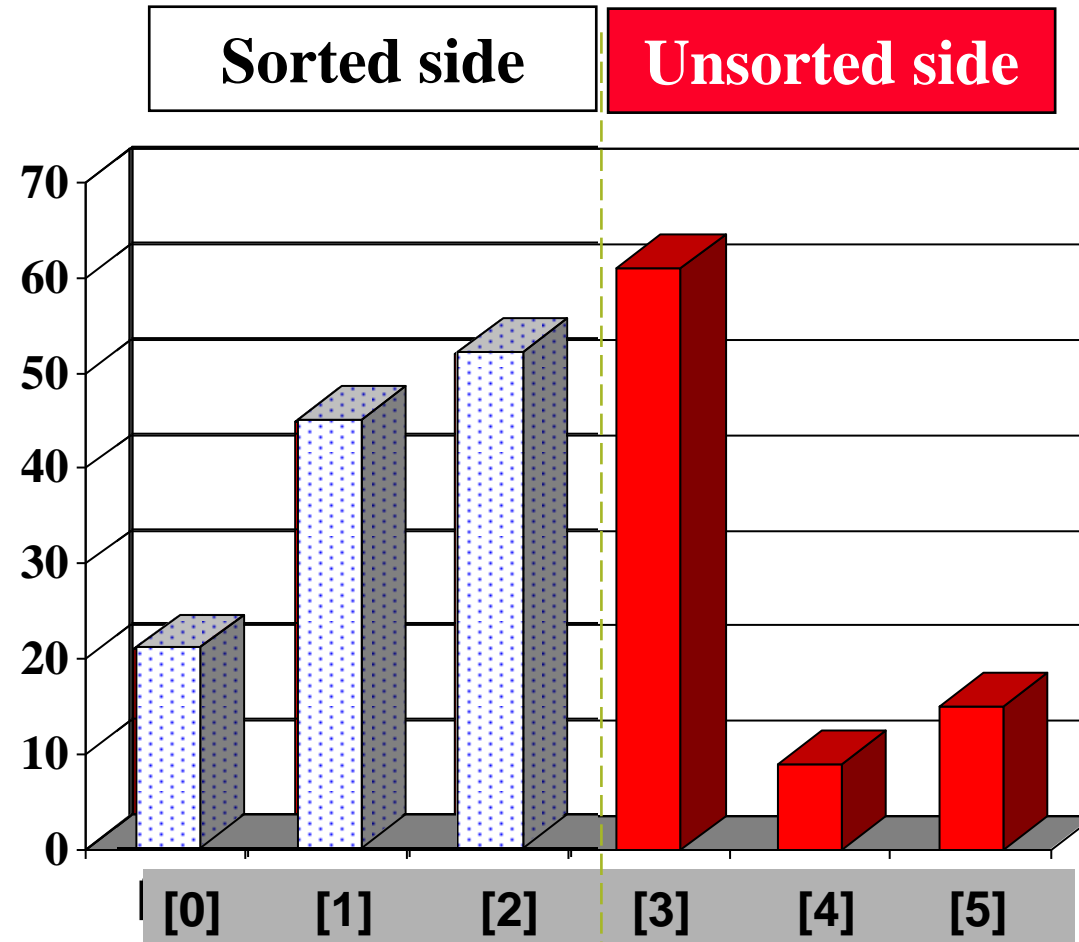


The Insertion Sort Algorithm



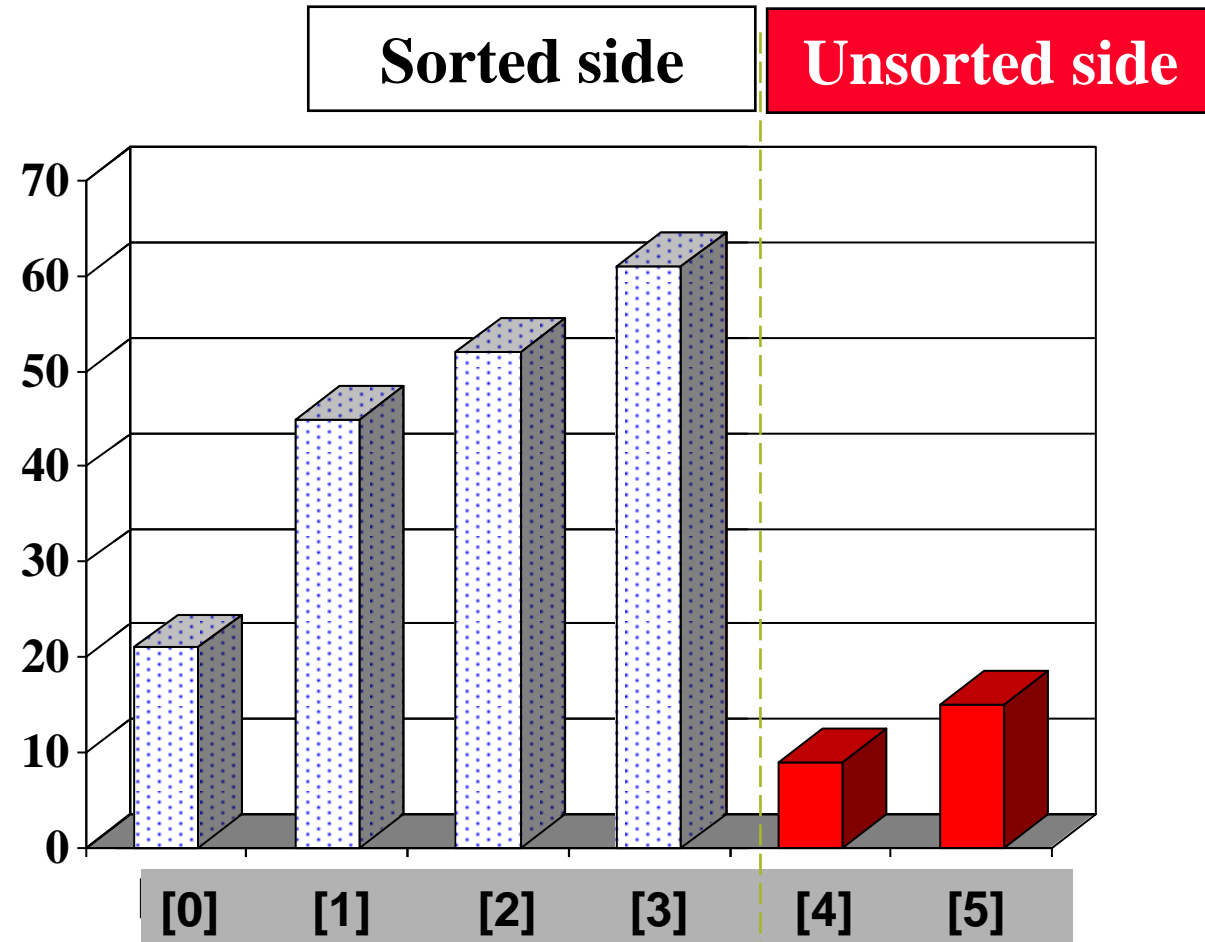
The Insertion Sort Algorithm

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



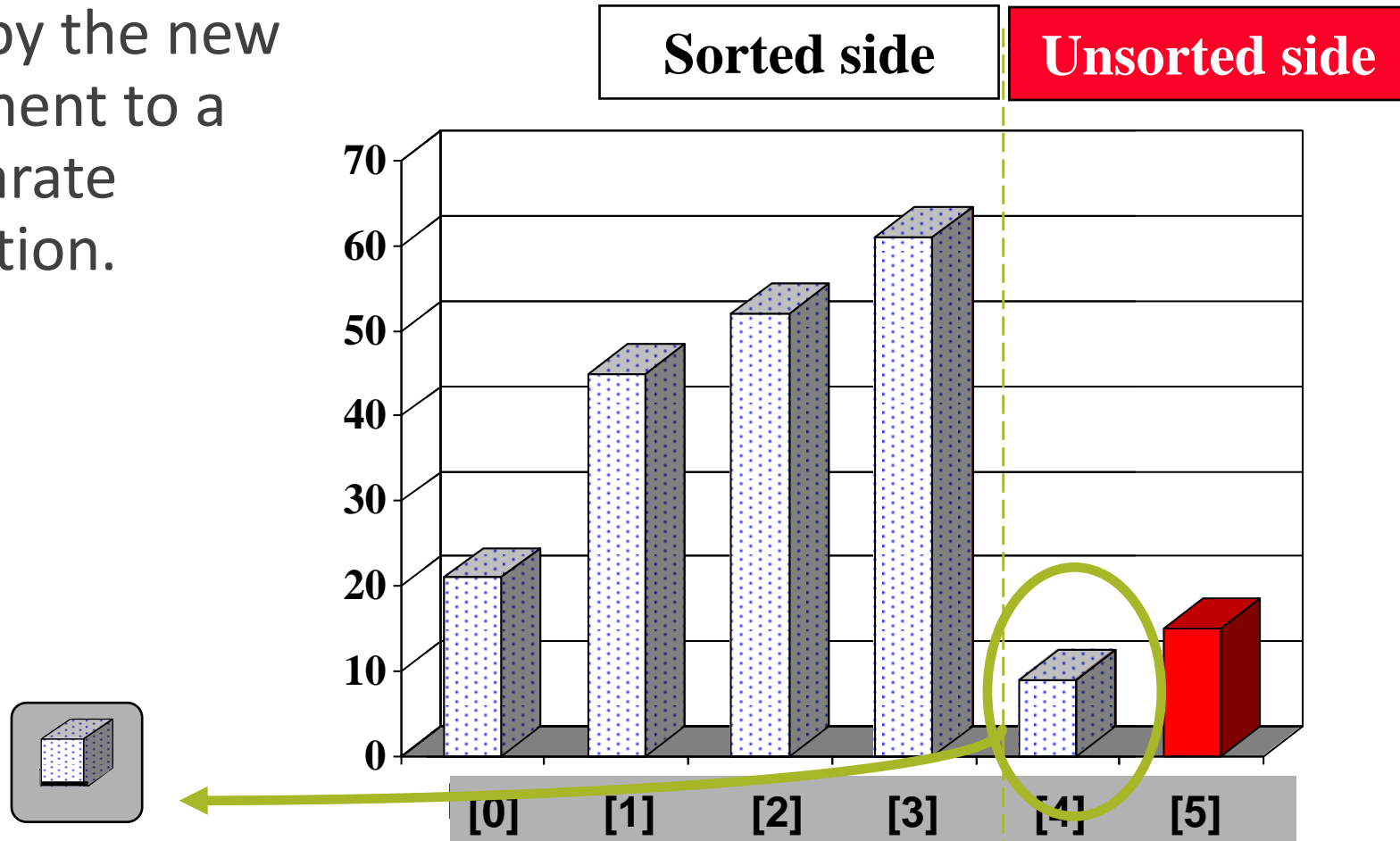
The Insertion Sort Algorithm

Sometimes we are lucky twice in a row.



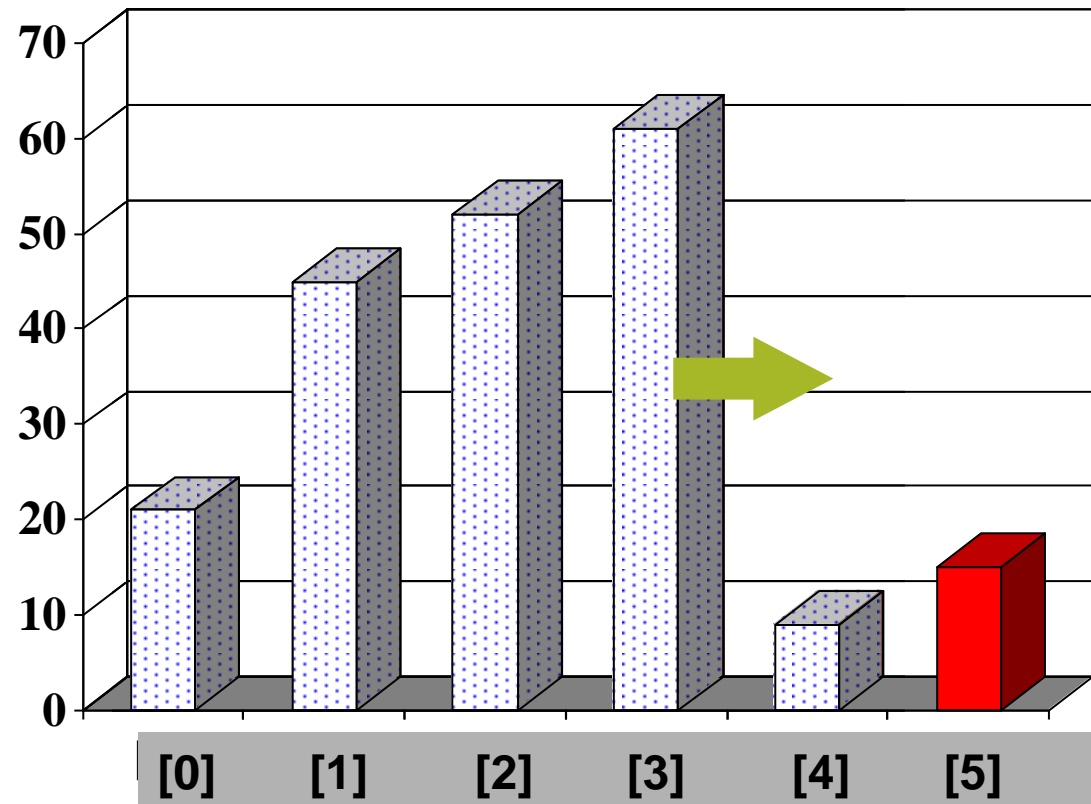
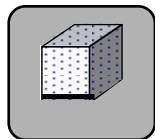
How to Insert One Element

- Copy the new element to a separate location.



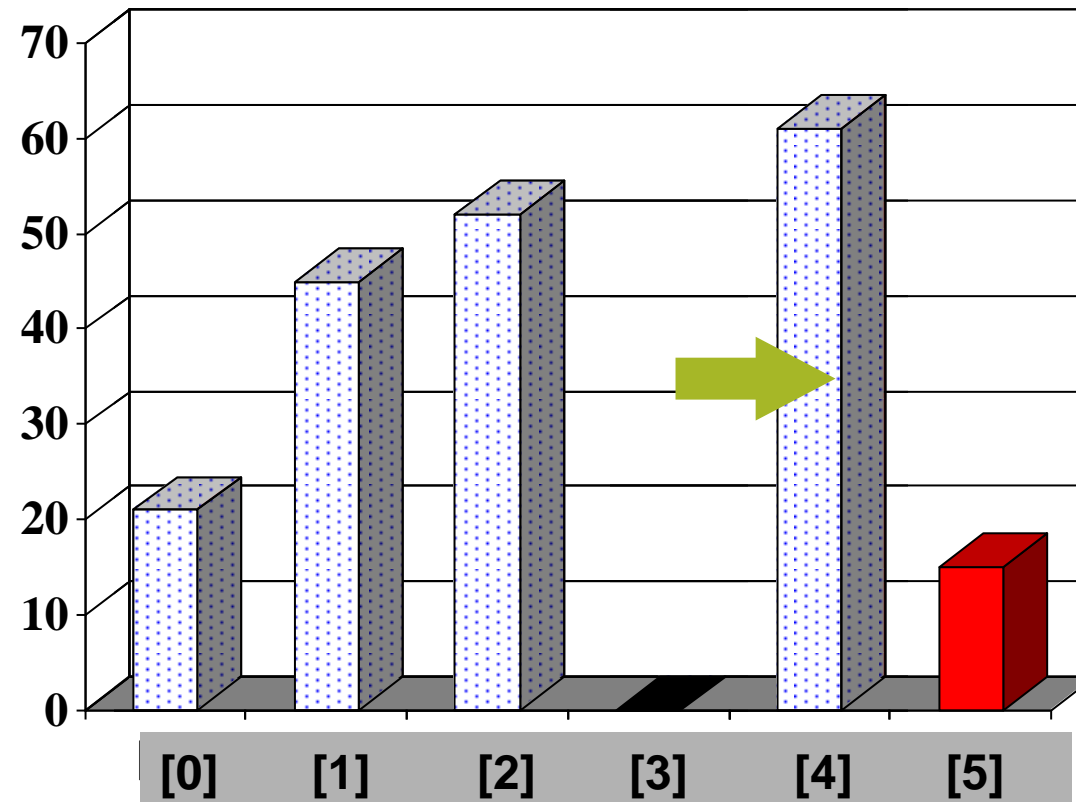
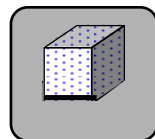
How to Insert One Element

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

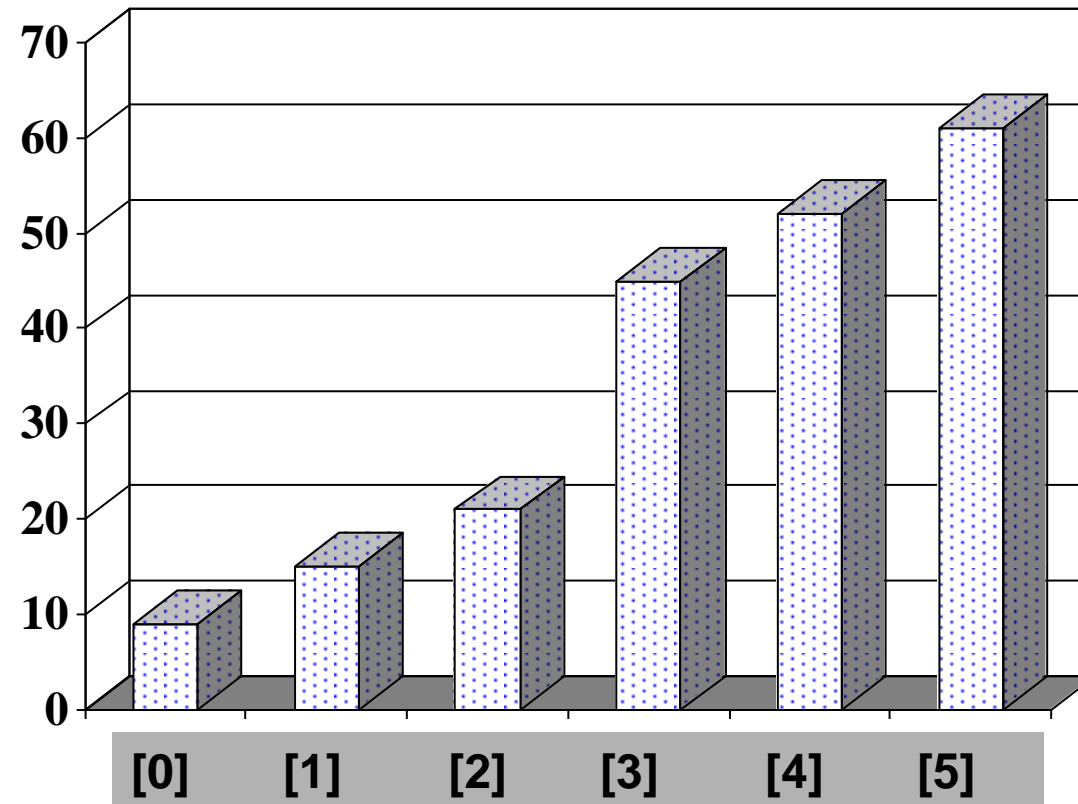


How to Insert One Element

□ 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 $n-1$

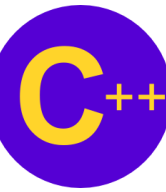
 take next key from unsorted part of array

 insert in appropriate location in sorted part of array:

 for $j = i$ down to 0,

 shift sorted elements to the right if $\text{key} > \text{key}[i]$

 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[j]
  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)

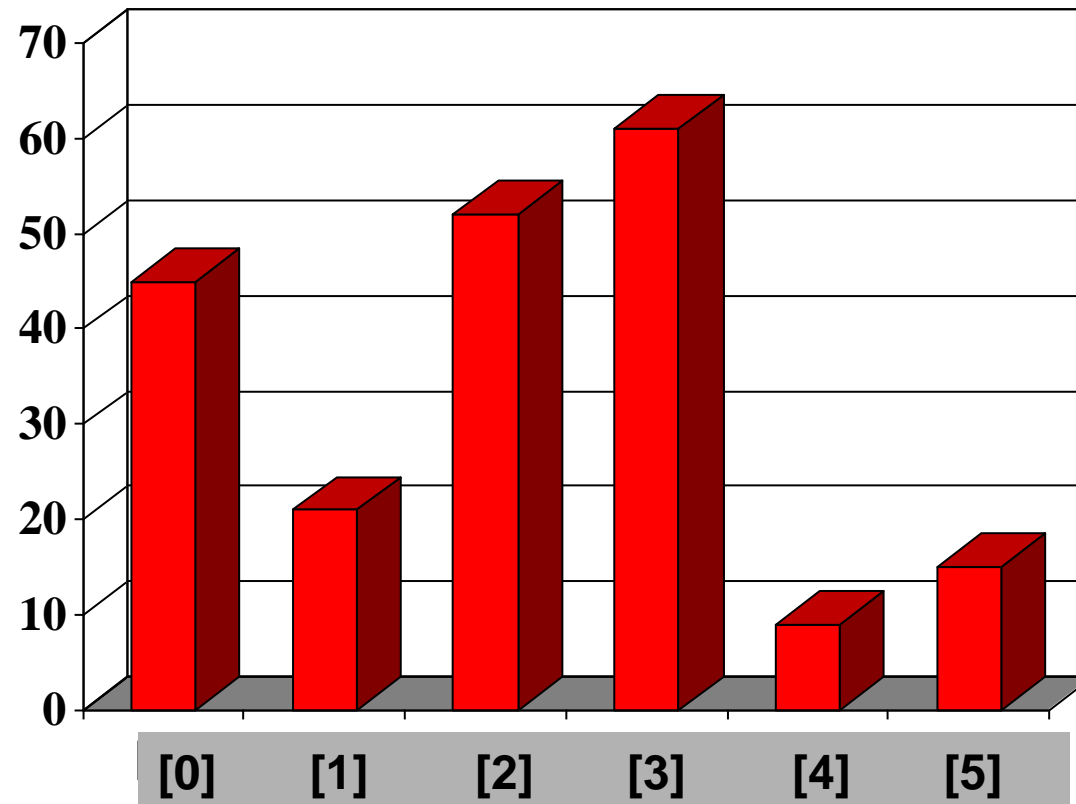
O(n)

LECTURE 6

Bubble Sort

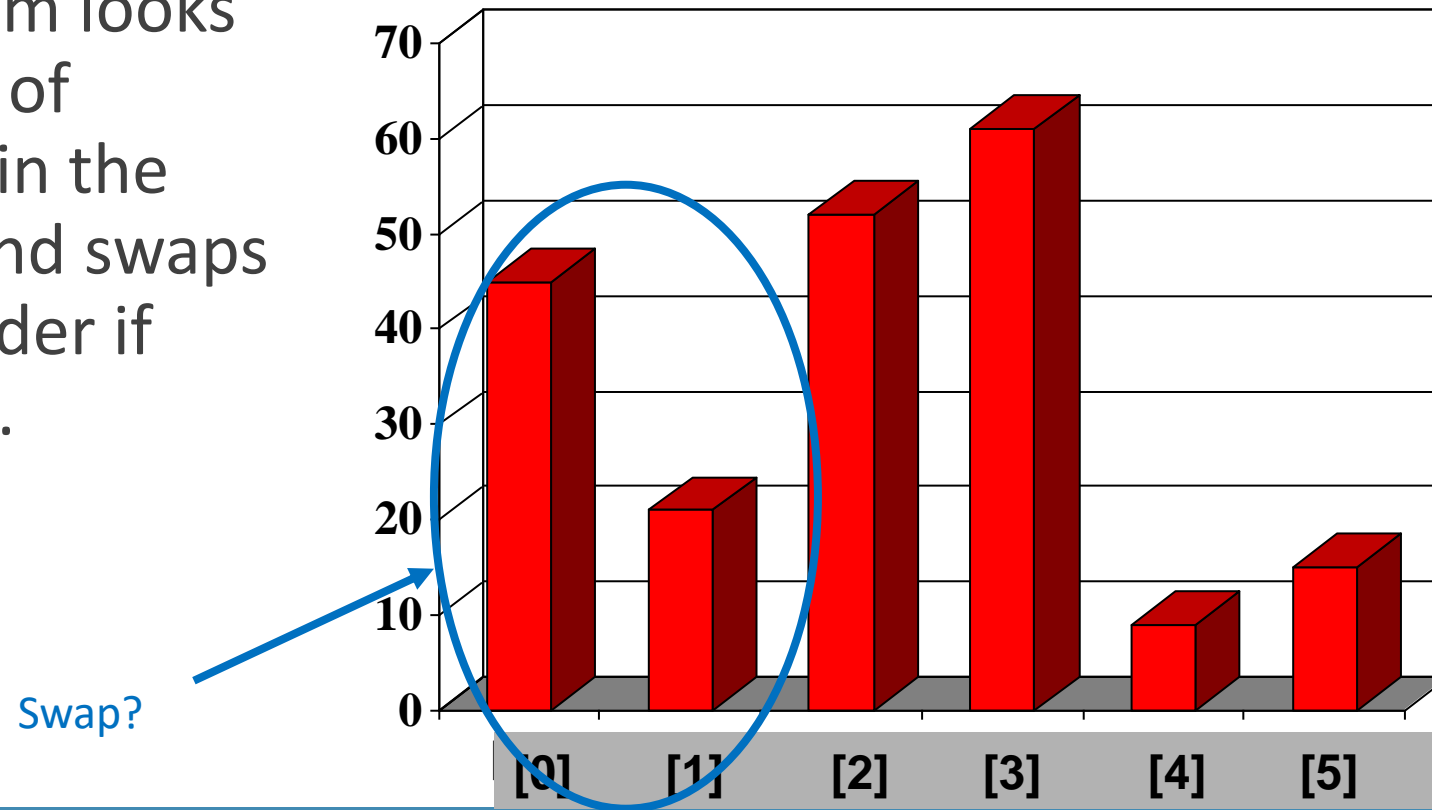
The Bubble Sort Algorithm

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



The Bubble Sort Algorithm

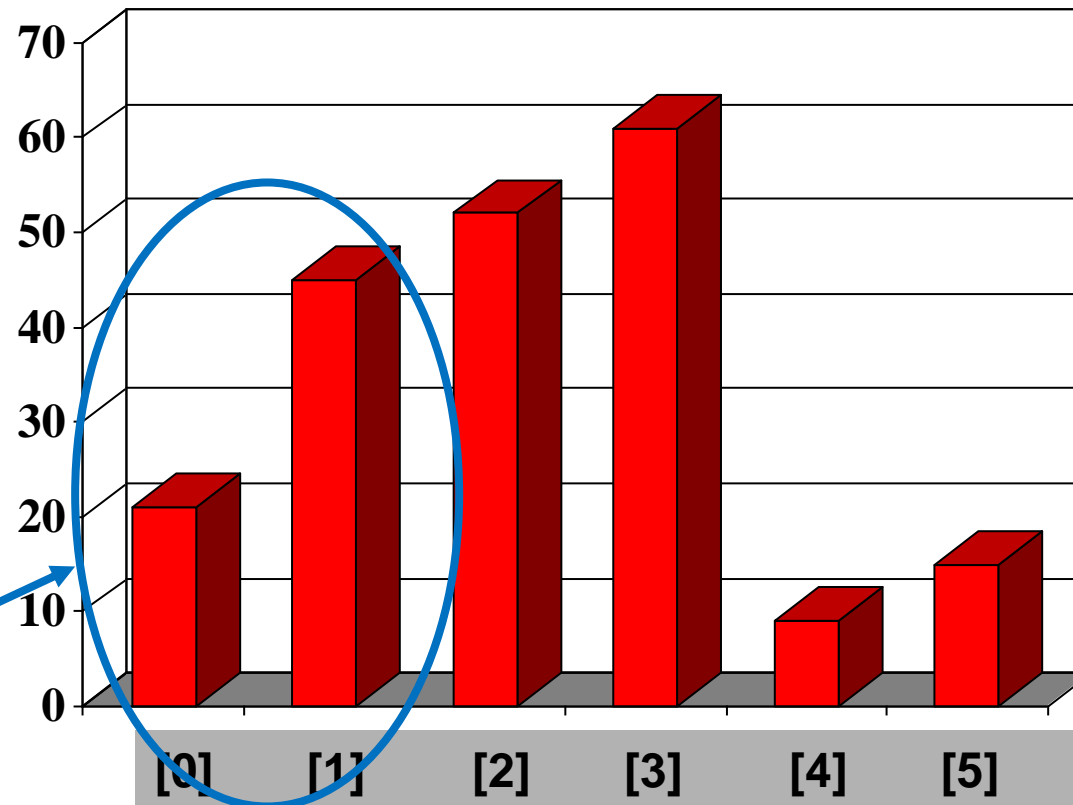
The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



The Bubble Sort Algorithm

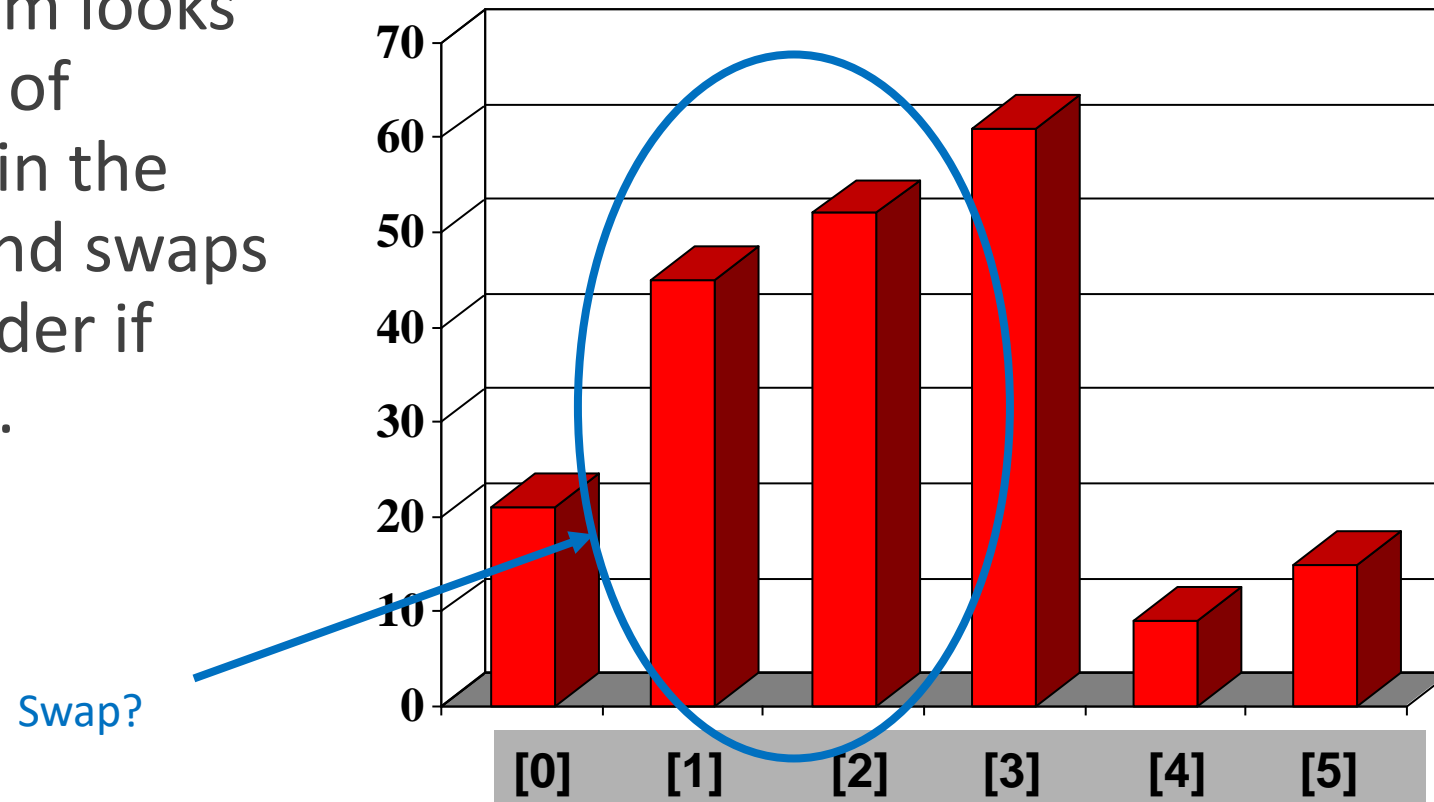
The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.

Yes!



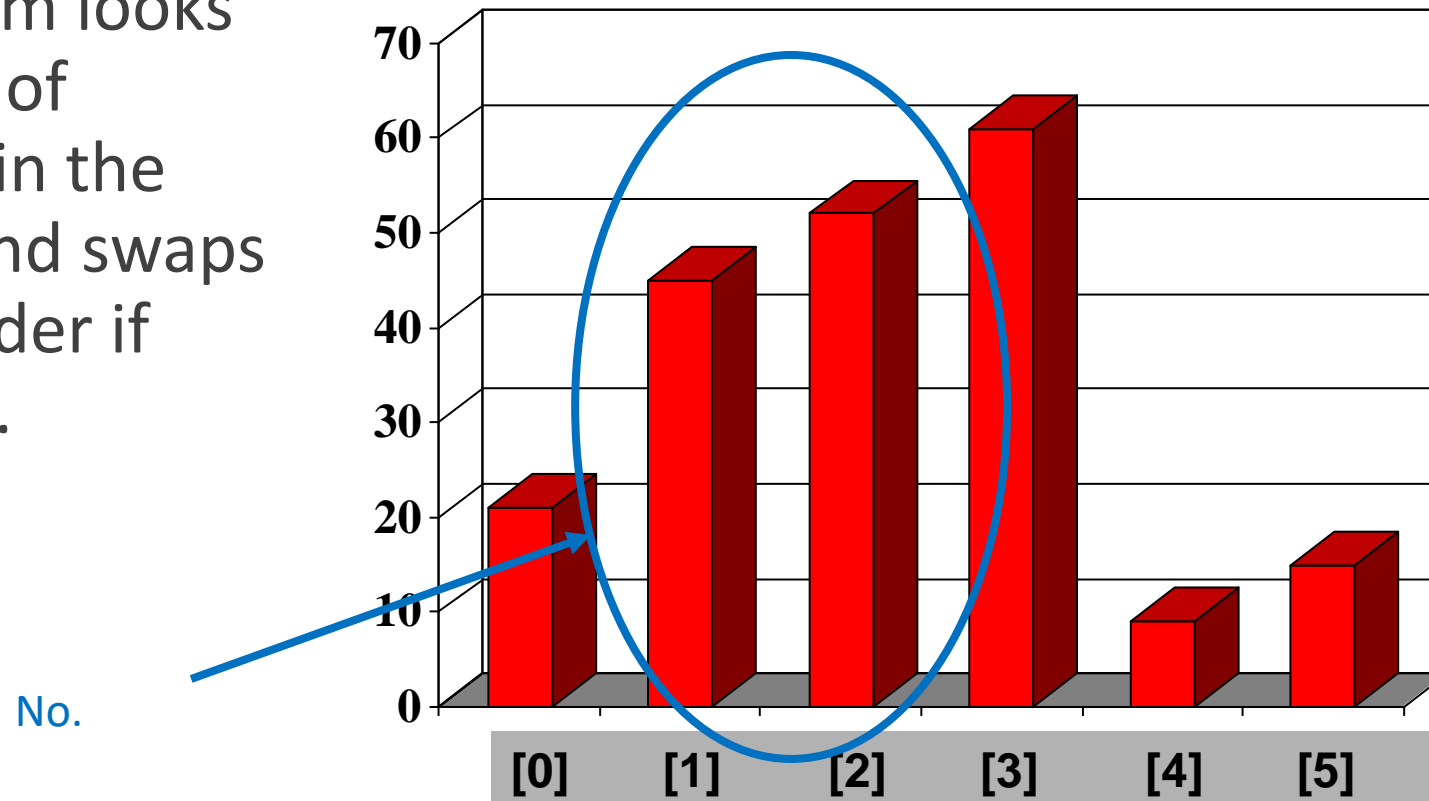
The Bubble Sort Algorithm

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



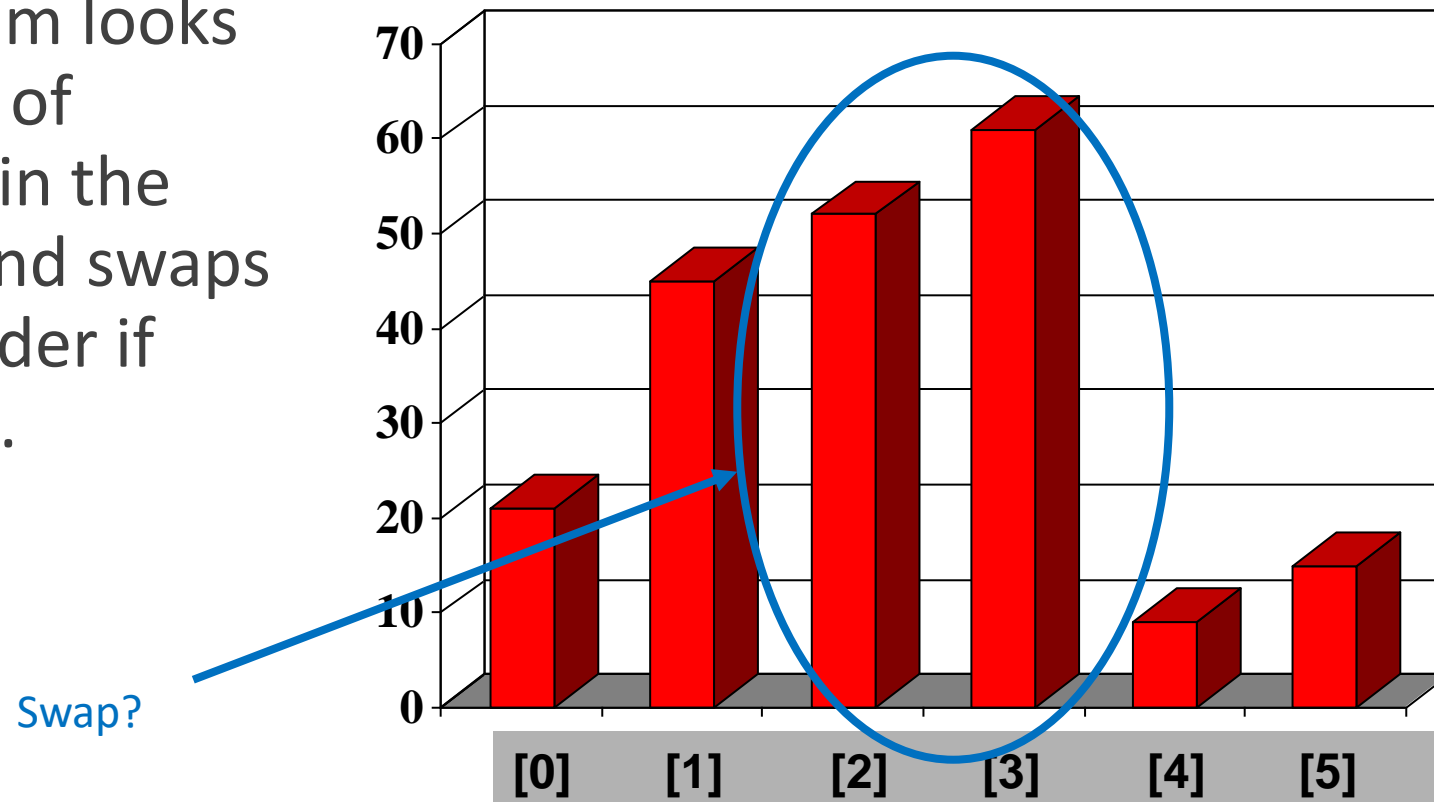
The Bubble Sort Algorithm

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



The Bubble Sort Algorithm

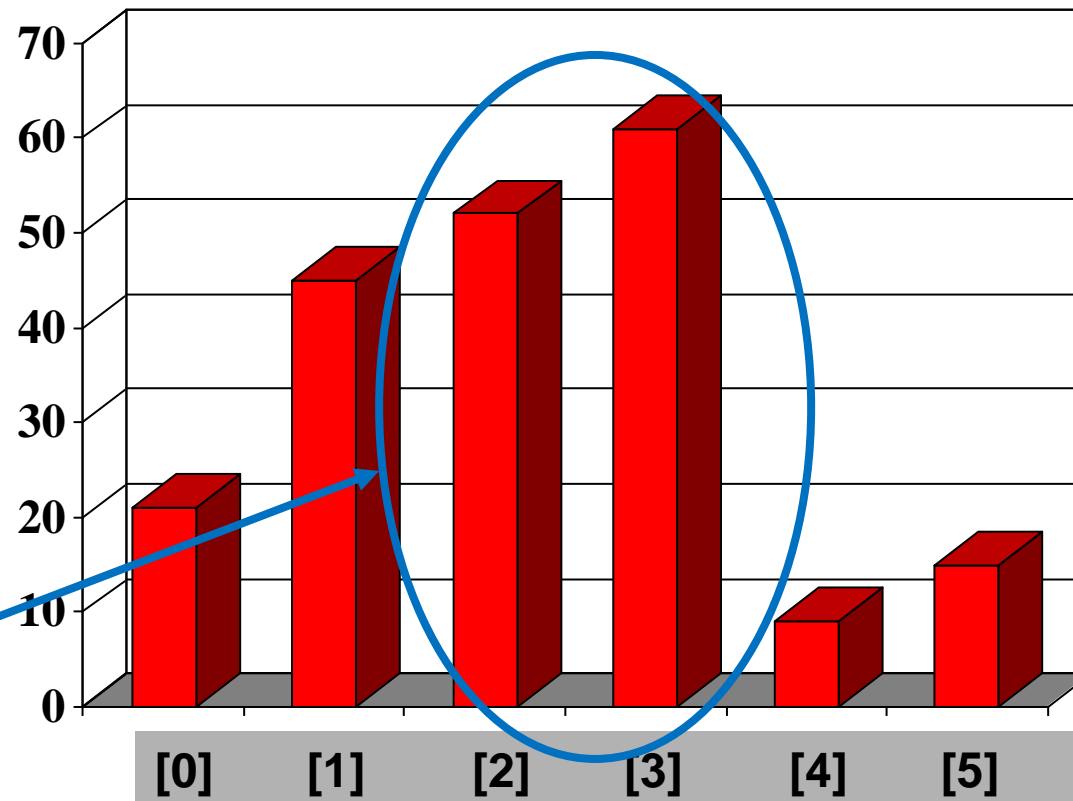
The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



The Bubble Sort Algorithm

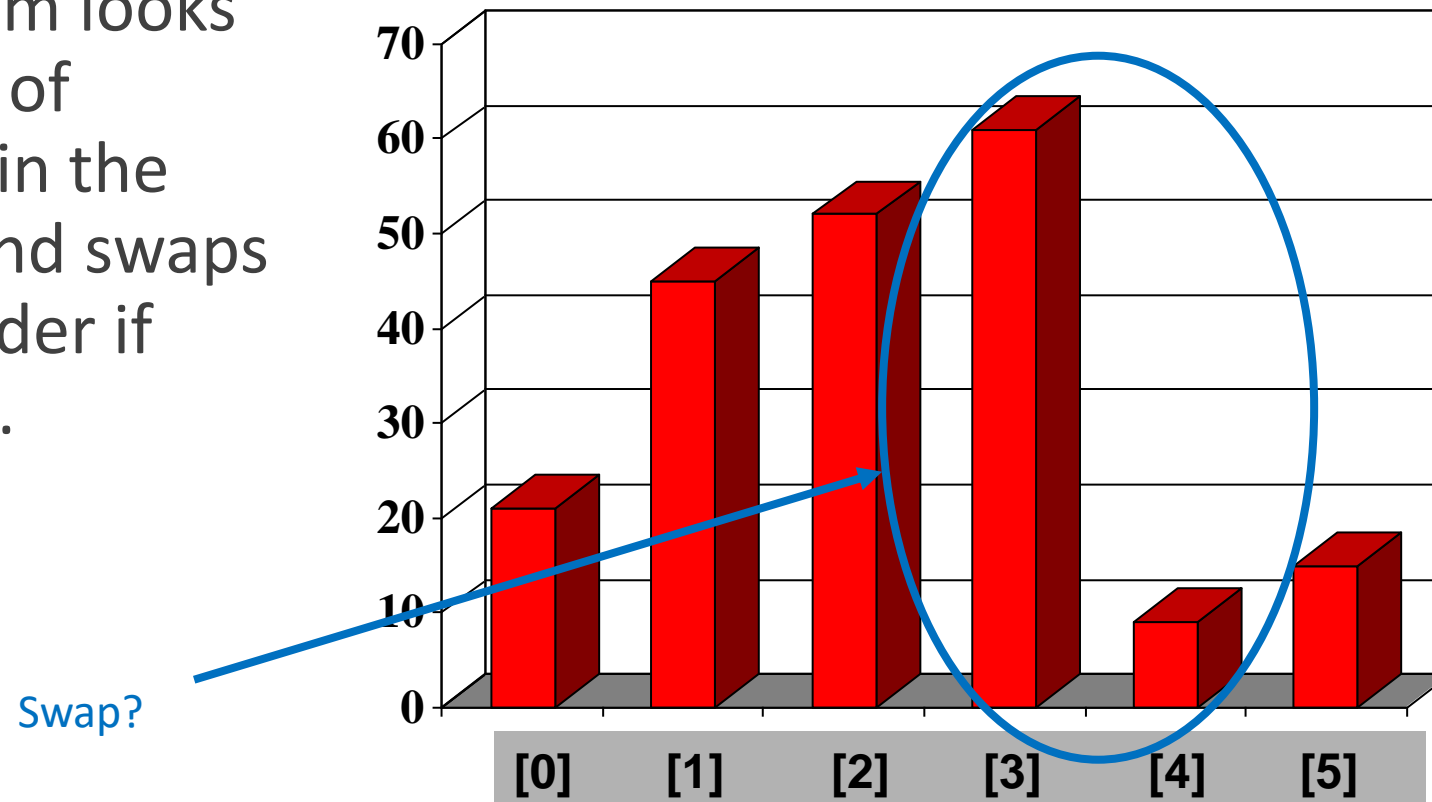
The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.

No.



The Bubble Sort Algorithm

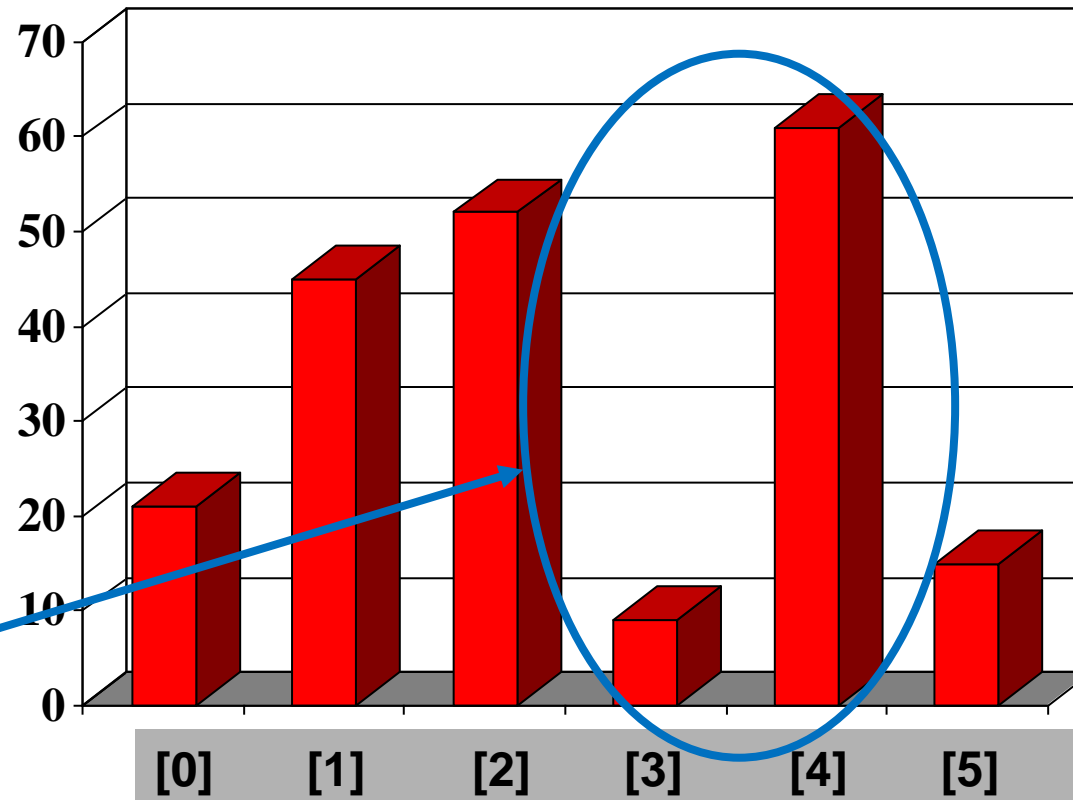
The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



The Bubble Sort Algorithm

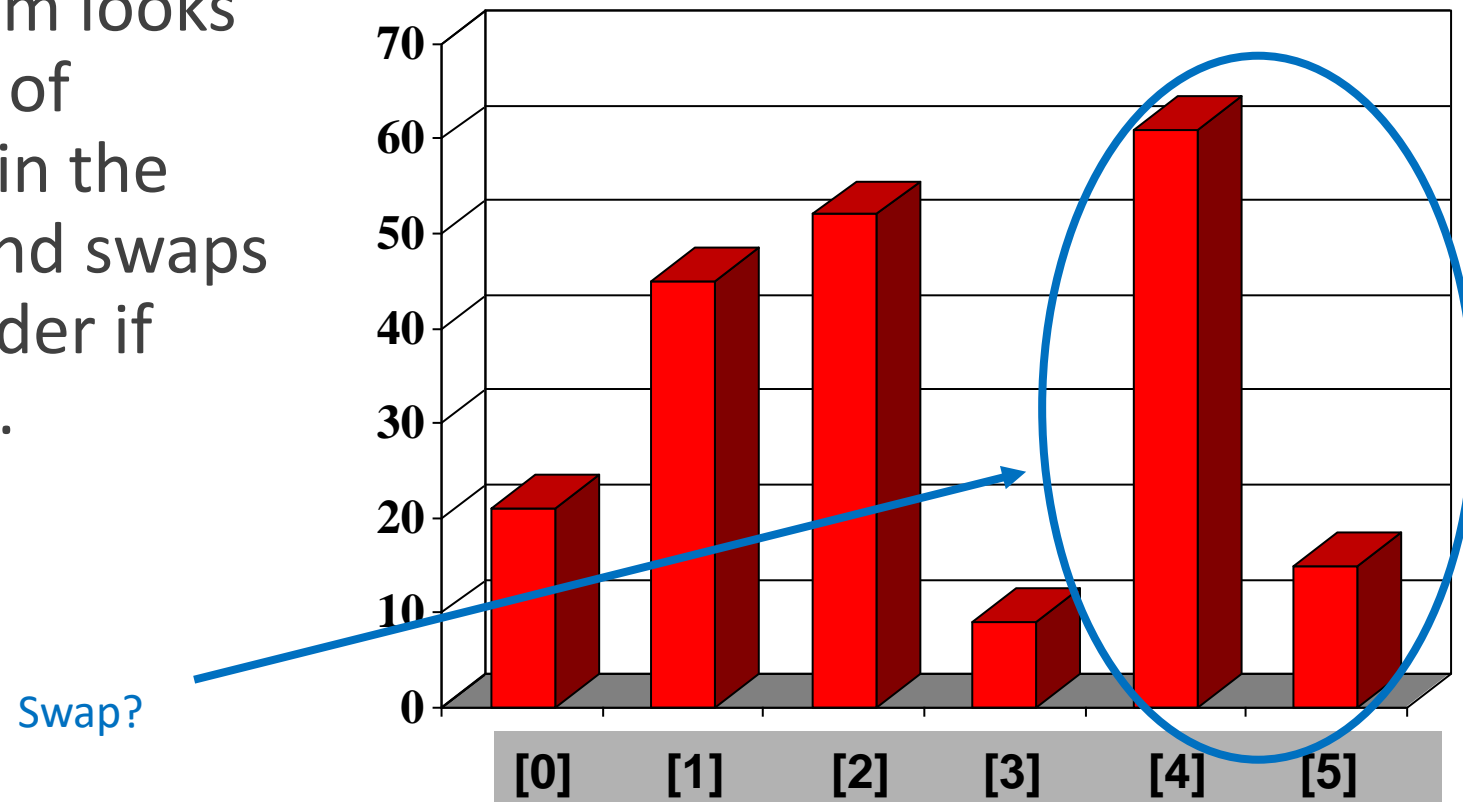
The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.

Yes!



The Bubble Sort Algorithm

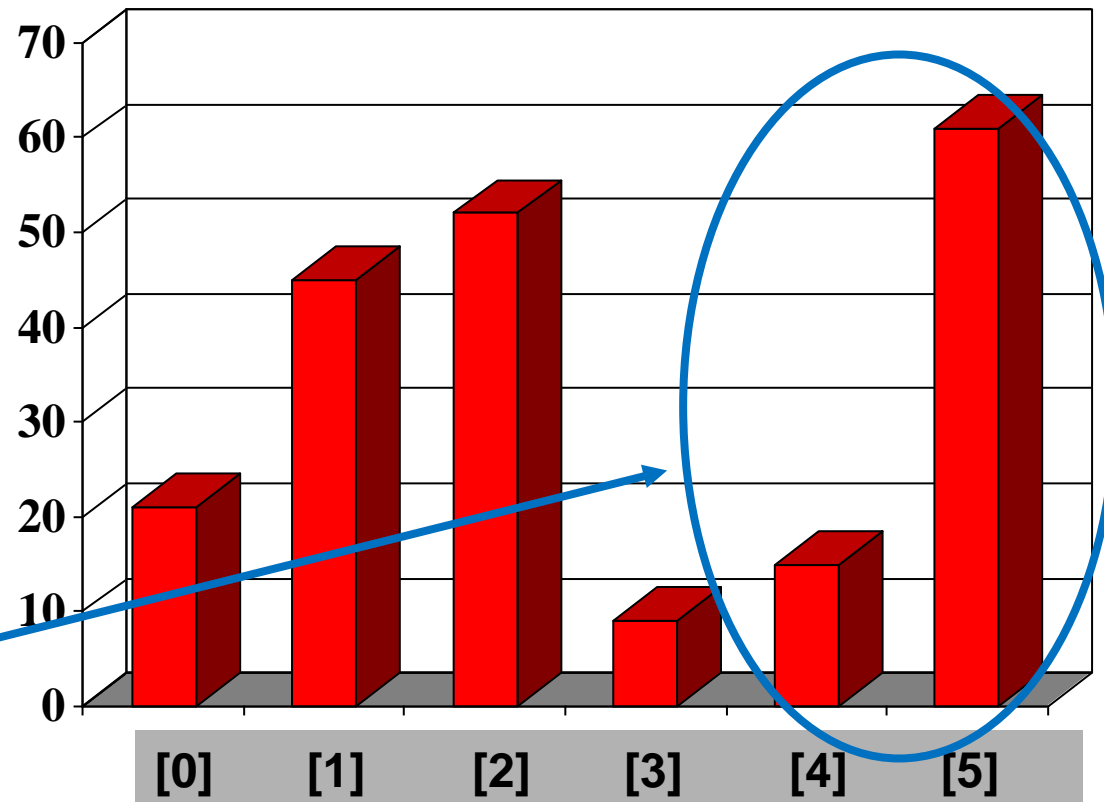
The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



The Bubble Sort Algorithm

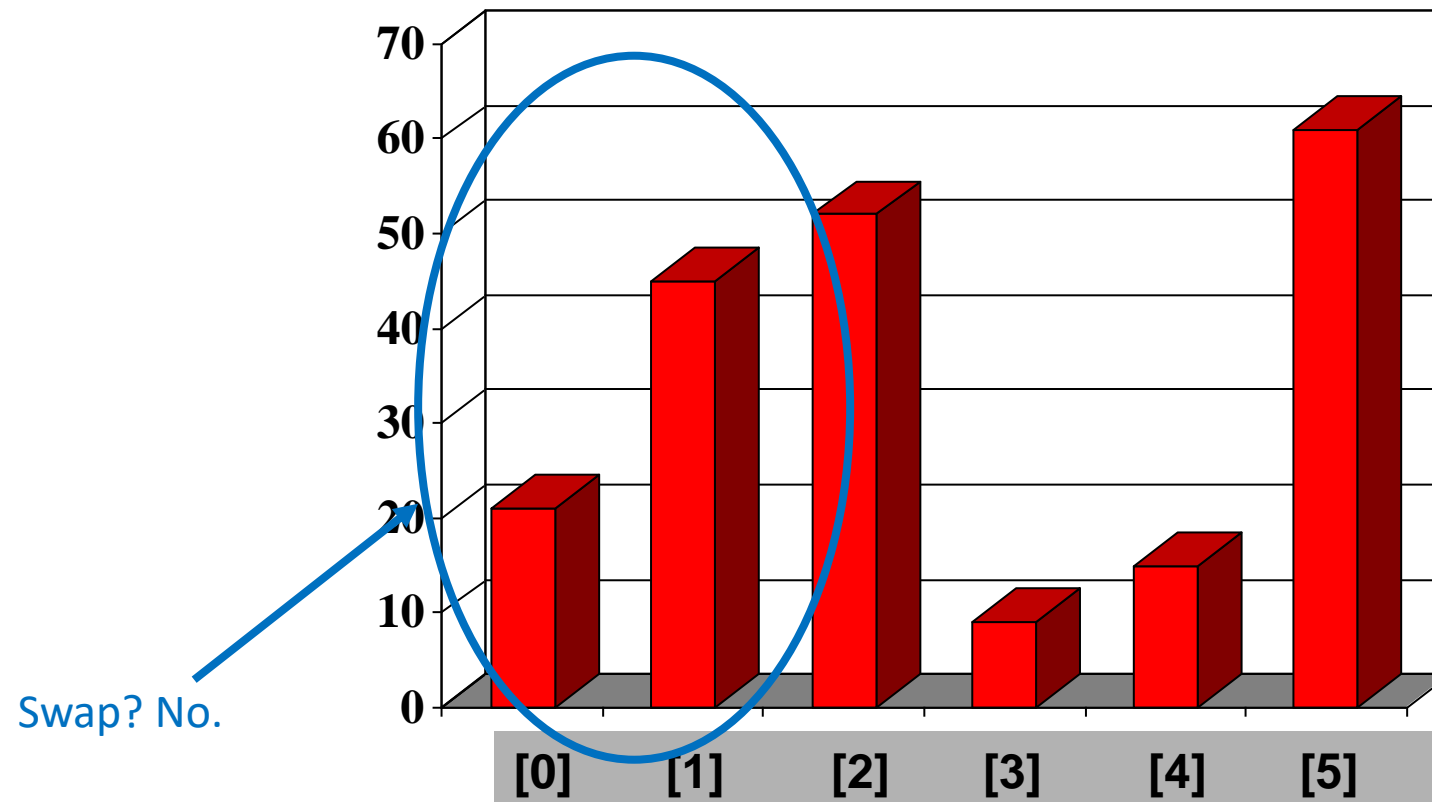
The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.

Yes!



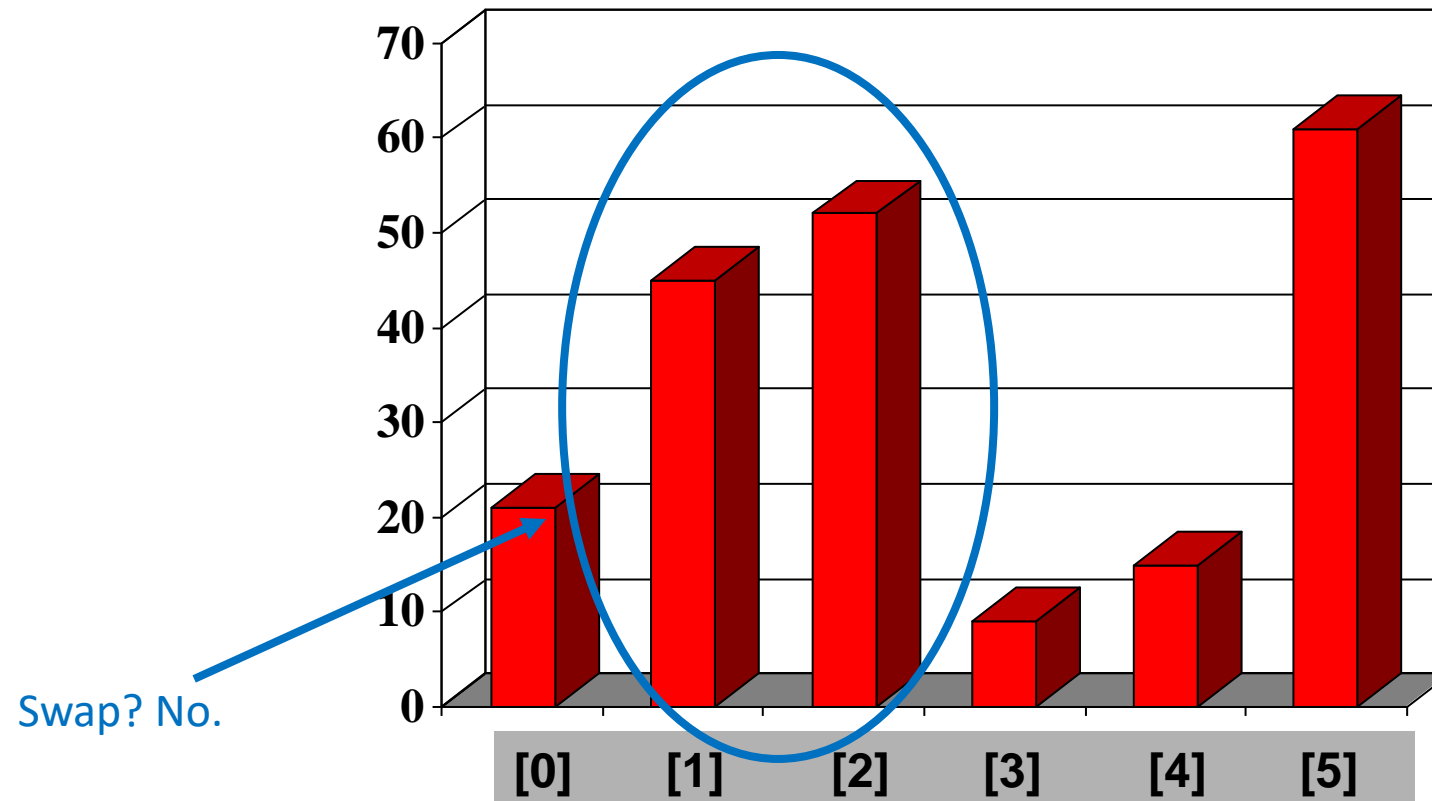
The Bubble Sort Algorithm

Repeat.



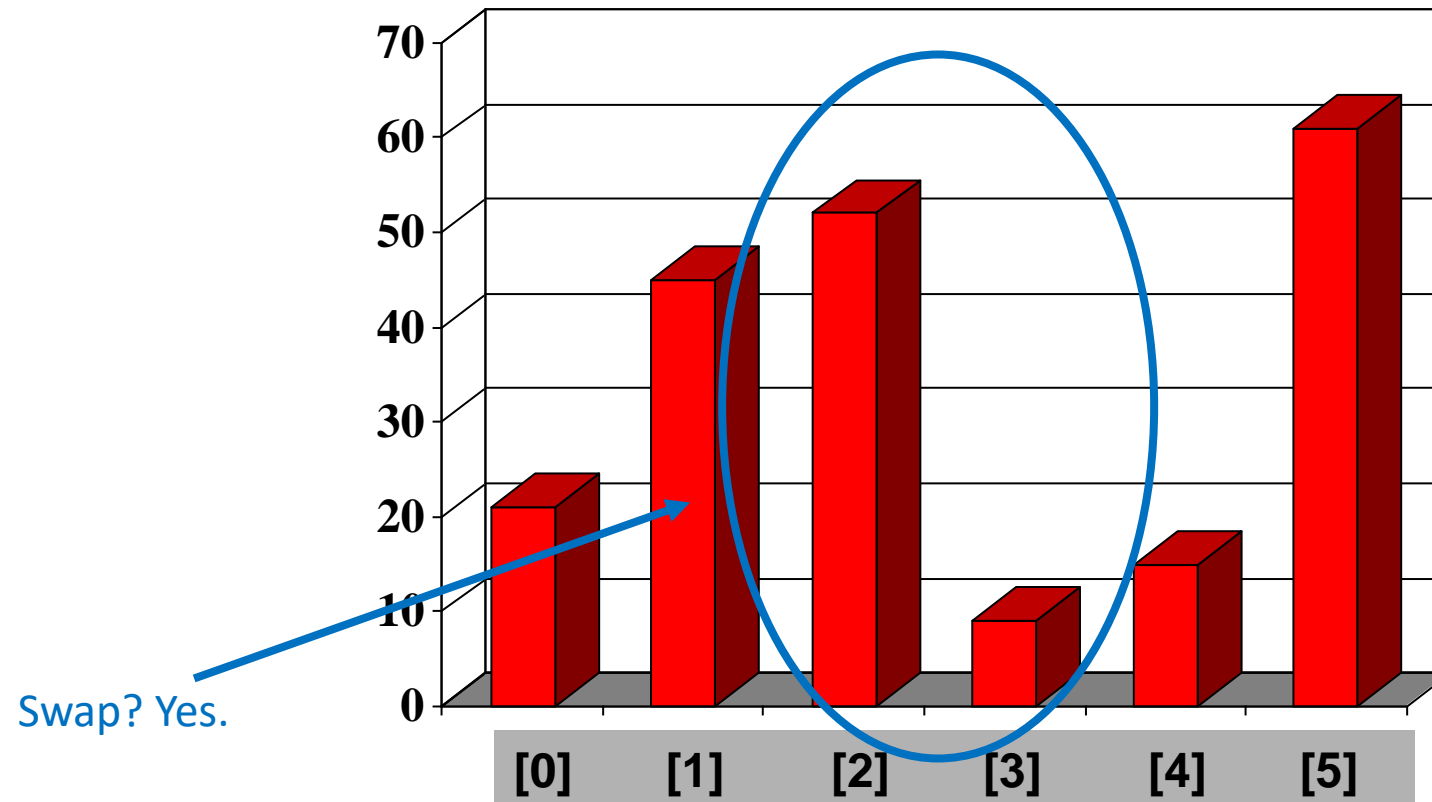
The Bubble Sort Algorithm

Repeat.



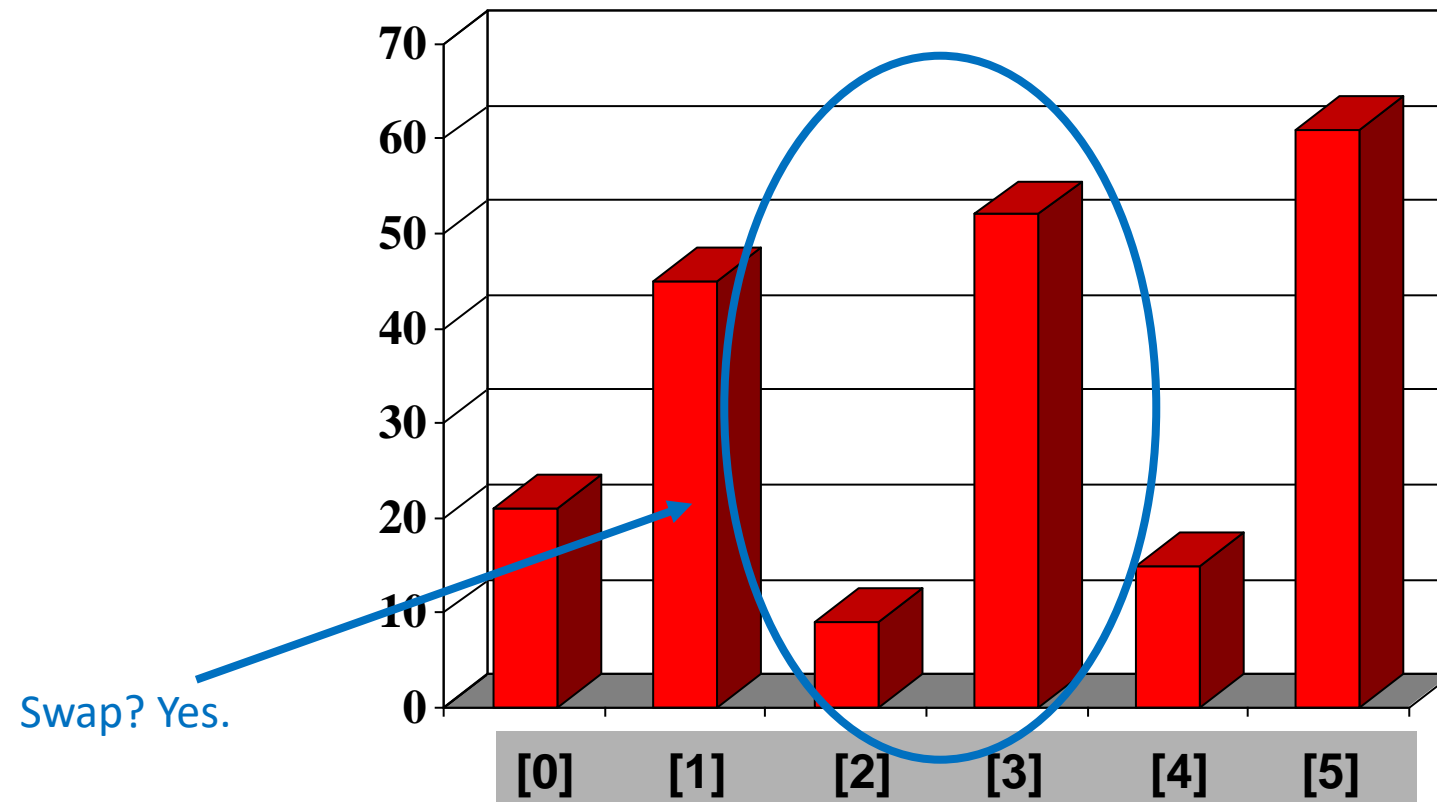
The Bubble Sort Algorithm

Repeat.



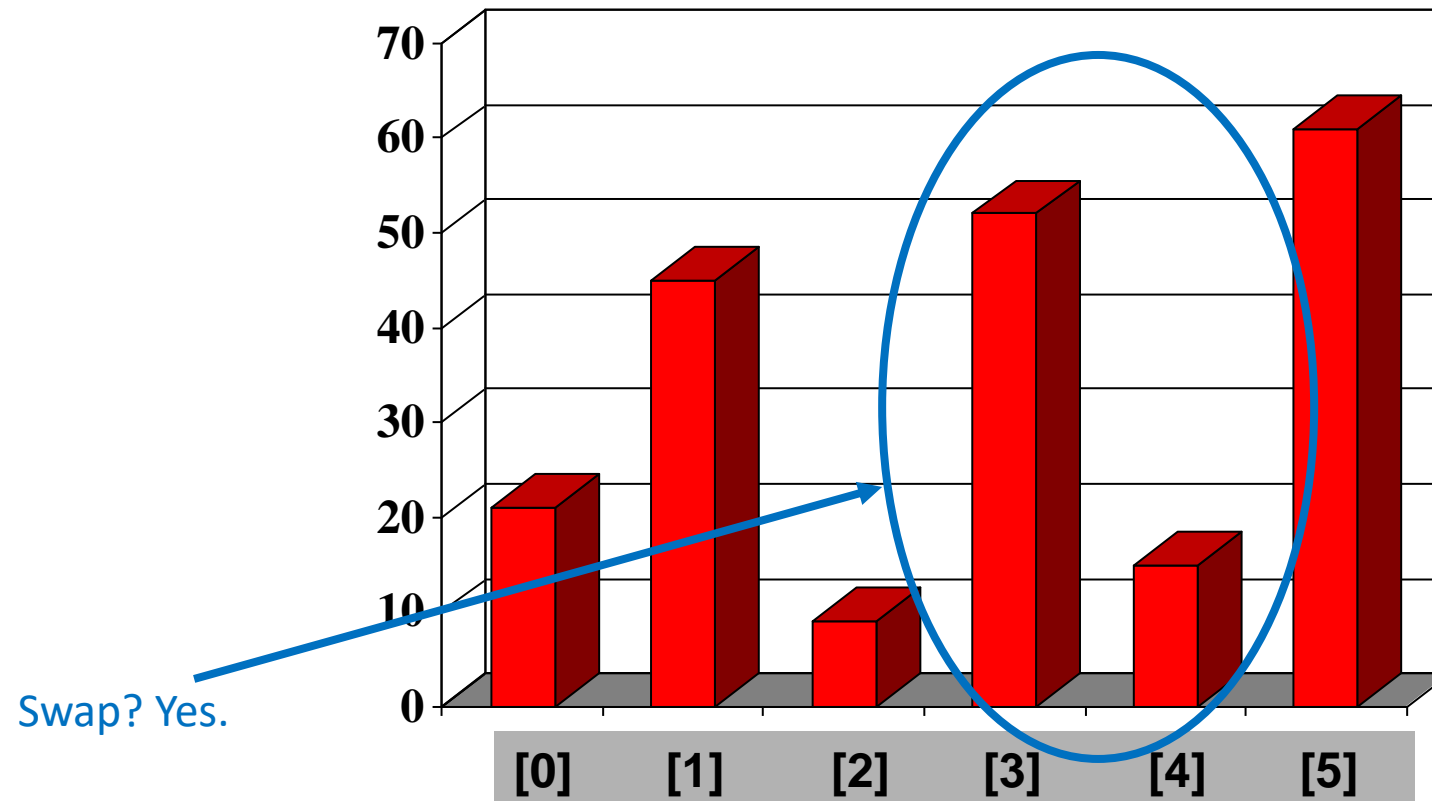
The Bubble Sort Algorithm

Repeat.



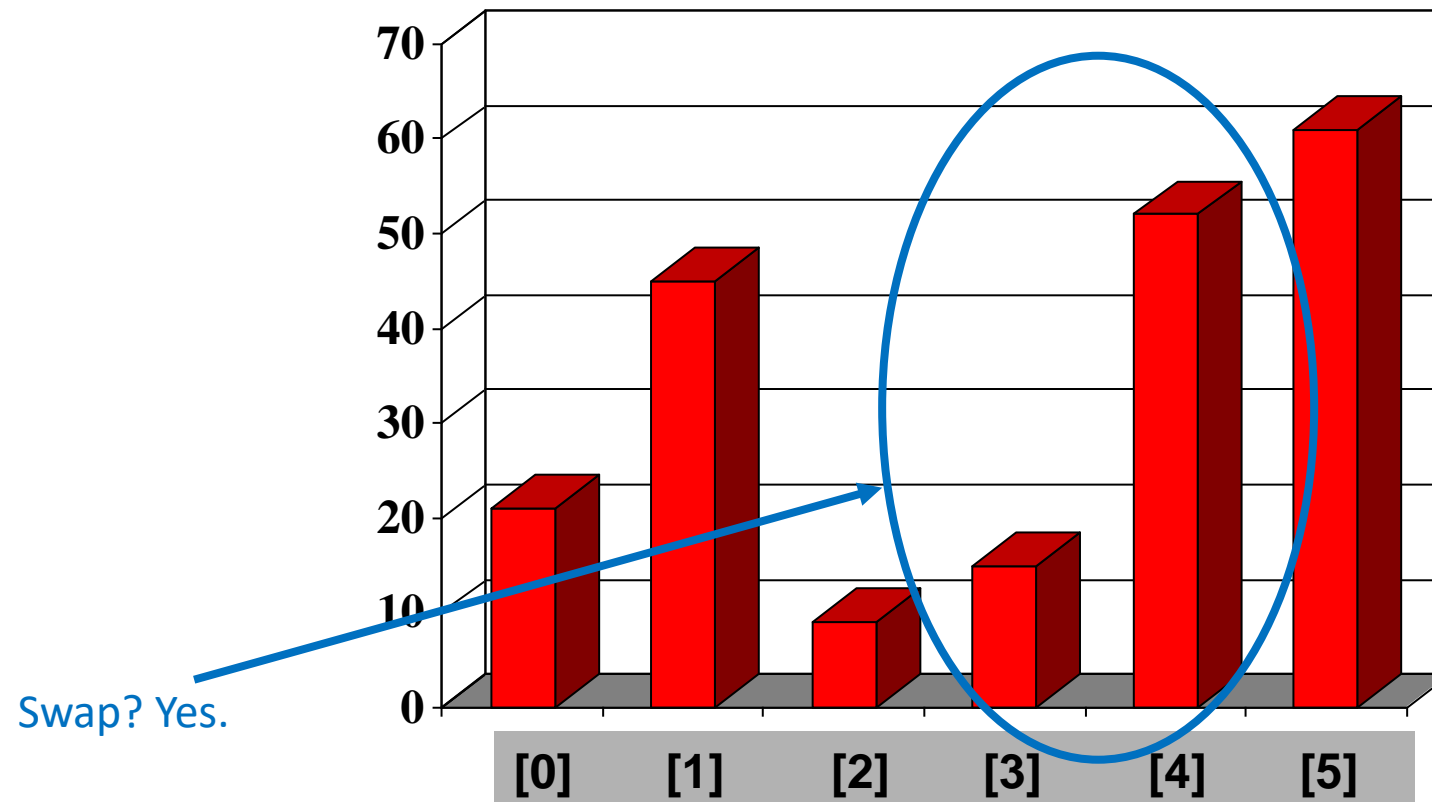
The Bubble Sort Algorithm

Repeat.



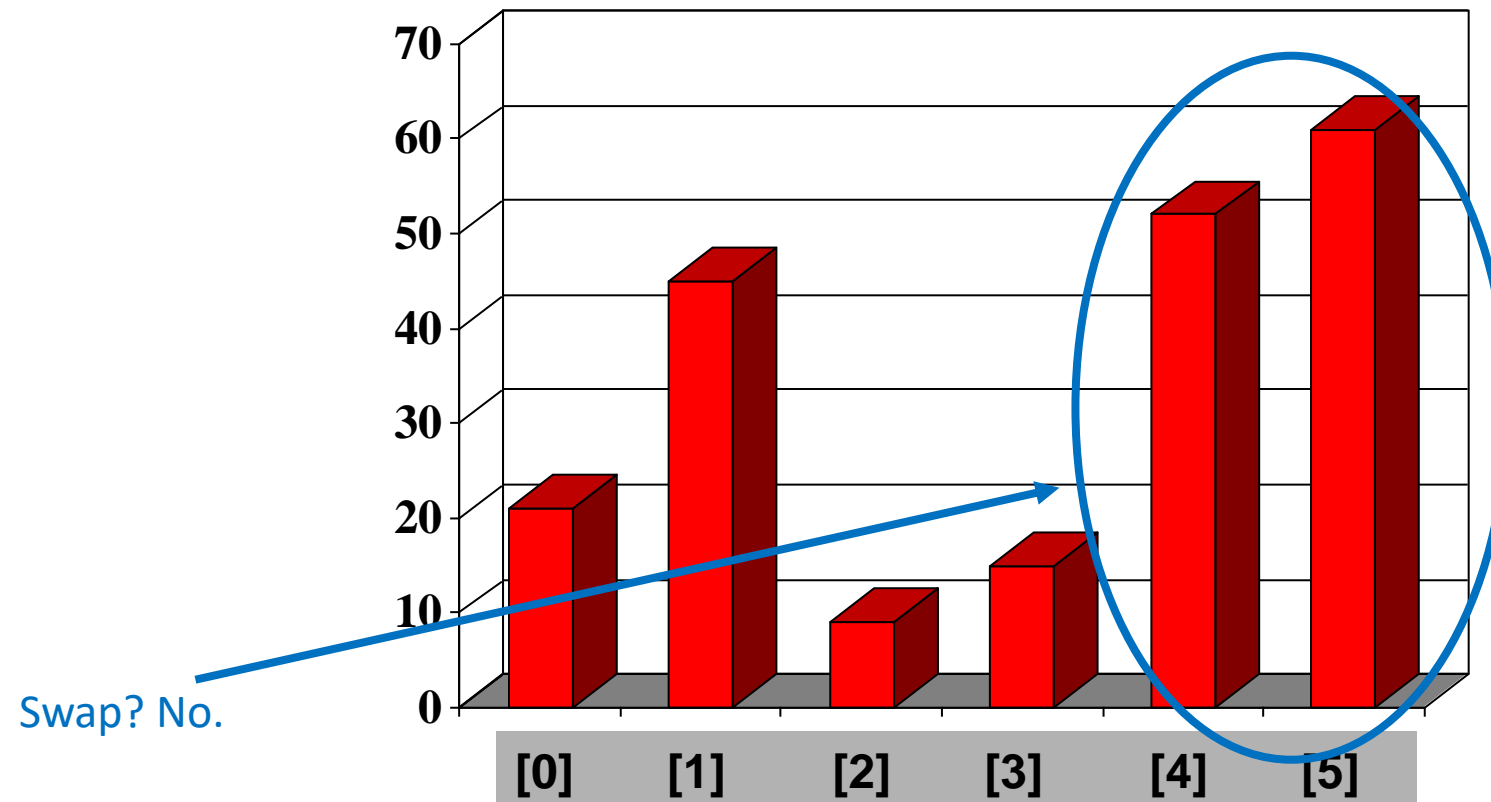
The Bubble Sort Algorithm

Repeat.



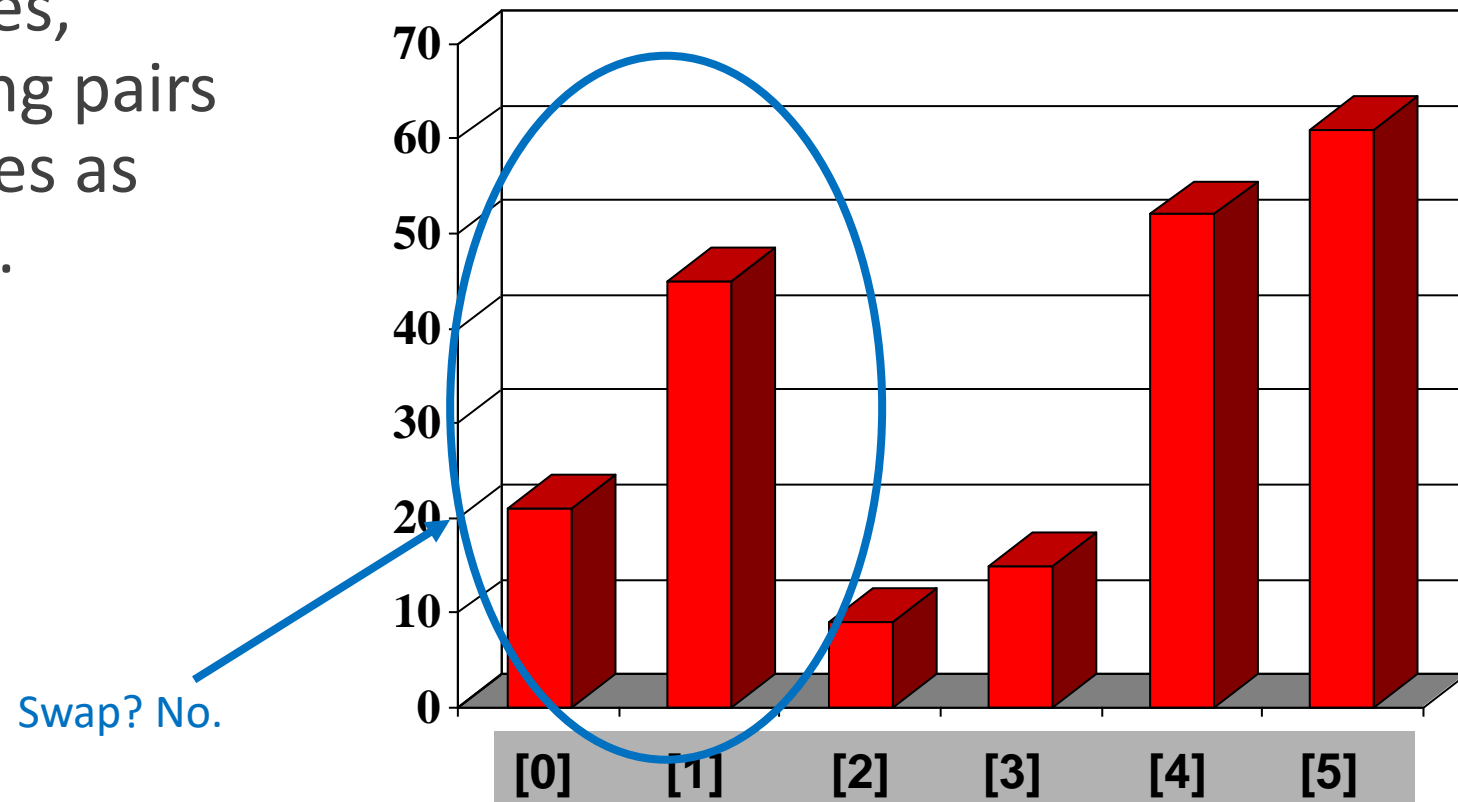
The Bubble Sort Algorithm

Repeat.



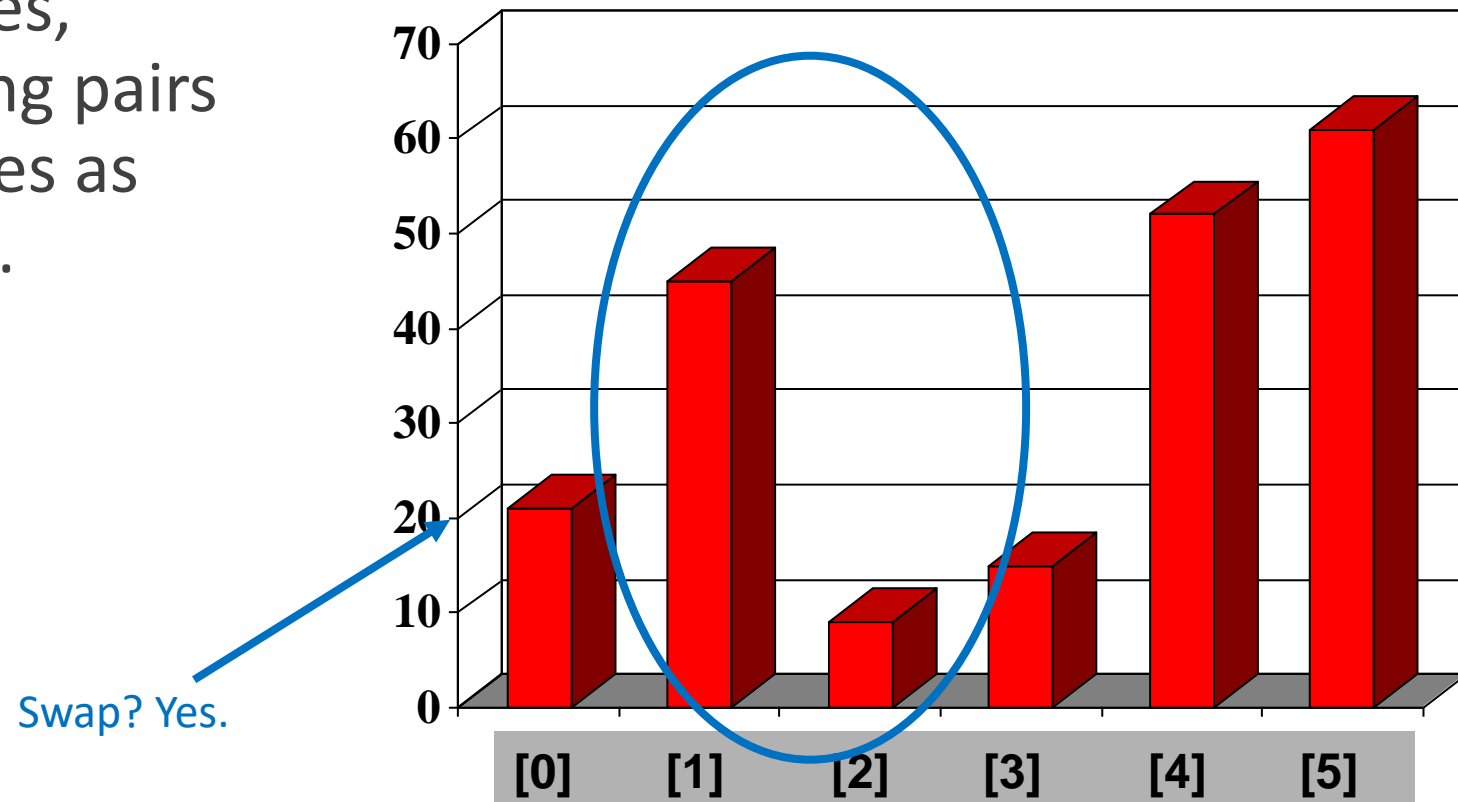
The Bubble Sort Algorithm

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



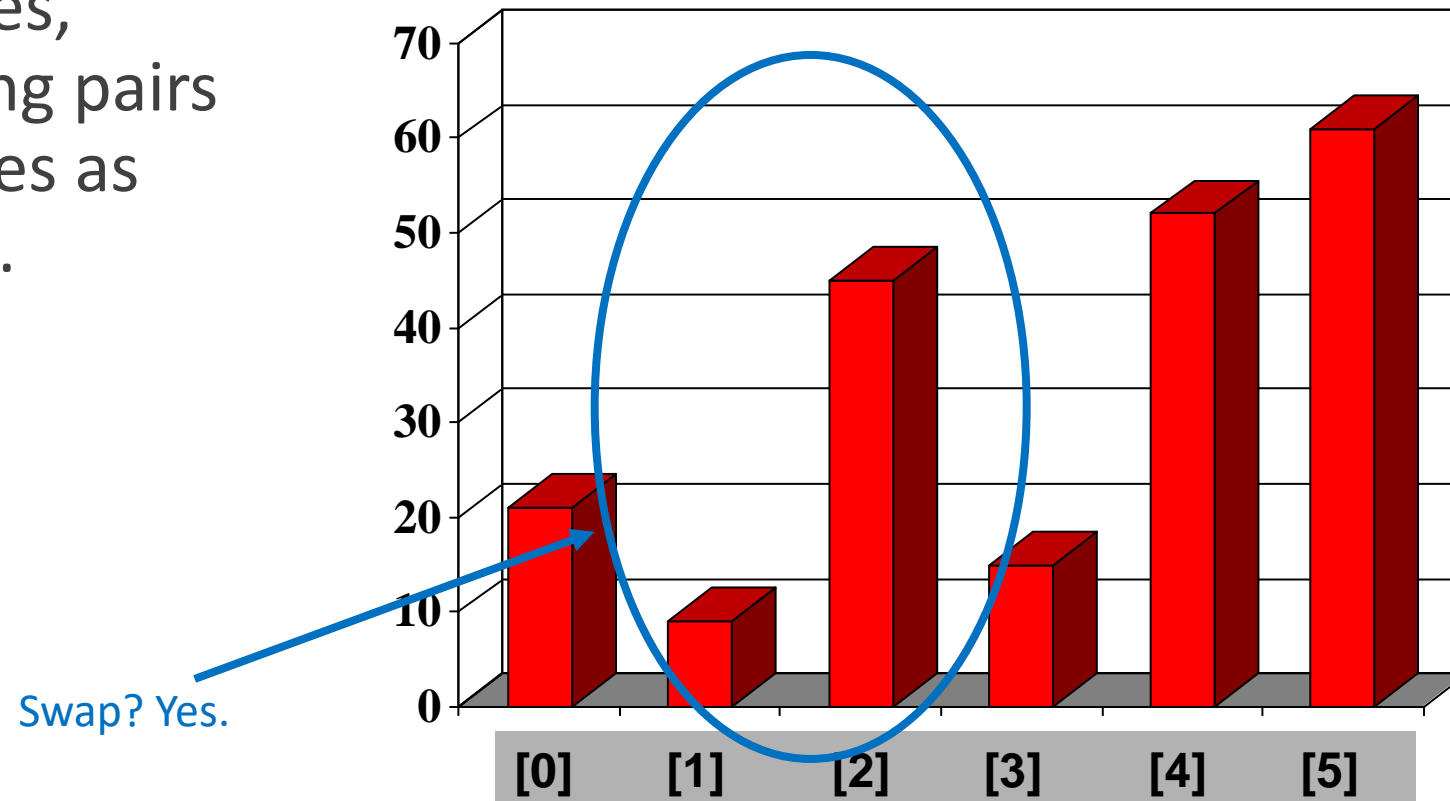
The Bubble Sort Algorithm

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



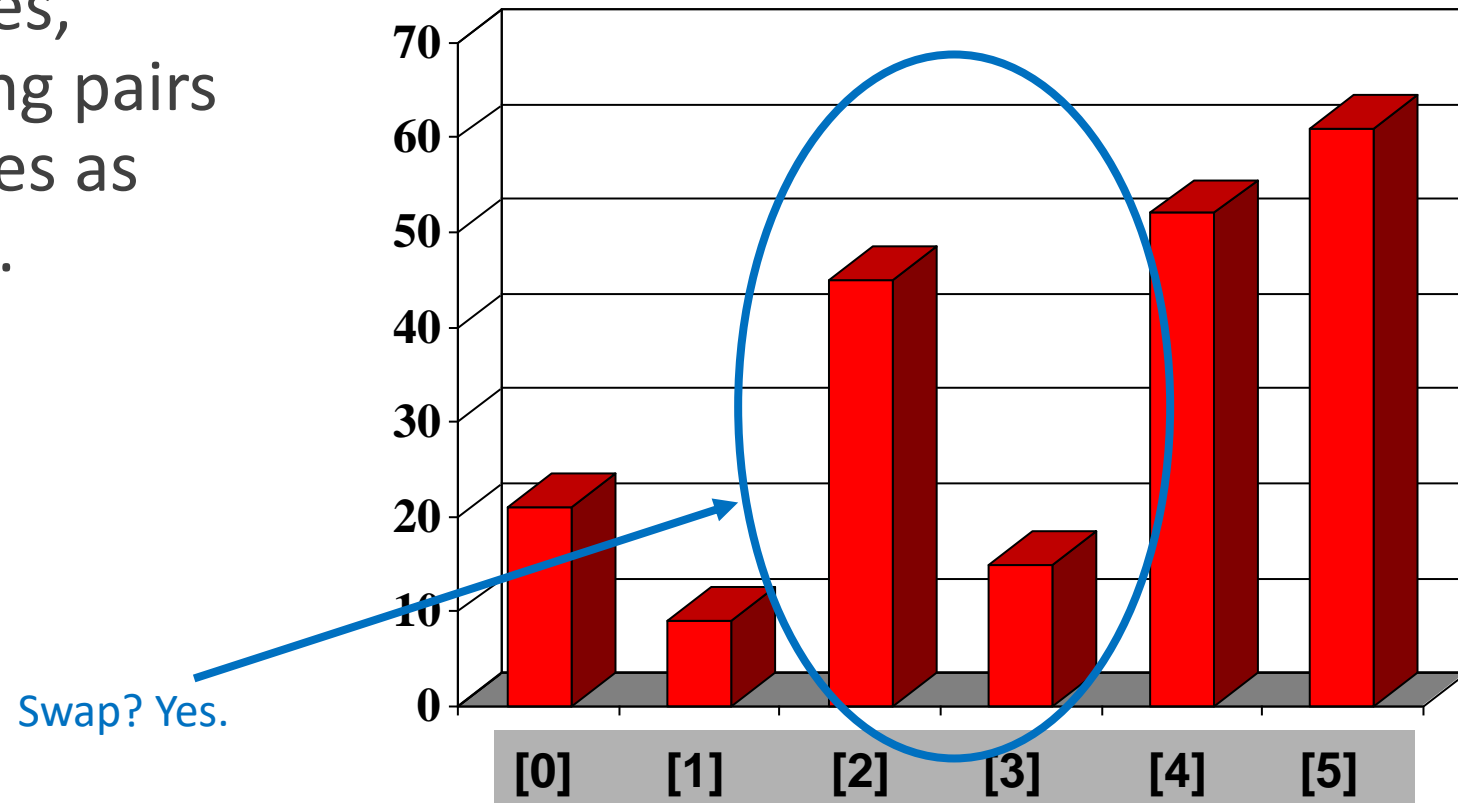
The Bubble Sort Algorithm

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



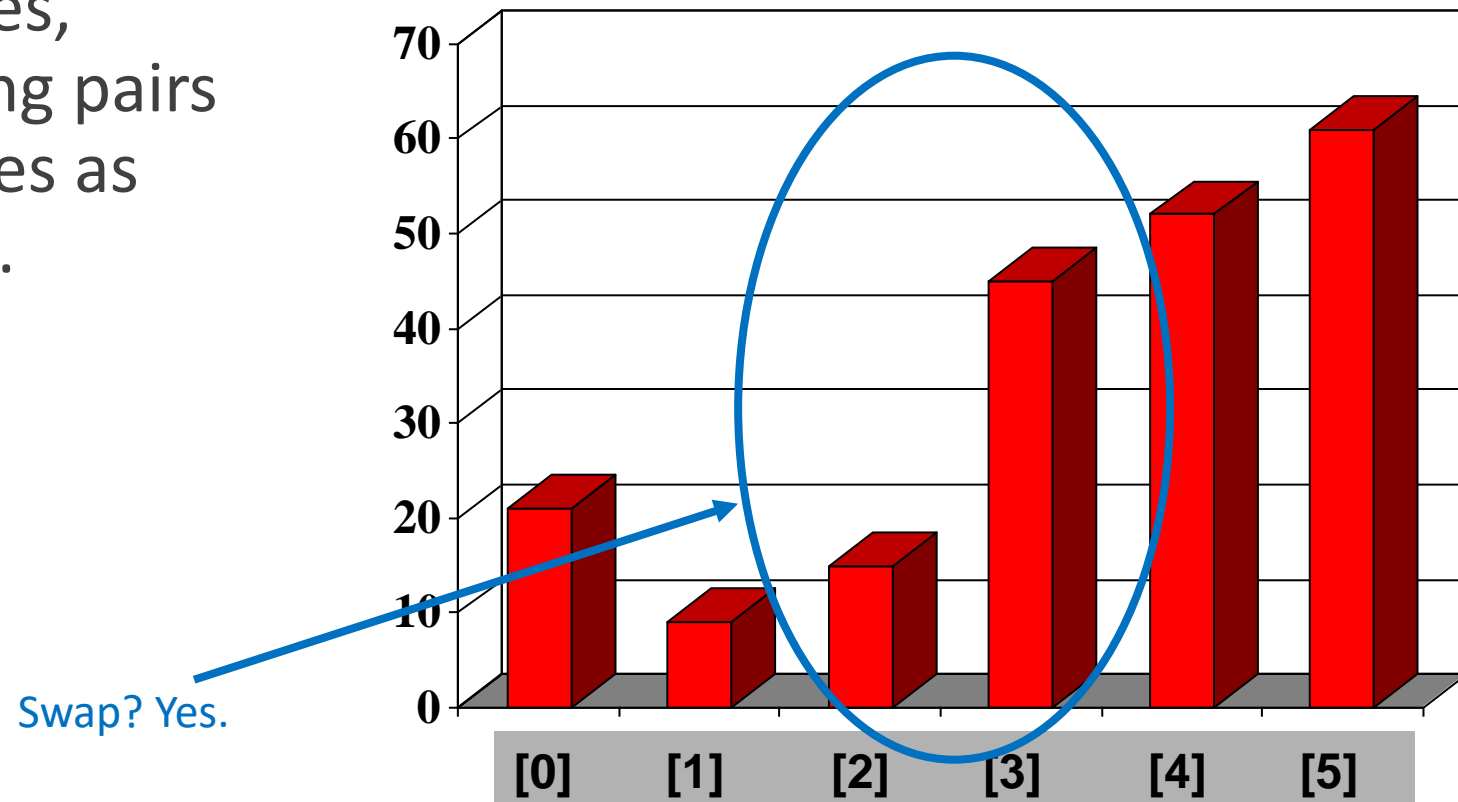
The Bubble Sort Algorithm

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



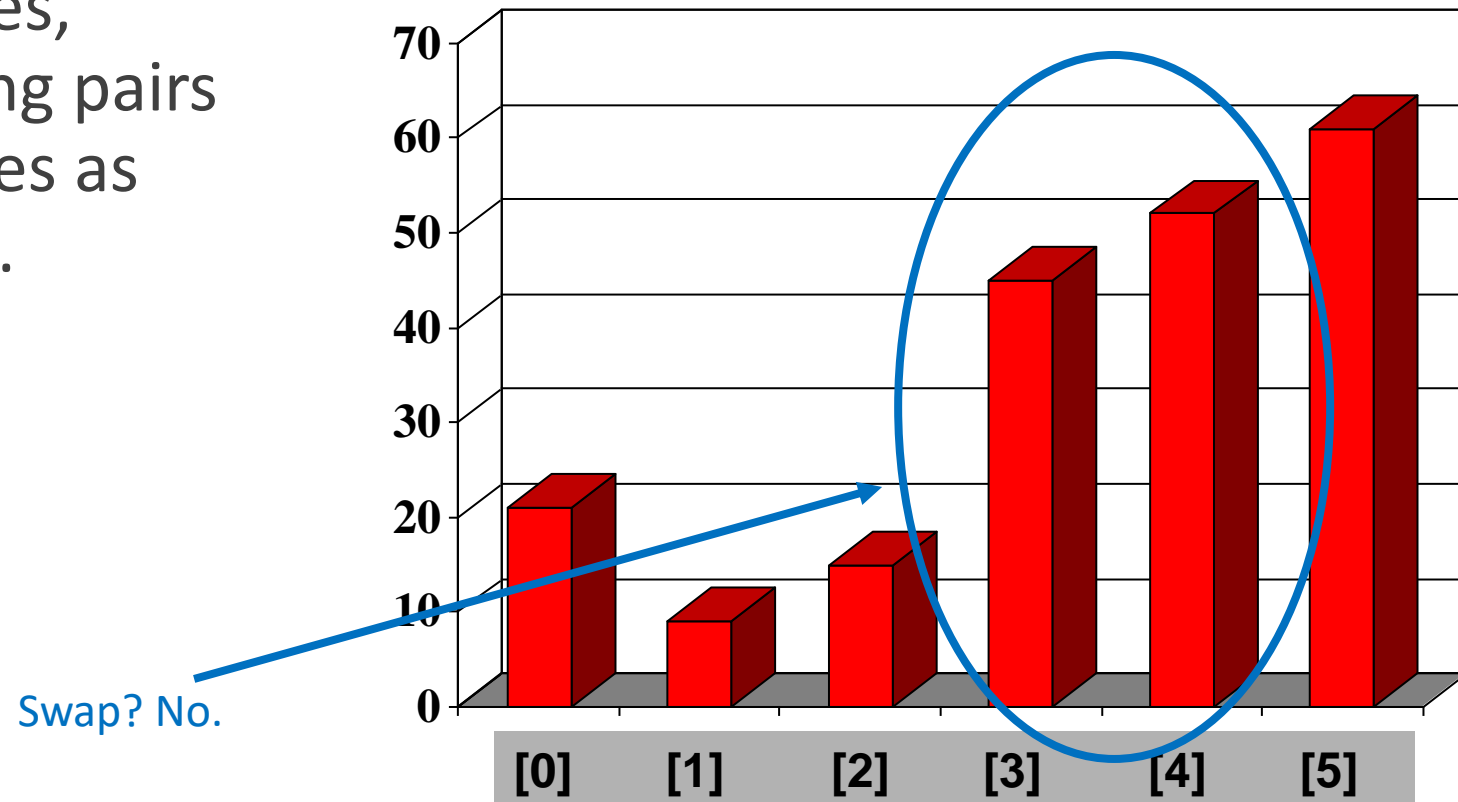
The Bubble Sort Algorithm

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



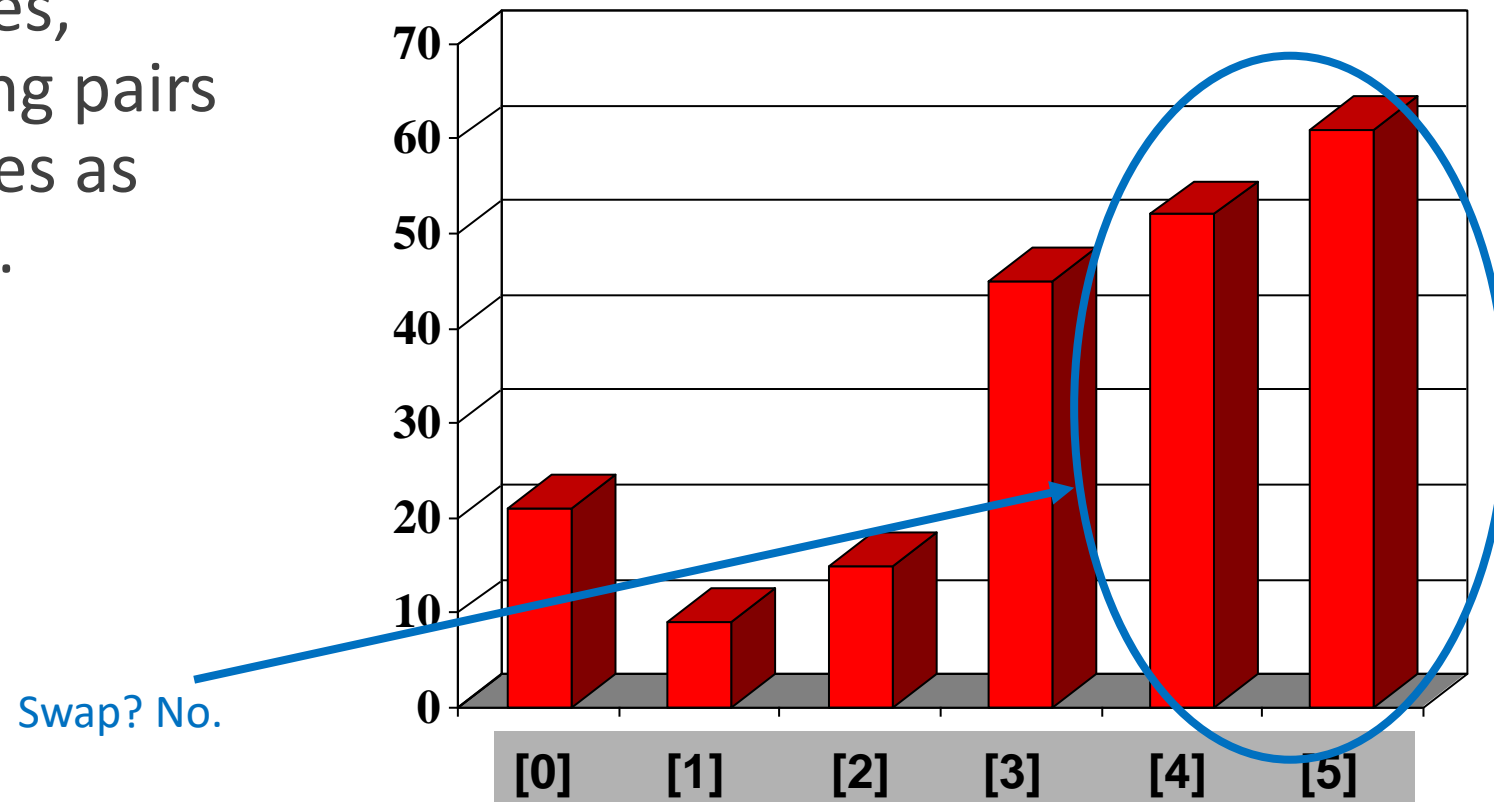
The Bubble Sort Algorithm

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



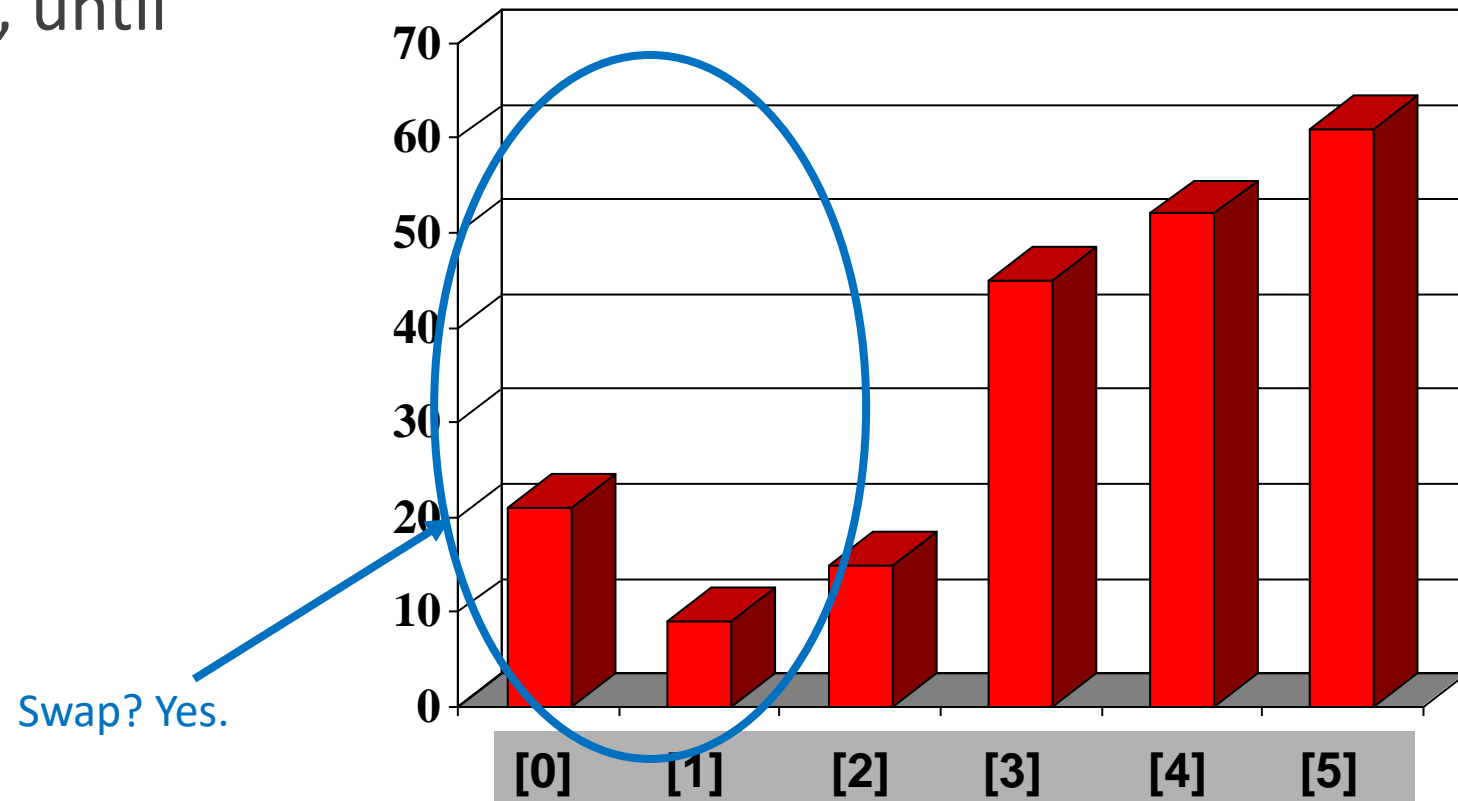
The Bubble Sort Algorithm

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:

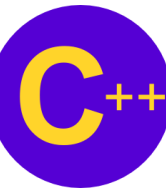
for $i = 0$ to $n-1$

 for $j = 0$ to $n-2$

 if $\text{key}[j] > \text{key}[j+1]$ 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 $\text{key}[j] > \text{key}[j+1]$ then swap

if no elements swapped in this pass through array, done.

otherwise, continue



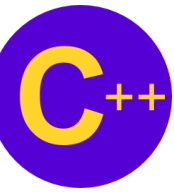
Timing and Other Issues

- 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 7

Merge Sort



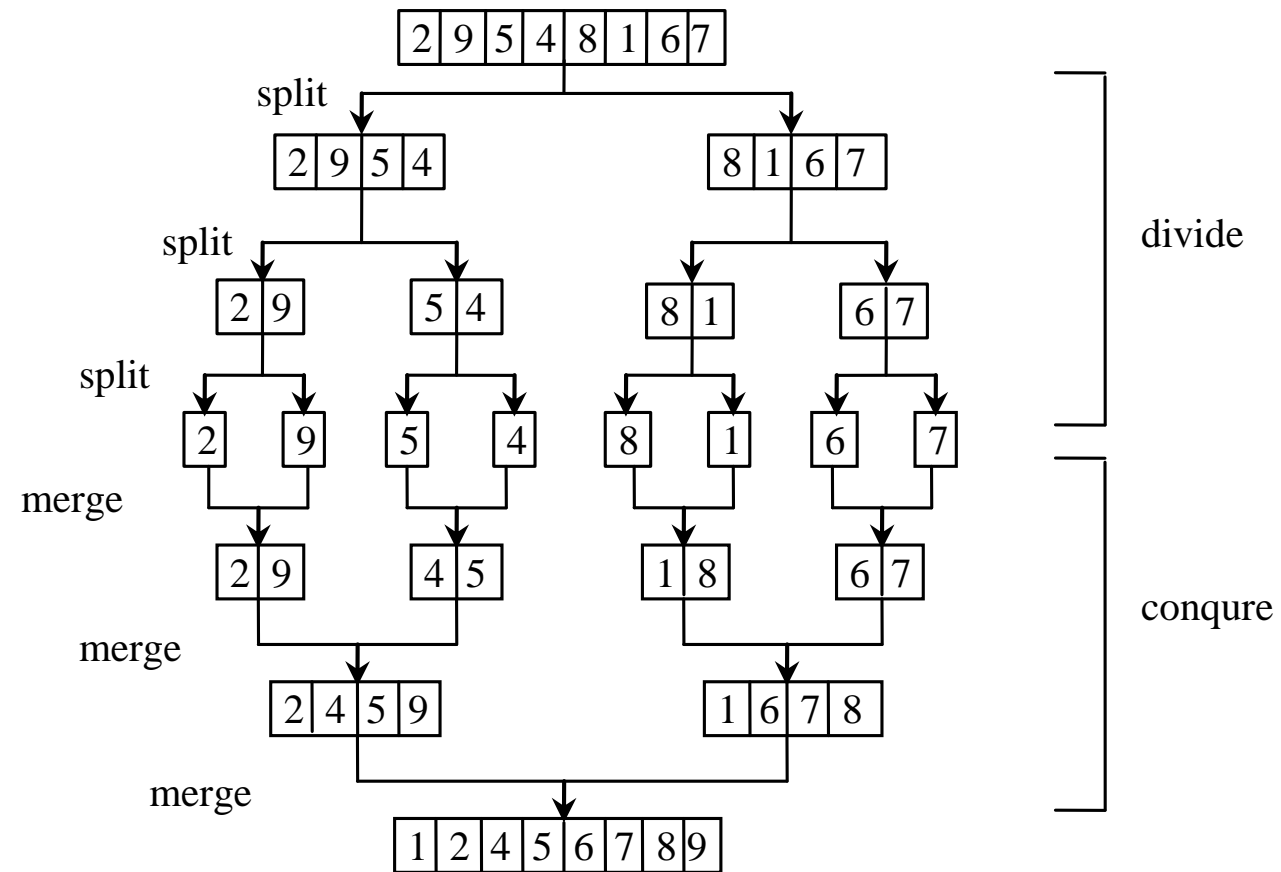
Merge Sort

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.

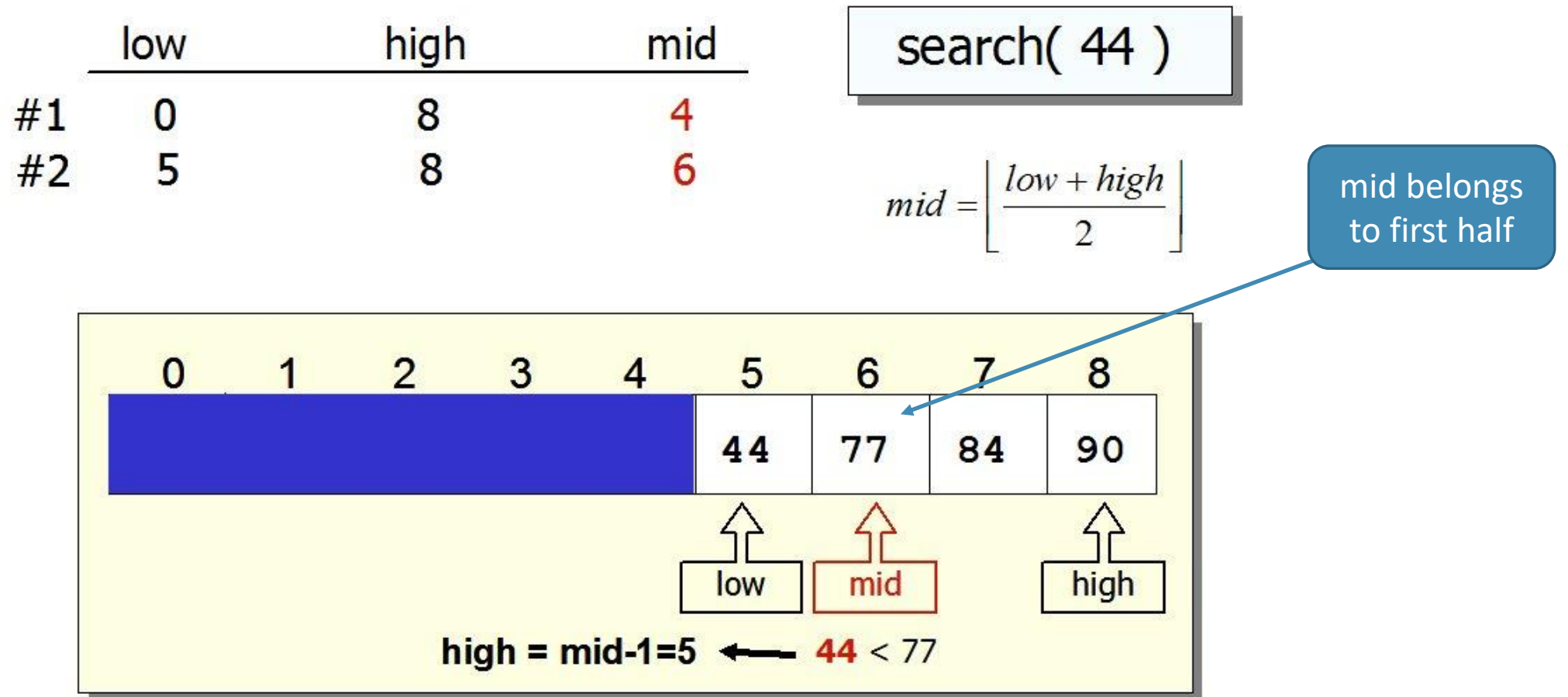
Merge Sort Algorithm

```
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



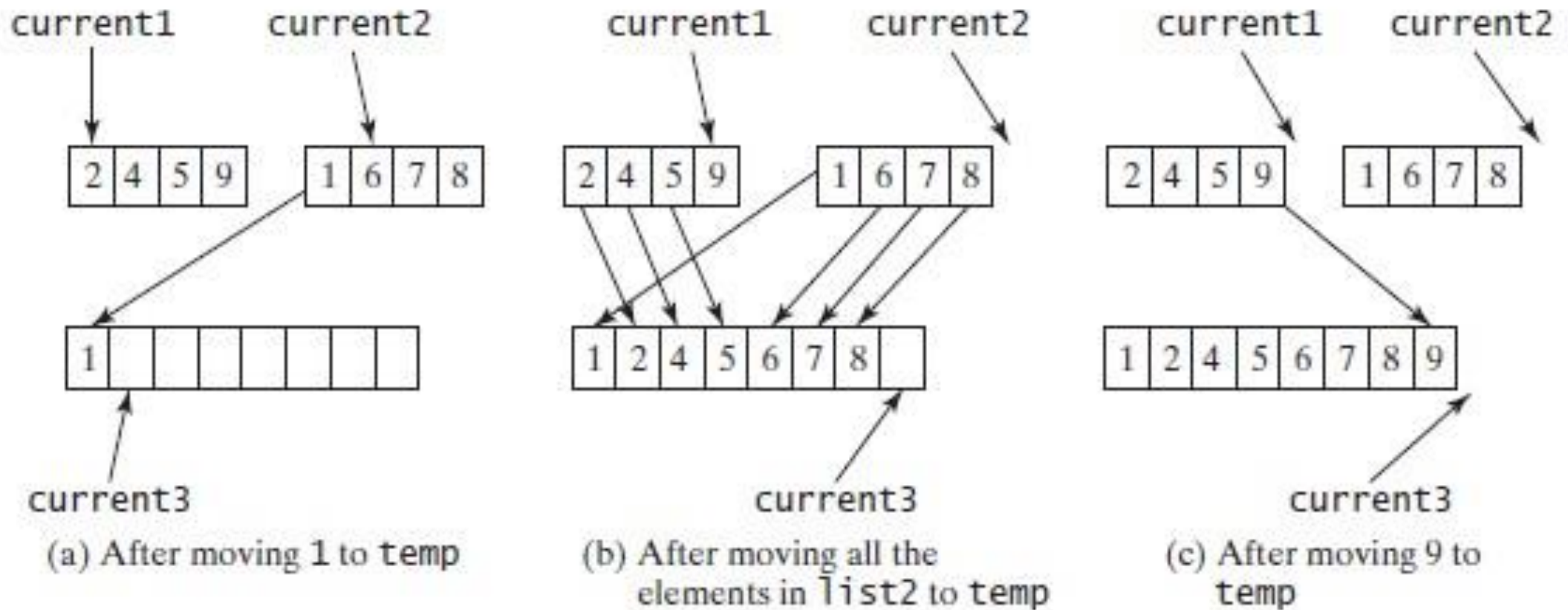
We use the same low, mid, high system as the Binary Search

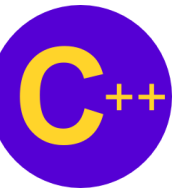


Top Level mergeSort Method

```
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

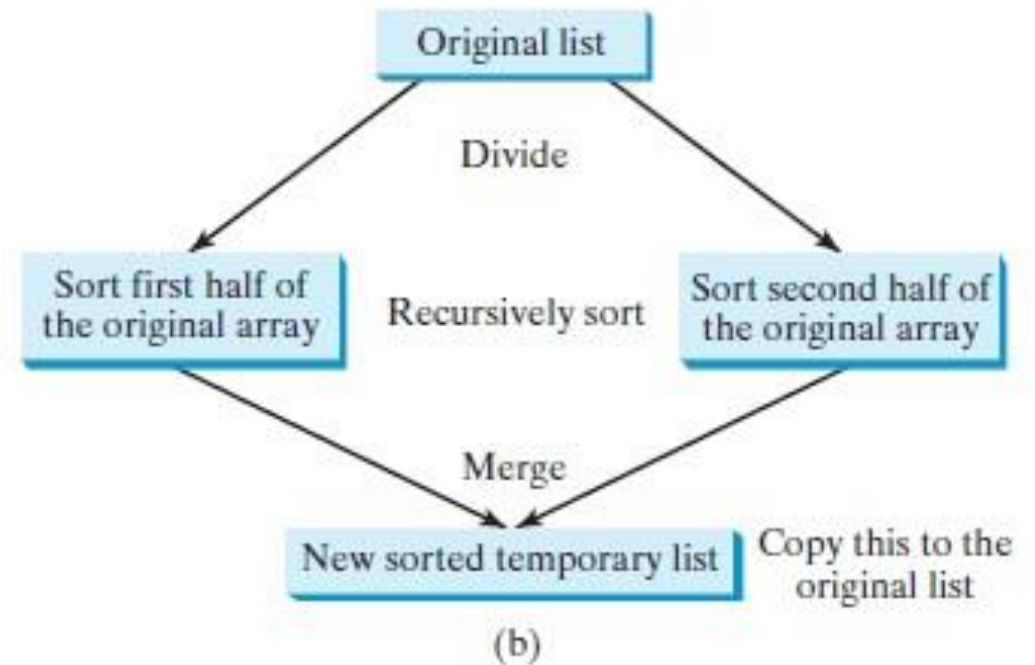
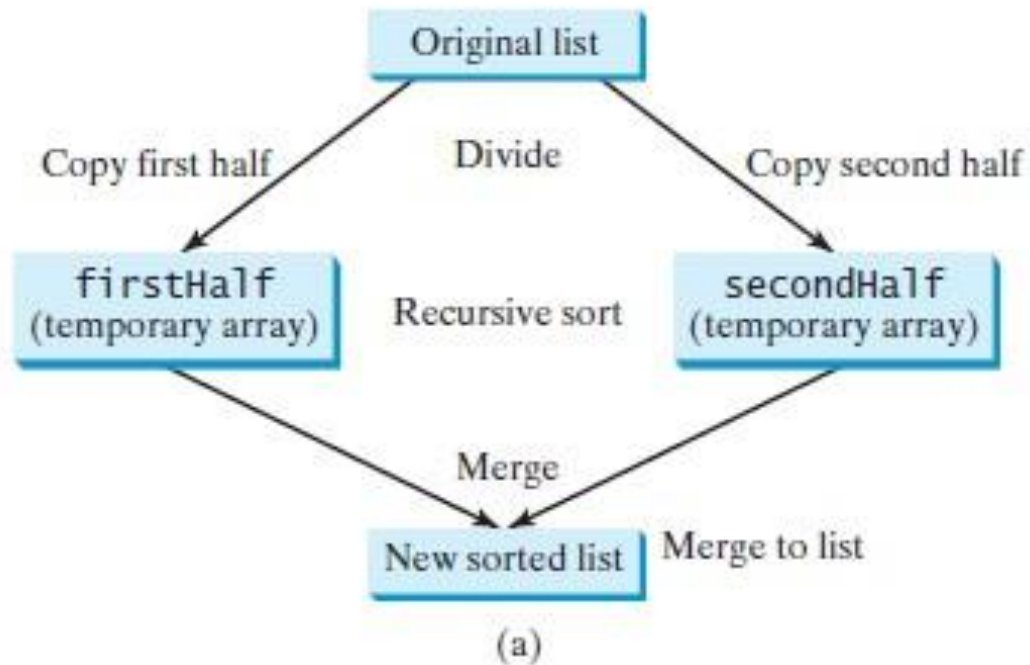




merge Method

```
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,

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

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,

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

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

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

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



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;
}
```