Sorting I

Introduction

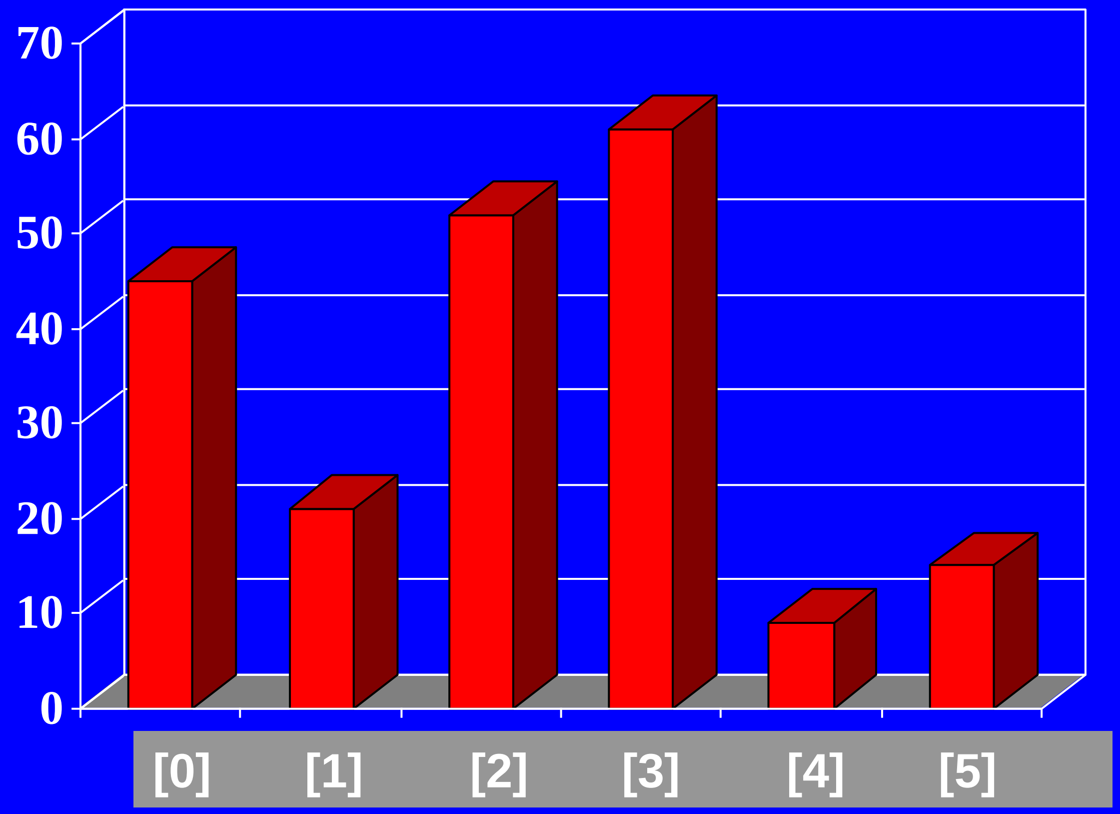
- Common problem: sort a list of values, starting from lowest to highest.
  - List of exam scores
  - Words of dictionary in alphabetical order
  - Students names listed alphabetically
  - Student records sorted by ID#
- Generally, we are given a list of records that have *keys*. These keys are used to define an ordering of the items in the list.

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

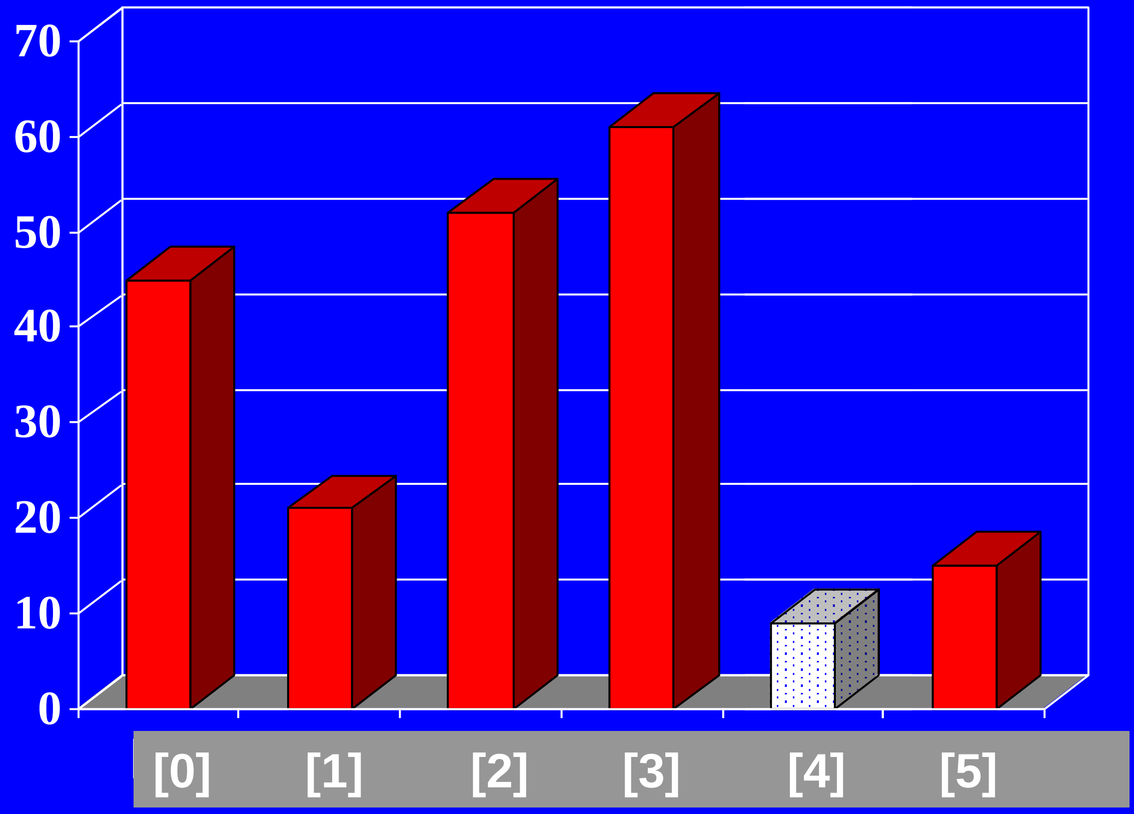
Sorting an Array of Integers

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



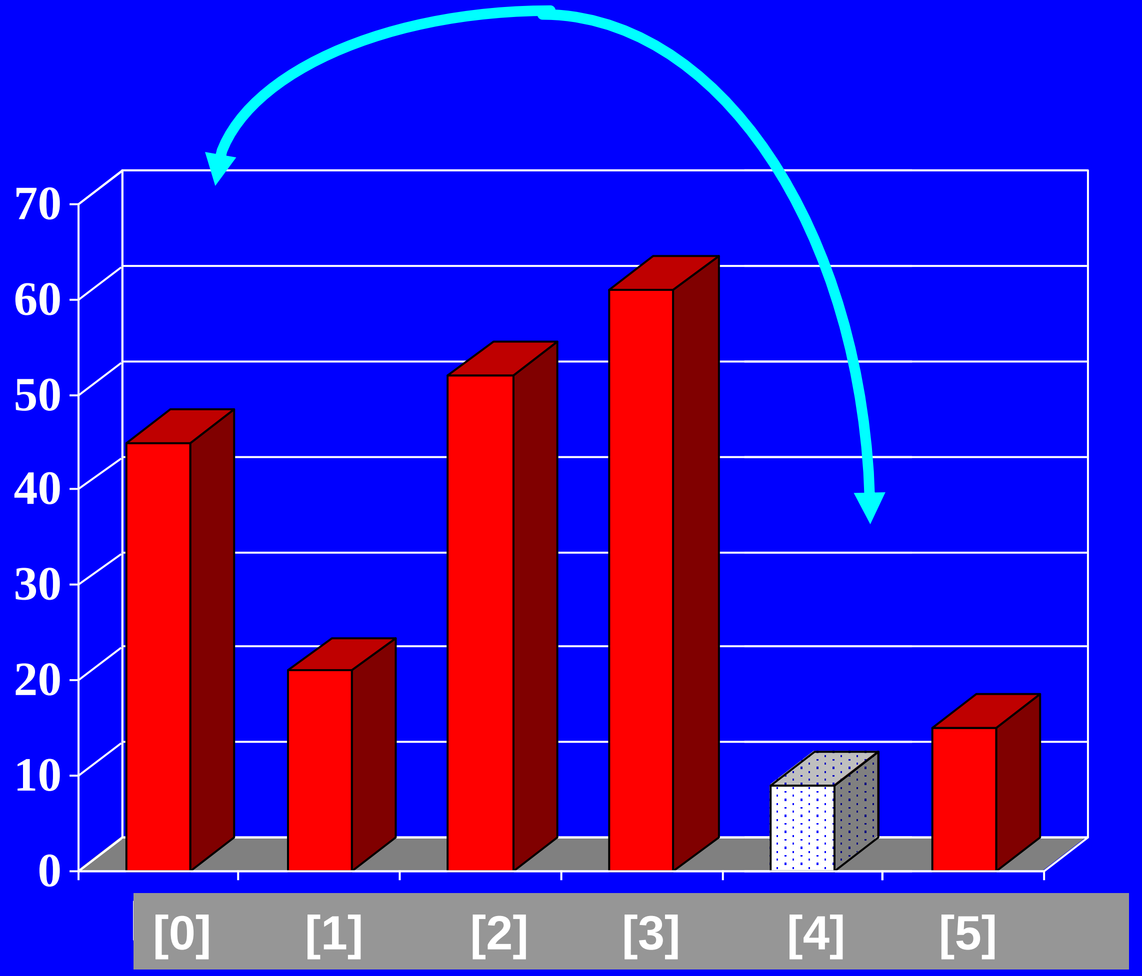
The Selection Sort Algorithm

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



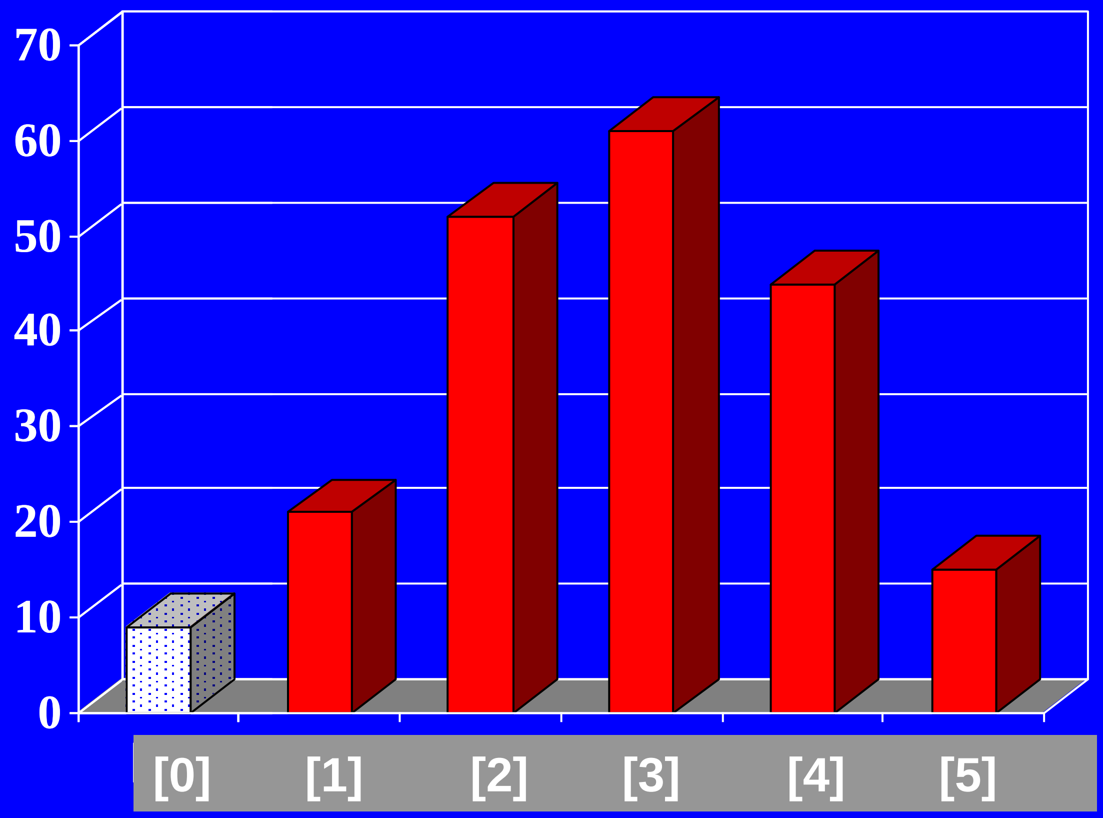
The Selection Sort Algorithm

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



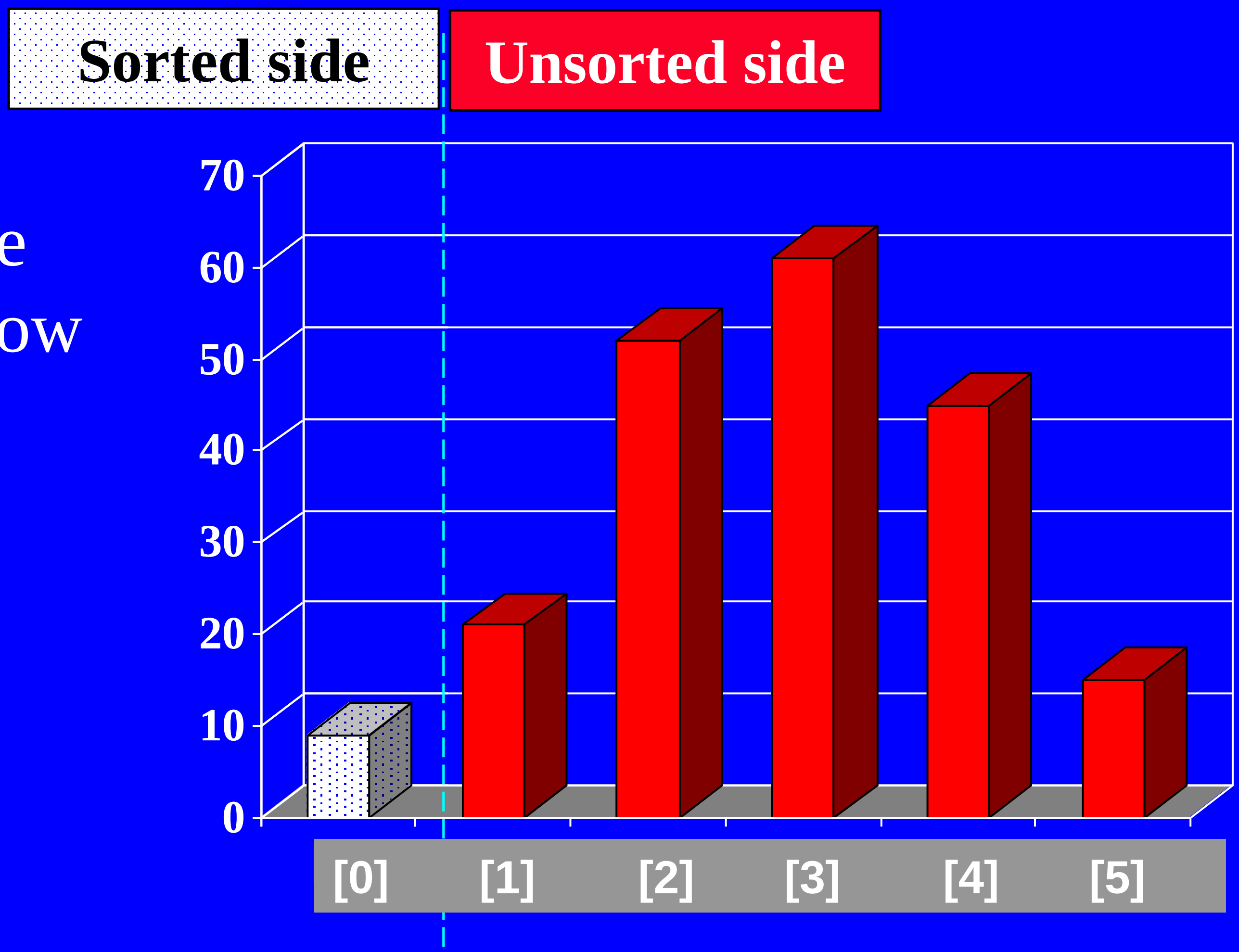
The Selection Sort Algorithm

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



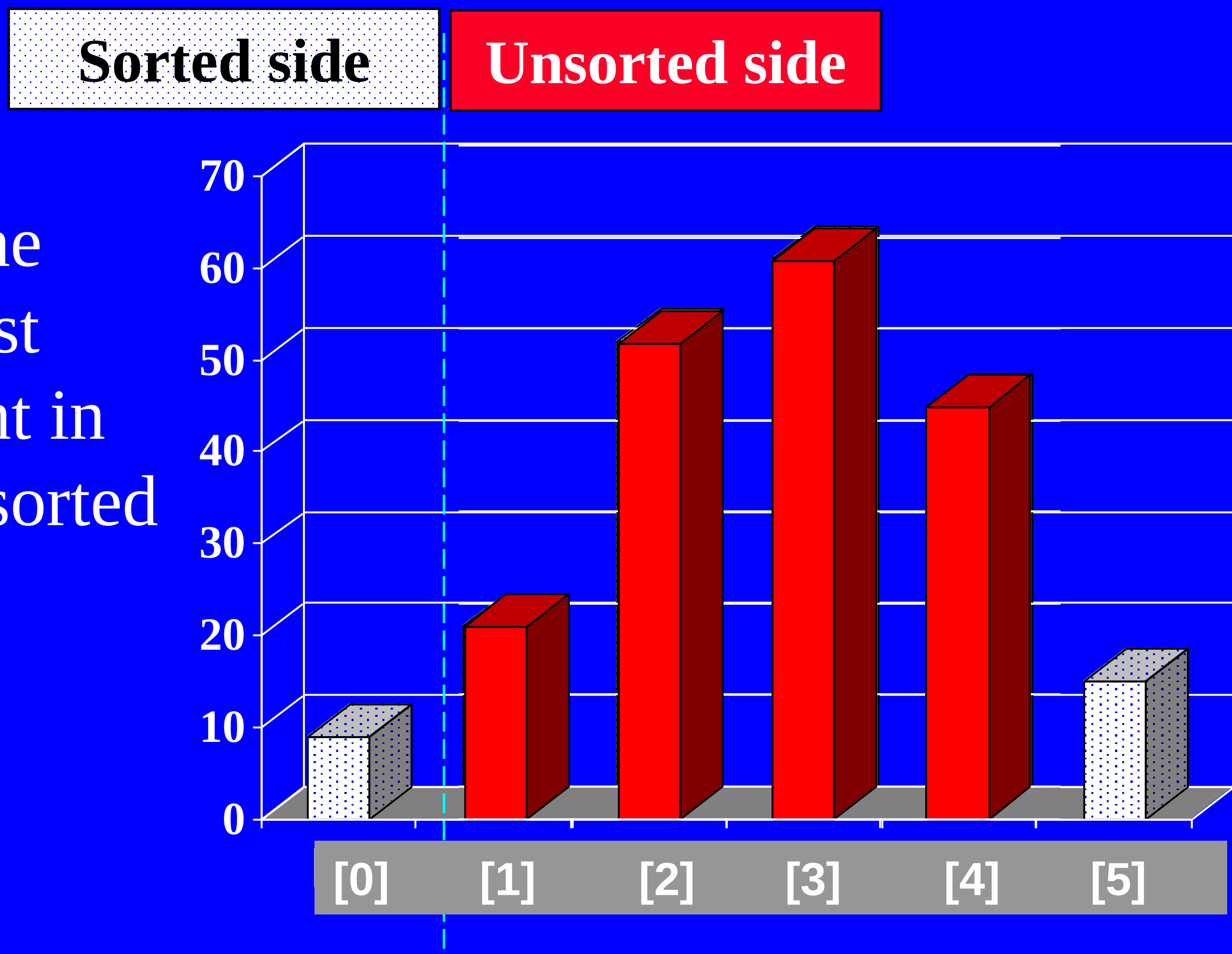
The Selection Sort Algorithm

- Part of the array is now sorted.



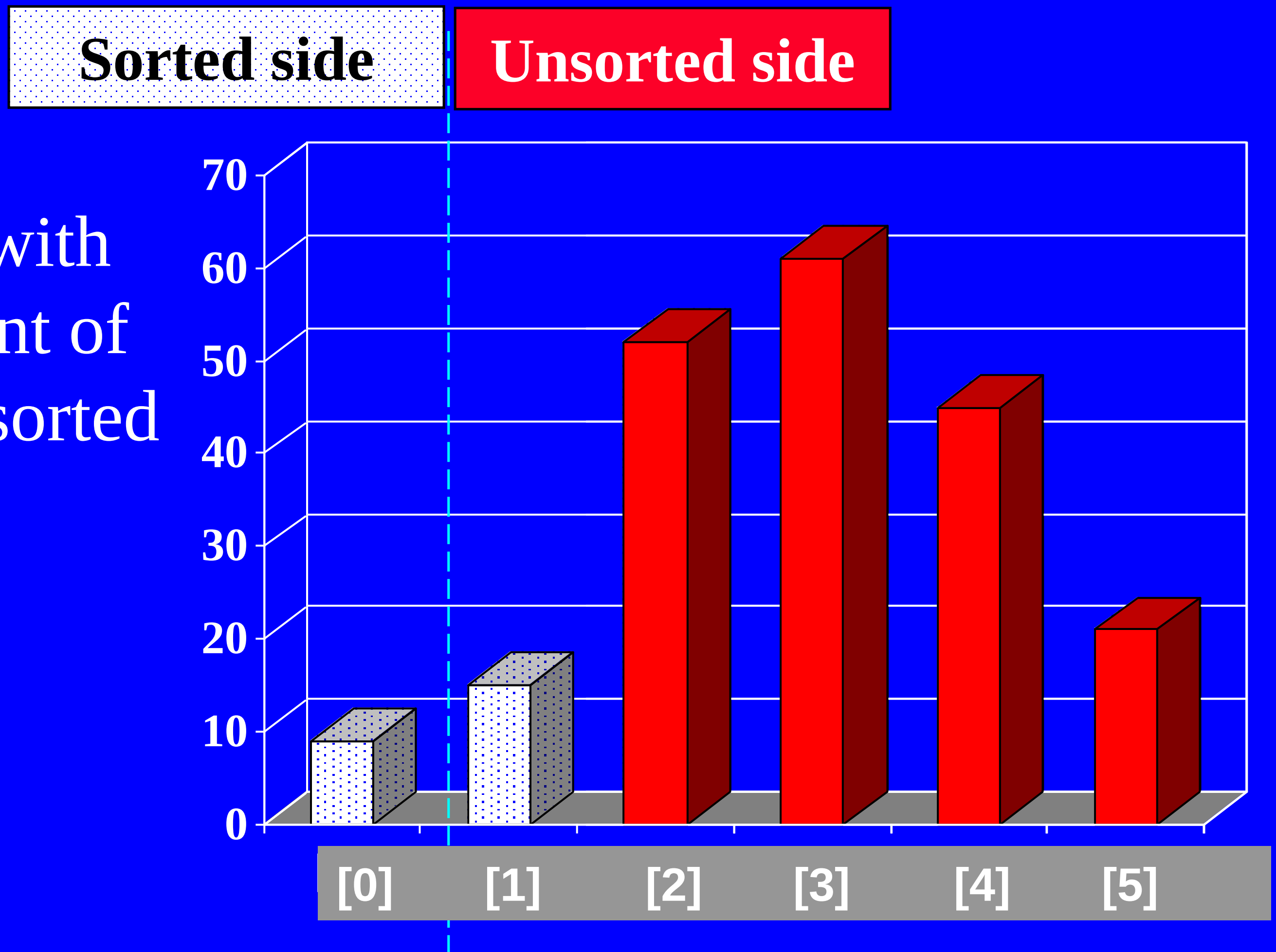
The Selection Sort Algorithm

- Find the smallest element in the unsorted side.



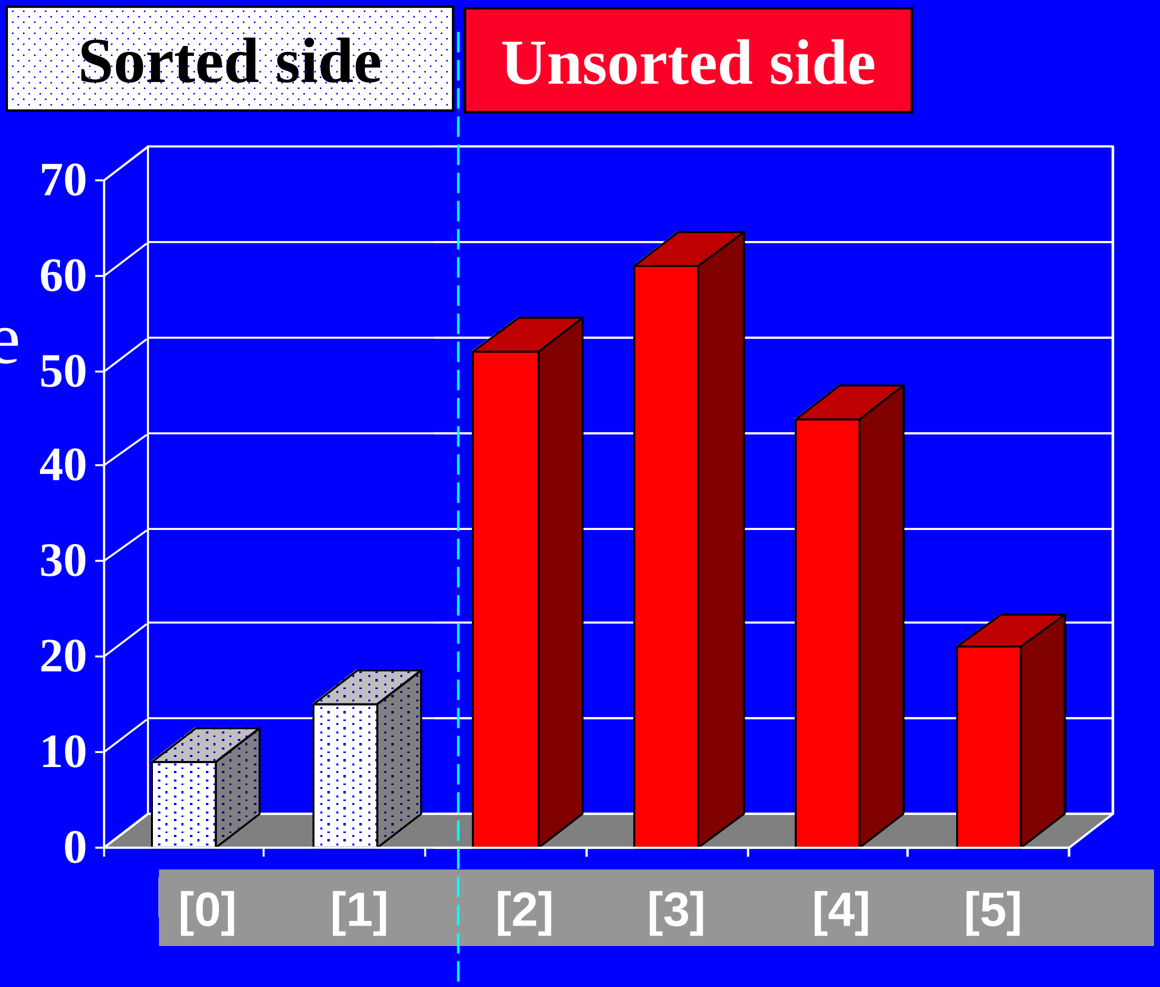
The Selection Sort Algorithm

- Swap with the front of the unsorted side.



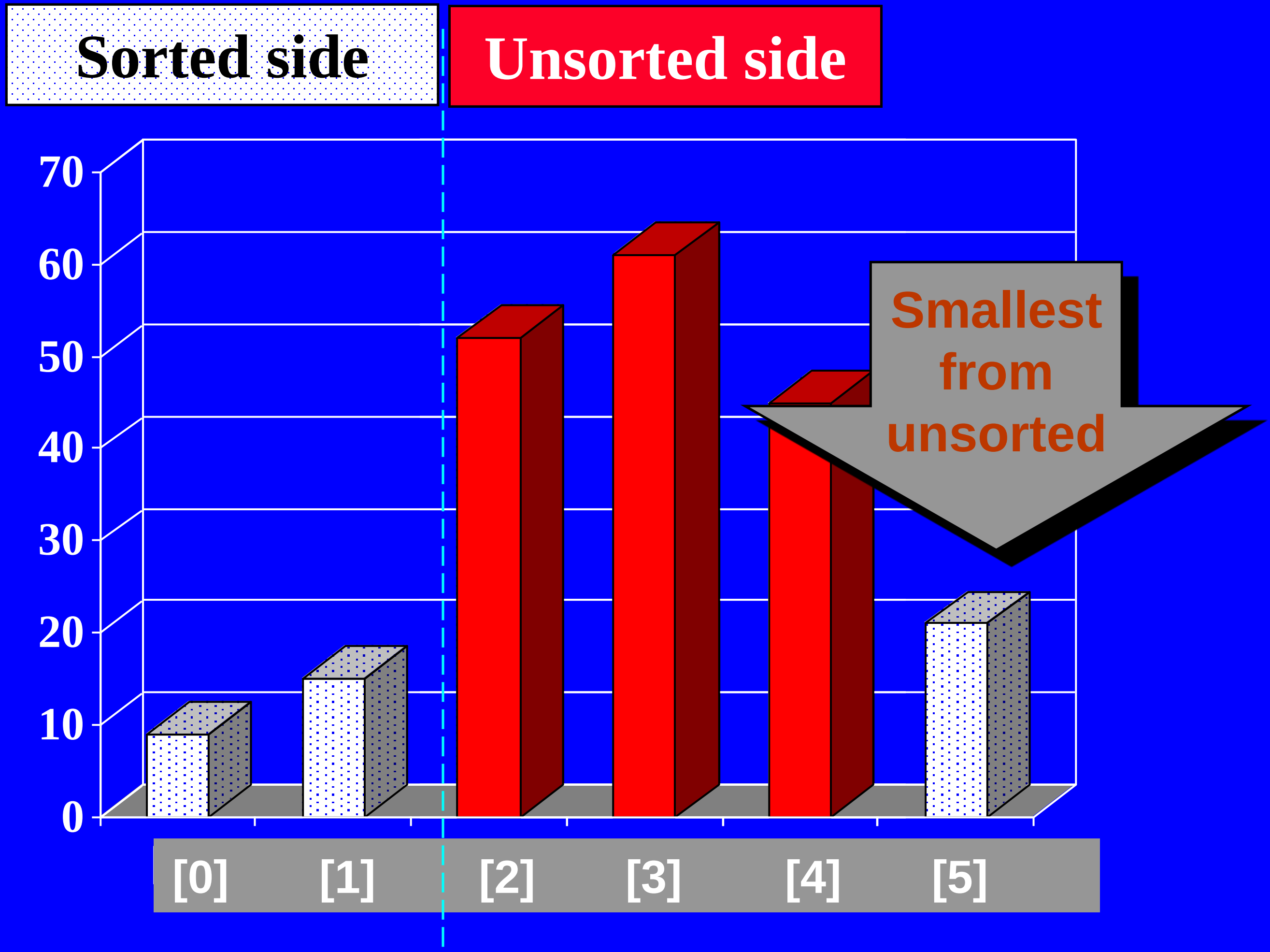
The Selection Sort Algorithm

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



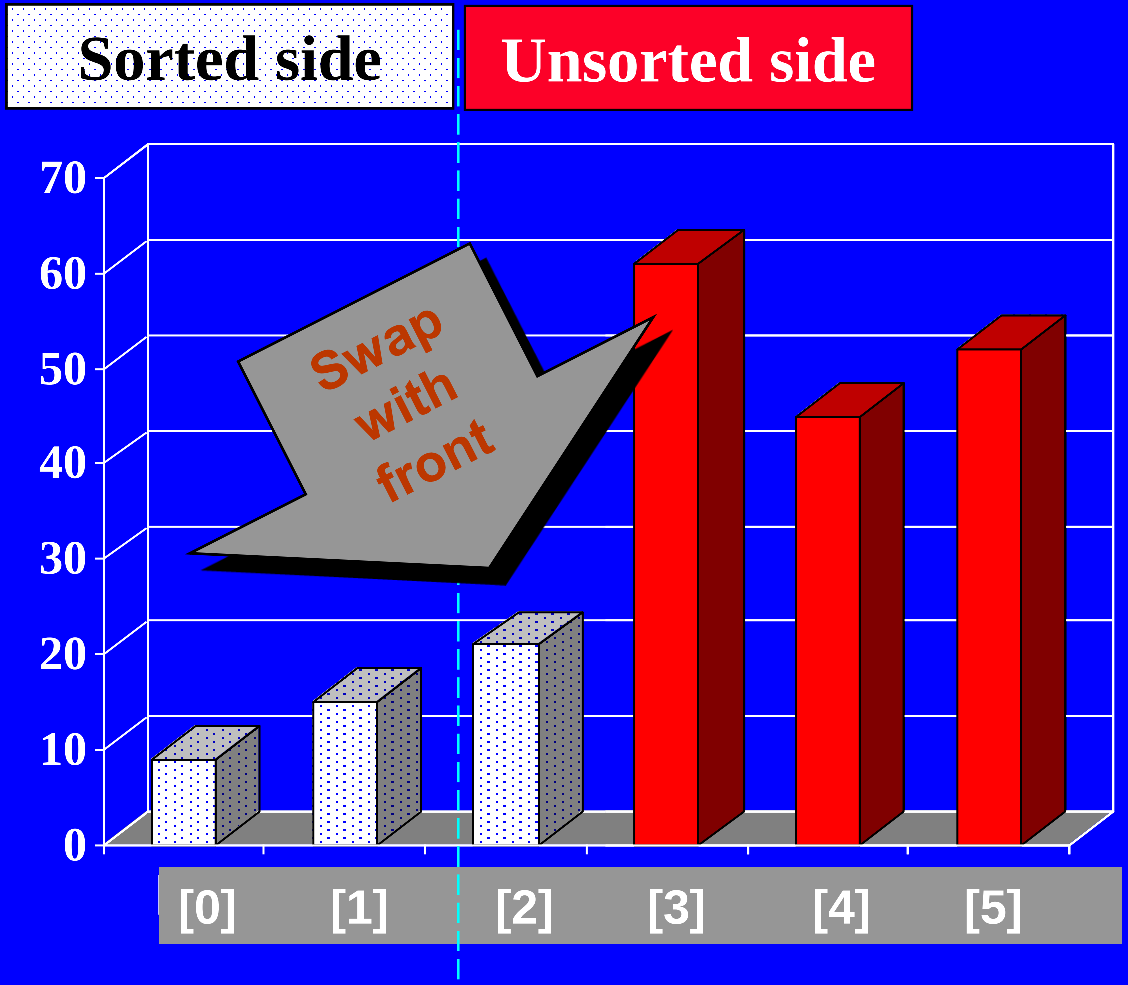
The Selection Sort Algorithm

- The process continues...

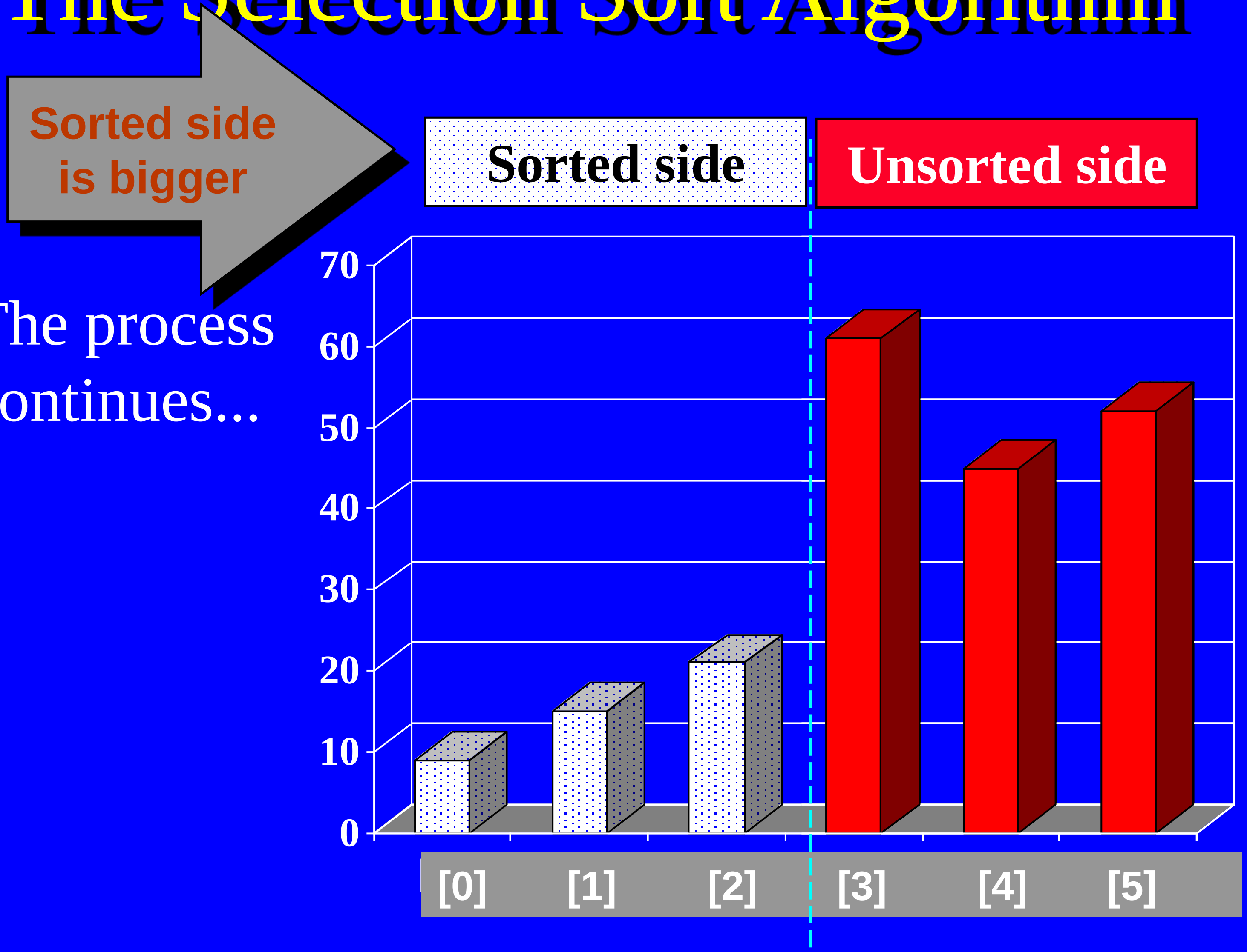


The Selection Sort Algorithm

- The process continues...



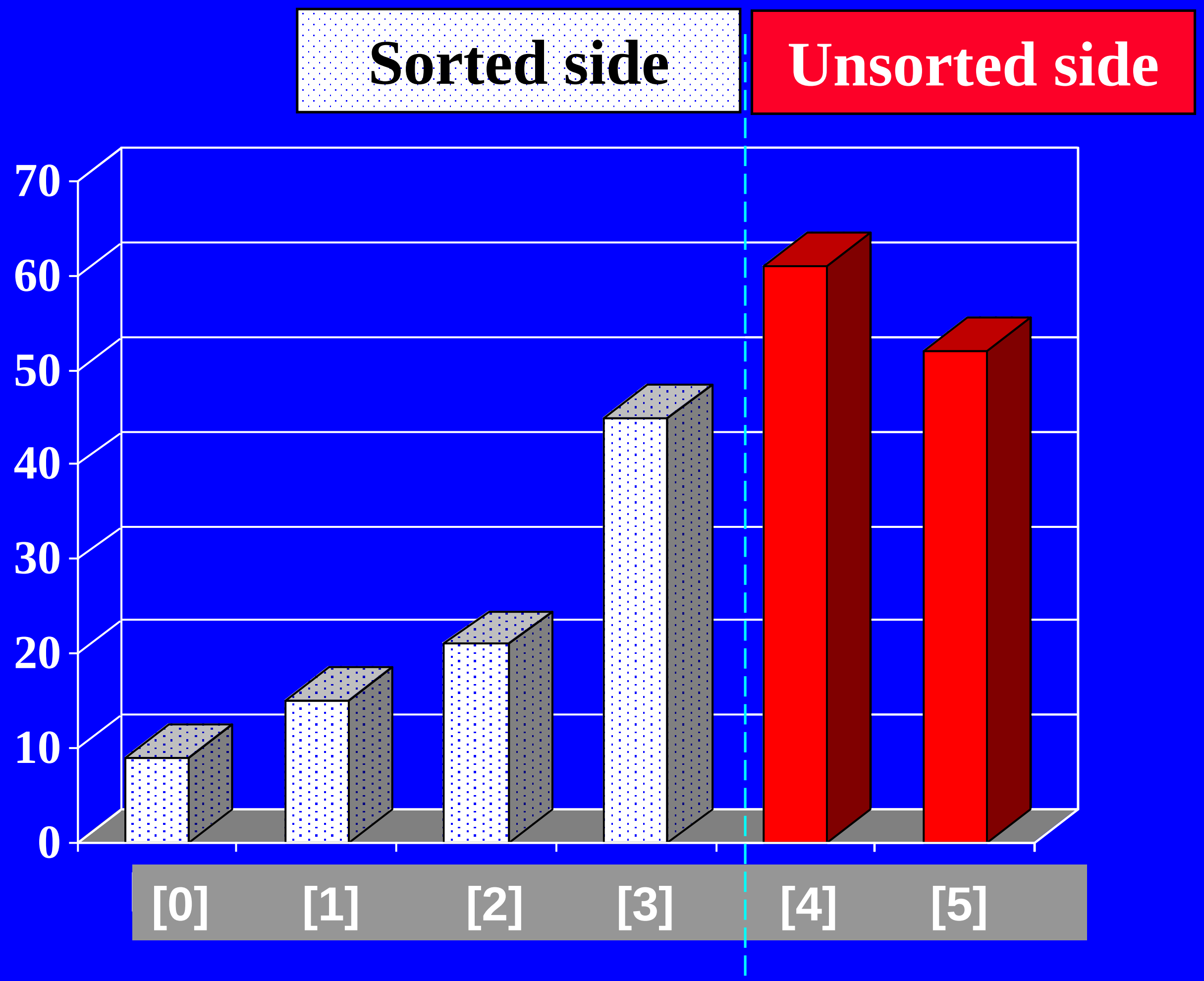
The Selection Sort Algorithm



- The process continues...

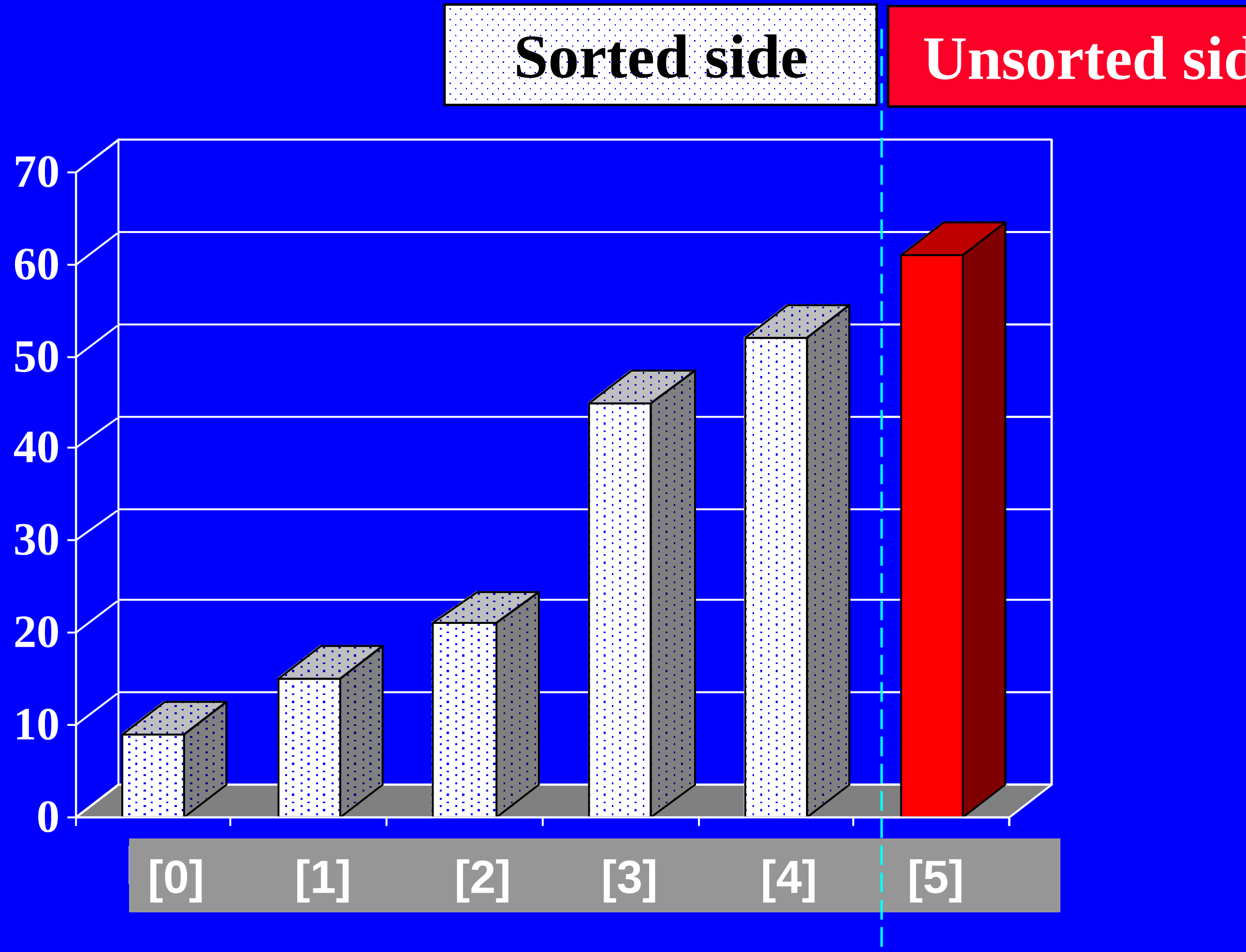
The Selection Sort Algorithm

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



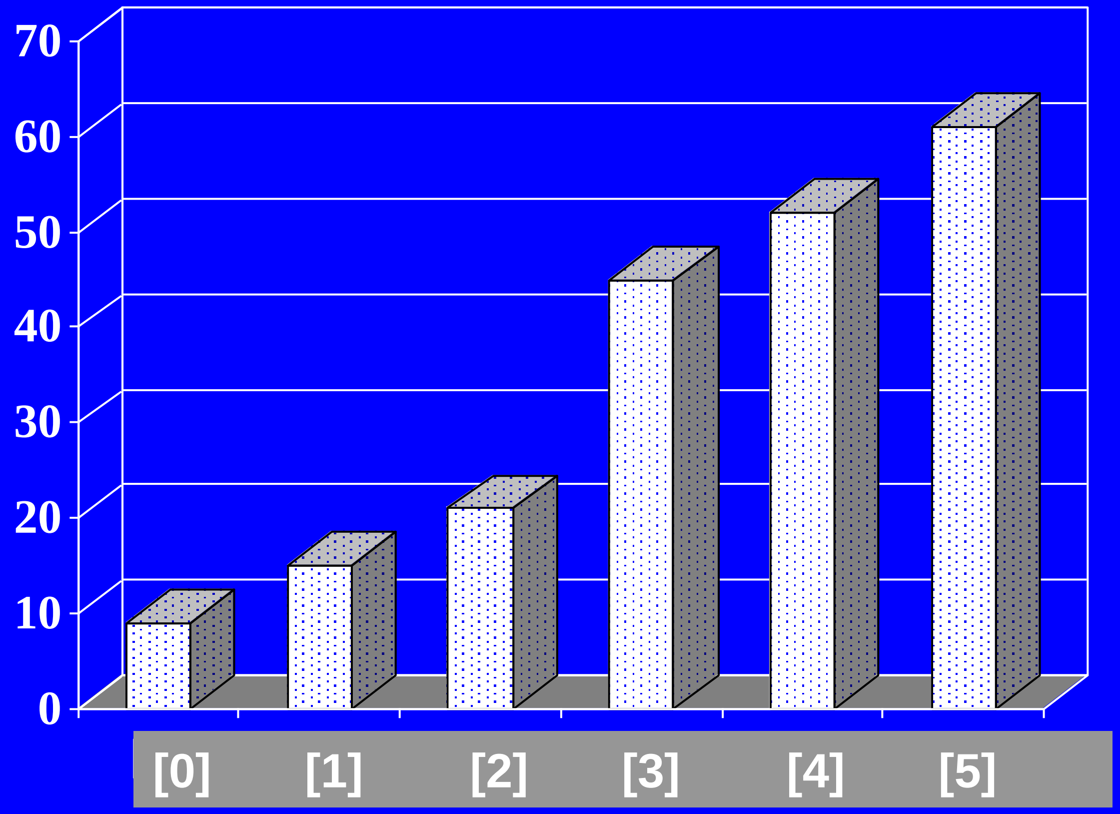
The Selection Sort Algorithm

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



The Selection Sort Algorithm

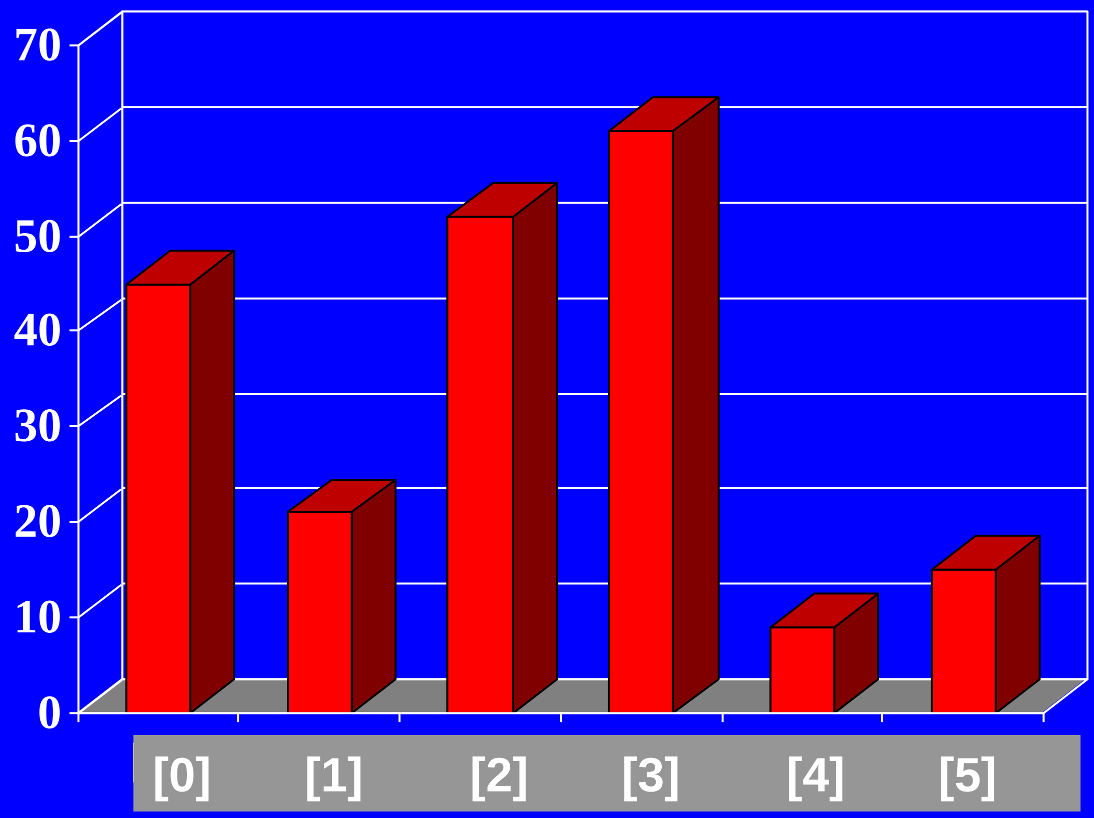
- The array is now sorted.
- We repeatedly **selected** the smallest element, and moved this element to the front of the unsorted side.



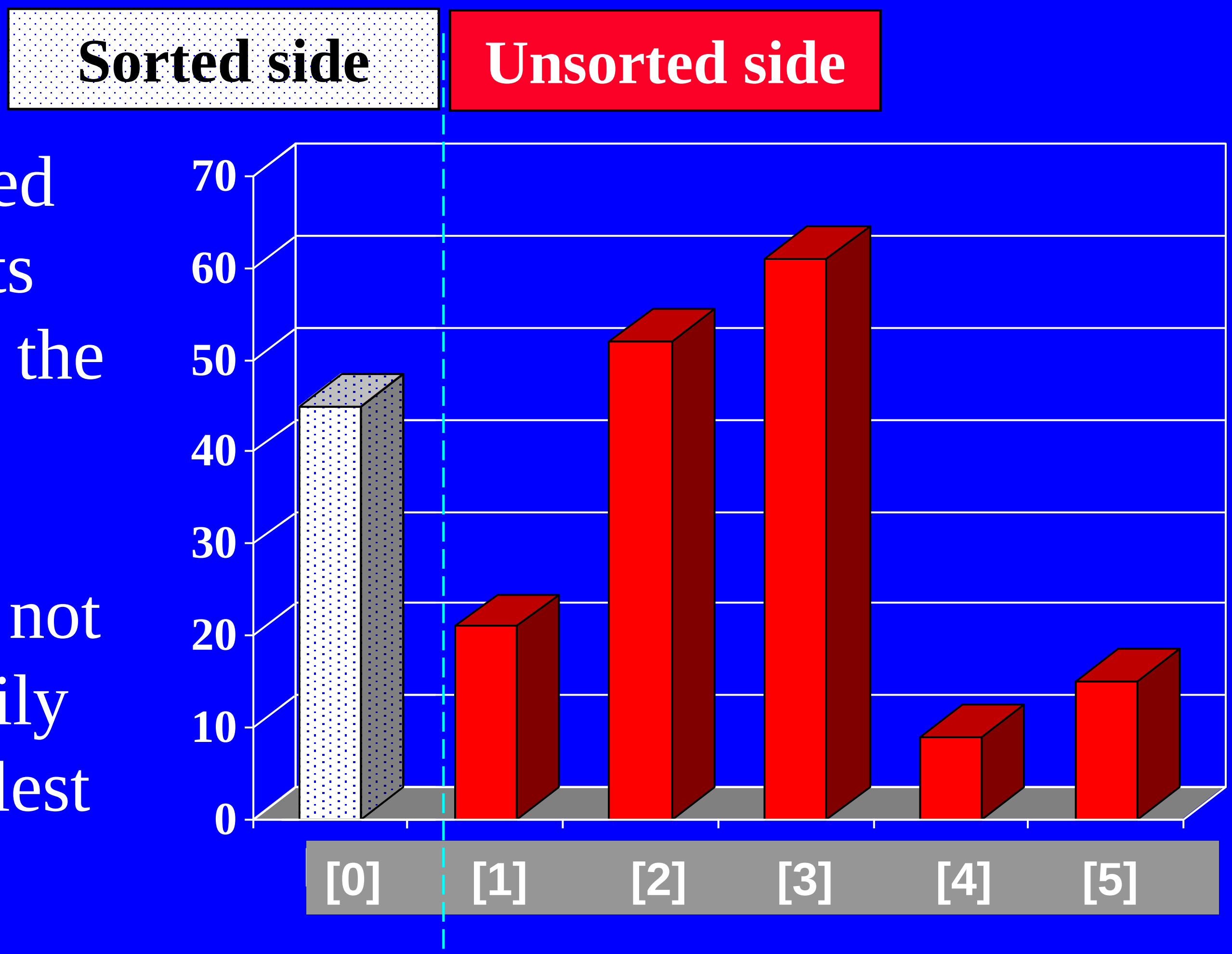
```java
public static void selectionsort(int a[],int n)
{

    for(int i=0;i<n-1;i++)
    {
        int min=i;
        for(int j=i+1;j<n;j++)
        {
            if(a[j]<a[min])
                min=j;
        }
        int t=a[i];
        a[i]=a[min];
        a[min]=t;

    }
```

The Insertion Sort Algorithm

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



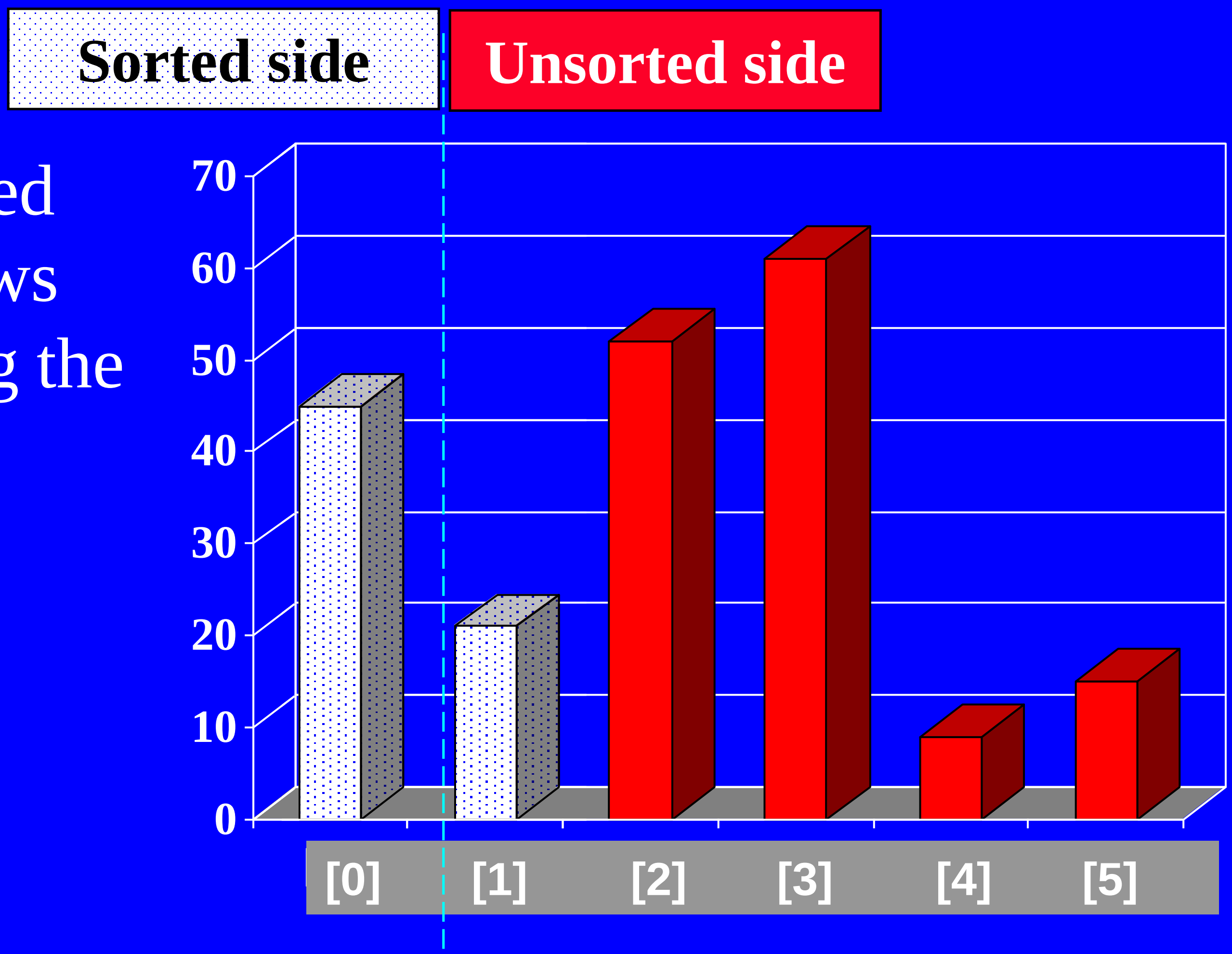
The Insertion Sort Algorithm



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

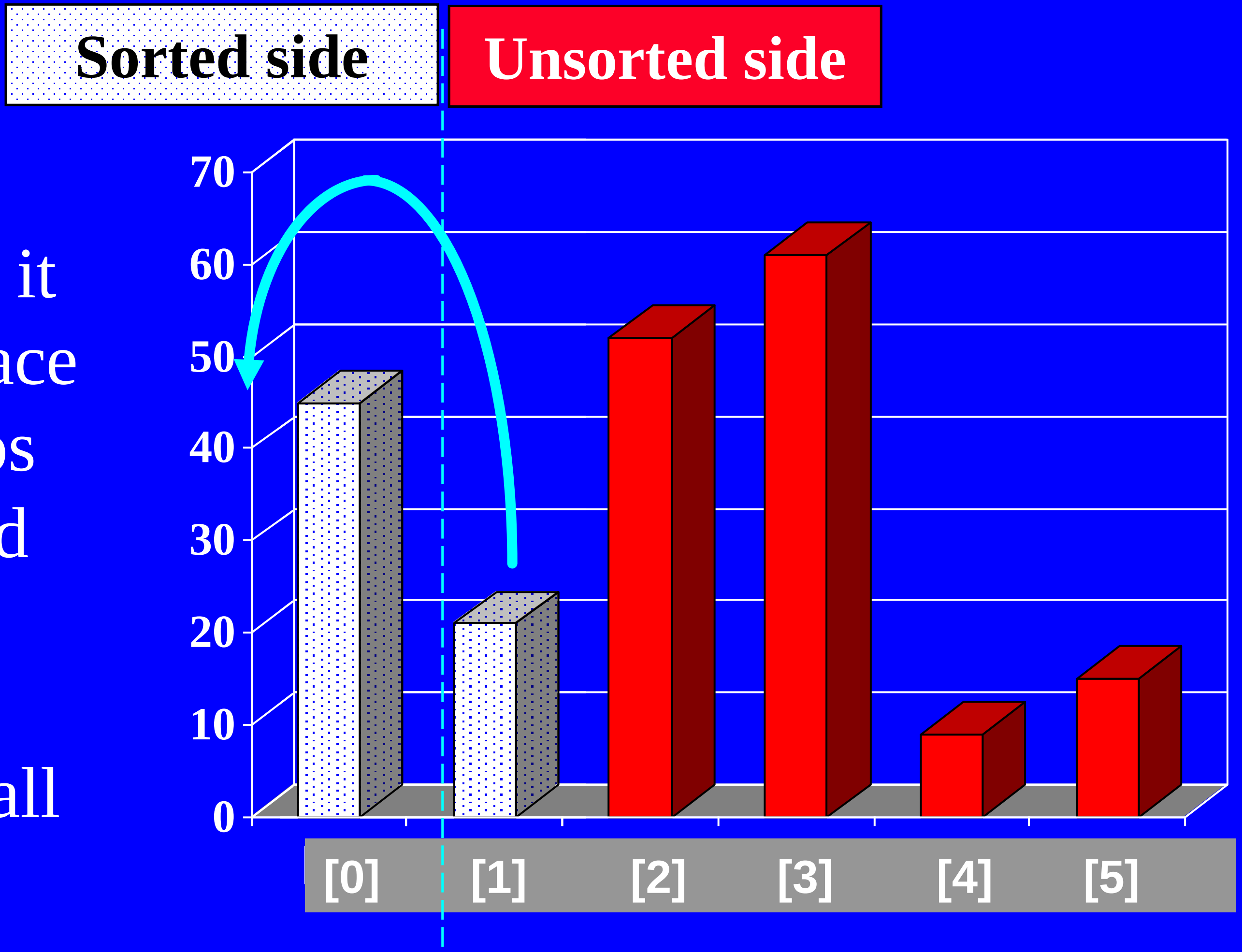
The Insertion Sort Algorithm

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

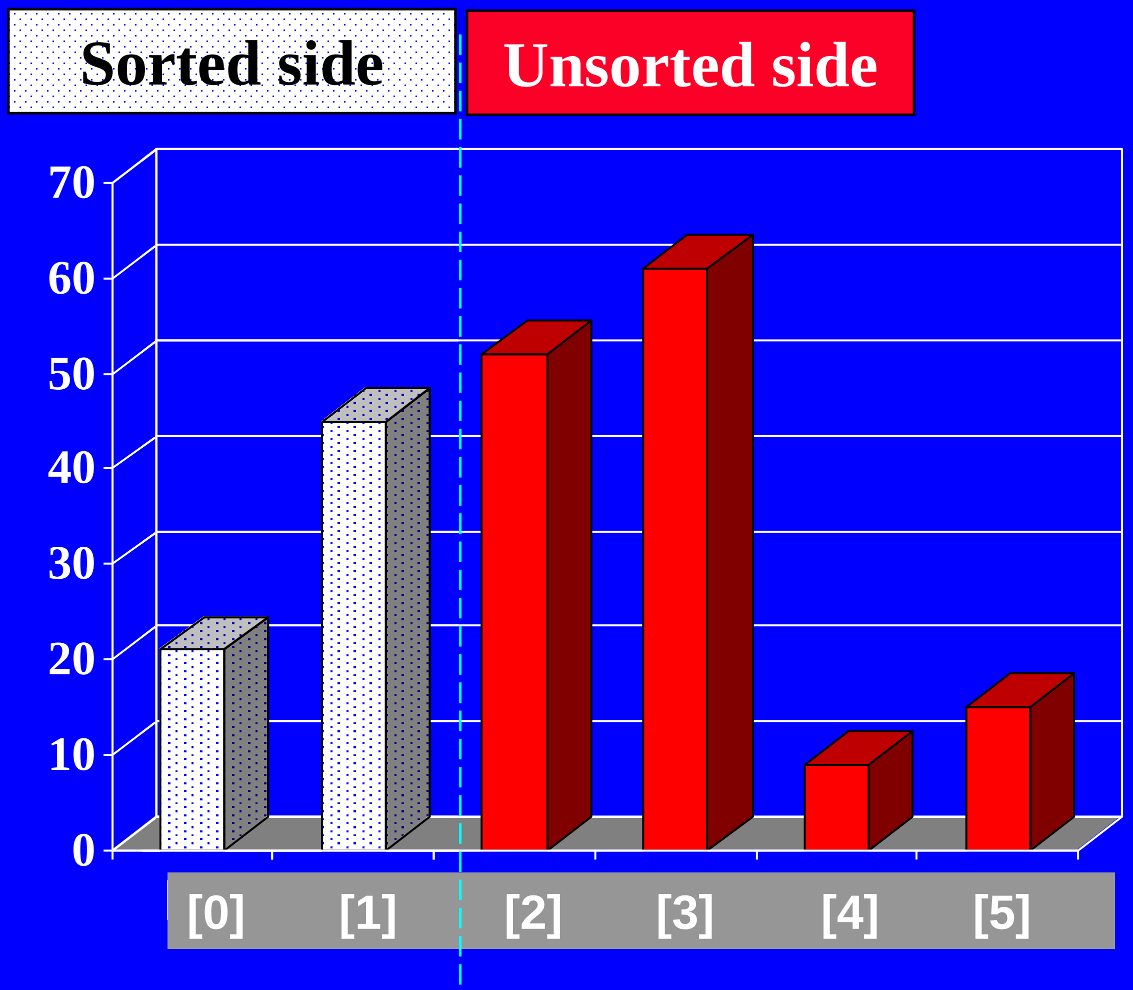


The Insertion Sort Algorithm

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

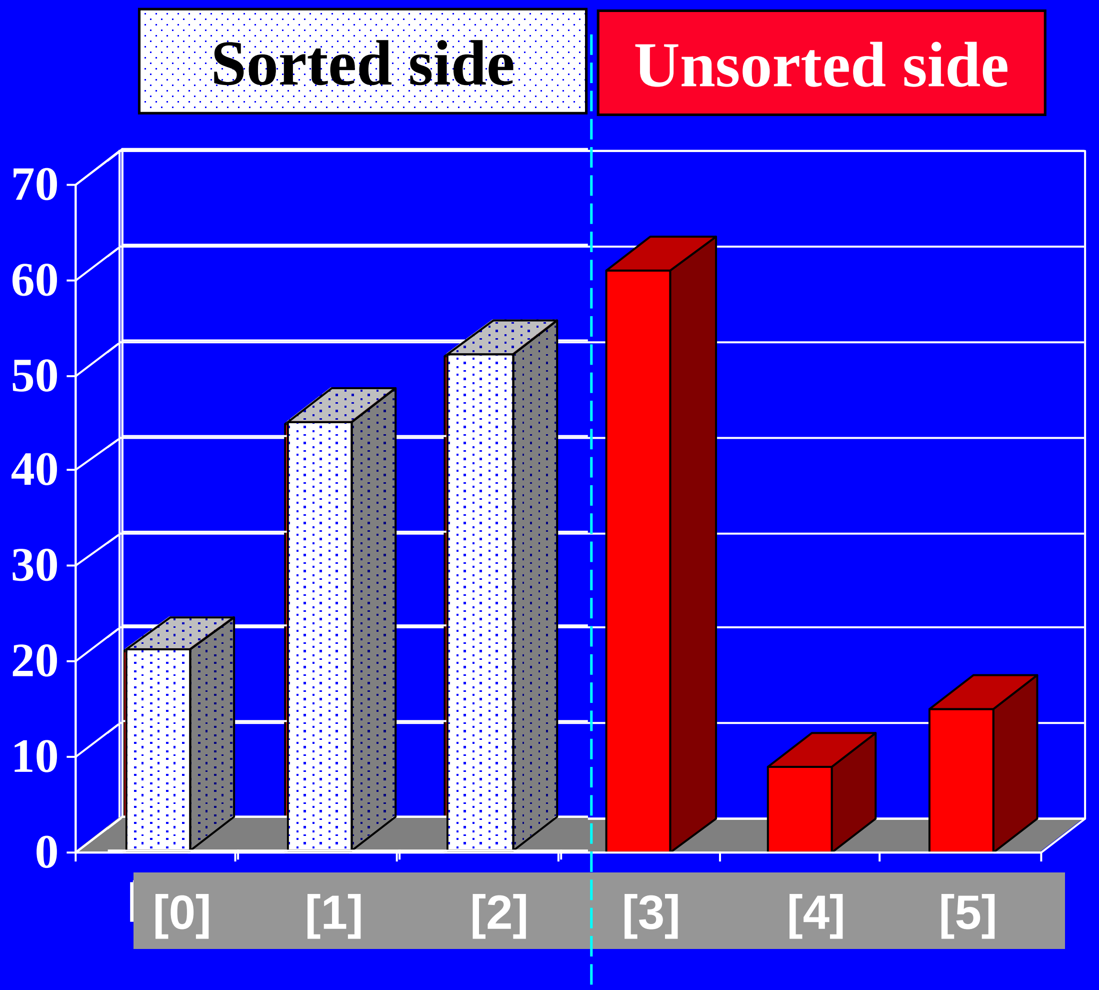


The Insertion Sort Algorithm



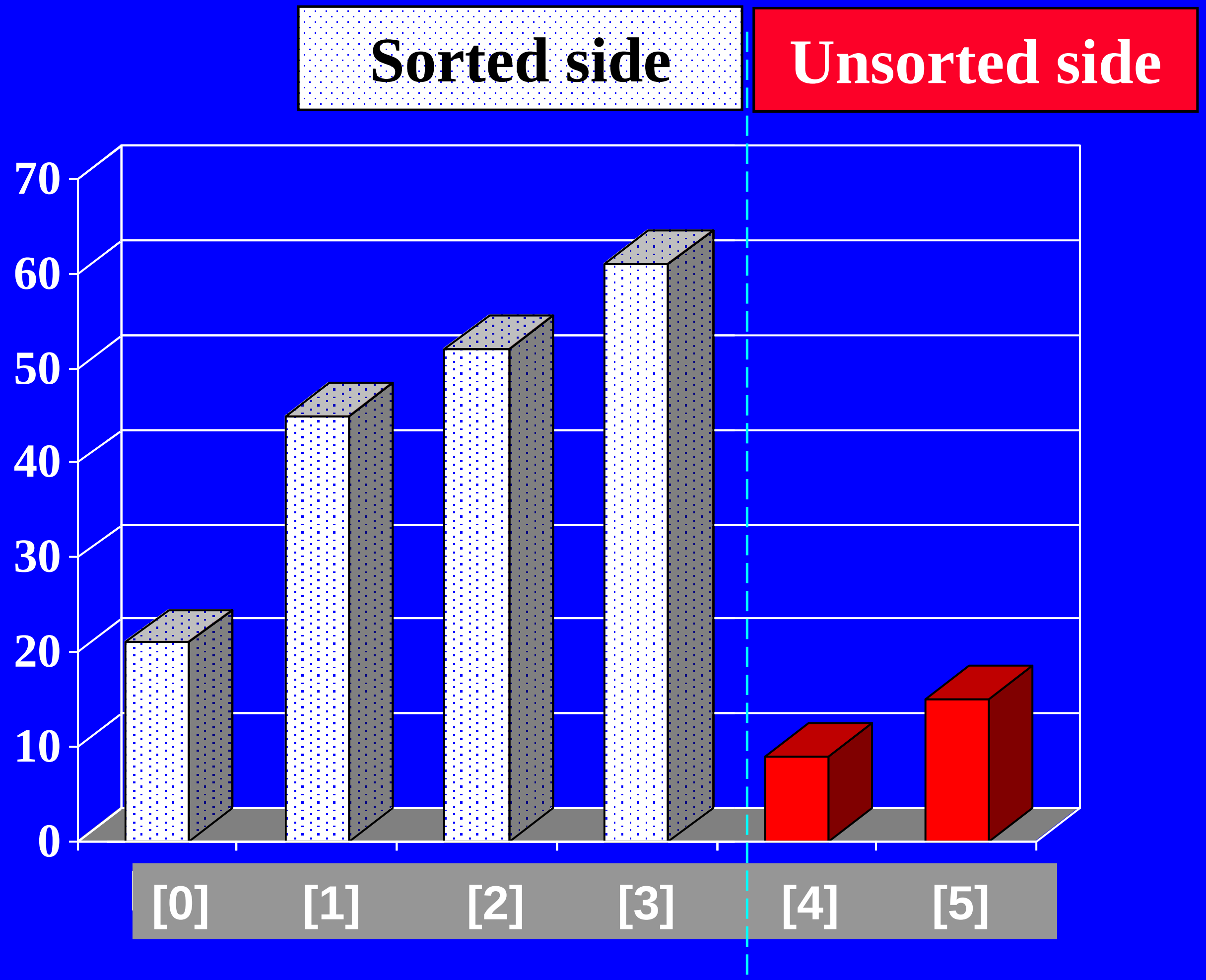
The Insertion Sort Algorithm

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



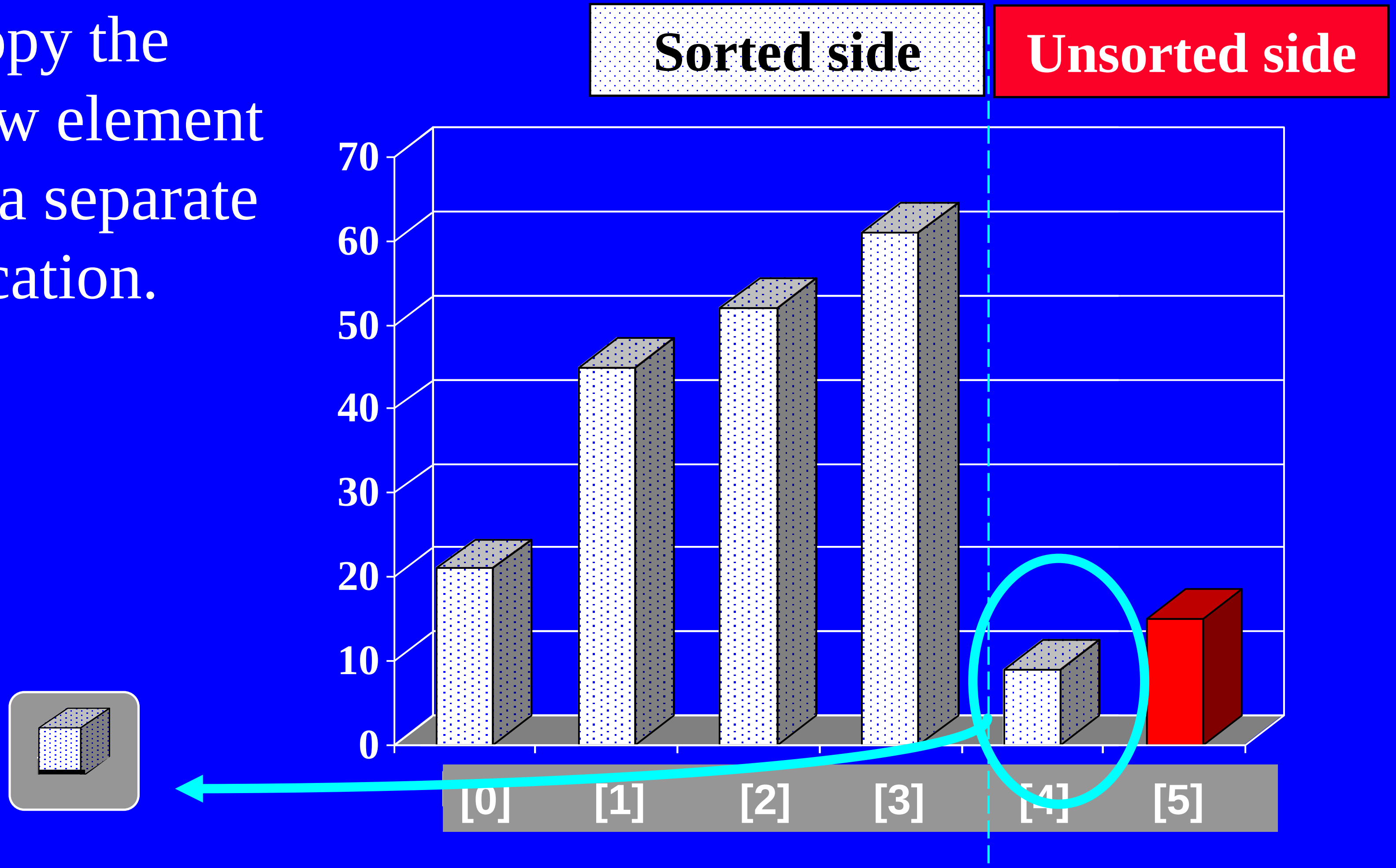
The Insertionsort Algorithm

- Sometimes we are lucky twice in a row.



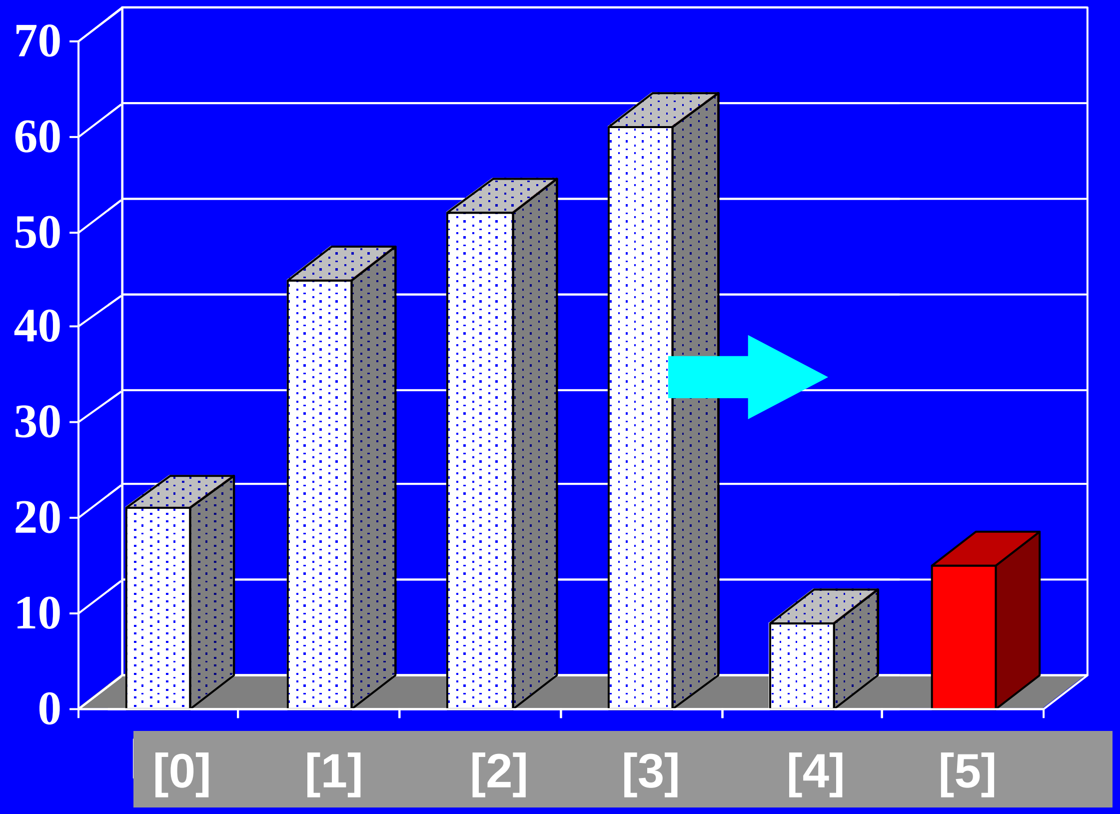
How to Insert One Element

- ☆ Copy the new element to a separate location.



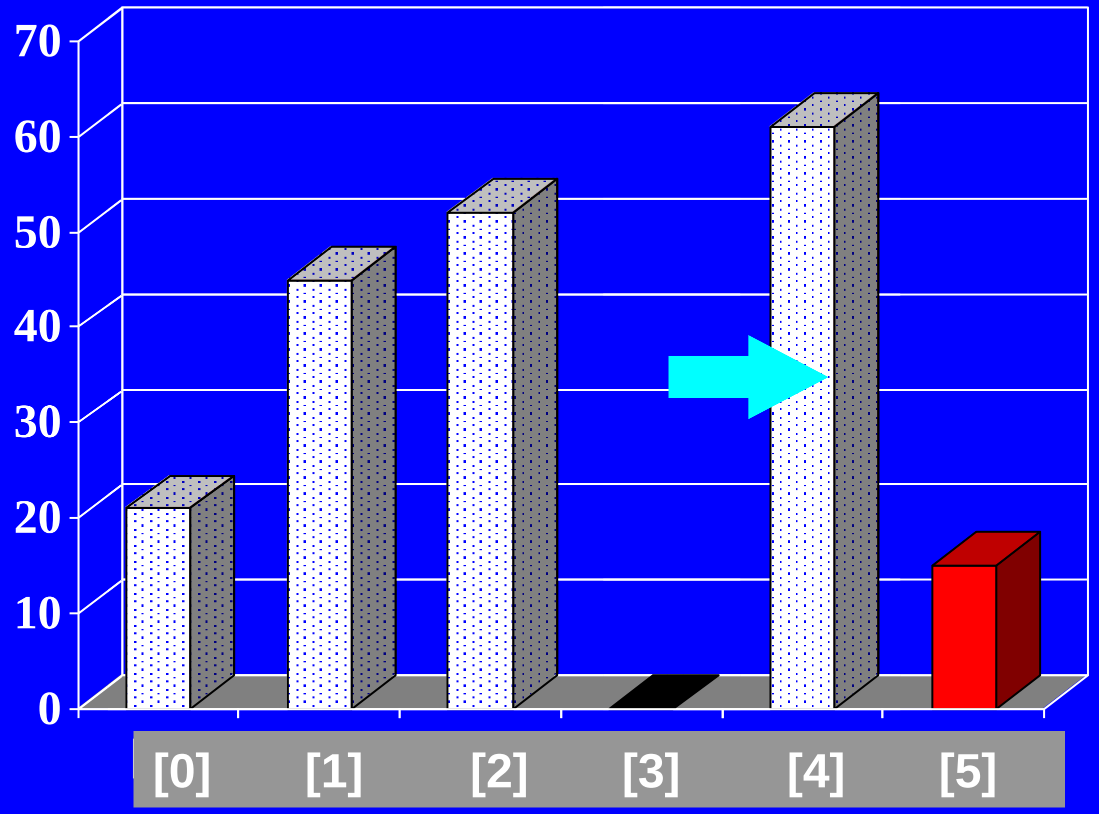
How to Insert One Element

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



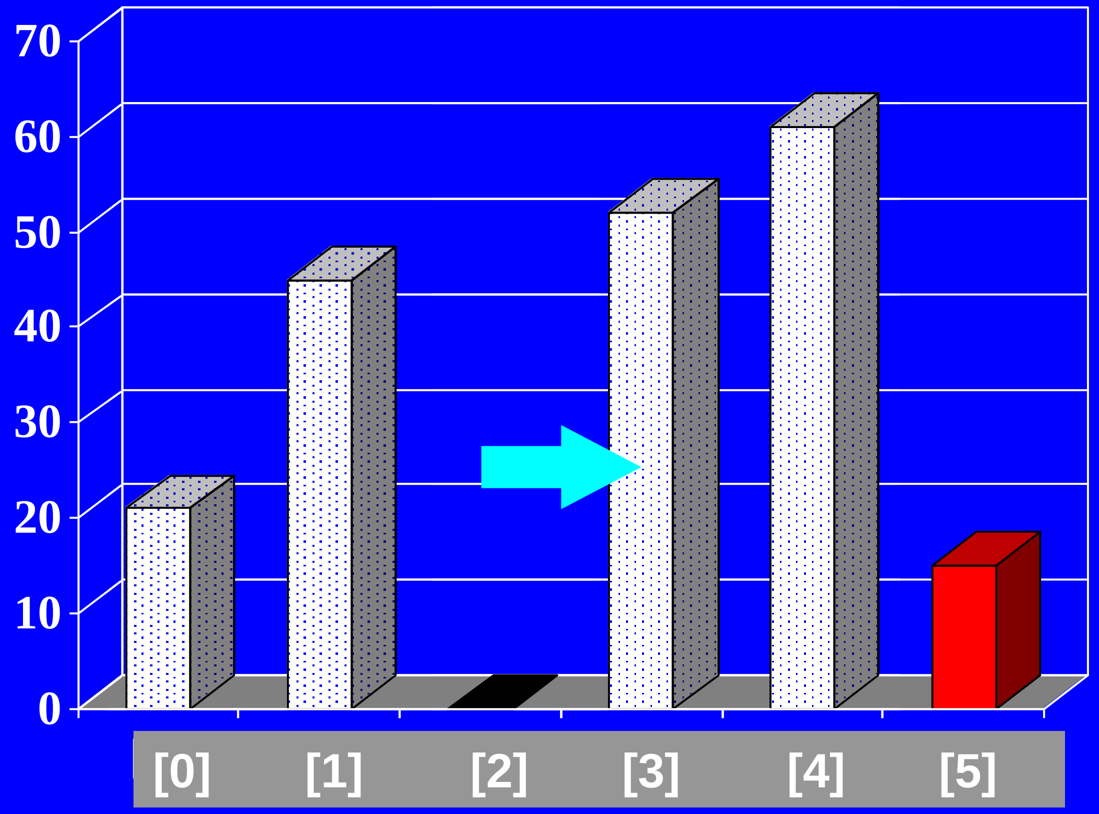
How to Insert One Element

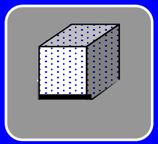
- Shift elements in the sorted side, creating an open space for the new element.



How to Insert One Element

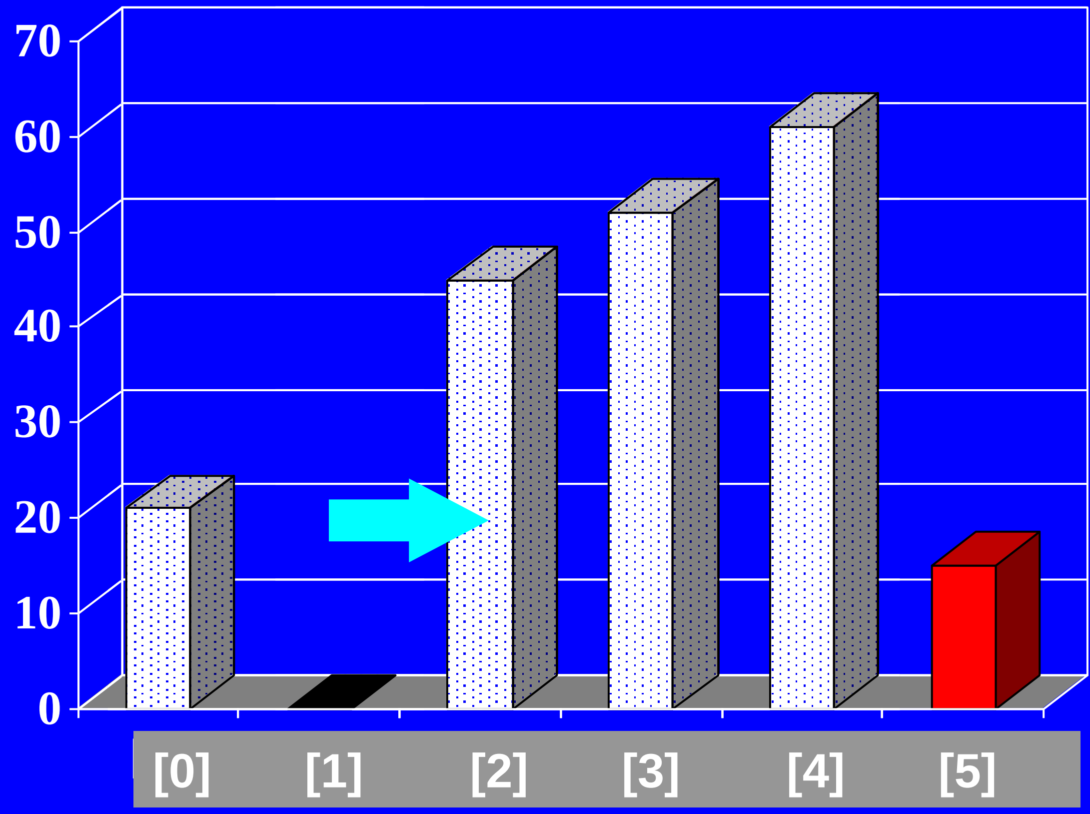
- Continue shifting elements...





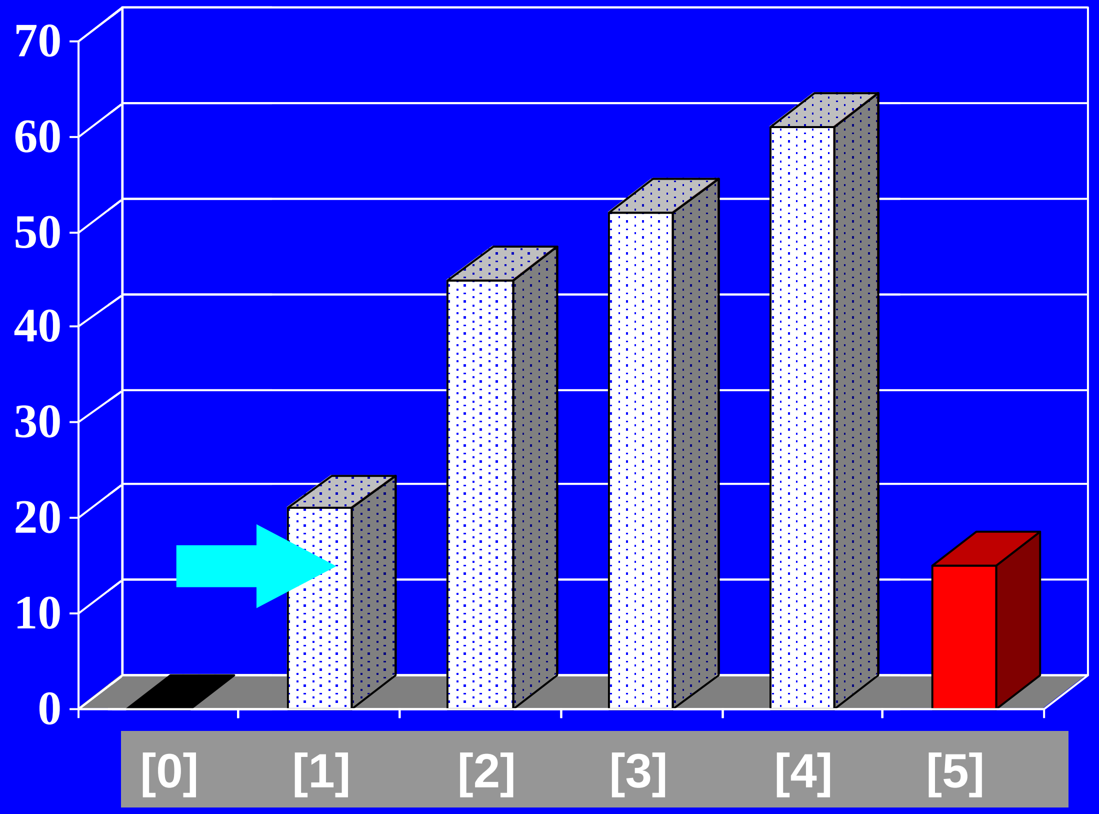
How to Insert One Element

- Continue shifting elements...



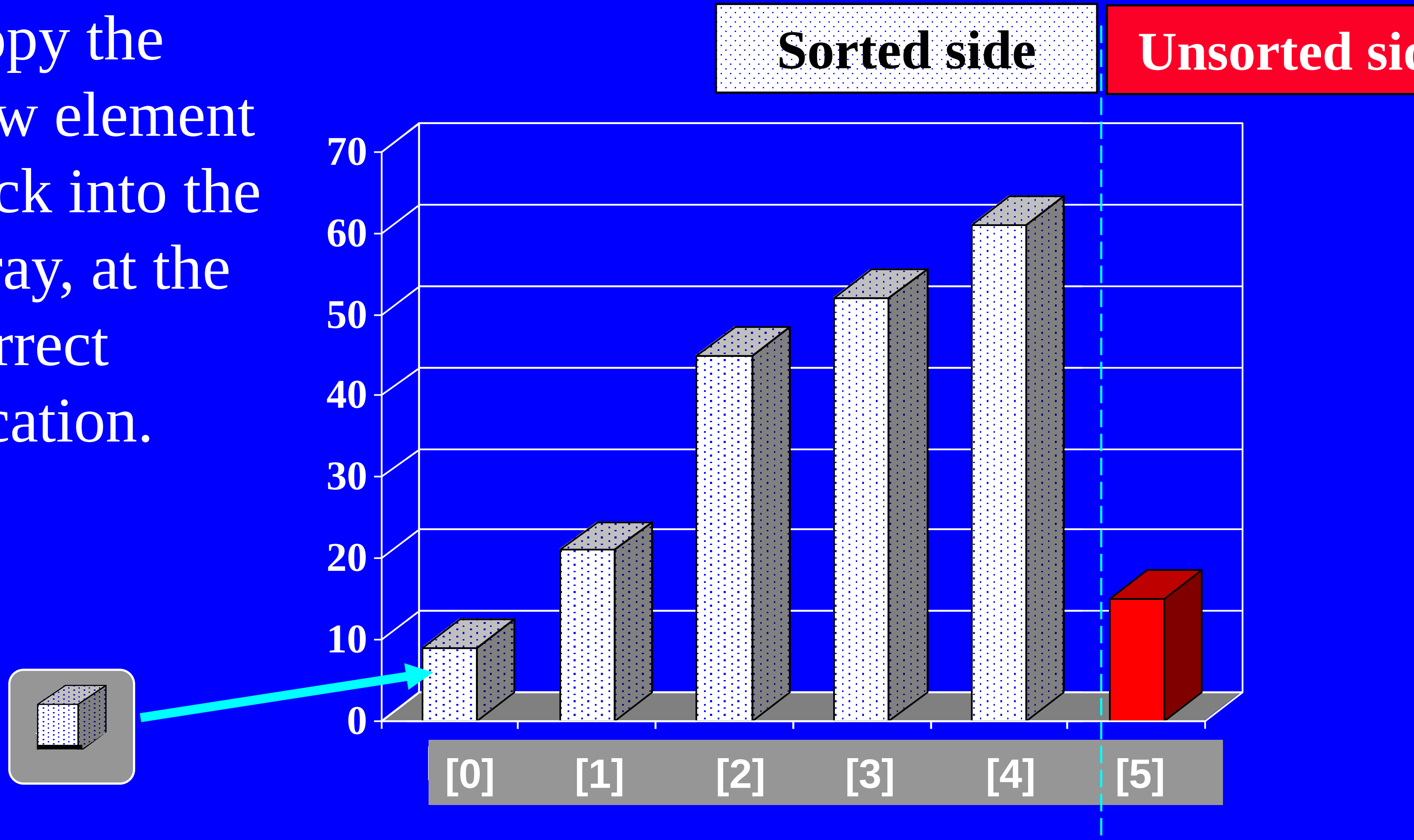
How to Insert One Element

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



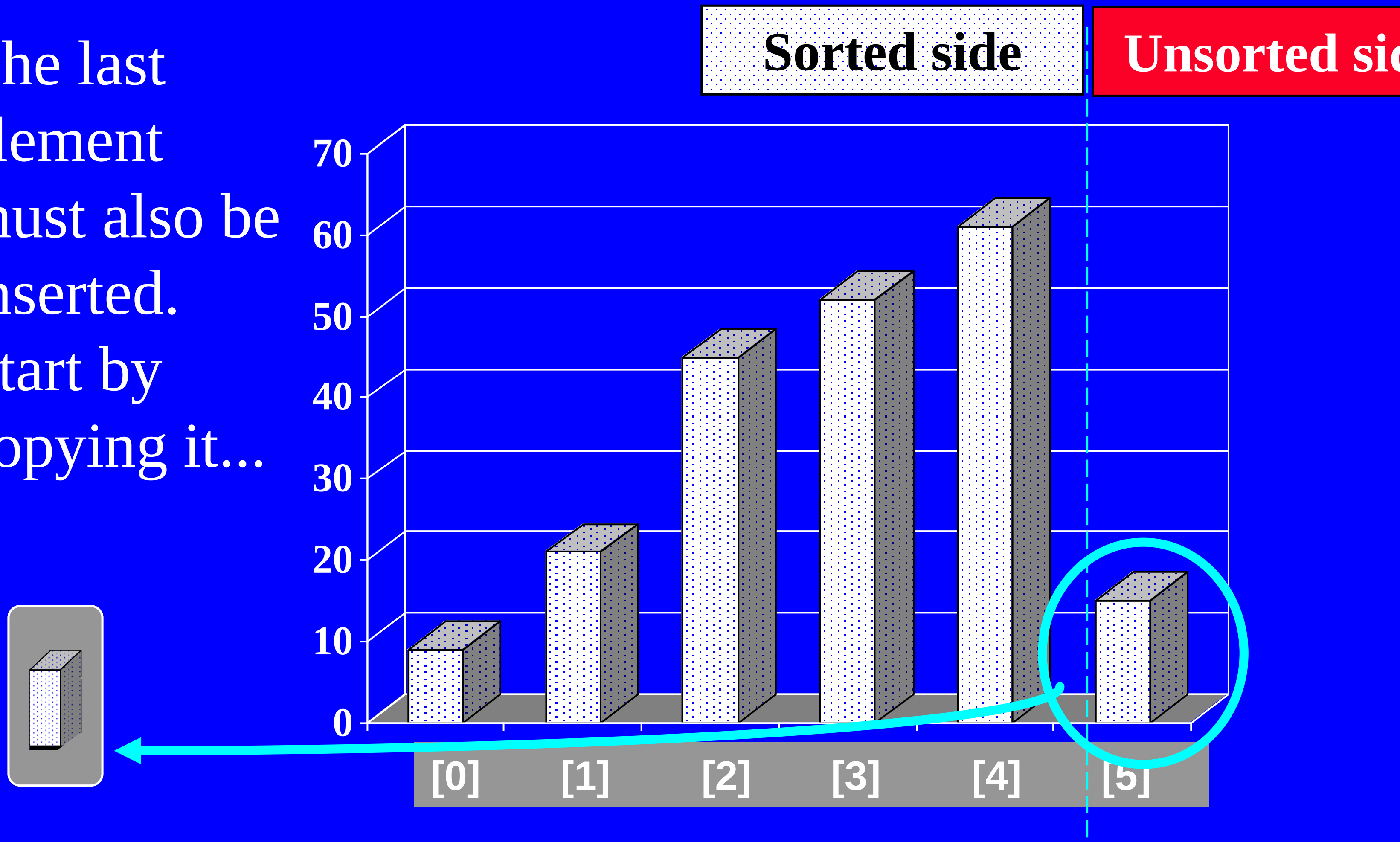
How to Insert One Element

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

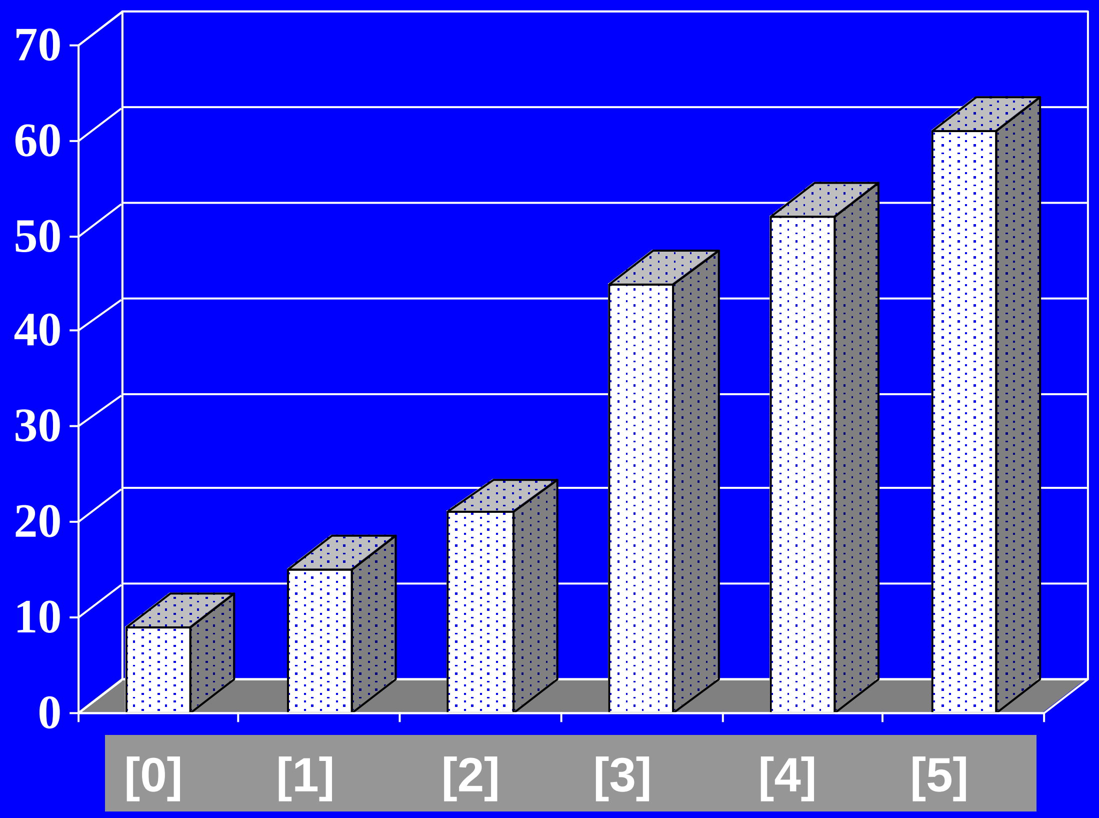


How to Insert One Element

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



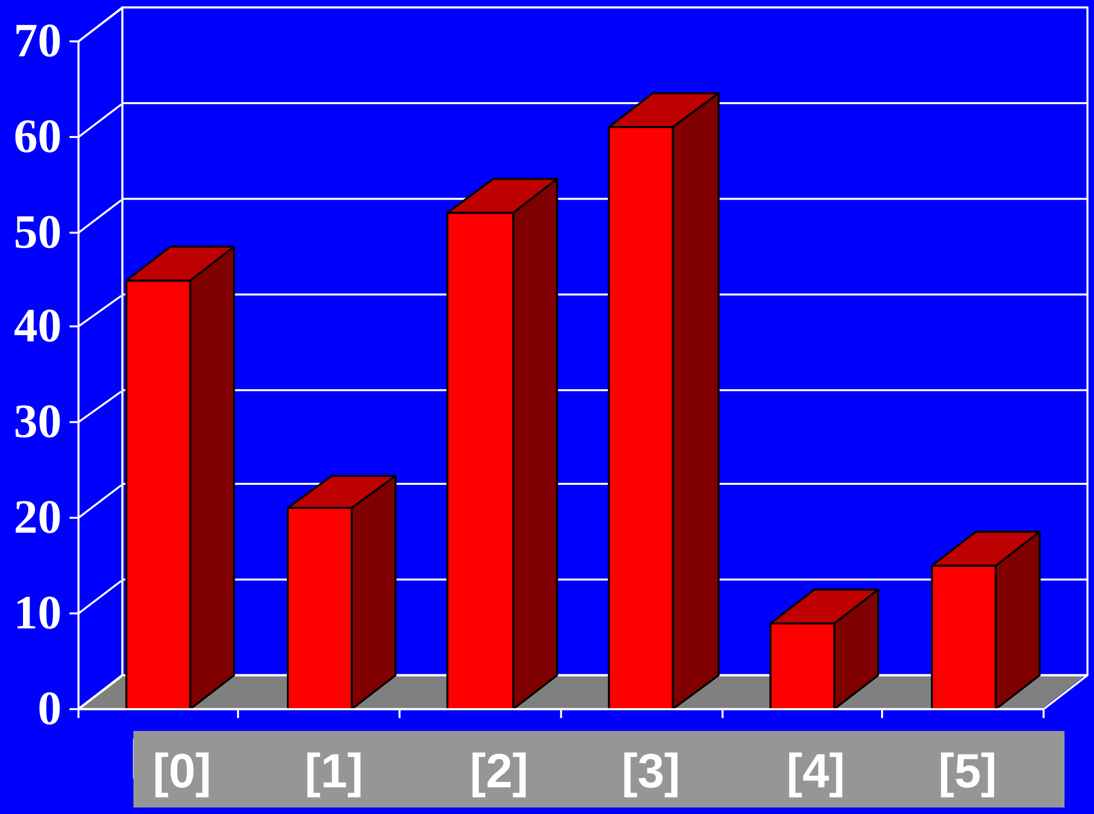
Sorted Result



```java
public static void insertionsort(int a[],int n)
{
  int space,value;

  for(int i=1;i<n;i++)
    {
          value = a[i];
          space = i;
          while(space>0 && a[space-1]>value)
          {
             a[space]=a[space-1];
             space = space-1;
          }
          a[space]=value;
    }
```

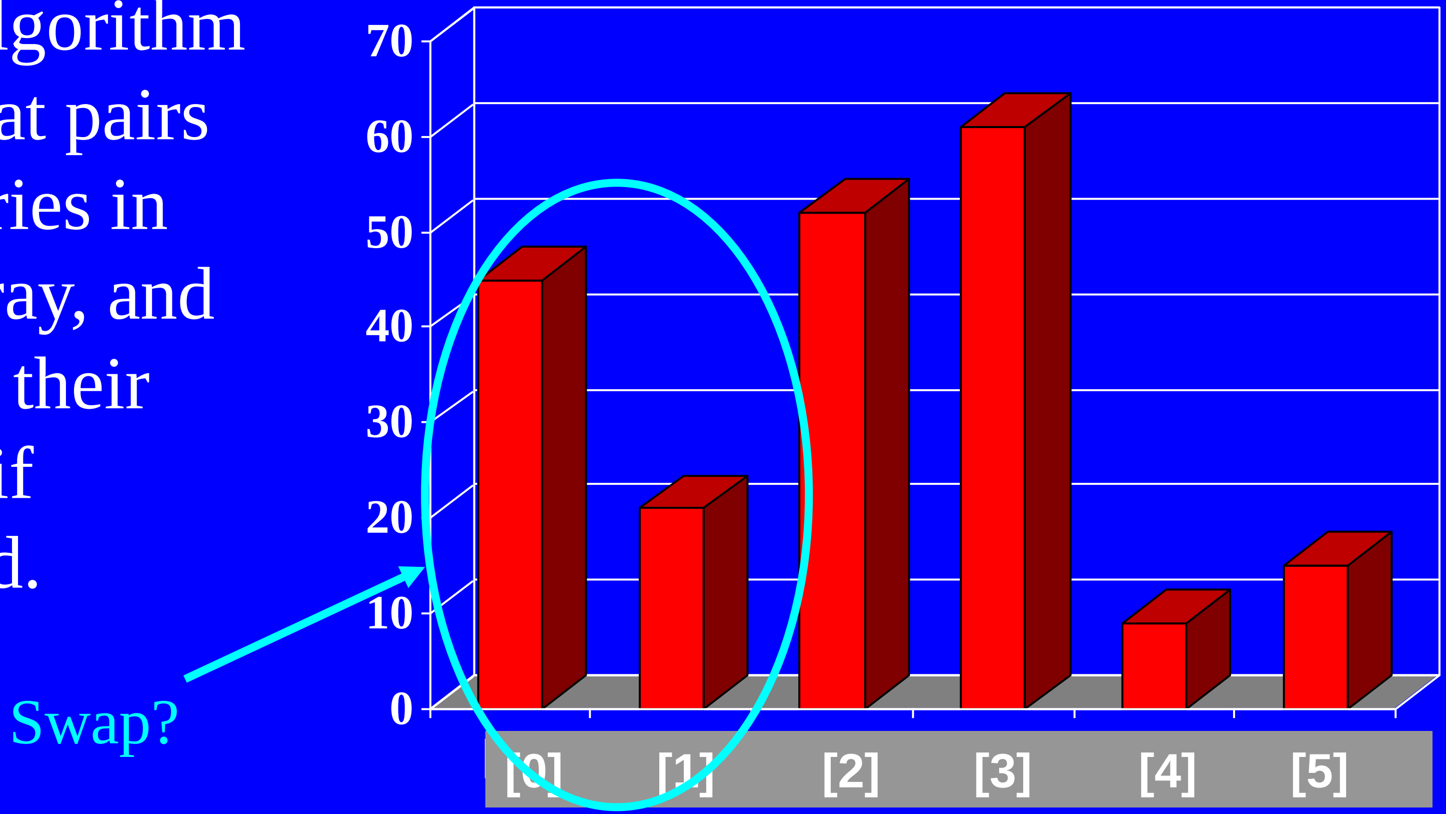
The Bubble Sort Algorithm

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



The Bubble Sort Algorithm

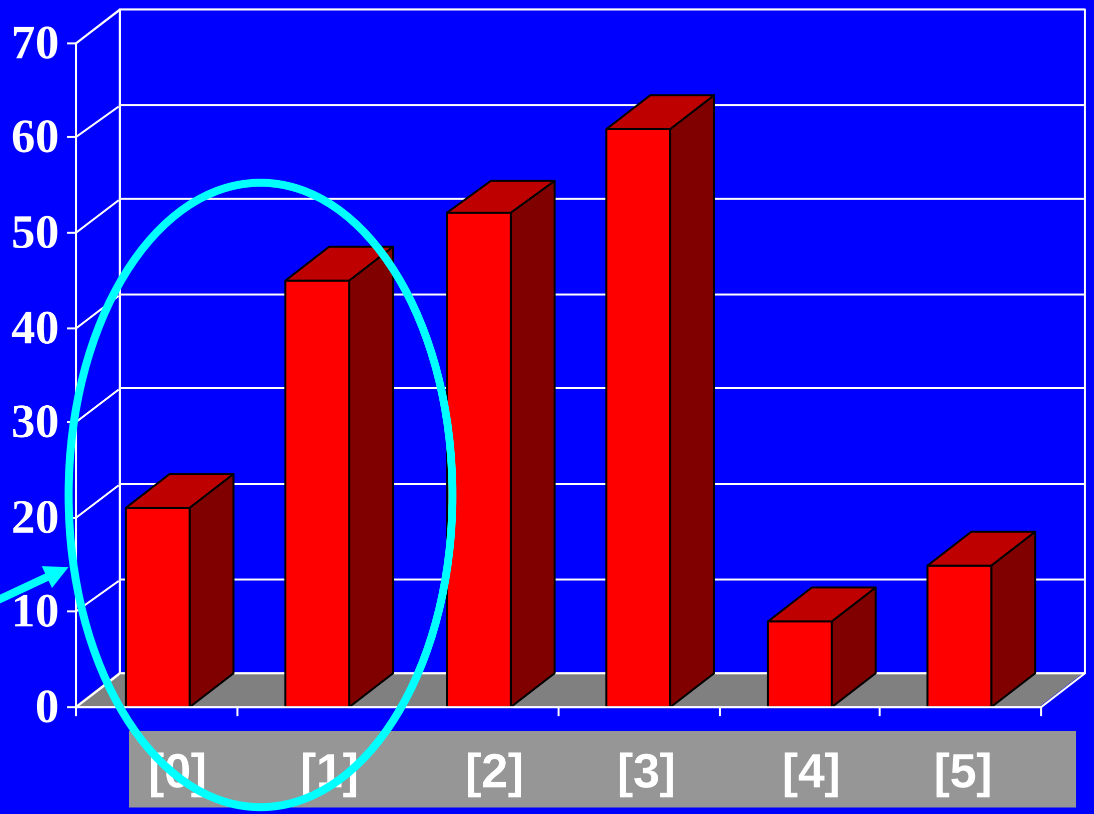
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



The Bubble Sort Algorithm

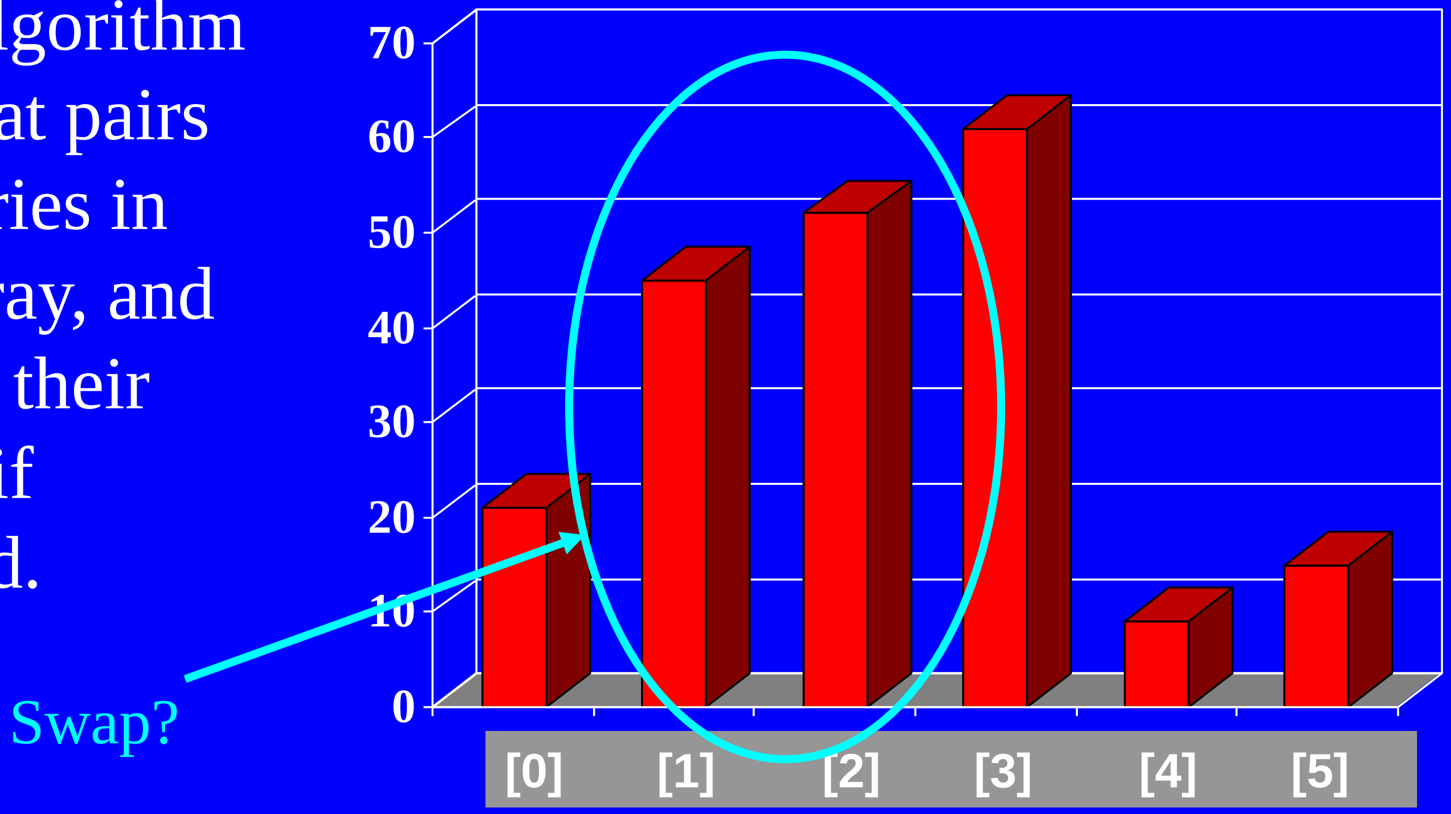
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.

Yes!



The Bubble Sort Algorithm

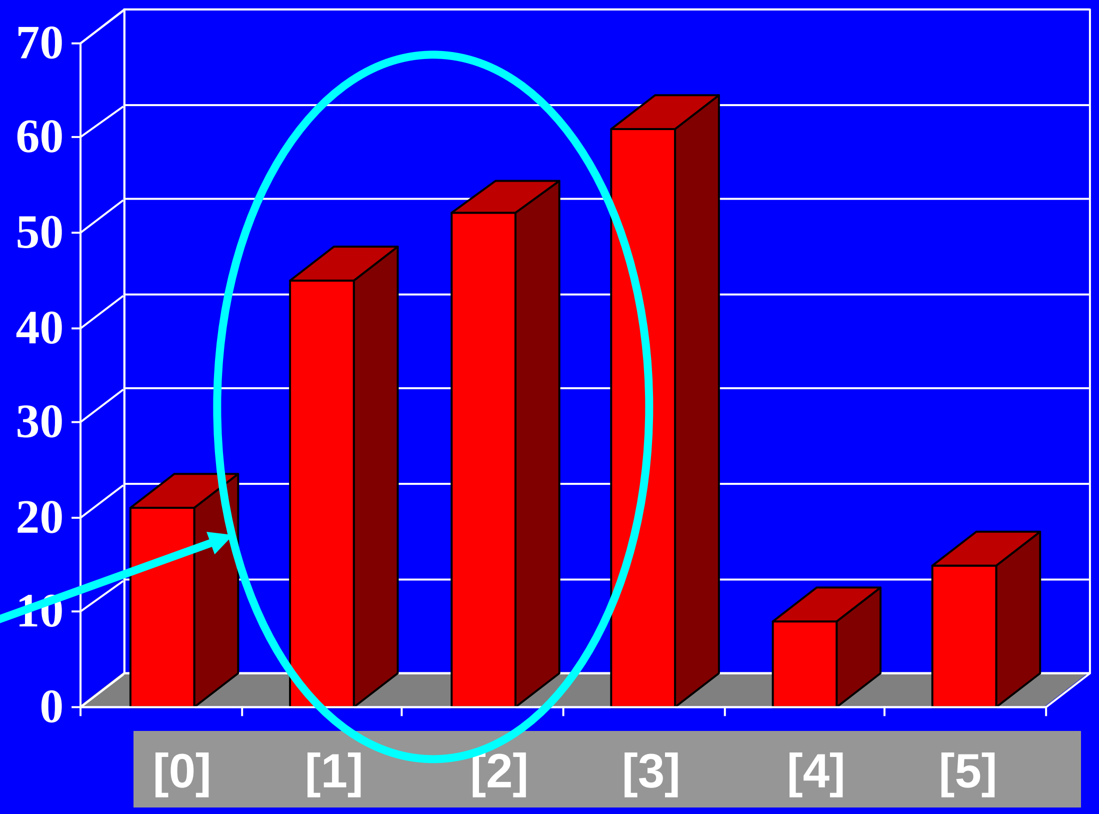
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



The Bubble Sort Algorithm

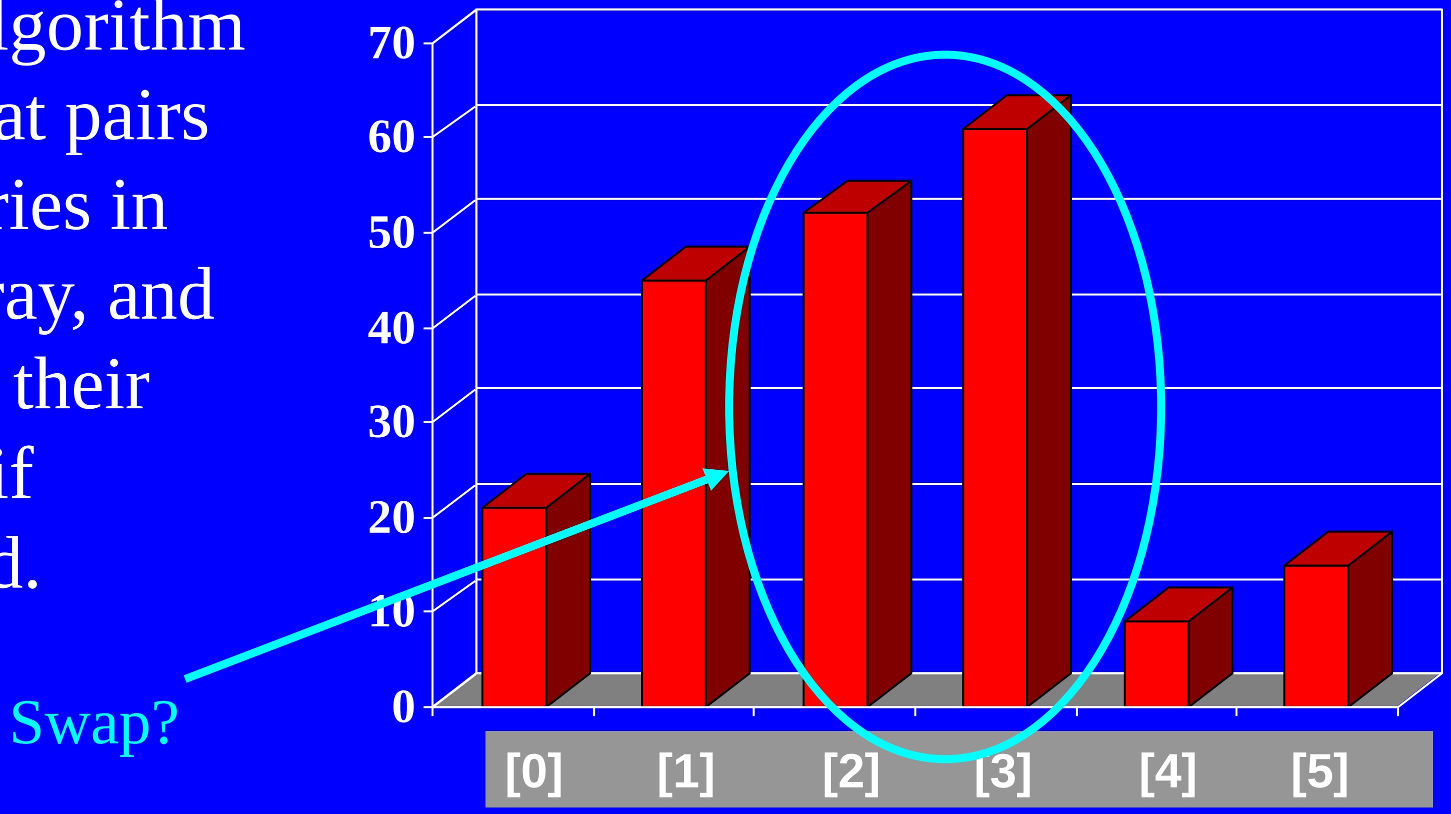
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.

No.



The Bubble Sort Algorithm

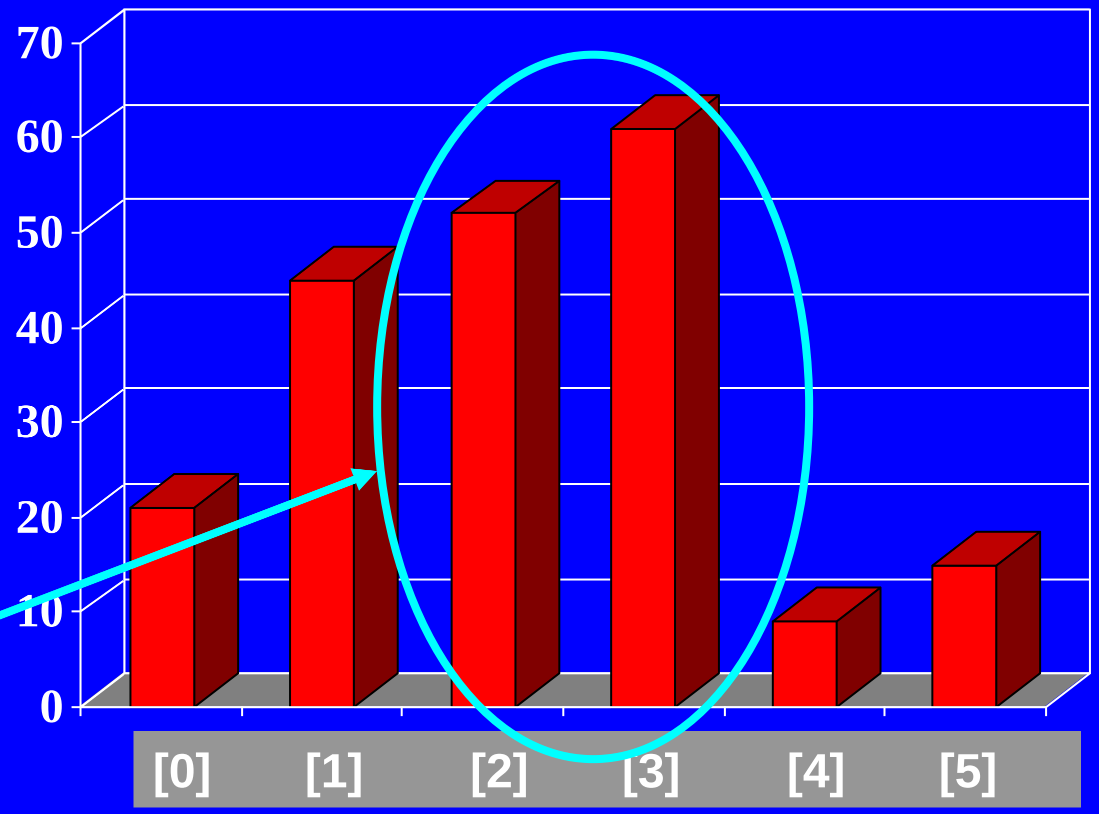
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



The Bubble Sort Algorithm

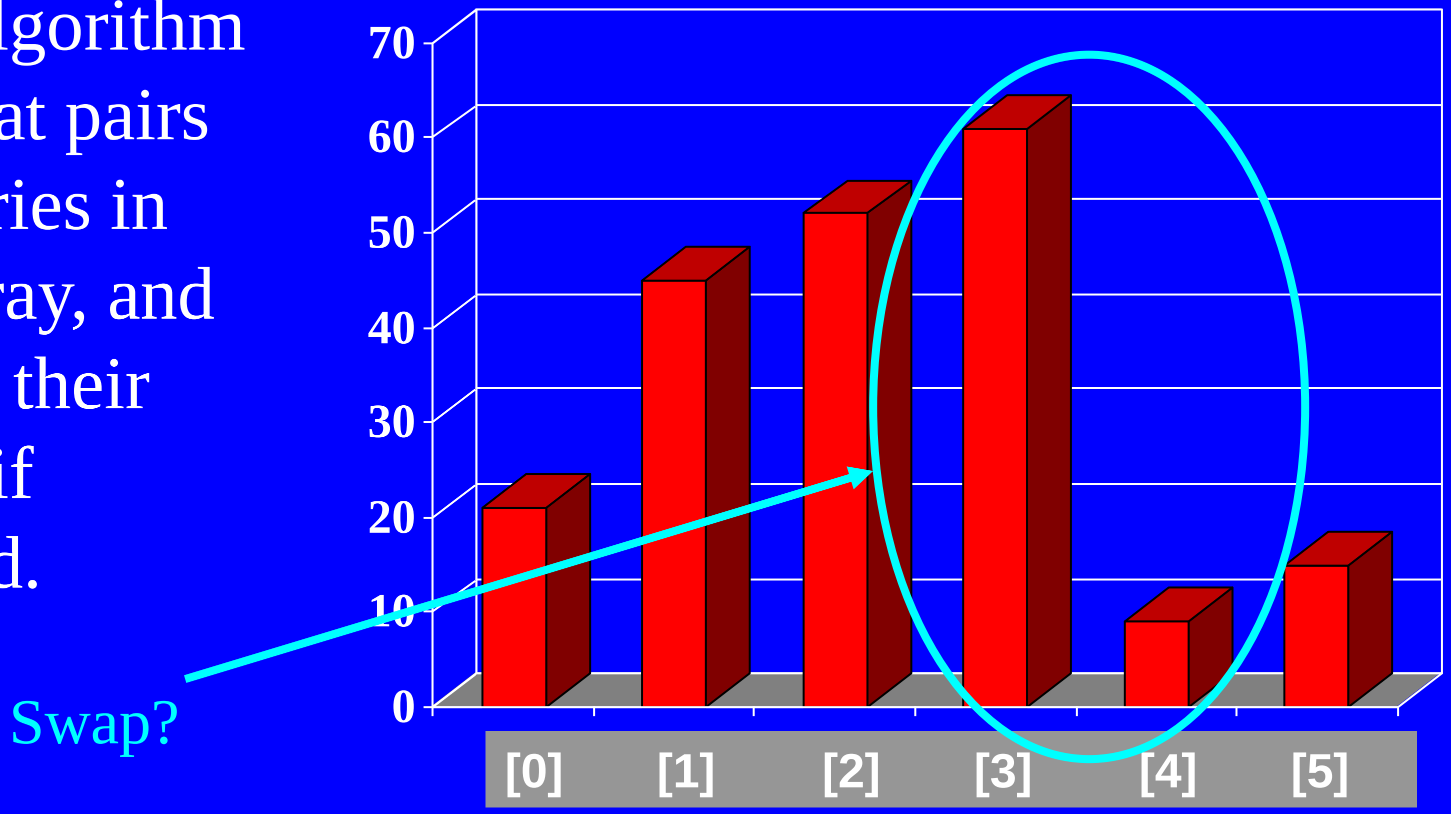
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.

No.



The Bubble Sort Algorithm

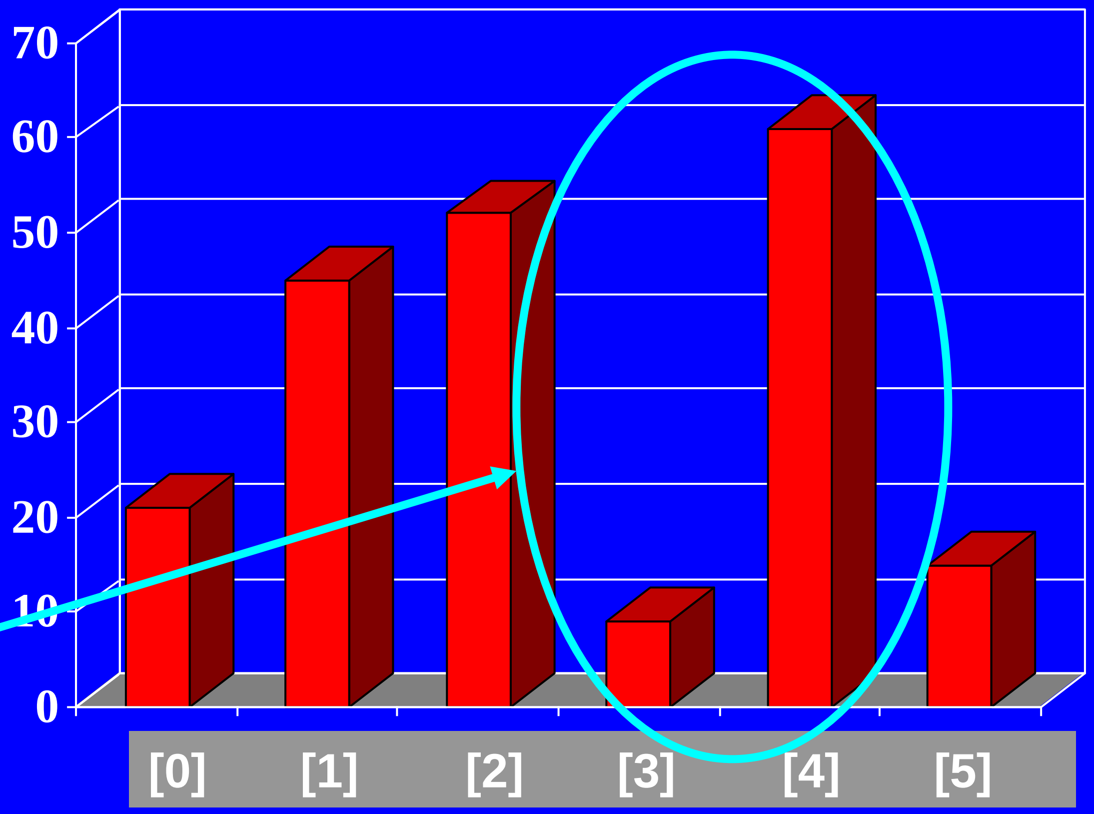
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



The Bubble Sort Algorithm

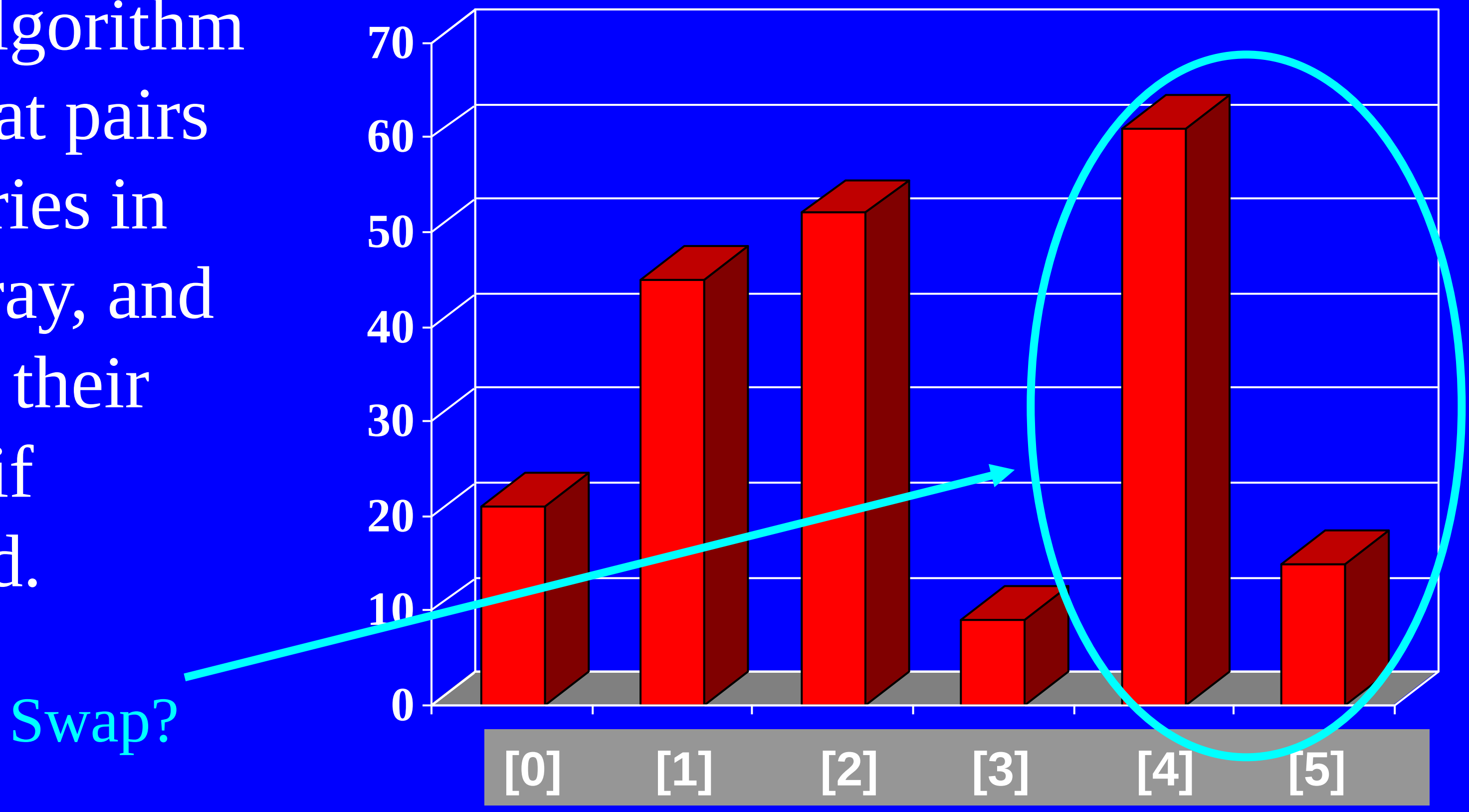
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.

Yes!



The Bubble Sort Algorithm

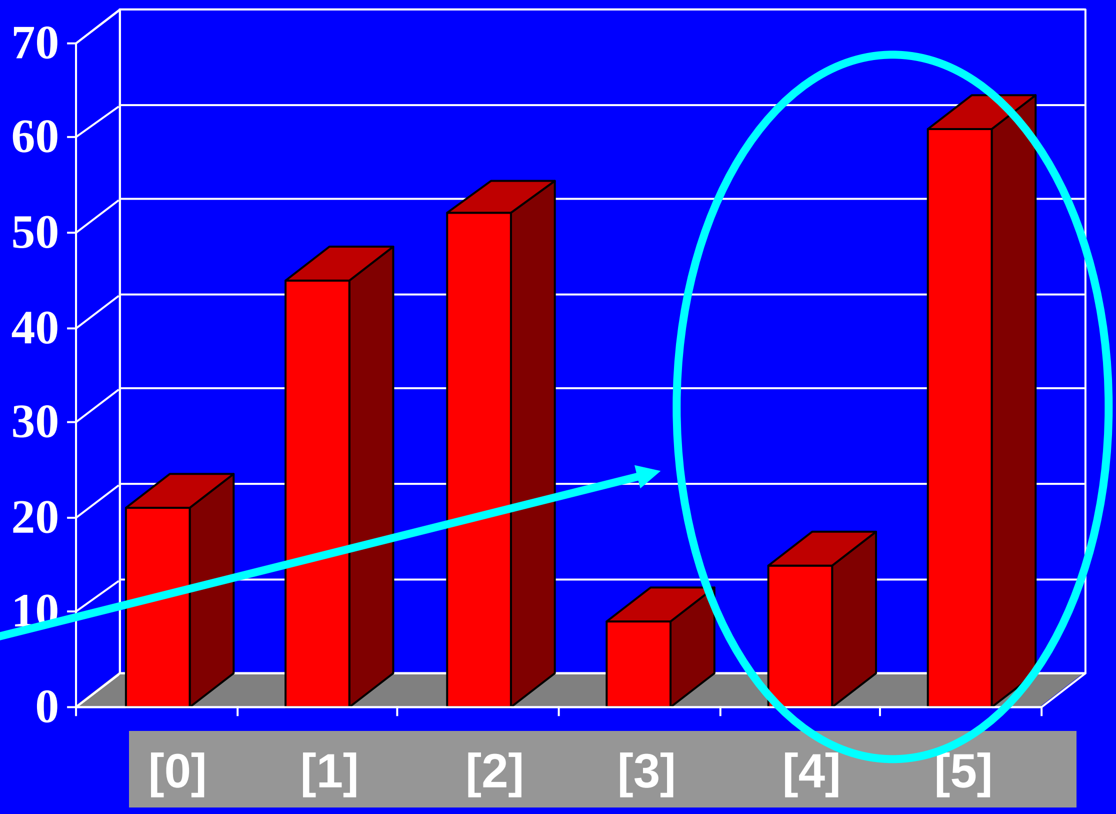
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



The Bubble Sort Algorithm

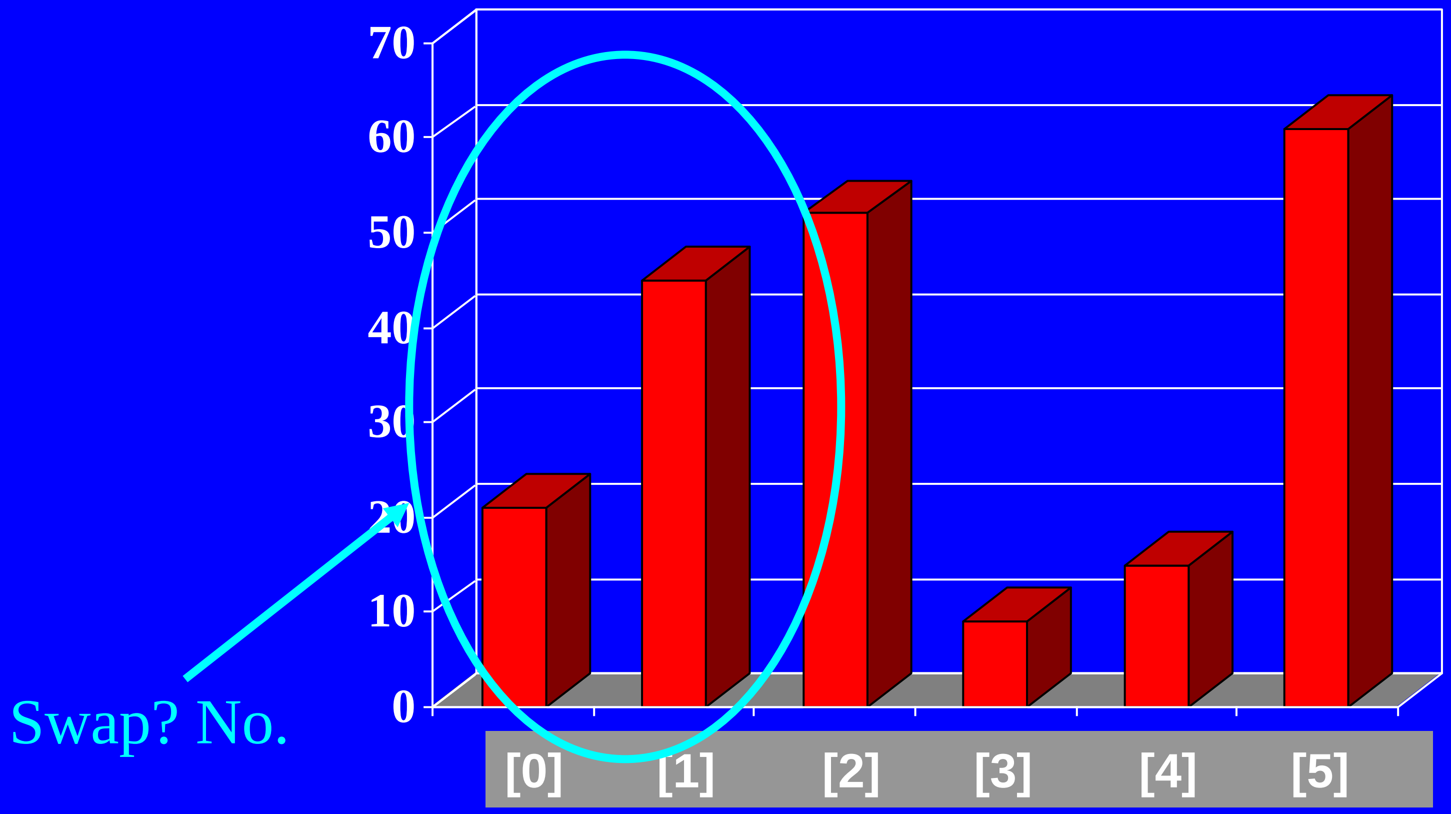
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.

Yes!



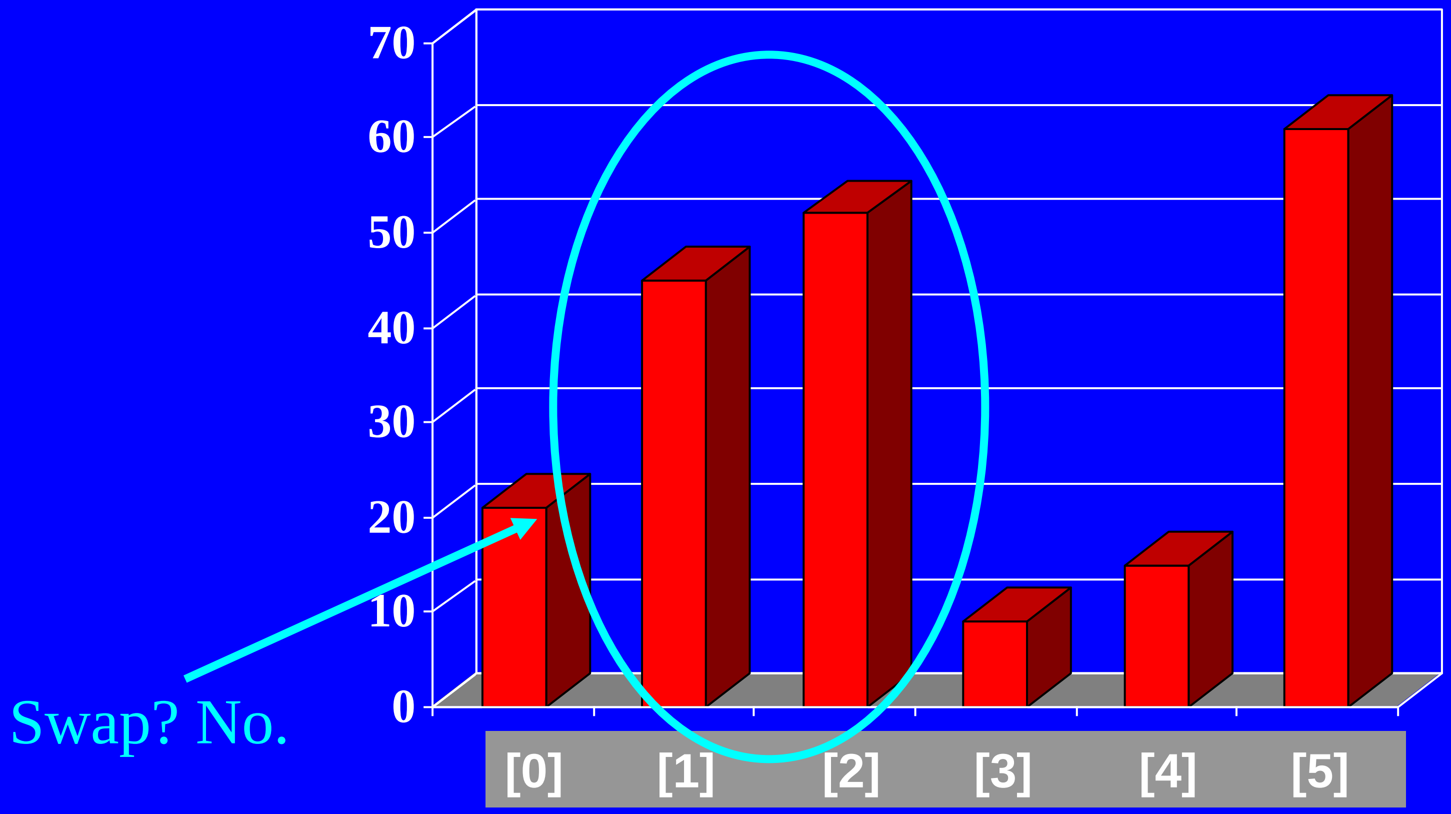
The Bubble Sort Algorithm

- Repeat.



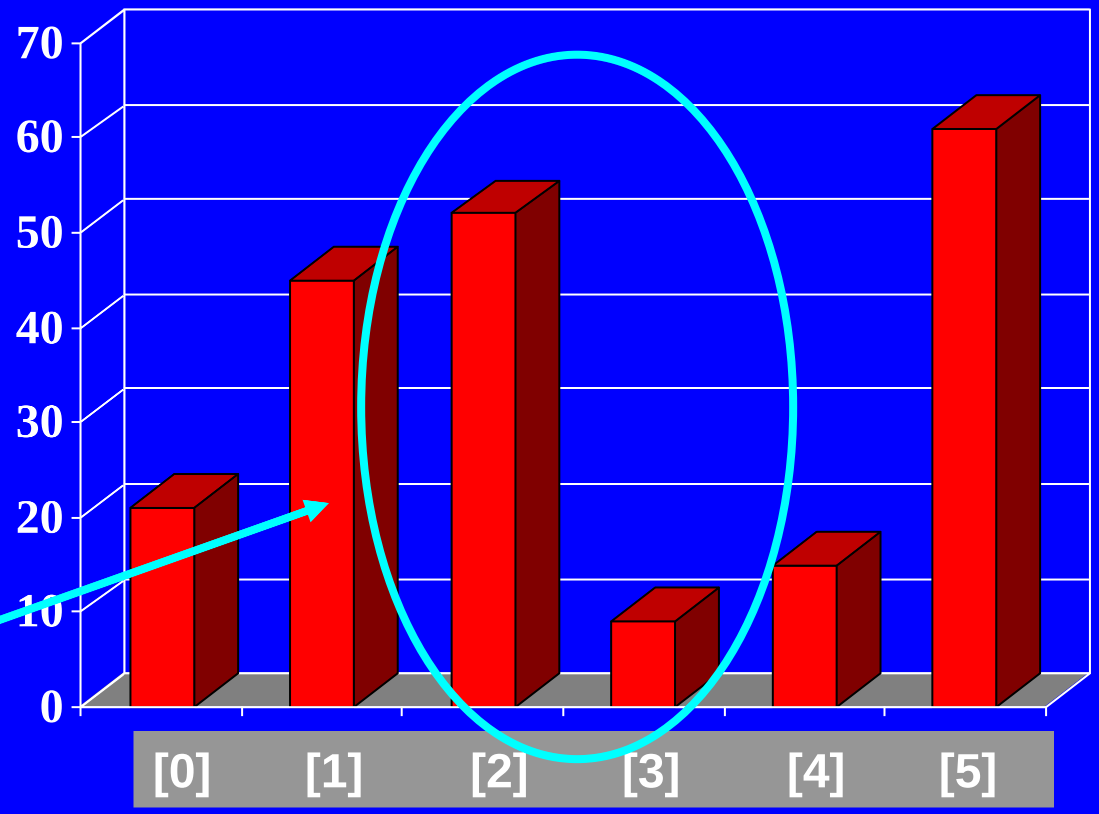
The Bubble Sort Algorithm

- Repeat.



The Bubble Sort Algorithm

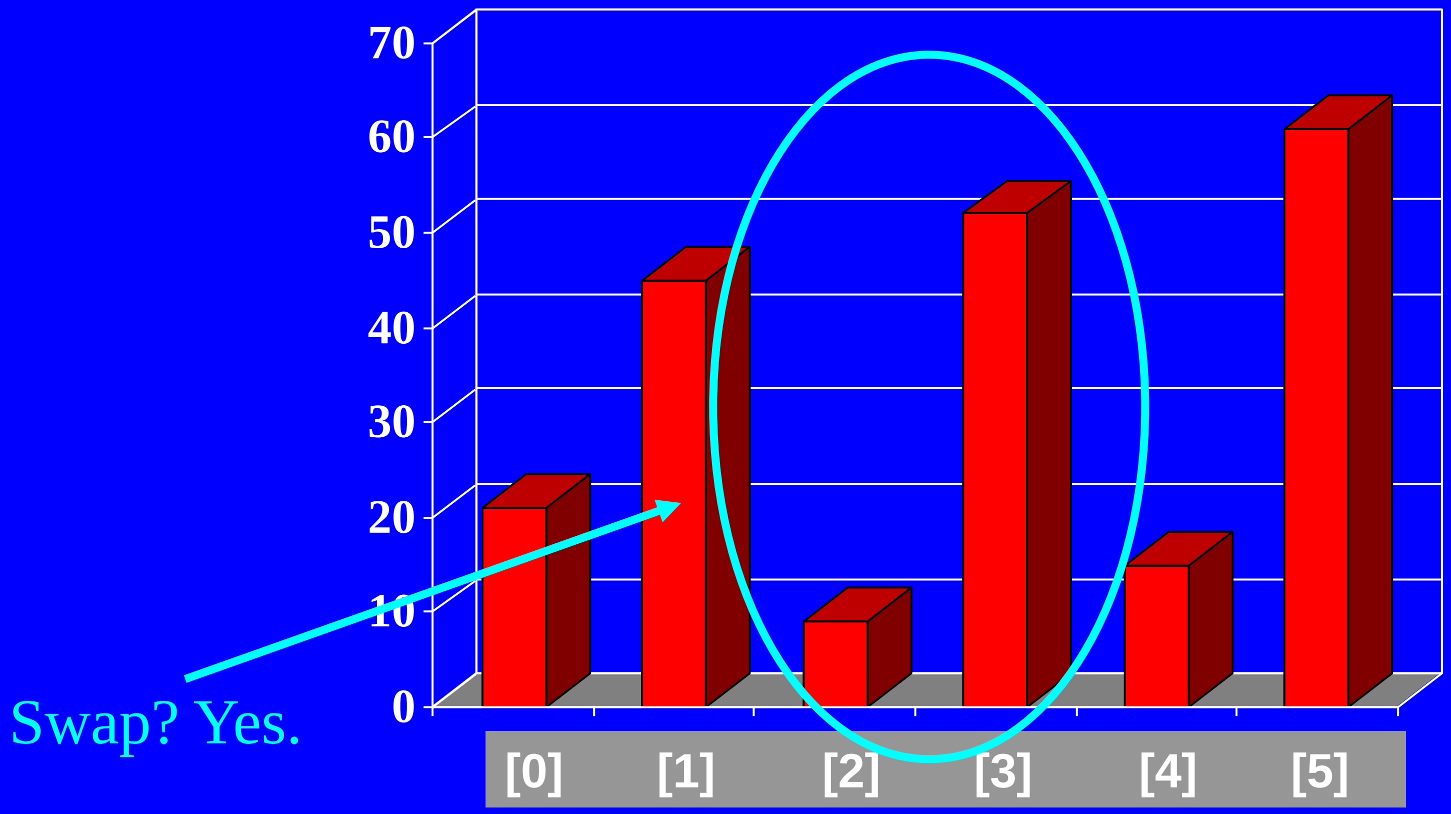
- Repeat.



Swap? Yes.

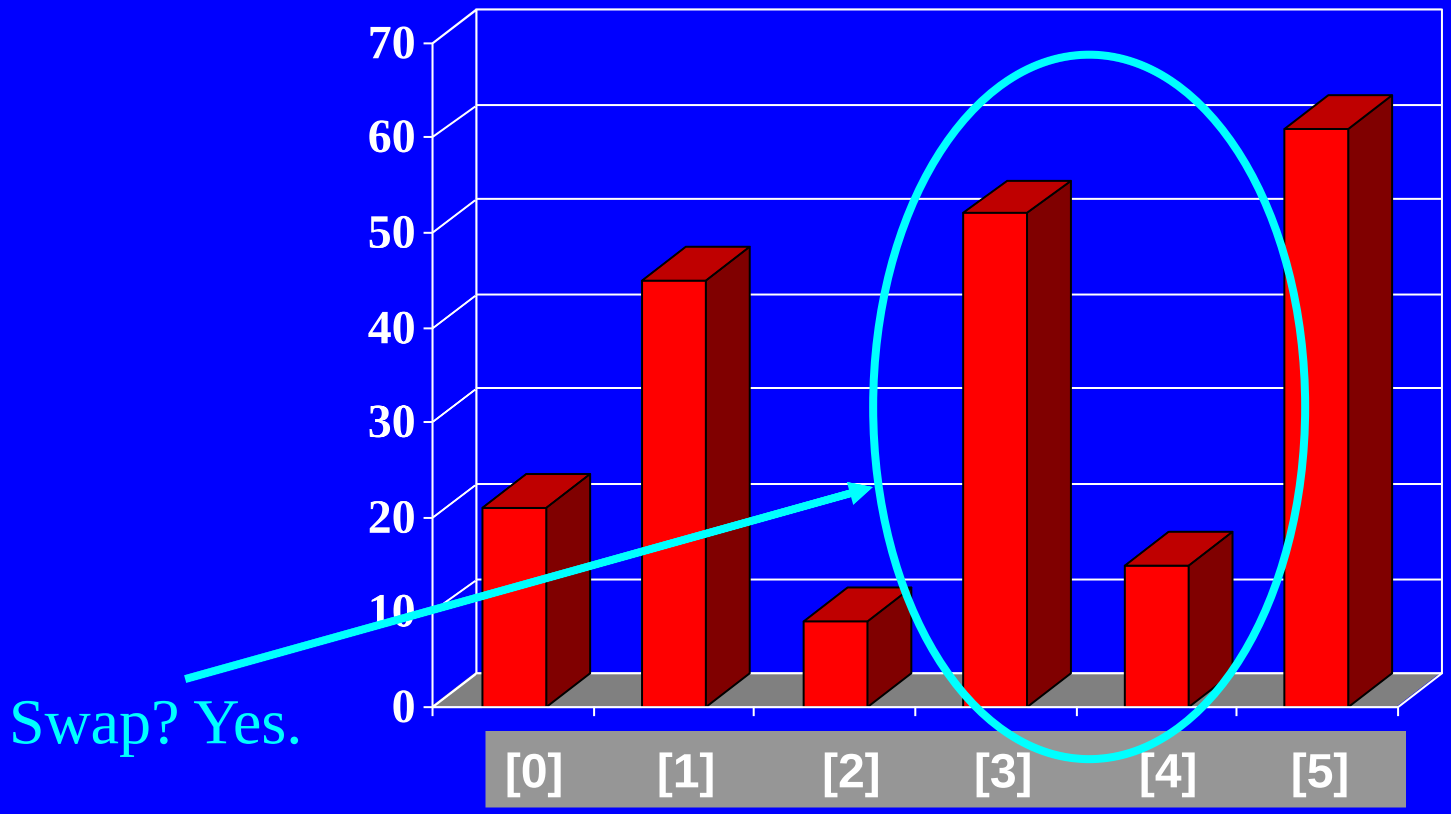
The Bubble Sort Algorithm

- Repeat.



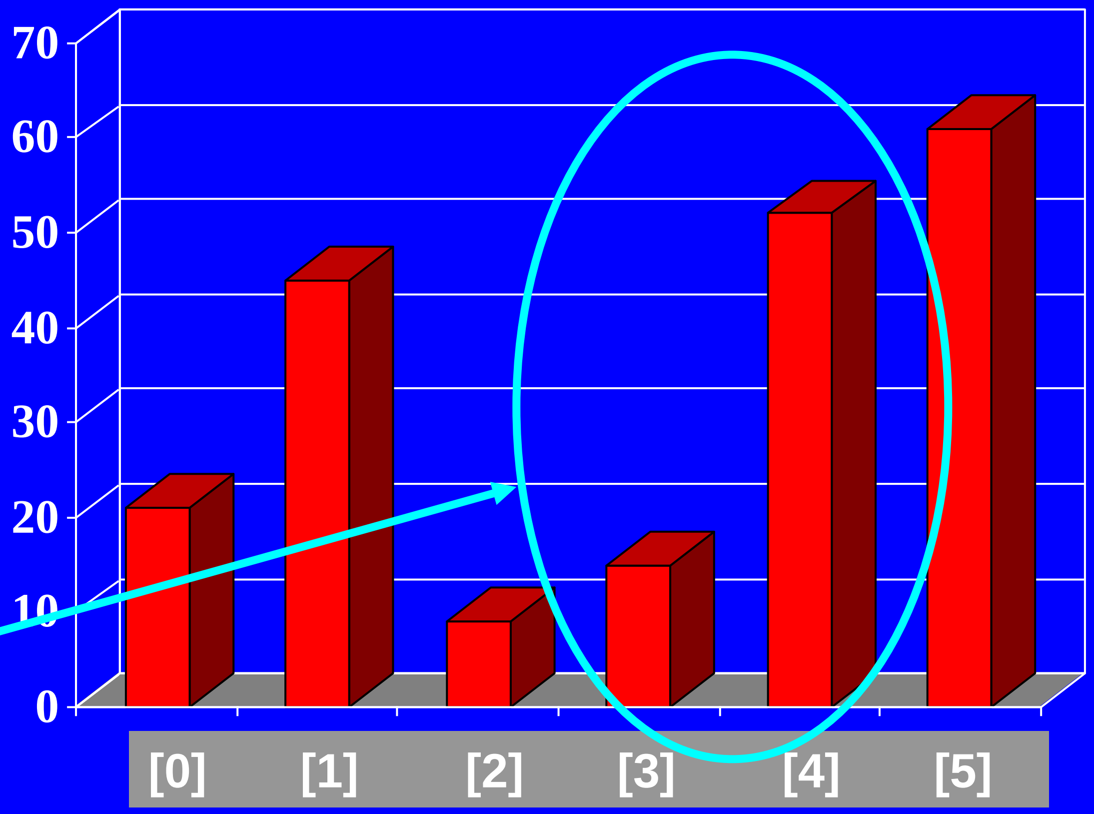
The Bubble Sort Algorithm

- Repeat.



The Bubble Sort Algorithm

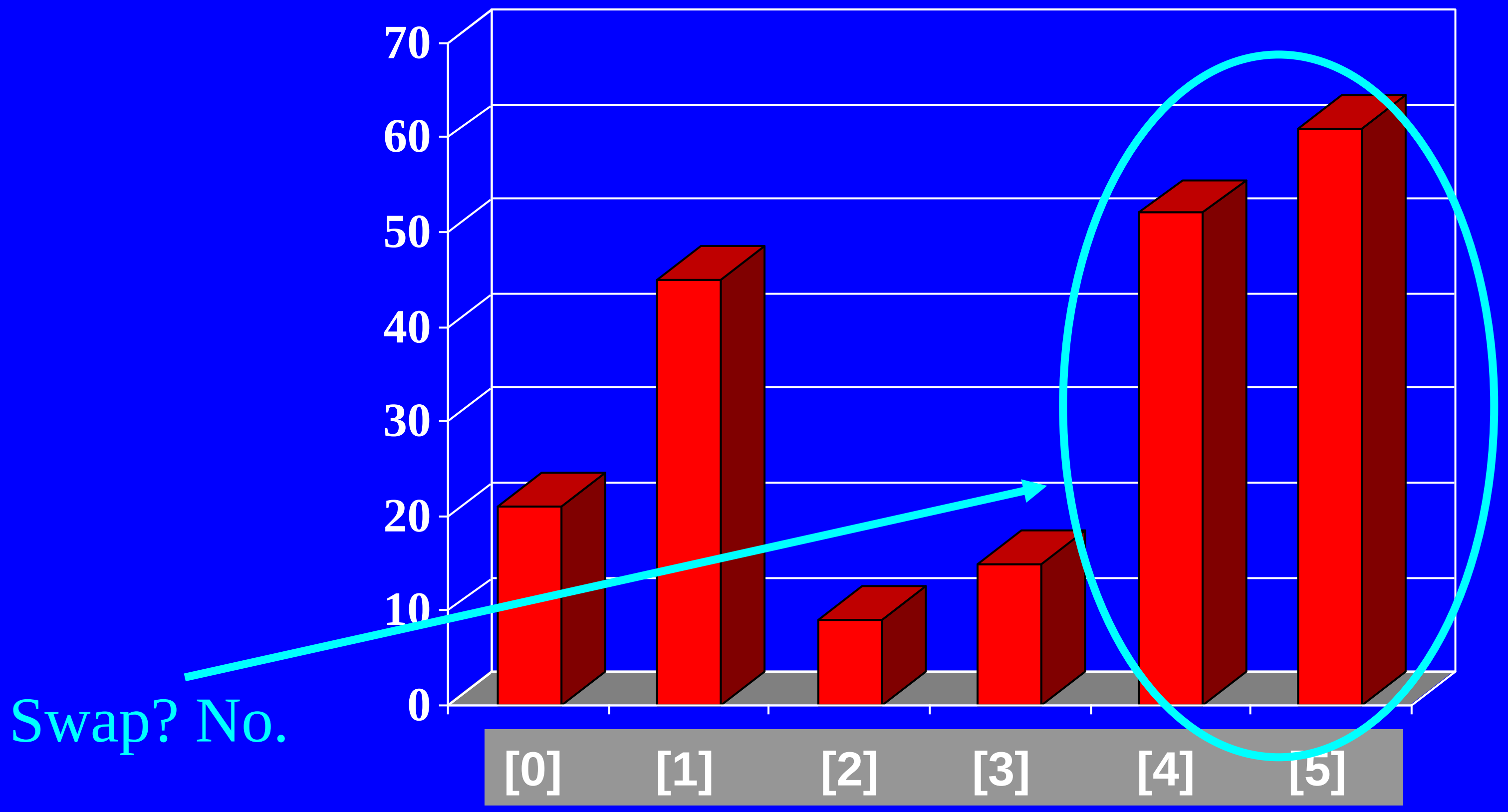
- Repeat.



Swap? Yes.

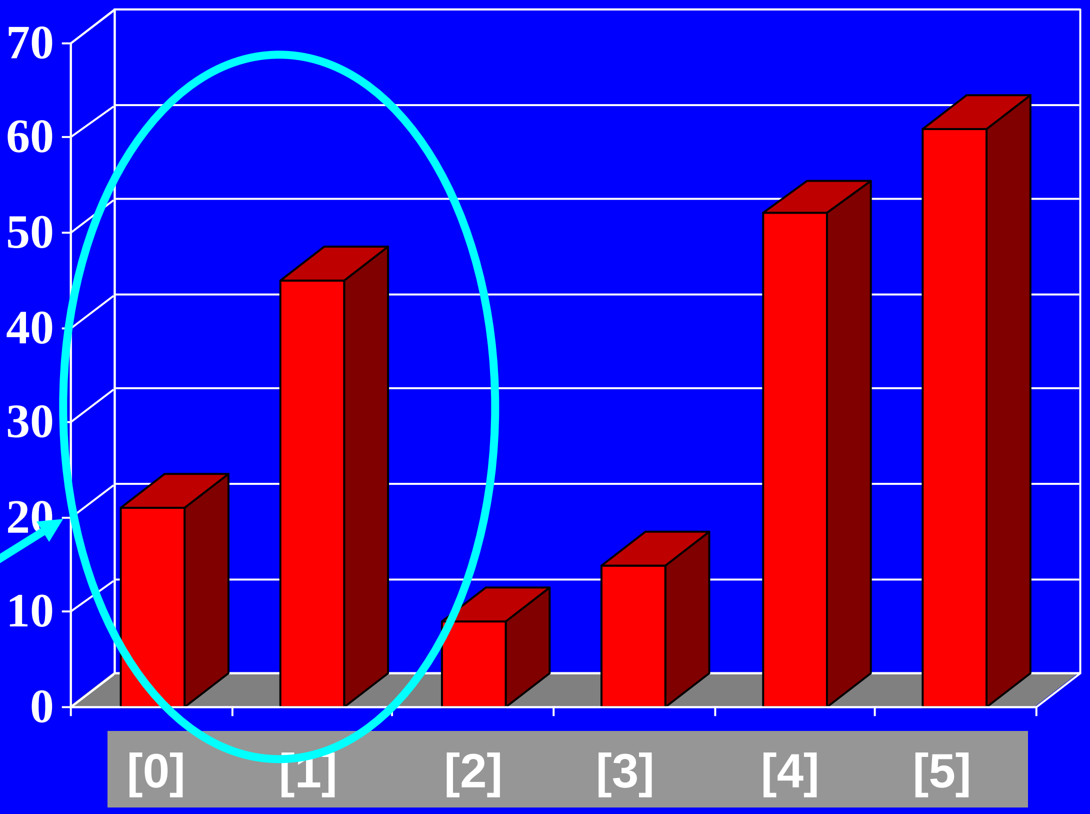
The Bubble Sort Algorithm

- Repeat.



The Bubble Sort Algorithm

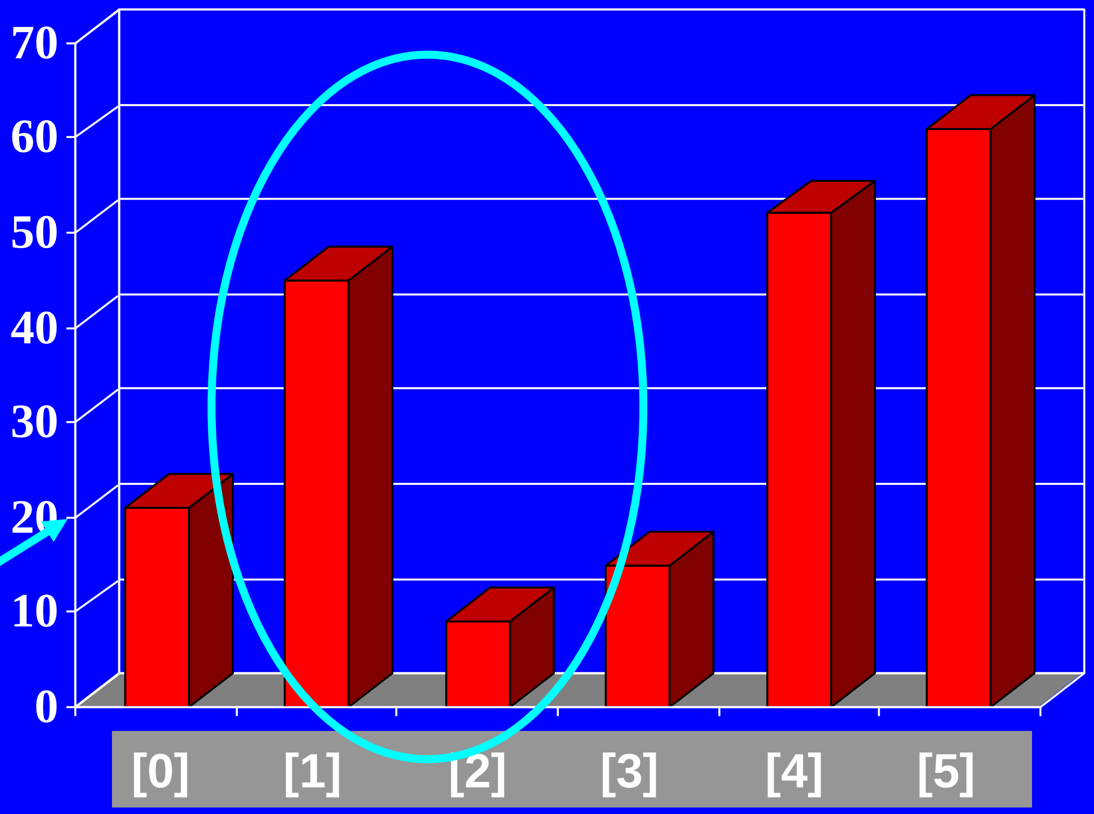
- Loop over array n-1 times, swapping pairs of entries as needed.



Swap? No.

The Bubble Sort Algorithm

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

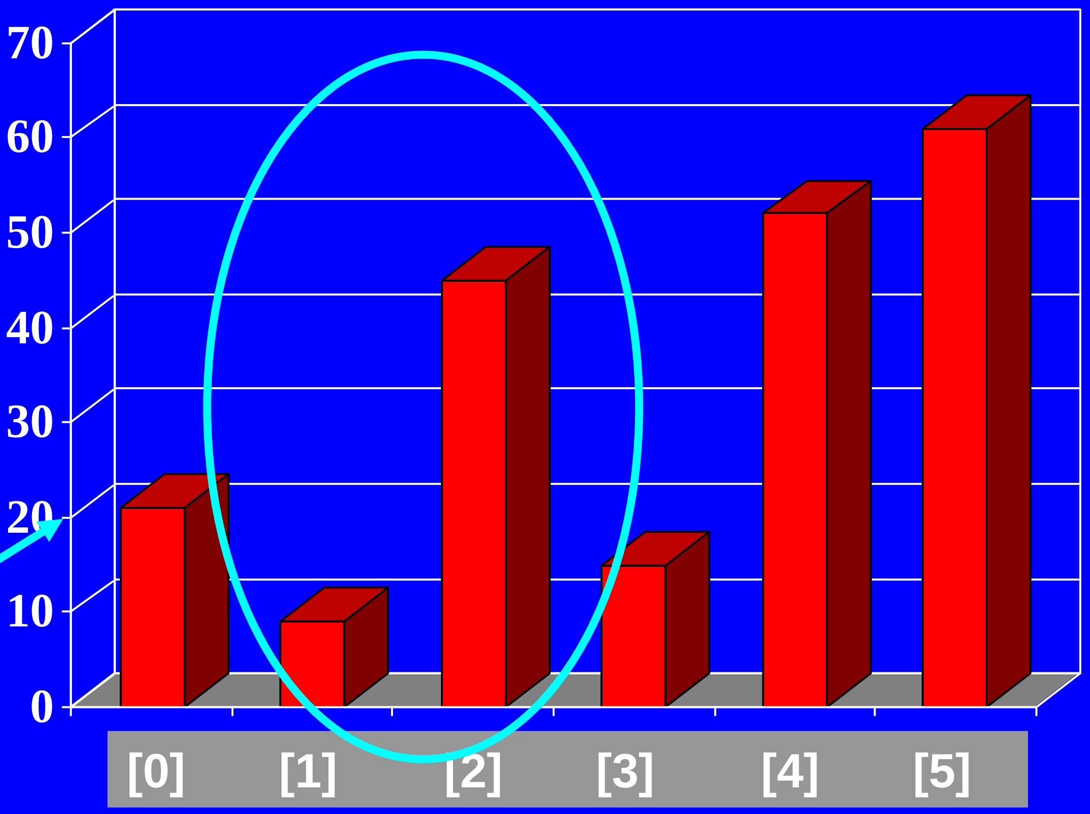


Swap? Yes.

The Bubble Sort Algorithm

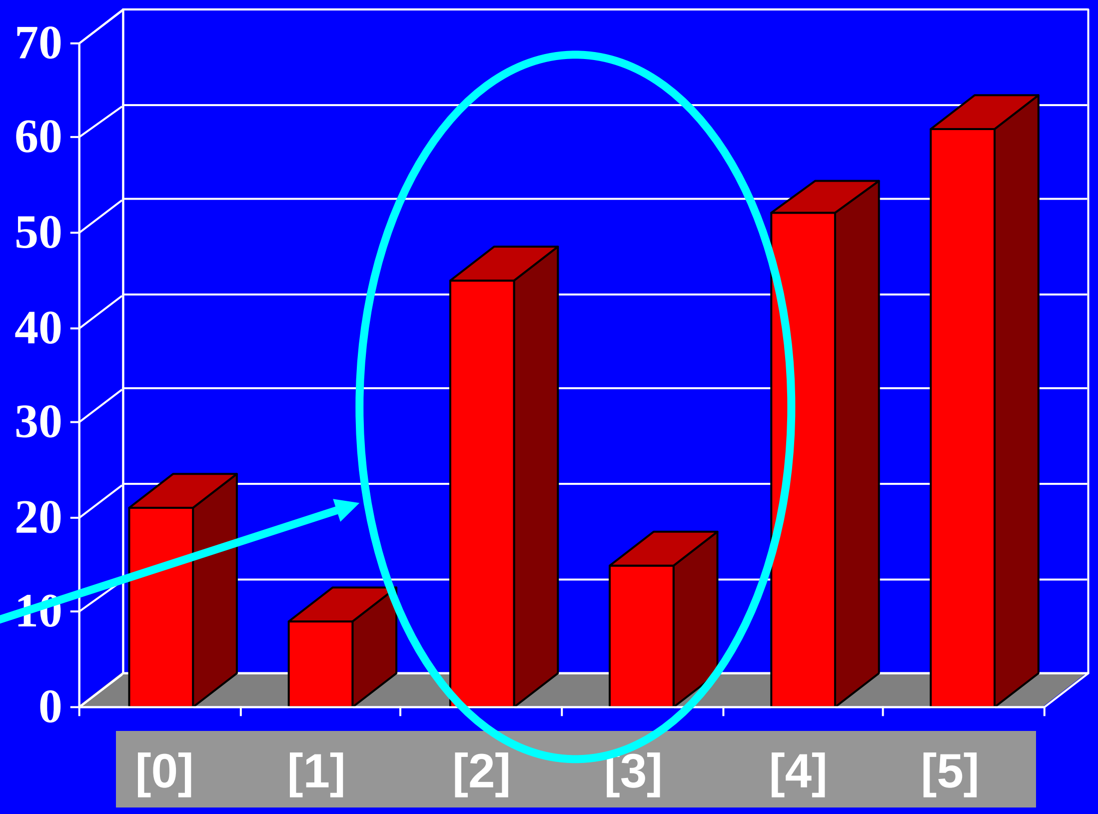
- Loop over array n-1 times, swapping pairs of entries as needed.

Swap? Yes.



The Bubble Sort Algorithm

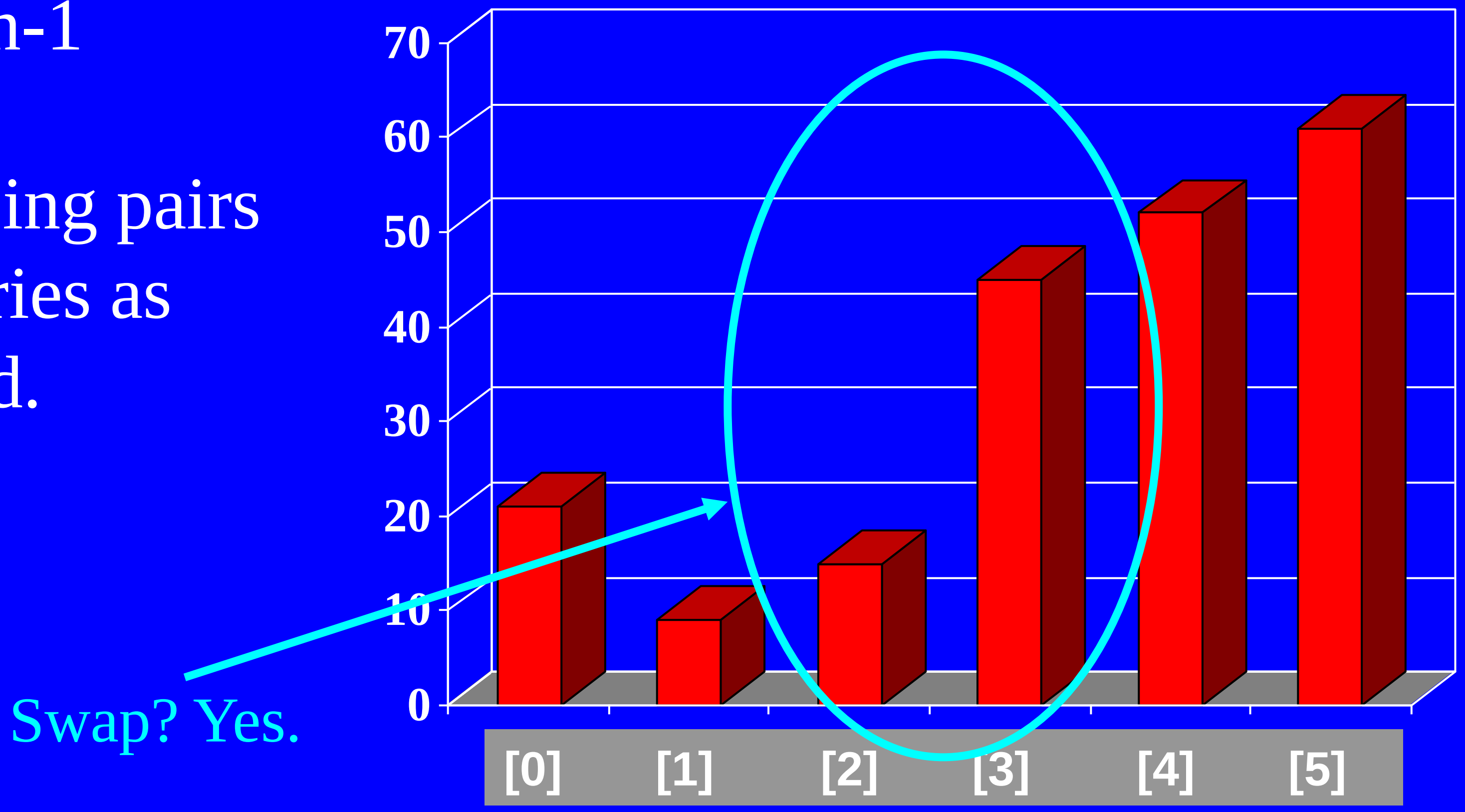
- Loop over array n-1 times, swapping pairs of entries as needed.



Swap? Yes.

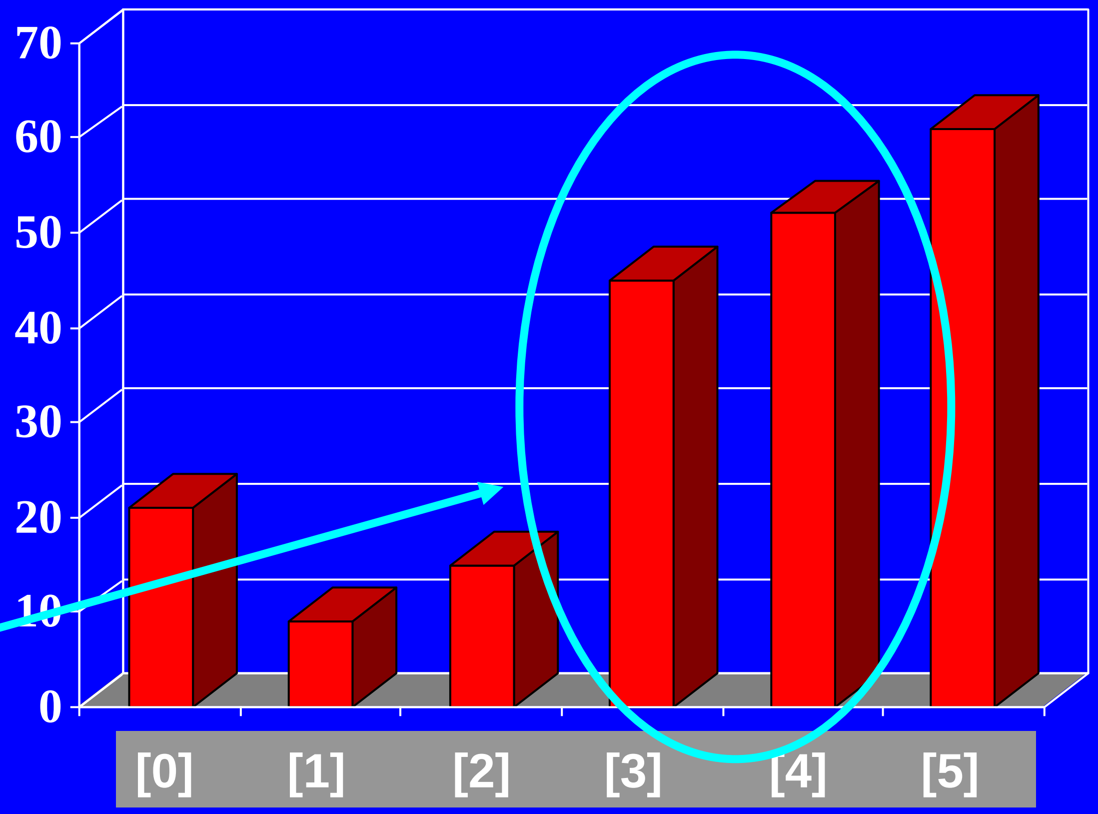
The Bubble Sort Algorithm

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



The Bubble Sort Algorithm

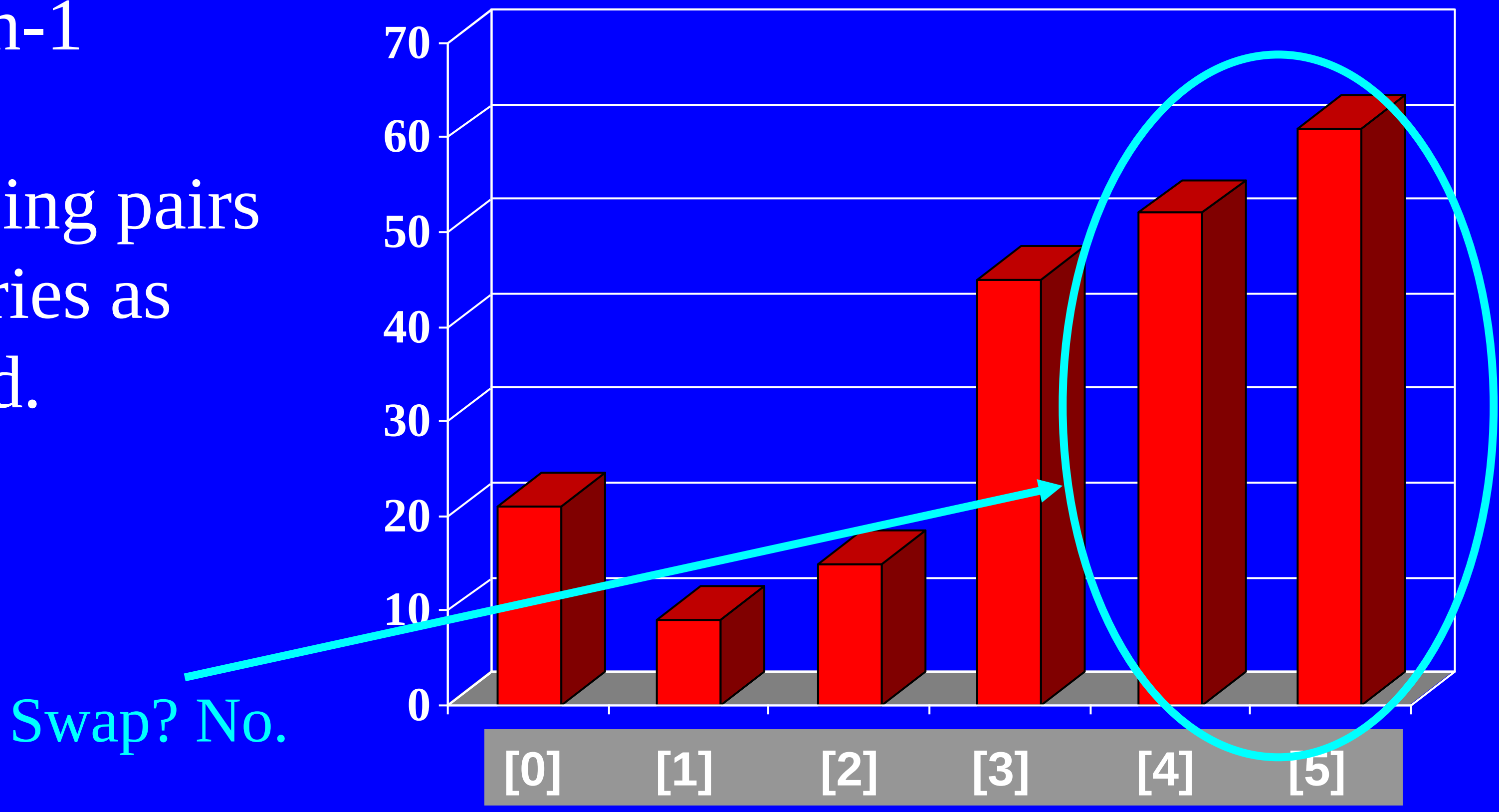
- Loop over array n-1 times, swapping pairs of entries as needed.



Swap? No.

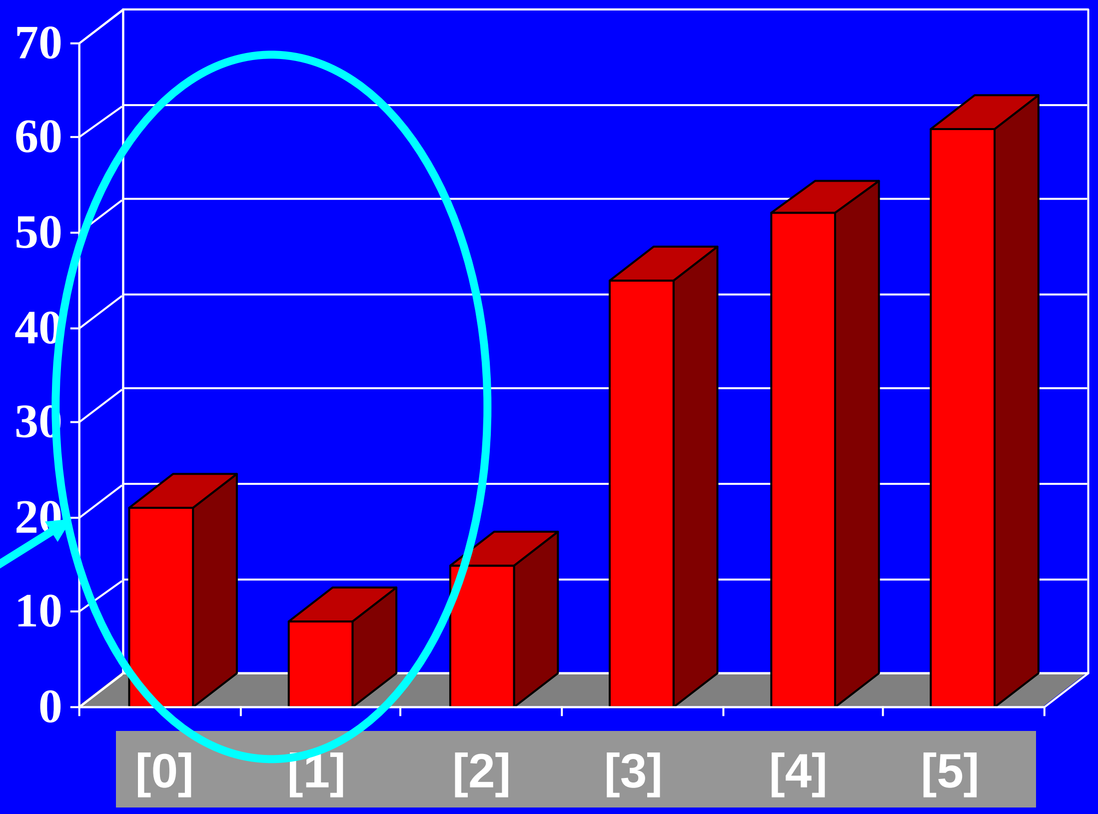
The Bubble Sort Algorithm

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



The Bubble Sort Algorithm

- Continue looping, until done.



Swap? Yes.

```java
public static void bubblesort(int a[],int n)
{
    for(int i=0;i<n-1;i++)
    {
        for(int j=0;j<n-1;j++)
        {
            if(a[j]>a[j+1])
            {
                int t=a[j];
                a[j]=a[j+1];
                a[j+1]=t;
            }
        }
    }
```

```
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;
            }
    }
}
```

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.

Quicksort

Introduction

- Fastest known sorting algorithm in practice
- Average case: O(N log N)
- Worst case: $O(N^2)$
  - But, the worst case seldom happens.
- Another divide-and-conquer recursive algorithm, like mergesort

Quicksort

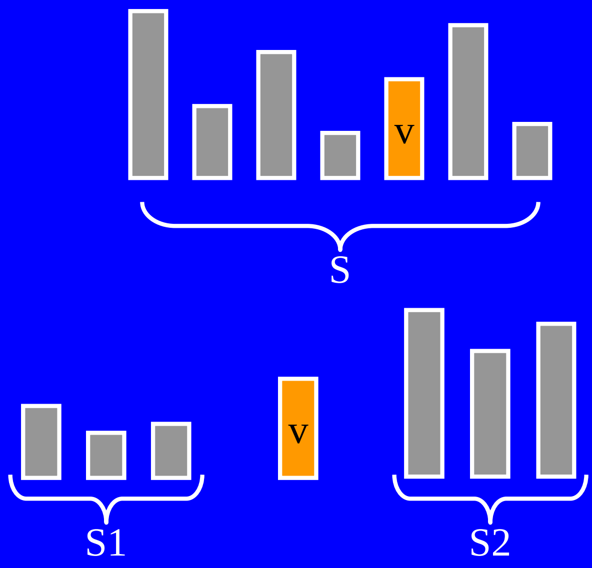
- Divide step:
  - Pick any element (***pivot***) v in S
  - Partition S – {v} into two disjoint groups

    $S1 = \{x \in S - \{v\} \mid x <= v\}$

    $S2 = \{x \in S - \{v\} \mid x \geq v\}$

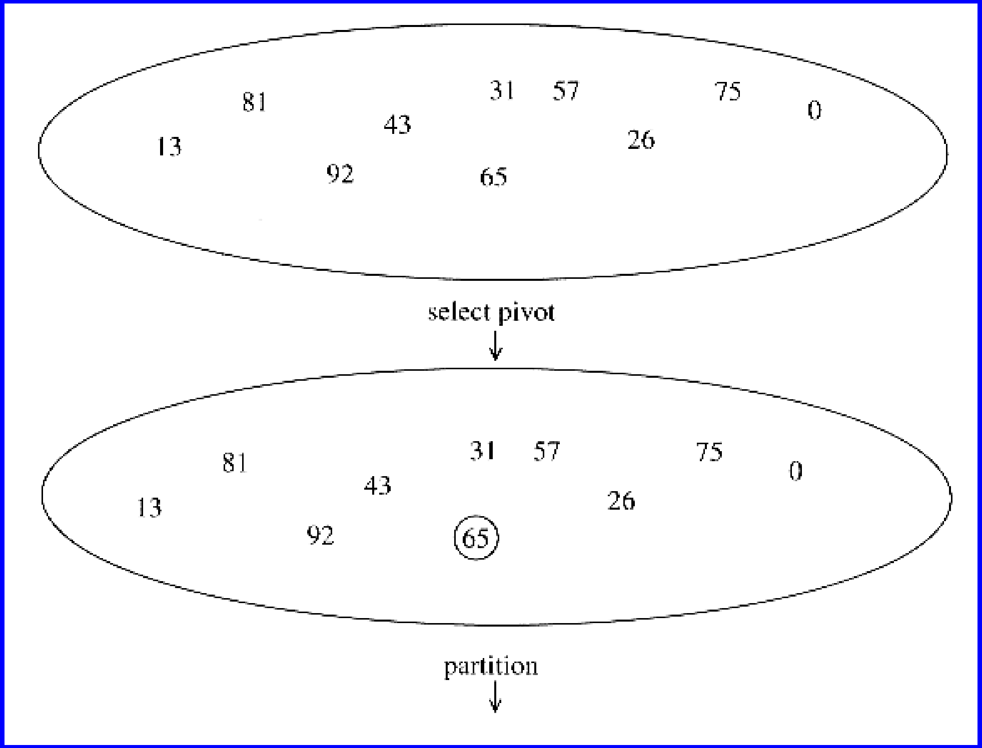
- Conquer step: recursively sort S1 and S2

- Combine step: the sorted S1 (by the time returned from recursion), followed by v, followed by the sorted S2 (i.e., nothing extra needs to be done)

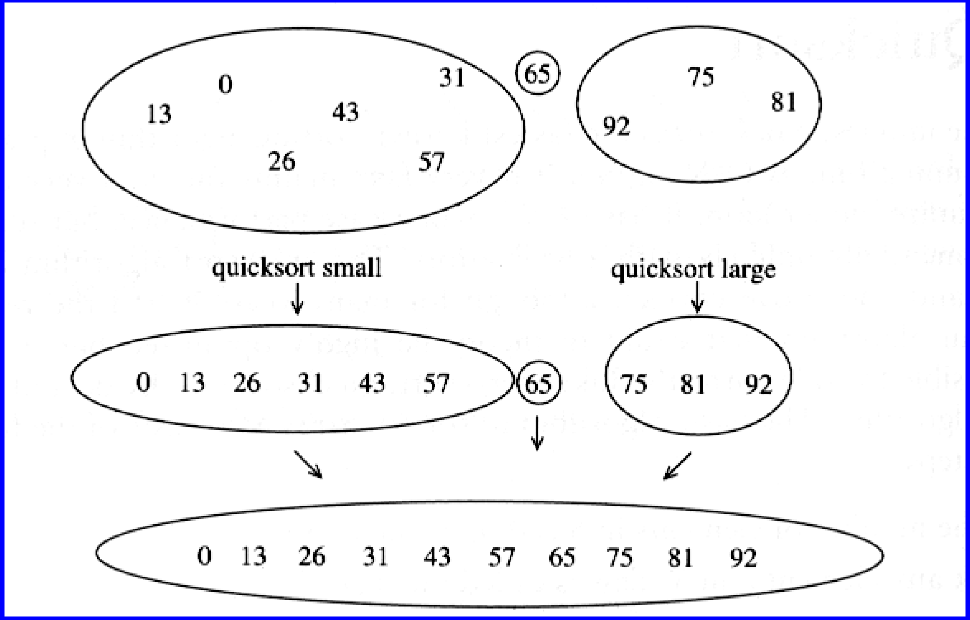


To simplify, we may assume that we don't have repetitive elements,
So to ignore the 'equality' case!

Example



81
31
57
75
0
43
13
26
92
65
select pivot
81
31
57
75
0
43
13
26
92
65
partition



0 13 31 43 26 57
65
75 81 92
quicksort small
quicksort large
0 13 26 31 43 57
65
75 81 92
0 13 26 31 43 57 65 75 81 92

Pseudo-code

```
Input: an array a[left, right]

QuickSort (a, left, right) {
     if (left < right) {
          pivot = Partition (a, left, right)
          Quicksort (a, left, pivot-1)
          Quicksort (a, pivot+1, right)
     }
}
```

Compare with MergeSort:

```
MergeSort (a, left, right) {
     if (left < right) {
          mid = divide (a, left, right)
          MergeSort (a, left, mid-1)
          MergeSort (a, mid+1, right)
          merge(a, left, mid+1, right)
     }
}
```

Quicksort Algorithm

Given an array of *n* elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element to use as *pivot.*
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

Example

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100

Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100

Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

The sub-arrays are stored in the original data array.

Partitioning loops through, swapping elements below/above pivot.

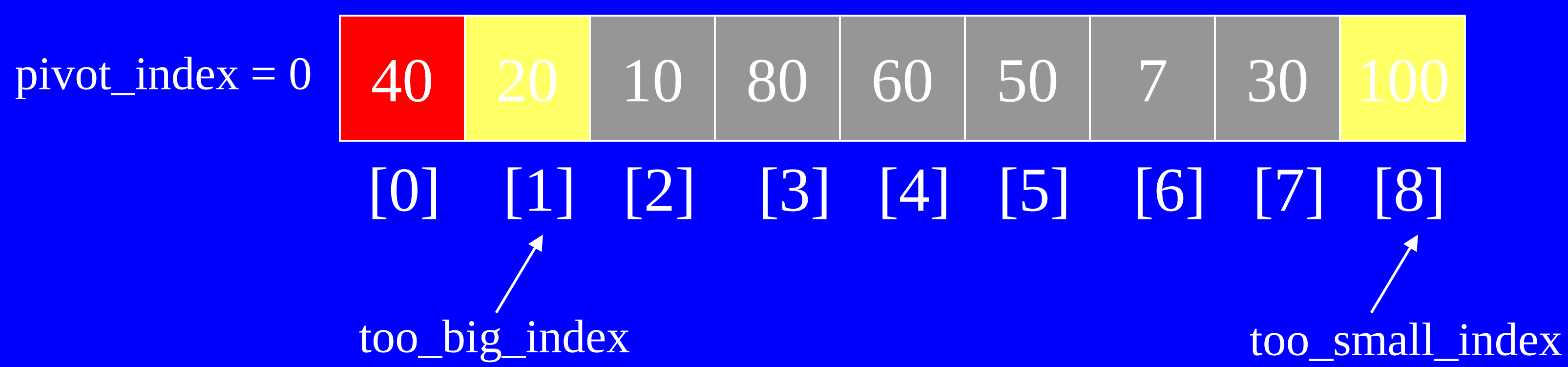
pivot_index = 0

40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index



1. While data[too_big_index] <= data[pivot]
   ++too_big_index

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	80	60	50	7	30	100

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]
    ++too_big_index

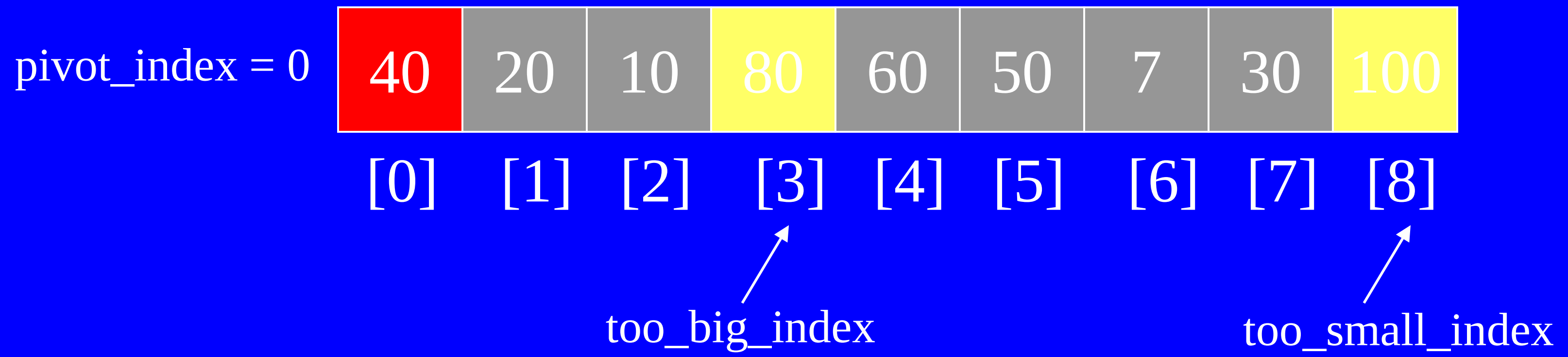
pivot_index = 0

40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index

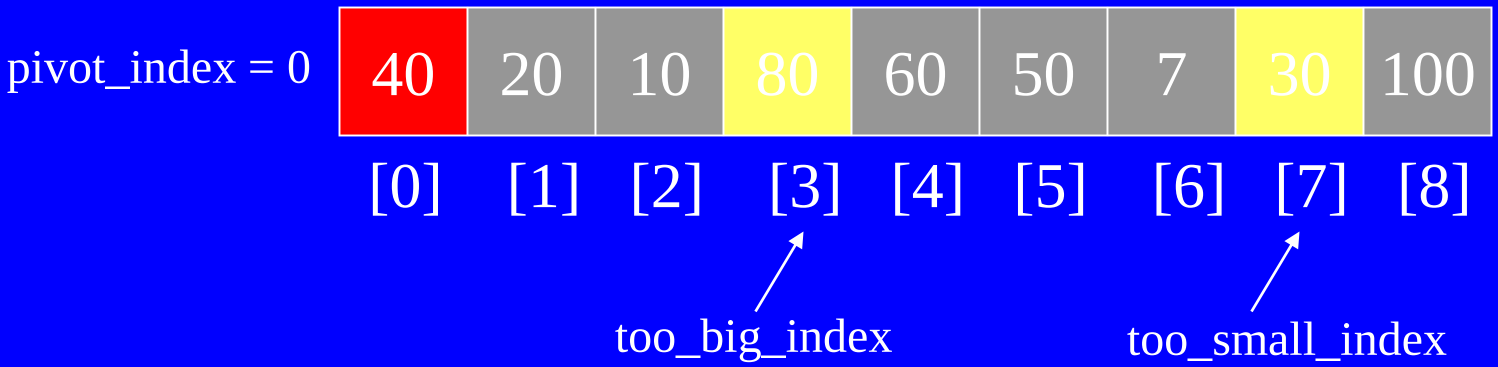


1. While data[too_big_index] <= data[pivot]

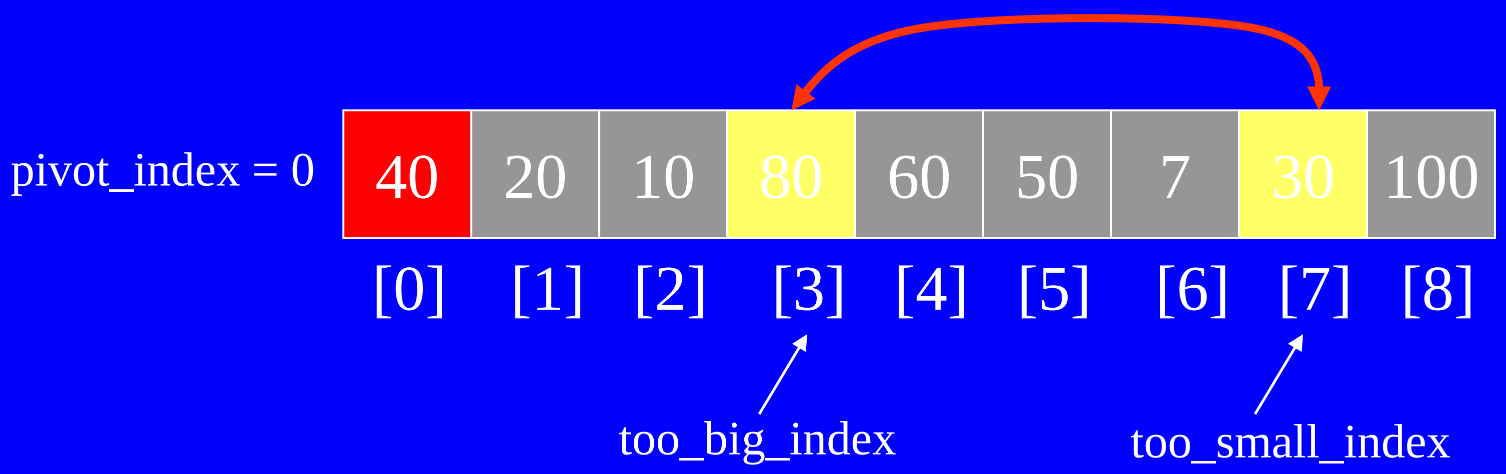
   ++too_big_index

2. While data[too_small_index] > data[pivot]

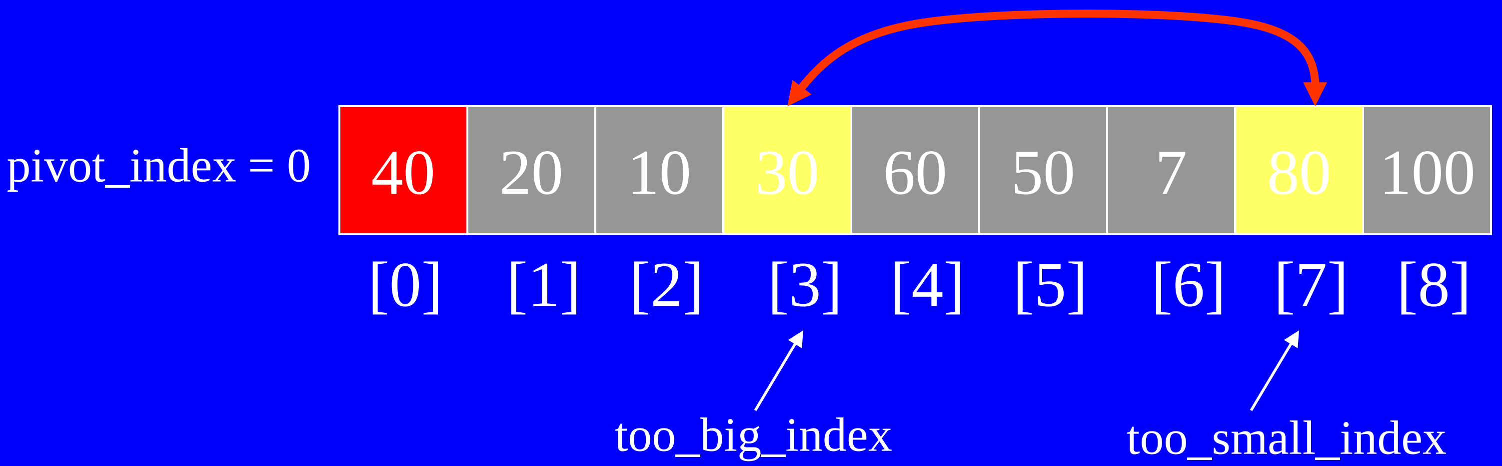
   --too_small_index



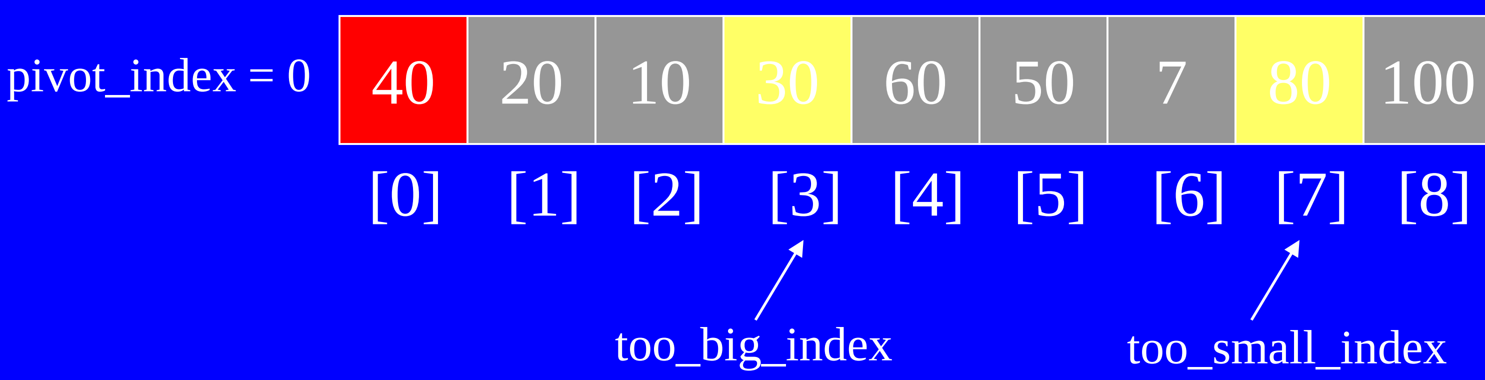
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]

   ++too_big_index

2. While data[too_small_index] > data[pivot]

   --too_small_index

3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]

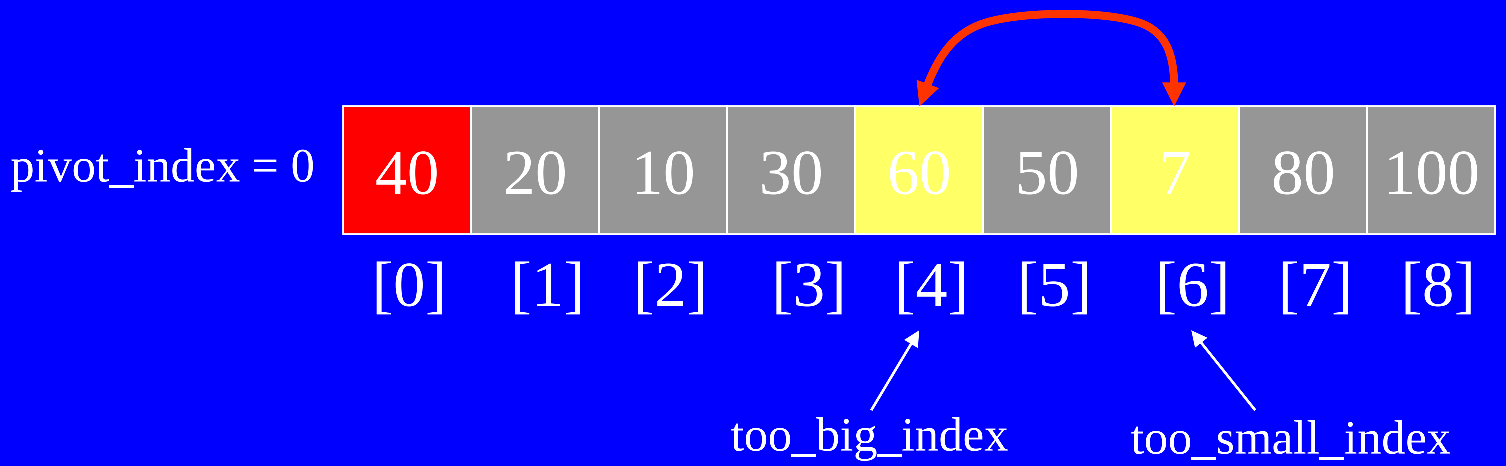
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

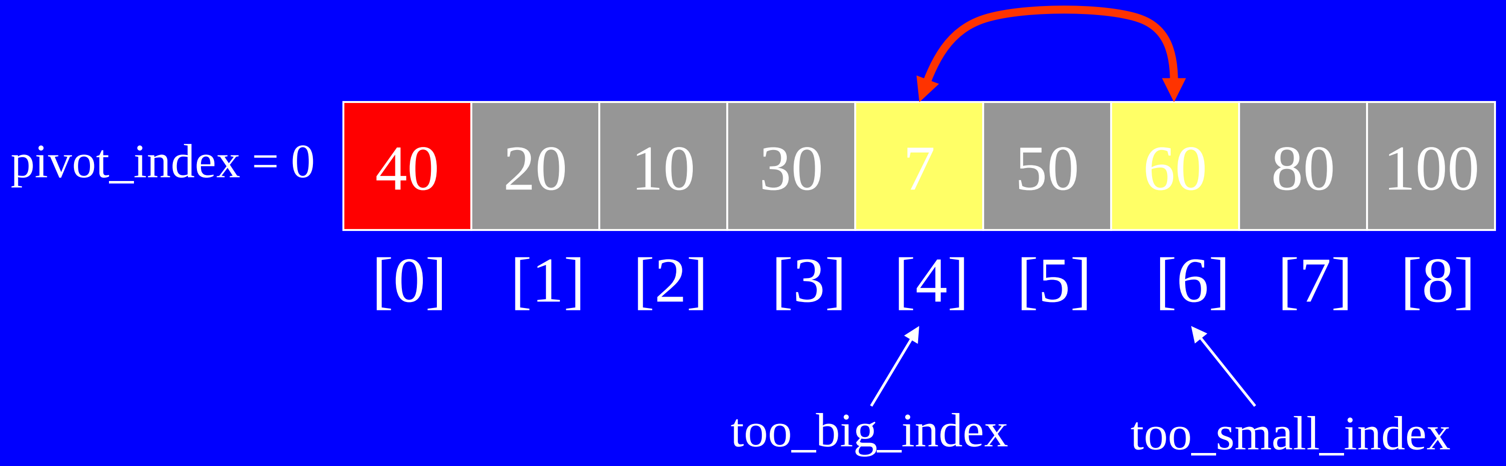
40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
				too_big_index		too_small_index		

1. While data[too_big_index] <= data[pivot]
    ++too_big_index
2. While data[too_small_index] > data[pivot]
    --too_small_index
3. If too_big_index < too_small_index
    swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

too_big_index   too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

too_big_index → [5]

too_small_index → [6]

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
7	20	10	30	40	50	60	80	100

too_big_index too_small_index

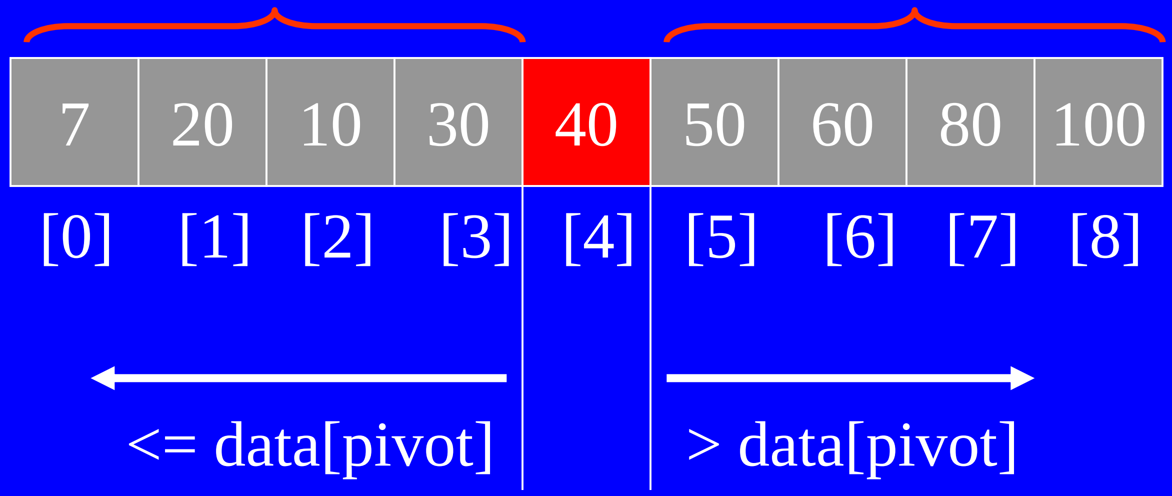
Partition Result

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
7	20	10	30	40	50	60	80	100

<= data[pivot]

> data[pivot]

Recursion: Quicksort Sub-arrays



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition? $O(n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	◈ in-place ◈ slow (good for small inputs)
insertion-sort	$O(n^2)$	◈ in-place ◈ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	◈ in-place, randomized ◈ fastest (good for large inputs)
heap-sort	$O(n \log n)$	◈ in-place ◈ fast (good for large inputs)
merge-sort	$O(n \log n)$	◈ sequential data access ◈ fast (good for huge inputs)