

Data Structures and Algorithms (CEN3016)

Dr. Muhammad Umair Khan

Assistant Professor

Department of Computer Engineering

National University of Technology

Sorting

- A fundamental operation in computer science
- Task of rearranging data in an order such as
 - Ascending
 - Descending
- Data may be of any type like numeric, alphabetical or alphanumeric
- It also refers to rearranging a set of records based on their key values when the records are stored in a file
- Sorting task arises more frequently in the world of data manipulation

Sorting

- Sorting and searching frequently apply to a file of records
- Each record in a file F can contain many fields but there may be one particular field whose values uniquely determine the records in the file called primary key
- The key or key values are k_1, k_2, \dots
- Sorting the file F usually refers to sorting F with respect to a particular primary key
- Searching in F refers to searching for a record with a given key value

Sorting

- Let A be a list of n elements in memory $\square A_1, A_2, \dots, A_n$
- Sorting refers to the operations of rearranging the contents of A so that they are increasing in order numerically or lexicographically so that $\square A_1 \leq A_2 \leq A_3 \leq \dots \leq A_n$
- Since A has n elements, there are $n!$ ways that contents can appear in A
- These ways correspond precisely to the $n!$ permutations of $1, 2, \dots, n$
- Accordingly each sorting algorithms must take care of these $n!$ possibilities

Basic Terminology

- Internal sort

- When a set of data to be sorted is small enough such that the entire sorting can be performed in a computer's internal storage (primary memory)

- External sort

- Sorting a large set of data which is stored in low speed computer's external memory such as hard disk, magnetic tape, etc.

- Ascending order

- An arrangement of data if it satisfies “less than or equal to \leq ” relation between two consecutive data

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

Basic Terminology

- Descending order
 - An arrangement of data if it satisfies “greater than or equal to \geq ” relation between two consecutive data
 - e.g. [9, 8, 7, 6, 5, 4, 3, 2, 1]
- Lexicographic order
 - If the data are in the form of character or string of characters and are arranged in the same order as in dictionary
 - e.g. [ada, bat, cat, mat, max, may, min]
- Collating sequence
 - Ordering for a set of characters that determines whether a character is in higher, lower or same order compared to another
 - e.g. alphanumeric characters are compared according to their ASCII code
 - e.g. [AmaZon, amaZon, amazon, amazon1, amazon2]

Basic Terminology

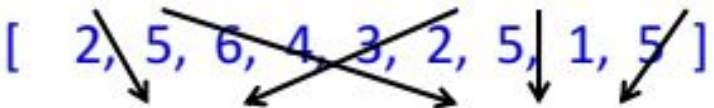
- Random order
 - If a data in a list do not follow any ordering mentioned above, then it is arranged in random order
 - e.g. [8, 6, 5, 9, 3, 1, 4, 7, 2]
 - [may, bat, ada, cat, mat, max, min]
- Swap
 - Swap between two data storages implies the interchange of their contents.
 - e.g. Before swap $A[1] = 11$, $A[5] = 99$
 - After swap $A[1] = 99$, $A[5] = 11$
- Item
 - Is a data or element in the list to be sorted.
 - May be an integer, string of characters, a record etc.
 - Also alternatively termed key, data, element etc.

Basic Terminology

- Stable Sort

- A list of data may contain two or more equal data. If a sorting method maintains the same relative position of their occurrences in the sorted list then it is stable sort

▫ e.g. [2, 5, 6, 4, 3, 2, 5, 1, 5]



▫ [1, 2, 2, 3, 4, 5, 5, 5, 6]

- In Place Sort

- Suppose a set of data to be sorted is stored in an array A
- If a sorting method takes place within the array A only, i.e. without using any other extra storage space
- It is a memory efficient sorting method

Stability

- Stable sorting algorithms maintain the relative order of records with **equal keys**.
 - A **key** is that portion of record which is the basis for the sort
 - it may or may not include all of the record
- If **all keys are different** then this distinction is not necessary.
- But **if there are equal keys**, then a sorting algorithm is stable if whenever there are two records (let's say R and S) with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list.

Stability

- When **equal elements** are **indistinguishable**, such as with integers, or more generally, any data where the entire element is the key, **stability is not an issue**.
- However, assume that the following pairs of numbers are to be sorted by their first component:
 - (4, 2) (3, 7) (3, 1) (5, 6)
 - **Two different results are possible**,
 - one which maintains the relative order of records with equal keys, and one which does not:
 - (3, 7) (3, 1) (4, 2) (5, 6) **(order maintained)**
 - (3, 1) (3, 7) (4, 2) (5, 6) **(order changed)**

Sorting Methods

Bubble Sort
Selection Sort
Insertion Sort

Heap Sort
Merge Sort
Quick Sort

Bubble Sort

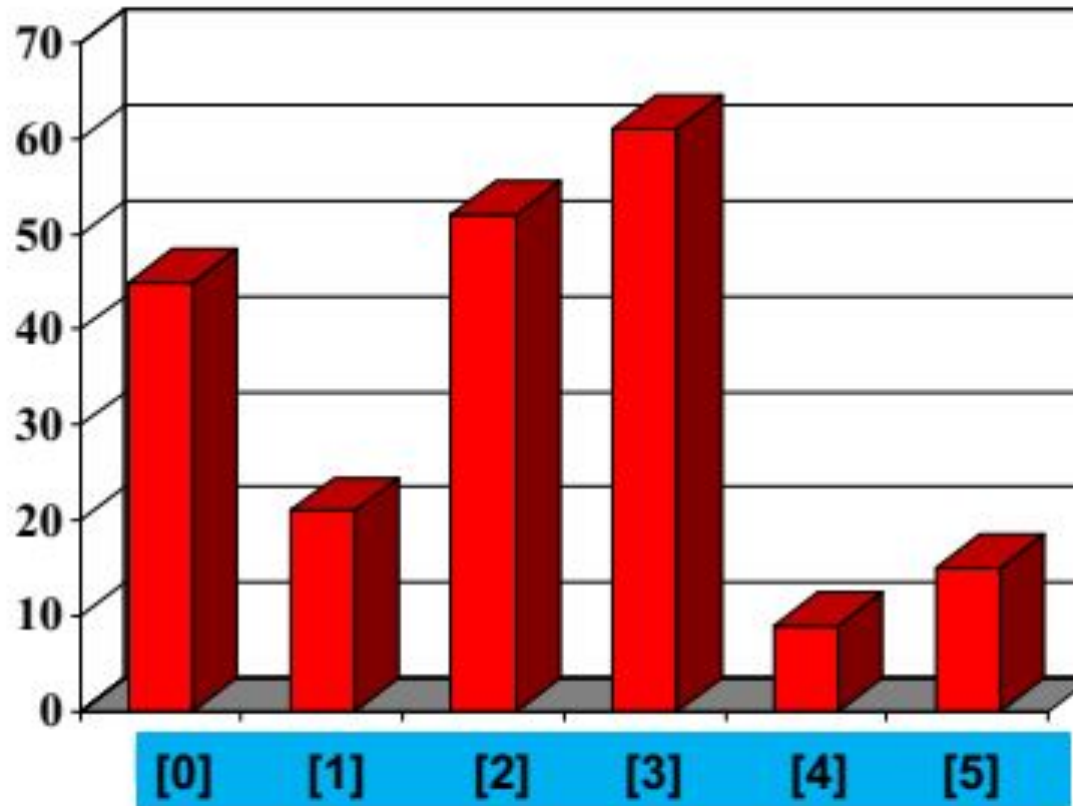
- Sometimes incorrectly referred to as **sinking sort**, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- The algorithm gets its name from the way smaller elements **"bubble"** to the top of the list.
- Because it only uses comparisons to operate on elements, it is a **comparison sort**.

Bubble Sort

- Bubble sort is a **simple sorting algorithm**
- The algorithm starts at the beginning of the data set.
 - It compares the first two elements, and if the first is greater than the second, it swaps them.
 - It continues doing this for each pair of adjacent elements to the end of the data set.
 - It then starts again with the first two elements, repeating until no swaps have occurred on the last pass.
- Note that the largest end gets sorted first, with smaller elements taking longer to move to their correct positions.

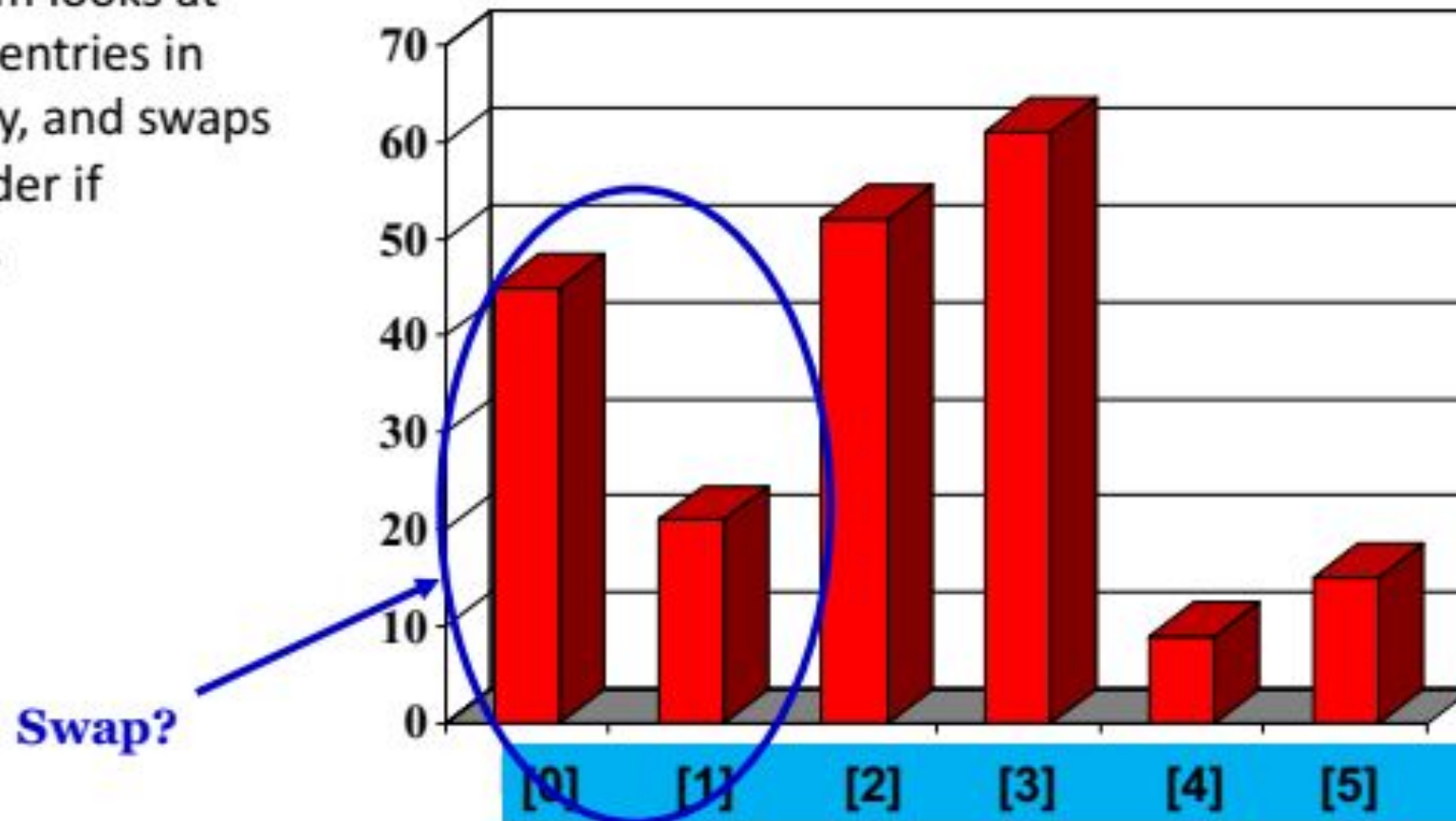
Bubble Sort Algorithm

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



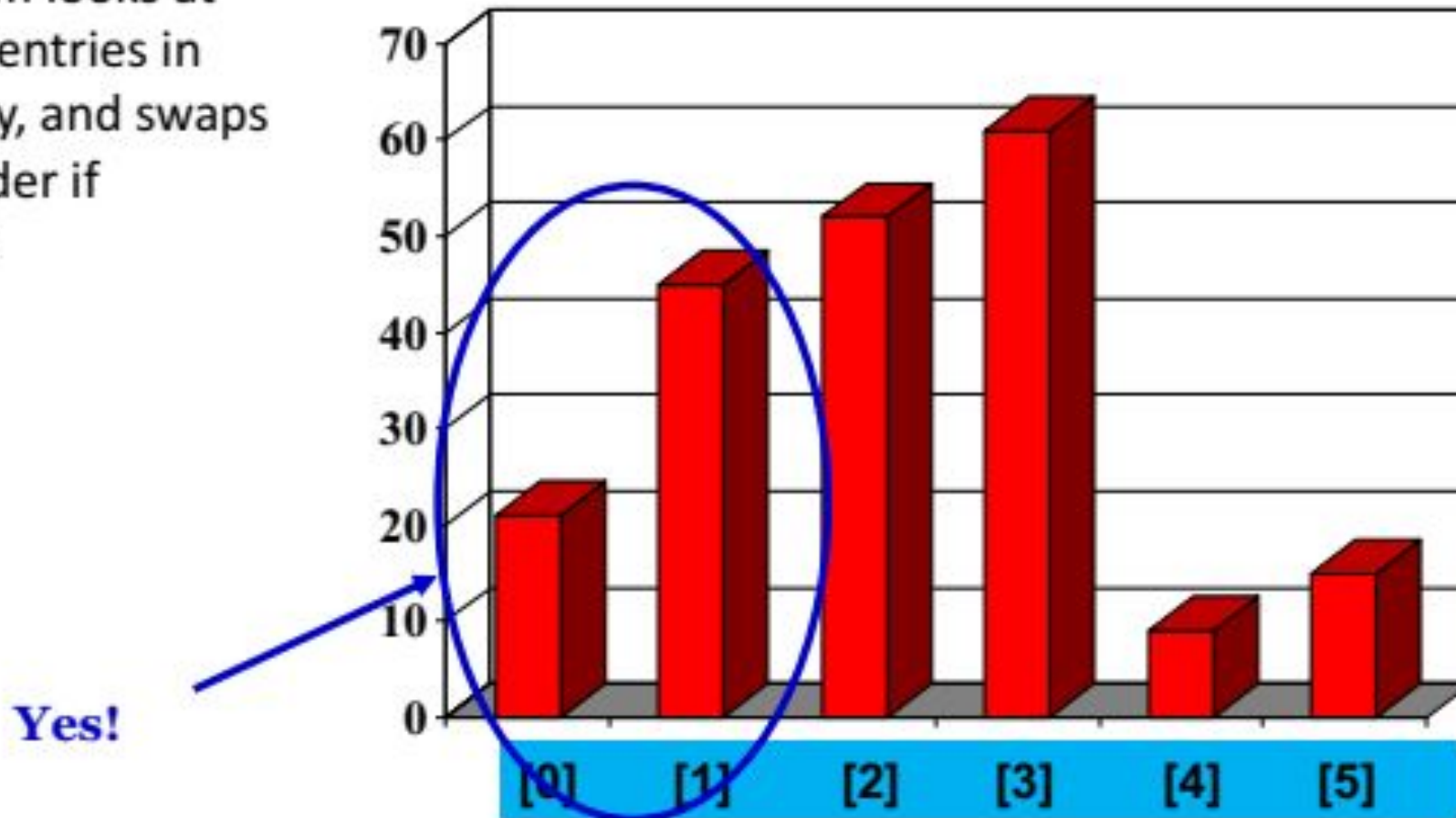
Bubble Sort Algorithm

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



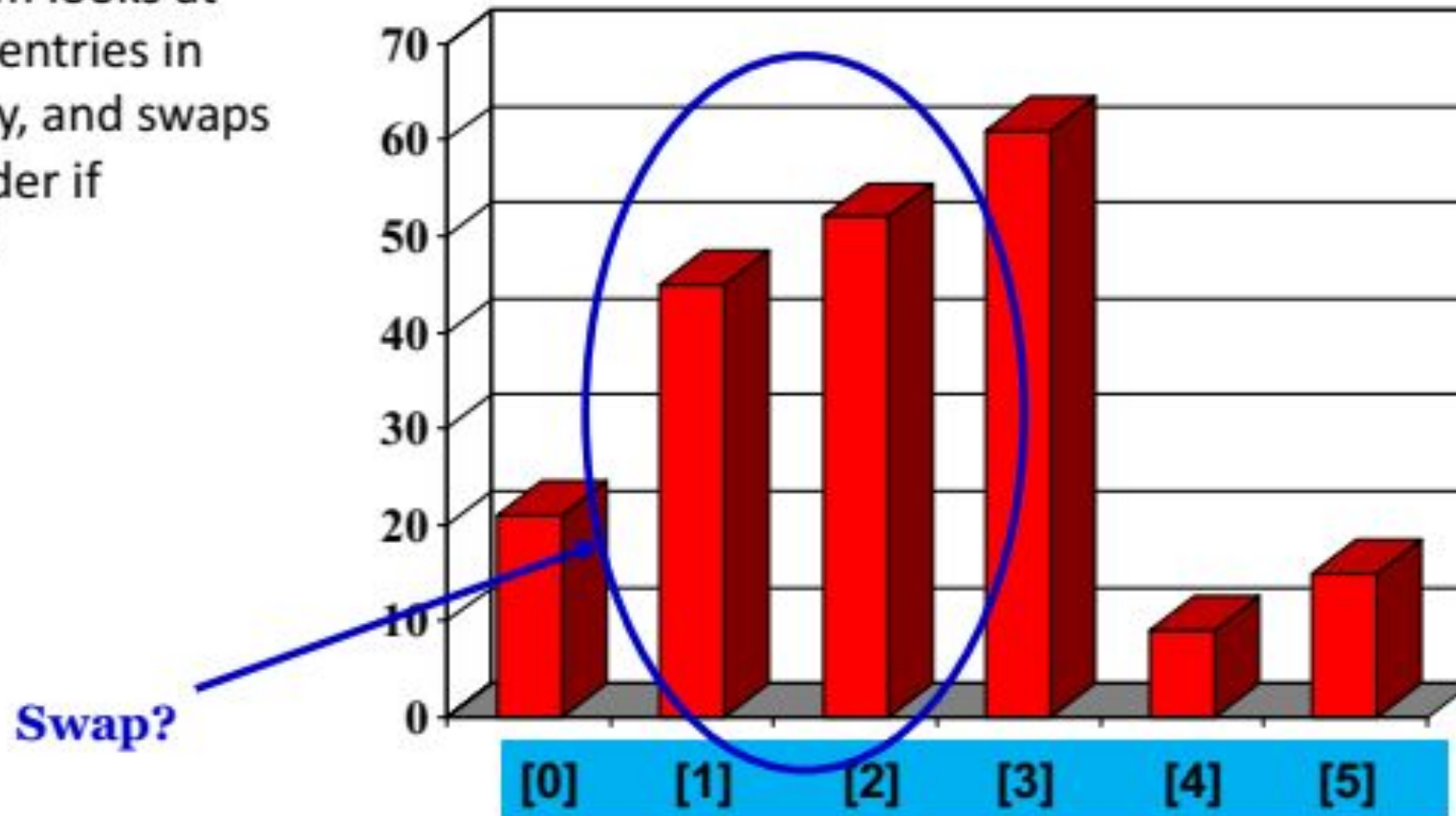
Bubble Sort Algorithm

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



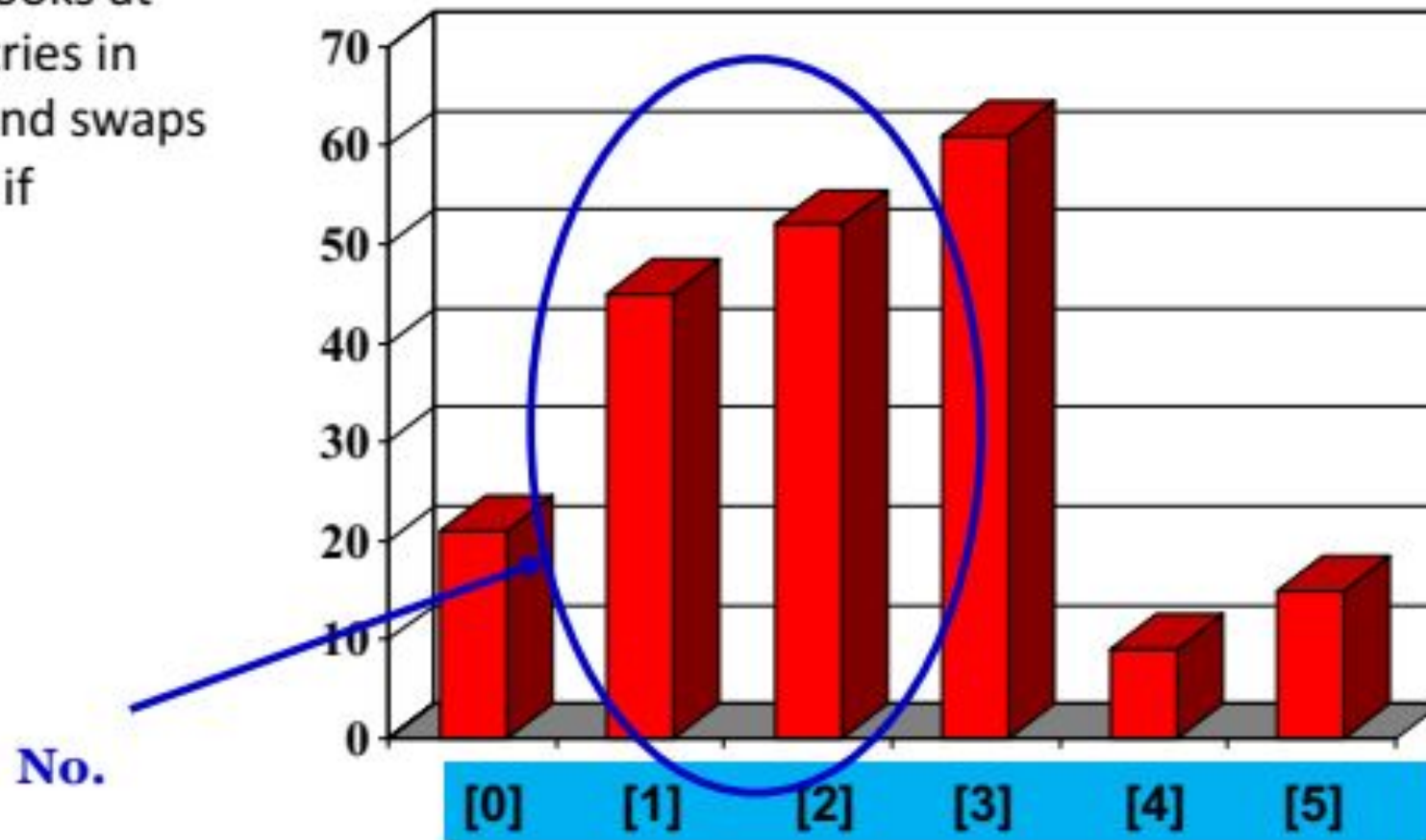
Bubble Sort Algorithm

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



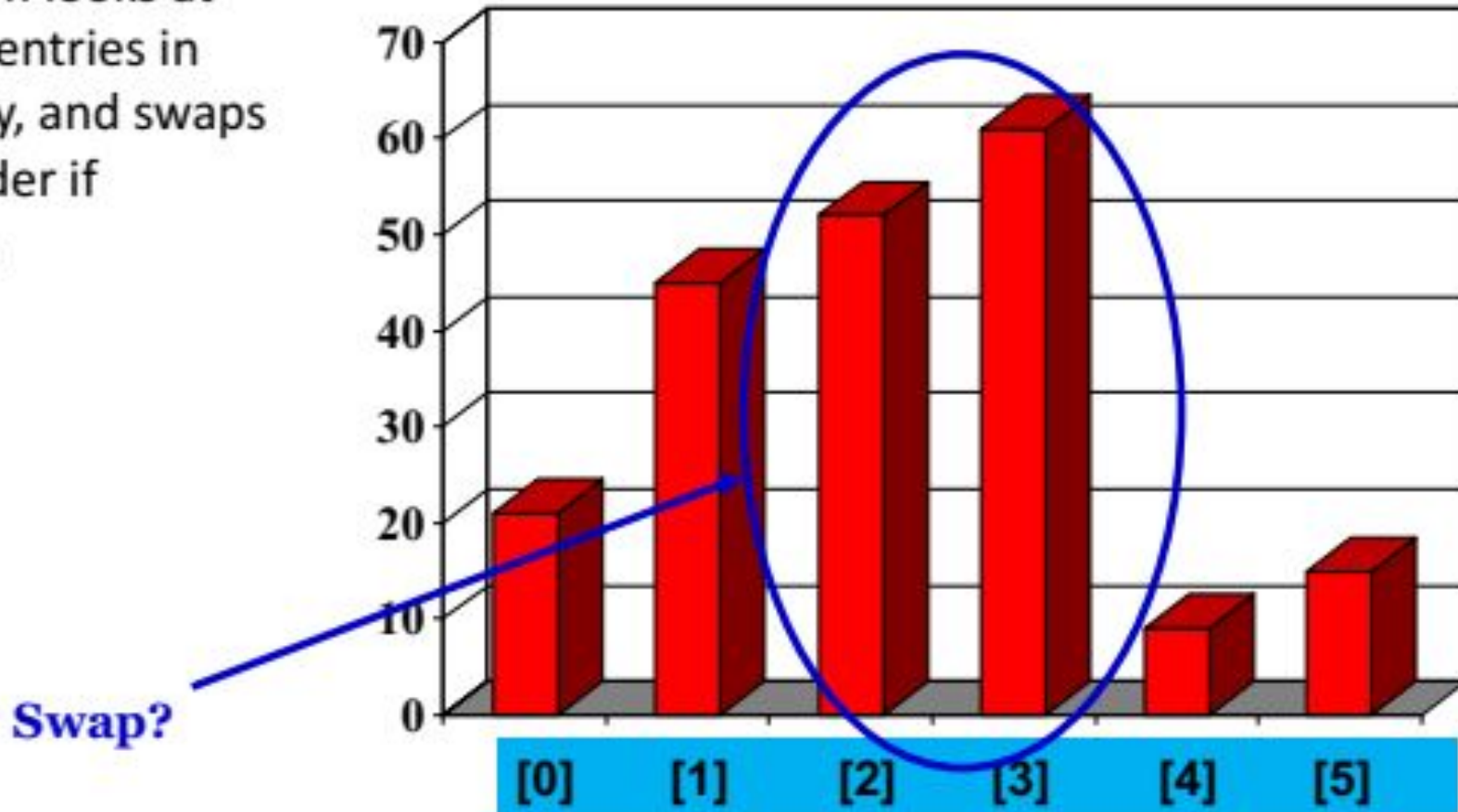
Bubble Sort Algorithm

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



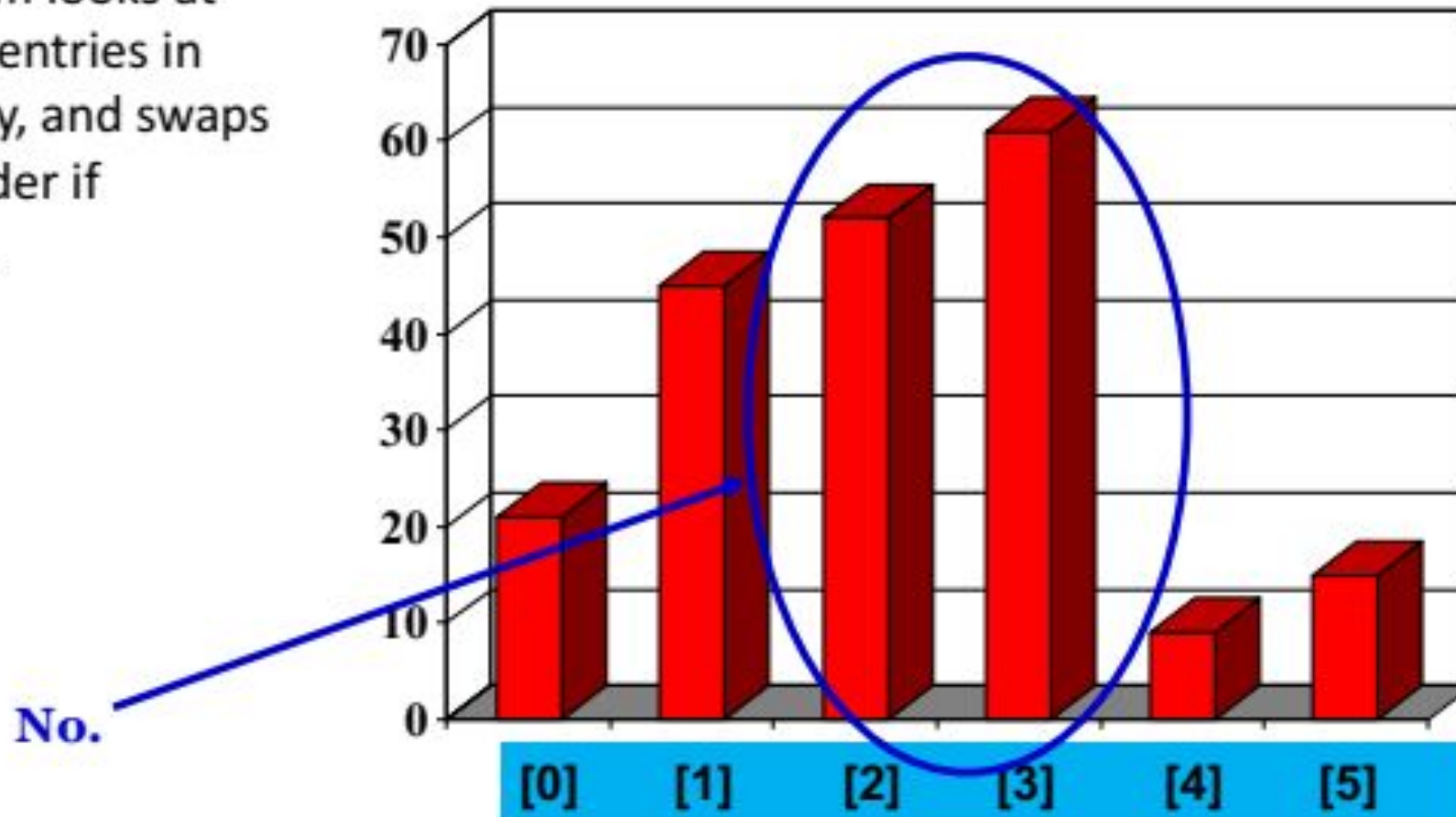
Bubble Sort Algorithm

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



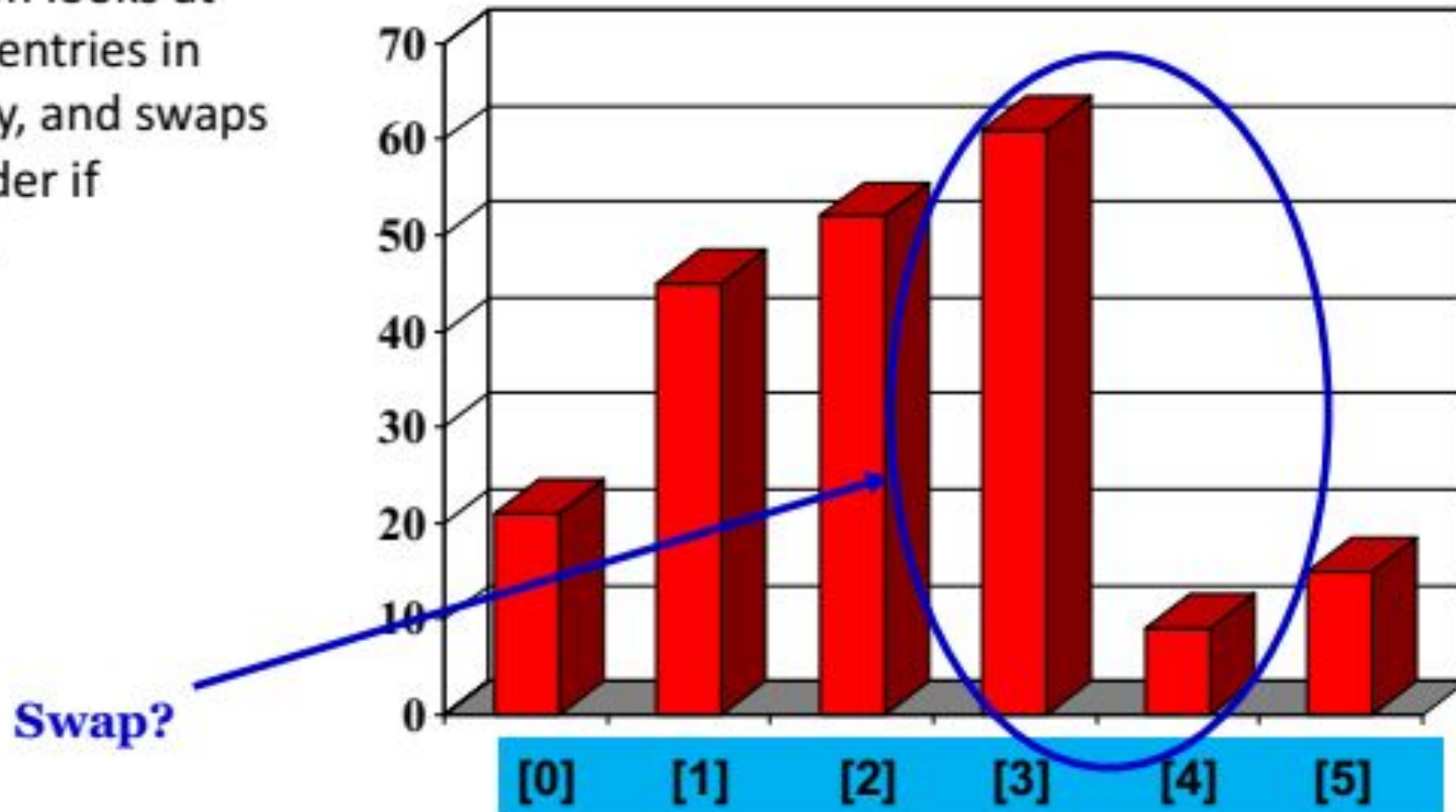
Bubble Sort Algorithm

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



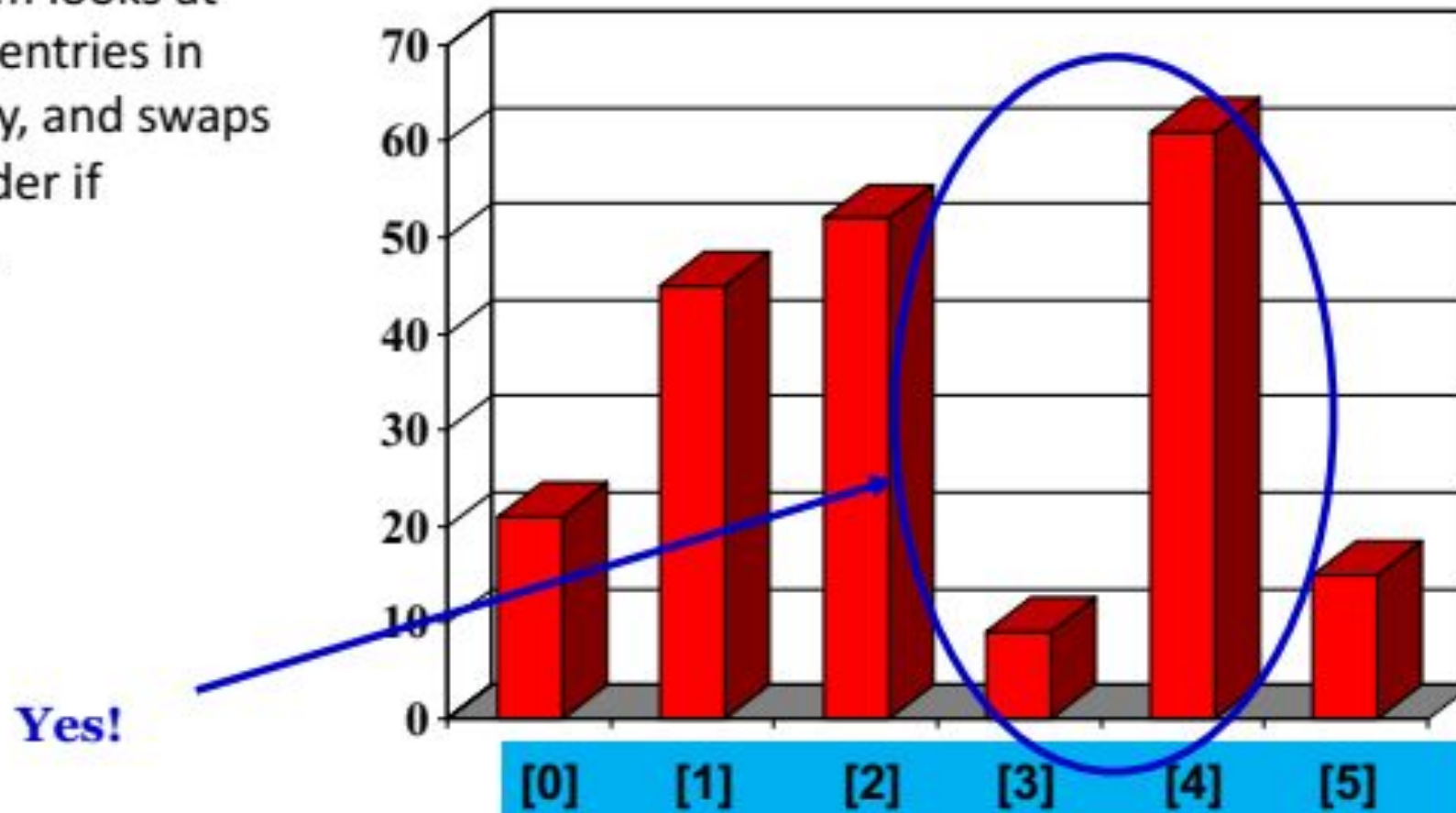
Bubble Sort Algorithm

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



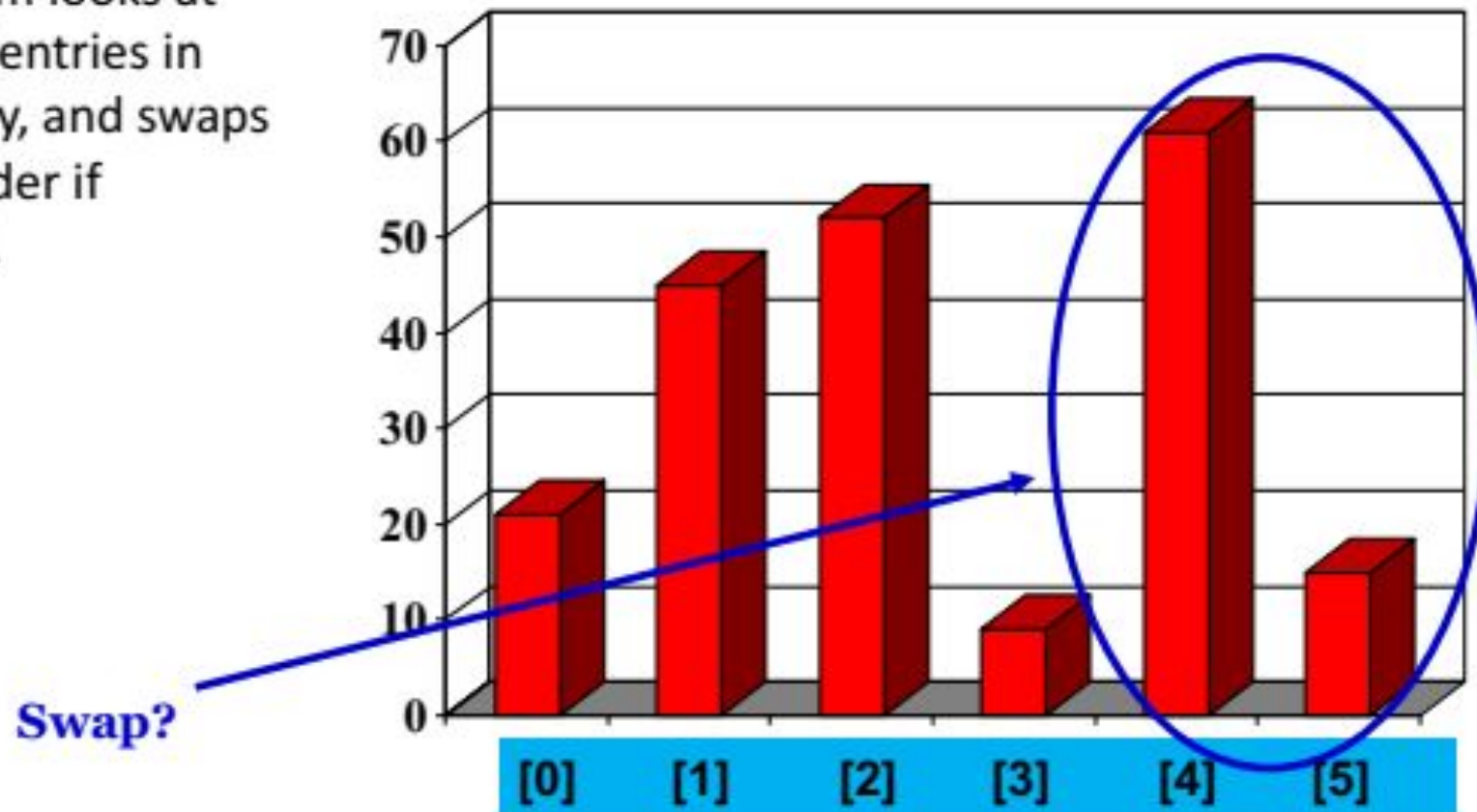
Bubble Sort Algorithm

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



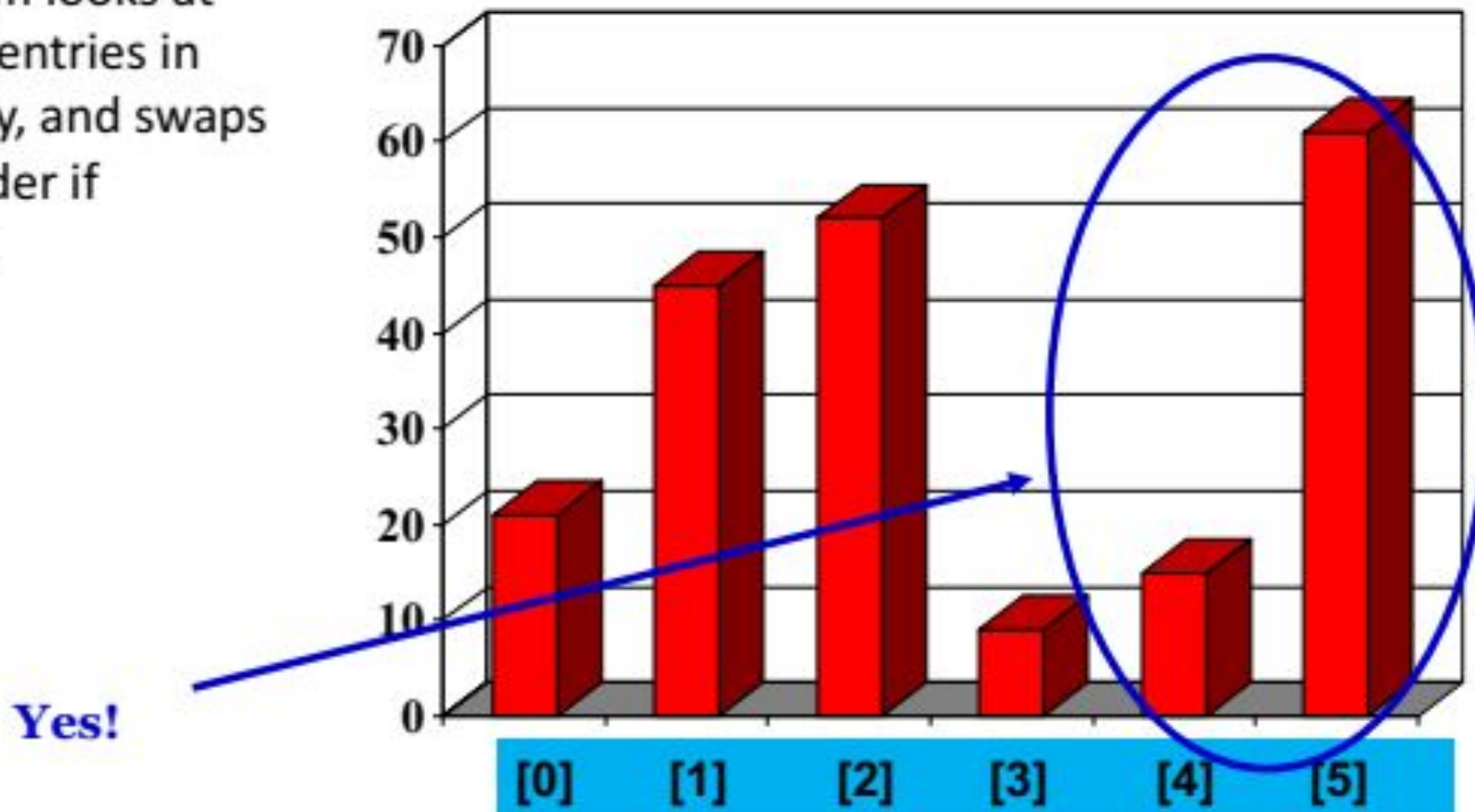
Bubble Sort Algorithm

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



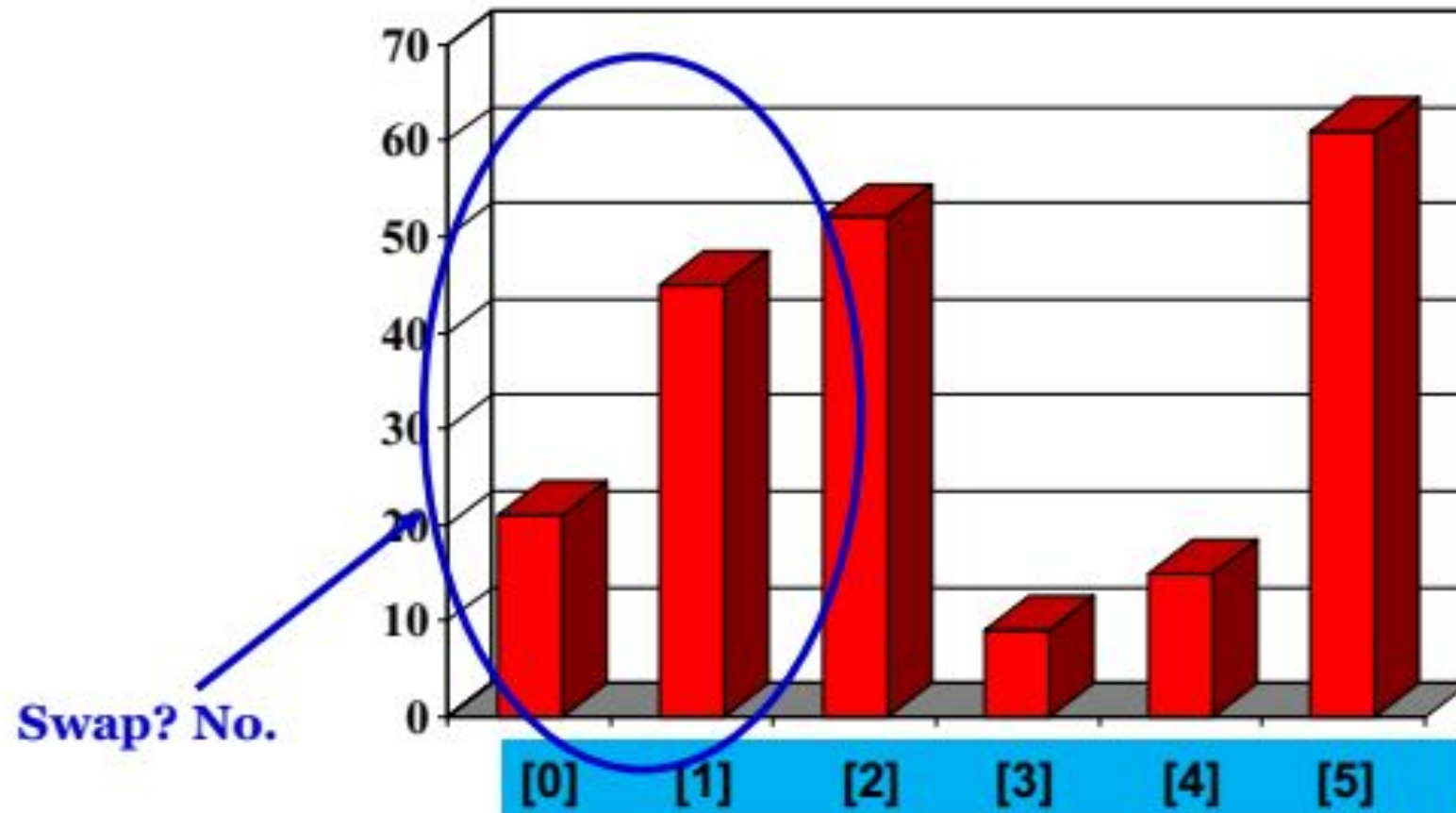
Bubble Sort Algorithm

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



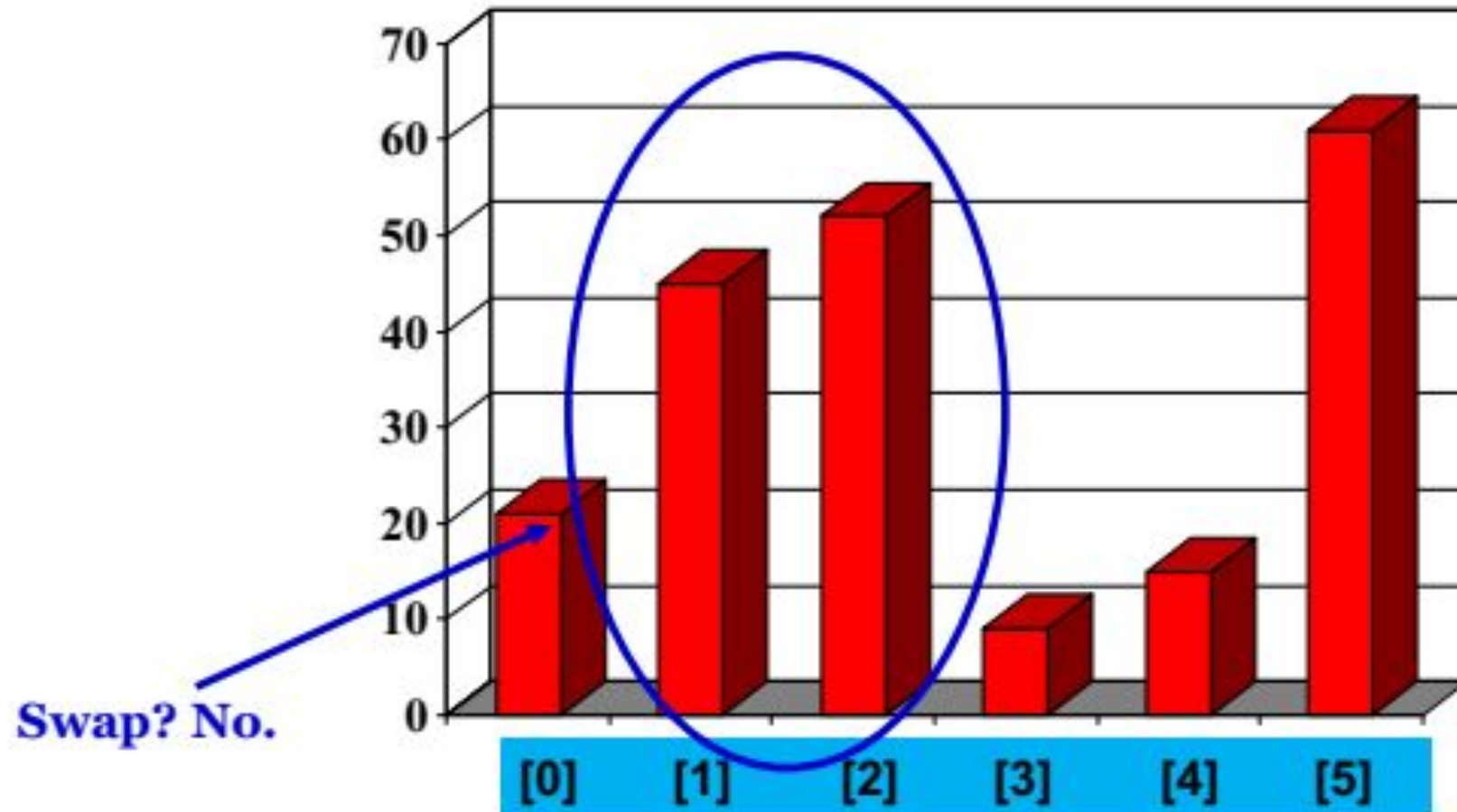
Bubble Sort Algorithm

- Repeat.



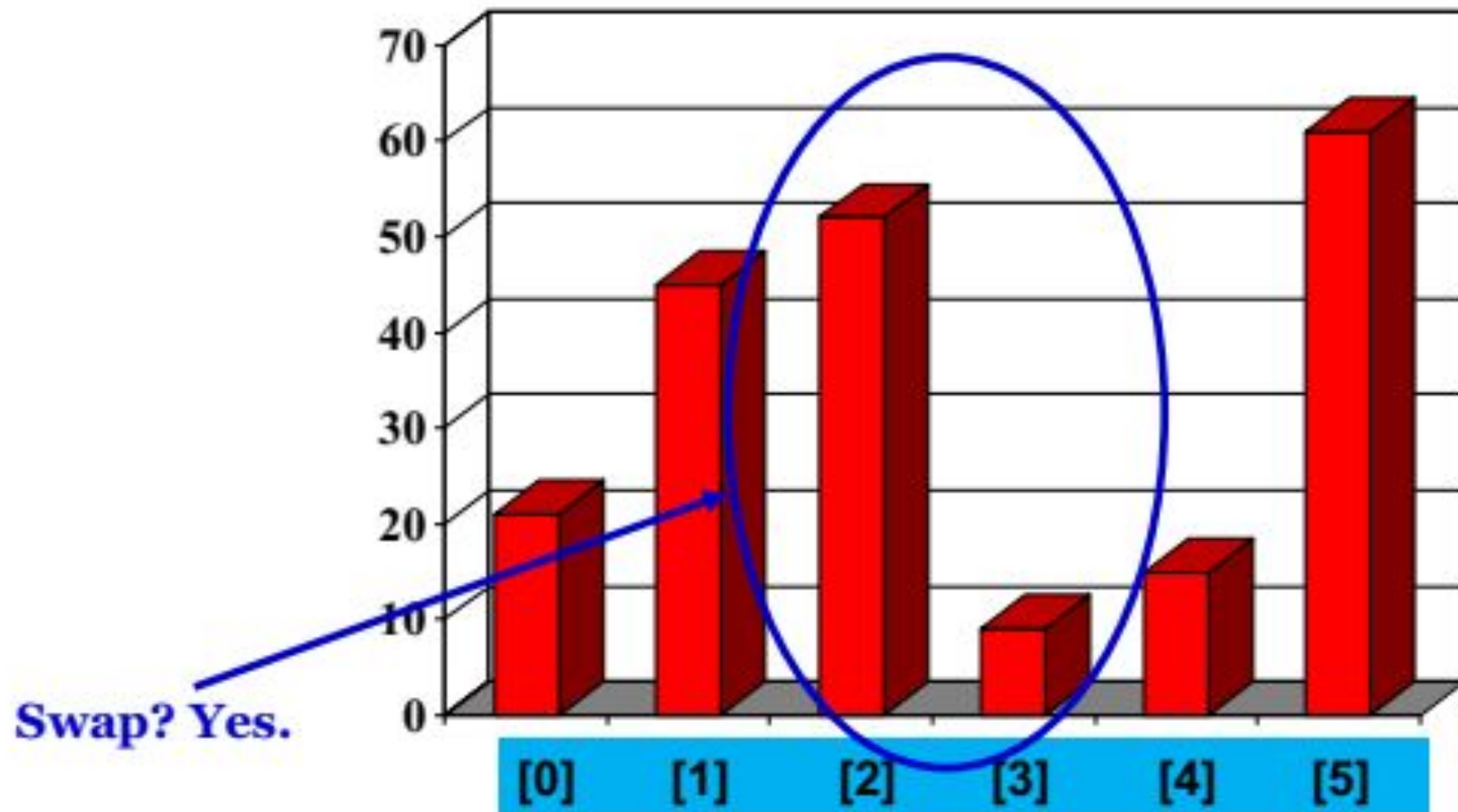
Bubble Sort Algorithm

- Repeat.



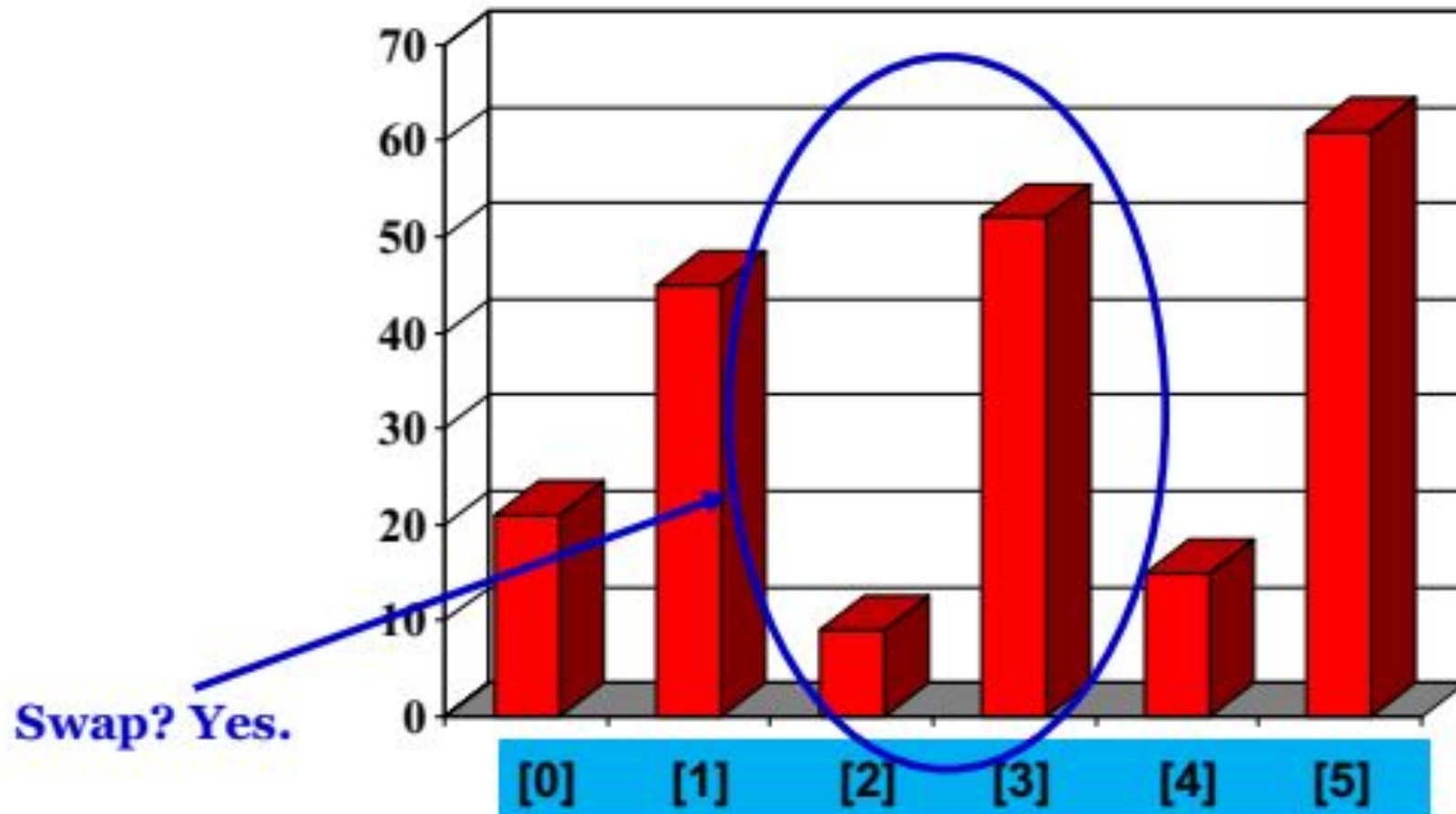
Bubble Sort Algorithm

- Repeat.



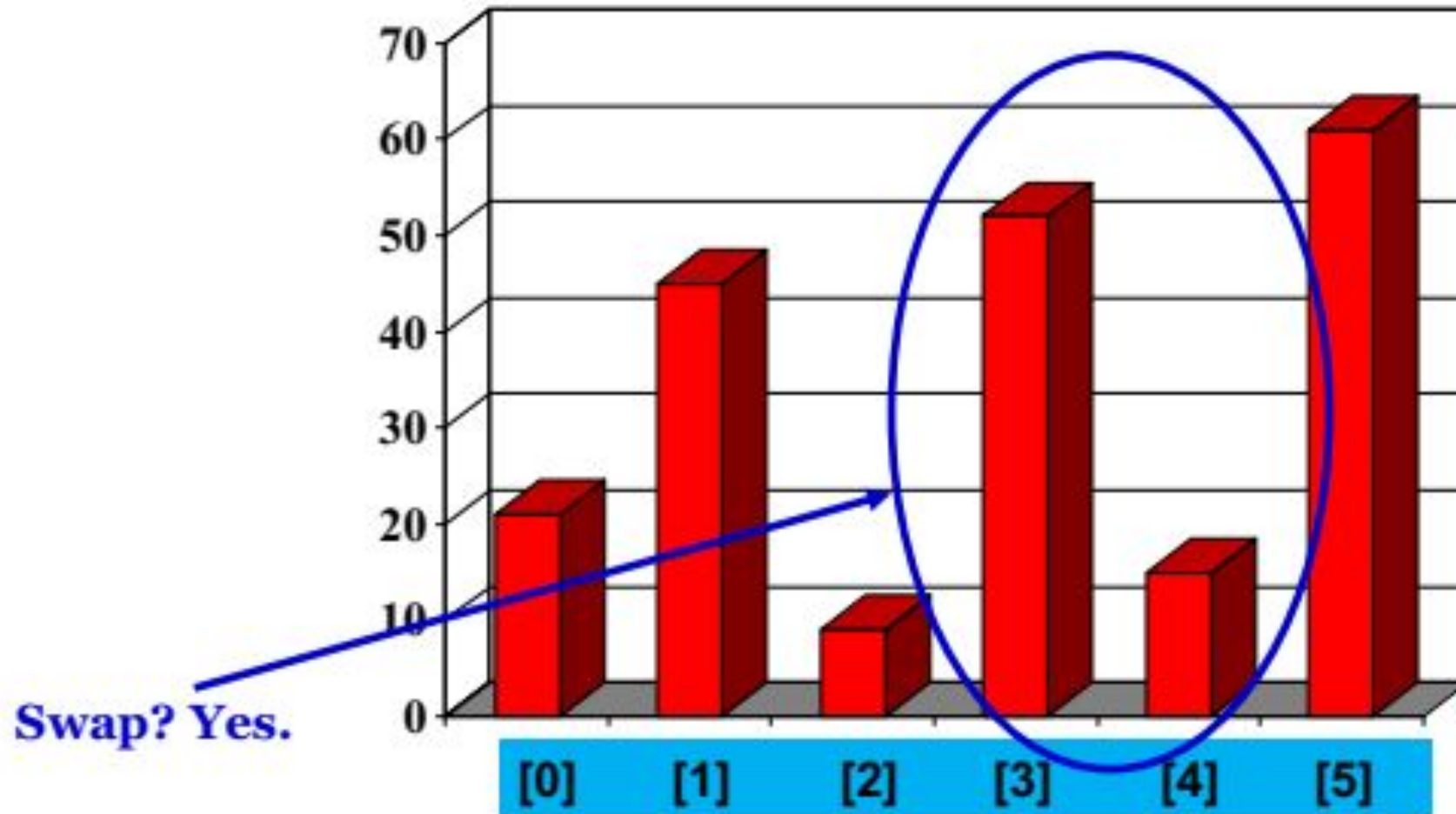
Bubble Sort Algorithm

- Repeat.



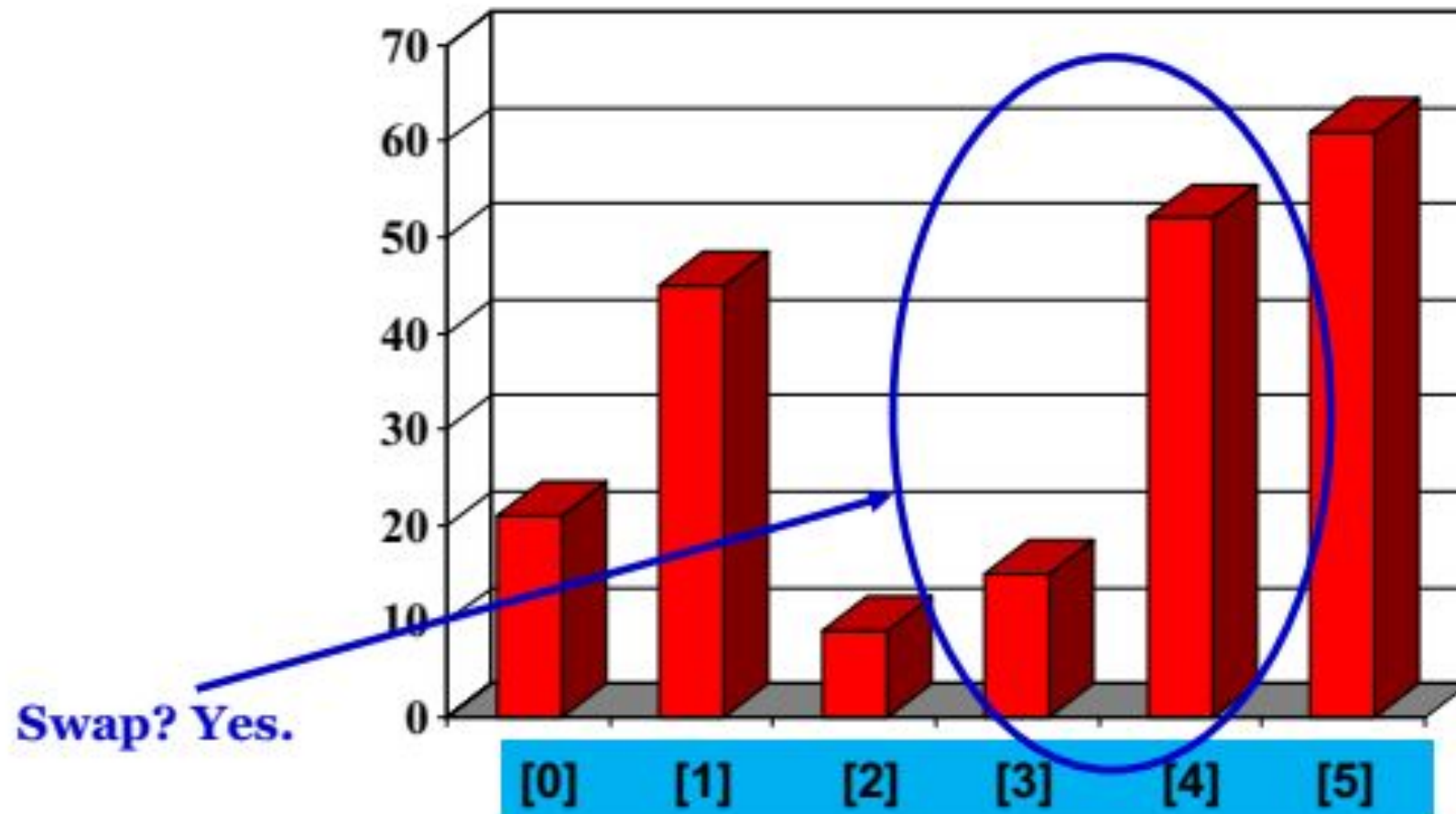
Bubble Sort Algorithm

- Repeat.



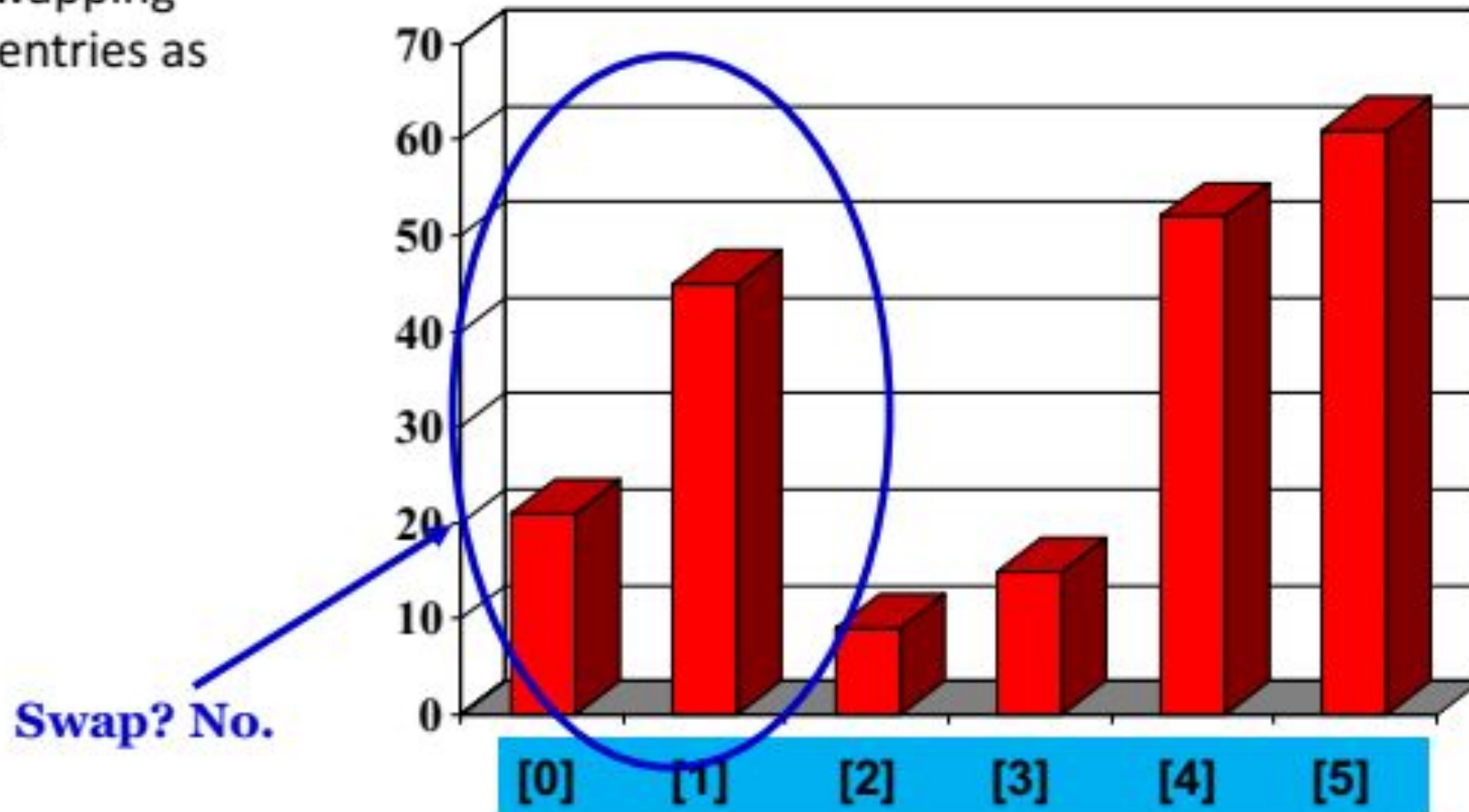
Bubble Sort Algorithm

- Repeat.



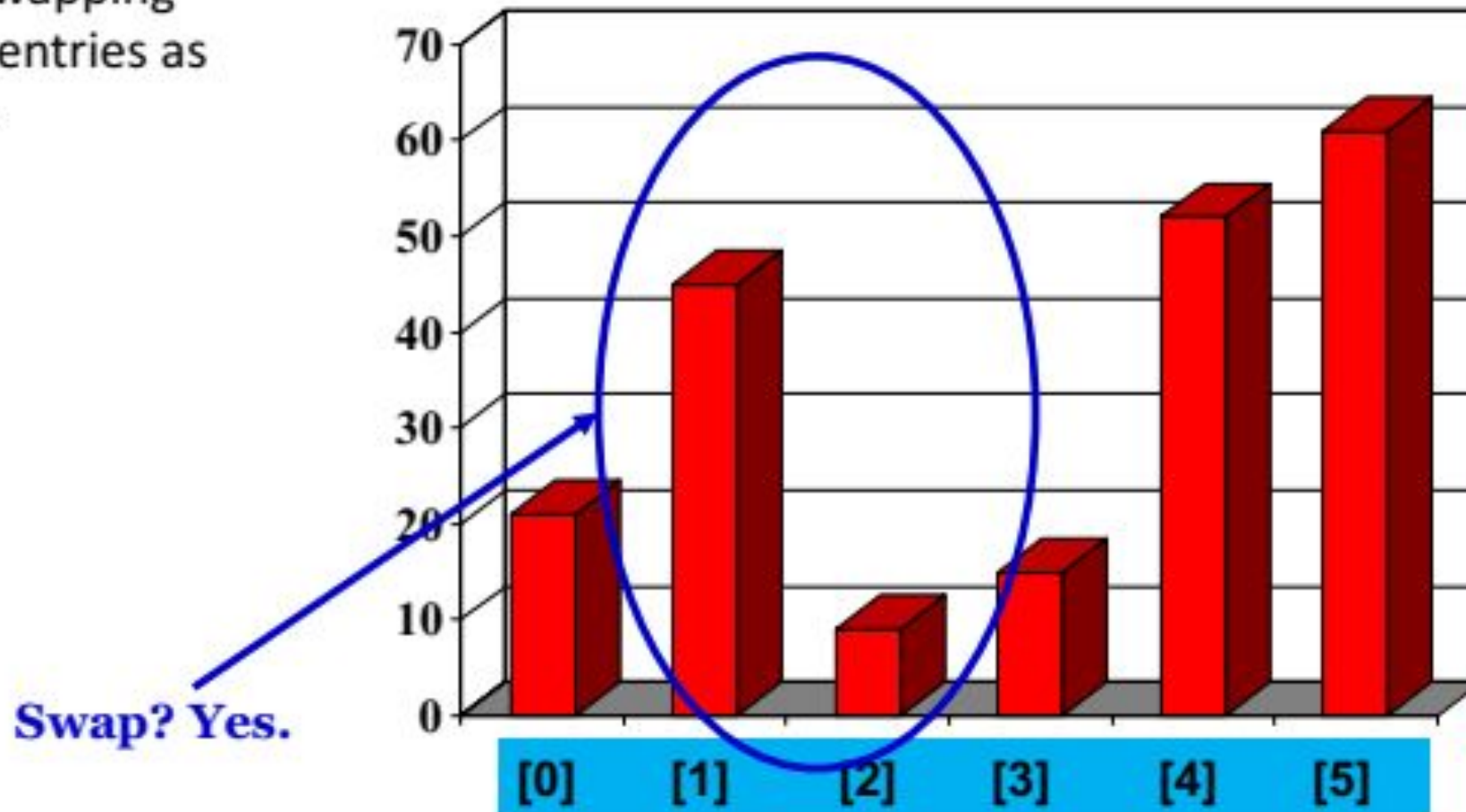
Bubble Sort Algorithm

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



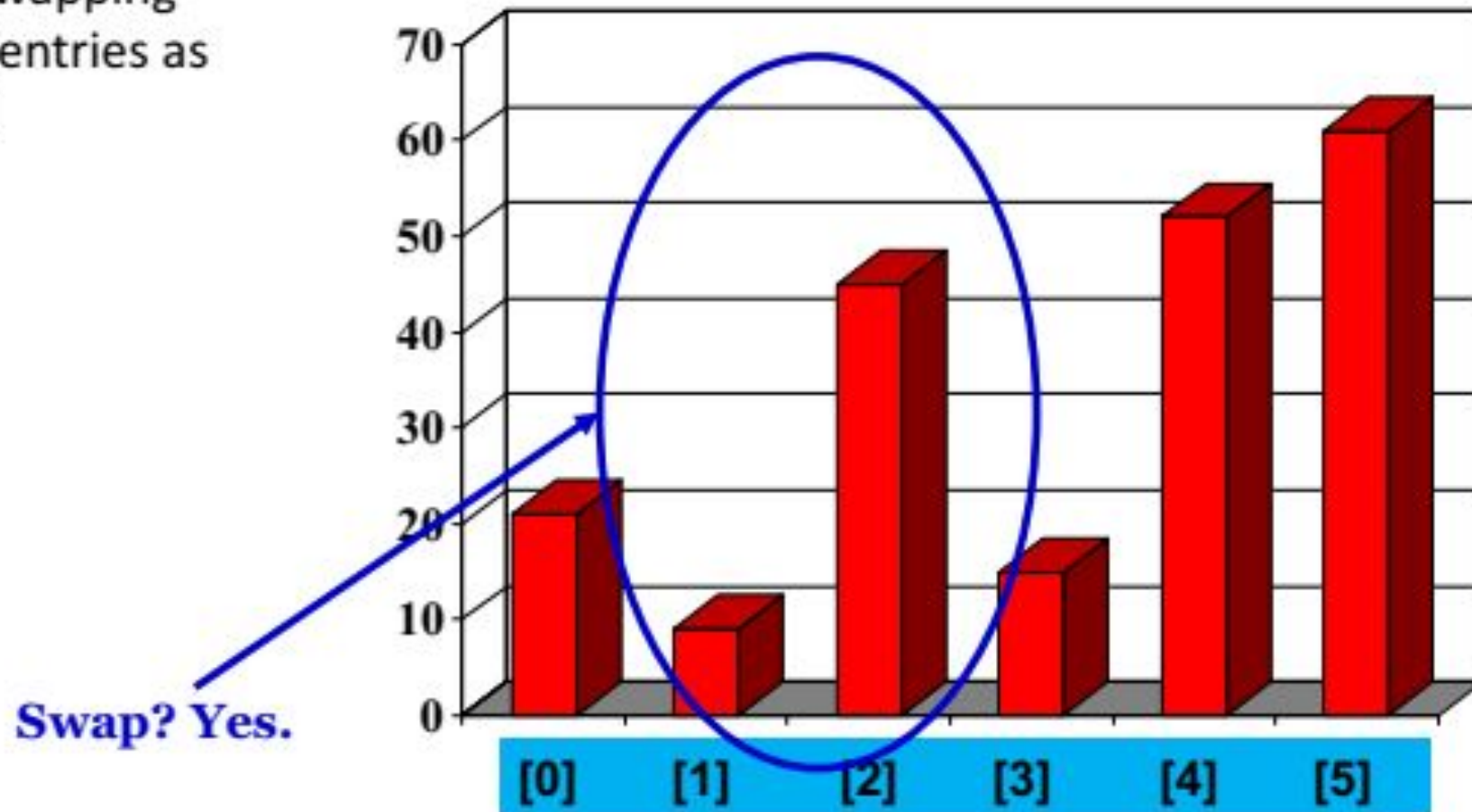
Bubble Sort Algorithm

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



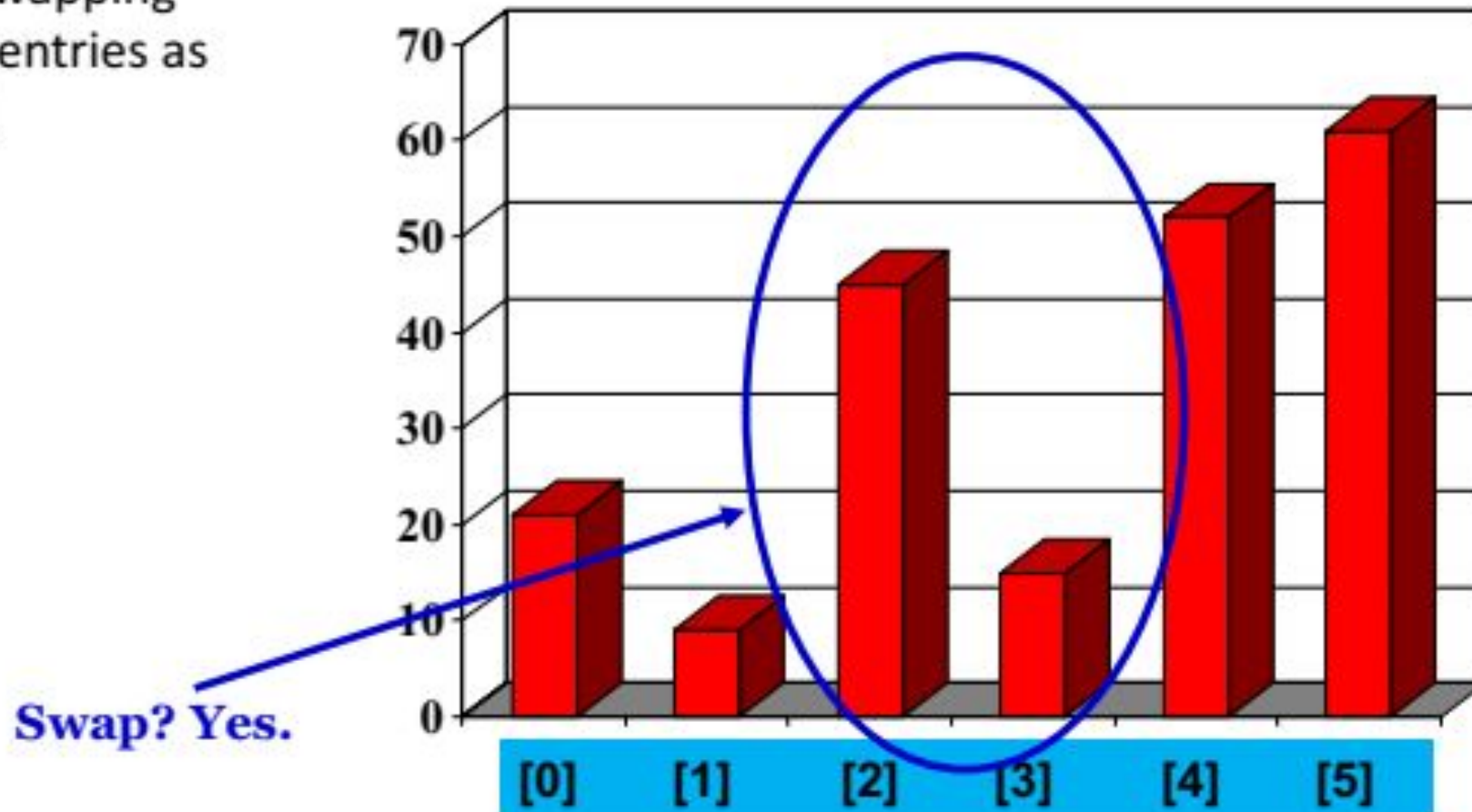
Bubble Sort Algorithm

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



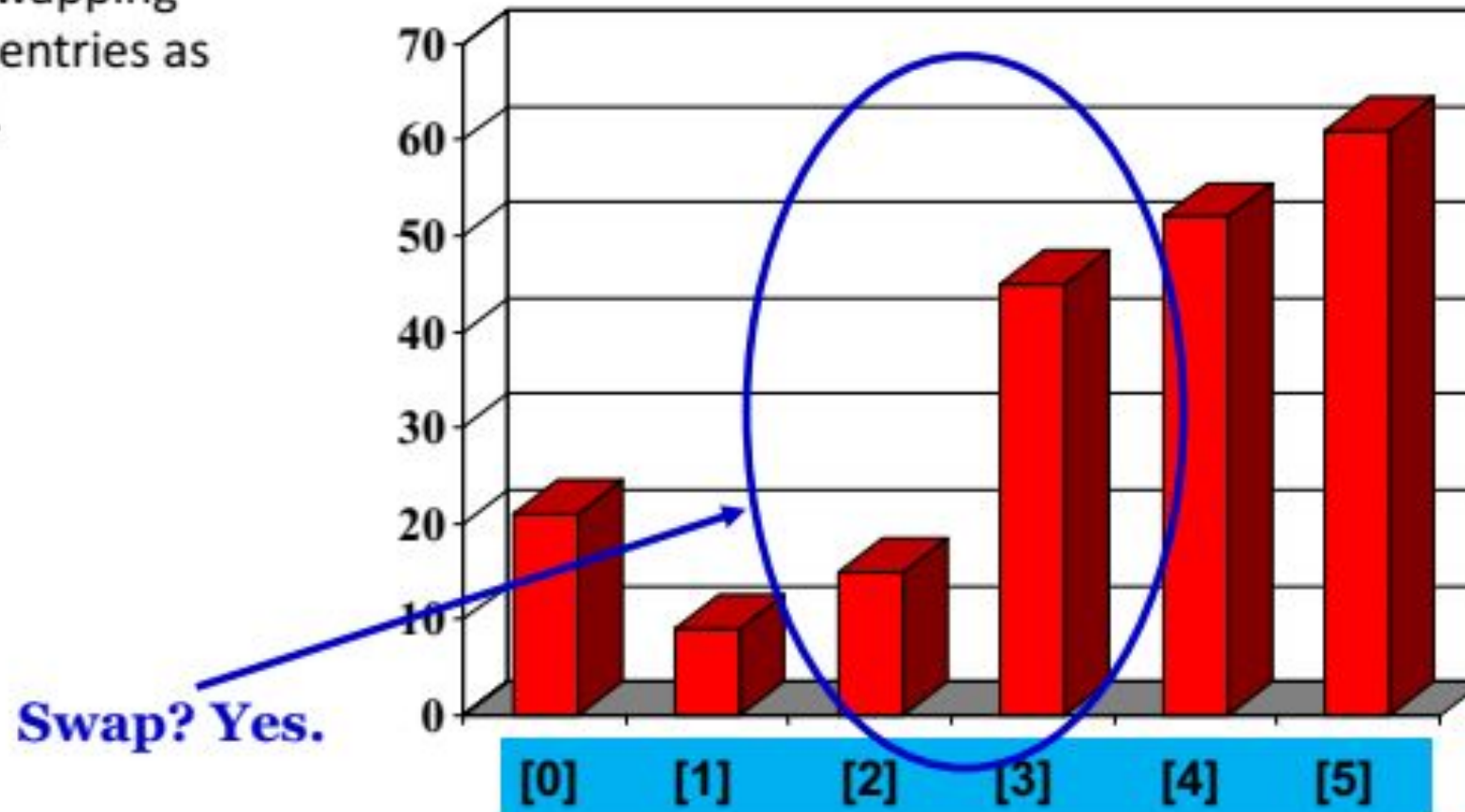
Bubble Sort Algorithm

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



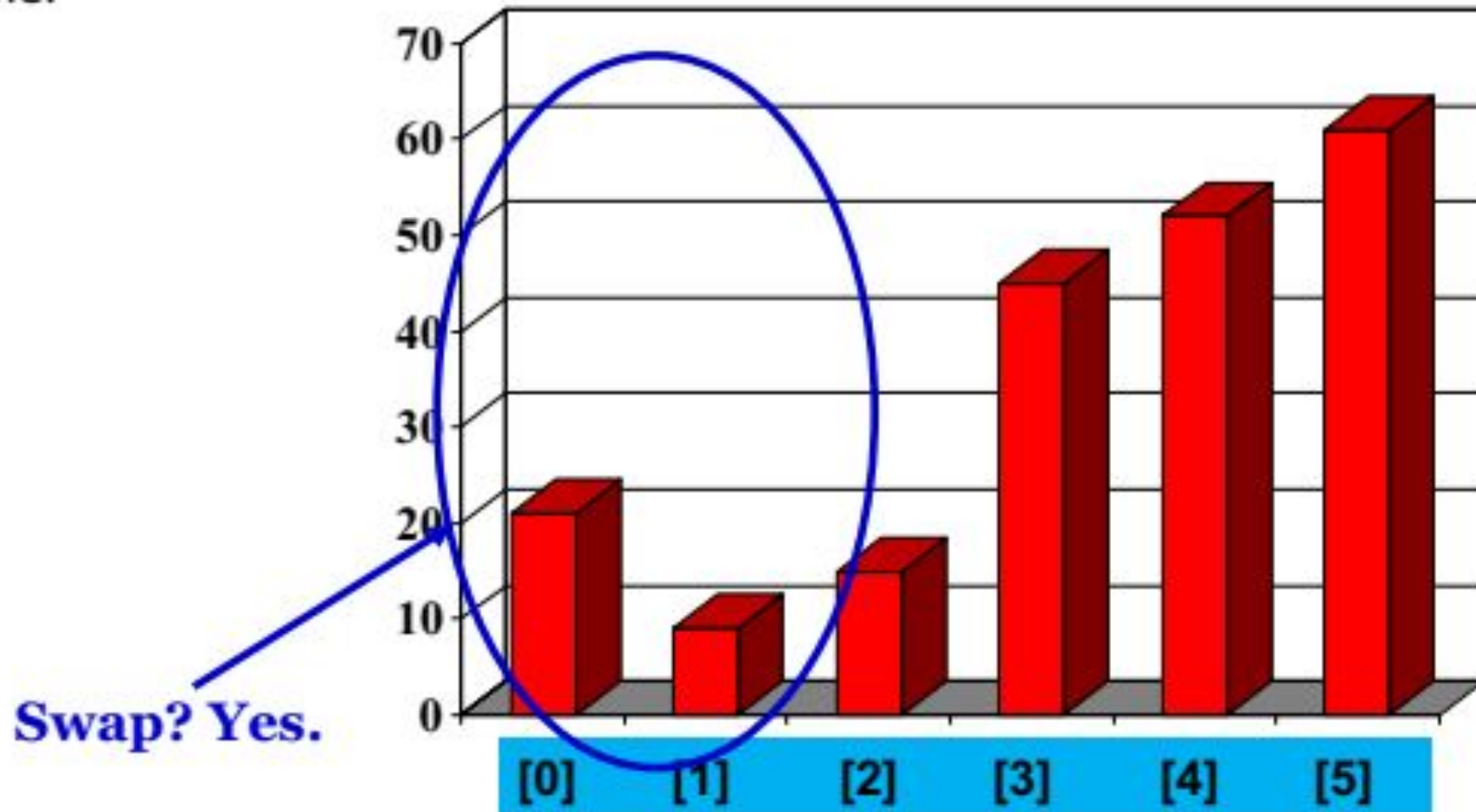
Bubble Sort Algorithm

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



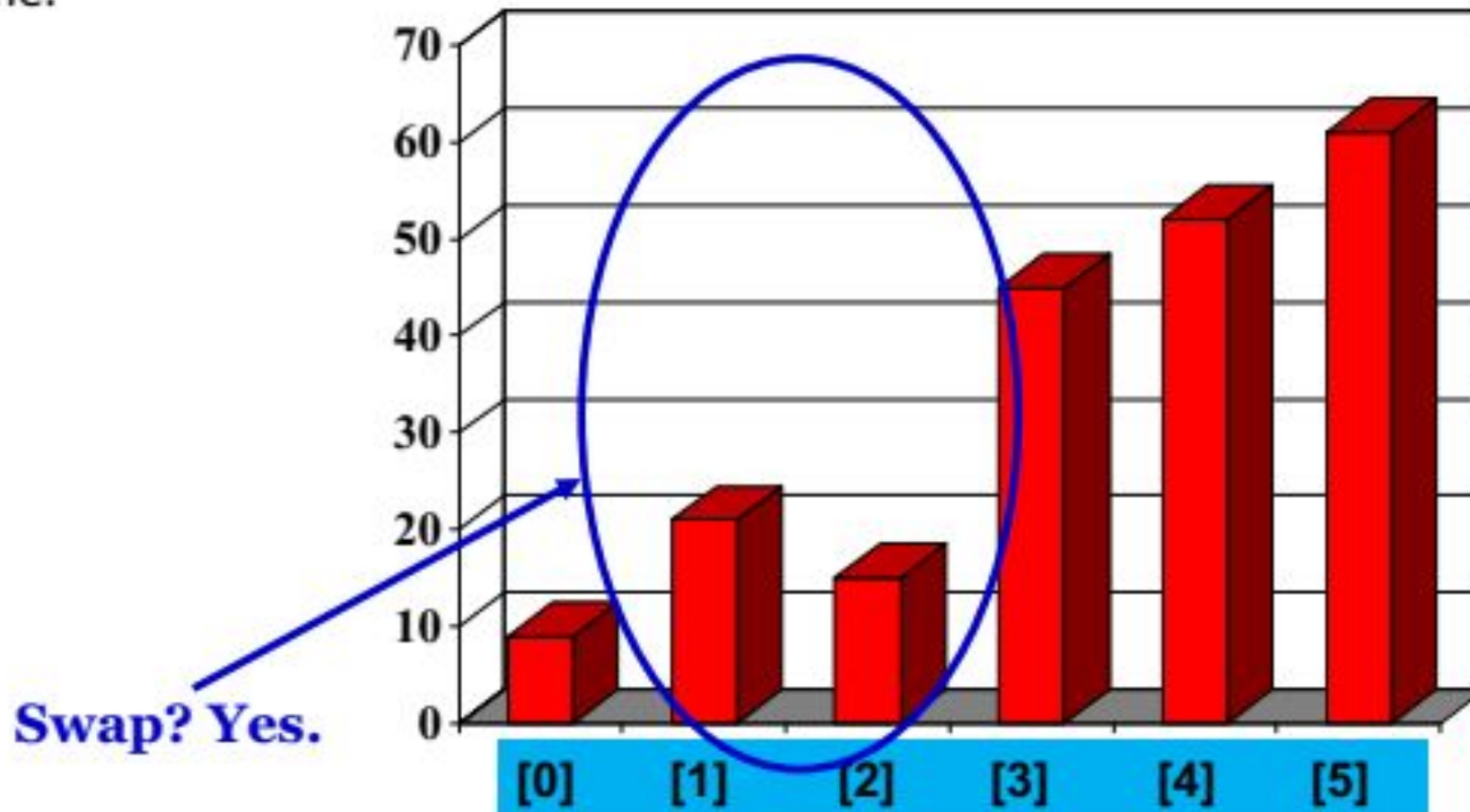
Bubble Sort Algorithm

- Continue looping, until done.

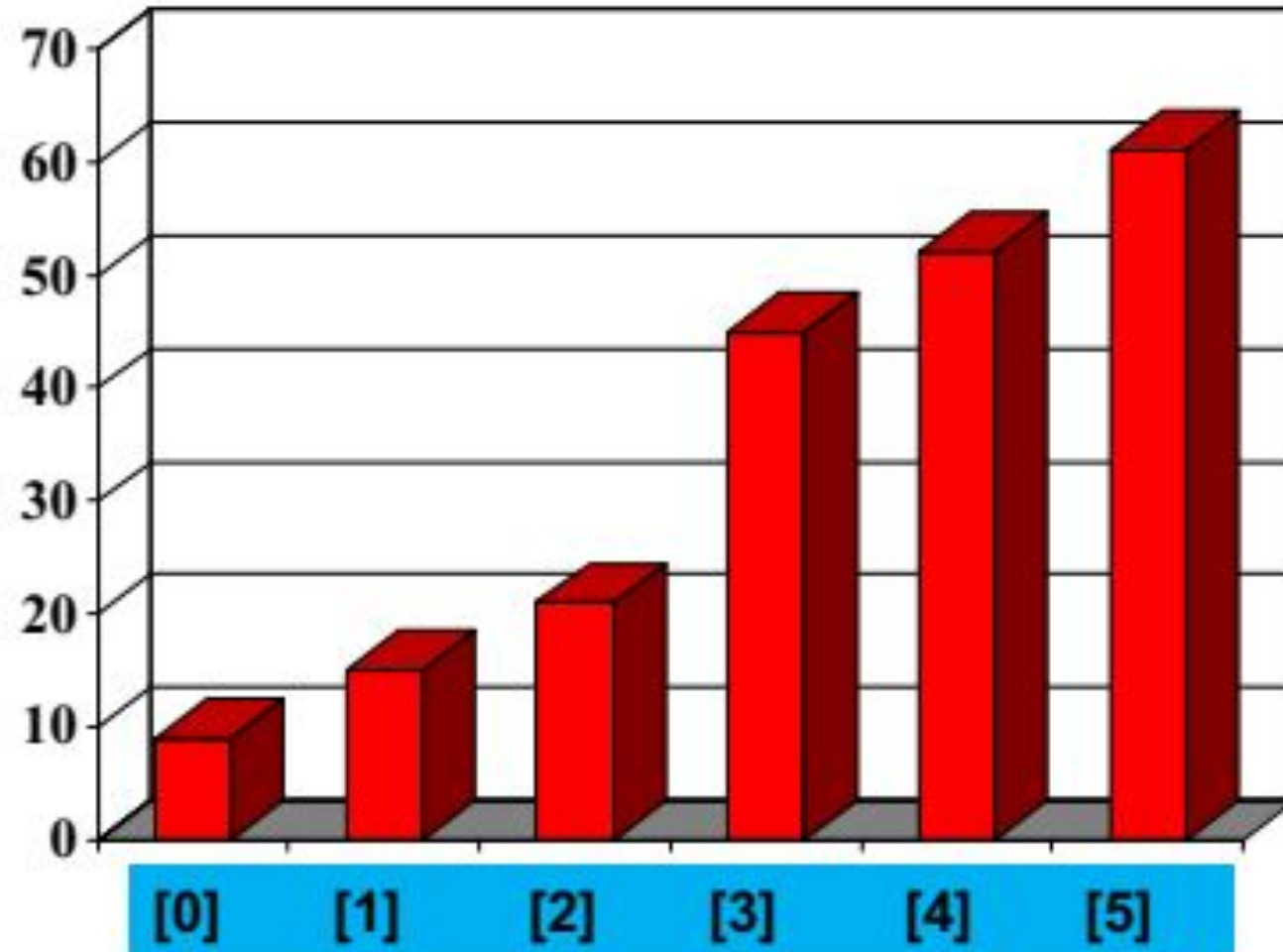


Bubble Sort Algorithm

- Continue looping, until done.



Bubble Sort Algorithm



Algorithm

◉ Bubble Sort:

Algorithm of bubble sort includes two steps repeated until the list is sorted.

- Compare adjacent elements, if the element on right side is smaller then swap their positions.
- Compare first element, second element and so on on completion of Pass 1 the largest element is at last position.

Bubble Sort Implementation

```
void bubbleSort (int list[ ], int size) {  
    int i, j, temp;  
    for ( i = 0; i < size; i++ ) { /* controls passes through the list */  
        for ( j = 0; j < size - 1; j++ ) /* performs adjacent comparisons */  
        {  
            if ( list[ j ] > list[ j+1 ] ) /* determines if a swap should occur */  
            {  
                temp = list[ j ]; /* swap is performed */  
                list[ j ] = list[ j + 1 ];  
                list[ j+1 ] = temp;  
            } // end of if statement  
        } // end of inner for loop  
    } // end of outer for loop  
} // end of function
```


Performance

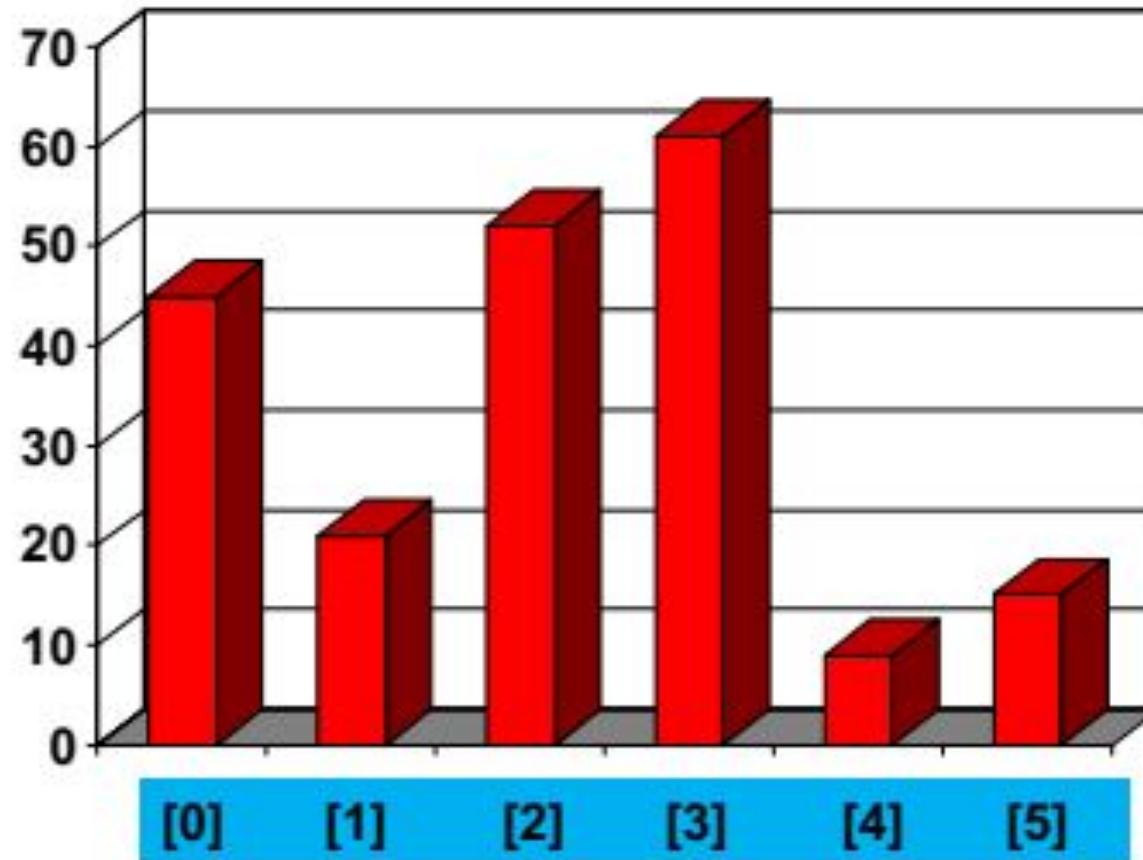
- **Worst and Average Case Time Complexity:** $O(n^2)$. Worst case occurs when array is reverse sorted.
- **Best Case Time Complexity:** $O(n)$. Best case occurs when array is already sorted.
- **Sorting In Place:** Yes
- **Stable:** Yes

Selection Sort

- It is specifically an **in-place comparison sort**
- Noted for its simplicity,
- It has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited
- The algorithm
 - finds the **minimum** value,
 - **swaps** it with the value in the first position, and
 - repeats these steps for the remainder of the list
- It does no more than **n swaps**, and thus is useful where swapping is very expensive

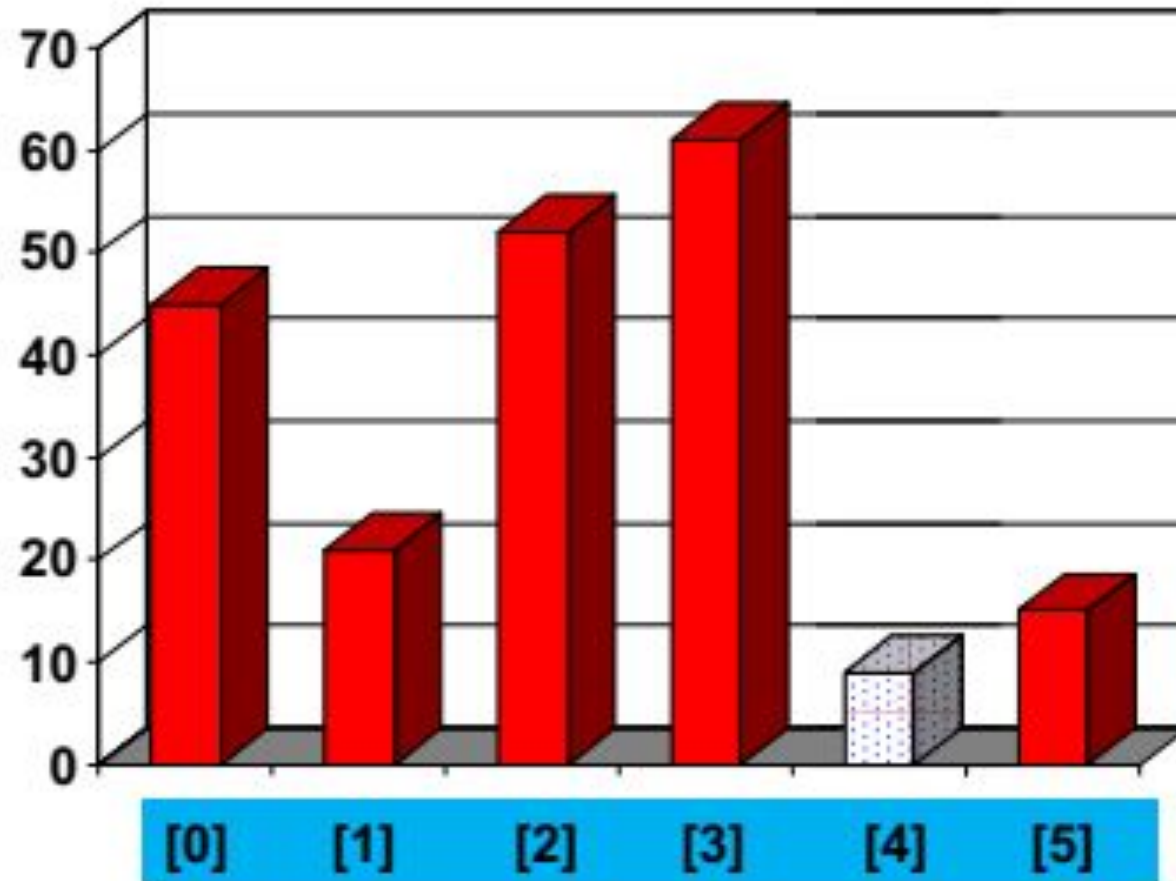
Selection Sort

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



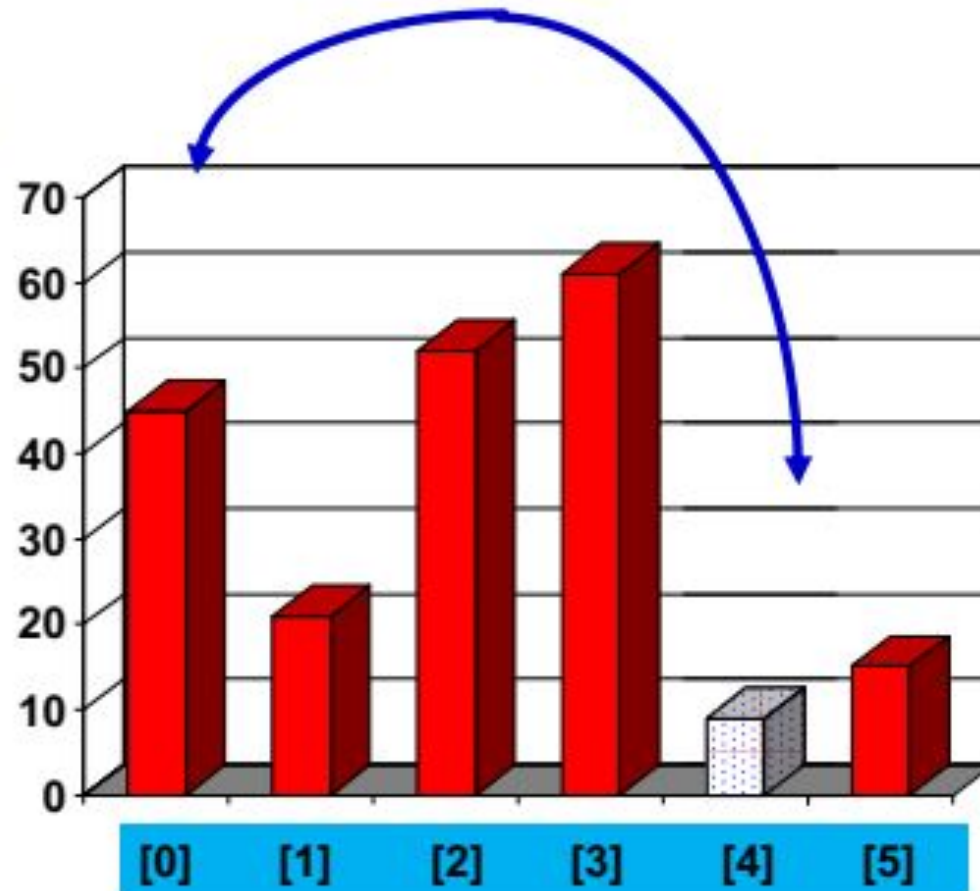
Selection Sort

- Start by finding the smallest entry.



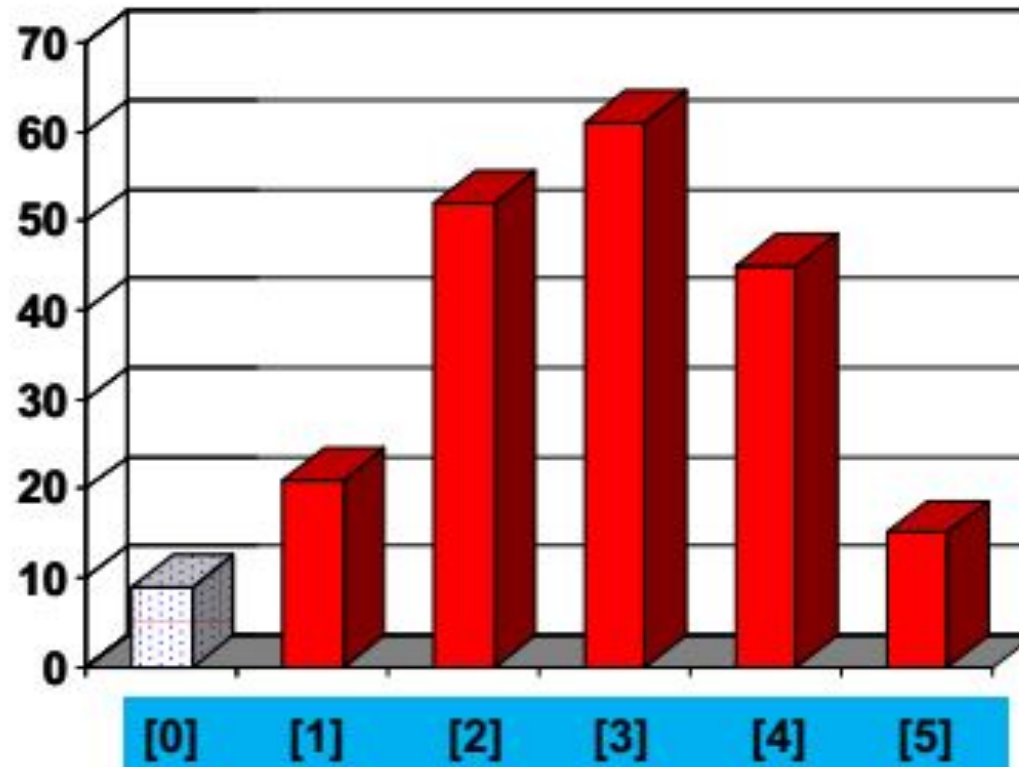
Selection Sort

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



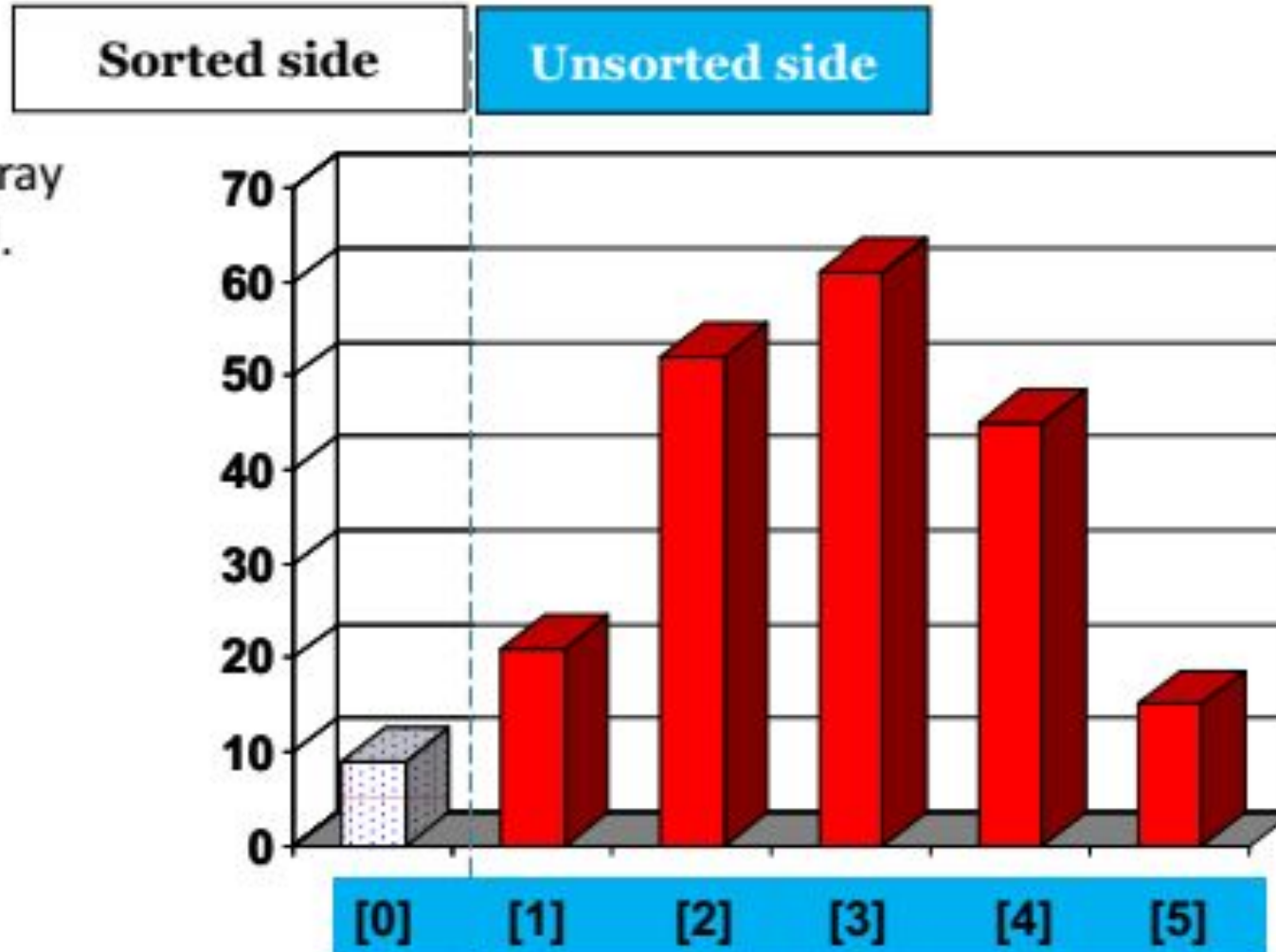
Selection Sort

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

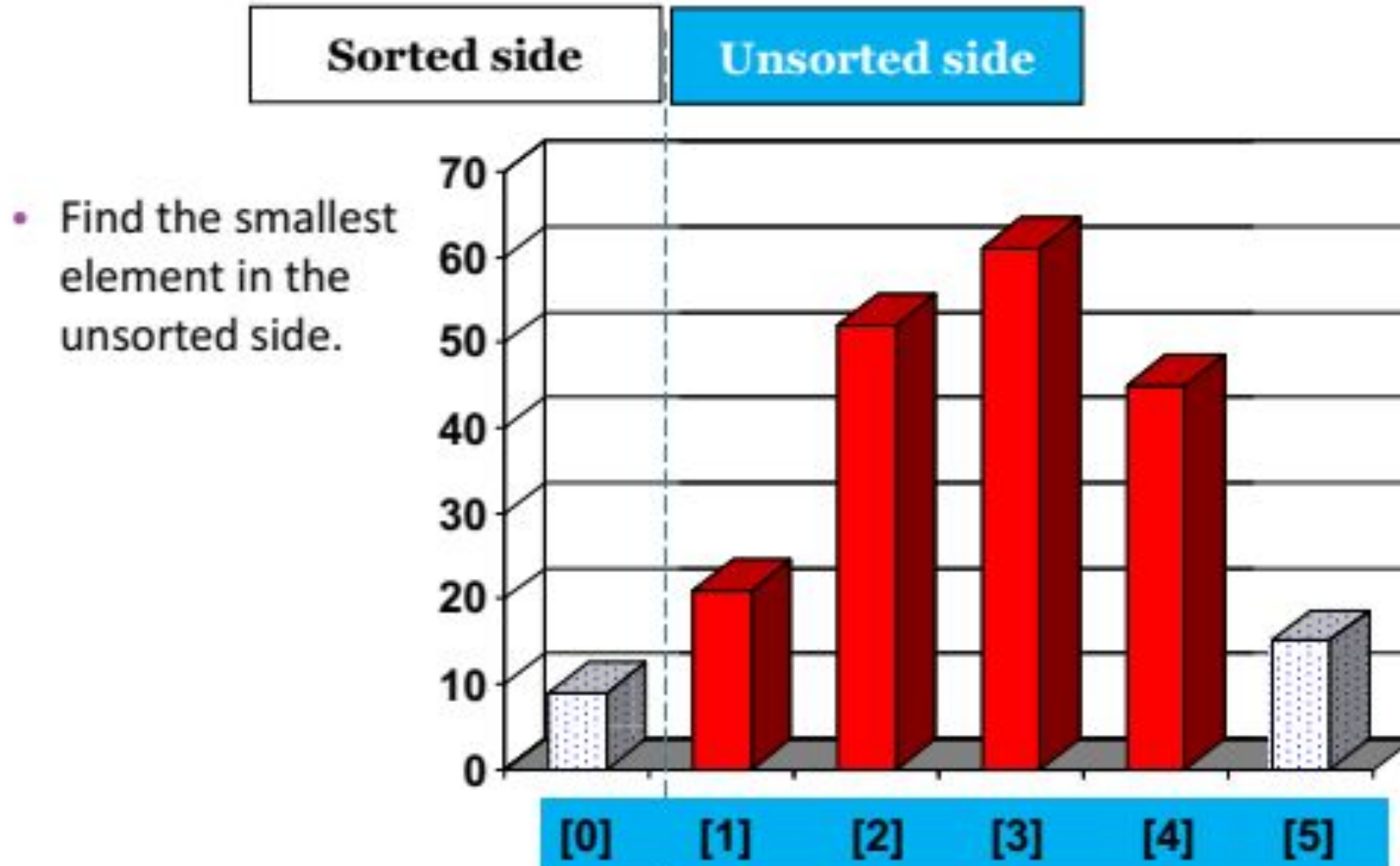


Selection Sort

- Part of the array is now sorted.

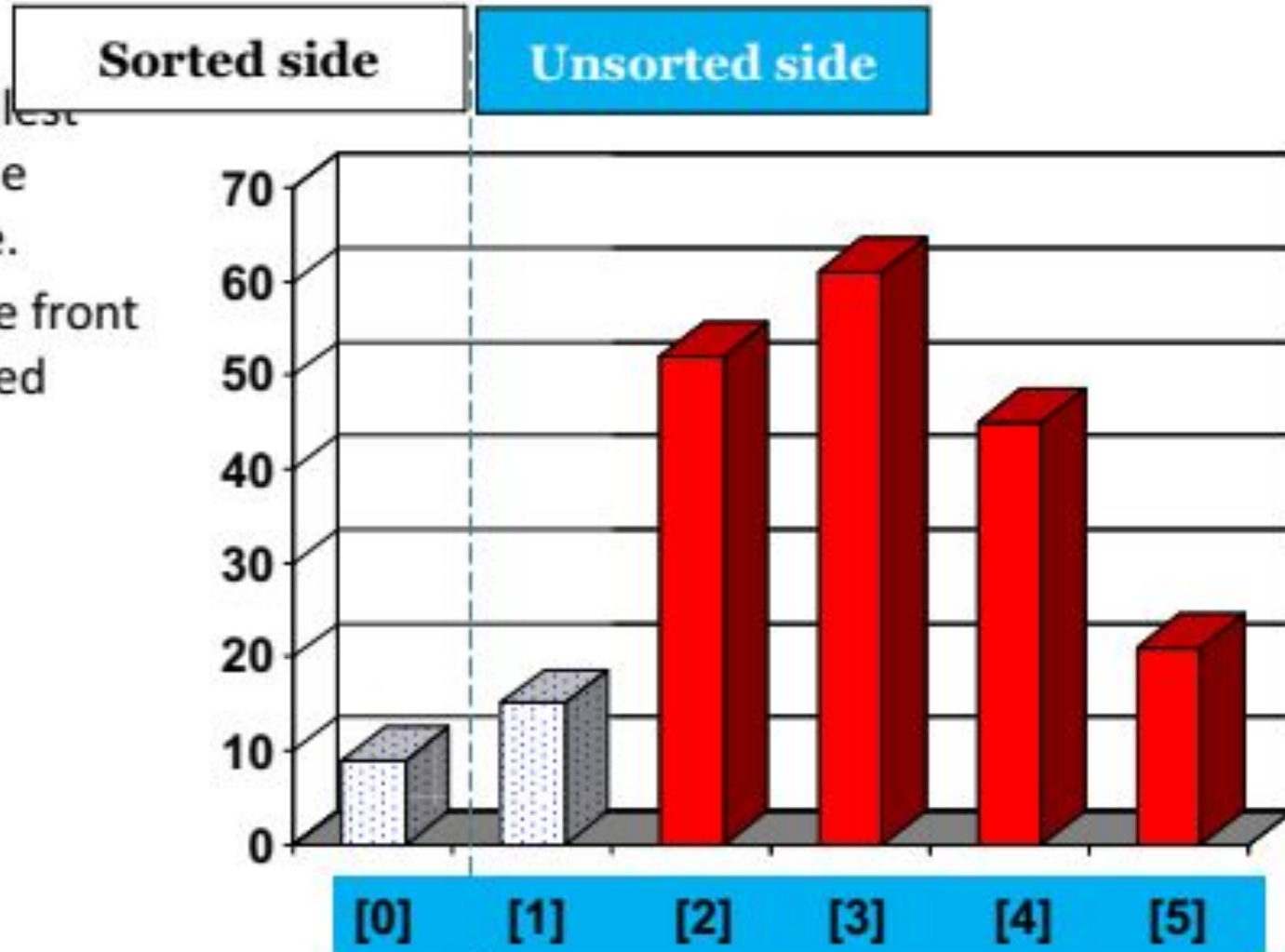


Selection Sort



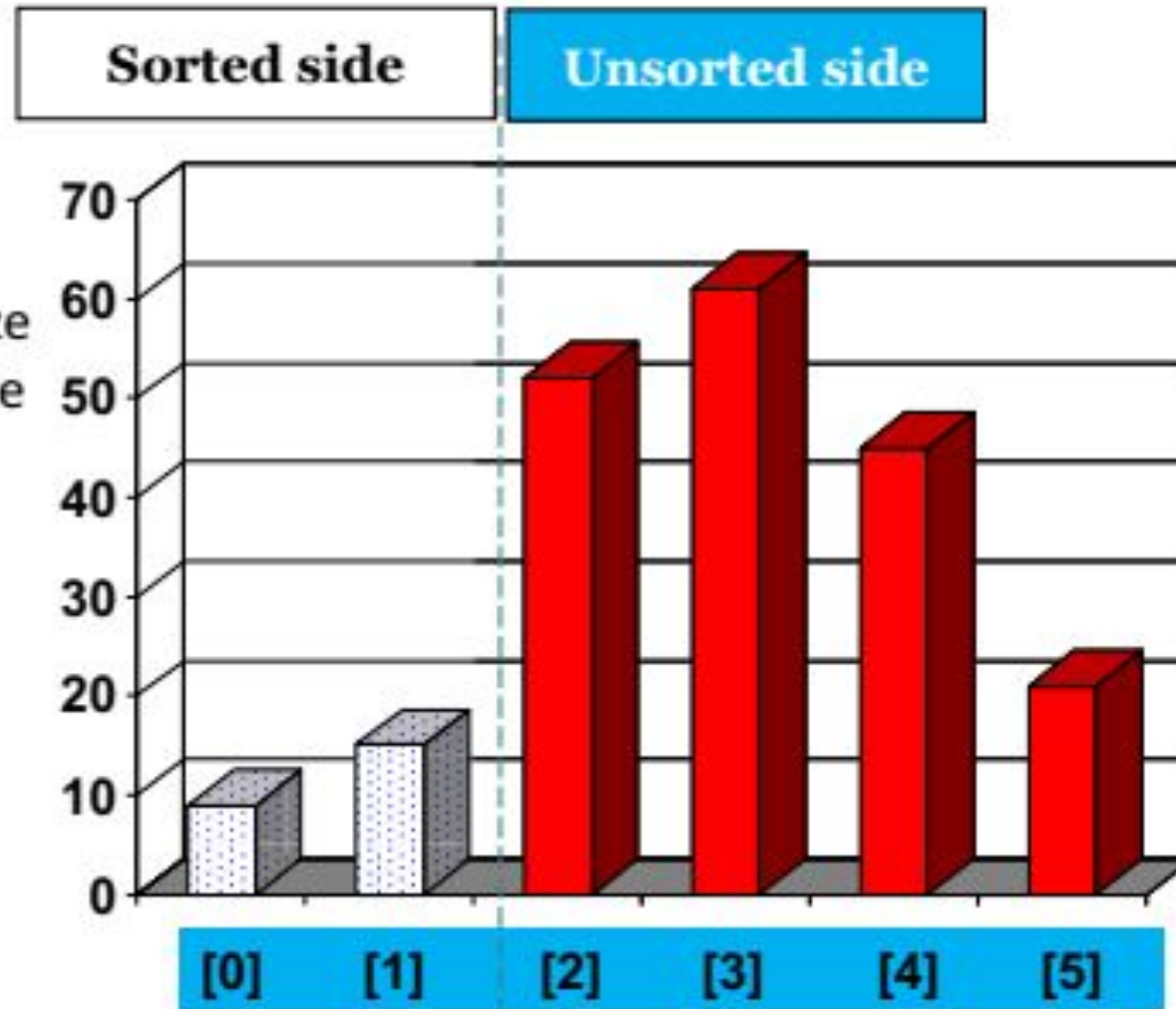
Selection Sort

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



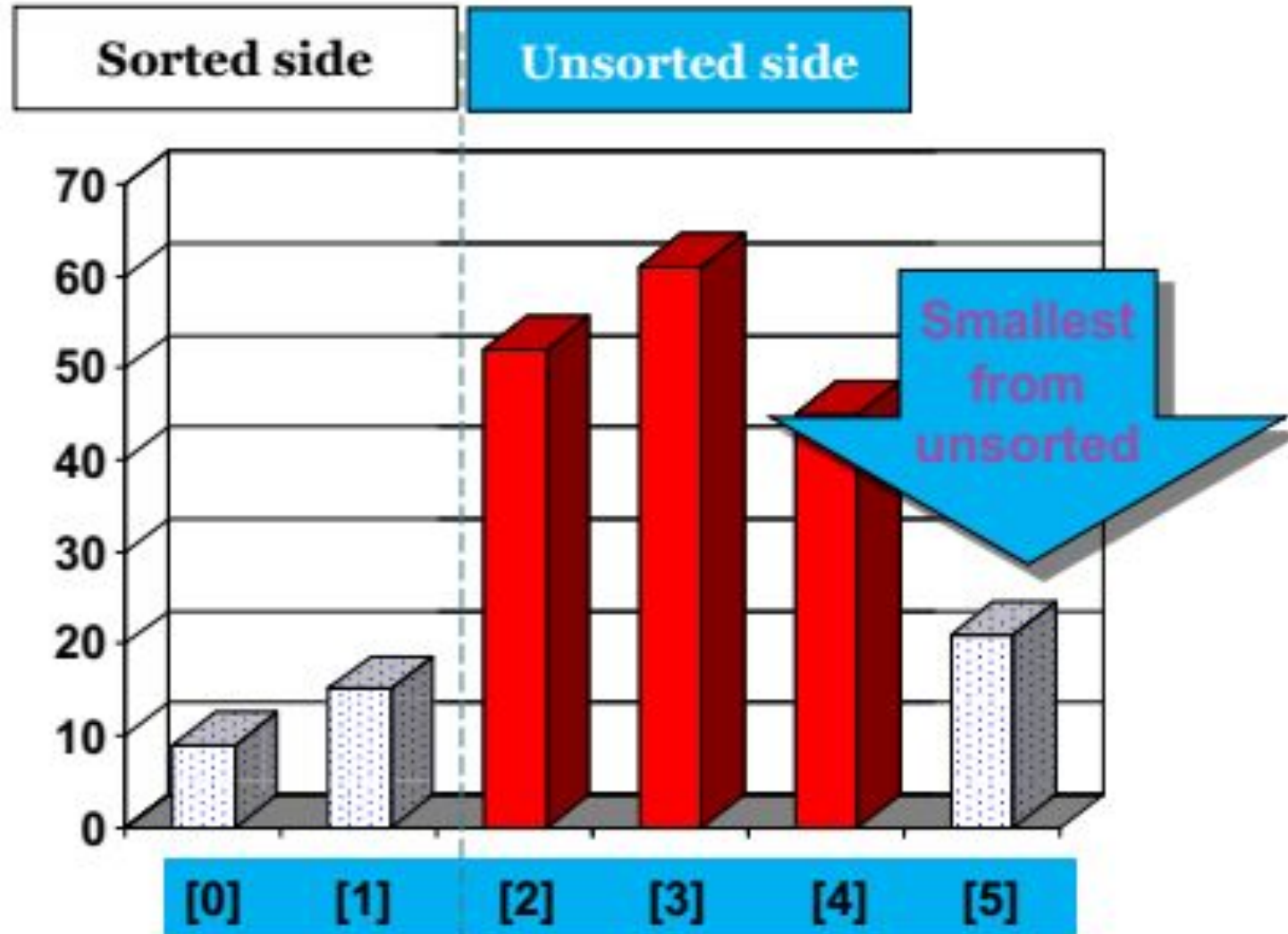
Selection Sort

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



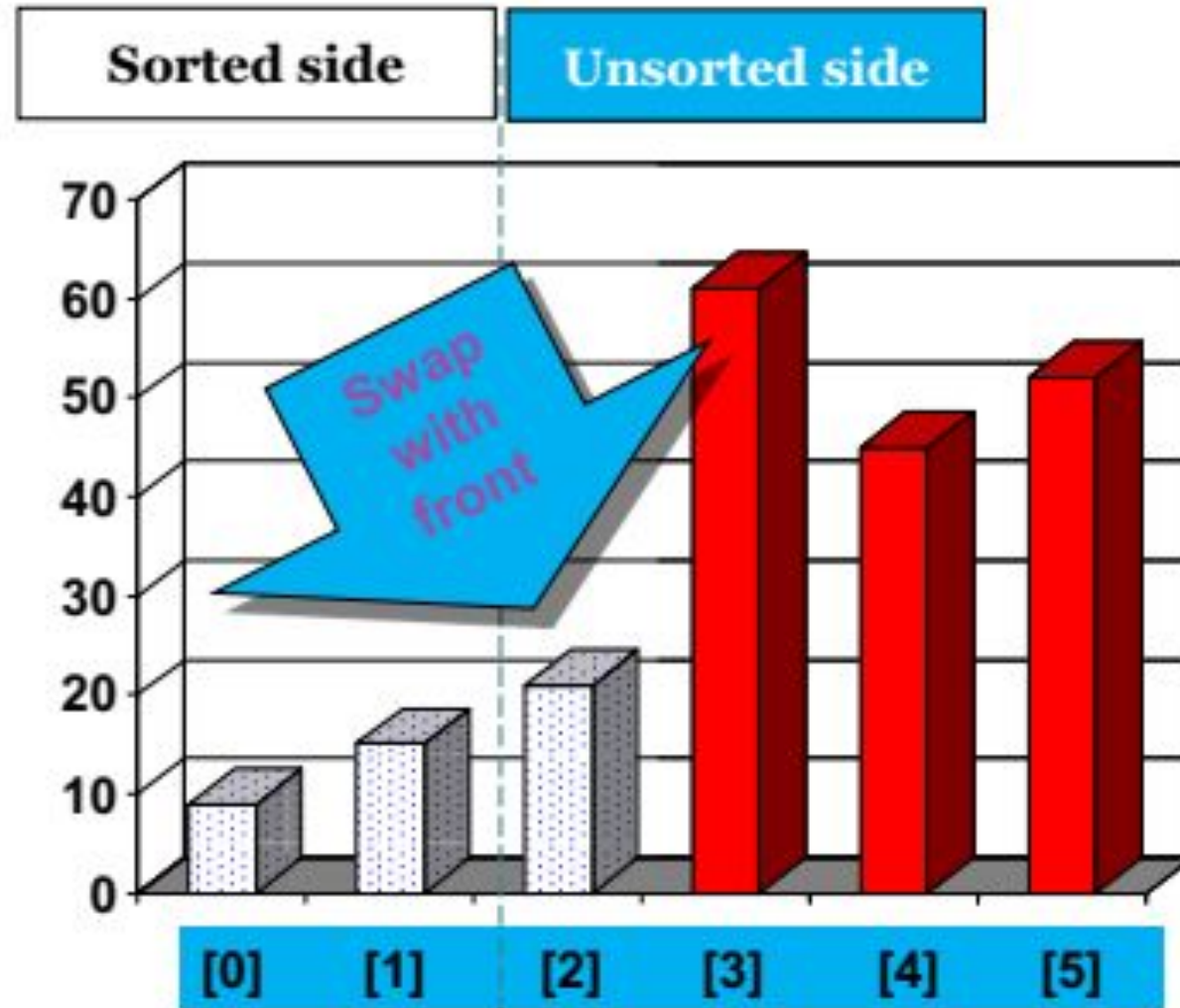
Selection Sort

- The process continues...

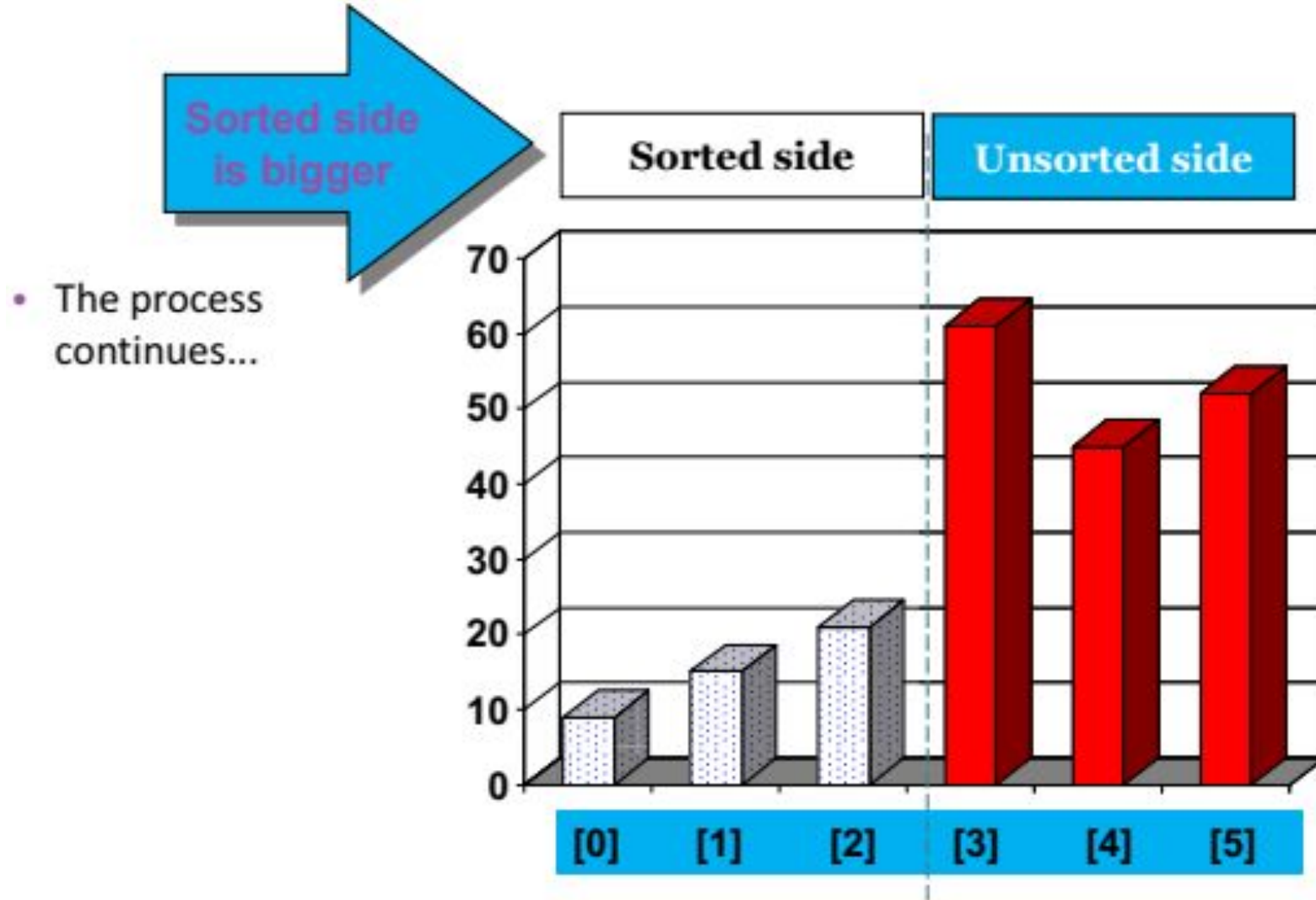


Selection Sort

- The process continues...

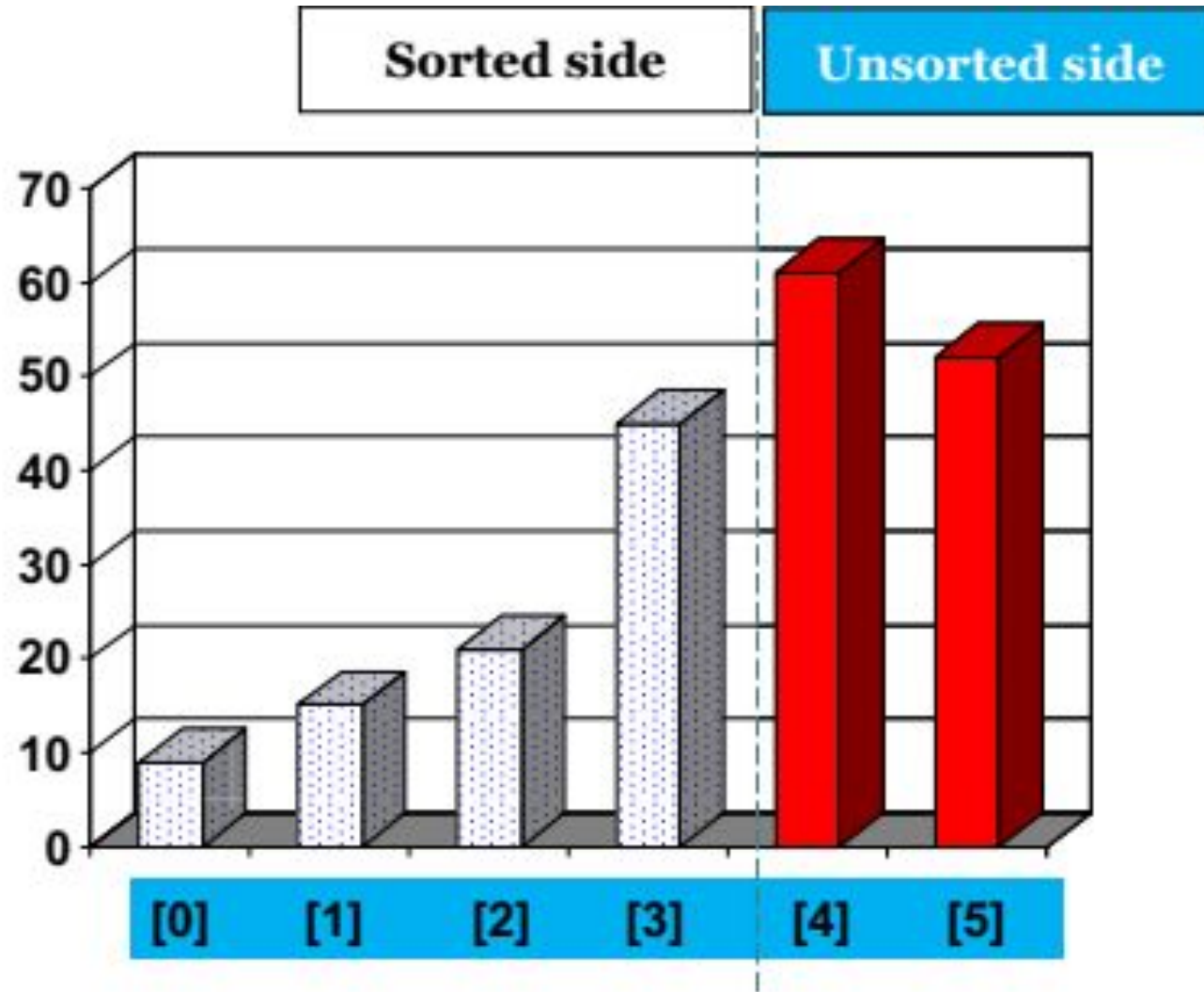


Selection Sort



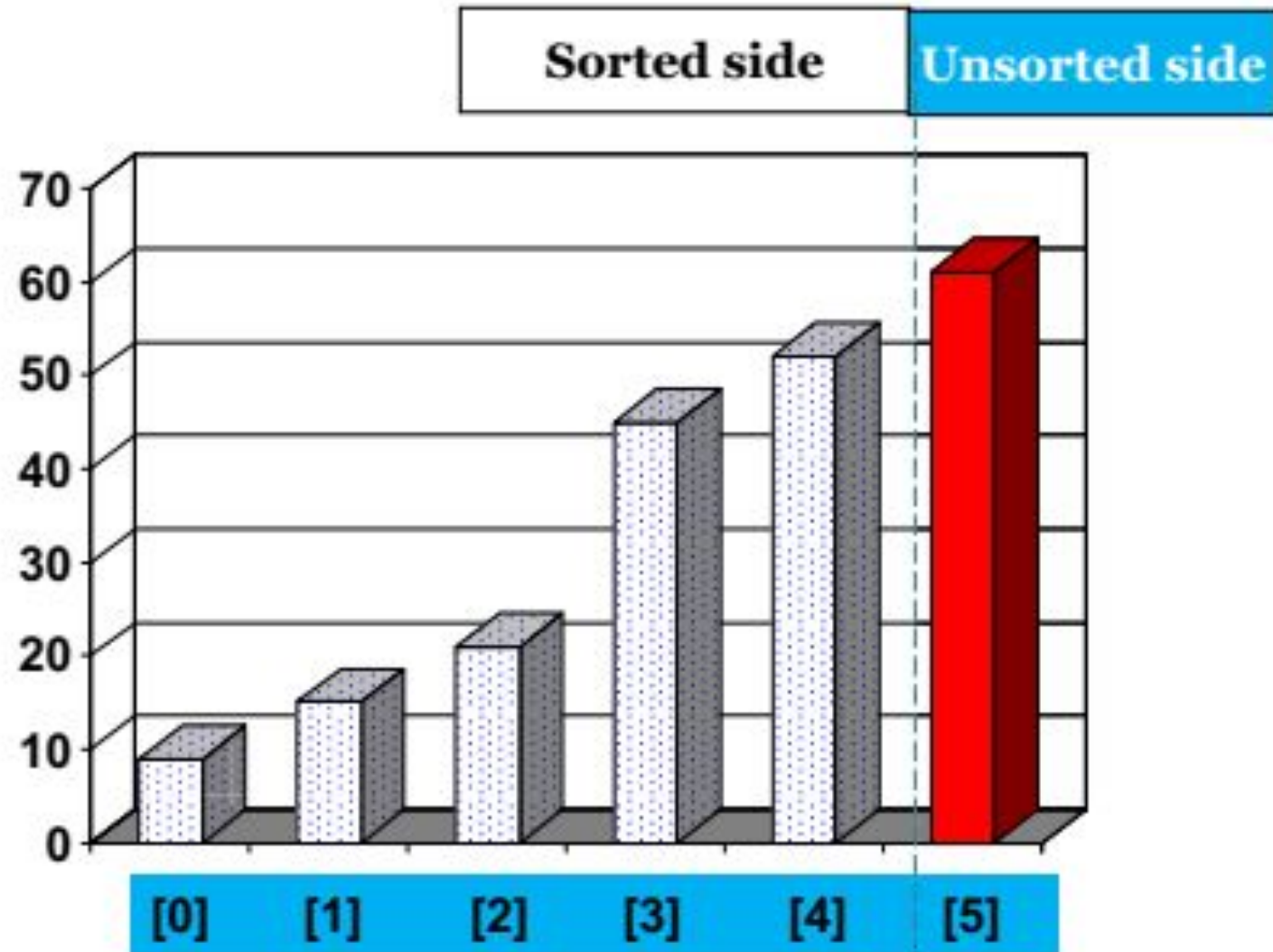
Selection Sort

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



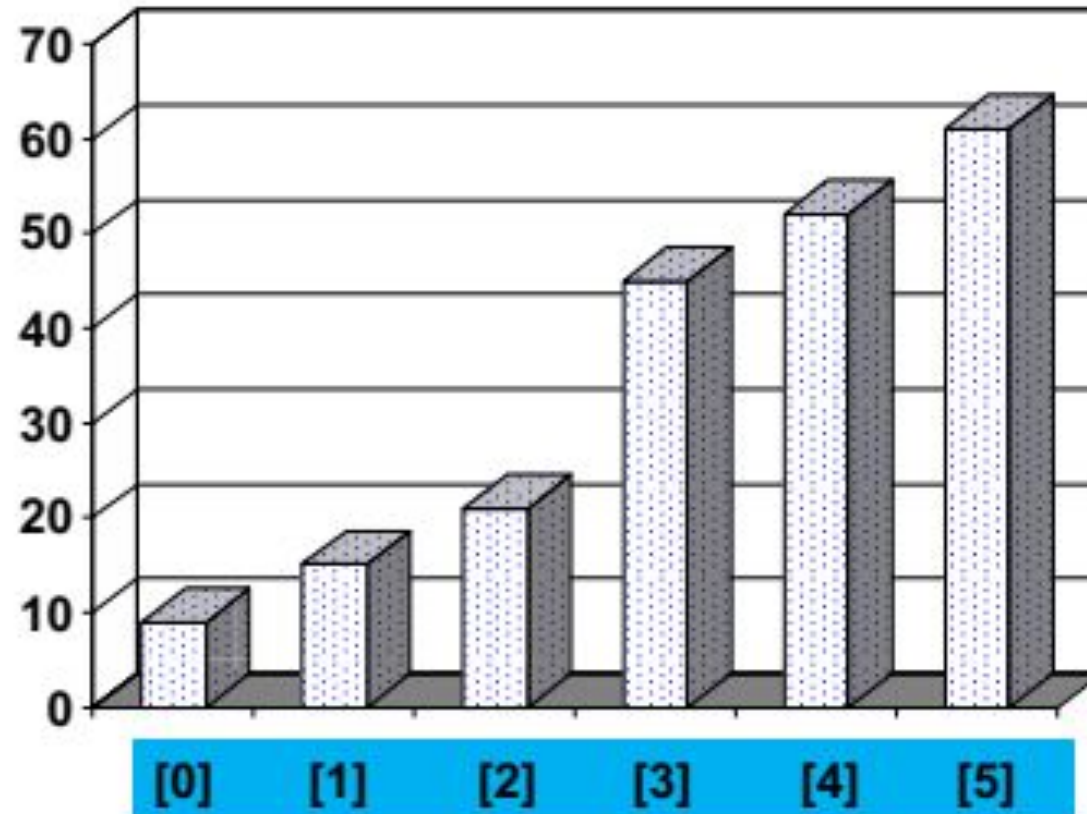
Selection Sort

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



Selection Sort

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



Selection Sort

- The algorithm
 - finds the **minimum** value,
 - **swaps** it with the value in the first position, and
 - repeats these steps for the remainder of the list

Selection Sort

Input: An array $A[1..n]$ of n elements.

Output: $A[1..n]$ sorted in descending order

1. for $i \leftarrow 1$ to $n - 1$
2. $\text{min} \leftarrow i$
3. for $j \leftarrow i + 1$ to n {Find the i th smallest element.}
4. if $A[j] < A[\text{min}]$ then
5. $\text{min} \leftarrow j$
6. end for
7. if $\text{min} \neq i$ then interchange $A[i]$ and $A[\text{min}]$
8. end for

Selection Sort

```
int A[SIZE] = {54,15,32,78,23,90,48,86,23,65};
int i, j, min, temp;

for (i = 0; i < SIZE - 1; i++){
    min = i;
    // find the minimum number in the array
    for (j = i+1; j < SIZE; j++){
        if (A[j] < A[min]){
            min = j;
        }
    }
    // swap the two numbers
    temp = A[i];
    A[i] = A[min];
    A[min] = temp;
}
```

Complexity of Selection Sort

- An in-place comparison sort
- $O(n^2)$ complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort.
- Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array

Complexity of Selection Sort

- Selecting the lowest element requires scanning all n elements (this takes $n - 1$ comparisons) and then swapping it into the first position
- Finding the next lowest element requires scanning the remaining $n - 1$ elements and so on,
- for $(n - 1) + (n - 2) + \dots + 2 + 1 = n(n - 1) / 2 \in O(n^2)$ comparisons
- Each of these scans requires **one swap** for $n - 1$ elements (**the final element is already in place**).

Complexity of Selection Sort

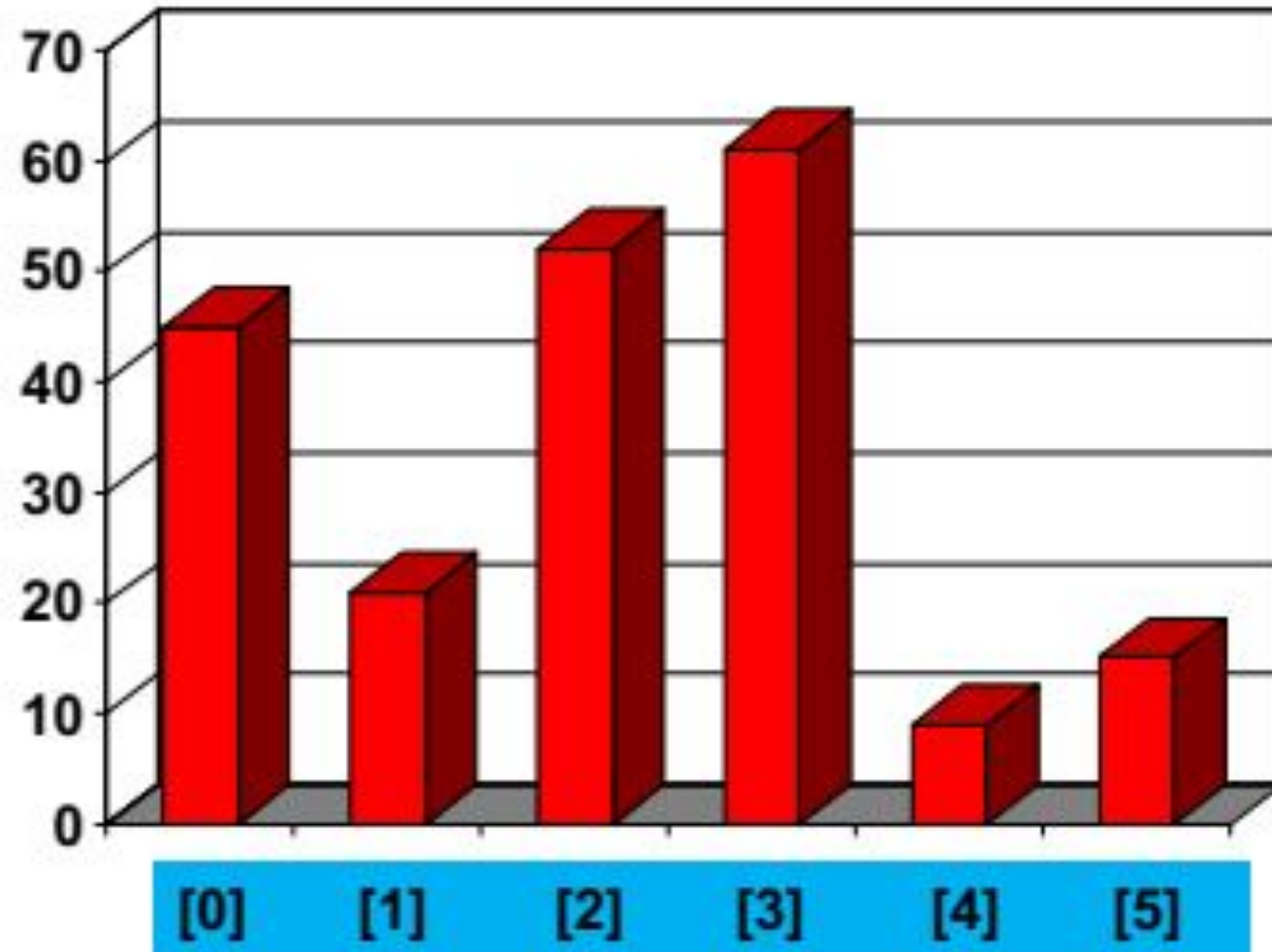
- Worst case performance $O(n^2)$
- Best case performance $O(n^2)$
- Average case performance $O(n^2)$

Insertion Sort

- Insertion sort is not as slow as bubble sort, and it is easy to understand.
- Insertion sort keeps making the **left side** of the array sorted until the whole array is sorted.
- Real life example:
 - Insertion sort works the same way as arranging your hand when playing cards.
 - To sort the cards in your hand you extract a card, shift the remaining cards, and then insert the extracted card in the correct place.

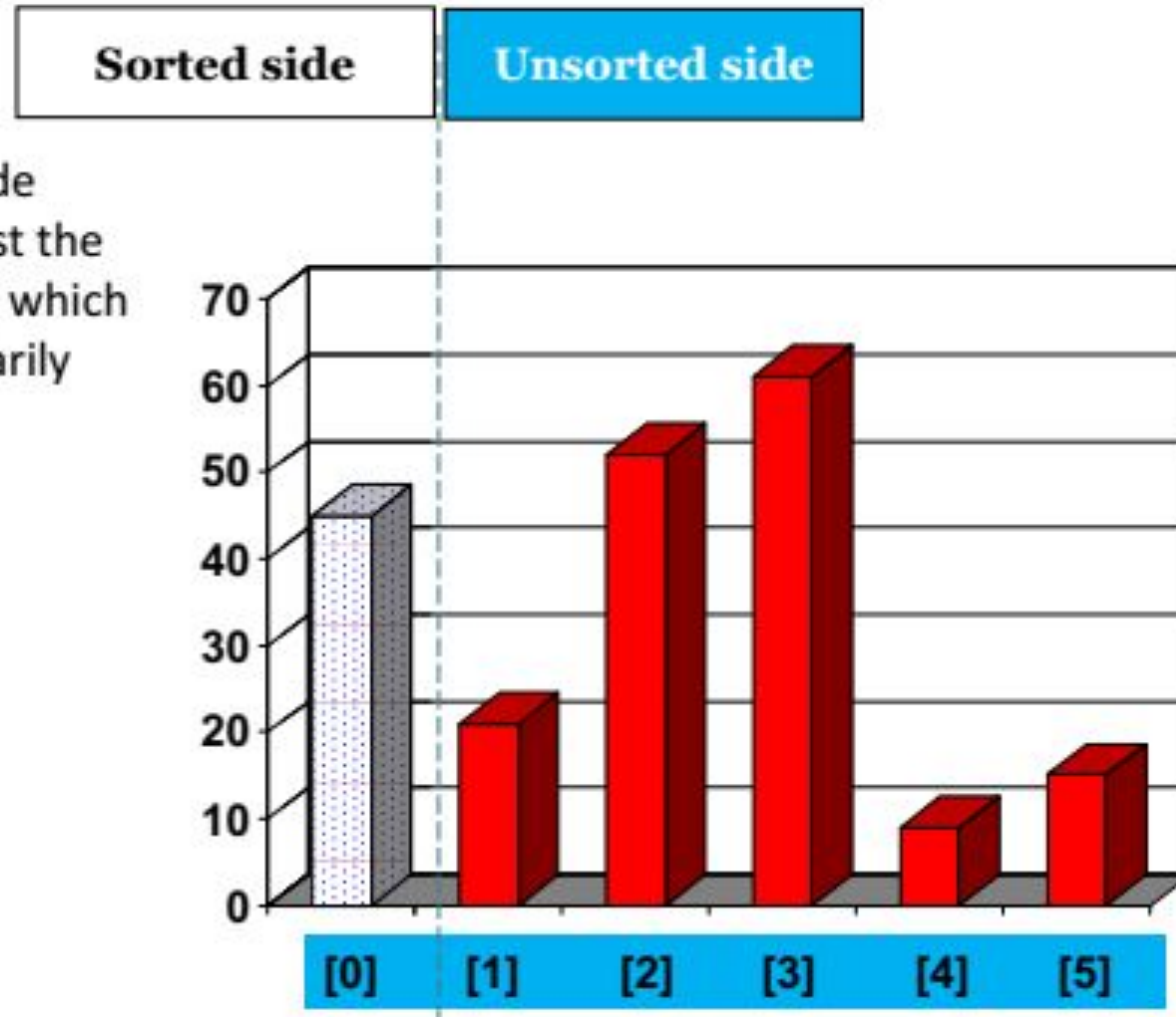
Insertion Sort

- Views the array as having two sides
- a sorted side and
- an unsorted side.



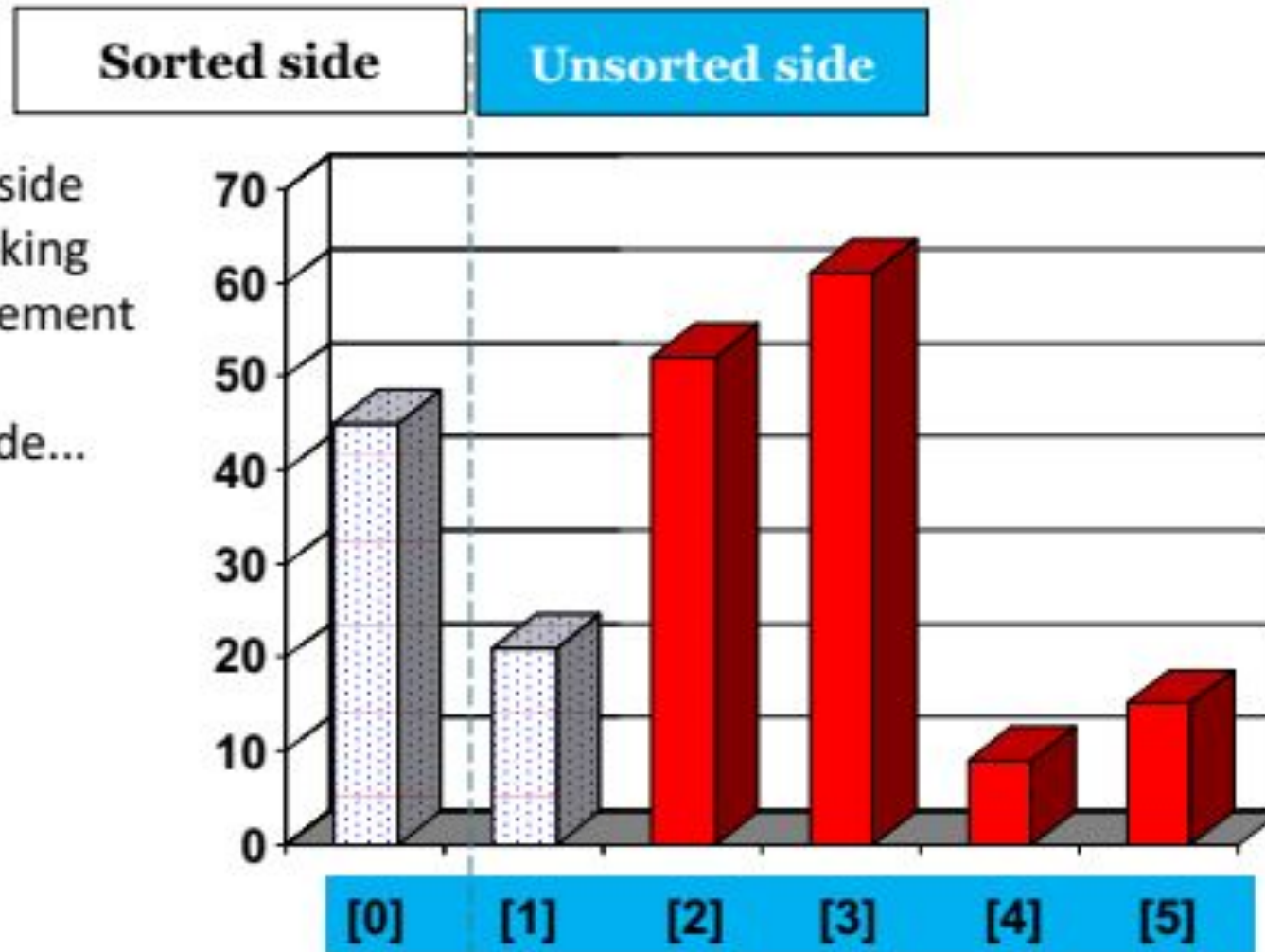
Insertion Sort

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



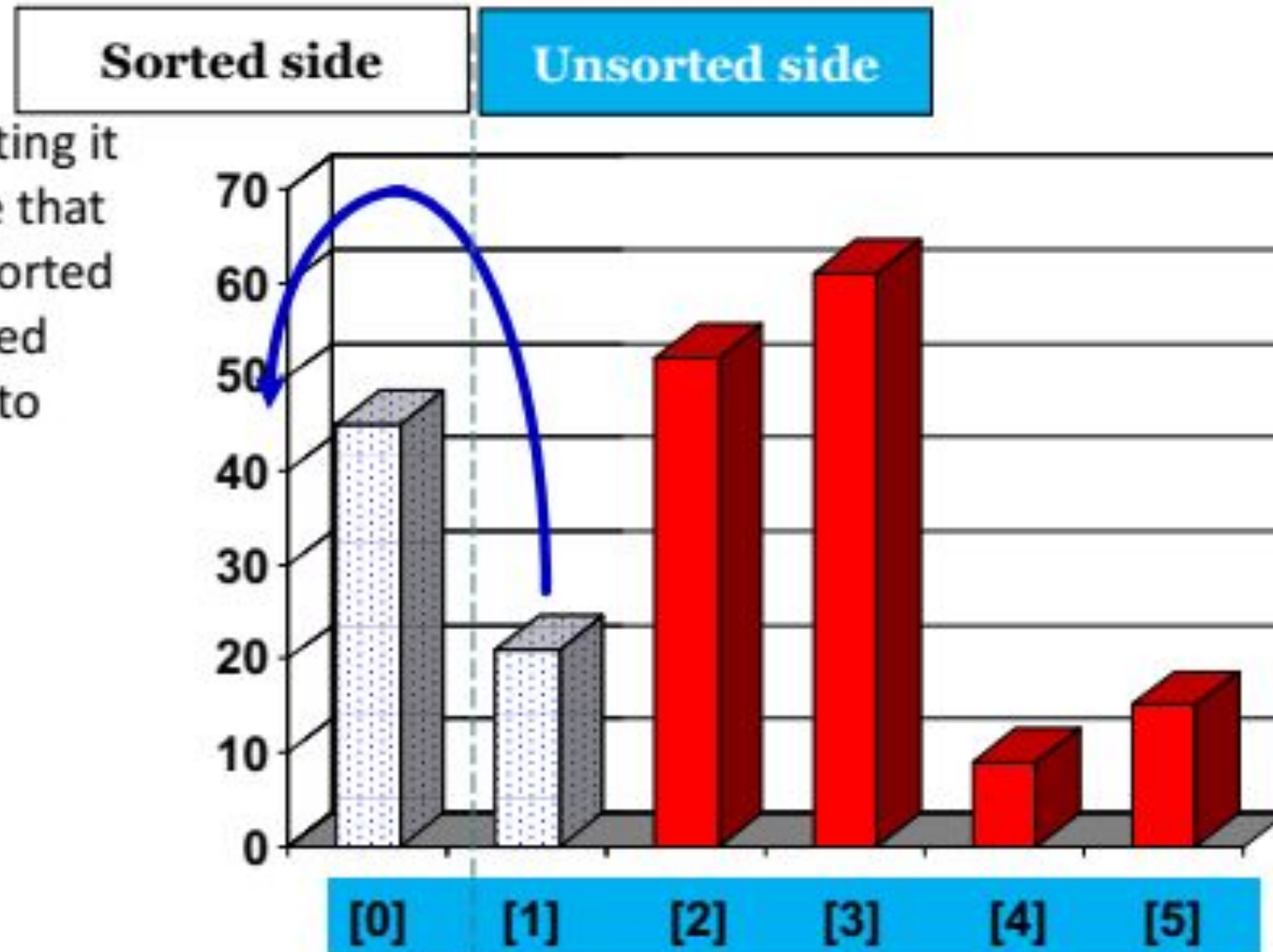
Insertion Sort

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



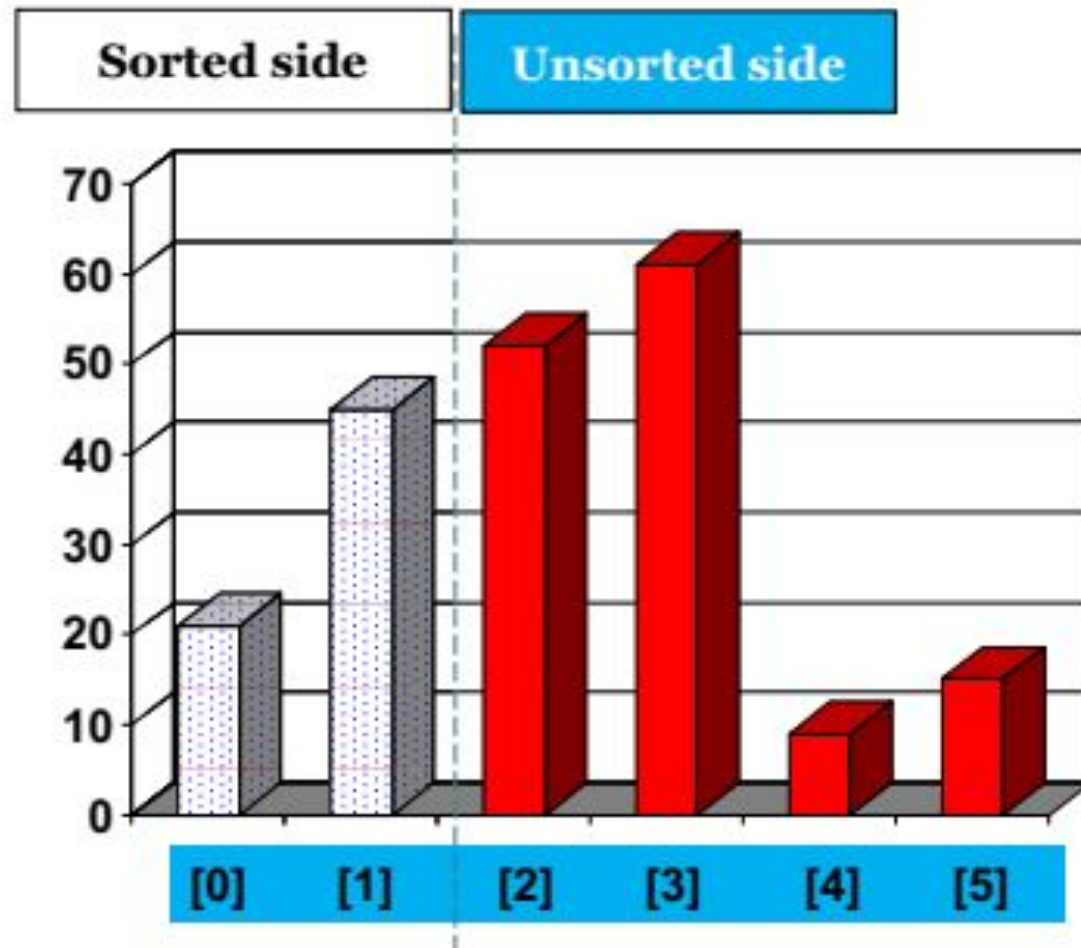
Insertion Sort

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



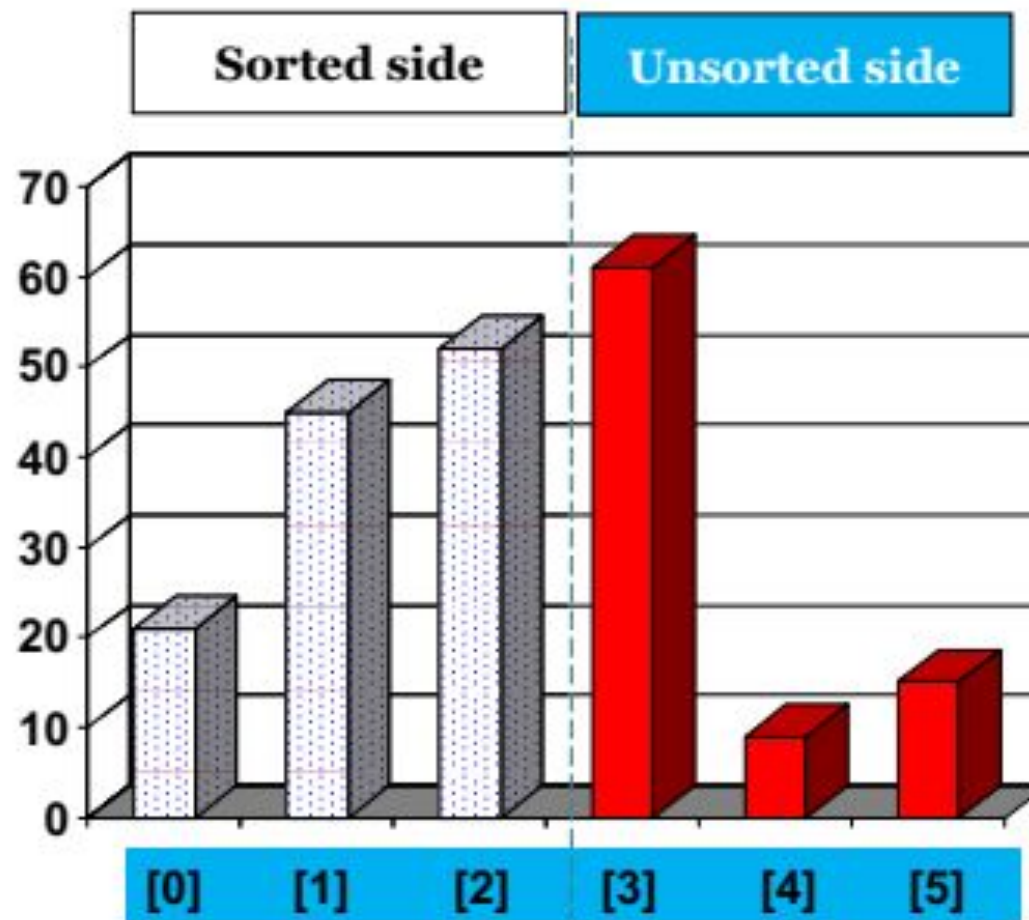
Insertion Sort

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



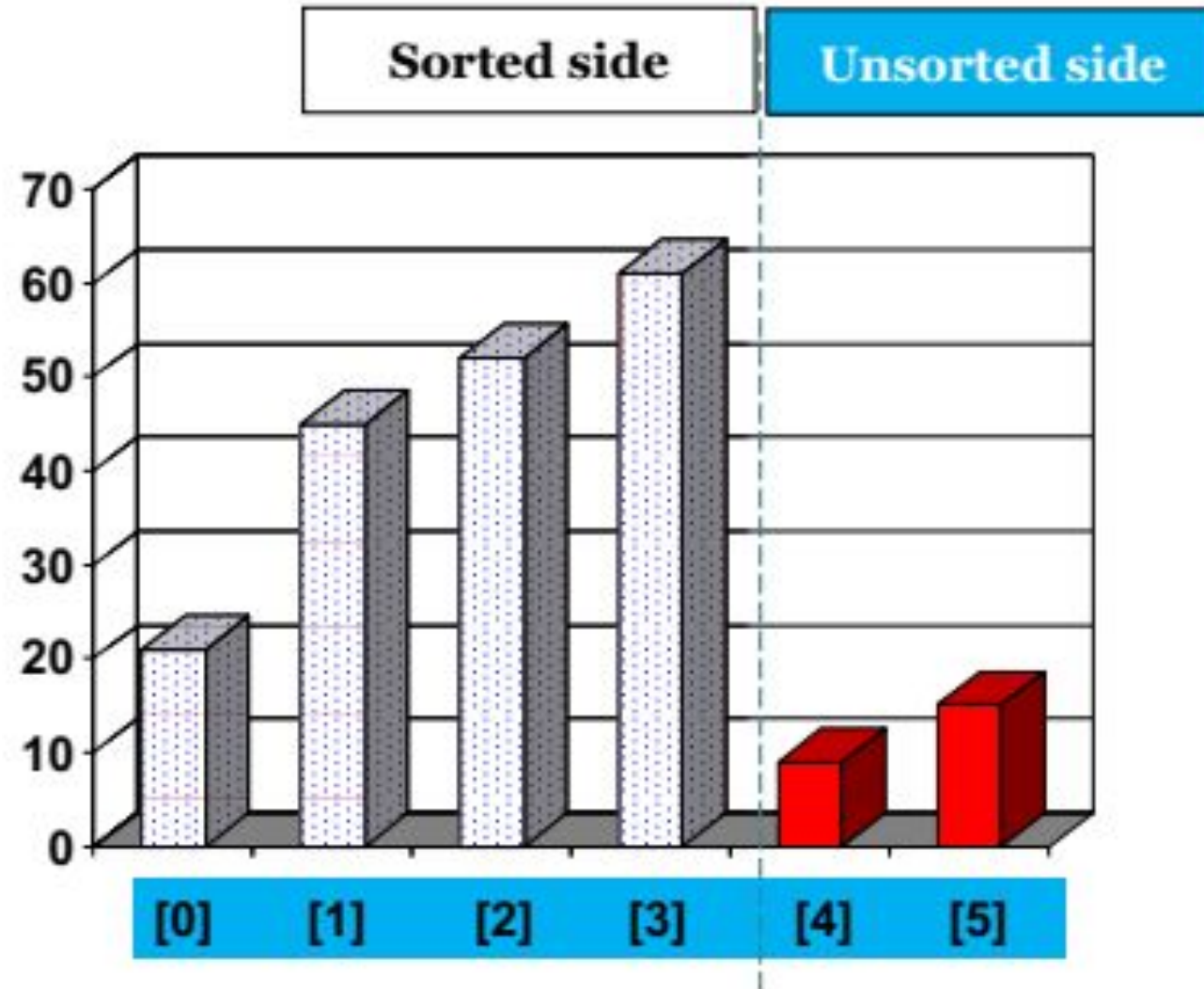
Insertion Sort

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



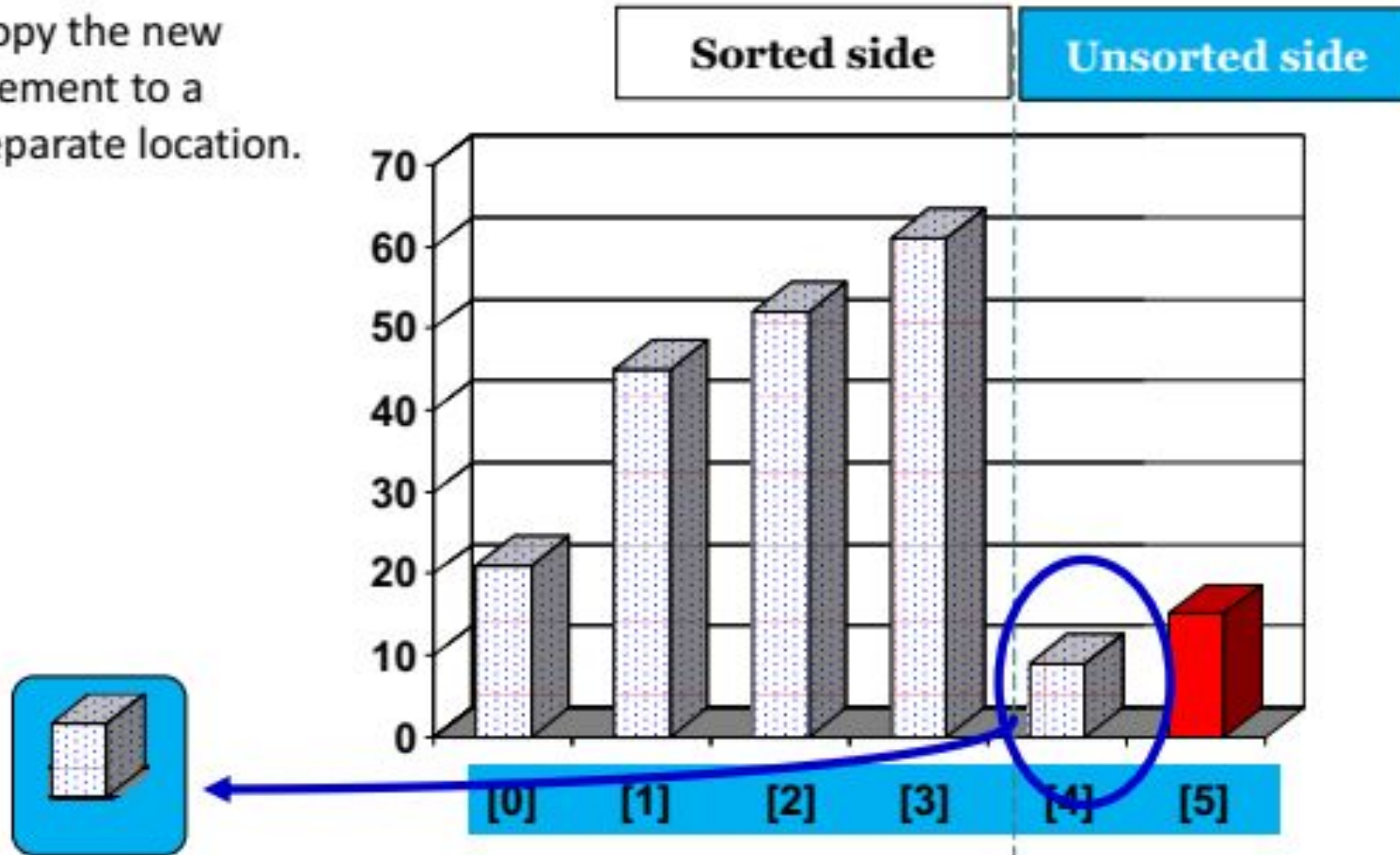
Insertion Sort

- Sometimes we are lucky twice in a row.



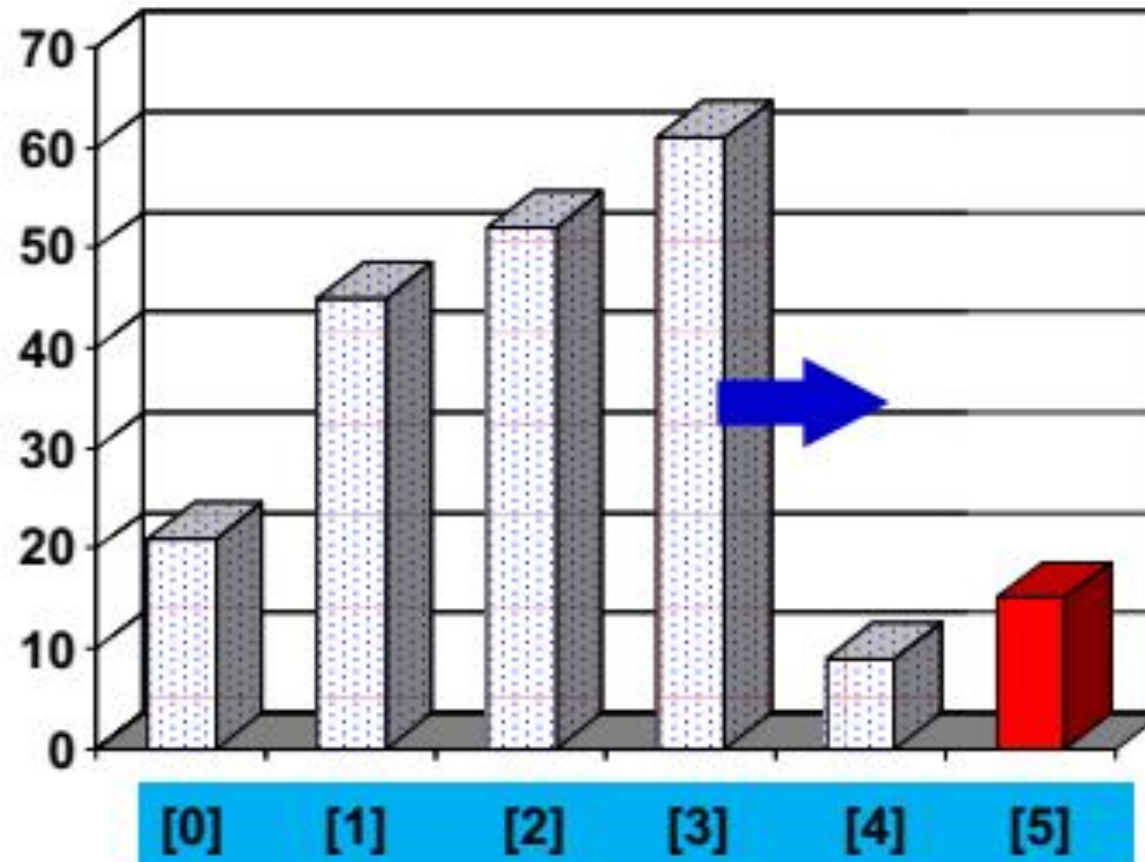
Insertion Sort

☆ Copy the new element to a separate location.



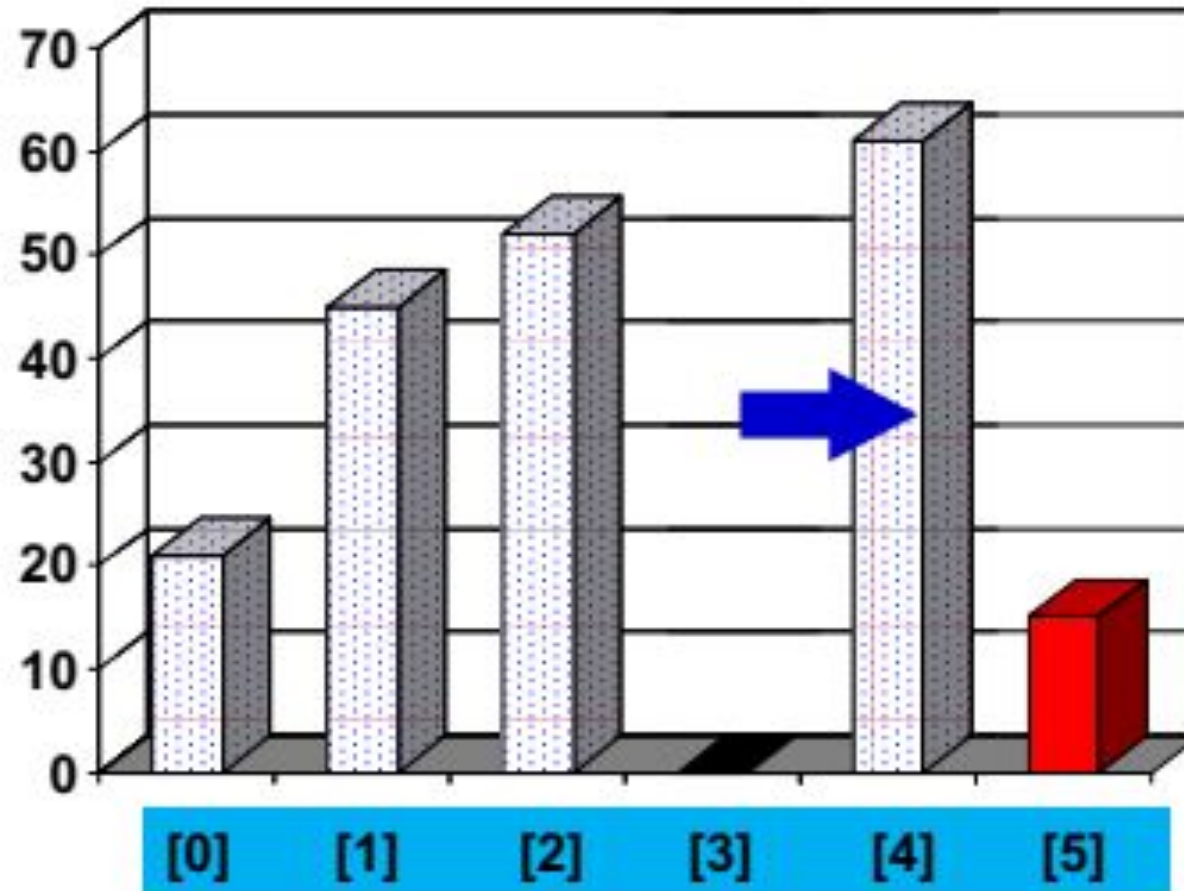
Insertion Sort

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



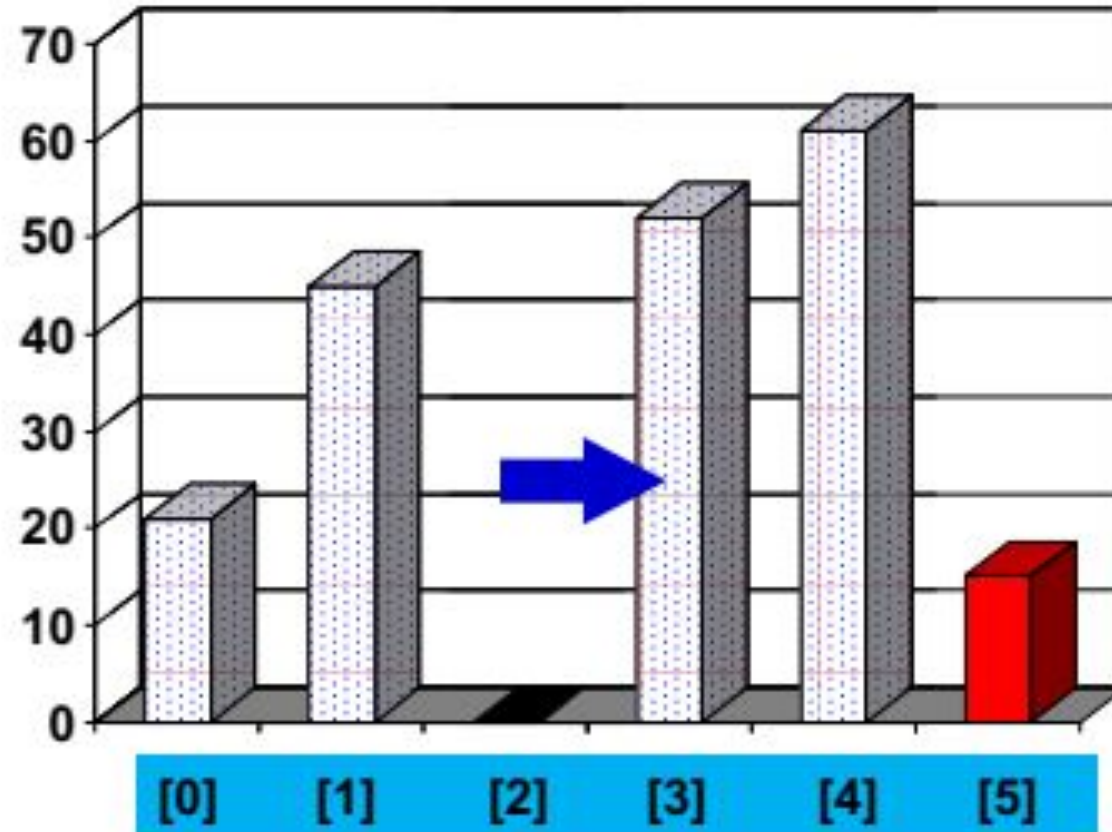
Insertion Sort

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



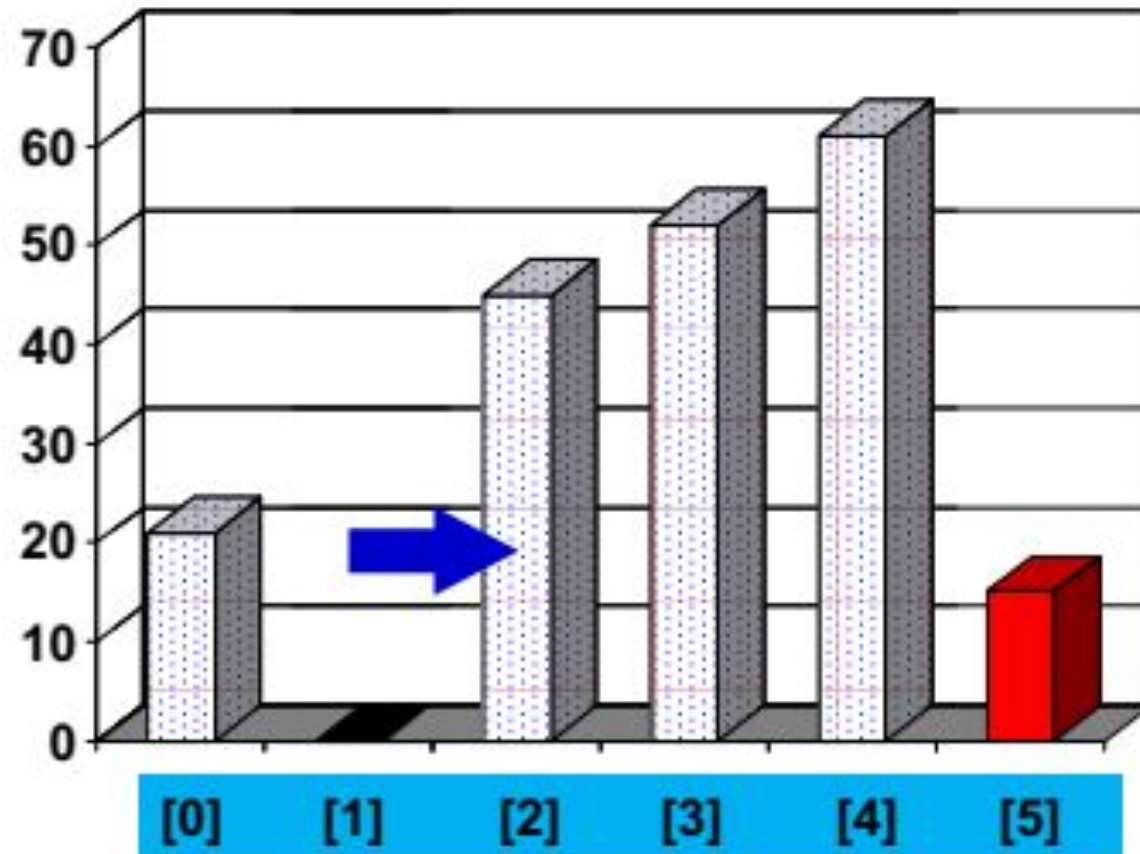
Insertion Sort

⌚ Continue shifting elements...



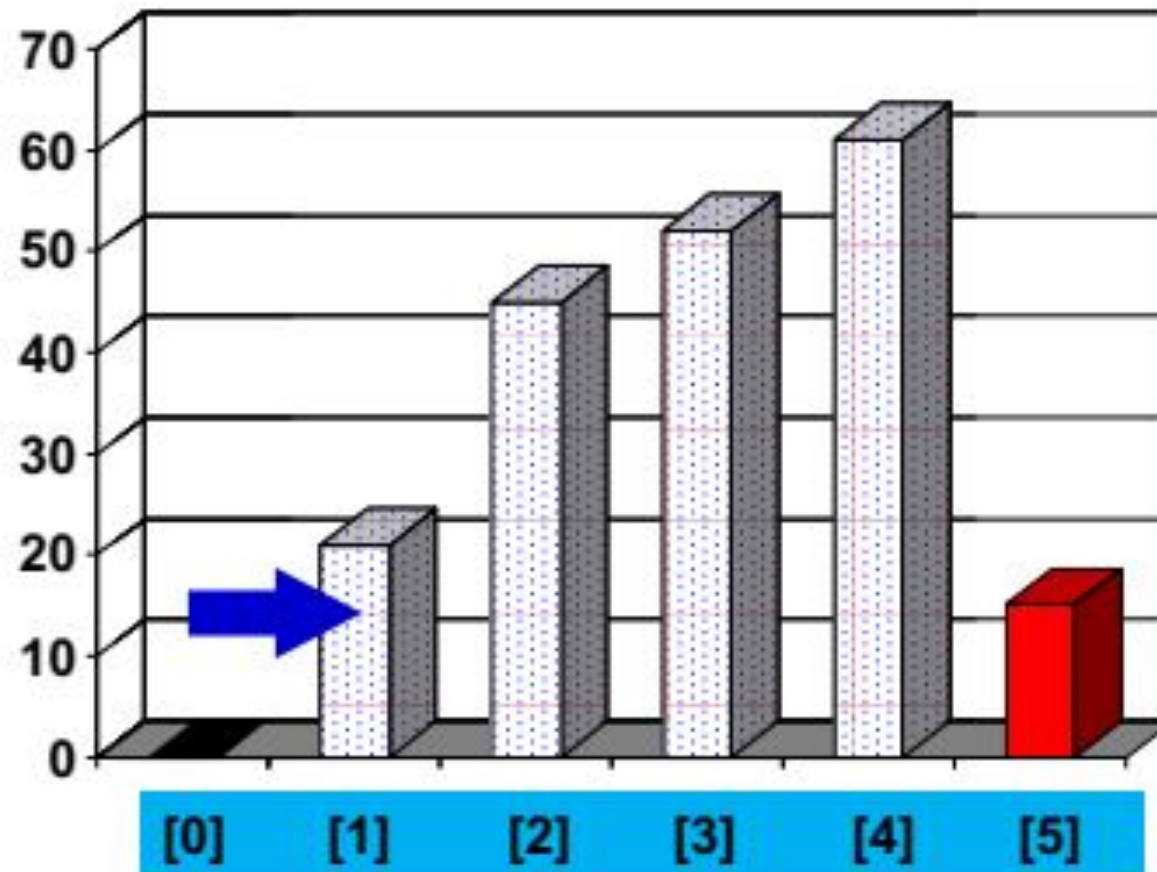
Insertion Sort

⌚ Continue shifting elements...



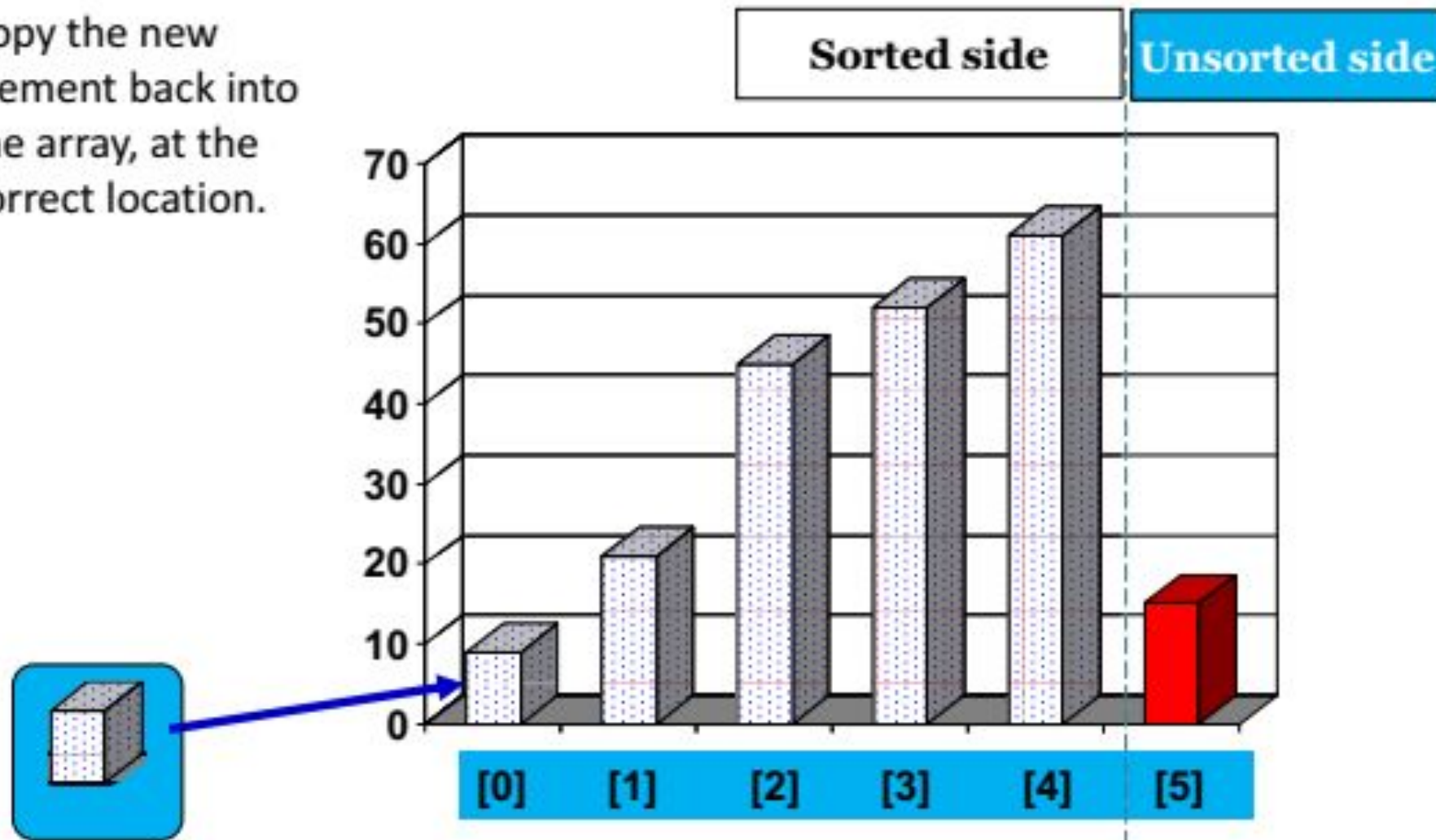
Insertion Sort

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



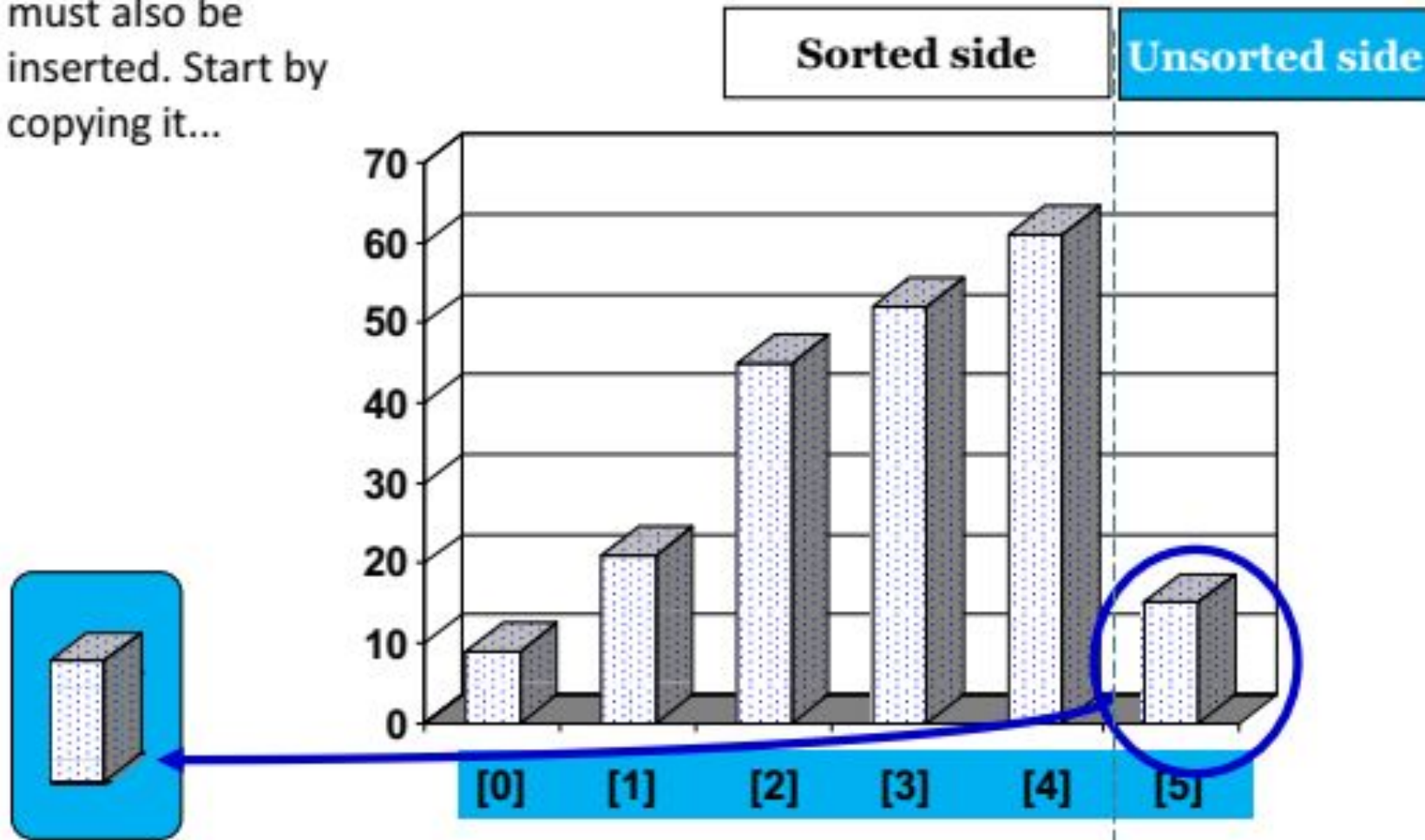
Insertion Sort

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



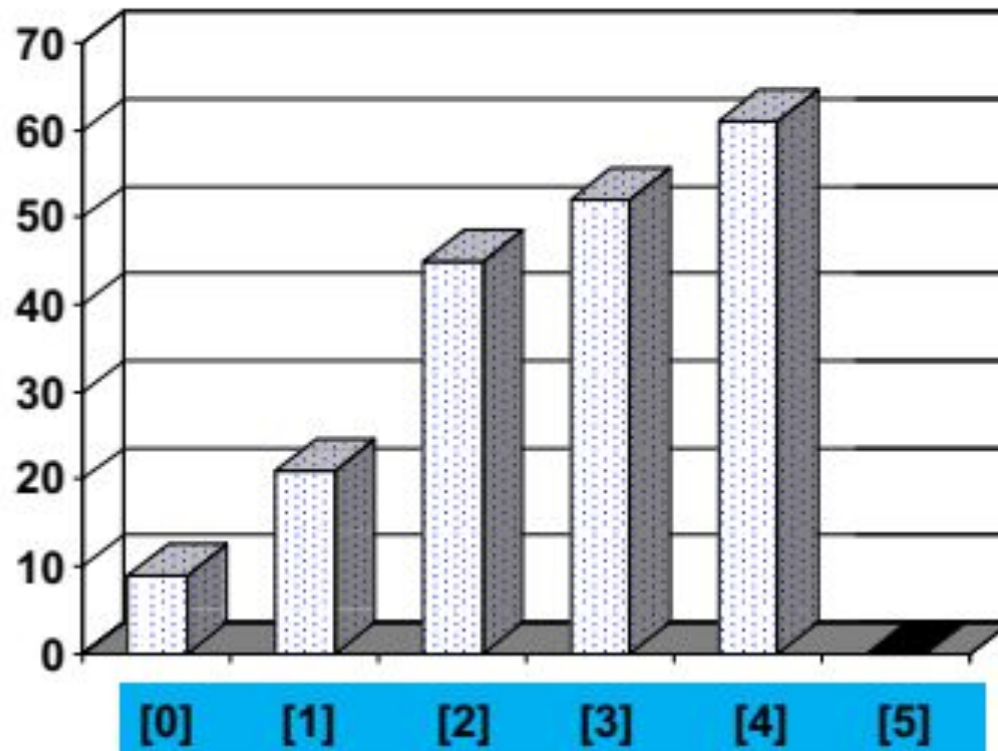
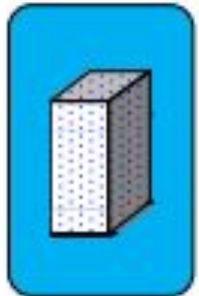
Insertion Sort

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



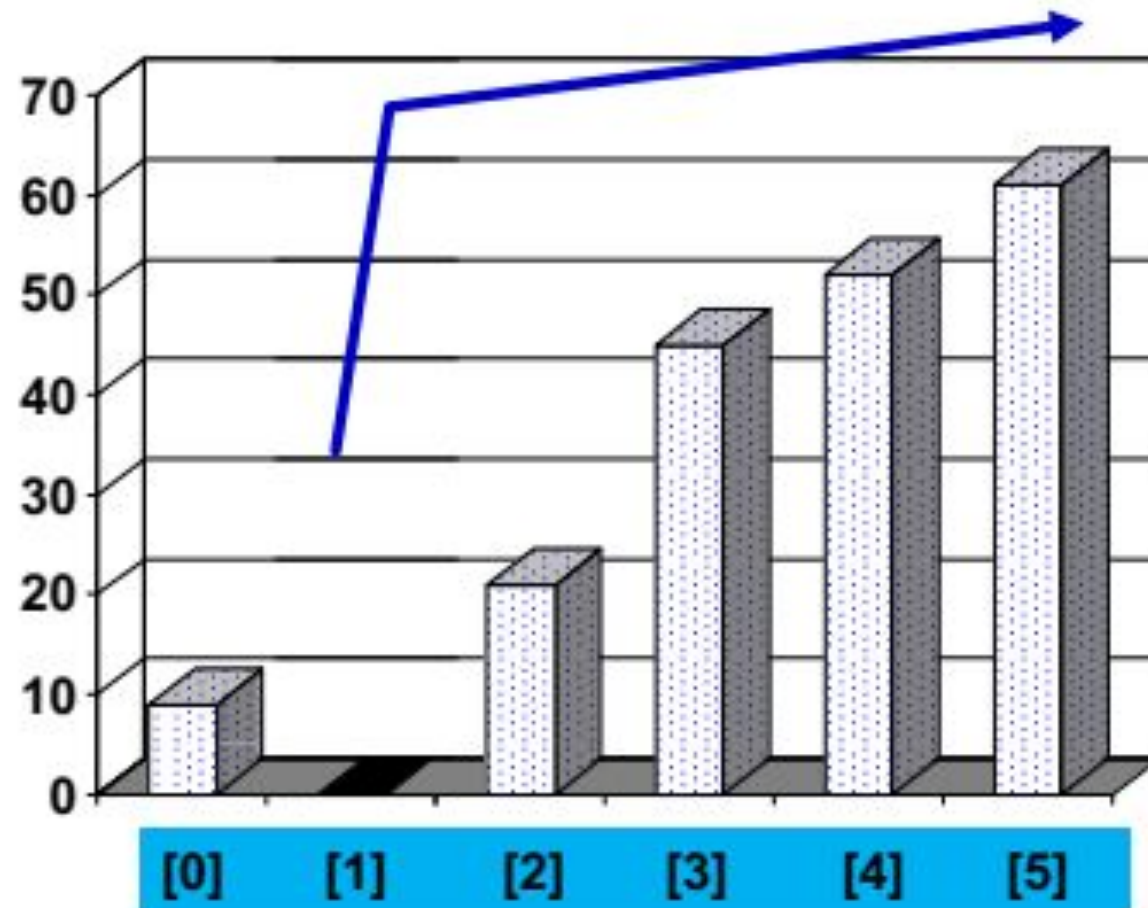
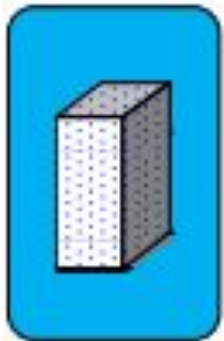
Insertion Sort

How many shifts
will occur before
we copy this
element back into
the array?



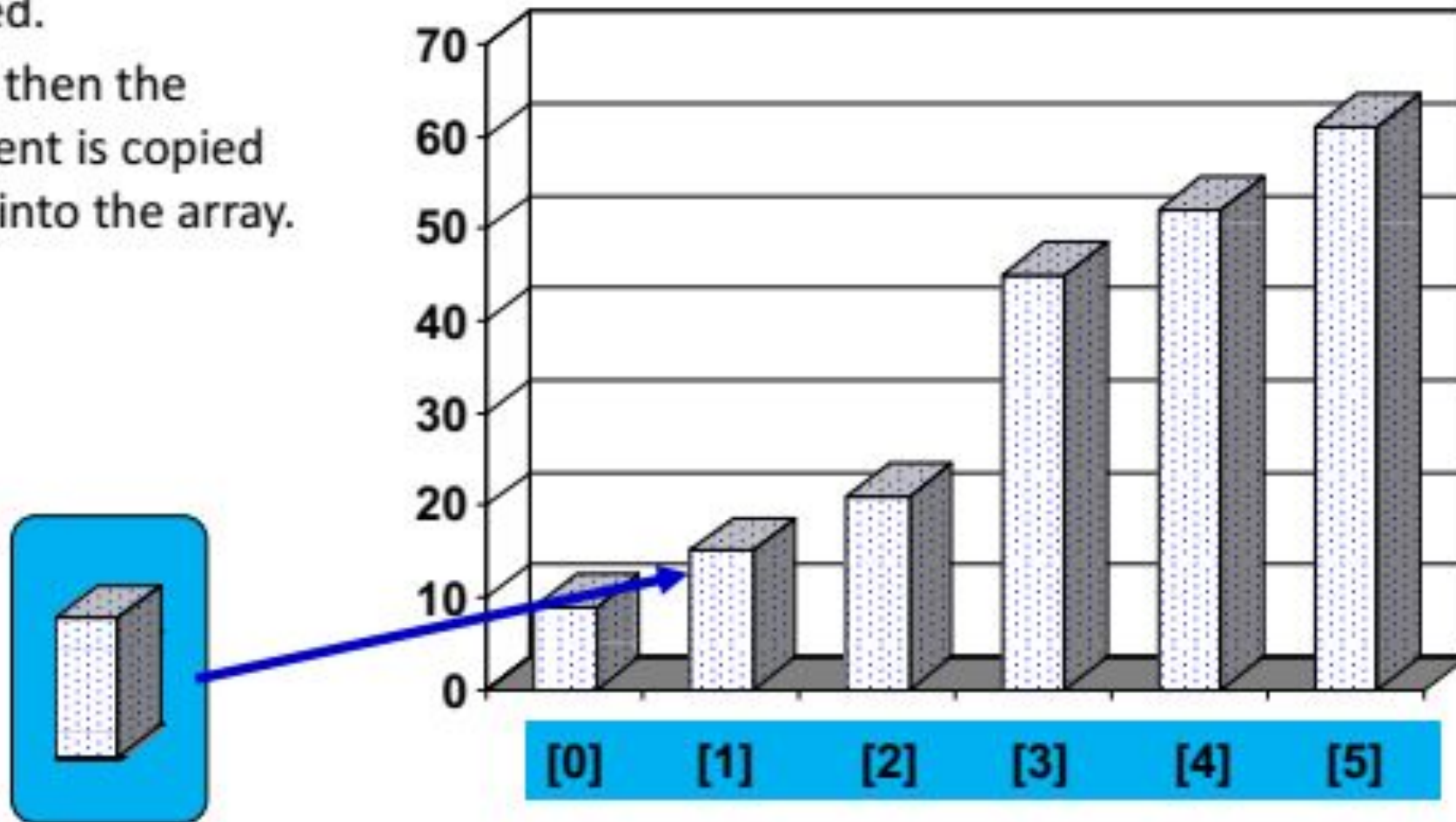
Insertion Sort

- Four items are shifted.



Insertion Sort

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



Insertion Sort

- To sort an array with k elements, Insertion sort requires $k - 1$ passes.
- Example:

	pass1	pass2	pass3	pass4	pass5	pass6	pass7
40	40	25	25	25	25	15	15
82	25	40	33	33	15	25	25
25	82	33	40	15	33	33	33
98	33	54	15	40	40	40	40
33	54	15	40	40	40	40	40
54	15	40	54	54	54	54	54
15	40	82	82	82	82	82	82
40	98	98	98	98	98	98	98

Insertion Sort

Input: An array $A[1..n]$ of n elements.

Output: $A[1..n]$ sorted in nondecreasing order.

```
1. for  $i \leftarrow 1$  to  $n$ 
2.    $x \leftarrow A[i]$ 
3.    $j \leftarrow i$ 
4.   while  $(j > 0)$  and  $(A[j - 1] > x)$ 
5.      $A[j] \leftarrow A[j - 1]$ 
6.      $j \leftarrow j - 1$ 
7.   end while
8.    $A[j] \leftarrow x$ 
9. end for
```

Insertion Sort

```
void InsertionSort(int list[], int size){  
    int i,j,k,temp;  
    for(i=1;i < size;i++) {  
        temp=list[i];  
        j=i;  
        while((j > 0)&&(temp < list[j-1])) {  
            list[j]=list[j-1];  
            j--;  
        } // end while  
        list[j]=temp;  
    } // end for loop  
} // end function
```

Complexity of Insertion Sort

- Let a_0, \dots, a_{n-1} be the sequence to be sorted. At the beginning and after each iteration of the algorithm the sequence consists of two parts: the first part a_0, \dots, a_{i-1} is already sorted, the second part a_i, \dots, a_{n-1} is still unsorted (i in $0, \dots, n$).
- The worst case occurs when in every step the proper position for the element that is inserted is found at the beginning of the sorted part of the sequence.

Complexity of Insertion Sort

The minimum # of element comparisons (**best case**) occurs when the array is already sorted in nondecreasing order. In this case, the # of element comparisons is exactly $n - 1$, as each element $A[i]$, $2 \leq i \leq n$, is compared with $A[i - 1]$ only.

The maximum # of element comparisons (**Worst case**) occurs if the array is already sorted in decreasing order and all elements are distinct. In this case, the number is

$$\sum_{i=2}^n (i - 1) = \sum_{i=1}^{n-1} (i) = n(n-1)/2$$

This is because each element $A[i]$, $2 \leq i \leq n$ is compared with each entry in subarray $A[1 .. i-1]$

- **Pros:** Relatively simple and easy to implement.
- Cons:** Inefficient for large lists.

Complexity of Insertion Sort

- **Best case:** $O(n)$. It occurs when the data is in sorted order. After making one pass through the data and making no insertions, insertion sort exits.
- **Average case:** $\theta(n^2)$ since there is a wide variation with the running time.
- **Worst case:** $O(n^2)$ if the numbers were sorted in reverse order.

Complexity of Insertion Sort

- Best case performance $O(n)$
- Average case performance $O(n^2)$
- Worst case performance $O(n^2)$
- Where n is the number of elements being sorted
 - | $\approx n^2/2$ comparisons and exchanges

Comparison Bubble and Insertion Sort

- Bubble sort is **asymptotically equivalent** in running time **$O(n^2)$** to insertion sort in the worst case
- But the two algorithms differ greatly in the number of swaps necessary
- Experimental results have also shown that insertion sort performs considerably better even on random lists.
- For these reasons many modern algorithm textbooks avoid using the bubble sort algorithm in favor of insertion sort.

Comparison Bubble and Insertion Sort

- Bubble sort also interacts poorly with modern CPU hardware. It requires
 - at least **twice** as many **writes** as insertion sort,
 - **twice** as many **cache misses**, and
 - asymptotically **more** branch mispredictions.
- Experiments of sorting strings in Java show bubble sort to be
 - roughly **5** times slower than **insertion sort** and
 - **40%** slower than **selection sort**

Comparison of Sorts

	Bubble	Selection	Insertion
Best Case	$O(n)$	$O(n^2)$	$O(n)$
Average Case	$O(n^2)$	$O(n^2)$	$O(n^2)$
Worst Case	$O(n^2)$	$O(n^2)$	$O(n^2)$
Space complexity	$O(1)$	$O(1)$	$O(1)$