ICS 141 4/1 Programming with Objects

Jessica Maistrovich

Metropolitan State University

Searching and **Sorting**

- Searching and sorting are fundamental problems in computer science and programming.
- Sorting is usually done first to make searching more efficient.
- There are multiple algorithms to search or sort.

Searching

Introduction to Searching

- Given a list of data, the goal of the **search** algorithm is to find the location of a particular value or report that value is not found in the given list.

- Two possible implementations:
  - return the position of the value if found and -1 otherwise

```
public static int search (int[] data, int target)
```

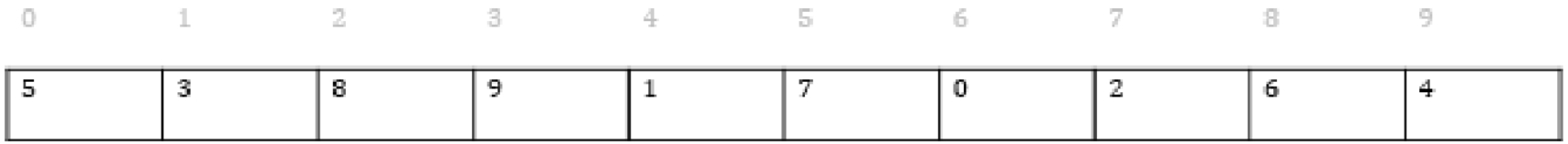
  - return `true` if found and `false` otherwise

```
public static boolean search (int[] data, int val)
```

Linear Search

- Also called, sequential or serial search
- Intuitive approach
- Must be used when the list is not sorted
- Linear search algorithm:
  - start at first item
  - is it the one I am looking for?
  - if not go to next item
  - repeat until found or all items checked

Linear search example



In the above array, how many comparisons would you make using linear search to find 7?

How many comparisons would you do for best case scenario?

How many comparisons in the worst case scenario?

Number of comparisons performed by Linear Search

- For an array of `n` elements:
  - the average case time for linear search requires (n+1)/2 comparisons
  - the best case time is 1 comparison (when the element is found at position 0)
  - The worst case time is n when the element is not found in the array.

Linear Search Implementation

```
//return the index of the first occurrence
//of target in data or -1 if target not found in  data

public int linearSearch(someDataType[] data, someDataType target){
    for(int i = 0; i < data.length; i++){
        if( data[i] == target ){
            return i;
        )
    }
    return -1;
}
```

Try it!

- Create the following three methods in the AnimalShelter class:
  - contains(): takes one input of type Dog. It returns true if that animal is in the shelter, and false if it is not. Remember to use .equals to determine if two dogs are equal.
  - cageNumber(): takes one input of type Dog. Returns the cage number (position in the array) of that dog if it is in the shelter, returns -1 otherwise.
  - minAge(): Returns the age of the youngest dog.
- In the driver class, create two Animals. Add one to the AnimalShelter. Call the contains method twice – once for each of the Animals you created. Print the results. What should they be?
- In the driver class, test the other two methods. How will you do this?

Binary Search

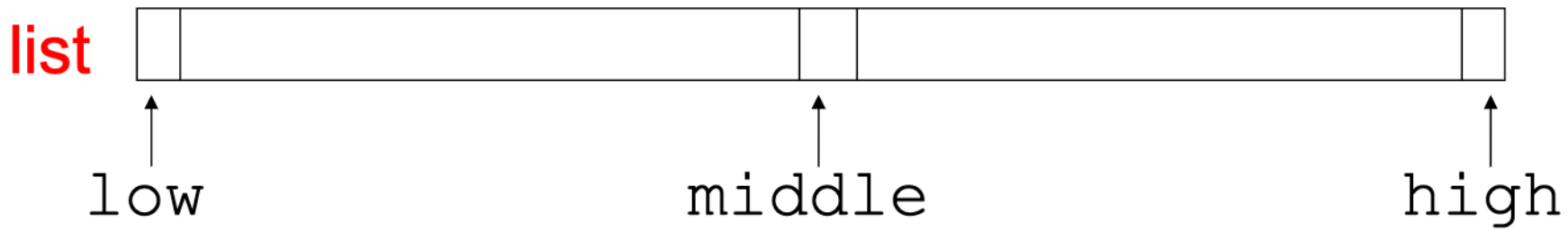
Searching in a sorted list: Binary Search

- Works only on **SORTED** arrays.

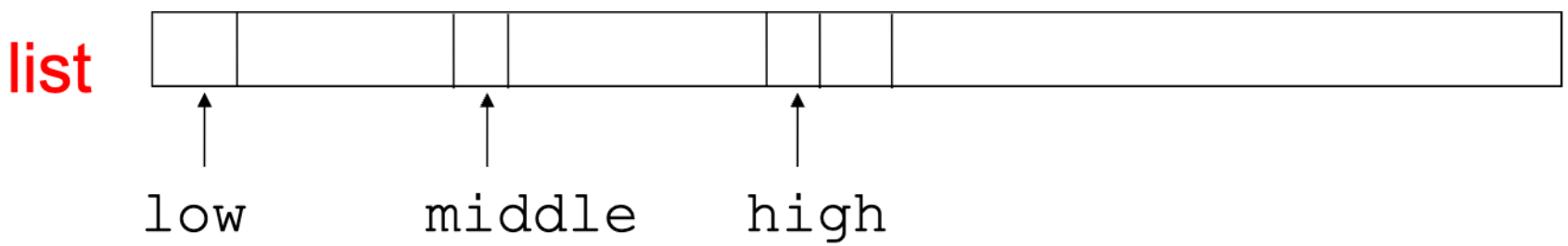
- Uses the *divide and conquer method in which you divide the* work in half with each step
  - generally a good thing

- Assume, the list is sorted in ascending order:
  - Start at the middle element in list
  - is that middle element equals to *target*?
  - If not, the check whether the middle element is less than or greater than *target*?
    - If less than, then move to second half of list
    - If greater than, move to first half of list
  - repeat until *target* is found or sub-list size = 0

Binary Search

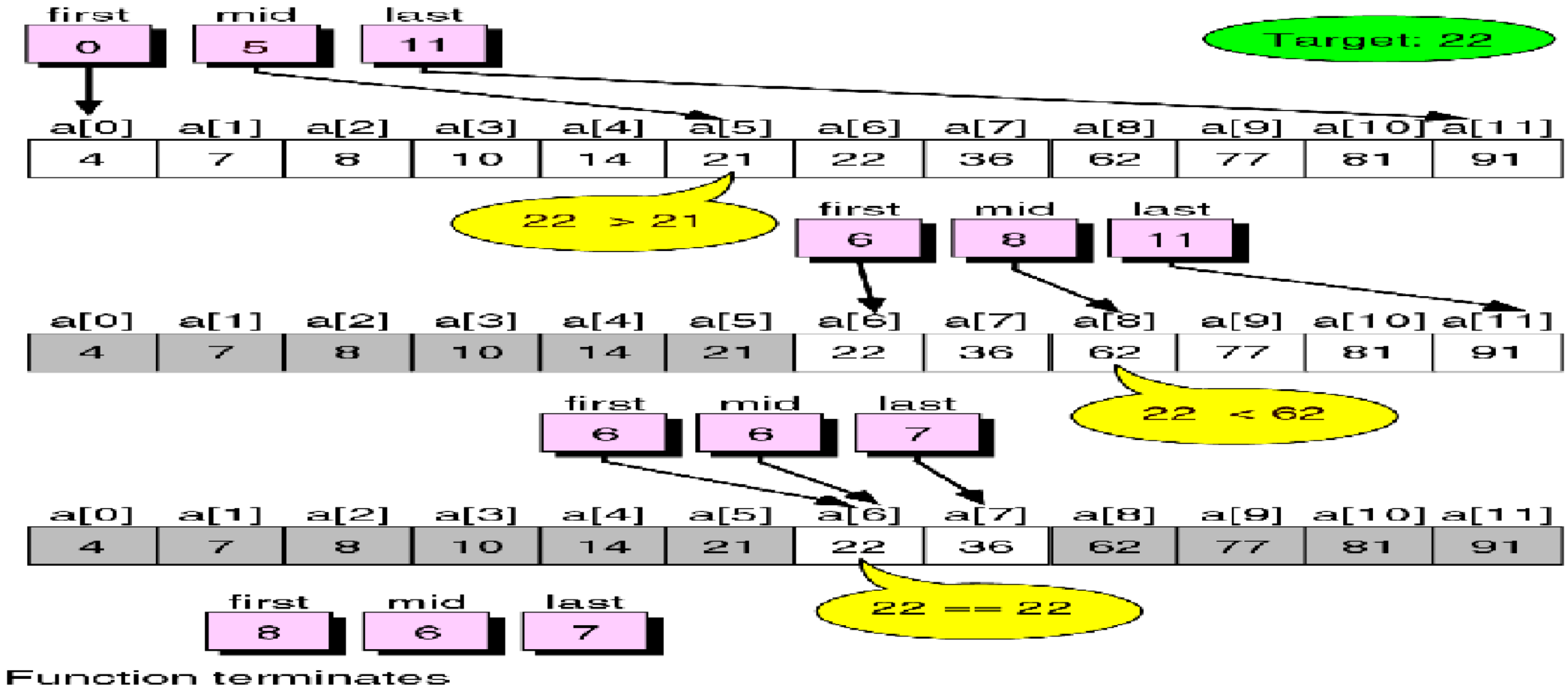


Is the *middle* item equals to *target*? If not, then check whether it is greater than or less than *target*?
If the *middle* is greater than *target*, then the next *middle* is the middle element of the sub-list before current middle



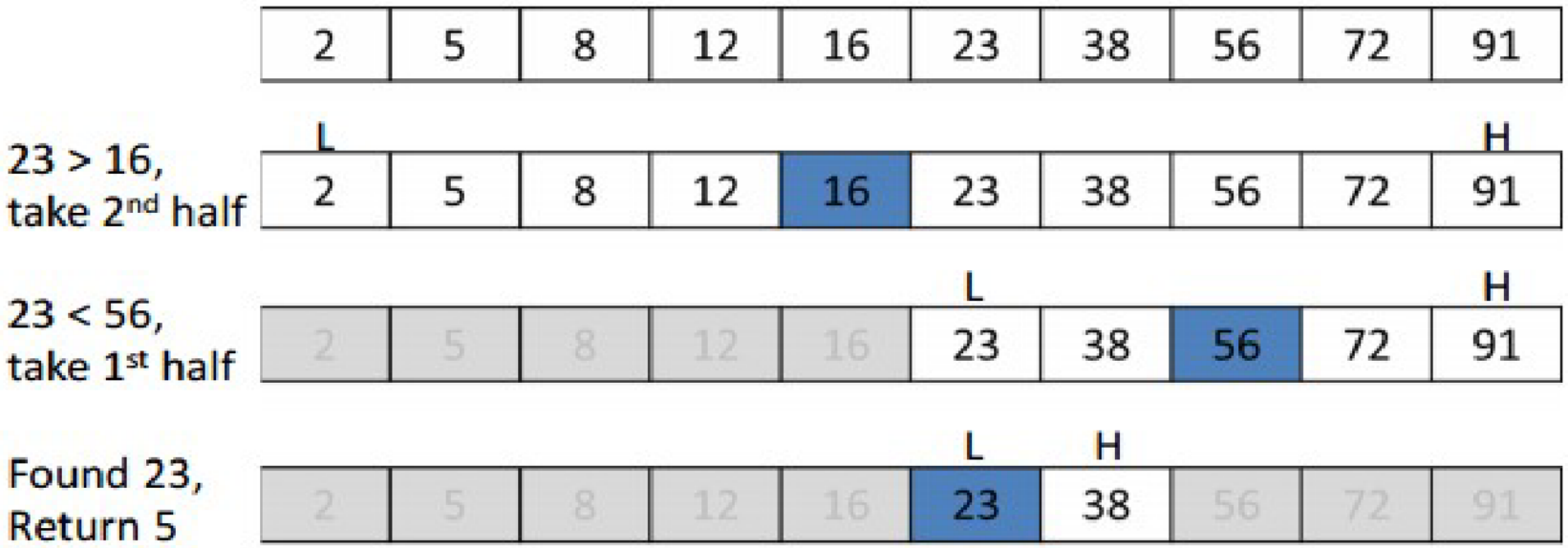
and so forth…

Example 1 on Binary Search



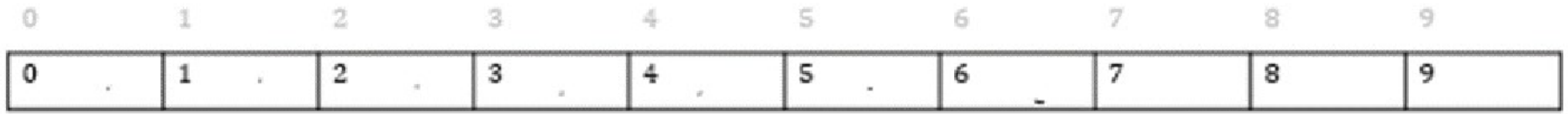
Example 2 on Binary Search

If searching for 23 in the 10-element array:



Binary search example

Please use the formula mid = (high + low)/2 for the binary search



In the above array, how many comparisons would you make using linear search to find 7?

In the above array, how many comparisons would you make using binary search to find 7?

How many comparisons would you do for best case scenario?

How many comparisons in the worst case scenario?

Number of comparisons performed by Binary Search

- What is the maximum number of operations that are done to search in an array `data` of size `n`?
  - If n = 8 ⇒ 3 comparisons
  - If n = 16 ⇒ 4 comparisons

- In every iteration of the binary search loop, the array size is halved

- The maximum number of comparisons is $\log_2 n$.

- **Binary** search performs $\log n$ operations

Pseudo Code of Binary search

```
middle = index of middle element in the array
if (target == data[middle])
   the target is found at position middle
else if (target < data[middle])
   search for the target in the array part before midpoint
else if (target > data[middle])
   Search for the target in the array part  after midpoint
```

Binary Search Implementation

```
public static int iterativeBinarySearch(someDataType[] data, someDataType
target){
    int result = -1;
    int low = 0;
    int high = data.length - 1;
    while( result == -1 && low <= high ){
        int middle = low + ((high - low) / 2);
        if( target == data[middle] ){
            result = middle;
        }else if(target > data[middle]){
            low = middle + 1;
        }else
            high = middle - 1;
                    }
    }
    return result;
}
```

Sorting

Searching and Sorting objects

- Linear search for objects is intuitive (just remember to use .equals)!
- Binary search needs to be sorted first
- Can we sort objects? What concept is needed in addition to equals?

Searching and Sorting objects

- Linear search for objects is intuitive (just remember to use .equals)!
- Binary search needs to be sorted first
- Can we sort objects? What concept is needed in addition to equals?

Comparing!!

Before we talk about comparing objects, we need some other concepts

Final Modifier

Can’t be replaced

Final Modifier

- For a variable
  - Can not be reassigned (after initialization)
- For a method
  - Can not be overridden
- For a class
  - Can not be extended

Abstract Modifier

Not real

Abstract Method

- Has only a method header

```
public abstract int get();
public abstract void print();
public abstract void find(int target);
```

Interface

Contract between the class and the interface. The class agrees to add functionality to the methods included in the interface.

Interface

- An interface is essentially a list of variables your class will have and methods your class will add functionality to.
- It has static final constants and abstract methods
- It is not a class – can not create objects from an interface
- An interface is implemented by a class

```
class Apple implements Comparable{
}
```

- A class can implement multiple interfaces (separate with commas)

```
class Apple implements Comparable, Cloneable{
}
```

Creating an Interface

```
public interface NameOfInterface
{
    // Any number of final, static fields
    // Any number of abstract method declarations
}
```

Interface

- Each field in an interface is implicitly public, static, and final. Therefore, we don’t need to use those modifiers.

- Methods in an interface are implicitly public and abstract. Therefore, we don’t need to use those modifiers.

Using generic class or interface

Generics

- Allow for reuse of code

Example:

Imagine that there is a class (or interface) called Suitcase<T>.

This means that when you order (instantiate) the suitcase, you must tell it what “type” of object it is going to be holding. You also can leave off the <> (angle brackets) and it will automatically be holding Objects. Once you have made the suitcase, it can’t hold anything other than what it was made for.

Comparable Interface

https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

`compareTo` method

- Similar to the `equals` method, the `compareTo` method is a special method that is to be defined for an object whenever the objects need to be sorted.

- The implementation of `compareTo` is application-dependent.

- The output of `compareTo` should be:
  - 0 if the two objects are equal
  - >0 if the this object > input object
  - <0 if the this object < input object

Try it!

- Implement Comparable for the Dog class and the Account class.
  - Can use id numbers for comparison to maintain consistency with .equals

Sorting Algorithms

- Sorting is a fundamental computer operation. It is important for preparing data for other operations.

- Often, data is much easier to process if it is arranged into sorted order.
  - For example, searching for a word in a dictionary.

- Basic sorting algorithms:
  - **Insertion sort**
  - **Selection sort**
  - Bubble sort
  - Merge sort
  - Quick sort
  - Heap sort

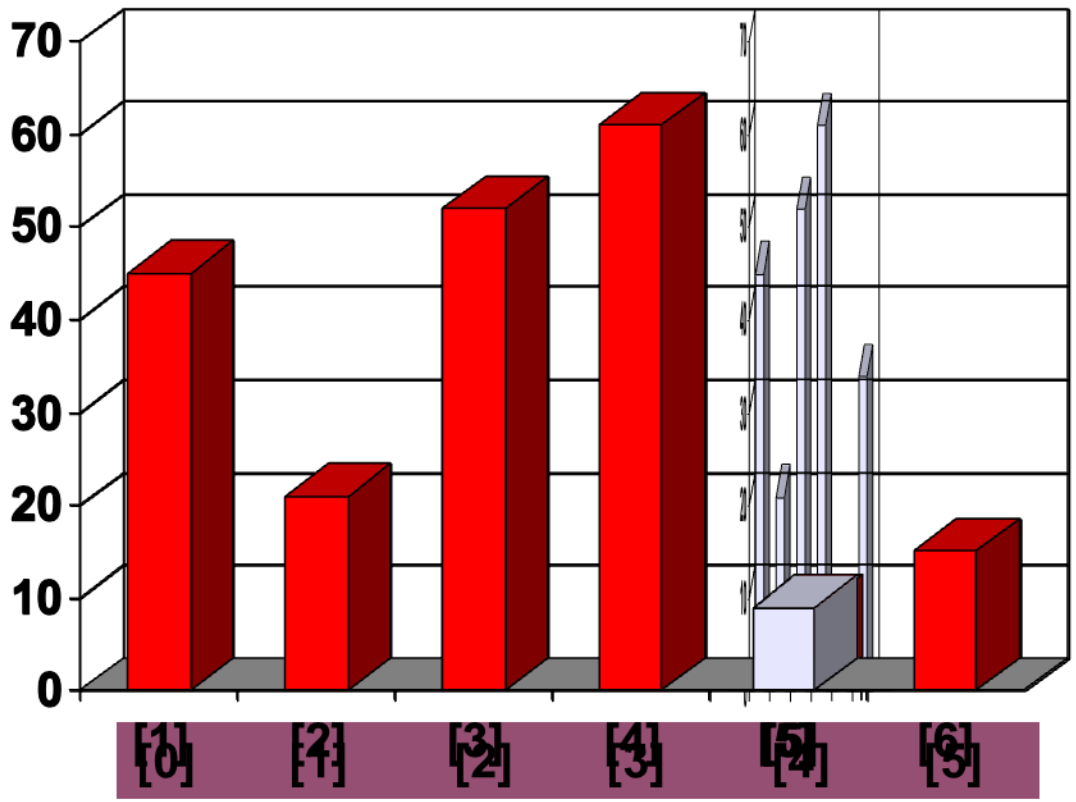
- Animation of sorting algorithms: https://visualgo.net/sorting

Selection Sort

“Selecting” the element that belongs in the current spot.

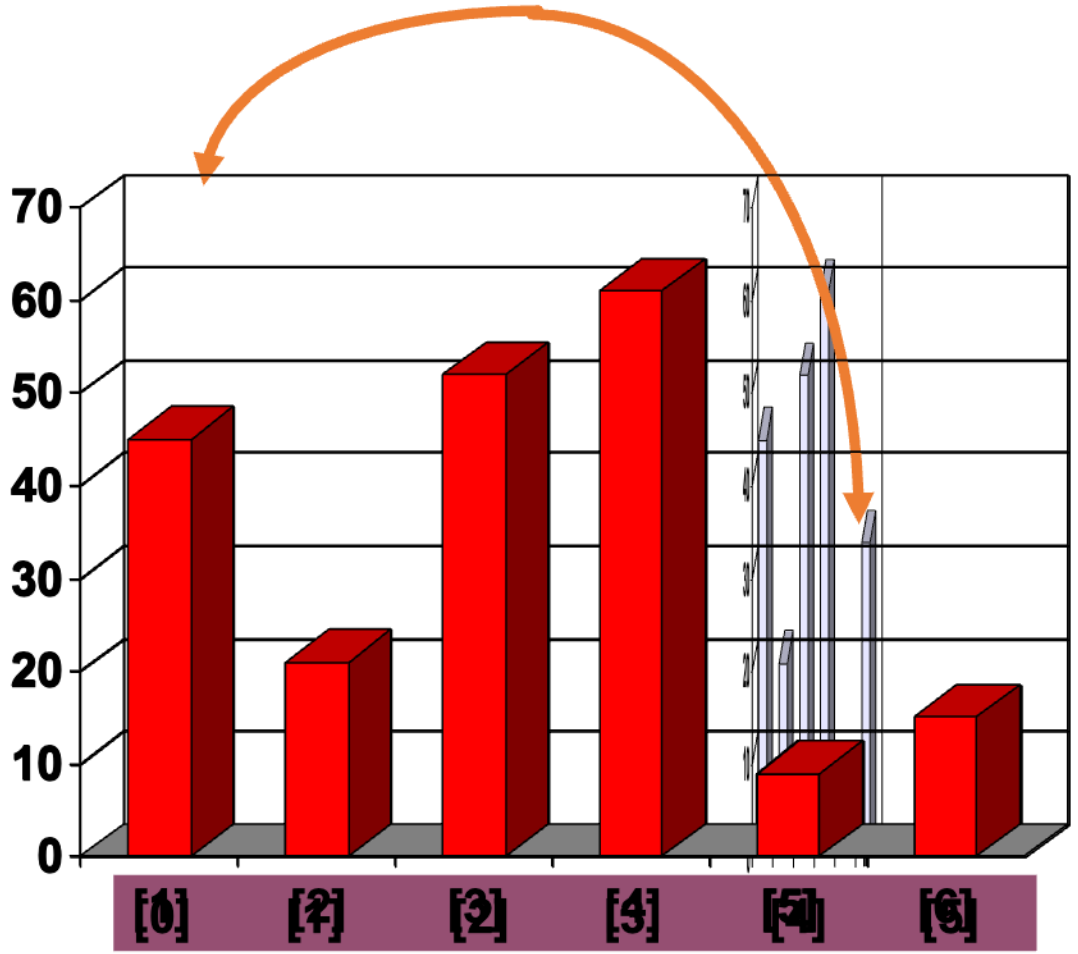
The `Selection sort` Algorithm (1)

- begins by going through the entire array and finding the smallest element.
- In this example, the smallest element is the number `8` at location `[4]` of the array



The `Selection sort` Algorithm (2)

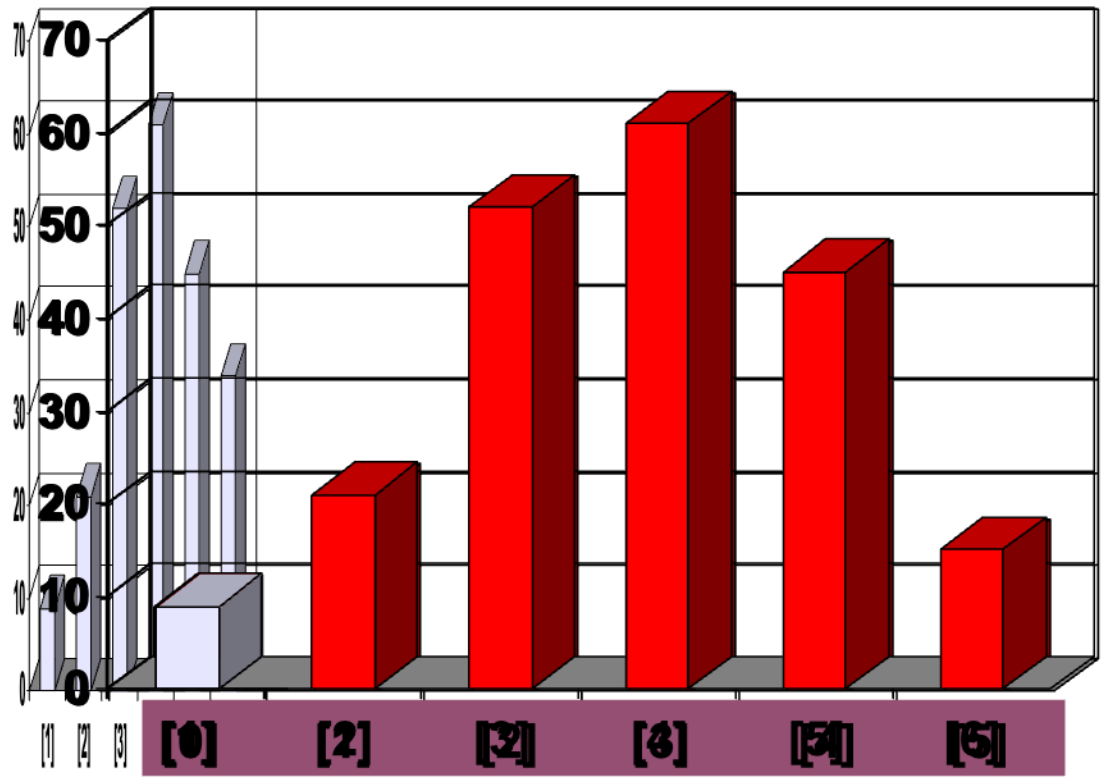
- Once we find the smallest element, that element is swapped with the first element of the array.



The `Selection sort` Algorithm (3)

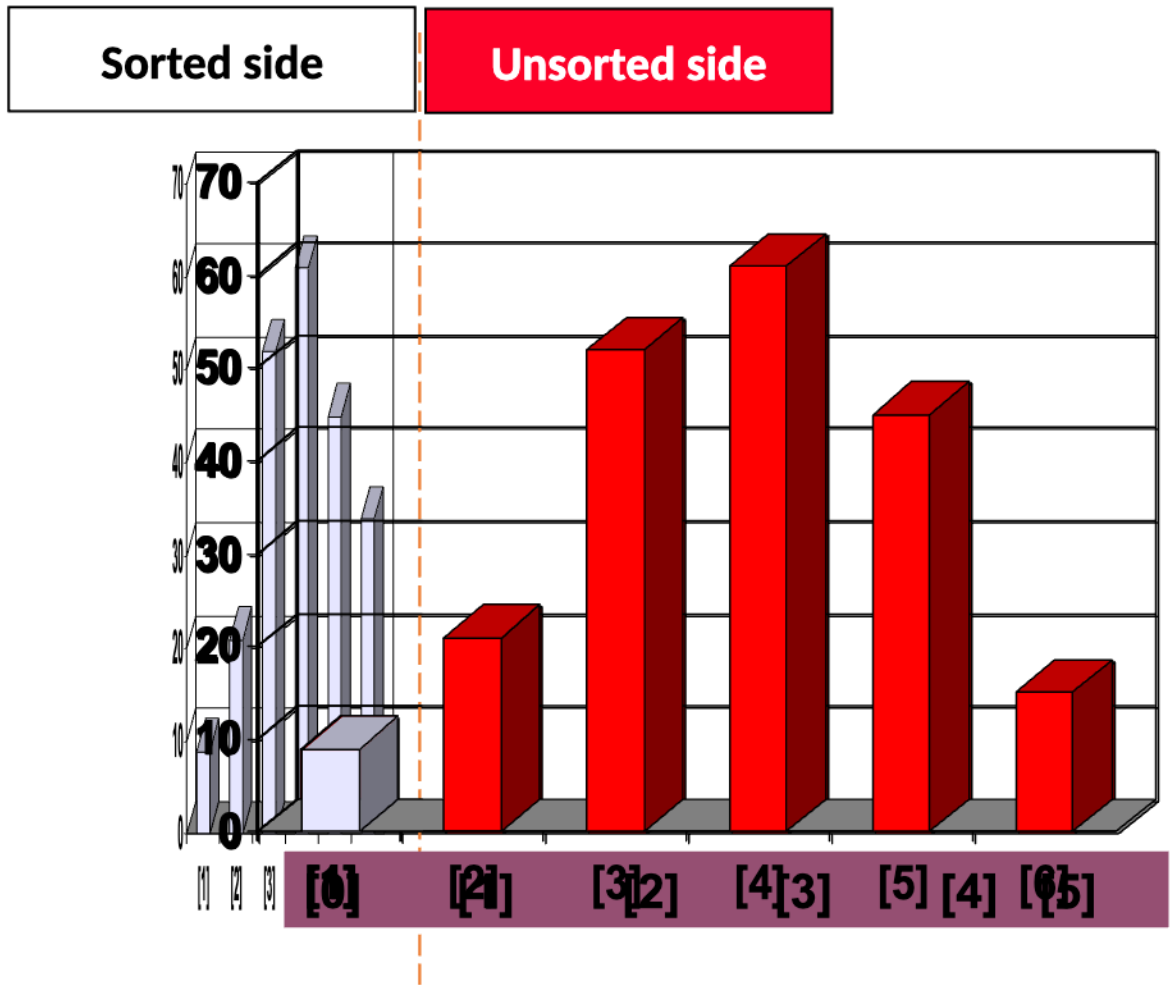
- After one iteration, we are sure that the smallest element in the array is in its final position (which is at index 0).

- After that, the same process is repeated to sort the array elements from position 1 to position n-1.



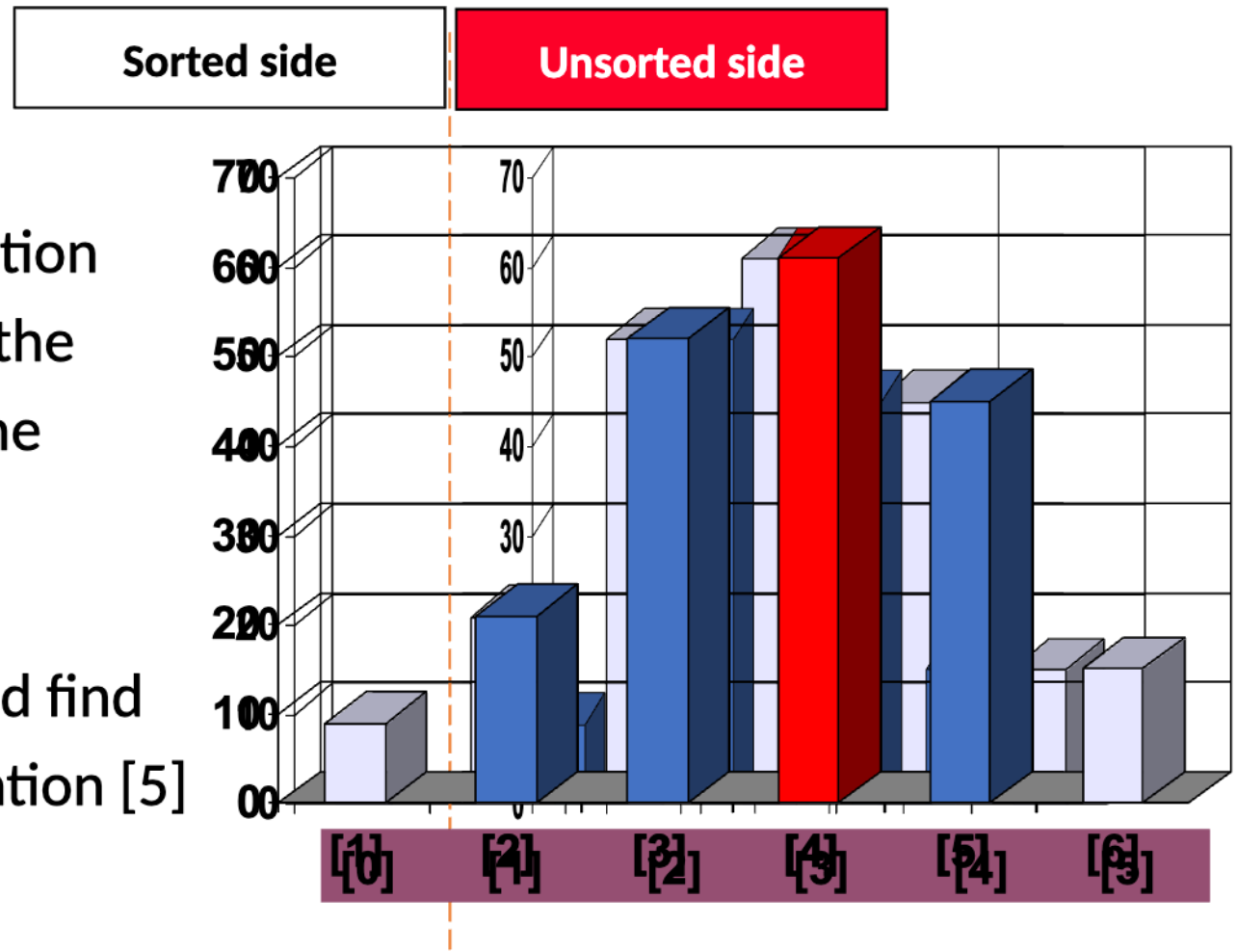
The `Selection sort` Algorithm (4)

- At this point, we can view the array as being split into two sides: To the left of the dotted line is the "sorted side", and to the right of the dotted line is the "unsorted side".
- The goal now is to push the dotted line forward, increasing the number of elements in the sorted side, until the entire array is sorted.



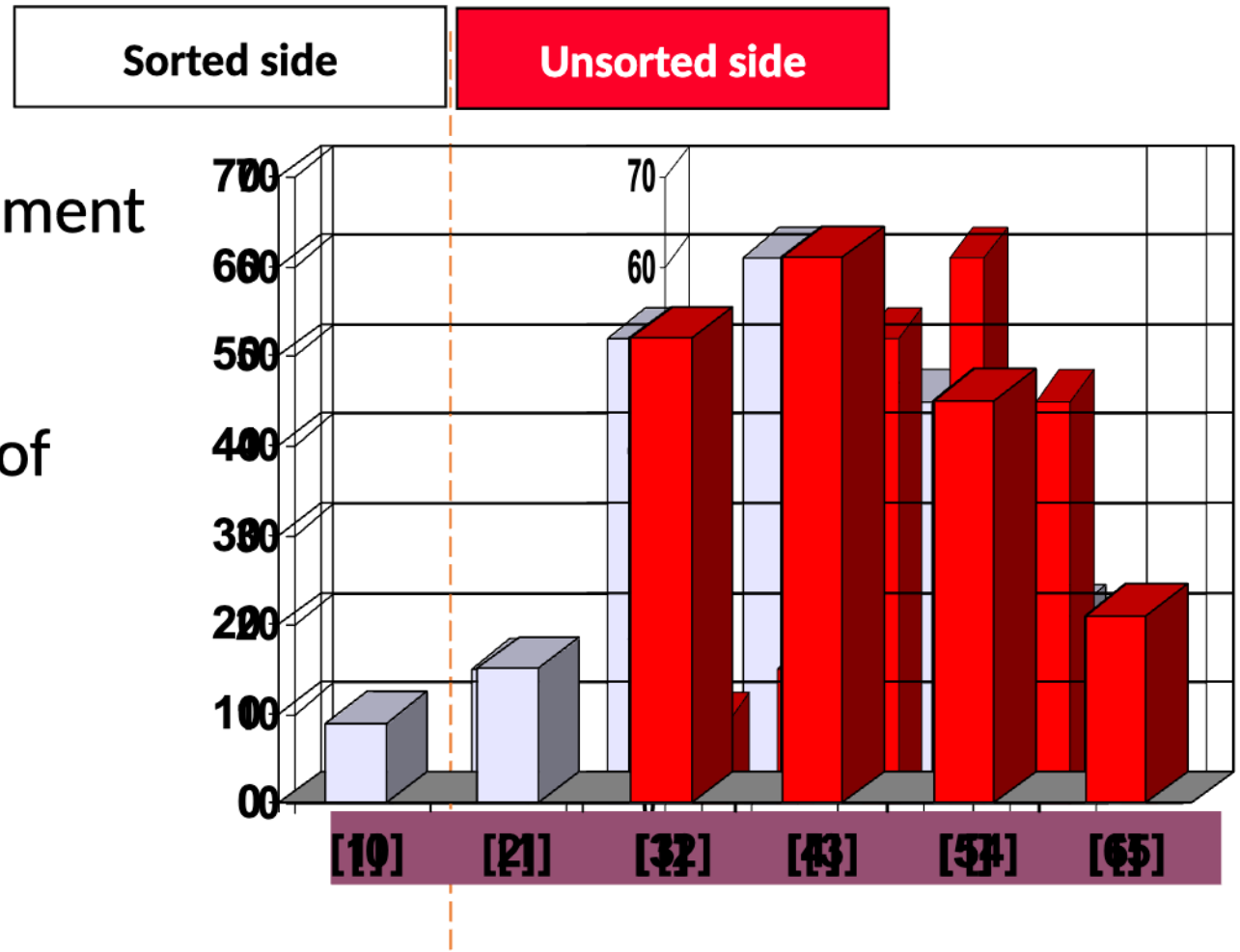
The `Selection sort` Algorithm (5)

- Each step of the Selection sort works by finding the smallest element in the unsorted side.
- At this point, we would find the number 15 at location [5] in the unsorted side.



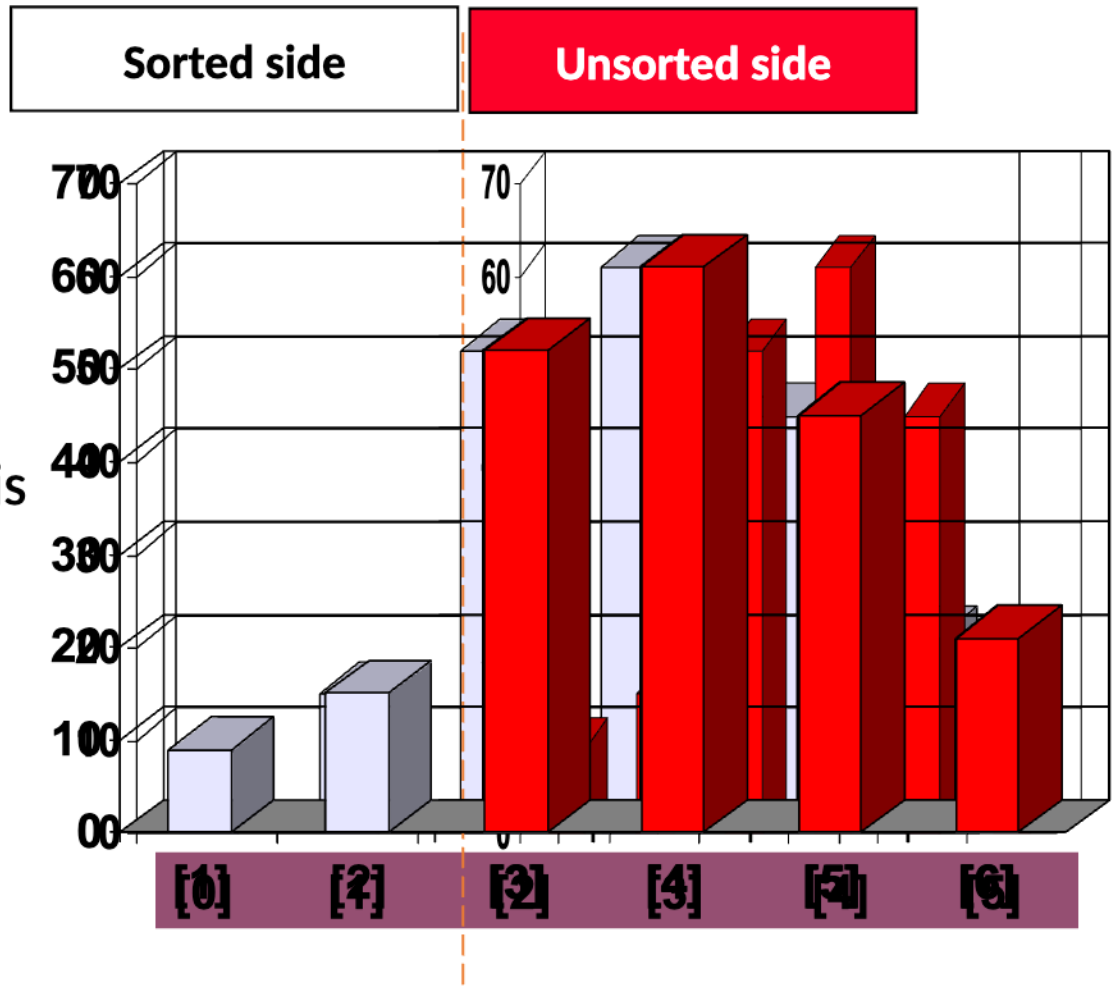
The `Selection sort` Algorithm (6)

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



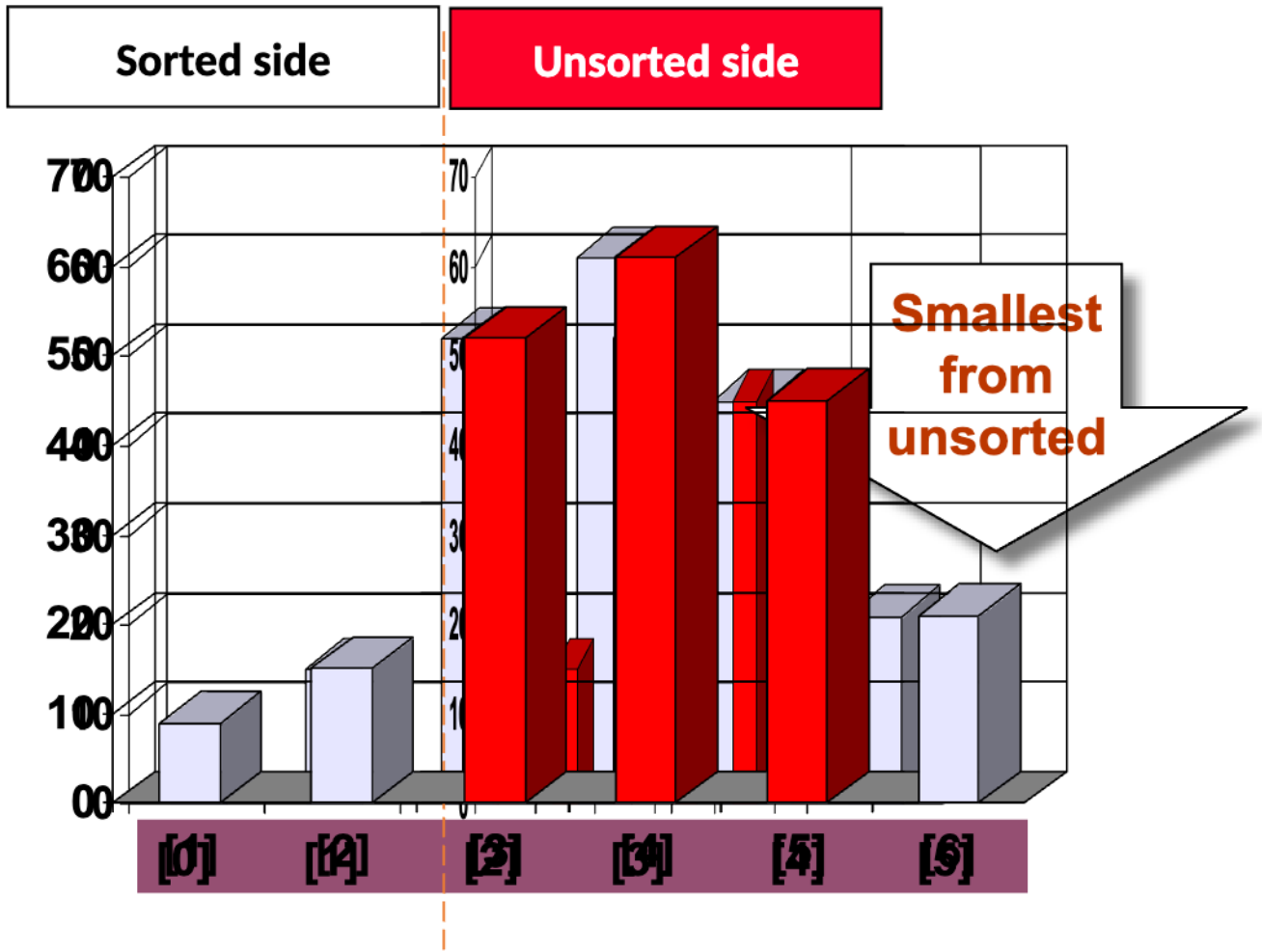
The `Selection sort` Algorithm (7)

- The size of the sorted side is increased by one element.
- The size of the unsorted side is decreased by one.



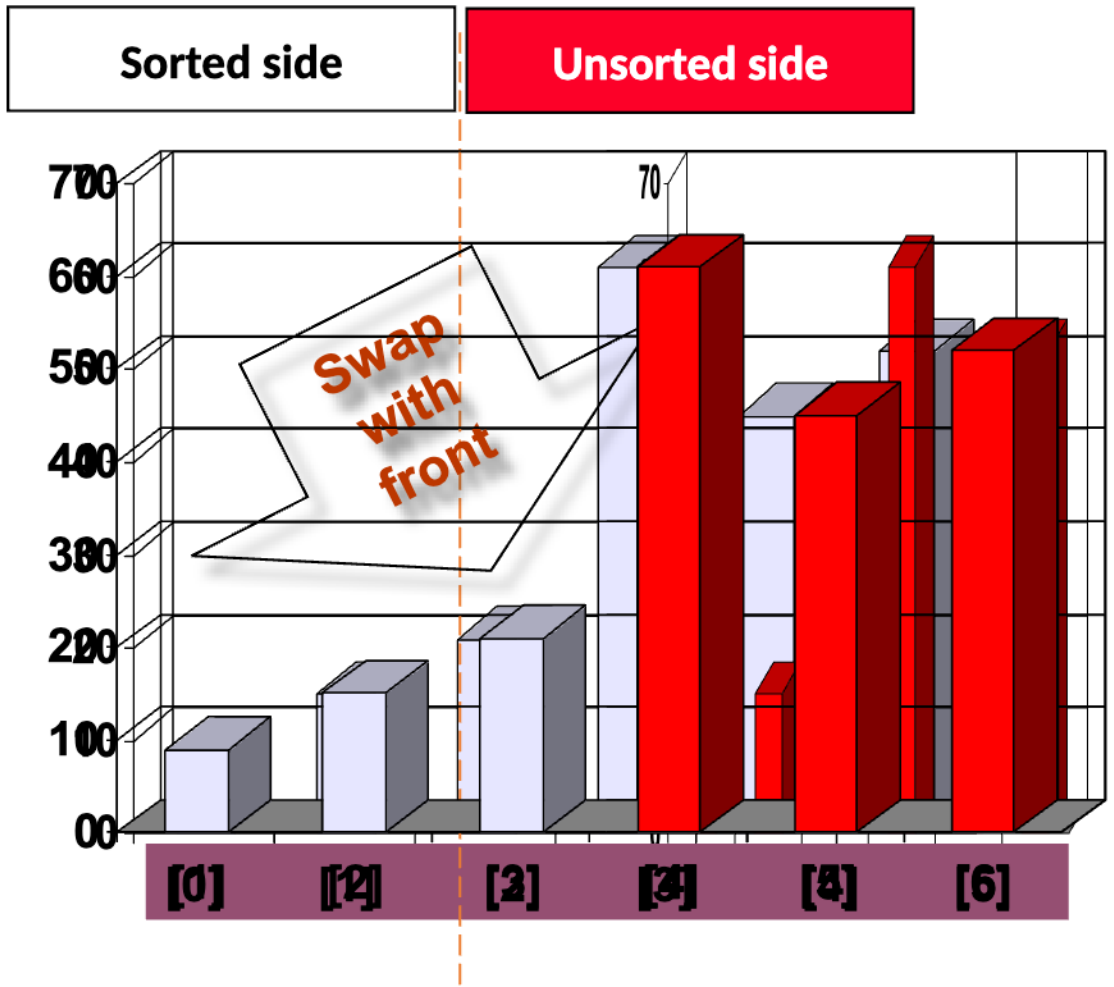
The `Selection sort` Algorithm (8)

- The process continues…

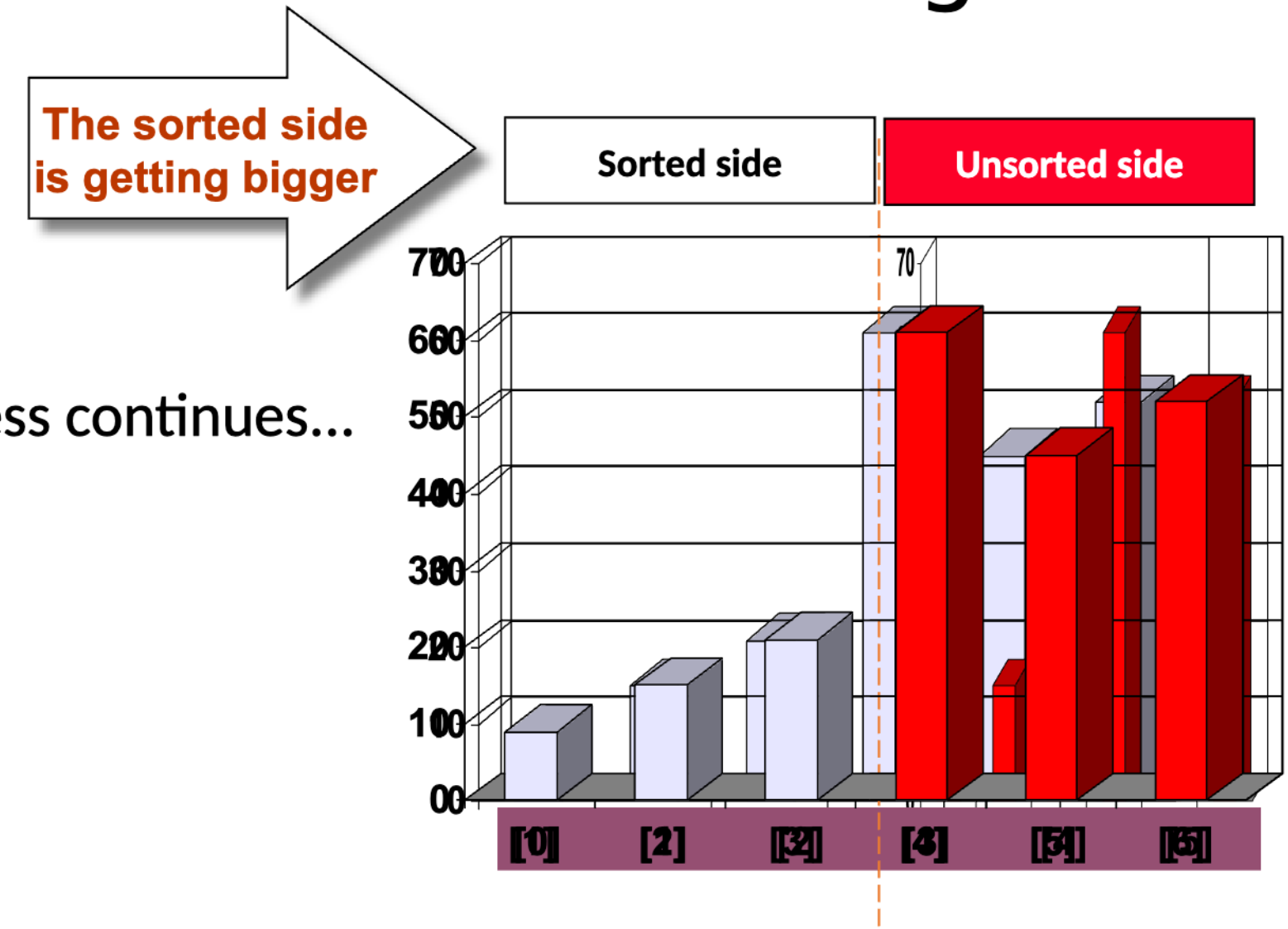


The `Selection sort` Algorithm (9)

- The process continues…



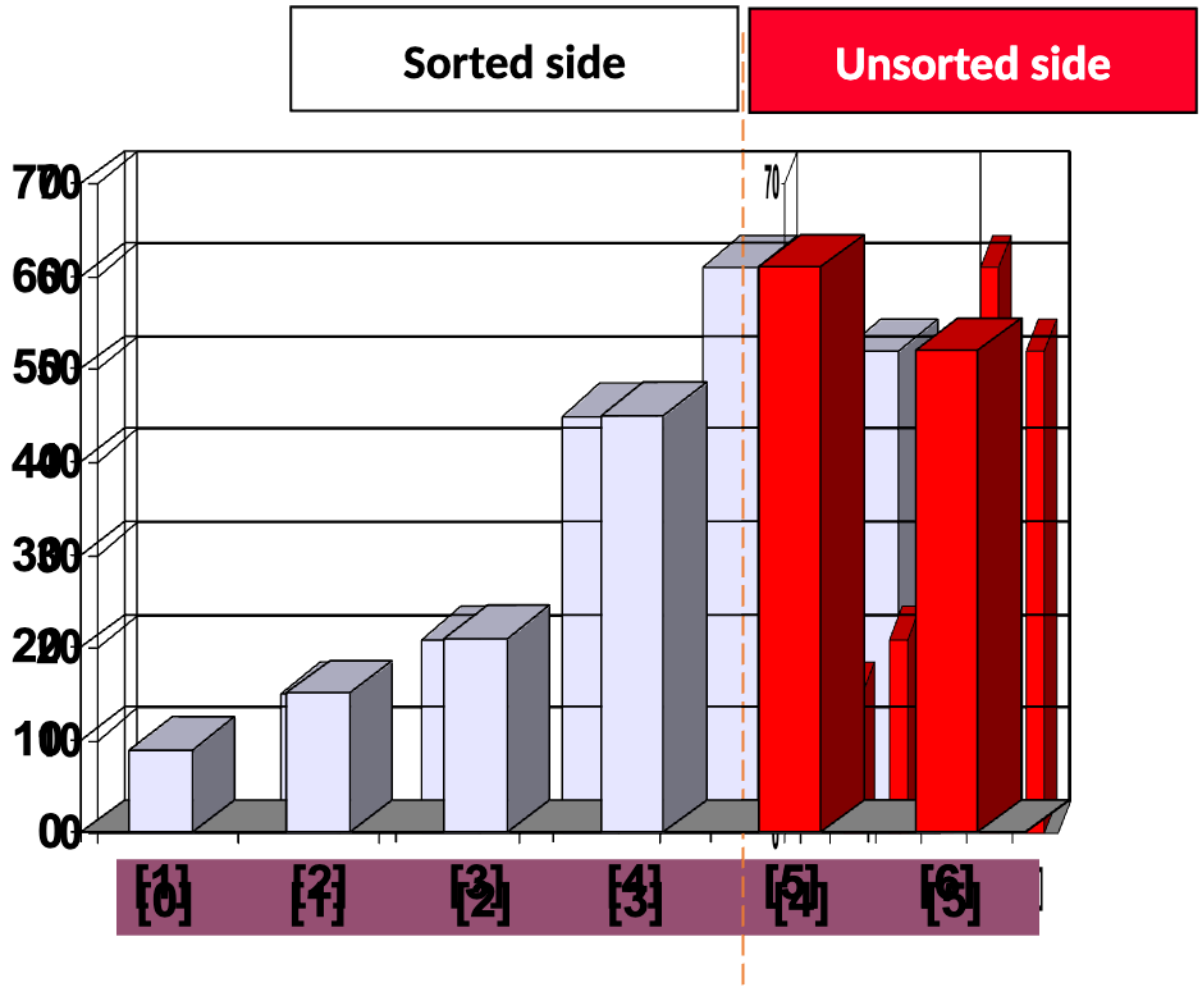
The `Selection sort` Algorithm (10)



- The process continues…

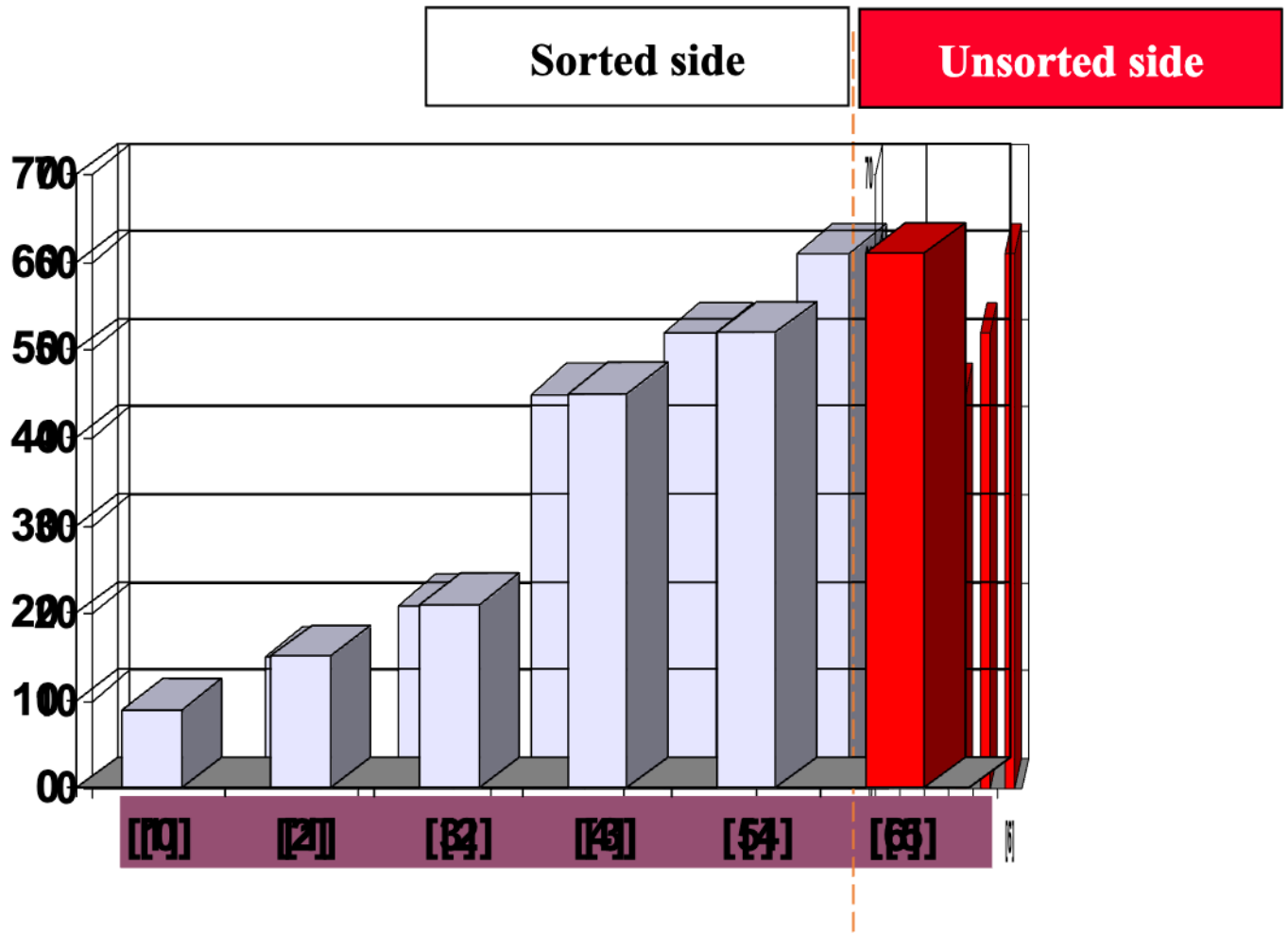
The `Selection sort` Algorithm (11)

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



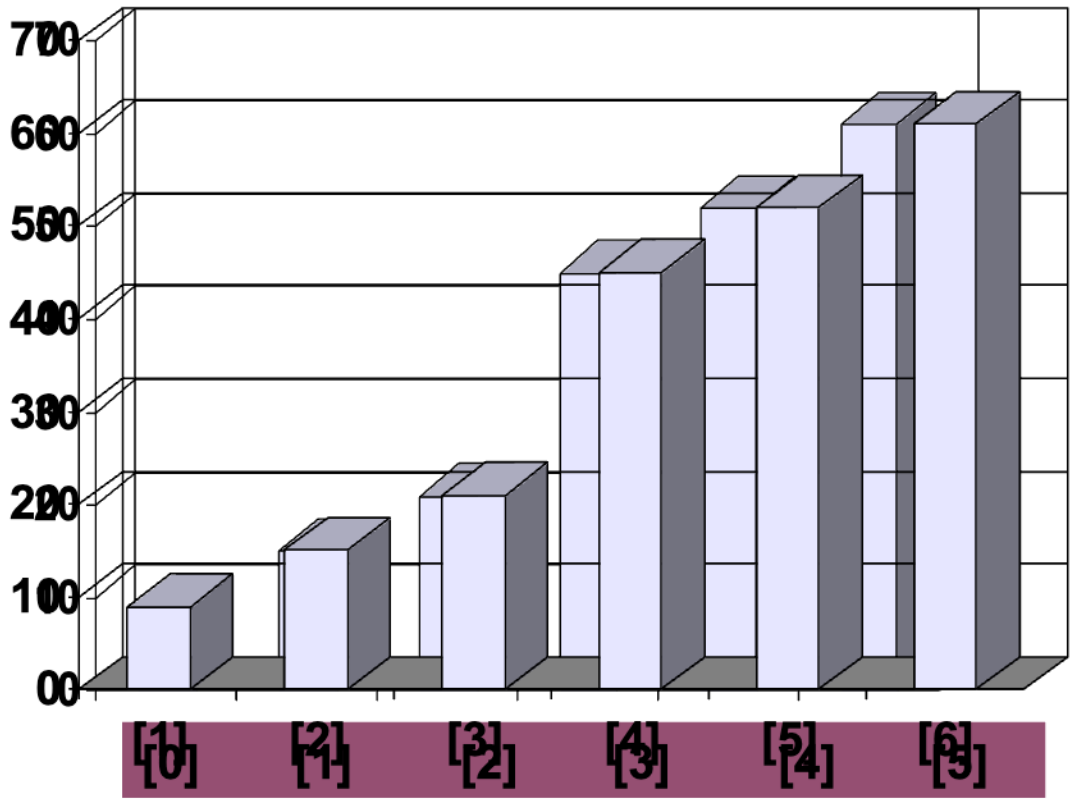
The `Selection sort` Algorithm (12)

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



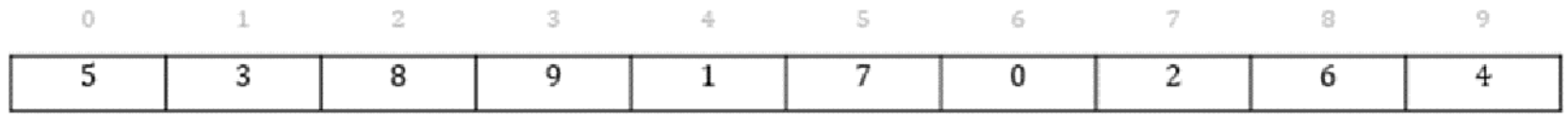
The `Selection sort` Algorithm (13)

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



Selection Sort Example

- Given the following array of integers, show the array after four iterations of selection sort (assuming the smallest element is chosen in each iteration).



- Iteration zero: 5 3 8 9 1 7 0 2 6 4
- Iteration one:
- Iteration two:
- Iteration three:
- Iteration four:

Pseudo code of selection sort Algorithm

Sorting in ascending order

```
public static void selectionSort(someDataType[] data){
        for ( i = 0; i < n-1; i++){
                small = index of the smallest element of the segment data[i] …data[n-1]
                swap data[small] and data [i]
            }
}
```

Sorting in descending order

```
public static void selectionSort(someDataType[] data){
        for ( i = 0; i < n-1 ; i++){
                big = index of the largest element of the segment data[i] …data[n-1]
                swap data[big] and data[i]
            }
}
```

How to swap array elements at positions `i` and `j`

You must use a third (temporary) variable in order to swap two array elements (or any two variables in general)

```
//swap data[i] and data[j]
temp = data[i];
data[i] = data[j];
data[j] = temp;
```

Complete `selectionSort` code

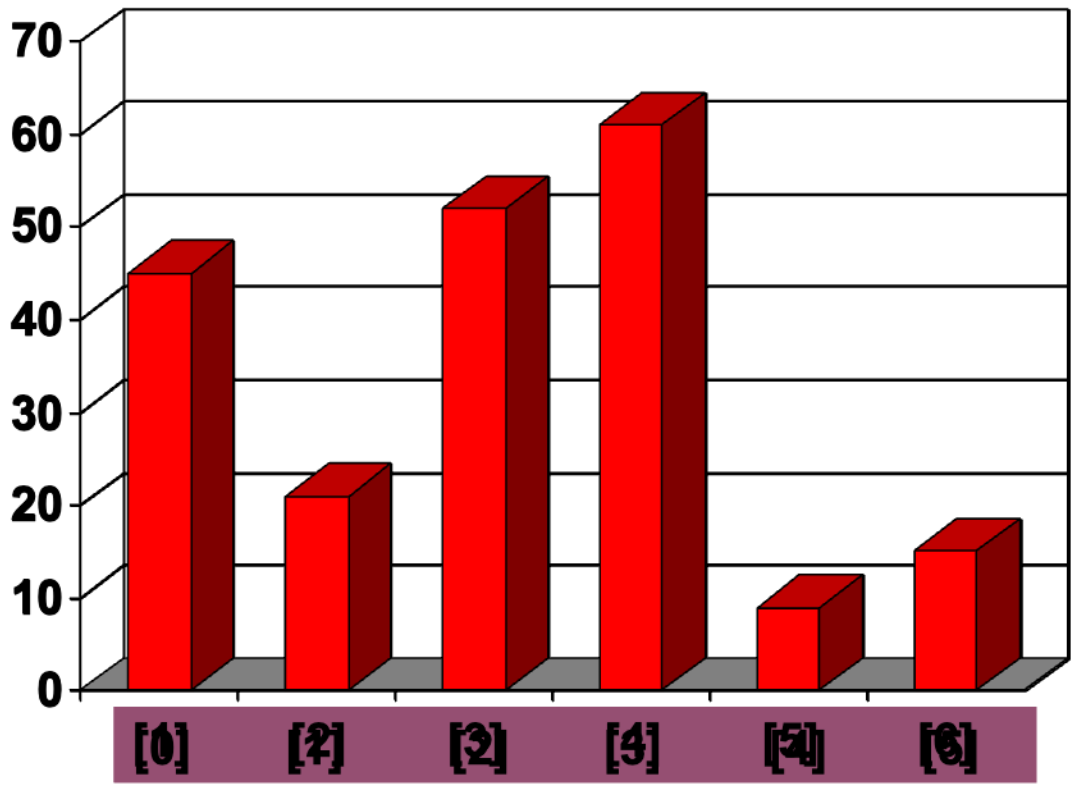
```
public static void selectionSort( someDataType[] array ){

            for ( int j=0; j<array.length-1; j++ ){
                int min = j;
                for ( int k=j+1; k<array.length; k++ ){
                     if ( array[k] < array[min] ) {
                           min = k;
                     }
                }
                someDataType temp = array[j];
                array[j] = array[min];
                array[min] = temp;
            }

}
```

Insertion Sort

“Inserting” the next element in the unsorted portion into the sorted portion.

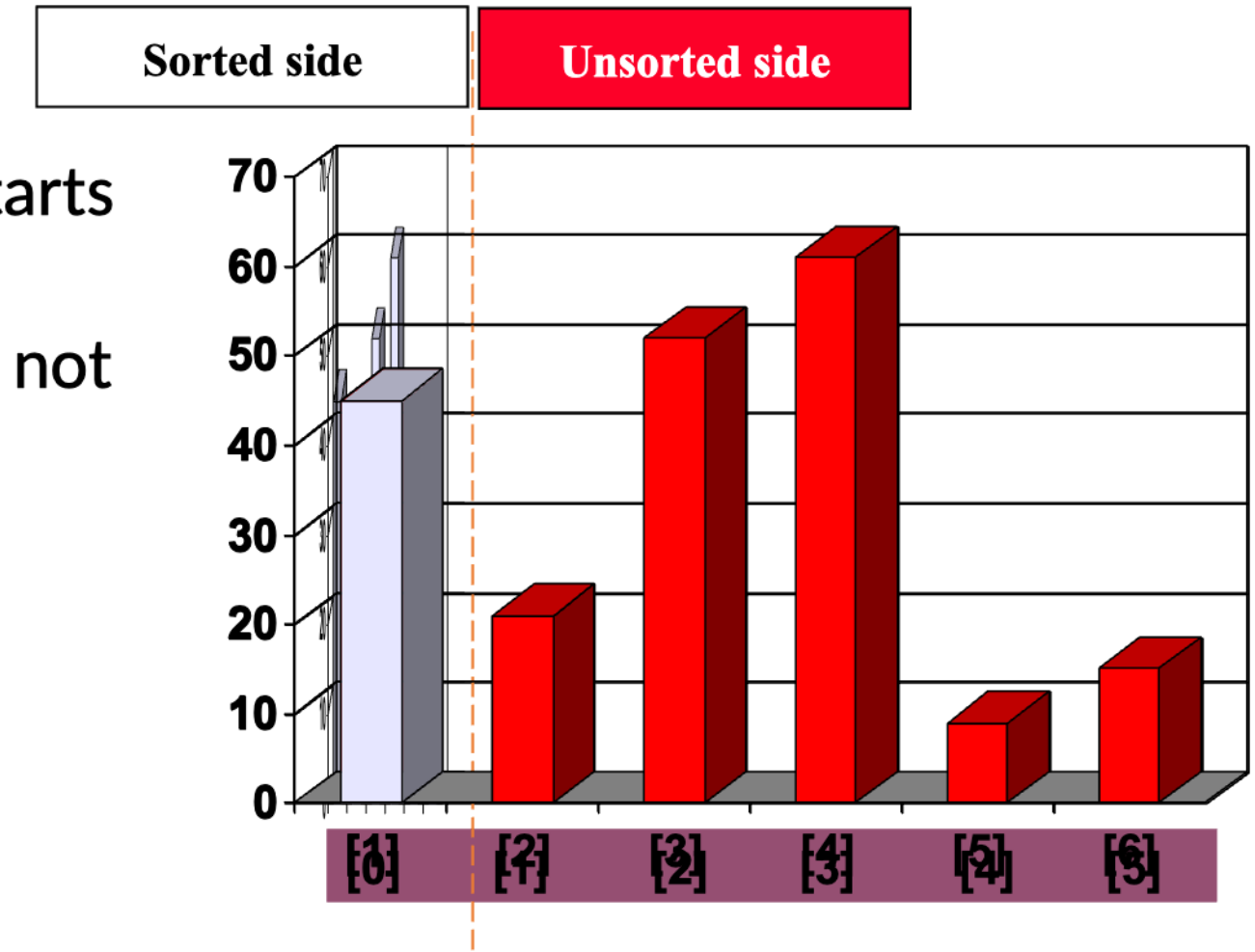
The `insertion sort` Algorithm (1)

- Same to `selection` sort, the `insertion` sort algorithm also views the array as two parts, a sorted part and an unsorted side.



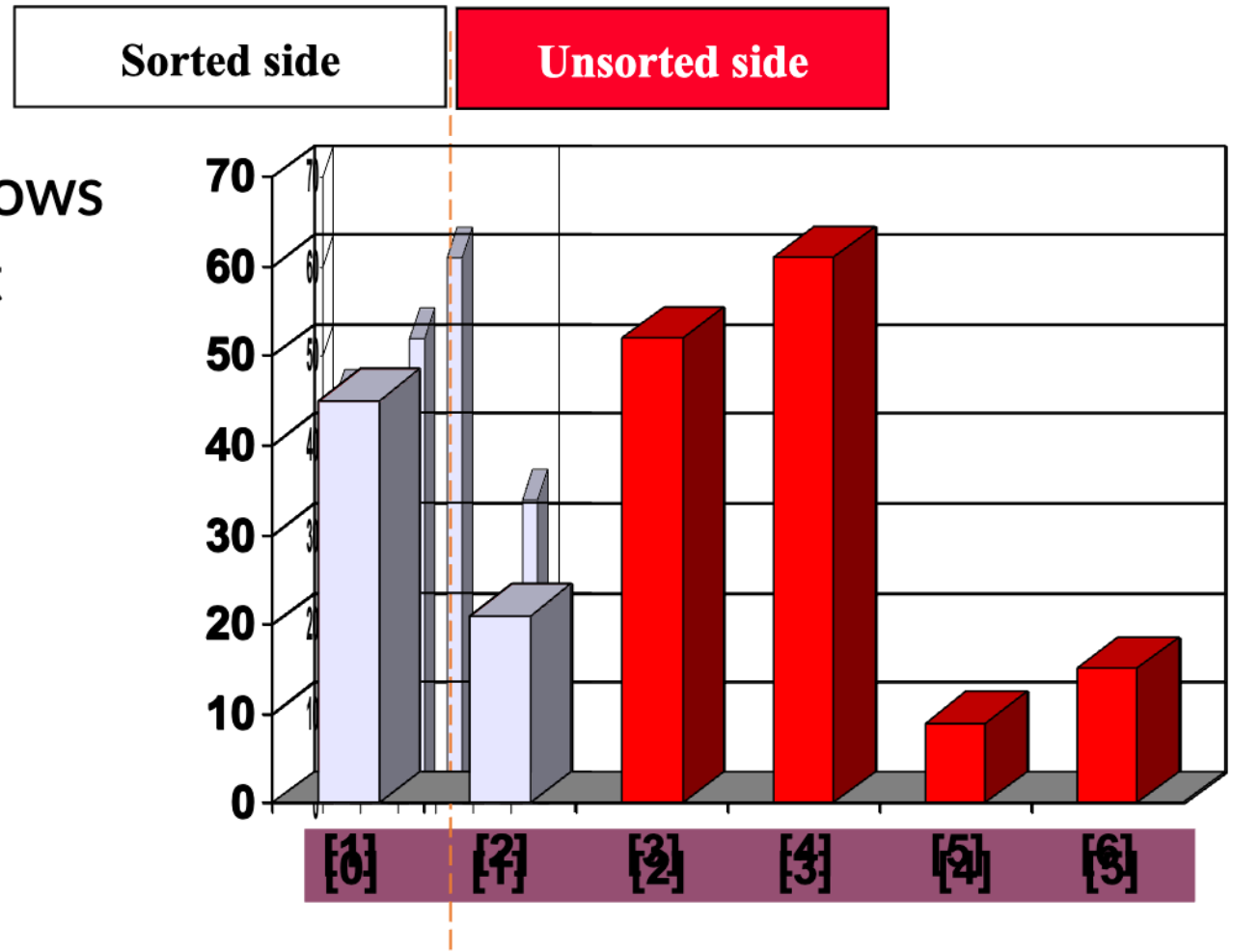
The `insertion sort` Algorithm (2)

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



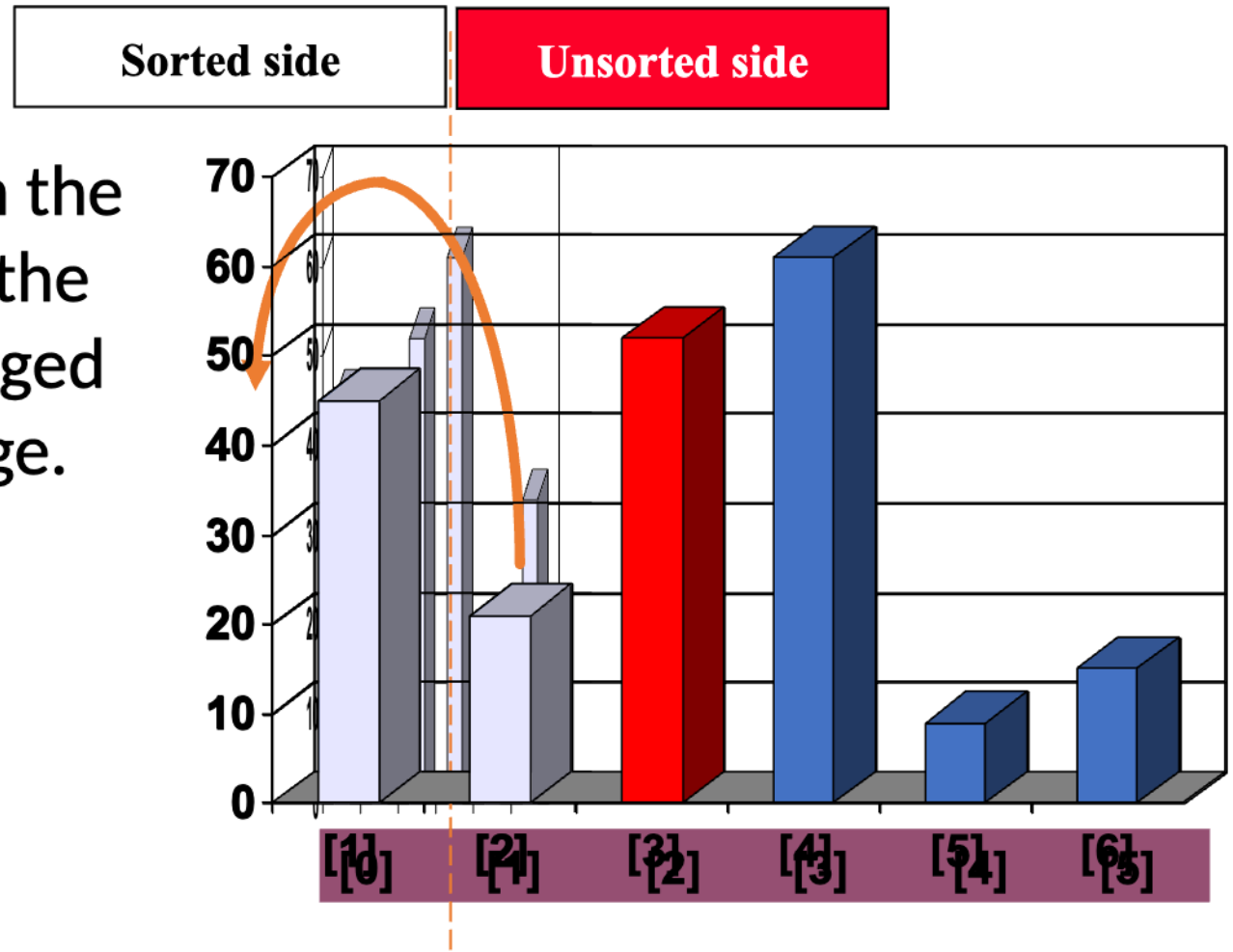
The `insertion sort` Algorithm (3)

- The sorted part grows by taking the front element from the unsorted part



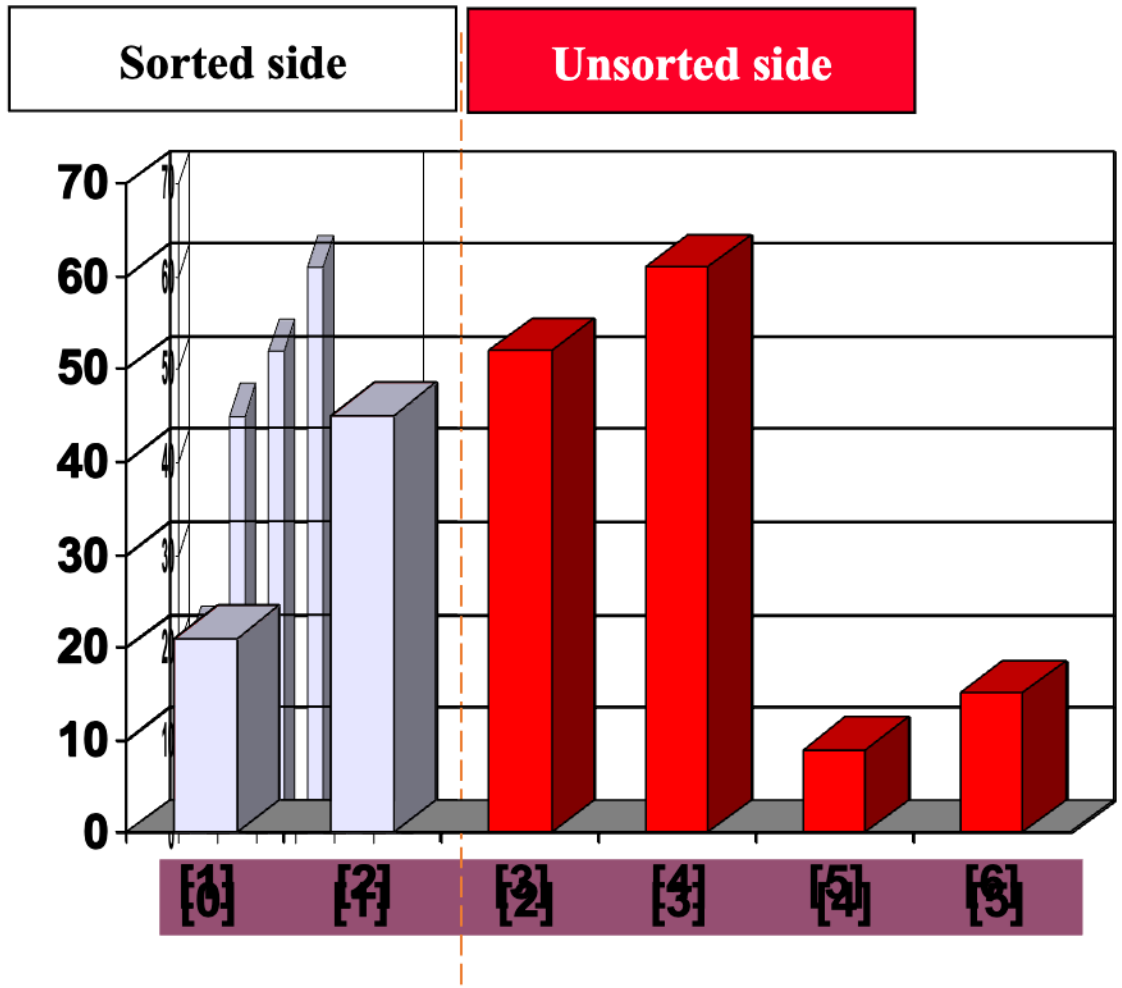
The `insertion sort` Algorithm (4)

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



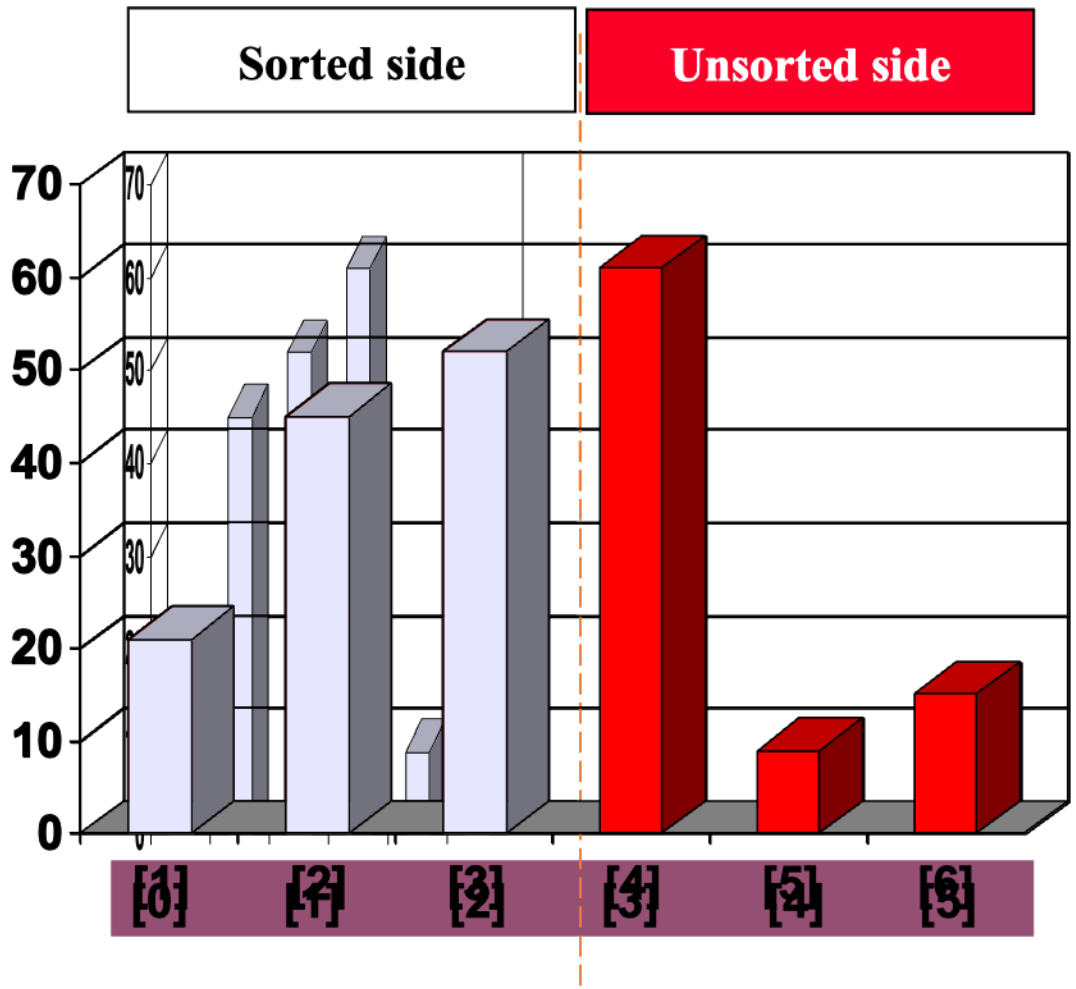
The `insertion sort` Algorithm (5)

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



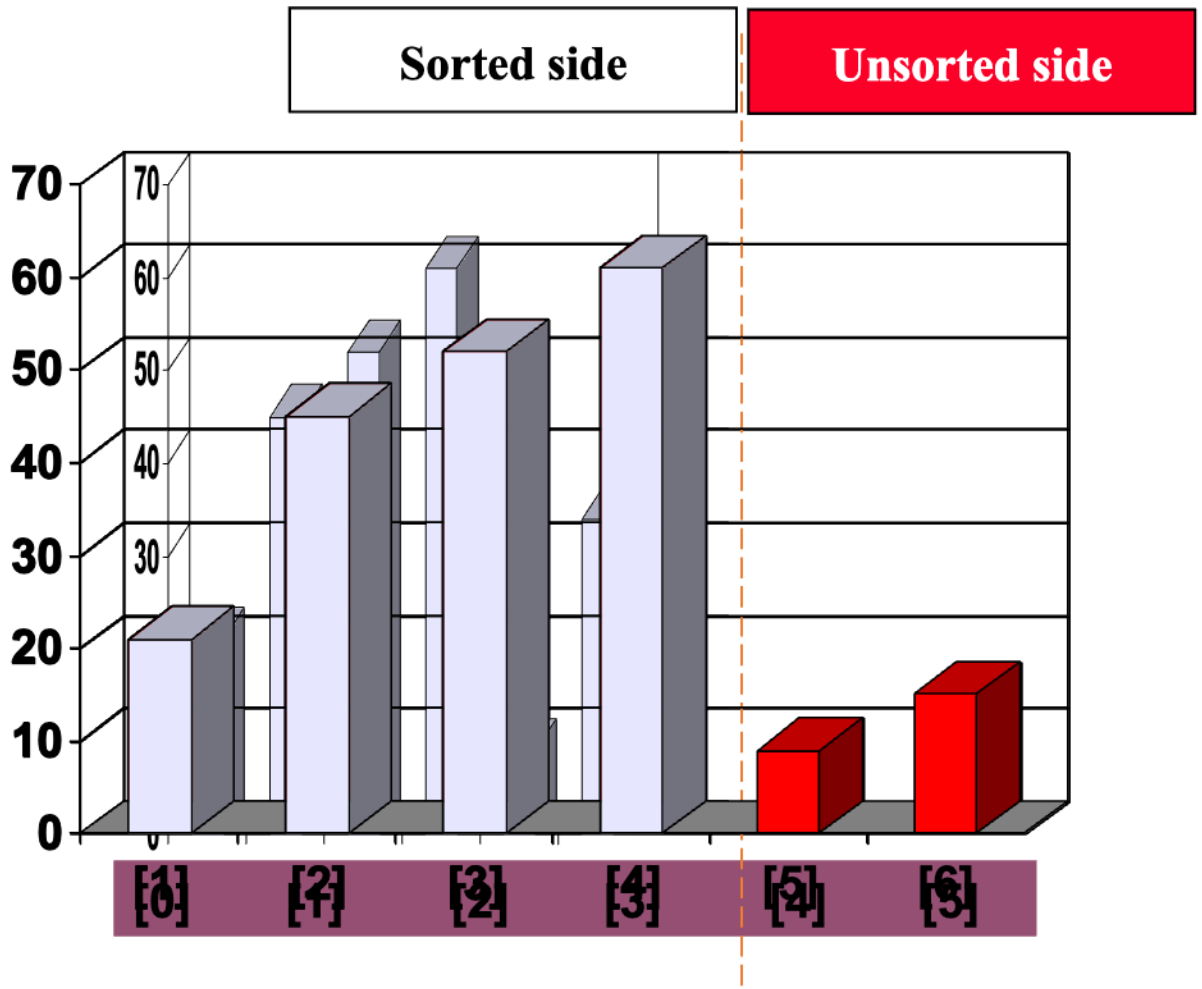
The `insertion sort` Algorithm (6)

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



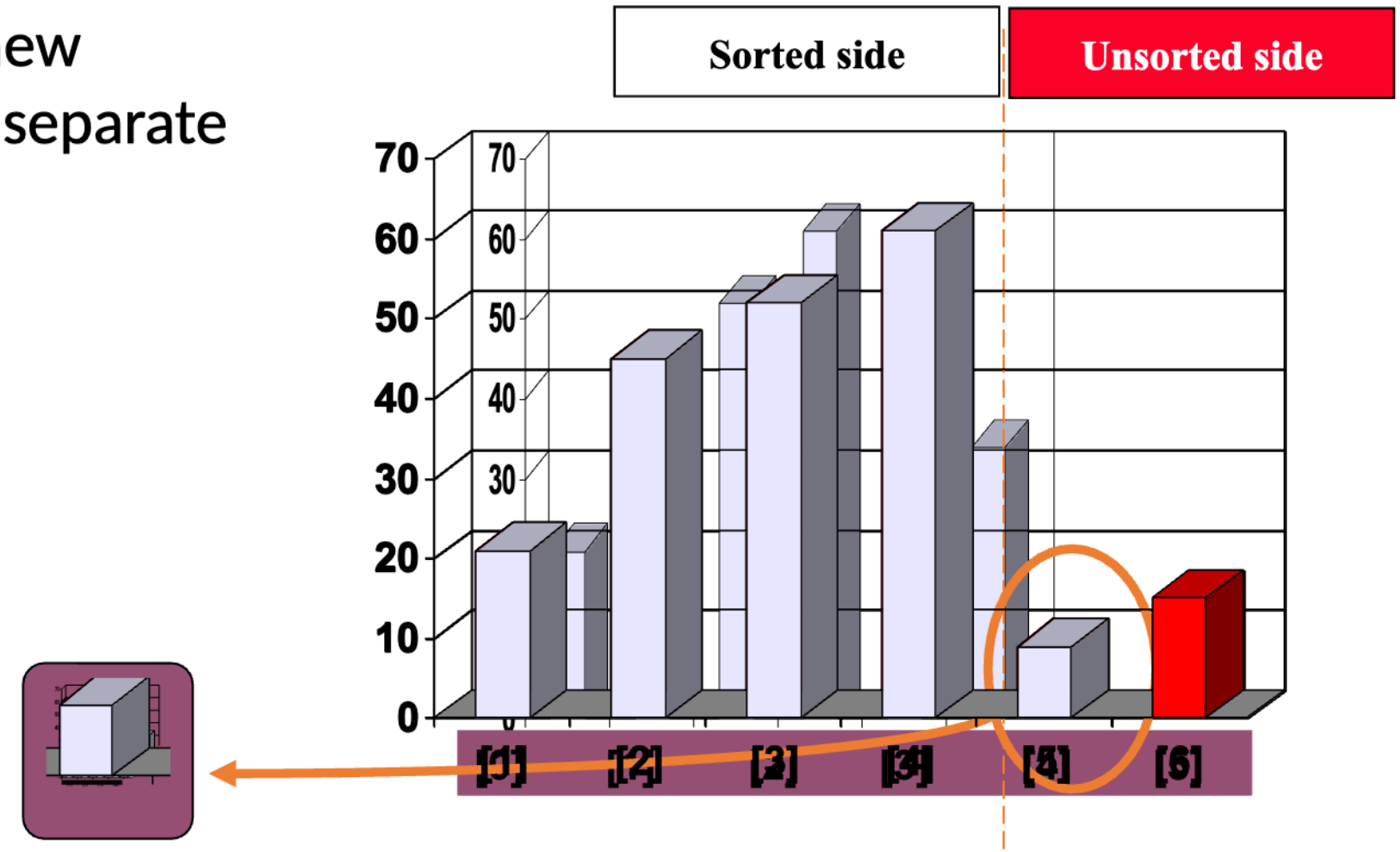
The `insertion sort` Algorithm (7)

- Sometimes we are lucky twice in a row.



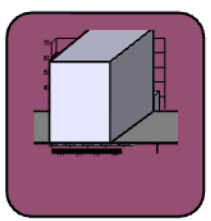
How to insert an Element to the sorted part? (1)

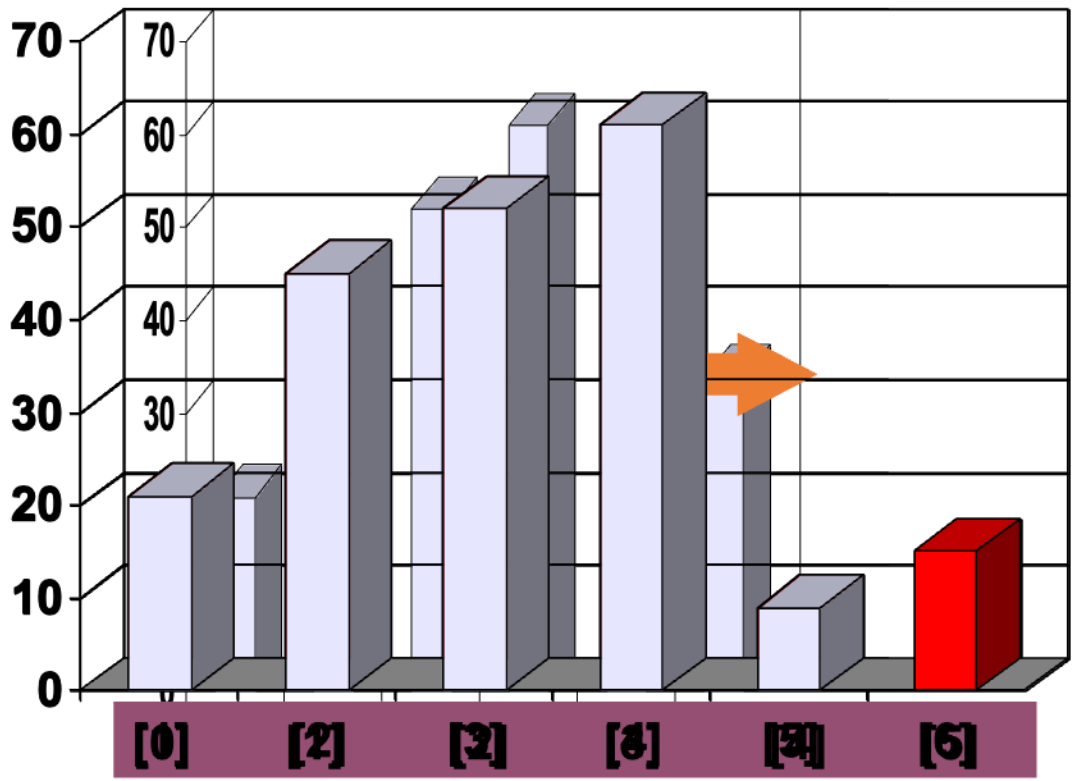
1- Copy the new element to a separate location.



How to insert an Element to the sorted part? (2)

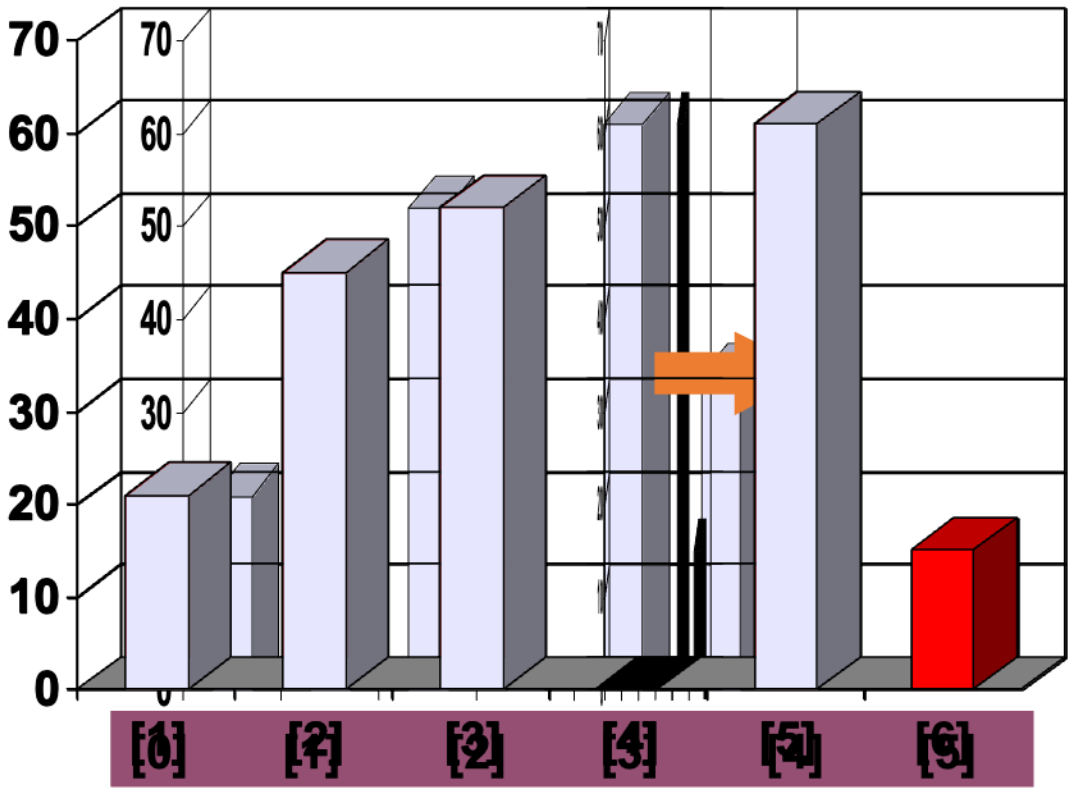
2- Shift elements in the sorted side, creating an open space for the new element.

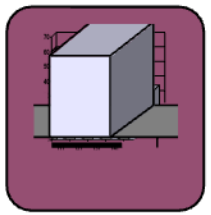




How to insert an Element to the sorted part? (3)

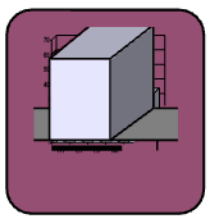
3- Shift elements in the sorted side, creating an open space for the new element.

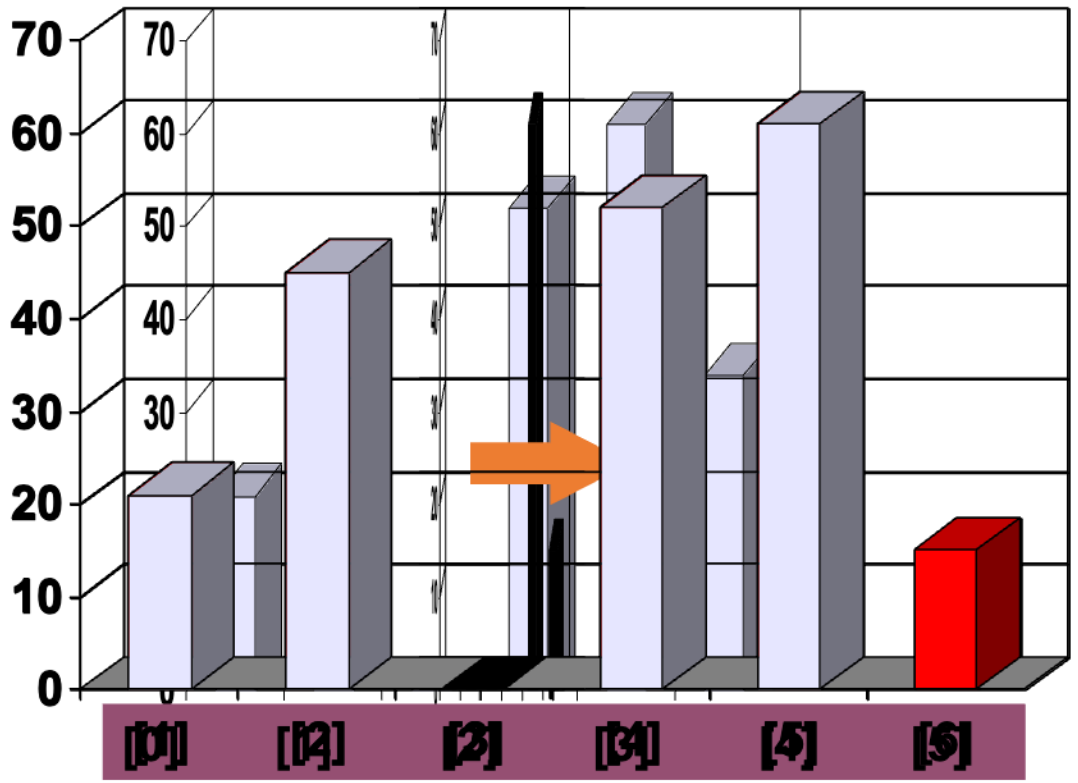




How to insert an Element to the sorted part? (4)

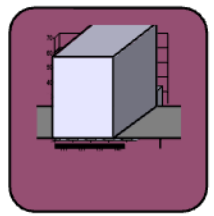
4- Continue shifting elements…

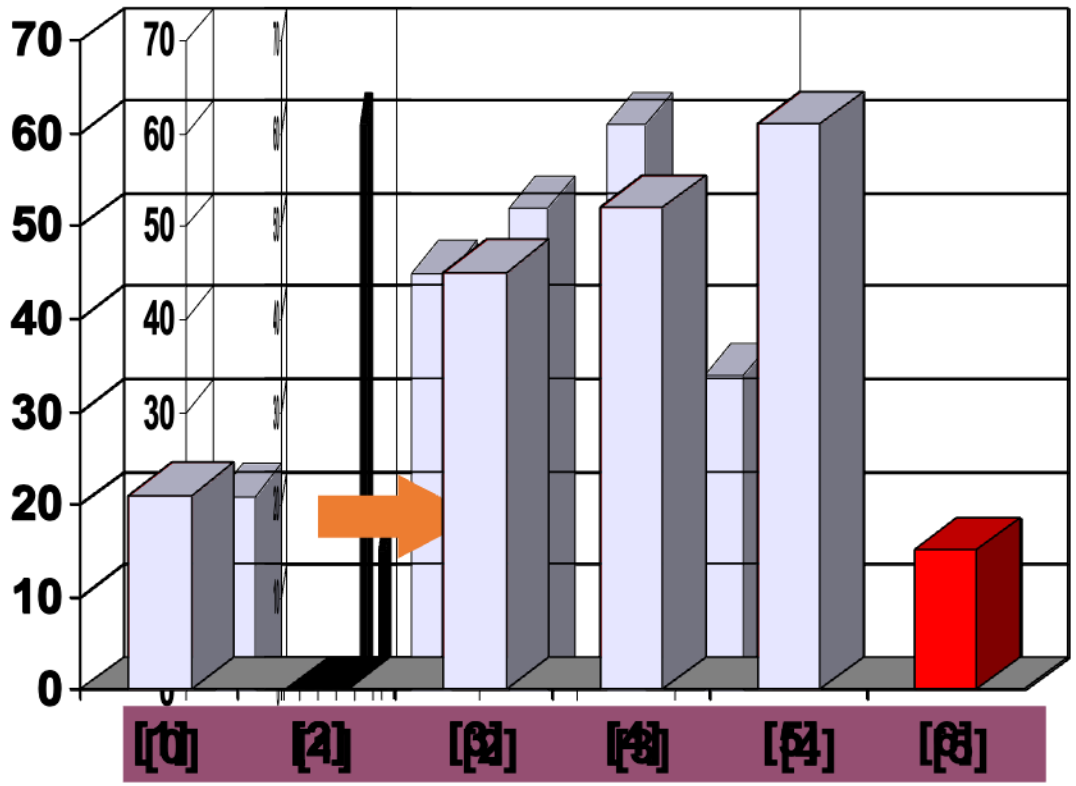




How to insert an Element to the sorted part? (5)

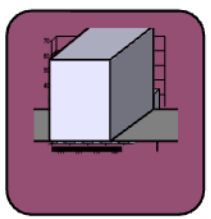
5- Continue shifting elements…

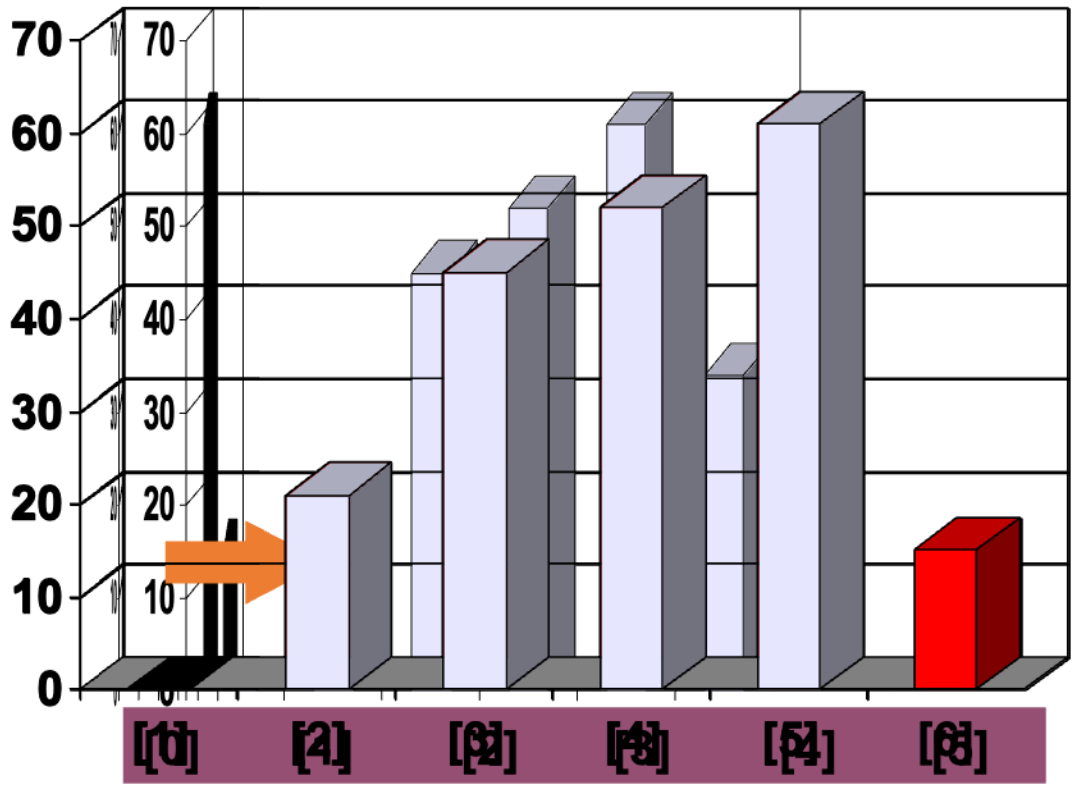




How to insert an Element to the sorted part? (6)

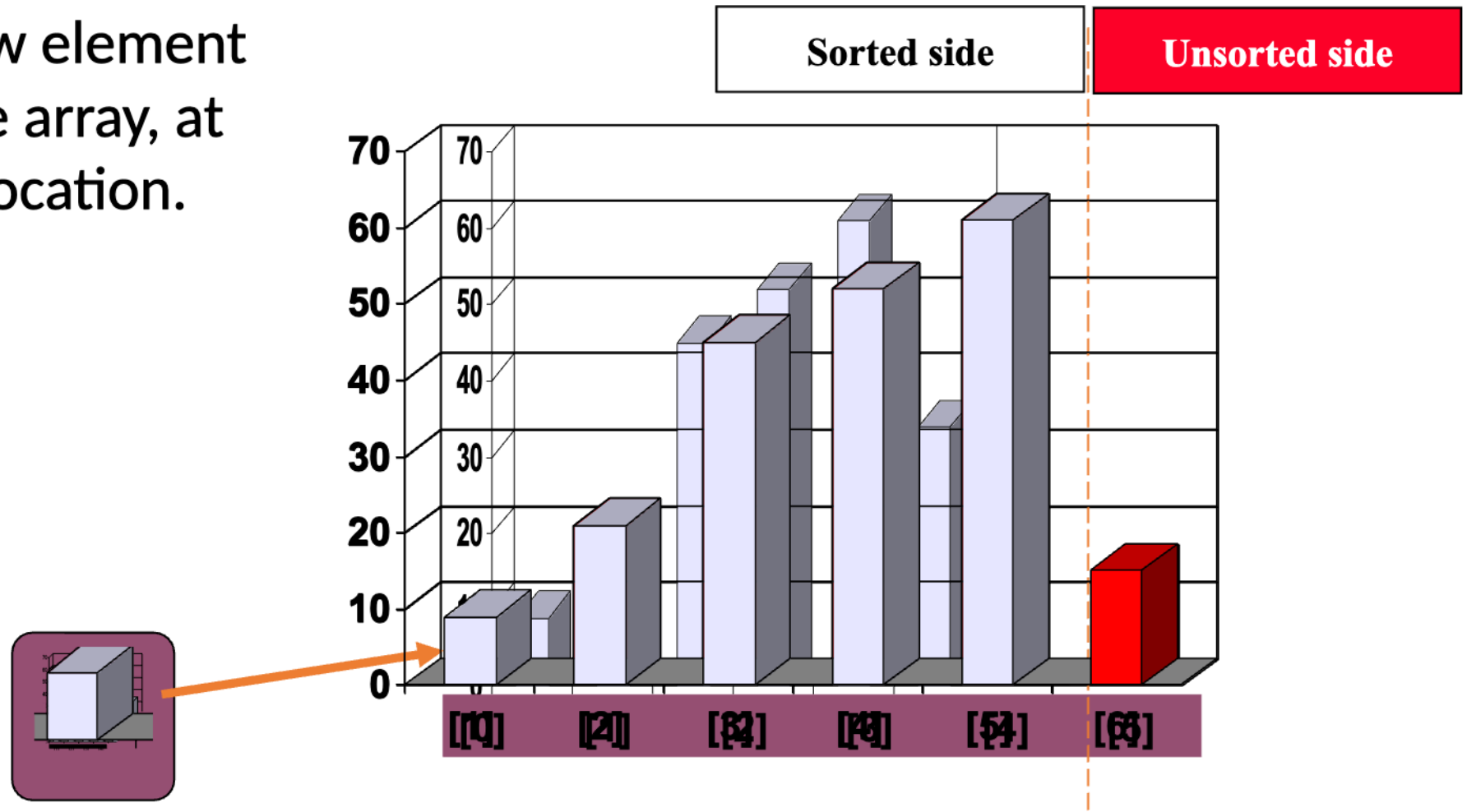
6-…until you reach the location for the new element.





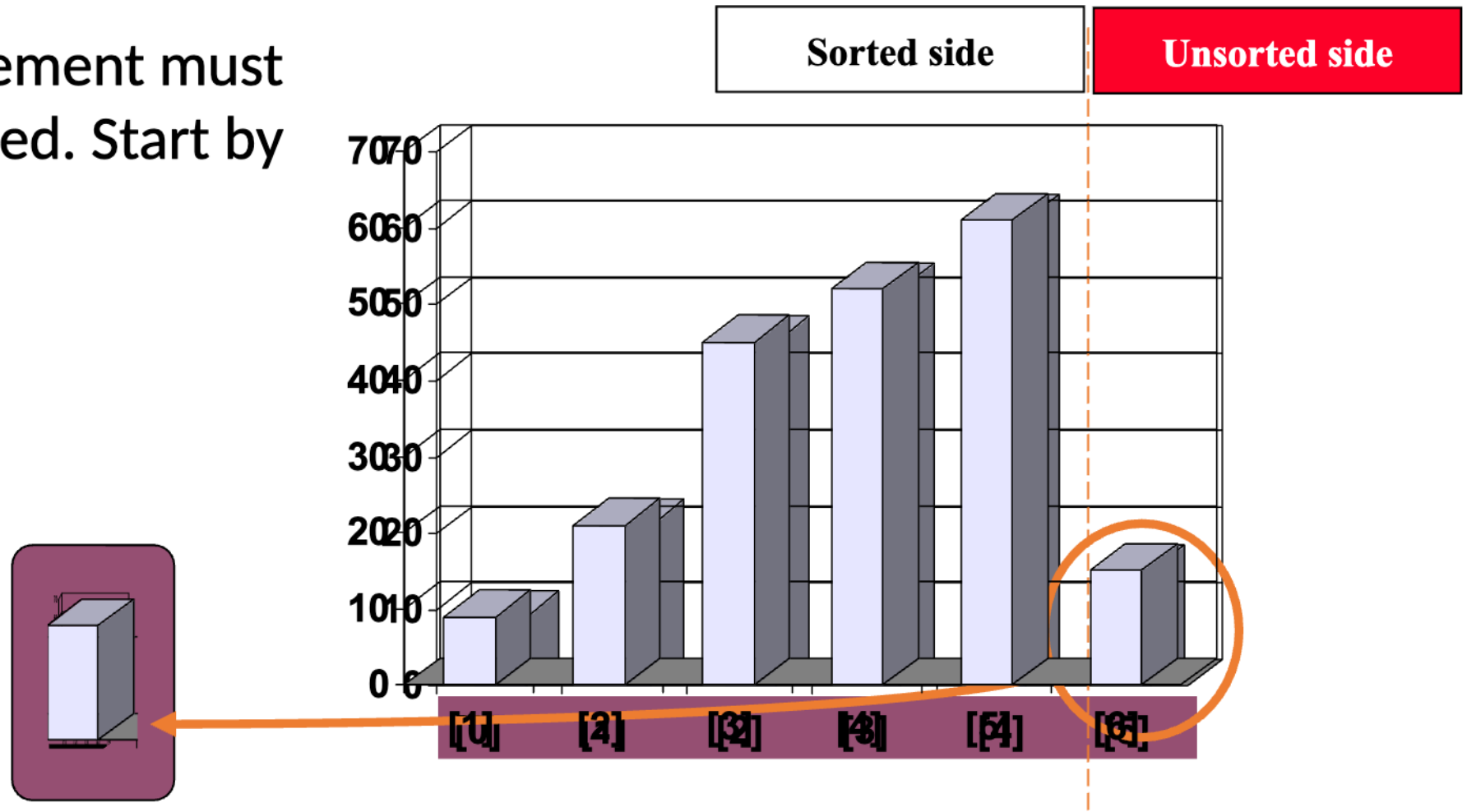
How to insert an Element to the sorted part? (7)

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



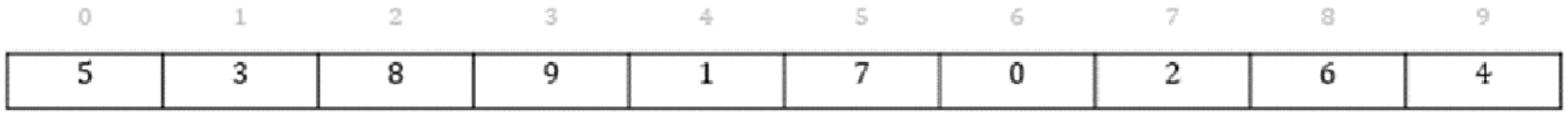
How to insert an Element to the sorted part? (8)

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



Insertion Sort Example

- Given the following array of integers, show the array after four iterations of the outer loop of insertion sort (assuming the smallest element is chosen in each iteration).



- Iteration zero: 5 3 8 9 1 7 0 2 6 4
- Iteration one:
- Iteration two:
- Iteration three:
- Iteration four:

Pseudo code of `insertion sort` Algorithm

```
public static void insertionSort(someDataType[] data){


        for ( i = 0; i < data.length; i++){
            elementToInsert = data[i];
    starting with the element at data[i-1] and continue to the
                left, if the current element is greater than elementToInsert,
                shift current element to the right;
    insert elementToInsert in the open slot;
    }
}
```

Complete `insertion sort` code

```
public static void insertionSort(someDataType[] data){
                //  For each unsorted integer
              for (int j = 1; j < data.length; j++){
                 //  Keep swapping with its left neighbor
                 //  until it is larger or equal to left neighbor
                 int k = j;
                 while (k > 0 && data[k-1] > data[k] ){
                    someDataType temp  = data[k-1];
                    data[k-1] = data[k];
                    data[k]   = temp;
                    k--;
                 }
              }
}
```

Conclusions

- On average, both selection sort and insertion sort have similar performance.
- Insertion sort performs better than selection sort if the input data is **almost** sorted.
  - Because, in each iteration, the element to be inserted will be towards the end of the sorted part and hence not too many elements will be moved.
- If an application is interested in finding only the minimum/top k elements, then selection sort can be used.
  - Because, in selection sort, the top k elements are at the beginning of the array after k iterations.