Arrays (Static)

Data Structures

- Algorithms vs. Data Structures
  - Algorithms tell the computer how to do something
  - Data Structures are ways to store information so that certain algorithms can be used on that data
  - e.g. Arrays, Linked-Lists, Stacks, Queues, Binary-Search Trees, Red-Black Trees, etc, etc.

- Data Structures always have two parts:
  - The data organization
  - Operators (the algorithms defined for that data)

Data Structures

Stacks example:

- Free to organize the data anyway you want so long as the order in which the items were put into the stack is preserved, so that the last thing added to the list can be retrieved first (LIFO – Last-in-First-Out).
- There are only two operations:
  - "Push" which means put something on the top of the stack
  - "Pop" which means get (and remove) whatever is on top of the stack

Data Structures - Arrays

- An array is also a Data Structure (the most primitive one):
- The data is organized contiguously in memory.
- The operations are “assign” and “read” (just like with variables).
- We read the value of an element in an array like this Array[i].
- We assign a value to an element in an array like this:
  - Array[i] = 3; where Array is the name of the array, i is the index of the element in the array we want to assign a value to and 3 is the value we are putting in position i of the array

Arrays

- Array
  - Consecutive group of memory locations
  - Same name and type (**`int`**, **`char`**, etc.)
- To refer to an element
  - Specify array name and position number (index)
  - Format: arrayname[position number]
  - First element is at position 0
- N-element array c

  `c[ 0 ]`, **`c[ 1 ]`** … **`c[ n - 1 ]`**
  - Nth element as position N-1

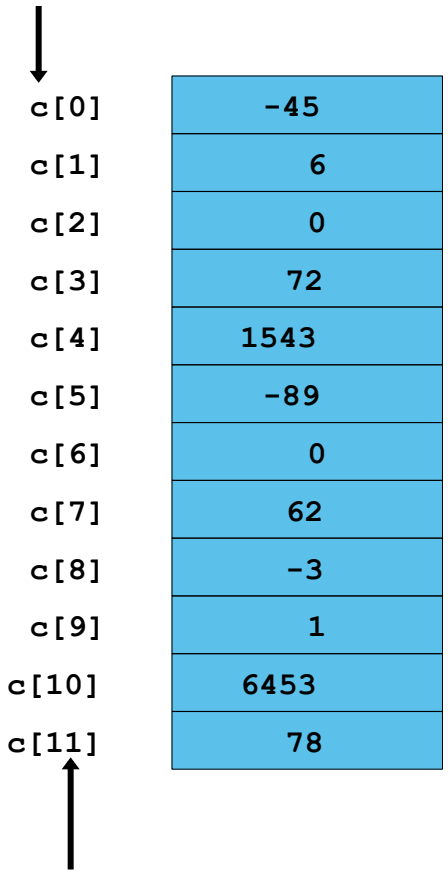
Arrays

When we declared a variable we reserved a single location in memory and gave it a label

(e.g int x)

With an array we reserve many adjacent memory locations and give them a *single* label

Name of array (Note that all elements of this array have the same name, **c**)



Position number of the element within array **c**

Arrays

- Array elements act like other variables
  - Assignment, printing for an integer array **c**

```
c[ 0 ] =  3;
cout << c[ 0 ];
```

- Can perform operations inside subscript

  c[5 - 2] same as **c[3]**

- The [] operator returns a specific memory location within the array.
- [5] counts memory locations (starting at the 0^{th} one) and returns the value in the 6^{th} memory location from the start of the array.

Declaring Arrays

- When declaring arrays, specify
  - Name
  - Type of array
    - Any data type
  - Number of elements
  - *type arrayName* **[** *arraySize* **];**

```
int c[ 10 ];  // array of 10 integers
float d[ 3284 ]; // array of 3284 floats
```

- Declaring multiple arrays of same type
  - Use comma separated list, like regular variables

```
int b[ 100 ], x[ 27 ];
```

Initializing Arrays

- Initializing arrays
  - For loop
    - Set each element individually
  - Initializer list
    - Specify each element when array declared

```
int n[ 5 ] = { 1, 2, 3, 4, 5 };
```

    - If not enough initializers, rightmost elements 0
    - If too many syntax error
  - To set every element to same value

```
int n[ 5 ] = { 0 };
```

  - If array size omitted, initializers determine size

```
int n[] = { 1, 2, 3, 4, 5 };
```

    - 5 initializers, therefore 5 element array

```
// Initializing an array.
#include <iostream>

using namespace std;

int main()
{
   int n[ 10 ];  // n is an array of 10 in

   // initialize elements of array n to 0
   for ( int i = 0; i < 10; i++ )
      n[ i ] = 0;   // set element at location i to 0

   cout << "Element\tValue" << endl;

   // output contents of array n in tabular format
   for ( int j = 0; j < 10; j++ )
      cout << j << "\t" << n[ j ] << endl;

return 0;
}
```

Declare a 10-element array of integers.

Initialize array to **0** using a for loop. Note that the array has elements **n[0]** to **n[9]**.

The program prints:

```
Element         Value
      0             0
      1             0
      2             0
      3             0
      4             0
      5             0
      6             0
      7             0
      8             0
      9             0
```

```
// Initializing an array with a declaration.
#include <iostream>

using namespace std;

int main()
{
   // use initializer list to initialize array n
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

   cout << "Element\tValue" << endl;

   // output contents of array n in tabular format
   for ( int i = 0; i < 10; i++ )
      cout << i << “\t” << n[ i ] << endl;

   return 0;  // indicates successful execution

} // end main
```

Note the use of the initializer list.

Element	Value
0	32
1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37

Tip: use constants to keep track of array sizes

- Array size
  - Since the size of an array cannot change once it has been declared use a constant to keep track of that size (why?)
  - Can be specified with constant variable (**`const`**)
    - **`const int size = 20;`**
  - Constants cannot be changed
  - Constants must be initialized when declared
  - Also called named constants or read-only variables

```
// Using a properly initialized constant variable.
#include <iostream>
int main()
{
   const int x = 7;  // initialized constant variable

   cout << "The value of constant variable x is: "
        << x << endl;

   return 0;  // indicates successful termination

} // end main
```

Proper initialization of **const** variable.

```
The value of constant variable x is: 7
```

```
// Mistakes using const
int main()
{
   const int x;  // Error: x must be initialized

   x = 7;        // Error: cannot modify a const variable

   return 0;     // indicates successful termination

} // end main
```

Uninitialized **const** results in a syntax error. Attempting to modify the **const** is another error.
Can only assign a value when the const variable is declared.

```
d:\cpphtp4_examples\ch04\Fig04_07.cpp(6) : error C2734: 'x' :
   const object must be initialized if not extern
d:\cpphtp4_examples\ch04\Fig04_07.cpp(8) : error C2166:
   l-value specifies const object
```

```
// Initialize array s to the even integers from 2 to 20.
#include <iostream>

using namespace std;

int main()
{
   // constant variable can be used to specify array size
   const int arraySize = 10;

   int s[ arraySize ];  // array s has 10 elements

   for ( int i = 0; i < arraySize; i++ )  //
      s[ i ] = 2 + 2 * i;

   cout << "Element\tValue" << endl;

   // output contents of array s in tabular f
   for ( int j = 0; j < arraySize; j++ )
   {
      cout << j << "\t" << s[ j ] << endl;
   }

   return 0;  // indicates successful termina

} // end main
```

Note use of **`const`** keyword. Only **`const`** variables can specify array sizes.

The program becomes more scalable when we set the array size using a **`const`** variable. We can change **`arraySize`**, and all the loops will still work (otherwise, we'd have to update every loop in the program).

The previous program prints:

```
Element         Value
      0             2
      1             4
      2             6
      3             8
      4            10
      5            12
      6            14
      7            16
      8            18
      9            20
```

```
// Compute the sum of the elements of the array.
#include <iostream>
using namespace std;

int main()
{
  const int arraySize = 10;
   int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   int total = 0;

   // sum contents of array a
   for ( int i = 0; i < arraySize; i++ )
   {
      total += a[ i ];
   }

   cout << "Total of array element values is " << total << endl;

   return 0;  // indicates successful termination
} // end main
```

```
Total of array element values is 55
```

```
// Histogram printing program.
#include <iostream>
using namespace std;

int main()
  {
     const int arraySize = 10;
     int n[ arraySize ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };

     cout << "Element\tValue\tHistogram" << endl;

     // for each element of array n, output a bar in the histogram
     for ( int i = 0; i < arraySize; i++ )
     {
        cout << i << "\t" << n[ i ] << "\t";

        for ( int j = 0; j < n[ i ]; j++ )   // print one bar
        {
           cout << '*';
        }
       cout << endl;
      }
```

Prints the number of asterisks corresponding to the value in array element, **n[i]**.

The previous program prints:

```
Element         Value         Histogram
      0            19         ********************
      1             3         ***
      2            15         ***************
      3             7         *******
      4            11         ***********
      5             9         *********
      6            13         *************
      7             5         *****
      8            17         *****************
      9             1         *
```

```
 // Roll a six-sided die 6000 times.
 #include <iostream> // For I/O
#include <cstdlib> // For rand function
#include <ctime> // For time function

int main()
{
   const int arraySize = 7;
   int frequency[ arraySize ] = { 0 };
   srand( time( 0 ) );  // seed random-numbe

   for ( int roll = 1; roll <= 6000; roll++
   {
      ++frequency[ 1 + rand() % 6 ]; // repl
   }

   // output frequency elements 1-6 in tabul
   cout << "Face\tFrequency" << endl;
   for ( int face = 1; face < arraySize; face++ )
   {
      cout << face << "\t" << frequency[ face ] << endl;
   }
   return 0;  // indicates successful program execution
} // end main
```

Remake of old program to roll dice from your book. An array is used instead of 6 regular variables, and the proper element can be updated easily (without needing a selection structure).

This creates a number between 1 and 6, which determines the index of **`frequency[]`** that should be incremented.

Output from previous program:

```
Face    Frequency
   1         1003
   2         1004
   3          999
   4          980
   5         1013
   6         1001
```

```
// Student poll program.
#include <iostream>

using namespace std;

int main()
{
   // define array sizes
   const int responseSize = 40;   // size of array responses
   const int frequencySize = 11;  // size of array frequency

   // place survey responses in array responses
   int responses[ responseSize ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
       10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
       5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };

   // initialize frequency counters to 0
   int frequency[ frequencySize ] = { 0 };
```

```
   // for each answer, select value of an element of array
   // responses and use that value as subscript in array
   // frequency to determine element to increment
   for ( int answer = 0; answer < responseSize; answer++ )
   {
      ++(frequency[ responses[answer] ]);
   }

   // display results
   cout << "Rating\tFrequency" << endl;

   // output frequencies in tabular format
   for ( int rating = 1; rating < frequencySize; rating++ )
   {
      cout << rating << frequency[ rating
   }

   return 0;  // indicates successful term
} // end main
```

`responses[answer]` is the rating (from 1 to 10). This determines the index in **`frequency[]`** to increment.

Rating	Frequency
1	2
2	2
3	2
4	2
5	5
6	11
7	5
8	7
9	1
10	3

Character Arrays (strings)

- Strings
  - Arrays of characters
  - All strings end with the **`null`** character, **`'\0'`**
  - Examples
    - **`char string1[] = "hello";`**
      - **`Null`** character implicitly added
      - **`-string1`** has 6 elements
  - **`char string1[]={'h','e','l','l','o','\0'};`**
  - Subscripting is the same

    `String1[ 0 ]` is **`'h'`**

    `string1[ 2 ]` is **`'l'`**

Examples Using Arrays

- Input from keyboard

```
char string2[ 10 ];
cin >> string2;
```

  - Puts user input in string
    - Stops at first whitespace character
    - Adds **`null`** character
  - If too much text entered, data written beyond array
    - We want to avoid this (section 5.12 explains how)
- Printing strings
  - **`cout << string2 << endl;`**
    - Does not work for other array types
  - Characters printed until **`null`** found

```cpp
// Treating character arrays as strings.
#include <iostream>

using namespace std;

int main()
{
   char string1[ 20 ],                  // reserves 20 characters
   char string2[] = "string literal"; // reserves 15 characters

   // read string from user into array string2
   cout << "Enter the string \"hello there\": ";
   cin >> string1;  // reads "hello" [space terminates input]

   // output strings
   cout << "string1 is: " << string1 << "\nstring2 is: " << string2;

   cout << "\nstring1 with spaces between characters is:\n";

   // output characters until null character is reached
   for ( int i = 0; string1[ i ] != '\0'; i++ )
   {
      cout << string1[ i ] << ' ';
   }
   cin >> string1;  // reads "there"
   cout << "\nstring1 is: " << string1 << endl;

   return 0;  // indicates successful termination

} // end main
```

Two different ways to declare strings. **`string2`** is initialized, and its size determined automatically .

Examples of reading strings from the keyboard and printing them out.

```
Enter the string "hello there": hello there
string1 is: hello
string2 is: string literal
string1 with spaces between characters is:
h e l l o
string1 is: there
```

Passing Arrays to Functions

- Specify name without brackets
  - To pass array **`myArray`** to **`myFunction`**

```
int myArray[ 24 ];
myFunction(myArray,24);
```

  - Array size usually passed, but not required
    - Useful to iterate over all elements

Passing Arrays to Functions

- Arrays passed-by-reference
  - Functions can modify original array data
  - Value of name of array is address of first element
    - Function knows where the array is stored
    - Can change original memory locations
- Individual array elements passed-by-value
  - Like regular variables
  - `square( myArray[3] );`

Passing Arrays to Functions

- Functions taking arrays
  - Function prototype
    - `void modifyArray( int b[], int arraySize );`
    - `void modifyArray( int [], int );`
      - Names optional in prototype
    - Both take an integer array and a single integer
  - No need for array size between brackets
    - Ignored by compiler
  - If declare array parameter as **`const`**
    - Cannot be modified (compiler error)
    - `void doNotModify( const int [] );`

```
// Passing arrays and individual array elements to functions.
#include <iostream>

using namespace std;

void modifyArray( int [], int );  // appears strange
void modifyElement( int );

int main()
{
   const int arraySize = 5;                  // size of array a
   int a[ arraySize ] = { 0, 1, 2, 3, 4 };   // initialize a

   cout << "Effects of passing entire array by reference:"
        << "\n\nThe values of the original array are:\n";

   // output original array
   for ( int i = 0; i < arraySize; i++ )
      cout << "\t" << a[ i ];
```

Syntax for accepting an array in parameter list.

```
   cout << endl;

   // pass array a to modifyArray
   modifyArray( a, arraySize );

   cout << "The values of the modified array are:\n";

   // output modified array
   for ( int j = 0; j < arraySize; j++ )
      cout << setw( 3 ) << a[ j ];

   // output value of a[ 3 ]
   cout << "\n\n\n"
        << "Effects of passing array element by value:"
        << "\n\nThe value of a[3] is

   // pass array element a[ 3 ] by v
   modifyElement( a[ 3 ] );

   // output value of a[ 3 ]
   cout << "The value of a[3] is " << a[ 3 ] << endl;

   return 0;  // indicates successful termination

} // end main
```

Pass array name (**a**) and size to function. Arrays are passed-by-reference.

Pass a single array element by value; the original cannot be modified.

```
// in function modifyArray, "b" p
 // the original array "a" in memo
 void modifyArray( int b[], int si
 {
    // multiply each array element
    for ( int k = 0; k < sizeOfAr
       b[ k ] *= 2;

 } // end function modifyArray

 // in function modifyElement, "
 // array element a[ 3 ] passed
 void modifyElement( int e )
 {
    // multiply parameter by 2
    cout << "Value in modifyElement is "
         << ( e *= 2 ) << endl;

 } // end function modifyElement
```

Although named **b**, the array points to the original array **a**. It can modify **a**'s data.

Individual array elements are passed by value, and the originals cannot be changed.

```
Effects of passing entire array by reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8


Effects of passing array element by value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```

```
// Demonstrating the const type qualifier.
#include <iostream>

using namespace std;

void tryToModifyArray( const int [] );  // function
prototype

int main()
{
   int a[] = { 10, 20, 30 };

   tryToModifyArray( a );

   cout << a[ 0 ] << ' ' << a[ 1 ] <<
'\n';

   return 0;  // indicates successful termination

} // end main
```

Array parameter declared as **const**. Array cannot be modified, even though it is passed by reference.

```
// In function tryToModifyArray, "b" cannot be used
// to modify the original array "a" in main.
void tryToModifyArray( const int b[] )
{
   b[ 0 ] /= 2;    // error
   b[ 1 ] /= 2;    // error
   b[ 2 ] /= 2;    // error

} // end function tryToModifyArray
```

```
d:\cpphtp4_examples\ch04\Fig04_15.cpp(26) : error C2166:
   l-value specifies const object
d:\cpphtp4_examples\ch04\Fig04_15.cpp(27) : error C2166:
   l-value specifies const object
d:\cpphtp4_examples\ch04\Fig04_15.cpp(28) : error C2166:
   l-value specifies const object
```

Comparing Algorithms

- When choosing a particular algorithm there are three things to consider:

1) How easy is the algorithm to understand and implement?
2) How fast is the algorithm?
3) How much memory does the algorithm use?

- Implementing an algorithm because it is easy to write and debug, even if it is slow is often the right choice.
- We can sometimes trade time (execution time) for space (memory) and vice versa.
- Whether time is more important than space depends on the particular problem you have.

Sorting Arrays

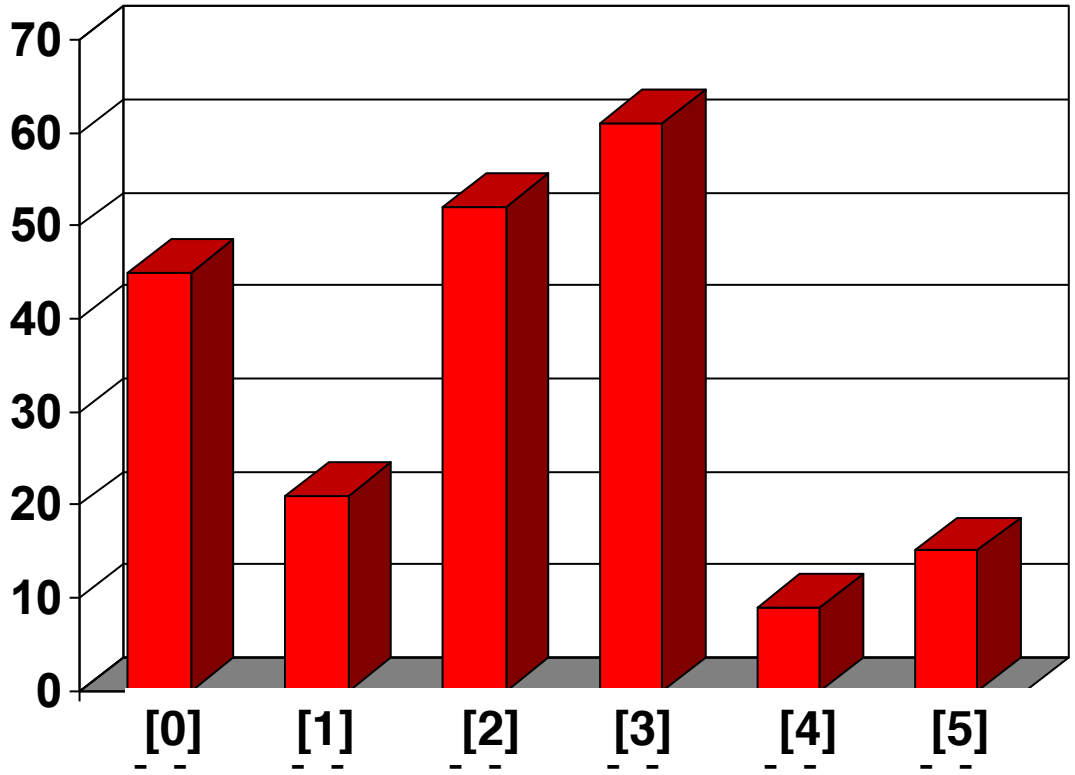
- Sorting data
- Definition of a sorting:
  - Given an array of values [a1, a2, a3, a4, …, an] produce a permutation of that array such that: [a1 ≤ a2 ≤ a3 ≤ a4 … ≤ an].
  - Important computing application
  - Virtually every organization must sort some data
    - Massive amounts must be sorted

Selection Sort

- Think of the array as being split into two – the sorted part and the unsorted part. (the sorted array initially has zero elements)
  - Find the smallest element *i* of the unsorted part.
  - Swap *i* with the element at the end of the sorted part of the array.
  - This increases the size of the sorted portion by one element and decreases the size of the unsorted array.
  - When the unsorted part has size zero and the sorted part has all the elements we are done.

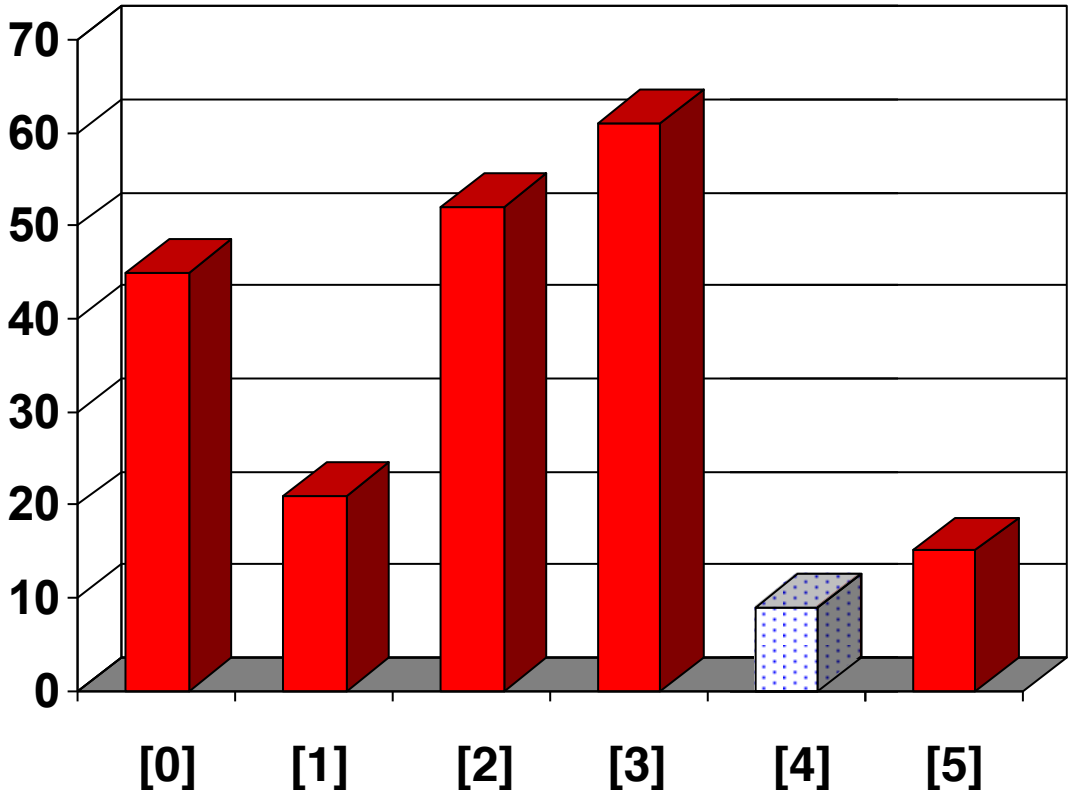
Selection Sort

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



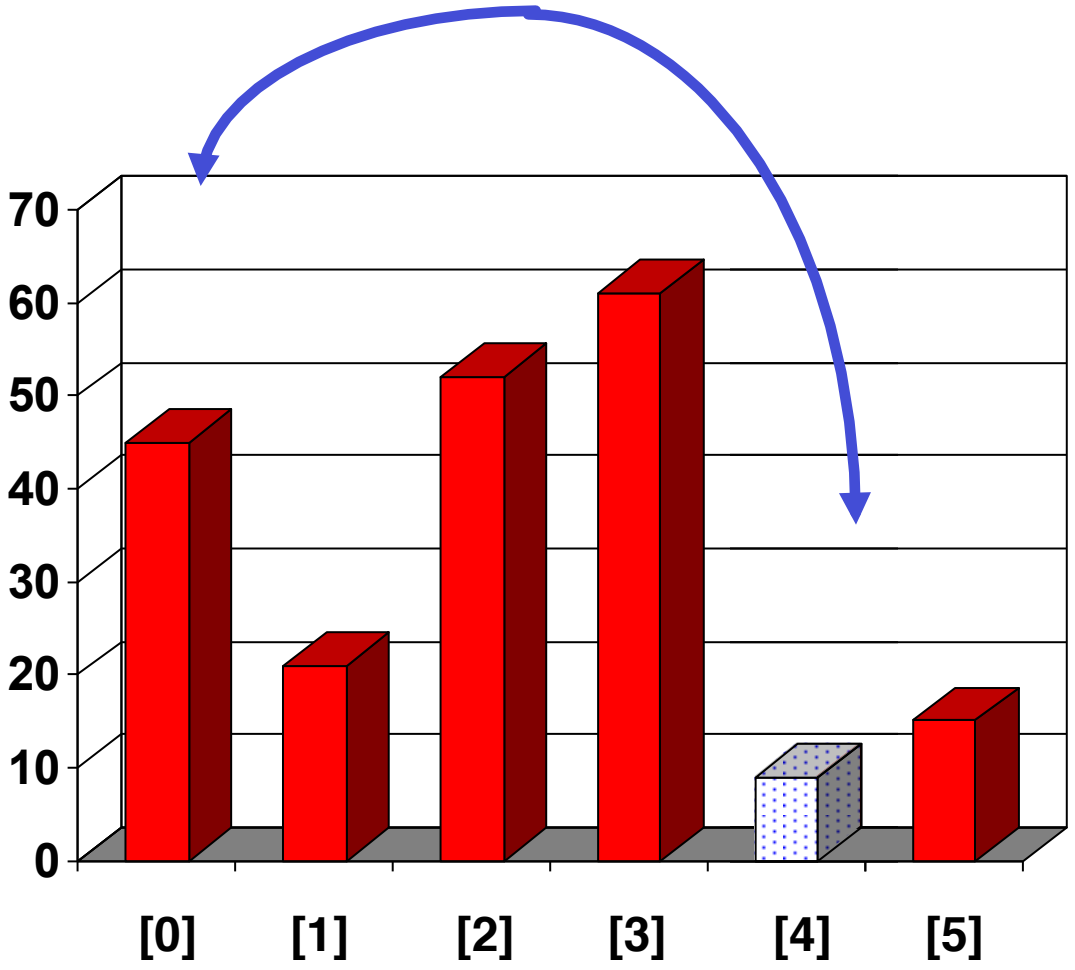
Selection Sort

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



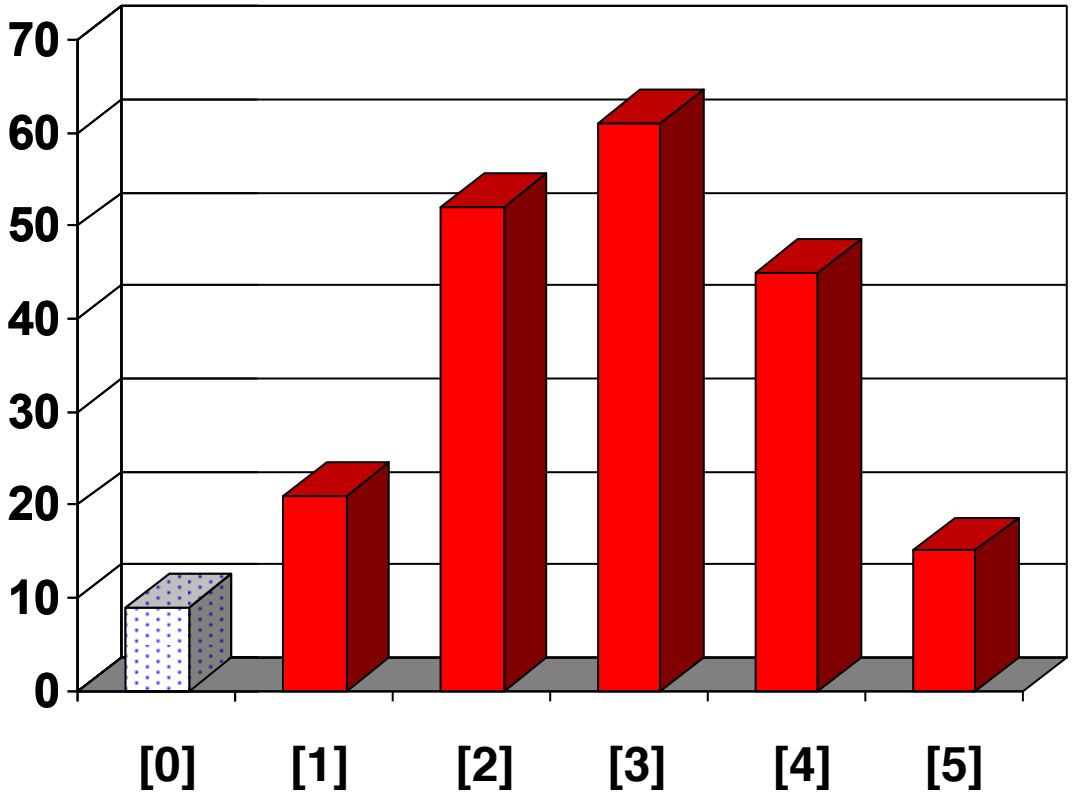
Selection Sort

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



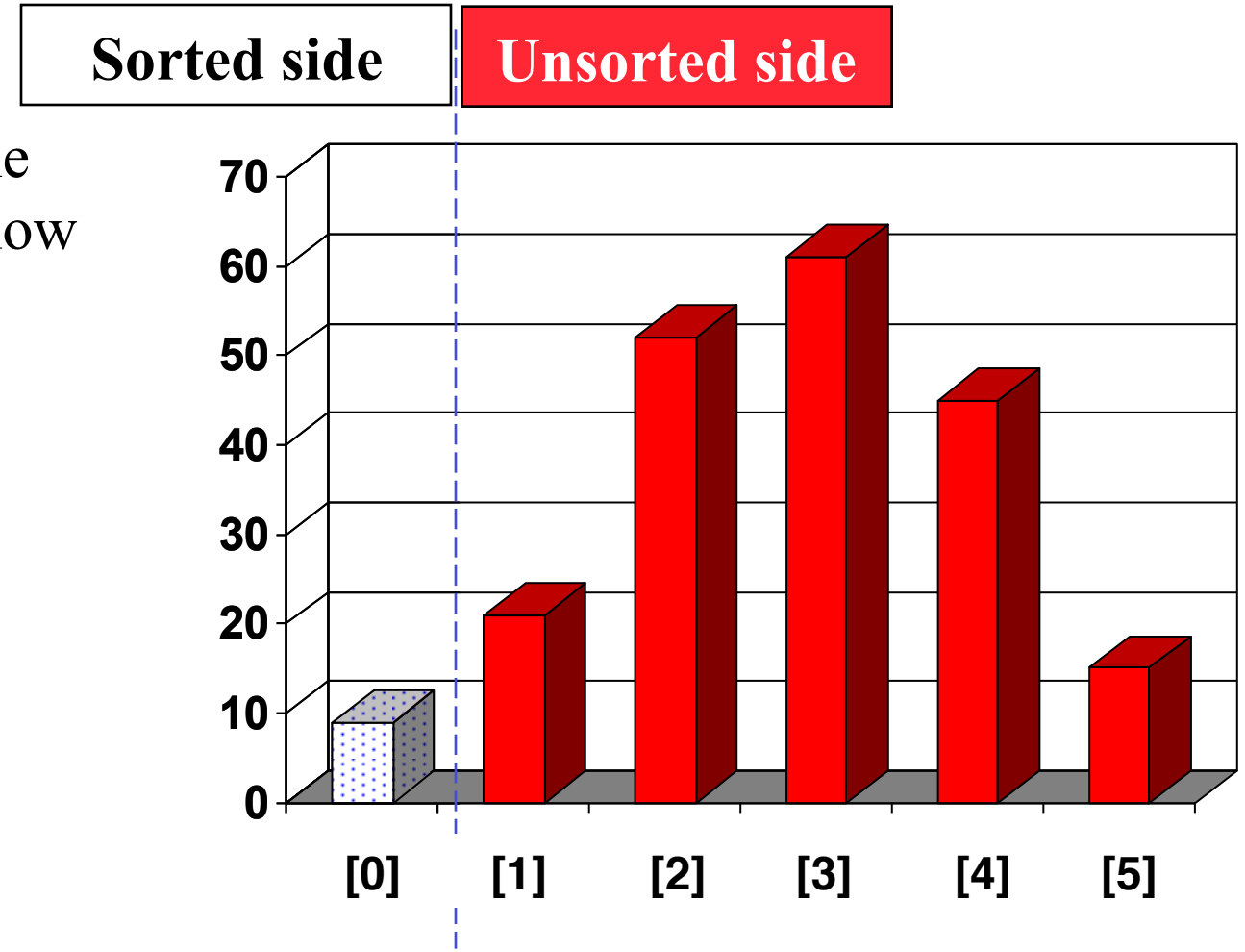
Selection Sort

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



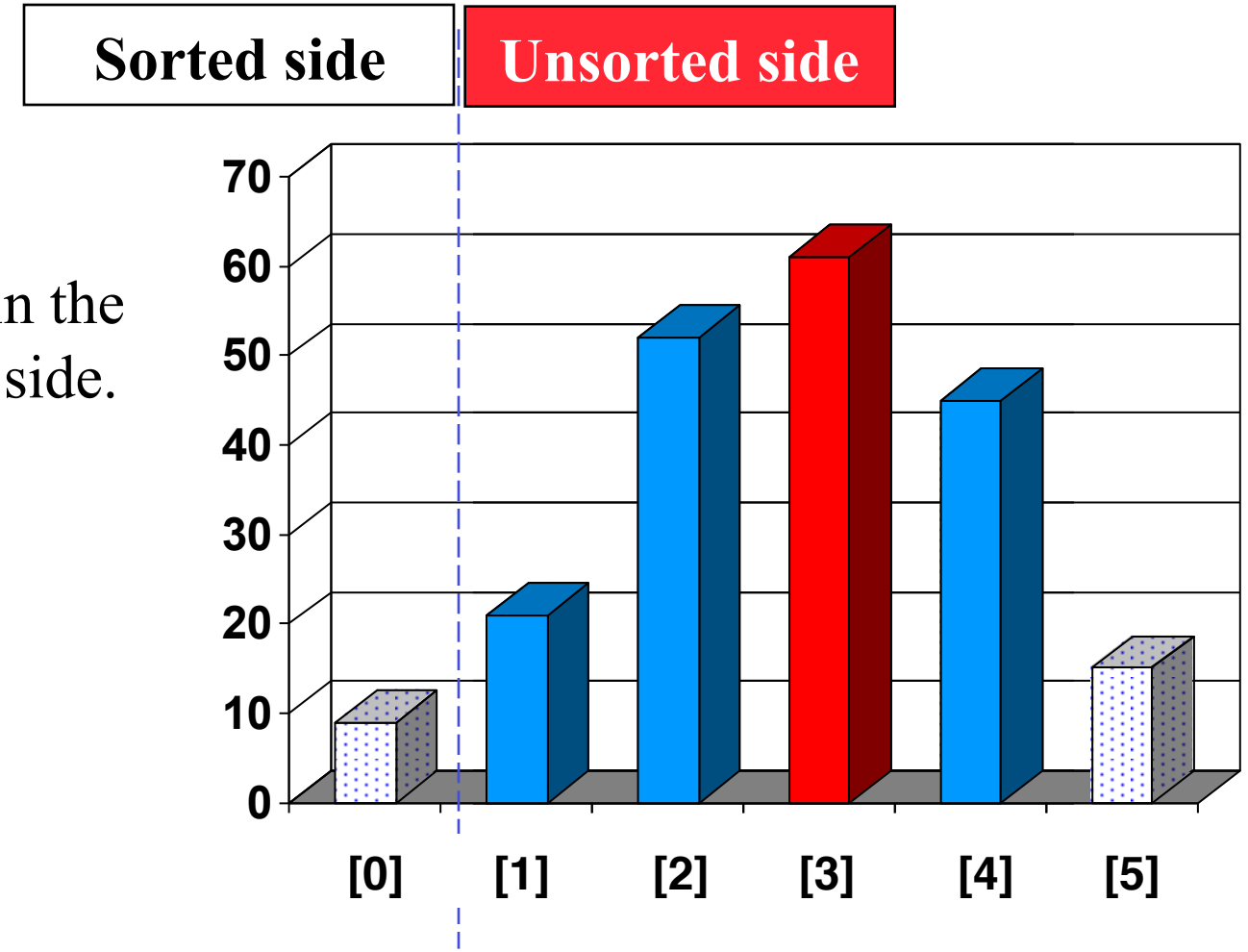
Selection Sort

- Part of the array is now sorted.



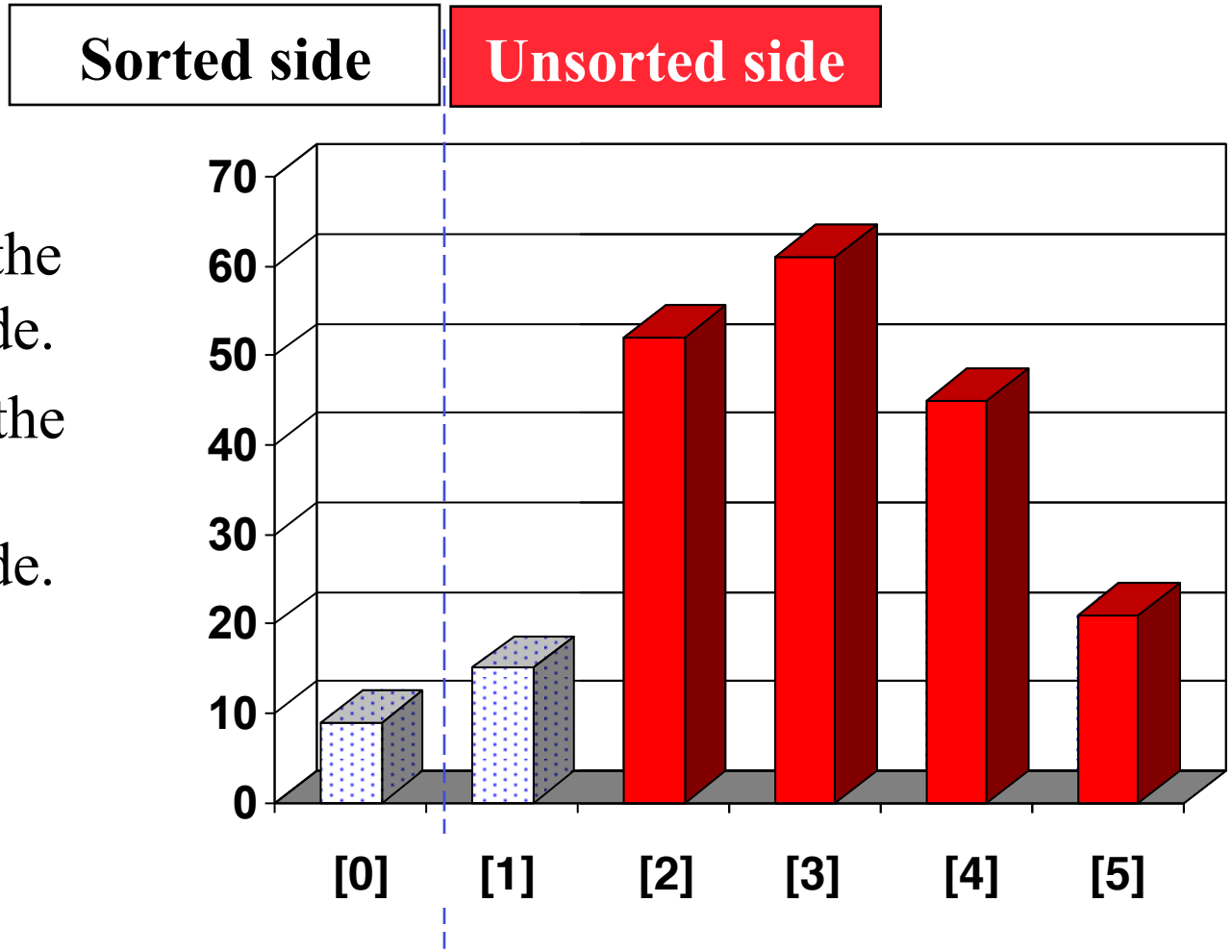
Selection Sort

- Find the smallest element in the unsorted side.



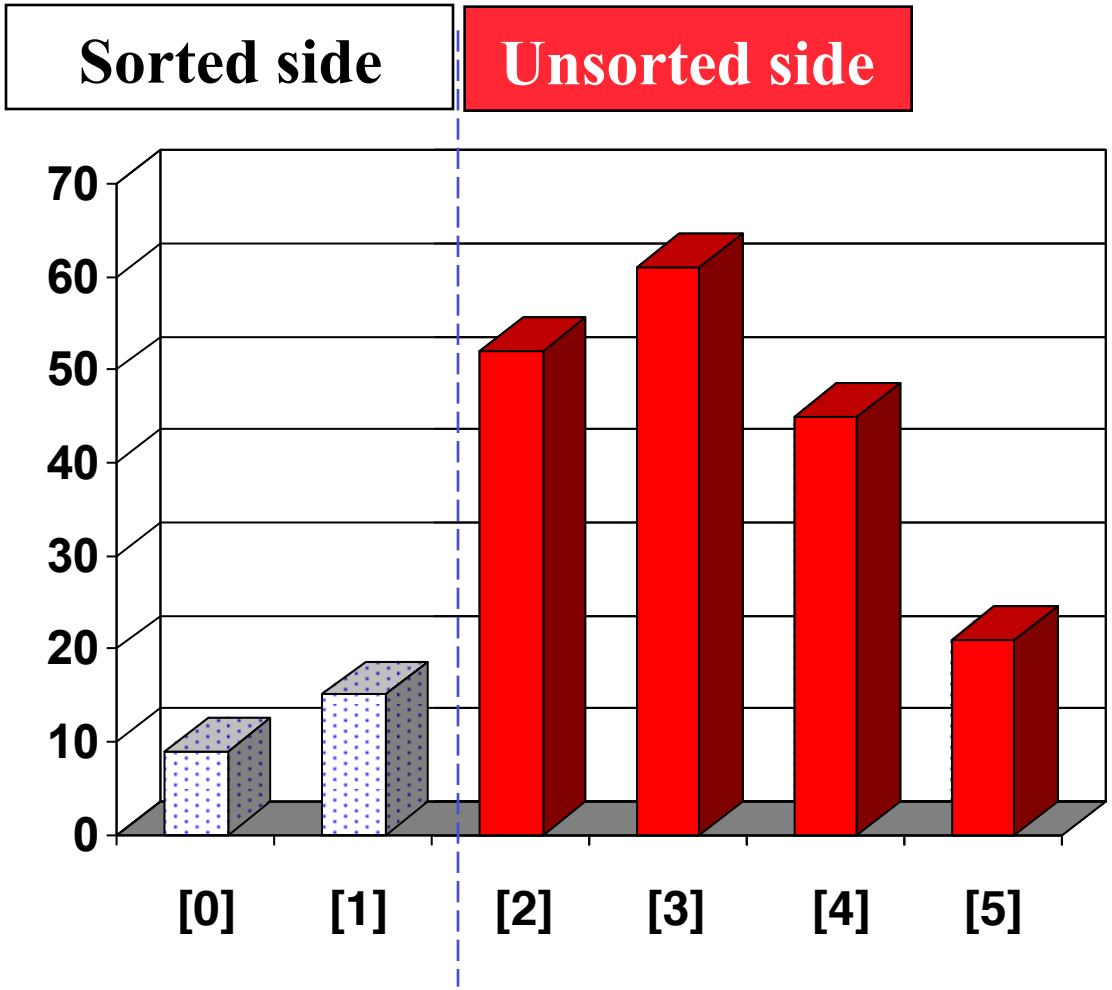
Selection Sort

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



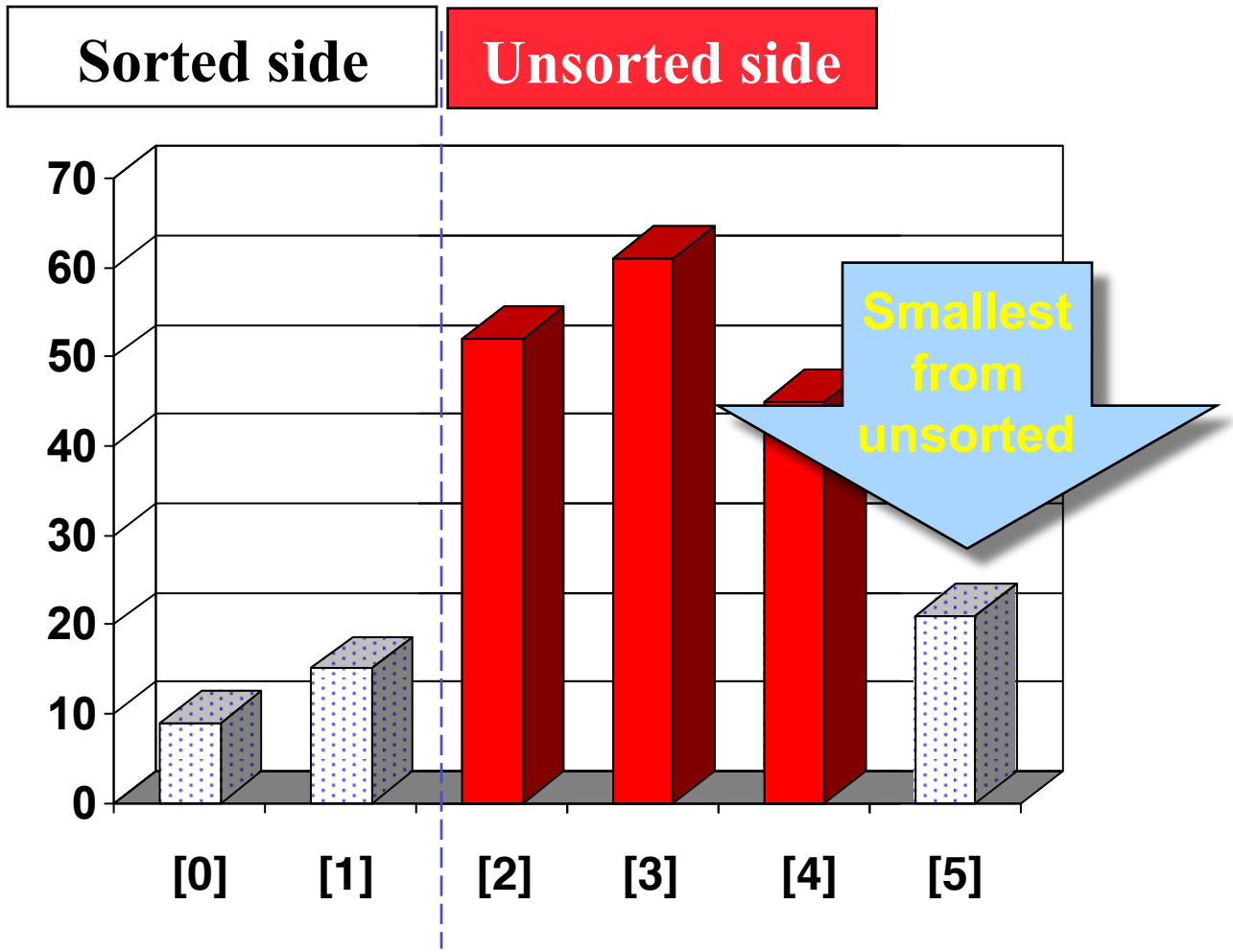
Selection Sort

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



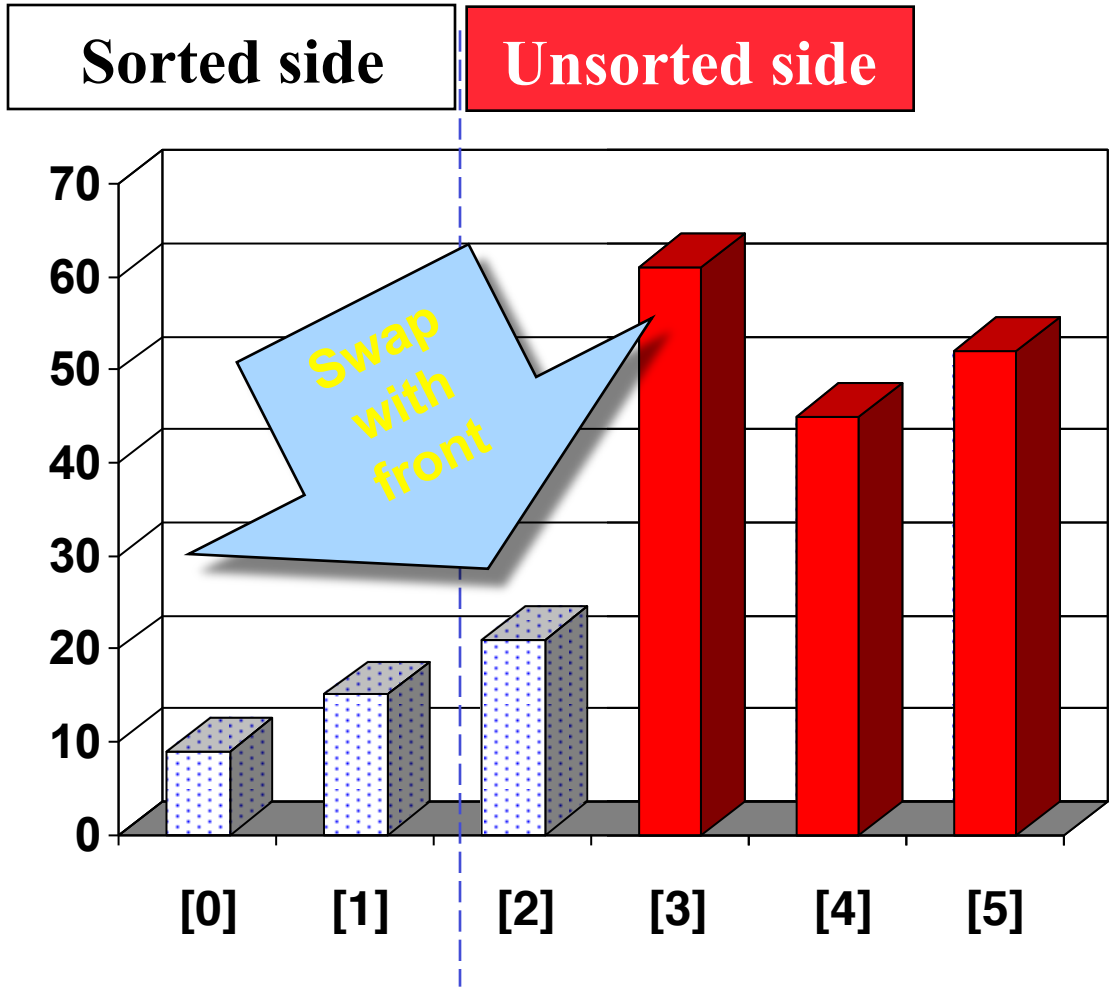
Selection Sort

- The process continues...

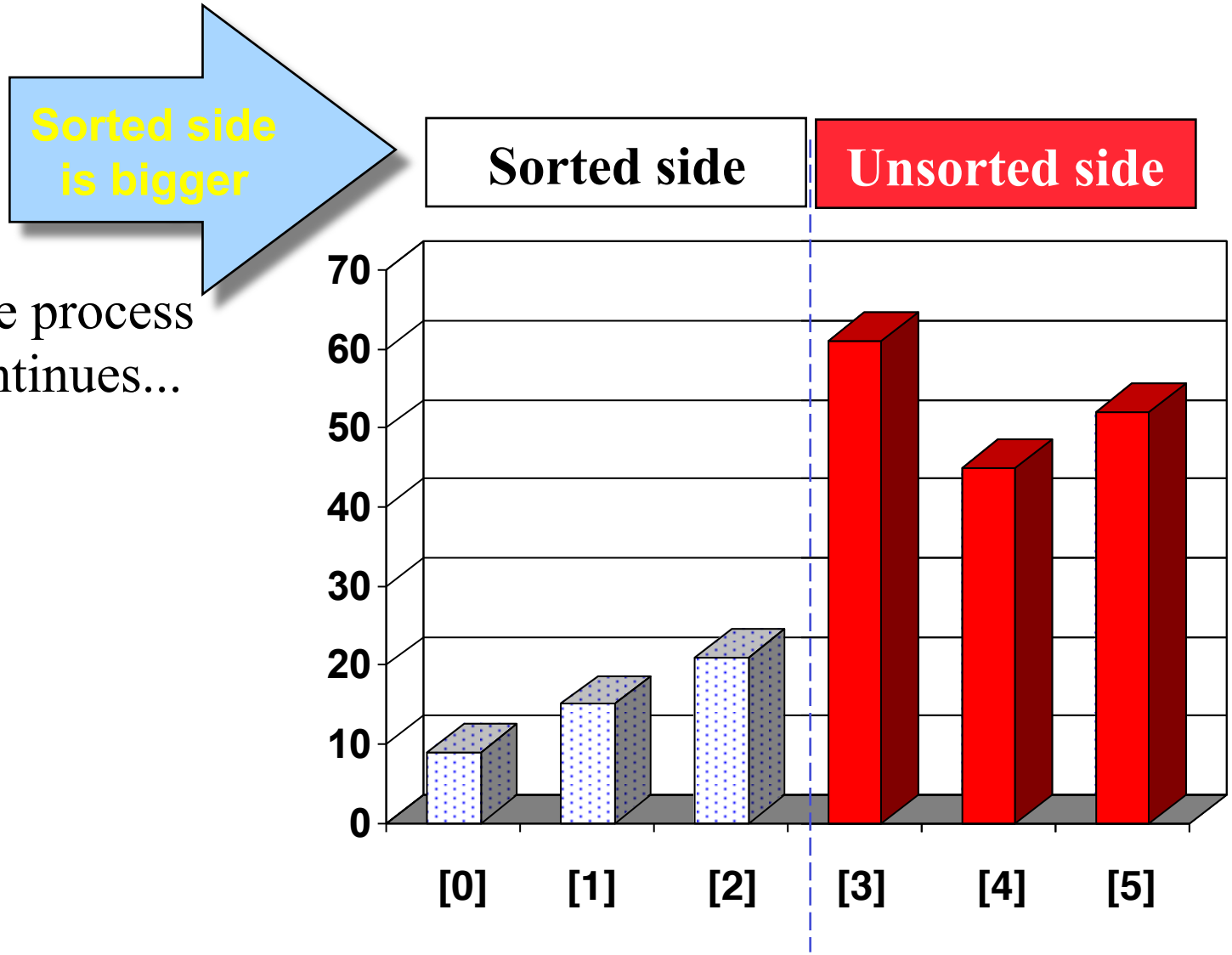


Selection Sort

- The process continues...



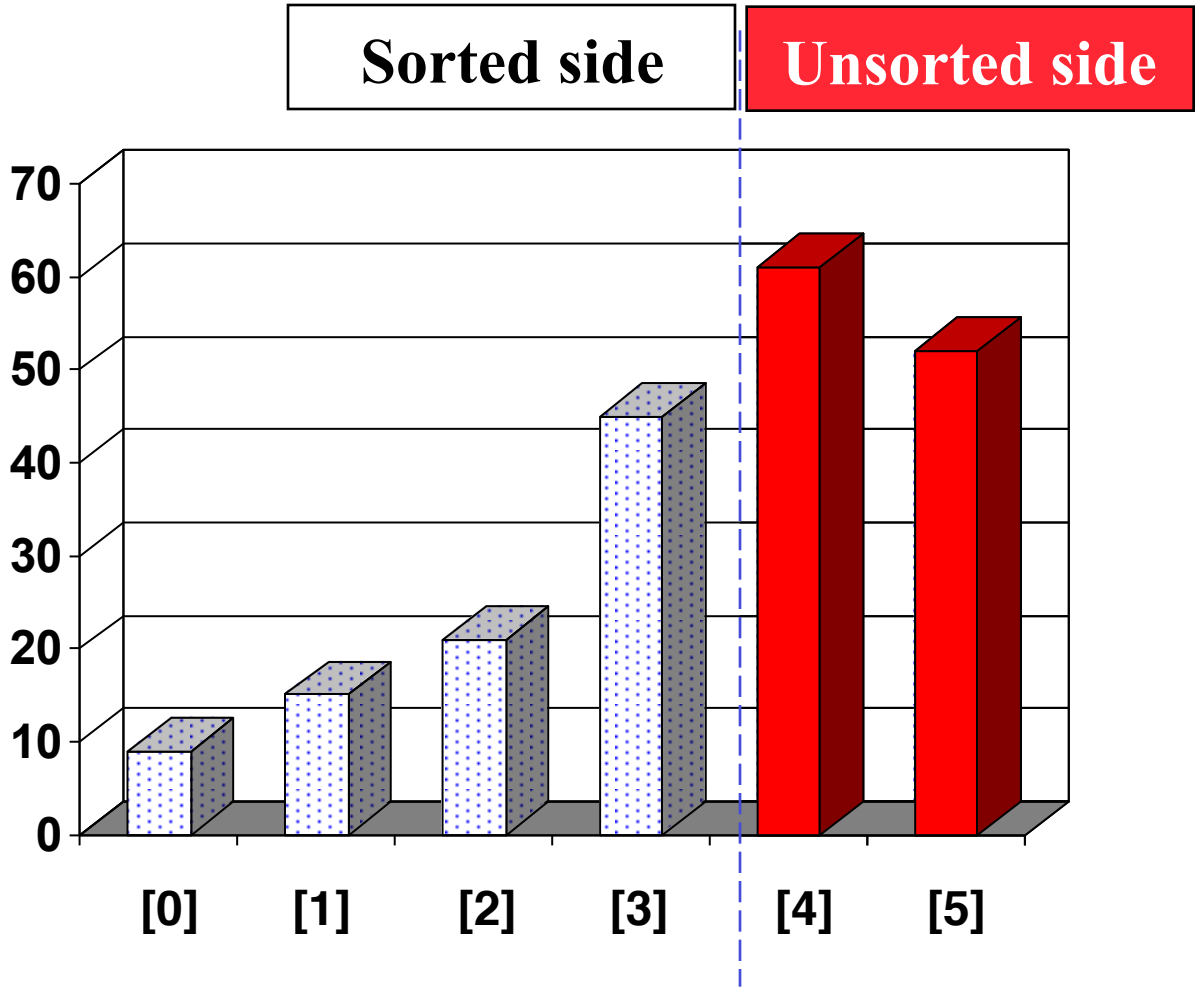
Selection Sort



- The process continues...

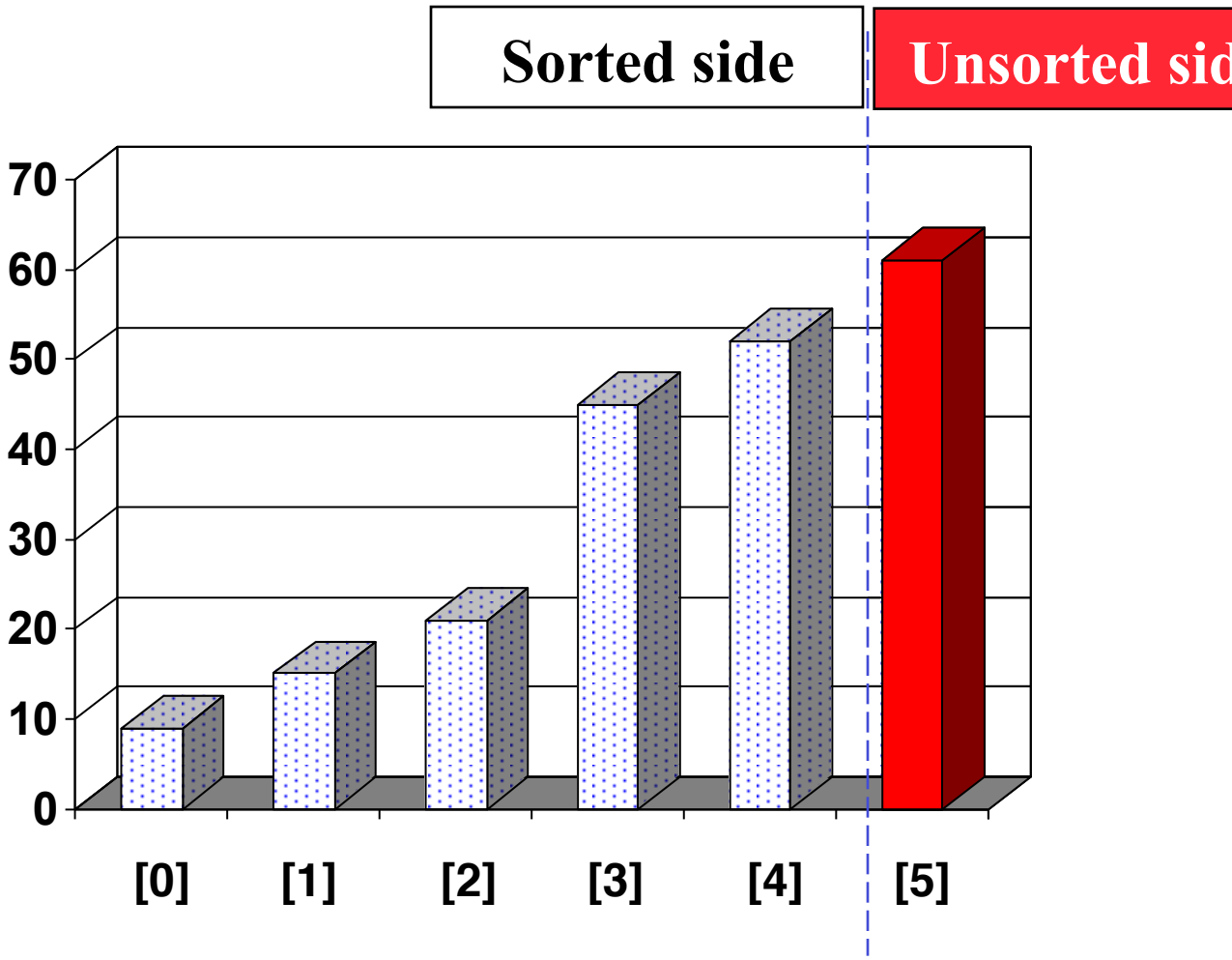
Selection Sort

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



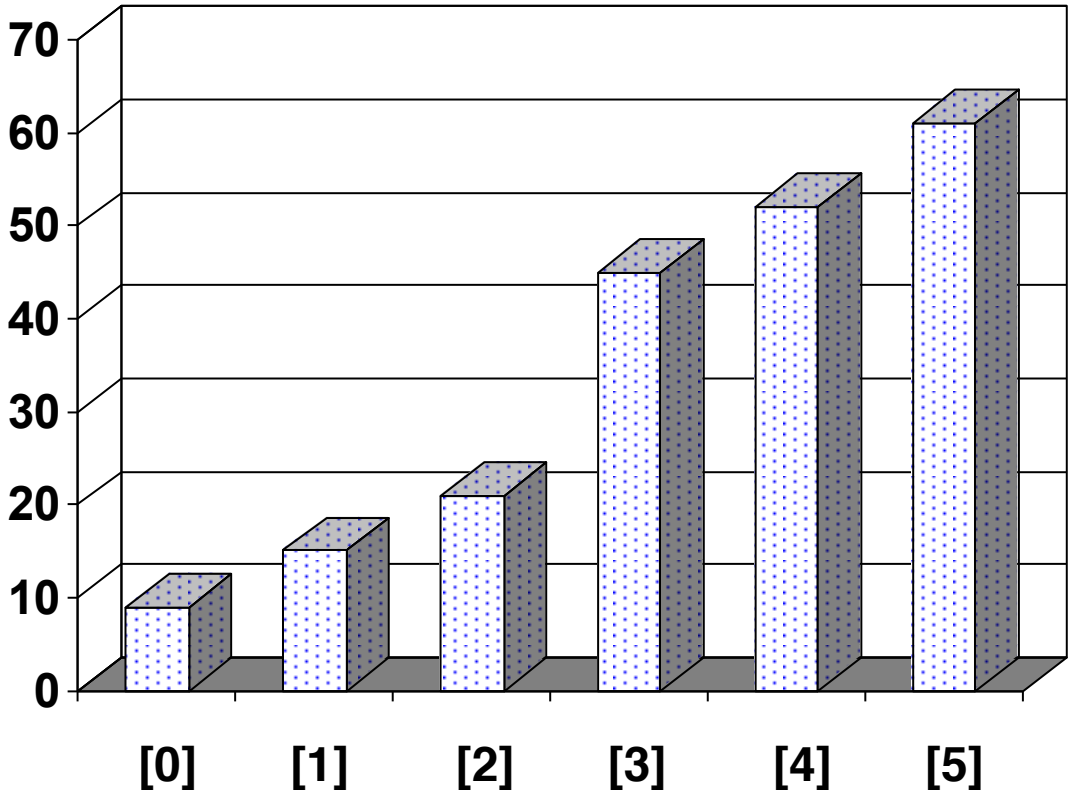
Selection Sort

- 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 code (Iterative Version):

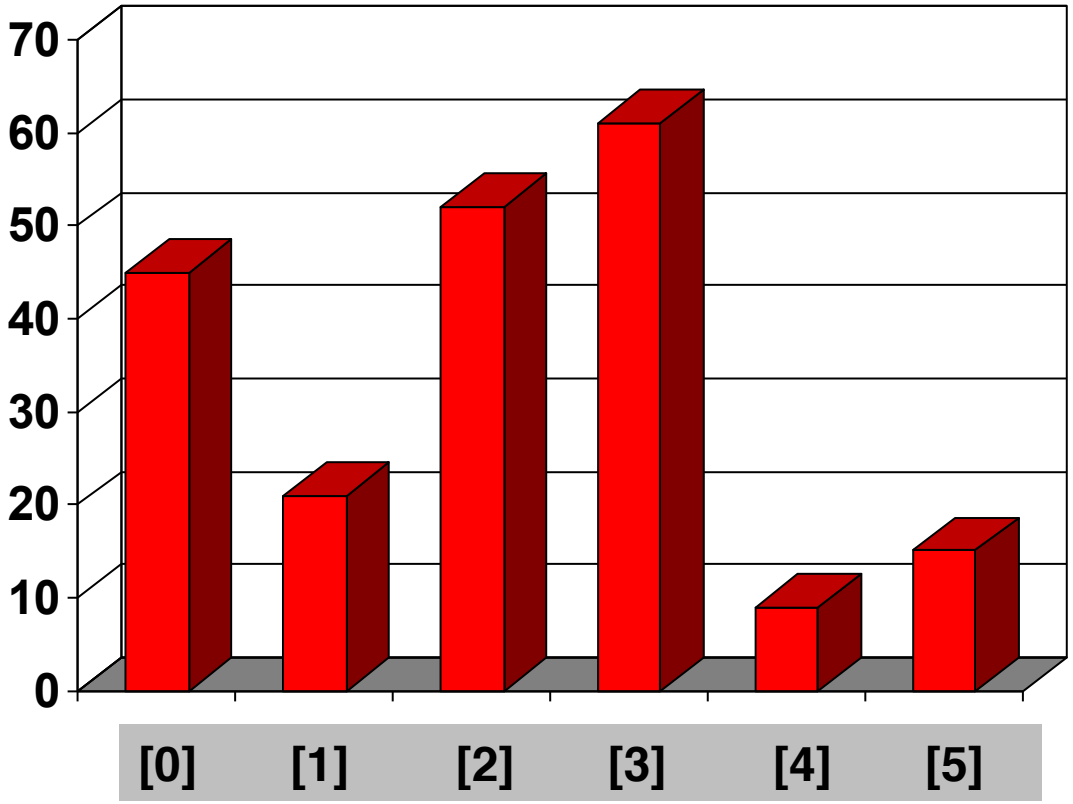
```
template <typename Item>
void selectionsort( Item a[], int left, int right)
{
        for ( int i = left; i < right; i++ )
        {
                int min = i;

                for ( int j = i + 1; j < = right; j++ )
                {
                        if ( a[j] < a[min])
                        {
                                min = j;
                        }
                        swap( a[j], a[min] );
                }
        }
}
```

Sorting Algorithms

- Insertion Sort (Iterative Version)
- Outer Loop
  - In turn take each element *i* of the input array *input_array* to be sorted and *insert* it into a new array *sorted_array* (or if we are clever the same array).
- Inner Loop
  - Iterate through sorted_array until an element smaller than i is found. Put

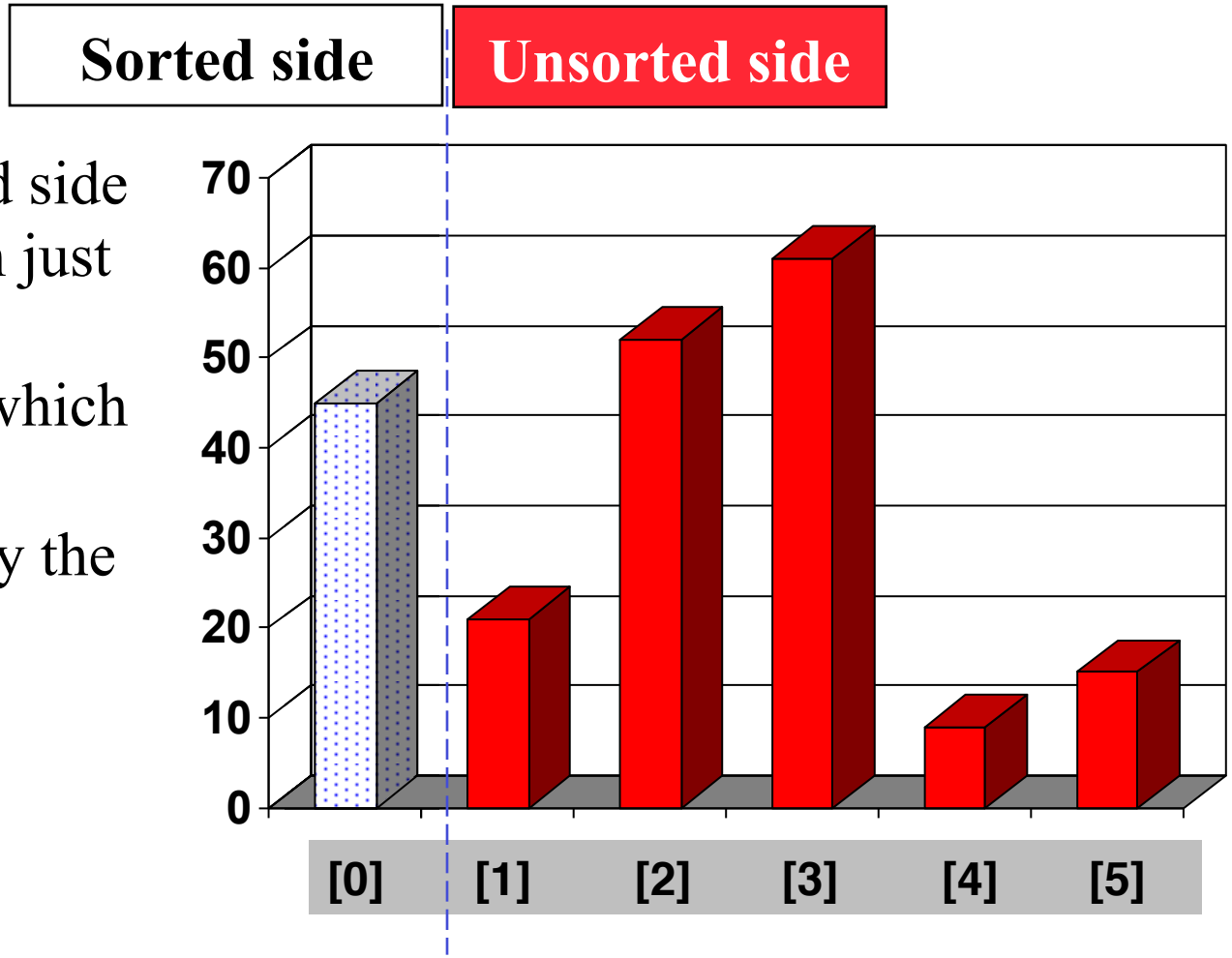
Insertion Sort

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



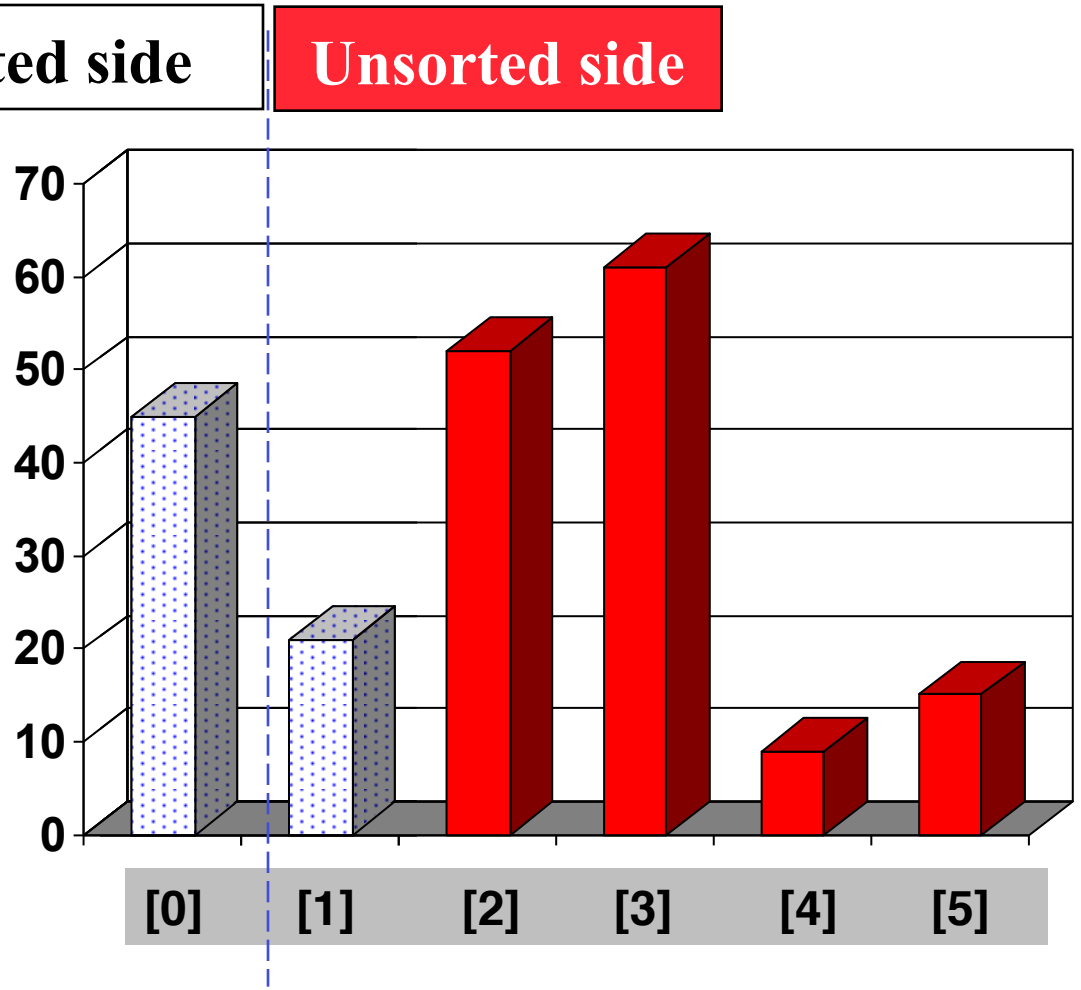
Insertion Sort

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



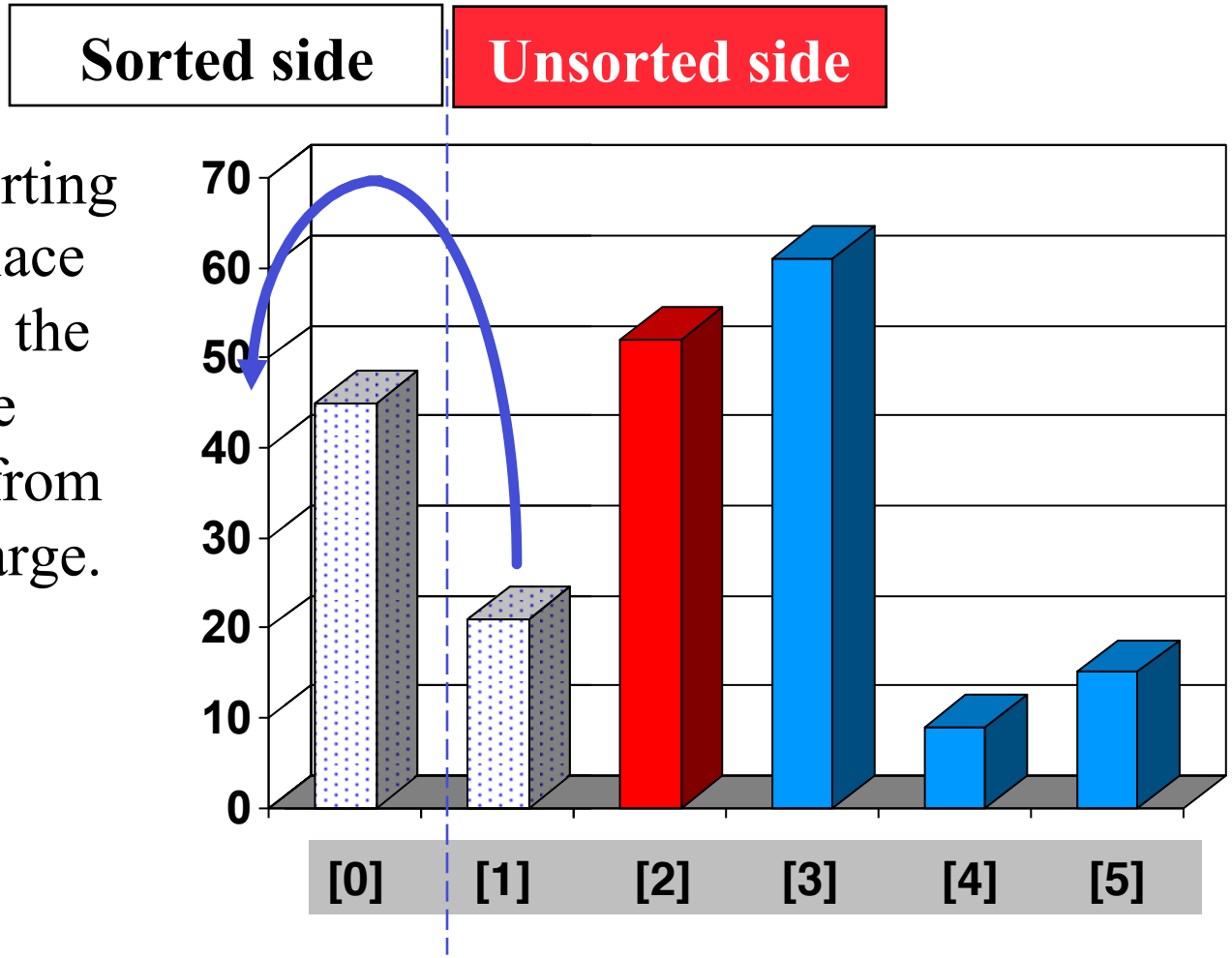
Insertion Sort

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



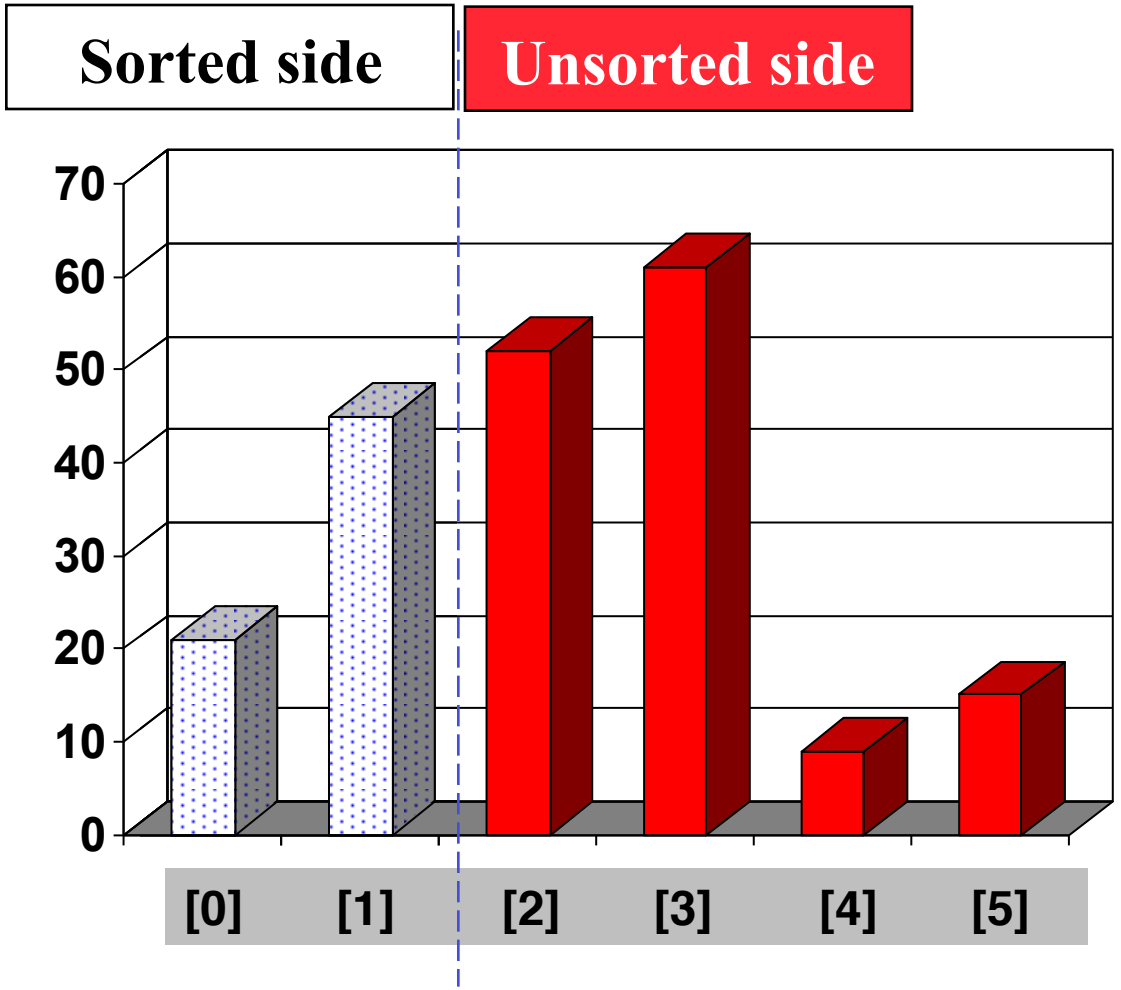
Insertion Sort

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



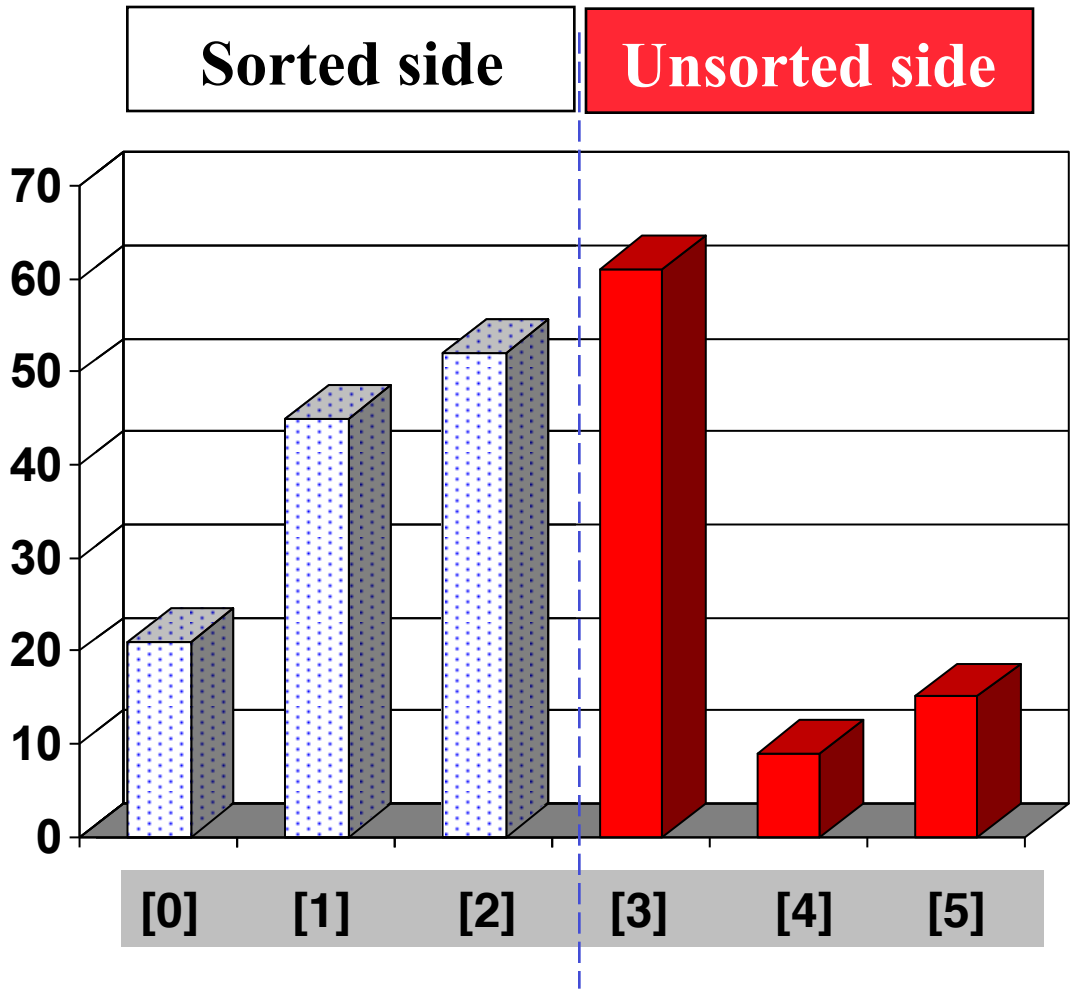
Insertion Sort

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



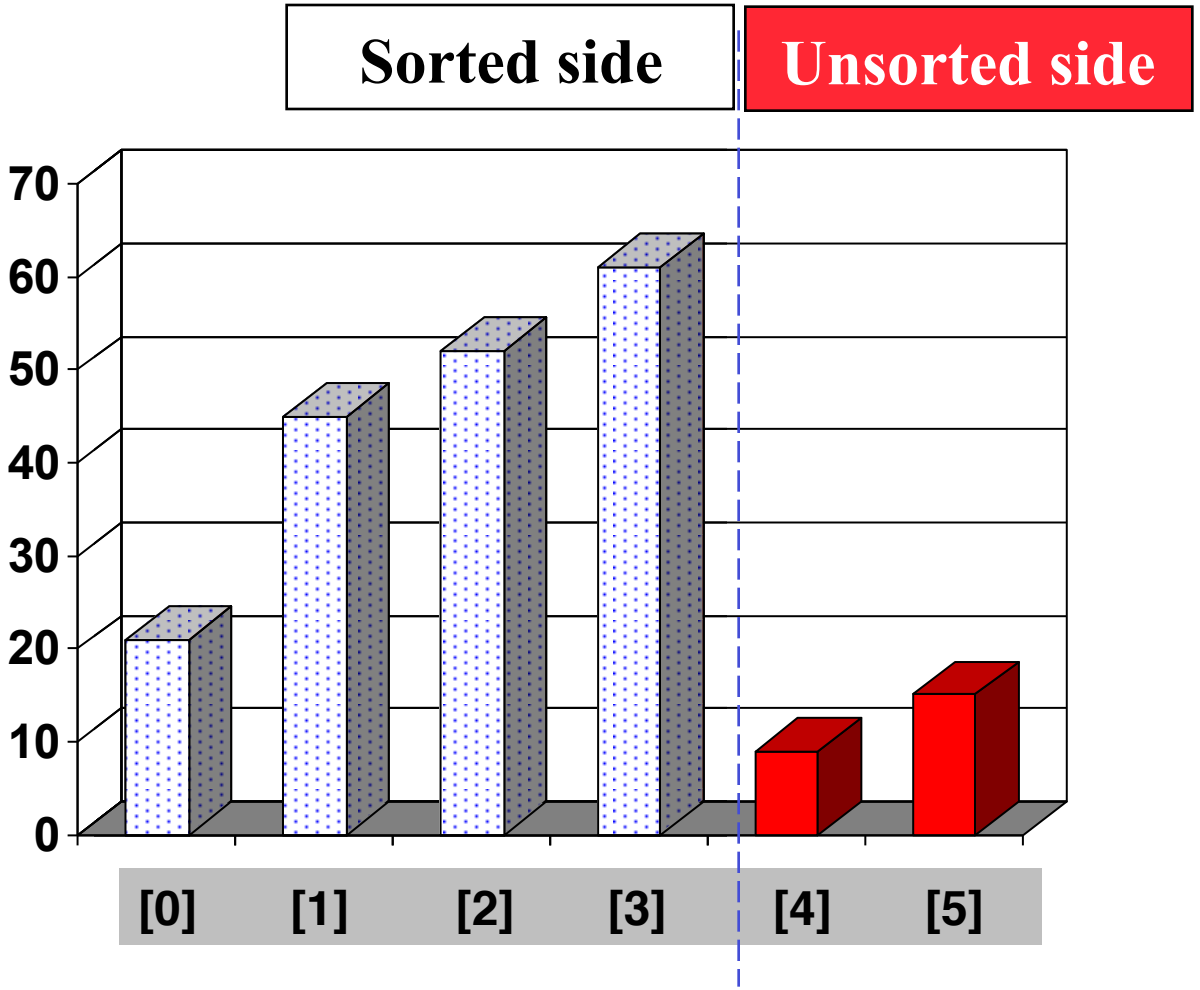
Insertion Sort

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



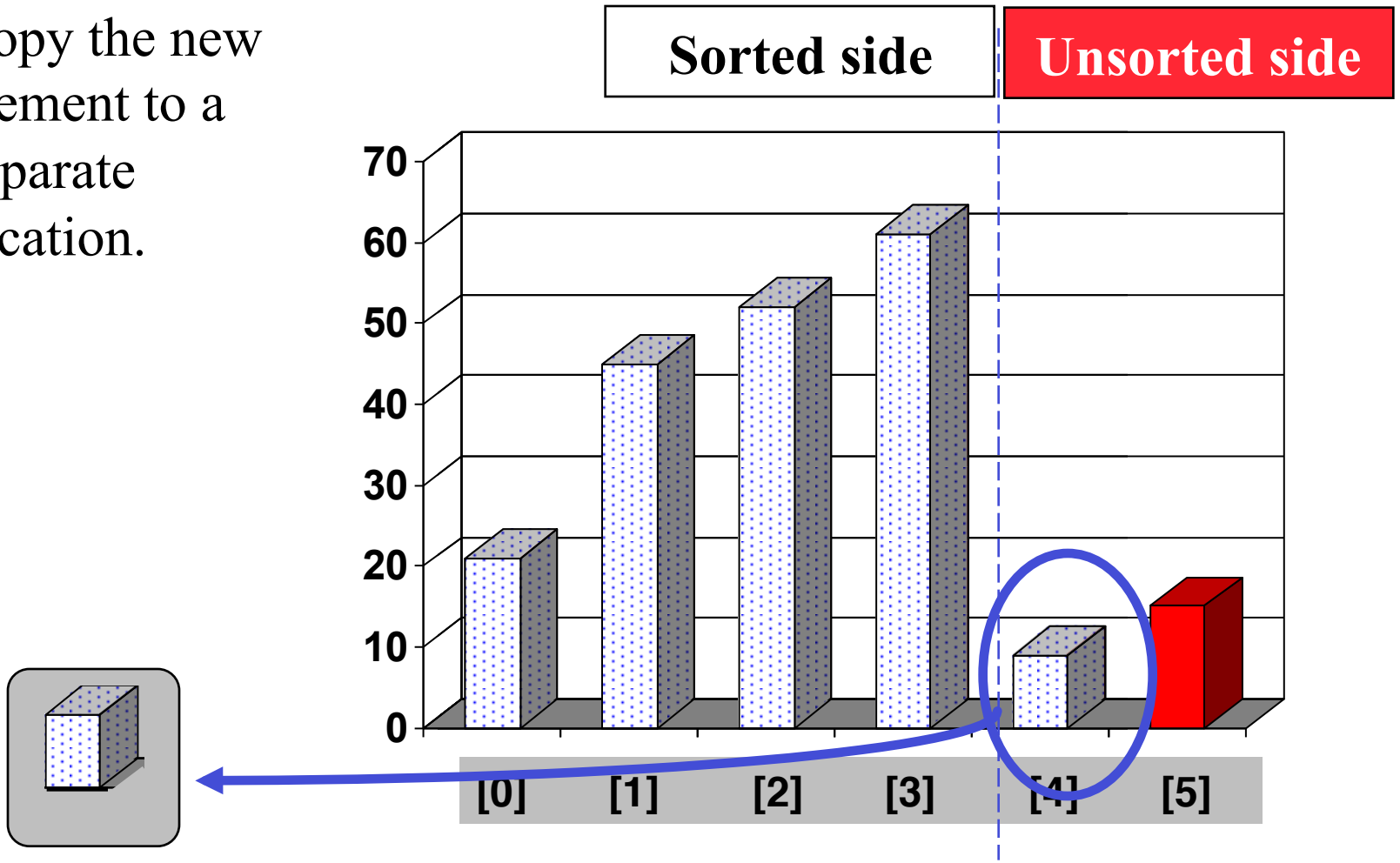
Insertion Sort

- Sometimes we are lucky twice in a row.



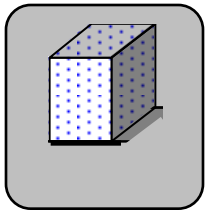
Insertion Sort

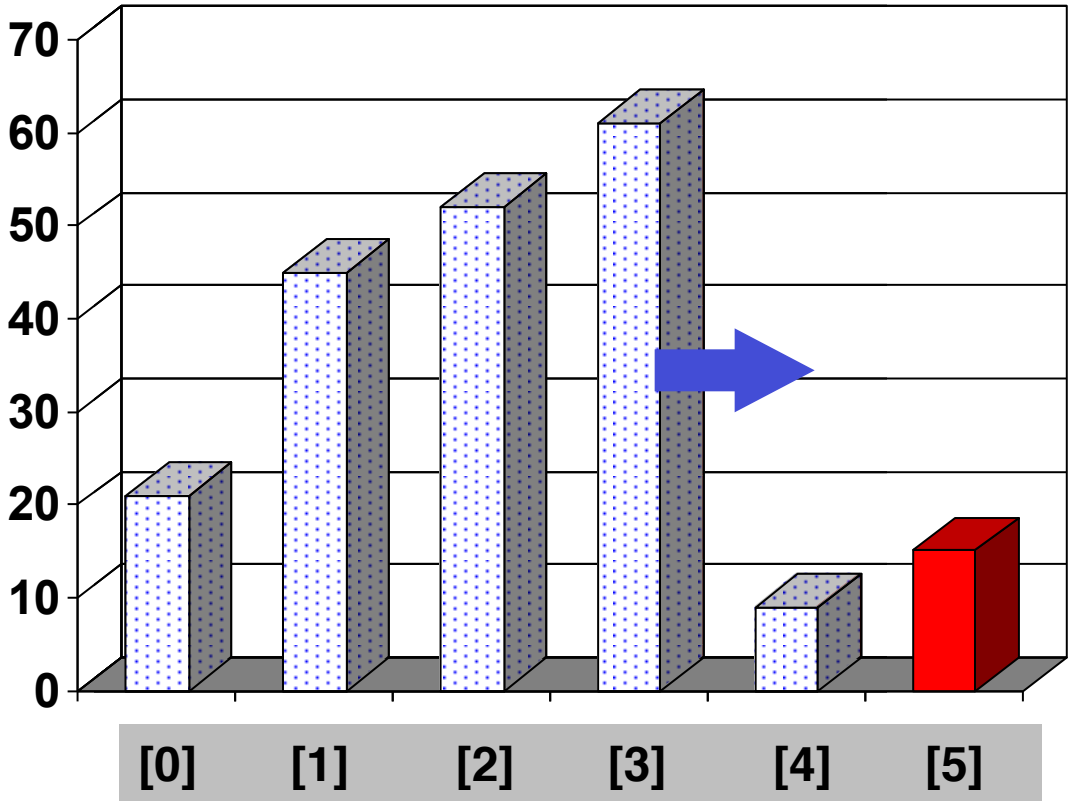
❶ Copy the new element to a separate location.



Insertion Sort

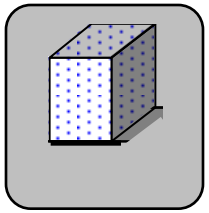
❷ Shift elements in the sorted side, creating an open space for the new element.

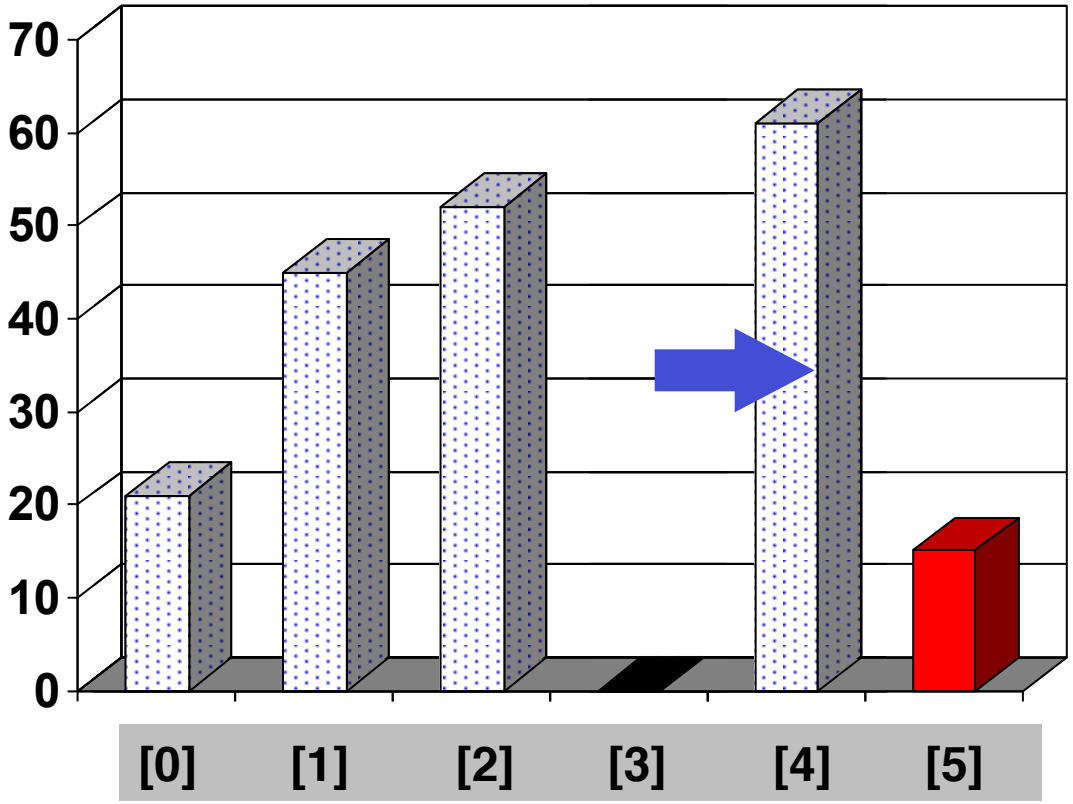




Insertion Sort

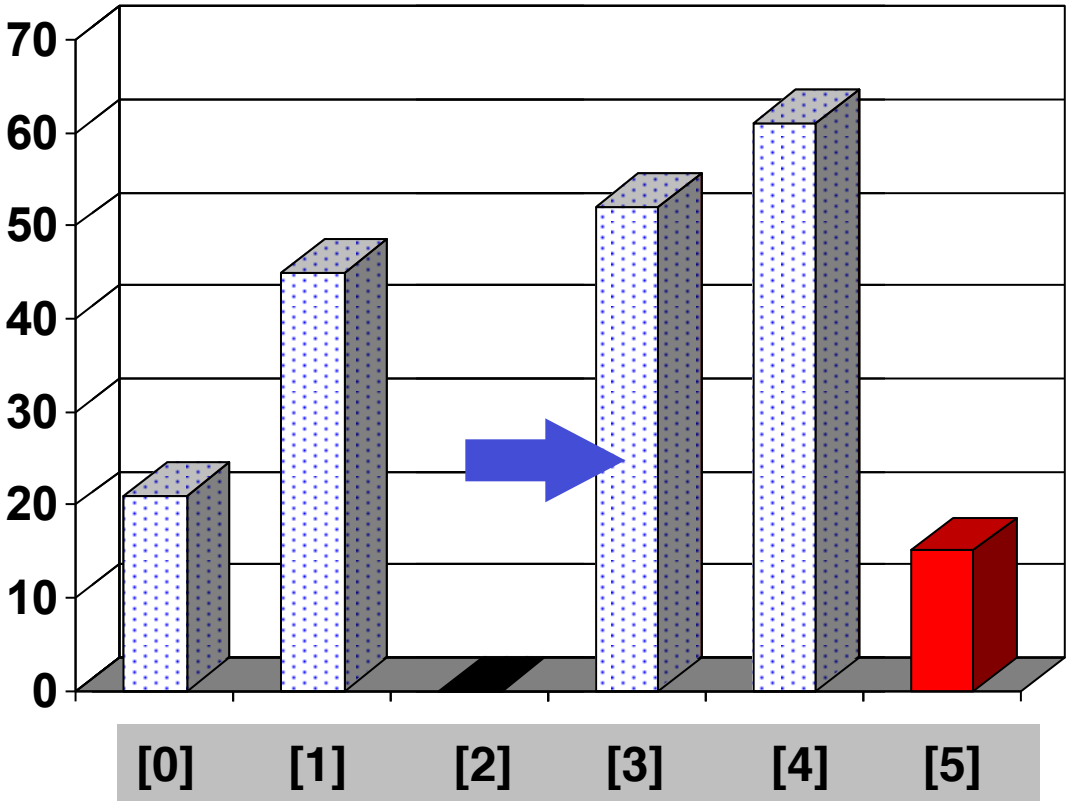
❷ Shift elements in the sorted side, creating an open space for the new element.

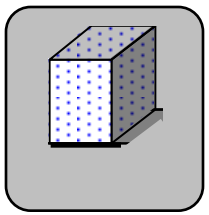




Insertion Sort

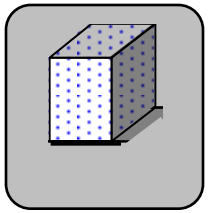
❷ Continue shifting elements...

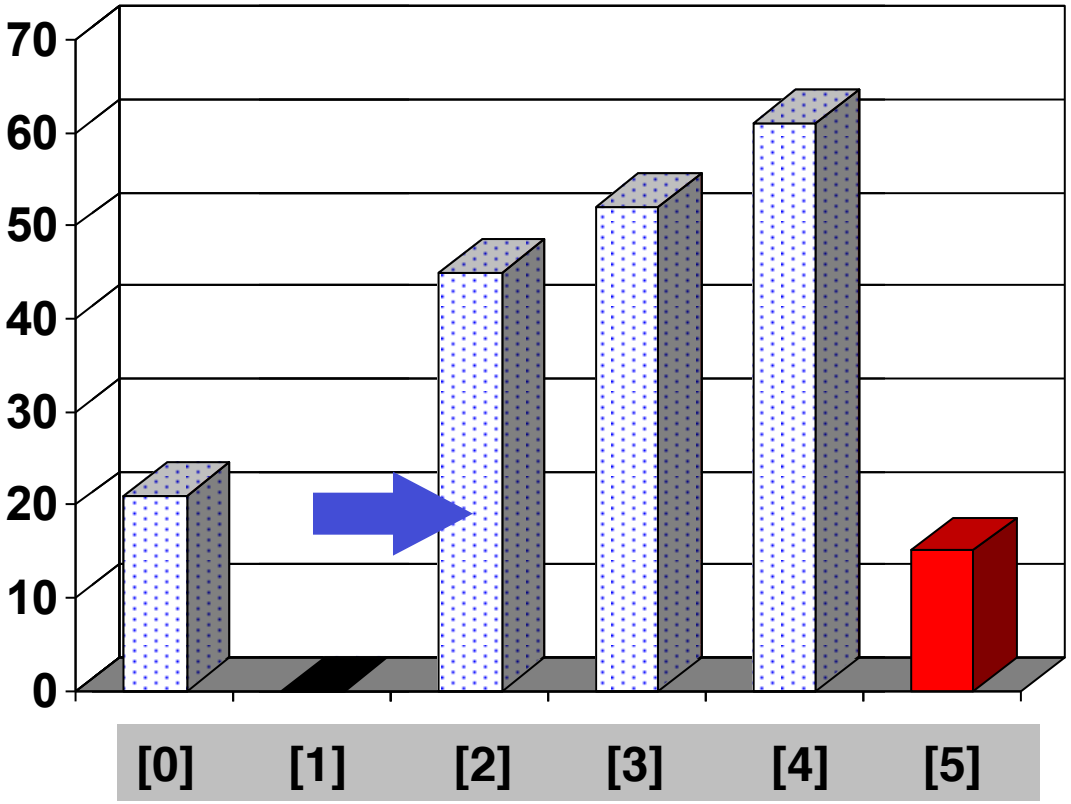




Insertion Sort

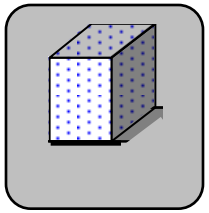
❷ Continue shifting elements...

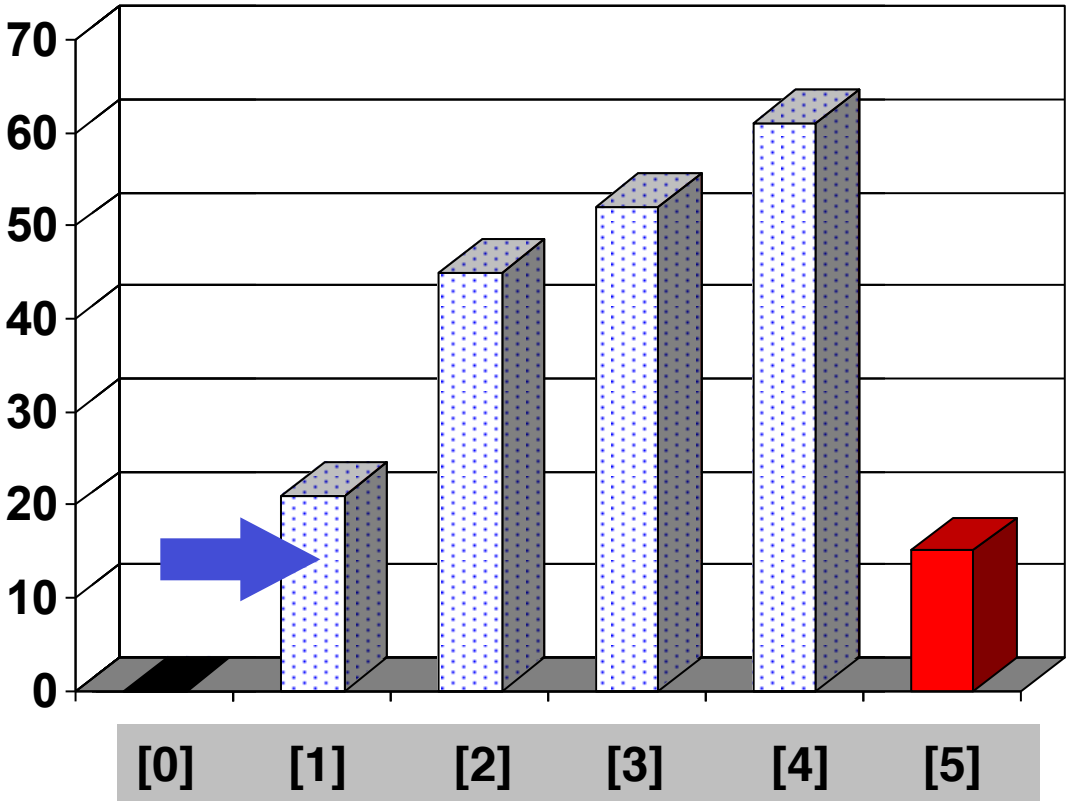




Insertion Sort

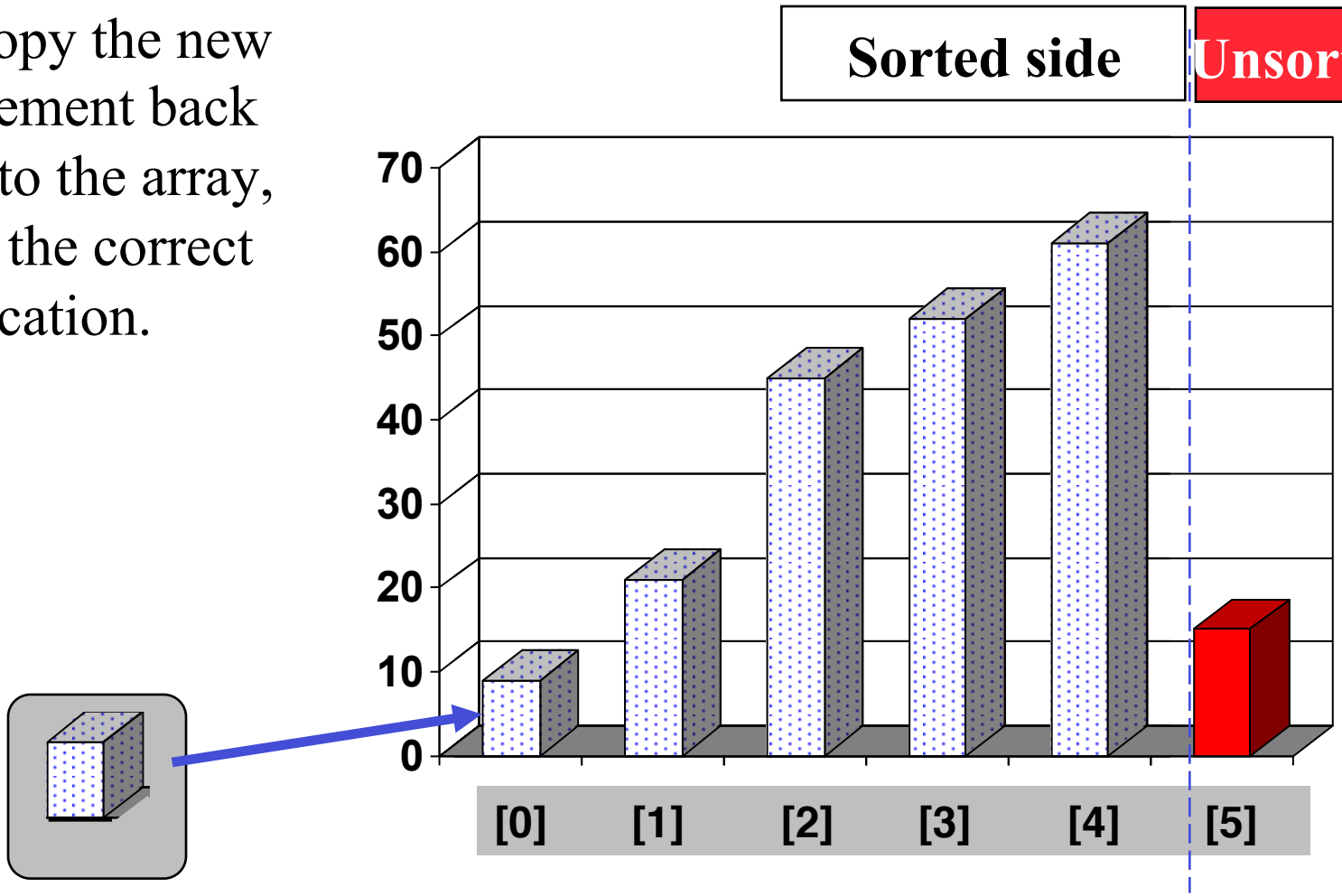
❷ ...until you reach the location for the new element.





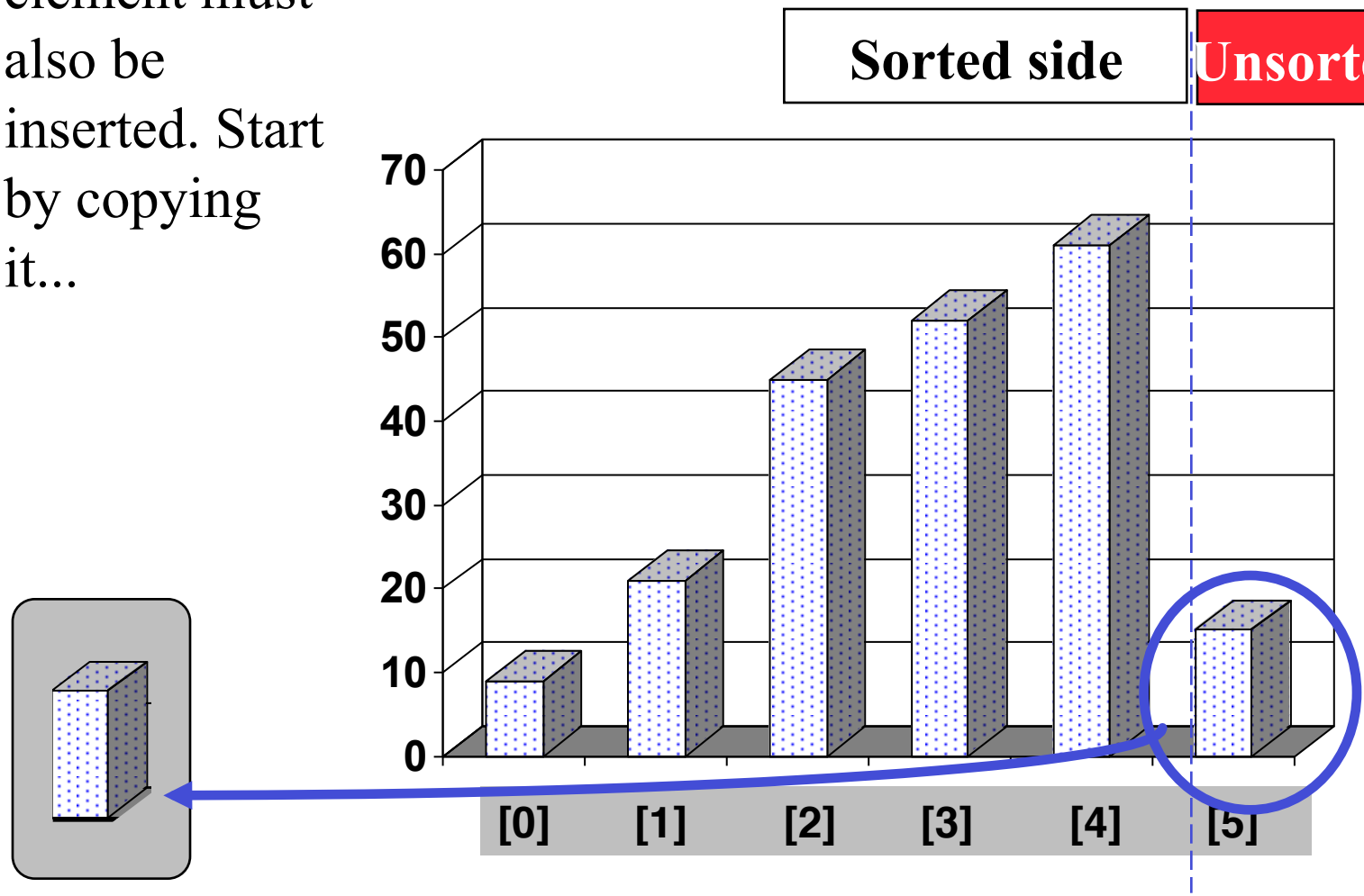
Insertion Sort

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



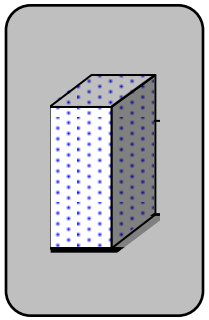
Insertion Sort

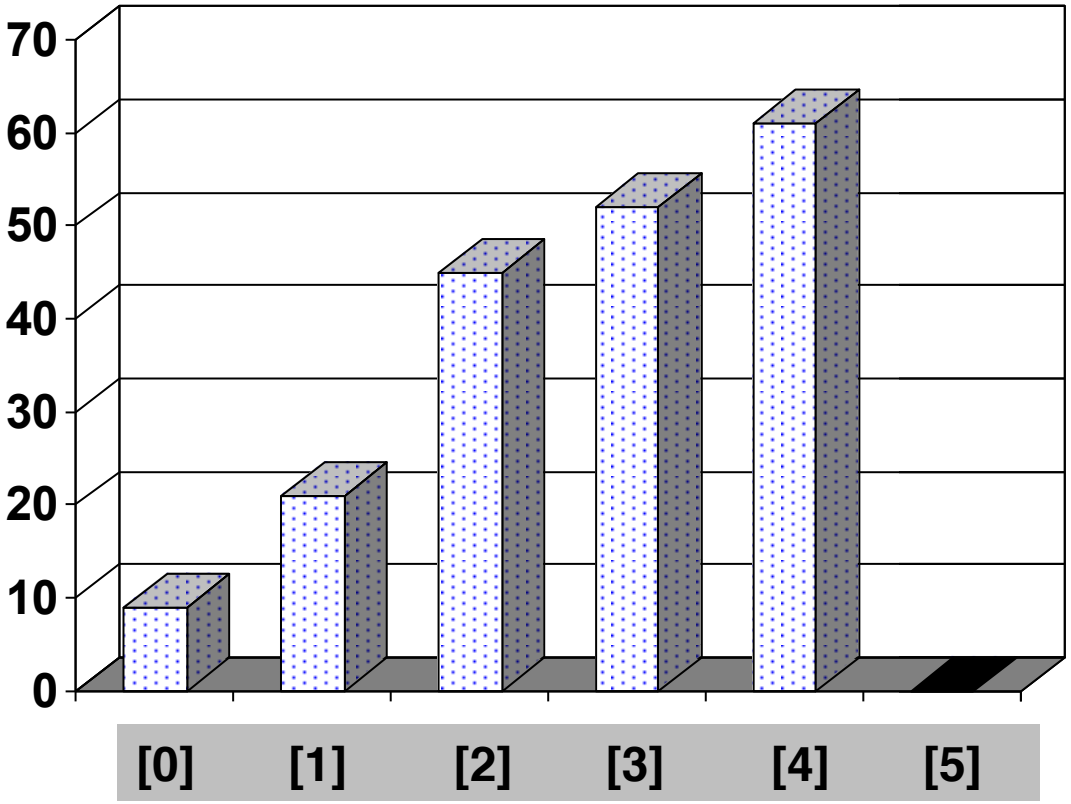
- The last element must also be inserted. Start by copying it...



Insertion Sort

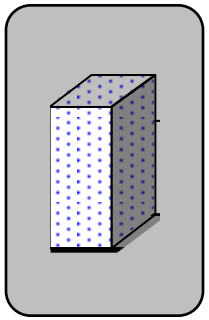
How many shifts will occur before we copy this element back into the array?

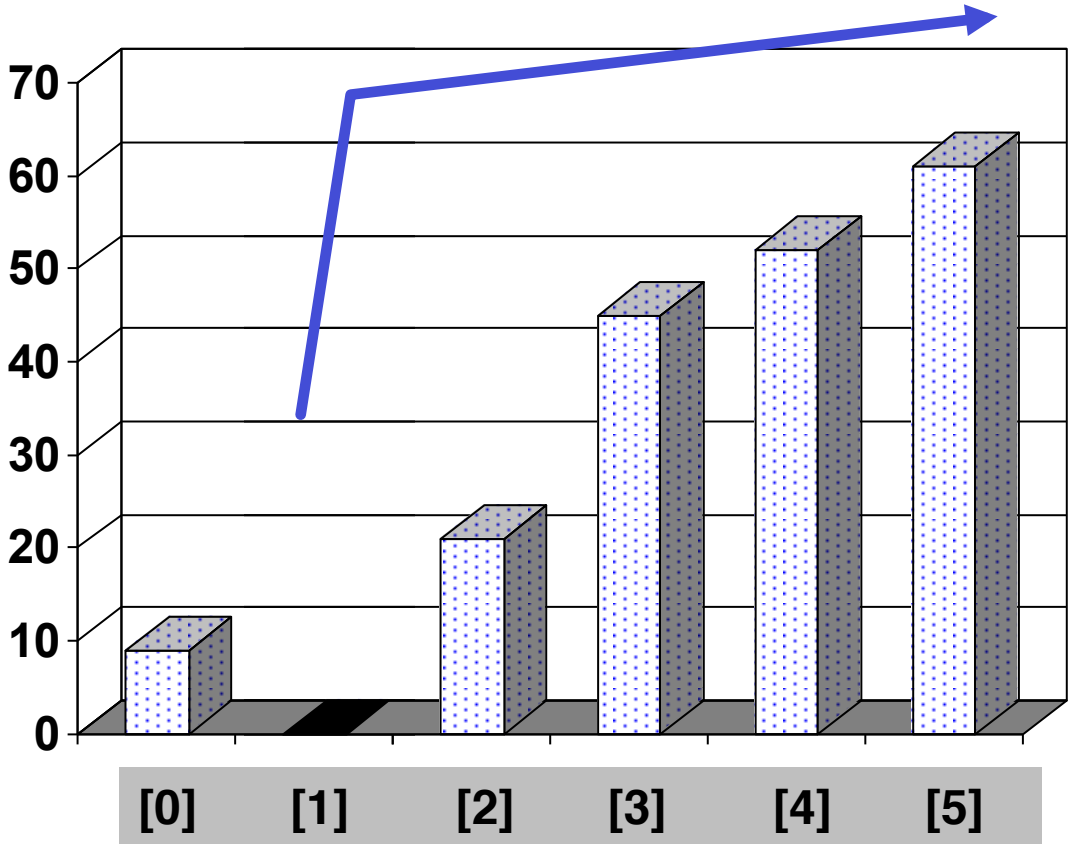




Insertion Sort

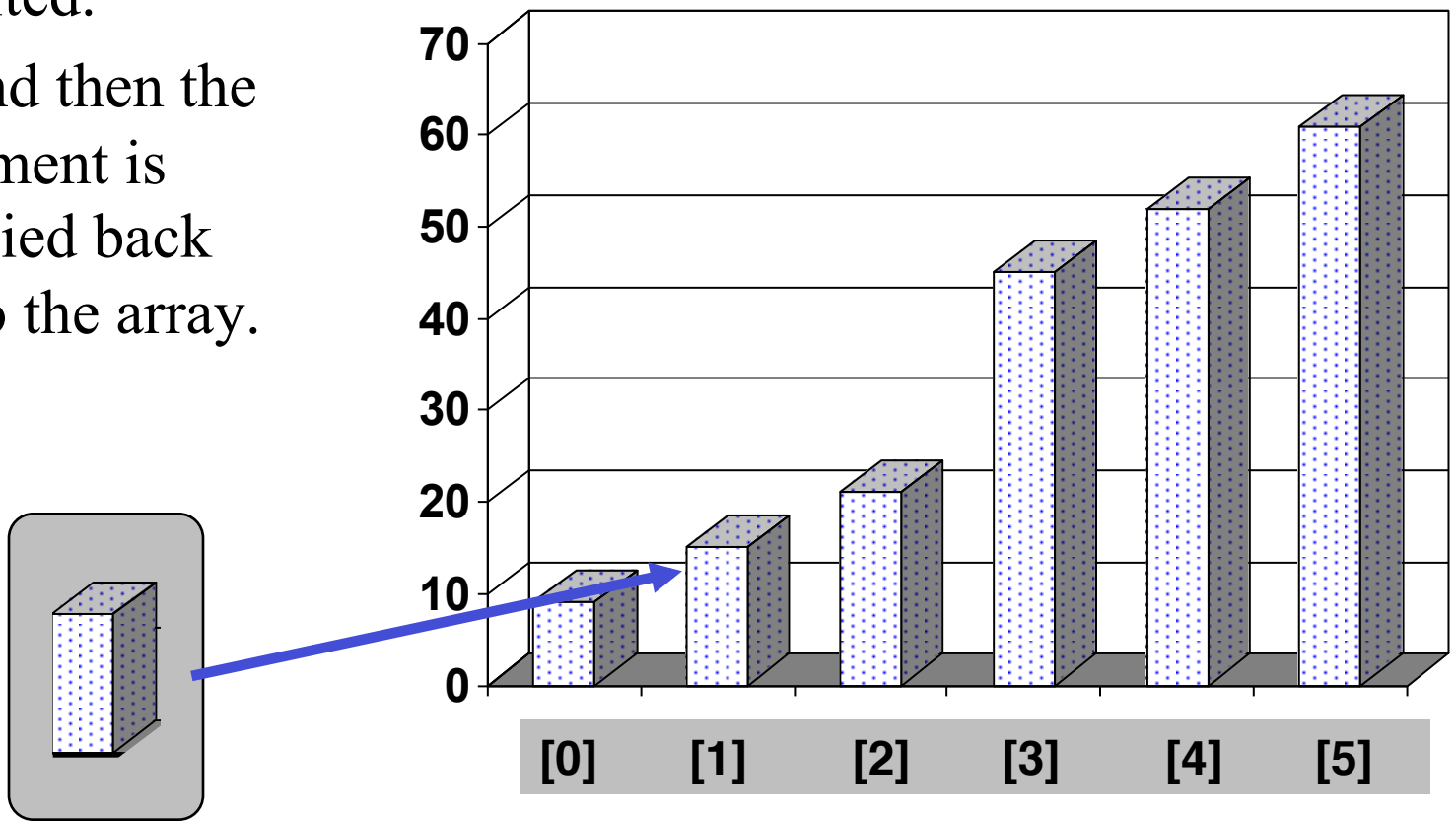
• Four items are shifted.





Insertion Sort

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



Insertion Sort

- The code (Iterative Version):

```
template <typename Item>
void insertionsort( Item a[], int left, int right)
{
        int i, j, index;
        for (i=1; i < array_size; i++)
        {
                index = numbers[i]; j = i;
                while ((j > 0) && (numbers[j-1] > index))
                {
                        numbers[j] = numbers[j-1]; j = j - 1;
                }
                numbers[j] = index; }
        }
}
```

Insertion Sort

Here is a more efficient version (nearly twice as fast, but harder to understand)

```
template <typename Item>
void insertionsort( Item a[], int left, int right)
{
        for ( int i = right; i > left; i-- )
        {
                if ( a[i-1] < a[i])
                {
                        swap( a[i-1], a[i]);
                }
        }
        for ( int i = left + 2; i <= right; i++ )
        {
                int j = i; Item v = a[i];
                while( v < a[j-1])
                {
                        a[j] = a[j-1];
                        j--;
                }
                a[j] = v;
        }
}
```

Sorting Algorithms

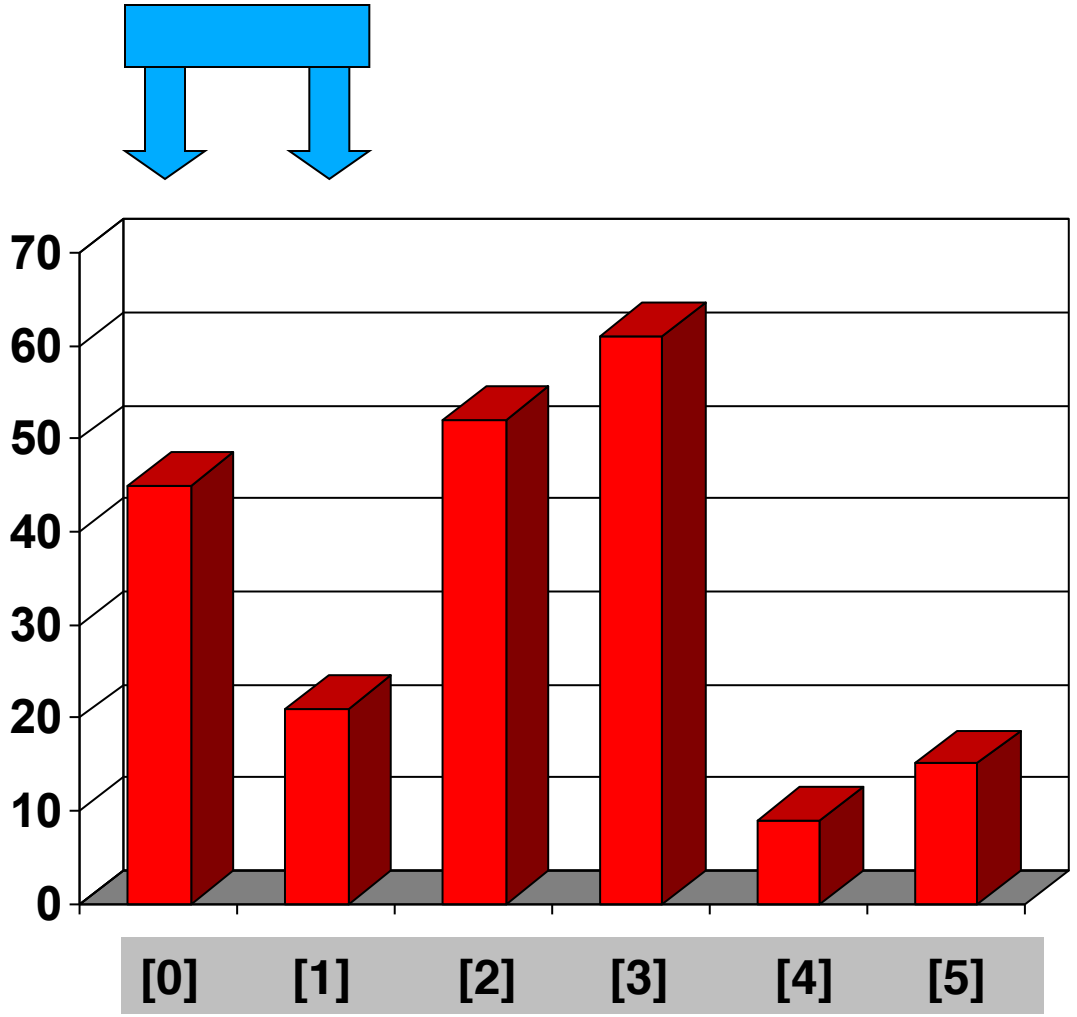
- Bubble Sort (Iterative Version)
- Outer Loop

  Performs the inner loop n times, where n is the size of the array. This guarantees that even an element which is at completely the wrong end of the array will end up in the right place.
- Inner Loop

  Compares adjacent elements and swaps them if they are out of order.

Bubble Sort

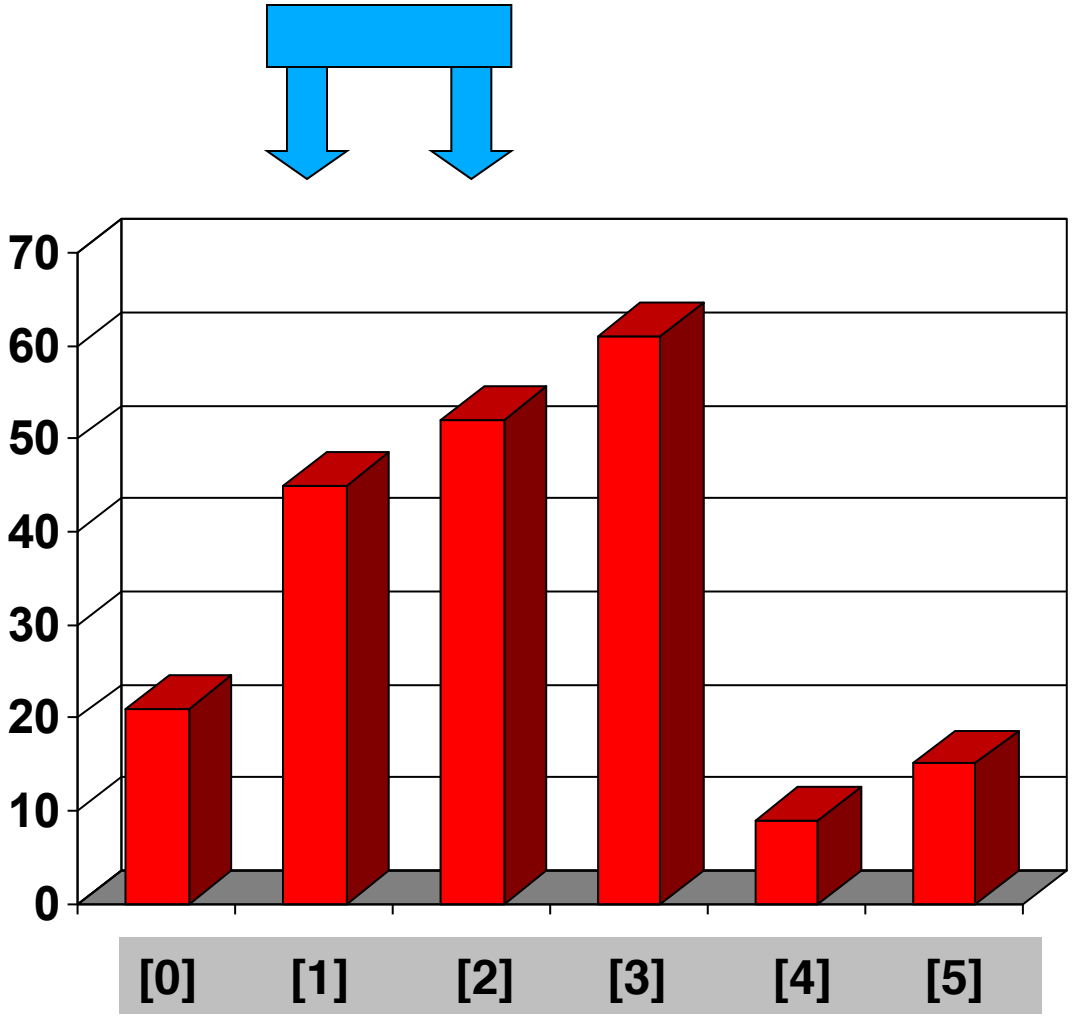
- Bubble Sort does pair-wise comparisons and swaps



Bubble Sort

- 0 and 1 swapped

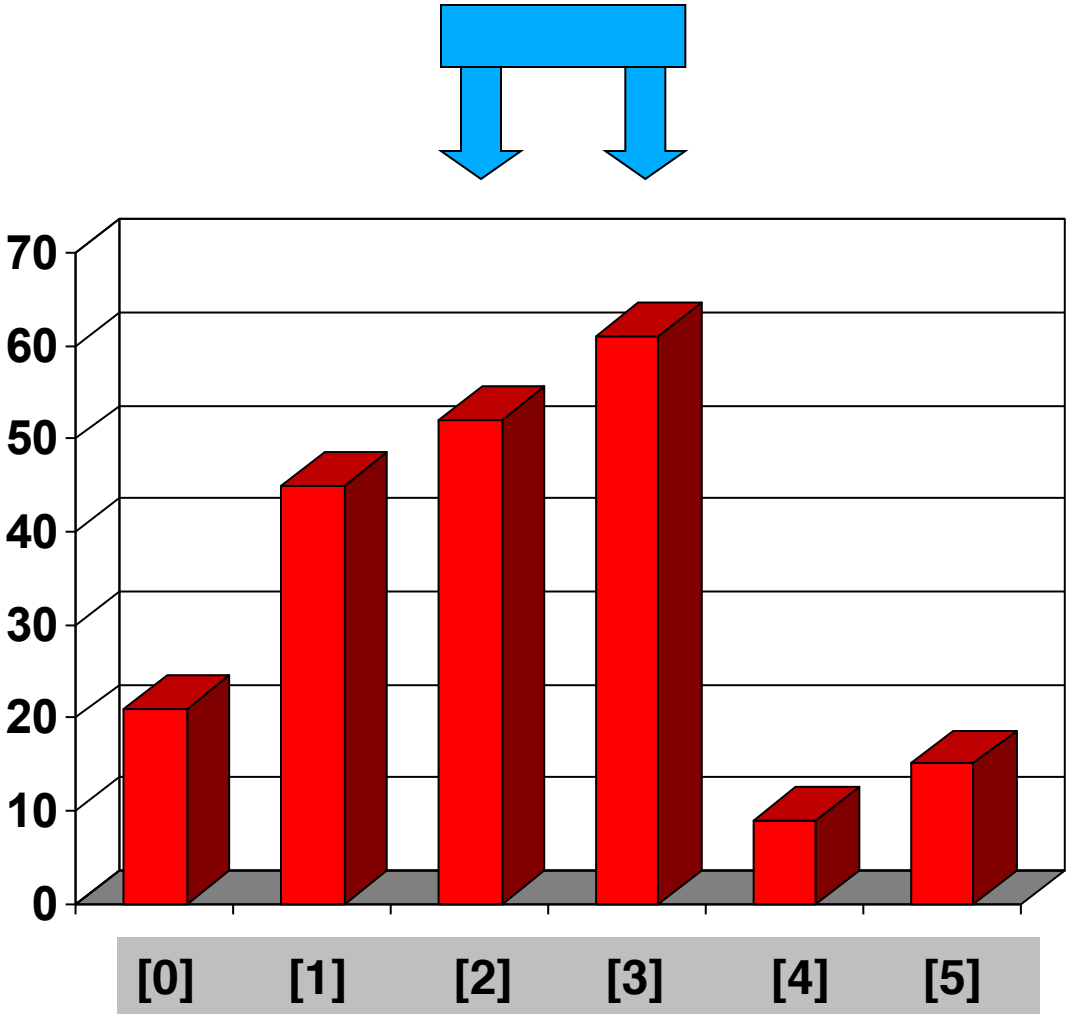
Pass 1 of 6



Bubble Sort

- 1 and 2 not swapped

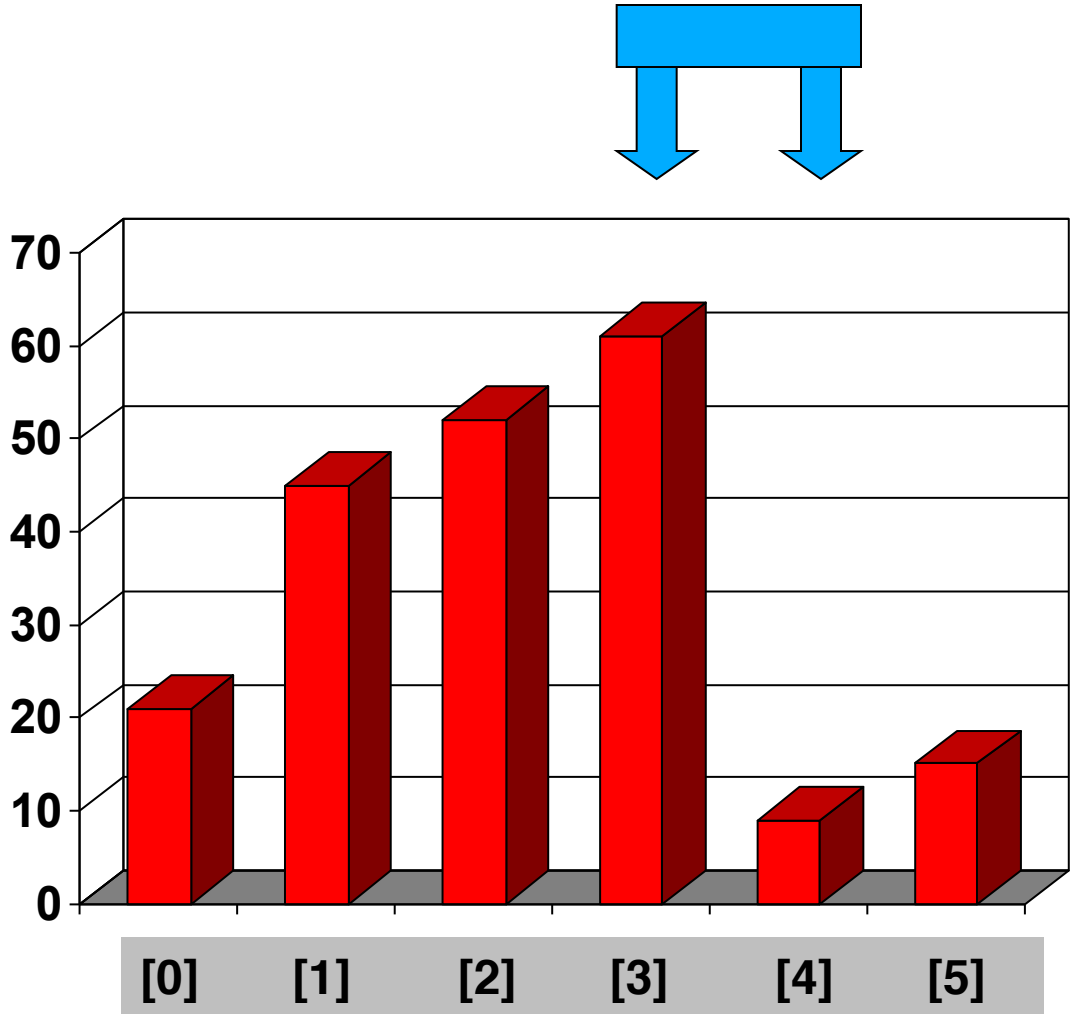
Pass 1 of 6



Bubble Sort

- 2 and 3 not swapped

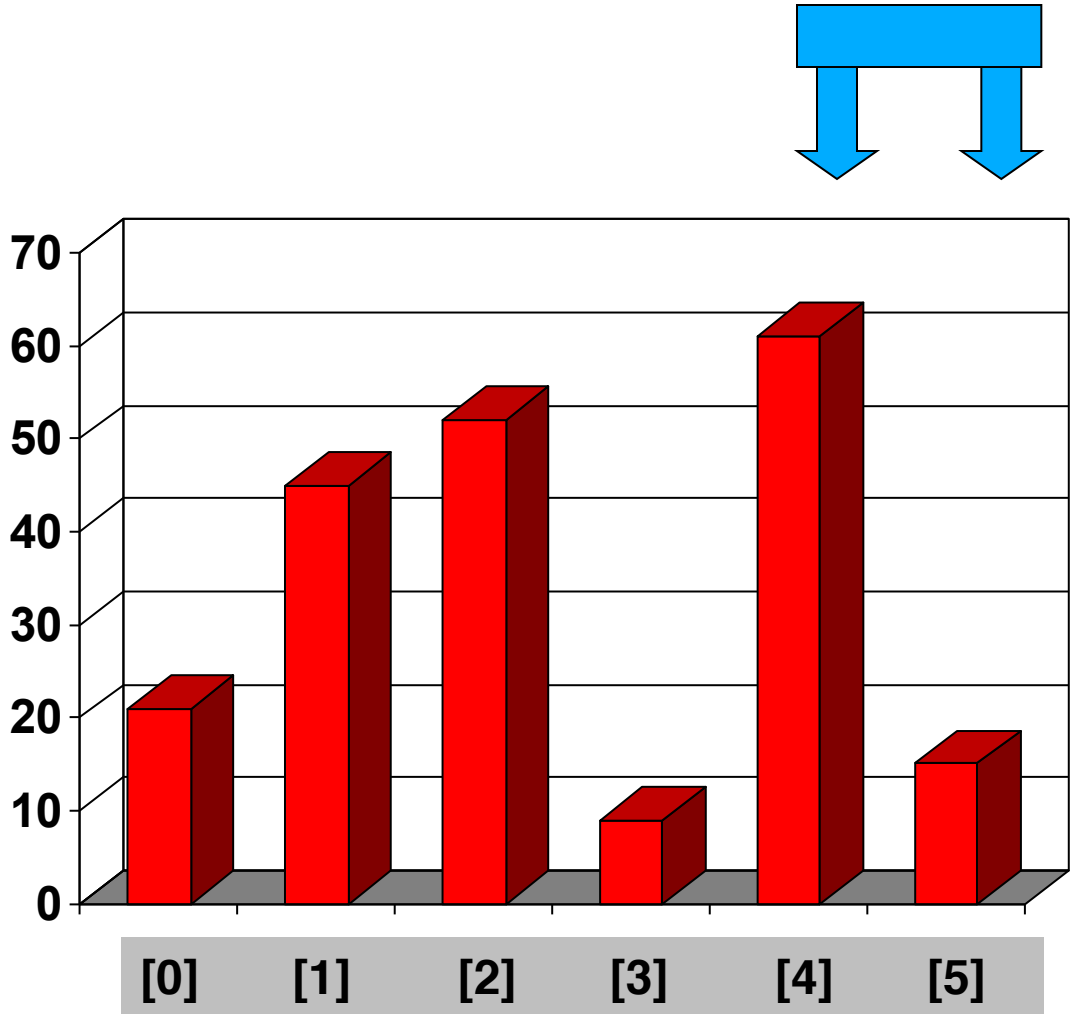
Pass 1 of 6



Bubble Sort

- 3 and 4 swapped

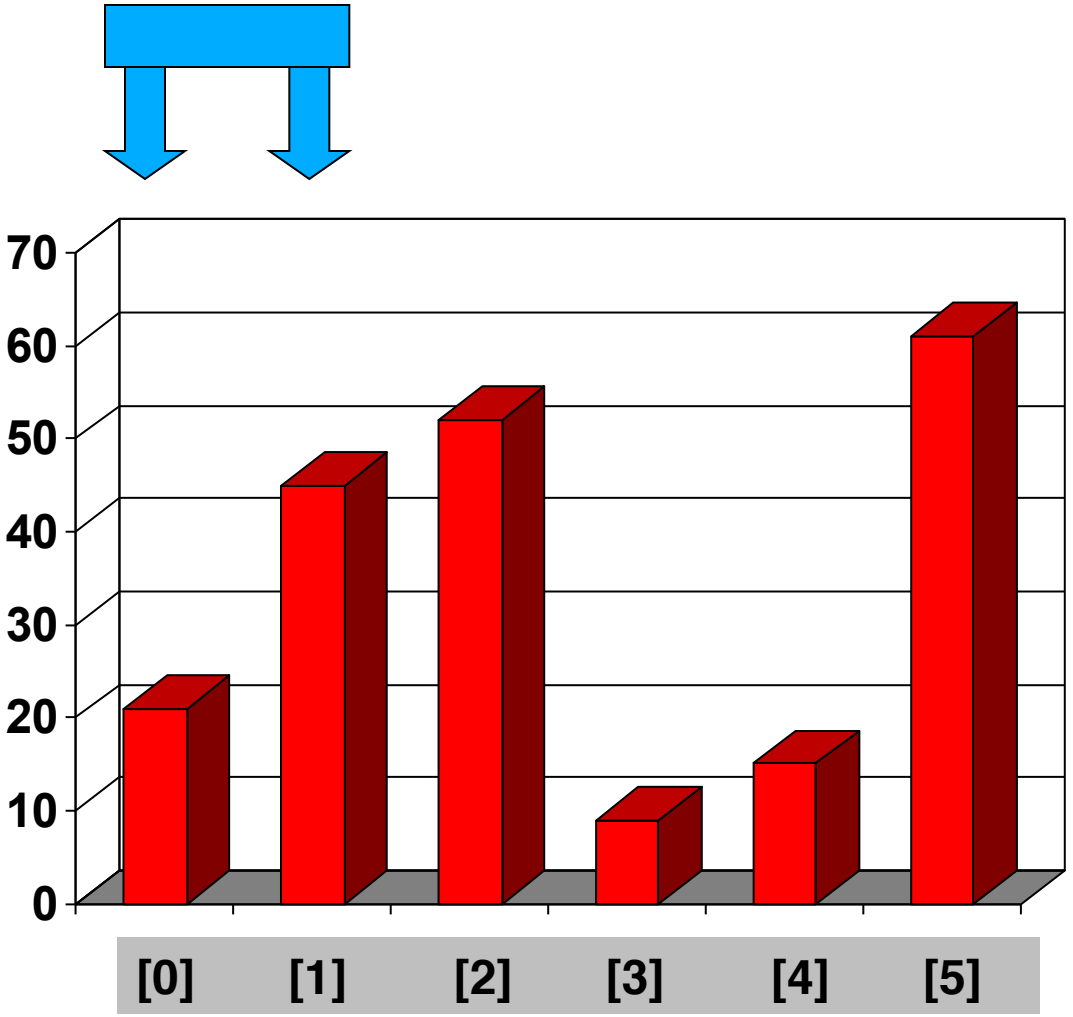
Pass 1 of 6



Bubble Sort

- 4 and 5 swapped

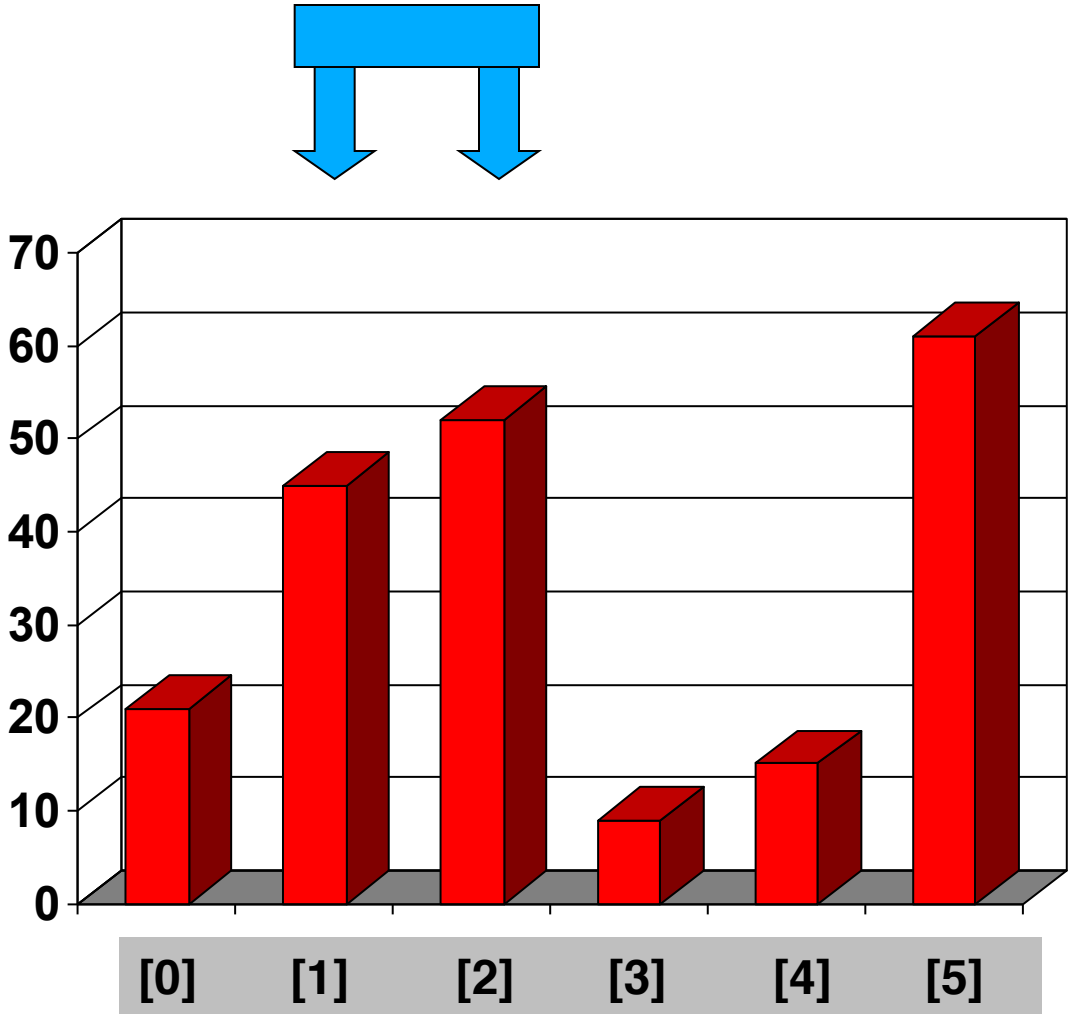
Pass 1 of 6



Bubble Sort

- 0 and 1 not swapped

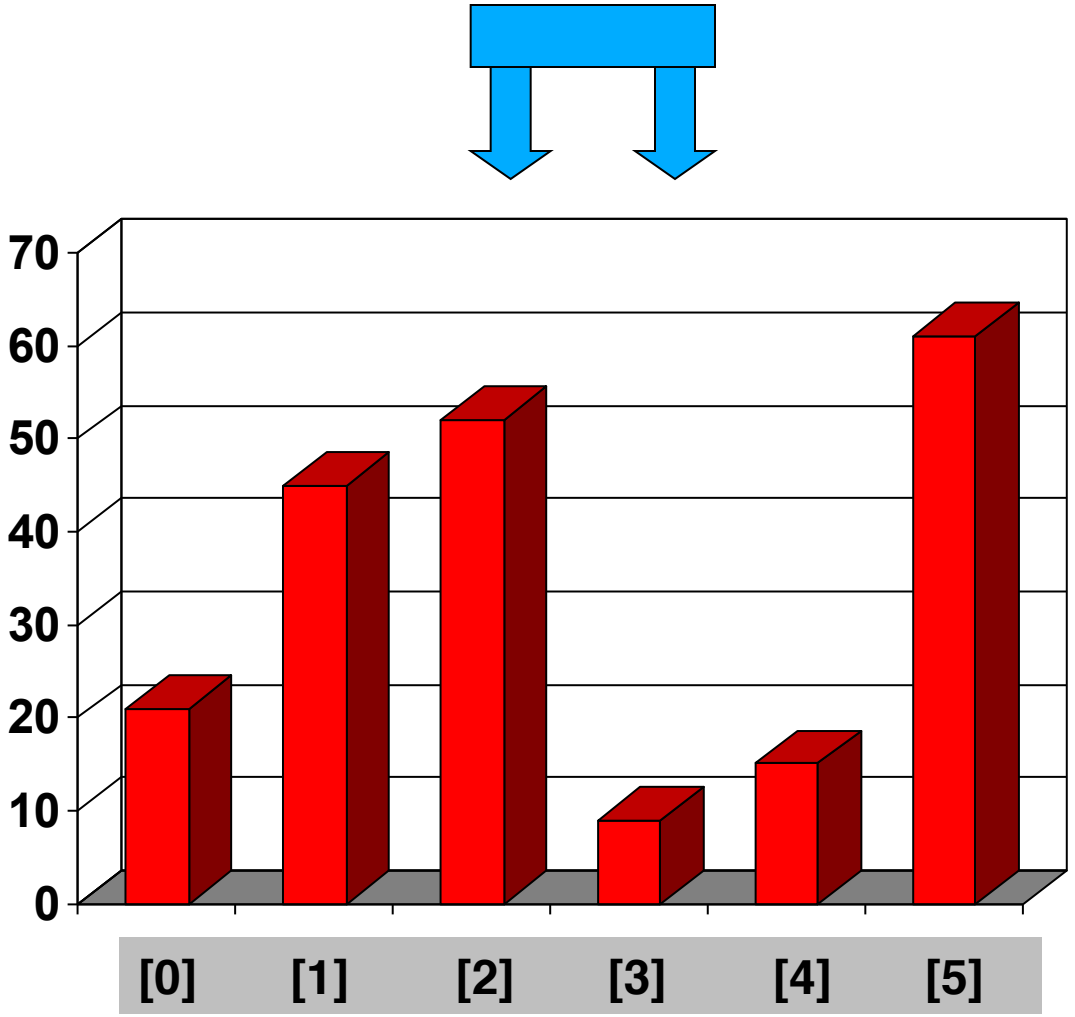
Pass 2 of 6



Bubble Sort

- 1 and 2 not swapped

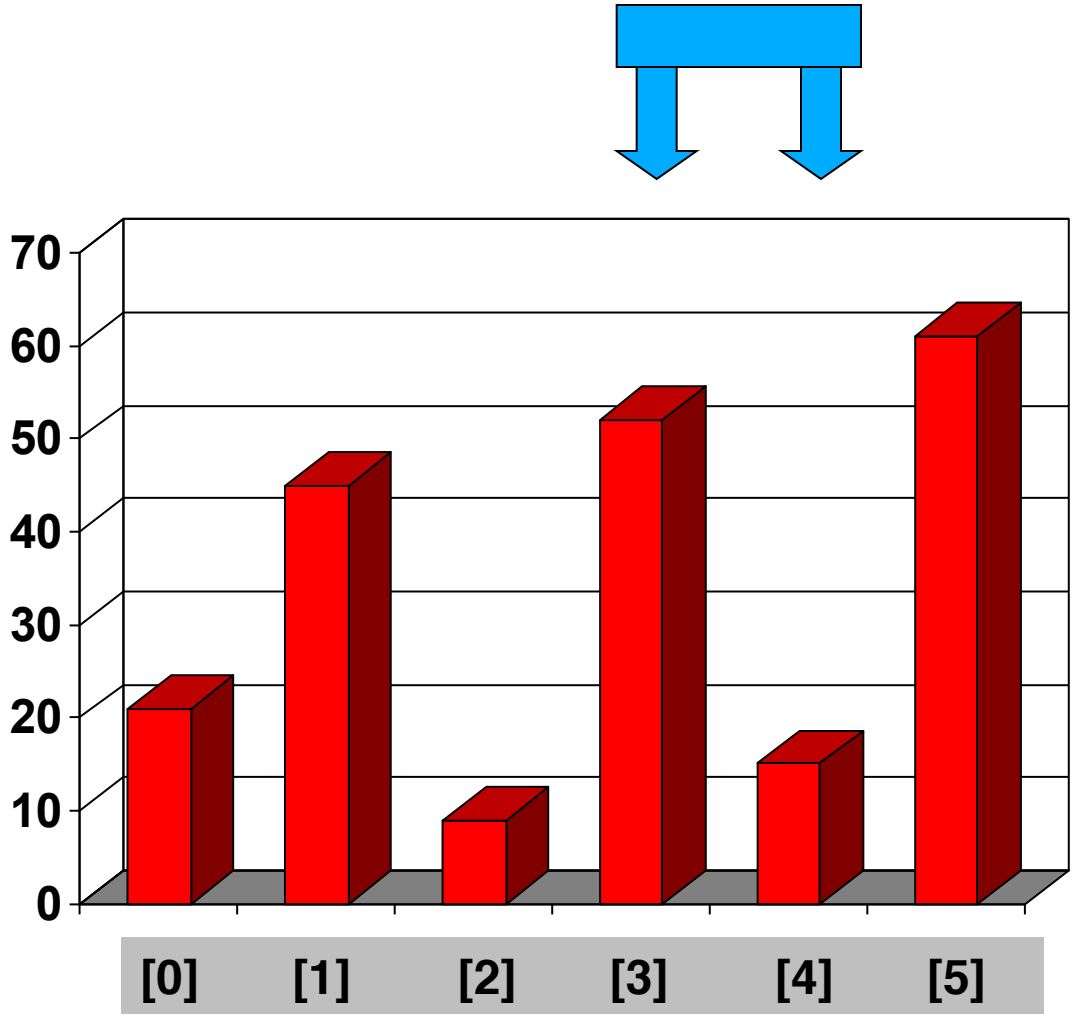
Pass 2 of 6



Bubble Sort

- 2 and 3 swapped

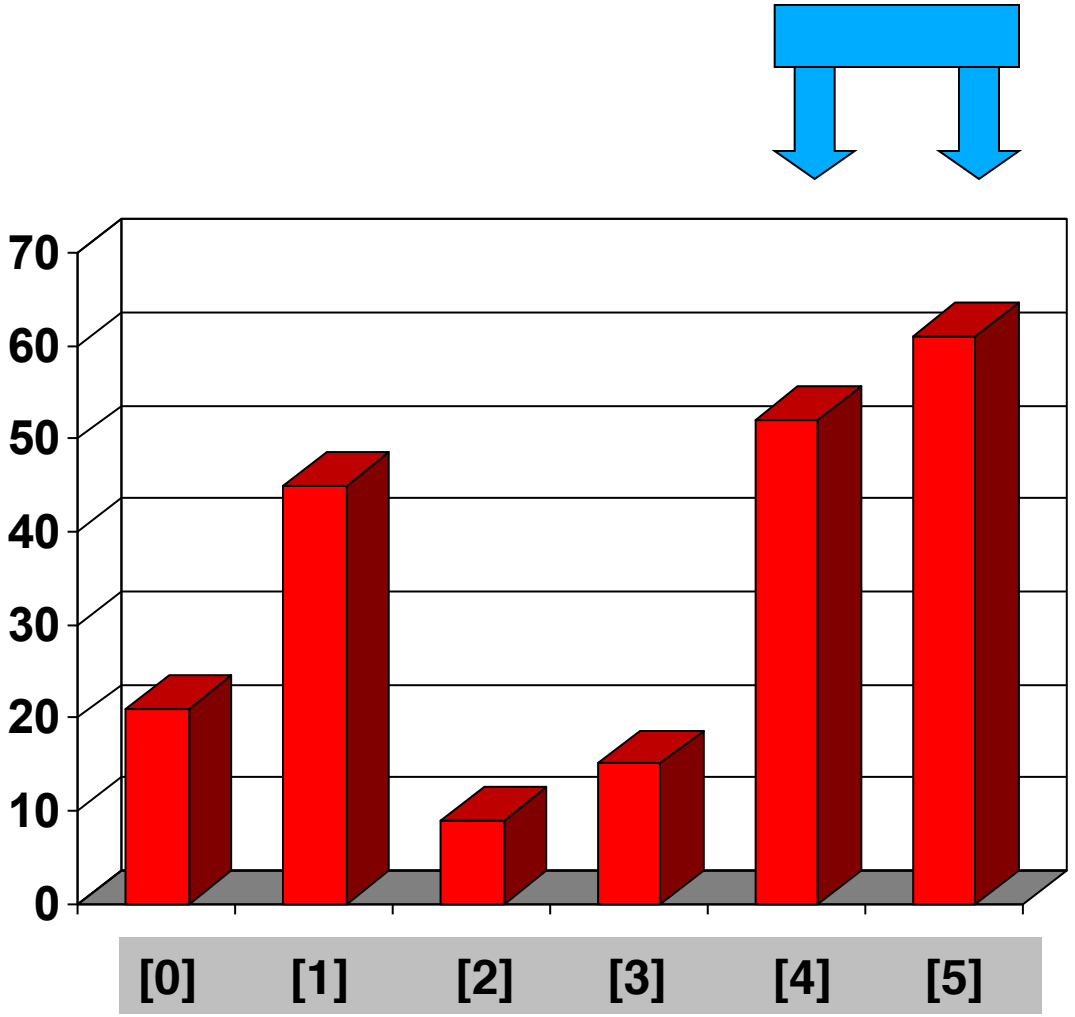
Pass 2 of 6



Bubble Sort

- 3 and 4 swapped

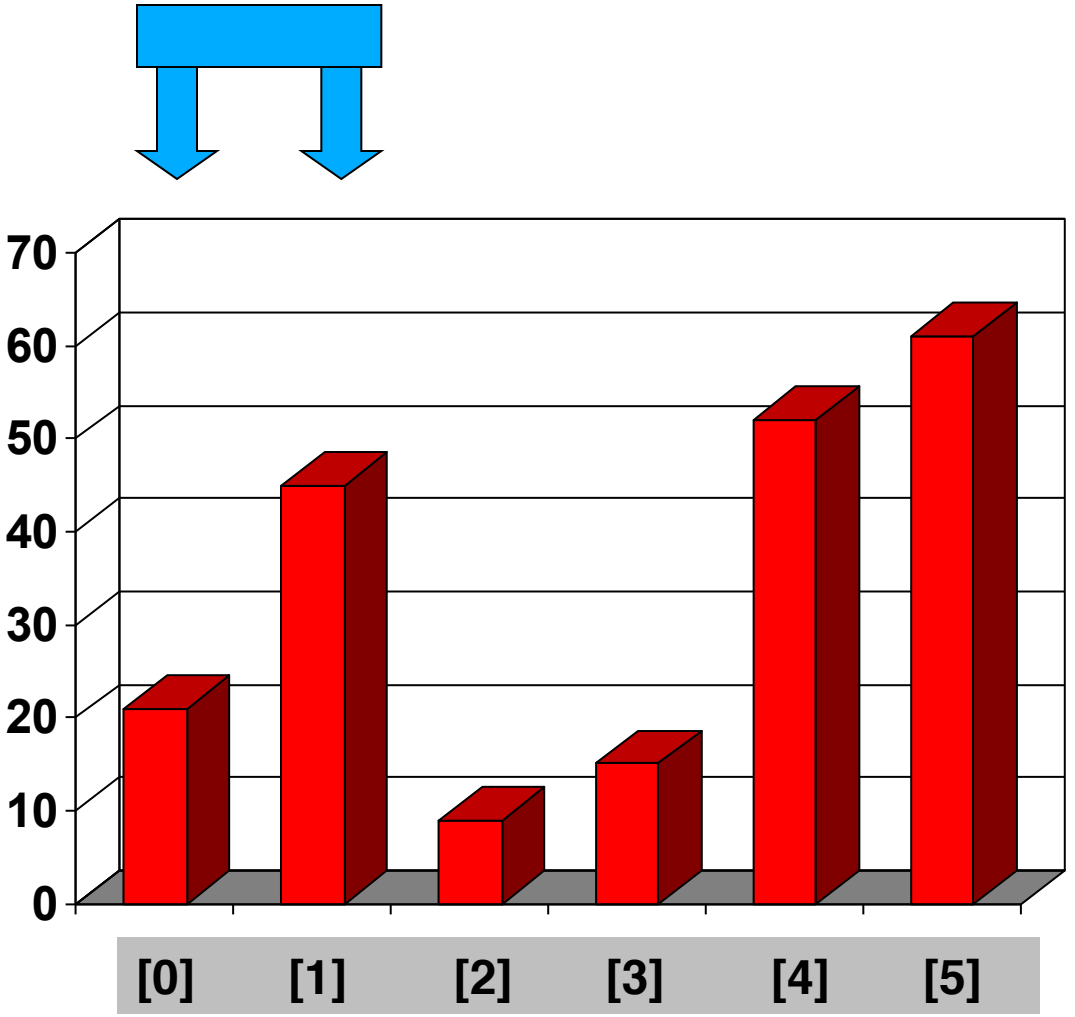
Pass 2 of 6



Bubble Sort

- 4 and 5 not swapped

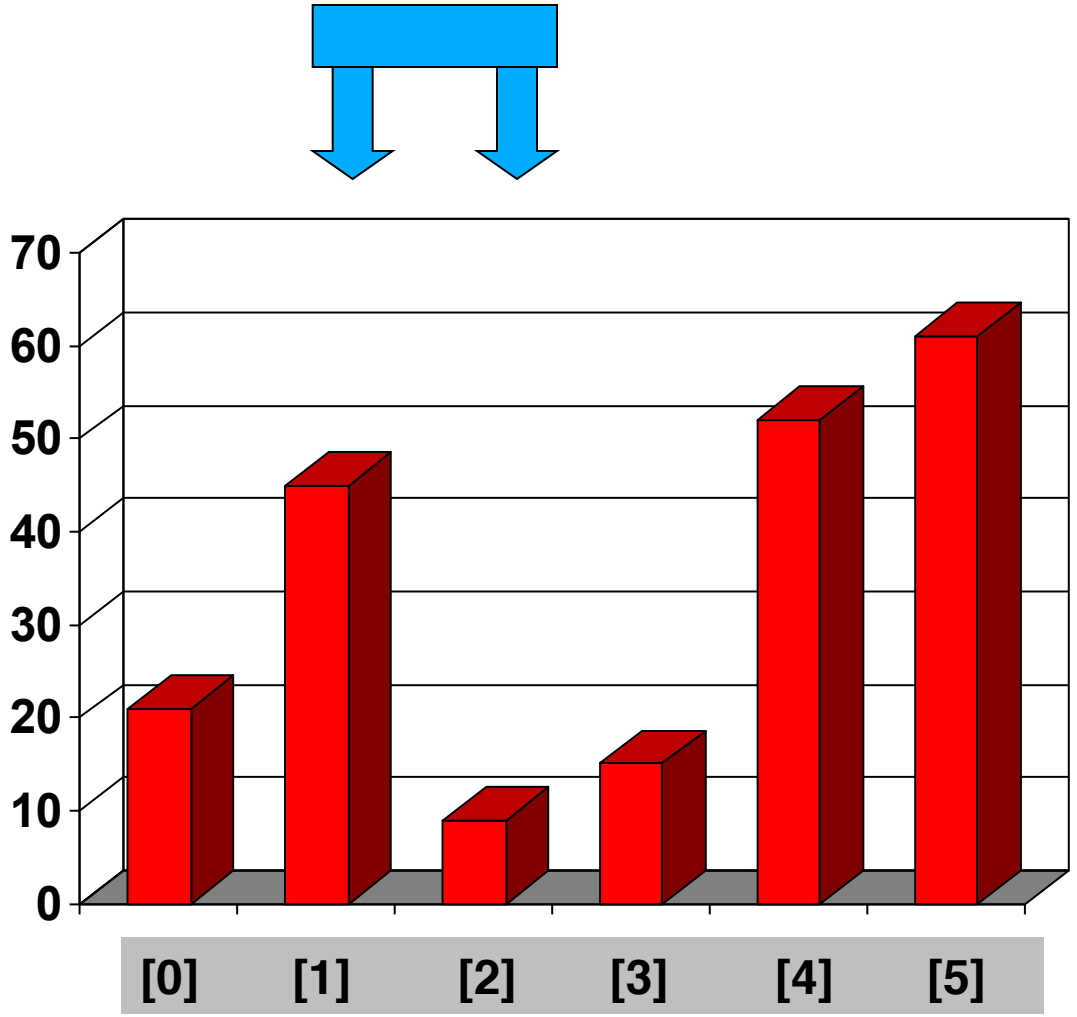
Pass 2 of 6



Bubble Sort

- 0 and 1 not swapped

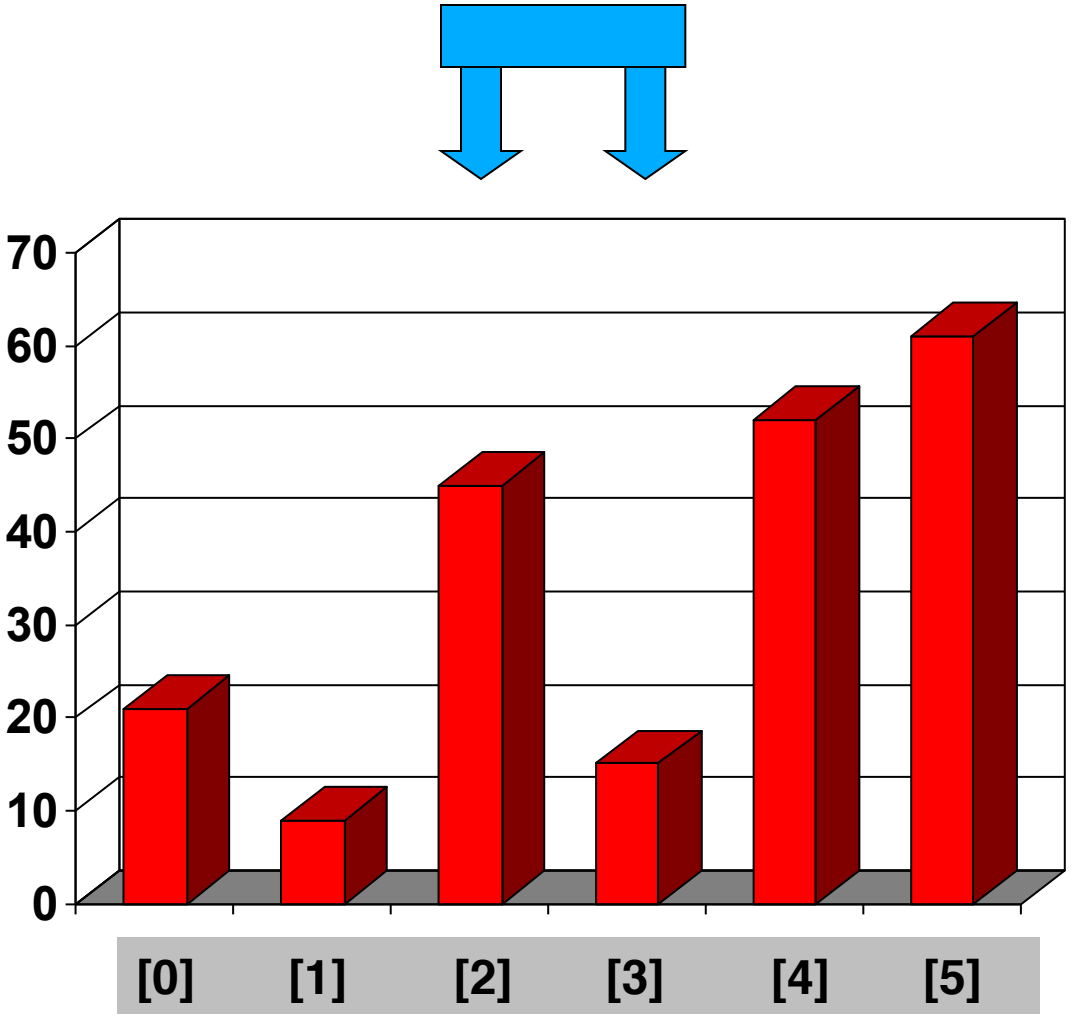
Pass 3 of 6



Bubble Sort

- 1 and 2 swapped

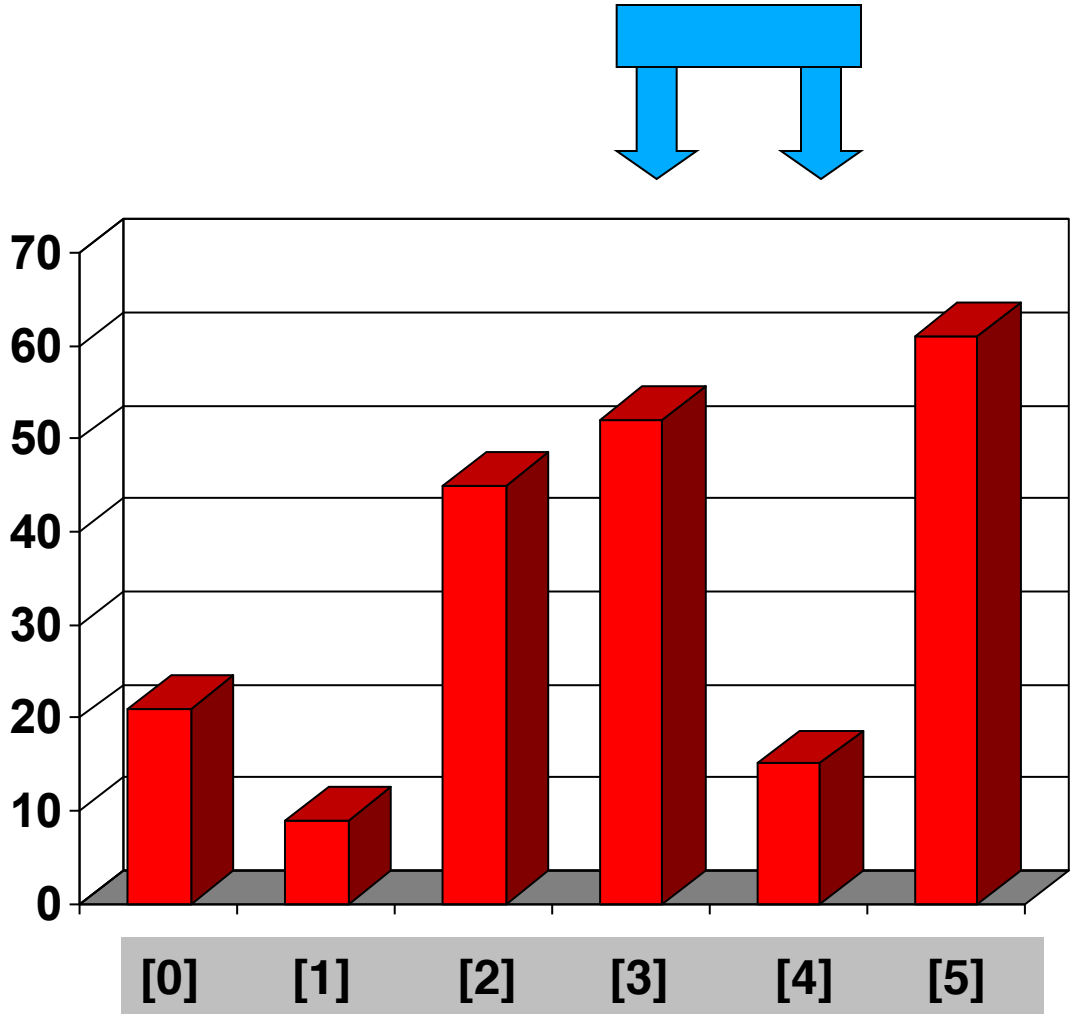
Pass 3 of 6



Bubble Sort

- 2 and 3 swapped

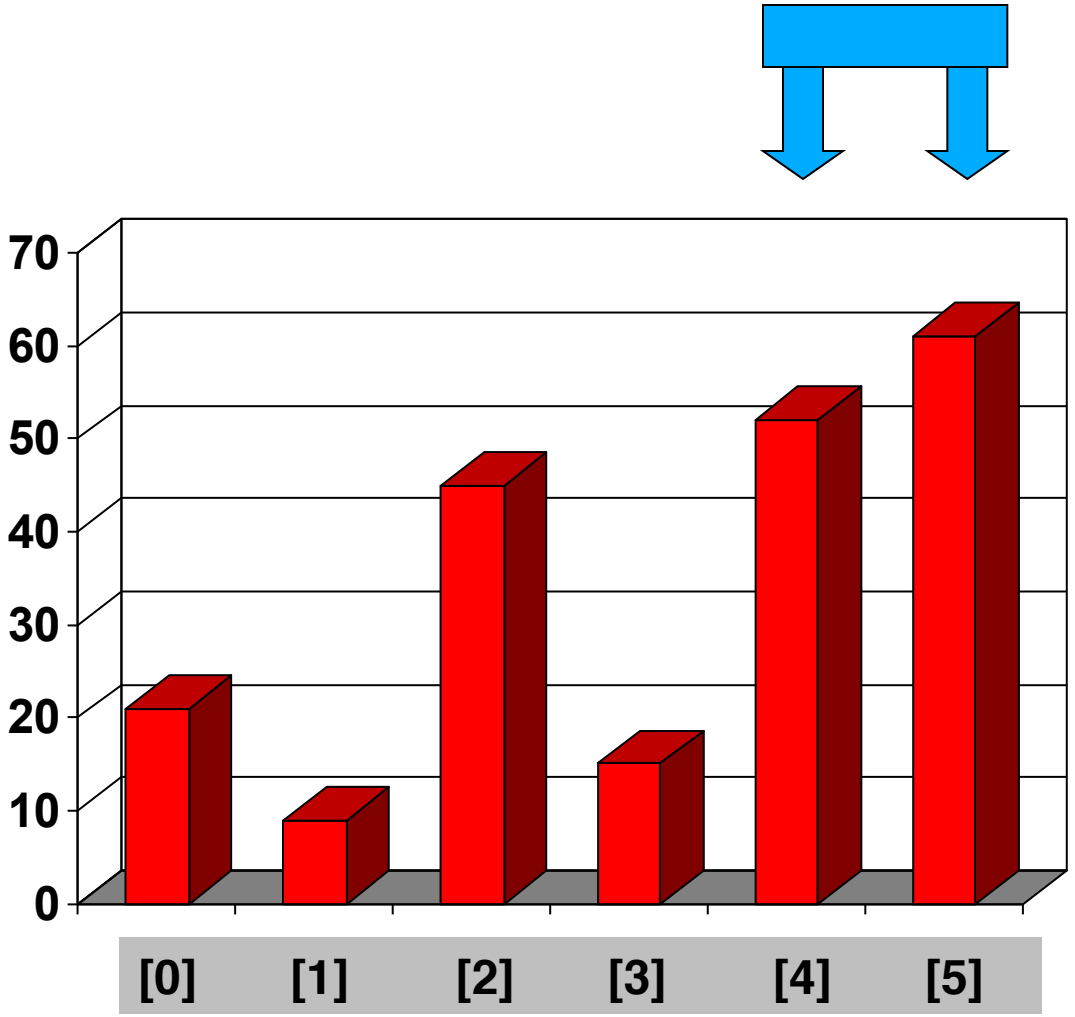
Pass 3 of 6



Bubble Sort

- 3 and 4 swapped

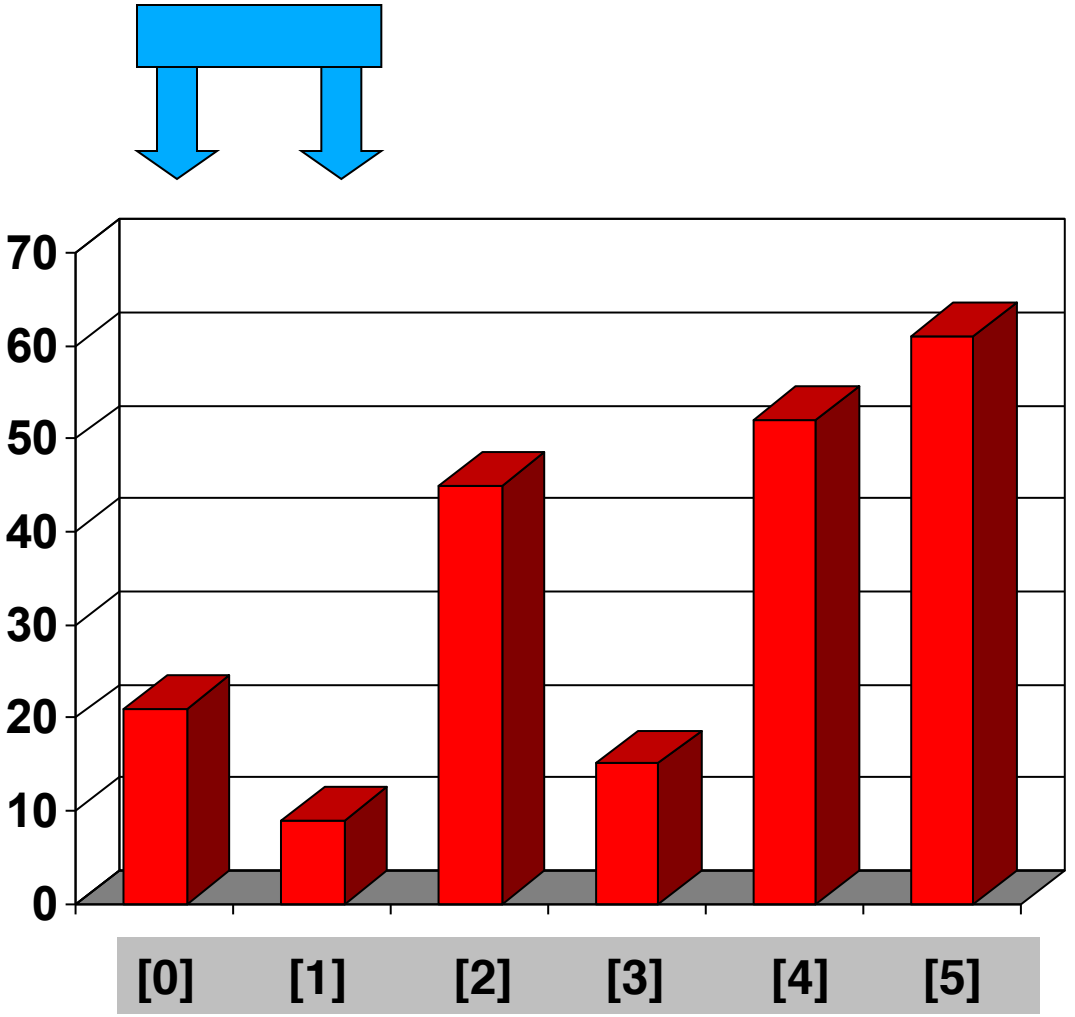
Pass 3 of 6



Bubble Sort

- 4 and 5 not swapped

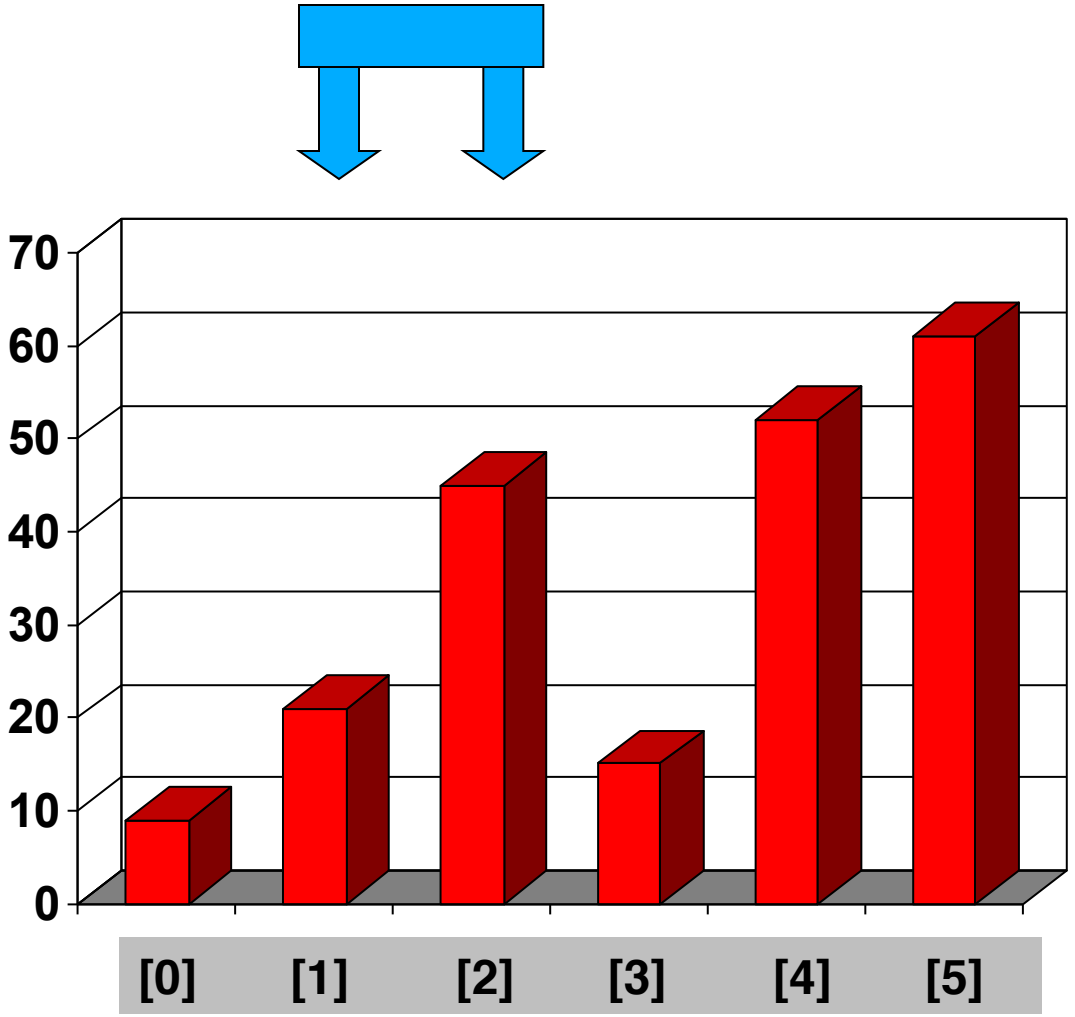
Pass 3 of 6



Bubble Sort

- 0 and 1 swapped

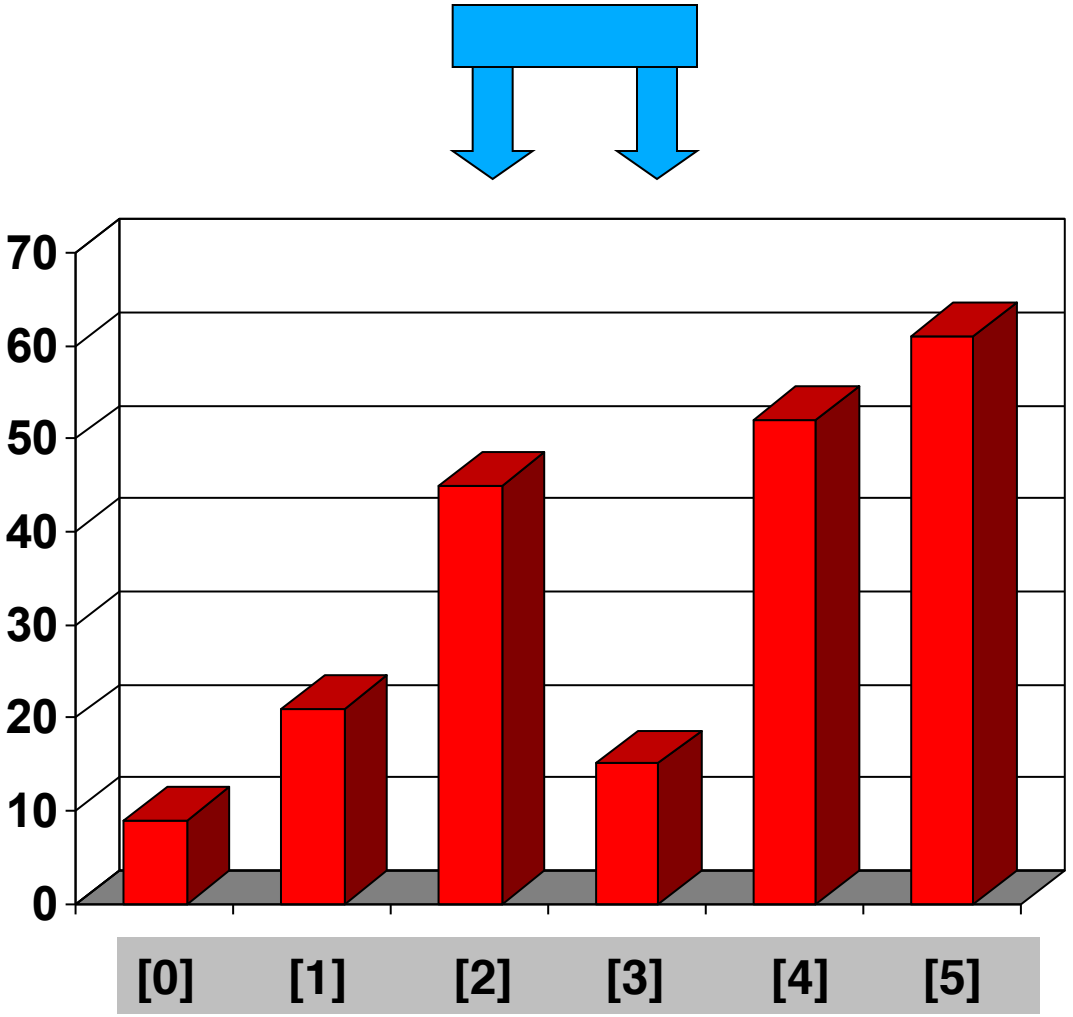
Pass 4 of 6



Bubble Sort

- 1 and 2 not swapped

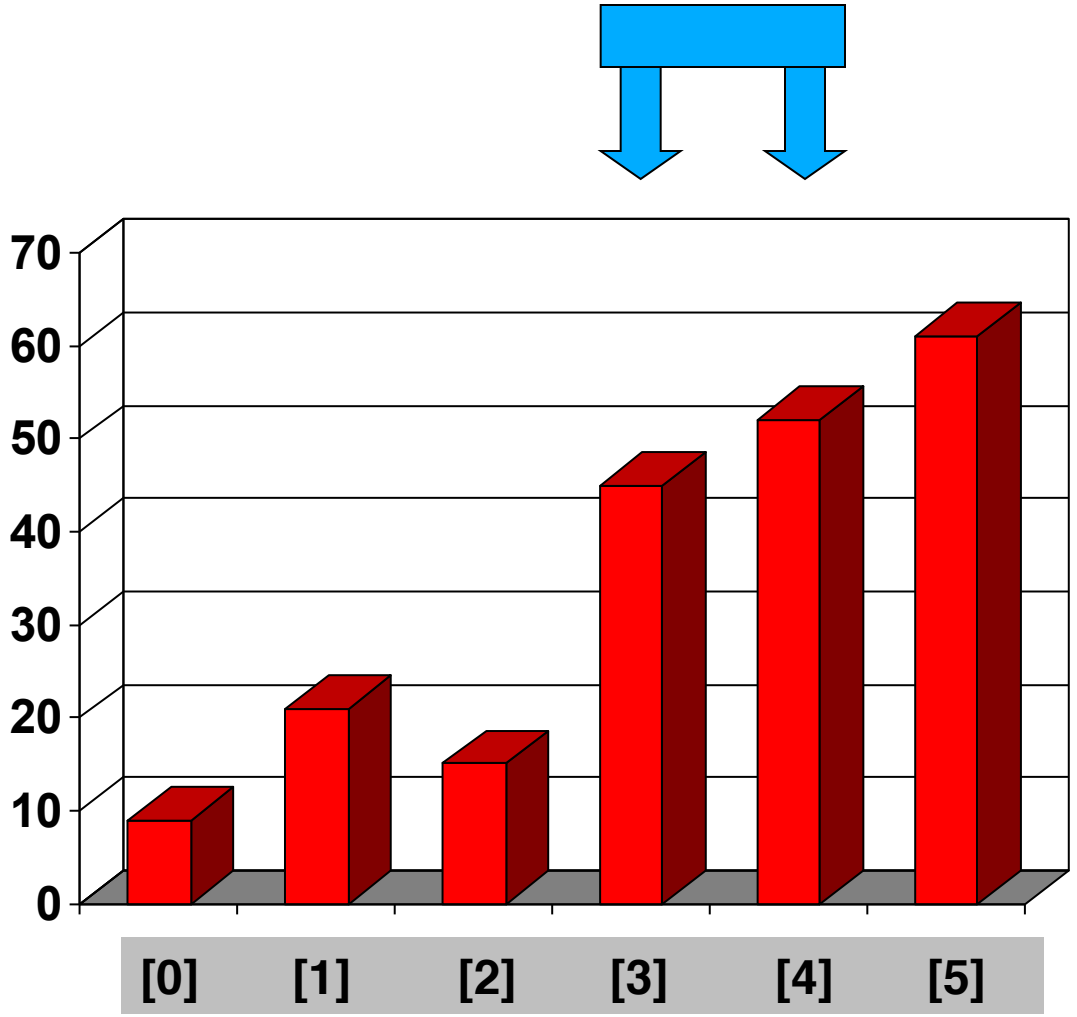
Pass 4 of 6



Bubble Sort

- 2 and 3 swapped

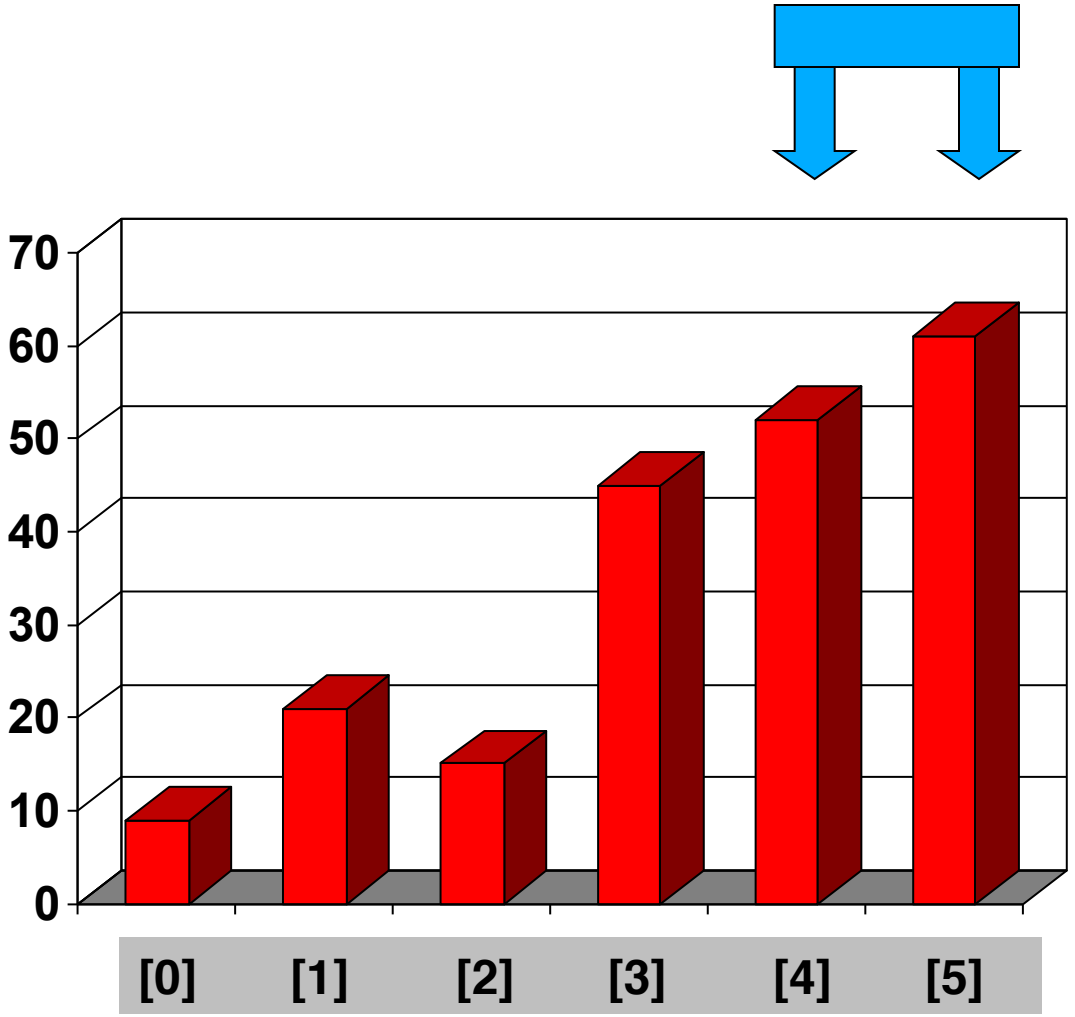
Pass 4 of 6



Bubble Sort

- 3 and 4 not swapped

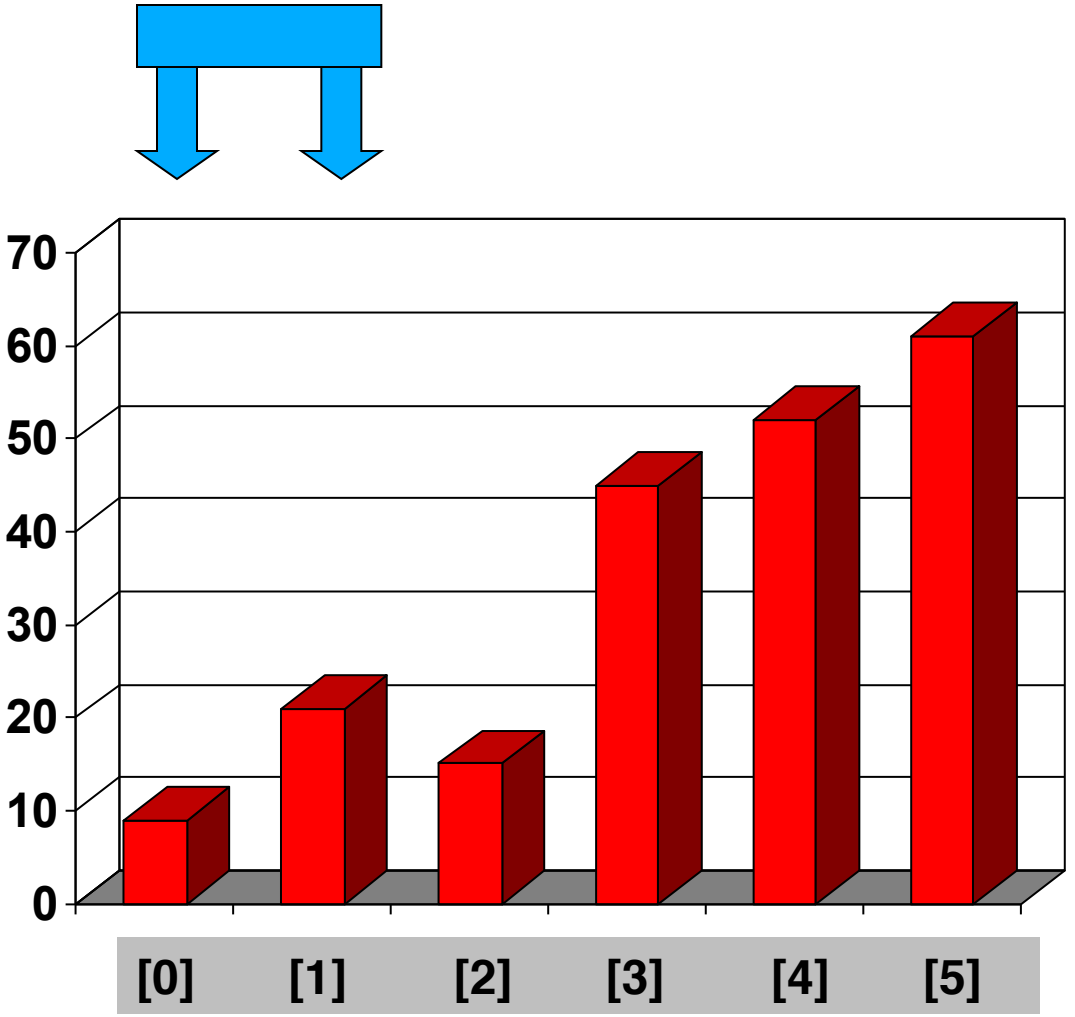
Pass 4 of 6



Bubble Sort

- 4 and 5 not swapped

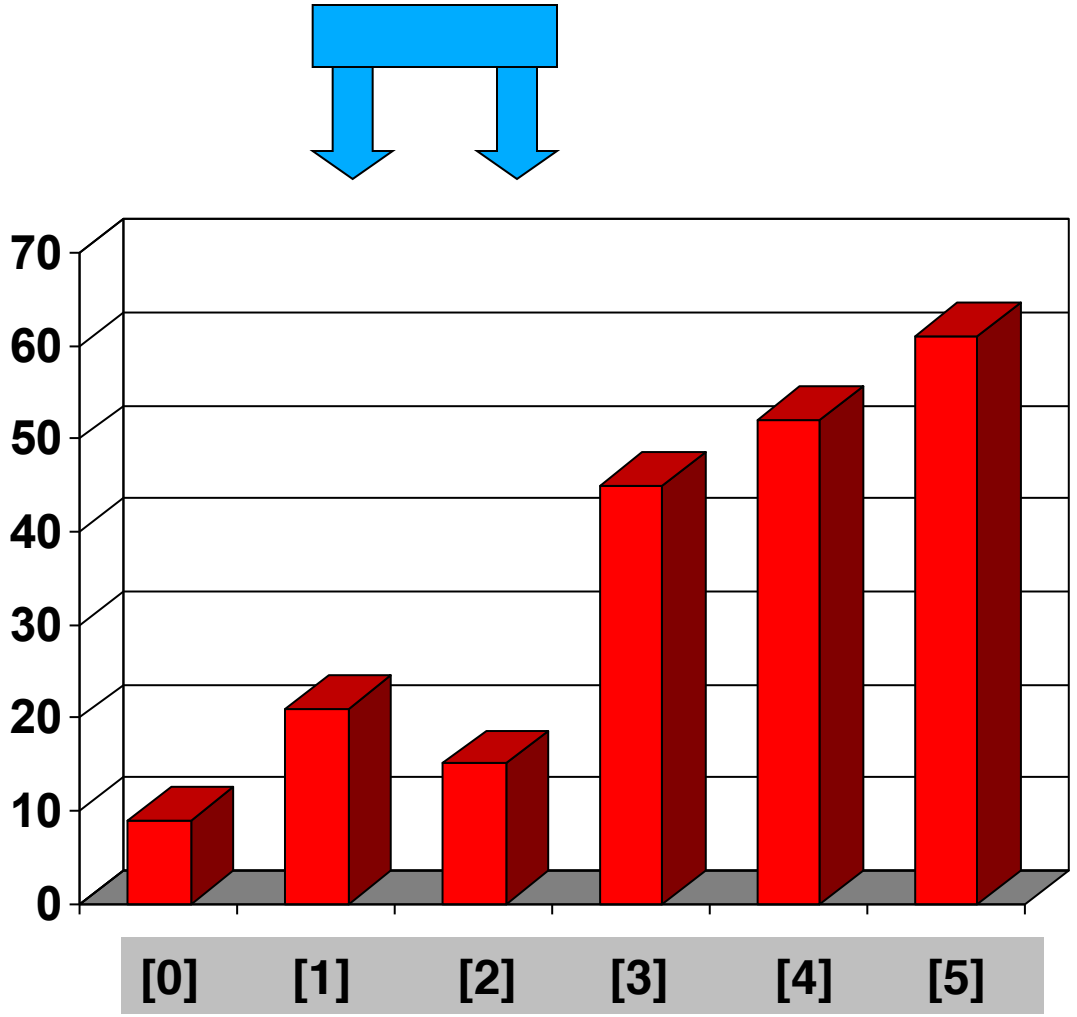
Pass 4 of 6



Bubble Sort

- 0 and 1 not swapped

Pass 5 of 6

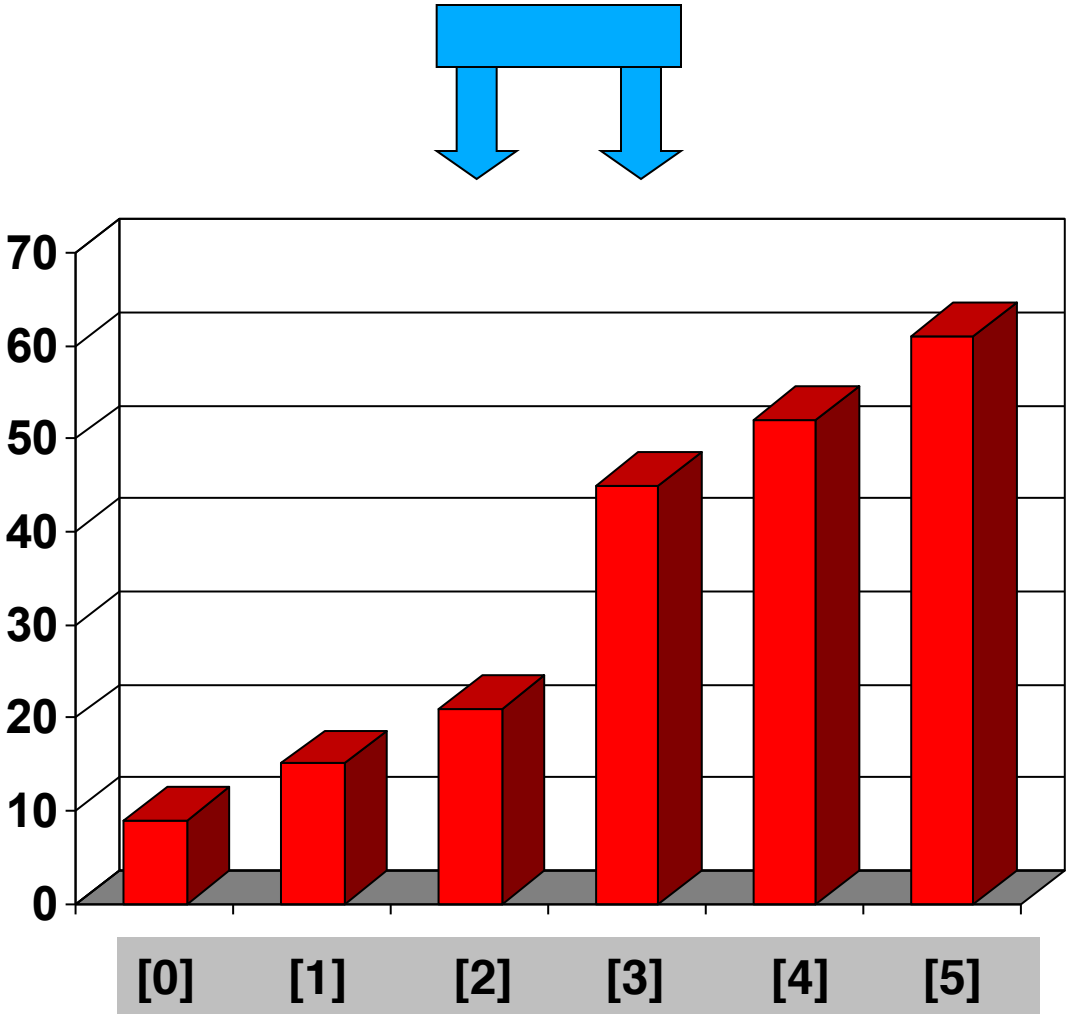


Bubble Sort

- 1 and 2 swapped

Pass 5 of 6

The array is now sorted but there is no way for the program to know that so it continues blindly on

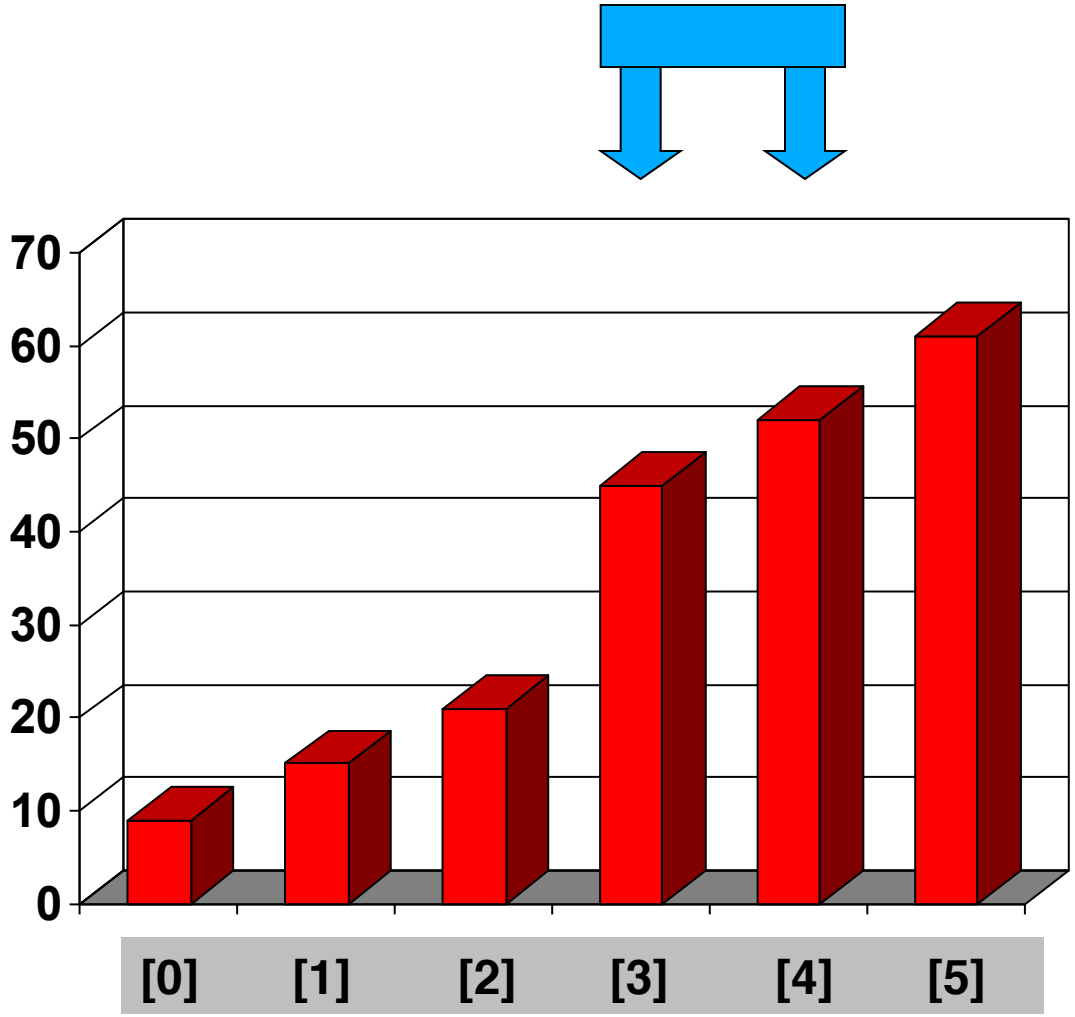


Bubble Sort

- 2 and 3 not swapped

Pass 5 of 6

The array is now sorted but there is no way for the program to know that so it continues blindly on

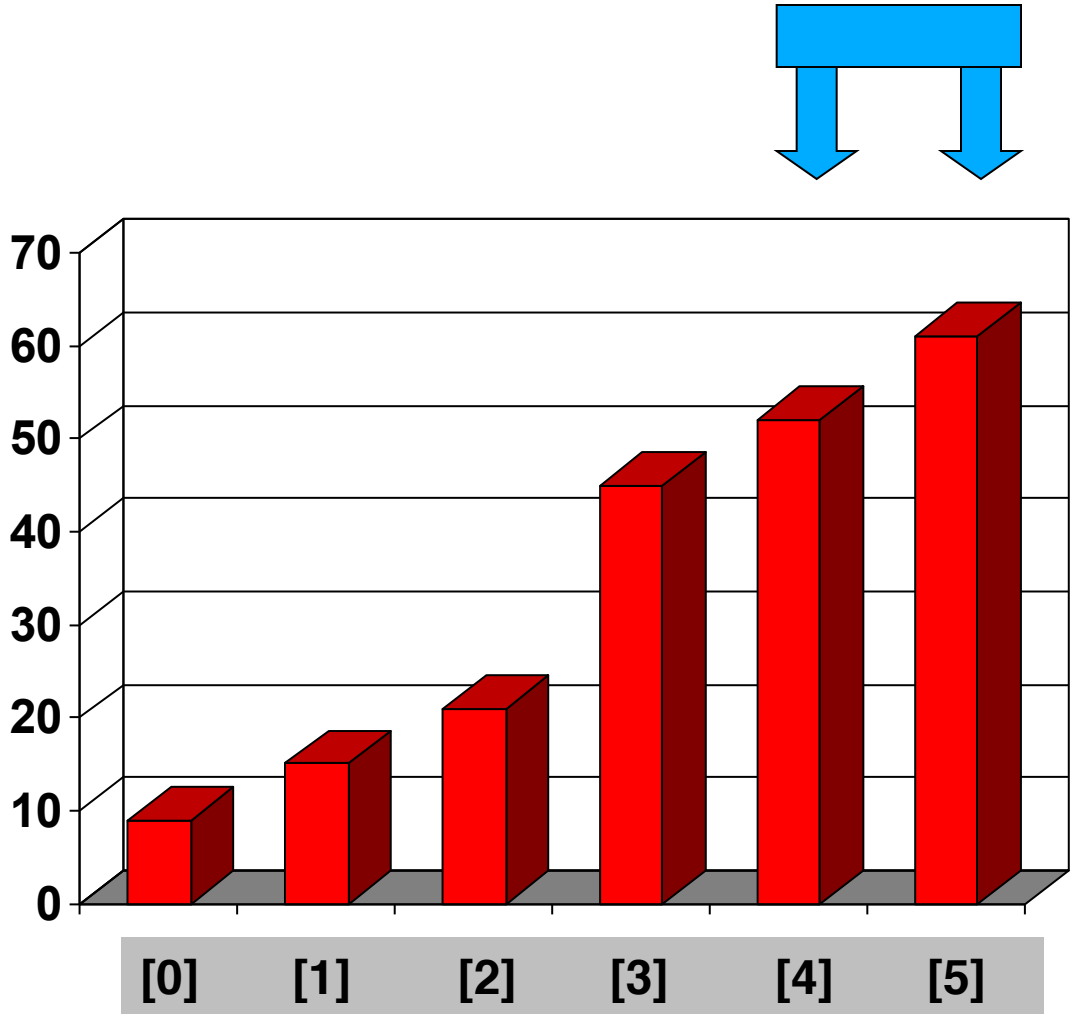


Bubble Sort

- 3 and 4 not swapped

Pass 5 of 6

The array is now sorted but there is no way for the program to know that so it continues blindly on

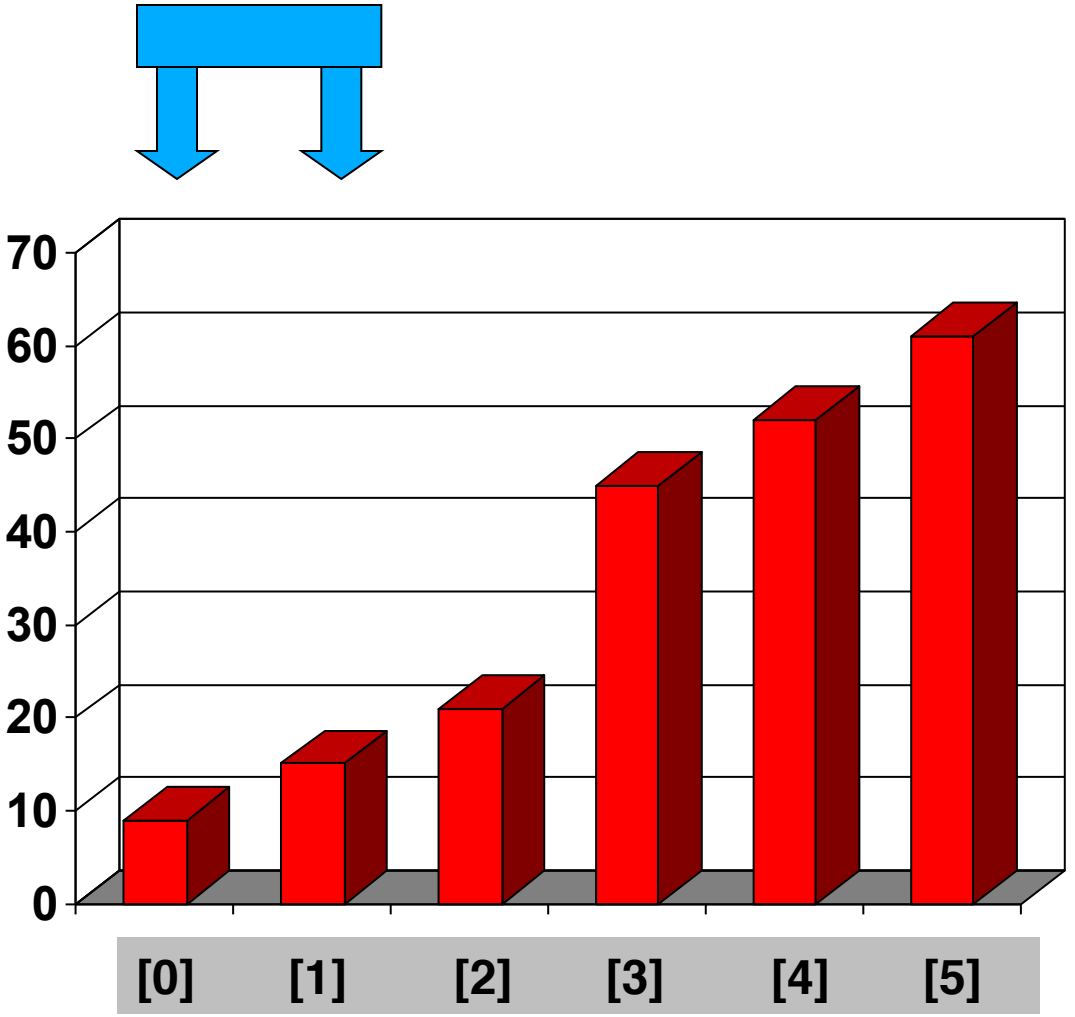


Bubble Sort

- 4 and 5 not swapped

Pass 5 of 6

Would continue to perform the last pass with no swaps.



Insertion Sort

- The code (Iterative Version):

Bubble Sort

- The code (Iterative Version):

```
template <typename Item>
void bubblesort( Item a[], int left, int right)
{
        for ( int i = left; i < right; i++ )
        {
                for ( int j = right; j > i; j--)
                {
                        if ( a[j-1] > a[j] )
                        {
                                swap( a[j-1], a[j]);
                        }
                }
        }
}
```

```
// This program sorts an array's values into ascending order.
#include <iostream>

int main()
  {
     const int arraySize = 10;  // size of array a
     int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
     int hold;  // temporary location used to swap array elements

     cout << "Data items in original order\n";

     // output original array
     for ( int i = 0; i < arraySize; i++ )
        cout << “\t” << a[ i ];
```

Continued next slide…

```
   // bubble sort
   // loop to control number of passes
   for ( int pass = 0; pass < arraySize - 1; pass++ )
   {
      // loop to control number of comparisons per pass
      for ( int j = 0; j < arraySize - 1; j++ )
      {
         // compare side-by-side elements and swap them if
         // first element is greater than second element
         if ( a[ j ] > a[ j + 1 ] )
        {
            hold = a[ j ];
            a[ j ] = a[ j + 1 ];
            a[ j + 1 ] = hold;
         } // end if
       }
   }

   cout << "\nData items in ascending order\n";
   // output sorted array
   for ( int k = 0; k < arraySize; k++ )
      cout << "\t" << a[ k ];

   cout << endl;

   return 0;  // indicates successful termination

} // end main
```

Do a pass for each element in the array.

If the element on the left (index **`j`**) is larger than the element on the right (index **`j + 1`**), then we swap them. Remember the need of a temp variable.

```
Data items in original order
   2    6    4    8   10   12   89   68   45   37
Data items in ascending order
   2    4    6    8   10   12   37   45   68   89
```

Case Study: Computing Mean, Median and Mode Using Arrays

- Mean
  - Average (sum/number of elements)
- Median
  - Number in middle of sorted list
  - 1, 2, 3, 4, 5 (3 is median)
  - If even number of elements, take average of middle two
- Mode
  - Number that occurs most often
  - 1, 1, 1, 2, 3, 3, 4, 5 (1 is mode)

```
// This program introduces the topic of survey data analysis.
// It computes the mean, median, and mode of the data.
#include <iostream>

using std::cout;
using std::endl;
using std::fixed;
using std::showpoint;

#include <iomanip>

using std::setw;
using std::setprecision;

void mean( const int [], int );
 void median( int [], int );
  void mode( int [], int [], int );
  void bubbleSort( int[], int );
  void printArray( const int[], int );

int main()
{
   const int responseSize = 99;  // size of array responses
```

```
   int frequency[ 10 ] = { 0 };  // initialize array frequency

   // initialize array responses
   int response[ responseSize ] =
          { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
            7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
            6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
            7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
            6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
            7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
            5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
            7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
            7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
            4, 5, 6, 1, 6, 5, 7, 8, 7 };

   // process responses
   mean( response, responseSize );
   median( response, responseSize );
   mode( frequency, response, responseSize );

   return 0;  // indicates successful termination

} // end main
```

```
// calculate average of all response values
void mean( const int answer[], int arraySize )
{
   int total = 0;

   cout << "********\n  Mean\n********\n";

   // total response values
   for ( int i = 0; i < arraySize; i++ )
      total += answer[ i ];

   // format and output results
   cout << fixed << setprecision( 4 );

   cout << "The mean is the average value of the
        << "items. The mean is equal to the total of\n"
        << "all the data items divided by the number\n"
        << "of data items (" << arraySize
        << "). The mean value for\nthis run is: "
        << total << " / " << arraySize << " = "
        << static_cast< double >( total ) / arraySize
        << "\n\n";

} // end function mean
```

We cast to a double to get decimal points for the average (instead of an integer).

```
// sort array and determine median element's value
void median( int answer[], int size )
{
   cout << "\n********\n Median\n********\n"
        << "The unsorted array of responses is";

   printArray( answer, size );  // output unsorted array

   bubbleSort( answer, size );  // sort array

   cout << "\n\nThe sorted array is";
   printArray( answer, size );  // output so

   // display median element
   cout << "\n\nThe median is element " << size / 2
        << " of\nthe sorted " << size
        << " element array.\nFor this run the median is "
        << answer[ size / 2 ] << "\n\n";

} // end function median
```

Sort array by passing it to a function. This keeps the program modular.

```
// determine most frequent response
void mode( int freq[], int answer[], int size )
{
   int largest = 0;     // represents largest frequency
  int modeValue = 0;  // represents most frequent response

  cout << "\n********\n  Mode\n********\n";

  // initialize frequencies to 0
  for ( int i = 1; i <= 9; i++ )
     freq[ i ] = 0;

  // summarize frequencies
  for ( int j = 0; j < size; j++ )
     ++freq[ answer[ j ] ];

  // output headers for result columns
  cout << "Response" << setw( 11 ) << "Frequency"
       << setw( 19 ) << "Histogram\n\n" << setw( 55 )
       << "1    1    2    2\n" << setw( 56 )
       << "5    0    5    0    5\n\n";
```

```
   // output results
   for ( int rating = 1; rating <= 9; rating++ ) {
      cout << setw( 8 ) << rating << setw( 11 )
           << freq[ rating ] << "           ";

      // keep track of mode value and largest fequency value
      if ( freq[ rating ] > largest ) {
         largest = freq[ rating ];
         modeValue = rating;

      } // end if

      // output histogram bar representing frequency value
      for ( int k = 1; k <= freq[ rating ]; k++ )
         cout << '*';

      cout << '\n';  // begin new line of output

   } // end outer for

   // display the mode value
   cout << "The mode is the most frequent value.\n"
        << "For this run the mode is " << modeValue
        << " which occurred " << largest << " times." << endl;

} // end function mode
```

The mode is the value that occurs most often (has the highest value in **`freq`**).

```
// function that sorts an array with bubble sort algorithm
void bubbleSort( int a[], int size )
{
   int hold;  // temporary location used to swap elements

   // loop to control number of passes
   for ( int pass = 1; pass < size; pass++ )

      // loop to control number of comparisons per pass
      for ( int j = 0; j < size - 1; j++ )

         // swap elements if out of order
         if ( a[ j ] > a[ j + 1 ] ) {
            hold = a[ j ];
            a[ j ] = a[ j + 1 ];
            a[ j + 1 ] = hold;

         } // end if

} // end function bubbleSort
```

```
// output array contents (20 values per row)
void printArray( const int a[], int size )
{
   for ( int i = 0; i < size; i++ ) {

      if ( i % 20 == 0 )  // begin new line every 20 values
         cout << endl;

      cout << setw( 2 ) << a[ i ];

   } // end for

} // end function printArray
```

```
********
  Mean
********
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items (99). The mean value for
this run is: 681 / 99 = 6.8788
********
 Median
********
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

The median is element 49 of
the sorted 99 element array.
For this run the median is 7
```

```
********
  Mode
********
Response  Frequency        Histogram

                                    1    1    2    2
                               5    0    5    0    5

       1          1          *
       2          3          ***
       3          4          ****
       4          5          *****
       5          8          ********
       6          9          *********
       7         23          ***********************
       8         27          ***************************
       9         19          *******************
The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```

Searching Arrays: Linear Search and Binary Search

- Search array for a key value
- Linear search
  - Compare each element of array with key value
    - Start at one end, go to other
  - Useful for small and unsorted arrays
    - Inefficient
    - If search key not present, examines every element

Searching Arrays: Linear Search and Binary Search

- Binary search
  - Only used with sorted arrays
  - Compare middle element with key
    - If equal, match found
    - If key < middle
      - Repeat search on first half of array
    - If key > middle
      - Repeat search on last half
  - Very fast
    - At most N steps, where 2^{N} > # of elements
    - 30 element array takes at most 5 steps

      2^{5} > 30

```
// Linear search of an array.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int linearSearch( const int [], int, int );  // prototype

int main()
{
   const int arraySize = 100;  // size of array a
   int a[ arraySize ];         // create array a
   int searchKey;              // value to locate in a

   for ( int i = 0; i < arraySize; i++ )  // create some data
      a[ i ] = 2 * i;

   cout << "Enter integer search key: ";
   cin >> searchKey;

   // attempt to locate searchKey in array a
   int element = linearSearch( a, searchKey, arraySize );
```

Takes array, search key, and array size.

```
   // display results
   if ( element != -1 )
      cout << "Found value in element " << element << endl;
   else
      cout << "Value not found" << endl;

   return 0;  // indicates successful termination

} // end main

// compare key to every element of array until location is
// found or until end of array is reached; return subscript of
// element if key or -1 if key not found
int linearSearch( const int array[], int key, int sizeOfArray )
{
   for ( int j = 0; j < sizeOfArray; j++ )

      if ( array[ j ] == key )  // if found,
         return j;              // return location of key

   return -1;  // key not found

} // end function linearSearch
```

```
Enter integer search key: 36
Found value in element 18

Enter integer search key: 37
Value not found
```

```
// Binary search of an array.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::setw;

// function prototypes
int binarySearch( const int [], int, int, int, int );
void printHeader( int );
void printRow( const int [], int, int, int, int );

int main()
{
   const int arraySize = 15;  // size of array a
   int a[ arraySize ];        // create array a
   int key;                   // value to locate in a

   for ( int i = 0; i < arraySize; i++ )  // create some data
      a[ i ] = 2 * i;
```

```
   cout << "Enter a number between 0 and 28: ";
   cin >> key;

   printHeader( arraySize );

   // search for key in array a
   int result =
      binarySearch( a, key, 0, arraySize - 1, arraySize );

   // display results
   if ( result != -1 )
      cout << '\n' << key << " found in array element "
           << result << endl;
   else
      cout << '\n' << key << " not found" << endl;

   return 0;  // indicates successful termination

} // end main
```

```
// function to perform binary search of an array
int binarySearch( const int b[], int searchKey, int low,
    int high, int size )
{
    int middle;

    // loop until low subscript is greater than high
subscript
    while ( low <= high ) {

        // determine middle elemen                    g
searched
        middle = ( low + high ) / 2;

        // display subarray used in this loop iteration
        printRow( b, low, middle, high, size );
```

Determine middle element

```
      // if searchKey matches middle element, return middle
      if ( searchKey == b[ middle ] )  // match
         return middle;

      else

         // if searchKey less than m
         // set new high element
         if ( searchKey < b[ middle ] )
            high = middle - 1;  // search low end of array

         // if searchKey greater than middle element
         // set new low element
         else
            low = middle + 1;    // search
   }

   return -1;  // searchKey not found

} // end function binarySearch
```

Use the rule of binary search:
If key equals middle, match

If less, search low end

If greater, search high end

Loop sets low, middle and high dynamically. If searching the high end, the new low is the element above the middle.

```
// print header for output
void printHeader( int size )
{
   cout << "\nSubscripts:\n";

   // output column heads
   for ( int j = 0; j < size; j++ )
      cout << setw( 3 ) << j << ' ';

   cout << '\n';  // start new line of output

   // output line of - characters
   for ( int k = 1; k <= 4 * size; k++ )
      cout << '-';

   cout << endl;  // start new line of output

} // end function printHeader
```

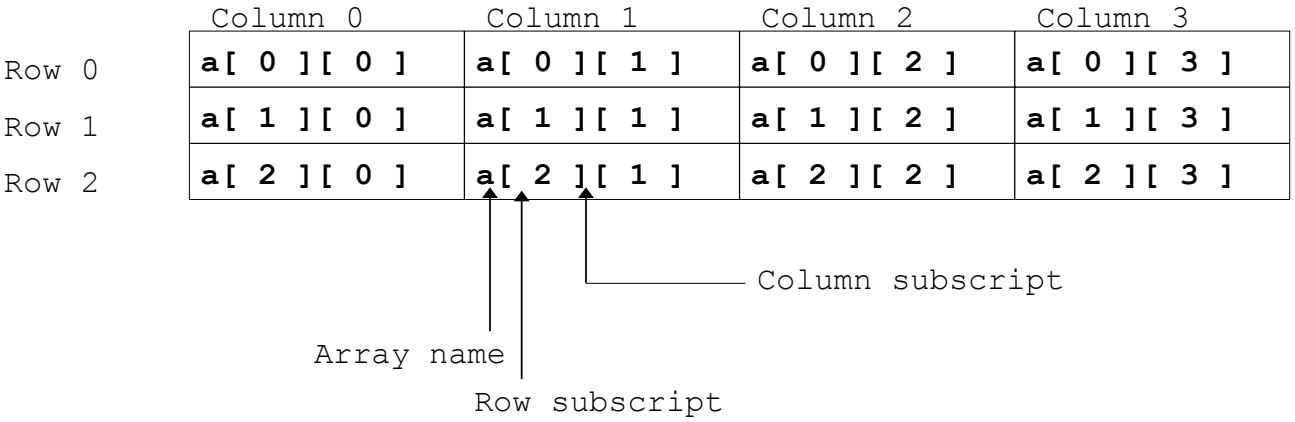
```
// print one row of output showing the current
// part of the array being processed
void printRow( const int b[], int low, int mid,
   int high, int size )
{
   // loop through entire array
   for ( int m = 0; m < size; m++ )

      // display spaces if outside current subarray range
      if ( m < low || m > high )
         cout << "    ";

      // display middle element marked with a *
      else

         if ( m == mid )           // mark middle value
            cout << setw( 3 ) << b[ m ] << '*';

         // display other elements in subarray
         else
            cout << setw( 3 ) << b[ m ] << ' ';

   cout << endl;  // start new line of output

} // end function printRow
```

```
Enter a number between 0 and 28: 6

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
---------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12

6 found in array element 3



Enter a number between 0 and 28: 25

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
---------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
                              16  18  20  22* 24  26  28
                                          24  26* 28
                                          24*

25 not found
```

```
Enter a number between 0 and 28: 8

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
---------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12
                  8  10* 12
                  8*

8 found in array element 4
```

Multiple-Subscripted Arrays (Multi-dimensional Arrays)

- Multiple subscripts
  - `a[ i ][ j ]`
  - Tables with rows and columns
  - Specify row, then column
  - "Array of arrays"
    - `a[0]` is an array of 4 elements
    - `a[0][0]` is the first element of that array



Multiple-Subscripted Arrays

- To initialize
  - Default of **0**
  - Initializers grouped by row in braces

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

Row 0 **Row 1**

1	2
3	4

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

1	0
3	4

Multiple-Subscripted Arrays

- Referenced like normal

```
cout << b[ 0 ][ 1 ];
```

1	0
3	4

  - Outputs **0**
  - Cannot reference using commas

```
cout << b[ 0, 1 ];
```

    - Syntax error

- Function prototypes
  - Must specify sizes of subscripts
    - First subscript not necessary, as with single-scripted arrays
  - `void printArray( int [][ 3 ] );`

```
1  // Fig. 4.22: fig04_22.cpp
2  // Initializing multidimensional arrays.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  void printArray( int [][ 3 ] );
9
10 int main()
11 {
12    int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
13    int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
14    int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
15
16    cout << "Values in array1 by row are:" << endl;
17    printArray( array1 );
18
19    cout << "Values in array2 by row are:" << endl;
20    printArray( array2 );
21
22    cout << "Values in array3 by row are:" << endl;
23    printArray( array3 );
24
25    return 0;  // indicates successful termination
26
27 } // end main
```

Note the format of the prototype.

Note the various initialization styles. The elements in **`array2`** are assigned to the first row and then the second.

```
28
29 // function to output array with two rows
30 void printArray( int a[][ 3 ] )
31 {
32    for ( int i = 0; i < 2; i++ ) {     // f
33
34       for ( int j = 0; j < 3; j++ )   // output column values
35          cout << a[ i ][ j ] << ' ';
36
37       cout << endl;  // start new line of output
38
39    } // end outer for structure
40
41 } // end function printArray
```

For loops are often used to iterate through arrays. Nested loops are helpful with multiple-subscripted arrays.

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Multiple-Subscripted Arrays

- Next: program showing initialization
  - After, program to keep track of students grades
  - Multiple-subscripted array (table)
  - Rows are students
  - Columns are grades

	Quiz1	Quiz2
Student0	95	85
Student1	89	80

```cpp
// Fig. 4.23: fig04_23.cpp
// Double-subscripted array example.
#include <iostream>

using std::cout;
using std::endl;
using std::fixed;
using std::left;

#include <iomanip>

using std::setw;
using std::setprecision;

const int students = 3;    // number of students
const int exams = 4;       // number of exams

// function prototypes
int minimum( int [][ exams ], int, int );
int maximum( int [][ exams ], int, int );
double average( int [], int );
void printArray( int [][ exams ], int, int );

```

```
24 int main()
25 {
26    // initialize student grades for three students (rows)
27    int studentGrades[ students ][ exams ] =
28       { { 77, 68, 86, 73 },
29         { 96, 87, 89, 78 },
30         { 70, 90, 86, 81 } };
31
32    // output array studentGrades
33    cout << "The array is:\n";
34    printArray( studentGrades, students, exams );
35
36    // determine smallest and largest grade values
37    cout << "\n\nLowest grade: "
38         << minimum( studentGrades, students, exams )
39         << "\nHighest grade: "
40         << maximum( studentGrades, students, exams ) << '\n';
41
42    cout << fixed << setprecision( 2 );
43
```

```
44    // calculate average grade for each student
45    for ( int person = 0; person < students; person++ )
46       cout << "The average grade for student " << person
47            << " is "
48            << average( studentGrades[ person ], exams )
49            << endl;
50
51    return 0;  // indicates successful termin
52
53 } // end main
54
55 // find minimum grade
56 int minimum( int grades[][ exams ], int pupi
57 {
58    int lowGrade = 100; // initialize to highest possible grade
59
60    for ( int i = 0; i < pupils; i++ )
61
62       for ( int j = 0; j < tests; j++ )
63
64          if ( grades[ i ][ j ] < lowGrade )
65             lowGrade = grades[ i ][ j ];
66
67    return lowGrade;
68
69 } // end function minimum
```

Determines the average for one student. We pass the array/row containing the student's grades. Note that **`studentGrades[0]`** is itself an array.

```
70
71 // find maximum grade
72 int maximum( int grades[][ exams ], int pupils, int tests )
73 {
74    int highGrade = 0;  // initialize to lowest possible grade
75
76    for ( int i = 0; i < pupils; i++ )
77
78       for ( int j = 0; j < tests; j++ )
79
80          if ( grades[ i ][ j ] > highGrade )
81             highGrade = grades[ i ][ j ];
82
83    return highGrade;
84
85 } // end function maximum
86
```

```
87 // determine average grade for particular student
88 double average( int setOfGrades[], int tests )
89 {
90    int total = 0;
91
92    // total all grades for one student
93    for ( int i = 0; i < tests; i++ )
94       total += setOfGrades[ i ];
95
96    return static_cast< double >( total ) / tests;  // average
97
98 } // end function maximum
```

```
99
100 // Print the array
101 void printArray( int grades[][ exams ], int pupils, int tests )
102 {
103    // set left justification and output column heads
104    cout << left << "                [0]  [1]  [2]  [3]";
105
106    // output grades in tabular format
107    for ( int i = 0; i < pupils; i++ ) {
108
109       // output label for row
110       cout << "\nstudentGrades[" << i << "] ";
111
112       // output one grades for one student
113       for ( int j = 0; j < tests; j++ )
114          cout << setw( 5 ) << grades[ i ][ j ];
115
116    } // end outer for
117
118 } // end function printArray
```

```
The array is:
                 [0]  [1]  [2]  [3]
studentGrades[0] 77   68   86   73
studentGrades[1] 96   87   89   78
studentGrades[2] 70   90   86   81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```