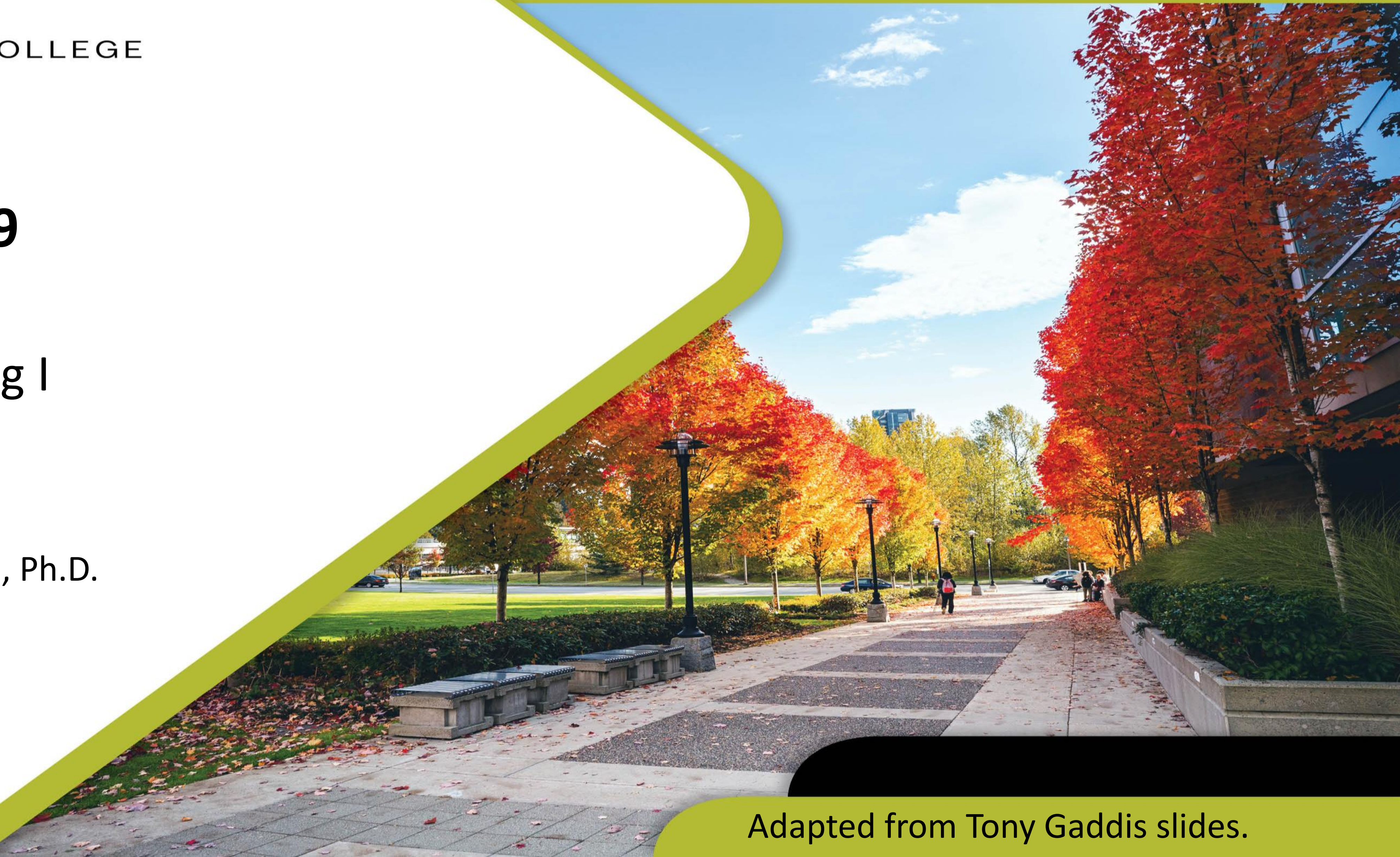
DOUGLAS COLLEGE

CMPT 1109

Programming I

Shahriar Khosravi, Ph.D.

Lecture 7



Adapted from Tony Gaddis slides.

Plan for Today

- Introduction to Search Algorithms
- Introduction to Sorting Algorithms

DOUGLAS COLLEGE

Search Algorithms

Search Algorithms

- A **search algorithm** is a method of locating a specific item in a larger collection of data.
- We will examine two algorithms for searching in an array:
  - Linear search (also known as the sequential search)
  - Binary search

Linear Search

- Starting at the first element, this algorithm sequentially steps through an array examining each element until it locates the value it is searching for.
- Example - array **`numlist`** contains:

17	23	5	11	2	29	3

- Searching for the value **11**, linear search examines **17**, **23**, **5**, and **11** before a match is found.
- Searching for the value **7**, linear search examines all elements and will not find a match.

Linear Search - Algorithm

```
Set found to false
Set position to −1
Set index to 0
While found is false and index < number of elements
   If list[index] is equal to search value
      found = true
      position = index
   End If
   Add 1 to index
End While
Return position
```

Linear Search – Example

```cpp
// This program demonstrates the linear search algorithm.
#include <iostream>
using namespace std;

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

int main()
{
    const int SIZE = 5;
    int tests[SIZE] = { 87, 75, 98, 100, 82 };
    int results;

    // Search the array for 100.
    results = linearSearch(tests, SIZE, 100);

    // If linearSearch returned -1, then 100 was not found.
    if (results == -1)
        cout << "You did not earn 100 points on any test\n";
    else
    {
        // Otherwise results contains the subscript of
        // the first 100 in the array.
        cout << "You earned 100 points on test ";
        cout << (results + 1) << endl;
    }
    return 0;
}

int linearSearch(const int arr[], int size, int value)
{
    int index = 0;       // Used as a subscript to search array
    int position = -1;   // To record position of search value
    bool found = false;  // Flag to indicate if the value was found

    while (index < size && !found)
    {
        if (arr[index] == value)  // If the value is found
        {
            found = true;         // Set the flag
            position = index;     // Record the value's subscript
        }
        index++;                  // Go to the next element
    }
    return position;              // Return the position, or -1
}
```

Linear Search Notes

- Benefits:
  - An easy algorithm to understand and code.
  - Array can be in any order (i.e., does not have to be sorted prior to searching).
- Disadvantages:
  - Inefficient (i.e., slow): for array of **N** elements, examines **N/2** elements on average if the search value exists in the array.
  - Otherwise, examines **N** elements to determine value is not present in the array.

DOUGLAS COLLEGE

Binary Search

Binary Search

- The **binary search algorithm** requires array elements to be in order (i.e., sorted).

1. Divides the array into three sections:
   - middle element
   - elements on one side of the middle element
   - elements on the other side of the middle element
2. If the middle element is the correct value, we are done. Otherwise, return to step 1, using only the half of the array that may contain the correct value.
3. Continue steps 1 and 2 until either the value is found or there are no more elements to examine.

Binary Search – Example

- Array **`numlist2`** contains:

2	3	5	11	17	23	29

- Searching for the value **11**, the algorithm examines the middle element and stops right there.
- Searching for the value **7**, linear search examines **11**, the middle element first. Then, since **11** is not the correct value and that **11** is larger than the correct value, it examines the lower-half of the array in the next pass.
- The lower half contains elements **2**, **3**, and **5**. Element **3** is the middle one, so it is examined first.
- Since **3** is not the correct value and that **3** is smaller than the correct value, it examines **2**.
- The algorithm has now conclusively determined that **7** is not present in this array.

Binary Search – Example

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

// Function prototype
int binarySearch(const int[], int, int);
const int SIZE = 20;

int main()
{
    // Array with employee IDs sorted in ascending order.
    int idNums[SIZE] = { 101, 142, 147, 189, 199, 207, 222,
                        234, 289, 296, 310, 319, 388, 394,
                        417, 429, 447, 521, 536, 600 };
    int results;   // To hold the search results
    int empID;     // To hold an employee ID

    // Get an employee ID to search for.
    cout << "Enter the employee ID you wish to search for: ";
    cin >> empID;

    // Search for the ID.
    results = binarySearch(idNums, SIZE, empID);

    // If results contains -1 the ID was not found.
    if (results == -1)
        cout << "That number does not exist in the array.\n";
    else
    {
        // Otherwise results contains the subscript of
        // the specified employee ID in the array.
        cout << "That ID is found at element " << results;
        cout << " in the array.\n";
    }
    return 0;
}

int binarySearch(const int array[], int size, int value)
{
    int first = 0,             // First array element
        last = size - 1,       // Last array element
        middle,                // Mid point of search
        position = -1;         // Position of search value
    bool found = false;        // Flag

    while (!found && first <= last)
    {
        middle = (first + last) / 2;     // Calculate mid point
        if (array[middle] == value)      // If value is found at mid
        {
            found = true;
            position = middle;
        }
        else if (array[middle] > value)  // If value is in lower half
            last = middle - 1;
        else
            first = middle + 1;           // If value is in upper half
    }
    return position;
}
```

Binary Search Notes

- Benefits:
  - Much more efficient than linear search. For an array of N elements, it performs at most $\log_2 N$ comparisons.
- Disadvantages:
  - Requires that array elements be sorted first. The binary search algorithm will not work properly unless the values in the array are sorted.

DOUGLAS COLLEGE

Sort Algorithms

Sort Algorithms

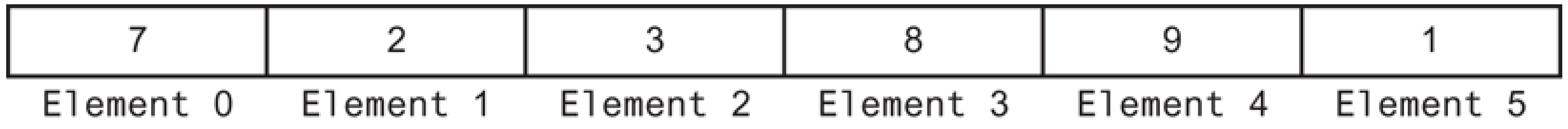
- Sorting algorithms are used to arrange data into some order:
  - Alphabetical
  - Ascending numeric
  - Descending numeric
- We will examine two algorithms for searching in an array:
  - Bubble sort
  - Selection sort

Bubble Sort

- How it works:
    - Compare 1st two elements.
        - If out of order, exchange them to put in order.
    - Move down one element, compare 2nd and 3rd elements, exchange if necessary to put in correct order. Continue doing this until the end of the array is reached.
    - Pass through array again, exchanging as necessary
    - Repeat until pass made with no exchanges

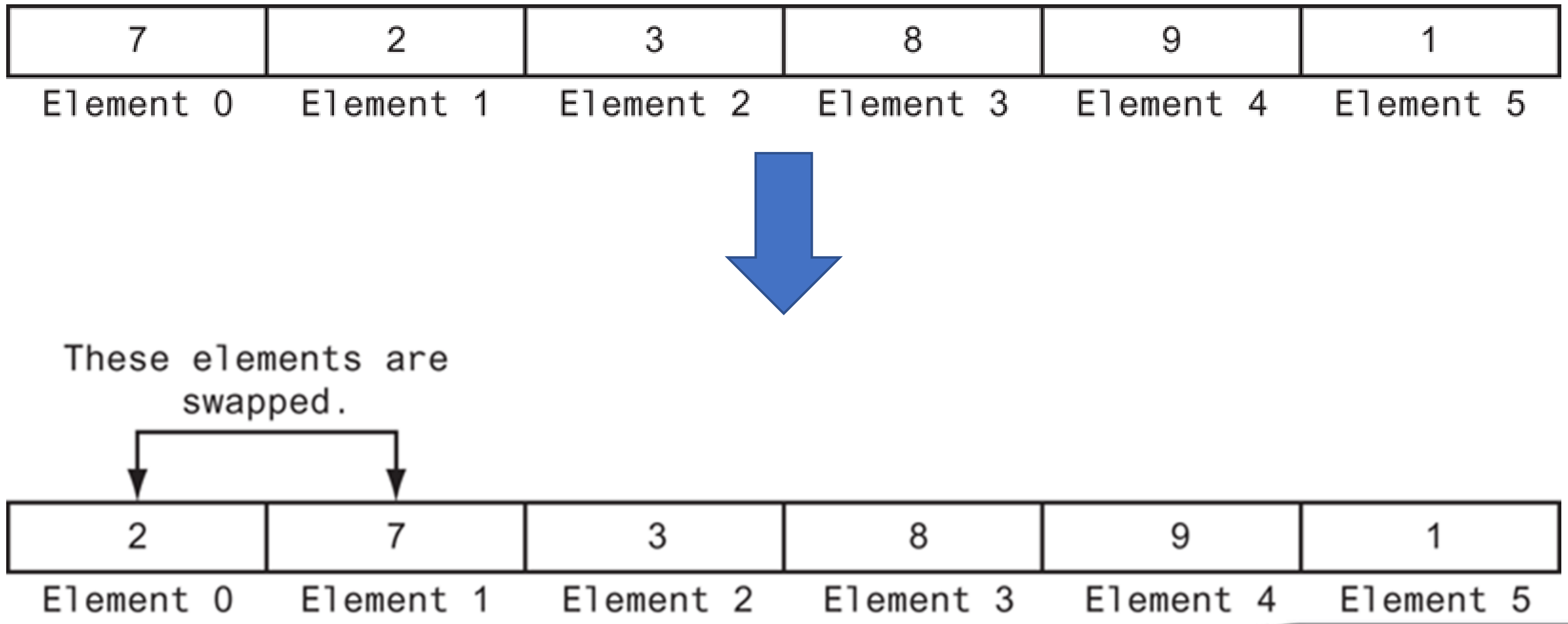
Bubble Sort

- Suppose we have the array shown below.
- The bubble sort starts by comparing the first two elements in the array. If element **`0`** is greater than element **`1`**, they are swapped.



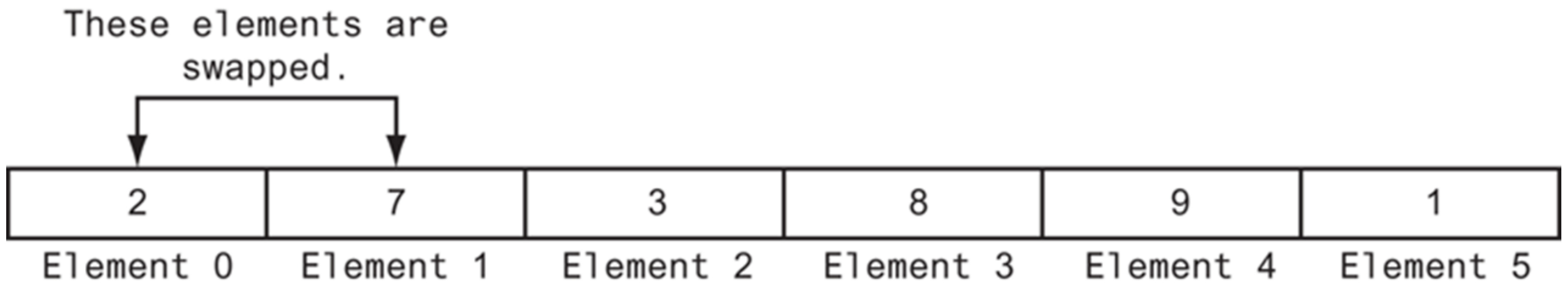
Bubble Sort

- Suppose we have the array shown below.
- The bubble sort starts by comparing the first two elements in the array. If element **0** is greater than element **1**, they are swapped.



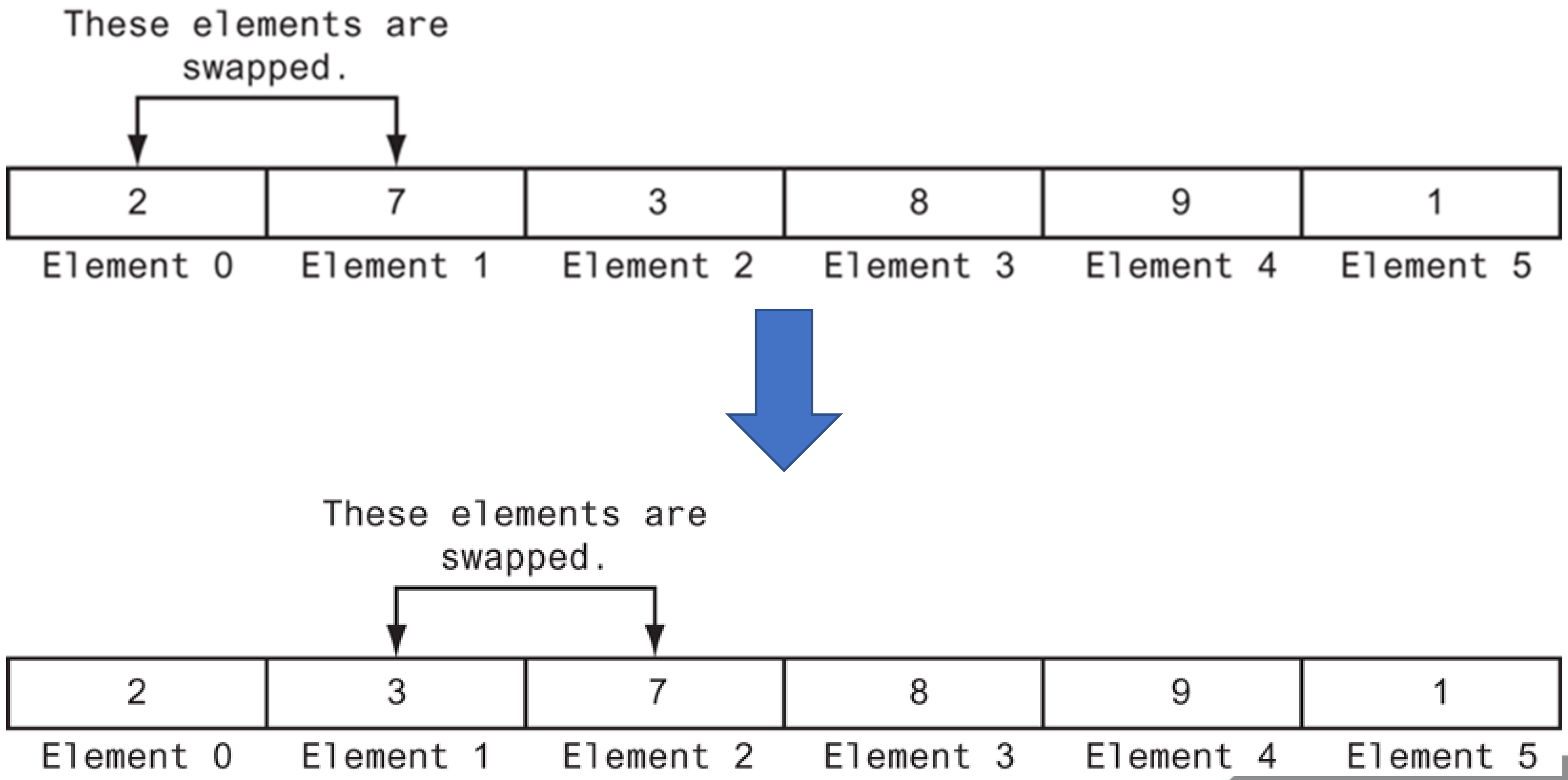
Bubble Sort

- This method is repeated with elements **1** and **2**. If element **1** is greater than element **2**, they are swapped.



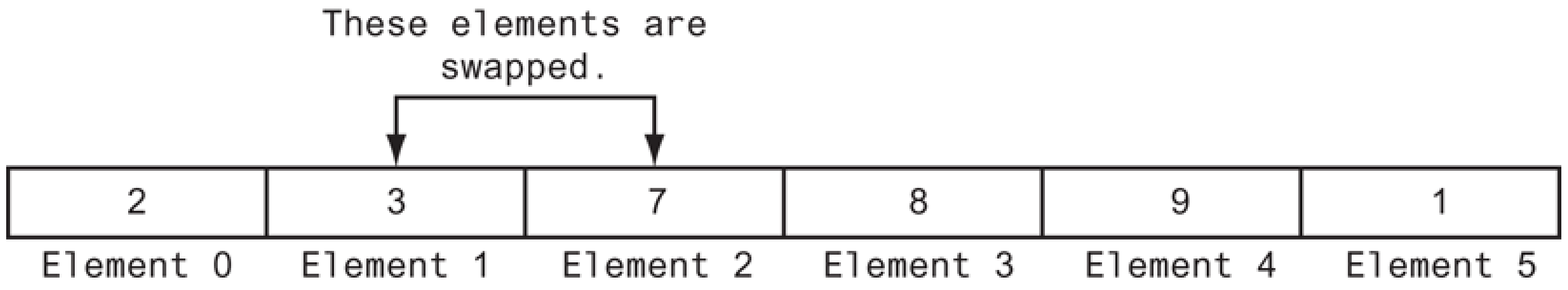
Bubble Sort

- This method is repeated with elements **1** and **2**. If element **1** is greater than element **2**, they are swapped.



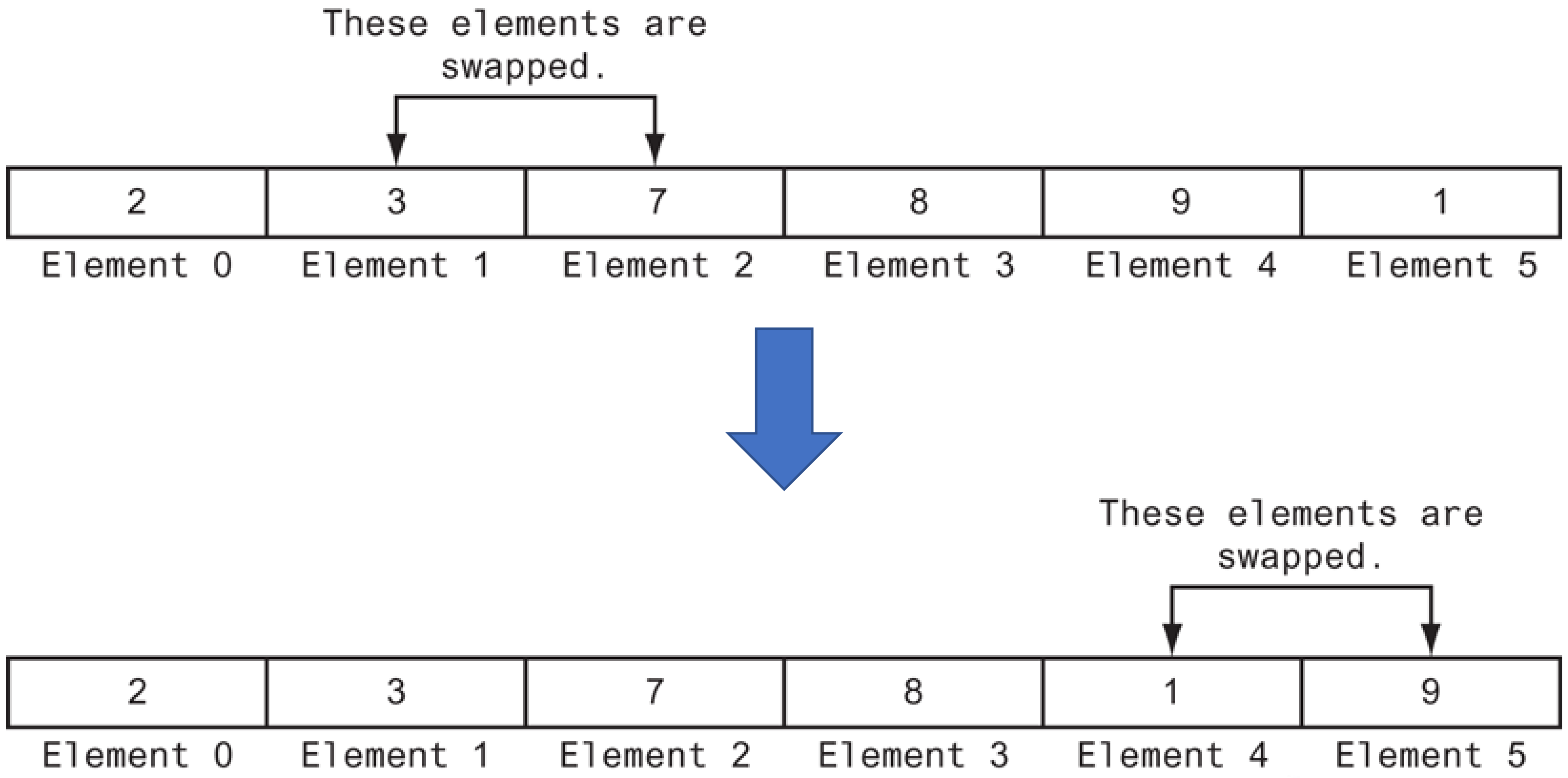
Bubble Sort

- Next, elements **2** and **3** are compared. In this array, these elements are already in the proper order (element **2** is less than element **3**), so no values are swapped.
- As the cycle continues, elements **3** and **4** are compared. Once again, it is not necessary to swap the values because they are already in the proper order.



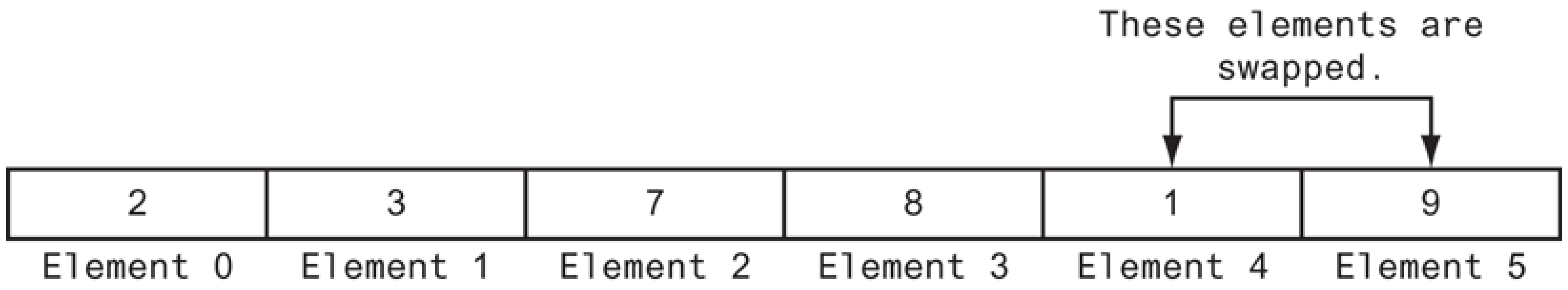
Bubble Sort

- When elements **4** and **5** are compared, however, they must be swapped because element **4** is greater than element **5**.



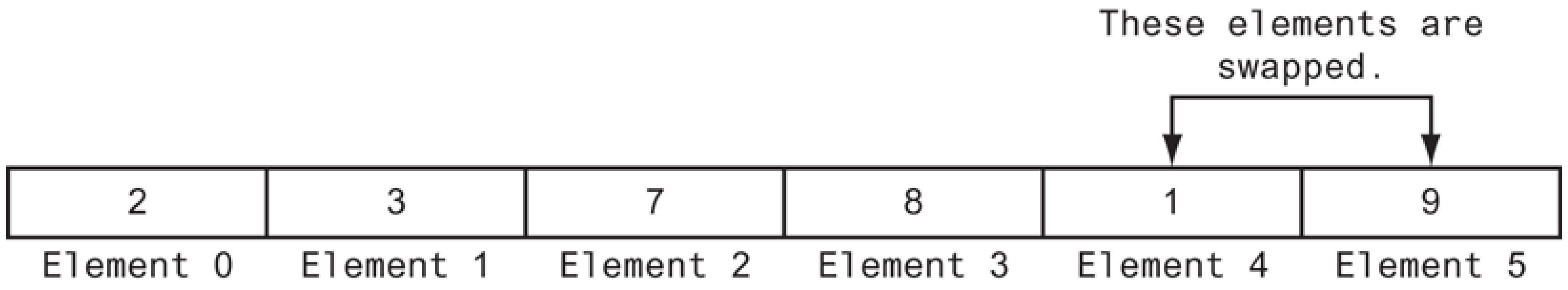
Bubble Sort

- At this point, the entire array has been scanned once, **and the largest value, 9, is in the correct position**.
- But there are other elements that are not yet in their final positions.
- The algorithm will then make another pass through the array, comparing each element with its neighbor.
- In this pass pass, **it will stop comparing after reaching the next-to-last element because the last element already contains the correct value**.



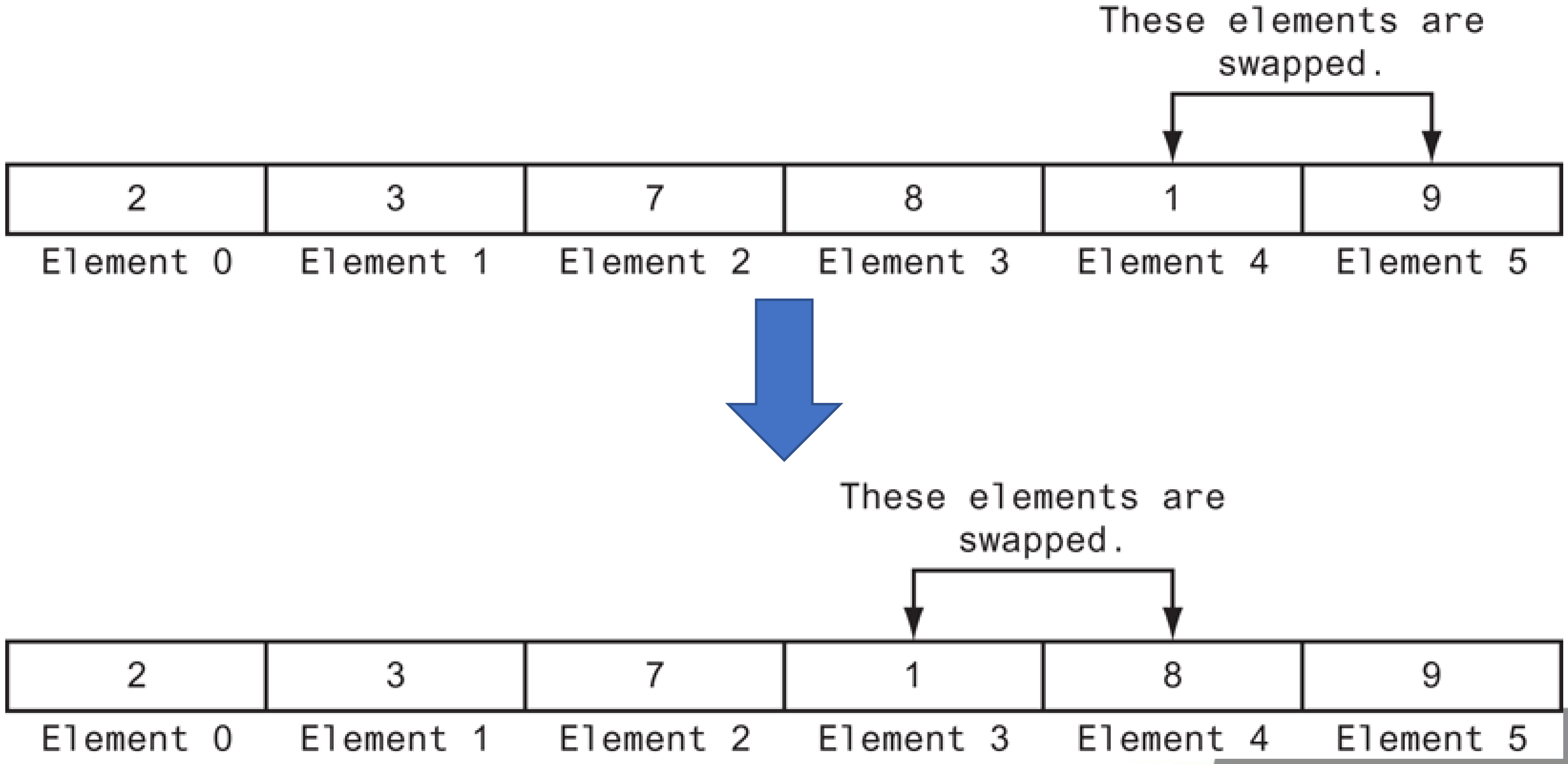
Bubble Sort

- The second pass starts by comparing elements **`0`** and **`1`**.
- Since those two are in the correct order, they are not swapped.
- Elements **1** and **2** are compared next, but once again, they are not swapped.



Bubble Sort

- his continues until elements **3** and **4** are compared.
- Since element **3** is greater than element **4**, they are swapped.
- Element **4** is the last element that is compared during this pass, so this pass stops.



Bubble Sort

- At the end of the second pass, the last two elements in the array contain the correct values.
- The third pass starts now, comparing each element with its neighbor.
- The third pass will not involve the last two elements because they have already been sorted.
- When the third pass is finished, the last three elements will hold the correct values.

2	3	1	7	8	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

- Each time the algorithm makes a pass through the array, the portion of the array that is scanned is decreased in size by one element, and the largest value in the scanned portion of the array is moved to its final position.

1	2	3	7	8	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Bubble Sort – Example

```cpp
// This program demonstrates the Bubble Sort algorithm.
#include <iostream>
using namespace std;

// Function prototypes
void bubbleSort(int[], int);
void swap(int&, int&);

int main()
{
    const int SIZE = 6;

    // Array of unsorted values
    int values[SIZE] = { 6, 1, 5, 2, 4, 3 };

    // Display the unsorted array.
    cout << "The unsorted values:\n";
    for (auto element : values)
        cout << element << " ";
    cout << endl;

    // Sort the array.
    bubbleSort(values, SIZE);

    // Display the sorted array.
    cout << "The sorted values:\n";
    for (auto element : values)
        cout << element << " ";
    cout << endl;

    return 0;
}


void bubbleSort(int array[], int size)
{
    int maxElement;
    int index;

    for (maxElement = size - 1; maxElement > 0; maxElement--)
    {
        for (index = 0; index < maxElement; index++)
        {
            if (array[index] > array[index + 1])
            {
                swap(array[index], array[index + 1]);
            }
        }
    }
}

void swap(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

Bubble Sort Notes

- Benefits:
  - Simple, easy to understand and implement.

- Disadvantages:
  - Inefficient because values move by only one element at a time toward their final destination in the array.

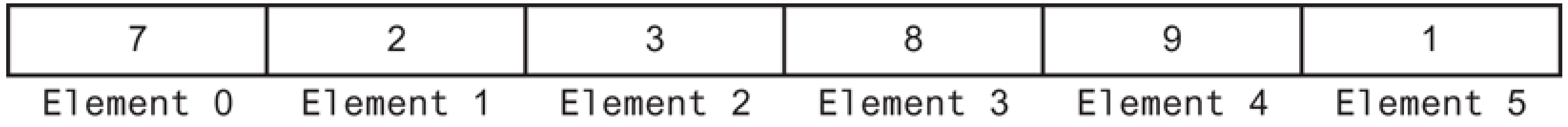
Selection Sort

Selection Sort

- How it works (sort in ascending order):
  - Locate smallest element in array. Exchange it with element in position `0`.
  - Locate next smallest element in array. Exchange it with element in position `1`.
  - Continue until all elements are arranged in order.

Selection Sort

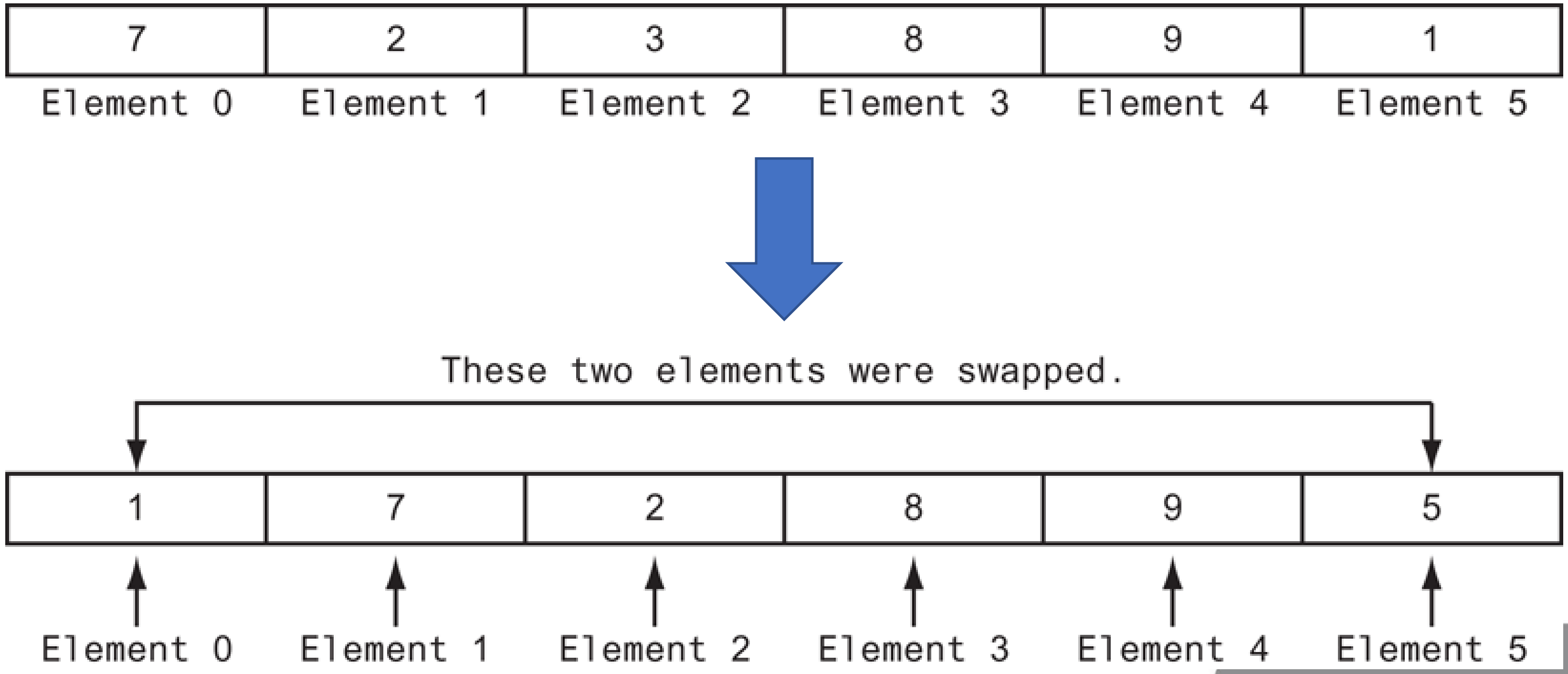
- Suppose we have the array shown below.
- The selection sort scans the array, starting at element 0, and locates the element with the smallest value.
- Then, the contents of this element are swapped with the contents of element `0`.



7	2	3	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

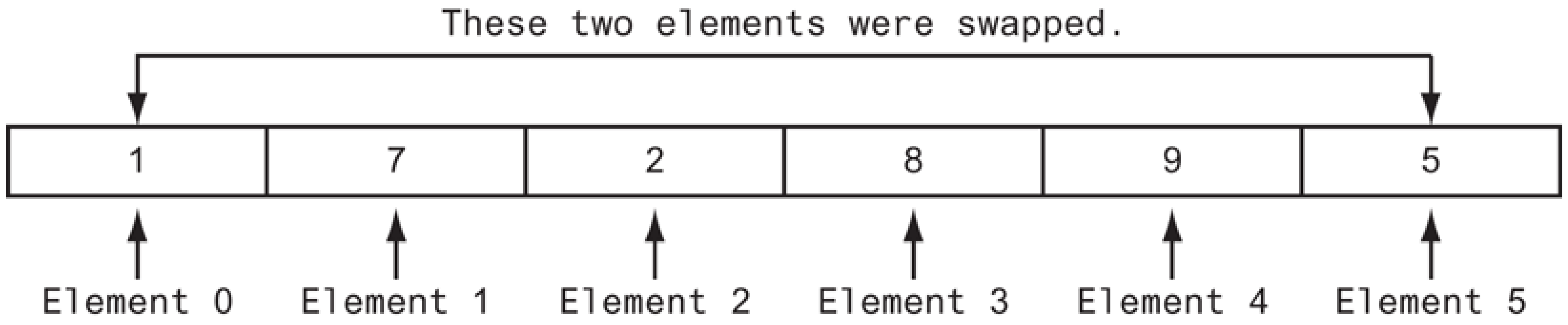
Selection Sort

- Suppose we have the array shown below.
- The selection sort scans the array, starting at element 0, and locates the element with the smallest value.
- Then, the contents of this element are swapped with the contents of element **`0`**.



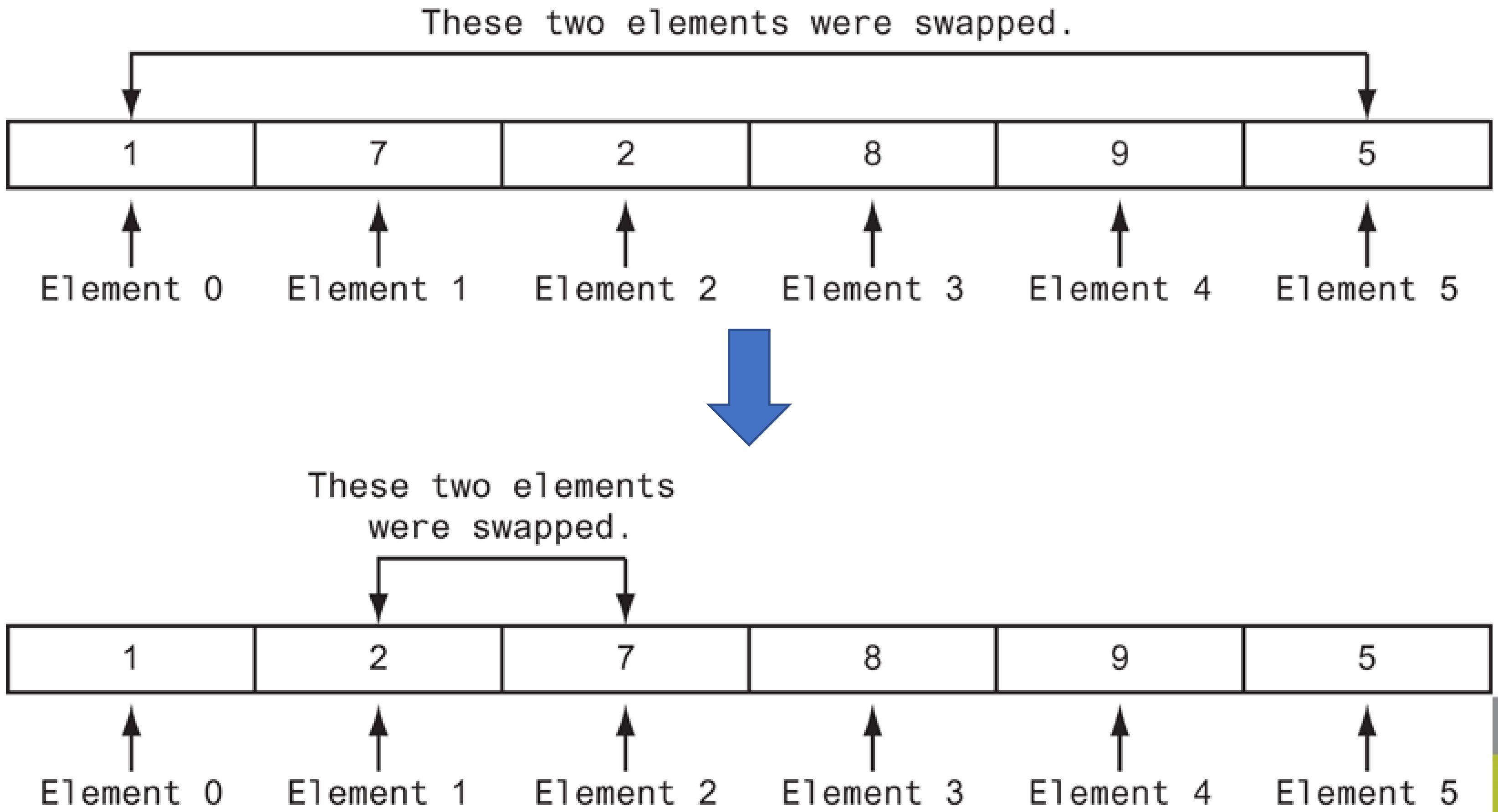
Selection Sort

- The algorithm then repeats the process, but since element **0** already contains the smallest value in the array, it can be left out of the procedure.
- This time, the algorithm begins the scan at element **1**.



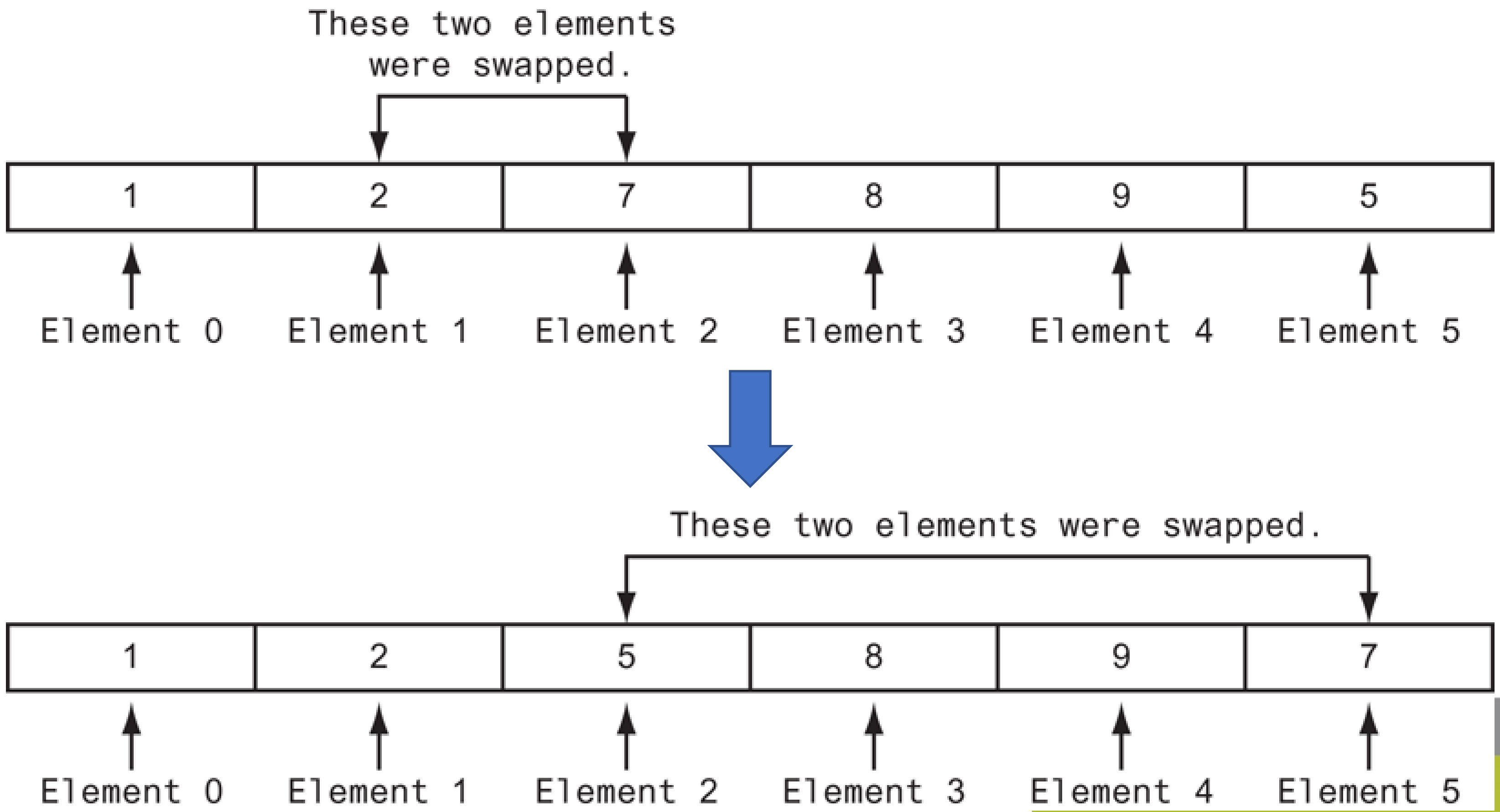
Selection Sort

- The algorithm then repeats the process, but since element **`0`** already contains the smallest value in the array, it can be left out of the procedure.
- This time, the algorithm begins the scan at element **`1`**.



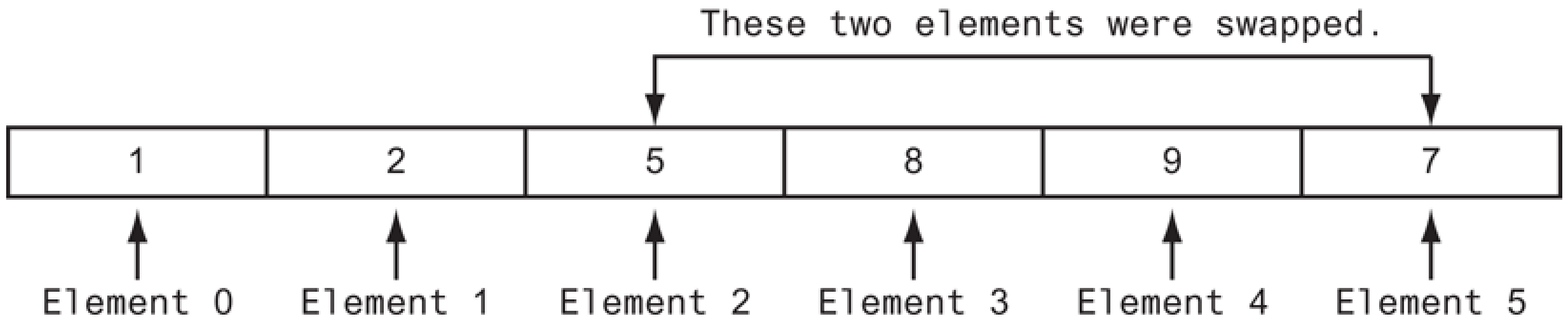
Selection Sort

- Once again the process is repeated, but this time the scan begins at element **2**.
- It will find that element **5** contains the next smallest value. This element's value is swapped with that of element **2**.



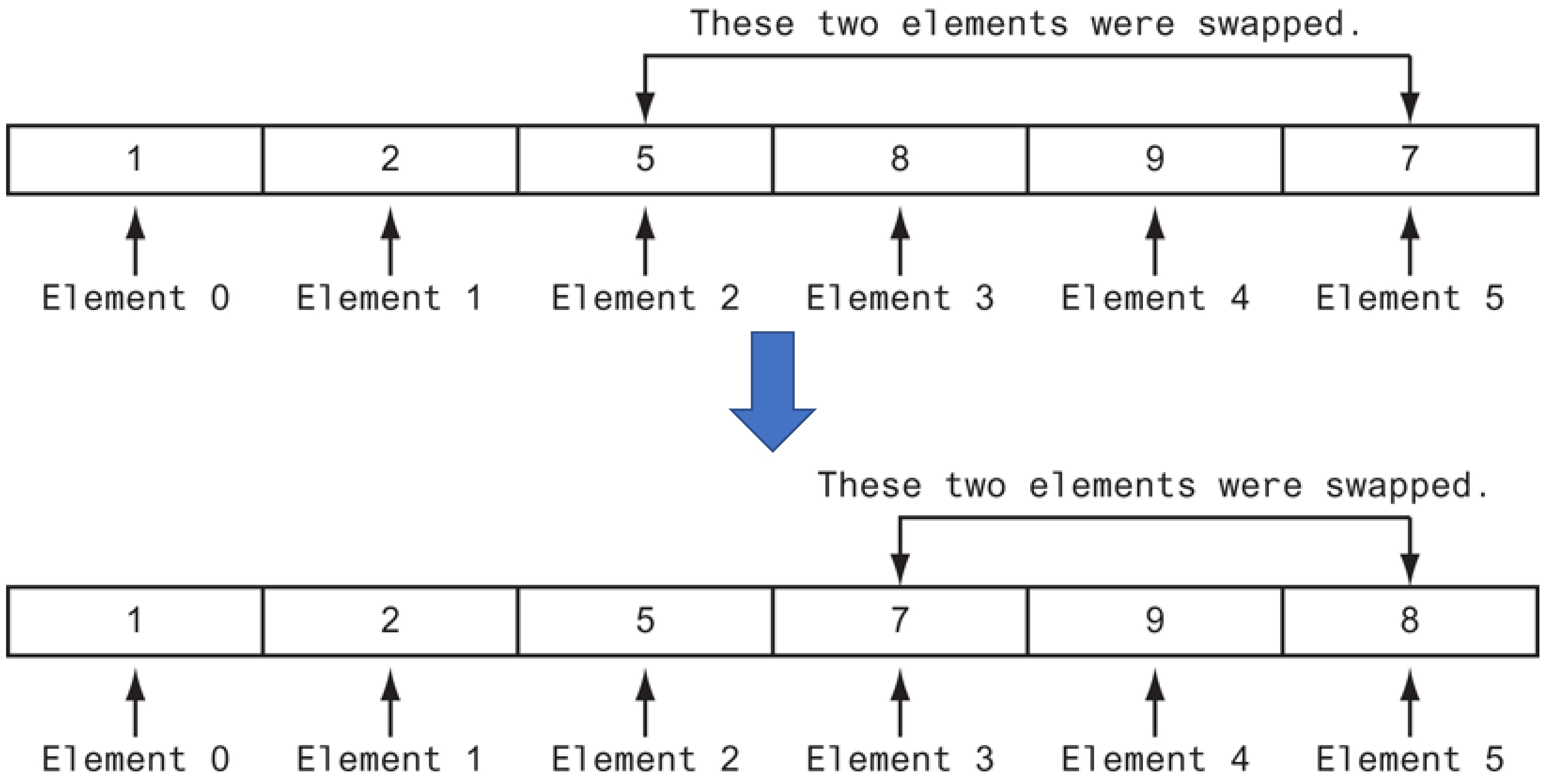
Selection Sort

- Next, the scanning begins at element 3. Its value is swapped with that of element 5.



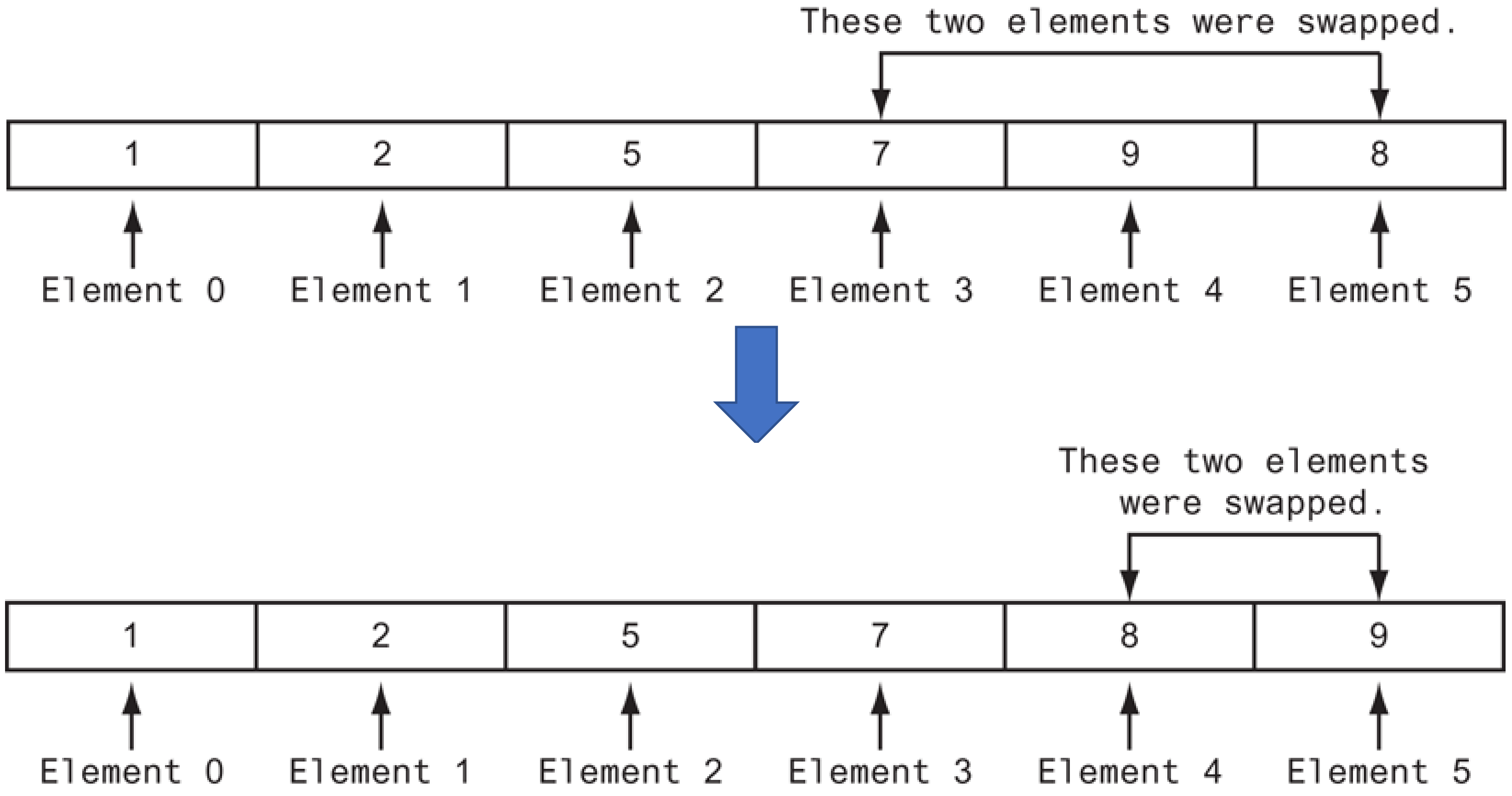
Selection Sort

- Next, the scanning begins at element **3**. Its value is swapped with that of element **5**.



Selection Sort

- At this point, there are only two elements left to sort.



Selection Sort – Example

```cpp
// This program demonstrates the Selection Sort algorithm.
#include <iostream>
using namespace std;

// Function prototypes
void selectionSort(int[], int);
void swap(int&, int&);

int main()
{
    const int SIZE = 6;

    // Array of unsorted values
    int values[SIZE] = { 6, 1, 5, 2, 4, 3 };

    // Display the unsorted array.
    cout << "The unsorted values:\n";
    for (auto element : values)
        cout << element << " ";
    cout << endl;

    // Sort the array.
    selectionSort(values, SIZE);

    // Display the sorted array.
    cout << "The sorted values:\n";
    for (auto element : values)
        cout << element << " ";
    cout << endl;

    return 0;
}

void selectionSort(int array[], int size)
{
    int minIndex, minValue;

    for (int start = 0; start < (size - 1); start++)
    {
        minIndex = start;
        minValue = array[start];
        for (int index = start + 1; index < size; index++)
        {
            if (array[index] < minValue)
            {
                minValue = array[index];
                minIndex = index;
            }
        }
        swap(array[minIndex], array[start]);
    }
}

void swap(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

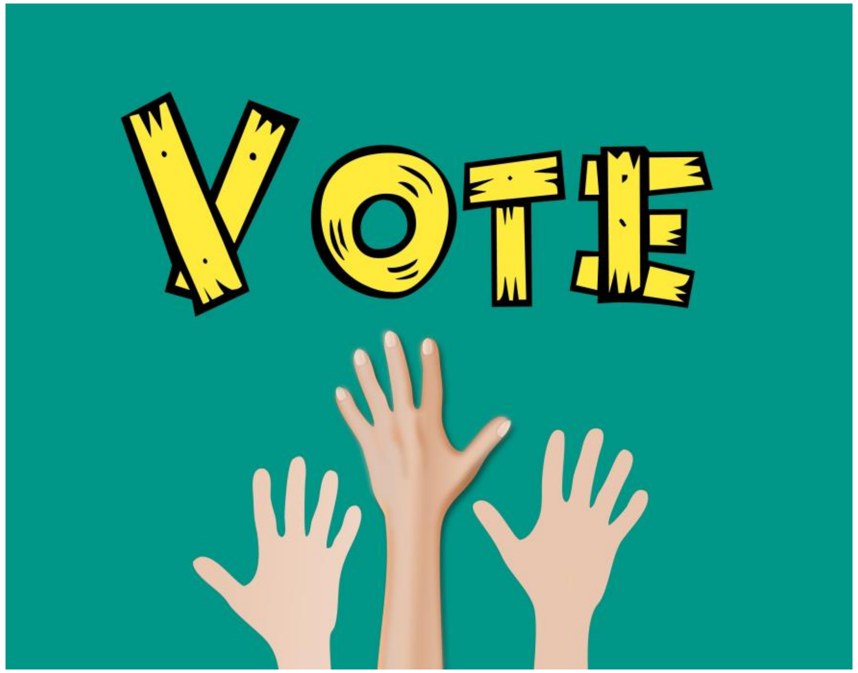
In-Class Exercise

Write a C++ program that has an array of at least 20 randomly-generated integers. It should call a function that uses the linear search algorithm to locate one of the values. The function should keep a count of the number of comparisons it makes until it finds the value. The program then should call a function that uses the binary search algorithm to locate the same value. It should also keep count of the number of comparisons it makes. Display these values on the screen.

Poll 1 (Extra Credit)

a) Yay!
b) Nay!

Please use the "Poll" window to participate for extra credit! One answer only please!



Thank you.

DOUGLAS COLLEGE

DOUGLAS COLLEGE