Searching and Sorting
Complexity of Algorithms

In this Week

- Complexity of Search Algorithms
  - Sequential (Linear) Search: **O(n)**
  - Binary Search: **O(log n)**
- Complexity of Sorting Algorithms
  - Selection Sort: **$O(n^2)$**
  - Bubble Sort: **$O(n^2)$**
  - Insertion Sort: **$O(n^2)$**
- Counting Critical Operations

Class of Mathematical Functions

- Mathematical functions are grouped according to how fast they grow into several classes
- Given some appropriate real numbers a, b, c, and d; some of the common class of functions are
    - Constant functions: **$f(x) = a$**
    - Logarithmic functions: **$f(x) = a \log_b x$ ….. (assume $b > 1$)**
    - Linear functions: **$f(x) = ax + b$**
    - Quadratic functions: **$f(x) = ax^2 + bx + c$**
    - Cubic functions: $f(x) =$ **$ax^3 + bx^2 + cx + d$**
    - Exponential functions: **$f(x) = b\, a^x$ ….. (assume $a > 1$)**

Growth of Mathematical Functions

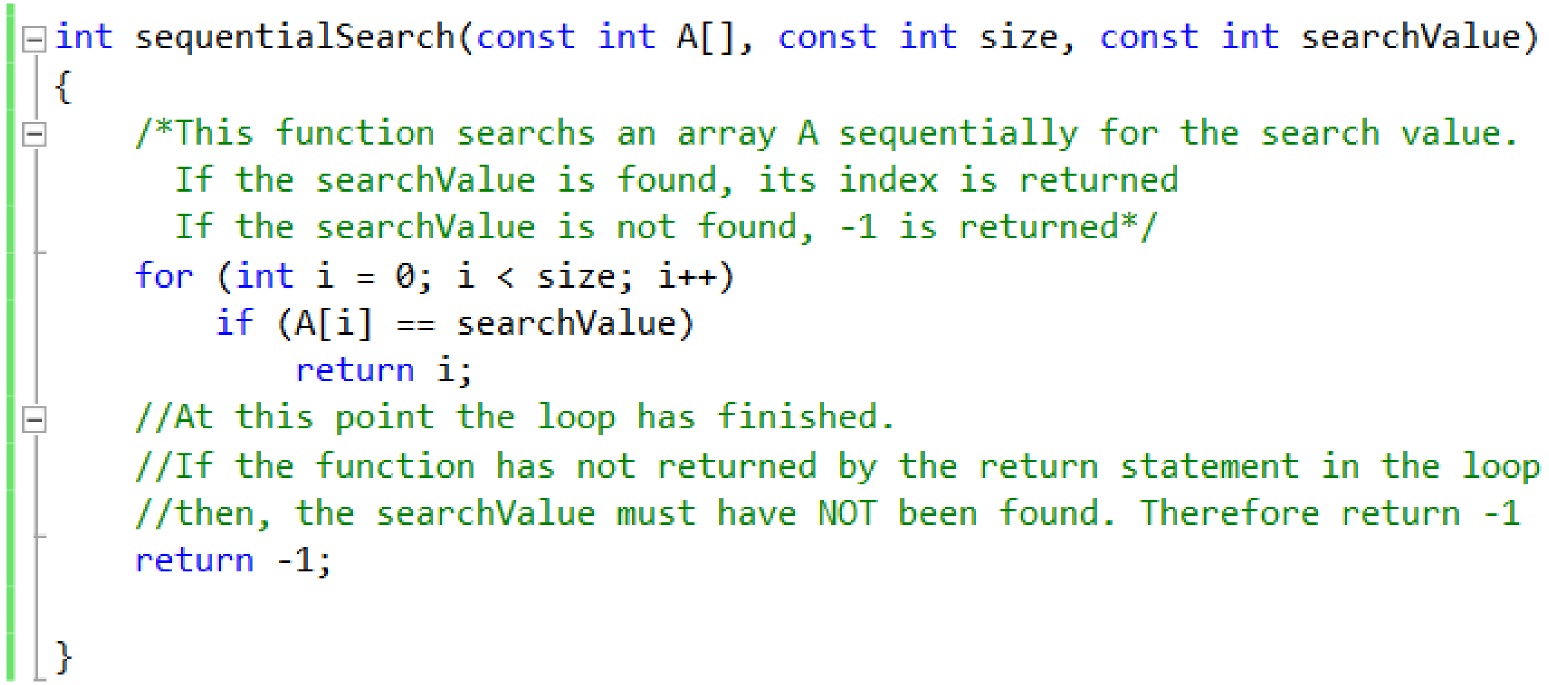
- Different classes of functions grow differently
- For example, the constant function is the slowest growing class of functions while the exponential function is the fastest growing class of functions
- The class of functions described above are actually listed according to the speed of their growth
- Moreover, the growth of any class of functions is influenced mainly by the variable in the function; the coefficients don't play any role
- For example $f(x) = 18x^2 + 7x + 5$ and $f(x) = 100000x^2 + 999x - 25$ are in the same class of functions (quadratic)
- Finally the highest power of the variable determines the growth of a function; the other variables don't play role
- For this reason $f(x) = 0.000000001x^2$ grows faster than $f(x) = 100000x$
- For this reason we specify class of functions with the leading power only
- Such description of functions is known as **Big-O** notation

Big-O Notation

- Thus using Big-O notation, the class of functions described earlier can be written as follows
  - Constant functions: **$f(x) = O(1)$**
  - Logarithmic functions: **$f(x) = O(\log x)$**.....**(assume base > 1)**
  - Linear functions: **$f(x) = O(x)$**
  - Quadratic functions: **$f(x) = O(x^2)$**
  - Cubic functions: $f(x) =$ **$O(x^3)$**
  - Exponential functions: **$f(x) = O(a^x)$**.....**(assume $a > 1$)**

Sequential Search

- Search for a value in an array sequentially!
  - ➢ The following code demonstrates searching a C++ static array sequentially for a specific value



```cpp
int sequentialSearch(const int A[], const int size, const int searchValue)
{
    /*This function searchs an array A sequentially for the search value.
      If the searchValue is found, its index is returned
      If the searchValue is not found, -1 is returned*/
    for (int i = 0; i < size; i++)
        if (A[i] == searchValue)
            return i;
    //At this point the loop has finished.
    //If the function has not returned by the return statement in the loop
    //then, the searchValue must have NOT been found. Therefore return -1
    return -1;

}
```

Sequential Search

```
int main()
{
    const unsigned int size = 300000000;
    int *A = new int[size];
    srand(time(0));

    //Fill the array
    for (int i = 0; i < size; i++)
        A[i] = rand() % 200;


    //Perform sequencial Search
    int value = 200;    //Select a number not found in the array
    cout << "Press the ENTER key to Start sequential search..." << endl;
    getchar();  //This will pause your program until you press the ENTER key
    int index = sequencialSearch(A, size, value);
    cout << "Finished sequencial search..." << endl;

    //Print concluding remarks
    if (index == -1)
        cout << value << " not found in the array." << endl;
    else
        cout << value << " found in the array at index " << index << endl;

    //Delete the array
    delete[] A;

    system("Pause");
    return 0;
}
```

Sequential Search (Cont.)

- Algorithm description
  - We compare the elements of the array with the search value; one by one starting from the first element and going up to the last
  - If we find an element equal to the search value, then we stop searching and return the index
  - If none of the elements of the array is equal to the search value; then we return -1, to mean the search value was not found in the array

Sequential Search Critical Operation

- Given an algorithm, it is customary to identify a critical operation of the algorithm in order to **count** how many times that critical operation will be executed for a given input size
- The critical operation is defined as the operation that determines as to when the algorithm will stop execution
- The count of the critical operation will give the **time** needed for the algorithm to do the required computation
- In the case of the Sequential Search algorithm, we see that the algorithm will stop execution when the if **(A[i] == searchValue)** is evaluated to true
- Therefore the critical operation is this **if** statement
- Also, the number of times the critical operation will be executed will depend on the actual input data (it may execute a few times for some input data and more/less than that for a different data)
- For this reason, we will be interested to count how many times the critical operation is executed in the **worst case** scenario
- The analysis of an algorithm based on the worst case scenario of the count is known as worst case scenario analysis

Sequential Search and Big-O Notation

- The question now is, assuming the array has **n** elements how many times does the algorithm perform the critical operation

  if (A[i] == searchValue)

  on a worst case scenario?
- It is easy to see that the algorithm will perform **n** comparisons in the worst case scenario, where **n** is the length of the array!
- Therefore, we say sequential search has a worst case running time of **n** times i.e. linear
- Put differently, we say the order of growth of the worst case running time is **linear** function i.e., **O(n)**

Sequential Search and Big-O Notation

- Let **f(n)** be the worst case running time for sequential search. That is f is the function of the running time and the array has **n** elements
- Then we say the order of growth of **f(n)** is linear
- Formally, we write it as

$$f(n) = O(n)$$

Mathematical Proof

- While it was easy to see the order of the running time for sequential search, let us put it in a formal mathematical form so that to help us in the subsequent sections
- We start with the case **n**=1, i.e. when the array has only one element
- In that case, we see that the algorithm does exactly one comparison operation. Therefore

$$f(1) = 1$$

Mathematical Proof (Cont.)

- How about when the array has **n** elements?
- Well, the algorithm will first compare the first element of the array with the value; hence one comparison is performed so far
- Next, assuming the value is not equal to the first element; the algorithm has to search again in the remaining **n-1** elements
- Since the search in the **n-1** elements will be using the exact same procedure, we see that...

Mathematical Proof (Cont.)

$$f(n) = 1 + f(n-1)$$

- This is the formula for the worst case running time for the sequential search!
- Importantly, the function is defined implicitly (not explicitly)! Such formulae are called Recurrence Relations!
- What is **f(n-1)**? Well using the same relationship, we must have

$$f(n-1) = 1 + f(n-2)$$

- Similarly,

$$f(n-2) = 1 + f(n-3)$$

and so on so forth

Mathematical Proof (Cont.)

- Now, we ask ourselves, can we solve the recurrence relationship in order to get an explicit (closed form) solution for **f(n)** that only involves **n**
- Yes! We can...
- The following simple proof, using telescoping method, shows that **f(n) = n** and therefore we say **f(n) = O(n)**

Mathematical Proof (Cont.)

f(n) = 1 + f(n-1)
f(n-1) = 1 + f(n-2)
f(n-2) = 1 + f(n-3)
⋮
f(2) = 1 + f(1)
f(1) = 1

Sum of Left Hand Side = f(n) + f(n-1) + f(n-2)+ ... + f(2) + f(1)

Sum of Right Hand Side = [1 + f(n-1)] + [1 + f(n-2)] + [1 + f(n-3)] + ... + [1 + f(1)] + 1

= [f(n-1) + f(n-2) + ... + f(2) + f(1)] +

1 + 1 + ... + 1 + 1] + 1

= [f(n-1) + f(n-2) + ... + f(2) + f(1)] + [n-1] + 1

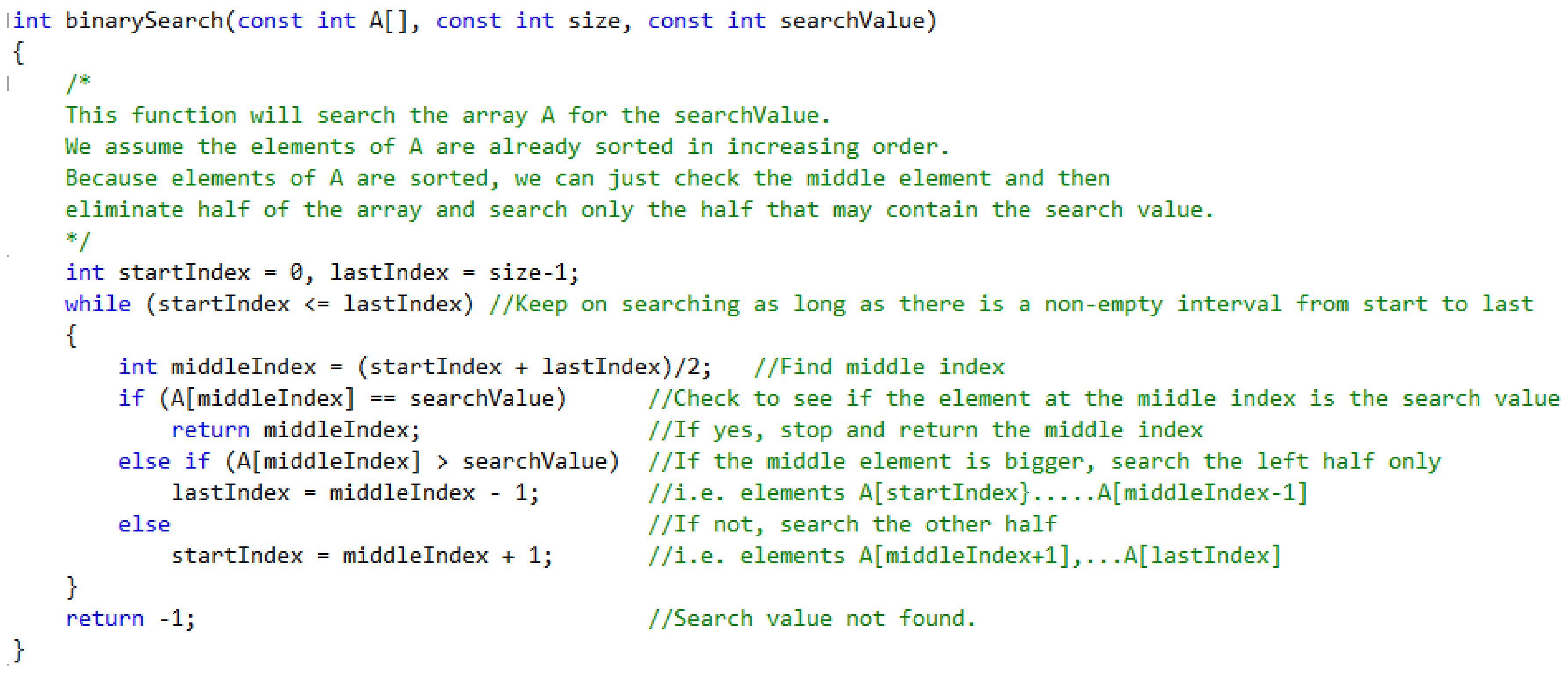
Equating the Left and the Right Hand Side equations, we get

f(n) + f(n-1) + f(n-2)+ ... + f(2) + f(1) = f(n-1) + f(n-2) + ... + f(2) + f(1) + n

Cancelling like terms then gives: **f(n) = n**

Binary Search

- Search a sorted array for a value by checking the middle element and then eliminating half of the array for the next search
- The following code demonstrates the idea:



```
int binarySearch(const int A[], const int size, const int searchValue)
{
    /*
    This function will search the array A for the searchValue.
    We assume the elements of A are already sorted in increasing order.
    Because elements of A are sorted, we can just check the middle element and then
    eliminate half of the array and search only the half that may contain the search value.
    */
    int startIndex = 0, lastIndex = size-1;
    while (startIndex <= lastIndex) //Keep on searching as long as there is a non-empty interval from start to last
    {
        int middleIndex = (startIndex + lastIndex)/2;   //Find middle index
        if (A[middleIndex] == searchValue)      //Check to see if the element at the miidle index is the search value
            return middleIndex;                 //If yes, stop and return the middle index
        else if (A[middleIndex] > searchValue)  //If the middle element is bigger, search the left half only
            lastIndex = middleIndex - 1;        //i.e. elements A[startIndex}.....A[middleIndex-1]
        else                                    //If not, search the other half
            startIndex = middleIndex + 1;       //i.e. elements A[middleIndex+1],...A[lastIndex]
    }
    return -1;                                  //Search value not found.
}
```

Binary Search

```
int main()
{
    const unsigned int size = 300000000;
    int *A = new int[size];
    srand(time(0));

    //Fill the array
    A[0] = 5;
    for (int i = 1; i < size; i++)
        A[i] = A[i-1] + rand() % 2;


    //Perform binary Search
    int value = A[0] - 1;   //Select a number not found in the array
    cout << "Press the ENTER key to Start binary search..." << endl;
    getchar();  //This will pause your program until you press the ENTER key
    int index = binarySearch(A, size, value);
    cout << "Finished binary search..." << endl;

    //Print concluding remarks
    if (index == -1)
        cout << value << " not found in the array." << endl;
    else
        cout << value << " found in the array at index " << index << endl;

    //Delete the array
    delete[] A;

    system("Pause");
    return 0;
}
```

Binary Search Algorithm Running Time

- It is easy to see that **f(n) = 1 + f(n/2)**. Then

```
f(n)   = 1 + f(n/2)
f(n/2) = 1 + f(n/4)
f(n/4) = 1 + f(n/8)
⋮
f(2)   = 1 + f(1)
f(1)   = 1
```

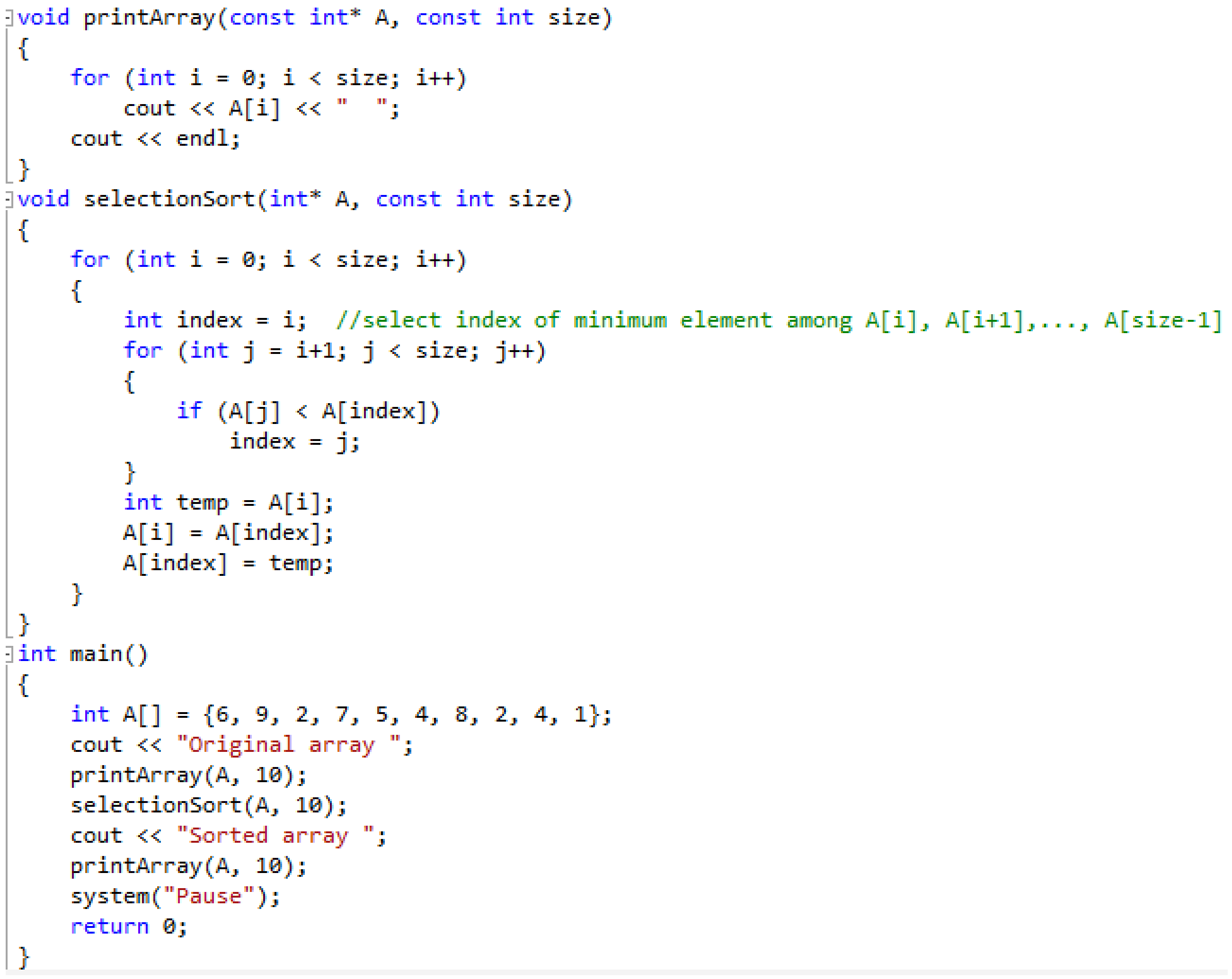
```
Sum of Left Hand Side  = f(n) + f(n/2) + f(n/4) + ... + f(1)
Sum of Right Hand Side = [1 + f(n/2)] + [1 + f(n/4)] + [1 + f(n/8)] + ... + [1 + f(1)] + 1
                       = [f(n/2) + f(n/4) + f(n/8) + ... + f(1)] +
                         [   1    +    1   +    1   + ... +  1  ] + 1
                       = [f(n/2) + f(n/4) + f(n/8) + ... + f(1)] + [log₂ n] + 1
```

Therefore, equating the left hand side and the right hand side and cancelling like terms results to

$$f(n) = 1 + \log_2 n \ldots \textit{which is} \ldots O(\log n) \ldots \textit{that is} \ldots \textit{Logarithmic}$$

Sorting C++ Arrays
Selection Sort Algorithm

- The following function sorts a C++ array using **selection sort algorithm**



```cpp
void printArray(const int* A, const int size)
{
    for (int i = 0; i < size; i++)
        cout << A[i] << "  ";
    cout << endl;
}
void selectionSort(int* A, const int size)
{
    for (int i = 0; i < size; i++)
    {
        int index = i;  //select index of minimum element among A[i], A[i+1],..., A[size-1]
        for (int j = i+1; j < size; j++)
        {
            if (A[j] < A[index])
                index = j;
        }
        int temp = A[i];
        A[i] = A[index];
        A[index] = temp;
    }
}
int main()
{
    int A[] = {6, 9, 2, 7, 5, 4, 8, 2, 4, 1};
    cout << "Original array ";
    printArray(A, 10);
    selectionSort(A, 10);
    cout << "Sorted array ";
    printArray(A, 10);
    system("Pause");
    return 0;
}
```

Selection Sort Algorithm Running Time

- In order to select the **k-th** smallest element, we see that we have to perform the comparison operation **(n-k)** times in the worst case
- Therefore, it is easy to see that the selection sort algorithm will require

  n-1 operations to select the smallest element

  n-2 operatiions to select the second smallest element

  n-3 operations to insert the third smallest element

  ⋮

  1 operation to select **(n-1)-th** smallest element

Selection Sort Algorithm Running Time

- Therefore

$$f(n) = n-1 + n-2 + n-3 + \ldots + 2 + 1 = n(n-1)/2$$

- That is

$$f(n) = 0.5n^2 - 0.5n$$

- Once again, in Big-O notation, we are concerned with the behaviour or order of the function **f(n)** as **n** becomes bigger and bigger
- Therefore we see that the f(n) for selection sort is simply quadratic
- We write it as $f(n) = O(n^2)$ **i.e. Quadratic**

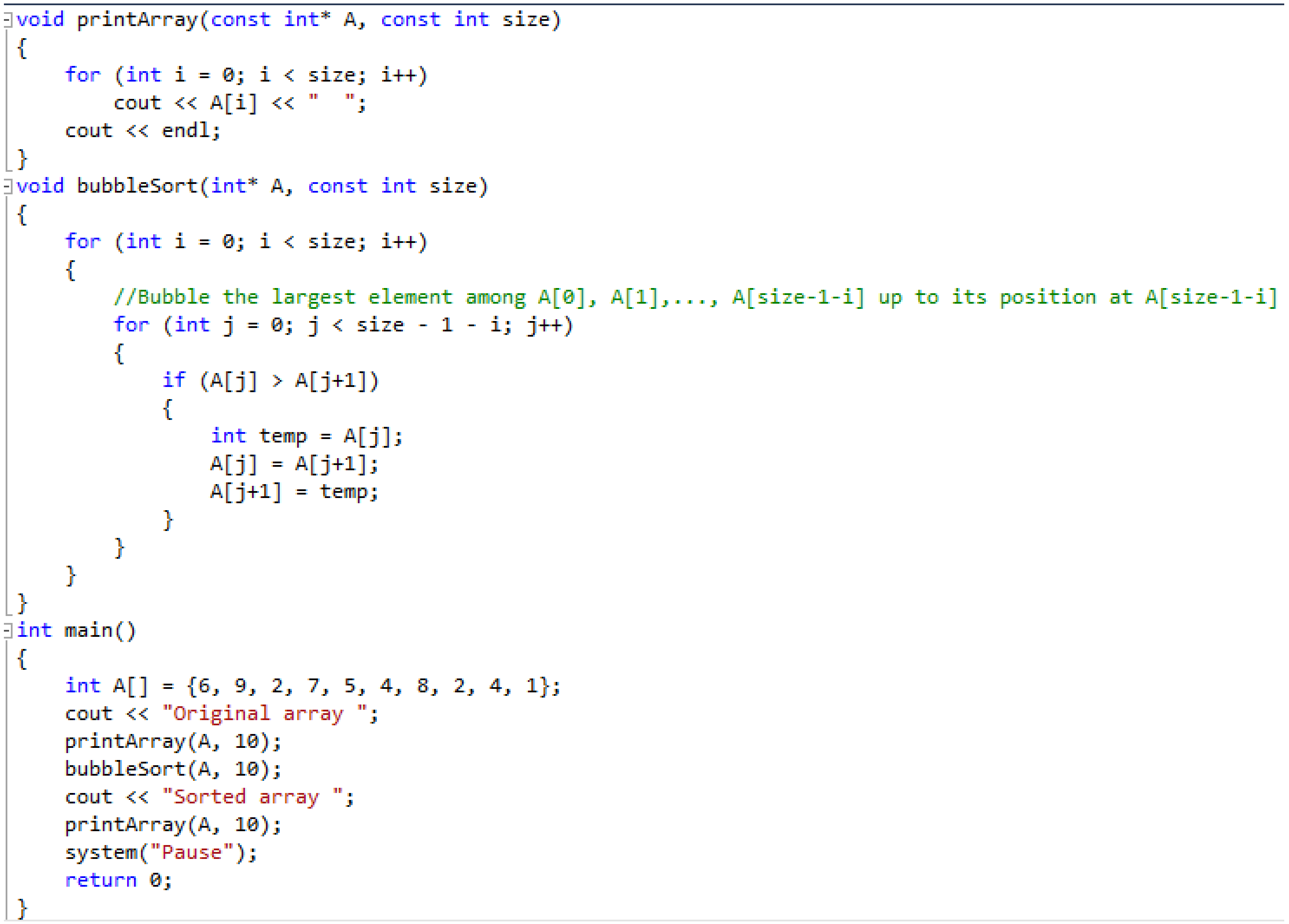
Sorting C++ Arrays Bubble Sort Algorithm

- In bubble sort, as its name implies, the idea is to bubble up elements upwards until all elements are sorted
- Consider the procedure for putting the largest element of an array to its right place:
  - Compare **A[0]** and **A[1]** and re-arrange them so that **A[1]** >= **A[0]**
  - Then compare **A[1]** and **A[2]** and re-arrange them so that **A[2]** >= **A[1]**
  - Then do the same for **A[2]** and **A[3]**
  - **⋮**
  - Finally do the same for **A[n-2]** and **A[n-1]**. Remember the last element has index **(n-1)**

Bubble Sort (Cont.)

- In one pass (that is **n-1** comparisons and swappings), the largest element will be bubbled up to its right position at **A[n-1]**
- In the next pass, the bubbling procedure is performed starting with **A[0]** and **A[1]** and going all the way to **A[n-3]** and **A[n-2]** to put the second largest element to its right place at **A[n-2]**
- And so on so forth... until all elements are sorted
- The following code does the job!

Bubble Sort (Cont.)



```
void printArray(const int* A, const int size)
{
    for (int i = 0; i < size; i++)
        cout << A[i] << "  ";
    cout << endl;
}
void bubbleSort(int* A, const int size)
{
    for (int i = 0; i < size; i++)
    {
        //Bubble the largest element among A[0], A[1],..., A[size-1-i] up to its position at A[size-1-i]
        for (int j = 0; j < size - 1 - i; j++)
        {
            if (A[j] > A[j+1])
            {
                int temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
int main()
{
    int A[] = {6, 9, 2, 7, 5, 4, 8, 2, 4, 1};
    cout << "Original array ";
    printArray(A, 10);
    bubbleSort(A, 10);
    cout << "Sorted array ";
    printArray(A, 10);
    system("Pause");
    return 0;
}
```

Bubble Sort Algorithm Running Time

- It is easy to see that the upper loop will iterate for **n** times
- The inner loop will iterate

  n-1 times when **i = 0**

  n-2 times when **i = 1**

  n-3 times when **i = 2**

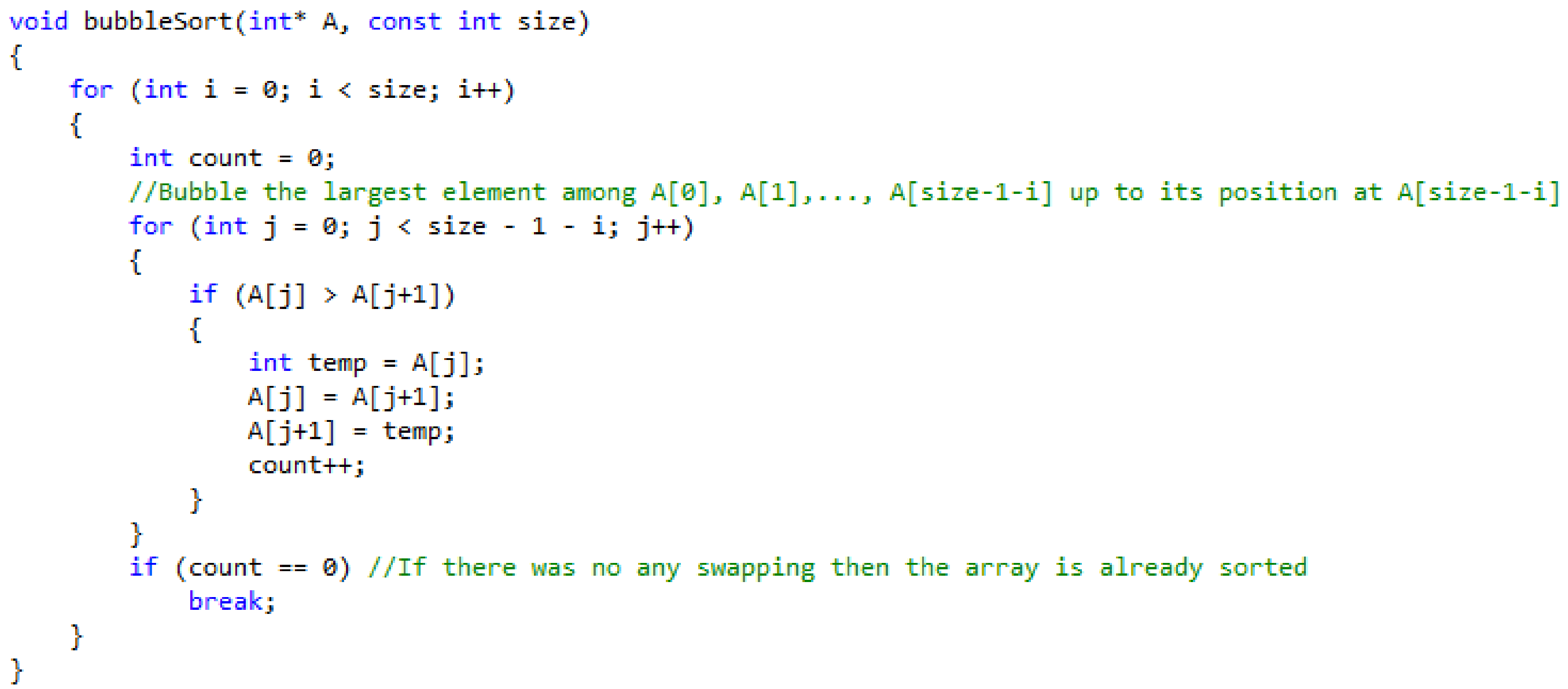
  ⋮

  1 time when **i = n-2**
- Summing up, we get **f(n) = n(n-1)/2** or **$f(n)=O(n^2)$**

Modified Bubble Sort Algorithm

- The bubble sort can be modified a little bit for practical efficiency purposes although its worst case running time f(n) will not change
- Consider the following implementation of the modified bubble sort and analyze it to see how it will on practical cases will perform better than the original implementation of the bubble sort algorithm

Modified Bubble Sort Algorithm



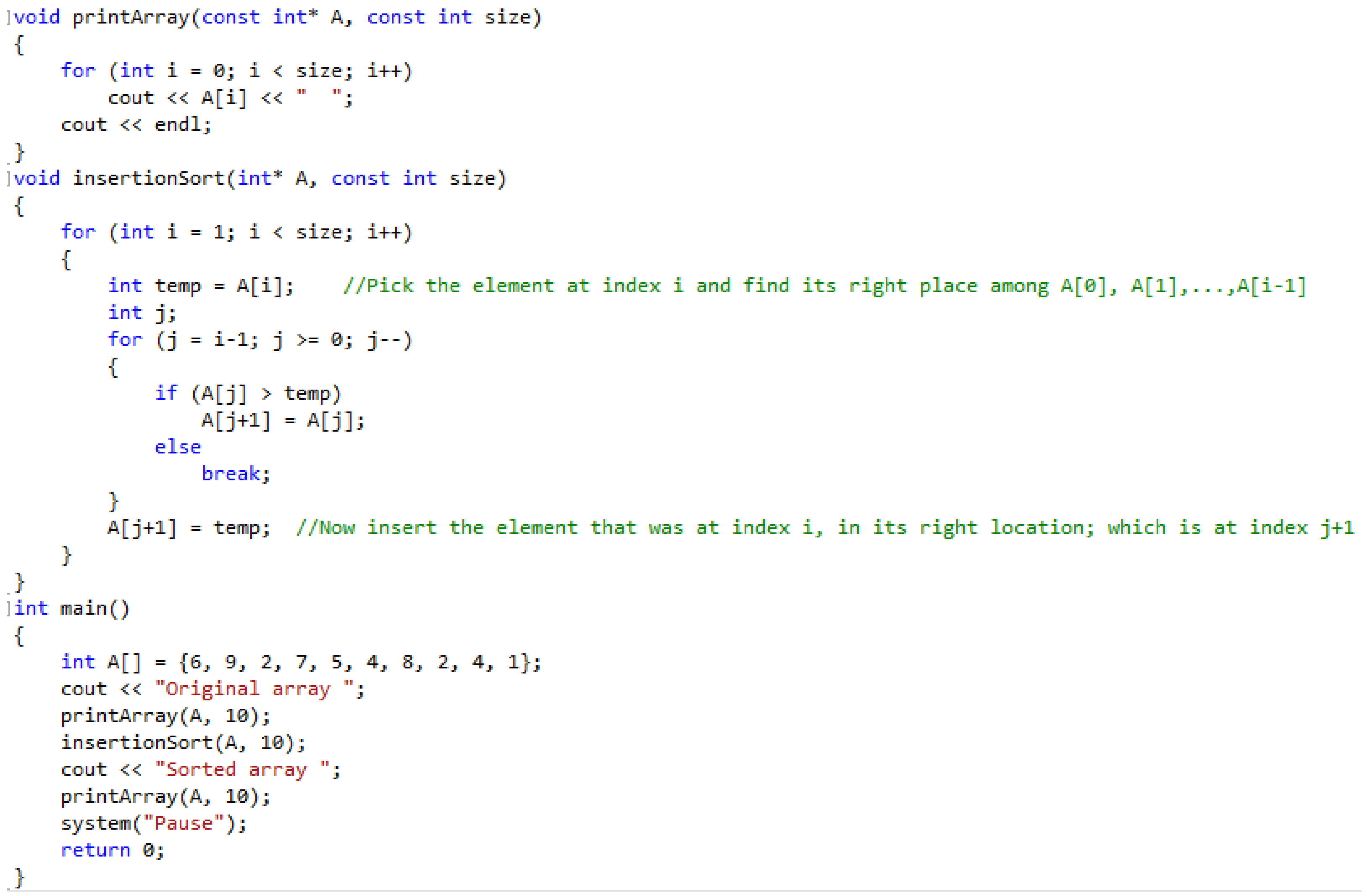
```
void bubbleSort(int* A, const int size)
{
    for (int i = 0; i < size; i++)
    {
        int count = 0;
        //Bubble the largest element among A[0], A[1],..., A[size-1-i] up to its position at A[size-1-i]
        for (int j = 0; j < size - 1 - i; j++)
        {
            if (A[j] > A[j+1])
            {
                int temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
                count++;
            }
        }
        if (count == 0) //If there was no any swapping then the array is already sorted
            break;
    }
}
```

Practice Question:- Give an example of an array for which this modified bubble sort algorithm will perform better than the original implementation of the bubble sort algoritm.

Insertion Sort

- Yet another algorithm to sort arrays is the insertion sort
- In this algorithm, assuming we are sorting in increasing order,
  - We first assume the first element is already in its right place
  - Then pick the second element and insert it either before or after the first element depending if it is less than or greater than the first element
  - Next, pick the third element and insert it either before the first element or between the first and second elements or after the second element
  - Generally, pick element at index **k** and insert it in its right place between indexes **0** and **k**
  - After inserting the last element, the array will be sorted
- See the following implementation

Insertion Sort (cont.)



```cpp
void printArray(const int* A, const int size)
{
    for (int i = 0; i < size; i++)
        cout << A[i] << "  ";
    cout << endl;
}
void insertionSort(int* A, const int size)
{
    for (int i = 1; i < size; i++)
    {
        int temp = A[i];    //Pick the element at index i and find its right place among A[0], A[1],...,A[i-1]
        int j;
        for (j = i-1; j >= 0; j--)
        {
            if (A[j] > temp)
                A[j+1] = A[j];
            else
                break;
        }
        A[j+1] = temp;  //Now insert the element that was at index i, in its right location; which is at index j+1
    }
}
int main()
{
    int A[] = {6, 9, 2, 7, 5, 4, 8, 2, 4, 1};
    cout << "Original array ";
    printArray(A, 10);
    insertionSort(A, 10);
    cout << "Sorted array ";
    printArray(A, 10);
    system("Pause");
    return 0;
}
```

Insertion Sort Algorithm Running Time

- It is easy to see that the upper loop will iterate n-1 times; where n is the number of elements in the array
- In the first pass, the critical operation (that is the **if-statement**) will executed once
- In the second pass, it will be executed twice in the worst case scenario
- In the third pass trice, etc
- In total, it will be executed 1+2+3+…+(n-1) which shows

$$\mathbf{f(n) = 0.5n^2-0.5n}$$

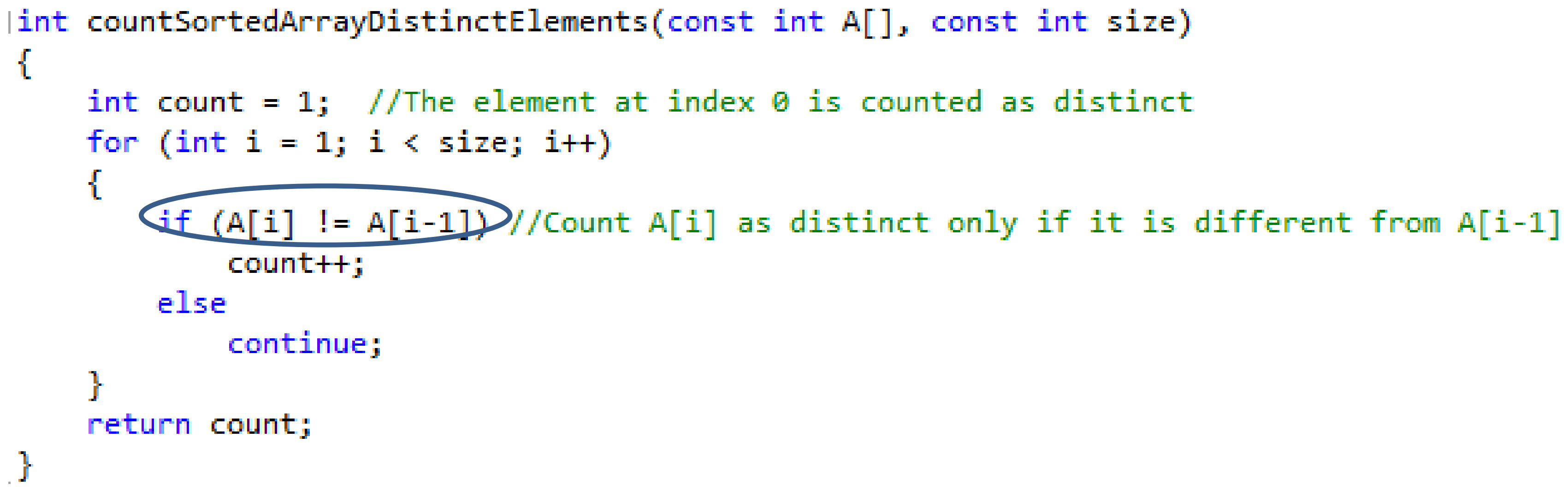
- Therefore we conclude that the insertion sort is $\mathbf{O(n^2)}$

Counting Critical Operations

- In order to count critical operations of any given procedure (that is an implementation of an algorithm); we need to carefully count the operation of interest (example “if statement”) inside loops and other code segments.
- In particular, we look for some mathematical relationship between our count and the size of the object under consideration
- The following example demonstrates the idea

Counting Critical Operations

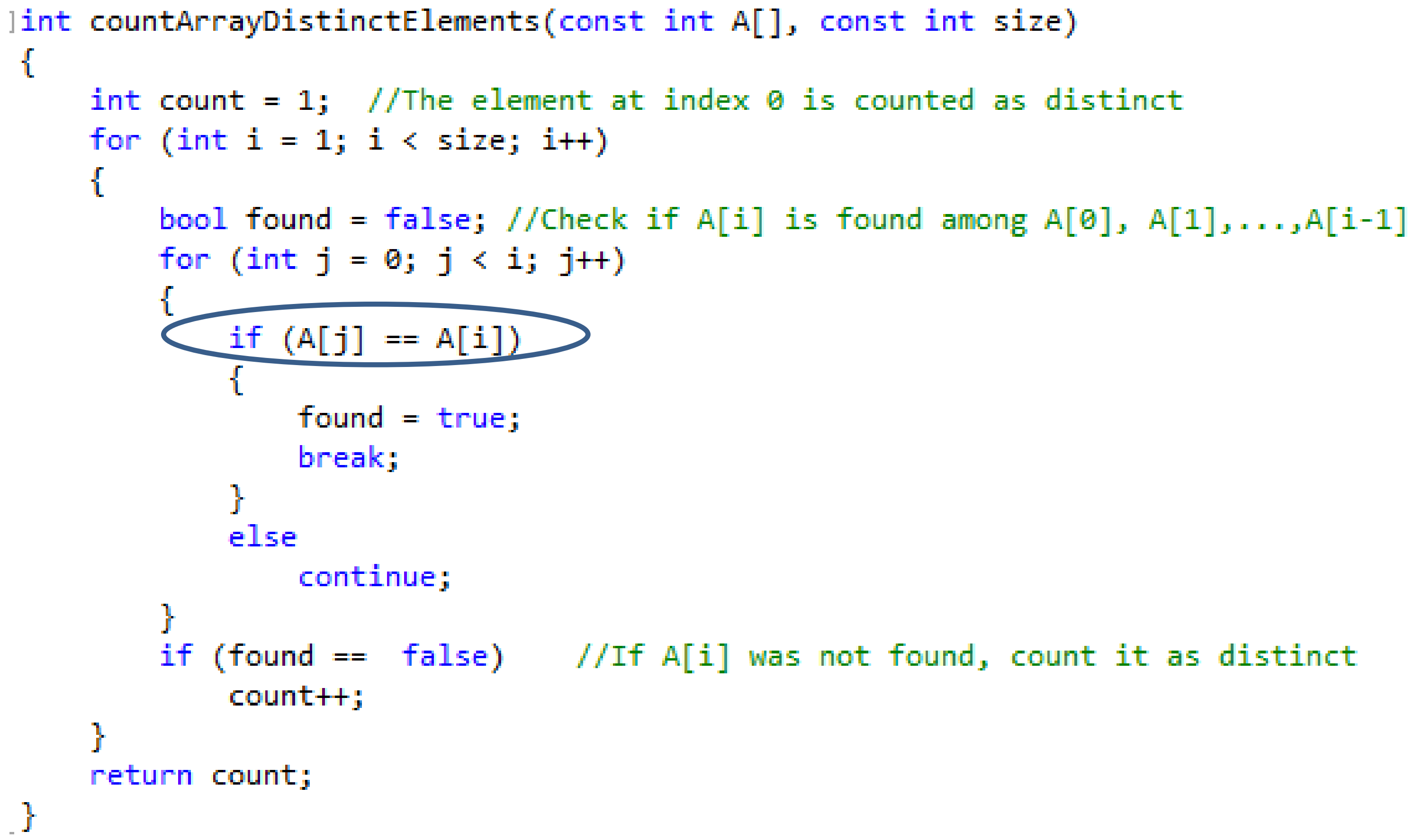
- **Example:-** Given the following procedure to count the distinct elements of a sorted array; how many times does the critical operation, shown inside an elliptical curve, get executed?



```
int countSortedArrayDistinctElements(const int A[], const int size)
{
    int count = 1;  //The element at index 0 is counted as distinct
    for (int i = 1; i < size; i++)
    {
        if (A[i] != A[i-1]) //Count A[i] as distinct only if it is different from A[i-1]
            count++;
        else
            continue;
    }
    return count;
}
```

Counting Critical Operations

- **Example:- Now, consider a similar procedure but this time** to count the distinct elements of any array; how many times does the critical operation, shown inside an elliptical curve, get executed?



```
int countArrayDistinctElements(const int A[], const int size)
{
    int count = 1;  //The element at index 0 is counted as distinct
    for (int i = 1; i < size; i++)
    {
        bool found = false; //Check if A[i] is found among A[0], A[1],...,A[i-1]
        for (int j = 0; j < i; j++)
        {
            if (A[j] == A[i])
            {
                found = true;
                break;
            }
            else
                continue;
        }
        if (found ==  false)    //If A[i] was not found, count it as distinct
            count++;
    }
    return count;
}
```

Complexity of Algorithms Summary

Algorithm	Sequenti al Search	Binary Search	Selection Sort	Bubble Sort	Insertion Sort	Counting Distinct (sorted array)	Counting Distinct (any array)
Complexity	O(n)	O(log n)	O(n^2)	O(n^2)	O(n^2)	O(n)	O(n^2)
	Linear	Logarithmic	Quadratic	Quadratic	Quadratic	Linear	Quadratic