ANALYSIS OF SORTING AND SEARCHING ALGORITHMS

LECTURE THREE

BRUTE FORCE

- A brute force algorithm is a simple, comprehensive search strategy that systematically explores every option until a problem's answer is discovered.
- It's a generic approach to problem-solving that's employed when the issue is small enough to make an in-depth investigation possible.
- However, because of their high temporal complexity, brute force techniques are inefficient for large-scale issues.

FEATURES OF THE BRUTE FORCE ALGORITHM

- It is an intuitive, direct, and straightforward technique of problem-solving in which all the possible ways or all the possible solutions to a given problem are enumerated.
- Many problems are solved in day-to-day life using the brute force strategy, for example, exploring all the paths to a nearby market to find the minimum shortest path.
- Arranging the books in a rack using all the possibilities to optimize the rack spaces, etc.
- Daily life activities use a brute force nature, even though optimal algorithms are also possible.

PROS AND CONS OF BRUTE FORCE ALGORITHM

Pros:

- The brute force approach is a guaranteed way to find the correct solution by listing all the possible candidate solutions for the problem.
- It is a generic method and not limited to any specific domain of problems.
- The brute force method is ideal for solving small and simpler problems.
- It is known for its simplicity and can serve as a comparison benchmark

PROS AND CONS OF BRUTE FORCE ALGORITHM

Cons:

- The brute force approach is inefficient. For real-time problems, algorithm analysis often goes above the O(N!) order of growth.
- This method relies more on compromising the power of a computer system for solving a problem than on a good algorithm design.
- Brute force algorithms are slow.
- Brute force algorithms are not constructive or creative compared to algorithms that are constructed using some other design paradigms.

SORTING ALGORITHMS

- Sorting refers to arranging data in a particular format.
- Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.
- The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats.

TYPES OF SORTING ALGORITHMS

- Sorting algorithms can be broadly classified into two categories:
    i. Comparison-based and
    ii. Non-comparison-based algorithms

TYPES OF SORTING ALGORITHMS

- **Comparison-based algorithms** compare elements in the input to determine their order, and thus their time complexity is lower bounded by the number of comparisons needed to correctly order the input.
- All the sorting algorithms mentioned in this blog post, including Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Heap Sort, are comparison-based algorithms.

TYPES OF SORTING ALGORITHMS

- **Non-comparison-based algorithms**, on the other hand, do not rely on element comparisons to determine their order.
- These algorithms are often more specialized and are used in specific situations where the input has certain properties.
- For example, counting sort is a non-comparison-based algorithm that can be used to sort inputs consisting of integers with a small range.

TYPES OF SORTING ALGORITHMS

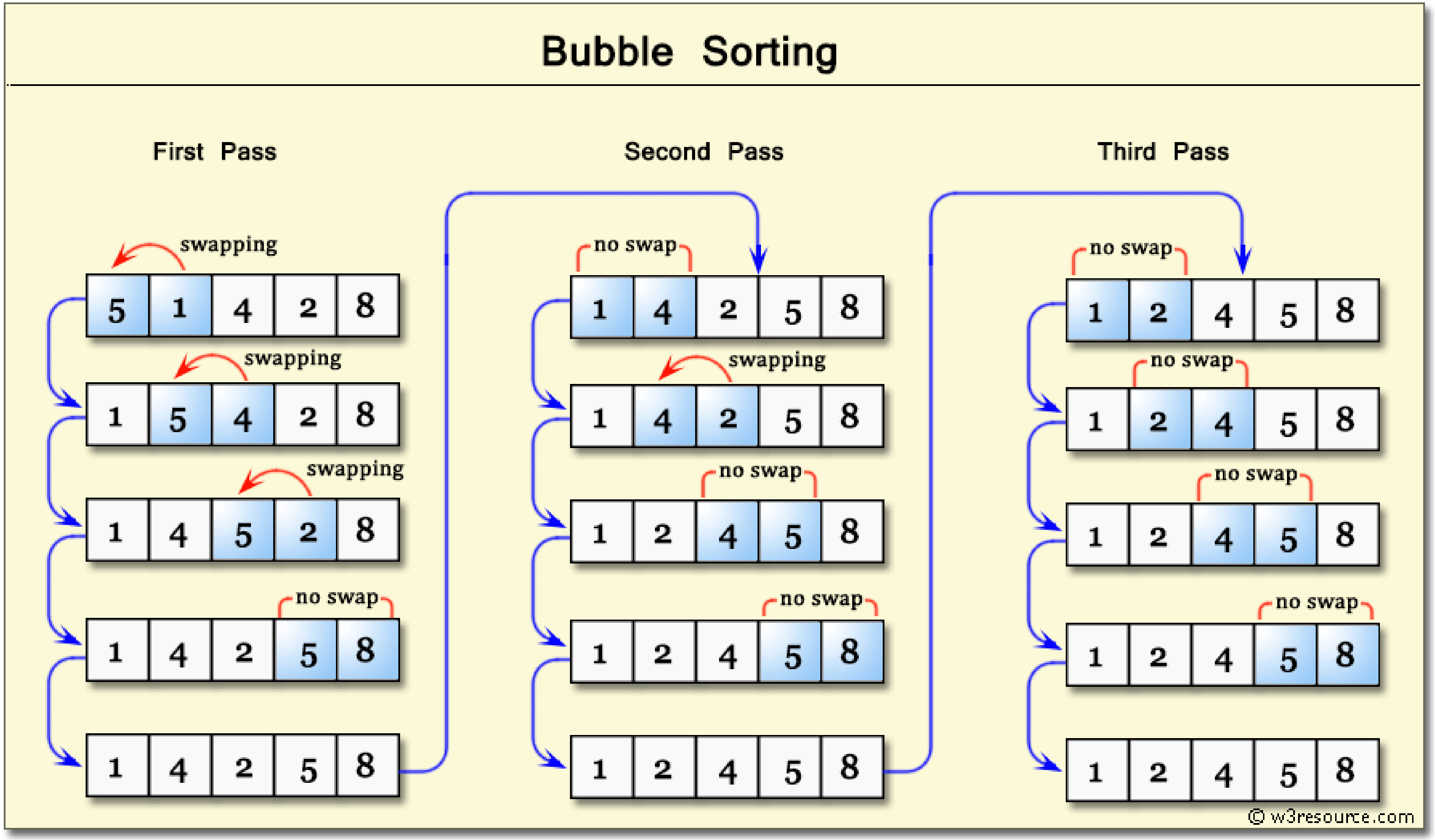
- Another way to classify sorting algorithms is based on their stability.
- A sorting algorithm is said to be stable if it preserves the relative order of equal elements in the input.
- Merge Sort and Insertion Sort are stable algorithms, while Quick Sort and Heap Sort are unstable algorithms.

SORTING TECHNIQUES

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort

BUBBLE SORT

- Bubble sort is a simple sorting algorithm.
- This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.
- This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where n is the number of items.

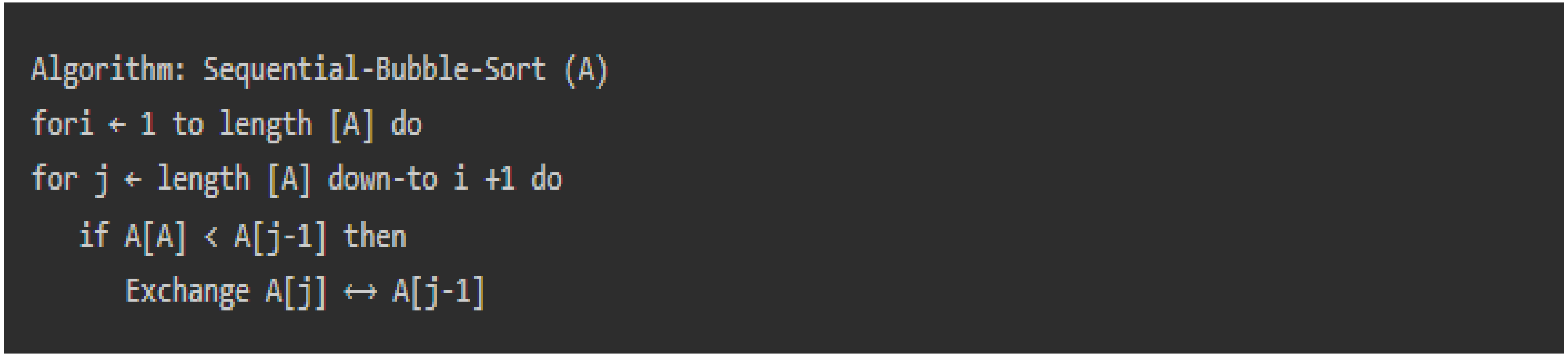

Bubble Sorting
First Pass
Second Pass
Third Pass
swapping
5 1 4 2 8
swapping
1 5 4 2 8
swapping
1 4 5 2 8
no swap
1 4 2 5 8
1 4 2 5 8
no swap
1 4 2 5 8
swapping
1 4 2 5 8
no swap
1 2 4 5 8
no swap
1 2 4 5 8
1 2 4 5 8
no swap
1 2 4 5 8
no swap
1 2 4 5 8
no swap
1 2 4 5 8
no swap
1 2 4 5 8
1 2 4 5 8
© w3resource.com

BUBBLE SORT ALGORITHM

- Bubble Sort is an elementary sorting algorithm, which works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.
- We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.
- Step 1 − Check if the first element in the input array is greater than the next element in the array.
- Step 2 − If it is greater, swap the two elements; otherwise move the pointer forward in the array.

BUBBLE SORT ALGORITHM

- Step 3 − Repeat Step 2 until we reach the end of the array.
- Step 4 − Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.
- Step 5 − The final output achieved is the sorted array.



```
Algorithm: Sequential-Bubble-Sort (A)
fori ← 1 to length [A] do
for j ← length [A] down-to i +1 do
   if A[A] < A[j-1] then
      Exchange A[j] ↔ A[j-1]
```

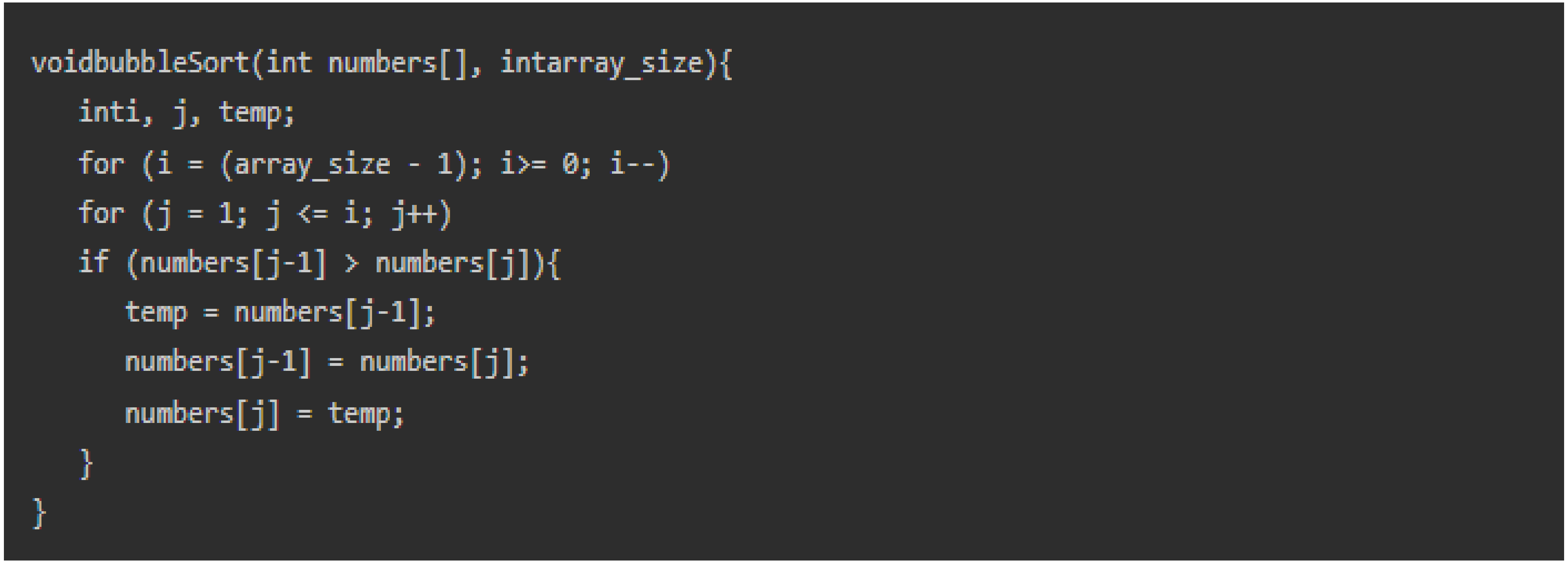
BUBBLE SORT ALGORITHM

Pseudocode

- We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.
- To ease-out the issue, we use one flag variable swapped which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

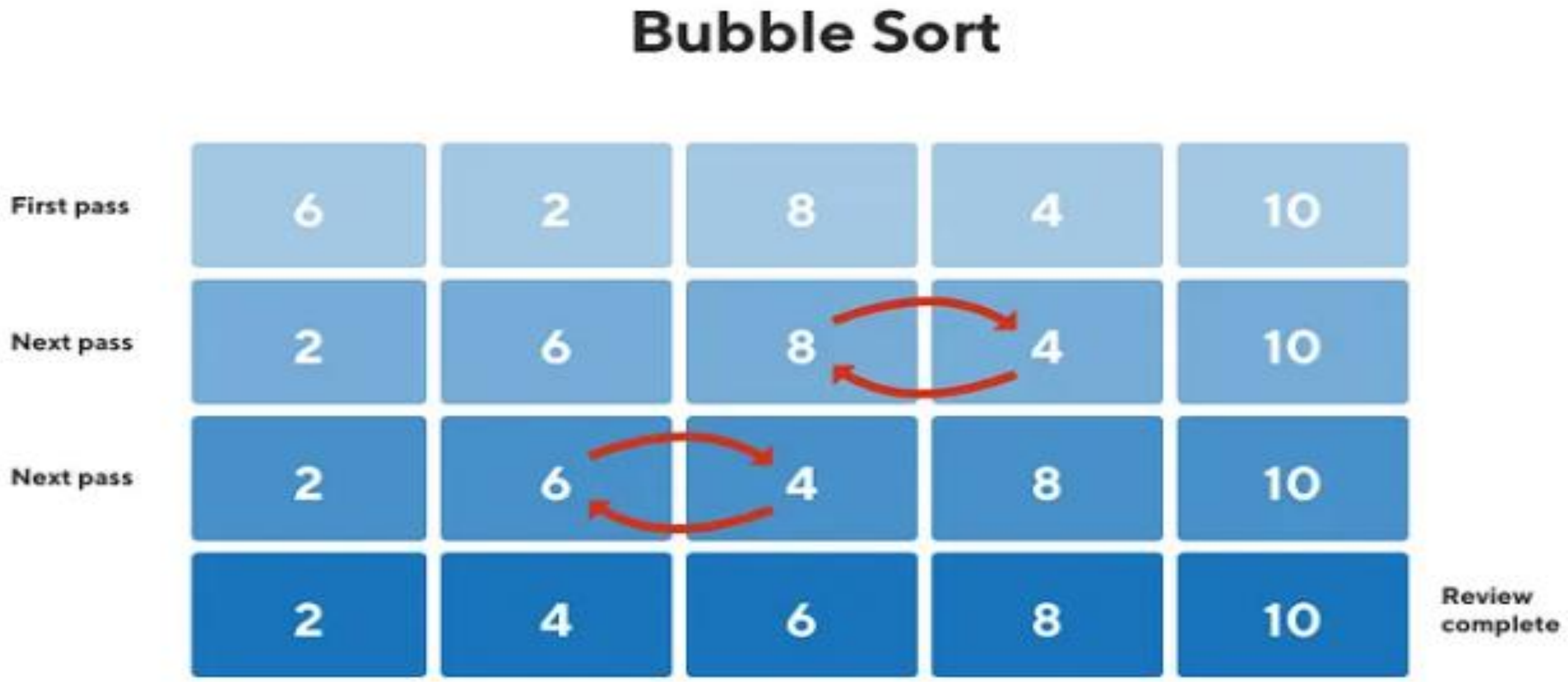
BUBBLE SORT ALGORITHM

Pseudocode of bubble sort algorithm can be written as follows



```
voidbubbleSort(int numbers[], intarray_size){
   inti, j, temp;
   for (i = (array_size - 1); i>= 0; i--)
   for (j = 1; j <= i; j++)
   if (numbers[j-1] > numbers[j]){
      temp = numbers[j-1];
      numbers[j-1] = numbers[j];
      numbers[j] = temp;
   }
}
```

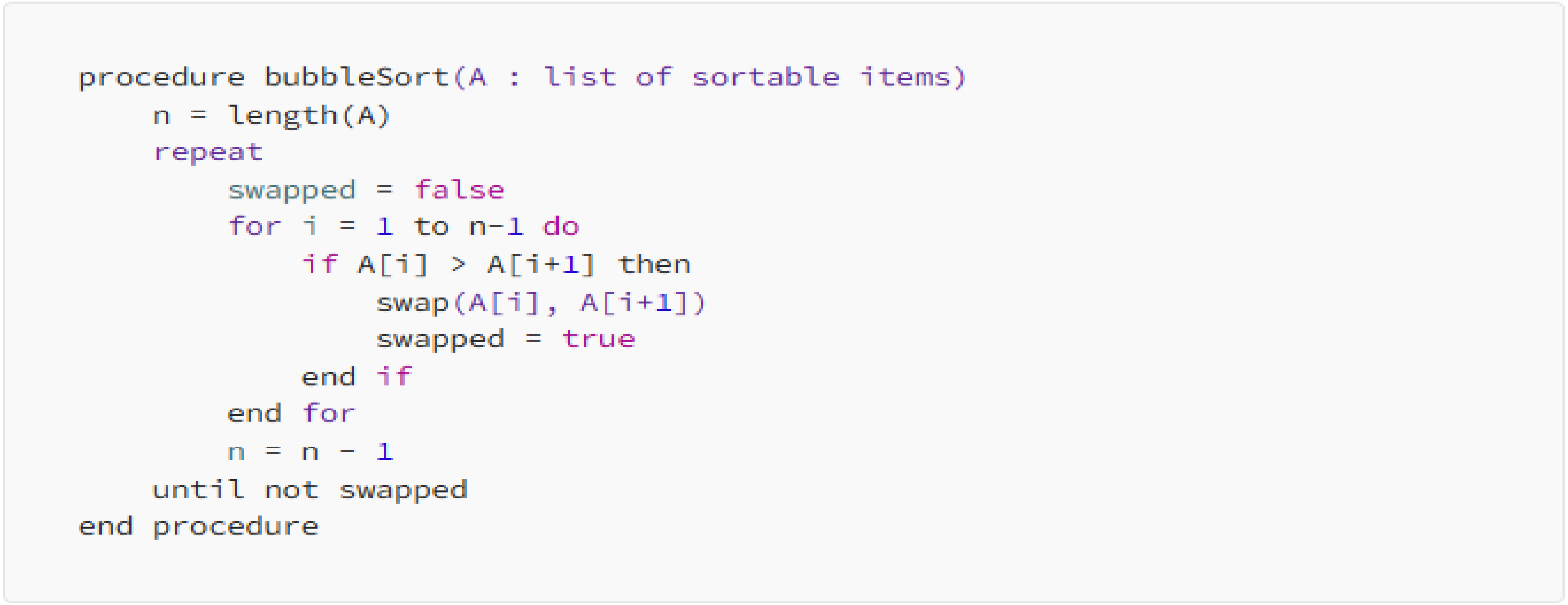
BUBBLE SORT ALGORITHM



First pass	6	2	8	4	10	
Next pass	2	6	8	4	10	
Next pass	2	6	4	8	10	
	2	4	6	8	10	Review complete

BUBBLE SORT ALGORITHM

here’s, pseudocode for the same.



```
procedure bubbleSort(A : list of sortable items)
    n = length(A)
    repeat
        swapped = false
        for i = 1 to n-1 do
            if A[i] > A[i+1] then
                swap(A[i], A[i+1])
                swapped = true
            end if
        end for
        n = n - 1
    until not swapped
end procedure
```

INSERTION SORT

- Insertion sort is a very simple method to sort numbers in an ascending or descending order.
- This method follows the incremental method. It can be compared with the technique how cards are sorted at the time of playing a game.
- This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted.
- For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted' in this sorted sub-list, has to find its appropriate place and then it has to be inserted there.

INSERTION SORT

- The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).
- This algorithm is not suitable for large data sets as its average and worst case complexity are of (n2), where n is the number of items.

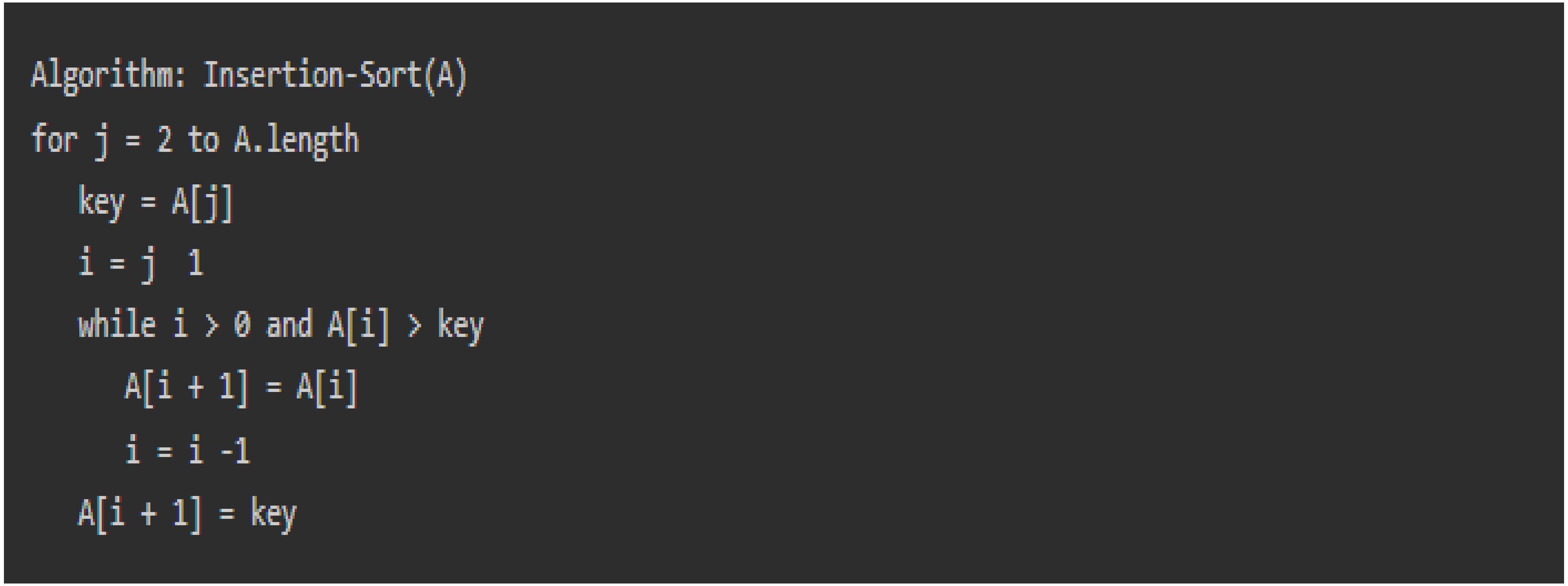
- Insertion algorithm: a simple sorting algorithm that builds a sorted list one element at a time by iteratively inserting each element into its correct position within the sorted portion of the list.

INSERTION SORT ALGORITHM

- Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.
- Step 1 − If it is the first element, it is already sorted. return 1;
- Step 2 − Pick next element
- Step 3 − Compare with all elements in the sorted sub-list
- Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
- Step 5 − Insert the value
- Step 6 − Repeat until list is sorted

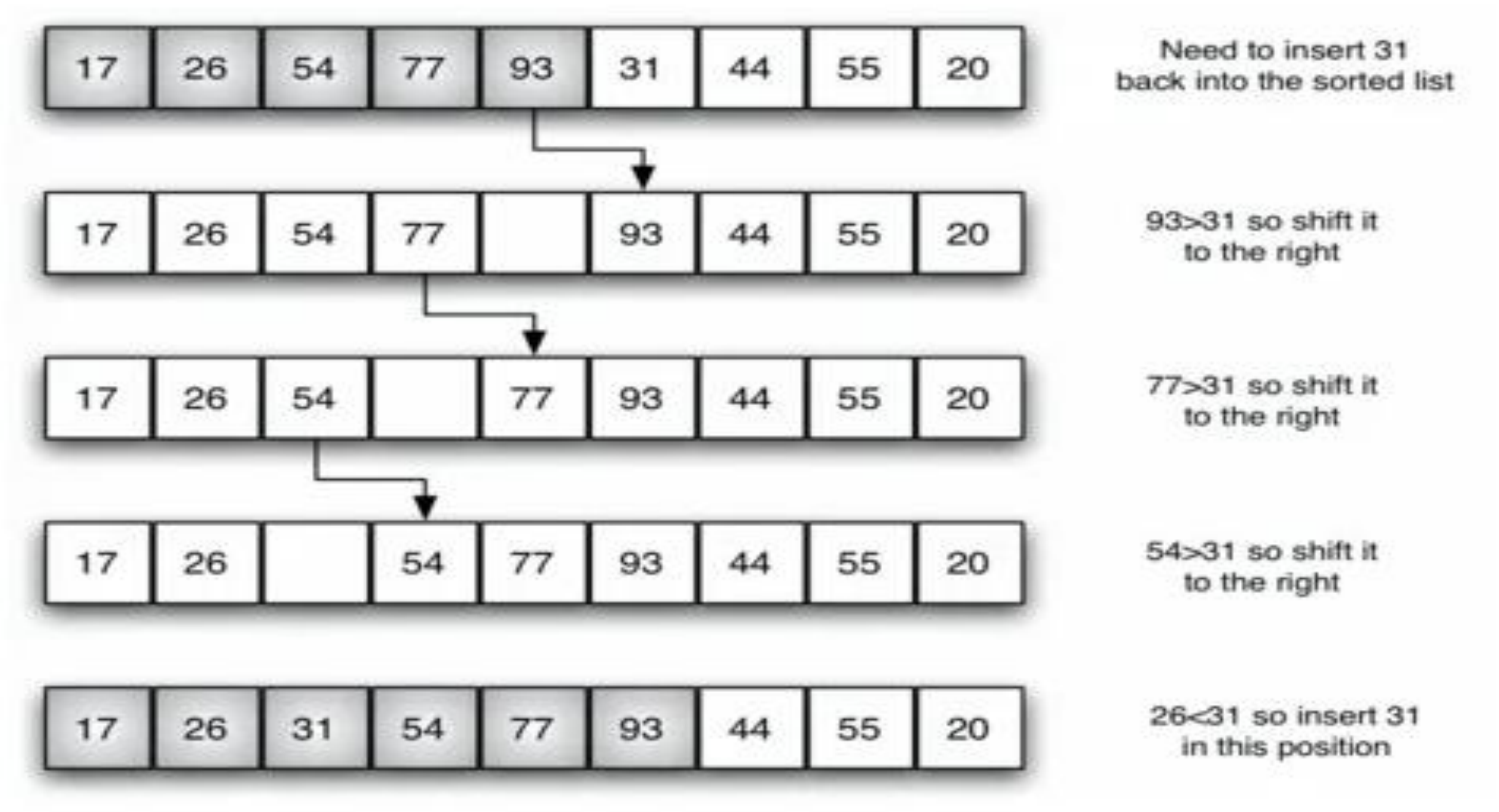
INSERTION SORT ALGORITHM

Pseudocode

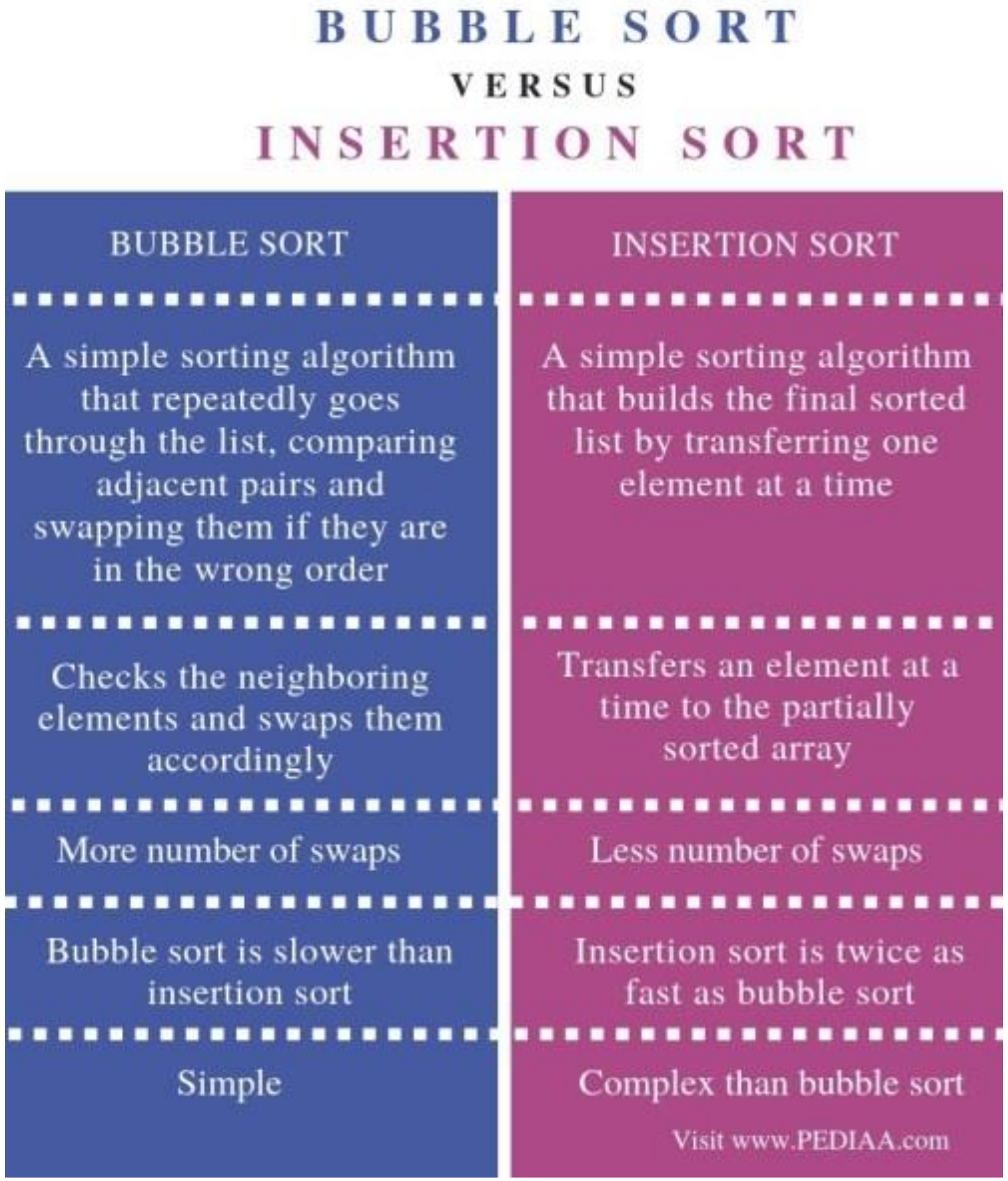


```
Algorithm: Insertion-Sort(A)
for j = 2 to A.length
   key = A[j]
   i = j  1
   while i > 0 and A[i] > key
      A[i + 1] = A[i]
      i = i -1
   A[i + 1] = key
```

INSERTION SORT ALGORITHM



17 26 54 77 93 31 44 55 20
Need to insert 31 back into the sorted list
17 26 54 77 93 44 55 20
93>31 so shift it to the right
17 26 54 77 93 44 55 20
77>31 so shift it to the right
17 26 54 77 93 44 55 20
54>31 so shift it to the right
17 26 31 54 77 93 44 55 20
26<31 so insert 31 in this position



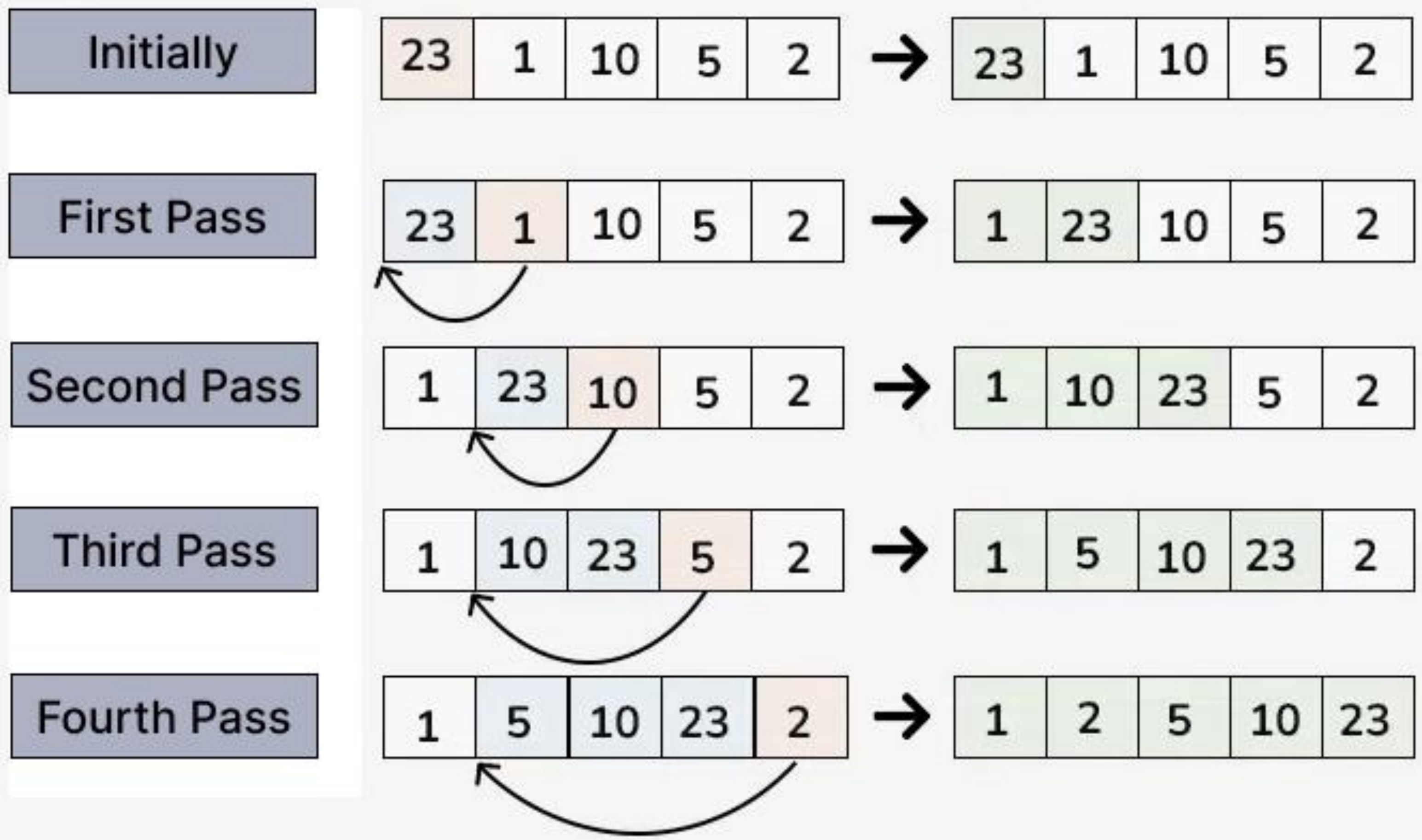
BUBBLE SORT

VERSUS

INSERTION SORT

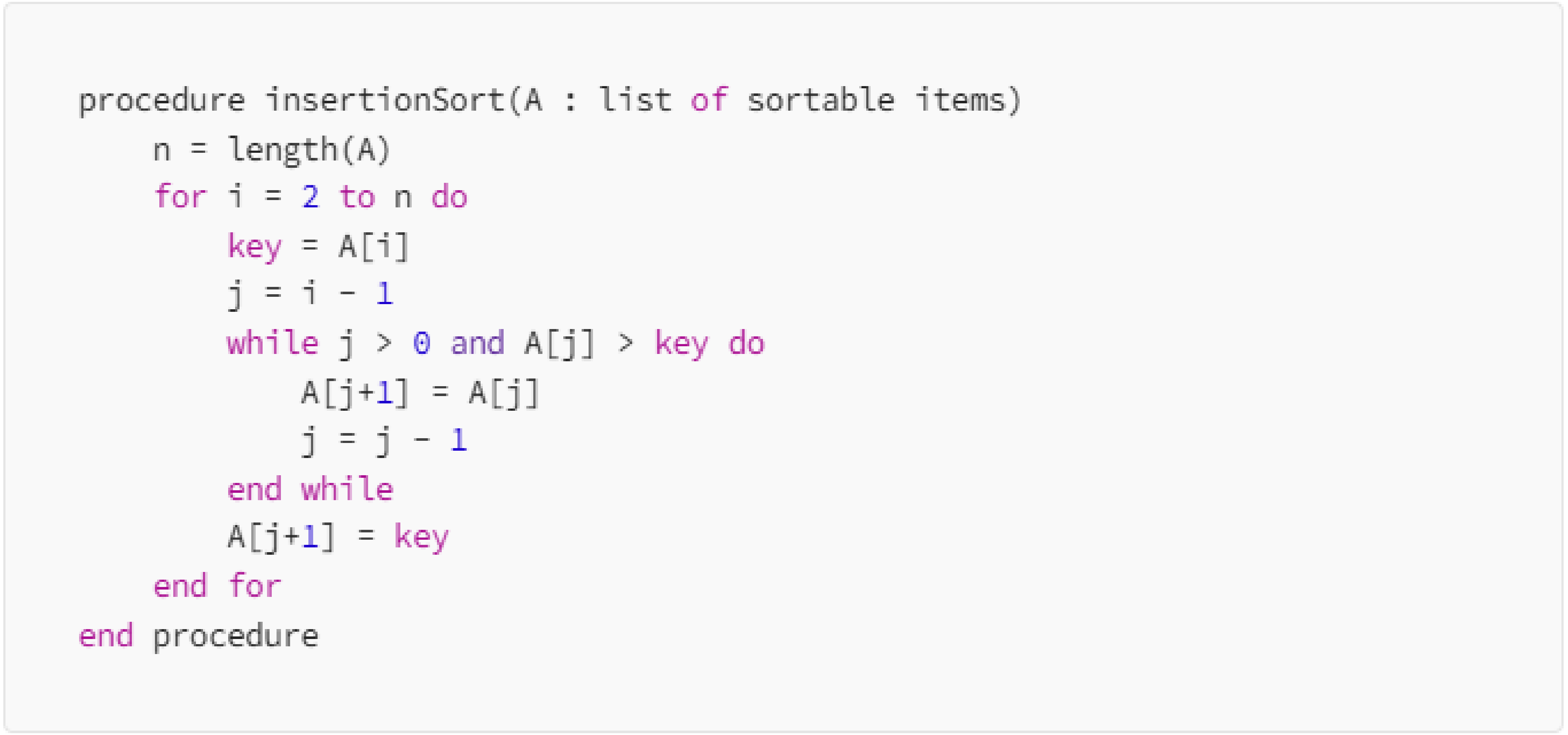
BUBBLE SORT	INSERTION SORT
A simple sorting algorithm that repeatedly goes through the list, comparing adjacent pairs and swapping them if they are in the wrong order	A simple sorting algorithm that builds the final sorted list by transferring one element at a time
Checks the neighboring elements and swaps them accordingly	Transfers an element at a time to the partially sorted array
More number of swaps	Less number of swaps
Bubble sort is slower than insertion sort	Insertion sort is twice as fast as bubble sort
Simple	Complex than bubble sort

Visit www.PEDIAA.com



Insertion Sort

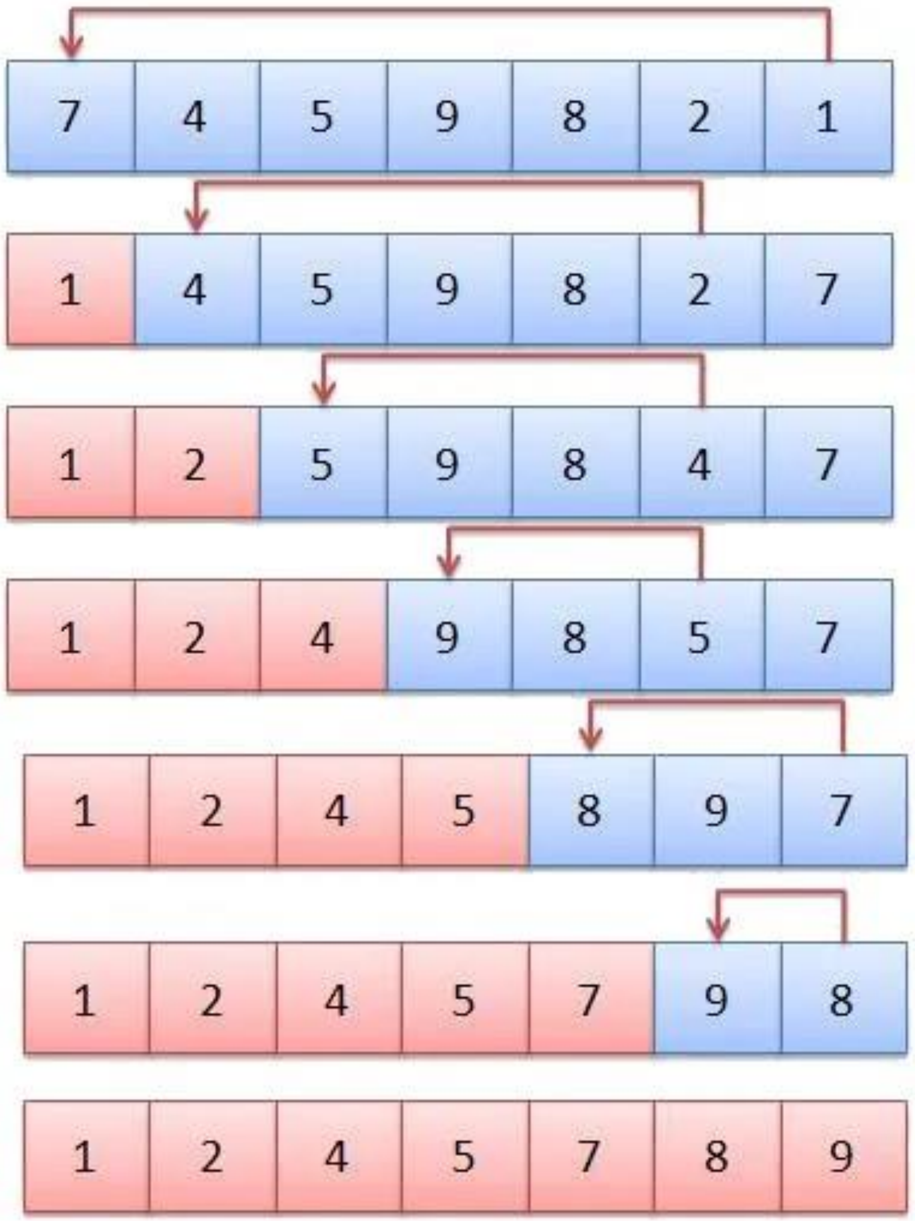
INSERTION SORT ALGORITHM



```
procedure insertionSort(A : list of sortable items)
    n = length(A)
    for i = 2 to n do
        key = A[i]
        j = i - 1
        while j > 0 and A[j] > key do
            A[j+1] = A[j]
            j = j - 1
        end while
        A[j+1] = key
    end for
end procedure
```

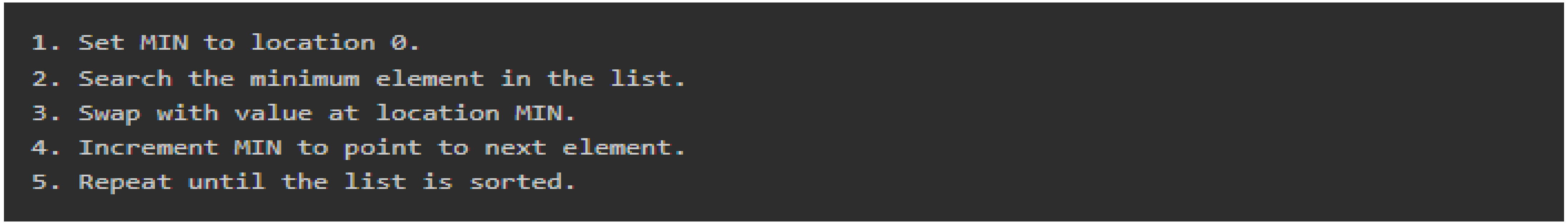
SELECTION SORT

- Selection sort is a simple sorting algorithm.
- This sorting algorithm, like insertion sort, is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
- The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundaries by one element to the right.
- This algorithm is not suitable for large data sets as its average and worst case complexities are of O(n2), where n is the number of items.



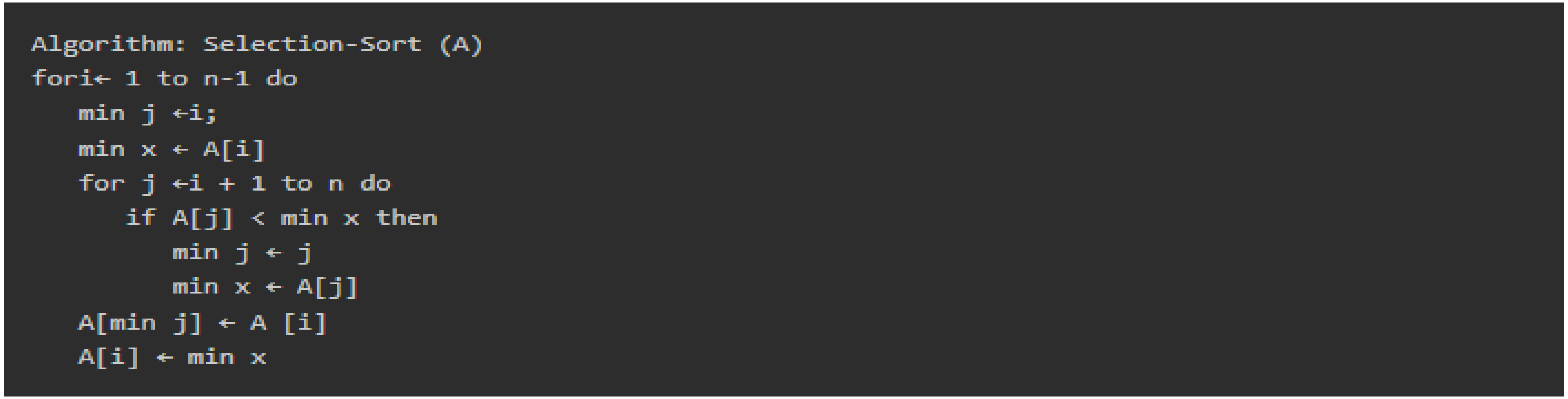
7 4 5 9 8 2 1
1 4 5 9 8 2 7
1 2 5 9 8 4 7
1 2 4 9 8 5 7
1 2 4 5 8 9 7
1 2 4 5 7 9 8
1 2 4 5 7 8 9

SELECTION SORT ALGORITHM



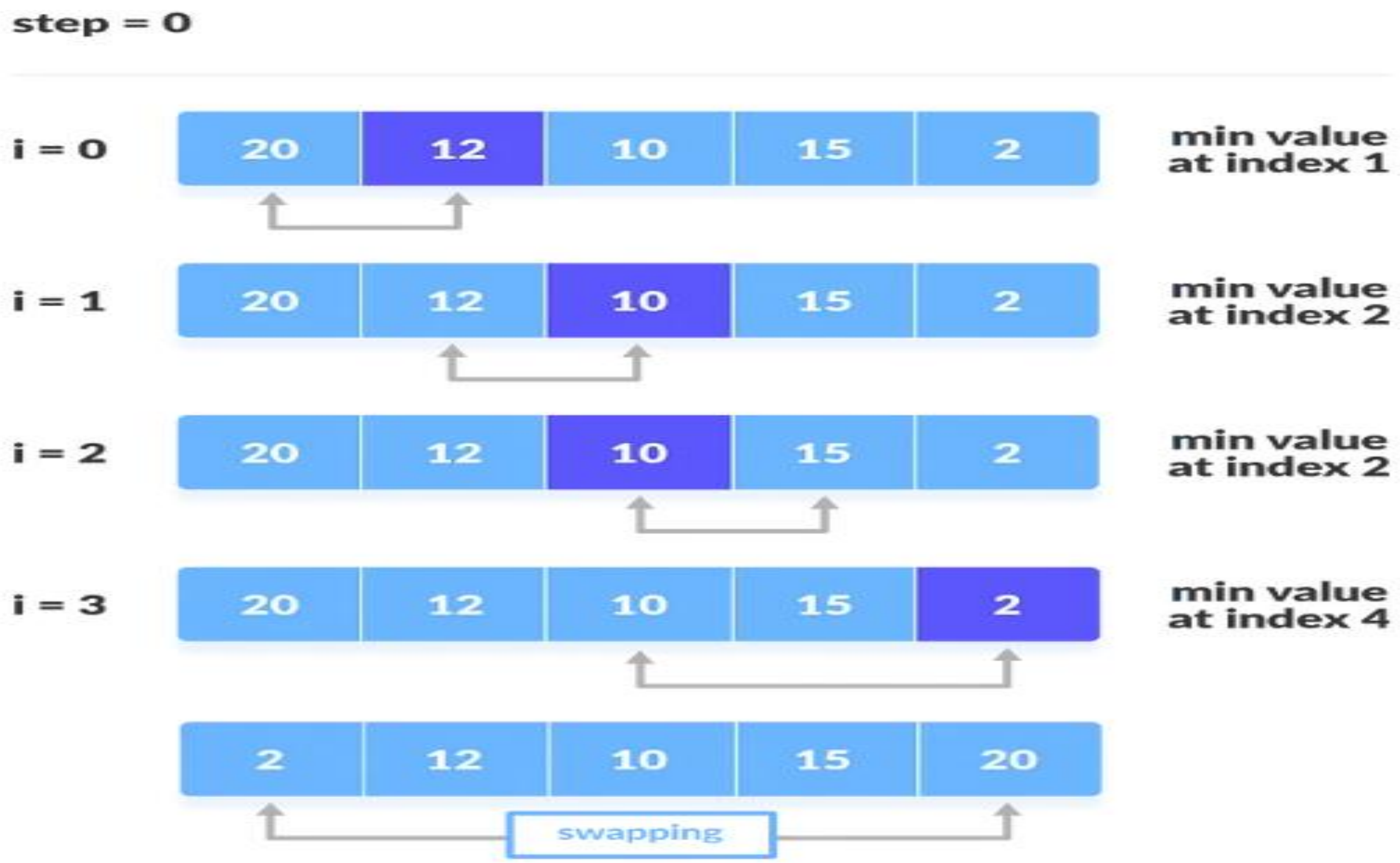
```
1. Set MIN to location 0.
2. Search the minimum element in the list.
3. Swap with value at location MIN.
4. Increment MIN to point to next element.
5. Repeat until the list is sorted.
```

Pseudocode

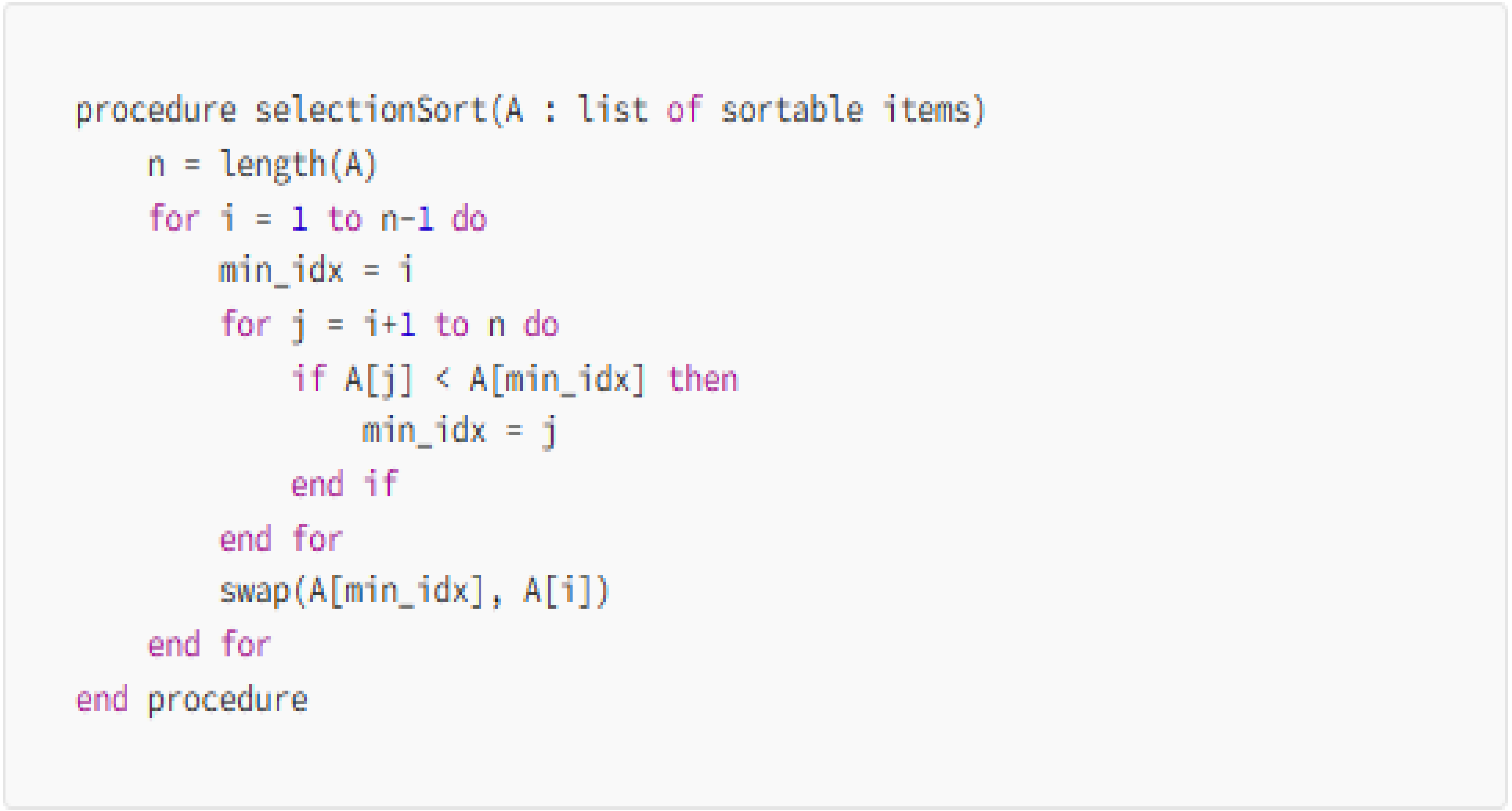


```
Algorithm: Selection-Sort (A)
fori← 1 to n-1 do
   min j ←i;
   min x ← A[i]
   for j ←i + 1 to n do
      if A[j] < min x then
         min j ← j
         min x ← A[j]
   A[min j] ← A [i]
   A[i] ← min x
```

SELECTION SORT ALGORITHM



SELECTION SORT ALGORITHM

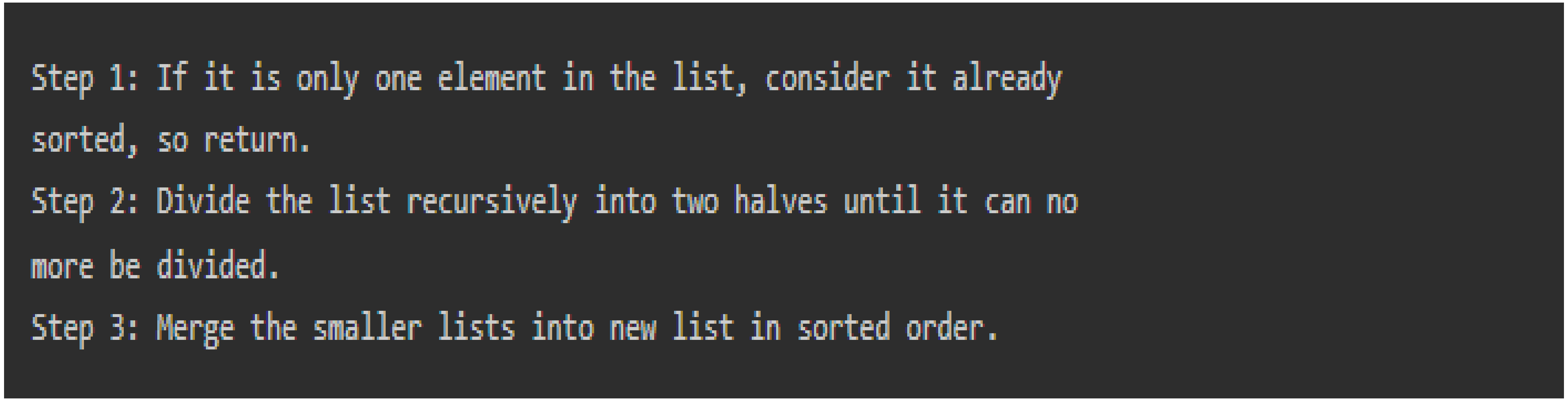


```
procedure selectionSort(A : list of sortable items)
    n = length(A)
    for i = 1 to n-1 do
        min_idx = i
        for j = i+1 to n do
            if A[j] < A[min_idx] then
                min_idx = j
            end if
        end for
        swap(A[min_idx], A[i])
    end for
end procedure
```

MERGE SORT ALGORITHM

- Merge sort is a divide-and-conquer algorithm that works by dividing the list into smaller sub-lists, sorting them, and then merging them back together.
- The algorithm recursively divides the list in half until each sub-list contains only one element. It then merges the sub-lists back together, comparing the elements in each sub-list and merging them in the correct order.
- Merge sort has a time complexity of O(n log n) and is more efficient than the previous algorithms for large datasets.

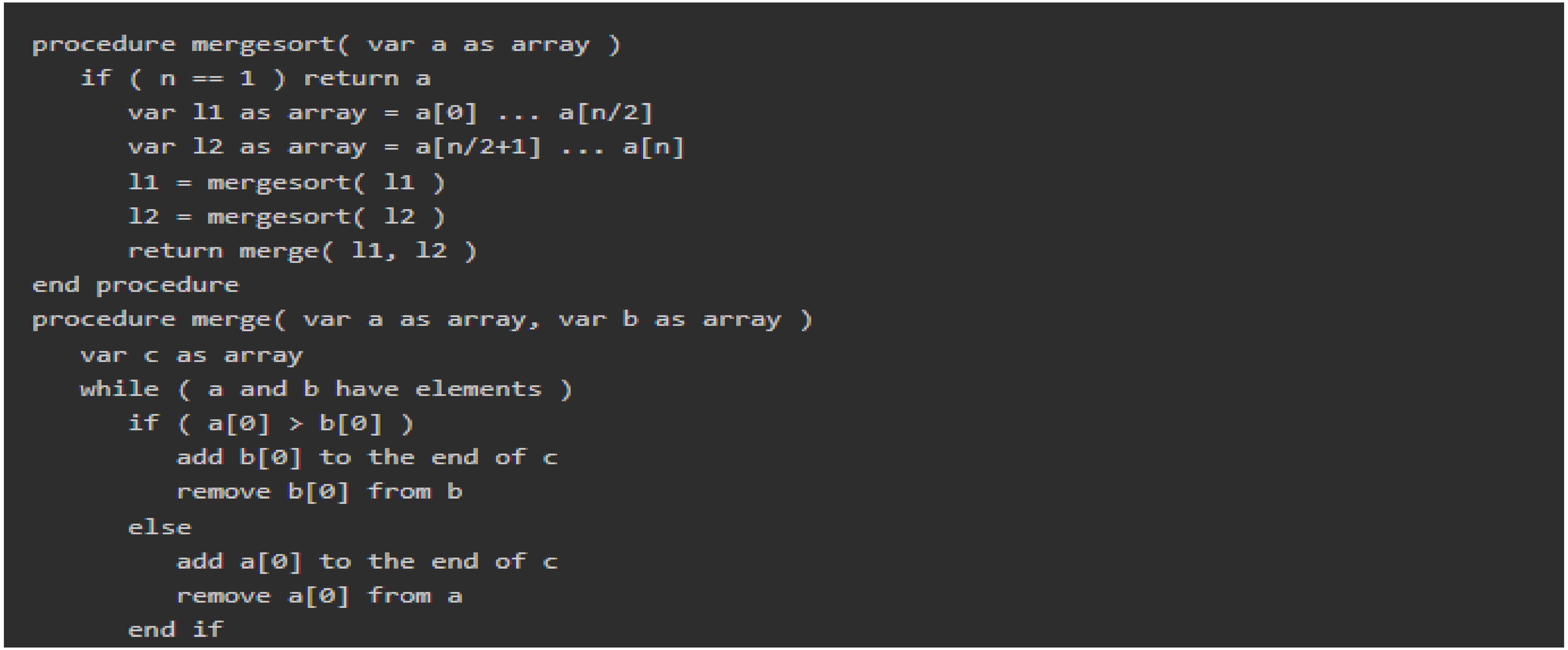
MERGE SORT ALGORITHM



```
Step 1: If it is only one element in the list, consider it already
sorted, so return.
Step 2: Divide the list recursively into two halves until it can no
more be divided.
Step 3: Merge the smaller lists into new list in sorted order.
```

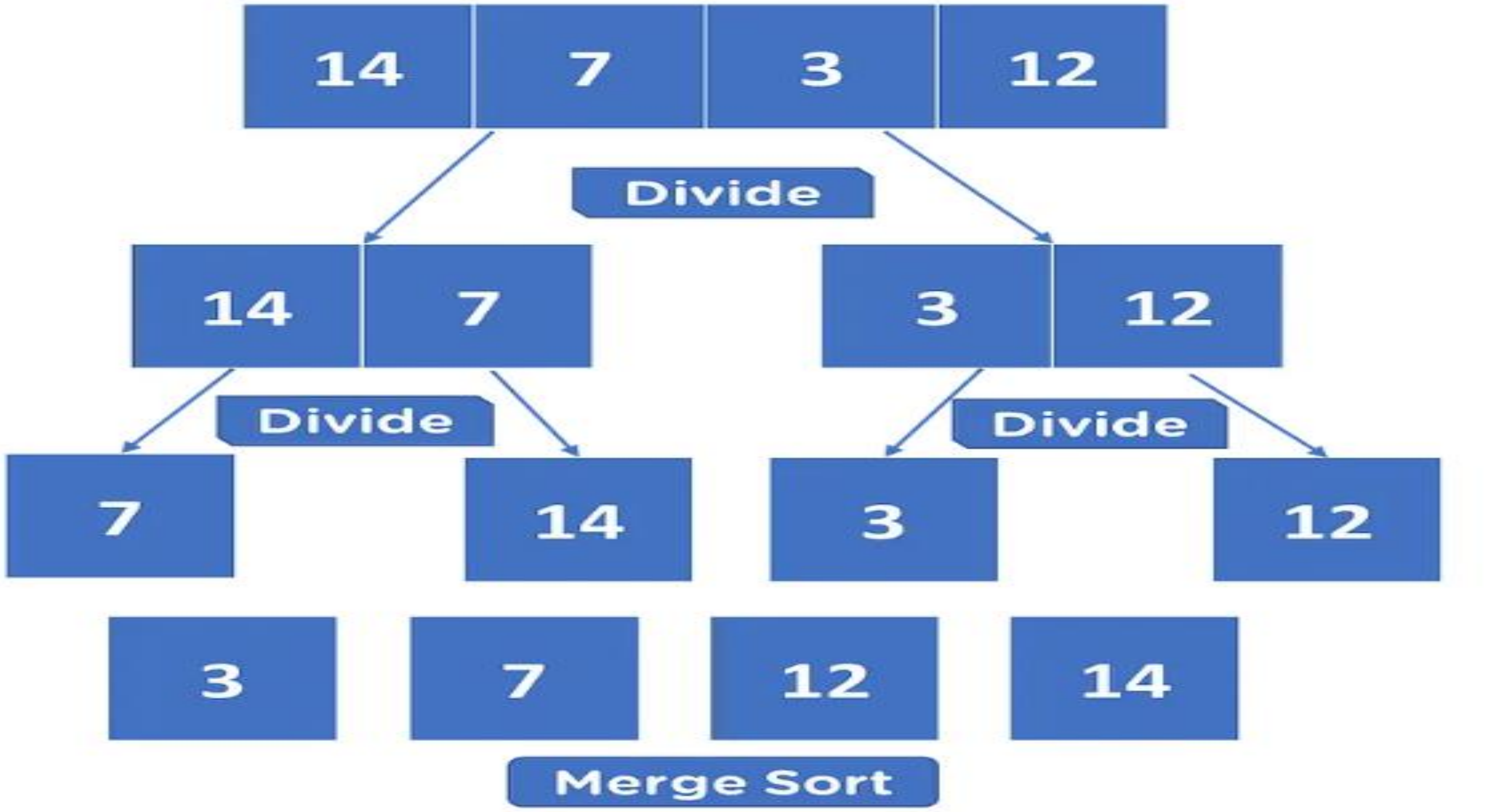
MERGE SORT ALGORITHM

- **Pseudocode**

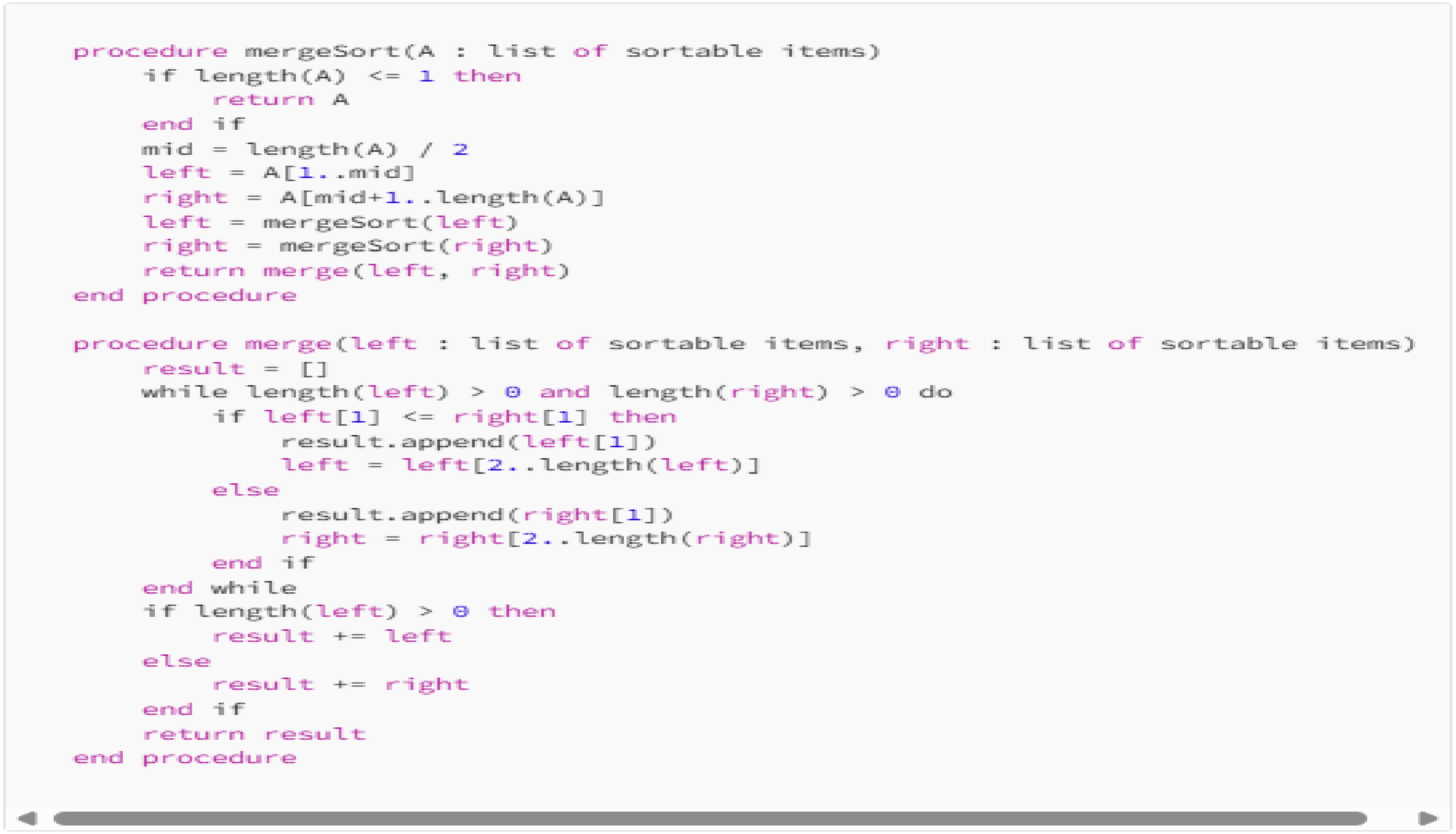


```
procedure mergesort( var a as array )
   if ( n == 1 ) return a
      var l1 as array = a[0] ... a[n/2]
      var l2 as array = a[n/2+1] ... a[n]
      l1 = mergesort( l1 )
      l2 = mergesort( l2 )
      return merge( l1, l2 )
end procedure
procedure merge( var a as array, var b as array )
   var c as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
```

MERGE SORT ALGORITHM



SELECTION SORT ALGORITHM



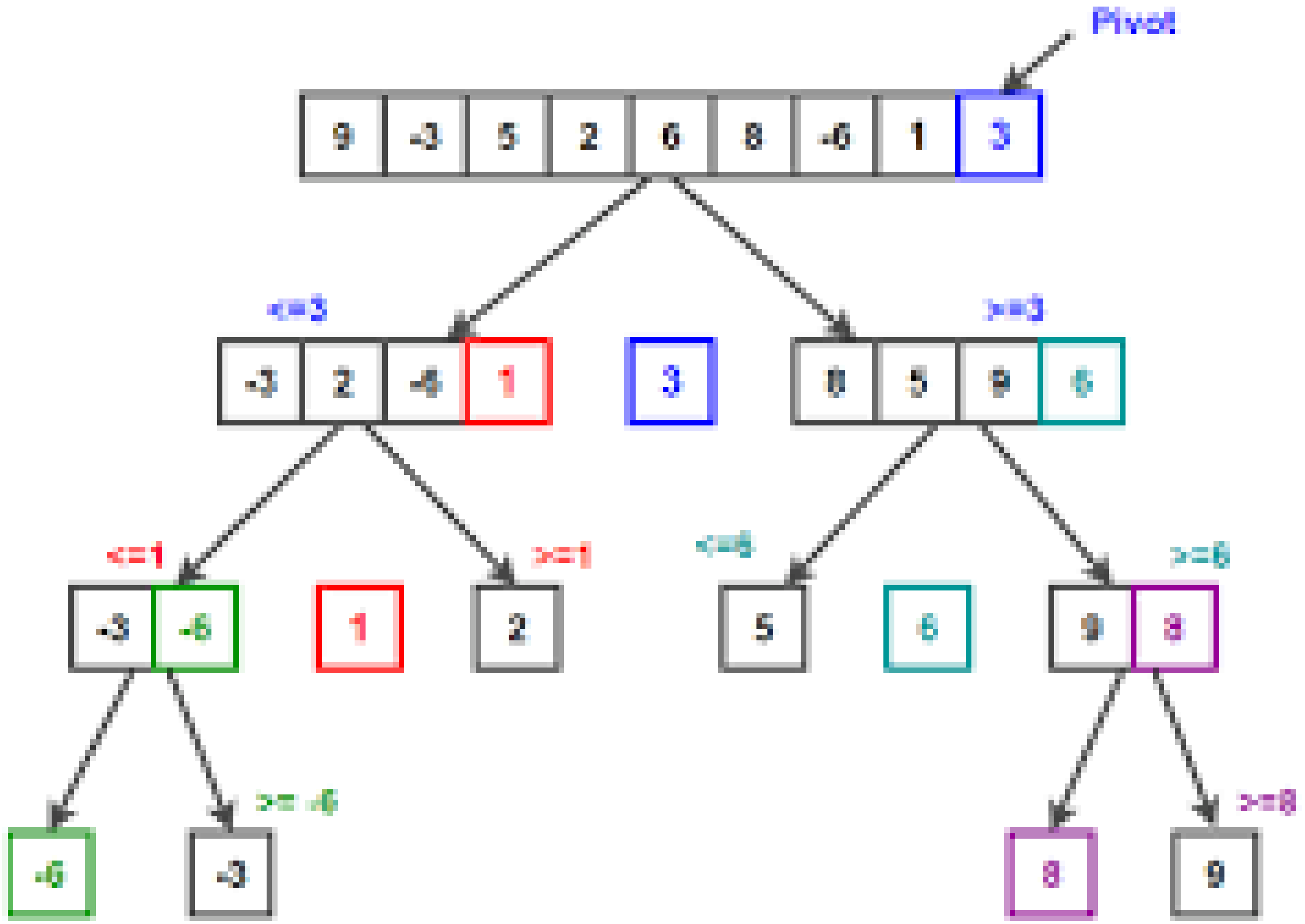
```
procedure mergeSort(A : list of sortable items)
    if length(A) <= 1 then
        return A
    end if
    mid = length(A) / 2
    left = A[1..mid]
    right = A[mid+1..length(A)]
    left = mergeSort(left)
    right = mergeSort(right)
    return merge(left, right)
end procedure

procedure merge(left : list of sortable items, right : list of sortable items)
    result = []
    while length(left) > 0 and length(right) > 0 do
        if left[1] <= right[1] then
            result.append(left[1])
            left = left[2..length(left)]
        else
            result.append(right[1])
            right = right[2..length(right)]
        end if
    end while
    if length(left) > 0 then
        result += left
    else
        result += right
    end if
    return result
end procedure
```

QUICK SORT

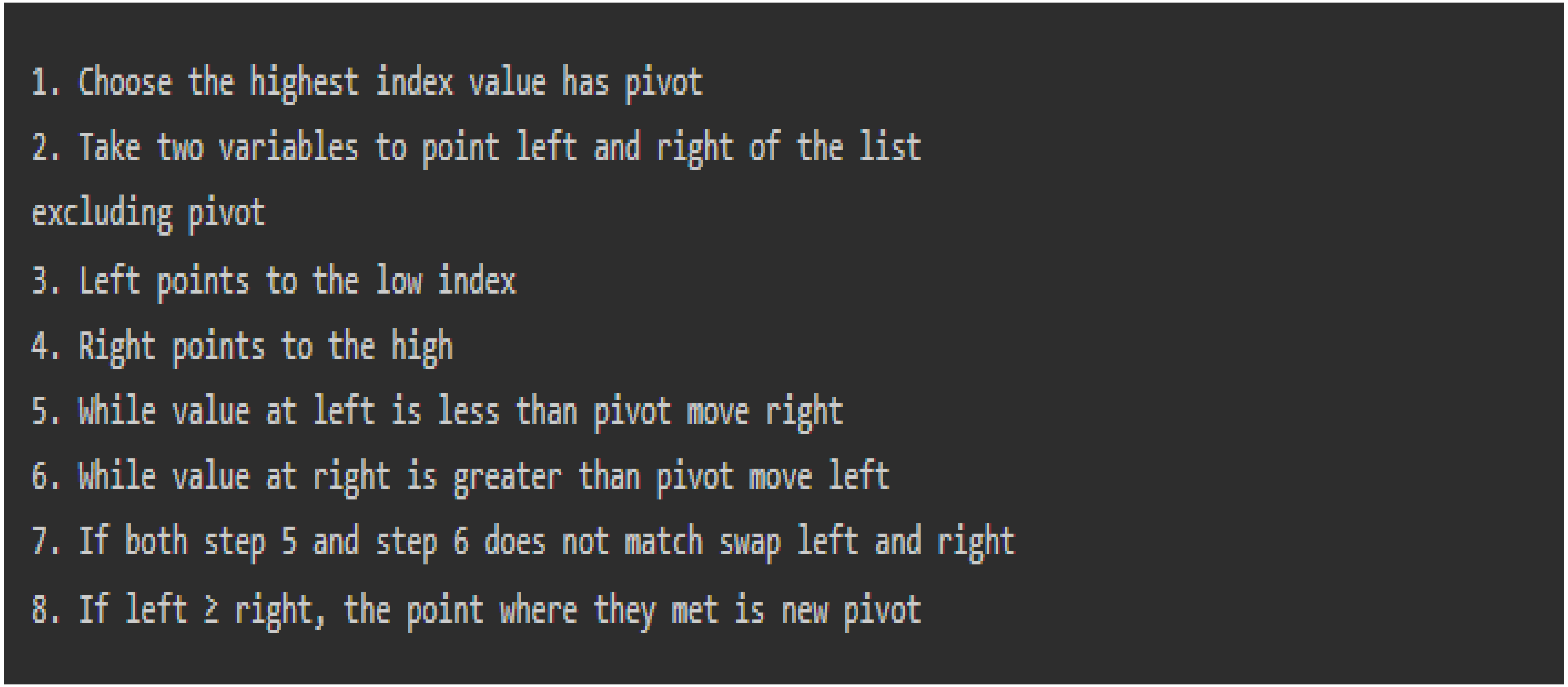
- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.
- Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.
- This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n2), respectively.

QUICK SORT ALGORITHM

- Quick sort is another divide-and-conquer algorithm that works by selecting a pivot element from the list and partitioning the other elements into two sub-lists, according to whether they are less than or greater than the pivot element.
- The algorithm then recursively sorts the sub-lists, using the same pivot element selection and partitioning process until the entire list is sorted.
- Quick sort has a time complexity of O(n log n) and is more efficient than merge sort for small to medium-sized datasets.

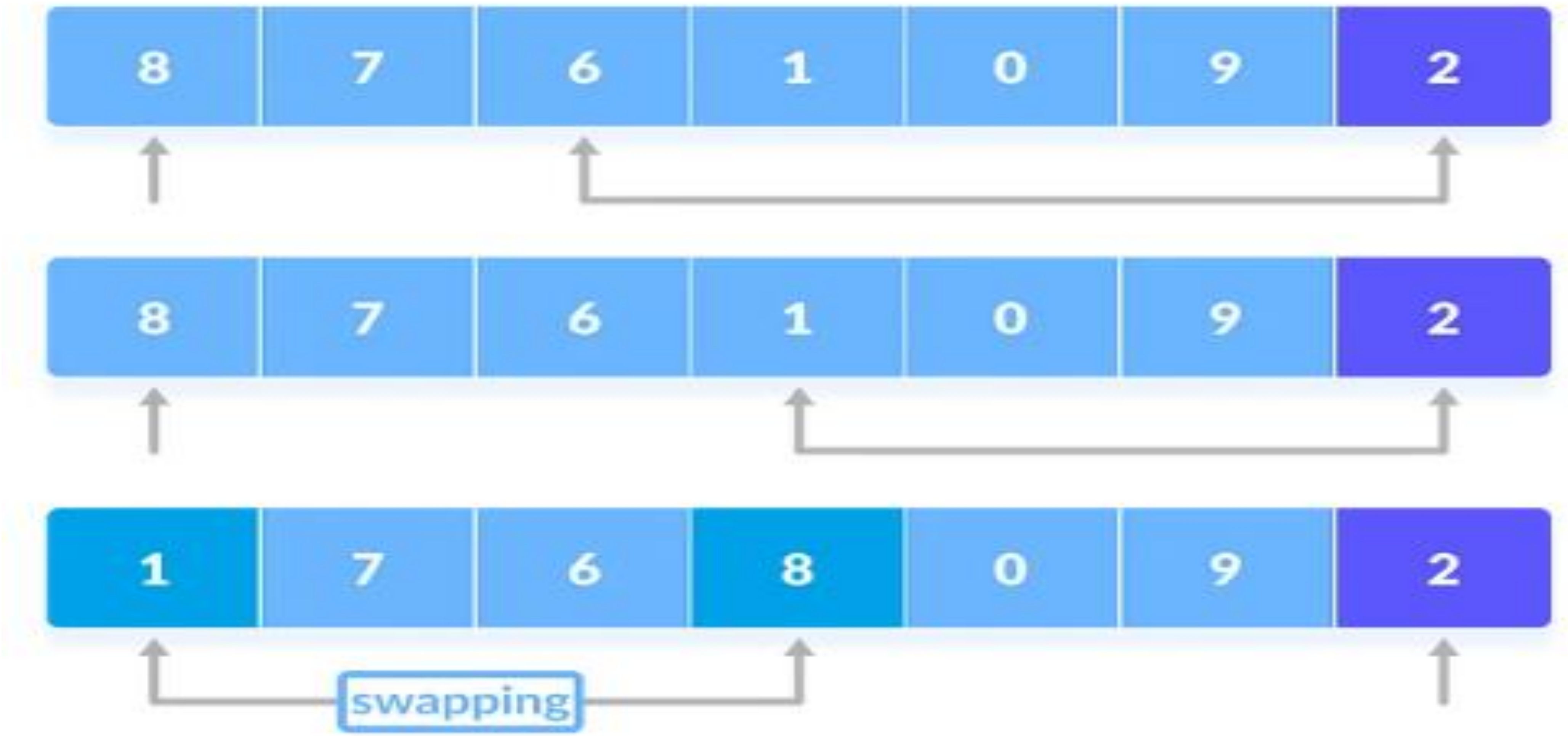

Pivot
9 -3 5 2 6 8 -6 1 3
<=3
-3 2 -6 1
3
>=3
8 5 9 6
<=1
-3 -6
1
>=1
2
<=6
5
6
>=6
9 8
-6
>= -6
-3
8
>=8
9

QUICK SORT PIVOT ALGORITHM

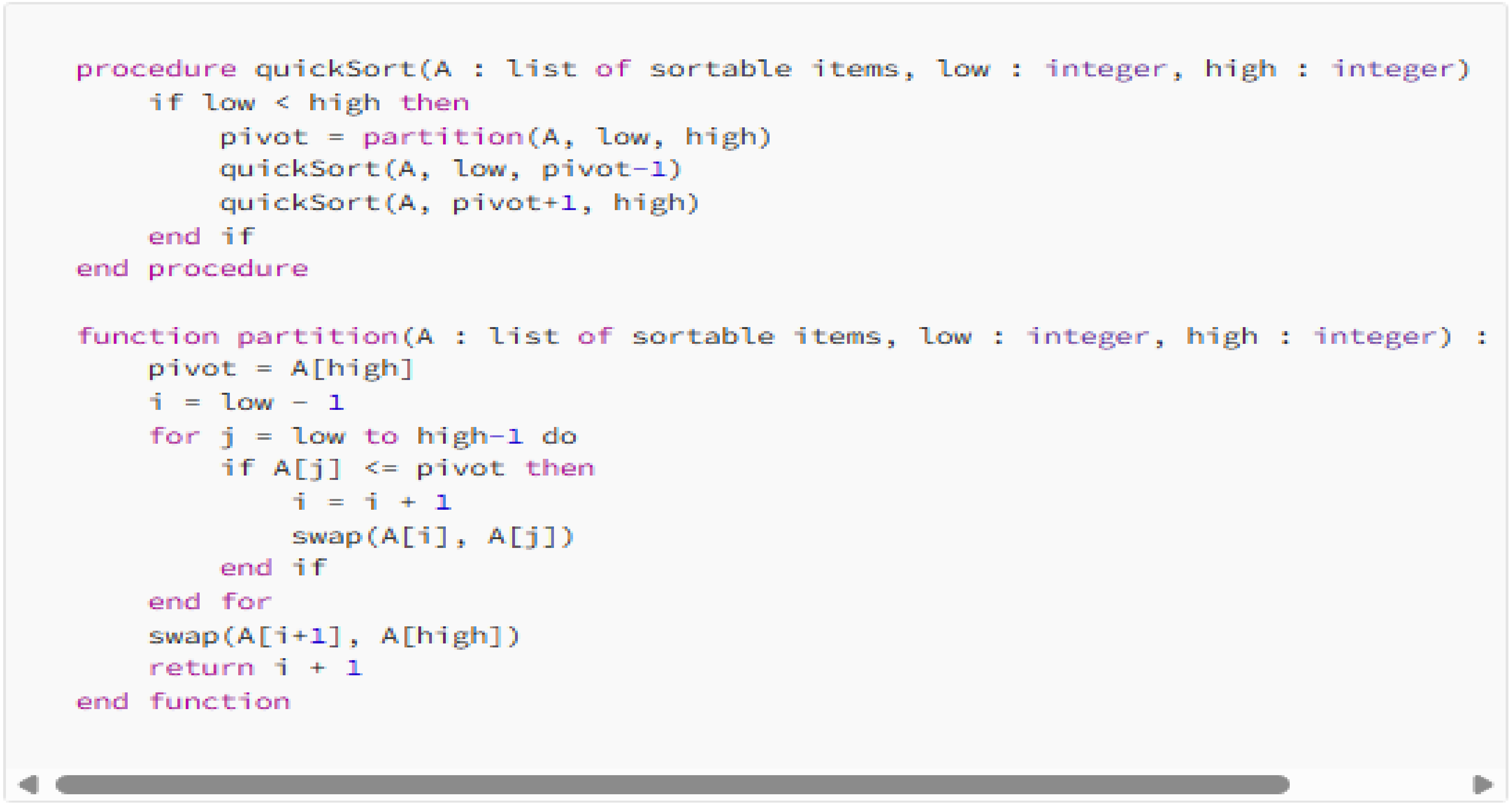


1. Choose the highest index value has pivot
2. Take two variables to point left and right of the list excluding pivot
3. Left points to the low index
4. Right points to the high
5. While value at left is less than pivot move right
6. While value at right is greater than pivot move left
7. If both step 5 and step 6 does not match swap left and right
8. If left ≥ right, the point where they met is new pivot

QUICK SORT PIVOT ALGORITHM



QUICK SORT PIVOT ALGORITHM



```
procedure quickSort(A : list of sortable items, low : integer, high : integer)
    if low < high then
        pivot = partition(A, low, high)
        quickSort(A, low, pivot-1)
        quickSort(A, pivot+1, high)
    end if
end procedure

function partition(A : list of sortable items, low : integer, high : integer) :
    pivot = A[high]
    i = low - 1
    for j = low to high-1 do
        if A[j] <= pivot then
            i = i + 1
            swap(A[i], A[j])
        end if
    end for
    swap(A[i+1], A[high])
    return i + 1
end function
```

HEAP SORT ALGORITHM

- Heap Sort is an efficient sorting technique based on the heap data structure.
- The heap is a nearly-complete binary tree where the parent node could either be minimum or maximum.
- The heap with minimum root node is called min-heap and the root node with maximum root node is called max-heap.
- The elements in the input data of the heap sort algorithm are processed using these two methods.

HEAP SORT ALGORITHM

- Heap sort has a time complexity of O(n log n) and is more efficient than the previous algorithms for large datasets.

HEAP SORT ALGORITHM

The heap sort algorithm follows two main operations in this procedure

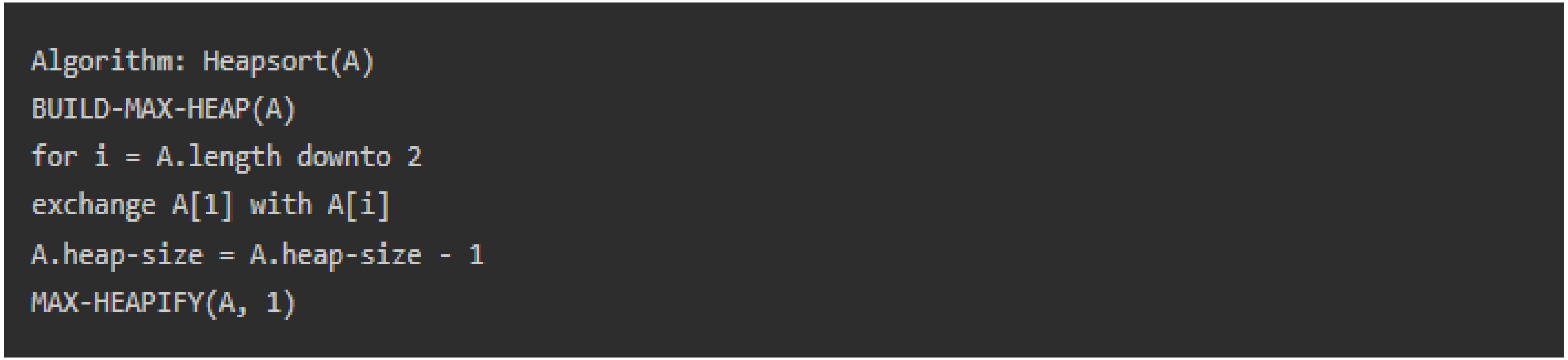
- Builds a heap H from the input data using the heapify (explained further into the chapter) method, based on the way of sorting ascending order or descending order.
- Deletes the root element of the root element and repeats until all the input elements are processed.

HEAPIFY METHOD

- The heapify method of a binary tree is to convert the tree into a heap data structure. This method uses recursion approach to heapify all the nodes of the binary tree.
- The binary tree must always be a complete binary tree as it must have two children nodes always.
- The complete binary tree will be converted into either a max-heap or a min-heap by applying the heapify method.

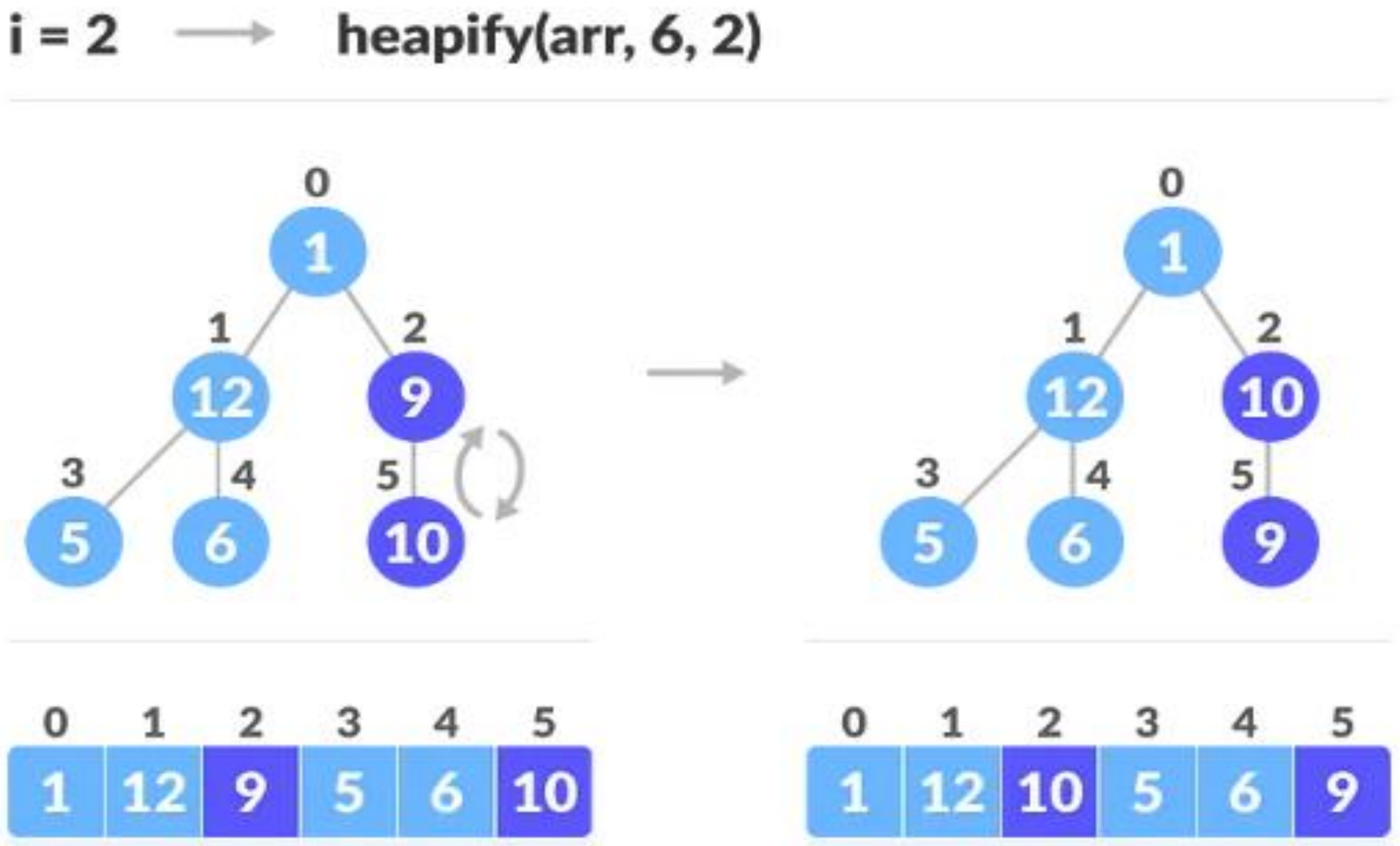
HEAP SORT ALGORITHM

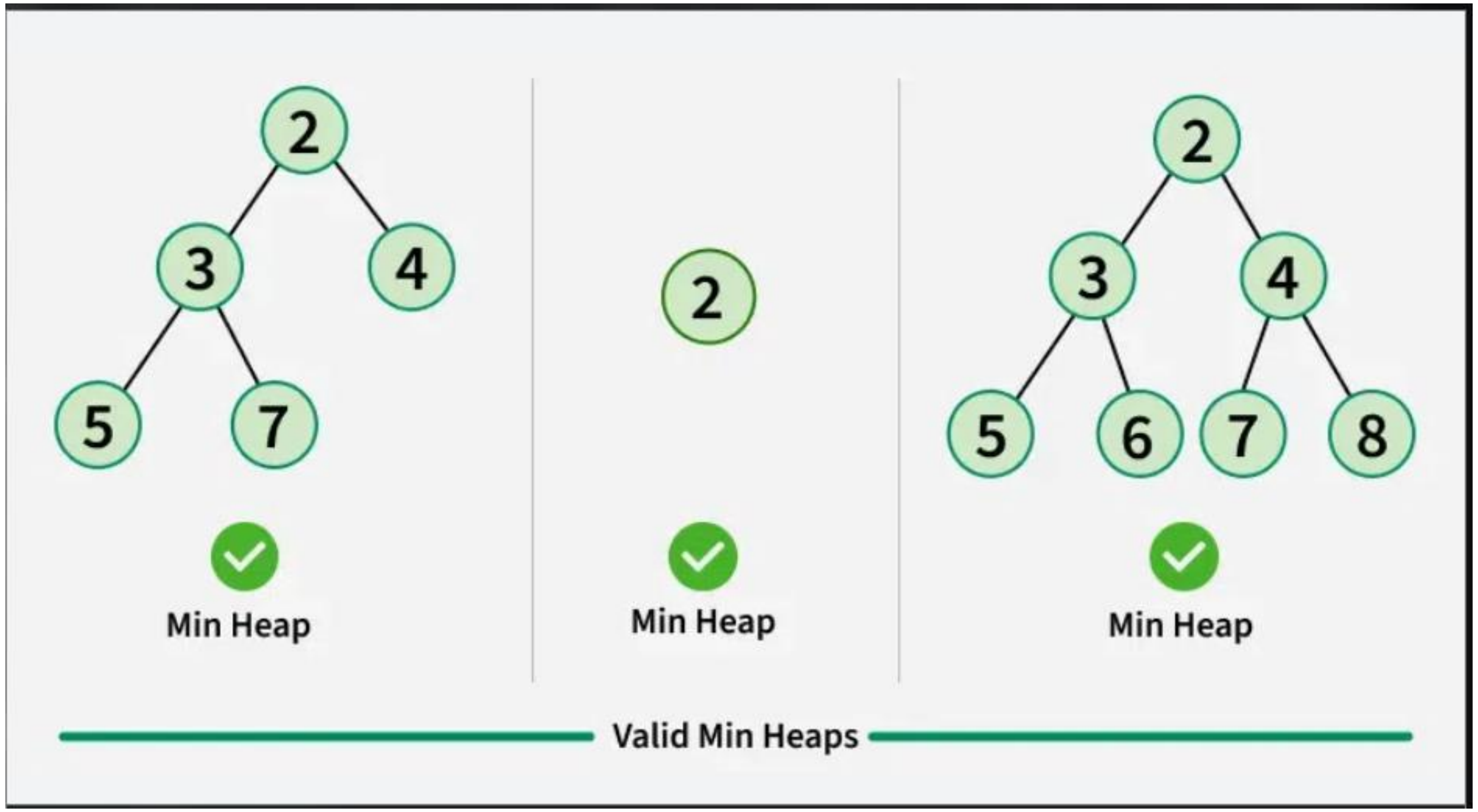
- As described in the algorithm below, the sorting algorithm first constructs the heap ADT by calling the Build-Max-Heap algorithm and removes the root element to swap it with the minimum valued node at the leaf. Then the heapify method is applied to rearrange the elements accordingly.

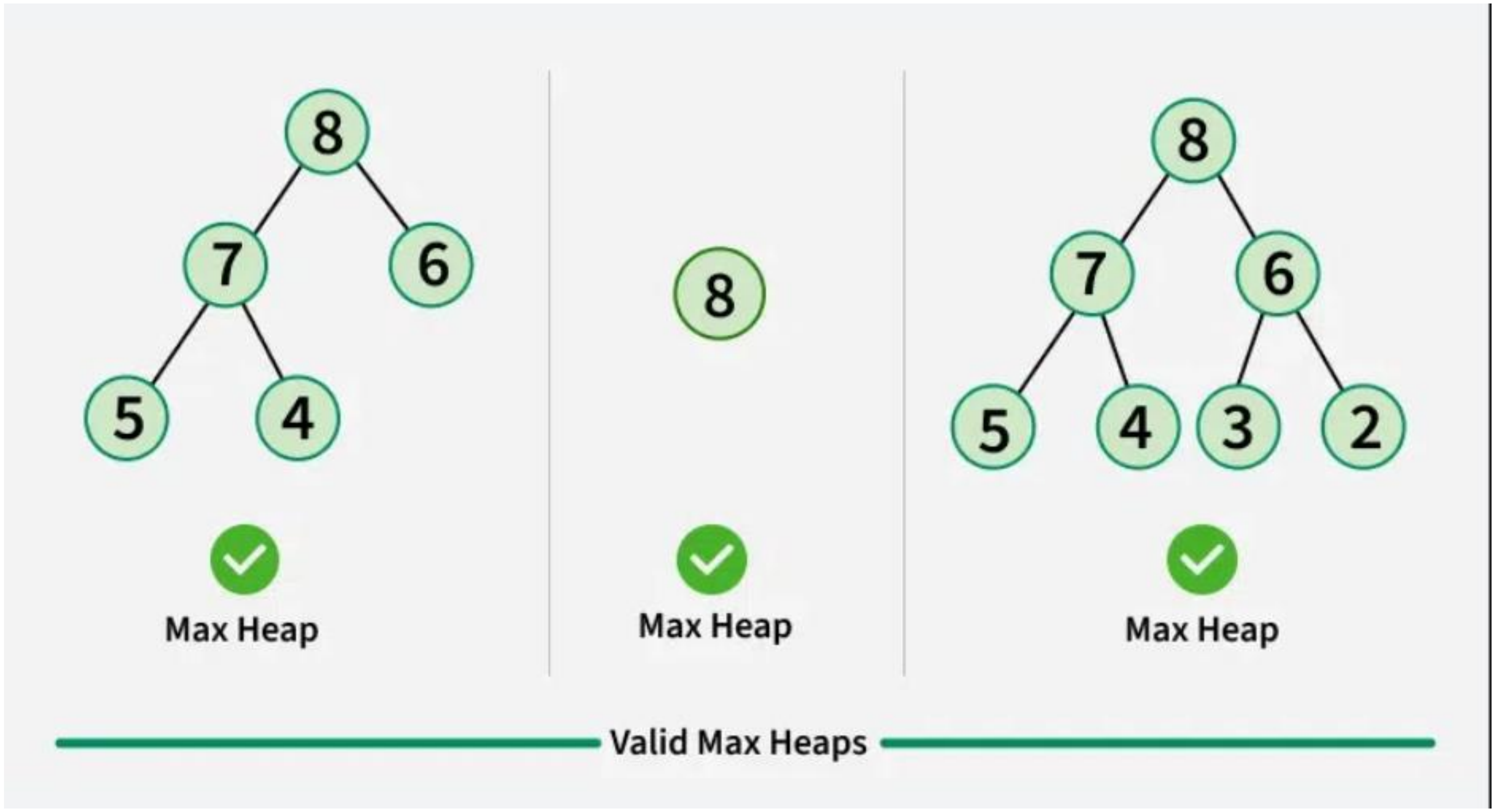


```
Algorithm: Heapsort(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
exchange A[1] with A[i]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
```

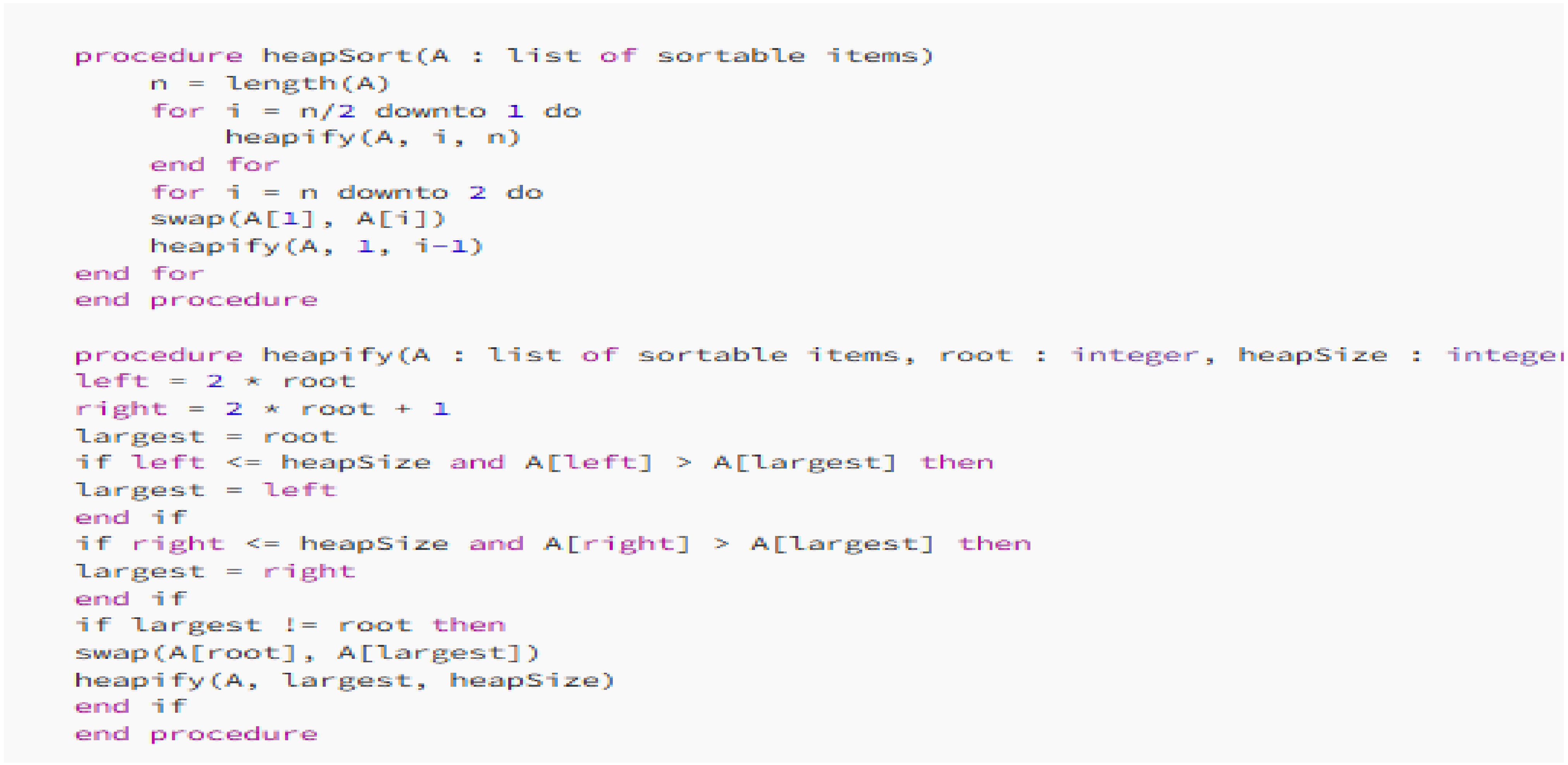
HEAP SORT ALGORITHM




2
3
4
5
7
Min Heap
2
Min Heap
2
3
4
5
6
7
8
Min Heap
Valid Min Heaps


8
7
6
5
4
Max Heap
8
Max Heap
8
7
6
5
4
3
2
Max Heap
Valid Max Heaps

HEAP SORT ALGORITHM



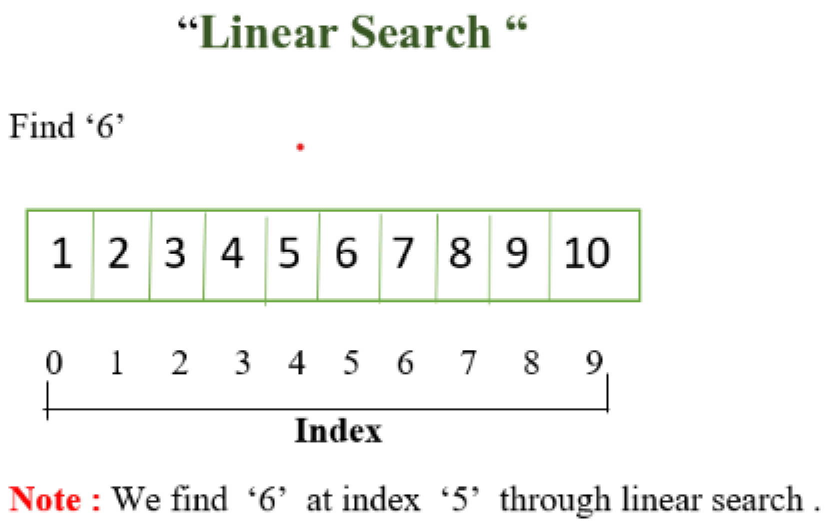
```
procedure heapSort(A : list of sortable items)
    n = length(A)
    for i = n/2 downto 1 do
        heapify(A, i, n)
    end for
    for i = n downto 2 do
    swap(A[1], A[i])
    heapify(A, 1, i-1)
end for
end procedure

procedure heapify(A : list of sortable items, root : integer, heapSize : intege
left = 2 * root
right = 2 * root + 1
largest = root
if left <= heapSize and A[left] > A[largest] then
largest = left
end if
if right <= heapSize and A[right] > A[largest] then
largest = right
end if
if largest != root then
swap(A[root], A[largest])
heapify(A, largest, heapSize)
end if
end procedure
```

SEARCHING ALGORITHMS

- Searching algorithms are essential tools in computer science used to locate specific items within a collection of data. In this tutorial, we are mainly going to focus upon searching in an array.
- When we search an item in an array, there are two most common algorithms used based on the type of input array.
  1. Linear Search
  2. Binary Search

LINEAR SEARCH ALGORITHM

- Linear search is defined as the searching algorithm where the list or data set is traversed from one end to find the desired value

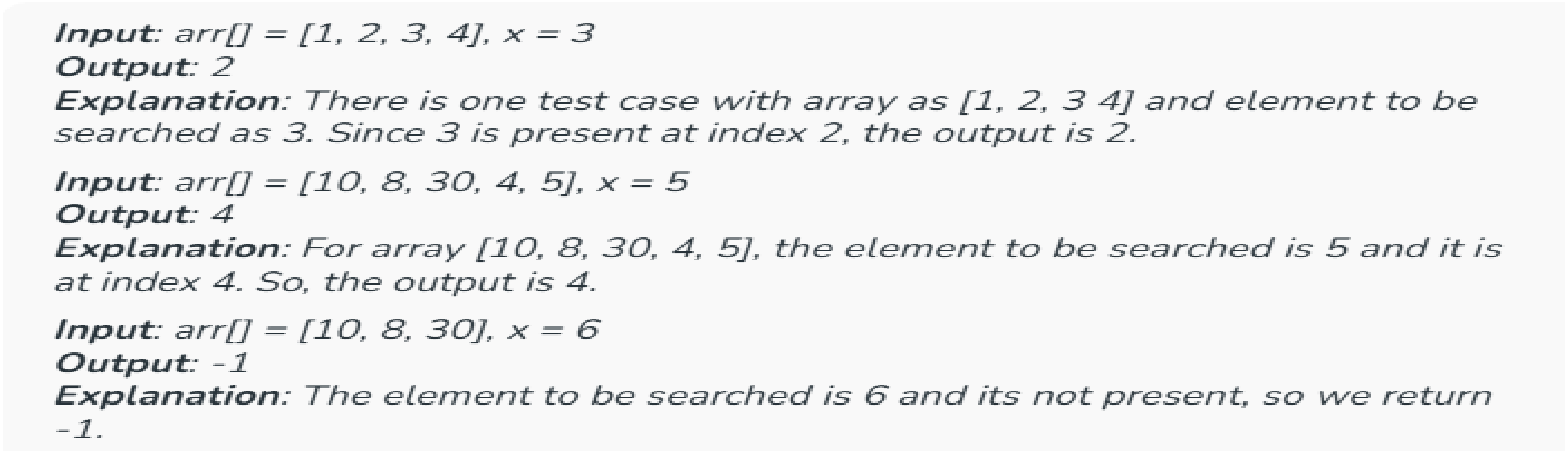


LINEAR SEARCH ALGORITHM

- It is used for an unsorted array.
- It mainly does one by one comparison of the item to be search with array elements. It takes linear or O(n) Time.

LINEAR SEARCH ALGORITHM

- Given an array, arr of n integers, and an integer element x, find whether element x is present in the array.
- Return the index of the first occurrence of x in the array, or -1 if it doesn't exist.



Input: *arr[] = [1, 2, 3, 4], x = 3*
Output: *2*
Explanation: *There is one test case with array as [1, 2, 3 4] and element to be searched as 3. Since 3 is present at index 2, the output is 2.*

Input: *arr[] = [10, 8, 30, 4, 5], x = 5*
Output: *4*
Explanation: *For array [10, 8, 30, 4, 5], the element to be searched is 5 and it is at index 4. So, the output is 4.*

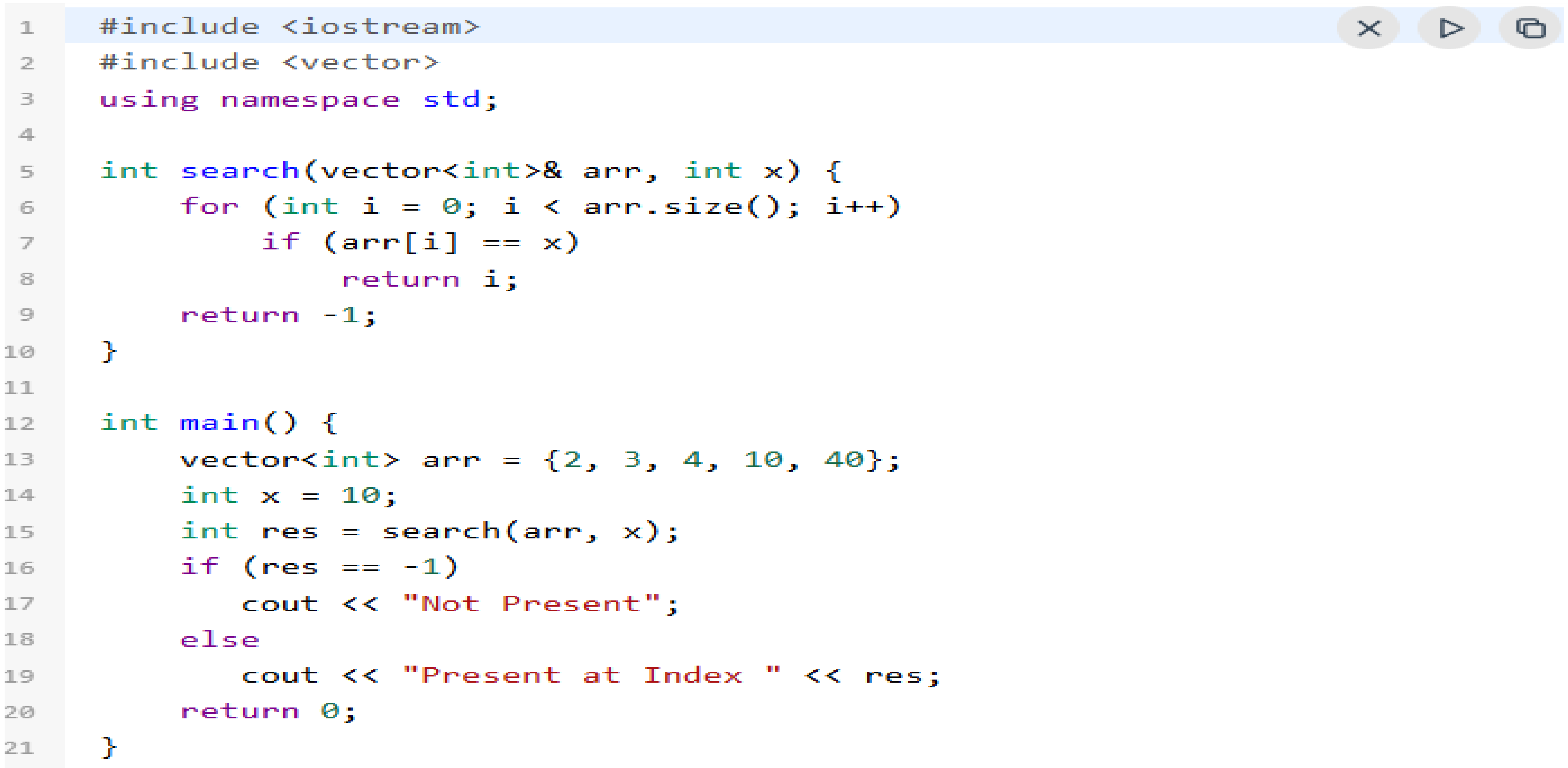
Input: *arr[] = [10, 8, 30], x = 6*
Output: *-1*
Explanation: *The element to be searched is 6 and its not present, so we return -1.*

LINEAR SEARCH ALGORITHM

- In Linear Search, we iterate over all the elements of the array and check if it the current element is equal to the target element.
- If we find any element to be equal to the target element, then return the index of the current element. Otherwise, if no element is equal to the target element, then return -1 as the element is not found.
- Linear search is also known as sequential search.

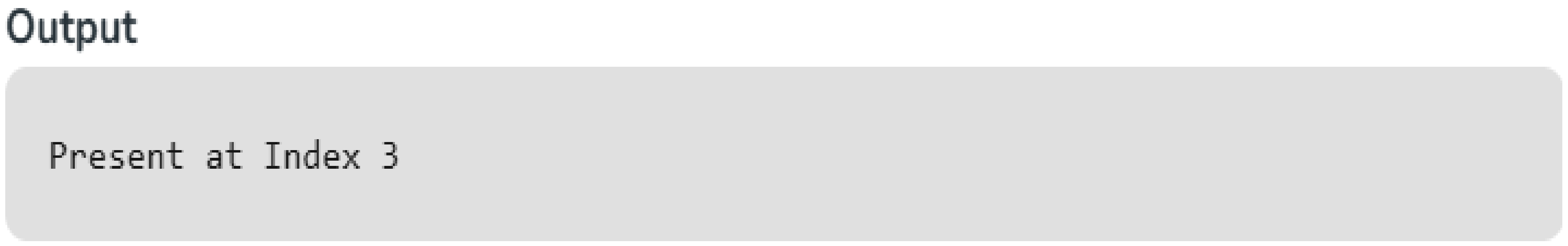
LINEAR SEARCH ALGORITHM

- Below is the implementation of the linear search algorithm:



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

int search(vector<int>& arr, int x) {
    for (int i = 0; i < arr.size(); i++)
        if (arr[i] == x)
            return i;
    return -1;
}

int main() {
    vector<int> arr = {2, 3, 4, 10, 40};
    int x = 10;
    int res = search(arr, x);
    if (res == -1)
        cout << "Not Present";
    else
        cout << "Present at Index " << res;
    return 0;
}
```

LINEAR SEARCH ALGORITHM


Output

```
Present at Index 3
```

TIME AND SPACE COMPLEXITY OF LINEAR SEARCH ALGORITHM

- **Time Complexity:**
- Best Case: In the best case, the key might be present at the first index. So the best case complexity is O(1)
- Worst Case: In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is O(N) where N is the size of the list.
- Average Case: O(N)
- **Auxiliary Space:** O(1) as except for the variable to iterate through the list, no other variable is used.

APPLICATIONS OF LINEAR SEARCH ALGORITHM

- Unsorted Lists: When we have an unsorted array or list, linear search is most commonly used to find any element in the collection.
- Small Data Sets: Linear Search is preferred over binary search when we have small data sets with
- Searching Linked Lists: In linked list implementations, linear search is commonly used to find elements within the list. Each node is checked sequentially until the desired element is found.
- Simple Implementation: Linear Search is much easier to understand and implement as compared to Binary Search or Ternary Search

ADVANTAGES OF LINEAR SEARCH ALGORITHM

- Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
- Does not require any additional memory.
- It is a well-suited algorithm for small datasets.

DISADVANTAGES OF LINEAR SEARCH ALGORITHM

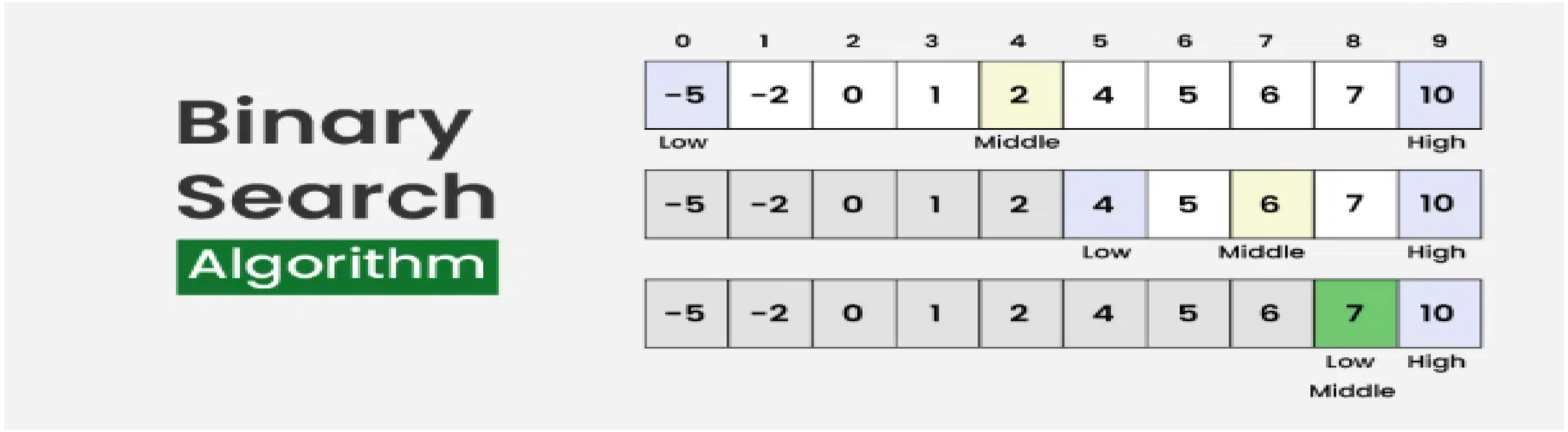
- Linear search has a time complexity of O(N), which in turn makes it slow for large datasets.
- Not suitable for large arrays.

DISADVANTAGES OF LINEAR SEARCH ALGORITHM

- Linear search has a time complexity of O(N), which in turn makes it slow for large datasets.
- Not suitable for large arrays.

BINARY SEARCH ALGORITHM

- Binary Search Algorithm is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half.
- The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).



CONDITIONS TO APPLY BINARY SEARCH ALGORITHM IN A DATA STRUCTURE

- The data structure must be sorted.
- Access to any element of the data structure should take constant time

BINARY SEARCH ALGORITHM

- Divide the search space into two halves by finding the middle index "mid".
- Compare the middle element of the search space with the key.
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
  - If the key is smaller than the middle element, then the left side is used for next search.
  - If the key is larger than the middle element, then the right side is used for next search.

BINARY SEARCH ALGORITHM

- This process is continued until the key is found or the total search space is exhausted

HOW TO IMPLEMENT BINARY SEARCH ALGORITHM

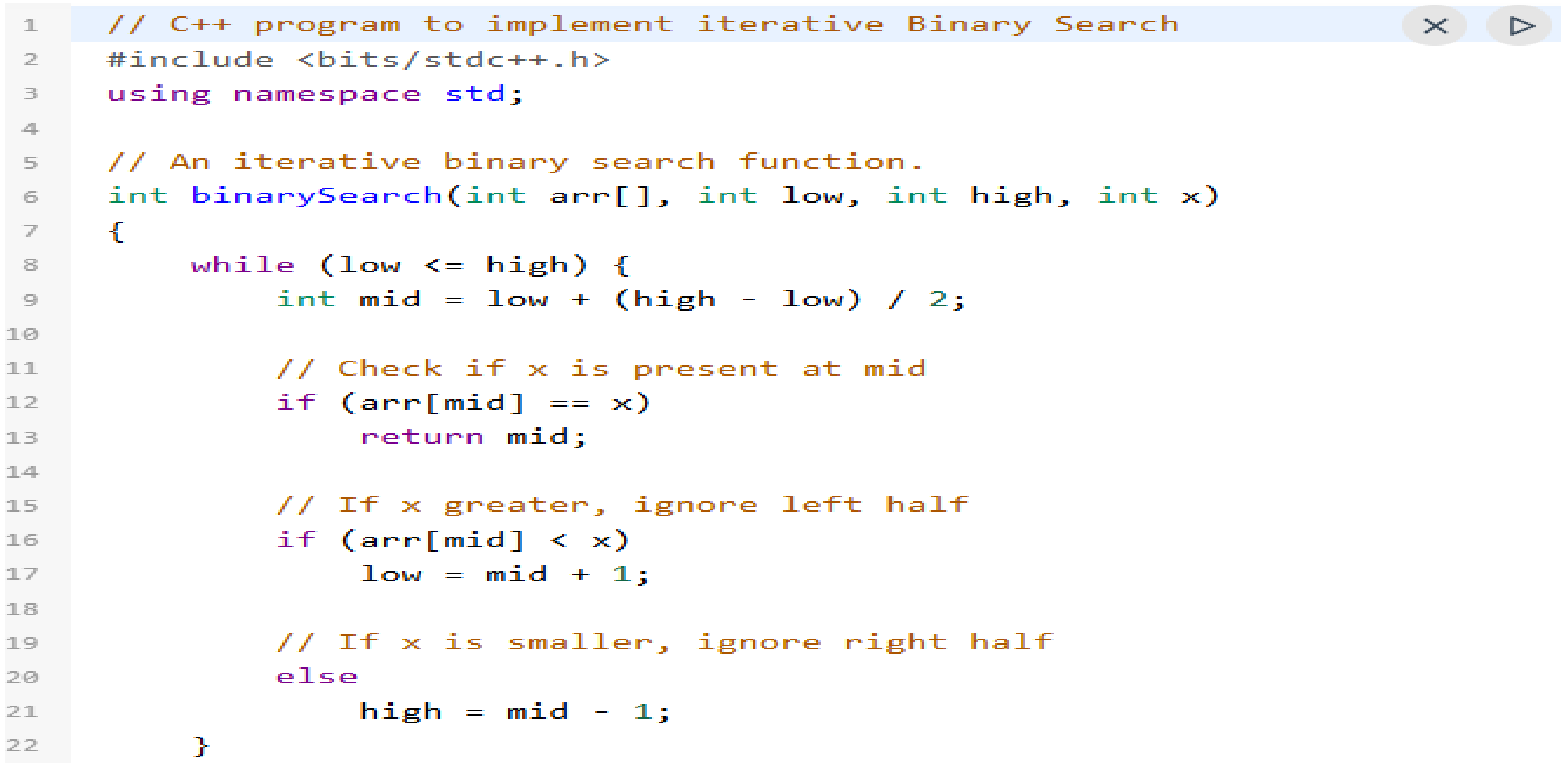
The Binary Search Algorithm can be implemented in the following two ways

- Iterative Binary Search Algorithm
- Recursive Binary Search Algorithm

ITERATIVE BINARY SEARCH ALGORITHM

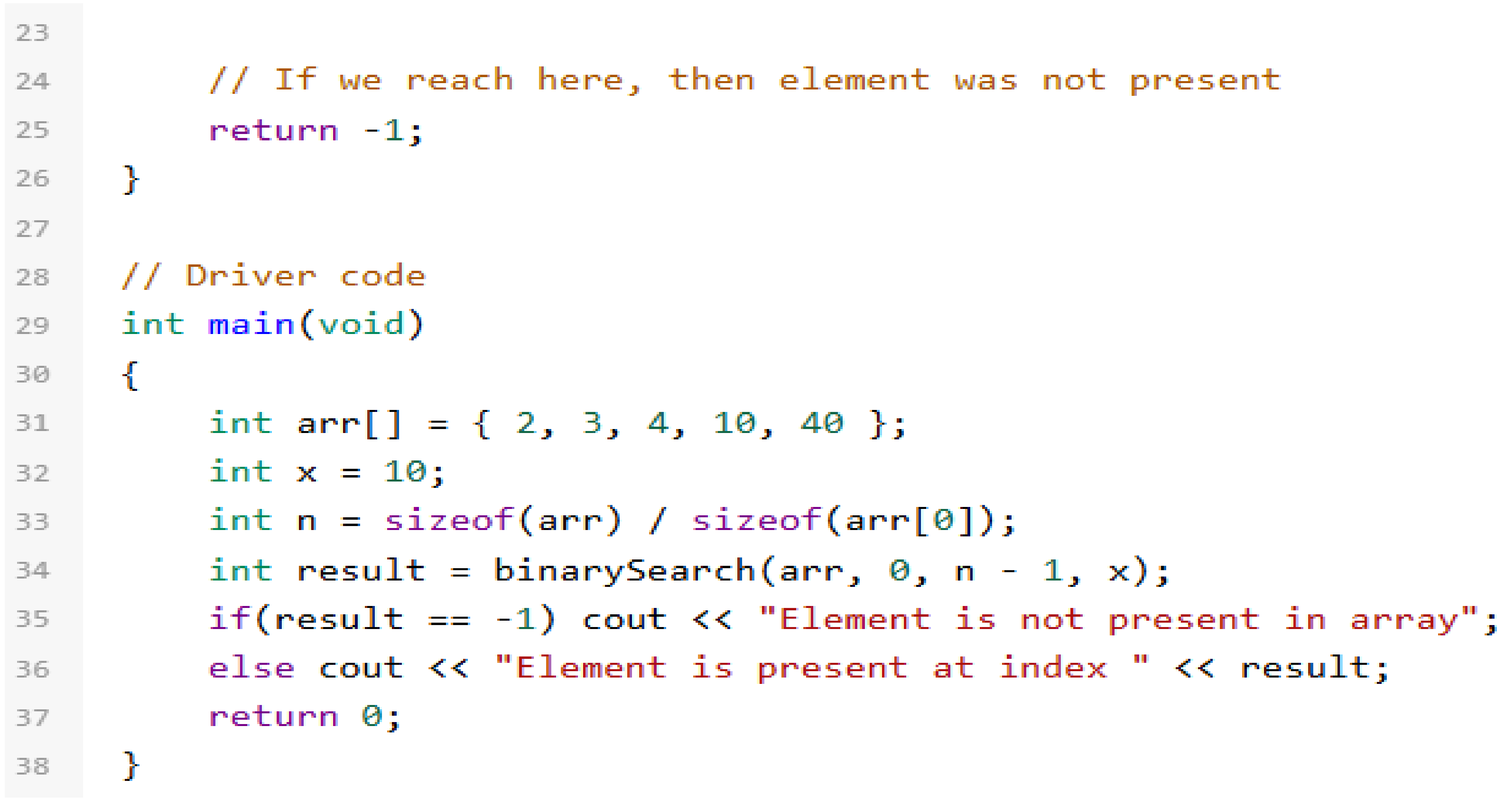
- Here we use a while loop to continue the process of comparing the key and splitting the search space in two halves.

ITERATIVE BINARY SEARCH ALGORITHM



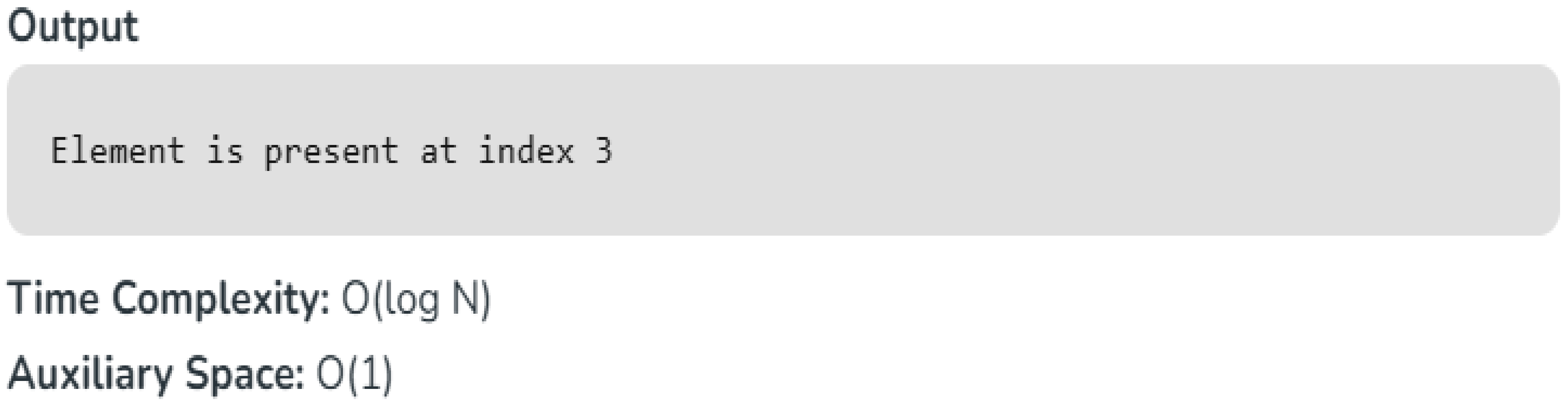
```cpp
// C++ program to implement iterative Binary Search
#include <bits/stdc++.h>
using namespace std;

// An iterative binary search function.
int binarySearch(int arr[], int low, int high, int x)
{
    while (low <= high) {
        int mid = low + (high - low) / 2;

        // Check if x is present at mid
        if (arr[mid] == x)
            return mid;

        // If x greater, ignore left half
        if (arr[mid] < x)
            low = mid + 1;

        // If x is smaller, ignore right half
        else
            high = mid - 1;
    }
```

ITERATIVE BINARY SEARCH ALGORITHM



```

    // If we reach here, then element was not present
    return -1;
}

// Driver code
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    if(result == -1) cout << "Element is not present in array";
    else cout << "Element is present at index " << result;
    return 0;
}
```

ITERATIVE BINARY SEARCH ALGORITHM



Output

```
Element is present at index 3
```

Time Complexity: O(log N)

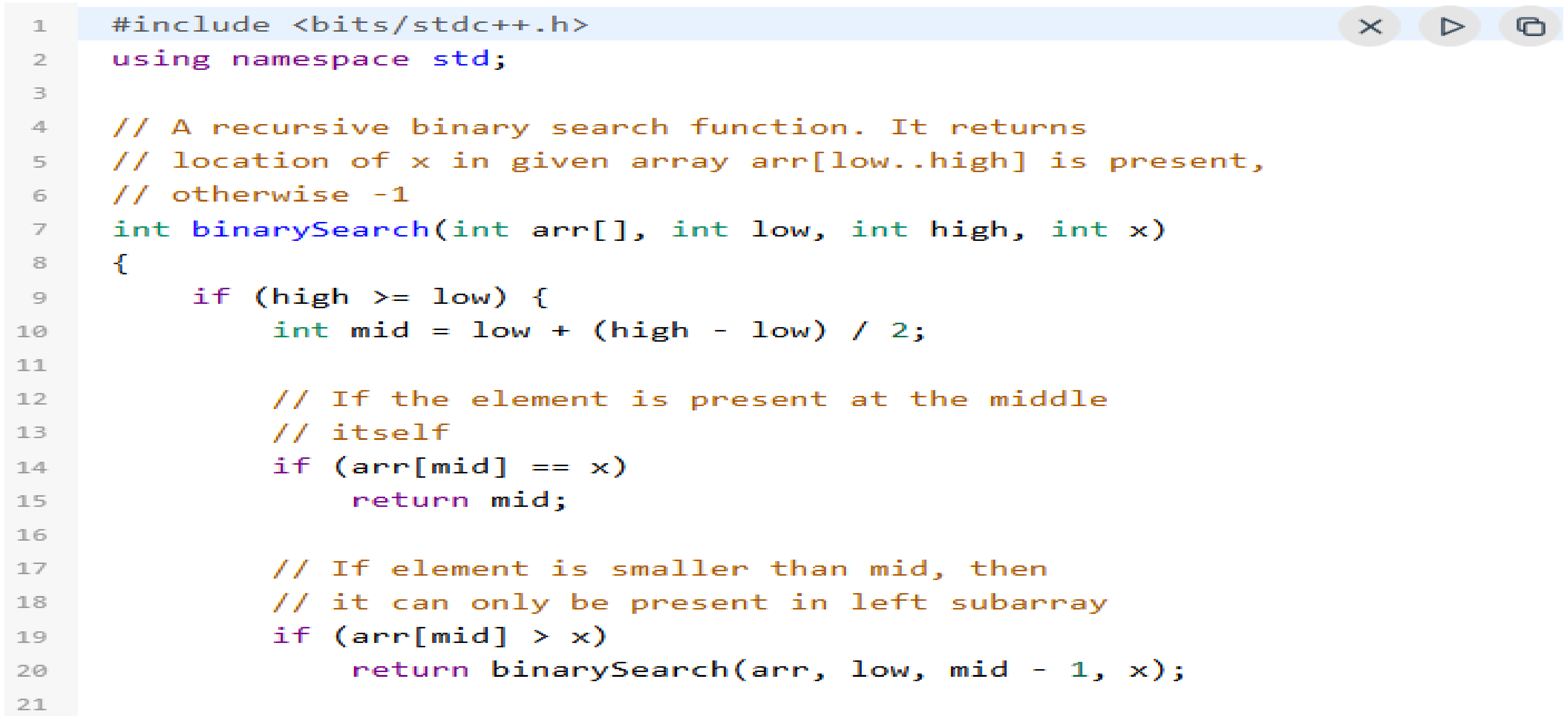
Auxiliary Space: O(1)

RECURSIVE BINARY SEARCH ALGORITHM

- Create a recursive function and compare the mid of the search space with the key.
- And based on the result either return the index where the key is found or call the recursive function for the next search space

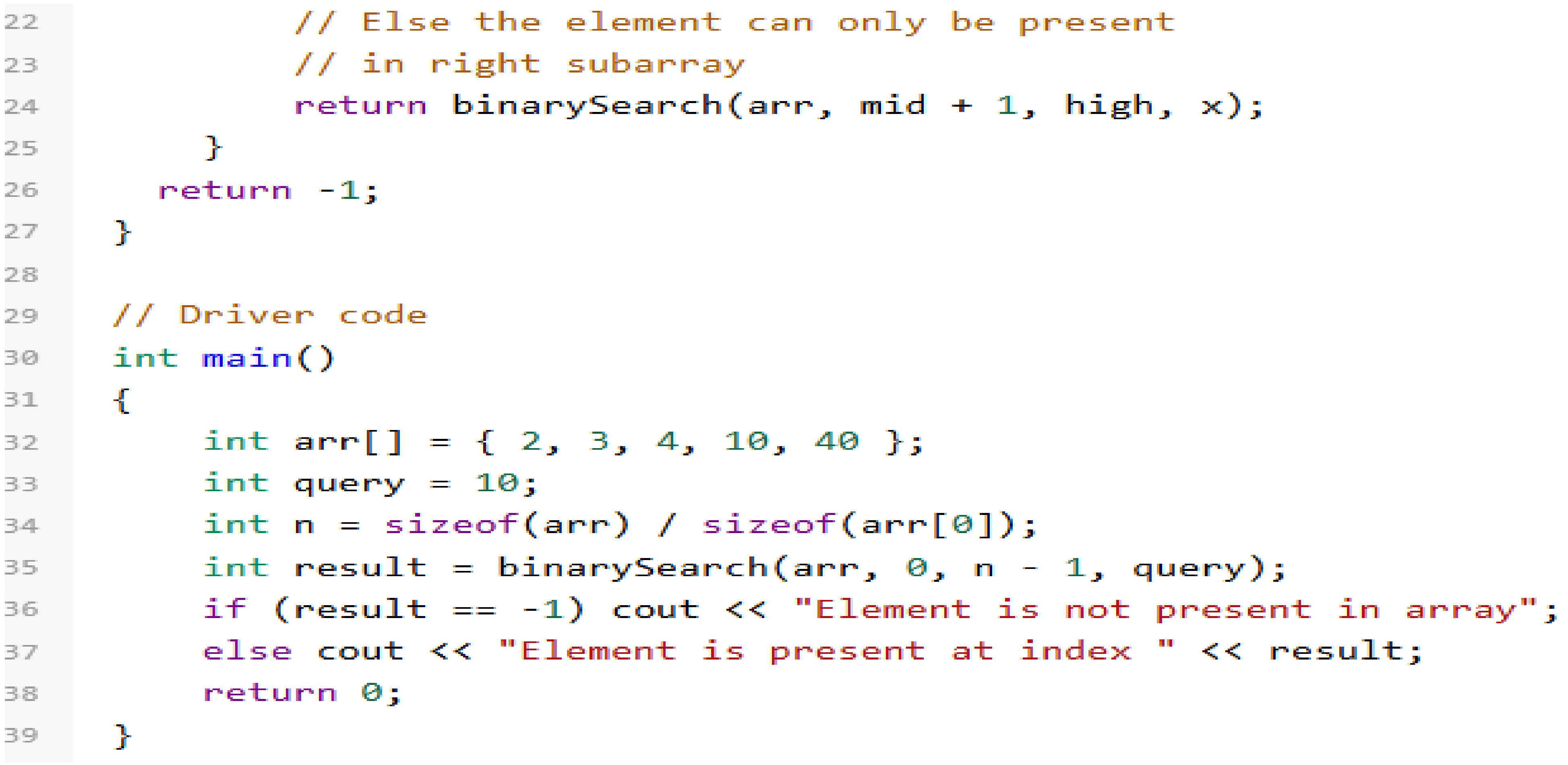
- Recursive binary search algorithm **repeatedly divides the search interval in half**. It's implemented recursively.

RECURSIVE BINARY SEARCH ALGORITHM



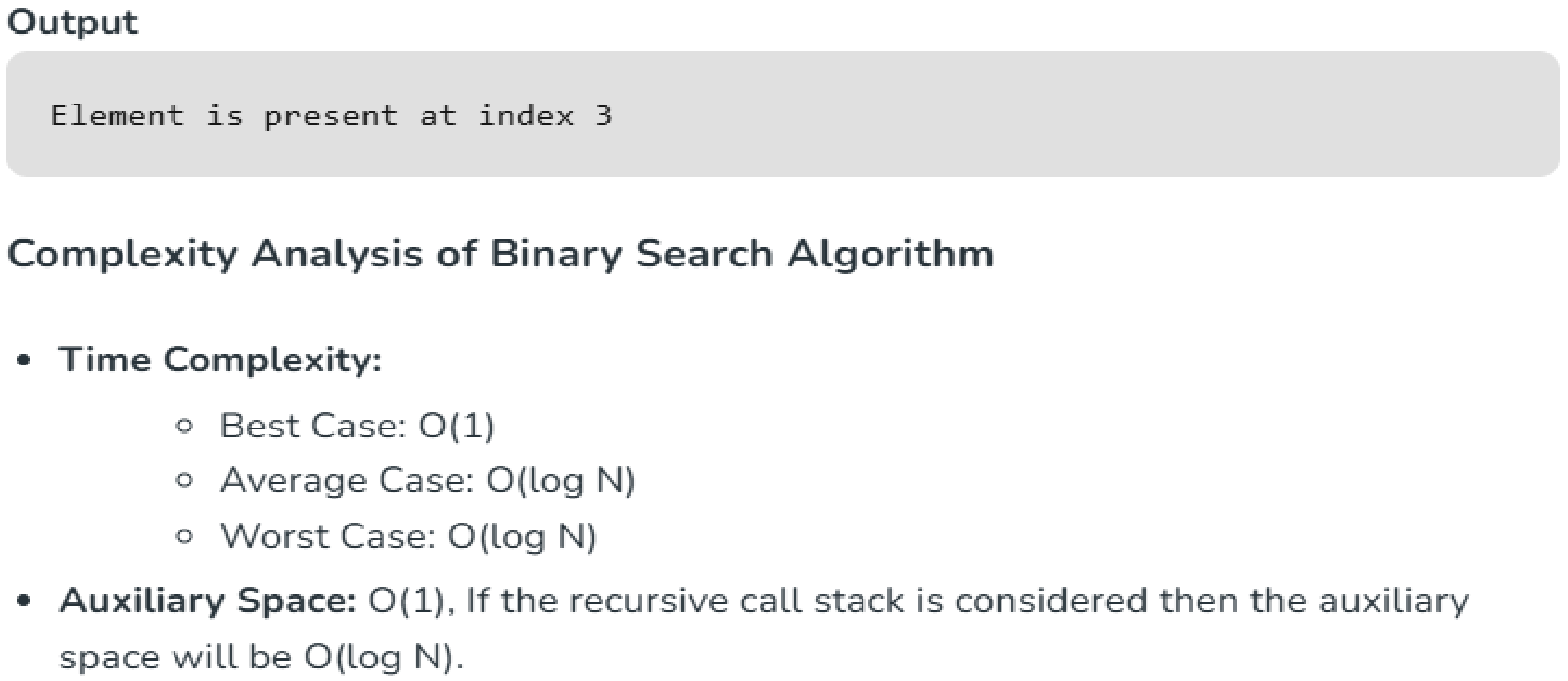
```
#include <bits/stdc++.h>
using namespace std;

// A recursive binary search function. It returns
// location of x in given array arr[low..high] is present,
// otherwise -1
int binarySearch(int arr[], int low, int high, int x)
{
    if (high >= low) {
        int mid = low + (high - low) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, low, mid - 1, x);

```

RECURSIVE BINARY SEARCH ALGORITHM



```
            // Else the element can only be present
            // in right subarray
            return binarySearch(arr, mid + 1, high, x);
        }
    return -1;
}

// Driver code
int main()
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int query = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, query);
    if (result == -1) cout << "Element is not present in array";
    else cout << "Element is present at index " << result;
    return 0;
}
```

RECURSIVE BINARY SEARCH ALGORITHM



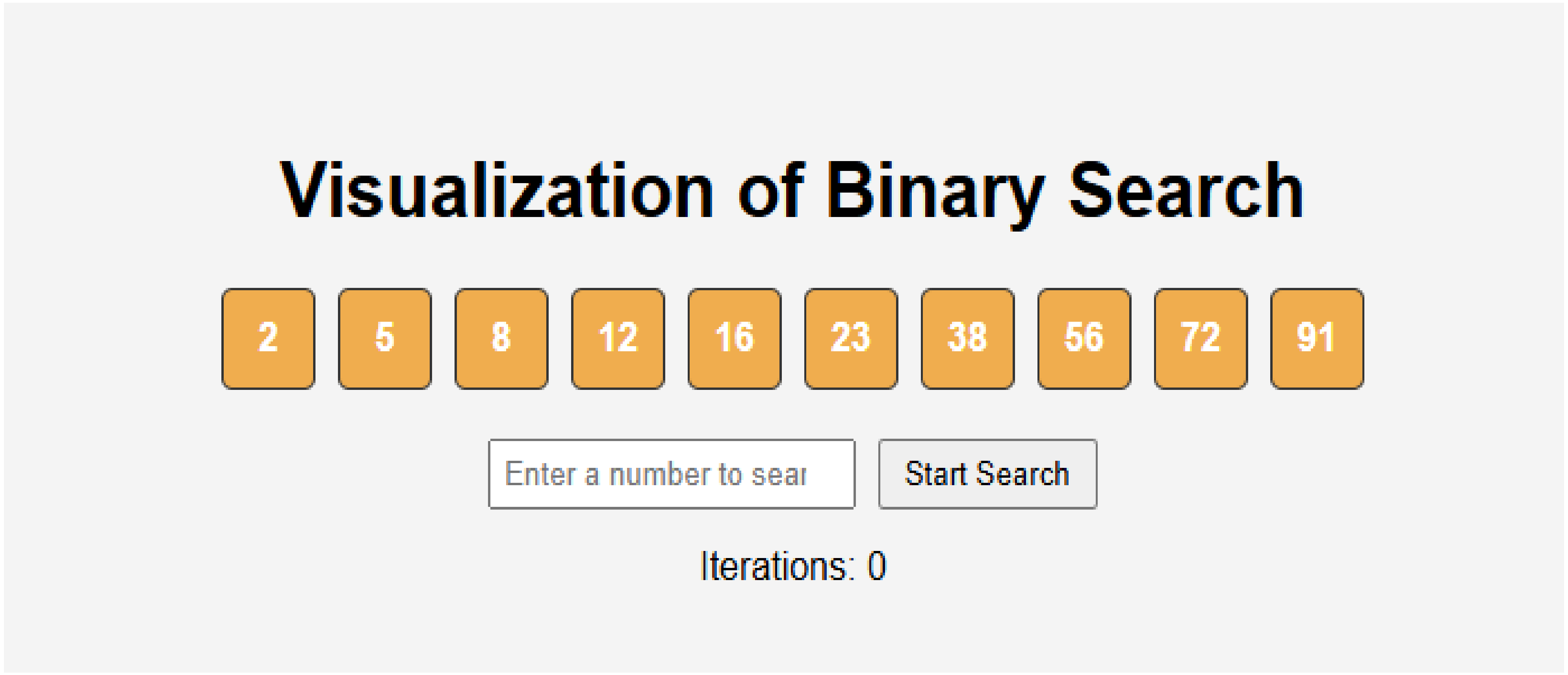
Output

```
Element is present at index 3
```

Complexity Analysis of Binary Search Algorithm

- **Time Complexity:**
    - Best Case: O(1)
    - Average Case: O(log N)
    - Worst Case: O(log N)
- **Auxiliary Space:** O(1), If the recursive call stack is considered then the auxiliary space will be O(log N).

BINARY SEARCH VISUALIZER



APPLICATIONS OF BINARY SEARCH ALGORITHM

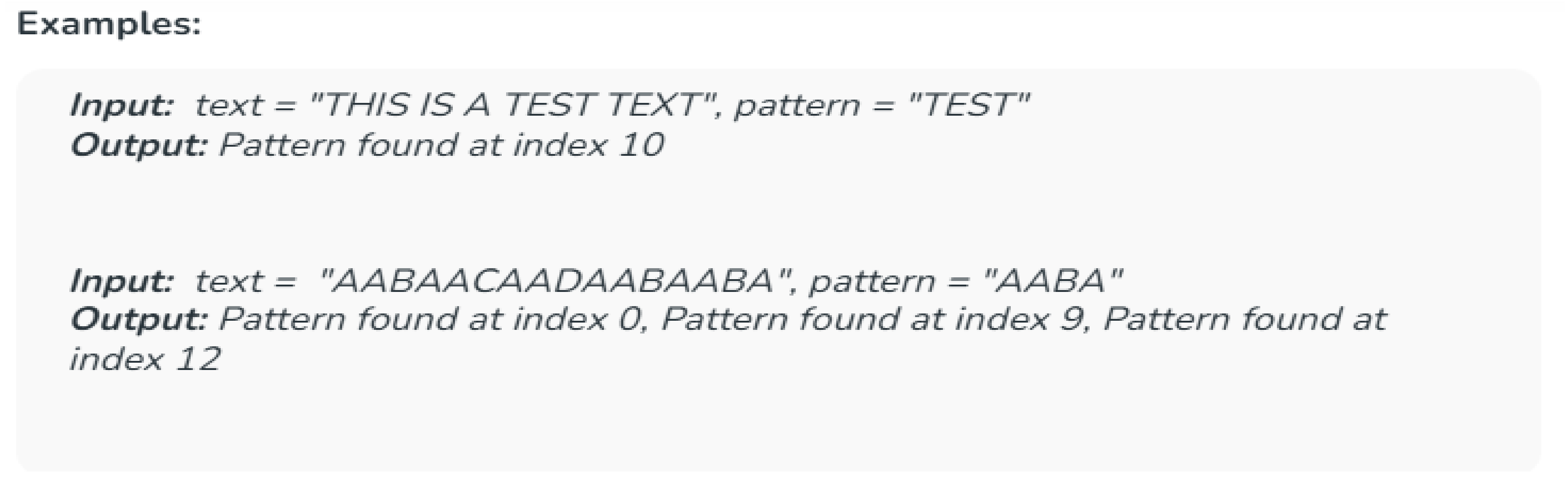
- Binary search can be used as a building block for more complex algorithms used in machine learning, such as algorithms for training neural networks or finding the optimal hyperparameters for a model.
- It can be used for searching in computer graphics such as algorithms for ray tracing or texture mapping.
- It can be used for searching a database.

THE BRUTE FORCE STRING MATCHING ALGORITHM

- The Brute Force string matching algorithm, also known as the naive algorithm, compares a pattern string with all possible substrings of a larger text string.
- It works by sliding the pattern across the text, character by character, and checking for matches. If a mismatch is found, the pattern is shifted one position to the right in the text.
- This process continues until a match is found or the pattern reaches the end of the text.

NAIVE ALGORITHM FOR PATTERN SEARCHING

- Given text string with length n and a pattern with length m, the task is to prints all occurrences of pattern in text.
- Note: You may assume that n > m.

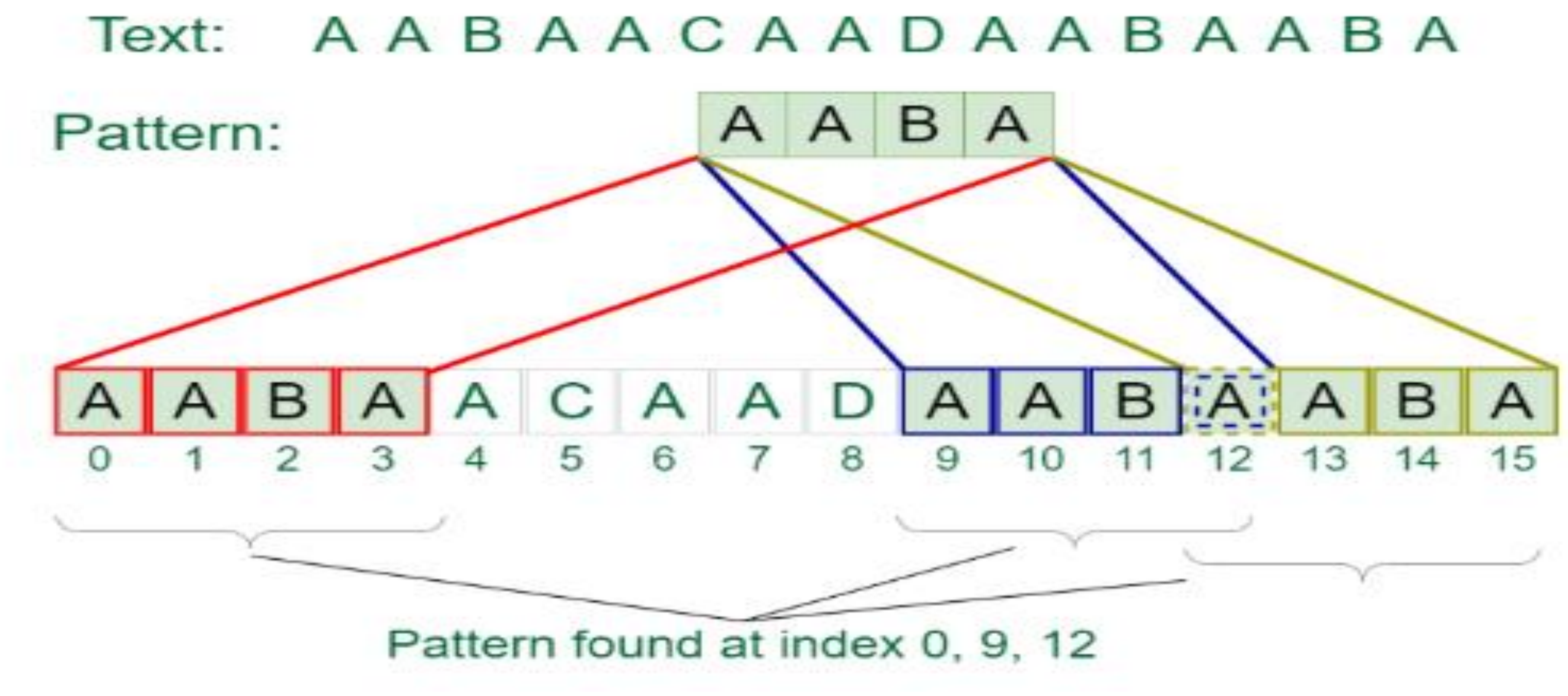


Examples:

Input: *text = "THIS IS A TEST TEXT", pattern = "TEST"*
Output: *Pattern found at index 10*

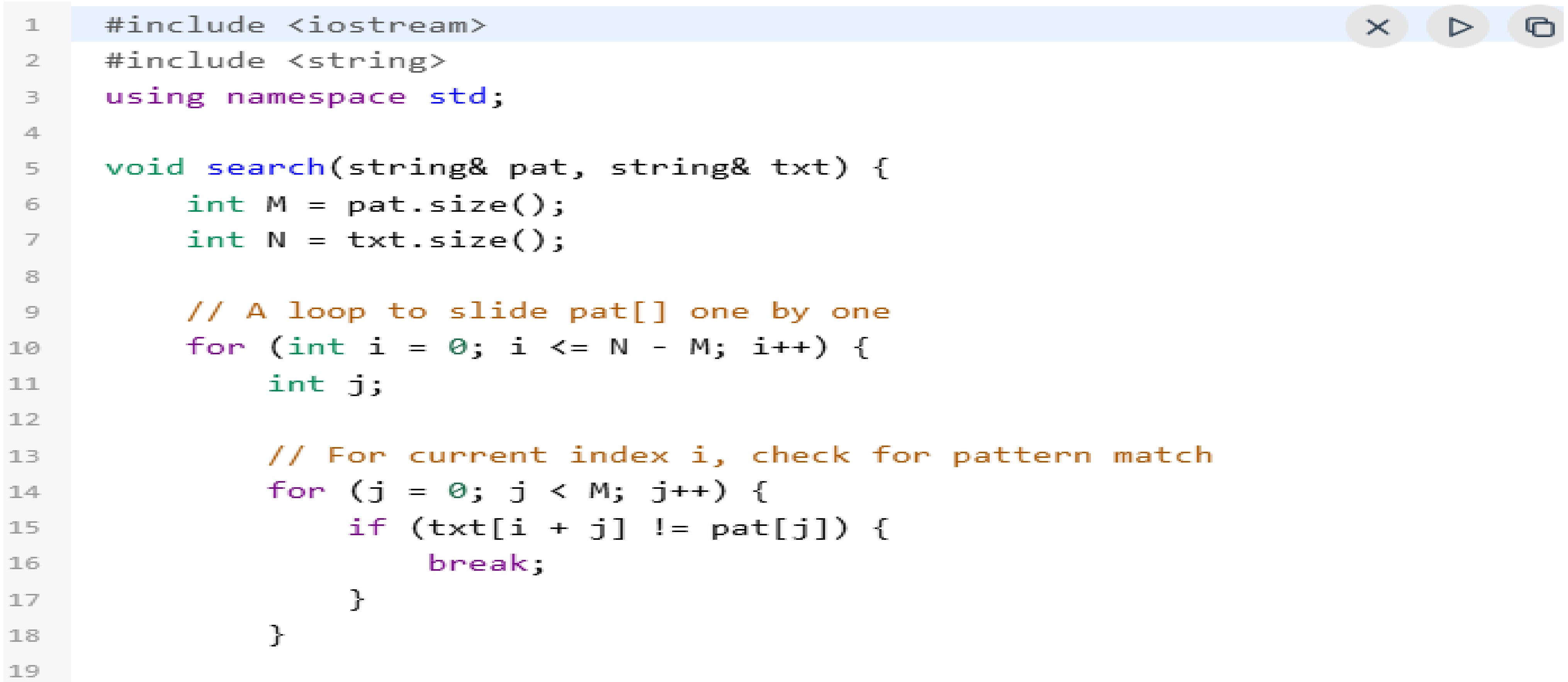
Input: *text = "AABAACAADAABAABA", pattern = "AABA"*
Output: *Pattern found at index 0, Pattern found at index 9, Pattern found at index 12*

NAIVE ALGORITHM FOR PATTERN SEARCHING


Text: A A B A A C A A D A A B A A B A
Pattern: A A B A
A A B A A C A A D A A B A A B A
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Pattern found at index 0, 9, 12

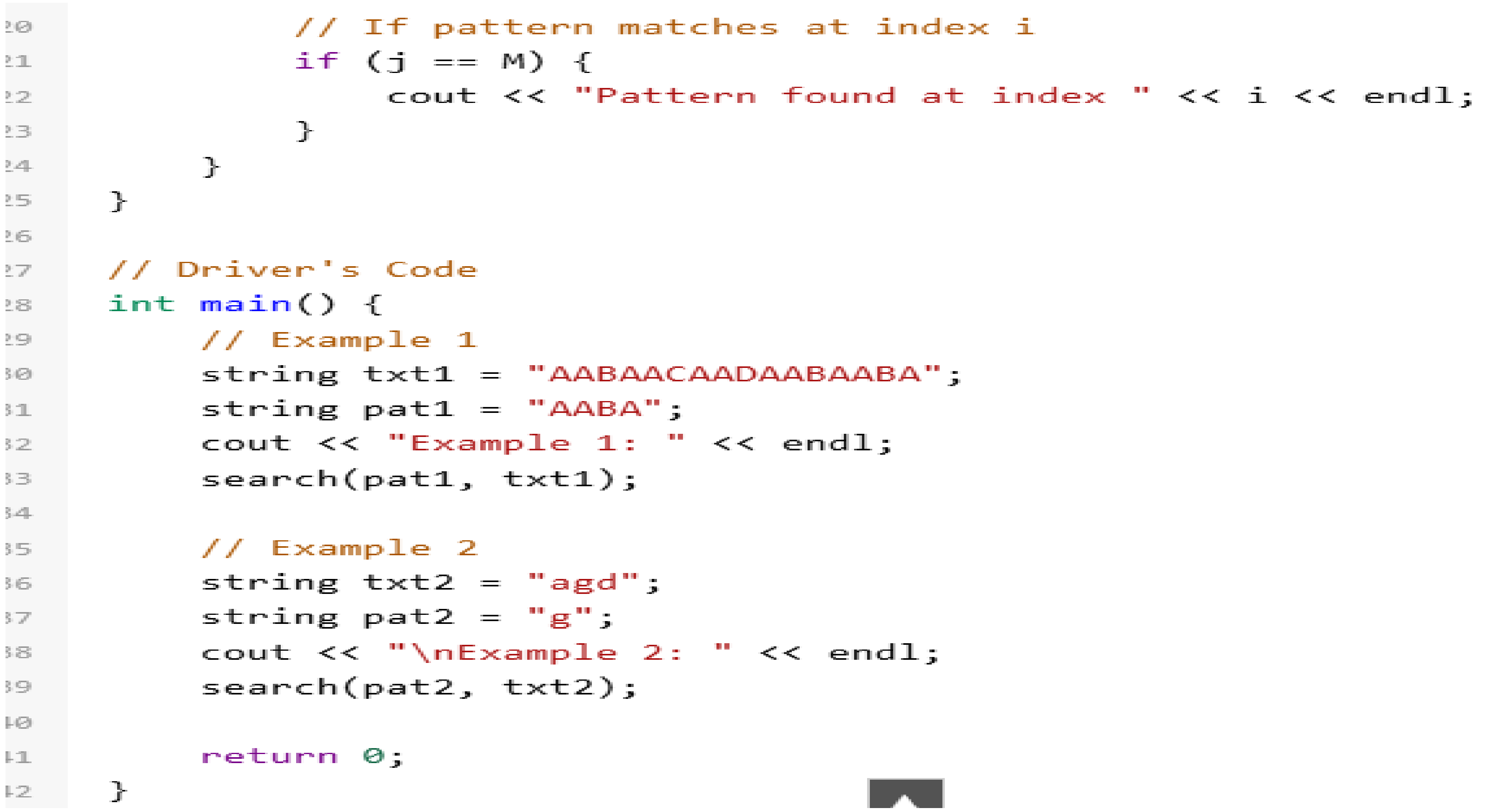
NAIVE PATTERN SEARCHING ALGORITHM

- Slide the pattern over text one by one and check for a match. If a match is found, then slide by 1 again to check for subsequent matches.



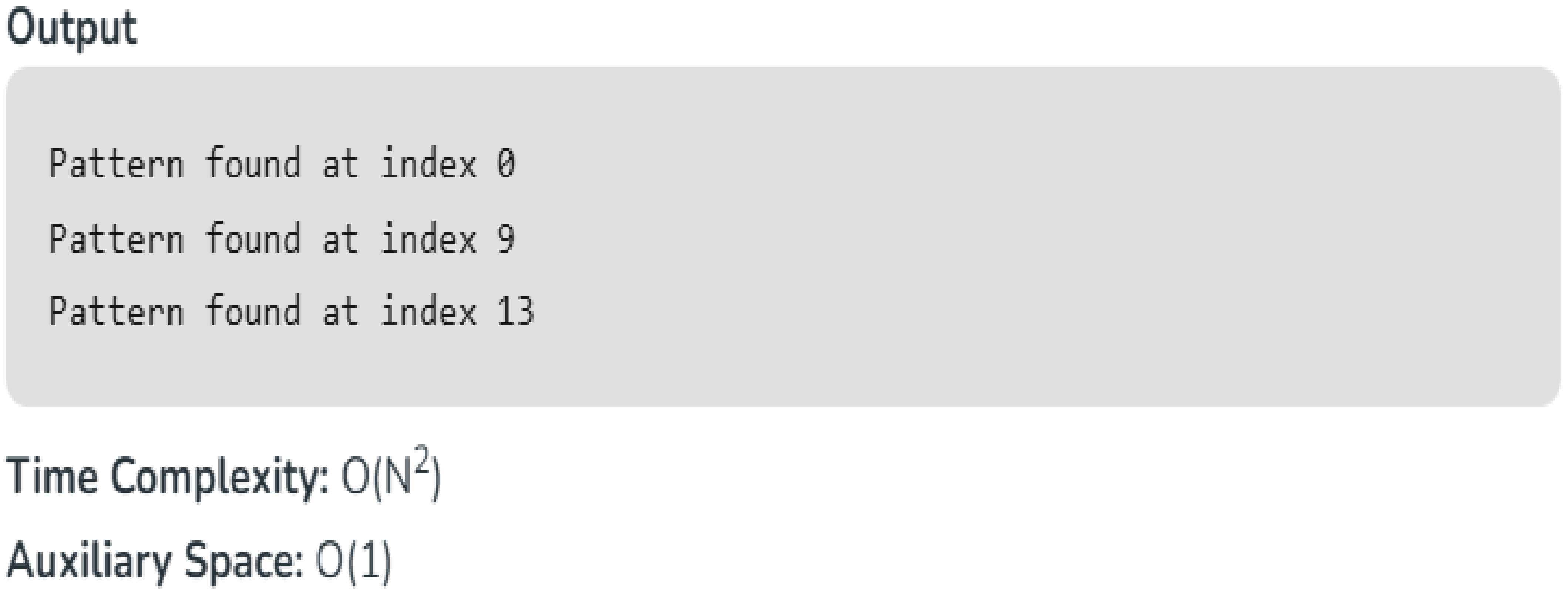
```
#include <iostream>
#include <string>
using namespace std;

void search(string& pat, string& txt) {
    int M = pat.size();
    int N = txt.size();

    // A loop to slide pat[] one by one
    for (int i = 0; i <= N - M; i++) {
        int j;

        // For current index i, check for pattern match
        for (j = 0; j < M; j++) {
            if (txt[i + j] != pat[j]) {
                break;
            }
        }

```

NAIVE PATTERN SEARCHING ALGORITHM



```cpp
        // If pattern matches at index i
        if (j == M) {
            cout << "Pattern found at index " << i << endl;
        }
    }
}

// Driver's Code
int main() {
    // Example 1
    string txt1 = "AABAACAADAABAABA";
    string pat1 = "AABA";
    cout << "Example 1: " << endl;
    search(pat1, txt1);

    // Example 2
    string txt2 = "agd";
    string pat2 = "g";
    cout << "\nExample 2: " << endl;
    search(pat2, txt2);

    return 0;
}
```

NAIVE PATTERN SEARCHING ALGORITHM



Output

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

Time Complexity: O(N^2)

Auxiliary Space: O(1)

COMPLEXITY ANALYSIS OF NAIVE ALGORITHM FOR PATTERN SEARCHING

Best Case: O(n)

- When the pattern is found at the very beginning of the text (or very early on).
- The algorithm will perform a constant number of comparisons, typically on the order of O(n) comparisons, where n is the length of the pattern.

COMPLEXITY ANALYSIS OF NAIVE ALGORITHM FOR PATTERN SEARCHING

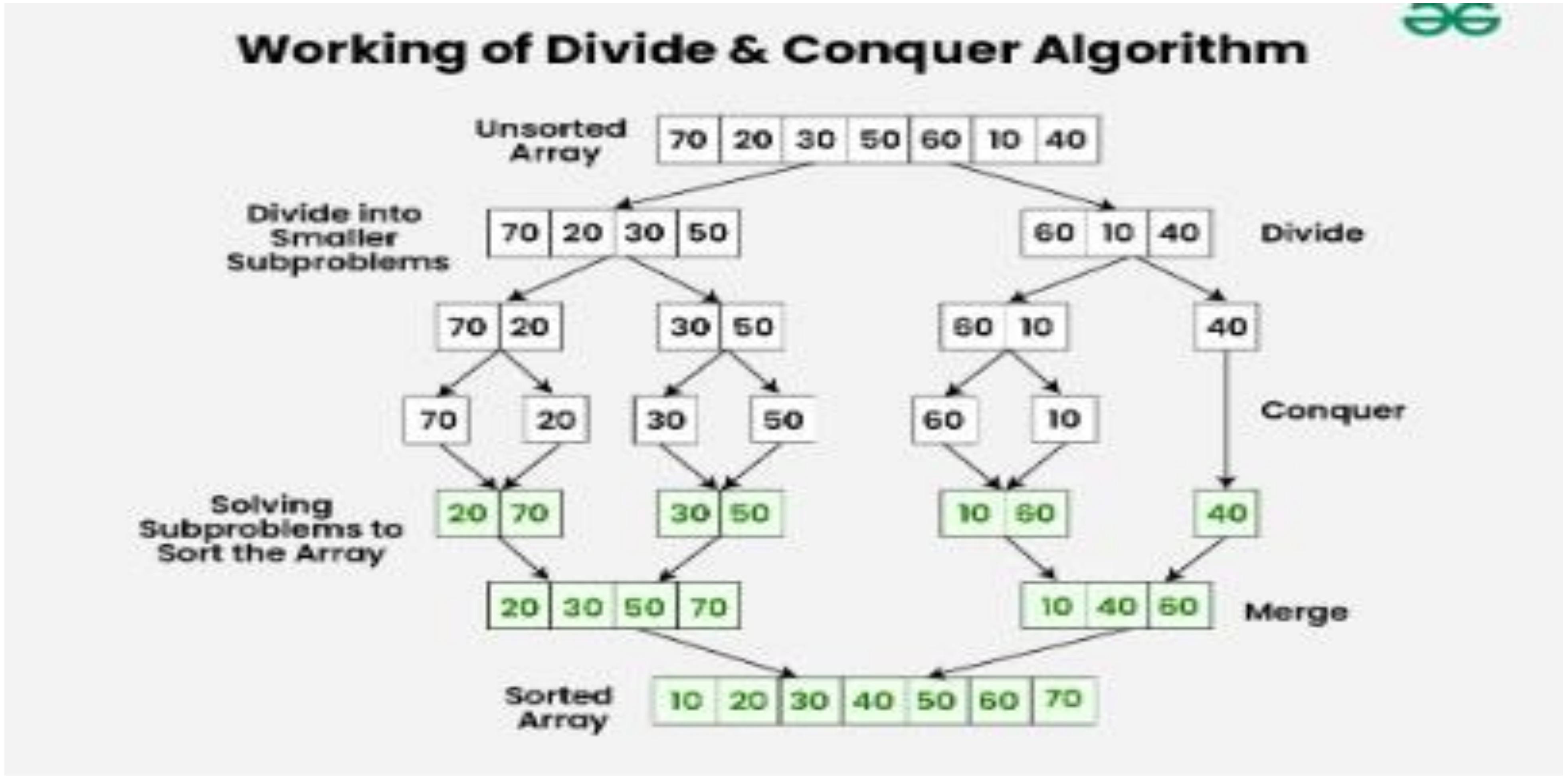
Worst Case: O(n2)

- When the pattern doesn't appear in the text at all or appears only at the very end.
- The algorithm will perform O((n-m+1)*m) comparisons, where n is the length of the text and m is the length of the pattern.
- In the worst case, for each position in the text, the algorithm may need to compare the entire pattern against the text.

DIVIDE AND CONQUER ALGORITHM

- Divide and Conquer Algorithm is a problem-solving technique used to solve problems by dividing the main problem into subproblems, solving them individually and then merging them to find solution to the original problem.
- Divide and Conquer is mainly useful when we divide a problem into independent subproblems. If we have overlapping subproblems, then we use Dynamic Programming.

WORKING OF DIVIDE AND CONQUER ALGORITHM



WORKING OF DIVIDE AND CONQUER ALGORITHM

- The above diagram shows working with the example of Merge Sort which is used for sorting.

Divide:

- Break down the original problem into smaller subproblems.
- Each subproblem should represent a part of the overall problem.
- The goal is to divide the problem until no further division is possible.

In Merge Sort, we divide the input array in two halves. Please note that the divide step of Merge Sort is simple, but in Quick Sort, the divide step is critical. In Quick Sort, we partition the array around a pivot.

WORKING OF DIVIDE AND CONQUER ALGORITHM

Conquer:

- Solve each of the smaller subproblems individually.
- If a subproblem is small enough (often referred to as the “base case”), we solve it directly without further recursion.
- The goal is to find solutions for these subproblems independently.

In Merge Sort, the conquer step is to sort the two halves individually.

WORKING OF DIVIDE AND CONQUER ALGORITHM

Merge:

- Combine the sub-problems to get the final solution of the whole problem.
- Once the smaller subproblems are solved, we recursively combine their solutions to get the solution of larger problem.
- The goal is to formulate a solution for the original problem by merging the results from the subproblems.

In Merge Sort, the merge step is to merge two sorted halves to create one sorted array. Please note that the merge step of Merge Sort is critical, but in Quick Sort, the merge step does not do anything as both parts become sorted in place and the left part has all elements smaller (or equal(than the right part.

CHARACTERISTICS OF DIVIDE AND CONQUER ALGORITHM

- Dividing the Problem: The first step is to break the problem into smaller, more manageable subproblems. This division can be done recursively until the subproblems become simple enough to solve directly.
- Independence of Subproblems: Each subproblem should be independent of the others, meaning that solving one subproblem does not depend on the solution of another. This allows for parallel processing or concurrent execution of subproblems, which can lead to efficiency gains.

CHARACTERISTICS OF DIVIDE AND CONQUER ALGORITHM

- Conquering Each Subproblem: Once divided, the subproblems are solved individually. This may involve applying the same divide and conquer approach recursively until the subproblems become simple enough to solve directly, or it may involve applying a different algorithm or technique.
- Combining Solutions: After solving the subproblems, their solutions are combined to obtain the solution to the original problem. This combination step should be relatively efficient and straightforward, as the solutions to the subproblems should be designed to fit together seamlessly.

ADVANTAGES OF DIVIDE AND CONQUER ALGORITHM

- Solving difficult problems: Divide and conquer technique is a tool for solving difficult problems conceptually. e.g. Tower of Hanoi puzzle. It requires a way of breaking the problem into sub-problems, and solving all of them as an individual cases and then combining sub- problems to the original problem.
- Algorithm efficiency: The divide-and-conquer algorithm often helps in the discovery of efficient algorithms. It is the key to algorithms like Quick Sort and Merge Sort, and fast Fourier transforms.

ADVANTAGES OF DIVIDE AND CONQUER ALGORITHM

- Parallelism: Normally Divide and Conquer algorithms are used in multi-processor machines having shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors.
- Memory access: These algorithms naturally make an efficient use of memory caches. Since the subproblems are small enough to be solved in cache without using the main memory that is slower one. Any algorithm that uses cache efficiently is called cache oblivious.

DISADVANTAGES OF DIVIDE AND CONQUER ALGORITHM

- Overhead: The process of dividing the problem into subproblems and then combining the solutions can require additional time and resources. This overhead can be significant for problems that are already relatively small or that have a simple solution.
- Complexity: Dividing a problem into smaller subproblems can increase the complexity of the overall solution. This is particularly true when the subproblems are interdependent and must be solved in a specific order.

DISADVANTAGES OF DIVIDE AND CONQUER ALGORITHM

- Difficulty of implementation: Some problems are difficult to divide into smaller subproblems or require a complex algorithm to do so. In these cases, it can be challenging to implement a divide and conquer solution.
- Memory limitations: When working with large data sets, the memory requirements for storing the intermediate results of the subproblems can become a limiting factor.

TREE TRAVERSAL

- Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node.
- That is, we cannot randomly access a node in a tree.

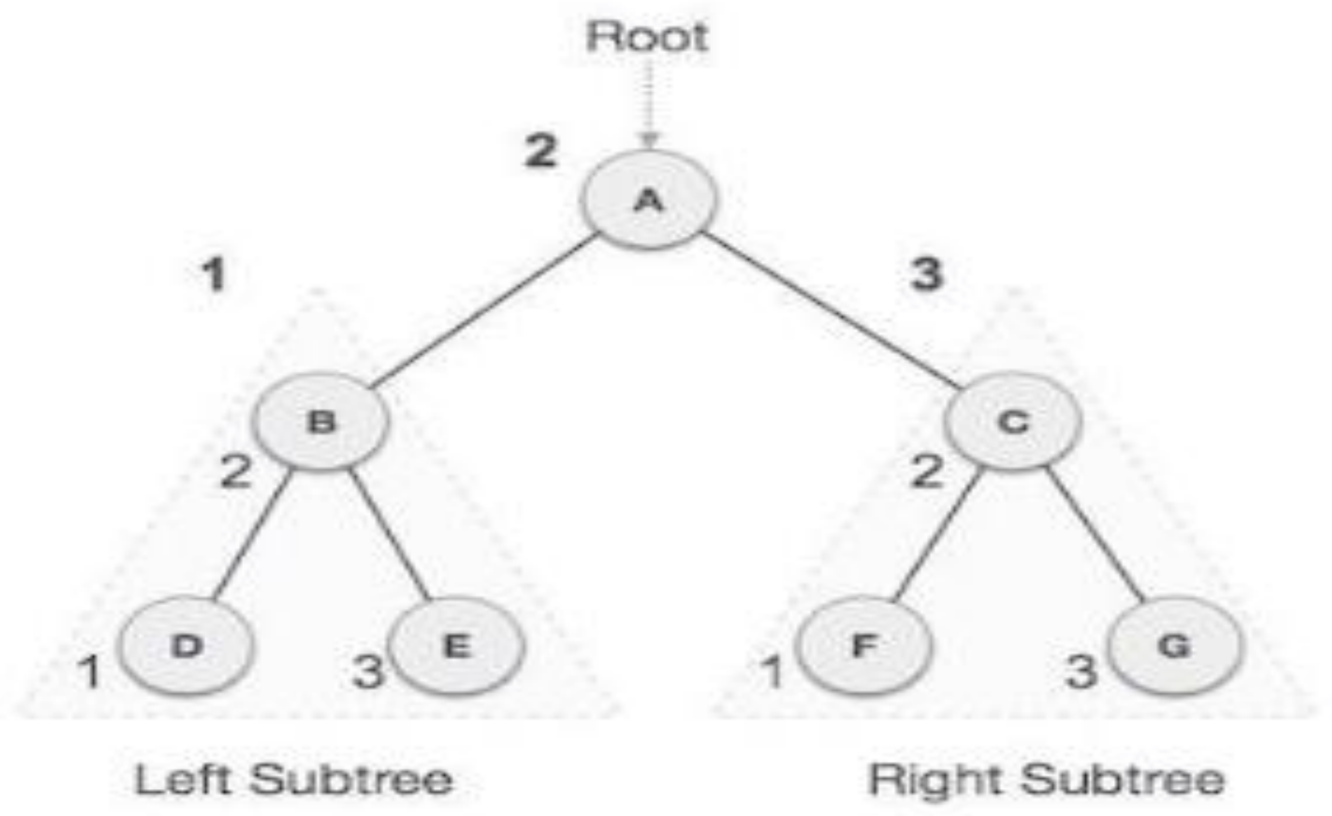
TREE TRAVERSAL

- There are three ways which we use to traverse a tree –
    i. In-order Traversal
    ii. Pre-order Traversal
    iii. Post-order Traversal

IN-ORDER TRAVERSAL

- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.
- We should always remember that every node may represent a subtree itself.
- If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.

IN-ORDER TRAVERSAL

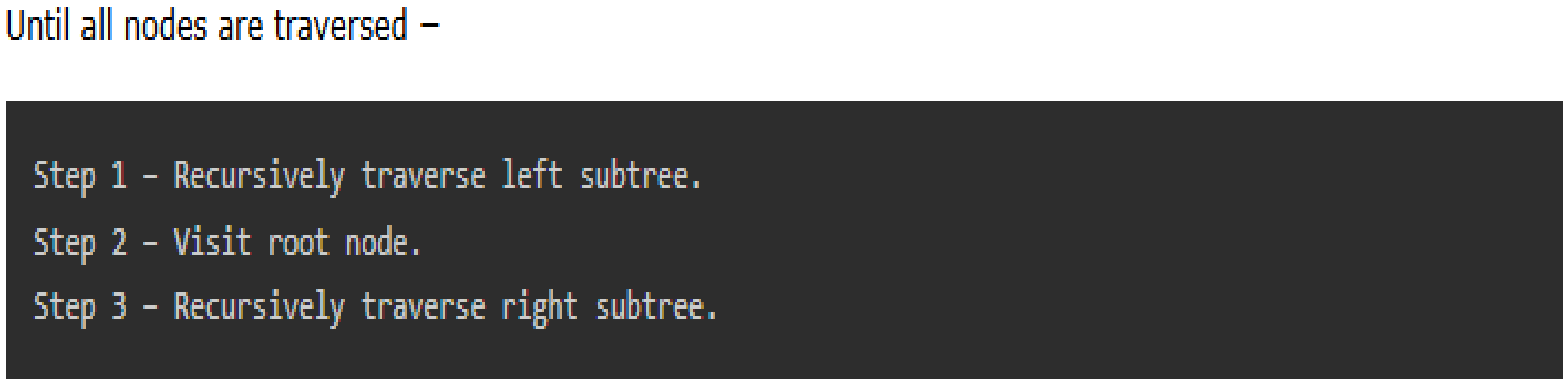


IN-ORDER TRAVERSAL

- We start from A, and following in-order traversal, we move to its left subtree B.B is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be −

D B E A F C G

ALGORITHM

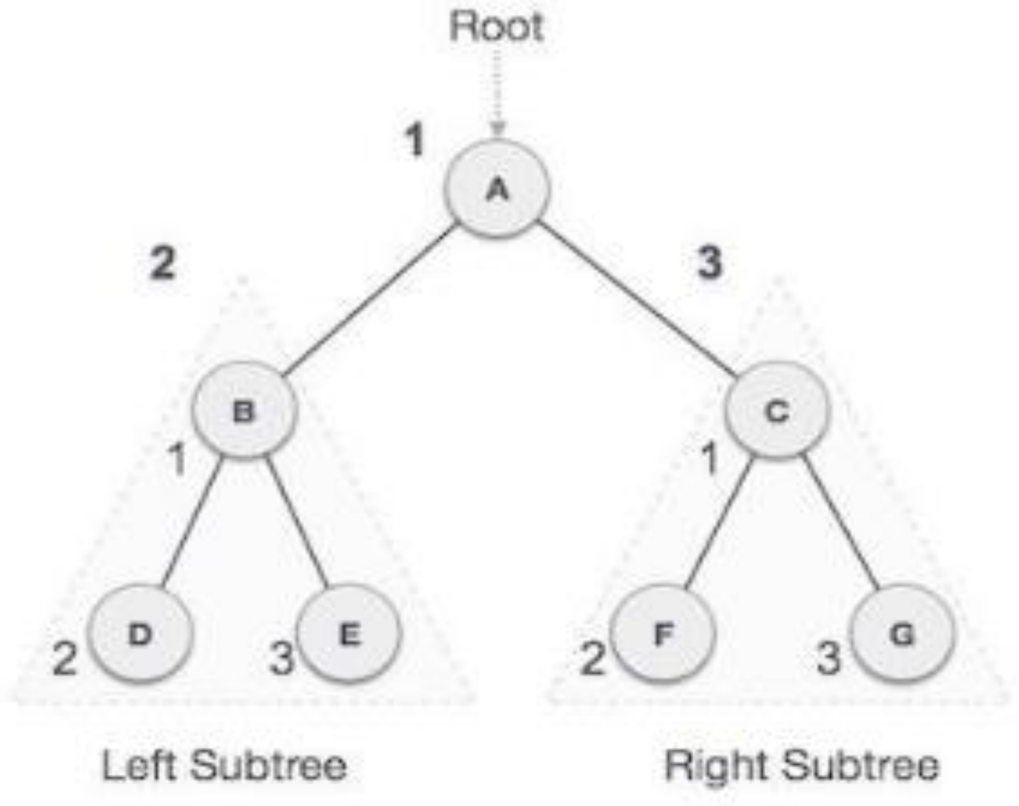


Until all nodes are traversed −

```
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.
```

PRE-ORDER TRAVERSAL

- In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



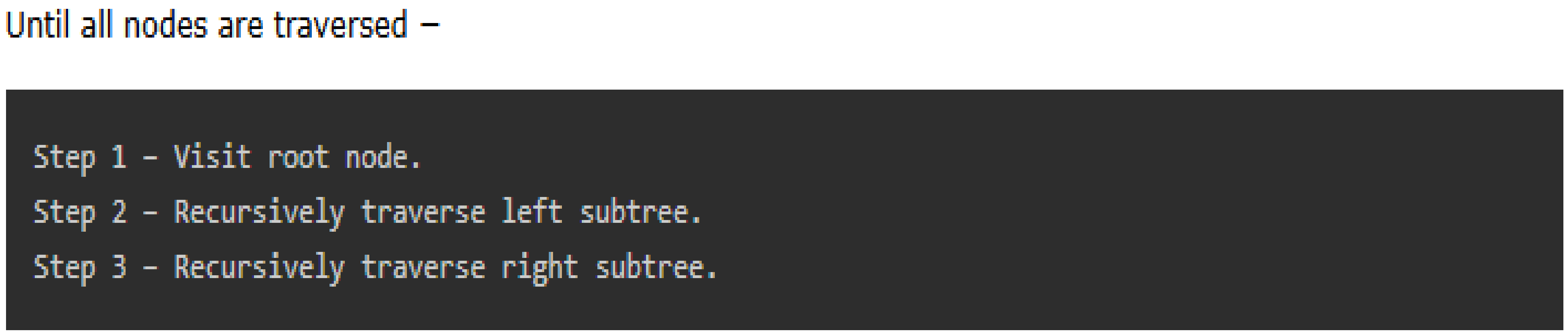
PRE-ORDER TRAVERSAL

- We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

ABDECFG

A → B → D → E → C → F → G

ALGORITHM

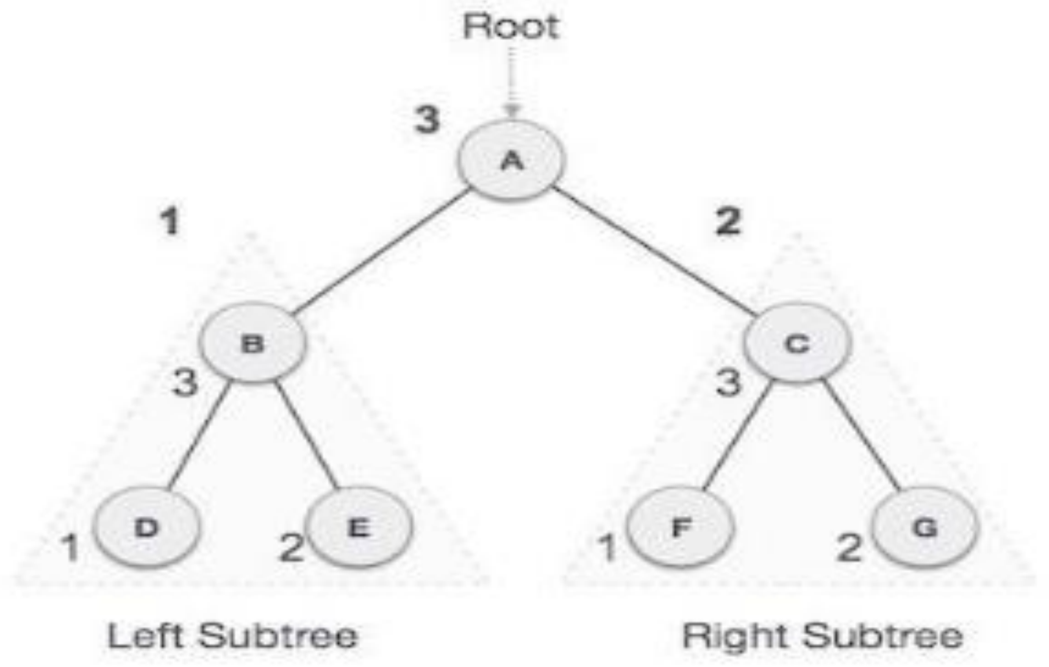


Until all nodes are traversed −

```
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.
```

POST-ORDER TRAVERSAL

- In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

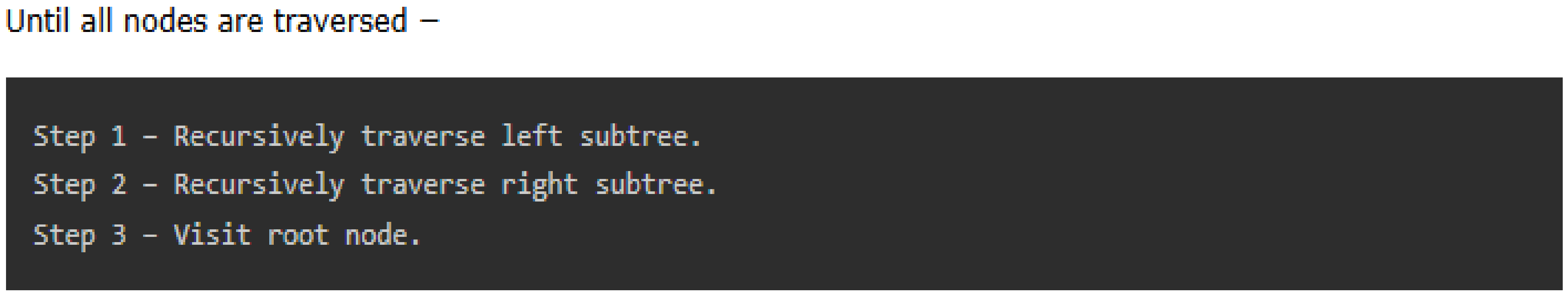


POST-ORDER TRAVERSAL

- We start from A, and following pre-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be

D → E → B → F → G → C → A

ALGORITHM



Until all nodes are traversed −

```
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.
```

Structure of Heap and Merge Sort

DECREASE AND CONQUER

- Decrease and conquer is a technique used to solve problems by reducing the size of the input data at each step of the solution process.
- This technique is similar to divide-and-conquer, in that it breaks down a problem into smaller subproblems, but the difference is that in decrease-and-conquer, the size of the input data is reduced at each step.

DECREASE AND CONQUER

- The technique is used when it's easier to solve a smaller version of the problem, and the solution to the smaller problem can be used to find the solution to the original problem.
  1. Some examples of problems that can be solved using the decrease-and-conquer technique include binary search, finding the maximum or minimum element in an array, and finding the closest pair of points in a set of points.
  2. The main advantage of decrease-and-conquer is that it often leads to efficient algorithms, as the size of the input data is reduced at each step, reducing the time and space complexity of the solution. However, it's important to choose the right strategy for reducing the size of the input data, as a poor choice can lead to an inefficient algorithm.

IMPLEMENTATIONS OF DECREASE AND CONQUER

- This approach can be either implemented as top-down or bottom-up.
    1. Top-down approach : It always leads to the recursive implementation of the problem.
    2. Bottom-up approach : It is usually implemented in iterative way, starting with a solution to the smallest instance of the problem.

VARIATIONS OF DECREASE AND CONQUER

There are three major variations of decrease-and-conquer:

- Decrease by a constant
- Decrease by a constant factor
- Variable size decrease

VARIATIONS OF DECREASE AND CONQUER

There are three major variations of decrease-and-conquer:

- Decrease by a constant
- Decrease by a constant factor
- Variable size decrease

VARIATIONS OF DECREASE AND CONQUER

- Decrease by a Constant : In this variation, the size of an instance is reduced by the same constant on each iteration of the algorithm.
- Typically, this constant is equal to one , although other constant size reductions do happen occasionally.
- Below are example problems :
    i. Insertion sort
    ii. Graph search algorithms: DFS, BFS
    iii. Topological sorting
    iv. Algorithms for generating permutations, subsets

VARIATIONS OF DECREASE AND CONQUER

- Decrease by a Constant factor: This technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm.
- In most applications, this constant factor is equal to two. A reduction by a factor other than two is especially rare. Decrease by a constant factor algorithms are very efficient especially when the factor is greater than 2 as in the fake-coin problem.

Below are example problems :

i. Binary search
ii. Fake-coin problems
iii. Russian peasant multiplication

VARIATIONS OF DECREASE AND CONQUER

- Variable-Size-Decrease : In this variation, the size-reduction pattern varies from one iteration of an algorithm to another. As, in problem of finding gcd of two number though the value of the second argument is always smaller on the right-handside than on the left-hand side, it decreases neither by a constant nor by a constant factor.
- Below are example problems :
    i. Computing median and selection problem.
    ii. Interpolation Search
    iii. Euclid’s algorithm

VARIATIONS OF DECREASE AND CONQUER

- There may be a case that problem can be solved by decrease-by-constant as well as decrease-by-factor variations, but the implementations can be either recursive or iterative.
- The iterative implementations may require more coding effort, however they avoid the overload that accompanies recursion.

ADVANTAGES OF DECREASE AND CONQUER

- Simplicity: Decrease-and-conquer is often simpler to implement compared to other techniques like dynamic programming or divide-and-conquer.
- Efficient Algorithms: The technique often leads to efficient algorithms as the size of the input data is reduced at each step, reducing the time and space complexity of the solution.
- Problem-Specific: The technique is well-suited for specific problems where it's easier to solve a smaller version of the problem

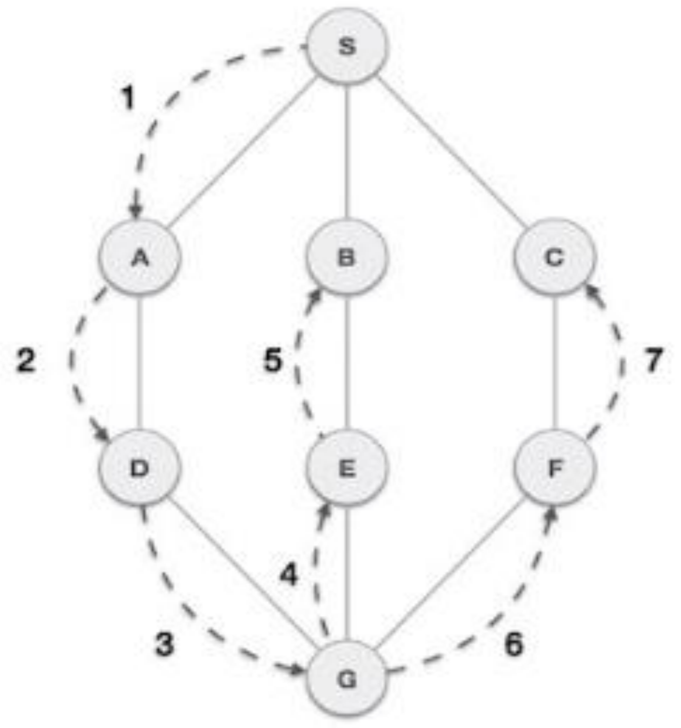
DISADVANTAGES OF DECREASE AND CONQUER

- Problem-Specific: The technique is not applicable to all problems and may not be suitable for more complex problems.
- Implementation Complexity: The technique can be more complex to implement when compared to other techniques like divide-and-conquer, and may require more careful planning.

DEPTH FIRST SEARCH (DFS) ALGORITHM

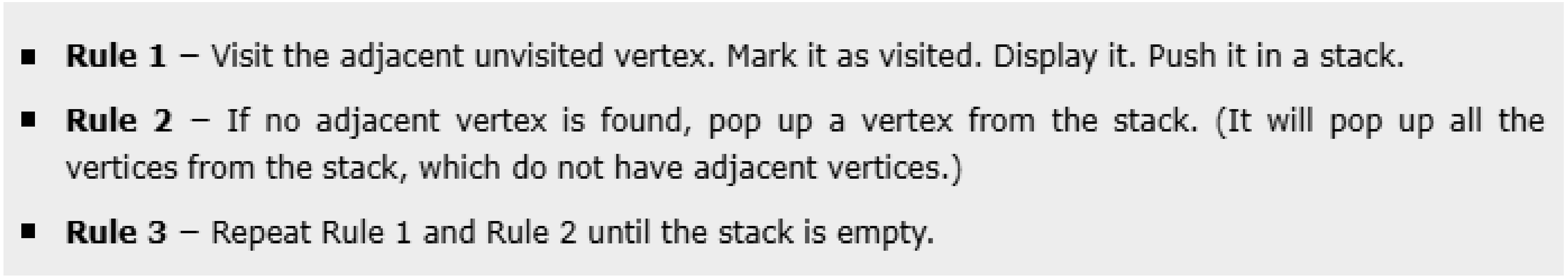
- Depth First Search (DFS) algorithm is a recursive algorithm for searching all the vertices of a graph or tree data structure.
- This algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration

DEPTH FIRST SEARCH (DFS) ALGORITHM



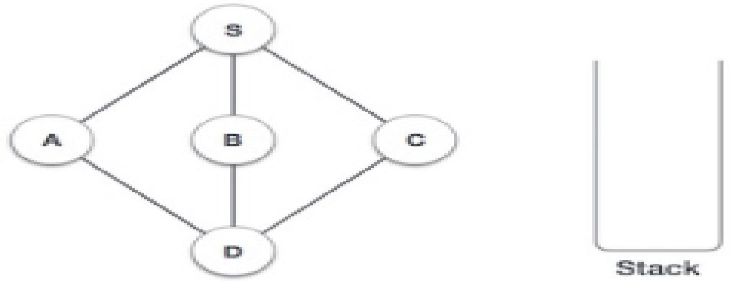
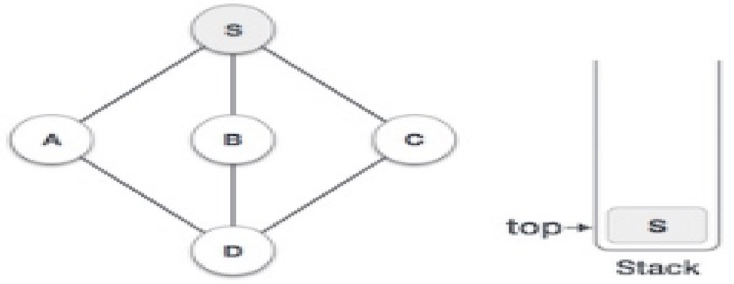
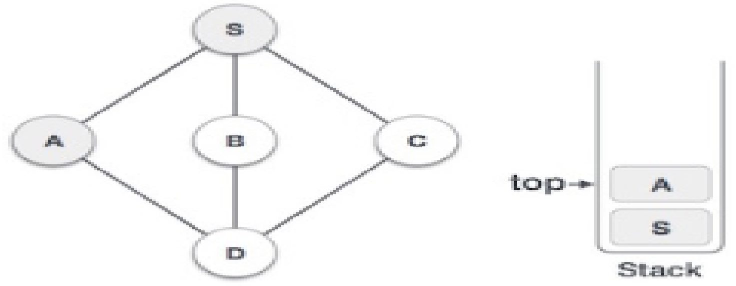
DEPTH FIRST SEARCH (DFS) ALGORITHM

- As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C.
- It employs the following rules.

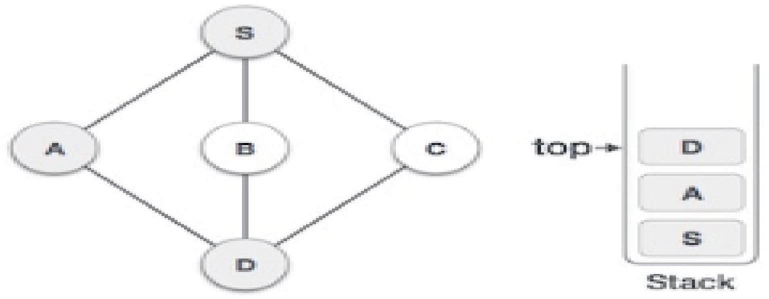
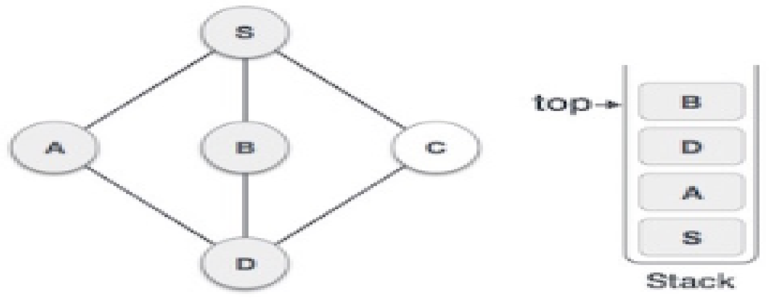
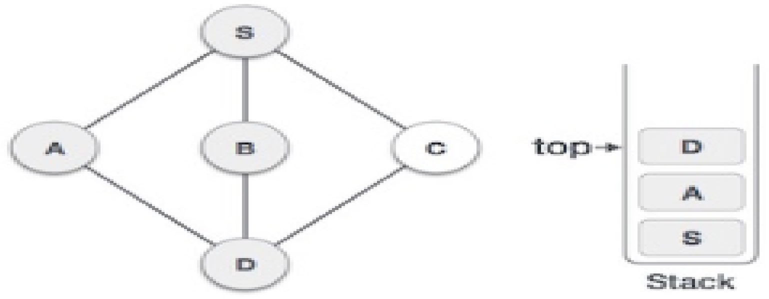


- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

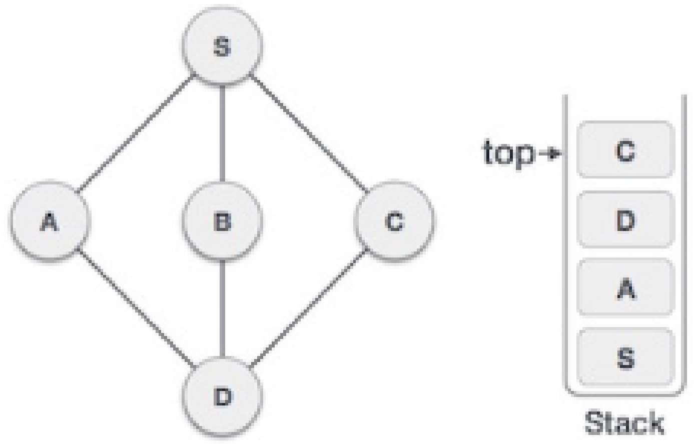
DEPTH FIRST SEARCH (DFS) ALGORITHM

Step	**Traversal**	**Description**
1		Initialize the stack.
2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

DEPTH FIRST SEARCH (DFS) ALGORITHM

4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

DEPTH FIRST SEARCH (DFS) ALGORITHM

7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

COMPLEXITY OF DFS ALGORITHM

Time Complexity:

- The time complexity of the DFS algorithm is represented in the form of $O(V + E)$, where V is the number of nodes and E is the number of edges.

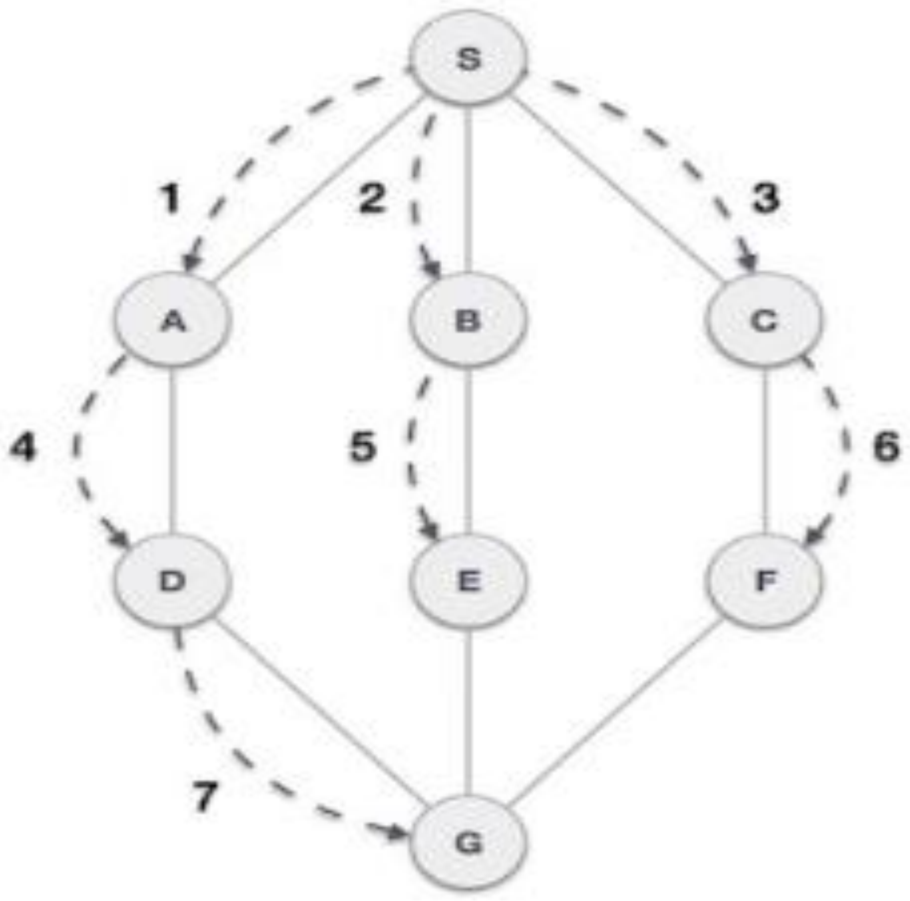
Space Complexity:

- The space complexity of the DFS algorithm is $O(V)$

BREADTH FIRST SEARCH (BFS) ALGORITHM

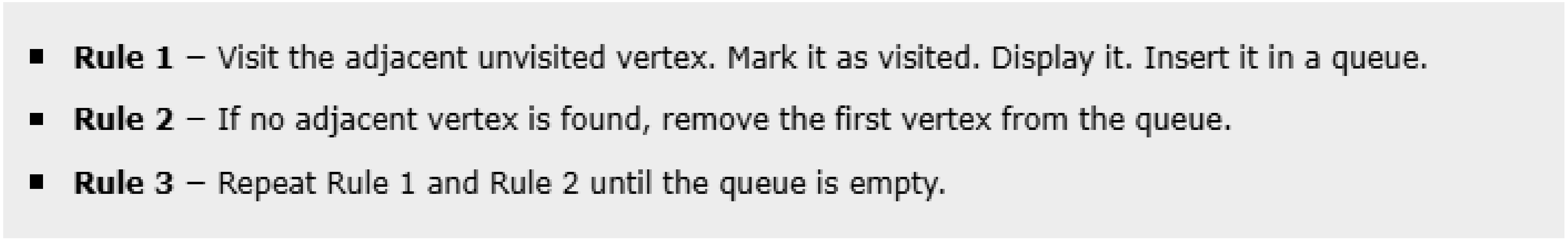
- Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion to search a graph data structure for a node that meets a set of criteria.
- It uses a queue to remember the next vertex to start a search, when a dead end occurs in any iteration.
- Breadth First Search (BFS) algorithm starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.

BREADTH FIRST SEARCH (BFS) ALGORITHM



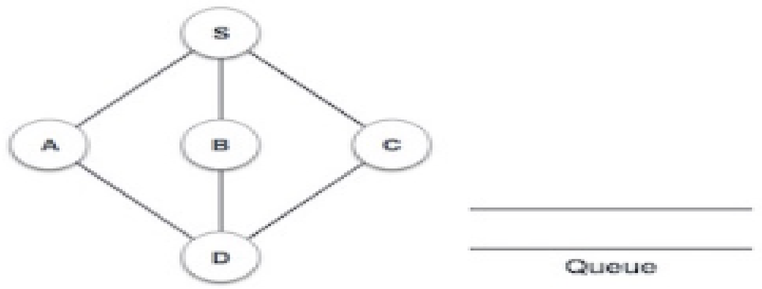
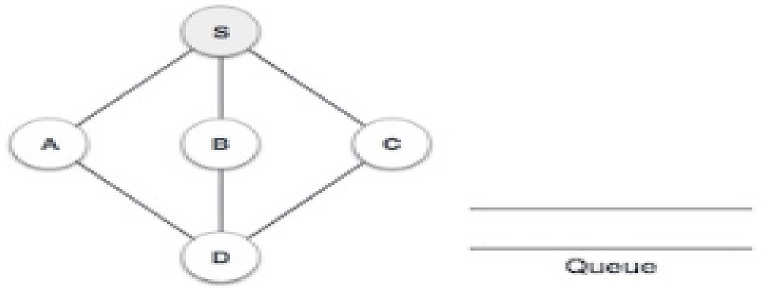
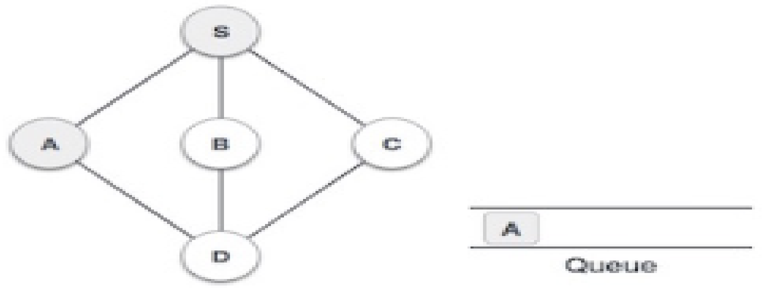
BREADTH FIRST SEARCH (BFS) ALGORITHM

- As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D.
- It employs the following rules.

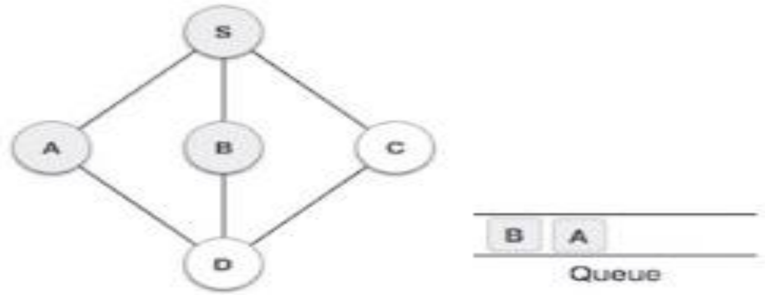
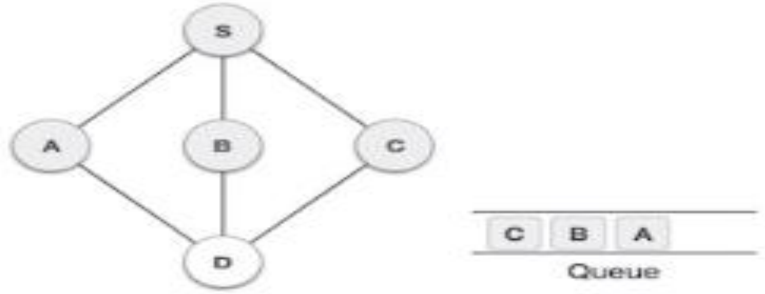
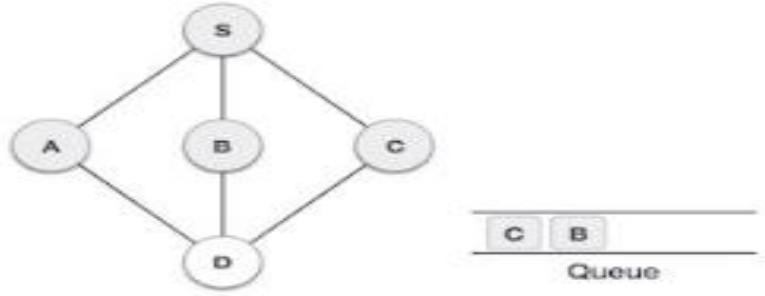


- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

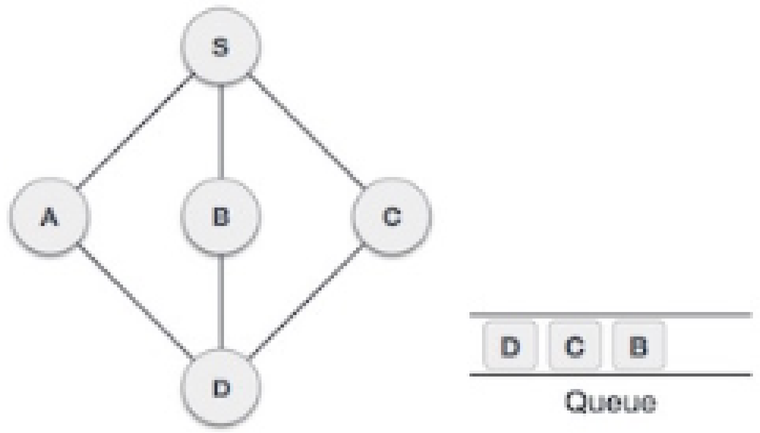
BREADTH FIRST SEARCH (BFS) ALGORITHM

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting **S** (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.

BREADTH FIRST SEARCH (BFS) ALGORITHM

4		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.
6		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.

BREADTH FIRST SEARCH (BFS) ALGORITHM

7		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

COMPLEXITY OF BFS ALGORITHM

Time Complexity

- The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

Space Complexity

- The space complexity of the BFS algorithm is O(V).

Code for Prim Algorithm: