**Searching and Sorting Algorithms**

**Searching:**

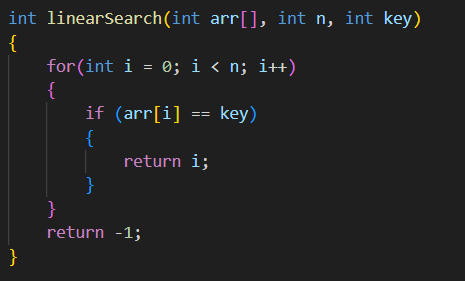
**Linear Search:**

* Defined as a sequential search algorithm
* It starts at one end and goes through each and every element in an array until the element is found.

**Algorithm:**

* **Start:** Begin at the first element of the collection of elements.
* **Compare:** Compare the current element with the desired element.
* **Found:** If the current element is equal to the desired element, return true or index to the current element.
* **Move:** Otherwise, move to the next element in the collection.
* **Repeat:** Repeat steps 2-4 until we have reached the end of collection.
* **Not found:** If the end of the collection is reached without finding the desired element, return that the desired element is not in the array.

**Implementation:**

****

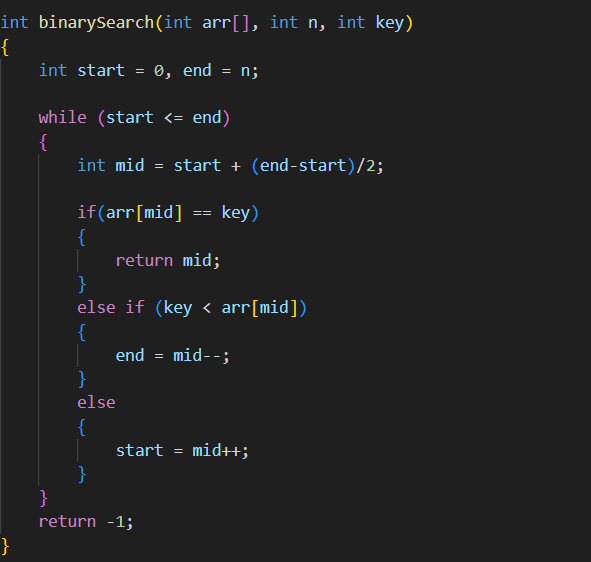
**Binary Search:**

* A searching algorithm used in a *Sorted Array*.
* Used to find the position of an element in an array.
* It works by repeatedly dividing itself in half until the target value is found or the interval is empty.

**Algorithm:**

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

**Implementation:**

****

**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(logN).

**Sorting:**

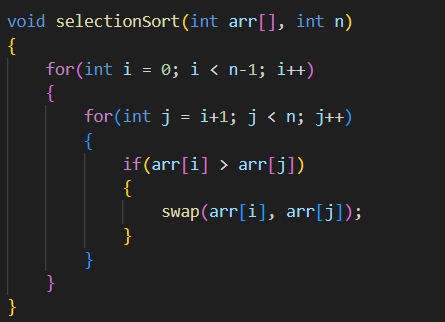
**Selection Sort:**

* Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.
* It is not a stable algorithm.
* It is an in-place algorithm as it does not require extra space.

**Algorithm:**

* Iterate till n-1 elements
* Iterate from i+1 till n elements
* Find the minimum element in the array and swap it the element at ‘i’ index.

**Implementation:**

****

**Time Complexity:**

* The time complexity of Selection Sort is O(N2) as there are two nested loops

**Auxiliary Space:**

* O(1) as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than O(N) swaps and can be useful when memory writing is costly.

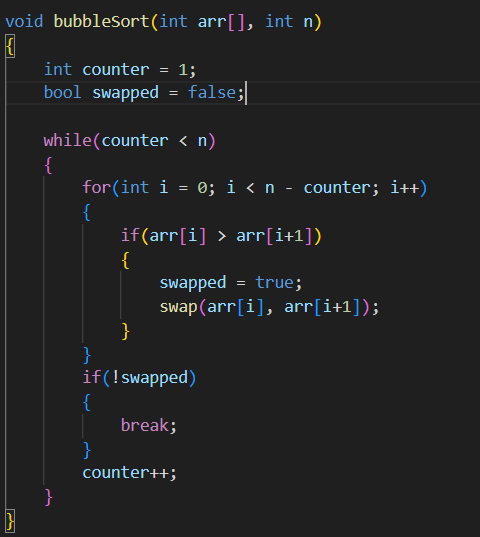
**Bubble Sort:**

* Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.
* It does not require any additional memory space.
* It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

**Algorithm:**

* traverse from left and compare adjacent elements and the higher one is placed at right side.
* In this way, the largest element is moved to the rightmost end at first.
* This process is then continued to find the second largest and place it and so on until the data is sorted.

**Implementation:**

****

**Time Complexity: O(N2)**

**Auxiliary Space: O(1)**

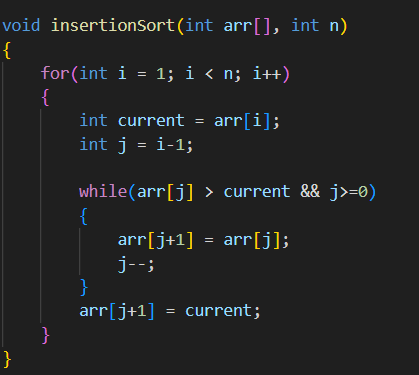
**Insertion Sort:**

* Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list.
* It is a stable sorting algorithm, meaning that elements with equal values maintain their relative order in the sorted output.

**Algorithm:**

* We have to start with second element of the array as first element in the array is assumed to be sorted.
* Compare second element with the first element and check if the second element is smaller then swap them.
* Move to the third element and compare it with the second element, then the first element and swap as necessary to put it in the correct position among the first three elements.
* Continue this process, comparing each element with the ones before it and swapping as needed to place it in the correct position among the sorted elements.
* Repeat until the entire array is sorted.

**Implementation:**



**Time Complexity of Insertion Sort**

* Best case: O(n), If the list is already sorted, where n is the number of elements in the list.
* Average case: O(n2), If the list is randomly ordered
* Worst case: O(n2), If the list is in reverse order

**Space Complexity of Insertion Sort**

Auxiliary Space: O(1), Insertion sort requires O(1) additional space, making it a space-efficient sorting algorithm.