Selection Sort: - The Selection Sort algorithm finds the lowest value in an array and moves it to the front of the array.

The algorithm looks through the array again and again, moving the next lowest values to the front, until the array is sorted.

How it works:

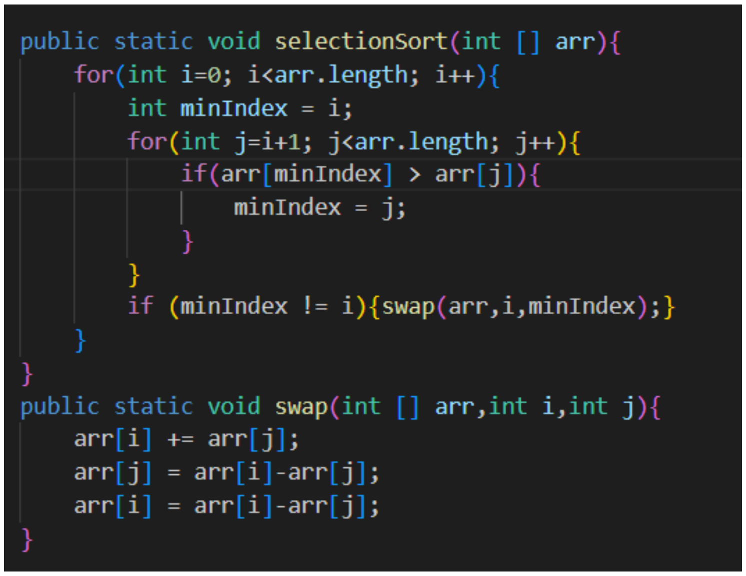
1. Go through the array to find the lowest value.
2. Move the lowest value to the front of the unsorted part of the array.
3. Go through the array again as many times as there are values in the array.

Selection Sort Implementation

To implement the Selection Sort algorithm in a programming language, we need:

1. An array with values to sort.
2. An inner loop that goes through the array, finds the lowest value and moves it to the front of the array. This loop must loop through one less value each time it runs.
3. An outer loop that controls how many times the inner loop must run. For an array with n values, this outer loop must run n−1 times.

The resulting code looks like this:



```java
public static void selectionSort(int [] arr){
    for(int i=0; i<arr.length; i++){
        int minIndex = i;
        for(int j=i+1; j<arr.length; j++){
            if(arr[minIndex] > arr[j]){
                minIndex = j;
            }
        }
        if (minIndex != i){swap(arr,i,minIndex);}
    }
}
public static void swap(int [] arr,int i,int j){
    arr[i] += arr[j];
    arr[j] = arr[i]-arr[j];
    arr[i] = arr[i]-arr[j];
}
```

Selection Sort Time Complexity

Selection Sort sorts an array of n values.

On average, about n/2 elements are compared to find the lowest value in each loop.

Selection Sort must run the loop to find the lowest value approximately n times.

We get time complexity:- O((n/2).n) = O(n^2)

Insertion sort:-

The Insertion Sort algorithm uses one part of the array to hold the sorted values, and the other part of the array to hold values that are not sorted yet.

The algorithm takes one value at a time from the unsorted part of the array and puts it into the right place in the sorted part of the array until the array is sorted.

How it works:

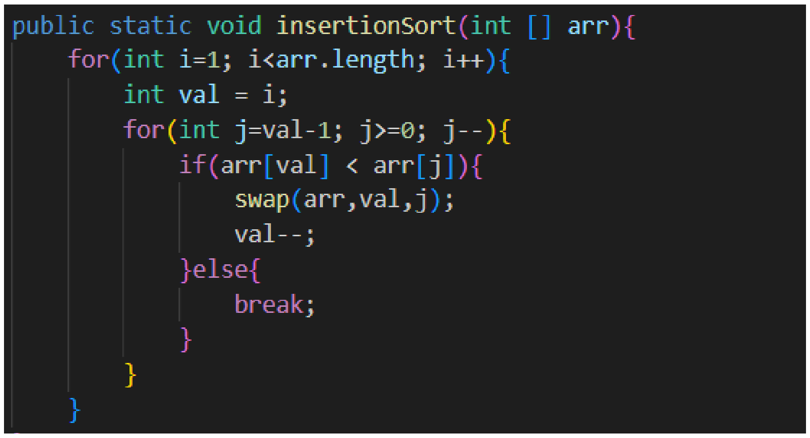
1. Take the first value from the unsorted part of the array.
2. Move the value into the correct place in the sorted part of the array.
3. Go through the unsorted part of the array again as many times as there are values.

Insertion Sort Implementation

To implement the Insertion Sort algorithm in a programming language, we need:

1. An array with values to sort.
2. An outer loop that picks a value to be sorted. For an array with n values, this outer loop skips the first value and must run n−1 times.
3. An inner loop that goes through the sorted part of the array, to find where to insert the value. If the value to be sorted is at index i, the sorted part of the array starts at index 0 and ends at index i−1.

The resulting code looks like this:



```
public static void insertionSort(int [] arr){
    for(int i=1; i<arr.length; i++){
        int val = i;
        for(int j=val-1; j>=0; j--){
            if(arr[val] < arr[j]){
                swap(arr,val,j);
                val--;
            }else{
                break;
            }
        }
    }
```

Insertion Sort Time Complexity

Selection Sort sorts an array of n values.

On average, each value must be compared to about n/2 other values to find out where to insert it.

Selection Sort must run the loop to insert a value in its correct place approximately n times.

We get time complexity for Insertion Sort: O((n/2).n) = O(n^2).

Counting sort:- The Counting Sort algorithm sorts an array by counting the number of times each value occurs.

Counting Sort does not compare values like the previous sorting algorithms we have looked at, and only works on non-negative integers.

How it works:

1. Create a new array for counting how many there are of the different values.
2. Go through the array that needs to be sorted.
3. For each value, count it by increasing the counting array at the corresponding index.
4. After counting the values, go through the counting array to create the sorted array.
5. For each count in the counting array, create the correct number of elements, with values that correspond to the counting array index.

Conditions for Counting Sort

These are the reasons why Counting Sort is said to only work for a limited range of non-negative integer values:

Integer values: Counting Sort relies on counting occurrences of distinct values, so they must be integers. With integers, each value fits with an index (for non-negative values), and there is a limited number of different values so that the number of possible different values k is not too big compared to the number of values n.

Non-negative values: Counting Sort is usually implemented by creating an array for counting. When the algorithm goes through the values to be sorted, value x is counted by increasing the counting array value at index x. If we tried sorting negative values, we would get in trouble with sorting value -3, because index -3 would be outside the counting array.

Limited range of values: If the number of possible different values to be sorted k is larger than the number of values to be sorted n, the counting array we need for sorting will be larger than the original array we have that needs sorting, and the algorithm becomes ineffective.

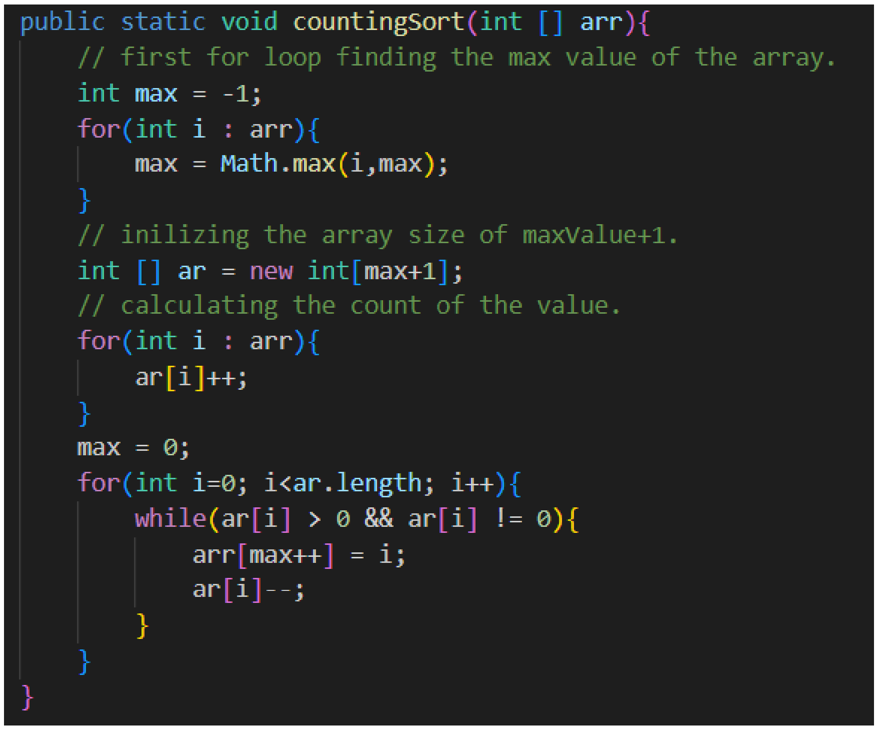
Counting Sort Implementation

To implement the Counting Sort algorithm in a programming language, we need:

1. An array with values to sort.
2. A 'countingSort' method that receives an array of integers.
3. An array inside the method to keep count of the values.
4. A loop inside the method that counts and removes values, by incrementing elements in the counting array.
5. A loop inside the method that recreates the array by using the counting array, so that the elements appear in the right order.

One more thing: We need to find out what the highest value in the array is so that the counting array can be created with the correct size. For example, if the highest value is 5, the counting array must be 6 elements in total, to be able to count all possible non-negative integers 0, 1, 2, 3, 4, and 5.

The resulting code looks like this:



```java
public static void countingSort(int [] arr){
    // first for loop finding the max value of the array.
    int max = -1;
    for(int i : arr){
        max = Math.max(i,max);
    }
    // inilizing the array size of maxValue+1.
    int [] ar = new int[max+1];
    // calculating the count of the value.
    for(int i : arr){
        ar[i]++;
    }
    max = 0;
    for(int i=0; i<ar.length; i++){
        while(ar[i] > 0 && ar[i] != 0){
            arr[max++] = i;
            ar[i]--;
        }
    }
}
```

Counting Sort Time Complexity

How fast the Counting Sort algorithm runs depends on both the range of possible values k and the number of values n.

In general, the time complexity for Counting Sort is O(n+k).

In a best-case scenario, the range of possible different values k is very small compared to the number of values n and Counting Sort has time complexity O(n).

But in a worst-case scenario, the range of possible different values k is very big compared to the number of values n and Counting Sort can have time complexity O(n2) or even worse.

Merge sort:- The Merge Sort algorithm is a divide-and-conquer algorithm that sorts an array by first breaking it down into smaller arrays, and then building the array back together the correct way so that it is sorted.

Divide: The algorithm starts with breaking up the array into smaller and smaller pieces until one such sub-array only consists of one element.

Conquer: The algorithm merges the small pieces of the array back together by putting the lowest values first, resulting in a sorted array.

The breaking down and building up of the array to sort the array is done recursively.

How it works:

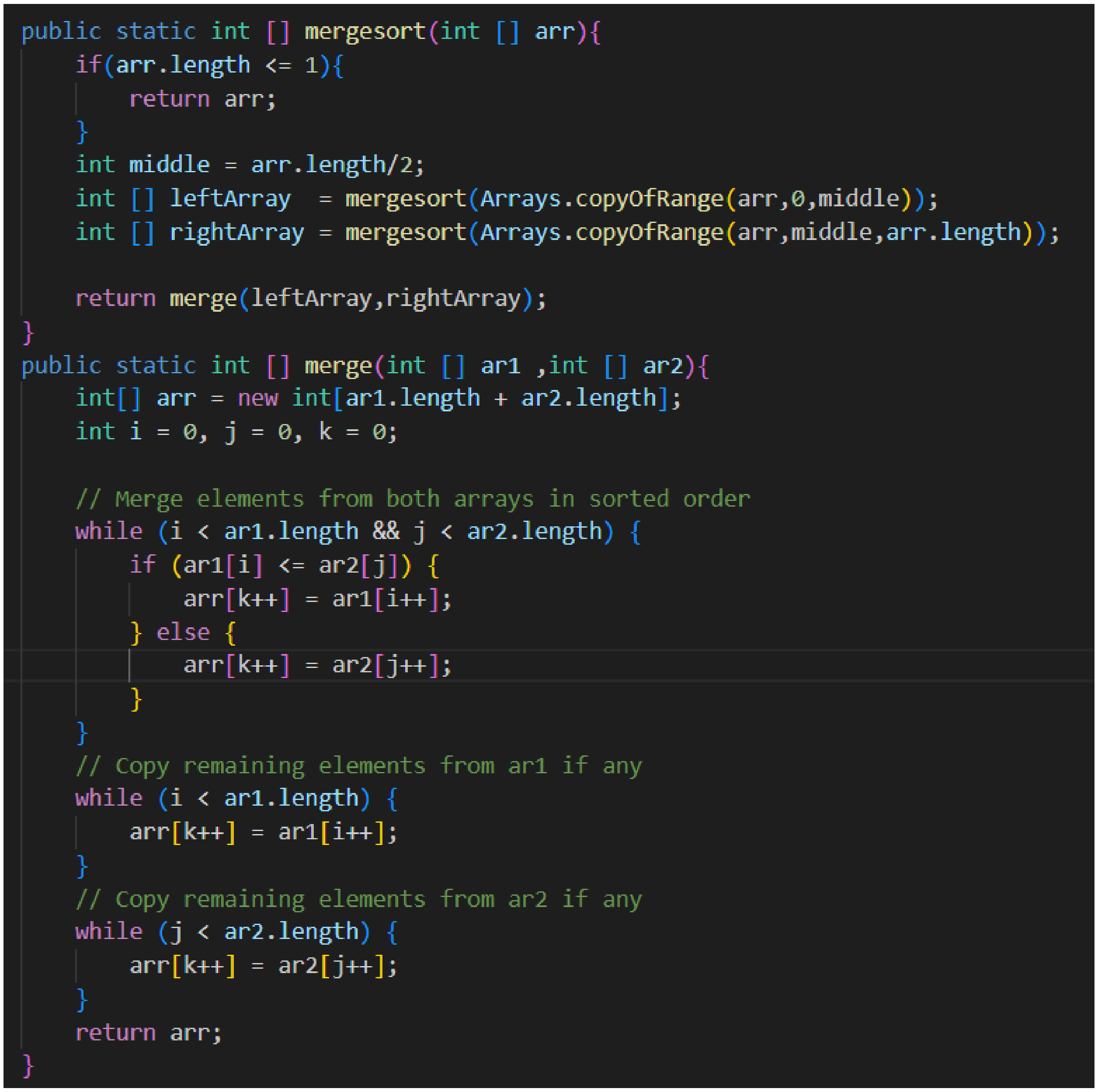
1. Divide the unsorted array into two sub-arrays, half the size of the original.
2. Continue to divide the sub-arrays as long as the current piece of the array has more than one element.
3. Merge two sub-arrays together by always putting the lowest value first.
4. Keep merging until there are no sub-arrays left.

Merge Sort Implementation

To implement the Merge Sort algorithm we need:

1. An array with values that needs to be sorted.
2. A function that takes an array, splits it in two, and calls itself with each half of that array so that the arrays are split again and again recursively until a sub-array only consists of one value.
3. Another function that merges the sub-arrays back together in a sorted way.

The resulting code looks like this:



```java
public static int [] mergesort(int [] arr){
    if(arr.length <= 1){
        return arr;
    }
    int middle = arr.length/2;
    int [] leftArray  = mergesort(Arrays.copyOfRange(arr,0,middle));
    int [] rightArray = mergesort(Arrays.copyOfRange(arr,middle,arr.length));

    return merge(leftArray,rightArray);
}
public static int [] merge(int [] ar1 ,int [] ar2){
    int[] arr = new int[ar1.length + ar2.length];
    int i = 0, j = 0, k = 0;

    // Merge elements from both arrays in sorted order
    while (i < ar1.length && j < ar2.length) {
        if (ar1[i] <= ar2[j]) {
            arr[k++] = ar1[i++];
        } else {
            arr[k++] = ar2[j++];
        }
    }
    // Copy remaining elements from ar1 if any
    while (i < ar1.length) {
        arr[k++] = ar1[i++];
    }
    // Copy remaining elements from ar2 if any
    while (j < ar2.length) {
        arr[k++] = ar2[j++];
    }
    return arr;
}
```

Merge Sort Time Complexity

The time complexity for Merge Sort is:- O(n·logn).

Searching Algorithm

Many search algorithms are available to search for an element in an array, like Linear Search, Binary Search, Hash Table (Dictionary), etc.

Here we are learning to basic search algorithm

1. Linear Search
2. Binary Search

Linear Search: The Linear Search algorithm searches through an array and returns the index of the value it searches for.

This algorithm is very simple and easy to understand and implement.

How it works:

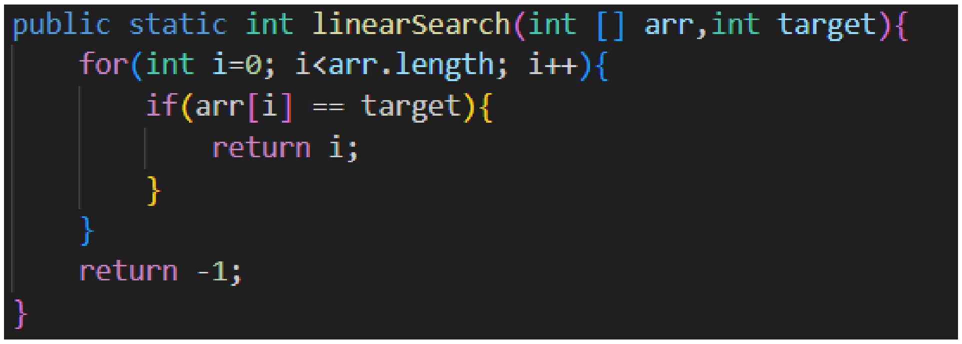
1. Go through the array value by value from the start.
2. Compare each value to check if it is equal to the value we are looking for.
3. If the value is found, return the index of that value.
4. If the end of the array is reached and the value is not found, return -1 to indicate that the value was not found.

Linear Search Implementation

To implement the Linear Search algorithm we need:

1. An array with values to search through.
2. A target value to search for.
3. A loop that goes through the array from start to end.
4. An if-statement that compares the current value with the target value, and returns the current index if the target value is found.
5. After the loop, return -1, because at this point we know the target value has not been found.

The resulting code for Linear Search looks like this:



```
public static int linearSearch(int [] arr,int target){
    for(int i=0; i<arr.length; i++){
        if(arr[i] == target){
            return i;
        }
    }
    return -1;
}
```

Linear Search Time Complexity

If Linear Search runs and finds the target value as the first array value in an array with n values, only one comparison is needed.

But if Linear Search runs through the whole array of n values, without finding the target value, n compares are needed.

This means that the time complexity for Linear Search is O(n).

Binary Search:- The Binary Search algorithm searches through an array and returns the index of the value it searches for.

Binary Search is much faster than Linear Search but requires a sorted array to work.

The Binary Search algorithm works by checking the value in the center of the array. If the target value is lower, the next value to check is in the center of the left half of the array. This way of searching means that the search area is always half of the previous search area, and this is why the Binary Search algorithm is so fast.

How it works:

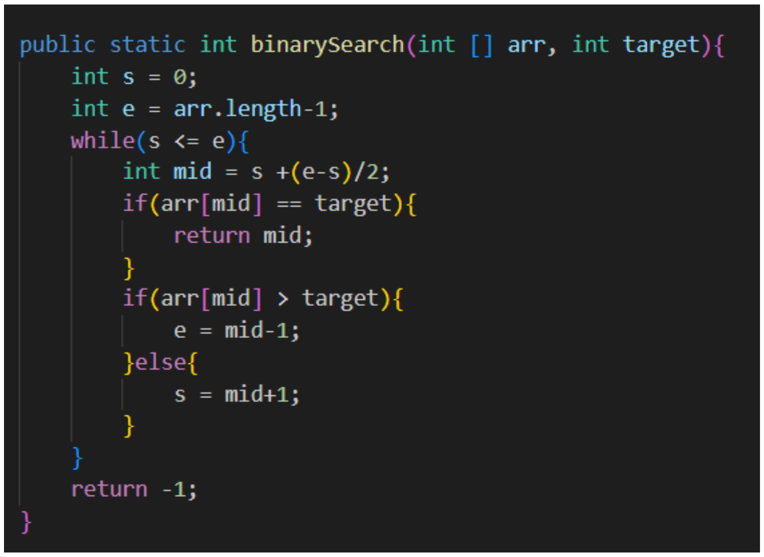
1. Check the value in the center of the array.
2. If the target value is lower, search the left half of the array. If the target value is higher, search the right half.
3. Continue steps 1 and 2 for the new reduced part of the array until the target value is found or until the search area is empty.
4. If the value is found, return the target value index. If the target value is not found, return -1.

Binary Search Implementation

To implement the Binary Search algorithm we need:

1. An array with values to search through.
2. A target value to search for.
3. A loop that runs as long as the left index is less than, or equal to, the right index.
4. An if-statement that compares the middle value with the target value, and returns the index if the target value is found.
5. An if-statement that checks if the target value is less than, or larger than, the middle value, and updates the "left" or "right" variables to narrow down the search area.
6. After the loop, return -1, because at this point we know the target value has not been found.

The resulting code for Binary Search looks like this:



```java
public static int binarySearch(int [] arr, int target){
    int s = 0;
    int e = arr.length-1;
    while(s <= e){
        int mid = s +(e-s)/2;
        if(arr[mid] == target){
            return mid;
        }
        if(arr[mid] > target){
            e = mid-1;
        }else{
            s = mid+1;
        }
    }
    return -1;
}
```

Binary Search Time Complexity

Each time Binary Search checks a new value to see if it is the target value, the search area is halved.

This means that even in the worst case scenario where Binary Search cannot find the target value, it still only needs $\log_2 n$ comparisons to look through a sorted array of n values.

Time complexity for Binary Search is $O(\log_2 n)$