Data Structures [R1UC308B]

Module-IV: Searching & Sorting

Dr. Subhash Chandra Gupta

GALGOTIAS UNIVERSITY

School of Computer Science and Engineering
Plat No 2, Sector 17A, Yamuna Expressway
Greater Noida, Uttar Pradesh - 203201

July 31, 2025

Contents

Searching 3
- Linear Search 4
- Binary Search 7
- Indexed Sequential Search 11
- Hashing 13

Sorting 19
- Insertion Sort 20
- Bubble Sort 22
- Selection Sort 24
- Quick Sort 26
- Merge Sort 29

Searching

Searching algorithms are essential tools in computer science used to locate specific items within a collection of data. These algorithms are designed to efficiently navigate through data structures to find the desired information, making them fundamental in various applications such as databases, web search engines, and more.
Different searching algorithms are:

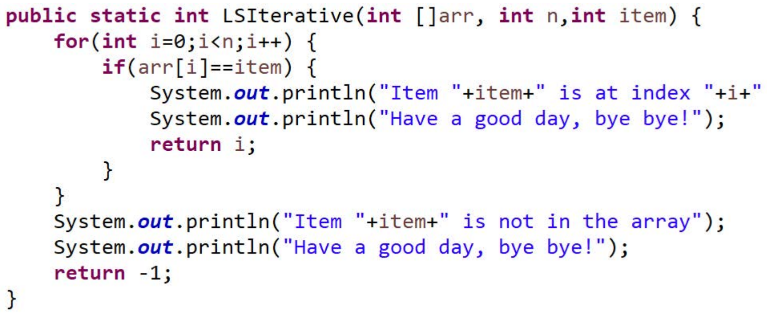
- Linear Search
- Binary Search
- Indexed Sequential Search
- Hashing

Linear Search

Linear search is a method for searching for an element in a collection of elements. Each element of the collection is visited one by one in a sequential fashion to find the desired element. Linear search is also known as sequential search.

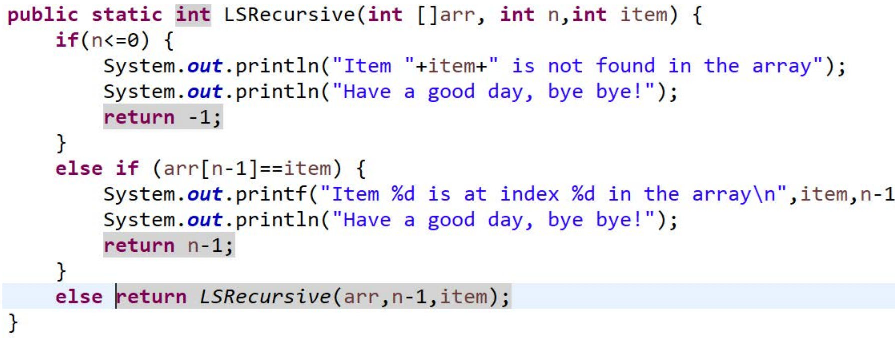
Linear Search Algorithm:

- Every element is considered as a potential match for the key and checked for the same.
- If any element is equal to the key, the search is successful and the index of that element is returned.
- If no element is found equal to the key, the search yields "No match found".



```
public static int LSIterative(int []arr, int n,int item) {
    for(int i=0;i<n;i++) {
        if(arr[i]==item) {
            System.out.println("Item "+item+" is at index "+i+"
            System.out.println("Have a good day, bye bye!");
            return i;
        }
    }
    System.out.println("Item "+item+" is not in the array");
    System.out.println("Have a good day, bye bye!");
    return -1;
}
```

Figure: Linear Search using Iteration



```
public static int LSRecursive(int []arr, int n,int item) {
    if(n<=0) {
        System.out.println("Item "+item+" is not found in the array");
        System.out.println("Have a good day, bye bye!");
        return -1;
    }
    else if (arr[n-1]==item) {
        System.out.printf("Item %d is at index %d in the array\n",item,n-1
        System.out.println("Have a good day, bye bye!");
        return n-1;
    }
    else return LSRecursive(arr,n-1,item);
}
```

Figure: Linear Search using Recursion

Binary Search

Binary search is a search algorithm used to find the position of a target value within a sorted array. It works by repeatedly dividing the search interval in half until the target value is found or the interval is empty. The search interval is halved by comparing the target element with the middle value of the search space.

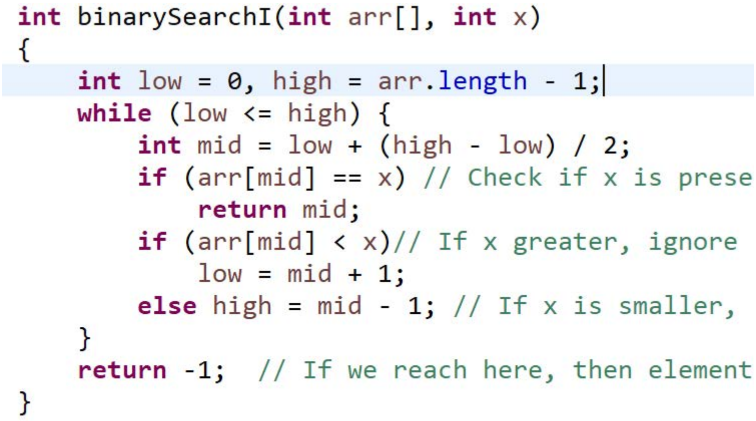
Conditions to apply Binary Search

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

Binary Search Algorithm:

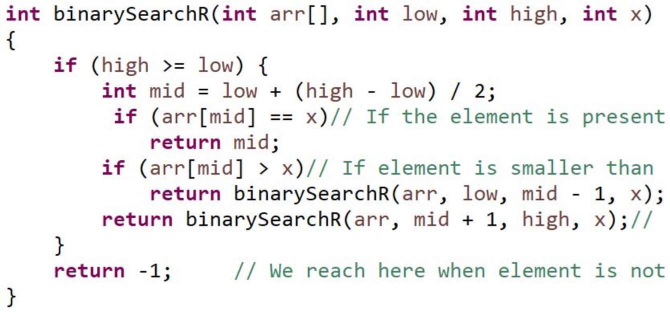
- Divide the search space into two halves by finding the middle index "mid".
- Compare the middle element of the search space with the key.

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



```
int binarySearchI(int arr[], int x)
{
    int low = 0, high = arr.length - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == x) // Check if x is prese
            return mid;
        if (arr[mid] < x)// If x greater, ignore
            low = mid + 1;
        else high = mid - 1; // If x is smaller,
    }
    return -1;  // If we reach here, then element
}
```

Figure: Binary Search using Iteration



```
int binarySearchR(int arr[], int low, int high, int x)
{
    if (high >= low) {
        int mid = low + (high - low) / 2;
         if (arr[mid] == x)// If the element is present
            return mid;
        if (arr[mid] > x)// If element is smaller than
            return binarySearchR(arr, low, mid - 1, x);
        return binarySearchR(arr, mid + 1, high, x);//
    }
    return -1;     // We reach here when element is not
}
```

Figure: Binary Search using Recursion

Indexed Sequential Search

Indexed Sequential search: In this searching method, first of all, an index file is created, that contains some specific group or division of required record when the index is obtained, then the partial indexing takes less time because it is located in a specified group. When the user makes a request for specific records it will find that index group first where that specific record is recorded.

Characteristics:

- In Indexed Sequential Search a sorted index is set aside in addition to the array.
- Each element in the index points to a block of elements in the array or another expanded index.
- The index is searched 1st then the array and guides the search in the array.

Indexed Sequential Search actually does the indexing multiple time, like creating the index of an index.

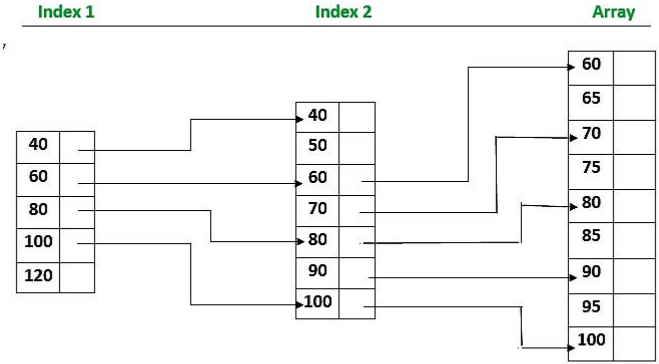


Figure: Indexed Sequential Search

Hashing

Hashing is a technique used in data structures that efficiently stores and retrieves data in a way that allows for quick access.
- Hashing refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions.
- This technique determines an index or location for the storage of an item in a data structure. - It involves mapping data to a specific index in a hash table using a hash function that enables fast retrieval of information based on its key.
- This method is commonly used in databases, caching systems, and various programming applications to optimize search and retrieval operations.
- The great thing about hashing is, we can achieve all three operations (search, insert and delete) in O(1) time on average.

Components of Hashing: There are majorly three components of hashing:

1. Key: A Key can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. Hash Function: The hash function receives the input key and returns the index of an element in an array called a hash table. The index is known as the hash index.
3. Hash Table: Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.

Collision: in Hashing occurs when two different keys map to the same hash value.
- The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value.
- The situation where the newly inserted key maps to an already occupied key value then it must be handled using some collision handling technology.

Causes of Hash Collisions:

- Poor Hash Function: A hash function that does not distribute keys evenly across the hash table can lead to more collisions.
- High Load Factor: A high load factor (ratio of keys to hash table size) increases the probability of collisions.
- Similar Keys: Keys that are similar in value or structure are more likely to collide.

- There are mainly two methods to handle collision:

- Open Addressing:
  Linear Probing: Search for an empty slot sequentially
  Quadratic Probing: Search for an empty slot using a quadratic function
- Closed Addressing:
  Chaining: Store colliding keys in a linked list or binary search tree at each index
  Cuckoo Hashing: Use multiple hash functions to distribute keys Separate Chaining

Applications of Hashing: Hash tables are used wherever we have a combinations of search, insert and/or delete operations.

- Dictionaries: To implement a dictionary so that we can quickly search a word.

- Databases: Hashing is used in database indexing. There are two popular ways to implement indexing, search trees (B or B+ Tree) and hashing.
- Cryptography: When we create a password on a website, they typically store it after applying a hash function rather than plain text.
- Caching: Storing frequently accessed data for faster retrieval. For example browser caches, we can use URL as keys and find the local storage of the URL.
- Symbol Tables: Mapping identifiers to their values in programming languages
- Network Routing: Determining the best path for data packets

- Associative Arrays: Associative arrays are nothing but hash tables only. Commonly SQL library functions allow you retrieve data as associative arrays so that the retrieved data in RAM can be quickly searched for a key.

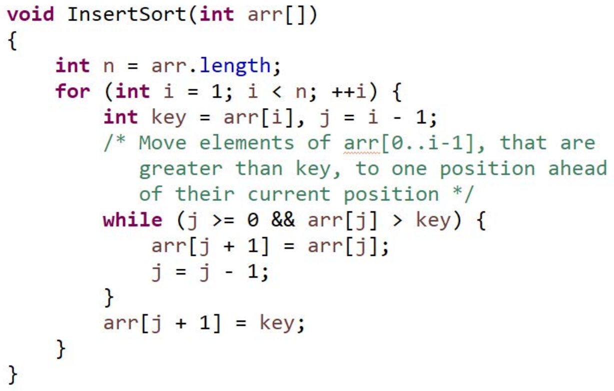
Sorting

A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure. For example arranging students acoording to hight in morning assembly, seating roll no wise in exams, arranging names marks wise in merit list etc. There are different algorithms for sorting:

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

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.
- Insertion sort is like sorting playing cards in your hands.
- You split the cards into two groups: the sorted cards and the unsorted cards.
- Then, you pick a card from the unsorted group and put it in the right place in the sorted group.



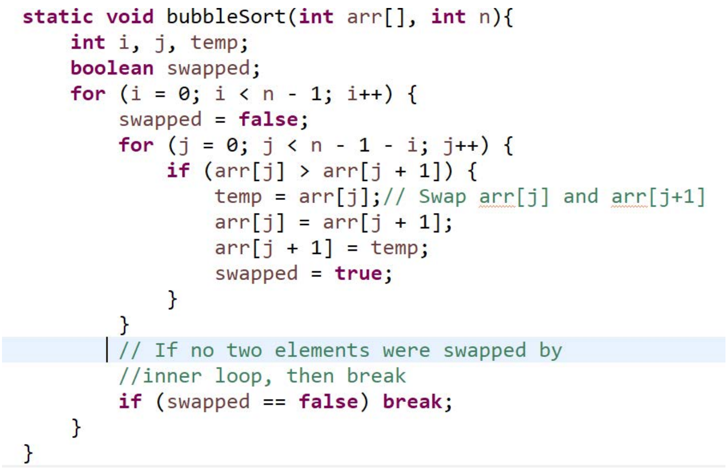
```
void InsertSort(int arr[])
{
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i], j = i - 1;
        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Figure: Insertion Sort

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.
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.

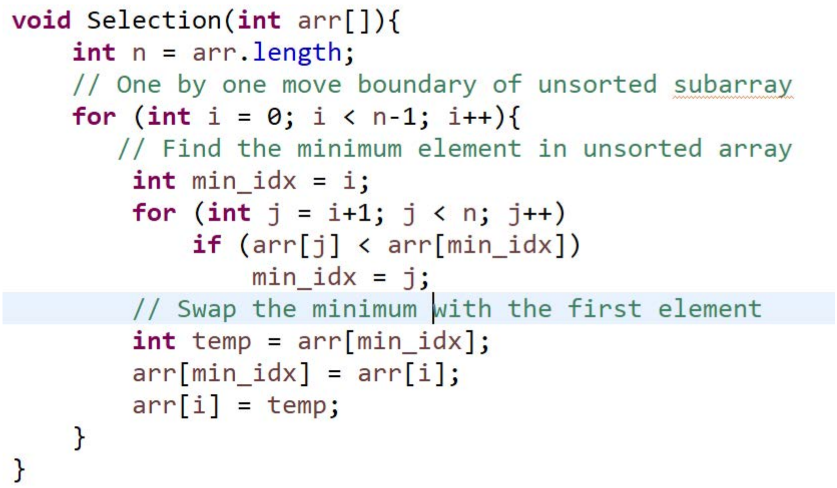


```
static void bubbleSort(int arr[], int n){
    int i, j, temp;
    boolean swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];// Swap arr[j] and arr[j+1]
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        // If no two elements were swapped by
        //inner loop, then break
        if (swapped == false) break;
    }
}
```

Figure: Bubble Sort

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.
- The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part.
- This process is repeated for the remaining unsorted portion until the entire list is sorted.

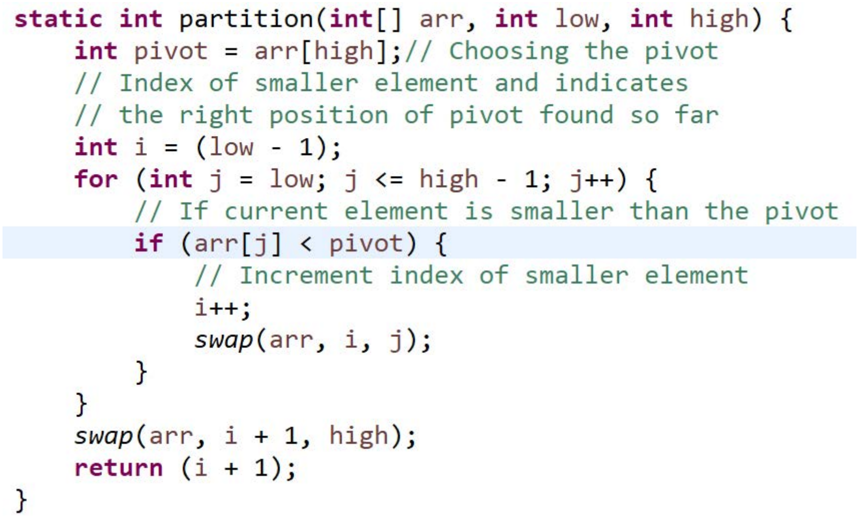


```
void Selection(int arr[]){
    int n = arr.length;
    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n-1; i++){
       // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        // Swap the minimum with the first element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

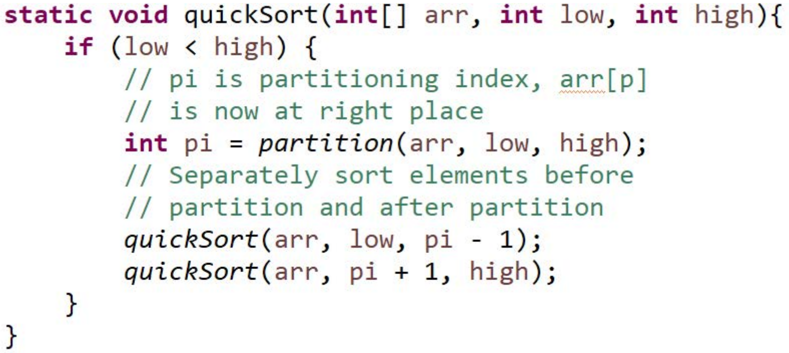
Figure: Selection Sort

Quick Sort

- QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.
- There are mainly three steps in the algorithm.
- 1. Choose a pivot
- 2. Partition the array around pivot. After partition, it is ensured that all elements are smaller than all right and we get index of the end point of smaller elements. The left and right may not be sorted individually.
- 3. Recursively call for the two partitioned left and right subarrays. We stop recursion when there is only one element is left.



```
static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];// Choosing the pivot
    // Index of smaller element and indicates
    // the right position of pivot found so far
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        if (arr[j] < pivot) {
            // Increment index of smaller element
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, high);
    return (i + 1);
}
```

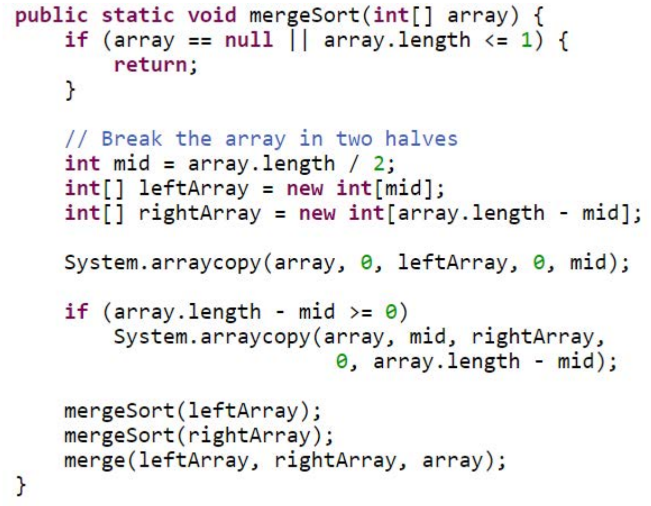


```
static void quickSort(int[] arr, int low, int high){
    if (low < high) {
        // pi is partitioning index, arr[p]
        // is now at right place
        int pi = partition(arr, low, high);
        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Figure: Quick Sort

Merge Sort

- Merge sort is a sorting algorithm that follows the divide-and-conquer approach.
- It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.
- In simple terms, the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together.
- This process is repeated until the entire array is sorted.



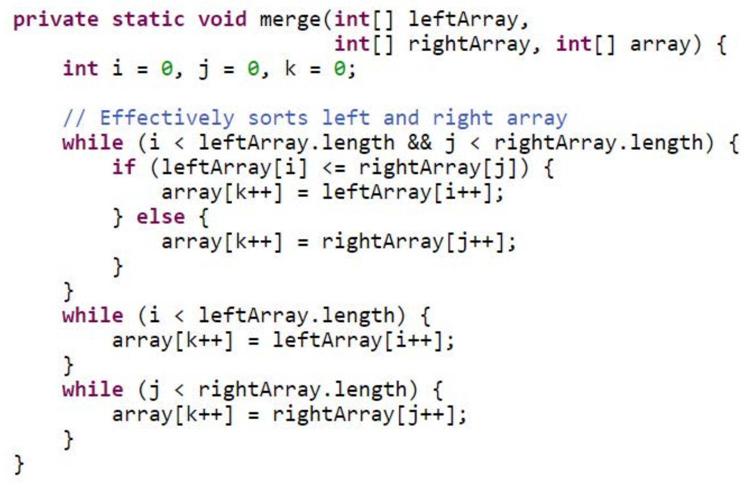
```
public static void mergeSort(int[] array) {
    if (array == null || array.length <= 1) {
        return;
    }

    // Break the array in two halves
    int mid = array.length / 2;
    int[] leftArray = new int[mid];
    int[] rightArray = new int[array.length - mid];

    System.arraycopy(array, 0, leftArray, 0, mid);

    if (array.length - mid >= 0)
        System.arraycopy(array, mid, rightArray,
                         0, array.length - mid);

    mergeSort(leftArray);
    mergeSort(rightArray);
    merge(leftArray, rightArray, array);
}
```



```
private static void merge(int[] leftArray,
                          int[] rightArray, int[] array) {
    int i = 0, j = 0, k = 0;

    // Effectively sorts left and right array
    while (i < leftArray.length && j < rightArray.length) {
        if (leftArray[i] <= rightArray[j]) {
            array[k++] = leftArray[i++];
        } else {
            array[k++] = rightArray[j++];
        }
    }
    while (i < leftArray.length) {
        array[k++] = leftArray[i++];
    }
    while (j < rightArray.length) {
        array[k++] = rightArray[j++];
    }
}
```

Figure: Merge Sort

Thank you

Please send your feedback or any queries to
subhash.chandra@galgotiasuniversity.edu.in