Unit 1

**Syllabus:**

UNIT -1 (15 hrs) **Algorithm Analysis**: Introduction to algorithms, analysing and designing algorithms, Growth functions, asymptotic notations, Recursive algorithm complexity, **solving recurrences**: Substitution method, recursion tree method, master method. **Searching**: Binary search, **Hashing**: Hashing, Hash tables, Hash functions, collision resolution techniques. **Sorting**: Quick sort, Counting sort, Radix sort, Merge sort, Heap sort, Insertion sort and selection sort.

## Introduction to algorithm:

1. **What is means algorithm:**
2. An **algorithm** is step by step description of any program in general language
3. An **algorithm** is a sequence of a clear instructions used to solving a problem such a way that it can implemented as a program for computer

1. An **algorithm** is like a recipe for solving a specific problem. It’s a sequence of well-defined steps or instructions that guide a computer (or a human) through a process to achieve a desired outcome.
2. **Characteristics of algorithm:**

**Characteristics of a Good Algorithm:**

1. **Correctness**:
   * An algorithm must produce the correct output for all valid inputs.
   * It should solve the problem accurately without errors.
   * Verifying correctness often involves mathematical proofs or extensive testing.
2. **Efficiency**:
   * A good algorithm should be efficient in terms of time and memory usage.
   * Efficiency matters because we want solutions that run quickly and don’t waste resources.
   * Common measures of efficiency include Big O notation and space complexity.
3. **Clarity and Simplicity**:
   * An algorithm should be easy to understand.
   * Clear, straightforward steps make it easier to implement and maintain.
   * Avoid unnecessary complexity.
4. **Language Independence**:

* An algorithm is expressed in a high-level, abstract manner that doesn’t depend on any specific programming language.
* Whether you write an algorithm in Python, Java, C++, or any other language, the underlying logic remains the same.

1. **Generality**:
   * A general-purpose algorithm can be applied to a wide range of inputs.
   * Specific algorithms (e.g., sorting, searching) should work for various data types and sizes.
2. **Deterministic**:
   * Algorithms are deterministic—they produce the same output for the same input every time.
   * There’s no randomness involved.
3. **Finiteness**:
   * An algorithm must terminate after a finite number of steps.
   * Infinite loops are not acceptable.

Remember, algorithms are everywhere! When you use your phone, search the web, or even cook a new recipe, you’re interacting with algorithms. They’re the backbone of modern computing and problem-solving.

1. Advantages and Disadvantages of algorithm:

5M

**Advantages of Algorithms:**

1. **Effective communication:**

Since it is written in natural language like English it become easy to understand the step-by-step definition of a solution to any particular problem.

1. **Easy debugging:**

A well-designed algorithm facilities easy debugging to detect the logical errors that occurred inside the program.

1. **Easy and efficient coding:**

An algorithm is nothing but a blue print of the program that helps to develop a program.

1. **Independent of programming language:**

Since it is language independent, it can be easily coded by incorporating any high level language.

**Disadvantages of Algorithms:**

* 1. **Inefficiency**

Some algorithms are inefficient in terms of time and space complexity. Leading to lower processing speed and high resources usage.

* 1. **Not universal applicable**

Not all algorithms are suitable for every problem. Using the wrong algorithm can result in suboptimal solutions.

* 1. **Optimization challenges**

Optimizing algorithm for specific use cases or dataset can be complex and time-consuming process requiring in depth knowledge and expertise

* 1. **Scalability issues:**

Some algorithm may not scale well when applied to large dataset causing performance issue as the size of database increases.

**2M**

**Advantages of Algorithms:**

* It is easy to understand.
* An algorithm is a step-wise representation of a solution to a given problem.
* In an Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

**Disadvantages of Algorithms:**

* Writing an algorithm takes a long time so it is time-consuming.
* Understanding complex logic through algorithms can be very difficult.
* Branching and Looping statements are difficult to show in Algorithms.

## Algorithm analysis

## **Algorithm analysis** is a crucial aspect of computer science that **involves evaluating and understanding the efficiency of algorithms.**

## **Definition:**

## Algorithm analysis refers to the study of algorithms behaviour and performance.

## It aims to determine how much time and memory resources an algorithm requires to solve a specific computational problem.

* Need of algorithm analysis:
  1. **Efficiency:**

Algorithm analysis helps us to understand how fast and how much space an algorithm will use, allowing us to select the most efficient one for a specific task.

* 1. **Performance comparison:**

By analysing algorithm, we can compare them to see which one works best for the particular problem. Helping us make smarter choice.

* 1. **Resource management:**

Algorithm analysis assist in managing resources effectively, ensuring we use computational resources like memory and processing power optimally.

* 1. **Scalability:**

It helps to predict how algorithm will perform as the input size increases. Aiding in designing solutions that can handles varying data sizes efficiently.

* 1. **Optimization:**

Algorithm analysis helps identify areas for improvements. Leading to better performance and efficiency of algorithm and system.

## Phase of Algorithm

## The algorithm can be examined in 2 phases. The two analyses of an algorithm are as follows:

## **Priori Analysis:**

## Priori analysis is the theoretical analysis of an algorithm performed prior to its implementation.

## Before running or executing the algorithm, other parameters might be considered, such as the speed of the processor, which has no impact on the execution phase.

* + Time complexity and space complexity are determined theoretically.
  + Often uses asymptotic notations (like Big O) to represent time complexity.

## **Posterior Analysis**:

## It is also known as the practical analysis of an algorithm.

## The algorithm is implemented in any computer language to obtain practical analysis.

## This analysis is used to determine how much running time and space the technique consumes.

* + Time and space complexity are observed during actual runs.
  + Results may vary from system to system.

| **Aspect** | **A Priori Analysis** | **A Posteriori Analysis** |
| --- | --- | --- |
| **Definition** | Theoretical analysis based on models | Empirical analysis based on actual runs |
| **Focus** | Idealized behaviour | Real-world performance |
| **Resource Independence** | Independent of specific hardware, compiler, or system conditions | Dependent on specific system conditions |
| **Accuracy** | Approximate results | Exact results |
| **Notations Used** | Asymptotic notations (e.g., Big O) | Actual execution time |
| **Credit for Speed** | Goes to the programmer | May go to compiler or hardware |
| **Timing** | Done before execution | Done after execution |
| **Cost** | Cheaper | Costlier (requires software and hardware) |
| **Maintenance Phase** | Not required | Required for tuning |

## Quick sort

* Quicksort is one of the fastest sorting algorithms.
* The Quicksort algorithm takes an array of values, chooses one of the values as the 'pivot' element, and moves the other values so that lower values are on the left of the pivot element, and higher values are on the right of it.
* **Recursion is when a function calls itself.**
* After the Quicksort algorithm has put the pivot element in between a sub-array with lower values on the left side, and a sub-array with higher values on the right side, the algorithm calls itself twice, so that Quicksort runs again for the sub-array on the left side, and for the sub-array on the right side. The Quicksort algorithm continues to call itself until the sub-arrays are too small to be sorted.

**Algorithm :**

* **Step 1** − Choose the last index value has pivot
* **Step 2** − Take two variables to point left and right of the list excluding pivot
* **Step 3** − left points to the low index
* **Step 4** − right points to the high
* **Step 5** − while value at left is less than pivot move *right*
* **Step 6** − while value at right is greater than pivot move *left*
* **Step 7** − if both step 5 and step 6 does not match swap left and right
* **Step 8** − if left ≥ right, the point where they met is new pivot

**Example**:

***Step 1:****We start with an unsorted array.*

*[ 11, 9, 12, 7, 3]*

***Step 2:****We choose the last value 3 as the pivot element.*

*[ 11, 9, 12, 7, 3]*

***Step 3:****The rest of the values in the array are all greater than 3, and must be on the right side of 3. Swap 3 with 11.*

*[ 3, 9, 12, 7, 11]*

***Step 4:****Value 3 is now in the correct position. We need to sort the values to the right of 3. We choose the last value 11 as the new pivot element.*

*[ 3, 9, 12, 7, 11]*

***Step 5:****The value 7 must be to the left of pivot value 11, and 12 must be to the right of it. Move 7 and 12.*

*[ 3, 9, 7, 12, 11]*

***Step 6:****Swap 11 with 12 so that lower values 9 and 7 are on the left side of 11, and 12 is on the right side.*

*[ 3, 9, 7, 11, 12]*

***Step 7:****11 and 12 are in the correct positions. We choose 7 as the pivot element in sub-array [ 9, 7], to the left of 11.*

*[ 3, 9, 7, 11, 12]*

***Step 8:****We must swap 9 with 7.*

*[ 3, 7, 9, 11, 12]*

*And now, the array is sorted. [ 3, 7, 9, 11, 12]*

***Code:***

public class CWS\_04\_QuickSort {

    // function for partition

    public static int partition(int[] arr, int low, int high) {

        int pivot = arr[high];

        int i = low - 1;

        // dividing array with resepect to pivot

        for (int j = low; j < high; j++) {

            if (arr[j] < pivot) {

                // make emt space

                i++;

                // swap

                int temp = arr[i];

                arr[i] = arr[j];

                arr[j] = temp;

            }

        }

        // now add pivot to its correct position

        // make emt space

        i++;

        int temp = arr[i];

        arr[i] = arr[high];

        arr[high] = temp;

        return i;// pivot index

    }

    // recursive function for sort

    public static void quickSort(int[] arr, int low, int high) {

        if (low < high) {

            int pivotIndex = partition(arr, low, high);

            // recursive sort for smaller elemenets from pivot

            quickSort(arr, low, pivotIndex - 1);

            // recursive sort for higher elemenets from pivot

            quickSort(arr, pivotIndex + 1, high);

        }

    }

    //main

    public static void main(String[] args) {

        int[] arr = { 5, 3, 4, 7, 15, 8 };

        // length

        int n = arr.length;

        // fn call

        quickSort(arr, 0, n - 1);

        // print

        for (int i : arr) {

            System.out.println(i);

        }

        //output

        //3 4 5 7 8 15

    }

}

## Merge sort

## Selection sort

* + Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest 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.
  + **Algorithm**:

The steps of selection sort are as follows:

* Step 1- Set MIN to location 0.
* Step 2- Find the smallest element in the list.
* Step 3- Swap the smallest element with the value at location MIN.
* Step 4- Increment MIN to point to the next element.
* Step 5- Repeat until the list is sorted.
  + **Example**:

Consider you have an unsorted array: **[29, 10, 37, 14, 13],**

| Pass | Operation | Resulting Array |
| --- | --- | --- |
| 1 | Find minimum (10) and swap with the first element (29) | **[10, 29, 37, 14, 13]** |
| 2 | Find minimum (13) in the unsorted section and swap it with the first unsorted element (29) | **[10, 13, 37, 14, 29]** |
| 3 | Find minimum (14) in the unsorted section and swap it with the first unsorted element (37) | **[10, 13, 14, 37, 29]** |
| 4 | Find minimum (29) in the unsorted section and swap it with the first unsorted element (37) | **[10, 13, 14, 29, 37]** |

* + **Code**:

class SelectionSort {

    public static void main(String args[]) {

        int[] arr = { 10, 20, 30, 0, -45, -120 };

        // length of array

        int n = arr.length;

        // for loop till length-1

        for (int i = 0; i < n - 1; i++) {

            int smallest = i;

            // for loop till length with starting index from i+1

            for (int j = i + 1; j < n; j++) {

                if (arr[j] < arr[smallest]) {

                    smallest = j;

                }

            }

            // always swaping number if smaller or not

            int temp = arr[i];

            arr[i] = arr[smallest];

            arr[smallest] = temp;

        }

        for (int i : arr) {

            System.out.println(i);

        }

        //output -120 -45 0 10 20

    }

}

## 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.
  + 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.
  + Insertion Sort Algorithm:
  + Insertion sort is a simple sorting algorithm that works by building a sorted array one element at a time. It is considered an **in-place** sorting algorithm, meaning it doesn’t require any additional memory space beyond the original array.
  + **Algorithm:**

## **Step 1 - If the element is the first element, assume that it is already sorted. Return 1.**

## **Step2 - Pick the next element, and store it separately in a key.**

## **Step3 - Now, compare the key with all elements in the sorted array.**

## **Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.**

## **Step 5 - Insert the value.**

## **Step 6 - Repeat until the array is sorted.**

* + **Example:**

Consider an array having elements: {23, 1, 10, 5, 2}

1. *First Pass:*

* *Current element is****23***
* *The first element in the array is assumed to be sorted.*
* *The sorted part until****0th****index is:****[23]***

**

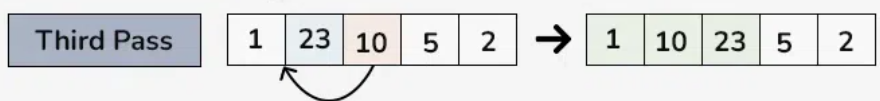
1. *Second Pass:*

* *Compare****1****with****23****(current element with the sorted part).*
* *Since****1****is smaller, insert****1****before****23.***
* *The sorted part until****1st****index is:****[1, 23]***

**

1. *Third Pass:*

* *Compare****10****with****1****and****23****(current element with the sorted part).*
* *Since****10****is greater than****1****and smaller than****23,*** *insert****10****between****1****and****23.***
* *The sorted part until****2nd****index is:****[1, 10, 23]***

**

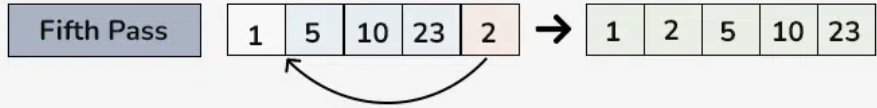
1. *Fourth Pass:*

* *Compare****5****with****1****,****10****, and****23****(current element with the sorted part).*
* *Since****5****is greater than****1****and smaller than****10****, insert****5****between****1****and****10****.*
* *The sorted part until****3rd****index is****: [1, 5, 10, 23]***

**

1. *Fifth Pass:*

* *Compare****2****with****1, 5, 10****, and****23****(current element with the sorted part).*
* *Since****2****is greater than****1****and smaller than****5****insert****2****between****1****and****5****.*
* *The sorted part until****4th****index is:****[1, 2, 5, 10, 23]***

**

1. *Final Array:*

* *The sorted array is:****[1, 2, 5, 10, 23]***
  + **Code**:

public class CWS\_03\_InsertionSort {

    public static void main(String[] args) {

        int[] arr = { 10, 70, 14, -5, 7, 6 };

        // length of array

        int n = arr.length;

        // for loop from index 1(element 2) to array length

        for (int i = 1; i < n; i++) {

            // element at i

            int current = arr[i];

            // previous sorted elements position

            int j = i - 1;

            // if element at i less than element at i-1 also checking array index never be

            // negative

            while (j >= 0 && current < arr[j]) {

                // if that swap the element with element at i with element at i-1

                arr[j + 1] = arr[j];

                j--;

            }

            // if not element at i as element at i

            arr[j + 1] = current;

        }

        for (int i : arr) {

            System.out.println(i);

        }

        // output

        // -5 6 7 10 14 70

    }

}

## Binary search:

Example:

void main() {

  List<int> a = [10, 20, 30, 40, 50];

  int l = 0; //lower index

  int u = 4; //upper index

  int mid = 0; //for middle value

  int f = 0; //for found or not

  int s = 50; //search this

  // loop till lower index<=upper index

  while (l <= u) {

    mid = ((l + u) / 2).floor();

    if (a[mid] == s) {

      f = 1;

      break;

    }

    if (a[mid] < s) {

      l = mid + 1;

    } else {

      u = mid - 1;

    }

  }

  if (f == 1) {

    print("$s is found at position ${mid + 1}");

  } else {

    print("$s in not found in $a");

  }

}