MODULE 1:

DAA introduction

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way.

Characteristics of Algorithms

The main characteristics of algorithms are as follows –

- Algorithms must have a unique name
- Algorithms should have explicitly defined set of inputs and outputs
- Algorithms are well-ordered with unambiguous operations
- Algorithms halt in a finite amount of time. Algorithms should not run for infinity, i.e., an algorithm must end at some point

Pseudocode

Pseudocode gives a high-level description of an algorithm without the ambiguity associated with plain text but also without the need to know the syntax of a particular programming language.

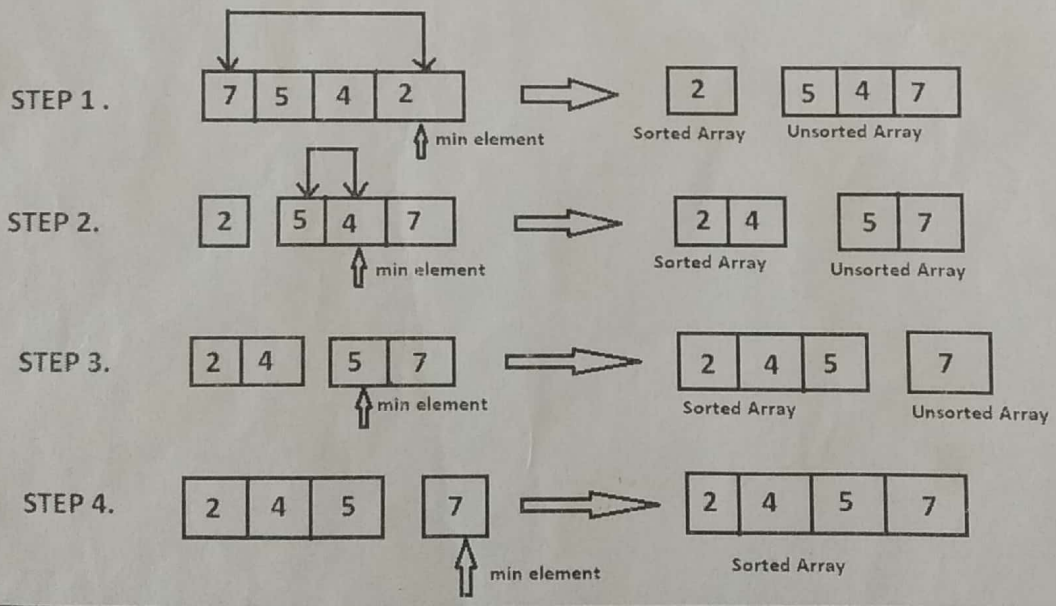
Difference between Algorithm and Pseudocode

- An algorithm is a formal definition with some specific characteristics that describes a process, which could be executed by a Turing-complete computer machine to perform a specific task.
- pseudocode is an informal and (often rudimentary) human readable description of an algorithm leaving many granular details of it.
- Writing a pseudocode has no restriction of styles and its only objective is to describe the high level steps of algorithm in a much realistic manner in natural language

Algorithm of selection sort

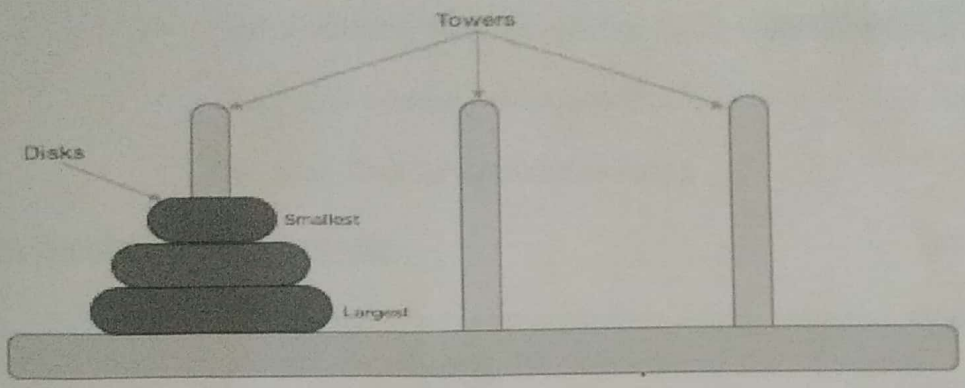
- The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.
- Assume that the array A=[7,5,4,2] needs to be sorted in ascending order.
- The minimum element in the array i.e. 2 is searched for and then swapped with the element that is currently located at the first position, i.e. 7. Now the minimum element in the remaining unsorted array is searched for and put in the second position, and so on.

- 1.Void SelectionSort(Type a[],int n)
- 2.// sort the array a[1:n} into nondecreasing order
- 3{
- 4. For (int i=1;i<=n;i++){
- 5. Int j=i; // assuming the first element to be the minimum of the unsorted array .
- 6. for(int k=i+1;k<=n;k++) //// gives the effective size of the unsorted array
- 7. if(a[k]<a[j]) j=k; //finds the minimum element
- 8. Type t=a[i];a[i]=a[j];a[j]=t; // putting minimum element on its proper position.
- 9.}
- 10.}



Recurive Algoritham

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted



- These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one.
- **Rules**
- The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are –
- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Recursive Algorithms: X-source

Y-destination

Z-auxiliary

1.TowersOfHanoi

- 1.Algorithm TowerofHanoi(n,x,y,z)
- 2.//Move the top n disks from tower x to tower y
- 3.{
- 4.if(n>=1) then
- 5.{ TowerOfHanoi(n-1,x,z,y);//Move top (n-1)disk from x to z
- 6.Write("move top disk from tower",x,"to top of tower",y);
- 7.TowerOfHanoi(n-1,z,y,x)//Move n-1 disk from Z to y
- 8.}
- 9.}

Algorithm Design Techniques

1.Greedy algorithms

2.Divide and conquer

3 .Dynamic Programming

4 .Randomized Algorithms

5. Backtracking Algorithms

Optimization Problem

1. In an optimization problem we are given a set of constraints and an optimization function.
2. Solutions that satisfy the constraints are called feasible solutions.
3. A feasible solution for which the optimization function has the best possible value is called an optimal solution.

1. Greedy Algorithms

- Greedy algorithms seek to optimize a function by making choices (greedy criterion) which are the best locally but do not look at the global problem.
- The result is a good solution but not necessarily the best one.
- The greedy algorithm does not always guarantee the optimal solution however it generally produces solutions that are very close in value to the optimal.

2.Divide and Conquer

- To solve a large instance :
- 1. Divide it into two or more smaller instances.
- 2. Solve each of these smaller problems, and
- 3. combine the solutions of these smaller problems to obtain the solution to the original instance.
- The smaller instances are often instances of the original problem and may be solved using the divide-and-conquer strategy recursively

3.Dynamic programming

- Used for **optimization problems**
    - A set of choices must be made to get an optimal solution

- Find a solution with the optimal value (minimum or maximum)
- There may be many solutions that lead to an optimal value
- Our goal: **find an optimal solution**

4. Randomized Algorithms

- A randomized algorithm is an algorithm where a random number is used to make a decision at least once during the execution of the algorithm.
- The running time of the algorithm depends not only on the particular input, but also on the random numbers that occur

5. Backtracking

- Backtracking is a systematic way to search for the solution to a problem.
- This method attempts to extend a partial solution toward a complete solution.
- In backtracking, we begin by defining a solution space or search space for the problem

Algorithm Analysis

Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as **time complexity**, or volume of memory, known as **space complexity**.

The main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis −

- **Worst-case** − The maximum number of steps taken on any instance of size **a**.
- **Best-case** − The minimum number of steps taken on any instance of size **a**.
- **Average case** − An average number of steps taken on any instance of size **a**.
- **Amortized** − A sequence of operations applied to the input of size **a** averaged over time.

Asymptotic notations

In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.

- Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.
- The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

Big-O(Big Oh)

- Big-O, commonly written as **O**

- It provides us with an ***asymptotic upper bound*** for the growth rate of runtime of an algorithm.

- Say f(n) - algorithm runtime,

  and g(n) - arbitrary time complexity.

  f(n) is O(g(n)), if for some real constants c (c > 0) and n_0,

  f(n) <= c * g(n) for every input size n (n > n_0).

Example 1

f(n) = 3log n + 100

g(n) = log n

Is f(n) O(g(n))? Is 3 log n + 100

O(log n)? Let's look to the definition of Big-O.

3log n + 100 <= c * log n

Is there some pair of constants c, n_0 that satisfies this for all n > $_0$?

3log n + 100 <= 150 * log n, n > 2 (undefined at n = 1)

Big-Omega

- Big-Omega, commonly written as Ω
- It provides us with an ***asymptotic lower bound*** for the growth rate of runtime of an algorithm.
- f(n) is Ω(g(n)), if for some real constants c (c > 0) and n_0 (n_0 > 0),

f(n) >= c * g(n) for every input size n (n > n_0).

Eg: The function 3n+2=Ω(n) as 3n+2>=3n for n>=1

Theta ((Θ))

The function f(n)= Θ(g(n)) iff there exist positive constants c1,c2 and n0 such that

c1*g(n) <=f(n) <=c2*g(n) for all n,n>=no.

Little Oh

The function f(n)=o (g(n)) iff

Lim n-> ∞ f(n)/g(n) =0

Little omega: The function f(n)=ω(g(n)) iff

Lim n-> ∞ g(n)/f(n) =0

Performance Analysis

Performance analysis of an algorithm depends upon two factors i.e. amount of memory used and amount of compute time consumed on any CPU. Formally they are notified as complexities in terms of:

- **Space Complexity.**

- **Time Complexity.**

Space Complexity of an algorithm is the amount of memory it needs to run to completion from start of execution to its termination.

Space need by any algorithm is the sum of following components:

1. **Fixed Component**: This is independent of the characteristics of the inputs and outputs. This part includes: Instruction Space, Space of simple variables, fixed size component variables, and constants variables.
2. **Variable Component**: This consist of the space needed by component variables whose size is dependent on the particular problems instances(Inputs/Outputs) being solved, the space needed by referenced variables and the recursion stack space is one of the most prominent components. Also this included the data structure components like Linked list, heap, trees, graphs etc.

Therefore the total space requirement of any algorithm 'A' can be provided as

```
Space(A) = Fixed Components(A) + Variable Components(A)
```

Example: Space Complexity

```
Algorithm Sum(number,size)\\ procedure will produce sum of all numbers provided in
'number' list
{
    result=0.0;
    for count = 1 to size do              \\will repeat from 1,2,3,4,....size times
        result= result + number[count];
    return result;
}
```

In above example, when calculating the space complexity we will be looking for both fixed and variable components. here we have

Fixed components as 'result','count' and 'size' variable there for total space required is three(3) words.

Variable components is characterized as the value stored in 'size' variable (suppose value store in variable 'size 'is 'n'). because this will decide the size of 'number' list and will also drive the for loop. therefore if the space used by size is one word then the total space required by 'number' variable will be 'n'(value stored in variable 'size').

therefore the space complexity can be written as **Space(Sum) = 3 + n;**

letion i.e.

Time Complexity :It is the amount of computer time it needs to run to completion. The time taken by a program is the sum of the **compile time and the run/execution time .** The compile time is independent of the instance(problem specific) characteristics.

following factors effect the time complexity:

1. Characteristics of compiler used to compile the program.
2. Computer Machine on which the program is executed and physically clocked.
3. Multiuser execution system.
4. Number of program steps.

Therefore the again the time complexity consist of two components fixed(factor 1 only) and variable/instance(factor 2,3 & 4), so for any algorithm 'A' it is provided as:

Time(A) = Fixed Time(A) + Instance Time(A)

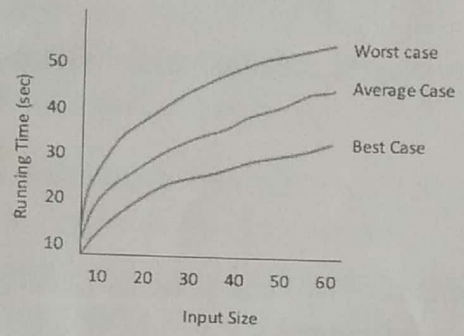
Here the number of steps is the most prominent instance characteristics and The number of steps any program statement is assigned depends on the kind of statement like

- comments count as zero steps,
- an assignment statement which does not involve any calls to other algorithm is counted as one step,
- for iterative statements we consider the steps count only for the control part of the statement etc.
- **Example**: Time Complexity

Statement	Steps per execution	Frequency	Total Steps
Algorithm Sum(number,size)	0	-	0
{	0	-	0
result=0.0;	1	1	1
for count = 1 to size do	1	size+1	size + 1
result= result + number[count];	1	size	size
return result;	1	1	1
}	0	-	0
Total			2size + 3

Best / Average/ Worst Case

- Varying input size



- The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size n. It represents the curve passing through the highest point of each column.
- The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size n. It represents the curve passing through the lowest point of each column.
- Finally, the *average-case complexity* of the algorithm is the function defined by the average number of steps taken on any instance of size n.

We can have three cases to analyze an algorithm:

1) Worst Case
2) Average Case
3) Best Case

Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be $\Theta(n)$

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity. Average Case Analysis (Sometimes done) In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

Best Case Analysis (Bogus)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\Theta(1)$