Course Code	Advanced Data Structures & Algorithms (Common to CSE, IT, CSE(DS), CSE (IoT), CSE (AI), CSE (AI & ML) and AI & DS)		L	T	P	C
20A05301T			3	0	0	3
Pre-requisite	Data Structures	Semester	III			

Course Objectives:

- Learn asymptotic notations, and analyze the performance of different algorithms.
- Understand and implement various data structures.
- Learn and implement greedy, divide and conquer, dynamic programming and backtracking algorithms using relevant data structures.
- Understand non-deterministic algorithms, polynomial and non-polynomial problems.

Course Outcomes (CO):

After completion of the course, students will be able to

- Analyze the complexity of algorithms and apply asymptotic notations.
- Apply non-linear data structures and their operations.
- Understand and apply greedy, divide and conquer algorithms.
- Develop dynamic programming algorithms for various real-time applications.
- Illustrate Backtracking algorithms for various applications.

UNIT - I	**Introduction to Algorithms**	9 Hrs

Introduction to Algorithms:
Algorithms, Pseudocode for expressing algorithms, Performance Analysis-Space complexity, Time complexity, Asymptotic Notation- Big oh, Omega, Theta notation and Little oh notation, Polynomial Vs Exponential Algorithms, Average, Best and Worst Case Complexities, Analysing Recursive Programs.

UNIT - II	**Trees Part-I**	8 Hrs

Trees Part-I
Binary Search Trees: Definition and Operations, AVL Trees: Definition and Operations, Applications.
B Trees: Definition and Operations.

UNIT - III	**Trees Part-II**	8 Hrs

Trees Part-II
Red-Black Trees, Splay Trees, Applications.
Hash Tables: Introduction, Hash Structure, Hash functions, Linear Open Addressing, Chaining and Applications.

UNIT - IV	**Divide and conquer, Greedy method**	9 Hrs

Divide and conquer: General method, applications-Binary search, Finding Maximum and minimum, Quick sort, Merge sort, Strassen's matrix multiplication.
Greedy method: General method, applications-Job sequencing with deadlines, knapsack problem, Minimum cost spanning trees, Single source shortest path problem.

UNIT - V	**Dynamic Programming & Backtracking**	9 Hrs

Dynamic Programming: General method, applications- 0/1 knapsack problem, All pairs shortest path problem, Travelling salesperson problem, Reliability design.
Backtracking: General method, applications-n-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles.
Introduction to NP-Hard and NP-Complete problems: Basic Concepts.

Textbooks:

1. Data Structures and algorithms: Concepts, Techniques and Applications, G A V Pai.
2. Fundamentals of Computer Algorithms, Ellis Horowitz, Sartaj Sahni and Rajasekharam, Galgotia publications Pvt. Ltd.

Reference Books:

1. Classic Data Structures by D. Samanta, 2005, PHI
2. Design and Analysis of Computer Algorithms by Aho, Hopcraft, Ullman 1998, PEA.
3. Introduction to the Design and Analysis of Algorithms by Goodman, Hedetniemi, TMG.

Online Learning Resources:

https://www.tutorialspoint.com/advanced_data_structures/index.asp
http://peterindia.net/Algorithms.html

CONTENTS

1	**Unit-I :**	Page NO
1.1	Introduction	1
1.2	Unit-I notes	1-16
1.3	Part A Questions	17
1.4	Part B Questions	18
2	**Unit-II :**	
2.1	Introduction	19
2.2	Unit-II notes	19 - 41
2.3	Part A Questions	42
2.4	Part B Questions	44
3	**Unit-III :**	
3.1	Introduction	45
3.2	Unit-III notes	45 - 61
3.3	Part A Questions	62
3.4	Part B Questions	64
4	**Unit-IV :**	
4.1	Introduction	65
4.2	Unit-IV notes	65 - 98
4.3	Part A Questions	99
4.4	Part B Questions	100
5	**Unit-V :**	
5.1	Introduction	101
5.2	Unit-V notes	101 - 141
5.3	Part A Questions	142
5.4	Part B Questions	145

UNIT –I

Introduction to Algorithms:

Algorithms, Pseudocode for expressing algorithms, Performance Analysis-Space complexity, Time complexity, Asymptotic Notation- Big oh, Omega, Theta notation and Little oh notation, Polynomial Vs Exponential Algorithms, Average, Best and Worst Case Complexities, Analysing Recursive Programs.

1.1. INTRODUCTION TO ALGORITHMS: WHAT IS AN ALGORITHM?

Informal Definition:

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the input into the output.

Formal Definition:

An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms should satisfy the following criteria.

- **INPUT** → Zero or more quantities are externally supplied.
- **OUTPUT** → At least one quantity is produced.
- **DEFINITENESS** → Each instruction is clear and unambiguous.
- **FINITENESS** → If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
- **EFFECTIVENESS** → Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Issues or study of Algorithm:

1. How to device or design an algorithm → creating and algorithm.
2. How to express an algorithm → definiteness.
3. How to analysis an algorithm → time and space complexity.
4. How to validate an algorithm → fitness.
5. Testing the algorithm → checking for error.

Algorithm Specification:

Algorithm can be described in three ways.

1. **Natural language like English:**

When this way is choused care should be taken, we should ensure that each & every statement is definite.

2. **Graphic representation called flowchart:**

   This method will work well when the algorithm is small& simple.

3. **Pseudo-code Method:**

   In this method, we should typically describe algorithms as program, which resembles language like Pascal & algol.

1.2. PSEUDO-CODE FOR EXPRESSING AN ALGORITHM:

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces {and}.
3. An identifier begins with a letter. The data types of variables are not explicitly declared.
4. Compound data types can be formed with records. Here is an example,

```
Node. Record
{
  data type – 1   data-1;
          .
          .
          .
  data type – n  data – n;
  node * link;
}
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

5. Assignment of values to variables is done using the assignment statement.
   <Variable>:= <expression>;

6. There are two Boolean values TRUE and FALSE.
   → Logical Operators AND, OR, NOT
   →Relational Operators <, <=,>,>=, =, !=

7. The following looping statements are employed.
   For, while and repeat-until

While Loop:

```
While < condition > do
{
        <statement-1>
                .
                .
                .
        <statement-n>          }
```

For Loop:

For variable: = value-1 to value-2 step step do

{

<statement-1>

.

.

.

<statement-n>

}

repeat-until:

repeat

<statement-1>

.

.

.

<statement-n>

until<condition>

8. A conditional statement has the following forms.

   → If <condition> then <statement>

   → If <condition> then <statement-1>

   Else <statement-1>

   Case statement:

   Case

   {

   : <condition-1> **:** <statement-1>

   .

   .

   .

   : <condition-n> **:** <statement-n>

   : else **:** <statement-n+1>

   }

9. Input and output are done using the instructions read & write.
10. There is only one type of procedure: Algorithm, the heading takes the form,

    Algorithm Name (Parameter lists)

→ As an example, the following algorithm fields & returns the maximum of 'n' given numbers:

1. algorithm Max(A,n)
2. // A is an array of size n

```
3.  {
4.  Result := A[1];
5.  for I:= 2 to n do
6.     if A[I] > Result then
7.           Result :=A[I];
8.    return Result;
9.  }
```

In this algorithm (named Max), A & n are procedure parameters. Result & I are Local variables.

→ Next we present 2 examples to illustrate the process of translation problem into an algorithm.

Selection Sort:

- Suppose we Must devise an algorithm that sorts a collection of n>=1 elements of arbitrary type.
- A Simple solution given by the following.
- (From those elements that are currently unsorted ,find the smallest & place it next in the sorted list.)

Algorithm:

```
1. For i:= 1 to n do
2. {
3.                Examine a[I] to a[n] and suppose the smallest element is at a[j];
4.                Interchange a[I] and a[j];
5.  }
```

→ Finding the smallest element (sat a[j]) and interchanging it with a[i]

- We can solve the latter problem using the code,

```
t   := a[i];
a[i]:=a[j];
a[j]:=t;
```

- The first subtask can be solved by assuming the minimum is a[I];checking a[I] with a[I+1],a[I+2]…….,and whenever a smaller element is found, regarding it as the new minimum. a[n] is compared with the current minimum.
- Putting all these observations together, we get the algorithm Selection sort.

Theorem: Algorithm selection sort(a,n) correctly sorts a set of n>=1 elements .The result remains is a a[1:n] such that a[1] <= a[2] ….<=a[n].

Selection Sort:

Selection Sort begins by finding the least element in the list. This element is moved to the front. Then the least element among the remaining element is found out and put into second position. This procedure is repeated till the entire list has been studied.

Example: List L = 3,5,4,1,2

1 is selected , → 1,5,4,3,2

2 is selected, →1,2,4,3,5

3 is selected, →1,2,3,4,5

4 is selected, →1,2,3,4,5

Proof:

- We first note that any I, say I=q, following the execution of lines 6 to 9,it is the case that a[q] Þ a[r],q<r<=n.
- Also observe that when 'i' becomes greater than q, a[1:q] is unchanged. Hence, following the last execution of these lines (i.e. I=n).We have a[1] <= a[2] <=……a[n].
- We observe this point that the upper limit of the for loop in the line 4 can be changed to n-1 without damaging the correctness of the algorithm.

Algorithm:

```
1. Algorithm selection sort (a,n)
2. // Sort the array a[1:n] into non-decreasing order.
3.{
4.      for I:=1 to n do
5.      {
6.              j:=I;
7.              for k:=i+1 to n do
8.                      if (a[k]<a[j])
9.                      t:=a[I];
10.                     a[I]:=a[j];
11.                     a[j]:=t;
12.     }
13. }
```

1.3. PERFORMANCE ANALYSIS:

1. **Space Complexity:**

   The space complexity of an algorithm is the amount of money it needs to run to compilation.

2. **Time Complexity:**

   The time complexity of an algorithm is the amount of computer time it needs to run to compilation.

Space Complexity:

Space Complexity Example:

```
Algorithm abc(a,b,c)
{
return a+b++*c+(a+b-c)/(a+b) +4.0;
}
```

→ The Space needed by each of these algorithms is seen to be the sum of the following component.

1.**A fixed part** that is independent of the characteristics (eg:number,size)of the inputs and outputs.

The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

2.**A variable part** that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that is depends on instance characteristics), and the recursion stack space.

a. The space requirement s(p) of any algorithm p may therefore be written as,

S(P) = c+ Sp(Instance characteristics) Where 'c' is a constant.

Example :

```
Algorithm sum(a,n)
{
    s=0.0;
    for I=1 to n do
    s= s+a[I];
```

```
        return s;
    }
```

- The problem instances for this algorithm are characterized by n,the number of elements to be summed. The space needed d by ‘n’ is one word, since it is of type integer.
- The space needed by ‘a’a is the space needed by variables of tyepe array of floating point numbers.
- This is atleast ‘n’ words, since ‘a’ must be large enough to hold the ‘n’ elements to be summed.
- So,we obtain Ssum(n)>=(n+s) [n for a[ ],one each for n,I a& s]

Time Complexity:

The time T(p) taken by a program P is the sum of the compile time and the run time(execution time)

→The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation .This rum time is denoted by tp(instance characteristics).

→ The number of steps any problem statemn t is assigned depends on the kind of statement.

For example, comments → 0 steps.

Assignment statements → 1 steps. [Which does not involve any calls to other algorithms]

Interactive statement such as for, while & repeat-until→ Control part of the statement.

1. **We introduce a variable, count into the program statement to increment count with initial value 0.Statement to increment count by the appropriate amount are introduced into the program.**

This is done so that each time a statement in the original program is executes count is incremented by the step count of that statement.

Algorithm:

```
Algorithm sum(a,n)
    {
      s= 0.0;
      count = count+1;
      for I=1 to n do
```

```
{
 count =count+1;
s=s+a[I];
count=count+1;
}
count=count+1;
count=count+1;
return s;
}
```

→ If the count is zero to start with, then it will be 2n+3 on termination. So each invocation of sum execute a total of 2n+3 steps.

2. The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

→First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.

→By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

Statement	S/e	Frequency	Total
1. Algorithm Sum(a,n)	0	-	0
2.{	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3

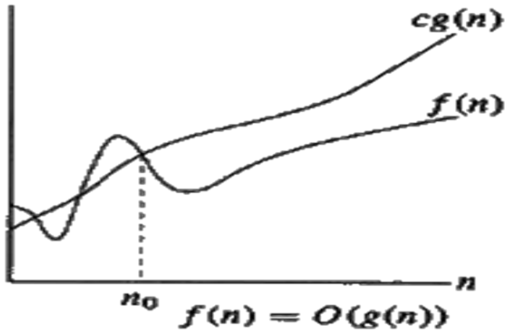
1.4. ASYMPTOTIC NOTATIONS

There are different kinds of mathematical notations used to represent time complexity. These are called Asymptotic notations. They are as follows:

1. Big oh(O) notation
2. Omega(Ω) notation
3. Theta(ɵ) notation

1. Big oh(O) notation:

- Big oh(O) notation is used to represent upperbound of algorithm runtime.
- Let f(n) and g(n) are two non-negative functions
- The function f(n) = O(g(n)) if and only if there exists positive constants c and n_0 such that $f(n) \leq c*g(n)$ for all n , $n \geq n_0$.



Example:

If f(n)=3n+2 then prove that f(n) = O(n)

Let f(n) =3n+2, c=4, g(n) =n

if n=1 $3n+2 \leq 4n$

$3(1)+2 \leq 4(1)$

$3+2 \leq 4$

$5 \leq 4$ (F)

if n=2 $3n+2 \leq 4n$

$3(2)+2 \leq 4(2)$

$8 \leq 8$ (T)

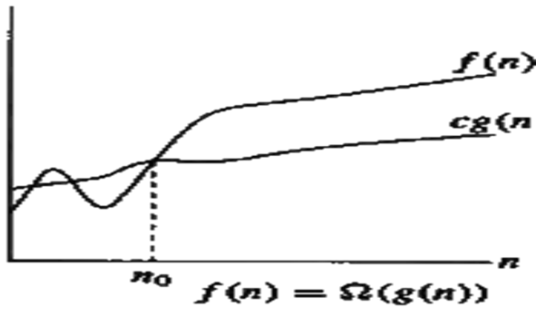
$3n+2 \leq 4n$ for all $n \geq 2$

This is in the form of $f(n) \leq c*g(n)$ for all $n \geq n_0$, where c=4, n_0 =2

Therefore, f(n) = O(n),

2. Omega(Ω) notation:

- Big oh(O) notation is used to represent lowerbound of algorithm runtime.
- Let f(n) and g(n) are two non-negative functions
- The function f(n) = Ω(g(n)) if and only if there exists positive constants c and n_0 such that $f(n) \geq c*g(n)$ for all n , $n \geq n_0$.



Example

f(n)=3n+2 then prove that f(n) = Ω(g(n))

Let f(n) =3n+2, c=3, g(n) =n

if n=1 $3n+2 \geq 3n$

$3(1)+2 \geq 3(1)$

$5 \geq 3$ (T)

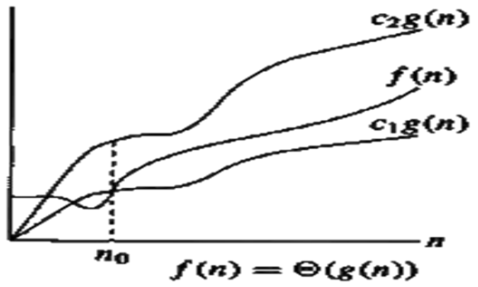
$3n+2 \geq 4n$ for all $n \geq 1$

This is in the form of $f(n) \geq c*g(n)$ for all $n \geq n_0$, where $c=3$, $n_0=1$

Therefore, $f(n) = \Omega(n)$.

3. Theta(ɵ) notation:

- Theta(ɵ) notation is used to represent the running time between upper bound and lower bound.
- Let f(n) and g(n) be two non-negative functions.
- The function $f(n) = \theta(g(n))$ if and only if there exists positive constants c_1 , c_2 and n_0 such that $c_1*g(n) \leq f(n) \leq c_2* g(n)$ for all n, $n \geq n_0$.



Example:

f(n)=3n+2 then Prove that f(n) = θ(g(n))

Lower bound = $3n+2 \geq 3n$ for all $n \geq 1$

$c_1=3, g(n)=n, n_0=1$

Upper Bound = $3n+2 \leq 4n$ for all $n \geq 2$

c2=4, g(n)=n , n0=2

$3(n) \leq 3n+2 \leq 4(n)$ for all n, $n \geq 2$

This is in the form of $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0$ Where $c_1=3$, $c_2=4$, $g(n)=n$, $n_0=2$

Therefore $f(n)= \theta(n)$

1.5. POLYNOMIAL VS EXPONENTIAL ALGORITHMS

The **time complexity**(generally referred as running time) of an algorithm is expressed as **the amount of time taken by an algorithm for some size of the input to the problem**. **Big O notation** is commonly used to express the time complexity of any algorithm as this suppresses the lower order terms and is described asymptotically. Time complexity is estimated by counting the operations(provided as instructions in a program) performed in an algorithm. Here each operation takes a fixed amount of time in execution. **Generally time complexities are classified as constant, linear, logarithmic, polynomial, exponential etc**. Among these the **polynomial and exponential are the most prominently considered** and defines the complexity of an algorithm. These two parameters for any algorithm are always influenced by size of input.

Polynomial Running Time

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is $O(n^k)$ for some non-negative integer k, where n is the complexity of the input. Polynomial-time algorithms are said to be "fast." Most familiar mathematical operations such as addition, subtraction, multiplication, and division, as well as computing square roots, powers, and logarithms, can be performed in polynomial time. Computing the digits of most interesting mathematical constants, including pi and e, can also be done in polynomial time.

All basic arithmetic operations ((i.e.) Addition, subtraction, multiplication,division), comparison operations, sort operations are considered as polynomial time algorithms.

Exponential Running Time

The set of problems which can be solved by an exponential time algorithms, but for which no polynomial time algorithms is known.

An algorithm is said to be exponential time, if $T(n)$ is upper bounded by $2^{\text{poly}(n)}$, where poly(n) is some polynomial in n. More formally, an algorithm is exponential time if $T(n)$ is bounded by $O(2^{nk})$ for some constant k.

Algorithms which have exponential time complexity grow much faster than polynomial algorithms.

The difference you are probably looking for happens to be where the variable is in the equation that expresses the run time. Equations that show a **polynomial time** complexity have variables in the bases of their terms.

Examples: $n^3 + 2n^2 + 1$. Notice n is in the base, NOT the exponent.

In exponential equations, the variable is in the exponent.

Examples: 2^n. As said before, exponential time grows much faster. If n is equal to 1000 (a reasonable input for an algorithm), then notice 1000^3 is 1 billion, and 2^{1000} is simply huge! For a reference, there are about 2^{80} hydrogen atoms in the sun, this is much more than 1 billion.

1.6.AVERAGE, BEST AND WORST CASE COMPLEXITIES

Best case: This analysis constrains on the input, other than size. Resulting in the fasters possible run time

Worst case: This analysis constrains on the input, other than size. Resulting in the fasters possible run time

Average case: This type of analysis results in average running time over every type of input.

Complexity: Complexity refers to the rate at which the storage time grows as a function of the problem size.

1.7. ANALYSING RECURSIVE PROGRAMS.

For every recursive algorithm, we can write recurrence relation to analyse the time complexity of the algorithm.

Recurrence relation of recursive algorithms

A recurrence relation is an equation that defines a sequence where any term is defined in terms of its previous terms.

The recurrence relation for the time complexity of some problems are given below:

Fibonacci Number

$T(N) = T(N-1) + T(N-2)$

Base Conditions: $T(0) = 0$ and $\mathbf{T}(1) = 1$

Binary Search

$T(N) = T(N/2) + C$

Base Condition: $T(1) = 1$

Merge Sort

$T(N) = 2\ T(N/2) + CN$

Base Condition: $T(1) = 1$

Recursive Algorithm: Finding min and max in an array

$T(N) = 2\ T(N/2) + 2$

Base Condition: $T(1) = 0$ and $\mathbf{T}(2) = 1$

Quick Sort

$T(N) = T(i) + T(N-i-1) + CN$

The time taken by quick sort depends upon the distribution of the input array and partition strategy. T(i) and T(N-i-1) are two smaller subproblems after the partition where i is the number of elements that are smaller than the pivot. **CN** is the time complexity of the partition process where **C** is a constant. .

Worst Case: This is a case of the unbalanced partition where the partition process always picks the greatest or smallest element as a pivot(Think!).For the recurrence relation of the worst case scenario, we can put i = 0 in the above equation.

$$T(N) = T(0) + T(N-1) + CN$$

which is equivalent to

$$\mathbf{T}(N) = T(N-1) + CN$$

Best Case: This is a case of the balanced partition where the partition process always picks the middle element as pivot. For the recurrence relation of the worst case scenario, put i = N/2 in the above equation.

$$T(N) = T(N/2) + T(N/2-1) + CN$$

which is equivalent to

$$\mathbf{T}(N) = 2T(N/2) + CN$$

Average Case: *For* average case analysis, we need to consider all possible permutation of input and time taken by each permutation.

$$T(N) = (\textbf{for}\ i = 0\ \text{to}\ N-1) \sum (\ T(i) + T(N-i-1)\ ) / N$$

Note: This looks mathematically complex but we can find several other intuitive ways to analyse the average case of quick sort.

Analyzing the Efficiency of Recursive Algorithms

Step 1: Identify the number of sub-problems and a parameter (or parameters) indicating an input's size of each sub-problem (function call with smaller input size)

Recursive Problems	Input Size	Number of Subproblems	Input size of Subproblems
Finding nth Fibonacci	N	2	(N-1) and (N-2)
Binary Search	N	1	N/2
Merge Sort	N	2	N/2 each
Recursive: Finding min and max	N	2	N/2 each
Karastuba algorithm	N (Total number of digits in each Integer)	3	N/2 each
Quick Sort	N	2	(i) and (n-i-1)
Strassen's Matrix Multiplication	N (Size of each matrix)	7	N/2 each
Recursive: Longest Common Subsequence	(N, M) length of both the subsequences	3	(N-1, M-1), (N-1, M) and (N, M-1)

Step 2: Add the time complexities of the sub-problems and the total number of basic operations performed at that stage of recursion.

Step3: Set up a recurrence relation, with a correct base condition, for the number of times the basic operation is executed.

Recursive Problems	Time Complexity of the subproblems	Total Number of basic operations	Recurrence Relation
Finding nth Fibonacci	T(N-1) + T (N-2)	O(1) for one addition	T(N) = T(N-1) + T(N-2) + C
Binary Search	T(N/2)	O(1) for one comparison	T(N) = T(N/2) + C
Merge Sort	2 T(N/2)	O(N) for merging two sorted halves	T(N) = 2 T(N/2) + CN
Finding min and max (Recursive)	2 T(N/2)	O(1) for one comparison	T(N) = 2 T(N/2) + C
Karatsuba algorithm	3T(N/2)	O(N) (How? Think)	T(N) = 3 T(N/2) + CN
Quick Sort	T(i) + T(N - i - 1)	O(N) for partition process	T(N) = T(i) + T(N - i - 1) + CN
Strassen's Matrix Multiplication	7 T(N/2)	O(N^2) for addition and subtraction of two matrices	T(N) = 7 T(N/2) + CN^2
Recursive: Longest Common Subsequence	T(N-1,M-1), if (A[M-1] = B[N-1]) T(N-1,M)+T(N, M-1), otherwise	O(1) for one comparison	T(N,M) = T(N-1,M-1)+O(1), if (X[M-1] = Y[N-1]) = T(N-1,M)+T(N, M-1) +O(1), otherwise

Step4: Solve the recurrence or, at least, ascertain the order of growth of its solution. There are several ways to analyse the recurrence relation but we are discussing here two popular approaches of solving recurrences:

- **Method 1**: Recursion Tree Method
- **Method 2**: Master Theorem

Method 1: Recursion Tree Method

A recurrence tree is a tree where each node represents the cost of a certain recursive subproblem. We take the sum of each value of nodes to find the total complexity of the algorithm.

Steps for solving a recurrence relation

1. Draw a recursion tree based on the given recurrence relation.
2. Determine the number of levels, cost at each level and cost of the last level.
3. Add the cost of all levels and simplify the expression.

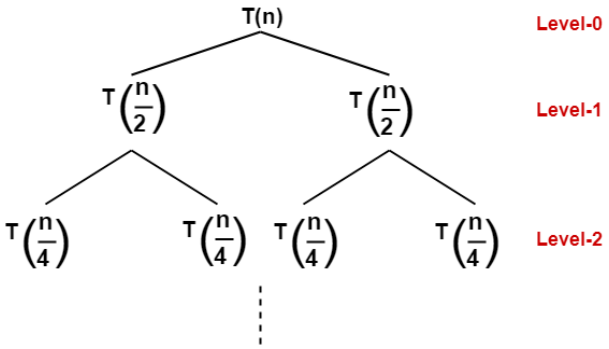
Let us solve the given recurrence relation by Recurrence Tree Method

$$T(N) = 2*T(N/2) + CN$$

From the above recurrence relation, we can find that

1. The problem of size N is divided into two sub-problems of size N/2.
2. The cost of dividing a sub-problem and then combining its solution of size N is CN.
3. Each time, the problem will be divided into half, until the size of the problem becomes 1.

The recursion tree for the above relation will be



Method 2: Master theorem

Master theorem states that for a recurrence relation of form

$T(N) = aT(N/b) + f(N)$ where $a >= 1$ and $b > 1$

If **f**$(N) = O(N^k)$ and $k \geq 0$, then

Case 1: T(N) = O(N^logb(a)), **if** k < logb(a).

Case 2: T(N) = O((N^k)*logN), **if** k = logb(a).

Case 3: T(N) = O(N^k), **if** k > logb(a)

Example 1

T(N) = T(N/2) + C

The above recurrence relation is of binary search. Comparing this with master theorem, we get a = 1, b = 2 and k = 0 because f(N) = C = C(N^0)

Here logb(a) = k, so we can apply case 2 of the master theorem.

T(n) = (N^0*log(N)) = O(logN).

Example 2

T(N) = 2*T(N/2) + CN

The above recurrence relation is of **merge sort**. Comparing this with master theorem,a = 2, b = 2 and f(N) = CN. Comparing left and right sides of f(N), we get k = 1.

logb(a) = log2(2) = 1 = K

So, we can apply the case 2 of the master theorem.

=> T(N) = O(N^1*log(N)) = O(NlogN).

PART-A (2 Marks)

1. **What is performance measurement?**

Ans. Performance measurement is concerned with obtaining the space and the time requirements of a particular algorithm.

2. **What is an algorithm?**

Ans. An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.

3.What are the characteristics of an algorithm?

Ans. 1) Input

2) Output

3) Definiteness

4) Finiteness

5) Effectiveness

4. What is recursive algorithm?

Ans. An algorithm is said to be recursive if the same algorithm is invoked in the body. An algorithm that calls itself is direct recursive. Algorithm A is said to be indeed recursive if it calls another algorithm, which in turn calls A.

5.What is space complexity?

Ans. The space complexity of an algorithm is the amount of memory it needs to run to completion.

6.What is time complexity?

Ans. The time complexity of an algorithm is the amount of computer time it needs to run to completion.

7.Define the asymptotic notation "Big Oh" (O) ,"Omega" (Ω) and "theta" (ɵ)

Ans. Big Oh(O) : The function f(n) = O(g(n)) iff there exist positive constants C and no such that f(n) ≤ C * g(n) for all n, n ≥ n0.

Omega (Ω) : The function f(n) =Ω(g(n)) iff there exist positive constant C and no such that f(n) ≥ C * g(n) for all n, n ≥ n0.

theta(ɵ) :The function f(n) = ɵ (g(n)) iff there exist positive constant C1, C2, and no such that C1 *g(n) ≤ f(n) ≤ C2* g(n) for all n, n ≥ n0.

PART-B (10 Marks)

1. Write the merge sort algorithm. Find out the best, worst and average cases of this algorithm. Sort the following numbers using merge sort:

   10, 12, 1, 5, 18, 28, 38, 39, 2, 4, 7

2. What is asymptotic notation? Explain different types of notations with example.

3. **Solve** the following recurrence relation $T(n) = 7T(n/2)+cn^2$

4. **Solve** the following recurrence relation $T(n) = \left\{2T\left(\frac{n}{2}\right) + 1, \quad and\ T(1) = 2\right.$

5. **Define** the term algorithm and state the criteria the algorithm should satisfy.

6. If $f(n)=5n^2 + 6n + 4$, then **prove** that f(n) is $O(n^2)$.

7. **Use** step count method and analyze the time complexity when two n×n matrices are added.

8. **Describe** the role of space complexity and time complexity of a program ?

9. **Discuss** various the asymptotic notations used for best case average case and worst case analysis of algorithms.