GUDLAVALLERU ENGINEERING COLLEGE

(An Autonomous Institute with Permanent Affiliation to JNTUK, Kakinada)

Seshadri Rao Knowledge Village, Gudlavalleru – 521 356.

Department of Computer Science and Engineering

HANDOUT

on

DESIGN AND ANALYSIS OF ALGORITHMS

III Year - II Semester 2019-20 CSE

Vision

To be a Centre of Excellence in computer science and engineering education and training to meet the challenging needs of the industry and society

Mission

- To impart quality education through well-designed curriculum in tune with the growing software needs of the industry.
- To be a Centre of Excellence in computer science and engineering education and training to meet the challenging needs of the industry and society.
- To serve our students by inculcating in them problem solving, leadership, teamwork skills and the value of commitment to quality, ethical behavior & respect for others.
- To foster industry-academia relationship for mutual benefit and growth.

Program Educational Objectives

- Identify, analyze, formulate and solve Computer Science and Engineering problems both independently and in a team environment by using the appropriate modern tools.
- Manage software projects with significant technical, legal, ethical, social, environmental and economic considerations
- Demonstrate commitment and progress in lifelong learning, professional development, leadership and Communicate effectively with professional clients and the public.

HANDOUT ON DESIGN AND ANALYSIS OF ALGORITHMS

Class & Sem. :III B.Tech – II Semester Year : 2019-20
Branch : CSE & IT Credits : 3

==========

1. Brief History and Scope of the Subject

Algorithms play the central role both in the science and practice of computing. Recognition of this fact has led to the appearance of a considerable number of textbooks on the subject. By and large, they follow one of two alternatives in presenting algorithms. One classifies algorithms according to a problem type. Such a book would have separate chapters on algorithms for sorting, searching, graphs, and so on. The advantage of this approach is that it allows an immediate comparison of, say, the efficiency of different algorithms for the same problem. The drawback of this approach is that it emphasizes problem types at the expense of algorithm design techniques.

An algorithm is a receipe or a systematic method containing a sequence of instructions to solve a computational problem. It takes some inputs, performs a well defined sequence of steps, and produces some output. Once we design an algorithm, we need to know how well it performs on any input. In particular we would like to know whether there are better algorithms for the problem. An answer to this first demands a way to analyze an algorithm in a machine-independent way. Algorithm design and analysis form a central theme in computer science

2. Pre-Requisites

- Data structures using C
- Discrete mathematical structures

3. Course Objectives:

- Analyze the asymptotic performance of algorithms.
- Apply efficient algorithmic design paradigms.

4. Course Outcomes:

At the end of the course, the students will be able to

CO1: Analyze the time and space complexity of an algorithm.

CO2: Perform union and find operations on disjoint sets.

CO3: Find bi-connected components of a graph.

CO4: Identify algorithm design technique to solve problems.

5. Program Outcomes:

Graduates of the Computer Science and Engineering Program will have an ability to

a. apply knowledge of computing, mathematics, science and engineering fundamentals to solve complex engineering problems.

b. formulate and analyze a problem, and define the computing requirements appropriate to its solution using basic principles of mathematics, science and computer engineering.

c. design, implement, and evaluate a computer based system, process, component, or software to meet the desired needs.

d. design and conduct experiments, perform analysis and interpretation of data and provide valid conclusions.

e. use current techniques, skills, and tools necessary for computing practice.

f. understand legal, health, security and social issues in Professional Engineering practice.

g. understand the impact of professional engineering solutions on environmental context and the need for sustainable development.

h. understand the professional and ethical responsibilities of an engineer.

i. function effectively as an individual, and as a team member/ leader in accomplishing a common goal.

j. communicate effectively, make effective presentations and write and comprehend technical reports and publications.

k. learn and adopt new technologies, and use them effectively towards continued professional development throughout the life.

l. understand engineering and management principles and their application to manage projects in the software industry.

6. Mapping of Course Outcomes with Program Outcomes:

	a	b	c	d	E	f	g	h	i	j	k	l
CO1												
CO2												
CO3												
CO4												

7. Prescribed Text Books

1. Ellis Horowitz,Satraj Sahni and Rajasekharam, Fundamentals of Computer Algorithms, Galgotia publications pvt. Ltd.
2. Aho, Ullman and Hopcroft, Design and Analysis of algorithms, Pearson education.

8. Reference Text Books

1. T.H.Cormen,C.E.Leiserson, Introduction to Algorithms, PHI Pvt. Ltd./ Pearson Education, 2nd edition.
2. Allen Weiss, Data structures and Algorithm Analysis in C++, Pearson education, 2nd edition.
3. M.T.Goodrich,R.Tomassia,John wiley and sons, Algorithm Design: Foundations, Analysis and Internet examples.
4. Steven S .Skiena, The algorithm Design Manual, 2nd edition, Springer.

9. URLs and Other E-Learning Resources

URLs:

- **http://freevideolectures.com/Course/2281/Design-and-Analysis-of-Algorithms**
- **http://nptel.ac.in/courses/106101060/#**
- **http://nptel.ac.in/video.php?subjectId=106101060**
- **https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/lecture-videos/**

E-Learning Materials:

Journals:

INTERNATIONAL JOURNALS:

- IEEE transactions on evolutionary computation.
- ACM transactions on Algorithms.

NATIONAL JOURNALS:

- Journal of Discrete algorithms.
- Journal of Graph Algorithms and applications.

10. Digital Learning Materials:

a. SONET CDs – Design and analysis of Algorithms

b. IIT CDs - Design and analysis of Algorithms

11. Lecture Schedule / Lesson Plan

Topic	No. of Periods	
	Theory	Tutorial
UNIT –1: Introduction		
Algorithm	1	1
Pseudo code for expressing algorithms	2	
Performance Analysis-space complexity	2	
Time complexity	2	1
Asymptotic Notations- Big oh, Omega	2	
Theta and Little oh notations.	1	
UNIT – 2: Disjoint sets & Divide and conquer Disjoint Sets		

Disjoint sets	1	1
union and find algorithms	2	
Bi-connected components	2	2
Divide and conquer - General method	2	
Applications-Binary search, Quick sort, Merge sort.	3	
UNIT – 3: Greedy method		
General method	1	1
knapsack problem	2	
Job sequencing with deadlines	2	1
Minimum cost spanning trees.	3	
UNIT – 4: Dynamic Programming		
General method	1	1
Matrix chain multiplication	2	
Optimal binary search trees	2	2
0/1 knapsack problem	2	
Travelling sales person problem	2	
UNIT – 5: Backtracking		
General method	1	1
n queens problem	2	
Sum of subsets problem	2	1
Graphcoloring, Hamiltonian cycle.	2	
UNIT – 6: Branch and Bound		
General method	1	1
Applications	2	
Travelling sales person problem,	2	1
0/1 knapsack problem - LC BB, FIFO BB solutions.	2	
Total No.of Periods:	**58**	**14**

UNIT – I

Objective:

- To analyze the time and space complexity of an algorithm.

Syllabus:

Introduction Algorithm, Pseudo code for expressing algorithms, Performance Analysis-Space and Time complexity, Asymptotic Notation- Big oh, Omega, Theta and Little oh notations.

Learning Outcomes:

At the end of the course, Students will be able to

- Design an algorithm for a problem.
- Analyze the time and space complexity of an algorithm.
- Apply asymptotic notations to express time and space complexity of an algorithm.

Learning Material

UNIT - I
INTRODUCTION

1.1Algorithm

- The word "**Algorithm"** came from the name of a Persian aithor "**Abu Jafar Mohammad Ibn Musa Al Khowarizmi**".
- It refers to a "method that can be used by a computer for the **solution** of a problem".

Definition:

- An **Algorithm** is **a finite set of instructions that if followed, accomplishes a particular task.**

Characteristics of an algorithm:

- **Input:** An algorithm has zero or more but only finite number of inputs.
- **Output:** At least one quantity must be produced as output.
- **Definiteness:** Each instruction must be clear and ambiguous. It must be clear what should be done.
    - For example consider the instruction: **add 6 or 7 to x.** It is not clear about which one of the two possibilities should be done.
    - For example consider the instruction: **compute 5/0.** It is not clear because it result in infinite value.
- **Finiteness:** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
    - An algorithm must terminate after a finite number of operations, but the time for termination should be reasonably short.
    - For example, an algorithm could be devised that decides whether any given position in a game of chess is a winning position. An algorithm works by examining all possible positions and counter moves, but this takes billions of years to make a decision, even with more modern computers. So we say that algorithm is not finite.

- **Effectiveness:** Every instruction must be very basic so that it can be carried out by a person using only pencil and paper.
  - Each operation must be feasible and effective and can be done in a finite amount of time.
  - For example, performing integer arithmetic is effective but performing real arithmetic is not effective as it involves long decimal expansions.

1.2 Pseudo code for expressing Algorithms

- An algorithm can be expressed in many ways. We can use a natural language like English but the instructions must be finite.
- We can use graphical representations like **Flow Chart,** but it works well only if the algorithms are small and simple.
- **Pseudo code** is a compact and informal high-level description that uses the structural conventions for expressing algorithms.
- The fol1owing are the conventions of pseudo code belonging to C and Pascal:
  - **Comments** begin with // and continue until the end of line.
  - **Blocks** are indicated with matching braces :{ and}. A compound statement(i.e.,a collection of simple statements) can be represented as a block. The body of a procedure also forms a block. Statements are delimited by semicolon(;).
  - An **identifier** begins with a letter. The data types of variables are not explicitly declared. The types will be clear from the context..We assume simple data types such as integer, float, char, boolean, and so on. Compound data types can be formed with records.

    Here is an example: node= record { datatype_1 data_1;
    datatype_n data_n;
    node *link;
    }

    In this example, link is a pointer to the recordtype node.

    Individual data items of a record can be accessed with -> and period. For instance if p points to a record of type node, p->data_1 stands for the value of the first field

in the record. On the otherhand, if q is a record of type node, q.data_1 will denote its first field.

- **Assignment** of values to variables is done using the assignment statement **(variable):=(expression);**
- There are two **Boolean values true and false**. In order to produce these values, the logical operators and, or, and not and the relational operators < and > are provided.
- Elements of multidimensional arrays are accessed using [and]. For example, if A is a two dimensional array, the (i, j) th element of the array is denoted as - A[i,j].Array indices start at zero.
- The following looping statements are employed: **for, while, and repeat- until**.
  - ❖ The **while loop** takes the following form:

While (condition) do
{
<statement 1)>
…….
<statement n>
}

As long as <condition> is true, the statements get executed. When <condition> becomes false, the loop is exited. The value of <condition> is evaluated at the top of the loop.

- ❖ The general form of a **for loop** is:

for variable:=value l to value2 step step do
{
<statement 1>
……
<statement n>
}

- ❖ Here valuel, value2, and step are arithmetic expressions.

- A variable of type integer or real or a numerical constant is a simple form of an arithmetic expression.
- The clause\"step step\" is optional and taken as +1 if it does not occur, step could either be positive or negative.
- Variable is tested for termination at the start of each iteration. The for loop can be implemented as a while loop as follows:

variable:=value1;

fin :=value2;

incr:=step;

while ((variable- fin) * step<=0) do

{

<statement 1>

……..

<statement n>;

Variable:=variable+ incr;

}

- A **repeat-until** statement is constructed as follows:

repeat

<statement 1>

…….

<statement n>

until <condition>

- The statements are executed as long as <condition> is false.
- The value of <condition> is computed after executing the statements.
- The instruction **break** can be used within any of the above looping instructions to force exit.
- In case of nested loops, break results in the exit of the innermost loop that it is a part of.

- ❖ A return statement within any of the above also will result in exiting the loops.
- ❖ A return statement results in the exit of the function itself.

- A **conditional statement** has the following forms:

  if <condition> then <statement>

  if <condition> then <statement 1> else <statement 2>

  Here <condition> is a Boolean expression and <statement>, <statement1> and <(statement2> are arbitrary statements (simple or compound).

  case statement is as follows:

  case

  {

  :< condition 1> : <statement1>

  …….

  : <condition n> : <statement n>

  : else : <statement n+1>

  - ❖ Here <statement 1>, <statement 2>, etc. could be either simple statements or compound statements.
  - ❖ A case statement is interpreted as follows. If <condition1> is true, <statement 1> gets executed and the case statement is exited.
  - ❖ If <condition1> is false, <condition2> is evaluated.
  - ❖ If <condition2> is true, <statement2> gets executed and the case statement exited, and so on.
  - ❖ If none of the conditions (condition1),..., (condition n) are true,(statementn+1)is executed and the case statement is exited.
  - ❖ The else clause is optional.

- **Input and output** are done using the instructions: **read and write**. No format is used to specify the size of input or output quantities.
- There is only one type of procedure: **Algorithm**. An algorithm consists of a heading and a body. The heading takes the form

  Algorithm Name (<parameterlist>)

- ❖ where Name is the name of the procedure and <parameter list> is a listing of the procedure parameters.
- ❖ The body has one or more (simple or compound) statements enclosed within braces {and}.An algorithm may or may not return any values.
- ❖ Simple variables to procedures are passed by value. Arrays and records are passed by reference.
- ❖ An array name or a record name is treated as a pointer to the respective data type.

- **Example**: The following algorithm finds and returns the maximum of n given numbers:

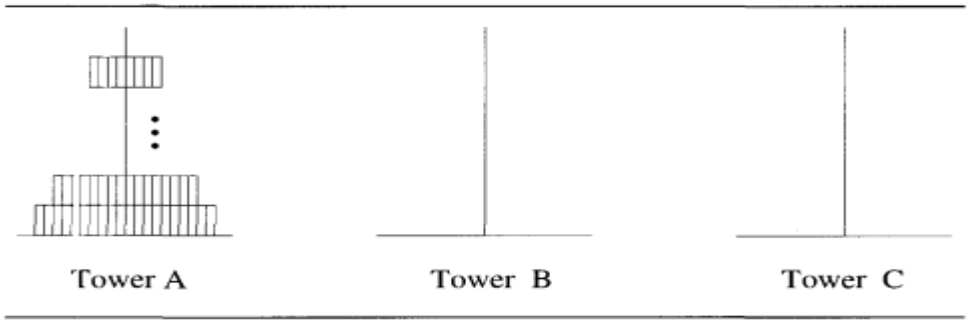
```
Algorithm Max (A, n)
// A is an array of size n.
{
Result:=A[l];
for i :=2 ton do
if A[i] >Result then Result:=A[i];
return Result;
}
```

In this algorithm is named Max, A and n are procedure parameters. Result and i are local variables.

1.3 Recursive Algorithms

- 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 **indirect recursive** if it calls another algorithm which in turn calls A.
- The recursive mechanisms are extremely powerful and they can express a complex process very clearly.
- An algorithm that can be written using **assignment, if-then-else statement, and while statement** can also be written using **assignment, the if-then-else statement, and recursion.**
- **Example: Towers of Hanoi**

- At the time the world was created, there was a diamond tower (labeled A) with 64 golden disks.
- The disks were of decreasing size and were stacked on the tower A in decreasing order of size from bottom to top.
- Besides this tower there were two other diamond towers (labeled B and C).
- We have to move the disks from tower A to tower B using tower C for intermediate storage.
- As the disks are very heavy, they can be moved only one at a time.
- In addition, at no time can a disk be on top of a smaller disk.



- A solution results from the use of recursion.
- Assume that the number of disks is n.
- To get the largest disk to the bottom of tower B, we move the remaining n-1disks to tower C and then move the largest to tower B.
- Now we are left with the task of moving the disks from tower C to tower B.
- To do this, we have towers A and B available.

```
1   Algorithm TowersOfHanoi(n, x, y, z)
2   // Move the top n disks from tower x to tower y.
3   {
4       if (n ≥ 1) then
5       {
6           TowersOfHanoi(n − 1, x, z, y);
7           write ("move top disk from tower", x,
8           "to top of tower", y);
9           TowersOfHanoi(n − 1, z, y, x);
10      }
11  }
```

1.4 Performance Analysis

- There are two criteria for judging the performance of an algorithm:
  - Computing time
  - Storage Requirements
- Performance evaluation can be loosely divided into two major phases:

  (1) **Priori estimates** referred to as **performance analysis**

  (2) **Posteriori testing** referred to as **performance measurement.**
- The **space complexity** of an algorithm is the amount of memory it needs to run to completion.
- The **time complexity** of an algorithm is the amount of computer time it needs to run to completion.

1.5 Space Complexity

- The space needed by the algorithm is the sum of the following components:
  - **A fixed part** that is independent of the characteristics (e.g., number, size) of the inputs and outputs.
  - This part typically includes the instruction space (i.e., space for the code), space for simple variables and fixed-size component variables (also called aggregate), space for constants, and soon.
  - **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 this depends on instance characteristics), and the recursion stack space (insofar as this space depends on the instance characteristics)
- The **space requirement S(P)** of any algorithm P may be written as :

  S (P)= c + Sp (instance characteristics), where c is a constant.

  When analyzing the space complexity of an algorithm, we concentrate on estimating Sp (instance characteristics).

For any given problem, we need first to determine which instance characteristics to use to measure the space requirements.

- **Example 1:**

Algorithm abc computes a+ b+b*c+(a+ b -c)/(a+6)+4.0;

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

- The problem instance is characterized by the specific values of a, b, and c.
- Making the assumption that one word is adequate to store the values of each of a, b, c, and the result, we see that the space needed by abc is independent of the instance characteristics.
- Consequently, **Sp (instance characteristics)=0.**

- **Example 2:**

Algorithm Sum compute sum of a[1] to a[n] iteratively, where the a[i]'s are real numbers

```
Algorithm Sum (a,n)
{
s :=0.0;
for i :=1 to n do
s :=s+ a[i];
returns;
}
```

- The problem instances for are characterized by n, the number of elements to be summed.
- The space needed by n is one word, since it is of type integer.
- The space needed by a is the space needed by variables of type array of floating point numbers.
- This is at least n words, since a must be large enough to hold the n elements to be summed.

- So,we obtain Ssum(n) >= (n +3) (n for a[ ]one each fo r n, i and s).

1.6 Time Complexity

- The **time T(P)** taken by a program P is the **sum of the compile time and the run (or execution)time.**
- The **compile time does not depend on the instance characteristics**. We assume that a compiled program will be run several times without recompilation. This run time is denoted by tp (instance characteristics).
- The factors on which **tp depends on are not known at the time a program is conceived**, it is reasonable to attempt only to **estimate tp**.
- If we knew the characteristics of the compiler to be used, we could proceed to **determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores** and so on, that would be made by the code for P.
- An expression for tp(n) is of the form :

  tP (n) = caADD(n) + csSUB(n) + cmMUL(n)+ cdDIV(n)+…….

  where n denotes the instance characteristics, and ca,cs,cm, cd, and so on, respectively, denote the time needed for an addition, subtraction, multiplication, division, and soon, and ADD, SUB, MUL, DIV, and so on, are functions whose values are the numbers of additions, subtractions, multiplications, divisions, and soon, that are performed when the code for P is used on an instance with characteristic n.
- The time needed for an addition, subtraction, multiplication, and soon, often depends on the numbers being added, subtracted, multiplied, and soon. The value of tp(n) for any given n can be obtained only experimental.
- The program is typed, compiled, and run on a particular machine. The execution time is **physically clocked, and tp(n) obtained.**
- In a **multi user system**, the execution time depends on such factors as **system load ,the number of other programs running on the computer** at the time program P is run, the characteristics of these other programs, and soon.
- Since the exact number of additions, subtractions, and so on, that are needed to solve a problem instance with characteristics given by n is difficult, we can calculate the operations together (provided that the time required by each is relatively independent of

the instance characteristics)and obtain a count for the total number of operations. We can go one step further and count only the number of program steps.

- A **program step** is defined as a segment of a program that has an execution time that is independent of the instance characteristic.

  For example, the entire statement return **a + b+b*c+(a+ b-c)/(a+ b) + 4.0;**

  This statement can be regarded as a step since its execution time is independent of the instance characteristic.

- The **number of steps any program statement** is assigned depends on the **kind of statement**.
  - **Comments** count as zero steps.
  - **An assignment statement** which does not involve any calls to other algorithms is counted as one step.
  - In an **iterative statement** such as the **for, while, and repeat-until statements**, we consider the **step counts only for the control part of the statement**.
    - The control parts for **for and while statements** have the following forms:

      for i := <expr> to <exprl> do

      while (<expr>)do
    - Each execution of the **control part of a while statement** is given a step count, equal to the number of step counts assignable to <expr>.
    - The step count for each execution of the **control part of a for statement** is one, unless the counts attributable to <expr>and <exprl> are functions of the instance characteristics. The first execution of the control part of the for has a step count equal to the sum of the counts for <expr> and <exprl>. Remaining executions of for statement have a step count of one; and so on.

- We can **determine the number of steps** needed by a program to solve a particular problem by using one of the two methods as follows:

Method1: Count method

- We introduce a new variable**, count**, into the program. This is a global variable with **initial value 0**. Statements 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 executed; count is incremented by the step count of that statement.

- **Example1:** By including a global variable **Count (Iterative Algorithm)**

```
Algorithm Sum (a,n)
{
 s:=0.0;
 count :=count+ 1; // count is global; it is initially zero.
for i :=1 to n do
{
count :=count+ 1; // For for
 s:=s+ a[i];
count :=count+ 1; // For assignment
 }
count :=count+ 1; // For last time of for
count:=count+ 1; // For the return returns;
}
```

- For the **for** loop, the value of count will increase by a total of **2n**.
- If count is zero to start with, then it will be 2n + 3 on termination.
- So each invocation of Sum executes a total of **2n + 3** steps.

- **Example 2: (Recursive Algorithm)**

```
Algorithm RSum(a, n)
{
    count := count + 1; // For the if conditional
    if (n ≤ 0) then
    {
        count := count + 1; // For the return
        return 0.0;
    }
    else
    {
        count := count + 1;  // For the addition, function
                             // invocation and return
        return RSum(a, n − 1) + a[n];
    }
}
```

- When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count.
- Let tRSum(n) be the increase in the value of count when the Algorithm terminates.
- We see that tRSum(0) = 2.When n >0, count increases by 2 .

$$t_{\mathsf{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\mathsf{RSum}}(n-1) & \text{if } n > 0 \end{cases}$$

- These recursive formulas are referred to as **recurrence relations**.
- One way of solving any such recurrence relation is to make repeated substitutions for each occurrence of the function t_{RSum} on the right-hand side until all such occurrences disappear.

$$\begin{aligned} t_{\mathsf{RSum}}(n) &= 2 + t_{\mathsf{RSum}}(n-1) \\ &= 2 + 2 + t_{\mathsf{RSum}}(n-2) \\ &= 2(2) + t_{\mathsf{RSum}}(n-2) \\ &\vdots \\ &= n(2) + t_{\mathsf{RSum}}(0) \\ &= 2n + 2, \qquad\qquad n \geq 0 \end{aligned}$$

So the step count for RSum is 2n+2.

Method 2: Table method

- 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 contributed by each statement.
- This figure is often arrived at by first determining the number of steps per execution(s/e) of the statement and the total number of times (i.e., frequency) each statement is executed.
- The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.
- By combining these two quantities, the total contribution of each statement is obtained.
- By adding the contributions of all statements, the step count for the entire algorithm is obtained

- **Example 3: The step count of an algorithm can be determined by using the table method as follows. Constructing Frequency Table**

Statement	s/e	frequency	total steps
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			$2n+3$

Statement	s/e	frequency		total steps	
		$n=0$	$n>0$	$n=0$	$n>0$
1 **Algorithm** RSum(a, n)	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 RSum$(a, n-1) + a[n]$;	$1+x$	0	1	0	$1+x$
7 }	0	–	–	0	0
Total				2	$2+x$

$x = t_{\text{RSum}}(n-1)$

1.6 AsymptoticNotation(O,Ω, θ)

- We introduce some terminology that enables us to make meaningful statements about the time and space complexities of an algorithm.

Big Oh Notation:

- **Defnition:** The function f(n) = O(g(n)) (read as f of n is big O of g of n") iff (if and only if) there exist positive constants c and n0 such that f(n)<=c*g(n) for all n, n> n_0.
- The statement f(n) = O(g(n)) states only that g(n) is an uppe rbound on the value of f(n) for all n, n > =n_0.
- For the statement/(n) = O(g(n)) to be informative, g(n) should be as small a function of n as one can come up with for which f(n) = O(g(n)
  - **O(1)** to mean a **computing time that is a constant**.
  - **O(n)** is called **linear.**

- **$O(n^2)$**is called **quadratic.**
- **$O(n^3)$**) is called **cubic.**
- **$O(2^n)$** is called **exponential.**
- If an algorithm takes time **O (log n),** it is faster, for sufficiently larg en, than if it had taken **O(n).**
- Similarly, **O(n log n)**is better than **$O(n^2)$** but not as good as **O(n).**
- These are the seven computing times-**O(l), O(log n),O(n),O(nlog n), $O(n^2)$, $O(n^3)$**, **$O(2^n)$** .

➢ It should be clear that **f(n) = O(g(n)) is not the same as O(g(n)) = f(n).**

➢ Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C x g(n)** for all values of **C > 0** and **n_0>= 2**

f(n) <= C g(n)

□3n + 2 <= C n

Above condition is always TRUE for all values of **C = 4** and **n >= 2**.

By using Big - Oh notation we can represent the time complexity as follows...

3n + 2 = O(n)

Big Omega Notation:

➢ **Defnition:** The function f(n) = Ω(g(n)) (read as f of n is omega of g of n) iff there exist positive constants c and n0 such that f(n) >=c * g(n) for all n, n >=n0.

➢ The function g(n) is only a lower bound on f(n).

➢ For the statement f(n) = Ω(g(n)) to be informative, g(n) should be as large a function of n as possible for which the statement f(n) = Ω(g(n)) is true.

➢ Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n0>= 1**

f(n) >= C g(n)

□3n + 2 <= C n

Above condition is always TRUE for all values of **C = 1** and **n >= 1**.

By using Big - Omega notation we can represent the time complexity as follows...

3n + 2 = Ω(n)

Theta Notation:

- **Definition:** The function f(n) = θ(g(n)) (read as f of n is theta of g of n) iff there exist positive constants c1,c2,and n0 such that c1g(n)<=f(n)<=c2g(n)for all n, n>=n0.
- The theta notation is more precise than both the big oh and omega notations. The function f(n) = θ(g(n))) iff g(n) is both an upper and lower bound on f(n)
- Consider the following f(n) and g(n)...

  f(n) = 3n + 2

  g(n) = n

  If we want to represent f(n) as Θ(g(n)) then it must satisfy C1 g(n) <= f(n) >= C2 g(n) for all values of C1, C2 > 0 and n0>= 1

  C1 g(n) <= f(n) >= □C2 g(n)

  C1 n <= 3n + 2 >= C2 n

  Above condition is always TRUE for all values of C1 = 1, C2 = 4 and n >= 1.

  By using Big - Theta notation we can represent the time complexity as follows...

  3n + 2 = Θ(n)

Little Oh Notation:

- **Defnition:** Thefunction f(n) = o(g(n)) (read as f of n is little oh of g of n) iff

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Example The function $3n + 2 = o(n^2)$ since $\lim_{n\to\infty} \frac{3n+2}{n^2} = 0$. $3n + 2 = o(n \log n)$. $3n + 2 = o(n \log \log n)$. $6 * 2^n + n^2 = o(3^n)$. $6 * 2^n + n^2 = o(2^n \log n)$. $3n + 2 \neq o(n)$. $6 * 2^n + n^2 \neq o(2^n)$.

Little Omega Notation:

- Defnition: The function f(n) =w(g(n)) (read as f of n is little omega of g of n) iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

UNIT-I

Assignment-Cum-Tutorial Questions

SECTION-A

Objective Questions

1. For the following program fragment, the time complexity is [ ]

```
for (i=0; i<n; i++)
        a[i] = i;
```

A) O(n-1) B) O(n) C) $O(n^2)$ D) O(log n)

2. What is time complexity of fun()? [ ]

```
intfun(int n)
{
        int count = 0;
        for (inti = n; i> 0; i /= 2)
                for (int j = 0; j <i; j++)
                        count += 1;
         return count;
}
```

A) $O(n^3)$ B) O(n) C) $O(n^2)$ D) O(nlog n)

3. For the following program portion, the running time is [ ]

```
for (i=0; i< n; i++)
        for (j=i; j< n; j++)
                for (k=j; k< n; k++)
                        s++;
```

A) O(n) B) $O(n^2)$ C) $O(n^3)$ D) O(nlog n)

4. For the following program, the running time is [ ]

```
for (i=0; i< n*n; i++)
        a[i] = i;
```

A) O(n) B) $O(n^2)$ C) $O(n^3)$ D) O(nlog n)

5. What is the time complexity of the following algorithm [ ]

```
Algorithm Add(a, b, c, m. n)
{
        for i = 1 to m do
            for j = 1 to n do
                c[i,j] = a[i,j]+b[i,j];
}
```

A) 2mn+2n B) 2mn+2m C) 2mn+2m+1 D) 2mn+2n+1

6. Which of the given options provides the increasing order of asymptotic complexity of functions f1, f2, f3 and f4? [ ]

f1(n) = 2^n

f2(n) = n^(3/2)

f3(n) = nLogn

f4(n) = n^(Logn)

A) f3,f2,f4,f1 B) f3,f2,f1,f4 C) f2,f3,f1,f4 D) f2,f3,f4,f1

7. What does it mean when we say that an algorithm X is asymptotically more efficient than Y? [ ]

A) X will be a better choice for all inputs

B) X will be a better choice for all inputs except small inputs

C) X will be a better choice for all inputs except large inputs

D) Y will be a better choice for small inputs

8. What is the space complexity of the following algorithm [ ]

```
Algorithm FindFact(n)
{
        fact=1;
        for i =1 to n do
                fact=fact*i;
        return fact;
}
```

A) n B) n+3 C) 2n + 3 D) n+2

9. Which of the following are true? [ ]

a) $33n^3 + 4n^2 = O(n^3)$ b) $n! = O(n^n)$

c) $10n^2 + 9 = O(n^2)$ d) $6n^3 / (\log n + 1) = O(n^3)$

A) a, b and c B) a and c C) a and b D) all are true

10. The task of determining how much computing time and storage space that an Let $f(n) = n^2 \log n$ and $g(n) = n(\log n)^{10}$ be two positive functions of n. Which of the following statements is correct? [ ]

A) $f(n) = O(g(n))$ and $g(n) \neq O(f(n))$ B) $g(n) = O(f(n))$ and $f(n) \neq O(g(n))$

C) $f(n) \neq O(g(n))$ and $g(n) \neq O(f(n))$ D) $f(n) = O(g(n))$ and $g(n) = O(f(n))$

11. Which of the following is false? [ ]

A) $100n \log n = O(n \log n / 100)$ B) $\sqrt{\log n} = O(\log \log n)$

C) If $0<x<y$ then $n^x = O(n^y)$ D) $2n \neq O(nk)$

12. Consider the following functions

$f(n) = 3n^{\sqrt{n}}$ $g(n) = 2^{\sqrt{n} \log n}$ $h(n) = n!$

Which of the following is true? [ ]

A) h(n) is O(f(n)) B) h(n) is O(g(n))

C) g(n) is not O(f(n)) D) f(n) is O(g(n))

13. Consider the following segment of C code [ ]

```
int j, n;
j=1;
while(j<=n)
        j=j*2;
```

The number of comparisons made in the execution of the loop for any n>0 is

1. n^2 B) n C) logn D) logn +1

14. What is the time complexity of the following algorithm? [ ]

```
Algorithm FindFact(n)
{
        fact=1;
        for i =1 to n do
```

```
        fact=fact*i;
    return fact;
}
```

A) 2n+4 B) n C) 2n + 3 D) n+3

15. Given $f(n) = \log_2 n$, $g(n) = \sqrt{n}$ which function is asymptotically faster [ ]

A) f(n) is faster than g(n) B) g(n) is faster than f(n)

C) Either f(n) or g(n) D) Neither f(n) nor g(n)

16. Suppose $T(n) = 2T(n/2) + n$, $T(0) = T(1) = 1$.

Which of the following is FALSE [ ]

A) $T(n) = O(n^2)$ B) $T(n) = \Theta(n\log n)$ C) $T(n) = O(n\log n)$ D) $\Omega(n^2)$

17. Arrange the following functions in increasing asymptotic order [ ]

i. $n^{1/3}$ ii. e^n iii. $n^{7/4}$ iv. $n \log^9 n$ v) 1.0000001^n

A) i, iv, iii, v, ii B) v, iv, iii, i, ii C) i, ii, iv, iii, v D) i, iii, iv, ii, v

18. Consider the following three claims

1. $(n + k)^m = \Theta(n^m)$, where k and m are constants
2. $2^{n+1} = O(2^n)$
3. $2^{2n+1} = O(2^n)$

Which of these claims are correct? [ ]

A) 1 and 2 B) 1 and 3 C) 2 and 3 D) 1, 2, and 3

SECTION-B

SUBJECTIVE QUESTIONS

1. Define algorithm. Explain the characteristics of an algorithm.
2. Write an algorithm for matrix multiplication and compute its time complexity.
3. What is space complexity? With suitable example, explain how it is computed.
4. Explain asymptotic notations with suitable examples.
5. Write an algorithm for finding the maximum element in an array of elements and show its time complexity.

6. When the space complexity of an algorithm does becomes zero? Illustrate with an example.
7. Write recursive algorithm to find n^{th} Fibonacci number.
8. The factorial function *n!* has value 1when n < =1and value n * (n- 1)! when n > 1. Write both a recursive and an iterative algorithms to compute n!.
9. Calculate the time complexity for the following program segment:

```
Algorithm Add(a, b, c, m, n)
{
        for i= 1 to m do
                for j = 1 to n do
                        C[i,j]=a[i,j]+b[i,j];
}
```

10. Show that the following equalities are correct:

    i. $5n^2 - 6n = \Theta(n^2)$ ii. $n! = O(n^n)$

    iii. 2n22n + n logn = $\Theta(n^2 2n)$ iv. $\sum_{i=0}^{n} i^2 = O(n^3)$

11. Calculate the space complexity for the following piece of code:

```
intsum(int A[], int n)
{
        int sum = 0, i;
        for(i = 0; i< n; i++)
                sum = sum + A[i];
         return sum;
}
```

12. Calculate the time and space complexities for the following program segment:

```
i=1;
while(i<=n) do {
        x=x+1;
        i=i+1;    }
```

SECTION-C

QUESTIONS AT THE LEVEL OF GATE

1. The running time of the following algorithm is [ ]

Algorithm A(n) [GATE 2002]

```
{
    If n<=2 return 1
    else return A(√n)
}
```

A) O(n) B) O(logn) C) O(log logn) D) O(1)

2. Consider the following functions

$f(n) = 2^n$

$g(n) = n!$

$h(n) = n^{\log n}$

Which of the following statements about the asymptotic behavior of f(n), g(n), h(n) is true? [GATE 2008] [ ]

A) $f(n)=O(g(n))$; $g(n)=O(h(n))$ B) $f(n)=\Omega(g(n))$; $g(n)=O(h(n))$

C) $g(n)=O(f(n))$; $h(n)=O(f(n))$ D) $h(n)=O(f(n))$; $g(n)=\Omega(f(n))$

3. Let $f(n)=n$ and $g(n)=n^{(1+\sin n)}$, where n is a positive integer. Which of the following statements is/are correct? [GATE 2015] [ ]

1. $f(n)=O(g(n))$ II. $f(n)=\Omega(g(n))$

A) only I B) only II C) both I and II D) neither I and II

4. Consider the following function;

```
intfun(int n)
{   inti, j;
    for (i=1; i<=n; i++)
        for(j=1; j<n; j+=i)
            printf("%d %d", i,j);
}
```

The time complexity of fun in terms of Θ notation is [GATE 2017]

[ ]

A) $\Theta(n\sqrt{n})$ B) $\Theta(n^2)$ C) $\Theta(n\log n)$ D) $\Theta(n^2\log n)$

5. Consider the following functions from positive numbers to real numbers;

10, $\sqrt{n}$, n, logn, 100/n

The correct arrangement of the above functions in increasing order of asymptotic complexity is; [GATE 2017] [ ]

A) logn, 100/n, 10, $\sqrt{n}$, n B) 100/n, 10, logn, $\sqrt{n}$, n

C) 10, 100/n, $\sqrt{n}$,logn, n D) 100/n, logn, 10, $\sqrt{n}$, n