Course Code 18CSC204J
Course Name DESIGN AND ANALYSIS OF ALGORITHMS

Dr.S.PRASANNA DEVI

Professor & Head, DCSE

SRMIST, VDP

SYLLABUS

Duration (hour)		15	15	15	15	15
S-1	SLO-1	Introduction-Algorithm Design	Introduction-Divide and Conquer	Introduction-Greedy and Dynamic Programming	Introduction to backtracking - branch and bound	Introduction to randomization and approximation algorithm
	SLO-2	Fundamentals of Algorithms	Maximum Subarray Problem	Examples of problems that can be solved by using greedy and dynamic approach	N queen's problem - backtracking	Randomized hiring problem
S-2	SLO-1	Correctness of algorithm	Binary Search	Huffman coding using greedy approach	Sum of subsets using backtracking	Randomized quick sort
	SLO-2	Time complexity analysis	Complexity of binary search	Comparison of brute force and Huffman method of encoding	Complexity calculation of sum of subsets	Complexity analysis
S-3	SLO-1	Insertion sort-Line count, Operation count	Merge sort	Knapsack problem using greedy approach	Graph introduction	String matching algorithm
	SLO-2	Algorithm Design paradigms	Time complexity analysis	Complexity derivation of knapsack using greedy	Hamiltonian circuit - backtracking	Examples
S 4-5	SLO-1 SLO-2	Lab 1: Simple Algorithm-Insertion sort	Lab 4: Quicksort, Binary search	Lab 7: Huffman coding, knapsack and using greedy	Lab 10: N queen's problem	Lab 13: Randomized quick sort
S-6	SLO-1	Designing an algorithm	Quick sort and its Time complexity analysis	Tree traversals	Branch and bound - Knapsack problem	Rabin Karp algorithm for string matching
	SLO-2	And its analysis-Best, Worst and Average case	Best case, Worst case, Average case analysis	Minimum spanning tree - greedy Kruskal's algorithm - greedy	Example and complexity calculation. Differentiate with dynamic and greedy	Example discussion
S-7	SLO-1	Asymptotic notations Based on growth functions.	Strassen's Matrix multiplication and its recurrence relation	Minimum spanning tree - Prim's algorithm	Travelling salesman problem using branch and bound	Approximation algorithm
	SLO-2	O, O, Θ, ω, Ω	Time complexity analysis of Merge sort	Introduction to dynamic programming	Travelling salesman problem using branch and bound example	Vertex covering
S-8	SLO-1	Mathematical analysis	Largest sub-array sum	0/1 knapsack problem	Travelling salesman problem using branch and bound example	Introduction Complexity classes
	SLO-2	Induction, Recurrence relations	Time complexity analysis of Largest sub-array sum	Complexity calculation of knapsack problem	Time complexity calculation with an example	P type problems
S 9-10	SLO-1 SLO-2	Lab 2: Bubble Sort	Lab 5: Strassen Matrix multiplication	Lab 8: Various tree traversals, Krukshall's MST	Lab 11: Travelling salesman problem	Lab 14: String matching algorithms

S-11	SLO-1	Solution of recurrence relations	Master Theorem Proof	Matrix chain multiplication using dynamic programming	Graph algorithms	Introduction to NP type problems
	SLO-2	Substitution method	Master theorem examples	Complexity of matrix chain multiplication	Depth first search and Breadth first search	Hamiltonian cycle problem
S-12	SLO-1	Solution of recurrence relations	Finding Maximum and Minimum in an array	Longest common subsequence using dynamic programming	Shortest path introduction	NP complete problem introduction
	SLO-2	Recursion tree	Time complexity analysis-Examples	Explanation of LCS with an example	Floyd-Warshall Introduction	Satisfiability problem
S-13	SLO-1	Solution of recurrence relations	Algorithm for finding closest pair problem	Optimal binary search tree (OBST)using dynamic programming	Floyd-Warshall with sample graph	NP hard problems
	SLO-2	Examples	Convex Hull problem	Explanation of OBST with an example.	Floyd-Warshall complexity	Examples
S 14-15	SLO-1 SLO-2	Lab 3: Recurrence Type-Merge sort, Linear search	Lab 6: Finding Maximum and Minimum in an array, Convex Hull problem	Lab 9: Longest common subsequence	Lab 12: BFS and DFS implementation with array	Lab 15: Discussion over analyzing a real time problem

TEXT BOOKS

1. Thomas H Cormen, Charles E Leiserson, Ronald L Revest, Clifford Stein, Introduction to Algorithms, 3rd ed., The MIT Press Cambridge, 2014
2. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2006
3. Ellis Horowitz, Sartajsahni, Sanguthevar, Rajesekaran, Fundamentals of Computer Algorithms, Galgotia Publication, 2010
4. S. Sridhar, Design and Analysis of Algorithms, Oxford University Press, 2015

Unit I

Introduction-Algorithm Design

Define Algorithm.

- An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

Properties of Algorithm

- An algorithm takes zero or more **inputs.**
- An algorithm results in one or more **outputs**.
- All operations can be carried out in a **finite time**.
- An algorithm should be **efficient and flexible**.
- It should use less **memory space** as much as possible.
- An algorithm must terminate after a **finite number** of steps.
- Each step in the algorithm must be **easily understood** for someone reading it.
- An algorithm should be concise and compact to facilitate verification of their **correctness**.

Fundamentals of Algorithms

- Understand some rules for writing the algorithm.

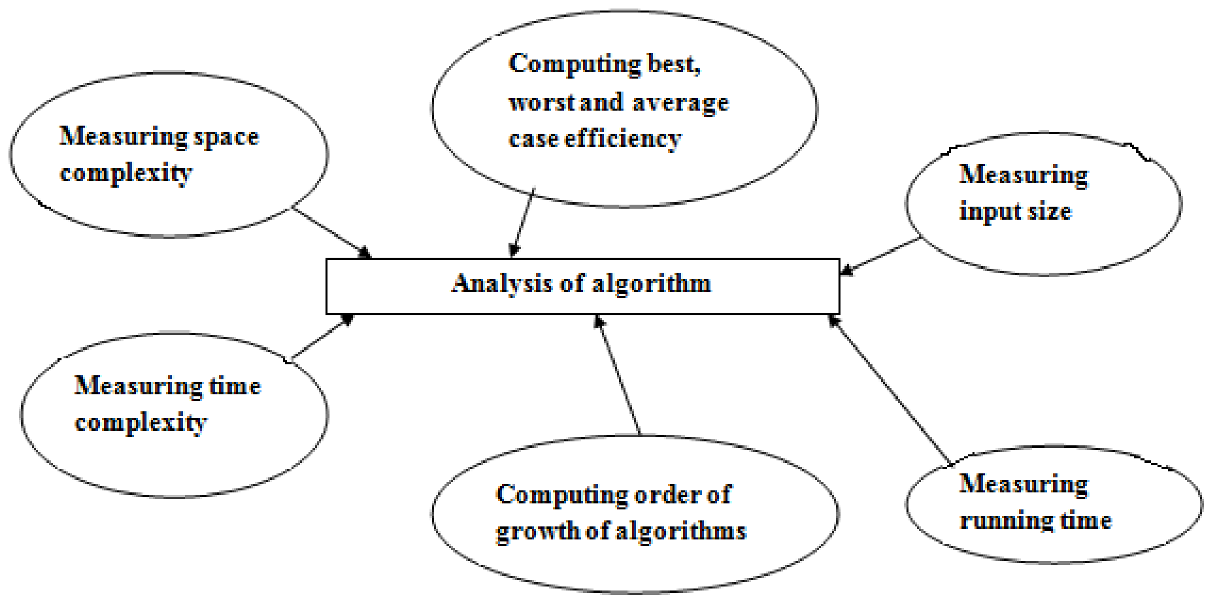
ALGORITHM numbertest(val)
// Problem description : This algorithm test for even / odd number
//Input: The number to be tested
//Output: Appropriate message indicating given number is even/odd
if val%2 = 0 then write("Given number is even")
else write("Given number is odd")

Write an algorithm to perform multiplication of two matrices
ALGORITHM Mul(A,B,n)
//Problem description : This algorithm is for computing multiplication of two matrices.
//Input: The two matrices A,B and order of them as n
//Output: The multiplication result will be in matrix C
for i ← 1 to n do
for j ← 1 to n do
C[i,j] ← 0
for k ← 1 to n do
*C[i,j] ← C[i,j] + A[i,k] * B[k,j]*

CORRECTNESS OF ALGORITHM
- FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

A general framework for analyzing the efficiency of algorithm is
- Measuring an input's size
- units for measuring running time
- orders of growth
- worst-case, best case and average case efficiencies

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		



Time & Space complexity analysis

Frequency count is a count denoting number of times of execution of statement.

Example:-

For(i=0; i<n; i++)
{
Sum = sum + a[i];
}

Statement	Frequency count
i = 0	1
i < n	➔ This statement executes for (n+1) times. When conditions is true (i.e; when i<n is true) ➔ This statement executes for n times. When conditions is false (i.e; when i<n is false)
i++	n times
Sum = sum + a[i]	n times
Total	**2n+1**

Algorithm Add (a, b, c)
*// **Problem description:** this algorithm computes the addition of 3 elements*
*//**Input:** a, b and c are of floating type*
*// **Output**: the addition is returned*
return a+b+c

The space requirement S(p) can be given as **S(p) = C +** $\mathbf{S_p}$

Time Complexity Analysis

Mathematical Analysis

Step 1 : The input size is simply order of matrices(n).

Step 2 : The basic operation in the inner most loop and which is

*C[i, j] ← C[i, j]+ A[i, k] * B[k, j]*

.We should note that in this basic operation both addition and multiplication are performed. But we will not choose any one of them as basic operation because on each repetition of innermost loop, each of the two will be executed exactly once. So by counting one automatically other will be counted. Hence we consider multiplication as a basic operation.

Step 3 : The basic operation depends only upon input size. There are no best case, worst case and average case efficiencies. Hence now we will go for computing sum. There is just one multiplication which is repeated on each execution of inner most loop. Hence we will compute the efficiency for inner most loops as.

Step 4 : The sum can be denoted by M(n).

M(n) = Outer Most Loop * Inner Loop * Inner Most Loop

Step 5 : Now we will simplicity **M(n)** as follows

Thus the simplified sum is n^3.

Thus the time complexity of matrix multiplication is

Matrix Multiplication Problem C=A.B:

Given two $n \times n$ matrices A and B, find the time efficiency of the definition-based algorithm for computing their product $C = AB$. By definition, C is an $n \times n$ matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:

where $C[i, j] = A[i, 0]B[0, j] + \dots + A[i, k]B[k, j] + \dots + A[i, n-1]B[n-1, j]$

for every pair of indices $0 \le i, j \le n - 1$.

ALGORITHM *MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])*

*// **Problem Description:** Multiplies two square matrices of order n by*

//the definition-based algorithm

*//**Input**: Two n × n matrices A and B*

*//**Output:** Matrix C = AB*

for** i ← 0 **to** n − 1 **do

for** j ← 0 **to** n − 1 **do

C[i, j] ← 0.0

for** k ← 0 **to** n − 1 **do

*C[i, j] ← C[i, j]+ A[i, k] * B[k, j]*

***return** C*

Insertion Sort
-Line count, Operation count

INSERTION-SORT(A[1..N])	cost	times
for j ← 2 to n	c_1	n
key ← A[j]	c_2	$n-1$
i ← j - 1	c_3	$n-1$
while i > 0 and A[i] > key	c_4	$\sum_{j=2,3,\ldots,n} t_j$
A[i+1] ← A[i]	c_5	$\sum_{j=2,3,\ldots,n} (t_j-1)$
i ← i - 1	c_6	$\sum_{j=2,3,\ldots,n} (t_j-1)$
A[i+1] ← key	c_7	$n-1$

Now let's calculate the running time as a function of n:

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 (n-1)$$

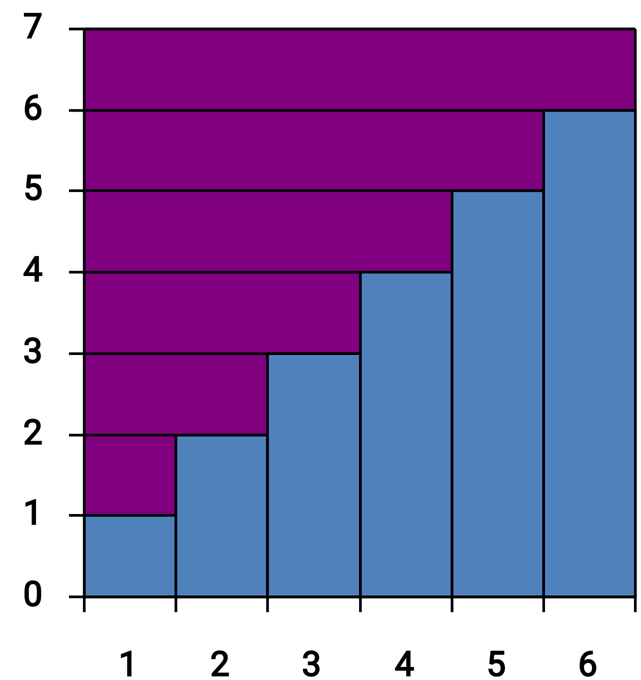
Question: So what is the running time?

Insertion Sort

Complexity may depend on the input!

- Best Case: Already sorted. t [redacted]

- Worst Case:
  - Neglecting the constants, the worst-base running time of ***insertion sort*** is proportional to $1 + 2 + \ldots + n$
  - The sum of the first n integers is $n(n+1)/2$
  - Thus, algorithm ***insertion sort*** required almost n^2 operations.

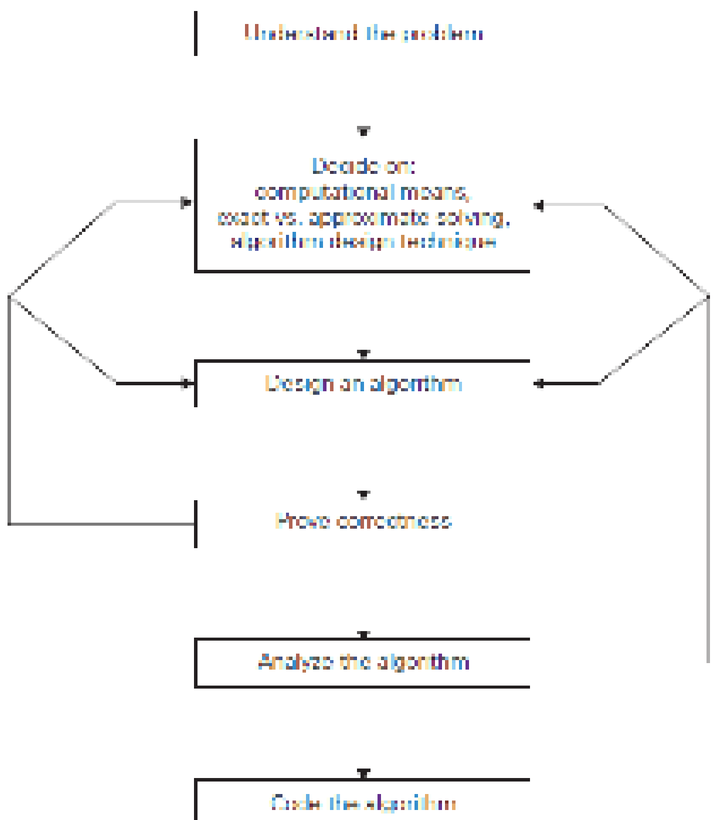


Insertion Sort – Average Case complexity

- Idea:
  - Assume that each of the n! permutations of A is equally likely.
  - Compute the average over all possible different inputs of length N.
- Difficult to compute!
- In this course we focus on Worst Case Analysis!

Designing an algorithm

The steps need to be followed while designing and analyzing an algorithm



The most important problem types:
- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

Analysis-Best, Worst and Average case

Efficiency of an algorithm is generally classified as 3 type

- Best case
- Average case
- Worst case

Best case: Best case is the shortest time that the algorithm will use over all instance of size n for a given problem to produce a desired result.

Average case : Average case is the average time that the algorithm will use over all instances of size n for a given problem to produce a desired result. It depends on the probability distribution of instances of the problem.

Worst case : Worst case is the longest time that an algorithm will use all instances of size n for a given problem to produce a desired result.

Asymptotic notations Based on growth functions.

The efficiency analysis framework concentrates on the order of growth of an algorithm’s basic operation count as the principal indicator of the algorithm’s efficiency. To compare and rank such orders of growth, computer scientists use three notations:

- O (big oh),
- Ω (big omega),
- Θ (big theta).

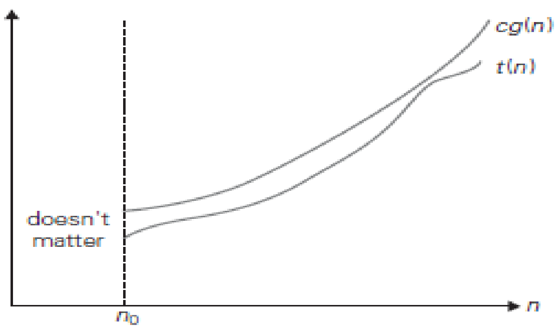
O, Ω ,Ө, o, ω

$t(n) \geq cg(n)$ for all $n \geq n$ c

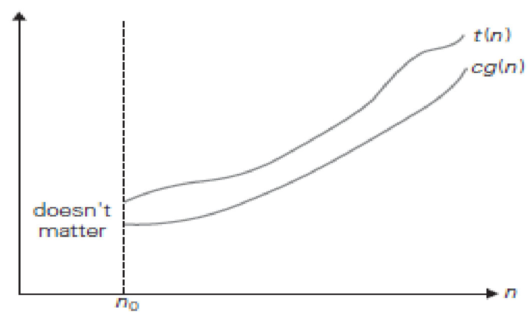
Here is an example of the formal proof that $n^3 \in \Omega(n^2)$:

$n^3 \geq n^2$ for all $n \geq 0$

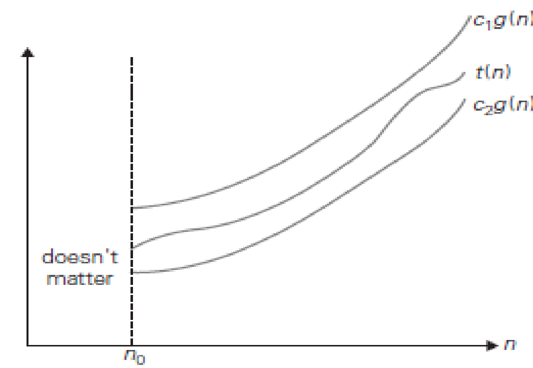
$t(n) \leq cg(n)$ for all $n \geq n$



Big-oh notation: $t(n) \in O(g(n))$.



Big-omega notation: $t(n) \in \Omega(g(n))$.



Big-theta notation: $t(n) \in \Theta(g(n))$.

o – notation

This notation is used to describe the worst case analysis of algorithms and concerned with small values of n

t(n) = o(g(n)) **iff** $\lim_{n \to \infty} \frac{g(n)}{t(n)} = 0$

ω - notation

This notation is used to describe the best case analysis of algorithm and concerned with small values of n.

t(n) = ω(g(n)) iff $\lim_{n \to \infty} \frac{g(n)}{t(n)} = 0$

Mathematical analysis

1. Let f(n) = 7n + 8 and g(n) = n. Is f(n) ∈ O(g(n))?

For 7n + 8 ∈ O(n), we have to find c and n0 such that 7n + 8 ≤ c · n, ∀n ≥ n0.

By inspection, it's clear that c must be larger than 7. Let c = 8.

Now we need a suitable n0. In this case, f(8) = 8 · g(8). Because the definition of O() requires that f(n) ≤ c · g(n), we can select n0 = 8, or any integer above 8 – they will all work.

We have identified values for the constants c and n0 such that 7n + 8 is ≤ c · n for every n ≥ n0, so we can say that 7n + 8 is O(n).

Definition	$\boxed{?}\ c > 0$	$\boxed{?}\ n_0 \geq 1$	$f(n)\ \boxed{?}\ c \cdot g(n)$
$O()$	$\exists$	$\exists$	$\leq$
$o()$	$\forall$	$\exists$	$<$
$\Omega()$	$\exists$	$\exists$	$\geq$
$\omega()$	$\forall$	$\exists$	$>$

2. Let f(n) = 7n + 8 and g(n) = n. Is f(n) ∈ o(g(n))?

In order for that to be true, for any c, we have to be able to find an n0 that makes f(n) < c · g(n) asymptotically true.

However, this doesn't seem likely to be true. Both 7n + 8 and n are linear, and o() defines loose upper bounds.

To show that it's not true, all we need is a counter–example. Because any c > 0 must work for the claim to be true, let's try to find a c that won't work.

Let c = 100. Can we find a positive n0 such that 7n + 8 < 100n? Sure; let n0 = 10. Try again! Let's try c = 1 100 . Can we find a positive n0 such that 7n + 8 < n 100 ? No; only negative values will work.

Therefore, 7n + 8 ∈/ o(n), meaning g(n) = n is not a loose upper-bound on 7n + 8.

3. Is 7n + 8 ∈ o(n^2)?

Solution of recurrence relations
Substitution method

Forward Substitution

This method makes use of an initial condition in the initial term and a value for the next term is generated. This process is continued until some formula is guessed.

Example: Consider recurrence relation T(n) = T(n-1) + n. initial condition: T(0) = 0.

let T(n) = T(n-1) + n

if **n = 1** then

T(1) = T(1-1) + 1 = T(0) + 1 = 0 + 1 = 1

if **n = 2** then

T(2) = T(2-1) + 2 = T(1) + 2 = 1 + 2 = 3

if **n = 3** then

T(3) = T(3-1) + 3 = T(2) + 3 = 3+ 3 = 6

by observation we can generate

[illegible]

Solution of recurrence relations

Backward Substitution

In this method backward value are substitute recursively in order to derive some formula.

Example: Consider recurrence relation T(n) = T(n-1) + n. initial condition: T(0) = 0.

let T(n) = T(n-1) + n ——————(1)

T(n-1) = T((n-1)-1) + (n-1)

T(n-1) = T(n-2) + (n-1) ——————(2)

put (2) in (1)

T(n) = [T(n-2) + (n-1)]+ n ——————(3)

T(n-2) = T((n-2)-1) + (n-2)

T(n-2) = T(n-3) + (n-1) ——————(4)

put (4) in (3)

T(n) = [[T(n-3) + (n-1)] + (n-1)]+ n

….

T(n) = T(n-k) + (n-k+1) + (n-k+2) +….+ n

if k=n then

T(n) = T(0) + 1 + 2 + …. + n

$(n(n+1))/2 = n\text{^}2/2 + n/2 \approx O(n\text{^}2)$

T(n) = 0 + 1 + 2 + …. + n =

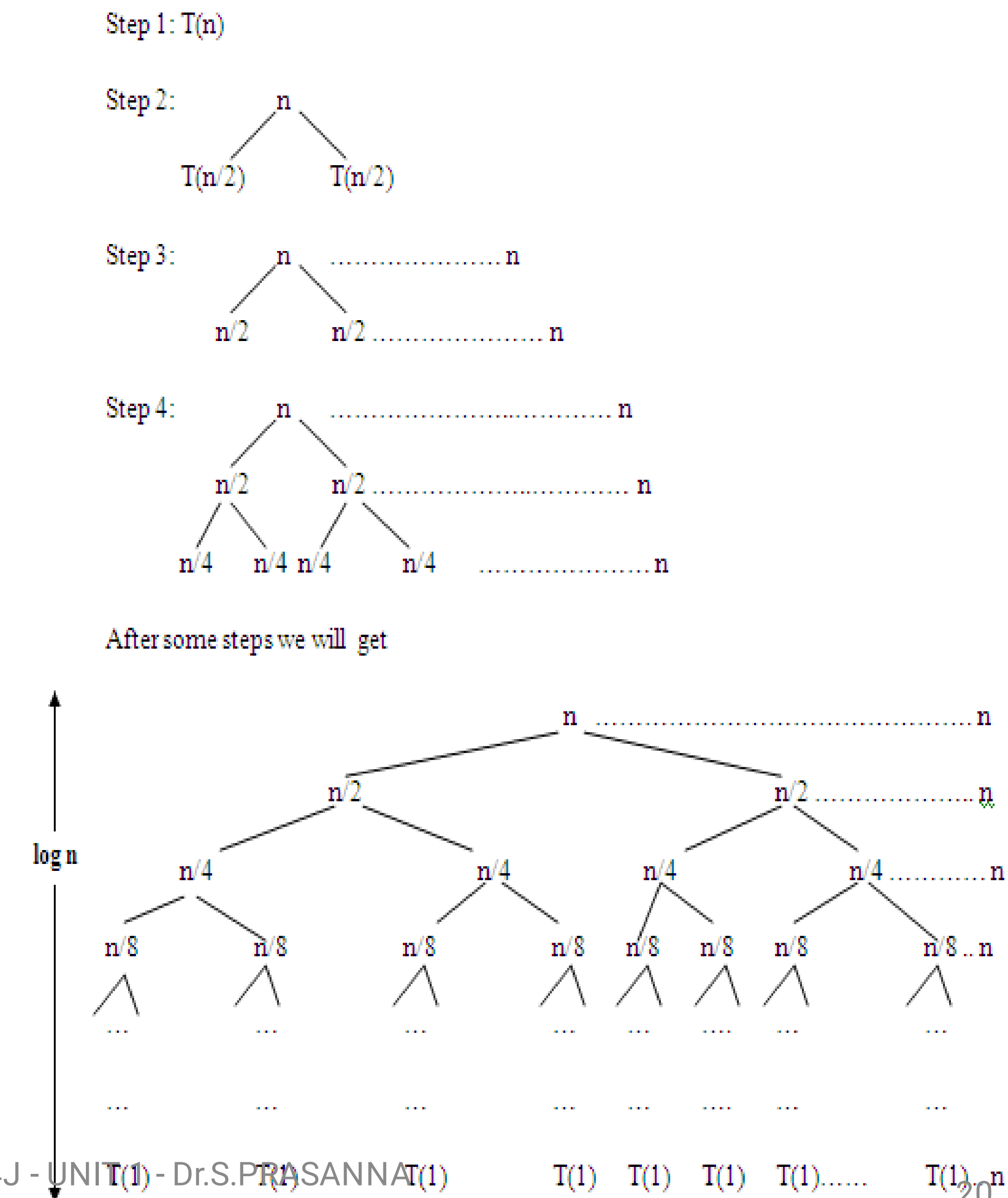
Solution of recurrence relations
Tree Method

Solve the given recurrence relation using recursion tree

$T(n) = 2T(n/2) + n$ with $T(1) = O(1)$

The depth of the tree is **log n.** Therefore total cost is **n log n T(1).**

The total overall cost as **<u>O(n log n)</u>**



Solution of recurrence relations

Mathematical Analysis:-

Step 1: The factorial algorithm works for **input size n**.

Step 2: The basic operation in computing factorial in **multiplication.**

Step 3: The recursive function call can be formulated as **F(n) = F(n-1) * n where n > 0**

Then the basic operation multiplication is given as M(n). and M(n) is multiplication count to compute factorial(n).

M(n) = M(n-1) + 1

These multiplications are required to compute factorial (n-1)

To multiply factorial (n-1) by n

Step 4: In step 3 the recurrence relation is obtained **M(n) = M(n-1) + 1.**

Step 5: Now we will solve recurrence using

Forward Substitution

M(1) = M(0) + 1 = 1

M(2) = M(1) + 1 = 1 + 1 =2

M(3) = M(2) + 1 = 2 + 1 = 3

Backward Substitution

M(n) = M(n-1) + 1

M(n) = [M(n- 2) + 1] + 1

M(n) = [M(n- 3) + 1] + 1 + 1

From the substitution methods we can establish a general formula as : **M(n) = M(n - i) + i**

Now let us prove correctness of this formula using mathematical induction as follows:

Prove M(n) = n by using mathematical induction

Basis: let n = 0 then M(n) = 0

i.e; M(0) = 0 = n

Induction : if we assume M(n-1) = n-1 then M(n) = n

M(n) = M(n-1) + 1

= n-1 + 1

= n

Thus the time complexity of factorial function is **Θ(n).**

- **Factorial Function Problem :**

Compute the factorial function *F(n) = n* ! for an arbitrary nonnegative integer *n.* Since

n ! = 1 . . . . . *(n − 1) . n* = *(n − 1)*!. *n* for *n* ≥ 1

and 0! = 1 by definition, we can compute *F(n) = F(n − 1) . n* with the following recursive algorithm.

ALGORITHM *F(n)*

*//**Problem Description:** Computes n! recursively*

*//**Input:** A nonnegative integer n*

*//**Output:** The value of n!*

if *n = 0*

return *1*

else

return *F(n − 1) * n*

Solution of recurrence relations

<u>Finding The Largest Element in The List Of N Numbers Problem:</u>
Consider the problem of finding the value of the largest element in a list of n numbers. For simplicity, we assume that the list is implemented as an array. The following is pseudocode of a standard algorithm for solving the problem.

```
ALGORITHM MaxElement(A[0..n − 1])
// Problem Description: Determines the value of the largest
element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ←A[0]
for i ←1 to n − 1 do
      if A[i]>maxval
            maxval←A[i]
return maxval
```

<u>Mathematical Analysis</u>

Step 1 : The input size is 'n'.

Step 2 : The basic operation is comparison in loop for finding large value.

Step 3 : The comparison is executed on each repetition of the loop. As the comparison is made for each value of n there is no need to find best case, worst case and average case analysis.

Step 4 : Let **C(n)** be the number of times the comparison is executed. The algorithm makes comparison each time the loop executes. That means with each new value of I the comparison is made. Therefore we can formulate C(n) as

C(n) = one comparison made for each value of i

Step 5 :

Thus the efficiency of algorithm is Θ(n).

Non – recursive algorithms do not involve recursive function.

<u>General Plan for Analyzing the Time Efficiency of Non - recursive Algorithms</u>

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation, either find a closed form formula for the count or, at the very least, establish its order of growth.

Unit I – Review Questions

1. What is an Algorithm?
2. Write the algorithm for GCD calculation?
3. What is algorithm design Technique?
4. Differentiate time and Space efficiency?
5. Design an algorithm to compute the area and Circumference of a circle
6. List the important problem types.
7. How will you measure input size of algorithms
8. Define best, worst and average case efficiency?
9. Define big oh(O),Big omega(Ω) and big theta(Θ) notations
10. List the basic efficiency classes
11. Define recurrence relation?
12. What is non recursion relation?
13. Define nonrecursive algorithm?
14. What does this algorithm compute? How many times is the basic operation executed?
15. Write an algorithm using recursive function to find the sum of n numbers.
16. List the factors which affects the running time of the algorithm.
17. What is meant by substitute methods?
18. Write the general plan for analyzing Time efficiency of recursive algorithm.

```
S=0
for i=1 to n do
 S=S+i
return i
```

Unit I – Review Questions (Contd..)

1. Discuss in detail about fundamentals of algorithmic problem solving?
2. Explain the necessary steps for analyzing the efficiency of recursive algorithms
3. Explain the general framework for analyzing the efficiency of algorithm.
4. Write the asymptotic notations used for best case ,average case and worst case analysis of algorithms and Write an algorithm for finding maximum element of an array perform best , worst and average case complexity with appropriate order notations.
5. Explain the method of solving recurrence equations with suitable example.
6. Explain the method of solving Non recursive equations with suitable examples .
7. i)Describe the basic efficiency classes in detail. ii) Write an algorithm for Fibonacci numbers generation and compute the following a) How many times is the basic operation executed b) What is the efficiency class of this algorithm.
8. Solve the following recurrence relations
   a) $x(n)=x(n-1) + 5$ for $n > 1$ $x(1)=0$
   b) $x(n)=3x(n-1)$ for $n > 1$ $x(1)=4$
   c) $x(n)=x(n-1)+n$ for $n > 0$ $x(0)=0$
   d) $x(n)=x(n/2)+n$ for $n > 1$ $x(1)=1$ (solve for $n=2k$)
   e) $x(n)=x(n/3)+1$ for $n >1$ $x(1)=1$ (solve for $n=3k$)
9. Consider the following recursion algorithm

```
Min1(A[0 -------n-1])
 If n=1
 return A[0]
 Else
temp = Min1(A[0…….n-2])
If temp <= A[n-1]
return temp
Else Return A[n-1]
```

a) What does this algorithm compute?

b) Setup a recurrence relation for the algorithms basic operation count and solve it.

Unit I – Review Questions

- Problems discussed in class for calculation of time/space complexity.
- Problems discussed in class for solving recurrence equations.
- Calculation of time/space complexity for renowned algorithms – Insertion sort, Bubble sort, Merge sort, Binary search.

Unit 1, CW –Problems Solve by recurrence

Solve by recurrence tree T(n)=T(n-1)+n

```
Void recursion test(int n)
{
if(n>0)
{
for( i=0; i<n; i++)
{
print(n);
}
recursion test (n-1);
}
```

2. Solve T(n)=T(n-1)+n by backward recursion.

3. Find complexity using recursion tree T(n)=T(n-1)+log n.
Solve using fwd recursion.

```
Void recursion (n)
{
if(n>0)
{
for( i=1; i<n; i*2)
{
print(i)
}
recursion (n-1)
}
}
```

4. Solve by recursion tree $T(n)=T(n-1)+n^2$.
5. Solve by recursion tree T(n)=T(n-2)+1.
6. Solve by recursion tree T(n)+T(n-100)+n.
7. Solve by recursion tree T(n)=2T(n-1)+1.
8. Solve by recursion tree $T(n)=T(n-1)+n^3$.

Unit 1, CW –Problems
Find Time Complexity

1. Algorithm swap (a ,b)
```
{
}
```
2. Algorithm sum (A ,X)
```
{
}
```
3. Algorithm sum (A ,B ,n)
```
{
}
```
4. Algorithm multiply (A ,B ,X)
```
{
}
```
5.
```
For ( i=1 ; i<n ; i=i*2 )
{
}
```
6.
```
For ( i=n ; i>=1 ; i=i/2 )
{
}
```
7.
```
For ( i=0 ; i*i<n ; i++ )
{
}
```
8.
```
For ( i=0 ; i<n ; i++ )
{
}
 For ( j=0 ; j<n ; j++ )
   {
   }
```
9
```
For ( i=1 ; i<n ; i=i*2 )
        {
P++;
}
For ( j=1 ; j<p ;j=j*2 )
{
}
```
10.
```
For ( i=0 ; i<n ; i++ )
{
For ( j=0 ; j<n ; j++ )
{
}
}
```
11.
```
i=0;
while (i<n)
{
stmt ;
i++;
}
```
12.
```
a=1;
while (a<b)
{
Stat;
a=a*2;
}
```

Unit 1, CW – Problems Asymptotic Notations

1. Given $f(n) = 2n+3$. Show $f(n)=o(g(n))$.
2. Compare functions :

   $f(n) = n \log n$

   $g(n) = n(\log n)$

Say true/false.

1. $(n+k)= O(n)$.
2. $2 = O(2)$.
3. $n = O(2)$.
4. $5n-6n = O(n)$.
5. $n! = O(n)$.
6. 2n [redacted]
7. $33n+4n = \Omega(n)$.
8. $f(n)=2n+3=O(n)$.
9. $f(n)=2n+3=O(n)$.
10. $f(n)=\Omega(g(n))$.
11. $f(n)=\Theta(g(n))$.
12. $10n+15n+100n2=O(100n2)$.
13. $n+n \log n = \Theta(n)$.

Thank You!