→ PCC [Theory and Lab Separately] Date: / /

Analysis Design and Algorithm [ADA]

* Course Objectives:

1) To learn the methods for analysing algorithms and evaluating their performance.
2) To determine the efficiency of algorithms using Asymptotic notations.
3) To solve problems using various algorithm design methods includes brute force, Greedy method, divide and conqure, Dynamic programing, Backtracking, Branch and Bound.
4) To learn the concepts of P and NP complexity classes.

* Course Outcomes: [CO]

1) Apply Asymptotic notational method to analyse the performance of algorithm in terms of time complexity and space complexity.
2) Demonstrate divide and conqure approaches to solve computational problems.
3) Make use of transform and conqure dynamic programming design approaches to solve the given real world or complex computational problems.
4) Apply greedy method and input enhancement methods to solve graph and string based computational problems.
5) Analysis of various classes: P, NP and NP complete of problems.
6) Illustrate backtracking, branch and bound methods.

→ (After 4 years of engineering)

PEO's → Program Educational Outcomes

PO → Program Outcome

↳ (During the 4 years of engineering)

* Applications of ADA

1) Software Development
2) Data Science and ~~Visual~~ Machine learning
3) Telecommunication
4) Gaming

* Algorithm:

→ Algorithm is a step by ~~step~~ step process to solve a particular task.

problem to be solved
↓
algorithm created for performing particular task
↓
I/P → computer → O/P → correct / error

- **Properties:**

1) Non ambiguity
2) Range of input
3) Multiplicity → can be written in multiple place & multiple language
4) Speed
5) Finiteness → should be finite

- **Issues:**

→ How to design, validate, analyse, device an algorithm

* How to write an algorithm:

(Q) Write an algorithm to find the sum of n-numbers

Algorithm:
Sum (1, n)
// problem description :- Sum of 'n' numbers
// input : 1 to n numbers
// output : the sum of numbers
result ← 0
for i ← 1 to n do i ← i+1
result ← result + i
~~result~~ return result;

(Q) Write an algorithm ~~for~~ to find factorial for n numbers.
Fact ← 1
for i ← 1 to n do i ← i+1
Fact ← Fact * i
return fact;

* How to write an algorithm:

1) Header → Algorithm name of algorithm (parameter list)
2) Body → "main" algorithm

- **Steps:**

S i) Algorithm heading
↳ name, input, output & problem description
S ii) Algorithm body
↳ various programming constrains along with the assignment statements
↳ The heading section we should write following steps.
// problem description
// input
// output
S iii) The body of algorithm is written in various programming constrains like if, while, for or some assign statements may be written.

papergrid
Date: / /

S iv) The compound statements should be enclosed with open and closed brackets ({ })
S v) Single line comments written using double slash (//).
S vi) Identifiers should begin by a letter not by a digit. and it is a combination of alphabet strings.
S vii) Using assignment operator ← an assignment statement (← this arrow symbol is used)
variable ← expression. Eg:- a ← b. (a = b)
S viii) There are other types of operators such as boolean operator (true or false), logical operators (&&, ||, !), relational operators (<, >, >=, <=).
S ix) The array index starts from 0 to n-1 and indicates square brackets([ ])
S x) The input and output can be done using read and write.
Eg:- write ("Hi!")
read (value)
S xi) The conditional statements such as 1) if-then, 2) if-then else, (3) if (condition) then statement, (4) if (condition) then statement else statement.

if, while, for

syntax → while (condition)
{
Block of statements
}

Syntax → for (___ ; ___ ; ___)
for variable ← value 1 to value do
{
statement 1;
:
statement n;
}

S xii) Repeat until statement
syntax: return
statement 1
statement n
until (condition)

papergrid
Date: / /

Break statement → Terminates the whole loop
Continue statement → Terminates a particular loop or skips a condition.

(Q) Write an algorithm for sorting of an element.

Algorithm Sort (n, a) (Sort → name; (n, a) → parameters)
// problem description: To sort array of given elements
// input: n value to the list of elements
// output: the sorted list.

```
For i ← 1 to n do
For j ← i+1 to n-1 do
{
  if (a[i] > a[j]) then
  {
   temp ← a[i]
   a[i] ← a[j]
   a[j] ← temp
  }
}
```

(Q) Write a algorithm for GCD of elements (GCD → greatest common diviser)

GCD can be found in 3 ways → Euclids algorithm; → consecutive integer checking; → Finding the repetative factors

Euclids algorithm
gcd (a, b) ⇒ gcd (b, a mod b)
Eg:- (30, 18).

a	b	Remainder
30	18	12
18	12	6
12	6	0
(6)	0	

when b encounters the value 0 then a value will be the GCD
Here GCD is 6.

Algorithm GCD (a, b)
// Problem description : GCD of elements
// input : a value, b value
// output: the GCD value.

```
while (b != 0)
{
  c ← a mod b
  a ← b
  b ← c
}
return a.
```

* Fundamentals of algorithm solving :

Steps to be followed while designing and analysing an algorithm.

S1) ~~Understand~~ Understand the problem (Analyse)
S2) Decision making on
  i) capabilities of computational devices
  ii) choice of either exact or approximate solving methods
  iii) Data Structures
  iv) Algorithmic Strategies
S3) Specification of an algorithm
S4) Algorithm verification
S5) Analysis of an algorithm
S6) ~~Implementation~~ Implementation of coding of an algorithm

(•*) Algorithmic ~~Algorithmic~~ Strategies

1) ~~Brute~~ Brute force → Straight forward technique & navie approach
2) Divide and Conquere → divide the problems & combine the solutions (smaller approach) (problem is divided into smaller approaches)
3) Decrease and conquere → The instance size is decreased, then we combine the solution to solve the problem
4) Tranform and conquere → The instance is modified & then solved

(5) Dynamic programming → The result of smaller recursive instances are obtained to solve the problem
(6) Greedy method → To solve the problem locally, optimal decisions are made.

(Units)

(•*) Specification of Algorithm [3 ways]

1) Using natural language → easy & understandable
2) Pseudo code → natural language + programming language
3) Flowchart.

Examples :-

1) Using Natural language	2) Pseudo code
Read a variable.	step1: start step2: Read a variable
Read b variable.	3: Read b variable
add c = a+b	4: add c ← a+b
display c	5: stop.

3) Flow chart

- → start/stop
- → Processing
- → i/p & o/p.
- → loop
- → decision/condition
- → connectors
- (→ ↓) flow of program.

(•*) Verification of Algorithm :-

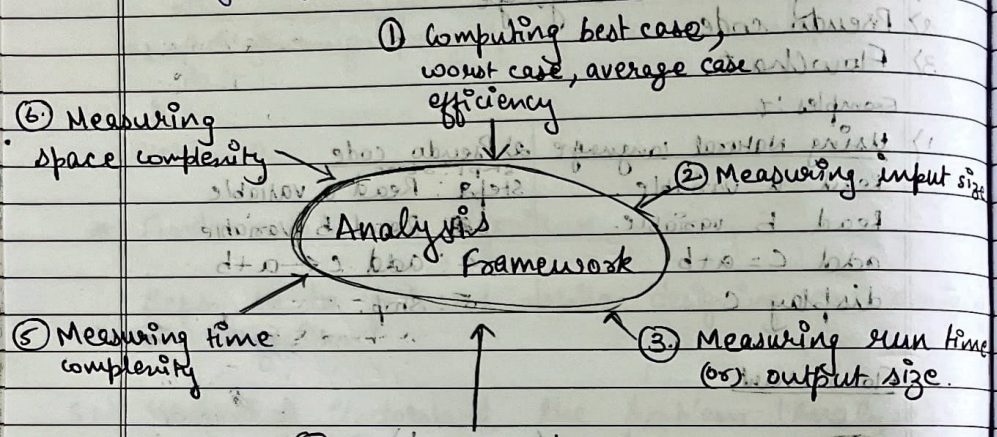
Checking of correctness of input & output.

(•*) Analysis of algorithm :

1) Time efficiency of an algorithm → Also known as Time complexity
The total amount of time to execute an algorithm is called Time Complexity (slow or fast)
2) Space efficiency of an algorithm → space complexity
The total amount of space which is required for an algorithm is called space complexity (less / more).

3) Simplicity → The sequence of instructions which are easy to understand.
4) Generality → General form which everyone are familier with
5) Range of inputs

* Analysis frame work :-



• Space Complexity

→ Space Complexity : It can be defined as amount of memory required by an algorithm to run.

→ Space Complexity can be calculated by
$S(P) = C + SP$
(S: space complexity, C: constant [constant path], SP: instance characteristics also called as variable paths)

[Constant path also contain 'input & output']

Example : Sum of 2 no's

```
alg Sum(p, q, r)
{
  p=1   → 1 unit / time / block.
  q=1
  r=p+q
}
```
(p, q & r are constants) Each constant is considered as 1 unit
p = 1U
q = 1U
r = 1U

$S(P) = C + SP$
$= 1+1+1 = 3$
$S(P) = 3.$
$= O(1)$

For all constants we have to give O(1). (Notation)

Example : Sum of n numbers.

```
Sum(s, n)
{
  total = 0;
  for i = 0 to n do
    total = total + s[i];
}
```
constant paths → total, n, i — 1U, 1U, 1U
variable paths → s → dependent variable
dependent on i
→ i to n
→ (n times)

$S(P) = C + SP$
$= 1+1+1+n$
$= n+3$ → constant is eliminated
$= O(n)$

→ When there is no variable path or SP then the notation is big O of 1 → O(1)
Here each constant is considered as 1unit

→ When there is a variable path or SP then the constant path is calculated as 1unit = only but it is eliminated during calculation and the notation becomes big O of n → O(n)
Here n represents variable path
[n can differ in each program]

Example: Algorithm of matrix addition.

```
Alg add (A, B, n)
{
for(i=0; i<n; i++)
{
 for(j=0; j<n; j++)
 {
  C[i,j] = A[i,j] + B[i,j]
 }
}
}
```

constant	variable
$i \leftarrow i$	$A \rightarrow n^2$
$j \leftarrow j$	$B \rightarrow n^2$
$n \leftarrow n$	$C \rightarrow n^2$

$C[i,j]$ → $n \times n$; $A[i,j]$ → $n \times n$; $B[i,j]$ → $n \times n$

[i loop for n times
j loop for n times]

$S(p) = C + SP$
$= 1 + 1 + 1 + n^2 + n^2 + n^2$
$= 3 + 3n^2$
$= 3n^2 + 3$ → constant is eliminated
$= O(n^2)$

- Time Complexity

→ Time complexity: The time complexity of an algorithm is an amount of computer time required by an algorithm to run for completion.

→ It is difficult to calculate time complexity in terms of physical clock, it may vary system load, number of other programs running instruction set, speed of hardware part.

$T(n) = Cop \times C(n)$ (× : multiply)

- $T(n)$ → Time complexity of n
- Cop → Time taken to perform basic operations [C of P]
- $C(n)$ → No of times the operations need to be executed.

This formula is used only for Nested loops. For normal loops we can directly get the time complexity

→ Declaration
→ Initialization
→ Iteration → normal (single) / nested loops → dependent / independent
→ Conditional statements → if there are 2 blocks lets say if & else blocks we have to only consider it as 1 block because out of the 2 only 1 will be executed the time maximum of the 2 blocks is considered.
[if: if loop is the maximum if loop is taken, if: else loop is the maximum else loop is taken].

Ex:

①
```
for(i=0; i<=n; i++)
{
}
```
will be executed n times

→ Time complexity = $O(n)$

②
```
for(i=1; i<=n; i=i*2)
{
}
```
$i=1, i=2, 2 \le n$
2^1 $i=2, i=4, 4 \le n$
2^2 $i=4, i=8, 8 \le n$
2^3 $i=8, i=16, 16 \le n$
$2^k \le n$
$k = \log_2 n$

Time complexity = $O(\log n)$
Base will become constant

③
```
for(i=0; i<=m; i++) → m
{
 for(j=0; j<=n; j++) → n
 {
 }
}
```
Time complexity = $O(m \times n)$

papergrid
Date: / /

④
```
If (condition)
{
  for (i=0; i<n; i++)        → n times
  {
   ——
  }
}
else
{
  for (j=0; j<=n; j*2)       log n times
  {
   ==
  }
}
```
n is the highest

Time complexity = O(n)

⑤
```
void main()
{
  int i;  — 1 unit
  i=0;    — 1 unit
  for (i=0; i<=n; i++)       → n times
  {
   printf("Welcome");
  }
}
```
1+1+n

2+n

n

T(n) = O(n)

Time Complexity = O(n)

⑥
```
Alg Sum(a, n)
{
  sum = 0;  → 1
  for (i=1; i<=n; i++) → n
  sum = sum + a[i] →
  return sum
}
```
1+1+n

=> 2+n

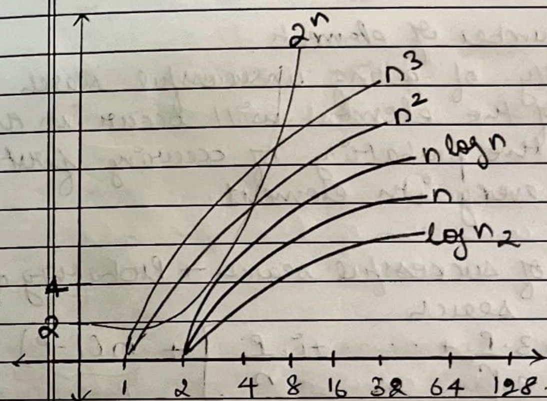
T(n) = O(n)

Time Complexity = O(n)

papergrid
Date: / /

* Order of Growth

n	log n	n log n	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,29



$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

* Best, worst and average case analysis

Linear search algorithm

```
For (i=0; i<n; i++)
{
  If (key == a[i])
  return i
}
```

key value = 10

(10) 20 30 40 50

a[i] key

best case

→ When we find the desired element at the first place then it is known as best case
$C_{best} = 1$

→ When searching an element if the desired element is found at the end it is known as worst case
$C_{worst} = n$

→ Let P is the probability of getting success or successful search
n is the total number of elements.
1-P is the probability of getting unsuccessful search
The first match of the element will occur in the i^{th} location. Hence the probability of occuring first match is P/n for every i^{th} element

C_{avg} = Probability of successful search + Probability of unsuccessful search

$$C_{avg} = \underbrace{\left[\frac{1\cdot P}{n} + \frac{2\cdot P}{n} + \frac{3\cdot P}{n} + \cdots + \frac{i\cdot P}{n}\right]}_{\text{Probability of successful search}} + \underbrace{n(1-P)}_{\text{Probability of unsuccessful search}}$$

$$C_{avg} = \frac{P}{n}[1+2+\cdots+i+\cdots+n] + n(1-P)$$

$$= \frac{P}{n}\left[\frac{n(n+1)}{2}\right] + n(1-P)$$

$$C_{avg} = \frac{P(n+1)}{2} + n(1-P)$$

If P=0

$$C_{avg} = \frac{0(n+1)}{2} + n(1-0)$$
$$= 0 + n$$
$$= n$$

If P=1

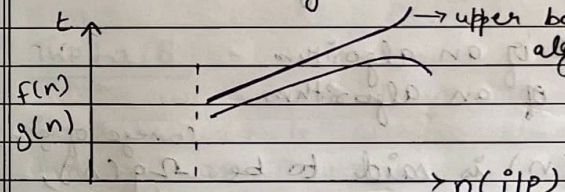
$$C_{avg} = \frac{1(n+1)}{2} + n(1-1)$$
$$= \frac{n+1}{2} + 0$$
$$= \frac{n+1}{2}$$

* Asymptotic notation:

→ These notations are used to check the efficiency of an algorithm
→ It is used to measure the time complexity of an algorithm
→ By using this we can find fastest possible, slowest possible or average time.
→ The notations are
O, Ω, θ, o, ω
O → big oh
Ω → bmega
θ → theta
o → small oh
ω → small omega
} mostly used

- Big oh (O) → worst case time complexity of algorithm.

→ It is used to represent the upper bound of algorithm run time. (longest time)



- f(n) & g(n) are two non-negative functions
- C is the constant and C should be greater than 0. (C > 0)
- n is greater than n0 (n > n0)
- f(n) ≤ g(n) if g(n) is multiply of some constant C
  $f(n) \le C * g(n)$
  $f(n) \in Og(n)$

$f(n) = 2n+2$ $\quad c>0$
$g(n) = n^2$ $\quad c=1$

$n=1$
$f(n) = 2(1)+2 = 4$ $\quad f(n) \leq c \cdot g(n)$
$g(n) = 1^2 = 1$ $\quad 4 \leq 1 \quad \times$

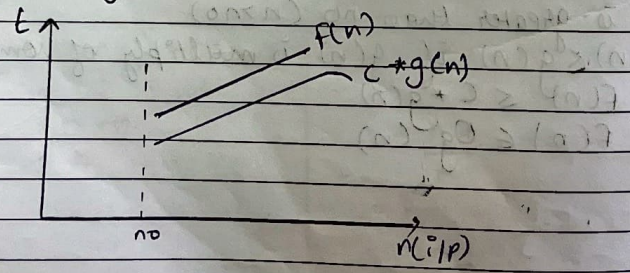
$n=2$
$f(n) = 2(2)+2 = 6$ $\quad 6 \leq 4 \times$
$g(n) = 2^2 = 4$

$n=3$
$f(n) = 2(3)+2 = 8$ $\quad 8 \leq 9 \checkmark$
$g(n) = 3^2 = 9$
$\therefore n \geq 2$
It satisfies the condition of $f(n) \leq c \cdot g(n)$

- **Omega (Ω)**

→ Represents the simple omega
→ shortest case
→ lowest bound of an algorithm
→ best case TC of an algorithm.

→ A function $f(n)$ is said to be $\Omega g(n)$ (omega of $g(n)$), if $f(n)$ is bounded below by some positive constants multiply of $g(n)$ such that
$f(n) \geq c * g(n)$
$f(n) \in \Omega g(n)$



Eg: $f(n) = 2n^2+2$ $\quad c>0$
$g(n) = 7n$

If $n=1$ $\quad f(n) \geq c*g(n)$
$f(n) = 2(1)^2+2 = 4 \quad 4 \geq 7 \times$
$g(n) = 7(1) = 7$

$n=2$
$f(n) = 2(2)^2+2 = 10 \quad 10 \geq 14 \times$
$g(n) = 7(2) = 14$

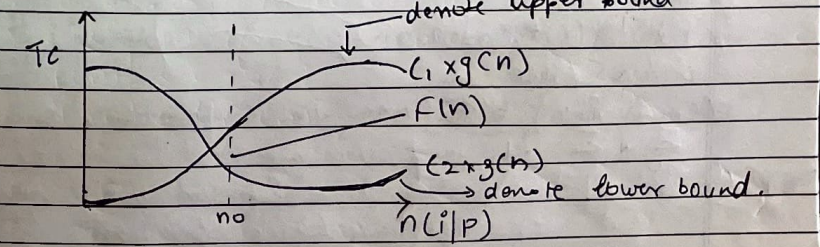
$n=3$
$f(n) = 2(3)^2+2 = 20 \quad 20 \geq 21 \times$
$g(n) = 7(3) = 21$

$n=4$
$f(n) = 2(4)^2+2 = 34 \quad 34 \geq 28 \checkmark$
$g(n) = 7(4) = 28$
$\therefore n \geq 3$
It satisfies the condition $f(n) \geq c*g(n)$.

- **Theta (Θ)**

→ Used to denote simple theta
→ Average case Time complexity of an algorithm

consider
$c_2 g(n) \leq f(n) \leq c_1 g(n)$
$f(n) \in \Theta(g(n))$



papergrid
Date: / /

Eg $f(n) = 2n+8$ $\quad c>0$

$c_1 g(n_0) = 5n$

$c_2 g(n) = 7n$

$c_2 g(n) \leq f(n) \leq c_1 g(n)$

If $n=1$

$f(n) = 2(1)+8 = 10$ $\quad 7 \leq 10 \leq 5$

$c_1 g(n_0) = 5n = 5$

$c_2 g(n) = 7n = 7$

$n=2$

$f(n) = 2(2)+8 = 12$ $\quad 14 \leq 12 \leq 10$ ×

$c_1 g(n) = 5n = 5\times 2 = 10$

$c_2 g(n) = 7n = 7\times 2 = 14$

$n=3$

$f(n) = 2(3)+8 = 14$ $\quad 21 \leq 14 \leq 15$ ×

$c_1 g(n) = 5n = 15$

$c_2 g(n) = 7n = 21$

$n=4$

$f(n) = 2(4)+8 = 16$

$c_1 g(n) = 5n = 20$

$c_2 g(n) = 7n = 28$

$n=5$

$f(n) = 2(5)+8 = 18$

$c_1 g(n) = 5n = 25$

$c_2 g(n) = 7n = 35$

$n=6$

$f(n) = 2(6)+8 = 20$

$c_1 g(n) = 5(6) = 30$

$c_2 g(n) = 7(6) = 42$

big oh → it is less than equal to
$f(n) \leq c \cdot g(n)$

small oh → it is only less than
$f(n) < c \cdot g(n)$

papergrid
Date: / /

- **o (small oh)**

→ It is same as big O.
that is $f(n) < o g(n)$

- **ω (small/little omega)**

→ similar to omega
$f(n) > \omega g(n)$

* **Mathematical analysis of recursive and non-recursive algorithm**

- **Mathematical analysis of non-recursive algorithm.**

- **General conditions / plan**

1) Decide the input size based on the n value
2) Identify the basic operations
3) Check how many times basic operation is executed, determine the best, average, worst case.
4) Setup sum for the number of ~~type~~ times basic operation is executed
5) Simplify the sum using standard formula

Formulas

1) $\sum_{i=1}^{n} 1 = 1+1+1+\cdots+1 = n \in \Theta(n)$

2) $\sum_{i=1}^{n} i = 1+2+3+\cdots+n = \frac{n(n+1)}{2} \in \Theta(n^2)$

3) $\sum_{i=1}^{n} i^K = 1+2^K+3^K+\cdots+n^K = \frac{n^{K+1}}{K+1} \in \Theta(n^{K+1})$

4) $\sum_{i=1}^{n} a^i = 1+a+\cdots+a^n = \frac{a^{n+1}-1}{a-1} \in \Theta(a^n)$

5) $\sum_{i=1}^{n} (a_i \pm b_i) = \sum_{i=1}^{n} a_i \pm \sum_{i=1}^{n} b_i$