Module 1

Introduction to Analysis of Algorithms

ALGORITHM:- is a finite set of instructions, if followed properly accomplishes a particular task.

All algorithm must satisfy following criteria:-

1) **Input** → zero or more quantities are externally supplied.
2) **Output** → Atleast one quantity is produced.
3) **Definiteness** → Each instruction must be clear & unambiguous.
4) **Finiteness** → If instructions of the algorithm are traced, then for all cases, algorithm must terminate after finite number of steps.
5) **Effectiveness** → (i) Every instruction must be basic i.e can be carried out by a person using only pen & paper.
   (ii) Every operation must be feasible.

Four distinct areas of study in the field of algo:

1. **How to devise algorithms ?**

- Creating an algorithm is an art which may never be fully automated. (art - automated)
- There are various design techniques that have proven to be useful as they have often yielded good algorithms.
- By mastering, these design strategies, it will become easier to devise new and useful algorithms.

2. How to validate algorithms ?

- Once an algorithm is devised, it is necessary to show that it computes the correct answer for all legal inputs. The process is referred to as "Algorithm Validation".
- The purpose of validation is to assure that the algorithm will work correctly independent of the programming issues.
- The second phase is known as "Program proving or program verification".
- There are two forms : for the proof of correctness of soln:

(i) A program which is annoted by set of assertion about the input and output variables of the program. (Expressed in predicate calculus).

(ii) Specification : It is also expressed in predicate calculus.

- A proof of program correctness requires that each statement of programming language be precisely defined and all basic operations be proved correct.

3. How to analyze algorithms ?

- also known as "Performance Analysis".
- It refers to the task of determining how much computing time and storage an algorithm requires.
- In this area, mathematical skills are required.
- The analysis allows to make quantative judgement about the value of one algorithm over another.
- Three cases for performance analysis : 1) Best case,
  2) Worst case and
  3) Average case.

4. How to test a program ?

There are 2 phases in this area :

(i) Debugging : Process of executing programs on sample data sets to determine whether faulty results occur & if so, correct them.

(ii) Profiling / Performance Measurement : The process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

Performance Analysis

- Can be done in 2 ways
  (i) Space Complexity
  (ii) Time Complexity.

⇒ Space Complexity

- Space Complexity of an algorithm is the amount of memory it needs to run to its completion.

It is calculated as,

$$S(P) = C + S_p$$

where,

P → algorithm to be analyzed
C → Fixed / constant part [independent of size of i/p or o/p's]
S_p → Instance Characteristics i.e space needed by component variables.
- Size is dependent on problem instance.

Examples :

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

In this, variable parameters are absent.

$\therefore S_p = 0$

Variable a, b, c will require 3 memory space or register

$\therefore \quad C = 3$

$\therefore \quad S(p) = C + S_p$

$= 3 + 0$

$\boxed{S(p) = 3}$

2. Algorithm Sum (A, n)

{

$S = 0.0;$

for $i = 1$ to n do

{

$S = S + A[i];$

}

return S;

}

In this, size of array is variable.

$\therefore \quad S_p = n$

Three memory spaces is required to store value of n, i, S.

$\therefore \quad C = 3$

$\therefore \quad S(p) = C + S_p$

$S(p) = n + 3$

$\boxed{S_{Sum}(n) \geq n+3}$

3. Algorithm RSum (A, n)

```
{
    if (n<=0) then
    {
        return 0;
    }
    else
    {
        return RSum(A, n-1) + A[n];
    }
}
```

- Recursion stack space includes
  - Formal Parameters
  - Local Variables
  - Return Addresses.

- In this, each stack space includes,
  (i) Pointer to A[]
  (ii) Value of n
  (iii) return address.

∴ Space requirement for each call = 3

⇒ Total No. of calls = Depth of recursion = n+1

∴ Total space requirement = 3(n+1)

∴ $S(P) \geq 3(n+1)$

⇒ Time Complexity

- Time complexity is sum of compile & execution (run) time taken by a program.

- Compile time - Independent of instance characteristics, hence it is constant.

- Run time - It is expression of time complexity.
  - denoted by tp.
  - Calculated as,

$$tp(n) = C_a ADD(n) + C_s SUB(n) + C_m MUL(n) + C_d DIV(n) + \ldots$$

where,

n - Instance characteristics.

C_a - time require for addition

C_s - time require for subtraction

C_m - time require for multiplication

C_d - time require for division.

ADD, SUB, MUL, DIV - functions whose value are no. of operation performed.

- Time require for all these operations depends on different parameters. So, we require a unique way to obtain time complexity.

- There are different ways for calculating time complexity, one is to count program steps.

- Program step is loosely defined as syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.

⇒ 2 ways to determine time complexity

[I] By using one temporary variable "count" :

- In this, the variable count is incremented by one after execution of each statement.
- After execution of entire program, the value of count is determined.

Examples :

1.
```
Algorithm Sum (A, n)
{
    S = 0;
    for i = 1 to n do
    {
        S = S + A[i];
    }
    return S;
}
```

By using the method,

```
Algorithm Sum (A, n)
{
    s = 0;
    count = count + 1;  // count is declared global
                        // & initialized to zero.
    for i = 1 to n do
    {
        count = count + 1;  // for "for"
        s = s + A[i];
        count = count + 1;  // for assignment of
                            s
    }
    count = count + 1;  // Termination of "for" loop.
    return
    count = count + 1;  // for return statement.
    return s;
}
```

In this, for loop will be executed n times
count will be incremented by 2n.
There are 3 more steps to increment count.
∴ Expression for step count is,
∴ $t_p = 2n + 3$ or $t_{Sum}(n) = 2n + 3$

freq

2. Algorithm RSum (A, n)

```
{
    if (n<=0) then
    {
        return 0;
    }
    else
    {
        return RSum (A, n-1) + A[n];
    }
}
```

By applying temporary variable method, algorithm can be rewritten as,

```
Algorithm RSum (A, n)
{
    count = count+1;        // For conditional if
    if (n<=0) then
    {
        count = count+1;    // for Return stmt
        return 0;
    }
    else
    {
        count = count+1;    // for Return stmt
        return (RSum (A, n-1) + A[n]);
    }
}
```

By analyzing above algorithm, expression for step count as,

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

Above recursive formula is also called as Recurrence Relation.

By applying repeated substitution i.e. iteration method

$$\begin{aligned} t_{RSum}(n) &= 2 + t_{RSum}(n-1) \\ &= 2 + 2 + t_{RSum}(n-2) \\ &= 2 + 2 + 2 + t_{RSum}(n-3) \\ &= 2(3) + t_{RSum}(n-3) \\ &\vdots \\ &= 2(a) + t_{RSum}(n-a) \end{aligned}$$

If we put $a = n$ then,

$$\begin{aligned} t_{RSum}(n) &= 2(n) + t_{RSum}(n-n) \\ &= 2n + t_{RSum}(0) \\ &= 2n + 2 \\ &= 2(n+1) \end{aligned}$$

$$\boxed{t_{RSum}(n) = 2(n+1)}$$

3- 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]
        }
    }
}
```

By applying temporary variable method, algo can be written as,

Algorithm Add (A, B, C, m, n)

```
{
    for i=1 to m do
    {
        count = count +1;        // for 'i' loop
        for j=1 to n do
        {
            count = count +1;    // for 'j' loop
            C[i,j] = A[i,j] + B[i,j];
            count = count +1;    // for the assignment
        }
        count = count +1;        // terminating of j th loop.
    }
    count = count +1;            // terminating of i th loop.
}
```

Analysis :

Inner for loop will be executed $n * n$ times.
∴ count will be incremented by $2nn$

Outer for loop will be executed n times
∴ count will be incremented by $2n$

Finally, there is one single statement outside outer for loop.

∴ It will increment count by 1.

∴ $$t_{Add}(n) = 2nn + 2n + 1$$

Time Complexity (Table Method)

II Table Method

- To build a table in which list the total number of steps contributed by each statement.
- **No. of steps per execution (s/e):**
  It is the amount by which the count changes as a result of the execution of that statement.
- **Frequency:** It is determined for all the statements when it is multiplied with s/e, we will get **total count** for that statement.
- When total count for all the statements are added together, we will get time complexity.

Examples

		s/e	freq	total steps
1.	Algorithm Sum (A, n)	0	—	0
	{	0	—	0
	s = 0;	1	1	1
	for i = 1 to n do	1	n+1	n+1
	{	0	—	0
	s = s + A[i]	1	n	n
	}	0	—	0
	return s	1	1	1
	}	0	—	0
				2n+3

$t_{sum}(n) = 2n + 3$

Ex. 2

	s/e	freq n=0	freq n>0	total n=0	total n>0
Algorithm RSum(A, n)	0	–	–	0	0
{	0	–	–	0	0
if (n ≤ 0) then	1	1	1	1	1
{	0	–	–	0	0
return 0;	1	1	0	1	0
}	0	–	–	0	0
else	.				
{	0	–	–	0	0
return (RSum(A, n-1) + A[n]);	1+x	0	1	0	1+x
}	0	–	–	0	0
}	0	–	–	0	0
				2	2+x

Here,

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

$\therefore$ Relation is,

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

$t_{RSum}(n) = 2 + t_{RSum}(n-1)$

$= 2 + 2 + t_{RSum}(n-2)$

$= \vdots$

$= 2(a) + t_{RSum}(n-a)$

Substitute $a = n$,

$t_{RSum}(n) = 2n + t_{RSum}(n-n)$

$= 2n + 2$

$$\boxed{t_{RSum}(n) = 2(n+1)}$$

```
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]
    }
}
}
```

s/e	freq	Total steps
0	—	0
0	—	0
1	m+1	(m+1)
0	—	—
1	m(n+1)	m(n+1)
0	—	—
1	mn	mn
1	—	—
1	—	—
0	—	—

Total = (m+1) + m(n+1) + mn

= 2mn + 2m + 1

Ex 4.

```
Algorithm
I = 1
while (I<=n)
{
    x = x+I.
    I = I+1
}
```

s/e	freq	Total steps.
1	1	1
1	n+1	n+1
0	—	—
1	n	n
1	n	n
0	—	—

Total = (n+1) + n + n + 1

= 3n + 2