UNIT – I

INTRODUCTION

Algorithm:

Informal Definition:

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

Formal Definition:

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

Characteristics of Algorithm:

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

Issues or study of Algorithm:

- How to device or design an algorithm → creating and algorithm.

  Creating an algorithm is an art which may never be fully authomated. By studying various design techniques will become easier to devise new and useful algorithms.

- How to validate an algorithm →fitness.

  Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. This process is called algorithm validation. The purpose of validation is to assure us that this

algorithm will work correctly independently of the issues concerning the programming language it will be written in.

- How to analysis an algorithm → time and space complexity.

  Analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage an algorithm requires.

- How to test an algorithm → Checking for error.

  Testing a program consists of two phases:
  i) Debugging ii) Profiling

  Debugging is process of executing programs on sample data sets to determine whether faulty results occur and if so correct them.

  Profiling or Performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to compute the result.

Algorithm Specification:

Algorithm can be described in three ways.

1. Natural language like English:
When this way is choosed, care should be taken such that we should ensure that each & every statement is definite.

2. Graphic representation called flowchart:
This method will work well when the algorithm is small& simple.

3.Pseudo-code Method:
In this method, we should typically describe algorithms as program, which resembles language like Pascal & algol.

Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces {and}.

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

4. Compound data types can be formed with records. Here is an example,

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

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

5. Assignment of values to variables is done using the assignment statement.

<Variable>:= <expression>;

6. There are two Boolean values TRUE and FALSE.

→ Logical Operators AND, OR, NOT
→Relational Operators <, <=,>,>=, =, !=

7. The following looping statements are employed.

For, while and repeat-until

While Loop:

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

For Loop:

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

{
<statement-1>
.
.
<statement-n>
}

repeat-until:

repeat
<statement-1>
.
.
<statement-n>
until<condition>

8. A conditional statement has the following forms.

→ If <condition> then <statement>
→ If <condition> then <statement-1>
Else <statement-1>

Case statement:

Case
{
: <condition-1> : <statement-1>
.
.
: <condition-n> : <statement-n>
: else : <statement-n+1>
}

9. Input and output are done using the instructions read & write.

10. There is only one type of procedure:
Algorithm, the heading takes the form,

Algorithm Name (Parameter lists)

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

```
1. algorithm Max(A,n)
2. // A is an array of size n
3. {
4. Result := A[1];
5. for I:= 2 to n do
6.     if A[I] > Result then
7.           Result :=A[I];
8.    return Result;
9. }
```

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

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

Selection Sort:

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

Algorithm:

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

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

- We can solve the latter problem using the code,

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

- The first subtask can be solved by assuming the minimum is a[I];checking a[I] with a[I+1],a[I+2].....,and whenever a smaller element is found, regarding it as the new minimum. a[n] is compared with the current minimum.

- Putting all these observations together, we get the algorithm Selection sort.

Theorem:

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

Selection Sort:

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

Example:

LIST L = 3,5,4,1,2

1 is selected , → 1,5,4,3,2
2 is selected, →1,2,4,3,5
3 is selected, →1,2,3,4,5
4 is selected, →1,2,3,4,5

Proof:

- We first note that any I, say I=q, following the execution of lines 6 to 9,it is the case that a[q] Þ a[r],q<r<=n.

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

Algorithm:

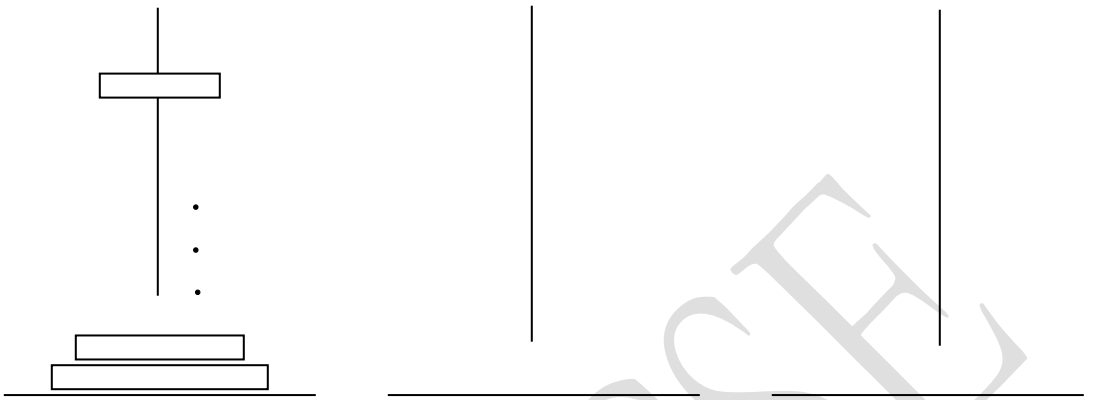
```
1. Algorithm selection sort (a,n)
2. // Sort the array a[1:n] into non-decreasing order.
3.{
4.      for I:=1 to n do
5.      {
6.              j:=I;
7.              for k:=i+1 to n do
8.                      if (a[k]<a[j])
9.                      t:=a[I];
10.                     a[I]:=a[j];
11.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

Recursive Algorithms:

- A Recursive function is a function that is defined in terms of itself.
- Similarly, 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 turns calls 'A'.
- The Recursive mechanism, are externally powerful, but even more importantly, many times they can express an otherwise complex process very clearly. Or these reasons we introduce recursion here.
- The following 2 examples show how to develop a recursive algorithms.

  → In the first, we consider the Towers of Hanoi problem, and in the second, we generate all possible permutations of a list of characters.

1. Towers of Hanoi:



- It is Fashioned after the ancient tower of Brahma ritual.
- According to legend, 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 in decreasing order of size bottom to top.
- Besides these tower there were two other diamond towers(labeled B & C)
- Since the time of creation, Brehman priests have been attempting 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.
- According to legend, the world will come to an end when the priest have completed this task.
- A very elegant 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-1' disks to tower C and then move the largest to tower B.
- Now we are left with the tasks of moving the disks from tower C to B.
- To do this, we have tower A and B available.
- The fact, that towers B has a disk on it can be ignored as the disks larger than the disks being moved from tower C and so any disk scan be placed on top of it.

Algorithm:

```
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 disk from ", X ,"to " ,Y);
8.          Towersofhanoi(n-1,z,y,x);
9.        }
10.}
```

2. Permutation Generator:

- Given a set of n>=1elements, the problem is to print all possible permutations of this set.
- For example, if the set is {a,b,c} ,then the set of permutation is,

  { (a,b,c),(a,c,b),(b,a,c),(b,c,a),(c,a,b),(c,b,a)}
- It is easy to see that given 'n' elements there are n! different permutations.
- A simple algorithm can be obtained by looking at the case of 4 statement(a,b,c,d)
- The Answer can be constructed by writing

1. a followed by all the permutations of (b,c,d)
2. b followed by all the permutations of(a,c,d)
3. c followed by all the permutations of (a,b,d)
4. d followed by all the permutations of (a,b,c)

Algorithm:

```
Algorithm perm(a,k,n)
{
if(k=n) then write (a[1:n]); // output permutation
else   //a[k:n] ahs more than one permutation
         // Generate this recursively.
for I:=k to n do
{
```

```
t:=a[k];
a[k]:=a[I];
a[I]:=t;
perm(a,k+1,n);
//all permutation of a[k+1:n]
t:=a[k];
a[k]:=a[I];
a[I]:=t;
}
}
```

Performance Analysis

1. **Space Complexity:**

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

2. **Time Complexity:**

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

Space Complexity:

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

a) A fixed part that is independent of the characteristics (eg:number,size)of the inputs and outputs. The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

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

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

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

Example :

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

- The problem instances for this algorithm are characterized by n, the number of elements to be summed. The space needed by 'n' is one word, since it is of type integer.
- The space needed by 'a' a is the space needed by variables of type array of floating point numbers.
- This is atleast 'n' words, since 'a' must be large enough to hold the 'n' elements to be summed.
- So,we obtain Ssum(n)>=(n+3)
  [n for a[],one each for n,I a& s]

Time Complexity:

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

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

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

For example, comments → 0 steps.
Assignment statements → 1 steps.

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

1.First method :

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

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

Algorithm:

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

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

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

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

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

Statement	*S/e*	*Frequency*	*Total*
1. Algorithm Sum(a,n)	0	-	0

2.{	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3

AVERAGE –CASE ANALYSIS

- Most of the time, average-case analysis are performed under the more or less realistic assumption that all instances of any given size are equally likely.
- For sorting problems, it is simple to assume also that all the elements to be sorted are distinct.
- Suppose we have 'n' distinct elements to sort by insertion and all n! permutation of these elements are equally likely.
- To determine the time taken on a average by the algorithm ,we could add the times required to sort each of the possible permutations ,and then divide by n! the answer thus obtained.
- An alternative approach, easier in this case is to analyze directly the time required by the algorithm, reasoning probabilistically as we proceed.
- For any I,2≤ I≤n, consider the sub array, T[1....i].
- The partial rank of T[I] is defined as the position it would occupy if the sub array were sorted.
- For Example, the partial rank of T[4] in [3,6,2,5,1,7,4] in 3 because T[1....4] once sorted is [2,3,5,6].
- Clearly the partial rank of T[I] does not depend on the order of the element in
- Sub array T[1...I-1].

Analysis

Best case:

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

Worst case:

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

Average case:

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

Complexity:

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

Asymptotic analysis:

Expressing the complexity in term of its relationship to know function. This type analysis is called asymptotic analysis.

Asymptotic notation

Big 'oh': the function f(n)=O(g(n)) if and only if there exist positive constants c and n_o such that **$f(n) \leq c*g(n)$ for all n, $n \geq n_o$.** The function g(n) is an upper bond on the value of f(n) for all n, where $n>=n_0$.

Example: The function 3n+2=O(n) as 3n+2<=4n for all n>=2.

Omega: the function $f(n)=\Omega(g(n))$ if and only if there exist positive constants c and n_o such that **$f(n) \geq c*g(n)$ for all n, $n \geq n_o$.** The function g(n) is a lower bond on the value of f(n) for all n, where $n>=n_0$.

Example: The function 3n+2=Ω(n) as 3n+2>=3n for n>=1

Theta: the function f(n)=ɵ(g(n)) iff there exist positive constants c1,c2 and no such that c1. g(n) ≤ f(n) ≤ c2. g(n) for all n, n ≥ no. The function g(n) provides upper and lower bond for the function f(n).

Example: The function 3n+2=theta(n) as 3n+2>=3n for all n>=2 and 3n+2<=4n for all n>=2.

Small 'oh' Notation (o):

Let f(n) and g(n0 be two non-negative functions.

F(n) is said to be o(g(n)) if and only if there exists positive constants k and n_0 such that f(n)<k*g(n) for all non-negative values of n, where n>=n_0.

It can also be defined as

Randomized Algorithms:

A randomized algorithm is one that makes use of a randomizer (such as random number generator). Since output of the randomizer might differ in an unpredictable way from run to run, the output of randomized algorithm could also differ from run to run for the same input. The execution time of randomized algorithm could also vary from run to run for the same input.

Randomized algorithm can be categorized into two classes:

1. Las Vegas Algorithms
2. Monte Carlo Algorithms

Las Vegas Algorithms:

Algorithms that always produce the same output for the same input are called Las Vegas Algorithms. The execution of a Las Vegas algorithm depends on the output of the randomizer.

Example for Las Vegas Algorithm:
Identifying the Repeated Element:

Consider an array a[] of n numbers that has n/2 distint elements and n/2 copies of another element. The problem is to identify the repeated element.

Any deterministic algorithm for solving this problem will need atleast n/2+2 time steps in the worst case. The algorithm given below returns the array index of one of the copies of the repeated element in O(log n) time.

```
RepeatedElement(a,n)
// Finds the repeated element from a[1:n]
{
  while (true) do
   {
       I :=Random() mod n+1;
       J :=Random() mod n+1;
       // I and j are random numbers in the range [1,n]
      If ((i≠j) and (a[i]=a[j]) then return I;
  }
}
```

The above algorithm randomly picks two array elements and checks whether they come from two different cells and have the same value. If they do the repeated ele ments has been found. If not, this basic step of sampling is repeated as many times as it takes to identify the repeated element.

Monte Carlo algorithms:

Algorithms whose outputs might differ from run to run are called Monte Carlo algorithms.

Example of Monte Carlo algorithm:
Primality testing:

The problem of deciding whether n is a prime known as primality testing.

Naïve primality testing algorithm has a runtime of $O(\sqrt{n})$ (here $\sqrt{n} = 2^{1/2\log n}$) where as a Monte Carlo randomized algorithm for primality testing that runs in time $O((\log n)^2)$.

Fermat theorem: If n is prime, then $a^{n-1} = 1 \pmod n$ for any integer $a<n$.

Fermat's theorem suggests the following algorithm for primality testing: Randomly choose an a<n and check whether $a^{n-1} = 1 \pmod n$. If fermat's equation is not satisfied, n is composite. Fermat algorithm may give an incorrect answer.
Miller-Rabin's algorithm is a slight modification of the Fermat's algorithm which will never give an incorrect answer if an input is prime.

Algorithm for Miller-Rabin's primality tesing algorithm:

```
Prime(n, α)
//Return true if n is a prime and false otherwise
//α is the probability parameter
{
    q:= n-1;
    for i:= 1 to α * lon(n) do
    {
        m := q; y :=1;
        a := Random() mod q+1;
        //Choose a random number in the range [1, n-1].
        z := a;
        // Compute a^(n-1) mod n.
        while (m > 0) do
        {
            while (m mod 2 = 0) do
            {
                x :=z; z:=z^2 mod n;
                // if x is a nontrivial square
                // root of 1, n is not a prime.
                if ((z =1) and (x ≠ 1) and (x ≠ q)) then
                return false;
                m := ⌊ m/2 ⌋ ;
            }
            m := m-1; y:=(y*z) mod n;
        }
        if (y ≠ 1) return false;
        // If a^(n-1) mod n is not 1, n is not a prime.
    }
    return true;
}
```

* * * * *