UNIT – I

Analysis of Algorithm:

INTRODUCTION – ANALYZING CONTROL STRUCTURES-AVERAGE CASE ANALYSIS-SOLVING RECURRENCES.

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, an algorithm is a sequence of computational steps that transforms the i/p into the o/p.

Formal Definition:

An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms should satisfy the following criteria.

1. INPUT → Zero or more quantities are externally supplied.
2. OUTPUT → At least one quantity is produced.
3. DEFINITENESS → Each instruction is 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 be fundamental so that it can be carried out, in principle, by a person using only pencil & paper.

Issues or study of Algorithm:

- How to devise or design an algorithm → creating an algorithm.
- How to express an algorithm → definiteness.
- How to analyze an algorithm → Time and space complexity.
- How to validate an algorithm → fitness.
- Testing the algorithm → checking for error.

Algorithm Specification:

An algorithm can be described in three ways.

1. Natural language like English: When this way is choused, care should be taken; 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 a program that resembles language like Pascal & algol.

Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of the 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, the 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 two examples to illustrate the process of translating a problem into an algorithm.

Selection Sort:

- Suppose we devise an algorithm that sorts a collection of n>=1 elements of arbitrary type.
- A Simple solution is given by the following.
- (From those currently unsorted elements,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 more minor 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 most minor element in the list. This element is moved to the front. Then, the most minor element among the remaining elements is found and put into the second position. This procedure is repeated till the entire list has been studied.

Example:

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

One is selected, → 1,5,4,3,2
Two is selected, →1,2,4,3,5
Three are selected, →1,2,3,4,5
Four are 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 line 4 can be changed to n-1 without damaging the correctness of the algorithm.

Algorithm:

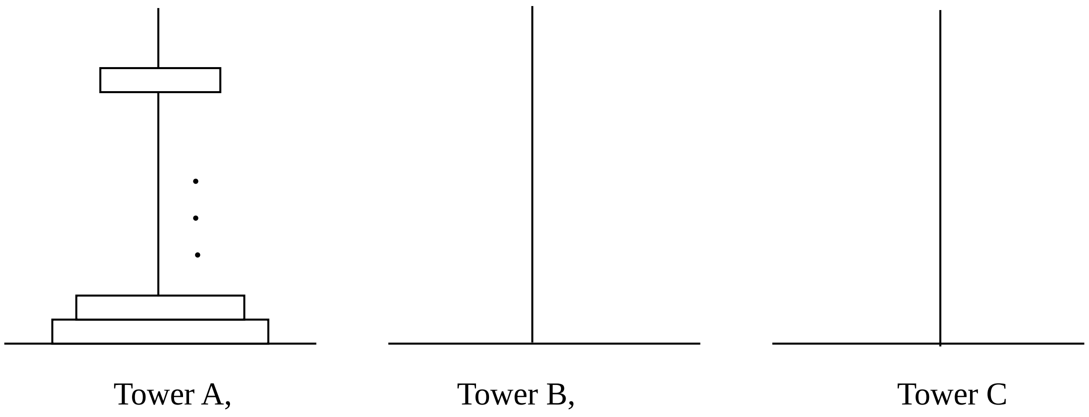
```
1. Algorithm selection sort (a,n)
2. // Sort the array a[1:n] into non-decreasing order.
3. {
4.      for I:=1 to n, do
5.      {
6.           j:=I;
7.           for k:=i+1 to n do
8.                if (a[k]<a[j])
9.                t:=a[I];
10.               a[I]:=a[j];
11.               a[j]:=t;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

Recursive Algorithms:

- A Recursive function is a function that is defined in terms of itself.
- Similarly, an algorithm is considered recursive if the same algorithm is invoked in the body.
- An algorithm that calls itself Direct Recursive.
- Algorithm 'A' is called Indirect Recursive if it calls another algorithm, which calls 'A'.
- The Recursive mechanisms are externally powerful, but even more importantly, they can often express an otherwise complex process very clearly. For these reasons, we introduce recursion here.
- The following two examples show how to develop recursive algorithms.

  → In the first, we consider the Towers of Hanoi problem; 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 the Brahma ritual.
- According to legend, a diamond tower (labeled A) with 64 golden disks was created when the world was created.
- The disks were of decreasing size and were stacked on the tower in decreasing order of size from bottom to top.
- Besides this tower, there were two other diamond towers(labeled B & C)
- Since 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, a disk cannot be on top of a smaller disk.
- According to legend, the world will come to an end when the priest has completed this task.

- an exquisite solution results from the use of recursion.
- Assume that the number of disks is ‘n’.
- To get the giant 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 moving the disks from tower C to B.
- To do this, we have towers A and B available.
- The fact that tower B has a disk on it can be ignored as the disks are more significant than the disks are being moved from tower C so that any disk can 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 top disk from tower “ X,” to top of the tower, “Y);
8.  Towersofhanoi(n-1,z,y,x);
9.  }
10. }
```

2. Permutation Generator:

- Given a set of n>= 1 elements, 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 statements (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] has 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 money it needs to run to compile.