Introduction to Algorithms

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

In addition, all algorithms must satisfy the following criteria :

Input. Zero or more quantities are externally supplied.

Output. At least one quantity is produced.

Definiteness. Each instruction is clear and unambiguous

Finiteness. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

Effectiveness. Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible.

Process of translating a problem into an algorithm

Sum of n no's

Abstract solution:

Sum of n elements by adding up elements one at a time.

Little more detailed solution:

```
Sum=0;
add a[1] to a[n] to sum one element at a time.
return Sum;
```

More detailed solution which is a formal algorithm:

```
Algorithm Sum (a,n)
{
    sum:= 0.0;
    for i:= 1 to n do
        sum:=sum+a[i];
    Return sum;
}
```

Find largest among list of elements

Abstract solution:

Find the Largest number among a list of elements by considering one element at a time.

Little more detailed solution:

1. Treat first element as largest element
2. Examine a[2] to a[n] and suppose the largest element is at a [i];
3. Assign a[i] to largest;

4. Return largest**;**

More detailed solution which is a formal algorithm:

```
Algorithm Largest (a,n)
{
        largest := a[1];
        for i := 2 to n do
        {
            if ( a[i] > largest ) largest := a[i];
        }
        Return largest;
}
```

Linear search

Abstract solution:

From the given list of elements compare one by one from first element to last element for the required element

Little more detailed solution:

```
for i := 1 to n do
{
     Examine a[i] to a[n] and suppose
                    the required element is at a [ j ];
      Write match is found and return position  j;
 }
```

More detailed solution which is a formal algorithm:

```
Algorithm Linear search (a, req_ele)
{
     Flag=0
     for i := 1 to n do
     {
              If (a[i] = req_ele) then
              {
                 Flag=1;
                 Pos = i;
                 break;
            }
     }
     If (flag) then
         Write "element found at 'pos'  position"
     Else
         Write "element not found"
     End if
}
```

Selection Sort

Abstract solution:

From those elements that are currently unsorted, find the smallest and place it next in the sorted list.

Little more detailed solution:

```
for i := 1 to n do
{
    Examine a[i] to a[n] and suppose
            the smallest element is at a [ j ];
    Interchange a[i] and a[j];
}
```

More detailed solution which is a formal algorithm:

```
Algorithm Selection Sort (a, n)
// Sort the array a[1 : n] into non decreasing order.
{
    for i := 1 to n do
    {
        min:=i;
        for k : i + 1 to n do
            If (a[k]< a[min]) then  min:=k;
        t :=a[i]; a[i] := a[min];a[min]:= t;
    }
}
```

Bubble Sort

Abstract solution:

Comparing adjacent elements of an array and exchange them if they are not in order. In the process, the largest element bubbles up to the last position of the array first and then the second largest element bubbles up to the second last position and so on.

Little more detailed solution:

```
for i := 1 to n do
{
        Compare a[1] to a [n-i]  pair-wise (each element with its next)
        and interchange a[j] and a[j+1] whenever  a[j] is greater than a[j+1]
}
```

More detailed solution which is a formal algorithm:

```
Algorithm BubbleSort (a,n)
{
   for i := 1 to n do
   {
     for j :=1 to n-i do
```

```
        {
          if ( a[j] > a[j+1] )
          {
            // swap a[j] and a[j+1]
            temp := a[j]; a[j] := a[j+1];a[j + 1] := temp;
          }
        }
      }
    }
```

Insertion sort

Abstract solution:

Insertion sort works by sorting the first two elements and then inserting the third element in its proper place so that all three are sorted. Repeating this process, k+1 st element is inserted in its proper place with respect to the first k elements (which are already sorted) to make all k+1 elements sorted. Finally, 'n' th element is inserted in its proper place with respect to the first n-1 elements so all n elements are sorted.

Little more detailed solution:

```
Sort a[1] and a [2]
For i from 3 to n do
{
    Suppose a[k] (k between 1 and i) is such that a[k-1] <= a[i] <= a[k].
    Then, move a[k] to a[i-1] by one position and insert a[i] at a[k].
}
```

More detailed solution which is a formal algorithm:

```
Algorithm insertionSort(a,n)
{
   for i := 2 to n do
   {
      value := a[i]; j := i - 1;
      done := false;
      repeat
         if a[j] > value
         {
            a[j + 1] := a[j];  j := j - 1;
            if (j < 1) done := true;
         }
         else
            done := true;
      until done;
      a[j + 1] := value;
   }
}
```

Pseudo-code conventions

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces: { and }. A compound statement (i.e., a collection of simple statements) can be represented as a block. The body of a procedure also forms a block. Statements are delimited by;.
3. An identifier begins with a letter. The data types of variables are not explicitly declared. The types will be clear from the context. Whether a variable is global or local to a procedure will also be evident from the context. We assume simple data types such as integer, float, char, Boolean, and so on. Compound data types can be formed with records. Here is an example:

```
Node = record
{     datatype_1  data_1;
      ::::::::::::::::::
      Datatype_n  data_n;
      node        *link;
}
```

   In this example, link is a pointer to the record type node. Individual data items of a record can be accessed with $\rightarrow$ and period. For instance if p points to a record of type node, p$\rightarrow$data_1 stands for the value of the first field in the record. On the other hand, if q is a record of type node, q. dat_1 will denote its first field.
4. Assignment of values to variables is done using the assignment statement

   (variable) :=(expression);

5. There are two Boolean values **true** and **false.** In order to produce these values, the logical operators **and**, **or**, and **not** and the relational operators <, $\leq$, =, $\neq$, $\geq$, and > are provided.
6. Elements of multidimensional arrays are accessed using [and]. For example, if A is a two dimensional array, the (i,j) the element of the array is denoted as A[i,j]. Array indices start at zero.
7. The following looping statements are employed: **for**, **while**, and **repeat- until**. The **while** loop takes the following form:

```
While (condition) do
{
      (statement 1)
      ::::::::::::::
      (statement n)
}
```

As long as (condition) is **true,** the statements get executed. When (condition) becomes **false**, the value of (condition) is evaluated at the top of loop.

The general form of a **for** loop is

```
for variable := value 1 to value 2 step step do
{
        (statement 1)
        ::::::::::::::
        (statement n)
}
```

Here value1, value2, and step are arithmetic expressions. A variable of type integer or real or a numerical constant is a simple form of an arithmetic expression. The clause **"step** step" is optional and taken as +1 if it does not occur. Step could either be positive or negative variable is tested for termination at the start of each iteration. The **for** loop can be implemented as a **while** loop as follows:

```
Variable := value1;
fin :=value2;
incr:=step;
while ((variable –fin) * step ≤,0) do
{
        (statement 1)
        ::::::::::::::
        (statement n)
}
```

A **repeat-until** statement is constructed as follows :

```
repeat
        (statement 1)
        ::::::::::::::
        (statement n)
Until (condition)
```

The statements are executed as long as (condition) is **false**. The value of (condition) is computed after executing the statements.

The instruction **break;** can be used within any of the above looping instructions to force exit. In case of nested loops, **break;** results in the exit of the innermost loop that it is a part of. A return statement within any of the above also will result in exiting the loops. A **return** statement results in the exit of the function itself.

8. A conditional statement has the following forms:

if (condition) **then** (statement)

if (condition) **then** (statement 1) **else** (statement 2)

Here (condition) is a boolean expression and (statement),(statement 1),

and (statement 2) are arbitrary statements (simple or compound).

We also employ the following **case** statement:

```
Case
{
   :(condition 1 ) : (statement 1)
      ::::::::::::::
   :(condition n ) : (statement 1)
   :else : (statement n+1)
}
```

Here (statement 1) and (statement 2), etc. could be either simple statements or compound statements. A **case** statement is interpreted as follows. **If** (condition 1) is **true,** (statement 1) gets executed and the **case** statements is exited. If (statement 1) is **false**,(condition 2) is evaluated. **If** (condition 2) is **true,** (statement 2) gets executed and the case statement exited, and so on. **If** none of the conditions (condition 1),…,(condition n) are true, (statement n+1) is executed and the case statement is exited. The **else** clause is optional.

9. Input and output are done using the instructions **read** and **write,** No format is used to specify the size of input or output quantities.

10. There is only one type of procedure: **Algorithm.** An algorithm consists of a heading and a body. The heading takes the form.

Algorithm Name ((parameter list))

Where Name is the name of the procedure and ((parameter list)) is a listing of the procedure parameters. The body has one or more (simple or compound) statements enclosed within braces { and }. An algorithm may or may not return any values. Simple variables to procedures are passed by value. Arrays and records are passed by reference. An array name or a record name is treated as a pointer to the respective data type.

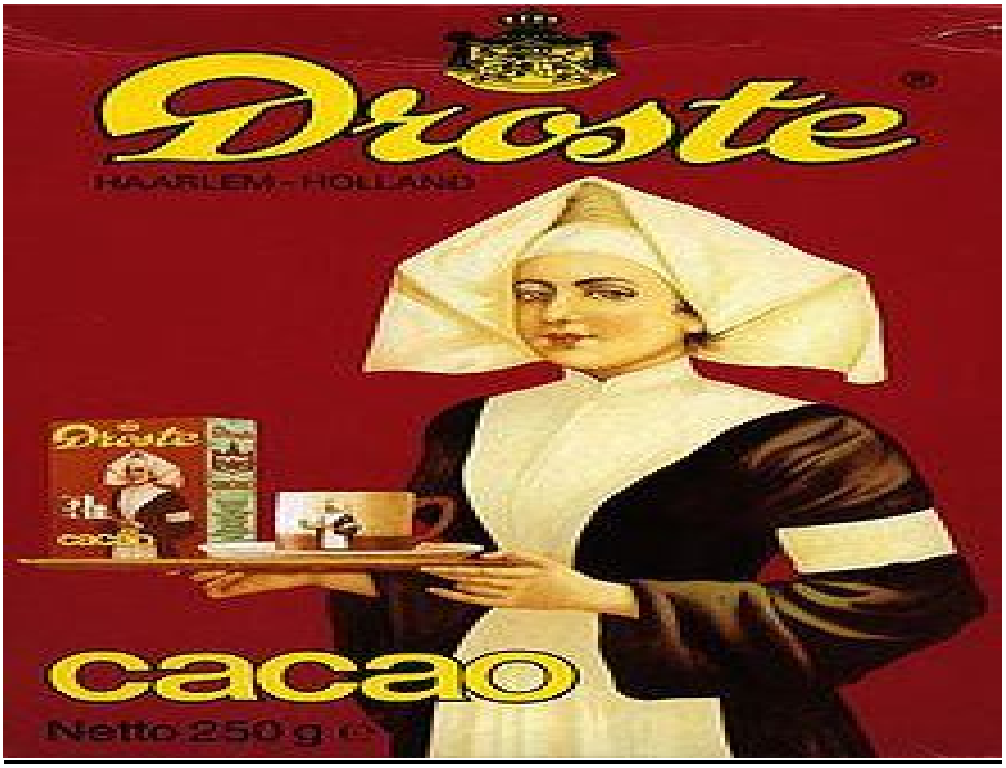
As an example. The following algorithm finds and returns the maximum of n given numbers:

```
Algorithm Max (A, n)
// A is an array of size n.
 {
      Result := A[1];
      for i: = 2 to n do
            If A[i] > Result then Result := A[i];
      Return Result;
 }
```

RECURSION

Definition: It is the process of repeating items in a self similar way.

Example 1:



Recursion has most common applications in Mathematics and computer science.

Example2:

The following definition for natural numbers is Recursive.

1 is a Natural Number.

If k is a natural number then k+1 is a Natural Number.

DEFINITION(Computer Science): This is the method of defining a function in which the function being defined is applied within its own definition.

In simple terms if a function makes a call to itself then it is known as Recursion.

Example3: Observe the recursive definition for a function Suml(n) to compute

Sum of first n natural numbers.

```
If  n=1 then
        return 1
 else
        return n + Sum(n-1)
```

Steps to be followed to write a Recursive function

1.BASIS: This step defines the case where recursion ends.

2.RECURSIVE STEP: Contains recursive definition for all other cases(other than base case)to reduce them towards the base case

For Example in Example 3,

Basis is n=1 and

Recursive step is n + Sum(n-1)

Note: if any one of the two or both, wrongly defined makes the program infinitely Recursive.

Recursive solution for the Sum of nos

First Call for Recursion:

RecursiveSum **(a,n);**

```
Algorithm RecursiveSum(a,k)
{
    if (k<=0) then
        Return 0.0;
    else
        Return (RecursiveSum(a,k-1) + a[k]);
}
```

Recursive solution for the Largest of 'n' numbers

First Call for Recursion:

RecursiveLargest **(a,n);**

```
Algorithm RecursiveLargest (a,k)
{
        if (k = 1) then Return a[k];
        else
        {
                x := RecursiveLargest (a,k-1);
                if (x > a[k]) Return x;
                else Return a[k];
         }
}
```

Recursive solution for the SelectionSort

First call of recursion is:

RecursiveSelectionSort **(a,1);**

```
Algorithm RecursiveSelectionSort (a, m)
{
    if (m = n) Return; // Do nothing
    else
    {
            min:=m;
            for k := m + 1 to n do
                    if (a[k]< a[min]) then  min:=k;
            //swap
            t :=a[m]; a[m] := a[min];a[min]:= t;
            RecursiveSelectionSort (a, m+1);
        }
}
```

Recursive solution for the Linear Search

```
Algorithm linearSearch(a, key, size)
{
      if (size == 0) then return -1;
      else if (array[size ] = key) return size ;
      else return linearSearch (array, key, size - 1);
}
```

Recursive solution for the BubbleSort

```
Algorithm RecursiveBubbleSort (a,k)
{
   if (k = 1)
         Return; // Do nothing
   else
   {
      for j :=1 to k-1 do
      {
         if ( a[j] > a[j+1] )
         {
           // swap a[j] and a[j+1]
           temp := a[j]; a[j] := a[j+1];
           a[j + 1] := temp;
         }
      }
      RecursiveBubbleSort (a,k-1);
   }
}
```

Recursive solution for the InsertionSort

First call of recursion is:

RecursiveInsertionSort(a,n);

```
Algorithm RecursiveInsertionSort(a,k)
{
   if (k > n)
      Return;  // Do nothing
   else
   {
        RecursiveInsertionSort(a,k-1);
        //Insert 'k'th element into k-1 sorted list
        value := a[k]; j := k - 1;
        done := false;
        repeat
            if a[j] > value
            {
                    a[j + 1] := a[j];  j := j - 1;
                    if (j < 0) done := true;
            }
            else done := true;
        until done;
        a[j + 1] := value;
   }
}
```

Recursive solution for the Printing an array in normal order

First call of recursion is:

Recursive Prin_arr(a,n-1);

```
Recursive Algorithm prin_arr(int a[],int n)
{
        if(n=-1)
                return;
        else
                prin_arr(a,n-1);
        write a[n]);
}
```

Recursive solution for the Decimal to binary conversion

```
Algorithm bin(int n)
{
  if (n=1 or n=0)
  {
        write  n;
```

```
            return;
        }
        else
            bin=n/2;
        write  n mod 2;
    }
```

Recursive solution for the Fibonacci number

```
    Recursive Algorithm fib(int x)
    {
        if(x=0 or x=1) then
            return 0;
        else
        {
            if(x=2)
                return 1;
            else
            {
                f=fib(x-1) + fib(x-2);
                return(f);
            }
    }
```

Recursive solution for the Factorial of a given number

```
  Recursive Algorithm fact (int n)
{
   {
   int fact;
   if n=1 tehn
   return 1;
   else
   fact=n*(n-1);
   }
return fact;
}
```

Towers of Hanoi with recursion:

Rules:

(1) Can only move one disk at a time.
(2) A larger disk can never be placed on top of a smaller disk.
(3) May use third post for temporary storage.

```
    Algorithm TowersOfHanoi (numberOfDisks, x,y,z)
    {
        if (numberOfDisks >= 1)
```

```
    {
        TowersOfHanoi (numberOfDisks -1, x,z, y);
        move (start, end);
        TowersOfHanoi (numberOfDisks -1, z, y, x);
    }
}
```

Performance Analysis

Space Complexity:

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

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

1. 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.
2. 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 2:

```
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 d 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 at least 'n' words, since 'a' must be large enough to hold the 'n' elements to be summed.
   - So, we obtain Sum(n)>=(n+s)
     [n for a[ ],one each for n, I a& s]

```
Alogorithm RSum(a,n)
{
```

```
        If(n<= 0) then
                return 0.0;
        Else
                return RSum(a,n-1)+a[n];
}
```
Recursive function for Sum

Time Complexity:

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

For Ex: **Return a+b+b*c+(a+b-c)/(a+b)+4.0;**

Program step count method – examples:

```
Algorithm Sum (a,n)
{
  s:= 0.0;
  Count :=Count+=1; // count is global;it is initially zero.
  for i:=1 to n do
  {
        Count :=count+1 //for  for
        S :=S+a[i];count:=count+1;For assignment
  }
  Count:=count+1; // For last time of for
  Count:=count +1; //For the return
  return s;
  }
```
Algorithm SUM with count statements added

```
Algorithm  Sum (a,n)
{
for i:=to n do count:=count+2;
  count :=count +3;
}
```
Algorithm for Sum Simplified version for previous sum

```
Algorithm   RSum(a,n)
{
    Count :=count+1;    // For the if conditional
        if (n ≤ 0)then
        {
                count:=count+1//For  the return
                return  0.0;
        }
        else
```

```
{
                count :=count+1 // For the addition, function invocation and return
                return RSum(a,n-1)+a[n];
        }
    }
```

Algorithm Recursive sum with count statements added

```
Algorithm  Add (a,b,c,m,n)
    {
       for i:= 1to m do
        for j:=1 to n do
                c[i,j] := a[i,j] + b[i,j];
    }
```

Matrix Addtion

```
    Algorithm  Add(a,b,c,m,n)
    {
      for i:=1 to m do
      {
         count :=count +1;  // For 'for i'
         for j :=1 to n do
        {
          count :=count+1;      // For 'for j'
          c[i, j] := a[i, j]+b[i, j];
          count :=count +1;    //For the assignment
         }
         count := count +1 ;// For loop initialization and last time of 'for  j'
      }
      count := count+1 ;    // For loop initialization and last time of 'for i'
     }
```

Algorithm for Matrix addition with counting statements

Step table method:

Statement	s/e	frequency	total steps
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**	1	1	1
7 }	0	-----	0
TOTAL			2n +3

Step table for Algorithm finding Sum

Statement	s/e	Frequency n=0	Frequency n>0	total steps n=0	total steps n>0
1.Algorithm RSum(a,n)	0	-	-	1.	0
2.{					
3. if(n≤ 0) then	1	1	1	1	1
4. return 0.0:	1	1	0	1	0
5. else return					
6.RSum (a,n-1) + a[n];	1+x	0	1	0	1+x
7. }	0	---	---	0	0
Total				2	2+x

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

Step table for Algorithm RSum

Statement	s/e	frequency	Total steps
1. **Algorithm Add(a,b,c,m,n)**	0	-----	0
2. {	0	-----	0
3. for i := 1 to m do	1	m+1	m+1
4. for j := 1 to n do	1	m(n+1)	mn + m
5. c[i,j] :=a[i,j] + b[i,j];	0	mn	mn
6. }		------	0
Total			2mn +2m +1

Step table for Algorithm Addition of two matrices

Statement	s/e	frequency	Total steps

Algorithm BubbleSort (a,n)
{
f**or** i := 1 to n **do**
{
for j :=1 to n-i do
{
if (a[j] > a[j+1])
{
// swap a[j] and a[j+1]
temp := a[j];
a[j] := a[j+1];
a[j + 1] := temp;
}
}
}
}			

Total			

Fill the blank columns in the above Table

Statement	s/e	frequency	Total steps

Algorithm Linear search (a, req_ele)
{
Flag=0
for i := 1 **to** n **do**
{
If (a[i] = req_ele) **then**
{
Flag=1;
Return i;
}
}
If (flag) then
Write “element found at i position”
Else
Write “element not found
}			

Total			

Fill the blank columns in the above Table