CMR INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | |

# Internal Assessment Test 1 – September 2019

| **Sub:** | **Design and Analysis of Algorithms** | | | | | | | **Code:** | **18MCA33** |
|---|---|---|---|---|---|---|---|---|---|
| *Date:* | *7-09-2019* | *Duration:* | *90 mins* | *Max Marks:* | *50* | *Sem:* | **III** | *Branch:* | *MCA* |

**Answer any five of the following** **5 x 10 = 50 Marks**

**Q1** What is an algorithm? What are the characteristics of a good algorithm? Explain with example of GCD of two numbers **(5)**

**Def** : An algorithm is a sequence of unambiguous instructions for solving a problem. i.e., for obtaining a required output for any legitimate input in a finite amount of time.

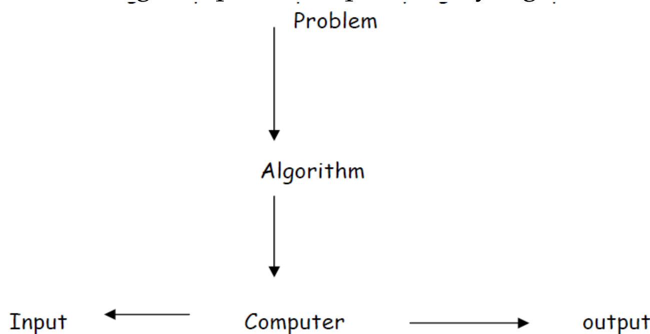



Figure : Notion of the Algorithm

**Characteristics of Algorithms:**

i) **Finiteness:**

An algorithm must terminate after a finite number of steps and further each step must be executable in finite amount of time or it terminates (in finite number of steps) on all allowed inputs

ii) **Definiteness (no ambiguity):**

Each step of an algorithm must be precisely defined; the action to be carried out must be rigorously and unambiguously specified for each case. For example : an instruction such as y=sqrt(x) may be ambiguous since there are two square roots of a number and the step does not specify which one.

iii) **Inputs:**

An algorithm has zero or more but only finite, number of inputs.

iv) **Output:**

An algorithm has one or more outputs. The requirement of at least one output is obviously essential, because, otherwise we cannot know the answer/ solution provided by the algorithm. The outputs have specific relation to the inputs, where the relation is defined by the algorithm.

v) **Effectiveness:**

An algorithm should be effective. This means that each of the operation to be performed in an algorithm must be sufficiently basic that it can, in principle, be done exactly and in a finite length of time, by person using pencil and paper. Effectiveness also indicates correctness, i.e. the algorithm actually achieves its purpose and does what it is supposed to do.

**Example:**

Below is given the psuedocode of the algorithm to find the GCD of two numbers

```
Algorithm Euclid (m, n)
// Computer gcd (m, n) by Euclid's algorithm.
// Input: Two nonnegative, not-both-zero integers m&n.
//output: gcd of m&n.
While n# 0 do
        R=m mod n
        m=n
        n=r
return m
```

Considering the above algorithm it is finite. Though we do not offer a proof here, it can be seen that the pair of m and n after every step decreases. If we start with m and n as positive numbers then eventually the value of n has to reduce and become 0 thus guaranteeing termination and thus ***finiteness.***

***Definiteness*** **–** Every step in this algorithm is well specified and has no ambiguity

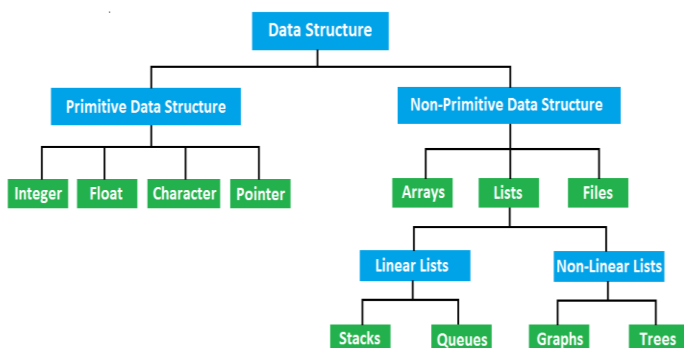
***Inputs / Ouput*** – The algorithm has two inputs and one output – gcd.

***Effectiveness*** – Each step is presented in sufficient detail and the result is a correct computation of GCD.

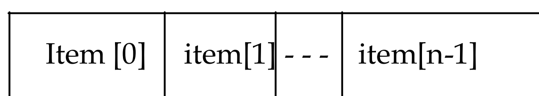
**( 2) Explain the fundamental data structures used for designing algorithms.**

A data structure can be defined as the logical or mathematical model of a particular organization of data. In otherwords,  An efficient way of storing and organizing data in the computer such as queue, stack, linked list and tree.

Classification of Data structures:




The two most important elementary data structure are the array and the linked list. Array is a sequence contiguously in computer memory and made accessible by specifying a value of the array's index.

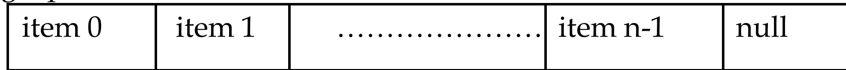



Array of n elements.

The index is an integer ranges from 0 to n-1. Each and every element in the array takes the same amount of time to access and also it takes the same amount of computer storage.

Arrays are also used for implementing other data structures. One among is the string: a sequence of alphabets terminated by a null character, which specifies the end of the string. Strings composed of zeroes and ones are called binary strings or bit strings. Operations performed on strings are: to concatenate two strings, to find the length of the string etc.

A linked list is a sequence of zero or more elements called nodes each containing two kinds of information: data and a link called pointers, to other nodes of the linked list. A pointer called null is used to represent no more nodes. In a singly linked list, each node except the last one contains a single pointer to the next element.




Singly linked list of n elements.

To access a particular node, we start with the first node and traverse the pointer chain until the particular node is reached. The time needed to access depends on where in the list the element is located. But it doesn't require any reservation of computer memory, insertions and deletions can be made efficiently.

There are various forms of linked list. One is, we can start a linked list with a special node called the header. This contains information about the linked list such as its current length, also a pointer to the first element, a pointer to the last element.

Another form is called the doubly linked list, in which every node, except the first and the last, contains pointers to both its success or and its predecessor.

The another more abstract data structure called a linear list or simply a list. A list is a finite sequence of data items, i.e., a collection of data items arranged in a certain linear order. The basic operations performed are searching for, inserting and deleting on element.

Two special types of lists, stacks and queues. A stack is a list in which insertions and deletions can be made only at one end. This end is called the top. The two operations done are: adding elements to a stack (popped off). Its used in recursive algorithms, where the last- in- first-out (LIFO) fashion is used. The last inserted will be the first one to be removed.

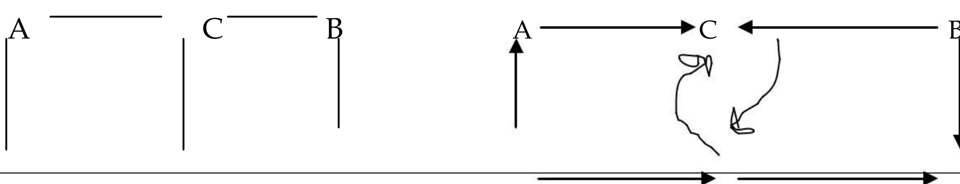
A queue, is a list for, which elements are deleted from one end of the structure, called the front (this operation is called dequeue), and new elements are added to the other end, called the rear (this operation is called enqueue). It operates in a first- in-first-out basis. Its having many applications including the graph problems.

A priority queue is a collection of data items from a totally ordered universe. The principal operations are finding its largest elements, deleting its largest element and adding a new element. A better implementation is based on a data structure called a heap.

**Graphs**:

A graph is informally thought of a collection of points in a plane called vertices or nodes, some of them connected by line segments called edges or arcs. Formally, a graph G=<V, E > is defined by a pair of two sets: a finite set V of items called vertices and a set E of pairs of these items called edges. If these pairs of vertices are unordered, i.e. a pair of vertices (u, v) is same as (v, u) then G is undirected; otherwise, the edge (u, v), is directed from vertex u to vertex v, the graph G is directed. Directed graphs are also called digraphs.

Vertices are normally labeled with letters / numbers



D———— E ——— F D E F

1. (a) Undirected graph 1.(b) Digraph

The 1st graph has 6 vertices and seven edges.

V = {a, b, c, d, e,f },
E = {(a,c) ,( a,d ), (b,c), (b,f ), (c,e),( d,e ), (e,f) }

The digraph has four vertices and eight directed edges:

V = {a, b, c, d, e, f},
E = {(a,c), (b,c), (b,f), (c,e), (d,a), (d, e), (e,c), (e,f) }

Usually, a graph will not be considered with loops, and it disallows multiple edges between the same vertices. The inequality for the number of edges | E | possible in an undirected graph with |v| vertices and no loops is :

$0 <= |E| <= |v| ( |V| - ) / 2.$

A graph with every pair of its vertices connected by an edge is called complete. Notation with |V| vertices is $K_{|V|}$ . A graph with relatively few possible edges missing is called dense; a graph with few edges relative to the number of its vertices is called sparse.

Q3. Describe the various asymptotic notations with a neat diagrams and examples.

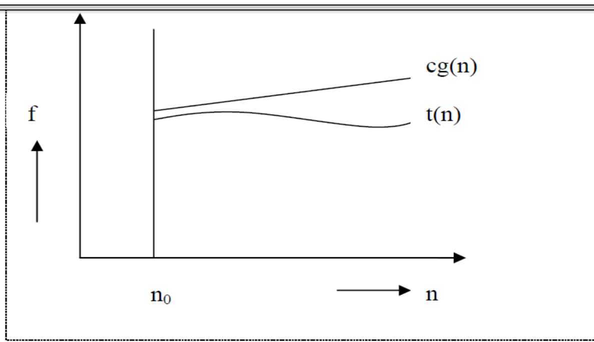
Different Notations

1. Big oh Notation
2. Omega Notation
3. Theta Notation

**1.** Big oh (O) Notation : A function t(n) is said to be in O[g(n)], t(n) ∈ O[g(n)] , if t(n) is bounded above by some constant multiple of g(n) for all large n ie.., there exist some positive constant c and some non negative integer no such that t(n) ≤ cg(n) for all n≥no.
Eg. t(n)=100n+5 express in O notation

100n+5 < = 100n + n for all n>=5
< = 101 (n2)
Let g(n)= n2 ; n0=5 ; c = 101
i.e 100n+5 <=101 n2
t(n) <= c* g(n) for all n>=5

There fore , t(n) ∈ O(n2)




2. Omega(Ω) -Notation:

Definition: A function t(n) is said to be in Ω[g(n)], denoted t(n) ∈ Ω[g(n)] , if t(n) is bounded below by some positive constant multiple of g(n) for all large n, ie., there exist some positive constant c and some non negative integer n0 such that

$t(n) \geq cg(n)$ for all $n \geq n0$.

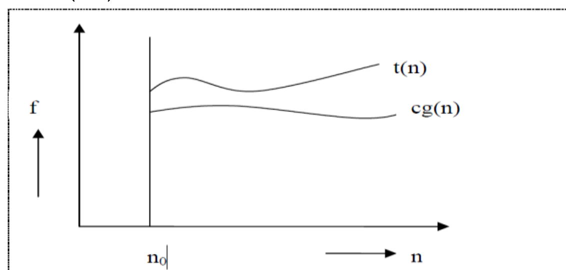
For example:

$t(n) = n3 \in \Omega(n2)$,

$n3 \geq n2$ for all $n \geq n0$.

we can select, g(n)= n3 , c=1 and n0=0

$t(n) \in \Omega(n2)$,




3. Theta (θ) - Notation:

Definition: A function t(n) is said to be in θ [g(n)], denoted t(n) ∈ θ (g(n)), if t(n) is bounded both above and below by some positive constant multiples of g(n) for all large n , ie., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that $c2g(n) \leq t(n) \leq c1g(n)$ for all $n \geq n0$.

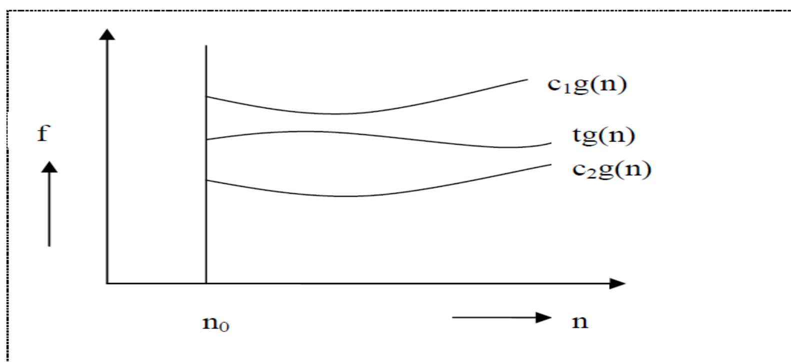
For example 1:

t(n)=100n+5 express in θ notation

100n <= 100n+5 <= 105n for all n>=1

c1=100; c2=105; g(n) = n;

Therefore , $t(n) \in \theta(n)$

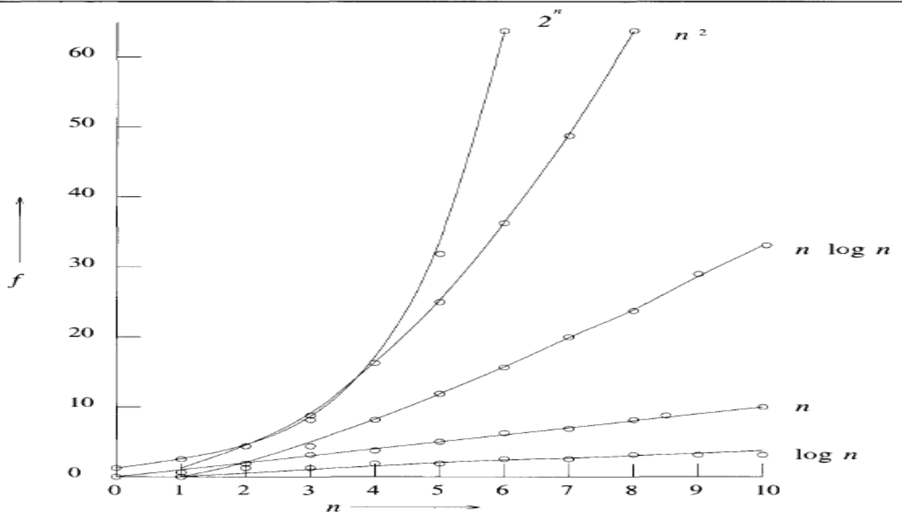



**Describe various Basic Efficiency classes**

Sol: The time complexity of a large number of algorithms fall into only a few classes. These classes are listed in Table in increasing order of their orders of growth. Although normally we would

expect an algorithm belonging to a lower efficiency class to perform better than an algorithm belonging to higher efficiency classes, theoretically it is possible for this to be reversed. For example if we consider two algorithms with orders (1.001)n and n1000. Then for lot of values of n (1.001)n would perform better but it is rare for an algorithm to have such time complexities.

| Class | Name | Comments |
|---|---|---|
| 1 | Constant | Constant time algorithm execute number of steps independent of input size/values. E.g. finding sum of two numbers. |
| logn | Logarithmic | Algorithms in this category are very ef ficient e.g. binary search. |
| n | Linear | Algorithms that scan a list of size n, eg., sequential search, finding the max/min element in an array etc. |
| nlogn | nlogn | Many divide & conquer algorithms including mergersort quicksort fall into this class. |
| n2 | Quadratic | Characterizes with two embedded loops, mostly sorting and matrix operations. E.g. adding two square matrices, bubble sort. |
| n3 | Cubic | Efficiency of algorithms with three embedded loops. For example : matrix multiplication , Floyd Warshall's algorithms |
| 2n | Exponential | Algorithms that generate all subsets of an n-element set . |
| n! | factorial | Algorithms that generate all permutations of an n-element set e.g. Travelling Salesman problems |




Plot of function Values

Q4.Write the algorithm for the Towers of Hanoi problem. Explain the solution with 3 disks. Solve the recurrence relation M(n) = 2 M(n-1)+1 for all n > 1 , M(1)=1.

Sol :

In Towers of Hanoi problem We have n disks of different sizes that can slide onto any of three pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. We can move only one disk at a time, and it is forbidden to place a larger disk on top of asmaller one.

To move n>1 disks from peg 1 to peg 3 (with peg 2 as auxiliary), we first move recursively n − 1 disks from peg 1 to peg 2 (with peg 3 as auxiliary), then move the largest disk directly from peg 1 to peg 3, and, finally, move recursively n − 1 disks from peg 2 to peg 3 (using peg 1 as auxiliary). Of course, if n = 1, we simply move the single disk directly from the source peg to the destination peg.

**Algorithm Towers( n,L,M,R)**

//Input : No.of Disks n, three pegs L, M & R

```
//Output : the steps to move from L to  R
Begin
   If( n=1)
       Print( " Move disk from L to R")
  Else
      Towers( n-1,L,R,M)
       Print( " Move nth  disk from L to R")
     Towers( n-1,M,L,R)
End
```

**Analysis**

Let us apply the general plan outlined above to the Tower of Hanoi problem.
The number of disks $n$ is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation. Clearly, the number of moves $M(n)$ depends on $n$ only, and we get the following recurrence equation for it:
$M(n) = M(n-1) + 1 + M(n-1)$ for $n > 1$.
With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves $M(n)$:

$M(n) = 2M(n-1) + 1$ for $n > 1$, **(2.3)**
$M(1) = 1$.

We solve this recurrence by the same method of backward substitutions:

$M(n) = 2M(n-1) + 1$ sub. $M(n-1) = 2M(n-2) + 1$
$= 2[2M(n-2) + 1] + 1 = 22M(n-2) + 2 + 1$ sub. $M(n-2) = 2M(n-3) + 1$
$= 22[2M(n-3) + 1] + 2 + 1 = 23M(n-3) + 22 + 2 + 1$.

The pattern of the first three sums on the left suggests that the next one will be $24M(n-4) + 23 + 22 + 2 + 1$, and generally, after $i$ substitutions, we get
$M(n) = 2iM(n-i) + 2i-1 + 2i-2 + \ldots + 2 + 1 = 2iM(n-i) + 2i - 1$.
Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence (2.3):
$M(n) = 2n-1M(n-(n-1)) + 2n-1 - 1$
$= 2n-1M(1) + 2n-1 - 1 = 2n-1 + 2n-1 - 1 = 2n - 1$.

**Q5. Explain the methods to analyze recursive and non-recursive algorithms with examples. (10)**

**General Plan for Analyzing Efficiency of Nonrecursive Algorithms**

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in its innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation either find a closed-form formula for the count or, at the very least, establish its order of growth.

For example Consider the **element uniqueness problem:** check whether all the elements in a given array are distinct. This problem can be solved by the following straightforward algorithm.

```
ALGORITHM UniqueElements(A[0..n - 1])
    //Checks whether all the elements in a given array are distinct
    //Input: An array A[0..n - 1]
    //Output: Returns "true" if all the elements in A are distinct
    // and "false" otherwise.
    for i «— 0 to n — 2 do
       for j' <- i + 1 to n - 1 do
          if A[i] = A[j]
              return false
    return true
```

Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation. There are two kinds of worst-case inputs (inputs for which the algorithm does not exit the loop prematurely): arrays with no equal elements and arrays in which the last two elements are the only pair of equal elements. For such inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop's variable j between its limits i + 1 and n - 1; and this is repeated for each value of the outer loop, i.e., for each value of the loop's variable i between its limits 0 and n - 2. Accordingly, we get:

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - [(n-2)(n-1)]/2$$

$$= (n-1)^2 - [(n-2)(n-1)]/2 = [(n-1)n]/2 \approx \tfrac{1}{2} n^2 \in \Theta(n^2)$$

**<u>A General Plan for Analyzing Efficiency of Recursive Algorithms :</u>**

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or at least ascertain the order of growth of its solution.

For example: consider the recursive algorithm for finding factorial of a number

```
ALGORITHM F(n)
  // Computes n! recursively
  // Input: A nonnegative integer n
  // Output: The value of n!
  If n =0 return 1
  else return F(n — 1) * n
```

The basic operation is the multiplication which is performed once. There is one subproblem

generated which is of size n-1, where n is the size of the original problem. Thus if T(n) is the time to execute F(n) then the recurrence relation can be set up as

T(n) = T(n-1)+1, if, n>=1
1 , if n=0

Solving this through back substitution:
T(n) = T(n-1)+1 = T(n-2)+1+1= T(n-2)+2 = T(n-3)+1+2= T(n-3)+3 ….. T(n-i)+i

The argument n-i will become zero when n=i. Substituting this value in the equation above:
T(n) = T(0)+n=1+n (Since T(0) = )1

Thus T(n) = θ(n)

Q6. Explain Write the recursive algorithm and analysis of the problem to count the number of digits in the binary representation of a decimal number.
(10)

**Sol.**
The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

**ALGORITHM** *Binary(n)*
//Input: A positive decimal integer *n*
//Output: The number of binary digits in *n*'s binary representation
*count* ←1
**while** *n* > 1 **do**
*count* ←*count* + 1
*n*←_*n*/2_
**return** *count*

**Analysis**

First, notice that the most frequently executed operation here is not inside the while loop but rather the comparison n > 1 that determines whether the loop's body will be executed. Since the number of times the comparison will be executed is larger than the number of repetitions of the loop's body by exactly 1, the choice is not that important. A more significant feature of this example is the fact that the loop variable takes on only a few values between its lower and upper limits; therefore, we have to use an alternative way of computing the number of times the loop is executed. Since the value of n is about halved on each repetition of the loop, the answer should be about $\log_2 n$. The exact formula for the number of times the comparison n>1 will be executed is actually $[\log_2 n]+1$—the number of bits in the binary representation of n according to formula (2.1). We could also get this answer by applying the analysis technique based on recurrence relations; we discuss this technique in the next section because it is more pertinent to the analysis of recursive algorithms.

**Q 7** Write the algorithm and analysis of the element uniqueness problems. Explain with an example.

Sol.
Consider the element uniqueness problem: check whether all theelements in a given array of n elements are distinct. This problem can be solved by the following straightforward algorithm.

**ALGORITHM** UniqueElements(A[0..n − 1])
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
**for** *i* ←0 **to** *n* − 2 **do**
**for** *j* ←*i* + 1 **to** *n* − 1 **do**
**if** *A*[*i*]= *A*[*j* ] **return false**

**return true**

## Analysis

The natural measure of the input's size here is again n, the number of elements in the array. Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation. Note, however, that the number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only. By definition, the worst case input is an array for which the number of element comparisons Cworst(n) is the largest among all arrays of size n. An inspection of the innermost loop reveals that there are two kinds of worst-case inputs—inputs for which the algorithm does not exit the loop prematurely: arrays with no equal elements and arrays in which the last two elements are the only pair of equal elements. For such inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable j between its limits i + 1 and n − 1; this is repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and n − 2. Accordingly, we get

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] = \sum_{i=0}^{n-2}(n-1-i)$$

$$= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).$$

We also could have computed the sum $\sum_{i=0}^{n-2}(n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2},$$

where the last equality is obtained by applying summation formula (S2). Note that this result was perfectly predictable: in the worst case, the algorithm needs to compare all n(n − 1)/2 distinct pairs of its n elements.

Q8. Explain the various stages of the algorithm design and analysis process with the help of a flowchart.

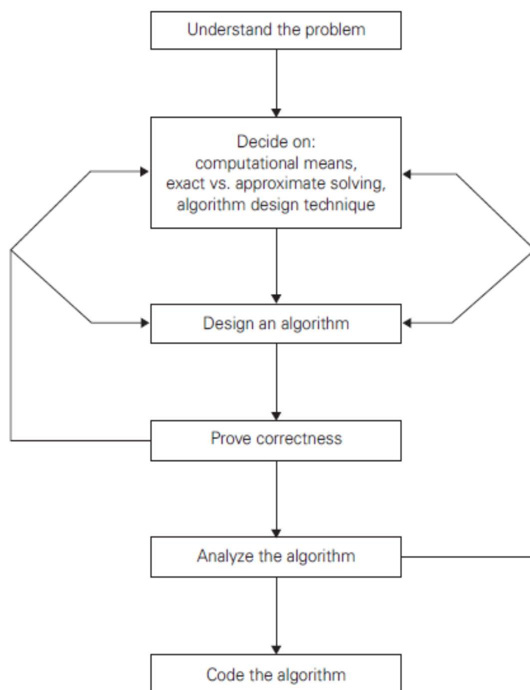



**Fig : Algorithm Design and Analysis Process**

1. **Understanding the problem:**

Before designing algorithm, one should understand the problem correctly. This may require

the problem to be read multiple times, asking questions if required and working out smaller instances of problem by hand. Any input to an algorithm specifies an instance or event of the problem. So, it is very important to set the range of inputs so that the algorithm works for all legitimate inputs i.e work correctly under all circumstances..

**2. Ascertaining the capabilities of a computational Device:**

After understanding the problem, one must think of the machines that execute instructions. The machines that are capable of executing the instructions one after the other is known as sequential machines and algorithms which run on these machines are known as sequential algorithms

Newer machines can run instructions concurrently re known as parallel machines and algorithms which have written for such machines are called parallel algorithms.

If we are dealing with the small problems, we need not worry about the time and memory requirements. But some complex problems which involve processing large amounts of data in real time are required to know about the time and memory requirements where the program is to be executed on the machine.

**3. Choosing between exact and approximate problem solving:**

The algorithms which solves the problem and gives the exact solution is known as Exact Algorithm and one which gives approximate results is known as Approximation Algorithms.

There are two situations in which we may have to go for approximate solution:

i) If the quantity to be computed cannot be calculated exactly. For example finding square roots, solving non linear equations etc.

ii) Complex algorithms may have solutions which take an unreasonably long amount of time if solved exactly. In such a case we may opt for going for a fast but approximate solution.

4. **Deciding on data structures:**

Algorithms use different data structures for their implementation. Some use simple ones but some other may require complex ones. But, Data structures play a vital role in designing and analyzing the algorithms.

**5. Algorithm Design Techniques:**

An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing. These techniques will provide guidance in designing algorithms for new problems. Various design methods for algorithms exist, some of which are – divide and conquer, dynamic programming, greedy algorithms etc.

**6. Methods of specifying an Algorithm:**

Algorithm can be specified using natural language and psuedocode. Due to the inherent ambiguity of the natural language, the most prevelant method of specifying an algorithm is using psuedocode.

**7. Proving an Algorithm's correctness:**

Correctness has to be proved for every algorithm. To prove that the algorithm gives the required result for every legitimate input in a finite amount of time. For some algorithms, a proof of correctness is quite easy; for others it can be quite complex. Mathematical Induction is normally used for proving algorithm correctness.

**8. Analyzing an algorithm:**

Any Algorithm must be analysed for its efficiency time and space . Time efficiency indicates how fast the algorithm runs; space efficiency indicates how much extra memory the algorithm needs. Another desirable characteristic is simplicity. A code which is simple reduces the effort in understanding and writing it and thus leads to less chances of error. Another desirable characteristic is generality. An algorithm can be general if it addresses a more general form of the problem for which the algorithm is to be designed and is able to handle all legitimate inputs.

**9. Coding an algorithm:**

Programming the algorithm by using some programming language. Formal verification by proof is done for small programs. Validity of large and complex programs is done through testing and debugging.

Q9. Write an algorithm for Bubble sort . Explain with an example and derive the time complexity

Sol :

Bubble sort is a sorting algorithm that works by repeatedly stepping through lists that need to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. This passing procedure is repeated until no swaps are required, indicating that the list is sorted.

The algorithm for bubble sort is as follows:

**ALGORITHM** *BubbleSort*$(A[0..n-1])$

//Sorts a given array by bubble sort

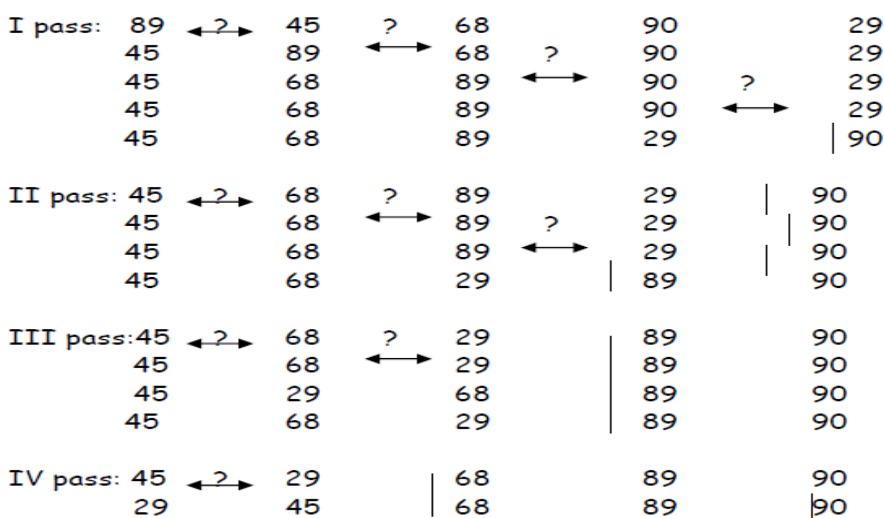
//Input: An array $A[0..n-1]$ of orderable elements

//Output: Array $A[0..n-1]$ sorted in nondecreasing order

**for** $i \leftarrow 0$ **to** $n-2$ **do**

  **for** $j \leftarrow 0$ **to** $n-2-i$ **do**

    **if** $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$




**Analysis:**

The no of key comparisons is the same for all arrays of size n, it is obtained by a sum which is similar to selection sort.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

$$= \sum_{i=0}^{n-2} [(n-2-i) - 0+1]$$

$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$= [n(n-1)]/2 \in \Theta(n^2)$$

The no.of key swaps depends on the input. The worst case is same as the no.of key comparisons.

$C(n) = [n(n-1)]/2 \in \Theta(n^2)$

Q10. Write an algorithm for Selection sort and derive the time complexity.

**Sol.**

Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

```
Algorithm selection sort (A[0...n-1])
// The algorithm sorts a given array
//Input: An array A[0..n-1] of orderable elements
// Output: Array A[0..n-1] sorted increasing order
for i← 0 to n-2 do
        min ← i
for j← i + 1 to n-1 do
        if A[j] < A[min] min← j
swap A[i] and A[min]
```

Analysis:

The input's size is the no of elements 'n" the algorithms basic operation is the key comparison A[j]<A[min]. The number of times it is executed depends on the array size and it is the summation:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i)$$

Either compute this sum by distributing the summation symbol or by immediately getting the sum of decreasing integers:, it is

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$= [n(n-1)]/2$$

Thus selection sort is a $\theta(n^2)$ algorithm on all inputs.