1. What is an algorithm? What are the characteristics of a good algorithm? Explain with example of GCD of two numbers.

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.

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

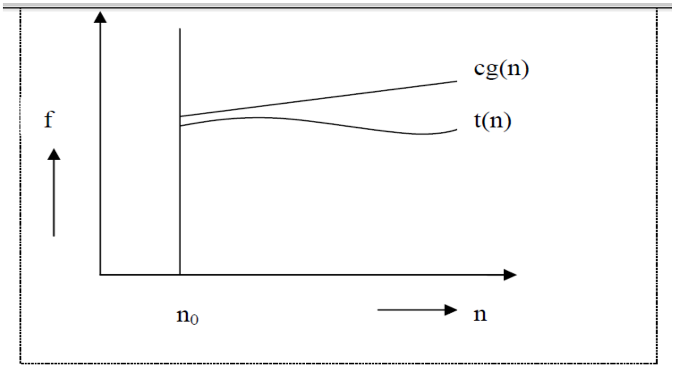
Different Notations

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

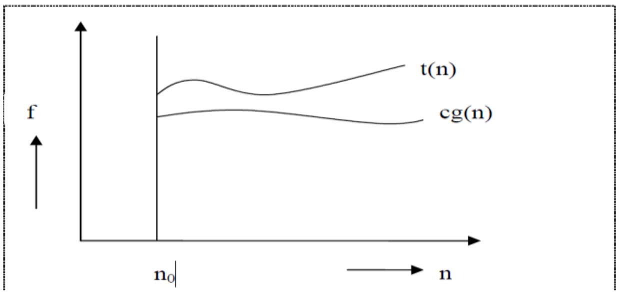
100n+5 < = 100n + n for all n>=5
< = 101 (n2)
Let g(n)= n2 ; n0=5 ; c = 101
i.e 100n+5 <=101 n2
t(n) <= c* g(n) for all n>=5
There fore , t(n) ∈ O(n2)



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) ≥ cg(n) for all n ≥ n0.

For example:
t(n) = n3 ∈ Ω(n2),
n3 ≥ n2 for all n ≥ n0.
we can select, g(n)= n3 , c=1 and n0=0
t(n) ∈ Ω(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) ≤ t(n) ≤ c1g(n) for all n ≥ n0.

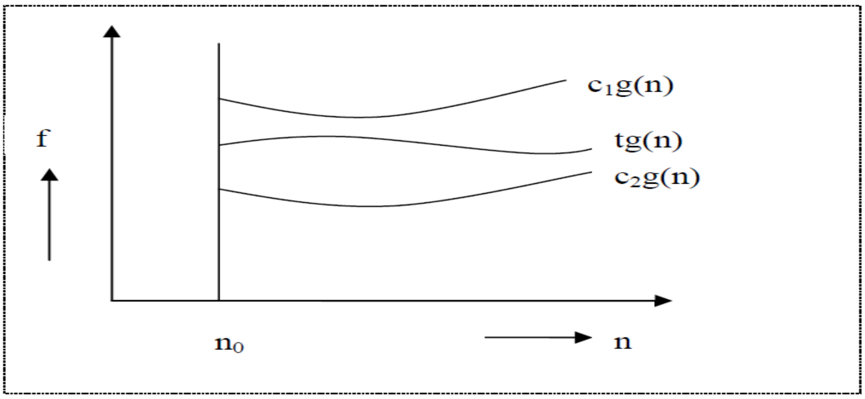
For example 1:

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) ∈ θ (n)

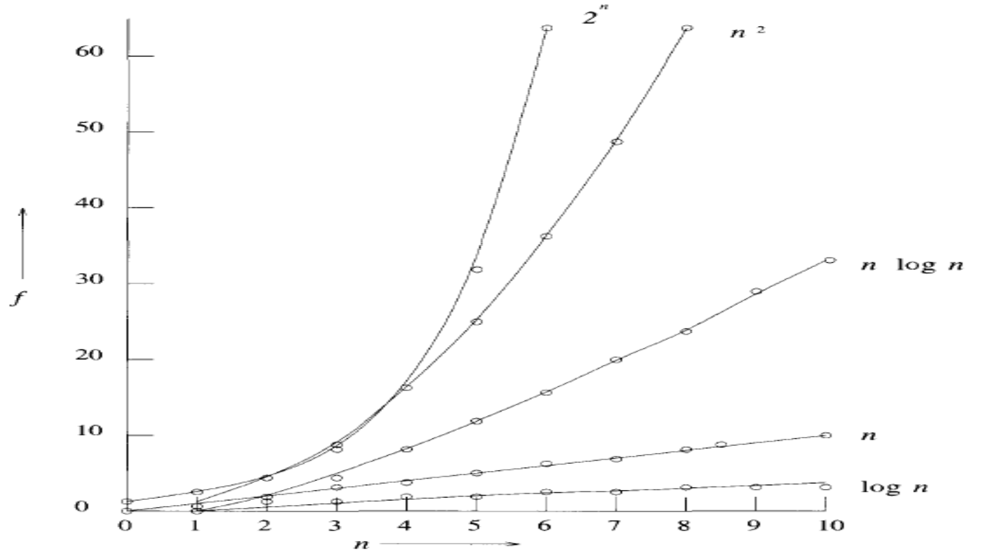


Describe various Basic Efficiency classes

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 i input size/values. E.g. finding sum of two number
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 the max/min element in an array etc.

nlogn	nlogn	Many divide & conquer algorithms including mergersort qui fall into this class.
n2	Quadratic	Characterizes with two embedded loops, mostly sorting and matrix operations. E.g. adding two square matrices, bubble sc
n3	Cubic	Efficiency of algorithms with three embedded loops. For exar 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 se Travelling Salesman problems



Plot of function Values

3. 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.

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 a smaller 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 + . . . + 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.

4. Explain the methods to analyze non-recursive algorithms with examples.

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:

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

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

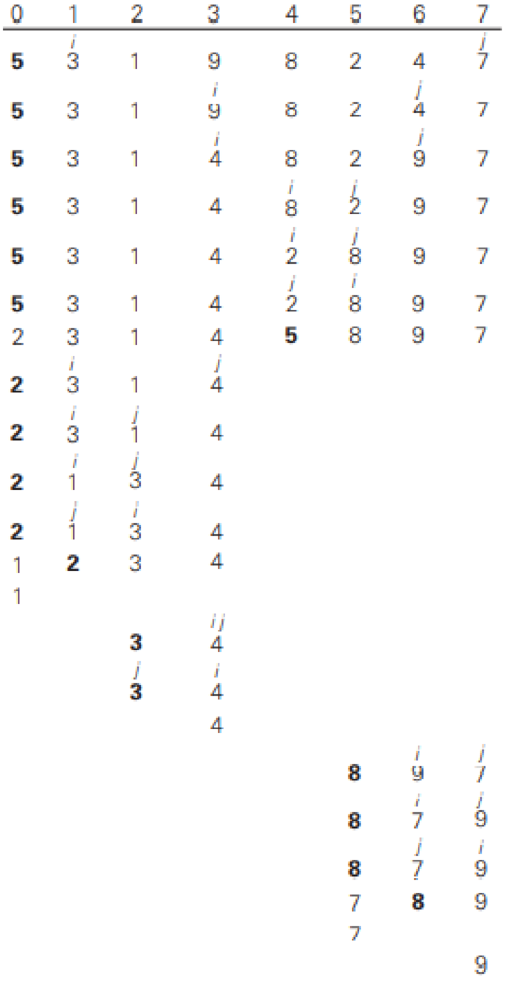
$$= (n-1)^2 - [(n-2)(n-1)]/2 = [(n-1)n]/2 \approx \tfrac{1}{2} n^2 \in \Theta(n^2)$$

5. Write an algorithm for Quick sort. Explain with an example and derive the time complexity.

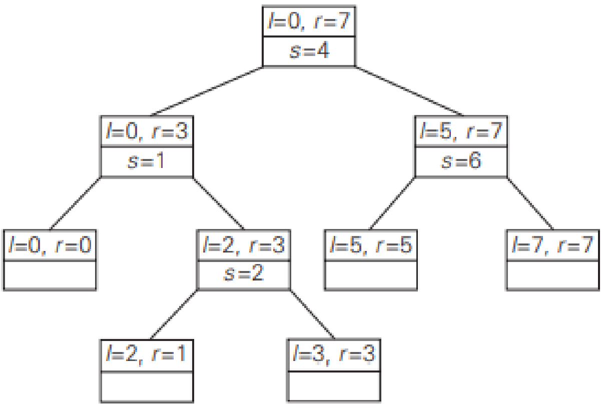
```
Algorithm Partition(A[l..r])
p← A[l]; i ← l; j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j ] ≤ p
    swap(A[i], A[j ])
until i ≥ j
```

```
swap(A[i], A[j ]) //undo last swap when i ≥ j
swap(A[l], A[j ])
return j
```

```
Algorithm Quicksort(A[l..r])
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right
// indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l<r
        s ←Partition(A[l..r]) //s is a split position
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])
```



0	1	2	3	4	5	6	7
5	$\overset{i}{3}$	1	9	8	2	4	$\overset{j}{7}$
5	3	1	$\overset{i}{9}$	8	2	$\overset{j}{4}$	7
5	3	1	$\overset{i}{4}$	8	2	$\overset{j}{9}$	7
5	3	1	4	$\overset{i}{8}$	$\overset{j}{2}$	9	7
5	3	1	4	$\overset{i}{2}$	$\overset{j}{8}$	9	7
5	3	1	4	$\overset{j}{2}$	$\overset{i}{8}$	9	7
2	3	1	4	**5**	8	9	7
2	$\overset{i}{3}$	1	$\overset{j}{4}$				
2	$\overset{i}{3}$	$\overset{j}{1}$	4				
2	$\overset{i}{1}$	$\overset{j}{3}$	4				
2	$\overset{j}{1}$	$\overset{i}{3}$	4				
1	**2**	3	4				
1							
		3	$\overset{i\,j}{4}$				
		$\overset{j}{\mathbf{3}}$	$\overset{i}{4}$				
			4				
					8	$\overset{i}{9}$	$\overset{j}{7}$
					8	$\overset{i}{7}$	$\overset{j}{9}$
					8	$\overset{j}{7}$	$\overset{i}{9}$
					7	**8**	9
					7		
							9



(b)

Let us assume we have an unsorted list of n numbers that partition from the middle every time. So, if we form a recursion tree, at each level, there will be n comparisons. Number of levels in the tree will be equal to the number of times n can be divided by 2 till the result is 1. Let us say n can be divided by 2 k times.

So, $n/2^k = 1$

$K=\log_2 n$ [log (base 2) n]

So, if there are log n levels and in each level there are n comparisons, the time taken is O(nlogn). This is the best-case time complexity of quicksort.

Now let us consider we have a sorted list on n numbers as input. Now, the partition will always happen from one side of the array. So, the recursion tree will grow only on one side for n levels and the number of comparisons will be n in first partition, (n-1) in the second and so on till 1 comparison in the last.
Thus, time taken is:
$n+(n-1)+(n-2)+\ldots+2+1$
$=n(n+1)/2$
$=O(n^2)$

This is the worst case time complexity of quicksort.

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

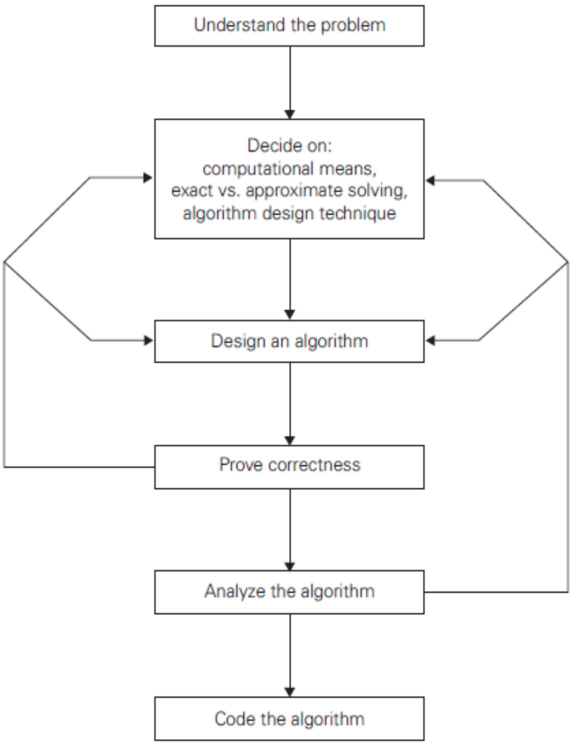


Fig : Algorithm Design and Analysis Process

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.

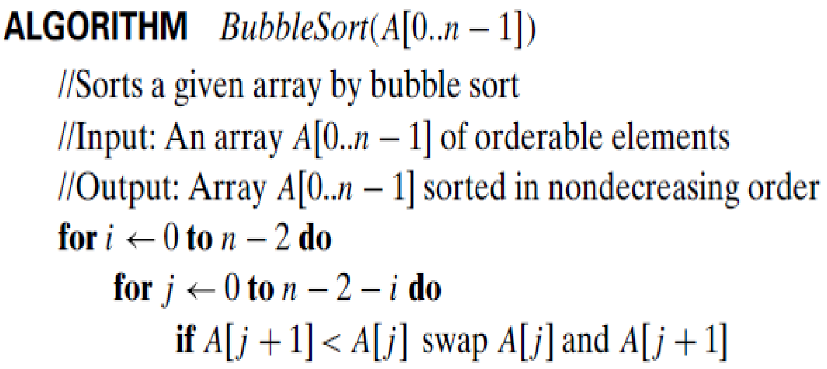
9. **Coding an algorithm:**

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.

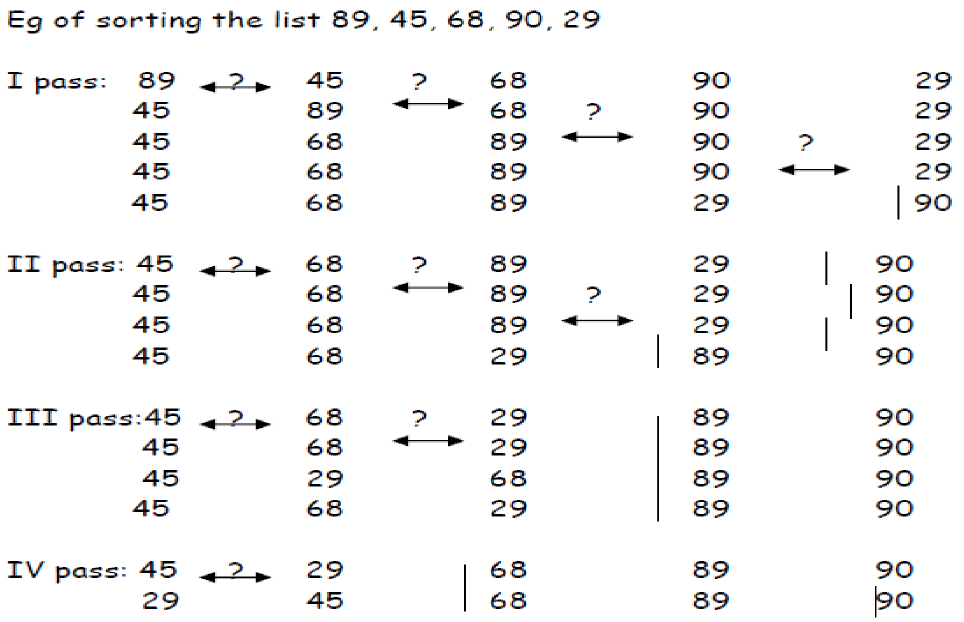
7. Write an algorithm for Bubble sort. Explain with an example and derive the time complexity

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 $A[j+1]$
```



```
Eg of sorting the list 89, 45, 68, 90, 29

I pass:  89  <-?->  45   ?    68         90          29
         45         89  <->   68    ?    90          29
         45         68        89   <->   90     ?    29
         45         68        89         90    <->   29
         45         68        89         29         | 90

II pass: 45  <-?->  68   ?    89         29    |   90
         45         68  <->   89    ?    29     |  90
         45         68        89   <->   29    |   90
         45         68        29       | 89        90

III pass:45  <-?->  68   ?    29       | 89        90
          45        68  <->   29       | 89        90
          45        29        68       | 89        90
         45         68        29       | 89        90

IV pass: 45  <-?->  29      | 68         89        90
          29        45      | 68         89       |90
```

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.

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

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

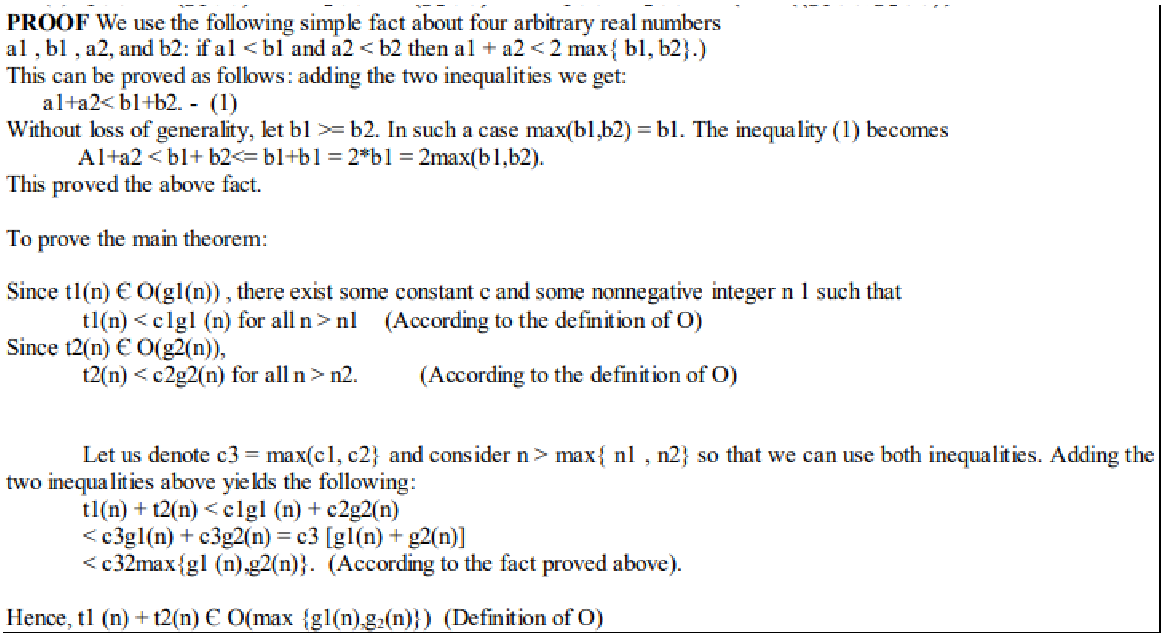
$$= \sum_{i=0}^{n-2} (n- 1-i)$$

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

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

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

8. If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then prove that:
$\boldsymbol{t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})}$



PROOF We use the following simple fact about four arbitrary real numbers
a1 , b1 , a2, and b2: if a1 < b1 and a2 < b2 then a1 + a2 < 2 max{ b1, b2}.)
This can be proved as follows: adding the two inequalities we get:
a1+a2< b1+b2. - (1)
Without loss of generality, let b1 >= b2. In such a case max(b1,b2) = b1. The inequality (1) becomes
A1+a2 < b1+ b2<= b1+b1 = 2*b1 = 2max(b1,b2).
This proved the above fact.

To prove the main theorem:

Since t1(n) Є O(g1(n)) , there exist some constant c and some nonnegative integer n 1 such that
t1(n) < c1g1 (n) for all n > n1 (According to the definition of O)
Since t2(n) Є O(g2(n)),
t2(n) < c2g2(n) for all n > n2. (According to the definition of O)

Let us denote c3 = max(c1, c2} and consider n > max{ n1 , n2} so that we can use both inequalities. Adding the two inequalities above yields the following:
t1(n) + t2(n) < c1g1 (n) + c2g2(n)
< c3g1(n) + c3g2(n) = c3 [g1(n) + g2(n)]
< c32max{g1 (n),g2(n)}. (According to the fact proved above).

Hence, t1 (n) + t2(n) Є O(max {g1(n),g_2(n)}) (Definition of O)

Sum of i=0 to n-2 (n-1-i)

(n-1-(n-2))+(n-1-(n-3))+(n-1-(n-4))+….+(n-1-0)
1+2+3+…+(n-1)
(n-1)n/2
(n^2-n)/2

Theta(n^2)

Number of elements*(upper bound_lower bound)/2