CMR

INSTITUTE OF

TECHNOLOGY

USN

| 1 | C | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|

CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

# Internal Assessment Test 1 – September 2018

| Sub: | Design and Analysis of Algorithms | | | | | | | Code: | 17MCA33 |
|---|---|---|---|---|---|---|---|---|---|
| Date: | 08-09-18 | Duration: | 90 mins | Max Marks: | 50 | Sem: | III | Branch: | MCA |

Note: Answer any full 5 questions. All questions carry equal marks. Total marks: 50

| | | Marks | OBE | |
|---|---|---|---|---|
| | | | CO | RBT |
| 1. | What is an algorithm? What are the characteristics of a good algorithm? Explain with example. (OR) | 10 | CO1 | L1 |
| 2. | Explain the fundamental data structures used for designing algorithms | 10 | CO1 | L1 |
| 3. | Describe the various asymptotic notations with a neat diagrams and examples. Describe various Basis Efficiency classes . (OR) | 10 | CO2 CO3 | L2 |
| 4. | Write the algorithm for the Tower 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. | 10 | CO2 CO3 | L2 L3 |
| 5. | Explain the methods to analyze recursive and non-recursive algorithms with examples. (OR) | 10 | CO2 CO3 | L2 |
| 6. | Explain and write algorithm for the brute force String Matching process and analyze it. | 10 | CO2 CO3 | L2 |
| 7 | a Describe the general method for divide and conquer.<br>b Write an algorithm for binary search and analyze its time complexity<br>(OR) | 3 7 | CO2 CO4 | L1 L2 |
| 8 | Write and explain the Quicksort algorithm using divide and conquer. Also analyze its best, worst and average case time efficiency using recurrence relations. | 10 | CO2 CO4 | L2 L3 |
| 9 | Write and explain the Mergesort algorithm using divide and conquer. Also analyze its worst case time efficiency using recurrence relations OR | 10 | CO2 CO4 | L2 L3 |
| 10 | Write an algorithm for Selection sort and derive the time complexity. | 10 | CO4 | L2 |
| | | | | |

CMR
INSTITUTE OF
TECHNOLOGY

**Internal Assessment Test 1 – September 2018**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Sub:** | **Design and Analysis of Algorithms** | | | | | | | **Code:** | **17MCA33** |
| *Date:* | *8-09-2018* | *Duration:* | *90 mins* | *Max Marks:* | *50* | *Sem:* | **IIIA** | *Branch:* | *MCA* |

***Dr. Vakula Rani***

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

**Q1(a) What is an algorithm? What are the characteristics of a good algorithm? Explain with Example** **(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.

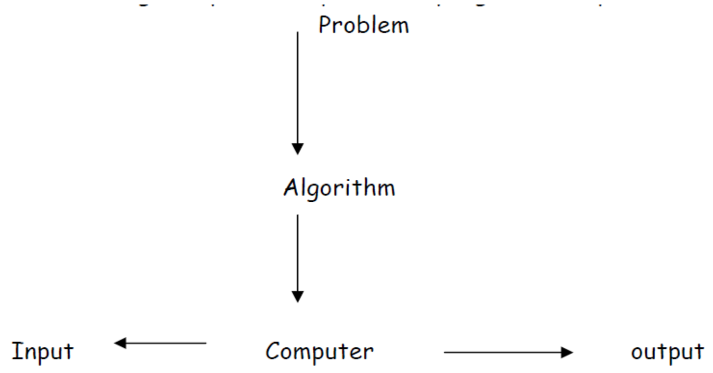


Figure : Notion of the Algorithm

**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

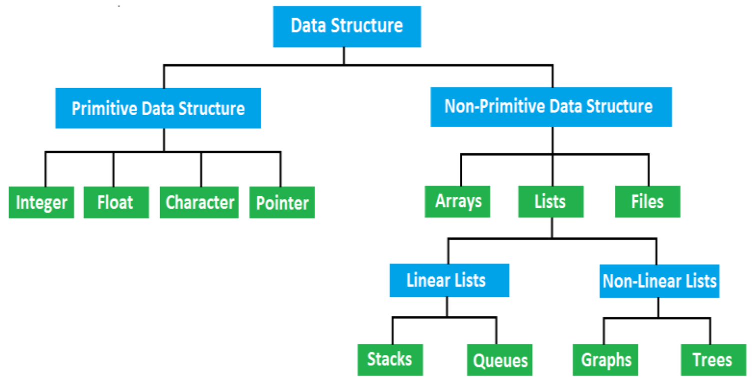
***Inputs / Ouput*** – The algorithm has two inputs and one output – gcd.

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

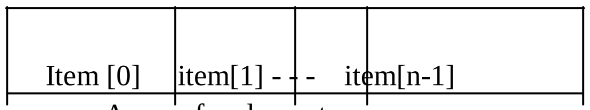
**( 2) Explain the fundamental data structures used for designing algorithms.**

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.

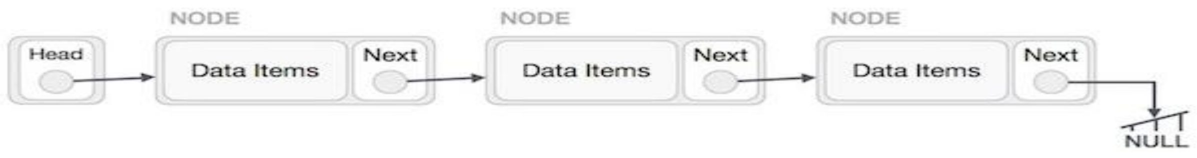


Array of n elements.

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.

Linked list is a linear data structure that contains sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called as "Node". It is a dynamic data structure , can grow or shrink



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.

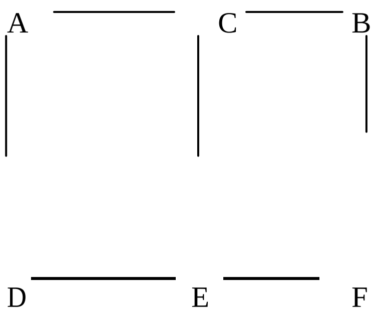
A queue, is a list for, which elements are deleted from one end of the structure, called the front (this operation is called dequeue), and new elements are added to the other end, called the rear (this operation is called enqueue). It operates in a first- in-first-out basis. Its having many applications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

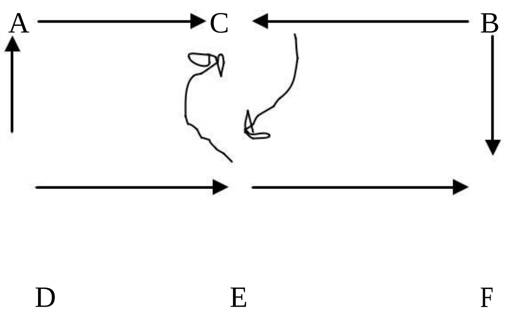
**Graphs**:

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



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.(a) 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

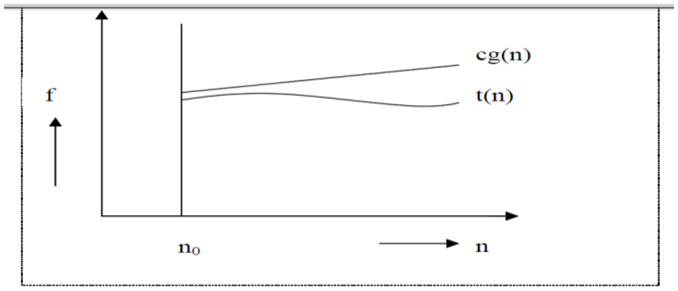
< = 101 (n2)

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.

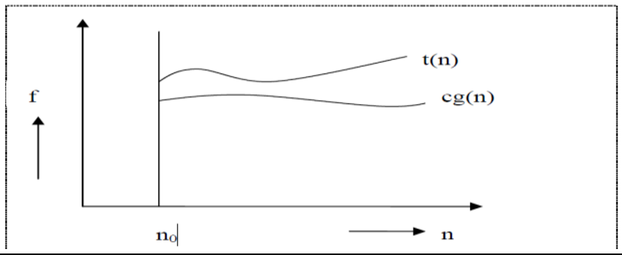
For example:

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.

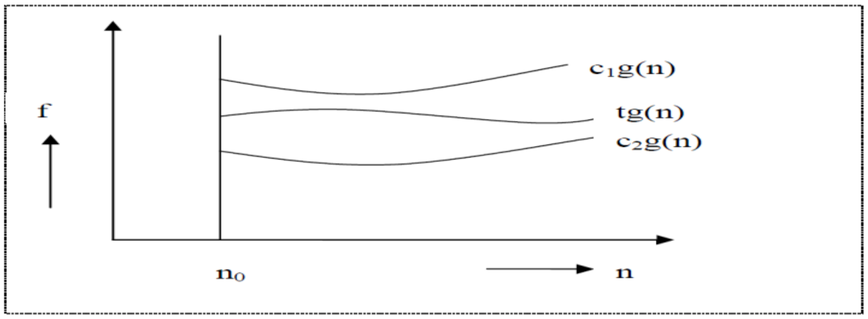
For example 1:

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 independent of input size/values. E.g. finding sum of two numbers. |
| logn | Logarithmic | Algorithms in this category are very efficient 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 |

**Q4.(a) 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
```

**<u>Analysis</u>**

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
$M(n) = 2iM(n-i) + 2i-1 + 2i-2 + \dots + 2 + 1 = 2iM(n-i) + 2i - 1$.
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)**

**<u>General Plan for Analyzing Efficiency of Nonrecursive Algorithms</u>**

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:

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

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

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

**A General Plan for Analyzing Efficiency of Recursive Algorithms :**

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 and write algorithm for the brute force string matching process and analyze it. (10)**

```
Algorithm Brute Force string match (T[0..,n-1], P[0..m-1])
// Input: An array T [0..n-1] of n chars, text
//          An array P [0..m-1] of m chars , a pattern.
// Output: The position of the first character in the text that starts the first
//          matching substring if the search is successful and -1 otherwise.

for i ← 0 to n-m do
        j ← 0
        while j < m and P[j] = T[i+j] do
                j ← j+1
        if j = m return i
return -1
```

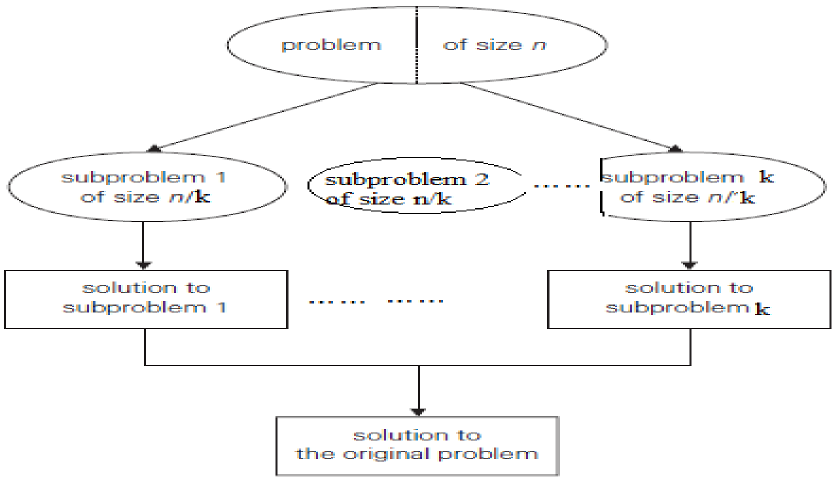
The time complexity would be analyzed by finding the number of times the basic operation j=j+1 is executed.
The inner loop will be executed a maximum of m times (j=0 to m-1).
Therefore

T(n)= $\sum_{i=0}^{n-m}\sum_{j=0}^{m-1} 1 = \sum_{i=0}^{n-m} m$ = (n-m)*m = θ(mn).

Where m is the length of pattern and n is the length of text.

**Q 7a** Describe the general method for divide and conquer.

Divide-and-conquer algorithms work according to the following general plan:
1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.
The general method is shown diagrammatically as below:



The pseudo code for the same is given by:
Algorithm DivideAndConquer(P,S)
    Divide the problem P into k subproblems P1,P2…Pk
    For each i in [1..k]

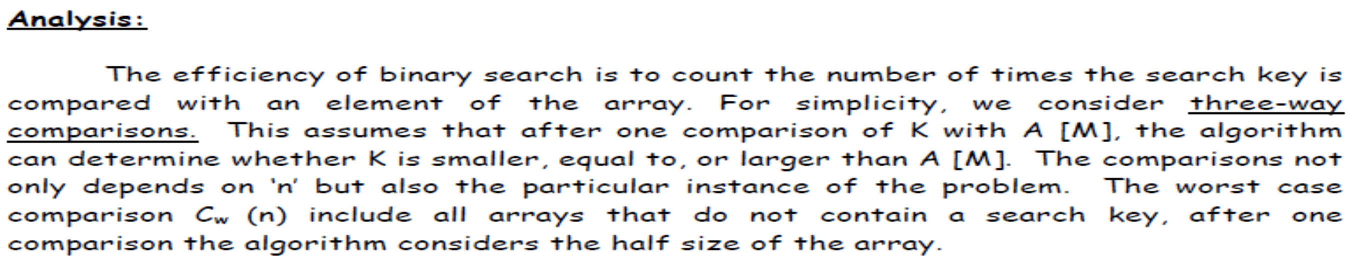
```
        / /solve each of the problem recursively by using the same technique
        Si← DivideAndConquer(Pi)
     Combine the solutions to the subproblems P1,P2… Pk i.e. S1, S2…, Sk to form the solution S
   Return S
```

**Q 7b Write an algorithm for binary search and analyze its time complexity**

This search algorithm works on the principle of divide and conquer. For this algorithm, the data should be in the sorted order. Binary search stars comparing the middle element with the key value. If it matches, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

**Algorithm binsearch(A[0..n-1],key,l,u)**

```
Begin
   If  l>u
       Return -1
   While (l <=u )
       Mid ← (l+u)/2
       If A[mid] = key
            Return mid
      Else if A[mid] < key
            Return  binsearch(A, l,mid-1)
      Else
            Return binsearch(A,mid+1,u)
end
```



**Analysis:**

The efficiency of binary search is to count the number of times the search key is compared with an element of the array. For simplicity, we consider three-way comparisons. This assumes that after one comparison of K with A [M], the algorithm can determine whether K is smaller, equal to, or larger than A [M]. The comparisons not only depends on 'n' but also the particular instance of the problem. The worst case comparison $C_w$ (n) include all arrays that do not contain a search key, after one comparison the algorithm considers the half size of the array.

Thus the recurrence would be C(n) = C(n/2)+1 , n>1 and C(1) = 0

The problem size at each step reduces by half each time. The additional amount in computing the mid element and comparison with the key is constant time operation and thus the 1 in the above expression.

Using master's method, we find a=1 , b=2 and d=0. The a=bd and thus it is the second case.

Hence C(n) = O(lgn)

**Q8. Write and explain the Quicksort algorithm using divide and conquer. Also analyze its best, worst and average case time efficiency using recurrence relations.**

QuickSort is a highly efficient sorting algorithm and it uses Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the pivot. Usually, pick first element as pivot. It partitions the large array of data into smaller arrays , one of which holds values smaller than the pivot value and the other holds values greater than the pivot value.

**Algorithm Quicksort(A, l,u)**

```
// sort only if there are more than two elements in the array
// Input: Array A[0..n-1] , l lower bound, u Upper bound
//Output : Sorted Array A
```

```
Begin
        If ( l < u )
          p<-- partition(A,l,u)
         Quicksort(A,l,p-1)
         Quicksort(A,p+1,u)
End
```

```
Algorithm partition(A,l,u)
Begin
        piv <-- A[l]
        i <-- l
        j <-- u +1

        // keep moving i and j till they meet
        repeat
              repeat   i <-- i +1;   until (A[i] >= piv)
               repeat  j <-- j-1 ;   until (A[j]  <= piv)
               if ( i < j)  swap(A[i],A[j])
        until (i>=j)

        swap(A[l],A[j])
  return j
End
```

**Analysis:**

Analyzing partition we notice that I and j start from the two ends of the array and for each iteration in the algorithm either I moves or j moves. For each move we can have maximum of one swap. Therefore the total number of operations in partition is O(n).

If we consider Quicksort on n elements, then after the partition if one partition has I elements then the other partition has n-i-1 elements(excluding the pivot element). The time taken for quicksort is therefore :

1. The time taken to partition (cn)
2. The time taken for doing quicksort of the first partition
3. The time taken for doing quicksort of the second partition

Thus if T(n) is the time taken by quicksort to sort n elements then
$T(n) = T(i) + T(n-i) + cn$

**Best Case:**

The best case for quicksort occurs when both the partitions are always equal . This happens mostly when the input array is random. In such a case the recurrence becomes
$T(n) = T(n/2)+T(n/2)+cn = 2T(n/2)+cn$
Applying master's method we find that $T(n) = \theta(n\lg n)$.

**Worst case:**

In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned. This unfortunate situation will happen for arrays sorted in increasing order. In such a case the recurrence would be
$T(n) = T(n-1)+T(0)+n$
Assuming $T(0) = 0$
$T(n) = T(n-1)+n$.
Using back substitution we find that $T(n) = T(n-1)+n = T(n-2)+n-1+n =$

T(n-3)+n-2+n-1+n

Expanding till ith step we find

T(n-i) + n-i+1 +….+n

The expansion ends when T(0) is reached. Assigning n-I = 0 => I = n. Substituting in the equation above:

$T(n) = T(0) + n-n+1+\ldots n = 1+2\ldots n = n(n+1)/2 = \theta(n^2)$ .

Thus the worst case performance of quicksort is $\theta(n^2)$

**Q9.Write and explain the Mergesort algorithm using divide and conquer. Also analyze its worst case time efficiency using recurrence relations.**

Mergesort is a perfect example of a successful application of the divide-and conquer technique. It sorts a given array A[0..n − 1] by dividing it into two halves A[0.._n/2_ − 1] and A[_n/2_..n − 1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.  The pseudocode for Merge sort is as follows:

**Algorithm merge(arr,l,mid, u)**

```
        Create a temporary array C[0..u]
        i<-- l
        j <-- mid+1
        k <-- l // index into temporary array
        while i <=mid and j <=u
                if arr[i] <= arr[j]
                        C[k] <-- arr[i]
                        i <-- i+1
                else
                        C[k] <-- arr[j]
                        j <-- j+1
        k <-- k+1
//copying rest of elements from first subarray
 while i<=mid
        C[k] <-- arr[i]
        i <-- i+1
        k <-- k+1
//copying rest of elements from second subarray
while j<=u
        C[k] <-- arr[j]
        j <-- j+1
        k <-- k+1
for i in l to u        // copying all elements from temp array to original array
arr[i] <-- C[i]
```

**Algorithm mergesort(arr,l,u**)

```
// only do it if the array contains atleast 2 elements
    if ( l < u  )
        mid = (l+u)/2
        mergesort(A,l,mid)
        mergesort(A,mid+1,u)
        Merge(A,l,mid,u)
```

**Analysis**

We first analyze the merge function used for mergesort. We notice that to merge an array with n elements at every step( in the first three loops) an element is always copied to the temporary array C. Since there are n elements to be copied the number of operations in the first three loops is n. Similarly in the last loop when the elements are copied from temporary array to the original array (arr) there are again “n” copies. Thus the total number of copy operations in the algorithm merge is O(n).

Analyzing the mergesort algorithm we find that each call involves two recursive calls to mergesort with the problem size half and a call to merge which takes O(n) time . Thus the recurrence can be wtitten as:

$$T(n) = 2\,T(n/2)+cn.$$

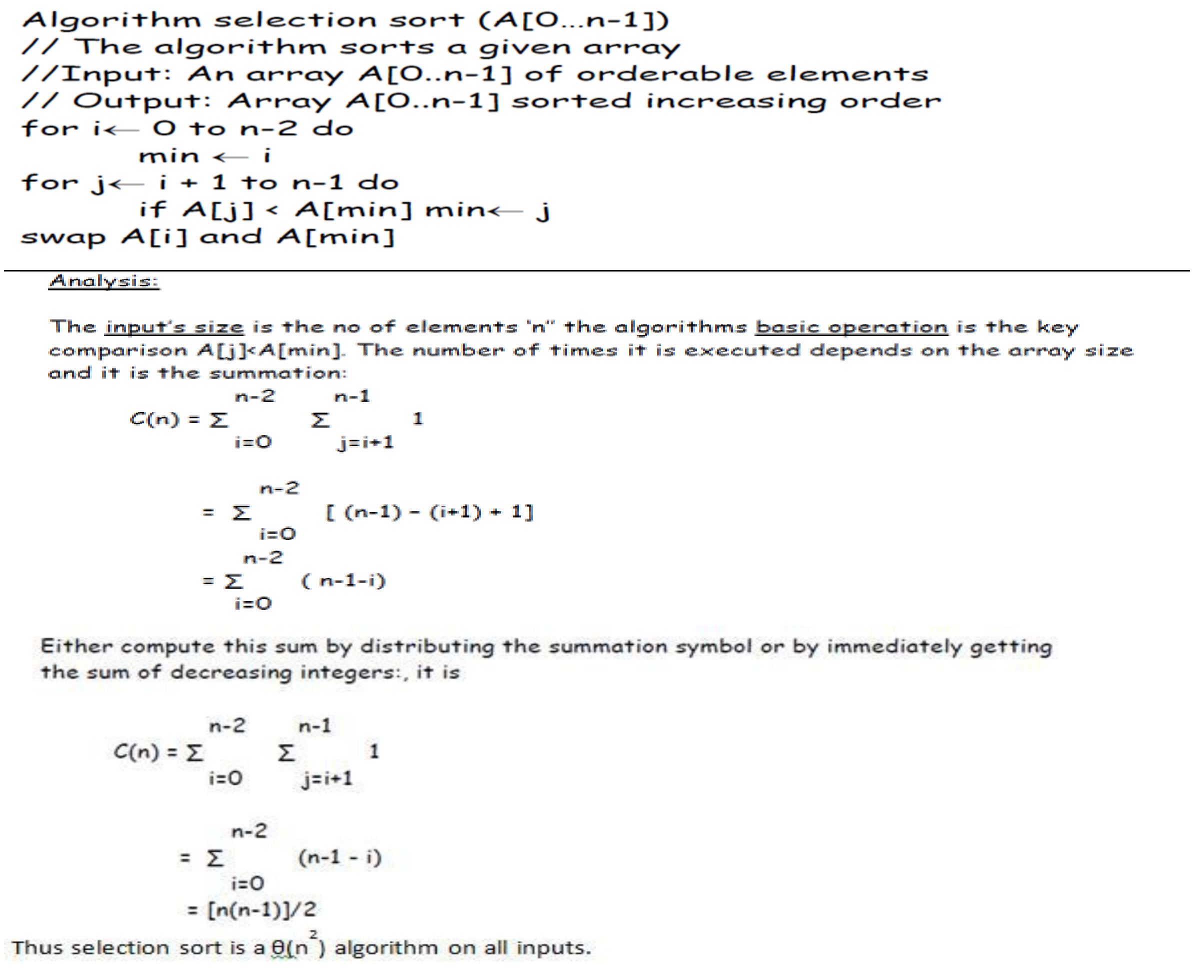
Applying the master's method,

a=2, b=2 and d=1.

Thus a=bd and thus case 2 of Master's method applies.

Thus T(n) = O(nlgn).

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



```
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.