CMR
INSTITUTE OF
TECHNOLOGY

USN

Internal Assessment Test 1 – March 2017

Sub:	Design and Analysis of Algorithms							Code:	13MCA41
Date:	*27-03-2017*	*Duration:*	*90 mins*	*Max Marks:*	*50*	*Sem:*	IV	*Branch:*	*MCA*

All Questions carry 10 marks each. Answer any five of the following **5 x 10 = 50 Marks**

Q1. Justify $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ $then$ $t_1(n) + t_2(n) \in O(\max((g_1(n), g_2(n)))$

Sol: Proof of a1 + a2 < 2 max{ b1, b2} - 4M
Main proof - 6M

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)

Q2. Explain the methods to analyze recursive and non-recursive algorithms with examples.

Sol: Non Recursive : General Method - 2M, Generic Pseudocode - 2M, Example - 1M
Recursive : General Method - 2M, Generic Pseudocode - 2M, Example - 1M

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:

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

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

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

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

$$T(n) = \begin{cases} T(n-1)+1, & \text{if, } n>=1 \\ 1 \quad , & \text{if } n=0 \end{cases}$$

Solving this through back substitution:

$T(n) = T(n-1)+1 = T(n-2)+1+1= T(n-2)+2 = T(n-3)+1+2= T(n-3)+3 \ldots.. T(n-i)+i$

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 $T(n) = \Theta(n)$

Q3. Explain and write algorithm for the brute force string matching process and analyze it.

Sol: Algorithm - 4M, Explanation - 2M, Analysis - 4M

```
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.

Q4. Describe an efficient method to multiply two nxn matrices. Analyze the method.

Sol: Explanation of method - 6M. Analysis - 4M

Strassen's matrix multiplication method is an efficient method for multiplying two matrices. Let A and B be two square matrices of dimension nxn. According to Strassen's formula's the product of two n x n matrixes are obtained as:

$[m_1 + m_4 - m_5 + m_7 \qquad m_3 + m_5 \qquad]$
$[m_2 + m_4 \qquad m_1 + m_3 - m_2 + m_6 \qquad]$

Where,

$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$
$m_2 = (a_{10} + a_{11})* b_{00}$
$m_3 = a_{00} * (b_{01} - b_{11})$
$m_4 = a_{11} * (b_{10} - b_{00})$
$m_5 = (a_{00} + a_{01}) * b_{11}$
$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$
$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

Thus, to multiply two 2-by-2 matrixes, Strassen's algorithm requires seven multiplications and 18 additions / subtractions, where as the brute-force algorithm requires eight multiplications and 4 additions. Let A and B be two n-by-n matrixes when n is a power of two. (If not, pad the rows and columns with zeroes). We can divide A, B and their product C into four n/2 by n/2 sub matrices as follows:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} a_{00} & a_{01} \\ \hline a_{10} & a_{11} \end{array}\right] * \left[\begin{array}{c|c} b_{00} & b_{01} \\ \hline b_{10} & b_{11} \end{array}\right]$$

Analysis:

The efficiency of this algorithm, M(n) is the number of multiplications in multiplying two n by n matrices according to Strassen's algorithm. The recurrence relation is as follows:

$M(n) = 7M(n/2)$ for $n>1$, $M(1) = 1$

Solving it by backward substitutions for $n=2^k$ yields.

$M(2^K) = 7M(2^{K-1}) = 7[7M(2^{K-2})] = 7^2 M(2^{K-2})$

$= \ldots.7^i M(2^{k-i}) = \ldots..7^K M(2^{K-K}) = 7^K$

Since $k = \log_2 n$,

$M(n) = 7^{\log_2 n}$

$= n^{\log_2 7}$

$\approx n^{2.807}$

which is smaller than n^3 required by Brute force algorithm.

Since this saving is obtained by increasing the number of additions, A (n) has to be checked for obtaining the number of additions. To multiply two matrixes of order n>1, the algorithm needs to multiply seven matrices of order n/2 and make 18 additions of matrices of size n/2; when n=1, no additions are made since two numbers are simply multiplied.

The recurrence relation is

$A(n) = 7A(n/2) + 18(n/2)^2$ for $n>1$

$A(1) = 0$

This can be deduced based on Master's Theorem, as $A(n) \in \Theta(n^{\log_2 7})$. In other words, the number of additions has the same order of growth as the number of multiplications. Thus in Strassen's algorithm it is $\Theta(n^{2.8})$, which is better than $\Theta(n^3)$ of brute force.

Q5. Write and explain the mergesort algorithm using divide and conquer. Also analyze its worst case time efficiency using recurrence relations.

Sol: Algorithm - 4M, Explanation - 2M Analysis - 4M

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

// copying all elements from temp array to original array
for i in l to u
          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 analyse the merge function used for mergesort. We notice that to merge an array with n elements at every step(in the first three loops) anelement 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 quicksort 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 $a=b^d$ and thus case 2 of Master's method applies. Thus T(n) = O(nlgn).

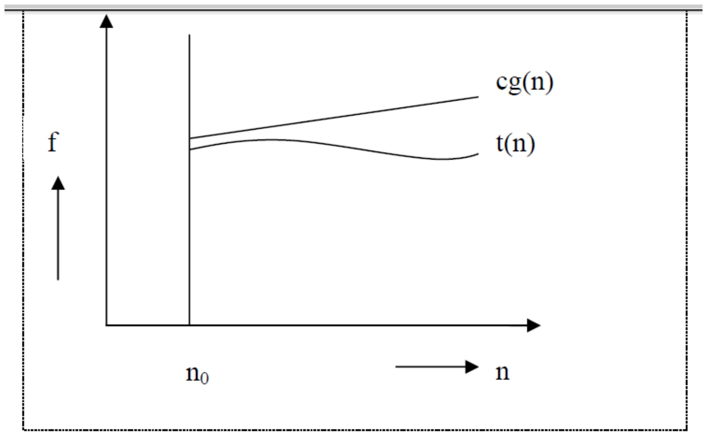
Q6. Explain the various asymptotic notations.

Sol: Definition of Theta, Omega and Big Oh - 1 M each = 3M

Graph of Theta - 2M, Omega and Big Oh - 1 M each = 4M
Examples of Theta, Omega and Big Oh - 1 M each = 3M

Sol: Definition: A function t(n) is said to be in O[g(n)]. Denoted 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.

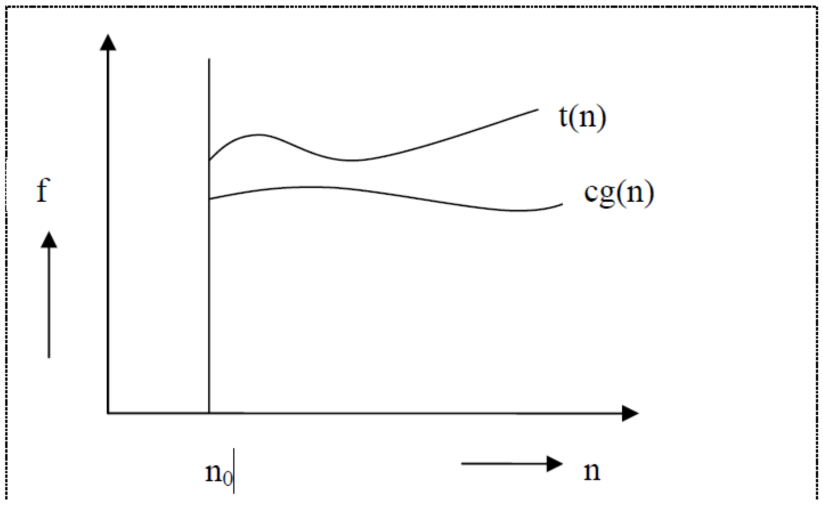
Eg. $100n+5 \in O(n^2)$



Ω -Notation:

Definition: A fn 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 s.t.
t(n) ≥ cg(n) for all n ≥ n0.

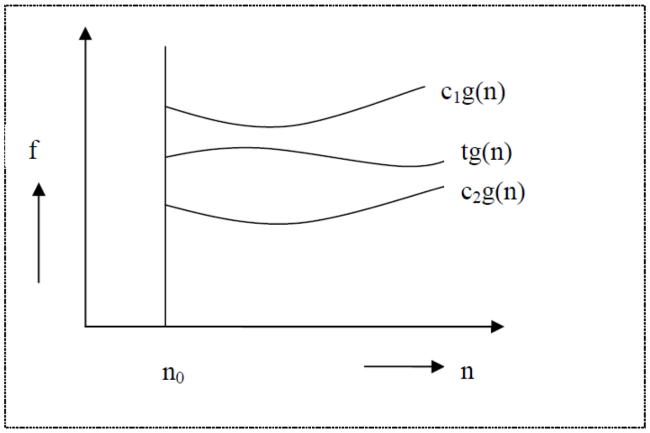
For example: $n^3 \in \Omega(n^2)$, Proof is $n^3 \geq n^2$ for all n ≥ n0. i.e., we can select c=1 and n0=0.



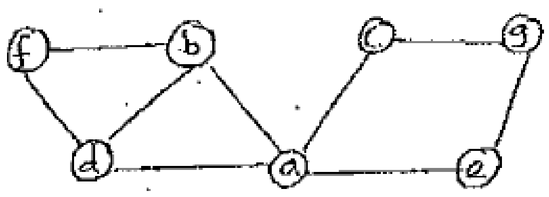
θ - Notation:

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: $n^2 \in \theta(n^2+4n+1)$, Proof is $7n^2 \geq n^2+4n+1$ for all n ≥ 0. i.e., we can select c=7 and n0=0.



Q7. Write Depth First Search algorithm. . Apply this algorithm on a given graph with C as the source node.



Sol: Explanation - 2M , Pseudocode - 4M, Example - 4M

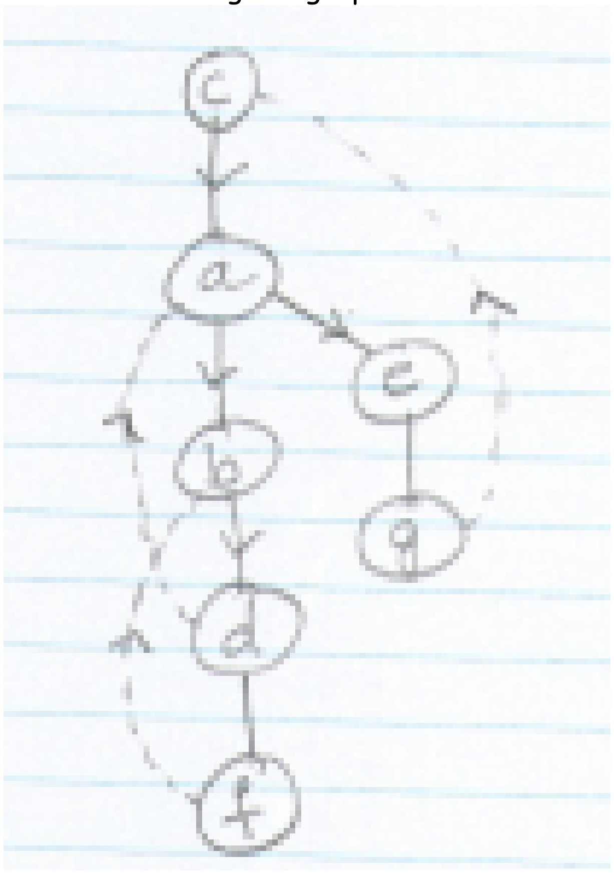
Depth-first search starts a graph's traversal at an arbitrary vertex by marking it as visited. On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. This process continues until a dead end—a vertex with no adjacent unvisited vertices— is encountered. At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there. The algorithm eventually halts after backing up to the starting vertex.

The algorithm for DFS is as follows:

```
ALGORITHM  DFS(G)
    //Implements a depth-first search traversal of a given graph
    //Input: Graph G = ⟨V, E⟩
    //Output: Graph G with its vertices marked with consecutive integers
    //          in the order they are first encountered by the DFS traversal
    for each u in V
        visit[u] <-- 0 // Mark each vertex unvisited
    for each vertex v in V do
        if v is marked with 0
            dfs(v)
```

```
ALGORITHM dfs(v)
    //visits recursively all the unvisited vertices connected to vertex v
    //by a path and numbers them in the order they are encountered
    //via global variable count
    visit[v] <-- 1     // mark v as visited
    for each vertex w in V adjacent to v do
        if w is marked with 0
            dfs(w)
```

The DFS of the given graph:



Q8. (a) Write the recurrence for quicksort if the array is divided into two parts 1/3 rd and 2/3 rd of the original size. (2)

Sol: Recurrence relation - 1M , Explanation - 1M

$T(n) = T(n/3)+T(2n/3)+n$.

Here $T(n/3)$ is the time to perform quicksort on the first part and $T(2n/3)$ is the time taken for quicksort on the second part. $O(n)$ is the time taken to perform partition.

$T(1)=1$, since an array of size one requires constant number of operations to sort.

(b) Consider the following recursive algorithm for computing the sum of first n cubes $S(n) = 1^3+2^3+...+n^3$

```
Algorithm S(n)
 if ( n== 1)
     return 1
```

```
  else
     return s(n-1)+n*n*n
  end of if
end
```

Set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed.

Sol: Recurrence relation - 1M , Solving - 3M

Let T(n) be the time taken for executing S(n) for input of size 'n'.

Then each non-base requires a constant number of operations and a recursive call with input size one less.
Therefore the recurrence is $T(n)= T(n-1)+1$
For input size 1 only constant number of operations are done . Hence $T(1) = 1$

Solving this recurrence relation:

$T(n) = T(n-1)+1 = T(n-2)+2....=T(n-i)+i$ --- (1)
The recursion stope when $n-i = 1$(base case) or $i = n-1$.
Substituting value of i in (1)
$T(n) = T(1)+n-1 = 1+n-1=n$

Thus the solution is $T(n) = n$.

(c) Solve the recurrence relation $T(n) = 2 T(n-1)+1.$, $T(0)=1$. (4)

Sol: Solution of recurrence relation - 4M

Using substitution method , expanding the recurrence:

$T(n) = 2T(n-1)+1 = 2[2T(n-2)+1]+1 = 2^2T(n-2)+2+1$

Expanding further:

$T(n) = 2^2[2T(n-3)+1] + 2 + 1 = 2^3T(n-3)+2^2+2+1$

Generalizing for i expansions :

$T(n) = 2^iT(n-i)+2^{i-1}+2^{i-2}+...+1 = 2^iT(n-i)+(2^i-1)/(2-1) = 2^iT(n-i)+2^i-1$ --- (1)

The expansion will continue till the base case is reached . Equating $n-i = 0$ (base case)

$i = n$. Substituting this value of i in Eq. (1) we get

$T(n) = 2^n T(0)+2^n-1=2^n.1+2^n-1=2^n+2^n-1 = 2.\ 2^n-1 = 2^{n+1}-1$

Thus the solution is $T(n) = 2^{n+1}-1$