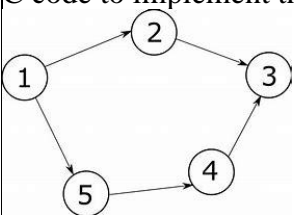


| | | | | | | | | | | | | | | | | | | | |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| | | | | | | | | | | | | | | | | | | | |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

Internal Assessment Test 1 – March 2024

| | | | | | | | | | |
|--------------|--|------------------|-----------------|-------------------|-----------|-------------|------------------|----------------|------------|
| Sub: | Design and Analysis of Algorithms | | | | | | Sub Code: | 22MCA15 | |
| Date: | 14.03.24 | Duration: | 90 min's | Max Marks: | 50 | Sem: | I | Branch: | MCA |

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

| PART I | | MARKS | OBE | |
|-----------------|--|---------|-----|-----|
| | | | CO | RBT |
| 1 | What is an algorithm? What are the characteristics of a good algorithm? Explain with example of GCD of two numbers. OR | [2+3+5] | CO1 | L1 |
| 2 | Describe the various asymptotic notations with a neat diagrams and examples. | [10] | CO2 | L1 |
| PART II | | [10] | | |
| 3 | Explain the methods to analyze non-recursive algorithms with examples. OR | | CO2 | L2 |
| 4 | Compare the order of growth of the following using limits: $\log_2 n$ and \sqrt{n} | [10] | CO2 | L4 |
| PART III | | | | |
| 5 | Write an algorithm for Quick sort. Explain with an example and derive the time complexity OR | [5+5] | CO3 | L3 |
| 6 | Write a recursive function to implement binary search. Take an example and compare linear and binary search to determine which one is better in terms of time taken. | [5+5] | CO3 | L3 |
| PART IV | | | | |
| 7 | Explain what is Divide and Conquer and what are its advantages and disadvantages. How is it different from Decrease and Conquer and Transform and Conquer OR | [4+6] | CO3 | L3 |
| 8 | Explain how Topological sort works considering the graph below as input. Write C code to implement the algorithm.  | [4+6] | CO3 | L3 |
| PART V | | | | |
| 9 | Take an example and explain how Heap sort works. Write C code to implement the algorithm. OR | [4+6] | CO3 | L3 |
| 10 | Explain Strassen's Matrix Multiplication algorithm and discuss how the algorithm follows divide and conquer | [6+4] | CO3 | L3 |

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 / Output - 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 $O[g(n)]$, $t(n) \in O[g(n)]$, 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 n_0 such that $t(n) \leq cg(n)$ for all $n \geq n_0$.

Eg. $t(n)=100n+5$ express in O notation

$$100n+5 \leq 100n + n \quad \text{for all } n \geq 5$$

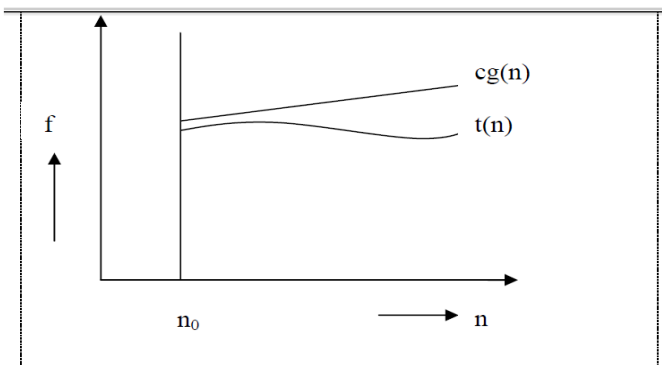
$$\leq 101 (n)$$

$$\text{Let } g(n)=n^2 \quad ; \quad n_0=5 \quad ; \quad c = 101$$

i.e $100n+5 \leq 101 n^2$

$$t(n) \leq c * g(n) \quad \text{for all } n \geq 5$$

There fore , $t(n) \in O(n^2)$



2. Omega(Ω) -Notation:

Definition: A function $t(n)$ is said to be in $\Omega[g(n)]$, denoted $t(n) \in \Omega[g(n)]$, 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 n_0 such that

$$t(n) \geq cg(n) \quad \text{for all } n \geq n_0.$$

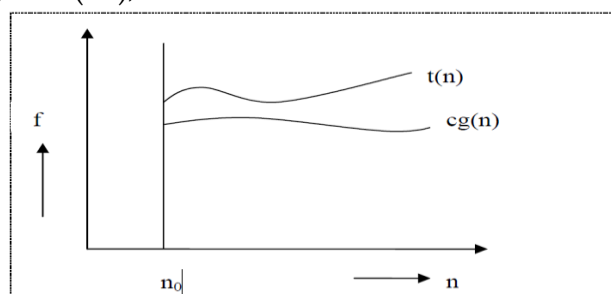
For example:

$$t(n) = n^3 \in \Omega(n^2),$$

$$n^3 \geq n^2 \quad \text{for all } n \geq n_0.$$

we can select, $g(n)=n^2$, $c=1$ and $n_0=0$

$$t(n) \in \Omega(n^2),$$



3. Theta (θ) - Notation:

Definition: A function $t(n)$ is said to be in $\theta [g(n)]$, denoted $t(n) \in \theta (g(n))$, 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 c_1 and c_2 and some nonnegative integer n_0 such that $c_2g(n) \leq t(n) \leq c_1g(n)$ for all $n \geq n_0$.

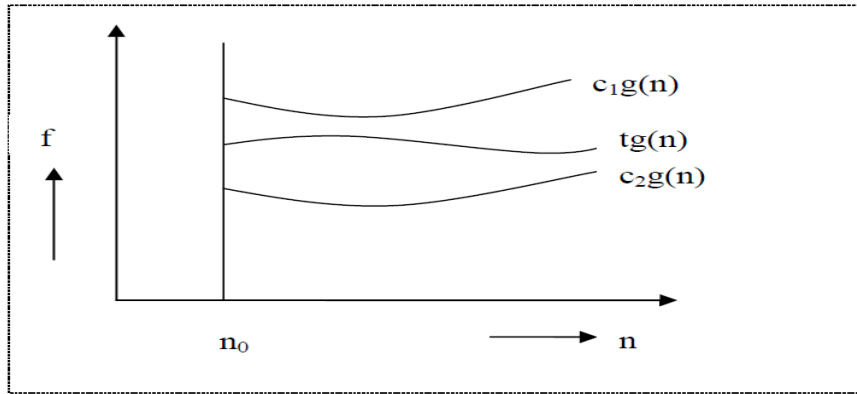
For example 1:

$t(n)=100n+5$ express in θ notation

$$100n \leq 100n+5 \leq 105n \quad \text{for all } n \geq 1$$

$$c_1=100; \quad c_2=105; \quad g(n) = n;$$

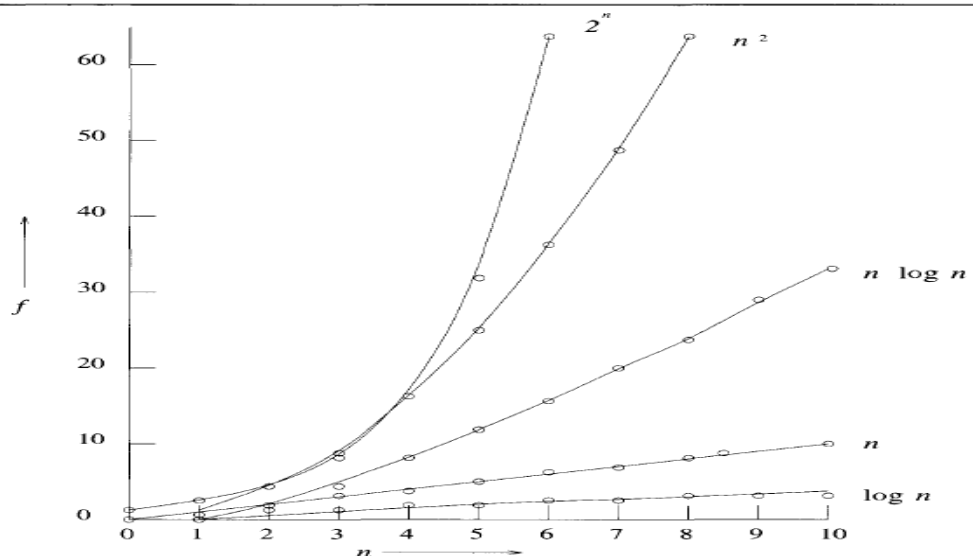
Therefore , $t(n) \in \theta (n)$



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 $n/1000$. 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 number |
| 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, the max/min element in an array etc. |
| nlogn | nlogn | Many divide & conquer algorithms including merge sort, fall into this class. |
| n ² | Quadratic | Characterizes with two embedded loops, mostly seen in matrix operations. E.g. adding two square matrices |
| n ³ | Cubic | Efficiency of algorithms with three embedded loops like matrix multiplication, Floyd Warshall's algorithms |
| 2 ⁿ | Exponential | Algorithms that generate all subsets of an n-element set |
| n! | factorial | Algorithms that generate all permutations of an n-element set, Travelling Salesman problems |



3. 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:

$$\begin{aligned}
 C_{\text{worst}}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) \\
 &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - [(n-2)(n-1)]/2 \\
 &= (n-1)^2 - [(n-2)(n-1)]/2 = [(n-1)n]/2 \approx \frac{1}{2} n^2 \in \Theta(n^2)
 \end{aligned}$$

4. Compare the order of growth of the following using limits: $\log_2 n$ and \sqrt{n}

EXAMPLE 2 Compare the orders of growth of $\log_2 n$ and \sqrt{n} . (Unlike Example 1, the answer here is not immediately obvious.)

$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n \rightarrow \infty} \frac{(\log_2 e) \frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2 \log_2 e \lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} = 0.$$

Since the limit is equal to zero, $\log_2 n$ has a smaller order of growth than \sqrt{n} . (Since $\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = 0$, we can use the so-called **little-oh notation**: $\log_2 n \in o(\sqrt{n})$. Unlike the big-Oh, the little-oh notation is rarely used in analysis of algorithms.)

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

Algorithm Partition(A[l..r])

$p \leftarrow A[l]$; $i \leftarrow l$; $j \leftarrow r + 1$

repeat

 repeat $i \leftarrow i + 1$ until $A[i] \geq p$

 repeat $j \leftarrow j - 1$ until $A[j] \leq p$

 swap(A[i], A[j])

until $i \geq j$

swap(A[i], A[j]) //undo last swap when $i \geq j$

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

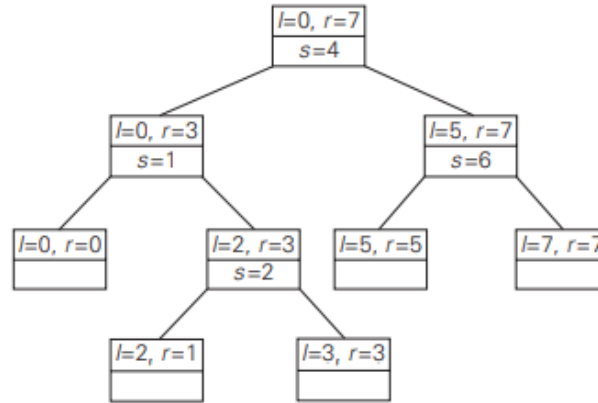
if $l < r$

$s \leftarrow$ 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 | <i>i</i> | 1 | 9 | 8 | 2 | 4 | <i>j</i> |
| 5 | 3 | 1 | <i>i</i> | 8 | 2 | <i>j</i> | 7 |
| 5 | 3 | 1 | <i>i</i> | 4 | 8 | 2 | <i>j</i> |
| 5 | 3 | 1 | 4 | <i>i</i> | <i>j</i> | 9 | 7 |
| 5 | 3 | 1 | 4 | 2 | 8 | 9 | 7 |
| 5 | 3 | 1 | 4 | <i>i</i> | <i>j</i> | 9 | 7 |
| 2 | 3 | 1 | 4 | 5 | 8 | 9 | 7 |
| 2 | <i>i</i> | 1 | <i>j</i> | 4 | | | |
| 2 | 3 | <i>j</i> | 4 | | | | |
| 2 | <i>i</i> | <i>j</i> | 4 | | | | |
| 2 | 1 | 3 | 4 | | | | |
| 2 | <i>j</i> | <i>i</i> | 4 | | | | |
| 1 | 2 | 3 | 4 | | | | |
| 1 | | | | | | | |
| | | 3 | <i>ij</i> | 4 | | | |
| | | <i>j</i> | <i>i</i> | 4 | | | |
| | | | 4 | | | | |
| | | | | 8 | <i>i</i> | <i>j</i> | |
| | | | | 8 | <i>i</i> | <i>j</i> | |
| | | | | 8 | <i>j</i> | <i>i</i> | |
| | | | | 7 | 8 | 9 | |
| | | | | 7 | | 9 | |
| | | | | | | 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.

$$\text{So, } n/2^k = 1$$

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

So, if there are $\log n$ levels and in each level there are n comparisons, the time taken is $O(n \log n)$.

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:

$$n + (n-1) + (n-2) + \dots + 2 + 1$$

$$= n(n+1)/2$$

$$= O(n^2)$$

This is the worst case time complexity of quicksort.

6. Write a recursive function to implement binary search. Take an example and compare linear and binary search to determine which one is better in terms of time taken.

Recursive function pseudo code for recursive binary search:

function binary_search(array, target, low, high):

if low > high:

```

return NotFound
else:
    mid = (low + high) / 2
    if array[mid] == target:
        return mid
    else if array[mid] > target:
        return binary_search(array, target, low, mid - 1)
    else:
        return binary_search(array, target, mid + 1, high)

```

Let's assume we have the following array: `arr = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]`

And we want to search for the element 10.

Linear Search:

Start from the beginning of the array.

Compare each element with the target until we find the element or reach the end of the array.

Dry run for linear search:

Compare 2 with 10: Not equal

Compare 4 with 10: Not equal

Compare 6 with 10: Not equal

Compare 8 with 10: Not equal

Compare 10 with 10: Found at index 4

Binary Search:

Start with the middle element.

If the middle element is the target, return its index.

If the middle element is smaller than the target, search in the right half of the array.

If the middle element is larger than the target, search in the left half of the array.

Dry run for binary search:

Start with the middle element 10.

10 is equal to the target, so return its index.

In this example, both linear and binary search found the target element. However, for small arrays like this, the difference in time complexity might not be apparent. Binary search shines when the array size is large because of its logarithmic time complexity.

To truly compare the time taken, you would need to perform these searches on much larger arrays and measure the time it takes for each. Typically, binary search will be faster for large arrays due to its $O(\log n)$ time complexity, compared to linear search's $O(n)$ time complexity.

7. Explain what is Divide and Conquer and what are its advantages and disadvantages. How is it different from Decrease and Conquer and Transform and Conquer

Divide and Conquer is a problem-solving technique that involves breaking down a problem into smaller, more manageable subproblems, solving these subproblems independently, and then combining their solutions to solve the original problem. It consists of three main steps:

Divide: Break the problem into smaller, more manageable subproblems that are similar to the original problem.

Conquer: Solve each subproblem recursively.

Combine: Combine the solutions of the subproblems to obtain the solution for the original problem.

Advantages of Divide and Conquer:

Efficiency: Divide and Conquer algorithms often have better time complexity compared to naive approaches.

Parallelism: The subproblems in Divide and Conquer can often be solved independently, making it suitable for parallel computing.

Simplicity: It simplifies complex problems by breaking them down into smaller, more manageable parts.

Modularity: Each subproblem can be solved independently, allowing for easier debugging and maintenance.

Disadvantages of Divide and Conquer:

Overhead: There can be overhead associated with dividing the problem, solving subproblems, and combining solutions, which may make the approach less efficient for very small problem sizes.

Memory Usage: Recursive implementations may require additional memory due to function calls and maintaining the call stack.

Complexity: Developing a Divide and Conquer algorithm may require a deep understanding of the problem and its substructures.

Difference between Decrease and Conquer and Transform and Conquer:

Decrease and Conquer:

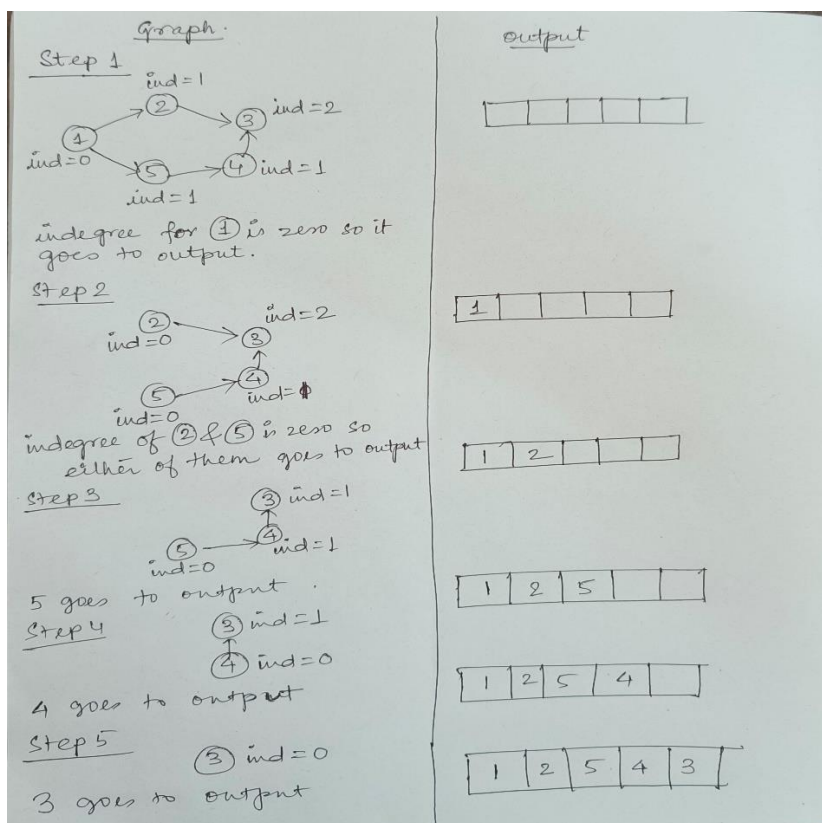
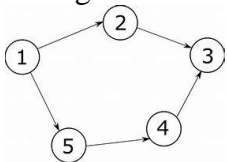
Decrease and Conquer is a problem-solving technique where the problem is reduced to a smaller instance of the same problem in such a way that the smaller instance is easier to solve than the original problem.

This reduction is usually achieved by transforming the problem into an instance of the same problem with a smaller input size. Examples of Decrease and Conquer include Binary Search and Quicksort.

Transform and Conquer:

Transform and Conquer is a problem-solving technique where the problem is transformed into a different problem for which an existing efficient solution is available. The transformed problem is usually simpler or easier to solve than the original problem. This technique involves changing the representation or structure of the problem to make it more amenable to efficient solutions. Examples of Transform and Conquer include Radix Sort and the Fast Fourier Transform (FFT).

8. Explain how Topological sort works considering the graph below as input. Write C code to implement the algorithm.



```

#include<stdio.h>
int main()
{
    int a[20][20],visit[20],ind, n,i,j,flag=0,count=0;
    printf("Enter the value of n\n");
    scanf("%d",&n);
    printf("Enter the adjacency matrix\n");
    for(i=0;i<n;i++)
    {
        visit[i]=0;
        for(j=0;j<n;j++)
            scanf("%d",&a[i][j]);
    }
    while(flag==0)
    {
        flag=1;
        for(i=0;i<n;i++)
        {
            if(visit[i]==0)
            {
                ind=0;
                for(j=0;j<n;j++)
                {
                    if(!(visit[j]==1 || a[j][i]==0))
                    {
                        ind=1;
                        break;
                    }
                }
                if(ind==0)
                {
                    //printf("%s",count==0 ?" \n topological ordering is" : " ");
                    visit[i]=1;
                    printf("%d\t",i+1);
                    flag=0;
                    count++;
                    break;
                }
            }
        }
    }

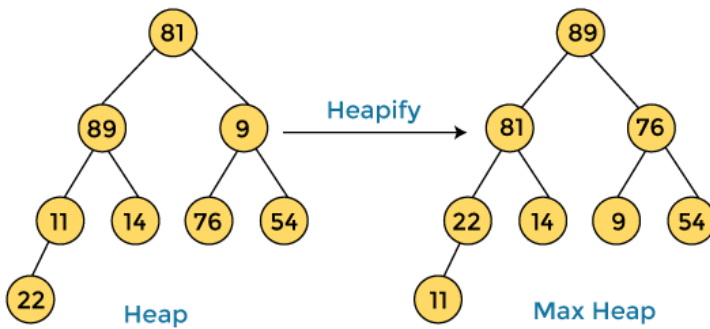
    if(count!=n)
    {
        printf("topological order is not possible");
    }
}

```

9. Take an example and explain how Heap sort works. Write C code to implement the algorithm.

| | | | | | | | |
|----|----|---|----|----|----|----|----|
| 81 | 89 | 9 | 11 | 14 | 76 | 54 | 22 |
|----|----|---|----|----|----|----|----|

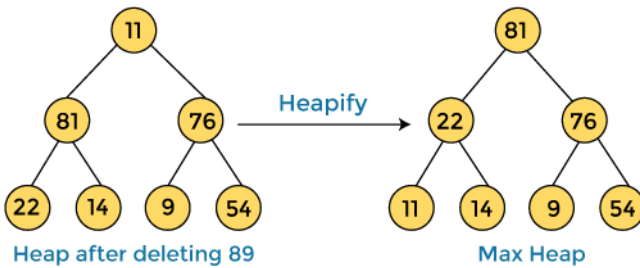
First, we have to construct a heap from the given array and convert it into max heap.



After converting the given heap into max heap, the array elements are -

| | | | | | | | |
|----|----|----|----|----|---|----|----|
| 89 | 81 | 76 | 22 | 14 | 9 | 54 | 11 |
|----|----|----|----|----|---|----|----|

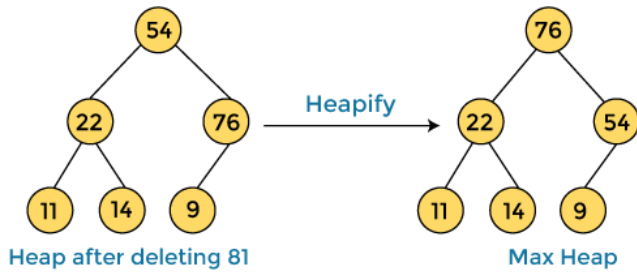
Next, we have to delete the root element (**89**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**11**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **89** with **11**, and converting the heap into max-heap, the elements of array are -

| | | | | | | | |
|----|----|----|----|----|---|----|----|
| 81 | 22 | 76 | 11 | 14 | 9 | 54 | 89 |
|----|----|----|----|----|---|----|----|

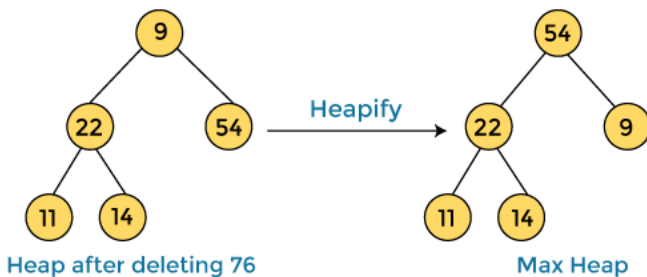
In the next step, again, we have to delete the root element (**81**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**54**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -

| | | | | | | | |
|----|----|----|----|----|---|----|----|
| 76 | 22 | 54 | 11 | 14 | 9 | 81 | 89 |
|----|----|----|----|----|---|----|----|

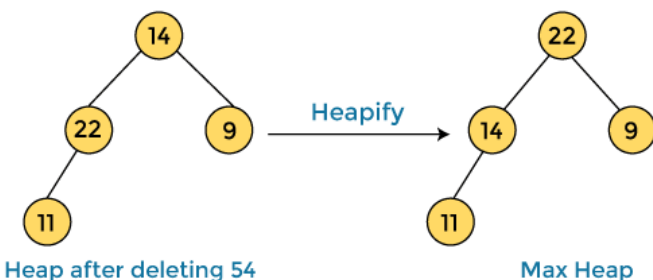
In the next step, we have to delete the root element (**76**) from the max heap again. To delete this node, we have to swap it with the last node, i.e. (**9**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -

| | | | | | | | |
|----|----|---|----|----|----|----|----|
| 54 | 22 | 9 | 11 | 14 | 76 | 81 | 89 |
|----|----|---|----|----|----|----|----|

In the next step, again we have to delete the root element (**54**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**14**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -

| | | | | | | | |
|----|----|---|----|----|----|----|----|
| 22 | 14 | 9 | 11 | 54 | 76 | 81 | 89 |
|----|----|---|----|----|----|----|----|

In the next step, again we have to delete the root element (**22**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**11**). After deleting the root element, we again have to heapify it to convert it into max heap.



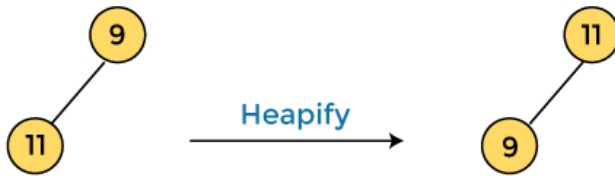
Heap after deleting 22

Max Heap

After swapping the array element 22 with 11 and converting the heap into max-heap, the elements of array are -

| | | | | | | | |
|----|----|---|----|----|----|----|----|
| 14 | 11 | 9 | 22 | 54 | 76 | 81 | 89 |
|----|----|---|----|----|----|----|----|

In the next step, again we have to delete the root element (14) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.



Heap after deleting 14

Max Heap

After swapping the array element 14 with 9 and converting the heap into max-heap, the elements of array are -

| | | | | | | | |
|----|---|----|----|----|----|----|----|
| 11 | 9 | 14 | 22 | 54 | 76 | 81 | 89 |
|----|---|----|----|----|----|----|----|

In the next step, again we have to delete the root element (11) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.



Heap after deleting 11

Max Heap

After swapping the array element 11 with 9, the elements of array are -

| | | | | | | | |
|---|----|----|----|----|----|----|----|
| 9 | 11 | 14 | 22 | 54 | 76 | 81 | 89 |
|---|----|----|----|----|----|----|----|

Now, heap has only one element left. After deleting it, heap will be empty.



After completion of sorting, the array elements are -

| | | | | | | | |
|---|----|----|----|----|----|----|----|
| 9 | 11 | 14 | 22 | 54 | 76 | 81 | 89 |
|---|----|----|----|----|----|----|----|

Now, the array is completely sorted.

```

#include <stdio.h>
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of heap. */
void heapify(int a[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child
    // If left child is larger than root
    if (left < n && a[left] > a[largest])
        largest = left;
    // If right child is larger than root
    if (right < n && a[right] > a[largest])
        largest = right;
    // If root is not largest
    if (largest != i) {
        // swap a[i] with a[largest]
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;

        heapify(a, n, largest);
    }
}
/*Function to implement the heap sort*/
void heapSort(int a[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(a, n, i);
    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        /* Move current root element to end*/
        // swap a[0] with a[i]
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;

        heapify(a, i, 0);
    }
}
/* function to print the array elements */

```

```

void printArr(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
    {
        printf("%d", arr[i]);
        printf(" ");
    }

}

int main()
{
    int a[] = {48, 10, 23, 43, 28, 26, 1};
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    heapSort(a, n);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
    return 0;
}

```

10. Explain Strassen's Matrix Multiplication algorithm and discuss how the algorithm follows divide and conquer

Matrix multiplication is based on a divide and conquer-based approach. Here we divide our matrix into a smaller square matrix, solve that smaller square matrix and merge into larger results. For larger matrices this approach will continue until we recurse all the smaller sub matrices.

Suppose we have two matrices, A and B, and we want to multiply them to form a new matrix, C.

$C=AB$, where all A,B,C are square matrices. We will divide these larger matrices into smaller sub matrices $n/2$; this will go on.

$$\begin{array}{c} \left[\begin{array}{cc|c} a & b & \\ \hline c & d & \end{array} \right] \times \left[\begin{array}{cc|c} e & f & \\ \hline g & h & \end{array} \right] = \left[\begin{array}{cc|c} ae + bg & af + bh & \\ \hline ce + dg & cf + dh & \end{array} \right] \\ A \qquad B \qquad C \end{array}$$

Now from above we see:

$$r=ae+bg$$

$$s=af+bh$$

$$t=ce+dg$$

$$u=cf+dh$$

Each of the above four equations satisfies two multiplications of $n/2 \times n/2$ matrices and addition of their $n/2 \times n/2$ products. Using these equations to define a divide and conquer strategy we can get the relation among them as:

$$T(N) = 8T(N/2) + O(N^2)$$

From the above we see that simple matrix multiplication takes eight recursion calls.

$$T(n) = O(n^3)$$

Thus, this method is faster than the ordinary one.

It takes only seven recursive calls, multiplication of $n/2 \times n/2$ matrices and $O(n^2)$ scalar additions and subtractions, giving the below recurrence relations.

$$T(N) = 7T(N/2) + O(N^2)$$

Steps of Strassen's matrix multiplication:

1. Divide the matrices A and B into smaller submatrices of the size $n/2 \times n/2$.
2. Using the formula of scalar additions and subtractions compute smaller matrices of size $n/2$.
3. Recursively compute the seven matrix products $P_i = A_i B_i$ for $i=1,2,\dots,7$.
4. Now compute the r,s,t,u submatrices by just adding the scalars obtained from above points.

Submatrix Products:

We have read many times how two matrices are multiplied. We do not exactly know why we take the row of one matrix A and column of the other matrix and multiply each by the below formula.

$$P_i = A_i B_i$$

$$= (\alpha_1 i_a + \alpha_2 i_b + \alpha_3 i_c)(\beta_1 i_e + \beta_2 i_f + \beta_3 i_g)$$

Where a,b , β , α are the coefficients of the matrix that we see here, the product is obtained by just adding and subtracting the scalar.

$$\begin{aligned}
 p_1 &= a(f - h) & p_2 &= (a + b)h \\
 p_3 &= (c + d)e & p_4 &= d(g - e) \\
 p_5 &= (a + d)(e + h) & p_6 &= (b - d)(g + h) \\
 p_7 &= (a - c)(e + f)
 \end{aligned}$$

The $A \times B$ can be calculated using above seven multiplications.

Following are values of four sub-matrices of result C

$$\begin{array}{c}
 \left[\begin{array}{c|c} a & b \\ \hline c & d \end{array} \right] \times \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array} \right] = \left[\begin{array}{c|c} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ \hline p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{array} \right] \\
 \text{A} & \qquad \qquad \text{B} & \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{C}
 \end{array}$$