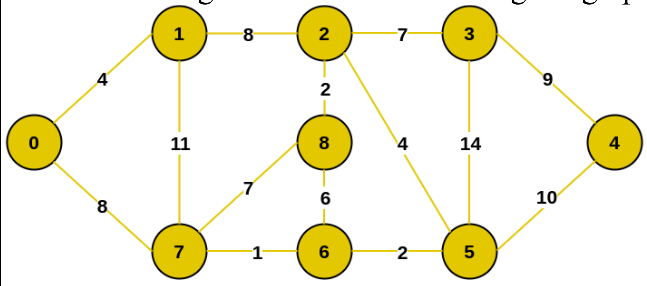
CMR
INSTITUTE OF
TECHNOLOGY

USN

Internal Assessment Test 1 – March 2023

Sub:	**Design and Analysis of Algorithms**					**Sub Code:**			**22MCA15**
Date:	**15.03.23**	**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
3	**PART II** Explain the methods to analyze non-recursive algorithms with examples. **OR**	[10]	CO2	L2
4	Show that if t1(n) ∈ O(g1(n)) and t2(n) ∈ O(g2(n)), then t1(n) + t2(n) ∈ O(max{g1(n), g2(n)}).	[10]	CO2	L4
5	**PART III** Write an algorithm for Quick sort. Explain with an example and derive the time complexity **OR**	[5+5]	CO3	L3
6	Write an algorithm to implement Brute Force's string-matching process and apply the same for the given input. Text String = [Hello, How Are You?] Pattern String=[How]	[5+5]	CO3	L3
7	**PART IV** Write Prim's algorithm and trace it using the graph below as input  0 1 2 3 4 5 6 7 8 4 8 7 9 2 11 4 14 8 7 6 10 1 2 **OR**	[4+6]	CO3	L3
8	Take an example and explain how Topological sort works. Write C code to implement the algorithm.	[4+6]	CO3	L3
9	**PART V** 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 / 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

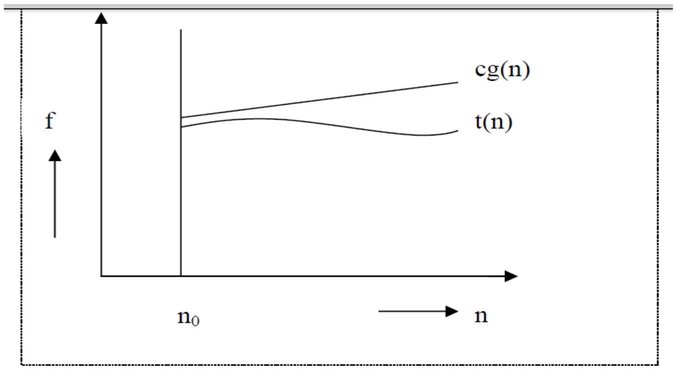
< = 101 (n2)

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.

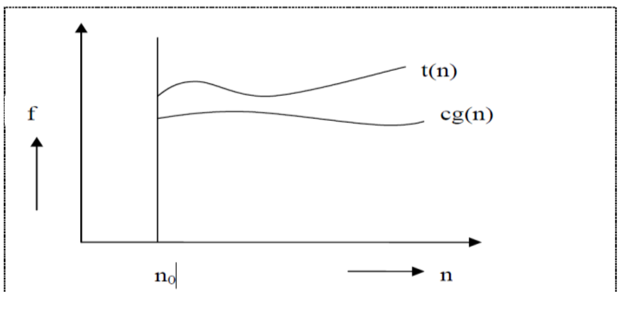
For example:

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.

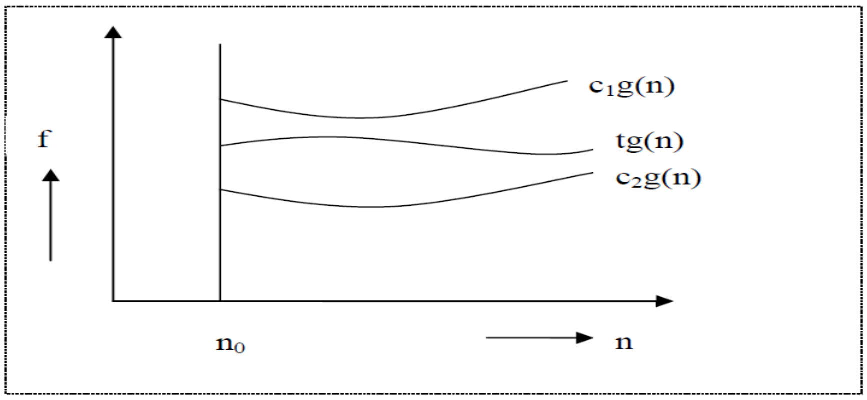
For example 1:

t(n)=100n+5 express in θ notation

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

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

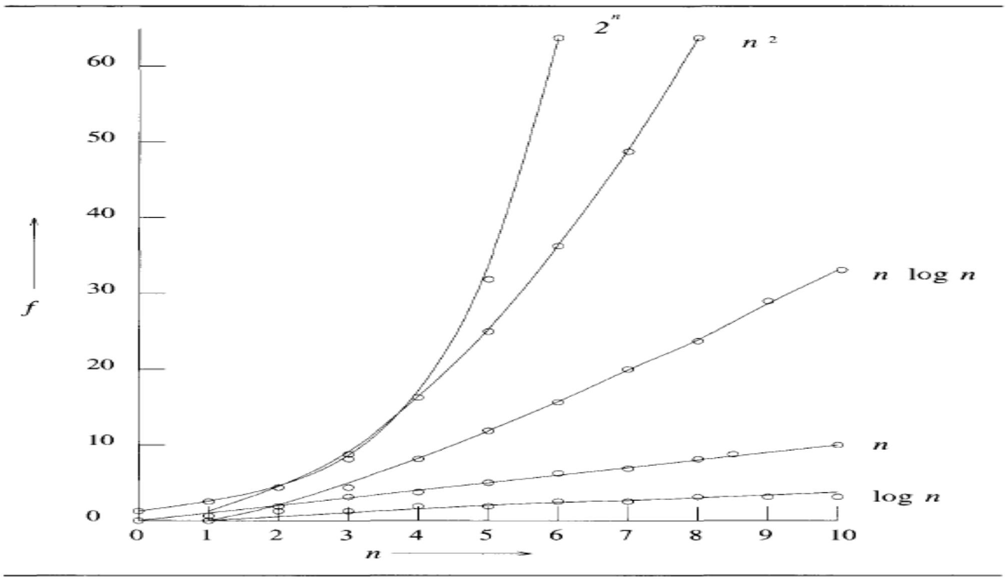
Therefore , t(n) ∊ θ (n)



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 mer fall into this class.
n2	Quadratic	Characterizes with two embedded loops, mostly s matrix operations. E.g. adding two square matrices
n3	Cubic	Efficiency of algorithms with three embedded loop matrix multiplication , Floyd Warshall's algorithms
2n	Exponential	Algorithms that generate all subsets of an n-eleme
n!	factorial	Algorithms that generate all permutations of an n- Travelling Salesman problems



Plot of function Values

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
```

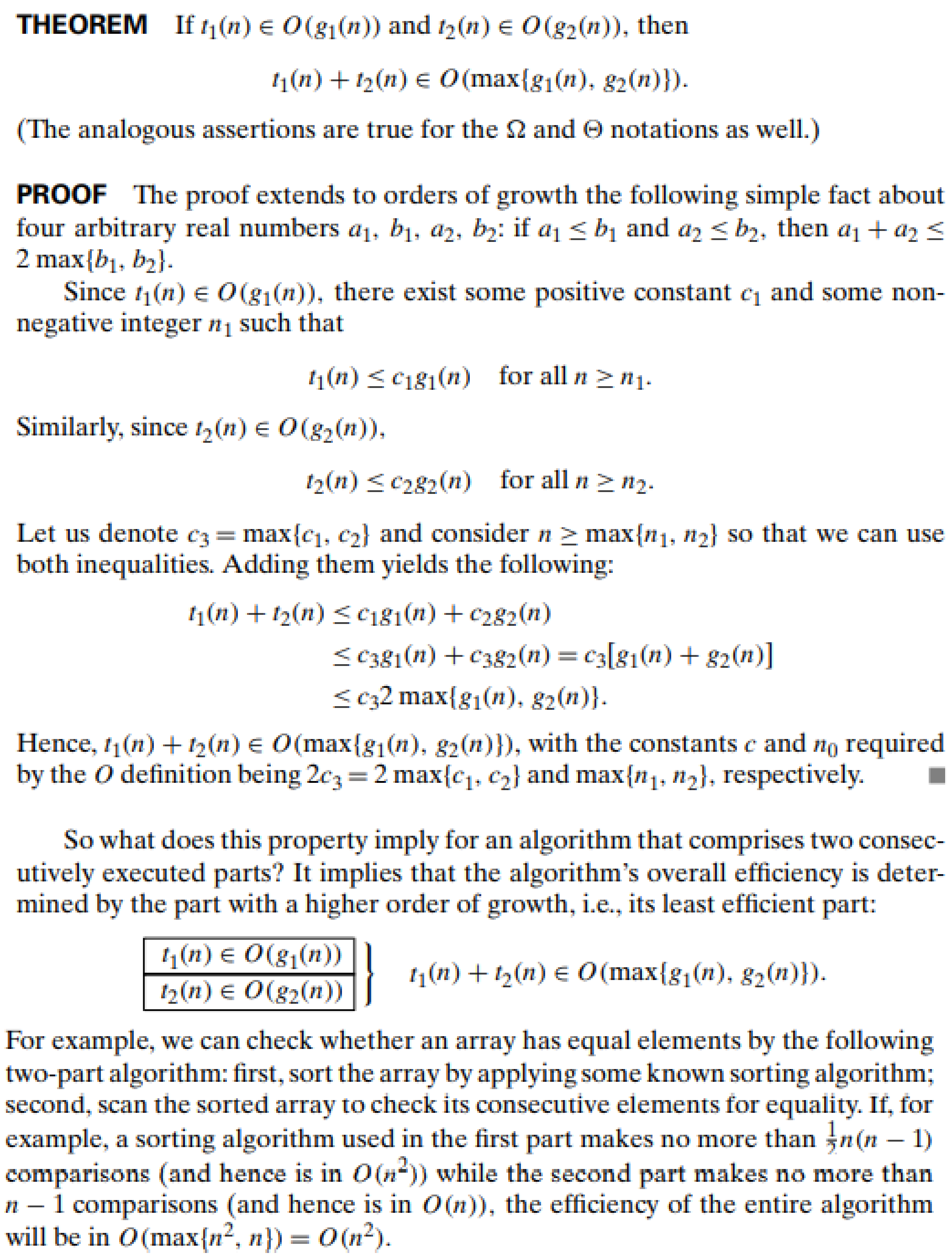
Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation. There are two kinds of worst-case inputs (inputs for which the algorithm does not exit the loop prematurely): arrays with no equal elements and arrays in which 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:

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

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

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

4. Show that if t1(n) ∈ O(g1(n)) and t2(n) ∈ O(g2(n)), then t1(n) + t2(n) ∈ O(max{g1(n), g2(n)}).



THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

(The analogous assertions are true for the Ω and Θ notations as well.)

PROOF The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2\max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \ge \max\{n_1, n_2\}$ so that we can use both inequalities. Adding them yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3 2\max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2\max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively. ■

So what does this property imply for an algorithm that comprises two consecutively executed parts? It implies that the algorithm's overall efficiency is determined by the part with a higher order of growth, i.e., its least efficient part:

$$\left.\begin{array}{|c|}\hline t_1(n) \in O(g_1(n)) \\ \hline t_2(n) \in O(g_2(n)) \\ \hline \end{array}\right\} \quad t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

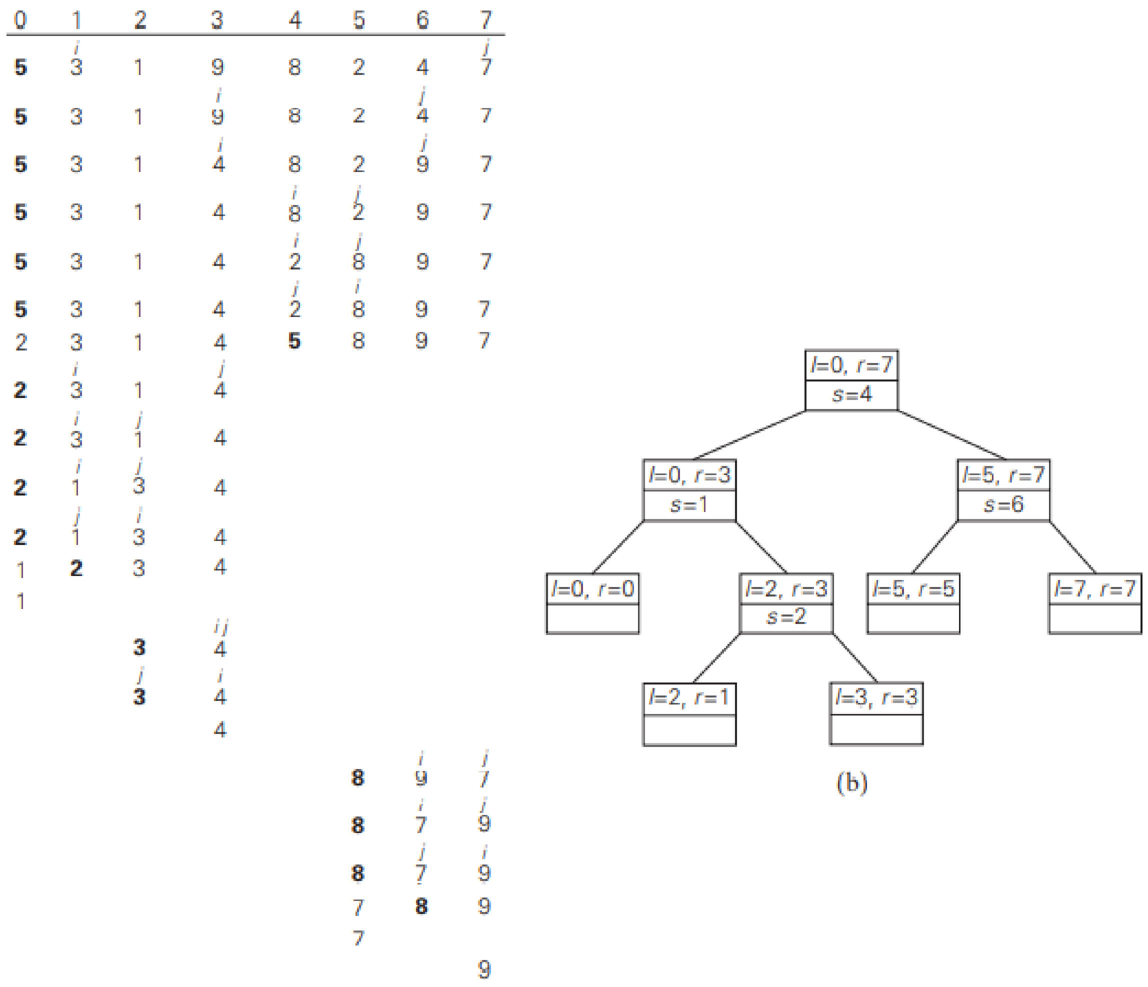
For example, we can check whether an array has equal elements by the following two-part algorithm: first, sort the array by applying some known sorting algorithm; second, scan the sorted array to check its consecutive elements for equality. If, for example, a sorting algorithm used in the first part makes no more than $\frac{1}{2}n(n-1)$ comparisons (and hence is in $O(n^2)$) while the second part makes no more than $n-1$ comparisons (and hence is in $O(n)$), the efficiency of the entire algorithm will be in $O(\max\{n^2, n\}) = O(n^2)$.

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])
```



(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.

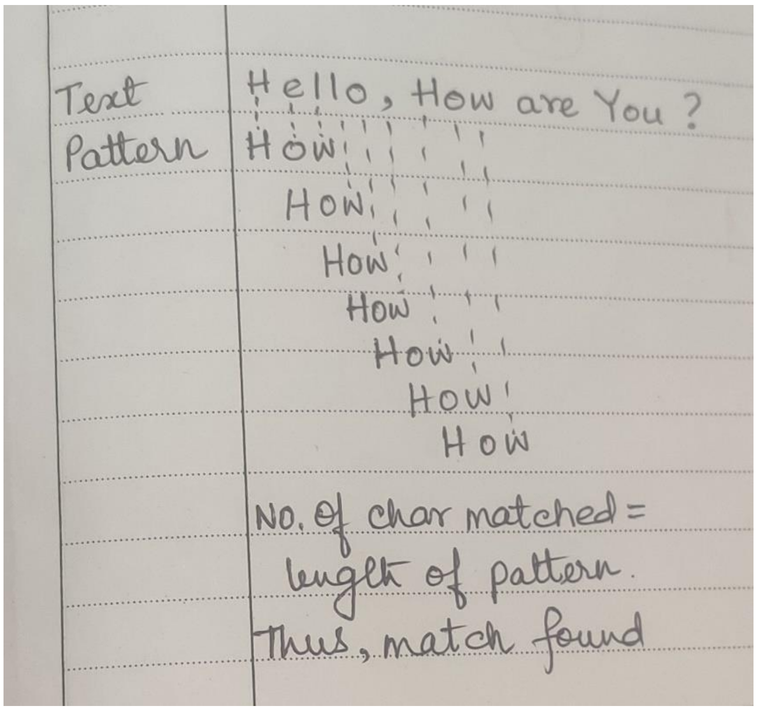
6. Write an algorithm to implement Brute Force’s string-matching process and apply the same for the given input. Text String = [Hello, How Are You?]   Pattern String=[How]

Given a string of n characters called the text and a string of m characters (m ≤ n) called the pattern, find a substring of the text that matches the pattern. To put it more precisely, we want to find i—the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

$$\begin{array}{ccccccccccc} t_0 & \ldots & t_i & \ldots & t_{i+j} & \ldots & t_{i+m-1} & & \ldots & t_{n-1} & \text{text } T \\ & & \updownarrow & & \updownarrow & & \updownarrow & & & & \\ & & p_0 & \ldots & p_j & \ldots & p_{m-1} & \text{pattern } P & & & \end{array}$$

If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted. align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all the m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered.

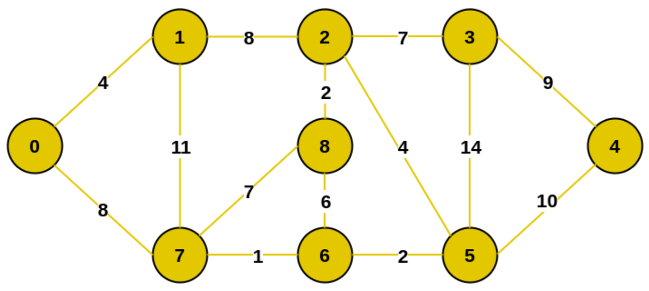
```
Algorithm Brute Force string match (T[0..,n-1], P[0..m-1])
// Input: An array T [0..n-1] of n chars, text
//            An array P [0..m-1] of m chars , a pattern.
// Output: The position of the first character in the text that starts the first
//            matching substring if the search is successful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



Text   Hello, How are You ?
Pattern How
        How
         How
          How
           How
            How
             How

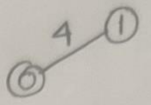
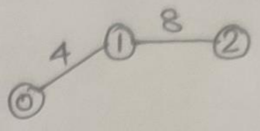
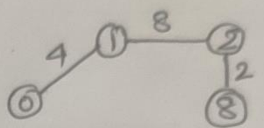
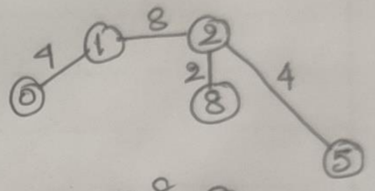
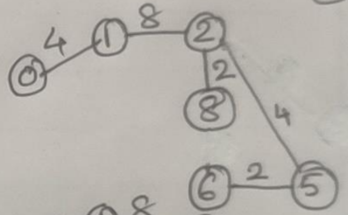
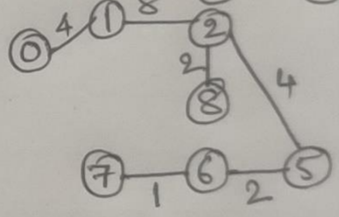
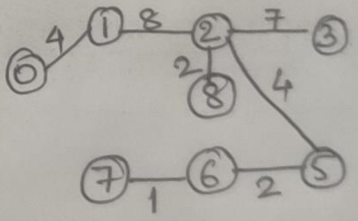
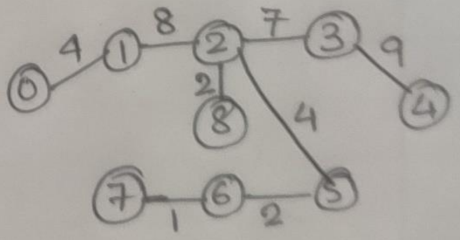
No. of char matched = length of pattern.
Thus, match found

7. Write Prim’s algorithm and trace it using the graph below as input



```
ALGORITHM Prim(G) //Prim’s algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = V,E
//Output: ET , the set of edges composing a minimum spanning tree of G VT ← {v0}
//the set of tree vertices can be initialized with any vertex ET ← ∅
for i ← 1 to |V | − 1 do
        find a minimum-weight edge e∗ = (v∗, u∗) among all the edges (v, u)
        such that v is in VT and u is in V − VT
        VT ← VT ∪ {u∗}
        ET ← ET ∪ {e∗}
return ET
```

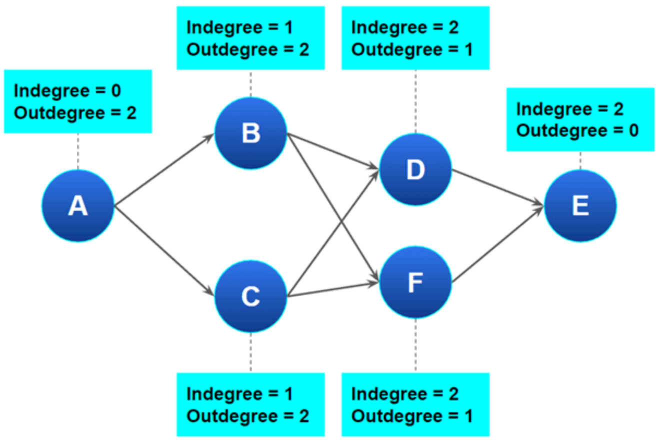
0(–,–)	$\underline{1(0,4)}$, 2(–,∝), 3(–,∝), 4(–,∝) 5(–,∝), 6(–,∝), 7(0,8), 8(–,∝)
1(0,4)	$\underline{2(1,8)}$, 3(–,∝), 4(–,∝), 5(–,∝) 6,(–,∝), 7(0,8), 8(–,∝)
2(1,8)	3(2,7), 4(–,∝), 5(2,4), 6(–,∝) 7(0,8), $\underline{8(2,2)}$
8(2,2)	3(2,7), 4(–,∝), $\underline{5(2,4)}$, 6(8,6) 7(8,7)
5(2,4)	3(2,7), 4(5,10), $\underline{6(5,2)}$, 7(8,7)
6(5,2)	3(2,7), 4(5,10), $\underline{7(6,1)}$
7(6,1)	$\underline{3(2,7)}$, 4(5,10)
3(2,7)	$\underline{4(3,9)}$

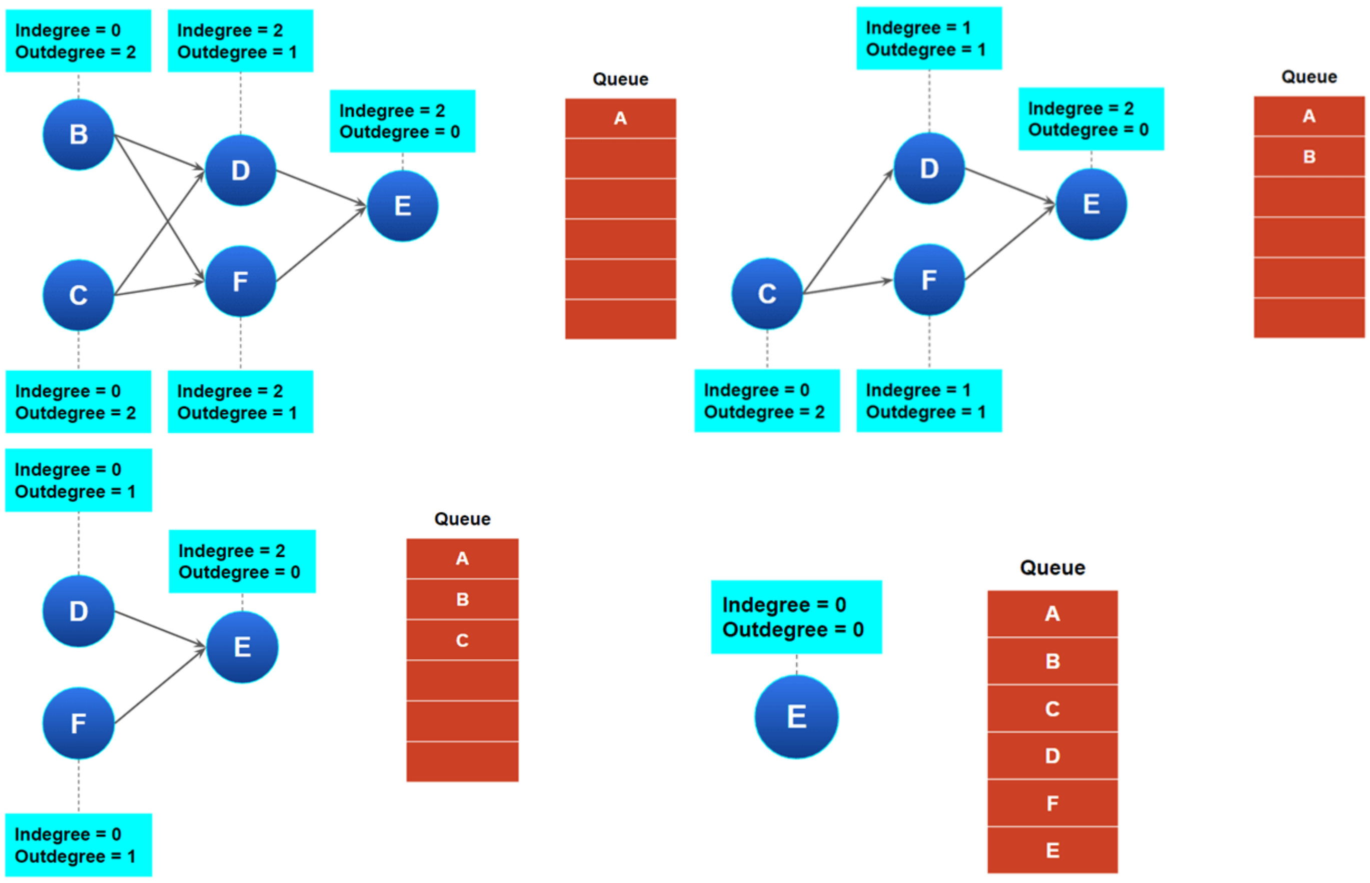
Minimum Spanning Tree
Cost = 37.

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

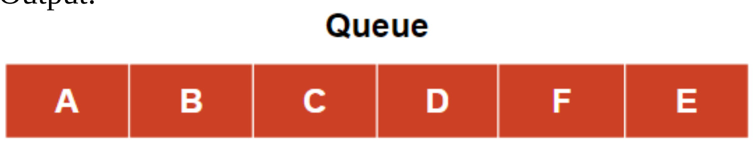
Step 1: Indegree of A is 0, so add A to queue and remove A->B and A->C from graph


Indegree = 0 Outdegree = 2
Indegree = 1 Outdegree = 2
Indegree = 2 Outdegree = 1
Indegree = 2 Outdegree = 0
Indegree = 1 Outdegree = 2
Indegree = 2 Outdegree = 1
A
B
C
D
F
E

Step 2: Indegree of B and C is 0, so send either of them to queue and remove the corresponding edges. Continue doing the same for all other nodes till end of graph.

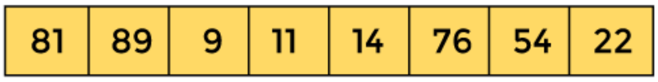

Indegree = 0 Outdegree = 2
Indegree = 2 Outdegree = 1
Indegree = 2 Outdegree = 0
Indegree = 0 Outdegree = 2
Indegree = 2 Outdegree = 1
B
C
D
F
E
Queue
A
Indegree = 1 Outdegree = 1
Indegree = 2 Outdegree = 0
Indegree = 0 Outdegree = 2
Indegree = 1 Outdegree = 1
C
D
F
E
Queue
A
B
Indegree = 0 Outdegree = 1
Indegree = 2 Outdegree = 0
Indegree = 0 Outdegree = 1
D
F
E
Queue
A
B
C
Indegree = 0 Outdegree = 0
E
Queue
A
B
C
D
F
E

Output:



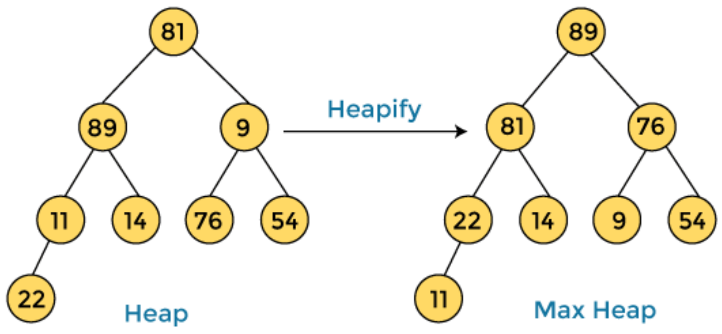
```
#include<stdio.h>
int main()
{
        int a[20][20],visit[20],ind, n,i,j,flag=0,count=0;
        printf("Enter the value of n\n");
        scanf("%d",&n);
        printf("Enter the adjacency matrix\n");
        for(i=0;i<n;i++)
        {
                visit[i]=0;
                for(j=0;j<n;j++)
                        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.

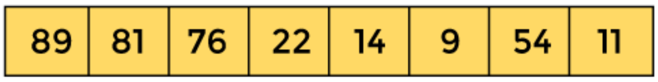


81	89	9	11	14	76	54	22

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

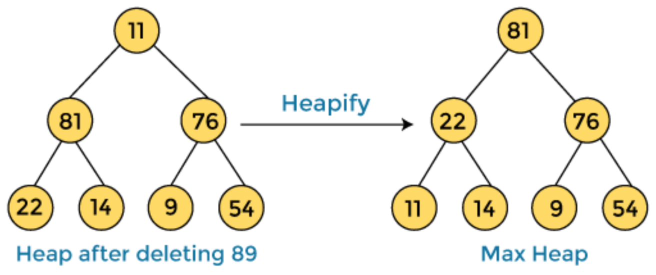


After converting the given heap into max heap, the array elements are -

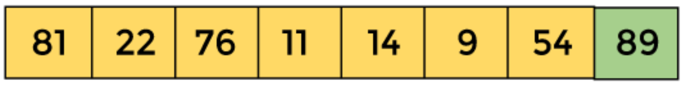


89	81	76	22	14	9	54	11

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.

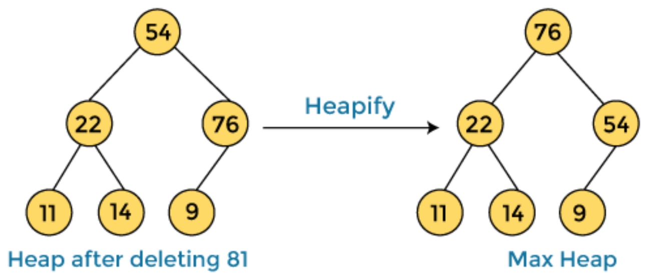


After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -



81	22	76	11	14	9	54	89

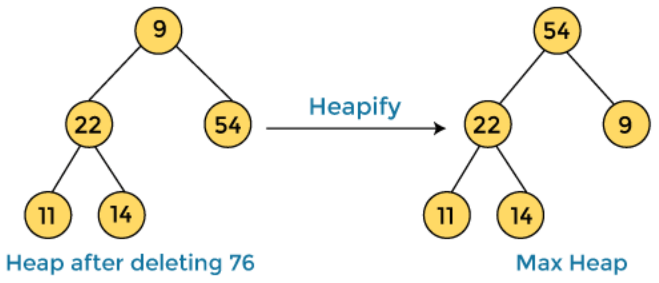
In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



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

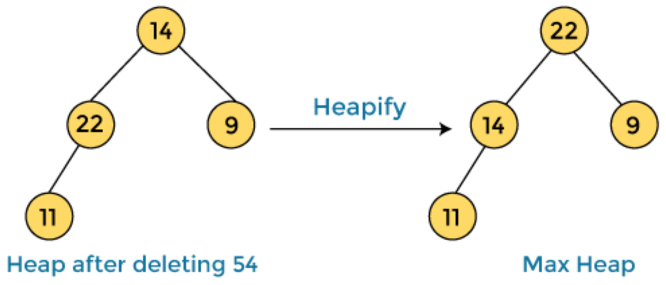
In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



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

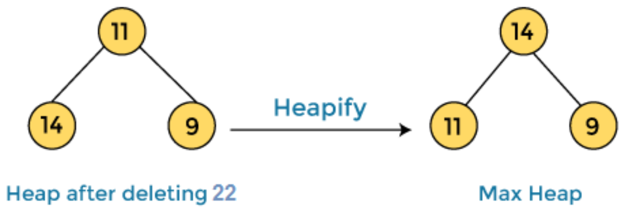
In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



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

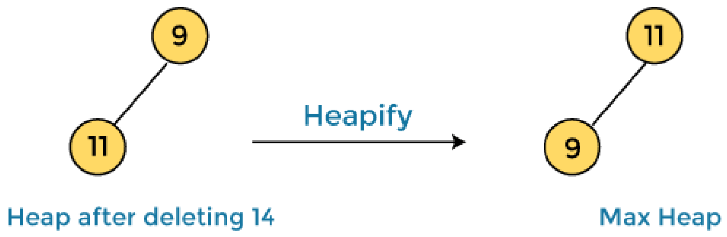
In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



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

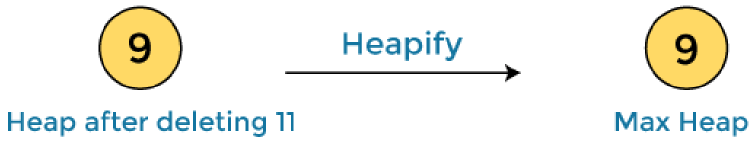
In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



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.



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.

9
Remove 9
Empty

After completion of sorting, the array elements are -

9	11	14	22	54	76	81	89

Now, the array is completely sorted.

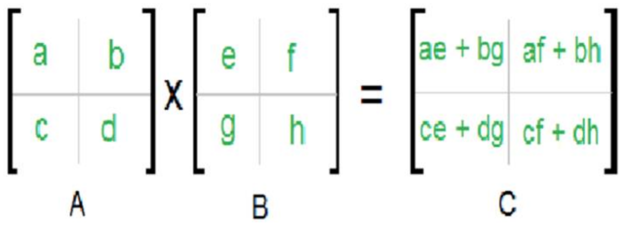
```
#include <stdio.h>
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of heap. */
void heapify(int a[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right 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.



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/2Xn/2 matrices and addition of their n/2xn/2 products. Using these equations to define a divide and conquer strategy we can get the relation among them as:

$T(N) = 8T(N/2) + O(N2)$
From the above we see that simple matrix multiplication takes eight recursion calls.
$T(n)=O(n\text{^}3)$
Thus, this method is faster than the ordinary one.
It takes only seven recursive calls, multiplication of n/2xn/2 matrices and O(n^2) scalar additions and subtractions, giving the below recurrence relations.

$T(N) = 7T(N/2) + O(N2)$
Steps of Strassen's matrix multiplication:

1. Divide the matrices A and B into smaller submatrices of the size n/2xn/2.
2. Using the formula of scalar additions and subtractions compute smaller matrices of size n/2.
3. Recursively compute the seven matrix products Pi=AiBi for i=1,2,…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.
Pi=AiBi
=(α1ia+α2ib+α3ic)(β1ie+β2if+β2ig)
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.

$p1 = a(f - h)$
$p2 = (a + b)h$
$p3 = (c + d)e$
$p4 = d(g - e)$
$p5 = (a + d)(e + h)$
$p6 = (b - d)(g + h)$
$p7 = (a - c)(e + f)$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}$$