Important Instructions to examiners:

1) The answers should be examined by key words and not as word-to-word as given in the model answer scheme.
2) The model answer and the answer written by candidate may vary but the examiner may try to assess the understanding level of the candidate.
3) The language errors such as grammatical, spelling errors should not be given more importance. (Not applicable for subject English and Communication Skills).
4) While assessing figures, examiner may give credit for principal components indicated in the figure. The figures drawn by candidate and model answer may vary. The examiner may give credit for any equivalent figure drawn.
5) Credits may be given step wise for numerical problems. In some cases, the assumed constant values may vary and there may be some difference in the candidate's answers and model answer.
6) In case of some questions credit may be given by judgement on part of examiner of relevant answer based on candidate's understanding.
7) For programming language papers, credit may be given to any other program based on equivalent concept.

Q.1. Attempt any FIVE: 20M

a) Define algorithm and write the algorithm to find sum and average of three numbers.
(Definition 2Marks, algorithm for Sum 1Mark, Average 1Mark)

Ans:

An algorithm is step-by-step procedure to be followed in order to solve a given problem. An Algorithm is a self-contained step-by-step set of operations to be followed.
A program is one type of algorithm.
Directions to somebody's house is an algorithm.

Algorithm for sum of three numbers:

i. Start.
ii. Read the numbers.
iii. Add three numbers and store the result in a variable.
iv. Display the result from the variable.
v. End

Algorithm for average of three numbers:

i. Start.
ii. Read three numbers.
iii. Add three numbers and divide the result by 3.
iv. Display the Average.
v. End.

b) Write Algorithm of merge sort.
(Correct Algorithm 4Marks)

Ans:

Algorithm:

- The *mergesort* algorithm is based on the divide-and-conquer paradigm. It operates as follows:
- *DIVIDE*: Partition the n-element sequence into two subsequences of n/2 elements.
- *CONQUER*: Sort the two subsequences recursively using the merge sort.
- *COMBINE*: Merge the two sorted subsequences of size n/2 each to produce the sorted sequence consisting of n elements.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine' step. To perform the merging, we use an auxiliary procedure MERGE(A,p,q,r), where A is an array and p, q, and r are indices numbering elements of the array $p \le q <$ r . the procedure assumes that the sub arrays $A[p .. q]$ and $A[q + 1 .. r]$ are sorted order. It merge them to form a single sorted sub array that replaces the current sub array $A[p .. q]$.
MERGE (A, p, q, r)

```
1.  n1 ← q − p + 1
2.  n2 ← r − q
3.  Create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
4.  for i ← 1 TO n1
5.      do L[i] ← A[p + i − 1]
6.  for j ← 1 TO n2
7.      do R[j] ← A[q + j ]
8.  L[n1 + 1] ← ∞
9.  R[n2 + 1] ← ∞
10. i ← 1
11. j ← 1
12. for k ← p to r
13.     do if L[i ] ≤ R[ j]
14.           then A[k] ← L[i]
15.                i ← i + 1
16.           else A[k] ← R[j]
17.                j ← j + 1
```

c) Give the concept of knapsack programming.
(Relevant explanation 4Marks)

Ans:

Knapsack problem says:

i. Given *n* objects each have a *weight* w_i and a *value* v_i, and given a knapsack of total *capacity* W.

ii. The problem is to pack the knapsack with these objects in order to maximize the total value of those objects packed without exceeding the knapsack's capacity.

iii. Determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible.

iv. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most useful items.

v. More formally, let x_i denote the fraction of the object i to be included in the knapsack, $0 \le x_i \le 1$, for $1 \le i \le n$. The problem is to find values for the x_i such that,

$$\sum_{i=1}^{n} x_i w_i \le W \text{ and } \sum_{i=1}^{n} x_i v_i \text{ is maximized.}$$

d) Explain the term node, edge, cycle and path for the graph.
(Correct explanation of each term 1 Mark)

Ans:

Node: Nodes are nothing but the element/vertices in graph which are connected to each other.

Edge: These are connection lines in graph which represent relationships between the nodes of graph.

Cycle: Cycle is nothing but the path from a node to itself. In Graph cycle the initial and final node is the same.

Path: Path is nothing but the direction or route from one node to another.

e) Explain recursion and give the recursive procedure for Fibonacci series.
(Explanation of Recursion 2Marks, Fibonacci Series Procedure 2Marks)

Ans:

i. Recursion is the process of repeating the items in a self-similar way.
ii. Recursion is a programming technique in which a call to a method appears in that method's body (i.e., a method calls itself)
iii. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.
iv. While using recursion programmer need to specify exit condition in the function, otherwise the program will go into infinite loop.

Recursive Procedure for Fibonacci series:

i. Read number.
ii. Call the function Fib(n);
iii. int Fib (int n)
if (n==0 || n==1)
return n
else
return Fib (n-1) + Fib (n-2);
iv. Display returned value.

f) Write the algorithm for Kruskal algorithm.

(Correct Algorithm /Steps - 4Marks)

Ans:

Steps

1. Sort the edge list on weight in non decreasing order.
2. Select next edge and include it in the subset if it does not form cycle.
3. Go to step 2 if all vertices are not included in subset.

KRUSKAL(V,E,W)

1) A ⟵ { } --set A ultimately contains the edges of MST.
2) for each vertex v in V
3) do MAKE_SET (V)
4) sort E in non-decreasing order by weight w
5) for each (u, v) from sorted list
6) do if FIND_SET(u) = FIND_SET(u)
7) Then A ⟵ A∪{(u, v)}
8) UNION(u, v)
9) Return A

g) Explain how is graph represented in different ways.

(Each Method 2Marks)

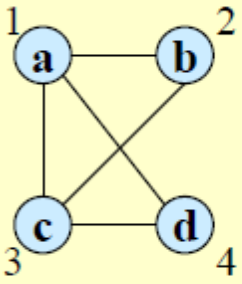
Ans:

Graph: a graph is non-empty set of vertices & edges denoted by G & given by G=(V, E)

Two methods for sequential representation of graph:

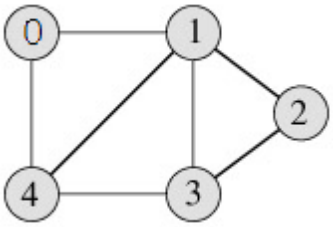
1) Adjacency matrix.
2) Linked representation.

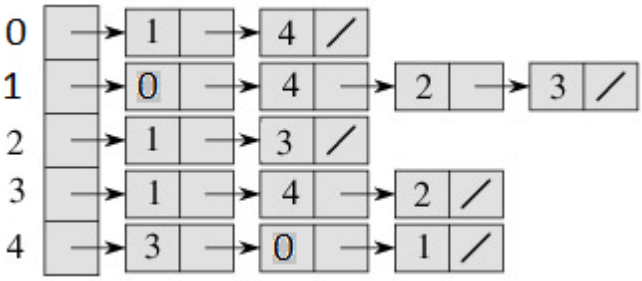
Adjacency matrix representation of the graph:



	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

Linked representation of graph:





Q.2. Attempt any TWO **16M**

a) Write a program for quick sort and explain with example.

(Correct Program 4Marks, Example 4Marks)

Note: Any other relevant example shall be considered

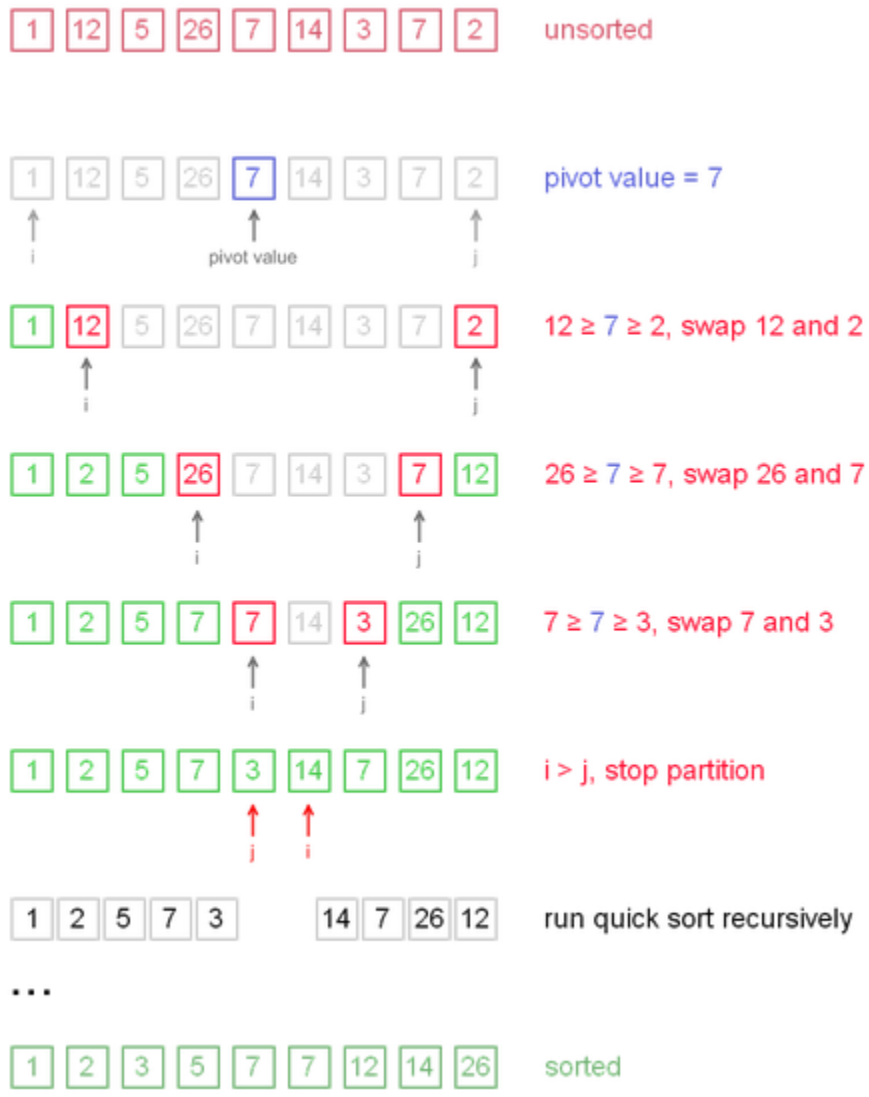
Ans:

Program –

```
#include<stdio.h>
#include<conio.h>

//quick Sort function to Sort Integer array list
void quicksort(int array[], int firstIndex, int lastIndex)
{
  //declaaring index variables
  int pivotIndex, temp, index1, index2;

  if(firstIndex < lastIndex)
  {
        //assigninh first element index as pivot element
        pivotIndex = firstIndex;
        index1 = firstIndex;
        index2 = lastIndex;

        //Sorting in Ascending order with quick sort
        while(index1 < index2)
        {
        while(array[index1] <= array[pivotIndex] && index1 < lastIndex)
        {
              index1++;
        }
        while(array[index2]>array[pivotIndex])
        {
        index2--;
        }
        if(index1<index2)
        {
             //Swapping opertation
              temp = array[index1];
              array[index1] = array[index2];
              array[index2] = temp;
        }
  }
```

```
  //At the end of first iteration, swap pivot element with index2 element
  temp = array[pivotIndex];
  array[pivotIndex] = array[index2];
  array[index2] = temp;

  //Recursive call for quick sort, with partiontioning
  quicksort(array, firstIndex, index2-1);
  quicksort(array, index2+1, lastIndex);
   }
}

int main()
{
  //Declaring variables
  int array[100],n,i;

  //Number of elements in array form user input
  printf("Enter the number of element you want to Sort : ");
  scanf("%d",&n);

  //code to ask to enter elements from user equal to n
  printf("Enter Elements in the list : ");
  for(i = 0; i < n; i++)
  {
          scanf("%d",&array[i]);
   }
  //calling quickSort function defined above
  quicksort(array,0,n-1);
  //print sorted array
  printf("Sorted elements: ");
  for(i=0;i<n;i++)
     printf(" %d",array[i]);

  getch();
  return0;
}
```

Example –



b) Explain process scheduling.

(Explanation of process scheduling 4 Marks, Explanation of scheduling queues-4Marks)

Ans:

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and loaded process shares the CPU using time multiplexing**.**

Whenever the CPU becomes idle, it is the job of the CPU scheduler (the short term scheduler) to select another process from the ready queue to run next.

The storage structure for the ready queue and the algorithm used to select the next process are not necessarily a FIFO queue.

Almost all programs have some alternating cycle of CPU number crunching and waiting for I/O of some kind. (Even a simple fetch from memory takes a long time relative to CPU speeds.

In a simple system running a single process, the time spent waiting for I/O is wasted, and those CPU cycles are lost forever.

A scheduling system allows one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles.

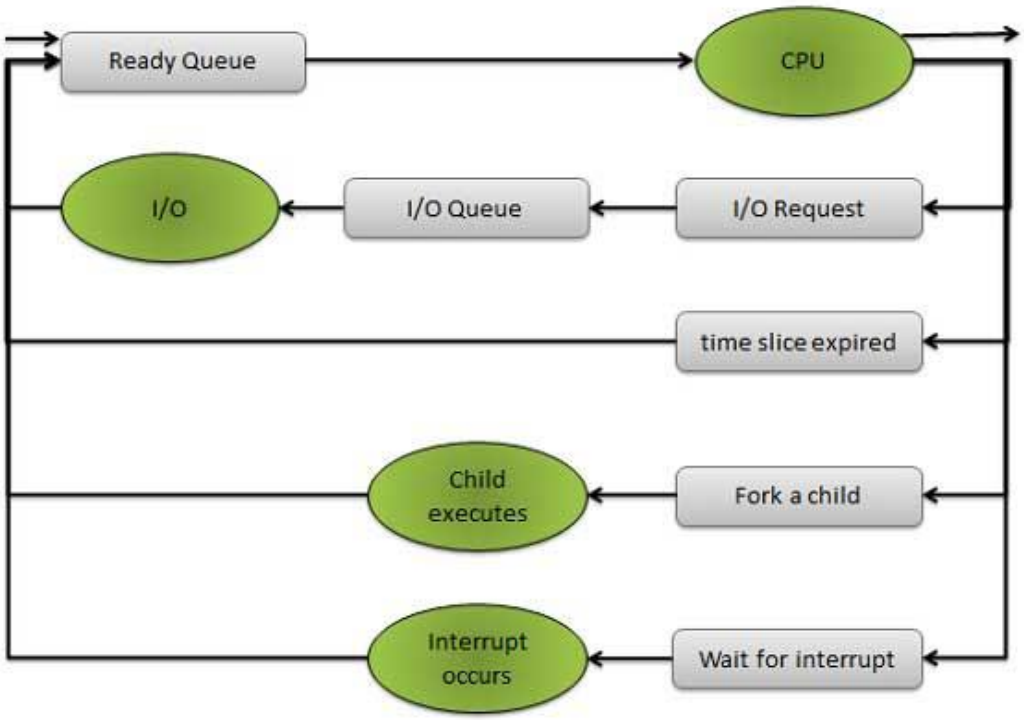
The challenge is to make the overall system as "efficient" and "fair" as possible, subject to varying and often dynamic conditions and where "efficient" and "fair" are somewhat subjective terms, often subject to shifting priority policies.

Scheduling Queues

Scheduling queues refers to queues of processes or devices. When the process enters into the system, then this process is put into a job queue. This queue consists of all processes in the system. The operating system also maintains other queues such as device queue. Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.

This figure shows the queuing diagram of process scheduling.

- Queue is represented by rectangular box.
- The circles represent the resources that serve the queues.
- The arrows indicate the process flow in the system.



Queues are of two types

- Ready queue
- Device queue

A newly arrived process is put in the ready queue. Processes waits in ready queue for allocating the CPU. Once the CPU is assigned to a process, then that process will execute. While executing the process, any one of the following events can occur.

- The process could issue an I/O request and then it would be placed in an I/O queue.
- The process could create new sub process and will wait for its termination.
- The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

c) Explain the algorithm for breadth first search.

(Algorithm 4Marks, Example 4Marks)(Any other relevant example shall be considered)

Ans:

Breadth-first search starts at a given vertex s, which is at level 0. In the first
the vertices that are at the distance of one edge away. When we visit t
"visited," the vertices adjacent to the start vertex s - these vertices are plac
the second stage, we visit all the new vertices we can reach at the distance c
from the source vertex s. These new vertices, which are adjacent to level
previously assigned to a level, are placed into level 2, and so on. T
terminates when every vertex has been visited.

To keep track of progress, breadth-first-search colors each vertex. Each ver
in one of three states:

1. Undiscovered;
2. Discovered but not fully explored; and
3. Fully explored.

The state of a vertex, u, is stored in a color variable as follows:

1. color[u] = White - for the "undiscovered" state,
2. color [u] = Gray - for the "discovered but not fully explored" state, and
3. color [u] = Black - for the "fully explored" state.

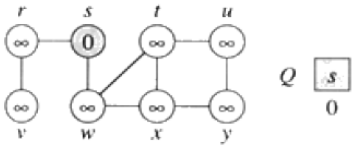
The BFS(G, s) algorithm develops a breadth-first search tree with the sour
root. The parent or predecessor of any other vertex in the tree is the vertex
first discovered. For each vertex, v, the parent of v is placed in the varia
variable, d[v], computed by BFS contains the number of tree edges on the
The breadth-first search uses a FIFO queue, Q, to store gray vertices.
BFS(V, E, s)

```
1.      for each u in V − {s}          ▷ for each vertex u in V[G] except s.
2.         do color[u] ← WHITE
3.            d[u] ← infinity
```

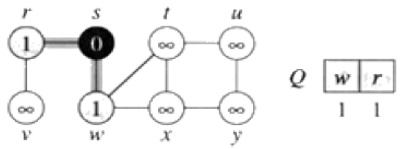
```
4.          π[u] ← NIL
5.      color[s] ← GRAY          ▷ Source vertex discovered
6.      d[s] ← 0                 ▷ initialize
7.      π[s] ← NIL               ▷ initialize
8.      Q ← {}                   ▷ Clear queue Q
9.      ENQUEUE(Q, s)
10      while Q is non-empty
11.         do u ← DEQUEUE(Q)            ▷ That is, u = head[Q]
12.            for each v adjacent to u  ▷ for loop for every node along with edge.
13.                do if color[v] ← WHITE   ▷ if color is white you've never seen it before
14.                     then  color[v] ← GRAY
15.                           d[v] ← d[u] + 1
16.                           π[v] ← u
17.                           ENQUEUE(Q, v)
18.            DEQUEUE(Q)
19.      color[u] ← BLACK
```

The following figure (from CLRS) illustrates the progress of breadth-first search on the undirected sample graph.

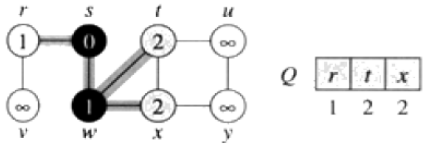
a. After initialization (paint every vertex white, set d[u] to infinity for each vertex u, and set the parent of every vertex to be NIL), the source vertex is discovered in line 5. Lines 8-9 initialize Q to contain just the source vertex s.



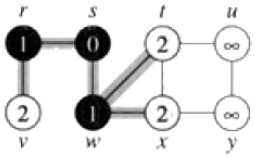
b. The algorithm discovers all vertices 1 edge from s i.e., discovered all vertices (w and r) at level 1.



c.

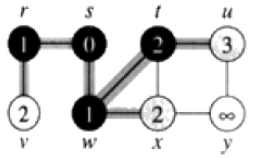


d. The algorithm discovers all vertices 2 edges from s i.e., discovered all vertices (t, x, and v) at level 2.



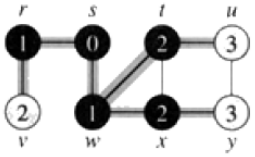
Q | t | x | v |
2 2 2

e.



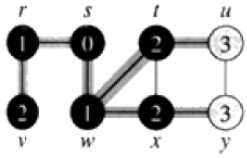
Q | x | v | u |
2 2 3

f.



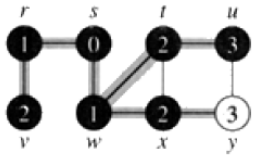
Q | v | u | y |
2 3 3

g. The algorithm discovers all vertices 3 edges from s i.e., discovered all vertices (u and y) at level 3.



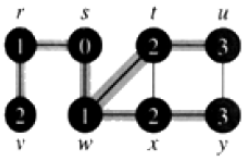
Q | u | y |
3 3

h.



Q | y |
3

i. The algorithm terminates when every vertex has been fully explored.



Q $\varnothing$

Q. 3 Attempt any FOUR: **16M**

a) Give fundamentals and properties of algorithm.

(*Any four points of Fundamentals 2 Marks & any four Properties 2Marks)*

Ans:

An algorithm is "a finite set of precise instructions for performing a computation or for solving a problem".

Fundamentals:

1. A sequence of instructions
2. Sequence is finite
3. Each instruction is unambiguous
4. Initial instructions acquire valid input
5. Intermediate instructions perform processing
6. Final instructions produce valid output for P
7. Algorithm terminates after finite number of steps

Properties:

1. Input: what the algorithm takes in as input.
2. Output: what the algorithm produces as output.
3. Definiteness: the steps are defined precisely.
4. Correctness: should produce the correct output.
5. Finiteness: the steps required should be finite.
6. Effectiveness: each step must be able to be performed in a finite amount of time.
7. Generality: the algorithm *should* be applicable to all problems of a similar form.

b) Explain divide and conquer strategy with suitable example.

(Divide and conquer strategy 2Marks, Example 2Marks)

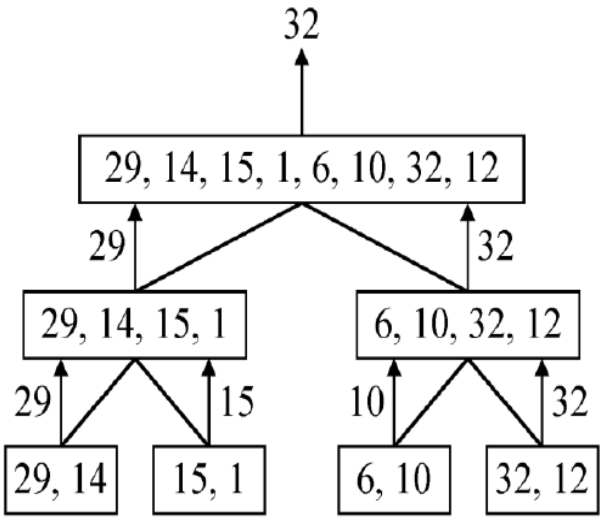
Note: Any other relevant example shall be considered

Ans:

Divide- and conquer is a top-down technique for designing algorithm that consists of dividing the problem into smaller sub problems hoping that the solutions of the sub problems are easier to find and then composing the partial solutions into the solution of the original problem. So, divide and conquer algorithms break the problem into several sub problems that are similar to the original problem but smaller in size, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem.

Little more formally, divide-and conquer paradigm consists of following major phases:
Breaking or divide the problem into several sub problem that are similar to the original problem but smaller size,
Conquer sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straightforward manner and then
Combine these solutions to sub problems to create a solution to the original problem.
The divide–and-conquer strategy solves problem by:

1. Breaking it into sub problems that are themselves smaller instances of the same type of problem.
2. Recursively solving these sub problems.
3. Appropriately combining their answers.



c) Sort the following list in ascending order using merge sort. Show intermediate passes.
(Correct Steps 4Marks)
Ans:

90, 20, 80, 89, 70, 65, 85, 74

Initial Array:

90	20	80	89	70	65	85	74

Pass 1:
Split the Array:

90	20	80	89	70	65	85	74

Pass 2:
Split the Array:

90	20	80	89	70	65	85	74

Pass 3:
Sort Arrays Internally:

20	90	80	89	65	70	74	85

Pass 4:
Merge the Arrays:

20	80	89	90	65	70	74	85

Pass 5:
Merge the Arrays:

20	65	70	74	80	85	89	90

d) Give general characteristics of greedy method
(Any four relevant Characteristics 1Mark each)
Ans:

1. Greedy is a strategy that works well on optimization problems with the following characteristics:
2. Greedy-choice property: A global optimum can be arrived at by selecting a local optimum.
3. Optimal substructure: An optimal solution to the problem contains an optimal solution to sub problems.
4. A function that checks whether chosen set of items provide a solution.
5. A function that checks the feasibility of a set.
6. The selection function tells which of the candidates is the most promising.
7. An objective function, which does not appear explicitly, gives the value of a solution.

e) Consider three items :

i	W_i	P_i
1	18	30
2	15	21
3	10	18

Also W=20, obtain the solution for the above given Knapsack problem.
(Correct solution 4 Marks)
Note: Any one method shall be considered
Ans:

There are two versions of the problem:
1. "0-1 knapsack problem"

Items are indivisible; you either take an item or not. Some special instances can be solved with *dynamic programming*

2. "Fractional knapsack problem"

Items are divisible: you can take any fraction of an item

Initially

Items	Wi	Pi
I1	18	30
I2	15	21
I3	10	18

Taking profit per weight ratio i.e. Ri= Pi/Wi

Items	Wi	Pi	Ri=Pi/Wi
I1	18	30	1.6
I2	15	21	1.4
I3	10	18	1.8

Now arrange the value of Ri in descending order

Items	Wi	Pi	Ri=Pi/Wi
I3	10	18	1.8
I1	18	30	1.6
I2	15	21	1.4

Now, fill the knapsack according to decreasing value of Ri

First we choose item I3 whose weight is 10. Now the total weight in knapsack is 10. Now the next item is I1 whose weight is 18 but we want only 10. (The capacity of knapsack (W) =20, so we choose fractional part of it is,

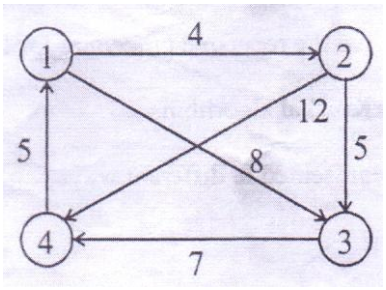
10	= 20
10	

The value of fractional part of I1=(30/18*10)=16.6 . Thus the maximum value is=18+16.6=34.6

Resultant Vector =

I1	I2	I3
0.56	0.0	1.0

f) Obtain all pair shortest paths for the following graph:



(Each correct step 1Mark)

Ans:

D^0 =

	1	2	3	4
1	0	4	8	∞
2	∞	0	5	12
3	∞	∞	0	7
4	5	∞	∞	0

	1	2	3	4
1	0	4	8	∞
2	∞	0	5	12
3	∞	∞	0	7
4	5	∞	∞	0

$D1=D^0 W$

	1	2	3	4
1	0	4	8	15
2	17	0	5	12
3	12	∞	0	7
4	5	9	13	0

$D2=D^1 W$

	1	2	3	4
1	0	4	8	15
2	17	0	5	12
3	12	16	0	7
4	5	9	13	0

$D3=D^2W$

Q.4 Attempt any FOUR: **16M**

a) **Explain time and space complexity. Give example.**

(Time complexity description 1Mark &any example 1Mark, space complexity description 1Mark & any example 1Mark)

Ans:

Time complexity

Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

e.g.

Algorithm	Time Complexity
a=a+1	1
for x=1 to n a=a+1 Loop	n
for x=1 to n step1 for y=1 to n step2 a=a+1 Loop Loop	n^2

Space complexity

Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm. We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural(but fixed-length) units to measure this.

e.g.
Memory space S(P)
Program P
Fixed Space Requirements (C)
S(P) = c + Sp(instance)
Simple arithmetic function
float abc (float a, float b, float c)

```
{
        return a + b + b * c + (a + b -c) / (a + b) + 4.00;
}
```

For every instance 3 computer words required to store variables: a, b, and c. Therefore Sp()= 3. S(P) = 3.

b) Explain the term pivot element with example in quick sort.

(Pivot Element- 2Marks, any relevant example-2Marks)

Ans:

Step 1. Choosing the Pivot Element

Choosing the pivot element can determine the complexity of the algorithm i.e. whether it will be n*logn or quadratic time:

a. Normally we choose the first, last or the middle element as pivot. This can harm us badly as the pivot might end up to be the smallest or the largest element, thus leaving one of the partitions empty.

b. We should choose the Median of the first, last and middle elements. If there are N elements, then the ceiling of N/2 is taken as the pivot element.

Example:

8, 3, 25, 6, 10, 17, 1, 2, 18, 5
first element: 8
middle element: 10
last element: 5

Therefore the median on [8,10,5] is 8.

Step 2. Partitioning

a. First thing is to get the pivot out of the way and swapping it with the last number.

Example: (shown using the above array elements)
5, 3, 25, 6, 10, 17, 1, 2, 18, 8

b. Now we want the elements greater than pivot to be on the right side of it and similarly the elements less than pivot to be on the left side of it.

For this we define 2 pointers, namely i and j. i being at the first index and j being and the last index of the array.

* While i is less than j we keep in incrementing i until we find an element greater than pivot.
* Similarly, while j is greater then i keep decrementing j until we find an element less than pivot.
*After both i and j stop we swap the elements at the indexes of i and j respectively.

c. Restoring the pivot
When the above steps are done correctly we will get this as our output:
[5, 3, 2, 6, 1] [8] [10, 25, 18, 17]

Step 3. Recursively Sort the left and right part of the pivot.

c) Prove that for quick sort, the quick sort efficiency is T(N) = O(N^2).
(Correct steps 4Marks)
Ans:
The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element. One of *L* and *G* has size *n* - 1 and the other has size 0 .The running time is proportional to the sum*n* + (*n* - 1) + … + 2 + 1
Thus, the worst-case running time of quick-sort is *O* (*n*2)
Depth time

0	N
1	N-1
….	….
….	….
N-1	1

Worst-Case Analysis
The pivot is the smallest (or the largest) element
T (N) = T (N-1) + cN, N > 1
Telescoping:
T (N-1) = T (N-2) + c (N-1)
T (N-2) = T (N-3) + c (N-2)
T (N-3) = T (N-4) + c (N-3)
……...
T (2) = T (1) + c.2

T (N) + T (N-1) + T (N-2) + … + T (2) =
= T (N-1) + T (N-2) + … + T (2) + T (1) +
C (N) + c (N-1) + c (N-2) + … + c.2
T (N) = T (1) +c times (the sum of 2 thru N)
= T (1) + c (N (N+1) / 2 -1) = O (N2)

d) Give the difference between divide and conquer and greedy algorithm.

(Method- 1Mark each, any example-1Mark each)
Ans:

Divide and Conquer basically works in three steps.

1. **Divide -** It first divides the problem into small chunks or sub-problems.
2. **Conquer -** It then solve those sub-problems recursively so as to obtain a separate result for each sub-problem.
3. **Combine -** It then combine the results of those sub-problems to arrive at a final result of the main problem.

Some Divide and Conquer algorithms are Merge Sort, Binary Sort, etc.

Dynamic Programming is similar to Divide and Conquer when it comes to dividing a large problem into sub-problems. But here, each sub-problem is solved only **once**. **There is no recursion**. The key in dynamic programming is **remembering**. That is why we store the result of sub-problems in a table so that we don't have to compute the result of a same sub-problem again and again.

Some algorithms that are solved using Dynamic Programming are Matrix Chain Multiplication, Tower of Hanoi puzzle, etc..

Another difference between Dynamic Programming and Divide and Conquer approach is that -
In Divide and Conquer, the sub-problems are independent of each other while in case of Dynamic Programming; the sub-problems are not independent of each other (Solution of one sub-problem may be required to solve another sub-problem).

e) Write the algorithm for Prim's Algorithm.

(Correct Algorithm 4 Marks)

Ans:

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. It finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. This algorithm is directly based on the MST(minimum spanning tree) property.

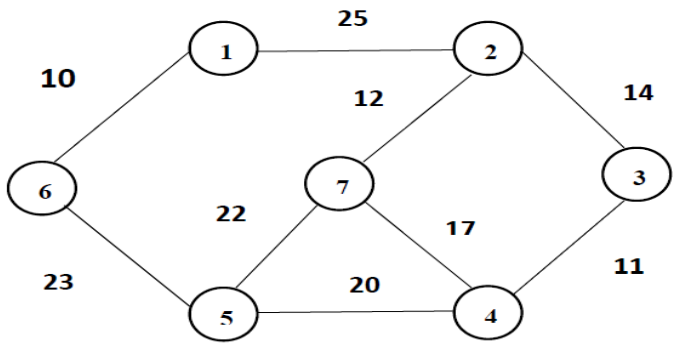
Algorithm:

i. Select edge with smallest weight includes it in subset.
ii. Select next adjacent vertex with minimum weight.
iii. Include it in the all vertices are not included in subset.
iv. Go to step ii, if all vertices are not included in subset.
v. After every step graph are connected graph.

OR

a. Create an array $B[1..n]$ to store the nodes of the MST, and an array $T[1..n-1]$ to store the edges of the MST.
b. Starting with node 1 (actually, any node can be the starting node), put node 1 in $B[1]$, find a node that is the closest (i.e., an edge connected to node 1 that has the minimum weight, ties broken arbitrarily).
c. Put this node as $B[2]$, and the edge as $T[1]$.
d. Next look for a node connected from either $B[1]$ or $B[2]$ that is the closest, store the node as $B[3]$, and the corresponding edge as $T[2]$. In general, in the kth iteration, look for a node not already in $B[1..k]$ that is the closest to any node in $B[1..k]$.
e. Put this node as $B[k+1]$, the corresponding edge as $T[k]$.
f. Repeat this process for $n-1$ iterations ($k = 1$ to $n-1$).
g. This is a greedy strategy because in each iteration, the algorithm looks for the minimum weight edge to include next while maintaining the tree property (i.e., avoiding cycles).
h. At the end there are exactly $n-1$ edges without cycles, which must be a spanning tree.

f) Find the minimum spanning tree using Kruskal's algorithm.



(Correct steps 4Marks)

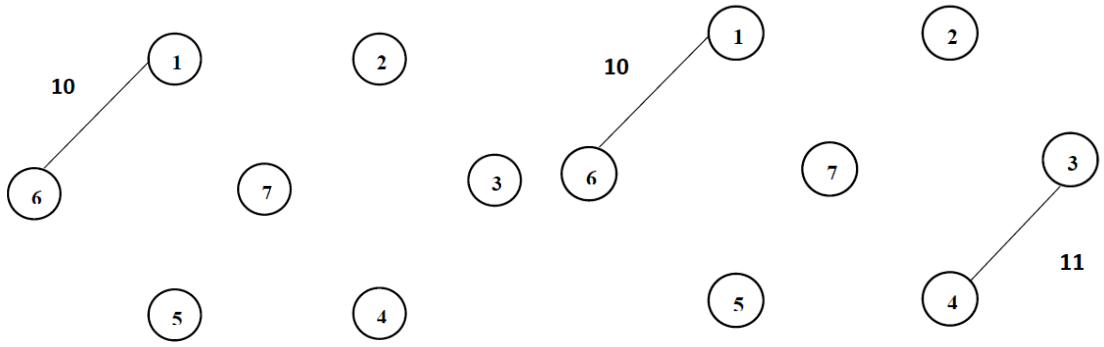
Ans:

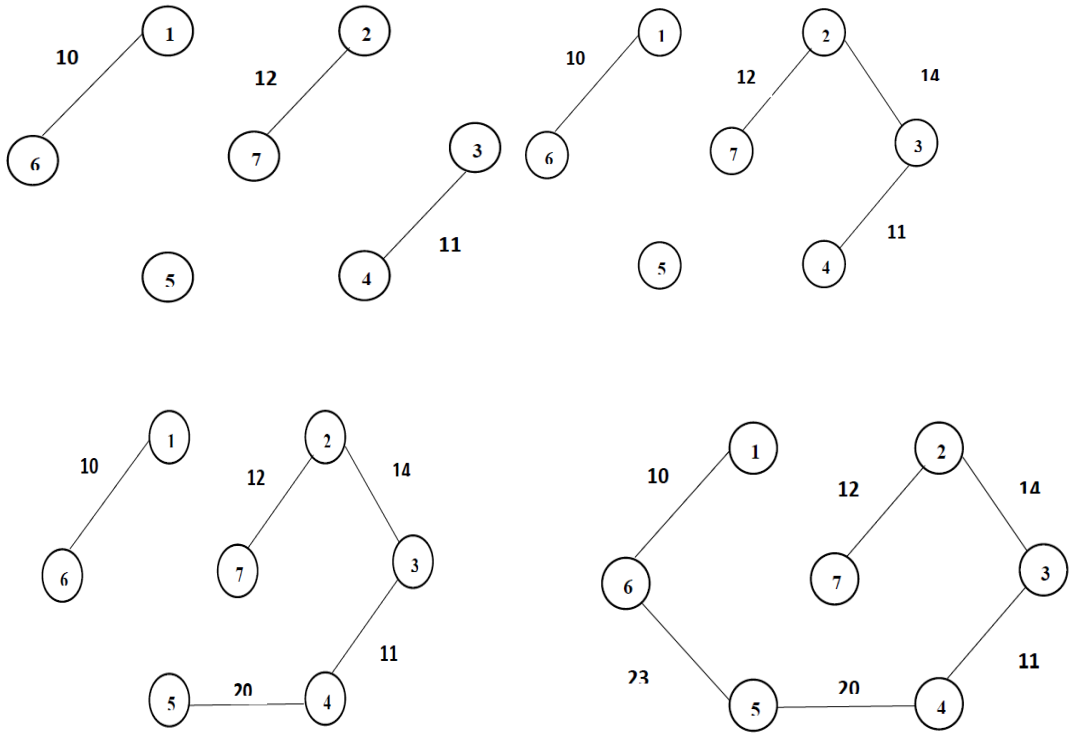
Steps

1. Sort the edge list on weight in non-decreasing order.
2. Select next edge and include it in the subset if it does not form cycle.
3. Go to step 2 if all vertices are not included in subset.

Set of edges in non-decreasingorder

A={ (1,6)=10, (3,4)=11, (2,7)=12, (2,3)=14, (7,4)=17, (5,4)=20, (7,5)=22, (5,6)=23, (1,2)=25 }



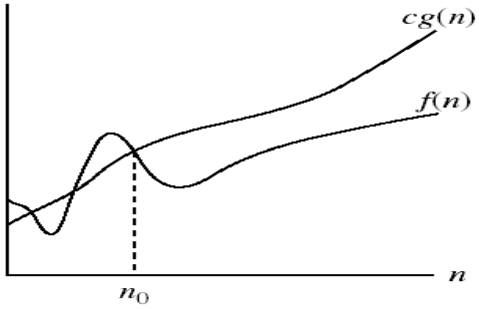


MST of above Example = 90

Q.5 Attempt any TWO **16M**

a) Explain big-oh ,omega and theta notation with the help of an example.
(Big-oh notation- 3Marks, Omega notation-3Marks, Theta notation -2Marks)
Ans:
Big-oh notation:
Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time it could possibly take for the Algorithm to complete. More formally, for non-negative functions, f(n) and g(n), if there Exists an integer n0 and a constant c>0 such that for all integers n>n0

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

- When we have a polynomial that describes the time requirements of an algorithm, we simplify it by:
  - Throwing out all but the highest-order term
  - Throwing out all the constants
- If an algorithm takes $12n^3+4n^2+8n+35$ time, we simplify this formula to just n^3
- We say the algorithm requires $O(n^3)$ time
  - We call this Big O notation
  - (More accurately, it's Big Ω, but we'll talk about that later)
- Recall that, if n is the size of the set, and m is the size of the (possible) subset:
  - We go through the loop in subset m times, calling member each time
  - We go through the loop in member n times
- Hence, the actual running time should be k*(m*n) + c, for some constants k and c
- We say that subset takes O(m*n) time
- Big O notation is a *huge* simplification; can we justify it?
  - It only makes sense for *large* problem sizes

- **For sufficiently large problem sizes, the highest-order term swamps all the rest!**

- Consider $R = x^2 + 3x + 5$ as x varies:

x = 0	$x^2 = 0$	3x = 0	5 = 5	R = 5
x = 10	$x^2 = 100$	3x = 30	5 = 5	R = 135
x = 100	$x^2 = 10000$	3x = 300	5 = 5	R = 10,305
x = 1000	$x^2 = 1000000$	3x = 3000	5 = 5	R = 1,003,005
x = 10,000	$x^2 = 10^8$	$3x = 3*10^4$	5 = 5	R = 100,030,005
x = 100,000	$x^2 = 10^{10}$	$3x = 3*10^5$	5 = 5	R = 10,000,300,005

Omega notation

Big-**Omega Notation**[edit] For non-negative functions, f(n) and g(n), if there exists an integer and a constant c > 0 such that for all integers , f(n) ≥ cg(n), then f(n) is **omega** of g(n). This is denoted as "f(n) = Ω(g(n))".

Definition 1.5 [Omega] The function $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") iff there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all n, $n \geq n_0$. □

Example 1.12 The function $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for $n \geq 1$ (the inequality holds for $n \geq 0$, but the definition of Ω requires an $n_0 > 0$). $3n+3 = \Omega(n)$ as $3n+3 \geq 3n$ for $n \geq 1$. $100n+6 = \Omega(n)$ as $100n+6 \geq 100n$ for $n \geq 1$. $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for $n \geq 1$. $6 * 2^n + n^2 = \Omega(2^n)$ as $6 * 2^n + n^2 \geq 2^n$ for $n \geq 1$. Observe also that $3n + 3 = \Omega(1)$, $10n^2 + 4n + 2 = \Omega(n)$, $10n^2 + 4n + 2 = \Omega(1)$, $6 * 2^n + n^2 = \Omega(n^{100})$, $6 * 2^n + n^2 = \Omega(n^{50.2})$, $6 * 2^n + n^2 = \Omega(n^2)$, $6 * 2^n + n^2 = \Omega(n)$, and $6 * 2^n + n^2 = \Omega(1)$. □

Theta notation

Definition 1.6 [Theta] The function $f(n) = \Theta(g(n))$ (read as "f of n is theta of g of n") iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$. □

Example 1.13 The function $3n + 2 = \Theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$ and $3n + 2 \leq 4n$ for all $n \geq 2$, so $c_1 = 3$, $c_2 = 4$, and $n_0 = 2$. $3n + 3 = \Theta(n)$, $10n^2 + 4n + 2 = \Theta(n^2)$, $6 * 2^n + n^2 = \Theta(2^n)$, and $10 * \log n + 4 = \Theta(\log n)$. $3n + 2 \neq \Theta(1)$, $3n + 3 \neq \Theta(n^2)$, $10n^2 + 4n + 2 \neq \Theta(n)$, $10n^2 + 4n + 2 \neq \Theta(1)$, $6 * 2^n + n^2 \neq \Theta(n^2)$, $6 * 2^n + n^2 \neq \Theta(n^{100})$, and $6 * 2^n + n^2 \neq \Theta(1)$. □

The theta notation is more precise than both the the big oh and omega notations. The function $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper and lower bound on $f(n)$.

Notice that the coefficients in all of the $g(n)$'s used in the preceding three examples have been 1. This is in accordance with practice. We almost never find ourselves saying that $3n + 3 = O(3n)$, that $10 = O(100)$, that $10n^2 + 4n + 2 = \Omega(4n^2)$, that $6 * 2^n + n^2 = O(6 * 2^n)$, or that $6 * 2^n + n^2 = \Theta(4 * 2^n)$, even though each of these statements is true.

b) Write a program to sort the series of numbers using radix sort.

(Correct Program -8 Marks)

Note: Any other logic shall be considered

Ans:

Radix sort is an algorithm. Radix Sort sorts the elements by processing its individual digits. Radix sort processing the digits either by Least Significant Digit(LSD) method or by Most Significant Digit(MSD) method.

```
#include <stdio.h>
#define MAX 100
#define SHOWPASS
void print(int *a, int n) {
  int i;
  for (i = 0; i < n; i++)
  printf("%d\t", a[i]);
}
```

```
void radix_sort(int *a, int n) {
  int i, b[MAX], m = 0, exp = 1;
  for (i = 0; i < n; i++) {
        if (a[i] > m)
          m = a[i];
  }
  while (m / exp> 0) {
        int box[10] = {
              0
        }
        ;
        for (i = 0; i < n; i++)
        box[a[i] / exp % 10]++;
        for (i = 1; i < 10; i++)
        box[i] += box[i - 1];
        for (i = n - 1; i >= 0; i--)
        b[--box[a[i] / exp % 10]] = a[i];
        for (i = 0; i < n; i++)
        a[i] = b[i];
        exp *= 10;
        #ifdef SHOWPASS
        printf("\n\nPASS   : ");
        print(a, n);
        #endif
  }
}
int main() {
  intarr[MAX];
  int i, num;
  printf("\nEnter total elements (num< %d) : ", MAX);
  scanf("%d", &num);
  printf("\nEnter %d Elements : ", num);
  for (i = 0; i <num; i++)
  scanf("%d", &arr[i]);
  printf("\nARRAY  : ");
  print(&arr[0], num);
  radix_sort(&arr[0], num);
```

```
  printf("\n\nSORTED  : ");
  print(&arr[0], num);
  return 0;
}
```

c) Explain the concept of depth first search.
(DFS concept – 4 Marks, any relevant example – 4 Marks)
Ans:

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

1. Depth-first search selects a source vertex s in the graph and paint it as "visited." Now the Vertex s becomes our current vertex. Then, we traverse the graph by considering an Arbitrary edge (u, v) from the current vertex u. If the edge (u, v) takes us to a painted Vertex v, then we back down to the vertex u. On the other hand, if edge (u, v) takes us to an unpainted vertex, then we paint the vertex v and make it our current vertex, and repeat the above computation.

2. DFS time-stamps each vertex when its color is changed.
1. When vertex v is changed from white to gray the time is recorded in d[v].
2. When vertex v is changed from gray to black the time is recorded in f[v].
3. The discovery and the finish times are unique integers, where for each vertex the finish time is always after the discovery time. That is, each time-stamp is an unique integer in the range of 1 to 2|V| and for each vertex v, d[v] < f[v]. In other words, the following inequalities hold:

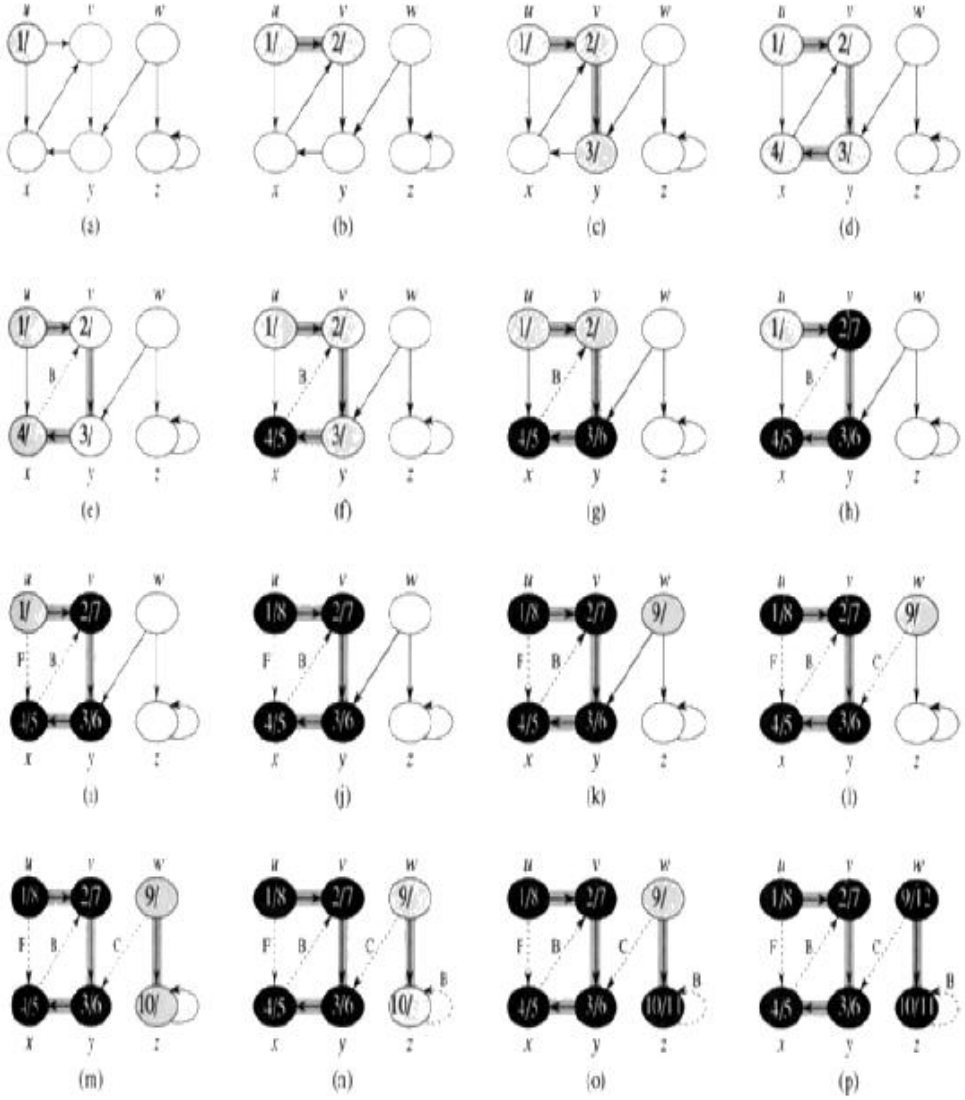
$1 \leq d[v] < f[v] \leq 2|V|$

The DFS forms a depth-first forest comprised of more than one depth-first trees. Each tree is made of edges (u, v) such that u is gray and v is white when edge (u,v) is explored. The following pseudo code for DFS uses a global timestamp time.

DFS (V, E)
1. for each vertex u in V[G]
2. do color[u] ← WHITE
3. π[u] ← NIL
4. time ← 0
5. for each vertex u in V[G]
6. do if color[u] ← WHITE
7. then DFS-Visit(u) ▷ build a new DFS-tree from u

DFS-Visit(u)

1. color[u] ← GRAY ▷ discover u
2. time ← time + 1
3. d[u] ← time
4. for each vertex v adjacent to u ▷ explore (u, v)
5. do if color[v] ← WHITE
6. then π[v] ← u
7. DFS-Visit(v)
8. color[u] ← BLACK
9. time ← time + 1
10. f[u] ← time ▷ we are done with u


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)
(o)
(p)

Q.6.Attempt any FOUR **16M**

a) Compare any three sorting algorithms given their time complexity.

(Any three algorithms with four comparison points 1-Mark each)(any three algorithms shall be considered.)

Ans:

II. Comparison Of Sorting Algorithms

	Time					
Sort	Avg.	Best	Worst	Space	Stability	Remarks
Bubble sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable	Always use a modified bubble sort
Selection Sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable	Even a perfectly sorted input requires scanning the entire array
Insertion Sort	O(n^2)	O(n)	O(n^2)	Constant	Stable	In the best case (already sorted), every insert requires constant time
Merge Sort	O(n*logn)	O(n*logn)	O(n*logn)	Depends	Stable	On arrays, merge sort requires O(n) space; on linked lists, merge sort requires constant space
Quick Sort	O(n*logn)	O(n*logn)	O(n^2)	Constant	Stable	Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array.

b) Give the best case, Worst case and average case analysis of merge sort.

(Worst case-1Mark, Average case-2Marks, Best case-1Mark)

Ans:

Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one.

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

Average Case Time = $\frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$

$= \frac{\theta((n+1)*(n+2)/2)}{(n+1)}$

$= \Theta(n)$

Best Case Analysis (Bogus)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\Theta(1)$

Time Complexity.	Merge Sort
Best case	O(n log(n))
Average case	O(n log(n))
Worst case	O(n log(n))

c) Sort the following numbers using heap sort:

96, 15, 12, 02, 25, 46, 72, 48, 85

(Correct Steps -4Marks)

Note: Ascending / Descending order shall be considered.

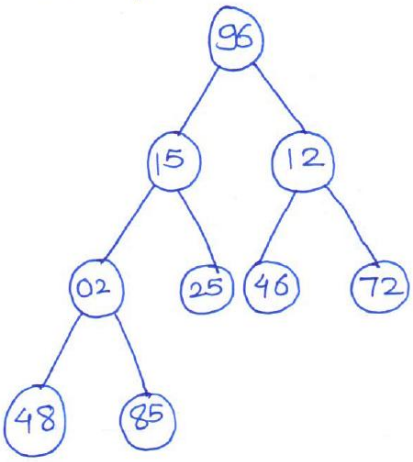
Ans:

i. Build Heap.
ii. Transform the Heap into Min Heap.
iii. Delete the root node.
iv. Put the last node of the heap in root position.
v. Repeat from 2 till all nodes are covered.

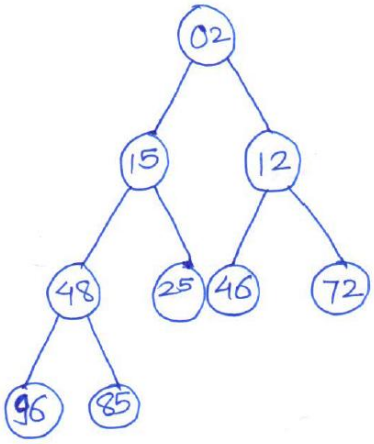
96	15	12	02	25	46	72	48	85

Steps for Heap Sort for above example:

1)



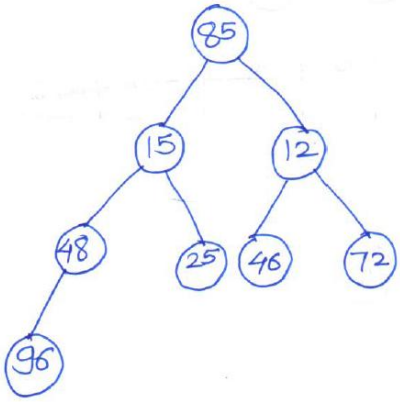
2)



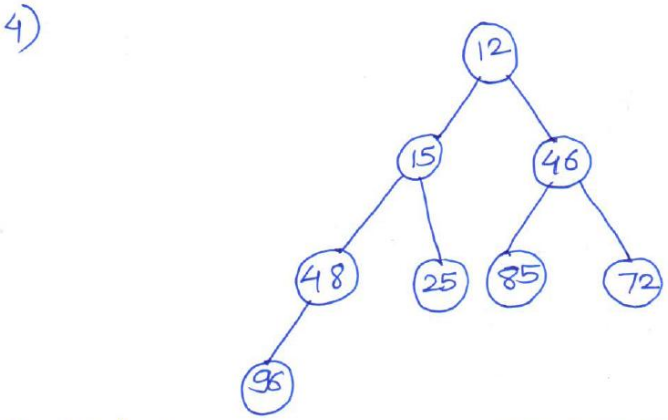
Sorted Array

0	1	2	3	4	5	6	7	8
02								

3)



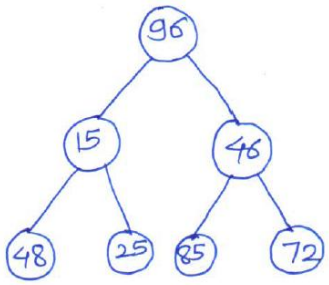
4)



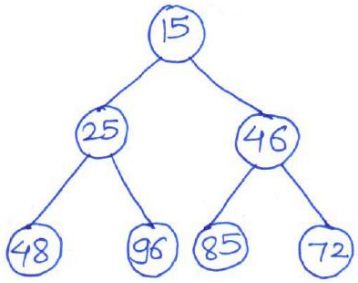
Sorted Array

0	1	2	3	4	5	6	7	8
02	12							

5)



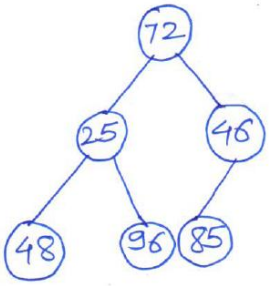
6)



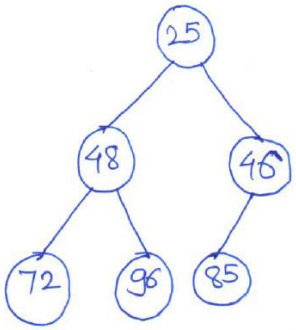
Sorted Array

0	1	2	3	4	5	6	7	8
02	12	15						

7)



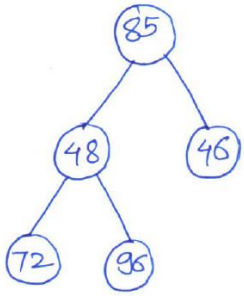
8)



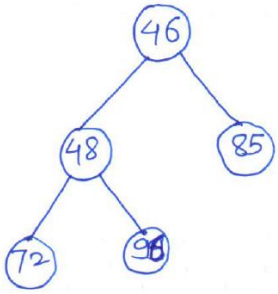
Sorted Array

0	1	2	3	4	5	6	7	8
02	12	15	25					

9)



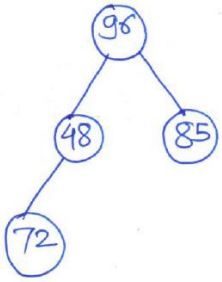
10)



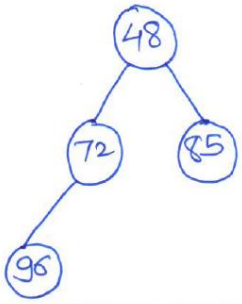
Sorted Array

0	1	2	3	4	5	6	7	8
02	12	15	25	46				

11)



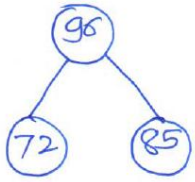
12)



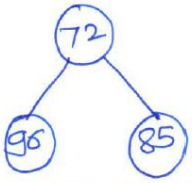
Sorted Array

0	1	2	3	4	5	6	7	8
02	12	15	25	46	48			

13)



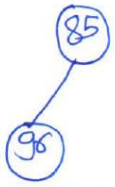
14)



Sorted Array

0	1	2	3	4	5	6	7	8
02	12	15	25	46	48	72		

15)



Sorted Array

0	1	2	3	4	5	6	7	8
02	12	15	25	46	48	72	85	

16)

Sorted Array

0	1	2	3	4	5	6	7	8
02	12	15	25	46	48	72	85	96

d) Explain the concept of counting sort with suitable example

(Explanation-2Marks, Any relevant example-2Marks)

Ans:

Counting sort is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm. It is a linear time sorting algorithm used to sort items when they belong to a fixed and finite set.
The algorithm proceeds by defining an ordering relation between the items from which the set to be sorted is derived (for a set of integers, this relation is trivial).Let the set to be sorted be called A. Then, an auxiliary array with size equal to the number of items in the superset is defined, say B. For each element in A, say e, the algorithm stores the number of items in A smaller than or equal to e in B(e). If the sorted set is to be stored in an array C, then for each e in A, taken in reverse order, C[B[e]] = e. After each such step, the value of B(e) is decremented.

Example:

10, 07, 12, 04, 09, 13

Find min and max value from above input.

Create array named index from min value to max value.

index	04	05	06	07	08	09	10	11	12	13

Now, create array that will hold the count of each number. Count the number of times each number appears in the input and insert the count across it.

index	04	05	06	07	08	09	10	11	12	13
count	1	0	0	1	0	1	1	0	1	1

Now create sumCount array that will hold the sum of counts for given index.

index	04	05	06	07	08	09	10	11	12	13
sumCount	1	1	1	2	2	3	4	4	5	6

Now insert the input elements at their specified position in sorted array and decrement the way of sumCount of that element.

From above example element 10 is inserted at position 4.

position	1	2	3	4	5	6
sorted array				10		

Now sumCount value of element 10 is decreased by 1.

index	04	05	06	07	08	09	10	11	12	13
sumCount	1	1	1	2	2	3	3	4	5	6

Next is element 07 it is inserted at position 2.

position	1	2	3	4	5	6
sorted array		07		10		

Now sumCount value of element 07 is decreased by 1.

index	04	05	06	07	08	09	10	11	12	13
sumCount	1	1	1	1	2	3	3	4	5	6

Next is element 12 it is inserted at position 5.

position	1	2	3	4	5	6
sorted array		07		10	12	

Now sumCount value of element 12 is decreased by 1.

index	04	05	06	07	08	09	10	11	12	13
sumCount	1	1	1	1	2	3	3	4	4	6

Next is element 04 it is inserted at position 1.

position	1	2	3	4	5	6
sorted array	04	07		10	12	

Now sumCount value of element 04 is decreased by 1.

index	04	05	06	07	08	09	10	11	12	13
sumCount	0	1	1	1	2	3	3	4	4	6

Next is element 09 it is inserted at position 3.

position	1	2	3	4	5	6
sorted array	04	07	09	10	12	

Now sumCount value of element 04 is decreased by 1.

index	04	05	06	07	08	09	10	11	12	13
sumCount	0	1	1	1	2	2	3	4	4	6

Next is element 13 it is inserted at position 6.

position	1	2	3	4	5	6
sorted array	04	07	09	10	12	13

Now sumCount value of element 04 is decreased by 1.

index	04	05	06	07	08	09	10	11	12	13
sumCount	0	1	1	1	2	2	3	4	4	5

Sorted List is:04,07,09,10,12,13.

e) Explain job sequencing for the instance n=5,(p1,p2,p3,p4,p5)=(20,15,10,5,1)
(Process execution sequence 2Marks, Correct Maximum profit value 2Marks)
Note: Deadline is not mentioned so students can take any deadline.
Ans:

(d1,d2,d3,d4,d5)=(2,1,2,1,2).

Given,

Job's	Profit	Deadline
Job1	20	2
Job2	15	1
Job3	10	2
Job4	5	1
Job5	1	2

First sort the Jobs in descending order by using profit,

Job's	Profit	Deadline
Job1	20	2
Job2	15	1
Job3	10	2
Job4	5	1
Job5	1	2

Dmax (maximum deadline) = 2

So,

1	2	
empty	empty	← Clock cycle

So, first the job1 must be executed on or before 2 deadline (2nd clock cycle) to get its profit.

So we can assign job1 to 2nd clock cycle. (Because 2nd clock cycle is empty)

1	2	
empty	Job1	← Clock cycle

Then job2 must be executed on or before 1 deadline (1st clock cycle) to get its profit.

So we can assign job1 to 1st clock cycle. (Because 1st clock cycle is empty)

1	2	
Job2	Job1	← Clock cycle

Then Job3 must be executed on or before 2 deadline (2nd clock cycle) to get its profit but th available clock cycle so we leave this job and proceed to next.

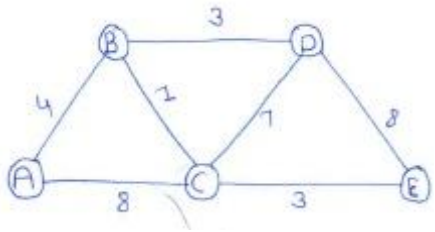
Then Job4 must be executed on or before 1 deadline (1st clock cycle) to get its profit but the available clock cycle so we leave this job and proceed to next.

Then Job5 must be executed on or before 2 deadline (2nd clock cycle) to get its profit but th available clock cycle so we leave this job. (All jobs are finished)

Maximum Profit = Job 1 profit + Job2 profit = 20 + 15 = 35

NO	Feasible solution	Processing sequence	value
1	(1,2)	2,1	35
2	(1,3)	1,3 or 3,1	30
3	(1,4)	4,1	25
4	(1,5)	5,1	21
5	(2,3)	3,2	25
6	(2,4)	4,2	20
7	(2,5)	5,2	16
8	(3,4)	4,3	15
9	(3,5)	5,3	11
10	(4,5)	5,4	6
11	1	1	20
12	2	2	15
13	3	3	10
14	4	4	5
15	5	5	1

f) Solve using Dijkstra by giving the distance between vertices and path.



(Correct steps 4Marks)

Ans:

1]

A	B	C	D	E
0	∞	∞	∞	∞

S = { A }

2]

A	B	C	D	E
0	∞	∞	∞	∞
	4	8	∞	∞

S = { A, B }

3]

A	B	C	D	E
0	∞	∞	∞	∞
	4	8	∞	∞
		5	7	8

S = { A, B, C }

4]

A	B	C	D	E
0	∞	∞	∞	∞
	4	8	∞	∞
		5	7	8

S = { A, B, C, D }

5]

A	B	C	D	E
0	∞	∞	∞	∞
	4	8	∞	∞
		5	7	8

S = { A, B, C, D, E }

Final Distance for above example : 24

----------**********----------