MAHARASHTRA STATE BOARD OF TECHNICAL EDUCATION
(Autonomous)
(ISO/IEC - 27001 - 2005 Certified)

MODEL ANSWER

SUMMER– 17 EXAMINATION

Subject Title: DESIGN AND ANALYSIS OF ALGORITHMS **Subject Code:** 17636

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. No.	Sub Q. N.	Answer	Marking Scheme
1.		**Solve any FIVE:**	**(5x4= 20) Marks**
	(a)	**Explain Recursion with suitable example.**	**4M**
	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. Example for recursion - Calculating the factorial of a number	**(Explanation : 2 marks, Example: 2 marks)**

```
#include<stdio.h>
int fact(int);
main()
{
int num;
printf("\n Enter the number : ");
scanf("%d", num);
printf("\n Factorial of %d = %d", num fact 9num));
return 0;
```

```
}
int fact (int n);
{
if (n==1)
return 1;
}
```

(b) **Define Graph. Directed graph and undirected graph. Mention how to represent a graph.** **4M**

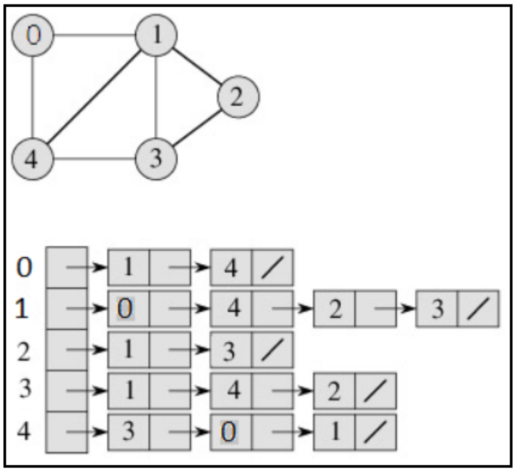
Ans: **Graph:** a graph is nonempty set of vertices & edges denoted by G & given by G=V, E)
Directed Graph: a directed graph is also identified as digraph with an ordered pair(u, v)

Undirected Graph: An undirected graph includes set of objects (called vertices or nodes) that are connected together, where all the edges are bidirectional. An undirected graph is sometimes called an undirected network.

Sequential representation of graphs:
Two methods for sequential representation of graph:

1. Adjacency matrix.
2. Linked representation.

Linked representation of graph:



(Definition: 2 marks, Representation of Graph: 2 marks)

(c) **Explain general Greedy Method. List various Greedy strategies.** **4M**

Ans:

(Explanation : 3 marks, Strategies: 1 mark)

The greedy method is the most straight forward design technique which has *n* inputs and requires obtaining a subset that satisfies some constraints. Any subset that satisfies these constraints is called a *feasible* solution. We are required to find a feasible solution that either maximizes or minimizes a given *objective function.* A feasible solution that does this is called an *optimal solution.* There is usually an obvious way to determine a feasible solution, but not necessarily an optimal solution.

The greedy method suggests that one can devise an algorithm which works in stages, considering one input at a time. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure. This measure may or may not be the objective function.

Greedy method control abstraction Algorithm

```
procedure GREEDY(A.n)
I IA(l:n) contains then inputs//
solution - ¢ I /initialize the solution to emptyI I
for i - 1 to n do
x - SELECT(A)
if FEASIBLE(solution,x)
        then solution - UNION(solution,x)
        endif
        repeat
return (solution)
end GREEDY
```

Various Greedy strategies:

Greedy-choice: A global optimum can be arrived at by selecting a local optimum

Optimal substructure: An optimal solution to the problem contains an optimal solution to sub problems

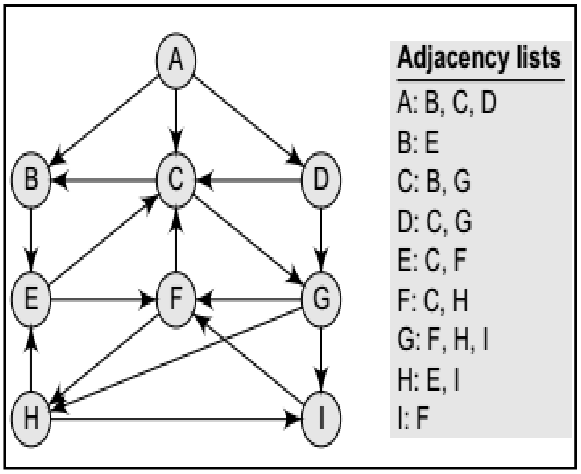
(d) **What is heap? Enlist its operation.** **4M**

Ans:

(Definition: 2 marks,

Heap is a special case of balanced binary tree data structure where the root-node key is

		compared with its children and arranged accordingly. If **α** has child node **β** then − **key(α) ≥ key(β)** As the value of parent is greater than that of child, this property generates **Max Heap**. Based on these criteria, a heap can be of two types – **Min-Heap** – Where, the value of the root node is less than or equal to either of its children. **Max-Heap** – Where, the value of the root node is greater than or equal to either of its children. **Operations in Heap:** • Find-max or Find-min • Insert • Extract-min [or extract-max] • Delete-max or delete-min	**Operation: 2 Marks)**
	(e)	**Explain Breath first search algorithm using any one example.**	**4M**
	Ans:	Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbor nodes, and so on, until it finds the goal. **Breadth First Search Algorithm** Step 1: SET STATUS = 1 (ready state) for each node in G Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state) Step 3: Repeat Steps 4 and 5 until QUEUE is empty Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state). Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP] Step 6: EXIT **Example:** Consider the graph G given in figure below. The adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to I with minimum stops. That is, find the minimum path P from A to I given that every edge has a length of 1.	**(Algorithm: 2 marks, Any Example: 2 marks)**



The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered. During the execution of the algorithm, we use two arrays: QUEUE and ORIG. While QUEUE is used to hold the nodes that have to be processed, ORIG is used to keep track of the origin of each edge. Initially, FRONT = REAR = –1. The algorithm for this is as follows:

(a) Add A to QUEUE and add NULL to ORIG.

FRONT = 0	QUEUE = A
REAR = 0	ORIG = \0

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

FRONT = 1	QUEUE = A B C D
REAR = 3	ORIG = \0 A A A

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

FRONT = 2	QUEUE = A B C D E
REAR = 4	ORIG = \0 A A A B

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G.

Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3	QUEUE = A B C D E G
REAR = 5	ORIG = \0 A A A B C

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4	QUEUE = A B C D E G
REAR = 5	ORIG = \0 A A A B C

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

FRONT = 5	QUEUE = A B C D E G F
REAR = 6	ORIG = \0 A A A B C E

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

FRONT = 6	QUEUE = A B C D E G F H I
REAR = 9	ORIG = \0 A A A B C E G G

Since F has already been added to the queue, we will only add H and I. As I is our final
destination, we stop the execution of this algorithm as soon as it is encountered and added
to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have P as A -> C -> G -> I.

(f) **Explain Dijkstra's algorithm finds the shortest path from a single source to the other nodes of a graph.** **4M**

Ans: Dijkstra algorithm, named after its discoverer, Dutch computer scientist Edsger Dijkstra, is a greedy algorithm that solves the signal-source shortest path problem fro a directed graph G=(V.E) with nonnegative edge weights i.e.. we assume that w(u,v)≥0 for each edge(u,v) *E.*

(Explanation : 4 marks)

Dijkstra algorithm maintain a set of S of vertices whose final shortest path weights from the source s have already been determined. That is, for all vertices v S, we have d[v]=δ(s,v). The algorithm repeatedly selects the vertex u V-S with the minimum shortest path estimate, inserts into S and relaxes all edges leaving u. we maintain a priority queue Q that contains all the vertices in v-s, keyed by their d values. Graph G is represented by adjacency lists.

DIJKSTRA (G, w, s)

```
INITIALIZE SINGLE-SOURCE (G, s)
S ← // S will ultimately contains vertices of final shortest-path weights from s
Initialize priority queue Q i.e., Q ← V[G]
while Q= priority queue Q is not empty do
do u ← EXTRACT_MIN(Q) // Pull out new vertex
S ← S {u} // Perform relaxation for each vertex v adjacent to u
for each vertex v [u]
do Relax (u, v, w)
```

(g) **Explain job scheduling with appropriate example.** **4M**

Ans: Job scheduling is the process of allocating system resources to many different tasks by an operating system (OS). The system handles prioritized job queues that are awaiting CPU time and it should determine which job to be taken from which queue and the amount of time to be allocated for the job. This type of scheduling makes sure that all jobs are carried out fairly and on time.

(Explanation : 2 marks, Example: 2 marks)

Example

Let n = 4,

Process (P1, P2, P3, P4) = (100, 10, 15, 27)

Deadline (D1, D2, D3, D4) = (2, 1, 2, 1)

$d_{max} = 2$

Feasible Solution	Processing Sequence	Profit value
(1, 2)	2, 1	110
(1, 2)	1, 3 or 3, 1	115
(1, 2)	4, 1	127
(1, 2)	2, 3	25
(1, 2)	4, 3	42
(1, 2)	1	100
(1, 2)	2	10
(1, 2)	3	15
(1, 2)	4	27

Solution (1, 4) is optimal. In this solution only job1 and job4 are processed and the value is 127. These jobs must be followed in order, job4 followed by job1. Thus the processing of job4 begins at time 0 and that of job1 is completed.

2. **Solve any TWO :** **(2x8=16) Marks**

(a) **Explain with suitable example the Depth first search, write the algorithm for DFS.** **8M**

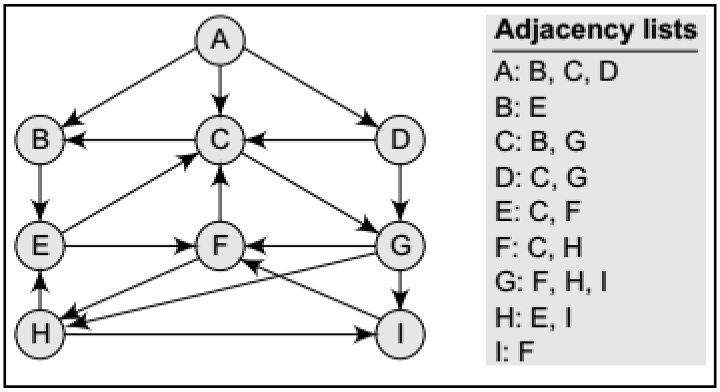
Ans: The **Depth-first search (DFS)** algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.In other words, depth-first search begins at a starting node A which becomes the current node.Then, it examines each node N along a path P which begins at A. That is, we process a neighbor of A, then a neighbour of neighbour of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.

(Explanation :4 marks, Example:4 marks)

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)
Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
[END OF LOOP]
Step 6: EXIT
```

Example: Consider the graph G given in figure. The adjacency list of G is also given. Suppose we want to print all the nodes that can be reached from the node H (including H itself).

One alternative is to use a depth-first search of G starting at node H.



Adjacency lists
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Solution

a) Push H onto the stack.

STACK: H

b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H STACK: E, I

c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I STACK: E, F

d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. The STACK now becomes

PRINT: F STACK: E, C

e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C STACK: E, B, G

f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

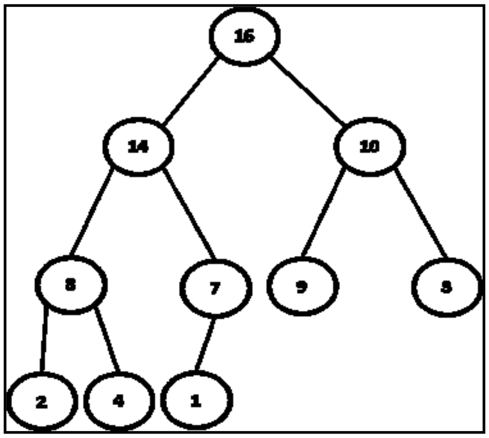
PRINT: G STACK: E, B

g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

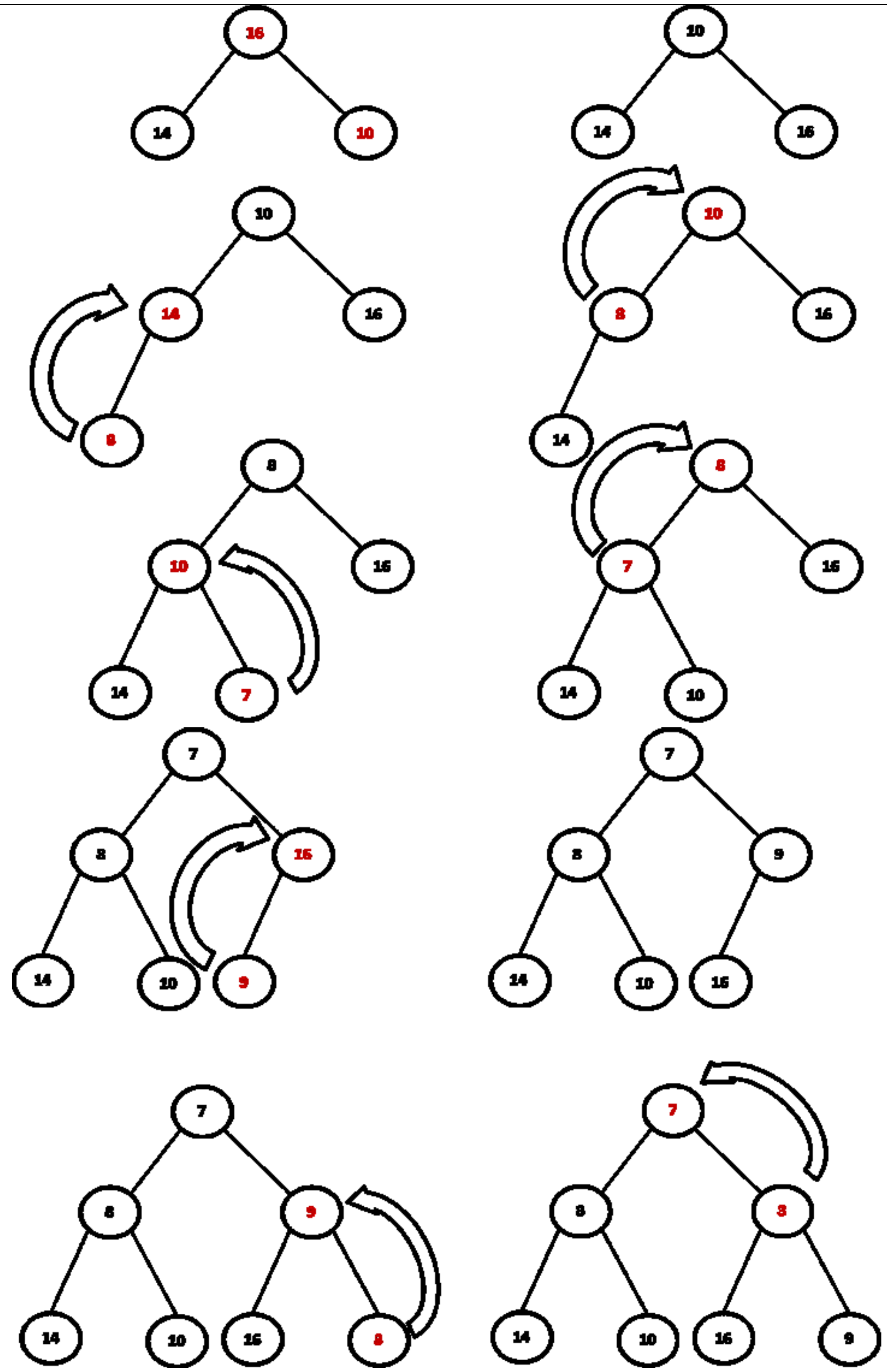
PRINT: B STACK: E

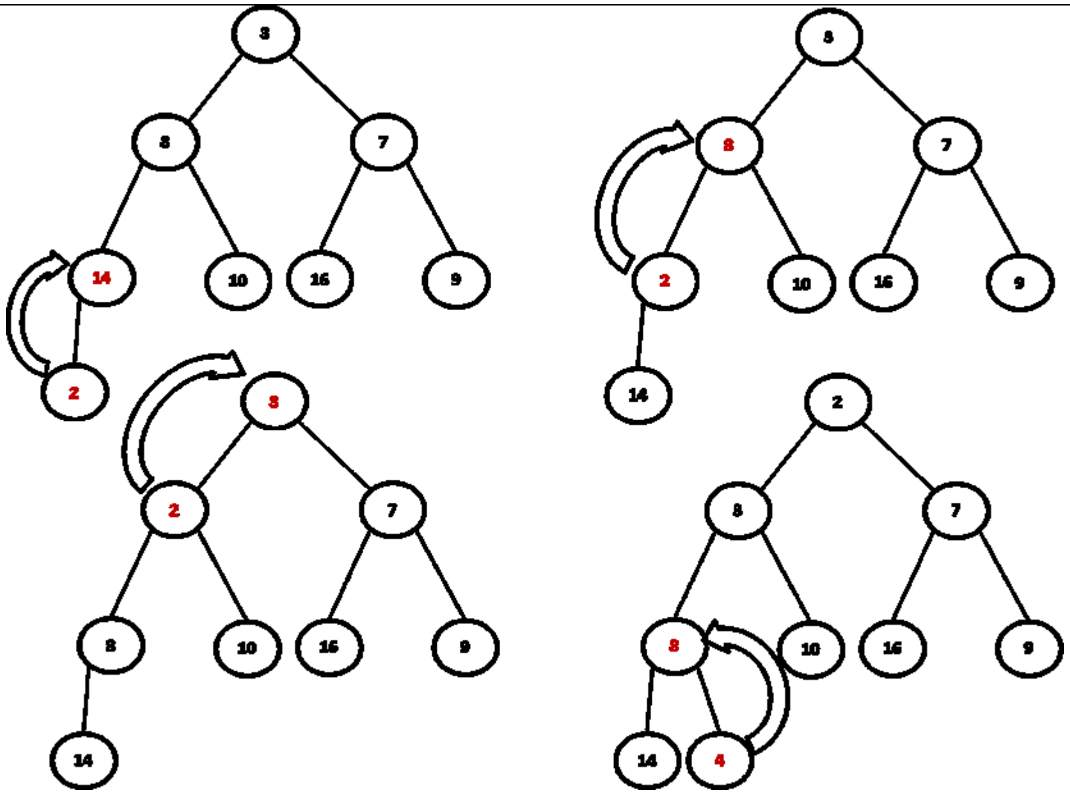
h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

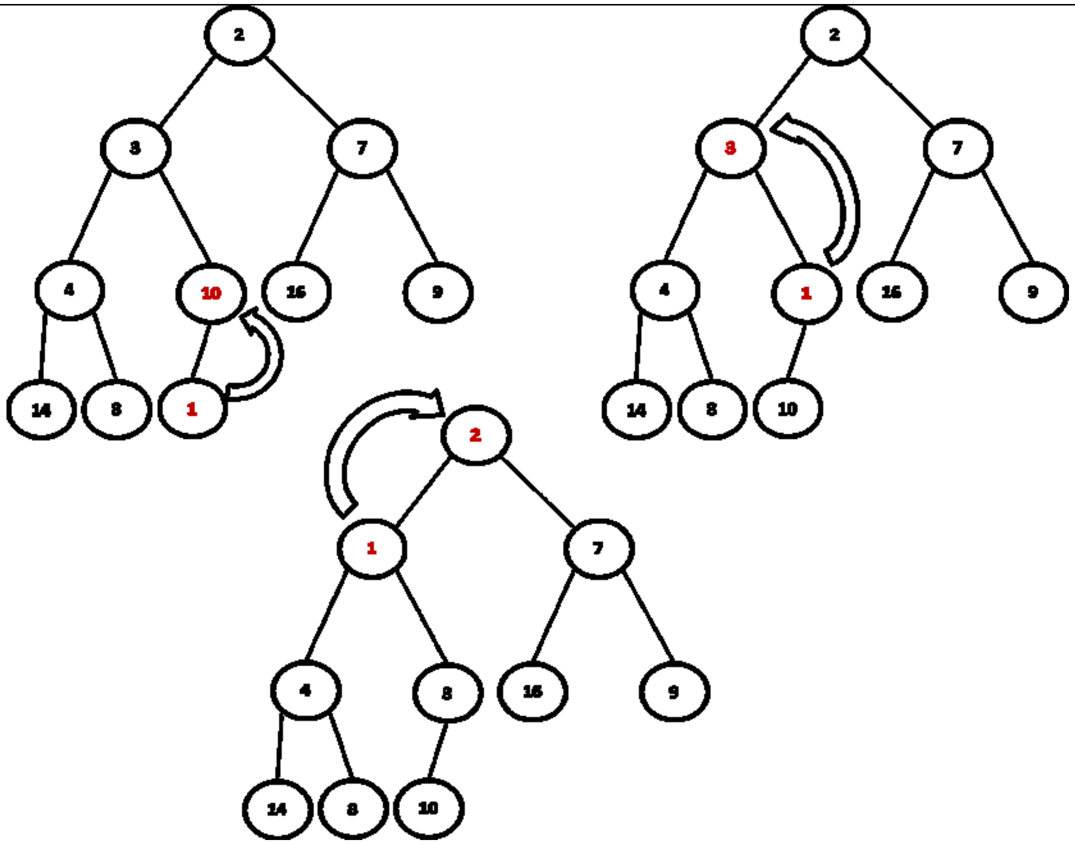
PRINT: E STACK:

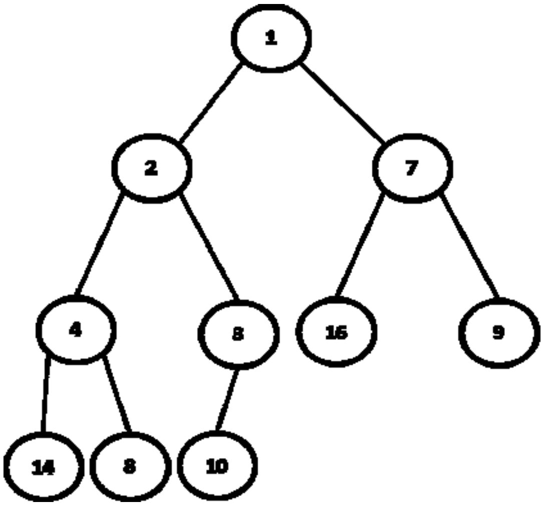
<table>
<tr><td></td><td>(b)</td><td>Explain the concept heap sort, and sort the following number using heap sort.
(Use Min and Max method to sort)
16, 14, 10, 8, 7, 9, 3, 2, 4, 1</td><td>8M</td></tr>
<tr><td></td><td>Ans:</td><td>Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts:
Creating a Heap of the unsorted list.
Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.
Given,
<table><tr><td>16</td><td>14</td><td>10</td><td>8</td><td>7</td><td>9</td><td>3</td><td>2</td><td>4</td></tr></table>
The given numbers are already on max-heap.


Max-Heap
So we need to calculate min-heap.</td><td>(Explanation :2 marks, Sorting: 6 marks)</td></tr>
</table>


16
14
10
10
14
16
10
14
16
8
10
8
16
14
8
10
16
14
7
8
7
16
14
10
7
8
16
14
10
9
7
8
9
14
10
16
7
8
9
14
10
16
8
7
8
8
14
10
16
9


3
8
7
14
10
16
9
2
8
8
7
2
10
16
9
14
8
2
7
8
10
16
9
14
2
8
7
8
10
16
9
14
4





Final Min-Heap

1	**2**	**7**	**4**	**3**	**16**	**9**	**14**	**8**

(c) **Describe process scheduling with any one algorithm.** **8M**

Ans:

(Description: 4 marks , Algorithm: 4 marks)

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.

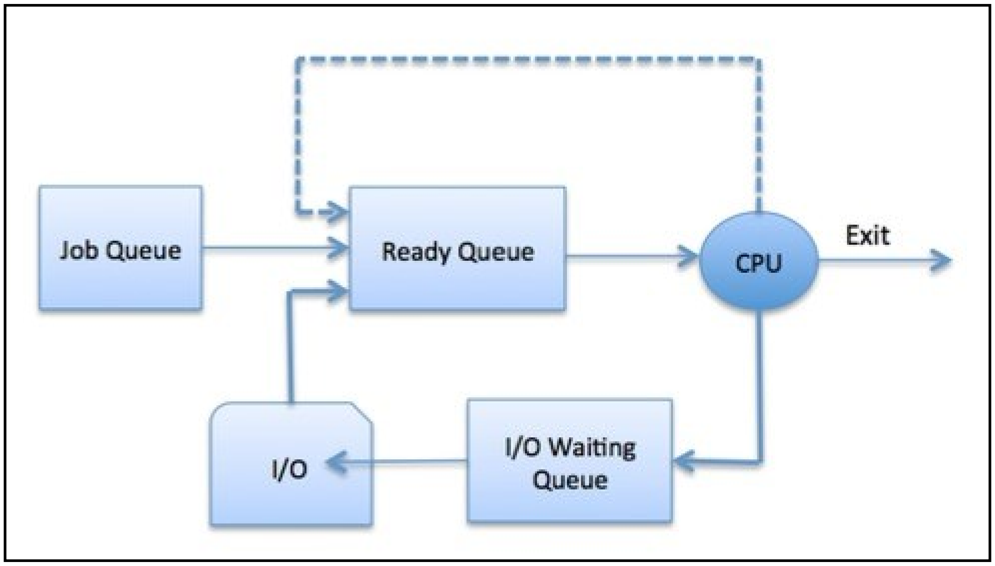
Process scheduling is an essential part of Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

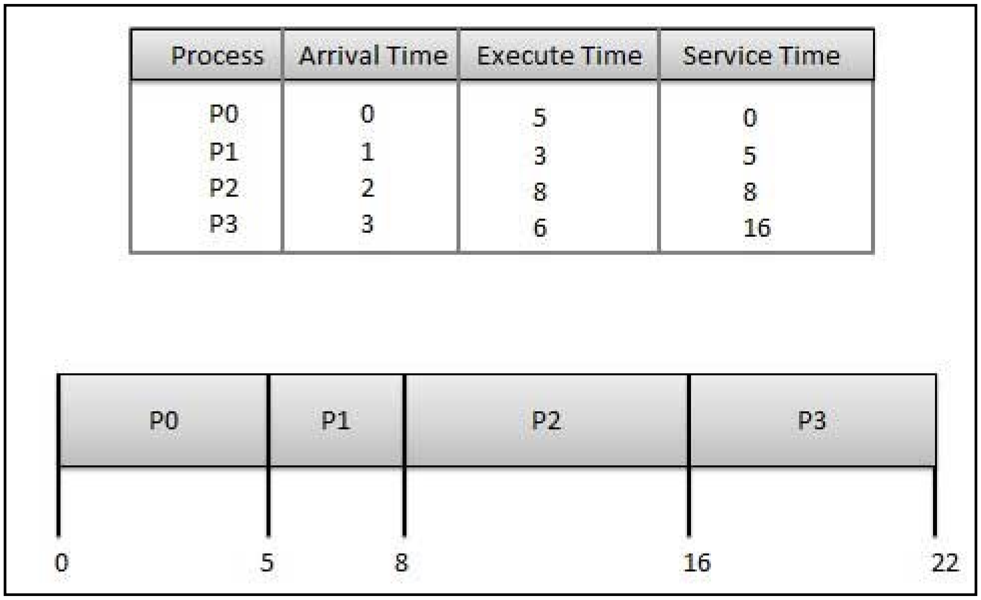
The Operating System maintains the following important process scheduling queues −

- **Job queue** − this queue keeps all the processes in the system.
- **Ready queue** − this queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** − the processes which are blocked due to unavailability of an I/O device constitute this queue.



First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.



Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16

Wait time of each process is as follows :

Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

3. **Solve any TWO:** **(2x8=16) Marks**

(a) **Explain Job scheduling in detail.** **8M**

Ans: Job scheduling is the process of allocating system resources to many different tasks by an operating system (OS).

(Job scheduling description: 4 marks, any relevant example: 4 marks)

The problem is stated as below.

There are n jobs to be processed on a machine. Each job i has a deadline $d_i \geq 0$ and profit $p_i \geq 0$ .Pi is earned if the job is completed by its deadline. The job is completed if it is processed on a machine for unit time. Only one machine is available for processing jobs. Only one job is processed at a time on the machine. A feasible solution is a subset of jobs J such that each job is completed by its deadline. An optimal solution is a feasible solution with maximum profit value.

Example : Let n = 4, $(p_1,p_2,p_3,p_4) = (100,10,15,27)$, $(d_1,d_2,d_3,d_4) = (2,1,2,1)$

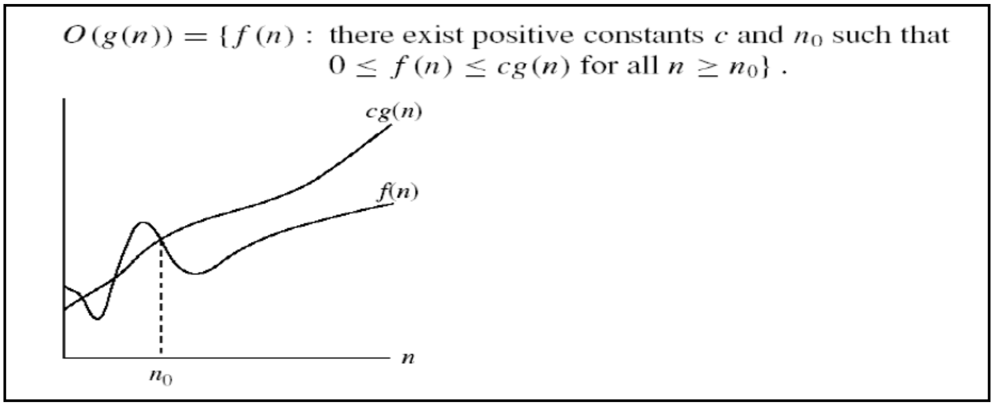
Sr.no	Feasible solution	Processing sequence	value
1	(1,2)	2,1	110
2	(1,3)	1,3 or 3,1	115
3	(1,4)	4,1	127 this is optimal one
4	(2,3)	2,3	25
5	(3,4)	4,3	42
6	1	1	100
7	2	2	10
8	3	3	15
9	4	4	27

(b) **What is big-Oh and theta? Write objectives of time analysis of algorithm.** **8M**

Ans: **Big-oh notation**:

(Big-Oh Explanation: 2 marks, Theta Explanation: 2 marks, Definition of time Analysis: 2 marks, Example: 2 marks)

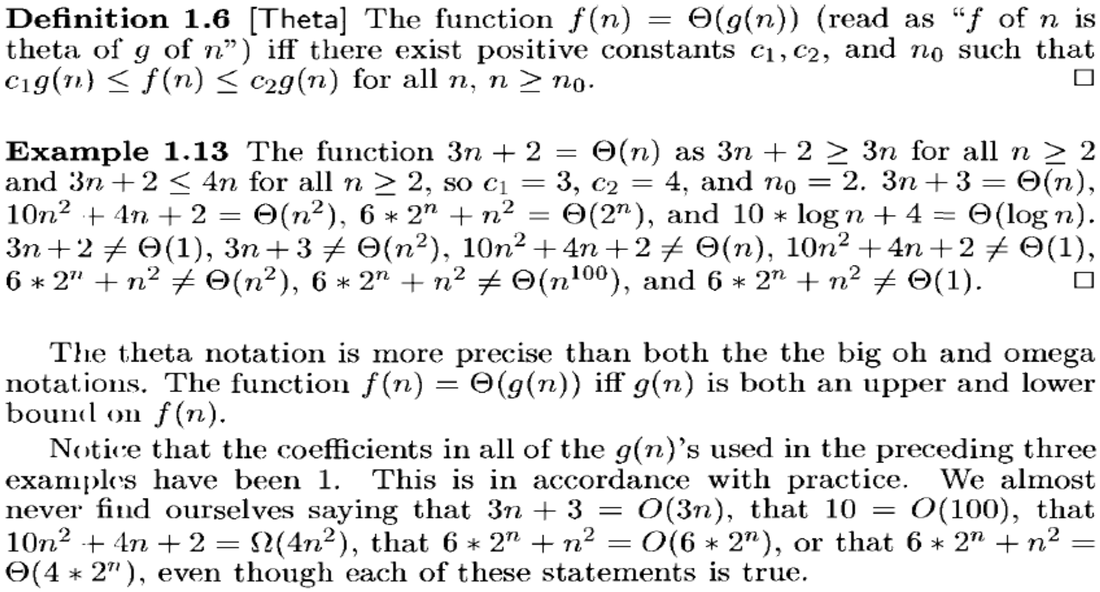
Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time it could possibly take for the Algorithm to complete. More formally, for non-negative functions, f(n) and g(n), if there Exists an integer n0 and a constant c>0 such that for all integers n>n0 .



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

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

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, $n \geq n_0$. $\square$

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 $6 * 2^n + n^2 \neq \Theta(1)$. $\square$

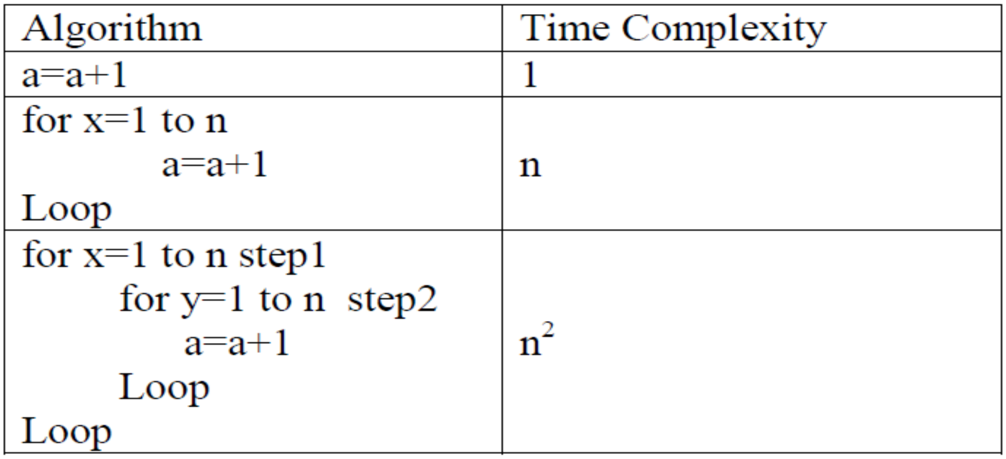
The theta notation is more precise than both the the big oh and omega notations. The function $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper and lower bound on $f(n)$.

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.

Time complexity:

Time complexity of an algorithm is the amount of computer time required to execute an algorithm. 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.

Example:



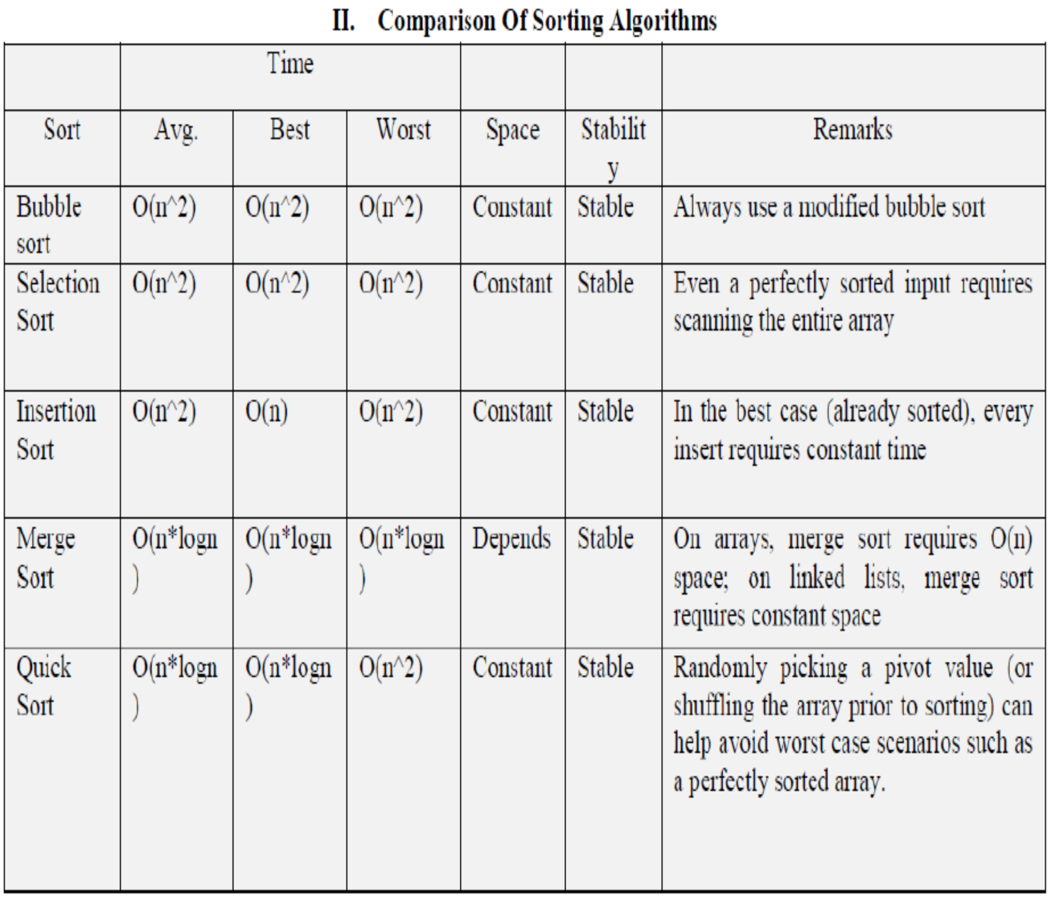
Algorithm	Time Complexity
a=a+1	1
for x=1 to n a=a+1 Loop	n
for x=1 to n step1 for y=1 to n step2 a=a+1 Loop Loop	n^2

(c) Compare any two searching and sorting algorithm **8M**

Ans:

Binary search	**Linear search**
Binary search requires the input data to be sorted	linear search doesn't requires the input data to be sorted
Binary search requires an ordering comparison;	linear search only requires equality comparisons
Binary search has complexity O(log n);	linear search has complexity O(n)
Binary search requires random access to the data	linear search only requires sequential access
Binary search is considered to be a more efficient method that could be used with large lists.	Linear search is too slow to be used with large lists due to its o(n) average case performance.

(Difference Between Binary and Linear Search: 4 Marks, 1 Mark for 1 Point)



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.

(Any two sorting algorithms can be considered: 4 marks, any 4 points)

4.		**Solve any TWO:**	**(2x8=16) Marks**
	(a)	**Explain quick sort algorithm with suitable example. Also explain time complexity of quick sort algorithm.**	**8M**

Ans:

Algorithm 5 Quick Sort

```
procedure QUICK SORT($list, start, end$)
    if $start<end$ then
        $index = PARTITION(list, start, end)$
        $QUICKSORT(list, start, index - 1)$
        $QUICKSORT(list, index - 1, end)$
    end if
end procedure
```

```
procedure PARTITION($list, start, end$)
    $pivot = list[end]$
    $key = start$
    for $i = start, end - 1$ do
        if $list[i] \leq pivot$ then
            $swap(list[i], list[key])$
            $key + +$
        end if
    end for
    $swap(list[key], list[end])$
    return $key$
end procedure
```

Quick sort uses divide and conquer approach for solving problems. Quick sort is quite similar to merge sort. It works by selecting elements from unsorted array named as a pivot and split the array into two parts called sub arrays and reconstruct the former part with the elements smaller than the pivot and the latter with elements larger than the pivot. This operation is called as partitioning. The algorithm repeats this operation recursively for both the sub arrays. In general, the leftmost or the rightmost element is selected as a pivot. Let us consider an array of elements A [3, 7, 8, 5, 2, 1, 9, 5, 4] and sorting the array into ascending order using quick sort. We use divide and conquer approach with recursive method.

(Explanation of quick sort algorithm:4 Marks, Example:2 Marks, Time complexity of quick sort:2 Marks)

Example:

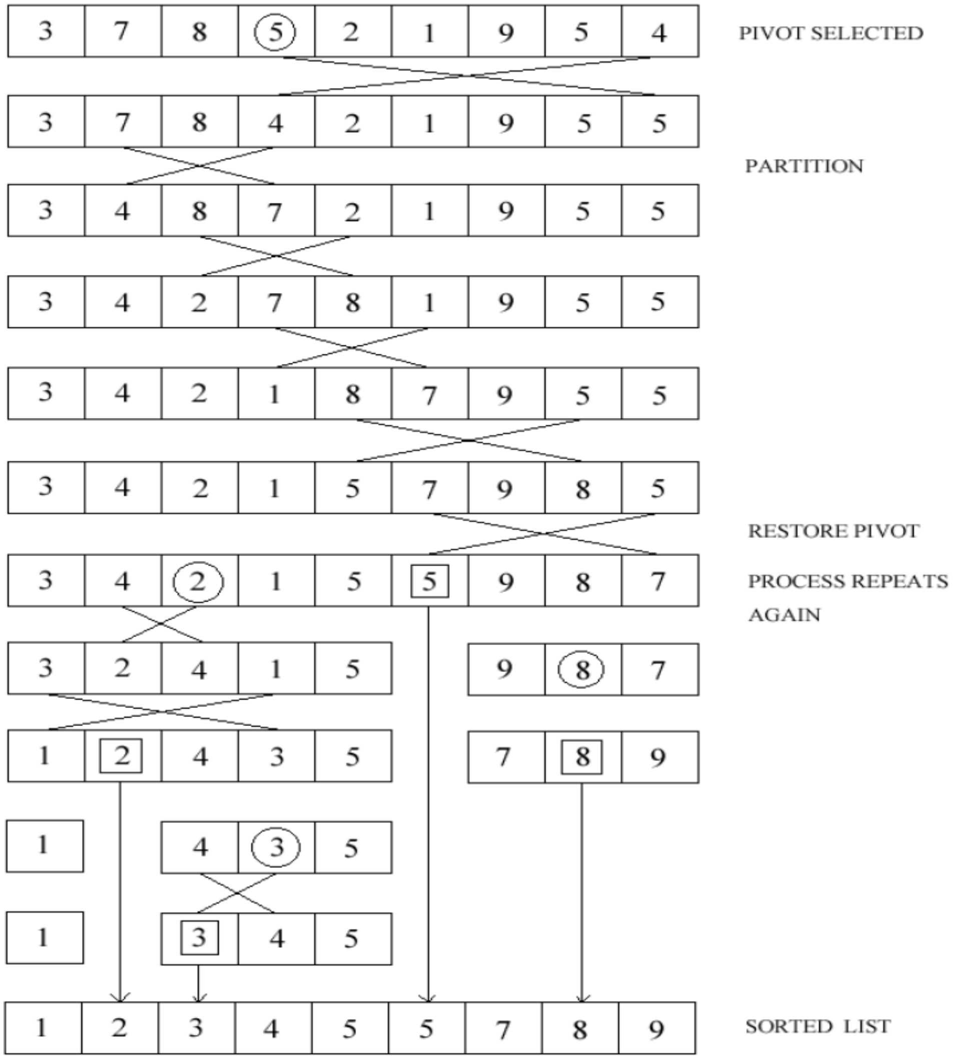


Fig: Quick Sort

Time complexity of quick sort:

Quick sort is the fastest sort on the average running time complexity of O (n log n) When compared to other sophisticated algorithms. Usually, selecting the leftmost or rightmost element as a pivot causes the worst case running time of O (n2) when the array is already sorted. Likewise, it is not efficient if all the input elements are equal, the algorithm will take quadratic time O (n2) to sort an array of equal elements. However, these worst case scenarios are infrequent. There are more advanced version of quick sort are evolved with a solution to selecting pivot.

(b) **Solve the following problems:**
Consider 5 items along their respective weight and values.

8M

$I = I_1, I_2, I_3, I_4, I_5$

W = 5,10,20,30,40

V = 30,20, 100, 90, 100

The capacity of knapsack W = 60, obtain the solution for the above given knapsack problem.

Ans: Initially,

(Correct solved problem: 8 Marks)

Items	Wi	Vi
I1	5	30
I2	10	20
I3	20	100
I4	30	90
I5	40	160

Taking value per weight ratio i.e Pi=Vi/Wi

Items	Wi	Vi	Pi=Vi/Wi
I1	5	30	6
I2	10	20	2
I3	20	100	5
I4	30	90	3
I5	40	160	4

Now arrange the value of Pi in descending order,

Items	Wi	Vi	Pi=Vi/Wi
I1	5	30	6
I3	20	100	5
I5	40	160	4
I4	30	90	3
I2	10	20	2

Now fill the Knapsack according to decreasing value of Pi.
First we choose item I1 whose weight is 5.
Now the remaining capacity of Knapsack =55.
Next we choose item I3 with weight of 20.
After inserting I3, remaining capacity of Knapsack =35,
Now insert I5 with weight of 40, but remaining capacity =35, so we can insert only 35 out of 40.so we choose fractional part of item I5.the value of fractional part of I5=(160/40*35)=140.
Thus the maximum value obtained=30+100+140=270.

Resultant vector=

I1	I2	I3	I4	I5
1.0	0.0	1.0	0.0	.87

(c) Explain Lower Bonds for Comparison Based sorting with appropriate example **8M**

Ans: A comparison based sorting algorithm sorts objects by comparing pairs of them. Example selection sort, merge sort are comparison based algorithms.

(Description of lower bound: 4 marks, Any Relevant Example: 4 marks)

In lower bound the goal is to prove that any algorithm must take time $\Omega(g(n))$ time to solve the problem, where now our goal is to do this for g(n) as large as possible. Lower bounds help us understand how close we are to the best possible solution to some problem: e.g., if we have an algorithm that runs in time O(n log2 n) and a lower bound of Ω(n log n), then we have a log(n) “gap”: Often, we prove lower bounds in restricted models of computation, that specify what types of operations may be performed on the input and at what cost. So, a lower bound in such a model means that if we want to do better, we would need somehow to do something outside the model. Sorting algorithms only operate on the input array by comparing pairs of elements and moving elements around based on the results of these comparisons. In particular, let us make the following definition.

Definition: A comparison-based sorting algorithm takes as input an array [a1,a2,... ,an] of n items, and can only gain information about the items by comparing pairs of them. Each comparison ("is ai > aj?") returns YES or NO and counts a 1 time-step. Reorder items based on the results of comparisons made. In the end, the algorithm must output a permutation of the input in which all items are in sorted order. For instance, Quicksort, Merge sort, and Insertion-sort are all comparison-based sorting algorithms

Theorem: Any deterministic comparison-based sorting algorithm must perform Ω(n log n) comparisons to sort n elements in the worst case. Specifically, for any deterministic comparison-based sorting algorithm A, for all n ≥ 2 there exists an input I of size n such that A makes at least log2 (n!) = Ω(n log n) comparisons to sort I.

To prove this theorem, we cannot assume the sorting algorithm is going to necessarily choose a pivot as in Quicksort, or split the input as in Merge sort — we need to somehow analyze any possible (comparison-based) algorithm that might exist. The way we will do this is by showing that in order to sort its input, the sorting algorithm is implicitly playing a game of "20 questions" with the input, and an adversary by responding correctly can force the algorithm to ask many questions before it can tell what is the correct permutation to output.

Proof: Recall that the sorting algorithm must output a permutation of the input [a1,a2,... ,an]. The key to the argument is that (a) there are n! different possible permutations the algorithm might output, and (b) for each of these permutations, there exists an input for which that permutation is the only correct answer. For instance, the permutation [a3,a1,a4,a2] is the only correct answer for sorting the input [2, 4, 1, 3]. In fact, if you fix a set of n distinct elements, then there will be a 1-1 correspondence between the different orderings the elements might be in and the permutations needed to sort them. Given (a) and (b) above, this means we can fix some set of n! inputs (e.g., all orderings of {1, 2,... ,n}), one for each of the n! output permutations. let S be the set of these inputs that are consistent with the answers to all comparisons made so far (so, initially, |S| = n!). We can think of a new comparison as splitting S into two groups: those inputs for which the answer would be YES and those for which the answer would be NO.

Let's do an example with n = 3, and S as initially consisting of the 6 possible orderings of {1, 2, 3}: (123),(132),(213),(231),(312),(321).
Suppose the sorting algorithm initially compares the first two elements a1 and a2.
Half of the possibilities have a1 > a2 and half have a2 > a1. So, the adversary can answer either way and let's say it answers that a2 > a1. This narrows down the input to the three possibilities: (123),(132),(231).
Suppose the next comparison is between a2 and a3. In this case, the most popular answer is that a2 > a3, so the adversary returns that answer which removes just one ordering, leaving the algorithm with: (132),(231).
It now takes one more comparison to finally isolate the input ordering and determine the

correct permutation to output. Alternative view of the proof: Another way of looking at the proof we gave above is as follows. For a deterministic algorithm, the permutation it outputs is solely a function of the series of answers it receives (any two inputs producing the same series of answers will cause the same permutation to be output). So, if an algorithm always made at most k < lg(n!) comparisons, then there are at most 2 k < n! different permutations it can possibly output. In other words, there is some permutation it can't output. So, the algorithm will fail on any input for which that permutation is the only correct answer.

5. **Solve any TWO :** **(2x8=16) Marks**

(a) **Describe greedy method for job scheduling with deadlines profits using example.** **8M**

Ans: Given a set of 'n' jobs, where each job 'i' is associated with an integer deadline di>=0 and a profit Pi > 0. For any job 'I' the profit 'Pi' is earned if and only if the job is completed within its deadline. To complete the job, one has to process he job on a given single machine for a one unit of time. A feasible solution for this problem is to identify a subset J of jobs in such way that, each job must be completed by its deadline. The value of the feasible solution J is the sum of the profits of the job in the subset J i.e

$$\sum_{i \in j} Pi$$

There is need to find out the feasible solution which must be an optimal solution which gives maximum profit value. Therefore, in a job scheduling problem there is need to identify a subset of jobs which are scheduled in a proper order to give maximum profit during the processing of them.

(Explanation :4 Marks, Example: 4 Marks)

Example:

Let n=4 , four jobs are there having profits (p1,p2,p3,p4) = (100,10,15,27) with deadlines (d1,d2,d3,d4) = (2,1,2,1). The feasible solutions are defined in the following way:

Solution #	Feasible Solution	Processing Sequence	Profit Value
1	(1,2)	2,1	110
2	(1,3)	1,3 or 3,1	115
3	(1,4)	4,1	127
4	(2,3)	2,3	25
5	(3,4)	4,3	42
6	(1)	1	100
7	(2)	2	10
8	(3)	3	35
9	(4)	4	27
10	(2,4)	(2,4)	37

In the above listed feasible solutions, the solution number 3 is the optimal since only job number 1 and 4 are processed that gives the maximum profit value as the 127. In this feasible solution, first job number 4 must be processed which is followed by the processing of the job number 1. Thus the processing of job number 4 begins at time of zero units and as it is completed in 1 unit of time immediately job number one will be start and processing will be completed in 2 unit of time.

To obtain optimal solution by applying greedy approach, there is need to formulate an optimization measure to determine the strategy for defining proper sequencing of jobs for processing.

Suppose, in a first attempt the objective function is chosen as the sum of profit as an optimal solution which is shown in the following equation.

$$\sum_{i \in j} Pi$$

In this case, in job scheduling problem, the next job for processing will be selected that increases the sum of profit. To apply this strategy, there is required to sort the jobs in a non-decreasing order of profit Pi.

In example number 1: the jobs are sorted and shown in the following way:

Jobs	(J1, J2, J3, J4)	(J1, J4, J3, J2)
Profit	(100, 10, 15, 27)	(100, 27,15,10)
Deadline	(2, 1, 2,, 1)	(2, 1, 2, 1)

In this case,
If job J1 is added to feasible solution i.e J ={J1}, Profit = 100
If job J2 is added to J = {J1, J2} then , Profit = 100 + 27 = 127
So this sequence gives the maximum profit of 127 so it is the optimal solution.

(b) Explain the concept radix sort. Write a program to sort the series of number using radix sort. **8M**

Ans:

Radix Sort is a sorting algorithm that is useful when there is a constant "d" such that all the keys are d digit numbers. To execute Radix-Sort, for p=1 toward "d ", it sorts the numbers with respect to the p^{th} digit from the right using any linear time stable sort. Radix sort is sometimes used to sort records of information that are keyed by multiple fields.

The following procedure assumes that each element in the n element array A has d digits where digit 1 is the lowest-order digit and digit d is the highest order digit.

(Explanation : 4 Marks, Program:4 Marks, Program with other logic and other Language

should be Considered)

RADIX-SORT(*A*,*d*)

```
1. for i - 1 to d
2.  do use a stable sort to sort Array A on digit i
```

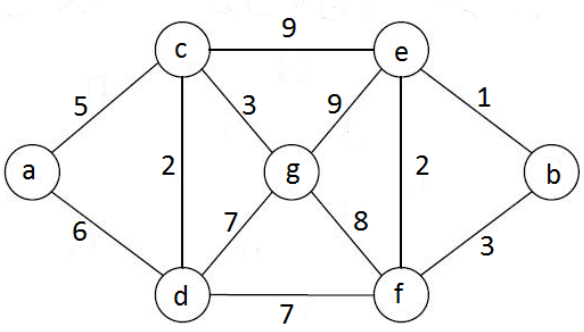
Since a linear time sorting algorithm is used „d“ times and d is a constant, the running time of Raix Sort is linear. When each digit is in the range 1 to *k*, and *k* is not too large, counting sort is the obvious choice. Each pass over *n d*-digit numbers takes (*n* + *k*)

Program to sort the elements by using Radix Sort

```
#include<stdio.h>

int getMax(int arr[], int n) {
    int mx = arr[0];
    int i;
    for (i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

void countSort(int arr[], int n, int exp) {
    int output[n]; // output array
    int i, count[10] = { 0 };

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using Radix Sort
void radixsort(int arr[], int n) {
    int m = getMax(arr, n);

    int exp;
    for (exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

void print(int arr[], int n) {
    int i;
```

```
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
}

int main() {
    int arr[] = { 170, 45, 75, 90, 802, 24, 2, 66 };
    int n = sizeof(arr) / sizeof(arr[0]);
    radixsort(arr, n);
    print(arr, n);
    return 0;
}
```

(c) **Design minimum spanning tree for given graph. Give the assumption to simulate the given graph with the help of prims algorithm.** **8M**



Ans:

Step 1: The start vertex is taken as " a"

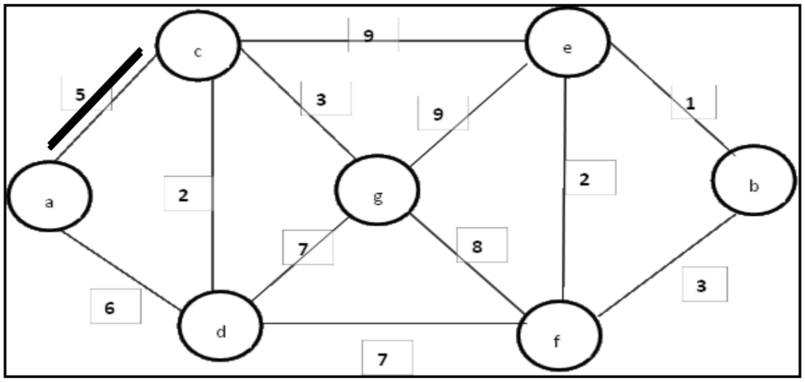
S={a}, V/S = {b, c, d, e, f, g}

Select the edge with minimum weight as a -> c = 5, so lightest edge ={a, c}.

OR

Suppose A vertex is the root i.e. r. by EXRACT-MIN(Q) procedure. Now u=r & Adj[a]={c.d}. Removing u from the set Q and adds it to the set V-Q of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in the tree. Key[c]=∞ W[a,c]=2 i.e., w(u,v)< Key[c] So, π[c]=0 & Key[c]=5 And Key[c]=∞

(Correct spanning Tree: 8 Marks)



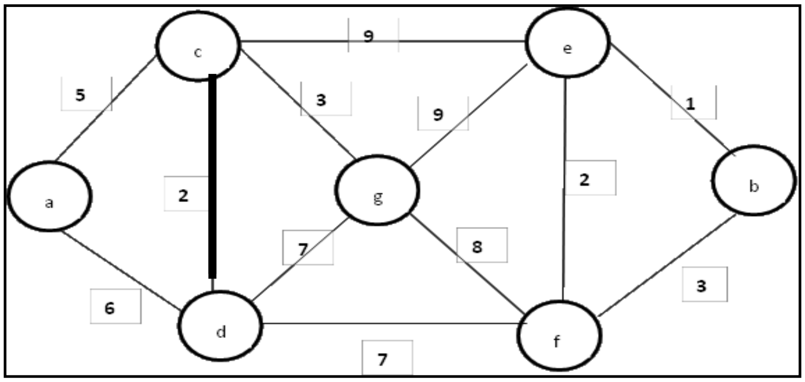
Step 2: The vertex 'c' is taken and from c there are three edges exist to reach to the destination vertex 'b' , which are c -> e = 9, c -> g = 3 and c->d = 2. With respective of this, c - > d edge is taken as it has minimum cost of 2.

S={a, c},

V/S = {b, d, e, f, g}

Select the edge with minimum weight as c -> d = 2,

so lightest edges ={ {a, c}, {c, d}}



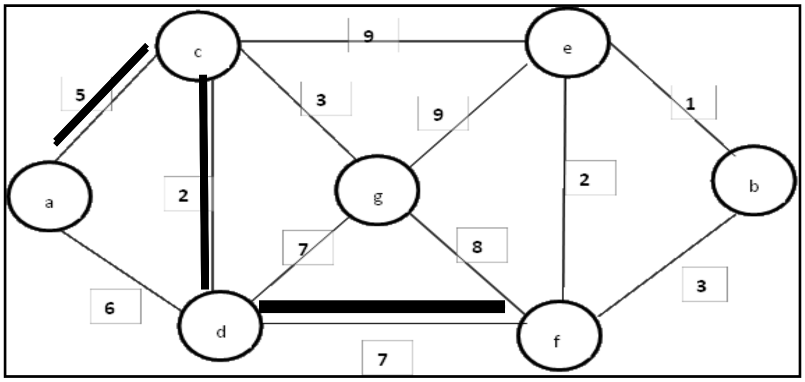
Step 3: The vertex 'd' is taken and from d there are two edges exist to reach to the destination vertex 'b' , which are d -> g = 7, d -> f = 7 with same cost. With respective of this, d - > f edge is taken as it has minimum cost of 7.

S={a, c, d, f},

V/S = {b, e, g}

Select the edge with minimum weight as d -> f = 7,

So lightest edges ={ {a, c}, {c, d}, {d, f}}



Step 4: The vertex 'f' is taken and from f there are two edges exist to reach to the destination vertex 'b' , which are f -> e = 2, f -> b = 3. With respective of this, f - > e edge is taken as it has minimum cost of 2.

S={a, c, d, e},

V/S = {b, f, g}

Select the edge with minimum weight as f - > e = 2

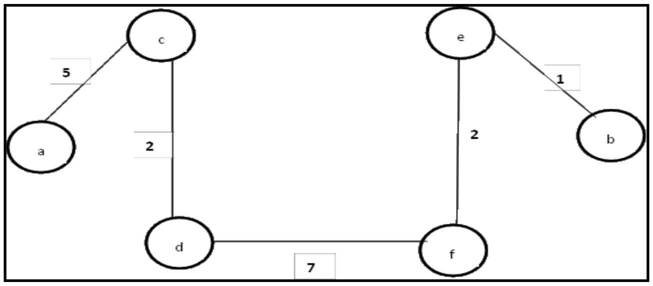
So lightest edges ={ {a, c}, {c, d}, {d, f}, {f, e} }

Step 4: The vertex 'e' is taken and from e there is single edge exist to reach to the destination vertex 'b', which is e -> b = 1. With respective of this, e -> b edge is taken as it has minimum cost of 1.

S={a, c, d, e, b},

V/S = { g}

Select the edge with minimum weight as e - > b = 1

		So lightest edges ={ {a, c}, {c, d}, {d, f}, {f, e}, {e, b} } **The minimum spanning tree is**  	
6.		**Solve any TWO:**	**(2x8=16) Marks**
	(a)	**Explain the following term:** **(i) Algorithm and its properties.** **(ii) Linear searching.**	**8M**
	Ans:	An algorithm is "a finite set of precise instructions for performing a computation or for solving a problem". To solve a problem step-by-step procedure need to follow in order to get a solution. An Algorithm is a self-contained step-by-step set of operations to be followed. So that, before developing a program always there is need to define the logic of the program in terms of the algorithm. **Properties of Algorithm** 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.	**(To Explain Algorithm: 2 Marks, to write four properties: 2 Marks)**

7. Generality: the algorithm *should* be applicable to all problems of a similar form.

(ii) Linear Searching

Explanation: Linear search, also called as sequential search, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted). For example, if an array A[] is declared and initialized as, int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};

(Explanation :2 Marks, Algorithm:2 Marks)

In the array, the value to be searched is 7, then searching means to find whether the value 7 is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).

Algorithm LINEAR_SEARCH (A, N, VAL)

```
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3: Repeat Step 4 while I<=N
Step 4: IF A[I] = VAL
            SET POS = I
            PRINT POS
            Go to Step 6
            [END OF IF]
[END OF LOOP]
Step 6: EXIT
SET I = I + 1
Step 5: IF POS = –1
PRINT VALUE IS NOT PRESENT
IN THE ARRAY
[END OF IF]
```

In Steps 1 and 2 of the algorithm, the value of POS and I are initialized.
In Step 3, a while loop is executed that till I is less than N (total number of elements in the array).
In Step 4, a check is made to see if a match is found between the current array element and VAL.
If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with VAL. However, if all the array elements have been compared with VAL and no match is found, then it means that VAL is not

present in the array.

(b) **Explain the following term:** **8M**

(i) Topological sorting

(ii) Graph representation

Ans: **Topological sort:** an ordering of the vertices in a directed acyclic graph, such that: If there is a path from u to v, then v appears after u in the ordering.

(Explanation 2 Marks, Algorithm Example: 2 Marks)

Topological sorting problem: given digraph G = (V , E) , find a linear ordering of vertices such that: for any edge (v , w) in E , v precedes w in the ordering.

Example: Consider three DAGs shown in Fig.1 and their possible topological sorts.

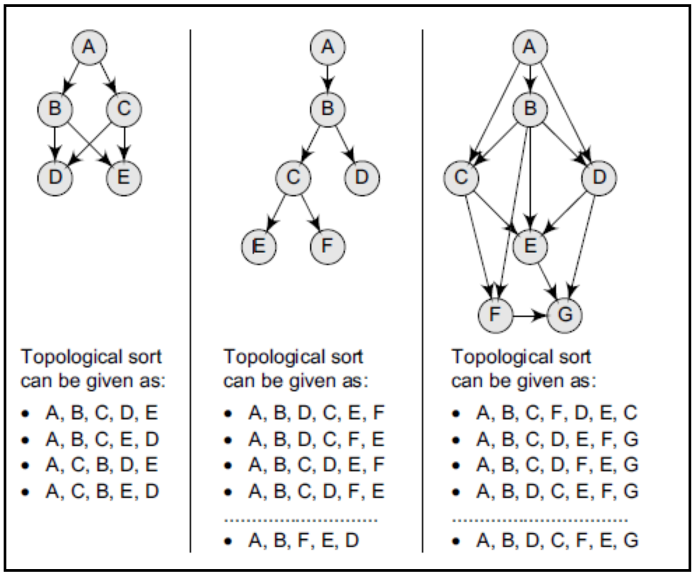


Figure No. 1

Algorithm

The algorithm for the topological sort of a graph (for figure number 1)that has no cycles focuses on selecting a node N with zero in-degree, that is, a node that has no predecessor. The two main steps involved in the topological sort algorithm include:

1. Selecting a node with zero in-degree
2. Σ Deleting N from the graph along with its edges

Algorithm Topological Sort

Step 1: Find the in-degree INDEG(N) of every node graph

Step 2: Enqueue all the nodes with a zero in-degree

Step 3: Repeat Steps 4 and 5 until the QUEUE is empty

Step 4: Remove the front node N of the QUEUE by setting

```
FRONT = FRONT + 1
Step 5: Repeat for each neighbour M of node N:
        a) Delete the edge from N to M by setting INDEG(M) = INDEG(M) - 1
        b) IF INDEG(M) = , then Enqueue M, that is, add M to the rear of the queue
[END OF INNER LOOP]
[END OF LOOP]
Step 6: Exit
```

ii) Graph representation

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

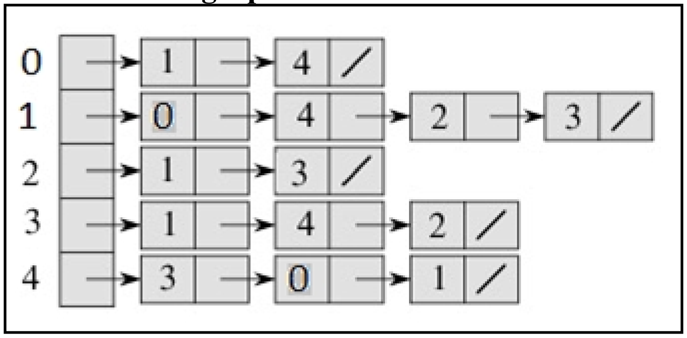
Sequential representation of graphs:

Two methods for sequential representation of graph:

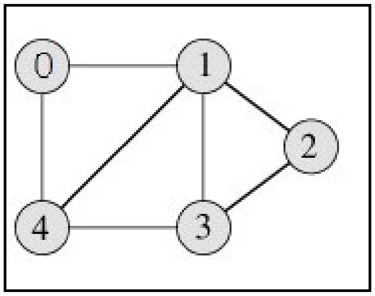
1) Adjacency matrix.

2) Linked representation.

1) Adjacency representation of graph



2) Linked representation of graph:



(To explain Graph and Graph Representation: Each Method of representation 2 Marks *2: 4 Marks)

	(c)	**Write an algorithm to illustrate the use of Binary search algorithm. Give an example.**	**8M**
	Ans:	Binary search algorithm is used to search a particular element in the list of 'n' number of elements. The primary condition for binary search is that, number must be in a sorted order.	**(Explanation with Algorithm: 6 Marks, and Example:2 Marks)**

Example:

Let a_i is an array, where $1\leq i \leq n$ containing list of elements in a non-decreasing sorted order. Consider a problem of determining whether a given element 'x' is present in the list or not. With respective of this problem, if element 'x' is present in the list then there is need to determine a value I such that ai = x. If 'x' is not present in the list, then i must be set to zero.

Let p=(n,ai,……………,al,x) denote an arbitrary instance of this search problem where:

n = number of elements in a search problem p

ai to l containing list of "n" number of elements

To search the element 'x' in the given list, Divide-and -Conquer method can be used in which if problem P containing more than one elements then P is divided into sub-problems. To divide the problem 'P' in number of sub-problems, the index 'q' is selected in the range [i , l] and x is compared with a_q . The three possible conditions are there

1) If x = aq : that is present at the first position of array a_i i.e x = aq, in this case problem 'P' is solved immediately.
2) If x < aq : in this case, x has to be search in a sub list ai, ai+1, ……., aq-1. The problem P is reduces to (qi-1, ai,…….,aq-1, x)
3) If x > aq : In this case, x will be searched from aq+1 to l. The problem P is reduces to (l-q, aq+1, ……., al,x)

Algorithm

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: Initialize
        SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3: SET MID = (BEG + END)/2
Step 4: IF A[MID] = VAL
                SET POS = MID
                PRINT POS
        Go to Step 6
        ELSE IF A[MID] > VAL
                SET END = MID - 1
```

```
        ELSE
                SET BEG = MID + 1
        [END OF IF]
[END OF LOOP]
Step 5: IF POS = -1
                PRINT “VALUE IS NOT PRESENT IN THE ARRAY”
        [END OF IF]
Step 6: EXIT
```