CMR INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | |

Internal Assesment Test - II

Sub:	Analysis and Design of Algorithms							Code:	13MCA41
Date:	08 / 05 / 2017	Duration:	90 mins	Max Marks:	50	Sem :	III	Branch :	MCA

Answer Any FIVE FULL Questions

		Marks	OBE	
			CO	RBT
1 (a)	Write the Horspool's algorithm for Enhancement in string matching. Give all the cases of Horspool algorithm and give its efficiency. Explain it for matching DEMO in string "THIS IS A DEMO FOR STRING MATCHING".	[10]	CO3	L3

Sol: Cases - 4x1 - 4M

Pseudocode - 3M

Example with steps - 3M

Horspool's algorithm is used for string matching and performs better than the brute force string matching by attempting the largest possible shift after every mismatch. this however, is done at the cost of extra storage which is a shift table maintained. While matching a string with the pattern the following four cases occur

Case 1: If there are no C's in the pattern, shift the pattern by its entire length to right.

e.g : S_0............S............S_{n-1}

BARBER

BARBER

Case 2: If there are occurrences of character 'c' in the pattern but the last one, then shift should align the right most occurrence of c in the pattern

S_0...............B............S_{n-1}

BARBE R

BARBER

Case 3: If C happens to be the last character in the pattern then there are no C's among its m-1 character, shift same as case 1.

S_0...........M E R...........S_{n-1}

LEA D E R

LEADER

Case 4: If C happens to be the last character in the pattern and there are other C's among its first n-1 characters the shift in same as case 2.

S_0.....................O R..........S_{n-1}

≠ ||

REORD E R

RE O R DER

Input enhancement makes repetitive comparisons unnecessary. Shift sizes are precomputed and stored in a table. The shift value is calculated by the formula:

t(c)=
- the pattern's length m, if c is not among the first m-1 characters of the pattern
- the distance from the rightmost c among the 1st m-1 characters of the pattern to its last character, otherwise

```
Algorithm Shifttable(p[0..m-1])

// Fills the table by Horspool's & Boya-Moore
// Input: pattern p[0..m-1] and an alphabet of possible characters
// Output: Table[0..size-1] indexed by the alphabet's characters and filled with shift
            sizes computed using t(c)
initialize all the elements of Table with m.
for j←0 to n-1 do Table[p[j]] ← m-1-j.
      return table.
```

```
Algorithm HorspoolMatching(P[0..m-1], T[0..n-1])
// Input: Pattern P[0..m-1] and text T[0..n-1]
// Output: The index of the left end of the first matching substring or -1 if there a
//          no matches
shift table(P[0..m-1])  // generates table of shifts
i←m-1                   // position of the pattern's right end
while i≤n-1 do
      k←0               // number of matched characters
      while i≤m-1 and P[m-1-k]=T[i-k]
            k←k+1
      if k=m
            return i-m+1
      else i←i+Table[T[i]]
return -1.
```

To search for the pattern DEMO in the text THIS IS A DEMO FOR STRING MATCHING, we first find the shift table for DEMO

Here n(length of string)=34 and m=4

Calculating the shift only for the first 3 characters:

Shift for D = m – 1 – I = 4-1-0=3

Shift for E = 4-1-1=2

Shift for M = 4-1-2=1

For all characters the shift table will have entries 4.

Thus the shift table is

A	B	C	D	E	…	M	N	O	…
4	4	4	3	2	4	1	4	4	4

Matching the pattern with text

T	H	I	S		I	S		A		D	E	M	O		F O R S T R I N G M A T C H I
D	E	M	O												Shift for 'S' is 4, so shift by 4 positions
				D	E	M	O								Shift for ' '(space) is 4, so shift by 4
								D	E	M	O				Shift for 'E is 2, so shift by 2 positions
										D	E	M	O		**Match Found !!**

2 (a) Write an algorithm for counting sort and analyze its performance. Show how it will perform to sort 12,22,12,34,33,11,22,22,34,12. [10] CO3 L1

Sol: Algorithm - 4M, Explanation - 1M, Analysis - 2M, Example - 3M

Counting sort algorithm is an example of a situation where using extra memory results in a efficient sorting technique. The idea is to count the frequencies of occurrence of each value in an array. Thus if the set of elements belong to a small range of numbers [l..u] then we can initially maintain a count of number of times each number occurs in an array F. Thus F[0] would store the frequency of occurance of l , F[1] would store the same of l+2 and F[u-l] would store the frequency of u occurring. The next step is to find the cumulative sum of elements in F. Thus F[i] would store the number of values in the original array which are less than the element associated with i, i.e. l+i. The elements of *A* whose values are equal to the lowest possible value *l* are copied into the first *F*[0]elements of *S*, i.e., positions 0 through *F*[0]– 1;the elements of value *l* + 1 are copied to positions from *F*[0] to (*F*[0]+ *F*[1]) – 1; and so on.. The algorithm for the same is given below:

```
Algorithm DistributionCounting (A[0..n-1])

// Sorts an array
// Input: A[0..n-1] of integers between l & u (l≤u)
// Output: S[0..n-1] of A's elements sorted in increasing order

for j←0 to u-l do D[j]←0                      // initialize frequencie
for i←0 to n-1 do D[A[i]-l]←D[A[i]-l]+1    // compute frequencies
for j←1 to u-l do D[j]←D[j-1]+D[j]          // reuse for distributic
for i←n-1 down to 0 do
        j ← A[i]-l
        S[D[j]-1] ← A[i]
        D[j] ← D[j]-1
return s
```

Step1: Calculating the frequencies

12,22,12,34,33,11,22,22,34,12.

The range is [12..34]

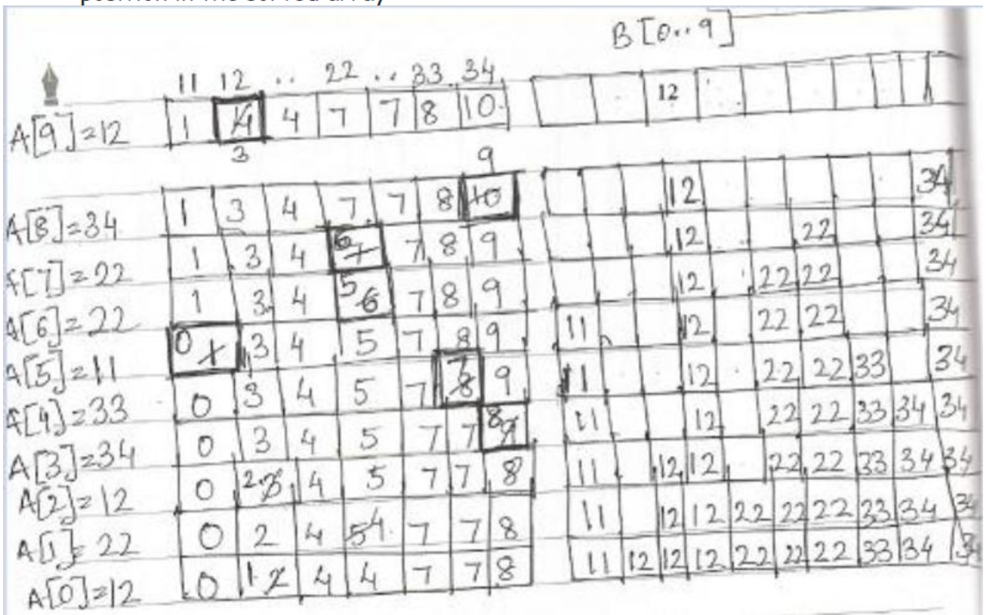
Hence finding the frequency of all elements

11	12	...	22	...	33	34
1	3	0	3	0	1	2

Step 2: Now finding cumulative frequency

11	12	...	22	...	33	34
1	4	4	7	7	8	10

Step 3: Starting from the end of the original array and inserting it in proper position in the sorted array



3 (a) Write pseudo code of the bottom-up dynamic programming algorithm for the knapsack problem and analyse it. [10] CO4, CO1 L2

Sol: Pseudocode - 5M, Explanation - 2M, Analysis - 3M

The knapsack problem is as follows: given n items of known weights w1,...,w n and values v1,...,v n and a knapsack of capacity M, find the most valuable subset of the items that fit into the knapsack.

The solution can be built recursively as follows: .Let us consider an instance defined by the first i items, 1≤i ≤n, with weights w1,...,wi, values v1,...,vi, and knapsack capacity j, 1≤j ≤M. Let F(i,j) be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j. The subsets of the first I items belong to two categories- one in which the ith item is present and the others in which the ith item is not present.

1. Among the subsets that do not include the ith item, the value of an optimal subset is, by definition, F(i−1,j).
2. Among the subsets that do include the ith item (possible only when ,j >= wi),an optimal subset is made up of this item and an optimal subset of the first i −1 items that fits into the knapsack of capacity j −wi. The value of such an optimal subset is vi +F(i−1,j−wi).

Thus the recursive solution is:

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

If i=0 or j=0 then F(I,j)=0 since it means no objects and no capacity respectively.

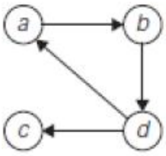
The recursive function aboe can be solved in a bottom up manner by solving the smallest of subprogblems first before solving the bigger ones. The algorithm for Knapsack is given below:

The recursive function aboe can be solved in a bottom up manner by solving the smallest of subprogblems first before solving the bigger ones. The algorithm for Knapsack is given below:

```
Algorithm Knapsack(n,w,c, M)
// n- number of items
// w- array containing weights of items from 0.. n-1
// c- array containing costs of items from 0.. n-1
// M - total capacity of knapsack

   Create a matrix F[0..n,0.. M]
  // initializing the matrix
 For I <- 0 to n
     F[i][0] <-  0

 For j <- 0 to M
     F[0][j] <-  0

 For i← 1 to n
     For j← 1 to M
    {
          F[i][j] ← F[i-1][j];
          If  j >=w[i] and c[i] +F[i-1][j-w[i]] > F[i-1][j]
                F[i][j] ← c[i] +F[i-1][j-w[i]]
    }

 Return F[n][M]
```

Analysis:
The first two loops in the algorithm take Θ(n) and Θ(M) respectively. The third nested loop has the outer loop running n times and the inner loop running M times for a total of Θ(nM) which is the complexity of the algorithm.

4 (a)	Find the transitive closure of the graph given below using Warshall's algorithm Sol: Steps (4Matrices) - 4 x1.5 = 6M The diagram below shows the working of Warshall’s on a given graph	[6]	CO2	L3



$R^{(0)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$

$R^{(1)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$

$R^{(2)} =$

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$R^{(3)} =$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$R^{(3)} =$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$R^{(4)} =$

	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

(b) Explain memory function with respect to knapsack problem. [4] CO4 L2

Sol: Memory function benefits - 2M, Knapsack explanation - 2M

Dynamic programming deals with problems whose solutions satisfy a recurrence relation with overlapping subproblems. The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient. The classic dynamic programming approach,on the other hand , works bottom up: it fills a table with solutions to all smaller subproblems, but each of them is solved only once.A problem with this approach is that solutions to some of these smaller subproblems are often not necessary for getting a solution to the problem given.Since this drawback is not present in the top-down approach, it is natural to try to combine the strengths of the top-down and bottom-up approaches. The goal is to get a method that solves only subproblems that are necessary and does so only once.Such a method can be done using memory functions.

This method solves a given problem in the top-down manner but, in addition, maintains a table of the kind that would have been used by a bottom-up dynamic programming algorithm. Initially, all the table's entries are initialized with a special"null" symbol to indicate that they have not yet been calculated. Thereafter, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first: if this entry is not "null," it is simply

	retrieved from the table; otherwise, it is computed by the recursive call whose result is then recorded in the table. For the following instance of knapsack problem :			

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$.

Only a few entries need to be calculated in the table constructed using the top down approach as shown below:

	i	capacity j 0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3, v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2, v_4 = 15$	4	0	—	—	—	—	**37**

Except for the base case the entire table is filled with nulls. We start by trying to calculate F(4,5) which requires F(3,3) and F(3,5). We store a null in all entries. Whenever an entry needs to be calculated we check if it already has a value. If yes we use the value, else we recursively compute it.

5 (a)	Write and explain the Floyd's algorithm for finding the All pairs shortest path and analyze its time complexity. Explain it with example.	[10]	CO1, CO6	L2,L4

Sol: Explanation of Floyd's algorithm-5M. Analysis - 2M, Example -4M

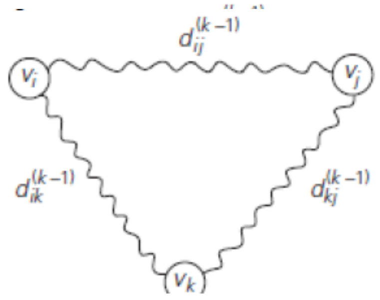
Sol: Given a weighted connected graph (undirected or directed), the ***all-pairs shortest paths problem*** asks to find the distances—i.e., the lengths of the shortest paths—from each vertex to all other vertices. This problem has a wide variety of applications in communication, transportation etc.

It is convenient to record the lengths of shortest paths in an $n \times n$ matrix D called the ***distance matrix***: the element d_{ij} in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex.

Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of $n \times n$ matrices:

$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}$.

The element d_{ij}^{k} in the *i*th row and the *j*th column of matrix $D^{(k)}$ *(i, j = 1, 2, . . . , n, k = 0, 1, . . . , n)* is equal to the length of the shortest path among all paths from the *i*th vertex to the *j*th vertex with each intermediate vertex, if any, numbered not higher than *k*.The series starts with $D^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $D^{(0)}$ is simply the weight matrix of the graph. The last matrix in the series, $D^{(n)}$, contains the lengths of the shortest paths among all paths that can use all *n* vertices as intermediate and hence is nothing other than the distance matrix being sought. Let d_{ij}^{k} be the element in the *i*th row and the *j*th column of matrix $D^{(k)}$. We can partition all paths between I and j into two disjoint subsets : those that do not use the *k*th vertex v_k as intermediate and those that do. Since the paths of the first subset have their intermediate vertices numbered not higher than *k* – 1, the shortest of them is of length d_{ij}^{k-1}.



Now if we introduce the kth vertex as an intermediate vertex, then it is possible that the path from vi to vj through vk may be shorter than the already existing shortest path. In such a case a new shortest path through k has been discovered and this may be recorded . However if the new path has a cost higher than an already existing path, this may be ignored. This can be expressed through the recursion:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1, \quad d_{ij}^{(0)} = w_{ij}.$$

Dynamic programming solution for the problem can be expressed as :

ALGORITHM *Floyd*($W[1..n, 1..n]$)

```
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W  //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

Analysis:

The basic operation in this case is the statement inside the innermost loop. Writing the number of times the basic operation is executed in terms of summation.

T(n) = $\sum_{k=1}^{n}\sum_{i=1}^{n}\sum_{j=1}^{n} 1 \models \sum_{k=1}^{n}\sum_{i=1}^{n} n = \sum_{k=1}^{n} n * n = n \times n \times n = n^3$

T(n) = $\Theta(n^3)$

Consider a graph whose adjacency matrix is given below:

6 0 3 2 ∞
∞ ∞ 0 4 ∞
∞ ∞ 2 0 3
3 ∞ ∞ ∞ 0

We start with D^0 initialized to the weight matrix. In general the matrix D^k will be a nxn matrix with $D^k_{ij} = d^k_{ij}$.

D^0 =

	a	b	c	d	e
a	0	2	∞	1	8
b	6	0	3	2	∞
c	∞	∞	0	4	∞
d	∞	∞	2	0	3
e	3	∞	∞	∞	0

Lengths of shortest path with no intermediate vertices. (D^0 is equal to weight matrix).

$D^{(1)}$ =

	a	b	c	d	e
a	0	2	∞	1	8
b	6	0	3	2	14
c	∞	∞	0	4	∞
d	∞	∞	2	0	3
e	3	5	∞	4	0

Length of shortest path with intermediate vertices not more than 1 (shortest path from b to e, e to b and e to d were discovered).

D^3 =

	a	b	c	d	e
a	0	2	5	1	8
b	6	0	3	2	14
c	∞	∞	0	4	∞
d	∞	∞	2	0	3
e	3	5	8	4	0

Length of shortest path with intermediate vertices not more than 3 (no new paths were discovered).

D^4 =

	a	b	c	d	e
a	0	2	3	1	4
b	6	0	3	2	5
c	∞	∞	0	4	7
d	∞	∞	2	0	3
e	3	5	6	4	0

Length of shortest path with intermediate vertices not more than 4 (new ~~edges~~ shortest path were added from a to c, a to e, b to e, c to e and e to c.

D^5 =

	a	b	c	d	e
a	0	2	3	1	4
b	6	0	3	2	5
c	10	12	0	4	7
d	6	8	2	0	3
e	3	5	6	4	0

Length of shortest path with intermediate vertices not more than 5 (ie: all vertices). (shortest paths added from c to a, c to b, d to a and d to b)

6 (a) Explain the Dijkstra's single source shortest path algorithms and analyze its time complexity. [10] CO3 L1

Sol: Explanation - 3M, Algorithm - 4M, Analysis - 3M

Sol: Dijkstra's algorithm is an algorithm for solving the single-source shortest-paths problem: for a given vertex called the source in a weighted connected graph with non negative edges, find shortest paths to all its other vertices. Some of the applications of the problem are transportation planning, packet routing in communication networks finding shortest paths in social networks, etc. First, it finds the shortest path from the source. to a vertex nearest to it, then to a second nearest, and so on. In general, before its ith iteration starts, the algorithm has already identified the shortest paths to i − 1 other vertices nearest to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree Ti of the given graph. The set of vertices adjacent to the vertices in T called "fringe vertices"; are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source. To identify the ith nearest vertex, the algorithm computes, for every fringe vertex u, the sum of the distance to the nearest tree vertex v and the length dv of the shortest path from the source to v and then selects the vertex with the smallest such d value. d indicates the length of the shortest path from the source to that vertex till that point. We also associate a value p with each vertex which indicates the name of the next-to-last vertex on such a path, . After we have identified a vertex u* to be added to the tree, we need to perform

- Move u^* from the fringe to the set of tree vertices.
- For each remaining fringe vertex u that is connected to u^* by an edge of weight $w(u^*, u)$ such that $d_{u^*} + w(u^*, u) < d_u$, update the labels of u by u^* and $d_{u^*} + w(u^*, u)$, respectively.

two operations.

The psuedocode for Dijkstra's is as given below:

```
ALGORITHM  Dijkstra(G, s)
    //Dijkstra's algorithm for single-source shortest paths
    //Input: A weighted connected graph G = ⟨V, E⟩ with nonnegative weights
    //         and its vertex s
    //Output: The length d_v of a shortest path from s to v
    //           and its penultimate vertex p_v for every vertex v in V
    Initialize(Q)    //initialize priority queue to empty
    for every vertex v in V
        d_v ← ∞;   p_v ← null
        Insert(Q, v, d_v)   //initialize vertex priority in the priority queue
    d_s ← 0;   Decrease(Q, s, d_s)   //update priority of s with d_s
    V_T ← ∅
    for i ← 0 to |V| − 1 do
        u* ← DeleteMin(Q)   //delete the minimum priority element
        V_T ← V_T ∪ {u*}
        for every vertex u in V − V_T that is adjacent to u* do
            if d_u* + w(u*, u) < d_u
                d_u ← d_u* + w(u*, u);   p_u ← u*
                Decrease(Q, u, d_u)
```

Analysis:

The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself.

Graph represented by adjacency matrix and priority queue by array:

In loop for initialization takes time |V| since the insertion into the queue would just involve appending the vertices at the end(since it is an array implementation). For the second loop, the loop runs |V|

	times. Each time the DeleteMin operation would take a maximum of Θ(|V|) time since it would involve finding the vertex in the array with min d value, for a total time of |V|2. The for loop (for iupdating the neighbor vetices) would run |V| times again. However the Decrease would take Θ(1) time because the index of the vertex would be known. Thus the total time complexity is Θ(|V|2). Graph represented by adjacency list and priority queue by binary heap: All heap operations take Θ(lg|V|) time. Thus the first loop runs |V| times and each time the Insert would take Θ(lg|V|) time. The second loop runs |V| times and the DeleteMin would again take lg|V| time. Thus the total number of time DecreaseMin would run across all iterations is Θ(Vlg|V|). In the second loop the basic operation is Decrease(Q,u,du) whoch is run the maximum number of times. Across all iterations using adjacency list, since for each vertex Decrease is called for a maximum of all its adjacent vertices, the number of times Decrease is invoked |E| times. For each time it is onvoked , it takes O(lg|V|) time to execute. Thus the total time complexity is Θ((|E|+|V|)lg|V|). Graph represented by adjacency list and priority queue by fibonacci heap: The time taken in this case Θ(|E|+|V|lg|V|).			
7 (a)	Outline Prim's algorithm for finding the minimal spanning tree of a graph. Analyze its time complexity. Sol: Explanation - 3M, Algorithm - 4M, Analysis - 3M Prim's algorithm is used for solving the minimal spanning tree problem. ***Spanning tree*** of an undirected connected graph is its connected acyclic subgraph(tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph. Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set *V* of the graph's vertices. On each iteration, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex(i.e. connected using the min weight) not in that tree. The algorithm stops after all the graph's vertices have been included in the tree being constructed. Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is *n* − 1, where *n* is the number of vertices in the graph.	[10]	CO2	L2,L3

The pseudocode of this algorithm is as follows.

ALGORITHM *Prim(G)*

```
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = ⟨V, E⟩
//Output: E_T, the set of edges composing a minimum spanning tree of G
V_T ← {v_0}   //the set of tree vertices can be initialized with any vertex
E_T ← ∅
for i ← 1 to |V| − 1 do
    find a minimum-weight edge e* = (v*, u*) among all the edges (v, u)
    such that v is in V_T and u is in V − V_T
    V_T ← V_T ∪ {u*}
    E_T ← E_T ∪ {e*}
return E_T
```

To implement Prim's algorithm we attach two labels to a vertex: the name of the nearest tree vertex and the length (the weight) of the corresponding edge. Vertices that are not adjacent to any of the tree vertices can be given the ∞ label indicating their "infinite" distance to the tree vertices and a null label for the name of the nearest tree vertex. With such labels,

finding the next vertex to be added to the current tree *T* =(VT,ET) becomes a simple task of finding a vertex with the smallest distance label in the set *V* − *VT*. Afterwehave identified a vertex u* to be added to the tree, we need to perform two operations:

Analysis:

Graph is represented by its weight matrix and the priority queue is implemented as an unordered array:

The algorithm's running time will be in _(|*V* |2). Indeed, on each of the |*V*| − 1

iterations, the array implementing the priority queue is traversed to find and delete the minimum and then to update, if necessary, the priorities of the remaining vertices

Graph is represented by its adjacency lists and the priority queue is implemented

as a min-heap,

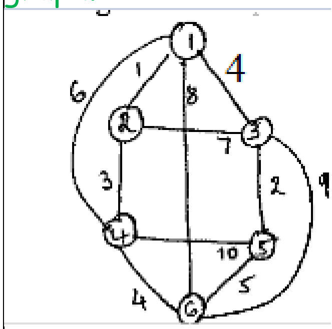
the running time of the algorithm is in *O*(|*E*| log |*V* |).

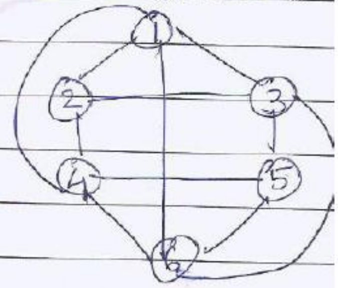
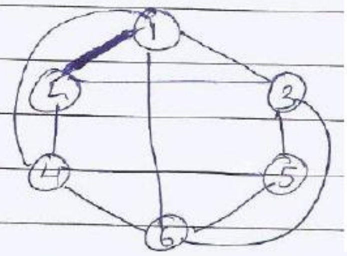
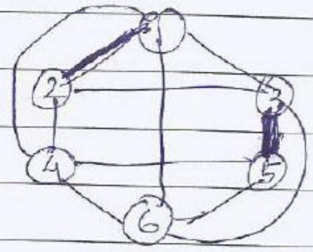
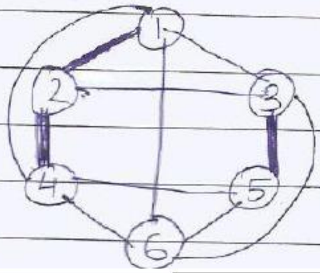
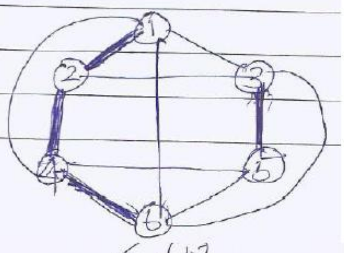
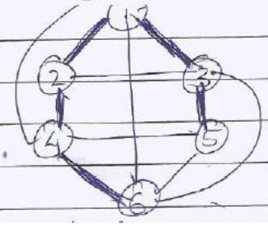
This is because the algorithm performs |*V*| − 1 deletions of the smallest element and makes |*E*| verifications and, possibly, changes of an element's priority in a min-heap of size not exceeding |*V* |. Each of these operations, as noted earlier, is a *O*(log |*V* |) operation. Hence, the running time of this implementation of Prim's algorithm is in (|*V*| − 1+ |*E*|)*O*(log |*V* |) = *O*(|*E*| log |*V* |) because, in a connected graph, |*V*| − 1≤ |*E*|.

Q	Question	Marks	CO	Level
8 (a)	Explain Kruskal's method to find the minimal spanning tree. How is it different from Prim's?	[5]	CO6	L4

Sol: Method - 3M, Difference (2 atleast) - 2M

Kruskal's algorithm is used for solving the minimal spanning tree problem. ***Spanning tree*** of an undirected connected graph is its

	connected acyclic subgraph(tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph. Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph $G = (V, E)$ as an acyclic subgraph with $\|V\| - 1$ edges for which the sum of the edge weights is the smallest. Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages of the algorithm. The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise. The differences between Prim;s and Kruskal's is: • During intermediate stages of tree construction the partial structure in csae of Prim's is always a tree whereas in Kruskal's it may be a forest. • Prim's algorithm is more appropriate when the |E| >> |V| else Kruskal's is more suitable. • In Prim's the next edge chosen is the minimum edge connecting a tree vertex with a vertex which is not in the tree whereas in Kruskal's the next chosen is the one with the minimum cost which does not cause a cycle.			
(b)	Apply Kruskal's algorithm to find minimum cost spanning tree for the graph.  Sol: Steps - 4M, Final Solution - 1M	[5]	CO3	L3

Tree edges	Remaining edges	Illustration
-	12 (1), 35 (2), 24 (3), 46 (4), 13 (4), 56 (5), 14 (6), 23 (7), 16 (8), 36 (9), 45 (10)	
12 (1)	35 (2), 24 (3), 46 (4), 13 (4), 56 (5), 14 (6), 23 (7), 16 (8), 36 (9), 45 (10)	
35 (2)	24 (3), 46 (4), 13 (4), 56 (5), 14 (6), 23 (7), 16 (8), 36 (9), 45 (10)	
24 (3)	46 (4), 13 (4), 56 (5), 14 (6), 23 (7), 16 (8), 36 (9), 45 (10)	
46 (4)	13 (4), 56 (5), 14 (6), 23 (7), 16 (8), 36 (9), 45 (10)	
13 (4)	56 (5), 14 (6), 23 (7), 16 (8), 36 (9), 45 (10)	

Hence the edges in the minimal spanning tree using Kruskal's is: 1-2, 2-4, 4-6,3-5 and 1-3, having cost of 1+2+3+4+4 = 14

Course Outcomes		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8
CO1:	Categorize problems based on their characteristics and practical importance.	4	2	1	0	0	2	2	0
CO2:	Understand the basic asymptotic notations and various efficiency classes.	4	3	4	0	0	0	3	0
CO3:	Compute the efficiency of algorithms in terms of asymptotic notations	2	4	1	0	0	0	1	0
CO4:	Design algorithm using an appropriate design paradigm for solving a given problem	4	3	4	0	0	0	3	0
CO5:	Classify problems as P, NP or NP Complete	2	4	1	0	0	2	1	0
CO6:	Implement algorithms using various design strategies and determine their order of growth.	3	3	4	0	0	1	2	0

PO1 – Apply *knowledge*; PO2 - *Problem analysis*; PO3 - *Design/development of solutions*; PO4 – team work ; PO5 – *Ethics* ; PO6 -Communication; PO7- *Business Solution*; PO8 – Life-long learning

Cognitive level	KEYWORDS
L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.