USN | | | | | | | | | |

Internal Assessment Test 2 – August 2023

Sub:	Design and Analysis of Algorithm				Sub Code:	21CS42	Branch:	AIML &AIDS
Date:	09/08/2023	Duration:	90 mins	Max Marks: 50	Sem/Sec:	4 A		OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1 a)	Design an algorithm to solve knapsack problem using dynamic programming. Apply the same to find an optimal solution to the knapsack instance, n = 4, W= 10 (Total Capacity) (p1, p2, p3, p4) = (42, 15, 20, 30) and (w1, w2, w3, w4) = (6, 4, 2, 5)	[10] Algo: 2 + sorted: 1 + formula: 1 + Matrix: 5 + initial sol: 1)	CO3	L3

Solution:

Algorithm DPknapsack (n, wt, p, M)

```
// input: n objects, array of weights wt, array of profits p, Max capacity M
// Output: Total Profit (Maximum value of items that fit in knapsack)
Begin
    V [n+1] [m+1];     // declare table
     for ( i = 0, i<=n; i++)
      {  for (w = 0, w<=M; w++)
          {      if( i == 0 || w == 0)      V[i][w] = 0;
                 else if (wt[i] <= w)
                          V[i][w] = max (V[i-1][w], V[i-1][w-wt[i]  + p[i])
                      else
                           V[i][w] = V[i-1][w];     }
          }
      Print V[n][w];
    // find the solution set
    i = n ;    j = M;
    while (i>0 && j>0)
     {    if (V[i][j] == V[i-1][j]
               print ( i = "0" ); i--;
          else
                print ( i = "1" ); i--; j= j-wt[i];
      }   // end while
End
```

1. Arrange the items according to increasing order of weights

Item	P_i	w_i	w / i	0	1	2	3	4	5	6	7	8	9	10
			0	0	0	0	0	0	0	0	0	0	0	0
3rd	20	2	1	0	0	20	20	20	20	20	20	20	20	20
2nd	15	4	2	0	0	20	20	20	20	35	35	35	35	35
4th	30	5	3	0	0	20	20	20	30	35	50	50	50	50
1st	42	6	4	0	0	20	20	20	30	42	50	62	62	62

V[i, w] = max { V[i-1, w], V[i-1, w-w[i]] + P[i]}

Total Profit = 20 + 42 = 62

		MARKS	CO	RBT
2 (a)	Write the control abstraction of greedy approach. Discuss advantages and disadvantages of greedy method.	[5] 2+3	CO3	L1

General Method:

Given n inputs choose a subset that satisfies some constraints.

A subset that satisfies that constraints is called a feasible solution

A feasible solution that maximizes or minimizes a given objective function is said to be optimal

Often it is easy to find the optimal solution.

```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
    solution := ∅; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

Advantages:

1. Very easy to implement
2. Can be applied to a wide range of problems in CS, Operation research, economics etc.
3. Typically have less time complexity.
4. Can solve problems in real-time, such as scheduling problems or resource allocation problems, because it does not require the solution to be computed in advance.
5. Often used as a first step in solving optimization problems, because they provide a good starting point for more complex optimization algorithms.
6. Can be used in conjunction with other optimization algorithms, such as local search or simulated annealing, to improve the quality of the solution.
7. Greedy algorithms are often faster than other optimization algorithms, such as dynamic programming or branch and bound, because they require less computation and memory.

Disadvantages:

1. The local optimal solution may not always be globally optimal.
2. Sensitive to small changes in the input, which can result in large changes in the output. This can make the algorithm unstable and unpredictable in some cases.
3. Relies heavily on the problem structure and the choice of criteria used to make the local optimal choice. If the criteria are not chosen carefully, the solution produced may be far from optimal.
4. May require a lot of preprocessing to transform the problem into a form that can be solved by the greedy approach.

(b) What is dynamic programming technique? Compare Dynamic Programming with Greedy Technique and Divide and conquer strategy. [5] 1+2+2 CO4 L1

Dynamic programming is a technique for solving problems with **overlapping subproblems**. Typically, these subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type. Dynamic programming suggests solving each smaller subproblem once and recording the results in a table from which a solution to the original problem can be then obtained.
It can be considered as both mathematical optimization technique and algorithmic paradigm. It is a general but powerful optimization technique.
Simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner. These sub-problems are solved and then re-used. This leads to concept of **Optimal Substructure**.

Dynamic Programming	**Divide and Conquer**
Divides a problem into multiple sub-problems and uses either the top-down or bottom -up strategy to solve problems	Uses the top-down approach for solving problems
Subproblems are overlapping	Subproblems are independent
Suitable for solving optimization problems	Suitable for solving non-optimization problems
Dynamic Programming	**Greedy Approach**
Useful for solving multistage optimization problems	Useful for solving optimization problems

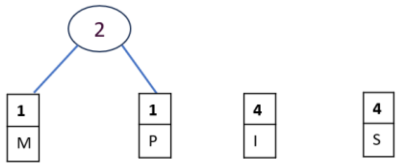
Generate multiple sequences of solutions for the given problem	Generates only one solution sequence
Definitely gives optimal solution (if exists)	May or may not give optimal Solution

3 (a) How many bits may be required for encoding the message "MISSISSIPI"? Use Huffman coding for encoding.

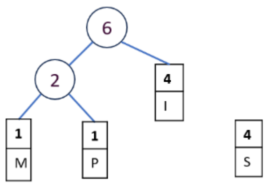
[5] Freq 1 + codeword 3 + bits 1 | CO3 | L3

Characters	**M**	**I**	**S**	**P**
Frequency	1	4	4	1

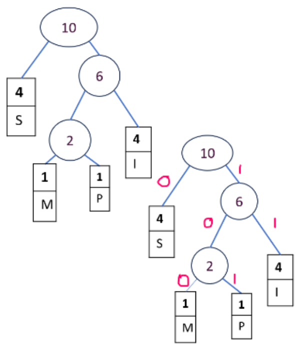
1. Sort in ascending order of frequencies {M, P, I, S} {1,1,4,4}
2. Pick least two frequencies



3. The combined frequency {2} replaces M and P. New list of frequencies is {2, 4, 4}.
4. Again pick two least frequencies
   6 = 2+ 4 (combined frequency)
5. 6 replaces I. New list of frequencies is {4, 6}
6. Pick least two frequencies



7. Assign 0 and 1 to the left and right child of each node.



Codewords:
M: 100
P: 101
I: 11
S: 0
Bits: 3*1 + 3* 1 + 2* 4 + 1* 4 = 3+3+8+4 = 18

(b) Define Heap. Write bottom-up heap construction algorithm and compute its efficiency.

[5] 1+3+1 | CO2 | L3

Heap can be defined as a binary tree with keys assigned to its nodes (one key per node) such that following two conditions are met:

1. **Tree's shape Requirement-** Binary tree is complete, that is, all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
2. **The parental dominance requirement-** the key at each node is greater than or equal to the keys at its children. (for max-heap)

BuildHeap(A, n)
{ for i = n/2 to 1
MaxHeapify(A, n, i)
}

MaxHeapify (A, n, i)
{
largest = i;

```
    lchild = 2*i;
    rchild = 2*i +1;
    while(lchild<= n && A[lchild] > A[largest])
            largest = lchild;
    while (rchild <= n && A[rchild]> A[largest])
            largest = rchild;
     if(largest != i)
       {swap (A[largest] , A[i])
         heapify(A, n, largest)
        }
  }
Heapsort (A, n)
{
    BuildHeap( A, n);
    // Delete the elements
   for (i = n; i> = 1; i --)
       Swap (A[1], A[i]);
       MaxHeapify( A, n, i);
}
```

Two steps:

Creation and Deletion

Insertion (Creation)

Inserting 1 element: Best case: O(1)

Worst Case: create an array of numbers. The number to be inserted is added as the last element of an array. To bring in to its place in a heap, few elements have to be swapped within an array. Comparing and swapping requires Time= O(n)

Deletion:

Deleting 1 element:

Best case: O(1) (only one element in heap)

Worst case: O(log n). Traversing down the tree height.

For n elements : O(n log n)

Total time complexity for Heap Sort: O(n) + O(n log n)

= O(n log n)

4 (a)	State the greedy strategy to solve the Job Sequencing with deadlines problem. Design an algorithm to solve the same and analyse the same.	[6]	CO3	L2

- The sequence of jobs on a single processor with deadline constraints is called as Job Sequencing with Deadlines.

Given an array of n jobs, Every Job is assigned a deadline $d_i >= 0$ *for any job I* , Every Job has an associated Profit $p_i >= 0$

Conditions:

1. Profit is earned if and only if the job is completed by its deadline.
2. Every Job takes a single unit of time for processing.
3. Only one machine (uniprocessor) is available for job.
4. Pre-emption is not allowed

Goal is to choose a subset of jobs such that the profit is maximized.

- Solution subset J of jobs such that each job in this subset can be completed by its deadline.
- Value of a feasible solution J is the sum of the profits of the Jobs in J
    - $\sum_{i \in J} p_i$

The greedy strategy to solve job sequencing problem is:

" At each time, select the job that satisfies the constraints and gives the maximum profit"

Algorithm GreedyJob (d, J, n)

// Input: Arrays of profit an deadlines for each job. n is the number of jobs

// Output: Set of Jobs J that can be completed by the deadlines.

Begin

Identify maximum number of timeslots.

Arrange the jobs in decreasing order of profits.

J = {1}; // initialize the solution set
for i =2 to n do
{ if <u>(all jobs in J U {i} can be completed by their deadlines)</u> then J = J U {i};
}
End

Note: For each Job (m_i) do linear search to find particular slot in array of timeslots.

<u>Analysis:</u>

1. Identify the maximum number of timeslots O(1)
2. Arrange the jobs in decreasing order O(n log n)
3. Do linear search to find particular slot in array of timeslots

If Number of Jobs is *m*

Maximum deadlines or number of jobs added to the solution set = n

Then, linear search takes O(n x m) time on an average

In worst case, m <<< n , The time take is $O(n^2)$.

Thus, the total time taken = O(1) + O(n logn) +O (n^2) = $O(n^2)$

(b) Consider the below table for Jobs given with profit and deadline. Find the maximum profit earned. [4] timeslots 1 +3 | CO3 | L3

Job	J1	J2	J3	J4	J5	J6	J7	J8	J9
Profit	15	20	30	18	18	10	23	16	25
Deadline	7	2	5	3	4	5	2	7	3

1. Identify the maximum number of timeslots required
   = min (n, max (d[ ])) where, n= number of jobs
   d[ ] = array of deadlines
   = min (9, 7) = 7

Arrange the Jobs in decreasing order of profit

Job	**Slot Assigned**	**Solution**	**Profit**
-	-	ϕ	-
J3	[4,5]	J3	30
J9	[2,3][4,5]	J9, J3	30 + 25 = 55
J7	[1,2] [2,3][4,5]	J7, J9, J3	55 + 23 = 78
J2	[0,1][1,2][2,3][4,5]	J2, J7, J9, J3	78 + 20 = 98
J4	[0,1][1,2][2,3][4,5]	J2, J7, J9, J3	98
J5	[0,1][1,2][2,3][3,4][4,5]	J2, J7, J9, J5, J3	98 + 18 = 116
J8	[0,1][1,2][2,3][3,4][4,5][6,7]	J2, J7, J9, J5, J3, 8	116 + 16 = 132
J1	[0,1][1,2][2,3][3,4][4,5][5,6][6,7]	J2, J7, J9, J5,J3, J1, J8	132+15 = 147
J6	[0,1][1,2][2,3][3,4][4,5][5,6][6,7]	J2, J7, J9, J5,J3, J1, J8	132+15 = 147

Solution Set: J ={J2, J7, J9, J5,J3, J1, J8}
Profit = 147

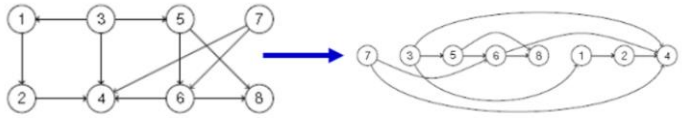
5 (a) What is topological sorting? Apply the same to the below graph using a) DFS algorithm b) source removal method. [10] 2+4+4 | CO2 | L3

Scheduling a sequence of jobs or tasks based on their dependencies.

Jobs ---- Vertices Edge (x to y) ------x should be completed before y

E.g: When washing clothes, the washing machine must finish washing before we can put these clothes in a dryer.

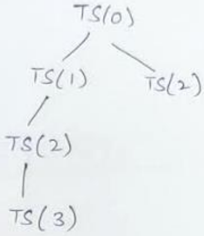
Given a directed graph G = (V, E) a topological sort of G is a linear ordering of V such that for any edge (u, v), u comes before v in the ordering.



DFS Method

1) Call TS(0). Source = 0.

2) Call TS(1)
Call TS(2)
Call TS(3)
No adj nodes to 3.
No more recursive call
∴ visited [3] = True
Push 3 on Stack



Stack: | 3 |

3) Go 1 step back.
No ~~ad~~ unvisited adj nodes of 2.
∴ No more recursive call
∴ visited [2] = True
Push 2 on stack

Stack | 3 | 2 |

4) Go 1 step back.
No unvisited adj nodes of 1
visited [1] = True
Push 1 on stack

Stack: | 3 | 2 | 1 |

5) Go 1 step back
No unvisited adj nodes of 0
visited [0] = True
Push 0 on stack

Stack | 3 | 2 | 1 | 0 |

6) Call TS(1) ; Already visited
Call TS(2) ; Already visited
Call TS(3) ; Already visited.
Call TS(4)
No adj nodes. No recursive call
visited [4] = True
Push 4 on stack

Stack | 3 | 2 | 1 | 0 | 4 |

7) Call TS(5).
Call TS (3) ; Already visited
Call TS (4) ; Already visited
No more unvisited adj nodes;
visited [5] = True
Push 5 on stack

Stack | 3 | 2 | 1 | 0 | 4 | 5 |

8) Call TS(6)
Call TS(1); Already visited
Call TS(5); Already visited
No more unvisited adj nodes
visited [6] = True
Push 6 on stack.

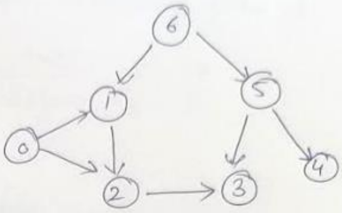
Stack: | 3 | 2 | 1 | 0 | 4 | 5 | 6 |

9) Pop Stack for topological sorted nodes.

6 → 5 → 4   0 → 1 → 2 → 3

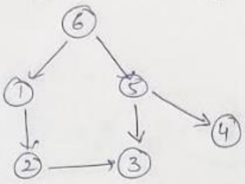
DFS Method Source Removal

1) Arrange the nodes according to increasing order of indegrees



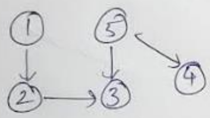
node	0	6	4	5	1	2	3
indegree	0	0	1	1	2	2	2

Step 2 Remove node (0) and all edges emanating from it.



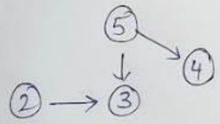
node	6	1	2	4	5	3
indegree	0	1	1	1	1	2

Step 3 Remove node (6) and all edges emanating from it.



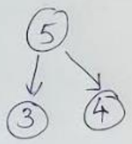
node	1	5	2	4	3
indegree	0	0	1	1	2

Step 4 Remove node (1) and all edges emanating from it.



node	2	5	4	3
indegree	0	0	1	2

Step 5 Remove node (2) and all edges emanating from it



node	5	3	4
indegree	0	1	1

Step 6 Remove (5) and all edges emanating from it

(3) (4)

node	3	4
indegree	0	0

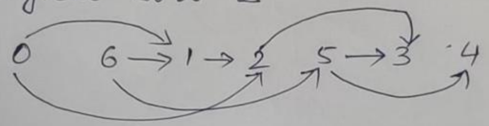
Step 7 Remove (3)

(4)

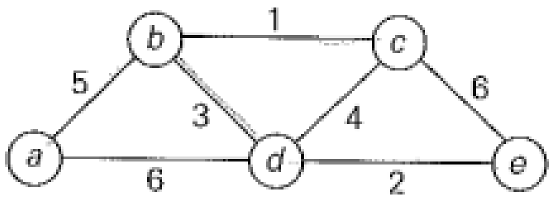
node	4
indegree	0

Step 8 Remove (4)

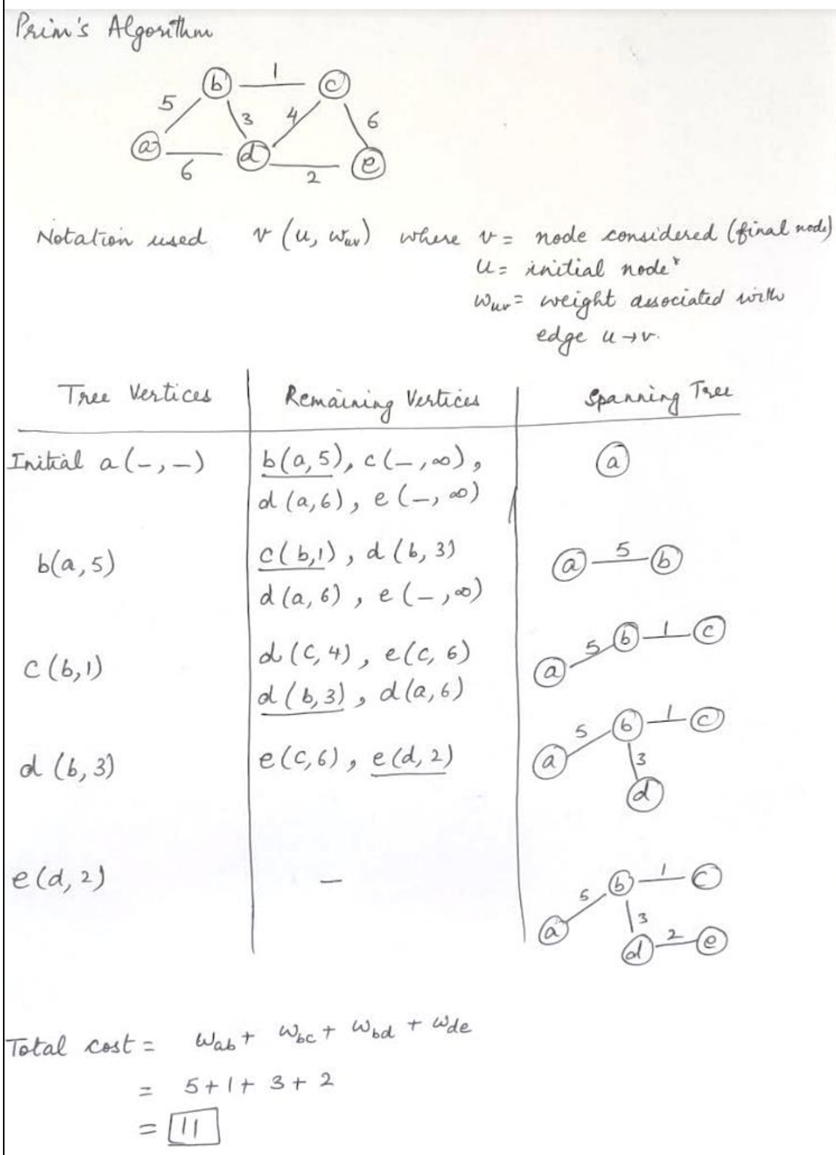
Topological Sort is



6 (a)	Write Prim's algorithm and apply it to obtain the minimum cost spanning tree of the following weighted graph:	[6]	CO3	L2



Algorithm Prim(G)
// Input: Graph G(V,E)
// Output: Minimum Spanning Tree T
Begin
s= pick up any vertex of G
$V_T = \{s\}$
$E_T = \Phi$ // Initially $T(V_T, E_T)$ has only starting vertex and no edges
n= |V|
repeat |n| - 1 times // n is number of vertices and T should have exactly n-1 edges
Pick an edge (v, u) such that v ϵ V_T and u ϵ V-V_T and there is no cycle
$V_T = V \cup \{v\}$
$E_T = E_T \cup \{v, u\}$
End repeat
return $T(V_T, E_T)$
End



Prim's Algorithm

Notation used $v(u, w_{uv})$ where v = node considered (final node)
u = initial node
w_{uv} = weight associated with edge u → v.

Tree Vertices	Remaining Vertices	Spanning Tree
Initial a(–,–)	b(a,5), c(–,∞), d(a,6), e(–,∞)	
b(a,5)	c(b,1), d(b,3) d(a,6), e(–,∞)	
c(b,1)	d(c,4), e(c,6) d(b,3), d(a,6)	
d(b,3)	e(c,6), e(d,2)	
e(d,2)	–	

Total cost = $w_{ab} + w_{bc} + w_{bd} + w_{de}$
= 5+1+3+2
= 11

(b)	Write Dijkstra's algorithm to find single source shortest path and analyse the algorithm.	[4] 3+1	CO2	L1

Approach: Greedy
Input: Weighted graph G=(V,E) and source vertex vϵ V, such that all edge weights are

nonnegative
Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex v϶ V to all other vertices.
Algorithm **Dijkstra (Graph, source)**
// Input: A weighted connected graph G= (V,E) with non negative weights and its vertex s
// Output: The length d(v) of a shortest path from s to v

Begin
Intialize (Q) ------------ *Creation of Priority Queue takes O(|V|) time*
for** (every vertex v in G)* ***do -----------***repeated |V| times***
d(v) = ∞
Insert (Q, v, d(v)) ------- *log |V| * |V| times*
d(s) = 0; ---------*O (1)*
Decrease(Q, s, d(s)) // update priority of s with d(s) ------------ *O (|V|log |V|)*
$V_T = \Phi$ ---------*O (1)*
for** (i = 0 to |V| -1)* ***do
u= DeleteMin (Q) // delete the minimum priority element ------- *insertion in priority Queue* --- *O(log |V|)*
$V_T = V_T \cup \{u\}$ ------ *O(1)*
for** (every vertex u in V- V_T adjacent to u)* ***do
if (d(u) + w(u, v) < d(v)) then
d(v) = d(u) + w(u, v) *Relaxation checks every edge = O (|E| log |V|)*

Decrease(Q, u, d(u)) // update priority of u with d(u)
End

Total complexity = O(|V|) + O(|V| log|V|) + O(|V| log|V|) + O(log|V|) + O(|E| log|V|)
|E| >> |V|
Therefore,
Complexity = O(|E| log|V|)

CI **CCI** **HOD/AIML**