USN

Internal Assessment Test 2 – July 2024

Sub:	Analysis & Design of Algorithms					Sub Code:	BCS401	Branch:
								AIDS & CSE(AIDS)
Date:	**08/7/2024**	Duration:	**90 minutes**	Max Marks:	**50**	Sem/Sec:	**IV -A, B & C**	**OBE**

Answer any FIVE FULL Questions			**MARKS**	**CO**	**RBT**
1	a	Write a C function for Floyd Warshall.	4	3	L1

```
#define INF 1000000 // A very large number to represent infinity
#define V 4         // Number of vertices in the graph


// Function to print the shortest distance matrix
void printSolution(int dist[V][V]) {
    printf("The shortest distances between every pair of vertices:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
```

```
}



// Floyd-Warshall algorithm

void floydWarshall(int graph[V][V]) {

    int dist[V][V];



    // Initialize the solution matrix as a copy of the input graph matrix

    for (int i = 0; i < V; i++) {

        for (int j = 0; j < V; j++) {

            dist[i][j] = graph[i][j];

        }

    }



    // Update dist[][] for each intermediate vertex k

    for (int k = 0; k < V; k++) {

        for (int i = 0; i < V; i++) {

            for (int j = 0; j < V; j++) {

                // If vertex k is on the shortest path from i to j, update dist[i][j]

                if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] <
dist[i][j]) {

                    dist[i][j] = dist[i][k] + dist[k][j];

                }
```

```
            }

        }

    }



    // Print the shortest distance matrix

    printSolution(dist);

}



int main() {

    /* Example graph with 4 vertices:

       INF means there is no direct edge between two vertices.

    */

    int graph[V][V] = {

        {0, 3, INF, 7},

        {8, 0, 2, INF},

        {5, INF, 0, 1},

        {2, INF, INF, 0}

    };



    floydWarshall(graph);
```

```
    return 0;

}
```

	b	Apply Heapsort for the list [9,7,1,8,3,6,2,4,10,5] **Answer:** **1.** Build a max heap from the input array. 2. Extract the maximum element (root of the heap) repeatedly and adjust the heap. Input Array: [9, 7, 1, 8, 3, 6, 2, 4, 10, 5] Step 1: Build the Max Heap Start from the last non-leaf node (index) and heapify each subtree. Initial Array: [9, 7, 1, 8, 3, 6, 2, 4, 10, 5] Heapify Process (Bottom-up): 1. Heapify subtree rooted at index 4 (value 3): Children: 10 (index 9), 5 (index 10). Largest = 10. Swap 3 and 10. Result: [9, 7, 1, 8, 10, 6, 2, 4, 3, 5]. 2. Heapify subtree rooted at index 3 (value 8): Children: 4 (index 7), 3 (index 8). Largest = 8. No swap needed. 3. Heapify subtree rooted at index 2 (value 1): Children: 6 (index 5), 2 (index 6). Largest = 6. Swap 1 and 6. Result: [9, 7, 6, 8, 10, 1, 2, 4, 3, 5]. 4. Heapify subtree rooted at index 1 (value 7): Children: 8 (index 3), 10 (index 4).	6	3	L2

Largest = 10. Swap 7 and 10.

Result: [9, 10, 6, 8, 7, 1, 2, 4, 3, 5].

Now heapify the subtree rooted at index 4 (value 7):

Children: 5 (index 9), no second child.

Largest = 7. No swap needed.

5. Heapify subtree rooted at index 0 (value 9):

Children: 10 (index 1), 6 (index 2).

Largest = 10. Swap 9 and 10.

Result: [10, 9, 6, 8, 7, 1, 2, 4, 3, 5].

Now heapify the subtree rooted at index 1 (value 9):

Children: 8 (index 3), 7 (index 4).

Largest = 9. No swap needed.

Max Heap:

[10, 9, 6, 8, 7, 1, 2, 4, 3, 5]

Repeatedly extract the maximum element (swap root with the last element) and reduce the heap size.

1. Extract max (10):

Swap 10 with 5 (last element).

Result: [5, 9, 6, 8, 7, 1, 2, 4, 3, 10].

Heapify root (index 0):

Children: 9 (index 1), 6 (index 2).

Largest = 9. Swap 5 and 9.

Result: [9, 5, 6, 8, 7, 1, 2, 4, 3, 10].

Now heapify subtree rooted at index 1:

Children: 8 (index 3), 7 (index 4).

Largest = 8. Swap 5 and 8.

Result: [9, 8, 6, 5, 7, 1, 2, 4, 3, 10].

Heap after extraction: [9, 8, 6, 5, 7, 1, 2, 4, 3]

2. Extract max (9):

Swap 9 with 3 (last element).

Result: [3, 8, 6, 5, 7, 1, 2, 4, 9, 10].

Heapify root:

Children: 8 (index 1), 6 (index 2).

Largest = 8. Swap 3 and 8.

Result: [8, 3, 6, 5, 7, 1, 2, 4, 9, 10].

Now heapify subtree rooted at index 1:

Children: 5 (index 3), 7 (index 4).

Largest = 7. Swap 3 and 7.

Result: [8, 7, 6, 5, 3, 1, 2, 4, 9, 10].

Heap after extraction: [8, 7, 6, 5, 3, 1, 2, 4]

3. Extract max (8):

Swap 8 with 4 (last element).

Result: [4, 7, 6, 5, 3, 1, 2, 8, 9, 10].

Heapify root:

Children: 7 (index 1), 6 (index 2).

Largest = 7. Swap 4 and 7.

Result: [7, 4, 6, 5, 3, 1, 2, 8, 9, 10].

Now heapify subtree rooted at index 1:

Children: 5 (index 3), 3 (index 4).

Largest = 5. Swap 4 and 5.

Result: [7, 5, 6, 4, 3, 1, 2, 8, 9, 10].

Heap after extraction: [7, 5, 6, 4, 3, 1, 2]

<table>
<tr><td></td><td></td><td>4. Repeat the process until the heap size reduces to 1.

Final Sorted Array:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]</td><td></td><td></td><td></td></tr>
<tr><td>2</td><td>a</td><td>Explain the key concept of dynamic programming and provide a simple example where it is used.
Answer:
Key Concept of Dynamic Programming:
Dynamic Programming (DP) is a method for solving complex problems by breaking them into simpler overlapping subproblems, solving each subproblem once, and storing its result (memoization) to avoid redundant computations. It is particularly effective for optimization problems and problems that exhibit the overlapping subproblems and optimal substructure properties.
Overlapping Subproblems: The problem can be divided into smaller subproblems that are solved multiple times.
Optimal Substructure: The solution to a problem can be constructed from the solutions of its subproblems.
DP is commonly implemented using:
1. Top-Down Approach: Recursion with memoization.
2. Bottom-Up Approach: Iterative method with a table to store results.
Simple Example: Fibonacci Sequence
The Fibonacci sequence is defined as:
F(n) =
\begin{cases}
0 & \text{if } n = 0, \\
1 & \text{if } n = 1, \\
F(n-1) + F(n-2) & \text{if } n > 1.
\end{cases}</td><td>3</td><td>3</td><td>L3</td></tr>
<tr><td></td><td>b</td><td>Obtain the Huffman tree and the code for the following
<table><tr><td>Char</td><td>A</td><td>B</td><td>F</td><td>H</td><td>I</td><td>Y</td></tr><tr><td>Frequency</td><td>10</td><td>7</td><td>4</td><td>2</td><td>8</td><td>1</td></tr></table>
Answer:</td><td>7</td><td>3</td><td>L3</td></tr>
</table>

Step 1: Build the Huffman Tree

We combine the two smallest frequencies iteratively until one root remains.

1. Initial frequencies:

A (10), B (7), F (4), H (2), I (8), Y (1)

2. Combine smallest (H=2, Y=1):

Create a new node with frequency .

Remaining: A (10), B (7), F (4), I (8),

3. Combine smallest (N1=3, F=4):

Create a new node with frequency .

Remaining: A (10), B (7), I (8),

4. Combine smallest (B=7, N2=7):

Create a new node with frequency .

Remaining: A (10), I (8),

5. Combine smallest (I=8, A=10):

Create a new node with frequency .

Remaining: ,

6. Combine last two (N3=14, N4=18):

Create the root node with frequency .

Step 2: Assign Huffman Codes

Traverse the tree to assign binary codes (0 for left, 1 for right).

Resulting Huffman Codes:

char	**code**
A	**11**
B	**000**
F	**0011**
H	**00100**
I	**10**
Y	**00101**

Final Answer:

Huffman Tree: The tree structure is built as explained in Step 1.

Huffman Codes:

A = 11

		B = 000 **F = 0011** **H = 00100** **I = 10** **Y = 00101**			
3	a	Write a C function for performing quicksort, apply the same to the following set of numbers 15,5,24,8,1,3,16,10,20	10	3	L3

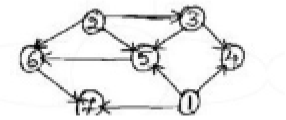
Answer:

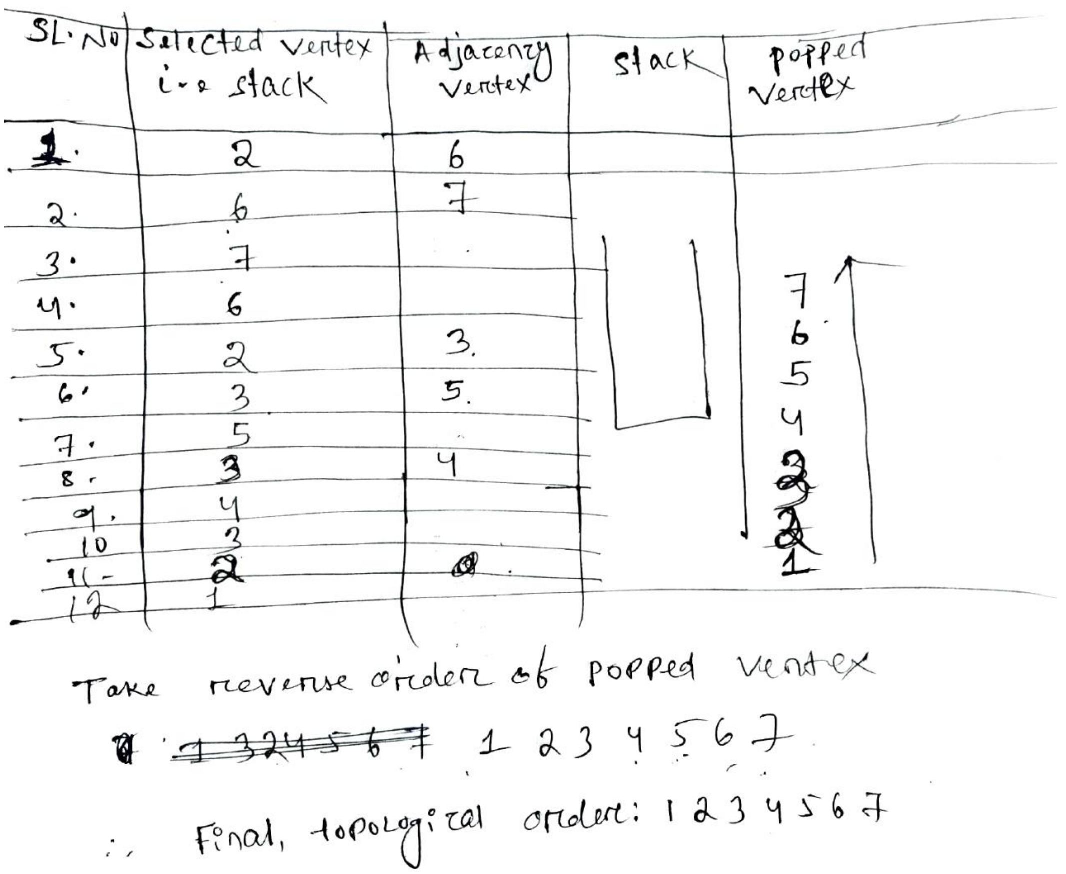
```
#include <stdio.h>
// Function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choose the last element as pivot
    int i = low - 1;       // Index of the smaller element

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    // Swap pivot to the correct position
    swap(&arr[i + 1], &arr[high]);
    return i + 1; // Return the partition index
}

// Quick sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high); // Partition index

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
```

```
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    int arr[] = {15, 5, 24, 8, 1, 3, 16, 10, 20};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array:\n");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    printf("Sorted array:\n");
    printArray(arr, n);

    return 0;
}
```

Step-by-Step Execution:
Given array: 15, 5, 24, 8, 1, 3, 16, 10, 20
Step 1: First Partition (Pivot = 20)
Initial array: 15, 5, 24, 8, 1, 3, 16, 10, 20
Elements smaller than 20 are moved to the left.
Partition result: 15, 5, 10, 8, 1, 3, 16, 20, 24
Partition index = 7 (pivot 20 placed in the correct position).
Step 2: Recursively Sort Left Subarray (15, 5, 10, 8, 1, 3, 16)
Pivot = 16.
Partition result: 15, 5, 10, 8, 1, 3, 16, 20, 24
Partition index = 6.
Step 3: Recursively Sort Left Subarray (15, 5, 10, 8, 1, 3)
Pivot = 3.
Partition result: 1, 3, 10, 8, 5, 15, 16, 20, 24
Partition index = 1.
Step 4: Recursively Sort Left Subarray (1)
Single element, no sorting needed.
Step 5: Recursively Sort Right Subarray (10, 8, 5, 15)
Pivot = 15.
Partition result: 10, 8, 5, 15, 16, 20, 24
Partition index = 5.
Step 6: Recursively Sort Left Subarray (10, 8, 5)
Pivot = 5.
Partition result: 5, 8, 10, 15, 16, 20, 24
Partition index = 0.
Final Sorted Array:

1, 3, 5, 8, 10, 15, 16, 20, 24

4	a	Discuss Strassen's matrix multiplication with an example. and derive its time complexity. **Answer:** **Strassen’s algorithm is a divide-and-conquer algorithm that improves the efficiency of matrix multiplication compared to the conventional algorithm. It was introduced by Volker Strassen in 1969 and reduces the number of multiplications required to compute the product of two matrices.** **Key Idea** **Strassen’s algorithm reduces the number of scalar multiplications required to compute the product of two matrices. The standard approach uses 8 scalar multiplications and 4 additions/subtractions for matrices, whereas Strassen’s algorithm uses only 7 scalar multiplications but increases the number of additions/subtractions to 18. For large matrices, this reduction in multiplications leads to faster computations.**	5	3	L1
	b	Obtain the topological sort for the graph by using source removal method and DFS method  **Answer:**	5	3	L2

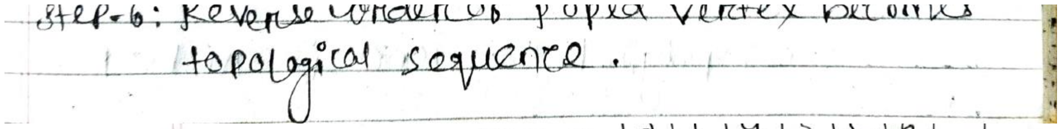


SL. No	Selected vertex i.e stack	Adjacency vertex	stack	popped vertex
1.	2	6		
2.	6	7		
3.	7	.		
4.	6			7
5.	2	3.		6
6.	3	5.		5
7.	5			4
8.	3	4		3
9.	4			
10	3			2
11.	2	0		1
12	1			

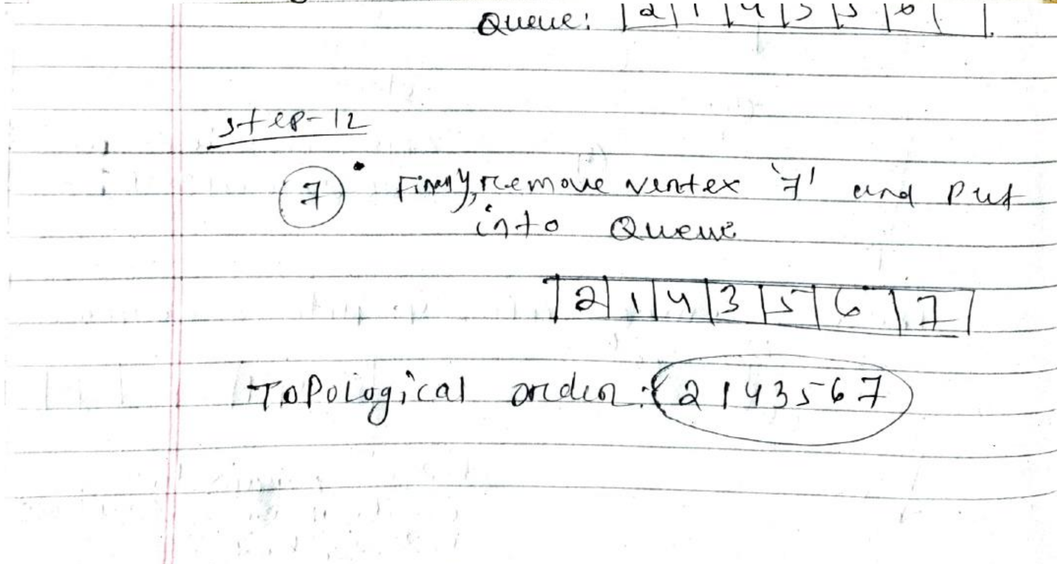
Take reverse order of popped vertex

0 : ~~1 3 2 4 5 6 7~~ 1 2 3 4 5 6 7

∴ Final, topological order: 1 2 3 4 5 6 7



Step-6: Reverse order of popped vertex becomes topological sequence.



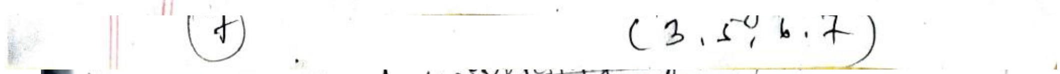
Queue: |2|1|4|3|5|6| |

step-12

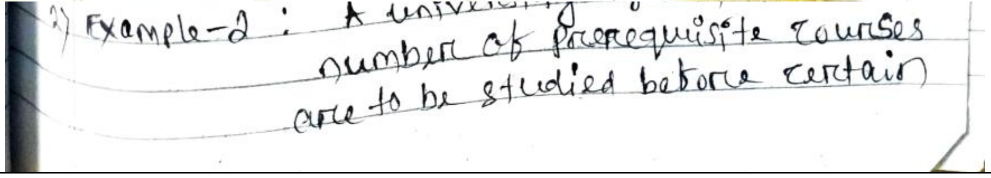
(7) Finally, remove vertex '7' and put into Queue

|2|1|4|3|5|6|7|

Topological order: (2 1 4 3 5 6 7)



(7) (3, 5, 6, 7)



2) Example-2 : A university ... number of prerequisite courses are to be studied before certain

5	a	Write a algorithm and solve the following instance of dynamic knapsack problem where n=4, m=40, p = (40, 42, 25, 12) and w = (5, 15, 25, 35) **Answer:**	10	3	L3

he 0/1 Knapsack Problem involves maximizing the total profit of selected items such that their total weight does not exceed a given capacity. We solve it using a dynamic programming approach. Here's the solution step by step:

Given:

n = 4 (number of items)

m = 40 (capacity of the knapsack)

P = {40, 42, 25, 12} (profits of items)

w = {5, 15, 25, 35} (weights of items)

Step 1: Define the DP Table
We define a 2D table dp[i][j], where:
i represents the items (1 to n).
j represents the knapsack capacity (0 to m).
dp[i][j] stores the maximum profit we can achieve with the first i items and a knapsack capacity of j.
Step 2: Initialization
When the capacity j = 0, the profit is 0 for all items: dp[i][0] = 0.
When there are no items (i = 0), the profit is 0 for all capacities: dp[0][j] = 0.
Step 3: Transition Formula
For each item i and capacity j:
If the item's weight w[i-1] > j (doesn't fit in the knapsack), exclude the item:
dp[i][j] = dp[i-1][j]
Otherwise, consider the maximum of including or excluding the item:
dp[i][j] = max(dp[i-1][j], dp[i-1][j - w[i-1]] + P[i-1])
Step 4: Fill the DP Table
We'll iteratively compute the values for all i and j using the transition formula.
Step 5: Extract the Result
The maximum profit will be stored in dp[n][m].
Solution: Step-by-Step Table Filling
Initialization
n = 4, m = 40
P = {40, 42, 25, 12}

w = {5, 15, 25, 35}

The dp table starts as:

dp[i][j] = 0 for all i and j

Fill the Table

Now, iterate through items and capacities:

1. Item 1 (weight = 5, profit = 40): For j from 1 to 40:

If j < 5, dp[1][j] = dp[0][j] = 0.

If j >= 5, dp[1][j] = max(dp[0][j], dp[0][j-5] + 40).

Result after processing Item 1:

dp[1][j] = [0, 0, 0, 0, 0, 40, 40, 40, ..., 40] (for j >= 5)

2. Item 2 (weight = 15, profit = 42): For j from 1 to 40:

If j < 15, dp[2][j] = dp[1][j].

If j >= 15, dp[2][j] = max(dp[1][j], dp[1][j-15] + 42).

Result after processing Item 2:

dp[2][j] = [0, 0, 0, 0, 0, 40, 40, ..., 82 (at j = 20), ..., 82 (for j >= 15)]

3. Item 3 (weight = 25, profit = 25): For j from 1 to 40:

If j < 25, dp[3][j] = dp[2][j].

If j >= 25, dp[3][j] = max(dp[2][j], dp[2][j-25] + 25).

Result after processing Item 3:

dp[3][j] = [0, 0, 0, 0, ..., 82 (at j = 25), ..., 82 (for j >= 25)]

4. Item 4 (weight = 35, profit = 12): For j from 1 to 40:

If j < 35, dp[4][j] = dp[3][j].

If j >= 35, dp[4][j] = max(dp[3][j], dp[3][j-35] + 12).

Result after processing Item 4:

dp[4][j] = [0, 0, ..., 82 (for j < 35), ..., 82 (for j >= 35)]

Final Table:

dp[4][40] = 82

Step 6: Traceback for Selected Items

To find which items are included:

1. Start from dp[4][40] = 82.

2. Check if dp[4][40] == dp[3][40]. If true, item 4 is not included.

3. Repeat this process to identify included items:

Item 3 is not included.

Item 2 is included (42 profit).

Item 1 is included (40 profit).

Final Answer:

		Maximum Profit: 82 **selected Items: Item 1 and Item 2 (weights = 5 and 15, profits = 40 and 42).**			
6	a	Explain the Heap Sort technique	3	3	L2

Answer:
Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure to sort elements. It has a time complexity of and is considered efficient and in-place since it requires only a constant amount of extra space.

Steps of Heap Sort:

1. Build a Max-Heap:

A binary heap is a complete binary tree where each parent node is greater than or equal to its child nodes (in the case of a Max-Heap).

Convert the given array into a Max-Heap. This ensures the largest element is at the root (index 0).

2. Extract Elements:

Swap the root element (largest) with the last element of the heap.

Reduce the size of the heap by one (exclude the last element from the heap).

Restore the Max-Heap property for the remaining heap (heapify).

3. Repeat:

Repeat the extraction process until the heap size is reduced to 1. At this point, the array is sorted.

Key Operations:

1. Heapify:

A process to ensure the Max-Heap property is maintained. Starting from a given node, compare it with its children, and if needed, swap it with the largest child. Repeat this process recursively for the affected child.

2. Building the Heap:

To build the heap, heapify all non-leaf nodes starting from the last non-leaf node and moving upward.

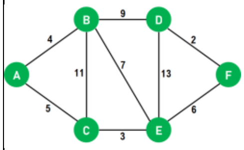
Algorithm in Pseudocode:

```
HeapSort(array):
    n = length(array)

    # Step 1: Build a Max-Heap
    for i = n/2 - 1 to 0:
        Heapify(array, n, i)

    # Step 2: Extract elements from the heap
```

```
   for i = n-1 to 1:
      Swap(array[0], array[i])  # Move the largest element to the end
      Heapify(array, i, 0)      # Restore the Max-Heap property for the reduced heap

Heapify(array, heap_size, root):
   largest = root
   left = 2*root + 1
   right = 2*root + 2

   if left < heap_size and array[left] > array[largest]:
      largest = left

   if right < heap_size and array[right] > array[largest]:
      largest = right

   if largest != root:
      Swap(array[root], array[largest])
      Heapify(array, heap_size, largest)
```

b

Solve the given graph to Dijistra’s method where Source is A.



Answer:

o solve the given graph using Dijkstra's algorithm with A as the source node, follow these steps:

Step 1: Initialize distances and visited set

1. Assign an initial distance of 0 to the source node (A) and infinity (∞) to all other nodes.

2. Mark all nodes as unvisited.

3. Set the source node (A) as the current node.

Step 2: Relax edges from the current node

For the current node, calculate the tentative distance to each neighboring node as: Tentative Distance = Distance to Current Node + Edge Weight.

If the tentative distance is smaller than the currently assigned distance for the neighbor, update it.

Step 3: Mark the current node as visited

7

3

L3

Once all neighbors of the current node have been processed, mark the current node as visited. A visited node will not be revisited.

Step 4: Move to the next node

Select the unvisited node with the smallest tentative distance as the next current node.

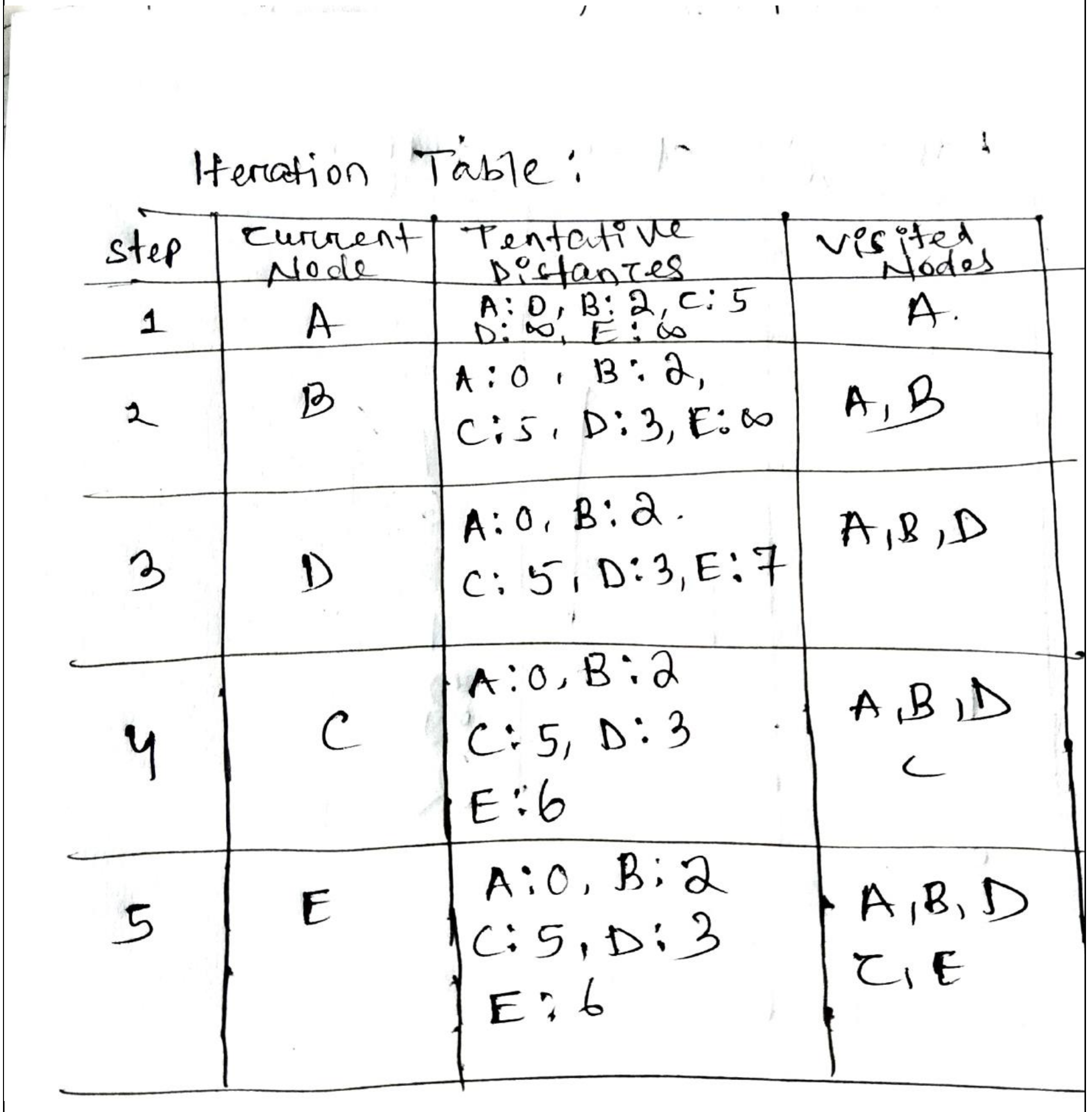
Repeat steps 2–4 until all nodes have been visited.

Step-by-Step Solution

Let's assume the graph's nodes and edge weights are as follows (based on your image):

Nodes: A, B, C, D, E
Edges with weights:
A → B: 2, A → C: 5
B → C: 3, B → D: 1
C → D: 2, C → E: 3
D → E: 4

Iteration Table



Iteration Table :

Step	Current Node	Tentative Distances	Visited Nodes
1	A	A: 0, B: 2, C: 5 D: ∞, E: ∞	A.
2	B	A: 0, B: 2, C: 5, D: 3, E: ∞	A, B
3	D	A: 0, B: 2. C: 5, D: 3, E: 7	A, B, D
4	C	A: 0, B: 2 C: 5, D: 3 E: 6	A, B, D C
5	E	A: 0, B: 2 C: 5, D: 3 E: 6	A, B, D C, E

Explanation of Each Step

1. Initialization (Step 1):

Start at A (distance = 0).

Distance to B = 2 (A → B).

Distance to C = 5 (A → C).

Other nodes remain at infinity.

2. Processing B (Step 2):

From B:
Distance to C = 2 + 3 = 5 (no change, already 5).
Distance to D = 2 + 1 = 3 (update).

3. Processing D (Step 3):

From D:
Distance to E = 3 + 4 = 7 (update).

4. Processing C (Step 4):

From C:
Distance to E = 5 + 3 = 6 (update, smaller than 7).

5. Processing E (Step 5):

E is already at the smallest distance (6), no further updates.

Final Shortest Distances:

Final

Node	Shortest Distance from A
A	0
B	2
C	5
D	3
E	6

Shortest Distances

		Let me know if you need further clarification or visualization!			

CI **CCI** **HoD**