ANSWER NO - 01

a)

Pseudocode of Dijkstra's algorithm :

a. Create a distance array “d”

b. Initialize all values of that array to infinity

c. d[source] = 0

d. Create a visited aray and mark all nodes as unvisited

e. For i=0 to n-1:

- pick the unvisited node with minimum d[node]
- visited[node] = 1 - For all adj_node of that node :
    If(d[node] + cost(node, adj_node) < d[adj_node] :
        - d[adj_node] = d[node] + cost(node, adj_node)

f. Output the array “d”

b)

1. **Limited to Non-negative weights:**
   Dijkstra's algorithm works correctly only when all the edge weights in the graph are non-negative. If there are negative weights, then the algorithm may not give the correct shortest path as it doesn't account for the possibility of a negative cycle.

2. **Does not work with disconnected graphs:**
   Dijkstra's algorithm only works with connected graphs. If the graph is disconnected, then the algorithm may not provide a correct shortest path as it won't be able to reach all the nodes in the graph.

3. **Time Complexity :**
   The main advantage of Dijkstra’s algorithm is its considerably low complexity, which is almost linear.
   However, when working with negative weights, Dijkstra’s algorithm can’t be used.

   Also, when working with dense graphs, where E is close to V^2, if we need to calculate the shortest path between any pair of nodes, using Dijkstra’s

algorithm is not a good option. The reason for this is that Dijkstra's time complexity is O(V + E \cdot log(V)). Since E equals almost V^2, the complexity becomes O(V + V^2 \cdot log(V)).

ANSWER NO - 02

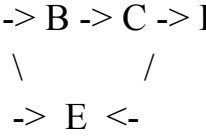
a)
Topological Sort With Example :

Topological sort is a graph algorithm that sorts the vertices of a directed acyclic graph (DAG) in such a way that if there is a path from vertex A to vertex B, then vertex A appears before vertex B in the sorted order. In other words, it produces a linear ordering of the vertices such that for every directed edge (u, v), vertex u comes before vertex v in the ordering. One practical application of topological sorting is scheduling tasks with dependencies, where the dependencies between tasks are modeled as edges in a directed graph.

Here's an example to illustrate the concept of topological sort: Suppose we have a graph representing the prerequisites for some courses in a computer science curriculum:

Example :
Course A has no prerequisites
Course B requires Course A
Course C requires Course B and Course E
Course D requires Course A and Course C
Course E has no prerequisites

We can represent this as a directed graph:



To find the topological order of this graph, we can use a modified depth-first search algorithm. Starting from any node with no incoming edges (indegree of 0), we can visit all its outgoing edges (prerequisites), and remove them from the graph. After we remove a node, we add it to the final ordering, and repeat the process with the remaining nodes until there are no more nodes left in the graph.

One possible topological sort of the given graph is: A, B, E, C, D.

This ordering satisfies the requirement that every prerequisite of a course appears before the course itself. For example, Course B requires Course A, and we have A appearing before B in the ordering. Similarly, Course C requires Course B and Course E, and we have B and E appearing before C in the ordering.

b)

No, a topological sort cannot be implemented on a Directed Cyclic Graph (DCG) because a topological sort requires a Directed Acyclic Graph (DAG) as input. This is because a DAG has a topological order, while a cyclic graph does not.

In a cyclic graph, there exist cycles, which means that there is a path from a node back to itself. This creates a problem for topological sort because it requires that a node with incoming edges (indegree) be visited only after all its prerequisites have been visited, but in a cycle, this condition cannot be met.

For example, if we have a cycle A -> B -> C -> A, then we cannot visit any of these nodes first because they all have incoming edges from each other. If we attempt to run a topological sort on a DCG, we may get an incomplete or incorrect result. The algorithm may not be able to produce a valid topological order and may even get stuck in an infinite loop due to the cycles.

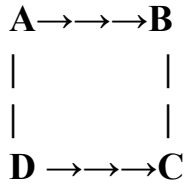
Therefore, before running a topological sort algorithm, we need to ensure that the input graph is a DAG. If we find a cycle in the graph, we can conclude that a

topological sort is not possible, and we need to take a different approach to solving the problem

ANSWER NO - 03

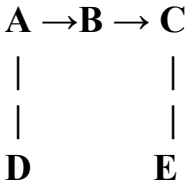
a)

1. Directed Cyclic Graph (DCG) :



In this graph, we have a cycle A -> B -> C -> D -> A, which makes it a DCG.

2. Directed Acyclic Graph (DAG) :



In this graph, there are no cycles, which makes it a DAG. We can see that we can assign a topological order to the nodes as follows: A, D, B, C, E.

b)

Differences between Directed Cyclic Graphs (DCG) and Directed Acyclic Graphs (DAG):

1. **Cycles:** A DCG contains at least one cycle, i.e., a path that starts and ends at the same node, while a DAG has no cycles. This means that in a DCG, we can follow a sequence of edges and eventually come back to the starting node, while in a DAG, we can't.

2. **Topological order:** A topological order is a linear ordering of the nodes of a DAG, such that if there is an edge from node A to node B, then A appears before B in the ordering. This is not possible in a DCG because we cannot define a topological order when there is a cycle. A topological order is useful in many applications, such as scheduling tasks with dependencies or resolving dependencies in software libraries.

3. **Shortest path:** Finding the shortest path between two nodes in a DCG is not always well-defined because there can be multiple paths between the nodes that have different lengths. In a DAG, on the other hand, the shortest path between two nodes is well-defined and can be found using algorithms such as Dijkstra's algorithm

4. **Connectivity:** A DCG can have isolated nodes, i.e., nodes that are not reachable from any other node in the graph. In contrast, a DAG is always strongly connected, meaning that there is a path from every node to every other node in the graph. If we remove all edges in a DAG, we will get multiple disconnected DAGs, each of which is still a DAG.

5. **Algorithmic complexity:** Many graph algorithms have better time and space complexity on DAGs than on DCGs. For example, topological sort can be done in linear time on a DAG, but not on a DCG. Similarly, shortest path algorithms have better time complexity on DAGs than on DCGs, due to the absence of cycles.

ANSWER NO - 4

a)

The purpose of a base case in a recursive function is to provide a stopping condition for the recursion. Without a base case, the recursive function will keep calling itself indefinitely, eventually leading to a stack overflow error or an infinite loop.

A base case defines the simplest version of the problem that can be solved without recursion. When the input to the function meets the criteria of the base case, the recursion stops, and the function returns a value.

In essence, the base case serves as a termination condition for the recursive function.

For example, consider the following recursive function that computes the factorial of a positive integer:

```
function factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

The base case is when n is equal to 0, and in that case, the function returns 1. This is necessary because the factorial of 0 is 1. Without the base case, the function would call itself indefinitely, leading to a stack overflow error.

In general, it is always necessary to include a base case in a recursive function to ensure that the recursion terminates.

b)

Recursion and iteration are both techniques used in programming to repeat a block of code multiple times. Recursion is a technique in which a function calls itself one

or more times to solve a problem, while iteration is a technique in which a block of code is repeated using a loop.

Recursion involves solving a problem by breaking it down into smaller subproblems of the same type. Each subproblem is solved using the same recursive function, which calls itself with smaller input until it reaches a base case where the problem is simple enough to solve without recursion. The solutions to the subproblems are combined to solve the original problem. Recursion is often used when the problem can be easily divided into smaller subproblems that are identical to the original problem.

On the other hand, iteration involves repeating a block of code a fixed number of times, or until a certain condition is met. Iteration is typically used when we need to repeat a block of code a known number of times, or when we need to repeat a block of code until a specific condition is satisfied. In iteration, a loop is used to repeat the block of code, and the loop variable is updated with each iteration until the loop terminates.

Here is an example of a recursive function to compute the factorial of a positive integer:

```
function factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

And here is an example of an iterative function to compute the same thing:

```
function factorial(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
```

In conclusion, Both of these functions compute the factorial of a positive integer, but they use different techniques to do so. Recursion is often considered more elegant and expressive than iteration, but it can be less efficient in some cases due to the overhead of function calls. Iteration, on the other hand, is often more efficient, but it can be harder to understand and maintain.

ANSWER NO - 05

a)

A bipartite graph is a type of graph in which the vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set. In other words, there are no edges that connect vertices within the same set.

Formally, a graph G = (V, E) is bipartite if we can partition the vertex set V into two disjoint sets V1 and V2 such that every edge in E connects a vertex in V1 to a vertex in V2.

Bipartite graphs have many applications, including in computer science, biology, and social sciences. For example, a bipartite graph can be used to model relationships between two different types of objects, such as users and movies in a recommendation system, or genes and diseases in a gene expression analysis.

One way to test whether a graph is bipartite is to use a graph coloring algorithm. We can assign one color (e.g., red) to the vertices in one set (e.g., V1) and another color (e.g., blue) to the vertices in the other set (e.g., V2), and then check whether there are any adjacent vertices with the same color. If there are no adjacent vertices with the same color, the graph is bipartite; otherwise, it is not.

b)

Bipartite Graph Detection :

There are several ways to detect whether a graph is bipartite. Here are two common methods:

1. **Graph coloring:** One way to detect whether a graph is bipartite is to use a graph coloring algorithm. We can assign one color (e.g., red) to the vertices in one set and another color (e.g., blue) to the vertices in the other set. We then traverse the graph using a graph traversal algorithm (e.g., DFS or BFS) and color each vertex according to its distance from the starting vertex. If there are any adjacent vertices with the same color, the graph is not bipartite; otherwise, it is bipartite. This method has a time complexity of O(V + E), where V is the number of vertices and E is the number of edges in the graph.

2. **Depth-first search (DFS):** Another way to detect whether a graph is bipartite is to use a modified version of the DFS algorithm. We start by choosing an arbitrary vertex as the starting vertex and assigning it to set A. We then traverse the graph using DFS, coloring each vertex in set B if it is an uncolored neighbor of a vertex in set A, and vice versa. If we encounter a vertex that has already been colored with a different color than the one we want to assign to it, the graph is not bipartite; otherwise, it is bipartite. This method also has a time complexity of O(V + E). Both methods can be used to detect whether a graph is bipartite, and the choice of which method to use depends on the specific problem and the characteristics of the graph.

Both methods can be used to detect whether a graph is bipartite, and the choice of which method to use depends on the specific problem and the characteristics of the graph.

#Steps of Graph coloring:

1. Pick an arbitrary vertex in the graph and assign it a color, say red. Let's choose vertex A as the starting vertex and color it red.

2. Color all neighbors of the starting vertex with a different color, say blue. In this case, neighbors B and C are adjacent to A, so we color them blue.

3. Proceed to color the neighbors of each blue vertex with the original red color. In this case, vertex D is adjacent to both B and C, so we color it red.

4. Continue coloring the neighbors of each vertex with the opposite color until all vertices have been colored or until we encounter two adjacent vertices with the same color. If we encounter two adjacent vertices with the same color, then the graph is not bipartite. Otherwise, the graph is bipartite.

In this example, all vertices can be successfully colored with two colors, red and blue, without any adjacent vertices having the same color. Therefore, the graph is bipartite.