Answer to the question no-01

(a)Ans:

Pseudocode of Dijkstra Algorithm:

input-> a weighted graph and source.
output-> distance of all nodes from the source node.

Void Dijkstra(int src){

1. Create a distance array 'Dis'.
2. Initialize all the values of 'Dis' to infinity or a large number.
3. Initialize distance from source is 0, Dis[src]=0.
4. Create a visited array and mark them as unvisited.
5. Start a loop from i=0 to i=n-1
    1. We have to pick the shortest Dis[node] which is unvisited.
    2. Make visited[node]=1.
    3. Again run a loop for all adjacent nodes of selected node's.
        1. Check the condition if Dis[node]+ cost from selected node to adjacent node<Dis[adjacent node].
            -Dis[adjacent node]=Dis[node]+cost from selected node to adjacent node.
6. Output the distance array.

}

(b)Ans:

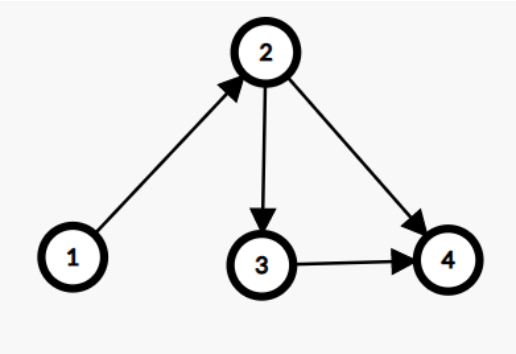
The limitation of Dijkstra Algorithm:

1. It takes too much time when performing all pairs of nodes to the shortest distance.
2. If edge cost is negative, sometimes it will give the wrong answer.
3. As it is working in a greedy approach, I think it can not be the optimal solution as there are several algorithms for this.
4. When using priority queue the time complexity becomes low but in other languages like C there is no built in priority queue , for that we have to implement it using a function otherwise time complexity becomes so high.

Answer to the question no-02

(a)Ans:

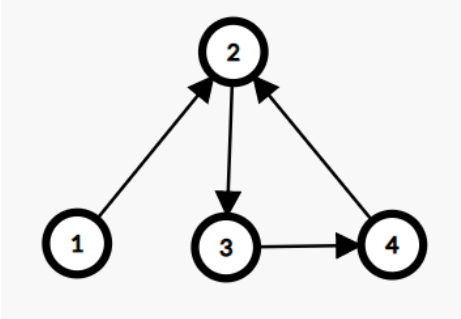
If there is any dependency on the graph then we can sort them according to its dependence from first to last node.
To use topological sorting the graph should be directed acyclic graph(DAG).
If there is connection like
(u)--------->(v) it means that (v) is dependent on (u).
If we see a graph:



Here 2 is dependent on 1. 3 is dependent on 2. 4 is dependent on 2 and 3.
If we do topological sorting on it, it will look like:
1 2 3 4.
We firstly have to take node 1 as it is independent. Then comes 2 as we have visited 1. Then take 3 as we have visited 2. Then take 4 as we have visited 2 and 4.

(b)Ans:

No, topological sorting can not be implemented in directed cyclic graphs.
Because in cyclic graphs we can come to a node again which was visited already.
If we see a DCG:

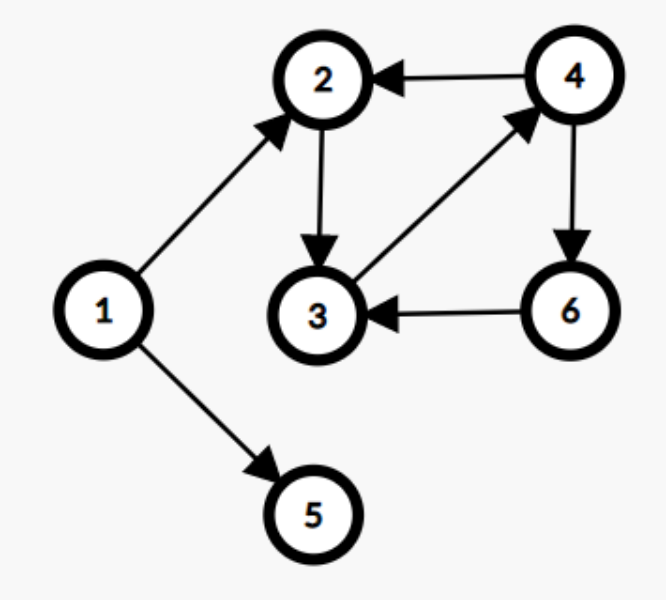


Here 1 is independent. But 2 is dependent on 1 and 4. Whereas 4 is dependent on 3. Again 3 is dependent on 2. So this can not be finished. So it is impossible to implement it in DCG.

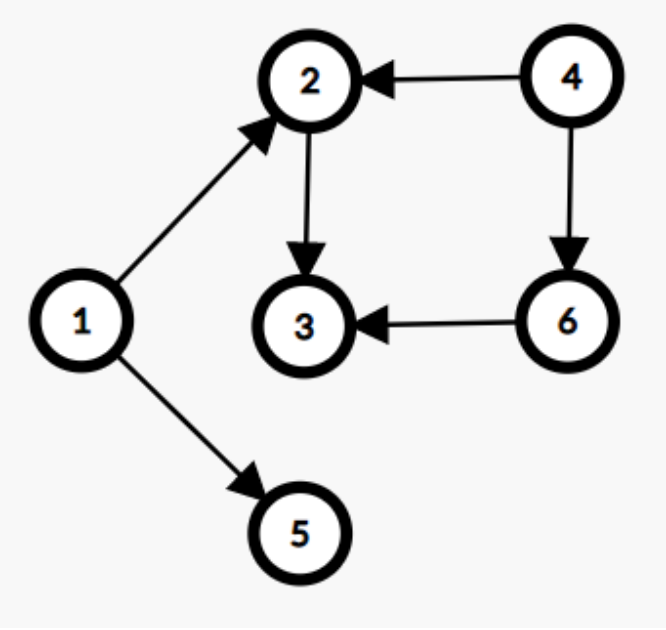
Answer to the question no-03

(a)Ans:

Directed Cyclic Graph:



Directed Acyclic Graph:



(b)Ans:

Directed Cyclic Graph	Directed Acyclic Graph
1. It has at least one cycle	1. It has no cycle anywhere.
2. Possible to find the visited node in same move	2. It is possible to find the done node.
3. Can not implement topological sorting.	3. Easy to implement topological sorting.
4. As it is a one way movement so we can easily come back to a visited node when we need to.	4. As there is no cycle we can not come back to the visited node one way without taking another node's way.

Answer to the question no-04

(a)Ans:

Base case is a termination condition of a recursive function. When the recursive function is working it will go until there comes any ending point. When it gets a base case the function terminates its work and executes.
If the base case is not included the function falls in an infinite loop and it will not terminate anymore.
If we see an example:

```
int Pow(int n,int m)
{
if(m==1)
{
return n;
}
return n*Pow(n,m-1);
}
```

It calculates the n^m.
Here the base case is m==1. If it is not included the program will always be run and the value of m will be decreased in an infinite loop. It will look like

```
int Pow(int n,int m)
{
return n*Pow(n,m-1);
}
```

(b)Ans:

Recursion is the process of calling the same function again and again for the same work until finding any base case to stop.

Iteration is the process of doing the same thing or doing calculation in a bounded loop in a function.

Recursion is a function. Iteration is a syntax.
Recursion is a function but iteration is used in a function.
Recursion functions can be many types but iteration is 3 types: for, while, do while.
If we see an example:

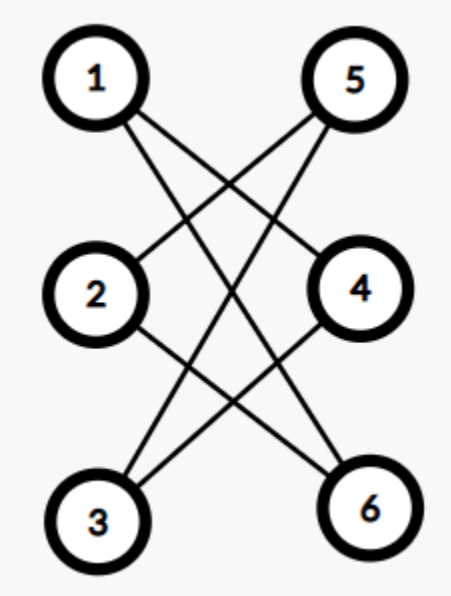
int Pow(int n,int m)
{
if(m==1)
{
return n;
}
return n*Pow(n,m-1);
}
	int a=1;
	for(int i=1;i<=m;i++)
	{
	a=a*n;
	}

We have written the same thing in different ways. It calculates n^m.
Here m==1 is a base case of a recursive function and i<=m is the loop's running time condition.

Answer to the question no-05

(a)Ans:

The bipartite graph is a graph where the same side of nodes are not connected to each other.
If we see an example:



It is a bipartite graph.
Here the left nodes are 1 2 3.
Here the right nodes are 5 4 6.
We can see that node 1 , 2 ,3. They are not connected with each other.
Again on the right side 5 4 6. They are not connected with each other. But they are interconnected with the left nodes to right nodes.

(b)Ans:

We can detect a bipartite graph by graph coloring.
In this case no 2 adjacent nodes can have the same color.
For this steps are:

1. Give a source node a particular color.
2. Always checking the color of adjacent nodes.

If it is not possible to give 2 different colors between 2 adjacent nodes then it is not a bipartite graph.

Here I give an example with a pseudocode:

bool color(){

1. Run a loop from node i=1 to i=n.
   -if it is not visited[i].
     -visited[i]=1
     -color[i]=1
     -Run a loop for the node's of its adjacent node.
       -if visited[adjacent node]==0
         -visited[adjacent node]=1
         -if color[node]==1
           -color[adjacent node]=2
          else if color[adjacent node]==color[node]
            -return false.
          else
            -color[adjacent node]=1;
2. return true.

From this method we can identify the bipartite graph easily.