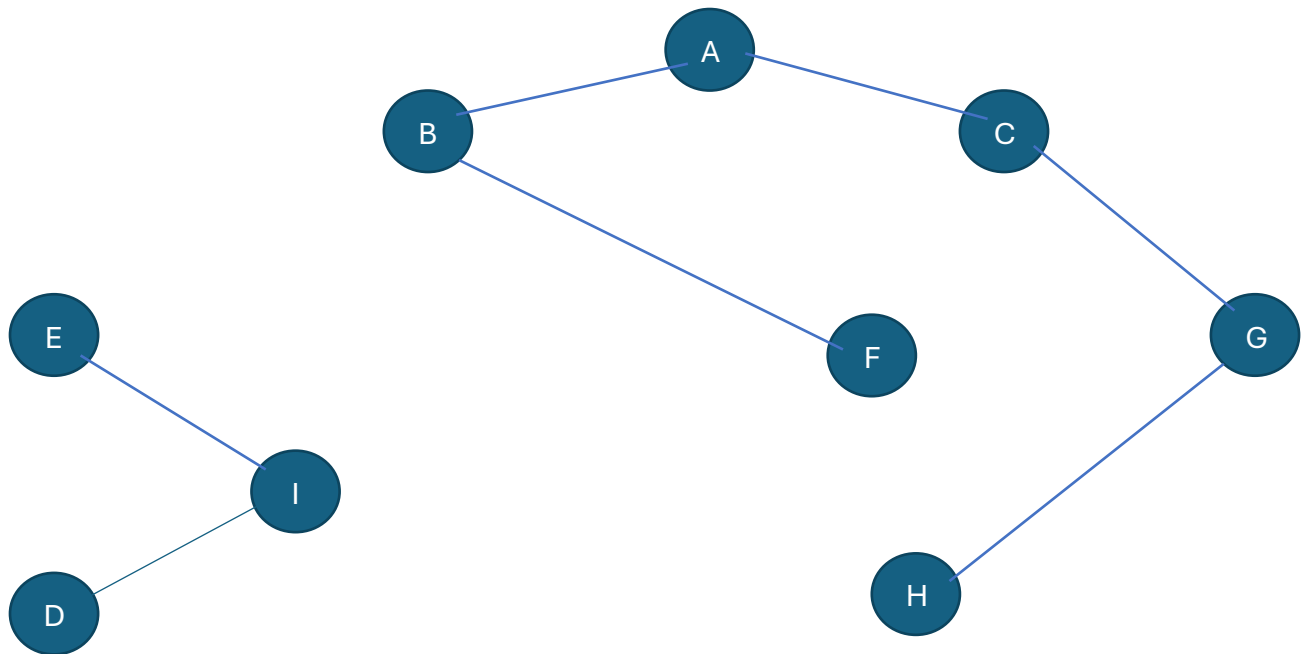


Q1.

- a. No the graphs are not connected. BACGHF and DEI are the connected components for graph G.

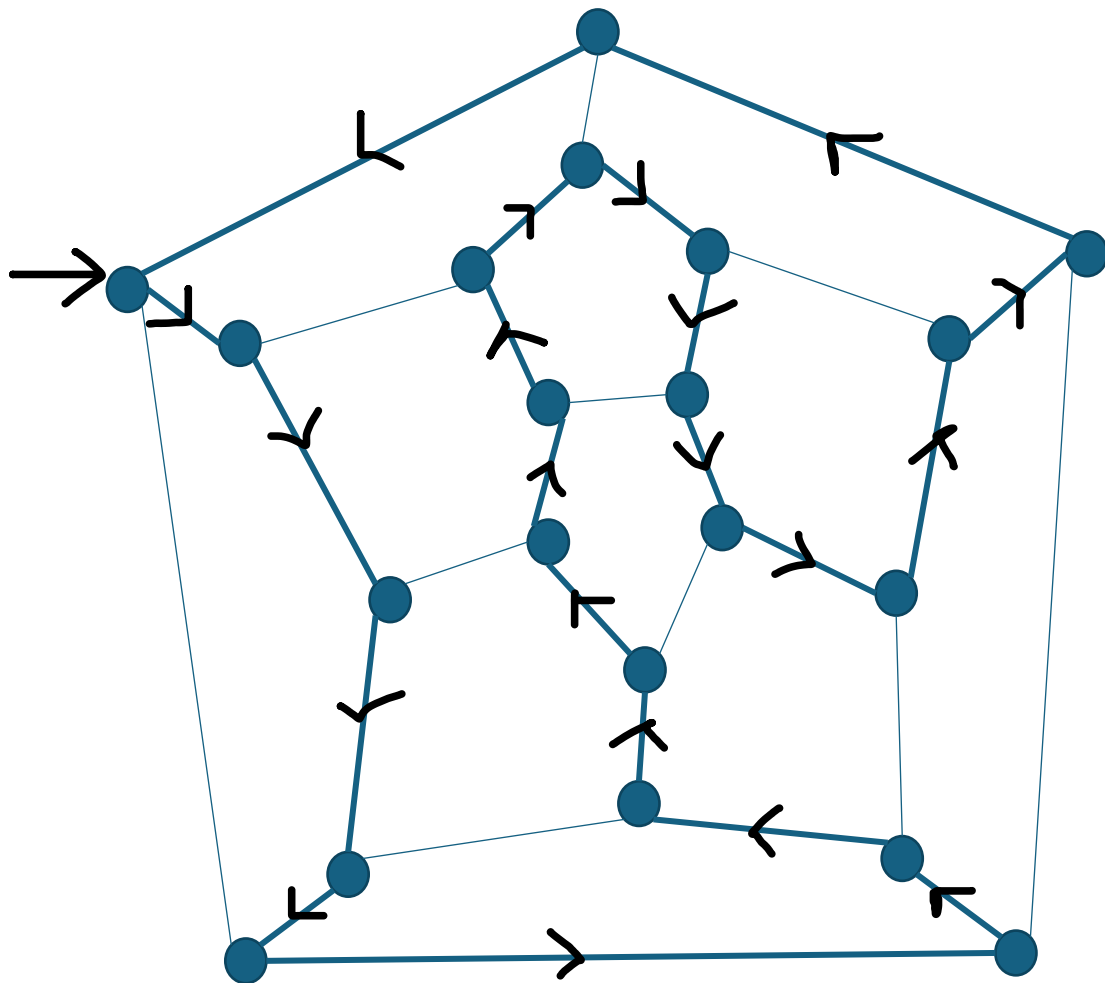


b.

- c. No Graph G is not a hamiltonian cycle, because of its disconnected nature we cannot reach to all of its vertices.
- d. Yes, there is a vertex cover of size less than or equal to 5.

$$Vc1 = \{E, I\}, Vc2 = \{F, G, A\} \Rightarrow Vc = \{A, G, F, E, I\}$$

Q2.



Q3.

Algorithm SubsetList(list,subset,arr,i)

Input: result list, subset is current subset, arr original elements , i current index

Output: list containing subsets of arr

```
if i>arr.length then
    return list
if i==arr.length then
    list.add(new List(subset))
    return list
```

```
SubsetList(list,subset,arr,i+1)
subset.add(arr[i])
SubsetList(list,subset,arr,i+1)
subset.removeLast()
return list
```

```
/*
*
* Time complexity =>  $O(n * 2^n)$ 
*
*/
```

Algorithm powerSets(arr)

Input: arr containing list of items

Output: List of list of subsets of arr

return SubsetList(new List(), new List(), arr, 0)

Algorithm computeEndpoints(edge)

Input: edge of a graph

Output: List of the vertices of the edge

Algorithm belongsTo(vertices,set)

Input: two lists of vertices

Output: true if at least one of the item in vertices is in set

```
/*
*
* Time complexity =>  $O(n * 2^n) + (2^n * k)$  ; n is # of vertices and k is # of edges
*
*/
```

Algorithm smallestVertexCover(v,e)

Input: v-> set of vertices, e -> set of edges

Output: smallest vertex cover

smallestVertexCover <- v;

```

powerSet <- powerSets(v)
for set in powerSet do
  isSVC <- true
  for edge in e do
    vertices <- computeEndpoints(edge)
    isSVC <- isSVC && belongsTo(vertices,set)
  if isSVC then
    if set.length < smallestVertexCover.length then
      smallestVertexCover = set
return smallestVertexCover

```

Q4.

```

class Solution {
private void visit(char[][]grid,boolean[][]visited,int i, int j){
  if(i < 0 || i >= grid.length || j < 0 || j >= grid[i].length || visited[i][j]==true || grid[i][j]=='0'){
    return ;
  }
  visited[i][j] = true;
  visit(grid,visited,i-1,j);
  visit(grid,visited,i+1,j);
  visit(grid,visited,i,j-1);
  visit(grid,visited,i,j+1);
}
public int numIslands(char[][] grid) {
  boolean[][] visited = new boolean[grid.length][grid[0].length];
  int islands = 0;
  for(int i=0;i<grid.length;i++){
    for(int j=0;j<grid[i].length;j++){
      if(!visited[i][j] && grid[i][j]=='1'){
        visit(grid,visited,i,j);
        islands++;
      }
    }
  }
  return islands;
}
}

```

Q5.

```
class Solution {
    private int areaofIsland(int[][] grid, boolean[][] visited, int i, int j, int area){
        if(i < 0 || j < 0 || grid.length <= i || grid[i].length <= j || grid[i][j]==0 || visited[i][j]==true){
            return area;
        }
        visited[i][j] = true;
        return 1 + area + areaofIsland(grid, visited, i-1, j, area)
            + areaofIsland(grid, visited, i+1, j, area)
            + areaofIsland(grid, visited, i, j+1, area)
            + areaofIsland(grid, visited, i, j-1, area);
    }

    public int maxAreaOfIsland(int[][] grid) {
        boolean[][] visited = new boolean[grid.length][grid[0].length];
        int maxArea = 0;

        for(int i=0; i<grid.length; i++){
            for(int j=0; j<grid[i].length; j++){
                if(!visited[i][j] && grid[i][j]!=0){
                    int area = areaofIsland(grid, visited, i, j, 0);
                    if(area > maxArea){
                        maxArea = area;
                    }
                }
            }
        }

        return maxArea;
    }
}
```

Q6.

```
public class WordSearch {
    private static boolean backtrack(char[][] board, String word, int i,int j, int strIndex){
        if(strIndex >= word.length()) return true;

        if(i<0||j<0||i>=board.length||j>=board[i].length){
            return false;
        }
        if(board[i][j]!=word.charAt(strIndex)){
            return false;
        }

        char temp = board[i][j];
        board[i][j] = '*';
        if(
            backtrack(board,word,i+1,j,strIndex+1)||
            backtrack(board,word,i-1,j,strIndex+1)||
            backtrack(board,word,i,j+1,strIndex+1)||
            backtrack(board,word,i,j-1,strIndex+1)
        ){
            return true;
        };

        board[i][j] = temp;
        return false;
    }

    public static boolean exist(char[][] board, String word) {

        for(int i=0;i<board.length;i++){
            for(int j=0;j<board[i].length;j++){
                if(backtrack(board,word,i,j,0)){
                    return true;
                }
            }
        }
        return false;
    }
}
```

```
}  
}
```