Forward Star: This is a forward star representation for a directed graph with $|V| = 11$ vertices and $|E| = 16$ edges.

```
Vertex Number:    1    2    3    4    5    6    7    8    9   10   11   12
Array First:   { 1,   3,   4,   5,   7,   8, 12, 12, 14, 14, 15, 17 }
```

```
Edge Number:  1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17
Array Edge: { 2,   6,   6,   7,   3,   7,   8,   5,   8,   9, 10,   9, 11,   9,   9, 10, -1 }
```

(a) Draw the graph on the template in Fig. **??**. (HINT: You may want to do part b first.)

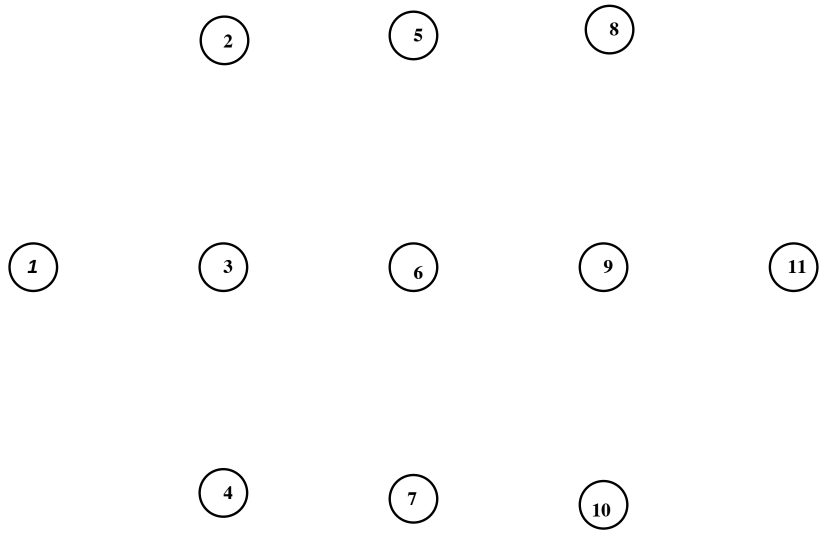

Figure 2:

(b) Represent this graph as an adjacency list.

(c) Is this graph a DAG?

Connected Components:

An undirected graph G(V,E) is defined interm of two arrays: `First[0:Vsize]` and `Edge[0:Esize]` that label the Vertice(nodes) from $0, 1, \cdots, Vsize - 1$ and Edges (arcs) from $0, 1, \cdots, Esize - 1$ respectively. The last "fake" vertex (`Frist[Vsize] = Esize`) point to the null "fake" edge with "null" value (`Edge[Esize] = -1`), See Exercise 3 above for an small example.

Implement your algorithm as a C/C++ function that counts the number of connected components. On GitHub there is the main file that reads input and writes output the result. You only write the required functions. A code `makeGraph.cpp` has been posted that generates "random" undirected graphs, which you use for you amusement if you want to test your algorithm small or large graphs of varying sparsity.

Your connected component function needs to call **over and over again** a `Grow(...)` function using either BFS (or DFS) until you have visited **all** the nodes in the graph. It is preferable to use BFS as a more efficient way to grow each connected componet from its start node. The functions for BFS have been included. The number of time you call `Grow(..)` is the number of connected components. All you need to return is one extra a global array `Found[0:Vsize]` initialize to -1 for not found and then set to something else when each node is visited so your next call to search `Grow(..)` goes only to nodes that have been left out.

The task is to first implement connected components with BFS and a Queue and then to add to the code another function doing DFS using a stack.

Write the function:

```
        int find_connected_components_BFS()
        Stack * createStack()
        Push(..)
        Pop(..)
        int find_connected_compoents_DFS()
```

Find the number of time and connected components using BFS and then add the find DFS method so you can compare the time of execution and show that the number of connected components are the same of course!

Do not make any changes to the `infield` reading format. so that it easy for the grader to run you code against a set of input files to test the code correctness. When you implement the DFS version the `outfile` writing format in `main()` You are given as set of input graphs and few solutions.

Analysis For extra credit if you pick the label found as a distinct number inside each connected component (e.g. $1, 2, \cdots,$ Number-Connect-Compoents), you can learn a lot more – like how many live in the largest most popular cluster etc.

Let's discuss more extension class and how to graph resutls. Might lead to a fun project as well.

```
        More fun on conected components
        Google look at the curious
        Regular graphs
        Cluster finding variations etc
```