1. Write an algorithm to identify the ”back arcs” in a spanning forest generated by DFS of a digraph.

Along with the usual “visited” array of DFS, we will also use two extra arrays called startTime array and endTime array.

The ith cell of the startTime array will indicate the time at which node i was visited for the first time and corresponding cell in the endTime array will indicate the time at which all of the neighbours of the node i were done visited. Its not necessary that these 2 arrays have to store the absolute time. The only condition on the time value is that if a node i is visited before node j then startTime[i] < startTime[j] and the same condition should hold for endTime in the similar situation.

We are assuming that the digraph is given in adjacency list form. We are going to use following data structures as follows.

visited array: to keep track of the vertices that have been visited during the BFS traversal. Initially every cell will contain false.

startTime array: In cell i, it will store the time at which node i was visited for the first time. Initially, every cell will be initialized to zero.

endTime array: In cell i, it will store the time at which neighbours of node i was done visiting. Initially, every cell will be initialized to zero.

time variable: It will keep track of current time

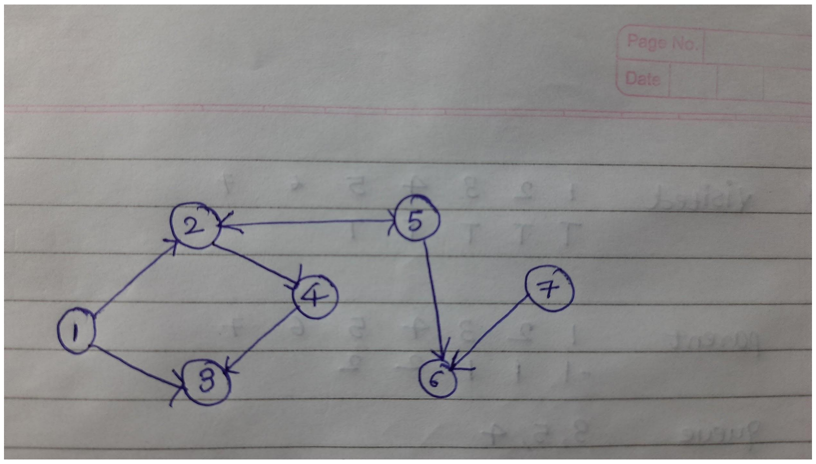
The procedure to find back arcs is as follows.

```
DFS(v):
    For i in adjList.size():
        If visited[i] == false:
            walk(i)

Walk(node):
    visited[node]=true
    startTime[node]=time++

    For neighbour in adjList[node]:
        If visited[neighbour] == true:
            If startTime[node] > startTime[neighbour] and endTime[node] <
endTime[neighbour]:
            # back arc found. Stop the process
            print(“back arc from”, node ” to “, neighbour)
```

```
break;
```

2. Represent a digraph using an adjacency list and write the procedure to find the shortest path between a given pair of vertices. Illustrate the process with the diagram.

We will take an example of the following digraph to illustrate the solution.



Its adjacency list representation will be as follows.

1 - 2, 3
2 - 4, 5
3 - (empty list)
4 - 3
5 - 2, 6
6 - (empty list)
7 - 6

To find the shortest path between a given pair of vertices, we will use BFS since the graph is non-weighted. We will also use the following two arrays. Both of them will be 1-based indexed.

visited array - to keep track of the vertices that have been visited during the BFS traversal
parent array - in this array, the ith cell will indicate that the vertex parent[i] is used to visit the vertex i for the first time in BFS traversal

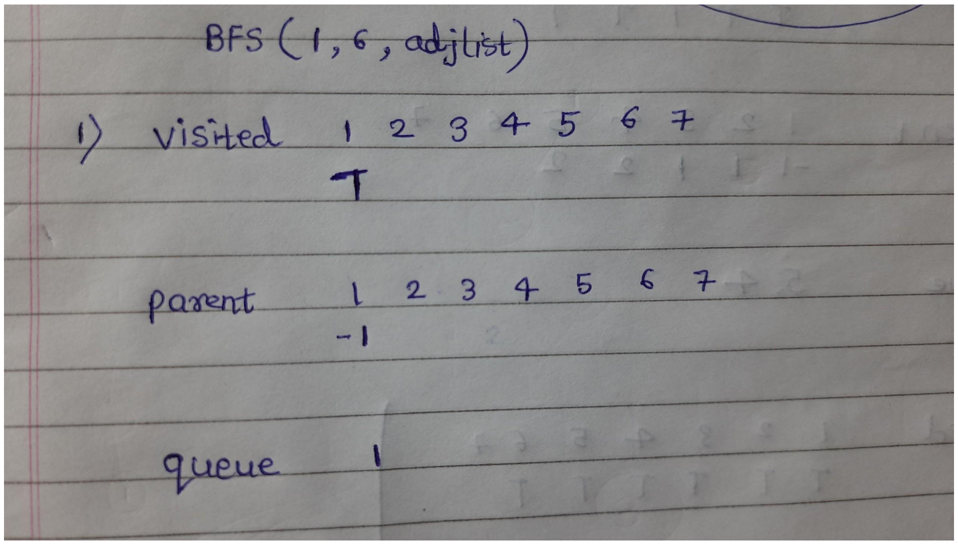
Following is a procedure used to find the shortest path between a given pair of vertices.

v1 : the starting vertex
v2 : the destination vertex
adjList: the adjacency list of the given digraph

```
BFS(v1, v2, adjList):
        Create a queue called q
        q.enqueue(v1)
```

```
    visited[v1] = true
    parent[v1] = -1

    # flag will  be used to indicate whether we v1 and v2 are connected or not
    flag = false
    while q is not empty && !flag:
        top = q.dequeue()
        for v in adjList[top]:
            if visited[v] == false:
                visited[v] = true
                parent[v] = top
                if v == v2:
                    Break
                q.enqueue(v)

    if flag:
        # we will use a stack to store all the edges that leads to v2 from v1
        # since, we are adding edges using parent array from v2 to v1, the edges of path
from v1 to v2 will be reverse order. Hence after adding all the edges we will use pop() to print
the edges in required order.

        st = Create new stack
        v = v2
        While parent[v] != -1:
            st.push(pair(v, parent[v]))
            v = parent[v]

        while st is not empty:
            print("edge ", st.pop())
    else:
        print("Path does not exists between given pair of points")
```

The following is an illustration of the values of visited, parent array and stack used in the procedure when BFS(1, 6, adjList) is called.



BFS (1, 6, adjlist)

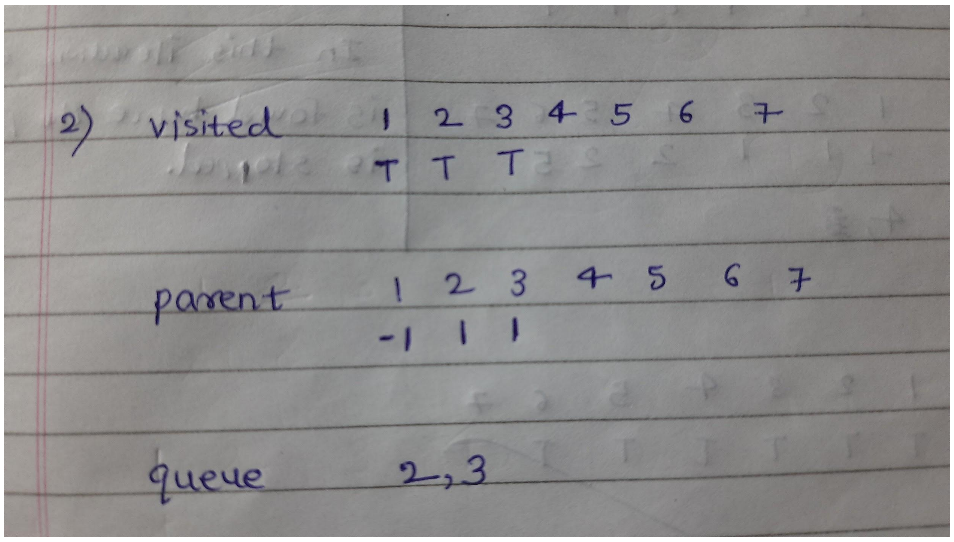
1) visited

1	2	3	4	5	6	7
T						

parent

1	2	3	4	5	6	7
-1						

queue 1



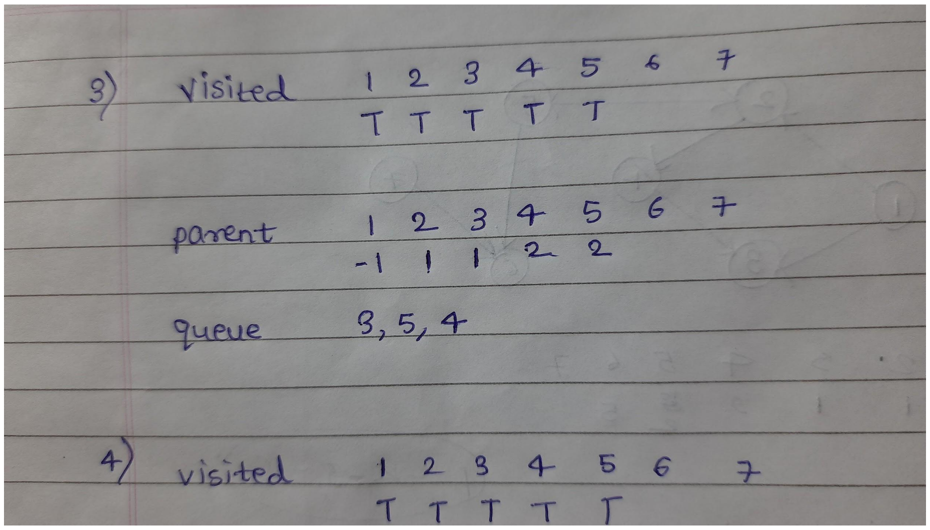
2) visited

1	2	3	4	5	6	7
T	T	T				

parent

1	2	3	4	5	6	7
-1	1	1				

queue 2, 3



3) visited

1	2	3	4	5	6	7
T	T	T	T	T		

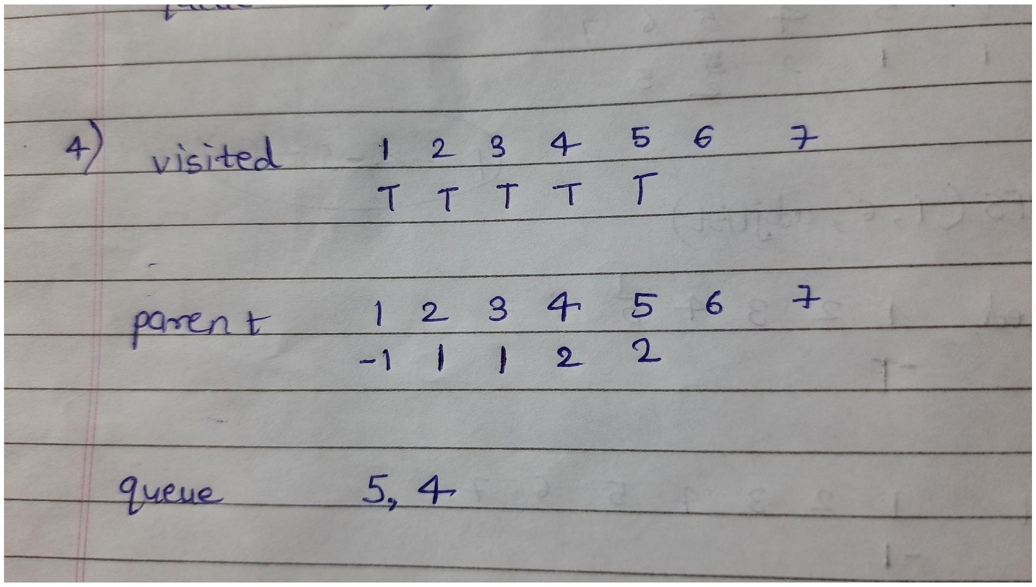
parent

1	2	3	4	5	6	7
-1	1	1	2	2		

queue 3, 5, 4

4) visited

1	2	3	4	5	6	7
T	T	T	T	T		

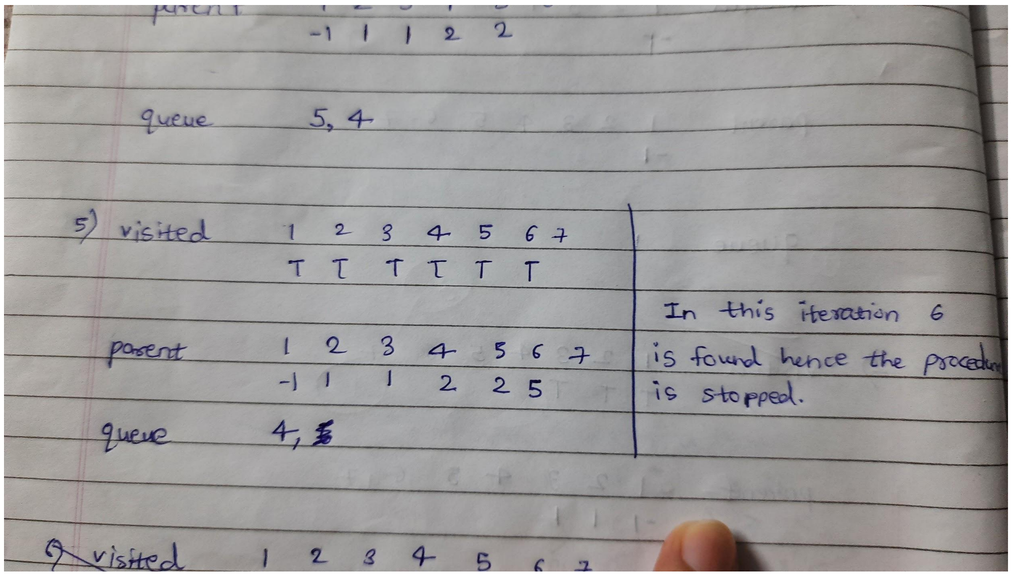


4)

visited	1	2	3	4	5	6	7
	T	T	T	T	T		

parent	1	2	3	4	5	6	7
	-1	1	1	2	2		

queue 5, 4



-1 1 1 2 2

queue 5, 4

5)

visited	1	2	3	4	5	6	7
	T	T	T	T	T	T	

parent	1	2	3	4	5	6	7
	-1	1	1	2	2	5	

queue 4, ~~6~~

In this iteration 6 is found hence the procedur is stopped.

6) visited 1 2 3 4 5 6 7

3. If a digraph (G) is reversed (Gr), how many steps will be required? Estimate it. If only adjacency list representation is available, how many steps would be required? Calculate it. (Hint: Gr contains all the edges of G in reverse direction).

The number of steps or time complexity required to reverse a given digraph will be dependent on how the digraph is represented. For different representations of digraph like adjacency list, adjacency matrix, set of edges etc, the number of steps will be different.

If the digraph is represented using an adjacency list then we can use the following steps to get the reverse of the digraph.

1. Create a new adjacency list, say revAdjList, that will have n cells where n is the number of vertices in the given graph. Initially, all the cells will contain an empty list.
2. For each list in the given adjacency list, traverse the list and add the reverse edge in the revAdjList. For ex. when traversing adjList[i], if we encounter a vertex j then we will add i in the revAdjList[j].

We can see that the above process will traverse every cell of the adjacency list and also every edge of each list. Hence, the time complexity to build a reverse digraph from a given digraph when it is represented in the adjacency list will be equal to O(n + e), where n is the number of vertices and e is the number of edges in the original digraph.