```
n = 8
edges = [[0,1],[0,2],[1,2],[1,3],[1,4],[4,5],[6,7]]

adj_list = [[] for j in range(n)]
for edge in edges:
        adj_list[edge[0]].append(edge[1])
        adj_list[edge[1]].append(edge[0])

# create CSR
offset = [0]*(n+1);
csr_edges = [];
for i in range(n):
        offset[i] = len(csr_edges)
        csr_edges.extend(adj_list[i])
offset[n] = len(csr_edges)
```

```
# BFS
# A bad implementation
def BFS(u):
        visited = [False] * n
        queue = []


        queue.append(u)
        visited[u] = True
        while queue:
                s = queue.pop(0)

                print(s)
                for i in range(offset[s],offset[s+1]):
                        nbr_of_s = csr_edges[i]
                        if visited[nbr_of_s]: continue
                        queue.append(nbr_of_s)
                        visited[nbr_of_s] = True
```

```
# BFS
from collections import deque

def BFS(u):
        visited = [False] * n
        # queue = []
        q = deque()

        q.append(u)
        visited[u] = True
        while len(q) > 0:
                s= q.popleft()
                print(s)
                for i in range(offset[s],offset[s+1]):
                        nbr_of_s = csr_edges[i]
                        if visited[nbr_of_s]: continue
                        q.append(nbr_of_s)
                        visited[nbr_of_s] = True
```

```
# DFS recursive
visited = [False] * n
def DFS_recursive(u):
        print(u)
        visited[u] = True
        for i in range(offset[u],offset[u+1]):
                nbr_of_u = csr_edges[i]
                if visited[nbr_of_u]: continue
                DFS_recursive(nbr_of_u)
```

```
# DFS iterative
def DFS_iterative(u):
        visited = [False] * n
        stack = []
        stack.append(u)

        while (len(stack)):
                s = stack.pop()
                if(visited[u]):
                        continue;

                visited[u] = True
                for i in range(offset[s],offset[s+1]):
                        nbr_of_s = csr_edges[i]
                        if visited[nbr_of_s]: continue
                        stack.append(nbr_of_s)
```

Doing the assignment:

Types of Time complexity:

Worst case | Average | Amortized

How to write pseudocode:

A safe way is to just write any programming language

add any explanation for your pseudocode

```
## connectivity / reachability for undirected graphs
def connectivity(u,v):
        visited = [False] * n
        queue = deque()

        queue.append(u)
        visited[u] = True
        while queue:
                s = queue.popleft()
```

```
                # dequeue.popleft() for bfs & pop() for dfs
                for i in range(offset[s],offset[s+1]):
                        nbr_of_s = csr_edges[i]
                        if nbr_of_s == v: return True
                        if visited[nbr_of_s]: continue
                        queue.append(nbr_of_s)
                        visited[nbr_of_s] = True
        return False

# better idea? Bidirectional BFS
# Why Bidirectional BFS is more efficient?









Output of Connected Components:
assume we have a graph with six vertices: 0,1,2,3,4,5

[
        [0,3,5],
        [1,2],
        [4]
]


[0,1,1,0,2,0]

linear => equivalent






## naive connected component

def naive_cc():
        undiscovered_vertices = [i for i in range(n)]
        result = []

        while undiscovered_vertices:
                current_cc = []
                u = undiscovered_vertices.pop()
                queue = []
                queue.append(u)
                while queue:
                        s = queue.pop(0)
                        current_cc.append(s)
                        for i in range(offset[s],offset[s+1]):
                                nbr_of_s = csr_edges[i]
                                if nbr_of_s in undiscovered_vertices:
                                        queue.append(nbr_of_s)
                                        undiscovered_vertices.remove(nbr_of_s)
                result.append(current_cc)
        return result

# suggestions:







### optimized connected component
```

```
# 1. swap the visited vertex with the last unvisited
...

# 2. doubly linked list
...


# 3. lazy updates

def cc():

        result = []
        visited = [False] * n

        for i in range(n):
                if visited[i]: continue
                current_cc = []
                queue = deque()
                queue.append(i)
                visited[i] = True
                current_cc.append(i)

                while queue:
                        s = queue.popleft()
                        for i in range(offset[s],offset[s+1]):
                                nbr_of_s = csr_edges[i]
                                if visited[nbr_of_s]: continue
                                queue.append(nbr_of_s)
                                visited[nbr_of_s] = True
                                current_cc.append(nbr_of_s)
                result.append(current_cc)

        return result

# print(cc())




### disjoint set

parent = [i for i in range(n)]
size   = [1 for i in range(n)]

def find(x):
        if parent[x] != x:
                parent[x] = find(parent[x])
        return parent[x]

def union(x,y):
        x = find(x)
        y = find(y)
        if x == y:
                return
        if size[x] < size[y]:
                size[y] += size[x]
                parent[x] = y
        else:
                size[x] += size[y]
                parent[y] = x

def ds_cc():
        for edge in [[0,1],[0,2],[1,2],[1,3],[1,4],[4,5],[6,7]]:
                union(edge[0],edge[1])

# ds_cc()
# print(find(1) == find(7)
```

Topological Sort:

```
def ts():
        result = []
        s = []
        in_deg = [0 for i in range(n)]

        # calculate in-degree
        for u in range(n):
                for i in range(offset[u],offset[u+1]):
                        nbr = csr_edges[i]
                        in_deg[nbr] = in_deg[nbr]+1

        for u in range(n):
                if in_deg[u] == 0: s.append(u)

        while s:
                u = s.pop()
                result.append(u)
                for i in range(offset[u],offset[u+1]):
                        nbr = csr_edges[i]
                        in_deg[nbr] = in_deg[nbr]-1
                        if in_deg[nbr] == 0: s.append(nbr)
        return result
```

Queue VS Stack

what if only in-neighbors of each vertex is stored?