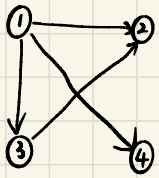
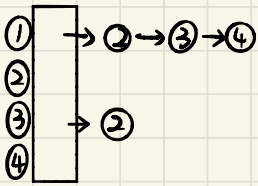
- Graph

$G = (V, E)$ V: Set of vertices $n = |V|$
E: Set of Edges $m = |E|$



Graph Representation

1. Adjacency List



Space : $O(n+m)$ m could be as large as n^2

2. Adjacency Matrix

	1	2	3	4
1		T	T	T
2				
3		T		
4				

$$A[u,v] = \begin{cases} T & , \exists (u,v) \in E \\ F & \text{otherwise} \end{cases}$$

Space : $O(n^2)$

operation	List	Matrix
existance of edge	$O(d)$ d: # of neighbors of u	$O(1)$
traverse neighbors of u	$O(d)$	$O(n)$

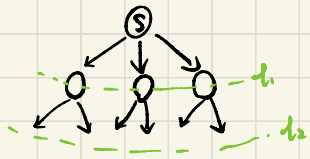
Graph Search (For DAG only)

- BFS (Breadth First Search)
- DFS (Depth First Search)

```
BFS ( G , S) {   // S is the source vertex
    q.enqueue(s);                Modify to record the shortest path
    visited[s] = True;           dist[s] = 0;   π[s] = -1
    while ( !q.isEmpty() ) {
        u = q.dequeue();
        // Some Processing
        for ( v in G.neighbors(u) ) {      Adj list is prefered
            if ( ! visited[v] ) {
                q.enqueue(v);        dist[v] = dist[u] + 1;
                visited[v] = True;   π[v] = u;
            }
        }
    }
```

Running Time: $O(m)$

$O(n+m)$ is also true considering the init of array



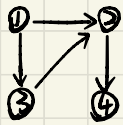
parent π

dist

```
BFS-ALL (G) {        O(n+m)
    visited[1, ..., n] = False
    for (s = 1, ..., n) {
        if ( !visited[s] ) {
            BFS ( G , s , visited );
        }                    shared
    }
```

- DFS

```
DFS(G, u, visited[]){
    visited[u] = T;
    // processing with vertex u
    for( v in G.neighbor(u)){
        if(!visited[v]){
            DFS(G, v, visited);
        }
    }
}
```

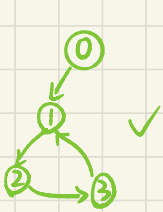


visit order
1 2 4 3

using DFS to solve connect component

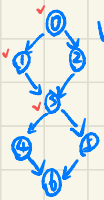
```
DFS_ALL(G){
    visited[1,...,n] = False
    cc[1,...,n] = 0; cnt = 1;
    for(s = 1... n){
        if(!visited){
            DFS(G, s, visited, cc, cnt);
            cnt += 1;
        }
    }
}
```

```
in DFS(G, s     ){
    visited[s] = T;
    cc[s] = cnt;
}
```



- Cycles (Detect cycle)


```
DFS(G, u){
    visited[u] = T;
    for( v in G.neighbor(u)){
        if( visited[v] = True)
            cycle!!!
        else
            DFS(G, v);
    }
```



visited[u] = F; // works, but produce bad running time
// which will be exponential

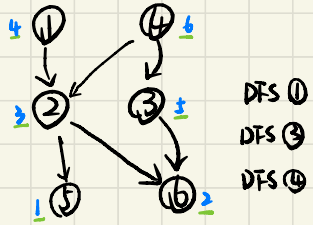
White : new vertex
Grey : on recursion call stack
Black : visited, not on a call stack

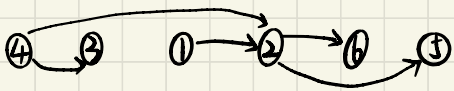
```
bool DFS(G, u, color[]){    O(n)
    color[u] = Grey;
    for( v in G.neighbor(u)){
        if(color[v] == white){
            DFS(G, v, color[]);}
        elif(color[v] == Grey){
            Cycle!!!
        else{
            // do nothing;}
    }
    color[u] = Black;
}
```

Directed Graph

DAG Directed Acyclic Graph

Topological Sort
All the edges are pointing from left to right





Claim: Sorting the finish time by decreasing order, we will get the topological order.

#1. BFS

Record finish time for each vertex

```
DFS-ALL(G){                                   O(n+m)
    visited[1,...,n]
    finish[1,...,n]
    timer = 1;
    for(u=1,...,n){
        if(!visited[u]){
            DFS(G, u, visited, finish, timer);
        }
    }
    return l.reverse();
}

DFS(G, u, visited, finish, timer){
    visited[u] = T;
    for(v in G.neighbor(u)){
        if(!visited[v]){
            DFS(G, v, visited, finish, timer);
        }
    }
    finish[u] = timer;
    timer += 1;  // timer is the number of DFS being called
}
```

l

⇒ l.append(u) // l will have the reverse order of topological sort of list

(u) ⟶ (v)

case 1. DFS(u) first

case 2. DFS(v) first

In either case, the finish time of (u) is always greater than (v)

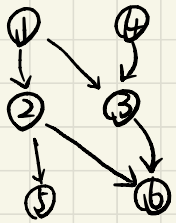
#2. Indegree + BFS

First, find ones that have no dependencies

⇒ 1, 4 . (remove)

⇒ 3, 2 (remove)

⇒ 5, 6 (...)



① Calculate the indegree of all vertices in G

```
for(u=1,...,n){                    O(n+m)
    for(v in G.neighbor(u)){
        indegree[v]++;
    }
}

for(u=1,...,n)                     O(n)
    if(indegree[u]==0)
        q.enqueue(u);

while(!q.isEmpty()){               O(n+m)
    u=q.poll(); print(u)
    for(v in G.neighbor(u)){
        indegree[v]--;
        if(indegree[v]==0)
            q.enqueue(v);
    }
}
```

	1	2	3	4	5	6
indegree	0	1	2	0	1	2

q: ~~1~~ ~~4~~ 2 3 5 6

① ④ ② ③ ⑤ ⑥

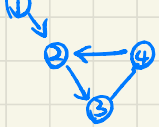
Modify to detect cycle

l.append(u)

if l.length < n ⇒ cycle

else No cycle.

If there is a cycle



-Boggle

You're given a dictionary of valid word and matrix.
↔ directed. find all valid words.

4

4	H	E	A	T
	E	L	O	E
	U	Y	X	B
	P	P	A	S

```
Boggle (B[ ]) {
    for(i=1,...,4)
        for(j=1,...,4){
            DFS(B[], visited[], i, j, "", res);
        }
    }
    return res;
}

DFS(B[], visited[], i, j, cur, res){
    visited[i,j] = T;
    potential = cur + B[i,j];
    if(dict.has(potential))   // use Trie
        res.add(potential);
    if(i>1 && !visited[i-1,j]) DFS(B[], visited[], i-1, j, potential, res);
    if(j>1 && !visited[i,j-1]) DFS(B[], visited[], i, j-1, potential, res);
    if(i<m && !visited[i+1,j]) DFS(B[], visited[], i+1, j, potential, res);
    if(j<n && !visited[i,j+1]) DFS(B[], visited[], i, j+1, potential, res);
    visited[i,j] = F
}
```

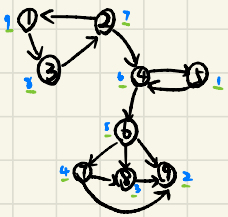
- Scc (Strongly connected component)

Weakly connected component ∀ u,v u→v or v→u

Strongly connected component ∀ u,v u→v and v→u

Reverse edge direction

① In the reversed graph, starting from the one with largest finishing time.
Running DFS in G^T



Scc1 {6,7,8,9}
Scc2 {4,5}
Scc3 {1,2,3}

② Get finish time in G^T
[6,9,8,7,4,5,1,2,3]
Running DFS on G in the order of decreasing order of finish time we get from G^T.