Topic : Connectivity ① Paths ② Cycles * ③ Sorting on graph.

① Paths.

Q. graph → v_1 and v_2 ⇒ if there is a path bet. those 2 nodes or not. if yes, return me path.

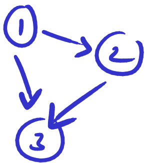
* undirected — easy

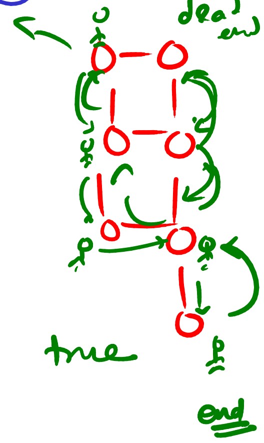
* directed — tricky

logic : generic

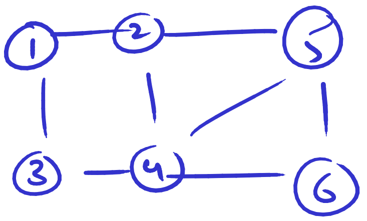
```
bool getPath ( g , v1 , v2 , visited , path ) :
    path.push(v1)
    visited [v1] = true
*   if v1 == v2 : return true          ↙ line ×
    for each neighbour v in g[v1] :
        if ! visited[v] :
*           if getPath (g, v, v2 , visited, path) : return true
    path.pop()
    return False
```

this line.
↓





Why am I getting only 1 path ?



1 2 5 4
1 2 4
1 3 4
1 2 5 6 4

① to ④

s ~ v_1
d ~ v_2

```
void printAllPaths (g, s, d) :
    visited [n] = {false}
    all_paths = [ ]
    path = [ ]
    dfsVisit (g, s, d, path, all_paths, visited)

dfsVisit (g, s, d, path, all_paths, visited) :
    path.push(s) ;      visited [s] = true ;
    if s==d : all_path.push (path)      return
    for all v in g[s] :     if ! visited [v] : dfsVisit (g, v, d, path, all_paths, visited)
    path.pop() ;     visited [s] = false ;
```

② Cycles.

a) undirected

b) directed

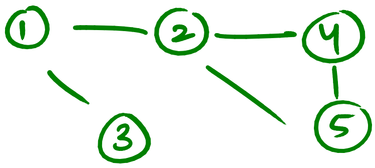
undirected graphs:

n nodes

n-1 edges.

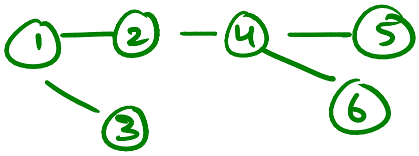
components = 1

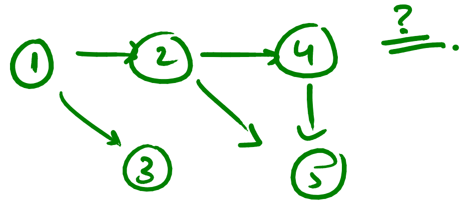
edges == n-1



cycle. ? Yes

no cycles : tree-like.





cycle ? No.

Debate: DFS if I come across a node visited, cycle is there.

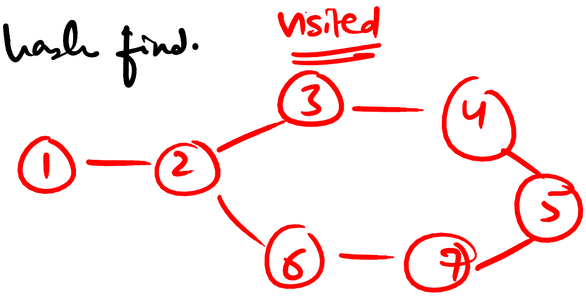
Yes.

undirected

Q. directed ? same ? or not ?

recursion_stack. set | find.

O(1) hash find.

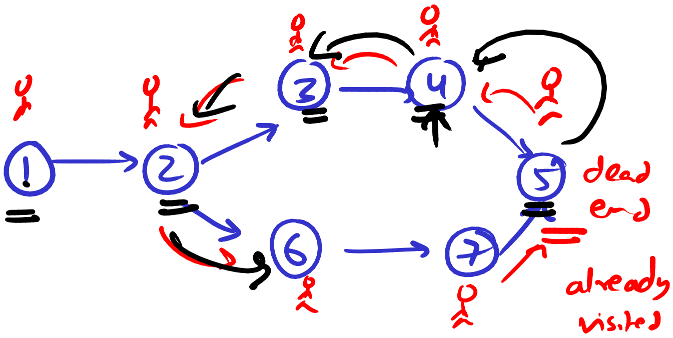


rec stack

[1 2 ~~3~~ ~~4~~ ~~5~~ 6 7]

visited

[1 2 3 4 5 6 7



cycle ⇒ ? No. ↓ Yes. cycle

logz

cycle: if v is in stack : cycle | Backedge

Undirected : DFS ↓

```
bool hasCycle ( g ):
    visited [n] = {false}
    for i=0 i<n i++
        if !visited [i] :
            if hasCycleVisit(g, i, visited)
                return true
    return false
```

```
bool hasCycleVisit ( g, u, visited):
    visited [u] = true
    for each v in g[u] :
        if !visited [v] :
            if hasCycleVisit(g,v,..)
                return true
        else:
            return true
    return false
```

directed :

```
bool hasCycle (g):
    visited [n] = {false}
    rec_stack = set()

    for i=0 i<n i++
        if hasCycleUtil (g, i, visited, rec_stack)
            return true

    return false.
```

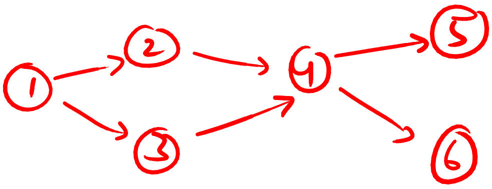
```
bool hasCycleUtil (g, u, visited, rec_stack)
    if !visited[u] :
        visited [u] = true    rec_stack.insert
                                           (u)
        for each v in g[u] :
            if !visited [v] and
               hasCycleUtil (g, v, vis, rec)
              return true

            else if v in rec_stack:
                return true

    rec_stack.remove (u)
    return false
```

③ sorting on graphs.

Real world : DAGs → Directed acyclic graphs.

Why?

① Chronology | Dependency .



Package managers :

JS/TS → npm, yarn
Python → pip
Java → maven
Rust → cargo

} what is this?
How they operate ?

npm i react   node_modules/

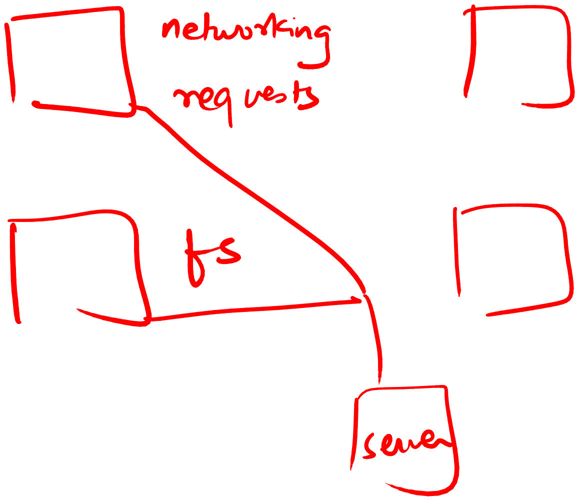
pip i tensorflow   pip freeze > requirements.txt

Can I use my lang for dev? → No!!

py

Community .

lang



registries

npmjs.com } packages

server } calc. dependencies

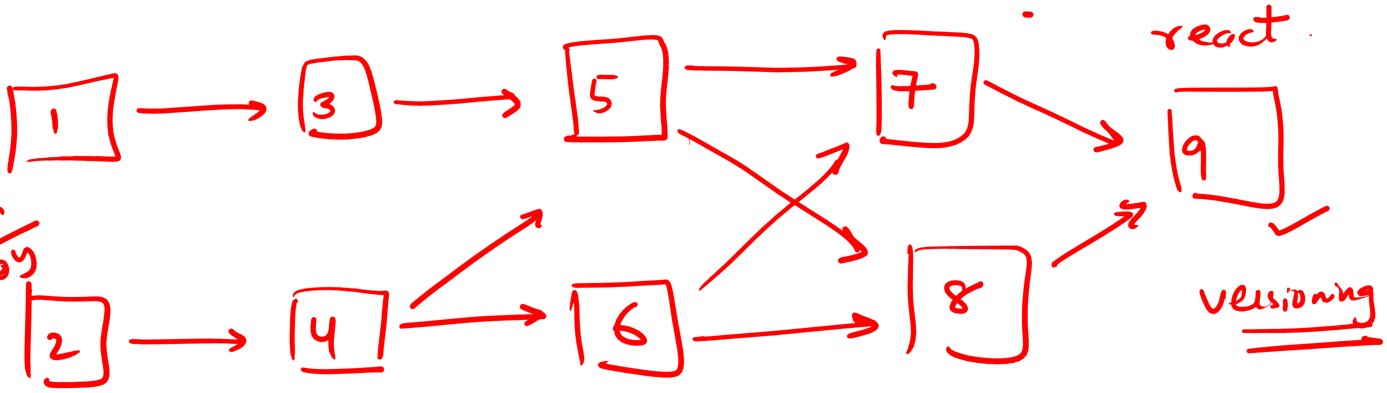
python requirements.txt

js package.json

DAG

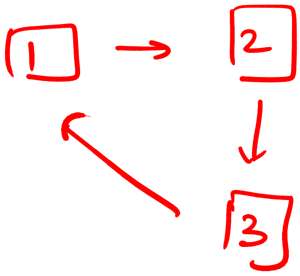
illustration

chronology



npm i react.

DFS to construct dependency graph.



chicken-egg problem / cyclic dependency

S_1
- depends on: S_2

S_4

S_2
- depends on: S_3

S_5

X cyclic dependency

S_3
- depends on: S_1

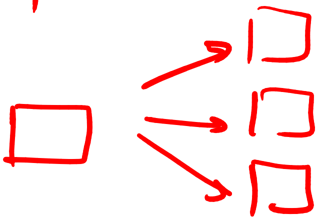
S_6

Docker containers

docker-compose.yml

AWS / Cloud

master-slave



n/w of computers

↓

DAG

① Create a graph → ensure no cycle. | Ensure DAG.

② install the libraries (eg pip, npm)

What should be the order to install lib ?

1 3 2 4 5 6 8 7 9   npm i pkg

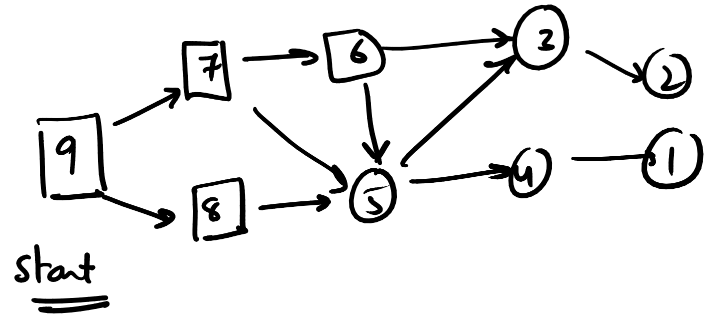
Sorted order in DAG → topological sort. | no unique topo sort.

* Cond^n to install package X, dependencies should be there before it.

DFS.   hint a DS.

Stack.

How?



pseudo :   DFs → Backtrack.

```
void topoSort(g):
    stack s;    visited[n] = {false}
    for i=0 i<n i++
        if !visited[i]
            toposortVisit(g, i, visited, s)

    // print the stack
    while !s.empty():   print(s.top())
                        s.pop()
```

```
toposortVisit(g, u, visited, s):
    visited[u] = true
    for each v in g[u]:
        if !visited[v]:
            topoSortVisit
            (g, v, ...)
    → s.push(u)
```

3 2 5 4 1