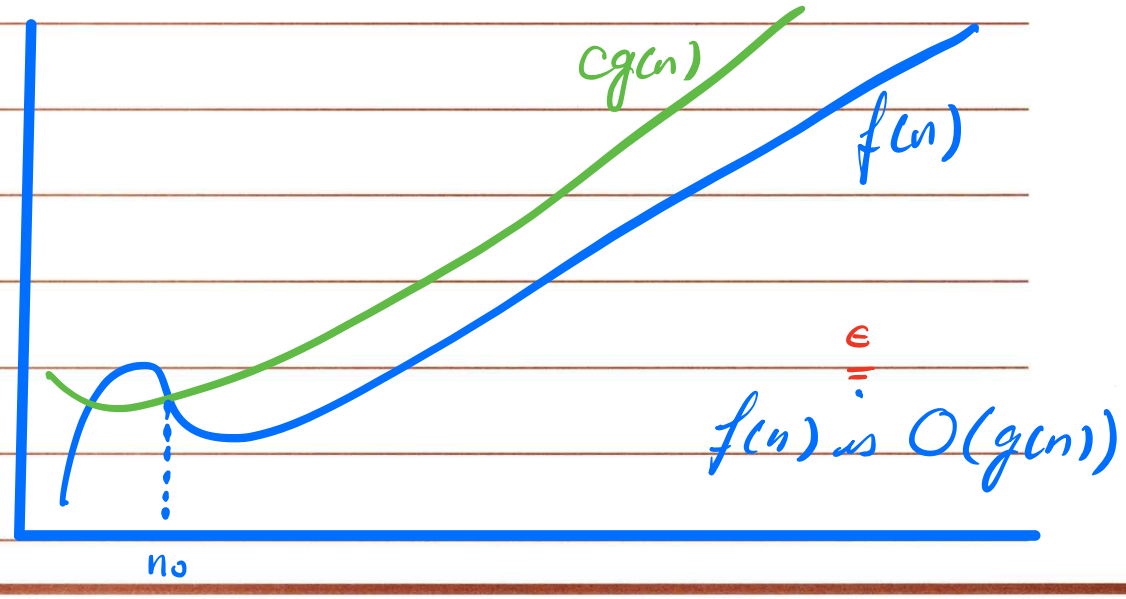
Asymptotic Notation

Def. $O(g(n)) = \{f(n) \mid$ there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$



True or False?

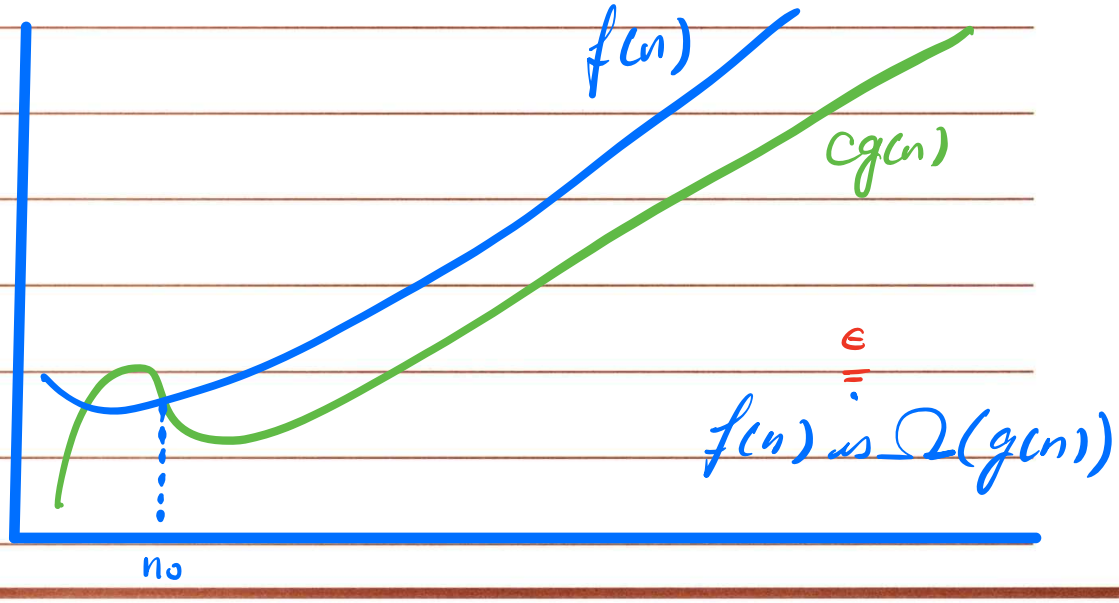
T Every quadratic function (in n) is $O(n^2)$

T Every linear function (in n) is $O(n^2)$

F Every cubic function (in n) is $O(n^2)$

In general, let $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ where $a_0, a_1, \ldots, a_n$ are real numbers. Then $f(n)$ is $O(n^k)$

Def. $\Omega(g(n)) = \{f(n) \mid$ there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$



True or False?

T Every quadratic function (in n) is $\Omega(n^2)$

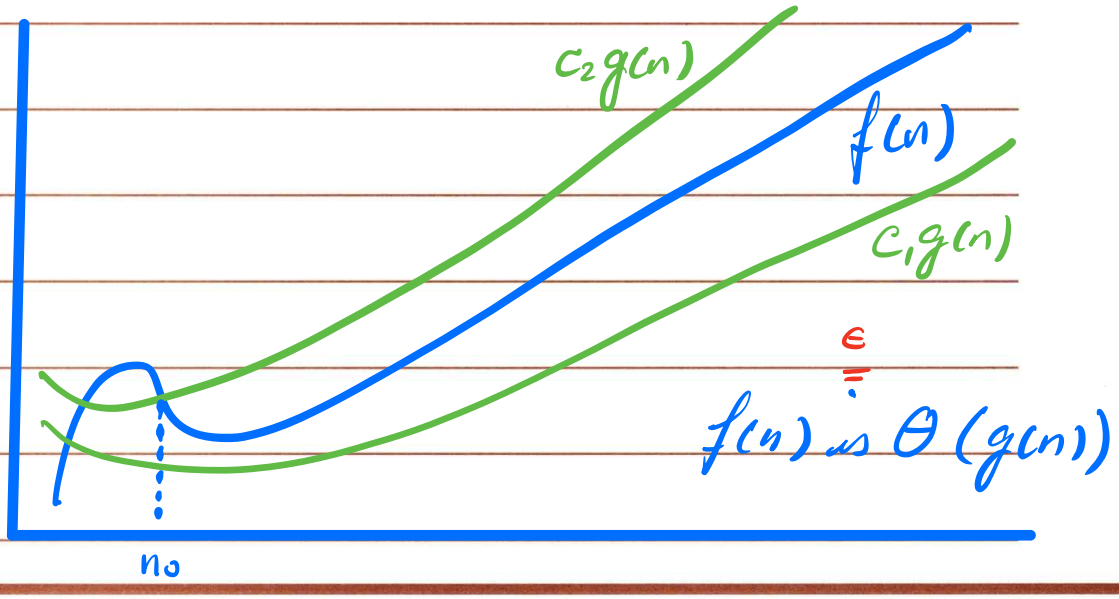
F Every linear function (in n) is $\Omega(n^2)$

T Every cubic function (in n) is $\Omega(n^2)$

In general, let $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ where $a_0, a_1, \ldots, a_n$ are real numbers & $a_k > 0$

Then $f(n)$ is $\Omega(n^k)$

Def. $\Theta(g(n)) = \{ f(n) \mid$ there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$



True or False?

T Every quadratic function (in n) is $\Theta(n^2)$

F Every linear function (in n) is $\Theta(n^2)$

F Every cubic function (in n) is $\Theta(n^2)$

In general, let $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ where $a_0, a_1, \ldots, a_k$ are real numbers & $a_k > 0$

Then $f(n)$ is $\Theta(n^k)$

Other asymptotic notations

$f(n) = o(g(n))$ means

$f(n) = O(g(n))$, but $f(n) \neq \Theta(g(n))$

$f(n) = \omega(g(n))$ means

$f(n) = \Omega(g(n))$, but $f(n) \neq \Theta(g(n))$

	Worst Case	Best Case
Linear Search	$O(n), \Omega(n), \Theta(n)$	$O(1), \Omega(1), \Theta(1)$
Binary Search	$O(\lg n), \Omega(\lg n), \Theta(\lg n)$	$O(1), \Omega(1), \Theta(1)$

	Worst Case	Best Case
Insertion Sort	$O(n^2), \Omega(n^2), \Theta(n^2)$	$O(n), \Omega(n), \Theta(n)$
Merge Sort	$O(n \lg n), \Omega(n \lg n), \Theta(n \lg n)$	$O(n \lg n), \Omega(n \lg n), \Theta(n \lg n)$

Comparing growth of functions

$f_1(n) = 3^n n^2 \lg^5 n$

$f_2(n) = 2^n n^8 \lg n$

Which function grows faster?

- First compare the exponential components
- only if the exponential components are the same, compare the polynomial components.
- only if the exponential and polynomial components are the same, compare the polylogarithmic components

Two algorithms A and B that solve the same problem have the following worst-case runtime complexities

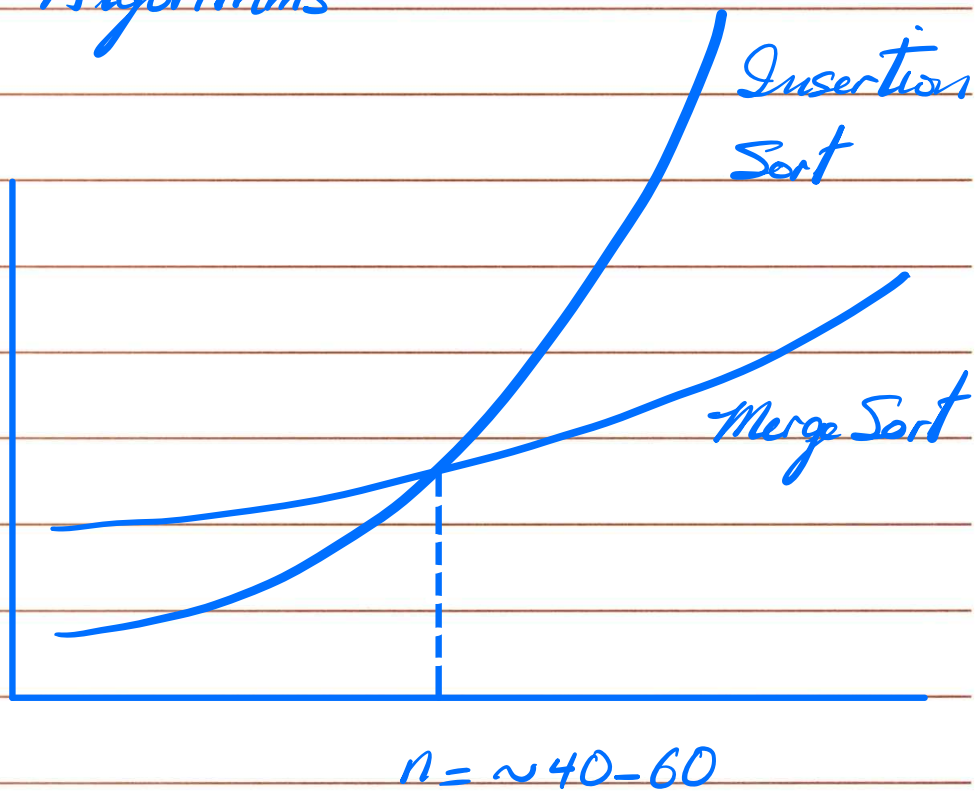
Algorithm A: $O(3^n n^2 \lg^5 n)$

Algorithm B: $O(2^n n^8 \lg n)$

Which algorithm runs faster?

The asymptotic bounds on runtime are not enough information to determine which algorithm runs faster for a given input set.

Hybrid Algorithms



Can combine the two sorting algorithms to produce an algorithm that is faster than either one.

Average Case Analysis

Why don't we perform average case performance analysis?

Average performance of an algorithm is dependent on the characteristics of its input in a given environment

We cannot study average case performance without this knowledge

Review of BFS & DFS

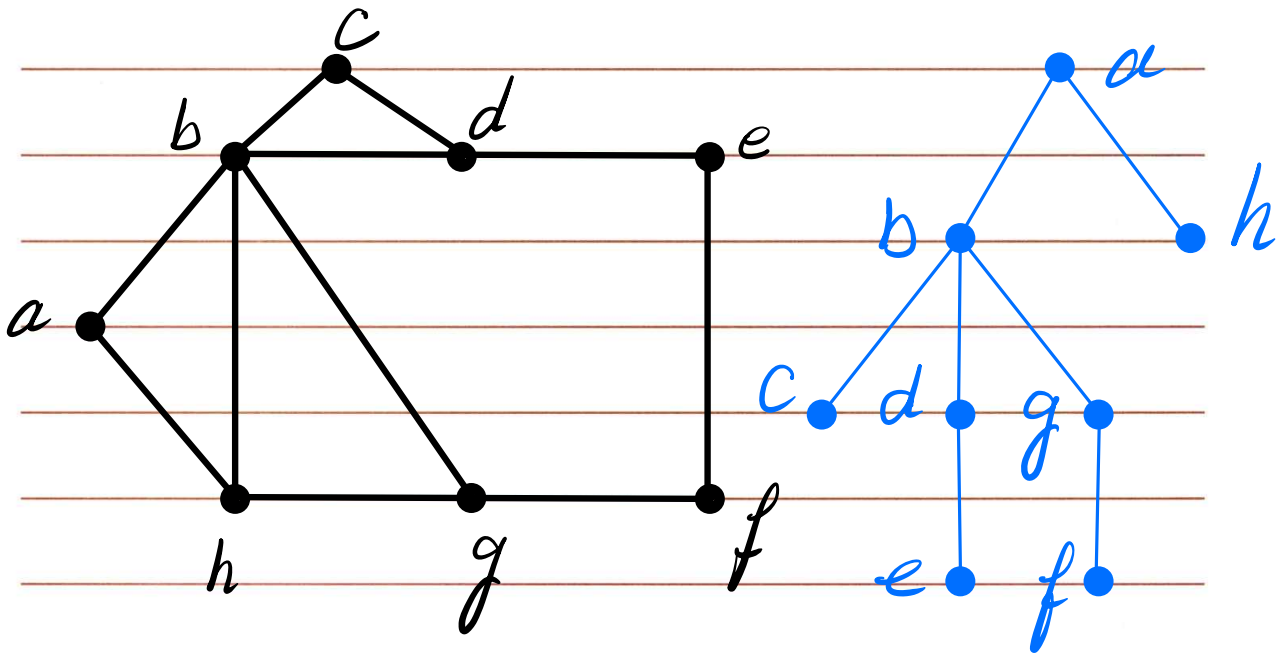
Q: What are we searching for?

- To find out if there is a path from node A to node B.
- To find all nodes that can be reached from A.

Graph Search Algorithms

Breadth First Search (BFS)

First finds the immediate neighbors of the starting point (s) (@ distance 1 from s). Then finds nodes @ distance 2 from s, etc.



BFS Tree

BFS Algorithm

$\Theta(m+n)$

$\Theta(n)$

```
BFS(node s)
    Initialize d(v) = false, for all v∈V
    d(s) = true
    Queue q
    q.enqueue(s)
    while(!q.isEmpty())
        node u = q.dequeue()
        for(all outgoing edges from u: u→w)
            if(!d(w))
                d(w) = true
                q.enqueue(w)
            end if
        end for
    end while
```

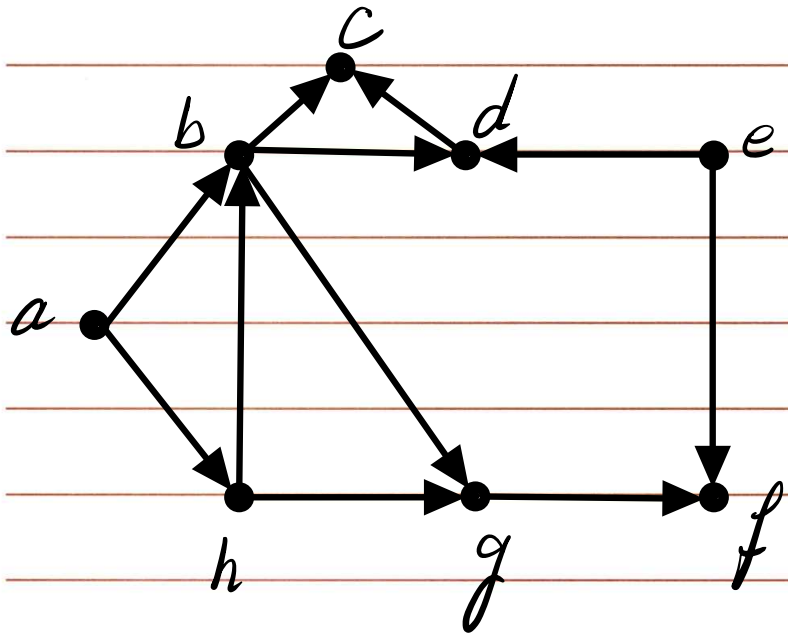
$\sum_{u \in V} d^+(u) = \Theta(m)$

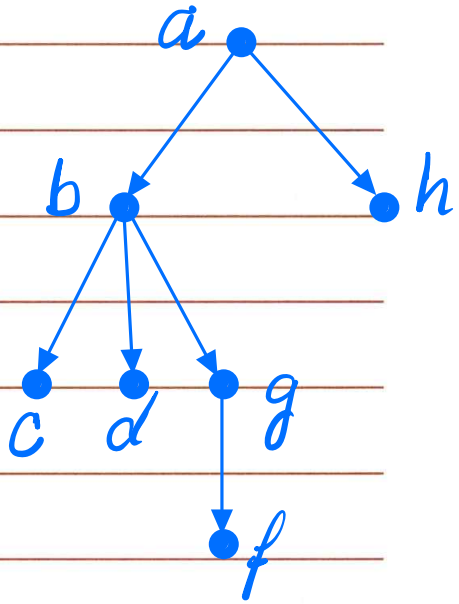
$\Theta(d^+(u))$

Graph Search Algorithms

Depth First Search (DFS)

Follows a path until it hits a dead-end. Then backtracks, until it finds a new path it can take.





DFS Tree

DFS Algorithm

$\Theta(m+n)$

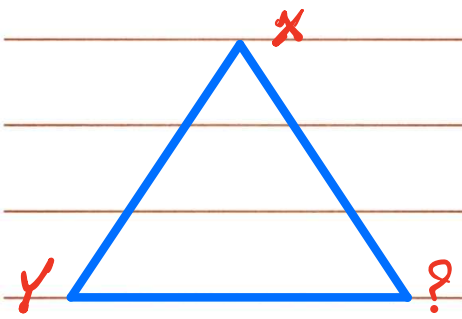
$\Theta(n)$

```
DFS(node s)
    Initialize d(v) = false, for all v∈V
    Stack st
    st.push(s)
    While (!st.isEmpty())
        node u = st.pop()
        if (!d(u))
            d(u) = true
            for (all outgoing edges from u: u → w)
                st.push(w)
            endfor
        endif
    endwhile
```

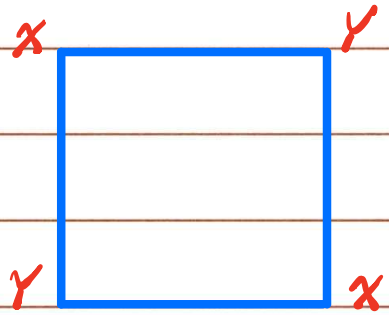
$\Theta(d^+(u))$

$\sum_{u \in V} d^+(u) = \Theta(m)$

Q: Given an undirected connected graph G, how do you determine if G is bipartite?

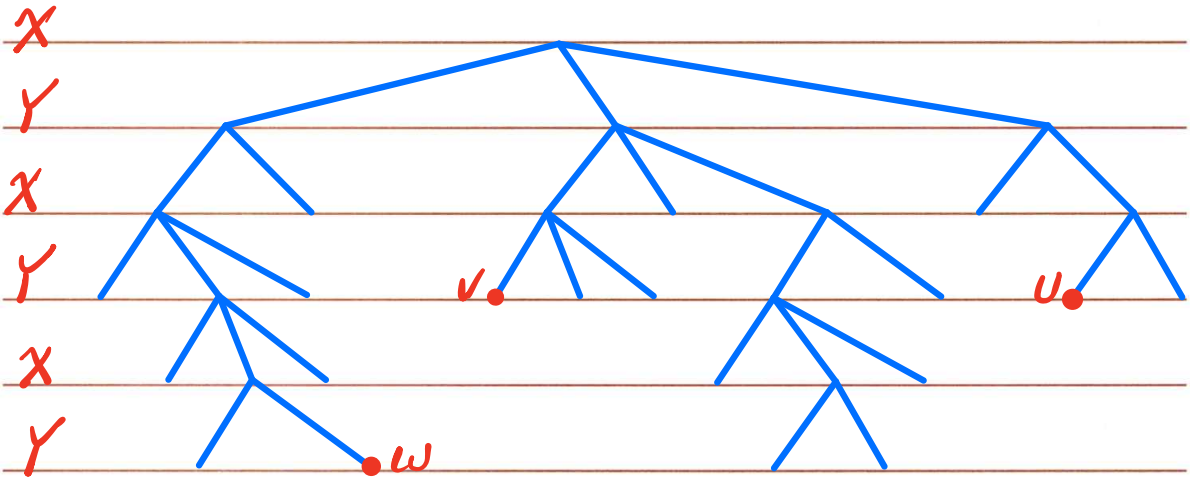


Not bipartite



bipartite

Starting from any node s in G, perform a BFS search.

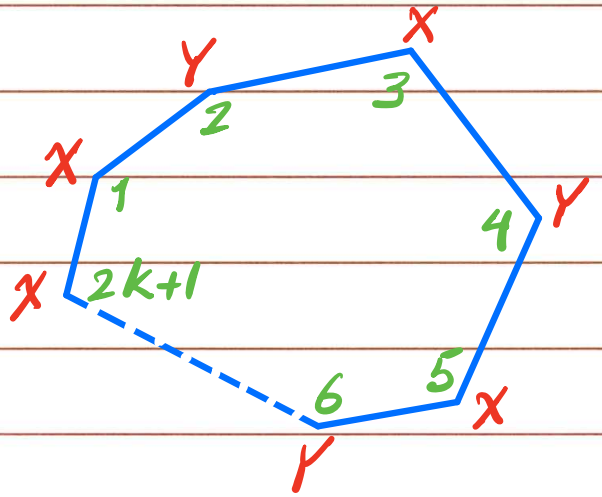


Observations:

- Not possible to have an edge between V & W in G (since they are more than one level apart in the BFS tree.)

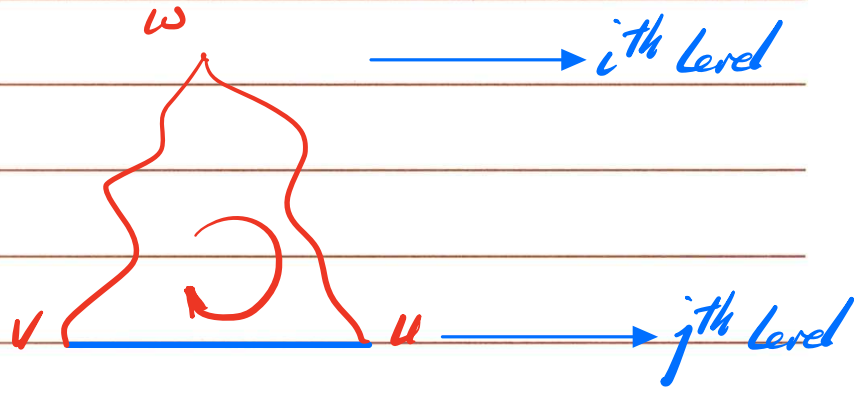
- Edges in G that end up with both ends in the same set (X or Y) must have both ends at the same level in the BFS tree, e.g. between V & u.

Consider a cycle with an odd number of edges



FACT: If a graph G is bipartite, then it cannot contain an odd cycle.

Suppose we find an edge in G with both ends in the same set. Let w be the lowest common ancestor of v & u.



length of this cycle is 2*(j-i)+1, which is odd!

Solution:

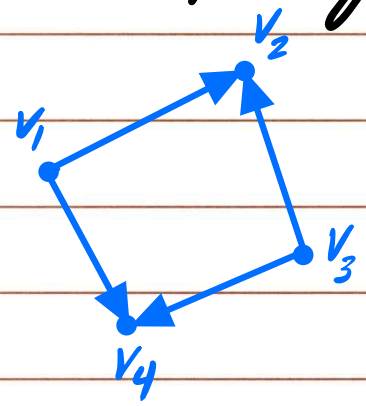
O(m+n) { Run BFS starting from any node, say s. Label each node X or Y depending on whether they appear at an odd or even level on the BFS tree.

O(m) { Then, go through all edges and examine the labels at the two ends of the edge. If all edges have one end in X and the other in Y, then the graph is bipartite. Otherwise it is not bipartite.

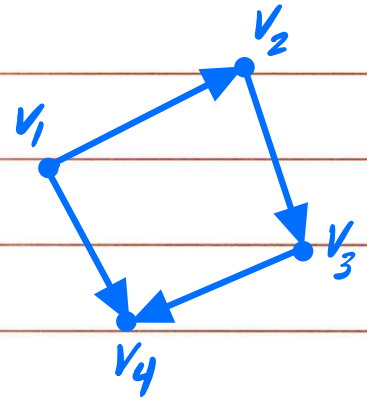
overall complexity = O(m+n)

Connectivity in a directed graph

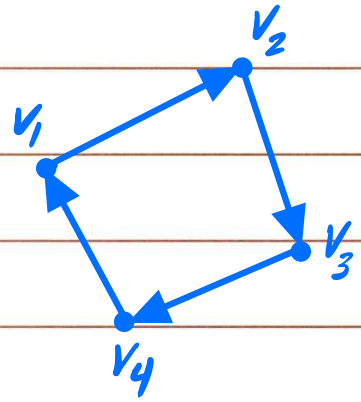
A directed graph is weakly connected if there is a path from any point to any other point in the graph by ignoring edge directions.



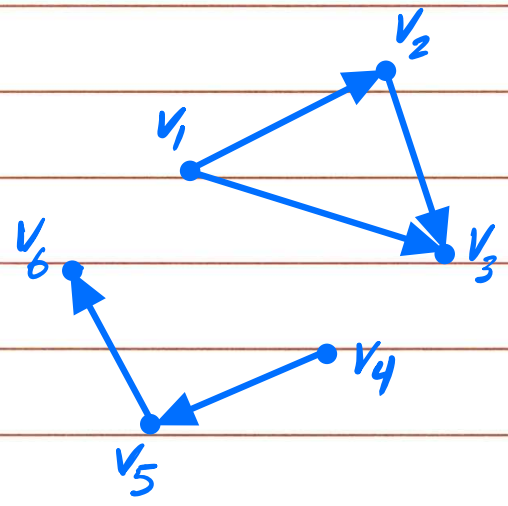
A directed graph is connected if, for any pair of nodes (v, u), there is either a path from v to u, or a path from u to v.



A directed graph is strongly connected if, for any pair of nodes (v, u), there is a path from v to u, and a path from u to v.



A directed graph is not connected (unconnected) if it is not connected in any of the above 3 ways.

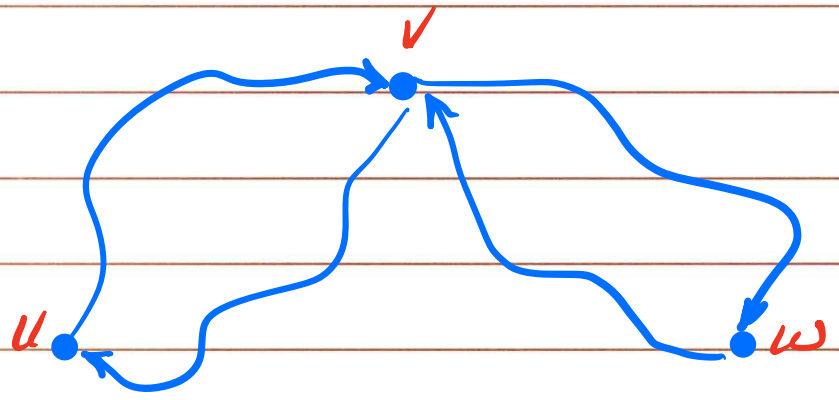


Q: How can you determine if a given graph G is strongly connected?

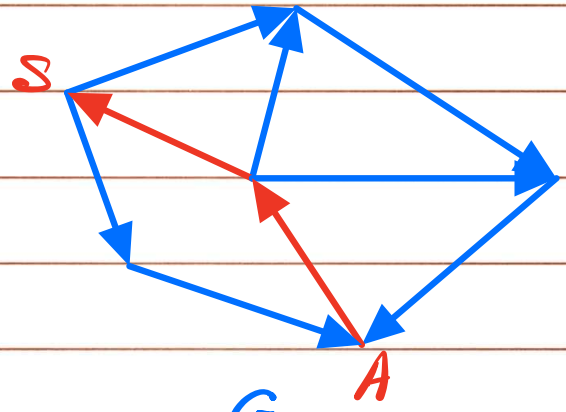
Brute force approach:
Run BFS or DFS from every node.
If in each search, all nodes are found then G is strongly connected

Takes $O(n^2+nm)$

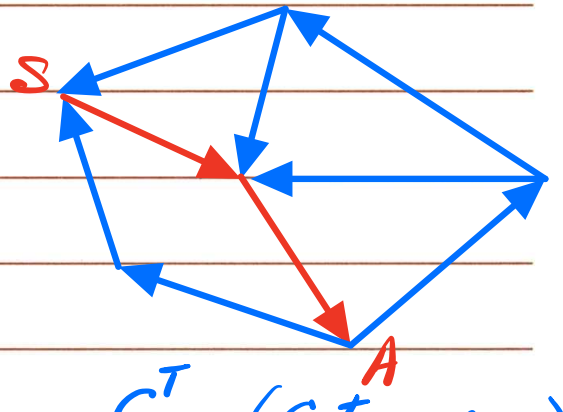
Can we do better?



- if v and u are mutually reachable, and
- v and w are mutually reachable, then
- u and w are mutually reachable.



G



G^T (G transpose)

If there is a path from s to A in G^T, then this path goes from A to S in G.

Solution:

O(m+n) { 1- Use BFS or DFS to find all nodes reachable from s (an arbitrary node) in G.
If some nodes are not reachable from s, stop. The graph is not strongly connected.
Otherwise, continue with step 2.

O(m+n) 2- Create G^T

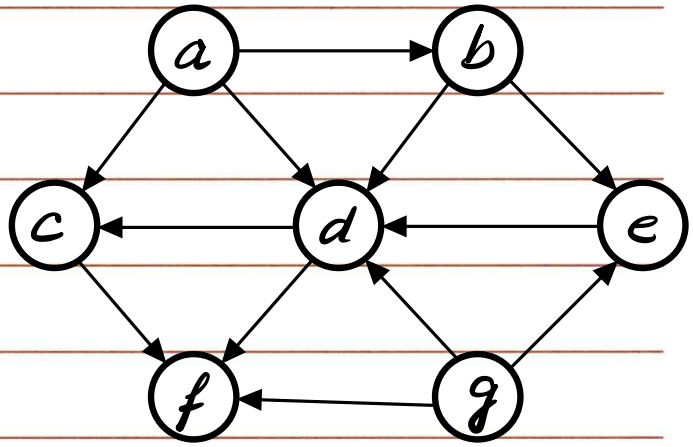
O(m+n) { 3- Use BFS or DFS to find all nodes reachable from s in G^T.
If some nodes are not reachable from s, stop. The graph is not strongly connected.

Otherwise, The graph is strongly connected. (since all nodes are mutually reachable through s)

Overall Complexity = O(m+n)

Finding a Topological Ordering (in a DAG)

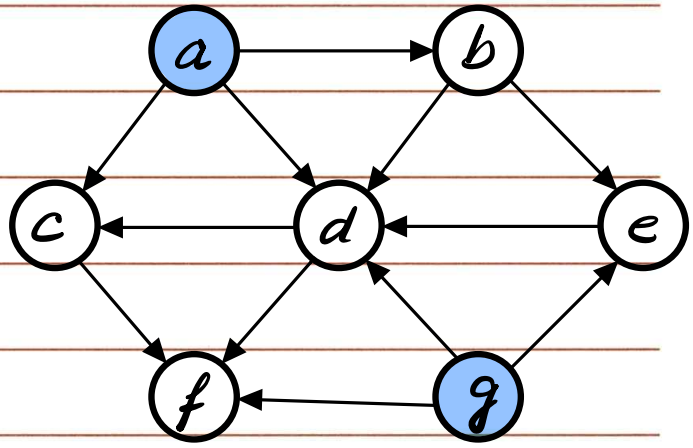
Which node(s) can be first in the topological order?

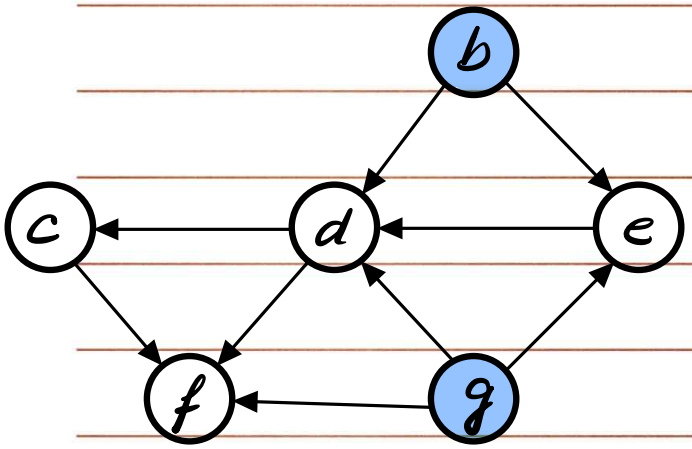


High Level Algorithm

- Find a node v with no incoming edges and order it first.
- Delete v from the graph (G)
- Recursively compute the topological ordering of G-v and append this order after v.

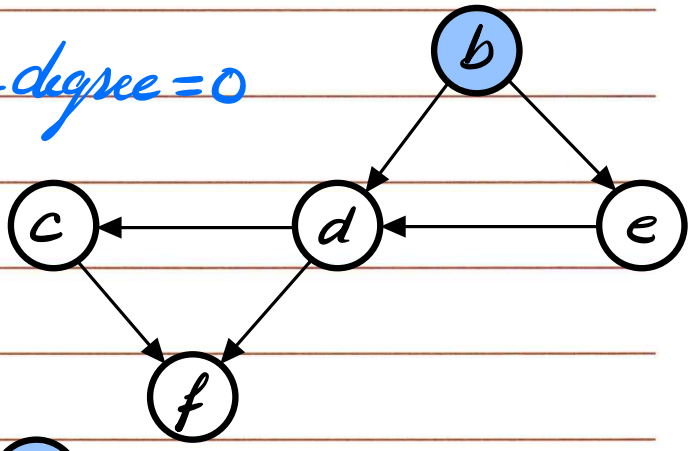
Candidate nodes are a and g.
Remove a.

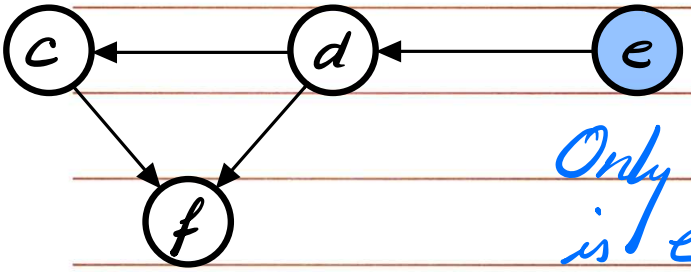




Candidate nodes are b and g.
Remove g.

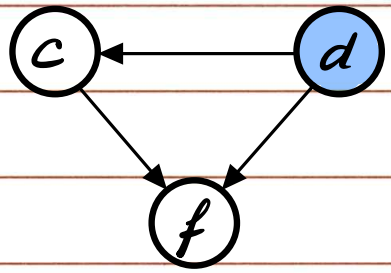
Only node with in-degree=0 is b.
Remove b.

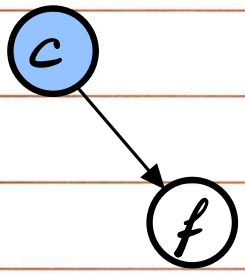




Only node with in-degree=0 is e.
Remove e.

Only node with in-degree = 0 is d.
Remove d.





Only node with in-degree = 0 is c. Remove c.

f is the last node in this topological order

Implementation Plan:

We will compute and maintain two things:

- For each node w, the number of incoming edges.
- The set S of all active nodes in G with no incoming edges.

topologicalOrdering (G)

$\Theta(n)$ Initialize the in-degree of all nodes to 0

$\sum_{v \in V} d^+(v) = \Theta(m)$

  for all v in V

    $\Theta(d^+(v))$

    for all w in Adj(v)

      increment the in-degree of w by 1.

    endfor

  endfor

$\Theta(1)$ Create empty list S

$\Theta(n)$

  for all v in V

    if in-degree(v) = 0, Add v to S

  endfor

$\sum_{v \in V} d^+(v) = \Theta(m)$

  for i = 1 to n

    remove any node v from S

    place v in position i in the topological order

    $\Theta(d^+(v))$

    for all w in Adj(v)

      decrement the in-degree of w by 1.

      if in-degree(w) = 0, Add w to S

    endfor

  endfor

Overall Complexity = $O(m+n)$