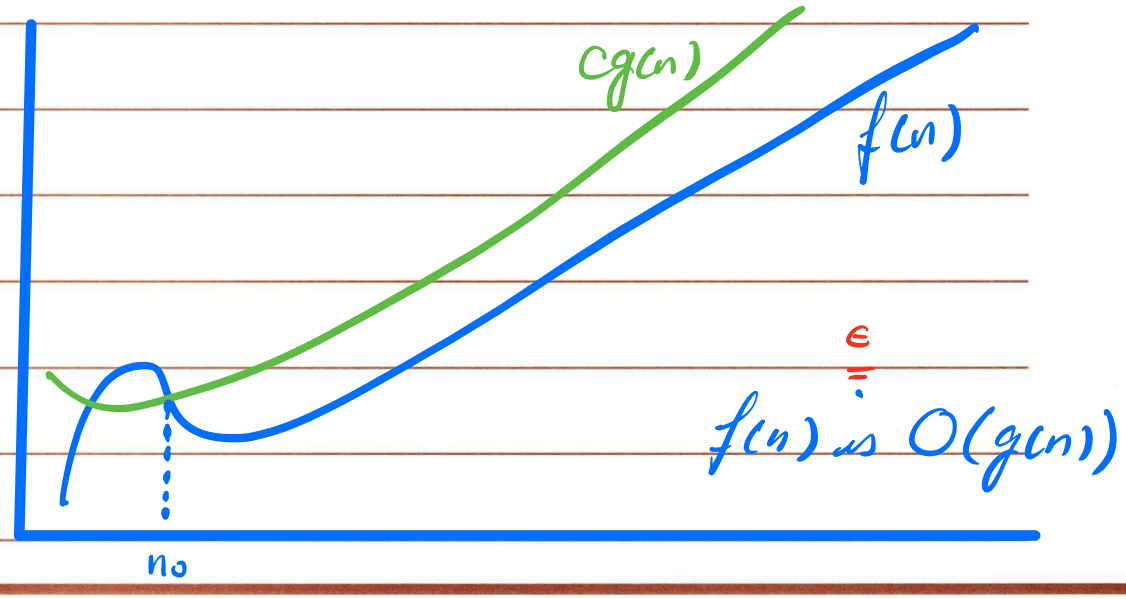
Asymptotic Notation

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?

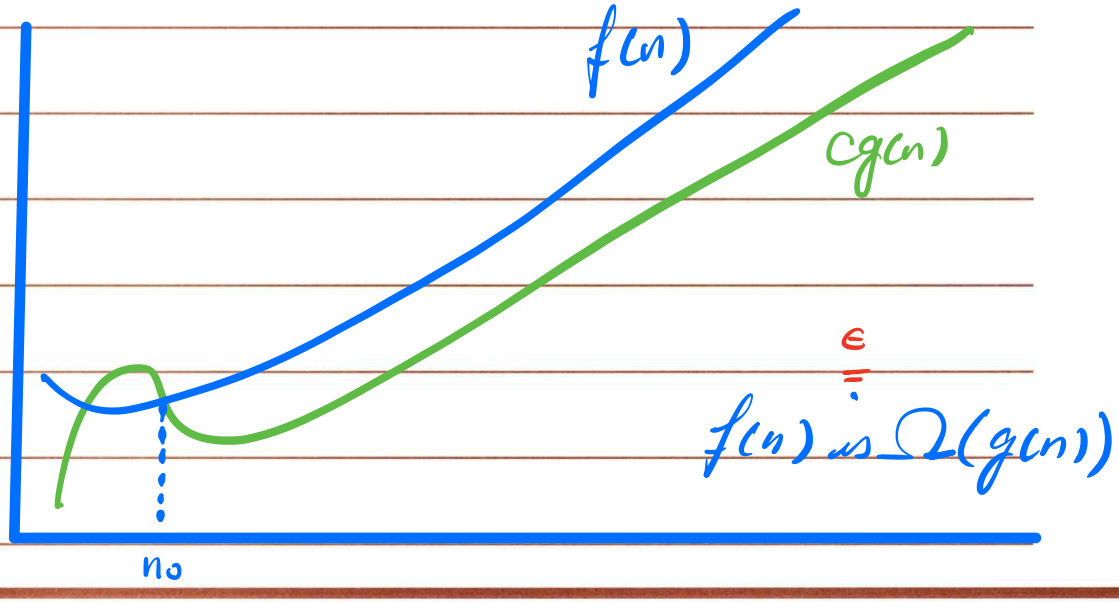
T Every quadratic function (in n) is $O(n^2)$

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?

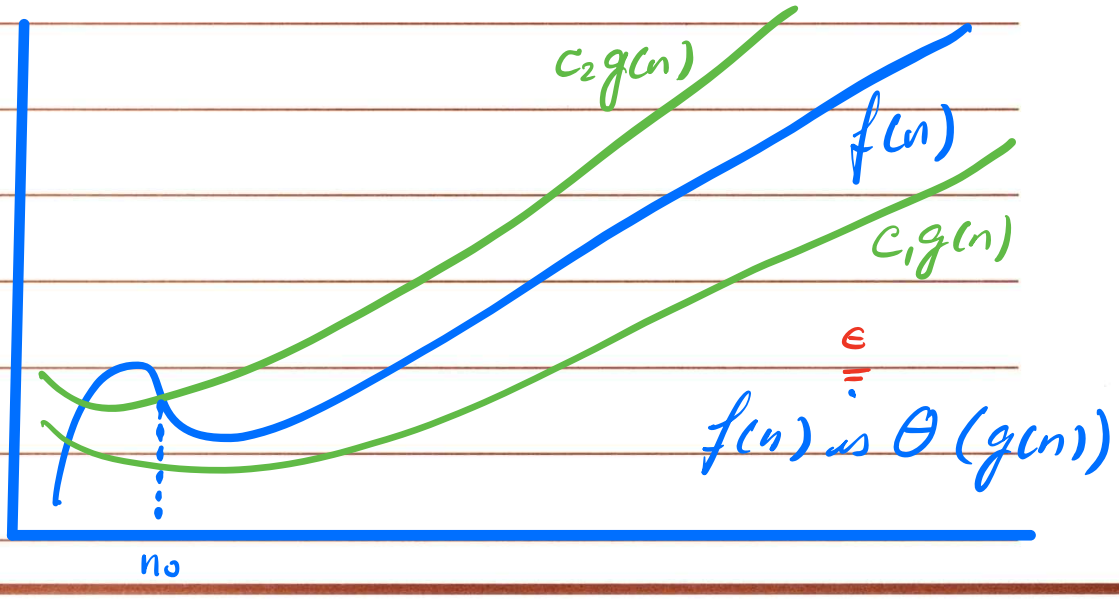
T Every quadratic function (in n) is $\Omega(n^2)$

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. 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 $a_0, a_1, \ldots, a_n$ are real numbers. 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

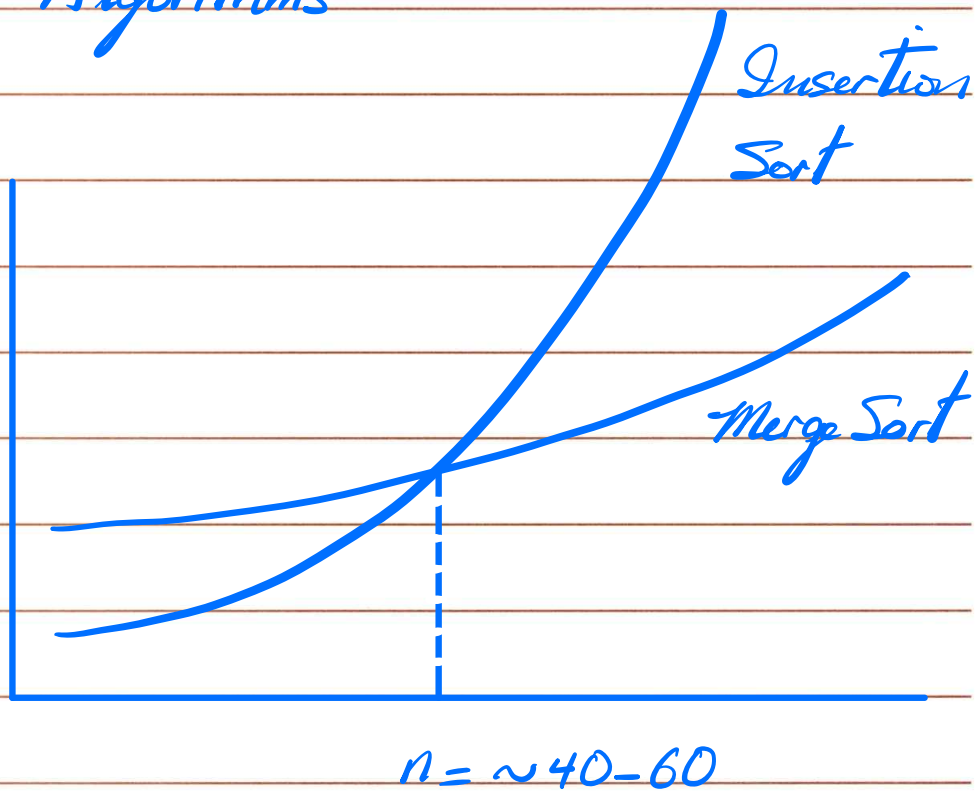
Algorithm A: $O(3^n n^2 \lg^5 n)$

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

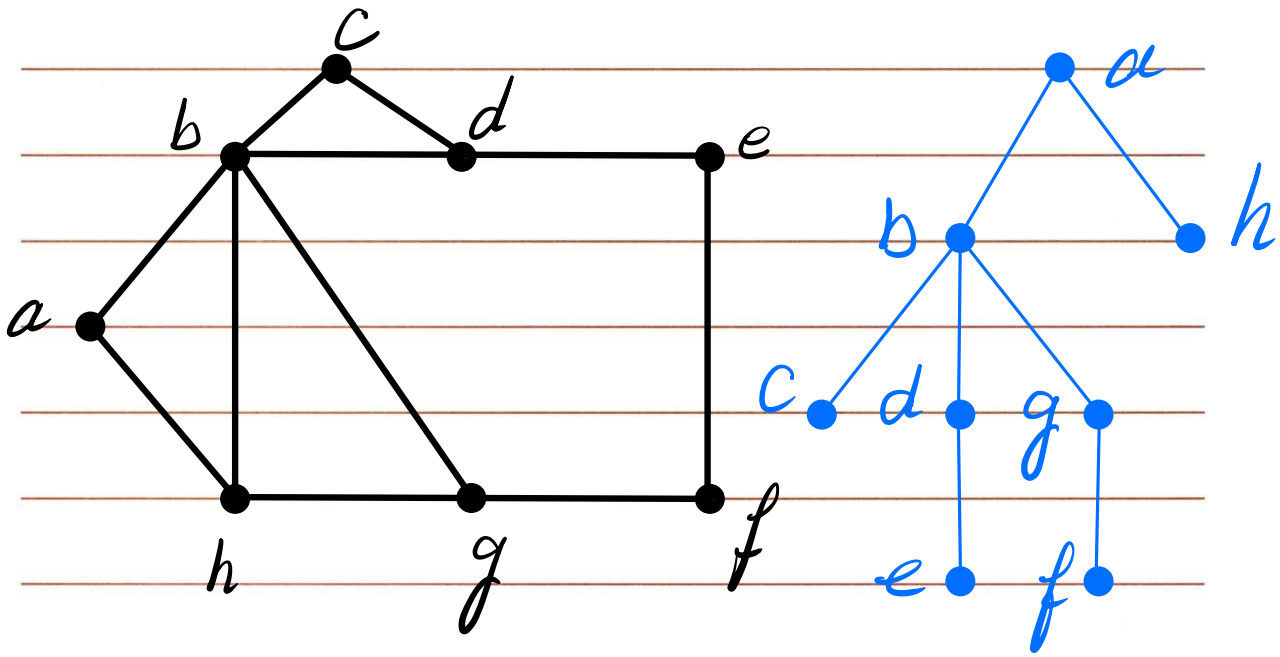
Q: What are we searching for?

- 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
```

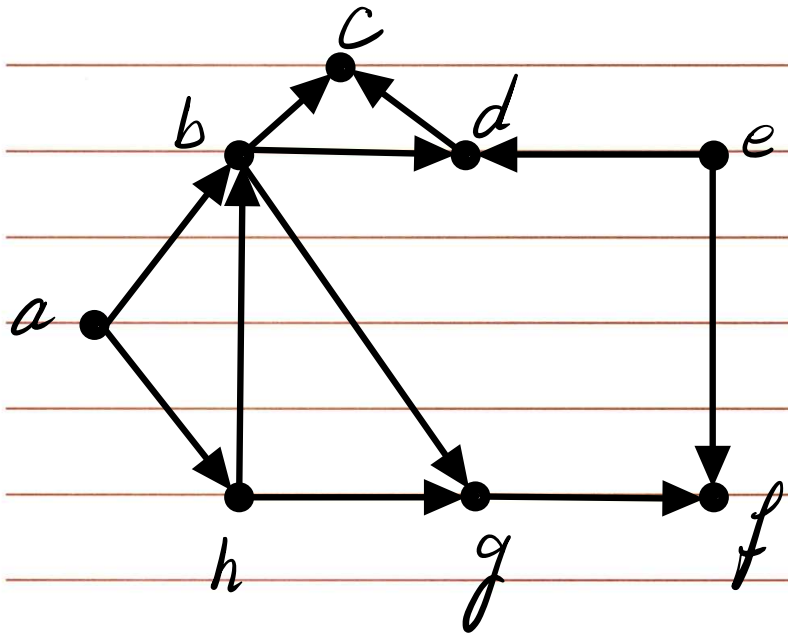
$\sum_{u \in V} d^+(u) = \Theta(m)$

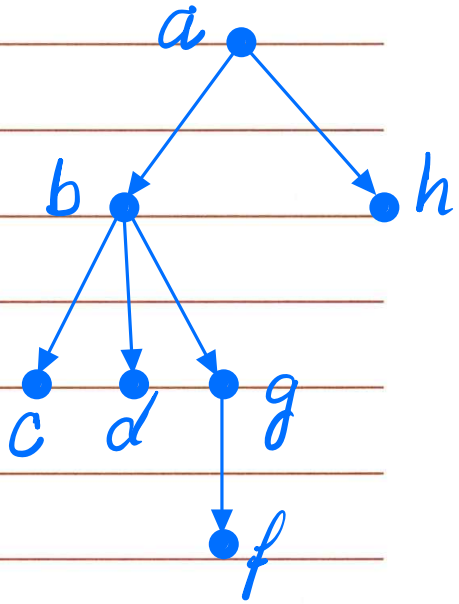
$\Theta(d^+(u))$

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

$\Theta(m+n)$

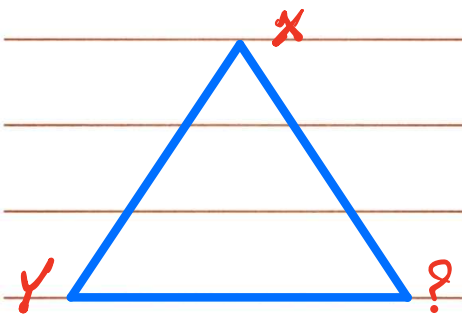
$\Theta(n)$

```
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
```

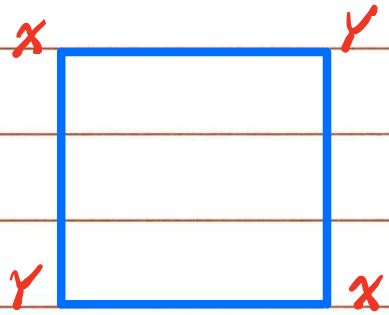
$\Theta(d^+(u))$

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

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

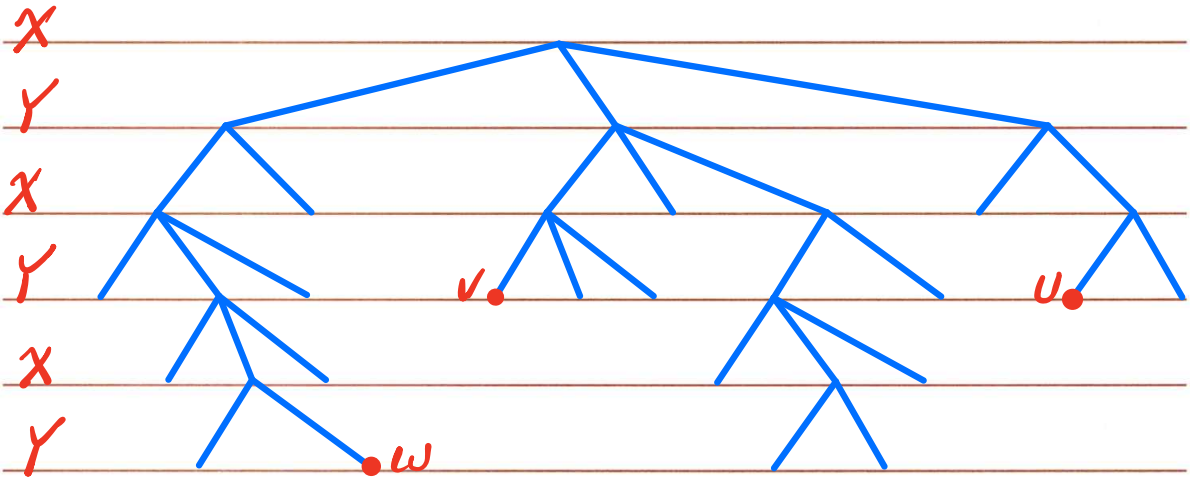


Not bipartite



bipartite

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

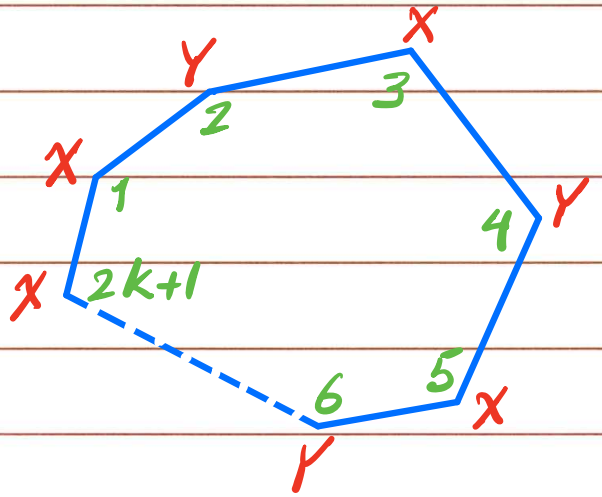


Observations:

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

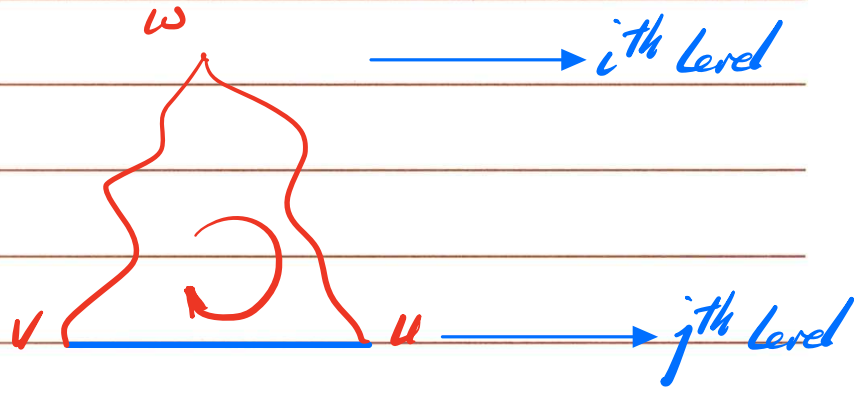
- Edges in G that end up with both ends in the same set (X or Y) must have both ends at the same level in the BFS tree, e.g. between V & u.

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:

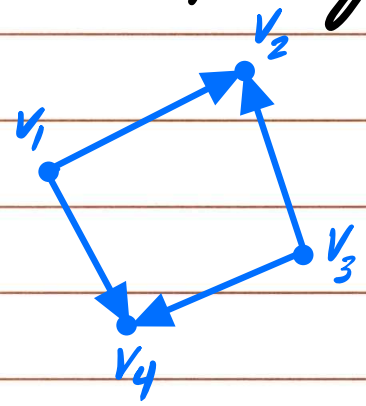
O(m+n) { Run BFS starting from any node, say s. Label each node X or Y depending on whether they appear at an odd or even level on the BFS tree.

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.

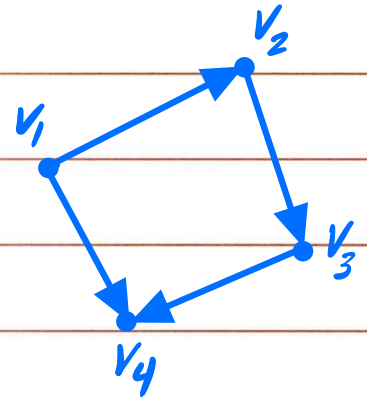
overall complexity = O(m+n)

Connectivity in a directed graph

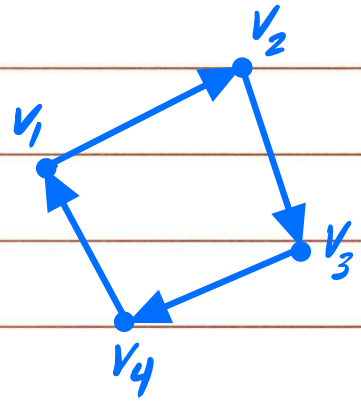
A directed graph is weakly connected if there is a path from any point to any other point in the graph by ignoring edge directions.



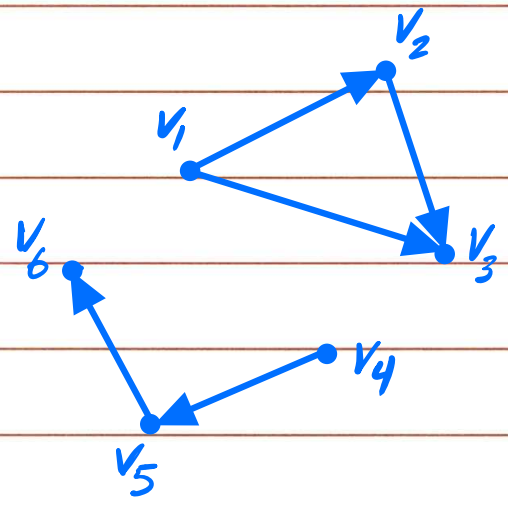
A directed graph is connected if, for any pair of nodes (v, u), there is either a path from v to u, or a path from u to v.



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.

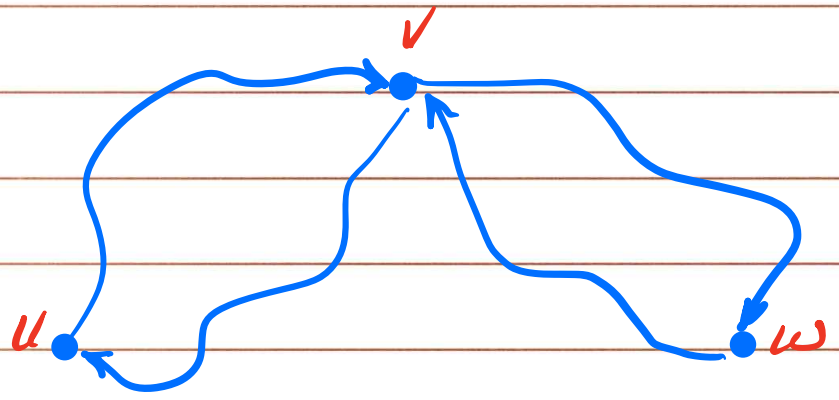


Q: How can you determine if a given graph G is strongly connected?

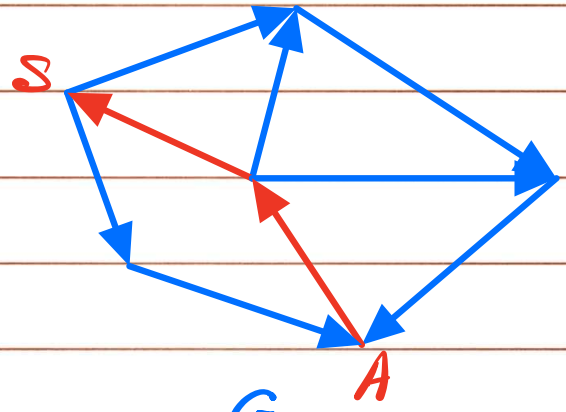
Brute force approach:
Run BFS or DFS from every node.
If in each search, all nodes are found then G is strongly connected

Takes $O(n^2+nm)$

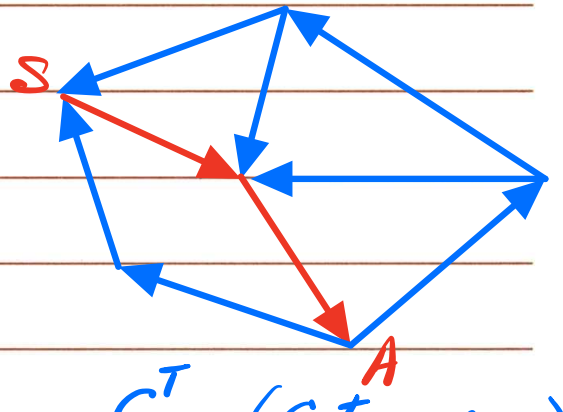
Can we do better?



- 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

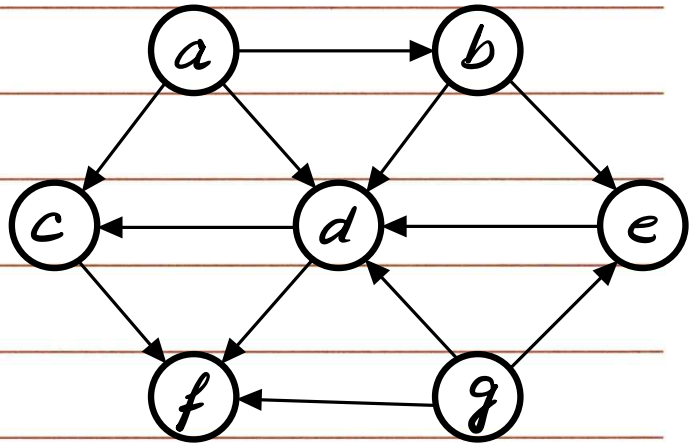
O(m+n) { 3- Use BFS or DFS to find all nodes reachable from s in G^T.
If some nodes are not reachable from s, stop. The graph is not strongly connected.

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)

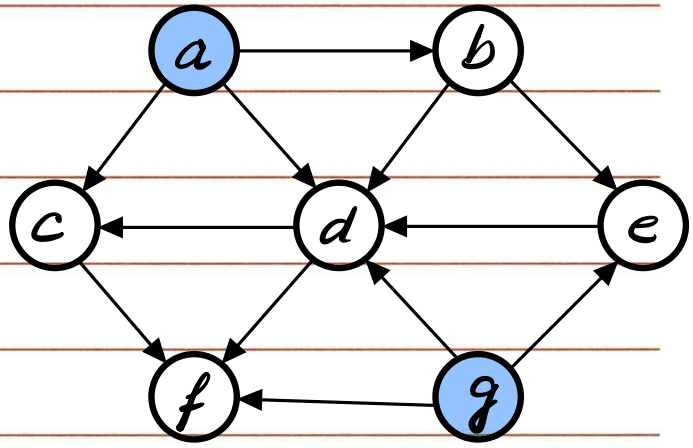
Which node(s) can be first in the topological order?

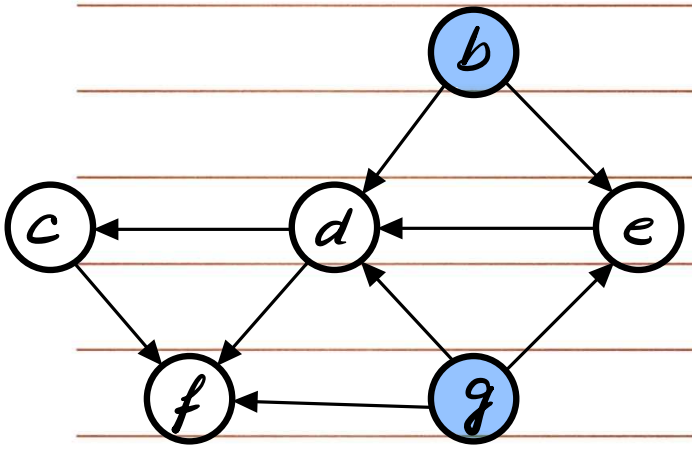


High Level Algorithm

- 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.

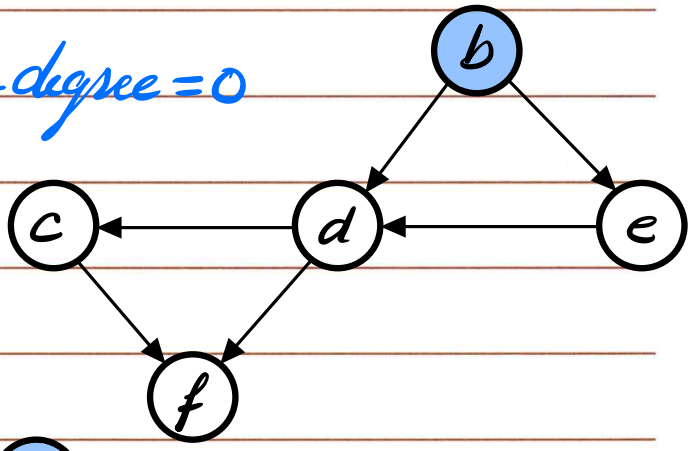
Candidate nodes are a and g.
Remove a.

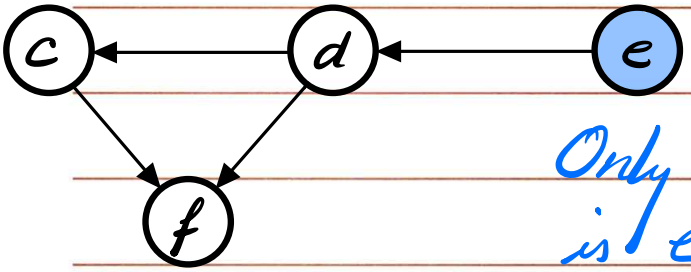




Candidate nodes are b and g.
Remove g.

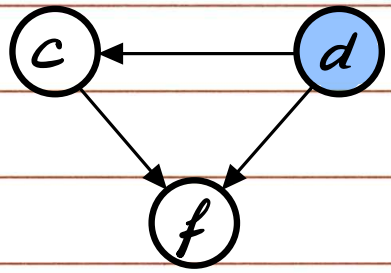
Only node with in-degree=0 is b.
Remove b.

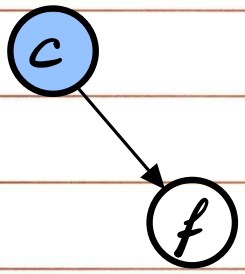




Only node with in-degree=0 is e.
Remove e.

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 = $O(m+n)$

Discussion 2

1. Arrange the following functions in increasing order of growth rate with g(n) following f(n) in your list if and only if *f(n) = O(g(n))*

$$\log n^n,\ n^2,\ n^{\log n},\ n \log \log n,\ 2^{\log n},\ \log^2 n,\ n^{\sqrt{2}}$$

2. Suppose that *f(n)* and *g(n)* are two positive non-decreasing functions such that *f(n) = O(g(n)).* Is it true that $2^{f(n)} = O(2^{g(n)})$?

3. Find an upper bound (Big O) on the worst case run time of the following code segment.

```
void bigOh1(int[] L, int n)
   while (n > 0)
      find_max(L, n); //finds the max in L[0…n-1]
      n = n/4;
```

Carefully examine to see if this is a tight upper bound (Big θ)

4. Find a lower bound (Big Ω) on the best case run time of the following code segment.

```
string bigOh2(int n)
   if(n == 0) return "a";
   string str = bigOh2(n-1);
   return str + str;
```

Carefully examine to see if this is a tight lower bound (Big θ)

5. What Mathematicians often keep track of a statistic called their Erdős Number, after the great 20th century mathematician. Paul Erdős himself has a number of zero. Anyone who wrote a mathematical paper with him has a number of one, anyone who wrote a paper with someone who wrote a paper with him has a number of two, and so forth and so on. Supposing that we have a database of all mathematical papers ever written along with their authors:

a. Explain how to represent this data as a graph.
b. Explain how we would compute the Erdős number for a particular researcher.
c. Explain how we would determine all researchers with Erdős number at most two.

6. In class, we discussed finding the shortest path between two vertices in a graph. Suppose instead we are interested in finding the *longest* simple path in a directed acyclic graph. In particular, I am interested in finding a path (if there is one) that visits all vertices.
Given a DAG, give a linear-time algorithm to determine if there is a simple path that visits all vertices.

1. Arrange the following functions in increasing order of growth rate with g(n) following f(n) in your list if and only if *f(n) = O(g(n))*

$\log n^n$, n^2, $n^{\log n}$, $n \log \log n$, $2^{\log n}$, $\log^2 n$, $n^{\sqrt{2}}$

2. Suppose that *f(n)* and *g(n)* are two positive non-decreasing functions such that *f(n) = O(g(n))*. Is it true that $2^{f(n)} = O(2^{g(n)})$?

3. Find an upper bound (Big O) on the worst case run time of the following code segment.

```
void bigOh1(int[] L, int n)
   while (n > 0)
      find_max(L, n); //finds the max in L[0…n-1]
      n = n/4;
```

Carefully examine to see if this is a tight upper bound (Big θ)

4. Find a lower bound (Big Ω) on the best case run time of the following code segment.

```
string bigOh2(int n)
   if(n == 0) return "a";
   string str = bigOh2(n-1);
   return str + str;
```

Carefully examine to see if this is a tight lower bound (Big θ)

5. What Mathematicians often keep track of a statistic called their Erdős Number, after the great 20th century mathematician. Paul Erdős himself has a number of zero. Anyone who wrote a mathematical paper with him has a number of one, anyone who wrote a paper with someone who wrote a paper with him has a number of two, and so forth and so on. Supposing that we have a database of all mathematical papers ever written along with their authors:

a. Explain how to represent this data as a graph.
b. Explain how we would compute the Erdős number for a particular researcher.
c. Explain how we would determine all researchers with Erdős number at most two.

6. In class, we discussed finding the shortest path between two vertices in a graph. Suppose instead we are interested in finding the *longest* simple path in a directed acyclic graph. In particular, I am interested in finding a path (if there is one) that visits all vertices.
Given a DAG, give a linear-time algorithm to determine if there is a simple path that visits all vertices.

Discussion 2

1. Arrange the following functions in increasing order of growth rate with g(n) following f(n) in your list if and only if *f(n) = O(g(n))*

$$\log n^n,\ n^2,\ n^{\log n},\ n \log\log n,\ 2^{\log n},\ \log^2 n,\ n^{\sqrt{2}}$$

Solution: First separate functions into logarithmic, polynomial, and exponential
Logarithmic: $\log^2 n$
Exponential: $n^{\log n}$
Polynomial: everything else

So we just need to order the ones that are polynomial now:
Since $2^{\log n} = n$ (assuming log base 2), and $\log n^n = n \log n$, we will have the following order:

$$n\ ,\ n \log\log n\ ,\ n \log n\ ,\ n^{\sqrt{2}}\ ,\ n^2$$

Now we put the whole list in this order: Logarithmic, polynomial, exponential which gives us:

$$\log^2 n\ ,\ n\ ,\ n \log\log n\ ,\ n \log n\ ,\ n^{\sqrt{2}}\ ,\ n^2\ ,\ n^{\log n}$$

2. Suppose that *f(n)* and *g(n)* are two positive non-decreasing functions such that *f(n) = O(g(n)).* Is it true that $2^{f(n)} = O(2^{g(n)})$?

Solution: Not true. Will not work for f(n) = 2n and g(n) = n

3. Find an upper bound (Big O) on the worst case run time of the following code segment.

```
void bigOh1(int[] L, int n)
   while (n > 0)
      find_max(L, n); //finds the max in L[0…n-1]
      n = n/4;
```

Carefully examine to see if this is a tight upper bound (Big θ)

Solution: The call to find_max takes linear time with respect to n, and we know that the while loop terminates after log n (base 4) iterations. So, it is easy to say that this code runs in O(n log n). But if you look closer and add up the cost of the calling find_max at every iteration, we will get:

1st call:	cn
2nd call:	cn/4
3rd call:	cn/16
….	

The sum of this series adds up to less than 2cn -- without doing any math. (Since we know that cn + cn/2 + cn/4 + cn/8 + … + c adds up to 2cn and the above sum is less than that)

So, O(n log n) is not a tight upper bound. The tight upper bound is $\theta (n)$

4. Find a lower bound (Big Ω) on the best case run time of the following code segment.

```
string bigOh2(int n)
   if(n == 0) return "a";
   string str = bigOh2(n-1);
   return str + str;
```

Carefully examine to see if this is a tight lower bound (Big θ)

Solution: We quickly see that the recursive call bigOh2 calls itself with a problem size one less than the current size. So, the function will call itself exactly n times. So, we can say that the best case run time is $\Omega (n)$. But if we look more carefully, we will see the string concatenation operation will cost a lot more:

n	output string
----	----------
0	a
1	aa
2	aaaa
3	aaaaaaaa
….	
n	2^n

So, $\Omega (n)$ is not a tight lower bound. In fact, the tight lower bound here is exponential $\theta (2^n)$ ($2+4+8+\ldots+2^n = 2^{n+1} = \theta (2^n)$)

5. What Mathematicians often keep track of a statistic called their Erdős Number, after the great 20th century mathematician. Paul Erdős himself has a number of zero. Anyone who wrote

a mathematical paper with him has a number of one, anyone who wrote a paper with someone who wrote a paper with him has a number of two, and so forth and so on. Supposing that we have a database of all mathematical papers ever written along with their authors:

a. Explain how to represent this data as a graph.
b. Explain how we would compute the Erdős number for a particular researcher.
c. Explain how we would determine all researchers with Erdős number at most two.

Solution:

a) Create a graph where nodes represent mathematicians and edges represent co-authorship.
b) Run BFS in that graph starting from Erdős and see at which level that particular mathematician appears in the BFS tree. That will be his/her Erdős number.
c) All mathematicians within the first 2 levels of the BFS tree (staring from Erdős) have Erdős numbers of at most two.

6. In class, we discussed finding the shortest path between two vertices in a graph. Suppose instead we are interested in finding the *longest* simple path in a directed acyclic graph. In particular, I am interested in finding a path (if there is one) that visits all vertices.
Given a DAG, give a linear-time algorithm to determine if there is a simple path that visits all vertices.

Solution: Find a topological order (See textbook for details on finding topological ordering in DAGs). See if there is a path that goes through the nodes of the graph in that topological order. If there is, then this will be the longest path we are looking for. In fact, if this path exists, it gives us a strict precedence order from the starting node to the end node on the path, and therefore the graph can only have one topological ordering. So, if our topological order does not provide such a path, then we can conclude that such a path does not exist in the graph.