

# Running Times, BFS

CSE 417 Winter 24  
Lecture 5

# Announcements

HW1 is out!

Due in one-week, 11:59 PM on Fri. Jan. 19.

There are two types of problems:

“mechanical” usually: execute an algorithm, come up with an example of something or a very short proof.

“long-form” usually: design an algorithm, write code for an algorithm, think about an algorithm in the real world, or write a longer proof.

The directions include how many of each count. You can submit extras, we count the best ones.

# Announcements

We have grade guarantees in the syllabus. Meeting all requirements in a row guarantees you that grade or higher.

If you're wondering "how am I doing?" this table should help.

Grade	Mechanical Problems	Long-Form Problems
4.0	6E and 2S	22E and 1S
3.5	6E and 1S	18E and 4S
3.0	5E and 1S	14E and 3S
2.5	4E and 1S	8E and 8S
2.0	2E and 2S	6E and 7S
0.7	none	10S, with at least one on five different HWs.

# Announcements

We try to keep the problems in each category approximately the same difficulty, but that isn't always possible. It's a good idea to read all of them.

For coding questions, the gradescope box remains open---you can upload a new piece of code and rerun the tests, but you still need to use a resubmission slot to actually have the grade update on our end.

On each later homework, you can submit up to two old problems to be (re-)graded.

You don't need an initial submission to resubmit.

# Announcements

Collaboration policy (in brief)

PLEASE collaborate!

The types of questions in this course really benefit from bouncing ideas off of others.

But you must submit your own independent writeup

That means waiting 30 minutes between discussing with others and producing your writeup.

And not relying on notes/pictures, etc. from the discussion (more details on the webpage).

If you can't solve it after 30 minutes, then you couldn't solve it if you get a similar problem later.

# Announcements

What about AI (chat-GPT, co-pilot, etc.)?

For coding problems, you may not use AI. You must write all of your own code yourself.

For written problems, you may use AI in limited ways.

You cannot have the AI “do your homework for you” (don’t put the exact problem, or portions of the problem into AI systems).

You may use AI to help you word a response better (e.g., translate an idea or format an explanation more succinctly or more formally). But that’s helping you word your response not generating your response from scratch.

These systems are new. We’re experimenting with allowing them.

Use them wisely. You will need to be able to do algorithms questions without having them “do the problem for you” in the near future.

# Today

What running times are good?

Review some graph terms

BFS and applications

# Running Times

Recall the definition of big-O,  
big-Omega, big-Theta

Big-O is "at most" – it's a fancy version of " $\leq$ "

Big-Omega is "at least" – it's a fancy version of " $>$ "

Big-Theta is "about equal to" – it's a fancy version of " $\approx$ "

## Big-O

$f(n)$  is  $O(g(n))$  if there exist positive constants  $c, n_0$  such that for all  $n \geq n_0$ ,

$$f(n) \leq c \cdot g(n)$$

## Big-Omega

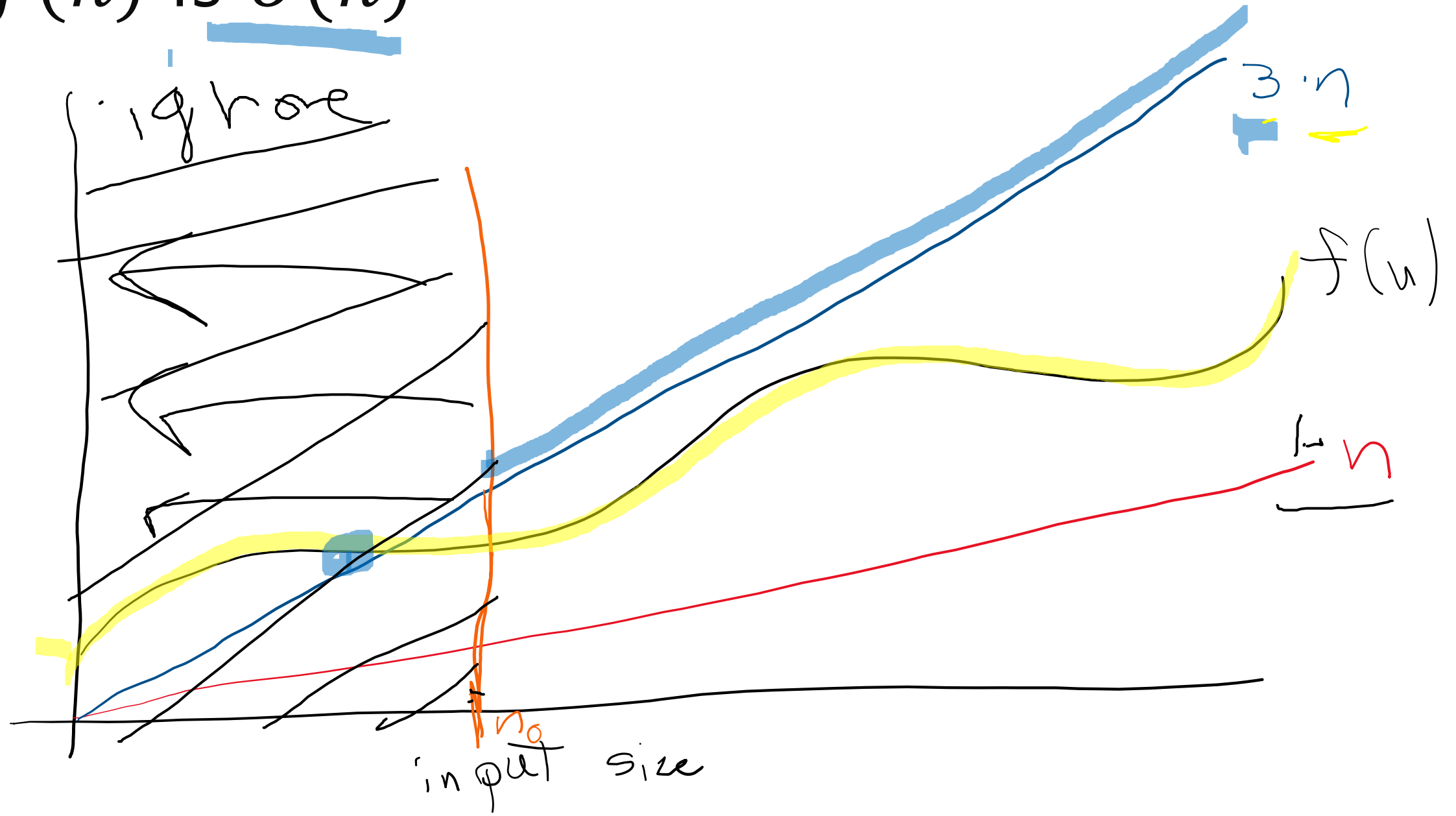
$f(n)$  is  $\Omega(g(n))$  if there exist positive constants  $c, n_0$  such that for all  $n \geq n_0$ ,

$$f(n) \geq c \cdot g(n)$$

## Big-Theta

$f(n)$  is  $\Theta(g(n))$  if  
 $f(n)$  is  $O(g(n))$  and  $f(n)$  is  $\Omega(g(n))$ .

$f(n)$  is  $O(n)$



# What don't we care about?

Ignore lower-order terms.

If there's a  $5n^2$  that's more important than  $10n$  for very large  $n$

Ignore constant factors.

We can't see them clearly with the java code.

Ignore small inputs.

Small enough and it happens in the blink of an eye anyway...

# Big-O isn't perfect!

$f(n) = 1,000,000,000,000,000,000,000,000n$  is slower than  $g(n) = 2n^2$  for "practical" sizes of  $n$ . (but big- $O$ ,  $\Omega$ ,  $\Theta$  says treat  $f$  as faster)

$f(n) = n$  and  $g(n) = 1000000n$  aren't the same for practical purposes. But big- $O$ ,  $\Omega$ ,  $\Theta$  treat them identically.

# Polynomial vs. Exponential

We'll say an algorithm is "efficient" if it runs in polynomial time

## Polynomial Time

We say an algorithm runs in polynomial time if on an input of size  $n$ , the algorithm runs in time  $O(n^c)$  for some constant  $c$ .

Sorting algorithms (e.g. the  $\Theta(n \log n)$  ones) – polynomial time.

Graph algorithms from 373 – polynomial time

# Why Polynomial Time?

Most “in-practice efficient” algorithms are polynomial time, and most polynomial time algorithms **can be made** “in-practice efficient.”

Not all of them! But a good number.

It's an easy definition to state and check.

It's easy to work with (a polynomial time algorithm, run on the output of a polynomial time algorithm is overall a polynomial time algorithm).

e.g. you can find a minimum spanning tree, then sort the edges. The overall running time is polynomial.

It lets us focus on the big-issues.

Thinking carefully about data structures might get us from  $O(n^3)$  to  $O(n^2)$ , or  $O(2^n n)$  to  $O(2^n)$ , but we don't waste time doing the second one.

# Polynomial vs. Exponential

If you have an algorithm that takes exactly  $f(n)$  microseconds, how large of an  $n$  can you handle in the given time?

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$	$2^{10^6}$	$2^{6 \cdot 10^7}$	$2^{36 \cdot 10^8}$	$2^{864 \cdot 10^8}$	$2^{25920 \cdot 10^8}$	$2^{315360 \cdot 10^8}$	$2^{31556736 \cdot 10^8}$
$\sqrt{n}$	$10^{12}$	$36 \cdot 10^{14}$	$1296 \cdot 10^{16}$	$746496 \cdot 10^{16}$	$6718464 \cdot 10^{18}$	$994519296 \cdot 10^{18}$	$995827586973696 \cdot 10^{16}$
$n$	$10^6$	$6 \cdot 10^7$	$36 \cdot 10^8$	$864 \cdot 10^8$	$2592 \cdot 10^9$	$31536 \cdot 10^9$	$31556736 \cdot 10^8$
$n \lg n$	62746	2801417	133378058	2755147513	71870856404	797633893349	68654697441062
$n^2$	1000	7745	60000	293938	1609968	5615692	56175382
$n^3$	100	391	1532	4420	13736	31593	146677
$2^n$	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17



# Polynomial vs. Exponential

For polynomial time, throwing (a lot) more time/compute power can make a significant difference. For exponential (or worse) time, the improvement is minimal.

Running Time	Handle $n = \dots$	Twice as fast processor
$O(n)$	$10^6$	$10^6 \cdot 2$
$O(n^3)$	100	$100 \cdot 2^{1/3} \approx 126$
$O(2^n)$	19	$19 + 1 = 20$

# Polynomial Time isn't perfect.

It has all the problems big-O had.

$f(n) = n^{10000}$  is polynomial-time.  $g(n) = 1.0000000001^n$  is not. You'd rather run a  $g(n)$  time algorithm.

Just like big-O, it's still useful, and we can handle the edge-cases as they arise.

# Tools for running time analysis

Recurrences

Solved with Master Theorem, tree method, or unrolling

Facts from 373

Known running times of data structures from that course—just use those as facts.

We have a reference for you on the [webpage](#)

Style of analysis you did in 373

How many iterations do loops need, and what's the running time of each?

Occasionally, summations to answer those questions.

# Be Careful with hash tables

In-practice hash tables are amazing --  $O(1)$  for every dictionary operation.

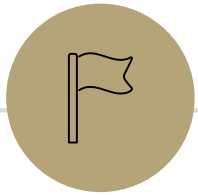
But what about in-theory? In the worst-case  $O(n)$  operations are possible.

Only use a dictionary if you can be sure you'll have  $O(1)$  operations.

Usually the way we accomplish that is by assuming our input comes to us numbered.

E.g. our riders and horses were numbered  $0$  to  $n - 1$ .

And for graphs are vertices are numbered  $0$  to  $n - 1$ .



# Graphs

---

# Graphs

Represent data points and the relationships between them.

That's vague.

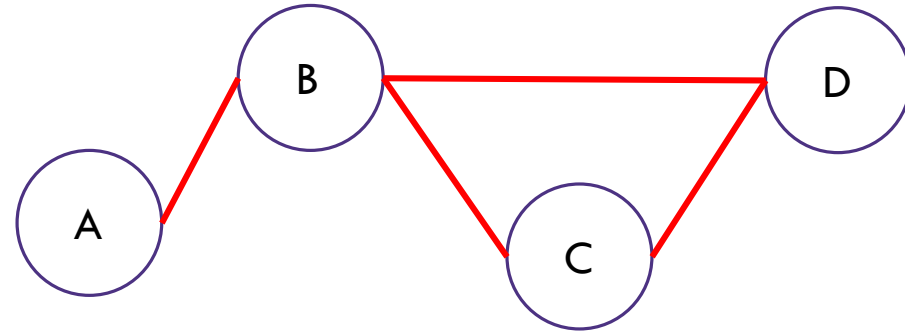
Formally:

A graph is a pair:  $G = (V, E)$

$V$ : set of vertices (aka nodes)  $\{A, B, C, D\}$

$E$ : set of edges  $\{(A, B), (B, C), (B, D), (C, D)\}$

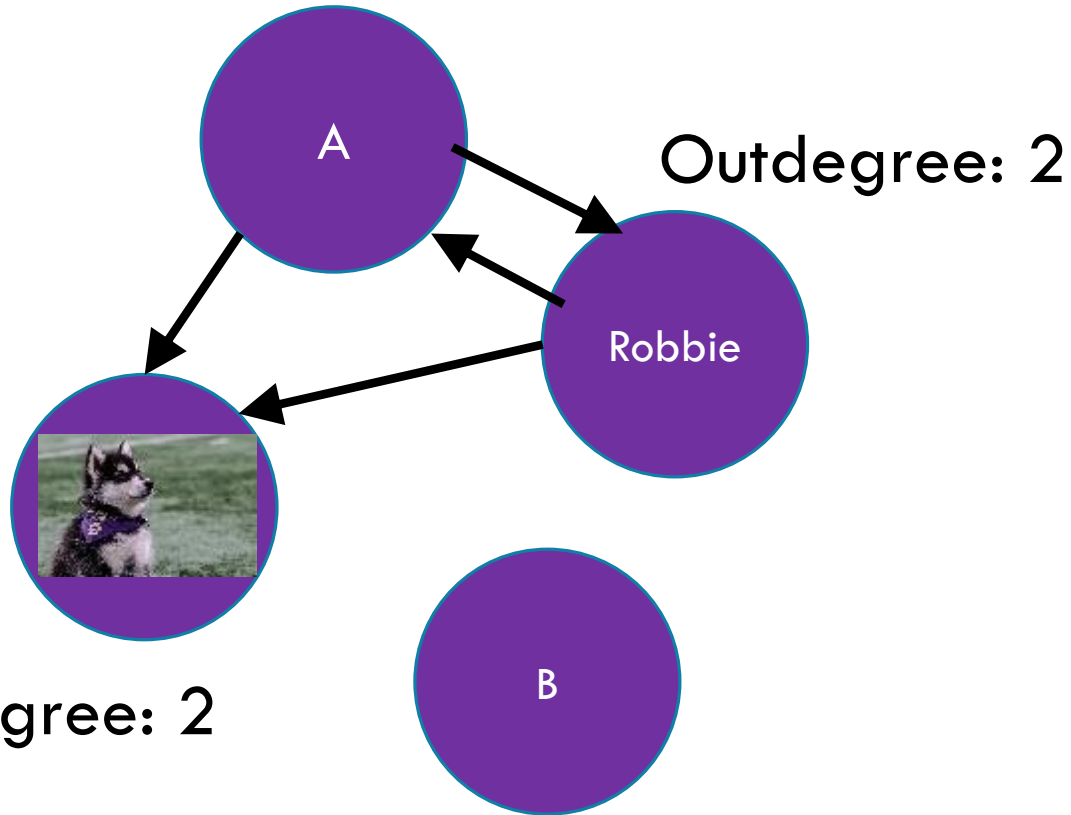
Each edge is a pair of vertices.



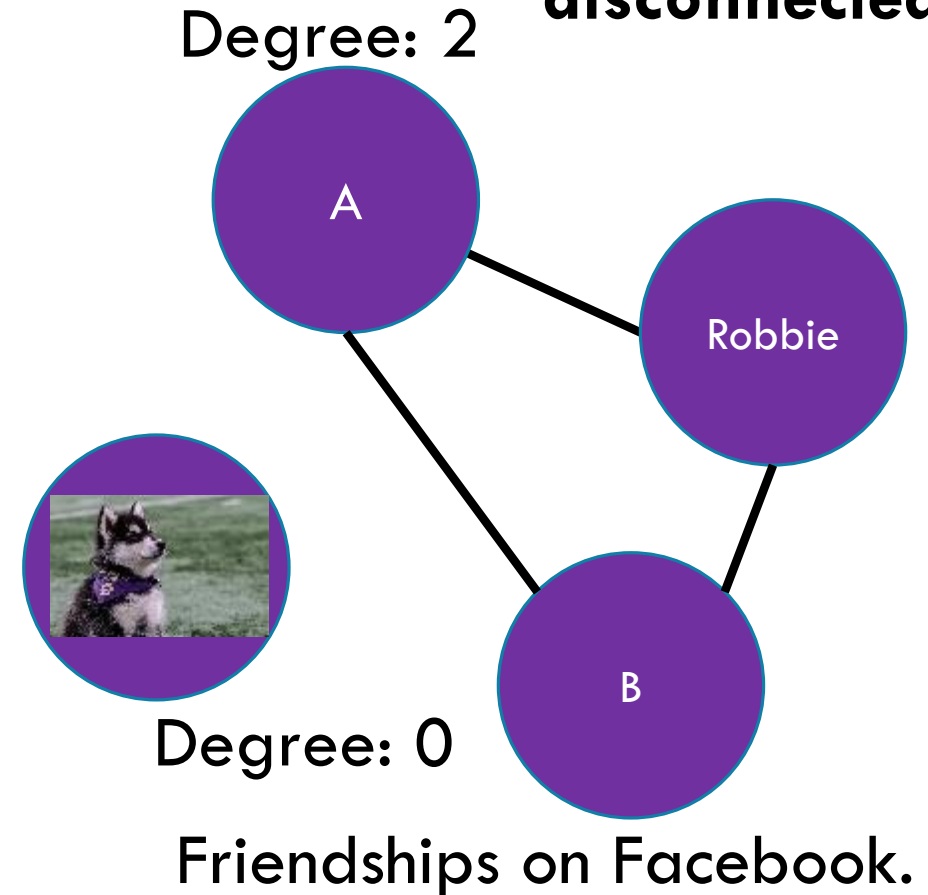
# Graph Terms

Graphs can be directed or undirected.

Following on twitter.



This graph is **disconnected**.



# Making Graphs

If your problem has **data** and **relationships**, you might want to represent it as a graph

How do you choose a representation?

Usually:

Think about what your “fundamental” objects are

Those become your vertices.

Then think about how they’re related

Those become your edges.

# Adjacency Matrix

In an adjacency matrix  $a[u][v]$  is 1 if there is an edge  $(u,v)$ , and 0 otherwise.

Worst-case Time Complexity

( $|V| = n$ ,  $|E| = m$ ):

Add Edge:  $\Theta(1)$

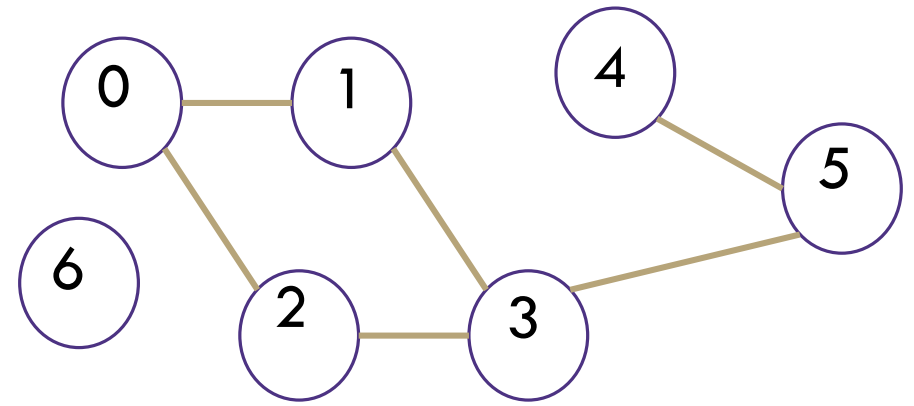
Remove Edge:  $\Theta(1)$

Check edge exists from  $(u,v)$ :  $\Theta(1)$

Get outneighbors of  $u$ :  $\Theta(n)$

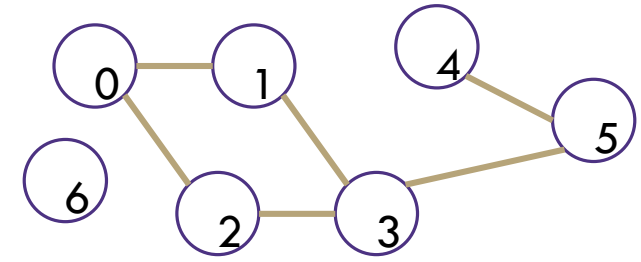
Get inneighbors of  $u$ :  $\Theta(n)$

Space Complexity:  $\Theta(n^2)$



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

# Adjacency List



An array where the  $u^{\text{th}}$  element contains a list of neighbors of  $u$ .

Directed graphs: list of out-neighbors (a[u] has v for all  $(u,v)$  in E)

Time Complexity ( $|V| = n, |E| = m$ ):

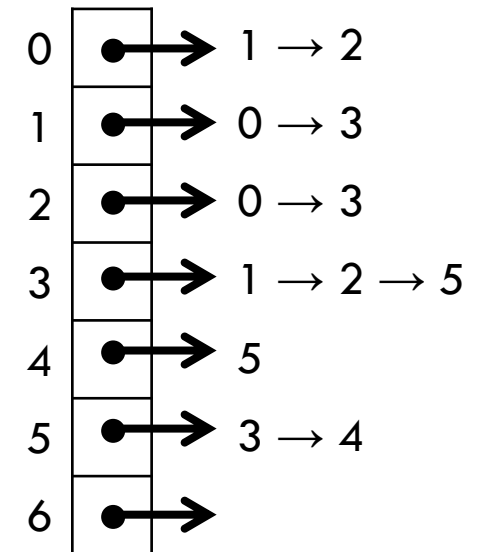
Add Edge:  $\Theta(1)$

Remove Edge:  $\Theta(1)$

Check edge exists from  $(u,v)$ :  $\Theta(1)$

Get neighbors of  $u$  (out):  $\Theta(\text{deg}(u))$

Get neighbors of  $u$  (in):  $\Theta(n)$



Assume we have hash tables AND linked lists

Space Complexity:  $\Theta(n + m)$

# Tradeoffs

Adjacency Matrices take more space, and have slower  $\Theta()$  bounds, why would you use them?

For **dense** graphs (where  $m$  is close to  $n^2$ ), the running times will be close

And the constant factors can be much better for matrices than for lists.

Sometimes the matrix itself is useful ("spectral graph theory")

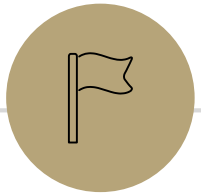
For this class, unless we say otherwise, we'll assume we're using Adjacency Lists and the following operations are all  $\Theta(1)$

Checking if an edge exists.

Getting the next edge leaving  $u$  (when iterating over them all)

"following" an edge (getting access to the other vertex)

To make this work, we usually assume the vertices are numbered.

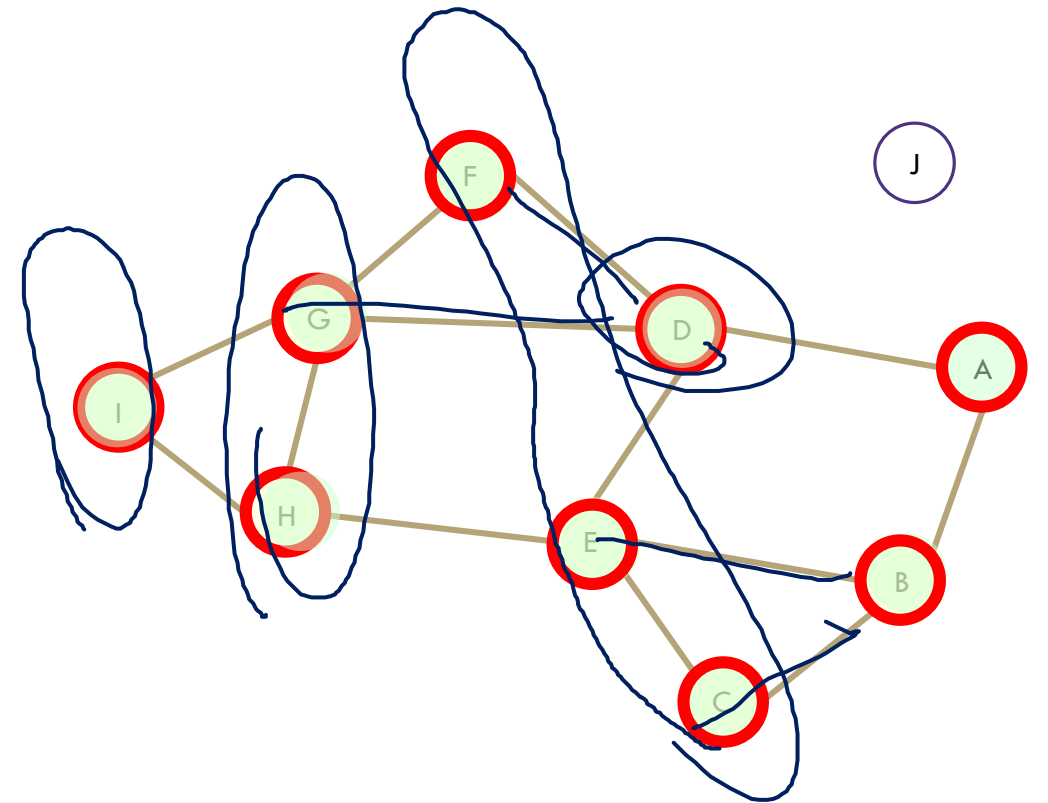


# Traversals



# Breadth First Search

```
search(graph)
  toVisit.enqueue(first vertex)
  mark first vertex as seen
  while(toVisit is not empty)
    current = toVisit.dequeue()
    for (v : current.neighbors())
      if (v is not seen)
        mark v as seen
        toVisit.enqueue(v)
```



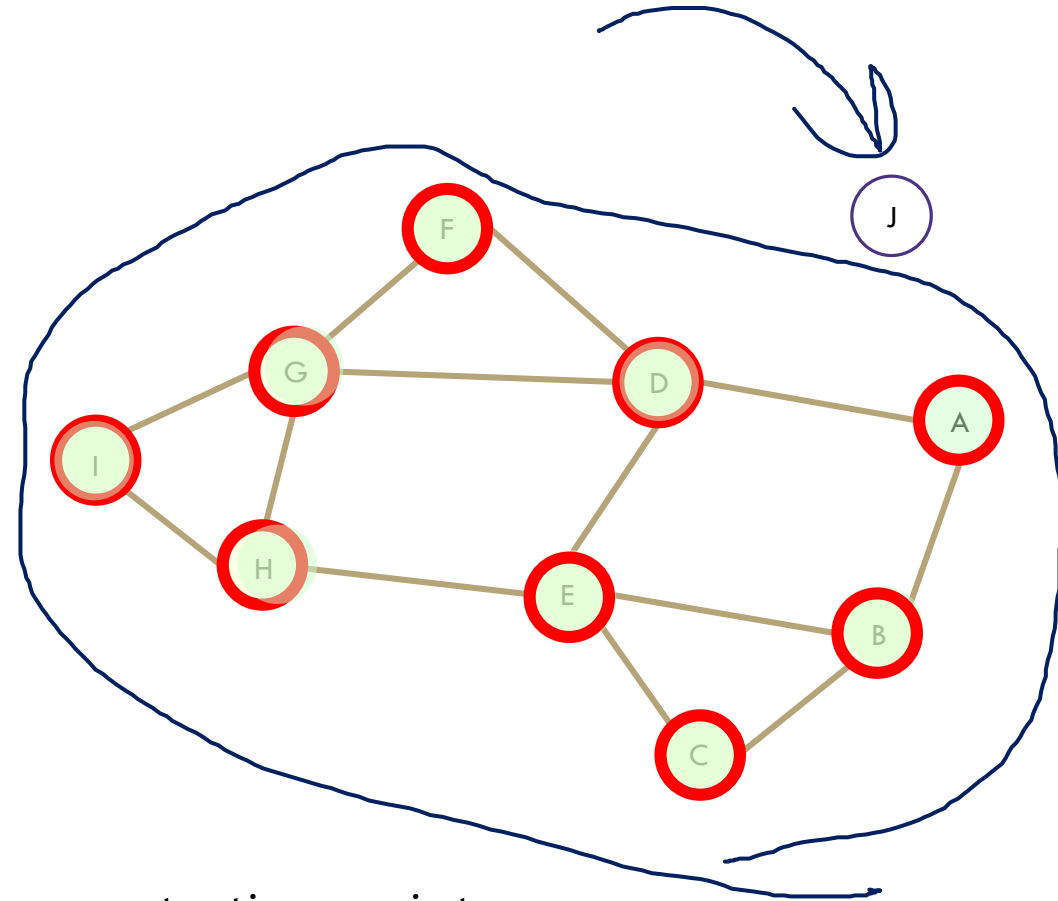
Current node: I ,

Queue: B D E C F G H I

Finished: A B D E C F G H I

# Breadth First Search

```
search(graph)
  toVisit.enqueue(first vertex)
  mark first vertex as seen
  while(toVisit is not empty)
    current = toVisit.dequeue()
    for (v : current.neighbors())
      if (v is not seen)
        mark v as seen
        toVisit.enqueue(v)
```



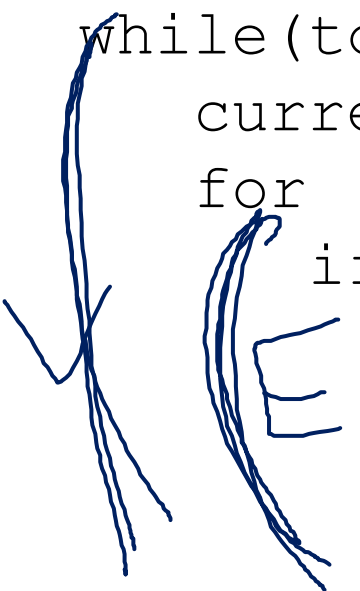
Hey we missed something...

We're only going to find vertices we can "reach" from our starting point.

If you need to visit everything, just start BFS again somewhere you haven't visited until you've found everything.

# Running Time

```
search(graph)
  toVisit.enqueue(first vertex)
  mark first vertex as seen
  while(toVisit is not empty)
    current = toVisit.dequeue()
    for (v : current.neighbors())
      if (v is not seen)
        mark v as seen
        toVisit.enqueue(v)
```



This code might look like:  
a loop that goes around  $m$  times  
Inside a loop that goes around  $n$  times,  
So you might say  $O(mn)$ .

That bound is not tight,  
Don't think about the loops, think about  
what happens overall.  
How many times is `current` changed?  
How many times does an edge get used  
to define `current.neighbors`?

$$O(n+m)$$

We visit each vertex at most twice, and each edge at most once:  $\Theta(|V| + |E|)$



# Old Breadth-First Search Application

Shortest paths in an **unweighted** graph.

Finding the connected components of an **undirected** graph.

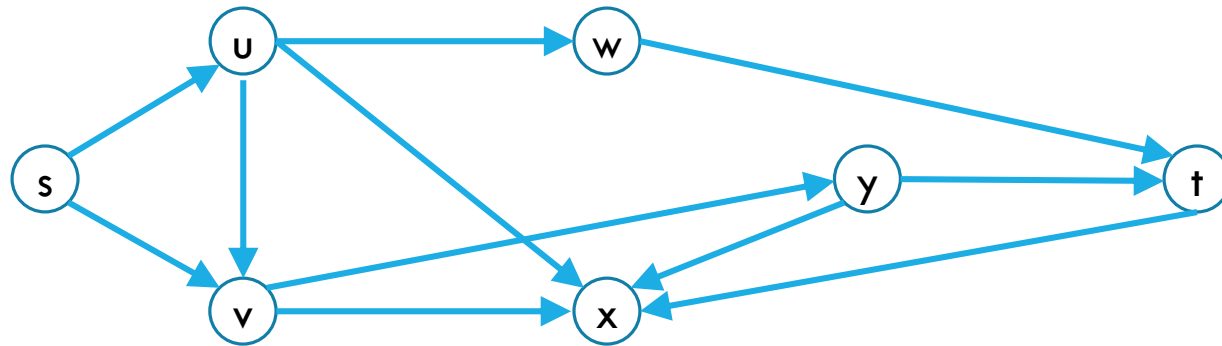
Both run in  $\Theta(m + n)$  time,

where  $m$  is the number of edges (also written  $E$  or  $|E|$ )

And  $n$  is the number of vertices (also written  $V$  or  $|V|$ )

# Unweighted Graphs

If the graph is unweighted, how do we find a shortest paths?



What's the shortest path from s to s?

Well....we're already there.

What's the shortest path from s to u or v?

Just go on the edge from s

From s to w,x, or y?

Can't get there directly from s, if we want a length 2 path, have to go through u or v.

# Not single algorithms: ways to design new algorithms

In 373, we gave you an algorithm and said “implement it, and see an application”

In this class, we usually give you a way to frame your thinking to solve a new problem.

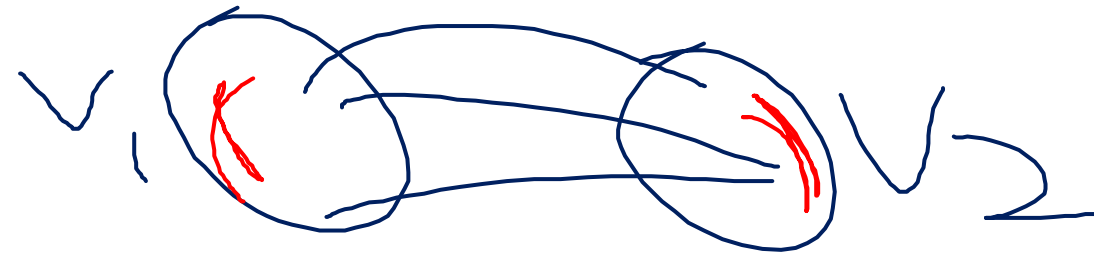
Today and next week: examples of using “graph search” as a way to frame your thinking

You’ll need to understand the specific applications we see.

But the end goal is “create your own new application”

Also pay attention to how we prove things, and how we modify algorithms, and how we saw what changes to make.

# A new application



## Bipartite (also called "2-colorable")

A graph is bipartite (also called 2-colorable) if the vertex set can be divided into two sets  $V_1, V_2$  such that the only edges go between  $V_1$  and  $V_2$ .

Called "2-colorable" because you can "color"  $V_1$  red and  $V_2$  blue, and no edge connects vertices of the same color.

We'll adapt BFS to find if a graph is bipartite

And prove a graph theory result along the way.

# A new application



## Bipartite (also called "2-colorable")

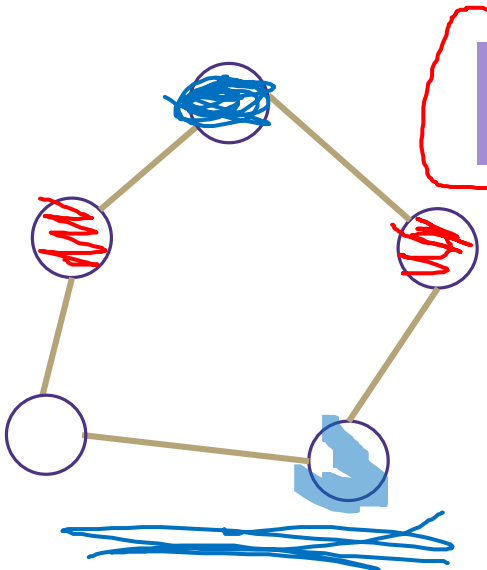
A graph is bipartite (also called 2-colorable) if the vertex set can be divided into two sets  $V_1, V_2$  such that the only edges go between  $V_1$  and  $V_2$ .

Called "2-colorable" because you can "color"  $V_1$  red and  $V_2$  blue, and no edge connects vertices of the same color.

If a graph contains an odd cycle, then it is not bipartite.

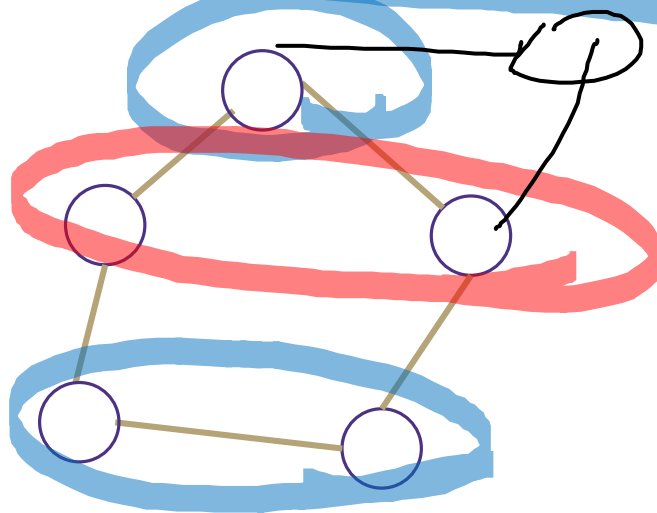
Try the example on the right, then proving the general theorem in the light purple box.

Help Robbie figure out how long to make the explanation  
[Pollev.com/robbie](http://Pollev.com/robbie)



# Lemma 1

If a graph contains an odd cycle, then it is not bipartite.



Start from any vertex, and give it either color.

Its neighbors **must** be the other color.

Their neighbors must be the first color

...

The last two vertices (which are adjacent) must be the same color.

Uh-oh.

# BFS with Layers

Why did BFS find distances in unweighted graphs?

You started from  $u$  ("layer 0")

Then you visited the neighbors of  $u$  ("layer 1")

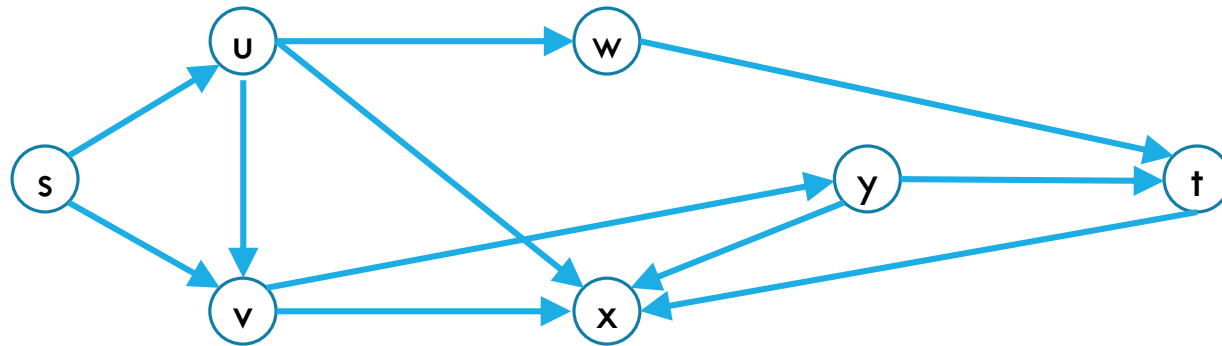
Then the neighbors of the neighbors of  $u$ , that weren't already visited ("layer 2")

...

The neighbors of layer  $i - 1$ , that weren't already visited ("layer  $i$ ")

# Unweighted Graphs

If the graph is unweighted, how do we find a shortest paths?



What's the shortest path from s to s?

Well....we're already there.

What's the shortest path from s to u or v?

Just go on the edge from s

From s to w,x, or y?

Can't get there directly from s, if we want a length 2 path, have to go through u or v.

# BFS With Layers

```
search(graph)
  toVisit.enqueue(first vertex)
  mark first vertex as seen
  toVisit.enqueue(end-of-layer-marker)
  l=1
  while(toVisit is not empty)
    current = toVisit.dequeue()
    if(current == end-of-layer-marker)
      l++
      toVisit.enqueue(end-of-layer-marker)
    current.layer = l
  for (v : current.neighbors())
    if (v is not seen)
      mark v as seen
      toVisit.enqueue(v)
```

It's just BFS!

With some  
extra bells and  
whistles.

# Layers

Can we have an edge that goes from layer  $i$  to layer  $i + 2$  (or lower)?

No! If  $u$  is in layer  $i$ , then we processed its edge while building layer  $i + 1$ , so the neighbor is no lower than layer  $i$ .

Can you have an edge within a layer?

Yes! If  $u$  and  $v$  are neighbors and both have a neighbor in layer  $i$ , they both end up in layer  $i + 1$  (from their other neighbor) before the edge between them can be processed.

# Testing Bipartiteness

How can we use BFS with layers to check if a graph is 2-colorable?

Well, neighbors have to be “the other color”

Where are your neighbors?

Hopefully in the next layer or previous layer...

Color all the odd layers red and even layers blue.

Does this work?

# Lemma 2

If BFS has an intra-layer edge, then the graph has an odd-length cycle.

An “intra-layer” edge is an edge “within” a layer.

Follow the “predecessors” back up, layer by layer.

Eventually we end up with the two vertices having the same predecessor in some level (when you hit layer 1, there’s only one vertex)

Since we had two vertices per layer until we found the common vertex, we have  $2k + 1$  vertices – that’s an odd number!

# Lemma 3

If a graph has no odd-length cycles, then it is bipartite.

Prove it by **contrapositive**

The contrapositive implication is “the same one” prove that instead!

# Lemma 3

If a graph has no odd-length cycles, then it is bipartite.

Prove it by **contrapositive**

The contrapositive implication is “the same one” prove that instead!

We want to show “if a graph is not bipartite, then it has an odd-length cycle.

Suppose  $G$  is not bipartite. Then the coloring attempt by BFS-coloring must fail.

Edges between layers can't cause failure – there must be an intra-level edge causing failure. By Lemma 2, we have an odd cycle.

# The Big Result

## Bipartite (also called "2-colorable")

A graph is bipartite if and only if it has no odd cycles.

Proof:

Lemma 1 says if a graph has an odd cycle, then it's not bipartite (or in contrapositive form, if a graph is bipartite, then it has no odd cycles)

Lemma 3 says if a graph has no odd cycles then it is bipartite.

# The Big Result

The final theorem statement doesn't know about the algorithm – we used the algorithm to prove a graph theory fact!

# Wrapping it up

```
BipartiteCheck(graph) //assumes graph is connected!  
  toVisit.enqueue(first vertex)  
  mark first vertex as seen  
  toVisit.enqueue(end-of-layer-marker)  
  l="odd"  
  while(toVisit is not empty)  
    current = toVisit.dequeue()  
    if(current == end-of-layer-marker)  
      l++  
      toVisit.enqueue(end-of-layer-marker)  
  current.layer = l  
  for (v : current.neighbors())  
    if (v is not seen)  
      mark v as seen  
      toVisit.enqueue(v)  
    else //v is seen  
      if(v.layer == current.layer)  
        return "not bipartite" //intra-level edge  
  return "bipartite" //no intra-level edges
```

# Testing Bipartiteness

Our algorithm should answer "yes" or "no"

"yes  $G$  is bipartite" or "no  $G$  isn't bipartite"

Whenever this happens, you'll have two parts to the proof:

If the right answer is yes, then the algorithm says yes.

If the right answer is no, then the algorithm says no.

OR

If the right answer is yes, then the algorithm says yes.

If the algorithm says yes, then the right answer is yes.

# Proving Algorithm Correct

If the graph is bipartite, then by Lemma 1 there is no odd cycle. So by the contrapositive of lemma 2, we get no intra-level edges when we run BFS, thus the algorithm (correctly) returns the graph is bipartite.

If the algorithm returns that the graph is bipartite, then we have found a bipartition. We cannot have any intra-level edges (since we check every edge in the course of the algorithm). We proved earlier that there are no edges skipping more than one level. So if we assign odd levels to "red" and even levels to "blue" the algorithm has verified that there are no edges between vertices of the same color.