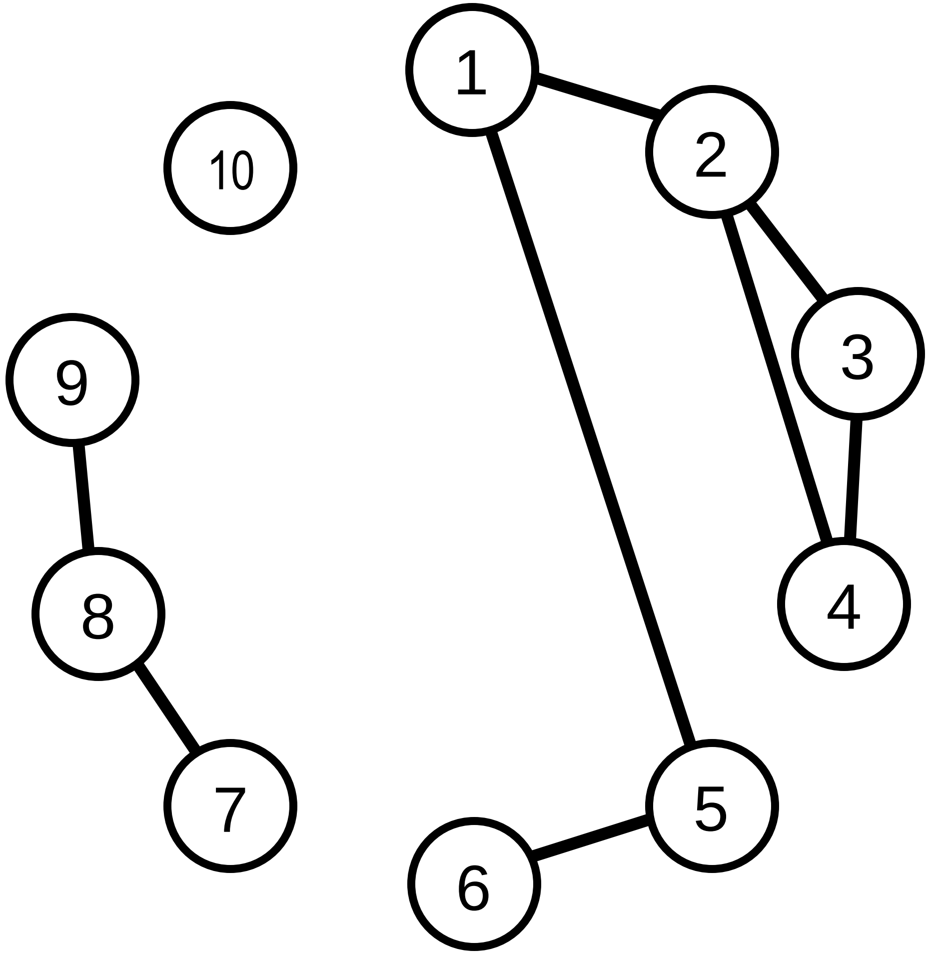
It's Your Birthday! Again



It's Your Birthday! Again

- You want to invite everyone to your party
- You tell a handful of people about your party
- Why not tell everyone? Because people like to gossip – everyone will eventually hear about it, or so you hope

How Gossip Spreads

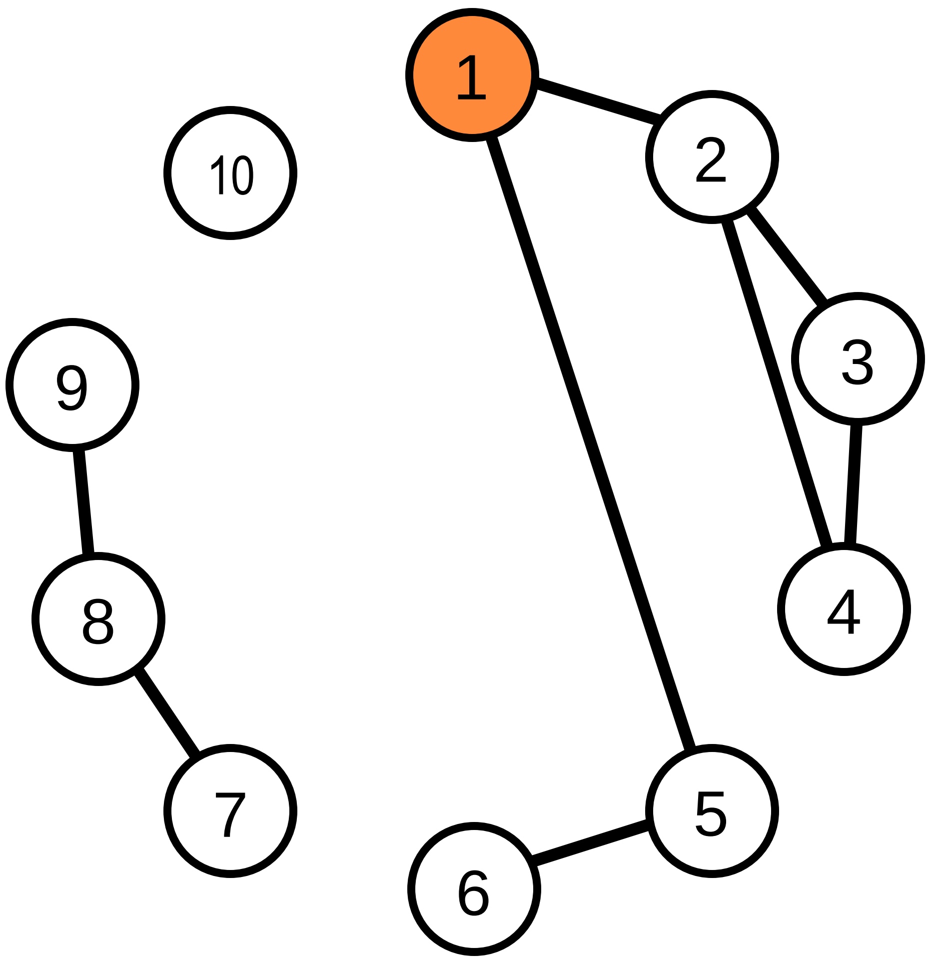
- When someone hears the gossip for the first time, they spread it to all of their friends
- Any of these friends who hear this gossip for the first time will spread the gossip to all of their friends
- Any of their friends who hear the gossip for the first time will then continue spreading the gossip to their friends, and so on
- The gossip continues to spread from friend to friend, until everyone who can hear about it has already heard it

Some Natural Questions to Ask

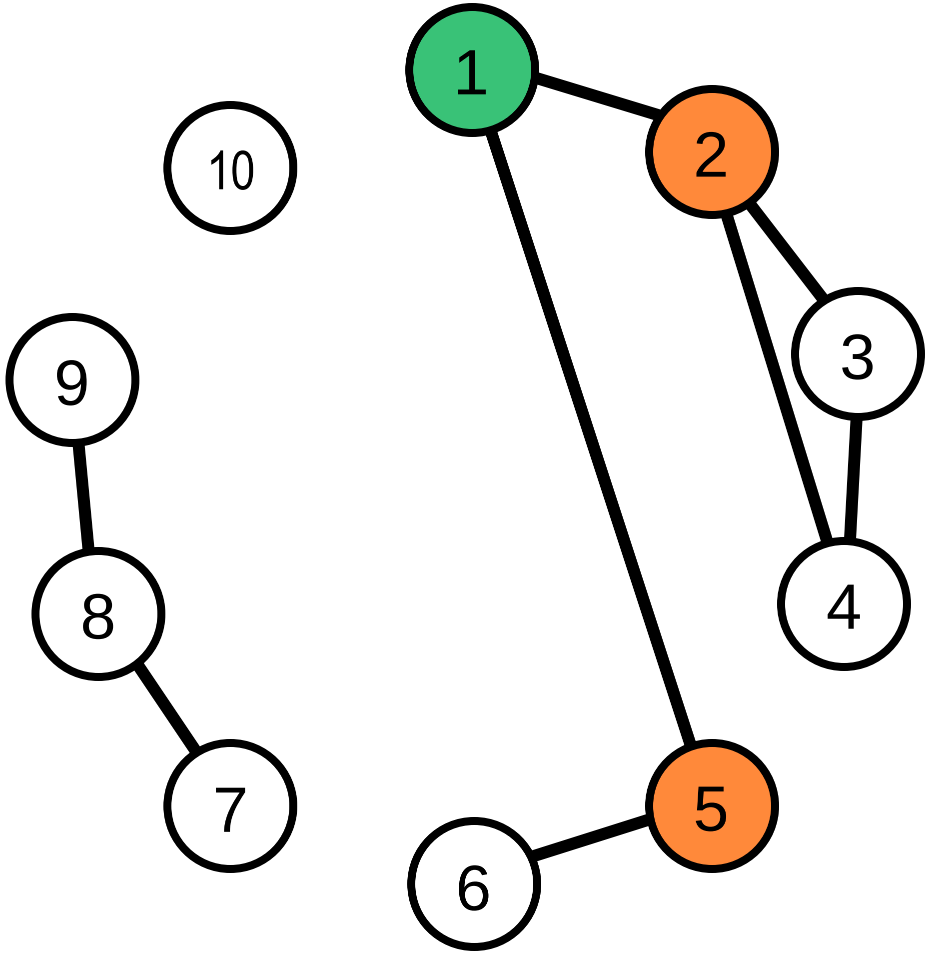
1. If you initially tell the gossip to only one given person and no one else, which people will eventually hear about it?
2. If you want everyone to hear the gossip, what is the minimum number of people you need to tell directly?
3. When is it possible for everyone to hear the gossip by only telling one person directly?

- To answer these questions, let’s study how gossip spreads for a specific case: the network above, directly telling the gossip to only person 1

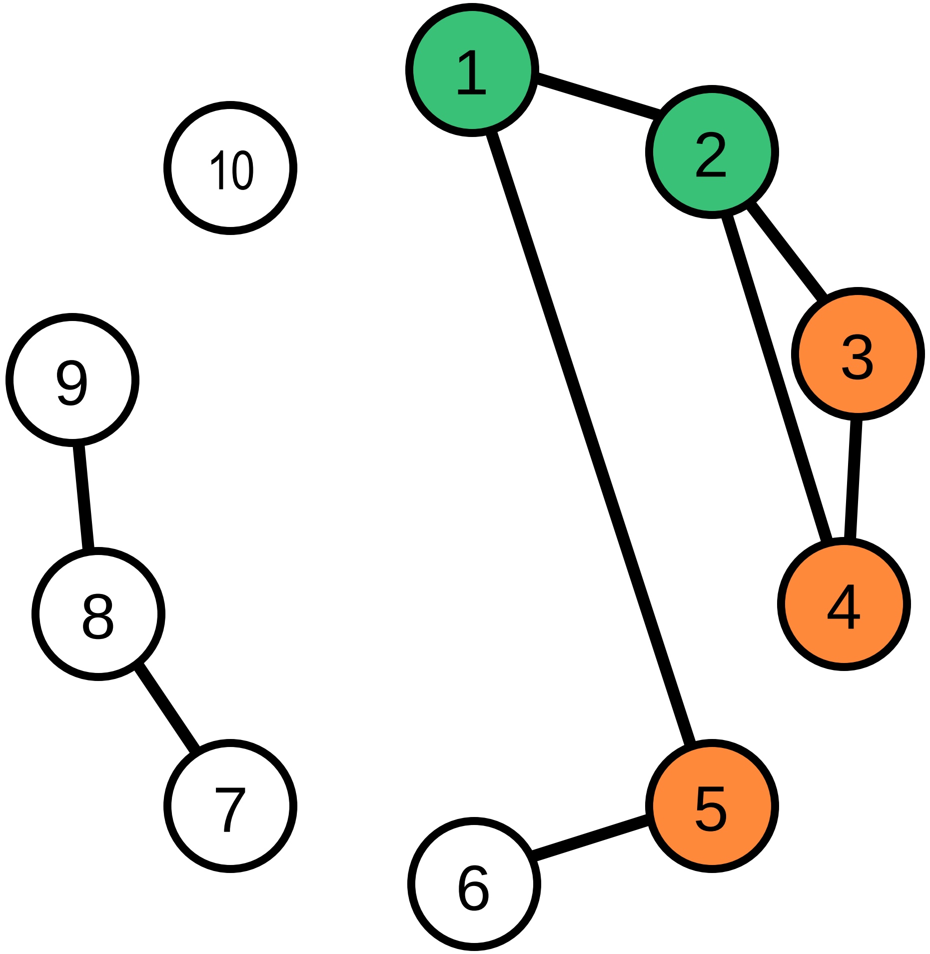
first tell the gossip to 1



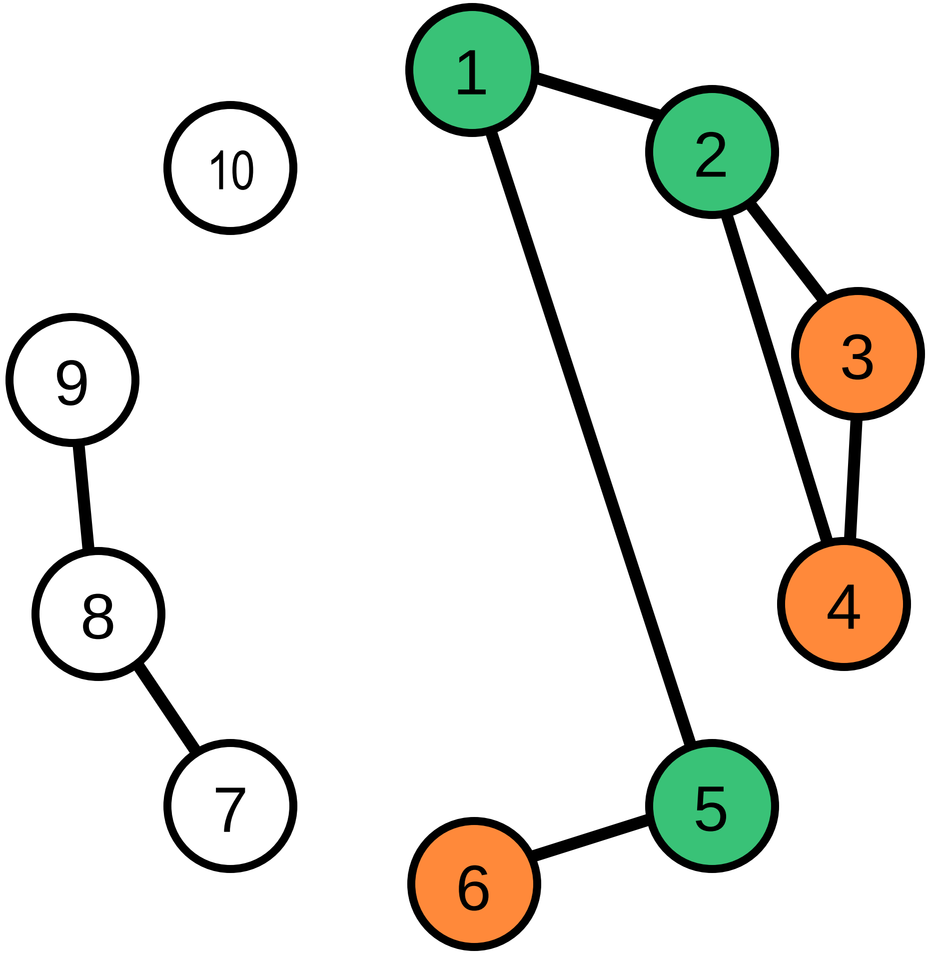
1 spreads gossip to 2 and 5



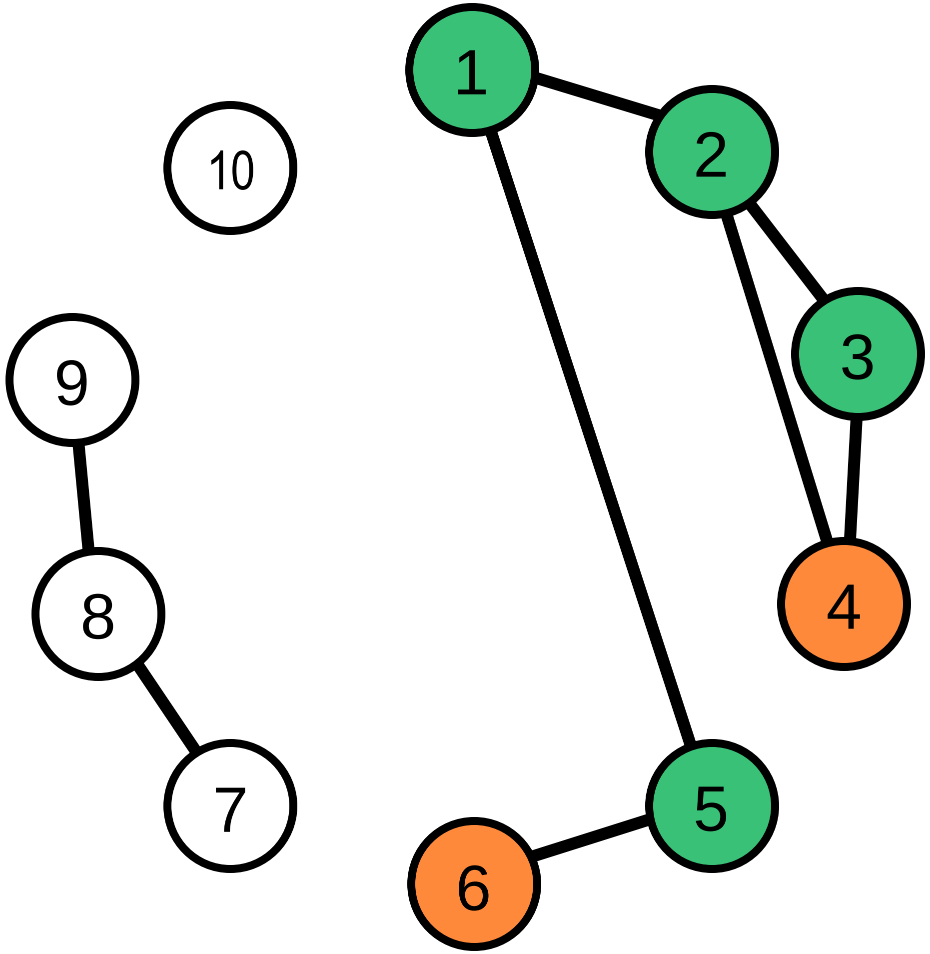
2 spreads gossip to 3 and 4



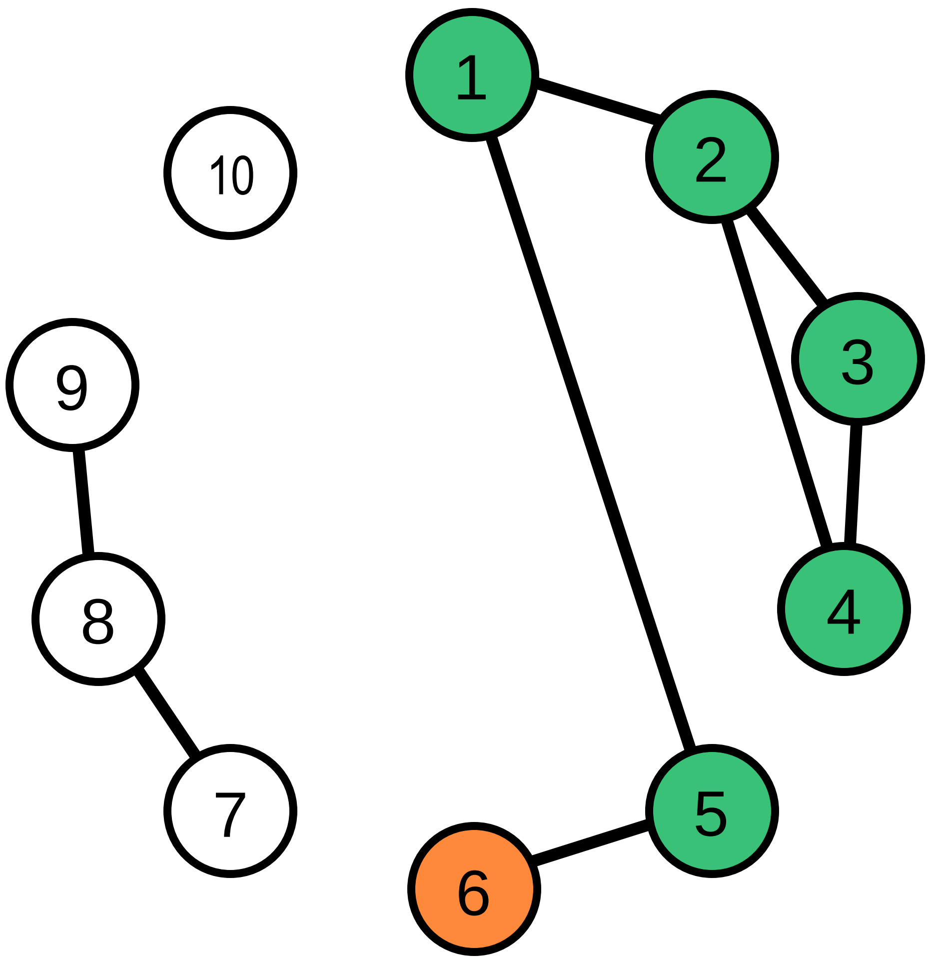
5 spreads gossip to 6



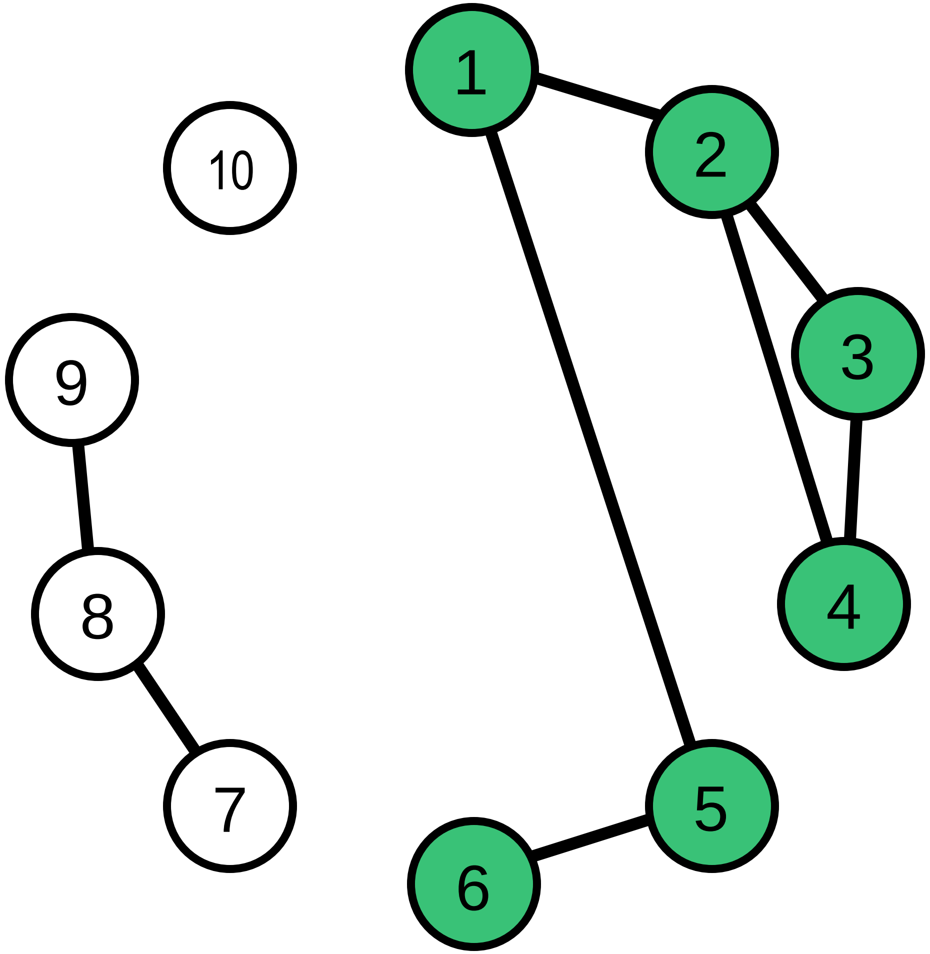
3 spreads gossip to 4
(but 4 just says “already heard”)



4 has no more friends to tell

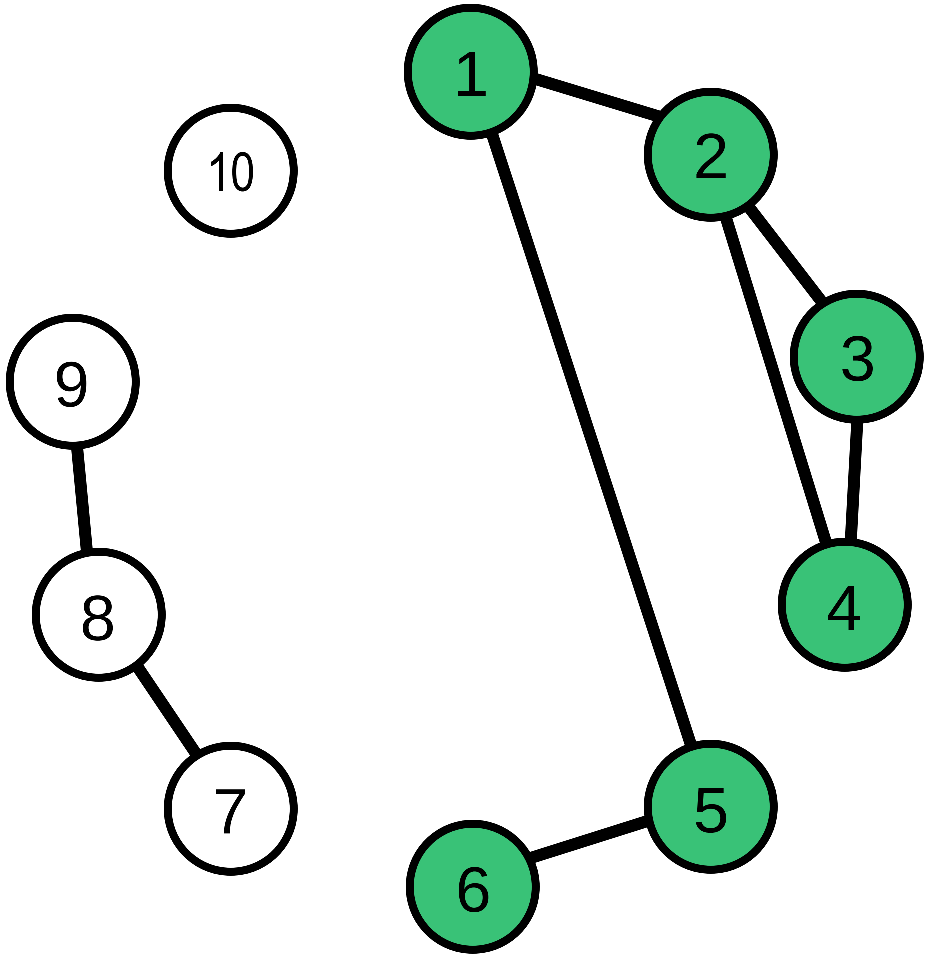


6 has no more friends to tell



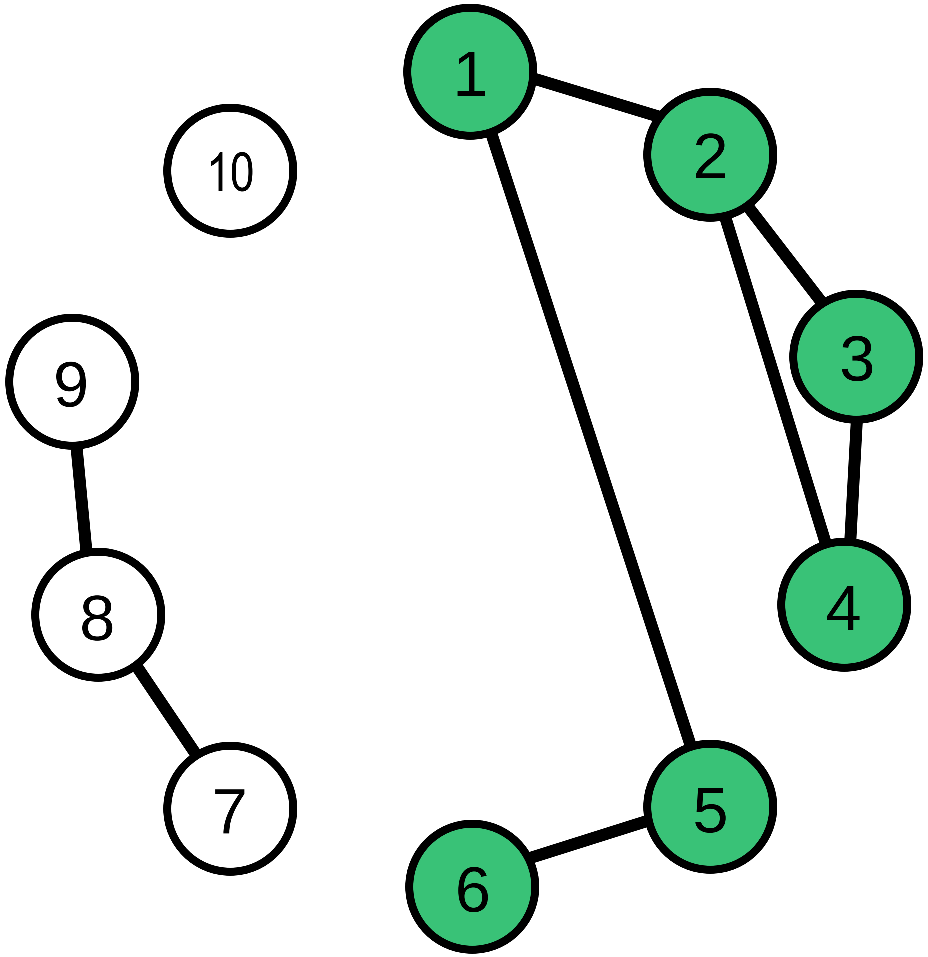
Question

- If we tell the gossip only to person 6 instead, does it change who eventually get to hear the gossip?
- Answer: No



1 and 6 are "indirectly" connected to each other

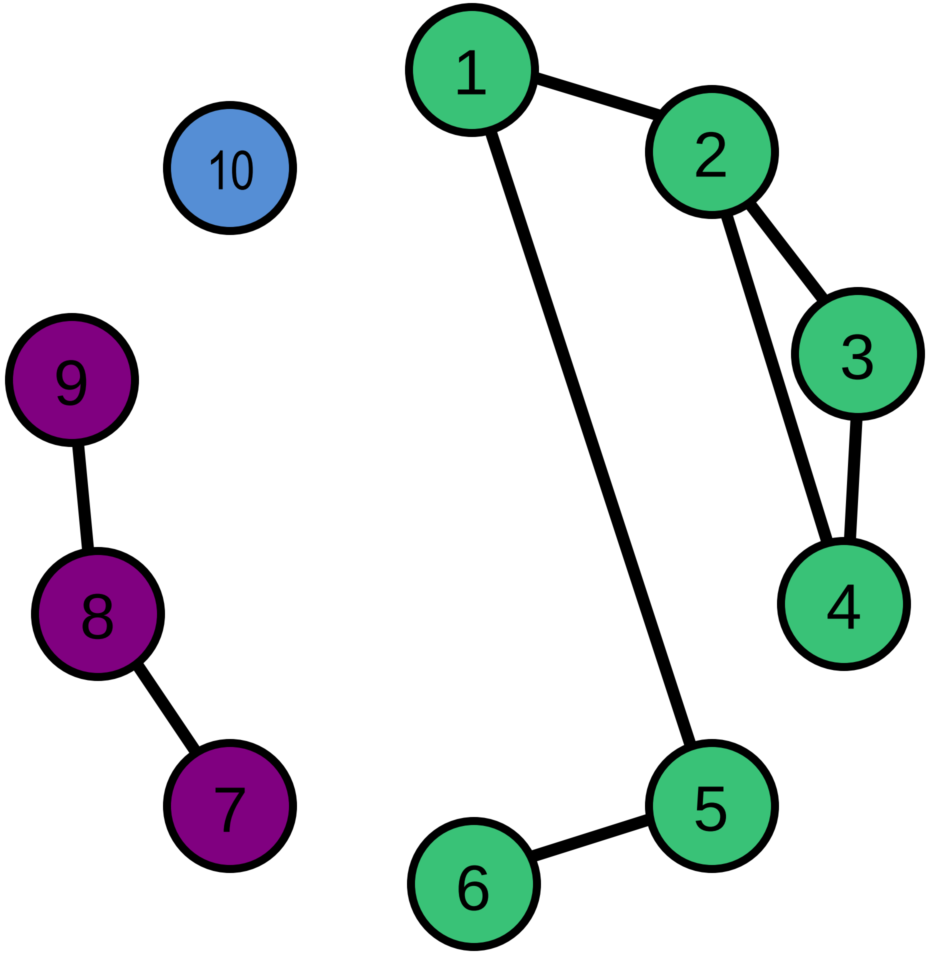
- And so are all the people highlighted, to them and to each other
- If you invite one, the other eventually gets invited, even if they're not directly friends
- Meanwhile some pairs of friends are not connected at all, even indirectly
  - Can you give examples?



Nodes Can be Connected Directly, or Indirectly

- A **connected component** is a set of nodes that are all connected to each other, whether directly or indirectly
  - Shorter word: nodes are *reachable* from each other
- Inviting any one from a connected component results in inviting all of them

Any Network Can Be Broken Down into Its Connected Components



Now We Can Answer

1. If you initially tell the gossip to only one given person and no one else, which people will eventually hear about it?
   - Only those reachable from that person (including the person itself)
   - Only those in the same connected component
2. If you want everyone to hear the gossip, what is the minimum number of people you need to tell directly?
   - This is the same as the number of connected components
3. When is it possible for everyone to hear the gossip by only telling one person directly?
   - Only if the network is **connected** (exactly one component)

But What if 10^5 Nodes? We Don't Want to Answer These Questions by Pen-and-Paper

- It's time to code!
- Let's focus on the first question first and think how to answer the others later
- The "gossip-spreading" algorithm is exactly what we need!
- We just need to translate into code
- In order to code anything, we must first make the algorithm more precise

Spreading Gossip, <u>with Precision</u>

- There is a repetitive process here:
  1. A person hears the gossip for the first time
  2. They tell it to all their friends
     - Each friend who hears it for the first time is going to repeat the same process
- This is already the main bulk of the algorithm, which we can express with a loop

Spreading Gossip, <u>with Precision</u>

```
while someone_hears_for_the_first_time:
    gossiper = person_who_hears_for_the_first_time
    for friend in friends[gossiper]:
        tell_gossip_to(friend)
        if heard_the_first_time_by(friend):
            # friend should repeat somehow
```

Spreading Gossip, with More Precision

- To spread gossip, we need to identify who a person's friends are
- This is precisely the information given to us in the person's **neighbor list**, so we can simply loop through it
- To determine if a friend is hearing the gossip for the first time, we can have a Boolean list `has_heard_gossip`
  - `has_heard_gossip[friend]` is `False` initially for all friends
    - Except for the starting person
  - We determine if the friend is hearing the gossip for the first time by consulting this list *before* telling the gossip
  - Then set to `True` once they hear the gossip
  - Think carefully why we need to read this value before overwriting it

Spreading Gossip, with More Precision

```
has_heard_gossip = [False for _ in range(n)]
has_heard_gossip[starting_person] = True

while someone_hears_for_the_first_time:
    gossiper = person_who_hears_for_the_first_time
    for friend in friends[gossiper]:
        first_time_to_hear = not has_heard_gossip[friend]
        has_heard_gossip[friend] = True
        if first_time_to_hear:
            # friend should repeat somehow
```

Spreading Gossip, with Way More Precision

- One issue: when a person tells the gossip to all their friends, there may be multiple who hear it for the first time and need to repeat the same process "at the same time"
- We (as of now) don't have a way to tell the computer to do several things in parallel

Instead, we will *queue* these people up

- Put all people who must eventually be processed in a list
- The computer will process the people in this list one after the other, instead of at the same time
- For the problems we are solving, this is ok
  - No need to literally spread the gossip in parallel
  - As long as everyone who must eventually hear the gossip does hear
- The way to check if there are more gossipers and to get the next gossiper is through accessing this list
- Initially, the only gossiper is the person we directly told the gossip to, so we can initialize the list with that person

Spreading Gossip, with Way More Precision

```
has_heard_gossip = [False for _ in range(n)]
has_heard_gossip[starting_person] = True
queue = [starting_person]

while len(queue) > 0:
    gossiper = queue.pop(0) # access and remove first
    for friend in friends[gossiper]:
        first_time_to_hear = not has_heard_gossip[friend]
        has_heard_gossip[friend] = True
        if first_time_to_hear:
            queue.append(friend)
```

One Simplification

- Instead of just telling all friends, like you might in real life, then determining if that friend heard it for the first time and hence must spread the word
- First ask the friend if they know about it
- Only if they don't, tell them about it, and then they spread the word

One Simplification
(Think About Why They're Logically Equivalent)

```
first_time_to_hear = not has_heard_gossip[friend]
has_heard_gossip[friend] = True
if first_time_to_hear:
```

↓

```
if not has_heard_gossip[friend]:
    has_heard_gossip[friend] = True
```

The Complete Algorithm (in Python)

```python
has_heard_gossip = [False for _ in range(n)]
has_heard_gossip[starting_person] = True
queue = [starting_person]

while len(queue) > 0:
    gossiper = queue.pop(0)
    for friend in friends[gossiper]:
        if not has_heard_gossip[friend]:
            has_heard_gossip[friend] = True
            queue.append(friend)
```

The Complete Algorithm (in C++) (`friend` is a Keyword and `std::queue` Exists)

```
vector<bool> has_heard_gossip(n); // auto-initializes to false
has_heard_gossip[starting_person] = true;
vector<int> q = {starting_person};

while(q.size() > 0) {
    int gossiper = q[0];
    q.erase(q.begin());
    for(int kaibigan : friends[gossiper]) {
        if(not has_heard_gossip[kaibigan]) {
            has_heard_gossip[kaibigan] = true;
            q.push_back(kaibigan);
        }
    }
}
```

A Detail You Might Be Thinking About

- In real life, a person is smart enough to not tell the gossip back to the person he heard it from
- Doing this in the algorithm requires keeping track for each person, who they heard it from, making the code more complicated
- We don't need to model real life exactly – what matters is that our algorithms are correct and efficient
- Filtering the friends list to remove person-heard-from doesn't make things more efficient for the computer

Now We Can Answer

1. If you initially tell the gossip to only one given person and no one else, which people will eventually hear about it?
   - Those who have `has_heard_gossip` set to `True`

Now We Can Answer

2. If you want everyone to hear the gossip, what is the minimum number of people you need to tell directly?
   - If we start at some random person, and everyone's `has_heard_gossip` set to `True`, then the answer is 1
     - Notice it doesn't matter who the starting person is

Now We Can Answer

2. If you want everyone to hear the gossip, what is the minimum number of people you need to tell directly?
   - Otherwise, we need to pick another person whose `has_heard_gossip` is `False`, start spreading the gossip from that person
     - If everyone's `has_heard_gossip` becomes `True` after this, then the answer is 2
     - Again, notice it doesn't matter who the next starting person is, as long as they haven't heard the gossip before

Now We Can Answer

2. If you want everyone to hear the gossip, what is the minimum number of people you need to tell directly?
   - There might still another person who hasn't heard the gossip, so we keep repeating this process
     - Every time we need to pick another person, we know they are in a different connected component to the persons we've picked before
     - Every person picked corresponds to one new connected component
     - Since we need to spread the gossip starting from multiple people, it'll be convenient to wrap our code earlier in a function
       - We don't know who those people are in advance, so we can't just collect them in the queue at the start and only run the algorithm once

Like This

```
has_heard_gossip = [False for _ in range(n)]

def spread_gossip(starting_person):
    has_heard_gossip[starting_person] = True
    queue = [starting_person]
    while len(queue) > 0:
        gossiper = queue.pop(0)
        for friend in friends[gossiper]:
            if not has_heard_gossip[friend]:
                has_heard_gossip[friend] = True
                queue.append(friend)
```

And Then

```
while someone_has_not_heard_gossip:
    spread_gossip(rando_who_has_not_heard_gossip)
```

Which We Can Translate As

```
def person_who_has_not_heard_gossip():
    for i in range(n):
        if not has_heard_gossip[i]:
            return i

while not all(has_heard_gossip):
    spread_gossip(person_who_has_not_heard_gossip())
```

- Instead of literally picking randomly, just pick the one with smallest label

Now to Answer the Question

```
num_components = 0

while not all(has_heard_gossip):
    spread_gossip(person_who_has_not_heard_gossip())
    num_components += 1

print(num_components)
```

Now We Can Answer

3. When is it possible for everyone to hear the gossip by only telling one person directly?
   - Easy!
   - But can you see how to do it without directly computing `num_components`?

Some Optimizations

- This code is redundant

```
def person_who_has_not_heard_gossip():
    for i in range(n):
        if not has_heard_gossip[i]:
            return i

num_components = 0
while not all(has_heard_gossip):
    spread_gossip(person_who_has_not_heard_gossip())
    num_components += 1
```

Some Optimizations

- Instead, we can simply do

```
num_components = 0
for i in range(n):
    if not has_heard_gossip[i]:
        spread_gossip(i)
        num_components += 1
```

- Take a moment to convince yourself this works
- Saves us from repeatedly going through prefixes of consecutive `True`s in `has_heard_gossip`

Some Optimizations

- Removing from the front of a list takes linear time, because all elements need to be shifted by one index forward
  - Worst case: started with a person who is friends with everyone
- Don't want to do this repeatedly
- Instead, never remove and just keep track of where the first unprocessed person is

Spreading Gossip, with Precision and Speed

```
queue = [starting_person]
index_of_unprocessed = 0
while index_of_unprocessed < len(queue):
    gossiper = queue[index_of_unprocessed]
    index_of_unprocessed += 1
    for friend in friends[gossiper]:
        if not has_heard_gossip[friend]:
            has_heard_gossip[friend] = True
            queue.append(friend)
```

Spreading Gossip, with Precision and Speed (in C++)

```
vector<int> q = {starting_person};
int index_of_unprocessed = 0;
while(index_of_unprocessed < q.size()) {
    int gossiper = q[index_of_unprocessed];
    index_of_unprocessed++;
    for(int kaibigan : friends[gossiper]) {
        if(not has_heard_gossip[kaibigan]) {
            has_heard_gossip[kaibigan] = true;
            q.push_back(kaibigan);
        }
    }
}
```

Spreading Gossip, with Precision and Speed (in C++)

- If you know how to use `std::queue`, that works too and makes the code even clearer

Spreading Gossip, with Precision and Speed (in C++)

```
queue<int> q;
q.push(starting_person);
while(q.size() > 0) {
    int gossiper = q.front();
    q.pop();
    for(int kaibigan : friends[gossiper]) {
        if(not has_heard_gossip[kaibigan]) {
            has_heard_gossip[kaibigan] = true;
            q.push(kaibigan);
        }
    }
}
```

Running Time Analysis

- First, if there is only one component…
- `spread_gossip` processes each node exactly once
  - A node is added to the queue exactly at the moment its `has_heard_gossip` switches from `False` to `True`, and this can happen only once
  - This includes the starting person

Running Time Analysis

- Processing a node means…
  - "Removing" it from the queue
    - Because of our optimization, this is just $\Theta(1)$
  - Going through all its neighbors exactly once
    - This is the real bottleneck
- Each node's neighbor list is looped through once and $\Theta(1)$ work is done on each neighbor
- Work done is directly proportional to the amount of stuff in all the neighbor lists, or $\Theta(L)$ where L is the number of links

Running Time Analysis

- If there are multiple components…
- Same thing happens, except each component is processed separately
- Each component is processed exactly once
- Work done *from looping through the neighbor lists* is $\Theta(L)$
- No matter how many links there are, there is a $\Theta(N)$ loop outside `spread_gossip`
  - If there are zero links, running time must still be at least $\Theta(N)$
- The total is $\Theta(N + L)$

Questions

- Why is the `if not has_heard_gossip[friend]` check important?
- Answer: without it, we'll have an infinite loop of two friends spreading the gossip back and forth between each other forever
- Say we do filter out person-heard-from from the friends list, now can we remove the `if not has_heard_gossip[friend]` check?
- Answer: still no, you can have a loop of three or more friends

Questions to Think About at Home

- Say the network is directed and there are no cycles, can we remove the `if not has_heard_gossip[friend]` check?
  - Cycle: a pair of nodes (u, v) such that $u \neq v$, v is reachable from u, and u is reachable from v
- Why is it important that `has_heard_gossip` is defined globally, rather than inside `spread_gossip`?
- If we did not optimize the queue, what would be the worst-case running time instead?

Questions to Think About at Home

- If there were multiple connected components, and instead of just having one loop, we repeatedly checked `not all(has_heard_gossip)` and called `person_who_has_not_heard_gossip`, what would be the worst-case running time instead?
- For now, note that all these details are important; otherwise, the running time degrades
  - If you get TLE, it means you mis-implemented some detail

Treasure-Finding

- A video game dungeon has a number of rooms
- And a number of two-way corridors that connect different rooms
- There is a special "entrance" room where the game starts
- There is treasure in a different special room

Treasure-Finding

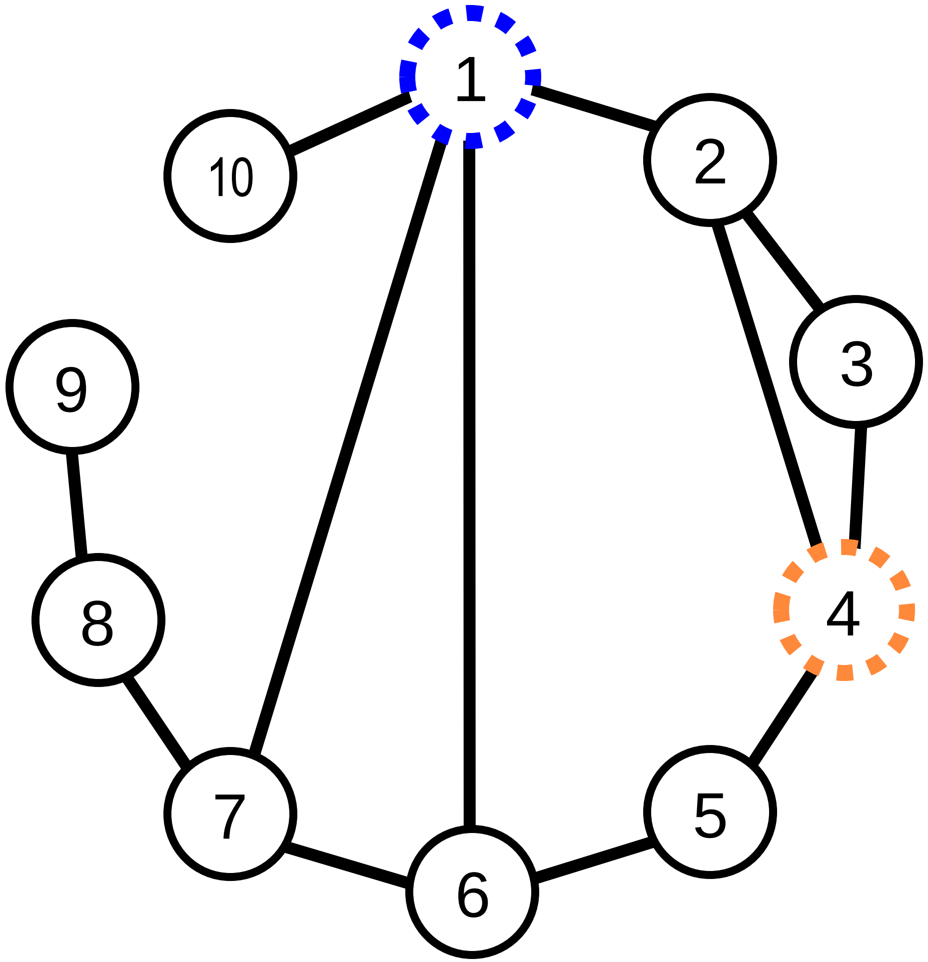
- You want to get the treasure as soon as possible, so you want to find the **shortest path** to the treasure
  - All corridors have the same length: assume it takes the same amount of time to pass through any of them
  - The corridors are long: assume the time spent within a room is negligible
  - We'll define the length of the shortest path to be the number of corridors crossed along the way
- The game has lots of these dungeons, so you want a program that works for any configuration of rooms and corridors

Some Additional Assumptions

- No corridor from a room to itself
- At most one corridor between two rooms
- Can get from one room to any other room by passing through a sequence of corridors

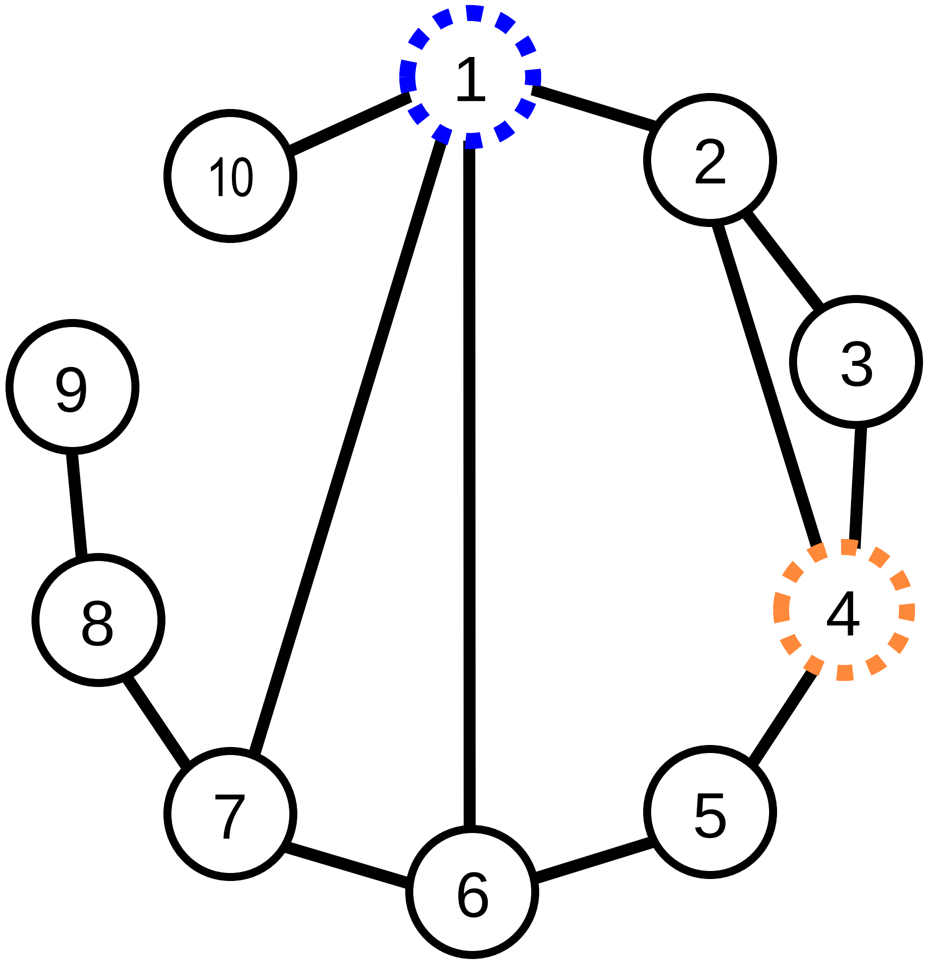
We can model this as a network
Rooms = Nodes, Corridors = Links

- Let's study this specific dungeon
- Suppose the entrance is at room 1, the treasure at room 4
- There are several possible paths
  - $1 \to 2 \to 3 \to 4$
  - $1 \to 2 \to 4$
  - $1 \to 6 \to 5 \to 4$
  - $1 \to 7 \to 6 \to 5 \to 4$
  - $1 \to 10 \to 1 \to 2 \to 4$
  - etc.



We can model this as a network
Rooms = Nodes, Corridors = Links

- We can clearly see that from 1, we must go to 2, but why?
- Among the neighbors 2, 6, 7, and 10, what makes 2 special?
- It's the one "nearest" to 4
- Idea: if we already knew the **distance** of each room from the treasure, to find the shortest path, repeatedly go to the neighbor with smallest distance



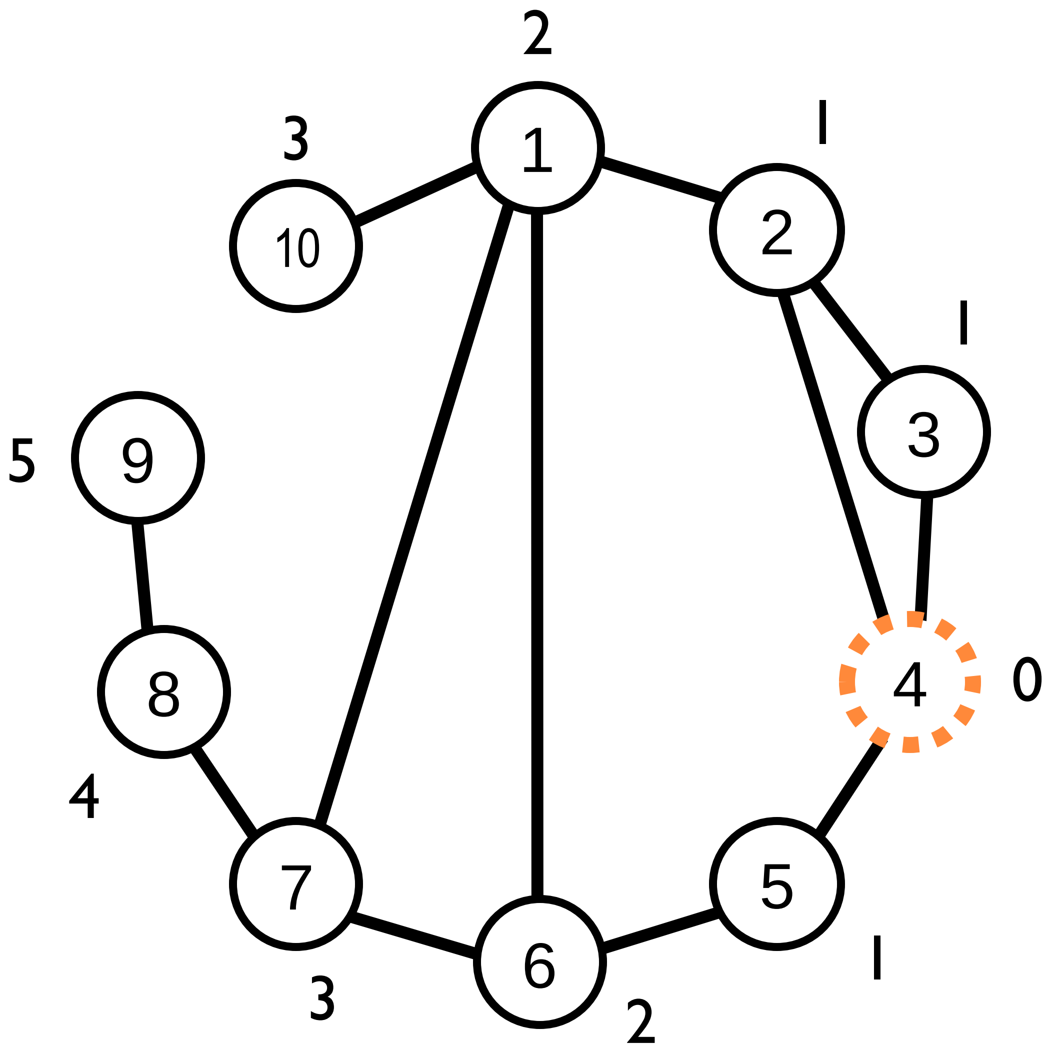
Shortest Path-Finding, Assuming We Knew Distances to Treasure

```
path = [entrance]
while path[-1] != treasure:
    current_room = path[-1]
    best = None
    for neighbor in neighbors[current_room]:
        if best is None or dist[neighbor] < dist[best]:
            best = neighbor
    path.append(best)
```

- Only works because we assume connected; otherwise, need more care

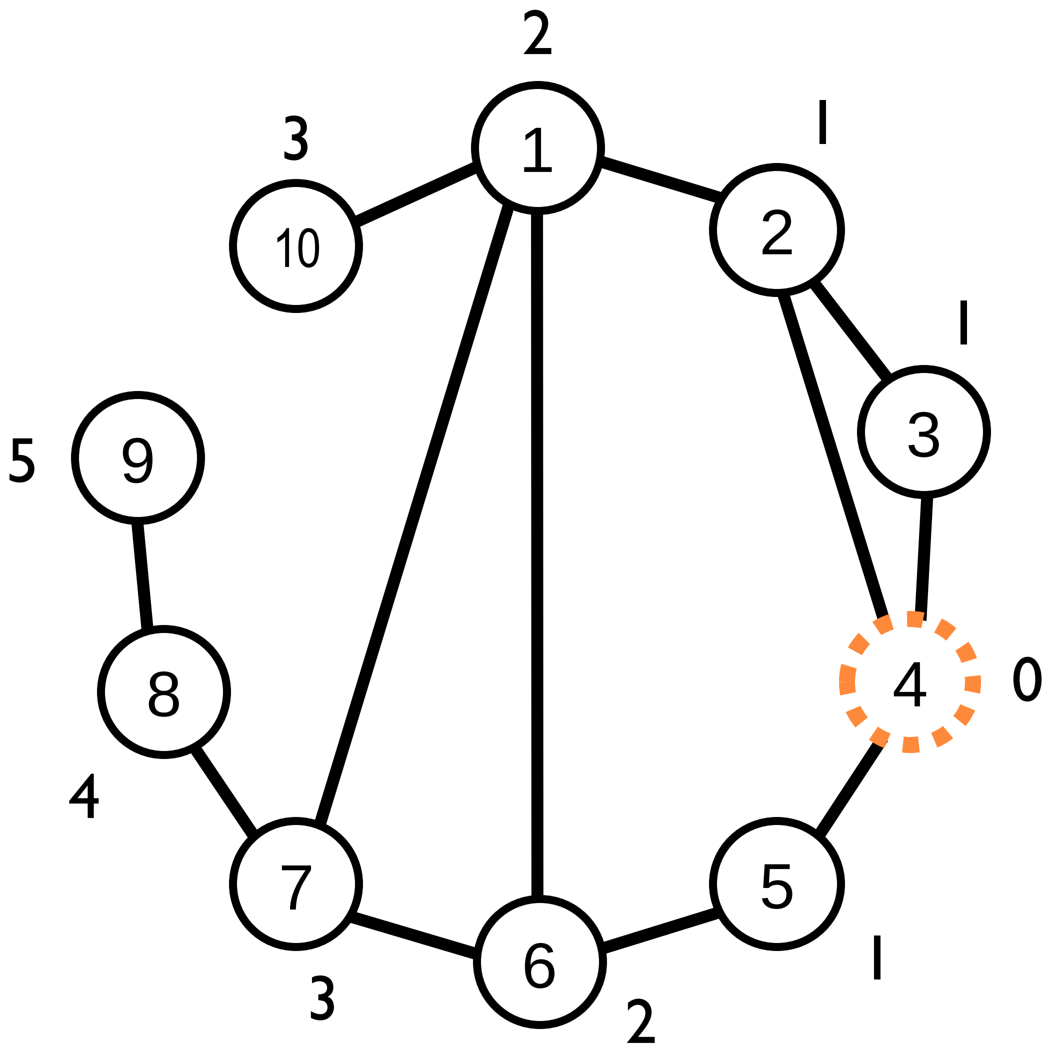
To Find Shortest-Paths, We First Need to Find Distances

- Let’s try to find the distances for each node in order from 1 to 10
- How did we do it?
- One way: to know the distance of some node, n find the shortest path from n to the treasure and take its length
  - No good: we just returned to the problem we started with



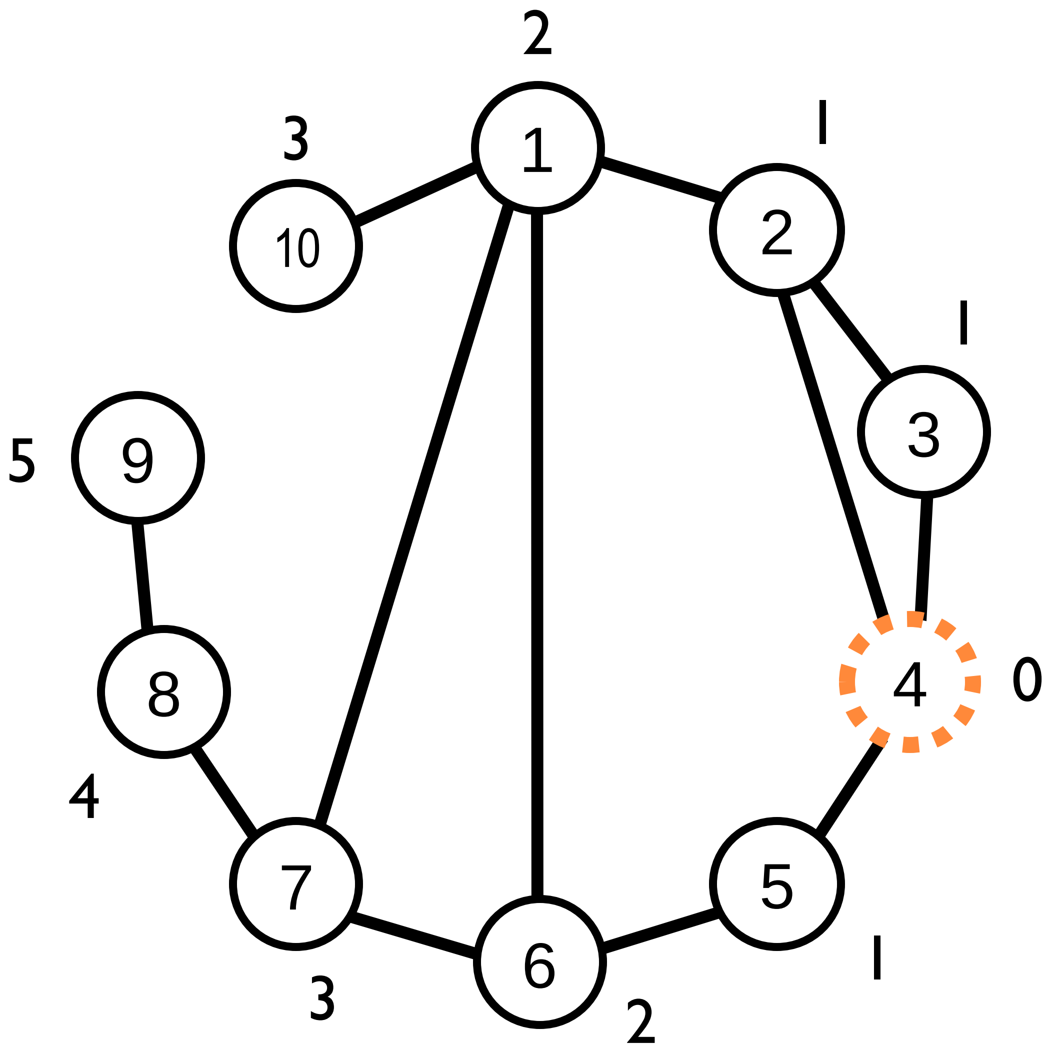
To Find Shortest-Paths, We First Need to Find Distances

- Is there some “nice order” to fill in `dist` that makes it obvious what each entry should be?
  - Without needing to find the shortest paths first



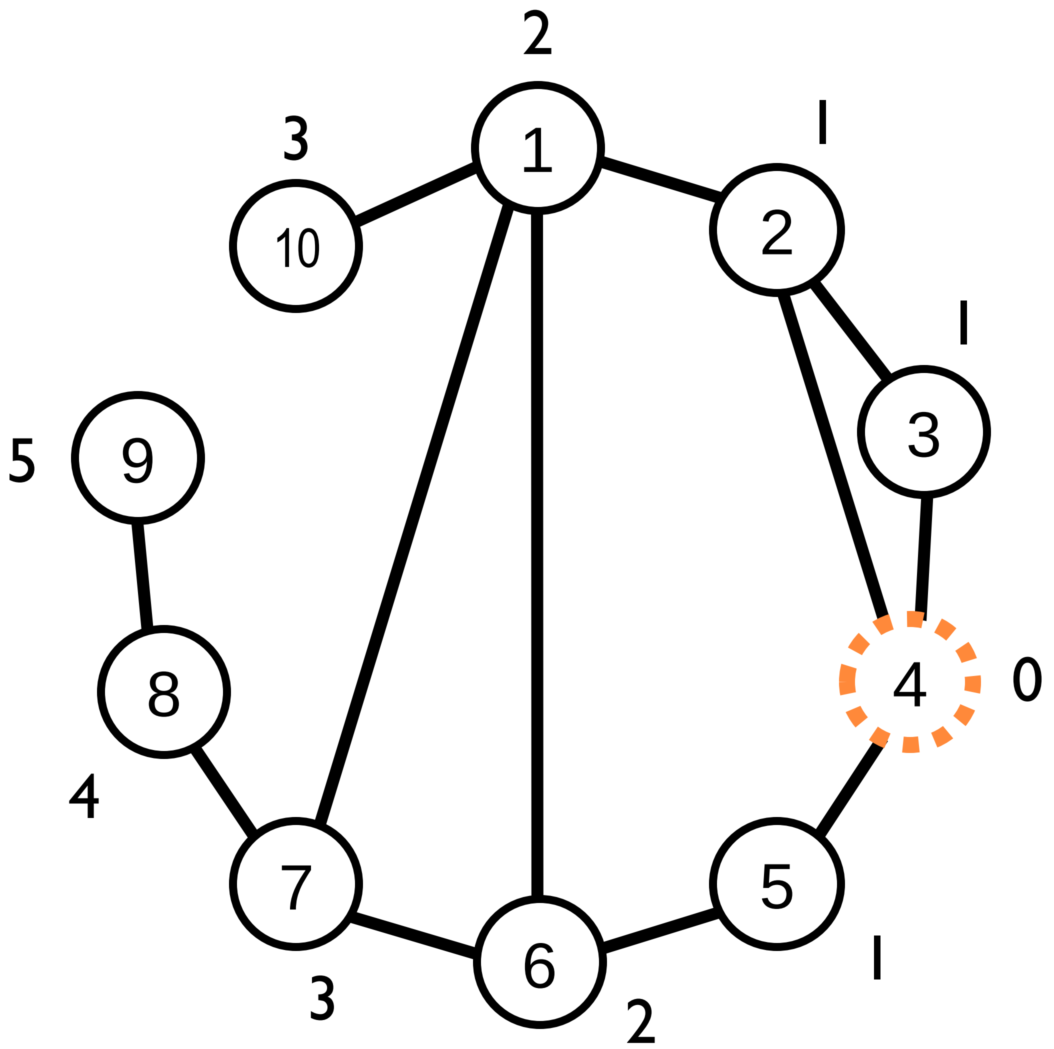
To Find Shortest-Paths, We First Need to Find Distances

- Node 4 is distance 0
- Nodes 2, 3, 5 are distance 1
  - They are neighbors of node 4
- Nodes 1 and 6 are distance 2
  - They are neighbors of 2 or 5, which are neighbors of node 4
  - They are (neighbors of) x 2 node 4
- Nodes 7 and 10 are distance 3
  - They are (neighbors of) x 3 node 4



Generalizing

- The treasure is distance 0
- The neighbors of the treasure are distance 1
- The neighbors of those nodes are distance 2
  - Except those already marked
- The neighbors of nodes at distance i are distance $i + 1$
  - Except those already marked



Turning This into an Algorithm

- Need to repeatedly do the following:
  - Have some node, let's say it's at distance i
  - Tell its neighbors that they are at distance $i + 1$, *but only those not already marked*
  - Repeat the process for newly marked neighbors
- Familiar?
- This is the gossip-spreading algorithm!

What Needs to Change Here?

```
has_heard_gossip = [False for _ in range(n)]
has_heard_gossip[starting_person] = True
queue = [starting_person]

while len(queue) > 0:
    gossiper = queue.pop(0)
    for friend in friends[gossiper]:
        if not has_heard_gossip[friend]:
            has_heard_gossip[friend] = True
            queue.append(friend)
```

Note: We use the unoptimized version for simplicity – you can optimize later

First, Let's Use Less Silly Variable Names

```python
marked = [False for _ in range(n)]
marked[treasure] = True
queue = [treasure]

while len(queue) > 0:
    u = queue.pop(0)
    for v in neighbors[u]:
        if not marked[v]:
            marked[v] = True
            queue.append(v)
```

Now We Need to Store the Distances

```
marked = [False for _ in range(n)]
marked[treasure] = True
dist = [None for _ in range(n)]
dist[treasure] = 0
queue = [treasure]

while len(queue) > 0:
    u = queue.pop(0)
    for v in neighbors[u]:
        if not marked[v]:
            marked[v] = True
            dist[v] = dist[u] + 1
            queue.append(v)
```

A Simplification

- `dist[v] is None` implies `marked[v] == False`, so don't need `marked` anymore

A Simplification

```python
dist = [None for _ in range(n)]
dist[treasure] = 0
queue = [treasure]

while len(queue) > 0:
    u = queue.pop(0)
    for v in neighbors[u]:
        if dist[v] is None:
            dist[v] = dist[u] + 1
            queue.append(v)
```

In C++, Use Some Dummy Value in Place of None

```
vector<int> dist(n, -1); // initializes all to -1
dist[treasure] = 0;
vector<int> queue = {treasure};

while(queue.size() > 0) {
    int u = queue[0];
    queue.erase(queue.begin());
    for(int v : neighbors[u]) {
        if(dist[v] == -1) {
            dist[v] = dist[u] + 1;
            queue.push_back(v);
        }
    }
}
```

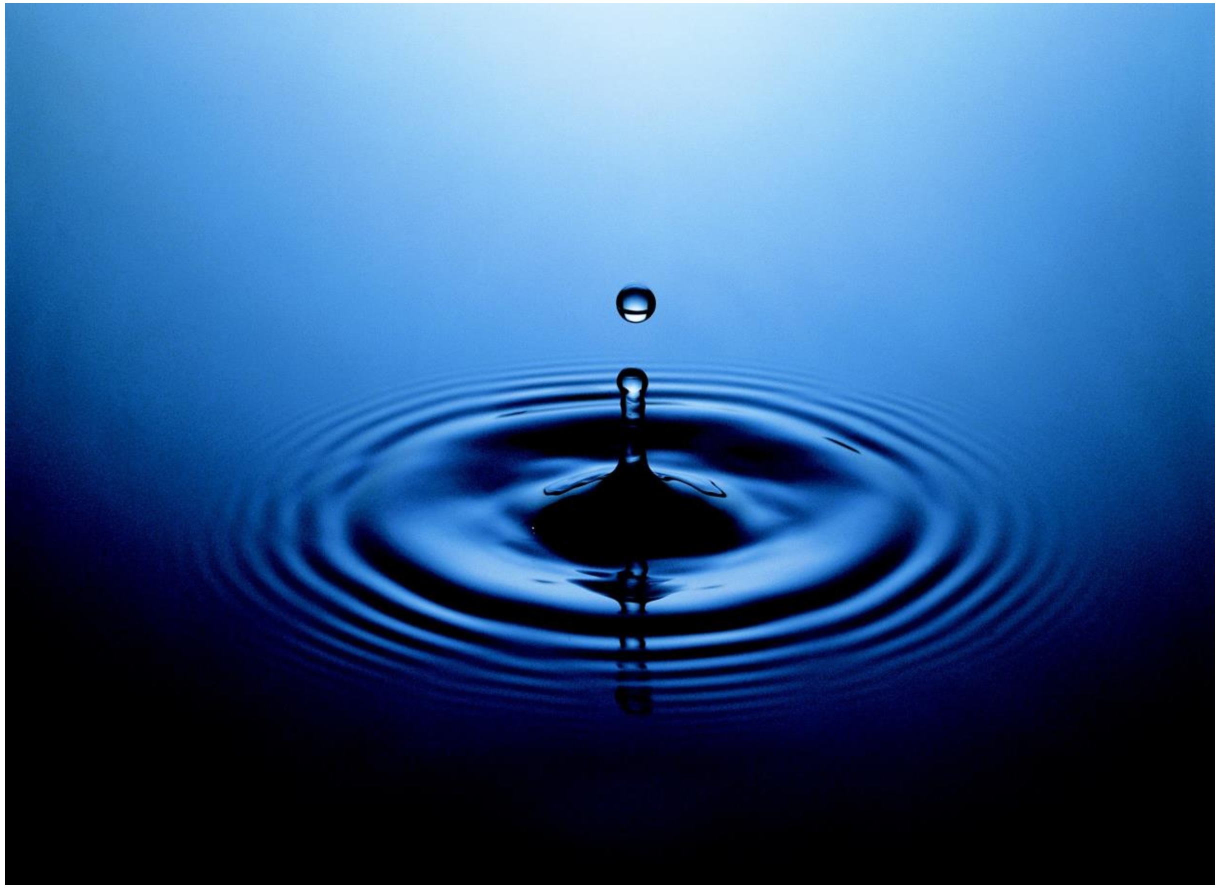
Note: We choose -1 because it can never be a valid distance

Question

- Why do we start with `treasure` in the queue, instead of `entrance`?
- Answer: to construct the path to the treasure *starting from the entrance*, we need to know the distances of every room *from the treasure*, not from the entrance
- It's possible to start with `entrance` in the queue instead, computing the distances of every room *from the entrance*
  - But in the case, the path needs to be constructed *starting from the treasure* instead (and then reversed)

Breadth-First Search

- No one calls this algorithm "gossip-spreading" – everyone calls it **breadth-first search** or BFS for short
- Because the algorithm repeatedly expands the set of "visited" nodes "by layers," expanding along the *breadth* of the previous layer to produce the next layer



Versus Depth-First Search

- Remember this from a few slides ago?
  - One issue: when a person tells the gossip to all their friends, there may be multiple who hear it for the first time and need to repeat the same process "at the same time"
  - We (as of now) don't have a way to tell the computer to do several things in parallel
- Actually, there kinda is a way, using recursion
  - Come back to this after the recursive backtracking lesson if you want to think about it

Versus Depth-First Search

- Remember this from a few slides ago?
  - Removing from the front of a list takes linear time, because all elements need to be shifted by one index forward
    - Worst case: started with a person who is friends with everyone
  - Don’t want to do this repeatedly
  - Instead, never remove and just keep track of where the first unprocessed person is
- We can also just remove from the back – it is $\Theta(1)$
  - Since we said the order people are processed doesn’t matter for finding connected components

Versus Depth-First Search

- Doing either of these results in a slightly different algorithm called **depth-first search**
- Not discussed in detail here – you can study on your own
- Just note that for shortest paths, the order in which the nodes are processed does matter
  - DFS solves connected components, but not shortest paths
  - You can think about why at home!

Practice Problems

- https://progvar.fun/problemsets/bfs-dfs

Thanks!

accenture

PLDT

stratpoint
FAST FORWARD TO THE FUTURE

MERALCO

Thinking Machines
Data Science