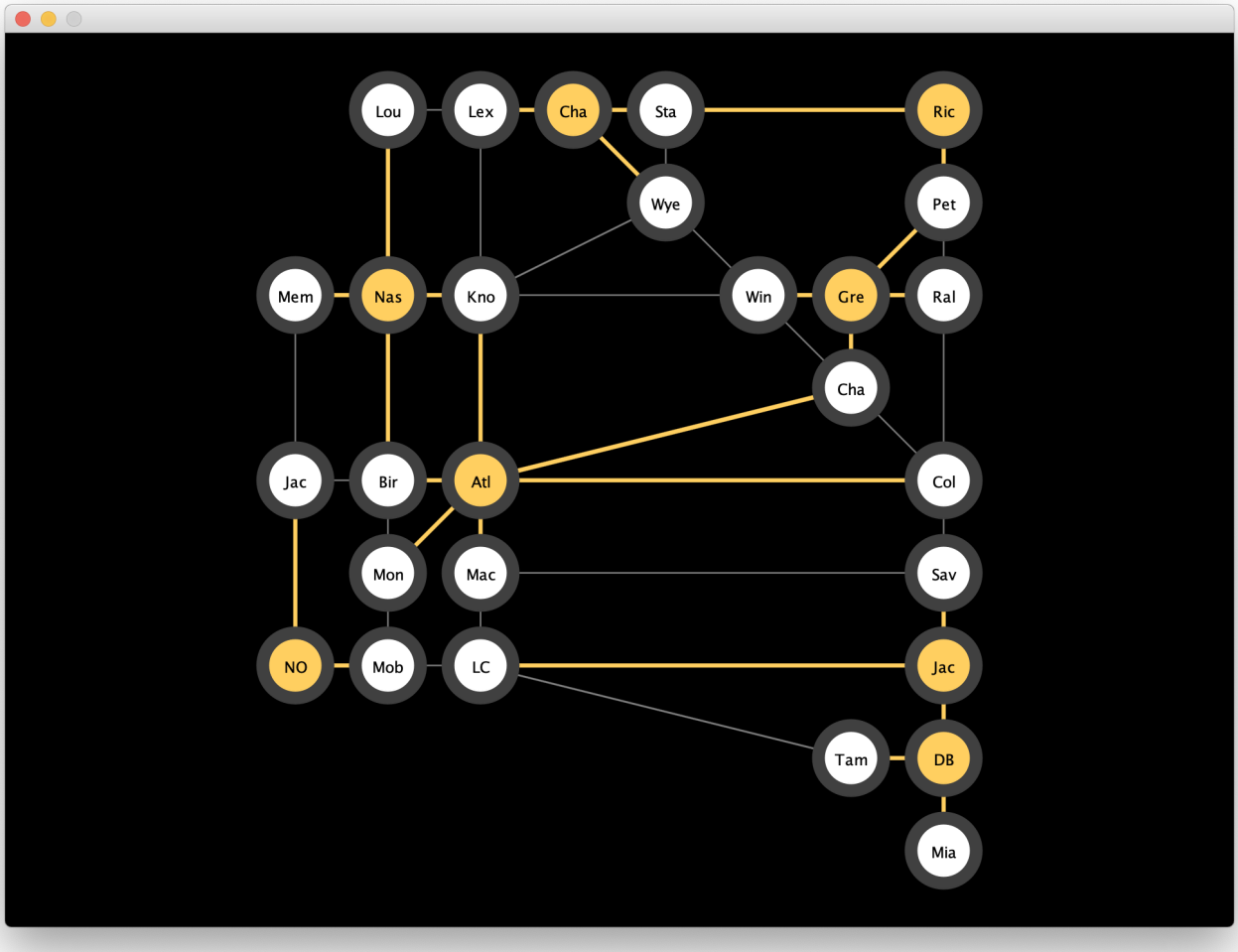
Problem 2: Disaster Preparations

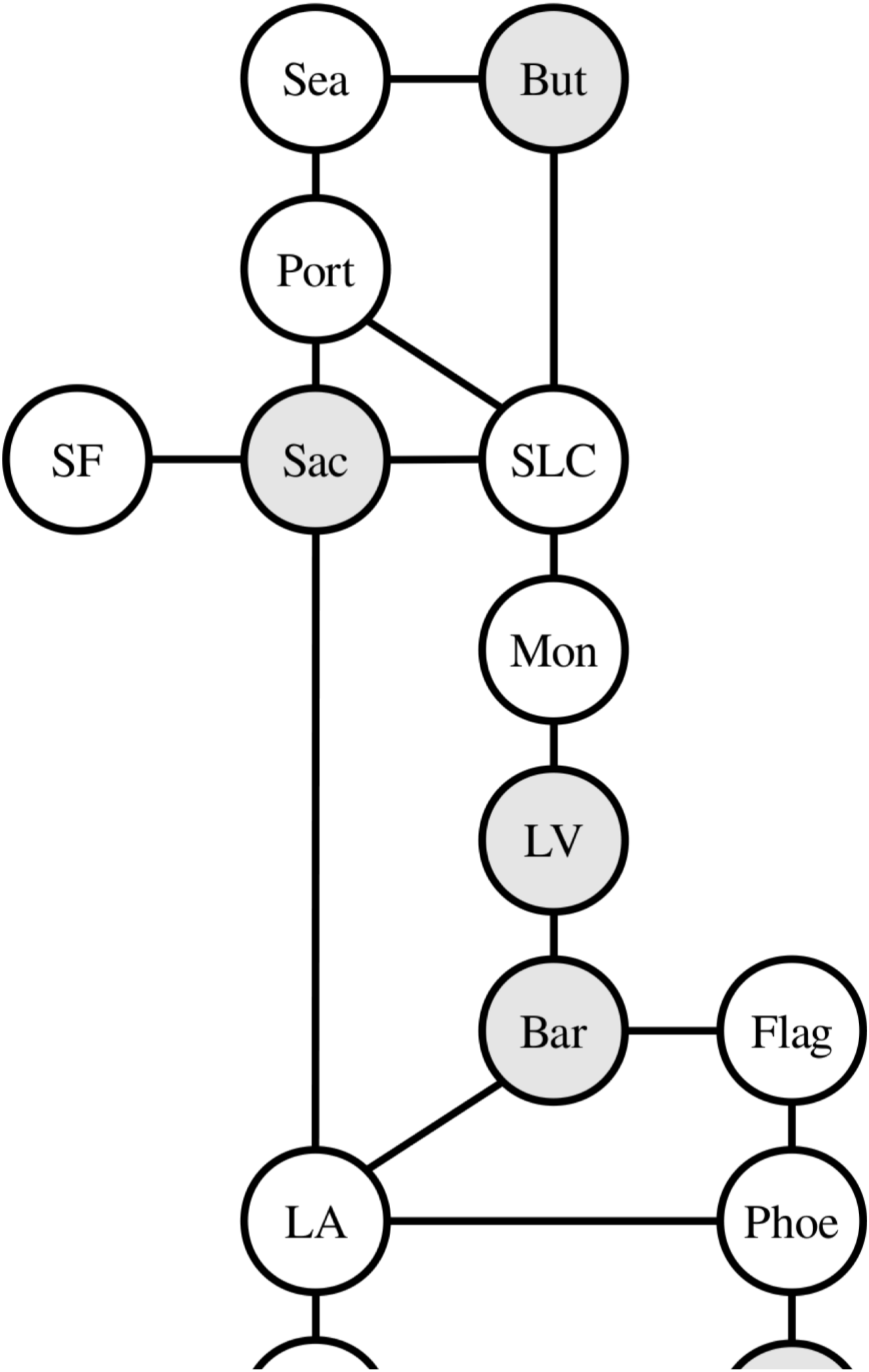


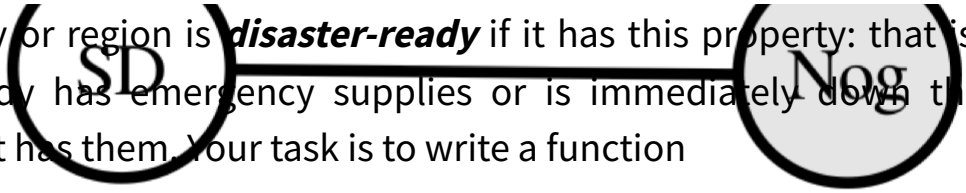
Disasters – natural and unnatural – are inevitable, and cities need to be prepared to respond to them when they occur. The problem is that stockpiling emergency resources can be really, really expensive. As a result, it's reasonable to have only a few cities stockpile emergency resources, with the plan that they'd send those resources from wherever they're stockpiled to where they're needed when an emergency happens. The challenge with doing this is to figure out where to put resources so that (1) we don't spend too much money stockpiling more than we need, and (2) we don't leave any cities too far away from emergency supplies.

Imagine that you have access to a country's major highway networks. We can imagine that there are a number of different cities, some of which are right down the highway from others. To the right is a fragment of the US Interstate Highway

System for the Western US. Suppose we put emergency supplies in Sacramento, Butte, Las Vegas, Barstow, and Nogales (shown in gray). In that case, if there's an emergency in any city, that city either already has emergency supplies or is immediately adjacent from a city that does. For example, any emergency in Nogales would be covered, since Nogales already has emergency supplies. San Francisco could be covered by supplies from Sacramento, Salt Lake City is covered by both Sacramento and Butte, and Barstow is covered both by itself and by Las Vegas.

Although it’s possible to drive from Sacramento to San Diego, for the purposes of this problem the emergency supplies stockpiled in Sacramento wouldn’t provide coverage to San Diego, since they aren’t immediately next to one another.


Sea
But
Port
SF
Sac
SLC
Mon
LV
Bar
Flag
LA
Phoe



We'll say that a country or region is ***disaster-ready*** if it has this property: that is, every city either already has emergency supplies or is immediately down the highway from a city that has them. Your task is to write a function

```
bool canBeMadeDisasterReady(const Map<string, Set<string>>& roadNetwork,
                            int numCities, Set<string>& locations);
```

that takes as input a **Map** representing the road network for a region (described below) and a number of cities that can be made to hold supplies, then returns whether it's possible to make the region disaster-ready by placing supplies in at most **numCities** cities. If so, the function should then populate the argument **locations** with all of the cities where supplies should be stored.

In this problem, the road network is represented as a map where each key is a city and each value is a set of cities that are immediately down the highway from them. For example, here's a fragment of the map you'd get from the above transportation network:

```
"Sacramento":    {"San Francisco", "Portland", "Salt Lake City", "Los Angeles"}
"San Francisco": {"Sacramento"}
"Portland":      {"Seattle", "Sacramento", "Salt Lake City"}
```

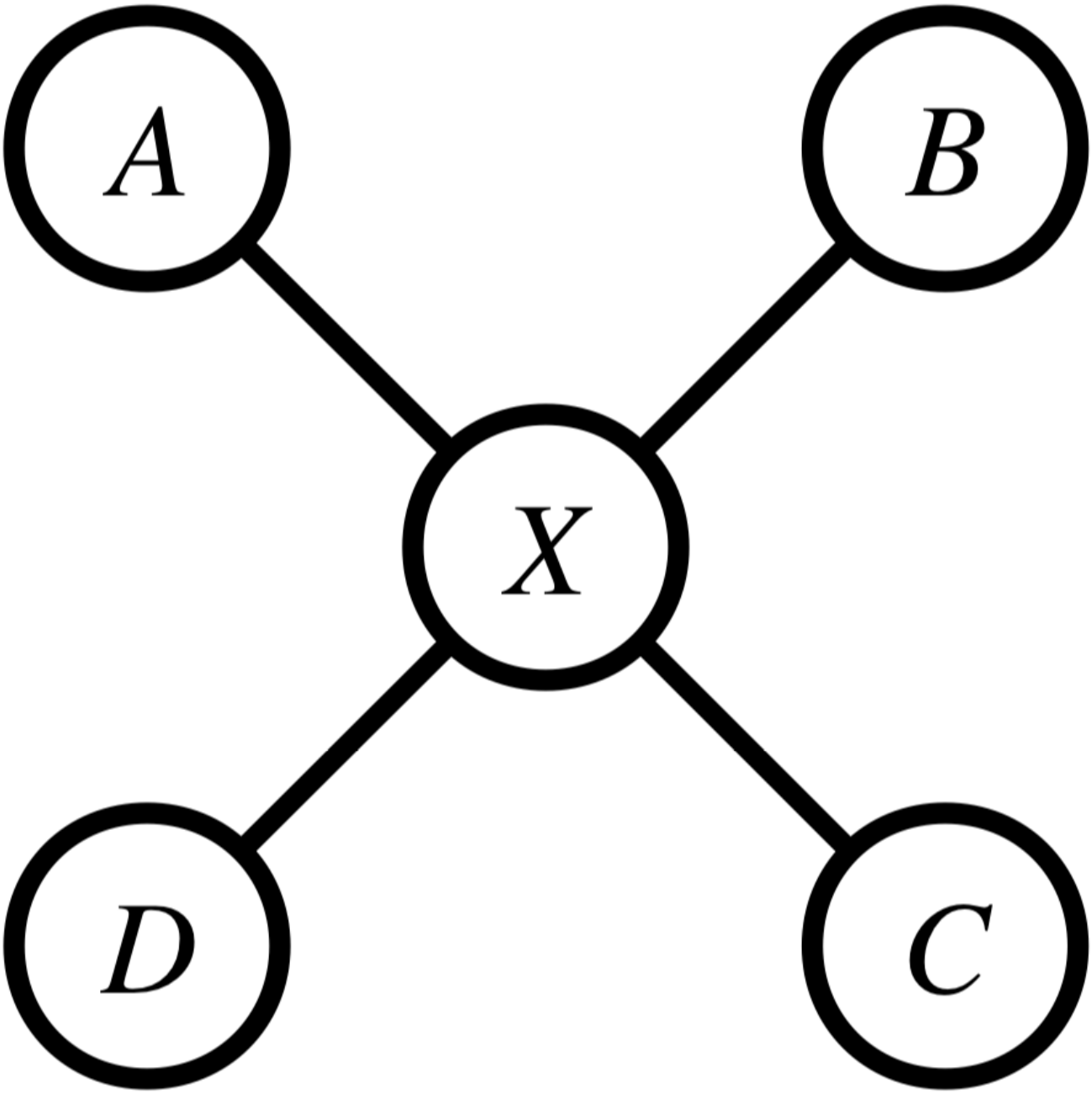
As in the first part of this assignment, you can assume that locations is empty when this function is first called, and you can change it however you'd like if the function returns false.

There are many different strategies one could use to solve this problem, and some are more efficient than others. For example, one option would be to treat this problem as a combinations problem, trying out each combination of cities of the given size and seeing whether any of them would make the cities disaster ready. This option works, but it will be extremely slow on some of the larger test cases where there are over thirty cities – so slow in fact that your program might never give back an answer!

Instead, here is the approach you **must** use in this problem. Imagine there's some city X in the transportation grid that's adjacent to four neighboring cities, as shown here. Any collection of cities that makes this grid disaster ready is going to have to provide some kind of coverage to city X. Even if you have no idea which cities get chosen in the long run, you can say for certain that you'll need to include at least one of A, B, C, D, or X.

Therefore, the approach should be to find a city that's uncovered, then think of all the different ways you could cover it (either by choosing an adjacent city or by choosing the city itself). If you can cover all the remaining cities after making any of those choices, congrats! You're done. On the other hand if, no matter which of these choices you commit to, you find that there's no way to cover all the cities, you know that no solution to your particular subproblem exists.

This second approach tends to run a lot faster than the first. The recursion focuses more of its effort looking at adding cities that make an impact, and it's a lot more obvious when you're in a dead-end.



Some notes on this problem:

- The road network is bidirectional. If there's a road from city A to city B, then there will always be a road back from city B to city A in the network, and both

roads will be present in the parameter **`roadNetwork`**. You can rely on this.

- Every city appears as a key in the map. Cities can exist that aren't connected to any other cities in the transportation network. If that happens, the city will be represented by a key in the map associated with an empty set of adjacent cities.
- If you're allowed to use up to, say, **k** cities, but you find a way to solve the problem using fewer than **k** cities, that's fine! The set of cities you return can contain any number of cities provided that you don't use more than **k** and you properly cover everything.
- Remember that if you try stockpiling somewhere and it doesn't work out, it and its neighbors may still be covered! Always remember that there may be other stockpiled cities that cover them as well.

The test cases we've bundled with the starter code here include some simplified versions of real transportation networks from around the world. Play around with them and let us know if you find anything interesting as you do! Our starter code also contains an option to use your code to find the minimum number of cities needed to provide disaster protection in a region, which you might find interesting to try out once you've gotten your code working.

A note: some of the test files that we've included have a *lot* of cities in them. The provided test cases whose names start with **`VeryHard`** are, unsurprisingly, very hard tests that may require some time to solve. It's okay if your program takes a long time (say, at most two minutes) to answer queries for those maps, though if you use the strategy outlined above you should probably be able to get solutions back for them in only a matter of seconds.

Debugging: the visualizer built in to the starter code may help you visualize the network being run and find bugs, if there are any. As an additional (optional) benefit, you can also call the following additional graphics methods as part of your solution that can highlight what your algorithm is doing. You are **not required** to use this in your solution, but feel free to do so if it helps you implement it.

Method	Description

`setTryingToCoverCity(cityName, tryingToCover)`	Indicate to the visualizer that you are looking at a city to think of the different ways you could cover it (pass in **`true`** for **`tryingToCover`**), or are no longer doing so (pass in **`false`** for **`tryingToCover`**). The visualizer will highlight the most recent call with light blue, and earlier calls that are still being covered with dark blue.

`setStockpileInCity(cityName, stockpile)`	Indicate to the visualizer that you are marking this city as holding a stockpile (pass in **`true`** for **`stockpile`**) or not holding a stockpile (pass in **`false`** for **`stockpile`**). The visualizer will highlight stockpiles, and roads directly connected to them, in yellow. It will also highlight cities covered by stockpiled neighbors in white.

You can also consider adding **`pause(ms);`** calls to slow down the highlighting and have it animate at a reasonable speed. The **`ms`** parameter is the number of milliseconds to pause the program for. Note that the visualizer expects cities to be marked "uncovered" in the reverse order they were marked "covered" (e.g. trying to cover A, and then trying to cover B, it expects you first stop trying to cover B, then stop trying to cover A).
