YEAH A4

Recursive Backtracking

Logistics

- This assignment is due next Friday, 2/12 at the start of class.

Logistics

- This assignment is due next Friday, 2/12 at the start of class.
- Once again, you **are** allowed to work in pairs on this assignment. We think this one's pretty tricky, so be sure to start early!

Part 1: Warmups

We have two **debugging** warmups for you to complete before the coding portions of the assignment!

Part 1: Warmups

We have two **debugging** warmups for you to complete before the coding portions of the assignment!

- In the first one, you'll be examining **The Towers of Hanoi**, a famous recursive problem. You'll be responsible for stepping through the recursive function and reporting back various info to us. Here are the helpful steps in the debugger:

- Stepping **over** a recursive call can be helpful when thinking holistically. A recursive call is simply a "magic" black box that completely handles the smaller subproblem.
- Stepping **into** a recursive call allows you to trace the nitty-gritty details of moving from an outer recursive call to the inner call.
- Stepping **out** of a recursive call allows you to follow along with the action when backtracking from an inner recursive call to the outer one.

Part 1: Warmups

We have two **debugging** warmups for you to complete before the coding portions of the assignment!

- In the second warmup, you'll be examining a **buggy** implementation of code that finds permutations. It'll be your job to figure out the little error that causes the bug, and after, you'll need to reflect as to *why* the that bug was so catastrophic.

Part 1: Warmups

We have two **debugging** warmups for you to complete before the coding portions of the assignment!

- In the second warmup, you'll be examining a **buggy** implementation of code that finds permutations. It'll be your job to figure out the little error that causes the bug, and after, you'll need to reflect as to *why* the that bug was so catastrophic.
    - We designed these warmups specifically because we've seen these error come up many times in the past! If you have a good understanding of why the permutations code doesn't work, you'll have a much better understanding of recursion / backtracking!

Let's hop into the code!

Poster for the 2015 film "Backtrack." Critics gave it a paltry 30% on Rotten Tomatoes, citing "not enough recursion."

Part 2: Doctors Without Orders

- The more programming knowledge we know, the more we can use CS to solve real life problem!

Part 2: Doctors Without Orders

- The more programming knowledge we know, the more we can use CS to solve real life problem!
- A growing application of CS is in the field of Healthcare.

Stanford AI in Healthcare

Part 2: Doctors Without Orders

- The more programming knowledge we know, the more we can use CS to solve real life problem!
- A growing application of CS is in the field of Healthcare.
- Let's utilize the recursion skill we have practiced so far to solve a cool problem!

Stanford AI in Healthcare

Part 2: Doctors Without Orders

- We want to see if there is a way to match the patients with the doctors such that all of the patients are taken care of.

Part 2: Doctors Without Orders

- We want to see if there is a way to match the patients with the doctors such that all of the patients are taken care of.
- Each doctor has a maximum number of hours they can work for.
  - `Map<string, int> doctors;`

Part 2: Doctors Without Orders

- We want to see if there is a way to match the patients with the doctors such that all of the patients are taken care of.
- Each doctor has a maximum number of hours they can work for.
  - `Map<string, int> doctors;`
- Each patient has a number of hours they need to be seen for.
  - `Map<string, int> patients;`

Part 2: Doctors Without Orders

- We want to see if there is a way to match the patients with the doctors such that all of the patients are taken care of.
- Each doctor has a maximum number of hours they can work for.
  - `Map<string, int> doctors;`
- Each patient has a number of hours they need to be seen for.
  - `Map<string, int> patients;`
- If a valid matching exist, we also want to keep track of such a matching.
  - `Map<string, Set<string>> schedule;`

Part 2: Doctors Without Orders

- We want to see if there is a way to match the patients with the doctors such that all of the patients are taken care of.
- Each doctor has a maximum number of hours they can work for.
  - `Map<string, int> doctors;`
- Each patient has a number of hours they need to be seen for.
  - `Map<string, int> patients;`
- If a valid matching exist, we also want to keep track of such a matching.
  - `Map<string, Set<string>> schedule;`

Name of doctor

Part 2: Doctors Without Orders

- We want to see if there is a way to match the patients with the doctors such that all of the patients are taken care of.
- Each doctor has a maximum number of hours they can work for.
  - **Map<string, int> doctors;**
- Each patient has a number of hours they need to be seen for.
  - **Map<string, int> patients;**
- If a valid matching exist, we also want to keep track of such a matching.
  - **Map<string, Set<string>> schedule;**

Name of doctor

List of patients matched

Part 2: Doctors Without Orders

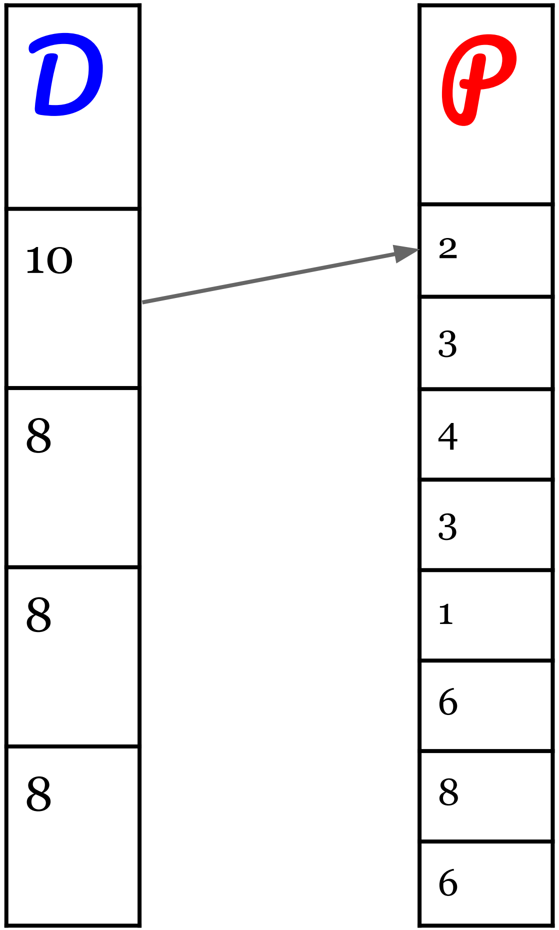
- Let’s walk through a quick example (low animation budget ahead!)

Strategy 1: Grab-n-Go!

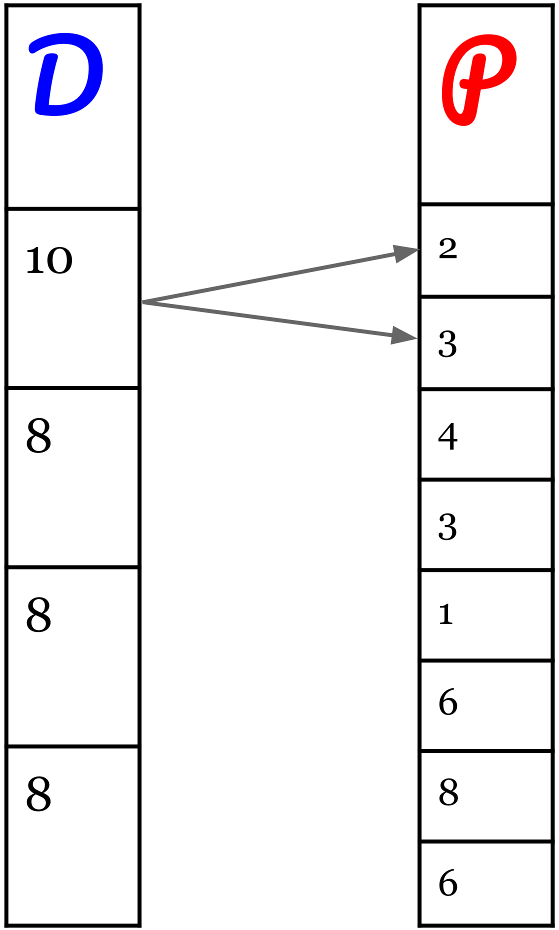
D
10
8
8
8

P
2
3
4
3
1
6
8
6

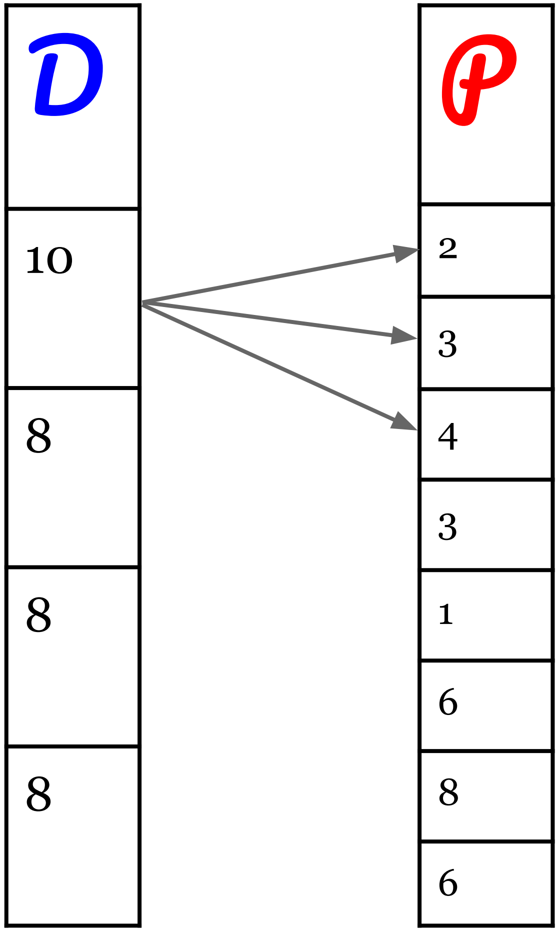
Strategy 1: Grab-n-Go!



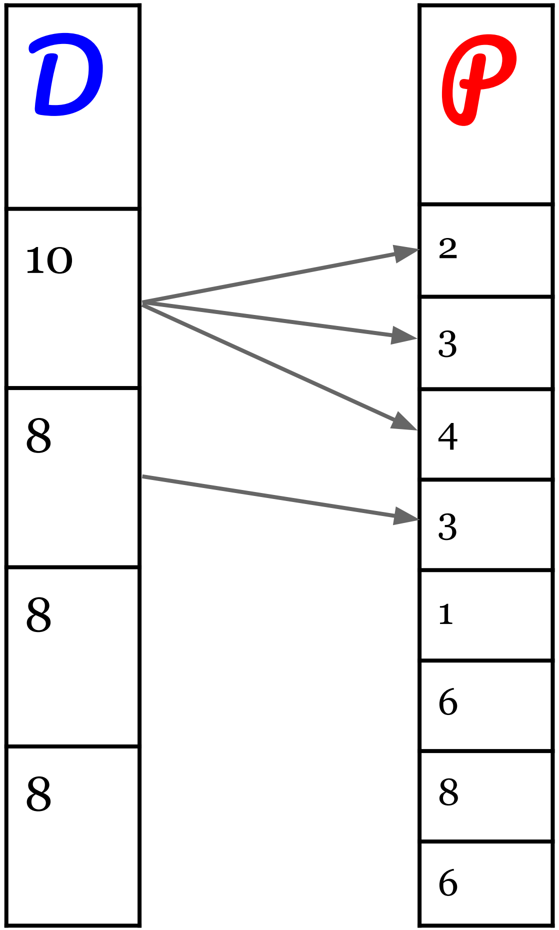
Strategy 1: Grab-n-Go!



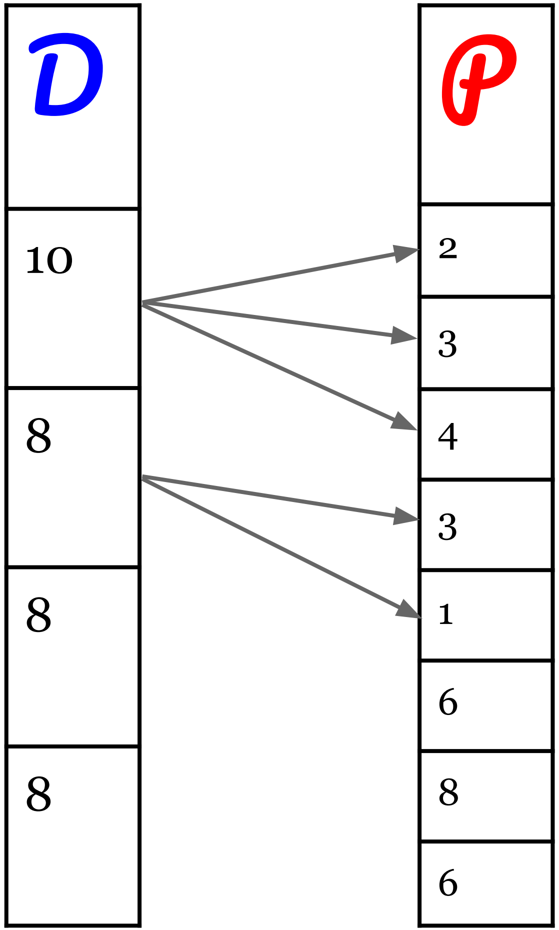
Strategy 1: Grab-n-Go!



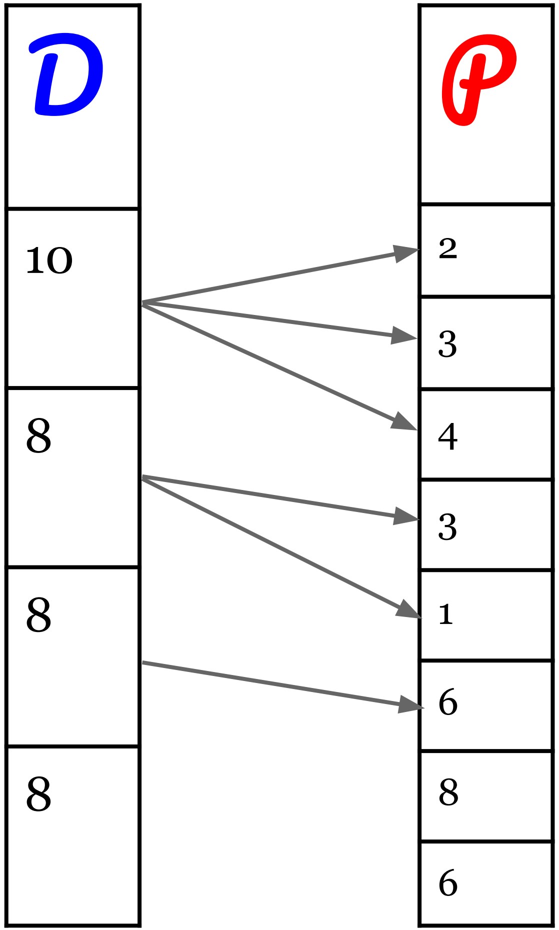
Strategy 1: Grab-n-Go!



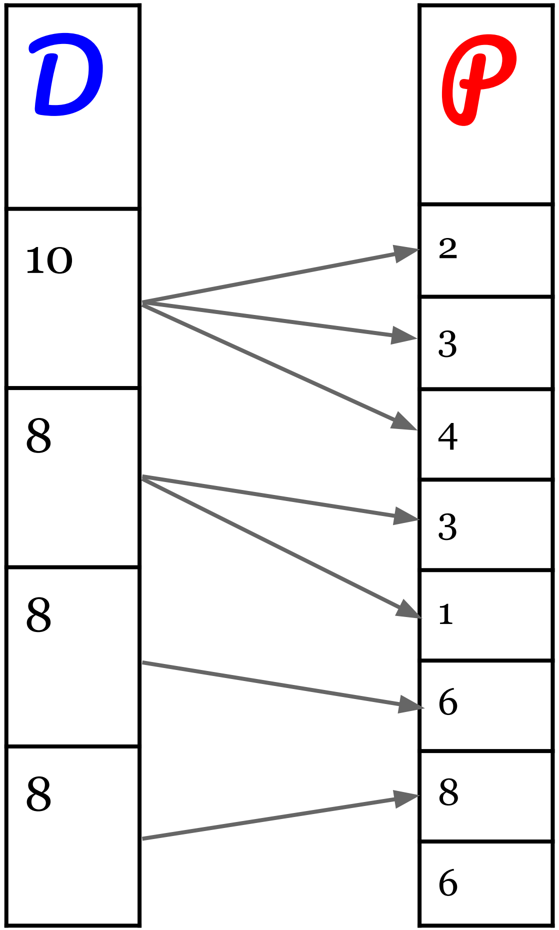
Strategy 1: Grab-n-Go!



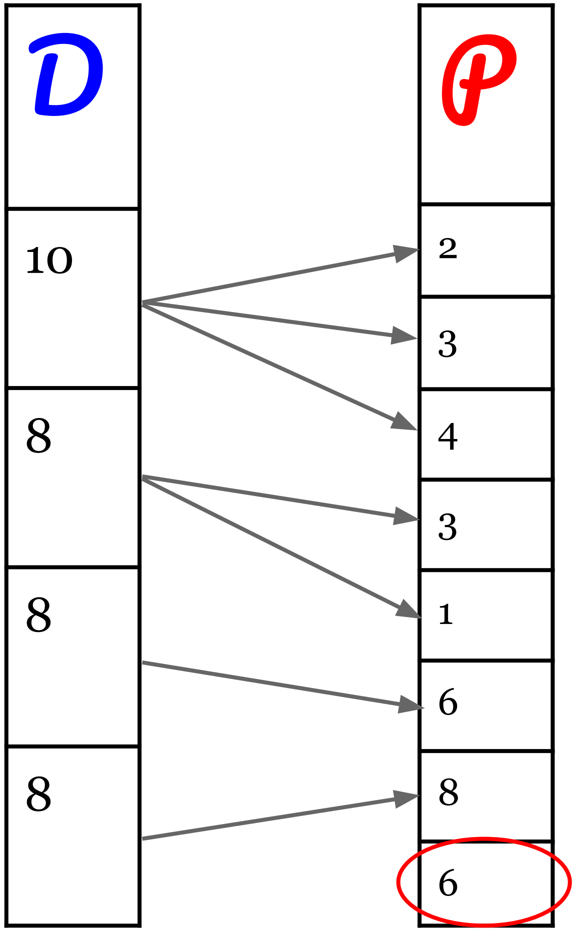
Strategy 1: Grab-n-Go!



Strategy 1: Grab-n-Go!

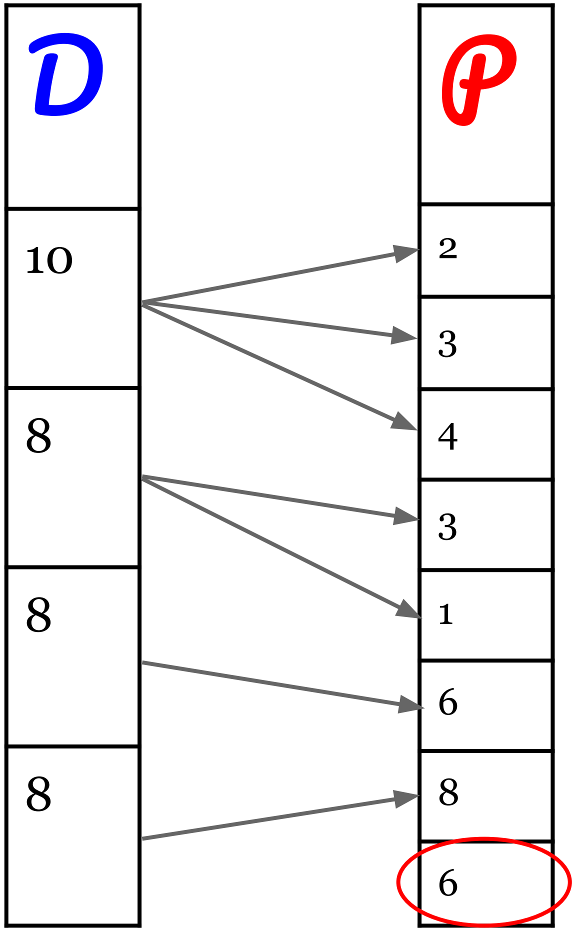


Strategy 1: Grab-n-Go!



Strategy 1: Grab-n-Go!

Surprise! That's didn't work!

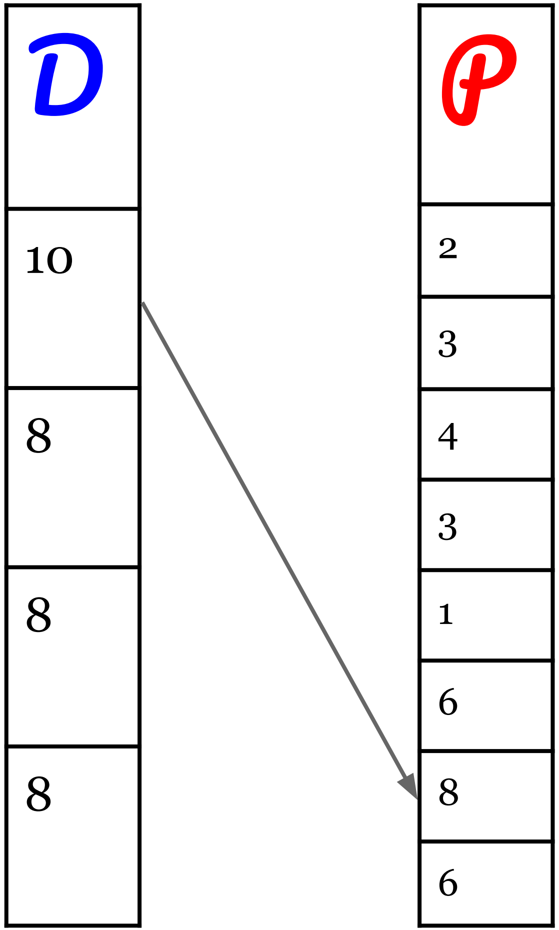


Strategy 2: Greedy

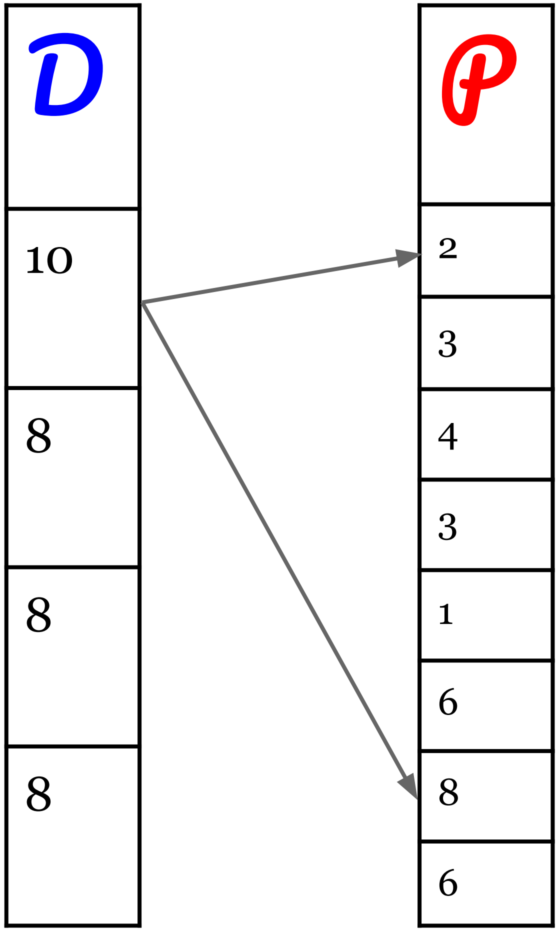
D
10
8
8
8

P
2
3
4
3
1
6
8
6

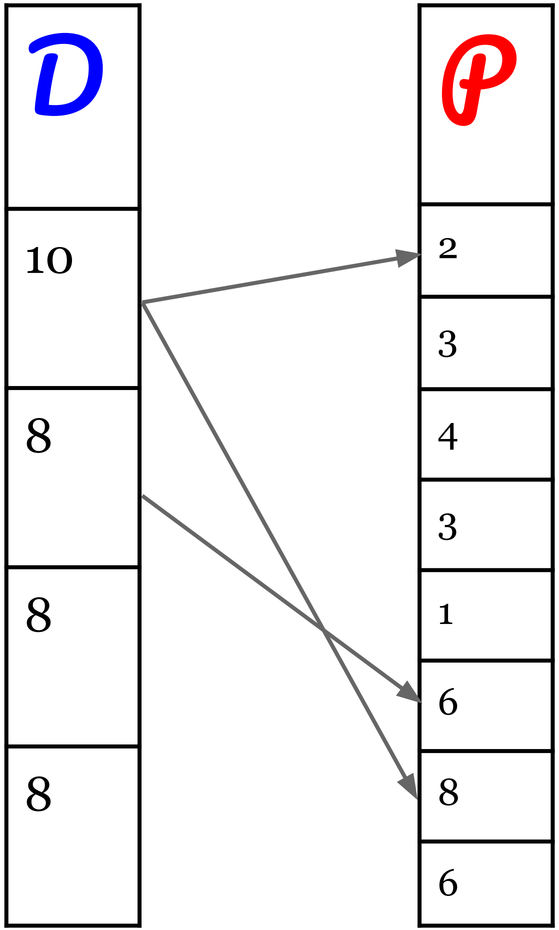
Strategy 2: Greedy



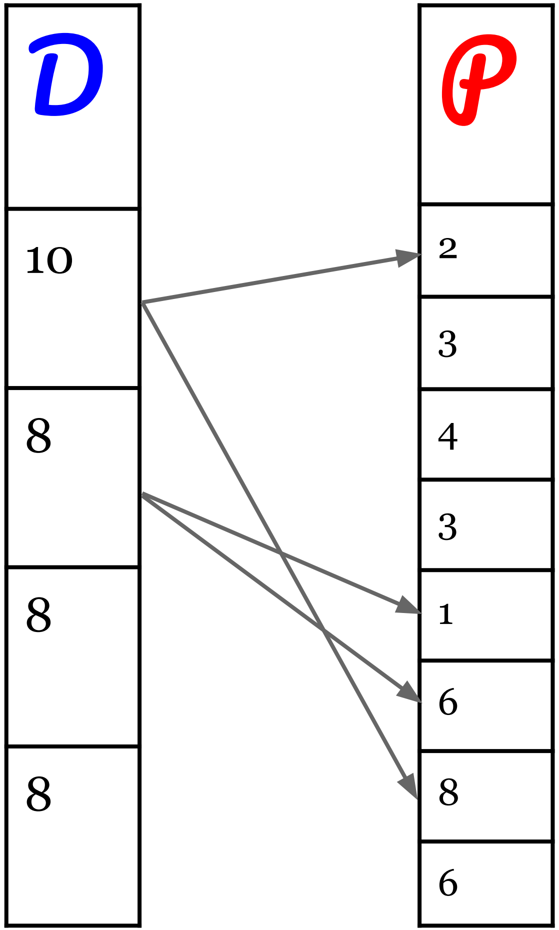
Strategy 2: Greedy



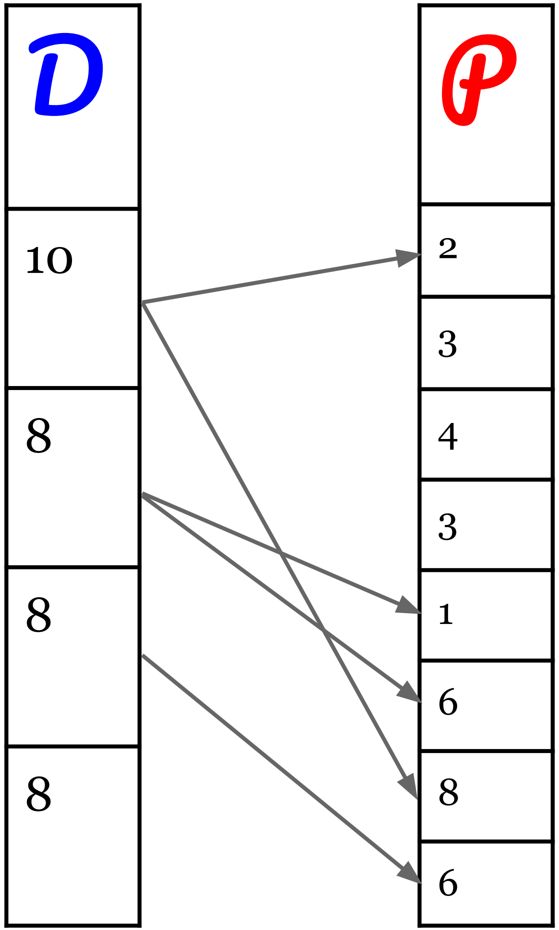
Strategy 2: Greedy



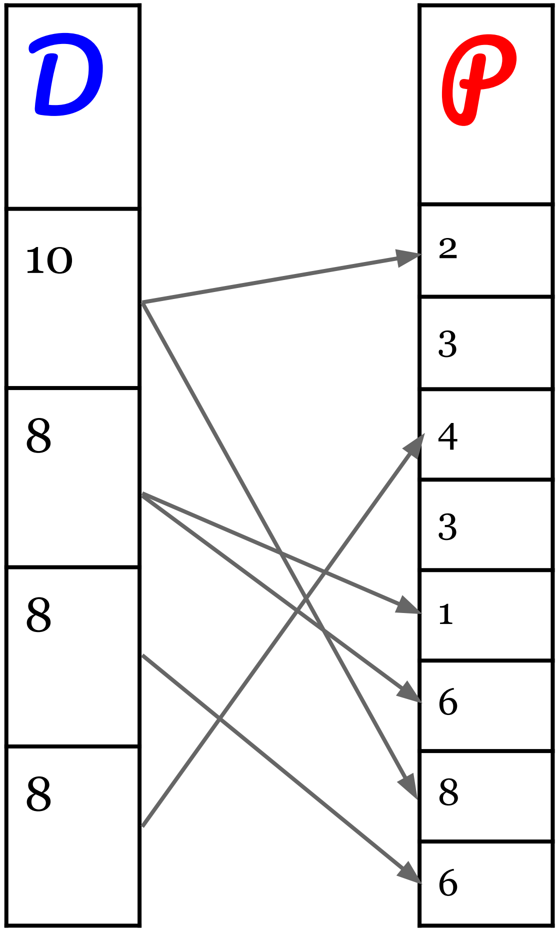
Strategy 2: Greedy



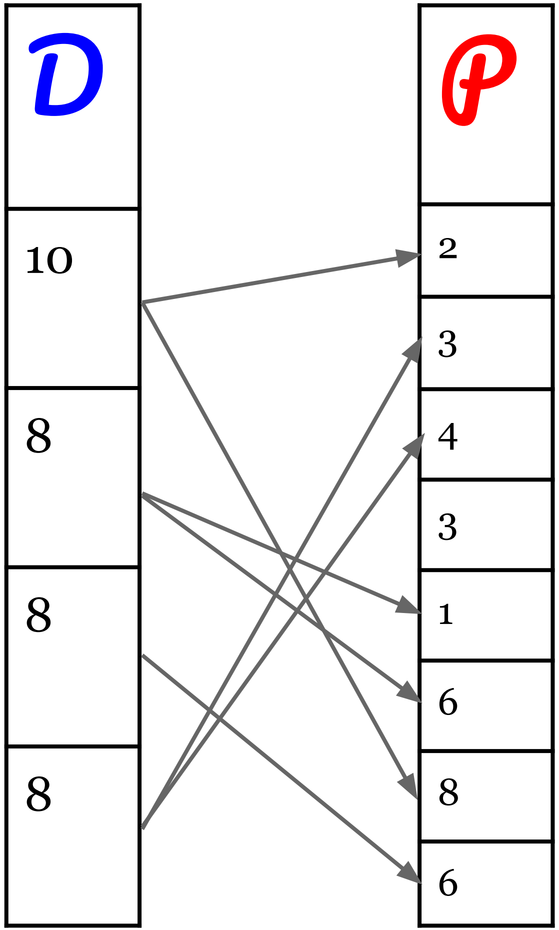
Strategy 2: Greedy



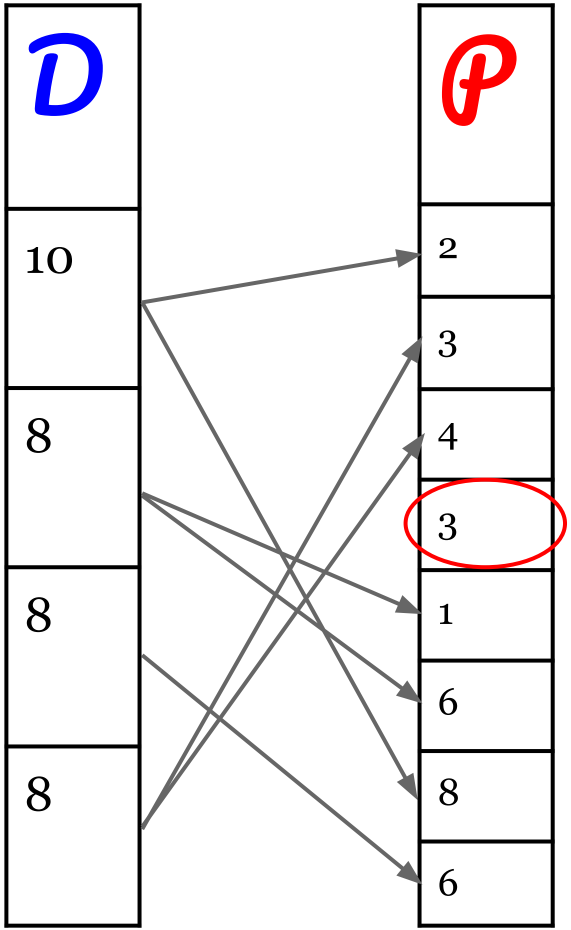
Strategy 2: Greedy



Strategy 2: Greedy

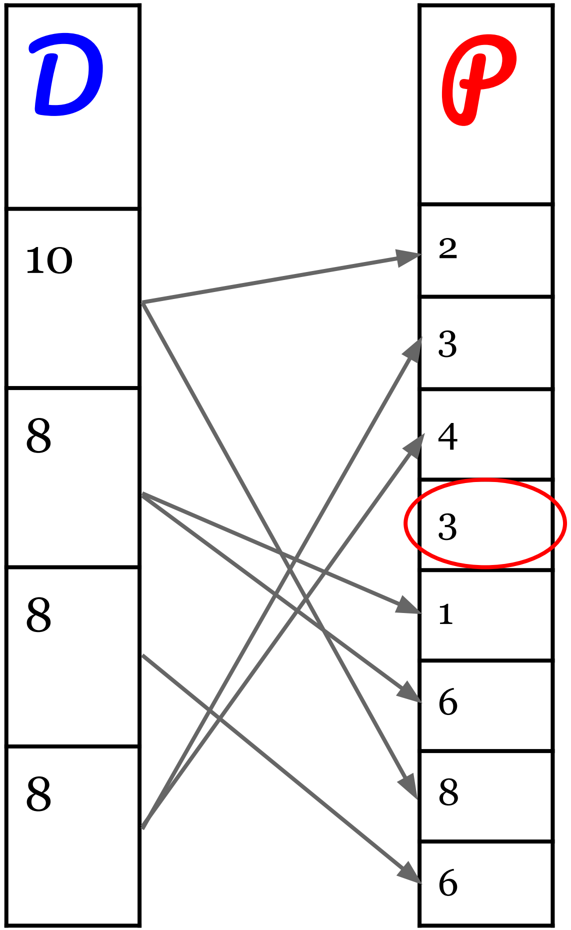


Strategy 2: Greedy



Strategy 2: Greedy

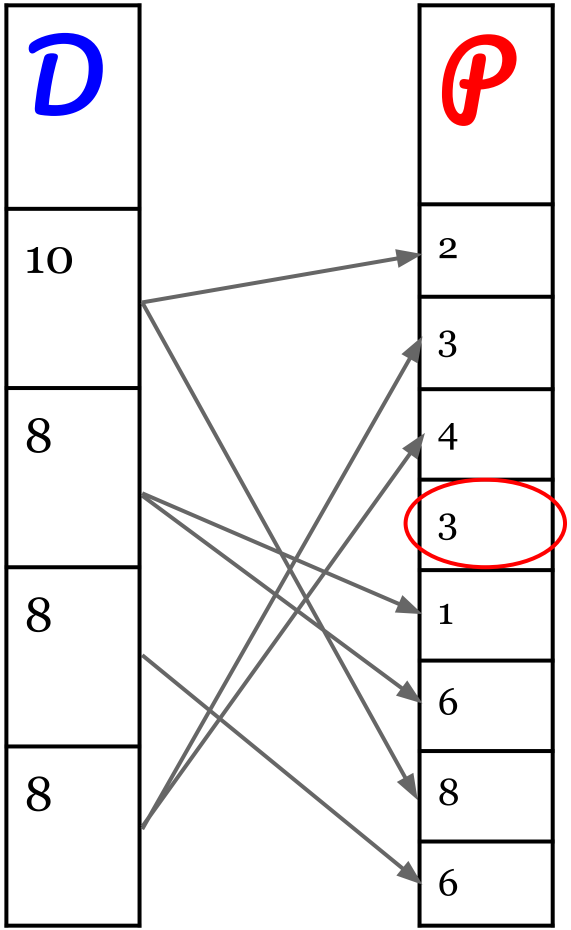
Surprise (this time less sarcastically)!



Strategy 2: Greedy

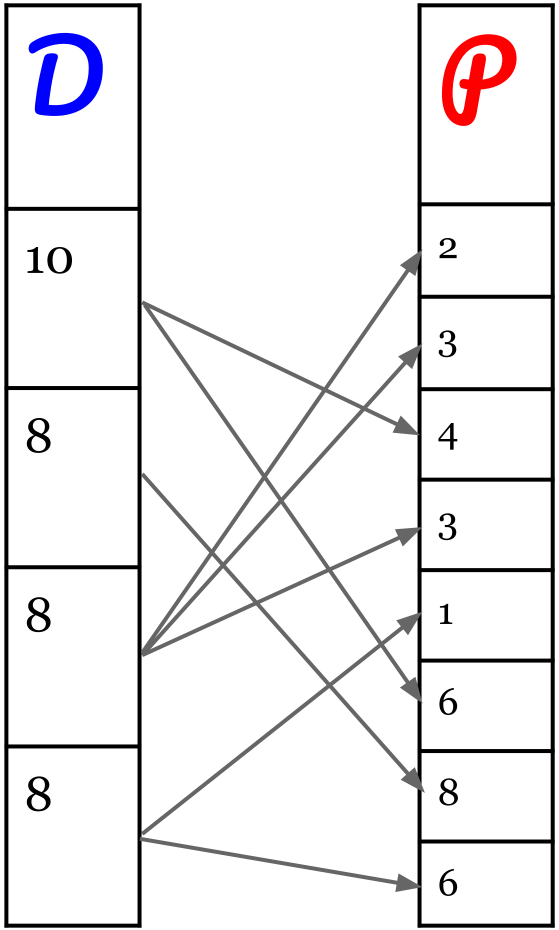
Surprise (this time less sarcastically)!

Note: This is provable! Take CS 161 to find out :)



“Strategy” 3: Oracle

There is indeed a solution here.



Part 2: Doctors Without Orders

- Some notes:
  - Each patient can only be assigned to ONE doctor.

```
bool canAllPatientsBeSeen(const Map<string, int>& doctors,
                          const Map<string, int>& patients,
                          Map<string, Set<string>>& schedule);
```

Part 2: Doctors Without Orders

- Some notes:
    - Each patient can only be assigned to ONE doctor.
    - Doctors don’t have to use up all of their hours.
        - In fact, it is totally fine to have a doctor not matched with any patient.

```
bool canAllPatientsBeSeen(const Map<string, int>& doctors,
                          const Map<string, int>& patients,
                          Map<string, Set<string>>& schedule);
```

Part 2: Doctors Without Orders

- Some notes:
  - Each patient can only be assigned to ONE doctor.
  - Doctors don’t have to use up all of their hours.
    - In fact, it is totally fine to have a doctor not matched with any patient.
  - We don’t care what “schedule” contains if not matching is possible.
    - Although we imagine that you don’t have to explicitly address this case.

```
bool canAllPatientsBeSeen(const Map<string, int>& doctors,
                          const Map<string, int>& patients,
                          Map<string, Set<string>>& schedule);
```

Part 2: Doctors Without Orders

- Some implementation thoughts:
  - Start simple! We can ignore **`schedule`** at first.

```
bool canAllPatientsBeSeen(const Map<string, int>& doctors,
                          const Map<string, int>& patients,
                          Map<string, Set<string>>& schedule);
```

Part 2: Doctors Without Orders

- Some implementation thoughts:
  - Start simple! We can ignore **`schedule`** at first.
  - There are no known greedy approach that works efficiently.

```
bool canAllPatientsBeSeen(const Map<string, int>& doctors,
                          const Map<string, int>& patients,
                          Map<string, Set<string>>& schedule);
```

Part 2: Doctors Without Orders

- Some implementation thoughts:
  - Start simple! We can ignore **`schedule`** at first.
  - There are no known greedy approach that works efficiently.
  - There are two possible approaches when making a “choice”:

```
bool canAllPatientsBeSeen(const Map<string, int>& doctors,
                          const Map<string, int>& patients,
                          Map<string, Set<string>>& schedule);
```

Part 2: Doctors Without Orders

- Some implementation thoughts:
  - Start simple! We can ignore **`schedule`** at first.
  - There are no known greedy approach that works efficiently.
  - There are two possible approaches when making a “choice”:
    - 1. Go one doctor at a time, deciding which subset of patients that doctor should see.

```
bool canAllPatientsBeSeen(const Map<string, int>& doctors,
                          const Map<string, int>& patients,
                          Map<string, Set<string>>& schedule);
```

Part 2: Doctors Without Orders

- Some implementation thoughts:
  - Start simple! We can ignore **`schedule`** at first.
  - There are no known greedy approach that works efficiently.
  - There are two possible approaches when making a “choice”:
    - 1. Go one doctor at a time, deciding which subset of patients that doctor should see.
    - 2. Go one patient at a time, deciding which doctor should see them.

```
bool canAllPatientsBeSeen(const Map<string, int>& doctors,
                          const Map<string, int>& patients,
                          Map<string, Set<string>>& schedule);
```

Part 2: Doctors Without Orders

- Some implementation thoughts:
  - Start simple! We can ignore **`schedule`** at first.
  - There are no known greedy approach that works efficiently.
  - There are two possible approaches when making a “choice”:
    - 1. Go one doctor at a time, deciding which subset of patients that doctor should see.
    - 2. Go one patient at a time, deciding which doctor should see them.
    - Take some time to decide which approach is better!

```
bool canAllPatientsBeSeen(const Map<string, int>& doctors,
                          const Map<string, int>& patients,
                          Map<string, Set<string>>& schedule);
```

Part 2: Doctors Without Orders

- Some implementation thoughts:
    - Start simple! We can ignore **`schedule`** at first.
    - There are no known greedy approach that works efficiently.
    - There are two possible approaches when making a “choice”:
        - 1. Go one doctor at a time, deciding which subset of patients that doctor should see.
        - 2. Go one patient at a time, deciding which doctor should see them.
        - Take some time to decide which approach is better!
    - **`map.firstKey()`** can give you a “random” key from the map.

```
bool canAllPatientsBeSeen(const Map<string, int>& doctors,
                          const Map<string, int>& patients,
                          Map<string, Set<string>>& schedule);
```

Part 2: Doctors Without Orders

- Some implementation thoughts:
  - Start simple! We can ignore **`schedule`** at first.
  - There are no known greedy approach that works efficiently.
  - There are two possible approaches when making a “choice”:
    - 1. Go one doctor at a time, deciding which subset of patients that doctor should see.
    - 2. Go one patient at a time, deciding which doctor should see them.
    - Take some time to decide which approach is better!
  - **`map.firstKey()`** can give you a “random” key from the map.
  - Not the first priority, but make sure your code is efficient (there is a stress test).

```
bool canAllPatientsBeSeen(const Map<string, int>& doctors,
                          const Map<string, int>& patients,
                          Map<string, Set<string>>& schedule);
```

Part 2: Doctors Without Orders

- Some implementation thoughts:
    - Start simple! We can ignore **`schedule`** at first.
    - There are no known greedy approach that works efficiently.
    - There are two possible approaches when making a “choice”:
        - 1. Go one doctor at a time, deciding which subset of patients that doctor should see.
        - 2. Go one patient at a time, deciding which doctor should see them.
        - Take some time to decide which approach is better!
    - **`map.firstKey()`** can give you a “random” key from the map.
    - Not the first priority, but make sure your code is efficient (there is a stress test).
        - Don’t make repeated schedules.
        - Don’t go down impossible paths (like intentionally ignoring a patient).

```
bool canAllPatientsBeSeen(const Map<string, int>& doctors,
                          const Map<string, int>& patients,
                          Map<string, Set<string>>& schedule);
```

Questions?

Part 3: Disaster Planning

- Time for our next real life application!

Part 3: Disaster Planning

- Time for our next real life application!
- A region wants to be **disaster-ready** (we will define this more formally later).

Part 3: Disaster Planning

- Time for our next real life application!
- A region wants to be **disaster-ready** (we will define this more formally later).
- A few limitations:

Part 3: Disaster Planning

- Time for our next real life application!
- A region wants to be **disaster-ready** (we will define this more formally later).
- A few limitations:
    - Limited supplies: We can't afford to stockpile **all** cities, so we can only pick a **strict subset** of the vulnerable cities to cover.

Part 3: Disaster Planning

- Time for our next real life application!
- A region wants to be **disaster-ready** (we will define this more formally later).
- A few limitations:
    - Limited supplies: We can't afford to stockpile **all** cities, so we can only pick a **strict subset** of the vulnerable cities to cover.
    - Need for proximity: A city cannot react to a disaster fast enough if the closest emergency supply is too far away.

Part 3: Disaster Planning

- In this last part of the assignment, your job is to find a way to make a region **disaster-ready**.

Part 3: Disaster Planning

- In this last part of the assignment, your job is to find a way to make a region **disaster-ready**.
- A region is represented by a set of cities, and each pair of cities can be optionally connected by a road.
    - `Map<string, Set<string>>& roadNetwork;`

Name of city

List of cities it is connect to

Part 3: Disaster Planning

- In this last part of the assignment, your job is to find a way to make a region **disaster-ready**.
- A region is represented by a set of cities, and each pair of cities can be optionally connected by a road.
  - `Map<string, Set<string>>& roadNetwork;`
- There are only a limited number of cities we can directly supply.
  - `int numCities;`

Part 3: Disaster Planning

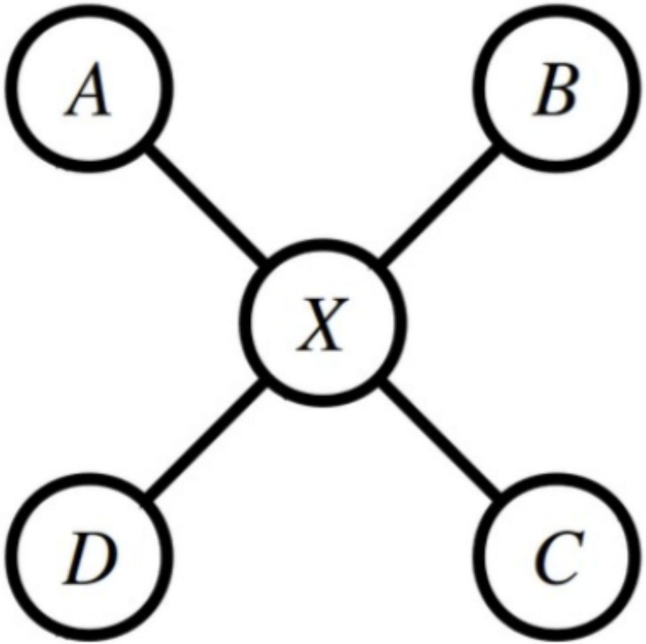
- In this last part of the assignment, your job is to find a way to make a region **disaster-ready**.
- A region is represented by a set of cities, and each pair of cities can be optionally connected by a road.
    - `Map<string, Set<string>>& roadNetwork;`
- There are only a limited number of cities we can directly supply.
    - `int numCities;`
- We would like to know, if possible, what cities should be supplied so, for every city in the region, it is either directly supplied, or adjacent to a city that is directly supplied.
    - `Set<string> supplyLocations;`

Part 3: Disaster Planning

- Let’s go over that again -- what does it mean to **cover** a city?

Part 3: Disaster Planning

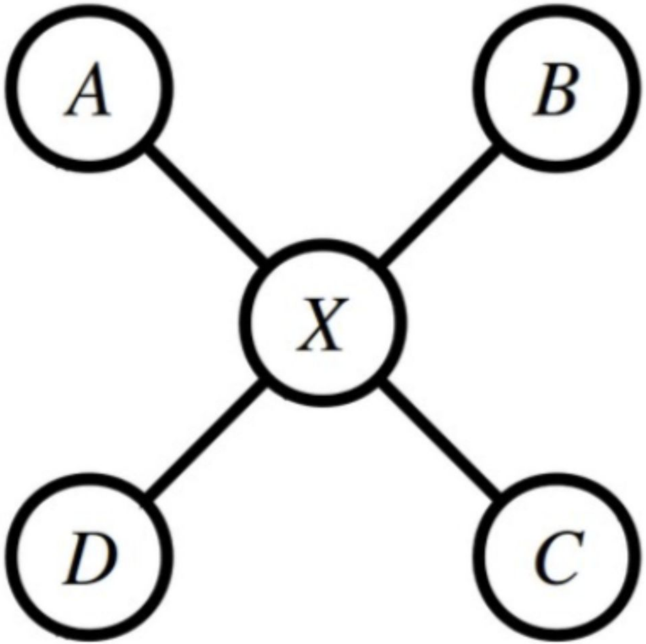
- Let's go over that again -- what does it mean to **cover** a city?



Say we have 5 Cities, A, B, X, D and C!

Part 3: Disaster Planning

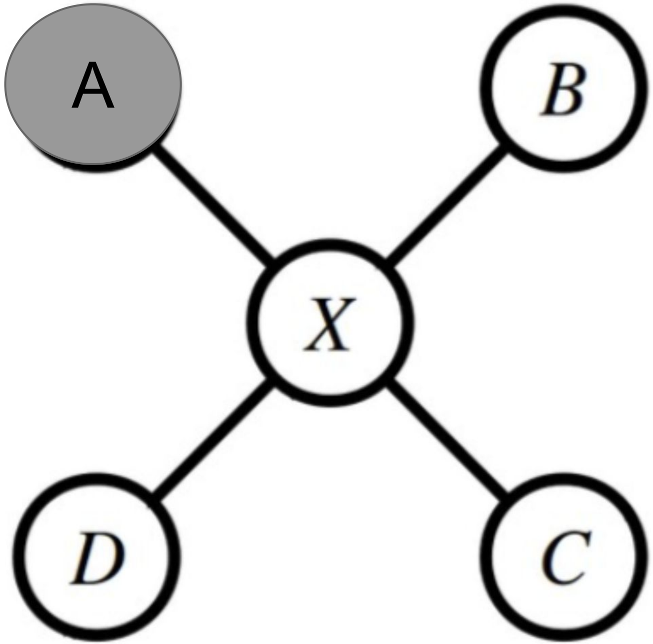
- Let's go over that again -- what does it mean to **cover** a city?



Suppose we cover city A...

Part 3: Disaster Planning

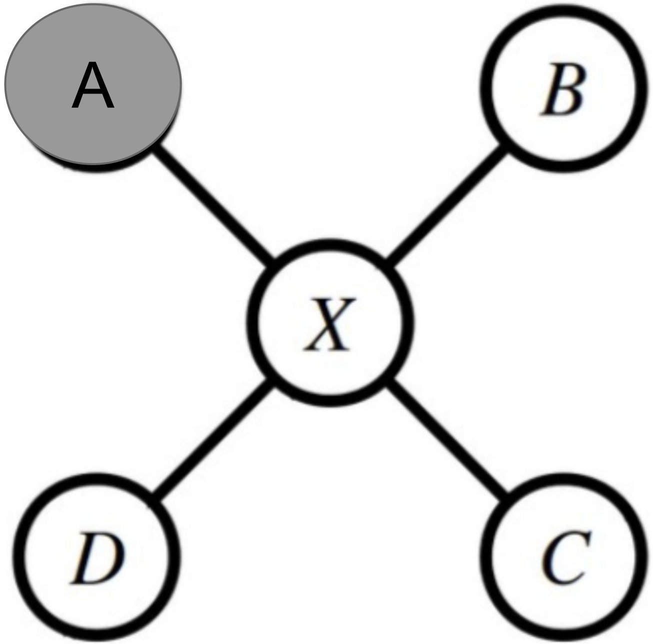
- Let's go over that again -- what does it mean to **cover** a city?



Suppose we cover city A...

Part 3: Disaster Planning

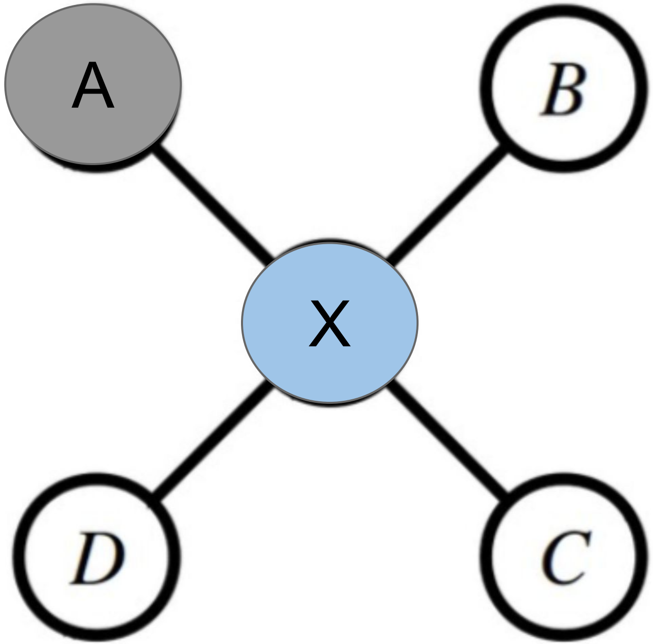
- Let's go over that again -- what does it mean to **cover** a city?



City A is now stocked up and safe, but because city A is neighbors with city X, city X is also covered due to its proximity!

Part 3: Disaster Planning

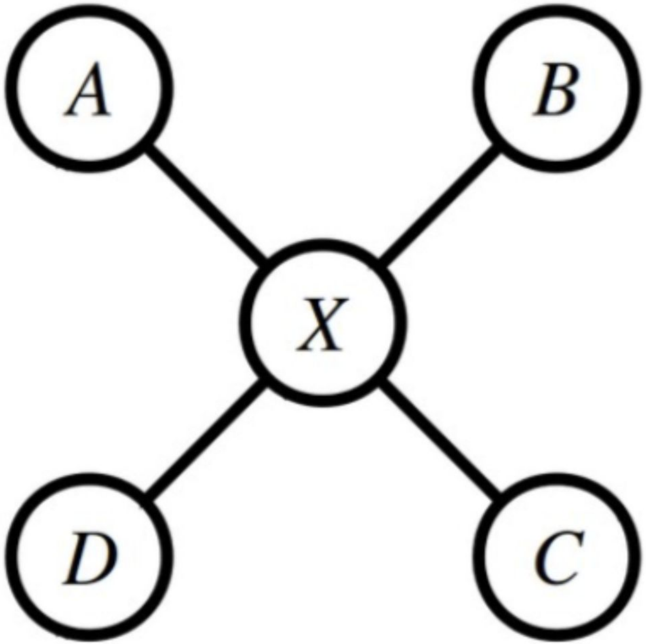
- Let's go over that again -- what does it mean to **cover** a city?



City A is now stocked up and safe, but because city A is neighbors with city X, city X is also covered due to its proximity!

Part 3: Disaster Planning

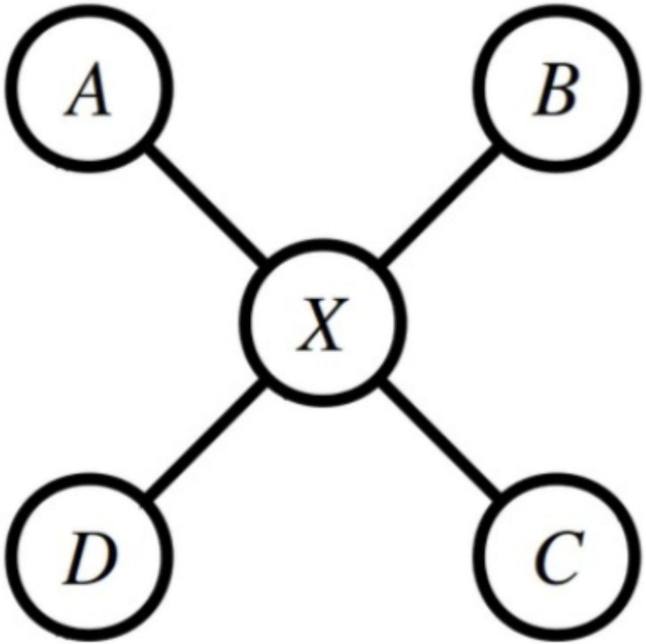
- Let's go over that again -- what does it mean to **cover** a city?



Question, what's the minimum number of cities we need to supply to "cover" all cities in this region?

Part 3: Disaster Planning

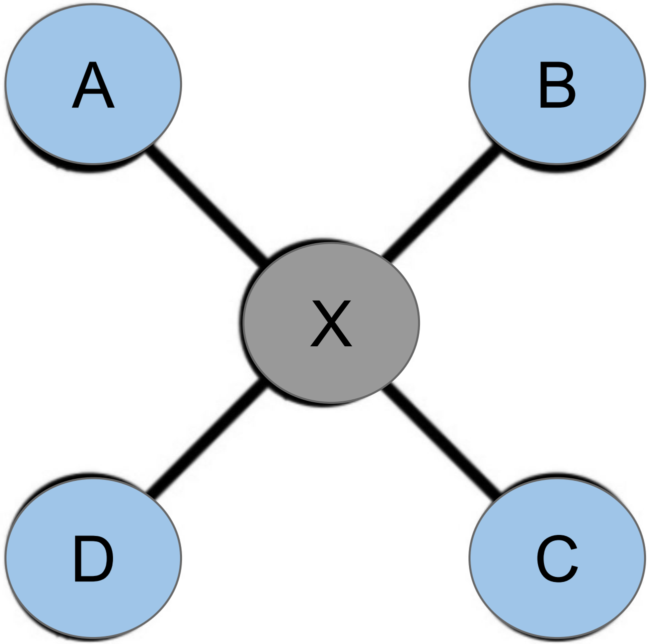
- Let's go over that again -- what does it mean to **cover** a city?



One! City X!

Part 3: Disaster Planning

- Let’s go over that again -- what does it mean to **cover** a city?



Remember that cities can be covered by their neighbors! This will come in handy in this assignment!

Part 3: Disaster Planning

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

Let’s go over the algorithm to solve this problem:

Part 3: Disaster Planning

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

Let’s go over the algorithm to solve this problem:

1. Pick a city that has not yet been covered. (hint: are you given a data structure that represents uncovered cities, or will you have to make one?)

Part 3: Disaster Planning

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

Let's go over the algorithm to solve this problem:

1. Pick a city that has not yet been covered. (hint: are you given a data structure that represents uncovered cities, or will you have to make one?)
2. For each way that it can be covered (i.e. covering the city or ANY of its neighbors), try covering that option, and then see whether the result of covering it returns **true**. If so, then covering that option was the correct choice, and you can return **true**. If all choices return false, then there's no way to cover that city, meaning that you cannot cover all cities -- you should return **false.**

Part 3: Disaster Planning

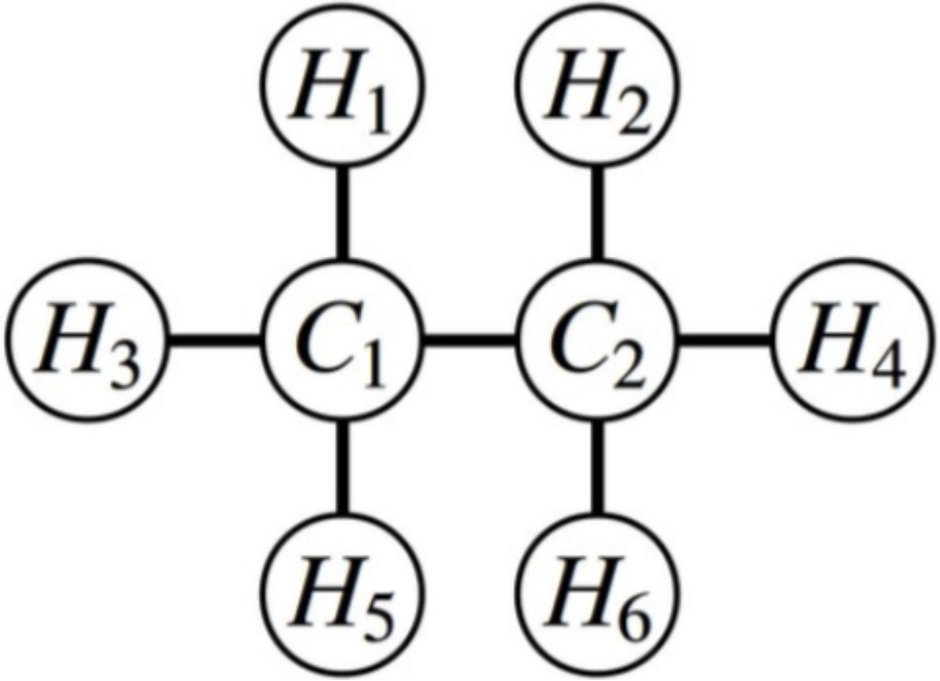
```
bool canBeMadeDisasterReady(const Map<string, Set<string>>& roadNetwork,
                            int numCities,
                            Set<string>& supplyLocations);
```

Let's go over the algorithm to solve this problem:

1. Pick a city that has not yet been covered. (hint: are you given a data structure that represents uncovered cities, or will you have to make one?)
2. For each way that it can be covered (i.e. covering the city or ANY of its neighbors), try covering that option, and then see whether the result of covering it returns **true**. If so, then covering that option was the correct choice, and you can return **true**. If all choices return false, then there's no way to cover that city, meaning that you cannot cover all cities -- you should return **false.**

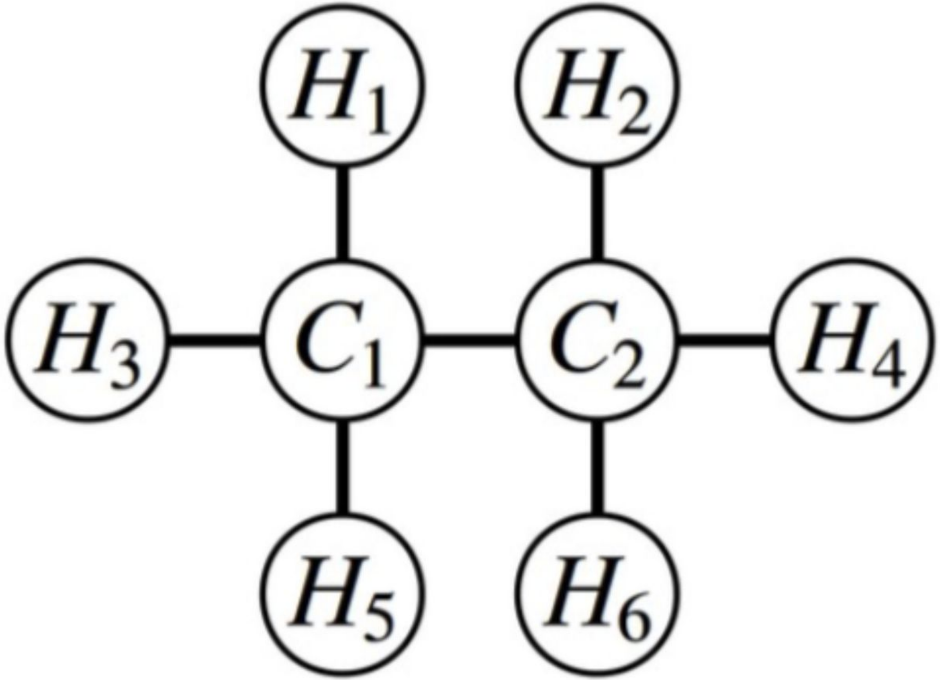
Part 3: Disaster Planning

Let's see some brief examples of this:



Part 3: Disaster Planning

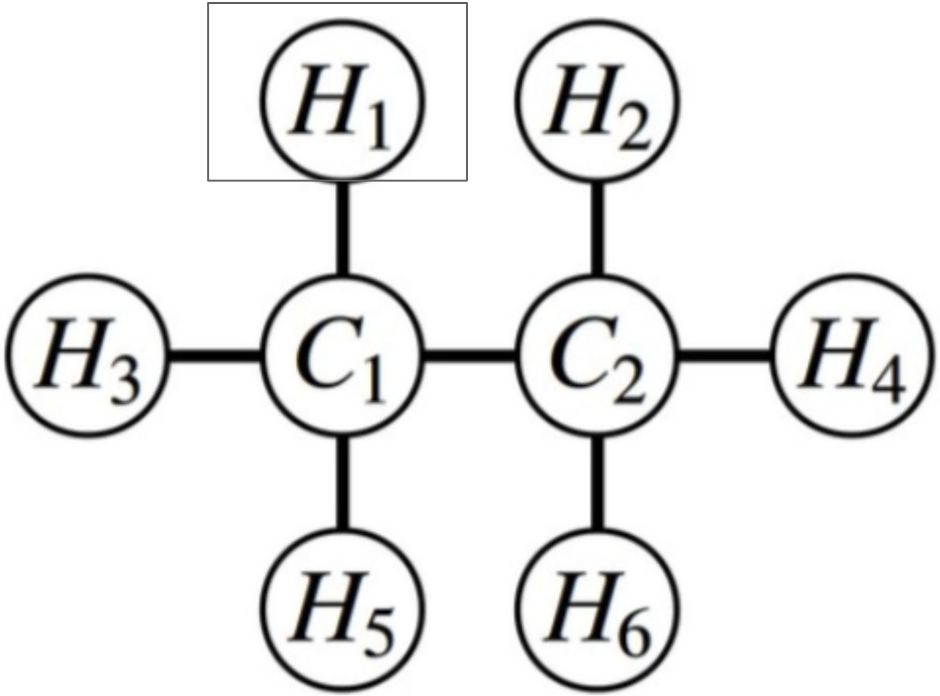
Let’s see some brief examples of this:



Let’s pick an uncovered city, say, H1

Part 3: Disaster Planning

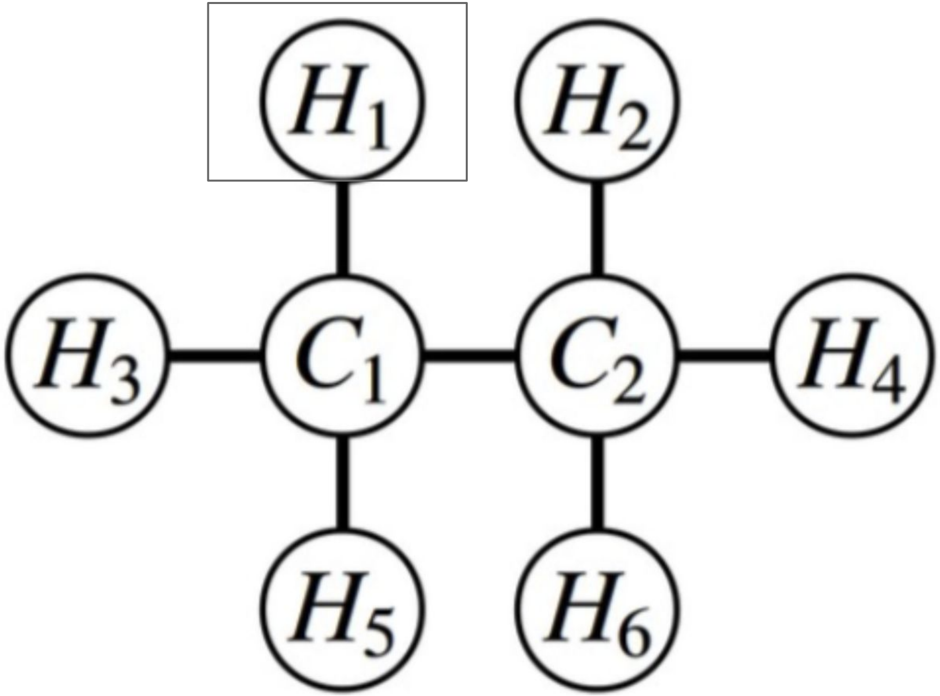
Let's see some brief examples of this:



Let's pick an uncovered city, say, H1

Part 3: Disaster Planning

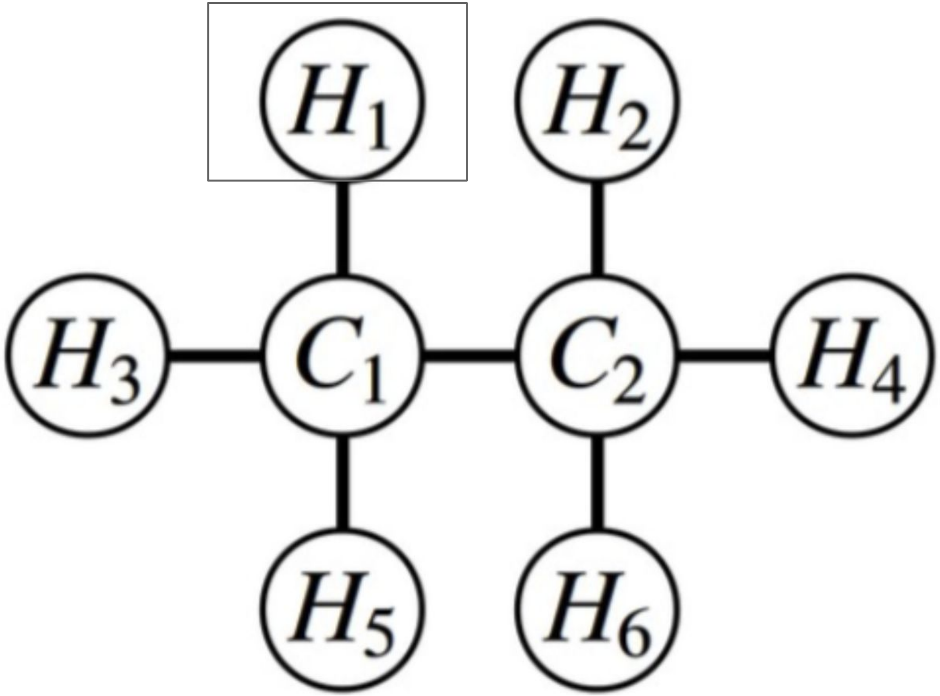
Let's see some brief examples of this:



How many ways can we cover H1?

Part 3: Disaster Planning

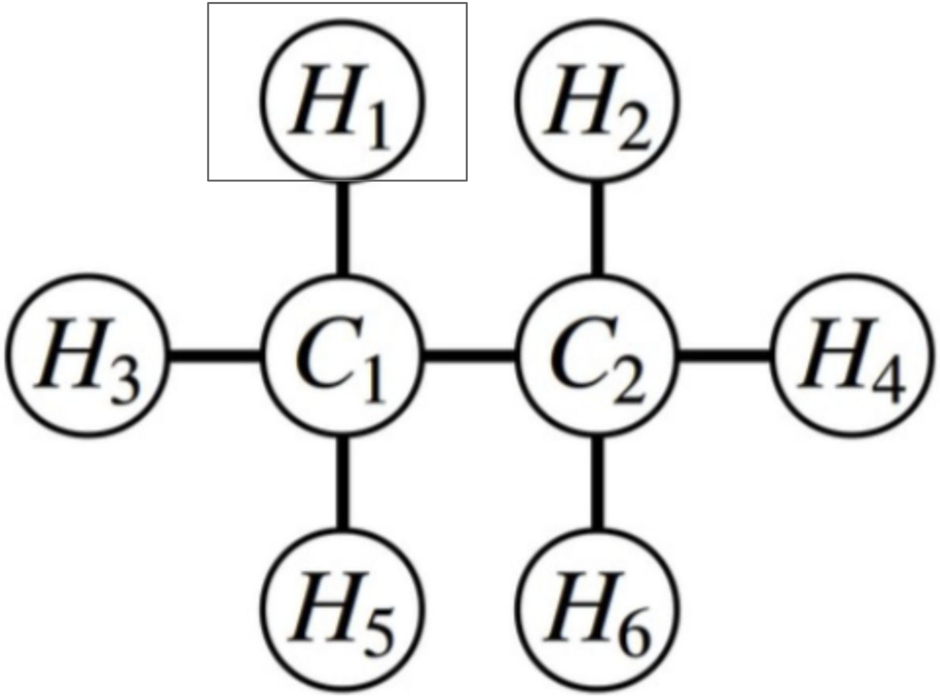
Let's see some brief examples of this:



Only 2 ways: either by covering H1 or by covering C1

Part 3: Disaster Planning

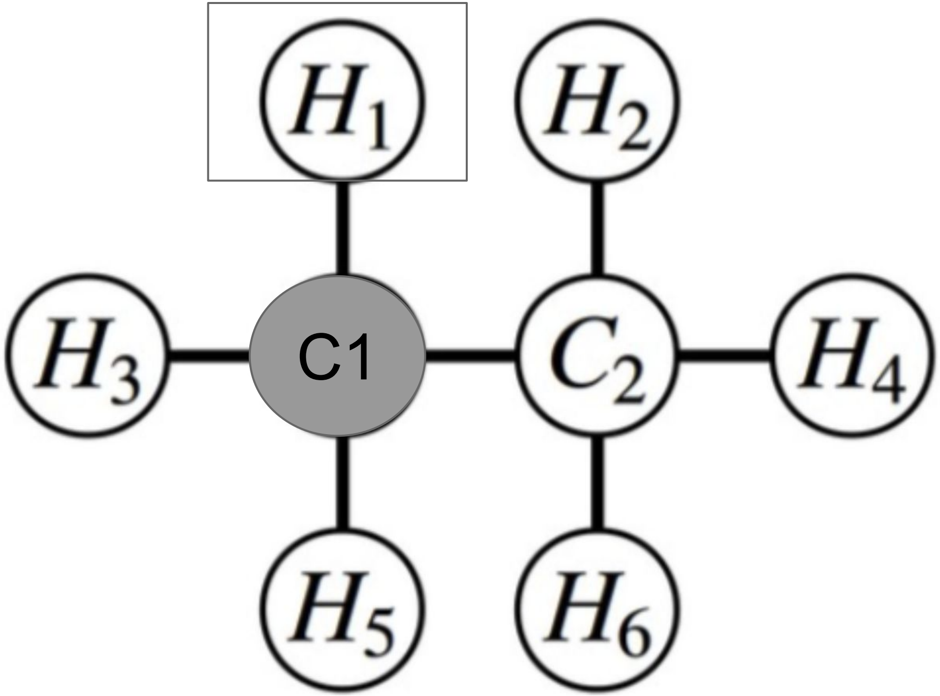
Let's see some brief examples of this:



Let's say that we first try covering C1:

Part 3: Disaster Planning

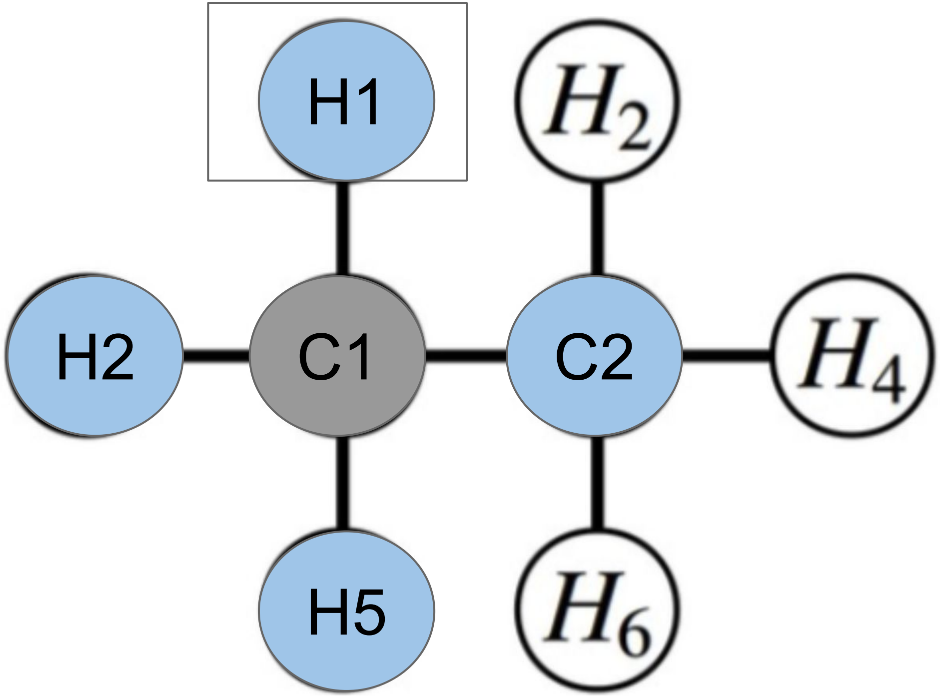
Let's see some brief examples of this:



Let's say that we first try covering C1:

Part 3: Disaster Planning

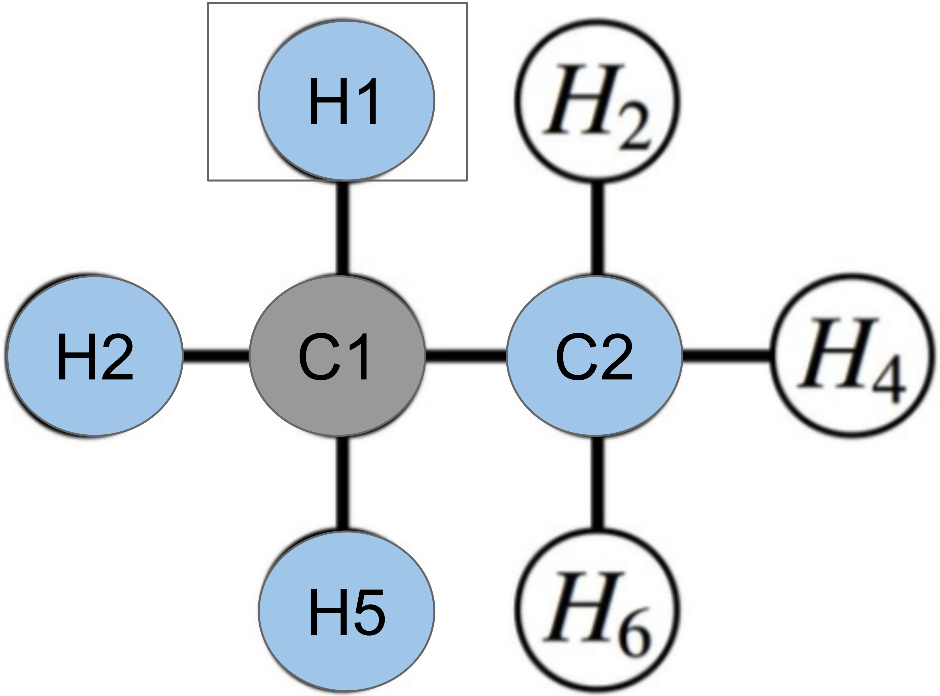
Let's see some brief examples of this:



Let's say that we first try covering C1:

Part 3: Disaster Planning

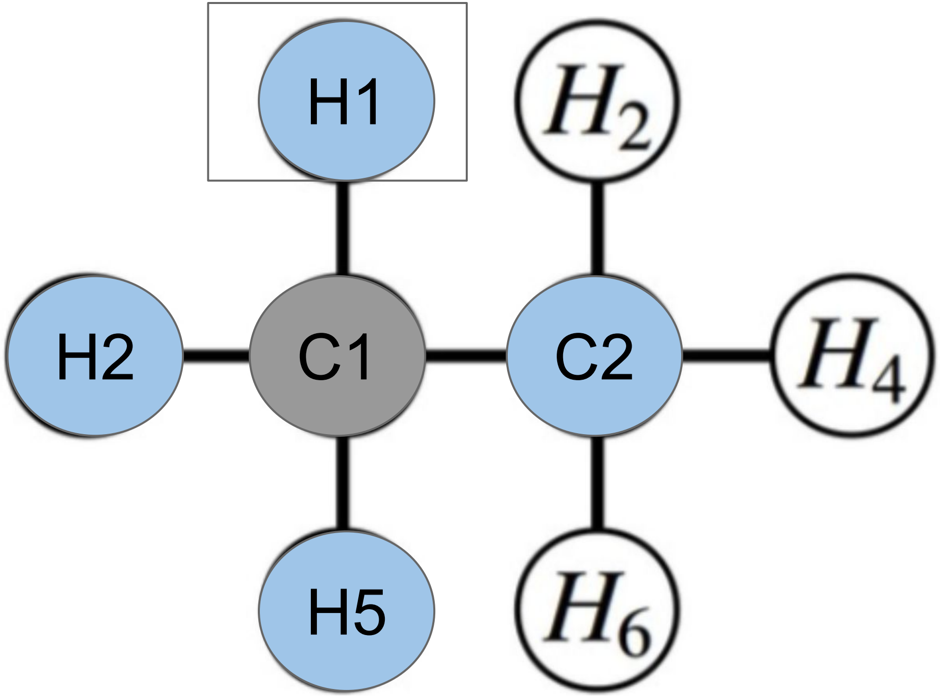
Let's see some brief examples of this:



Look at this -- by covering C1, we ended up covering **4 other locations by proximity!**

Part 3: Disaster Planning

Let's see some brief examples of this:

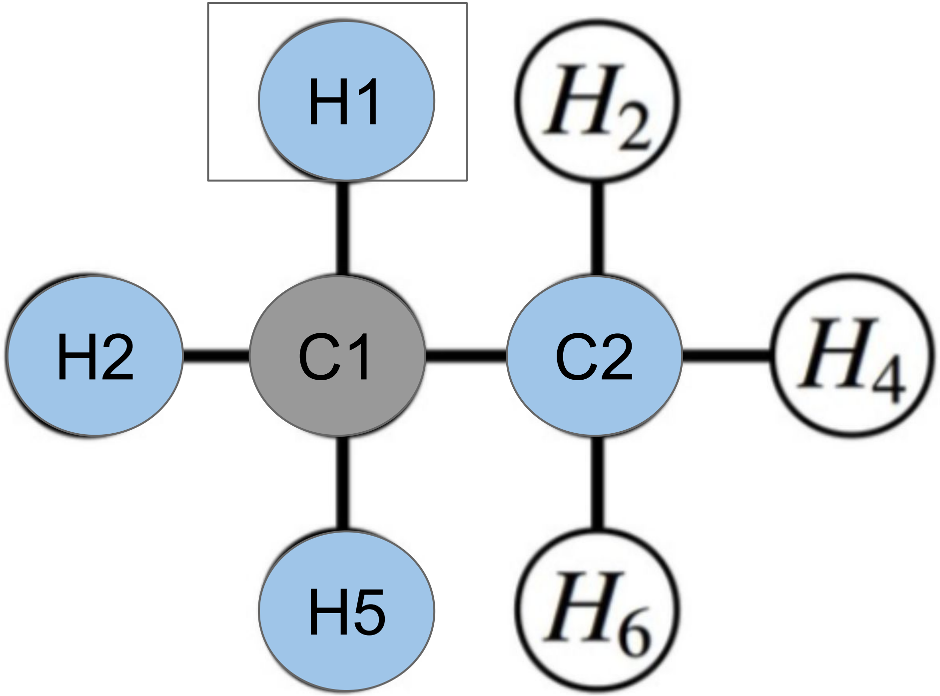


Look at this -- by covering C1, we ended up covering **4 other locations by proximity!**

In technical terms, by covering some city **C**, you can remove **roadNetwork[C]** (all of **C's** neighbors) from the set of uncovered cities, including C

Part 3: Disaster Planning

Let's see some brief examples of this:



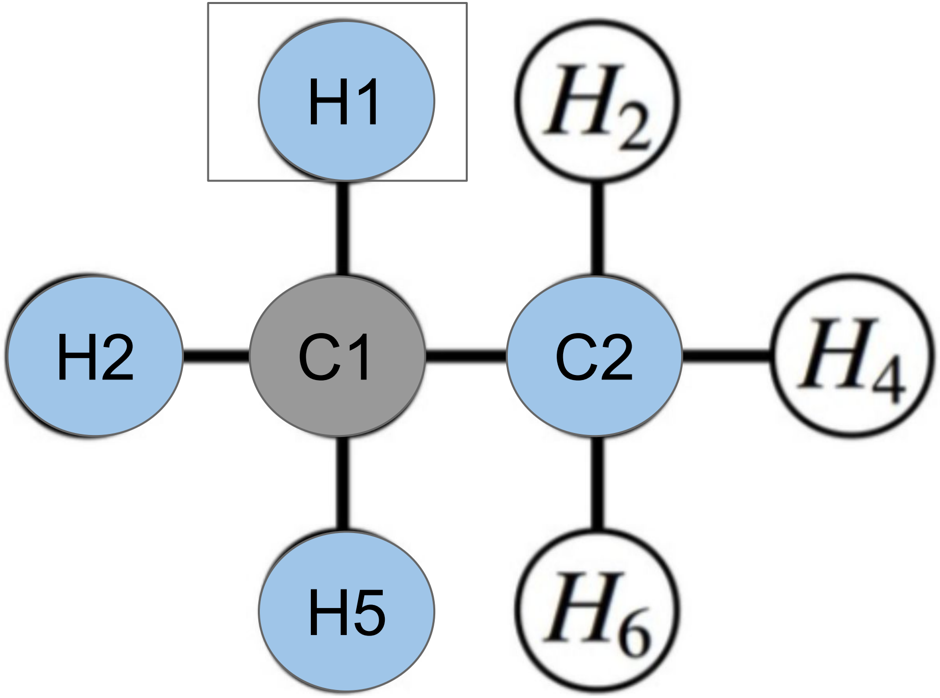
Look at this -- by covering C1, we ended up covering **4 other locations by proximity!**

In technical terms, by covering some city **C**, you can remove **roadNetwork[C]** (all of **C's** neighbors) from the set of uncovered cities, including C

Be sure to account for these changes in every recursive call!

Part 3: Disaster Planning

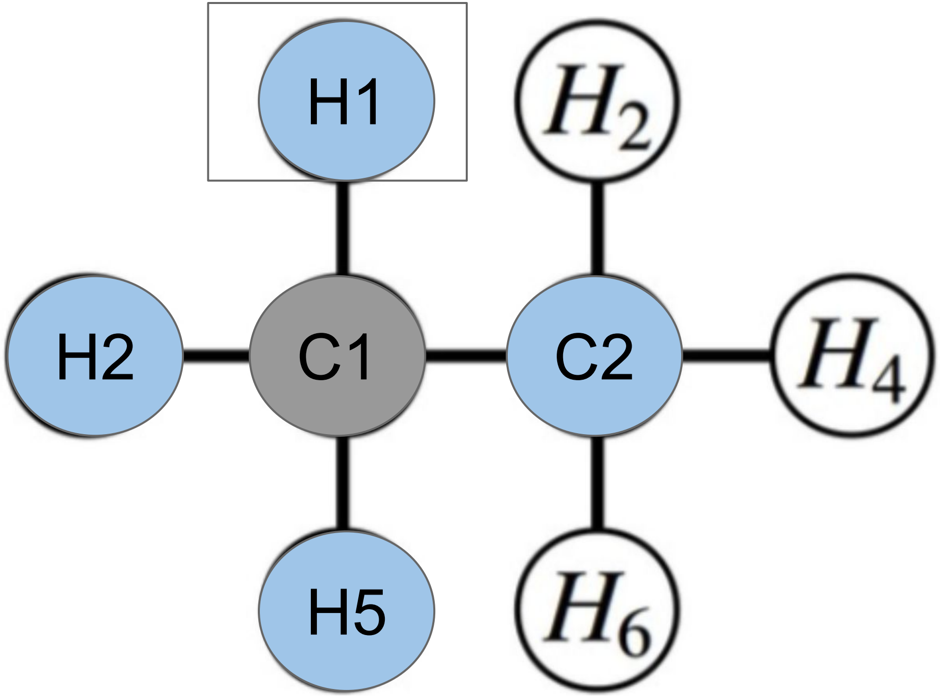
Let's see some brief examples of this:



What's the next optimal choice to cover all cities?

Part 3: Disaster Planning

Let's see some brief examples of this:

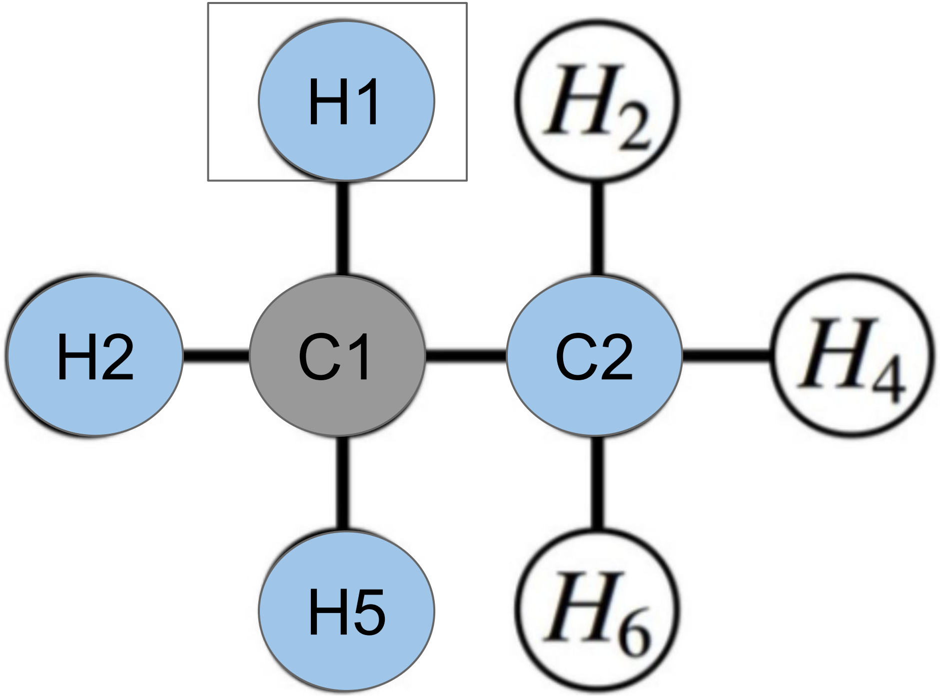


What's the next optimal choice to cover all cities?

C2! Even though it's already "covered," it is worth stockpiling the city because it will cover the last 3 uncovered cities!

Part 3: Disaster Planning

Let's see some brief examples of this:



What's the next optimal choice to cover all cities?

C2! Even though it's already "covered," it is worth stockpiling the city because it will cover the last 3 uncovered cities!

Your computer won't be able to analyze the diagrams, so it needs to try all options :p

Part 3: Disaster Planning

- Some tips/tricks:

Part 3: Disaster Planning

- Some tips/tricks:
  - The road network is bidirectional, and **`roadNetwork`** can accurately represent that.

Part 3: Disaster Planning

- Some tips/tricks:
  - The road network is bidirectional, and **`roadNetwork`** can accurately represent that.
  - **Do not** take the “greedy” approach here, which would be trying to cover cities with the most neighbors first. This won’t always work, trust us!

Part 3: Disaster Planning

- Some tips/tricks:
  - The road network is bidirectional, and **`roadNetwork`** can accurately represent that.
  - **Do not** take the “greedy” approach here, which would be trying to cover cities with the most neighbors first. This won’t always work, trust us!
  - Think about your parameters, **`roadNetwork`**, **`numCities`**, and **`supplyLocations`**. How do these change for any given recursive call? How will you keep track of cities that are covered / uncovered?

Part 3: Disaster Planning

- Some tips/tricks:
  - The road network is bidirectional, and **`roadNetwork`** can accurately represent that.
  - **Do not** take the “greedy” approach here, which would be trying to cover cities with the most neighbors first. This won’t always work, trust us!
  - Think about your parameters, **`roadNetwork`**, **`numCities`**, and **`supplyLocations`**. How do these change for any given recursive call? How will you keep track of cities that are covered / uncovered?
  - In a similar vein, **don’t modify `roadNetwork`**. We strongly encourage that you add parameters via a wrapper function, but if you change **`roadNetwork`**, all bets are off WRT your functionality.

Part 3: Disaster Planning

- Some tips/tricks:
  - The road network is bidirectional, and **`roadNetwork`** can accurately represent that.
  - **Do not** take the "greedy" approach here, which would be trying to cover cities with the most neighbors first. This won't always work, trust us!
  - Think about your parameters, **`roadNetwork`**, **`numCities`**, and **`supplyLocations`**. How do these change for any given recursive call? How will you keep track of cities that are covered / uncovered?
  - In a similar vein, **don't modify `roadNetwork`**. We strongly encourage that you add parameters via a wrapper function, but if you change **`roadNetwork`**, all bets are off WRT your functionality.
  - **`numCities`** can be zero, but you should raise an **`error()`** if it's negative.

Part 3: Disaster Planning

- Some tips/tricks:
  - The road network is bidirectional, and **`roadNetwork`** can accurately represent that.
  - **Do not** take the “greedy” approach here, which would be trying to cover cities with the most neighbors first. This won’t always work, trust us!
  - Think about your parameters, **`roadNetwork`**, **`numCities`**, and **`supplyLocations`**. How do these change for any given recursive call? How will you keep track of cities that are covered / uncovered?
  - In a similar vein, **don’t modify `roadNetwork`**. We strongly encourage that you add parameters via a wrapper function, but if you change **`roadNetwork`**, all bets are off WRT your functionality.
  - **`numCities`** can be zero, but you should raise an **`error()`** if it’s negative.
  - Keith recommends getting the return statement correct before filling the outparameter **`supplyLocations`**. Once you’ve gotten the functionality correct that determines whether a region can be supplied, *then* you can start working on the outparameter.

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

Questions?