Recursion to the Rescue!

Thanks to Nick Bowman and Eli Echt-Wilson for providing historical election data.
Thanks to Julie Zelenski, Marty Stepp, Jerry Cain, and Nick Troccoli for their input on this assignment.

Recursion is a powerful problem-solving tool with tons of practical applications. This assignment consists of three real-world recursion problems, each of which we think is interesting in its own right. By the time you're done with this assignment, we think you'll have a much deeper appreciation both for recursive problem solving and for what sorts of areas you can apply your newfound skills to.

There are three problems in this assignment.

- ***Doctors Without Orders:*** Doctors have limited time. Patients are waiting for help. Can everyone be seen?
- ***Disaster Planning:*** Cities need to plan for natural disasters. Emergency supplies are expensive. What's the cheapest way to prepare for an emergency?
- ***Winning the Presidency:*** US Presidential Elections are decided not by a popular vote, but by the Electoral College. How few popular votes can you win and still get elected?

Problem One: Doctors Without Orders

The small country of Recursia faces a crisis – no one has told the Recursian doctors which patients they're supposed to see. They're Doctors Without Orders! As Minister of Health, it's time to help the Recursians with their medical needs.

Each doctor has a number of hours that they're capable of working in a day, and each patient has a number of hours that they need to be seen for. The question then arises: is it possible for every patient to be seen by a doctor for the appropriate number of hours, and to do so without exceeding the amount of time each doctor has available?

Your task is to write a function

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

that takes as input a group of doctors and a group of patients, then returns whether it's possible to schedule all the patients so that each one is seen by a doctor for the appropriate amount of time. Each patient must be seen by a single doctor, so, for example, a patient who needs five hours of time can't be seen by five doctors for one hour each. If it is possible to schedule everyone, the function should fill in the final `schedule` parameter by associating each doctor's name (as a key) with the set of the names of patients she should see (the value).

The `doctors` parameter is map from the names of the doctors how many hours each doctor has free in a day. The `patients` map associates the names of patients with how many hours they need to be seen for.

For example, suppose we have these doctors and these patients:

- Doctor Thomas: 10 Hours Free
- Doctor Taussig: 8 Hours Free
- Doctor Sacks: 8 Hours Free
- Doctor Ofri: 8 Hours Free

- Patient Lacks: 2 Hours Needed
- Patient Gage: 3 Hours Needed
- Patient Molaison: 4 Hours Needed
- Patient Writebol: 3 Hours Needed
- Patient St. Martin: 1 Hour Needed
- Patient Washkansky: 6 Hours Needed
- Patient Sandoval: 8 Hours Needed
- Patient Giese: 6 Hours Needed

In this case, everyone can be seen:

- Doctor Thomas (10 hours free) sees Patients Molaison, Gage, and Writebol (10 hours total)
- Doctor Taussig (8 hours free) sees Patients Lacks and Washkansky (8 hours total)
- Doctor Sacks (8 hours free) sees Patients Giese and St. Martin (7 hours total)
- Doctor Ofri (8 hours free) sees Patient Sandoval (8 hours total)

However, minor changes to the patient requirements can completely invalidate this schedule. For example, if Patient Lacks needed to be seen for three hours rather than two, then there is no way to schedule all the patients so that they can be seen. On the other hand, if Patient Washkansky needed to be seen for seven hours instead of six, then there would indeed a way to schedule everyone. (Do you see how?)

The main challenge in solving this problem is coming up with the right recursive strategy. When generating subsets, the question we ask is "do we want to include or exclude this element?" When generating permutations, the question we ask is "which element do we want to pick next?" Now, think about what you need to do for this assignment. What question should you ask at each level of the recursion?

There are two general strategies you can use to solve this problem. One of them is to go one doctor at a time, deciding which subset of patients that doctor should see. Another is go to one patient at a time, deciding which doctor should see her. One of these strategies, in our opinion, is *much* easier than the other. Take a few minutes to think through which approach might be easier before starting to code anything up.

Here's what you need to do:

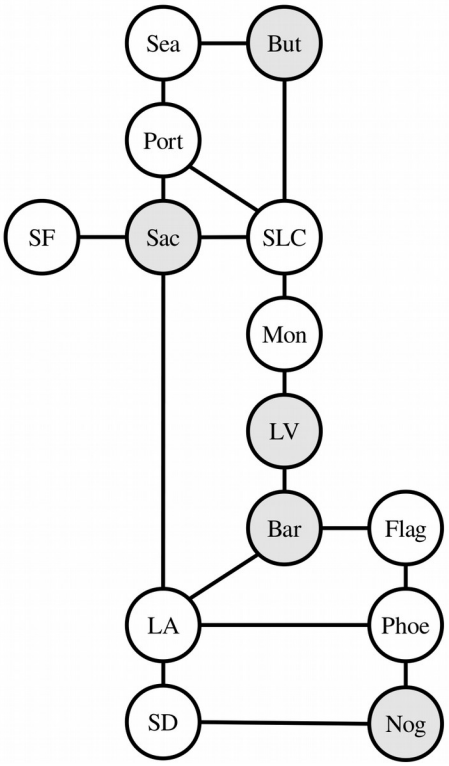
1. Add at least one custom test case to `DoctorsWithoutOrders.cpp`. This is a great way to confirm that you understand what the function you'll be writing is supposed to do.
2. Implement the `canAllPatientsBeSeen` function in `DoctorsWithoutOrders.cpp`. You should determine whether there is a schedule in which every patient is scheduled and no doctor needs to work more hours than they have available. If so, you should fill in the `schedule` outparameter with one such schedule.
3. Test your code thoroughly. Once you're confident that it works – and no sooner – pull up our bundled demo application and see what sorts of schedules your program produces!

Some notes on this problem:

- You may find it easier to solve this problem first by simply getting the return value right, completely ignoring `schedule`. Once you're sure that your code is always producing the right answer, update it so that you actually fill in the schedule. Doing so shouldn't require too much code, and it's way easier to add this in at the end than it is to debug the whole thing all at once.
- You can assume that `schedule` is empty when the function is called.
- If your function returns false, the final contents of the schedule don't matter, though we suspect your code will probably leave it blank.
- If you need to grab a key out of a `HashMap` and don't care which key you get, use the function *map*`.front()`.
- Although the parameters to this function are passed by `const` reference, you're free to make extra copies of the arguments or to set up whatever auxiliary data structures you'd like. If you find you're "fighting" your code – an operation that seems simple is taking a lot of lines – it might mean that you need to change your data structures.
- You can assume no two doctors have the same name and no two patients have the same name.
- If there's a doctor who doesn't end up seeing any patients, you can either include the doctor's name as a key in the schedule associated with an empty set of patients or leave the doctor out entirely, whichever you'd prefer.
- You might be tempted to solve this problem by repeatedly taking the patient requiring the most time and assigning them to the doctor with the most available hours, or by taking the doctor with the least time and giving them the patients requiring the fewest hours, or something like this. Solutions like these are called ***greedy algorithms***, and while greedy algorithms do work well for some problems, this problem is not one of them. In fact, there are no known ways to solve this problem efficiently using greedy algorithms!

Problem Two: Disaster Planning

Disasters – natural and unnatural – are inevitable, and cities need to be prepared to respond to them. 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 supplies, 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 and know which cities are 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, Nogales, Las Vegas, and Barstow (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 to 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 adjacent.

We'll say that a country is ***disaster-ready*** if 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 HashMap<string, HashSet<string>>& roadNetwork,
                            int numCities,
                            HashSet<string>& supplyLocations);
```

that takes as input a `HashMap` representing the road network for a region (described below) and the number of cities you can afford to put supplies in, then returns whether it's possible to make the region disaster-ready without placing supplies in more than `numCities` cities. If so, the function should then populate the argument `supplyLocations` with all of the cities where supplies should be stored.

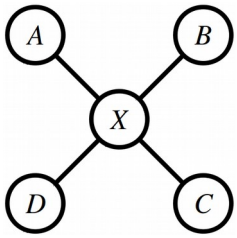
In this problem, the road network is represented as a map where each key is a city and each value is a set of cities that are immediately down the highway from them. For example, here's a fragment of the map you'd get from the above transportation network:

```
"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 `supplyLocations` is empty when this function is first called, and you can change it however you'd like if the function returns false.

You might be tempted to solve this problem by approaching it as a combinations problem. We need to choose some group of cities, and there's a limit to how many we can pick, so we could just list all combinations of `numCities` cities and see if any of them provide coverage to the entire network. The problem with this approach is that as the number of cities rises, the number of possible combinations can get way out of hand. For example, in a network with 35 cities, there are 3,247,943,160 possible combinations of 15 cities to choose from. Searching over all of those options can take a very, *very* long time, and if you were to approach this problem this way, you'd likely find your program grinding to a crawl on many transportation grids.

To speed things up, we'll need to be a bit more clever about how we approach this problem. There's a specific insight we'd like you to use that focuses the recursive search more intelligently and, therefore, reduces the overall search time.



Here's the idea. Suppose you pick some city that currently does not have disaster coverage. You're ultimately going to need to provide disaster coverage to that city, and there are only two possible ways to do it: you could stockpile supplies in that city itself, or you can stockpile supplies in one of its neighbors. For example, suppose city *X* shown to the right isn't yet covered, and we want to provide coverage to it. To do so, we'd have to put supplies in either *X* itself or in one of *A*, *B*, *C*, or *D*. If we don't put supplies it at least one of these cities, there's no way *X* will be covered.

With that in mind, ***use the following strategy to solve this problem***. Pick an uncovered city, then try out each possible way of supplying that city (either by stockpiling in that city itself or by stockpiling in a neighboring city). If after committing to any of those decisions you're then able to cover all the remaining cities, fantastic! You're done. If, however, none of those decisions ultimately leads to total coverage, then there's no way to supply all the cities.

In summary, here's what you need to do:

1. Add at least one custom test case to `DisasterPlanning.cpp`. This is a great way to confirm that you understand what the function you'll be writing is supposed to do.
2. Implement the `canBeMadeDisasterReady` function in `DisasterPlanning.cpp` using the recursive strategy outlined above. Specifically, do the following:
   - Choose a city that hasn't yet been covered.
   - For each way it could be covered – either by stockpiling supplies in that city or by stockpiling in one of its neighbors – try providing coverage that way. If you can then (recursively) cover all cities having made that choice, great! If not, that option didn't work, so you should pick another one.

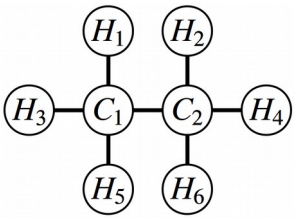
   If `numCities` is negative, your code should use the `error()` function to report an error.
3. Test your code thoroughly using our provided test driver. Once you're certain your code works – and no sooner – run the demo app to see your code in action. (More on that later.)

Some notes on this problem:

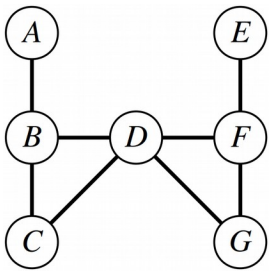
- We recommend proceeding in two steps. First, just focus on getting the return value right – that is, write a function that answers the question "is it possible to cover everything with only this many cities having supplies?" and which ignores the outparameter. Once that's working – and no sooner – edit the code to then fill in the outparameter with which cities should be chosen.
- 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*. 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 adjacent 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.
- Feel free to use *set*`.front()` or *map*`.front()` to get a single element or key from a `HashSet` or `HashMap`, respectively.
- The `numCities` parameter denotes the maximum number of cities you're allowed to stockpile in. It's okay if you use fewer than `numCities` cities to cover everything, but you can't use more.
- The `numCities` parameter may be zero, but should not be negative. If it is negative, call `error()`.

(Continued on the next page...)

- Get out a pencil and paper when debugging this one and draw pictures that show what your code is doing as it runs. Step through your code in the debugger to see what your recursion is doing. Make sure that the execution of the code mirrors the high-level algorithm described above. Can you see your code picking an uncovered city? Can you see it trying out all ways of providing coverage to that city?
- Make sure you're correctly able to tell which cities are and are not covered at each point in time. One of the most common mistakes we've seen people make in solving this problem is to accidentally mark a city as uncovered that actually is covered, usually when backtracking. Use the debugger to inspect which cities are and are not covered at each point in time.
- There are cases where the best way to cover an uncovered city is to stockpile in a city that's already covered. In the example shown to the right, which is modeled after the molecular structure of ethane, the best way to provide coverage to all cities is to pick the two central cities C_1 and C_2, even though after choosing C_1 you'll find that C_2 is already covered by C_1.



- You might be tempted to solve this problem by repeatedly taking the city adjacent to the greatest number of uncovered cities and then stockpiling there, repeating until all cities are covered. Surprisingly, this approach will not always work. In the example shown to the right here, which we've entitled "Don't be Greedy," the optimal solution is to stockpile in cities B and F. If, on the other hand, you begin by grabbing city D, which would provide coverage to five of the seven cities, you will need to stockpile in at least two more cities (one of A and B, and one of E and F) to provide coverage to everyone. If you follow the recursive strategy outlined above, you won't need to worry about this, since that solution won't always grab the city with the greatest number of neighbors first.



Once you're sure that your code works – and no sooner – choose the "Disaster Planning" option from the main menu. The bundled demo will let you run your code out on some realistic data sets. It makes multiple calls to your recursive function to find the *minimum* number of cities needed to provide coverage. Play around with the sample transportation grids provided – find anything interesting?

A note: some of the sample files that we've included have a *lot* of cities in them. The samples whose names start with `VeryHard` are, unsurprisingly, very hard tests that may require some time for your code to solve. It's okay if your program takes a long time (say, at most two minutes) to answer queries for those transportation grids, though the other samples shouldn't take very long to complete.

Problem Three: Winning the Presidency

The President of the United States is not elected by a popular vote, but by a majority vote in the Electoral College. Each state, plus DC, gets some number of electors in the Electoral College, and whoever they vote in becomes the next President. For the purposes of this problem, we're going to make some simplifying assumptions:

- You need to win a majority of the votes in a state to earn its electors, and you get all the state's electors if you win the majority of the votes. For example, in a small state with 999,999 people, you'd need 500,000 votes to win all its electors. These assumptions aren't entirely accurate, both because in most states a plurality suffices and some states split their electoral votes in other ways.
- You need to win a majority of the electoral votes to become president. In the 2008 election, you'd need 270 votes because there were 538 electors. In the 1804 election, you'd need 89 votes because there were only 176 electors. (You can technically win the presidency without winning the Electoral College; we'll ignore this for simplicity.)
- Electors never defect. The electors in the Electoral College are free to vote for whomever they please, but the expectation is that they'll vote for the candidate that won their home state. As a simplifying assumption, we'll just pretend electors always vote with the majority of their state.

This problem explores the following question: under these assumptions, what's the fewest number of popular votes you can get and still be elected President?

Your task is to write a function

```
MinInfo minPopularVoteToWin(const Vector<State>& states);
```

that takes as input a list of all the states that participated in the election (plus DC, if applicable), then returns some information about the minimum number of popular votes you'd need in order to win the election (namely, how many votes you'd need, and which states you'd carry in the process).

Here's a quick overview of the types involved here. First, there's the `State` type, defined here:

```
struct State {
    string name;        // The name of the state
    int electoralVotes; // How many electors it has
    int popularVotes;   // The number of people in that state who voted
};
```

The input to `minPopularVoteToWin` is a `Vector<State>` containing information about all the states that participated in the election. The `minPopularVoteToWin` function then returns a `MinInfo`, a type that contains information about the minimum popular vote needed and which states you'd carry:

```
struct MinInfo {
    int popularVotesNeeded;   // How many popular votes you'd need.
    Vector<State> statesUsed; // The states you'd win in getting those votes.
};
```

This particular recursive function requires a little creativity to get working quickly, so rather than having you dive headfirst into it, we've broken this task down into three smaller pieces that build into the finished product. Our suggested solution route for this problem looks like this:

- *Step One:* Implement a recursive helper function that solves a slightly more general version of this problem.
- *Step Two:* Implement `minPopularVoteToWin` as a wrapper around that function, getting a preliminary implementation that's good for small inputs but too slow for full elections.
- *Step Three:* Add in memoization to make your program run lightning fast. Then, go play around with the real election data bundled with the starter files to see everything in action!

The rest of this section describes these steps in detail.

Step One: Solve a More General Problem

Your first task is to implement this recursive helper function:

```
MinInfo minPopularVoteToGetAtLeast(int electoralVotes,
                                   int startIndex,
                                   const Vector<State>& states);
```

This function should solve the following problem:

How few popular votes are needed to get at least `electoralVotes` electoral votes, using only states from index `startIndex` and greater in a `Vector` of all states?

How is this a generalization of the original problem? Well, in the original problem, you need to specifically get a majority of the electoral votes, and here you need to hit some arbitrary target number of votes. In the original problem, you can use any of the states, and in this modified problem, you can only use states from the specified index and forward.

You must not make this a wrapper function, and ***you must not add any additional parameters***. Later on, you're going to revisit this function and add in memoization, and that will not work correctly unless this function is implemented recursively and only uses the specified parameters.

You might find yourself in the unfortunate situation where, for some specific values of `electoralVotes` and `startIndex`, there's no possible way to get `electoralVotes` votes using only the states from index `startIndex` and forward. For example, if you're working with real-world data, are short 75 electoral votes, and only have a single state left, there's nothing that you can do to win the election. In that case, you may want to have this helper function return a `MinInfo` whose `popularVotesNeeded` field is set to sentinel value indicating that it's not possible to secure that many votes. We've provided you with a constant `kNotPossible` that you can use to indicate this; we chose `kNotPossible` to be something that's so big that it will never be the correct answer. The advantage of this sentinel is that you're already planning on finding the strategy that requires the fewest popular votes, so if your sentinel value is greater than any possible legal number of votes, always choosing the option that requires the minimum number of votes will automatically pull you away from the sentinel value.

To summarize, here's what you need to do.

1. Add at least one custom test case to `WinningThePresidency.cpp` to specifically test your `minPopularVoteToGetAtLeast` function.
2. Implement the `minPopularVoteToGetAtLeast` function in `WinningThePresidency.cpp` using the recursive strategy outlined above. As a reminder, you must not add any parameters to this function, nor should you make it a wrapper.

Some notes on this problem:

- In lecture, you've seen a number of structures you can explore recursively (subsets, permutations, and combinations). Which of those options does this most closely resemble? Based on that, what sort of recursive approach might you want to take here?
- Implement this function in two stages. First, completely ignore the `statesUsed` field in the `MinInfo` you return and just fill in the `popularVotesNeeded` field. Once you're giving back the correct number of popular votes, update your function to fill in which states you end up using.
- You need ***strictly more*** than half the popular votes to win all the electors from a state. For example, if there are 100,000 or 100,001 people in a state, you need 50,001 votes to win its electors.
- This is probably the toughest of the three steps in this problem, so don't get discouraged if it seems a bit tricky. You've taken on a number of tough recursion problems before and you are absolutely capable of handling this challenge. Use the skills you've learned along the way: craft strategic test cases to stress-test unusual scenarios, pull up the debugger if need be, add in some `cout` statements to have the code tell you what it's doing, etc. *You can do this!*

Step Two: Implement a Preliminary Solution

Now that you have `minPopularVoteToGetAtLeast` implemented, go and implement the top-level function `minPopularVoteToWin` by calling `minPopularVoteToGetAtLeast` with the right set of arguments. You shouldn't need to write much code for this – this is more about thinking through how the function you wrote in the first part works than it is about crazy complex recursion.

Specifically, do the following:

1. Add at least one custom test case to `WinningThePresidency.cpp` to specifically test your `minPopularVoteToWin` function.
2. Implement the `minPopularVoteToWin` function in `WinningThePresidency.cpp`.

Some things to think about:

- The historical election data – and our sample test cases – do not always include all 50 US states plus DC, either because those states didn't exist yet, or DC didn't have the vote, or because those states didn't participate in the election, so you shouldn't assume that you'll get them as input.
- You need to get strictly more than half the possible electoral votes to get elected President. If, as in 2016, there are 538 electors, then you'd need to get 270 votes to win. If, as in 1788, there are 69 electors, then you'd need 35 votes to get elected. In other words, don't assume you specifically need to get to 270; determine the total based on the collection of states provided as input.

Be sure to ***test your solution*** before moving on. You should find that for smaller inputs (say, 15 or fewer states) that your code is relatively fast, but that for larger inputs things start to slow down to a crawl. If you try running what you have on real elections data with all fifty states plus DC, chances are that your code won't even produce an answer. That's okay for now; you'll speed things up in the next step. But do make sure the answers that you do produce are correct; it's not going to matter that you end up with fast code if the information you generate has no relationship with reality. 😄

Step Three: Add Memoization

At this point, you have an implementation of `minPopularVotesToWin` that technically works correctly, but which is going to be way too slow to work on real elections data. As an example, all fifty states, plus the District of Columbia, participated in the 2016 election. That means that there are 2^{51} possible subsets to explore when trying to find the optimal combination. Unfortunately, 2^{51} is 2,251,799,813,685,248, which is such a huge number that even checking a billion combinations each second would take about a month to process.

On the other hand, look back at the recursive helper function you implemented in Step One. It takes as input a number of electoral votes to reach and an index into the `Vector`. In the 2016 election, that number of electoral votes is 270, and there are only 51 states (counting DC as a state). That means there are only $271 \times 52 = 14{,}092$ different inputs possible for this helper function (the number of electoral votes ranges from 0 to 270, inclusive, and the start index can range from 0 to 51, inclusive). If you were to memoize the results and only compute each value once, it should take the computer under a second to crunch through all of them. That's a *dramatic* improvement!

As your last task, update the `minPopularVoteToGetAtLeast` function to use memoization. There are many ways you can do this, and we'll leave those decisions up to you. Adding memoization will change the signature of this function, and that's completely okay at this stage of the assignment. This may cause some of your older tests to no longer compile; you'll need to make appropriate edits to them.

Once you've gotten this working, try running your code on full elections data. Isn't that amazing?

To summarize, here's what you need to do:

1. Edit your `minPopularVoteToGetAtLeast` function to support memoization. In doing so:
    1. You'll need to add a new parameter, or several new parameters, to the function.
    2. You'll need to edit your test cases from before and minPopularVoteToWin to pass in appropriately-constructed arguments to that function.
    3. You'll need to fill in the memoization table to avoid recomputing answers.
2. Test your solution thoroughly. Once you're passing all the tests – and no sooner – try out the demo app to see your code in action!

Right before the 2016 election, NPR reported that 23% of the popular vote would be sufficient to win the election, based on the 2012 voting data. They arrived at this number by looking at states with the highest ratio of electoral votes to voting population. This was a correction to their originally-reported number of 27%, which they got by looking at what it would take to win the states with the highest number of electoral votes. But the optimal strategy turns out to be neither of these and instead uses a blend of small and large states. Once you've gotten your program working, try running it on the data from the 2012 election. What percentage of the popular vote does your program say would be necessary to secure the presidency?

(A note: the historical election data here was compiled from a number of sources. In many early elections state legislatures chose electors rather than voters, so in some cases we estimated the voting population based on the overall US population at the time and the total fraction of votes cast. This may skew some of the earlier election results. However, to the best of our knowledge, data from 1868 and forward is complete and accurate. Please let us know if you find any errors in our data!)

Part Four: (Optional) Extensions!

There are tons of variations on these problems that you could imagine trying to solve and lots of ways you could apply them. Here are some suggestions:

- ***Doctors Without Orders:*** What happens if doctors have specialties (ophthalmology, physiatry, cardiology, neurology, etc.) and each patient can only be seen by a specialist of a given type?

  What happens if you want to distribute the load in a way that's as "fair" as possible, in the sense that the busiest and least busy doctors have roughly the same hourly load? What happens if you can't see everyone, but you want to see as many people as possible?

  Imagine that not everyone can be seen in one day. What multiday schedule minimizes the total number of days required to see everyone?

- ***Disaster Planning:*** There are a number of underlying assumptions in this problem. We're assuming that there will only be a disaster in a single city at a time, that the road network won't be disrupted, and that there's only a single class of emergency supplies. What happens if those assumptions are violated? For example, what if there's a major earthquake in the Cascadia Subduction Zone, striking both Portland and Seattle (with some aftereffects in Sacramento) and disrupting I-5 up north? What if you need to stockpile blankets, food, and water separately, and each city can only store one?

  You may have noticed that the `VeryHardSouthernUS` sample takes a *long* time to solve, and that's because while the approach we've suggested for solving this problem is much better than literally trying all combinations of cities, it still has room for improvement. See if you can speed things up! Here's a simple idea to get you started: what if you choose which uncovered city to cover by selecting the city with the fewest neighbors? Those are the hardest cities to cover, so handling them first can really improve performance.

  Are there any other maps worth exploring? Feel free to create and submit a map of your own!

- ***Winning the Presidency:*** If you look at historical election data, you'll see that it's not all that uncommon for a third-party candidate to win a good number of electoral votes. The Election of 1860, for example, was a four-way race, as was the one in 1912. (The Election of 1836 was an impressive five-way race; the Whig party strategy was really interesting!) The 1992 election was the most recent one in which a major third-party candidate got a good share of the popular vote. Imagine that instead of having to win at least half of the votes in a state to win its electors, you only need to get a third, or a fourth of the votes. How does that change your results?

  In the event that there's an Electoral College tie, the choice of who becomes President gets sent to the House of Representatives. Given the historical data, how many different possible outcomes are there that result in an exact Electoral College tie?

  On the non-technical side, are there any stories you can tell based on the data you have and the results you're getting? What policy recommendations, if any, could you make from them?

Good luck, and have fun!