CS106B
Winter 2020

Handout 14
January 31, 2020

Assignment 4: 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.
Debugging exercise adapted from one developed by Julie Zelenski.

Recursion is a powerful problem-solving tool with tons of practical applications. This assignment centers on two 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. The first one is a warmup, and the next two are the main coding exercises.

- ***Debugging Practice:*** The debugger is a powerful tool for helping understand what a program is doing. Learning how to harness it is important to developing as a programmer.
- ***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?

You have plenty of time to work on this assignment if you make slow and steady progress. Here's our recommended timetable:

- Aim to complete the debugging exercises within one day of this assignment going out.
- Aim to complete Doctors Without Orders within three days of this assignment going out.
- Aim to complete Disaster Planning within seven days of this assignment going out.

Due Friday, February 7th at the start of lecture.

You are encouraged to work in pairs on this assignment.
Just don't work in pairs by splitting the assignment in half. 😃

Problem One: Debugging Practice

When you encounter a bug in a program, your immediate instinct is probably to say something like

"Why isn't my program doing what I want it to do?"

One of the best ways to answer that question is to instead answer this other one:

*"What **is** my program doing, and why is that different than what I intended?"*

The debugger is powerful tool for answering questions like these. You explored the debugger in Assignment 0 (when you learned how to set breakpoints and use Step In, Step Over, and Step Out) and in Assignment 1 (when you learned how to walk up and down the call stack). This part of the assignment is designed to refresh your skills in those areas and to give you practice working the debugger in more elaborate recursive problems.

Milestone 1: Explore the Towers of Hanoi

The ***Towers of Hanoi*** problem is a classic puzzle that has a beautiful recursive solution. If you haven't yet done so, take a few minutes to read Chapter 8.1 of the textbook, which explores this problem in depth.

We've provided you with a `TowersOfHanoi.cpp` file, which includes a *correct*, *working* solution to the Towers of Hanoi problem. Take a minute to run the "Towers of Hanoi" demo from the main program. Choose the "Go!" button to begin the animation, and marvel at how that tiny recursive function is capable of doing so much. Isn't that amazing?

With that said, let's dive into the meat of what you'll be doing. We're going to ask you to use the debugger and its features to get a better sense for how the code works.

Open the `TowersOfHanoi.cpp` file, then set a breakpoint on the first line of the `solveTowersOfHanoi` function. Run the program in debug mode, choose the "Towers of Hanoi" option from the menu at the top of the program, but ***don't click the "Go!" button yet***. When the debugger engages, it halts execution of the running program so that you can inspect what's going on. This means that the graphics window might not be operational – you might find that you can't drag it around, or resize it, or move it, etc. Therefore, we recommend that before you hit the "Go!" button to bring up the debugger, you resize the demo app window and the Qt Creator window so that they're both fully visible.

Once you're ready, hit the "Go!" button. This will trigger the breakpoint. You'll see a yellow arrow pointing at the line containing the breakpoint, and the local variables window will have popped up.

First, investigate the pane in the debugger that shows local variables and their values. Because `totalMoves` has not yet been initialized, its value is unspecified; it might be 0, or it might be a random garbage value. The function's parameters, though, should be clearly visible at this point.

You should now be able to answer the following questions. To do so, edit the file `res/DebuggingAnswers.txt` with your answers:

Q1: What are the values of all the parameters to the `solveTowersOfHanoi` function?

Q2: Some function in our starter code called `solveTowersOfHanoi`. What file was that function defined in, and what was the name of that function? (Hint: use the call stack!)

Once you've answered these questions, go back to the `TowersOfHanoi.cpp` file, and make sure you see a yellow arrow pointing at the line containing your breakpoint. Let's now single-step through the program. Use the "Step Over" button to advance past the call to the function `initHanoiDisplay`, which configures the graphics window. If you've done this correctly, you should see the disks and spindles.

Now, keeping clicking "Step Over" to advance through the other lines in the function. When you step over the line containing the call to `moveTower`, you should see the disks move to solve the Towers of Hanoi. Doesn't get old, does it? 😃

You should now be ready to answer the following question in `res/DebuggingAnswers.txt`.

> Q3: How many total moves were required to solve this instance of Towers of Hanoi?

At this point, hit the “Continue” button to let the program keep running as usual. Click the “Go!” button again to trigger your breakpoint a second time.

This time, instead of using Step Over, we’re going to use ***Step Into***. Rather than stepping over function calls, Step Into goes inside the function being called so you can step through each of its statements. (If the current line is not a function call, Step Into and Step Over do the same thing.)

Use Step Into to enter the call to `initHanoiDisplay`. The editor pane will switch to show the contents of the `src/Demos/TowersOfHanoiGUI.cpp` file and the yellow arrow will point to the first line of the `initHanoiDisplay` function. This code is unfamiliar, you didn't write it, and you didn't intend to start tracing it. Step Out is your escape hatch. This “giant step” executes the rest of the current function up to where it returns. Use Step Out to return to `solveTowersOfHanoi`.

The next line of code in `solveTowersOfHanoi` is the `pause` function, another library function you don't want to trace through. You could step in and back out, but it's simpler to just Step Over.

You are interested in tracing through the `moveTower` function, so use Step Into to go inside. Once inside, single-step through the code until the program is just about to execute the first recursive call to `moveTower`. Now, press Step Over to execute it. The GUI window will show the left tower, except for the bottom disc, moving from the left peg to the middle peg, leaving the bottom disk uncovered. This should also cause the value of `totalMoves` to count all moves made by that recursive call.

Now, answer the following question:

> Q4: What is the value of the `totalMoves` variable inside the first `moveTower` call after stepping over its first recursive sub-call? (In other words, just after stepping over the first recursive sub-call to `moveTower` inside the `if` statement in the recursive function.)

The next Step Over moves the bottom disk. The final Step Over moves the smaller tower on top. Use Continue to resume normal execution and finish the demo.

Press the “Go!” button a third time. This time, do your own tracing and exploration to solidify your understanding of recursion and its mechanics. Watch the animated disks and consider how this relates to the sequence of recursive calls. Observe how stack frames are added and removed from the debugger call stack. Select different levels on the call stack to see the value of the parameters and the nesting of recursive calls. Here are some suggestions for how stepping can help:

- 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.

Milestone 2: Debug a Broken Permutations Functions

Your next task is to use the debugger to do what it's designed for – to debug a program!

In `Permutations.cpp`, we have provided you an ***incorrect*** implementation of a function to generate permutations recursively. The `permutationsRec` function contains a small but significant error. It's not that far from working correctly – in fact, *there is a one-character mistake in that function* – but what difference a single character can make! Your task is to use the debugger to figure out the following:

- What is the one-character mistake in the program?
- With the one-character mistake in the program, what does the program actually do? And why is that not what we want it to do?

Choose "Permutations" from the top menu and you'll pull up a window that lets you type in strings, call the broken `permutationsOf` function, and see the output that's produced. Try various inputs and observe the difference between what's produced and what's supposed to be produced. (How can you see what's supposed to happen? You could always run the lecture code from Monday when we wrote a correct recursive permutations function!) It can be difficult to tease out the impact of the bug when you are tracing through a deep sequence of recursive calls. Try a variety of simple inputs to find the *smallest possible input* for which you can observe an error and use that as your test case. Specifically, you're aiming to find an input where

- the output produced is wrong, and
- no shorter input produces the wrong answer.

Using your minimized test case, trace the operation of `permutationsRec` to observe what's going on internally. Diagram the decision tree that is being traversed and match the tree to what you see in the debugger as you step in/out/over. Select different stack frames in the call stack to see the state being maintained in each of the outer frames.

Eventually, you should find the bug. Once you have, answer the following questions by editing the `res/DebuggingAnswers.txt` file.

> Q5: What is the smallest possible input that triggers the bug?
>
> Q6: What is the one-character error in the program?
>
> Q7: Explain why that one-character bug causes the function to return the exact output you see when you feed in the input you provided in Q5. You should be able to specifically account for where things go wrong and how the place where the error occurs causes the result to change from "completely correct" to "terribly wrong."

As a hint on this problem: you might have noticed that we didn't pass the parameters in by `const` reference. If you've tried hunting the bug for fifteen minutes and haven't found it yet, try changing the parameters to use pass-by-`const`-reference and see if you notice anything. That might help you *find* the bug, but to *understand* the bug you'll need to do some more exploration in the debugger.

We've asked you to answer these questions because this sort of bug-hunting is useful for understanding recursive functions and what makes them break. In particular, keep the following in mind:

- When trying to debug a recursive function, ***look for the simplest case where the recursion gives the wrong answer***. Having a small test case makes it easy to reproduce the error and to trace through what's happening in the debugger.
- Using Step In, Step Over, and Step Out, it's possible to watch recursion work at different levels of detail. Step In lets you see what's going on at each point in time. Step Over lets you see what a recursive function does in its entirety. Step Out lets you run the current stack frame to completion to see how the code behaves as a whole.

Problem Two: 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.

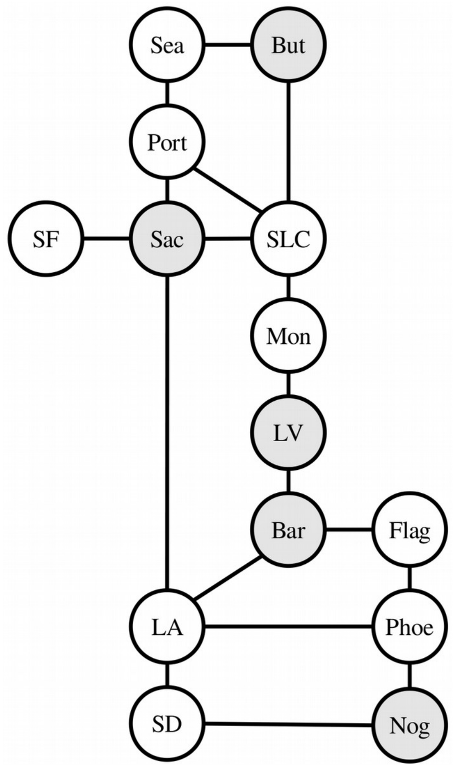
Here's what you need to do:

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 Three: 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.

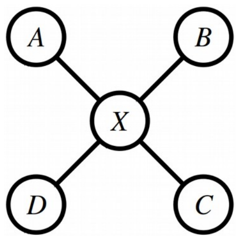
In this problem, the road network is represented as a map where each key is a city and each value is a set of cities that are immediately down the highway from them. For example, here's a fragment of the map you'd get from the above transportation network:

```
"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.

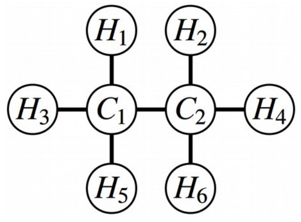
   If `numCities` is negative, your code should use the `error()` function to report an error.
3. Test your code thoroughly using our provided test driver. Once you're certain your code works – and no sooner – run the demo app to see your code in action. (More on that later.)

Some notes on this problem:

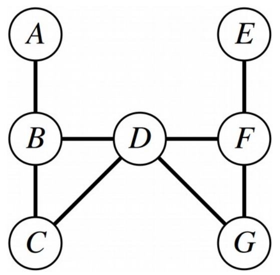
- We recommend proceeding in two steps. First, just focus on getting the return value right – that is, write a function that answers the question "is it possible to cover everything with only this many cities having supplies?" and which ignores the outparameter. Once that's working – and no sooner – 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.

(Optional) Part Three: Extensions!

There are tons of variations on these problems. 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: instead of picking an arbitrary uncovered city at each point in the recursion, what if you pick the uncovered 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! You can add a new map file into the `res/` directory by creating a file with the `.dst` suffix. Use the existing `.dst` files as a reference.

Submission Instructions

To submit your assignment, go through the Assignment Submission Checklist to make sure your code is ready to submit. In particular, make sure to auto-indent your code! For each of your source code files, highlight the code you've written, right-click it, and choose "Auto-Indent Selection." Isn't that pretty?

Then, upload these files to https://paperless.stanford.edu:

- `res/DebuggingAnswers.txt`
- `DoctorsWithoutOrders.cpp`
- `DisasterPlanning.cpp`

If you modified any other files in the course of coding things up – for example, if you were doing extensions – please submit those files as well.

And then do a victory jig – you've just built some really, really impressive pieces of software and demonstrated a thorough command of recursive problem-solving. Seriously, think back to the start of the quarter. Did you think you'd be making programs like these five weeks ago?

Good luck, and have fun!