CS106X
Winter 2018

Handout 17
January 31st, 2018

Assignment 4: ADTs and Recursion

Huge props for Keith Schwarz for devising all of these problems, and kudos to Julie Zelenski, Marty Stepp, and Jerry Cain for assisting Keith with their construction. Special thanks to Nick Bowman and Eli Echt-Wilson for providing historical election data.

Motivation

Recursion is an extremely powerful problem-solving tool with tons of practical applications. This assignment consists of four real-world recursion problems, each of which is interesting in its own right. By the time you're done, 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.

This assignment, like the preceding one, explores a lot of different concepts in recursion. The assignment, however, intentionally leaves the gaming domain behind and asks that apply recursion and recursive backtracking to scientifically interesting problems that really come up in practice. It also has the lovely side effect of revisiting a good number of the containers—**`Set`**, **`Map`**, **`Vector`**, etc.—that you learned about during the first two weeks of the quarter. TL;DR: It's great preparation for your first midterm.

If you start typing out code without a clear sense of how your recursive solution will work, chances are you'll end up going down the wrong path. Before you begin coding, try thinking about the recursive structure of the code you're writing. Does anything fall into one of the nice categories we've seen before (listing subsets, permutations, combinations, etc.)? When using backtracking, what choices can you make at each step? Talking through these questions before you start coding and making sure you have a good conceptual understanding of what it is that you need to do can save you many, many hours of coding and debugging. (Note: we're not supplying working solutions to the problems as we have for past assignments, since they're all isolated functions that work within a larger test framework, and we provide code for that framework.)

Due: Wednesday, February 7th at 11:59 pm[1]

[1] You are strongly discouraged from taking any late days on this assignment, since doing so will impede your ability to prepare for the in-class midterm on Friday, February 9th.

Problem 1: Doctors Without Orders

You've helped Recursia build out its health care system, but it now faces a crisis! No one has told the Recursian doctors which patients to see – they're Doctors Without Orders! As Minister of Health, it's time to, once again, help the Recursians with their medical needs.

Consider the following two record definitions designed to represent doctors and patients:

```
struct Doctor {
   string name;
   int hoursFree;
};

struct Patient {
   string name;
   int hoursNeeded;
};
```

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. Your task is to write a function

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

that takes as input a list of available doctors, a list of available patients, and then returns whether it's possible to schedule all the patients so that each one is seen by some doctor for the required amount of time. If it is possible to schedule everyone, the function should fill in the final **`schedule`** parameter by associating each doctor's name with the set of the names of patients he or she can see.

For example, suppose we have these doctors and 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 the result. If, for example, Patient Lacks needed three hours instead of just two, there wouldn't be a way to schedule all the patients such that all could be seen. On the other hand, if Patient Washkansky needed seven hours instead of six, there'd still be a way to schedule everyone. (Do you see how?)

This problem is all about recursive backtracking. Think about what decision you might make at each point in time. How do you commit to the decision and then undo it if it doesn't work out? As always, feel free to introduce as many helper functions as you'd like. You may even want to make the primary function a wrapper around some other recursive function.

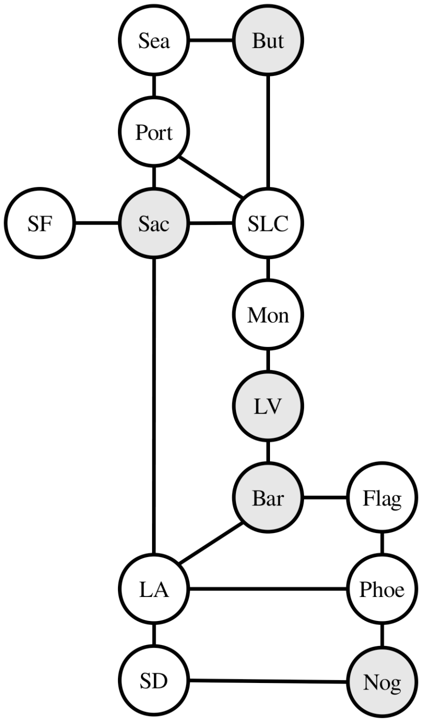
Some notes on this problem:

- You can assume that **`schedule`** is empty when the function is called.
- If your function returns **`false`**, the final contents of **`schedule`** don't matter (though we suspect your code will probably leave it blank).
- 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 in the course of solving this problem.
- You can assume no two doctors have the same name and no two patients have the same name.
- You may find it easier to solve this problem first by simply getting the return value right and ignoring the **`schedule`** parameter. Once you're sure that your code is always producing the right answer, update it so that you actually fill in **`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.
- 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 **`schedule`** (associated with an empty set of patients) or leave the doctor out entirely, whichever you'd prefer.

Problem 2: Disaster Preparation

Disasters—natural and unnatural—are inevitable, and cities need to be prepared to respond to them when they occur.

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



Imagine that you have access to a country's major highway networks. We can imagine that there are a number of different cities, some of which are right down the highway from others. To the right is a fragment of the US Interstate Highway System for the Western US. Suppose we put emergency supplies in Sacramento, Butte, Las Vegas, Barstow, and Nogales (shown in gray). In that case, if there's an emergency in any city, that city either already has emergency supplies or is immediately adjacent from a city that does. For example, any emergency in Nogales would be covered, since Nogales already has emergency supplies. San Francisco is covered by Sacramento, Salt Lake City is covered by both Sacramento and Butte, and Barstow is covered both by itself and by Las Vegas.

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

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

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

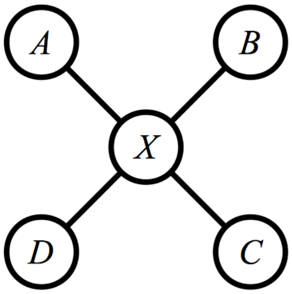
that takes as input a **`Map`** representing the road network for a region (described below) and a number of cities that can be made to hold supplies, then returns whether it's possible to make the region disaster-ready by placing supplies in at most **`numCities`** cities. If so, the function should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

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

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

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

There are many different strategies you can use to solve this problem, and some are more efficient than others. For example, one option would be to treat this problem as a combinations problem, trying out each combination of cities of the given size and seeing whether any of them would make all cities disaster ready. This option works, but it will be extremely slow on some of the larger test cases where there are over thirty cities – so slow in fact that your program might never give back an answer. That's not good enough.



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

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

Some notes on this problem:

- The road network is bidirectional. If there's a road from city A to city B, then there will always be a road back from city B to city A in the network, and both roads will be present in the parameter **roadNetwork**. You can rely on this.
- Every city appears as a key in the map, although some cities might be isolated (think Honolulu!) If that happens, the city will be represented by a key in the map associated with an empty set of adjacent cities.
- If you're allowed to use up to, say, k cities, but you find a way to solve the problem using fewer than k cities, that's fine! The set of cities you return can contain any

number of cities provided that you don't use more than k and you properly cover everything.

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

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

Problem 3: DNA Detective

You're working in a hospital and, to your dismay, there's been an outbreak of the antibiotic-resistant bacterium MRSA. You're not sure where the disease came from, whether it's endemic in the community, or whether it's a well-known strain. Fortunately, you've got a number of DNA samples from the bug, and your handy gene sequencer has returned back to you a rough copy of its genome. If you can figure out whether the different samples you're getting from different patients are all from the same strain of MRSA, you can take specific actions to try to end the spread quickly.

DNA consists of paired strands of nucleotides. The nucleotides used in DNA are adenine, cytosine, guanine, and thymine, usually just abbreviated A, C, G, and T, and DNA strands are often represented as strings made from these letters.

When DNA sequencing machines return back the specific sequence of nucleotides that make up a particular strand of DNA, they often have a small number of errors. For example, if there's a part of a DNA strand that actually reads GTGTC, then the sequencer might report it as GTGTA (substituting one letter for another), GTGT (dropping a letter), or GTGTAC (inserting a letter).

One way of quantifying how "close" two DNA strands are to one another is to use a measure called ***edit distance***. Specifically, the edit distance between two strings is defined as the number of single-letter insertions, deletions, and replacements needed to turn the first string into the second. (Incidentally, Mehran Sahami surfaced the idea of a different kind of edit distance between Karel the Robot instruction sequences during his Friday lecture, so this problem is directly relevant to a presentations you’ve seen before.) For example:

- The strings **cat** and **dog** have edit distance 3. The fastest way to turn **cat** into **dog** is to replace each letter in **cat** with the corresponding letter from **dog**.
- The strings **table** and **maple** have edit distance 2. The fastest way to turn table into **maple** is to replace the **t** in **table** with an **m** and the **b** in **table** with a **p**.
- The strings **rate** and **pirate** have edit distance 2. The fastest way to turn **rate** into **pirate** is to prepend a **p** and an **i**.
- The strings **edit** and **distance** have edit distance 6. Specifically, delete the **e** from **edit**, insert an **s** between the **i** and the **t**, and append the letters **ance** onto the back.

Let's jump back to our outbreak example. Imagine you sequence the DNA of a sample of MRSA that you've found in one patient and the DNA of a sample of MRSA you found in another patient. The sequencer will likely come back with some small number of errors in it. If the edit distance between the two DNA strands is low, it likely means that you're looking at essentially the same organism, meaning that the patients either got the bug from the same source or one gave it to another. On the other hand, if the edit distance is large, it likely means you're looking at two independent sources of infection.

Your task is to write a function

```
   bool approximatelyMatch(const string& one, const string& two, int maxDistance);
```

that takes as input two strings representing strands of DNA and a number **`maxDistance`**, then returns whether the edit distance between the two DNA strands is **`maxDistance`** or less.

As a hint for this problem, look at the first characters of **`one`** and **`two`** and think about what you might do with what you find. What happens if they match? If they don't match, you have three possible ways that you can get them to match – what are they, and how might you try them? And what happens if either input string is empty?

Some notes on this problem:

- You can assume the strings passed as input consist purely of the letters A, C, T, and G.
- It's entirely possible to solve this problem without using loops of any kind. In fact, if you do find yourself writing loops in your solution to this particular problem, it might indicate that you're missing a much easier line of reasoning.
- You're welcome to implant memoization for this problem, as a real DNA sequence match algorithm would. However, we won't be testing your function with DNA strands so large that you absolutely need memoization.

Problem 4: 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 the person 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 only 99 people, you'd need 50 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, but we'll ignore this.)

- 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?

Imagine that we have a list of information about each state, represented by this handy **struct** definition:

```
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
};
```

This record contains the name of the state, its number of electors, and its voting population. 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 appropriate), 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). The **MinInfo** structure is defined in the

RecursionToTheRescue.h header and is essentially just a pair of a minimum popular vote total plus the list of states you'd carry in the course of reaching that total:

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

To implement this function, we **strongly recommend** implementing a helper function that answers the following question:

What is the minimum number of popular votes needed to get at least V electoral votes, using only states from index i and greater in the Vector?

Notice that if you solve this problem with V set to a majority of the total electoral votes and with *i = 0*, then you've essentially solved the original problem (do you see why?). We strongly recommend making your original function a wrapper around a helper function that solves this specific problem.

In the course of solving this problem, you might find yourself in the unfortunate situation where, for some specific values of V and i, there's no possible way to get V votes using only the states from index i and forward. For example, if you're 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 some kind of sentinel value indicating that it's not possible to secure that many votes. We recommend using the special value **INT_MAX**, which represents the maximum possible value that you can store in an integer. The advantage of this sentinel value 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.

When you first implement this function, we strongly recommend testing it out using the simplified test cases we've provided you in the test framework's main menu. These test cases use real election data, but only consider ten states out of a larger election. That should make it easier for you to check whether your solution works on smaller examples.

Without using memoization, it's almost guaranteed that your code won't be fast enough to work with the full election datasets, since there are just too many subsets of states to consider. To scale this up to work with actual elections data, **you'll need to work memoization into your solution**. The good news is that, if you've followed the strategy we've outlined above, you should find that it's relatively straightforward to introduce memoization into your solution. Once you've gotten that working, try running your code on full elections data. I think you'll be pleasantly surprised by how fast it runs!

Here are some general notes on this problem:

- The historical election data – and our reduced test cases – do not always include all 50 current 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 you'll get them as input.

- The total number of Electoral College votes to win the election depends on the number of electors, which varies over time. Although you currently need 270 electoral votes to become President, you should not assume this in your solution.

- Remember that in all elections you need ***strictly more*** than half the votes to win. If there are either 100 or 101 people in a state, you need 51 votes to win its electors. If there are 538 or 539 total electoral votes, you'd need 270 electoral votes to become president.

- You can represent the table in the memoization step in a number of different ways. One option that isn't mentioned in the textbook is the **SparseGrid** type, which depending on your approach might be nice to know about. Check the documentation up on the CS106X course website for more information.

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 previously 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 arrived at a working solution, 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?[2]

[2] The historical election data here was compiled from a number of sources. In many early elections the state legislatures decided how to choose electors, and so in some cases we extrapolated to estimate 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!