CS106B Handout 17
Winter 2020 February 3, 2020

Practice Midterm Exam I

This exam is closed-book and closed-computer. You may have a double-sided, 8.5" × 11" sheet of notes with you when you take this exam. You may not have any other notes with you during the exam. You may not use any electronic devices (laptops, cell phones, etc.) during the exam. Please write all of your solutions on this physical copy of the exam.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you get partial credit if they help us determine what you were trying to do. You do not need to worry about efficiency unless a problem states otherwise. You don't need to prototype helper functions you write, and you don't need to explicitly `#include` libraries you want to use.

This exam is "self-contained" in the sense that if you're expected to reference code from the lectures, textbook, assignments, or section handouts, we'll explicitly mention what that code is and provide sufficient context for you to use it. You should have received a C++ reference sheet before starting this exam with a refresher of common functions and types.

This practice exam is completely optional and does not contribute to your overall grade in the course. However, we think that taking the time to work through it under realistic conditions is an excellent way to prepare for the actual exam and figure out where you need to focus your studying.

You have three hours to complete this exam. There are four questions and 32 total points.

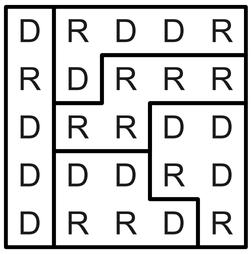
Problem One: Containers I (8 Points)

Representation in the United States House of Representatives is based on the number of voters in each state; states with higher total population (say, California) have a greater number of representatives than a state with a lower total population (say, Wyoming). Each state is divided into electoral districts, with each district electing a representative. Every district holds separate elections for its representative, and the candidate that receives the majority of the votes within a district is chosen as the representative.

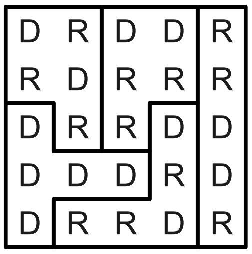
While this system means that representatives are accountable to the voters within their district, it makes it possible for one group of voters to have representation disproportionate to their size. For example, suppose that the cities in a state vote either primarily for Democrats or primarily for Republicans. Let's suppose that the cities are laid out like this:

D	R	D	D	R
R	D	R	R	R
D	R	R	D	D
D	D	D	R	D
D	R	R	D	R

Here, there are 25 cities – 13 that vote Democrat, and 12 that vote Republican. Suppose that we need to split them into five districts of five cities each. If we split the cities this way:



Then four of the five districts have a Democrat majority, so with high probability there will be four Democrats and one Republican elected, even though the total votes cast are about 50/50 split between Democrats and Republicans. Similarly, if we split the cities this way:



Then the outcome is reversed, with four Republicans likely elected and only one Democrat.

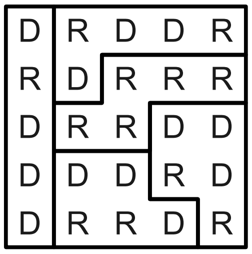
Given a political party, let's define the ***gerrymandering ratio*** of that political party to be the ratio between the percent of districts in which the political party has a majority to the percent of total statewide votes for that party. That is,

$$\text{Gerrymandering ratio} = \frac{\text{Percent of districts with party majority}}{\text{Percent of total votes}}$$

For a given political party, if the gerrymandering ratio is high, it means that the district boundaries over-represent that party. If this number is low, it means that the district boundaries underrepresent that party. Your job is to write the following function, which computes the gerrymandering ratio for some party:

```
double gerrymanderingRatio(const Vector<Vector<string>>& districts, string party)
```

This function accepts two parameters. The first parameter, a `Vector<Vector<string>>`, is a list of all the voting districts. Each district is represented as a `Vector<string>` listing the voting preferences of all the cities within the district. For example, given this districting:



The input `Vector<Vector<string>>` might be

```
{{"D", "R", "D", "D", "D"},   // Vertical column at left
 {"R", "D", "D", "R", "D"},   // L-shaped district on top
 {"R", "R", "R", "R", "R"},   // S-shaped district in middle
 {"D", "D", "R", "R", "D"},   // Bottom-center district
 {"D", "D", "R", "D", "R"}}   // Bottom-right district
```

The second parameter to `gerrymanderingRatio` is the political party whose gerrymandering ratio should be computed. For example, if the input party was `"D"`, then given the above districts the output would be computed as follows:

- The percentage of districts with a Democratic majority is 80% (4 / 5 districts)
- The total percentage of cities that vote Democrat is 52% (13 / 25 cities).

So the gerrymandering ratio is 80 / 52 ≈ 1.54

In solving this problem you can assume the following:

- A political party has the majority of the cities in a district if it has 50% or more of the cities in that district.
- The names of political parties are capitalized consistently, so don't worry about case-sensitivity.
- Districts do not necessarily all have equal size.
- There may be any number of political parties, not just two.
- There is at least one district, and each district has at least one city in it.
- There is at least one city that will vote for the chosen political party, so the ratio you are computing is always defined.

Feel free to tear out this page and the previous one as a reference, and write your solution on the next page.

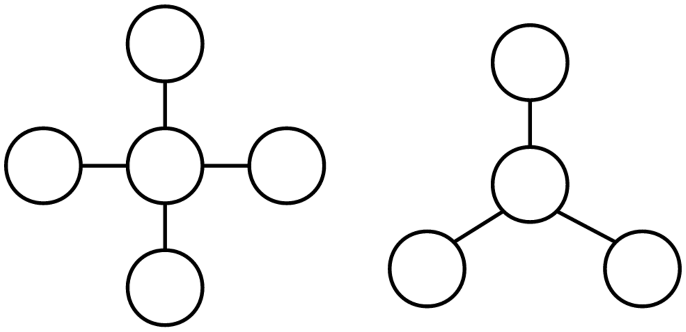
```
double gerrymanderingRatio(const Vector<Vector<string>>& districts, string party) {
```

(Extra space for your answer to Problem One, if you need it.)

Problem Two: Containers II (8 Points)

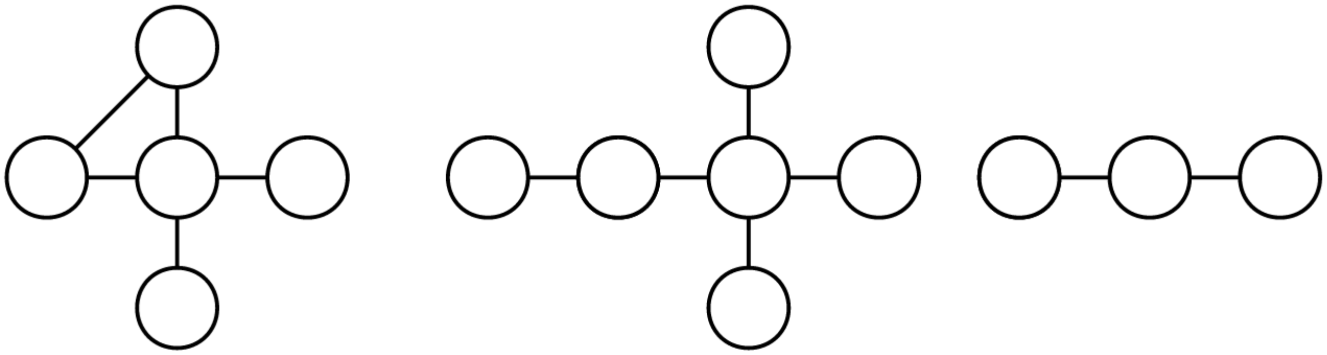
You have a road network for a country represented as a `HashMap<string, HashSet<string>>`. Each key is the name of a city, and the associated value consists of all the cities that are adjacent to that city. You can assume that the road network is bidirectional: if *A* is adjacent to *B*, then *B* is adjacent to *A*. You can also assume no city is adjacent to itself.

A ***star*** is a cluster of four or more cities arranged as follows: there's a single city in the center that's connected to all the other cities in the cluster, and every other city in the cluster is only connected to the central city. Here are some stars shown below:



Stars commonly arise in road networks in smaller island nations: you have a main, central city (often a capital city) and a bunch of smaller, outlying towns.

Here are some examples of groups of cities that aren't stars. The group on the left isn't a star because two of the peripheral cities are connected to one another (and therefore not just the central city). The group in the center isn't a star because one of the peripheral cities is connected to a city besides the central one. Finally, the group on the right isn't a star because it doesn't have the minimum required number of nodes.



If you have a country that consists of a large archipelago, you might find that its road network consists of multiple different independent stars. In fact, the number of stars in a road network is a rough proxy for how decentralized that country is.

Your task is to write a function

```
int countStarsIn(const HashMap<string, HashSet<string>>& network);
```

that takes as input the road network, then returns the number of stars in the network. You don't need to worry about efficiency, but do be careful not to count the same star multiple times.

```
int countStarsIn(const HashMap<string, HashSet<string>>& network) {
```

(Extra space for your answer to Problem Two, if you need it.)

Problem Three: Recursion I (8 Points)

You've been working on preparing a report as part of a larger team. Each person on the team has been tasked with writing a different section of the report. To make sure that the final product looks good and is ready to go, your team has decided to have each person in the team proofread a section that they didn't themselves write.

There are a lot of ways to do this. For example, suppose your team has five members conveniently named *A*, *B*, *C*, *D*, and *E*. One option would be to have *A* read *B*'s section, *B* read *C*'s section, *C* read *D*'s section, *D* read *E*'s section, and *E* read *A*'s section, with everyone proofreading in a big ring. Another option would be to have *A* and *B* each proofread the other's section, then have *C* proofread *D*'s section, *D* read *E*'s section, and *E* read *C*'s section. A third option would be to have *A* read *E*'s work, *E* read *C*'s work, and *C* read *A*'s work, then to have *B* and *D* proofread each other's work. The only restrictions are that (1) each section needs to be proofread by exactly one person and (2) no person is allowed to proofread their own work.

Write a function

```
void listAllProofreadingArrangements(const HashSet<string>& people);
```

that lists off all ways that everyone can be assigned a person's work to check so that no person is assigned to check their own work. For example, given the five people listed above, this function might print the following output:

```
A checks B,   B checks A,   C checks E,   D checks C,   E checks D
A checks B,   B checks A,   C checks D,   D checks E,   E checks C
A checks C,   B checks A,   C checks B,   D checks E,   E checks D
                  (… many, many lines skipped …)
A checks C,   B checks E,   C checks D,   D checks B,   E checks A
A checks D,   B checks C,   C checks E,   D checks B,   E checks A
A checks E,   B checks C,   C checks D,   D checks B,   E checks A
```

Some notes on this problem:

- You're free to list the proofreading assignments in any order that you'd like. However, you should *make sure that you don't list the same assignment twice*.
- Your function should print all the arrangements it finds to `cout`. It shouldn't return anything.
- Your solution needs to be recursive – that's kinda what we're testing here. ☺
- While in general you don't need to worry about efficiency, you should *not* implement this function by listing all possible permutations of the original group of people and then checking each one to see whether someone is assigned to themselves. That ends up being a bit too slow to be practical.
- *Your output doesn't have to have the exact same format as ours*. As long as you print out something that makes clear who's supposed to proofread what, you should be good to go. In case it helps, you may want to take advantage of the fact that you can use `cout` to directly print out a container class (`Vector`, `HashSet`, `HashMap`, `Lexicon`, etc.). For example, if you have a `HashMap` named `myMap`, writing `cout << myMap << endl;` prints out all the key/value pairs.
- As a hint, focus on any one person in the group. You know that they're going to have to proofread some section. Consider exploring each possible way they could do so.

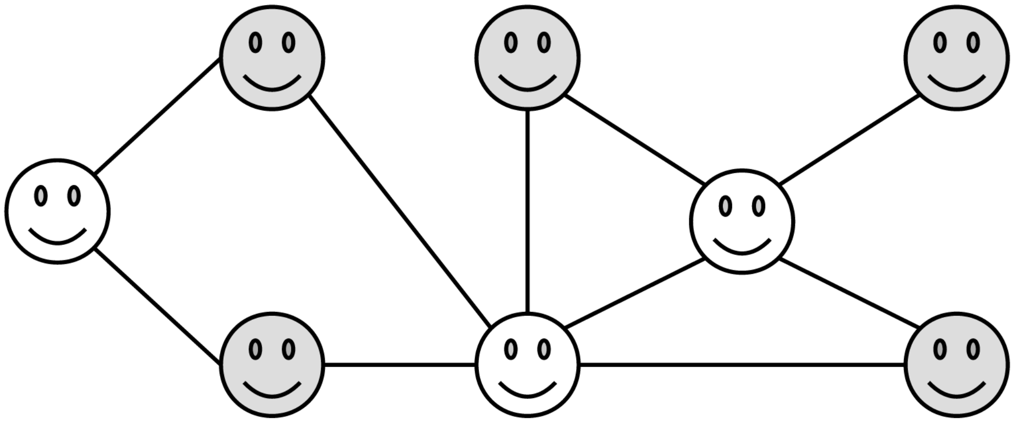
```
void listAllProofreadingArrangements(const HashSet<string>& people) {
```

(Extra space for your answer to Problem Three, if you need it.)

Problem Four: Recursion II (8 Points)

One of the risks that comes up when conducting field surveys is *sampling bias*, that you accidentally survey a bunch of people with similar backgrounds, tastes, and life experiences and therefore end up with a highly skewed view of what people think, like, and feel. There are a number of ways to try to control for this. One option that's viable given online social networks is to find a group of people of which no two are Facebook friends, then administer the survey to them. Since two people who are a part of some similar organization or group are likely to be Facebook friends, this way of sampling people ensures that you get a fairly wide distribution.

For example, in the social network shown below (with lines representing friendships), the folks shaded in gray would be an unbiased group, as no two of them are friends of one another.



Your task is to write a function

```
HashSet<string>
largestUnbiasedGroupIn(const HashMap<string, HashSet<string>>& network);
```

that takes as input a `HashMap` representing Facebook friendships (described later) and returns the largest group of people you can survey, subject to the restriction that you can't survey any two people who are friends.

The `network` parameter represents the social network. Each key is a person, and each person's associated value is the set of all the people they're Facebook friends with. You can assume that friendship is symmetric, so if person *A* is a friend of person *B*, then person *B* is a friend of person *A*. Similarly, you can assume that no one is friends with themselves.

Some other things to keep in mind:

- You need to use recursion to solve this problem – that's what we're testing here. 😃
- Your solution must not work by simply generating all possible groups of people and then checking at the end which ones are valid (i.e. whether no two people in the group are Facebook friends). This approach is far too slow to be practical.

```
HashSet<string>
largestUnbiasedGroupIn(const HashMap<string, HashSet<string>>& network) {
```

(Extra space for your answer to Problem Four, if you need it.)