Practice Midterm Exam II

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)

Google Translate is powered, in large part, by a technique called ***word2vec*** that builds an understanding of a language by looking at the context in which each word occurs.

Imagine you have a word like "well." There are certain words that might reasonably appear immediately before the word "well" in a sentence, like "feeling," "going," "reads," etc., and some words that that are highly unlikely to appear before "well," like "cake," "circumspection," and "election." The idea behind word2vec is to find connections between words by looking for pairs of words that have similar sets of words preceding them. Those words likely have some kind of connection between them, and the rest of the logic in word2vec works by trying to discover what those connections are.

Your task is to write a function

```
HashMap<string, Lexicon> predecessorMap(istream& input);
```

that takes as input an `istream&` containing the contents of a file, then returns a `Map<string, Lexicon>` that associates each word in the file with all the words that appeared directly before that word. For example, given JFK's quote

> "Ask not what your country can do for you; ask what you can do for your country,"

your function should return a `HashMap` with these key/value pairs:

```
    "not" : { "ask" }
   "what" : { "not", "ask" }
   "your" : { "what", "for" }
"country" : { "your" }
    "can" : { "country", "you" }
     "do" : { "can" }
    "for" : { "do" }
    "you" : { "for", "what" }
    "ask" : { "you" }
```

Notice that although the word "ask" appears twice in the quote, the first time it appears it's the first word in the file and so nothing precedes The second time it appears, it's preceded by some whitespace and a semicolon, but before that is the word "you," which is what ultimately appears in the `Lexicon`.

Some notes on this problem:

- You can assume that a token counts as a word if its first character is a letter. You can use the `isalpha` function to check if a character is a letter.
- You can assume you have access to the function

  ```
  Vector<string> tokenize(const string& input);
  ```

  that we provided you in Assignment 3.
- Your code should be case-insensitive, so it should return the same result regardless of the capitalization of the words in the file. The capitalization of the keys in the map is completely up to you.
- Your code should completely ignore non-word tokens (whitespace, punctuation, quotation marks, etc.) and just look at the words it encounters.
- It is not guaranteed that the file has any words in it, and there's no upper bound on how big the file can be.

```
Vector<string> tokenize(const string& str); // Provided to you
bool isalpha(char ch);                      // Provided to you

HashMap<string, Lexicon> predecessorMap(istream& input) {
```

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

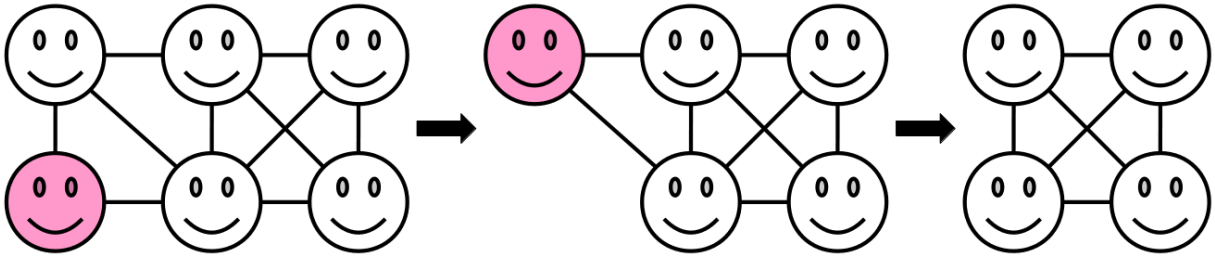
Problem Two: Containers II (8 Points)

Social networks – Facebook, LinkedIn, Instagram, etc. – are popular *because* they're popular. The more people that are on a network, the more valuable it is to join that network. This means that, generally speaking, once a network starts growing, it will tend to keep growing. At the same time, this means that if many people start *leaving* a social network, it will tend to cause other people to leave as well.

Researchers who study social networks, many of whom work in our CS department, have methods to quantify how resilient social networks are as people begin exiting. One way to do this is to look at the ***k-core*** of the network. Given a number k, you can find the k-core of a social network as follows:

- If everyone has at least k friends on the network, then the k-core consists of everyone on that network.
- Otherwise, there's some person who has fewer than k friends on the network. Delete them from the network and repeat this process.

For example, here's how we'd find the 3-core of the social network shown to the left. At each step, we pick a person with fewer than three friends in the network (shown in red) and remove them. We stop once everyone has three or more friends, and the remaining people form the 3-core.



Intuitively, the k-core of a social network represents the people who are unlikely to leave – they're tightly integrated into the network, and the people they're friends with are tightly integrated as well.

Your task is to write a function

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

that takes as input a social network (described below) and a number k, then returns the set of people in the k-core of that network.

Here, the social network is represented as a `HashMap<string, HashSet<string>>`, where each key represents a person and each value is the set of people they're friends with. You can assume that each person listed as a friend actually exists in the network and that friendship is symmetric: if A is friends with B, then B is also friends with A.

Some notes on this problem:

- You're free to implement this one either iteratively or recursively. It's up to you to decide.
- Normally, we'd ask you to do error-checking, but for the purposes of this problem you can assume that $k \geq 0$ and you don't need to validate this.
- It's possible that someone in the network has no friends. That person would be represented as a key in the map associated with an empty set.
- You can assume no one is their own friend.
- While you don't need to come up with the most efficient solution possible, you should avoid solution routes that are unnecessarily slow. For example, don't list off all possible sets of people, then check whether each of them is the k-core.
- In case it helps, you can assume no person's name is the empty string.

```
Set<string> kCoreOf(const Map<string, Set<string>>& network, int k) {
```

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

Problem Three: Recursion I (8 Points)

At present, it costs around $5,000 to place one kilogram of material into orbit. This means that if you're floating on the International Space Station, you probably shouldn't expect that much out of your meals. They're calibrated to be nutritionally complete and lightweight, and while there's some effort made to provide nice things like "flavor" and "texture," that's not always possible.

Let's imagine that you want to be nice and send a cosmic care package up to the ISS with some sweets for the crew to share. Every penny matters at five grand a kilo, so you'll want to be very sure that what you send up into the vast abyss will actually make the folks there happy. You have a list of the desserts that each individual ISS crew member would enjoy. What's the smallest collection of treats you could send up that would ensure everyone gets something they like?

For example, suppose the ISS crew members have the following preferences, with each row representing one person's cravings:

```
{ "Pumpkin pie", "Revani", "Sufganiyot" }
   { "Castella", "Kheer", "Melktert" }
  { "Po'e", "Tangyuan", "Tres leches" }
     { "Apfelstrudel", "Tangyuan" }
 { "Alfajores", "Brigadeiro", "Revani" }
        { "Baklava", "Revani" }
           { "Melktert" }
{ "Brownies", "Cannolis", "Castella", "Po'e", "Revani" }
     { "Cendol", "Kashata", "Tangyuan" }
{ "Baklava", "Castella", "Kheer", "Revani" }
```

The smallest care package that would cheer up the whole crew would have three items: revani, melktert, and tangyuan. Anything smaller than this will leave someone unhappy.

Write a function

```
HashSet<string> smallestCarePackageFor(const Vector<HashSet<string>>& preferences);
```

that takes as input a list containing sets of what treats each crew member would like, then returns the smallest care package that would make everyone on the ISS happy.

While in principle you could solve this problem by listing off all possible subsets of treats and seeing which ones satisfy everyone's sweet tooths, this approach is probably not going to be very fast if there are a lot of different choices for sweets. Instead, use the following approach: pick a person who hasn't yet had anything sent up that would make them happy, then consider each way you could send them something that would fill them with joy.

Some notes on this problem:

- Yep, you guessed it! You have to do this one recursively. 😀
- To clarify, if you send up some item to space, you can assume there's enough of it for everyone on the space station to share. Sending up revani, for example, means sending up enough revani for each crew member to have a piece.
- Each person's preferences will contain at least one item. There is no fixed upper size to how many preferences each person can have.
- If there are multiple care packages that are tied for the smallest, you can return any of them.
- In case it helps, you can assume no treat's name is the empty string.

```
HashSet<string> smallestCarePackageFor(const Vector<HashSet<string>>& preferences) {
```

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

Problem Four: Recursion II (8 Points)

That new hire you found a schedule for has been doing so well that you're now in a position to hire many additional workers. How exciting! The question now is how to schedule all of those workers so that they collectively produce as much value as possible.

Your task is to write a function

```
HashMap<string, HashSet<Shift>>
bestScheduleFor(const HashSet<Shift>& shifts,
                const HashMap<string, int>& hoursFree);
```

that takes as input a `HashSet` of all possible shifts, along with a `Hash` from employees (represented by their names) to the maximum number of hours those employees can work, then returns a `HashMap` from employees to the shifts they should take. Specifically, you should return an assignment of workers to shifts that provides the maximum total value out of all the possible ways you could assign shifts.

As in Assignment 3, you are not allowed to assign two overlapping shifts to the same employee, and you cannot assign an employee to work more hours than they have available. Additionally, you cannot assign two or more employees to the same shift.

As a refresher, you can assume you have access to these functions that operate on shifts:

```
int lengthOf(const Shift& shift);    // Returns the length of a shift.

int valueOf(const Shift& shift);     // Returns the value of a shift.

bool overlapsWith(const Shift& one,  // Returns whether two shifts overlap.
                  const Shift& two);
```

Some notes on this problem:

- You need to solve this problem recursively. That's kinda what we're testing here. 😀
- The input `HashMap` might say that some workers have zero hours free, but no worker will ever have a negative number of free hours.
- If a worker isn't assigned any shifts, you can either include them in the output `HashMap` associated with an empty set, or you can leave them out of the output `HashMap` entirely; your choice.
- Don't proceed one person at a time, giving the first person the set of shifts that maximizes their value produced, then the second person the set of shifts that maximizes their value produced, etc. Surprisingly, this strategy does not always give a schedule that produces the maximum value.
- The input `HashMap` may be empty. In that case, what do you think you should return?
- If there are multiple shift assignments that are all equally good, you can return any one of them.
- Your code isn't required to be as optimized as possible, but you should avoid unnecessary sources of inefficiency. For example, don't try assigning a worker shifts that conflict with other shifts they've been assigned or which exceed their hour limits.

```
HashMap<string, HashSet<Shift>>
bestScheduleFor(const HashSet<Shift>& shifts,
                const HashMap<string, int>& hoursFree) {
```

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