Practice CS106B Midterm Exam

This handout is intended to give you practice solving problems that are comparable in format and difficulty to the problems that will appear on the midterm examination on Tuesday, May 7. A solution set to this practice exam will be handed out on Friday.

Coverage

The midterm covers the material presented in class through the lecture on Wednesday, May 1, which means that you are responsible for the chapters in the text through Chapter 10. Topics covered exclusively in the course reader and not in lecture will not be tested in depth.

General instructions

Answer each of the questions given below. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit.

Each question is marked with the number of points assigned to that problem. The total number of points on the exam is 180. We intend for the number of points to be roughly comparable to the number of minutes you should spend on that problem.

In all questions, you may include functions or definitions that have been developed in the course. First of all, we will assume that you have included any of the header files that we have covered in the text. Thus, if you want to use a `Vector`, you can simply do so without bothering to spend the time copying out the appropriate `#include` line. If you want to use a function that appears in the book that is not exported by an interface, you should give us the page number on which that function appears. If you want to include code from one of your own assignments, we won't have a copy, and you'll need to copy the code to your exam.

Unless otherwise indicated as part of the instructions for a specific problem, comments are not 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 to get partial credit on a problem if they help us determine what you were trying to do.

The examination is open-book, and you may make use of any texts, handouts, or course notes. You may not, however, use a computer of any kind.

Normally, we would leave a lot of blank space for you to write your answers, but in the interest of saving trees we've removed most of the whitespace from this exam. You don't need to bring a blue book with you, but it might be useful to have scratch paper available.

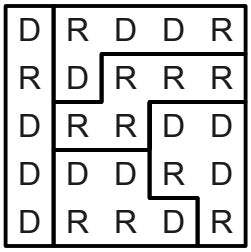
Problem One: Detecting Gerrymandering (35 Points)

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.

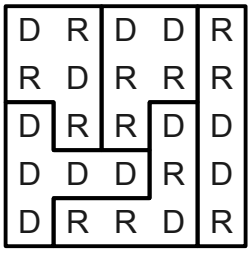
While this system means that representatives are accountable to the voters within their district, it makes it possible for one group of voters to have representation disproportionate to their size. For example, suppose that the cities in a state vote either primarily for Democrats or primarily for Republicans.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

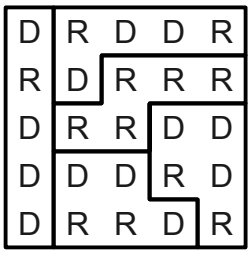
Given a political party, let's define the **gerrymandering ratio** of that political party to be the ratio between the percent of districts in which the political party has a majority to the percent of total statewide votes for that party. That is,

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

For a given political party, if the gerrymandering ratio is high, it means that the district boundaries overrepresent 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(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.

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

Problem Two: Cryptoquote Assistant (40 Points)

A *cryptoquote* is a piece of English text that has been encrypted by a *substitution cipher*. In a substitution cipher, every letter is replaced by some other letter according to an encryption key. For example, we might replace the 26 letters in the alphabet as follows:

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
I	X	E	C	G	Q	P	S	W	F	O	A	U	Y	D	B	R	J	T	K	Z	M	H	L	V	N

Using this encryption key, the word “programming” would be rendered “bjdpjiuuwyp,” and the word “cool” would be rendered “edda.”

Now, suppose that you find the encrypted word “bjdpjiuuwyp” but don't have the above encryption key. Could you figure out that the original word was “programming?” Since every letter is replaced consistently, you can get a few clues as to the original word. For example, since the fourth and last letter of “bjdpjiuuwyp” are the same ('p'), you would know that in the original word the fourth and last letters must also be the same. Similarly, because the second and fifth letters of “bjdpjiuuwyp” are the same ('j'), you know that the second and fifth letters of the original word must also be the same. However, the second and fifth letters of the word cannot be the same as the fourth and last letters of the word, since they're represented by different letters in the cryptoquote. By following this line of reasoning, you can narrow down which word was encrypted to one of three: “programming,” “progressing,” and “outguessing.” Notice that in each of these three words, the pattern of letters matches the pattern of letters in “bjdpjiuuwyp.”

More generally, you are interested in the following. Suppose that you have a pattern consisting of some combination of letters (which, like “bjdpjiuuwyp,” doesn't have to actually be a word). You want to determine all possible English words whose letters match that pattern. For example, the pattern “xqqx” matches the word “deed,” since the letters in “deed” have the same pattern as the letters in “xqqx.” However, “xqqx” does not match “meme,” because the letter pattern is different. Similarly, the pattern “abcde” can match the word “about,” since all the letters are different, but not “songs,” because the first and last letters are the same.

Write a function

```
Lexicon allWordsMatching(string pattern, Lexicon& words);
```

that accepts as input a lower-case pattern string and a **`Lexicon`** of all words in English, then returns a **`Lexicon`** containing all English words that match the given pattern. There are many ways to solve this problem, but it is probably easiest to just iterate over all the words in the **`Lexicon`** and check which words match the pattern string.

When checking if a word matches a pattern, remember that

- Each letter of the word must consistently map to the same letter in the pattern, and
- No two different letters of the word can map to the same letter in the pattern.

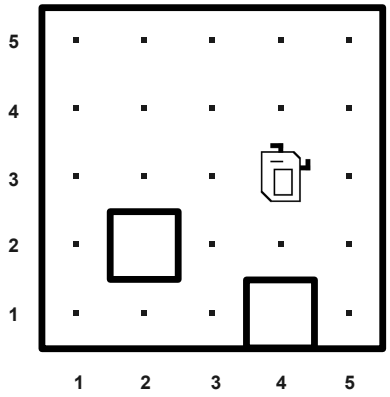
```
Lexicon allWordsMatching(string pattern, Lexicon& words) {
```

Problem Three: Karel is Blocked (35 Points)

In Assignment 3, one of the warmup problems ("Karel Goes Home") asked you to count how many ways Karel could get from a particular street corner back to his home at 1st Street and 1st Avenue. This question explores a variant of this problem.

(i) When Karel Comes Marching Home (20 Points)

Suppose that Karel is at a particular street corner and is interested in getting back home to 1st Street and 1st Avenue (recall that streets run horizontally and avenues run vertically). As before, Karel is constrained in that he must always move only leftward and downward. However, this time certain corners have been closed off, and Karel can neither enter nor leave them. For example:



In this world, there are only three paths home:

- Left three times, then down twice;
- Left, down twice, and left two more times; and
- Down, left, down, and left two more times.

Suppose that Karel's world is represented by a **`Grid<bool>`** whose elements are **`true`** if the corner at the given position is blocked and **`false`** otherwise. For example, in the above world, we have

```
world[2][2] = true;
world[1][4] = true;
```

Your task is to write a function

```
int numPathsHome(Grid<bool>& world, int street, int avenue)
```

that accepts as input a **`Grid<bool>`** describing Karel's world, along with Karel's position (given by his street number and avenue number), and returns the number of paths from Karel's current location back to the corner of 1st Street and 1st Avenue. Note that the **`Grid`** is zero-indexed, but Karel's world is one-indexed. This means that if the world has s streets and a avenues, the grid have size $(s + 1) \times (a + 1)$. You can assume that Karel's home is not blocked, so **`world[1][1]`** is always **`false`**. Your function should correctly be able to handle the case where Karel's position is outside of the bounds of the world.

(ii) Karel Analyzes His Strategy (15 Points)

Draw out a tree showing what recursive calls your function makes when Karel tries to go from (3, 3) back home in a world with no obstacles. Is there a way that you might make this tree smaller?

Problem Four: Scheduling Patients (40 Points)

You are working at a hospital trying to coordinate times in which non-emergency patients can meet with their doctors. Each doctor has a maximum number of hours each day that she can see patients. For each patient, you are given how much time she will need to spend with her doctor. Given the amount of time that each doctor is free on a given day, along with the amount of time required for each patient, you are interested in determining whether or not it is possible for every patient to be seen.

For example, suppose that you have the following doctors and the following patients:

Doctors	Patients
Dr. A: Available 7 hours	Patient M: 5 hours needed
Dr. B: Available 5 hours	Patient N: 2 hours needed
Dr. C: Available 6 Hours	Patient O: 4 hours needed
	Patient P: 4 hours needed
	Patient Q: 3 hours needed

In this case, it is possible for the doctors to see all the patients: Dr. A sees patients P and Q, Dr. B sees patient M, and Dr. C sees patients N and O. On the other hand, if you had the following setup:

Doctors	Patients
Dr. A: Available 7 hours	Patient M: 4 hours needed
Dr. B: Available 5 hours	Patient N: 4 hours needed
Dr. C: Available 6 Hours	Patient O: 4 hours needed
	Patient P: 4 hour needed

Then there is no way to schedule all the patients, even though the doctors are collectively working for 18 hours and only 16 total hours would be required.

Write a function

```
bool arePatientsSchedulable(Vector<int>& doctors, Vector<int>& patients);
```

that accepts as input the doctors' availabilities and patients' time requirements, then returns whether it is possible for all patients to be seen. Here, `doctors` is a `Vector<int>` representing the number of hours that each doctor can work, and `patients` is a `Vector<int>` holding the number of hours required for each patient. For example, the first set of requirements would be represented as

```
doctors:  7, 5, 6
patients: 5, 2, 4, 4, 3
```

and the second would be

```
doctors:  7, 5, 6
patients: 4, 4, 4, 4
```

```
bool arePatientsSchedulable(Vector<int>& doctors, Vector<int>& patients) {
```

Problem Five: Modifying Merge Sort (30 Points)

In merge sort, we repeatedly split the input array into two pieces of roughly equal size, recursively sort those pieces, then merge them back together. Suppose we modify merge sort so that it still operates recursively, but chooses how it splits the array differently. Specifically, consider this algorithm:

- **Base Case**: If the array has size zero or size one, the array is already sorted.
- **Recursive Step**: Remove the last element from the array. Recursively sort the remaining array elements, then merge the sorted sequence with the sequence containing just the last element.

(i) Calculating Complexity (20 Points)

What is the worst-case big-O complexity of this modified version of merge sort? Is this better or worse than the worst-case big-O complexity of normal merge sort? Explain your answer. As a hint, it may be useful to trace through the execution of this algorithm on a small input array.

(ii) Seem Familiar? (10 Points)

Although we arrived at this algorithm using the intuition for merge sort, this algorithm is more closely related to a different sorting algorithm. Which other sorting algorithm is this algorithm most similar to?