CS106B Midterm Exam #1

This midterm exam is open-book, open-note, but closed-computer. Answer each of the questions included in the exam. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit. In all questions, you may include functions, classes, or other definitions that have been developed in the course, either by writing the **`#include`** line for the appropriate header or by giving the name of the function or class and the handout, chapter number, or lecture in which that definition appears.

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 to 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 specifically requires an efficient solution.

SUNetID:	________________________
Last Name:	________________________
First Name:	________________________

I accept both the letter and the spirit of the Stanford Honor Code. I have not received any assistance on this test, nor will I give any. My answers represent my own work.

(signed) ____

You have three hours to complete this midterm. There are 180 total points, and this midterm is worth 20% of your total grade in this course. As a rough measure of the relative difficulty of the problems, there is one point on this exam per minute of testing time.

Question		**Points**	**Grader**
(1) Pet Matchmaking	25	/ 25	
(2) Shrinkable Words Revisited	40	/ 40	
(3) Balanced Parentheses	45	/ 45	
(4) The Hungry Frog	45	/ 45	
(5) Palindrome Efficiency	25	/ 25	
	(180)	**/ 180**	

Good Luck!

Problem One: Pet Matchmaking (25 Points)

Suppose that you own a pet store and want to help customers find their ideal pet. To do so, you create a list of all the pets in the store, along with all of the adjectives which best describe them. For example, you might have this list of pets and their adjectives:

Ada	Happy, Loving, Fuzzy, Big, Tricksy
Babbage	Loving, Tiny, Energetic, Quiet
Lucy	Happy, Loving, Big
Larry	Happy, Fuzzy, Tiny, Tricksy
Abby	Loving, Big, Energetic
Luba	Happy, Loving, Tiny, Quiet, Tricksy
Mitzy	Fuzzy, Big, Energetic, Quiet

If a customer gives you a list of adjectives, you can then recommend to them every pet that has all of those adjectives. For example, given the adjectives "Happy" and "Tricksy," you could recommend Ada, Larry, and Luba. Given the adjective "Fuzzy," you could recommend Ada, Larry, and Mitzy. However, given the adjectives "Energetic," "Quiet," and "Fuzzy," and "Tiny," you could not recommend any pets at all.

Write a function

```
Set<string> allPetsMatching(Map<string, Set<string> >& adjectiveMap,
                            Vector<string>& requirements);
```

that accepts as input a `Map<string, Set<string> >` associating each pet with the adjectives that best describe it, along with a customer's requested adjectives (represented by a `Vector<string>`), then returns a `Set<string>` holding all of the pets that match those adjectives.

There might not be any pets that match the requirements, in which case your function should return an empty set. You can assume that all adjectives have the same capitalization, so you don't need to worry about case-sensitivity. Also, if a client does not include any adjectives at all in their requirements, you should return a set containing all the pets, since it is true that each pet matches all zero requirements. Finally, your function should not modify any of the input parameters.

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

```
Set<string> allPetsMatching(Map<string, Set<string> >& adjectiveMap,
                            Vector<string>& requirements) {
```

Problem Two: Shrinkable Words Revisited (40 Points)

Recall from lecture that a *shrinkable word* is a word that can be reduced down to a single letter by removing one letter at a time, at each step leaving a valid English word.

In lecture, we wrote a function **`isShrinkableWord`** that determined whether a given string was a shrinkable word. Initially, this function just returned **`true`** or **`false`**. This meant that if a word was indeed shrinkable, we would have to take on faith that the word could indeed be reduced down to a single letter.

To help explain *why* a word was shrinkable, our second version of the function additionally produced a **`Stack<string>`** showing the series of words one would go through while shrinking the word all the way down to a single letter. Let's call a sequence of this sort a ***shrinking sequence***.

However, let's suppose that you're *still* skeptical that the **`Stack<string>`** produced by the **`isShrinkableWord`** function actually is a legal shrinking sequence for the word. To be more thoroughly convinced that a word is shrinkable, you decide to write a function that can check whether a given **`Stack<string>`** is indeed a shrinking sequence for a given word.

Write a function

```
bool isShrinkingSequence(string word, Lexicon& words, Stack<string> path);
```

that accepts as input a word, a **`Lexicon`** containing all words in English, and a **`Stack<string>`** containing an alleged shrinking sequence for that word, then returns whether the **`Stack<string>`** is indeed a legal shrinking sequence for that word. For example, given the word **`"pirate"`** and the stack

```
 pirate
  irate
   rate
   rat
    at
    a
_________
```

Your function would return **`true`**. However, given any of these stacks:

```
  pirate
   irate        avast
    rate         vast
    ate          vat         pirate
     te           at          rate
     e            a            at
  _______      _______      _______      _______
```

Your function would return **`false`** (the first stack is not a shrinking sequence because **`"te"`** and **`"e"`** are not words; the second is a legal shrinking sequence, but not for the word **`"pirate"`**; the third is not a shrinking sequence because it skips words of length 1, 3, and 5; and the fourth is not a shrinking sequence because it contains no words at all).

You can assume that **`word`** and all the words in **`path`** consist solely of lower-case letters.

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

```
bool isShrinkingSequence(string word, Lexicon& words, Stack<string> path) {
```

Problem Three: Balanced Parentheses (45 Points)

Your job in this problem is to write a function

```
Vector<string> balancedStringsOfLength(int n);
```

that accepts as input a nonnegative number **`n`**, then returns a **`Vector<string>`** containing all strings of exactly **`n`** pairs of balanced parentheses.

As examples, here is the one string of one pair of balanced parentheses:

```
()
```

Here are the two strings of two pairs of balanced parentheses:

```
(())     ()()
```

Here are all five strings of three pairs of balanced parentheses:

```
((()))     (()())     (())()     ()(())     ()()()
```

And here are the fourteen strings of four pairs of balanced parentheses:

`(((())))`	`(()()())`	`(())()()`	`()()(())`
`((()()))`	`((()))()`	`()((()))`	`()()()()`
`((())())`	`(()())()`	`()(()())`	
`(()(()))`	`(())(())`	`()(())()`	

As a hint, you might find the following observation useful: any string of $n > 0$ pairs of balanced parentheses can be split apart as follows:

(*some-string-of-balanced-parentheses* **)** *another-string-of-balanced-parentheses*

You might find it useful to try splitting apart some of the above strings this way to see if you understand why this works.

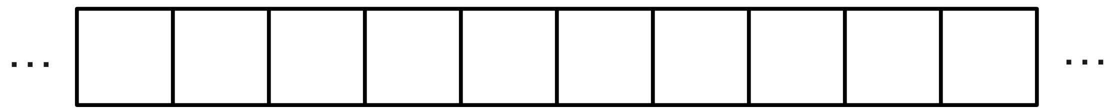
Your solution can return the strings of balanced parentheses in any order, as long as every string of balanced parentheses appears exactly once in the resulting **`Vector`**.

Write your solution on the next page, and feel free to tear out this page as a reference.

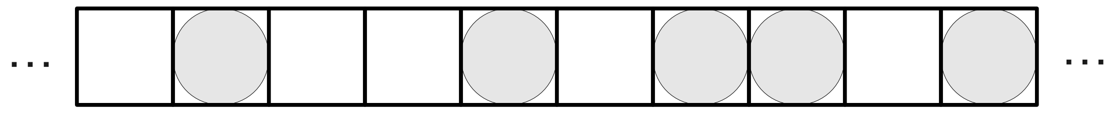
```
Vector<string> balancedStringsOfLength(int n) {
```

Problem Four: The Hungry Frog (45 Points)

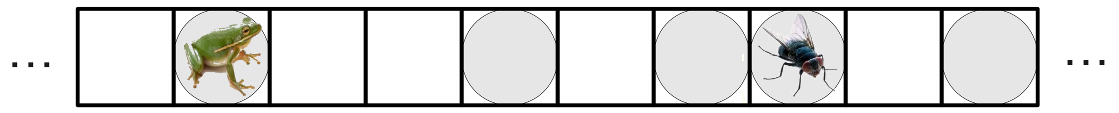
In the Land of Simplifying Assumptions, there is an infinitely long pond divided into a sequence of squares, as shown below:



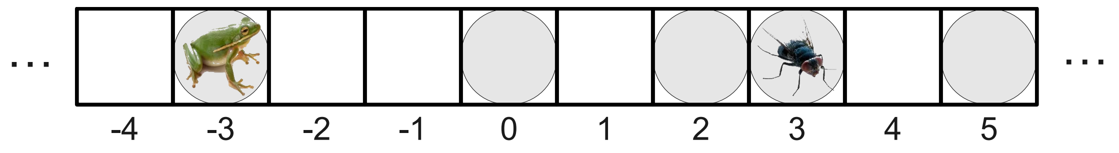
Some of these squares have lily-pads in them, while the rest are all water. We'll represent the lily-pads as gray circles:



One of the lily-pads has a frog on it, and another has a fly on it:



For simplicity, let's number all of the (infinitely many) squares by assigning them consecutive integers. For example, we might number the squares like this:



Our frog is capable of jumping from lily-pad to lily-pad and ultimately wants to get the lily-pad containing the fly. However, she is restricted in the size and number of jumps she can make. Specifically, the frog is given a list of jump sizes, and can only use each listed jump size once. Additionally, the frog can only jump on lily-pads and must not fall into the water.

For example, in the above pond, if the frog has jump sizes ⟨2, 3, 5⟩, then she could get to the fly by

- making a jump of size 3 from lily-pad -3 to lily-pad 0,
- making a jump of size 5 from lily-pad 0 to lily-pad 5, and
- making a jump of size 2 from lily-pad 5 back to lily-pad 3.

Similarly, given jump sizes ⟨1, 2, 3, 4, 5⟩, the frog could get to the fly by

- making a jump of size 5 from lily-pad -3 to lily-pad 2, then
- making a jump of size 1 from lily-pad 2 to lily-pad 1.

And given jump sizes ⟨3, 3⟩, the frog can get to the fly by

- making a first jump of size 3 from lily-pad -3 to lily-pad 0, and
- making a second jump of size 3 from lily-pad 0 to lily-pad 3

However, the frog could not get to the fly with jump sizes ⟨2, 4⟩, because making a jump of either of those sizes would put the frog into the water. The frog also could not get to the fly given jump sizes ⟨3⟩, because the frog cannot get to the fly in a single jump of size 3.

Your job is to write a function

```
          bool canFrogEatFly(int frogPos, int flyPos,
                             Vector<int>& jumpSizes,
                             Set<int>& lilypads);
```

That accepts as input the starting positions of the frog and the fly, the frog's available jump sizes, and the positions of the lily-pads, then returns whether the frog can get to the lily-pad holding the fly.

When writing up your solution, keep in mind that

- The frog does not need to use all of the jumps available to her.
- The frog can make the jumps in any order, not just the order in which they're provided in **jump-Sizes**.
- The frog can take each jump either forwards or backwards.
- Each jump size in the list can be used at most once.

Feel free to tear out the preceding page as a reference, and write your answer below.

```
bool canFrogEatFly(int frogPos, int flyPos,
                   Vector<int>& jumpSizes,
                   Set<int>& lilypads) {
```

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

Problem Five: Palindrome Efficiency (25 Points)

In lecture, we wrote the following recursive function to determine whether a string is a palindrome:

```
bool isPalindrome(string word) {
    if (word.length() <= 1) return true;
    if (word[0] != word[word.length() - 1]) return false;
    return isPalindrome(word.substr(1, word.length() - 2));
}
```

This function is elegant, but contains several sources of inefficiency. Note that the **word** parameter is passed by value, meaning that the recursive call makes a full copy of its argument. In general, it takes time O(n) to make a copy of a n-character string. Additionally, the **substr** function takes time O(n) to complete, where n is the number of characters in the generated substring. However, it only takes O(1) time to access a character in a string using **[]**, and the **length** function only takes time O(1).

(i) Analyzing the Efficiency (17 Points)

Let n be the length of the string given to **isPalindrome** as input. What are the ***best case*** and ***worst-case*** big-O time complexities of the **isPalindrome** function? Justify your answer.

To address these efficiency concerns, we will need to modify the function by

1. Making the function take its argument by reference, and
2. Avoiding calls to **`substr`**.

We can address issue (1) easily by changing the function signature. We can address issue (2) in the following way: when making the recursive call, we will pass in the input string unchanged. However, we will also pass two extra parameters into the recursive function indicating the start and end positions within the master string that the function should consider. This is shown here:

```
bool recIsPalindrome(string& word, int lhs, int rhs) {
    if (rhs <= lhs) return true;
    if (word[lhs] != word[rhs]) return false;
    return recIsPalindrome(word, lhs + 1, rhs - 1);
}

bool isPalindrome(string& word) {
    return recIsPalindrome(word, 0, word.length() - 1);
}
```

This code is more complicated than the original version. Are these changes actually worth it?

(ii) Analyzing the Efficiency (Again) (8 Points)

Let *n* be the length of the string given to the above **`isPalindrome`** as input. What are the ***best case*** and ***worst-case*** big-O time complexities of the **`isPalindrome`** function? Justify your answer.