CS 106B Practice Midterm Exam #6

(based on CS 106B Summer 2014 midterm)

This sample exam is intended to demonstrate an example of some of the kinds of problems that will be asked on the actual midterm exam. We do not guarantee that the number of questions in this sample exam will match the number on the real exam, nor that every kind of problem shown here will exactly match the kinds of problems shown on the final exam, though the real exam will be generally similar overall. Also, to save paper, this exam does not provide blank pages for writing space. The real exam will provide more blank space for you to write your answers.

In general, problem solutions are not graded on "style" as long as they work properly, though many questions do have some rules and constraints. For example, you should not use global variables, and some problems require recursion or a particular data structure.

1. C++ Basics and Parameters (read)

The following code C++ uses parameters/returns and produces **four lines** of output. What is the output?

```
string parameterMystery6(int a, int& b, string& c, string d) {
    a++;
    b++;
    c += "x";
    d += "x";
    cout << d << " " << c << " " << b << " " << a << endl;
    return d;
}

int main() {
    int a = 1;
    int b = 20;
    int x = 300;
    string c = "c";
    string d = "d";
    string z = "z";

    parameterMystery6(a, b, c, d);
    parameterMystery6(b, x, z, c);
    d = parameterMystery7(x, a, c, z);

    cout << a << " " << b << " " << x << " " << c << " " << d << " " << z
         << endl;
    return 0;
}
```

2. File I/O and Strings (write)

Write a function named **printMostCommonName** that accepts as its parameter a reference to an input stream (**ifstream**) representing a file of input. The corresponding file contains data that is a sequence of first names separated by white-space. Some names might occur more than once in the data; all occurrences of a given name will appear consecutively in the file. Your function's job is to **print** the name that occurs the most frequently in the file, along with how many times it occurs. You should also **return** the total number of unique names that were seen in the entire file.

If two or more names occur the same number of times, print the one that appears earlier in the file. If every name in the file is different, every name will have 1 occurrence, so you should just print the first name in the file.

For example, if the file `names1.txt` contains the following text:

```
Benson
Eric   Eric  Marty
Kim  Kim Kim   Jenny  Nancy
Nancy  Nancy  Paul  Paul
```

Then a call of `printMostCommonName` on an input stream for this file should produce the following output and should return `7`, because there are 7 unique names in the file:

```
Most common name: Kim, 3
```

This is because in this data, there is one occurrence of the name Benson, two occurrences of Eric, one occurrence of Marty, three occurrences of Kim, one of Jenny, three of Nancy, and two of Paul. Kim and Nancy appear the most times (3), and Kim appears first in the file. So for that line, your function should print that the most common is Kim.

As a second example, if the file `names2.txt` contains the following text:

```
 Stuart   Stuart
Stuart Ethan Alyssa  Alyssa Helene   Jessica
  Jessica Jessica    Jessica
```

Then a call of `printMostCommonName` on an input stream for this file should produce the following output and return `5`:

```
Most common name: Jessica, 4
```

As a third example, if the file `names3.txt` contains the following text:

```
Jared  Alisa

Yuki   Catriona
 Cody   Coral   Trent
Kevin  Ben Stefanie Kenneth
```

Notice that in this input data, every name is unique. So a call of `printMostCommonName` on an input stream for this file should produce the following output and should return `11`, because there are 11 unique names:

```
Most common name: Jared, 1
```

You may **assume** that the input stream has already been opened and that it refers to a valid input file that exists on the user's disk. You may assume that the input file will contain at least one name. But notice that the names might be separated by multiple spaces or lines, and that some lines might be blank. Your code should process the data properly regardless of the spacing between tokens. Each name will be separated by at least one whitespace character.

Constraints: For full credit, your solution should read the file only once, not make multiple passes over the file data. You may use data structures in your solution if you like, but it is possible/intended to solve it without any data structures.

---

3. ADTs / Collections (read)

Write the output produced by the following function when passed each of the following stack:

```
void collectionMystery6(Stack<int>& s) {
    Queue<int> q;
    Stack<int> s2;

    while (!s.isEmpty()) {
        if (s.peek() % 2 == 0) {
            q.enqueue(s.pop());
        } else {
            s2.push(s.pop());
        }
    }

    while (!q.isEmpty()) {
        s.push(q.dequeue());
    }
    while (!s2.isEmpty()) {
        s.push(s2.pop());
    }

    cout << s << endl;
}
```

Stack	Output
a) {1, 2, 3, 4, 5, 6}	
b) {42, 3, 12, 15, 9, 71, 88}	
c) {65, 30, 10, 20, 45, 55, 6, 1}	

4. ADTs / Collections (write)

In the English language, some combinations of adjacent letters are more common than others. For example, 'h' often follows 't' ("th"), but rarely would you see 'x' following 't' ("tx"). Knowing how often a given letter follows other letters in the English language is useful in many contexts. In cryptography, we use this data to crack substitution ciphers (codes where each letter has been replaced by a different letter, for example: A→M, B→T, etc.) by identifying which possible decoding substitutions produce plausible letter combinations and which produce nonsense.

For this problem, write a function named **`pairFrequencies`** that accepts a reference to a `Lexicon` representing a dictionary of words. Your function will examine the dictionary and print all 2-character sequences of letters along with a count of how many times each pairing occurs. For example, suppose the dictionary contains the following words:

```
{"banana", "bends", "i", "mend", "sandy"}
```

This dictionary contains the following two-character pairs: "ba", "an", "na", "an", "na" from banana; "be", "en", "nd", "ds" from bends; "me", "en", "nd" from mend; and "sa", "an", "nd", "dy" from sandy. (Note that "i" is only one character long, so it contains no pairs.) So your function would print the following output:

```
an: 3
ba: 1
be: 1
ds: 1
dy: 1
en: 2
me: 1
na: 2
nd: 3
sa: 1
```

Notice that pairings that occur more than once in the same word should be counted as separate occurrences. For example, "an" and "na" each occur twice in "banana".

Constraints: For full credit, obey the following restrictions. A solution that disobeys them can get partial credit.

- You may create **one additional data structure** (stack, queue, set, map, etc.) as auxiliary storage.
  A nested structure, such as a set of vectors, counts as one additional data structure.
  (You can have as many simple variables as you like, such as `int`s or `string`s.)
- Your solution should run in $O(N^2)$ time or faster, where N is the number of words in the lexicon.
- You should loop over the contents of the `Lexicon` no more than once.
- You should not modify the contents of the lexicon that is passed to your function.

---

5. Algorithm Analysis and Big-Oh (read)

Give a tight bound of the nearest runtime complexity class for each of the following code fragments in Big-Oh notation, in terms of variable *N*. (Write the growth rate as *N* grows.) Write a simple expression that gives only a power of *N*, such as $O(N^2)$ or $O(\log N)$, *not* an exact calculation like $O(2N^3 + 4N + 14)$. Write your answer in the blanks on the right side.

Question	Answer

a)
`int sum = 0;`
`for (int i = 1; i <= N; i++) {`
`  int k = 4000;`
`  for (int j = 1; j <= k; j++) {`
`  sum++;`
`  }`
`}`
`cout << sum << endl;`
	$O($_____$)$

b)
`int sum = 0;`
`for (int i = 1; i <= N; i++) {`
`  sum += 2;`
`}`
`for (int i = 1; i <= N; i++) {`
`  for (int j = 1; j <= N * N; j++) {`
`  sum++;`
`  }`
`}`
`cout << sum << endl;`
	$O($_____$)$

c)
`Stack<int> stack;`
`for (int i = 1; i <= N; i++) {`
`  stack.push(i);`
`}`
`Set<int> set;`
`while (!stack.isEmpty()) {`
`  int k = stack.pop();`
`  set.add(k);`
`}`
`cout << "done!" << endl;`
	$O($_____$)$

d)
`HashMap<int, int> map1;`
`for (int i = 1; i <= N; i++) {`
`  map1.put(i, i);`
`}`
 	
`Map<int, int> map2;`
`for (int i = 1; i <= N; i++) {`
`  int k = map1.get(i);`
`  map2.put(k, k);`
`  map2.put(n + k, n + k);`
`}`
`cout << "done!" << endl;`
	$O($_____$)$

e)
`Vector<int> v;`
`for (int i = 1; i <= N; i++) {`
`  for (int j = 1; j <= N; j++) {`
`  v.insert(0, i);`
`  }`
`  v.clear();`
`}`
`while (!v.isEmpty()) {`
`  v.remove(0);`
`}`
`cout << "done!" << endl;`
	$O($_____$)$

6. Recursion (read)

For each of the calls to the following recursive function below, indicate what value is returned:

```
string recursionMystery6(string s) {
    if (s.length() == 0) {
        return s;
    } else if (s.length() % 2 == 0) {
        string rest = s.substr(0, s.length() - 1);
        string last = s.substr(s.length() - 1, 1);
        return last + recursionMystery6(rest);
    } else {
        string first = s.substr(0, 1);
        string rest = s.substr(1);
        return "(" + first + ")" + recursionMystery6(rest);
    }
}
```

Call	Output
a) `recursionMystery6("hi");`	
b) `recursionMystery6("quirk");`	
c) `recursionMystery6("computer");`	

7. Recursion (write)

Write a recursive function named **printRange** that accepts integer parameters *x* and *y* and prints the sequential integers between *x* and *y* inclusive. The first half should be printed with the greater-than character (">") separating consecutive values. The second half should be printed with the less-than character ("<") separating consecutive values. When there are two values in the middle of the range, those two values should be separated by a pair of dashes ("--"), as shown in the second example output below. The following table shows several calls and their expected output:

Call	Output
`printRange(1, 9);`	`1 > 2 > 3 > 4 > 5 < 6 < 7 < 8 < 9`
`printRange(1, 10);`	`1 > 2 > 3 > 4 > 5 -- 6 < 7 < 8 < 9 < 10`
`printRange(23, 29);`	`23 > 24 > 25 > 26 < 27 < 28 < 29`
`printRange(13, 14);`	`13 -- 14`
`printRange(-8, -8);`	`-8`

Notice that in the first output, 5 is in the middle with the numbers before it separated by greater-than and the numbers after it separated by less-than. In the second output, 26 is in the middle with numbers before it separated by greater-than and numbers after it separated by less-than. The last output has no separators because that range includes one number.

Your function should throw an integer **exception** if *x* is greater than *y*.

Constraints: Do not declare any **global variables**. Also, **do not use any loops**; you must use recursion. You are allowed to define other "**helper**" functions if you like; they are subject to these same constraints.

---

8. Recursive Backtracking (write)

Write a recursive function named **`squishWord`** that uses backtracking to try to find a way to "squish" a word down to an empty string, one letter at a time. For this problem, "squishing" a word is defined as finding a way to repeatedly remove a single letter from the word, forming another string that is a valid English word at each step, until the string is empty. For example, here is a sequence of removals that can squish the word `"cart"`:

```
{"cart", "cat", "at", "a"}
```

Some surprisingly long words can be successfully "squished". For example, here is a sequence for `"startling"`:

```
{"startling", "starting", "staring", "string", "sting", "sing", "sin", "in", "i"}
```

Note that you cannot just remove any letter; if you removed the `"a"` in the first step, you'd get `"strtling"`, which is not a valid English word. Each word along the way must be an English word from the dictionary for the sequence to be valid.

Your function accepts **two parameters**: a string representing the word to try to "squish", and a reference to a `Lexicon` representing the English dictionary. If your function is able to find a valid "squish" sequence, your function should print it in the format shown in the two examples above: each word in the sequence is shown in order, starting with the original word passed in and ending with the empty string. (Including the empty string in the output is optional; you may include or omit it if you like.) Assume that the dictionary has already been read from a file and contains all legal English words. You can use it like a `Set` to check whether a given word is found in the English dictionary. Recall that a `Lexicon` is case-insensitive; you do not need to worry about any case sensitivity issues for this problem.

If there are multiple ways to "squish" the word, you should find and **print only one** of them. After printing one path, your code should stop without printing any other potential paths.

If the string passed to your function is not a single valid English word, or if it is not possible to produce to "squish" the given string, your function should print:

```
No sequence found.
```

Constraints: Do not declare any **global variables**. You can use any data structures you like, and your code can contain loops, but the overall algorithm must be recursive and must use backtracking. You are allowed to define other "**helper**" functions if you like; they are subject to these same constraints.

---

9. Implementing a Collection Class (write)

(That quarter's exam did not have an "implementing a collection class" problem.)

---

10. Pointers and Linked Nodes (write)

(That quarter's exam did not have an "Pointers and Linked Nodes" problem.)
