Practice Midterm 2

Based on a handout by Eric Roberts

Problem 1: Tracing C++ programs and big-O (10 points)

Assume that the function **puzzle** has been defined as follows:

```
int puzzle(int n) {
   if (n == 0) {
      return 0;
   } else {
      return puzzle(n - 1) + 1 + puzzle(n - 1);
   }
}
```

(a) What is the value of **puzzle(4)**?

(b) What is the computational complexity of the **puzzle** function expressed in terms of big-O notation, where *N* is the value of the argument **n**. In this problem, you may assume that **n** is always a nonnegative integer and that all arithmetic operators execute in constant time.

Problem 2: Vectors, grids, stacks, and queues (10 points)

The **`resize`** method in the **`Grid`** class resets the dimensions of the grid but also initializes every element of the grid to its default value. Write a function

```
void reshape(Grid<int> & grid, int nRows, int nCols);
```

that resizes the grid but fills in the data from the original grid by copying elements in the standard row-major order (left-to-right/top-to-bottom). For example, if **`myGrid`** initially contains the values

1	2	3	4
5	6	7	8
9	10	11	12

calling the function

```
reshape(myGrid, 4, 3)
```

should change the dimensions and contents of **`myGrid`** as follows:

1	2	3
4	5	6
7	8	9
10	11	12

If the new grid does not include enough space for all of the original values, the values at the bottom of the grid are simply dropped. For example, if you call

```
reshape(myGrid, 2, 5)
```

there is no room for the last two elements, so the contents of the grid will look like this:

1	2	3	4	5
6	7	8	9	10

Conversely, if there are not enough elements in the original grid to fill the available space, the entries at the end should simply retain the values they have after executing **`resize`**.

Problem 3: Lexicons, maps, and iterators (15 points)

For the first fifty years of the early 19th century, a London-born physician named Peter Mark Roget worked tirelessly to create a comprehensive catalogue of English words and their synonyms. The result of that work was Roget's Thesaurus, which first appeared in 1852 and which has remained in print ever since.

Suppose (perhaps as part of a future RandomWriter Contest project) that you want to create a C++ data structure that makes it easy to find synonyms for a specified word. The synonym list is stored in a data file that might begin something like this (all of these words are listed as synonyms in section 1 of the current edition of Roget's Thesaurus):

RogetSynonyms.txt

```
being existence
actuality reality
be exist
abide continue endure last remain stay
actual absolute real true
    .
    .
    .
```

Each line in the file consists of a sequence of synonyms separated by spaces. For example, the first line links the words *being* and *existence* as synonyms. The fourth line defines six words—*abide, continue, endure, last, remain,* and *stay*—as a new synonym category.

Implement a method

```
void readSynonymTable(ifstream& infile,
                      Map<string,Lexicon>& table);
```

that reads the specified file and constructs a table in which each word appearing in the file is mapped to a **`Lexicon`** containing the other words on the line. For example, if the file **`RogetSynonyms.txt`** has been opened as the input stream **`infile`**, calling

```
readSynonymTable(infile, synonymTable);
```

should initialize **`synonymTable`** so that each word is bound to a lexicon of its synonyms. For example, the entry for **`"endure"`** in **`synonymTable`** should be a **`Lexicon`** containing the strings **`"abide"`**, **`"continue"`**, **`"last"`**, **`"remain"`**, and **`"stay"`**. In much the same way, the entry for **`"exist"`** should be a **`Lexicon`** containing only the string **`"be"`**.

In working on this problem, you should not waste time trying to find an efficient solution. The steps you need to implement are reasonably straightforward. For each line in the file, use a scanner to separate out the words in the line. Then, for each word in the line, create an empty lexicon and add all the synonyms (the other words in that line) to the lexicon. The resulting lexicon contains all of the current word's synonyms and can therefore be stored as the value in the map for the current word.

As you write this function, you may make the following assumptions:

- Opening and closing the input file is the caller's responsibility.
- Each line in the file consists of a series of words, all in lowercase, separated by spaces.
- Each word appears in only one list of synonyms.

Problem 4: Recursive functions (10 points)

> *The waste of time in spelling imaginary sounds and their history (or etymology as it is called) is monstrous in English . . .*
>
> —George Bernard Shaw, 1941

In the early part of the 20th century, there was considerable interest in both England and the United States in simplifying the rules used for spelling English words, which has always been a difficult proposition. One suggestion advanced as part of this movement was the removal of all doubled letters from words. If this were done, no one would have to remember that the name of the Stanford student union is spelled "Tresidder," even though the incorrect spelling "Tressider" occurs at least as often. If doubled letters were banned, everyone could agree on "Tresider."

Write a **recursive** function

```
string removeDoubledLetters(string str);
```

that takes a string as its argument and returns a new string with any consecutive substring consisting of repeated copies of the same letter replaced by a single copy letter of that letter. For example, if you call

```
removeDoubledLetters("tresidder")
```

your function should return the string `"tresider"`. Similarly, if you call

```
removeDoubledLetters("bookkeeper")
```

your method should return `"bokeper"`. And because your function compresses strings of multiple letters into a single copy, calling

```
removeDoubledLetters("xxx")
```

should return `"x"`.

In writing your solution, you should keep the following points in mind:

- You do not need to write a complete program. All you need is the definition of the function `removeDoubledLetters` that returns the desired result.
- Your function should not try to consider the case of the letters. For example, calling the function on the name `"Lloyd"` should return the argument unchanged because `'L'` and `'l'` are different letters.
- Your function must be purely recursive and may not make use of any iterative constructs such as `for` or `while`.

Problem 5: Recursive procedures (15 points)

In CS 106A, I usually include a problem on the midterm exam to check whether students understand the precedence of operators. For example, I might ask students to evaluate the expression

```
9 + 7 * 5 - 3 + 1
```

To do so, those students would have to know that the multiplication was performed first. Of course, I also want to make sure that the answer comes out to be some recognizable value. In this case, for example, the expression evaluates to 42, which is "the answer to the ultimate question of life, the universe, and everything" from Douglas Adams's *Hitchhiker's Guide to the Galaxy.*

Generating such examples by hand is sufficiently difficult that it makes sense to use the power of recursion to solve it algorithmically. Your job in this problem is to write a function

```
void tryAllOperators(string exp, int target);
```

in which **`exp`** is an expression string in which the operators have been replaced by question marks and **`target`** is the desired value. The function operates by replacing those question marks with the four standard arithmetic operators (**`+`**, **`-`**, **`*`**, **`/`**), trying every possible combination to see if any produce the desired target value. For example, calling

```
tryAllOperators("9 ? 7 ? 5 ? 3 ? 1", 42)
```

should produce the following output, since that is the only combination of operators that gives 42 as a result:

```
9 + 7 * 5 - 3 + 1
```

If there is more than one way to produce the target value, the function should list all of them. Thus, calling

```
tryAllOperators("2 ? 3 ? 5 ? 7 ? 11 ? 13", 42)
```

should produce the following output, since there are two ways of achieving 42:

```
2 + 3 + 5 * 7 - 11 + 13
2 * 3 + 5 + 7 + 11 + 13
```

Writing this program would be very difficult if you had to write the code to evaluate an expression. We will do precisely that later in the quarter, but for now, you should assume that you have a function

```
int evaluateExpression(string exp);
```

that takes an arithmetic expression involving integers and the four arithmetic operators and returns the calculated value. Given this function, all you have to do for this problem is

1. Recursively generate every possible expression by replacing the question marks in the input string with each of the actual operators in turn.
2. Call **`evaluateExpression`** for each of those generated expressions.
3. Print out the expression if the result of **`evaluateExpression`** equals the target value.