Practice Midterm 1

Based on a handout by Eric Roberts

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 Thursday, May 3. A solution set to this practice exam will be handed out on Wednesday.

Time and place of the exam

The midterm exam is scheduled for a two-hour block at three different locations (note that the exams are not in the regular lecture room), divided by last name. Although the syllabus lists the time as 7:00PM to 10:00PM, the exam is only two hours long.

Coverage

The midterm covers the material presented in class through the lecture on Friday, April 27, which means that you are responsible for the chapters in the text through Chapter 10 ("Algorithmic Analysis").

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 120. We intend for the number of points to be roughly comparable to the number of minutes you should spend on that problem. This leaves you with an hour to check your work or recover from false starts.

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.

Problem 1: Tracing C++ programs and big-O (20 points)

Assume that the functions **`Mystery`** and **`Enigma`** have been defined as follows:

```
int mystery(int n) {
   if (n == 0) {
      return 1;
   } else {
      return enigma(2, mystery(n - 1));
   }
}

int enigma(int n1, int n2) {
   if (n1 == 0) {
      return 0;
   } else {
      return n2 + enigma(n1 - 1, n2);
   }
}
```

(a) What is the value of **`mystery(3)`**?

(b) What is the computational complexity of the **`mystery`** 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.

Problem 2: Vectors, grids, stacks, and queues (20 points)

The figures in my books are generated by creating pictures in PostScript®, a powerful graphics language developed by the Adobe Corporation in the early 1980s. PostScript programs store their data on a stack. Many of the operators available in the PostScript language have the effect of manipulating the stack in some way. You can, for example, invoke the **`pop`** operator, which pops the top element off the stack, or the **`exch`** operator, which swaps the top two elements.

One of the most interesting (and surprisingly useful) PostScript operators is the **`roll`** operator, which takes two arguments: *n* and *k*. The effect of applying **`roll(`***n*`, `*k*`)`** is to rotate the top *n* elements of a stack by *k* positions, where the general direction of the rotation is toward the top of the stack. More specifically, **`roll(`***n*`, `*k*`)`** has the effect of removing the top *n* elements, cycling the top element to the last position *k* times, and then replacing the reordered elements back on the stack. Figure 1 at the bottom of the page shows before and after pictures for three different examples of **`roll`**.

Your job in this problem is to write a function

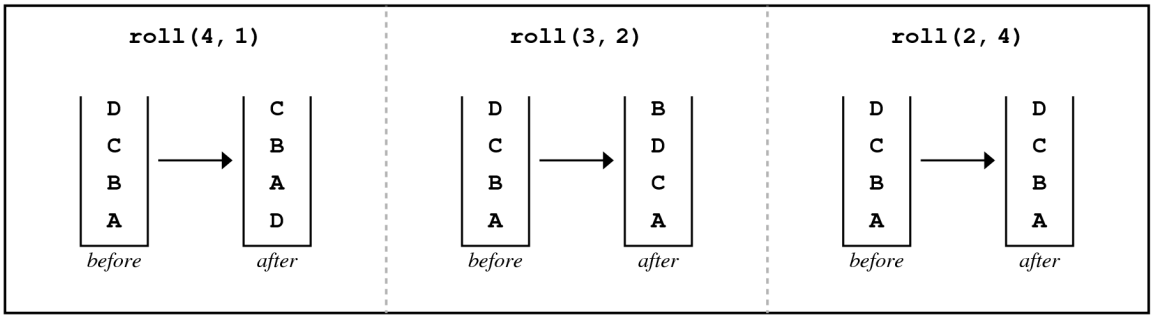
```
void roll(Stack<char> & s, int n, int k)
```

that implements the **`roll(`***n*`, `*k*`)`** operation on the character stack **`s`**. In doing so, you will probably find it useful to use other structures (stacks, queues, vectors, and so forth) as temporary storage. Your implementation should check to make sure that **`n`** and **`k`** are both nonnegative and that **`n`** is not larger than the stack size; if either of these conditions is violated, your implementation should call **`error`** with the message

```
roll: argument out of range
```

Note, however, that **`k`** can be larger than **`n`**, in which case the **`roll`** operation continues through more than a complete cycle. This case is illustrated in the final example in Figure 1, in which the top two elements on the stack are rolled four times, leaving the stack exactly as it started.

Figure 1. Three examples of the roll operator



Problem 3: Lexicons, maps, and sets (30 points)

Entering text using a phone keypad is problematic, because there are only 10 digits for 26 letters. As a result, each of the digit keys (other than 0 and 1, which could not be part of telephone exchange prefixes until about thirty years ago) is mapped to several letters, as shown in the following diagram:

Some cell phones use a "multi-tap" user interface, in which you tap the 2 key once for **a**, twice for **b**, and three times for **c**, which can get tedious. A streamlined alternative is to use a predictive strategy in which the cell phone guesses which of the possible letter you intended, based on the sequence so far and its possible completions.

For example, if you type the digit sequence 72, there are 12 possibilities: **pa**, **pb**, **pc**, **qa**, **qb**, **qc**, **ra**, **rb**, **rc**, **sa**, **sb**, and **sc**. Only four of these—**pa**, **ra**, **sa**, and **sc**—seem promising because they are prefixes of common English words like **party**, **radio**, **sandwich**, and **scanner**. The others can be ignored because there are no common words that begin with those sequences of letters. If the user enters 9956, there are 144 (4 x 4 x 3 x 3) possible letter sequences, but you can be assured the user meant **xylo** since that is the only sequence that is a prefix of any English words.

Write an **iterative** function

```
Vector<string> listCompletions(string digits, Lexicon& lex);
```

that returns a vector of all words from the lexicon that can be formed by extending the given digit sequence. For example, here are all the completions for 72547:

- palisade
- palisaded
- palisades
- palisading
- palish
- rakis
- rakish
- rakishly
- rakishness
- sakis

Problem 4: Recursive functions (20 points)

As you saw in Chapter 1, it is easy to implement an iterative function **raiseIntToPower** that computes **n** raised to the **k**th power:

```
int raiseIntToPower(int n, int k) {
   int result = 1;
   for (int i = 0; i < k; i++) {
      result *= n;
   }
   return result;
}
```

Rewrite this function so that it operates recursively, taking advantage of the following insight:

- If **k** is even, n^k is the square of **n** raised to the power **k** / 2.
- If **k** is odd, n^k is the square of **n** raised to the power **k** / 2 times **n**.

In solving this problem, you need to identify the simple cases necessary to complete the recursive definition. You must also make sure that your code is efficient in the sense that it makes only one recursive call per level of the recursive decomposition.

Problem 5: Recursive procedures (30 points)

The game of dominos is played with rectangular pieces composed of two connected squares, each of which is marked with a certain number of dots. For example, each of the following five rectangles represents a domino:

Dominos can be connected end-to-end to form chains, subject to the condition that two dominos can be linked together only if the numbers match. For example, you can form a chain consisting of all five of these dominos by connecting them in the following order:

As in the traditional game, dominos can be rotated by 180˚ so that their numbers are reversed. In this chain, for example, the 1-6 and 3-4 dominos have been "turned over" so that they fit into the chain.

Dominos can be represented in C++ using the following structure type:

```
struct Domino {
   int leftDots;
   int rightDots;
};
```

Given this domino type, write a recursive function

```
bool formsDominoChain(Vector<Domino>& dominos);
```

that returns **`true`** if it possible to build a chain consisting of every domino in the vector.

For example, if you initialized the variable **`myDominos`** to contain the five **`dominoT`** values shown at the top of the page, calling **`formsDominoChain(myDominos)`** would return **`true`** because it is possible to form the chain shown in the second diagram. On the other hand, if you remove the first domino, calling **`formsDominoChain(myDominos)`** returns **`false`** because there is no way to form a domino chain if the 1-4 domino is missing.