Eric Roberts
CS 106B

Handout #26
February 4, 2015

Practice Midterm Exam #1

Review session: Sunday, February 8, 7:00–9:00 P.M., Bishop Auditorium
Midterm #1: Tuesday, February 10, 2:15–4:15 P.M., Annenberg Auditorium
Midterm #2: Tuesday, February 10, 7:00–9:00 P.M., CEMEX Auditorium

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, February 10. A solution set to this practice exam will be handed out on Friday along with a second practice exam.

In addition to these practice exams, the handouts section of the web site contains a guide (Handout #26J) written by my colleague Julie Zelenski that provides useful strategic advice for taking exams in CS 106B.

Time and place of the exam

The midterm exam is scheduled for a two-hour block at two different times and places (note that the exams are not in the regular lecture room). You may take the exam at either time and need not give advance notice of which exam you plan to take. If you are unable to take the exam at either of the scheduled times or if you need special accommodations, please send an email message to **`eroberts@cs`** stating the following:

- The reason you cannot take the exam at either of the scheduled times.
- A two-hour period next week at which you could take the exam. This time must be during the regular working day, and must therefore start between 8:30 and 3:00 (so that it ends by 5:00).

In order to schedule an alternate exam, I must receive an email message from you by 5:00 P.M. on Thursday, February 5. Instructions for taking the midterm at an alternate time will be sent to you by email on Friday.

Review session

There will be a general review session for the midterm on Sunday, February 8, from 7:00–9:00 P.M. in Bishop Auditorium. At the review session, Kevin and some of the section leaders will go over problems from the practice exams, but you should also feel free to ask any additional questions that you have.

Coverage

The midterm covers the material presented in class through the lecture on Monday, February 2, which means that you are responsible for the chapters in the text through Chapter 12 ("Dynamic Memory Management").

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 60. 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: Function tracing and/or memory diagramming (10 points)

Assume that the functions **`mystery`** has been defined as follows:

```
int mystery(int n) {
   int v1 = 2;
   int v2 = 1;
   while (n > 0) {
      if (n % 2 == 1) v2 = v1 * v2;
      v1 = v1 * v1;
      n = n / 2;
   }
   return v2;
}
```

(a) What is the value of **`mystery(10)`**?

(b) What is the computational complexity of the **`mystery`** function expressed in terms of big-O notation, where *N* is the value of the integer argument **`n`**? You may assume that **`n`** is nonnegative and that all arithmetic operations run in constant time.

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

I produce the figures in my books 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

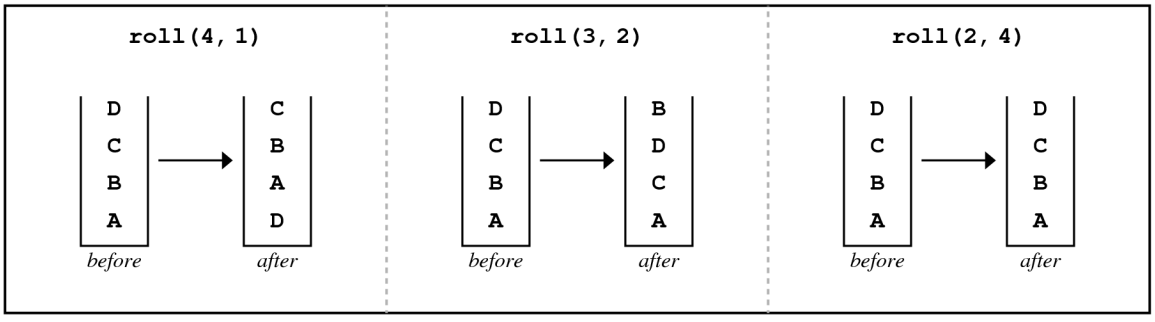
```
void roll(Stack<char> & s, int n, int k)
```

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 iterators (15 points)

For the cell-phone mind-reading problem on Assignment #3, you wrote a procedure

```
void listCompletions(string digits, Lexicon & lex);
```

that printed all words from the lexicon that could be formed by extending the given digit sequence. Suppose that your boss has come back to you with a last-minute change in the product design—a situation that happens all the time in the industry. The new plan is to embed this facility in a web service where many users may ask for such a list. The new concept has the following characteristics:

- Instead of printing the list of words to `cout`, the new version of the method must return it as a `Vector<string>`.
- Given that many customers in a particular telephone exchange will ask for the same prefix string, your boss now wants the implementation to store previously computed results in a map. The first time someone asks for the words generated by a given prefix, your implementation should compute the answer and store it under that prefix in the map. Thereafter, your implementation should simply return the stored value whenever asked about a prefix it has already seen. This technique is called ***caching***.
- Your boss has also become somewhat wiser on the issue of efficiency. Having learned that asking for a recursive solution generated too much panic among the programmers, your manager is now willing to let you use the easy solution previously rejected as too inefficient. That solution, as it was expressed on the assignment handout, was "to iterate through the words in the lexicon and print out every one that matches the specified digit string." In the redesign, the implementation has to add the word to a vector instead of printing it out, but the process is still far easier to implement than the recursive approach.

Write an **iterative** function

```
Vector<string> generateCompletions(string digits, Lexicon & lex,
                                   Map< string, Vector<string> > & cache);
```

that implements this new design. The first two arguments are the same as before; the third is the map that holds the cache of previously computed results. This map will initially be empty, but will grow over time as new prefixes are added to the cache of previously computed results.

Problem 4: Recursive functions (10 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;
}
```

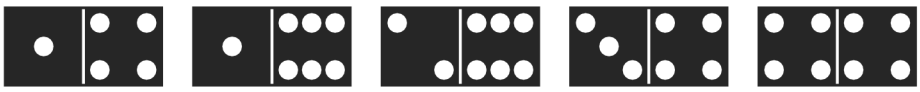
Rewrite this function so that it operates recursively, taking advantage of the following insight:

- 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 (15 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.