Eric Roberts
CS106B

Handout #25
January 30, 2013

Practice Midterm Exam #2

Review session: Sunday, February 3, 7:00–9:00 P.M., Hewlett 201 (next door)
Midterm #1: Tuesday, February 5, 3:15–5:15 P.M., Braun Auditorium (Chemistry)
Midterm #2: Tuesday, February 5, 7:00–9:00 P.M., CEMEX Auditorium (GSB)

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)

As most of you surely already know, the Japanese puzzle game ***Sudoku*** requires you to fill in the entries in a 9×9 grid of integers so that each of the digits between 1 and 9 appears exactly once in each row, each column, and each of the smaller 3×3 squares. A legal Sudoko grid (taken from Figure 5-13 of the reader on page 252) therefore looks something like this:

3	9	2	4	6	5	8	1	7
7	4	1	8	9	3	6	2	5
6	8	5	2	7	1	4	3	9
2	5	4	1	3	8	7	9	6
8	3	9	6	2	7	1	5	4
1	7	6	9	5	4	2	8	3
9	6	7	5	8	2	3	4	1
4	2	3	7	1	9	5	6	8
5	1	8	3	4	6	9	7	2

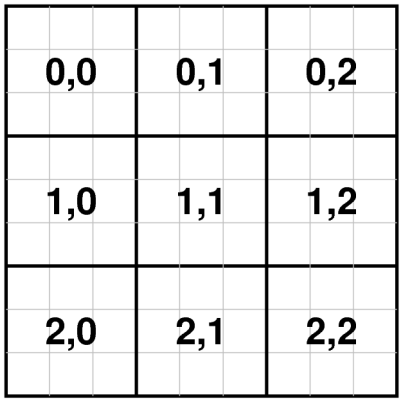
When you write a program to check whether a Sudoku grid is legal, it simplifies the code to decompose the problem into two phases. In the first phase, your program calls a

function that extracts a row, column, or 3×3 subsquare from the entire grid and returns it as a vector of nine integers. You can then use a single function to check the integrity of any row, column, or subsquare.

In this problem, your task is to write the function

```
Vector<int> extract3x3Subsquare(Grid<int> & grid, int bigRow,
                                int bigCol);
```

that takes a complete Sudoku grid and returns a vector containing the nine integers in the 3×3 subsquare indicated by the indices **bigRow** and **bigCol**, which correspond to the row and column of the desired subsquare as shown in the following diagram:



For example, if **grid** contains the complete Sudoku grid from the previous page, calling

```
extract3x3Subsquare(grid, 2, 0)
```

should extract the nine values in the square at the lower left corner, which means that the result should be a **Vector<int>** containing the following values:

```
[9, 6, 7, 4, 2, 3, 5, 1, 8]
```

In writing your solution, you may assume that the **grid** parameter is a 9×9 grid and that the **bigRow** and **bigCol** parameters are both legal values between 0 and 2, inclusive. The elements in the vector should be returned in ***row-major order*** (defined in the reader on page 239), in which the elements from the first row appear in left-to-right order, followed by the elements from the second row, and then the elements from the third row.

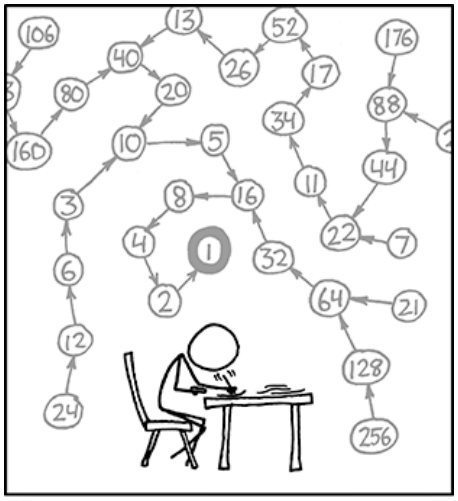
Problem 3: Lexicons, maps, and iterators (15 points)

Exercise 10 in Chapter 1 (which appears on page 54 of the reader) introduces the ***hailstone sequence,*** which is the basis for one of the most fascinating unsolved problems in mathematics. If you starting with a positive integer *n,* you can compute the terms in the hailstone sequence by repeatedly executing the following steps:

- If *n* is equal to 1, you've reached the end of the sequence and can stop.
- If *n* is even, divide it by two.
- If *n* is odd, multiply it by three and add one.

No matter what number you start with, this sequence always seems to get back to 1 eventually. So far, however, no one has yet been able to prove that claim, which is called the ***Collatz conjecture.***

Three years ago, the Collatz conjecture gained additional visibility when it appeared in Randall Monroe's **xkcd**. As the cartoon at the right makes clear, many of the sequences include common patterns. For example, all hailstone sequences (except for 1, 2, 4, and 8, of course) go through the value 16 and therefore share the last four elements of the path. If you've followed through this path once, you don't need to follow it again.



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

Keeping track of these common patterns can make calculating the length of hailstone sequences vastly more efficient. Suppose, for example, that you are calculating the number of steps in all the hailstone sequences between 1 and 100. What happens when you get to 24, which appears at the lower left corner of the cartoon? By the time you get to this point, you've already figured out that it takes nine steps to get from 12 down to 1, so 24 must take ten steps, given that 12 is just one step away from 24. Note that the count is the number of steps; if you start with 1, the number of steps is 0.

It's even more interesting to follow what happens with 7, which is in the middle of the right hand side. Here, figuring out the number of steps for 7 entails running through the number 22, 11, 34, 17, 52, 26, 13, 40, and 20 before you come to 10, which you've already encountered in counting the number of steps from 3 and 6. At this point, you have learned an enormous amount. Given that it takes six steps to reach 1 starting at 10, you know that 20 takes seven steps, 40 takes eight, 13 takes nine, and so forth, backwards along the list of numbers in the chain, until you determine that there are sixteen steps in the hailstone chain between 7 and 1. If you keep track of these values in, for example, a **Map<int,int>**, you'd already know the answers for the other values in the chain. Keeping track of previously computed values so that you can use them again without having to recompute them is called ***caching.***

Write a function

```
int countHailstoneSteps(int n, Map<int,int> & cache);
```

that determines the number of steps in the hailstone sequence starting at **n**. The second parameter is a map containing previously computed values. Your implementation should look up each value it encounters during the process to check whether the answer from that point is already known. If so, it should use that previously computed value not only to compute the current result, but also to add the counts for all the intermediate steps to the map. Thus, if you call **countHailstoneSteps** with 7 as the first parameter, your code should add the counts for 20, 40, 13, 26, 52, 17, 34, 11, 22, and 7 to the map before returning the answer. In this example, the other numbers in the sequence (10, 5, 16, 8, 4, and 2) are already in the map from previous calls and do not need to be added again.

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

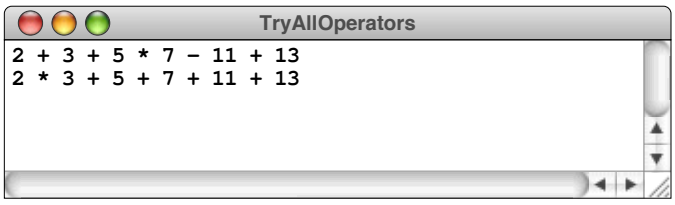
```
tryAllOperators("9 ? 7 ? 5 ? 3 ? 1", 42)
```

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

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:



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.