Homework Assignment #2

Assigned: Feb 19, 2012

Due: Feb 28, 2012, by 10pm

Theme: Mathematical Induction

Problem #1:

Using mathematical induction, prove that $1 + 2 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$.

Problem #2:

Find the flaw with the following inductive "proof" that any amount of three or more chocolate pieces can be formed by some combination of 3-piece chocolate bars and 4-piece chocolate bars.

Basis step: We can obviously have three chocolate pieces as one 3-piece chocolate bar, as well as four chocolate pieces as one 4-piece chocolate bar.

Inductive step: Assume that we can combine 3-piece chocolate bars and 4-piece chocolate bars to achieve a total of *q* chocolate pieces for all non-negative *q* <= *k*. We can then assemble *(k+1)* chocolate pieces by either replacing one 3-piece bar with a 4-piece bar from our existing *k*-piece chocolate collection, or by replacing two 4-piece bars with three 3-piece bars in that set.

By the principle of mathematical induction, this proves that *n* (any number, larger than 3) of chocolate pieces can be assembled successfully from 3-piece and 4-piece bars.

Theme: Algorithm Analysis

Problem #3:

A recursive algorithm that performs a computation on `n` integer numbers requires `T(n)` operations to complete.

(a) Derive a closed formula for `T(n)` as a function of `n`, proving it in the process, if it is known that:

```
T(n) = T(n-1) + n
T(1) = 1
```

(b) Prove, using mathematical induction, that `T(n) = 2n-1`, if it is known that

```
T(n) = T(n/2) + n
T(1) = 1
```

(c) Which of the two algorithms would complete faster on large problems (i.e., a large number of integers)? Why?

Theme: Recursion (revisited)

Problem #4:

Let the recursive function `F(n)` be defined as:

```
        0,          if n=1
F(n) =  F(n/2),     if n is an even (divisible by 2) positive number
        F(3n+1),    if n is an odd (non-divisible by 2) positive number
```

Write a C++ function that computes the number of transformation steps needed for a given positive number n to reach 0 via successive application of the function `F`.

For example, F(5) = F(16) (since 5 is odd, hence 3*5+1=16) = F(8) (since 16 is even, hence 16/2 = 8) = F(4) (similarly) = F(2) = F(1) = 0, which makes it a total of 6 transformation steps: 5 -> 16 -> 8 -> 4 -> 2 -> 1 -> 0.

You may assume (it can be proven mathematically) that 0 will be reached eventually, no matter what number you pick to start from.

Let the number n be supplied on the command-line (so that I can test your function), as a single argument to your code.

In your write-up for this homework, include the source of your function (it should be quite brief), and also submit the CPP file (please call it `num_computation.cpp`).

Theme: Debugging C++ Code

Problem #5:
(a) Find and list the flaws with the following C++ class declaration:

```
class MyClass
{
   private:
      char * pcMyString;

      MyClass( MyOldClass oOld ) { pcMyString = new char[strlen(oOld.pcMyOldString)]; };
      MyClass( MyClass * oObj ) { pcMyString = oObj->pcMyString; poMyOld = oObj->poMyOld; };
      void calcOffset_Helper() const;

   public:
      MyOldClass * poMyOld;

      bool isStringEmpty() const;
      void showString() const;
      void reverseString() const;
      ~MyClass() {};
};
```

(b) After applying appropriate fixes you deem necessary, show the corrected class declaration, without the flaws you reported.

Theme: Programming Mini-Project

Problem #6:
Note: This problem description looks rather long. It is the English description that is long; the actual meaning behind it is relatively simple, as you will realize – but you need to think through it *and* experience it (I strongly recommend doing so by hand, using a paper and pencil on the examples provided below) in order to understand the mechanics deeply, before doing any design or programming. When you're comfortable with your understanding of the problem, all will become much simpler. Be sure to read these instructions in great detail, several times if necessary. Every sentence below matters.

Write a C++ program `StreamProcess`, using classes, that processes two input streams (each in its own text file) and writes out an output result to the console. The file names will be provided on the command line, e.g.:

```
StreamProcess.exe input_data.txt processing_rules.txt
```

The output result (to the console) must have the following format:

```
Starting position: <a given number (see below for description)>
Input sequence:    <a given sequence of 0's and 1's (see below for description)>
Output sequence:   <a processed sequence of 0's and 1's (see below for rules of processing)>
Ending position:   <the position number at which the position pointer ends>
```

The first input stream, `input_data.txt`, contains two lines. The first line holds a single positive number. The second line holds an uninterrupted sequence of 0s and 1s.
Here is one example:

```
7
01110101011101011010111101010110
```

The positive number (7, in the example above) represents the position in the second-line sequence, from which processing will begin. (In essence, this position acts as a pointer in the sequence.)
The sequence on the second line is the input data that will be processed by your program. It represents the initial state of your output sequence.
The output sequence always starts out as a copy of the input sequence, a copy that changes over time as further processing (i.e., executing instructions, as described below) happens to this sequence. In the meantime, the input sequence is kept untouched.
Note: It is important that the first position in those input/output sequences have index 1, not 0 (unlike in C++). For the sample sequence above, this means that the 7th character is 0 (the zero sandwiched between two 1's), not 1.

The second input stream, `processing_rules.txt`, contains a sequence of instructions for how to process the input data from the first stream.
Each line (with one exception, see below) in that file represents exactly two instructions, separated by a comma. Lines of instructions (I refer to them as rows of instructions) are to be numbered starting from 1, i.e., the first row is on line #1, the second row is on line #2, etc. These line numbers are, for simplicity of processing, not included in the `processing_rules.txt` file, but you will need to carefully count (and assign) them as you read that instruction stream.
The first of the two given instructions in each row corresponds to the case when the current character in the output sequence (i.e., the character where the position pointer is) is '0'; the second instruction corresponds to the case when this current character from the output sequence is '1'.

The format of each instruction (notwithstanding the exception, to be described below) is: two characters followed by a positive number (possibly a multi-digit number), with no spacing in between (e.g., `0R16`; here '`0`' is the first character, '`R`' is the second, and `16` is the number).
The meaning of these three elements of an instruction is as follows:

- The first character (either '`0`' or '`1`') is the character that your program needs to write to the output sequence in place of (i.e., at the same position) the character that has just been read from that sequence is.
- The second character (one of '`L`', '`R`', or '`S`') represents how the sequence pointer will move at the end of the current instruction. Namely, '`L`' moves it to the left by one; '`R`' moves it to the right by one; and '`S`' means that the pointer will stay in place (i.e., no movement).
- The positive number at the end of each instruction represents the index of the row of instructions that needs to be looked at next, upon completion of the operations for the present row of instructions. (Once again, remember that row indices start from 1, not 0.)

As a rule, the very first instruction to be executed on the output sequence is always an instruction from row #1.

Exception: There is one instruction that does not conform to the above specification. It is the instruction "`end`" (without quotes). Upon reaching it, your program should complete its processing, i.e., display the output results, and exit. The instruction "`end`" is itself alone on a line. In the sample below, you will see an example of that and how it is interpreted.

Error Handling:
If the processing ever reaches the end of the input sequence (to the left or to the right) and is about to move beyond it, an appropriate error message is to be displayed and the program terminated after showing the calculated output result up to that point.

Example #1 (and walking through it):
Consider the following input data stream:

```
5
00110101
```

and processing instructions stream:

```
0R1, 0R2
1S3, 0S3
end
```

The position of your pointer (#5) is where the * is, below -- pointing at a 0 underneath, in the output sequence):

```
    *
00110101
```

The first instruction is, as mentioned above, always on row #1. Of the two instructions in row #1 (`0R1, 0R2`), we will execute the first one (`0R1`), since it corresponds to the '0' underneath our position pointer in the input stream. (If it was pointing at a '1' in the input stream, we would be executing the second instruction.) So, the instruction says, in English:
"(If you read 0 in the input stream,) write 0 to the output stream, move the pointer to the right, and execute instruction #1 next."
So, as a result, the output stream doesn't change (0 is replaced with 0), but the position pointer has moved and now points at the neighboring 1 to the right:

```
     *
00110101
```

The instruction also hasn't changed: we've been instructed to stay on instruction #1. So our next row of instructions is again (`0R1, 0R2`); but this time, since our position pointer is on a '1' in the input sequence, we'll execute the second of these instructions, `0R2`, which says:
"(If you read 1 in the input stream,) write 0 to the output stream, move the pointer to the right, and execute instruction #2 next."

As a result, the output stream changes (1 is replaced with 0), and the position pointer moves again to the right, now pointing over the adjoining 0:

```
        *
00110001
```

The next instruction is from row #2: (`1S3`, `0S3`). Since the position pointer is over a 0, the instruction to execute will be `1S3`, i.e.:
"(If you read 0 in the input stream,) write 1 to the output stream, keep the pointer in place, and execute instruction #3 next."
Hence, the output stream becomes:

```
        *
00110011
```

and the next instruction is from row #3: `end`. Here is where the processing completes and the program exits.

Example #2:
On the same input stream:

```
5
00110101
```

and with the following processing instruction stream:

```
0L1, 1L2
0L2, 0L3
1S4, 1S4
end
```

The output stream will be:

```
01010101
```

Implementation Requirements:
You must define and use a class for representing an individual instruction, as well as a class for representing a row of (two) instructions.
You must also use a doubly linked list for storing and manipulating the input and output sequence(s). (While it is possible to code this mini-project effectively using dynamic arrays instead, for the sake of your learning experience and further practice with pointers it would help to use doubly linked lists -- hence my insistence on that requirement.)

Tip:
Feel free to borrow (reuse) useful, working code you may have written for Lab #2. There are several important similarities with this mini-project, including parsing input streams, so it may save you effort to not have to rewrite code. On the other hand, if you wish to improve what you've written for Lab #2, you are very welcome to do so, too. To save you time, please reuse the same text reader wrapper class that I supplied for Lab #2 -- it will help you to easily read words (i.e., those between whitespaces) from the two text files. By now, you know how to use that class well and with confidence.

Pair Programming:
If you wish, you may pair up with another student from the class specifically for this programming mini-project. If you do so, be sure to let the instructor know by email by the end of Wednesday, Feb 22, who you are pairing with. Forming pairs after Wednesday will not be allowed.

Submission of Code:
This program must be submitted as code (since I will be running and testing it) -- with its corresponding .CPP and .H files -- unlike the other homework problems (which are to be included all in one write-up document).

Extra credit:
Extra credit is for this problem for students who you are finished with correctly implementing the main requirements. Below are examples of related problems for extra credit. Contact the instructor if you are interested in seeing other (additional) options for extra credit work.
(a) Write the sequence of instructions (in the same format of the `processing_rules.txt` file) that would result in "erasing" (i.e., replacing with '0') the first two '1's (not necessarily lying next to each other) that appear to the right of the position pointer, and changing the first '0' immediately after the second '1' to a '1', before stopping.
(b)* [harder] Do a variation of the above task, but now replace only the first two '1's to the right which are immediately adjoining each other, skipping over other (disconnected) 1's before them, if any.