Dalhousie University
CSCI 2132 — Software Development
Winter 2017
Assignment 4

Distributed Wednesday, February 15 2017.
Due 3:00PM, Wednesday, March 1 2017.

Instructions:

1. The difficulty rating of this assignment is *silver*. Please read the course web page for more information about assignment difficulty rating, late policy (no late assignments are accepted) and grace periods before you start.

2. Each question in this assignment requires you to create one or more regular files on bluenose. Use the exact names (case-sensitive) as specified by each question.

3. C programs will be marked for correctness, design/efficiency, and style/documentation. Please refer to the following web page for guidelines on style and documentation for short C programs (which applies to this assignment):

   `http://web.cs.dal.ca/~mhe/csci2132/assignments.htm`

   You will lose a lot of marks if you do not follow the guidelines on style and documentation.

4. Create a directory named `a4` that contains the following files (these are the files that assignment questions ask you to create): `a4q1.c` and `a4q2.c`. Submit this directory electronically using the command `submit`. The instructions of using `submit` can be found at:

   `http://web.cs.dal.ca/~mhe/csci2132/assignments.htm`

5. Do NOT submit hard copies of your work.

Questions:

1. [15 marks] What kind of solutions a programmer might come up with for the problem of making change in Assignment 3, if he/she does not know greedy algorithms at all? One possible way of solving this problem is exhaustive search, or the brute-force approach. In this approach, we test all possible answers, and compare those that are correct solutions to the given problem, in order to find the optimal answer.

   In particular, for the problem of marking change, given a cash amount in cents, we can first find out for any way of making change correctly (NOT necessarily using as

few coins as possible), the maximum numbers of toonies, loonies, etc. that we can use. We can then perform a six-level loop to iterate through all possibles combinations of coins, using the maximum numbers computed before as the upper bound for the corresponding coins in any solution. More precisely, in the outermost loop, we iterator the solutions that use 1 toonie, 2 toonies, ..., the maximum number of toonies computed. In the second outermost loop, we iterate the solutions that use 1 loonie, 2 loonies, ..., the maximum number of loonies computed (the number of toonies in these solutions is determined in the outermost loop), and so on. In the innermost loop, for each solution, we check whether it is a correct solution for the given cash amount. If it is, then we check whether it uses fewer coins than all the correct solutions that we have seen so far.

For example, if the cash amount is 200, then we can use at most 1 toonie, 2 loonies, 8 quarters, 20 dimes, 40 nickels and 200 pennies in any solution. We then check the following solutions (each solution contains 6 numbers that are the numbers of coins of each type used, starting from toonies): 0, 0, 0, 0, 0, 0; 0, 0, 0, 0, 0, 1; 0, 0, 0, 0, 0, 2; ..., 0, 0, 0, 0, 0, 200; 0, 0, 0, 0, 1, 0; 0, 0, 0, 0, 1, 1; 0, 0, 0, 0, 0, 1, 2, ..., 0, 0, 0, 0, 1, 200; 0, 0, 0, 0, 2, 0; 0, 0, 0, 0, 2, 1; .... In this particular example, any solution that we test should use no more than 1 toonie, 2 loonies, 8 quarters, 20 dimes, 40 nickels and 200 pennies.

This questions asks you to implement the above approach to solve the problem of making change. **No marks will be given if you implement a different algorithm**, such as the greedy algorithm described in Assignment 3.

General Requirements: Use `a4q1.c` as the name of your C source code file.

We will use the following command on bluenose to compile your program:

```
gcc -o coins a4q1.c
```

Your program should NOT print anything to prompt for user input. It reads a single integer from stdin.

Your program then prints the result as six integers, which, from the first to the last, are the numbers of toonies, loonies, quarters, dimes, nickels and pennies required. Separate any two numbers next to each other using one single space character. Do not output any space characters after the last integer. Output a newline symbol at the end so that the result will be printed on a separate line.

Therefore, when you run your program by entering `./coins`, the program will wait for your input. If you enter `372`, the program will output `1 1 2 2 0 2`

That is, the input and output format for a valid cash amount is the same as that specified in Assignment 3.

Error Handling: Your program should print an appropriate error message and terminate if the input amount is negative or greater than or equal to 500.

This is something required in this assignment, but not in Assignment 3.

Testing: To test your program automatically, we will use it as a UNIX filter.

For the example above, we will test it using the following command in the directory containing the executable program:

```
echo 372 | ./coins
```

The output should be:

```
1 1 2 2 0 2
```

Make use of the testing files provided for Assignment 3 to make sure that the input / output format of your program is correct (make sure to use `diff` as we even deduct marks for missing newlines):

```
/users/faculty/prof2132/public/a3test/
```

See Assignment 3 specifications to find out how to make use of the files in the above folder.

We will use different test cases to test your program thoroughly. Therefore, construct a number of test cases to test your program thoroughly before submitting it.

Examples: The following are some examples on running the executable file:

```
mhe@bluenose:~/csci2132/a4$ echo 178 | ./coins
0 1 3 0 0 3
mhe@bluenose:~/csci2132/a4$ echo -5 | ./coins
The amount must be at least 0 and at most 499
```

Additional notes: As you might have noticed after you solved this problem, the program using exhaustive search may take a few seconds to run on bluenose when the input value is big.

To find out the exact amount of CPU time to run your program, you can make use of the `time` command in UNIX (see pp. 130 of the UNIX textbook).

For the problem of making change, what should we do if we are using a coinage system for which the greedy algorithm does not work? The exhaustive search will still work, but it may take long. Say if there are 10 different types of coins and one coin may have a value of 1000 cents. Then it may take hours to find out the solution. To make it faster, we can use the branch-and-bound approach to greatly reduce the solution space. Even better, we can use dynamic programming, which is far more efficient than exhaustive search. You will learn all these in Algorithm I.

2. [15 marks] This question asks you to write a calculator that can perform *a single* arithmetic operation on two fractions, including addition, subtraction, multiplication and division. To simplify your work, both operands (fractions) must be non-negative, though the result of a subtraction could still be negative. The result to be printed must be reduced to the lowest terms. If the result happens to be an integer, simply print this integer.

   Thus, this questions essentially asks you to extend the example on the addition of two fractions shown in class (there is a similar example in the textbook, though the one shown in class is different to show more about using `scanf`). This assignment question is different not only because it asks you to handle more operators, but also because the result must be reduced to the lowest terms. It also requires you to perform error handling.

   The syntax of the language of the input for this calculator is:

   ```
   option numerator1/denominator1 numerator2/denominator2
   ```

   There are two and only two space characters in the syntax shown above. Here `option`, `numerator1`, `denominator1`, `numerator2` and `denominator2` are all non-negative integers.

   The value of `option` can only be one of the following four integers: 1, 2, 3 and 4, which correspond to addition, subtraction, multiplication and division, respectively. The left and right operands are `numerator1/denominator1` and `numerator2/denominator2`, respectively.

   For example, the input `2 4/7 5/15` means $(4/7) - (5/15)$.

   To reduce the result to lowest terms, it is necessary to compute the greatest common divisor of two integers. You are not required to use any advanced approach for this task; any reasonable solution (even the one that requires a simple loop) will do.

   General Requirements: Use `a4q2.c` as the name of your C source code file.

   We will use the following command on bluenose to compile your program:

   ```
   gcc -o fraction a4q2.c
   ```

   Your program should NOT print anything to prompt for user input. It accepts user input that meets the syntax specified above. If the result of the calculation is not an integer, your program then prints a fraction of the form `numerator/denominator`, without any spaces, but there is a trailing newline so that the result will be printed on a separate line. This fraction should be reduced to lowest terms. If the result is an integer, simply print this integer followed by a newline symbol.

Therefore, when you run your program by entering `./fraction`, the program will wait for your input. If you enter `2 4/7 5/15`, the program will output `5/21`

Error Handling: Your program should print an appropriate error message and terminate in each of the following cases:

(a) The user input a value for `option` that is not 1, 2, 3 or 4.

(b) In the input, at least one denominator is 0.

(c) The user supplies a negative value for `numerator1`, `denominator1`, `numerator2` or `denominator2`.

(d) In the case of division, the numerator of the divisor is 0.

If there is more than one problem with user input, your program just has to detect one of them.

Testing: To test your program automatically, we will use it as a UNIX filter.

For the example above, we will test it using the following command in the directory containing the executable program:

```
echo "2 4/7 5/15" | ./fraction
```

The output should be:

```
5/21
```

To help you make sure that the output format of your program is exactly what this questions asks for, several files are provided in the following folder on bluenose to show how exactly your program will be automatically tested:

```
/users/faculty/prof2132/public/a4test/
```

In this folder, open the file `a4q2test` to see the commands used to test the program with two test cases. The output files, generated using output redirection, are also given.

To check whether the output of your program on any test case is correct, redirect your output into a file, and use the UNIX utility `diff` (learned in Lab 3) to compare your output file with the output files in the above folder, to see whether they are identical.

Since these output files are given, we will **apply a penalty to any program that does not strictly meet the requirement on output format**. This includes the case in which your output has extra spaces, or has a missing newline at the end.

We will use different test cases to test your program thoroughly. Therefore, construct a number of test cases to test your program thoroughly before submitting it.

Examples: The following are some examples on running the executable file:

```
mhe@bluenose:~/csci2132/a4$ echo "1 1/3 1/5" | ./fraction
8/15
mhe@bluenose:~/csci2132/a4$ echo "1 1/2 1/2" | ./fraction
1
mhe@bluenose:~/csci2132/a4$ echo "2 1/2 1/3" | ./fraction
1/6
mhe@bluenose:~/csci2132/a4$ echo "2 1/2 1/2" | ./fraction
0
mhe@bluenose:~/csci2132/a4$ echo "2 7/4 23/5" | ./fraction
-57/20
mhe@bluenose:~/csci2132/a4$ echo "3 1/3 7/6" | ./fraction
7/18
mhe@bluenose:~/csci2132/a4$ echo "4 2/5 8/11" | ./fraction
11/20
mhe@bluenose:~/csci2132/a4$ echo "5 1/2 1/2" | ./fraction
The first value must be 1, 2, 3 or 4
mhe@bluenose:~/csci2132/a4$ echo "1 1/0 3/2" | ./fraction
Denominators cannot be zeroes.
```

Hint: As mentioned in the first lecture, for any programming question, the more test cases you pass, the more marks you will get. Thus, if you run out of time, even correctly handling some of the operators without performing reduction to lowest terms will give you some marks, as this will be sufficient for some of the test cases. Test your program for any feature that you implement.