Problems for Op 2016

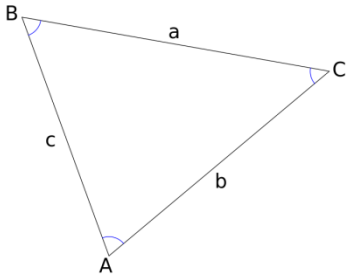
By Eric Durant, PhD, MBA, PE <durant@msoe.edu>
Friday 18 November 2016

- Note: Unless a problem specifies otherwise, you may assume that all user inputs are valid.
- Note: The precise wording of your outputs is not critical as long as they are completely clear.

1. Palindrome (10 points)

Ask the user to enter a lowercase word and then tell the user whether the word was a palindrome. Palindromes read the same forward and backward. For example, “abba” and “deified” are both palindromes.

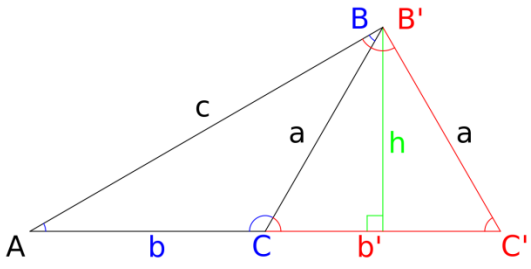
2. Law of sines (10 points)



Ask the user for the length of sides *a* and *c* of the triangle along with the degree measure of ∠A. Calculate ∠C according to the law of sines:

$$\frac{\sin A}{a} = \frac{\sin C}{c}$$

Note that there generally 2 (positive) solutions for ∠C, one of which is found by solving for the angle using the arcsine function. The other, larger value is given by 180°-∠C. You should report this 2nd value of the angle only if it is physically possible (∠A+∠C < 180°). (When sin C = 1 and ∠C = 90°, there is only one solution, but you will not be penalized if you report it twice.) Report all angles in degrees. The second figure illustrates how there might be 2 valid solutions for ∠C.



https://commons.wikimedia.org/wiki/File:{Acute_Triangle, PictureAmbitext}.svg

3. Lensmaker's equation (10 points)

Calculate f, the focal length of a lens, after inputting all the needed quantities from the user. The lens will be biconvex (both surfaces are spherical and protrude from the lens).

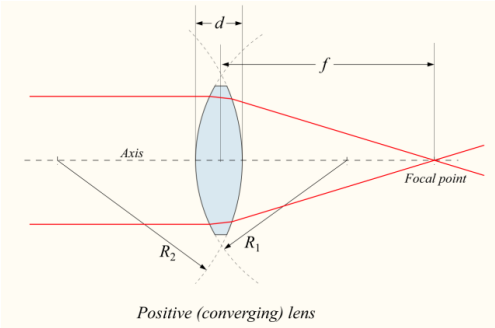
The quantities are related by the lensmaker's equation, where all variables are real numbers:

$$\frac{1}{f} = (n-1)\left[\frac{1}{R_1} - \frac{1}{R_2} + \frac{(n-1)d}{nR_1R_2}\right]$$

As shown in the diagram, the *R*s are the radii of the spherical lens surfaces and *d* is the thickness of the lens. Note that, for a biconvex lens (which is illustrated), $R_1>0$ and $R_2<0$; you may assume the user obeys these constraints.

n is the refractive index of the lens material and $n \geq 1$ for normal materials. You may assume that the user enters an *n* of at least 1.

Assume that all length values are in the same units (centimeters, feet, etc.). It is not important what the specific units are.



https://upload.wikimedia.org/wikipedia/commons/e/ef/Lens1.svg

4. Euler's totient function (20 points)

Write a function that, given a positive integer between 1 and 10,000, calculates ϕ(*n*), which is called Euler's totient function. Euler's totient function has many applications in mathematics including cryptography, which allows computers to privately exchange messages even when using a public network where everyone can see the data being transmitted.

Euler's totient function indicates how many integers between 1 and *n*, inclusive, are relatively prime to *n*. Two numbers are relatively prime if they share no prime factors. So, the following are all pairs of relatively prime numbers: (3,7), (4,5), (12,35).

Consider ϕ(10). The following are relatively prime to 10: 1, 3, 7, 9. All the others in range (2, 4, 5, 6, 8, 10) are not relatively prime since they have a factor of 2 or 5, which are the factors of 10. So, ϕ(10) = 4.

The formula for calculating the totient is based on the prime factorization of *n*. For each prime factor, *p*, it is important how many times, *k*, that factor is repeated. Each prime factor contributes to a product that forms the totient. For p^k, the contribution is $p^{k-1}(p-1)$. For example, if the prime factor 3 is repeated 4 times (thus $3^4 = 81$ is a factor of *n*, but $3^5 = 243$ is not), the contribution is $3^3(2) = 54$.

The contributions of each prime factor are multiplied together to arrive at the totient.

For example, $\phi(1500) = \phi(2 \times 2 \times 3 \times 5 \times 5 \times 5) = \phi(2^2 \times 3 \times 5^3) = \phi(2^2)\,\phi(3)\,\phi(5^3) = (2^1 1)(3^0 2)(5^2 4) = 2{\times}2{\times}100 = 400$.

5. Hangman (20 points)

Using the given "dictionary" file, generate a list of the letters 'a' through 'z'. Along with each letter, indicate its probability of appearing (e.g., 27.3% or 0.273) in a word. This list is helpful in implementing a reasonable good "0th order" (not dependent on context) strategy for guessing in the game of Hangman. (If you are not familiar with the game, please see https://en.wikipedia.org/wiki/Hangman_(game).) Note that letters that appear multiple times in a word do not affect the probability of that letter appearing differently than letters that appear once; it is only relevant whether (not how many times) a letter appears in a word.

The judges will test your program with different dictionary files, but they will be formatted the same as the given file (all lowercase, unique words in alphabetical order on separate lines).

6. Recursive difference equation (20 points)

Given an input sequence of numbers, $x(n)$ (e.g., 0, 1, 2, 3, 4, 5, …) a 2nd order recursive difference equation defines an output sequence of numbers, $y(n)$: $y(n) = b(0)x(n) + b(1)x(n\text{-}1) + b(2)x(\text{n-}2) - a(1)y(n\text{-}1) - a(2)\text{y}(n\text{-}2)$. It is helpful to think of n as a time index, such as how many milliseconds have elapsed. n is an integer and starts at 0. When the equation calls for values of x or y for an argument less than 0, use the value 0 (that is, the sequences x and y are taken to be 0 before $n = 0$).

You will prompt the user to enter the 3 b coefficients (be clear whether you are asking for $b(0)$ or $b(2)$ first) and the 2 a coefficients. You will then display in 2 columns (separate with tabs) the sequences y that result for 2 different functions x. Show your outputs for $0 \le n \le 50$. In the first column, you will show the value of y when x has the value 0 everywhere except at $n = 0$, where it has the value 1. This first column y is called the "unit impulse response" of the system defined by the equation. In the second column, you will show the value of y when x has the value of 1 everywhere beginning at $n = 0$ (but remember that x is always taken to be 0 at $n = -1$ and $n = -2$). This second column is called the "unit step response" of the system.

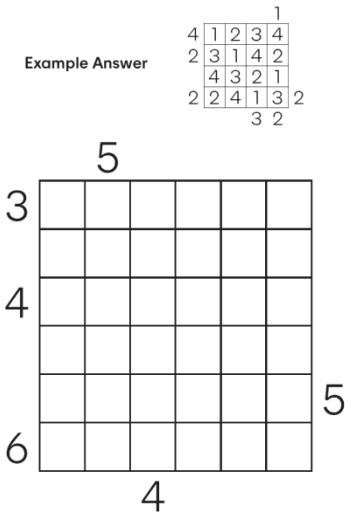
Sample output for a(1) = -0.8, a(2) = 0.6, b(0) = 0.7, b(1) = 0.3, and b(2) = -0.6:

```
 0.7000    0.7000
 0.8600    1.5600
-0.3320    1.2280
-0.7816    0.4464
-0.4261    0.0203
 0.1281    0.1484
 0.3581    0.5065
 0.2096    0.7162
-0.0472    0.6690
-0.1635    0.5055
```

…

7. Skyscrapers (40 points)

In this problem, you will write both a main() program and a helper method (or function). The helper method will return a positive integer indicating which "obvious error" exists in a player's attempt to complete a "Skyscrapers" puzzle or 0 if there is no obvious error. These puzzles come in various sizes, typically between 3 and 6, were designed by Wei-Hwa Huang, and ran in the *New York Times* in summer, 2016. Here is the description of a size 6 puzzle: "Fill the grid with numbers from 1 to 6. The numbers represent the respective heights of skyscrapers — 1 being the shortest, 6 being the tallest. The digits at the side of the grid indicate the number of skyscrapers that can be seen from that vantage point, with shorter buildings sometimes being hidden by taller ones. As in sudoku, no number is repeated in any row or column." To clarify, a shorter building is blocked whenever a taller building is in front of it.



First, your main program will ask the user for the name of a text file describing a puzzle. This file will contain

- First line: number indicating N, the size of the puzzle, which will be between 3 and 6
- Second line: number indicating C, the number of clues given on the outside of the puzzle
- Each subsequent line contains 3 numbers separated by a space and describing how many towers are visible from a particular vantage point
    - The first number is 0, 1, 2, or 3, representing the view from the north, west, south, or east, respectively.
    - The second number is between 0 and N-1, inclusive, and indicates the 0-based row or column, starting from the left or top, of the vantage point.
    - The third number is between 1 and N, inclusive, and indicates the ***exact*** number of buildings that will be visible in a correct solution.

The contest website contains sample files defining each of the illustrated puzzles.

Second, your main program will ask the user for the name of a text file describing the user's attempted partial (or full) solution to a puzzle. Each of the N lines of the file will contain N numbers between 0 and N, inclusive, separated by a space. 0 represents a blank cell, so there will be no 0s when the user has specified their complete solution. The file does not explicitly specify what N is, but you may assume the file is correctly formatted and matched to the previous file. This means, among other things, you may assume it only contains numbers between 0 and N; you may not assume, however, that numbers between 1 and N do not appear multiple times on a line (or in a column) (i.e., the user may have made an obvious mistake).

The contest website contains the complete solution to the size 4 puzzle shown. For testing, you will want to replace some of the numbers with 0s and also enter some invalid solutions (e.g., two buildings of height 3 in the same row).

Third, your main program will call your helper method (you can choose which arguments to pass), which will check if there are any “obvious errors” in the solution and return an integer indicating the result. Below is the complete list of “obvious errors.” If multiple obvious errors occur, you should just detect and return the one with the lowest ID number. If there are no obvious errors, your method will return 0.

- Error 1: The same number between 1 and N appears more than once in any row or column.
- Error 2: There is an empty cell that cannot be filled with a number since all the numbers 1 through N already appear in the 2N-2 cells in the same row and column.
- Error 3: There is a row or column that is completely filled that does not present the number of specified faces.

Fourth and finally, your main program will output a descriptive message based on the integer returned describing the error (you may use the text above) or lack of any obvious error.

8. Text expander (40 points)

Write a "text expander" that will prompt the user for the characters at the beginning of a word until enough information is given to determine which word the user is attempting to enter. The words that the user wants are in the given "dictionary" file containing an alphabetized list of lowercase words. After the user enters a character, the program will decide that one of the following is true and take the indicated action:

- 0 words are possible, so report this to the user and exit.
- Only 1 dictionary word is possible, so output that word and exit. (This is not the same as saying the user has entered a valid word. For example, a user who has entered "acid" might be aiming for "acidify.")
- Two or more dictionary words are possible, so show all options up to a maximum of 10 to the user indicating a single-digit number for each, beginning with 0. Then, wait for the user to enter a response.

When a response is needed, the user may enter a letter or a single-digit number. The program should ignore numbers when they're not expected (this includes entering a number that is too large).

9. Rotation code (40 points)

Write a program that generates and displays a "rotation code" based on an initial value given by the user. Each item in the code has the same number of characters, and each character is either a 0 or a 1. The program must handle items up to 9 characters long. A new item is generated from the previous item by moving each character one position to the left. The original leftmost item "falls off the left edge", changes its value, and appears as the new item on the right. New items are generated until (but not including) the item the user originally entered is reached. Be careful that you don't ignore any leading 0s in the user's input.

For example, if the user enters "10", the "0" moves to the left place while the leftmost digit (here a "1") falls off the left, changes into a "0" and becomes the rightmost digit, yielding "00". Following these rules the next item is "01", followed by "11". This is the final item since the next item would be "10," but that would repeat the initial value given by the user.

Generating Random Integers

Java

You will need a pseudorandom number generator to solve some problems. One such generator is provided by the **`java.lang.Math.random()`** static function, but here we discuss the more flexible **`java.util.Random`** class and, in particular, how it can be used to generate unsigned random integers.

First, create a generator object with

```
java.util.Random myGenerator = new java.util.Random();
```

In contrast to the C/C++ version, this is automatically seeded with the current time so that a different pseudorandom sequence is given on each run. If desired, a specific sequence can be requested by passing a **`long`** seed to the constructor.

Random integers are then generated as follows.

```
int myInt = myGenerator.nextInt(10);
// generates one of the 10 random integers between 0 and 9

double myDouble = myGenerator.nextDouble();
// generates a random value between 0 and 1
```

C++

You will need a pseudorandom number generator to solve some problems. One such generator is part of the C standard library and is included in standard C++. This generator consists of two functions. To use these two functions you must **`#include <cstdlib>`**.

The first of these functions, **`void srand(int)`**, "seeds" the random number generator, starting it at a specified point. Note that **`srand`** should normally only be called once in a program. So that the random number generator will usually start at a different point each time the program is run, you must seed it with a different value on each run. This can be accomplished by using the time, which changes each second. You can gain access to the standard C functions for working with time via **`#include <ctime>`**. The following will use the current time to seed the random number generator…

```
srand(static_cast<unsigned int>(time(NULL)));
```

The other function**`, int rand()`**, returns an effectively random **`int`**. The function **`rand()`** returns integer values between 0 and **`RAND_MAX`** where **`RAND_MAX`** is a constant defined in **`cstdlib`**. So, **`(double)rand() / RAND_MAX`** will return a random **`double`** between 0 and 1.

Note: Problems requiring the use of random numbers must properly seed the random number generator so that they generate different results each time they are run. If you do not seed the random number generator, it gives you the same sequence of "random" numbers every time.