Problems for Op 2011

By Eric Durant, PhD, MBA <durant@msoe.edu>
Friday 18 November 2011

1. Law of cosines (10 Points)

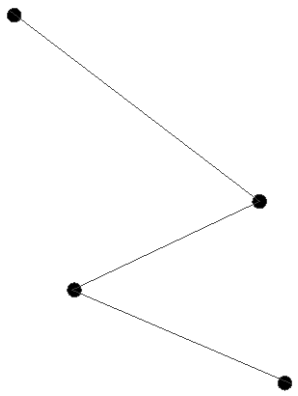
Write a program that asks the user for the length of 2 sides of a triangle and the angle between them in degrees (not radians). Your program will calculate and output the length of the other side using the law of cosines, $c^2 = a^2 + b^2 - 2\,ab \cos \gamma$, where a and b are the 2 input sides, γ is the angle between them, and c is the third side. You may assume that the numbers entered describe a valid triangle.

2. Crossword helper (10 Points)

Write a program that generates simple clues in the form of "*n*-letter word that begins with '*x*'" after the user enters a word. For example, if the user enters "Crossword" your program would output "9-letter word that begins with 'C'". Case is not important, but be sure to handle the error when the user enters a blank word by displaying the message "must have at least 1 letter" or making the user enter another word. That is the only error you need to handle.

3. Length of rope (10 Points)

A thin rope is wrapped around some small pegs. The user will type in the number of pegs followed by their (x,y) coordinates and your program will output the length of the rope. You must prompt the user so they know which value they are to enter (e.g., "Number of pegs?", "Y value for peg 3?"). If they enter 0 or a negative number of pegs, print an error message such as "Invalid number of pegs." You may assume the user enters only real numbers for the positions and only integers for the number of pegs. Note that if there is only 1 peg the length of the rope is 0.



4. Euclidean algorithm (20 Points)

You will prompt the user for 2 positive integers (you may assume they're entered correctly) and then calculate and display their greatest common divisor (GCD), which is the largest integer that divides them both without leaving a remainder. You must implement and use the "Euclidean algorithm" to calculate the GCD; you may want to write a method to do this.

Example 1 of GCD: 3 is the GCD of 21 and 39 since (a) it evenly divides both numbers (21 = 3 × 7 and 39 = 3 × 13) and (b) it is the greatest/largest number to do so.

Example 2 of GCD: 6 is the GCD of 408 and 906 since (a) it evenly divides both numbers (408 = 6 × 68 and 906 = 6 × 151) and (b) it is the greatest/largest number to do so.

The Euclidean algorithm is an efficient way to find the GCD of 2 integers. Here are its steps:

1. Start with the 2 numbers, *a* and *b*.
2. If either *a* or *b* is 0, the other number is the GCD and you're finished.
3. Replace the larger number with its remainder when divided by the smaller number. This can be calculated using the % operator in Java and C++.
4. Go to step 2.

Example of the Euclidean algorithm for *a* = 408 and *b* = 906:

- *b* is larger, so it is replaced with *b*%*a* = 906%408. So, *b* becomes 90 (906 = 2 × 408 + 90).
- *a* is larger, so it is replaced with *a*%*b* = 408%90. So, *a* becomes 48 (408 = 4 × 90 + 48).
- *b* is larger, so it is replaced with *b*%*a* = 90%48. So, *b* becomes 42.
- *a* is larger, so it is replaced with *a*%*b* = 48%42. So, *a* becomes 6.
- *b* is larger, so it is replaced with *b*%*a* = 42%6. So, *b* becomes 0.
- One of the numbers (*b*) is now 0, so the other number (*a*) is the answer (6).

5. Newton's method (20 Points)

Newton's method is an algorithm for finding where a function reaches 0. It works by starting with a guess of an x value and calculating f(x). Since x is only a guess, f(x) ≠ 0. The algorithm calculates y = f(x) and the slope of the function at the point (x,y). In calculus, this slope is called the first derivative and it is also a function that varies with x; it is symbolized f'(x). Newton's method makes use of f(x) (how close we are to the goal of 0) and f'(x) (how quickly and in which direction the function is sloping at x) to make a new guess x for the value where the function reaches 0:

$$x := x - f(x) / f'(x)$$

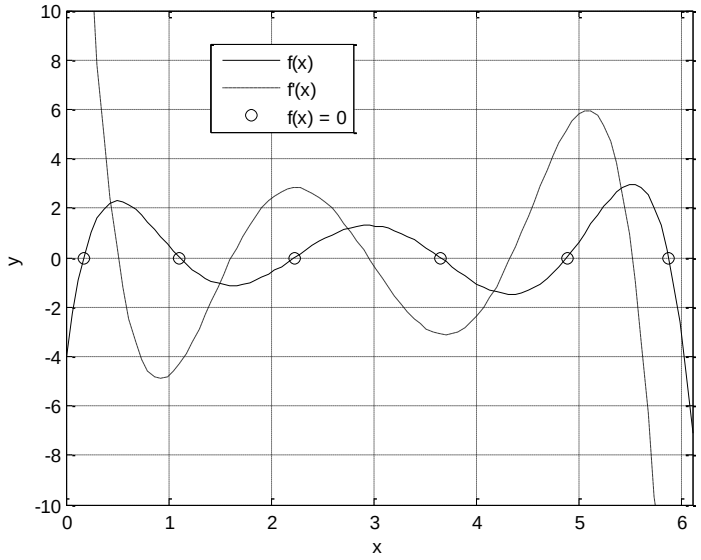
:= means that x is assigned a new value. This is just = (the assignment operator) in Java or C++, but := is used to make clear that this is not an algebraic equation stating 2 quantities are equivalent.

Specifically, you will work with

$$y = f(x) = a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

$$f'(x) = 6a_6x^5 + 5a_5x^4 + 4a_4x^3 + 3a_3x^2 + 2a_2x + a_1$$

where $a_6 = -0.09$, $a_5 = 1.6108$, $a_4 = -10.9167$, $a_3 = 34.7625$, $a_2 = -52.0433$, $a_1 = 31.1767$, and $a_0 = -4$. These values are exact – do not round them or remove any digits. They should be constants in your program that can be easily changed. Here is a plot of the function and its derivative (local slope):



Your program will ask the user for the initial guess of x and then run Newton's algorithm several times to find a more accurate value of x where f(x) reaches 0. You will keep finding a better value until the absolute value of y is less than 0.001. You must display the guess x after each iteration. For example, if the user enters a guess of 4, your program must refine the guess since f(4) = -1.042. It would ultimately find something near x = 3.64125. In more detail, your output in this case should be similar to:

```
Initial position: f(4.0) = -1.042
Refined zero: f(3.56235) =  0.23991
Refined zero: f(3.64288) = -0.00504
Refined zero: f(3.64124) = -0.00000107
```

6. Sudoku predicate (20 Points)

Write a method (and a program to use it) that determines whether a particular number can be placed in a particular square on a Sudoku board. Sudoku is a game played on a 9×9 grid where you start with several cells containing numbers 1 through 9, which are the only valid numbers in the game. The goal is to fill the entire grid so that in any row and any column each number 1 through 9 appears exactly once. Also, the 9×9 board is divided like a tic-tac-toe board into 9 3×3 subregions, and each subregion must contain each number exactly once. It may be helpful to see this article for an illustration: http://en.wikipedia.org/wiki/Sudoku

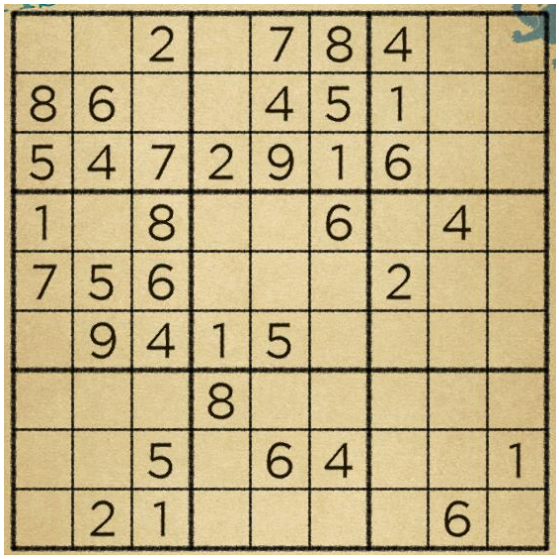
Your method does not implement any Sudoku strategy; it just checks whether the 3 basic rules (row, column, and subregion) are satisfied for a move that the player is trying to make. You can assume that the board does not violate any of the 3 rules before the move is made. Your method takes 4 arguments:

- a 9×9 array of int (one way to created this in Java is with `int[][] A = new int[9][9];`). You will have to decide how to represent empty squares. We recommend using 0 since that is not one of the 9 valid moves.
- A row number from 0 to 8

- A column number from 0 to 8
- A move to try to make in the given row and column, a number from 1 through 9.

Your method returns the `boolean` (`bool` in C++) `true` if the move would not violate any of the 3 rules and `false` otherwise.

You also must write a test program that prompts the user for a row, column, and move to try. If the square at the row and column is already filled, report the error, otherwise call the method and use the result to tell the user whether that move violates the rules. Declare an array representing the following board and use it for testing:



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

Rows and columns are numbered from the upper left starting at (0,0). We can see that the square at (0,0) is empty and it cannot contain 8, 5, 1, or 7 because of the column rule; 2, 7, 8, or 4 because of the row rule; or 2, 8, 6, 5, 4, 7 because of the subregion rule. So, if the user tries to put an 8 there, your method will return `false`. But, since the number 3 does not violate any of these rules, your method would return `true`. (This does not mean that 3 is the correct move in the solved puzzle, but just that it is a move that meets the basic rules; that is all you need to check.)

7. Sudoku potential moves (40 Points)

You should complete problem 6 before working on this problem. You may work on this problem before your problem 6 is graded, but be aware that any errors in your solution to 6 will probably prevent you from solving this problem. But, since you must implement 6 in a method, it should be easy to copy and paste any corrections you make to 6 over to your working area for this problem.

Write a program that uses your method from 6 and a new method that you will write for this problem. The new method takes 3 arguments (the same as the first 3 from the solution to problem 6) and returns an `int[]` of length 0 to 9 containing all the valid moves for the given square (In C++ you may also want to pass an argument by reference to receive the size of the array.). Call your method written in 6 to determine whether each move is valid.

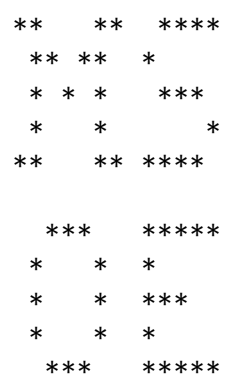
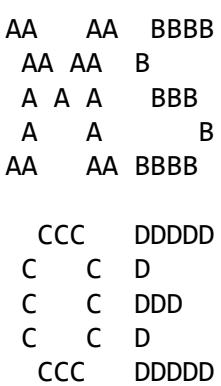
The main program you write for this problem will define the 9×9 board as in problem 6 and then call your new method for ***every empty square***. Depending on the size of the returned array, display one of the following sentences, filling in (x,y) and move values according to the data:

- "There are 0 valid moves for square (x,y), so the board is invalid." [This shouldn't happen because you're promised a valid board, but checking may be helpful in debugging your code.]
- "There is 1 valid move for square (x,y) and it is N. You should make this move."

- "There are multiple valid moves for square (x,y) and they are N O P Q." [Note: example for 4 possibilities; there may be anywhere from 2 to 9.]

8. Image processing (40 Points)

Automated optical character recognition (OCR) software often needs to identify all of the contiguous elements in a monochrome image. This requires that the image be scanned pixel-by-pixel to determine when two "black" pixels are adjacent to each other. Pixels are considered adjacent when they touch horizontally, vertically, or diagonally. Each region is marked with a unique code so that it can be extracted later by the character recognition software. For example, the following image would be grouped as follows (NOTE: Your use of the specific classification symbols, A,B,C,D, may vary as long as each is unique.):

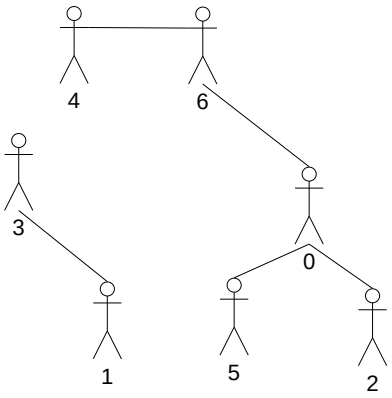
	
Before Processing	After Processing

In this case the file was separated into four separate elements and each pixel of each element was labeled with a letter unique to that element. The actual letter is irrelevant as long as each is unique.

Write a program that ***prompts*** the user for a data file and extracts the number of rows and columns in the image from the first line. Each subsequent line will contain one row of the image with space used for a white pixel and "*" used for a black pixel. Find the separate elements in the image and display the processed image to the console. For your convenience, two sample files have been provided. **`p8-image1.txt`** contains the example shown above and **`p8-image2.txt`** contains a second image.

9. Degrees of separation (40 Points)

Facebook wants to add a feature that lets you enter someone's email address and, if they are in the system, tells you how many "degrees of separation" they have from you. If you are friends with someone, you have 1 degree of separation. You have 0 degrees of separation from yourself. You have 2 degrees of separation from all your friends' friends who are not your direct friends or yourself. In the figure at right, for example, "4" and "0" have 2 degrees of separation, "5" and "5" have 0 degrees of separation, and "2" and "4" have 3 degrees of separation. "6" and "3" have infinity degrees of separation.



Write a program that takes a file from the Facebook server that has a list of all "friend pairs" in the system. To protect user identities, the file assigns each user a unique number from 0 to *N*-1, where *N* is the number of registered users. The file contains *N* on the first line, and a friend pair separated by a space on each additional line. After reading the file and performing the needed calculations, your program will prompt the user to enter 2 user IDs and then report the degrees of separation between them. For example, for the given diagram, the file might contain

```
7
4 6
6 0
0 5
0 2
3 1
```

Hints

- The above data file is provided on the contest website as `p9-sep1.txt`
- Keep track of the distances between people by using a matrix or 2-D array.
- Since the distance from *a* to *b* is the same as from *b* to *a*, you only ***need*** to use half of the matrix (perhaps the half where $a \le b$), but you ***can*** use the entire matrix.
- There are various ways to handle infinite degrees of separation, which is the case for every pair before considering any connections. For this program, use "9999" instead of infinity. When the result is "9999," it will be understood that the users aren't connected.
- Put "1"s in the matrix for each pair in the file. Now you have a matrix with 1s and 9999s but need to find where the 2s, 3s, etc. go.
- Iterate over the matrix as long as you find shorter connections. One correct algorithm is to consider each pair (*a*,*b*) and then consider all third parties *c* (you'll probably need 3 or 4 loops to do this). If the degrees of separation (*a*,*c*) added to (*b*,*c*) are less than what you have for (*a*,*b*), then you've found a smaller number of degrees of separation for (*a*,*b*) and should update your matrix.