Problems for Op 2017

By Eric Durant, PhD, MBA, PE <durant@msoe.edu>
Friday 17 November 2017

- 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. Popular letter (10 points)

Ask the user to enter two unique, individual letters in lowercase followed by a word in lowercase. Then output the letter that occurs more often in the word. If the letters occur equally often (including 0 times), output both of them.

2. Pythagorean triple (10 points)

A Pythagorean triple is a set of positive integers such that $a^2+b^2=c^2$. c represents the length of the hypotenuse of a right triangle, while a and b are the lengths of the other legs. Prompt the user for a, b, and c (you may assume they enter the largest value in c), and then tell them whether those numbers form a Pythagorean triple.

3. Crossword assembly (10 points)

Prompt the user for two words in lowercase and then describe all the ways they can cross in a crossword puzzle. They can cross wherever they have the same letter in a position. Start counting letter positions at 1. For example,

- Input: dog, bot; output: (2,2)
- Input: cat, dog; output: [blank may be used to indicate no crossings]
- Input: crisis, oasis: (3,4), (4,3), (4,5), (5,4), (6,3), (6,5)

4. Finding π (pi) (20 points)

Adding the reciprocals of all the squared positive integers converges to $\pi^2/6$. Prompt the user for a maximum number and then iterate through all the partial sums starting with the reciprocal of the square of 1, then adding the reciprocal of the square of 2, and so on through the reciprocal of the square of the user-specified maximum. At each iteration, estimate π by multiplying the sum by 6 and then taking the square root; output each of these estimates on a separate line. If you do this correctly, the output will converge towards π. For example, $\sqrt{(6\times(1/1 + 1/4 +1/9))} = 2.857\ldots$

5. Pizza schedule (20 points)

This problem is inspired by a problem by MSOE's Dr. Gary Au. A family has children who all love pizza, so they get pizza for a meal once per day. Unfortunately, all the children have a different favorite, but, due to pizza pricing ($20 for the "family pleaser" and $18 for the "tiny"), their parents order one "family pleaser" sized pizza each day. However, the children have been taught to compromise, so each child is content to eat their non-favorite pizza as long as they get their favorite with a certain regularity.

Your program will begin by prompting for each child's "tolerance period," that is how many days without their favorite pizza it takes for them to become discontented. For example, if the number of days is 2, the child must have their favorite pizza at least every other day, but a pattern like Monday, Tuesday, Thursday, Friday, Sunday, … would also work. No two children have the same tolerance period and each tolerance period is at least 2. It is unknown how many children there are, so the user will enter a 0 to indicate that the last child's tolerance period has been entered.

With this information in hand, you will attempt to put together a 365-day (days 0 through 364) pizza schedule for the family using a greedy approach. Specifically, you will first take the least tolerant child and schedule their favorite pizza on as few days as possible and starting as late as possible. For example, if the least tolerant child will become mad on the 3rd day without their pizza, you would schedule days 2, 5, 8, 11, 14, …, 359, 362. Then, you would proceed to schedule the second least tolerant child, again starting on the last possible day and scheduling their pizza as seldom as possible. If, for example, the next child would become mad on the 5th day, you would schedule their pizza on…

- Day 4 (if there were no pizza on days 0 through 4, that would be 5 days without their pizza)
- Day 9 (5 days later)
- Day 13 (you'd like to go to 9+5 = 14, but pizza 3 is already scheduled then, so you need to back off a day)
- …

It is possible that if the family is demanding enough you will determine that no schedule is possible using the greedy approach described above. If so, report that to the user. Otherwise, output the pizza schedule, which is a list of 365 numbers with each number representing a specific child by their "tolerance period". Use a 0 to represent any days on which it is not necessary to buy a pizza to keep any child happy.

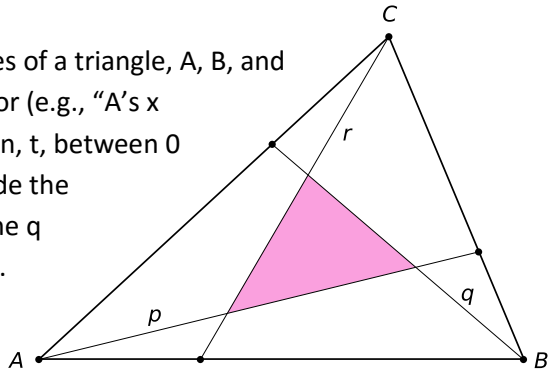
6. Dedekind ψ (psi) function (20 points)

The Dedekind ψ (psi) function is defined for positive integers n to have the value which is the product of n and all terms (1+1/p) where p is a distinct prime factor of n. Distinct means that a prime factor is only used once even if it appears multiple times in the prime factorization of n. Remember that 1 has no prime factors since the first prime is 2. Here are how some values of the function are calculated:

n	1	2	3	4	6	30
ψ(n)	1	2(1+1/2) = 2(3/2) = 3	3(1+1/3) = 3(4/3) = 4	4(1+1/2) = 4(3/2) = 6	6(1+1/2)(1+1/3) = 6(3/2)(4/3) = 12	30(1+1/2)(1+1/3)(1+1/5) = 30(3/2)(4/3)(6/5) = 72

Note that it is not necessary to use decimals to perform these calculations. The result will always be an integer. Your program will prompt the user for n and then calculate and output ψ(n). Do this using only integer data types (using floating point data types can introduce unwanted round off error.) Your program must work correctly for n from 1 to 10,000.

7. Internal triangle area (40 points)



Prompt the user to enter (x,y) coordinates for three vertices of a triangle, A, B, and C. At each step, it should be clear what you're prompting for (e.g., "A's x coordinate" or "Ax"). Then have the user enter a proportion, t, between 0 and 0.5. This proportion is used to construct a triangle inside the original triangle as shown in the figure. In particular, the line q starts at B and ends at a point t of the distance from C to A. Similarly, the line r starts at C and ends at a point t of the distance from A to B. And, the line p starts at A and ends at a point t of the distance from B to C. The internal triangle, shaded in the figure, is formed by these 3 internal lines. Using any method you chose, calculate the following values based on the inputs and constructed triangle. Remember that you are allowed to consult online references about formulas for areas of triangles, etc.

- The area of the original triangle entered by the user
- The area of the internal triangle
- The ratio of the 2nd area to the 1st one. This ratio cannot be greater than 1.

Image source: https://en.wikipedia.org/wiki/One-seventh_area_triangle#/media/File:One-seventh_area_triangle.svg

For this contest, do not worry about handling the case of infinite slope correctly; that is, you may assume that none of the considered lines are vertical. This means you can use point-slope representations if you wish, and do not need to check for division by 0.

8. Life simulation (40 points)

Write a program that executes Conway's Game of Life. Begin by prompting the user for a filename. The filename will then be used to read in a 2-D array of values representing an initial board position. The first line of the file contains two integers separated by whitespace; these integers represent the number of rows and the number of columns, respectively. The subsequent lines use 0s and 1s to indicate which elements of the array contain (1) or do not contain (0) a live cell:

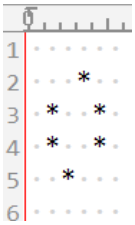
For example (go to contest website, click "Data Files" for this and more):

```
6 6
000000
000000
001110
011100
000000
000000
```

You will then output the pattern to the console, using * to indicate a live cell and a blank to indicate a dead cell. For example, the output corresponding to the above file would be as shown at right, where light dots are added for clarity to indicate spaces. After outputting the pattern, you will output "==========" (10 equals signs) on a line and then calculate what happens to the array of cells in the next generation:

- If a cell is alive but only has 0 or 1 live neighbors, it dies (underpopulation).
- If a cell is alive and has 2 or 3 live neighbors, it survives.
- If a cell is alive but has more than 3 live neighbors, it dies (overpopulation).
- If a cell is dead but has exactly 3 live neighbors, is becomes alive (reproduction).

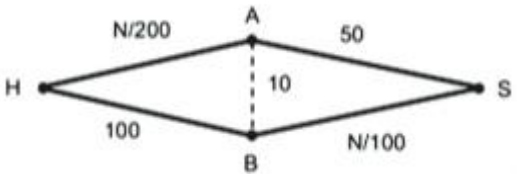
You will then output the new generation. For the above example, it would be as shown at right.



Your program will then calculate and output the next generation. This will continue forever (until the user uses means external to your code to terminate the program). Neighbors include diagonal cells, so there are generally 8 neighbors, but cells at the edges only have 5 neighbors while cells in the corners only have 3 neighbors. Be careful that each cell's status in the new generation is determined by the status of the cells surrounding it in the previous generation (not a mix of the previous and current generation).

9. Commuter dilemma (40 points)

This problem is inspired by a problem published in the Wall Street Journal on October 12, 2017. That problem is also the source of the graphic. (https://blogs.wsj.com/puzzle/2017/10/12/varsity-math-week-109/)



During every morning rush hour, M cars need to get from H to S, where travel times between points are shown on the map at right. Each driver needs to make 2 decisions: first, whether to head to intermediate point A or B, and, second, whether to take the AB bridge before proceeding to S. You will write a simulation tracking which cars take which route over a D-day period. On day 0, each driver selects a route randomly (both alternatives equally probable at 50%); you will need to remember this as the driver's preferred route, which may change during the simulation. The commute times from H to A (HA) and from B to S (BS) vary depending on how many cars are travelling that particular leg. On each day of the simulation you will output a row of a table indicating:

- Day number: 0 through D-1
- HA: number of cars going from H to A
- AB: number of cars going from A to B
- BA: number of cars going from B to A
- BS: number of cars going from B to S
- Average time: the average time it takes a car to get from H to S.
- Minimum time: the shortest time to complete the HS route among the 4 alternatives. Even if 0 cars take a particular path, it is still a valid alternative for this calculation.
- Maximum time: analogous to "minimum time", but by taking the longest among the alternatives

After each day, each driver learns of the minimum time for the day. A proportion, p ($0 < p \leq 1$, input by the user), of the drivers who did not make the commute in the minimum time change their routes. Specifically, a rerouting driver selects the optimal route using the leg commute times of the day just finished (and in doing so might change tomorrow's optimal route). The other drivers stay on their old route for at least another day.

At the beginning of your simulation, you will prompt the user for

- M, the number of cars,
- D, the number of days, and
- p, the probability of rerouting when suboptimal

and then proceed through your simulation, outputting each row as it is calculated. Be sure to test your program with various numbers of cars, including 0, 1, 10, 100, 1000, 10,000, and 100,000.

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.