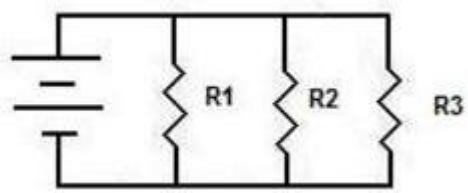
Problems for Op 2013

By Eric Durant, PhD, MBA <durant@msoe.edu>
Friday 22 November 2013

1. Parallel resistance (10 Points)



Calculate the "parallel resistance" of three loads (1st 3 inputs from user) placed across a "voltage source" (4th and final input from user). For example, the voltage source might be a battery and the loads might be an LED (light source), computer processor, and camera. Ohm's law tells us that the electrical current (proportional to electrons per second) flowing through a load is equal to the voltage divided by the resistance, or I = V/R. These quantities are given in units of amperes (A), volts (V), and ohms (Ω), respectively. For example, if we have a 5 V battery and an LED with resistance of 1000 Ω, the resulting current through the LED is 5/1000 V/ Ω = 0.005 A.

In our example, all 3 devices are placed with their ends touching the ends of the voltage source since they each require the voltage it provides. In this case, the voltage source is providing the calculated current to each of the 3 devices, thus we can add the 3 device currents to get the current out of the voltage source. Now that we have the total current, we can solve for the parallel resistance again using Ohm's law: $I_{total} = V/R_{parallel}$.

For example, if the user tells us the resistances are 500, 1000, and 2500 Ω, and the voltage is 5 V, the currents would be 0.01, 0.005, and 0.002 A. Adding the currents together, we get a total of 0.017 A. Then, we can solve for the resistance as seen from the voltage source by solving the equation for R: R = V / I = 5 V / 0.017 A = 294.12 Ω.

Note: There is a shortcut to this problem that allows you to ignore the input voltage. You may use it, but are not required to do so.

2. Scrabble score (10 Points)

Ask the user to input a word and calculate its Scrabble score. The user may use any mix of uppercase and lowercase, but case is not significant in your calculation. You may assume that the user enters only letters. The values of the letters are given below:

- 1 point: A, E, I, L, N, O, R, S, T, U
- 2 points: D, G
- 3 points: B, C, M, P
- 4 points: F, H, V, W, Y
- 5 points: K
- 8 points: J, X
- 10 points: Q, Z

3. Bertrand's ballot theorem (10 Points)

Bertrand's ballot theorem calculates the probability that the ultimate winner of a 2-candidate election is ahead after each individual vote is cast (so, they must receive the first vote; we are ignoring ties). Given that candidate A is the ultimate winner and receives *p* votes and candidate B receives *q* votes, the probability is:

$$\frac{p-q}{p+q}$$

Ask the user to input *p* and *q* and calculate the probability. Assume the user enters positive integers. Your answer should take the form of "Candidate B, who received 13 out of 20 votes, had a 0.3 chance of being ahead throughout the election." This example is for $p = 7$ and $q = 13$. Your sentence should begin with which ever candidate received more votes.

4. Anagrams (20 Points)

Ask the user to input 2 words. Ignore case. Assume the user enters only letters. Report whether the words are anagrams of each other, that is, whether they contain the identical number of each letter. Note that the words must be of exactly the same length in order to possibly be anagrams.

5. Angle between clock hands (20 Points)

Ask the user for the hour and minute and display the resulting angle between the hands of a clock. Assume that the hour is correctly entered as an integer between 1 and 12 and that the minute is correctly entered as a real number between 0 and 60. The angle must be presented as a value between 0 and 180 degrees inclusive.

6. Text histogram (20 Points)

A histogram is a graph that shows the number of values in each of a series of bins. It is a useful tool for showing where values are most popular. A histogram is computed by dividing the number line up into a series of bins of the same size and counting the number of values in each bin's range. For the sequence of data: 3 6 1 1 2 3 4 5 1 3 6 2 9 4 2 3 4 6 6 7 and bins starting at 0 with a width of 2, the following histogram would result:

```
0 - 1    ***
2 - 3    *******
4 - 5    ****
6 - 7    *****
8 - 9    *
```

This result is correct because there are 3 values in the range 0 to 1, 7 in the range 2 to 3, …, and 1 in the range 8-9.

Write a program that inputs from the user:

- a starting bin location (for the example above this is 0),
- the bin width (for the example above this is 2),
- and all the values on a single line, separated by spaces.

Compute and plot the histogram in a format similar to the example above. You may assume that all the data are integers. If the starting location is greater than the minimum of the data, respond as if the user entered the minimum for the starting location.

7. Bertrand's ballot theorem simulation (40 Points)

Write a program to simulate a large number of elections to confirm whether Bertrand's ballot theorem (see problem 3) is approximately correct. You will ask the user for 3 inputs, and you may assume they are all valid:

- The number of voters (an integer from 1 to 1000)
- The probability that each voter will vote for candidate A (between 0 and 1)
- The number of elections to simulate (an integer from 1 to 1000)

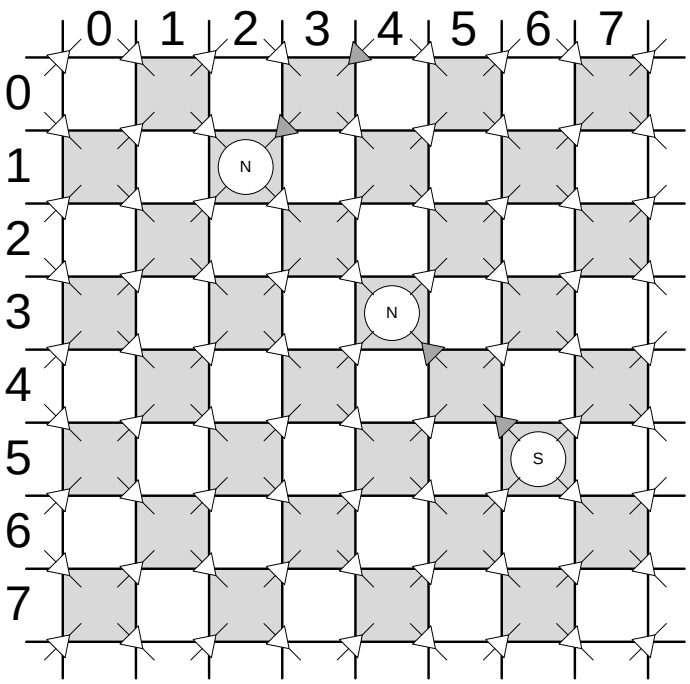
For each election, you must determine whether the ultimate winner was always ahead, starting with the first vote. Thus, if A receives the 1st vote, but falls behind or reaches parity at any point as votes are tallied (regardless of whether A wins), the election counts in the "not always ahead" category. Note that it is possible and okay that A and B each win some elections where they are always ahead.

After running all the elections (at least far enough to categorize them), you will divide the "always ahead" election count by the total election count and summarize the relevant facts for the user similarly to the following: "In 1000 simulated elections each with 100 voters, where candidate A received each vote with probability 0.65, the winner was always ahead 0.287 of the time."

Note: This is solving a similar, but different problem, than Bertrand's theorem. Results may differ greatly for small electorates. While Bertrand's theorem applies to exact numbers of votes, this problem concerns the probability of receiving each vote.

Note: Your program will need random numbers to solve this. See the end of this document for information on how to generate these.

8. Spin ice (40 Points)



A "spin ice" is a substance similar to a crystalline solid in that its atoms are locked in fixed, regular positions. Each vertex (intersection of a horizontal line with a vertical line) in the diagram represents an atom. Each atom can have one of two spin states (it is convenient to think of it as spinning either up or down, but spin is actually a quantum mechanical property that doesn't have such a physical interpretation). Usually the spins are balanced around alternating spaces between the atoms (indicated by gray squares; note they always occur where the sum of the row and column numbers is odd). Balanced means that there equal numbers of inward pointing spins (spins on the bottom pointing up and spins on the top pointing down) as outward pointing spins. Note also that spins never point at any of the white squares (gaps between atoms). Sometimes spins are reversed from a balanced pattern, however, such as the 4 spins marked by dark gray triangles. Note that the space at r3c4 (row 3, column 4), marked N, is out of balance since 3 spins are pointing in and 1 spin is pointing out. Similarly, the space at r5c6, marked S, is out of balance with 1 spin pointing in and 3 pointing out. The N and S represent the north and south poles of a microscopic magnet. There is also a north pole at r1c2. Note that there are more north poles than south poles! This actually happens in certain spin ices, although it is not something that normally makes sense when we think of normal magnets where north and south poles always come in pairs.

The spins in the diagram can be represented by the following array. The gray spins in the diagram are marked by ***bold-italic*** text and break the checkerboard pattern of 0s and 1s. Note that there is one more row and one more column in the atom spin array than there is in the array of spaces between atoms; atom [0,0] with its spin is at the upper left side of space [0,0], atom [8,8] with its spin is at the lower right of space [7,7].

You must write a program that inputs an array of 1s and 0s as shown from a user-specified file. You may assume it is always a 9×9 array as illustrated and that there are 9 lines each containing 9 characters; there are no row or column labels in the file. Your program will do the following:

```
   c012345678
r0: 1010***0***0101
r1: 010***0***01010
r2: 101010101
r3: 010101010
r4: 10101***1***101
r5: 010101***1***10
r6: 101010101
r7: 010101010
r8: 101010101
```

- Find the locations of all the contained N and S poles and report them. In this case, you would report, “N at [1,2][3,4]. S at [5,6].” The precise formatting isn’t critical as long as the meaning is clear.
- Report whether the net polarity is N or S and how strong it is. In the example, there are 2 Ns and 1 S, so the net polarity is 1 N. So, in this case you would report “Net polarity: 1 N”, but in some situations you might report “Net polarity: 2 S”, “Net polarity: 0”, etc.

Note: you may assume that no spaces have 4 arrows pointing to or away from them. That represents a spin configuration that is physically impossible.

Note: You may use the following code to read the file. You may use it directly, or modify it if you wish to represent the data differently:

```
private static int SIZE = 9; // specified as constant for problem
private static int[][] readSpins() throws FileNotFoundException {
        // Ask user to select file (beware: dialog is sometimes raised in background)
        JFileChooser chooser = new JFileChooser();
        chooser.setFileFilter(new FileNameExtensionFilter("Text files", "txt"));
        if(chooser.showOpenDialog(null) != JFileChooser.APPROVE_OPTION) {
                throw new IllegalArgumentException("User did not select a file");
        }
        // Open the specified file
        final Scanner fsc = new Scanner(chooser.getSelectedFile()); // throws FileNotFoundException
        // Read the file, assuming proper formatting; may throw runtime exceptions for formatting errors
        final int[][] spins = new int[SIZE][SIZE];
        for (int row = 0; row < spins.length; ++row) {
                final String line = fsc.nextLine();
                assert line.length() == spins[row].length;
                for (int col = 0; col < spins[row].length; ++col) {
                        spins[row][col] = (line.charAt(col) == '0') ? 0 : 1;
                }
        }
        fsc.close();
        return spins;
}
```

9. Dispatching cabs (40 Points)

You are given two data files (samples are available on the contest website). Cabs.txt contains the location of available cabs at the beginning of the day. Each cab has a sequential ID number beginning with 0. Passengers.txt contains the location of passengers in the order that they call for a cab. The passenger names in order are 'A', 'B', 'C', etc.; there are at most 26 passengers. For the purposes of this problem, people take ***very*** long cab rides; once a cab is used, it is never available again. Write a program that, given names of files containing cab locations and passengers, assigns the closest cab to each passenger and dispatches it as the calls come in; your program will output a list of cabs and passengers (for example, "Cab 3 dispatched to passenger N."). If there are more passengers than cabs, stop after all cabs are dispatched. Since the cabs need to travel streets that form a grid pattern with only right angles, "distance" between two points is defined as the distance in the x direction plus the distance in the y direction (no square roots or trigonometry are involved).

Cabs.txt

```
10
   81.4   90.5
   12.6   91.3
   63.2    9.7
   27.8   54.6
   95.7   96.4
   15.7   97.0
   95.7   48.5
   80.0   14.1
   42.1   91.5
   79.2   95.9
```

Passengers.txt

```
10
   65.5    3.5
   84.9   93.3
   67.8   75.7
   74.3   39.2
   65.5   17.1
   70.6    3.1
   27.6    4.6
    9.7   82.3
   69.4   31.7
   95.0    3.4
```

Sample Output

```
Cab 2 dispatched to passenger A at city-block distance 8.50000.
Cab 0 dispatched to passenger B at city-block distance 6.30000.
Cab 9 dispatched to passenger C at city-block distance 31.6000.
Cab 6 dispatched to passenger D at city-block distance 30.7000.
Cab 7 dispatched to passenger E at city-block distance 17.5000.
Cab 3 dispatched to passenger F at city-block distance 94.3000.
Cab 8 dispatched to passenger G at city-block distance 101.400.
Cab 1 dispatched to passenger H at city-block distance 11.9000.
Cab 4 dispatched to passenger I at city-block distance 91.0000.
Cab 5 dispatched to passenger J at city-block distance 172.900.
```

Generating Random Integers

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.

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
```