CSC 495 – Problem Set 3 – Due Tuesday, January 31

Problem 3.R1: Day of the Week

Required Problem
Points: 50 points

Background

In the United States we have used the Gregorian calendar since September, 1752. This is the calendar we all know, in which all non-leap years have 365 days, and leap years have an extra day in February for a total of 366 days. The rules for leap years are as follows:

- Every year that is a multiple of 4 but not a multiple of 100 is a leap year.
- Every year that is a multiple of 400 is a leap year.

Any year that does not satisfy one of those two conditions is not a leap year. So, for example, 1900 was not a leap year (being a multiple of 100, the first rule does not apply, nor does the second since 1900 is not a multiple of 400). However, 2000 *was* a leap year by the second rule, and 2012 is a leap year by the first rule. Based on these leap year rules, and knowledge of the number of days in each month, you can do many date calculations. With the knowledge that January 1, 1753 was a Monday, you can compute the day of week for any date after 1753 (and into the future, assuming these rules still apply).

The Problem to Solve

You will be give a list of dates, each one in 1753 or later. For each date, calculate and print out the day of the week for that date. *You may not use any library functions in your solution — you should use only basic arithmetic.*

Elegance Considerations

Try to write clean code that uses as few special cases as possible. In addition, a good solution should be able to compute the day of week from the date without using any loops – just arithmetic.

Input and Output

The input will consist of an integer `n`, followed by `n` lines that each contain a date. Each date is given as three numbers: a month (1–12), a day of the month, and a year. All dates will be valid, and the year will always be ≥ 1753.

Sample Input

```
4
1 1 1753
7 4 1776
1 1 2012
1 24 2012
```

Sample Output

```
Monday
Thursday
Sunday
Tuesday
```

Problem 3.R2: More Power!

Required Problem
Points: 50 points
Ninja Points: 15 extra ninja points if your program can handle timespans of 10,000 years within 60 seconds. If you have a solution for this extended range, submit that under problem “A3R2big” (submit under the basic problem “A3R2” as well).

Background

Consider a large sign that uses seven-segment displays for a big counter display (see Problem 8 in Column 3 of *Programming Pearls*, on page 31, for a description of a seven-segment display along with pictures of each of the 10 digits on such a display). This sign is designed to be seen from a long distance, so each segment is a 180 watt light. Looking at the digit representations in *Programming Pearls*, the digit “1” takes two segments using 360 watts of power, but the digit “8” takes all seven segments using 1260 watts of power.

People are charged for electricity by energy used, which is power over time — typically this is kilowatt-hours (kwh), but in this problem we will measure milliwatt-hours (mwh). A milliwatt-hour is the amount of energy used by a expending 1 milliwatt of power for an hour. Running a single segment for one hour thus uses 180,000 mwh, and so running this segment for a single second takes $\frac{180{,}000}{3{,}600} = 50$ mwh. As the counter steps through a sequence of numbers, each requiring a different number of segments, the amount of energy used varies. The six-digit number “888888” displayed for 1000 seconds takes $50 \cdot 7 \cdot 6 \cdot 1000 = 2,100,000$ mwh.

The Problem to Solve

You are given three numbers: A starting number, the number of seconds between counter increments, and the total number of seconds (the timespan). You are to compute the total power (in mwh) required by the counter display over the full timespan. The counter may be as large as 14 digits (there are 14 digits in the U.S. government debt!), the timespan can be as large as 10 years, and the counter can be incremented at often as once per second — under those conditions, your program should be able to compute the total energy in less than 60 seconds (see the “Ninja points” note at the beginning of the problem for more demanding conditions). Note that the number of digits in the counter may vary over time, and leading zeros are never displayed (so those digits take zero power).

Hints and Techniques

Carefully consider the ranges of numbers given above, and make sure your code can handle the full range of values specified — both in terms of the speed of your algorithm and in terms of the size of the variables you use.

Input and Output

The input will consist of a number n followed by n lines that contain three numbers each: A starting counter value, a time between counter increments, and a total timespan (times are given in seconds). In the example below, the first sample input is a counter with value "888888" that stays at that value for 1000 seconds, and since the total timespan is also 1000 seconds the counter stays at that value the entire time. This is the example described at the end of the "Background" section, and the sample output shows that this uses 2,100,000 mwh of energy.

Sample Input

```
3
888888 1000 1000
111111 1000 1000
1000 1 1000
```

Sample Output

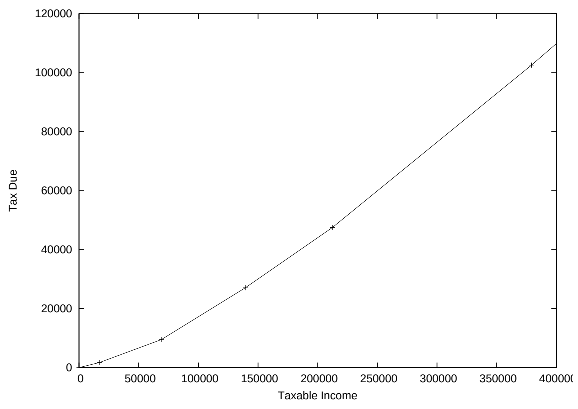
```
2100000
600000
820000
```

Problem 3.C1: A Taxing Problem

Challenge Problem
Ninja Points: This challenge problem is worth up to 20 base ninja points

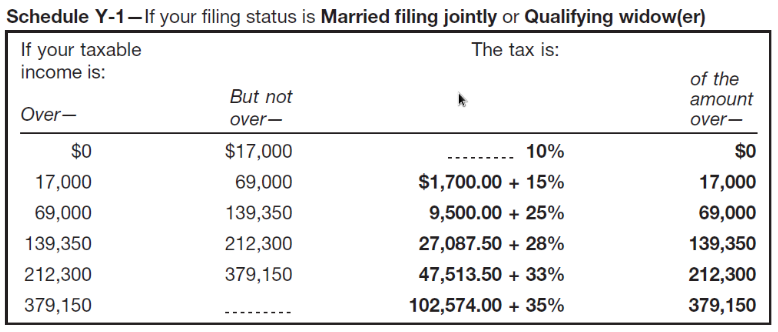
Background

In the United States, taxes are calculated using a *piecewise linear function*, meaning that if you graph the function it is a straight line between points where the marginal tax rates change. Those straight lines are the pieces that are linear. For example, the following graph shows the taxes due as a function of taxable income for a married couple filing jointly, using the 2011 tax rates.



It's hard to see precisely from this picture, but the graph is a straight line (i.e., linear function) between the marked points, which are the places where marginal tax rates change.

Problem 1 in Column 3 of *Programming Pearls* addresses the issue of computing taxes from a piecewise linear function, and the elegance of using generic code and a data table in order to compute the tax value. Here is how the IRS describes the rules for the 2011 tax rates that created the graph above:



Schedule Y-1—If your filing status is **Married filing jointly** or **Qualifying widow(er)**

If your taxable income is:		The tax is:	
Over—	*But not over—*		*of the amount over—*
$0	$17,000	--------- **10%**	**$0**
17,000	69,000	**$1,700.00 + 15%**	**17,000**
69,000	139,350	**9,500.00 + 25%**	**69,000**
139,350	212,300	**27,087.50 + 28%**	**139,350**
212,300	379,150	**47,513.50 + 33%**	**212,300**
379,150	---------	**102,574.00 + 35%**	**379,150**

Since this is a challenge problem, we're going to make the problem a little harder: Given samples of tax values, can you compute the tax rules? It turns out that if you are given at least two points in each of the linear segments, you can compute the full function!

The Problem to Solve

Your program is to read samples of tax data, where each sample consists of a taxable income amount and a tax amount. There are a few guarantees: Each tax rate will be an integer percent (so you won't have something like 27.5%, for example), and all tax bracket endpoints will be a multiple of 50. You are also guaranteed to receive at least two samples in each linear segment, although there may be more (there will be no more than 1000 samples total). The samples are in no particular order, however.

Hints and Techniques

When thinking about processing this data, think about "sweeping through" the data by increasing value of taxable income (note that this same way of visualizing the function was useful in the last problem set for the "danger points" problem). Can you detect when the marginal rate (i.e., the slope of the tax function) changes? That should be your first goal. Once you see how that is done, computing the bracket endpoints and marginal rates is a basic algebra problem.

Elegance Considerations

Think about Column 3, Problem 1, in *Programming Pearls* — how would you represent the function internally (i.e., how do you store the necessary bracket endpoints and rates) if you were given the table and asked to compute tax values? Note that the solution for this problem, given in the back of the book, talks about using a sentinel value — think about how that is useful. Once you have decided on that representation, use that as the goal for what you're calculating. As is the theme of Column 3, avoid special cases and make code as generic as possible.

Input and Output

The input will consist of an integer `n` followed by `n` lines, each containing a pair of values: a taxable income amount and a tax amount. Given this information, you are to compute a text representation of the tax rate structure similar to the information in the IRS table given earlier. You should not format into columns like that table, however: just give the information in text format, following the sample output format shown on the next page. Note that the bracket endpoints should be printed as integer values, the rate should be an integer percent, but the tax base amount should be a dollar amount with two decimal places. The sample data is generated from the same tax tables that were discussed in the "Background" section.

Sample Input

```
17
189800 41213.50
327800 85628.50
312900 80711.50
11300 1130.00
300 30.00
72600 10400.00
170100 35697.50
96500 16375.00
45900 6035.00
5600 560.00
3496400 1193611.50
112400 20350.00
25900 3035.00
64100 8765.00
310100 79787.50
2047800 686601.50
350000 92954.50
```

Sample Output

```
0..17000: 0.00 plus 10% of the amount over 0
17000..69000: 1700.00 plus 15% of the amount over 17000
69000..139350: 9500.00 plus 25% of the amount over 69000
139350..212300: 27087.50 plus 28% of the amount over 139350
212300..379150: 47513.50 plus 33% of the amount over 212300
379150 and up: 102574.00 plus 35% of the amount over 379150
```