Fall 2016 COP 3223H Program #5: Election Season Nears an End
Due date: Please consult WebCourses for your section

Objective(s)

1. To learn how to use 1D arrays to solve a problem in C.

Problem A: Expected Voting Outcome (votes.c)

For some critical small areas, your team has estimated probabilities that each individual voter actually shows up to the polls. In addition, you have probabilities that each voter will vote for your candidate, if they show up to vote. (Note we assume that there are only 2 candidates for this problem and that all voters who show up vote for one or the other.) Your job is to write a program that calculates the number of votes you expect to tally as well as the number of votes that your opponents expects to tally.

Input Specification

Input will come from stdin, but you will test your program via input files and file redirection (shown in class). The first line of input will have a single positive integer, *n* (*n ≤ 1000)*, representing the number of voters in the critical area in question. The second line of input will have *n* space separated doubles in between 0 and 1, representing the probability that a particular voter from the area will come to the polls. The third line of input will have *n* space separated doubles in between 0 and 1, representing the probability that the corresponding voter (to the previous list of doubles) will vote for your candidate. All doubles will be given to 2 decimal places precisely.

Output Specification

Output a single line to the screen with the following format:

```
Your expected votes: X, Your opponent's expected votes: Y
```

where X and Y are real numbers rounded to two decimal places. (Note: printf with the %.2lf code naturally does this.)

Sample Input

```
3
.6 .8 .9
.8 .2 .9
```

Sample Output

```
Your expected votes: 1.45, Your opponent's expected votes: 0.85
```

Problem B: What is my chance of winning? (win.c)

The US decides its presidential elections with a rather peculiar system called the electoral college. In practice, this means that each state has a number of electoral votes assigned to it and that the winner of the state takes ALL of those votes. For example, Florida is assigned 29 electoral votes. Thus, if a candidate wins by a margin of 51% to 49% of Florida voters, that candidate gets all 29 of Florida's electoral votes.

Polls can give some accurate probabilities of candidates winning a particular state. Putting these together with the number of electoral votes in a state, we can calculate the probability of a candidate winning an election.

The technique we will use to solve this problem is dynamic programming: The most number of electoral votes a candidate can have in the United States is 535. For this program, we'll just assume that the number of electoral votes is 1000 or less. The dynamic programming approach stores an array of size *n+1*, where *n* is the maximum number of electoral votes a candidate can achieve.

One Method to Solve the Problem - Dynamic Programming

Since the method to solve this problem is typically outside the scope of an introductory programming course, I'll simply outline the method for you here and the challenge for you will be implementing it from my description.

State by state information will come from an input file which will be piped into your program from the command line - both the number of electoral votes for a state as well as the probability of the candidate in question winning the state. (Read in the former as an integer and the latter as a floating point number in between 0 and 1, inclusive.)

As your program reads in information about each state, one by one, your program will update a list storing the probability of your candidate achieving each possible number of electoral votes.

In the beginning of the algorithm your array will look like the following:

```
probs--->[1,0,0,…,0]
```

since if we are considering no states, then the only possible outcome is 0 votes with a probability of 1, thus index 0 stores the corresponding probability of achieving that outcome, 1.

Let's say that the first state has 12 electoral votes with the candidate having a 45% chance of winning the state. Our algorithm will take each outcome stored in the probs array, and create a new list, tempprobs from it. Note that we have to initialize tempprobs to all 0s:

```
for (i=0; i<=MAX; i++)
    tempprobs[i] = 0
```

The way this will occur is that for each entry in the probs array we will apply two possibilities: the chance the candidate loses the current state and the chance the candidate wins the current state. In each case, we figure out the probability of the outcome by multiplying the old probability by the new probability. To figure out the electoral votes, what we do is add the new state to the old total (index in the probs array) in the case that the candidate wins the state, and we keep it the same if the candidate doesn't win the state. The difference in this implementation is that we do this per # of electoral votes, not each possible sequence of 2^n ways the states could vote. Thus, after processing the first state given in the example, our list looks as follows:

```
tempprobs--->[.55,0,0,0,0,0,0,0,0,0,0,.45,0…,0]
```

This indicates that our candidate has a 55% chance of having 0 electoral votes and a 45% chance of having 12 electoral votes. Before we move onto reading in the information about the next state, we must reassign the probs list. But we must do this manually with a for loop that copies each value, one by one:

```
for (i=0; i<=MAX; i++)
    probs[i] = tempprobs[i];
```

Now, let's say that the second state has 5 electoral votes and our candidate has a 60% chance of winning that state. At the very end of our second loop, after reassigning probs, our picture should look like this:

```
probs--->[.22,0,0,0,0,.33,0,0,0,0,0,0,.18,0,0,0,0,.27,0,…,0]
```

To obtain the first set of entries, we multiplyed .55 by .4, the probability of losing the second state to yield .22. If we lose the second state and previously had 0 electoral votes, we'd still have 0 electoral votes. To get the second set of entries, we multiplied .55 by .6, the probability of winning the second state to get .33 and added 19, the number of electoral votes in the second state to the old number of electoral votes (0) we had. To get the third set of entries, we multiplied .45 by .4, the probability of losing the second state to yield .18. The number of electoral votes in this scenario is simply 12, the old total. Finally, to get the last set of entries, we multiplied .45 by .6, the probability of winning the second state to yield .27. The number of electoral votes in this scenario is simply the old number(12) plus the 19 for winning this state.

To determine the winner, after processing each state, we can do a for loop through the second half of the probability list which corresponds to having more than half of all electoral votes. Just like assignment two, keep a total variable to keep track of the total number of electoral votes between all the states in the input.

Input Specification
The first line of input will contain a single positive integer, n ($n \leq 100$), the number of states. The following n lines will each contain information about one state. Each of these lines will contain two space separated values: e ($3 \leq e \leq 100$), the number of electoral votes for that state, followed by p ($0 \leq p \leq 1$), the probability that the candidate will win that state. The sum of all the electoral votes for all the states won't exceed 1000. The probabilities for the candidate winning each state will be given to at most 3 decimal places.

Output Specification
Output a single line of the format:

```
The candidate has a X chance of winning the election.
```

where X is the appropriate probability in between 0 and 1, inclusive, rounded to 3 decimal places.

Sample Input
```
10
29 .48
20 .7
10 .71
18 .43
11 .3
16 .21
15 .46
13 .75
16 .7
9 .65
```

Sample Output
```
The candidate has a .571 chance of winning the election.
```

Deliverables
Two source files:

1) *votes.c*, for your solution to problem A
2) *win.c* for your solution to problem B

All files are to be submitted over WebCourses.

Restrictions
Although you may use other compilers and coding environments, your program must run in Code::Blocks and gcc.

Grading Details

Your programs will be graded upon the following criteria:

1) Your correctness

2) Your programming style and use of white space. Even if you have a plan and your program works perfectly, if your programming style is poor or your use of white space is poor, you could get 10% or 15% deducted from your grade.

Note: As mentioned in class, the input specifications are there to help you. Those are the requirements I guarantee I'll stick to when testing your program. You don't need to check if the user enters values within those parameters.