2018 CCSC Eastern Conference Programming Competition

October 20th, 2018

Marymount University, Arlington, Virginia

This page is intentionally left blank.

Problem 1 — Big Blue Something

The Big Blue Something Company only sells blue things. The quality control engineer has set up a machine to spot check the colors of the items coming off the assembly line. The machine checks each item in several locations (from 2 to 100 locations per item), and it reports the color at each location. In order for an item to pass quality control, it must be blue in at least 50% of the locations.

The first line of input contains the number of items that have been checked. Each other line contains the colors detected in a single item. The colors are separated by spaces. There is no space at the end of the line. You can assume that colors will be spelled with lower case letters (a–z). For each item, output whether the item "passed" or "failed" the quality control check.

Example input:

```
5
blue red green
blue blue orange red
red green green red blue blue blue red
blue blue red
red orange blue
```

Example output (corresponding to the example input above):

```
failed
passed
failed
passed
failed
```

This page is intentionally left blank.

Problem 2 — Fruit Prices

You have been given the job of creating a new order processing system for the Yummy Fruit Company™. The system reads pricing information for the various delicious varieties of fruit stocked by YFC, and then processes invoices from customers, determining the total amount for each invoice based on the type and quantity of fruit for each line item in the invoice.

The program input starts with the pricing information. Each fruit price (single quantity) is specified on a single line, with the fruit name followed by the price. You can assume that each fruit name is a single word consisting of alphabetic characters (A–Z and a–z). You can also assume that prices will have exactly two decimal places after the decimal point. Fruit names and prices are separated by a single space character. The list of fruit prices is terminated by a single line consisting of the text `END_PRICES`.

After the fruit prices, there will be one or more invoices. Each invoice consists of a series of *line items*. A line item is a fruit name followed by an integer quantity, with the fruit name and quantity separated by a single space. You can assume that no line item will specify a fruit name that is not specified in the fruit prices. Each invoice is terminated by a line consisting of the text `END_INVOICE`. As a special case, if a line with the text `QUIT` appears instead of the beginning of an invoice, the program should exit immediately. The overall input will always be terminated by a `QUIT` line.

Example input:

```
orange 0.80
pomegranate 2.50
plum 1.20
peach 1.00
persimmon 1.75
lime 0.60
END_PRICES
persimmon 2
orange 3
peach 1
plum 10
pomegranate 5
END_INVOICE
peach 11
plum 5
orange 1
lime 9
END_INVOICE
QUIT
```

For each invoice, the program should print a single line of the form

```
Total: X.YY
```

where *X.YY* is the total cost of the invoice, which is the sum of the costs of all of the line items in the invoice. The cost should be printed with exactly two digits after the decimal point.

Example output (corresponding to the input shown above):

```
Total: 31.40
Total: 23.20
```

Problem 3 — Falling Dominoes

Consider the following configuration of dominoes represented by a sequence of characters:

```
||| /||| _
```

The | character represents an upright domino, the / character represents a domino that is tipping over, the _ character represents a domino that has fallen (and is horizontal), and a space represents a gap without a domino.

Your task is to implement a simulation showing the subsequent time steps following an arbitrary initial configuration, according to the following rules. On each time step,

- Each tipping domino will cause its neighbor to the right to change from upright to tipping, if the neighbor is currently upright
- Any tipping dominoes will change to horizontal
- All other positions in the sequence will remain unchanged

Your simulator should simulate as many time steps as necessary until a quiescent state is reached (where no further changes will occur.)

The input to the program is a series of domino configurations, one per line. You may assume that a domino configuration will have at most 40 positions. Leading and trailing whitespace on each line should be ignored. As a special case, if a line consists of the text `quit`, the program should exit immediately. You may assume that the input will be terminated with a `quit` line.

Example input:

```
||| /||| _
/|||||  /|_|/ ||
||//||||||
||__ _|_ __|
quit
```

For each input configuration, the program should print the configurations resulting from each time step of the simulation, starting with the initial configuration, until a quiescent state is reached. After printing the final state, it should print a line consisting of the text `END`.

Example output (corresponding to the example input above):

```
||| /||| _
||| _/|| _
||| __/| _
||| ____/ _
```

```
||| _____ _
END
/|||||  /|_|/ ||
_/||||  _/_|_ ||
__/|||  ____|_ ||
____/||  ____|_ ||
_____/|  ____|_ ||
______/  ____|_ ||
_______  ____|_ ||
END
||//||||||
||__/|||||
||____/||||
||_____/|||
||______/||
||_______/|
||________/
||_________
END
||__ _|_ __|
END
```

Problem 4 — Postfix Calculator

Postfix notation, also known as "reverse Polish notation" (RPN), is a way of writing mathematical expressions in an unambiguous way. A postfix expression has the form

operand operand operator

In other words, the operator (such as $+$ for addition, $-$ for subtraction, etc.) goes *after* the operands. Contrast this with an *infix* expression, where the operator goes *between* the operands. Infix expressions are inherently ambiguous. For example,

$$a + b \times 3$$

could mean either

$$a + (b \times 3)$$

or

$$(a + b) \times 3$$

In practice, we treat the first interpretation as the "correct" one because multiplication has a higher precedence than addition. In postfix notation, no such ambiguity exists. The first interpretation (do the multiplication first) would be written as

$$a\ b\ 3\ \times\ +$$

The second interpretation (do the addition first) would be written as

$$a\ b\ +\ 3\ \times$$

Your task is to implement a calculator for evaluating postfix expressions consisting of integers, variables, and the operators `+ - * / =`. The operators `+ - * /` mean addition, subtraction, multiplication, and division, respectively. The `=` operator means variable assignment: its first operand should be the name of a variable, and its second operand is the value that should be assigned to the variable. The result of evaluating an assignment expression is the value assigned. Variable names are single lower case letters (`a`–`z`). Literal integers are a sequence of one or more digit characters (`0`–`9`), optionally preceeded by a minus sign (`-`) to indicate a negative value. Both input values and results can be negative.

Each line of input to the program consists of a single postfix expression to be evaluated. On each input line, the tokens (variables, integer literals, and operators) will be separated by one or more space characters. Leading or trailing space characters are possible, and should be ignored. As a special case, if the input line is `quit`, the program should exit immediately. You can assume that all inputs will be terminated by a `quit` line.

For each input expression, the program should print a single line of text containing the result of evaluating the expression. You may assume that any variables used will be defined using the assignment operator (=) before being used as a value. You may also assume that the second operand of a division (/) operation will never be zero. Division operations should truncate: for example, the expression `11 2 /` should evaluate to the result `5`.

Example input:

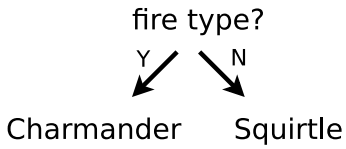
```
a 3 =
b 3 a * =
a b 3 * +
a b + 3 *
19 6 -
4 c 11 = *
c
-3 5 *
3 41 -
quit
```

Example output (corresponding to the example input above):

```
3
9
30
36
13
44
11
-15
-38
```

Problem 5 — Pokédex 2.0

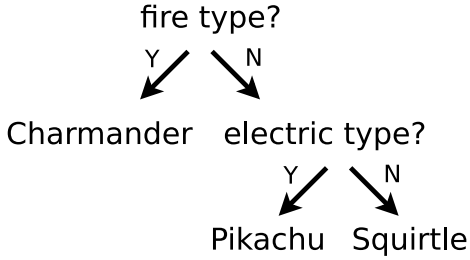
As the lead engineer developing the Pokémon classification system for the new Pokédex 2.0, you decide to incorporate advanced artificial intelligence technology in the form of an *expert system*. The expert system works by maintaining a decision tree of yes/no questions leading to classifications of Pokémon. This is the initial decision tree:



The expert system works by asking the user to answer a series of yes/no questions which navigate from the root of the decision tree to a leaf. When a leaf is reached, the expert system asks if the Pokémon named by the leaf is the correct Pokémon. If it is not, the system prompts the user to enter the correct Pokémon, and then to enter a question which would distinguish the correct Pokémon from the incorrect one, and updates the decision tree accordingly. Here is part of an example session, with the user's responses indicated in *italics*:

```
P:Begin!
P:fire type?
N
P:Are you Squirtle?
N
P:You are?
Pikachu
P:Distinguish Pikachu vs. Squirtle
electric type?
P:Is Pikachu electric type?
Y
P:Pikachu classified
```

The interaction shown above would result in the decision tree being updated as follows:



Message format	Meaning	When printed
P:Begin!	Starting to classify a Pokémon	At start of classification
P:*question*	Prompt yes/no answer to classification question	During classification
P:Are you *name*?	Prompt yes/no answer to confirm classification as *name*	When leaf node reached
P:Identified as *name*	Named Pokémon identified correctly	After correct classification
P:You are?	Prompt for correct Pokémon name	After incorrect classification
P:Distinguish *new* vs. *old*	Prompt for distinguishing question	After getting new (correct) name
P:Is *new question*	Prompt whether *new* is yes or no answer to *question*	After distinguishing question
P:*name* classified	Named Pokémon was added to the decision tree	After updating decision tree

Figure 1: Expert system message types: *italics* indicate placeholders

After the decision tree update shown above, the expert system can correctly classify Pikachu (again, user input in *italics*):

```
P:Begin!
P:fire type?
N
P:electric type?
Y
P:Are you Pikachu?
Y
P:Identified as Pikachu
```

The program should repeatedly use the decision tree (starting with the initial decision tree shown above) to classify Pokémon. If the classification is correct, the system prints a message indicating a successful classification. If a classification is incorrect, the program should add the new Pokémon and a distinguishing question to the decision tree. For each yes/no response, the input will be `Y` or `N` on a single line. Inputs naming a new Pokémon and specifying a new distinguishing question will also each be on a single line. As a special case, if an input line consisting of the text `Q` is given in response to an initial classification question, the program should exit immediately.

As the expert system is executed, it should generate the messages as indicated by Figure 1, each on a separate line.

Example input:

```
N
N
Pikachu
electric type?
Y
N
Y
Y
N
Y
N
Magnemite
mouse Pokemon?
N
N
Y
N
Y
Q
```

Example output (corresponding to the input shown above):

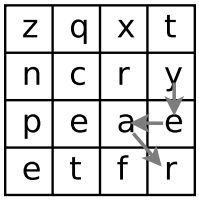
```
P:Begin!
P:fire type?
P:Are you Squirtle?
P:You are?
P:Distinguish Pikachu vs. Squirtle
P:Is Pikachu electric type?
P:Pikachu classified
P:Begin!
P:fire type?
P:electric type?
P:Are you Pikachu?
P:Identified as Pikachu
P:Begin!
P:fire type?
P:electric type?
P:Are you Pikachu?
P:You are?
P:Distinguish Magnemite vs. Pikachu
P:Is Magnemite mouse Pokemon?
P:Magnemite classified
```

```
P:Begin!
P:fire type?
P:electric type?
P:mouse Pokemon?
P:Are you Magnemite?
P:Identified as Magnemite
P:Begin!
P:fire type?
```

Problem 6 — Boggle™

In the game Boggle™, letters are arranged randomly on a grid, and the goal is to find words that can be spelled by starting at a particular grid location and continuing up, down, left, right, or diagonally to neighboring grid locations. Visiting the same grid location twice or more within a single path is not allowed. For example, consider the following grids. The grid on the left is an example configuration. The arrows on the grid on the right show a path which spells the word "year" within the example configuration.

z	q	x	t
n	c	r	y
p	e	a	e
e	t	f	r



Your task is to write a program that, when given an arbitrary list of words and one or more arbitrary Boggle grids, finds all paths within the grids which spell any of the words in the word list. A grid can be any size up to 26x26, and is not necessarily square. It is possible that a single word will occur multiple times in a single grid, in which case the program should find paths for all occurrences. A path is listed as a comma-separated (no spaces) sequence of coordinates, where letters A–Z specify the row (A is the top row) and numbers 0–25 indicate the columns (0 is the left column). For example, the coordinate B2 indicates the second row and the third column. All of the words and the configuration of the Boggle grid will consist exclusively of lower-case letters (a–z).

The input starts with a list of words, one word per line, terminated by a line consisting of the text `END`. (Note that `END` should *not* be considered to be a word.) You can assume there will be no more than 200 words, and that no word is more than 20 letters long. Then, there is a sequence of 1 or more Boggle grid configurations. A grid configuration starts with a line specifying the grid size as the number of rows followed by the number of columns (both integers, separated by a space.) Next in the grid configuration is one line per grid row specifying the letters in that row. As a special case, if the grid size line consists only of the value `-1`, the program should exit immediately.

Example input:

```
cat
car
care
year
tree
END
4 4
```

```
zqxt
ncry
peae
etfr
3 5
ranqm
kects
ydcaw
-1
```

For each board configuration in the input, the program should print a single line reading `Solutions:`, followed by the list of paths spelling the words in the word list. The paths should be ordered lexicographically by comparing pairs of coordinates. Coordinates should be compared first by row, then by column. (Note that, for example, the coordinate `C9` should be *before* `C10`, not after, because `C9`'s column is less than `C10`'s column.) To break ties where one path is a prefix of another, the shorter path should be ordered before the longer path.

Example output (corresponding to the example input above):

```
Solutions:
A3,B2,C1,D0
B1,C2,B2
B1,C2,B2,C1
B1,C2,B2,C3
B1,C2,D1
B1,C2,D3
B1,C2,D3,C3
B3,C3,C2,B2
B3,C3,C2,D3
Solutions:
B2,A1,A0
B2,A1,A0,B1
B2,C3,B3
C0,B1,A1,A0
C2,C3,B3
```

Problem 7 — Collections

It is important for Gerald Thermaine to keep things organized. He has large collections and once per year he rearranges the collections to their optimal configurations.

For each of his collections, Gerald wants to know how many ways there are to arrange the items in a line. Some of the items in a collection are indistinguishable. Others are special. For example, he has a collection of bottlecaps. He has 8 unique bottlecaps, as well as 15 bottlecaps of his favorite drink (Yum-E Soda) and 10 bottlecaps of his second-favorite drink (PrettyGood NRG Drink).

The first line of input gives the number of collections that Gerald has. Each other line of input represents one collection. The first number on a line is the number of unique items in the collection. Each other number on the line represents a group of identical items. There may be up to 10 different groups of identical items. Each group may contain 2 to 200 identical items. The first collection in the sample input represents Gerald's bottlecap collection. The total number of ways to arrange a collection will not exceed 9,200,000,000,000,000,000. For each collection, output the number of ways Gerald could arrange his collection in a line.

Example input:

```
7
8 15 10
6 8 5
4 5 4 3
1 200
4
0 100 10
0 150
```

Example output (corresponding to the example input above):

```
18298818222676992000
25141596480
1210809600
201
24
46897636623981
1
```

This page is intentionally left blank.