2012 Programming Competition
Problem Set

Women In Computer Science
Arizona State University
Table of Contents

1. Triangle Wrangler ................................................................. 1
2. Boolean Minimization ............................................................ 2
3. Kurt’s Klunkers ................................................................. 5
4. Coin Purse Scuffle ............................................................... 6
5. Stack Overflow? ................................................................. 7
6. Can You Decide It for ME? .................................................... 11
7. What’s the Point? ............................................................... 13
8. Don’t Cry ........................................................................... 15
9. The Social Network .......................................................... 17
10. Checkmate ....................................................................... 19
1. Triangle Wrangler

Background

Everyone knows triangles are composed of three points; however, special triangles called lattice triangles are formed by having vertexes at integer coordinates in a coordinate plane. Your job is: given an $M \times N$ grid, find all the possible lattice triangles within the given grid. Below is an example of a grid along with all the possibilities.

![Figure 1. A grid of lattice triangles](image)

Specification

The following lines will be two integers corresponding to $M$ and $N$ ($0 \leq M, N \leq 200$).

The output of your program should be the number of triangles for each case. This will be a single integer value, one per case.

<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Sample Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>4</td>
</tr>
<tr>
<td>1 2</td>
<td>18</td>
</tr>
<tr>
<td>2 2</td>
<td>76</td>
</tr>
</tbody>
</table>
2. Boolean Minimization

Background

The following information is provided for the benefit of readers who may be unfamiliar with the problem. More experienced readers may skip directly to the problem description in the next section.

The minimization of logical expressions is a problem with many applications in both computer science and engineering. The underlying set problem is of particular interest to computer scientists both for its manifold applicability to other problems as well as its intrinsic computational difficulty. Boolean minimization also finds extensive use outside the realm of computer science, such as in the minimization of digital logic circuits.

Performing Boolean minimization requires an understanding of the underlying principles of Boolean logic. Boolean logic involves operations on the values true and false, commonly denoted as 1 and 0, respectively. There are three basic operations, one unary and two binary. These operations are given in order of precedence as follows:

- **negation (not):** produces a true value if the operand is false and a false value if the operand is true. Commonly denoted by appending a prime symbol (') to the operand. The result is said to be the complement of the operand. Any variable, complemented or uncomplemented, is called a literal.

- **conjunction (and):** produces a true value if and only if both operands are true. Because this operation is analogous to multiplication, it is commonly denoted simply by writing the operands next to one another. The resulting term is called a product.

- **disjunction (or):** produces a true value if either of the operands is true. Because this operation is analogous to addition, it is commonly denoted with a + symbol. The resulting term is called a sum.

In accordance with the above definitions, we may speak of Boolean expressions as being written in sum of products form. A Boolean sum of products consists of a sum of terms which are themselves composed of literals joined only by conjunction. For example, \( a + bc + a'c \) is a sum of products.

Every Boolean expression can be associated with a set of vectors, where the elements of each vector correspond to the ordered set of variables in the expression. Expressions are said to cover the set of vectors for which they evaluate to true.

A sum of products is said to be minimal if and only if no other sum of products exists which covers the same vectors while having fewer literals. It is possible for a single set of vectors to be covered by more than one minimal sum of products expression. In the case that multiple minimal sums of products exist, all must have the same number of literals.
Problem Description

Given a complete set of vectors and result values for a Boolean logic function, compute all possible minimal sums of products expressions which define (that is, cover the true values of) the function.

Hint

Begin by finding terms that differ only in one bit position, e.g., abc + abc’ → ab.

Specification

Input

The first line of the input file contains an integer \( n \) giving the number of independent variables in the function. The second line contains the single character identifying the independent variable, followed by a colon. Following the colon and on the same line are the characters identifying the independent variables in descending order of significance.

The remaining \( 2n \) lines contain the complete set of bit vectors in the domain of the function and the value they produce. Each vector and result value has its own line. The result value is given first, followed by a colon. Following the colon and on the same line are the values of each variable in the bit vector, again given in descending order of significance.

You may assume that the lines for each bit vector are given in ascending order by their numeric binary value.

Output

At the beginning of the output, your program should write the number of minimal sum of products expressions it has computed, followed by a newline.

Your program should then write the equations for the function, each followed by a newline. The dependent variable is given first, followed by an equals (\( = \)) sign, and then the product terms separated by the conjunction sign (\( + \)) as needed. Conjunction (negation) is indicated by appending an asterisk (\( ' \)) to the variable to be negated.

Product terms and the literals within them may be given in any order. Any whitespace present in the output is ignored for purposes of evaluation, but its use is encouraged because it makes the output easier to read.
<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Sample Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>y:abc</td>
<td>y = ac + a'b'</td>
</tr>
<tr>
<td>1:000</td>
<td>y = ac + a'c'</td>
</tr>
<tr>
<td>1:001</td>
<td></td>
</tr>
<tr>
<td>1:010</td>
<td></td>
</tr>
<tr>
<td>0:011</td>
<td></td>
</tr>
<tr>
<td>0:100</td>
<td></td>
</tr>
<tr>
<td>1:101</td>
<td></td>
</tr>
<tr>
<td>0:110</td>
<td></td>
</tr>
<tr>
<td>1:111</td>
<td></td>
</tr>
</tbody>
</table>
3. Kurt’s Klunkers

Problem Description

You are the lot manager at Kurt’s Klunkers, a used auto mall. Kurt’s Klunkers has been very successful due to its novel way of selling cars: having each customer enter a price for each of the cars they want, and at the end of the day assigning each buyer to the car in a way that earns the store the most return. Until now the owner, Kurt, has been doing these calculations by hand. You have been hired to implement a system to do these calculations automatically, sending Kurt into retirement.

In this problem, you are to obtain a list of prices that customers are willing to pay for a set of automobiles. Then, you are to match each customer with a car in a way that earns the maximum possible amount of money for the auto lot.

Specification

The first line of input will contain a list of automobiles, comma delimited. All following lines will contain the name of a customer with the prices they are willing to pay for each item, respectively. For instance, in the example below Aaron is willing to pay $12 for the Mercedes, $4 for the Oldsmobile, and $2 for the Ford Focus.

Output the customers in the order they were received along with the car they bought. Finally, output the amount of money that Kurt makes on the given day.

Sample Input

Mercedes,Oldsmobile,Ford
Aaron,12,4,0
Bob,8,7,6
Chris,7,5,2

Sample Output

Aaron,Mercedes
Bob,Ford
Chris,Oldsmobile
23
4. Coin Purse Scuffle

Background

“This will be $14.”

“Here is $20.”

“Sorry, I don’t do change.”

“Fine, here’s $10 and $5. Keep the change.”

Due to many vendors not taking credit or debit, it is often useful to carry cash. However, sometimes some of these places don’t have change. Therefore you have to pay a bit more for a product. Your job is: given a price, try to match the price as close as possible, without going under, while using the least amount of bills or coins. This will allow you to reduce the amount you will have to pay as well as leaving you the most options to pay for later products.

Specification

You will take in a file as input. The first line of the input will be the amount you are being charged in cents \((1 \leq M \leq 1000)\). The next line will be the number \(N\) of coins and bills you have. The following \(N\) lines will give the denomination in cents for each bill or coin you have in your possession. It is possible for bills or coins of the same denomination to appear multiple times.

The output will be a single line with the minimum amount you have to pay and how many bills or coins you will need.

<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Sample Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1400</td>
<td>1500 2</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td></td>
</tr>
</tbody>
</table>
5. Stack Overflow?

Background

The following information is provided for the benefit of readers who may be unfamiliar with the problem. More experienced readers may skip directly to the problem description in the next section.

Virtually all general-purpose electronic computers make use of a stack data structure in memory to manage the data used by program subroutines. The MIPS microarchitecture uses a full stack, meaning that the value of the top-of-stack pointer register always stores the address of the topmost used position on the stack. This stack is also descending, meaning that the stack is grown by decrementing the value of the stack pointer (the “top” of the stack is always located at an address below the base of the stack in memory). The value of the stack pointer is assigned to a dedicated register, denoted $sp. The space on the stack reserved for use by a particular subroutine is referred to as that routine’s stack frame, customarily delineated by an additional pointer (the frame pointer, stored in another register $fp).

![Figure 2. MIPS 4 byte aligned, full descending stack (typ.)](image)

As subroutines are called, each stores any needed local (auto) variables by pushing them onto the stack. When a subroutine returns, all local variables are popped back off the stack and the space used is reclaimed. However, if many subroutines call other subroutines in sequence before returning, it is possible for the stack to grow beyond the space allocated for its use. This results in a (typically) fatal error known as a stack overflow.

In order to prevent overflows, it is common for compilers to allocate several times more stack space to programs than is ever likely to be used. However, in tightly constrained environments (such as in embedded devices) this approach may waste memory that might
be put to other uses. Precisely how much stack space a program may use when run is thus a very valuable question to have answered...

**Problem Description**

Given a short program in MIPS assembly language, give the maximum stack usage by the program in bytes and the execution trace (or traces) that result in this maximum usage.

**Hint**

Think about how you will determine the total stack usage before you write your parser. *Not all of the information given is needed to correctly solve the problem.*

**Specification**

**Input**

The input to your program shall consist of a simple MIPS assembly language program. MIPS assembly is a very simple language and widely known, so no attempt will be made to fully describe it here. To avoid confusion, however, you should note the following points:

- Jump instructions take a label name for their destination. Labels consist of a name followed by a colon; they cannot contain whitespace.
- Register-to-register instructions accept two source registers and one destination register, with the destination register given first. Registers are specified using a dollar sign ($) followed by the register number or mnemonic. There are 32 numbered registers, $0 to $31. The only mnemonics used in the test inputs are $ra, $sp, and $fp.
- Immediate instructions accept a single immediate value.
- Load and store instructions are specified with the value register first, followed by the register giving the source or destination address. Addresses may be enclosed in parentheses and prefixed with a small offset value. The offset value may be positive or negative.

For ease of processing, you may assume that

- The program consists only of labels and machine instructions, with no data declarations or other directives
- There is only one label or machine instruction given per line
- All subroutines are invoked using the *jal* (jump and link) instruction, whose target shall be given as a valid label name
- As a corollary to the previous point, no subroutines shall be invoked using a computed or variable offset (i.e. a function pointer)
- Subroutines modify the stack pointer register twice at most (once on entry, and again on exit)
• The value of the stack pointer register ($sp) is only modified by means of the addiu (add immediate unsigned) instruction, whose immediate value is always given in decimal
• All subroutines restore the caller’s stack pointer when they return
• All subroutines return by means of an unconditional jump to the location in register $ra.
• The sole entry point of the sample program is always called main.

Output

The output of your program shall consist of two or more lines. The maximum stack usage of the input program in bytes shall be given on the first line. All following lines shall give the execution trace(s) which produce the given stack usage. Each trace consists of the names (labels) of each subroutine, given in their order of invocation with the entry point (main) appearing first.
Your program should not attempt to determine whether any of the subroutines in the input program is called more than once. In general, this is computationally undecidable (by reduction to the halting problem). Stack traces shall not include the same subroutine twice.

Sample Input

x:
   addiu $sp,$sp,-8
   sw $fp,4($sp)
   move $fp,$sp
   li $2,3
   move $sp,$fp
   lw $fp,4($sp)
   addiu $sp,$sp,8
   j $ra

y:
   addiu $sp,$sp,-32
   sw $ra,28($sp)
   sw $fp,24($sp)
   move $fp,$sp
   jal x
   addiu $2,$2,5
   move $sp,$fp
   lw $ra,28($sp)
   lw $fp,24($sp)
   addiu $sp,$sp,32
   j $ra

main:
   addiu $sp,$sp,-32
   sw $ra,28($sp)
   sw $fp,24($sp)
move  $fp,$sp
jal   y
move  $sp,$fp
lw    $ra,28($sp)
lw    $fp,24($sp)
addiu $sp,$sp,32
j     $ra

Sample Output

72
main y x
6. Can You Decide It for ME?

Background

As you should know, a formal system consists of a set of axioms and set of inference or production rules. Theorems are results that can be obtained upon the axioms and the inference rules of the system. Deciding if a statement is a theorem of a given formal system is not a trivial matter; in fact, for some systems it is not even possible, as we know by Gödel's incompleteness theorem.

Problem Description

ME is a formal system with an infinite number of axioms and just one production rule. There are only three symbols that may appear in any axiom of ME: the capital letters M and E, and the question mark (?). In spite of the infinite number of axioms, there is an easy way of identifying them:

- **Axiom definition:** An axiom of ME is a string of the form xM?Ex? where x is a string of one or more ? symbols.

Notice that the word xM?Ex? itself is not an axiom, because x is not a valid symbol of ME; it is just a pattern to describe the axioms. But ??M?E??? is indeed an axiom, because it fits the pattern, just replacing x with ??

There is another type of valid strings in the system ME: the theorems. Every axiom is a theorem of ME. In addition, the following production rule gives rise to an infinite number of theorems:

- **Production rule:** If xMyEz is a theorem then xMy?Ez? is also a theorem, where x, y, z are strings of one or more ? symbols.

For instance, the string ??M?E??? is a theorem, because it is an axiom, and the string ??M??E???? is also a theorem, since it can be obtained upon the theorem ??M?E??? and the production rule.

Can we decide if a given string is a theorem of ME? Sure! There is a decision algorithm for the system ME. But your goal is less general: just write a program that reads strings with at most 50 symbols and decides if they are theorem of ME.
Specification

The input begins with a single positive integer $N$, on a line by itself, indicating the number of input lines following. Each input line contains a string with 1 to 50 characters (without spaces), but not necessarily restricted to the symbols of ME.

For each input line, the program must print an output line with the word "theorem" if the test line contains a theorem, or the word "no-theorem" in other case.

<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Sample Output</th>
</tr>
</thead>
</table>
| 7
??M?E???
xM?Ex?
??M??E????
M?E?
??m?e???
?M?E??
12M12E???? | theorem
no-theorem
theorem
no-theorem
no-theorem
theorem
no-theorem |
7. What’s the Point?

Problem Description

You are a surveyor in a major city. Often, your coworkers come to you with requests for requests regarding the ownership of land. As part of your job, you have to compute whether or not a certain location is within a certain piece of property. This job is tedious, and since your company cut the budget for interns, you are left to do this yourself. To get around this responsibility, you have decided to write a program to compute the answers for you.

Figure 3. The parcel defined in the sample input.

In this problem you will write a program to compute if a set of points are in the interior of the parcel of land defined in each test case. Points on the edge do not count as within the polygon.

Specification

The input will consist of multiple lines; the first line will consist of the points that define the polygon. The last point will be equivalent to the first point, signifying the end of the polygon definition. Each point is defined by \([X, Y]\), where \(X\) and \(Y\) correspond to floating point coordinates on the Cartesian plane. Point component values are enclosed by square braces ([ and ]) and comma delimited.

All following lines will consist of a single point that is to be tested to see if it is in the interior of the polygon.
Output will consist of lines containing the word “True” or “False”, depending on whether the point is within the polygon specified in the first line.

Sample Input
[0.0, 0.0], [1.0, 0.0], [1.0, 1.0], [0.0, 1.0], [0.0, 0.0]
[0.5, 0.5]
[1.0, 0.5]
[0.998, 0.998]

Sample Output
True
False
True
8. Don’t Cry

Problem Description

You live every day with the habit of waking up and drinking a full glass of milk. Every day you wake up and shake the carton of milk to know how much milk there is left. However, recently your favorite milk provider has changed the packaging and you are no longer able to guesstimate the amount of milk that is in your favorite container of milk. One morning, you woke up, shook the carton of milk, and it seemed fine... but everything was not fine: when you poured, the glass was only half full!

Furious, you set your mind to create an algorithm that can tell you how much milk there is left. Using the diagram below you can make some observations about the contents of the milk carton and determine the volumetric content of that carton.

![Figure 4. The milk carton](image)

Given this, you can observe that you will be dealing with four parameters: \( l \), \( w \), \( h \), and \( \theta \). You will only be getting measurements when the carton of milk is tilted and a triangle of empty space is formed at the top of the container. Also, assume that the thickness of the carton itself is negligible and that the milk inside is refreshing.

Specification

The input file will contain multiple test cases delimited by a new line. The order of the parameters is the following: \( l, w, h, \theta \).

For each input line there must be an output line containing the volume in milliliters, up to 3 decimal places.
<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Sample Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 6 19 70</td>
<td>937.555 mL</td>
</tr>
</tbody>
</table>
9. The Social Network

Problem Description

You are the proprietor of a new social networking site. Your marketing analysts have come to you noting that the users on your site are demanding a new feature: they want to know who is in their “friend of a friend of a friend” network. Wanting to please your users, you have decided to implement this feature.

In this task, you are to compute the “friend of a friend of a friend” network for a set of nodes.

Figure 5. A potential instance of the social network

Specification

First, you will read in an adjacency list, which consists of edges defined in the following format: \( A, B \), where \( A \) and \( B \) the integer identifiers of two users in the network. The edge list will terminate when the input reads a line containing only the text \texttt{STOP}. Queries for the network occur on all subsequent lines. There is no guarantee that the lowest identifier will be 1. The network shown in Figure 1 is used in our example.

For each query, print a single line that contains the list of nodes in the user’s “friend of a friend of a friend” in sorted order. If there are no users in a query’s computed network, print \texttt{NONE}.
<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Sample Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>1, 2</td>
</tr>
<tr>
<td>1, 5</td>
<td>NONE</td>
</tr>
<tr>
<td>4, 5</td>
<td></td>
</tr>
<tr>
<td>3, 2</td>
<td></td>
</tr>
<tr>
<td>5, 2</td>
<td></td>
</tr>
<tr>
<td>4, 3</td>
<td></td>
</tr>
<tr>
<td>6, 4</td>
<td></td>
</tr>
<tr>
<td>STOP</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
10. Checkmate

Problem Description

The chess club is hosting a tournament and needs to determine which players will play each other. To do this, they want to have a bracket styled tournament. Each player must play every round, regardless of whether or not they have won or lost in the past. Each player will play the adjacent winner or loser, depending on whether they won or lost the last round. In other words, a winner will never play a loser and vice versa. In the event there are an odd number of players, one player is allowed to be matched against themselves. If this is the case, it will always be the last player in either the winning or losing set, depending on which is odd. If no players have entered, then the “zero” player will play against himself. In the case of a tie, both players will be given as “winners”.

Specification

Input will be two lines. The first line will be the number of players in the tournament. The second line will be the winners of first round. Each player is identified by an integer.

Output will be the pairings for the next round, one per line. Integers on each line are space delimited. If a player is matched against himself, just list that player once.

<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Sample Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1 3</td>
</tr>
<tr>
<td>1 3 5 7</td>
<td>5 7</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>