The Generalised 7-11 Problem

This is a small Constraint Satisfaction problem. The author of this étude adapted it from a book called "How to Solve it: Modern Heuristics" (2nd edition) by Zbigniew Michaelwicz and David B. Fogel, and published by Springer in 2004. The problem has two parts.

The 7-11 problem

There is a chain of shops in Australia (and elsewhere) called 7-11. The author believes that the parent chain in the USA used to be open from 7 a.m. to 11 p.m., making this like our Night and Day shops. The story took place before 1991, so the 1c and 2c coins were still in circulation, and customers could be expected to pay exact sums with coins.

One day a customer arrived at a 7-11 shop and chose four items, all costing at least one cent. She took them to the counter. The clerk took his calculator, pressed some buttons, and said "The total price is $7.11."

The customer smiled and asked "Is that because this is a 7-11 shop?"

The clerk was too tired to get the joke, and replied, "Of course not. I multiplied the prices of these four items, and that's what it comes to."

The customer was surprised, and said "Shouldn't you have *added* the prices?"

"Oh heck yeah," yawned the clerk. Back to the calculator, more button presses, and two pairs of eyebrows went up. "It's still $7.11 when I add!" he said.

The 7-11 problem is this. Let the prices be $a \leq b \leq c \leq d$. Find the values of a, b, c, d such that $a + b + c + d = \$7.11$ and $a \times b \times c \times d = \$7.11$. There is a unique solution.

The Generalised 7-11 problem

Isn't it amazing that $a + b + c + d$ and $a \times b \times c \times d$ both come out to $7.11 and that 7-11 is the name of a shop?

Wait a minute. *Is* it amazing? Let's find out.

The generalised 7-11 problem is this: find all amounts $\$x.yz$ between \$1.00 and \$9.99 such that there is a unique solution to $a + b + c + d = \$x.yz$ and $a \times b \times c \times d = \$x.yz$ and $a \leq b \leq c \leq d$.

Task

Because numbers like 0.01 cannot be represented exactly in binary floating-point arithmetic, it is wise to avoid it for problems like this. Therefore, you **must not** use floating-point arithmetic in any form in your program, *including formatting* the output.

3.1 Part 1

Write a program called s711.c, s711.py, S711.java, or whatever, that solves the 7-11 problem and writes one line of output to the standard output stream, giving values for a, b, c, and d in that order, separated by single spaces. The values should be written in $\$x.yz$ form with a dollar sign, one decimal digit, a decimal point, and two decimal digits. You must get *exactly* this output whatever locale the program is run in, so your language's "how to print money" features *must not* be used.

If your program is written in C and compiled with -O, it should only take a few milliseconds—too fast to measure accurately.

3.2 Part 2

Write a program called g711.c, g711.py, G711.java, or whatever, that solves the *generalised* 7-11 problem. It should write one line of output to the standard output stream for each $\$x.yz$ that has a **unique** solution, followed by a summary line saying "nn solutions" where nn is the number of unique solutions found. Except for the summary line, each line should have the form t = a + b + c + d where each of the numbers is written with a dollar sign, one decimal digit (even if it's zero), a decimal point, and two decimal digits. You must get *exactly* this output whatever locale the program is run in, so your language's "how to print money" features *must not* be used. The

equal sign, plus signs, and single spaces around the equal and plus signs *must* appear.

Output should be written in increasing order of t.

If your program is written in C and compiled with -O, it should take under 10 seconds to try all 900 possibilities. (The model answer takes 0.05 seconds.)

(Pair 2)

Notes

This should be a fairly easy task. Experienced programmers, having taken less than half an hour to read the étude, might be expected to take less than half an hour to code each part.

You do not need any data structures other than a few numbers, and you do not need any control structures more exotic than a loop or conditional statement. What you *do* have to take care with is arithmetic.

- You need to consider numbers like $0.01, $1.73, and so on, which cannot be represented exactly in binary floating-point arithmetic. Straightforward code using 0.01, 1.73, and so on *will* go wrong. That is why the use of floating-point arithmetic is forbidden.
- If you work with number of cents, instead of number of dollars, you can get exact results, but you need to consider whether $a' \times b' \times c' \times d'$ will fit into a 32-bit integer, and you will need to print 173 as 1.73, that is, print 173/100, print ".", and print 173%100 (with leading zeros).

The biggest problem you are likely to find is your program taking too long. One trick in solving constraint satisfaction programs is to try to build some of the constraints into the candidate generation code so that you generate fewer candidates, which then don't need those constraints checked.

The commonest mistake in the second part is reporting totals that have more than one solution. You must seek totals that have a *unique* solution.

Testing

You want to find your mistakes before anyone else does. That means you want to test your code. It often helps to think about how you will test your code before you write it. So how can you test this program?

First, you can assume that the 7-11 problem has a unique solution. If your program finds no solution, or more than one solution, it is definitely wrong.

Second, while *finding* a solution by hand requires a fair bit of clear thinking or an intolerable amount of calculation, *checking* a solution requires a very small amount of simple arithmetic. If your program produces one answer and you verify it by hand, it is the right answer, even if your program got it by accident.

Third, when you get to the second part, you will *know* that there is at least one solution; if your generalised program doesn't report that $7.11 is a solution, it is certainly wrong.

Fourth, if your second part program finds any other answers, you can check a random selection of them by hand. What is not so easy to verify is if your program fails to report some right answers.