Twenty-Fourth Annual
Willamette University—TechStart High School Programming Contest

Saturday 20 February 2010

Contest Problems

(In the problem statements we assume that input/output will be done "console style" and we try to describe input formatting issues such as separation of parts by blank lines, etc. You are welcome to **prompt** *for the input as you need it, or to write a simple* **GUI** *interface, but you should accept the basic data as described.)*

1. Calculator conundrum

Your buddy Hewie has a calculator that only uses *postfix* notation, where an arithmetic operator (like plus or times) comes *after* its arguments. In other words, if you want to multiply 3 times 4 on Hewie's calculator, you type in 3, then 4, and *then* the multiply key. (Actually, on Hewie's calculator you hit the Enter key in between 3 and 4 so it doesn't get confused with 34: we'll just use spaces instead.) You want to help Hewie check his math, but this business of putting the operators at the end of the calculation is making it hard.

Write a program which will read in expressions in postfix form and put them back into the usual *infix* form, where the operators come *in between* the arguments. Put in one set of parentheses for every operator, *even the outermost one,* so that we can easily check your work. And just to make sure you've got it right, you should also output an *answer,* i.e., the result of actually performing the calculation described.

For example, runs of your program might look like this:

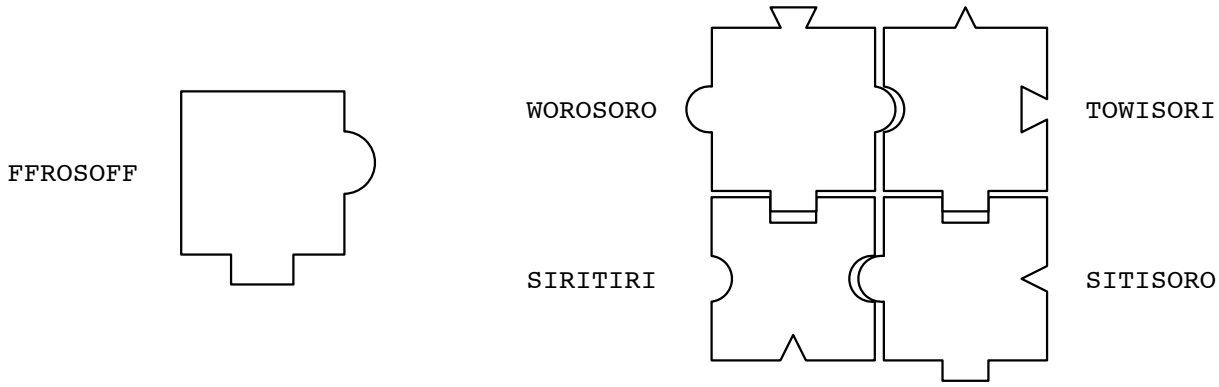
```
Input:  2 3 4 5 + * +    Output:    (2+(3*(4+5)))   =   29

Input:  1 2 + 3 4 * -    Output:    ((1+2)-(3*4))   =   -9
```

Your program should process inputs one per line until a blank line, or provide a similar GUI. We may include multi-digit numerals, but we'll always separate them by spaces. We'll only use the operators plus (`+`), times (`*`) and minus (`-`): our numerals will always be non-negative, but due to the use of minus, some of your intermediate or final results might be negative. *Our inputs will always be complete: no need to worry about "syntax errors" in the input!*

2. Put tab A into slot B …

Everyone has put together a jigsaw puzzle at one time or another — a pleasant enough diversion, but it can also be frustratingly difficult, depending on the size and complexity of the puzzle. Well, imagine how difficult it will be to write a *program* to put a puzzle together *for* you! Your program will read in a description of a puzzle as input, and then 'solve' the puzzle by printing out how the pieces can be fit together to make a completed, rectangular whole. We won't try to capture the *picture* on the puzzle in this problem, so instead we will use 'tabs' and 'slots' of different shapes on the sides of the pieces to constrain how they can fit together: tabs are the convex parts that "stick out", slots the concave ones that "stick in".

We'll describe the puzzle pieces as follows: each piece has 4 edges, which can be either `FF` for *flat* (such as on a side or corner piece), or a two-letter combination of a *shape* and a *direction.* (We use two letters for the flat edges just to make the inputs consistent: always two characters per edge.) The shapes are `R` (round), `S` (square), `T` (triangle) or `W` (wedge); and the directions are `I` (in) for a slot and `O` (out) for a tab. Each piece-description will thus consist of exactly eight upper-case ASCII letters and represents the piece in a specific 'starting' position: when you place the pieces into the puzzle, you may have to rotate them by 0, 1, 2 or 3 increments of 90 degrees (our puzzles are rectangular and aligned on the x-y axes, so no need for other increments of rotation!). Here are some sample pieces, labeled with the descriptive strings we would use for them (note also how the shapes of the tabs and slots constrain the fit in the diagram on the right):



The input to your program will consist simply of a sequence of piece-descriptions, each one an 8-letter code as in the diagram. (We will put these several-per-line, separated by spaces, or one-per-line, based on what you ask for; in any case, a blank line will signal the end of input if you need it). Your output should consist of N lines of M codes for an N-by-M puzzle solution, with the codes placed as the pieces would be. Each code will consist of a piece number (from the order they were read in, starting with 1), an '@' sign (for 'rotation') and a number from 0 to 3, telling how the numbered piece should be rotated to fit in that place.

```
Input:    SIFFFFWI  WOFFFFSO  SOTIFFFF  FFFFTOSI

Output:   2@2  4@0
          1@1  3@3
```

(You might try drawing this example out to see how it works.)

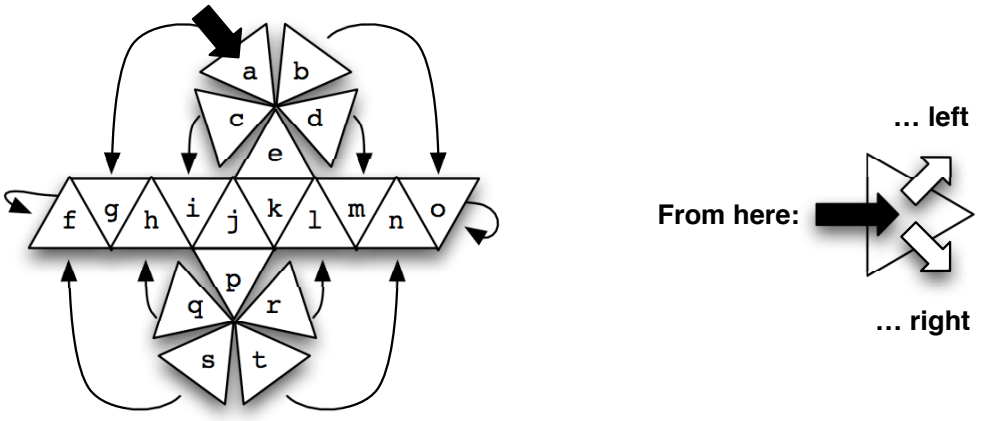
A few important things to note:

- Your solution may come out rotated from ours: that's OK, we'll allow for it in testing!
- You never need to flip a piece *over:* even though our pieces have no pictures, they are still one-sided.
- Any two pieces with the same description, or which look the same after some rotation, are basically interchangeable: we won't make any distinctions between them.
- Our puzzles, and their pieces, are always rectangular, and always at least 2-by-2: we won't give you any "thin" puzzles that are 1-by-something (not even 1-by-1!).
- We won't tell you the dimensions of the puzzle, but just give you the pieces; this could lead to an ambiguous puzzle (maybe 2-by-6, maybe 3-by-4?), but we'll accept any answer that works!

3. *Hunt the Wumpus*

Back in the day, we used to play a simple, text-based computer game called *Hunt the Wumpus.* It involved walking around from room to room looking for the Wumpus, trying to get *it* before it got *you.* (Not exactly *World of Warcraft,* was it?) One of the less well-known aspects of the game was that the rooms you explored while hunting the Wumpus were arranged in the form of an *icosahedron,* a 20-sided solid made from triangular faces or facets (you might be familiar with these solid shapes as they are used for 20-sided dice in role-playing games). Each room was thus a single triangular facet, with doorways leading to three connected rooms by way of the edges shared with the three adjacent triangles.

In order to record our travels in hunting the Wumpus, we'll letter the triangular faces from `a` to `t`, as in the map below, which shows an icosahedron "unfolded", flat on the table:



The slender arrows in the map on the left show how the edges of some of the more "squashed" parts connect; the diagram on the right shows the possible paths of a player who has entered a 'room' and then chooses between a left or right exit to an adjacent room. Notice that where the player goes next depends both on the choice of left or right and *also on the direction they are currently facing* (or, equivalently, on the direction from which they entered the current room).

Let's say that our journey starts in the triangle marked '`a`' in the map, facing toward the upper central node between rooms `a` through `e`, as indicated by the thick, black arrow on the map. We will input to your program a series of directions to travel, either '`L`' for **left** or '`R`' for **right** — our intrepid Wumpus hunters never go backwards! … *but,* they may end up going through the same room several times as they wander around the map. Your program should print out a list of the letters of rooms which our heros *never entered* during their quest (these are possible Wumpus lairs!). Notice that since we start in room '`a`', this letter will never be on the output list—we definitely visited this one room—but if the input to your program is an *empty line,* that might be the *only* room we did visit! As some more substantial examples, consider these:

```
Input:  L R L L R R L R L R L R L R R
Output: e f g h k l q s

Input:  L R R R R R R R L R
Output: f g h i l m n o p q r s t
```

Note: in order to make things easier for the judges, you should print your output letters in alphabetical order!

4. Format (and genre) wars

Your friend Wally runs a video, gaming and merchandise store called *Wallywood Video.* Unfortunately, he's hit some tough times, due to the rise of the downloadable content and internet auction sites. He's decided to streamline his business by focusing on his most successful *formats* (such as VHS, DVD, games, posters, t-shirts, etc.) and *genres* (such as detective, romance, superhero, fantasy, etc.) of the merchandise he sells.

Wally needs your help to implement this strategy: he wants you to write a program to analyze a series of sales data, including format and genre codes and dollar amounts, and then print out a report showing:

- which *format* was responsible for the most sales (with a dollar amount total);
- which *genre* was responsible for the most sales (with a dollar amount total);
- and which *combination of format and genre* were responsible, *as a unit,* for the most sales (with a total).

Each input line will include a code string for the format, one for the genre, and a sale amount (a number in dollars and cents, with an explicit dollar sign and decimal point). The code strings will always be single words, and all lowercase letters; for example: *dvd, vhs, tshirt,* or *western, scifi, detective.* (It doesn't really matter what these codes are: you just need to keep track of the totals associated with each one.) Your program should read in lines until a blank line or the end of input; it should then print out three lines of summary as detailed above.

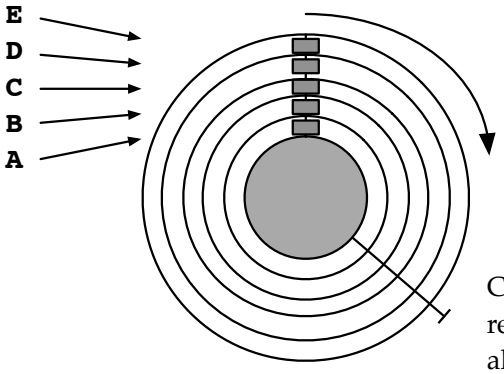
For example, consider the following input data and the corresponding output:

```
dvd         superhero   $14.99
dvd         western     $12.99
vhs         romance     $4.49
tshirt      superhero   $11.00
dvd         comedy      $10.99
vhs         romance     $3.99
poster      celebrity   $17.99

Best format: dvd at $38.97
Best genre: superhero at $25.99
Best combination: celebrity poster at $17.99
```

If you need a sentinel value to signal the end of input, we will use "`xxx xxx $0.00`" (you shouldn't be looking at stuff like that anyway!). Note that, as in the case of romance-VHS above, various formats and genres might be repeated in the input several times—every line item counts toward the totals separately.

5. Calling all cars!

Imagine that a race takes place between five cars on a circular race track with five lanes, each car in its own lane, all going the same direction, all starting at the same time. We'll refer to them as cars `A` through `E`, starting in the inner lane and working outwards. The five lanes will lie at increasing distances from the center of the track: 20, 24, 28, 32 and 36 meters, respectively, so that a car driving on the outer lane must go proportionately further in order to proceed around the track the same amount, measured as a fraction of the circumference. (The circumference of each lane is based in the usual way on its radius, as given above.) Even though it's natural to compare cars by their relative progress around the track, this puts the cars on the outer tracks at a disadvantage: this is an *intended part of the problem,* presumably to be offset by handicaps or some other mechanism.



Cars are compared for order based on their relative distance *around* the track: if they were all at this line, they would be "even".

Your program will be given successive lines of five floating point numbers, representing speed changes made by the five cars, in order from `A` through `E`; the numbers may be positive (acceleration), negative (deceleration) or zero (no speed *change,* but the car continues at its current speed). The first line of inputs will correspond to the initial starting speeds of the cars: we'll assume that the cars start and change speeds instantaneously (this isn't physically realistic, but it's easier to work with). From this point on, your program should keep track of the progress of the cars around the track and report on the relative order of the cars (1st place, 2nd place, etc.) *every 10 seconds,* printing out the time as well. *If two cars are "evenly matched" at the time of an event, you may show them in either order.* You should keep printing reports for the order of the cars every

10 seconds until you receive no more input.

For example, the following shows what the interleaved inputs and outputs might look like for a typical race (you do *not* have to print out the little side-comments: they're just there to help explain what's happening):

```
2.0   2.0   2.0   2.0   2.0          — this time, everyone starts at the same speed

Order at time 10:  A  B  C  D  E     — inner lane car has gone furthest, as a relative proportion

-1.0  -0.5   0.0   0.5   1.0         — let's even up the odds: outer cars speed up, inner slow down

Order at time 20:  A  B  C  D  E     — still the same order, despite the speed changes!

0.0   0.0   0.0   0.0   0.0

Order at time 30:  E  D  C  B  A     — the speed changes from t=10 catch up and reverse the order!

0.0   0.0   3.0  -1.0  -1.0          — car C speeds up, D & E slow down …

Order at time 40:  C  E  B  D  A     — … giving car C the edge!

                                     — etc., etc., until input stops
```

6. Digital product (ion)

Say we take all the digits of a number, individually, and multiply them together: we will get a new number as a result. We can then perform the same trick with the new number to get another one. This can go on for a few steps until we end up with a single digit. Of course, if even a single zero digit enters into the process at any point, it will rapidly bring things to a halt. *So let's agree to eliminate all zeros before continuing calculations:* the product of a number with a zero digit will therefore always by non-zero. When reporting results, you should include any zeros as part of the first number where they occur, but ignore them for later calculations.

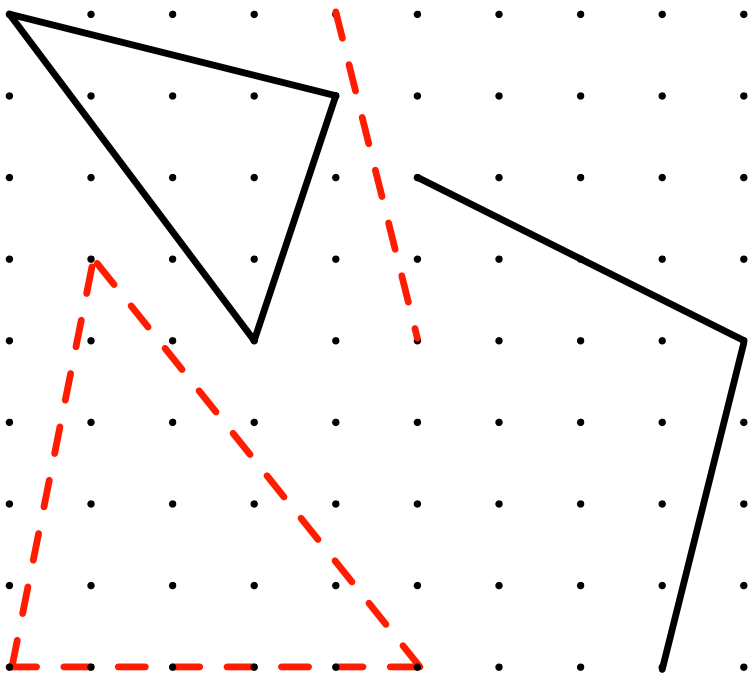
Write a program which will read in numbers, one at a time , and print out a list of all their successive digital product results, subject to the zero-elimination rule. Your output should start by "echoing" the original input number, just to verify that you read it in right. Let's agree to just end the program when a zero number is read; for example, a run of your program might look like this:

```
Input: 1984

      1984  =>  288  =>  128  =>  16  =>  6

Input: 255634

      255634  =>  3600  =>  18  =>  8

Input: 455370

      455370  =>  2100  =>  2

Input: 0

      Bye!
```

7. Playing the (tri)angles

Imagine you are implementing the following geometry game, played on a grid of points with *integer* co-ordinates (in other words, a grid of evenly-spaced dots). Two players take turns drawing a line segment from one point to another, trying to make triangles; players are not allowed to make a line which intersects or overlaps any previous line, except that either player may play a new *end-point* that is the same as an endpoint already played. The goal of the game is to end up with the *largest total area* enclosed by *triangles* made by lines you played, but only triangles count for the result, so any rectangles, hexagons, etc. are just wasted effort: they don't count in your total. The line segments are played by two alternating players, called A and B: A always starts the game, but it may end with a play by either A or B (note that you will have to keep track of the alternation yourself: we won't label the plays with A's and B's!).

Your program should read in a series of line segments for a game and determine the winner, either A or B, by listing out the *total amount of area* encompassed by each player's triangles (as a floating-point or decimal number). Inputs will consist of four *positive* integers per line; each input line represents one line *segment* played in the game, as a pair of (x,y) co-ordinates (we'll use a line of four zeros to signal the end of input, if you need it). There will be between 1 and 25 line segments, and the coordinates will always be between 1 and 100, inclusive. Your output should include the triangle-area total for each player and a determination of the winner, or a notice to the effect that it was a tie game. *Note: you do* **not** *need to test for intersecting lines and the like: all inputs will be valid games.*



By way of an example, the game board illustrated above (where (1,1) is in the lower-left-hand corner) might result from a game as shown below, including a sample output as from your program:

```
1 9    4 5
1 1    2 6
6 7   10 5
5 9    6 5
1 9    5 8
6 1    2 6
4 5    5 8
6 1    1 1
9 1   10 5
0 0    0 0

Results: B wins with total area 12.5, versus A with total area 6.5
```

8. Finding the mole

Another Winter Olympics is upon us! As usual, some events will be judged on a subjective basis and, as usual, there will be some people who suspect the judges of being biased, either in favor of athletes from their own country, or against those from another country … or both! Your company has been contracted out by Olympic security to try to discover, in an objective way, if the judging in these events has been fair.

The security team has discovered, by careful-but-secret analysis, that a judge's scoring of any given contestant should be considered suspicious if their score for that contestant is 2 or more points *higher or lower* than the average score given to that contestant by the *rest* of the judges. (In other words, each judge's score must be compared to the average of all the *other* judges, not including the one being checked.) Furthermore, if a particular judge has suspicious scores for more than *three contestants* in a match, then that judge should be flagged for further investigation. A *match* consists of a series of scores, one by each judge for each contestant in the match. Your job is to report any judge who should be flagged for further inquiry.

Your program should read in a set of scores for a single match: you will first get two integers `N` and `M` on a line, where `N` is the number of judges and `M` is the number of contestants. This will be followed by `N` lines of `M` floating-point (decimal) numbers, separated by spaces, all between `0.0` and `20.0`, inclusive. Here is a sample match with results as you would report them (you don't need to output the little side-comments):

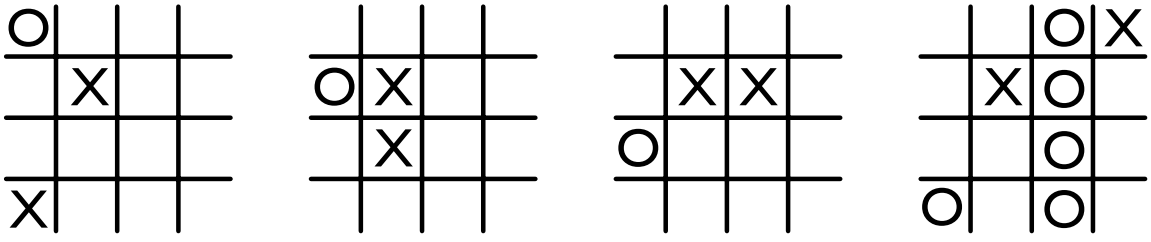
```
6 5
12.5   13.7  10.0  13.0  16.0          — judges 3 and 5 are suspicious
16.0   15.8  15.5  17.0  16.2          — nothing bad here
18.2   15.0  14.7  15.2  14.7          — judge 1 is suspicious
16.5   10.2  11.6  10.8  17.3          — everyone is suspicious! (why?)
10.0   15.2  15.9  15.7  15.5          — judge 1 is suspicious
18.5   18.3  16.1  18.2  19.0          — judge 3 is suspicious

Please investigate judges 1 and 3
```

(The 4th set of scores shows that this is not always an intuitively satisfying approach.)

(A local company, Portland-based **PTV America Inc.**, *actually* was *contracted to develop software for the Vancouver Winter Olympics, but to predict pedestrian and traffic flow; by all accounts their work was very successful!)*

9. Do we have a winner?

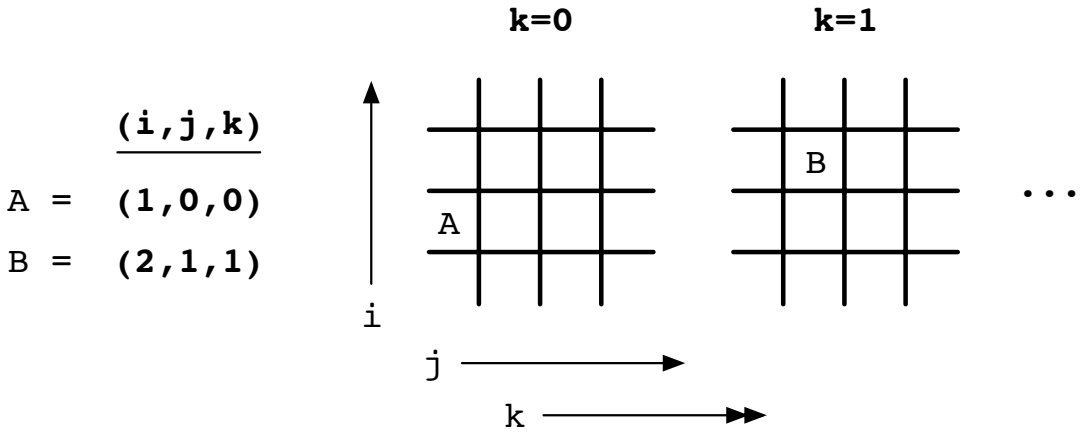
My friend Bruce and I used to get a little bored in physics class, so we would play tic-tac-toe … after we'd finished our homework, of course!. But we didn't play the usual 3-by-3 game: that would be too easy and just lead to tie games. Instead, we played a 3-dimensional version, 4-by-4-by-4 tic-tac-toe, by drawing four separate four-by-four layers on a piece of paper, in a row from left to right. The fun in this version of the game comes from the challenge of visualizing the third dimension: it's easy to miss seeing a strategy that runs through the four flat boards for a true "3D" win. Here's a picture of a typical game in progress … but this one actually shows four complete wins, and no other plays:



In order to avoid such embarrassing situations, we want you to write a program that will read in a description of a board and report any wins you find. Boards will be input as a sequence of 4 lines, each with a series of four rows. We'll write X's and O's using the corresponding ASCII letters, and unplayed positions as a period character (.); we will put one or more blank spaces between the rows of the individual boards to make it easier to enter the data. With this in mind, the game position above might be input like this:

```
o...    ....    ....    ..ox
.x..    ox..    .xx.    .xo.
....    .x..    o...    ..o.
x...    ....    ....    o.o.
```

In response to a game board input, your program should output a description of *any and all* winning sequences contained in the 3D board. (Of course in a *real* game there would be just *one* such sequence, at most: our approach just makes the problem a bit more substantial.) Each winning sequence should be reported *only once,* as a series of four co-ordinate triples, `(i,j,k)`, where each co-ordinate is a number from 0 to 3. The `i` coordinate runs vertically up from the bottom, the `j` co-ordinate runs horizontally from left to right, and the `k` co-ordinate runs from left to right across the four layers, as shown in this diagram:



Continuing the example from above, the four wins on the sample board would be described as follows:

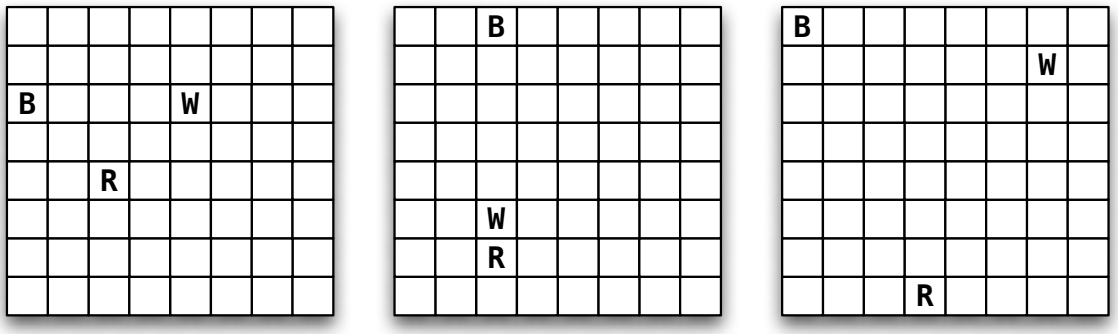
```
Win for o at: (3,0,0) (2,0,1) (1,0,2) (0,0,3)
Win for o at: (3,2,3) (2,2,3) (1,2,3) (0,2,3)
Win for x at: (2,1,0) (2,1,1) (2,1,2) (2,1,3)
Win for x at: (0,0,0) (1,1,1) (2,2,2) (3,3,3)
```

(Remember, only print each win once!)

10. We three queens of the chessboard are!

Three queens sit on a chessboard: black, white … and red? Colors aside, here is the question: do any of them "attack" the others? One chess piece attacks another if the first one could potentially move onto the second one's space, taking it in the process. Queens can move any distance, horizontally, vertically or diagonally across the board. If one queen is attacking another, clearly the reverse is also true, since they can both move the same way (we don't know whose turn it is to actually move in this tense situation). But, if one queen sits *between* the other two, along some line of attack, we will say that the middle queen *blocks* the attack between the other two. So, if the red queen is in the middle, then black and white each attack red, and red attacks both of them, but white and black do not attack each other.

Your program should read in six integers, all between 1 and 8, representing the positions of the three queens on an 8–by–8 chessboard. Assume that the positions are listed in the order Black (x,y), White (x,y) and Red (x,y). Print out all attacks (if any) between queens (e.g., "Red attacks Black"), except for any "blocked" attacks as described above. For example, here are three typical chessboards and the corresponding runs:



`1 6  5 6  3 4` => *(list out all six combinations of attacks)*

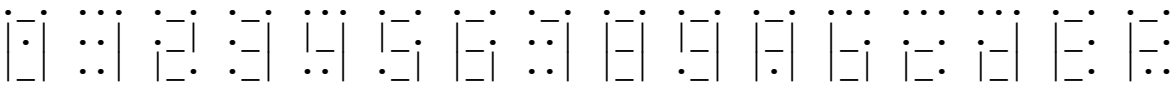
`3 8  3 3  3 2` => Black attacks White; White attacks Black; White attacks Red; Red attacks White

`1 8  7 7  4 1` => No attacks!

11. Seven segments, writ large

Despite the growing popularity of OLED displays and other pixel-addressable technologies, many consumer devices still use a traditional *seven-segment display* to show numbers. Such displays can even be used to display hexadecimal digits, in base 16, using the letters `A` through `F` for the digits larger than `9`, and displaying them as shown here (note the lowercase forms for `b`, `c` and `d`):

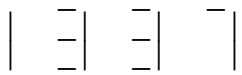
It turns out that we can write out these segmented displays pretty accurately using ASCII characters (especially in a mono-spaced font like Courier) using just spaces, underscores and the 'pipe' character (`|`), which is usually located near the upper-right of the keyboard, above the backslash. It might be a little easier to see how this works if we replace the blank spaces (representing turned-off segments) with periods instead:



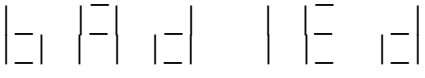
Note that each hex digit is thus displayed as a 3-by-3 arrangement of just blanks, underscores and pipes; *note also that we use spaces in between the digit to make the output easier to read* (you should do this, too!).

Your program should repeatedly read in numbers in *decimal* and print out a seven-segment display, in ASCII, of the corresponding *hexadecimal* numeral (you may request a "sentinel value" of -1 to end the input if you need it; and please use a monospaced font for output, if possible). Here are some sample inputs and outputs (the hex numerals on the right are `1337` and `bAd1Ed`):

```
Input: 4919          Output:
```



```
Input: 12243437      Output:
```



(This problem inspired by one from George Struble, Contest Founder and Director Emeritus, from the 5th Annual.)

12. Base-pair matching I

The Portland CSI (Computer Science Investigators) have hired you to write software to help them match DNA sequences found at crime scenes. As you may know, DNA comes in the form of a twisted double helix, composed of four molecules (called *nucleotides)* that must match up in certain *base-pairings* across from each other on the two strands of the helix. The four nucleotides are traditionally abbreviated `C`, `G`, `A` and `T`; in a proper DNA strand, `A` is always paired with `T`, and `C` with `G`, as in this example taken from Wikipedia:

```
ATCGATTGAGCTCTAGCG
TAGCTAACTCGAGATCGC
```

(Note that any of the four nucleotides may appear on either of the strands: the only restriction is that they appear across from the corresponding half of their base pair.) The CSI team has samples which are small snippets from one strand of DNA which they need to match against another, typically longer segment of the opposite strand. The proper direction of the strands are not known, so *the smaller snippet might appear running either left-to-right or right-to-left: take this into account!* Also, the CSI's need to know of *any* match that shows up, even overlapping ones, as this might be crucial for later stages of testing.

Your program should read in two strings, one after the other, both non-empty sequences of ASCII letters from the four `ACGT`. The first string will be the pattern you are looking for; the second will be the segment in which you search. You should report *all* matches of the first string whose base-pair *opposite* occurs within the second string, in either direction. Each match should be reported as a pair of integer indices between 0 and *n-1*, where *n* is the length of the second input: this indicates a match between the two indices *including both ends.* When reporting reversed matches, *the left index will be greater than the right*. For example, if you received the following inputs you would report matches as below:

```
CATCAT
CCGTAGTAGTATACGTAGTCGTAGTATGATG

Matches found at: 2-7, 5-10, 20-25, 30-25
```

(Remember, indices start at position 0 on the left!)

13. Base-pair matching II

The CSI lab up in Portland was so pleased with your last bit of work for them that they want you to tackle a new job. In this one, they are looking for *partial matches* of DNA strand snippets: they will use special characters in the pattern to specify single-letter wild cards or longer, substring wild cards. Let's use the asterisk (`*`) to mean "any single nucleotide may match here" (but exactly one) and the underscore (`_`) to mean "any sequence of nucleotides may match here" (in particular, the sequence might even be *empty).* In most other respects, the matching will be the same as for the last problem (opposite letters matched, left-or-right directions possible), except for one: we will now only require the *longest* possible match, if any, so you need to report only this one result to the input.

By way of example, here are what two runs of your program might look like:

```
AA**TC_GA

ATGTTGAAGATACTTGGAGCTAAG

Longest match found at: 3-20
```

```
C*G_T

ATGCCATACCTGTC

Longest match found at: 11-0
```

14. The check's (crossed) in the mail

Once upon a time, I lived in a house with three housemates, and each of us was responsible for some of the monthly bills (gas, electricity, water, newspaper, etc.). At the end of the semester, we would get together and figure out what we owed each other for the bills we had paid on each other's behalf. Everybody was responsible for an equal share of all of the bills, so in some sense each of us owed all of the others for a one-quarter share of the bills they had paid. One way to resolve this (but not a very good way!) would be to write a whole bunch of small checks representing these one-quarter shares of all the bills. This approach would clearly be wasteful, since two people would write checks to each other, when in fact a single check would do: *the person owing the larger amount could simply write a check for the difference between the two amounts.* Of course, if A owes B, and B owes C, depending on the relative amounts, we might be able to reduce the number of checks involved even further by having A pay C directly, thus "skipping the middle man" (B).

In fact, if there are N people in the house, we can *always* make do with N-1 checks, or possibly even fewer (if some of the amounts cancel exactly to zero, for example). You should think about the problem enough to convince yourself that this is true *before* you try to write a program!

Once you have determined a good strategy, write a program which will read in a series of N amounts, as floating-point numbers (if you need the integer N as well, just prompt for it). Let's call the four people by the names A, B, C and D: each amount represents the total of the bills which the corresponding person paid on behalf of the entire household, for persons A through D, in order. Your program should then print out N–1 (or fewer) lines of this form:

```
$25.20 from A to B
```

This sample line means *"a check for $25.20 should be written from person 1 to person 3"*. Your output should ensure that all people in the house are properly compensated for the bills they paid, while keeping the number of checks down to a minimum. In particular, if the problem can be solved with fewer than N-1 checks, your output should reflect this fact—in the "worst case" (or is it the best?), no checks would be required at all (for example, if all the pay-outs happened to be for the exact same amounts). Note that a person does not need to be compensated for the *whole* amount they paid out, but rather for *all but their own share* of the bills they paid.

As an example of input and output, consider this situation, where A has paid 20 dollars in household bills, B has paid 40 dollars, etc.:

```
20.0  40.0  60.0  80.0

$30.00 from A to D
$10.00 from B to C
```

You should be able to check that each person will have been reimbursed the proper amount … and we only needed two checks!

(Don't worry about floating-point rounding errors: we will allow for minor variations in the check amounts.)