Twenty-Eighth Annual
Willamette University—TAO Foundation High School Programming Contest

Saturday 15 March 2014
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. Word centipedes

Let's call a sentence a *word centipede* if the last letter of each word is the same as the first letter of the next (we won't bother with the first letter of the first word, or the last letter of the last word). Write a program which will read in lines of text, repeatedly, until a blank line is reached (or use an equivalent GUI that allows lines of text to be read when a button is pushed). For each line read in, tell whether or not it constitutes a word centipede—if it does *not* qualify, *report which words* cause it to be disqualified (remember to include *every* pair that "breaks the combo"). We will only use lowercase letters, with no punctuation, and words will be separated by some number of spaces (usually one). Candidate sentences will always include at least two words—and you do *not* need to check that they are actually valid English sentences!

For example, your program might look like this when run on a few lines of input as given:

```
> ten nimble ents stand dourly yet truly yonder

  This is a correct word centipede!

> gerbils surely yammer relentlessly

  This is a correct word centipede!

> the errant nimble fox jumps swiftly

  This is NOT a correct word centipede, due to the following pairs:

    errant/nimble;  nimble/fox;  fox/jumps
```

2. Number buddies

A number (let's stick with positive integers here) can be written out in different *bases,* such as the base 10 format we use in day-to-day life, or the base 2 format (also called binary) used internally in computers, or base 16, used by many programmers as a concise version of base 2. When we write a number in base *k,* we use digits between 0 and *k–1,* inclusive, to represent the number, writing the digits from left to right in decreasing order of significance. So, for example, 1027 in base 10 represents … well, 1027, of course! But 101011 in base 2 represents 43, since we have 1s in the 32, 8, 2 and 1 places. In bases higher than 10, the convention is to use letters of the alphabet for digits larger than 9, starting from `A` (which is therefore the "10 digit"), running through `F` (the "15 digit"), and all the way up to `Z`, which can be used as a "35 digit". Let's agree to stop there—we'll only use bases between 2 and 36 inclusive for this problem.

Now when we write out two numbers in some specific base, it can happen that they use exactly the same *sets* of digits: for example, the numbers 12 and 1222 both use only the digits 1 and 2 in base 10. Let's call two such numbers *buddy numbers* in that base. You should write a program which repeatedly reads in two numbers, *in the familiar base 10 format,* along with a third number, the base. You should then report whether or not those two numbers are buddies *in that base.* (You may assume the numbers are all written on a line, separated by spaces, or you may use a GUI.) For example, a run of your program might look like this:

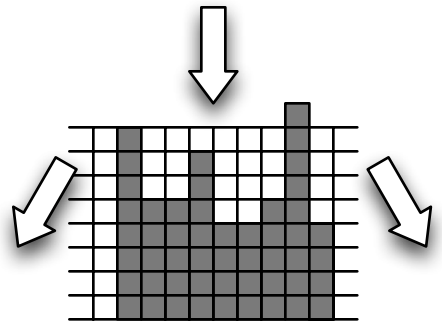
2. Number buddies ***(continued)***

```
> 12  1222  10

  12 and 1222 are number buddies in base 10 (12 and 1222).

> 31  7  2

  31 and 7 are number buddies in base 2 (11111 and 111).

> 897  96  5

  897 and 96 are NOT number buddies in base 5 (12042 and 341).

> 57167  66957  17

  57167 and 66957 are number buddies in base 17 (badd and dabb).
```

Note that you should echo the numbers in base 10, but also report them in the format of the given base, just as a check on your work. We will *never count leading zeros:* in other words, zero only counts as a digit being used if it appears *after* the first non-zero digit in a given base format. Finally, note that some numerals might consist entirely of letters, as in the last example above (the one using base 17).

3. The Twitter™ water-flow problem

Consider a sequence of numbers, specifically positive integers, and imagine that we use them to create a little bar chart, as we might do with a spreadsheet program. Now imagine pushing the bars together horizontally, so that they sit flush against each other—the bars form a sort of 2D terrain structure, seen in profile, with no gaps, as in this diagram:



Water flow: "cup" and sides

This being Oregon, if we leave our little structure outside, it's going to get rained on! And when it rains, some parts will fill up with water, specifically the parts that form "cups", or concave portions of the profile. Let's agree that any water that falls near the left or right extremes, but outside a "cup", will just slide off and flow away. (In the picture above, the middle arrow shows where water will "pool up" and stay, whereas the left and right arrows show areas where the water just flows off to the sides.)

We're interested in how *much* water stays behind, caught in those little "cups". Being in Oregon, we'll assume that as much rain as necessary falls to fill the cup areas all the way to the top. If we think of the numbers in the original sequence as representing some sort of units, then we can measure the water in terms of *square* units, assuming the "bars" are each one unit wide. For example, in the diagram above, we will accumulate a total of 18 square units of water overall, after a heavy rain.

Your program should repeatedly read in a sequence of positive integers (possibly on several lines; use a "-1" to signal termination, if necessary); for each line, it should report the total amount of water that will collect in the corresponding structure.

3. The Twitter™ water-flow problem *(continued)*

As a sample input, the picture above would be specified as follows, and would yield 18 units collected:

```
8 5 5 7 4 4 5 9 4

      18 units of water will be collected
```

(This problem was reportedly used by the Twitter company in their programmer job interviews, as a way to see how candidates thought through a problem; I found it being discussed on several coding blogs.)

4. Cipher seizure

The fine people at the NSA (that's the *National Snooping Agency)* need your help in deciphering some coded messages they intercepted, sent by two suspected terrorist organizations, the EFF and the ACLU. We know that the messages were coded with a so-called *caesar cipher,* in which the letters of the alphabet are shifted by a fixed amount, with the end of the alphabet "rotating" back through to the front. For example, if the shift amount is 5, then the letters "`ABC`" would become "`FGH`" and the letters "`XYZ`" would become "`CDE`". In order to *decode* a message with a known shift factor, you merely shift or rotate the code letters *backward* by the same amount.

Unfortunately, all the NSA agents have is the coded message and a certain *keyword* which they think the message contains. Your program should look for the keyword among all possible decodings (reverse shiftings) of the coded message, showing the corresponding decoded message whenever it finds a match. For example, if the keyword is "`HAL`", and the coded message contains the word "`IBM`", then a shift factor of 1 is a possibility, so you would print out the whole message shifted backward by 1.

However, note that you might find the keyword in several different versions of the decoded message, corresponding to different shift amounts. For example, if the keyword was "`OH`" and the coded message had both "`MF`" and "`AT`" in it, then the correct shift amount might be either 24 or 12, since "`OH`" shifted by 24 gives "`MF`", but "`OH`" shifted by 12 gives "`AT`". If you find different possible decodings of this kind, you should print all of them out: the NSA agents will then manually inspect the decoded messages to determine which one is likely to be the intended one. If the keyword *cannot* be matched in the message, say so. *Note that a zero shift factor is possible: some of these terrorists are not so bright!* All inputs will be words in uppercase separated by spaces, with no punctuation.

For example, if the input is:

```
coded message: AT FTQ DMHQZ RXUQE MF PMIZ
keyword: OH
```

your program should report something like this:

```
if message was shifted by 24, decoding = CV HVS FOJSB TZWSG OH ROKB
if message was shifted by 12, decoding = OH THE RAVEN FLIES AT DAWN
```

5. Compare the sides of starch! *[Et tu, Brute?]*

Fad diets are all the rage these days … but then, aren't they always? You have been hired by a new start-up dieting company to help them cull out candidate meal plans they think will help their clients take off the pounds. The company's nutrition experts have come up with a set of dietary components (e.g., protein, fiber, fat, starch) and, for each component, a range of preferred proportions that component should take up in a given meal. Your job is to read in these ranges, as well as summaries of meal contents, and then report whether each meal fits the criteria or not.

Your input will come in two stages (use a blank line to separate them, or just use a GUI with suitable buttons): in the first stage, you will see three items per line: the first item will be a word (lowercase, no

spaces), representing a name for the component (e.g., "starch"). The next two will be floating point numbers (i.e., they will have decimal points) representing the lower and upper limits of proportion recommended for this component (e.g., `0.10 0.15`, representing "between 10% and 15%" *inclusive*). After reading in these limits, you will then read in lines which contain meal descriptions: each line will consist of alternating pairs, a component name followed by the amount of that component, expressed in grams. If *all of the components* of the meal fit the guidelines given in the first stage, report it; otherwise report that they do not, and specify which components (*all* of them, by name) are out of range.

For example, you might see the following input and provide the responses below:

```
protein   0.20   0.35
fat       0.05   0.20
sugar     0.10   0.20
starch    0.20   0.40
      (blank line here)
sugar 4    protein 7    starch 10    fat 4

      This meal meets the guidelines!

protein 5    sugar 6    fat 1    starch 8

      This meal DOES NOT meet guidelines due to: sugar

fat 12    protein 1    starch 2    sugar 10

      This meal DOES NOT meet guidelines due to: fat, protein, sugar
```

Note that the order of components in the meals may differ from the order in the guidelines (that's why we use the names). And don't worry about whether the percentages add up to 100: there may be other, unspecified components to meals (spices, water, etc.) whose presence does not affect the overall calculation.

6. Time conversion

Your friend has joined the military and is "shipping out" to far parts of the globe. You'd like to stay in touch, but military rules require that all transmissions pass through a security check, which has the unfortunate consequence of significant delays in communication. You're having trouble figuring out when his messages were actually sent, given that there's a time difference and given that the military-approved system only includes time in "military form" (e.g., "oh-eight hundred hours").

Write a program to help you convert the times on your friend's messages into your own local time: his times will be expressed using 24-hour military standards and will be 8 hours later than local Oregon time. For example, when he mails you at 6:00pm his time, it will come to you as "18:00", but would be equivalent to "10:00 am" Oregon time.

Your program should read in lines containing a time formatted to military standards (two digits no greater than 23, followed by a colon, followed by two digits no greater than 59). You should print out an equivalent for each input time, time adjusted to the Oregon locale, and formatted according to civilian standards (two digits no greater than 12, followed by a colon, followed by two digits no greater than 59, followed by either "am" or "pm"). Assume that by military standards, "00:00" is midnight (not "24:00") and that "12:00" is noon; by civilian standards, "12:00 am" is midnight and "12:00 pm" is noon.

7. Collusion detection

The *International Skating Union* has been put in an uproar by recent Olympic events involving indiscretions on the parts of skating judges. In particular, there have been allegations that certain pairs of judges have been operating in concert to "fix" the scoring of the competitions by trading votes: the judges in collusion will artificially boost votes for skaters from their own (and each other's) countries, while artificially voting down skaters from the other countries.

7. Collusion detection *(continued)*

The ISU has contracted you to write a program to detect possible collusion among judges: the idea is that any two judges who are in "cahoots" will have scores which are consistently *similar* to each other and also consistently *different* from other judges in the competition. Your program will examine scores by six judges for a series of competitors, looking for any two judges who consistently have scores within 0.5 points of each other, but whose average score on *a per-skater basis* consistently differs from the average of the remaining four judges by 0.5 or more. *Note: there may be several potential "dirty" pairs of judges;* the ISU will engage in further investigation based on your program's alerts.

Your program should read a series of lines, each with six positive floating point numbers representing the scores of the six judges. The end of the input will be signaled by a line containing six 0.0 values (these don't count as scores). You should print out alerts indicating all pairs of judges who might be in collusion, using their ordinal numbers to identify the judges (1, 2, 3, 4, 5, or 6) . Of course, you would never report a judge and him- or her-self as a "pair"; furthermore, you should output a given pair only once, in order of increasing judge number (i.e., report "1 and 3" but not "3 and 1").

By way of example, consider the following inputs (not including the sentinel ending values):

```
5.3   5.9   5.4   3.8   5.8   4.0   5.7
4.6   3.9   4.6   5.8   4.0   6.0   4.7
5.0   6.0   4.8   3.5   6.0   3.6   4.7
```

You would report *both* of the following:

```
possible collusion between judges 2 and 5
possible collusion between judges 4 and 6
```

(Note that there are other potential collusions in the first set of scores, but these suspicions are not borne out by the other two sets.)

8. Cross country, round trip

It's time to set up the cross country ski course at the Olympics in Sochi! The course is organized as a series of "legs", each one a straight line, which are connected one after the other until they return to the starting point ... or so we hope! You should write a program which will check whether a series of course legs really *does* return to the starting point, and also compute the total length of the course. The input will be in the form of a series of lines, each containing pairs of integer numbers (possibly negative, and perhaps even 0). Each pair of numbers represents the x and y distance of the next point in the course, starting where the last leg left off. Your program should print a message indicating whether or not the course is a proper round trip (i.e., whether it returns back to its starting point), along with a floating-point number representing the overall distance covered. For example, here is a round trip starting at the lower two of the rightmost points shown in the picture:

```
Input:  0 3   -5 2   -4 -6   5 2   0 -4   4 3

Output: Good round trip! Length = 29.98
```

9. Cross country, round trip … no crossing!

Well, the cross-country skiers in the Olympic games certainly are a fussy bunch! After some initial dry runs, they have complained that whenever one leg of the course crosses over another leg, the snow gets all mushy and the trail gets messed up. (Well-groomed trails apparently lead to better race times.)

To accommodate the skiers' needs, you should write a new, advanced version of the program from the last problem: your program will read in the same sort of data, but *in addition* to determining whether the course constitutes a round trip and reporting the length, it should also print out *one warning for every pair of legs* that cross each other (refer to the legs by number, starting with 1; *don't print duplicate warnings* for the same pair of legs in the opposite order). For example, the following course (which starts at the lower right corner) is a good round trip, but legs 2 and 4 cross each other:

```
Input:  0 4  -6 0  3 -2  3 4  -7 0  -2 -4  9 -2
Output: Good round trip. Length = 39.30 Warning: legs 2 and 4 cross!
```

Note that the line segments representing the legs *must actually cross:* in some cases, the infinite lines corresponding to some legs will cross, even though the finite segments never do (for example, legs 3 and 6 in the picture above). To avoid ambiguities, you may assume that only adjacent legs will intersect at their endpoints, and that legs will never overlap each other (i.e., one literally on top of the other).

10. It's in the cards

In a certain variation on the game of poker, called *Oregon Hold 'Em,* a player is dealt five cards for their initial hand. Then, after a round of bets are placed, they are offered the chance to *exchange* some of the cards in their hand for new cards. In this particular style, the player is shown two new cards, and they must decide whether to exchange any cards (and which ones) from their original hand with an equal number of the new cards (so in the end they will have exactly five cards again, whether or not they choose to exchange). The idea, of course, is to pick cards to maximize one's chance of winning when all players' hands are compared.

So how are players' hands compared? Oregon Hold 'Em has only three kinds of winning hands:

- *straights:* five cards of any suit, in a row by *value;* aces are "high", so that values run from 2 up through 10, then jacks, queens, kings, and aces.
- *flushes:* five cards of any value, but all of the same *suit* (the suits are clubs, diamonds, hearts, and spades, in increasing order);
- *straight flushes:* combining both of the previous ideas to give five cards of the same suit *in a row* by value;

The relative strength of two hands is determined as follows: a straight flush always beats a "plain" flush and a "plain" flush always beats a "plain" straight. Two straights are compared by preferring the one with the highest value card or, if those are the same, by the best suit of that high card. Two flushes are compared according to which suit is best or, if they are the same suit, by the highest card in each. And two straight flushes are compared likewise, first according to suit and then by high card if necessary.

We will write cards using a combination of two characters, the first for the value (digits `2`-`9`, `X` for ten, `J` for jack, `Q` for queen, `K` for king and `A` for ace) and a suit (`C`, `D`, `H` and `S`). Your program should read in 5 cards

10. It's in the cards ***(continued)***

on one line and two on the next (or use separate fields in a GUI, but clearly labeled) and print out a final hand, after replacing up to two cards from the second set to enhance the first. You should also report the best type of hand you have (if any) after the exchange. For example:

```
XS  QH  9S  JC  7S
6S  8S

     Straight flush with: 6S 7S 8S 9S XS

3C  JD  QH  4S  7D
2S  KD

     No good hand found!

6C  2C  5C  7D  3C
AC  4D

     Flush with: 6C 2C 5C AC 3C
```

(Note in particular that in the last example above, cards were chosen to get the flush, not the possible straight.)

11. Doubled letters

Some words have two of the same letter repeated, one right after another, as in "look" and "apple". Longer examples may include several repeated pairs, as in "committee" and "bookkeeper". These repeated pairs can be either vowels or consonants, and both may occur in the same word, as in the last two examples. Let's call a word *lucky* if it has a repeated pair of vowels and if it has *no* repeated pair of consonants (only count 'a', 'e', 'i', 'o' or 'u' as vowels; all other letters are considered consonants, including 'y'). Three (or four, etc.) letters in a row also count—we don't know any English words like this, but maybe the input is not in English (fortunately, you don't need to check!).

Write a program which will read in a line of text and break it into words (there will be at least one space between words, and there will only be letters and spaces). Your program should print out all the lucky words on one line (if the same word is repeated in the input, it's OK to repeat it in the output). For example, on this input:

```
a big apple looks to a bee unlike it seems to a committee of bookkeepers
```

your program should print:

```
looks bee seems
```

12. Tomographic asteroid traversal

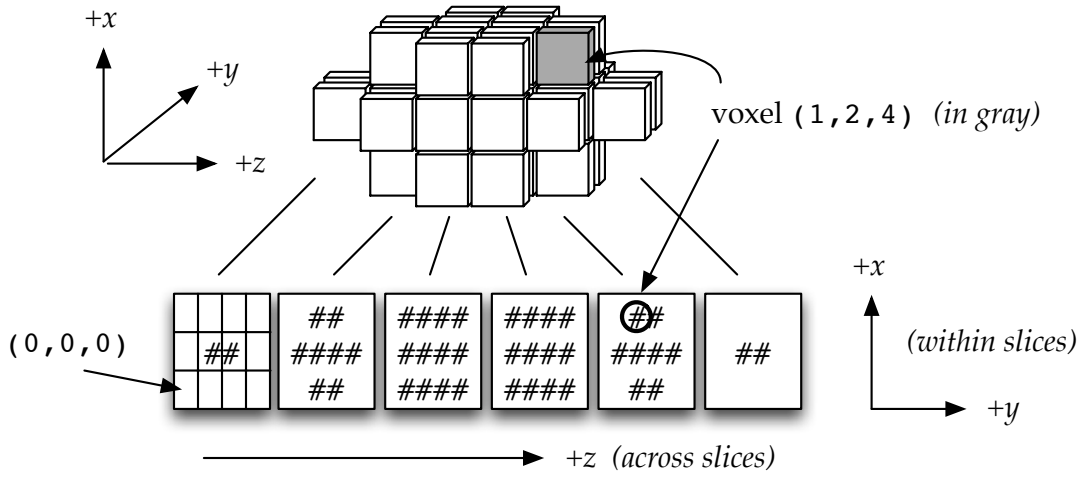
Imagine, if you will, an asteroid floating out in space: a hunk of rock and metal, roughly potato-shaped, but with craters and perhaps even caves dug out of its surface, after years of collisions with other space debris. NASA is sending a little rover up to investigate several asteroids, and you've been hired to help program its maneuvers. (The asteroids contain enough iron that the rover can "stick" to the surface magnetically—that way it can move about without danger of falling off, even though there is very little gravity.)

The "mother ship" that carts the rover around the asteroid belt is equipped with high-energy scanners, similar to the X-ray and CAT-scan devices used on Earth for medical studies. Specifically, the scanners will provide a description of the 3D shape of the asteroid, with its craters and caves, as a series of *tomographic slices:* each "slice" is in the form of a discrete (x,y)-grid, with the individual slices providing for a z-axis. The output of the scanner shows only the *presence* or *absence* of material, in cubic units called *voxels*—these are like

12. Tomographic asteroid traversal *(continued)*

the pixels of a screen image, but 3-dimensional. This representation is only accurate up to a point: like a screen image, it suffers from "jaggies" where the resolution is too low to accurately show curves. But it will have to suffice for our mission!

We will provide your program with an overall description of the dimensions of the tomographic data as three positive integers: the first two numbers represent the size of each slice, in terms of columns (x dimension) and rows (y dimension); the last is the number of slices (the z dimension). We will then provide the slices as lines of data with blanks and "hash marks" (#): the blanks represent empty space and the hash marks represent asteroid rock & metal. By convention, the lower-left corner of the first slice is coordinate (0,0,0), with the coordinates increasing to the right (x), upwards (y), and with each slice (z). In this way, we can refer to any voxel with three non-negative coordinates.



3D asteroid voxels and corresponding tomographic slices

In the diagram above, we have provided both a 3D representation of an asteroid and a sketch of how it would be described as rows of character data. The slices in the lower part of the picture correspond to vertical slices of the asteroid, running from left to right. We also have marked a particular voxel in gray, located at position (1,2,4), and have highlighted the corresponding hash mark in the input layout. *(Actual input data will provide the slices one after another as successive lines of input, not running across the page.)* Finally, we will describe a particular *face* of a voxel by specifying that it is the positive or negative face along a given axis; for example, the highlighted voxel has three exposed faces corresponding to positive x ($+x$), negative y ($-y$) and positive z ($+z$) directions.

Now that we have the basic ideas and coordinate system set, it's time to run our rover through its paces! At any given time, the rover's current state can be described in terms of the *voxel* it is on (given by 3-part coordinates); the exposed *face* of that voxel on which it currently rests (given as plus-or-minus and an axis letter; e.g., $+x$); and the *direction* that it is pointing (also a plus-or-minus axis specification). The rover will always be aligned along one of the axes (never 30 degrees or 45 degrees off, for example).

We will provide such a description for the rover in its initial position as it begins its journey. Due to limitations in the design, the rover can only move forward, or else "twist" around in place, to the left, to the right, or completely around ("about face"). When the rover moves forward, there are 3 possibilities for what it will encounter: flat terrain (the next voxel is "flush" with the current one); a "wall" (i.e., a solid voxel directly in its path); or a "cliff" (i.e., no material voxel in the next coordinate position, so that the floor "falls away"). Here is where the rover's magnetic hold comes into play: if it moves forward onto flat terrain, it will simply move to the same face of the next voxel. If it encounters a wall, it will turn upward to "climb" the wall. And if it encounters a "cliff", it will turn down the cliff face as the floor falls away. In this fashion the rover will always make progress when moving forward and never get "stuck".

12. Tomographic asteroid traversal *(continued)*

Once you have read the basic description of the asteroid and the initial position of the rover, we will provide a sequence of commands, as a list of characters from among `F` (forward one unit), `L` (left turn), `R` (right turn) or `A` (about face; all turns are *in place).* Your program's job is to simulate this sequence of commands, tracking the current state of the rover as it moves, and then report back its final state when the command sequence is done. Your description of the final state will use the same conventions as the initial placement: a voxel (3 coordinates), an exposed face (plus-or-minus along some axis), and a direction it is facing (again, plus-or-minus along some axis).

Here is an example asteroid description (corresponding to the diagram above) along with initial positioning for the rover, a sequence of commands, and the final outcome state of the rover:

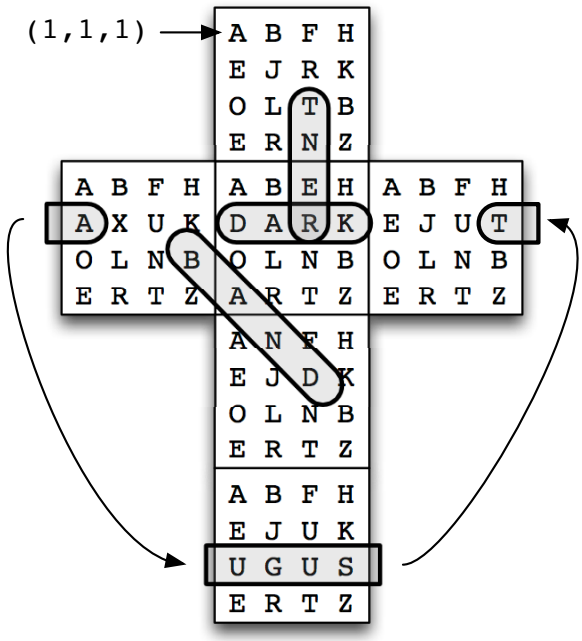
```
4 3 6                    (the dimensions of the asteroid voxel grid)
____
_##_                     (the voxel data, with blanks shown as underscores for emphasis)
____
_##_
####
_##_
...                      (12 more lines of voxel data)
1 2 4 +x -z              (initial voxel, face and direction)
F L F F F R F F F        (command sequence)

     Rover ends up at: 0 1 1 -y -z        (final state output)
```

13. Word search—the 3D edition

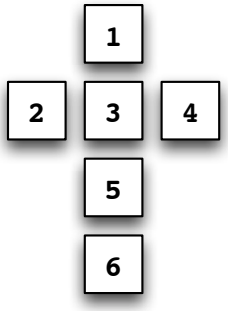
You've probably seen word search puzzles before: a grid of alphabet letters, plus a list of target words to find in the grid. These puzzles are a common sight on restaurant menus for kids, newspaper puzzle pages … in fact, you can even find one in last year's contest problems! In the trickier versions of the game, the words can run through the grid in any direction: horizontal, vertical, or diagonal, sometimes even "backwards" (that is, right-to-left when horizontal, or bottom-to-top when vertical, etc.).

For this year's problem, we'll allow words to run in *any* of these directions, but always "in a straight line". Well, a straight line that sometimes goes around corners! Our word search will be done on *the faces of a cube! (*In the sample diagram below, we've shown a cube in a form that suggests how it might be "folded up" to make a 3D shape—some of the letters would come out upside down, but that doesn't really matter to the problem). Your program should read first read in a number, the common length of the sides of the cube in each of the three dimensions (in the sample this would be 4). It should then read in rows of letters, according to the "order of cube faces" pattern shown below. (So, for this 4-by-4-by-4 cube, you would read in 24 rows of four letters each, as 6 sets of four lines, each set corresponding to a different cube face. The data for faces 2, 3, and 4 will come *separately, not* on the same lines as suggested by the diagram.)

Following the input of the cube letter data, your program should read in single target words, one at a time, and report for each word *where* it can be found on the cube (we will make sure that all our target words actually *are* on the cube somewhere, but in only one place per word). The letters may run in any direction, even if they continue around a corner onto another face of the cube, as long as they for a "straight" line.



A letter cube "unfolded"



Order of cube faces

For example, in the diagram above, we can see the following words highlighted: `RENT`, `DARK`, `BAND`, and `AUGUST`. Note that `BAND` runs around a couple of corners, and that `AUGUST` runs across three faces. (And note especially how the rows match up *once the cube is "folded" up:* row 2 on face 2 matches row 3 on face 6!*)*

Let's agree to the following coordinate system for reporting letter positions: we'll specify any given position as three numbers, writing the face number first (as shown above), then the row number (starting with row 1), and then the column number (starting with column 1), so that position (1,1,1) is the *upper-left corner* of the first face. *Note that we will use the layout as specified in the input, even though the rows and columns may seem to mis-match for words like* `AUGUST` *which wrap around folded-back faces!*

Using this coordinate system, we can report the coordinates of any word by listing the coordinates of its letters, in order. For example, for the cube shown in the diagram above we would have:

- `RENT = (3,2,3) (3,1,3) (1,4,3) (1,3,3)`
- `DARK = (3,2,1) (3,2,2) (3,2,3) (3,2,4)`
- `BAND = (2,3,4) (3,4,1) (5,1,2) (5,2,3)`
- `AUGUST = (2,2,1) (6,3,1) (6,3,2) (6,3,3) (6,3,4) (4,2,4)`

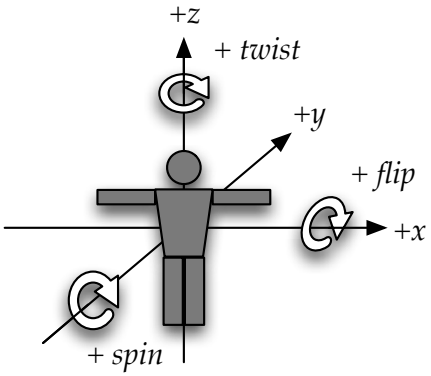
Your program should therefore read the cube size, the rows of cube letters, and then target words, on by one, reporting where they are found on the cube as above.

14. Slope-style flips, spins, and twists

It's winter Olympics time again, and for this problem we'll be going to the slope-style snowboarding events. If you've seen this event reported on TV, you know that the announcers describe the "tricks" that the athletes perform largely in terms of the various kinds of rotations they do as they flip, spin and twist their way through the air. More specifically, announcers call out the rotations in terms of degrees, so for example a "1080" is three full turns (1080 = 3 × 360), but a "540" is only one-and-a-half turns (540 = 1.5 × 360). We're going to simulate these snowboarding moves in order to determine how a boarder will be oriented at the end of a series of tricks. *Note: all the rotations we use in this problem will be in whole increments of 90 degrees!*

Our snowboarders will start out facing forward and "standing" straight up and down. Every time they perform a rotation, it can happen in one of three dimensions: we'll refer to these dimensions in terms of an *x* axis (running from left to right), a *y* axis (running from back to front), and a *z* axis (running from bottom to top)—see the diagram below! Furthermore, we'll refer to the *directions* (positive or negative) of a rotation so that the positive direction is the *clockwise* one when viewed from the *right,* the *back,* or the *top*, respectively (again, see the diagram below). I'm not sure of the actual jargon that the announcers use for rotations around these various axes, but let's agree that:

- a *flip* will be a rotation around the *x* axis (so a positive flip starts as if falling forward);
- a *spin* will be a rotation around the *y* axis (so a positive spin starts as if *leaning* to the right);
- a *twist* will be a rotation around the *z* axis (so a positive twist starts as if *turning* to the right).



Dimensions and rotation directions
(snowboarder seen from behind)

Now here comes the tricky parts: in actual practice, boarders rotate in these various dimensions *simultaneously,* but since that is difficult to describe and compute, we will stick with *sequential rotations.* That is, our boarders will always do a complete rotation (of however many degrees) in one dimension, *then* do another rotation in another dimension, etc. *When done sequentially, the order of the rotations actually matters for the final result!* One more complication: it seems most natural to describe the rotations themselves relative to the current orientation of the snowboarder, rather than in terms of the global coordinate system, i.e., from an "internal perspective" as opposed to an external one. (That way, when a boarder flips and then twists, the twist is still around the axis that runs from their toes to their head, even though most flips will leave them pointing along a different axis than before.) For example, a +90-degree flip followed by a +90-degree twist will leave the boarder with the top of their head pointing along the +*y* axis and their extended right arm pointing along the +*z* axis. If they did the rotations in the other order, a +90-degree twist and *then* a +90-degree flip, they would end up with the top of their head pointing along +*x* and their extended right arm pointing along -*y*.

Notice in the preceding example that the direction that the boarder's head points (which starts at +*z*) and the direction their extended right arms points (which starts at +*x*) are together sufficient to completely describe their orientation: we will therefore use these two indicators to describe their final orientation (and you should use something similar to track their current state as you work through the problem).

Finally, notice that although the rotations themselves are done from the boarder's perspective, we track *their* orientation relative to the outer frame of reference—otherwise they would *always* be in the same orientation relative to themselves.

OK, finally we get to the problem statement and input/output format!

We will provide as input a sequence of flips, spins and twists in some direction (i.e., positive or negative) and for some number of degrees (always a multiple of 90): you can think of this as a "game plan" for the boarder, or as a description of the trick as it would be called by an announcer. We'll use `F` for flips (around the *x* axis), `S` for spins (around the *y* axis), and `T` for twists (around the *z* axis). In addition to the letter code for the axis, we'll also give a plus or minus sign (`+` or `-`) and a number of degrees. (For console-style input, we'll assume these all come together on a line, and that the end of input is signaled by a blank line.) You should track the snowboarders changing orientation and report it at the end of the sequence of moves, using the convention from above, specifying the axis and direction that the top of their head and their extended right arm would point. (Thus they start at *+z/+x.)*

Here is a sample (you don't need to print the current state, as we have in comments, unless you want to):

```
F + 360          (no effect, still +z/+x)

S + 180          (now at -z/-x)

T - 180          (now at -z/+x)

F + 540          (back to +z/+x)

F + 810          (now +y/+x)

     Snowboarder ends up in orientation +y/+x
```

(Note that this example leads to an embarrassing final face-plant! Better luck next time!)