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

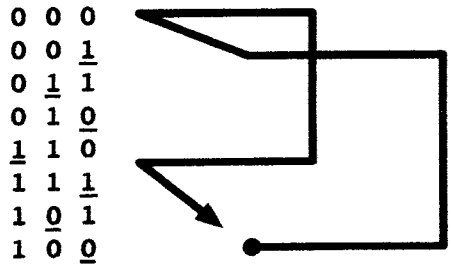
Saturday 20 April 2013
Contest Problems

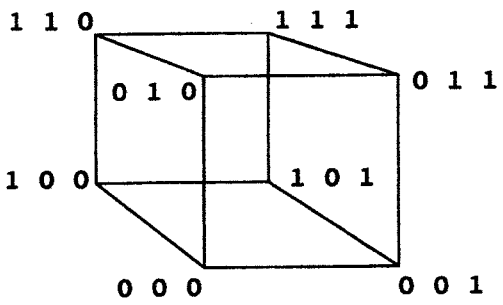
(In the problem statements we assume that input/output will be done "console style" and we try to describe input formatting issues such as separation of parts by blank lines, etc. You are welcome to **prompt** *for the input as you need it, or to write a simple* **GUI** *interface, but you should accept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

1. Captain Gray's hyper-dimensional space liner

You have been hired as a navigator on a hyper-dimensional space liner—Captain Frank Gray wants you to help him guide his ship on its grand tours of the cosmos. For each tour he needs a route that will visit every corner of hyperspace, *never returning to the same spot twice,* until it comes back to the beginning. Along the way, you must always avoid the dreaded "diagonal doldrums", points not along the edges of hyperspace, which cause the quantum flux-capacitors to fry. (This would leave the ship marooned between dimensions, where the ghosts of Schrödinger's cats are said to prowl the dark ... so better get it right!)

As every school-child of the 27th century knows, the cosmos is structured as a multi-dimensional hyper-cube: the exact dimensions are thought to be infinite, but the space liner will only ever tour a specific number of dimensions, which will be given to you in advance, before returning to its origins. Since the diagonal doldrums need to be avoided, the ship must pass from corner to corner of the hypercube, *always along a single edge.* You will use binary coordinates in N-dimensional space to plan your tour, so each stop will correspond to a list of binary digits of length N. The tour will always start and end with co-ordinates of all zeros, and between every two stops *only one binary co-ordinate should change,* either from 0 to 1 or from 1 to 0.





The diagrams and picture above show a typical tour for N=3 (three dimensional space), with the changed bits for each stop underlined—this example illustrates Captain Gray's own favorite touring technique. Next to the tour is a graphical representation of the route and a map of 3-D space with the corners labelled with their binary co-ordinates. Finally, we managed to dig up a picture of Captain Gray sitting in front of his hyperspace navigation console, just for inspiration!

(Note that this is not the *only* tour possible: you could manipulate the same basic path shape, for example, and get something slightly different, but with all the properties we want. You might want to see if you can figure out how Captain Gray's technique works, but you're welcome to come up with your own.)

Your program will get a single input, corresponding to the dimension number N; it should print out a list of binary co-ordinates for a tour of N-dimensional hyperspace (i.e., a list of bit-strings, each one of length N) such that (i) every corner of hyperspace is included (every bit pattern appears once) and (ii) only a single co-ordinate value changes between any two stops. (You don't need to underline the changed ones the way we did in the diagram, but you may do something similar to highlight them if you wish.)

(Our input value N will always be a positive integer greater than or equal to 1: we could consider a 0-dimensional space, but then there would only be one path to print, the empty one ... and there is no point in even printing it out, since it is Already Everywhere™.)

2. Digital sums

We can take the digits of a number and add them up to get another number. If *that* number has more than one digit, we can do it again, and keep on going until we get a single digit as a "final answer". For example:

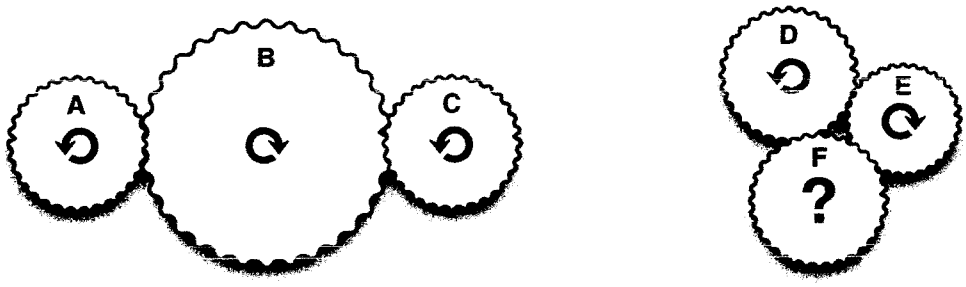
```
87234 → 8+7+2+3+4 → 24 → 2+4 → 6
```

Your program should read in integers, one per line, until a zero value is read—this one just signals the end of input. For each of the other inputs, your program should print out the original number and it's "final answer" (you do *not* need to print intermediate results as we did above). All inputs will be less than 2^{31} (so they should fit in a typical machine integer).

3. Grinding the gears

Imagine a collection of gears placed on a flat surface, like a table-top: some of the gears are big enough to touch and mesh together, so that when one turns, so does the other ... *but in the opposite direction!* (In other words, if two gears mesh, when one gear turns clockwise, the other will turn counter-clockwise.) Gears that don't touch won't interact at all, and for the gear systems we provide, any two gears will either mesh perfectly or not touch at all: in other words, any overlap at all just means they mesh perfectly.

Note, however, that if *three* gears all mesh together so that each one meshes with the other two, then they can't turn at all: one might turn clockwise, the next counter-clockwise ... but there's no option left for the last gear, since it touches each of the other two! In this case the gears will just grind against each other and eventually break.



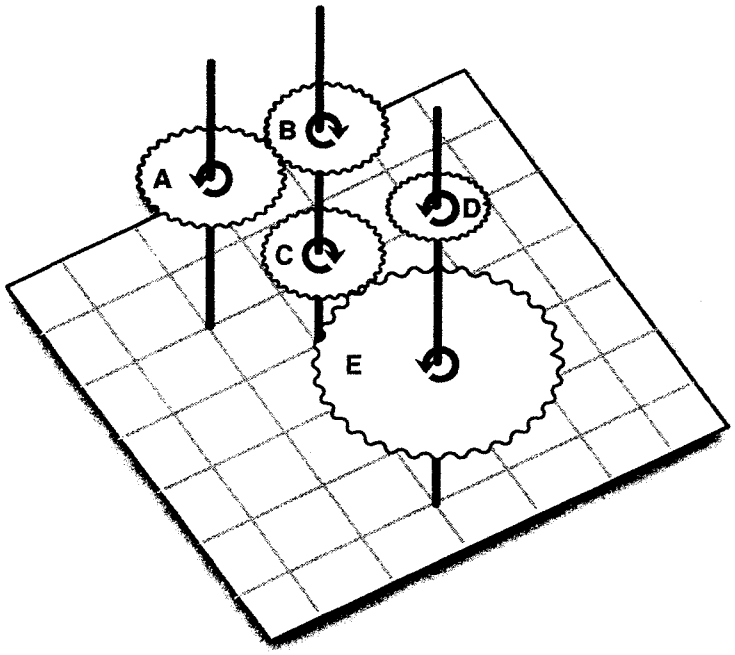
Sometimes 3 gears mesh just fine; sometimes they will grind!

We will provide you with a sequence of gears, described one per line; each line will include one letter (the gear's name, for reference) and three floating-point numbers. The first two numbers will be the x- and y- co-ordinates of the center of the gear, and the third will be its radius. The gear system may include several disconnected groups of gears, or even single gears sitting off to the side somewhere. But *usually* there will be at least some gears that mesh, and thus you will have to decide if they can turn together or not, depending on how many there are and exactly how they mesh. Since floating-point numbers and arithmetic are only approximate, we will design our test data so that there is always a bit of overlap: you can imagine that the overlap just allows for the size of the gear teeth.

Your program should decide if the all the gears in the whole system can turn: if any gears in the system grind together, so there is no way to consistently turn them, then the system as a whole fails and you should report this; specifically, you should **list out the groups of gears, by name, which cannot turn due to grinding.** Each group should be listed out separately (say, on different output lines) and include every gear in the "grinding group". If every gear in the whole system can turn, you should specifically indicate that. There will always be at least one gear, but sometimes there may be *only* one. (A system with a bunch of lonely, non-meshing gears is possible, and will always work, since gears that don't mesh can't conflict.) For the system of gears shown above, you would report (for example) that "`Group D E F grind together.`"

4. Gear-grinding—the 3D edition!

OK, it's time to ramp up our gear problem to the third dimension! Imagine a set of gears as in the last problem, except now the gears may be fixed up or down in the z axis as well. Every gear will be oriented flat and horizontally, but gears that are centered above or below the same (x,y)-point will be connected by a vertical shaft or **axle**. The gears will be welded to the axles so that they can rotate together, but an axle and all its gears *have to rotate the same way—either all clockwise or all counter-clockwise.* As before, some gears will mesh: specifically, if they are in the *same z-plane* and they overlap, they will mesh together and thus constrain each other's rotation. But now, in addition, you must take into account that gears on the same axle also constrain each other to rotate the same way.



In the diagram above, the gears can turn freely, as indicated by the directional arrows. But the addition of just one gear, placed between **B** and **D**, and meshing with both. would cause the whole group to grind. (Note in particular that a gear is part of a "grinding group" if it is attached by an axle to another grinding gear.)

Your program will get the data as one gear per line, just like the last problem, but now formatted as a name (single letter), *three* co-ordinates for the center of the gear (x, y and z), and finally a radius. The axles are not specifically included, but are implied for every gear, and are *the same axle* for gears with the same x and y. Your program should output all groups of grinding gears, as in the last problem, with each grinding group's gear names listed together. For the example pictured above, you would report that the whole system works; but if an extra gear were added between **B** and **D** , you would report all 6 gears as part of a grinding group.

5. Verse and reverse

A *palindrome* is a string that reads the same forwards as it does backwards, for example "`otto`" and "`bob`" are each palindromes. Write a program which will read in a full string on one line and tell if it is a palindrome. Actual inputs may have spaces, punctuation, etc., but we won't begin or end with spaces, just to make things easier to see. You should treat uppercase and lowercase as separate and unequal, so the string

```
Non-sense, eh?he ,esnes-noN
```

would be considered a palindrome, but the string "`Bananab`" would *not*.

6. Word search

A popular pass-time found in puzzle magazines, newspapers and kids' placemats at restaurants is the *word search:* a grid of letters is given, along with a list of "target" words. The goal is to find each of the words someplace in the grid, as a sequence of connected letters. Some searches allow only horizontal, vertical or diagonal "lines" of letters ... but my daughter got an iPad search puzzle that allows the words to be connected any way you like, snaking up, down and around the grid. The only requirement is that each letter *"touches"* the next one in an adjacent position, in any of the eight directions (diagonals included). For example, here's a small grid of letters and some sample words that can be found in the grid, along with information showing where the words were found:

a	x	k	d	u	i
o	b	a	e	p	r
l	e	s	n	a	k
p	w	c	d	o	q
e	n	b	y	t	m

```
able:    (0,0) (1,1) (2,0) (2,1)

baked:   (1,1) (1,2) (0,2) (1,3) (0,3)

random:  (1,5) (2,4) (2,3) (3,3) (3,4) (4,5)

butter:  <not possible>
```

In the example, word locations are described using a list of co-ordinates for the individual letters, with the first number for the *row* (counting down from the top, starting with row 0) and the second number for the *column* (counting from left to right, starting with 0). (By the way, all three words that occur in our sample grid happen to touch *each other,* but that may not be true for all input data: letters of a given word must touch, but words might be separated, run "through" each other, or even overlap for several letters.)

Your program will be given a letter grid as several lines with letters, separated by single spaces for readability. (The grid will always be a full rectangle.) This will be followed by a list of words—you should print a response for each word as a list of co-ordinate pairs or an error message, as shown above.

7. Puttin' in the *-fix*

Sometimes the spelling of two words is such that the end of the first word looks like the beginning of the second. For example, `CAT` ends the same way that `ATTACH` begins: the 2-letter substring `AT` is a *suffix* of the first word and a *prefix* of the second. Write a program which will read in a list of words, separated by spaces, all on a single line, and which will print out a series of numbers representing the *lengths of the longest suffix/prefix* that each pair of consecutive words has in common. For example, for the input:

```
CAT   ATTACH   ACHOO   CHOOSE   SECT   SECTARIAN   FOOBAR   BAROMETER
```

your program should produce the output `2 3 4 2 4 0 3` (note that 0 is a possibility if the words don't end and begin the same at all!):

8. Common factors

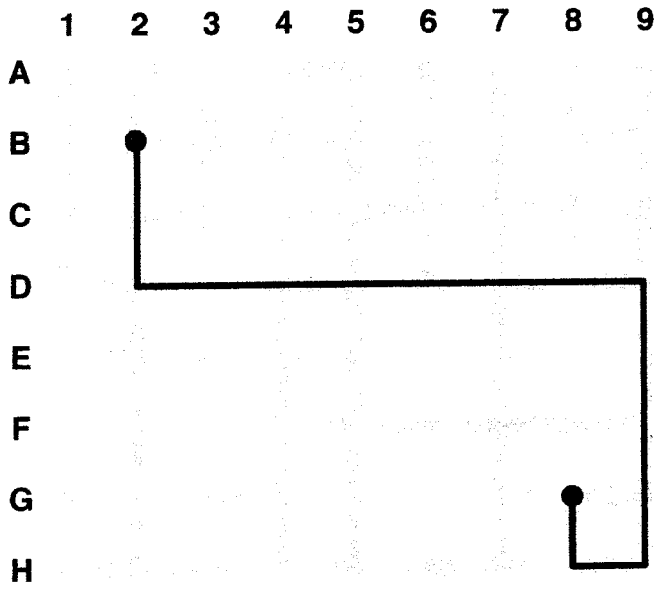
Every positive integer can be broken down into a set of prime factors: when multiplied together, these factors yield the value of the number. The factors are *prime* because they only divide evenly by the number 1 and themselves. For example, the prime factors of 51 are 17 and 3, and the prime factors of 12 are 2 and 3 (note that we don't care how many times the prime factors "factor in", we only care whether they *do* or *not;* also, note that 1 itself is *not* a prime factor, just by convention).

Write a program which will read in a series of positive integers > 1, separated by spaces and terminated by a zero value (which is not part of the series). It should then print out all the prime factors which are *common to the entire series of numbers.* For example, if you read in the numbers `60 90 210 120 0`, you should print out `2 3 5`, since these are the primes which are factors of *all* of the numbers. Finally, to make your output easier to check, print the factors in order, from smallest to largest ... but print each one only once!

9. Moogle Gaps

Everyone knows Moogle™: they're the company that provides on-line maps that can get you from point A to point B with simple directions: start here, turn left, go so far, turn right, go some more .... You've been hired by Moogle to help them generate some tricky directions for city maps that have some *gaps* in their street maps—that's why the product will be called *Moogle Gaps*™. (Sorry!)

In addition to gaps in the maps, these towns also have some one-way roads: this is important because you shouldn't give your users directions that take them the wrong way down a one-way street! The towns we'll be looking at *do* all share a common grid structure, with numbered streets running north-south and letter-named avenues running east-west. We'll describe the structure of the streets using a code detailed below; your program should read in the map codes, followed by two intersections; it should output directions to take a user from the first intersection to the second, avoiding gaps and using one-ways properly.



To describe the street and avenue structure, we'll use this simple scheme: each road will be described on a line by itself, with the *lettered avenues all coming first, then the numbered streets.* For each road, we will provide the name (letter or number, as appropriate) and then a description of each of its *segments:* for two-way segments, we'll just give the starting and ending roads that intersect with an equals sign between (=). For one way segments, we'll write the road names with a less than (<) or greater than (>) symbol. For east-west avenues, the less than will mean one-way going west (left on the map), greater than will mean east (or right). For north-south streets, less than means one-way north (up), greater means one-way south (down). The segments will be separated from the road name and each other by some spaces, and will always proceed from left-to-right or top-down, as appropriate for avenues or streets.

For example, the description of the map shown above might look like this (not all of it, but sample parts):

```
A   2<5
B   1=5   8=9
C   1=3   4=9
D   1=9
  ...
G   1<4   7>9
H   1<9
1   A=D
2   A=H
3   A=D
```

9. Moogle Gaps *(continued)*

Note that the use of these "co-ordinates" doesn't mean there is an actual intersection between a street segment and an avenue: for example, "`G 1<4`" doesn't mean that G Avenue and 1 Street actually intersect (see the map).

After reading in the map data, we would give you a source intersection and a destination intersection. (For the example above, these would be "`B2`" and "`G8`".) Your program should then output instructions of the following for, detailing a legal route from source to destination: the first instruction should specify a direction to face, e.g., "`Face south`". This should be followed by a list of alternating **distances** (in blocks) and **turns** (left or right). For example:

```
Face south. Go 2 blocks. Turn left. Go 7 blocks. Turn right.
Go 4 blocks. Turn right. Go 1 block. Turn right. Go 1 block.
```

(Note that blocks are based on the "background grid", not on actual streets passed—gaps count for their distance!)

10. Home, home on Lagrange

Here's a pleasant problem: the French mathematician Bachet conjectured in the 1600s that every *natural number* (that is, a non-negative integer) can be written as the *sum of four squares*—the Italian mathematician Lagrange finally proved Bachet's conjecture in 1770. The four squares needn't be different: in fact, for the number 2 one needs to use 0 and 1 twice each: $0^2 + 0^2 + 1^2 + 1^2 = 0 + 0 + 1 + 1 = 2$. Some numbers can only be written as the sum of four squares in one way, if we consider different orderings of the same numbers to be the same. For example, for 224 we have: $0^2 + 4^2 + 8^2 + 12^2 = 0 + 16 + 64 + 144 = 224$; and no other combinations work, except for different orderings of 0, 4, 8 and 12.

Your program should read in a number and determine *any* four squares that will sum to that number. Your output should use a special ~~Fermat~~ uh, I mean *format*, as follows: print the sum of the squares using the "hat notation" for the squaring operation, then print the sum using the results of the squarings, then print an equals sign, and then the final answer (i.e., the original goal number). Be a bit careful about the hat (^), by the way: in many languages this is *not* the exponentiation (or power) operator.

For convenience, your program should accept repeated inputs, printing the specially-formatted output for each one. For example, on inputs 0, 17, and 384, your program should print:

```
0^2 + 0^2 + 0^2 + 0^2 = 0 + 0 + 0 + 0 = 0

0^2 + 0^2 + 1^2 + 4^2 = 0 + 0 + 1 + 16 = 17

0^2 + 8^2 + 8^2 + 16^2 = 0 + 64 + 64 + 256 = 384
```

If you *cannot find any* combination of four squares that work, the Lagrange's proof is wrong, and your program should print out the following message:

```
No four-square combination found: mathematics is inconsistent
and the Universe will now dissolve into Nothingness!
```

If you are correct, the Universe will dissolve and the contest, along with everything else, will come to an end! In this case, you will be considered to have won the contest, but there will be no trophy, no competition to gloat over and no ride home ... nor anything else, for that matter! *(So, please,* do *try to find those combos ...)*

11. Hex hacks

As you probably know, digital graphics are made up of *pixels*, small square pieces of a picture, each with its own color. The colors themselves are specified as combinations of bits, 0s and 1s, although we usually write them down in hexadecimal notation, "hex" or base 16. You see, a combination of four bits can be seen as a four-digit binary number: there are 16 combinations in all, so we can abbreviate four bits with a single digit in base 16. In everyday usage we only have ten digits, 0 through 9, so the extra six "digits" are usually taken from the beginning of the alphabet: `'A'` for the ten digit, `'B'` for eleven, all the way up to `'F'` for fifteen.

A colored pixel can be described as a combination of red, green and blue components (these are kind of like primary colors, but for combining light, rather than paint or ink). If we use 256 possible "levels" of red, green and blue for each pixel, we can write out a digital graphic as a series of 6-digit hexadecimal numbers. For example, `5CE6B8` is a slightly muddy shade of blue green and `FFE066` is a bright shade of yellow-orange. (You can find charts and "color pickers" on the web ... but wait until after the contest!)

Now the fact that pictures can be described using hexadecimal or binary is nothing new: in fact, any data can be described this way (otherwise it wouldn't be data!). But here's something interesting: if you want to send somebody some *hidden data, secretly,* you can hide its bits *within* the bits of a digital picture! For example, if you use 8 bits per pixel to represent the blue component of your picture, you and your friend might agree to use the 7th bit to carry your secret message: you just replace the 7th bit of the blue part of every pixel with a bit that you want to send. The picture will come out very slightly distorted, but it probably won't be noticeable in a complex picture. Using this technique you could "hide" a bit in every pixel of a digital picture: once your friend extracts those bits, they could be interpreted as text (say ASCII letters), digital music, another picture, or any other data.

This technique of hiding one digital message inside another is known as *steganography,* presumably because early examples hid secret messages in pictures of dinosaurs with large triangular plates on their backs. ☺

For the problem, we will hide data more efficiently, putting 7 hidden bits in every 6-digit hexadecimal color. We'll describe these bits using a special pattern; for example, R6 will mean "the sixth bit of the red component", or B5 "the fifth bit of the blue component". The data we'll be hiding will be the ASCII codes of a text message, just lowercase letters and blanks (ASCII code 32). We'll always include exactly the same number of hexadecimal codes as we have letters in our message, so each letter will be hidden in one code.

Your program should first read in a sequence of seven patterns describing the bits to be used, then a line containing the letters of the message, and then one or more lines with 6-digit hex codes, separated by blanks, exactly as many codes as the letters. You should output a *modified* sequence of 6-digit hex codes, such that the ASCII codes for the letters, interpreted in binary (0s and 1s) have been used to systematically replace the bits in the original hex message. For example, given this input:

```
R5 R7 R8 G5 G7 B6 B8
hello
ABCDEF ABCDEF ABCDEF ABCDEF ABCDEF
```

your program would produce the output:

```
AACDEA AAC7EB AACFEA AACFEA AACFEF
```

(The hex codes won't always be just one code repeated in the input as is shown here—this just makes it easier to check this one by hand to see how it works.)

For reference, here are the binary versions of "he", two `ABCDEF` codes and the first 2 output codes:

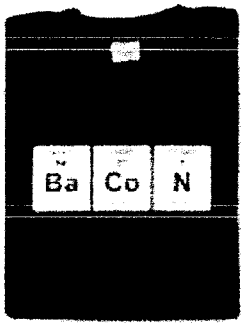
```
1101000                         1100101

101010111100110111101111       101010111100110111101111

101010101100110111101010        101010101100011111101011
```

12. It's elemental, dear Watson!

You may have seen the t-shirts they sell over at *ThinkGeek:* three rectangles with prominent letters, clearly taken from Mendeleev's famous *periodic table* of elements. The letters spell out the word "bacon", (popular with the geeky denizens of sites like Reddit and Slashdot), but they do it in a stylized way, using the codes for the elements Barium (Ba), Cobalt (Co) and Nitrogen (N), yielding "BaCoN" when all strung together.



Well, when a t-shirt design is a big hit, you know someone's going to milk the idea for everything they can. You've been hired by a t-shirt company to help generate ideas for similar shirts, culling out the ones that can be spelled with the letter combinations from the periodic table (well, for the first one hundred elements anyway). Those letter combinations appear below ... but don't type them into your program! Rather, we will input the whole set of letter combos to your program for you, as several lines with separating spaces, or in a GUI text area. All the letter combinations consist of an uppercase letter, possibly followed by a lowercase one (but several of the codes have just the one letter). Once you've read in the codes, we will provide you with a series of single words (no spaces, just letters, all lowercase). If you can build a word using the codes from the periodic table, you should print out a version of the word, *but with the appropriate uppercase/lowercase usage,* just like the famous "BaCoN" shirt from *ThinkGeek.*

```
H  He Li Be B  C  N  O  F  Ne Na Mg Al Si P  S  Cl Ar K  Ca Sc Ti V  Cr Mn
Fe Co Ni Cu Zn Ga Ge As Se Br Kr Rb Sr Y  Zr Nb Mo Tc Ru Rh Pd Ag Cd In Sn
Sb Te I  Xe Cs Ba La Ce Pr Nd Pm Sm Eu Gd Tb Dy Ho Er Tm Yb Lu Hf Ta W  Re
Os Ir Pt Au Hg Tl Pb Bi Po At Rn Fr Ra Ac Th Pa U  Np Pu Am Cm Bk Cf Es Fm
```

So, given the codes above, your program might produce the following outputs for the words given below:

Input	Output
kangaroo	<no possible coding>
clock	Cl O C K
kryptonite	Kr Y Pt O Ni Te
aluminum	<no possible coding>

(If you don't use a GUI or specify otherwise, we will input the codes first, on several lines, then a blank line, and then one input word per line, expecting your outputs to be interleaved with our input words. As always, if you use a GUI, it should be clearly labelled or explained so we understand what to do, and should supply the same basic information as we describe for a console-based style.)

13. Polynomial parsing

Everybody who does programming uses a little algebra: after all, we need variables and arithmetic to get the job done. Sometimes we end up with some pretty big formulas, or *expressions,* in our code: variables, constants, arithmetic operators, and lots of parentheses. Time to simplify!

Your program should read in a large expression, as described below, and "reduce" it down to a simple *polynomial,* i.e., something like $5x^2+3x-7$. In fact, since we don't want you to have to print out all those exponents, let's just write out what are called the *coefficients* of the polynomial. For our example, that would be: `5 3 -7`. We'll be able to tell which exponents go where by the length of the list: this one must start at x^2 because it has 3 coefficients. For this reason, you should be sure to write out any zero coefficients, ***but don't write out any leading zeros*** (just like for numbers: we write 103 and 256, but not 0103 or 00256).

The expressions you read in will consist of constant numbers (like 3 or 0, but never negative), the variable x (possibly several times), and the arithmetic operators +, – and * (that's multiplication): no division! Every use of an operator will have exactly two arguments and will be surrounded by parentheses, like (2+3). But, the arguments may themselves be other expressions; so, in general, you might see something like this:

```
((((x*x)+(3*x))-((8*x)*((6-x)*2)))*(x-1))
```

You could simplify this expression by hand like this (of course, your program might use another method):

```
((((x*x)+(3*x))-((8*x)*((6-x)*2)))*(x-1))
```

$= (((x^2+3x)-(8x*(12-2x)))*(x-1))$

$= (((x^2+3x)-(96x-16x^2))*(x-1))$

$= (17x^2-93x) * (x-1)$

$= (17x^3-93x^2) - (17x^2-93x)$

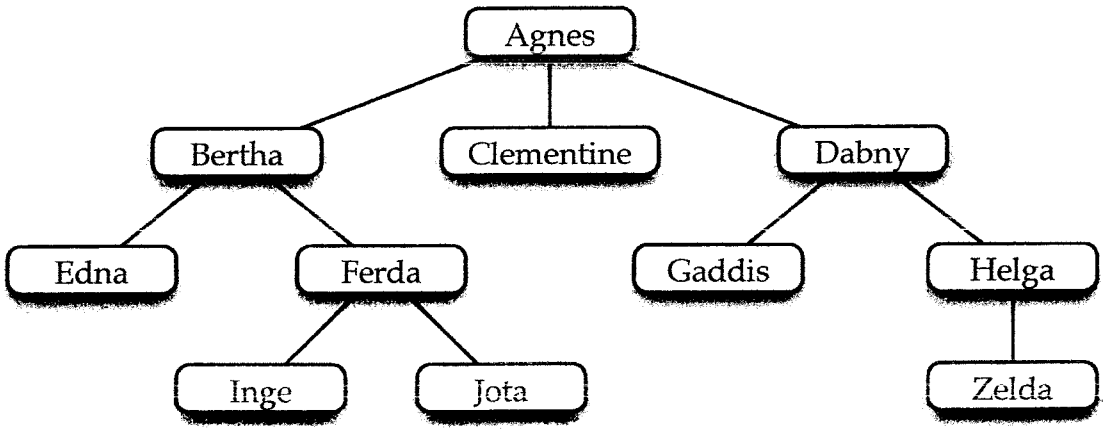
$= 17x^3 -110x^2 +93x +0$

So, if your program read the original expression above, it would print out: `17 -110 93 0.`

14. Royal lineage

After a series of suspicious "accidents" in the royal family of Syldavia, Queen Agnes and her royal investigators have decided that they need a program to keep track of the order of royal succession, and thus help ferret out anyone who might be giving fate a "helping hand". The rules of succession in Syldavia are as follows: the current queen of course holds the title now, and only her female descendants are eligible to rule in the unfortunate event of her demise. The queen's daughters come next in line after the queen, in the order of their age, followed by all the grand-daughters. But the grand-daughters' priority is such that the children of the eldest daughter of the queen all come first, in order of their age, followed by the daughters of the second eldest, etc., down to the daughters of the queen's youngest daughter. Next in line are the great-grand-daughters, etc. So in every generation, the priority is determined not just by age, but rather first by the priority of the mother, and only *then* by age within a set of sisters.

For example, the current situation in Syldavia looks like this, where Queen Agnes is shown at the top of the family tree, and where each set of children has been placed from left to right, oldest to youngest (none of the men are shown in the tree, since they aren't eligible to rule anyway):



Given this family tree, the current line of succession, in order of priority, would be:

```
Agnes  Bertha  Clementine  Dabny  Edna  Ferda  Gaddis  Helga  Ingrid  Jota  Zelda
```

(Note that the names here just happen to be in alphabetical order: this won't usually be true.) Your program should read in the following information: on the first line will be the name of the current queen; on each line thereafter you will find two names and a date of birth (as a whole year number, such as 2013). The interpretation is that the first name is that of the mother, the second name that of a daughter, and the date is the year of the daughter's birth. Other than the current Queen's name, which is always on the first line, the mother-daughter-date lines may come in any order, but you can be sure that once all the information is put together, it will form a valid family tree as above (in other words, no disconnected families, no two daughters born to one mother in the same year, no 'loops' or duplicate names, etc.). If you need a special line terminator (for console input) we will use a line of the form `'XXX XXX 0'`.

For example, the input for the tree above might look like this (with some lines left out to save space):

```
Agnes
Ferda   Ingrid   1965
Agnes   Clementine   1933
Ferda   Jota   1970
Agnes   Bertha   1931
...
Dabny   Gaddis   1942
XXX  XXX  0
```

If your program were to read this input, it should print out the line of succession as shown above.