Twenty-Sixth Annual

Willamette University—TechStart High School Programming Contest

Saturday 3 March 2012

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. Change is in the air

Carrying change around in your pocket can be a drag ... literally! As someone who wants to walk with a skip in their step, you make exact change whenever you can—but sometimes you just have to break a buck!

Write a program to determine if you can make exact change for a given amount, assuming that you have certain numbers of each coin in your pocket. Your program should read in 5 integers (they may be zero, but they won't be negative): the first number represents the amount of money you owe a cashier, and the next 4 represent the numbers of pennies, nickels, dimes and quarters you have in your pocket (in that order). If you can make *exact* change, you should print out a line describing how many of each coin to use; if not, you should print a message indicating that exact change is not possible. For example, on input:

```
87   4 3 3 2
```

your program might produce the output:

```
Make 87 cents with 2 quarters, 3 dimes, 1 nickel and 2 pennies.
```

Note: you don't have to print out the *best* way to make the change—any way that adds up right will do!

(But you might want to think about this idea as a follow-up question, after the contest: what does 'best' mean? Should you get rid of as many coins as possible? As much combined weight as possible? As many annoying, *small coins?)*

2. What's the frequency, Kenneth?

Write a program that will read in a string consisting of letters, spaces and punctuation (a typical sentence) and report back the frequency with which each of the *letters* occurs in the sentence. The report should list the letters in order of frequency, from the one occurring most often to the one occurring least often. *If two letters occur the same number of times, report them in alphabetical order.* You should ignore spaces, punctuation and differences in upper- and lowercase (i.e., count every occurrence of a letter, regardless of which case it is). Finally, you should *only report letters that actually occurred in the input—**don't report any zero frequencies!***

The input may span several lines—if you use a command-line interface, the end of input will be signaled by an empty line. Here are an example input and output (this input happens to be just one line):

Input:

```
The rains in Spain run mainly down the drain.
```

Output:

```
7n 5i 4a 3r 2d 2e 2h 2s 2t 1l 1m 1o 1p 1u 1w 1y
```

3. Multimedia maven

Your friend Kim is obsessed the latest multimedia offerings technology has to offer: he's constantly buying the latest pods, pads and widgets so he can listen to music, watch movies and read books wherever he goes. He's also constantly downloading digital "content" from `NetFlakes`, `iCrunes` and `Columbia.com`, but all this media is putting some strain on his bandwidth.

Let's say his total bandwidth is some number of megabytes (MB) per second (an input you'll get); his download software will split all of the available bandwidth equally between all the files he is currently downloading. So, once the first file has finished downloading, the full available bandwidth will be "re-split" among all of the remaining files equally. So, if he has 10MB per second bandwidth and there are 10 files to download in all, they will each get 1MB per second to start ... but after the first one finishes, the remaining nine will each see a slight increase in speed, up to 10 MB/sec divided by 9, or about 1.1 MB/sec.

Your program should read in a bandwidth amount (just a floating-point or "decimal" number) and then a list of file names and sizes. In response, it should print out, for each file name, the time that it will finish downloading, in minutes and seconds, relative to a fixed start time for the whole bunch. *You should print the files in the order they **finish**,* and *resolve any ties using alphabetical order by filename.* You should show the minutes and seconds for each file downloaded in the usual way, with a colon and leading zeros; for example: `03:15` for three minutes and fifteen seconds. *Round up to the nearest whole second to avoid fractions!*

For example, the list of files in the first column, under a total bandwidth of 6.0 MB per second, would produce the second column of file names and times in the order in which they will complete:

Input:

```
6.0
hoo   420.0
goo   240.0
foo   540.0
zoo   360.0
yoo   180.0
too   480.0
```

Output:

```
yoo   @  03:00
goo   @  03:50
zoo   @  05:10
hoo   @  05:40
too   @  06:00
foo   @  06:10
```

4. Taxation vexation

In a certain state (clearly not Oregon!) there is an 8.5% sales tax. (For those who haven't made a lot of purchases outside of Oregon, this means that buyers must pay an extra 8.5% on the *total* of each purchase, which the seller later remits to the state government to be spent for the public good.) In order to help you get used to the idea of sales tax (who knows what the future brings for Oregon?), we would like you to write a program which will compute the average price of an item in a purchase of n items, given the value of n and the *total purchase cost, tax included.*

Your program should read in the number n (a *non-negative* integer) and a floating-point (or "decimal") number (also non-negative) representing the total cost. You may assume there will be exactly two digits after the decimal point, if this helps. You may read these numbers in with a GUI, using a clearly-labeled interface, or on a command line, prompting for the input, as you see fit. Your program should output the ***average pre-tax cost of the items,*** rounded to the nearest penny (an exact half penny rounds down, anything more rounds up). Here are some examples, using a command-line style:

```
Number of items? 10
Total price? 10.85
Average pre-tax item cost was: 1.00

Number of items? 5
Total price? 10.91
Average pre-tax item cost was: 2.01
```

5. Sorting by hand

People playing card games usually sort their cards so they can find them easily and tell at a glance what sort of hand they have. For most games, they probably sort first by suit and then by rank ("rank" is the card number, or the face value if it's a face card). Before we get to our specific sorting problem, let's agree to use two-character codes for cards, a digit or letter for the rank, and then a letter for the suit, as follows:

`2` = two	`c` = clubs
`3` = three	`d` = diamonds
...	`h` = hearts
`9` = nine	`s` = spades
`j` = jack	
`q` = queen	
`k` = king	
`a` = ace	

So, for example, the nine of spades would be coded as "`9s`" and the queen of clubs would be "`qc`".

But our little coding key above gives me an idea: why not sort cards by the *English words* that describe them? And just to keep things interesting, let's also sort the way people usually do, *by suit first and then by rank,* even though our codes put things in the other order!

Your program should read in a series of card codes as strings (we should be able to fit them all on a line) and print out the same card codes, but in order sorted by suit-then-rank, using the English names to sort.

Input:

```
9s  jd  9c  7h  7c  jc
```

Output:

```
jc  9c  7c  jd  7h  9s      (here "nine" comes before "seven", but "clubs" before "diamonds")
```

6. Binary palindromes

Every non-negative integer can be represented as a binary numeral (base 2) using the digits 0 and 1. If the binary representation of a number reads the same forward and backward, let's call it a *binary palindrome.*

Write a program which will read non-negative integers in decimal (base 10), one per line, and report "`Yes`" or "`No`" depending on whether or not they have a binary palindrome. In order to make your results easy to verify, you should also print out the binary numeral itself. If you need it, your program should can request a `-1` in the input as a sign to stop (there is no need to give an answer this one). A sample run of your program might look like this (formatting is not too important, just the results):

```
102          binary: 1100110                          No
33           binary: 100001                           Yes
123456789    binary: 111010110111100110100010101      No
1193         binary: 10010101001                      Yes
```

Note: inputs will never be larger than about 2 to the 31st power, but may be *as small as 0.*

Also note: as you can see from our first example, we never include *leading* zeros (if the number is exactly 0, we just use that single digit).

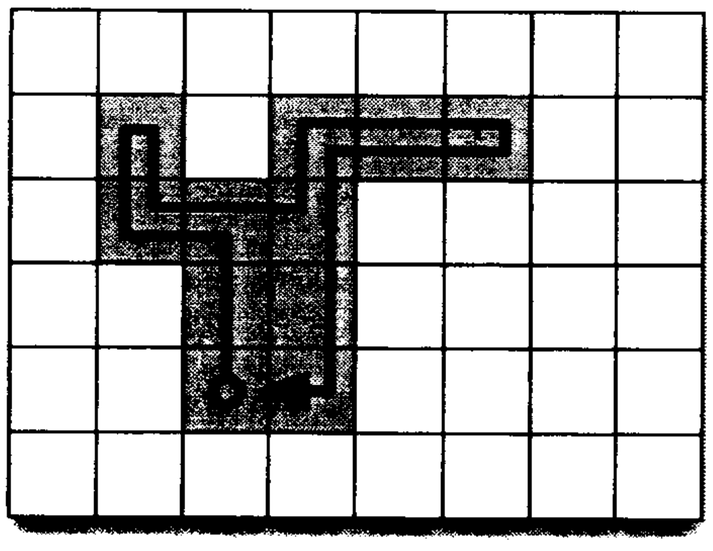
7. Martian meanderings

The next NASA probe is headed to Mars! Your friend Howard is working on the Martian rover, and he's asked you to help by writing some of the code for this latest robot explorer. The rover will use camera data and an on-board AI (artificial intelligence) to determine a proposed path of exploration through the Martian landscape: NASA engineers and exo-geologists will then review the plan and send modifications back to the rover by interplanetary radio.

Your job is to read in a proposed path from the robot and return a report with various statistics that the NASA scientists will use. The robot will roll around on a virtual grid, as if the Martian surface were tiled into squares. The planned path will be in terms of the directions the robot will take (North, South, East or West), with each move always the same unit length and only in those four directions, and turning in place when necessary. In this way the robot will enter and observe a number of grid squares during its trip, sometimes doubling back through the same square on its way.

Your program should read in a string representing the path (we'll spread some spaces in between the letters unless you request otherwise) and report the following summary statistics:

- the total **length** of the robot's path, i.e., how many steps it takes (this should be easy);
- the total **area**, in grid squares, covered by the robot—this may be different than the path length because the robot may cover the same square repeatedly while doubling back;
- the total number of **edges** between the grid squares the robot explored and those it left unexamined—this includes any "inside edges" due to snaky paths or even to "donut holes", as in the second example below. In other words, this is the outer perimeter of the area the robot covers, viewed as a solid shape made of square tiles, *plus the inside edges/perimeters of the donut holes.* (I think this edge policy makes it even easier to calculate than just the outer perimeter.)

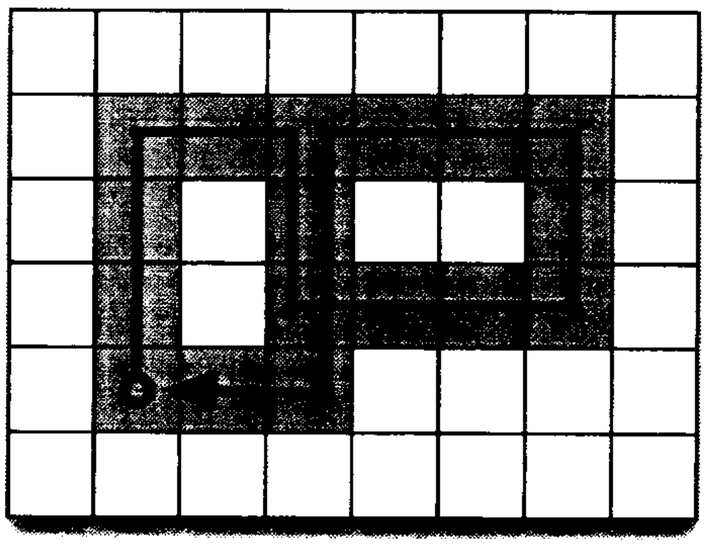


Path: `N N W N S E E N E E W W S S S W`

Length: `16 moves`

Area: `11 squares`

Edges: `20 units`



Path: `N N N E E S S E E E N N W W W S S S W W`

Length: `20 moves`

Area: `17 squares`

Edges: `32 units`

8. Web spinner

A certain spider lives on a web with a ten-fold symmetry, as shown below. A handful (or is it a mouthful?) of flies are stuck on various strands scattered around the web. Our eight-legged friend will scuttle quickly from his central lair, gobble them up one by one, and then return to wait for more prey. But he wants he makes his grisly rounds as *efficiently as possible* … .

Your job is to determine a **minimal path**, from the center spot where he sits *and back again,* that visits *all* of the flies he's caught. (It's *minim***al**, not *minim***um**, because there might be more than one way to do it, all of the same length.) Let's agree that the flies will always sit on some strand of the web—we can code their

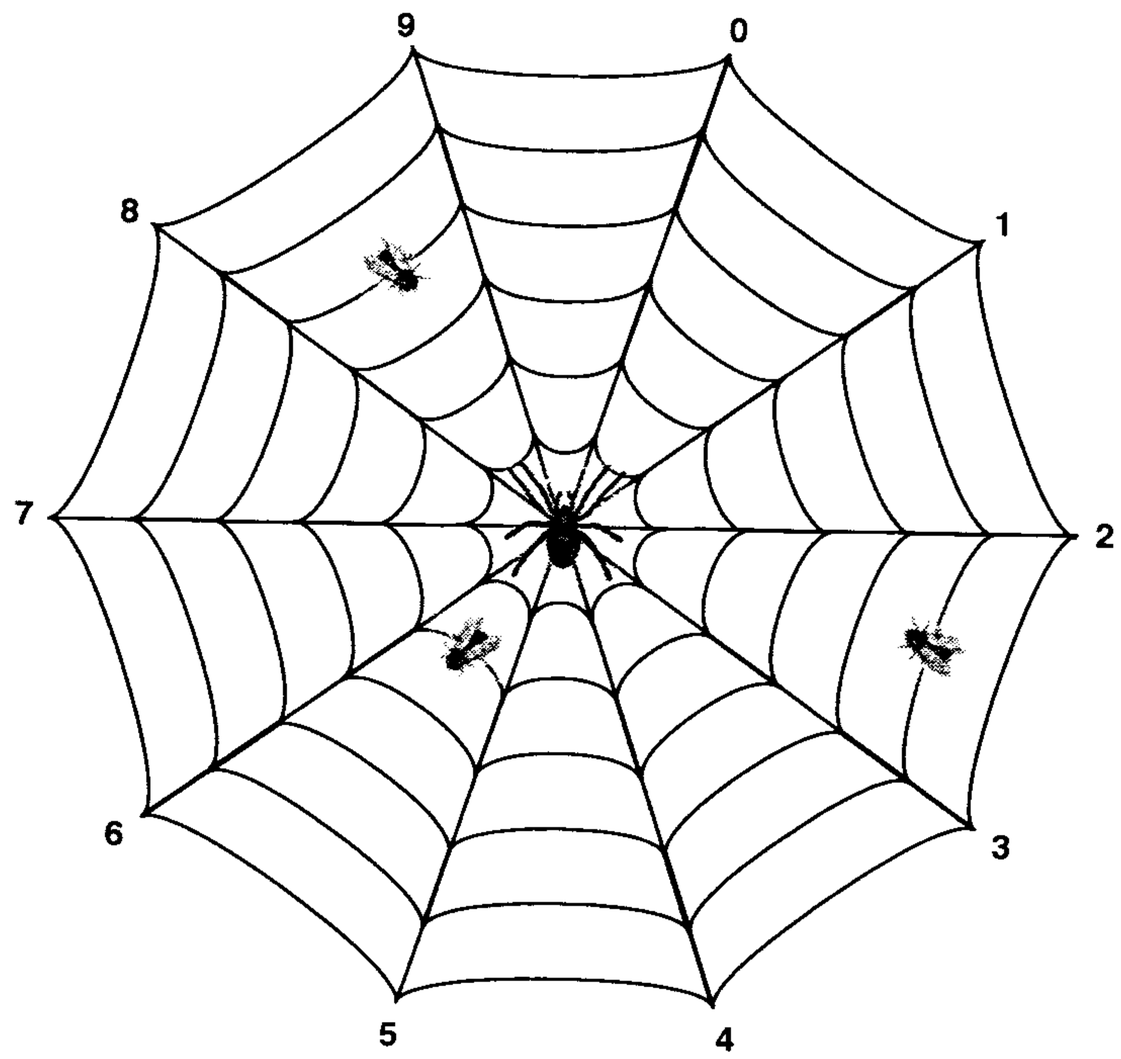
positions as two "coordinates", one which represents how far they are **out from the center**, and another to tell **how far around** they are on the web, going clockwise, as in the diagram below. Since flies always sit on web strands, at least one of their two coordinates will end in "`.0`", but the other one may be a "proper decimal" or floating-point number. (This doesn't matter much to the problem, but seems true anyway.)

Our spider's path, on the other hand, will always be taken in whole number steps, as he travels from one intersection to another by whole web strands. We'll describe his path as a **sequence of directions**: *in* or *out* (toward or away from the center) and *clockwise* or *counterclockwise* (around the web, "circularly"). We can use the letters **`I`** and **`O`** for in and out, and the letters **`C`** and **`X`** for clockwise and counter-clockwise.
Ahh, but what about the very middle? When he's in the center, we must tell which of the numbered "spokes" he follows—all moves ***out** from the center point* will be coded as a single digit, **`0`**–**`9`**. (Note that he starts in the center and ends there, but that he might also pass *through* it on the way to a tasty fly.)

For the sample pictured, we might have the following input and output:

Input: `4.0 8.6  5.0 2.5  2.0 5.5`

Output: `5 O C  C C O O C  C C C O C  I I I I I`



9. Triangular hull

A common problem in computational geometry is to find the "convex hull" of a set of points: roughly speaking, the smallest polygon (a shape with straight-line edges) which contains every point in the set and doesn't "bend inwards". Hmmm … that's a pretty hard problem, so let's save it for later! (You can read about it on Wikipedia when you get home.)

For this contest problem, let's try something a little bit easier: we will give you a shape made out of a bunch of *connected square tiles on a grid:* you should give us the *integer* vertices of the smallest triangle, **as measured by perimeter**, which would fully enclose the tiles. For example, in the diagram below, the gray squares show the areas of interest and the dark black lines show two enclosing triangles for the same shape. The one on the left has the smallest perimeter, so that's the one we want (but any other triangle with the same perimeter would be OK). Your input will consist of a series of pairs of integer numbers, all in a row separated by spaces, representing the coordinates of each tiles's *lower left-hand corner.* Your output should be a similar set of just 3 pairs of integer coordinates for the intersection points of the triangle you found, *plus your (correct!) calculation of the perimeter.*

9. Triangular hull *(continued)*

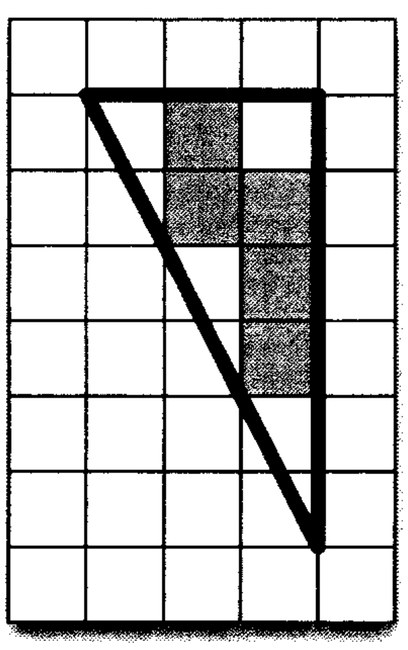
Assuming the lower left corner of the diagram is `0 0`, input and output for the pictured example would be:

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

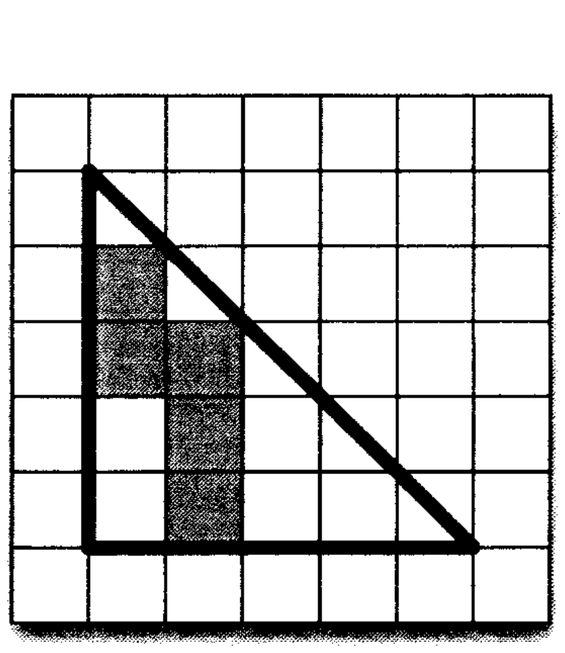
Output:
```
4 1  4 7  1 7

Perimeter = 15.7
```

Note: the squares will always be connected, but as with the Martian rover, there might be "donut holes".



Perimeter: 15.7



Perimeter: 17.07

10. Words within words

The NSA *(National Snooping Agency)* needs your help: they are looking for coded messages which have been hidden inside of texts in emails. They know that certain key words are being used, but they believe that these words have been obscured by mixing them in with other letters to make longer words. *All the letters of the code word must appear, **in order**, in the larger word.* For example:

```
spy         spray
            ^^  ^

bond        blonde
            ^ ^^^

tile        reticulated
              ^^  ^  ^
```

Your mission is to read in lines of text, one at a time, where each line contains two words. You should report back for each line read in whether or not the first word on the line is contained within the second word, as described above. For example:

```
spy  IS in  spray

bond  IS NOT in  trouble
```

You may assume that the words will never be longer than 30 letters, but make no assumptions about which word is longer (you should check whether the first word is contained in the second, but of course this is not possible if the first word is longer!). Once a letter in the second word has been "used up" in correspondence with one on the first word, it can't be used again (i.e., you must continue checking from the next letter forward).

11. Operation insertion

Your help is needed back at the NSA again, this time to help decode secret messages being exchanged by certain suspicious parties. The messages consist of sequences of numbers: the NSA believes that they are codes for important mathematical equations, but all the arithmetic operators and the equals sign have been left out. Your job is to find a way of inserting operators (only addition, subtraction, and multiplication) between the numbers, ***along with a single "equals" sign,*** so that the resulting equation becomes true (other NSA experts will then figure out what the equations mean).

Unfortunately, the people writing these coded messages have left out parentheses, so you must make sure to obey the usual "order of operations": multiplication is done *before* addition and subtraction, and adjacent additions and subtractions group to the left. Under these rules, `10-2-3` is interpreted as `(10-2)-3`, not as `10-(2-3)` (and thus evaluates to 5), whereas `10-2+3` is interpreted as `(10-2)+3`, not as `10-(2+3)` (and thus evaluates to 11).

Your program should read in a line with some integers on it (all will be on a single line, no more than 10 and no fewer than 2). You should output any equation that can be made by inserting operators and the equals sign so that, when evaluated according to the rules of order and grouping above, it will yield a true answer. If there is **no** such equation (no way to get a true answer), you should indicate this. (If there are several possible ways to make an equation, you only need print out one of them!)

Here are a couple of sample inputs and outputs:

```
Input:   1 2 3 10 3
Output: 1 + 2 * 3 = 10 - 3

Input:   1 5 8
Output: No equation is possible.
```

You may use any mix of operations *(including possibly none of them)* and you may use the same operation multiple times.

12. Traffic turmoil

Consider a typical suburban street intersection with a four-way stop: cars arrive at various times from the four directions (North, East, South and West), but they may have to wait their turn before they can proceed through to their destinations. Of course they must wait if there is a car in front of them, but they also wait for a car coming from another direction if that car arrived before they did. And if cars are waiting from several directions at the same time, they must proceed through "round robin" in a clockwise order, looking down from above, with the first car to arrive going through first (this is also known as the "yield to the car on your right" rule). OK, you say, but what if several cars arrive at the same time and no one else is there? In such cases we'll let the car coming from the North (if any) go first; and if none from the North, then one from the East, or South or West, in that order. (This rule is a bit arbitrary, but *someone* has to go first!)

In real life, two cars might go through at once if there's enough room and they don't interfere—but our careful drivers will only go **one car at a time!** Finally, we'll also say that cars take some **fixed time** to proceed through, say **10 seconds**.

OK, here's the problem: given a list of car labels (like `a`, `b`, `c`, `d`) and times of arrival at the intersection (in minutes and seconds), list out the cars in the order they go through the intersection, labelled with the times they *finish* going through (i.e., *after* their 10 seconds of passage).

Your program should read in a series of lines with a car label (just a single lowercase letter), arrival direction (an uppercase letter from `{N,E,S,W}` and an arrival time; it should print out a similar sequence of lines with letters and *leaving* times. (Note that it doesn't really matter which direction the cars leave the intersection.)

12. Traffic turmoil *(continued)*

We will always list cars in the order of their arrival time, and no two cars will arrive from the same direction at the same time (ouch!). *You should print the car labels and times out in order of their leaving the intersection!* Here is some sample input and output (you could read all the input before providing output—we list them side-by-side just to save space):

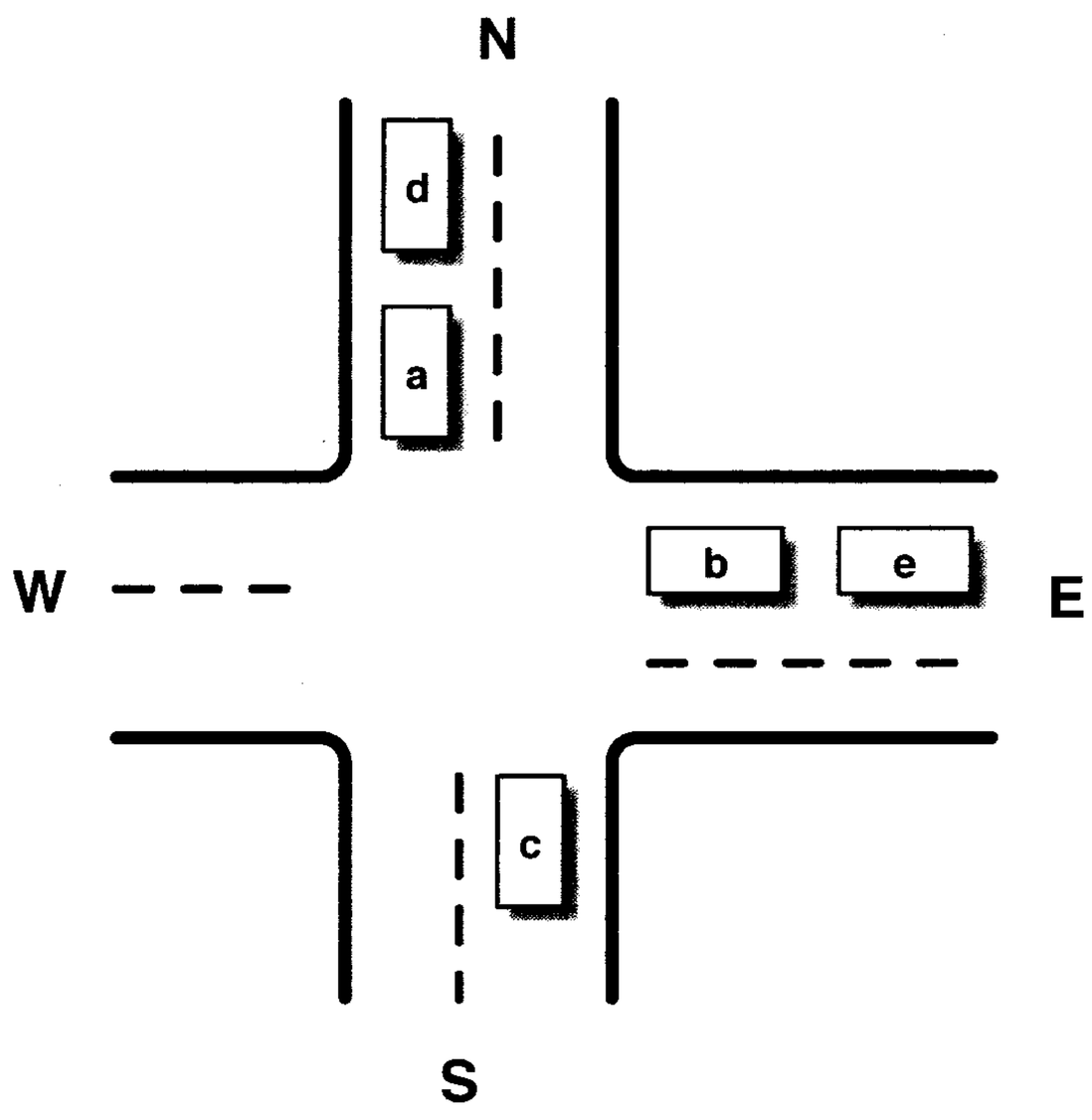
Sample input:

```
a   N   5:00
c   S   5:20
d   W   5:20
b   E   5:20
p   E   5:25
q   E   5:35
u   N   7:00
```

Sample output:

```
a   5:10
b   5:30      (b gets priority here, based on direction)
c   5:40
d   5:50
p   6:00      (p had to wait for his turn "round robin")
q   6:10      (q had to wait for his turn behind p)
u   7:10      (u must arrive before she leaves!)
```

(**Note:** *The diagram below is meant to illustrate the general situation—it's not accurate for the sample data.)*



13. Freaks, geeks and cliques

Everyone seems to be on `MyFace.com` these days, sharing the minutiae of their every daily lives with all their friends. But some things never change, including the "clique" phenomenon at high schools: you know how it goes: the jocks, the nerds, the space cadets, the Potterheads—students in each clique tend to hang together and not mix too much with the others … or do they? If we had access to the `MyFace` data (I hear it's available, for a price!), we could figure out how exclusive these so-called cliques really are: does the school really break up into a bunch of non-overlapping groups, or is there more mixing than meets the eye?

For this problem we will use lowercase letters to represent students at a school: your program will read in lines representing their friendship relations (connections) and determine how many *separate, non-overlapping cliques* there are. A clique is defined as a group of students each of whom is friends with someone else in the same group. Students in a clique don't have to all be friends with *everyone* else in the clique, but they will always have a connection, even if it's just indirect. And if there are two or more cliques, nobody in one clique will have *any* connection to anybody in the other. In the extreme cases, *all* the students might be connected in one big clique, or some students might have no friends at all and be just a clique by themselves (but let's hope they're not **Forever Alone**™!).

Your program should read in a series of lines with a student letter followed by a colon and then a (possibly empty!) list of their friend's letters. Every student involved will always have exactly one "defining line". You should print out a numbered list of all cliques (the order doesn't matter), of whatever size, showing all the members of that clique. Every student letter should be output *somewhere*, even if only by themselves.

Note: *sometimes the friendships may run just "one way", but the students still count as "connected".*

Input:

```
a: b c
t: f
d: p
m: n
p:
q: p
n: a
b: a z
z:
f: t
c: a z
```

Output:

```
Clique #1:  q d p
Clique #2:  m c a n b z
Clique #3:  f t
```

14. Doing it with class!

In many object-oriented languages, a *class* is a template for objects that have the same kind of parts. These parts, also called fields or members, are usually divided into values (let's call them *data fields*) and pieces of programs, called methods. For this problem we will **ignore the methods** and focus on **just the data fields**.

Our data fields will have *types:* some will be basic values like integers, but others will be references to other objects, each with their own data fields. For the basic values, we will use four lowercase letters as type names: `i`, `b`, or `d`—you can imagine these stand for integer, boolean and double (i.e., double-precision floating point values). For type names that are classes, we will use uppercase letters—but they will include *whatever class names are defined in the problem,* rather than a fixed set like the basic types. We will write a class definition all on a line, with the "`class`" keyword, a class name and a bunch of types and field name pairs, each terminated by a semi-colon. Here's a sample definition of a class `A` with integer (type `i`) fields named `x` and `y`, and a class-typed field called `p` of type `C` (we'll give a definition of class `C` below):

```
class A { i x; i y; C p; }
```

We will also allow *extended classes* (usually called sub-classes) and write them with a greater-than sign. When we have class `B > A`, it means that values of class `B` have all the fields of class `A`, plus some new ones which are described as above with braces and semi-colons. For example, here is a definition for a `class B` which extends `A` by adding a new field called `z`, of type integer (`i`), and a new field called `q`, of class type `A`:

```
class B > A  { d z; A q; }
```

Remember: every object of type `B` also has the `A` fields (`x`, `y` and `p`)—but not vice versa!

If we have a variable `b` whose type was class `B` and we wanted to get its `z` field, we would write this using the `var` keyword as follows (the order is: `var` keyword, type, variable name, colon, accessor expression):

```
var B b: b.z
```

and this expression would have the type `i`: we can just look it up in the definition of class `B`. But if we access the `q` field of `b`, it has type `A`, and so we can then access further "sub-fields", such as the `x` field:

```
var B b: b.q.x
```

Of course we might continue this kind of *accessor expression* indefinitely, as long as we continue to have class types, and as long as we use valid field names for the class types we have. Once we get to a basic type like `i` (for integer), we can no longer access further fields; but the final type of an expression may be a basic type *or* a class type. *(Note that you also have to watch out for fields in sub-classes!)*

OK, here is (finally!) the problem: your program will read in a series of class definitions, followed by some typed variables and proposed accessor expressions. You should check that each proposed expression is valid and tell what type the complete expression has. If it is *not valid* (because of bad field names, or fields not from the correct class), reply with `Invalid!` Here's a full example:

Input:

```
class A { i x; i y; C p; }
class B > A  { d z; A q; }
class C { d x; B r; C s; }

var B b: b.q.y
var C h: h.r.p.s.x
var A m: m.p.r.q.z
var B u: u.q
```

Output: `Valid with type i`

Output: `Valid with type d`

Output: `Invalid!`

Output: `Valid with type A`