UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

**Programming Contest** **P. N. Hilfinger**
**Fall 1995**

## 1995 Programming Problems

To set up your account, execute

```
source ~ctest/contest/bin/setup
```

in all shells that you are using. (This is for those of you using the C-shell. Others will have to examine this file and do the equivalent for their shells.)

You have 5 hours in which to solve as many of the attached eight problems as possible. Put each complete solution into a single $N$.c file (for C) or $N$.C file (for C++), where $N$ is the number of the problem. Each program must reside entirely in a single file. Each file should start with the line

```
#include "contest.h"
```

and must contain no other `#include` directives. Upon completion, each program must terminate by calling `exit(0)`.

Aside from files in the standard system libraries and those we supply, you may not use any pre-existing computer-readable files to supply source or object code; you must type in everything yourself. The standard system libraries do *not* include the `gcc` class library. You may not use utilities such as `yacc`, `bison`, `lex`, or `flex` to produce programs. Your programs may not create other processes (as with the `system`, `popen`, `fork`, or `exec` series of calls). You may use any inanimate reference materials you desire, but no people. You can be disqualified for breaking these rules.

When you have a solution to problem number $N$ that you wish to submit, use the command

```
submit N
```

from the directory containing $N$.c or $N$.C. Before actually submitting your program, `submit` will first compile it and run it on one sample input file. No submission that is sent after the end of the contest will count. You should be aware that `submit` takes some time before it actually sends a program. In an emergency, you can use

```
submit -f N
```

which submits problem $N$ without any checks.

You will be penalized for incorrect submissions that get past the simple test administered by `submit`, so be sure to test your programs. All tests will use the compilation command

`contest-gcc` $N$

followed by one or more execution tests of the form (Bourne shell):

$N$ `<` *test-input-file* `2>` *junk-file*

The output of each input file is then compared with a standard output file. In this comparison, leading and trailing blanks are ignored and sequences of blanks are compressed to single blanks. Otherwise, the comparison is literal; be sure to follow the output formats *exactly.* Make sure that the last line of output ends with a newline. Your program must not send any output to `stderr`; the temporary file *junk-file* must be empty at the end of execution. Each test is subject to a time limit of about 15 seconds. You will be advised by mail whether your submissions pass.

The command `contest-gcc [-g]` $N$, where $N$ is the number of a problem, is available to you for developing and testing your solutions (as usual, the optional `-g` is for debugging information). It is equivalent to

`gcc -Wall -o` $N$ `-O [-g] -I`*our-includes* $N$`.[Cc] -lg++ -lm`

The *our-includes* directory contains `contest.h`, which also supplies the standard header files.

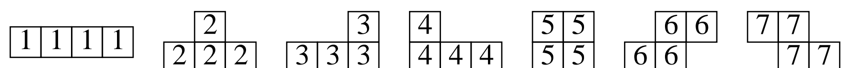
All input will be placed in `stdin`. You may assume that the input conforms to any restrictions in the problem statement; you need not check the input for correctness. Consequently, you are free to use `scanf` to read in numbers and strings.

**Scoring.** Scoring will be according to the ACM Contest Rules. You will be ranked by the number of problems solved. Where two or more contestants complete the same number of problems, they will be ranked by the *total time* required for the problems solved. The total time is defined as the sum of the *time consumed* for each of the problems solved. The time consumed on a problem is the time elapsed between the start of the contest and successful submission, plus 20 minutes for each unsuccessful submission, and minus the time spent judging your entries. Unsuccessful submissions of problems that are not solved do not count. As a matter of strategy, you can derive from these rules that it is best to work on the problems in order of increasing expected completion time.

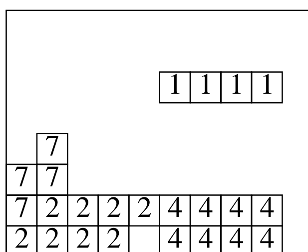
**Protests.** Should you disagree with the rejection of one of your problems, first prepare a file containing the explanation for your protest, and then use the `protest` command (without arguments). It will ask you for the problem number, the submission number (submission 1 is your first submission of a problem, 2 the second, etc.), and the name of the file containing your explanation. Do not protest without first checking carefully; groundless protests will be result in a 5-minute penalty (see Scoring above).

**Terminology.** The term *free-form input* indicates that input numbers, words, or tokens are separated from each other by arbitrary whitespace characters. By standard C/UNIX convention, a whitespace character is a space, tab, return, newline, formfeed, or vertical tab character.

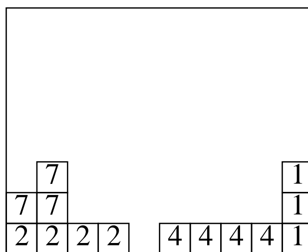
**1.** The familiar computer game TETRIS involves having objects fall one at a time from a height into a rectangular box, with the player able to control their horizontal positions and to change their orientations, rotating them by multiples of 90 degrees. Each object is composed of four unit squares glued together along their edges in one of the following seven shapes (each shown in what we'll call its *standard orientation*)




Each time the accumulation of objects in the box fills the entire width of the box with a row of squares, all of those squares disappear, and the squares above all move down one unit. For example, consider the following configuration, containing two #2 pieces, two #4 pieces, one #7 piece, and a falling #1 piece.




By rotating the falling piece 90 degrees (either direction) and moving it horizontally all the way to the right, it will fall into place and fill in the second row from the bottom, causing that row to vanish and giving rise to the following configuration.




The idea of the real game is to put off as long as possible having the accumulation of objects rise above the edge of the box.

You are to write a program to simulate a game of TETRIS that uses a simple strategy. The input to your program will consist of a sequence of sets of data in free format. Each set consists of two integers $W > 3$ and $H > 3$, giving the width and height, respectively, of the playing box, followed by a sequence of digits in the range 1–7, indicating types of pieces that fall from the top, terminated by a digit 0. You may assume that the pieces will not, in fact, overflow the top of the region (as indicated by $H$), but you may make no assumptions about the maximum values of $H$ and $W$.

Your program is to keep track of the accumulated objects (and fragments of objects) as each new one is added. It is to fit each new object into the existing heap by configuring it (positioning it horizontally and rotating it) according to the following heuristic.

- The object is to be configured so as to place one of its squares in the lowest possible position of all possible configurations.
- Of those configurations that put a square in the lowest position, the one that places that square farthest to the left is preferred.
- Of those configurations that put a square lowest and farthest to the left, the one that involves the least rotation clockwise from its standard orientation (see above) is preferred.

Conceptually, each object is configured before it falls past the top opening of the rectangle and then allowed to fall as far as possible (this is less flexible than what one can do in the actual game, but it simplifies the problem).

For each set, output is to consist of an identifying label for the set, followed by printout of the final configuration that results from following the heuristics given above, all in the format illustrated in examples that follow. Each set in the output is to be separated by a blank line from the one before it.

Input

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


4 6
2 5 6 7 3 6  0
```

Output

```
Set 0:

|          |
|          |
|          |
|          |
|      1   |
| 7    1   |
|77   21 55|
|2227721444|
+----------+

Set 1:

| 66 |
|66  |
| 733|
|77 3|
| 255|
|222 |
+----+
```

**2.** All fractions written in octal (base 8) notation may be expressed exactly in decimal notation. For example, 0.75 in octal is 0.953125 ($7/8+5/64$) in decimal. Specifically, a numeral requiring $N$ octal digits to the right of the octal point may always be written as a decimal numeral with no more than $3N$ digits to the right of the point. The reverse, of course, is not true.

Write a program to convert octal numerals between 0 and 1, inclusive, (without sign and with leading '0.') into equivalent decimal numerals. The input to your program will consist of zero or more octal numerals in the specified range separated by whitespace. Each numeral has the form $0.d_1d_2\cdots d_k$, where the $d_i$ are octal digits (0..7). There is no limit on $k$.

The output will consist of a sequence of lines—one for each input—having the form

$$\texttt{0.}d_1\cdots d_k\ \texttt{[8]}\ \ \texttt{= 0.}D_1\cdots D_m\ \texttt{[10]}$$

where the left side echos the input and the right side is the base-10 equivalent. The right side must satisfy the condition $D_m \neq 0$.

Here is an example of the desired format.

| Input | Output |
|---|---|
| `0.75 0.0001` | `0.75 [8] = 0.953125 [10]` |
| `0.01234567` | `0.0001 [8] = 0.000244140625 [10]` |
| | `0.01234567 [8] = 0.020408093929290771484375 [10]` |

**3.** CLUNK is a simple programming language that contains only assignments and conditional branches. A CLUNK program has the form

`0:` $S_0$`;` `1:` $S_1$`;` $\cdots$ $N$`: stop;`

The consecutive decimal numerals before the colons are *statement labels*; there is no upper bound on $N$. Input is in free form; there may be arbitrary whitespace around the statement labels, the components of the statements (the $S_i$) and the punctuation marks, but not within a component (e.g., `stop` must appear as shown, with no spaces separating its letters). Each statement ends with a semicolon.

Only the last statement in a CLUNK program is `stop`. Each of the other $S_i$ has either the form of

- an *assignment:*

  $I$ `= F(`$E_1, E_2, \cdots, E_k$`)`

  —where $k \geq 0$ and $I$ and all of the $E_i$ are *variables*, written as letters (upper- or lower-case, with 'a' and 'A' being different); or

- a *conditional branch:*

  `if F(`$E_1, E_2, \cdots, E_k$`) goto` $M$

  where the $E_i$ and $k$ are as before and $M$ is a statement label (between 0 and $N$, inclusive).

The value $k$ can be different in each statement. When $k = 0$, the expression is `F()`. (The `F` is just there for show; that's why it's the same all the time.)

You are to write a program to detect *use before definition* in CLUNK programs. That is, given any CLUNK program, you are to determine if it is possible that a variable in the program might apparently be *used* before it has been assigned a value (in compiler terminology, before it is *defined*). More specifically, you should report that use before definition is possible if there is some apparently possible sequence of statements from the input program that leads to a statement in which one of the $E_i$ is a variable that does not appear to the left of any preceding assignment statement in the sequence. A sequence of CLUNK statements is *apparently possible* if

- Statement 0 is the first statement in the sequence, and
- If an assignment statement numbered $j$ is in the sequence, statement $j + 1$ is the next statement in the sequence, and
- If a conditional branch statement numbered $j$ and containing the clause `goto` $L$ is in the sequence, then the next statement in the sequence is either number $j + 1$ or $L$.

For example, in the CLUNK program on the left, statement 3 might involve a use before definition (of the variable $x$).

```
0: y = F();
1: if F(y) goto 3;
2: x = F(y);
3: x = F(x, y);
4: stop;
```

```
0: y = F();
1: if F(y) goto 5;
2: x = F(y);
3: if F(y) goto 1;
4: if F() goto 6;
5: x = F(y);
6: x = F(x,y);
7: stop;
```

Even though x is set in statement 2, the sequence $< 0, 1, 3 >$, which does not contain statement 2, is possible according to the rules. The program on the right has no possible uses before definition; all apparently possible paths to statement 6 define x.

The input to your program will consist of zero or more programs in the specified format. For each input program, the output is to identify the input set, and then print either

```
No use before definition
```

or

```
Possible use before definition
```

as shown in the sample outputs below.

| Input | Output |
|---|---|
| 0: y = F(); 1: if F(y) goto 3;<br>2: x = F(y);<br>3: x =<br>F(x, y);<br>4:<br>stop;<br><br>0: y = F();<br>1: if F(y) goto 5;<br>2: x = F(y);<br>3: if F(y) goto 1;<br>4: if F() goto 6;<br>5: x = F(y);<br>6: x = F(x,y);<br>7: stop; | Set 0: Possible use before definition<br>Set 1: No use before definition |

**4.** Given a non-empty alphabet of lower-case letters and a string composed only of characters from that alphabet, it makes sense to talk about the *smallest* string over the alphabet that does *not* appear as a substring of that string. By "smallest" here, I mean the shortest, breaking ties between equally-short strings in favor of the one that is first in dictionary order.

For example, suppose that the alphabet consists of the letters `a`–`c` and the string is '`bcabacbaa`.' The smallest string that does not appear is '`bb`.' For the same alphabet, the smallest string that does not appear in '' (the empty string) is '`a`', and the smallest that does not appear in '`aabacbbcacc`' is '`aaa`.'

The input to your program will consist of zero or more sets of input. Each set will begin with two lower-case letters—call them $x$ and $y$—possibly separated by whitespace. It is guaranteed that $x \leq y$ in alphabetic order. The alphabet for a set will be all letters between $x$ and $y$, inclusive. Following $y$, and possibly separated from it by whitespace, will be a string of zero or more lower-case letters terminated by a period.

For each set of input, your program is to echo the input and then print the smallest non-occurring string in the format illustrated below. Separate sets of output by blank lines.

| Input | Output |
|---|---|
| <pre>a c<br>bcabacbaa.<br>a c.<br>a c aabacbbcacc.</pre> | <pre>Set 0:<br>The smallest string that does not occur in<br>    bcabacbaa<br>is bb for the alphabet a-c.<br><br>Set 1:<br>The smallest string that does not occur in<br><br>is a for the alphabet a-c.<br><br>Set 2:<br>The smallest string that does not occur in<br>    aabacbbcacc<br>is aaa for the alphabet a-c.</pre> |

**5.** Many organizations use some variety of the voting procedure known as *single transferable vote* to elect candidates to office. One is trying to elect someone to an office and has some number $C > 1$ of candidates for the position. The voting members submit ballots that are numbered lists of their preferences: the (single) candidate they list as #1 being their first preference, #2 their second, and so forth for as many candidates as they wish to indicate. When $C > 2$, no one candidate need get a majority (we'll ignore abstentions here). This is where the "transferable" part comes into play. A ballot for candidate Mary is *transferable* to candidate John if John has not yet been eliminated from the race and the ballot lists John after (that is, with a higher preference number than) Mary. Ballots are *non-transferable* if there is no such preference listed.

For as long as no candidate has a majority of the total ballots cast and there is more than one candidate, we eliminate one candidate with the lowest number of ballots, and transfer that candidate's ballots—each ballot going to the next not-yet-eliminated candidate listed on it. Ballots that do not list another uneliminated candidate are said to be *exhausted* and are not given to any candidate. When more than one candidate has the lowest number of ballots, one of them is eliminated at random.

You are to write a program to determine the outcomes of a set of elections by single transferable vote. The input to your program will consist of zero or more sets of input data, each corresponding to one election. Each set of data consists of a list of two or more candidates, one per line, in the form

```
John Doe
Mary Roe
```

A candidate's name always begins with a letter. You may assume that no name is longer than 80 characters. Each candidate has a *candidate number*; the first candidate listed is #1, the second #2, etc. The list of candidates is followed by a list of ballots, one per line. Each ballot consists of an ordered list of candidate numbers, most-preferred first. The list of ballots is terminated by a line containing a non-positive number.

After the list of ballots, there is a list of random numbers in the range 1 to $N$, where $N$ is the number of candidates, followed by a non-positive number. Whenever there is a choice of which of $K > 1$ candidates to eliminate, choose the next of the random numbers, call it $r$. If $r \bmod K = 0$, eliminate the candidate with the lowest candidate number; if $r \bmod K = 1$, eliminate the candidate with the second-lowest candidate number, etc. (For our purposes, the operator 'mod' means "remainder.") You may assume that there are enough random numbers for all the random choices you'll need. There may be too many random numbers, in which case you must be sure to discard the unused ones. It is possible that no candidate ever gets a majority, in which case the candidate who is left after all others are eliminated wins. ("Majority" always means more than half of the total number of ballots, *including* exhausted ballots.)

The output should have the form shown in the examples. Report the total number of ballots for each candidate at each round, and then report the winner at the end. A round ends whenever someone wins or someone is eliminated.

Input

```
John Black
Mary Brown
Walter Jones
Charles Bradford
1 3 2 4
1 2 3
2 1 4 3
2 4 1
2 1
2 1 4 3
4 3 1
1 4 2 3
3 2 1 4
3 4 2 1
1 2
2 1 3 4
4
-1
4 1 2 1 3 1
3 4 1 0
Dorothy Gale
Christopher Robin
Alice Liddell
1
2
3
-1
3 2 1 2 0
```

Output

```
Election 1.
Round 1.
John Black: 4
Mary Brown: 5
Walter Jones: 2
Charles Bradford: 2
> Walter Jones eliminated.

Round 2.
John Black: 4
Mary Brown: 6
Charles Bradford: 3
> Charles Bradford eliminated.

Round 3.
John Black: 5
Mary Brown: 7
> Mary Brown has a majority.

Mary Brown wins with 7 votes.

Election 2.
Round 1.
Dorothy Gale: 1
Christopher Robin: 1
Alice Liddell: 1
> Dorothy Gale eliminated.

Round 2.
Christopher Robin: 1
Alice Liddell: 1
> Christopher Robin eliminated.

Alice Liddell wins with 1 votes.
```

**6.** A *term* may be any of the following:

- A *symbolic constant,* denoted by a single lower-case letter.
- A *pattern variable,* denoted by a single upper-case letter.
- A *structure* having the form

$$(\,\phi\,\tau_1\;\tau_2\cdots\tau_n\,)$$

where $\phi$ is a symbolic constant, and each $\tau_i$ is a term, $1 \leq i \leq n$, $n \geq 0$. (Actual input is free-form, and spaces may appear anywhere in a term).

We say that two terms *unify* if one can replace each pattern variable with some term (possibly a different replacement for each distinct pattern variable, but the same replacement for any two instances of the same variable) so as to make the two terms identical (ignoring whitespace). The assignment of terms to the pattern variables is called a *unifier*.

For example, the following pairs of expressions all unify.

| Term 1 | Term 2 | Unifier |
|---|---|---|
| X | x | X=x |
| a | a | |
| (g z) | (g z) | |
| (f q) | (f Y) | Y=q |
| (p Z (r Q) (z)) | (p (g Q) P Q) | P=(r (z)) Q=(z) Z=(g (z)) |

Notice in particular that pattern variables may appear in *either or both* terms. The following do *not* unify:

| Term 1 | Term 2 |
|---|---|
| a | b |
| (f q) | (g Y) |
| (g Q (a)) | (g (b) Q) |

You are to write a program that takes as input pairs of terms, and for each term indicates whether it unifies. The input will be free-form as shown in the examples. The output echoes the input and includes a report in the format shown in the examples. You may assume that you will never have to "match up" two pattern variables against each other, as in

`(f Q)` *and* `(f Z)`

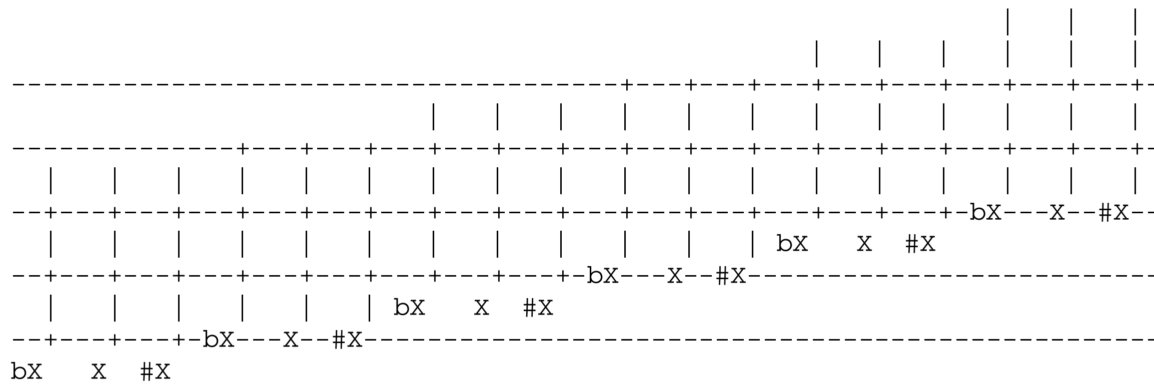
You may also assume that you will never encounter cases where you "match up" a variable against a term that contains that variable, as in

`(f Q)` *and* `(f (h Q))`

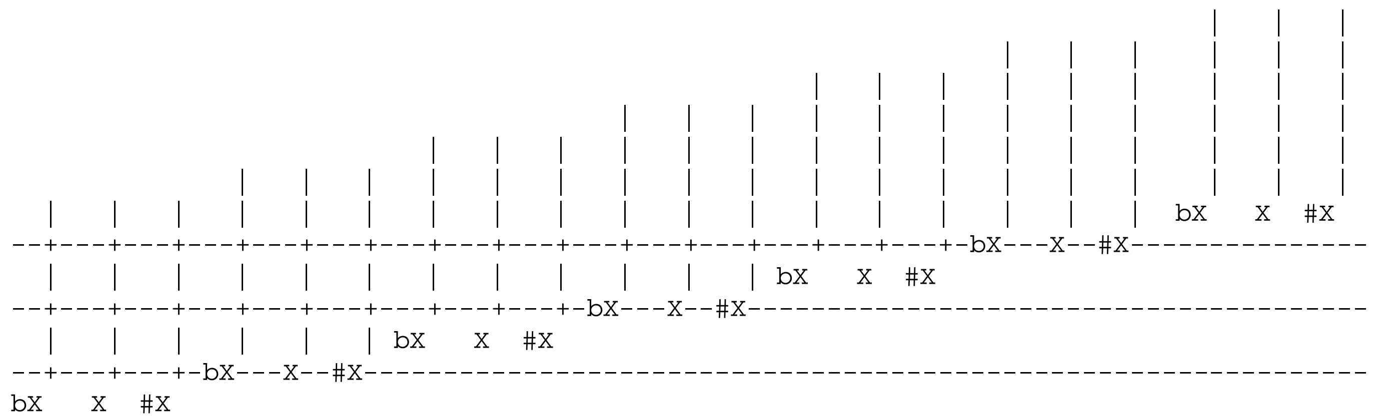
You need not check for either of these conditions. When listing unifiers, list the variables in alphabetic order.

| Input | Output |
|---|---|
| `X  x` | `X and x unify: X=x` |
| `a` | `a and a unify:` |
| `a` | `(g z) and (g z) unify:` |
| `(g z)  (gz)` | `(f q) and (f Y) unify: Y=q` |
| `(f q)` | `(p Z (r Q) (z)) and (p (g Q) P Q) unify: P=(r (z)) Q=(z) Z=(g (z))` |
| `(f Y)` | `a and b do not unify` |
| `(p Z (r Q) (z))` | `(f q) and (g Y) do not unify` |
| `(p (g Q) P Q)` | `(g Q (a)) and (g (b) Q) do not unify` |
| `a  b` | |
| `(f q)  (g Y)` | |
| `(g Q (a))` | |
| `(g (b) Q)` | |

**7.** A musical note has a pitch and a value (duration). It is traditionally written on a staff consisting of five horizontal lines. A note's vertical position and certain *accidental markings* indicate its pitch; its shape indicates its value. For this problem, we'll use a simplified version of the notation, rendered in plain ASCII text. The possible pitches and the textual names used for them in this problem are as follows:

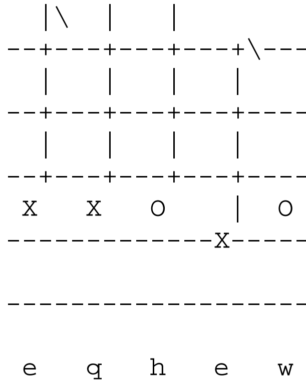



```
 Db   D    D#   Eb   E    E#   Fb   F    F#   Gb   G    G#  2Ab  2A   2A#  2Bb  2B   2B#
```




```
2Ab  2A   2A#  2Bb  2B   2B#  2Cb  2C   2C#  2Db  2D   2D#  2Eb  2E   2E#  2Fb  2F   2F#  2Gb  2G   2G#
```

The note values, with their notations, are as follows:




The letters stand for ‘eighth,’ ‘quarter,’ ‘half,’ and ‘whole’. I’ve shown two eighth notes to indicate how the flag part (‘\’) is to be drawn when it does and doesn’t cross a line of the staff.

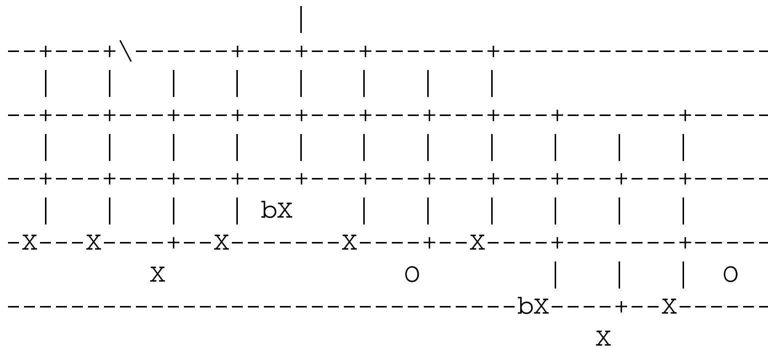
Your program is to take as input a list of notes, each of the form $PD$, where $P$ is one of the pitch values (`G`, `2A#`, etc.) and $D$ is one of the four values (‘e’, ‘q’, ‘h’, and ‘w’). The input is in free form, but no spaces appear in the middle of a note (e.g., ‘`D#e`’, not ‘`D# e`’.)

The output of your program is to be a staff with notes in the format drawn above (not including the text notations for pitches and note values). Here is an example.

```
Input:

Gq Ge Fq Gq        2Abq
Gq Fh     Gq Ebq Dq
Eq Fw


Output:
```



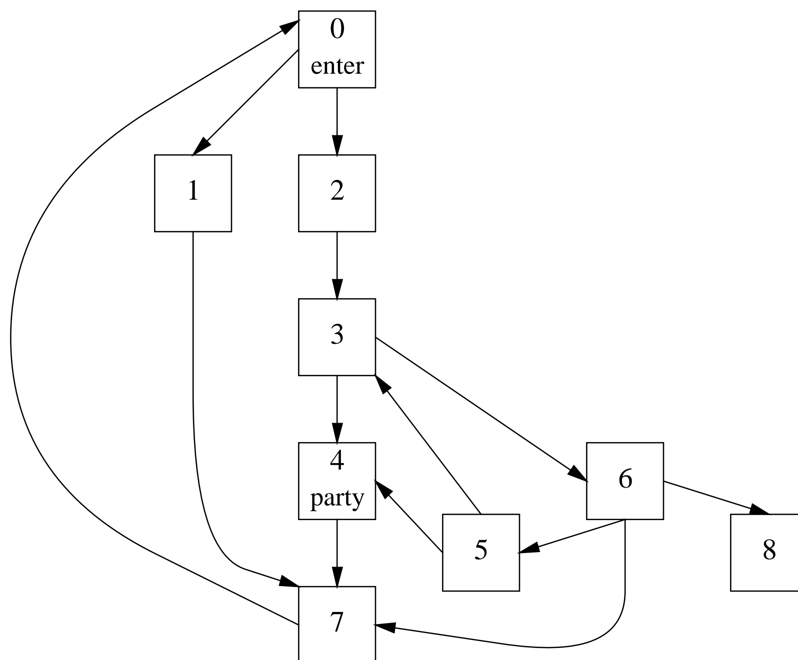
**8.** There is a party being held in one cavern of a large set of caverns. The caverns are connected by fast-moving beltways running through tunnels that run from cavern to cavern, each corridor in one direction (is this contrived or what?). The whole complex has one entrance cavern with doors to the outside world; the party is not in the entrance cavern.

Since the party has an open bar, people have to be carded before they enter. The party is expected to be pretty wild, and the management would just as soon not let rowdy underage minors even get near it. Since they own the caverns, they decide that they'll station guards to check IDs at the beginnings of corridors leading out of another cavern in such a way that all people who want to get to the party have to pass by these guards. To minimize the irritation to people who have no intention of going to the party, the management wants to find a cavern (just one; they have only so many guards, after all) such that

- In order to get to the party from the entrance cavern, one must pass through that cavern, and
- There is no other cavern with this property that is closer to the party (other than the cavern actually holding the party).

Under these conditions, it is guaranteed that there is exactly one such cavern.

For example, with the cavern map below, the management would station its guards in cavern 3 (cavern 2 would also do, but 3 is closer).




The input to your program is in free form. It will consist of an integer, $N$, giving the number of rooms, followed by a number $P$ in the range $1 \leq P < N$, giving the room containing the party, and followed by a sequence of pairs of numbers, each in the range 0 to $N - 1$. The first number of each pair indicates the cavern at which a tunnel begins and the second indicates the cavern at which it ends. Cavern 0 is always the entrance.

The output is to consist of a single line:

`Put guards in cavern` $N$.

where $N$ is a cavern number (0 to $N - 1$).

For example, here is the input and output for the sample configuration above.

| Input | Output |
| --- | --- |
| 9 4<br>0 2 2 3<br>3 4 5 3 5 4<br>3 6 6<br>5 6 7 6 8<br>4 7 0 1 1 7<br>7 0 | Put guards in cavern 3. |