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

Programming Contest, *Sponsored by* **Google** **P. N. Hilfinger**
Fall 2007

2007 Programming Problems

Please make sure your electronic registration is up to date, and that it contains the correct account you are going to be using to submit solutions (we connect names with accounts using the registration data).

To set up your account, execute

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

in all shells that you are using. (This is for those of you using csh-like shells. Those using `bash` should instead type

```
source ~ctest/bin/setup.bash
```

Others will have to examine this file and do the equivalent for their shells.)

This booklet should contain eight problems on 17 pages. You have 5 hours in which to solve as many of them as possible. Put each complete C solution into a file N`.c`, each complete C++ solution into a file N`.cc`, and each complete Java program into a file N`.java`, where N is the number of the problem. Each program must reside entirely in a single file. In Java, the class containing the main program for problem N must be named `P`N (yes, it is OK to have a Java source file whose base name consists of a number, even though it doesn't match the name of the class). Do not make class `P`N public, or the Java compiler will complain. Each C/C++ file should start with the line

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

and must contain no other `#include` directives, except as indicated below. Upon completion, each program *must* terminate by calling `exit(0)` (or `System.exit(0)` in Java).

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. Selected portions of the standard `g++` class library are included among of the standard libraries you may use: specifically, the headers `string`, `vector`, `iostream`, `iomanip`, `sstream`, `fstream`, `map`, and `algorithms`. Likewise, you can use the standard C I/O libraries (in either C or C++), and the math library (header `math.h`). In Java, you may use the standard packages `java.lang`, `java.io`, `java.text`, `java.math`,

and `java.util` and their subpackages. 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 or their Java-library equivalents). 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`, N`.cc`, or N`.java`. 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 compiling or running it.

You will be penalized for incorrect submissions that get past the simple test administered by `submit`, so be sure to test your programs (if you get a message from `submit` saying that it failed, you will *not* be penalized). All tests (for any language) will use the compilation command

`contest-gcc` N

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

`./`N `<` *test-input-file* `>` *test-output-file* `2>` *junk-file*

which sends normal output to *test-output-file* and error output to *junk-file*. The output from running each input file is then compared with a standard output file, or tested by a program in cases where the output is not unique. 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*. It will do no good to argue about how trivially your program's output differs from what is expected; you'd be arguing with a program. Make sure that the last line of output ends with a newline. Your program must not send any output to `stderr`; that is, the temporary file *junk-file* must be empty at the end of execution. Each test is subject to a time limit of about 45 seconds. You will be advised by mail whether your submissions pass.

In the actual ACM contests, you will not be given nearly as much information about errors in your submissions as you receive here. Indeed, it may occur to you to simply take the results you get back from our automated judge and rewrite your program to print them out verbatim when your program receives the corresponding input. Be warned that I will feel free to fail any submission in which I find this sort of hanky-panky going on (retroactively, if need be).

The command `contest-gcc` N, where N is the number of a problem, is available to you for developing and testing your solutions. For C and C++ programs, it is roughly equivalent to

`gcc -Wall -o` *N* `-O2 -g -I`*our-includes* *N*`.* -lm`

For Java programs, it is equivalent to

`javac -g` *N*`.java`

followed by a command that creates an executable file called *N* that runs the command

`java P`*N*

when executed (so that it makes the execution of Java programs look the same as execution of C/C++ programs). The *our-includes* directory (typically `~ctest/include`) contains `contest.h` for C/C++, which also supplies the standard header files. The files in `~ctest/submission-tests/`*N*, where *N* is a problem number, contain the input files and standard output files that `submit` uses for its simple tests.

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 C/C++ programmers are free to use `scanf` to read in numbers and strings and `gets` to read in lines.

Terminology. The terms *free format* and *free-format input* indicate 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. A *word* or *token,* accordingly, is a sequence of non-whitespace characters delimited on each side by either whitespace or the beginning or end of the input file.

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). The Judge will *not* answer technical questions about C, C++, Java, the compilers, the editor, the debugger, the shell, or the operating system.

Notices. During the contest, the Web page at URL

`http://inst.cs.berkeley.edu/~ctest/contest/announce.html`

will contain any urgent announcements, plus a running scoreboard showing who has solved what problems. Sometimes, it is useful to see what problems others are solving, to give you a clue as to what is easy.

1. [Miguel Revilla, adapted] The Soundex Index System was developed so that similarly sounding names or names with similar spelling could be encoded for easy retrieval. It has been used by the U.S. Bureau of the Census, and some states use it to help encode driver's license numbers. Your task is to convert a sequence of names into the corresponding Soundex codes.

A Soundex code always consists of a letter followed by three digits, such that:

1. The first letter of a name appears (unencoded) as the first character of the Soundex code, and is capitalized. It is also the only letter.

2. The letters 'A', 'E', 'I', 'O', 'U', 'Y', 'W' and 'H' are never encoded when they are not the first character in a word. They do serve, however, to break sequences of like-coded letters (see next rule).

3. All other letters are encoded according to the following table, *except* when they immediately follow a letter (including the first letter) that would be encoded with the same code digit, according to the table:

Code	**Key letters**
1	B, P, F, V
2	C, S, K, G, J, Q, X, Z
3	D, T
4	L
5	M, N
6	R

4. Codes are truncated after the third digit.

5. Trailing zeros are appended as needed so all names are encoded with a letter followed by three digits.

The input contains a sequence of words in free format, and ends at end of file. The output written to the file should consist of a column of names and a column of their corresponding soundex codes in the format shown in the example below.

Example:

Input	Output
`LEE`	`LEE => L000`
`Kuhne EBELL`	`Kuhne => K500`
`ebelson`	`EBELL => E140`
`SCHAEFER SCHAAK`	`ebelson => E142`
	`SCHAEFER => S160`
	`SCHAAK => S200`

2. A puzzle I once heard asks for an English word from which one can remove the most letters, one at a time, such that each resulting word is itself a valid English word. For example, you can remove seven letters from "starting":

$$\text{starting} \Rightarrow \text{stating} \Rightarrow \text{statin} \Rightarrow \text{satin} \Rightarrow \text{sati} \Rightarrow \text{sat} \Rightarrow \text{at} \Rightarrow \text{a}$$

For this problem, we'll look at the general problem of finding such chains in sets of arbitrary strings.

The input to your program will consist of a sequence of problems. Each problem will consist of a set of one or more words in free format. A word will consist of a sequence of one or more letters, possibly interspersed with hyphens and apostrophes (single quotes). Each set of words is terminated by a single semicolon (separated from any preceding or following input by whitespace). The final problem in the input is followed by another semicolon.

The output is to consist of a sequence of lines, one per problem, giving the problem number, the word from the set that allows the most letters to be removed, and the number of letters that may be removed, in the format shown in the examples below. When computing chains of words, ignore capitalization (treat upper-case letters as equivalent to their lower-case versions) and ignore all non-letters in a word. However, when printing the maximal word, print its input form, with all capitalization and punctuation marks intact. Words may be repeated in the input. When more than one word is maximal, choose the one that occurs first in the input.

Example.

Input	Output

`tatter satin skating`
`a I O stating bat cat job can`
`at sat baffle starting consume`
`satisfy sati`
`quote statin prose`
`zebra`
`;`
`tatter satin skating`
`a I O stating bat cat job can`
`at sat`
`baffle starting consume`
`satisfy`
`quote statin prose`
`zebra`
`;`
`ant boy a car an`
`can't any ;`
`;`
	`Problem 1: can remove 7 letters from "starting"`
	`Problem 2: can remove 3 letters from "starting"`
	`Problem 3: can remove 3 letters from "can't"`

3. [Miguel Revilla, adapted] During excavations in a far-off wasteland known as the Soude, archaeologists have discovered badly worn tablets containing mysterious symbols. After a long investigation, the project scientists have concluded that the symbols might be parts of equations. If this was true, it would indicate that the ancient residents were more sophisticated than previously believed.

The problem, however, is that the only symbols found on the sheets are digits and parantheses. The working hypothesis is that whatever symbols the former inhabitants of the Soude used for arithmetic symbols and for equality were relatively small, and have all worn away. The principal investigator is further assuming, on the basis of vague indications from these and other documents, that they knew only of three arithmetic operations—addition, subtraction, and multiplication—and that they did not use prioritization rules for arithmetic operators, but grouped all terms strictly left to right in the absence of parentheses. For example, for them the term $3 + 3 \times 5$ would be equal to 30, and not 18.

You are to find out whether the hypothesis is sensible or not by checking to see whether any possible addition of one '`=`', and some combination of the symbols '`+`', '`-`', and '`*`' to the lines on the tablets result in valid equations. For example, on one sheet, the string "`18␣7␣(␣5␣3␣)␣2`" has been discovered. Here, one possible solution is "`18=7+(5-3)*2`". But if there was a line containing "`5␣3␣3`", then this would mean that the people of the Soude did not mean it to be an equation. As with our equations, equal signs should not appear within parentheses. Consider binary operators only (so '`-`' never means unary negation).

The input to your program will consist of a sequence of putative equations, one per line. Each line contains up to 12 positive integer numerals whose product will be less than 2^{31}. There might be some parentheses, always balanced, around groups of one or more numerals. There will always be at least two numerals, and there will never be parentheses around the entire line, so that there will always be a valid place to put an equals sign. There is no other limit for the number of the parentheses in the equation. There will always be at least one space between each numeral or parenthesis and the next symbol. Otherwise the occurrence of white space is unrestricted. A line containing only the numeral 0 terminates the input, and should not be processed.

For each equation, output one line containing a solution to the problem, i.e., the equation with the missing '`+`', "`-`", "`*`", and "`=`" signs inserted. Do not print any white space in the equation. If there is no way to insert operators to make a valid equation, then output the line "`Impossible`". If there are multiple solutions, print any one. Use the format shown in the example below.

Example:

Input	Output
`18  7 ( 5 3 ) 2`	`Equation #1: 18=7+(5-3)*2`
`3 3 5  30`	`Equation #2: 3+3*5=30`
`18 3 3 5`	`Equation #3: 18-3=3*5`
`5 3 3`	`Equation #4: Impossible`
`0`	

4. [Problem inspired from an example in *Artificial Intelligence: A Modern Approach* by Russell and Norvig] The AI term *planning* refers to the selection of a *plan*—a sequence of *actions*—to achieve some *goal.* In a very simple version, the set of possible actions is finite and our goal is to achieve a *state,* which we describe as a set of named *conditions* from some finite set of conditions. We characterize each action by listing the *effects*—conditions that are achieved or removed by the action—and the *preconditions*—the conditions that must be present (or absent) before the action is possible. Your program's task is determine an appropriate plan for achieving a given goal.

For example, if our task involves moving two blocks (called 'A' and 'B'), our conditions might be called *A-On-Table* ("Block A is resting on the table"), *B-On-Table, A-On-B* ("Block A is resting on Block B"), and *B-On-A*, and our actions might be:

Action	**Pre-conditions**	**Effects**
Move-A-Table	*A-On-B*	*A-On-Table* $\wedge$ $\neg$*A-On-B*
Move-B-Table	*B-On-A*	*B-On-Table* $\wedge$ $\neg$*B-On-A*
Move-A-B	*A-On-Table B-On-Table*	*A-On-B* $\wedge$ $\neg$*A-On-Table*
Move-B-A	*A-On-Table B-On-Table*	*B-On-A* $\wedge$ $\neg$*B-On-Table*

The state of the world at any time consists of a set of conditions from the list above. An action is possible only if the state contains the positive conditions listed in the preconditions and does not contain the negated conditions (those preceded by $\neg$). The effect of an action is to add the positive conditions listed under "Effects" to the state (if they are not present) and remove the negated conditions (if they are present). We specify a task by giving the initial state and partially specifying the desired final state. For example, our initial state might be *A-On-B* $\wedge$ *B-On-Table* and our goal state might be *B-On-A.* In that case, a legal sequence of actions would be: *Move-A-Table, Move-B-A.* After *Move-A-Table,* the state would be *A-On-Table* $\wedge$ *B-On-Table,* and then after *Move-B-A,* the state would be *A-On-Table* $\wedge$ *B-On-A.* Since this state includes the desired condition *B-On-A,* the action sequence accomplishes our goal (the presence of other conditions in the goal doesn't matter). In this simple formalism, states and goals only include positive conditions ($\neg$A-On-B is never part of a state; it is used only in preconditions and effects to indicate required absence or removal of a condition, respectively).

The input to your program will consist of a series of problems in free format. Each problem starts with an initial state, consisting of a sequence of zero or more conditions separated by whitespace and terminated by a "word" consisting of a single semicolon (';'). Conditions are represented by strings of up to 20 non-whitespace characters that do not include ';' and do not start with '-'. Next comes a goal state with one or more conditions in the same form. This is followed by a non-negative integer numeral, N. Then come one or more action specifications, each of the form

action-name preconditions effects

where *action-name* is a word (same format as a condition) and both *preconditions* and *effects* have the same format as a state (sequence of zero or more conditions terminated by ';'), except that any of the conditions may be prefixed by -, indicating negation ($\neg$).

The last action specification is followed by an extra semicolon. End-of-file follows the last problem of the input.

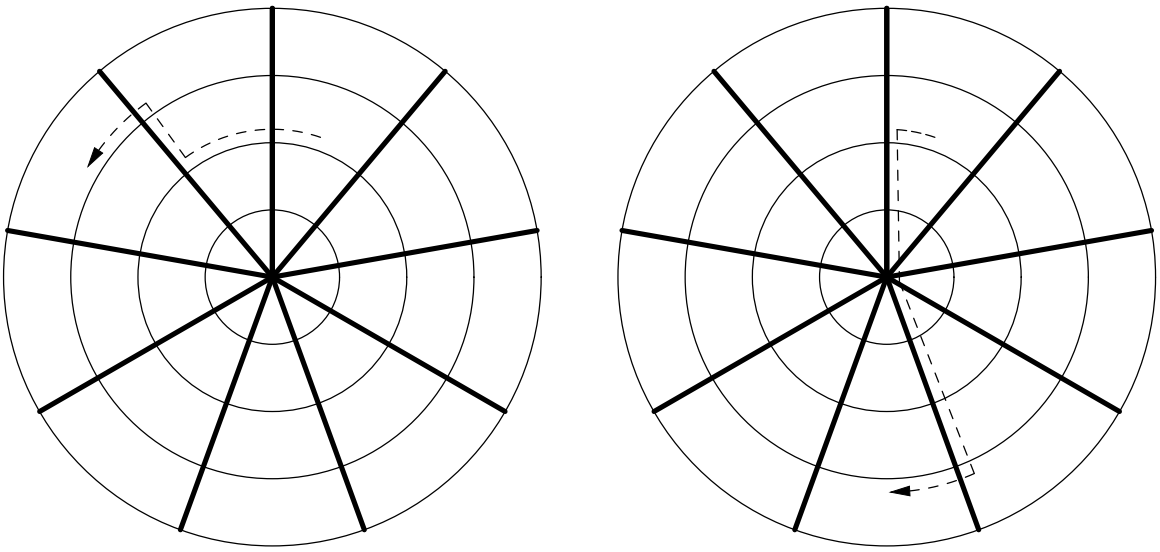
The output for a problem is a minimal-length sequence of no more than N actions that achieves the goal (on one line), or the string `-Impossible-` if no such plan exists. Use the format shown in the example.

Example.

Input	Output

`A-On-B B-On-Table ; B-On-A ; 5`
`Move-A-Table  A-On-B ;`
`  A-On-Table -A-On-B ;`
`Move-B-Table  B-On-A ;`
`  B-On-Table -B-On-A ;`
`Move-A-B  A-On-Table B-On-Table ;`
`  A-On-B -A-On-Table ;`
`Move-B-A  A-On-Table B-On-Table ;`
`  B-On-A -B-On-Table ;`
`;`
`A-On-B ; B-On-A ; 5`
`Move-A-Table  A-On-B ;`
`  A-On-Table -A-On-B ;`
`Move-B-Table  B-On-A ;`
`  B-On-Table -B-On-A ;`
`Move-A-B  A-On-Table B-On-Table ;`
`  A-On-B -A-On-Table ;`
`Move-B-A  A-On-Table B-On-Table ;`
`  B-On-A -B-On-Table ;`
`;`
`A-On-B B-On-Table ; B-On-A ; 1`
`Move-A-Table  A-On-B ;`
`  A-On-Table -A-On-B ;`
`Move-B-Table  B-On-A ;`
`  B-On-Table -B-On-A ;`
`Move-A-B  A-On-Table B-On-Table ;`
`  A-On-B -A-On-Table ;`
`Move-B-A  A-On-Table B-On-Table ;`
`  B-On-A -B-On-Table ;`
`;`
	`Problem 1: Move-A-Table Move-B-A`
	
	`Problem 2: -Impossible-`
	
	`Problem 3: -Impossible-`

5. Roads in the great city of Algopolis are laid out on a circular grid of evenly spaced concentric *circle roads* and a set of evenly spaced *spokes* radiating from their common center. Locations on this grid are identified by a *circle number*—a nonnegative integer identifying a circle, with 0 being the center point, 1 the smallest circle, 2 the next smallest, etc.—and an *azimuth*—an angle in degrees clockwise from due north indicating a position on the circle. The spokes are express roads, and we never have destinations or starting points along them other than at intersections. Here is such a grid with two possible routes through it.



A car driving on these roads may turn at any intersection onto any of the impinging roads (so that intersection in the center is pretty hairy, as you might guess). Your job is to write a program that, given the parameters of the road system and pairs of starting points and destinations, determines the shortest routes to these destinations. A *route* here is described as a list of words from the set `cw` and `ccw` ("proceed clockwise (counterclockwise) on your current circle road to the next intersection or your destination, whichever comes first"), and `in` and `out` ("proceed in(out)ward on the current spoke to the next intersection"). At the center, the `ccw` and `in` directions are not possible, and `cw` simply means to switch to the next spoke clockwise. On reaching the center, the "current spoke" is the one you came in on. Even if the spoke you use to leave the center is 180° from the spoke on which you entered, you must turn clockwise through all the spoke angles between your initial spoke and the outgoing spoke to get from one to the other, but these turns cover no distance. Always make the minimal number of turns at the center (no spinning around and around, in other words). The length of a route is the total distance along its line segments (spokes) and arcs (circle roads).

The input to your program will consist of a sequence of scenarios, each of which consists of five integers in free format: S, c_0, c_1, α_0, α_1. $S \geq 1$ is the number of evenly spaced spokes. There is always a spoke at azimuth angle 0 (due north). The rest of the numbers indicate a starting point and destination: c_0 and c_1, with $c_0 \geq 1$ and $c_1 \geq 0$, are the

numbers of the circles of start and destination; α_0 and α_1, $0 \leq \alpha_i < 360$, are the azimuth angles (in degrees) clockwise from due north of the start and destination. Circles are evenly spaced one unit apart. The center is "circle" 0; circle 1 is the first circle from the center; and so forth. The last scenario is followed by a line of five 0's.

The output consists of sequence numbers and lists of directions in the format shown in the example that follows. We will choose input data so as to ensure a unique answer.

Example.

Input	Output
`9 2 3 19 301`	`Case 1: ccw ccw out ccw`
`9 2 3 19 179`	`Case 2: ccw in in cw cw cw cw cw out out out cw`
`0 0 0 0 0`	

6. A *unified diff* file shows line-by-line differences between two files, which we'll call an *original* file and a *modified* file. For example, the following table shows two files, `f1` (original) and `f2` (modified), and a unified diff showing how to create `f2` from `f1`.

f1	**f2**	**diff**
The	The	--- f1
sanserif	sanserif	+++ f2
only seems	only seems	@@ -4,9 +4,10 @@
to be the	to be the	to be the
simpler	simpler	simpler
script.	script.	script.
It is a	It's a	-It is a
form that	form that	+It's a
was	was	form that
violently	most	was
reduced	violently	+most
for little	reduced	violently
children.	for little	reduced
For adults	children.	for little
it is more	For adults	@@ -17,7 +18,6 @@
difficult	it is more	to read
to read	difficult	than
than	to read	serifed
serifed	than	-roman
roman	serifed	type,
type,	type,	whose
whose	whose	serifs
serifs	serifs	
were never	were never	
meant to	meant to	
be	be	
ornamental.	ornamental.	

In the diff file, lines that begin with a blank are lines of context that appear in both files, lines beginning with a '`-`' appear only in `f1`, and lines beginning with '`+`' appear only in `f2`. Each group of such lines preceded by '`@`' is what we'll call a *hunk.* The *hunks* in a diff file apply to non-overlapping (and possibly widely separated) groups of lines in the files being compared. In actual diff files, the numbers on the '`@`' lines are significant, but in our simplified problem, you are to ignore them. In effect, the diff file is a set of instructions for converting `f1` into `f2`. It should be clear that from a given hunk, you can reconstruct the corresponding text in the original and modified files.

You are to write a "`patch`" program that takes an input file and a diff file and *applies* the diff file to the input, performing the same changes on the input file that changed the original to the modified file. The input file need not be the same as the original. For each hunk, the utility searches for a section of text in the input that *corresponds to* the hunk

and then, if that section matches the lines in the original exactly, replaces it with the lines in the modified file. If a given hunk does not correspond to anything in the input file, it is silently rejected, and has no effect. Likewise, if the hunk corresponds to something in the input file, but that section of the input does not exactly match the original, the hunk is rejected.

More precisely, a section of text in the input file corresponds to a hunk if

- It consists of complete lines from the input (no partial lines).
- It follows all lines that were successfully matched by preceding (non-rejected) hunks.
- It has the same number of lines as the text in the original (reconstructed from the hunk).
- The lines at the beginning of the hunk that are prefixed with a blank (if any) exactly match the first lines of the subsequence (i.e., once the blank is stripped off).
- The lines at the end of the hunk that are prefixed with a blank (if any) exactly match the last lines of the section of text.

When there is more than one section of text in the input that correspond to a hunk according to this definition, always select the first.

The input to your program consists of a unified diff file as described here. Any text before the first hunk (the first line starting with '`@`') are to be ignored. The diff file is followed by exactly one empty line (having no characters, not even blanks, before the end of line). That is followed by the file to be modified.

The output is the result of applying the patch to the file according to the rules above. The example on the next page illustrates the process. In it, the first hunk is rejected and the second is applied.

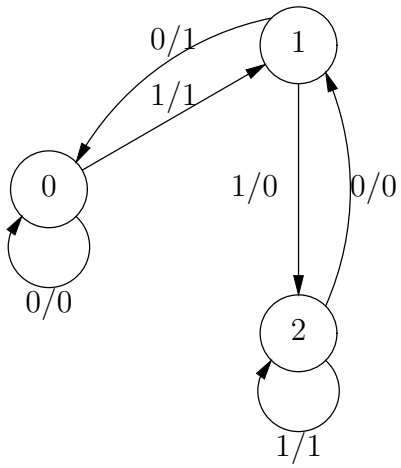
Input	Output

`--- f1`
`+++ f2`
`@@ -4,9 +4,10 @@`
` to be the`
` simpler`
` script.`
`-It is a`
`+It's a`
` form that`
` was`
`+most`
` violently`
` reduced`
` for little`
`@@ -17,7 +18,6 @@`
` to read`
` than`
` serifed`
`-roman`
` type,`
` whose`
` serifs`
 	
`The`
`sanserif`
`only seems`
`to be the`
`simpler`
`script.`
`'Tis a`
`form that`
`was`
`violently`
`reduced`
`for little`
`children.`
`Adults find it more difficult`
`to read`
`than`
`serifed`
`roman`
`type,`
`whose`
`serifs`
`were never`
`meant to be ornamental.`
	`The`
	`sanserif`
	`only seems`
	`to be the`
	`simpler`
	`script.`
	`'Tis a`
	`form that`
	`was`
	`violently`
	`reduced`
	`for little`
	`children.`
	`Adults find it more difficult`
	`to read`
	`than`
	`serifed`
	`type,`
	`whose`
	`serifs`
	`were never`
	`meant to be ornamental.`

7. A *Mealy machine* is a form of *finite-state machine* that produces output. Such a machine exists in one of a finite set of states (which we may identify by non-negative integer numbers), one of which is designated as its *initial state* (which will always be state 0 in this problem), in which it begins execution. The machine then proceeds to process an input consisting of a stream of input symbols from a finite *alphabet* (in this problem, the set $\{0, 1\}$) according to its *transition function,* which maps pairs consisting of current state and current input symbol to pairs consisting of a successor state (possibly the same) and output symbol (again, for our purposes, from the alphabet $\{0, 1\}$). Thus, by starting in its input state (0) and using the transition function to process each successive input symbol $(i_1, \ldots, i_n)$, the machine produces a string of output symbols $(o_1, \ldots, o_n)$.

For example, the machine below, given an n-digit binary numeral entered backwards (i.e., least-significant bit first), produces the least-significant n bits of the result of multiplying that number by 3 (also least-significant bit first):

State	**Input 0**		**Input 1**	
	Output	**Next**	**Output**	**Next**
0	0	0	1	1
1	1	0	0	2
2	0	1	1	2



The table on the left shows a 3-state machine. Each row displays a state number, output bit and next state to go to on an input of '0', and output bit and next state to go to on an input of '1'. The diagram on the right is a standard graphical representation of the same machine (where the notation '0/1', for example, means "follow this arrow on input of a 0, outputting a 1.") Given the input `0110100` (which is 22 in binary when read right-to-left), this machine produces the output `0100001` (66, when read right-to-left).

Your program is to read a series of integers in the range 0–1024 in free format, terminated by a single integer -1. For each integer, produce a table giving a finite-state machine for multiplying numbers by the input, in the format shown in the example on the next page (using the same meanings for the columns as in the table above). Precise spacing is not critical, but be sure to leave a blank line between sets of output. There are an infinite number of solutions for any given input, but the necessary number of states to multiply by k is $O(k)$.

Example.

Input	Output
3 1 -1	To multiply by 3: 0 0 0 1 1 1 1 0 0 2 2 0 1 1 2 To multiply by 1: 0 0 0 1 0

8. Whist is a game played by two pairs of partners with the usual deck of 52 playing cards, with thirteen ranks—(low) 2, 3,. . . , 10, Jack (J), Queen (Q), King (K), and (high) Ace (A)—and four suits—clubs (C), diamonds (D), hearts (H), and spades (S). The players sit on four sides of a table with partners opposite each other. We'll use the usual names from bridge and call these players South, West, North, and East. Each player gets thirteen cards, dealt clockwise around the table, with the dealer (South) getting the last card. The dealer shows this last card, and its suit becomes 'trump' for that hand.

The idea is for each partnership to try to win as many four-card *tricks* as possible. In each trick, each player, in clockwise order, plays one card from those remaining in his hand. After the first card of a trick, the remaining players must each play a card of the same suit as that of the first card, if they have one, and otherwise any card from their hand. The winner of a trick is the player who plays either the highest trump or the highest card with the same suit as the first card in the trick. For the first trick, West, on the dealer's left, goes first. The player who wins a trick plays the first card of the next. After all the tricks have been won, the partnership with the most tricks gets one point for every trick it won in excess of six.

For example, consider the following four hands:

```
West:   6C 10C  AC  5D  9D 10D  3H  7H  KH  6S 10S  KS  AS
North:  4C  5C  7C  KC  2D  3D  6D  JD  9H  5S  7S  JS  QS
East:   2C  8C  9C  JC  4D  AD  2H  4H  5H  6H  8H  2S  8S
South:  3C  QC  7D  8D  QD  KD 10H  JH  QH  AH  3S  4S  9S
```

and assume that the `3S` was dealt last, so that spades are trump. One possible play might proceed as shown in the table below. The left column shows which player leads on each trick. The middle shows the four cards played, starting with the lead. The right column shows who wins the trick. In this case, the East-West partnership wins eight tricks, and so scores two points. In tricks #9 and #11, a player wins a trick by trumping the lead.

Trick	Who leads	Cards played	Who wins trick
1.	West	`AC 5C JC QC`	West
2.	West	`6C 4C 8C 3C`	East
3.	East	`4D 7D 10D 2D`	West
4.	West	`AS JS 8S 9S`	West
5.	West	`KS QS 2S 3S`	West
6.	West	`6S 7S 6H 4S`	North
7.	North	`9H 5H AH 7H`	South
8.	South	`QD 5D 6D AD`	East
9.	East	`8H JH 3H 5S`	North
10.	North	`KC 9C KD 10C`	North
11.	North	`7C 2C 8D 10S`	West
12.	West	`KH 3D 2H 10H`	West
13.	West	`9D JD 4H QH`	North

Had South instead tried to take trick #1 by playing his 4S, it would have been illegal, since he could have "followed suit" and played a club.

You are to write a program that, given the sequence of cards played in a hand of whist, determines whether that sequence is valid according to the rules and, if so, who won and how many points they got.

The input consists of a number of hands—sequences of 53 card tokens in free format. A card token has the form RS, where R is a rank (2, 3,. . . , 10, J, Q, K, A), and S is a suit (S, H, D, C). Card tokens are separated by white space and contain no whitespace between R and S. The end of the file marks the end of the input. In each sequence, the first card is the one the dealer shows to determine trump (and which therefore the dealer must eventually play). The second card is then the first card played (by West), and likewise the subsequent cards are all the cards played, in sequence. After the first trick, your program must keep track of which player is playing each card, according to the rules.

The output consists of one line per hand, containing the hand number and either the message "`Illegal play`," "`North-South win` N `points`," or `East-West win` N `points`. Print the first message if either the rules are not followed or the sequence of cards is not a proper deck (contains duplicates) or a player other than the dealer plays the trump card. Use the format shown in the example below.

Example.

Input	Output
3S AC 5C JC QC 6C 4C 8C 3C 4D 7D 10D 2D AS JS 8S 9S KS QS 2S 3S 6S 7S 6H 4S 9H 5H AH 7H QD 5D 6D AD 8H JH 3H 5S KC 9C KD 10C 7C 2C 8D 10S KH 3D 2H 10H 9D JD 4H QH 3S AC 5C JC 4S 6C 4C 8C 3C 4D 7D 10D 2D AS JS 8S 9S KS QS 2S 3S 6S 7S 6H QC 9H 5H AH 7H QD 5D 6D AD 8H JH 3H 5S KC 9C KD 10C 7C 2C 8D 10S KH 3D 2H 10H 9D JD 4H QH	Hand 1: East-West win 2 points Hand 2: Illegal play