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

Programming Contest **P. N. Hilfinger**
Fall 2014

2014 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 8 problems on 14 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 (none this year), 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, for C++:

```
cstdlib cstdio climits stdarg.h cstring cctype iostream iomanip
string sstream vector list stack queue map set bitset algorithm cmath
```

and for C:

```
stdlib.h stdio.h limits.h strings.h stdarg.h ctype.h math.h>
```

In Java, you may use the standard packages `java.lang`, `java.io`, `java.math`, `java.text`, 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.

There are two ways to submit solutions: by a command-line program, and over the web. Submit from the command line on the instructional machines. 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.

To submit from the web, go to our contest announcement page:

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

and click on the "web interface" link. You will go to a page from which you can upload and submit files from your local computer (at home or in the labs). On this page, you can also find out your score, and look at error logs from failed submissions.

Regardless of the method you use for submission, your results are also mailed back to you at the account from which you submitted (in the case of web submission, that is the instructional account you used to validate yourself). Use the `https://imail.eecs.berkeley.edu` page to retrieve this mail.

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 10 seconds (on torus.cs). You will be advised by mail whether your submissions pass (use the imail account at

```
https://imail.eecs.berkeley.edu
```

and log in with the account you registered to use for the contest.) You can also view this information using the web interface described above.

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 -Iour-includes N.* -lm
```

For Java programs, it is equivalent to

```
javac -g -classpath .:our-classes N.java
```

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

```
java -cp .:our-classes PN
```

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 *our-libraries* and *our-packages* files and directories provide the additional tools we've provided this year. 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 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/index.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. You are probably familiar with the *Caesar cypher,* a simple substitution cypher in which each letter of the alphabet is encoded as the letter appearing a certain number of positions later in the alphabet (wrapping around from 'Z' to 'A'). For this problem, we will add blank (which we'll write as '␣' to make it visible) as one extra "letter" after 'Z'. For this extended alphabet, therefore, the cypher will instead wrap around from ␣ to 'A'. For example, shifting the message

```
GALLIA␣EST␣OMNIS␣DIVISA␣IN␣PARTES␣TRES
```

by five positions (so that 'A' becomes 'F' and '␣ becomes 'E'), we get

```
LFQQNFEJXYETRSNXEIN␣NXFENSEUFWYJXEYWJX
```

These cyphers are very easy to attack, even without extra information. Here, we consider the situation in which you want to decrypt an intercepted message and happen to know one of the words in it (a *known plaintext attack*). By "word," we mean a substring of the message that is bounded on both sides by a blank or an end of the string.

The input to your program will consist of an even number of lines of text, one pair of lines for each test case. The first line will contain the encrypted message, followed by a period. The second line will contain a single word known to be in the original (plaintext) message. Aside from the terminating period, all text will contain only upper-case letters and blanks.

For each pair of input lines, output one line, giving the translation of the encrypted input line (which will be unique in the datasets given), also followed by a period.

Example:

Input	Output
`LFQQNFEJXYETRSNXEIN␣NXFENSEUFWYJXEYWJX.`	`GALLIA␣EST␣OMNIS␣DIVISA␣IN␣PARTES␣TRES.`
`PARTES`	

2. [Suggested by Misha Dynin] A *numeral palindrome* (also known as a *palindromic number* or *Scheherazade number*), is a numeral—decimal in this problem—that is either 0 or reads the same from front to back and back to front and does not begin with the digit 0. For example, 314151413 is a numeral palindrome. You are to write a program to determine the count of numeral palindromes in a given range.

The input to your program will consist of one of more test cases in free format. Each case will consist of two integers, $0 \leq L < U < 10^{100}$. Unless it is 0, neither L nor U will start with the digit 0.

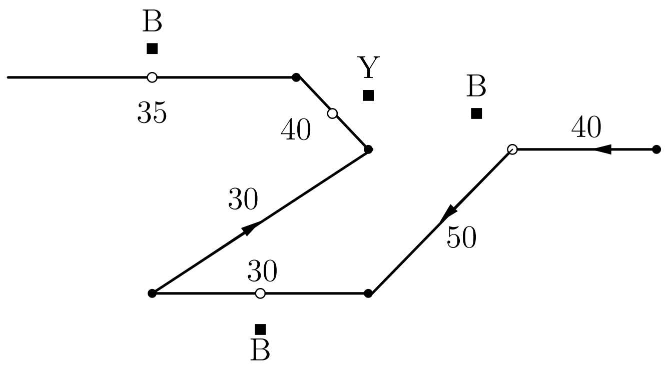
For each test case, print the sequence number of the set and the count of the number of numeral palindromes between L and $U - 1$ inclusive, using the format shown in the example. All counts will be less than 2^{60}.

Example:

Input	Output
`0 10 0 100`	`Case 1. 10`
`121 122  122 123`	`Case 2. 19`
`25 100`	`Case 3. 1`
	`Case 4. 0`
	`Case 5. 7`

3. You may have used one of those apps that predicts where and when busses will arrive near your location. This problem is a much-simplified version that involves a single bus route operating in a single direction. A route will consist of a sequence of straight line segments, each of which has an associated speed at which a bus travels along it. At a given time there may be one or more busses at various points along the route. Busses move along the route in one direction only. Bus stops are sufficiently common that we'll just assume that a bus may stop at any point along its route.

Given the route map, the positions of the busses, and your own position, compute the time(s) it will take busses along the route to arrive at your location along the route. To complicate the issue, the positions of both you and the busses are determined by a positioning system whose readings have a certain amount of error, so that neither you nor the busses may appear to be precisely on the route. As an approximation, you are to base your calculations on the points along the route that are nearest to your reported position and to those of the busses. In the example below, the B's mark the reported positions of the busses, Y marks your position, and the small open circles indicate the closest points on the route to these reported positions. The numbers along the route indicate speed limits.



The input will contain one route and set of positions in free format. Each route is described by $3N + 5$ integers:

- Two positive integers, N, giving the number of segments in the route, and B giving the number of busses.
- Two integers, x and y giving your reported position (according to your positioning device).
- Two integers x_0 and y_0 giving the starting point of the route (one endpoint of the first segment in the route).
- $3N$ integers—v_i, x_{i+1}, and y_{i+1}, for $0 \leq i \leq N$—each giving the speed limit and the x and y coordinates of the second endpoint of segment number i.
- $2B$ integers giving the reported x and y positions of each bus.

The output will consist of one line for each bus arrival at the point on the route nearest you, giving the time from when the positions were taken until that bus arrives. Busses that are already past your position are not listed. Round each time to the nearest integer value and report in increasing order.

Example:

Input	Output
`6 3`	`90`
`2500 -200`	`144`
`4500 -500`	
`40 3500 -500`	
`50 2500 -1500`	
`30 1000 -1500`	
`30 2500 -500`	
`40 2000 0`	
`35 0 0`	
`1000 200 1750 -1750 3250 -250`	

4. [Adapted from the Universidad de Valladolid's collection] An *Egyptian fraction* (the notation was developed in the Middle Kingdom of Egypt) is a sum of the form

$$1/d_0 + 1/d_1 + \ldots + 1/d_n$$

where the d_i are *distinct* positive integers. It turns out that any positive rational number may be represented in this form. For example,

$$\begin{aligned} \frac{2}{3} &= \frac{1}{2} + \frac{1}{6} \\ \frac{4}{5} &= \frac{1}{2} + \frac{1}{4} + \frac{1}{20} \\ \frac{3}{11} &= \frac{1}{4} + \frac{1}{44} \\ \frac{11}{12} &= \frac{1}{2} + \frac{1}{4} + \frac{1}{6} \end{aligned}$$

We can vary the problem by forbidding the use of certain denominators. For example, if the denominator 2 is forbidden, then

$$\begin{aligned} \frac{2}{3} &= \frac{1}{3} + \frac{1}{4} + \frac{1}{12} \\ \frac{11}{12} &= \frac{1}{2} + \frac{1}{3} + \frac{1}{12} \end{aligned}$$

Write a program to determine an Egyptian fraction expansion of given rational numbers, avoiding specified denominators.

The input to your program, in free format, will consist of one or more test cases. Each test case starts with three integers: $2 \le N < D \le 100$, $M \ge 0$, followed by M distinct positive integers greater than 1 indicating forbidden denominators, in ascending order. In each case, N/D is a fraction in lowest terms. You may assume that the product of the divisors in all solutions is less than 10^7 and no divisor is greater than 5000.

For each test case, your program should output a single line echoing the input and showing an expansion in the format used in the example. Arrange the fractions to the right of the '=' in ascending order. When there are multiple expansions possible, choose the one with the fewest fractions. When this leads to multiple solutions, choose the one with the smallest maximum denominator. If there are still multiple solutions, choose the one whose second-largest denominator is minimized, and so forth.

Example:

Input	Output
`2 3 0`	`Case 1: 2/3=1/2+1/6`
`19 45 0`	`Case 2: 19/45=1/5+1/6+1/18`
`2 3 1 2`	`Case 3: 2/3=1/3+1/4+1/12`
`5 121 0`	`Case 4: 5/121=1/33+1/121+1/363`
`5 121 1 33`	`Case 5: 5/121=1/45+1/55+1/1089`
`11 12 1 4`	`Case 6: 11/12=1/2+1/3+1/12`

5. [Steven & Felix Halim: Adapted from the Universidad de Valladolid's collection] "Beggar My Neighbor" is the name of a card game requiring no skill, and played with a standard deck of 52 cards. In this game, the four suits (spades, hearts, diamonds, and clubs) are ignored, and only the cards' ranks (ace, 2–10, Jack, Queen, and King) matter. The ace, Jack, Queen, and King are called *penalty cards.*

Both players get 26 cards face down. The game then proceeds in a series of rounds until one player (the loser) is unable to play a card when required to do so. It is not known whether this can go on forever, but none of the test data sets give rise to an infinite game.

A round proceeds as follows. The players alternately turn their top remaining card face up onto a pile on the table until one of the two players (call him player A) turns up a penalty card. At that point, the other player (B) is penalized by having to play a number of his cards onto the pile (also face up): one card if the penalty card was a Jack, two for a Queen, three for a King, and four for an ace. However, if one of these cards is itself a penalty card, the rest of the penalty is canceled and player A is penalized according to the same rule. Play continues until a penalty is completely paid. Then, whoever played the last penalty card turns the pile face-down and places it at the bottom of his hand. The next round begins with the winner of previous round (i.e., the player who took the pile) playing the first card. Again, play ceases at any point where a player is required to play, but has no cards.

Write a program to simulate a sequence of games of Beggar My Neighbor. The input will consist of a number of pairs of 26-card hands in free format, with the first hand of each pair going to the player who goes first. Each hand is presented top card first. A deck consists of 52 cards, with each card represented by two letters: the first for the suit—'C', 'D', 'H', or 'S'—and the second for the rank —the digits 2–9, 'T' for 10, 'A' for ace, 'J' for Jack, 'Q' for queen, or 'K' for King.

For each deck, output a single line giving the final number of cards in the winner's hand (i.e., the number of cards that are not on the pile).

Example:

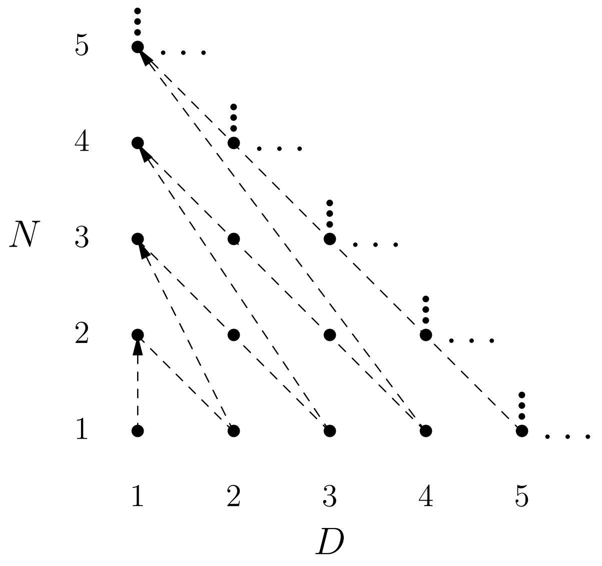
Input

```
H2 S7 SQ C3 CK DA D3 DK S4 H5 S3 C2 S8
CJ C9 HK S9 DT D7 HQ H7 C4 C5 SK H4 HA

D2 ST H8 HT CQ SA H9 D9 DQ DJ C7 S2 D8
C6 D6 C8 CT H6 SJ D4 HJ D5 S6 S5 CA H3

S2 D2 S3 D3 S4 D4 S5 D5 S6 D6 S7 D7 S8
D8 S9 D9 ST DT SJ DJ SQ DQ SK DK SA DA

H2 C2 H3 C3 H4 C4 H5 C5 H6 C6 H7 C7 H8
C8 H9 C9 HT CT HJ CJ HQ CQ HK CK HA CA
```

Output

```
44
0
```

6. Among other things, the mathematician Georg Cantor proved that the set of positive rational numbers is equipotent to (of the same size as) the set of positive integers. It's obvious that the fractions 1/1, 2/1, ... can be mapped onto the positive integers (so that there are at least as many rational numbers as positive integers). The other direction is more interesting. Cantor laid out the positive fractions N/D like this:



With this arrangement, we can line match up the positive integers to fractions by mapping 1 to 1/1, and then following the dashed line, so that 2 maps to 2/1, 3 to 1/2, 4 to 3/1, 5 to 2/2, 6 to 1/3, and so forth. (Of course, many rational numbers are repeated (e.g., 2/4 = 1/2), but that just means that the positive integers cover all the rationals many times, which doesn't hurt the argument).

You are to write a program that, given $1 < K \leq 10^{12}$, computes the fraction that integer matches up with, using this strategy. Input will consist of sequence of values of K in free format.

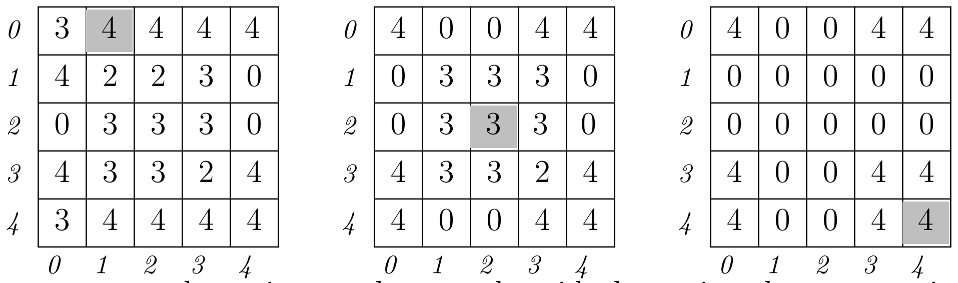
For each K, output a line that repeats the input and prints the corresponding fraction, using the format in the example below. (Careful: 10^{12} is larger than the largest Java **int**, so you'll have to use another type in your calculations.)

Example:

Input	Output
`6`	`6 => 1/3`
`1`	`1 => 1/1`
`30`	`30 => 7/2`
`1000000000`	`1000000000 => 6282/38440`

7. Cruller Locksmiths manufactures a puzzle lock in the shape of a donut. These particular donut locks are divided into 5 × 5 grids of touch-sensitive areas. For the purposes of this problem, we can think of one of these grids as an ordinary 5 × 5 square grid in which the top row of squares is treated as being adjacent to the bottom row, and the left column adjacent to the right column (so that going "up" from the top left corner takes you to the bottom left corner, going left from the top left corner takes you to the top right corner, and going diagonally up and left from the top left corner takes you to the bottom right corner). At any given time, each grid cell displays a number from 0–4, inclusive. Touching a grid cell increments this number by one (wrapping around to 0 if the number is 4), and also increments the eight adjacent squares by one. To solve the puzzle, you must push grid cells so as to get all the cells showing 0.

For example, given the initial grid on the left, pushing the shaded square gives the second grid; pushing the shaded square on that grid twice gives the third; and pressing the shaded square on the third grid gives all zeroes.



Write a program that, given such a puzzle grid, determines how many times to push each square (0–4 times) to get zero in all squares. The input will consist of a sequence of cases in free format. Each case consists of 25 integers in the range [0, 4].

The output consists of one grid for each test case, showing the number of times each square should be pushed displayed in the format illustrated in the example.

Example:

Input

```
3 4 4 4 4
4 2 2 3 0
0 3 3 3 0
4 3 3 2 4
3 4 4 4 4
```

Output

```
Case 1:
0 1 0 0 0
0 0 0 0 0
0 0 2 0 0
0 0 0 0 0
0 0 0 0 1
```

8. The Unix MAKE program—like other systems for managing the problem of building a large software systems—determines the order in which to perform specified tasks by the dependencies between them. So the line

```
flight-simulator: main.o physics.o display.o simulator.o
```

says (among other things) that building the target file `flight-simulator` requires that its *dependencies*—the files `main.o`, `physics.o`, etc.—be built (compiled) first. Unfortunately, modern languages such as Java don't fit this model quite as well, because of the possibility of circularities. Two Java classes may be mutually recursive, so that creating one class requires that the other class be built, and vice-versa. Fortunately, Java compilers can handle such situations by compiling all the files that participate in a circularity simultaneously. Your problem is to create a system that orders a set of tasks in the face of such circularities.

For example, given the dependencies

```
A : B C ;
C : A D ;
B : ;
D : ;
E : A ;
```

(the colons mean "depends on"), a valid ordering of tasks would be

```
B
D
A C
E
```

meaning "build B, then build D, then build A and C simultaneously, then build E."

The input will consist of a sequence of dependency declarations, one per line in free format. Each declaration has the form

$$T \;:\; D_1 D_2 \ldots D_n \;\;;$$

where $n \geq 0$. T and the D_i all consist of one or more characters other than ':', ';', or whitespace, and all are distinct. All the items in a declaration are separated by whitespace (including punctuation). Assume that each D_i appears as a target (a T) somewhere in the specification, and that each target appears exactly once.

The output will consist of a sequence of lines listing targets to be built simultaneously, separated by whitespace and in alphabetical order. The lists should always be minimal: if a target X does not have to be built simultaneously with Y, then they should not be listed on the same line. When lines that you output could validly be listed in any order, any of these orderings is acceptable.

Example:

Input	Output
`A : B C ;`	`B`
`C : A D ;`	`D`
`B : ;`	`A C`
`D : ;`	`E`
`E : A ;`	