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

Programming Contest **P. N. Hilfinger**
Fall 1998

1998 Programming Problems

To set up your account, execute

```
source ~ctest/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.)

This booklet should contain eight problems on 16 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` file and each complete C++ solution into a file N`.C` or N`.cc`, 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, except as indicated below. 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. Selected portions of the standard `g++` class library are included among of the standard libraries you may use: specifically, the headers `string`, `vector`, `iostream.h`, `iomanip.h`, and `fstream.h`. Likewise, you can use the standard C IO libraries (in either C or C++), and the math library (header `math.h`). 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`, N`.C`, or N`.cc`. 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 (if you get a message from `submit` saying that it failed, you will *not* be penalized). 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* `>` *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. 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 15 seconds. You will be advised by mail whether your submissions pass.

The command `contest-gcc` 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`.* -lstdc++ -lm`

The *our-includes* directory contains `contest.h`, 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 are free to use `scanf` to read in numbers and strings and `gets` to read in lines.

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.

Library. While you may not include code from other sources in electronic form, you may (if you find any use for it) use the code in `~ctest/include/elim.c`. To do so, simply place the line

```
#include "elim.c"
```

after the `#include` for `contest.h`. The principal routine in this file contains the header

```
/** Solve the equation Ax = B.  A is an NxN matrix; B is a column
 *  vector ([ B[0], ... B[N-1] ]).  The solution is placed in B,
 *  replacing its initial contents.  The contents of matrix A are
 *  modified; its final contents is undefined.   A is stored in
 *  row-major order: that is, its first row is in A[0..N-1], its
 *  second in A[N..2N-1], and so forth.  The result is undefined
 *  if A is singular or nearly so. */

void solve (double A[], double b[], int N)
```

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).

1. [With thanks to J. H. Conway] It is possible to express any rational number as some composition of the operations "add one" ($x \mapsto x+1$) and "negative reciprocal" ($x \mapsto -1/x$), starting with 0. For example,

$$\begin{aligned} 2 &= 0+1+1 \\ 1/2 &= -1/2+1 \\ -2 &= -1/(1/2) \\ -2/3 &= -1/(1/2+1) \\ 1/3 &= -2/3+1 \\ -3 &= -1/(1/3) \end{aligned}$$

Write a program that, given pairs of integers N and D, with $D > 0$, finds a sequence of these two operations that give N/D when applied to 0. This sequence is not unique. For example, since $0 = -1/(0+1)+1$, one can prefix a result with an arbitrary number of repetitions of the sequence "add 1, negative reciprocate, add 1."

The input to your program will be any number of pairs of integers (N and then D) in free form. For each pair, the output will echo the input, followed by a sequence of items, each of which has one of the forms

`+`N for N a positive integer with no leading zeroes, meaning "add N" (or rather "add 1 N times").

`/` meaning "take the negative reciprocal."

No two '+' items should be adjacent: output `+2` rather than `+1+1`. For example, the output '`+2/+1`' means "start with 0; add 2; take the negative reciprocal of the result; and add 1 to the result of that," or $-1/(0+2)\ +\ 1$. Use the format shown below.

Example:

Input	Output

`2 1`
`1 2 -2 1 -2 3`
`1 3 -3 1`
	`2/1: +2`
	`1/2: +2/+1`
	`-2/1: +2/+1/`
	`-2/3: +2/+2/`
	`1/3: +2/+2/+1`
	`-3/1: +2/+2/+1/`

2. A certain college professor posts his students' grades outside his door every semester. However, his university's privacy rules (not to mention his students' own feelings) require that he disguise the identities of the students in this list. Each student has a student ID, which is generally known only to the student, but the professor doesn't consider it wise to post these in undisguised form either. He hits instead on the following scheme: label each student's grade with just enough digits of the ID to identify the student uniquely. For example, if all IDs are five digits long and the students in the class have the names and IDs shown in the left two columns of the table below, then the professor might label the posted grades with the obfuscated IDs from the right column.

Name	ID	Obfuscated ID
H. Abelson	13795	xx7x5
J. Backus	12786	xxx8x
J. Conway	13595	xx5x5
E. Dijkstra	13695	xx6xx
L. Euler	13796	xx796
R. Feynman	13596	xx5x6

The idea here is that obfuscated IDs uniquely identify each student without revealing the entire ID. For example, only Abelson's ID has both a 7 in the third digit and a 5 in the last digit; only Backus's has an 8 in the fourth digit.

In general, there will be more than one possible obfuscation. However, in all cases, the digits shown (1) must suffice to identify the ID uniquely (out of all those in the class) and (2) must all be necessary. That is, it must not be possible to replace one of the digits in an obfuscated ID with an 'x' and still obey rule (1). In the example above, any valid obfuscation must have 'x' in all the first digits, since that digit is the same in all the IDs and therefore can never serve to differentiate them. Likewise, in the pathological case in which there is only one ID, it would necessarily be obfuscated to all 'x's.

The input to your program will consist of a positive integer, N, followed by any number of N-digit strings. All input is in free form. There is no limit on N, and strings may begin with the digit '0', so you are advised to input these IDs as strings, rather than as integers. You may assume that all the IDs are unique and are all N digits long.

The output of your program is to consist of the N-digit obfuscations of the input strings, one per line, in the same order as the input. There are sample inputs and outputs on the next page.

Example 1:

Input	Output
5 13795 12786 13595 13695 13796 13596	xx7x5 xxx8x xx5x5 xx6xx xx796 xx5x6

Example 2:

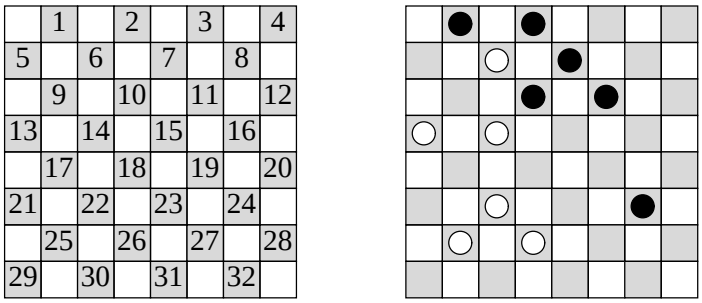
Input	Output
6 013795	xxxxxx

Example 3:

Input	Output
1 3 2 7 6 9	3 2 7 6 9

3. In the game of checkers (or *draughts* for some of you foreigners), one must always capture pieces if it is possible to do so, but when faced with more than one possible capture, one can choose any of the possible captures. A simple strategy, when faced with a choice of captures, is to capture so as to maximize the number of pieces you get over the number the opponent can capture on his next turn.

The game is played with black and white pieces on a chess board, using only the dark squares. For notational purposes, the squares are numbered as shown on the left:



We'll assume a simplified game in which there are no kings, so that the rules are as follows:

- The black pieces move down the board (toward larger numbers), the white pieces up.
- A piece A may capture (*jump*) a piece B of the opposite color, if B is on a (diagonally) adjacent square in the direction of A's motion *and* the square beyond B in the same direction is unoccupied. Piece A captures by jumping over B to the unoccupied square, and B is then removed from the board.
- If jumping over piece B causes A to land on a square from which another capture is possible, A must move again, capturing another piece. This continues until A moves to a square from which it cannot capture.
- When more than one piece of a given color can make a capture, exactly one of them captures; that same piece must continue capturing as long as it can.
- When a given piece has a choice of captures, it may make any one of them.

For example, suppose that it is Black's move in the configuration on the right above. Black has two possible jump sequences: either 2–9–18 (removing the two white pieces at 6 and 14 and moving the piece that was on 2 to 18) or 10–17 (removing the piece at 14 and moving the piece that was on 10 to 17). If Black chooses 2–9–18, White can respond with either 22–15–8 or 22–15–6. If Black chooses 10–17, White has no capture. So, if we look only at Black's move and White's next move, the 10–17 move gives the best gain (Black wins one piece, White none; when Black moves 2–9–18, on the other hand, White wins back two pieces, evening the score).

The input to your program will consist of a positive integer N_b followed by N_b integers in the range 1–32, giving the positions of the Black pieces, followed by a positive integer N_w and then N_w integers giving the positions of the white pieces. You may assume that the piece positions are all distinct and in the range 1–32. All input is in free form.

The output from your program should be the capture move sequence by Black that gives Black the greatest advantage (in relative numbers of pieces) after White's best next capture move. You may assume that Black will always have a capture move. Some moves by black may give White no capture to make, as in the example above, in which case Black's advantage for that move will simply be the number of pieces he captures. Otherwise, the advantage of a particular move by Black is the number of White pieces it captures, minus the largest number of Black pieces that White can capture from the resulting position. If there is more than one best move for Black, it doesn't matter which you report. The formats of input and output are illustrated below.

Example 1:

Input	Output

`6 1 2`
`7 10 11`
`24`
`6 6`
`13 14 22 25 26`
	`Best move sequence is 10-17`

Example 2:

Input	Output

`7 1 2 15`
`7 10 11`
`24`
`6 6`
`13 14 22 25 26`
	`Best move sequence is 2-9-18`

4. The **javadoc** program extracts descriptive comments and headers from a Java program and presents it with all the actual code stripped away. In this problem, you are to do a much simplified version of this task. The example on the next page shows an example, giving the format desired for the output.

The input consists of *documentation comments*—which begin with '`/**`' and end with '`*/`'—preceding a single class header and subsequent method declarations. You may assume that these comments always precede the class headers and methods, and are on separate lines from them (as in the example). You may assume there are no other documentation comments. In particular, field definitions are *not* so commented, and are *not* to be included in your output. Likewise, there are no documentation comments inside any of the method bodies. Each class header begins with `class` *class-name*, on the line following a documentation comment. There may be other things on the line, which are irrelevant. Each method header (again starting on a line after the end of a documentation comment) has one of the formats

$$T_0 \;\; N_0 \;\; (T_1 \;\; N_1, \cdots, T_k \;\; N_k)$$

$$N_0 \;\; (T_1 \;\; N_1, \cdots, T_k \;\; N_k)$$

where $k \geq 0$, the N_i are identifiers (containing letters, digits, and underscores, starting with a non-digit), and the T_i are types (which are the same as identifiers, except that they may contain the characters '`[`' and '`]`'). Each T_i is separated from N_i by whitespace, and there may be additional whitespace as well. You may assume that no line is longer than 256 characters, and that none of the N_i and T_i contain more than 32 characters. You may assume that there are no blanks in T_i (that is, you will see `int[]`, not `int []`.

As illustrated in the example on the next page, your program is to throw away all other portions of the input (method bodies, fields, comments that don't start with "`/**`"), and re-arrange what's left. For documentation comments, strip away the '`/**`' and '`*/`' and all leading whitespace and asterisks, but preserve everything else (including newlines). The output consists of three sections, introduced as shown in the example: the class followed by its documentation comment (indented two spaces), a "Method Synopsis" containing the headers without return types or parameter names (spaced as shown) and the first sentence (up to and including the first period) of each method's documentation comment, and a detailed "Methods" section containing the complete headers and documentation comments. Space and punctuate everything as shown.

Example:

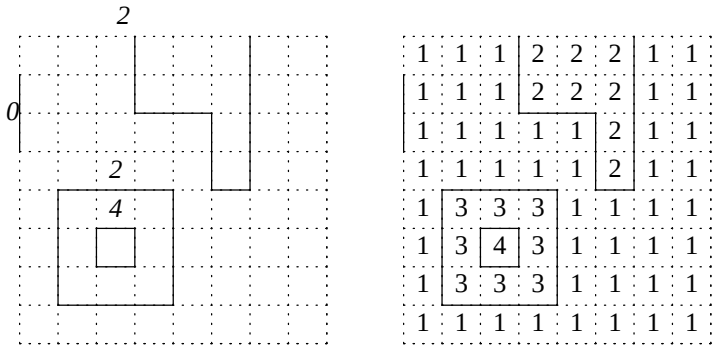
Input file

```
/** A linked-list cell.  Operations pro-
 *  vide for insertion and deletion. */
class ListCell {


   /** A new list with given HEAD
       and TAIL. */
   ListCell (
         Object head,ListCell tail) {
      this.head = head; this.tail = tail;
   }
   /** Insert list cell X after THIS.
    Requires X != null, and X.tail == null. */
   void insert(ListCell X) {
      X.tail = tail; tail = X;
   }



   /** Delete the list cell after THIS. Return
    *  the deleted cell and set its tail
    *  to null. Requires tail != null. */
   ListCell removeNext() throws NullPointerException {
      ListCell old = tail;
      tail = old.tail;
      old.tail = null;
      return old; }

   /* Data for this ListCell cell. */
   public Object head;
   /* Remainder of list */
   public ListCell tail;
}
```

Output file

```
Class ListCell:
  A linked-list cell.  Operations pro-
  vide for insertion and deletion.

Synopsis:
+ ListCell (Object, ListCell)
    A new list with given HEAD
    and TAIL.
+ insert (ListCell)
    Insert list cell X after THIS.
+ removeNext ()
    Delete the list cell after THIS.

Methods:
+ ListCell (Object head, ListCell tail)
    A new list with given HEAD
    and TAIL.

+ void insert (ListCell X)
    Insert list cell X after THIS.
    Requires X != null, and X.tail == null.

+ ListCell removeNext ()
    Delete the list cell after THIS. Return
    the deleted cell and set its tail
    to null. Requires tail != null.
```

5. The figure on the left below is what one might call a *Manhattan contour map* on a rectangular region:



Each of the contour lines is labeled with the value of whatever is being mapped (temperature, air pressure, etc.) along that line. I call it a Manhattan map because all contour lines are laid out on a square grid, as shown; each segment must begin and end at a grid point.

For this problem, I'd like to reconstruct values to go in the grid squares, as shown in the figure on the right. I determined what number to put in any given square of the grid by looking at all the contour lines reachable by starting from the center of the square and moving one square at a time left, right, up, or down, never crossing a contour line. I then look for any contour lines bordering the squares reached in this way. If all such contour lines have the same value, that is the value that goes in the grid square (e.g., the squares labeled 2 and 4 in the figure on the right). If there are two different values reachable in this way, we use the average of the two (e.g., the squares labeled 1 and 3 in the figure on the right). These maps don't "skip" contours: you may assume that there are never more than two contour values reachable from a given square. We will use even integer values for all contour-line values, so that integers will suffice for all squares' values.

The input to your program will be in free form. First, there will be two integers, W and H, giving the number of squares horizontally and vertically. The (dotted) vertical grid lines are numbered from 0 to W (0 on the left) and the horizontal grid lines are numbered from 0 to H (0 at the top). Next, there will be a set of contour-line segment specifications, each having either the form

V X Y N *or*

H X Y N

The first form (starting with V) denotes a one-unit vertical segment along vertical grid line X ($0 \leq X \leq W$). between horizontal grid lines Y and $Y+1$ ($0 \leq Y < H$). The second form (starting with H) denotes a one-unit horizontal segment along horizontal grid line Y ($0 \leq Y \leq H$). between vertical grid lines X and $X+1$ ($0 \leq X < W$). In either case, $N \geq 0$ is an even integer indicating the value of that contour line segment.

The output should be an array of integers giving values for all the $W \times H$ squares, in the format illustrated on the next page.

Example:

Input	Output
8 8 V 0 1 0 V 0 2 0 V 3 0 2 V 3 1 2 H 3 2 2 H 4 2 2 V 5 2 2 V 5 3 2 H 5 4 2 V 6 0 2 V 6 1 2 V 6 2 2 V 6 3 2 H 1 4 2 H 2 4 2 H 3 4 2 H 1 7 2 H 2 7 2 H 3 7 2 V 1 4 2 V 1 5 2 V 1 6 2 V 4 4 2 V 4 5 2 V 4 6 2 V 2 5 4 V 3 5 4 H 2 5 4 H 2 6 4	1 1 1 2 2 2 1 1 1 1 1 2 2 2 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 2 1 1 1 3 3 3 1 1 1 1 1 3 4 3 1 1 1 1 1 3 3 3 1 1 1 1 1 1 1 1 1 1 1 1

6. A certain primitive text-formatting system allows one to write superscripts and subscripts in one's text as follows:

```
In this equation, we may substitute for the
derivative to get x_{n+1} = x_{n} + (y-x_{n}^{2})/2x_{n},
which solves the problem.  For the cube root,
the corresponding equation is
    x_{n+1} = x_{n} + (y-x_{n}^{3})/3x^{2}_{n}.
```

and have them converted as follows:

```
In this equation, we may substitute for the
                                    2
derivative to get x    = x  + (y-x )/2x ,
                  n+1    n       n    n
which solves the problem.  For the cube root,
the corresponding equation is
                   3    2
    x    = x  + (y-x )/3x .
     n+1    n       n    n
```

Each character of regular text can have a subscript or superscript (or both). The formatter adds an extra line above any line containing superscripts and below any line containing subscripts, and places the text between the curly braces at the appropriate places on these lines. Use separate sub- and superscript lines for each line that needs them (that is, don't put superscripts from one line on the line containing subscripts from the one above it). Input text to the right of the sub- or superscripted component gets moved to so that it begins just to the right of the subscript or superscript on the preceding text, whichever is longer. For example,

```
                          3
x_{3}^{k+1}+1  becomes   x    +1
                          k+1
```

Write a program that performs these transformations. The input consists of a text file with subscript and superscript annotations. You may assume that the text of subscripts and superscripts does not contain subscripts, superscripts, newlines, or curly braces, and that there is at most one subscript and one superscript for each regular character (so '`x_{n}_{y}`' is illegal). The input will always be well formed: each underscore or caret is followed by a left curly brace, followed by text that is bracketed by a right curly brace.

Aside from moving around the subscripts and superscripts as shown, preserve all blanks and newlines. There will be no tabs in the input.

7. Optimizing compilers perform various transformations on programs to make them faster, one of which is known as *dead-variable elimination.* Consider the following C function and its re-write on the right, for example (which you would never write, of course, since it is contrived and contains gratuitous **goto** statements as well):

```
   int morbid (int a, int c) { |   int morbid (int a, int c) {
      int b, x, y;             |      int b, x, y;
      b = a * 3;  /* 1 */      |
      x = b + 1;  /* 2 */      |
      y = a - 2;  /* 3 */      |      y = a - 2;
 L4:  if (a > 3) goto L7;      |      if (a > 3) goto L7;
      x = 5;      /* 5 */      |
      goto L8;                 |      goto L8;
 L7:  y = 7;                   | L7:  y = 7;
 L8:  x = a + y;  /* 8 */      | L8:  x = a + y;
      x = x+1;    /* 9 */      |      x = x+1;
      c = c - 1;               |      c = c - 1;
      if (c > a) goto L4;      |      if (c > a) goto L4;
      return x + y;            |      return x + y;
   }                           |   }
```

The variable of `x` is said to be *dead* immediately after the assignments (2) and (5). That is, no matter which way all the **if** tests turn out, the value of `x` assigned in statements (2) and (5) will never be used before either it is wiped out by another assignment to `x`, or the function returns (in this case, statement (8) always assigns to `x` first). We say that therefore, statements (2) and (5) are *dead-variable assignments* and may be eliminated. Once statement (2) is gone, there is no use of `b` anywhere before the end of the function, so statement (1) can go, too. On the other hand, since the value of `y` assigned at (3) *may* be used at (8) (if `a<=3`), we do not eliminate it. Likewise, the value of `x` assigned at (8) will be used at (9), and therefore statement (8) cannot be eliminated (in an assignment statement, the variables on the right are "used" before the assignment happens). The value assigned to `x` at (9) may be used in the **return** statement (if `c<=a`). For our purposes here, we will assume that an **if** statement can go either way, even if its condition is something like `(c == c)`.

The input to your program will use a simplified version of C, consisting of one or more statements in free form, each having the one of the following formats:

var `=` *use* `+` *use*

`if` *use* `>` *use* `goto` *num*

`goto` *num*

`return` *use* `+` *use*

where a *var* is a single lower-case letter (a–z), *use* is either a *var* or a single digit, and *num* is a non-negative statement number. There is whitespace around all of the tokens, to make life simple.

There are no declarations or function headers, and we use + and > to stand for all operators (since for our purposes, it doesn't matter what specific operation is performed or what specific integer constant is used). There will be only one `return` statement, and it will be the last statement. We dispense with C statement labels and just use numbers; statements are implicitly numbered starting at 1.

Your program is to print "`Eliminate`" followed by a list of statement numbers (in ascending order on one line) that may be eliminated according to the criteria described above. If nothing can be eliminated, the output will simply have the word "`Eliminate`." Possible input and output for the preceding example is illustrated below.

Example:

Input	Output

`b = a + 3`
`x = b + 1`
`y = a + 2`
`if a > 3 goto 7`
`x = 5 + 0`
`goto 8`
`y = 7 + 0`
`x = a + y`
`x = x + 1`
`c = c + 1`
`if c > a goto 4`
`return x + y`
	`Eliminate 1 2 5`

8. An ambitious, but not terribly bright computer cracker was poking around at a site he believed to be the mysterious and infamous N.O.D. (National Obscurity Department) when he found a tempting description of an encryption algorithm "just lying around" in a text file. He decided to use it to keep his own communications secure. This is a great break for you—who have been asked to monitor his activities—because this particular encryption algorithm is extremely vulnerable to *known plaintext attacks*. That is, if you happen to see some encrypted messages and happen to know (or guess) what messages they encrypt, you can find the encryption key, and thus read the rest of his communication. (The N.O.D. *wouldn't* have arranged this on *purpose,* would it?)

The algorithm assumes that all communication uses ordinary ASCII characters—numbers in the range 1–127. It encodes blocks of five characters at a time. The parties to a communication choose a *key* consisting of nine characters, each of which is also in the range 1–127. Suppose that the key is $k_0 \cdots k_8$. If $p_0p_1p_2p_3p_4$ is a five-character block to be encoded, then we encrypt it into a sequence of five integers, $e_0, \ldots, e_4$, according to the definition

$$e_i = p_0 \cdot k_i + p_1 \cdot k_{i+1} + p_2 \cdot k_{i+2} + p_3 \cdot k_{i+3} + p_4 \cdot k_{i+4}.$$

That is, the algorithm is given a nine-character key, and then for each five characters of the input, it outputs five integers. It happens that if you know the plaintext (the characters p_i) and the corresponding integers e_i for two distinct messages, you can deduce the key that was used ($k_0 \cdots k_8$) quite quickly.

The input to your cracker-cracking program will consist of a string of at least ten characters of known plaintext (that is, at least two blocks of five characters) all alone on one line, followed by ten integers (the e values that result from encoding the first block of five characters and then the e values that result from encoding the second block). Any characters of the first line beyond the first ten are irrelevant. From these data you are to deduce the key (the $k_0 \cdots k_8$) that was used to do the encryption. The rest of the input will consist of a sequence of integers that are the encryption of an unknown message using the same key. There will be a multiple of five of these integers. All integer input will be free form.

The output should be simply the decrypted text. *Be sure* to add a newline to the very end. In the following example, the encryption key happens to be "`R0NR1VeSt`":

Example:

Input	Output

`John Q. Crackjack`
`30096 28880 32662 34217 36518`
`22426 23187 28934 27877 29942`
 	
`34391 35683 39956 40657 45237`
`33044 30700 34923 37133 38330`
`37200 37645 43055 43698 47491`
`32574 30389 35225 36978 38050`
	`Squeamish ossifrage.`