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

Programming Contest **P. N. Hilfinger**
Fall 2008

2008 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 19 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.

This year, for the first time, we have 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. One problem with this interface: at the moment it does not, unlike `submit`, pre-test your submission. At the moment, you can only do this on the instructional machines.

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 (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 -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. [From the Valladolid archives] An imaging device furnishes digital images of two machined surfaces that eventually will be assembled in contact with each other. The roughness of this final contact is to be estimated.

A digital image is composed of the two characters, '`X`' (marking places where material is present) and '␣' (blank, indicating space). The first column will always have an '`X`' in it and will be part of the left surface. The left surface can extend to the right as contiguous `X`s. Similarly, column N will always have an '`X`' in it and will be part of the right surface, where N is furthest right position of any '`X`' over all rows. The right surface can extend to the left from column N as contiguous `X`s. Both surfaces will have the same vertical extent (the same number of rows), so that there is a section of left surface and right surface on each row of the image.

For example, here is a possible image (with blanks shown as '␣':

```
XXXX␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣XXXXX

XXX␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣XXXXXXX

XXXXX␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣XXXX
```

Left Surface **Right Surface**

```
XX␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣XXXXXX

XXXX␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣XXXX

XXX␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣XXXXXX
```

In this example, each line is 30 characters long

In each row of the image, there can be zero or more blanks separating the left surface from the right surface. There will never be more than a single blank region in any row.

For each image given, you are to determine the total empty area—the total number of blanks between the surfaces—that will exist after the left surface has been brought into contact with the right surface. The two surfaces are brought into contact by displacing them strictly horizontally towards each other until a rightmost '`X`' of the left surface of some row is immediately to the left of the leftmost '`X`' of the right surface of that row. There is no rotation or twisting of these two surfaces as they are brought into contact; they remain rigid, and only move horizontally.

The input consists of a series of digital images. Each image consists of one or more lines of `X`s and blanks, all having the same length and all beginning and ending with `X`. There will be an empty line after every data set.

For each image in the series, output the total empty area, using the format shown in the example.

Example:

Input	Output
XXXX␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣XXXXX XXX␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣XXXXXXX XXXXX␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣XXXX XX␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣XXXXXX XXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXX␣␣␣␣␣␣␣␣␣␣␣␣␣XX	Image 1: 4 Image 2: 0 Image 3: 0

2. Using what is essentially long division, one can divide one polynomial by another just as can divide one integer by another to get a quotient and remainder. Thus we may compute

$$\frac{2x^4 - x^3 + 3x^2 - x + 3}{2x^2 + 1}$$

as follows:

$$\begin{array}{rr|lllll} & & x^2 & -\frac{1}{2}x & +1 & & \\ \hline 2x^2+1 & & 2x^4 & -x^3 & +3x^2 & -x & +3 \\ & & 2x^4 & & +x^2 & & \\ \hline & & & -x^3 & 2x^2 & -x & +3 \\ & & & -x^3 & & -\frac{1}{2}x & \\ \hline & & & & 2x^2 & -\frac{1}{2}x & +3 \\ & & & & 2x^2 & & +1 \\ \hline & & & & & -\frac{1}{2}x & +2 \end{array}$$

so that

$$\frac{2x^4 - x^3 + 3x^2 - x + 2}{2x^2 + 1} = x^2 - \frac{1}{2}x + 1, \text{ with a remainder of } -\frac{1}{2}x + 2.$$

As a check,

$$2x^4 - x^3 + 3x^2 - x + 3 = (2x^2 + 1)(x^2 - \frac{1}{2}x + 1) - \frac{1}{2}x + 2.$$

As a slight variation, the coefficients of the polynomial don't have to be real numbers; they can come from a finite field, such as GF(5), a fancy name for the integers modulo 5. In this field,

$$\frac{2x^4 - x^3 + 3x^2 - x + 3}{2x^2 + 1} = x^2 + 2x + 1, \text{ with a remainder of } 2x + 2,$$

since with mod 5 arithmetic,

$$2x^4 - x^3 + 3x^2 - x + 3 = 2x^4 + 4x^3 + 3x^2 + 4x + 2 = (2x^2 + 1)(x^2 + 2x + 1) + 2x + 2.$$

As another example, over the same field,

$$\frac{x^6 - 4}{x - 2} = x^5 + 2x^4 + 4x^3 + 3x^2 + x + 2$$

with a remainder of 0 (as you can see, the exponents of x are just ordinary integers, and not mod 5 like the coefficients).

Write a program that computes quotients and remainders of polynomials over GF(p) for any prime p (these aren't the only finite fields, but let's keep it simple). The input to your program will consist of a number of sets of integers in free format. Each set will start with a positive integer, $p \leq 1000003$, which will always be prime. Next will come two

integers M and N, indicating the degrees of the two polynomials to be divided. Finally will come $M + 1$ integers in the range 0 to $p - 1$ giving the coefficients of the dividend starting with the constant term, followed by $N + 1$ integers (also in the range 0 to $p - 1$) giving the coefficients of the divisor, starting with the constant term. The coefficients of the high-order (last input) term of each polynomial will be non-zero.

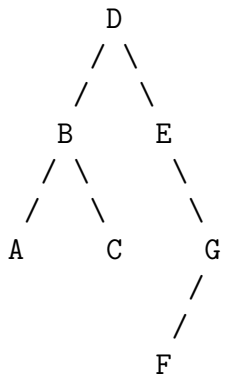
The output will echo the dividend and divisor and give the quotient and remainder using the format shown in the example below. Be careful about the details illustrated in the output. The 0 polynomial prints as '0'. Otherwise, suppress terms with coefficients of 0, the '+' sign on the high-order term, and remainders of 0.

Example:

Input	Output

`5`
`4 3 4 3 4 2`
`2 1 0 2`
 	
`2`
`4 1 1 0 1 1`
`2 1 1 1`
 	
`3  0 1  0 2`
	`Case #1: 2x^4+4x^3+3x^2+4x^1+3 / 2x^2+1 = 1x^2+2x^1+1 remainder 2x^1+2 mod 5`
	
	`Case #2: 1x^4+1x^3+1x^1+1 / 1x^2+1x^1+1 = 1x^2+1 mod 2`
	
	`Case #3: 1 / 2 = 2 mod 3`

3. [Adapted from the Valladolid archives] Consider binary trees whose nodes are labeled with single digits and letters (upper- and lowercase, case-sensitive). Given the preorder (root, left subtree, right subtree) node order and the inorder (left subtree, root, right subtree) node order for the same tree, it is possible to reconstruct the tree, assuming no two nodes in the tree have the same label. For example, given the preorder traversal order "DBACEGF" and the inorder traversal order "ABCDEFG," you can compute that the tree these come from is



You are to write a program to do this reconstruction on any such tree.

The input consists of one or more cases in free format. Each case consists of two non-empty strings *PRE* and *IN*, representing the preorder traversal and inorder traversal of a non-empty binary tree, and consisting of letters and digits. Case is significant.

For each test case, print the reconstructed tree's postorder traversal (left subtree, right subtree, root), using the format shown in the example.

Example:

Input	Output

`DBACEGF`
`ABCDEFG`
`BCAD CBAD`
	`Case 1: ACBFGED`
	
	`Case 2: CDAB`

4. Most of the inhabitants of the as-yet undiscovered planet Alpha Centauri B III worship the god V'tr'i, a rather volatile sort of diety. On any given day, this god, so the priests say, has one of five moods: content, vengeful, whimsical, joyous, or angry. The weather on a given day depends upon his mood that day, but not in an entirely predictable way. When he is content, for example, there can be sun, clouds, or rain. When angry, there can be rain, snow, or sleet. And so forth. His mood varies from day to day, with the mood on any given day influencing the mood on the next day. The Priesthood of V'tr'i is extremely meritocratic, with priests gaining status depending on how successful they are in keeping their god reasonably happy. Therefore, there is considerable interest in deducing the god's moods during the periods when a given priest is given charge of the sacred rites. Your task is to aid the Council of V'tr'i in its periodic evaluations by giving them a tool to guess retrospectively at the god's moods, specifically a program that will read in a history of the weather for a sequence of days and produce as output the sequence of the god's moods that would most likely account for it.

The priests believe they know the likelihoods of each possible weather condition, given each possible mood. They also believe they know the likelihood that the god will be in any given mood, given the mood he was in the previous day. However, priests, as always, have their factions, which generally possess differing ideas as to what these likelihoods might be. Each faction has sacred tables with entries giving the percentage probabilities of weather conditions and subsequent moods given the day's mood, like this:

	Weather					**Next Mood**				
Mood	**Sun**	**Clouds**	**Rain**	**Sleet**	**Snow**	**Joy.**	**Cont.**	**Whim.**	**Ang.**	**Ven.**
Joyous	60	20	10	5	5	40	40	10	10	0
Content	30	30	20	10	10	30	30	20	10	10
Whimsical	20	20	20	20	20	20	20	20	20	20
Angry	10	10	30	20	30	10	10	20	30	30
Vengeful	0	10	30	40	20	0	20	40	30	10

This particular table tells us, for example, that when joyous, V'tr'i has a probability of 0.6 (60%) of sending sun, and a probability of 0.1 (10%) of being angry the next day. If we think that on a particular day, the god has an equal (20%) probability of having any particular mood, then we initially expect that the probability that there will be sun on Monday and rain on Tuesday, and that V'tr'i will be joyous on both days is

$$\begin{aligned} & P(\text{initially joyous}) \times P(\text{sun when joyous}) \times P(\text{joyous next day when joyous today}) \\ & \times P(\text{rain when joyous}) \\ = \; & 0.2 \times 0.6 \times 0.4 \times 0.1 \\ = \; & 0.0048. \end{aligned}$$

and that the probability of having the same weather (sunny then rainy) and that the god will be joyous on Monday and content on Tuesday is $0.2 \times 0.6 \times 0.4 \times 0.2 = 0.0096$. If the weather is indeed sunny on Monday and rainy on Tuesday, the latter sequence of moods (joyous, then content) turns out to be the most likely.

The input to your program, in free format, will begin with the 50 non-negative integer percentages from such a table in the order shown in the example above. Next will come five integers giving a guess as to the initial probabilities on the first day of a priest's turn at officiating that the god is in each of his five possible moods, again in the order used in the table above. Finally, comes a sequence of words from the set { `sun`, `clouds`, `rain`, `sleet`, `snow` }, giving the weather on each of the days being considered.

The output will consist of a sequence of words from the set { `joyous`, `content`, `whimsical`, `angry`, `vengeful` }, giving the most likely corresponding sequence of moods for the god on those days. Use the format in the example below.

Example:

Input	Output

`60 20 10 5 5 40 40 10 10 0`
`30 30 20 10 10 30 30 20 10 10`
`20 20 20 20 20 20 20 20 20 20`
`10 10 30 20 30 10 10 20 30 30`
`0 10 30 40 20 0 20 40 30 10`
`20 20 20 20 20`
`sun rain`
	`joyous content`

5. [From the Valladolid archives] There are many different types of electoral system. In a *block voting system,* the members of a party do not vote individually as they like, but instead must collectively accept or reject a proposal. Although a party with many votes clearly has more power than a party with few votes, the votes of a small party can nevertheless be crucial when they are needed to obtain a majority. Consider for example the following five-party system:

Party	# of votes
A	7
B	4
C	2
D	6
E	6

Coalition $\{A, B\}$ has $7 + 4 = 11$ votes, which is not a majority. When party C joins coalition $\{A, B\}$, however, $\{A, B, C\}$ becomes a winning coalition with $7 + 4 + 2 = 13$ votes. So even though C is a small party, it can play an important role.

As a measure of a party's power in a block voting system, John F. Banzhaf III proposed to use the *power index.* The key idea is that a party's power is determined by the number of minority coalitions that it can join and turn into (winning) majority coalitions. A coalition forms a majority by having more than half the total votes. The empty coalition, therefore, is a minority coalition. In the example just given, a majority coalition must have at least 13 votes.

In an ideal system, a party's power index is proportional to the number of members of that party. Your task is to write a program that, given an input as shown above, checks compliance with this ideal by computing the power index of each party.

The input consists of a number of cases in free format, for each of which your program is to produce a report. Each case begins with an integer P, $1 \leq P \leq 20$, giving the number of parties for that case. This is followed by P positive integers, giving the numbers of members of each of the parties. No electoral system will have more than 1000 votes.

For each case, you must generate one line of output for each party, giving its power index. Use the format shown in the example on the next page.

Example:

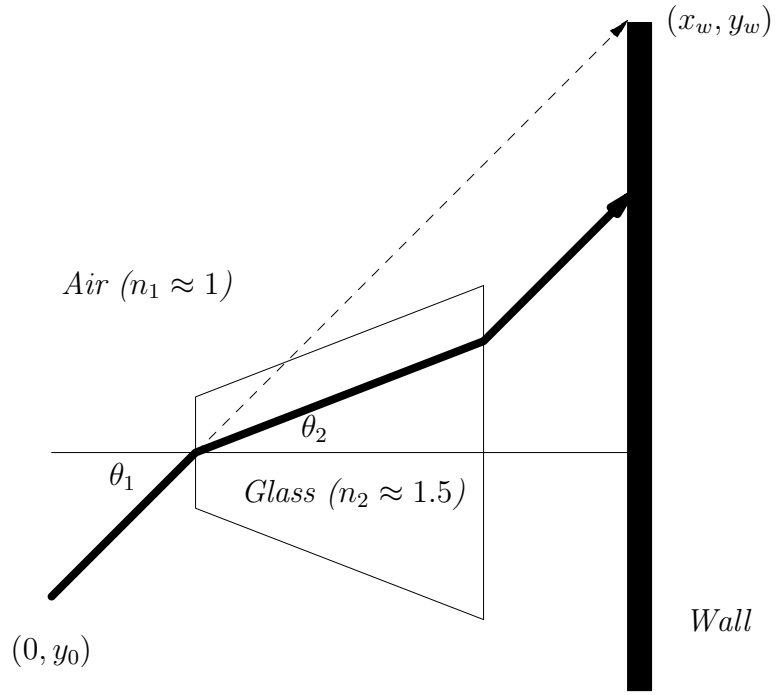
Input	Output

5 7 4 2 6 6
6 12 9 7 3 1 1
3 2 1 1
	Case 1:
	  Party 1 has power index 10
	  Party 2 has power index 2
	  Party 3 has power index 2
	  Party 4 has power index 6
	  Party 5 has power index 6
	
	Case 2:
	  Party 1 has power index 18
	  Party 2 has power index 14
	  Party 3 has power index 14
	  Party 4 has power index 2
	  Party 5 has power index 2
	  Party 6 has power index 2
	
	Case 3:
	  Party 1 has power index 3
	  Party 2 has power index 1
	  Party 3 has power index 1

6. When a beam of light traveling through one medium (air, for example) impinges on the flat surface of another medium (glass, for example), the beam bends according to Snell's law:

$$n_1 \sin\theta_1 = n_2 \sin\theta_2,$$

where n_1 and n_2 are the *refractive indices* of the two media and θ_1 and θ_2 are the angles between the beam and the normal (perpendicular) to the surface in the two media.



This diagram shows the path of a light beam through a glass object, indicating the definitions of θ_1 and θ_2.

We'd like a program that, given a description of a polygonal object, the initial path of a light beam, and the parameters n_1, n_2, θ_1, and θ_2, computes the point (if any) at which the light beam strikes a wall on the right.

The input will consist of multiple sets of data in free format. Each set will start with floating-point values y_0, x_w, y_w, n_2, and an integer N. The n_2 is the refractive index of the object; we fix $n_1 = 1.0$. The light beam starts at position $(0, y_0)$ and is aimed in such a way that if $n_2 = 1$ (so the object does not refract light), it would hit the wall at point (x_w, y_w). There then follow $2N$ floating-point values $x_1, y_1, \ldots, x_N, y_N$, where $0 < x_i < x_w$ for all i. The values (x_i, y_i) are the vertices of the polygon enclosing the refracting material in clockwise order. This polygon is non-self-intersecting. You may assume that the light beam will intersect it exactly twice, and that the beam will not intersect the polygon at a vertex.

The output will consist of one line for each set of input giving the y-coordinate at which the beam hits the far wall rounded to two decimal places, or the phrase "does not reach the

wall" if the beam is deflected away from the wall. Use the format shown in the examples below.

Example:

Input	Output
`-0.75 3.00 2.25 1.5 4` `0.75 0.29 2.25 0.87 2.25 -0.87 0.75 -0.29`	`Case 1: 1.55`
`-0.75 3.00 -1.0 1.5 4` `0.75 0.29 2.25 0.87 2.25 -0.87 0.75 -0.29`	`Case 2: -1.0`
`-0.75 3.00 2.25 20 4` `0.25 0.87 1.75 0.29 1.75 -0.29 0.25 -0.87`	`Case 3: does not reach the wall`

7. [From a USACO contest] Given a non-empty set of prime numbers, $P = \{p_1, \ldots, p_n\}$, we want to find the k^{th} smallest number in the set of all positive numbers whose prime factors are all in this set. For $k = 1$, this is always 1, since the set of all prime factors of 1 is empty.

For example, if $P = \{2, 3, 5\}$, then the first 20 conforming numbers are

```
1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36
```

The input to your program consists of one or more cases in free format. Each case begins with an integer N giving the number of primes and an integer K giving the index of the desired number ($K = 1$ for the smallest, $K = 2$ for the next larger, etc.). Next come N distinct prime numbers.

For each case, output the K^{th} smallest member of the set, using the format shown below. You may assume that the answers are always $\leq 2^{31} - 1$.

Example:

Input	Output
`3 20` `2 5 3`	`Case 1: 36`
`3 10` `3 7 11`	`Case 2: 63`
`4 5300 2 3 5 7`	`Case 3: 1120000000`

8. You have probably all seen examples of PDF417-style bar codes, such as the one shown here. This is a type of *stacked linear bar code:* it consists of rows of simple bar code that have been squished vertically and then stacked on top of each other. An engineer from the small country of Outer Freedonia noticed these on postage he received from the US, and thought it might be a nice idea to introduce its use in his country as well. Unfortunately, he couldn't afford (or was too cheap to pay) the fee for obtaining the full specification, so he did some reverse engineering and came up with a much simplified (albeit very inefficient) version.

Freedonian bar code encodes only decimal digits (0–9). Besides start and stop bars on the sides (which we'll ignore), there is a pattern of eight white or black bars for each symbol, which we can encode in the bits of a single byte as a number in the range 0–255. Now in fact, there are three different encodings of the digits 0–9 used, which we'll call encodings (or *clusters,* to use the original terminology) E_0, E_1, and E_2:

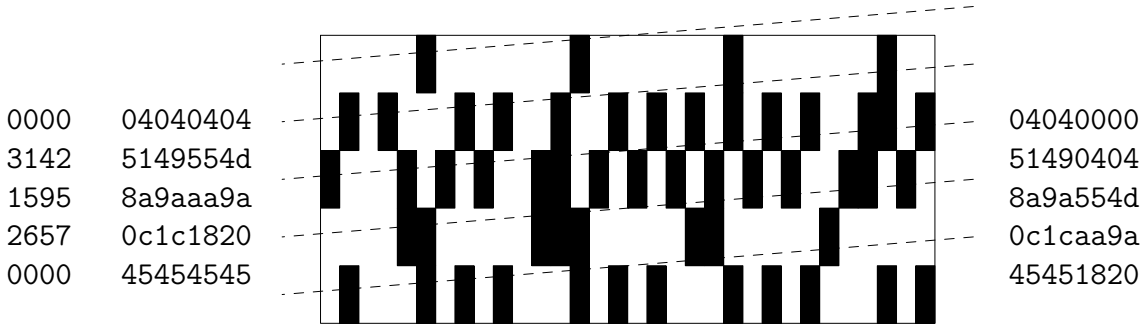
	Digit									
	0	1	2	3	4	5	6	7	8	9
E_0	0x04	0x08	0x0c	0x10	0x14	0x18	0x1c	0x20	0x24	0x2c
E_1	0x45	0x49	0x4d	0x51	0x55	0x59	0x5d	0x61	0x65	0x6d
E_2	0x86	0x8a	0x8e	0x92	0x96	0x9a	0x9e	0xa2	0xa6	0xaa

These encodings are disjoint: none of the 10 valid codes in E_0, for example, is a valid code in E_1 or E_2. The first row of a bar code uses encoding E_0, the next used E_1, the next E_2, and the next E_0, and so forth, cycling through the three encodings. There is a reason for this: when people use hand-held laser scanners—which read the bars of a given encoding one line at a time, left-to-right, top-to-bottom—they usually don't align them perfectly, so that, for example, a scan line starts start out scanning line #5 on the left, and then, half-way across, has drifted upwards and start reading from line #4 (or downwards and is reading from line #6). By using different, non-overlapping alphabets, you can detect this situation and reconstruct the code (since usually, your scan of the next line or the previous one will also be slanted, and will pick up the missing symbols). Sometimes, the scanner will skip to the next line in the middle of a symbol. When this happens, the scanner will output an invalid symbol (not in any encoding).

Each row in a given code has the same number of symbols. The last two symbols encode check digits, so that the sum of all the base-10 digits in a line (including the check digit) add up to 0 modulo 10. The first and last line of every bar code (whose decodings are thrown away) contain all 0-digits, in the appropriate encoding for those lines (thus making sure that some scan line picks up the first and last real lines of data). You may lose part of these lines if the scanner goes off the top or bottom of the barcode, but since they are thrown away anyway, it does not matter.

For example, the digit string '314159265' can be broken into five lines of three real symbols and a check digit as shown on the left below. These can then be encoded as shown in hex coding after that. This encoding corresponds to the bar code shown in the middle. If the scanner is tilted as shown by the dashed lines, the scanner might pick up

the values shown on the right. In this case, we lose the two digits (including the check digit) at the end of the last (all encoded 0s) line, and all the symbols on each line are valid. In other cases (where the scanner changes lines in the middle of a symbol), there will be invalid symbols scanned, and a decoder has to use the check digit to reconstruct the missing symbol. It is also possible for the check digit to be corrupted, but in that case, all other symbols on the line will have been scanned correctly.



Your job is to write a decoding program, that, given the streams of bits of each line from a scanner (in hex form), reconstructs the encoded numerals. Each scan line will have the same number of digits. If the scanner slips off the top or bottom of a bar code, it sees `0x00`'s (which are not valid codes in any of the E_i). You may assume that if the scanner is tilted, it scans at most two rows, as in the example, and that each symbol it returns is either a correct symbol (from some line) or is not in any of the encodings. We will not assume that the tilting is completely consistent between lines; the scanner may not always change lines at the same point for each scan line.

The input consists of a number of sets in free format. Each set begins with an even integer $N > 2$ indicating the number of symbols (including check digit) on each line, and an integer $M > 2$ indicating the number of lines. This is followed by M $2N - digit$ strings of hexadecimal digits (no leading '`0x`'), giving the scan lines. The first and last lines of the bar code encode all 0s followed by check digits 0 (also encoded). It is not necessary to reconstruct the first and last lines (since they are known). All input sets will be valid encodings according to the rules above.

For each input set, the output consists of the set number followed by the string of $(M-2)(N-1)$ decoded digits (without check digits and without the first and last all-zero lines) in the format shown in the examples below.

Example:

Input	Output
4 5 04040000 51490404 8a9a554d 0c1caa9a 45451820 4 5 04040000 51480404 8a9b554d 0c1caa9a 45451820	Set 1: 314159265 Set 2: 314159265