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

Programming Contest **P. N. Hilfinger**
Fall 1999

1999 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 15 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, 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. 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`, and `fstream`. 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`, and `java.util`. 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`.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 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 45 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. For C and C++ programs, it is equivalent to

```
gcc -Wall -o N -O -g -Iour-includes N.* -lstdc++ -lm
```

For Java programs, it is equivalent to

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

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

```
java 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 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 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.

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).

Notices. During the contest, the Web page at URL

`http://http.cs.berkeley.edu/~hilfingr/programming-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. Be sure to reload this page from time to time, to keep information up to date.

1. [With thanks to Michael Schindler.] Consider a code in which each letter of some alphabet is translated to a sequence of bits, whose length may differ from letter to letter. The encoding satisfies (a) the *unique-prefix property:* if the code for letter x is shorter than that for letter y, then the code for x is not a prefix of the code for y (as a result, if the binary codes for all letters in a message are concatenated, it is easy to tell where the code for one letter ends and the next begins). The encoding also satisfies the property (b) that if one letter has the code $b_1 \cdots b_{n-1}b_n$, where $n > 1$ and each b_i is either 0 or 1, then there must also be $n-1$ other letters whose codes begin respectively with $b_1 \cdots b_{n-2}\overline{b_{n-1}}$, $b_1 \cdots b_{n-3}\overline{b_{n-2}}$, ..., and $\overline{b_1}$ (where $\overline{b}$ is the complement of bit b). Huffman codes are an example. These two restrictions mean that if the alphabet contains the letters A, B, and C, then

$$A = 1,\ B = 00,\ C = 01$$

is a valid encoding while

$$A = 1,\ B = 00,\ C = 10 \ \text{ and} \tag{1}$$

$$A = 01,\ B = 000,\ C = 001 \tag{2}$$

are not valid encodings (the first violates (a) because the code for C starts with the code for A; the second doesn't have that problem, but violates (b) because all codes start with '0').

We'll call a valid code *canonical* if it also satisfies two other properties:

(c) If the code for some letter contains k bits, then it is larger than the first k bits of any longer code (when both are treated as binary numbers). For example, in a canonical code, if one letter has the code 10, some other letter could have the code 011, but not 111.

(d) If the codes for two letters have the same length, then the letter that comes first in the alphabet has the smaller encoding (again when treated as binary numbers).

(e) When there are multiple k-bit encodings for a letter according to the other rules, choose the largest allowed.

If a certain alphabet has been given a canonical encoding according to all these rules, then you can reconstruct the encoding given only the *lengths* in bits of the codes for each letter.

You are to write a program that, given the lengths of the codes for each letter in an alphabet, reconstructs the canonical encoding. The input consists of a sequence of pairs, each consisting of a single printable (non-whitespace) character and a positive integer indicating the length of its encoding. Input is in free form. Alphabetical order is defined as the order in which the characters are listed (*not* necessarily the built-in collating sequence). The output should echo the alphabet, one character per line, followed by a space and the computed encoding. If no encoding that obeys the rules is possible, the output should simply be the line "`Invalid input`". You may assume that the input has the right form, and that the alphabet size is limited by the number of printable characters.

Example 1:

Input	Output
`A 1 B 2 C 2`	`A 1`
	`B 00`
	`C 01`

Example 2:

Input	Output
`H 3 G 4 F 4`	`H 010`
`E 3 D 2`	`G 0000`
`C`	`F 0001`
`4`	`E 011`
`B 4`	`D 10`
	`C 0010`
`A 2`	`B 0011`
	`A 11`

Example 3:

Input	Output
`A 1 B 2 C 2 D 2`	`Invalid input`

2. When multiple computer programs are able to access the same memory simultaneously, there can be unpredictable behavior because of various possible *race conditions,* in which one program (we'll call it a *process* from now on) writes into a memory location (variable) that another process might read from or write to at about the same time—sometimes before, sometimes after, depending on the relative speeds of the processes, so that results differ depending on which processor wins the race. You are to write a program that analyzes *traces* executed by the processes to see if race conditions might exist.

A trace is just a sequence of instructions, in our case simplified instructions. A trace from a single process consists of a sequence of three kinds of instruction, written as follows:

rN where N is a non-negative integer, denotes a *read* from memory location N.

wN where N is a non-negative integer, denotes a *write* to memory location N (what value is written is irrelevant in this problem).

b denotes a *barrier.* When a process executes a barrier, it waits until all other processes are also waiting at a barrier. All processes then proceed to their next instruction.

To simplify some descriptions that follow, we'll assume that all traces start with a 'b' command.

Suppose that we have M traces, one for each of M processes. The idea is that all processes start at the beginning of their respective trace, synchronize at barriers, but otherwise execute instructions in their trace at unpredictable speeds. We say that process j $(0 \leq j < M)$ has a *read-write* conflict with process j' $(j \neq j')$ if after a point in which the two processes are synchronized, and before they reach their next barrier (if any), one of the processes performs an 'rn' command and the other performs a 'wn' command (same n in each case). We say that j has a *write-write* conflict with j' if after a point in which the two processes are synchronized, and before they reach their next barrier, both processes perform a 'wn' command (again, the same n in both cases). (Reads do not conflict with each other). The motivation for these definitions is that in both cases, the two operations can occur in either order, since between barriers the processes can execute instructions at independent speeds.

For example, here are two conflicting traces that display both a read-write conflict (on location 1) and a write-write conflict (on location 2).

```
Process 0: b r2 r1 b w1 w2
Process 1: b w1 r3 b w2 r3
```

The following traces are *not* in conflict:

```
Process 0: b r1 r2 b r3 w1 b w2 r1
Process 1: b r1 r2 b w2 r2 r3 b w3 r1
```

Here, even though the two processes both read or write locations 1, 2, and 3, they do so after different barriers. Also, since a process never conflicts with itself, a sequence like '`w2 r2 r3`' is perfectly good in a single trace.

The input to your program will consist of an integer, $M > 1$, followed by M traces. Each trace will consist of a sequence of read, write, and barrier commands separated by whitespace in the format described above, with each trace terminated by a single period (.), also separated by whitespace from other input. The first trace is for processor 0, the second for processor 1, etc. You may assume that all input is correctly formatted and begins with a 'b' command. You may also assume that all traces have the same number of 'b' commands. Finally, you may assume that memory locations are numbered 0–1023.

Your output is to consist of a report indicating each pair of processors that has a conflict, with a list (in ascending order with no repetitions) of all memory locations on which that pair of processors conflicts. Order the pairs of processors you report by the smaller processor's number, and then by the larger processor's number in the format shown in the examples below.

Example 1:

Input	Output

`2`
`b r2 r1 b w1 w2 .`
`b w1`
`r3 b`
`w2 r3 .`
	`Processes 0 and 1 conflict on 1 2`

Example 2:

Input	Output

`2`
`b r1 r2 b r3 w1 b w2 r1 .`
`b r1 r2 b w2 r2 r3 b w3 r1 .`
	`No conflicts`

Example 3:

Input	Output

`4`
`b r2 r1 b w1 w2 b .`
`b w1 r3 b w2 r3 b r3 .`
`b r1 r2 b r3 w1 b w2 r1 .`
`b r1 r2 b w2 r2 r3 b w3 r1 .`
	`Processes 0 and 1 conflict on 1 2`
	`Processes 0 and 2 conflict on 1`
	`Processes 0 and 3 conflict on 2`
	`Processes 1 and 2 conflict on 1`
	`Processes 1 and 3 conflict on 1 2 3`

3. We can write simple algebraic expressions two dimensionally, like this:

```
                2
x    = x  + (y-x )/2x
 n+1    n       n    n
```

or we can write them in TeX format, like this:

```
x_{n+1} = x_{n} + (y - x_{n}^{2})/2x_{n}
```

In this problem, you are to convert the first form to the second.

The input consists of up to three lines of text, a superscript line (possibly empty), a main line, and an optional subscript line. Each superscript applies to the last character on the main line that is below and to its left. The lines contain only printing characters and blanks (no tabs). Each subscript applies to the last non-blank character on the main line that is above and to its left. You are to convert subscripts and superscripts to the notation illustrated above, with subscripts looking like `_{...}` and superscripts like `^{...}`. When a character has both a subscript and a superscript, put the subscript first. Remove any blanks on the main line that have a sub- or superscript above or below them. Preserve any other blanks on the main line and within sub- or superscripts.

Example 1: The lines of dashes in these examples indicate the end of the input; they are not actually present in the input.

Input

```
                 2
 x    = x  + (y-x )/2x
  n+1    n       n    n
--------------------------
```

Output

```
x_{n+1} = x_{n} + (y - x_{n}^{2})/2x_{n}
```

Example 2:

Input

```
  3                2
 x    =  (x +(y-x) )/2x
           n           n
-----------------------------
```

Output

```
x^{3}   =  (x_{n}+(y-x)^{2})/2x_{n}
```

Example 3: Illustrating that the subscript line is optional, but not the superscript line (blank in this case).

Input

```

 x    =  (x +(y-x) )/2x
---------------------------
```

Output

```
x    =  (x +(y-x) )/2x
```

4. [With thanks to E. R. Berlekamp, J. H. Conway, and R. K. Guy] In the game of D.U.D.E.N.E.Y. (Deductions Unfailing, Disallowing Echoes, Not Exceeding Y), two players are given two positive numbers Y and N_0. Initially, we set a variable N to N_0. Now each player in turn must select a number between 1 and Y that does not exceed N and reduce N by that number. A player may not choose the number his opponent has just chosen (Disallowing Echoes). The winner is the last player who can move (Deductions Unfailing). For example, if Y is 3 and N_0 is 2, the first player can win by subtracting either 1 or 2 (2 reduces N to 0 and 1 reduces N to 1, from which the only move is to subtract 1, which would be an illegal echo).

The problem is to produce, for a given range of Y and N values, a strategy sheet showing what move to make in order to win the game (if there is one). Your input consists of four numbers in free format, Y_l, Y_u, N_l, and N_u, with $1 \leq Y_l \leq Y_u$, and $1 \leq N_l \leq N_u$. These indicate that you are to produce a table of possible winning plays for all values $Y_l \leq Y \leq Y_u$ and $N_l \leq N \leq N_u$. For any given Y and N, there may be several possible plays (in the middle of a game, one of these might be illegal because of the no-echo rule; ignore this possibility and print out moves that would win at the beginning of the game). Display them in the format shown in the examples below, listing possible plays for each (Y, N) pair in ascending order. If there is no winning move, print '`?`'. Spacing is *not* critical in this problem: columns need not line up as in the example, just so long as there is at least one blank between columns and no blanks within the comma-lists of numbers. Expect large values of N_l and N_u (say around 10,000,000) and moderate values of Y_l and Y_u (say less than 20).

Example Input:

```
1 3 1 6
```

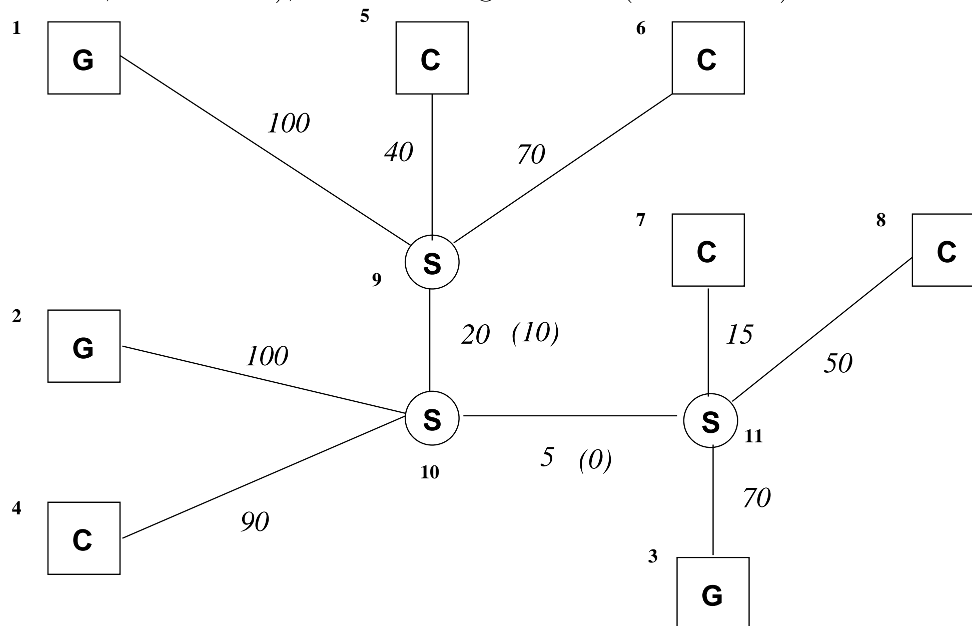
Output:

```
Y = 1.
N:          1   2   3   4   5   6
Subtract:   1   1   1   1   1   1

Y = 2.
N:          1    2    3    4    5    6
Subtract:   1    1,2  ?    1    1,2  ?

Y = 3.
N:          1    2    3    4    5    6
Subtract:   1    1,2  3    ?    1    1,2,3
```

5. Consider a simplified power grid containing generating plants (marked 'G'), loads (or consumers, marked 'C'), and switching stations (marked 'S') such as the one below:



The italic numbers outside parentheses on the lines indicate capacities (in some unit of power, such as kilowatts). The small numbers identify the various plants, customers, and switching stations (they are used in the first example below). There is always a single line out of each generator and into each customer, connecting on one end to a switching station. Switching stations can route power from along the lines running into them in any fashion that insures that the total amount flowing in is equal to the total flowing out, and no line's capacity is exceeded. We want to know whether there is some way to route enough power from some or all of the generators to all the customers so as to meet the total capacities of the lines going into the customers. We assume that all the generators can produce as much power as the lines going out of them can carry. In the example above, we can satisfy customer demand by routing the amount of power indicated in parentheses along the indicated lines between switching stations. It is only necessary that customer demand be satisfied; it doesn't matter if there is excess generating capacity.

The input will consist of the following items, in free format: an integer n_g indicating the number of generators, an integer n_c indicating the number of customers, an integer n_s indicating the number of switching stations, a sequence of connections, each represented by three integers f, t, c indicating a capacity of c units between locations f and t. Locations 1 to n_g are generators, $n_g + 1$ to $n_g + n_c$ are customers, and $n_g + n_c + 1$ to $n_g + n_c + n_s$ are switches.

Your output should either say "`Demand cannot be met.`" or "`Demand can be met.`" as shown in the following examples.

Example 1: This is a rendering of the diagram in the problem. Generators are 1–3, customers are 4-8, and switches are 9–11.

Input	Output

`3 5 3`
`1 9 100`
`4 10 90`
`2 10 100`
`5 9 40`
`6 9 70`
`9 10 20   10 11 5`
`3 11 70`
`8 11 50`
`7 11 15`
	`Demand can be met.`

Example 2: This is a rendering of the diagram in the problem with the capacity of one line between switching stations reduced.

Input	Output

`3 5 3`
`1 9 100`
`4 10 90`
`2 10 100`
`5 9 40`
`6 9 70`
`9 10 5   10 11 5`
`3 11 70`
`8 11 50`
`7 11 15`
	`Demand cannot be met.`

6. A simple kind of puzzle involves mutating a word, one character at a time, into another word of the same length, using only words of the same length from some dictionary. For example, to convert 'lie' to 'pay', one might use 'lie', 'pie', 'pin', 'pan', 'pay'. You are to write a program to do this.

The input (all free-form) will consist first of two words, w and w', of the same length, followed by an additional list of words (including w and w') in alphabetical order. Here, a "word" is a sequence of lower-case letters.

Your output should consist of a list of words, starting with w and ending with w', all on one line, separated by commas as shown in the examples below. Output the *shortest* sequence of words possible. When choosing between two sequences of the same length, look for the first point at which the two sequences differ and choose the one with the word that comes first alphabetically. Assume there is a solution. Also assume that words are at most 20 characters long. Don't assume anything else about the number of words in the dictionary (it may be large).

Example 1.

Input:

```
lie pay
aforesaid afterward barefaced bedfast beforehand bufflehead
featherbedding federal federate feedback fountainhead frayed
freehand lie lied lien lieu pam pan panacea panama pancake
pay pap pie pin shamefaced shuffleboard steadfast yam yard
zoo
```

Output:

```
lie, pie, pin, pan, pay
```

Example 2.

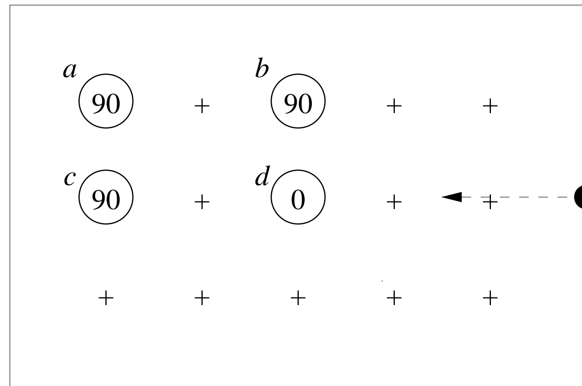
Input:

```
lie pay
aforesaid afterward barefaced bedfast beforehand bufflehead
featherbedding federal federate feedback fountainhead frayed
freehand lap lie lip lied lien lieu pam pan panacea panama pancake
pay pap pie pin shamefaced shuffleboard steadfast yam yard
zoo
```

Output:

```
lie, lip, lap, pap, pay
```

7. Consider the following very simplified pinball game:



This particular example consists of a 3×5 grid surrounded by a border, a single ball (the black circle shown emerging from the center right), and four *progressive twisters,* which function to turn the ball. Each twister is marked with the initial amount that it turns the ball's path when the ball goes through it (in degrees clockwise), which I'll call the twister's current *twist.* Each time the ball passes through a twister, it gets turned according to the twister's current twist, and that twist then gets incremented by 90 degrees clockwise.

Assume that the ball takes one unit of time to move one grid position (up, down, left, or right). Then in the diagram, it starts at time 0, hits d at time 3. It is not turned, since d's initial twist is 0, but that twist now becomes 90 degrees. Next the ball hits c at time 5 and gets turned upward. c's twist is now 180. Then we hit a at time 6 and turn right. Next comes b at time 8, and d at time 9. The ball now goes to its right (our left), since d's twist is now 90 (incremented to 180). The ball hits c at time 11 and comes back (180 degree turn) to d at 13, then back to c at time 15, and then to its left (c's twist being 270), and finally hits the border at time 17.

Your program is to take as input a description of such an initial setup and report when the ball hits the border (we will arrange that it always does). The input (in free form) consists first of two integers, m and n, which are the number of grid points vertically and horizontally (the border extends beyond the grid by one in each direction). Next is a list of twister positions and initial settings, each in the form of three integers x, y, and t, which are respectively its x position and y position, and its initial twist (0, 90, 180, or 270). All grid coordinates are positive, with (1,1) being the position of the lower-left grid point (and (0,0) is the position of the lower-left border corner). The ball always starts halfway up the right border (at y-coordinate $\lfloor (m+1)/2 \rfloor$).

The output consists of one line of the form

```
Game ends at time 17.
```

You may assume the game will always end eventually. See the next page for examples.

Example 1. The diagram could be entered like this:

Input	Output
`3 5` `1 2 90    1 3 90    3 3 90    3 2 0`	`Game ends at time 17.`

Example 2. This one goes straight through.

Input	Output
`3 5` `1 2 0    1 3 90    3 3 90    3 2 0`	`Game ends at time 6.`

8. You are given a base-36 numeral (the digits are 0–9, *a*–*z*, with *a* worth 10, *b* 11, etc.) and told that it comes from the set of all numbers having the same digits (that is, the same number of each digit the given number has). We allow leading 0's. You are to print out the next number from that set (in ascending order). For example, given `03snd3fk5ee2,` you should print `03snd3fke25e`.

The input will consist of any number of base-36 numerals up to 128 digits long in free format. For each, compute the next number in its set (you may assume there is one) in the format shown below:

Example.

Input	Output
`12 03snd3fk5ee2`	`12 -> 21`
	`03snd3fk5ee2 -> 03snd3fke25e`