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

Programming Contest **P. N. Hilfinger**
Fall 1997

1997 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 17 pages. You have 5 hours in which to solve as many of them as possible. Put each complete C solution into a file N`.c` 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. 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 2> junk-file
```

which sends normal output to *test-input-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 -Iour-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.

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).

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.

1. A number of games involve a player that moves around a playing board of some kind and attempts to avoid crossing its own trail, or those of its opponents. For this problem, you are to help score one such game. The players in this game are represented by pegs, and the board consists of an $N \times N$ array of holes, where N is a parameter. Players move in sequence. They may move to any peg that has never been occupied as long as the line segment to their destination from the peg they are leaving does not cross any previously traveled path (whether by the same or a different player). The player who violates these rules loses. Your program determines the winner of such a game.

The input to your program is in free-form, and consists of

- A positive integer, N, indicating the number of holes along each side of the board;
- A positive integer, P, indicating the number of players;
- A sequence of P pairs of numbers in the range 0 to $N - 1$, inclusive, indicating the initial row and column number of each player;
- A sequence of pairs of numbers in the range 0 to $N - 1$ indicating the row and column number of the next move for each player in sequence. That is, the first pair indicates the first move for Player 0; the second pair indicates the first move for Player 1; and so on. After player $P - 1$ moves, the next pair is for Player 0's second move, and so forth. The sequence is terminated by the pair '`-1 -1`'.

The output of your program is either the sentence

```
All players are still in the game
```

if you reach the terminating (`-1 -1`) pair without any paths being crossed, or

```
Player n loses on moving to peg (x, y)
```

if Player n ($0 \leq n < P$) loses by moving to peg (x, y)—that is, if peg (x, y) has been moved to before or if in moving to it from his former position, Player n crosses a path that has already been trod. There are sample inputs and outputs on the next page.

In the first example (on a 100,000×100,000 board with two players), Player 0 starts at (1000, 1000)—row 1000, column 1000—and then moves from there to (1500, 1000), then to (1400, 800), and finally to (1200, 1200). The last move, however, intersects the segment from (1000, 1000) to (1500,1000), so Player 0 loses. In the second example, Player 1 moves from (5,3) to (3,9), and then Player 2 moves from (1,6) to (7,2), crossing Player 1's path. Thus Player 2 loses (so would Player 0 if the game had continued, but we stop at the first loss).

You may assume that all input numbers are in the legal ranges indicated by these instructions, and that the players start from different squares. You may assume $P < N$, but you may not assume any limits on N, aside from it being a representable machine integer. In particular, expect the program to be tested with at least one very large value of N.

Example 1: Player 0 crosses his own track

Input

```
100000 2
1000 1000 2000 2000
1500 1000 1500 1500
1400 800  3000 1500
1200 1200 90000 10
-1 -1
```

Output

```
Player 0 loses on moving to peg (1200, 1200)
```

Example 2: Player 2 crosses player 1’s track

Input

```
10 3
0 0  5 3  1 6
0 8  3 9  7 2
8 8  2 1  9 9
-1 -1
```

Output

```
Player 2 loses on moving to peg (7, 2)
```

Example 3: No collisions

Input

```
10 3
0 0  5 3  1 6
0 8  3 9  5 2
2 8  6 8  3 0
-1 -1
```

Output

```
All players are still in the game
```

2. [Due to Geoff Pike] Consider the problem of computing

$$\frac{x_0}{y_0} \cdot \frac{x_1}{y_1} \cdots \frac{x_{n-1}}{y_{n-1}}$$

where the x_i and y_i are positive integers. In a conventional language such as C++, the difficulty, of course, is that the result might overflow the range of representable integers. Let us assume that the numerator and denominator of the answer (in lowest terms) are representable (specifically, that their magnitudes are less than 2^{31}). Even then, the product of the x_i, or the product of the y_i, or the product of some initial sequence of the x_i/y_i may not be representable (for example, consider

$$\underbrace{\frac{2}{3} \cdot \frac{2}{3} \cdots \frac{2}{3}}_{\text{40 times}} \cdot \underbrace{\frac{3}{2} \cdot \frac{3}{2} \cdots \frac{3}{2}}_{\text{40 times}}$$

which is simply 1).

The input to your program will consist of any number of pairs of integers separated by slashes. All integers in the input will be in the range 1 to 10000.

The output will be the product of all the input pairs (treated as fractions) in lowest terms. Again, assume that the numerator and denominator of the result (in lowest terms) will be less than 2^{31}.

Example:

Input	Output
`10000/1 10000/2 10000/3`	`2/7`
`10000/4`	
`10000/5`	
`1/2     4/7     5/10000`	
`4/10000 3/10000 2/10000`	
`1/10000`	

3. Sally supervises a group of programmers who are to work on the modules of a large product. She assigns one lead programmer to each module, and asks the lead programmers each to choose a (less-experienced) junior programmer to assist. Sometimes, however, two lead programmers both want the same assistant. At the same time, of course, the junior programmers have their own preferences about which lead programmers they wish to work with.

In an attempt to satisfy all parties at least to some extent, Sally decides on a simple criterion that any pairing of junior and lead programmers will have to meet. Specifically, if Jack (lead) and Mary (junior) form a team, there should never be another team—say Toni (lead) and Jason (junior)—in which Jack would prefer to work with Jason and Jason would prefer to work with Jack. Sally thinks this problem tricky enough to warrant a program, which she asks you to write.

The input to this program (in free form) will consist of a positive integer, N, giving the number of teams, followed by $2N$ lists. Each list consists of the name of a programmer (a string of up to 64 characters without embedded whitespace), followed by the names of N other programmers in decreasing order of preference. The first N lists each begin with the name of a lead programmer, followed by the names of all N junior programmers, and the last N preference lists begin with the names of the junior programmers, followed by the names of all N lead programmers. The output of your program should be a possible pairing, in the format shown in the example below. List the results in the order that the lead programmers were initially listed. Where there are multiple possible legal pairings, favor the preferences of the lead programmers—so that they get the assistant they most prefer, subject to the constraints of the problem.

Example:

Input

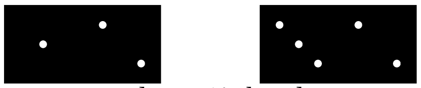
```
3
Tim Bryan Mary Binh
Jane Mary Binh Bryan
Tommy Mary Binh Bryan
Mary Tim Tommy Jane
Binh Tim Jane Tommy
Bryan Tommy Tim Jane
```

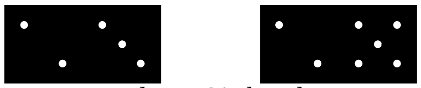
Output

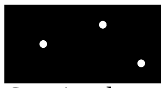
```
Tim Bryan
Jane Binh
Tommy Mary
```

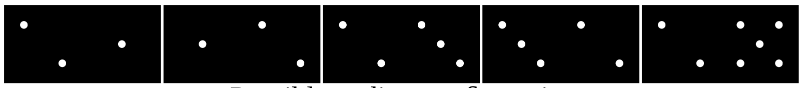
4. A simplified variety of the game of dominos is played with tiles called *bones* that bear a set of *spots* on each of two faces (*ends*), with from zero to six spots on each end. Play consists of two players alternately moving a bone from their *hand* (a collection of bones) to a valid place on the the table. The table always contains a set of adjacent bones laid so that when ends of adjacent bones touch, each bears the same number of spots. Each bone with differing numbers of spots on its two faces can touch up to two other bones—one for each end. Each bone having the same number of spots on each of its two ends (a *doublet*) may touch up to four other bones. In both cases, when the ends of two bones touch, the ends must have the same number of spots. Initially, the two player's hands have the same number of bones and the table contains a single, arbitrary bone.

The problem is to determine, given two hands and the table's starting bone, whether there is any way for the two players to play all their bones to the table, exhausting their hands. In the example below, each player starts with two bones, and there is a way to satisfy the conditions. The starting bone ends up second from the left. The example illustrates that bones can be flipped as necessary to put their ends together (the first player had to flip his (1,2) bone by 180°).


Player 1's hand

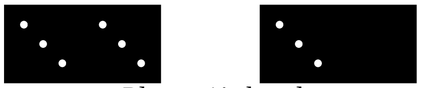

Player 2's hand

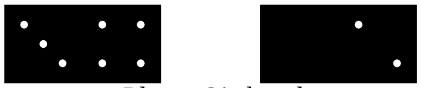

Starting bone

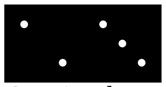

Possible ending configuration

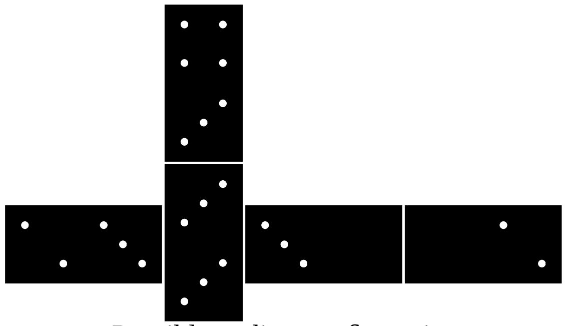
Had the second player's hand contained a (4,5) bone in place of the (2,5) bone, however, there would have been no way to reach a valid final configuration.

Here is another successful example that involves a doublet:


Player 1's hand


Player 2's hand


Starting bone


Possible ending configuration

As you can see, it is traditional to lay doublets crosswise. However, for this problem, the actual geometry of the layout is entirely irrelevant. It doesn't matter that rows of dominos would collide or overlay each other, for example.

The input to your program will consist (in free form) of a positive integer, N, indicating the number of bones in each hand, followed by $2N + 1$ pairs of numbers in the range 0–6. The first N pairs are Player 1's hand; the next N pairs are Player 2's; and the remaining pair represents the starting bone. The output should be as illustrated in the examples on the next page.

Example 1:

Input	Output
`2`	`It is possible to play all bones`
`1 2 3 2    2 3 2 5`	
`1 2`	

Example 2:

Input	Output
`2`	`It is not possible to play all bones`
`1 2 3 2    2 3 4`	
` 5`	
`1`	
`2`	

Example 3:

Input	Output
`2 3 3 3 0 3 4 0 2 2 3`	`It is possible to play all bones`

5. One use of a spell-checking program is to allow one to write in abbreviations, which a program replaces when there is a unique expansion. For this problem, we want a program that takes as input a sequence of words constituting a *lexicon*, followed by an abbreviated message. Each word, W, in the message is replaced by a word, W', from the lexicon if W' is the unique shortest word in the lexicon that can be formed by inserting 0 or more letters into W. If no W' satisfies this criterion, then W is passed through to the output unchanged.

For example, if the lexicon contains

```
arrived truly tally shipment has yours your our congratulations
```

and the message is

```
yr pmt hs arrd.  our congr. yrs try, jack
```

the output would be

```
your shipment has arrived.  our congratulations. yours truly, jack
```

Had the input contained the word `tly` instead of `try`, on the other hand, it would have remained unchanged, since both `truly` and `tally` match it.

The input to your program will consist of a sequence of words in free format (a ‘word’ here is simply a contiguous sequence of letters and digits), followed by a word consisting of a single slash—preceded by whitespace and followed by a newline—after which comes an arbitrary sequence of lines of abbreviated text. The words up to the slash are the lexicon. The abbreviated words are delimited by any characters that are not letters or digits.

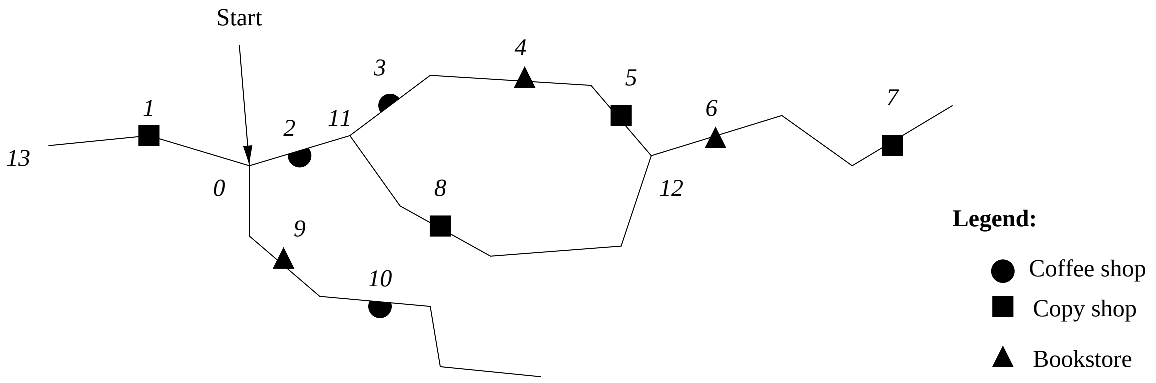
The output should consist of the lines after the slash, with all abbreviated words replaced as indicated above (and others unchanged). Preserve all characters that are not letters or digits (in particular, punctuation and whitespace).

Example:

Input	Output

`arrived truly shipment tally`
`has yours your`
`our`
`congratulations /`
`yr pmt hs arrd.`
`our congr.`
`yrs try, jack`
	`your shipment has arrived.`
	`our congratulations.`
	`yours truly, jack`

6. Consider a branching network of roads, sometimes doubling back on itself, along which one finds a variety of shops. The network is connected; if one starts walking from any fixed point in this network, one will encounter some sequence of shops, and one can reach any shop in the network. I'd like to know, however, what is the *first* shop of some particular kind (produce, meat, stereo equipment, etc.) that I might encounter on any particular walk. For example, on the following map, the first coffee shop I encounter, depending on the route I take, will be #2 or #10. The first bookstore will be #4, #6, or #9. The first copy shop will be #1, #8, or #5. You are to write a program to determine the general answer to the question, "Which shop of type X might I first encounter on a walk?"



The input (in free format) will consist of a non-negative integer called the *starting address,* another non-negative integer called the *target type,* a sequence of numbered *places* terminated by a pair of negative numbers, finally followed by a sequence of *paths* joining pairs of places. Each place takes the form of a pair of non-negative integers, the first of which (the *address*) is unique to each place, and the second of which (the *type*) corresponds to a type of shop; any number of addresses may have the same type. Each path consists of a pair of previously defined addresses, indicating that a direct path exists between the places with those addresses. Assume that all addresses are less than 1000.

Having read this input, your program must print out the list of places of the specified target type that one first encounters when traversing paths starting at the starting address (the shop at the starting address counts as an encounter). List the addresses of the places in increasing numerical order, without repetitions, using the format illustrated in the example.

Example:

Input:

```
0 2
0 0
1 2 8 2 5 2 7 2
2 1 3 1 10 1
9 3 4 3 6 3
11 0 12 0 13 0
-1 -1
13 1 1 0 0 9 9 10
0 2 2 11
11 3 11 8 3 4 4 5
5 12 8 12
12 6 6 7
```

Output:

```
Starting at 0, you might first encounter type 2 shop #1, 5, 8
```

7. The text formatting program TEX breaks paragraphs into lines so as to minimize the total *demerits* of the paragraph. Its definition of "demerit" is a bit complex; here we will deal with a simplified version. In this simple version, we'll break paragraphs into lines only at word boundaries (blanks). The demerits assigned to a particular choice of where to put line breaks is the sum of the demerits for all the resulting lines.

To compute the demerits, d, of one line, we need to know w, the number of words on the line; and c, the total number of characters on the line (including single blanks between words). Then for that line,

$$s = \begin{cases} \min(0, 30-c), & \text{on the last line,} \\ 30-c, & \text{otherwise.} \end{cases}$$

$$b = \begin{cases} \left(\frac{2s}{\max(1,w-1)}\right)^3, & \text{if } s \geq 0 \\ \left(\frac{-5s}{\max(1,w-1)}\right)^3, & \text{if } s < 0. \end{cases}$$

$$d = (0.1+b)^2$$

assuming that lines are normally 30 characters long. In these formulae, b, a floating-point number, is the "badness" of the line, and s is the total stretching of the blanks (or shrinking, if negative) needed to fit all the words in exactly 30 spaces. Again, the total demerits for an entire paragraph is the sum of the d values for the individual lines.

Your program is to find the best selection of line breaks (blanks at which to end one line and begin the next) for a paragraph—one that minimizes the sum of the values of d over all the lines. Each input paragraph will consist of a sequence of words (non-blank characters) separated by single spaces (no line breaks). The output paragraphs are the same, with some of the blanks turned into newlines, and with a blank line between paragraphs. Do not insert extra spaces to justify the lines; leave them ragged, as in the examples below. You may assume that no input paragraph is longer than 1500 characters, including blanks, or contains more than 500 words. If there is more than one way to break a paragraph optimally, favor one that puts a line break sooner.

Example: (The backslashes at the ends of lines indicate line continuations, not real line breaks. In the actual input, there would be only three lines of input, with the backslashes and following newlines replaced by single blanks).

Input:

```
In preparation for the 1997 Pacific Regionals of\
the 22nd Annual ACM Scholastic Programming Contest,\
there will be a semi-informal programming contest on\
Saturday, 25 October 1997, from 1000--1530.
The square of the hypotenuse of a right triangle is\
equal to the sum of the squares of the two adjacent\
sides.
You’d not tolerate letting your participle dangle,\
so please effect the self-same respect for your\
geometric sides.
```

Output:

```
In preparation for the 1997
Pacific Regionals of the
22nd Annual ACM Scholastic
Programming Contest, there will
be a semi-informal programming
contest on Saturday, 25 October
1997, from 1000--1530.

The square of the hypotenuse of
a right triangle is equal to
the sum of the squares of the
two adjacent sides.

You’d not tolerate letting your
participle dangle, so please
effect the self-same respect
for your geometric sides.
```

Note. As you can see, some lines can be more than 30 characters long. Also, some lines may not be filled as much as they could be (the second line of output, for example) if that helps later lines.

8. Consider a class of expressions defined recursively as consisting of:

- single-letter variables,
- *applications*: constructs of the form `.(`$\mathcal{E}_1$ $\mathcal{E}_2$`)`, where $\mathcal{E}_1$ and $\mathcal{E}_2$ are themselves expressions according to these rules,
- *lambdas*: constructs of the form `/`x $\mathcal{E}$, where x is a single-letter variable, and $\mathcal{E}$ is another expression formed according to these rules.

A *beta-reduction* is a substitution in which a subexpression of the form `.(/`x $\mathcal{E}_1$ $\mathcal{E}_2$`)` (that is, an application whose first operand is a lambda) is replaced by $\mathcal{E}_1'$, where $\mathcal{E}_1'$ is derived from $\mathcal{E}_1$ by replacing all free instances of the variable x in $\mathcal{E}_1$ with $\mathcal{E}_2$. A free instance of a variable x in $\mathcal{E}_1$ is one that is not inside another lambda construct with the same variable.

Write a program that reads in such an expression (ignoring all whitespace in it), and prints out the result after applying all possible beta reductions. For example, the input

```
.(.(/x /y .(x y) /z z) .(a b))
```

allows a reduction that substitutes for `x`, giving

```
.(/y .(/z z y) .(a b))
```

and then for `y`, giving

```
.(/z z .(a b))
```

and finally for `z`, giving

```
.(a b)
```

Alternatively, one can use the sequence

```
     .(.(/x /y .(x y) /z z) .(a b))
==>  .(/y .(/z z y) .(a b))
==>  .(/y y .(a b))
==>  .(a b)
```

arriving at the same result. You may assume the process will terminate with the inputs you are given (in which case, you will get the same result, regardless of the sequence of reductions). Put no whitespace in your answers (except for a newline at the end).

Example 1:

Input	Output
`.(.(/x /y .(x y) /z z) .(a b))`	`.(ab)`

Example 2:

Input	Output
`.(a b)`	`.(ab)`

Example 3:

Input	Output
`.(/x .(x /x x) a)`	`.(a/xx)`

Note: This last example illustrates that one does not substitute for non-free occurrences of `x`.