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

Programming Contest **P. N. Hilfinger**
Fall 2010

2010 Programming Problems (revised 9/27/2010)

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 13 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).

This year, we have supplied some additional functions that you are free to use. The `contest-gcc` program, described below, will make implementations of these routines available to you. Their headers and documentation are in `contest.h` (for C/C++) and in the `contest` package (Java). You'll probably want to review them *before* the contest; there are links on the contest web site.

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.

There are two ways to submit solutions: by a command-line program, and over the web. Submit from the command line on the instructional machines. When you have a solution to problem number N that you wish to submit, use the command

`submit` N

from the directory containing N`.c`, N`.cc`, or N`.java`. Before actually submitting your program, `submit` will first compile it and run it on one sample input file. No submission that is sent after the end of the contest will count. You should be aware that `submit` takes some time before it actually sends a program. In an emergency, you can use

`submit -f` N

which submits problem N without compiling or running it.

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

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

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

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

You will be penalized for incorrect submissions that get past the simple test administered by `submit`, so be sure to test your programs (if you get a message from `submit` saying that it failed, you will *not* be penalized). All tests (for any language) will use the compilation command

`contest-gcc` N

followed by one or more execution tests of the form (Bourne shell):

`./`N `<` *test-input-file* `>` *test-output-file* `2>` *junk-file*

which sends normal output to *test-output-file* and error output to *junk-file*. The output from running each input file is then compared with a standard output file, or tested by a program in cases where the output is not unique. In this comparison, leading and trailing blanks are ignored and sequences of blanks are compressed to single blanks. Otherwise, the comparison is literal; be sure to follow the output formats *exactly*. It will do no good to argue about how trivially your program's output differs from what is expected; you'd be arguing with a program. Make sure that the last line of output ends with a newline. Your program must not send any output to `stderr`; that is, the temporary file *junk-file* must be empty at the end of execution. Each test is subject to a time limit of about 10 seconds. 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`.*` *our-libraries* `-lm`

For Java programs, it is equivalent to

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

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

`java -cp .:`*our-classes* `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 *our-libraries* and *our-packages* files and directories provide the additional tools we've provided this year. The files in `~ctest/submission-tests/`N, where N is a problem number, contain the input files and standard output files that `submit` uses for its simple tests.

All input will be placed in `stdin`. You may assume that the input conforms to any restrictions in the problem statement; you need not check the input for correctness. Consequently, you C/C++ programmers are free to use `scanf` to read in numbers and strings and `gets` to read in lines.

Terminology. The terms *free format* and *free-format input* indicate that input numbers, words, or tokens are separated from each other by arbitrary whitespace characters. By standard C/UNIX convention, a whitespace character is a space, tab, return, newline, formfeed, or vertical tab character. A *word* or *token,* accordingly, is a sequence of non-whitespace characters delimited on each side by either whitespace or the beginning or end of the input file.

Scoring. Scoring will be according to the ACM Contest Rules. You will be ranked by the number of problems solved. Where two or more contestants complete the same number of problems, they will be ranked by the *total time* required for the problems solved. The total time is defined as the sum of the *time consumed* for each of the problems solved. The time consumed on a problem is the time elapsed between the start of the contest and successful submission, plus 20 minutes for each unsuccessful submission, and minus the time spent judging your entries. Unsuccessful submissions of problems that are not solved do not count. As a matter of strategy, you can derive from these rules that it is best to work on the problems in order of increasing expected completion time.z

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. UTF-8 is a character encoding that allows the conversion of streams of 8-bit bytes to and from streams of Unicode *code points* (the integer representations of characters) in the range 0–0x10ffff, inclusive. The encoding is defined as follows:

Values up to...	Encoded as...	Description
01111111 ␣aaaaaaa	0aaaaaaaa	7-bit ASCII characters unchanged
00000111 11111111 ␣␣␣␣␣aaa bbcccccc	110aaabb 10cccccc	For example, '±' (0xb1) encodes as 0xc2b1.
11111111 11111111 aaaabbbb ccdddddd	1110aaaa 10bbbbcc 10dddddd	For example, the old symbol for the French franc (0x20a3) encodes as 0xe282b3.
00010000 11111111 11111111 ␣␣␣aaabb ccccdddd eeffffff	11110aaa 10bbcccc 10ddddee 10ffffff	For example, the Babylonic cuneiform symbol for the digit 3 (0x12417) encodes as 0xf0929097.

Each character must be represented by the smallest encoding that contains it. For example, the 7-bit ASCII characters must be represented in one byte; a byte sequence such as 0xc1a1—a two-byte representation of 0x61 ('a')—is not valid.

Your program is to input a purported UTF-8 sequence of bytes encoded as hexadecimal digits and output the sequence of Unicode code points they represent, also as a sequence of hexadecimal numerals, but without leading 0's, using the format in the example. In this format, each 1-byte encoding is written as a two hex-digit numeral, each 2-byte encoding as a 4-digit hex numeral, and each 3-byte encoding as a 6-digit numeral. Separate code points with single blanks. Assume the input contains only valid bytes (and always has an even number of hexadecimal digits). If you encounter an invalid code sequence, print a single question mark (also separated by blanks from surrounding code points), delete the first byte of the erroneous sequence, and skip the minimum number of additional input bytes needed to get to another, valid code point.

Example:

Input	Output
6120c2b1f0929097c1a1e282b3	61 20 b1 12417 ? 20b3

2. [From the UVa online problem set] This problem involves determining the number of routes available to an emergency vehicle operating in a city of one-way streets. Given the intersections connected by one-way streets in a city, you are to write a program that determines the number of different routes between each pair of intersections. A route is a sequence of one-way streets connecting two intersections.

Intersections are identified by non-negative integers. A one-way street is specified by a pair of intersections: (j, k) indicates a street going from intersection j to intersection k. We can model two-way streets by specifying two one-way streets: (j, k) and (k, j) indicate that there is a two-way street between intersections j and k. We will input such pairs are integers separated by whitespace, dispensing with the comma and parentheses.

Consider a city of four intersections connected by the four one-way streets (0,1), (0,2), (1,2), and (2,3). There is one route from intersection 0 to 1, two routes from 0 to 2 (the routes are $0 \Rightarrow 1 \Rightarrow 2$ and $0 \Rightarrow 2$), two routes from 0 to 3, one route from 1 to 2, one route from 1 to 3, one route from 2 to 3, and no other routes.

It is possible for an infinite number of different routes to exist. For example if the intersections above are augmented by the street (3,2), there is still only one route from 0 to 1, but there are infinitely many different routes from 0 to 2. This is because the street from 2 to 3 and back to 2 can be repeated yielding a different sequence of streets and hence a different route. Thus the route $0 \Rightarrow 2 \Rightarrow 3 \Rightarrow 2 \Rightarrow 3 \Rightarrow 2$ is different from $0 \Rightarrow 2 \Rightarrow 3 \Rightarrow 2$.

The input is a sequence of city specifications. Each specification begins with the number of one-way streets in the city followed by that many one-way streets given as pairs of intersections. In all cities, intersections are numbered sequentially from 0 to the "largest" intersection. All integers in the input are separated by whitespace. The input is terminated by end-of-file. There will never be a one-way street from an intersection to itself. No city will have more than 30 intersections.

For each city specification, print a square matrix of the number of different routes from intersection j to intersection k is printed. That is, if we denote the matrix M, then M_{jk} (row j, column k) is the number of different routes from intersection j to intersection k. Print each matrix in the format shown in the examples, preceded by the string "matrix for city k" ($k \geq 0$).

Print -1 to denote an infinite number of different paths between two intersections. The amount of whitespace used to separate entries in a row is irrelevant; you need not worry about justifying and aligning the output of the matrices.

Example:

Input	Output
7 0 1 0 2 0 4 2 4 2 3 3 1 4 3	matrix for city 0
5	0 4 1 3 2
0 2	0 0 0 0 0
0 1 1 5 2 5 2 1	0 2 0 2 1
9	0 1 0 0 0
0 1 0 2 0 3	0 1 0 1 0
0 4 1 4 2 1	matrix for city 1
2 0	0 2 1 0 0 3
3 0	0 0 0 0 0 1
3 1	0 1 0 0 0 2
	0 0 0 0 0 0
	0 0 0 0 0 0
	0 0 0 0 0 0
	matrix for city 2
	-1 -1 -1 -1 -1
	0 0 0 0 1
	-1 -1 -1 -1 -1
	-1 -1 -1 -1 -1
	0 0 0 0 0

3. In the game *Guess,* one side (the "Chooser") chooses a hidden sequence of four digits between 1–6, and the other side (the "Guesser") has a limited number of tries in which to guess the sequence. In each try, the Guesser proposes a sequence of digits, and the Chooser indicates how many of the proposed digits are correct, and how many others would be correct if they appeared in a different position. For example, if the Chooser selects the sequence '1443', and the Guesser proposes '2434', the Chooser would say that 1 position is correct and that 2 others would be correct if in different positions, since the Guesser has placed '4' in the right place, and has an additional '4' and a '3', but not in the right places.

You are to write an *interactive* Guesser program that outputs a sequence of up to eight guesses, and reads the resulting responses from a Chooser program. Your program succeeds if it makes a correct guess. It should stop when it guesses correctly.

Each guess consists of a single line (terminated, as usual, by a newline character) containing a sequence of four digits (1–6), written to the standard output. Each response from our Chooser program will then consist of two integers, c and d, separated by whitespace and terminated by a newline, where c is the number of correctly placed digits and d is the number of additional correct digits that incorrectly placed.

Warning: Be careful how you read input and write output. It is important *not* to attempt to read more than one line at a time, nor to read a response before outputting the guess with its newline. In addition, you should flush the output stream after each write, using the `flush` method or function appropriate for your chosen programming language. Violating any of these guidelines may cause the programs to hang.

Example:

Guesser Outputs	Chooser Outputs
`1111`	`1 0`
`2221`	`0 1`
`3313`	`1 1`
`4143`	`2 2`
`1443`	`4 0`

4. [From the UVa online problem set] Consider the sequence of all words formed entirely of lower-case letters and having the following properties:

- A word x appears before a word y if x is shorter than y.
- Any two words of the same length appear in alphabetical order.
- The list contains exactly the words whose letters appear in strictly increasing order (for example, 'a', 'ab', 'abc', but not 'ba' or 'bb').

To each word in this sequence, associate a positive integer index, starting with 1:

$$\begin{aligned} a &\rightarrow 1 \\ b &\rightarrow 2 \\ &\vdots \\ z &\rightarrow 26 \\ ab &\rightarrow 27 \\ ac &\rightarrow 28 \\ &\vdots \\ az &\rightarrow 51 \\ &\vdots \\ vwxyz &\rightarrow 83681 \end{aligned}$$

Your program is to read a series of lower-case words from one to five letters long, separated by whitespace. For each word read, if the word is invalid print the number 0, and otherwise print its index in the sequence.

Example:

Input	Output
`z a`	`26`
`cat`	`1`
`vwxyz`	`0`
	`83681`

5. If you've ever taken a chemistry course, you've seen this sort of thing denoting a chemical reaction:

$$2\mathrm{Na} + 2\mathrm{H_2O} = 2\mathrm{NaOH} + \mathrm{H_2}.$$

The symbols 'H', 'O', and 'Na' stand for various kinds of atom (hydrogen, oxygen, and sodium, respectively); each term (separated by '+' and '=') represents a molecule; the subscripted integer numerals after each atom (defaulting to 1) represent the number of that atom in the molecule (however, atoms can occur multiple times—as in acetic acid, '$\mathrm{CH_3COOH}$'—which contains two carbon atoms, two oxygen atoms, and four hydrogen atoms); and the integer numeric coefficients in front of a molecule (again defaulting to 1) represent the number of participating molecules of the attached type. You are to write a program that checks such equations for balance. For example, your program will accept

$$6\mathrm{H_2O} + 6\mathrm{CO_2} = 6\mathrm{O_2} + \mathrm{C_6H_{12}O_6},$$

but indicate that

$$\mathrm{O_2} + \mathrm{H_2} = \mathrm{H_2O}$$

is erroneous.

The input to your program will consist of equations of the form shown above, separated by whitespace, except that the equations themselves contain no whitespace and subscripted numerals are not written with subscripts. Each chemical element is denoted either by a single upper-case letter or by an upper-case letter followed by a lower-case letter.

For each equation, produce a line of output that echoes the equation followed either by the phrase "balances" or "does not balance" in the format shown in the examples below.

Example:

Input	Output
`6H2O+6CO2=6O2+C6H12O6`	`6H2O+6CO2=6O2+C6H12O6 balances`
`2Na+2H2O=2NaOH+H2  C6H12O6=3C2H2+3O2`	`2Na+2H2O=2NaOH+H2 balances` `C6H12O6=3C2H2+3O2 does not balance`

6. [Adapted from a 2009 high-school programming olympiad in St. Petersburg as translated by Misha Dynin] Alice has adopted a cat, which is very fond of sleeping. If he falls asleep, he sleeps without interruption for at least A hours. Moreover, the cat can't stay awake for more than B hours in a row. The cat could stay sleeping forever, but sometimes something interesting happens that the cat can't miss, such as Alice coming home from school, or him being fed.

Your task is to help the cat plan the daily routine so he doesn't miss interesting events. Each day the cat wants to live according to the same schedule.

The input to your program will consist of positive integers $A \leq 24$ and $B \leq 24$ (in hours) and up to 20 events (which occur daily). If the start of event is earlier than the end, that means that the event includes midnight. Each event has the form $HH{:}MM\text{-}HH{:}MM$, giving hours and minutes on a 24-hour clock.

If there is no solution, output the single word "No"; and otherwise a sleep schedule in the form of a sequence of non-overlapping "sleep events" in the same format as the input that, if repeated each day, conforms to the stated restrictions and has the cat awake during all the daily events. There is a 5-second time limit on this problem.

Example 1:

Input	Output

`3 4`
`07:00-07:15`
`19:00-19:59`
	`08:00-18:59`
	`20:00-06:59`

Example 2:

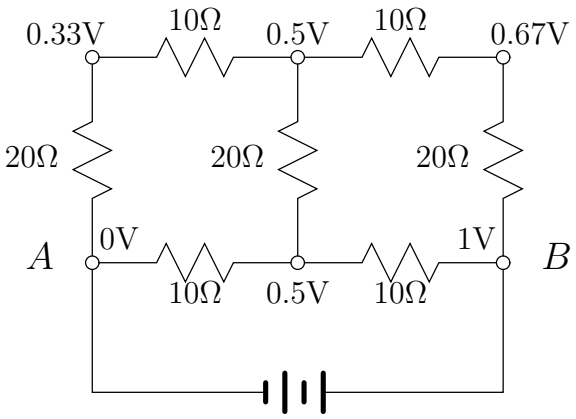
Input	Output

`3 4`
`07:00-08:01`
`11:00-11:09`
`19:00-19:59`
	`No`

Example 3:

Input	Output
`12 12` `23:00-01:00`	`01:07-22:13`

7. Consider a simple resistor grid and a battery, such as in the diagram below:



Given the values of the resistors, we'd like to determine the voltages at each of the nodes (the small circles), given that the battery establishes a voltage of 0 at A and 1 at B. The diagram shows the voltages that obtain for a particular choice of resistances. For those of you who might have forgotten, the net flow of current into a node must sum to 0 except at nodes A and B, and the current through a resistor with resistance R is V/R, where V is the voltage difference between the two terminals of the resistor.

The input to your program is in free format and consists of integers $M > 0$ and $N > 1$, giving the number of rows of nodes and the number of nodes in each row, respectively. Next will follow $2MN - M - N$ resistances (positive integers), giving the values of the resistors from left to right, top to bottom, as they are laid out in the diagram. The output will be M rows of N voltage values, giving the voltages at the nodes rounded to two decimal places (you already know the values of the first and last voltages in the last row!).

Example:

Input	Output
`2 3`	`0.33 0.50 0.67`
`10 10`	`0.00 0.50 1.00`
`20 20 20`	
`10 10`	

8. [A problem posed by Jon Bentley] The *diameter* of a set of points on the plane is the distance between its two most widely separated points. For example, the diameter of the set

$$\{(1,1),(0,0),(2,3),(3,4),(1,0)\}$$

is 5, the distance between the points (0,0) and (3,4). Given a set of points, you are to compute its diameter to within an accuracy of $\pm 5\%$. There is a 5-second execution-time limit on this problem.

The input to your program will consist of up to 100,000 pairs of integers giving x-y coordinates of points in free format (that is, separated by whitespace only, with no commas or parentheses). The output should be a single floating-point (or integer) number giving an approximate diameter.

Example:

Input	Output

`1 1 0 0 2 3`
`3 4 1`
`0`
	`5.0`