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

Programming Contest **P. N. Hilfinger**
Fall 2011

2011 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 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.

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 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/index.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. [M. Dynin] In 1976, Kenneth Appel and Wolfgang Haken proved that four colors suffice to color any *planar graph.* A somewhat more familiar formulation is that any map of contiguous regions on a plane may be colored with four or fewer colors so that any two regions that share a boundary of positive length (that is, that meet at more than a point) have different colors.

Your task is to write a program that finds such a coloring. The input will consist of a rectangular array of characters (up to 50×50), such as

```
0000
1223
1133
```

where the characters may be any graphic characters: all ASCII characters between '`!`' and '`~`' inclusive. A contiguous region in such a map is defined as any subset of the rectangle composed of the same character such that any character in the group can be reached from any other by a path from the first to the second that touches only other characters in the group in steps that move up, down, left, or right (but not diagonally). It is possible for two separate regions to use the same character; they count as distinct regions and may be assigned different colors.

In honor of Our Sponsor, the coloring assigns colors of red, green, blue, and yellow. For the input map above, the solution could be (there are 16 possible solutions):

```
RRRR
GBBY
GGYY
```

Example 1:

Input	Output
`0000`	`RRRR`
`1223`	`GBBY`
`1133`	`GGYY`

Example 2:

Input	Output
`///a`	`RRRB`
`a,,a`	`BYYB`
`,aa,`	`RGGR`

2. [M. Dynin] In Russia, bus tickets have six-digit serial numbers (from 000000 to 999999.) A ticket is considered *lucky* if the sum of the first three digits is equal to the sum of the last three digits (e.g. 068437: $0+6+8=4+3+7$) [Some suggest eating the lucky ticket, not that that's relevant to the problem.]

Your task is to count the number of lucky tickets in the general case. As input, you'll get a sequence of test cases in free format, each consisting of two numbers: the number of digits that make up the ticket's serial number (a positive even number) and the base of the ticket's serial number (a number between 2 and 10).

As output, you need to print the count of lucky tickets (in decimal) for each test case, in the format of the example. The answers will always be less than 2^{63}.

Example:

Input	Output
`2 10 4 10 6`	`There are 10 2-digit base 10 lucky numbers.`
`2`	`There are 670 4-digit base 10 lucky numbers.`
	`There are 20 6-digit base 2 lucky numbers.`

3. [M. Dynin] Morse code allows encoding of text with "dots" and "dashes". The dots and dashes making up each letter in the Morse alphabet weren't assigned randomly; the most frequently used letters in English have the shortest encodings: 'E' is one dot, 'I' is two dots, 'T' is one dash, and so on.

Your task is to reinvent Morse code: given a text corpus, come up with the shortest encoding for that text. Taking the smallest unit of time as the time required to transmit a dot, the rules of Morse code transmission are as follows:

- A dash is equal (in length) to three dots.
- The space separating successive dashes and dots in a given letter is one dot.
- The space between two letters of the same word is equal to three dots.
- The space between two words is equal to seven dots.

For example, in standard Morse code, the word "SOS" is encoded '`... --- ...` and takes 27 dot-times to transmit (5 for each 'S', 11 for 'O', and a total of 6 dot-times for the spaces between the letters). An optimal encoding of the single message "SOS" would assign a single dot to 'S' and either a single dash or two dots to 'O', for a transmission time of 11 dot-times. (In this problem, we will ignore the size of the encoding dictionary.) To encode the word "HELP" requires 21 dot-times ('H' could be '`.`', 'E' could be '`..`', 'L' could be '`-`', and 'P' could be '`.-`').

The input to your program will consist of a sequence of one-line messages consisting of upper-case letters and single blanks (with no leading or trailing blanks). For each line, your program will print just the total transmission time (in dot-times) for an optimal encoding tailored to that message.

Example:

Input	Output
`SOS`	`11`
`HELP HELP`	`49`

4. You are probably familiar with postfix notation for arithmetic expressions, where one writes the main operator of a subexpression at the end, after the operands, so that, for example '`(5+6)*7`' becomes '`5 6 + 7 *`' and `5+6*7` becomes '`5 6 7 * +`'. In this problem, you are to write a routine that does the reverse encoding, converting postfix expressions into equivalent infix expressions, with operators between the operands (or before in the case of unary operators such as negation), and with parentheses to indicate groupings other than those dictated by the usual rules for grouping of operands.

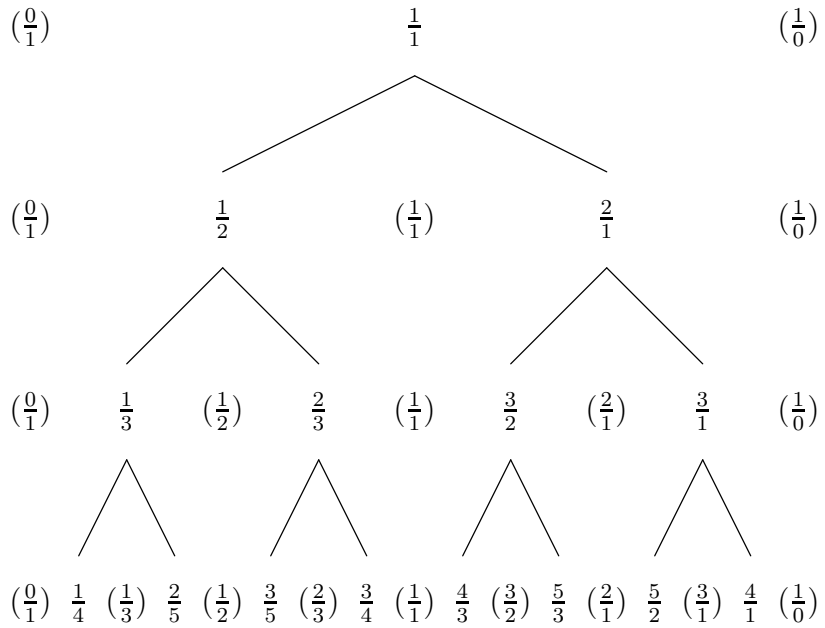
The input to your program will consist of postfix expressions, one per line, with numerals and operators all separated from each other by whitespace, and no leading or trailing whitespace. Numerals will be single decimal digits, and the possible operators are '`+`', '`-`', '`*`', '`/`', and '`_`' (which denotes unary minus).

The output is to consist of translated infix expressions, one per line, without spaces. Each expression should have the minimal number of parentheses needed to correctly translate the expression. For this purpose, the four binary operators group to the left, unary minus has highest precedence, followed by '`*`' and '`/`' (which have equal precedence), and then by '`+`' and '`-`' (which also have equal precedence). Unary and binary minus in infix expressions are both denoted with a dash. It is OK to have unary minus operators adjacent to other operators (including unary minus) so that `--3` and `2+-3` are valid infix expressions. Finally, do not use the associativity of addition and multiplication in deciding parentheses: `1+(2+3)` (which results from `1 2 3 + +`) is different from `1+2+3` (which results from `1 2 + 3 +`).

Example:

Input	Output
`5 6 + 7 *`	`(5+6)*7`
`5 6 7 * +`	`5+6*7`
`5 _ 7 * 6 / _`	`-(-5*7/6)`

5. [Aleksej Viktorchik, Leonid Shishlo] In number theory, the Stern-Brocot tree is a method of listing all positive rational numbers:



The tree may be created by an iterative process. It is easiest to describe as a list. Beginning with the list 0/1, 1/0 representing 0 and infinity respectively, one places between any two fractions the *mediant* of the fractions (the mediant of a/c and b/d is $(a+b)/(c+d)$). The first few steps of this process yield:

```
[0/1, 1/0]
[(0/1), 1/1, (1/0)]
[(0/1), 1/2, (1/1), 2/1, (1/0)]
[(0/1), 1/3, (1/2), 2/3, (1/1), 3/2, (2/1), 3/1, (1/0)]
```

(we show previously added fractions in parentheses). Each positive rational number appears exactly once in the tree of added fractions (rooted at 1/1). The position of a fraction in the tree can be specified as a path consisting of L(left) and R(right) moves along the tree starting from the root. Your task is to find the fraction reached by a given path.

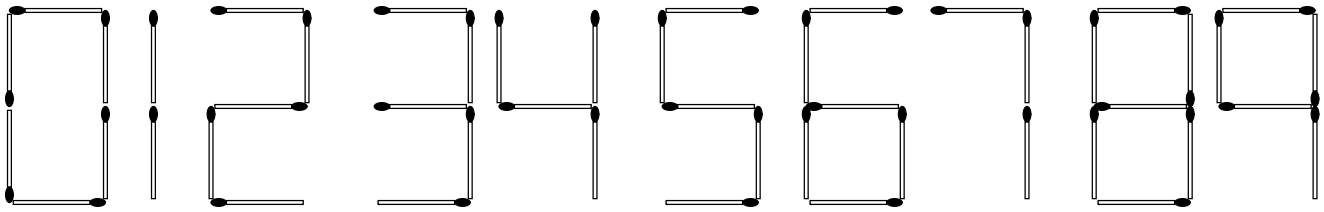
The input consists of a sequence of paths, one per line. Each path is a string consisting of from 0 to 40 capital letters 'L' and 'R'.

For each test case print the corresponding fraction in the form A/B, one fraction per line, as shown in the example. WARNING: actually building the tree level-by-level as suggested by the description above will take much too long for longer paths.

Example:

Input	Output
RL	3/2
RLR	5/3
RRL	5/2

6. [Mak Yan Kei] We can make digits with matches as shown below:



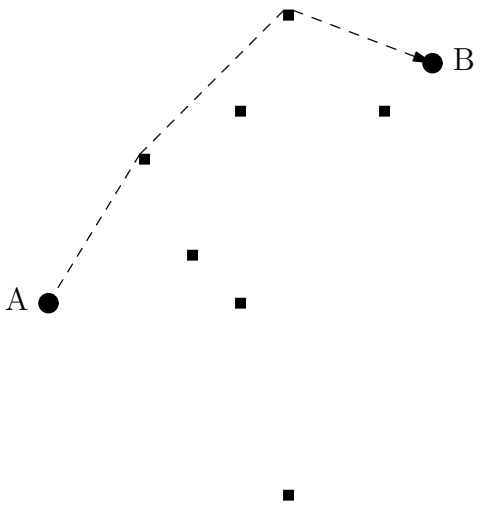
Given N matches, find the number of different numbers representable using the matches. We shall only make numbers greater than or equal to 0, so no negative signs should be used. For instance, if you have 3 matches, then you can only make the numbers 1 or 7. If you have 4 matches, then you can make the numbers 1, 4, 7 or 11. Superfluous leading zeros are not allowed (e.g. 001, 042, etc. are illegal; only 0 may start with 0).

The input is a sequence of positive integers in free format. For each N, $1 \leq N < 80$, output the number of different (non-negative) numbers representable with $\leq N$ matches. The answers will all be $< 2^{62}$.

Example:

Input	Output
3 4	2
2	4
	1

7. You are asked to provide planning software to help a robot navigate undetected through fields containing enemy surveillance equipment. The fields will contain a set of sensor devices, whose positions the robot can detect. Any pair of these sensors may have set up some detection beams allowing them to detect anything that comes between them. The robot cannot detect which pairs of sensors have such beams set up, so it must conservatively assume that all possible pairs do. The problem is to get from some given point to another along the shortest possible path that is guaranteed to avoid detection. For example, the diagram below shows the shortest undetectable path from A to B, with sensors shown as small square dots:



The input will consist of sets of integers in free format. Each set begins with an integer, $0 \leq N \leq 500$ giving the number of sensors. This is followed by $2(N + 2)$ integers, giving the x and y coordinates of A, B, and the N sensors, in that order.

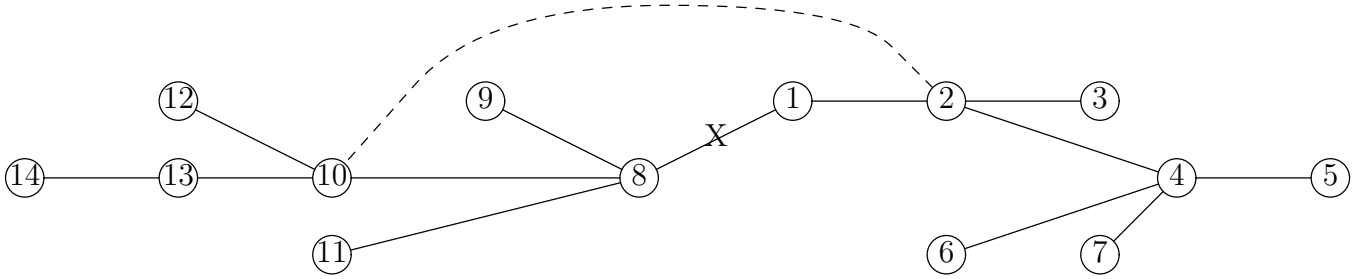
For each input set, output the length of the shortest path from A to B, rounded to the nearest integer, one line per set. Assume that neither point A nor B will be surrounded by sensors (that is, no group of sensors could ring either around with sensing beams). This implies that there will always be a path.

Example:

Input	Output
7 10 50 90 100 30 80 60 110 80 90 50 90 40 60 50 50 60 10 7 10 100 90 100 30 80 60 99 80 90 50 90 40 60 50 50 60 10	110 80

8. [2009 ACM Nordic Collegiate Programming Contest] The airline company NCPC Airways has flights to and from N cities, numbered from 1 to N, around the entire world. However, they only have $N-1$ different flights (operating in both directions), so in order to travel between any two cities you might have to take several flights. In fact, since the management has made sure that it's possible to travel between any pair of cities, there is exactly one possible sequence of flights a passenger has to take in order to travel between two cities (assuming he wants to use NCPC airlines exclusively).

Recently many of NCPC Airways frequent fliers have complained that they have had to change flights too often to get to their final destination. Since NCPC Airways doesn't want to lose their customers to other airline companies, but still keep the nice property of traveling between any of pair out of N cities while having only $N-1$ flights, they have decided to cancel one of their current flights and replace it with another flight. They may have to do it again if travel patterns change, and so would like a program that finds the best flight to cancel and the best new flight to add so that the maximum number of flight changes a passenger might have to make when traveling between any pair of cities in which NCPC Airways operates is minimized. The input will be constructed so that it is always possible to improve the maximum number of flight changes needed. For example, given the cities and flights denoted by solid lines, there are 6 intermediate stops needed to go from city #14 to city #5. Cutting out the flight marked 'X' and adding the dashed flight reduces this to 4 stops (we don't count arriving at the destination as a stop).



The input consists of any number of test cases in free format. Each test case consists of an integer N, $4 \leq N \leq 200$, giving the number of cities NCPC Airways operates in. Then follow $n-1$ pairs of integers specifying the flights. Each flight is given as a pair of city numbers a and b, with $a \neq b$ and $1 \leq a, b \leq N$.

For each test case print the pair of cities involved in the flight to cancel, the pair of cities in the flight to add, and the resulting maximum number of stops in any flights (the number of stops is one less than the number of flight segments; a direct flight has 0 stops). Use the format shown in the example on the next page. If there is more than one optimal solution, any one of them will be accepted.

Example:

Input	Output
4 1 2 2 3 3 4 14 1 2 1 8 2 3 2 4 8 9 8 10 8 11 4 5 4 6 4 7 10 12 10 13 13 14	Cancel flight 3:4 and add 2:4 for maximum of 1 stops. Cancel flight 1:8 and add 2:10 for maximum of 4 stops.