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

Programming Contest **P. N. Hilfinger**
Fall 2012

2012 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 14 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`, `set`, `unordered_map`, `unordered_set`, 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`, and `java.util` and their subpackages (but this year, *not* java.math). 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 (on torus.cs). 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 -Iour-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 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 (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.

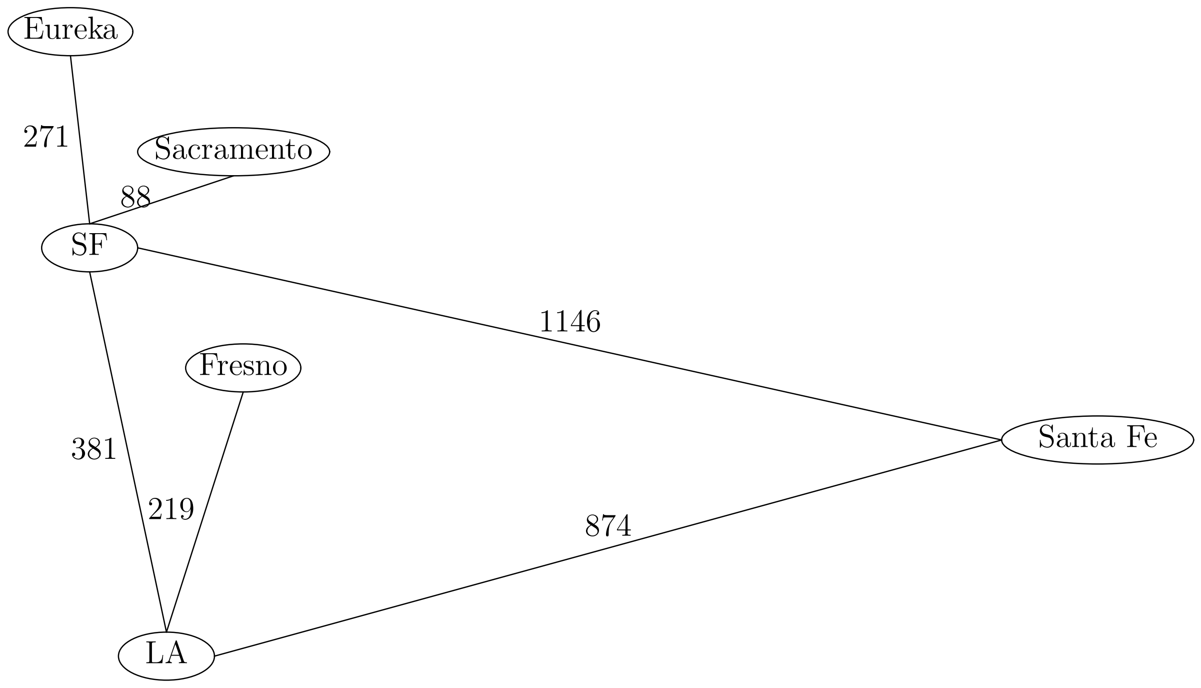
Protests. Should you disagree with the rejection of one of your problems, first prepare a file containing the explanation for your protest, and then use the `protest` command (without arguments). It will ask you for the problem number, the submission number (submission 1 is your first submission of a 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. A transportation planner is interested in knowing the maximum distance one must travel to get between any two points in a country (that is, the maximum of all the shortest distances between pairs of cities) and wants you to develop a program to do so. For example, consider this simple road system, where all roads are bidirectional:



the maximum distance would be that between Eureka and Santa Fe: 1417 miles.

The input to your program will consist of one or more sets of data in free format. Each set starts with an integer, N, indicating the number of road segments between cities. There then follow two city names (containing no embedded blanks) followed by a positive integer distance. There is at most one road between any pair of cities and no road from a city directly to itself. There will be no more than 100 cities. The last set is followed by the integer 0.

For each set, output the maximum distance in the format shown in the example.

Example:

Input	Output
`6`	`Set #1: 1417 miles`
`Eureka SF 271  SF Sacramento 88`	`Set #2: 180 miles`
`Fresno LA 219 Santa_Fe LA 874`	
`SF Santa_Fe 1146`	
`SF LA 381`	
`1 Berkeley Fresno 180`	
`0`	

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). **Restriction:** Java programmers may *not* use the classes `BigInteger` or `BigDecimal` from `java.math`.

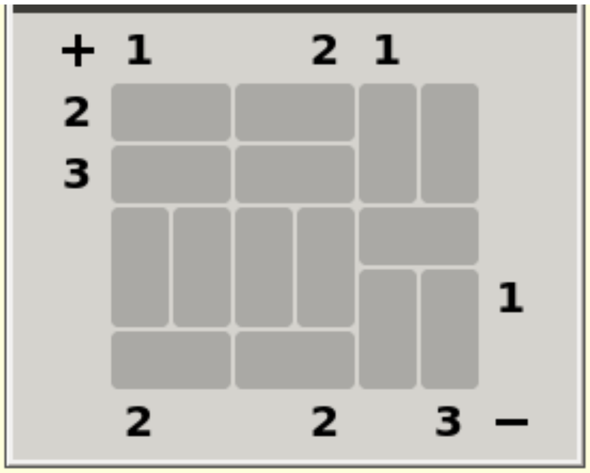
The input to your program will consist of any number of pairs of integers separated by slashes. All integers in the input will be in the range 1 to $2^{31} - 1$. You may assume that there will be no more than 10,000 pairs of integers.

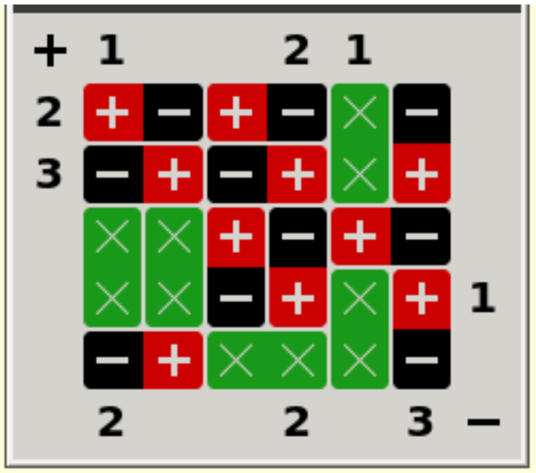
The output will be the product of all the input pairs (treated as fractions) in lowest terms, represented as a (possibly improper) fraction. Again, assume that the numerator and denominator of the result (in lowest terms) will be less than 2^{31}. (Warning: there is a 10-second time limit on execution).

Example:

Input	Output
`1000/1 1000/2 1000/3`	`2/7`
`1000/4`	
`1000/5`	
`1/2     4/7      5/1000`	
`4/1000 3/1000 2/1000`	
`1/1000`	

3. The puzzle game Magnets involves placing a set of domino-shaped magnets (or electrets or other polarized objects) in a subset of slots on a board so as to satisfy a set of constraints. For example, the puzzle on the left has the solution shown on the right[1]:





Each slot contains either a blank entry (indicated by 'x's), or a "magnet" with a positive and negative end. The numbers along the left and top sides show the numbers of '+' squares in particular rows or columns. Those along the right and bottom show the number of '−' signs in particular rows or columns. Rows and columns without a number at one or both ends are unconstrained as to the number of '+' or '−' signs, depending on which number is not present. In addition to fulfilling these numerical constraints, a puzzle solution must also satisfy the constraint that no two orthogonally touching squares may have the same sign (diagonally joined squares are not constrained).

The input to your program will consist of a series of puzzle specifications in free format. Each specification will consist of

- Positive integers $W \leq 7$ and $H \leq 7$, indicating the width and height of the board in squares. At least one will be even.
- Two sequences of W integers, indicating the numbers along the top (+) and bottom (-) edges respectively. Values of -1 indicate missing numbers.
- Two sequences of H integers, indicating the numbers along the left (+) and right (-) edges respectively (from top to bottom). Values of -1 again indicate missing numbers.
- H strings of W characters each, indicating the slots, or portions thereof, contained in each of the H rows (from top to bottom). Each string gives one row. Each character in a string is one of 'T', 'B', 'L', or 'R'. A 'T' indicates that the square is the top of a vertical slot, 'B' the bottom of a vertical slot, 'L' the left of a horizontal slot, and 'R' the right of a horizontal slot. See the example for the representation of the diagram above.

[1]Source: "Magnets" game from Simon Tatham's Portable Puzzle Collection.

The last puzzle specification will be followed by two 0s.

For each puzzle, print out a solution, using the format shown in the example below. Leave one blank line after each printed solution.

Example:

Input

```
6 5
1 -1 -1 2 1 -1
2 -1 -1 2 -1 3
2 3 -1 -1 -1
-1 -1 -1 1 -1
LRLRTT
LRLRBB
TTTTLR
BBBBTT
LRLRBB
3 4
2 -1 -1    -1 -1 2
-1 -1 -1   0 -1 -1
TTT BBB TLR BLR
0 0
```

Output

```
Set 1:
   +-+-x-
   -+-+x+
   xx+-+-
   xx-+x+
   -+xxx-

Set 2:
   +x+
   -x-
   +-+
   -+-
```

4. In an effort to develop a system to check for plagiarism, a client asks you to develop a program that measures how different two texts are from each other. Specifically, given two texts (strings) A and B, he'd like to be able to compute the minimum number of characters that need to be added to and/or removed from A to get B (or equivalently, the total number of characters that must be added to A and B to make them equal). For example, if A is "`written␣word`" and B is "`kitten␣purred`" we can first delete 4 characters from A to get "`itten␣rd`" and then add 5 new ones to get B, for a total of 9 characters. Alternatively, we can add 4 characters to A to get "`wrkitten␣wopurred`" and add 4 different ones to B to get the same thing. We'll say that the *dissimilarity* of A and B is 9.

Write a program to compute the dissimilarity of texts of up to 1000 characters. The input consists of pairs of strings, with each string on a single line. For each pair, print one line containing the dissimilarity between the texts.

Example:

Input	Output
`written word`	`9`
`kitten purred`	`33`
`sally sells sea shells by the seashore`	
`sarah sold salt sellers at the salt mines`	

5. [From the 1993 ACM ICPC] A subdivision consists of plots of land with each plot having a polygonal boundary. A surveyor has surveyed the plots, and has given the location of all boundary lines. That is the only information available, however, and more information is desired about the plots in the subdivision. Specifically, planners wish to classify the lots by the number of boundary line segments (B=3,4,5,...) on the perimeters of the lots.

Figures 1 and 2 show two hypothetical subdivisions. In Figure 1 there are 12 boundary line segments, and in Figure 2 there are 27. The sample input file below contains the data for these two test cases. The plot in the upper left hand corner of Figure 2 has one line running from (16,16) to (17,18) and another from (17,18) to (19,22). Thus this lot has a perimeter comprising 5 boundary line segments, though geometrically the lot is a 4-sided region. Similarly the perimeter of the plot in the upper left hand corner of Figure 1 comprises 6 boundary line segments, though the lot is pentagonal in shape.

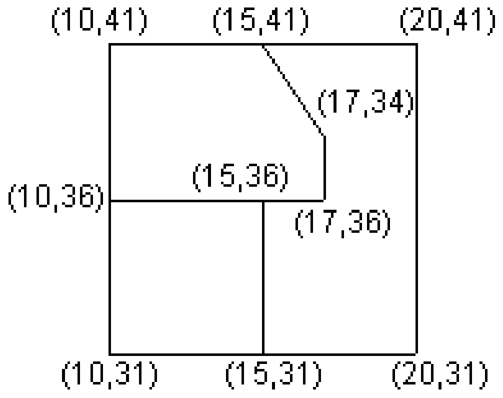


Figure 1

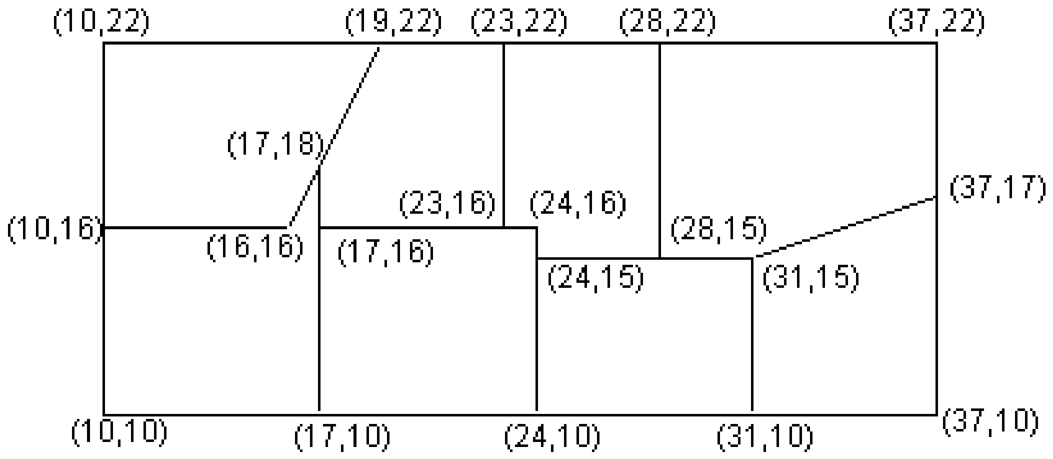


Figure 2

Write a program that will take as input the surveyor's data and produce as output the desired information about the nature of the lots in the subdivision. The input file consists of several data sets in free format. Each data set begins with the number of line segments ($4 \leq N \leq 200$) in the survey. There follow N groups of four integers each representing the Cartesian (x, y) coordinate pairs giving the end points of a boundary line segment. The input file is terminated with a 0.

Each data set corresponds to a rectangular subdivision whose boundaries are parallel to the x and y axes (as in Figures 1 and 2.) Boundary line segments in the input file do not extend past corners of lots. For example, in Figure 1 the surveyor must survey from the point (10,41) to (15,41) and from (15,41) to (20,41) rather than surveying the entire line (10,41) to (20,41). At least one boundary line segment in each lot lies on the subdivision's bounding rectangle.

For each data set, provide output listing the number of lots for each number of boundary line segment counts ($B = 3, 4, 5, \ldots$). Do not include in your output those values of B with no members. The output for each data set will begin with a line containing an appropriately labeled data set number, as shown in the example. Output for successive data sets will be separated by a blank line.

Example:

Input

```
12
10 41 15 41  15 41 20 41
10 36 15 36  15 36 17 36
10 31 15 31  15 31 20 31
10 41 10 36  10 36 10 31
15 41 17 34  17 34 17 36
15 36 15 31  20 41 20 31
27
10 22 19 22  19 22 23 22
23 22 28 22  28 22 37 22
10 16 16 16  17 16 23 16
23 16 24 16  24 15 28 15
28 15 31 15  10 10 17 10
17 10 24 10  24 10 31 10
31 10 37 10  10 22 10 16
10 16 10 10  17 18 17 16
17 16 17 10  24 16 24 15
24 15 24 10  23 22 23 16
28 22 28 15  31 15 31 10
37 22 37 17  37 17 37 10
16 16 17 18  17 18 19 22
31 15 37 17
0
```

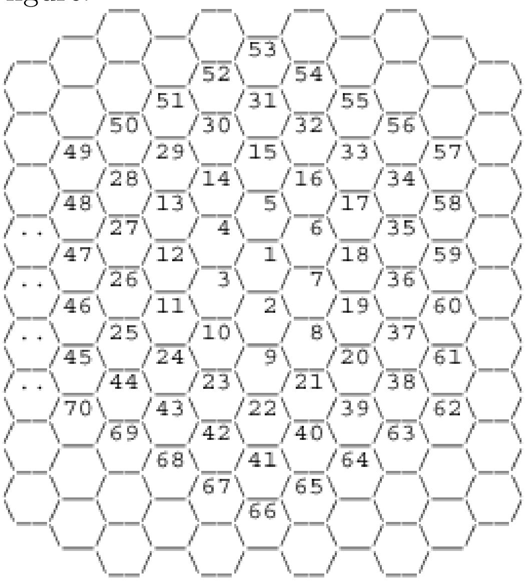
Output

```
Case 1:
  Lots with 4 surveyor's lines = 1
  Lots with 6 surveyor's lines = 1
  Lots with 7 surveyor's lines = 1
Total number of lots = 3

Case 2:
  Lots with 4 surveyor's lines = 1
  Lots with 5 surveyor's lines = 4
  Lots with 6 surveyor's lines = 3
Total number of lots = 8
```

6. [From the 1999 ACM ICPC] Professor B. Heif is conducting experiments with a species of South American bees that he found during an expedition to the Brazilian rain forest. The honey produced by these bees is of superior quality compared to the honey from European and North American honey bees. Unfortunately, the bees do not breed well in captivity. Professor Heif thinks the reason is that the placement of the different maggots (for workers, queens, etc.) within the honeycomb depends on environmental conditions, which are different in his laboratory and the rain forest.

As a first step to validate his theory, Professor Heif wants to quantify the difference in maggot placement. For this he measures the distance between the cells of the comb into which the maggots are placed. To this end, the professor has labeled the cells by marking an arbitrary cell as number 1, and then labeling the remaining cells in a clockwise fashion, as shown in the following figure.



For example, two maggots in cells 19 and 30 are 5 cells apart. One of the shortest paths connecting the two cells is via the cells $19 \rightarrow 7 \rightarrow 6 \rightarrow 5 \rightarrow 15 \rightarrow 30$, so you must move five times to adjacent cells to get from 19 to 30.

Professor Heif needs your help to write a program that computes the distance, defined as the number of cells in a shortest path, between any pair of cells. The input consists of several data sets in free format, each consisting of two integers a and b ($a, b \leq 10000$), denoting numbers of cells. The integers are always positive, except in the last set where $a = b = 0$. This last set terminates the input and should not be processed.

For each pair of numbers (a, b) in the input file, output the distance between the cells labeled a and b. The distance is the minimum number of moves to adjacent cells to get from a to b.

Example:

Input	Output
`19 30 0 0`	`The distance between cells 19 and 30 is 5.`

7. Consider a kind of *random walk* consisting of a sequence of integer values starting at 0 in which each value differs from its predecessor randomly by -1, 0, or 1. For example: $0, 1, 1, 2, 2, 3, 3, 3, 2, 1, 0, -1$. To describe such a sequence, we could list it in this fashion or give some kind of compressed description. This problem involves a particular compression scheme.

The notation has the following terms:

1. The symbol '+' indicates a step in which the value increases by 1.
2. The symbol '-' indicates a step in which the value decreases by 1.
3. The symbol '.' indicates a step in which the value does not change.
4. Any sequence of terms may be enclosed in parentheses, which serve to group them into a single term without changing their meaning.
5. Any of the preceding terms may be followed by a positive integer numeral, N, denoting N repetitions of that term.

For example, the sample sequence in the first paragraph may be written as "`(+.)3.-4`" because after the (implicit) initial 0, the sequence increases by 1 and then increases by 0 three times, then increases by 0 once, and then decreases by 1 four times.

Write a program to read such descriptions and print out the average value of the denoted sequence to two decimal places. For example, the average value of the sample sequence should be reported as 1.42. The input will consist of descriptions with the syntax described above in free format. The descriptions contain no whitespace. Echo each description, followed by the average value of the sequence it describes in the format illustrated in the example below.

Example:

Input	Output
`(+.)3.-4   (((++--)+)3-3)2`	`Average value of (+.)3.-4 is 1.42`
`(+-)`	`Average value of (((++--)+)3-3)2 is 1.78`
	`Average value of (+-) is 0.33`

8. Given a non-negative integer N as input, print $P(N)$, the position of N in the numerically ordered sequence of all non-negative integers with the same number of 1-bits as N in binary representation, numbering from 0. For example, $17 = 10001_2$ has two 1-bits in its representation. There are 6 numbers less than 17 that also have two 1-bits—3, 5, 6, 9, 10, and 12—so $P(17) = 6$.

Inputs will be non-negative decimal numbers less than 2^{63}, entered in free format on the standard input. The binary representation of each number will have 10 or fewer one-bits. Thus, all valid inputs will be representable using a Java `long` or a G++ `long long int`. For each input number, your program must output one line in the format shown in the example.

Example:

Input	Output
`3 17 7`	`3 is number 0 in the sequence of numbers with 2 1-bits.`
`8`	`17 is number 6 in the sequence of numbers with 2 1-bits.`
	`7 is number 0 in the sequence of numbers with 3 1-bits.`
	`8 is number 3 in the sequence of numbers with 1 1-bits.`