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

Programming Contest **P. N. Hilfinger**
Fall 2009

2009 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 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`, 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).

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.

This year, we have 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. One problem with this interface: at the moment it does not, unlike `submit`, pre-test your submission. At the moment, you can only do this on the instructional machines.

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 45 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`.*  -lm`

For Java programs, it is equivalent to

`javac -g` N`.java`

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

`java 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 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 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. [Miguel A. Revilla, from the Valladolid archives] The Baudot code, invented in 1870, was a distant ancestor of ASCII and was used to encode characters as 5-bit symbols. To extend its beyond 32 symbols, the code uses up-shift and down-shift modes as on a typewriter. In the Baudot code, each five bits transmitted must be interpreted according to whether they are up-shifted (figures) or down-shifted (letters). The bit pattern 11111 (31) represents the up-shift character, and the bit pattern 11011 (27) represents the down-shift. Everything between an up-shift and the subsequent down-shift (if any) is taken to be up-shifted, and all other characters (other than down-shift itself) are taken to be down-shifted. There are variations in the encoding; your job is to write a translator that accommodates them.

The input consists of two parts. The first two lines encode the character set. The first line contains the 32 down-shifted characters (with 00000 first) and the second line contains the 32 up-shifted characters. The characters up-shift and down-shift themselves are represented by blanks. The remainder of the input file consists of one or more messages encoded using the described Baudot code. Each message will be on a line in the input file. Each line will consist of a multiple of five 1's and 0's, with no characters between the bits.

The output should consist of one line of text for each message, containing the translated message in the format shown in the example. Make sure each line ends with an end-of-line sequence (newline on Unix).

Example:

Input	Output
`<T*O HNM=LRGIPCVEZDBSYFXAWJ UQK`	`DIAL:911`
`>5@9 %,.+)4&80:;3"$?#6!/-2' 71(`	`NOV 5, 8AM`
`10010011001100001001111110111000011111011110 1`	
`00110000110111100100111111000010011000100011001101111000001 11`	

2. The United States might avoid some of the quirks of its presidential election system if it adopted some kind of *Condorcet method* of voting. Here we look at one version. the idea is that each voter submits a ballot that *ranks* all the candidates in order of preference. Ties are allowed, and all the candidates the voter chooses not to rank are counted as being tied for last preference.

Voting proceeds by considering the candidates in pairs. For each pair of candidates A and B, we count how many ballots rank A higher than B and how many rank B higher than A. If one candidate wins in more of the ballots, then that candidate wins the A/B pairing. Then, if some candidate C wins *all* of his possible pairwise contests, then candidate C wins the election. Otherwise, as now, the choice goes to the House of Representatives.

As an example (adapted from Wikipedia), suppose that we have candidates Larry, Moe, Curly, and Bob and the ballots cast fall into the following categories:

Percent of Ballots	Ranking
40%	1. Larry; 2. Moe; 3. Curly; 4: Bob
24%	1. Moe; 2. Curly; 3. Bob; 4. Larry
16%	1. Curly; 2. Bob; 3. Moe; 4. Larry
15%	1. Bob; 2. Curly; 3. Moe; 4. Larry
5%	1. Bob, Curly (tie); 2. Moe

So, 40% of voters prefer Larry to Moe, Moe to Curly, and Curly to Bob. Likewise, 5% of voters prefer Bob and Curly to Moe, have no preference between Bob and Curly, and don't rank Larry at all (meaning that he ranks last). Now consider all the possible pairings:

Pair	Winner
Larry (40%) vs. Moe (60%)	Moe
Larry (40%) vs. Curly (60%)	Curly
Larry (40%) vs. Bob (60%)	Bob
Moe (64%) vs. Curly (36%)	Moe
Moe (64%) vs. Bob (36%)	Moe
Curly (80%) vs. Bob (15%)	Curly

Since Moe wins all his pairings, Moe wins the election. On the other hand, it is possible to have cases where there is no such winner. Naturally, this happens when all votes are ties. But there can also be cycles (Condorcet paradoxes) like this:

Percent of Ballots	Ranking
30%	1. Larry; 2. Moe; 3. Curly; 4: Bob
30%	1. Moe; 2. Curly; 3. Larry; 4. Bob
30%	1. Curly; 2. Larry; 3. Moe; 4. Bob
10%	1. Bob; 2. Curly, Larry, Moe (tie)

where Larry beats Moe, Moe beats Curly, and Curly beats Larry (and Bob loses to everyone). In this case, there is no winner.

Your program is to accept a sequence of words in free format, representing the ballots in a single election. First comes an integer $N > 1$, giving the number of candidates. Next

come one or more N-tuples indicating the ranking of each of the N candidates. Each ranking is either a positive integer (smaller is better), or a hyphen ('-') indicating no vote for a given candidate. There is no requirement that the ranking numbers be limited to the range $1..N$, nor are there restrictions against gaps (you can rank candidates 2, 4, 7, and 5, for example).

The output consists of either the sentence "`Candidate #`N `wins.`" (the first candidate is #1) or "`Election goes to the House.`" if there is no winner.

Example 1:

Input	Output

`4`
`1 2 3 4 1 2 3 4 1 2 3 4`
`1 2 3 4 1 2 3 4 1 2 3 4`
`4 1 2 3 4 1 2 3`
`4 1 2 3 4 1 2 3`
`4 3 1 2 4 3 1 2 4 3 1 2`
`1 2 3 4 1 2 3 4`
`4 3 2 1 4 3 2 1 4 3 2 1`
`- 2 1 1`
	`Candidate #2 wins.`

Example 2:

Input	Output

`4`
`1 2 3 4 1 2 3 4`
`1 2 3 4`
`3 1 2 4 3 1 2 4`
`3 1 2 4`
`2 3 1 4 2 3 1 4 2 3 1 4`
`5 5 5 2`
	`Election goes to the House.`

3. [From the Valladolid archives] A certain communication protocol sends data messages with a restricted syntax.

0. There are only 16 symbols used in these messages, which we will denote with the characters 'p' through 'z' and upper-case characters 'N', 'C', 'D', 'E', and 'I'.
1. Every character from p through z is a correct message.
2. If σ is a correct message, then so is Nσ.
3. If σ and τ are correct messages, then so are C$\sigma\tau$, D$\sigma\tau$, E$\sigma\tau$ and I$\sigma\tau$.
4. Only messages conforming to Rules 0–3 are valid.

You are asked to write a program that checks if messages satisfy the syntax rules given in Rule 0–4.

The input consists of a number of purported messages containing only whitespace and the 16 characters listed above, in free format. If necessary, you may assume that each sentence has at most 256 characters and at least 1 character.

The output consists of echoes of the messages, followed by one of the words YES for each well-formed message or NO for invalid messages, with one message per line in the format shown below.

Example:

Input	Output
`Cp Isz`	`Cp NO`
`NIsz`	`Isz YES`
`Cqpq`	`NIsz YES`
	`Cqpq NO`

4. A certain municipality has set up some automated measurement stations along a street to detect speeders. The equipment consists of a number of independent infrared speed detectors that each yield a reading taken from different locations to take into account the differing sizes and positions of passing cars. Each reading is actually a range of possible speeds, reflecting the uncertainty of the measurement. The problem is to come up with a combined result that represents the smallest range that is consistent with the largest possible number of individual measurements. Suppose there are N devices and that for a given car, each reports an interval of possible speeds (a_i, b_i), where $a_i < b_i$. We'll call the range (a, b) a *common interval* for these measurements if $a < b$, a is maximum of the a_i, and b is the minimum of the b_i.

So, for example, the common interval of the measurements $\{(10, 15), (12, 17), (8, 14)\}$ is $(12, 14)$. On the other hand, the set $M_2 = \{(10, 15), (14, 19), (8, 12)\}$ has no common interval. When a set has no common interval, a simple rule would be to find a common interval for a *largest possible subset* that contains $> N/2$ measurements (let's call such an interval a *consensus interval.*) For example, the intervals $(10, 12)$ and $(14, 15)$ are common intervals for two different subsets of M_2 of size 2. Since we want to err on the side of innocence, in such cases we'll choose the interval with the smaller starting point. If the only subsets with a common interval have $\leq N/2$ members, then there is no consensus interval.

Your task is to write a program that takes sets of measurements and reports the consensus intervals for each, if they exist. The input consists of one or more sets of data in free format. Each set begins with a single integer, $N > 0$, giving the number of intervals in the set. This is followed by $2N$ floating-point numbers, each pair representing the range of possible speeds reported by a single device.

Print one line of output for each set in the format shown in the example. For each consensus, give both the interval and the the maximum number of measurements for which it is a consensus. Print floating-point numbers rounded to one digit after the decimal point.

Example:

Input	Output
`3 10 15 12 17 8 14`	`Set 1: (12.0, 14.0) from 3 measurements.`
`3 10 15`	`Set 2: (10.0, 12.0) from 2 measurements.`
`14 19`	`Set 3: No consensus.`
`8 12`	
`4 10.1 15.2 14 19 20 25 22 24`	

5. [From the Valladolid archives] In Contract Bridge, players must assess the strength of their hands prior to bidding against one another. Most players use a point-count scheme, such as the following:

1. Each ace counts 4 points. Each king counts 3 points. Each queen counts 2 points. Each jack counts one point.
2. Subtract a point for any king of a suit in which the hand holds no other cards.
3. Subtract a point for any queen in a suit in which the hand holds only zero or one other cards.
4. Subtract a point for any jack in a suit in which the hand holds only zero, one, or two other cards.
5. Add a point for each suit in which the hand contains exactly two cards.
6. Add two points for each suit in which the hand contains exactly one card.
7. Add three points for each suit in which the hand contains no cards.

A suit is *stopped* if it contains an ace, or if it contains a king and at least one other card, or if it contains a queen and at least two other cards.

When considering whether to open the bidding, assuming all previous players have passed, the three most common possibilities are:

- If the hand evaluates to fewer than 14 points, then the player passes.
- Otherwise, one opens the bidding in "no trump" if the hand evaluates to 16 or more points ignoring rules 5, 6, and 7 and if all four suits are stopped.
- Otherwise, one opens the bidding in a suit. Bidding is always opened in the highest-ranking of the suits with the most cards. For this purpose, we rank spades ('S') highest, then hearts ('H'), diamonds ('D'), and lastly, clubs ('C').

As you've probably guessed, you are to write a program that determines the bid recommended by this system. Input to your program consists of a series of sets of 13 cards in free format. Each card consists of two characters. The first represents the rank of the card: 'A', '2', '3', '4', '5', '6', '7', '8', '9', 'T', 'J', 'Q', 'K'. The second represents the suit of the card: 'S', 'H', 'D', or 'C'.

For each line of the input, print one line containing the recommended bid, either "PASS", "BID *suit*", where suit is "S", "H", "D", or "C," or "BID NO-TRUMP".

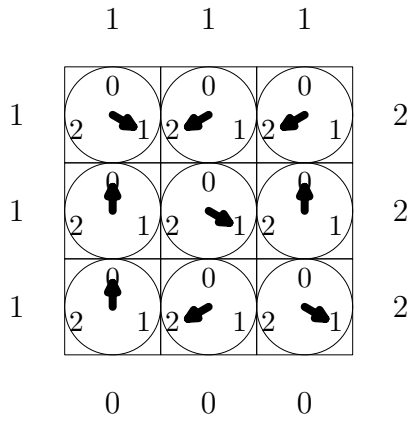
For the first hand in the example below, the evaluation starts with 6 points for the two kings, 4 for the ace, 6 for the three queens, and one for the jack. To this tally of 17 points, we add 1 point for having only two cards in spades, and subtract 1 for having a queen in spaces with only one other card in spades. The resulting 17 points is enough to justify opening in a suit. The evaluation for no-trump is 16 points, since we cannot count the

one point for having only two spades. We cannot open in no-trump, however, because the hearts suit is not stopped. Hence we must open bidding in a suit. The two longest suits are clubs and diamonds, with four cards each, so the possible outputs are "BID C", or "BID D", and we choose the latter as being higher ranking, as indicated previously.

Example:

Input	Output
KS QS TH 8H 4H AC QC TC 5C KD QD JD 8D AC 3C 4C AS 7S 4S AD TD 7D 5D AH 7H 5H	BID D BID NO-TRUMP

6. It is widely unknown that the ancient Egyptians established, for a time, a flourishing colony in what is now Burkina Faso (mostly in the Sahel, where it is drier). They left behind various buried cities of archaeological interest in which one intrepid researcher has discovered a collection of storage buildings whose rooms are locked by an unusual form of puzzle lock. Each such lock consists of a square array of three-position dials, with a sequence of carved digits (0–2) around the periphery, like this (where I have substituted digits 0–2 for the symbols that originally appeared):



The lock is unlocked when the dials are set, as they here, so that the number on each dial is equal to the sum of the numbers of its four neighbors (vertical and horizontal) modulo 3 (where neighbors can be either other dials or the fixed, carved numbers around the edge.) Your job is build a handy calculator that, given the carved numbers, computes the appropriate dial settings to open the lock.

The input consists of a number of sets of data, in free format. Each set begins with an integer, $1 \leq N < 8$, giving the number of rows and columns of dials. Next come $4N$ integers, indicating the carved numbers left-to-right across the top, then top-to-bottom down the right, then left-to-right across the bottom, the top-to-bottom down the left.

The output consists of the input set number and the dial settings, displayed in the format shown in the example. Not all possible puzzles have solutions, but we will only give you ones that do.

Example:

Input	Output
3 1 1 1 2 2 2 0 0 0 1 1 1 3 0 1 2 1 2 1 2 0 1 2 1 0	Lock 1: 1 1 1 --- 1 | 1 2 2 | 2 1 | 0 1 0 | 2 1 | 0 2 1 | 2 --- 0 0 0 Lock 2: 0 1 2 --- 2 | 0 2 2 | 1 1 | 2 2 0 | 2 0 | 2 1 0 | 1 --- 2 0 1

7. Hitori is a simple puzzle game played on an $N \times N$ grid of positive integers. The idea is simply to cross out integers in the grid (as few as possible) so that:

1. Any given number appears at most once in any given column or row;
2. The remaining (unerased) numbers form a single connected group of squares, so that any unerased square is reachable from any other unerased square by a path of unerased squares that are adjacent horizontally or vertically;
3. If a square is erased, its adjacent horizontal and vertical neighbors are not erased.

So, given the initial grid (1), below, grid (2) is a solution. The other grids are not solutions for various reasons: (3) has two adjacent squares erased, (4) erases more squares than needed, and (5) breaks the unerased squares into unconnected groups. Of course, (1) itself is not a solution because it has duplicate numbers in several rows and columns.

1	2	5	3
4	2	2	2
3	6	3	5
2	4	5	4

(1)

1	2	5	3
4	~~2~~	2	~~2~~
~~3~~	6	3	5
2	4	~~5~~	4

(2)

1	2	5	3
4	~~2~~	2	~~2~~
3	6	~~3~~	5
2	4	~~5~~	4

(3)

1	2	~~5~~	3
4	~~2~~	2	~~2~~
~~3~~	6	3	5
2	4	~~5~~	4

(4)

1	2	~~5~~	3
4	~~2~~	2	~~2~~
~~3~~	6	3	5
2	4	5	4

(5)

You are to write a program that, given such a grid, finds a solution satisfying the stated criteria. The input will consist of one or more sets in free format. Each set begins with an integer, $N \leq 10$, given the size of the grid, followed by N^2 positive integers less than 100, indicating the contents of the grid cells in order left-to- right, top-to-bottom (row-major order). The last set is followed by a single integer 0.

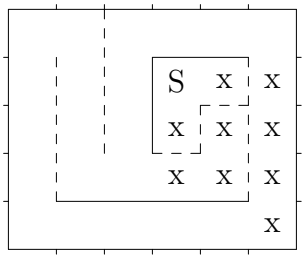
For each set, the output shows the grid with the erased cells parenthesized, using the format shown in the example.

Example:

Input	Output

4
  1 2 5 3
  4 2 2 2
  3 6 3 5
  2 4 5 1
2 1 2 3 4
0
	Grid 1:
	  1 2 5 3
	  4 (2) 2 (2)
	 (3) 6 3 5
	  2 4 (5) 1
	
	Grid 2:
	  1 2
	  3 4

8. In the distant principality of Egregia, evil Prince Mal has a habit of throwing his political opponents into a maze together with a member of his elite Royal Sniper Corps. If a political prisoner is unable to reach the maze exit (without being shot) within a certain length of time, the floor of the maze gives way, and he falls into a stake-studded pit. The walls of the maze have various heights, and the sniper is able to see (and assassinate) anyone taller than all the walls between the sniper and that unfortunate victim. We assume that the sniper can kill a victim in any square where he can see at least three corners of the room. He can see a corner if a line from the sniper to that corner does not first encounter a high wall (one taller than the victim), except possibly at corners where at most one wall is a high wall. The example below shows a maze in which the prisoner, who always starts in the lower left square, can reach the exit (which is always the upper right square) in 15 steps. Dashed lines indicate low walls; solid lines indicate high ones (the prisoner may not walk through either kind of wall). The sniper's location is marked 'S' and all other squares that the sniper can hit are marked 'x'. In this example, the sniper can hit the lower-right square because he can see all but its lower-left corner. He can't hit the upper-right square, because he can only see its two lower corners. As a member of the local resistance, you are tasked with providing a cell phone app for members of the opposition that directs them on the shortest safe path through the maze (fortunately, the Prince, being rather old-fashioned, is unaware of the capabilities of modern personal electronics).



The maze (as is so often the case in these programs) is a grid of 1×1 squares, with walls running between adjacent squares. The input to your program will consist of a sequence of integers in free format. First come two integer W and H, indicating the width and height of the maze. Next come two more integers, $0 \le S_x < W$ and $0 \le S_y < H$, indicating that the position of the sniper is the middle of square (S_x, S_y), where square $(0, 0)$ is the lower left corner (and therefore the entrance), and $(W-1, H-1)$ is the upper right (and therefore the exit). Next come descriptions of the walls, each in the form of (free-format) triple: one of

`H` y x_0 x_1 S
`V` x y_0 y_1 S

indicating either a horizontal segment across the top of squares (x_0, y) through (x_1, y), where $0 \le x_i < W$ and $0 \le y < H-1$, or a vertical segment along the right edge of squares (x, y_0) through (x, y_1), where $0 \le x < W-1$ and $0 \le y_i < H$. S is either 0, indicating a low wall segment (too high to jump across, but low enough to allow the sniper to see you), or 1, indicating a wall segment that hides you from the sniper. The walls around the outside of the maze are implicit.

The output of your program is simply the shortest number of steps (a step is a move from one square to a horizontal or vertical neighbor) to the exit via a safe route, or 'Sorry', if there is no safe route.

Example 1:

Input	Output
6 5 3 3 V 0 1 3 0 H 0 1 4 1 H 3 3 4 1 H 1 3 3 0 H 2 4 4 0 V 1 2 4 0 V 4 1 3 0 V 2 2 3 1 V 3 2 2 0	15

Example 2:

Input	Output
6 5 3 3 V 0 1 3 0 H 0 1 4 0 H 3 3 4 0 H 1 3 3 0 H 2 4 4 0 V 1 2 4 0 V 4 1 3 0 V 2 2 3 0 V 3 2 2 0	Sorry