1 Recursive Solvers

For today's practice, we look at the use of recursion to solver various constraint satisfaction problems (for example, typical puzzles). In all of these cases, the goal will be to construct a solution to some problem recursively, by effectively trying all possible solutions, but with the use of commonsense rules to prune partial solutions that must be infeasible. In most cases, it will be difficult for us to give a clean asymptotic analysis of the maximum running time, but we may hope that it will be fast enough in practice for given data sets.

As noted from the success percentages in the table below, these problems are solvable, but typically separate the top-flight teams from the field.

The judge's data for all of these problems are loaded onto turing so that you can use our submit script to test your solutions.
(See `http://cs.slu.edu/~goldwasser/icpc/resources/` for details and
`http://cs.slu.edu/~goldwasser/cgi-bin/icpc` for links to problem statements.)

shortname	Problem	contest	success	notes
bounce	Bounce	2012 Mid-Central	6.6%	
cubes	Letter Cubes	2013 Mid-Central	4.3%	
sudominoku	Su-domino-ku	2011 Mid-Central	2.1%	
hex	Hex Tile Equations	2008 Mid-Central	1%	

Problem H: Bounce

Source file: `bounce.{c, cpp, java}`
Input file: `bounce.in`

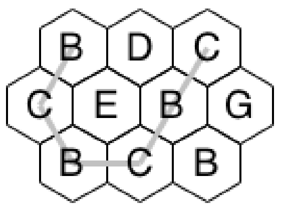

Figure 1

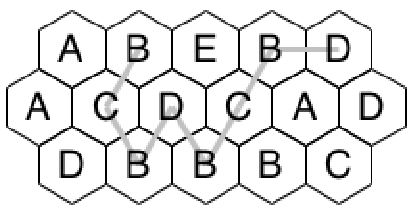

Figure 2

A puzzle adapted from a 2007 Games Magazine consists of a collection of hexagonal tiles packed together with each tile showing a letter. A *bouncing path* in the grid is a continuous path, using no tile more than once, starting in the top row, including at least one tile in the bottom row, and ending in the top row to the right of the starting tile. *Continuous* means that the next tile in a path always shares an edge with the previous tile.

Each bouncing path defines a sequence of letters. The sequence of letters for the path shown in Figure 1 is BCBCBC. Note that this is just BC repeated three times. We say a path has a *repetitive pattern of length n* if the whole sequence is composed of two or more copies of the first n letters concatenated together. Figure 2 shows a repetitive pattern of length four: the pattern BCBD repeated twice. Your task is to find bouncing paths with a repetitive pattern of a given length.

In each grid the odd numbered rows will have the same number of tiles as the first row. The even numbered rows will each have one more tile, with the ends offset to extend past the odd rows on both the left and the right.

Input: The input will consist of one to twelve data sets, followed by a line containing only 0.

The first line of a data set contains blank separated integers *r c n* , where *r* is the number of rows in the hex pattern ($2 \le r \le 7$), *c* is the number of entries in the odd numbered rows, ($2 \le c \le 7$), and *n* is the required pattern length ($2 \le n \le 5$). The next *r* lines contain the capital letters on the hex tiles, one row per line. All hex tile characters for a row are blank separated. The lines for odd numbered rows also start with a blank, to better simulate the way the hexagons fit together.

Output: There is one line of output for each data set. If there is a bouncing path with pattern length *n*, then output the pattern for the *shortest* possible path. If there is no such path, output the phrase: **`no solution`**. The data sets have been chosen such that the shortest solution path is unique, if one exists.

Example input:

```
3 3 2
 B D C
C E B G
 B C B
3 5 4
 A B E B D
A C D C A D
 D B B B C
3 3 4
 B D C
C E B G
 B C B
3 4 4
 B D H C
C E F G B
 B C B C
0
```

Example output:

```
BCBCBC
BCBDBCBD
no solution
BCBCBCBC
```

Last modified on October 18, 2012.

Problem E: Letter Cubes

Source file: `cubes.{c, cpp, java}`
Input file: `cubes.in`

This problem is based on a puzzle by Randall L. Whipkey.

In the game of Letter Cubes, there are a set of cubes, with each face of each cube having a letter of the alphabet, such that no letter appears more than once within the entire set. The maximum number of cubes is 4, allowing for up to 24 of the 26 letters of the alphabet to occur.

Words are formed by rearranging and turning the cubes so that the top letters of all the cubes together spell a word. The 13 words below have been made using a particular set of cubes.

CLIP
CLOG
CONE
DISH
FAZE
FURL
MARE
MOCK
QUIP
STEW
TONY
VICE
WARD

Only 23 distinct letters were used in the above words, so we will tell you the extra information that a B is included on one cube. Can you now determine the letters on each cube? For the above set of words, there is indeed a unique set of cubes. We will state this solution in canonical form as

ABCHTU DEKLQY FGIMNW OPRSVZ

Note that the letters on each individual cube are stated as a string of characters in alphabetical order, and the four 6-letter strings representing the four cubes are also listed in alphabetical order.

A simpler example relies on two cubes, forming the following 11 two-character strings (although the puzzles are more fun when the strings are actual words, they do not need to be):

```
PI
MU
HO
WE
WO
BE
MA
HI
RE
AB
PY
```

The only solution for the two cubes forming these strings is

```
AEIOUY BHMPRW
```

The same two cubes could be determined without the last pair PY being listed, as long as you were told that there was a Y on one cube. Your job is to make similar deductions.

Input: The input will contain from 1 to 20 datasets. The first line of each dataset will include a positive integer n ($6 \le n \le 30$) and a character c, described below. The next n lines will each contain a string of uppercase letters. Each string will be the same length, call it k, with $2 \le k \le 4$. Following the last dataset is a line containing only 0.

Returning to the issue of the special character, c, on the first input line for each dataset, there will be two cases to consider. Recall that the implicit set of k cubes must use $6*k$ distinct letters on their collective faces If all $6*k$ of those letters appear within the set of strings, then the character c on the first line of input is a hyphen, '-'. Otherwise, the strings have been chosen so that only one letter on the cubes does not appear. In this case, the character c on the first line of input will be that undisplayed letter. (For example, the `B` in our opening puzzle.)

Output: There is one line of output for each dataset, containing a 6-letter string for each cube, showing the letters on the faces of that cube. Each of those strings should have its letters in alphabetical order, and the set of strings should be given in alphabetical order with respect to each other, with one space between each pair. *We have chosen datasets so that each has a unique solution.*

Example input:	**Example output:**

```
13 B
CLIP
CLOG
CONE
DISH
FAZE
FURL
MARE
MOCK
QUIP
STEW
TONY
VICE
WARD
11 -
PI
MU
HO
WE
WO
BE
MA
HI
RE
AB
PY
10 Y
PI
MU
HO
WE
WO
BE
MA
HI
RE
AB
0
```
	```
	ABCHTU DEKLQY FGIMNW OPRSVZ
	AEIOUY BHMPRW
	AEIOUY BHMPRW
	```

Problem D: Su-domino-ku

Source file: `sudominoku.{c, cpp, java}`
Input file: `sudominoku.in`

As if there were not already enough sudoku-like puzzles, the July 2009 issue of Games Magazine describes the following variant that combines facets of both sudoku and dominos. The puzzle is a form of a standard sudoku, in which there is a nine-by-nine grid that must be filled in using only digits 1 through 9. In a successful solution:

- Each row must contain each of the digits 1 through 9.
- Each column must contain each of the digits 1 through 9.
- Each of the indicated three-by-three squares must contain each of the digits 1 through 9.

For a su-domino-ku, nine arbitrary cells are initialized with the numbers 1 to 9. This leaves 72 remaining cells. Those must be filled by making use of the following set of 36 domino tiles. The tile set includes one domino for each possible pair of unique numbers from 1 to 9 (e.g., 1+2, 1+3, 1+4, 1+5, 1+6, 1+7, 1+8, 1+9, 2+3, 2+4, 2+5, ...). Note well that there are not separate 1+2 and 2+1 tiles in the set; the single such domino can be rotated to provide either orientation. Also, note that dominos may cross the boundary of the three-by-three squares (as does the 2+9 domino in our coming example).

To help you out, we will begin each puzzle by identifying the location of some of the dominos. For example, Figure 1 shows a sample puzzle in its initial state. Figure 2 shows the unique way to complete that puzzle.

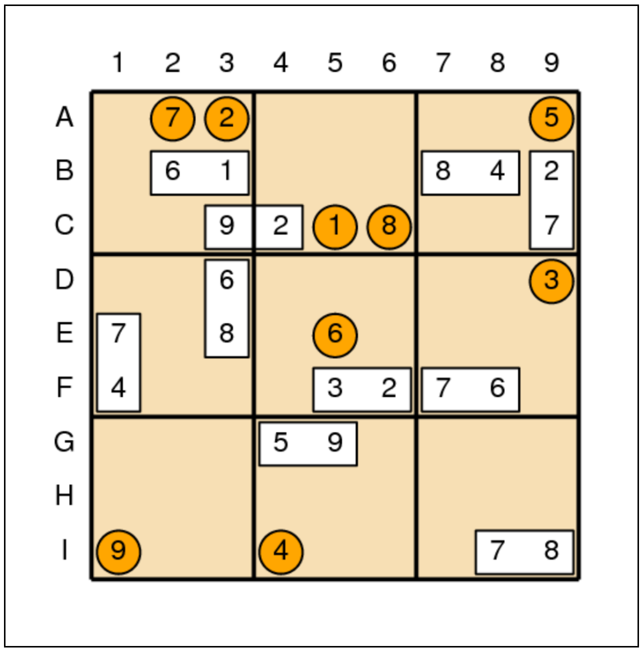

Figure 1: Sample puzzle

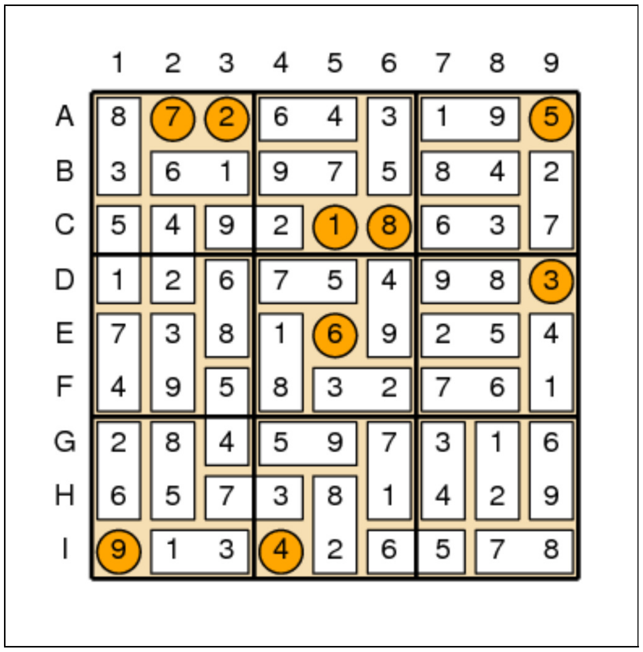

Figure 2: Solution

Input: Each puzzle description begins with a line containing an integer *N*, for $10 \leq N \leq 35$, representing the number of dominos that are initially placed in the starting configuration. Following that are *N* lines, each describing a single domino as *U LU V LV*. Value *U* is one of the numbers on the domino, and *LU* is a two-character string representing the location of value *U* on the board based on the grid system diagrammed in Figure 1. The variables *V* and *LV* representing the respective value and location of the other half of the domino. For example, our first sample input beings with a domino described as `6 B2 1 B3`. This corresponds to the domino with values 6+1 being placed on the board such that value `6` is in row `B`, column `2` and value `1` in row `B`, column `3`. The two locations for a given domino will always be neighboring.

After the specification of the `N` dominos will be a final line that describes the initial locations of the isolated numbers, ordered from 1 to 9, using the same row-column conventions for describing locations on the board. All initial numbers and dominos will be at unique locations.

The input file ends with a line containing 0.

Output: For each puzzle, output an initial line identifying the puzzle number, as shown below. Following that, output the 9x9 sudoku board that can be formed with the set of dominos. There will be a unique solution for each puzzle.

Example input:

```
10
6 B2 1 B3
2 C4 9 C3
6 D3 8 E3
7 E1 4 F1
8 B7 4 B8
3 F5 2 F6
7 F7 6 F8
5 G4 9 G5
7 I8 8 I9
7 C9 2 B9
C5 A3 D9 I4 A9 E5 A2 C6 I1
11
5 I9 2 H9
6 A5 7 A6
4 B8 6 C8
3 B5 8 B4
3 C3 2 D3
9 D2 8 E2
3 G2 5 H2
1 A2 8 A1
1 H8 3 I8
8 I3 7 I4
4 I6 9 I7
I5 E6 D1 F2 B3 G9 H7 C9 E5
0
```

Example output:

```
Puzzle 1
872643195
361975842
549218637
126754983
738169254
495832761
284597316
657381429
913426578
Puzzle 2
814267593
965831247
273945168
392176854
586492371
741358629
137529486
459683712
628714935
```

Problem C: Hex Tile Equations

Source file: `hex.{c, cpp, java}`
Input file: `hex.in`

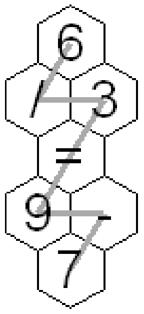

Figure 1

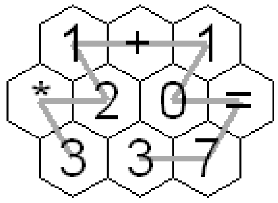

Figure 2

An amusing puzzle consists of a collection of hexagonal tiles packed together with each tile showing a digit or '=' or an arithmetic operation '+', '-', '*', or '/'. Consider continuous paths going through each tile exactly once, with each successive tile being an immediate neighbor of the previous tile. The object is to choose such a path so the sequence of characters on the tiles makes an *acceptable* equation, according to the restrictions listed below. A sequence is illustrated in each figure above. In Figure 1, if you follow the gray path from the top, the character sequence is"6/3=9-7". Similarly, in Figure 2, start from the bottom left 3 to get "3*21+10=73".

There are a lot of potential paths through a moderate sized hex tile pattern. A puzzle player may get frustrated and want to see the answer. Your task is to automate the solution.

The arrangement of hex tiles and choices of characters in each puzzle satisfy these rules:

1. The hex pattern has an odd number of rows greater than 2. The odd numbered rows will all contain the same number of tiles. Even numbered rows will have one more hex tile than the odd numbered rows and these longer even numbered rows will stick out both to the left and the right of the odd numbered rows.
2. There is exactly one '=' in the hex pattern.
3. There are no more than two '*' characters in the hex pattern.
4. There will be fewer than 14 total tiles in the hex pattern.
5. With the restrictions on allowed character sequences described below, there will be a unique acceptable solution in the hex pattern.

To have an acceptable solution from the characters in some path, the expressions on each side of the equal sign must be in acceptable form and evaluate to the same numeric value. The following rules define acceptable form of the expressions on each side of the equal sign and the method of expression evaluation:

6. The operators '+', '-', '*', and '/' are only considered as binary operators, so no character sequences where '+' or '-' would be a unary operator are acceptable. For example "-2*3=-6" and "1 =5+-4" are not acceptable.
7. The usual precedence of operations is not used. Instead all operations have equal precedence and operations are carried out from left to right. For example "44-4/2=2+3*4" is acceptable and "14=2+3*4" is not acceptable.