The puzzle *Roller-Board* consists of a *2D* rectangualr grid of cells, each of which is labelled either ‘0’ or ‘1’:

0	0	0	0	0
0	1	0	1	0
0	0	1	0	0
0	1	0	1	0
0	0	0	0	0

The challenge is to roll one row or column at a time, so that the board is returned to its ‘correct’ state:

1	1	1	1	1
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

having all 1*s* on the top row, and every other cell being a 0. Each ‘move’ can be either:

- Roll a column one place up - i.e. the cells in this column all move up one, and the cell at the top ‘rolls around’ and reappears at the bottom of this column.
- Roll a column one place down - i.e. the cells in this column all move down one, and the cell at the bottom ‘rolls around’ and reappears at the top of this column.
- Roll a row one place left - i.e. the cells in this row all move left one, and the cell on the left ‘rolls around’ and reappears on the right of this row.
- Roll a row one place right - i.e. the cells in this row all right one, and the cell on the right ‘rolls around’ and reappears on the left of this row.

To solve a 4×4 board:

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

the best solution might be:

Roll column 1 up :

1	1	0	0
0	0	0	0
0	0	1	0
0	0	0	1

Roll column 2 down :

1	1	0	0
0	0	0	0
0	0	0	0
0	0	1	1

Roll column 2 down :

1	1	1	0
0	0	0	0
0	0	0	0
0	0	0	1

Roll column 3 down :

1	1	1	1
0	0	0	0
0	0	0	0
0	0	0	0

Here’s another example of a board:

0	0	0	0
1	1	0	0
1	0	0	0
0	1	0	0
0	0	0	0

and how to solve it:

0	0	0	0
1	1	0	0
1	0	0	0
0	1	0	0
0	0	0	0

0	0	0	0
1	0	0	1
1	0	0	0
0	1	0	0
0	0	0	0

1	0	0	0
1	0	0	1
0	0	0	0
0	1	0	0
0	0	0	0

1	0	0	0
0	0	1	1
0	0	0	0
0	1	0	0
0	0	0	0

1	0	0	0
0	0	1	1
0	0	0	0
0	0	0	0
0	1	0	0

1	1	0	0
0	0	1	1
0	0	0	0
0	0	0	0
0	0	0	0

1	1	1	0
0	0	0	1
0	0	0	0
0	0	0	0
0	0	0	0

1	1	1	1
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Exercise 8.7 Write a program that reads in a roller-board file (specified on the command line), and shows the ‘moves’ to solve it. Such a file looks something like :

```
5 4
0000
1100
1000
0100
0000
```

The first line has two numbers; the height of the board (number of rows) and then the width (number of columns).
In the remainder of the file, the number of 1s must be equal to the width of the board, and only the characters ‘0’ and ‘1’ are valid.You may assume that the maximum height and width of a board is 6.
The brute-force algorithm for searching over all moves for a solution goes like this :

1. You will use an `alloc()`’d array (list) of boards.
2. Put the initial board into the front of this list, `f=0`.
3. Consider the board at the **front** of the list (index `f`).
4. For this (parent) board, find the resulting (child) boards which can be created from all the possible $(rows + columns) \times 2$ rolls. For each of these child boards:
   - If this board is unique (i.e. it has not been seen before in the list), add it to the end of the list.
   - If it has been seen before (a duplicate) ignore it.
   - If it is the ‘final’ board, stop and print the solution.
5. Add one of f. If there are more boards in the list, go to step 3.

To help with printing out the correct moves, when a solution has been found, each structure in the list will need to contain (amongst other things) a board and a record of its parent board, i.e. the board that it was created from. Since you’re using an array, this could simply be which element of the array was the parent.
The program reads the name of the board definition file from the command line. If it finds a successful solution, it prints out the number of boards that would be printed in the solution and **nothing else**, or else exactly the phrase ‘No Solution?” if none can be found :

```
$ ./rollerboard 4x4diag.rbd
4 moves
$ ./rollerboard 5x5lhs.rbd
8 moves
```

If the ‘verbose’ flag is used, your program will additionally print out the solution in the correct order :

```
$ ./rollerboard -v 4x4lr.rbd
0:
0000
1001
1001
0000

1:
1000
1001
0001
0000

2:
0100
1001
0001
0000

3:
1100
0001
0001
0000

4:
1101
0001
0000
0000

5:
1110
0001
0000
0000

6:
1111
0000
0000
0000

$ ./rollerboard -v 3x3crn.rbd
0:
000
001
011

1:
000
100
011

2:
100
000
011

3:
110
000
001

4:
111
000
000
```

Your program :

- **Must** use the algorithm detailed above (which is similar to a queue and therefore a breadth-first search). Do not use the other algorithms possible (e.g. best-first, guided, recursive etc.); the quality of your coding is being assessed against others taking the same approach.
- **Must not** use dynamic arrays or linked lists. Since boards cannot be any larger than 6×6, you can create boards of this size, and only use a sub-part of them if the board required is smaller. The list of boards can be a fixed (large) size.
- **Should** check for invalid board definition files, and report in a graceful way if there is a problem, aborting with `exit(EXIT_FAILURE)` if so.
- **Should not** print anything else out to screen after successfully completing the search, except that which is shown above. Automated checking will be used during marking, and therefore the ouput must be precise.
- **Should** call the function `test()` to perform any assertion testing etc.

Extension

Basic assignment = 90%. Extension = 10%.
If you’d like to try an extension, make sure to submit *extension.c* and a brief description in a *extension.txt* file. This could involve a faster search technique, better graphical display, user input or something else of your choosing.

■