1993/1994 Comprehensive Exam
Programming Portion

Overview

You will be writing a program to generate a particular kind of puzzle; do so will entail writing a program that can solve this kind of puzzle as well. The kind of puzzles you'll be working on are described in the attached article from *Games* magazine. You would are advised to solve a couple of these before starting on the programming, as it will speed the design and the algorithmic work. The puzzle-solving program is a little tricky, but not outrageous. A substantial part of your grade on these programs will therefore depend not on getting the code right (although that is an important part as well), but on good software engineering practice. Handing in a badly-written and undocumented program, even if it works, will not be sufficient for you to pass this exam.

General Rules

You can write this program in any language you want, and have it run on any platform you want. If, however, you choose to write it for another platform, such as a Macintosh or a PC, you must decide how to deal with adapting the X-specific portions of the problem, such as the specification of how to respond to clicks on the left, middle, and right buttons. You may make reasonable substitutes for these requirements (if you are writing a non-X program), but you should document your choices and reasons. Furthermore, in writing the program for a machine other than the one in my office, you should make certain that there is a machine nearby on which the program can be run, such as the Macintosh in the graphics lab, or the PC that's in there. If it runs on your Mac but not on mine, then it doesn't run well enough. Same goes for the PC, and for any other such special case.

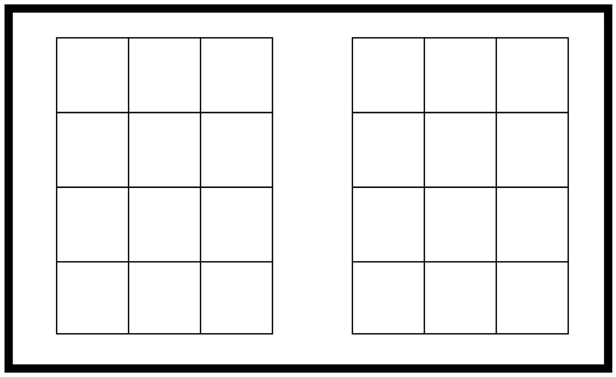
Requirements

You are to write two programs. The first should read in a puzzle description (the exact format is given below) and write out a puzzle solution (whose format is given below). If your program is given an unsolvable puzzle description, it may behave randomly without loss of credit, except that it should at least terminate without segmentation violations or floating exceptions, etc. You'll see it's worth having it do something reasonable in these cases, since it will become a part of the second program, where its behavior in these cases will actually matter.

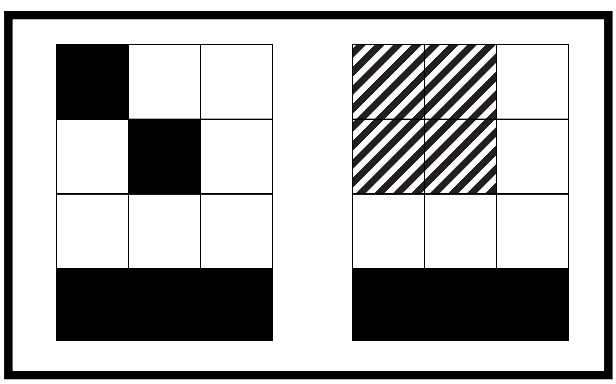
The second program takes either zero or two arguments; running it with no arguments should be equivalent to running it with the arguments "`20 20`". In general

```
program_name nr nc
```

should generate an X-window showing two "cell-grids," each having `nr` rows and `nc` columns; they should be side-by-side, but with some space in between. Thus if you invoked the program with arguments "`4 3`" you might see the following:



The left-hand grid is the "drawing area" and the right-hand grid is the "solution area." Your program should respond to a mouse-click (Left Button) within a cell in the drawing area by toggling the color of the cell from white to black, or vice versa. The program should respond to a mouse-click (Middle Button) anywhere in the X-window by attempting to solve the puzzle indicated by[1] the figure in the drawing area, and then indicating the results in the "solution area." In the solution area, those squares whose values can be determined should be drawn in black or white, according to the determined value. Those whose values cannot be determined should be drawn in red. Thus in the preceding example, if the drawing area was as shown on the left below, and the middle button had been clicked, then the solution area would be as shown (where crosshatching indicates red).



Finally, a mouse-click (Right Button) anywhere in the X-window should cause the user to be prompted (in the window from which the program was invoked) for a filename; when the user types the filename, the current puzzle description should be written into that file, in a format that matches the one that your solving-program can read, and the program should exit.

1. The puzzle "indicated by" the drawing area is one whose margin arrays contain the descriptions of the runs of black squares in each row and column of the drawing area.

How large a puzzle should you be able to handle? Up to about 40 x 40. How large should the window be, and the squares? You should make design decisions about this and defend them. For small puzzles, your solver may be so fast that you can, in the puzzle-design program, have the puzzle get re-solved (or re-attempted) every time you change the color of a square from white to black or vice-versa. In this case, the middle button will still solve the puzzle, but that's more or less irrelevant, since it will already have been solved. Doing this or not is a design choice that I leave to you to make and defend.

You are to make your program do at least what is described above. If you want to implement bells and whistles (such as letting "click-and-drag" indicate an entire area to turn black or white), you should feel free to do so, but you should maintain the minimal functionality described above. If you want to draw buttons in the window as well, that's fine too...but maintain the minimum functionality described above.

Your program should be written with both internal and external documentation. Coding conventions and standards for internal documentation can be found in a framemaker file in the directory `/u/jfh/comps`; you need not follow these standards, but your code should meet some clear and effective set of standards. If you choose your own naming convention, include a description of the convention in your internal documentation.

You should also provide external documentation that is so clear and detailed that it would let someone who only knows about Unix use your puzzle-design program. This means that the external documentation must include a description of the puzzles, how to create them, and what the rules for the puzzles are. It should also include the limits on the program (what's the largest possible puzzle, etc.).

A Bit of Help

The problems in the article have the property that they are all *linearly solvable*[1], which is a term I've just coined. It means that (1) the initial state of the puzzle has the property that for some row or column, the data in the margins (telling how long the spans of black are) can be used to determine the state (black or white) of at least one square in that row or column, and (2) at any subsequent time, for some row or column, the marginal data, together with any previously-determined state information in the row or column, can be used to determine the state of at least one *more* square in that row or column.

Such linearly solvable puzzles have the advantage that solving them requires no backtracking, and therefore (if you're doing them in pencil) no erasing. Your "solving" program is only required to solve problems that are linearly solvable, and may perform in random ways (except that it is required to terminate gracefully (i.e., no bus errors or segmentation violations)) on any other problem. Similarly, your puzzle-design program should be used to aid in the design of linearly-solvable problems; that is to say, the red squares in the solution area are those whose values cannot be determined by attempting linear solution (even though they might be found by using some more complex method).

1. This is a slight fib. I think that problem 5 may actually be not quite linearly solvable...

Data Specification

The data in a file that specifies a puzzle is in the following form. All symbols denote integers

```
nrows
size int int int ...
size int int int ...
.
.
ncols
size int int ...
size int int ...
```

The first number is the number of rows in the puzzle. The next nrows rows consist of lists: the first number tells how many items there are in the list, and the subsequent numbers are those items. Thus a row saying

```
3 1 4 5
```

indicates that there are three numbers in the list, and that they are `1, 4,` and `5`. This would indicate that this row of the puzzle had margin data that says "`1 4 5`".

Following the `nrows` of margin data for the rows, there is a similar structure indicating the margin data for the columns. To make this completely unambiguous, here is the data for the chicken shown in Figure 1 of the attached article:

```
10
1     2
2     1  1
1     4
2     2  1
2     3  1
1     8
1     8
1     7
1     5
1     3

10
1     1
1     2
2     1  6
1     9
1     6
```

```
1       5
1       5
1       4
1       3
1       4
```

Between the numbers in the input, there may be any amount of whitespace (i.e., blanks, tabs, or newline characters). It would, of course, be nice if your output were nicely formatted like the example above.

Output Specification

Your puzzle-solving program should produce output in the following form. For an $n \times k$ puzzle, you should print out an $n \times k$ array of characters. Print a period (".") where there is a blank, and an asterisk "*" where the square is filled in. Each row of the array should be terminated with a newline. The printout for the chicken puzzle should be the following

```
..**......
.*.*......
****......
..**.....*
..***....*
..********
..********
..*******.
...*****..
....***...
```

which I hope is fairly clear.

Windows and X

I realize that not everyone is familiar with X-windows, so I am providing a minimal, badly-designed, X-based library for you to use. I used it myself in writing the test version of this program, and it worked like a charm. You're welcome to hack this code in whatever way you want, by the way. You should also be warned that at least some versions of this code (and this may be one of them) have an off-by-one error in the `XGRAPHdraw_clines` code (and possibly in some of the other code).But `XGRAPHdraw_line` and `XGRAPHdraw_filled_rect_w_color` both seem to work fine. In the directory `/u/jfh/comps/src`, there is a wealth of stuff. There is the XGRAPH library (in `XGRAPH.c`), two sample programs that use XGRAPH, one in C (`txg.c`), and one in C++(`txg.C`), a Makefile for them, and documentation is in `xgraph.3`; there are also `include` files for C and C++ (but not ANSI C). All files are readable by everyone, but not writable.

Design Issues

In writing these two programs, you will have to make various design decisions (for example, how large should the window be? How large should the boxes be? What algorithm will I use to solve the puzzle? What sorts of data structures do I use? What language do I use?). You should document these design decisions carefully.

Final Remarks

Your finished code should preferably work on Solaris Machines, and the external documentation should make it clear how to invoke your programs. If the program runs on some other platform than a Solaris machine, you should carefully document how to invoke it on such a machine. A `README` file in your handin directory should give a brief description of the contents of that directory. You can choose to implement this program in any language you like, using any libraries you know of. I am most familiar with C, C++, and other procedural languages, although I am conversant with LISP, APL, and a few other miscellaneous languages as well. If you choose to use a language far from these, your documentation should be correspondingly more detailed.

What do hand in, and where

I will make a directory for each person taking the comps; it will be called

`/u/jfh/comps/xxx`,

where "`xxx`" is your login name. For Hagit Shatkay, the handin directory will therefore be `/u/jfh/comps/hs`. Since the directory is readable and writable by everyone, you should protect the individual files in the directory.

In your handin directory:
You should hand in the code for both programs (as source files and relevant Makefiles). There should be a `README` file that explains the contents of the directory. You should compile and execute your program in this directory to be certain that it will work, and will not depend on anything about your own environment (except that X is running, and the machine supports color). The directory should also contain your design document. In the event that you are handing in non-Sun source code, you should document this fact in the README file, which will presumably explain the lack of a Makefile, etc.

In my mailbox:
You should hand in a printout of the code, the external documentation, and the design document.