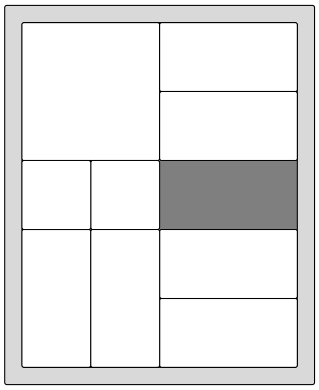
Algorithmic Programming Contest

The goal of this contest is to find interesting sliding-piece puzzles. The program which can assemble specified sets of pieces into the most difficult puzzles wins The Grand Prize: $100, plus a cool puzzle. Other contest entries that show originality, creativity, or higher-than-average performance will receive cash prizes and puzzles as well.

The contest is open to students in 6.046J/18.410J/SMA5503, but it is only for fun, not credit. Whether you participate or not will not in any way affect your final grade for the course. (Of course, outstanding entries may stick out in the professors' minds if you ever need a recommendation.) Students may enter alone or in groups. The contest begins Monday, December 3, 2001, and it ends Monday, December 10, 2001. We provide you with sample code that is complete, but inefficient. To enter, all you need to do is improve the provided code. We expect a good submission can be prepared in an evening.

1 The Problem

One interesting category of puzzle is the *sliding-piece puzzle*. Typically, one is given a rectangular box containing several rectangular pieces, which may slide about in the box without rotating. The goal is usually to move a particular piece to a particular location in the box. For example, here is a classic puzzle known as *Dad's Puzzler*:



In this puzzle, the goal is to move the square piece in the upper left to the lower left. The shortest solution requires 59 moves. (See below for the exact definition of a move.)

We are going to consider a generalization of this kind of puzzle, where neither the box nor the pieces need be rectangular. Instead, both kinds of shape will be specified as 2-dimensional bit-vectors, representing unions of 1x1 squares. (For example, the classic "pentomino" shapes may be represented this way – see `http://www.xs4all.nl/~gp/pentomino.html`.)

Your program will take as input a box shape, a set of “free” piece shapes, a (unique) goal piece shape, and a goal location. Your task is to design from these as difficult a puzzle as possible. That is, your output should specify a set of pieces drawn from the input set, their starting locations, and a minimum-length sequence of moves which results in the goal piece arriving at the goal location.

Details about the input and output formats and requirements for the implementation are given later in this handout. Sample implementations in C, C++, and Java have also been provided. You are encouraged use this code as a starting point, but you should be aware that it is not particularly efficient.

Your challenge is to apply your algorithmic and engineering sense, from this course and elsewhere, to improve the performance of the provided code or otherwise develop an efficient puzzle-designing program. As a bonus, the better your program is, the more interesting puzzles it can create! Particularly nice puzzles may be distributed outside MIT, to an international community of puzzle enthusiasts.

2 Getting Started

We provide you with sample code in C, C++, and Java. This code parses the input, searches (very naively) for interesting puzzles, and prints the output. The sample code is now available on the course website.

You may enter the contest by improving the supplied code, or by writing your own program, in any language you wish. The only submisssion requirement is that you provide us with simple Unix/Linux build instructions, so that we may run all submissions on the same machine for judging. If you choose not to use the supplied code, or have build instructions different from those provided with the sample code, we advise you to contact us early to make sure we will be able to build your program on our machine.

Your program must read its input data from standard input, and print its solution to standard output. It is likely that for many of the input specifications, your program will not be able to search all possible puzzles in any reasonable amount of time. To make it easier on you, your program may print out multiple solutions, and we will judge only the last solution completely printed before your allotted time runs out. The sample code already does this for you: it prints out each new solution it finds that is better (i.e. a more difficult puzzle) than the previous best.

Several sample inputs are available on the course website for testing purposes, but the actual inputs used for judging will be secret.

3 Detailed Specification

Much of this section can just be skimmed before getting started, because the supplied code already handles this specification. If you run into any specific questions, though, they should be answered here.

3.1 Solution Requirements

The problem is to find a difficult puzzle instance consistent with the input specifications. You are given a box shape, a list of "free" piece shapes, a unique goal piece shape, and a goal location. With these you need to build a puzzle. You may use any of the free piece shapes as many times as you wish in your puzzle, but the goal piece (which will have a unique shape) must be used exactly once. You do not need to consider rotations and reflections of the piece shapes; just use them in the orientations in which they are given. (For some inputs we may give rotated and reflected versions in the input.)

Your puzzle will be evaluated on how complex a solution it requires. This is defined to be the number of moves needed to put the goal piece into the goal location. Note that it is the *required* solution length that counts – it does no good to print out a solution using 100 moves, when the puzzle could be solved in 20.

It is important to note the definition of a move. It's customary to follow Martin Gardner, the famous recreational mathematician, and declare that any kind of motion involving just one piece counts as a single ***move***. A move can be thought of as a sequence of unit shifts of a piece, each one sliding the piece up, left, down, or right. The supplied code correctly generates all such legal moves from a given configuration. If you rewrite this code, be sure you also follow this definition of a move, because that is how the testing software will count moves.

3.2 Input Format

If you are using the supplied code, you can skim this section; we have written the input routines for you. However, you may need to modify them to take into account any data structure changes you make.

The input to your program is in four sections: a box shape, a set of "free" puzzle piece shapes, a goal piece shape (different from the free shapes), and a goal position. Each of the shapes is given as a 2-dimensional bit-vector. All positions are relative to upper left corners of bounding rectangles of shapes. The input sections are separated by lines containing a single '-' character.

Each bit-vector is preceded by a line containing its width and height. The bit-vector itself follows, with one line of text per row. Each line consists of space-separated '0' and '1' characters. '1' indicates that square is occupied; '0' indicates that square is empty. For the box shape, the '0's define the open space the pieces can occupy; a rectangular box would be all '0's. For the pieces, the '1's define the shape of the piece. Rectangular pieces would be all '1's.

The piece shape list is preceded by a count of (non-goal) piece shapes.

Here is an input specification consistent with Dad's Puzzler:

```
4 5
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
-
3
1 1
1
1 2
1
1
2 1
1 1
-
2 2
1 1
1 1
-
0 3
```

Note that the goal position "0 3" means that the upper left corner of the goal piece should be 0 units to the right and 3 units down from the upper left corner of the box. This would be true even if the upper left corner of the goal piece were not occupied (or if the upper left corner of the box were occupied).

3.3 Output Format

If you are using the supplied code, you can skim this section; we have written output routines for you. However, you may need to modify them to take into account any data structure changes you make.

You may print whatever debugging or visualization output you wish. Only output between the strings `BEGIN SOLUTION` and `END SOLUTION` will count as program output; furthermore, only the *last* such output will be counted. Within those two strings, the output must conform to the following format. DO NOT CHANGE THIS OUTPUT FORMAT, or your output may not be correctly parsed by our testing software.

A solution consists of three output sections: the starting configuration, the claimed minumum solution length, and an actual solution of that length. The output sections are separated by lines containing a single '-' character.

The starting configuration specifies for each piece in your puzzle what shape it is, and where it starts. Each line of the configuration should contain a shape index followed by a position, relative to the upper left corner of the box. Shapes are numbered starting with 0. If there are n free piece shapes in the input file, they are numbered $0..n-1$, and the goal piece is referred to as number n. This configuration defines the list of pieces in your puzzle. If there are m pieces, they are numbered $0..m-1$ for use in the following solution output.

The solution should be printed one move per line, where each move is given as a piece index to move, followed by any number of “up”, “left”, “right”, or “down” strings. These indicate the sequence of shifts that together count as a single move.

Here is a sample output for Dad’s Puzzler:

```
BEGIN SOLUTION
3 0 0
2 2 0
2 2 1
0 0 2
0 1 2
1 0 3
1 1 3
2 2 3
2 2 4
-
59
-
4 right right
3 right right
0 down
...
0 left
END SOLUTION
```

This should be read as follows: the first piece in the puzzle (piece 0) is of shape 3 (which in this case is the goal piece), and it is in the upper left corner of the box; the next piece (piece 1) is of shape 2, and its upper left corner is two units to the right of the upper left of the box, etc. The solution takes 59 moves. The first move is to move piece 4 (which is of shape 0 - a unit square) to the right two spaces. The last move moves the goal piece into the goal position (0 3).

(The sample code also prints a human-friendly puzzle diagram before the `BEGIN SOLUTION` line.)

4 Strategy

You are on your own as to how to find the best puzzles! However, a few hints to get you started are in order. The first thing that must be said is that to find the best solution, you must do an exponential search. As such, in a sense your program cannot be both optimal and efficient no matter what. But there is a lot of room for cleverness in speeding up the move generation and evaluation, and, much more importantly, in weeding out large sections of the search space with good heursitics.

The second thing that must be said is that actually, you must do two exponential searches! That is, you are trying to find difficult puzzles, so you are searching over potential puzzle layouts. But for each layout you consider, you must find the shortest solution to that layout – this requires another search.

Here is a partial list of techniques you may find useful: symmetry detection, iterative deepening, A* search, hashing, Zobrist keys, caching, randomized sampling, partial cover problems, backtrack searching methods.

Here are some potentially useful references. The URLs are also on the course webpage.

Hordern, L. E. *Sliding Piece Puzzles.* Oxford University Press, Oxford, 1986.

Winston, Patrick Henry. *Artificial Intelligence.* Addison-Wesley, third edition, 1992.

```
http://www.johnrausch.com/SlidingBlockPuzzles/default.htm
http://www1.ics.uci.edu/˜eppstein/180a/970408.html
http://cogsci.ucsd.edu/˜batali/108b/lectures/heuristic.html
http://www-cs-faculty.stanford.edu/˜knuth/papers/dancing.ps.gz
http://www.seanet.com/˜brucemo/topics/zobrist.htm
```

5 Entering

If you are interested in entering the contest, you should send mail to Bob Hearn, the TA in charge of the contest, as soon as possible. His email address is `rah@ai.mit.edu`. He will create a mailing list of all people interested in the contest. Any additional information, clarifications, bug fixes, etc. will be sent to people on this mailing list. This information will also be available on the course web page, so you may wish to check there periodically. You should send all questions, bug reports, and so forth to Bob. He can also be reached by phone at 617-253-8576.

When you are ready to submit your program, email Bob with instructions on where to get your source files (for example, put a tarball on a webpage and send the URL), and how to build them. To repeat: If you choose not to use our sample code, or have build instructions different from those provided with the sample code, we advise you to contact us early to make sure we will be able to build your program on our machine. We cannot guarantee acceptance of programs that are not straightforward to build.

All submissions must be received by **12:00 noon EST, Monday, December 10, 2001**.

We will compile and run your code on several inputs, ranging from very simple (i.e. small box and few pieces) to very complicated (i.e. large box and many pieces). We expect that no submission will be able to exhaustively search the entire puzzle space for some of the inputs.

For each output puzzle, the claimed solution length will be verified by us. Invalid solutions will be thrown out, so make sure you get the solution length right! Correctness is more important than speed, so be sure not to break the supplied code.

We will evaluate each submission based on the (correct) solution complexity for all the inputs, and on the time used for the runs that terminate before hitting the time limit. The amount of runtime granted each submission will depend on how many submissions there are, but you should assume somewhere between 5 minutes and 20 minutes per run. Even the supplied code will be able to complete the search in this time for many of the inputs.

For those who like to optimize, here are the characteristics of the machine we will be testing on (main memory is the most important number):

Pentium 4, 2GHz running Linux 2.4
256KB L2 cache
900MB main memory
2GB swap

6 Supplied Code

We supply working code in C, C++, and Java. These programs correctly read the input specifications, search for puzzles, and print out solutions. The sets of code are supplied as tarballs. They are now available on the course webpage. For the Java version, the javadoc documentation is also available on the course webpage.

Here is the strategy the supplied code uses:

1. Read puzzle specification from input.
2. Search through all possible puzzle layouts. Try to fill each vacant spot in the box in turn, either with some input piece shape, or with empty space. When the entire box is full, we have a candidate puzzle. Search the layout for solutions.
3. Solution search: Do a breadth-first search, starting from the layout configuration, searching for configurations that have the goal piece in the goal position. Stop when we find a solution.

   There are a couple of subtleties here. The vertices in our graph correspond to puzzle configurations that are reachable from the starting configuration. But we don't have these all in a nice list; they are implicitly defined. We generate new configurations on the fly, by trying all possible moves (graph edges) from each configuration we reach. In order to keep the BFS from running forever, we have to detect when we reach a configuration we've seen before. So we store all configurations we've seen during this search in an auxiliary list. Before adding a new configuration to the BFS queue, we search this list (and the queue) to make sure we haven't already seen it.

   The second subtlety lies in the definition of a move. In a single ***move***, a single piece can be slid horizontally or vertically (by one unit) one or more times. We use another BFS to find all available moves of a piece from a given configuration.
4. When we find a solution, we know the minimum number of moves necessary to reach it, by virtue of BFS. Print out the puzzle and its solution if it's longer than the previous longest solution, and terminate the solution search.
5. Keep looping through layouts, until we finish or (more likely) run out of time!

For specifics on class structures etc., you should look at the source files. They are reasonably well-commented.

Good luck, and happy hacking!