CS202 -- Lab 9

- **CS202 -- Data Structures and Algorithms I**
- **James S. Plank**
- **This file: http://web.eecs.utk.edu/~jplank/plank/classes/cs202/Labs/Lab9**
- **Lab directory: /home/jplank/cs202/Labs/Lab9/**

This is a recursion lab with two parts. The first part is worth 20 points, and the second is worth 80. Your lab submission should contain only t only two files: **src/enum.cpp** and **src/ss_solver.cpp**.

Part 1: Enumerating Strings

Your first job is write a program called **bin/enum**, which is called as follows:

```
bin/enum length ones
```

This program should enumerate all strings of length **length** that contain **ones** ones, and (**length**-**ones**) zeros, and print them on standard output in sorted order. For example:

```
UNIX> bin/enum 2 2
11
UNIX> bin/enum 2 1
01
10
UNIX> bin/enum 2 0
00
UNIX> bin/enum 3 1
001
010
100
UNIX> bin/enum 3 2
011
101
110
UNIX> bin/enum 5 3
00111
01011
01101
01110
10011
10101
10110
11001
11010
11100
UNIX> bin/enum 100 100
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
UNIX> bin/enum 100 0
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
UNIX>
```

You may puzzle about how to do this, but it is easy with recursion. I defined a recursive procedure:

```
void do_enumeration(string &s, int index, int n_ones);
```

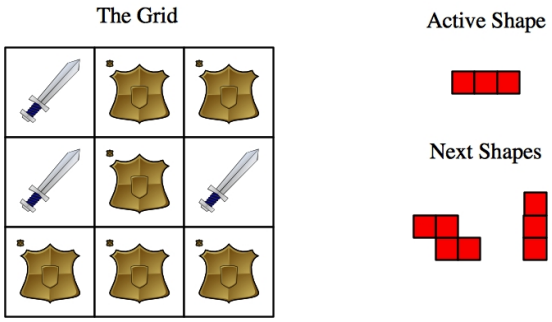
This procedure assumes that **s** has been initialized to be a string of size **length**, and that the characters in indices less than **index** have been set. In my code, I have the unset characters be dashes, because it makes it easier to see what's going on with the recursion. This procedure will enumerate all values ≥ **index** that have exactly **n_ones** ones, and print each of those strings. To help you, I have added code to print the recursive calls when you call:

- **bin/enum 3 2** in txt/enum-3-2.txt.
- **bin/enum 5 3** in txt/enum-5-3.txt.

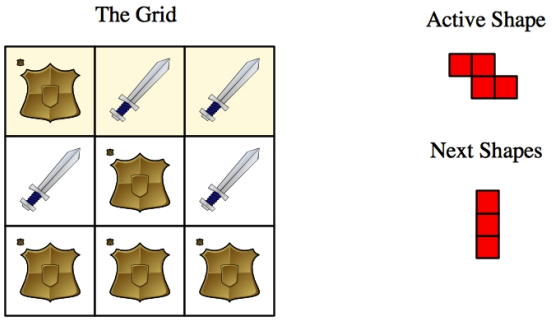
Part 2: ShapeShifter!!

The Game

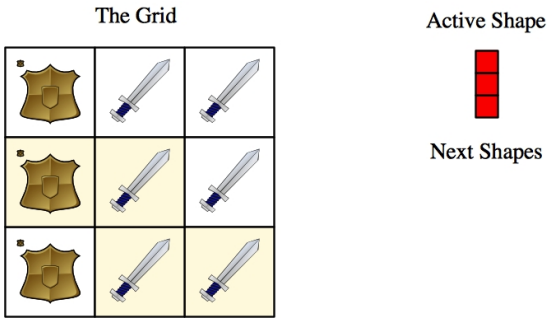
In neopets.com, there was a game called ShapeShifter. (I have no clue if it still exists -- my kids are grown). I won't screen-dump their pictures, since that probably violates a few copyrights, so I'll explain the game using my own pictures. You are given a setup like below, consisting of grid of swords and shields, an "active shape", and some "remaining shapes":



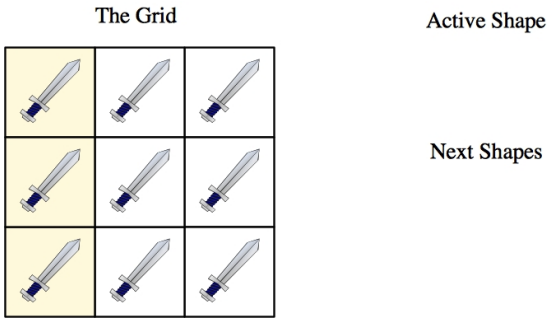
You apply the active shape to a part of the grid where it fits. For example, in the picture above, you can apply the active shape to row zero, row one or row two (life is zero-indexed, of course). When you apply the shape, all grid cells that the shape overlaps change swords to shields and vice versa. For example, suppose you apply the active shape to row zero. The state of the system then changes to:



The next shape can be applied to row 0 or row 1. Suppose we apply it to row 1. Now the state of the system is:

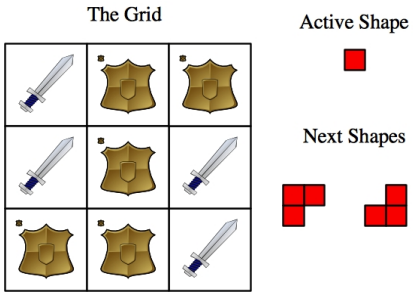


There is one more shape, which can be applied to columns 0, 1 or 2. We apply it do column zero, and the grid becomes all swords:

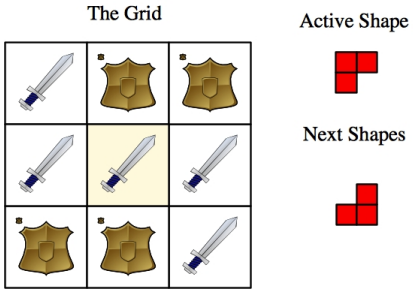


Those are the mechanics of the game. You are given a grid of swords and shields and a collection of shapes. Your goal is to apply *all* of the shapes to the grid so that you end up with a grid of all swords.

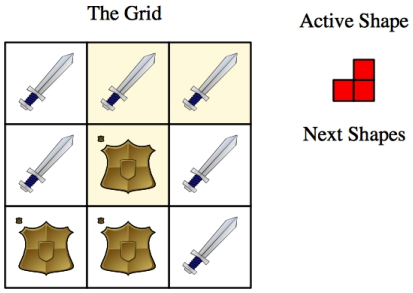
Here's a second example:



There are nine places for that first shape to go. Since I know the solution, I know it gets applied at row 1, column 1:



There are four places for the next shape -- we apply it to row 0, column 1:



I think you can figure out where that last shape goes.

The Interactive Game Player

Since we are computer scientists, we are going to represent shields with zeros, and swords with ones. We will represent any rectangular grid of zeros and ones that has *r* rows and *c* columns with a vector of *r* strings. Each string in the vector has exactly *c* characters, which are either '0' or '1'.

Thus, the grid in our first example is:

```
{ "100", "101", "000" }
```

and the grid in the second example is:

```
{ "100", "101", "001" }
```

We can also represent shapes as grids of zeros and ones. For example, the shapes in the first example are:

```
{ "111" }
{ "110", "011" }
{ "1", "1", "1" }
```

and the shapes in the second example are:

```
{ "1" }
{ "11", "10" }
{ "01", "11" }
```

I have written an interactive game player, called **bin/ss_player**, which takes a prompt and then a grid as command line arguments. After the prompt, each word on the command line specifies a different row of the grid. These words must all be the same size, and they must be composed solely of zeros and ones. Suppose I want to play using the first example above. Then I can play with:

```
UNIX> /home/jplank/cs202/Labs/Lab9/bin/ss_player ShapeShifter: 100 101 000

The Grid:
```

```
100
101
000

ShapeShifter:
```

At the prompt, I enter a shape and where to apply it. Each row of the shape is a separate word, and the last two words must specify a row and a column. Below, I show how you would specify the solution to the first example:

```
UNIX> /home/jplank/cs202/Labs/Lab9/bin/ss_player ShapeShifter: 100 101 000

The Grid:

100
101
000

ShapeShifter: 111 0 0

The Grid:

011
101
000

ShapeShifter: 110 011 1 0

The Grid:

011
011
011

ShapeShifter: 1 1 1 0 0

The Grid:

111
111
111

ShapeShifter: <CNTL-D>
UNIX>
```

Go ahead and try to do the second example on your own.

If you give a prompt of "-", there will be no prompt, which is useful for when you use the player with a file or output of a program as input.

Your job: ss_solver

Your job is to write the program **ss_solver.cpp**. It takes a grid on its command line, and then each line of standard input specifies a shape. After reading all of the shapes, your program should output how to apply each shape to solve the puzzle. If there is no solution, it should simply exit with no output.

For example, the file **txt/ex1.txt** contains the shapes for the first example:

```
111
110 011
1 1 1
```

txt/ex2.txt contains the shapes for the second example. Here's the program running the two examples:

```
UNIX> cd /home/jplank/cs202/Labs/Lab9/
UNIX> bin/ss_solver 100 101 000 < txt/ex1.txt
111 0 0
110 011 1 0
1 1 1 0 0
UNIX> bin/ss_solver 100 101 001 < txt/ex2.txt
1 1 1
11 10 0 1
01 11 1 0
UNIX>
```

Testing your program

The program **ss_random_game** generates a random game. It takes five arguments -- rows, columns, number of shapes, a seed, and whether the puzzle should be solvable. That last argument should either be "y" for "yes" or "u" for "unknown."

Here are two examples. The first creates a 3x4 grid that is solvable with the given pieces. The second creates a 4x4 grid whose solvability is unknown. The program **sed** is used to strip out the first two lines of standard input.

```
UNIX> cd /home/jplank/cs202/Labs/Lab9/
UNIX> bin/ss_random_game
usage: ss_random_game rows cols nshapes seed(0 for current time) solvable(y|u)
UNIX> bin/ss_random_game 3 4 4 1 y > $HOME/tmp.txt     # I'm using $HOME/tmp.txt, because if you run these commands from my directories,
UNIX>                                                  # you can't create files there.  You can create files in your $HOME directory.
UNIX> cat $HOME/tmp.txt
```

```
Grid: 0111 0110 0000
Shapes:
11 11
101 100 110
111 011
011 100 011
UNIX> sed 1,2d $HOME/tmp.txt     # This strips out the first two lines
11 11
101 100 110
111 011
011 100 011
UNIX> sed 1,2d $HOME/tmp.txt | bin/ss_solver 0111 0110 0000
11 11 0 1
101 100 110 0 0
111 011 0 1
011 100 011 0 1
UNIX> sed 1,2d $HOME/tmp.txt | bin/ss_solver 0111 0110 0000 | bin/ss_player - 0111 0110 0000

The Grid:

0111
0110
0000


The Grid:

0001
0000
0000


The Grid:

1011
1000
1100


The Grid:

1100
1011
1100


The Grid:

1111
1111
1111

UNIX> bin/ss_random_game 4 4 4 2 u > $HOME/tmp.txt
UNIX> cat $HOME/tmp.txt
Grid: 1010 1111 0011 1110
Shapes:
101
0110 0011 1110
11
1 1 1 1
UNIX> sed 1,2d $HOME/tmp.txt | bin/ss_solver 1010 1111 0011 1110
UNIX> rm $HOME/tmp.txt
UNIX>
```

Since there is no output from **ss_solver**, the puzzle is not solvable.

---

ss_tester.sh

In the lab directory, there's a testing shell script called **ss_tester.sh**. This is what the grading script uses. You call it with four arguments:

- Your solver
- My solver
- The player
- An input file, which is in the same format as the output to **ss_random_test**.

It then runs both your and my solver. If my solver says its not solvable, yours should too. Otherwise, it uses the player to make sure that your solution works. Your solution should not have any extraneous output.

When it is done running, there will be four new files:

- **tmp-pieces.txt** -- these are the input pieces
- **tmp-youroutput.txt** -- this is the output of your solver
- **tmp-yourgame.txt** -- this is the output of the player on your solver
- **tmp-yourpieces.txt** -- these are the pieces that you output.

Here's an example -- you should be in your directory that has **bin/ss_solver** in it. I'm using **bash** because it lets me set the environment variable **$l**, which saves us some typing

```
UNIX> bash
bash-3.2$ PS1='BASH> '
```

```
BASH> l=/home/jplank/cs202/Labs/Lab9
BASH> sh $l/scripts/ss_tester.sh
usage: sh ss_tester.sh your-solver my-solver player output-of-ss_random_game
BASH> cat tmp-test.txt
Grid: 0011 1010 0110 1101
Shapes:
010 101
1 1 1 1
101 010 001 001
1111 0100
BASH> sh $l/scripts/ss_tester.sh bin/ss_solver $l/bin/ss_solver $l/bin/ss_player tmp-test.txt
Correct
BASH> cat tmp-pieces.txt
010 101
1 1 1 1
101 010 001 001
1111 0100
BASH> cat tmp-yourpieces.txt
010 101
1 1 1 1
101 010 001 001
1111 0100
BASH> cat tmp-youroutput.txt
010 101 0 1
1 1 1 1 0 1
101 010 001 001 0 0
1111 0100 2 0
BASH> cat tmp-yourgame.txt

The Grid:

0011
1010
0110
1101


The Grid:

0001
1111
0110
1101


The Grid:

0101
1011
0010
1001


The Grid:

1111
1111
0000
1011


The Grid:

1111
1111
1111
1111

BASH> exit
UNIX>
```

Here are two more examples. In the first, I have a program called **bin/badsolver1**, which doesn't solve the problem. Don't try to duplicate this example -- just read.

```
UNIX> bash
bash-3.2$ PS1='BASH> '
BASH> l=/home/jplank/cs202/Labs/Lab9
BASH> sh $l/scripts/ss_tester.sh bin/badsolver1 $l/bin/ss_solver $l/bin/ss_player tmp-test.txt
Incorrect -- your solution does not solve the puzzle correctly.
BASH> cat tmp-youroutput.txt
11 0 0
1001 0 0
111 010 101 0 0
10 11 10 0 0
BASH> cat tmp-yourgame.txt

The Grid:

0110
1010
1110
0100


The Grid:
```

```
1010
1010
1110
0100


The Grid:

0011
1010
1110
0100


The Grid:

1101
1110
0100
0100


The Grid:

0101
0010
1100
0100

BASH> exit
UNIX>
```

The second program **badsolver2** solves the puzzle, but doesn't use the right pieces:

```
BASH> sh $l/scripts/ss_tester.sh bin/badsolver2 $l/bin/ss_solver $l/bin/ss_player tmp-test.txt
Incorrect -- your solution does not use the correct pieces in the correct order.
BASH> cat tmp-youroutput.txt
1010 0111 0000 0001 0 0
0010 0000 0010 1010 0 0
0100 0010 0010 0000 0 0
0101 0000 0001 0000 0 0
BASH>
```

If the tester fails, try to figure out why with the output files, and with my correct solver.

The grading script

The grading script runs **ss_tester.sh** on various input files. Some of them may take a few seconds. The **gradeall** script took 42 seconds for me on a hydra. Yours shouldn't take more than three minutes.

Program Structure

I'm going to let you structure this one yourself. The only requirement is that your program must use recursion. Here was my class definition -- you don't need to use this -- it's just to help you think about the problem:

```
class Shifter {
  public:

    /* Read_Grid_And_Shapes() initializes the grid from argc/argv, and the
       reads from standard input to get the shapes. */

    bool Read_Grid_And_Shapes(int argc, const char **argv);

    /* Apply_Shape() applies the shape in Shapes[index] to the grid,
       starting at row r and column c.  You'll note, if you call
       Apply_Shape twice with the same arguments, you'll end up
       with the same grid as before the two calls. */

    void Apply_Shape(int index, int r, int c);

    /* Find_Solution() is the recursive procedure.  It tries all possible
       starting positions for Shape[index], and calls Find_Solution()
       recursively to see if that's a correct way to use Shape[index].
       If a recursive call returns false, then it "undoes" the shape by
       calling Apply_Shape() again. */

    bool Find_Solution(int index);

    /* This prints the solution on standard output. */

    void Print_Solution() const;

  protected:

    /* This is the grid.  I have this be a vector of 0's and 1's, because it makes
       it easy to print the grid out. */

    vector <string> Grid

    /* These are the shapes, in the order in which they appear in standard input. */
```

```
    vector < vector <string> > Shapes;

    /* These two vectors hold the starting positions of the shapes, both as you
       are finding a solution, and when you are done finding a solution. */

    vector <int> Solution_Rows;
    vector <int> Solution_Cols;
};
```