```c
/**
 * fifteen.c
 *
 * Computer Science 50
 * Problem Set 3
 *
 * Rob Bowden (rob@cs.harvard.edu)
 *
 * Implements The Game of Fifteen (generalized to d x d).
 *
 * Usage: ./fifteen d
 *
 * whereby the board's dimensions are to be d x d,
 * where d must be in [MIN,MAX]
 *
 * Note that usleep is obsolete, but it offers more granularity than
 * sleep and is simpler to use than nanosleep; `man usleep` for more.
 */

#define _XOPEN_SOURCE 500

#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <ctype.h>

// board's minimal dimension
#define MIN 3

// board's maximal dimension
#define MAX 9

// board, whereby board[i][j] represents row i and column j
int board[MAX][MAX];

// keeps track of where the empty space is
int col_0, row_0;

// dimensions
int d;

// prototypes
void clear(void);
void greet(void);
void init(void);
```

```
49. void draw(void);
50. bool move(int tile);
51. bool won(void);
52. void save(void);
53.
54. int brute_force(int solution_path[],int steps);
55. void god_mode(void);
56.
57. int main(int argc, char *argv[])
58. {
59.     // greet user
60.     greet();
61.
62.     // ensure proper usage
63.     if (argc != 2)
64.     {
65.         printf("Usage: ./fifteen d\n");
66.         return 1;
67.     }
68.
69.     // ensure valid dimensions
70.     d = atoi(argv[1]);
71.     if (d < MIN || d > MAX)
72.     {
73.         printf("Board must be between %d x %d and %d x %d, inclusive.\n",
74.          MIN, MIN, MAX, MAX);
75.         return 2;
76.     }
77.
78.     // initialize the board
79.     init();
80.
81.     // accept moves until game is won
82.     while (true)
83.     {
84.         // clear the screen
85.         clear();
86.
87.         // draw the current state of the board
88.         draw();
89.
90.         // saves the current state of the board (for testing)
91.         save();
92.
93.         // check for win
94.         if (won())
95.         {
96.             printf("ftw!\n");
```

```
 97.             break;
 98.         }
 99.
100.         // prompt for move
101.         printf("Tile to move: ");
102.         string input = GetString();
103.
104.         // if string not entered (user just hit CTRL-D), start loop over
105.         if (input==NULL)
106.         {
107.             continue;
108.         }
109.
110.         // if the user requests GOD mode, call god_mode and end program after
111.         else if (!strcmp(input,"GOD"))
112.         {
113.             printf("You called? Here, let me help you.\n");
114.             god_mode();
115.             printf("Thank you for using GOD mode. Please come again!\n");
116.             return 0;
117.         }
118.
119.         // move the entered tile number if possible, else report illegality
120.         if (!move(atoi(input)))
121.         {
122.             printf("\nIllegal move.\n");
123.             usleep(250000);
124.         }
125.
126.         // sleep thread for animation's sake
127.         usleep(250000);
128.     }
129.
130.     // that's all folks
131.     return 0;
132. }
133.
134. /**
135.  * Clears screen using ANSI escape sequences.
136.  */
137. void clear(void)
138. {
139.     printf("\033[2J");
140.     printf("\033[%d;%dH", 0, 0);
141. }
142.
143. /**
144.  * Greets player.
```

```
 */
void greet(void)
{
    // prints an ascii-art "Welcome to the game of fifteen"
    clear();
    printf(" _    _  ___  __    __  __  __  __  ___    ____  __     ____  _  _"
           "  ___ \n( \\/\\/ )(  _)(  )  / _)/  \\(  \\/  )(  _)  (_  _)/  \\ "
           "  (_  _)( )( )(  _)\n \\    /  ) _) )(__( (_( () ))    (  ) _)    "
           ")( ( () )    )(   )__(  ) _)\n  \\/\\/  (___)(____)\\__)\\__/(_/\\"
           "/\\_)(___)   (__) \\__/    (__) (_)(_)(___)\n  __   __   __  __  _"
           "__      __  ___    ___  __  ___  ____  ___  ___  _  _ \n / _) (  ) "
           "(  \\/  )(  _)   /  \\(  _)  (  _)(  )(  _)(_  _)(  _)(  _)( \\( )"
           "\n( (/\\ /__\\  )    (  ) _)  ( () )) _)   ) _) )(  ) _)  )(   ) _"
           ") ) _) )  ( \n \\__/(_)(_)(_/\\/\\_)(___)   \\__/(_)    (_)  (__)("
           "_)   (__) (___)(___)(_)\\_)\n\n");
    usleep(1000000);
}

/**
 * Checks to see if the pseudorandom board that init() creates is actually
 * winnable by summing how many times a smaller number come AFTER larger
 * numbers on the board. If the sum is even, return winnable, else not.
 */
bool winnable_config(void)
{
    int count = 0;
    for (int i = 0; i < d * d - 1; i++)
    {
        for (int j = i + 1; j < d * d - 1; j++)
        {
            if (board[i / d][i % d] > board[j / d][j % d])
            {
                count++;
            }
        }
    }
    return !(count % 2);
}

/**
 * Initializes the game's board with tiles numbered 1 through d * d - 1,
 * (i.e., fills board with values but does not actually print them),
 * whereby board[i][j] represents row i and column j.  The tiles are
 * placed in a pseudorandom, yet solvable, configuration.
 */
void init(void)
{
    int tiles = d * d;
```

```
193.
194.      // fill a template array with the numbers that will go on the board
195.      int numbers[tiles - 1];
196.      for (int i = 1; i < tiles; i++)
197.      {
198.          numbers[i - 1] = i;
199.      }
200.
201.      // take a pseudrandom number out of the template array and put it on board
202.      srand(time(NULL));
203.      for (int i = 0, randnum; i < tiles - 1; i++)
204.      {
205.          randnum = rand() % (tiles - i - 1);
206.          board[i % d][i / d] = numbers[randnum];
207.
208.          // take the number out of array so it can't be reused
209.          for (int j = randnum; j < tiles - 1 - i; j++)
210.          {
211.              numbers[j]=numbers[j + 1];
212.          }
213.      }
214.
215.      // if the pseudorandom configuration is not solvable, swap the top-left two values
216.      if (!winnable_config())
217.      {
218.          int temp = board[0][0];
219.          board[0][0] = board[0][1];
220.          board[0][1] = temp;
221.      }
222.
223.      // initialize the position of the blank space to the bottom right corner
224.      col_0 = d - 1;
225.      row_0 = d - 1;
226.      board[row_0][col_0] = 0;
227.  }
228.
229.  /**
230.   * Prints the board in its current state.
231.   */
232.  void draw(void)
233.  {
234.      // useful string formatting patterns, including ANSI escape sequences
235.      string centering = "                      ";
236.      string normal = "\033[0m";
237.      string blue = "\033[34;40m";
238.      string green = "\033[32m";
239.      string white_on_red = "\033[41;37m";
240.
```

```
    // print the "Game of fifteen!" title bar
    printf("%s", centering);
    for(int i = 0; i < d / 2 - 1; i++)
    {
        printf("     ");
    }
    printf("%sGame of Fifteen!\n%s%s", white_on_red, blue, centering);

    // print the board itself
    for (int i = 0; i < d; i++)
    {
        // print the top border of a row
        for (int j = 0; j < d; j++)
        {
            printf("=====");
        }
        printf("=\n%s", centering);

        // print the left border of a box and the number it contains,
        // with a space for 0. Numbers in their correct location will be green.
        for (int j = 0; j < d; j++)
        {
            printf("%s| %s%2d%s ", blue,
                board[i][j]==i*d+j+1?green:normal,
                board[i][j],board[i][j]==0?"\b\b  ":"");
        }

        // print the right side of last tile and center next line
        printf("%s|\n%s", blue, centering);
    }

    // print the bottom of the board and return escape sequences back to normal
    for (int i = 0; i < d; i++)
    {
        printf("=====");
    }
    printf("=%s\n", normal);
}


/**
 * If tile borders empty space, moves tile and returns true, else
 * returns false.
 */
bool move(int tile)
{
    // if we can move tile, the new position of tile will be the blank space
    int col = col_0, row = row_0;
```

```
289.
290.      // look to the right of the blank space, if possible. If the tile is there,
291.      // the new position of the blank will be to the right.
292.      if (col_0 < d - 1 && board[row_0][col_0 + 1] == tile)
293.      {
294.          col_0++;
295.      }
296.
297.      // look to the left of the blank space
298.      else if (col_0 > 0 && board[row_0][col_0 - 1] == tile)
299.      {
300.          col_0--;
301.      }
302.
303.      // look below the blank space
304.      else if (row_0 < d - 1 && board[row_0 + 1][col_0] == tile)
305.      {
306.          row_0++;
307.      }
308.
309.      // look above the blank space
310.      else if (row_0 > 0 && board[row_0 - 1][col_0] == tile)
311.      {
312.          row_0--;
313.      }
314.
315.      // if the col and row #'s are still equal, the tile isn't next to the blank
316.      if (col == col_0 && row == row_0)
317.      {
318.          return false;
319.      }
320.
321.      // swap the blank space and the desired tile
322.      board[row][col] = tile;
323.      board[row_0][col_0] = 0;
324.      return true;
325.  }
326.
327.  /**
328.   * Returns true if game is won (i.e., board is in winning configuration),
329.   * else false.
330.   */
331.  bool won(void)
332.  {
333.      // checks to make sure each tile is in the right position
334.      for (int i = 0; i < d; i++)
335.      {
336.          for (int j = 0; j < d; j++)
```

```
        {
            if (board[i][j] != d * i + j + 1 && !(i == d - 1 && j == d - 1))
            {
                return false;
            }
        }
    }
    return true;
}

/************************************
 ************************************
 *                                  *
 *  GOD MODE functions start here!!  *
 *                                  *
 ************************************
 ************************************/

/**
 * A helper function to factor out code that recurs in the other slide
 * functions.
 */
void slide(int delta_col_0, int delta_row_0)
{
    // swap the values of the blank space and a neighboring tile
    board[row_0][col_0] = board[row_0 + delta_row_0][col_0 + delta_col_0];
    board[row_0 + delta_row_0][col_0 + delta_col_0]=0;

    // update the location of the blank space
    col_0 += delta_col_0;
    row_0 += delta_row_0;

    // god mode needs its own clear() and draw() since it does not get
    // cleared and drawn by the infinite loop in main.
    usleep(100000);
    clear();
    draw();
}

/**
 * Slides a tile DOWN into the blank space (the blank space moves up).
 */
void slide_down(void)
{
    slide(0, -1);
}

/**
```

```
385.  * Slides a tile UP into the blank space (the blank space moves down).
386.  */
387. void slide_up(void)
388. {
389.     slide(0, 1);
390. }
391.
392. /**
393.  * Slides a tile LEFT into the blank space (the blank space moves right).
394.  */
395. void slide_left(void)
396. {
397.     slide(1, 0);
398. }
399.
400. /**
401.  * Slides a tile RIGHT into the blank space (the blank space moves left).
402.  */
403. void slide_right(void)
404. {
405.     slide(-1, 0);
406. }
407.
408. /**
409.  * Moves tile UP, with the blank beginning and ending to the right of tile
410.  */
411. void move_up(int* row)
412. {
413.     slide_down();
414.     slide_right();
415.     slide_up();
416.     slide_left();
417.     slide_down();
418.     (*row)--;
419. }
420.
421. /**
422.  * Moves tile LEFT, with the blank beginning and ending to the right of tile
423.  */
424. void move_left(int* col)
425. {
426.     // if the tile is in the bottom row, we have to go ABOVE the tile in order
427.     // to move it left, else we can just go under it
428.     bool inverse = (row_0 == d - 1) ? true : false;
429.     (inverse) ? slide_down() : slide_up();
430.     slide_right();
431.     slide_right();
432.     (inverse) ? slide_up() : slide_down();
```

```
433.     slide_left();
434.     (*col)--;
435. }
436.
437. /**
438.  * Moves tile RIGHT, with the blank beginning and ending to the right of tile
439.  */
440. void move_right(int* col)
441. {
442.     // if the tile is in the bottom row, we have to go ABOVE the tile in order
443.     // to move it right, else we can just go under it
444.     bool inverse = (row_0 == d - 1) ? true : false;
445.     slide_right();
446.     (inverse) ? slide_down() : slide_up();
447.     slide_left();
448.     slide_left();
449.     (inverse) ? slide_up() : slide_down();
450.     (*col)++;
451. }
452.
453. /**
454.  * Moves tile DOWN, with the blank beginning and ending to the right of tile
455.  */
456. void move_down(int* row)
457. {
458.     slide_up();
459.     slide_right();
460.     slide_down();
461.     slide_left();
462.     slide_up();
463.     (*row)++;
464. }
465.
466. /**
467.  * Moves tile diagonally up and to the left, with the blank beginning and
468.  * ending to the right of tile.
469.  */
470. void move_up_left(int* row, int* col)
471. {
472.     slide_down();
473.     slide_right();
474.     slide_up();
475.     slide_right();
476.     slide_down();
477.     slide_left();
478.     (*row)--;
479.     (*col)--;
480. }
```

```
481.
482. /**
483.  * Puts the tile to the left of the blank into correct row and col.
484.  */
485. void move_toward(int correct_row, int correct_col)
486. {
487.     // movement will be slightly different if we are filling in a row vs col
488.     bool inverse = (correct_row > correct_col) ? true : false;
489.
490.     // when this function is called, we know that the blank tile is to
491.     // the right of our target tile
492.     int row = row_0;
493.     int col = col_0-1;
494.
495.     // move tile as far up-left as possible
496.     while (row > correct_row && col > correct_col)
497.     {
498.         move_up_left(&row, &col);
499.     }
500.
501.     // if filling in a row, then move the tile into the correct horizontal
502.     // position, and then up into the correct slot
503.     if (!inverse)
504.     {
505.         while (col > correct_col)
506.         {
507.             move_left(&col);
508.         }
509.         while (col < correct_col)
510.         {
511.             move_right(&col);
512.         }
513.         while (row > correct_row)
514.         {
515.             move_up(&row);
516.         }
517.     }
518.
519.     // if filling in a col, then move the tile into the correct vertical
520.     // position, and then left into the correct slot
521.     else
522.     {
523.         while (row > correct_row)
524.         {
525.             move_up(&row);
526.         }
527.         while (row < correct_row)
528.         {
```

```
            move_down(&row);
        }
        while (col > correct_col)
        {
            move_left(&col);
        }
    }
}

/**
 * Searches for the tile in the board and stores its location in row and col
 */
void locate(int tile, int* row, int* col)
{
    for(int i = 0; i < d; i++)
    {
        for(int j = 0; j < d; j++)
        {
            if (board[i][j] == tile)
            {
                *row = i;
                *col = j;
                return;
            }
        }
    }
}

/**
 * Moves the blank so that it is to the right of the tile
 * that we are working on.
 */
void arrange_blank(int tile, int row, int col)
{
    // move the blank down to the correct row
    for(int i = row_0; i < row; i++)
    {
        // if the tile is just beneath the blank, then move around the tile
        // and the blank will be in the correct position
        if (board[row_0 + 1][col_0] == tile)
        {
            // if we are in the right-most column, then we have to slide
            // the tile to the left so the blank can be right of it
            if (col_0 == d - 1)
            {
                slide_right();
                slide_up();
                slide_left();
```

```
                return;
            }
            else
            {
                slide_left();
                slide_up();
                return;
            }
        }
        else
        {
            slide_up();
        }
    }
    for (int i = row_0; i > row; i--)
    {
        // if the tile is just above the blank, then move around the tile
        // and the blank will be in the correct position
        if (board[row_0 - 1][col_0] == tile)
        {
            // if we are in the right-most column, then we have to slide
            // the tile to the left so the blank can be right of it
            if (col_0 == d - 1)
            {
                slide_right();
                slide_down();
                slide_left();
                return;
            }
            else
            {
                slide_left();
                slide_down();
                return;
            }
        }
        else
        {
            slide_down();
        }
    }

    // move the blank left to the correct row
    for (int i = col_0; i > col; i--)
    {
        if (board[row_0][col_0 - 1] == tile)
        {
            return;
```

```
625.         }
626.         else
627.         {
628.             slide_right();
629.         }
630.     }
631.
632.     // move the blank right to the correct row
633.     for (int i = col_0; i < col; i++)
634.     {
635.         if (board[row_0][col_0 - 1] == tile)
636.         {
637.             return;
638.         }
639.         else
640.         {
641.             slide_left();
642.         }
643.     }
644. }
645.
646. /**
647.  * A helper function to factor out common code in brute_force.
648.  */
649. int brute_helper(int tile, int solution_path[], int steps)
650. {
651.     // try moving the current tile and store it in solution_path
652.     solution_path[steps] = tile;
653.     move(tile);
654.
655.     // if that move put board in a winning config, return #  steps to that move
656.     if (won())
657.     {
658.         return steps + 1;
659.     }
660.
661.     // if we haven't one yet, we see if moves AFTER this move put the board in
662.     // a winning config, and if so return the total steps to the final move
663.     int total_steps;
664.     if ((total_steps = brute_force(solution_path, steps + 1)))
665.     {
666.         return total_steps;
667.     }
668.
669.     // if we get to this point, then this move did not lead us in the direction
670.     // of winning, so we move the tile back to where it was and return that
671.     // the move was unsuccesful
672.     move(tile);
```

```
    return 0;
}

/**
 * Applies bruce force.
 */
int brute_force(int solution_path[32], int steps)
{
    int total_steps;

    // if we've reached 32 steps, NO 3x3 board should take that long to solve,
    // so return that we have to backtrack
    if (steps == 32)
    {
        return 0;
    }

    // tries to see if moving a tile up, if possible, will lead
    // to a winning configuration
    if (row_0 > d - 3)
    {
        // checks to make sure we aren't backtracking by moving
        // a piece back and forth
        if (!steps || board[row_0 - 1][col_0] != solution_path[steps - 1])
        {
            // if brute_helper returns nonzero, then winning path found
            if ((total_steps = brute_helper(board[row_0 - 1][col_0], solution_path, steps)))
            {
                return total_steps;
            }
        }
    }

    // tries to see if moving a tile down, if possible, will lead
    // to a winning configuration
    if (row_0 < d - 1)
    {
        if (!steps || board[row_0 + 1][col_0] != solution_path[steps - 1])
        {
            if ((total_steps = brute_helper(board[row_0 + 1][col_0], solution_path, steps)))
            {
                return total_steps;
            }
        }
    }

    // tries to see if moving a tile right, if possible, will lead
    // to a winning configuration
```

```
721.     if (col_0 < d - 1)
722.     {
723.         if (!steps || board[row_0][col_0 + 1] != solution_path[steps - 1])
724.         {
725.             if ((total_steps = brute_helper(board[row_0][col_0 + 1], solution_path, steps)))
726.             {
727.                 return total_steps;
728.             }
729.         }
730.     }
731.
732.     // tries to see if moving a tile left, if possible, will lead
733.     // to a winning configuration
734.     if (col_0 > d - 3)
735.     {
736.         if (!steps || board[row_0][col_0 - 1] != solution_path[steps - 1])
737.         {
738.             if ((total_steps = brute_helper(board[row_0][col_0 - 1], solution_path, steps)))
739.             {
740.                 return total_steps;
741.             }
742.         }
743.     }
744.
745.     // if none of the above have lead to a winning configuration, then return
746.     // 0 since the move that led us here was futile so we have to backtrack
747.     return 0;
748. }
749.
750. /**
751.  * Function that will ultimately solve the board. It works by
752.  * filling in the top most unsorted row and left most unsorted column
753.  * until we are left with a 3x3 box in the bottom right corner, at which point
754.  * it tries to find a path to the winning puzzle by brute force.
755.  */
756. void god_mode(void)
757. {
758.     // This outermost for loop keeps track of the corner around which we are
759.     // filling in first the row, and then the column. This continues until
760.     // we are left with a 3x3 unsolved box in the bottom-right corner.
761.     for (int corner = 0, tile, row, col, correct_row, correct_col; corner < d - 3; corner++)
762.     {
763.         // fill in the top-most remaining row from left to right
764.         for (int j = corner; j < d; j++)
765.         {
766.             // calculate the tile that we want to move to this position
767.             tile = corner * d + j + 1;
768.
```

```
            // store the correct coordinates of the tile in new variables,
            // since we may have to change where we want the tile to go
            correct_row = corner;
            correct_col = j;
            locate(tile, &row, &col);

            // If a tile is already in its appropriate position, then we can
            // continue to the next tile.
            if (row == correct_row && col == correct_col)
            {
                // Make sure we don't continue unless both 2nd-to-last and
                // last tiles in row are correct since those are special cases
                if (correct_col != d - 2 || board[row][col + 1] == tile + 1)
                {
                    continue;
                }
            }

            // If we are focused on the last tile in a row, then we want to
            // shift where it should go down and to the left
            if (correct_col == d - 1)
            {
                correct_row++;
                correct_col--;
            }

            // arrange the blank space so that it is to the right of the tile
            arrange_blank(tile,row,col);

            // move the tile to its correct position
            move_toward(correct_row, correct_col);

            // If the tile is the 2nd-to-last # in the row, then it must
            // be slided right so we can put the last number into place
            if (correct_col == d - 2)
            {
                // If we encounter the last # in the row while trying to
                // move the 2nd-to-last #, we can't put the last tile to the
                // left of the 2nd-to-last, or we'll be stuck
                if (board[row_0 + 1][col_0 - 1] == tile + 1)
                {
                    slide_up();
                    slide_right();
                    slide_up();
                    slide_left();
                    slide_down();
                    slide_down();
                }
```

```
                slide_right();
            }
            // If we are on the last # in row, we need to do these slides
            // to place both the 2nd-to-last and last tiles correctly
            if (!(tile % d))
            {
                slide_down();
                slide_left();
                slide_up();
            }
        }

        // move the blank space down so the next arrange_blank does not
        // screw up a correct number in the next column to be arranged
        while (row_0 < d - 1)
        {
            slide_up();
        }


        // fill in the left-most remaining column from top to bottom. steps are
        // similar to the above, except changed for filling in columns
        for (int j = corner + 1; j < d; j++)
        {
            tile = j * d + corner + 1;
            correct_col = corner;
            correct_row = j;
            locate(tile, &row, &col);

            // if the number is already in the correct place, skip to next tile
            if (row == correct_row && col == correct_col)
            {
                if (j != d - 2 || board[row + 1][col] == tile + d)
                {
                    continue;
                }
            }

            // if last # in row, change to 1 col over from correct destination
            if (j == d - 1)
            {
                correct_col++;
            }

            arrange_blank(tile,row,col);
            move_toward(correct_row, correct_col);

            // if 2nd-to-last number, arrange it so we can fit in last number
```

```
            if (j == d - 2)
            {
                if (board[row_0 + 1][col_0] == tile + d)
                {
                    slide_left();
                    slide_up();
                    slide_right();
                    slide_right();
                    slide_down();
                    slide_left();
                }
                else
                {
                    slide_up();
                    slide_right();
                    slide_down();
                    slide_left();
                }
            }

            // if last number, do slides to put 2nd-to-last and last in place
            if (j == d - 1)
            {
                slide_down();
                slide_right();
                slide_right();
                slide_up();
                slide_left();
            }
        }
        // The column is complete!

        // Now we want to move the blank all the way to the right so that
        // arrange_blank doesn't mess up any already-correct tiles
        // in the next row
        while (col_0 < d - 1)
        {
            slide_left();
        }
    }
    // At this point, the whole board has been solved except for the final
    // 3x3 bottom-right corner box

    // Save the current state of 3x3 box since brute force moves tiles around
    int temp_board[3][3];
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
```

```
        {
            temp_board[i][j] = board[i + d - 3][j + d - 3];
        }
    }
    int old_row_0 = row_0;
    int old_col_0 = col_0;

    // The highest number of moves necessary to complete a 3x3 board from
    // any configuration is 31 steps, so we will limit the size of the solution
    // path to 32. brute_force stores the moves in solution_path and
    // returns the number of steps through solution_path we have to make
    int solution_path[32];
    int steps = brute_force(solution_path, 0);

    // Restore the state of the board from before brute_force was called
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            board[i + d - 3][j + d - 3] = temp_board[i][j];
        }
    }
    row_0 = old_row_0;
    col_0 = old_col_0;

    // make the moves that are stored in solution_path and redraw board
    for (int i = 0; i < steps; i++)
    {
        move(solution_path[i]);
        usleep(100000);
        clear();
        draw();
    }
    // The puzzle is solved!
}

/**
 * Saves the current state of the board to disk (for testing).
 */
void save(void)
{
    // log
    const string log = "log.txt";

    // delete existing log, if any, before first save
    static bool saved = false;
    if (!saved)
    {
```

```
        unlink(log);
        saved = true;
    }

    // open log
    FILE* p = fopen(log, "a");
    if (p == NULL)
    {
        return;
    }

    // log board
    fprintf(p, "{");
    for (int i = 0; i < d; i++)
    {
        fprintf(p, "{");
        for (int j = 0; j < d; j++)
        {
            fprintf(p, "%i", board[i][j]);
            if (j < d - 1)
            {
                fprintf(p, ",");
            }
        }
        fprintf(p, "}");
        if (i < d - 1)
        {
            fprintf(p, ",");
        }
    }
    fprintf(p, "}\n");

    // close log
    fclose(p);
}
```

```
#
# Makefile
#
# Computer Science 50
# Problem Set 3
#

fifteen: fifteen.c
    clang -std=c99 -O0 -Wall -Werror -o fifteen fifteen.c -lcs50 -lm

clean:
    rm -f *.o core fifteen
```

```c
/**
 * find.c
 *
 * Computer Science 50
 * Problem Set 3
 *
 * Prompts user for as many as MAX values until EOF is reached,
 * then proceeds to search that "haystack" of values for given needle.
 *
 * Usage: ./find needle
 *
 * where needle is the value to find in a haystack of values
 */

#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>

#include "helpers.h"

// maximum amount of hay
const int MAX = 65536;

int main(int argc, string argv[])
{
    // ensure proper usage
    if (argc != 2)
    {
        printf("Usage: ./find needle\n");
        return -1;
    }

    // remember needle
    int needle = atoi(argv[1]);

    // fill haystack
    int size;
    int haystack[MAX];
    for (size = 0; size < MAX; size++)
    {
        // wait for hay until EOF
        printf("\nhaystack[%d] = ", size);
        int straw = GetInt();
        if (straw == INT_MAX)
        {
            break;
        }

```

```
        // add hay to stack
        haystack[size] = straw;
    }
    printf("\n");

    // sort the haystack
    sort(haystack, size);

    // try to find needle in haystack
    if (search(needle, haystack, size))
    {
        printf("\nFound needle in haystack!\n\n");
        return 0;
    }
    else
    {
        printf("\nDidn't find needle in haystack.\n\n");
        return 1;
    }
}
```

```c
/**************************************************************************
 * generate.c
 *
 * Computer Science 50
 * Problem Set 3
 *
 * Generates pseudorandom numbers in [0,LIMIT), one per line.
 *
 * Usage: generate n [s]
 *
 * where n is number of pseudorandom numbers to print
 * and s is an optional seed
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "helpers.h"

int main(int argc, string argv[])
{
    // quits program and displays usage if the number of random numbers to
    // generate and an optional seed were not entered at the command line
    if (argc != 2 && argc != 3)
    {
        printf("Usage: generate n [s]\n");
        return 1;
    }

    // Converts desired number of random numbers from a string to an int
    int n = atoi(argv[1]);

    // if a seed was entered at the command line, use that for srand, else
    // just time(NULL)
    if (argc == 3)
    {
        srand((unsigned int) atoi(argv[2]));
    }
    else
    {
        srand((unsigned int) time(NULL));
    }

    // prints the desired number of random numbers, all 0 <= and < LIMIT
    for (int i = 0; i < n; i++)
    {
        printf("%d\n", rand() % LIMIT);
```

```
49.     }
50.
51.     // that's all folks
52.     return 0;
53. }
```

```c
/**
 * helpers.c
 *
 * Computer Science 50
 * Problem Set 3
 *
 * Rob Bowden (rob@cs.harvard.edu)
 *
 * Helper functions for Problem Set 3.
 */

#include <cs50.h>
#include "helpers.h"
#include <stdlib.h>

/**
 * Returns true if value is in array of n values, else false.
 * Uses a recursive binary search algorithm.
 */
bool search(int value, int array[], int n)
{
    // if array has size 0, can't search
    if (n == 0)
    {
        return false;
    }

    // if the "middle" element in array equals value, return true
    int middle = n / 2;
    if (array[middle] == value)
    {
        return true;
    }

    // if there are still more values in the array, then if the middle element
    // was lower/higher than what we are looking for, we recursively call
    // search on the upper/lower half (excluding the middle element) of the
    // array with the size of array cut from middle over in the first case
    // and set equal to middle in the latter
    else if (array[middle] < value)
    {
        return search(value, array + middle + 1, n - middle - 1);
    }
    else
    {
        return search(value, array, middle);
    }
}
```

```c

/**
 * Sorts array of n values.  Returns true if successful, else false.
 * Implemented using counting sort.
 */
void sort(int values[], int n)
{
    // counting_array is of size LIMIT, equal to the highest random number
    // generate can output
    int counting_array[LIMIT] = { 0 };

    // iterate through values, incrementing counting array at the index that
    // is equal to values[i]
    for (int i = 0; i < n; i++)
    {
        counting_array[values[i]]++;
    }

    // the int stored in counting_array[i] is the number of i's in values,
    // so we iterate over values, setting each index equal to the next number
    // encountered in counting_array; if counting_array[0] == 6, then the first
    // 6 elements of values are set to 0, and so on
    for (int i = 0, z = 0; i < n; i += counting_array[z], z++)
    {
        for (int j = 0; j < counting_array[z]; j++)
        {
            values[i + j] = z;
        }
    }
}
```

```c
/**
 * helpers.h
 *
 * Computer Science 50
 * Problem Set 3
 *
 * Helper functions for Problem Set 3.
 */

#include <cs50.h>

#define LIMIT 65536

/**
 * Returns true if value is in array of n values, else false.
 */
bool search(int value, int values[], int n);

/**
 * Sorts array of n values.
 */
void sort(int values[], int n);
```

```
#
# Makefile
#
# Computer Science 50
# Problem Set 3
#

all: find generate

find: find.c helpers.c helpers.h
    clang -ggdb3 -O0 -std=c99 -Wall -Werror -o find find.c helpers.c -lcs50 -lm

generate: generate.c
    clang -ggdb3 -O0 -std=c99 -Wall -Werror -o generate generate.c

clean:
    rm -f *.o a.out core find generate
```