C track: assignment 4

Goals

In this assignment you will write a program to play a solitaire game, as well as learn about recursion.

Language concepts covered this week

- arrays
- recursion

Reading

Read chapter 7 of Darnell and Margolis, and/or chapter 5 of K&R (especially section 5.3).

On recursion

In most programming languages, including C, it is legal for a function to call itself. This is known as *recursion*. Those who haven't seen recursively-defined functions before often find these functions confusing, as if some rule is being broken. In fact, each invocation of the function is a separate entity and contains its own arguments, local variables, and return value. Here is a simple program including a recursive function to calculate factorials:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

/* prototype */
int factorial(int n);


int
factorial(int n)
{
    assert(n >= 0);
```

```
    if (n == 0)
    {
        return 1;
    }
    else
    {
        /* Recursive call to the factorial function: */
        return (n * factorial(n - 1));
    }
}


int
main(int argc, char *argv[])
{
    int n, f;

    if (argc != 2)
    {
        fprintf(stderr, "usage: %s n\n", argv[0]);
        exit(1);
    }

    n = atoi(argv[1]);
    f = factorial(n);
    printf("factorial of %d = %d\n", n, f);

    return 0;
}
```

Copy this to a file, name it "factorial.c", and compile it with:

```
% gcc -Wall -Wstrict-prototypes -ansi -pedantic factorial.c -o factorial
```

where "%" is the unix prompt as usual. Run the program with various values. What happens when the input is greater than 12? What happens when the input is less than zero? Why do think that might be the case? **NOTE**: You don't need to hand this in.

A very important feature of a recursively-defined function is that there must be a *base case* to which all invocations of the function eventually reduce. In this example, the base case occurs when the argument to the **`factorial`** function is zero (giving the answer 1). The most common mistake in recursive functions is to omit the base case (or one of the base cases), leading to an infinite loop. It is also important to check that each recursive call of the function has an argument which is closer to the base case than the original argument (here, **`n-1`** is less than **`n`**).

This example is somewhat contrived, because it is easy to calculate factorials without using recursion. However, for many problems, the recursive solution is

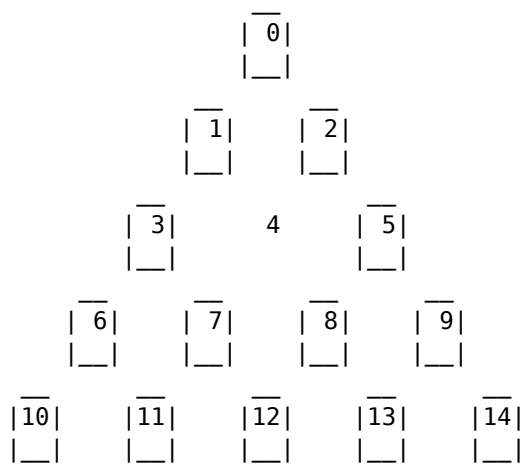
much easier to come up with than any other solution. This lab is an example of this.

Program to write

You will write a program to solve the "triangle game" which is described below.

Description of the game

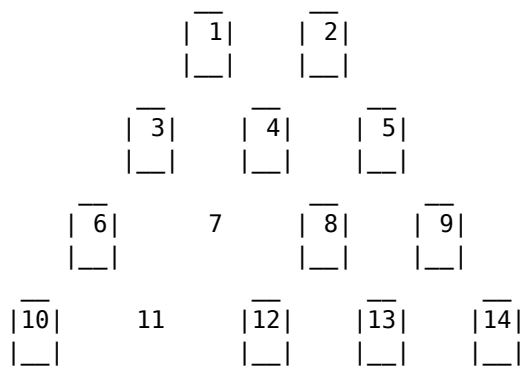
The Triangle Game is played on a triangular board with fifteen equally-spaced holes in it. The diagram below shows how the holes are arranged and numbered.



Initially, fourteen of the holes have pegs in them, while hole 4 starts out empty. We denote holes with pegs in them by drawing a box around the number.

A peg can move by jumping over an adjacent peg which is then removed, just like in checkers. However, unlike in checkers, it is okay to jump horizontally as well as diagonally (vertical jumps are not allowed). Also unlike checkers, you can't just move pegs around; the only way to move a peg is to jump over an adjacent peg into an empty space.

For instance, starting from the initial board configuration, we could take peg 11, jump it over peg 7, and land in space 4. Peg 7 is removed, leaving us with the board configuration shown here.



We could continue by taking peg 9, jumping over peg 8, and landing in space 7. Notice that there is one fewer peg after every move, which means that the longest possible game has thirteen moves, removing thirteen pegs, which leaves only one peg remaining.

The object of the game is to make the game last as long as possible (thirteen moves). It might also be nice if you could get the final peg to end up in position 4, thereby preserving a sense of harmony, but as it turns out, that isn't possible. In fact, the final peg will always be in position 12.

It's convenient to use the term "board" to refer to a particular arrangement of pegs -- for instance, we would say that the two diagrams above show two different boards. A move is said to be legal for a particular board if it complies with the rules described above. For instance, the move "jump 9 over 8 landing on 7" is a legal move for the second board shown above, but it is not a legal move for the initial board. The set of possible moves is just the set of all moves which are legal in at least one conceivable arrangement of pegs. So "jump 9 over 8 landing on 7" is a possible move. However, "jump 9 over 8 landing on 14" is not a possible move, because it is never legal.

Description of the program

You must write a program which does the following:

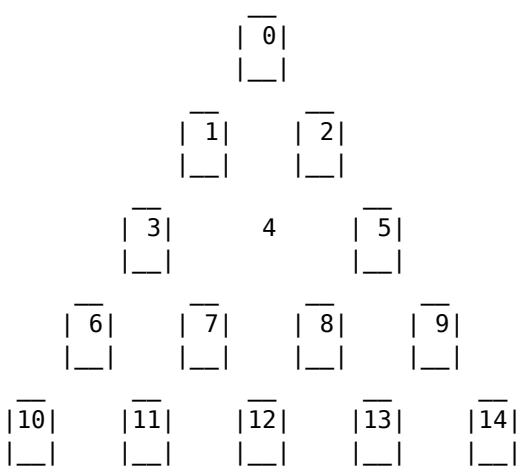
1. Get an initial board from the user. The board will be represented as a one-dimensional array of integers. At any location in the array, a value of 0 means that there is no peg at that location and 1 means that there is a peg at that location.

2. Find out if there are a set of moves which lead to only having a single peg left on the board.

    - If no such set of moves exist, report this fact and exit.

- If at least one set of moves exist, print out the board positions that the moves go through, starting with a single peg, and ending with the user-supplied board.

In order to make this easier, and to allow you to concentrate on the problem at hand, we are supplying you with the input/output routines for the program. These functions are:

Function	What it does
`void triangle_print(int board[])`	Takes **`board`**, an array of 15 integers, and prints it out in a visually-appealing way. For example, the starting board for this lab gets printed as: 
`void triangle_input(int board[])`	Asks the user to create a Triangle Game starting position, which is put into **`board`**.

To use these handy routines (and you really should use them), save the following files to the same place as your program will go:

- triangle_routines.h
- triangle_routines.c

You're not required to understand how they work, but by all means look through them if you're curious.

An Example

triangle_example.c is a short example of the use of these functions. Save it into the same directory as triangle_routines.c, and compile and run it using:

```
% gcc -Wall -o triangle_example triangle_example.c triangle_routines.c
```

```
% triangle_example
```

When you run it will simply ask you for a board and print out the board you entered.

A couple of things to notice in the example:

- You need to have the following line with your other **`#include`**s. Note the quotation marks --- that means that the preprocessor won't look in the standard location for the header file (which is `/usr/include` on Linux) but instead will look in the current directory.

  ```
  #include "triangle_routines.h"
  ```

- Compile your program like this:

  ```
  gcc -Wall -o triangle_game triangle_game.c triangle_routines.c
  ```

  Alternatively, you can use the Makefile we supply. Type "`make triangle_example`" to make the example program, and "`make triangle_game`" for the game itself. If you have any trouble with this, ask your instructor or TA.

We strongly encourage you to use this example as a starting point for your program.

Supporting files

All of these should be downloaded to your `lab4` directory, except for the style checker, which you should already have in your `~/bin` directory if you've followed the instructions in the style guide.

- The Makefile
- triangle_routines.h
- triangle_routines.c
- triangle_example.c
- The style checker
- A test input file

Testing your program

You can test your program by running it directly from the command line, or by typing `make test`. This will use the `test_input` file as the input to your program. This file loads all the pegs except the central one, and the resulting board has a

solution, so it should work.

Hints

This is a more difficult program than the previous ones in this track, so here are some hints.

Solving the problem

Your board-solving function should take one argument: an array representing a board. It should return 1 (true) if the board can be solved, and 0 (false) if the board cannot be solved. Other than that, it should not change the board. It's OK if it changes the board temporarily as long as it changes it back to its original state before returning.

To check if the board is solvable it's easiest if you use a recursive algorithm. One way to use recursion to solve a board would be to proceed as follows:

- First, identify the base case. What kind of board can you tell immediately is solvable? (*Hint*: what kind of board is *already* solved?)
- Second, let's say that the base case doesn't apply. Then you can make each legal move, recursively check if the resulting board (board #2) is solvable, and then unmake the move you just made. If the recursive call told you that board #2 was solvable, then you know that your original board is also solvable because you can make a single move, convert it into board #2, and solve that. If this is the case, you're done and you can return. If not, keep trying more moves.
- If you never find a successful first move, then the board is unsolvable.

In order to know what moves are allowed, you need to come up with a representation of the move. One way is to represent a move as an ordered triple of numbers. For instance, in the initial position we can take a peg at position 11, jump over position 7 and land in position 4. This can be represented as the triple: 11, 7, 4. Similarly, we can enumerate all possible moves (there are 36 of them), and stuff them into a global array:

```
#define NMOVES 36

int moves[NMOVES][3] =
{
  {0, 1, 3},
  {3, 1, 0},
```

```
  {1, 3, 6},
  {6, 3, 1},
  /* ...  (lots more, total of 36) */
  {12, 13, 14},
  {14, 13, 12}
};
```

This uses C's array initialization syntax (see lecture 3, or Darnell and Margolis sections 7.3 and 7.11, or K&R sections 4.9 and 5.7). Then you can iterate through the array, looking for moves which are legal given the current board position. When you find one, make the move and recursively call the function on the resulting board. If the move does not lead to a solution, un-make it and continue. Once there is only a single peg on the board, you have found a solution.

If there's a solution, you're required to also print out the solution in reverse order (*i.e.* starting with a board with only one peg and growing by one peg each time until the starting board is printed). This seems really hard but in fact is really easy. All you need to do is (in the board-solving function) to print out the board before any place where you return 1 (a true value *i.e.* success). Because of the way the recursive function works, this will end up printing out the successful solution in reverse order. Normally, it's bad design to print out something in a function whose main job is to do something else (like solve a board in this case); however, here we're mainly using the board printing as a debugging aid, so it's OK. Don't print the board in any function except for the `solve()` function.

Various other hints

Here are the function prototypes we used for our solution, along with comments stating what they do:

```
/* Return the number of pegs on the board. */
int npegs(int board[]);

/* Return 1 if the move is valid on this board, otherwise return 0. */
int valid_move(int board[], int move[]);

/* Make this move on this board. */
void make_move(int board[], int move[]);

/* Unmake this move on this board. */
void unmake_move(int board[], int move[]);

/*
 * Solve the game starting from this board.  Return 1 if the game can
 * be solved; otherwise return 0.  Do not permanently alter the board passed
```

```
 * in. Once a solution is found, print the boards making up the solution in
 * reverse order.
 */
int solve(int board[]);
```

Note that the `move[]` arguments in the above function prototypes don't refer to the entire 2-dimensional moves array but to a single move *i.e.* a one-dimensional array of 3 integers.

The `solve()` function is a bit tricky (it's the recursive one); the rest are quite straightforward. When we say "do not permanently alter the board passed in" what we mean is that the function can change the board, but it should undo the change before it exits. When you think about it, this makes sense; all the function really has to do is to figure out whether the board is solvable or not (which doesn't require the board to be altered when it exits) and to print out the solution if it finds one (ditto). The `unmake_move()` function will be useful to ensure that the board doesn't get permanently altered before the `solve()` function exits.

As we mentioned above, the fact that the `solve()` function prints the solution after it finds it is unusual and would normally be considered bad design; normally we would store the solution somewhere and print it in another function. That can be done, but it's quite a bit harder, so to make this easier you just have to print the solution immediately, and you don't have to store any old boards anywhere. That's also the reason for the reverse order; it makes it possible to print the boards without having to store them. To be absolutely clear, by reverse order we mean that the solved board (with one peg) is the first thing you print, followed by the previous board (with only two pegs) and so on up to the full board with *e.g.* 14 filled pegs and one empty peg. Of course, use the `triangle_print()` function to print the board.

We recommend that you concentrate on solving the board first, and only then worry about printing the boards in the solution. It won't require much extra code.

You should also realize that it's OK to use one-dimensional array components of a two-dimensional array. So the first move in the array is moves[0], which in the above example stands for the array `{0, 1, 3}`. A lot of students typically copy the move to a new array, but that isn't necessary.

Coding style hints

Don't use any global variables for this program except for the two-dimensional array of moves! It's OK for the moves array to be a global variable because it's really a global constant; you never change it. Global variables that get changed are occasionally useful too, but more often they just

make your program much harder to debug, so normal practice is to avoid using them if possible. None of the labs in the C track require global variables.

Don't use magic numbers! Don't put numbers like 36 or 15 in your code; use `#define` to create symbolic names for them. That's good programming style.

To hand in

Your completed program "triangle_game.c".

References

- Darnell and Margolis, chapter 7.
- K&R, chapters 4 and 5.
