Exercise 2 — Sorting and Gaming

Andreas Nüchter, Fabian Arzberger, Jasper Zevering

Winter 2025/2026

Goals: In this assignment you will learn about C arrays, C strings and how to make your program interact with the unix (i.e. Linux) command line. You will then write a simple sorting program and a recursive program. Finally, you will learn some useful strategies for testing code.

Prerequisites:

- arrays
- strings
- command-line argument processing
- recursion
- Makefiles again
- Using `assert` for debugging
- Test scripts again.

Programs to write:

1. Write a program called `sorter` which will
   (a) Accept up to 32 numbers (integers) on the command line.
   (b) If there are no arguments on the command line, the program should print out instructions on its use. There should only be one usage message, and it must follow the standard conventions.
   (c) If there are more than 32 numbers on the command line, or no numbers at all, the program should print out the usage message and exit.
   (d) If the special command-line options `-b` or `-q` are found anywhere in the command line, change the behavior of the program as described below. Note that these command-line options must be included in the usage message.
   (e) If any of the arguments to the program are not integers or one of the two command-line options, your program's response is undefined – it can do anything (i.e. you shouldn't worry about having to handle anything but integer arguments or the two command-line options). The way to deal with this is as follows: for each argument, first check to see if it's one of the command-line options. If so, proceed accordingly. If not, assume it represents an integer and convert it using the `atoi()` function (see below).
   (f) Sort the numbers using either the minimum element sort or the bubble sort algorithm (see below). *Do not use a global array to hold the integers*; use a locally-defined array in main and pass the array to the sorting function. Define separate functions for both sort algorithms. Use assert (see below) to check your sorting function for correctness.

(g) Print out the numbers from smallest to largest, one per line.

The Sample Output

If the above doesn't make sense, this is what your program should look like (bold is what you would type, and % is the Unix prompt):

```
% sorter 5 9 -2 150 -95 23 2 5 80
-95
-2
2
5
5
9
23
80
150
% sorter
usage: sorter [-b] [-q] number1 [number2 ... ] (maximum 32 numbers)
%
```

Command-line arguments

Your program begins in the `main` function, which up until now has looked like this:

```
int main(void)
{
   /* your code here */
   return 0;
}
```

but will now look like this:

```
int main(int argc, char *argv[])
{
   /* your code here */
   return 0;
}
```

So let's take the first line apart. There are a few parts to this:

int Declares that this function returns an integer.

main Declares this function's name, `main`. Recall that C programs always begin executing in the `main` function.

int argc `argc` is equal to the number of elements of `argv`.

char *argv[ ] Okay, this one's a bit trickier. The second argument, argv, is an array of `char *`s. `char *` is C's way of handling character strings, which are represented as arrays of characters where the last character is ASCII character 0 (often written as `\0`). This will make more sense when we have discussed pointers in a couple of weeks. argv contains one string for each of the command line arguments that your program is run with.

Converting strings to integers

Okay, so how do we turn these argv strings into integer values? Well, there's this handy function called atoi ("ascii to integer"). For example, atoi("5") equals 5. In order to use atoi, you need to put the following line at the top of your program:

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

`atoi` is a pretty dumb function; if you pass it a bogus value it'll just return 0 instead of signalling any kind of error. Watch out for this.

The minimum element sort algorithm

Alright, so what is this "minimum element sort" algorithm? The basic idea is that the smallest element in the array will be the zeroth element in the sorted array, the second-smallest will be the first element, etc. Here's how it works:

Let `array` be the array of integers, and `num_elements` be the total number of elements in array.

(a) Start with `start = 0` (for the zeroth element).

(b) Set `smallest = start` (smallest stores the smallest element encountered so far).

(c) Run through a loop with the variable `index` going from start to `num_elements`
If `array[index] < array[smallest]`, set `smallest = index`.

(d) Once the loop ends, swap `array[start]` and `array[smallest]` (moving the smallest element found to the beginning of the array you searched).

(e) Increment `start`, and if `start < num_elements`, go back to step **??**. Do not use a goto statement, however; use another loop.

(f) If `start >= num_elements` then you're done and the array is sorted.

Adding command-line options

You will add the ability to process two optional command-line arguments (usually called "command-line options") to your program. These options will be `-b` and `-q`. To test for these, you'll need to be able to compare strings. The function `strcmp(str1, str2)` returns `0` if `str1` and `str2` are the same and nonzero otherwise. So to check if the first argument is `-b`, you would have to do something like:

```
if (strcmp(argv[1],"-b") == 0)
{
     /* put stuff here */
}
```

To use the `strcmp` function, you need to put the following line with the other `#includes` at the top of your file:

```
#include <string.h>
```

To keep things simple, you are only required to handle integers or the two specific command-line options `-b` and `-q`, and any command-line argument (not counting the program name) that isn't `-b` or `-q` can be assumed to be an integer.

Here are the command-line options and what they mean:

- The `-b` option means that you should sort using a Bubble Sort instead of a Minimal Element Sort algorithm.
  Bubble Sort, like Minimal Element Sort, is not a very efficient algorithm. Much more efficient algorithms (such as Quicksort) exist, but they are best left until after you've had some experience with recursion, which is coming up next lesson :-) Also, the C standard library has a qsort (Quicksort) library function...
- The `-q` option suppresses the output (i.e. nothing gets printed). Why would we want to do this? See the next section.

Information on different sorting algorithms, including bubble sort, can be found all over the web, in algorithms textbooks, ...

Testing your program

Many programmers can write working code, but few programmers test their code systematically to see if it actually does work. It is important to learn different strategies for doing this, because it can not only improve the quality of your code, but also the quality of your life.

Using `assert` is probably the single easiest way to improve the quality of your code, and one of the least used. The idea behind `assert` is to make your program self-checking. Let's say you've just coded up an incredibly hairy algorithm which will balance the budget, cure cancer, send mankind to the stars and do other wonderful things. How do you know that your code doesn't have a bug? And if it does have a bug, how do you find it?

To use assert, make sure you add this to the top of your file:

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

In your sort function(s), write this right at the end of the function:

```
/* Check that the array is sorted correctly. */

for (i = 1; i < num_elements; i++)
{
    assert(array[i] >= array[i-1]);
}
```

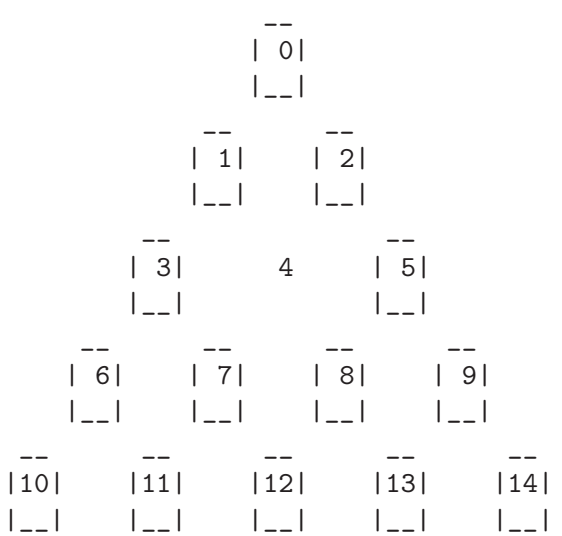
where `num_elements` is the number of elements in the array and array is the name of the `array`. Notice that here we're putting the `assert` statement in a loop. This bit of code is technically called a postcondition; a postcondition is what is required to be true after a function has completed. Some computer languages, such as Eiffel, have very sophisticated systems for checking postconditions (as well as preconditions and various kinds of invariants); these languages make it easier to write correct code than C does. Here, the postcondition will cause an assertion failure if the array isn't sorted at the end of the sorting function, which is what we want.

Notice how the postcondition is just three lines of trivial code, as compared to the sort routines themselves, which will probably be considerably longer. This illustrates the general principle that it's much easier to check if something is right than to make it right in the first place. Assertions are easy to write; use them!

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

Description of the game

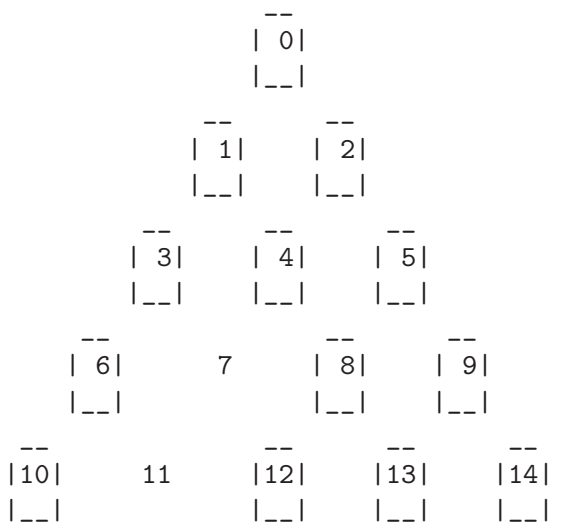
The Triangle Game is played on a triangular board with fifteen equally-spaced holes in it. The diagram below shows how the holes are arranged and numbered.



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

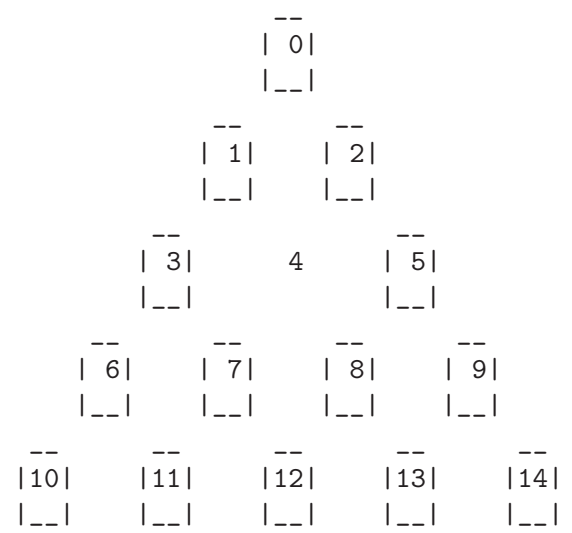
You must write a program which does the following:

(a) Get an initial board from the user.

(b) 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. To use them please include the "triangle_routines.h" header in your program. The suppplied functions are:

`void triangle_print(int board[])` Takes `board`, an array of 15 integers, and prints it out. For example, the starting board shown in Homework 2 gets printed as:



`void triangle_input(int board[])` Asks the user to create a Triangle Game starting position, which is put into `board`.

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.

By far the easiest way to solve this is to use recursion. One way to use recursion to solve a board would be to proceed as follows: given a board for which there exists at least one legal move, make each legal move and (recursively) solve the resulting board before undoing the move and going on to the next legal move. If, on the other hand, there is no legal move, then solving the board is easy. (This is the base case; at this point it should be obvious whether the game has been solved or not). It's better to check for the base case first, before trying to do the recursion.

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. 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. At this point, you have to print the board and back up so you can reconstruct the solution at each step by printing out each position in the solution.

Even more 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[]);
```

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.

Also, 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.

Also, *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.

Submission: The `sorter.c` file. We will check to see that it compiles, passes the style checker and passes the test script. Also, make sure that you describe the algorithms you use in the comments. You don't have to be exhaustively detailed, but give a brief description.

Furthermore, please submit the completed program `triangle_game.c`.

Please send the programs to `fabian.arzberger@uni-wuerzburg.de`
and `jasper.zevering@uni-wuerzburg.de` until October 29, 2025 23.59. Include `[CS4SE]` in your subject line and state the names of all team members in the message body. Please work in groups of two students. One submission per team is sufficient. Late homework won't be accepted!