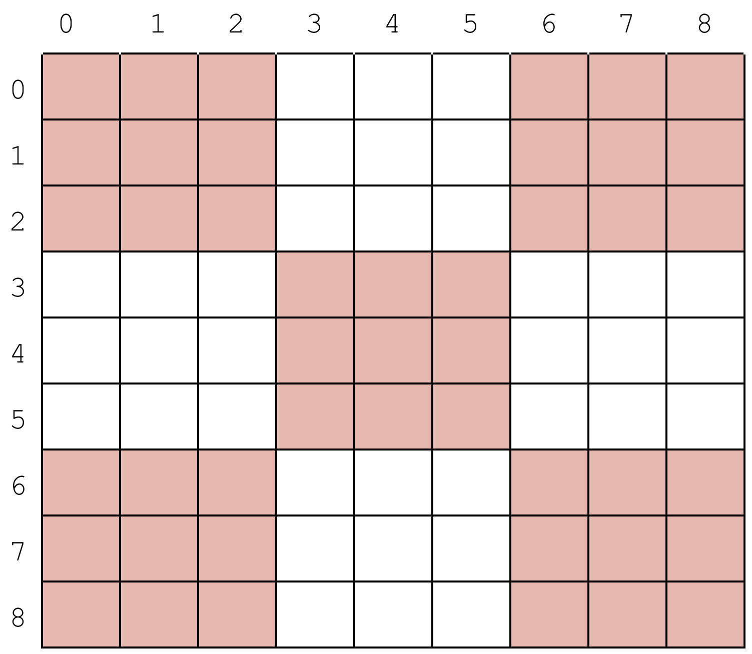
HW 4: A Sudoku Checker and two Recursive Programs
Due Mon Oct 20 by 2:59PM

Part I: Checking a Sudoku Board

You will write a `C` function for determining the correctness of a (filled-out) Sudoku board. In the unlikely event that you don't know the rules of Sudoku, here is a summary:

1) The board is a `9x9` grid.
2) A correct/legal completed board must obey the following:
    a) Each cell contains an integer in `{1, 2, 3, 4, 5, 6, 7, 8, 9}`
    b) Every row must be a permutation of `{1..9}`: each value must occur exactly once.
    c) Every column must be a permutation of `{1..9}`: each value must occur exactly once.
    d) Each of `9 3x3 boxes` must be a permutation of `{1..9}`: each value must occur exactly once. The `3x3` boxes are illustrated by the alternating red and white regions in the figure below.
3) If any of these are violated, the board is illegal/invalid.

The empty board below shows the 9 rows, 9 columns and 9 boxes.



From item 2 above, we can see that there are 27 9-cell “units”:

- 9 rows
- 9 columns
- 9 boxes

Each such unit is either “good” or “bad”. **The C function you write will return the number of “bad” units.**

Below is an example of a *legal board.* Since, as you may have guessed, it is natural to represent the board as a 2D array in C, we’ve included row and column indices.

	0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8	9
1	7	8	9	1	2	3	4	5	6
2	4	5	6	7	8	9	1	2	3
3	9	1	2	3	4	5	6	7	8
4	6	7	8	9	1	2	3	4	5
5	3	4	5	6	7	8	9	1	2
6	8	9	1	2	3	4	5	6	7
7	5	6	7	8	9	1	2	3	4
8	2	3	4	5	6	7	8	9	1

Let’s swap two values and see what happens. We swap entries `board[2][6]` with `board[3][6]` below (otherwise, the board is the same as above).

	0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8	9
1	7	8	9	1	2	3	4	5	6
2	4	5	6	7	8	9	6	2	3
3	9	1	2	3	4	5	1	7	8
4	6	7	8	9	1	2	3	4	5
5	3	4	5	6	7	8	9	1	2
6	8	9	1	2	3	4	5	6	7
7	5	6	7	8	9	1	2	3	4
8	2	3	4	5	6	7	8	9	1

Now the board is illegal. What are the violations?

- Row 2: two `6`s and missing a `1`.
- Row 3: two `1`s, missing a `6`.
- NorthEast box: two `6`s, missing a `1`
  Observation: we can refer to boxes by a convention like the coordinates of its upper-left cell. In this case `[0][6]`. We could say that the square is “rooted” at `[0][6]`.
- Box rooted at `[4][6]`: two 1s, missing a 6.

Note that column 6 is ok -- no violation. **So, for this board there are a total of 4 violating units.** Below is the C stub for the function you are to write.

```
/**
* Function:  scheck
* Purpose:  checking the correctness of a Sudoku board
* Parameter:  a 9x9 2D array of integers representing the board.
* Returns the number of 9-number “units” in which errors are found
*   A completely correct board will result in a return of 0
*       There are 27 “units”:
```

```
*          9 rows
*          9 columns
*          9 3x3 squares
*          9 3x3 squares
*  Note that an "out of range" value is also an error.
*/
int scheck(int board[9][9]) {






}
```

```
// file:  scheck tester
// Description:
//     primitive program exercising/testing the scheck
//     sudoku checker function
//
// Disclaimer:  only for illustration purposes.  Does not do
//     robust testing by any means.
//     Merely offered as a starting point.

#include "scheck.h"
#include <stdio.h>

int main() {
int nviolations;

int ok[][] =  {
                       {1, 2, 3, 4, 5, 6, 7, 8, 9},
                       {7, 8, 9, 1, 2, 3, 4, 5, 6},
                       {4, 5, 6, 7, 8, 9, 1, 2, 3},
                       {9, 1, 2, 3, 4, 5, 6, 7, 8},
                       {6, 7, 8, 9, 1, 2, 3, 4, 5},
                       {3, 4, 5, 6, 7, 8, 9, 1, 2},
                       {8, 9, 1, 2, 3, 4, 5, 6, 7},
                       {5, 6, 7, 8, 9, 1, 2, 3, 4},
                       {2, 3, 4, 5, 6, 7, 8, 9, 1}
                 };

int bad[][] =  {
                       {1, 2, 3, 4, 5, 6, 7, 8, 9},
                       {7, 8, 9, 1, 2, 3, 4, 5, 6},
                       {4, 5, 6, 7, 8, 9, 1, 2, 3},
                       {9, 1, 2, 3, 4, 5, 6, 7, 8},
                       {6, 7, 8, 9, 1, 2, 3, 4, 5},
                       {3, 4, 5, 6, 7, 8, 9, 1, 2},
                       {8, 9, 1, 2, 3, 4, 5, 6, 7},
                     {5, 6, 7, 8, 9, 1, 2, 3, 4},
                     {2, 3, 4, 5, 6, 7, 8, 9, 1}
               };

  nviolations = scheck(ok);
  if(nviolations != 0)
       printf("FAILED:  expected 0 violations; %i reported\n",
                nviolations);
  else
       printf("PASSED: %i violations reported\n", nviolations);

  nviolations = scheck(bad);
  if(nviolations != 4)
       printf("FAILED:  expected 4 violations; %i reported\n",
                nviolations);
  else
       printf("PASSED: %i violations reported\n", nviolations);
}
```

To Do

You have been given a trivial header file `scheck.h` which just contains the declaration of the `scheck` function. **You may not modify `scheck.h`.** You've also been giving the tester file above as `scheck_tst.c`

You need to complete the following:

- `scheck.c`: This contains your implementation of the scheck function. You may, of course, add helper functions as you see fit (you are encouraged to do so).
- `scheck_tst.c`: Your test program which may build on the example above, or you may start from scratch. This is primarily for our reference so we can see how you went about testing your implementation.
- `makefile`: you will complete a single makefile for this entire assignment. With respect to the `scheck` portion, we should be able to do the following to compile `scheck.o` and `scheck_tst` respectively:

```
$ make scheck.o
$ make scheck_tst
```

Items to Submit

```
scheck.c
makefile
scheck.h (unmodified)
scheck_tst.c
```

Part II: A program to tell a story

Write a program that generates stories of the following form.

A **child** couldn't sleep, so her mother told a story about a little **frog**,
    who couldn't sleep, so the **frog's** mother told a story about a little **bear**,
        who couldn't sleep, so the **bear's** mother told a story about a little **weasel**
            ...who fell asleep.
        ...and the little **bear** fell asleep;
    ...and the little **frog** fell asleep;
...and the **child** fell asleep.

Your collection of animals is:

```
frog, cat, dog, horse, snake, eagle, bear, weasel (and child).
```

The story always starts with the child, but the animals *must be randomized for full credit (i.e., the sequence of animals should be different from run-to-run of the program). No animal can appear twice and all animals are used in the story.*

Your output should be indented as in the example above (exact amount of indentation is up to you).

Items to Submit

Name your source file `story.c`

Your `makefile` (the same makefile as used for parts I and III) must include a target `story` -- i.e., we should be able to compile and run your program like this:

```
$ make story
$ ./story
```

Part III: Printing Section and Subsection Headings of an Outline

You will write a program which prints Section headings for an outline/document. Let's start with an example:

```
Section 1
    Section 1.A
        Section 1.A.1
        Section 1.A.2
    Section 1.B
        Section 1.B.1
        Section 1.B.2
Section 2
    Section 2.A
        Section 2.A.1
        Section 2.A.2
    Section 2.B
        Section 2.B.1
        Section 2.B.2
```

Some observations:

- There is a hierarchy; for example:
    - `Section 1` has subsections `1.A` and `1.B`
    - `Section 2.A` has subsections `2.A.1` and `2.A.2`
- Some sections have no subsections
    - these are “leaves” or “base-cases”
    - notice that all leaf sections are at the same “**depth**” or “level”
- The number of subsections is always the same. We will call this the “**width**” of the outline.

The above example has

- `depth = 3`
- `width = 2`: There are two top-level sections and each (sub)section contains two subsections.

The example below has `depth=2` and `width=3`.

```
Section 1
    Section 1.A
    Section 1.B
    Section 1.C
Section 2
    Section 2.A
    Section 2.B
    Section 2.C
Section 3
    Section 3.A
    Section 3.B
    Section 3.C
```

Program Usage

Your will submit a program file called `outline.c`. The user runs the program by specifying the desired depth and width as *command-line arguments* (in that order). Assuming the executable is named `outline`, the user would specify, for example depth=4, width=2 by running the program as:

```
$ ./outline 4 2
```

Details, Assumptions and Tips

Indent the headings as in the example. Each sub-heading should be indented some number of spaces more than its "parent" section's heading. The exact number of spaces is up to you.

Note that subsection specifiers alternate between numbers and letters. For full credit, your program should behave this way. Partial credit will be granted for solutions that just use numbers. You may assume that *width* will never be more than 26. This also puts an upper bound on the maximum length of a section heading (useful for declaring arrays of `char`).

String buffers. The library function `sprintf` might be useful. This function uses the same kind of formatting as the other `printf`-like functions members, but it's "target" is an array (instead of a file pointer or stdout). Try the code below and you'll see how it works!

```
char buf[100];
char *word1 = "Holy";
char *word2 = "Moly";

sprintf(buf, "%s %s", word1, word2);

printf("%s\n", buf);
```

Restrictions: As stated earlier, your program does not need to support `width` greater than 26. We will also place a similar simplifying constraint on `depth`: an upper-bound on depth for our purposes will be `10.` As a result, we can specify an upper-bound on the number of characters on any line of the output -- 200 should be safe. This means that you don't how to worry at runtime if you have enough space to store the output line (as with the usage of `buf[]` above, we used a max buffer size of 100).

Errors: If the user specifies a `depth` or `width` outside these limits, your program should detect this and terminate.

Recursion: Give careful thought to how you can specify "sub-problems" for recursion to handle. Let's take a look at the first example again reproduced below. However, I've put in **bold** the subsections produced from `Section 1`.

Recall that this outline has `depth=3` and `width=2.` What can we observe about the text in bold?

- It is a sub-outline with `depth=2` and `width=2`.
- It uses letters to designate its top-level sections.
- Each label is an extension of the label of its *parent* section label (underlined):

```
Section 1.A
Section 1.B
```

or you could say that the label of the parent section is a *prefix* of the child section labels.

```
Section 1
    Section 1.A
        Section 1.A.1
        Section 1.A.2
    Section 1.B
        Section 1.B.1
        Section 1.B.2
Section 2
    Section 2.A
        Section 2.A.1
        Section 2.A.2
    Section 2.B
        Section 2.B.1
        Section 2.B.2
```

So… think carefully about how to structure your recursive function; what parameters it needs, etc.

Items to Submit

Your source file `outline.c`

Your `makefile` (same makefile as used for parts I and II).