

Solving a Row in a Nonogram

Your task this week is to solve a logic problem and to print the solution. The logic problem forms a small part of solving a type of puzzle often called a Nonogram, in which a player tries to fill in a pixelated, black-and-white image based on lists of the lengths of the contiguous black regions in each row and column. In particular, given the number of pixels in up to four regions and the number of pixels in the corresponding row, you must identify those pixels known to be part of one of the regions.

The objective for this week is for you to gain some experience with control structures in C, particularly with loops and conditional statements.


Background

A sample of a simple Nonogram appears to the right. The image is 5x5 pixels. The sizes of contiguous black regions in each row are listed to the left of the row. These regions appear in order from left to right in the image. And the sizes of contiguous black regions in each column are listed at the top of the column. These regions appear in order from top to bottom in the image.







To solve the puzzle, a player must use the region sizes and orders to determine whether each pixel in the image is black or white. Players usually start with individual row or column constraints, which are the easiest to understand (and are the only part that concerns you for this assignment).

Consider the top row of the sample Nonogram: the row is five pixels wide, and there is one region of five contiguous black pixels (written as “5” to the left of the row). Obviously, the region comprises the entire row.

The fourth row from the top is slightly more complex. The row is again five pixels wide. However, there are now three regions, each consisting of one black pixel (written as “1, 1, 1” to the left of the row). Any two adjacent regions must be separated by at least one white pixel, as otherwise the regions are contiguous and represent a single region (a contradiction). The total number of pixels required is thus five: three for the black pixels in the regions, and two white pixels to separate the three regions. Only one solution exists, as shown below.

1, 1, 1 

The region sizes may not always suffice to identify all pixels in the row or column. For example, the region sizes for the middle row in the sample Nonogram allow the six possible solutions below.

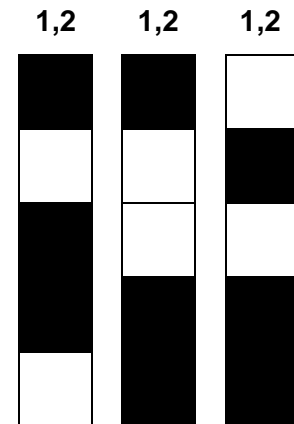
1, 1		1, 1	
1, 1		1, 1	
1, 1		1, 1	

	5	1,1	1,2	1,1	5
5					
1, 1					
1, 1					
1, 1, 1					
5					

Note that each of the pixels in the row can be either black or white, depending on which of the six solutions is the correct one. The region sizes alone, in this case, provide no information about any of the pixels!

As a final example, consider the region sizes of the middle column in the sample. Three possible solutions exist, as shown to the right. Notice again that, for most of the pixels, the color of the pixel depends on which solution is the correct one. However, the fourth pixel from the top must be black.

In some cases, such as the sample Nonogram given here, applying all row and column constraints as just discussed is sufficient to completely solve the puzzle. More generally, however, iterative application of these rules and even more sophisticated methods are needed. Fortunately, for this MP, you need only apply the constraints for a single row and determine which pixels, if any, are known to be black based on the region sizes for the row.



The Task

You must write the C function specified by the following:

```
int32_t print_row (int32_t r1, int32_t r2, int32_t r3, int32_t r4,
                  int32_t width);
```

The first four parameters are region sizes, allowing the row to have as many as four regions. All but the first region may be size 0, in which case the region does not exist. For example, given the row constraints specified as “1, 2” in the sample, your function executes with parameter values **r1=1**, **r2=2**, **r3=0**, and **r4=0**. The final parameter passed to **print_row** is the width of the row in pixels. In the sample Nonogram, **width=5**.

In some cases, the specified regions (and any necessary pixels in between) may not fit into the given **width**. In such cases, your function should print nothing, and return 0.

If the regions fit, your function should determine which pixels are known to be black, then print the row to the monitor, using 'x' (capital x) to represent known black pixels, and '.' (period) to represent other pixels. After printing the row, your function should print a newline character ('\n').

A few examples appear below. In each case, the program (“**onerow**”) is invoked with the four region sizes and the row width. Only in the first example does the function return 0 to indicate failure; for all other examples below, the function must return 1.

```
./onerow 3 5 2 4 16 [ returns 0; regions require at least 17 pixels ]
./onerow 3 5 2 4 17 prints  xxx.xxxxx.xx.xxxx (and a newline)
./onerow 3 5 2 4 18 prints  .xx..xxxx..x..xxx. (and a newline)
./onerow 3 5 2 4 19 prints  ..x...xxx.....xx.. (and a newline)
./onerow 3 5 2 4 20 prints  .....xx.....x... (and a newline)
./onerow 3 5 2 4 21 prints  .....x..... (and a newline)
./onerow 3 5 2 4 22 prints  ..... (and a newline)
```

Pieces

Your program will consist of a total of three files:

mp3.h	This header file provides function declarations and a brief description of the function that you must write for this assignment.
mp3.c	The main source file for your code (you must write it yourself). Include the mp3.h header file and be sure that your function matches the one in the header.

A second file is also provided to you:

main.c	A source file that interprets commands and calls your function.
---------------	---

You need not read this file, although you are welcome to do so.

Specifics

You should read the description of the function in the header file before you begin coding.

- Your code must be written in C and must be contained in a file named **mp3.c**. We will NOT grade files with any other name. **You may not make changes to other files.** If your code does not work properly without such changes, you are likely to receive 0 credit.
- You must implement the **print_row** function correctly.
 - You may assume that all region sizes and the width are between 0 and 50.
 - You may further assume that **r1** and the width are at least 1.
 - You may want to start by assuming that all four regions exist.
- Your routine's return values and outputs must be correct.
- Your code must be well-commented. Follow the commenting style of the code examples provided in class and in the textbook.

Compiling and Executing Your Program

Once you have created the **mp3.c** file and written the **print_row** function, you can compile your code by typing:

```
gcc -g -Wall main.c mp3.c -o onerow
```

The “**-g**” argument tells the compiler to include debugging information so that you can use **gdb** to find your bugs (you will have some).

The “**-Wall**” argument tells the compiler to give you warning messages for any code that it thinks likely to be a bug. Track down and fix all such issues, as they are usually bugs. Also note that if your code generates any warnings, you will lose points.

The “**-o onerow**” argument tells the compiler to name the resulting program “**onerow**”. If compilation succeeds, you can then execute the program as specified in the examples given earlier in this document.

The **onerow** program takes five command line arguments:

```
./onerow <r1> <r2> <r3> <r4> <width>
```

The first four arguments are the region sizes, and the fifth is the row width in pixels. If the arguments given are not valid (according to the assumptions given in the “Specifics” section), the code in **main.c** responds without calling your function.

Grading Rubric

We put a fair amount of emphasis on style and clarity in this class, as reflected in the rubric below.

Functionality (70%)

- 5% - Function returns 0 when regions cannot fit, and 1 when they can.
- 10% - Number of characters output corresponds to given width (plus a newline).
- 5% - Correct characters ('x' and ' . ') are used to represent pixels.
- 35% - Function works correctly when all four regions exist.
- 15% - Zero-length regions handled correctly.

Style (15%)

- 15% - Common code used to handle 0-length regions (does not rewrite the entire function based on how many regions exist).

Comments, Clarity, and Write-up (15%)

- 5% - introductory paragraph explaining what you did (even if it's just the required work)
- 10% - code is clear and well-commented, and compilation generates no warnings (note: any warning means 0 points here)

Note that some categories in the rubric may depend on other categories and/or criteria. For example, if you code does not compile, you will receive no functionality points.