1 Min Heaps (A2 Section)

For the first part of the assignment, you'll be implementing min-heaps using arrays as discussed in class. The relevant files in the starter code for this section are `minheap.h`, `minheap.c` and `test_minheap.c`.

First look at `minheap.h` to see how the data structures are stored for the min-heap implementation, and understand what's going on. In particular we are having to modify our structure to support fast `decrease_priority()` operations. The header file explains what these changes are.

Once you are familiar with the data layout, open `minheap.c` and complete the functions there to manipulate the heap. Be thoughful about how you write a code, writing some common helper functions can greatly reduce the amount of work you need to do.

Lastly, the `test_minheap.c` file provides some basic tests for your implementation. These are **not comprehensive**, and you should test for correctness yourself by adding more thorough test cases and tracing portions of your algorithm by hand to compare. You will need to use your min-heap implementation for the next section of the assignment, so make sure that your implementation is correct before you begin. You don't want to be debugging multiple different parts of your code when something goes wrong.

Marking Scheme (10 marks for this section)

- 1 marks: `newMinHeap()`
- 3 marks each for: `heapPush()`, `heapExtractMin()`, `heapDecreasePriority()`
- You will receive 0 marks if your code does not compile, or fails all the test cases, with possible penalties for compiling with warnings.

2 Pixel Marcher (A3 Section)

Some of you might have heard of the Code Mangler, now we introduce his less-villainous cousin - the *Pixel Marcher*. Given some image, his mission is to walk from the top-left corner to the bottom-right corner. Every step he takes costs him some energy (which is determined by a weight function that is provided to you). The *Pixel Marcher* wants to find the path that takes up the least energy.

The weight function which is provided is the key element that decides this least-energy path. For example, here is an input image with the output produced given 2 different weight functions:

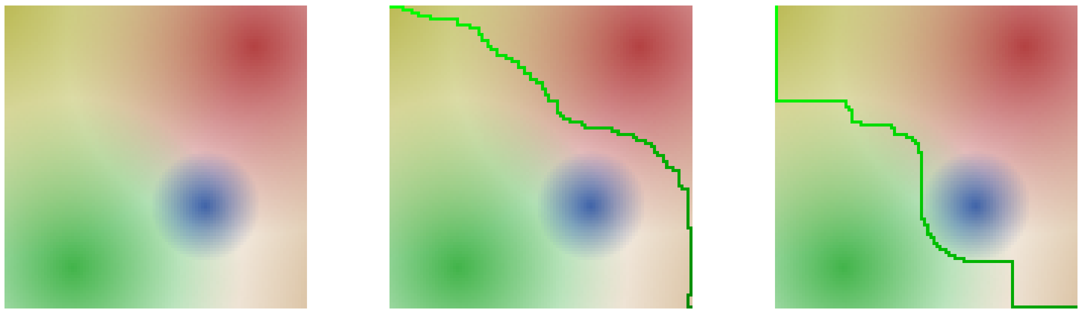

Figure 1: (Left) Input image, (Middle) Weight function 1, (Right) Weight function 2

In the above example, weight function (1) reflects how similar the pixel is to it's neighbour, and weight function (2) reflects how close the neighbouring pixel is to white. Both of these weight functions are provided to you as part of the starter code in the `test_marcher.c` and `driver.c` files.

Fun fact: You can use this algorithm exactly to help the Pixel Marcher solve mazes if you pick the correct weight function! Some small mazes have been provided as test cases for you.

For this section of the assignment, you have two tasks (Look at `marcher.c`):

(i) Given an input image and a weight function, complete the `findPath()` function that helps the Pixel Marcher find a least-energy path, and also the amount of energy he needs to spend to fulfill his mission.

(ii) Complete the weight function `allColourWeight()` so that:

- When `findPath()` is run with this weight function and the image `25colours.ppm`, the Pixel Marcher goes through at least 1 pixel of each of the 25 colours along the least-energy path.
- The Pixel Marcher can stay on pixels of the same colour for as many steps as needed, but once he leaves a colour, he can *never* step on any pixels of the same colour again. (He's very specific about this)
- The energy required to go between any two pixels is always **non-negative**.

The starter code provides comprehensive input specifications and requirements for both functions - so make sure you read everything before you ask any questions.

The starter code also has an `expected` folder contains images displaying least-energy paths from the test cases for you to compare against. These paths may not necessarily be unique, any other shortest path with an equivalent weight is fine.

Marking Scheme (10 marks for this section).

- Up to 7 marks can be earned by passing the test cases for `findPath()`. You may be tested on new images/weight functions. Each test case gets 1 second on the BV computers (though you will probably never reach this).
- 3 marks can be earned for having a correct solution to `allColourWeight()`. The order in which you visit the colours does **not** matter. Your function will be tested with *my* solution for `findPath()`, and must produce the correct result. There will be no part marks.
- You will receive 0 marks if your code does not compile, or fails all the test cases, with possible penalties for compiling with warnings.