C Final Test
Region Detection

Tuesday 9th June 2015, 14:00

THREE HOURS

(including 15 minutes planning time)

- Please make your swipe card visible on your desk.
- After the planning time, log in using your username as **both** your username and password.

The maximum mark is 30.
The test has three parts.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of C's various types and standard library functions.

Important note: THREE MARKS will be deducted from solutions that do not compile. Comment out any code that does not compile before you leave. Commented-out code will not be marked.

1 Introduction

This exercise invites you to develop a simple image processing application that identifies rectangular *regions* within a greyscale image. The image is represented as a two-dimensional array of $M \times N$ pixels ("picture elements"), $M, N > 0$, each of which is a non-negative integer between 0 and 255 representing the pixel's shade, where 0=black, 255=white. Each rectangular-shaped region comprises an $m \times n$ array of pixels, for some $0 \leq m < M, 0 \leq n < N$, within the image, and aligned in the vertical and horizontal directions (i.e. they are not rotated). One region may be wholly contained within another, but regions boundaries are not allowed to intersect. This means that each pixel belongs to at most one region boundary.

An example image of size M by N pixels is shown in Figure 1(a) involving just black (pixel value 0) and white (pixel value 255) regions. The image itself implicitly defines a top-level region (shaded black) and seven sub-regions arranged in a hierarchy. The outermost region contains two subregions shaded in white. The largest of these contains two subregions shaded in black and the largest of these contains three subregions shaded in white. Note that the non-overlapping property above implies that each region is surrounded by a 'halo' of pixels in its immediately enclosing region (the boundaries of two regions cannot be aligned).

The objective of the exercise is to analyse an image in order to detect each of the regions it contains. The output from this analysis will be an ordered list of C `struct`s, with each `struct` representing one region. A final objective, which will serve to test your code, is to re-draw the regions in the list but using a shading scheme that shades each region according to its nesting depth within the original image; the depth-0 region is defined to be the outermost region. For example, if regions at depths 0, 1, 2 and 3 are re-drawn in four shades graduating from white to black respectively then the image of Figure 1(a) might be re-drawn as shown in Figure 1(b).

2 Detecting regions

Detecting a region within an image involves initially identifying the top-leftmost pixel of that region. The dimensions (extent) of the region can then be determined by scanning the image in the horizontal and vertical directions until there is a change in pixel value. Recall that region boundaries are not allowed to overlap, so calculating the extent is fairly straightforward. The tricky part is the determination of the top-leftmost pixel. To illustrate the problem consider the two (black) regions R1 and R2 in Figure 2(a). The location of R1 is easy to determine by scanning the enclosing (white) region from left to right and top to bottom, starting from its top-leftmost pixel (this would be pixel (0,0) in the top-level region, for example). The extent of R1 is then easily found as described above. What about R2? The problem is that R1 "gets in the way" when we continue to scan the rest of the outer (white) region looking for R2's top-leftmost pixel. In general there may be many regions that "occlude" the region we're looking for in this sense.

There are many ways to solve this problem. In this exercise we're going to solve it by "erasing" a region as soon as we have detected it. In the example, we're going to "paint", i.e. *fill in* R1 with white pixels before continuing the search for R2. If we do this then the next black pixel that we encounter will be the top-leftmost pixel of R2: problem solved! The catch is that R1 may itself contain embedded regions and these in turn may contain others and we'll need to detect these regions before we can erase R1. In general this will need to continue *recursively*, as the nesting of regions within regions may be arbitrarily deep. In the example we see that R1 contains a region R3 (which is painted white), so before we can erase R1 we first need to locate all regions lying within it. The recursive search inside R1 is just like a search in R1's enclosing region except that:

1. The background shade is now black
2. The top-left and bottom-right pixel locations are now defined by the extent of R1, rather than the extent of the outermost enclosing region

The recursive search inside R1 will end up locating R3 and this will be erased, in this case by painting it black. The modified image after this erasure is shown in Figure 2(b). At this point every region inside R1 has been located, and erased, so R1 can now be itself be erased, this time by painting it with white

pixels, i.e. the colour of its own enclosing region. The situation at this point is shown in Figure 2(c). The scan can now continue from where we left off, i.e. from R1's (former) top-leftmost pixel position and the next non-white pixel that is found will be the top-leftmost pixel of R2. Following the same rules, R2 will be recursively searched for nested regions and then it, in turn, will be erased. When all regions have been located and erased the image will be entirely "blank", i.e. comprising only background pixels.

Note that the scan presented in this exercise to detect regions mutates the original image by erasing regions. Therefore you should *clone* the original image before invoking the detection algorithm. However, in this application you do not need to clone as the input image is always clone for you before the scan algorithm is applied.

2.1 The region list

As regions are located using the above scheme they need to be recorded somewhere and for this purpose we will use a list, which is initially empty and which grows as regions are located and added to the list. The order of the regions within the list is important as we'll later need to re-draw the regions in sequence (see below). The ordering is such that $R1 < R2$ if the top-leftmost pixel of region R1 appears before that of R2 in "scan order", i.e. proceeding from left to right and top to bottom through the image in which they are contained. For example a region starting at pixel location (15,9) comes before one starting at (3,11) in the ordering because it is nearer the top of the image and one starting at (5,10) comes before one starting at (17,10) because it is nearer the left border of the image. The list insertion function, which you will be required to write, must respect this ordering.

3 Given `structs` and `typedefs`

The file `typedefs.h` defines the following `structs` and `typedefs` for defining points and regions:

```
// Struct defining a point in two-dimensional space
typedef struct point
{
  int x;
  int y;
} point_t;


// Struct defining the size of a two-dimensional region
typedef struct extent
{
  int width;
  int height;
} extent_t;


// Defines a region
typedef struct region
{
  point_t position;
  extent_t extent;
  int depth;
} region_t;
```

These should be self-explanatory from the above description, but note the additional `depth` field within `region`. This should record the nesting depth of the region assuming that the outermost region has depth 0. For example in Figure 2(a) above R1 and R2 have depth 1 and R3 has depth 2. Note that the depth of a region is easily determined by increasing the depth of the region we are currently inspecting as we recurse down through the region hierarchy.

The following `structs` and `typedefs` are also provided in the file `typedefs.h` for maintaining doubly-linked lists:

```
// The internal element type used by a list_t
typedef struct list_elem
{
  struct list_elem *prev;
  struct list_elem *next;
  struct region *region;
} list_elem_t;


// A list of regions
typedef struct list
{
  list_elem_t *header;
  list_elem_t *footer;
} list_t;
```

These should be familiar to you from your lecture notes. Various familiar list processing functions are provided in `list.c`; others you are required to define as part of the exercise.

4 What to do

You are given two files, `list.c` and `region.c` that you should modify by implementing the various missing functions. There are also several additional files providing useful utilities and testing functions. DO NOT MODIFY THESE FILES!

You should use the structures and function templates defined, though you may define and use appropriate auxiliary functions should you wish. These should be appropriately scoped, however, and should not be visible outside the module in which they are defined. You may compile your code using the provided `Makefile`. By default your code will be compiled with debugging symbols (via `gcc`'s `-g` option) so you will be able to debug your code with a capable debugger – `gdb` or `eclipse`, for example.

The `Makefile` will instruct `make` to build an executable program called `regions` which will print out a textual summary of the regions located in a given input image file. Various PGM (Portable Greymap Format) image files are provided in the `images` directory.

Helper functions have been provided for performing all of the dynamic memory allocation you will require during this exercise. You can assume that these functions never return `NULL` and that deallocation is identical to that of memory allocated using `malloc`. If for some reason you need to use C's dynamic memory allocation, ensure that you check for failure and handle it accordingly, such as printing the cause and exiting the program.

Part I – Region list management

The following two functions should be implemented in the file `region.c`.

1. Define a function `int region_compare(const region_t *r1, const region_t *r2)` that will return true (1) if region `r1` is less than region `r2` in (y, x) "lexicographical order", as defined in Section 2.1 above; false (0) otherwise. The header file `test_regions.h` defines an array of regions (`test_regions`) that you can use for testing purposes. The regions in the given `test_regions` array are actually ordered (y, x) lexicographically so, for example, `region_compare(test_regions[1], test_regions[2])` should return 1 and `region_compare(test_regions[5], test_regions[3])` should return 0.

   Hint: You may find it helpful to implement the function `point_compare_less`.

   [2 marks]

2. Define a function `void print_regions(FILE *out, list_t *regions)` that will print a summary of each region in a given list of regions using the predefined function `print_region` provided in `region.c`. **[3 marks]**

The following functions should be implemented in the file `list.c`.

1. Define a function `void list_insert(list_iter iter, region_t *region)` that will insert a given region into a given list immediately before the element pointed to by `iter`. Recall that a list iterator (`iter`) is a pointer to a list cell (`list_elem_t`) that contains a pointer to a region and `prev` and `next` pointers to adjacent cells. [**4 marks**]

2. Using `int region_compare` and `list_insert` define a function `void list_insert_ascending(list_t *list, region_t *region)` that will insert a given region into a given *ordered* list of regions, preserving the ordering, where the ordering is as defined by `region_compare` above. Assume that the list is already initialised (no null pointers!). The file `check_list_functions.c` also contains functions that you can use to test your list implementation. You can exercise your functions by running the `check_list_functions` executable that is built by the supplied Makefile. e.g.

   ```
   % ./check_list_functions
   ```

   [**2 marks**]

3. Define a function `void list_destroy(list_t *list)` that will free the memory used by the given list. The regions contained within each list cell should also be freed (you may find it helpful to implement the function `region_destroy` defined in `region.c` if you wish to do this in a higher-order manner).

   [**4 marks**]

Part II – Region detection

The following functions should be implemented in the file `region.c`.

1. Define a function `void image_fill_region(image_t *image, const region_t *region, uint8_t value)` that sets the specified region of the given image to the given pixel intensity value (shade). Note that two functions are provided in `image.c` to get and set the value of of pixel `x,y` in a given image (recall that `y` is the row index and `x` the column index):

   ```
   uint8_t get_pixel(image_t *image, int x, int y);

   void set_pixel(image_t *image, int x, int y, uint8_t colour);
   ```

   Both `get_pixel` and `set_pixel` will throw assertion failures for out of bounds accesses. Such failures indicate bugs in your implementation.

   [**1 marks**]

2. Define a function `void find_extent(extent_t *extent, image_t *image, const point_t *position)` that computes the extent of a region within the given image. The location of the top-leftmost pixel of the region is defined by `position`. You can use `init_extent` to reset the `extent_t` to its new `width` and `height`.

   [**2 marks**]

3. Define a recursive function `void find_sub_regions(list_t *regions, image_t *image, const region_t *current)` that will find all (sub)regions of a given "current" region within an image using the algorithm described in Section 2. As you locate regions you should add them to the given list (`regions`) using the `list_insert_ascending` function you defined earlier. The top-level call to `find_sub_regions` is contained in a function `find_regions` that has been defined for you:

   ```
   void find_regions(list_t *regions, image_t *image)
   {
     region_t *image_region = region_allocate();

     image_region->depth = 0;
   ```

```
  init_point(&image_region->position, 0, 0);
  init_extent(&image_region->extent, image->width, image->height);

  list_insert_ascending(regions, image_region);
  find_sub_regions(regions, image, image_region);
}
```

Note that the top-level call is primed with the original image and the outermost region (`image_region`), that has depth 0. Remember that at each recursive sub-call to `find_sub_regions` you need to increase the depth of the "current" region by 1. Note also that to implement the erasure process described in Section 2 you will need to overwrite the pixels in the given image using `image_fill_region` function that you defined earlier.

[9 marks]

4. Define a function `void render_regions(image_t *image, list_t *regions, colour_function_t get_colour)` that will "paint" a list of regions by adding them one by one to an initially blank `image`. Use the `image_fill_region` you defined earlier to fill in each region in turn. The `colour` (shade) used for each region should be determined by its depth. The function that maps depths to colours is provided in the `get_colour` parameter. Note that this is a function pointer (akin to a higher-order function in Haskell!). For this to work correctly the list of regions must be ordered according to the `region_compare` ordering function you defined earlier and it is a precondition of the function that the given list is ordered in this way. This ensures that if you traverse the list in order then the regions will be painted from the background toward the foreground and this will result in nested regions being painted on top of previously painted enclosing regions. You can test your function using the sample images provided in the `images` directory, e.g.

```
% ./regions ../images/input.pgm
% ./regions ../images/input1.pgm
% ./regions ../images/input2.pgm
% ./regions ../images/input3.pgm
% ./regions ../images/input4.pgm
% ./regions ../images/input5.pgm
% ./regions ../images/input6.pgm
```

Running `regions` generates an image file `output.pgm` containing the re-rendered regions that can be opened in most image viewers, and a `regions.txt` file containing a listing of the regions detected in the image (also printed to standard output). You can find reference outputs for some of the supplied inputs in the `reference_outputs` directory.

[3 marks]

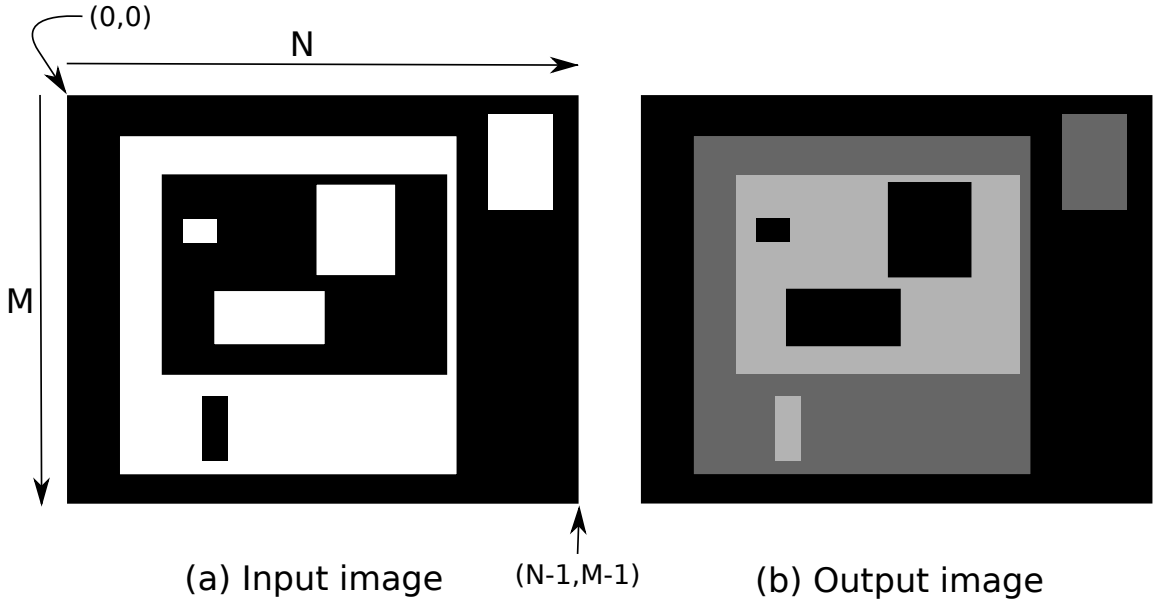
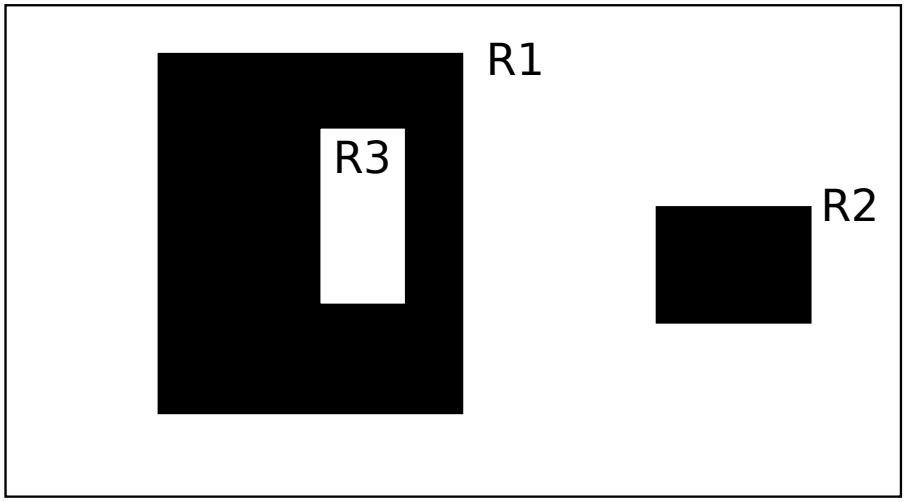
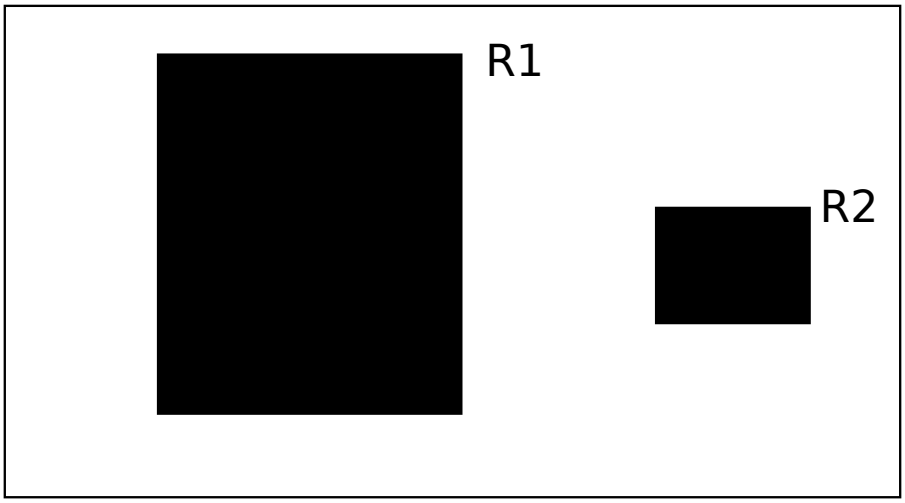


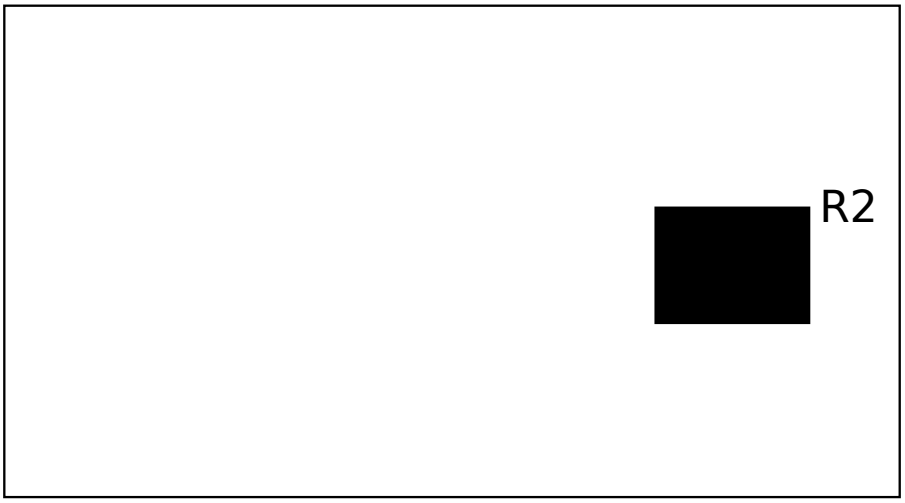
Figure 1: Example input and output images



(a)



(b)



(c)

Figure 2: Erasing regions