COMP 273, Winter 2022 - Assignment 3

School of Computer Science
McGill University
Available On: February 28th, 2022
Due Date: March 21th, 2022. 11:59pm.
(late policy: 10% off per day late, up to 2 days late. 0 After that.)
Sharing code is strictly prohibited.
Make sure to follow format instructions carefully!

Things to consider

- Welcome to Assignment 3! We will be going through some fun image manipulation exercises in this one, developing some tools for your image processing toolbox. **An A3 tutorial will be uploaded to Mycourses in a few days where we will go through several steps of the assignment.**
- Make sure to follow conventions. **We will use auto-graders for A3 and the default conventions should not be changed**. Please get familiar with the breakpoints, pause ... in MIPs: Mips Debugging
- YOU WILL BE GRADED ON WHAT YOU SUBMIT IN YOUR ASSIGNMENT FOLDER AND IT IS YOUR RESPONSIBILITY TO CHECK THAT THE FILES YOU INTENDED TO SUBMIT ARE ACTUALLY IN THAT FOLDER!
- For this assignment, **you need to put your Mars in the same directory as the assignment folder, Settings -> Assemble all files in the directory.** Please don't zip your Mars and main.asm upon submission stage.
- GIMP is an image editing software. Though you can use anything you'd like, GIMP is a free software that can be used to view the .pgm file after generating the image is complete.
  https://www.gimp.org/
- In this assignment we will be working with '.pgm' files. Headers of .pgm files have a specific format.
  $P2$
  $width\ height$
  $maxValue$
  or
  $P5$
  $width\ height$
  $maxValue$
  PGM Format Explain contains more information about proper '.pgm' formatting. In this assignment we will only use pgm files with a maximum value less than 256. So each value can be stored in one byte.

  P2 images are encoded in ASCII value, to see how it works, open feepP2.txt and see that the first 3 rows containing the header info, and the remaining text are the contents for the pixels. Now change the suffix of .txt to .pgm, your text should convert to an image now. So when you read P2 images into the image struct, you need to convert ASCII to integers. Similarly, when you write to P2 images, they should be converted to ASCII values. P5 images are easier to deal with, for its contents you can read it directly using syscalls, they will automatically convert to integers. *For P5 images, you may need to use GIMP to view it correctly, for Mac users Preview will fail.*

- You may add any extra .data you like, but make sure you don't remove any existing data or labels. The data specified in the templates should be used accordingly.
- For all the questions below, we will test your codes with different size images. Instead of hard-coded your buffer, dynamically allocating your array will be a better choice, you are free to use the *malloc.asm*.
- For all the questions below, you can choose to 1) rewrite the content in the input image struct or 2) create a new image struct and work from there, you can choose whichever is convenient for you, as long as the return address contains desired contents.

1 Reading and writing an Image (30 marks)

In this section, we will implement functions to read and write image files where these image files are stored in ASCII text format. To accomplish this task, you need to write helper functions that convert between ASCII values and integers. They will become useful later when reading/writing P2 images.

(a) Write a helper procedure called '***str2int.asm***' that takes the ASCII value and converts it to the integer. i.e. "32" -> 32. The input is an address that points to the beginning of the ASCII digits, with a *null* terminator, and the output is the converted integers. You can safely assume that the numbers you have to handle are non-negative, and the input ASCII integers consist of valid digits from 0-9. Make sure it can convert up to 3 digit integers. i.e. "999". (*hint: You need to use this helper procedure in readImage.asm*)

(b) Write a helper procedure called '***int2str.asm***' that takes the integer value and converts it to a string. i.e. 32 -> "32". The output string will be blank space terminated. You can safely assume that the input integer is non-negative, and they are valid integers. Make sure it can convert up to 3 digit integers. i.e. "999". (*hint: You need to use this helper procedure in writeImage.asm*)

(c) Implement '***readImage.asm***' provided in the template, which takes in a '.pgm' image filename as input and reads the contents into a struct. An image will be read into a data structure containing information about the image an array of bytes containing the contents of the image.

Specifically, an image should be represented as the following struct.

```
struct image{
    int width;                          // number of columns
    int height;                         // number of rows
    int maxValue;                       // maximum value of list
    char contents [width*height]; // image content as an array
}
```

Where contents is a `char` array of size *width* × *height* which stores values of the image as a 1D array. Your subroutine should return the address of the image struct. The subroutine should support 'pgm' images of type 'P2' and 'P5'. Note that while contents is a 1D array it represent a 2D image. Note that 2D arrays (or ND arrays in general) are an abstraction provided by high language like C or java while in reality they are represented as 1D array in the memory. As you know, in a language like Java or C, we would simply specify array positions as *array*[i][j]. That is, we would let i represent the row we are currently at, and j represent the column we are currently at. In MIPS, however, our 2D array is stored as values in a 1D array. It is clear to see that for any position [i,j] in our 2D array, we can retrieve this position by simple computing (i * width) + j. Since i represents rows, whenever we add a width we are essentially going to the next row in our conceptual 2D array. j simply represents which column we are currently looking at. In the context of an image, value of image array at a location say [i,j] is referred to as the *pixel value* of the image at [i,j] and location [i,j] is called the *pixel location*.

(d) Implement '***writeImage.asm***' which takes the image struct and a filename (of the form 'abcd.pgm') and inputs and writes it to a '.pgm' (P5) file (and a third input which may be 0 or 1. If the input is 0 the function should write a 'pgm' file of type P5 otherwise it should write a image of type P2). To write an image, you will need to write the header:
$P2$
$width\ height$
$maxValue$
...
or

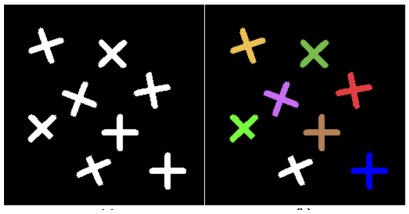
$P5$
$width\ height$
$maxValue$
...

Where "..." are the contents that was read into the buffer. It should then close the file. Note that for P2 file, you will need to use *int2str.asm* to convert the integer back to a sequence of ASCII digits. You can verify that your subroutine is working correctly by opening the written '.pgm' file in GIMP. *(For P5 images, you may need to use GIMP to view it correctly, for Mac users Preview will fail.)*

(e) If you implement Q1 correctly, you should be able to read a P2 image and write as a P5 image, and vice versa.

2 Connected Components (25 marks)

Connected components in a 2D image are defined by clusters of pixels with the same label value, which are connected to each other by 4-pixel or 8-pixel connections. 4 connected structures consider only left-right (horizontal) or up-down (vertical) directions for neighbors (Fig. 1(b)). In this section, you will implement the 2-Pass algorithm to detect the number of connected components and assign each component a unique label value. We will use p_{top} and p_{left} to indicate the upper and left neighbors of the current pixel p_i, and l_{top} and l_{left} to denote the labels they get during the algorithm. Foreground pixels will be pixels with non-zero values, and background pixels are pixels equal to 0.



(a) Each connected component is assigned with a color, for this assignment the colors are equivalent to different pixel values.

(b) Red pixel indicates p_i, and the gray pixels corresponds to p_i's upper and left neighbours.

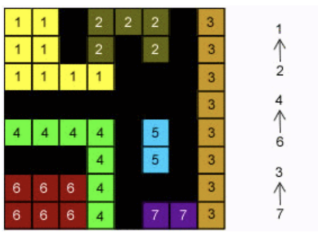
Figure 1: Connected Components

Algorithm 1 2-Pass (4-connectivity)

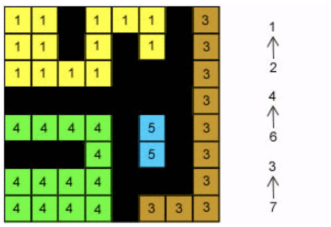
```
1: //Pass One
2: for p_i every pixel in image do
3:     if p_i is not 0 then
4:         if both labels l_top and l_left are 0 then
5:             Create a new label value l_i, and assign it to p_i
6:         else if only one of l_top and l_left is 0 then
7:             Set l_i = max(l_top, l_left).
8:         else if l_top and l_left are both non-zero and different then
9:             Set l_i = min(l_top, l_left), and store the labels l_top and l_left as equivalent labels;
10:        end if
11:    end if
12: end for
13: //Pass Two
14: for p_i every pixel in image do
15:    Scan the image and replace each label l_i by the lowest label in its equivalent labels.
16: end for
```

(a) The '***connectedcomponents.asm***' will take the image struct as input. You can directly work with the current struct and change its contents. The outputs will be the address of the struct, v_0 which contains the a "label version" of the image, and the number of connected components, v_1, found in the image. We will implement the **4-connectivity** way.

(b) After Pass One, each connected component will have a group containing its equivalent labels, and you can allocate new space in your code to store these equivalent groups. In Pass Two, you need to scan each group and assign the lowest label to the connected component. You are free to come up with any data structure that will do the job. An easy way is to create a local buffer K and use it as an array, where each element has a fixed size (i.e. 20 bytes) and you'll choose a delimiter to divide each element. You will move between arrays using a fixed offset you define. You can safely assume that the total number of labels appearing in the algorithm will be less than 256, so you can store it within 1 byte. We will test your code with up to 10 connected components.



(a) You need to find the connected components of foreground pixels (non-black pixels). After Pass 1, the equivalent labels are (1,2), (3,7) and (4,6).



(b) After Pass 2, each connected component is assigned with the smallest label in its equivalence group.

Figure 2: 2 Pass algorithm

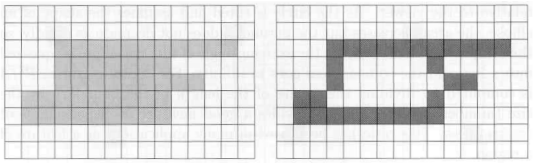
3 Region Boundary (15 marks)

The boundary of a region R is the set of pixels in the region that have one or more neighbors that are not in R. One can find the boundary of a region using 4-neighbors, by examining the top, bottom, left, right neighbors of the current pixels, or using 8-neighbors, that use all 8 neighboring pixels surrounding p_i. We will implement the **4-neighbors** way.

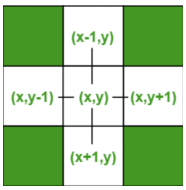
Algorithm 2 Boundary labelling

```
for p_i every pixel in image do
    Examine the neighbors (4-neighbors) of p_i
    if At least one of the neighbors is a background pixels then
        Mark p_i as a boundary pixel.
    end if
end for
```

(a) In this section we will implement '***boundary.asm***', that uses just 4-neighboring pixels. The input is the image struct a_0, and the output is an image struct with pixel values (0 or 1), where the boundary pixels are set with 1. *(Hint: it's easier to write a new image struct because you don't have to worry about replacing the interior pixels.)*



(a) Boundary pixels of connected components. If a pixel's neighbors intersect with a background pixel, then we say it is a boundary pixel.



(b) The upper, lower, left, right, consists of 4 neighbors of a pixel $p_i = (x, y)$. In 8 neighbors, all surrounding pixels of p_i need to be counted.

Figure 3: Boundary of a connected region

4 Transpose Image (15 marks)

Implement the '***transpose.asm***' which works as follows. It takes 1 input argument, that is the pointer to an image struct that contains the image. You will transpose the image, i.e., matrix transpose. The output should be an image struct that stores the contents of the transposed image. Note that you should write the correct header information to the new struct. You can use the ***writeImage.asm*** to test your image struct. The new image can only be viewed if you have properly written the header information and the contents.

Figure 4: Transpose of a matrix

5 Mirror Image (15 marks)

Write a procedure '***mirror.asm***' which works as follows. It takes the pointer to an image struct which containing the image. You will mirror the image **horizontally**. The output should be an image struct that stores the contents of the mirrored image. Note that mirroring does not change the header information, you only update the content. You can use writeImage.asm to test your image structure. New images can only be viewed if the header information and content are written correctly.

Assignment Submission Instructions

By handing in this your assignment you acknowledge that the work you are submitting is your own, and that you have read the COMP 273 FAQs document under "Content" in *mycourses*.

1. In addition to subroutine templates provided in *boundary.asm, connectedcomponents.asm, readImage.asm, writeImage.asm , mirror.asm, malloc.asm, transpose.asm, str2int.asm* and *int2str.asm*. You may additionally add as many helper subroutines as you want. You may also include additional 'asm' files containing helper functions. Please ZIP ***boundary.asm, connectedcomponents.asm, readImage.asm, writeImage.asm , mirror.asm, malloc.asm, transpose.asm, str2int.asm, int2str.asm***, and any additional helper files to the submission folder. However, do not include any images with your submission. Your submission will be tested on separate wider set of input images. Testing will be done automatically, so make sure you don't change any labels and follow register conventions. If you fail to follow the conventions or change labels in the templates provided you will loose marks.

2. Submit your solution to *myCourses* before the due date. Hints, suggestions and clarifications may be posted on the discussion board on *mycourses* as questions arise. Even if you don't have any questions, it is a good idea to check the discussion board.

3. Take all your '.asm' files and zip them. The zipped file should be named <studentID>.zip. If my student ID is 123456789, then the zip file to submit will be named *123456789.zip*. **Do not upload individual files or any other compression format**. Additionally, make sure to add your student ID as a comment on top of all the files you submit. It is your responsibility to ensure that you upload the correct version of all the files before the assignment deadline.

4. Copying code from other students or internet sources is strictly prohibited. The assignments will be checked for plagiarism using plagiarism detection tools. When dealing with plagiarism cases, we have no way of telling if you copied the code from someone or let others copy from you, so all students found in violation will be treated equally. Needless to say that you will be better off submitting your own partial work both in terms of grade and learning. However, if you are tempted to copy code from someone else, here is a guide on how to do it without getting caught.

5. Comment your code. If there are no comments, and the code is incorrect, you will get ZERO marks. If you make any special assumptions in your programs, or if you feel there are ideas that need explanation, describe them in your comments.

6. Your code *must* run and assemble, even if the final solution is not quite working. If something is not working, comment-out the broken parts of code and write a comment or two about what you expect to happen, what is happening, and what the problem may be. **You are expected to follow conventions. Doing so will save you time and a headache during debugging, so its great for everyone!** *If your code does not assemble you will receive 0 points for that question.*