COMP 202

Fall 2022

Assignment 3

Due: Monday, November 21^{st}, 11:59 p.m.

Please read the entire PDF before starting. You must do this assignment individually.

Question 1:	100 points
	100 points total

It is very important that you follow the directions as closely as possible. The directions, while perhaps tedious, are designed to make it as easy as possible for the TAs to mark the assignments by letting them run your assignment, in some cases through automated tests. While these tests will never be used to determine your entire grade, they speed up the process significantly, allowing the TAs to provide better feedback and not waste time on administrative details.

To get full marks, you must follow all directions below:

- Make sure that all file names and function names are **spelled exactly** as described in this document. Otherwise, a 50% penalty per question will be applied.
- Make sure that your code **runs without errors**. Code with errors will receive a very low mark.
- Write your name and student ID in a comment at the top of all `.py` files you hand in.
- Name your variables appropriately. The purpose of each variable should be obvious from the name.
- Avoid writing repetitive code, but rather call helper functions! You are welcome to add additional functions if you think this can increase the readability of your code.
- Lines of code should NOT require the TA to scroll horizontally to read the whole thing. If the line is getting way too long, then it's best to split it up into two different statements.
- Vertical spacing is also important when writing code. Separate each block of code (also within a function) with an empty line.
- **Up to 30% can be removed for bad indentation of your code, omission of docstrings, and/or poor coding style as discussed in our lectures.**

Hints & tips

- **Start early.** Programming projects always take more time than you estimate!
- Do not wait until the last minute to submit your code. **Submit early and often**—a good rule of thumb is to submit every time you finish writing and testing a function.
- Write your code **incrementally**. Don't try to write everything at once. That never works well. Start off with something small and make sure that it works, then add to it gradually, making sure that it works every step of the way.
- Read these instructions and make sure you understand them thoroughly before you start. Ask questions if anything is unclear!
- Seek help when you get stuck! Check our discussion board first to see if your question has already been asked and answered. Ask your question on the discussion board if it hasn't been asked already.

Talk to a TA during office hours if you are having difficulties with programming. Go to the instructor's office hours if you need extra help with understanding a part of the course material.

  - At the same time, beware not to post anything on the discussion board that might give away any part of your solution—this would constitute plagiarism, and the consequences would be unpleasant for everyone involved. If you cannot think of a way to ask your question without giving away part of your solution, then please drop by our office hours.

- If you come to see us in office hours, please do not ask "Here is my program. What's wrong with it?" We expect you to at least make an effort to start to debug your own code, a skill which you are meant to learn as part of this course. Reading through someone else's code is a difficult process—we just don't have the time to read through and understand even a fraction of everyone's code in detail.

  - However, if you show us the work that you've done to narrow down the problem to a specific section of the code, why you think it doesn't work, and what you've tried in order to fix it, it will be much easier to provide you with the specific help you require and we will be happy to do so.

Revisions

Nov. 9:

- An `AssertionError` should be raised instead of a `ValueError` for all functions that ask to raise an exception.
- The `crop` function should not modify the input list.
- Added an example of raising an `AssertionError` for `load_compressed_image`.
- Fixed the typos in the example for the `load_image` function and in the example shown for the compressed PGM format.
- Noted that, as discussed in our lecture, examples that contain a backslash in a docstring must have two consecutive backslash characters.
- Noted that, as discussed in our lecture, when using doctest the return value for a function must appear all on one line (even if it becomes a very large line).
- Noted that, for the loading functions, the file could have anything inside it, not necessarily a valid image matrix.
- Clarified that you cannot open compressed PGM files in image viewing programs.

Part 1 (0 points): Warm-up

*Do **NOT** submit this part, as it will not be graded. However, doing these exercises might help you to do the second part of the assignment, which will be graded. If you have difficulties with the questions of Part 1, then we suggest that you consult the TAs during their office hours; they can help you and work with you through the warm-up questions. You are responsible for knowing all of the material in these questions.*

Warm-up Question 1 (0 points)

Write a function `split_at_space` that takes as input a string of words with spaces in between each word and returns a list of the words in the string. Do not use the string method `split` to do this.

```
>>> split_at_space('aardvark butterfly carrot')
['aardvark', 'butterfly', 'carrot']
```

Warm-up Question 2 (0 points)

Write a function `generate_random_list` which takes an input an integer `n` and returns a list containing `n` random integers between 0 and 100 (both included). Use the `random` module to do this.

Warm-up Question 3 (0 points)

Write a function `sum_numbers` which takes as input a list of integers and returns the sum of the numbers in the list. Do not use the built-in function `sum` to do this.

Warm-up Question 4 (0 points)

Write a function `same_elements` which takes as input a two dimensional list and returns `True` if all the elements in each sublist are the same and `False` otherwise. For example,

```
>>> same_elements([[1, 1, 1], ['a', 'a'], [6]])
True
>>> same_elements([[1, 6, 1], [6, 6]])
False
```

Warm-up Question 5 (0 points)

Write a function `flatten_list` which takes as input a two dimensional list and returns a one dimensional list containing all the elements of the sublists. For example,

```
>>> flatten_list([[1, 2], [3], ['a', 'b', 'c']])
[1, 2, 3, 'a', 'b', 'c']
>>> flatten_list([[]])
[]
```

Part 2

The questions in this part of the assignment will be graded.

The main learning objectives for this assignment are:

- Apply what you have learned about lists and nested lists.
- Understand how to raise exceptions in cases of invalid inputs.
- Solidify your understanding of working with loops and strings.
- Apply what you have learned about file IO and string manipulation.
- Understand how to write a docstring and use doctest, in particular when working with files.
- Practice making copies of lists so that inputs are not modified.

Note that this assignment is designed for you to be practicing what you have learned in our lectures up to and including Lecture 22 (File IO II). For this reason, you are NOT allowed to use anything seen after that lecture or not seen in class at all. You will be heavily penalized if you do so.

For full marks, in addition to the points listed on page 1, make sure to add the appropriate documentation string (docstring) to *all* the functions you write. The docstring must contain the following:

- The type contract of the function.
- A description of what the function is expected to do.
- At least three (3) examples of calls to the function. You are allowed to use *at most one* example per function from this PDF.

Examples

For each question, we provide several **examples** of how your code should behave. All examples are given as if you were to call the functions from the shell.

When you upload your code to codePost, some of these examples will be run automatically to test that your code outputs the same as given in the example. However, **it is your responsibility to make sure your code/functions work for any inputs, not just the ones shown in the examples.** When the time comes to grade your assignment, we will run additional, private tests that will use inputs not seen in the examples. You should make sure that your functions work for all the different possible scenarios. Also, when testing your code, know that mindlessly plugging in various different inputs is not enough—it's not the quantity of tests that matters, it's having tests that cover all of the possible scenarios, and that requires thinking about possible scenarios.

Furthermore, please note that your code files **should not contain any function calls in the main body of the program** (i.e., outside of any functions). Code that contains function calls in the main body **will automatically fail the tests on codePost and thus be heavily penalized**. It is OK to place function calls in the main body of your code for testing purposes, but if you do so, make certain that you remove them before submitting. You can also test your functions by calling them from the shell. Please review what you have learned in our lecture on Modules if you'd like to add code to your modules which executes only when you run your files.

Safe Assumptions

For all questions on this assignment, you can safely assume that the **type** of the inputs (both to the functions and those provided to the program by the user) will always be correct. For example, if a function takes as input a string, you can assume that a string will always be provided to it during testing. The same goes for user input. At times you will be required to do some input validation, but this requirement will always be

clearly stated. Otherwise, your functions should work with any possible input that respect the function's description. For example, if the description says that the function takes as input a positive integer, then it should work with all integers greater than 0. If it mentions an integer, then it should work for *any* integer. Make sure to test your functions for edge cases!

Code Repetition

One of the main principles of software development is DRY: Don't Repeat Yourself. One of the main ways we can avoid repeating ourselves in code is by writing functions, then calling the functions when necessary, instead of repeating the code contained within them. Please pay careful attention in the questions of this assignment to not repeat yourself, and instead call previously-defined functions whenever appropriate. As always, you can also add your own helper functions if need be, with the intention of reducing code repetition as much as possible.

Question 1: Fun with images (100 points)

In this question we will work with greyscale images and perform simple operations on them: flipping, cropping, inverting, reading from disk[1], writing to disk, compressing and decompressing.

A digital image can be thought of as a two-dimensional grid of **pixels**: little squares, each with their own colour, arranged in rows and columns. You can see the individual pixels of an image for yourself by opening an image file on your computer and zooming in to the maximum level.

As you may know, images can be stored in many different kinds of file formats on a computer system. Some of the most popular image formats include PNG, JPEG and GIF. We will work with images of the **PGM format** for this question.

The PGM format is a very simple image format. Below we show the contents of a sample PGM file (`comp.pgm`, provided to you with this PDF, along with two other sample PGM files for you to experiment with: `dragon.pgm` and `mountain.pgm`[2].)

```
P2
24 7
255
0  0  0  0  0  0 0   0   0   0   0   0 0   0   0   0   0   0 0   0   0   0   0  0
0 51 51 51 51 51 0 119 119 119 119 119 0 187 187 187 187 187 0 255 255 255 255  0
0 51  0  0  0  0 0 119   0   0   0 119 0 187   0 187   0 187 0 255   0   0 255  0
0 51  0  0  0  0 0 119   0   0   0 119 0 187   0 187   0 187 0 255 255 255 255  0
0 51  0  0  0  0 0 119   0   0   0 119 0 187   0 187   0 187 0 255   0   0   0  0
0 51 51 51 51 51 0 119 119 119 119 119 0 187   0 187   0 187 0 255   0   0   0  0
0  0  0  0  0  0 0   0   0   0   0   0 0   0   0   0   0   0 0   0   0   0   0  0
```

The first three lines contain information about the image. The first line is always 'P2' – it indicates the formatting of the image file. The second line contains the width and height of the image (i.e., number of pixels in each row and number of rows). The third line indicates the maximum white value – we will use 255 for this assignment.

All subsequent lines of the file (starting from the fourth line) contain the data for the actual image. Each row of pixels of the image is on its own line: the fourth line is the first row of pixels, and the last line of the file is the last row of pixels. Each row contains a series of space-separated integers ranging from 0 to 255 (with any number of spaces separating each). Each integer is the greyscale colour for each individual pixel. A value of 0 means black and a value of 255 means white, while a value in between corresponds to some shade of grey.

We will write functions in this question that load in a PGM file into our Python program, using a matrix (nested list) of integers to store the pixel values, which we will call a **PGM image matrix**. We will also write functions that save a matrix of integers into a PGM file.

We will also deal with a second image format, called **compressed PGM format**. **Compression** is the process of taking a piece of data and performing some operation on it to reduce its size without losing any data. For instance, creating a zip file can make a file smaller. Compression is important especially when transferring files over the net, since a smaller file size will result in a faster transfer (and images are transmitted more than anything else over the net!).

There are many compression algorithms, some much more sophisticated and advanced than others[3]. As part of this question, we will implement a simple compression algorithm to reduce the size of a PGM file. For example, if we compress the image matrix from the file above, and then save the compressed matrix back to disk, we would get the resulting (smaller) file:

[1] Disk means the hard drive!

[2] Adapted from image files from `https://people.sc.fsu.edu/~jburkardt/data/pgma/pgma.html`

[3] For a discussion of the Lempel-Ziv compression algorithm, used as part of the GIF file format and also a simple one to understand, I encourage you to check out Security Now no. 205 at `https://twit.tv/shows/security-now/episodes/205`.

```
P2C
24 7
255
0x24
0x1 51x5 0x1 119x5 0x1 187x5 0x1 255x4 0x1
0x1 51x1 0x5 119x1 0x3 119x1 0x1 187x1 0x1 187x1 0x1 187x1 0x1 255x1 0x2 255x1 0x1
0x1 51x1 0x5 119x1 0x3 119x1 0x1 187x1 0x1 187x1 0x1 187x1 0x1 255x4 0x1
0x1 51x1 0x5 119x1 0x3 119x1 0x1 187x1 0x1 187x1 0x1 187x1 0x1 255x1 0x4
0x1 51x5 0x1 119x5 0x1 187x1 0x1 187x1 0x1 187x1 0x1 255x1 0x4
0x24
```

You will note some key differences. The first line is 'P2C' instead of 'P2', and the pixel values are no longer integers. Instead, they are strings of the form AxB, where A is an integer between 0 and 255 and B is any positive integer.

In a value of the form AxB, the A stands for the actual pixel value (i.e., shade of grey), and the B stands for how many times that pixel is repeated consecutively. For example, in the first row of the original image file (on the previous page), there were 24 zeroes. In the compressed image file, instead of having 24 zeroes with spaces in between each (for a total of 70 characters on the line), we have only the four characters '0x24' (meaning 0 repeated 24 times). Thus we have reduced the size of this line by over 90%! We see smaller improvements in the other lines because there are not as many consecutive numbers, but it is still a significant reduction in total file size (40% decrease overall).

Finally, while working on the functions for this question, you will find that it can help to open the PGM files created by your functions to view their contents. PGM files can be opened natively on Macs with the built-in Preview application. On Windows, there is no built-in support, though you can download programs (such as IrfanView) that can open them. We also provide a file `show_image.py` with this PDF that contains a function `show_image`; this function takes in an image matrix and displays it to the screen, so you can view the image without leaving Thonny if you like. (You will need to install the `matplotlib` library before calling this function.)

Note however that you will not be able to open compressed PGM files in these programs.

If your program writes a (regular) PGM file to disk which you are then unable to open in Preview or Irfanview, it means that the file was not in the proper format. To check what went wrong, you can open the PGM file in a text-editing program like TextEdit or Notepad and manually inspect the contents to see what the problem is.

For this question, write the following functions in a file `image_processing.py`.

- `is_valid_image`: takes a nested list as input, and returns `True` if the nested list represents a valid (non-compressed) PGM image matrix and `False` otherwise. A PGM image matrix is simply a nested list that contains only integers between 0 and 255, and where each row of the matrix has the same length.

```
>>> is_valid_image([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
True

>>> is_valid_image([[0], [0, 0]])
False

>>> is_valid_image([["0x5", "200x2"], ["111x7"]])
False
```

- `is_valid_compressed_image`: takes a nested list as input, and returns `True` if the nested list represents a valid compressed PGM image matrix and `False` otherwise. A compressed PGM image matrix is one that contains only strings of the form 'AxB', where A is an integer between 0 and 255 and B is a positive integer. Further, the sum of all B values in each row must be the same (e.g., if one row had the values '0x5' and '200x2', then the sum of the B values would be 7, and all other rows would also need a sum of 7 for the matrix to be valid).

```
>>> is_valid_compressed_image([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
False

>>> is_valid_compressed_image([[0], [0, 0]])
False

>>> is_valid_compressed_image([["0x5", "200x2"], ["111x7"]])
True
```

- `load_regular_image`: takes a filename (string) of a PGM image file as input, and loads in the image contained in the file and returns it as an image matrix. If, during or after loading, the resulting image matrix is found to not be in PGM format, then a `AssertionError` with an appropriate error message should be raised instead. (The file could have anything inside it, not necessarily a valid image matrix.)

  Note: For this and further examples, we put each sublist on its own line for readability purposes. When you write your examples in the docstring, however, the output must be all on one line for the doctest to work properly.

```
>>> load_regular_image("comp.pgm")
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 51, 51, 51, 51, 51, 0, 119, 119, 119, 119, 119, 0, 187, 187, 187, 187, 187, 0, 255,
  255, 255, 255, 0],
 [0, 51, 0, 0, 0, 0, 0, 119, 0, 0, 0, 119, 0, 187, 0, 187, 0, 187, 0, 255, 0, 0, 255, 0],
 [0, 51, 0, 0, 0, 0, 0, 119, 0, 0, 0, 119, 0, 187, 0, 187, 0, 187, 0, 255, 255, 255, 255,
  0],
 [0, 51, 0, 0, 0, 0, 0, 119, 0, 0, 0, 119, 0, 187, 0, 187, 0, 187, 0, 255, 0, 0, 0, 0],
 [0, 51, 51, 51, 51, 51, 0, 119, 119, 119, 119, 119, 0, 187, 0, 187, 0, 187, 0, 255, 0,
  0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

- `load_compressed_image`: takes a filename (string) of a compressed PGM image file as input, and loads in the image contained in the file and returns it as a compressed image matrix. If, during or after loading, the resulting image matrix is found to not be in compressed PGM format, then a `AssertionError` with an appropriate error message should be raised instead. (The file could have anything inside it, not necessarily a valid image matrix.)

```
>>> load_compressed_image("comp.pgm.compressed")
[['0x24'],
 ['0x1', '51x5', '0x1', '119x5', '0x1', '187x5', '0x1', '255x4', '0x1'],
 ['0x1', '51x1', '0x5', '119x1', '0x3', '119x1', '0x1', '187x1', '0x1', '187x1', '0x1',
   '187x1', '0x1', '255x1', '0x2', '255x1', '0x1'],
 ['0x1', '51x1', '0x5', '119x1', '0x3', '119x1', '0x1', '187x1', '0x1', '187x1', '0x1',
   '187x1', '0x1', '255x4', '0x1'],
 ['0x1', '51x1', '0x5', '119x1', '0x3', '119x1', '0x1', '187x1', '0x1', '187x1', '0x1',
   '187x1', '0x1', '255x1', '0x4'],
 ['0x1', '51x5', '0x1', '119x5', '0x1', '187x1', '0x1', '187x1', '0x1', '187x1', '0x1',
   '255x1', '0x4'],
```

```
 ['0x24']]

>>> fobj = open("invalid_test.pgm", "w")
>>> fobj.write("P2C\\n30 5\\n255\\nabc3x23 0x0x7\\n")
27
>>> fobj.close()
>>> load_compressed_image("invalid_test.pgm")
Traceback (most recent call last):
AssertionError: More than one x in a pixel.
```

- `load_image`: takes a filename (string) of a file as input. Checks the first line of the file. If it is `'P2'`, then loads in the file as a regular PGM image and returns the image matrix. If it is `'P2C'`, then loads in the file as a compressed PGM image and returns the compressed image matrix. If it is anything else, then a `AssertionError` with an appropriate error message should be raised instead. (The file could have anything inside it, not necessarily a valid image matrix.)

  Hint: Don't repeat code when you don't need to. Instead, consider how to call your previously defined functions.

```
>>> load_image("comp.pgm.compressed")
[['0x24'],
 ['0x1', '51x5', '0x1', '119x5', '0x1', '187x5', '0x1', '255x4', '0x1'],
 ['0x1', '51x1', '0x5', '119x1', '0x3', '119x1', '0x1', '187x1', '0x1', '187x1', '0x1',
   '187x1', '0x1', '255x1', '0x2', '255x1', '0x1'],
 ['0x1', '51x1', '0x5', '119x1', '0x3', '119x1', '0x1', '187x1', '0x1', '187x1', '0x1',
   '187x1', '0x1', '255x4', '0x1'],
 ['0x1', '51x1', '0x5', '119x1', '0x3', '119x1', '0x1', '187x1', '0x1', '187x1', '0x1',
   '187x1', '0x1', '255x1', '0x4'],
 ['0x1', '51x5', '0x1', '119x5', '0x1', '187x1', '0x1', '187x1', '0x1', '187x1', '0x1',
   '255x1', '0x4'],
 ['0x24']]
```

- `save_regular_image`: takes a nested list and a filename (string) as input, and saves it in the PGM format to a file with the given filename. If the image matrix given as input is not a valid PGM image matrix, instead raise a `AssertionError` with an appropriate error message.

  Note: When you run these examples, you will only see a single backslash character. We include two below because examples in the docstring that contain a backslash must have two backslashes, otherwise the doctest will not work for those examples.

```
>>> save_regular_image([[0]*10, [255]*10, [0]*10], "test.pgm")
>>> fobj = open("test.pgm", 'r')
>>> fobj.read()
'P2\\n10 3\\n255\\n0 0 0 0 0 0 0 0 0 0\\n255 255 255 255 255 255 255 255 255 255\\n0 0 0
0 0 0 0 0 0 0\\n'
>>> fobj.close()

>>> image = [[0]*10, [255]*10, [0]*10]
>>> save_regular_image(image, "test.pgm")
>>> image2 = load_image("test.pgm")
>>> image == image2
True
```

- `save_compressed_image`: takes a nested list and a filename (string) as input, and saves it in the compressed PGM format to a file with the given filename. If the image matrix given as input is not a valid compressed PGM image matrix, instead raise a `AssertionError` with an appropriate error message.

```
>>> save_compressed_image([["0x5", "200x2"], ["111x7"]], "test.pgm.compressed")
>>> fobj = open("test.pgm.compressed", 'r')
>>> fobj.read()
'P2C\\n7 2\\n255\\n0x5 200x2\\n111x7\\n'
>>> fobj.close()

>>> image = [["0x5", "200x2"], ["111x7"]]
>>> save_compressed_image(image, "test.pgm")
>>> image2 = load_compressed_image("test.pgm")
>>> image == image2
True
```

- `save_image`: takes a nested list and a filename (string) as input. Checks the type of elements in the list. If they are integers, then saves the nested list as a PGM image matrix into a file with the given filename. If they are strings, then saves the nested list as a compressed PGM image matrix into a file with the given filename. If they are anything else, then a `AssertionError` with an appropriate error message should be raised instead.

```
>>> save_image([["0x5", "200x2"], ["111x7"]], "test.pgm.compressed")
>>> fobj = open("test.pgm.compressed", 'r')
>>> fobj.read()
'P2C\\n7 2\\n255\\n0x5 200x2\\n111x7\\n'
>>> fobj.close()
```

- `invert`: takes a (non-compressed) PGM image matrix as input, and returns the inverted image. Should not modify the input matrix. If the input matrix is not a valid PGM image matrix, then a `AssertionError` with an appropriate error message should be raised instead.

  Inverting an image matrix means to take the complement of each integer in the matrix, i.e., subtract each number from 255, so that black becomes white and vice-versa.

```
>>> image = [[0, 100, 150], [200, 200, 200], [255, 255, 255]]
>>> invert(image)
[[255, 155, 105], [55, 55, 55], [0, 0, 0]]

>>> image == [[0, 100, 150], [200, 200, 200], [255, 255, 255]]
True
```

- `flip_horizontal`: takes a (non-compressed) PGM image matrix as input, and returns the image matrix flipped horizontally. Should not modify the input matrix. If the input matrix is not a valid PGM image matrix, then a `AssertionError` with an appropriate error message should be raised instead.

  Flipping an image matrix horizontally means to reverse the elements of each row.

```
>>> image = [[1, 2, 3, 4, 5], [0, 0, 5, 10, 10], [5, 5, 5, 5, 5]]
>>> flip_horizontal(image)
[[5, 4, 3, 2, 1], [10, 10, 5, 0, 0], [5, 5, 5, 5, 5]]
```

- `flip_vertical`: takes a (non-compressed) PGM image matrix as input, and returns the image matrix flipped vertically. Should not modify the input matrix. If the input matrix is not a valid PGM image matrix, then a `AssertionError` with an appropriate error message should be raised instead.

  Flipping an image matrix horizontally means to reverse the elements of each column.

```
>>> image = [[1, 2, 3, 4, 5], [0, 0, 5, 10, 10], [5, 5, 5, 5, 5]]
>>> flip_vertical(image)
[[5, 5, 5, 5, 5], [0, 0, 5, 10, 10], [1, 2, 3, 4, 5]]
```

- `crop`: takes a (non-compressed) PGM image matrix, two non-negative integers and two positive integers as input. The two non-negative integers correspond to the top left row and top left column of the target rectangle, and the two positive integers correspond to the number of rows and number of columns of the target rectangle. The function should return an image matrix corresponding to the pixels contained in the target rectangle. Should not modify the input list. If the input matrix is not a valid PGM image matrix, then a `AssertionError` with an appropriate error message should be raised instead.

```
>>> crop([[5, 5, 5], [5, 6, 6], [6, 6, 7]], 1, 1, 2, 2)
[[6, 6], [6, 7]]

>>> crop([[1, 2, 3, 4], [4, 5, 6, 7], [8, 9, 10, 11]], 1, 2, 2, 1)
[[6], [10]]
```

- `find_end_of_repetition`: takes a list of integers and two non-negative integers (an index and a target number) as input. You can assume that the list will contain the target number at the given index. Looks through the list starting after the given index, and returns the index of the last consecutive occurrence of the target number. That is, as soon as you come across a number that is not the target, then return the previous index.

```
>>> find_end_of_repetition([5, 3, 5, 5, 5, -1, 0], 2, 5)
4

>>> find_end_of_repetition([1, 2, 3, 4, 5, 6, 7], 6, 7)
6
```

- `compress`: takes a (non-compressed) PGM image matrix as input. Compresses the matrix according to the compression algorithm described earlier in this PDF and returns the compressed matrix. If the input matrix is not a valid PGM image matrix, then a `AssertionError` with an appropriate error message should be raised instead.

  Hint: Think about how you could use the previous function to help here.

```
>>> compress([[11, 11, 11, 11, 11], [1, 5, 5, 5, 7], [255, 255, 255, 0, 255]])
[['11x5'], ['1x1', '5x3', '7x1'], ['255x3', '0x1', '255x1']]
```

- `decompress`: takes a compressed PGM image matrix as input. Decompresses the matrix by undoing the compression algorithm described earlier in this PDF and returns the decompressed matrix. If the input matrix is not a valid compressed PGM image matrix, then a `AssertionError` with an appropriate error message should be raised instead.

```
>>> decompress([['11x5'], ['1x1', '5x3', '7x1'], ['255x3', '0x1', '255x1']])
[[11, 11, 11, 11, 11], [1, 5, 5, 5, 7], [255, 255, 255, 0, 255]]

>>> image = [[11, 11, 11, 11, 11], [1, 5, 5, 5, 7], [255, 255, 255, 0, 255]]
>>> compressed_image = compress(image)
>>> image2 = decompress(compressed_image)
>>> image == image2
True
```

- `process_command`: takes a string as input corresponding to a series of space-separated image processing commands, and executes each command in turn. Does not return anything.

  A command string can contain any of the following commands, which each correspond to one of the above functions: `'LOAD'`, `'SAVE'`, `'INV'`, `'FH'`, `'FV'`, `'CR'`, `'CP'`, `'DC'`.

  Some commands will have an extra part after them to indicate further needed information. For example, a LOAD command will always have the form `'LOAD<x.pgm>'` where 'x.pgm' will be a filename, and same for the SAVE command. A crop command will have the form `'CR<y,x,h,w>'` where y and x are two non-negative integers and h and w are two positive integers, and y, x, h and w correspond to the four integer inputs needed for the crop command.

  You can assume that the command string will always begin with a LOAD command and end with a SAVE command.

  If an unrecognized command is part of the command string, a `AssertionError` should be raised with an appropriate error message.

```
>>> process_command("LOAD<comp.pgm> CP DC INV INV SAVE<comp2.pgm>")
>>> image = load_image("comp.pgm")
>>> image2 = load_image("comp2.pgm")
>>> image == image2
True

>>> process_command("LOAD<comp.pgm> CP SAVE<comp.pgm.compressed>")
>>> load_compressed_image("comp.pgm.compressed")
[['0x24'],
 ['0x1', '51x5', '0x1', '119x5', '0x1', '187x5', '0x1', '255x4', '0x1'],
 ['0x1', '51x1', '0x5', '119x1', '0x3', '119x1', '0x1', '187x1', '0x1', '187x1', '0x1',
  '187x1', '0x1', '255x1', '0x2', '255x1', '0x1'],
 ['0x1', '51x1', '0x5', '119x1', '0x3', '119x1', '0x1', '187x1', '0x1', '187x1', '0x1',
  '187x1', '0x1', '255x4', '0x1'],
 ['0x1', '51x1', '0x5', '119x1', '0x3', '119x1', '0x1', '187x1', '0x1', '187x1', '0x1',
  '187x1', '0x1', '255x1', '0x4'],
 ['0x1', '51x5', '0x1', '119x5', '0x1', '187x1', '0x1', '187x1', '0x1', '187x1', '0x1',
  '255x1', '0x4'],
 ['0x24']]
```

What To Submit

You must submit all your files on codePost (https://codepost.io/). The files you should submit are listed below. Any deviation from these requirements may lead to lost marks.

`image_processing.py`

`README.txt` In this file, you can tell the TA about any issues you ran into while doing this assignment.
Remember that this assignment like all others is an `individual` assignment and must represent the entirety of your own work. You are permitted to verbally discuss it with your peers, as long as no written notes are taken. If you do discuss it with anyone, please make note of those people in this `README.txt` file. If you didn't talk to anybody nor have anything you want to tell the TA, just say "nothing to report" in the file.

You may make as many submissions as you like, but we will only grade your final submission (all prior ones are automatically deleted).

Note: If you are having trouble, make sure the names of your files are exactly as written above.

Assignment debriefing

In the week(s) following the due date for this assignment, you will be asked to meet with a TA for a 20-25 minute meeting (exact duration TBD). In this meeting, the TA will discuss with you what you should improve for future assignments.

Only your code will determine your grade. You will not be able to provide any clarifications or extra information in order to improve your grade. However, you will have the opportunity to ask for clarifications regarding your grade.

You may also be asked during the meeting to explain portions of your code. Answers to these questions will not be used to determine your grade, but inability to explain your code may be used as evidence to support a charge of plagiarism later in the term.

Details on how to schedule a meeting with the TA will be shared with you in the days following the due date of the assignment.

If you do not attend a meeting, you will receive a 0 for your assignment.