Homework 1: Lists

CS 201 Data Structures 2
Habib University

Spring 2023

This write-up has 3 parts. The first part provides an explanation of your implementation tasks. It refers to code that is provided in the appendices in the third part. The second part lists the tasks for you to do. It contains space for you to enter your solutions to some of the problems. The third part lists code which is referred to in the first part.

You have to fill in the solutions in the second part, and complete the code files in the accompanying `src/` folder as described in Part 1. Please work in this file, `hw1.tex`, and not in a copy. When submitting, please remove the first and third parts from this file.

Part I
Explanation

In this assignment, we will implement an *ArrayList* to represent a *List*. We will use the *List* to implement an image and will write operations for the image.

1 Image Operations

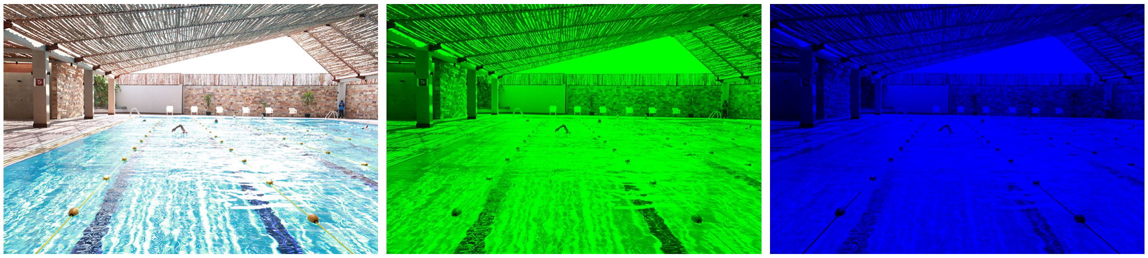
We will work with RGB images and perform four operations on them–channel suppression, rotations, mask application, and resize. None of these operations is *destructive*. That is, the operations do not alter the original image, rather they return a new image containing the result of the operation.

1.1 Channel Suppression

An image is said to contain color values in different *channels*. In an RGB image, the channels are Red, Blue, and Green. Each channel contains the intensities for that color for every pixel in the image. The values from all three channels at a pixel yield the RGB value at the pixel. A channel suppression operation switches off a specific channel. That is, all intensities in that channel are turned to zero, or turned off. Figure 1 shows an original image and two modifications, one with the blue channel turned off, and the other with only the blue channel turned on, i.e. the red and green channels turned off.

1.2 Rotation

Given a square image, i.e. one whose width is equal to its height, this operation generates a new image that contains rotations of the original image. Figure 2 shows an example of applying the operation. The resulting image has twice the dimensions of the original image, i.e. twice the width and twice the height. It contains 4 appropriately placed sub-images which, going anti-clockwise are the original image rotated anti-clockwise by increments of 90°.



(a) An RGB image of a swimming pool. (b) The image with the blue channel turned off. (c) The original image with only the blue channel turned on.

Figure 1: Example of channel suppression.

(a) A square image.

(b) The image obtained as a result of applying rotations to the original image.

Figure 2: Example of rotation.

1.3 Applying a Mask

A *mask* specifies certain *weights* and applying the mask to an image entails replacing the value at each pixel in the image with a *weighted sum* or *weighted average* of the values of its *neighbors*. The weights and the neighbors to consider for the average are specified by the mask.

A mask is an $n \times n$ array of integers representing weights. For our purposes, n must be odd. This means that the $n \times n$ array has a well defined center–the *origin*. The weights in the mask can be arbitrary integers–positive, negative, or zero.

For each pixel in the input image, think of the mask as being placed on top of the image so its origin is on the pixel we wish to examine. The intensity value of each pixel under the mask is multiplied by the corresponding value in the mask that covers it. These products are added together. Always use the original values for each pixel for each mask calculation, not the new values that you compute as you process the image.

For example, refer to Figure 3a, which shows the 3×3 mask,

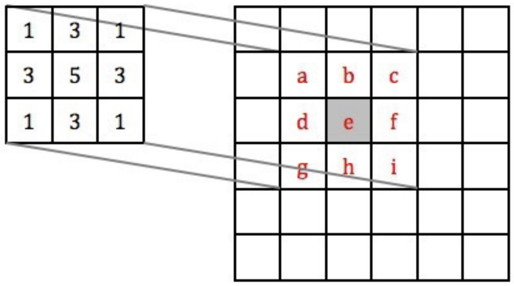
$$\begin{bmatrix} 1 & 3 & 1 \\ 3 & 5 & 3 \\ 1 & 3 & 1 \end{bmatrix}$$

and an image on which we want to perform the mask computation. Suppose we want to compute the result of the mask computation for pixel e. This result would be:

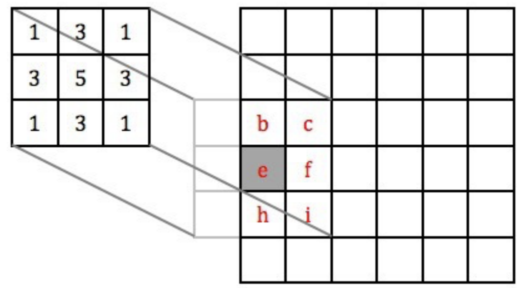
$$a + 3b + c + 3d + 5e + 3f + g + 3h + i$$

Some masks require a *weighted average* instead of a weighted sum. The weighted average in the case of Figure 3a for pixel e would be:

$$\frac{a + 3b + c + 3d + 5e + 3f + g + 3h + i}{1 + 3 + 1 + 3 + 5 + 3 + 1 + 3 + 1}$$



(a) Overlay the 3×3 mask over the image so it is centered on pixel e to compute the new value for pixel e.



(b) If the mask hangs over the edge of the image, use only those mask values that cover the image in the weighted sum.

Figure 3: Applying a mask to an image.

Instead of doing this calculation for each channel individually at a pixel, for the purpose of this calculation, replace the value of each channel at the pixel with the average channel value at the pixel. For example, if the pixel is given by $(r, g, b) = (107, 9, 218)$, then apply the mask to the average value $(107 + 9 + 218)//3 = 111$ (integer division) and copy the result to each channel of the corresponding pixel in the output image. This effectively converts the output image to grayscale.

Note that sometimes when you center the mask over a pixel, the mask will hang over the edge of the image. In this case, compute the weighted sum of only those pixels that the mask covers. For the example shown in Figure 3b, the weighted sum for the pixel e is given by:

$$3b + c + 5e + 3f + 3h + i$$

and the weighted average is as follows.

$$\frac{3b + c + 5e + 3f + 3h + i}{3 + 1 + 5 + 3 + 3 + 1}$$

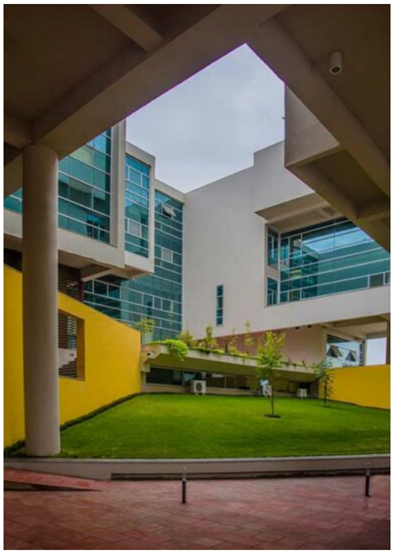
Integer division is used when computing averages in order to ensure that pixel intensities are integers.

1.3.1 Applications of Masks

Applying different masks leads to different properties. For example, applying the following mask leads to blurring of the image. Figures 4a and 4b show the blurring effect of this mask. Note that color information is lost as mentioned above.

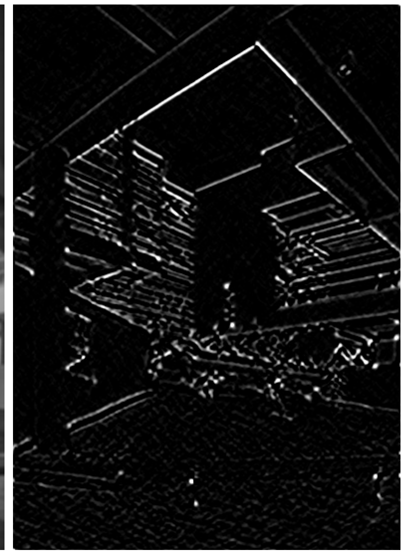
$$\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}.$$

Another application we use is an implementation of *Canny Edge Detection* using *Sobel Operators*. Once the image has been blurred as above, two more *filters*, or masks, (the Sobel operators) are applied in succession to the blurred image. These filters determine the change in intensity, which approximates the



(a) An image with sharp details and several lines.

(b) Result of applying the blur mask to the original image.



(c) Result of applying the Sobel filters to the blurred image.

Figure 4: Blurring and detection of edges in an image using masks.

horizontal and vertical derivatives.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

After these operations are applied one after the other to the blurred image, the values obtained are used to search for edges based on the magnitude and direction of the change in intensity. An example of the final result is shown in Figure 4c.

1.4 Resize

Given an image, this operation generates a new image that has twice the dimensions of the original image i.e, twice the width and twice the height. Figure 5 shows a 4x3 image which is resized to twice its size. The resized image has 4 times as many pixels and some of them take on the values from the original image as shown. For the pixels shown to be blank, color values are not known and have to be computed from the known values. Consider the labeled pixels in the resized image below. One way to fill in the missing color information is as follows.

$$P = \frac{1}{2}(A+B),\ T = \frac{1}{2}(C+D),\ Q = \frac{1}{2}(A+C),\ S = \frac{1}{2}(B+D)$$

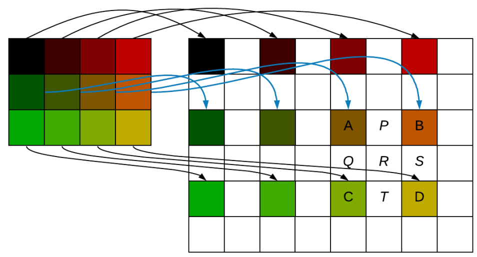


Figure 5: Bilinear Interpolation

There are various ways to compute R, all of which are ultimately equivalent.

$$R = \frac{1}{2}(P+T) = \frac{1}{2}(S+Q) = \frac{1}{4}(A+B+C+D) = \frac{1}{4}(P+Q+S+T) = \frac{1}{8}(A+B+C+D+P+Q+S+T)$$

The boundary pixels pose a problem as some of the neighboring pixels required for the average do not exist. In such cases, only the existing neighbors are used for the average.

Notice how all the above expressions are affine combinations. Furthermore, the colors for P, Q, S, and T are *linearly interpolated* from their horizontal or vertical neighbors. The color for R is a *bilinear interpolation*: it is a linear interpolation of P and T, or Q and S, which are themselves linear interpolations.

Note: All divisions in the above expressions are integer divisions.

2 Image

We treat an image as a grid of *pixels* where each pixel is represented as an RGB value indicating the red, green, and blue intensities of the pixel. An image has *dimensions*, namely *width* and *height*, which determine the number of *rows* and *columns* in the image. Every pixel in the image is at a unique combination of row and column numbers which can therefore be used as a coordinate system in the image. An image with width w and height h is said to be of size $w \times h$. Figure 6a shows the column and row numbers in a $w \times h$ image along with the resulting pixel coordinates. Note that the coordinate is just a means to locate a pixel in the image, it is not the value stored at the pixel. The value stored at a pixel is a triplet denoting the red, green, and blue intensities respectively.

We will work with a *flattened* representation of an image. That is, we will store the pixel values in a 1-dimensional list structure as opposed to a 2-dimensional structure (programming languages generally store multi-dimensional arrays in their flattened form) . The list stores pixel values as they appear in the image from left to right and top to bottom. Figure 6b shows a 5×5 image with some supposed RGB values. Note that each value would be a triplet of integers, each integer between 0 and 255 inclusive. Using our representation, the image in Figure 6b will be represented as the list:

$$[a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y]$$

Rows	Columns: 0	1	2	…	$(w-1)$
0	$(0,0)$	$(0,1)$	$(0,2)$	…	$(0,w-1)$
1	$(1,0)$	$(1,1)$	$(1,2)$	…	$(1,w-1)$
2	$(2,0)$	$(2,1)$	$(2,2)$	…	$(2,w-1)$
⋮	⋮	⋮	⋮	⋱	⋮
$(h-1)$	$(h-1,0)$	$(h-1,1)$	$(h-1,2)$	…	$(h-1,w-1)$

(a) Row and column numbers of an image with width w and height h. Pixel coordinates are also shown.

a	b	c	d	e
f	g	h	i	j
k	l	m	n	o
p	q	r	s	t
u	v	w	x	y

(b) A 5×5 image with supposed pixel values.

Figure 6: Image dimensions and pixel coordinates.

3 Implementation Details and Tasks

We will be working with a `MyImage` class as shown in Listing 1 on Page 9, also included in the accompanying file `src/myimage.py`. Its implementation is complete but requires a concrete implementation of `MyList` which is our implementation of the *List* interface. The implementation to be used is specified in the constructor of `MyImage`.

An implementation of `MyList` is shown in Listing 2 on Page 12, also included in the accompanying file `src/mylist.py`. The implementation is mostly complete except for the segments marked as `pass`. These are to be implemented appropriately in the subclasses of `MyList` which are indicated at the end of the listing and whose implementation is completely missing. Writing their implementations is one of your tasks in this assignment.

Once you are done implementing `MyList` subclasses, the `MyImage` class is ready to be operated on. Functions corresponding to the operations described in Section 1 are shown in Listing 3 on Page 16 and included in the accompanying file `src/image_operations.py`. None of the operations is destructive. That is, each operates on a `MyImage` instance and returns the result as a new `MyImage` instance. The functions are missing implementations. Writing their implementations is another of your tasks in this assignment.

3.1 Tasks

- Go over the provided files thoroughly in order to understand what they do or are expected to do.
- Provide implementations for unimplemented methods, i.e. those that have `pass` in their body.
- Derive `ArrayList` class from `MyList` and provide its implementation in the same file. `ArrayList` implements the list using python arrays.

3.2 Requirement

You will need to install Pillow which will prove the `PIL` module used in the provided code.

3.3 Tips

Below are some tips to avoid the errors that have previously caused tests to fail. Following these may save you many frustrating hours of debugging!

- Delay division as much as possible and perform `int` division wherever needed.
- When writing gray values to file, make sure to clamp them to [0,255].
- Take care about imagine indexing.
- Be careful when creating a copy of the image. Use the copy where needed and the original where needed.

- Do not forget to average the RGB values when the corresponding flag in `apply_mask` is enabled.
- Take care about efficiency. Some structures are slow. If, on top, your code is inefficient, the automated tests may fail due to time out.

3.4 Testing

Once you have successfully implemented the subclasses and image operations, you can test your code by creating an image and performing operations on it. Your submission will be tested automatically by GitHub using the accompnaying `pytest` file, `test_image.py`.

4 Credits

This homework is adapted from Homework 3 of the Fall 2014 offering of 15-122: Principles of Imperative Computation at Carnegie Mellon University (CMU).

Part II

Problems

The grading is defined in the accompanying rubric file.

1. Implementation

Complete the tasks listed in Section 3.1 by providing the implementations in the indicated files.

2. Amortized Analysis

Consider an `ArrayStack` implementation of the `List` interface with a slightly altered `resize()` operation. Instead of reserving space for $2n$ elements in the new array, it reserves space for $n + \lceil \frac{n}{4} \rceil$ elements. Prove that the `append()` operation still takes $O(1)$ time in the amortized sense.

Solution:

Traditional `resize()` where the length of the array doubles when the array is full works by allocating an array b of size $2n$, and copies the n elements of the original array a into the new array b and then sets a to b. Therefore, a has length of $2n$ now. This takes $O(n)$ time where the `append()` operation takes $O(1)$ time.

If the new `resize()` operation reserves space for $n + [\frac{n}{4}]$ elements, then the `append()` operation would still take $O(1)$ time in the amortized sense.

This can be undertood better by first understanding the cost of `append()` for when `resize()` reserves space for $2n$ elements;

Resize of $2n$:

Consider an array of size 1. The cost of appending an element would be $O(1)$ $[n = 1]$.
Now for the second `append()`, resize would be called, and then the total cost of appending the second element would be $O(2)[n = 2]$.
For the third `append()`, `resize()` would again be called and the total cost would be $O(3)[n = 3]$.
For the fourth `append()`, `resize()` won't be called as the array is not full, and only the cost of `append()` is taken into account, therefore the total cost is $O(4)$, however, this takes into account traversal cost as well, and the cost of `append()` remains $O(1)$.

With the next `append()` operation[$n = 5$], a new array will be created with space reserved for 8 elements. Then the total cost of appending all elements into the new array is $O(5)$, however, the individual cost of `append()` remains $O(1)$.

Then using the above, we can make a series such that:
$Series = 1 + (1 + 1) + (1 + 2) + 1 + (1 + 4) + 1 + 1 + 1 + (1 + 8) + \cdots$
Therefore, the total cost becomes:
$Cost = 1 + 2 + 4 + 8 + \cdots$ [for `resize()`] $+1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \cdots$ [for `append()`]

The above series for resize can be written as a geometric series where `resize()` $= 2^{\log_2(n)-i} + 1$. Hence the total cost of `resize()` comes out to be $O(1)$ with each `append()` operation having a cost of $O(1)$.

Resize of $n + \frac{n}{4}$:

Then we can prove that the `append()` operation has a cost of $O(1)$ in the amortized sense.
The new size increase is of $n + \frac{n}{4} \implies \frac{5n}{4}$. So space is reserved in the new array for $\frac{5n}{4}$ elements(we will consider ceil value for decimal numbers).

Now consider an array of size 1. The cost of appending just one element would be $O(1)$ [n = 1].
For another `append()` operation [n = 2], the total cost would be considering the cost of resize, and the cost of append. The `resize()` function would give us $\frac{1}{4} = 0.25$. Ceil$(0.25) = 1$. Therefore, the total cost becomes $1 + \text{ceil}(\frac{n}{4}) = 1 + 1 = O(2)$.
For the third `append()` [n = 3] operation, the total cost would be considering the cost of resize, and the cost of append. The `resize()` function would give us $\frac{2}{4} = 0.5$. Ceil of this is equal to 1. Therefore the total cost would be $1 + 1 + 1 = O(3)$.
For the fourth `append()` operation, the same takes place.
For the fifth `append()` operation, `resize()` reserves space for 2 elements, as `resize()` gives us $\frac{5}{4} = 1.25$, and the ceil of this is 2. Therefore, space is reserved for two elements. Therefore, the following `append()` operation would cost $O(1)$.

Then using the above, we can make a series such that:
$Series = 1 + (1 + 1) + (1 + 2) + (1 + 3) + (1 + 4) + (1 + 5) + 1 + (1 + 7) + \cdots$
Therefore, the total cost becomes:
$Cost = 1 + 2 + 3 + 4 + 5 + 7 + \cdots$ [for `resize()`] $+1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + \cdots$ [for `append()`]

Then the series can be written as a geometric series where `resize()` $= n + \dfrac{5^{\log \frac{5n}{4} + 1}}{4} \implies O(n)$ as n gets larger.

Then the cost of appending one element is $\frac{O(n)}{n} = O(1)$. Hence proved that the `append()` operation still takes $O(1)$ time in the amortized sense.

Part III
Appendices

A MyImage

```python
from PIL import Image
from src.mylist import ArrayList


class MyImage:
    """Holds a flattened RGB image and its dimensions. Also implements Iterator
    methods to allow iteration over this image.
    """

    def __init__(self, size: (int, int)) -> None:
        """Initializes a black image of the given size.

        Parameters:
        - self: mandatory reference to this object
        - size: (width, height) specifies the dimensions to create.

        Returns:
        none
        """
        # Save size, create a list of the desired size with black pixels.
        width, height = self.size = size
        self.pixels: ArrayList = ArrayList(width * height,
                                           value=(0, 0, 0))

    def __iter__(self) -> 'MyImage':
        '''Iterator function to return an iterator (self) that allows iteration over
        this image.

        Parameters:
        - self: mandatory reference to this object

        Returns:
        an iterator (self) that allows iteration over this image.
        '''
        # Initialize iteration indexes.
        self._iter_r: int = 0
        self._iter_c: int = 0
        return self

    def __next__(self):
        ''''Iterator function to return the next value from this list.

        Image pixels are iterated over in a left-to-right, top-to-bottom order.

        Parameters:
        - self: mandatory reference to this object

        Returns:
        the next value in this image since the last iteration.

        '''
        if self._iter_r < self.size[1]:  # Iteration within image bounds.
            # Save current value as per iteration variables. Update the
            # variables for the next iteration as per the iteration
            # order. Return saved value.
            value = self.get(self._iter_r, self._iter_c)
            self._iter_c += 1
            if self._iter_c == self.size[0]:
```

```
                self._iter_c = 0
                self._iter_r += 1
            return value
        else:  # Image bounds exceeded, end of iteration.
            # Reset iteration variables, end iteration.
            self._iter_r = self._iter_c = 0
            raise StopIteration

    def _get_index(self, r: int, c: int) -> int:
        """Returns the list index for the given row, column coordinates.

        This is an internal function for use in class methods only. It should
        not be used or called from outside the class.

        Parameters:
        - self: mandatory reference to this object
        - r: the row coordinate
        - c: the column coordinate

        Returns:
        the list index corresponding to the given row and column coordinates
        """
        # Confirm bounds, compute and return list index.
        width, height = self.size
        assert 0 <= r < height and 0 <= c < width, "Bad image coordinates: "\
            f"(r, c): ({r}, {c}) for image of size: {self.size}"
        return r*width + c

    def open(path: str) -> 'MyImage':
        """Creates and returns an image containing from the information at file path.

        The image format is inferred from the file name. The read image is
        converted to RGB as our type only stores RGB.

        Parameters:
        - path: path to the file containing image information

        Returns:
        the image created using the information from file path.
        """
        # Use PIL to read the image information and store it in our instance.
        img: Image = Image.open(path)
        myimg: MyImage = MyImage(img.size)
        # Covert image to RGB. https://stackoverflow.com/a/11064935/1382487
        img: Image = img.convert('RGB')
        # Get list of pixel values (https://stackoverflow.com/a/1109747/1382487),
        # copy to our instance and return it.
        for i, rgb in enumerate(list(img.getdata())):
            myimg.pixels.set(i, rgb)
        return myimg

    def save(self, path: str) -> None:
        """Saves the image to the given file path.

        The image format is inferred from the file name.

        Parameters:
        - self: mandatory reference to this object
        - path: the image has to be saved here.

        Returns:
        none
        """
        # Use PIL to write the image.
        img: Image = Image.new("RGB", self.size)
        img.putdata([rgb for rgb in self.pixels])
```

```python
        img.save(path)

    def get(self, r: int, c: int) -> (int, int, int):
        """Returns the value of the pixel at the given row and column coordinates.

        Parameters:
        - self: mandatory reference to this object
        - r: the row coordinate
        - c: the column coordinate

        Returns:
        the stored RGB value of the pixel at the given row and column coordinates.
        """
        return self.pixels[self._get_index(r, c)]

    def set(self, r: int, c: int, rgb: (int, int, int)) -> None:
        """Write the rgb value at the pixel at the given row and column coordinates.

        Parameters:
        - self: mandatory reference to this object
        - r: the row coordinate
        - c: the column coordinate
        - rgb: the rgb value to write

        Returns:
        none
        """
        self.pixels[self._get_index(r, c)] = rgb

    def show(self) -> None:
        """Display the image in a GUI window.

        Parameters:

        Returns:
        none
        """
        # Use PIL to display the image.
        img: Image = Image.new("RGB", self.size)
        img.putdata([rgb for rgb in self.pixels])
        img.show()
```

Code Listing 1: Image Type

B MyList

```python
import array as arr

class MyList:
    '''A list interface. Also implements Iterator functions in order to support
    iteration over this list.
    '''

    def __init__(self, size: int, value=None) -> None:
        """Creates a list of the given size, optionally intializing elements to value.

        The list is static. It only has space for size elements.

        Parameters:
        - self: mandatory reference to this object
        - size: size of the list; space is reserved for these many elements.
        - value: the optional initial value of the created elements.

        Returns:
        none
        """
        self.lst_arr = [value] * size

    def __len__(self) -> int:
        '''Returns the size of the list. Allows len() to be called on it.

        Ref: https://stackoverflow.com/q/7642434/1382487

        Parameters:
        - self: mandatory reference to this object

        Returns:
        the size of the list.
        '''
        return len(self.lst_arr)

    def __getitem__(self, i: int):
        '''Returns the value at index, i. Allows indexing syntax.

        Ref: https://stackoverflow.com/a/33882066/1382487

        Parameters:
        - self: mandatory reference to this object
        - i: the index from which to retrieve the value.

        Returns:
        the value at index i.
        '''
        # Ensure bounds.
        assert 0 <= i < len(self),\
            f'Getting invalid list index {i} from list of size {len(self)}'
        return self.lst_arr[i]

    def __setitem__(self, i: int, value) -> None:
        '''Sets the element at index, i, to value. Allows indexing syntax.

        Ref: https://stackoverflow.com/a/33882066/1382487

        Parameters:
        - self: mandatory reference to this object
        - i: the index of the elemnent to be set
        - value: the value to be set

        Returns:
        none
```

```python
        '''
        # Ensure bounds.
        assert 0 <= i < len(self),\
            f'Setting invalid list index {i} in list of size {len(self)}'
        self.lst_arr[i] = value

    def __iter__(self) -> 'MyList':
        '''Iterator function to return an iterator (self) that allows iteration over
        this list.

        Parameters:
        - self: mandatory reference to this object

        Returns:
        an iterator (self) that allows iteration over this list.
        '''
        # Initialize iteration index.
        self._iter_index: int = 0
        return self

    def __next__(self):
        ''''Iterator function to return the next value from this list.

        Parameters:
        - self: mandatory reference to this object

        Returns:
        the next value in this list since the last iteration.
        '''
        if self._iter_index < len(self):
            value = self.get(self._iter_index)
            self._iter_index += 1
            return value
        else:
            # End of Iteration
            self._index = 0
            raise StopIteration

    def get(self, i: int):
        '''Returns the value at index, i.

        Alternate to use of indexing syntax.

        Parameters:
        - self: mandatory reference to this object
        - i: the index from which to retrieve the value.

        Returns:
        the value at index i.
        '''
        return self[i]

    def set(self, i: int, value) -> None:
        '''Sets the element at index, i, to value.

        Alternate to use of indexing syntax.

        Parameters:
        - self: mandatory reference to this object
        - i: the index of the elemnent to be set
        - value: the value to be set

        Returns:
        none
        '''
        self[i] = value
```

```python


class ArrayList(MyList):
    '''A list interface. Also implements Iterator functions in order to support
    iteration over this list.
    '''

    def __init__(self, size: int, value=None) -> None:
        """Creates a list of the given size, optionally intializing elements to value.

        The list is static. It only has space for size elements.

        Parameters:
        - self: mandatory reference to this object
        - size: size of the list; space is reserved for these many elements.
        - value: the optional initial value of the created elements.

        Returns:
        none
        """
        self.arr_red = arr.array('i', [value[0] for i in range(size)])
        self.arr_green = arr.array('i', [value[1] for i in range(size)])
        self.arr_blue = arr.array('i', [value[2] for i in range(size)])

    def __len__(self) -> int:
        '''Returns the size of the list. Allows len() to be called on it.

        Ref: https://stackoverflow.com/q/7642434/1382487

        Parameters:
        - self: mandatory reference to this object

        Returns:
        the size of the list.
        '''
        return len(self.arr_blue)

    def __getitem__(self, i: int):
        '''Returns the value at index, i. Allows indexing syntax.

        Ref: https://stackoverflow.com/a/33882066/1382487

        Parameters:
        - self: mandatory reference to this object
        - i: the index from which to retrieve the value.

        Returns:
        the value at index i.
        '''
        # Ensure bounds.
        assert 0 <= i < len(self),\
            f'Getting invalid list index {i} from list of size {len(self)}'
        return ((self.arr_red[i], self.arr_green[i], self.arr_blue[i]))

    def __setitem__(self, i: int, value) -> None:
        '''Sets the element at index, i, to value. Allows indexing syntax.

        Ref: https://stackoverflow.com/a/33882066/1382487

        Parameters:
        - self: mandatory reference to this object
        - i: the index of the elemnent to be set
        - value: the value to be set

        Returns:
        none
```

```
        '''
        # Ensure bounds.
        assert 0 <= i < len(self),\
            f'Setting invalid list index {i} in list of size {len(self)}'
        self.arr_red[i] = value[0]; self.arr_green[i] = value[1]; self.arr_blue[i] = value[2
                                                    ]
```

Code Listing 2: List Type

C Image Operations

```python
from src.myimage import MyImage

def remove_channel(src: MyImage, red: bool = False, green: bool = False,
                   blue: bool = False) -> MyImage:
    """Returns a copy of src in which the indicated channels are suppressed.

    Suppresses the red channel if no channel is indicated. src is not modified.

    Args:
    - src: the image whose copy the indicated channels have to be suppressed.
    - red: suppress the red channel if this is True.
    - green: suppress the green channel if this is True.
    - blue: suppress the blue channel if this is True.

    Returns:
    a copy of src with the indicated channels suppressed.
    """
    src_copy = MyImage(src.size)
    rows = src.size[1]
    cols = src.size[0]
    for row in range(rows):
        for col in range(cols):
            src_copy.set(row, col, src.get(row, col))

    if red == False and green == False and blue == False:
        for row in range(rows):
            for col in range(cols):
                temp = list(src_copy.get(row, col)); temp[0] = 0
                src_copy.set(row, col, tuple(temp))
    else:
        for row in range(rows):
            for col in range(cols):
                temp = list(src_copy.get(row, col))
                if red == True: temp[0] = 0
                if green == True: temp[1] = 0
                if blue == True: temp[2] = 0
                src_copy.set(row, col, tuple(temp))
    return src_copy

def convert_to_matrix(lst, m, n): #converts a flattened representation into an mxn matrix
                                   such that m and n are known
    mat = []
    for i in range(0, len(lst), n):
        mat.append(lst[i:i+n])
    return mat[:m]

def rotate_90(matrix): #Rotates a given m x n matrix 90 degress clockwise
    res = []
    for i in range(len(matrix[0])):
        lst = []
        for j in range(len(matrix)):
            lst.append(matrix[j][i])
        # Reversing the matrix for 90 degree
        lst.reverse()
        res.append(lst)
    return res

def rotations(src: MyImage) -> MyImage:
    """Returns an image containing the 4 rotations of src.

    The new image has twice the dimensions of src. src is not modified.

    Args:
    - src: the image whose rotations have to be stored and returned.
```

```

    Returns:
    an image twice the size of src and containing the 4 rotations of src.
    """
    rows = src.size[1]
    cols = src.size[0]
    src_copy = MyImage((cols * 2, rows * 2))
    rot_90_lst = []
    rot_180_lst = []
    rot_270_lst = []
    rot_360_lst = []
    temp = []

    for row in range(rows):
        for col in range(cols):
            temp.append(src.get(row, col))
    rot_360_lst = convert_to_matrix(temp, rows, cols) #Og image
    # 90 degree clockwise rotation
    rot_90_lst = rotate_90(rot_360_lst)
    #Image - 180
    rot_180_lst = rotate_90(rot_90_lst)
    #Image - 90 anticlockwise
    rot_270_lst = rotate_90(rot_180_lst)
    src_copy_lst = []
    for i in range(rows * 2):
        if i < rows:
            for x in range(len(rot_270_lst[i])):
                src_copy_lst.append(rot_270_lst[i][x])
            for x in range(len(rot_360_lst[i])):
                src_copy_lst.append(rot_360_lst[i][x])
        else:
            for x in range(len(rot_180_lst[i % rows])):
                src_copy_lst.append(rot_180_lst[i % rows][x])
            for x in range(len(rot_90_lst[i % rows])):
                src_copy_lst.append(rot_90_lst[i % rows][x])
    lst1 = convert_to_matrix(src_copy_lst, rows*2, cols*2)
    for row in range(rows*2):
        for col in range(cols*2):
            src_copy.set(row, col, lst1[row][col])
    return src_copy

def resize(src: MyImage) -> MyImage:
    """Returns an image which has twice the dimensions of src.

    The new image has twice the dimensions of src. src is not modified.

    Args:
    - src: the image which needs to be resized.

    Returns:
    an image twice the size of src.
    """
    rows = src.size[1]
    cols = src.size[0]
    src_copy = MyImage((cols * 2, rows * 2))
    for row in range(rows):
        for col in range(cols):
            src_copy.set(row*2, col*2, src.get(row, col))
    for row in range(cols*2):
        for col in range(rows * 2):
            if row % 2 == 0: #All even rows - not black rows
                if col % 2 == 1:
                    if col == (cols * 2) - 1: #The last column - edge case
                        pix = src_copy.get(row, col - 1)
                        src_copy.set(row, col, pix)
                    else:
```

```
                        pix1 = src_copy.get(row, col-1)
                        pix2 = src_copy.get(row, col+1)
                        pix = ((pix1[0] + pix2[0])//2, (pix1[1] + pix2[1])//2, (pix1[2] + pix2[2]) // 2)
                        src_copy.set(row, col, pix)
                else: pass
            if row % 2 == 1: #Black rows
                if col % 2 == 0: #Not completely black columns
                    if row == (rows * 2) - 1: #Last row - edge case
                        pix = src_copy.get(row - 1, col)
                        src_copy.set(row, col, pix)
                    else:
                        pix1 = src_copy.get(row - 1, col)
                        pix2 = src_copy.get(row + 1, col)
                        pix = ((pix1[0] + pix2[0]) // 2, (pix1[1] + pix2[1]) // 2, (pix1[2] + pix2[2]) // 2)
                        src_copy.set(row, col, pix)
                if col % 2 == 1:
                    if col == (cols * 2) - 1:
                        pix = src_copy.get(row, col - 1)
                        src_copy.set(row, col, pix)
                    else:
                        if row != (rows * 2) - 1:
                            pix1 = src_copy.get((row - 1), (col - 1))
                            pix2 = src_copy.get((row - 1), (col + 1))
                            pix3 = src_copy.get((row + 1), (col + 1))
                            pix4 = src_copy.get((row + 1), (col - 1))
                            pix = ((pix1[0] + pix2[0] + pix3[0] + pix4[0])//4, (pix1[1] + pix2[1] + pix3[1] + pix4[1]) // 4, (pix1[2] + pix2[2] + pix3[2] + pix4[2]) // 4)
                            src_copy.set(row, col, pix)

    for row in range(rows * 2):
        for col in range(cols * 2):
            if row % 2 == 1 and col % 2 == 1:
                if col == (cols * 2) - 1: #Edge case - last col
                    pix = src_copy.get(row, col - 1)
                    src_copy.set(row, col, pix)
                elif row == (row * 2) - 1: #Edge case - last row
                    pix = src_copy.get(row - 1, col)
                    src_copy.set(row, col, pix)
                else:
                    if row != (rows * 2) - 1:
                        pix1 = src_copy.get((row - 1), (col - 1))
                        pix2 = src_copy.get((row - 1), (col + 1))
                        pix3 = src_copy.get((row + 1), (col + 1))
                        pix4 = src_copy.get((row + 1), (col - 1))
                        pix = ((pix1[0] + pix2[0] + pix3[0] + pix4[0])//4, (pix1[1] + pix2[1] + pix3[1] + pix4[1]) // 4, (pix1[2] + pix2[2] + pix3[2] + pix4[2]) // 4)
                        src_copy.set(row, col, pix)
                    else:
                        pix1 = src_copy.get(row, col - 1)
                        pix2 = src_copy.get(row, col + 1)
                        pix = ((pix1[0] + pix2[0]) // 2, (pix1[1] + pix2[1]) // 2, (pix1[2] + pix2[2]) // 2)
                        src_copy.set(row, col, pix)
    return src_copy

def maskreader(maskfile): #Reads the maskfile and returns a list with values of maskfile
    '''
```

```python
    Returns a tuple of mask list and length.
    This is a helper function that is used in apply_mask function to read
    the mask file and return a list of n by n.

    maskfile is a text file containing n by n mask.

    Args:
    - maskfile: path to file specifying mask

    Returns:
    Tuple of lst of n by n and length
    '''
    f = open(maskfile, 'r');
    lst = []
    for i in f:
        lst.append(int(i))
    mat_len = lst[0]; mask_lst = lst[1::]
    f.close()
    return (mask_lst, mat_len)

def apply_mask(src: MyImage, maskfile: str, average: bool = True) -> MyImage:
    """Returns an copy of src with the mask from maskfile applied to it.

    maskfile specifies a text file which contains an n by n mask. It has the
    following format:
    - the first line contains n
    - the next n^2 lines contain 1 element each of the flattened mask

    Args:
    - src: the image on which the mask is to be applied
    - maskfile: path to a file specifying the mask to be applied
    - average: if True, averaging should to done when applying the mask

    Returns:
    an image which the result of applying the specified mask to src.
    """
    rows = src.size[1]; cols = src.size[0]
    src_copy = MyImage(src.size)
    # for row in range(rows):
    #     for col in range(cols):
    #         src_copy.set(row, col, src.get(row, col))
    vals = maskreader(maskfile)
    mask_lst = vals[0]; n = vals[1]
    # print(f"Mask list: {mask_lst} \n length = {n} \n center at {center} with value {
        mask_lst[center]}")
    for i , pixels in enumerate(src):
        row, col = divmod(i,cols) #gives row index with corresponding column index to
            iterate over
        # print(row, col)
        val = 0; total = 0
        for x in range(n):
            for y in range(n):
                mask_row = x - (n // 2) #Computing row mask
                mask_col = y - (n // 2) #Computing col mask
                if row + mask_row >= 0 and row + mask_row < rows and col + mask_col >= 0 and
                    col + mask_col < cols:
                    total += mask_lst[n * x + y]
                    pix = src.get(row + mask_row, col + mask_col)
                    # print(f"Row {row}  Col {col}  RowMask {mask_row}  ColMask {mask_col}
                        pixel {pix}")
                    val += ((pix[0] + pix[1] + pix[2])) // 3 * mask_lst[n * x + y]
        if average == True: val = val // total
        if val > 255: val = 255
        if val < 0: val = 0
        src_copy.set(row, col, (val,val,val))
    return src_copy
```

Code Listing 3: Image Operations