Contents

1 Basic Test Results

```
Starting tests...
Thu 20 Jun 2024 00:23:00 IDT
1aa5e0677bb19f300acf485807b73723c281dd7b  -


Archive:  /tmp/bodek.51vlmx5k/intro2cs2/ex6/daniel.rez/presubmission/submission
  inflating: src/image_editor.py


Running presubmit code tests...
11 passed tests out of 11 in test set named 'presubmit'.
result_code    presubmit    11    1
Done running presubmit code tests

Finished running the presubmit tests

Additional notes:

Make sure to thoroughly test your code.

```

2 image editor.py

```python
###################################################################
# FILE : image_editor.py
# WRITER : daniel_riazanov , daniel.rez , 336119300
# EXERCISE : intro2cs ex6 2024
# DESCRIPTION: A simple program that...
# STUDENTS I DISCUSSED THE EXERCISE WITH: None
# WEB PAGES I USED: None
# NOTES: None
###################################################################

################################################################################
#                                    Imports                                   #
################################################################################
from ex6_helper import *
from typing import Optional
from math import floor
import sys


################################################################################
#                                   Functions                                  #
################################################################################


def separate_channels(image: ColoredImage) -> List[SingleChannelImage]:
    """
     Separates the color channels of a given RGB image into individual single-channel images.

     :param image: A three-dimensional list representing the colored image (dimensions: rows × columns × channels)
     :return: A list of two-dimensional lists, each representing a single color channel (dimensions: channels × rows × column
    """

    # Determining the dimensions of the input image
    rows = len(image)
    columns = len(image[0])
    channels = len(image[0][0])

    # Initializing return value - list of separated img channels
    img_with_separated_channels = [
        [[image[i][j][k] for j in range(columns)] for i in range(rows)]
        for k in range(channels)
    ]

    return img_with_separated_channels


def combine_channels(channels: List[SingleChannelImage]) -> ColoredImage:
    """
        Combines separate single-channel images into a single multichannel image.

        :param channels: A list of two-dimensional lists (dimensions: rows × columns), each representing a single color chann
        :return: A three-dimensional list (dimensions: rows × columns × channels) representing the combined multichannel imag
    """

    # Determining the dimensions of the input single-channel images
    num_channels = len(channels)
    rows = len(channels[0])
    columns = len(channels[0][0])

    # Initializing the combined image with the same dimensions as the input channels
    combined_image = [
        [
            [channels[k][i][j] for k in range(num_channels)]
            for j in range(columns)
        ]
        for i in range(rows)
    ]

    return combined_image


def RGB2grayscale(colored_image: ColoredImage) -> SingleChannelImage:
    """
       Converts a color image to grayscale.

       :param colored_image: A three-dimensional list representing the color image in RGB format
       :return: A two-dimensional list representing the grayscale image
    """
    # Initializing the list to hold the grayscale image
    grayscale_image = []

    # Iterating over each row in the colored image
    for row in colored_image:
        # Initializing the list to hold the grayscale values for the current row
        grayscale_row = []
        # Iterating over each pixel in the row
        for pixel in row:
            # Calculating the grayscale value using the weighted sum formula
            grayscale_value = round(pixel[0] * 0.299 + pixel[1] * 0.587 + pixel[2] * 0.114)
            # Appending the grayscale value to the current row
            grayscale_row.append(grayscale_value)
        # Appending the current row to the grayscale image
        grayscale_image.append(grayscale_row)

    return grayscale_image


def blur_kernel(size: int) -> Kernel:
    """
        Creates a blur kernel of a given size.

        :param size: The size of the blur kernel (must be an odd positive integer)
        :return: A 2D list representing the blur kernel
    """
    # Calculating the value for each element in the kernel
    value = 1 / (size * size)
    # Initializing the blur kernel with the calculated value
    kernel = [[value for _ in range(size)] for _ in range(size)]
    return kernel


def apply_kernel(image: SingleChannelImage, kernel: Kernel) -> SingleChannelImage:
    """
        Applies convolutional kernel to an image.

        :param image: A two-dimensional list representing the single-channel image
        :param kernel: A two-dimensional list representing the convolutional kernel
        :return: A two-dimensional list representing the new image after applying the kernel
    """
    # Getting dimensions of the image and kernel
    image_height = len(image)
    image_width = len(image[0])
    kernel_size = len(kernel)
    offset = kernel_size // 2  # Calculating the offset for the kernel

    # Creating a new image with the same dimensions as the original image
    new_image = [[0 for _ in range(image_width)] for _ in range(image_height)]
```

```python

    # Defining a helper function to clamp values between 0 and 255
    def clamp(value):
        return max(0, min(255, round(value)))

    # Applying the kernel to each pixel in the image
    for i in range(image_height):
        for j in range(image_width):
            # Initializing the sum for the current pixel
            pixel_sum = 0.0

            # Iterating over the kernel
            for ki in range(kernel_size):
                for kj in range(kernel_size):
                    # Calculating the corresponding image coordinates
                    ni = i + ki - offset
                    nj = j + kj - offset

                    # Handling edge cases by using the value of the border pixel for out-of-bounds coordinates
                    if ni >= image_height or ni < 0 or nj < 0 or nj >= image_width:
                        nj = j
                        ni = i

                    # Adding the weighted value to the pixel sum
                    pixel_sum += image[ni][nj] * kernel[ki][kj]

            # Assigning the clamped sum to the new image
            new_image[i][j] = clamp(pixel_sum)

    return new_image


def bilinear_interpolation(image: SingleChannelImage, y: float, x: float) -> int:
    """
       Performs bilinear interpolation on an image at a specific floating-point coordinate.

       :param image: A two-dimensional list representing the single-channel image
       :param y: The y-coordinate (row) where interpolation is being performed
       :param x: The x-coordinate (column) where interpolation is being performed
       :return: An integer representing the interpolated pixel value clamped between 0 and 255
    """
    # Getting the dimensions of the image
    height = len(image)
    width = len(image[0])

    # Get the integer parts of the coordinates
    x1 = int(x)
    y1 = int(y)

    # Getting the fractional parts of the coordinates
    x_diff = x - x1
    y_diff = y - y1

    # Getting the neighboring pixel values
    x2 = min(x1 + 1, width - 1)
    y2 = min(y1 + 1, height - 1)

    # Getting the values of the four surrounding pixels
    Q11 = image[y1][x1]
    Q21 = image[y1][x2]
    Q12 = image[y2][x1]
    Q22 = image[y2][x2]

    # Performing bilinear interpolation
    R1 = Q11 * (1 - x_diff) + Q21 * x_diff
    R2 = Q12 * (1 - x_diff) + Q22 * x_diff
    P = R1 * (1 - y_diff) + R2 * y_diff

    # Rounding to the nearest integer and clamping the result to the range [0, 255]
    return max(0, min(255, round(P)))


def resize(image: SingleChannelImage, new_height: int, new_width: int) -> SingleChannelImage:
    """
       Resizes an image to new dimensions using bilinear interpolation.

       :param image: A two-dimensional list representing the single-channel image
       :param new_height: The desired height for the resized image
       :param new_width: The desired width for the resized image
       :return: A two-dimensional list representing the resized image
    """
    # Getting the original dimensions of the image
    original_height = len(image)
    original_width = len(image[0])

    # Creating a new image with the specified dimensions
    new_image = [[0 for _ in range(new_width)] for _ in range(new_height)]

    # Calculating the scaling factors
    y_scale = (original_height - 1) / (new_height - 1)
    x_scale = (original_width - 1) / (new_width - 1)

    # Mapping each pixel in the new image to the source image
    for i in range(new_height):
        for j in range(new_width):
            # Calculating the corresponding source coordinates
            y = i * y_scale
            x = j * x_scale

            # Performing bilinear interpolation to get the pixel value
            new_image[i][j] = bilinear_interpolation(image, y, x)

    return new_image


def rotate_90(image: Image, direction: str) -> Image:
    """
    Rotates an image by 90 degrees in the specified direction ('R' for right or 'L' for left).

    :param image: A two-dimensional or three-dimensional list representing the image (single-channel or color)
    :param direction: A string indicating the direction of rotation ('R' for right, 'L' for left)
    :return: A rotated image with the same type as the input image
    """

    # Checking if the direction is valid
    if direction not in ('R', 'L'):
        raise ValueError("Invalid direction. Use 'R' for right or 'L' for left.")

    # Different approach for colored and chanel-separated image
    # When image is a color image (3D list)
    if isinstance(image[0][0], list):
        height = len(image)
        width = len(image[0])
        depth = len(image[0][0])  # Counting the number of color channels in the image
        # Initializing the new image with swapped dimensions for rotation
        new_image = [[[0 for _ in range(depth)] for _ in range(height)] for _ in range(width)]
        if direction == 'R':
            # Rotating the image 90 degrees to the right
            for i in range(height):
                for j in range(width):
                    for k in range(depth):
                        new_image[j][height - 1 - i][k] = image[i][j][k]
        elif direction == 'L':
            # Rotating the image 90 degrees to the left
            for i in range(height):
                for j in range(width):
```

```python
                    for k in range(depth):
                        new_image[width - 1 - j][i][k] = image[i][j][k]
    # When image is single-channel image (2D list)
    else:
        height = len(image)
        width = len(image[0])
        # Initializing the new image with swapped dimensions for rotation
        new_image = [[0 for _ in range(height)] for _ in range(width)]
        if direction == 'R':
            # Rotating the image 90 degrees to the right
            for i in range(height):
                for j in range(width):
                    new_image[j][height - 1 - i] = image[i][j]
        elif direction == 'L':
            # Rotating the image 90 degrees to the left
            for i in range(height):
                for j in range(width):
                    new_image[width - 1 - j][i] = image[i][j]

    return new_image


def get_edges(image: SingleChannelImage, blur_size: int, block_size: int, c: float) -> SingleChannelImage:
    """
      Detects edges in an image using the specified parameters.

      :param image: A two-dimensional list representing the single-channel image
      :param blur_size: The size of the kernel to be used for blurring
      :param block_size: The size of the block to be used for threshold calculation
      :param c: The constant to be subtracted from the block average for thresholding
      :return: A two-dimensional list representing the edge-detected image
    """
    # Creating a blur kernel and applying it to the image to get a blurred image
    kernel = blur_kernel(blur_size)
    blurred_image = apply_kernel(image, kernel)

    # Getting the dimensions of the image
    image_height = len(image)
    image_width = len(image[0])
    r = block_size // 2  # Calculating the radius of the block

    # Initializing the new image with the same dimensions as the original
    new_image = [[0 for _ in range(image_width)] for _ in range(image_height)]

    # Calculating the threshold for each pixel
    for i in range(image_height):
        for j in range(image_width):
            # Calculating the average value in the block around (i, j)
            block_sum = 0
            block_count = 0
            for m in range(-r, r + 1):
                for n in range(-r, r + 1):
                    ni = max(0, min(image_height - 1, i + m))
                    nj = max(0, min(image_width - 1, j + n))
                    block_sum += blurred_image[ni][nj]
                    block_count += 1
            block_avg = block_sum / block_count
            threshold = block_avg - c

            # Determining the value of the pixel in the new image
            if blurred_image[i][j] < threshold:
                new_image[i][j] = 0  # Black
            else:
                new_image[i][j] = 255  # White
    return new_image


def quantize(image: SingleChannelImage, N: int) -> SingleChannelImage:
    """
       Quantizes an image to N levels.

       :param image: A two-dimensional list representing the single-channel image
       :param N: The number of quantization levels
       :return: A two-dimensional list representing the quantized image
    """
    # Getting the dimensions of the image
    height = len(image)
    width = len(image[0])

    # Creating a new image with the same dimensions as the original
    qimg = [[0 for _ in range(width)] for _ in range(height)]

    # Iterating over each pixel in the image
    for i in range(height):
        for j in range(width):
            # Getting the original pixel value
            original_value = image[i][j]
            # Calculating the quantized value
            quantized_value = round(floor((original_value * N) / 256) * (255 / (N - 1)))
            # Assigning the quantized value to the new image
            qimg[i][j] = quantized_value

    return qimg


def quantize_colored_image(image: ColoredImage, N: int) -> ColoredImage:
    """
        Quantizes a colored image to N shades per channel.

        :param image: A three-dimensional list representing the colored image
        :param N: The number of quantization levels
        :return: A three-dimensional list representing the quantized colored image
    """

    # Separating the channels
    separated_channels = separate_channels(image)
    # Quantizing each channel
    quantized_channels = [quantize(channel, N) for channel in separated_channels]
    # Combining the channels back into a colored image
    combined_image = combine_channels(quantized_channels)

    return combined_image


def main():
    """
       Running the main program for image editing. The user defines the path of the image and can choose various
       operations to perform on the selected image during the program runtime. At the end, the user defines a path to save a
       modified copy of the original image.

       The program accepts one command-line argument:
       1. <image_path> - The path to the image file to be edited.

       Operations that can be performed on the image include:
       1. Convert to grayscale
       2. Blur image
       3. Resize image
       4. Rotate image by 90 degrees
       5. Create outline (edges) image
       6. Quantize image
       7. Display image
       8. Exit

       Usage:
       python image_editor.py <image_path>
    """
```

```python

    # Checking if the correct number of command-line arguments is provided, otherwise terminating program
    if len(sys.argv) != 2:
        print("Error: Invalid number of arguments. Usage: python image_editor.py <image_path>")
        return

    # Loading the image from the specified path
    image_path = sys.argv[1]
    image = load_image(image_path)

    while True:
        # Displaying the menu of operations
        print("\nChoose an operation:")
        print("1. Convert to grayscale")
        print("2. Blur image")
        print("3. Resize image")
        print("4. Rotate image by 90 degrees")
        print("5. Create outline (edges) image")
        print("6. Quantize image")
        print("7. Display image")
        print("8. Exit")

        # Getting the user's choice
        choice = input("Enter the humber of the operation: ").strip()

        if choice == '1':
            # Converting the image to grayscale if it is a color image
            if isinstance(image[0][0], list):
                image = RGB2grayscale(image)
                print("Image successfully converted to grayscale")
            else:
                print("Image is already in grayscale ")

        elif choice == '2':
            # Defining kernel for blurring based on user's arguments
            kernel_size = input("Enter the kernel size (positive and odd integer): ").strip()
            if kernel_size.isdigit() and int(kernel_size) > 0 and int(kernel_size) % 2 == 1:
                # If valid kernel, assign kernel_size
                kernel_size = int(kernel_size)
                bluring_kernel = blur_kernel(kernel_size)
                # 2 different approaches for colored and grayscale image

                # Blurring Colored image:
                if isinstance(image[0][0], list):
                    channels = separate_channels(image)
                    blurred_channels = [apply_kernel(channel, bluring_kernel) for channel in channels]
                    image = combine_channels(blurred_channels)

                # Blurring Grayscale image:
                else:
                    image = apply_kernel(image, bluring_kernel)

                # Notifying that the operation finished successfully
                print(f"Image blurred with kernel size {kernel_size}.")

            else:
                print("Invalid kernel size. It must be a positive odd integer.")

        elif choice == '3':
            # Defining the dimension for a new image based on user's arguments
            dimensions = input("Enter new dimensions (height, width): ").strip()
            # Trying to map arguments if not succeeded the user entered wrong format
            try:
                new_height, new_width = map(int, dimensions.split(","))
                if new_height > 1 and new_width > 1:
                    # Passed all validations

                    # Colorful image:
```

```
                    if isinstance(image[0][0], list):
                        channels = separate_channels(image)
                        resized_channels = [resize(channel, new_height, new_width) for channel in channels]
                        image = combine_channels(resized_channels)

                    # Grayscale image:
                    else:
                        image = resize(image, new_height, new_width)

                    # Notifying that the operation finished successfully
                    print(f"Image resized to {new_height}x{new_width}.")

                else:
                    print("Invalid dimensions. Height and width must be greater than 1.")
            except:
                print("Invalid input format. Enter dimension as <height,weight>")

        elif choice == '4':
            # Defining the direction to rotate the original image  based on user's arguments
            direction = input("Enter rotation direction ('R' or 'L'): ").strip().upper()
            if direction in ('R', 'L'):
                # valid direction entered
                image = rotate_90(image, direction)

                # Notifying that the operation finished successfully
                print(f"Image rotated to {direction}.")

            else:
                print("Invalid direction. Enter 'R' for right or 'L' for left")

        elif choice == '5':
            # Defining the parameters to outline image edges
            params = input("Enter blur size, block size, and c value <blur_size,block_size,c>: ").strip()
            # Trying to map arguments if not succeeded the user entered wrong format
            try:
                blur_size, block_size, c = map(int, params.split(","))
                if blur_size % 2 == 1 and block_size % 2 == 1 and blur_size > 0 and block_size > 0 and c >= 0:
                    # Passed validations
                    # If image is colorful, firstly convert to greyscale :
                    if isinstance(image[0][0], list):
                        image = RGB2grayscale(image)
                    # Otherwise apply get_edges instantly
                    image = get_edges(image, blur_size, block_size, c)

                    # Notifying that the operation finished successfully
                    print("Outline (edges) image created.")

                else:
                    print(
                        "Invalid values. Blur size and block size must be positive odd integers, and c must be "
                        "non-negative.")
            except:
                print("Invalid input format. Enter values as <blur_size,block_size,c>.")

        elif choice == '6':
            # Defining the number of tones to quantize image
            tones = input("Enter the number of tones for quantization (positive integer greater than 1): ").strip()
            if tones.isdigit() and int(tones) > 1:
                # Passed validations
                tones = int(tones)
                #  Approach for Colored image
                if isinstance(image[0][0], list):
                    image = quantize_colored_image(image, tones)
                else:
                    #  Approach for chanel-separated image
                    image = quantize(image, tones)

                # Notifying that the operation finished successfully
                print(f"Image quantized to {tones} tones.")

            else:
                print("Invalid number of tones. It must be a positive integer greater than 1.")

        elif choice == '7':
            # Representing current image after modifications during program lifetime based on helper file
            show_image(image)

        elif choice == '8':
            # Saving image to a specified path based on helper file
            save_path = input("Enter path to save the image: ").strip()
            save_image(image, save_path)

            # Notifying that the operation finished successfully
            print(f"Image saved to {save_path}")

            # Finishing program
            break
        # Notifying that the choice is invalid and asking for re-input
        else:
            print("Invalid choice. Please enter a number between 1 and 8.")


if __name__ == '__main__':
    main()
```