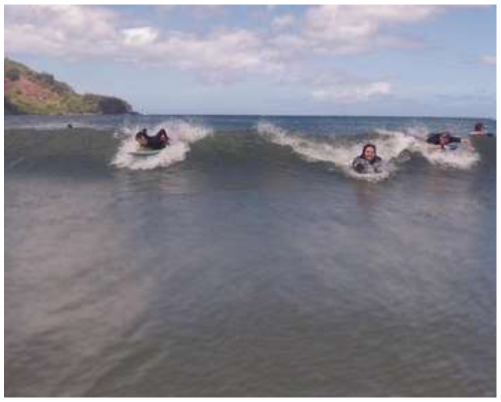
CSCI 340
Programming Assignment 1
Seam Carving

Due Date: February 13
Points: 40

This assignment is similar to one used at Princeton University.

Seam-carving is a content-aware image resizing technique where the image is reduced in size by one pixel of height (or width) at a time. A *vertical seam* in an image is a path of pixels connected from the top to the bottom with one pixel in each row; a *horizontal seam* is a path of pixels connected from the left to the right with one pixel in each column. Below left is the original 505-by-287 pixel image; below right is the result after removing 150 vertical seams, resulting in a 30% narrower image. Unlike standard content-agnostic resizing techniques (such as cropping and scaling), seam carving preserves the most interest features (aspect ratio, set of objects present, etc.) of the image.

The underlying algorithm is simple and elegant, but the technique was not discovered until 2007 by Shai Avidan and Ariel Shamir. The original SIGGRAPH video can be seen at: https://www.youtube.com/watch?v=6NcIJXTlugc, and is well worth watching. Now, seam carving is a core feature in Adobe Photoshop and other computer graphics applications.



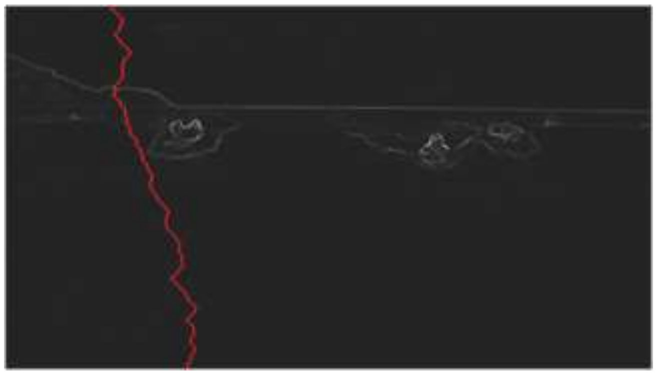
In this assignment, you will create a data type that resizes an image using the seam-carving technique. Finding and removing a seam involves three parts and a tiny bit of notation:

> *Notation.* In image processing, pixel (*r, c*) refers to the pixel in row r, and column c, with pixel (0, 0) at the upper left corner and pixel (*height* − 1, *width* − 1) at the bottom right corner.
>
> We also assume that the color of each pixel is represented in RGB (red, green, blue) space, using three integers between 0 and 255. This is consistent with the java.awt.Color data type.

1. *Energy calculation.* The first step is to calculate the *energy* of a pixel, which is a measure of its importance—the higher the energy, the less likely that the pixel will be included as part of a seam (as we'll see in the next step). In this assignment, you will use the *dual-gradient* energy function, which is described below. Here is the dual gradient of the surfing image above:

The energy is high (white) for pixels in the image where there is a rapid color gradient (such as the boundary between the sea and sky and the boundary between the surfer on the left and the ocean behind him). The seam-carving technique avoids removing such high-energy pixels.

2. *Seam identification.* The next step is to find a vertical seam of minimum total energy. The red line below shows the seam found. Seams cannot wrap around the image (e.g., a vertical seam cannot cross from the leftmost column of the image to the rightmost column).



3. *Seam removal.* The final step is remove from the image all of the pixels along the vertical or horizontal seam.

EasyBufferedImage

Java comes with a fairly complete set of classes for manipulating images. These classes are a bit hard to use, however, so you instructor is giving you the EasyBufferedImage class, developed by Prof. Kenny Hunt at UW – La Crosse, as well as several sample programs using the class, to make manipulation easier. Documentation for EasyBufferedImage can be found here.

The SeamCarver API. Your task is to implement the following mutable data type:

```
public class SeamCarver {
   // create a seam carver object based on the given picture
   public SeamCarver(EasyBufferedImage picture)

   // energy of pixel at column x and row y
   public double getEnergy(int row, int col)

   // sequence of indices for vertical seam
   public int[] findVerticalSeam()

   // sequence of indices for a horizontal seam
   public int[] findHorizontalSeam()
```

```
// find and remove vertical seam from the picture
public void findAndRemoveVerticalSeam()

// find and remove horizontal sema from the picture
public void findAndRemoveHorizontalSeam()
```

Your code should throw an exception when called with invalid arguments, as specified below.

- In `getEnergy()` the indices *row and col* are integers between 0 and *H* − 1 and between 0 and *W* − 1 respectively, where *W* is the width of the current image and *H* is the height. Throw a java.lang.`IndexOutOfBoundsException` if either r or c is outside its prescribed range in `getEnergy()`.
- In the constructor, if the picture is null, throw a `NullPointerException`.

- **Computing the energy of a pixel.** You will use the *dual-gradient energy function*: The energy of pixel (*r,c*) is $\Delta_x^2(r,c) + \Delta_y^2(r,c)$, where the square of the *x*-gradient $\Delta_x^2(r,c) = R_x(r,c)^2 + G_x(r,c)^2 + B_x(r,c)^2$, and where the central differences $R_x(r,c)$, $G_x(r,c)$, and $B_x(r,c)$ are the differences of red, green, and blue components between pixel (*r,c*+1) and pixel (*r,c-1*). The square of the *y*-gradient $\Delta_y^2(r,c)$ is defined in an analogous manner. To handle pixels on the borders of the image, calculate energy by defining the leftmost and rightmost columns as adjacent and the topmost and bottommost rows as adjacent. For example, to compute the energy of a pixel (r,0) in the leftmost column, we use its right neighbor (r,1) and its left neighbor (r,*W* − 1).

Consider the 4-by-3 image with RGB values (each component is an integer between 0 and 255) as shown in the table below.

(255, 101, 51)	(255, 101, 153)	(255, 101, 255)
(255,153,51)	(255,153,153)	(255,153,255)
(255,203,51)	(255,204,153)	(255,205,255)
(255,255,51)	(255,255,153)	(255,255,255)

Non-border pixel example. The energy of pixel (2,1) is calculated from pixels (2,0) and (2, 2) for the *x*-gradient

$R_x(2,1) = 255 - 255 = 0$,
$G_x(2,1) = 205 - 203 = 2$,
$B_x(2,1) = 255 - 51 = 204$,

yielding $\Delta_x^2(2,1) = 2^2 + 204^2 = 41620$;

and pixels (1, 1) and (3,1) for the *y*-gradient

$R_y(2,1) = 255 - 255 = 0$,
$G_y(2,1) = 255 - 153 = 102$,
$B_y(2,1) = 153 - 153 = 0$,

yielding $\Delta_y^2(2,1) = 102^2 = 10404$.

Thus, the energy of pixel (2,1) is 41620 + 10404 = 52024. Similarly, the energy of pixel (1, 1) is $204^2 + 103^2 = 52225$.

Border pixel example. The energy of the border pixel (0,1) is calculated by using pixels (0, 0) and (0,2) for the *x*-gradient

$R_x(0,1) = 255 - 255 = 0$,
$G_x(0,1) = 101 - 101 = 0$,
$B_x(0,1) = 255 - 51 = 204$,

yielding $\Delta_x^2(0,1) = 204^2 = 41616$;

and pixels (3,1) and (1, 1) for the *y*-gradient

$R_y(0,1) = 255 - 255 = 0$,
$G_y(0,1) = 255 - 153 = 102$,
$B_y(0,1) = 153 - 153 = 0$,

yielding $\Delta_y^2(2,1) = 102^2 = 10404$.

Thus, the energy of pixel (0,1) is 41616 + 10404 = 52020.

20808.0	52020.0	20808.0
20808.0	52225.0	21220.0
20809.0	52024.0	20809.0
20808.0	52225.0	21220.0

Finding a vertical seam. The `findVerticalSeam()` method returns an array of length *H* such that entry *i* is the column number of the pixel to be removed from row *i* of the image. For example, consider the 5-by-6 image below (supplied as 5x6.png).

(78,209, 79)	(63,118,247)	(92,175, 95)	(243, 73,183)	(210,109,104)	(252,101,119)
(224,191,182)	(108, 89, 82)	(80,196,230)	(112,156,180)	(176,178,120)	(142,151,142)
(117,189,149)	(171,231,153)	(149,164,168)	(107,119, 71)	(120,105,138)	(163,174,196)
(163,222,132)	(187,117,183)	(92,145, 69)	(158,143, 79)	(220, 75,222)	(189, 73,214)
(211,120,173)	(188,218,244)	(214,103, 68)	(163,166,246)	(79,125,246)	(211,201, 98)

- The corresponding pixel energies are shown below, with a minimum energy vertical seam highlighted in red. In this case, the method `findVerticalSeam()` returns the array { 3, 4, 3, 2, 2 }.

57685.0	50893.0	91370.0	25418.0	33055.0	37246.0
15421.0	56334.0	22808.0	54796.0	11641.0	25496.0
12344.0	19236.0	52030.0	17708.0	44735.0	20663.0
17074.0	23678.0	30279.0	80663.0	37831.0	45595.0
32337.0	30796.0	4909.0	73334.0	40613.0	36556.0

- Remember that seams cannot wrap around the image. When there are multiple vertical seams with minimal total energy, your method can return any such seam.

The Algorithm

There are an exponential number of potential vertical paths through the image, so a brute force approach to finding the one with minimal energy is bound to fail. We need to be smarter. The algorithm we will use to find the vertical seam uses a technique called ***dynamic programming***. The idea is to create another array, the same size as the image, to store partial results. Your instructor used the very original name, `pathWeight[][]`.

Each element of the array will contain the total energy of the minimal path from the top of the image down to that element.

The zeroth row will just contain the energy for that pixel. For example, in the figure below, pathWeight[0][3] = energy(0,3) = 25418.

The first row will contain the sum of the energy for that pixel, plus the min of the pathWeights for one of the three elements immediately above it. For example, in the figure below:
pathWeight[1][4] = min(pathWeight[0][3], pathWeight[0][4], pathWeight[0][5]) + getEnergy(1,4).

Similarly, the second row will only have to look at the path weights for the first row to do the calculations.

Once the array is full, we find the minimum value in the last row, and work our way back up, by looking at the three elements above it.

59700.0	50893.0	91370.0	25418.0	33055.0	37246.0
66314.0	132802.0	48226.0	80214.0	37059.0	78546.0
112293.0	67462.0	100256.0	54767.0	81794.0	57722.0
84536.0	91140.0	85046.0	135430.0	92598.0	103317.0
116873.0	115332.0	89955.0	158380.0	133211.0	129154.0

Finding a horizontal seam is done in an analogous manner.

Submission. Submit `SeamCarver.java` using the `submit` command on the lab machines. Emailed submissions will NOT be accepted.

Use the default package for you class. If you have a package statement in your code, your code will NOT compile with your instructor's test main and you will receive 0 points for the assignment.

Note that your instructor will use his own test main to test your code. Thus, you must follow the API given above exactly.

Your instructor will use the following commands to compile and run your program:

```
javac SeamCarver.java EasyBufferedImage.java TestMain.java
java TestMain
```