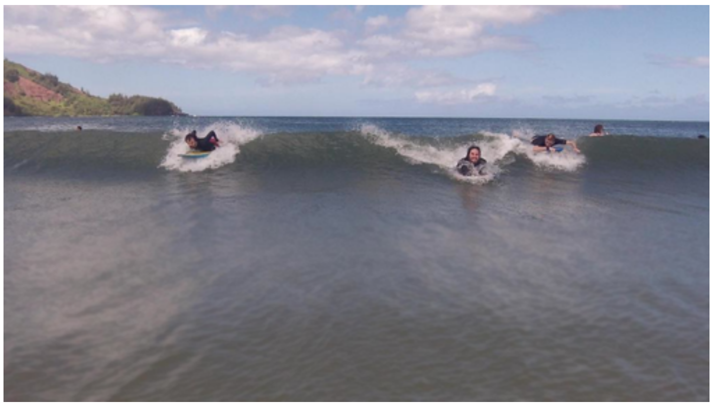
Homework 5: Seam Carving

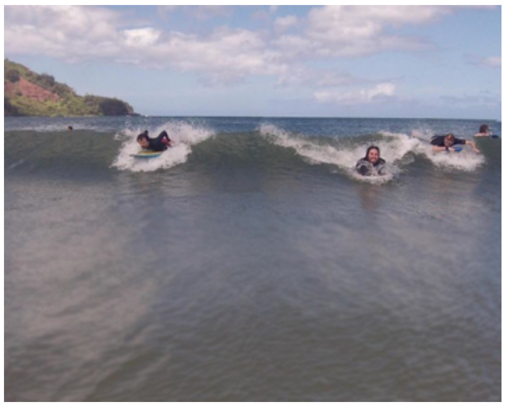
Getting the Skeleton Files

As usual, run `git pull skeleton master` to get the skeleton files. If you're using IntelliJ, you might need to manually add the SeamRemover.jar file to your project. If you're working from the command line, you'll need to make sure SeamRemover.jar is in your classpath. The easiest way to do this is to copy it into your course-materials-sp16/javalib/ folder.

Introduction

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 is the original 505-by-287 pixel image; further below we see the result after removing 150 vertical seams, resulting in a 30% narrower image. Unlike standard content-agnostic resizing techniques (e.g. cropping and scaling), the most interesting features (aspect ratio, set of objects present, etc.) of the image are preserved.





In this assignment, you will create a data type that resizes a W-by-H image using the seam-carving technique.

Finding and removing a seam involves three parts and a tiny bit of notation:

1. *Notation*. In image processing, pixel (x, y) refers to the pixel in column x and row y, with pixel (0, 0) at the upper left corner and pixel (W − 1, H − 1) at the bottom right corner. This is consistent with the Picture data type in stdlib.jar. Warning: this is the opposite of the standard mathematical notation used in linear algebra where (i, j) refers to row i and column j and with Cartesian coordinates where (0, 0) is at the lower left corner.

(0, 0)	(1, 0)	(2, 0)
(0, 1)	(1, 1)	(2, 1)
(0, 2)	(1, 2)	(2, 2)
(0, 3)	(1, 3)	(2, 3)

We also assume that the color of a pixel is represented in RGB 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 each pixel, which is a measure of the importance of each pixel—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 implement the dual gradient energy function, which is described below. Here is the dual gradient of the surfing image above:

A high-energy pixel corresponds to a pixel where there is a sudden change in color (such as the boundary between the sea and sky or the boundary between the surfer on the left and the ocean behind him). In the image above, pixels with higher energy values have whiter values. The seam-carving technique avoids removing such high-energy pixels.

1. *Seam identification*. The next step is to find a vertical seam of minimum total energy. This is similar to the classic shortest path problem in an edge-weighted digraph except for the following:
   - The weights are on the vertices instead of the edges.
   - We want to find the shortest path from any of W pixels in the top row to any of the W pixels in the bottom row.
   - The digraph is acyclic, where there is a downward edge from pixel (x, y) to pixels (x − 1, y + 1), (x, y + 1), and (x + 1, y + 1), assuming that the coordinates are in the prescribed range.
2. *Seam Removal*. The final step is remove from the image all of the pixels along the seam. The logic for this method has been implemented for you in the supplementary SeamRemover class, provided in SeamRemover.jar.

```
public class SeamRemover {
    // These methods are NOT destructive
    public static Picture removeHorizontalSeam(Picture picture, int[] seam)  // retu
    public static Picture removeVerticalSeam(Picture picture, int[] seam)    // retu
}
```

SeamCarver

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

```
public class SeamCarver {
    public SeamCarver(Picture picture)
    public Picture picture()                          // current picture
    public     int width()                            // width of current picture
    public     int height()                           // height of current picture
    public  double energy(int x, int y)               // energy of pixel at column x and row
    public    void removeHorizontalSeam(int[] seam)   // remove horizontal seam from pict
    public    void removeVerticalSeam(int[] seam)     // remove vertical seam from pictur
}
```

energy(): Computing the Energy of a Pixel

We will use the dual gradient energy function: The energy of pixel (x, y) is $\Delta_x^2(x,y) + \Delta_y^2(x,y)$, where the square of the x-gradient $\Delta_x^2(x,y) = R_x(x,y)^2 + G_x(x,y)^2 + B_x(x,y)^2$, and where the central differences $R_x(x,y)$, $G_x(x,y)$, and $B_x(x,y)$ are the absolute value in differences of red, green, and blue components between pixel (x + 1, y) and pixel (x − 1, y). The square of the y-gradient $\Delta_y^2(x,y)$ is defined in an analogous manner. We define the energy of pixels at the border of the image to use the same pixels but to replace the non-existant pixel with the pixel from the opposite edge.

As an example, consider the 3-by-4 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)

Example 1: We calculate the energy of pixel (1, 2) in detail:

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

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

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

yielding $\Delta_y^2(1,2) = 102^2 = 10404.$ Thus, the energy of pixel (1, 2) is $41620 + 10404 = 52024.$

Test your understanding: The energy of pixel (1, 1) is $204^2 + 103^2 = 52225.$

Example 2: We calculate the energy of the border pixel (1, 0) in detail:

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

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

Since there is no pixel (x, y - 1) we wrap around and use the corresponding pixel from the bottom row the image, thus performing calculations based on pixel (x, y + 1) and pixel (x, height − 1).

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

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

Thus, the energy of pixel (1, 2) is $41616 + 10404 = 52020.$

Examples Summary: The energies for each pixel is given in the table below:

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

findVerticalSeam(): Finding a Minimum Energy Path

number of the pixel to be removed from row x of the image. For example, consider the 6-by-5 image below (supplied as 6x5.png).

- **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 6-by-5 image below (supplied as 6x5.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 pink. 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

When there are multiple vertical seams with minimal total energy, your method can return any such seam.

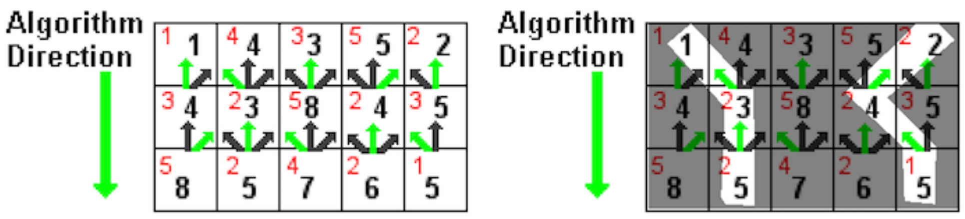
Your `findVerticalSeam` method should utilize dynamic programming. Recall the key idea behind any dynamic programming algorithm: the subproblem. Suppose we have the following definitions:

$M(i,j)$ - cost of minimum cost path ending at (i, j)
$e(i,j)$ - energy cost of pixel at location (i, j)

Then each subproblem is the calculation of $M(i,j)$ for some i and j. The top row is trivial, $M(i,0)$ is just $e(i,0)$ for all i. For lower rows, we can find $M(i,j)$ simply by adding the $e(i,j)$ to the minimum cost path ending at its top left, top middle, and top right pixels, or more formally:

$$M(i,j) = e(i,j) + min(M(i-1,j-1), M(i,j-1), M(i+1,j-1))$$

In short, we start from one side of the 2D image array and process row-by-row or column-by-column (for vertical and horizontal seam carving respectively).



Addendum: The Java language does not deal well with deep recursion, and thus a recursive approach will almost certainly not be able to handle images of largish size (say 500x500). We recommend writing your code iteratively.

An equivalent (but slower approach) is to build an explicit Graph object and run the DAGSPT algorithm. You are welcome to try this approach, but be warned it is slower, and it may not be possible to sufficiently optimize your code so that it passes the autograder timing tests.

findHorizontalSeam(): Avoiding Redundancy

should return an array of W such that entry y is the row number of the pixel to be removed from column y of the image. Your `findHorizontalSeam` method should NOT be a copy and paste of your `findVerticalSeam` method! Instead, considering transposing the image, running `findVerticalSeam`, and then transposing it back. The autograder will not test this, but a similar idea could easily appear on the final exam.

Other Program Requirements

Performance requirements. The `width()`, `height()`, and `energy()` methods should take constant time in the worst case. All other methods should run in time at most proportional to `W H` in the worst case.

Exceptions. Your code should throw an exception when called with invalid arguments. * By convention, the indices x and y are integers between 0 and W − 1 and between 0 and H − 1 respectively. Throw a `java.lang.IndexOutOfBoundsException` if either x or y is outside its prescribed range.

* Throw a `java.lang.IllegalArgumentException` if `removeVerticalSeam()` or `removeHorizontalSeam()` is called with an array of the wrong length or if the array is not a valid seam (i.e., two consecutive entries differ by more than 1).

Some Useful Files

PrintEnergy.java: For printing the energy calculations per pixel for an input image.

PrintSeams.java: Prints the energies and computed horizontal and vertical seams for an input image.

ShowEnergy.java: Shows the grayscale image corresponding to the energy computed by pixel.

ShowSeams.java: Displays the vertical and horizontal minimum energy seams for a given image.

SanityCheckTest.java: Basic JUnit tests consisting of the energy and path examples given in this spec.

SCUtility.java: Some utilities for testing SeamCarver.

SeamRemover.jar: Contains a SeamRemover class file with `removeHorizontalSeam()` and `removeVerticalSeam()` methods to use in your SeamCarver.

SeamCarverVisualizer.java: For the purposes of visualizing the frame-by-frame actions of your SeamCarver, we've provided you with a `SeamCarverVisualizer` class which you can run using the following command:

```
java SeamCarverVisualizer [filename] [numPixels to remove] [y (if horizontal carving)
```

Example:

```
java SeamCarverVisualizer images/HJoceanSmall.png 50 y
```

Extra Fun

Fun #1: Try out your SeamCarver on various real world images. I recommend human faces.

Fun #2: Try to implement a version of the `SeamCarver` class that avoids the need to recompute the entire energy matrix every time a seam is removed. This will require getting fancy with your data structures. If you do this, email Josh and let him know. This should make your SeamCarver class extremely fast.

Submission

submit `SeamRemover.jar` .

Homework 5 Navigation

FAQ

How do I debug this?

Make sure to try out the "Useful Files" above, especially the PrintEnergy and PrintSeams classes.

My code is slow (failing timing tests), what can I do to speed it up?

Some possible optimizations include (in decreasing order of likely impact):

- **Avoiding recalculation of energies for the same pixel over and over** (e.g. through creation of an explicit energy matrix of type `double[][]`). Essentially you want to memoize energy calculations.
- Don't use a HashMap for looking up data by row and column. Instead, use a 2D array. They are much faster. HashMaps are constant time, but the constant factor is significant.
- Not using `Math.pow` or `Math.abs` .
- Not storing an explicit `edgeTo` data structure. It is possible to rebuild the seam ONLY from the values for `M(i, j)` ! That is, you don't need to actually record the predecessor like you did in the 8puzzle assignment.
- Using a more clever approach than transposing your images (though this is not required to pass the autograder).

Credits

This assignment was originally developed by Josh Hug, with supporting development work by Maia Ginsburg and Kevin Wayne at Princeton University.