**Carved Algorithm Description**

*SeamCarver(Picture pic)*

The constructor for the seam carving class takes in a Picture. From this it stores the height and width of the image as ints, then stores the image’s pixels as a 2D array, set up as a list of x lists filled with y pixels. The Picture object itself is not stored.

*double energy(int x, int y)*

Each pixel’s energy is calculated as it is needed. This means that it is calculated when the energy method is called as well as when needed in the seam finding methods. The pixels to the left, right, top and bottom are all indexed in constant time through use of the 2D array, and their color values are then plugged into the basic mathematical energy algorithm we were given. Initial I thought to calculate every pixel’s energy value at once and store them, but decided against this as all of these energy values would potentially become incorrect as seams are removed, and they would then have to be updated repeatedly, despite the fact that only certain pixels’ energy values were required at any time.

*Picture getPicture()*

When this method is called a new Picture object is made and returned, using the current stored width and height values, and the pixel colors being stored in the 2D array.

*int width()* and *int height()*

Both of these methods return their respective values in constant time since each are being stored and updated as seams are removed.

*int[] findHorizontalSeam()*

First I search the first column of the image (x is 0 and y is from 0 to height-1) for the lowest energy pixel. This will be my starting pixel in the seam. Using its coordinates and the image’s width and height I calculate its index and store this in a new array the size of the image’s width. From here I go through each of the three pixels to the right of this one (at (x+1, y), (x+1, y+1), and (x+1, y-1)) and choose the one with the lowest energy value. I calculate this pixel’s index, store it in the array, and then use it to repeat the process, finding the lowest energy neighbor of each next pixel and storing their index, until I reach the rightmost edge of the image. I then return the array of indices.

The efficiency of this method is Θ(H+W), H being the height of the image and W being the width. This is because the first part of the method goes through each pixel in the first column of the image to find the lowest energy one, making it H times some constant number of small operations. After this it goes through each pixel to the right of it, incrementing x by 1 until it reaches the width, giving an additional W times some other constant operations. I have chosen not to estimate values for these other constants in the algorithm because the number of operations done in these cases are not dependent on the width and height of the image, being indeed constants, and because many of the exact numbers are dependent on Java’s implementations of the small utilities I used, as well as a number of arithmetic operations. Additionally, because I stored the pixels in a 2D array, referencing any of them is simply a matter of indexing the array with their coordinates, which can be done in constant time. What I chose to focus on instead is that these numbers of operations is always being done W or H number of times, respectively.

*int[] findVerticalSeam()*

Finding a vertical seam in an image is done in a similar fashion as finding the horizontal seam. The first row of the image is scanned to find the lowest energy pixel, and indices are stored as the method moves through each lowest energy neighbor of each pixel until it reaches the bottom edge of the image. The main difference here is that it is the y value of the pixel coordinates that is being incremented, and each considered neighbor is a pixel lower than the current on rather than to the right.

Also like the findHorizontalSeam method, I found this method to have an efficiency of Θ(W+H), essentially being the same classification but with the W and H rearranged to clarify that the image’s fully width is scanned, followed by its height.

*void removeHorizontalSeam(int[] indices)*

First of all, a variety of checks are run throughout this method to ensure the legality of the indices being passed in. I will not include these operations as part of my explanation or efficiency analysis because the algorithm could run without them. They are simply precautions to avoid invalid inputs and the breaks they would cause. They were also required by the lab instructions.

Assuming that all arguments are valid, the method iterates through each of the passed in indices and calculates the corresponding x and y coordinates, the x in this case corresponding directly to the value’s index in the indices array. Then, starting from the calculated y value, each pixel in the image’s x column (or y value in the x list of the 2D array) is shifted up by one pixel. Once this is done for every index in the array (or once for every pixel in the image’s width) the image’s height is decremented by 1.

In the worst case scenario, the seam being removed would be exactly the top row of the image, meaning that every pixel in every column of the image besides that top row would have to be shifted up by one. On simple idea to fix this would be to either shift all pixels up or down, depending on how far up or down the horizontal seam is on the image, but this would still require some sort of fix to every pixel’s indexing as shifting down would mean the first index is off by one and no longer 0, or a selective copy of the entire 2D array would have to be made, either solution just ending with the negatively impacted efficiency. Taking all of this into account I find the efficiency of the method to be Θ(H\*W), as every pixel in the image could potentially have to be altered.

*void removeVerticalSeam(int[] indices)*

Once again, the vertical implementation of seam removal is very similar to the horizontal method. Checks are made, and each index is iterated through, being a number of indices corresponding to the image’s height. The indices are used to find the corresponding x and y coordinates, y corresponding to the current index in the indices array. Then, each y row of the image is shifted to the left, starting from pixel x and proceeding until the pixel at the far right of the image (x=width-1) has been shifted over to the left by one. The stored width value is then decremented by 1.

My analysis of this method is identical to the previous one for horizontal removal. In the worst case, the furthest left column of the image is what makes up the seam, and so every pixel of every row besides that column would have to be altered, resulting in Θ(H\*W).

**The Secret message:**

So it seams