DarkRoom YEAH Hours

Peter Maldonado and Andrew Tierno

Outline For Today

- Array Review
- 2D Arrays (Grids)
- `GImages` and Pixels
- DarkRoom Overview
- Tips and Tricks

Array Review

How do we efficiently store data?

Arrays

Why Arrays?

- Great for representing a *fixed size* list of *homogenous* data
- Efficient and lightweight

Arrays access data using numerical indices with zero-indexing

- First element at index `0`
- Last element at index `length - 1`

Can store both objects (references) and primitives (values)

Arrays are objects but don't have any methods!

Array Operations

To create a new Array, we specify a `Type` and max `SIZE` in a call to `new`

```
Type[] myArray = new Type[SIZE];
```

To access an element in the array, use square brackets to select the index

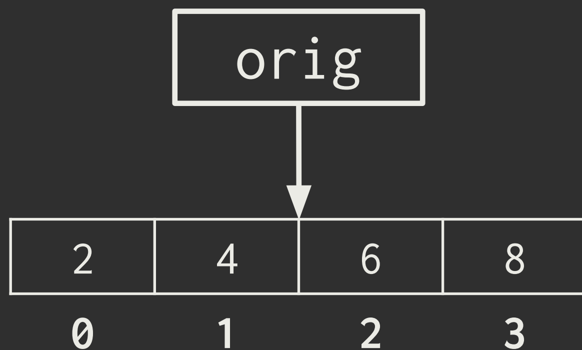
```
myArray[index];
```

(This expression is to a reference to that position in the array)

Arrays have a single variable that stores their length

```
myArray.length;
```

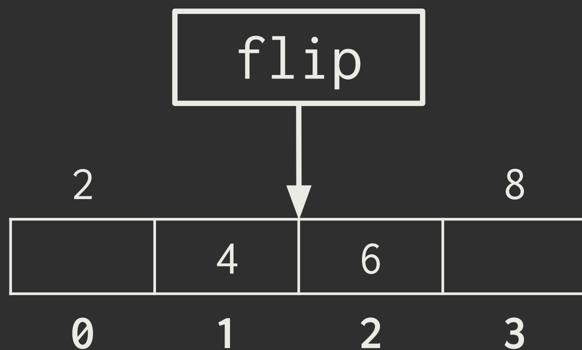
Flipping an Array



How many elements do we flip?

Is this number relative to the size of the array?

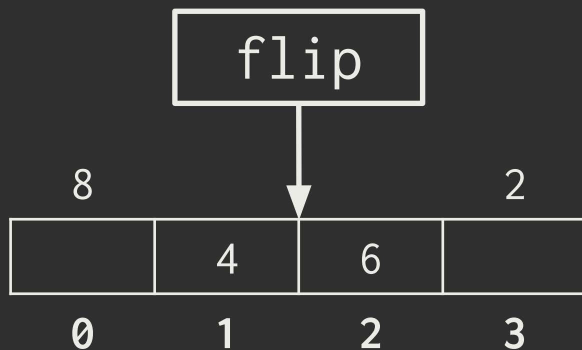
Flipping an Array



How many elements do we flip?

Is this number relative to the size of the array?

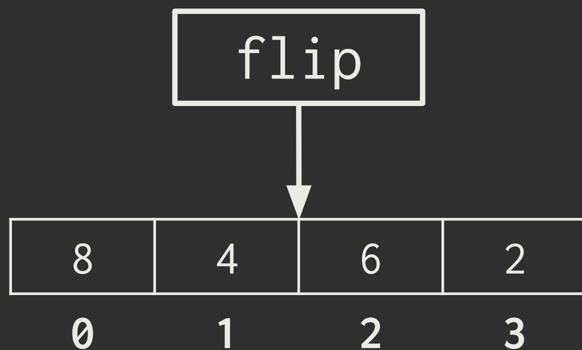
Flipping an Array



How many elements do we flip?

Is this number relative to the size of the array?

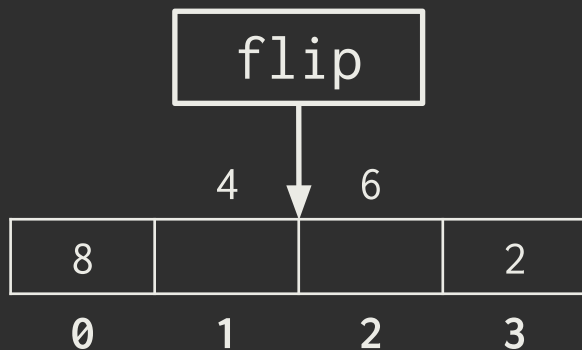
Flipping an Array



How many elements do we flip?

Is this number relative to the size of the array?

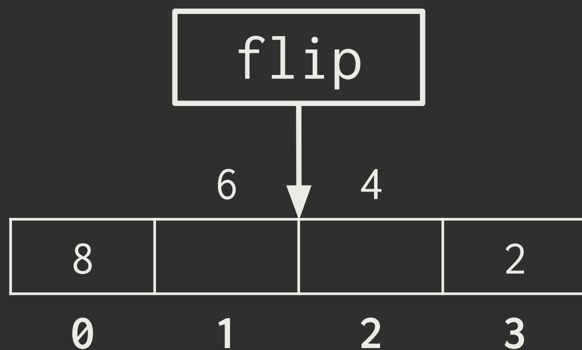
Flipping an Array



How many elements do we flip?

Is this number relative to the size of the array?

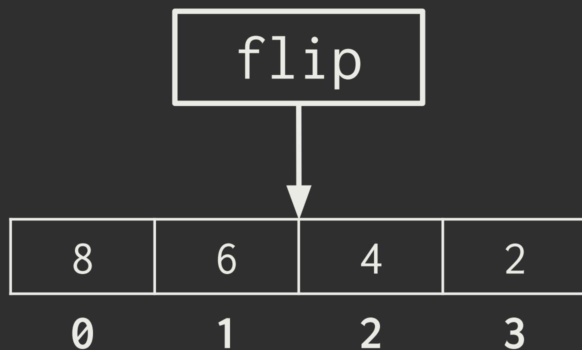
Flipping an Array



How many elements do we flip?

Is this number relative to the size of the array?

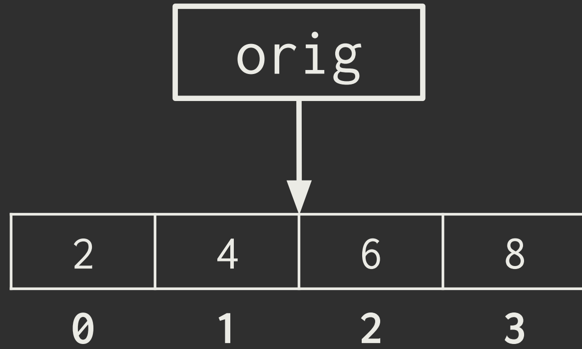
Flipping an Array

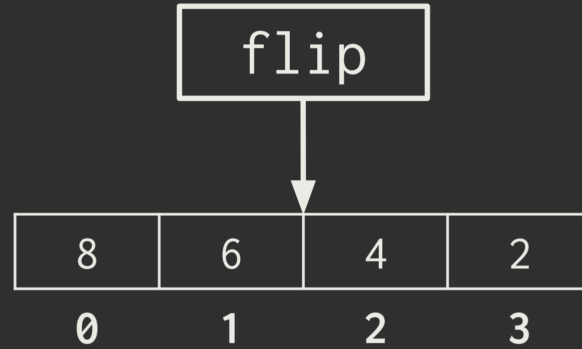


How many elements do we flip?

Is this number relative to the size of the array?

Flipping an Array





How many elements do we flip?

Swap each element at the front with one at the back.

Is this number relative to the size of the array?

Only need to perform `orig.length` swaps!

Flipping an int Array

```
private void flip(int[] toFlip) {
    for (int i = 0; i < toFlip.length / 2; i++) {
        int tmp = toFlip[i];
        toFlip[i] = toFlip[toFlip.length - i - 1];
        toFlip[toFlip.length - i - 1] = tmp;
    }
}
```

How would we change this to work with types other than `int`?

How might we flip a `GImage`? (Think about this...)

2D Arrays! (Grids)

What if our `Type` was an array? (e.g. `int[]`)

```
Type[] myGrid = new
Type[numElems];
```

```
Type[] myGrid = new
Type[numElems];
```

Let's try setting `Type` to `int[]`

```
int[][] myGrid = new
int[numArrays][numElems];
```

```
int[][] myGrid = new
int[numArrays][numElems];
```

Why does this work?

Because each `int[]` is an object!

```
int[][] myGrid = new
int[numArrays][numElems];
```

We make an array of integer arrays!

(Why we say the array spans two dimensions)

```
int[][] myGrid = new
int[numArrays][numElems];
```

Note: we specify the size of each dimension

(`numArrays` and `numElems`)

2D Arrays (or Grid)

```
Type[][] myGrid = new Type[rows][cols];
```

We say the grid is “row-major”

- First dimension is rows
- Second dimension is columns

Each row of the grid is an array

Each column is defined by an index in each row array

Each element is located in a specific column in a specific row

Interpreting Multidimensional Arrays

As a 2D Grid

Looking up `arr[row][col]` selects the element in the grid at position `(row, col)`

Remember that the bracket order matters!
The first set of brackets is row, the second is column

As an Array of Arrays

Looking up `arr[n]` `gives back a one-dimensional array representing the` `n+1`-th row
Remember that expression is a reference to that array!

First dimension indexes into different arrays, the second indexes elements in a given array

2D Arrays Examples

```
int[][] multiArr = new int[4][5];
```

What type do these expressions evaluate to? (What is returned?)

`multiArr[1]`?

`int[5]` -> a reference to an array of five integers

`multiArr[3][4]`?

`int` -> a single integer value

Changing 2D Arrays with References

```
public void run() {
    int[][] full = new int[3][3];
    int[] row = full[1];
    row[1] = 4;
    int[] newRow = new int[3];
    newRow[0] = 6;
    full[2] = newRow;
}
```

What does this do?

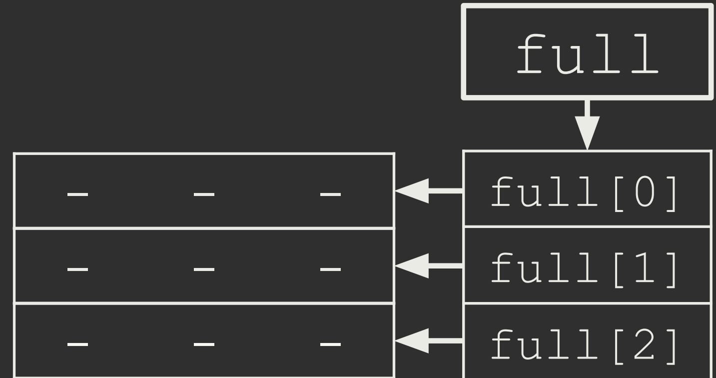
Changing 2D Arrays with References

```
public void run() {
    int[][] full = new int[3][3];
    int[] row = full[1];
    row[1] = 4;
    int[] newRow = new int[3];
    newRow[0] = 6;
    full[2] = newRow;
}
```

What does this do?

Changing 2D Arrays with References

```
public void run() {
    int[][] full = new int[3][3];
    int[] row = full[1];
    row[1] = 4;
    int[] newRow = new int[3];
    newRow[0] = 6;
    full[2] = newRow;
}
```

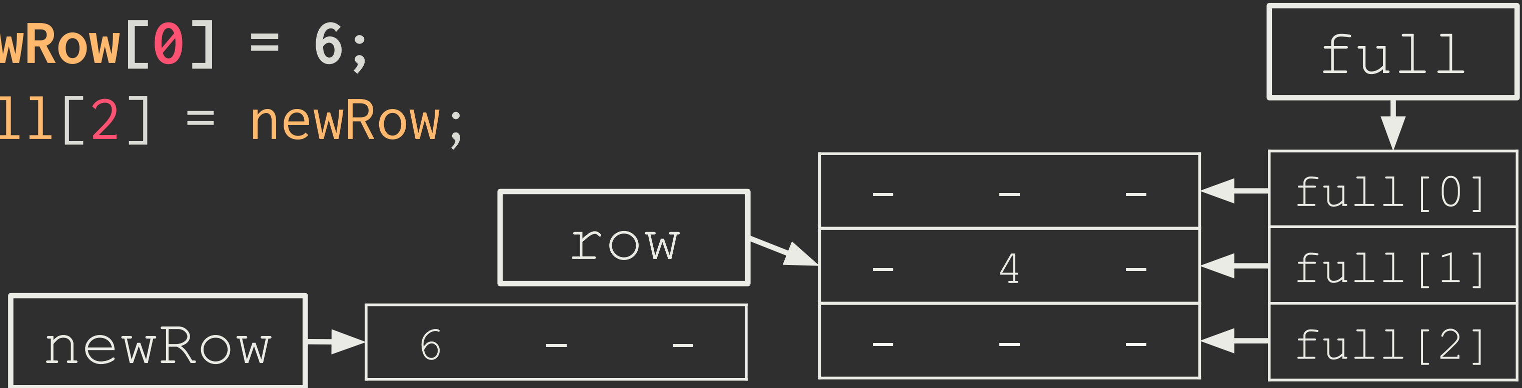


Changing 2D Arrays with References

```
public void run() {
    int[][] full = new int[3][3];
    int[] row = full[1];
    row[1] = 4;
    int[] newRow = new int[3];
    newRow[0] = 6;
    full[2] = newRow;
}
```

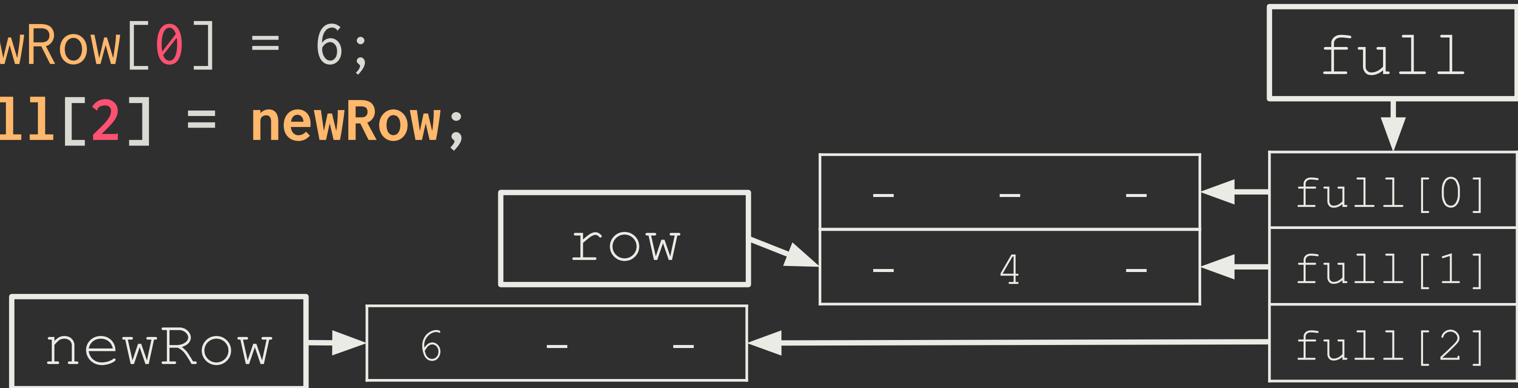
Changing 2D Arrays with References

```
public void run() {
    int[][] full = new int[3][3];
    int[] row = full[1];
    row[1] = 4;
    int[] newRow = new int[3];
    newRow[0] = 6;
    full[2] = newRow;
}
```



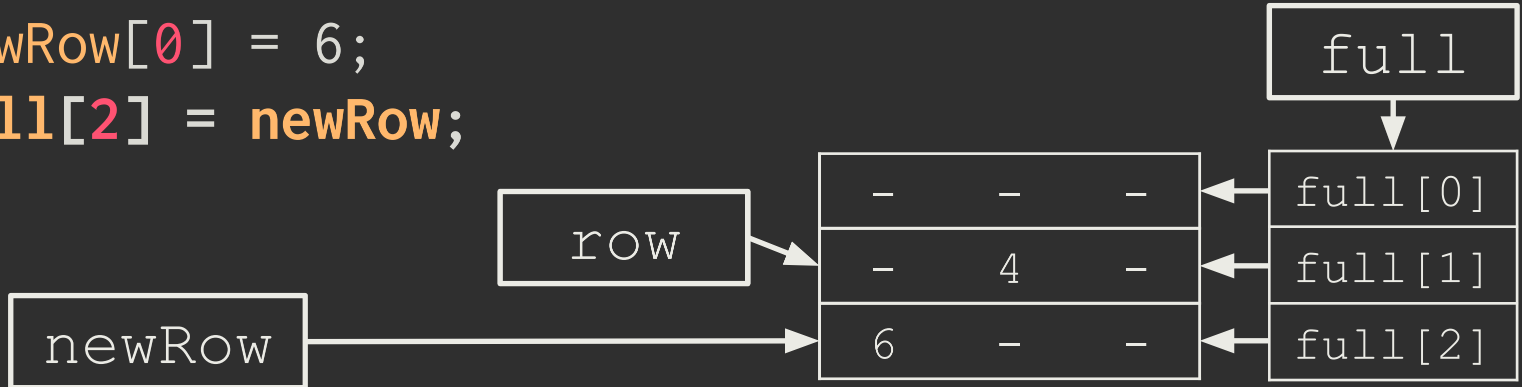
Changing 2D Arrays with References

```
public void run() {
    int[][] full = new int[3][3];
    int[] row = full[1];
    row[1] = 4;
    int[] newRow = new int[3];
    newRow[0] = 6;
    full[2] = newRow;
}
```

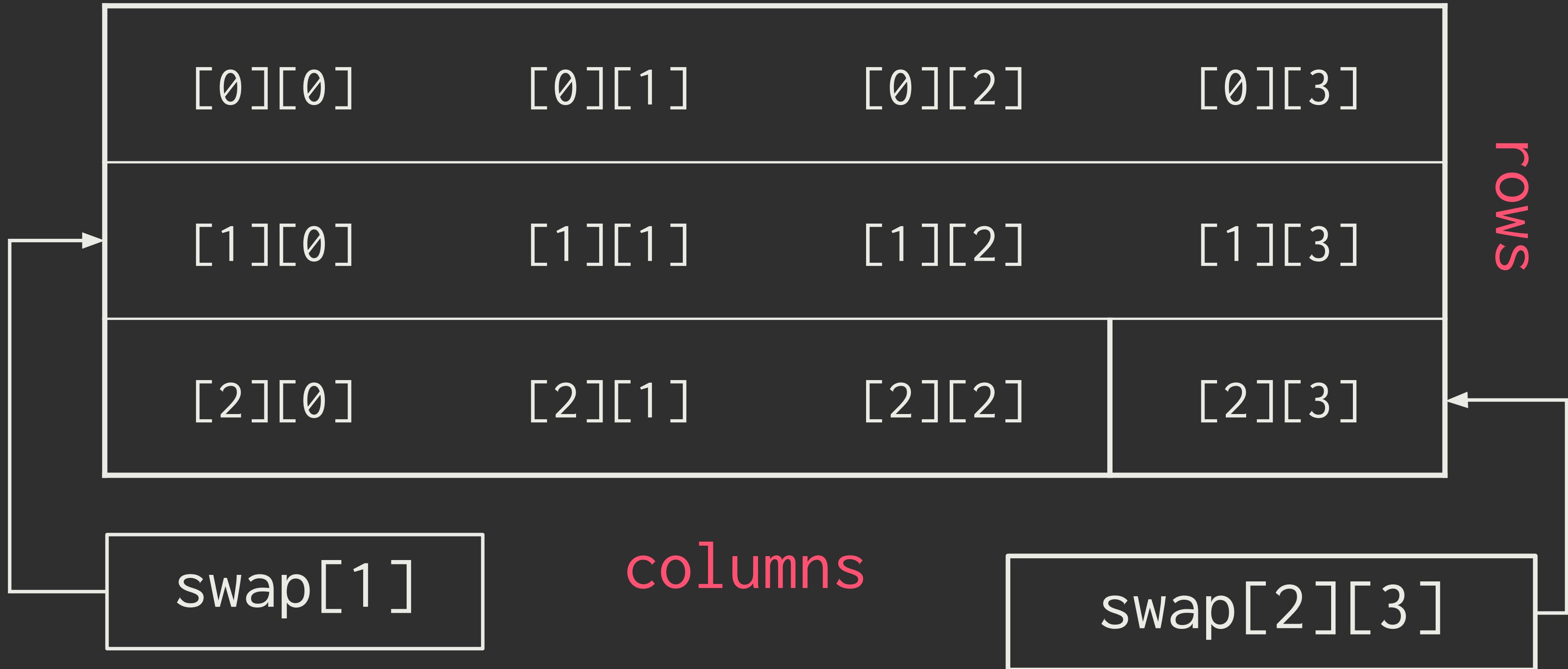


Changing 2D Arrays with References

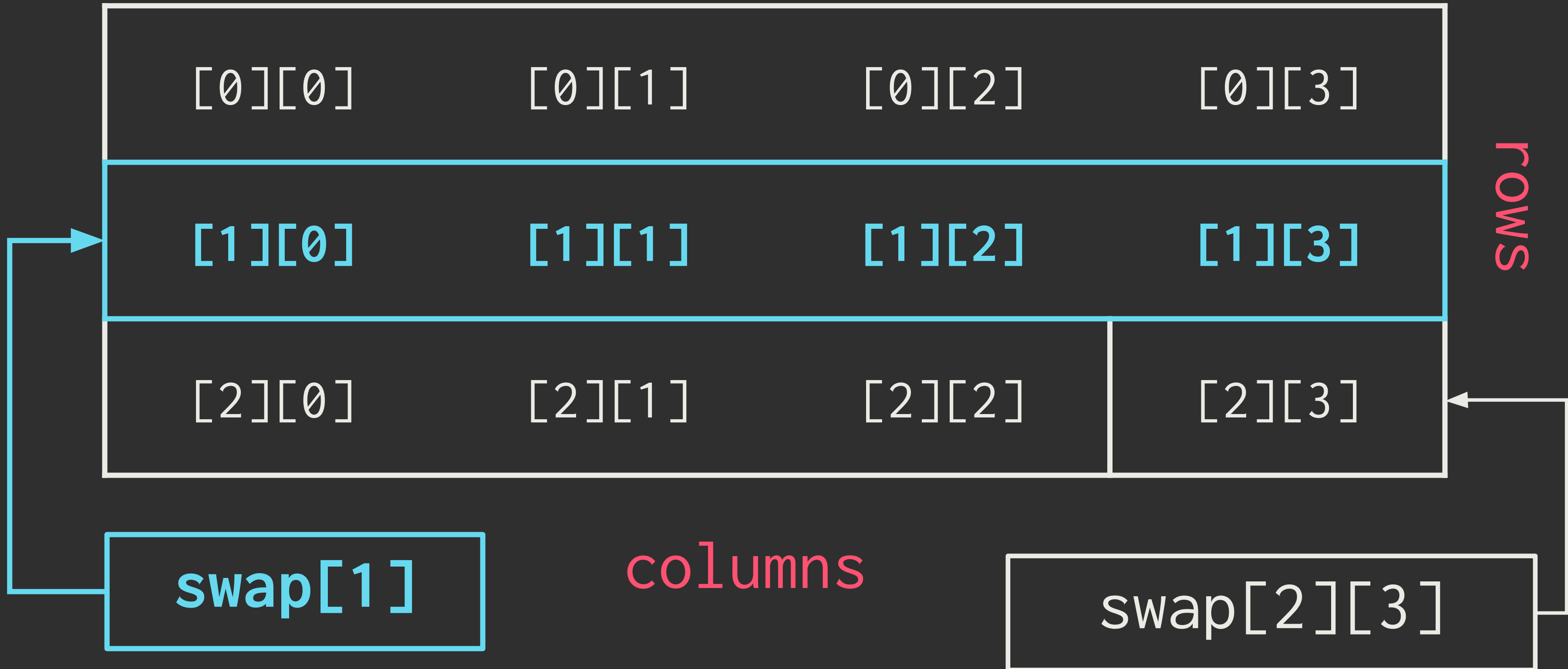
```
public void run() {
    int[][] full = new int[3][3];
    int[] row = full[1];
    row[1] = 4;
    int[] newRow = new int[3];
    newRow[0] = 6;
    full[2] = newRow;
}
```



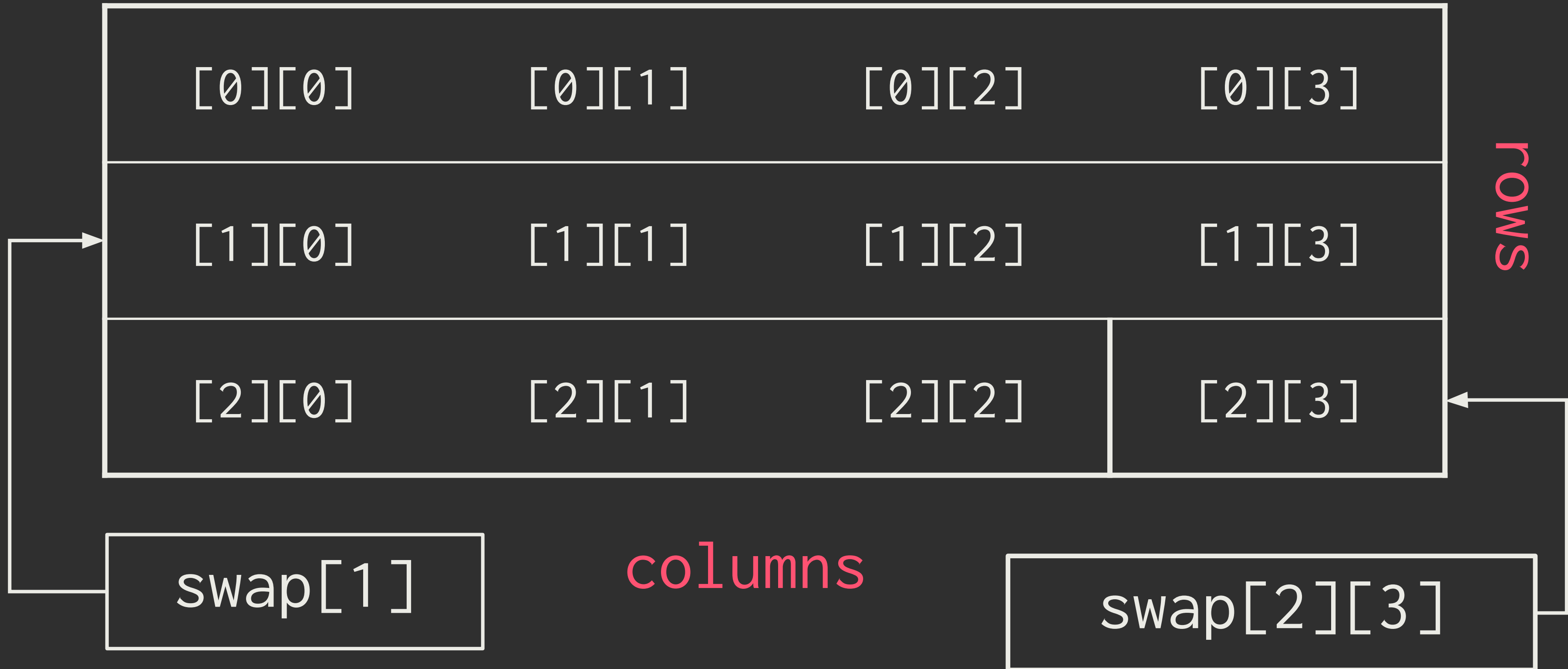
Dimension Swapping — `int[][] swap = new int[3][4];`



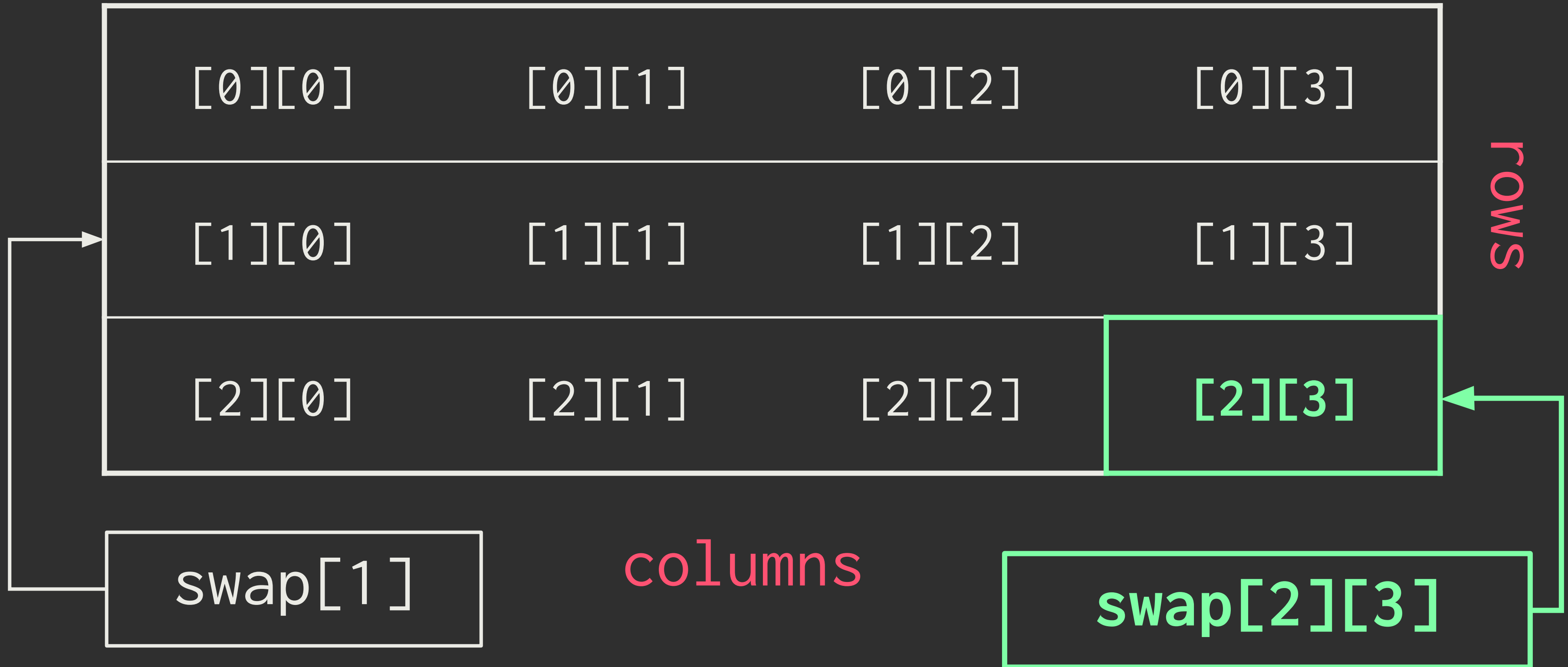
Dimension Swapping – `int[][] swap = new int[3][4];`



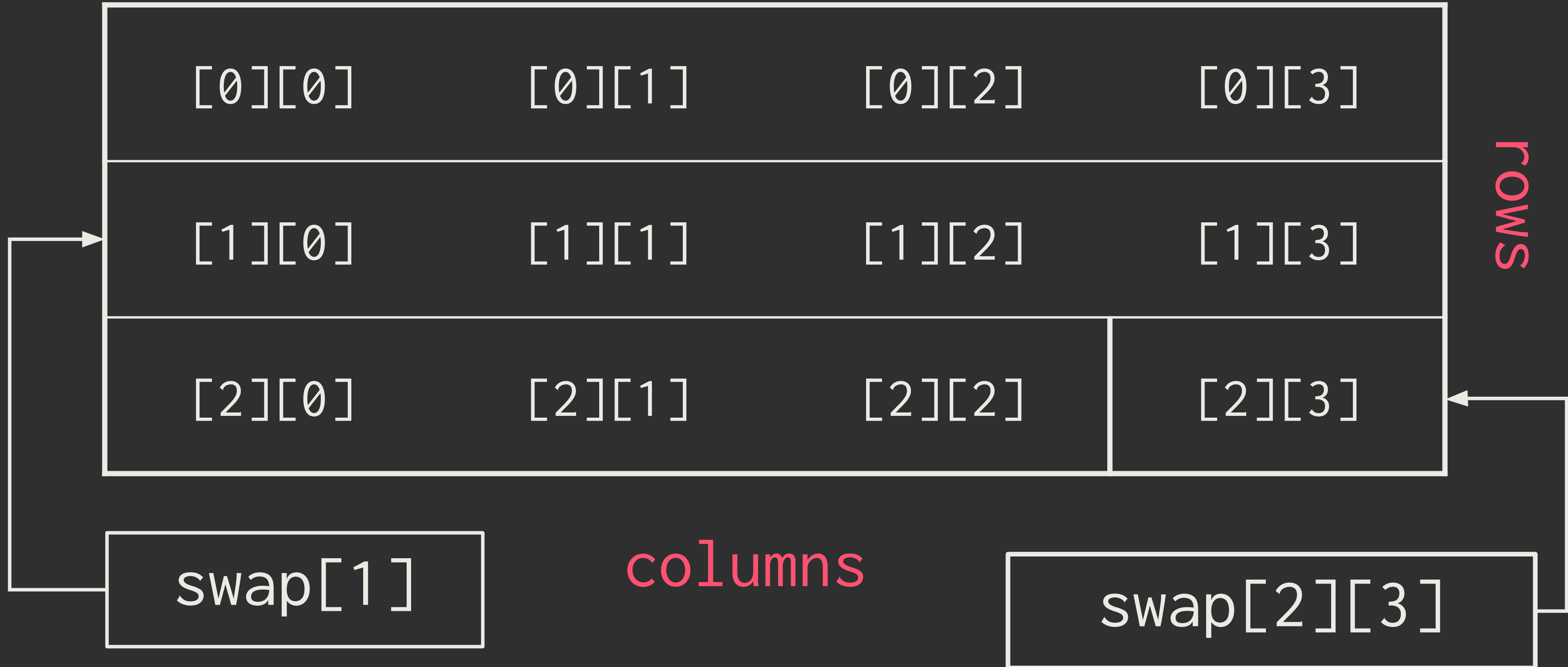
Dimension Swapping – `int[][] swap = new int[3][4];`



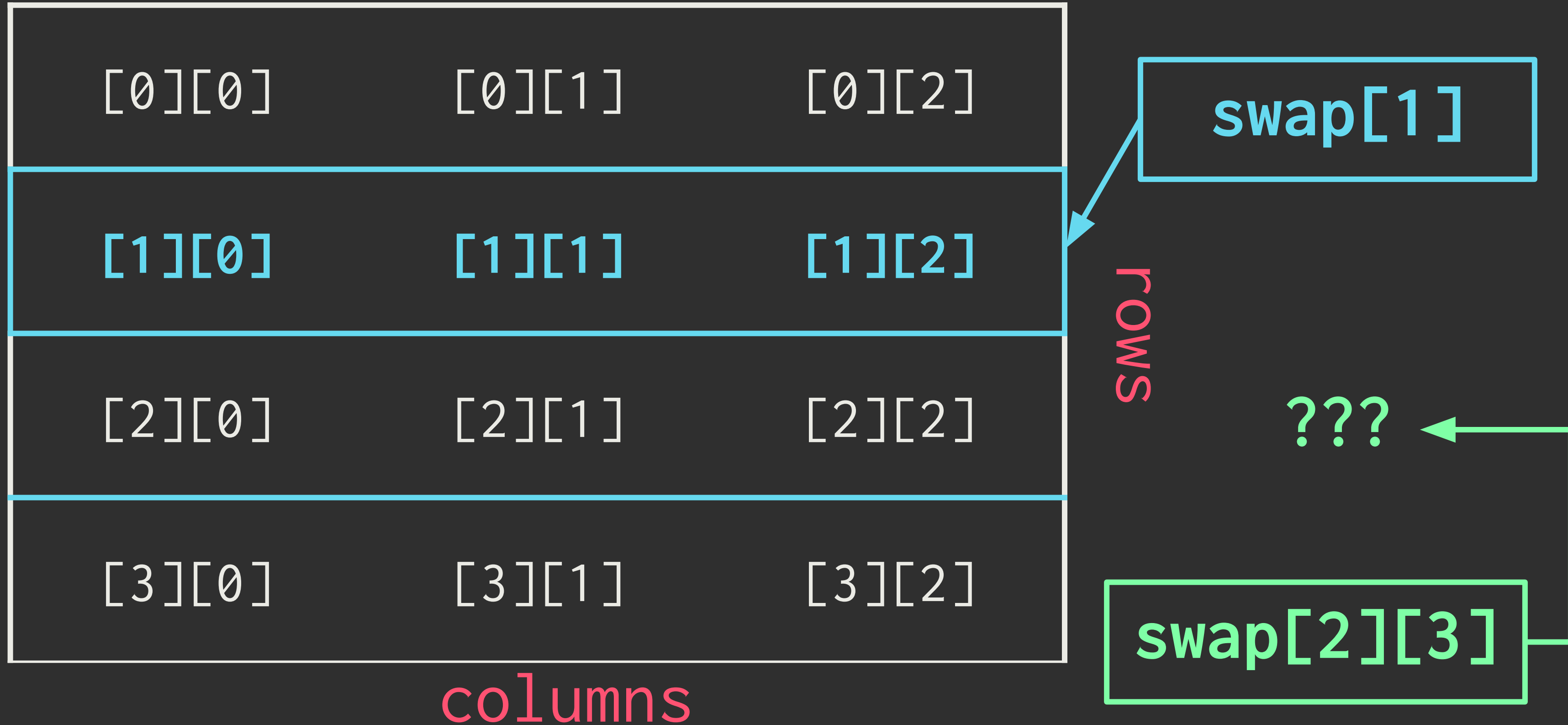
Dimension Swapping – `int[][] swap = new int[3][4];`



Dimension Swapping – `int[][] swap = new int[3][4];`



Dimension Swapping – int[][] swap2 = new int[4][3];



Dimension Swapping

Swapping the order of dimension sizes has real consequences on how data is stored!

Remember that 2D arrays always follow

```
int[][] myGrid = new int[numArrays][numElems];
```

This doesn't change the total number of elements, but flips number of rows with number of columns

Often times we'll iterate through an entire grid and won't see struggle with this directly

Iterating Through 2D Arrays

```
int[][] arr = /* ... */

/* arr.length is numArrays from declaration */
for (int row = 0; row < arr.length; row++) {

    /* arr[row] is an array of size numElems */
    for (int col = 0; col < arr[row].length; col++) {
        /* access arr[row][col] */
    }
}
```

Iterating Through 2D Arrays Example 1

```
double[][] foo = new double[2][3];

for (int row = 0; row < foo.length; row++) {
    for (int col = 0; col < foo[row].length; col++) {
        arr[row][col] = col / (double) (row + 1);
    }
}
```

0.0	1.0	2.0
0.0	0.5	1.0

Iterating Through 2D Arrays Example 2

```
int[][] bar = new int[2][3];
for (int row = 0; row < bar.length; row++) {
    for (int col = 0; col < bar[row].length; col++) {
        if (row == col) {
            arr[row][col] = 1;
        } else {
            arr[row][col] = 0;
        }
    }
}
```

1	0	0
0	1	0

2D Array Summary

There are two main ways to think about a 2D Array

```
Type[][] myGrid = new Type[rows][cols];
```

```
Type[][] myArr = new Type[numArr][numElem];
```

When using a 2D Array

- First dimension represents rows and indexing evaluates to an array
- Second dimension represents columns and indexing evaluates to a value
- Must index into rows *before* columns

GImages and Pixels

How do manipulate images?

(Based on slides by Gaby Candes)

What's a `GImage`?

`GImages` are grids (or 2D Arrays!) of pixels

```
GImage img = new GImage("watermelon.jpg");
```

We represent pixels as integers, and can convert `GImage` to `int[][]`

```
int[][] pixels = img.getPixelArray();
```

ProTip: Make Helper Methods!

When using a pixel array from a `GImages`, the image height is the array `numRows` and the image width is the array `numCols`

```
int[][] pixelArr =
img.getPixelArray(); //int[numRows][numCols
```

Use helper methods to keep track of your image height/width!

```
private int numRows(int[][] pixels) {
    return pixels.length; // height
}
```

```
private int numCols(int[][] pixels) {
    return pixels[0].length; // width
}
```

What’s a pixel? Why is it an int?

Each pixel is an integer whose value represents the red, green, blue (RGB) intensity at that location.

We use a scale of `0-255` for the intensity of each color component

```
// get the R, G, B values for a particular pixel (type int)
int red = GImage.getRed(pixel);
int green = GImage.getGreen(pixel);
int blue = GImage.getBlue(pixel);

// make a new pixel (type int) from certain R, G, B values
int newPixel = GImage.createRGBPixel(red, green, blue);
```

Iterating Through Pixel Array

```
// get the dimensions of the array
int imgRows = numRows(pixels);
int imgCols = numCols(pixels);

// iterate over pixel array
for (int r = 0; r < imgRows; r++) {
    for (int c = 0; c < imgCols; c++) {
        // get a specific pixel from your image
        int newPixel = pixels[r][c];
        // do something to newPixel, like make it cardinal
        pixels[r][c] = GImage.createRGBPixel(196, 30, 58);
    }
}
```

Making a New GImage

```
// 1. make (or get) a 2D array
int[][] pixels = /* ... */

// 2. fill (or modify) the array

// 3. make a GImage from 2D array
GImage newImage = new GImage(pixels);
```

DarkRoom Overview

Assignment 5

Big **Picture**

`DarkRoom.java` and `DarkRoomAlgorithmsInterface.java` are written for you and provided with the starter code

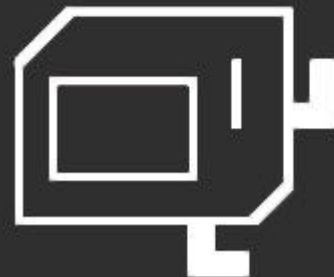
- Together, these files handle user interaction (making buttons, handling clicks, loading images, etc)
- You aren't responsible for understanding the code in these

You need to write `DarkRoomAlgorithms.java`

- Consists of writing different methods that take in `GImage` parameters and return modified `GImages`

What methods do you have to do?

Rotations!



Rotations!

	0	1	2	3	4	5
0	A	B	C	D	E	F
1	G	H	I	J	K	L
2	M	N	O	P	Q	R
3	S	T	U	V	W	X

orig

	0	1	2	3
0	F	L	R	X
1	E	K	Q	W
2	D	J	P	V
3	C	I	O	U
4	B	H	N	T
5	A	G	M	S

rotated

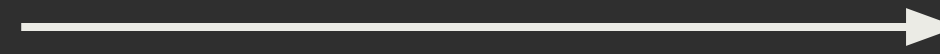
What are the dimensions of `orig`? What are the dimensions of `rotated`?

What are the coordinates of Q in `orig`? What are they in `rotated`?

Rotations!

	0	1	2	3	4	5
0	A	B	C	D	E	F
1	G	H	I	J	K	L
2	M	N	O	P	Q	R
3	S	T	U	V	W	X

`orig`



	0	1	2	3
0	F	L	R	X
1	E	K	Q	W
2	D	J	P	V
3	C	I	O	U
4	B	H	N	T
5	A	G	M	S

`rotated`

What are the dimensions of `orig`? What are the dimensions of `rotated`?

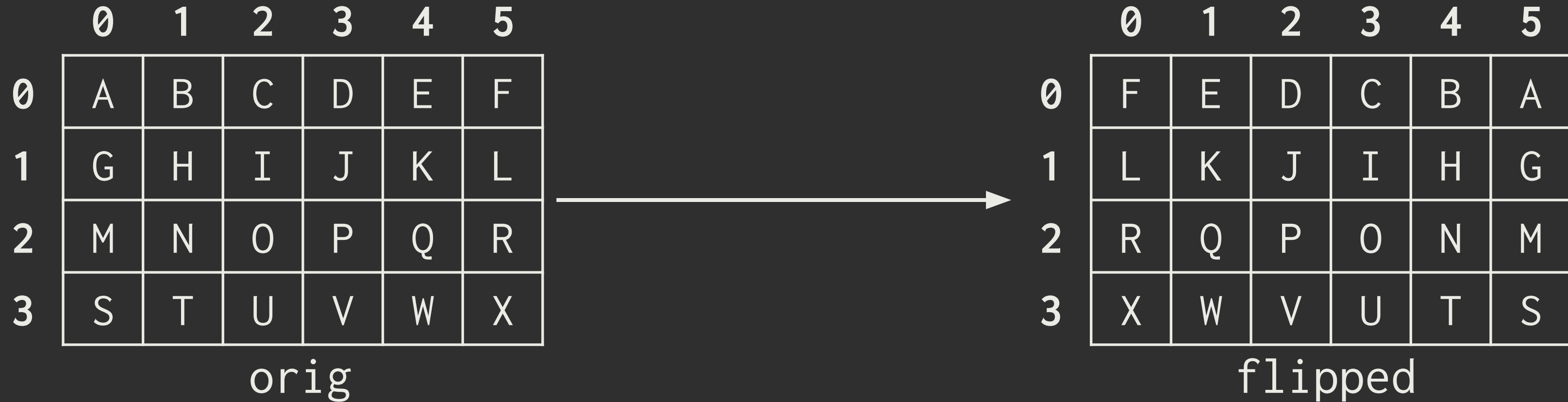
`orig` has 4 rows and 6 columns, `rotated` has 6 rows and 4 columns

What are the coordinates of Q in `orig`? What are they in `rotated`?

`orig`: row 2, column 4, `rotated`: row 1, column 2

Flipping!

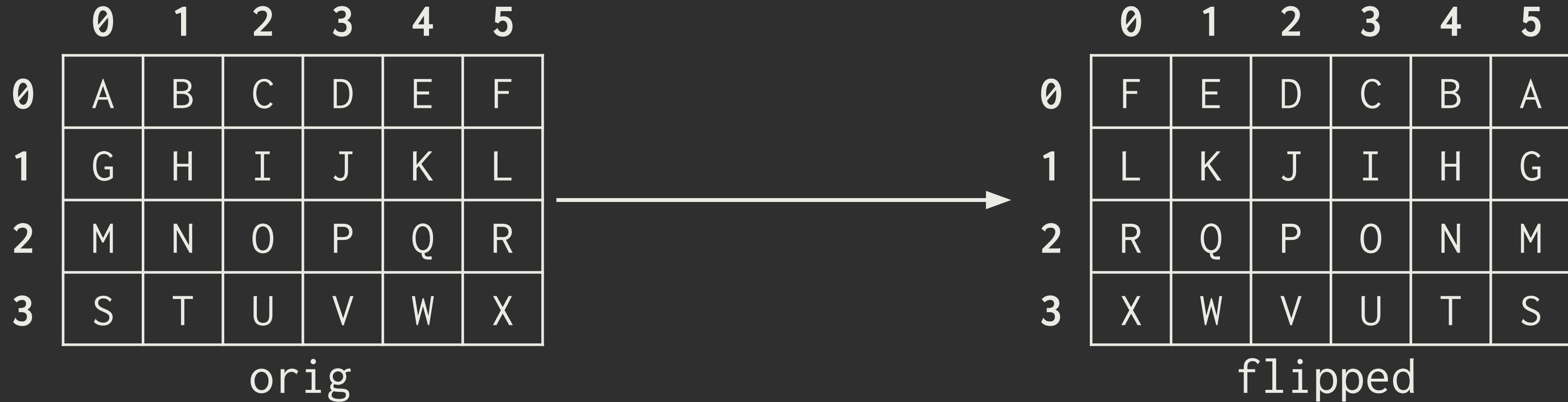
Flipping!



What are the dimensions of `orig`? What are the dimensions of `flipped`?

What are the coordinates of Q in `orig`? What are they in `flipped`?

Flipping!



What are the dimensions of `orig`? What are the dimensions of `flipped`?

`orig` has 4 rows and 6 columns, `flipped` has 4 rows and 6 columns

What are the coordinates of Q in `orig`? What are they in `flipped`?

`orig`: row 2, column 4, `flipped`: row 2, column 1

Check out `flipVertical` from Section 6!

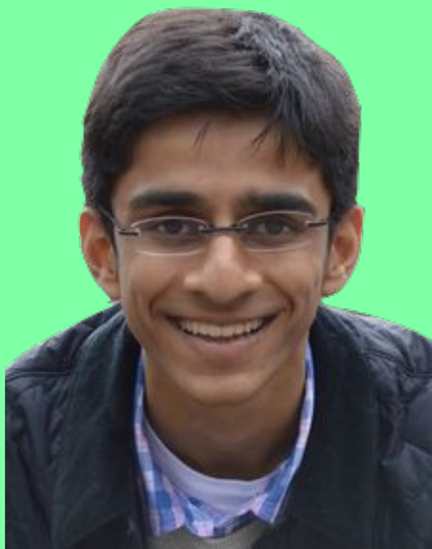
Negative!

The red, blue and green values of every pixel should be set to 255 - k where k represents each of these values (make 255 a constant!)

Remember we can use:

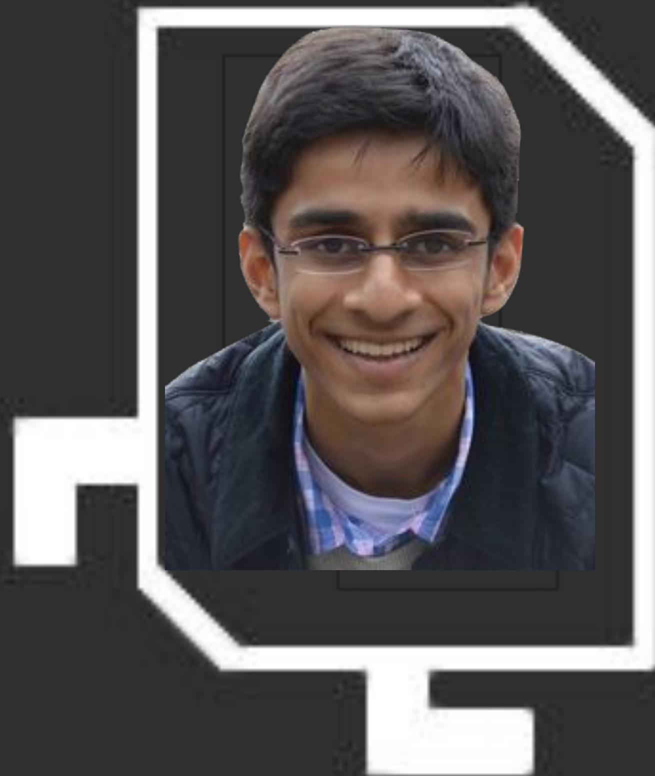
```
int red = GImage.getRed(pixel);
//Same for blue and green
//Write more cool code
int newPixel = GImage.createRGBPixel(newRed, newGreen, newBlue);
```

Green Screen



Green Screen

Super Duper Karel?



Capoorel?

Green Screen

Make any pixel whose green component is twice as big as the larger of its red or blue components transparent. *Hint: Try using* `Math.max(a,b)`*!*

Opacity is called alpha in CS terms. An alpha of 255 is completely opaque, an alpha of 0 is completely transparent. We can make a pixel with a particular alpha value using

```
int pixel = GImage.createRGBPixel(r, g, b, alpha);
```

So for transparency:

```
int transparentPixel = GImage.createRGBPixel(42, 88, 19, 0);
```

Blur

Set a pixel's RGB values to the average of the pixel and all of its neighbors' RGB values!

Things to consider:

How do you get all the neighbors of a pixel?

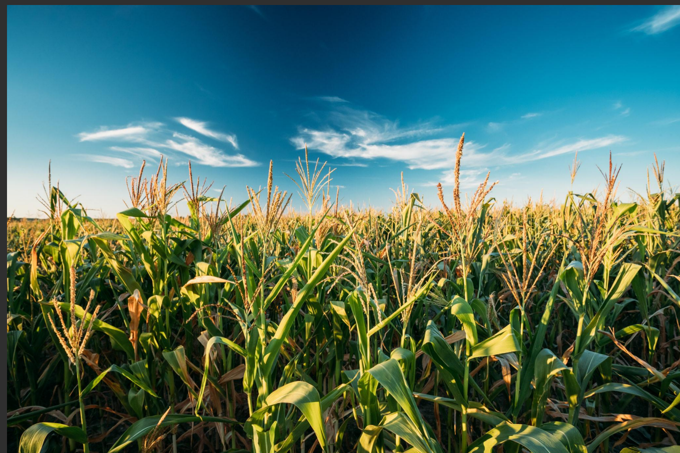
Don't write out nine lines of code! Think for loops...

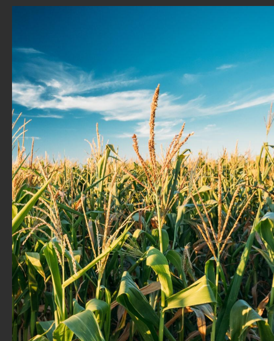
How do you account for the edges?

Sum over fewer values (make sure to adjust the denominator)

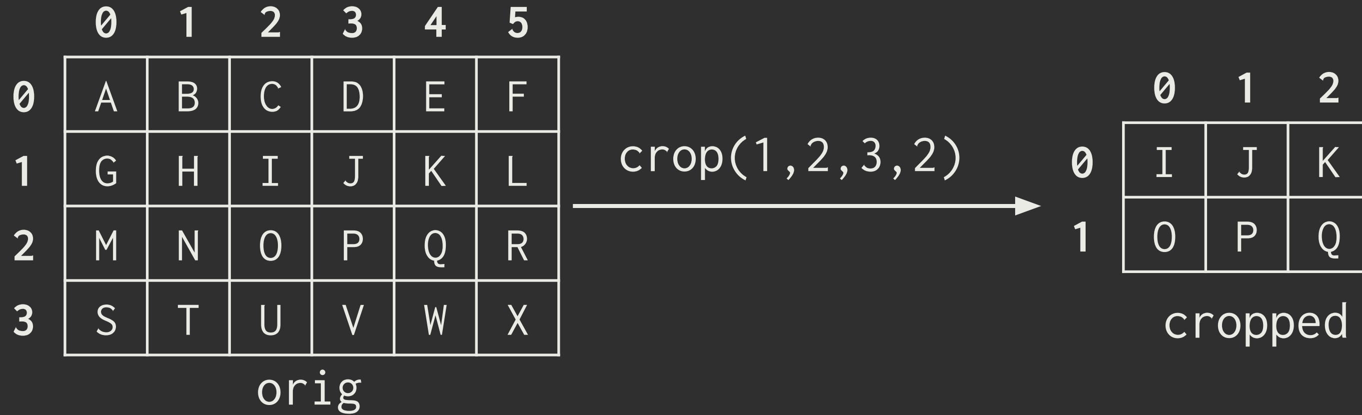
(r-1,c-1)	(r-1,c)	(r-1,c+1)
(r,c-1)	(r,c)	(r,c+1)
(r+1,c-1)	(r+1,c)	(r+1,c+1)

Cropping!





Cropping!



`crop(image,x,y,width,height)`

What are the dimensions of `orig`? What are the dimensions of `cropped`?

`orig` has 4 rows and 6 columns, `cropped` has 2 rows and 3 columns

equalize

We are trying to spread out the luminosity of an image (how bright it is). We can get the luminosity (brightness) of a pixel with

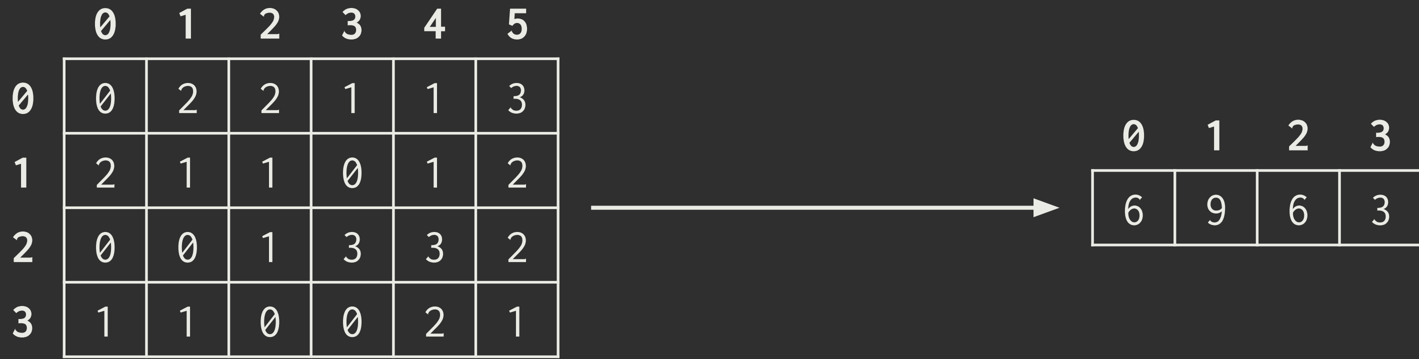
```
int luminosity = computeLuminosity(r, g, b);
```

We are going to be working with grayscale images meaning their r, g, and b values will all be set to the same number! (0, 0, 0) is black, (255, 255, 255) is white, and (x, x, x) for some other number x will be a shade of gray.

Three parts (that should be decomposed!):

- Compute the luminosity histogram
- Compute a *cumulative* luminosity histogram
- Modify each pixel based on cumulative luminosity

Compute the Luminosity Histogram



The histogram represents the number of times we’ve seen each pixel.
What’s the best way to represent this in our code?

Compute the Cumulative Luminosity Histogram

Given the original histogram, how can we generate these cumulative values?

Generate the New Image

	0	1	2	3
	6	15	21	24

	0	1	2	3	4	5
0	63	223	223	159	159	255
1	223	159	159	63	159	223
2	63	63	159	255	255	223
3	159	159	63	63	223	159

If the pixel at (r, c) has luminosity L:

New RGB at (r,c) = 255 * (# of pixels with luminosity <= L) / # of pixels in image

Why did we make you compute the cumulative histogram?

Tips and Tricks

Questions?

Sample Title

Sample body

Style Standards

Type something #FF5272

Type something #BD93F9

Type something #66D9EF

Type something #FFFFFF

Text font - Overpass Normal

`Code font - Inconsolata`

Type something #7FFFA6

Type something #FFB86C

Type something #DADAD4

Type something #2f2f2f

I’m a header

I’m a subheader